883II - Group 28

Large-Scale

Francesco Fattori, Nuno, Steffen

Fall 2021

INTRODUCTION

BookWorm provides a service whose main functionalities are based on collecting, organizing, and presenting information of a dataset of books. Each book is stored as an element in a database with the following properties: ISBN, book name, name of author, book format, description, genre, link to book on *Goodreads*, number of pages, average rating and image. Users will be able to perform certain functions such as searching for a book, searching for other users, follow/unfollow users, and rate books.

There will also be more advanced queries that the user will have access to. This includes a list of suggested books based on algorithms that take into account the different users that the user follows, and a function that returns a list of suggested users the user could follow.

DESIGN STAGE

Functional Requirements

A list of all the functional requirements of the applications follows bellow:

- The users will be divided into three different levels of accessibility: *Administrator*, which can access all functions within the application, *Registered User*, which will only have access to the simple functions within the application, and *Unregistered User* which will be able to login and register new user.
- To access the full application features, an **Unregistered User** will have to enter a *username* and a *password* through a login system that is displayed upon entering *BookWorm*. There will also be an option to fill in a sign-up form can be filled out in order to register as a new *Registered User*.
- The *Administrator* should be able to:
    - Ban *User* from the Application
    - Edit a *Book* from the database
    - Add a *Book* to the database

  - Browse for *Books* and *Users*

- The *Registered User* should be able to:
  - Search of *Book* based on title
  - Search for a *User* based on username mask
  - Follow/unfollow *User*
  - *Rate* a book between a scale from 1 to 5
  - See the average *Rating* of a *Books*
  - Request a list of suggested *Books*
  - Request a list of suggested *Users*
  - View highest rated books based on average rating from other users

Non-Functional Requirements

A list of all the non-functional requirements of the applications follows bellow:

- The Application should have a good GUI for the User
- The Application must handle multiple users logged at the same time from different computers.

UML Class Diagram

There are two main classes: Book and User. The rating class will also be available but will be viewed by the system mainly inside a User object and so it is a secondary class. The user class divides in to 3 types, Admin, User and Banned. Depending on which type of user you are you will have access to more or less functionalities in the program.

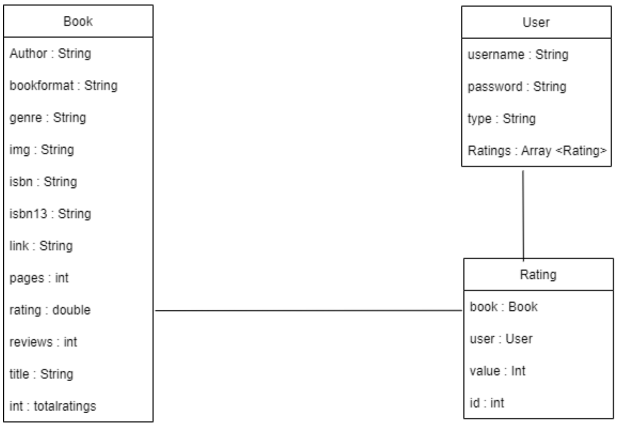


Figure 1: Class Diagram of our program

Hypothetical Work

A few assumptions have been made without implementing them into the applications. This includes the handling of a *User*'s data privacy, which is not taken into account during application development.

Also it was considered to develop a sharding strategy but because it seemed to do not a great improvement and other parts of the project appeared more important to develop with the time we had we decided against it.

Also, the book database is from a different website so the ratings are considered users from that website. However, we were not able to import the user dataset, so the number of ratings of our users do not correspond to the ratings count of the books. This means we can still rate a book which will impact its average rating and number of ratings but we have a number of ”ghost ratings” that are counted in the books but not in the users.

Although it would be possible to reset ratings on every book this would make the statistics and searches on those books useless so we built our ratings on top of the already available numbers.

SPECIFICATIONS

Actors and Use Cases Diagram

The *BookWorm* application will be used by three different types of actors: *Administrator*, *User*, *Anonymous User*. Bellow is a description of the different actors:

- *Anonymous User* - is allowed to log in or register through the *sign up* button. This is done through a *username/password* system. As Anonymous User (or Guest) is also allowed to interact with most of the operations within the applications such as search for *User* and *Book*, and view details about *User* and *Book*.
- *Registered User* - is allowed to interact with most of the operations within the applications such as search for *User* and *Book*, and view details about *User* and *Book*, browse a list of featured *Users* and *Books*, *Rate* different *Books*, *Follow* and unfollow specific *Users*.
- *Administrator* - is allowed to interact with all operations that Anonymous User can, but also includes the operations of highest privilege: ban *User*, adding new *Book* to dataset, editing a *Book*, get the most active *Users*.

The aforementioned actors with their use case are displayed in figure 2.

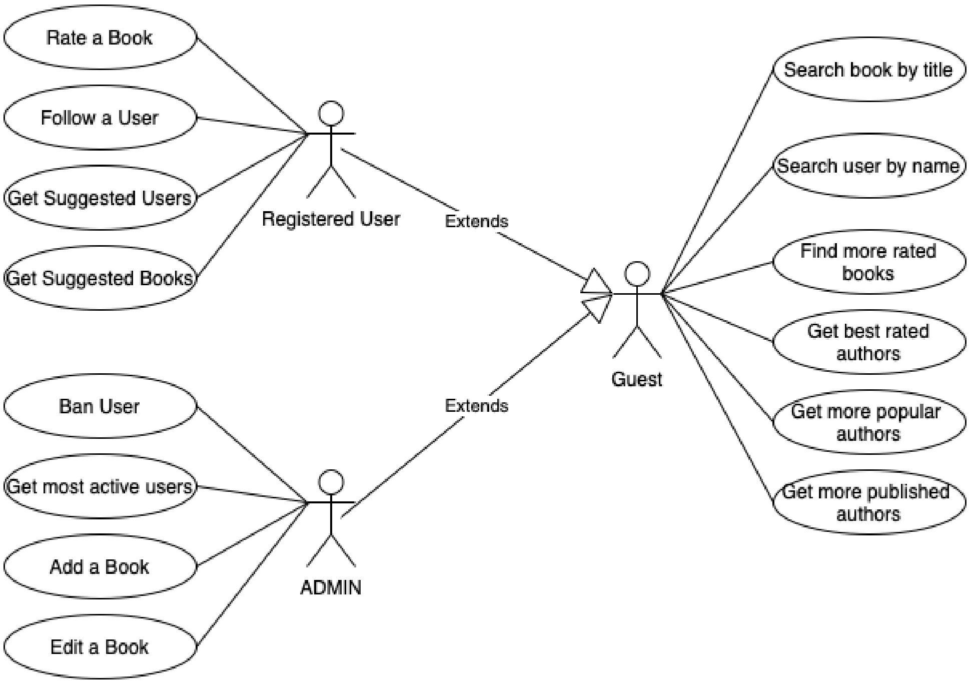


Figure 2: Use case Diagram

Classes and Definitions

The main classes of our project are:

- **Book** - contains all the information about the Book entity
- **User** - contains all the information about the User entity
- **Rating** - contains the value of the rating and the title of the book rated
- **Role** - enum with different role a user can have

Classes and Attributes

	Class	Description
Book	**String** author	The name of the book author
	String bookformat	The format in which the book is sold
	String desc	A short description of the book
	String genre	List of genre of the book
	String img	URL to the cover picture
	String isbn	ISBN code
	String isbn13	ISBN13 code
	String link	Link to goodreads
	Int pages	Number of pages in the book
	Double rating	Average rating of the book
	String reviews	Number of written reviews in GoodReads
	String title	Title of the book
	String totalratings	Number of ratings of the book

	Class	Description
User	**String** username	Username of the user
	String password	Password of the user
	String type	Type (or Role) of the user
	Array of Rating ratings	List of genre of the book

	Class	Description
Rating	**String** book	Title of rated book
	Double value	How many star the user gave

	Class	Description
Role	**ENUM** Role	{ GUEST, ADMIN, USER, BANNED }

ARCHITECTURAL DESIGN

In terms of architecture the Application has been built with Java and JavaFX library (for the Graphical User Interface) with a MVC like paradigma. The application has been developed and compiled with OpenJDK 17.0.1.

Java Libraries needed:

- **openJDK 17** - core Java development library
- **JavaFX** - library for the GUI
- **mongodb-driver-*** - various library for the mongodb driver
- **neo4j-java-driver** - library for neo4j driver
- **reactivestreams** - library for asynchronous tasks

For the databases we used MongoDB and Neo4j. While we used Neo4j in local on our machine (powered by Docker), we wanted to deploy our MongoDB on a real server with replicas. For this we used the free tier of MongoDB Atlas (https://www.mongodb.com/atlas/). In this way we had our MongoDB up and running on the cloud with a 3 Replica Set. The Application will connect to this set with the write policy of W:1.

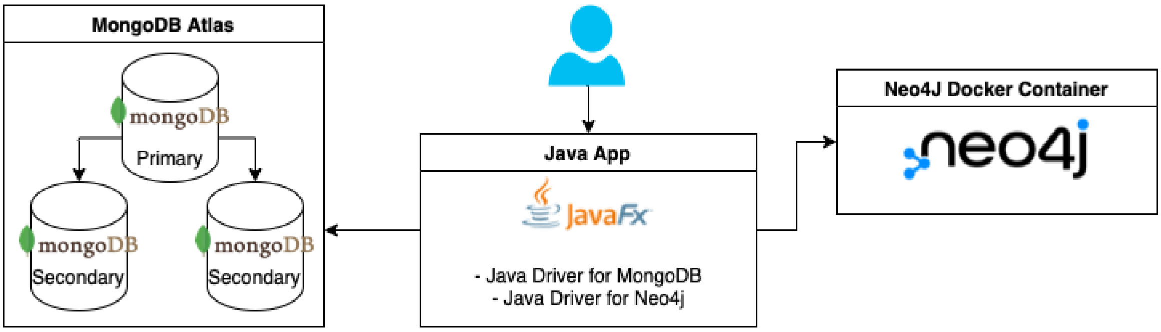
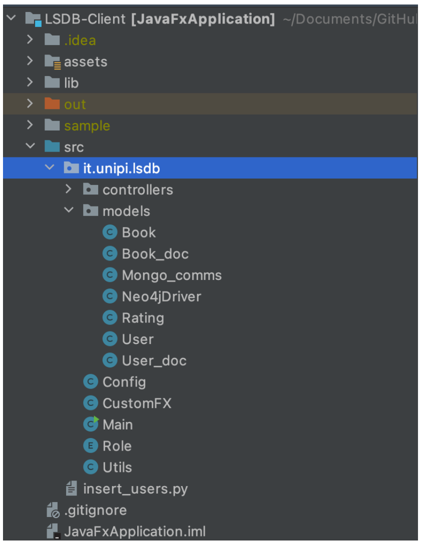


Figure 3: Architecture

The main package of the application is "it.unipi.lsdb". Inside we have:

- **Config** - static class with runtime variables
- **CustomFX** - code for some custom list in JavaFX
- **Main** - Main class to start db driver and GUI
- **Role** - enum class with user roles
- **Utils** - static class with various help functions
- **controllers** - package with GUI controllers
- **models** - package with entity from our dbs and query classes (ex. Book_doc and User_doc)



```
LSDB-Client [JavaFxApplication] ~/Documents/GitHu
  .idea
  assets
  lib
  out
  sample
  src
    it.unipi.lsdb
      controllers
      models
        Book
        Book_doc
        Mongo_comms
        Neo4jDriver
        Rating
        User
        User_doc
      Config
      CustomFX
      Main
      Role
      Utils
    insert_users.py
  .gitignore
  JavaFxApplication.iml
```

Figure 4: Code Structure

MONGODB DESIGN AND IMPLEMENTATION

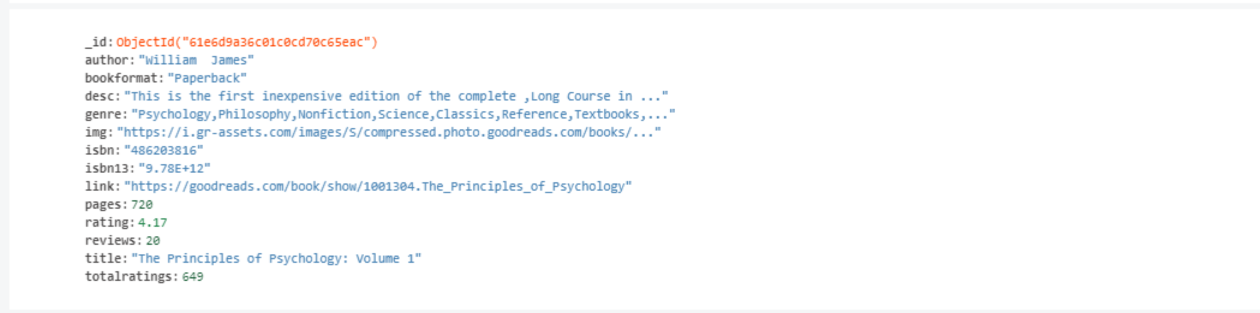
MongoDB is a type of document database. In this project it will be used to store most of the information, it will store all the information on the books as well as the Users and the Ratings.

Architecture

The Database has two collections, to maximize efficiency as there are two main types of objects, Books and Users, and so when we search for one we do not need to search for another. The ratings will be embedded as an Array on every User Document and will have a reference to a book by name as well as the value of the rating given by the User.

This could of been done in two other ways: it could be embedded in the book ,but that could cause problems because just one book can have thousands of reviews while with users is more distributed;it could be on a different collection, but it was not desirable to get more collections as every time we want to see a user's ratings we would have to scan all the collection.

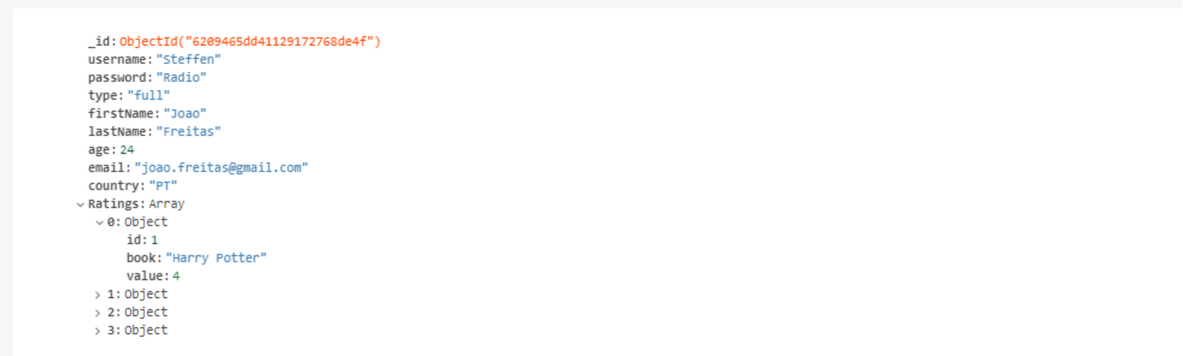
This way, when a user logs in he/she already has all the necessary information.



```
_id: ObjectId("61e6d9a36c01c0cd70c65eac")
author: "William  James"
bookformat: "Paperback"
desc: "This is the first inexpensive edition of the complete ,Long Course in ..."
genre: "Psychology,Philosophy,Nonfiction,Science,Classics,Reference,Textbooks,..."
img: "https://i.gr-assets.com/images/S/compressed.photo.goodreads.com/books/..."
isbn: "486203816"
isbn13: "9.78E+12"
link: "https://goodreads.com/book/show/1001304.The_Principles_of_Psychology"
pages: 720
rating: 4.17
reviews: 20
title: "The Principles of Psychology: Volume 1"
totalratings: 649
```

Figure 5: Example of a document of a Book

In the User Document we can see an array of Ratings and the contents of one of the objects of set array.



```
  _id: ObjectId("6209465dd41129172768de4f")
  username: "Steffen"
  password: "Radio"
  type: "full"
  firstName: "Joao"
  lastName: "Freitas"
  age: 24
  email: "joao.freitas@gmail.com"
  country: "PT"
⌄ Ratings: Array
  ⌄ 0: Object
      id: 1
      book: "Harry Potter"
      value: 4
  › 1: Object
  › 2: Object
  › 3: Object
```

Figure 6: Example of a document of an User

This architecture allows us to easily access a User's Ratings by accessing the User.

Indexes

Two Indexes were made to facilitate commonly used searches. As the rating of a book will be important to sort a search it was created descending index that led to lower response times in many queries. Similarly an Index was created for the totalratings variable. This because it is needed to separate the books that are unknown when sorting books by preference (example: the highest rated book could have one unique 5 star Rating but it should not appear on the main page).

In order to test the implementation of our indexes 3 different types of operations were tested in response time 4 times and then the average for response time with and without indexes were tested.

Operation	Indexes	T1(ms)	T2(ms)	T3(ms)	T4(ms)	AVG
10 Best Rated Books	No	157	117	177	164	153.75
10 Best Rated Books	Yes	70	28	18	16	33.00
Find Books by name	No	699	609	468	445	555.25
Find Books by name	Yes	182	242	208	190	205.50
Best Rated Authors	No	446	394	345	322	376.75
Best Rated Authors	Yes	87	50	48	48	58.25

Even though it is a simple way to test the Indexes the results are promising. We chose 3 different functions to test on different situations, and used commonly used functions. We can see that the response time goes down in every situation so the Indexes were well chosen to make the database more effective.

Queries

The Queries on this Database range from the simplest possible,like finding users and books by name to more complex such as finding the most active users.

The simplest queries are all very similar: find user by username, find book by title, find books by author and login. They all are basically just scanning the database to search if a specific field matches or contains the input.

The more common queries are more different and require normally sorting the database by rating and amount of ratings: rate a book.

Operation	Type	Frequency	Cost
1-Retrieve Information on a User/ Book	Read	High	Low
2-Edit Information on a Book/User	Write	Low	Low
3-Give a Rating	Read/Write	Medium	Medium (Requires access to both collections)
4-Amount of Users/Books	Read	Low	Low
5-Best Rated	Read	High	Low(Index)
6-Best rated/Most published/Most rated authors	Read	Low	Medium(Aggregation)
7-Most active Users	Read	Low	High(Complex Aggregation)
8-Login	Read	Low	High(Complex Aggregation)
9-Rating distribution total and by Book	Read	Medium	Low

Operations 1 and 8 are simple read operations of very little cost. Operation 2 is a simple write of also little cost to the database.

Operations of point 6 are aggregation operations that create statistics on authors and organize them accordingly letting admins see the most prolific authors on their website.

Operation 7 is the very expensive because it requires an Unwind operation, which means to separate the Ratings arrays of every User, in order to count the number of arrays per User and see which are the most active on the website. This allows admins to track highly engaged Users as well as possibly bots that might have too much activity. Operations 9 are also expensive because it needs complex aggregation and makes 3 consecutive steps that search in the Users dataset. However, because the User dataset is not as big as the books dataset the response time of this operations is not too high.

Sharding

As stated before sharding was considered for this project but was decided against. The options studied were in Mongo DB, based on the Indexes created.

Initially it was considered to separate the books collection based on the totalratings attributes. This means we could make most of the searches on only the Shard with the highest numbers because the most famous books have the most interactions with the users. This would minimize the scope of searches, however, traffic would also be focused on one shard so it is not the best option as this would also create a problem.

Another option would be to separate the User and Book collection by id but the results of that would be more unpredictable because there would be no way of knowing which elements would be where.

Our best idea for sharding was to use the Book Language as key, but we don't have that value in our collection so we would have to compute it with other services or analyzing the title against some language dictionary.

Implementation

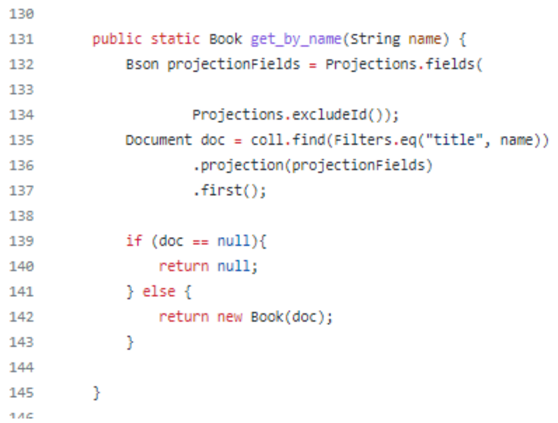
The application developed was developed with abstraction. This means the Mongo DB communications were developed in a parallel class with resources for the main program.

Some of the queries developed are pretty similar so we are using reduced examples here as not to extend the document too much.

Simple Queries

The simplest queries are all very similar: find user by username, find book by title, find books by author and login. They all are basically just scanning the database to search if a specific field matches or contains the input.

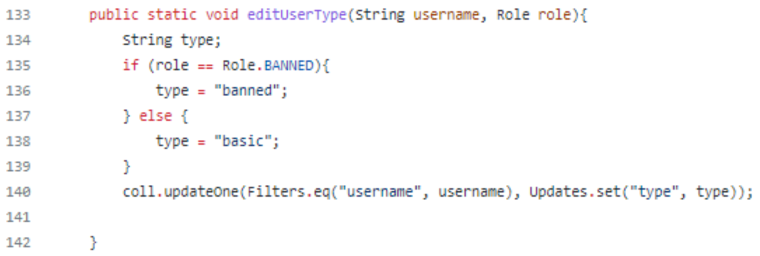
A simple query like Operations 1 and 8 are all just simple searches based on one field. This can be represented by the *get by name* function that searches a book by title.



```java
public static Book get_by_name(String name) {
    Bson projectionFields = Projections.fields(

            Projections.excludeId());
    Document doc = coll.find(Filters.eq("title", name))
            .projection(projectionFields)
            .first();

    if (doc == null){
        return null;
    } else {
        return new Book(doc);
    }

}
```

Figure 7: Get by name function.

Operations 2 are simple write operations, so they can also be modeled the same way by the editUsertype function that changes a User's role.

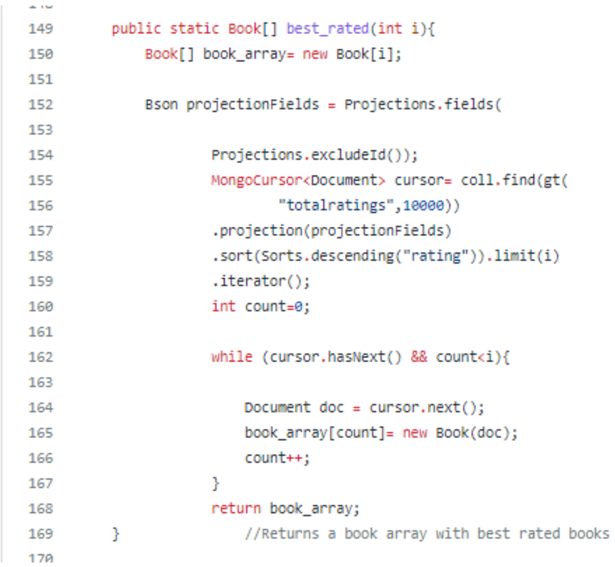


```java
public static void editUserType(String username, Role role){
    String type;
    if (role == Role.BANNED){
        type = "banned";
    } else {
        type = "basic";
    }
    coll.updateOne(Filters.eq("username", username), Updates.set("type", type));

}
```

Figure 8: Get by name function.

Normal Queries

Operations 5 will be a very frequent operation as best rated books are a good way of suggesting new readings for users. It has to filter books above a certain limit of *totalratings* or otherwise it could present a book that was given 5 starts but was rated by only one person. It also sorts the book by rating in descending order (after the introduction of the index the response time of this operation was cut in half) and has to limit the number of results to 10, this should be done in the query to minimize response time.

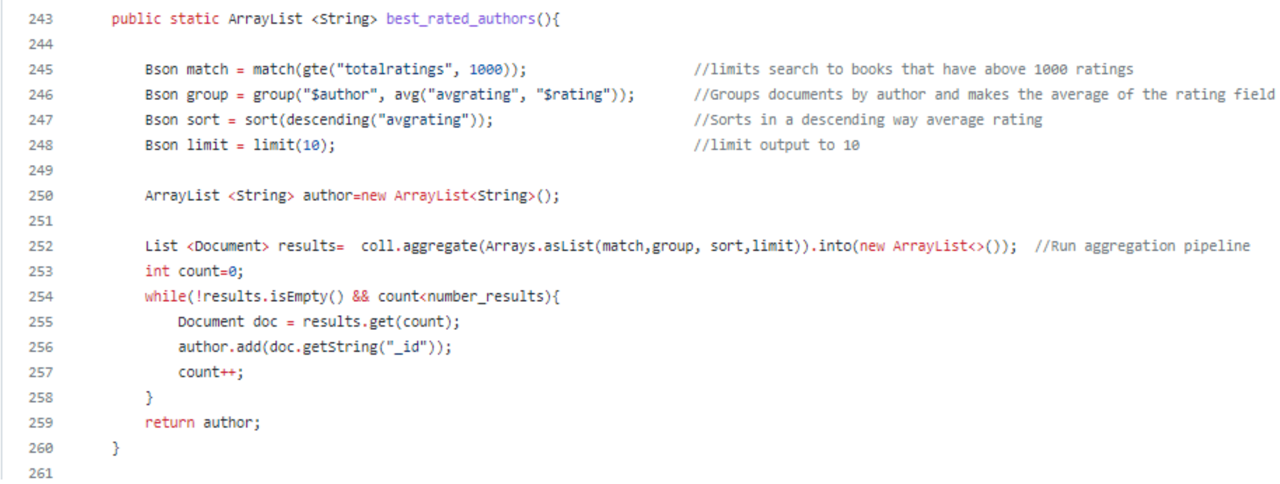


```
public static Book[] best_rated(int i){
    Book[] book_array= new Book[i];

    Bson projectionFields = Projections.fields(

            Projections.excludeId());
            MongoCursor<Document> cursor= coll.find(gt(
                    "totalratings",10000))
            .projection(projectionFields)
            .sort(Sorts.descending("rating")).limit(i)
            .iterator();
            int count=0;

            while (cursor.hasNext() && count<i){

                Document doc = cursor.next();
                book_array[count]= new Book(doc);
                count++;
            }
            return book_array;
}               //Returns a book array with best rated books
```

Figure 9: Best Rated Books function.

Complex Queries

The more complex controls use aggregated pipelines. This is a method of Mongo DB that facilitated queries. It basically divides Queries into stages (every stage performs a specific operation) and the output of every stage is the input of the next. All documentation on it be found here.

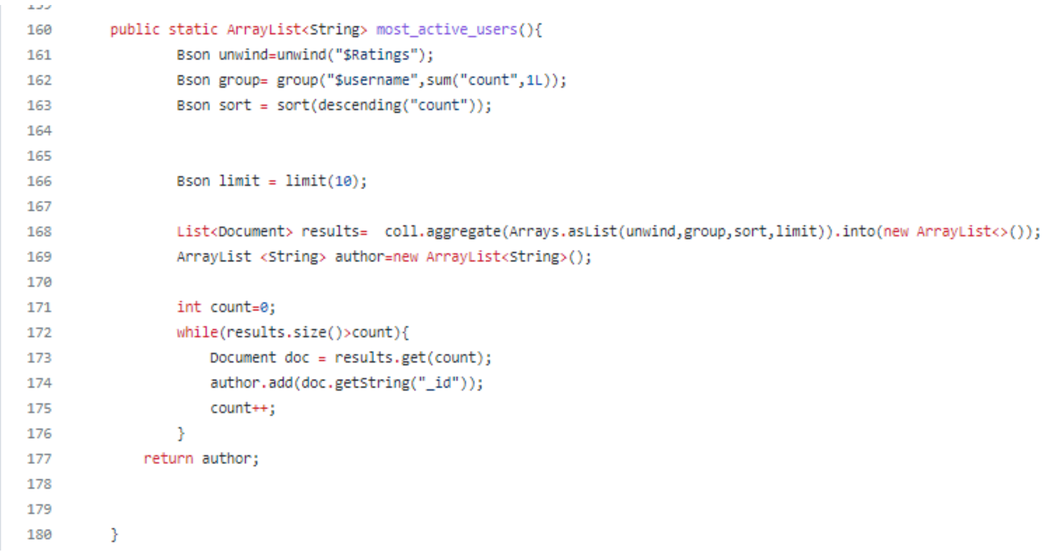


```
public static ArrayList <String> best_rated_authors(){

    Bson match = match(gte("totalratings", 1000));                    //limits search to books that have above 1000 ratings
    Bson group = group("$author", avg("avgrating", "$rating"));       //Groups documents by author and makes the average of the rating field
    Bson sort = sort(descending("avgrating"));                        //Sorts in a descending way average rating
    Bson limit = limit(10);                                           //limit output to 10

    ArrayList <String> author=new ArrayList<String>();

    List <Document> results=  coll.aggregate(Arrays.asList(match,group, sort,limit)).into(new ArrayList<>());  //Run aggregation pipeline
    int count=0;
    while(!results.isEmpty() && count<number_results){
        Document doc = results.get(count);
        author.add(doc.getString("_id"));
        count++;
    }
    return author;
}
```

Figure 10: Best Rated Authors function.

In the picture you can see what every stage does, and every stage (match,group,sort,limit) will act upon the output of the previous. All other statistics on authors (most published and most famous) have similar structures. The most published returns the authors that appear in more books on the database while the most famous adds up the *totalratings* by author, so it returns the authors whose books have the most interactions.

Because of the embedded ratings on the Users the most active users, that returns users with the most ratings, is the most complex query.



```
public static ArrayList<String> most_active_users(){
        Bson unwind=unwind("$Ratings");
        Bson group= group("$username",sum("count",1L));
        Bson sort = sort(descending("count"));


        Bson limit = limit(10);

        List<Document> results=  coll.aggregate(Arrays.asList(unwind,group,sort,limit)).into(new ArrayList<>());
        ArrayList <String> author=new ArrayList<String>();

        int count=0;
        while(results.size()>count){
            Document doc = results.get(count);
            author.add(doc.getString("_id"));
            count++;
        }
    return author;


}
```

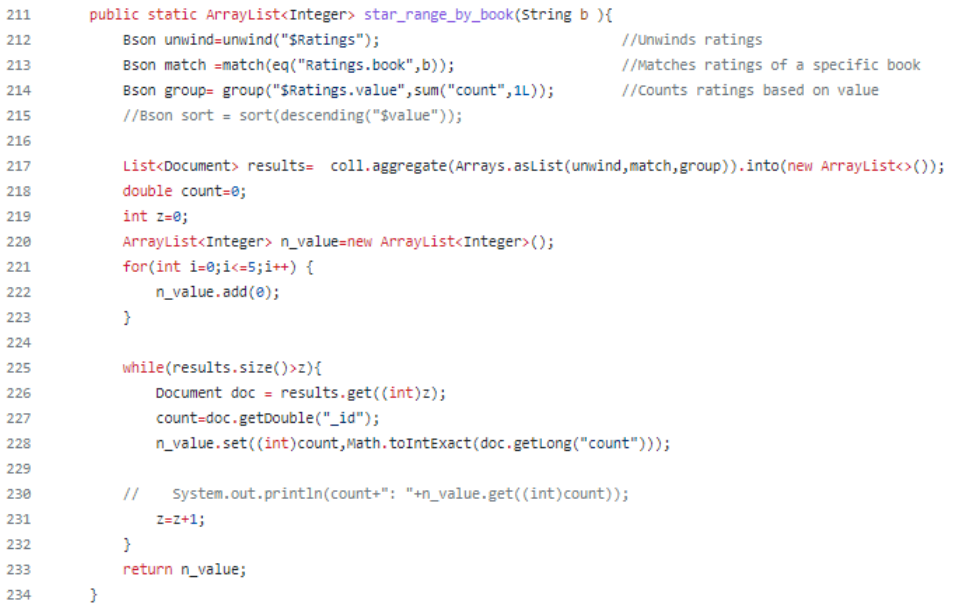
Figure 11: Most Active Users function.

In order to count the amount of ratings it is necessary to separate the ratings array. This will be done with the upwind command, which will give as output one different object for each array. Then the group operation will group the objects by username and count the number of objects.

In order to present statistics to the users on the Ratings a query was needed to search the distribution of ratings in total and in every book, meaning the amount of 1,2,3,4 and 5 ratings.

This will allow the admins to search ratings distribution and compare books distribution with total distribution.

This function is the more complicated aggregation pipeline used. It starts the same way with unwind to separate the Ratings Array and then it will use a match operation in order to separate the objects that are from the specific book. Then a group operation is utilised to group Objects by Value of rating and it counts the number of elements in every value.



```
public static ArrayList<Integer> star_range_by_book(String b ){
    Bson unwind=unwind("$Ratings");                         //Unwinds ratings
    Bson match =match(eq("Ratings.book",b));                //Matches ratings of a specific book
    Bson group= group("$Ratings.value",sum("count",1L));    //Counts ratings based on value
    //Bson sort = sort(descending("$value"));

    List<Document> results=  coll.aggregate(Arrays.asList(unwind,match,group)).into(new ArrayList<>());
    double count=0;
    int z=0;
    ArrayList<Integer> n_value=new ArrayList<Integer>();
    for(int i=0;i<=5;i++) {
        n_value.add(0);
    }

    while(results.size()>z){
        Document doc = results.get((int)z);
        count=doc.getDouble("_id");
        n_value.set((int)count,Math.toIntExact(doc.getLong("count")));

    //    System.out.println(count+": "+n_value.get((int)count));
        z=z+1;
    }
    return n_value;
}
```

Figure 12: Star Distribution by Book function.

NEO4J DESIGN AND IMPLEMENTATION

Graph Design

Nodes

Two main types of nodes are created within the graph database:

- **User Node** - Represent a user that is registered in the application, and is assigned one attribute which is the **username**. This is the only information needed to access the user's information in the document database.
- **Book Node** - Represent a book that rated by a user(s). A book can not exist within the graph database without being rated by at least one user. The information attributed to the *Book Node* is not all the attributes belonging to the *Book* class, but rather only the *title*.

Relationships

There are two different types of relationships between nodes in the graph database:

- **User** → **FOLLOW** → **User** - this relationship represent a *User* following another *User*, and will be removed once a *User* decides to unfollow the other *User*. There are no attributes associated with this relationship.
- **User** → **RATED** → **Book** - when a user want to rate a book upon completion, it is possible to give the particular book a rating from 1 to 5, which will create a relationship between the *User* and the *Book*. The rating will be an attribute associated with this relationship in the form of an integer.

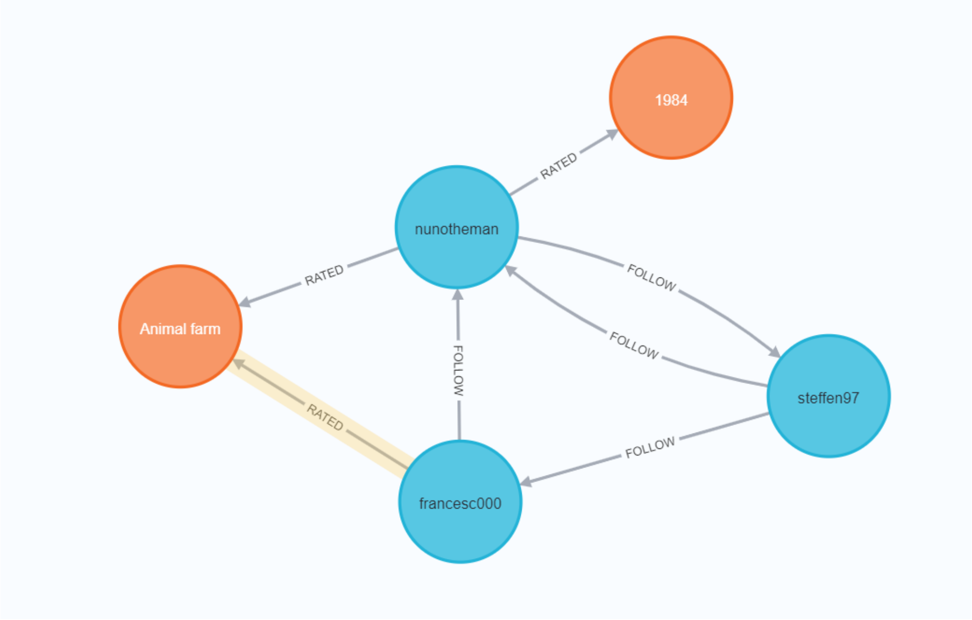
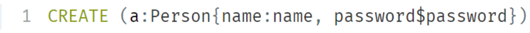


Figure 13: An overview of the nodes and relationships in the graph database created in Neo4j. Blue nodes are *Users* and orange nodes are *Books*.

The types of nodes and relationships can be seen in figure 13

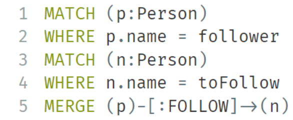
Queries in Neo4j

Create user node:



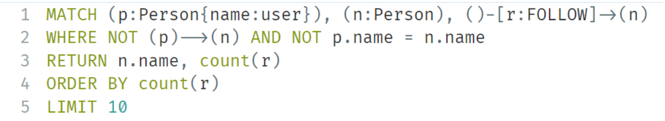
```
CREATE (a:Person{name:name, password$password})
```

Follow another user:



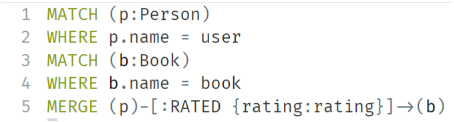
```
MATCH (p:Person)
WHERE p.name = follower
MATCH (n:Person)
WHERE n.name = toFollow
MERGE (p)-[:FOLLOW]→(n)
```

Get a list of maximum 10 users, based on users that the user follows, where the suggested users with most followers are ranked highest:



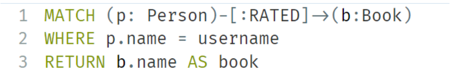
```
MATCH (p:Person{name:user}), (n:Person), ()-[r:FOLLOW]→(n)
WHERE NOT (p)⟶(n) AND NOT p.name = n.name
RETURN n.name, count(r)
ORDER BY count(r)
LIMIT 10
```

Rate a book on a scale from 1 to 5:



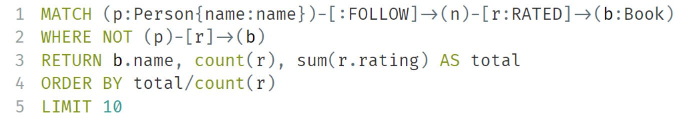
```
MATCH (p:Person)
WHERE p.name = user
MATCH (b:Book)
WHERE b.name = book
MERGE (p)-[:RATED {rating:rating}]→(b)
```

Get all the rated books from a user:



```
MATCH (p: Person)-[:RATED]→(b:Book)
WHERE p.name = username
RETURN b.name AS book
```

Get a suggestion of maximum 10 books, where the suggestions are based on the average rating from you friends. The book with the highest average rating will be highest ranked:



```
MATCH (p:Person{name:name})-[:FOLLOW]→(n)-[r:RATED]→(b:Book)
WHERE NOT (p)-[r]→(b)
RETURN b.name, count(r), sum(r.rating) AS total
ORDER BY total/count(r)
LIMIT 10
```

Neo4j - Read Operations

Operation	Frequency	cost
Get suggestions of users to follow	High	Medium (multiple reads)
Get suggestions of books to read	High	Low (multiple reads)

Neo4j - Write Operations

Operation	Frequency	cost
Create/Delete user	Low	Low (create/delete node)
Create/Delete book	Low	Low (create/delete node)
Follow/Unfollow user	Med	Low (create/delete relationship)
Rate a book	Med	Low (create relationship)

USER MANUAL

In this section we will explain how to run the application (in high-level) and how to perform the main actions.

- Clone from GitHub: https://github.com/francescofact/LSDB-BookWorm
- Create a Neo4j instance
- Run it.unipi.lsdb.Main throw your favourite IDE with all libraries linked

In your compiler configuration add the following ”VM Options”:

```
--module-path <PATH_TO_LIBS>/javafx-sdk-17.0.1/lib --add-modules=javafx.controls,javafx.fxml
```

To jump start your databases with Users, Ratings and Following we created a Python Script that will take care of it. It will create 200 users with common username and password from the relative dataset downloaded from Kaggle (not included because of their sizes).

Now that you have launched the application you will see the Guest HomePage. In this window we can go to the login, search some books for title, search some users, see the 6 best rated books of the platform and get some authors' statistics.

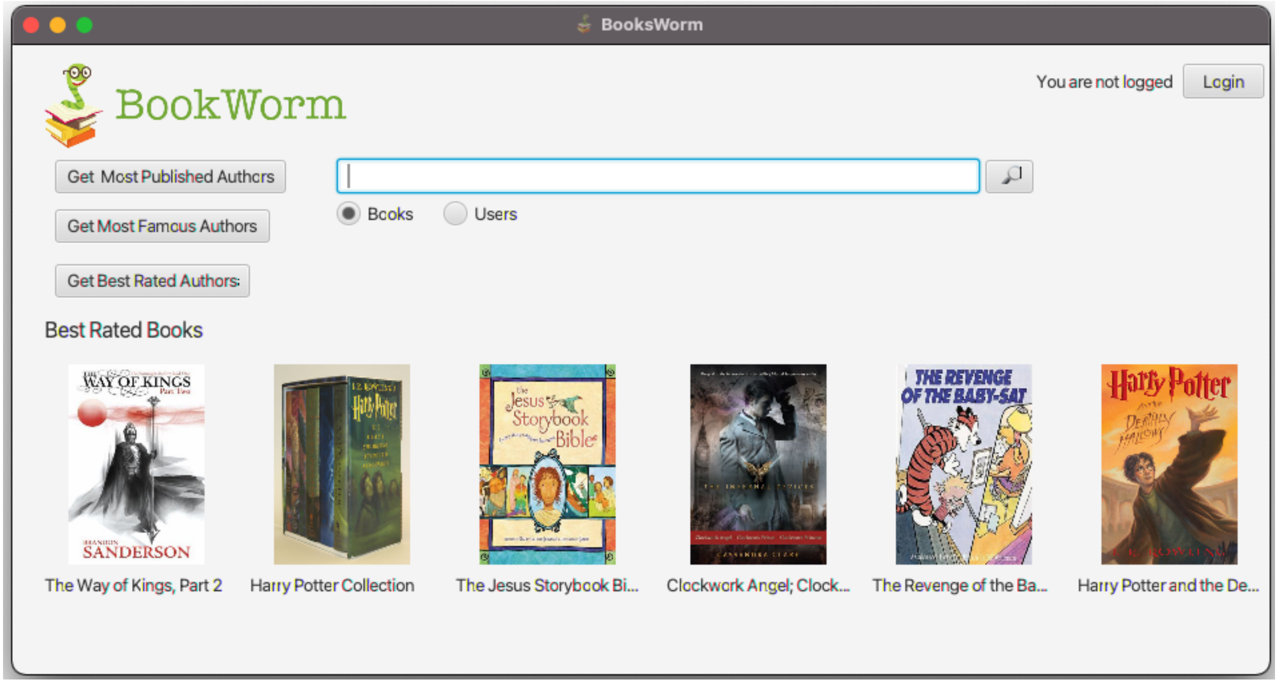


Figure 14: Guest Homepage

We can procede to login in as regular user clicking the ”Login” button

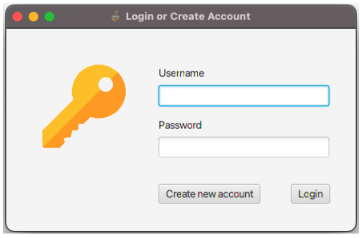
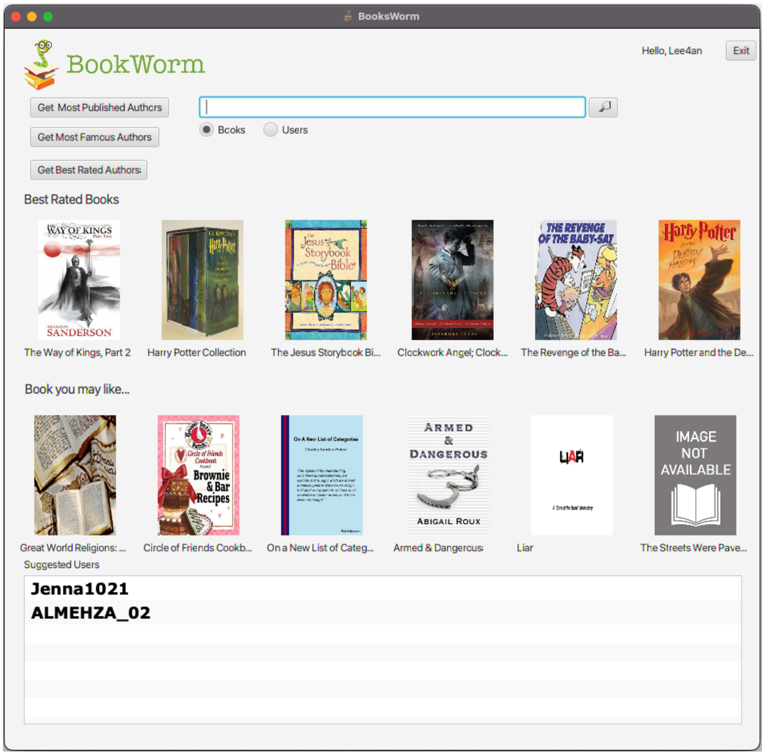


Figure 15: Login

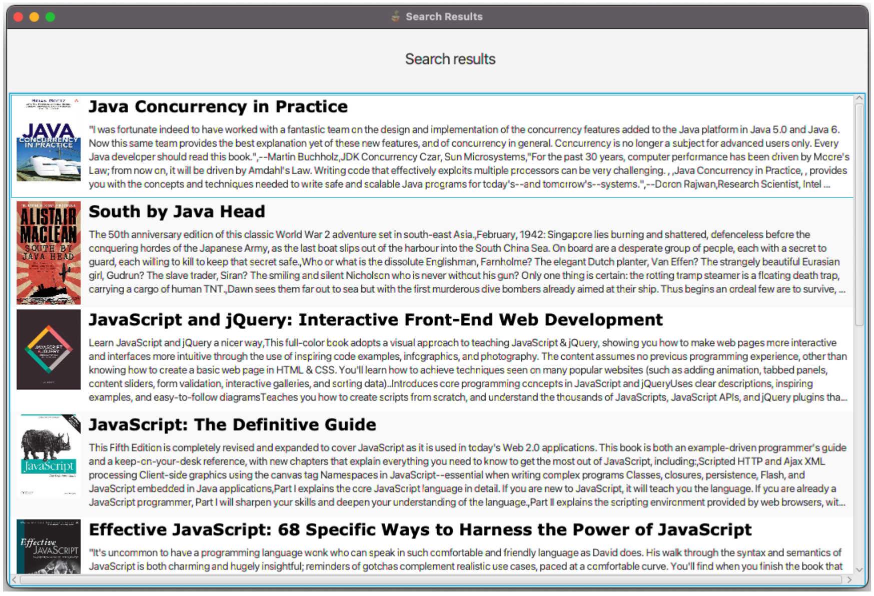
After entering our credentials and confirming we can access the Regular User Homepage. Here we can see also suggestions of Books and Users based on the user we follow.



Jenna1021
ALMEHZA_02

Figure 16: Homepage Regular User

From the home page we can search for book. For example this will be the output of the query for ”Java” books:



Search results

Java Concurrency in Practice

"I was fortunate indeed to have worked with a fantastic team on the design and implementation of the concurrency features added to the Java platform in Java 5.0 and Java 6. Now this same team provides the best explanation yet of these new features, and of concurrency in general. Concurrency is no longer a subject for advanced users only. Every Java develcper should read this book.",--Martin Buchholz,JDK Concurrency Czar, Sun Microsystems,"For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law. Writing code that effectively exploits multiple processors can be very challenging. , ,Java Concurrency in Practice, , provides you with the concepts and techniques needed to write safe and scalable Java programs for today's--and tomorrow's--systems.",--Doron Rajwan,Research Scientist, Intel ...

South by Java Head

The 50th anniversary edition of this classic World War 2 adventure set in south-east Asia.,February, 1942: Singapore lies burning and shattered, defenceless before the conquering hordes of the Japanese Army, as the last boat slips out of the harbour into the South China Sea. On board are a desperate group of people, each with a secret to guard, each willing to kill to keep that secret safe.,Who or what is the dissolute Englishman, Farnholme? The elegant Dutch planter, Van Effen? The strangely beautiful Eurasian girl, Gudrun? The slave trader, Siran? The smiling and silent Nicholson who is never without his gun? Only one thing is certain: the rotting tramp steamer is a floating death trap, carrying a cargo of human TNT.,Dawn sees them far out to sea but with the first murderous dive bombers already aimed at their ship. Thus begins an ordeal few are to survive, ...

JavaScript and jQuery: Interactive Front-End Web Development

Learn JavaScript and jQuery a nicer way,This full-color book adopts a visual approach to teaching JavaScript & jQuery, showing you how to make web pages more interactive and interfaces more intuitive through the use of inspiring code examples, infographics, and photography. The content assumes no previous programming experience, other than knowing how to create a basic web page in HTML & CSS. You'll learn how to achieve techniques seen on many popular websites (such as adding animation, tabbed panels, content sliders, form validation, interactive galleries, and sorting data)..Introduces core programming concepts in JavaScript and jQueryUses clear descriptions, inspiring examples, and easy-to-follow diagramsTeaches you how to create scripts from scratch, and understand the thousands of JavaScripts, JavaScript APIs, and jQuery plugins tha...

JavaScript: The Definitive Guide

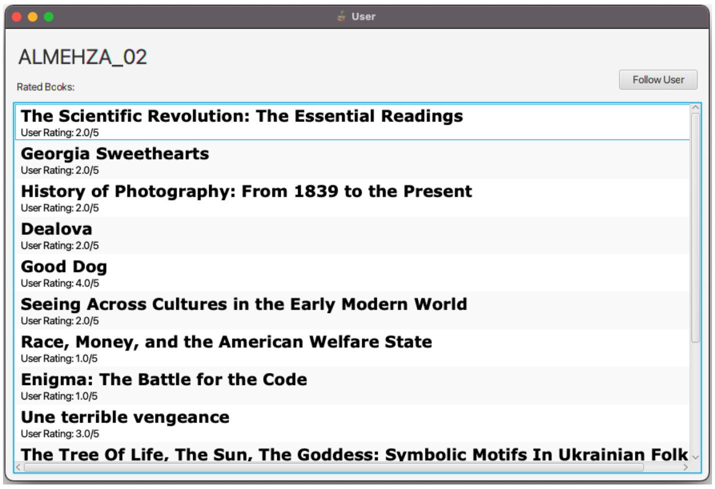
This Fifth Edition is completely revised and expanded to cover JavaScript as it is used in today's Web 2.0 applications. This book is both an example-driven programmer's guide and a keep-on-your-desk reference, with new chapters that explain everything you need to know to get the most out of JavaScript, including:,Scripted HTTP and Ajax XML processing Client-side graphics using the canvas tag Namespaces in JavaScript--essential when writing complex programs Classes, closures, persistence, Flash, and JavaScript embedded in Java applications,Part I explains the core JavaScript language in detail. If you are new to JavaScript, it will teach you the language. If you are already a JavaScript programmer, Part I will sharpen your skills and deepen your understanding of the language.,Part II explains the scripting environment provided by web browsers, wit...

Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript

"It's uncommon to have a programming language wonk who can speak in such comfortable and friendly language as David does. His walk through the syntax and semantics of JavaScript is both charming and hugely insightful; reminders of gotchas complement realistic use cases, paced at a comfortable curve. You'll find when you finish the book that

Figure 17: Searching for "Java" books

We can also look for some user and see what they have rated so far and also follow them:



ALMEHZA_02

Rated Books:

The Scientific Revolution: The Essential Readings
User Rating: 2.0/5

Georgia Sweethearts
User Rating: 2.0/5

History of Photography: From 1839 to the Present
User Rating: 2.0/5

Dealova
User Rating: 2.0/5

Good Dog
User Rating: 4.0/5

Seeing Across Cultures in the Early Modern World
User Rating: 2.0/5

Race, Money, and the American Welfare State
User Rating: 1.0/5

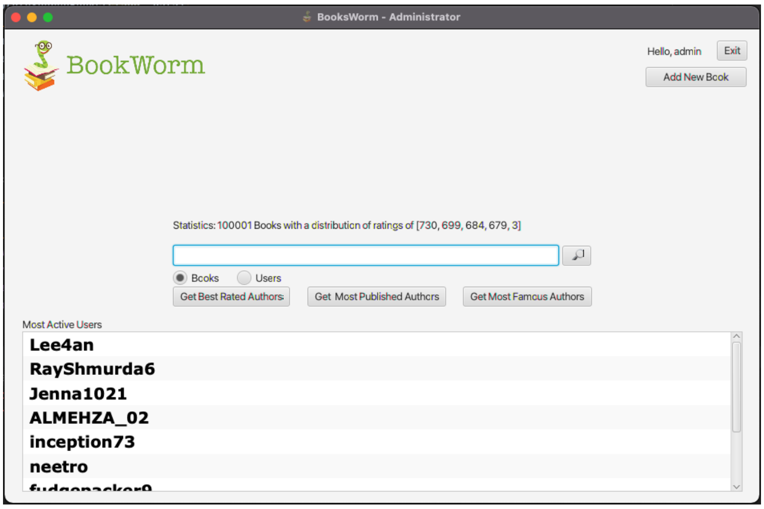
Enigma: The Battle for the Code
User Rating: 1.0/5

Une terrible vengeance
User Rating: 3.0/5

The Tree Of Life, The Sun, The Goddess: Symbolic Motifs In Ukrainian Folk

Figure 18: User Profile

If we logout from the regular user and login with the credentials of an Admin account we will see the Admin Homepage. Here we have also the 10 most active users and a button to create a new book.



Hello, admin

Statistics: 100001 Books with a distribution of ratings of [730, 699, 684, 679, 3]

● Books ○ Users

Most Active Users

Lee4an
RayShmurda6
Jenna1021
ALMEHZA_02
inception73
neetro
fudgepacker9

Figure 19: Admin Homepage

If we click on the ”Add New Book” it will open a new windows where we can input our book details:

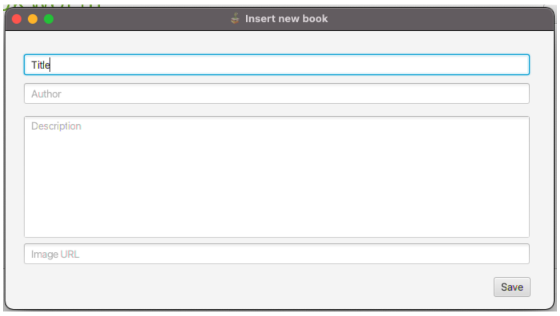
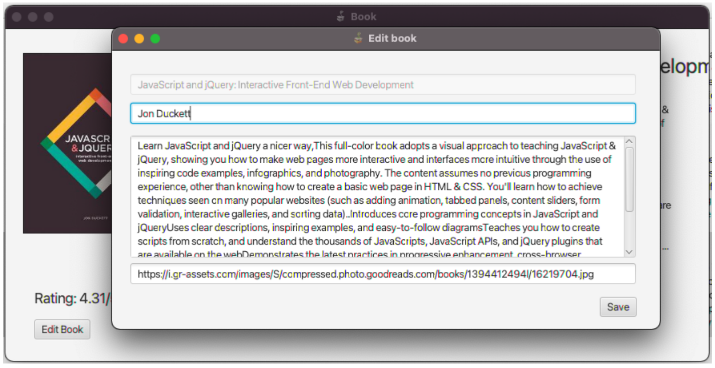


Figure 20: Creating a Book

Logged as an Admin we can also search for a book and edit its fields:



JavaScript and jQuery: Interactive Front-End Web Development

Jon Duckett

Learn JavaScript and jQuery a nicer way,This full-color book adopts a visual approach to teaching JavaScript & jQuery, showing you how to make web pages more interactive and interfaces more intuitive through the use of inspiring code examples, infographics, and photography. The content assumes no previous programming experience, other than knowing how to create a basic web page in HTML & CSS. You'll learn how to achieve techniques seen on many popular websites (such as adding animation, tabbed panels, content sliders, form validation, interactive galleries, and sorting data)..Introduces core programming concepts in JavaScript and jQueryUses clear descriptions, inspiring examples, and easy-to-follow diagramsTeaches you how to create scripts from scratch, and understand the thousands of JavaScripts, JavaScript APIs, and jQuery plugins that are available on the webDemonstrates the latest practices in progressive enhancement, cross-browser

https://i.gr-assets.com/images/S/compressed.photo.goodreads.com/books/1394412494l/16219704.jpg

Figure 21: Editing a Book