UNIVERSITÀ DI PISA

Large-Scale and Multi-Structured Databases

WeRead

Iacopo Bicchierini, Matteo Guidotti

Github repository: https://github.com/Bicchie/WeRead

Contents

Chapter 1

Introduction and Requirements

WeRead is a niche social networking application based on sharing several information about books. Users who are registered to the service will be able to search and find books of every type and category, by means of different filtering settings, in order to get information about it and read the reviews written by other users, that could be liked too. Each user can build its own favourite books list adding whatever books it wants, in addition to the possibility to create several reading list that will be shared with the other users too. There could be also different types of interactions between users, in fact one of the service is that a consumer can follow another user in order to receive suggestions based on the interests of the followed users. In the same way, a user can like reading lists created by others, in order to be suggested with books that are similar to the ones contained in that and in all the lists liked.

1.1 Functional Requirements

In the application we can have three different types of users: *unlogged*, *logged*, *admin*. Here we show the possible actions they each of them can perform

Unlogged user

It is the user who opens and uses the application for the first time

- **Registration**: In order to access the application an user must sign-up. Otherwise he is not allowed to access and to use all the functionalities.

Logged user

It is the normal user, who registered itself to the service times ago

- **Login/Logout**: The only way to access the application, as we said previously, is to sign-up and login. At the end the user can logout and close the session.

- **Search a book**: It's possible to search a book by means of its title, category, author and publisher
- **Search a user**: It's possible to search other users by means of the username
- **Browse book information**: Every book has a dedicated page in order to show all the information about it
- **Browse favourite books of other users**: It's possible to read and have information about all the books that another user put in its favourite books list
- **Browse reading lists of other users**: It's possible to read and have information about all the reading lists that another user has created
- **Browse reviews**: It's possible to read the reviews written by other users
- **Browse suggested books**: Suggestions are provided to the users based on their interactions with other users, books, reading lists and their interests
- **Browse suggested reading lists**: Suggestions are provided to the users based on their interactions with other users and reading lists
- **Like reviews**: It's possible to like a review written by another user
- **Add a book the favourite books list**: It's possible to add whatever book to the user personal favourite books list
- **Follow another user**: Users can follow each other

Admin

It is a special user who can perform privileged actions and obtain additional statistics

- **Add a new book**: Add a new book to the social network collection
- **Insights**: It is able to access to statistics about users, books and their relationships and statistics

1.2 Non Functional Requirements

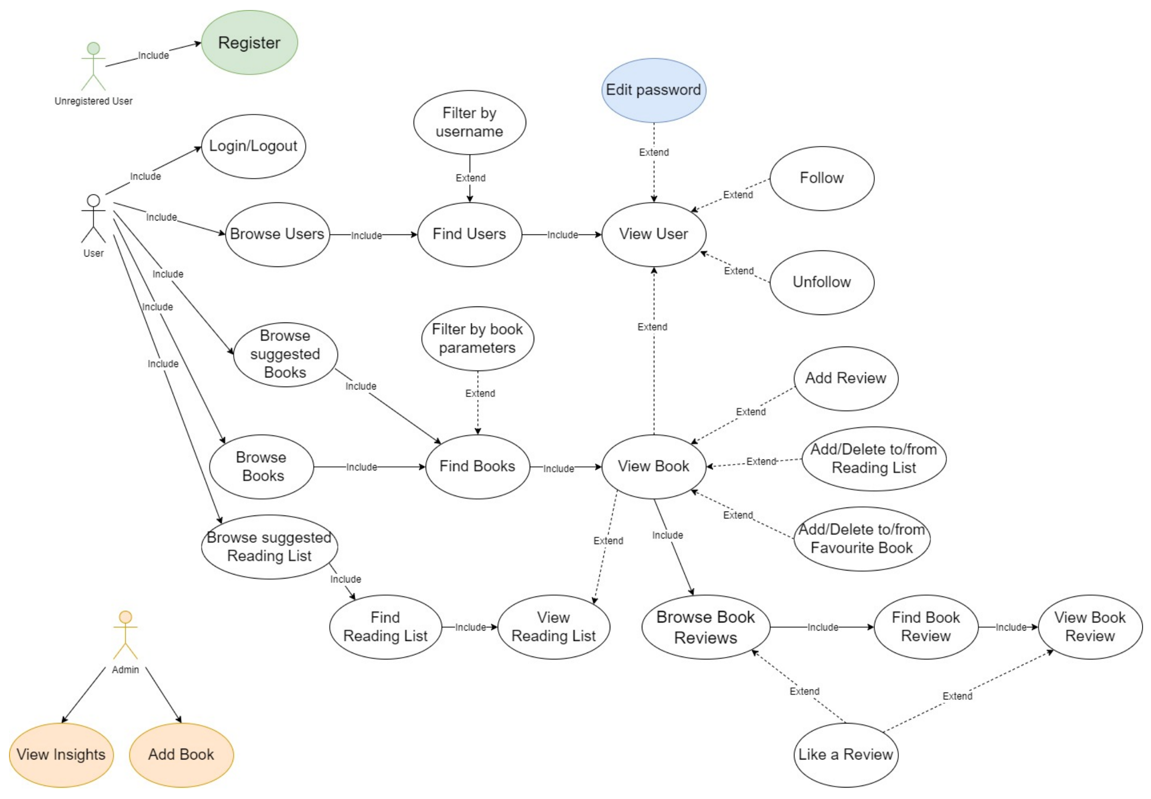
The non-functional requirements of the applications are described in the following lists:

- **Usability**: The application must be user-friendly so a GUI is adopted.
- **High Availability**: For modern shared-data systems and for a social network application, the most crucial requirement is that the service must be always available.
- **Reliability**: The application must work without crashes and so it must handle exceptions if they occur.
- **Fast Responsiveness**: The application must response in a short time, we want to avoid too long waits for the user.

Chapter 2

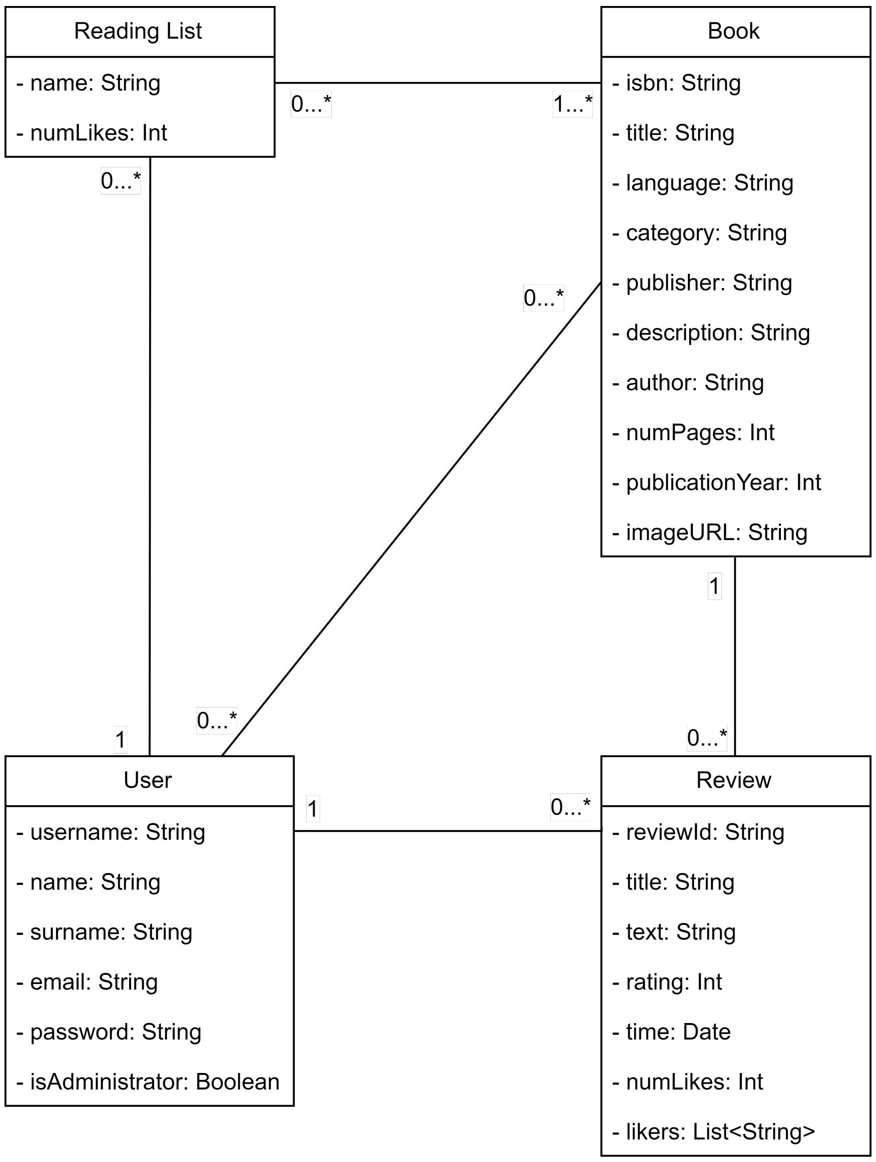
Specifications

2.1 Actors and Use Case Diagram



Please notice that the blue circles are the ones which describes actions available only for the owner of the object on which the actions are applied

2.2 UML Class Diagram



Let’s see the meanings of each attribute of these classes

User

- *username*: String that represents the username of the user. The username is unique between every user
- *name*: String containing the name of the user
- *surname*: String containing the surname of the user
- *email*: String containing the email address of the user

- *password*: String containing the password of the user
- *isAdministrator*: Boolean needed to differentiate between admins and normal users

Book

- *isbn*: String representing the isbn of the book (it is unique for definition)
- *title*: String containing the title of the book
- *language*: String representing the langugae in which the book is written
- *publisher*: String containing the publisher of the book
- *description*: String containing a brief description of the book
- *author*: String containing the name of the author of the book
- *numPages*: Integer containig the number of pages that compose the book
- *publicationYear*: Integer containig the year in which the book has been published
- *imageURL*: String containig the URL of the cover of the book

Review

- *reviewId*: String composed by the concatenation between the username of the user who wrote the review and the isbn of the reviewed book. As a consequence of the previously explained constraints, it will be unique
- *title*: String containing the title of the reviewed book
- *text*: String containing the text of the review
- *rating*: Integer between 0 and 5 assigned to the reviewed book
- *time*: Date variable representing the moment in which the review has been written
- *numLikes*: Integer representing the number of likes received by the review
- *likers*: List of usernames of the users that liked the review

Reading List

- *name*: String containig the name of the reading list
- *numLikes*: Integer representing the number of likes received by the reading list

Chapter 3

Architectural Design

Software Architecture

WeRead is based on the *client-server* paradigm. Let's see how our system is divided in these two parts

Client side

The code is contained in three packages

controller Package The classes of this package constitute the *front-end* module and exploits all the other classes in order to build the graphic interface and define the events involving the user's interaction with it.

model Package The *middleware* is constituted by the classes that belongs to this package, that are User, ReadingList, Review, Book. These classes have the objective to maintain information regarding the entitities that characterized our system and provide functionalities to obtain them to the other packages. The class Session belongs to this package too, it is important to maintain the information about the current session, like the logged user.

persistence Package It contains two classes used to configure the connection and access to the MongoDB and Neo4J databases, and it is the *back-end* of our application.

Server side

The server side is made up of 3 instances of MongoDB running in local and also a neo4j one.

Database choices and motivation

We decided to exploit two different types of databases:

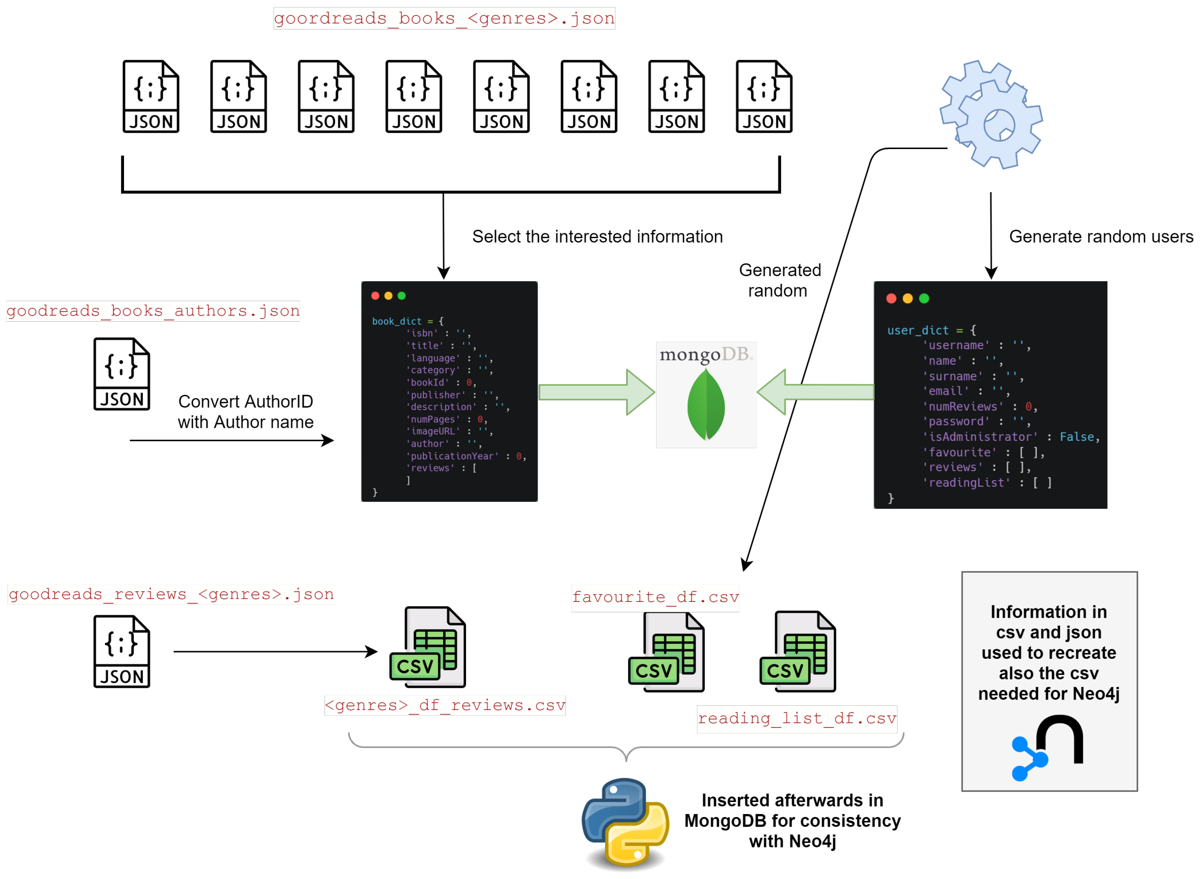
- **MongoDB**: We have chosen a documentDB to exploit the support that provides in terms of storage and document embedding. Another important aspect is its suitability to indexing and complex queries elaboration, this last point is fundamental for the analytic part of the application.

- **Neo4j**: We have chosen a graphDB mainly for the support to suggestion operations, it is fundamental to retrieve information, traversing from an entity of the database to another, without affecting too much the performances. Graph databases are very suitable for this task.

Dataset Composition and Data Preprocessing

We tried different techniques, from web crawling to call Books API from Google, then we found in the *Irwin Joan Jacobs School of Engineering* of San Diego site a huge collection of goodread.com data. Comparing the already crawled and gathered data with the new found, we discover that the latter contained the other ones, so we decided to keep only It.

They gives json file containing several books and reviews belonging to 8 genres: Children, Comics Graphics, Fantasy Paranormal, History Biography, Mystery Thriller Crime, Poetry, Romance and Young Adult.



goordreads_books_<genres>.json
JSON
Select the interested information
Generated random
Generate random users
goodreads_books_authors.json
Convert AuthorID with Author name
mongoDB
goodreads_reviews_<genres>.json
CSV
<genres>_df_reviews.csv
favourite_df.csv
reading_list_df.csv
Information in csv and json used to recreate also the csv needed for Neo4j
Inserted afterwards in MongoDB for consistency with Neo4j

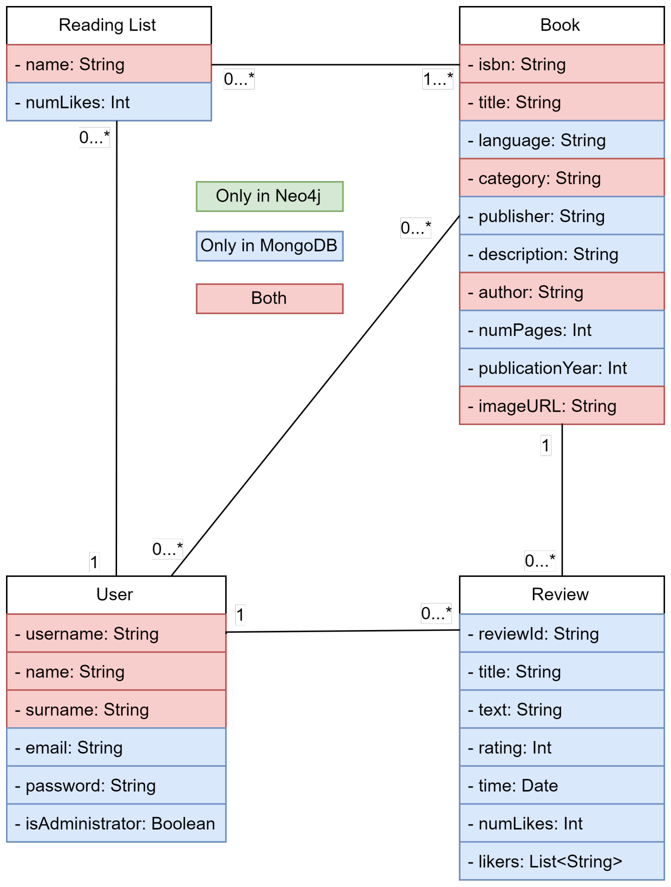
```
book_dict = {
        'isbn' : '',
        'title' : '',
        'language' : '',
        'category' : '',
        'bookId' : 0,
        'publisher' : '',
        'description' : '',
        'numPages' : 0,
        'imageURL' : '',
        'author' : '',
        'publicationYear' : 0,
        'reviews' : [
        ]
}
```

```
user_dict = {
     'username' : '',
     'name' : '',
     'surname' : '',
     'email' : '',
     'numReviews' : 0,
     'password' : '',
     'isAdministrator' : False,
     'favourite' : [ ],
     'reviews' : [ ],
     'readingList' : [ ]
}
```

We created using a Notebook Python, the two *json* with the Users and Books, needed to populate the two respective collections in **MongoDB**. Some operations was needed like the filtering of the original json file when some important information was missing, but also the mapping of the AuthorID with the Author name. The Users was generated randomly as well as the favourite books and the reading lists. To populate **Neo4j** instead we created the csv files exploiting the json file, some information like the followings of the User was created as well as the likes of the Reviews or the Reading Lists. In order to keep the two collections in MongoDB consistent, we create the *csv* files with Reviews, Favorite books and Reading List and with a Python script we populate the reviews, favorite and reading list field respectively.

Data among Databases

This is how we intend to keep the information related to the classes inside the two databases. MongoDB stores most of the informations, Neo4j contains only the relational informations and some information regarding the Book and the User.



Chapter 4

Design and Implementation of MongoDB

The database have to contain two collections in order to store all the information our system need. We introduced some redundancies between them in order to have a fast access to all the information that are required to be showed when a user or a book is requested. In particular, both the collections maintains same information about the reviews, because they have to be showned both in the user page and in the page related to a certain book. In addition to them, we have other redundancies in the *books* array of each reading list and in the *favourite* array; in both the cases, some information about each book are reported in the users collection, in order to be able to show favourite books and books belonging to reading lists of a user without having to load them from the books collection. Moreover the Documents structure is very close to the Object structure in code.

4.1 Collections

4.1.1 Users Collection

In addition to the information about the user, the collection collects data about the reading lists created by the user itself. In this way, the system is able to show the lists belonging to a certain user without having to merge different collections

```
{
  "_id": {
    "$oid": "635f93733c40a8d104e7dce0"
  },
  "username": "Donna_Freeman",
  "name": "Donna",
  "surname": "Freeman",
  "email": "Freeman.Donna@gmail.com",
  "password": "NpXEG1iq5vb31MDA",
  "isAdministrator": false,
  "favourite": [
    {
      "isbn": "9789934049354",
      "title": "urkburgers",
      "imageURL": "https://images.gr—assets.com/books/1465760936m/30529043.jpg",
      "author": "David Walliams"
    },
    {
      "isbn": "9781784452889",
      "title": "See You Later, Alligator",
      "imageURL": "https://images.gr—assets.com/books/1434065482m/25711578.jpg",
      "author": "Sally Hopgood"
    }
  ],
  "reviews": [
    {
      "reviewId": "Donna_Freeman:9780689865404",
      "title": "Specials (Uglies, #3)",
      "text": "The end was super lame. She was way too \"preachy\"",
      "rating": 2,
      "time": {
        "$date": {
          "$numberLong": "1545451200000"
        }
      },
      "numLikes": 0,
      "likers": []
    }
  ],
  "readingList": [
    {
      "name": "construe piscis",
      "numLikes": 1,
      "books": [
        {
          "isbn": "9780553494600",
          "title": "Last Shot: A Final Four Mystery (The Sports Beat, #1",
          "imageURL": "https://images.gr—assets.com/books/1407112788m/75641.jpg",
          "author": "John Feinstein"
        },
        {
          "isbn": "9781492618522",
          "title": "Awake",
          "imageURL": "https://images.gr—assets.com/books/1425499436m/22881022.jpg",
          "author": "Natasha Preston"
        }
      ]
    }
  ]
}
```

4.1.2 Books Collection

```json
{
  "_id": {
    "$oid": "636b956d475e8ea8ebf49417"
  },
  "isbn": "9780316939928",
  "title": "The Sound of Colors: A Journey of the Imagination",
  "language": "eng",
  "category": "Easy Reader",
  "publisher": "Little, Brown Books for Young Readers",
  "description": "A young woman losing her vision rides the subway with her dog in search of emotional healing.",
  "numPages": 80,
  "imageURL": "https://images.gr—assets.com/books/1344265410m/750456.jpg",
  "author": "Jimmy Liao",
  "publicationYear": "2006",
  "reviews": [
    {
      "reviewId": "Valerie_Perry:9780316939928",
      "title": "The Sound of Colors: A Journey of the Imagination",
      "text": "Accompanied by her scruffy dog companion, a woman who is losing or has lost most of her eyesight ventures
      out into the world. As she waits for and then rides on the subway train, she is careful not to ...",
      "rating": 4,
      "time": {
        "$date": {
          "$numberLong": "1568077200000"
        }
      },
      "numLikes": 0,
      "likers": []
    },
    {
      "reviewId": "Lynn_Obrien:9780316939928",
      "title": "The Sound of Colors: A Journey of the Imagination",
      "text": "Beautiful children's book about a blind girl and what she imagines on a typical subway ride. Lovely, drawn
      with detailed color and line.",
      "rating": 5,
      "time": {
        "$date": {
          "$numberLong": "1651028400000"
        }
      },
      "numLikes": 0,
      "likers": []
    }
  ]
}
```

4.2 Queries Implementation

4.2.1 CRUD operations

All the CRUD operations regarding MongoDB are contained in the persistence.MongoDBDriver class

	Operation
Create	*addUser*: Create a new user document with its information
	addBook : Create a new book document with its information
Read	*getUserInfo*: Get User information by username
	userExists: Check uniqueness of a username
	checkLogin: Check username and password of a certain User
	getBookInformation: Get Book information by book's isbn
	countBookByTitle: Count books given a title
	countBookByCategory: Count books given a book category
	countBookByAuthor: Count books given the name of an author
	countBookByPublisher: Count books list given the publisher
	countBookByYear: Count books list given the year of publication
	searchBookByTitle: Get a book list given a title
	searchBookByCategory: Get a book list given a book category
	searchBookByAuthor: Get a book list given the name of an author
	searchBookByPublisher: Get a book list given the publisher
	searchBookByYear: Get a book list given the year of publication
	getReadingList : Get a certaing reading list of a certain user
	getAvgRating: Get the average rating of the reviews given to a certain book
Update	*addBookToFavorite*: Add a book to the favourite books list of a user
	removeBookFromFavorite: Remove a book from the favourite books list of a user
	addNewReview: Add a new review to a book
	addLikeReview: Add a like to a review written by another user
	removeLikeReview: Remove a like from a review written by another user
	addNewReadingList : Create a new reading list
	removeReadingList : Delete a reading list
	addBookToReadingList : Add a new book to a reading list
	removeBookFromReadingList : Remove a book from a reading list
	addLikeReadingList : Add a like to a reading list of another user
	removeLikeReadingList : Remove a like from a reading list of a certain user
	changeUserPassword : Update the password of the user
Delete	Delete a User

4.2.2 Queries Analysis

The write operations specified above are less frequent then the read operations, the latter are more frequently and are performed on higher number of documents. In addition the requirements of the system are availability and fast response time, for these reasons we set:

- **Write Concern: 1**
- **Read Preference: nearest**

The first one is to allow the system to perform the write operation as quickly as possible, the writes affect only the primary node and replicas will be updated next in time (eventual consistency paradigm). The nearest read preferences specifies that the read operations are performed on the node with the fastest response, that is, the one with the least network latency.

4.2.3 Analytics queries

In order to be able to extract interesting and useful information from our system, we decided to have some queries that exploit aggregation pipelines

Best books by average review rating

It is important to understand which are the most appreciated books by the users, and in order to do that we obtain a list of books ordered by means of the average of the ratings they received in the reviews written about them. The maximum number of books to be returned is identified by the parameter **limit**

```
db.books.aggregate([
    {$match:
      {reviews: {$ne: []}}
    },
    {$project:
        {_id: 0, title: 1, author: 1, imageURL: 1,
        averageRating: {$avg: "$reviews.rating"}}
    },
    {$sort:
        {averageRating: -1}
    },
    {$limit: limit}
  ])
```

Users that have written the highest number of reviews

In order to understand which are the most active users in our social network, it is useful to detect which are the users that have written more reviews. In order to do that, this query returns a list of usernames ordered by means of the number of reviews they wrote. The maximum number of users to be returned is identified by the parameter **limit**

```
db.users.aggregate([
  {$match:
      {reviews: {$ne: []}}
  },
  {$project:
      {_id: 0, username: 1, numReviews: {$size: "$reviews"}}
  },
  {$sort:
      {numReviews: −1}
  },
  {$limit: limit}
])
```

Number of books per language

For the administrators of the system is important to understand the distribution of languages among the books, in order to understand which new books could be imported. In this way, this query returns how many books are in the database for each language

```
db.books.aggregate([
      {$group : {
          _id:"$language",
          count:
            {$sum:1}
        }},
      {$sort: {count: −1}}
])
```

Best authors by average rating

This query gives a list of authors ordered by the average of the ratings their books obtained. In that way we can assess which are the most liked authors by the users of our social network. The maximum number of authors to be listed is indicated by the parameter **limit**

```
db.books.aggregate([
  {
    $match: {reviews: {$ne: []}}
  },
  {
    $unwind: "$reviews"
  },
  {
    $group: {
      _id: '$author',
      avg_rat: {
        $avg: '$reviews.rating'
      }
    }
  },
  {
    $project: {_id: 0, author: "$_id", avg_rat: 1}
  },
  {
    $sort: {avg_rat: −1}
  },
  {
    $limit: limit
  }
])
```

4.2.4 Indexes structure

The indexes are useful in order to improve the read performance, let's see the cases.

User's Search Index

The **users** collection has the *username* field that can be considered to be indexed. This field is REQUIRED and UNIQUE for each User and it is involved in these queries:

Type	Query
W1	Insert a new *username* at registration time
R1	Check uniqueness of a *username* at registration time
R2	Check *username* at login time
R3	Find a User by *username*
R4	Get the User information by *username*

Assuming that the frequency of **W1** and **R1** is the same, but the **R2** and **R3** are much more frequent because involve the normal usage of the application and not the registration phase. The R4 operation is the most used because It is useful in order to get the information of the logged user when It see its profile or the information of other user profile. So we can assume that could be convenient to make a index for the username field.

```
db.users.find({username: 'Kimberly_Fernandez'}, {_id: 0, password: 0}).explain("
    executionStats")
```

After submitting the above command the results are these:

Before Index	After Index
executionTimeMillis: 82, totalKeysExamined: 0, totalDocsExamined: 115449,	executionTimeMillis: 8, totalKeysExamined: 1, totalDocsExamined: 1,

Using this command to create the index:

```
db.users.createIndex({username:1},{name:'username_index'})
```

With this 1.8MB index the performance of a read is increase by a x10.
The R1 operation is the same, with the exception of doing a count() instead of a find(), but according to the mongoDB manual the count() operation is equivalent to the db.collection.find(query).count() construct. R3 and R4 are exactly the same of R2.

Book's Search Index

The **books** collection has several fields that can be considered to be indexed. This fields are the ones that can be used to retrieve the books in the *Search Page* (title, isbn, category, author, publication year and publisher) and to get the Book Information so the isbn that is also REQUIRED and UNIQUE for a book. So all the Read **Read Operation** will be in the form of "Get a book list given the **X**" with **X** one of these fields. Instead the Write **Write Operation** will be only the Add Book since these information can't be changed. So It is convenient to create indexes on these fields, let's see:

ISBN

```
db.books.find({isbn: 9781840237641}, {_id: 0}).explain("executionStats")
```

Before Index	After Index
executionTimeMillis: 270, totalKeysExamined: 0, totalDocsExamined: 219861,	executionTimeMillis: 1, totalKeysExamined: 1, totalDocsExamined: 1,

```
db.books.createIndex({isbn:1},{name:'isbn_index'})
```

Title

```
db.books.find({title: /inferno/}, {_id: 0, title: 1, author: 1, category: 1,
    imageURL: 1}).explain('executionStats')
```

Before Index	After Index
executionTimeMillis: 350, totalKeysExamined: 0, totalDocsExamined: 219861,	executionTimeMillis: 1, totalKeysExamined: 15, totalDocsExamined: 15,

```
db.books.createIndex({title:1},{name:'title_index'})
```

Category

```
db.books.find({category: 'Erotic Romance'}, {_id: 0, title: 1, author: 1, category:
    1, imageURL: 1}).explain('executionStats')
```

Before Index	After Index
executionTimeMillis: 280, totalKeysExamined: 0, totalDocsExamined: 219861,	executionTimeMillis: 99, totalKeysExamined: 5676, totalDocsExamined: 5676,

```
db.books.createIndex({category:1},{name:'category_index'})
```

Author

```
db.books.find({author: 'Stephen King'}, {_id: 0, title: 1, author: 1, category: 1,
    imageURL: 1}).explain('executionStats')
```

Before Index	After Index
executionTimeMillis: 217, totalKeysExamined: 0, totalDocsExamined: 219861,	executionTimeMillis: 69, totalKeysExamined: 597, totalDocsExamined: 597,

```
db.books.createIndex({author:1},{name:'author_index'})
```

Publication Year

```
db.books.find({publicationYear: '1999'}, {_id: 0, title: 1, author: 1, category: 1,
    imageURL: 1}).explain('executionStats')
```

Before Index	After Index
executionTimeMillis: 190, totalKeysExamined: 0, totalDocsExamined: 219861,	executionTimeMillis: 9, totalKeysExamined: 15, totalDocsExamined: 15,

```
db.books.createIndex({publicationYear:1},{name:'pubyear_index'})
```

Publisher

```
db.books.find({publisher: 'Marvel'}, {_id: 0, title: 1, author: 1, category: 1,
    imageURL: 1}).explain('executionStats')
```

Before Index	After Index
executionTimeMillis: 192, totalKeysExamined: 0, totalDocsExamined: 219861,	executionTimeMillis: 13, totalKeysExamined: 2327, totalDocsExamined: 2327,

```
db.books.createIndex({publicationYear:1},{name:'pubyear_index'})
```

Can be clearly see as the usage of indexes increase the performance of the Read Operation.

Index	Size
author_index	2.4MB
category_index	1.1MB
isbn_index	3.5MB
publisher_index	2.1MB
pubyear_index	1.2MB
title_index	7.0MB

So we decided to keep all these indexes.

4.3 Sharding

To satisfy non-functional requirements especialy **Fast Responsiveness**, we have considered only in theory two possibles sharding methods.

Hashed Sharding

For both collections we could use as partitioning method the *Consistent Hashing*, a special kind of hashing technique that reduces the number of remapping operation when the hash table is resized. The **sharding keys** could be:

- **Users**: field *username*, which is unique among the users.
- **Books**: field *isbn*, which is unique among the books.

Zoned Sharding

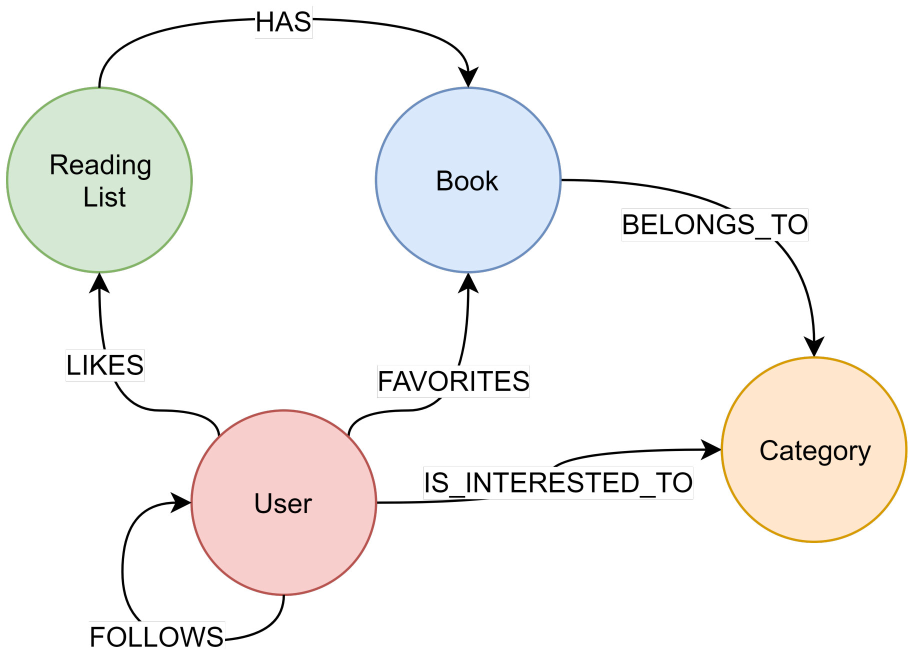
MongoDB offers also this type of sharding in which you can create zones of sharded data based on the shard key. You can associate each zone with one or more shards in the cluster. A shard can associate with any number of zones. In a balanced cluster, MongoDB migrates chunks covered by a zone only to those shards associated with the zone. In our case the **Sharding Key** could be:

- **Users**: a new possible field *country* of the User.
- **Books**: the *language* field of the Book.

These can be useful since we expect that english speaker users will read only english books and so on, and also the Users will followe Users with similar or the same language.

Chapter 5

Design and Implementation of Neo4J



5.1 Nodes

Each node represent an entity of our application, there are 4 types of entity

- **User**: contains *username, name, surname* of a registered user
- **Books**: contains *title, author, isbn, image URL* of a book
- **ReadingList**: contains the *name* of the reading list
- **Category**: contains the *name* of the category

These are all information that are required in order to return satisfying results for suggestion queries

5.2 Relations

- USER **- FOLLOWS ->** USER: this is added when an user starts following another user
- USER **- IS_INTERESTED_TO ->** CATEGORY: this relation is present when a user set its interests to a certain category. The user chooses its interests at registration time
- USER **- LIKES ->** READING_LIST: relation that is added when a user likes a reading list
- USER **- FAVORITES ->** BOOK: relation that is added when a user add a book to its favorite books list
- READING_LIST **- HAS ->** BOOK: relation that is added when a book is added to a certain reading list
- BOOK **- BELONGS_TO ->** CATEGORY: relation that is present between a book and the category to which it belongs

5.3 Queries Implementation

5.3.1 CRUD operations

Create

- **Create** a USER node:

```
CREATE (ee: User {username: <new_username>, name: <name>,
    surname: <surname>})
```

- **Create** a BOOK node:

```
CREATE (ee: Book {title: <new_book_title>, isbn: <book_isbn>,
    author: <book_author>, imageURL: <book_image>})
```

- **Create** a READING_LIST node:

```
CREATE (ee: ReadingList {name: <readingList_name>})
```

- **Create** a CATEGORY node

```
CREATE (ee: Category {name: <category_name>})
```

- **Create** a FOLLOWS relation

```
MATCH (u1: User) WHERE u1.username = <follower>
MATCH (u2: User) WHERE u2.username = <followed>
CREATE (u1) - [:FOLLOWS] -> (u2)
```

- **Create** a IS_INTERESTED_TO relation

```
MATCH (u: User) WHERE u.username = <logged_username>
MATCH (c: Category) WHERE c.name = <category>
CREATE (u) - [:IS_INTERESTED_TO] -> (c)
```

- **Create** a HAS relation

```
MATCH (rl: ReadingList) WHERE rl.name = <rl_name>
MATCH (b: Book) WHERE b.isbn = <book_isbn>
CREATE (rl) - [:HAS] -> (b)
```

- **Create** a LIKE relation

```
MATCH (u: User) WHERE u.username = <logged_username>
MATCH (rl: ReadingList) WHERE rl.name = <rl_name>
CREATE (u) - [:LIKES] -> (rl)
```

- **Create** a FAVORITES relation

```
MATCH (u: User) WHERE u.username = <logged_username>
MATCH (b: Book) WHERE b.isbn = <book_isbn>
CREATE (u) - [:FAVORITES] -> (b)
```

- **Create** a BELONGS_TO relation

```
MATCH (b: Book) WHERE b.isbn = <book_isbn>
MATCH (c: Category) WHERE c.name = <category>
CREATE (b) - [:BELONGS_TO] -> (c)
```

Read

- Retrieve the number of followed user

```
MATCH (:User {username: $username })-[r:FOLLOWS]->()
RETURN COUNT (r) AS numFollowed
```

- Retrieve the number of user's followers

```
MATCH (:User {username: $username })<-[r:FOLLOWS]-()
RETURN COUNT (r) AS numFollowers
```

- Retrieve the number of users that added a book to the favourite books list

```
MATCH (:Book {isbn: $isbn })<-[r:FAVORITES]-()
RETURN COUNT (r) AS numFavorites
```

- Retrieve the number of likes received by a reading list

```
MATCH (:ReadingList {id: $readinglistname })<-[r:LIKES]-()
RETURN COUNT (r) AS numLikes
```

- Retrieve the category of a book by isbn

```
MATCH (c:Category)<-[bt:BELONGS_TO]-(b:Book{isbn: $isbn})
RETURN c.name AS category
```

- Check if a user follows another user

```
MATCH (a:User{id: $usernameA })-[r:FOLLOWS]->(b:User{id: $usernameB })
RETURN COUNT (*)
```

- Check if a user likes a reading list

```
MATCH (a:User{id: $user })-[r:LIKES]->(b:ReadingList{id: $readinglistname })
RETURN COUNT (*)
```

- Check if a user has a book in its favourite books list

```
MATCH (a:User{id: $username })-[r:FAVORITES]->(b:Book{isbn: $isbnbook })
RETURN COUNT (*)
```

Update

- Update the username of a user

```
MATCH (u:User {username: $oldusername })
SET u.username = $newusername ,  u.id = $newusername
```

Delete

- Delete a User node and detach all its relations

```
MATCH (u:User) WHERE u.username = $username DETACH DELETE u
```

- Delete FOLLOWS relation

```
MATCH (:User {username: <logged_username>})-[r:FOLLOWS]->
(:User {username: <target_username>}) DELETE r
```

- Delete IS_INTERESTED_TO relation

```
MATCH (:User {username: <logged_username>})-[r:IS_INTERESTED_TO]->
(:Category {name: <target_category>}) DELETE r
```

- Delete a reading list node

```
MATCH (r:ReadingList {name: <readingList_name>}) DETACH DELETE r
```

- Delete LIKES relation

```
MATCH (:User {username: <logged_username>})-[r:LIKES]->
(:ReadingList {name: <target_rl>}) DELETE r
```

- Delete HAS relation

```
MATCH (:ReadingList {name: <target_rl>})-[r:HAS]->
(:Book {isbn: <target_isbn_book>}) DELETE r
```

- Delete FAVORITES relation

```
MATCH (:User {username: <logged_username>})-[r:FAVORITES]->
(:Book {isbn: <target_isbn_book>}) DELETE r
```

5.3.2 On-Graph Queries

Most followed users

Graph-centric query	**Domain query**
Considering all the User vertices, rank them by the number of ingoing FOLLOWS edges	Rank the Users by means of the number of users that follow them

```
MATCH (u:User)<-[r:FOLLOWS]-(:User)
RETURN DISTINCT u.username AS Username,
COUNT(DISTINCT r) AS numFollower ORDER BY numFollower DESC
LIMIT <limit>
```

Most liked reading lists

Graph-centric query	**Domain query**
Considering all the ReadingList vertices, rank them by the number of ingoing LIKES edges	Rank the reading lists by means of the number of users that liked them

```
MATCH (:User)-[l:LIKES]->(rl:ReadingList)
RETURN DISTINCT rl.name AS ReadingList,
COUNT(l) AS numLikes ORDER BY numLikes DESC
LIMIT <limit>
```

Books that have been added to the favourite books the most

Graph-centric query	**Domain query**
Considering all the Book vertices, rank them by the number of ingoing FAVORITES edges	Rank the books by means of the number of users that add them to the favorite books list

```
MATCH (:User)-[f:FAVORITES]->(b: Book)
RETURN b.title AS title, b.author AS author, b.imageURL AS imageURL,
    b.isbn AS isbn
COUNT(f) AS numFavorites ORDER BY numFavorites DESC
LIMIT <limit>
```

Categories summary taking into account books present in liked reading lists

Graph-centric query	Domain query
Select all the ReadingList vertices. For each vertex, for every ingoing LIKE edge, consider all the Books vertices reached by the HAS edge and count how many times a certain Category vertex is reached by a BELONGS_TO edge	Rank the categories by means of the number of each book belonging to them, contained by a reading list for each like that the list has received

```
MATCH (: User) -[: LIKES]-> (rl: ReadingList) -[h: HAS]-> (: Book)
    -[be: BELONGS_TO]-> (c: Category)
RETURN c.name AS Category,
COUNT(be) AS numTop ORDER BY numTop DESC
LIMIT <limit>
```

Categories summary taking into account books added to the list of favorite books

Graph-centric query	Domain query
Consider all the Book vertices, count for each book how many times is reached by a FAVORITES edges and sum this quantity to the Category that is reached by that book with a BELONGS_TO edge	Rank the categories by means of the number of times a book that belongs to it is added to the favorite books list of a user

```
MATCH (: User) -[: FAVORITES]-> (: Book) -[be: BELONGS_TO]-> (c: Category)
RETURN c.name AS Category,
COUNT(be) AS numFavorites ORDER BY numFavorites DESC
LIMIT <limit>
```

Suggesting users with same interests

Graph-centric query	Domain query
Consider all the User vertices reached by a FOLLOWS edge coming from the logged User edge and exlude them from the result. Now consider all the remaining User vertices reached by FOLLOWS edges coming from all the User vertices reached by FOLLOWS edges coming from the logged User vertex. Return the considered edges that reach with a IS_INTERESTED_TO relation a Category vertex that is reached by the logged User vertex with a IS_INTERESTED_TO relation too	Find users followed by users that the logged user follows, who are interested at least at one same category

```
MATCH (me:User {username: <logged_username>}) -[: FOLLOW]-> ()
  -[: FOLLOW]-> (target: User)
WHERE NOT EXISTS ((me) -[: FOLLOW]-> (target))
  AND (me) -[: IS_INTERESTED_TO]-> (: Category) <-[: IS_INTERESTED_TO]-(target)
  RETURN DISTINCT target.username AS username, target.name AS name,
      target.surname AS surname
  LIMIT <limit>
```

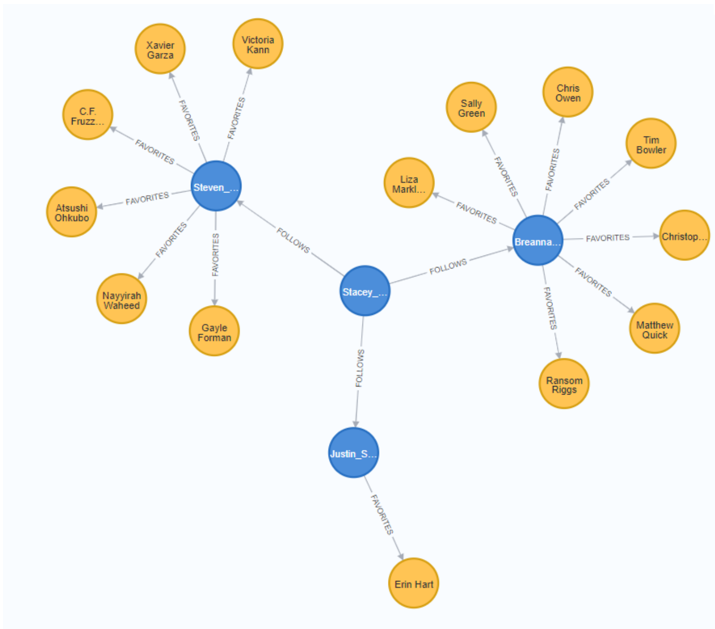
Suggesting books

The result is obtained by merging different partial results

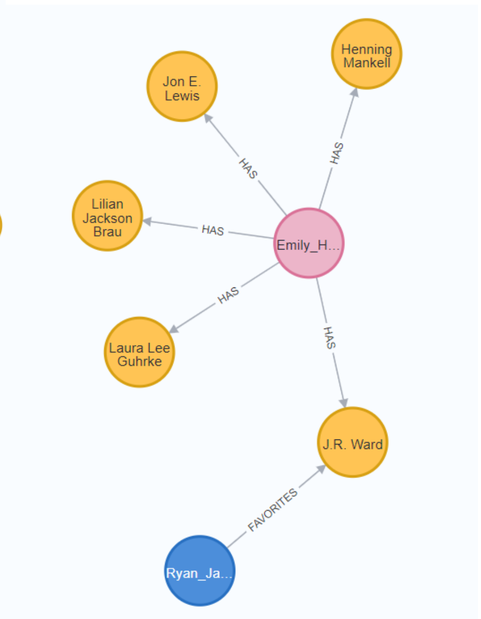
Graph-centric query	Domain query
Exclude all the Books vertices that are reached by the logged User with a FAVORITES edge. Consider the remaining Book vertices reached with a HAS edge by the ReadingList vertices that reaches with a HAS edge also Book vertices linked to the logged User vertex with a FAVORITES edge	Books that are in reading lists in which one or more favorites books of the logged user belong, exluding its favorite books
Consider the logged User vertex and get all the User vertices reached through a FOLLOWS relation. Now for each outgoing FAVORITES edge, return the Book reached in this way excluding all the Books vertices that are reached by the logged User vertex with a FAVORITES edge.	Books that are in the favorite books list of the users followed by the logged User, apart from its favorite Books
Consider the logged User vertex and take all the Category vertices reached through a IS_INTERESTED_TO edge. Return the Book vertices for which the outgoing BELONGS_TO edge reach such Category vertices. Exclude from the result the Book vertices that are reached with a FAVORITES edge by the logged User vertex	Books belonging to the categories the logged user is interested to, apart from the books that are already in its favorite books list
Consider the logged User vertex and all the outgoing FOLLOWS edges, for each of them take the so reached User vertices and take in consideration all the outgoing LIKES edge in order to reach ReadingList vertices. Return the Book vertices reached by one or more of these ReadingList with a HAS edge, excluding all the Books vertices that are reached by the logged User vertex with a FAVORITES edge	Books belonging to reading lists liked by the users followed by the logged User

```
MATCH (u:User{username: $username})-[i:IS_INTERESTED_TO]->(c:Category)
  <-[bt:BELONGS_TO]-(bf:Book)
WHERE NOT EXISTS((u)-[:FAVORITES]->(bf))
RETURN bf.title AS title, bf.author as author, bf.imageURL as imageURL,
  bf.isbn AS isbn, c.name AS name
LIMIT $limit
UNION
MATCH (u:User{username: $username})-[f:FOLLOWS]->(ua:User)-[l:LIKES]->
  (r:ReadingList)-[o:HAS]->(bf:Book)-[bt:BELONGS_TO]->(c:Category)
WHERE NOT EXISTS((u)-[:FAVORITES]->(bf))
RETURN bf.title AS title, bf.author as author, bf.imageURL as imageURL,
  bf.isbn AS isbn, c.name AS name
LIMIT $limit
UNION
MATCH (u:User{username: $username})-[f:FOLLOWS]->(uf:User)-[fav:FAVORITES]->
  (bf:Book)-[bt:BELONGS_TO]->(c:Category)
WHERE NOT EXISTS((u)-[:FAVORITES]->(bf))
RETURN bf.title AS title, bf.author as author, bf.imageURL as imageURL,
  bf.isbn AS isbn, c.name AS name
LIMIT $limit
UNION
MATCH (u:User{username: $username})-[f:FAVORITES]->(b:Book)
  <-[h:HAS]-(r:ReadingList)-[h2:HAS]->(bf:Book)-[bt:BELONGS_TO]->(c:Category)
WHERE NOT EXISTS((u)-[:FAVORITES]->(bf))
RETURN bf.title AS title, bf.author as author, bf.imageURL as imageURL,
  bf.isbn AS isbn, c.name AS name
LIMIT $limit
```

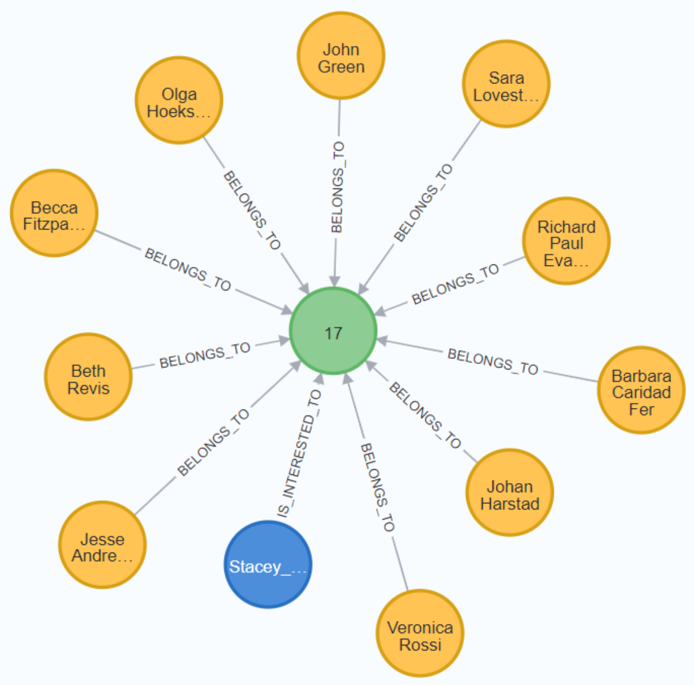
We think can be helpful to see graphically how this suggestion mechanism works, see the pictures.



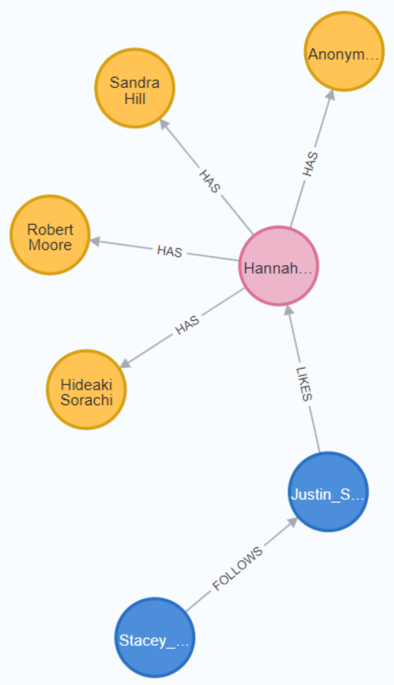
Books Favorites By Your Follows



Books Belonging RL in which your Fav Belongs



Books Belong To Category You Are Interested on



Books Belonging To RL liked By your Follows

Suggesting reading lists

The result is obtained by merging different partial results

Graph-centric query	Domain query
Consider the logged User vertex and all the outgoing FOLLOWS edges trough which other User vertices are reached. For each of them, return all the ReadingList vertices reached with a LIKES edge. Exclude from the result all the ReadingList vertices reached with a LIKES edge by the logged User vertex	Reading lists liked by the users followed by the logged User, excluding the lists liked by it too
Consider the logged User vertex and all the outgoings FOLLOWS edges through which we reach other Users vertices. For each of them, consider all the outgoings FOLLOWS edges that consent to reach a set of User vertices. For each vertex of this set, excluding the ones that are reached by a FOLLOWS edge starting from the logged User vertex, get the ReadingList vertices reached by LIKES edges. Exclude from the result all the ReadingList vertices reached with a LIKES edge by the logged User vertex	Reading lists liked by users that are followed by users followed by the logged User

```
MATCH (u:User{username: $username})-[f:FOLLOWS]->(ua:User)
  -[l:LIKES]->(rl:ReadingList)
WHERE NOT EXISTS((u)-[:LIKES]->(rl))
RETURN rl.name AS name
LIMIT $limit
UNION
MATCH (u:User{username: $username})-[f:FOLLOWS]->(ub:User)
  -[f2:FOLLOW]->(uc:User)-[l:LIKES]->(rl:ReadingList)
WHERE NOT EXISTS((u)-[:FOLLOW]->(uc))
RETURN rl.name AS name
LIMIT $limit
```

5.4 Index

After the creation of the Nodes entities, when we tried to create also the relationship, It became clear that without indexes would be unfeasible in term of time. Indeed we create indexes:

```
CREATE INDEX ON :Book(isbn)
CREATE INDEX ON :Category(name)
CREATE INDEX ON :User(username)
CREATE INDEX ON :ReadingList(name)
```

Chapter 6

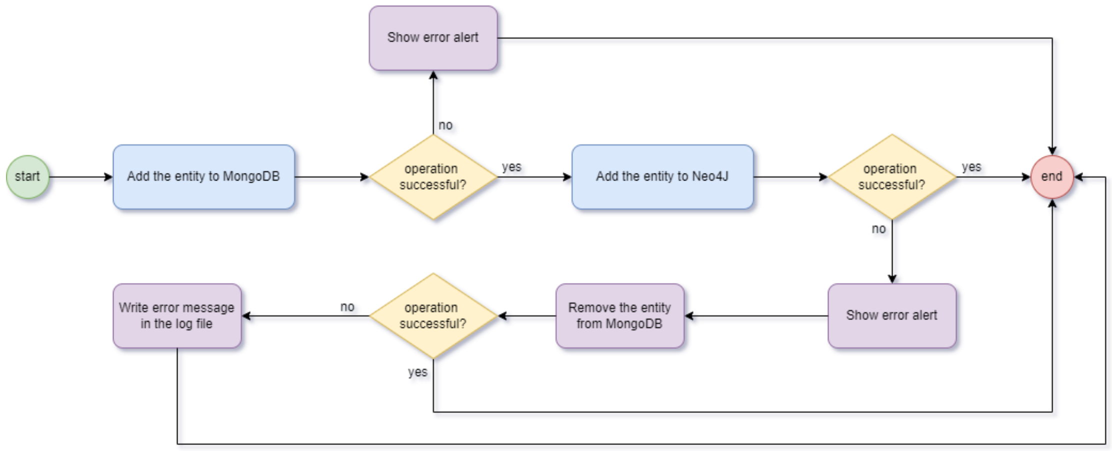
Cross Database Consistency

Given the fact that we have some data that are stored both in MongoDB and in Neo4J DBMSs, we need to guarantee the consistency between the two databases. The operations that must check the cross-database consistency are:

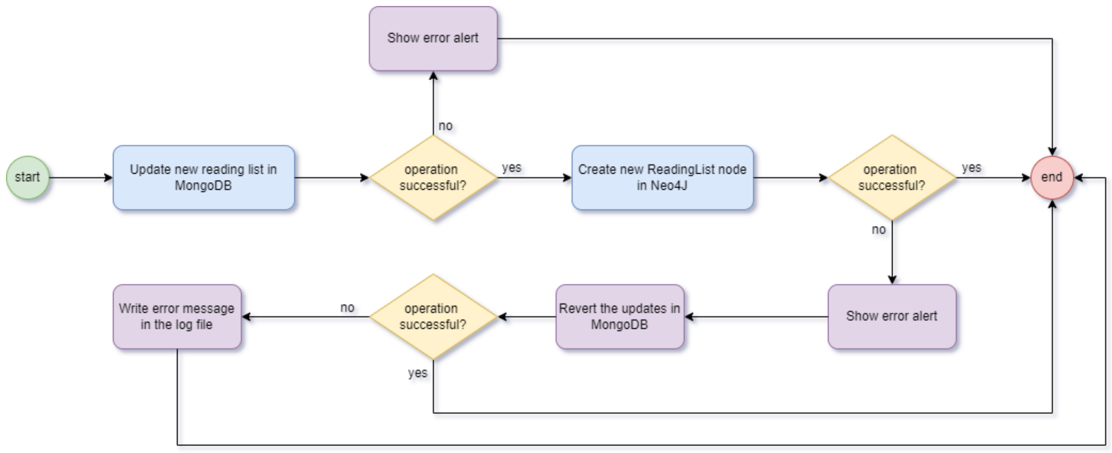
- *Add a book to favourites*
- *Remove a book from favourites*
- *Create a new reading list*
- *Add a book to a reading list*
- *Remove a book from a reading list*
- *Create a new user*
- *Create a new book entity*
- *Add a like to a reading list*
- *Remove a like from a reading list*

For all these operations, we decided to perform first the query in MongoDB, and if it is unsuccessful the system will return an error and an error alert will be showed. Otherwise, the system can proceed to perform the query in Neo4J and in case of success the overall operation is considered completed with success. If, instead, the Neo4J query is unsuccessful, the system must revert the previous MongoDB operation in order to keep consistent the information contained in the two databases. In order to do so, there is always a variable in which the previous copy of the MongoDB entity we are updating is stored, so that we can query the db to restore that value. The following diagrams shows the three summarized behaviours

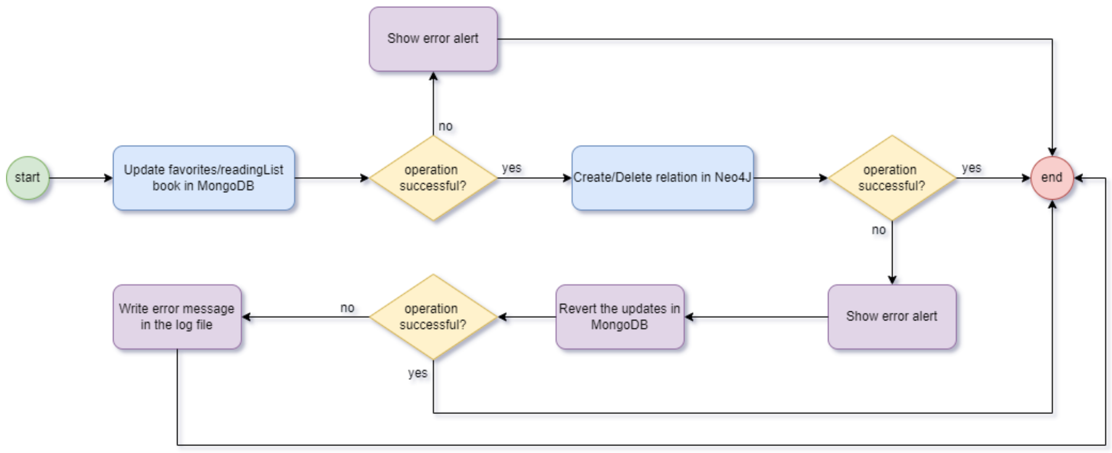
Creating new Entity (User/Book)



Creating new Reading List


start
Update new reading list in MongoDB
operation successful?
no
Show error alert
yes
Create new ReadingList node in Neo4J
operation successful?
yes
end
no
Show error alert
Revert the updates in MongoDB
operation successful?
no
Write error message in the log file
yes

Adding or removing a book to/from favorites/reading list


start
Update favorites/readingList book in MongoDB
operation successful?
no
Show error alert
yes
Create/Delete relation in Neo4J
operation successful?
yes
end
no
Show error alert
Revert the updates in MongoDB
operation successful?
no
Write error message in the log file
yes

Chapter 7

Manual

7.1 User Manual

The User has to register its profile, the page is the same as the login, so if the User is yet registered can directly put its informations in the login portion of the WelcomePage. Instead the registration needs the compilation of personal information fields and also what categories the user prefer, in order to better suggest books and reading lists to the user.

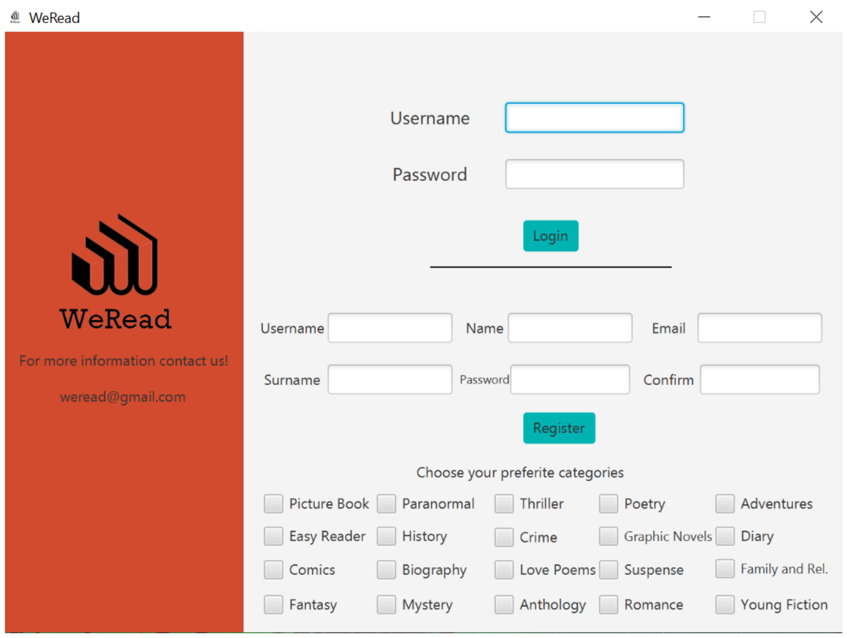
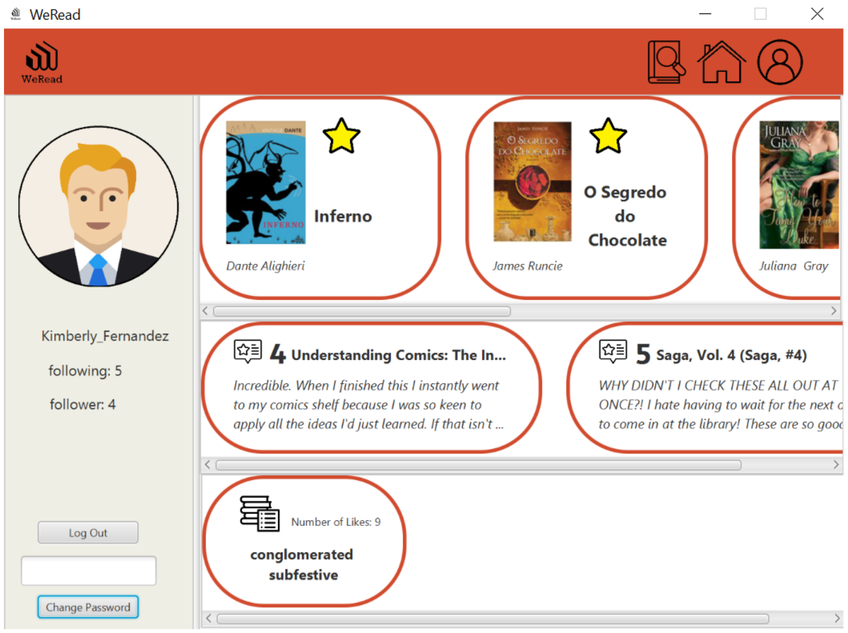


Figure 7.1: Welcome Page

After the login/registration, the User sees its User Page. The information like Username, number of follower and following is depictes as well as the Favorites books, Reviews done and Reading Lists created, if the User wants to see the Books page or Reading List page It can click the correspondent snapshot. The User can also do the Logout and also the password change field and button.



4 Understanding Comics: The In...

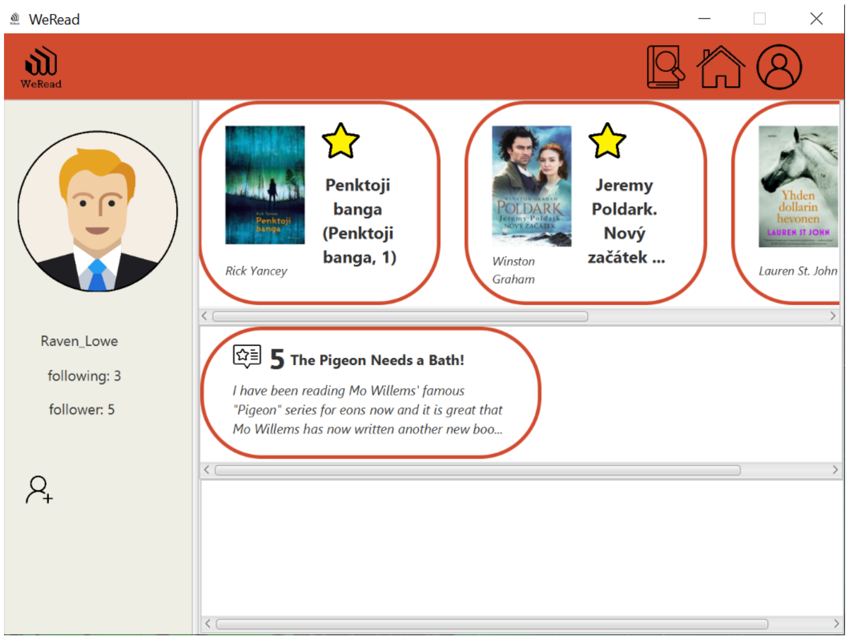
Incredible. When I finished this I instantly went to my comics shelf because I was so keen to apply all the ideas I'd just learned. If that isn't ...

5 Saga, Vol. 4 (Saga, #4)

WHY DIDN'T I CHECK THESE ALL OUT AT ONCE?! I hate having to wait for the next c to come in at the library! These are so good

Figure 7.2: User Page

In the case the User visit other User profiles, It can follow/unfollow clicking the correspondent buttons.



5 The Pigeon Needs a Bath!

I have been reading Mo Willems' famous "Pigeon" series for eons now and it is great that Mo Willems has now written another new boo...

Figure 7.3: User Page other User

The User can change the Page also using the Bar in the upper part of the window, clicking on the **Book with magnifying glass** the User goes to the Search Page, clicking on the **Home** the User goes to the Homepage and clicking on the stylized man to go to its User page.

The Homepage shows the "Discover New" feature, in which the User can discover Friends, Books and Reading lists based on its follows, favorites books, favorites categories and so on.

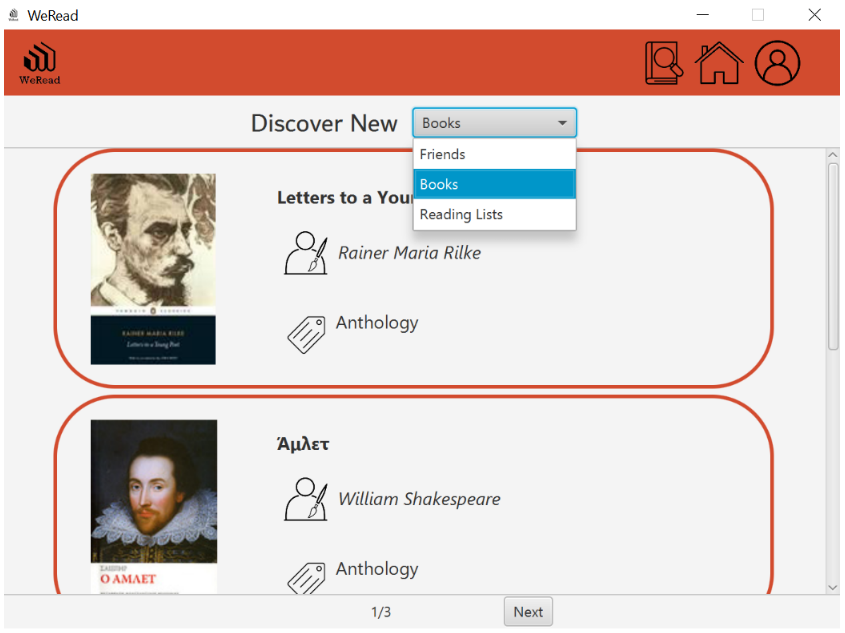


Figure 7.4: Homepage

The Search Page presents a bar in which the User can write the text to search Users and Books, in particular the books can be searched by Title, Category, Author, Publisher, Publication Year. The User can press Previous, Next button in order to see books and users.

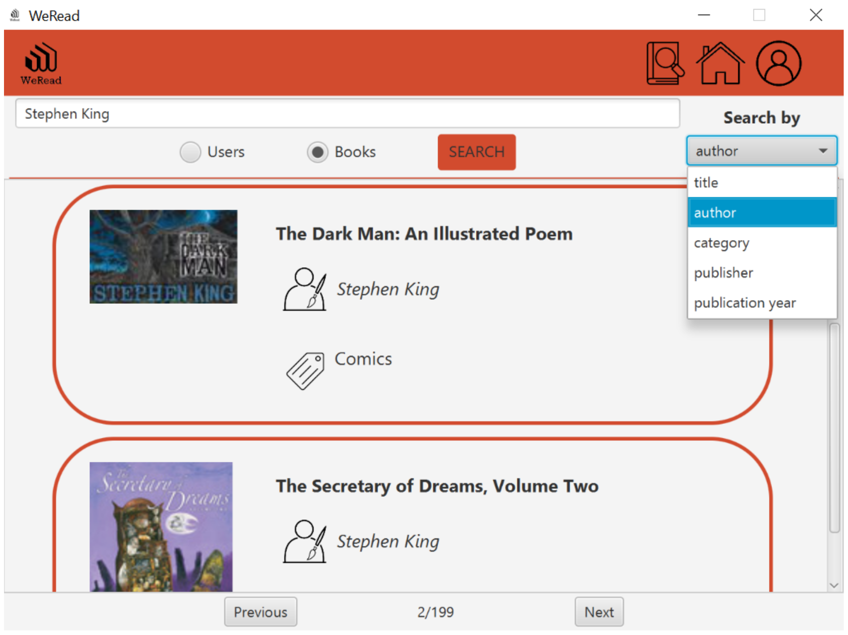
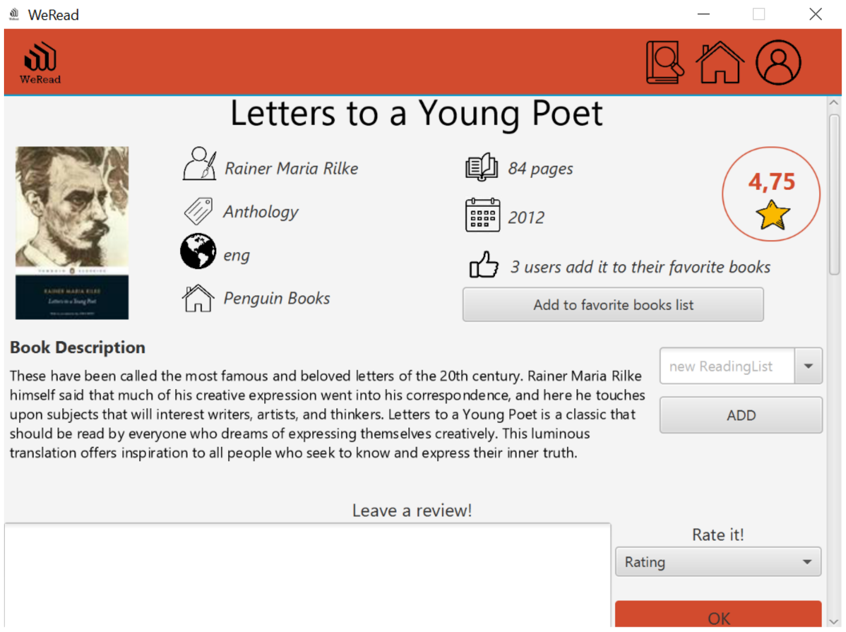


Figure 7.5: Search Page

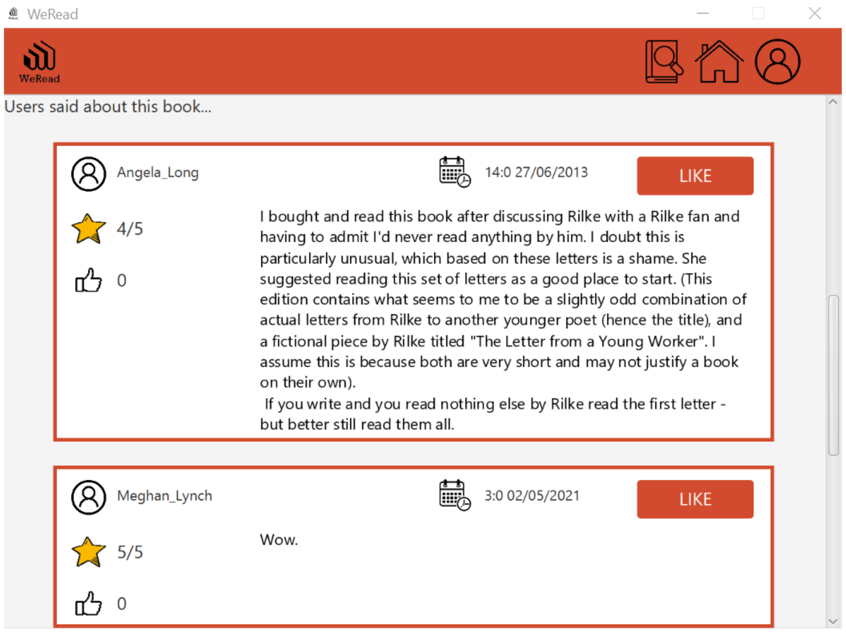
Let's see the remaining Pages, the Book Page presents all the information of the Book like category, publisher, publication year, author, language, the number of user that put this book in their favorites books, the average rating of the reviews received. The User can add this book to its favorites book list and also add to existing reading list previously created or creating new one. User can also rate the book and leave a review.



These have been called the most famous and beloved letters of the 20th century. Rainer Maria Rilke himself said that much of his creative expression went into his correspondence, and here he touches upon subjects that will interest writers, artists, and thinkers. Letters to a Young Poet is a classic that should be read by everyone who dreams of expressing themselves creatively. This luminous translation offers inspiration to all people who seek to know and express their inner truth.

Figure 7.6: Book Page

Below in the Page, there are the reviews made by other users that can be liked, clicking to the User icon in the review, the User can visit the Profile of the Reviewer.



Users said about this book...

Angela_Long 14:0 27/06/2013 LIKE

4/5

0

I bought and read this book after discussing Rilke with a Rilke fan and having to admit I'd never read anything by him. I doubt this is particularly unusual, which based on these letters is a shame. She suggested reading this set of letters as a good place to start. (This edition contains what seems to me to be a slightly odd combination of actual letters from Rilke to another younger poet (hence the title), and a fictional piece by Rilke titled "The Letter from a Young Worker". I assume this is because both are very short and may not justify a book on their own).
If you write and you read nothing else by Rilke read the first letter - but better still read them all.

Meghan_Lynch 3:0 02/05/2021 LIKE

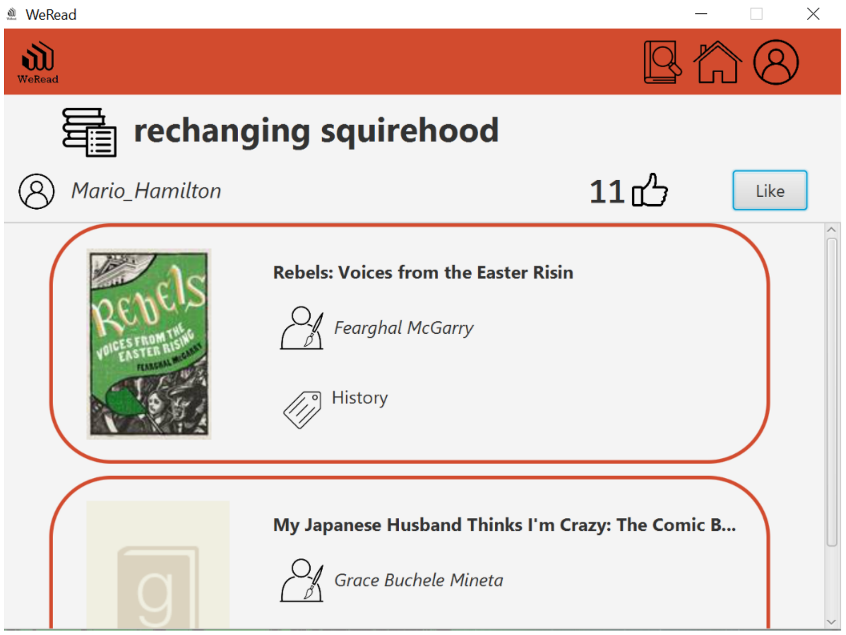
5/5

0

Wow.

Figure 7.7: Book Page Reviews

The Reading List page shows the books belonging to it, the creator, the name, the number of likes. The User can like It or in the case the Reading List is owned by the User, can delete It.



rechanging squirehood

Mario_Hamilton 11 Like

Rebels: Voices from the Easter Risin

Fearghal McGarry

History

My Japanese Husband Thinks I'm Crazy: The Comic B...

Grace Buchele Mineta

Figure 7.8: Reading List Page

7.2 Administrator Manual

The Administrator login normally but the page shown after the login is this one:

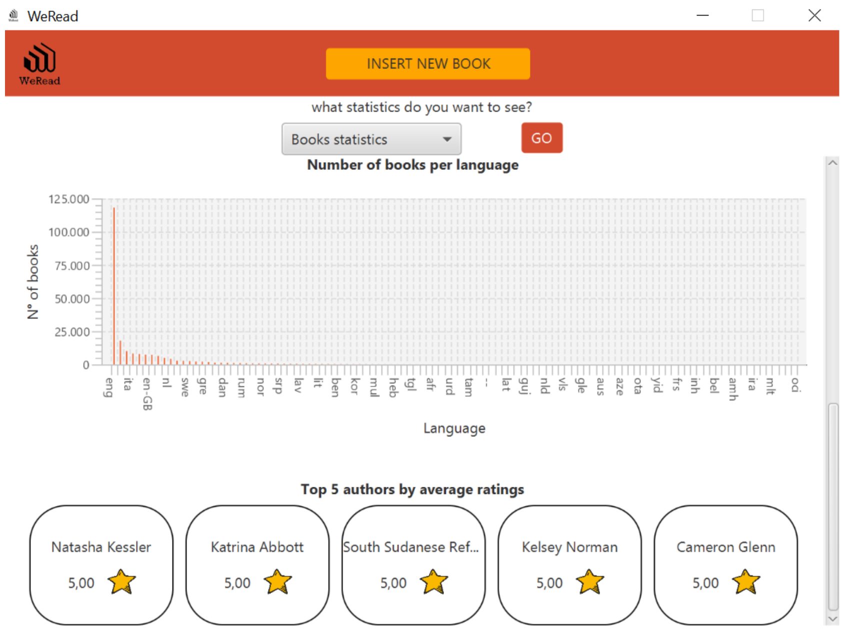


Figure 7.9: Books Statistics Page

The Administrator can press the button in yellow to change the page and go to the Add New Book page or change the statistics to be shown in this page, with the Users one. Some analytics are shown in this page.

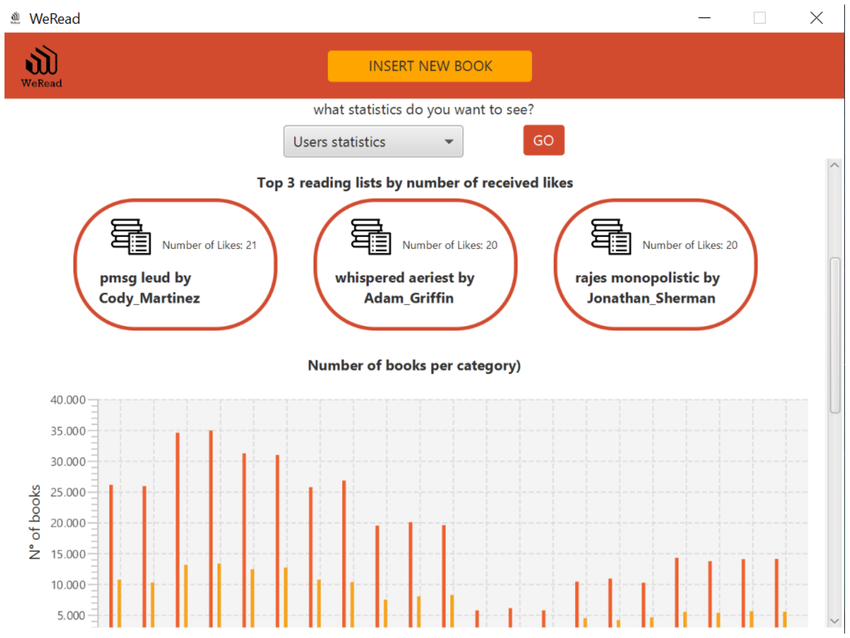

WeRead
INSERT NEW BOOK
what statistics do you want to see?
Users statistics
GO
Top 3 reading lists by number of received likes
Number of Likes: 21
pmsg leud by Cody_Martinez
Number of Likes: 20
whispered aeriest by Adam_Griffin
Number of Likes: 20
rajes monopolistic by Jonathan_Sherman
Number of books per category)
N° of books

Figure 7.10: Users Statistics Page

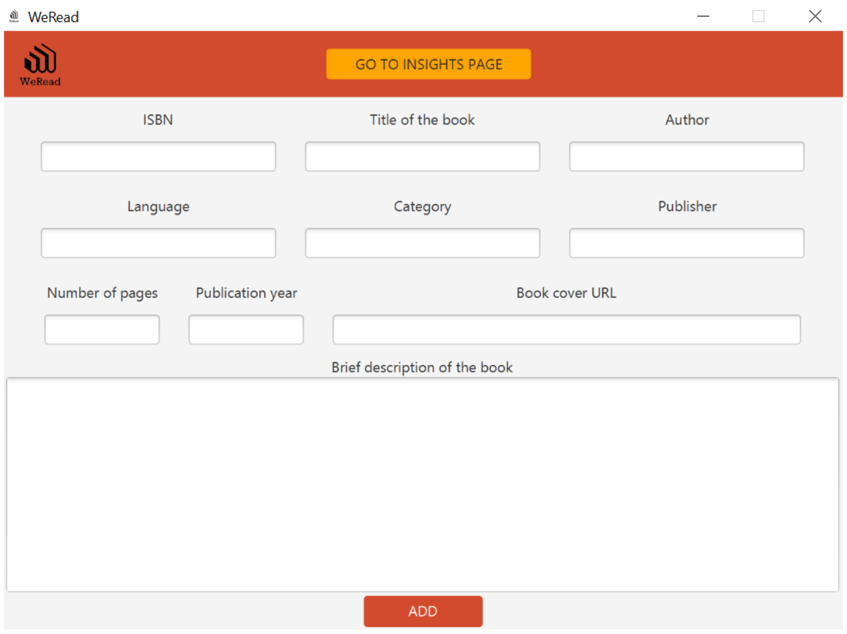


Figure 7.11: Insert New Book Page