Large-Scale and Multi-Structured Databases Project

BookAdvisor

Members

Daniele Giaquinta, Francesco Londretti, Giulio Bello

Content

1 Introduction and Requirements

The aim of this project is to develop a database suitable for managing a social platform for book reviews. This platform should provide readers with the ability to share their opinions and read others' opinions, which can help them choose new books to read. The databases utilized for this project are MongoDB and Neo4J. MongoDB is used to store three collections: *users*, *book*, and *review*, while Neo4J is exploited to help with the more social functionalities. Users will be able to search for specific books, view reviews, publish their own reviews, upvote and downvote others' reviews. Additionally, users can check other users' profiles to access social functionalities such as viewing a user's most helpful reviews, following other users, and receiving personalized book recommendations based on their reviews, recommendations of other users to follow with similar tastes in books, and much more. A simple and bare-bone GUI has been developed with the sole purpose of allowing interaction with the databases and their functionalities.

1.1 Functional Requirements and UML Use-Case

There are three types of users: *reviewer*, *author* and *admin*. Each of these users is of course a signed user; the basic functionalities that the different type of users must be able to use are:

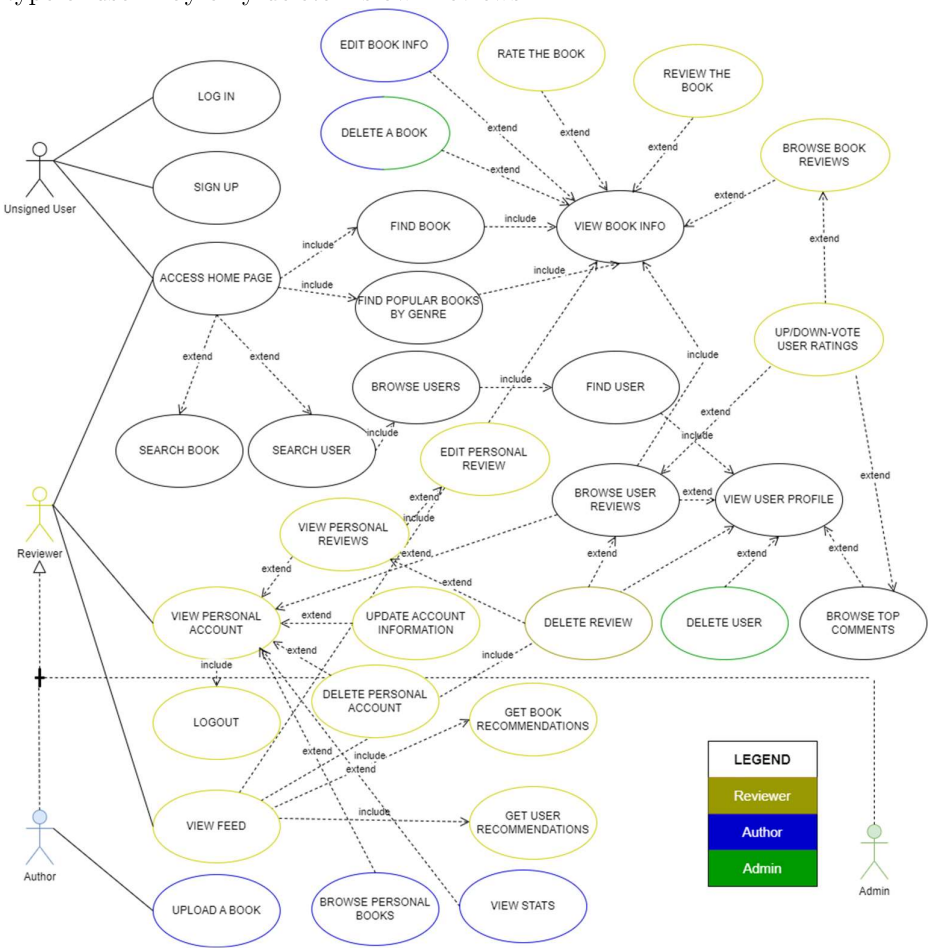
Unsigned User
- Log in
- Sign up
- Browse books
- Search book
- Find book
  - View book info

Author (inherits from Reviewer)
- All actions of a Reviewer
- Upload a book
- Delete a book
- View Stats

Admin (inherits from Reviewer)
- All actions of a Reviewer
- Delete any book
- Delete any user
- Delete any review

Reviewer
- All actions of a Signed User
- View personal reviews
  - Edit personal review
  - Delete review
- In view book info:
  - Rate the book
  - Review the book
  - Browse book reviews
    - Upvote/Downvote user ratings
- View personal account
  - Update account information
  - Delete personal account
  - View personal reviews
    - Edit personal review
- Search user
- Browse users
  - Find user
    - View user profile
      - Browse user reviews
      - Follow user
- Access feed
  - Get book recommendations
  - Get user recommendations

Note that this list is not specific, and some constraints may apply; for example, an admin may delete reviews regardless of who wrote them, while any other type of user may only delete his own reviews.



1.2 Non-Functional Requirements

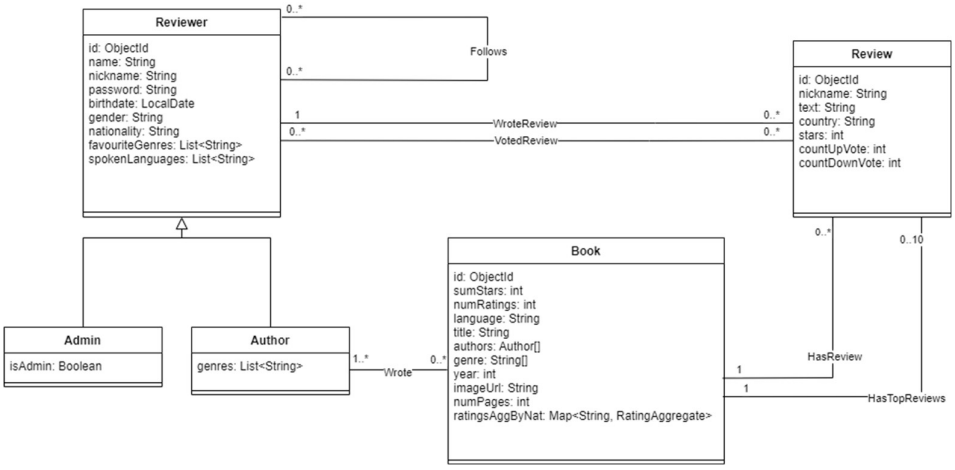
The application's structure is based on the concept of a social network where authors can share their books to the reviewers. The following points are essential for delivering an efficient and enjoyable service:

- The system must be fast-responsive.
- The system must ensure high availability of content.
- The system must be fault-tolerant, so that the failure of a node does not compromise the service.
- The system must be user-friendly and intuitive to use.
- The system must be available 24/7.

1.3 UML Class Diagram

Our application defines three main entities that can be later interpreted as collections for our document-based database:

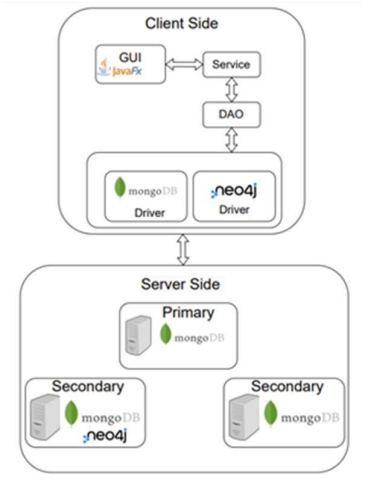
- User
- Book
- Review



2 Architecture

2.1 Software Architecture

The software architecture follows the DAO – Service -Model paradigm, its modules are:



Recipe Share was developed as a JavaFX application. The code was organized in packages, more specifically the packages are:

- Dao
- App
- Model
- Service

The Service module also contains all the functions which manage persistence among the two different types of databases.

2.1.1 App Package

This package contains the classes that implements the graphic interface and the events which handle the interactions of the user with application.

2.1.2 DAO Package

This package is split into three sub-packages:

- documentDB: it contains the classes that perform the queries to MongoDB.
- graphDB: it contains the classes that perform the queries to Neo4j.
- The MongoDB and Neo4J conncetors and drivers.

2.1.3 Service Package

It has the purpose to prevent direct access to the Databases from the Business logic, basically the classis in this package stands in the middle between the business classes and the Dao classes. In this package various checks make sure that data is coherent between MongoDB and Neo4J every time an attempt at modification is made.

2.1.4 Model Package

It contains the classes that represent the main entities of our application:

- Reviewer
- Author
- Admin
- Book
- Review
- Follow

2.1.5 Procedures File

For didactic purposes, the "complex" queries that must be implemented as indicated by project instructions (3 queries in MongoDB and 2 in Neo4J) are contained in this file.

2.2 Dataset

The application revolves around books, therefore the raw data used for the application come from datasets characterized by information of books already published online with reviews as well as reviewer and author profiles. The dataset is downloadable at:

- https://datarepo.eng.ucsd.edu/mcauley_group/gdrive/goodreads/UCSD%20Book%20Graph.html

We had to perform various preprocessing tasks, mainly carried out with Python, in order to obtain data in a form fitting for our task (data integration, augmentation, generation, reordering the json fields, cutting it due to size, etc...). All the redundancies as well as document linking and embedding were inserted through these Python scripts; in a functioning state such things are of course managed by the application and there is no need of external intervention.

2.3 Database Choices

Two different types of databases were used in the making of the application: MongoDB and Neo4j. Here below more details and explanations regarding these choices:

- **MongoDB**: it was chosen because in the stored data it allows the use of Indexes in order to retrieve information quickly: given that our application was made with the intent of being fast-responsive, meaning that we want reading operations to be fast, the indexes can help us in this aspect. Also, as it'll be shown later, some reviews of the books were stored as embedded documents in the books collection, so the embedding feature of MongoDB was also important in the realization of the service, as well as the document linking feature which was used in the books collection the review ids and in the users collection for the review ids, written book ids, ids of upvoted and downvoted reviews.
- **Neo4j**: It was fundamental for the realization of another social-network part of our application: authors that can follow each other. This aspect is also taken into account in the analytic section, in this sense a Graph Database allows us to perform this kind of queries in an optimal way, exploiting the relationship properties between nodes.

Neo4j only stores very few information about users and books only as will be explained in detail in a later section.

3 MongoDB Design and Implementation

In this section all the collections will be briefly discussed and later all the details about the various CRUD operations, queries and analytics will be given.

3.1 Users Collection

Remember that every user is a Reviewer automatically; a Reviewer can also be an Admin but there is no way to register as Admin through the application for safety reasons, an Admin entry must be manually added to MongoDB. Another expansion of the Reviewer is the Author; one user automatically registers as Authors if he fills up the "Genre" field when registering, such field contains the genres of the books the author is supposed to mainly write. We decided to link all the upvoted and downvoted reviews even though this is not a traditional linking since such field is not used to access the reviews collection. Here a sample of an Author sample:


```
{
  "_id": {
    "$oid": "60de18800000000000001510"
  },
  "name": "Louis Luna",
  "nickname": "kevin15",
  "birth": "1953-12-29",
  "nationality": "Puerto Rico",
  "spoken_languages": [
    "en-US",
    "en-GB",
    "eng",
    "spa",
    "en-CA"
  ],
  "gender": "M",
  "password": "974db8d0ec12c332b39a3c563edf06b3caa6f9b62ac3f29ae439420727bcec0d",
  "up_voted_reviews": [],
  "down_voted_reviews": [],
  "favourite_genres": [
    "comics",
```

```
    "graphic",
    "history",
    "historical fiction",
    "biography",
    "mystery",
    "thriller",
    "crime"
  ],
  "genres": [
    "non-fiction",
    "fiction"
  ]
}
```


3.2 Books Collection

The book element contains a lot fields. It is worth noting that this collection too supports linking of the reviews of each book by id. There is also a partial embedding of reviews; each book contains 10 full reviews embedded, i.e. the most relevant ones (only if present), where relevant means with the highest (upvote – downvote) count. Such selection of reviews is of course dynamic and for this reason we decided to update all the book collection in terms of most relevant reviews once a week.

These reviews are also used to instantly display some reviews when opening a book without the need to access the review collection, which can still be easily managed thanks to the review linking by id.

Another interesting information contained in book is an aggregate which for each country lists the number of reviews and the total rating; this is very useful to easily perform analytics by country and apart from some spatial cost, the computational cost to maintain this information updated is non existent as every time a review is added the book can be updated as well. A sample of book with only one review for readability:


```
{
        "_id": {
            "$oid": "60de1880000000000000053c"
        },
        "language": "eng",
        "title": "Until I Find You",
        "image_url": "https://s.gr-assets.com/assets/nophoto/book/111x148-
bcc042a9c91a29c1d680899eff700a03.png",
        "num_pages": 928,
        "year": 2006,
        "genre": [
```

```
        "fiction"
    ],
    "ratings_agg_by_nat": {
        "Austria": {
            "cardinality": 1,
            "sum_stars": 4
        }
    },
    "most_10_useful_reviews": [
        {
            "_id": {
                "$oid": "666c8513ffaccedfd624582c"
            },
            "user_id": {
                "$oid": "60de188000000000000bc499"
            },
            "book_id": {
                "$oid": "60de1880000000000000053c"
            },
            "rating": 4,
            "date_added": "2008-04-10 19:18:42",
            "review_text": "What can I say? I just love the way he writes.",
            "count_up_votes": 594,
            "count_down_votes": 268,
            "nickname": "elizabeth90",
            "country": "Austria"
        }
    ],
    "review_ids": [
        {
            "$oid": "666c8513ffaccedfd624582c"
        }
    ],
    "authors": [
        {
            "id": {
                "$oid": "60de18800000000000000c03"
            },
            "name": "Carol Smith"
        }
    ],
    "numRatings": 1,
    "sumStars": 4
```

```
    }
```


3.3 Reviews Collection

The review collection does not have anything particularly worth noting:


```
{
        "_id": {
            "$oid": "666c83a8ffaccedfd61229fb"
        },
        "user_id": {
            "$oid": "60de1880000000000000a6da"
        },
        "book_id": {
            "$oid": "60de1880000000000009f621"
        },
        "rating": 4,
        "date_added": "2013-12-27 12:20:41",
        "review_text": "A compelling drama that explores the depths of the human
psyche. The characters' development is both realistic and profound.",
        "count_up_votes": 912,
        "count_down_votes": 807,
        "nickname": "joanna56",
        "country": "Norfolk Island"
    }
```


3.4 CRUD Operations (DAO)

Create:

- addBook(Book book)
- addBook(ObjectId bookId, Book book)
- addReviewToBook(ObjectId bookId, ObjectId reviewId)
- addMostUsefulReview(ObjectId bookId, Document review)
- addReview(Review review)
- addReview(ObjectId reviewId, Review review)
- addUser(User user)
- addUser(ObjectId userId, User user)
- createUserFromDocument(Document doc)
- createReviewerFromDocument(Document doc)
- createAuthorFromDocument(Document doc)

Read:

- findBookById(ObjectId id)
- findBooksByTitle(String title)
- getBookById(ObjectId id)
- getBooksByTitle(String title)
- getBooksByGenre(String genre)
- getBooksByYear(int year)
- getBooksByLanguage(String language)
- getBooksByRating(double targetRating, boolean greaterOrEqual)
- getBooksByNumPages(int numPages, boolean greaterOrEqual)
- getBooksByAuthor(ObjectId authorId)
- getBooksByGenres(List<String> genres, boolean isAnd)
- getAllBooks()
- findReviewById(ObjectId id)
- findReviewsByBookId(ObjectId bookId)
- findReviewsByUserId(ObjectId userId)
- findReviewByUserIdAndBookId(ObjectId userId, ObjectId bookId)
- findReviewsByBookName(String bookName)
- findReviewsByUsername(String username)
- findReviewByUsernameAndBookName(String username, String bookName)
- findReviewsByStars(int stars)
- findReviewsByStarsAndBookName(int stars, String bookName)
- findReviewsByStarsAndUsername(int stars, String username)
- findUserById(ObjectId id)
- findReviewerById(ObjectId id)
- findAuthorById(ObjectId id)
- findUsersByUsername(String username)
- getReviewsByUserId(ObjectId userId)
- findUserByUsername(String username)
- listAllUsers()

Update:

- updateBook(ObjectId bookId, Document book)
- updateBookRating(ObjectId bookId, int rating, String nationality)
- updateReview(Review review)
- updateVoteCount(ObjectId reviewId, String voteType, int count)
- updateUser(User user)
- voteForReview(Reviewer user, ObjectId reviewId, boolean vote, ReviewDao reviewDao)

Delete:

- deleteBook(ObjectId id)
- removeReviewFromBook(ObjectId bookId, ObjectId reviewId)
- deleteReview(ObjectId id)
- deleteReviewsByUserId(ObjectId userId)

- deleteUser(ObjectId id)
- addReview(ObjectId userId, ObjectId reviewId)
- removeReview(ObjectId userId, ObjectId reviewId)

3.5 Replication

Our application emphasizes read operations over write operations. To optimize database access, we have implemented pagination, allowing only a limited number of documents (10) to be displayed at a time for queries that yield a large number of results. In consideration of our non-functional requirements, particularly high availability and fast responsiveness, we have set the following configurations:

- Write Concern: W1
- Read Preference: nearest

Setting the write concern to W1 ensures that the write operations are expedited by writing to the primary node and eventually updating secondary nodes, adhering to eventual consistency. This decision reflects our emphasis on Availability and Partition Tolerance from the CAP Theorem, prioritizing swift write over strict consistency.

For read operations, we have opted for the "nearest" read preference to minimize latency. This directs read queries to the node with the lowest network latency, optimizing response times and fulfilling our requirement for rapid responsiveness.

3.6 Analytics

In this section the three heavier analytics chosen for the project will be analysed. The first one returns useful statistics for an Author, the second one returns famous books given a genre and divided by country to get more customized recommendations, and the last one returns a Reviewer's most useful reviews in descending order of "usefulness". The code of these queries is consultable below in MongoDB language, it has however been adapted to java in the Procedures.java file.

These analytics only use the books collection, as it has been filled with various **redundancies and/or additional aggregate information** easily maintainable in order to facilitate specifically these analyses as well as other queries.

For example, each book contains the top 10 useful reviews, i.e. reviews relative to the book with the highest count of (up votes − down votes). This piece of data allows to perform the third analytic easily, as it is easy to find a user's useful reviews if those reviews are completely stored in the book. Furthermore, this field is also used to output some reviews of a book when opening its information window without the need for a query to the review collection.

Same reasoning applies for the rating aggregated by nationality, which can be used for both the analytic for the Author and to search for famous books given a genre for each country (first and second analytic).

3.6.1 Author Statistics

This analytic can be invoked by each Author w.r.t his own books. Once an Author opens his profile tab during the execution of the app, he can press the button "View stats" which opens a window containing **for each of his books, the number of reviews, the average rating and also the number of reviews and average rating in each country** to better understand the total appreciation of his books as well as the appreciation biased by nationality of the readers.

```
db.books.aggregate([
    // Filter books by the specified author's ObjectId
    {
        $match: {
            author: authorId
        }
    },
    // Add the field averageRating in the output as division of the two
    {
        $addFields: {
            averageRating: { $divide: ["$sumStars", "$numRatings"] }
        }
    },
    // Transform the ratings aggregation by nation into a more usable array format
    {
        $addFields: {
            countryRatings: {
                // Put inside the field countryRatings the name of the country,
mean score and the number of reviews
                $map: {
                    input: { $objectToArray: "$ratingsAggByNat" },
                    as: "rating",
                    // These 3 are the fields that compose countryRatings
                    in: {
                        country: "$$rating.k",
                        data: {
                            averageRating: { $divide: ["$$rating.v.sumRating",
"$$rating.v.cardinality"] },
                            numRatings: "$$rating.v.cardinality"
                        }
                    }
```

```
            }
        }
    }
},
// Use $reduce to reformat country ratings into a more detailed array
{
    $addFields: {
        detailedCountryRatings: {
            $reduce: {
                input: "$countryRatings",
                initialValue: [],
                in: {
                    $concatArrays: ["$$value", [
                        {
                            country: "$$this.country",
                            averageRating: "$$this.data.averageRating",
                            numRatings: "$$this.data.numRatings"
                        }
                    ]]
                }
            }
        }
    }
},
// Project final document structure without additionalInfo
{
    $project: {
        bookTitle: "$title",
        bookRating: "$averageRating",
        bookTotalRatings: "$numRatings",
        bookCountryDetails: {
            $map: {
                input: "$detailedCountryRatings",
                as: "countryDetail",
                in: {
                    country: "$$countryDetail.country",
                    averageRating: {
                        $round: ["$$countryDetail.averageRating", 2]
                    },
                    numRatings: "$$countryDetail.numRatings"
                }
            }
        }
```

```
        }
    },
    // Sorting by title
    {
        $sort: { "bookTitle": 1 }
    }
]);
```

3.6.2 Most Famous Books by Genre and Country

This analytic can be accessed at the home page of the app, it allows the user to select a specific genre and the query will return the most famous book (by number of reviews) in each country for the given genre. This is useful to understand what the preferences are in terms of a specific genre in each country, as cultural differences may result in different results.

```
          db.books.aggregate([
  // Match the books with the given genre
  {
    $match: {
      genre: "Adventure" // Replace "Adventure" with the genre input
    }
  },
  // Project necessary fields and initialize aggregation structures
  {
    $project: {
      title: 1,
      ratingsAggByNat: 1,
      countryReviews: {
        $map: {
          input: { $objectToArray: "$ratingsAggByNat" },
          as: "country",
          in: {
            country: "$$country.k",
            numReviews: "$$country.v.cardinality",
            bookTitle: "$title",
            bookId: "$_id"
          }
        }
      }
    }
  },
  // Deconstruct countryReviews array into individual documents
  {
```

```
    $unwind: "$countryReviews"
  },
  // Group by country and find the book with the most reviews for each country
  {
    $group: {
      _id: "$countryReviews.country",
      mostFamousBook: {
        $first: {
          title: "$countryReviews.bookTitle",
          bookId: "$countryReviews.bookId",
          numReviews: "$countryReviews.numReviews"
        }
      },
      maxNumReviews: { $max: "$countryReviews.numReviews" }
    }
  },
  // Project the final result
  {
    $project: {
      _id: 0,
      country: "$_id",
      title: "$mostFamousBook.title",
      bookId: "$mostFamousBook.bookId",
      numReviews: "$mostFamousBook.numReviews"
    }
  }
]);
```

3.6.3 Find a Reviewer's Top Comments

This last analytic can be invoked when visiting a user's profile. The button "View top useful reviews" opens a new window containing all the user's top comments, i.e. the ones that appear as the most useful comments in the various books, if any.

```
db.books.aggregate([
  // Match books that contain reviews by the given username
  {
    $match: {
      "most10UsefulReviews.userName": "AliceJones89"
    }
  },

  // Add a new field that calculates the usefulness of each review
```

```
{
  $addFields: {
    "most10UsefulReviews": {
      $map: {
        input: "$most10UsefulReviews",
        as: "review",
        in: {
          $mergeObjects: [
            "$$review",
            {
              usefulness: {
                $subtract: ["$$review.countUpVote", "$$review.countDownVote"]
              }
            }
          ]
        }
      }
    }
  }
},

// Filter reviews to only include those by the given username
{
  $addFields: {
    "most10UsefulReviews": {
      $filter: {
        input: "$most10UsefulReviews",
        as: "review",
        cond: { $eq: ["$$review.userName", "AliceJones89"] }
      }
    }
  }
},

// Sort the reviews by usefulness in descending order
{
  $addFields: {
    "most10UsefulReviews": {
      $slice: [
        {
          $sortArray: {
            input: "$most10UsefulReviews",
            sortBy: { usefulness: -1 }
```

```
            }
          },
          3
        ]
      }
    }
  },

  // Project the desired fields
  {
    $project: {
      _id: 0,
      bookTitle: "$title",
      author: 1,
      language: 1,
      year: 1,
      genre: 1,
      imageUrl: 1,
      numPages: 1,
      mostUsefulReviews: "$most10UsefulReviews"
    }
  }
]).pretty()
```

3.7 Indexes

In order to improve the efficiency of read operations from the database, indexes were implemented. The details are shown below.

3.7.1 Index on Title to Aid the Search

The first idea that came to our mind was to implement some index on the title to speed up the title-based search operations of books from the home tab, as it is probably the most frequent database operation in a real-world scenario. We tried two different index approaches, both on the title, one was the text index and the other is the standard index.

We concluded that **this index was not advantageous** for our database, and we looked elsewhere to increase our performance. The lack of gain in terms of performance is probably due to the fact that book search usually happens with words included in the book title, now strictly the first word.

3.7.2 Index on Author in the Books Collection

Another index idea is implementing an index on the author id in the books collection. This can be useful when an author tries to open his statistics and when showing all the books of an author. The test was performed on the second task using user "edward39".

Scenario	Stats
No index	executionTimeMillis: 24, totalKeysExamined: 0, totalDocsExamined: 8680
`authors.id:1`	executionTimeMillis: 6, totalKeysExamined: 6, totalDocsExamined: 6

We decided to keep this index.

3.7.3 Genre Index in Books Collection

We also decided to try and speed up our second heavy MongoDB query, the one that takes a genre as input and finds the most famous book for that genre in each country. We tested the index using the fiction genre.

Scenario	Stats
No index	executionTimeMillis: 104, totalKeysExamined: 0, totalDocsExamined: 8680
`genre:1`	executionTimeMillis: 85 totalKeysExamined: 6995, totalDocsExamined: 6995

We decided to keep this index.

3.7.4 Genre Index in Reviews Collection

We also decided to try and speed up the review browsing when opening the reviews of a user. We tested it with the reviews of "edward39".

Scenario	Stats
No index	executionTimeMillis: 165, totalKeysExamined: 0, totalDocsExamined: 99026
`user_id:1`	executionTimeMillis: 2, totalKeysExamined: 6, totalDocsExamined: 6

We decided to keep this index.

4 Neo4J Design and Implementation

Here's the Graph Database structure:

4.1 Composition

4.1.1 Nodes

In our implementation, there are two types of nodes:

- **User**: Composed by "id", "nickname" and "fav_genres";
- **Book**: Composed by "id", "title" and "language".

Those attributes will be used in the queries explained further below.

4.1.2 Relations

- **FOLLOWS**: User –FOLLOWS-> User: this is the classic follow relationship; it is added whenever a User follows another User, and it is removed whenever a User stops following another User;
- **RATES**: User –RATES-> Book: whenever a User rates a book, this relationship is added along with its rating value.

4.2 CRUD Operations

4.2.1 Create

- *UserGraphDAO.addUser*: creates a new User node;
- *BookGraphDAO.addBook*: creates a new Book node;
- *ReviewGraphDAO.addReview*: adds the relation REVIEWS between a User node and a Book node;
- *FollowGraphDAO.addFollow*: adds the relation FOLLOWS between a User node and another User node.

4.2.2 Read

- *UserGraphDAO.getUserById*: returns the user associated with a certain id;

- *BookGraphDAO.getBookById*: returns the book associated with a certain id;
- *ReviewGraphDAO.getReview*: returns the review associated with the user and book, if any;
- *ReviewGraphDAO.checkReview*: returns a boolean, true if the review is present, false otherwise;
- *FollowGraphDAO.getFollow*: returns the follow associated with the user and another one, if any;
- *FollowGraphDAO.checkFollow*: returns a boolean, true if the follow relathonship is present, false otherwise;

4.2.3 Update

- *BookGraphDAO.updateBook*: change book's informations;
- *ReviewGraphDAO.updateReview*: change review's star value.

4.2.4 Delete

- *UserGraphDAO.deleteUser*: deletes a User node;
- *BookGraphDAO.deleteBook*: deletes a Book node;
- *ReviewGraphDAO.deleteReview*: deletes the relation REVIEWS between a User node and a Book node;
- *FollowGraphDAO.deleteFollow*: deletes the relation FOLLOWS between a User node and another User node.

4.3 Complex Queries

4.3.1 Book Recommendations

This query is designed to retrieve a curated list of book recommendations based on the ratings given by users followed by a specified user. The recommendations are filtered based on the languages spoken by the specified user and are ordered by the rating in descending order, with a maximum of 10 results returned.

```
MATCH (user:User {id: '60de188000000000002392b'})-[:FOLLOWS]→(other:User)-[r:RATES]→(book:Book)
WHERE book.language IN [ 'en-US', 'en-GB', 'cat', 'eng', 'ita', 'spa' ]
RETURN other, book, r.rating AS rating
ORDER BY rating DESC
LIMIT 10
```

Query Breakdown

- MATCH: Connects a specified user (userId) to followed users via FOLLOWS and to books they rated via RATES;
- WHERE: Filters books by language, matching those spoken by the specified user (languages);

- **RETURN**: Retrieves followed users, books they rated, and ratings given;
- **ORDER BY & LIMIT**: Orders by rating descending and limits to 10 results.

4.3.2 Users with Similar Tastes

This query aims to find and rank users who have similar tastes to a given user based on their ratings of common books. The query ensures that these users also share at least one favourite genre with the given user.

```
MATCH (u1:User {id: '60de188000000000002392b'})-[r1:RATES]→(book:Book)←[r2:RATES]-(u2:User)
WHERE id(u1) <> id(u2)
AND abs(r1.rating - r2.rating) < 2
WITH u1, u2, COLLECT(DISTINCT book.title) AS commonBooks, COUNT(book) AS commonBookCount
WHERE SIZE([genre IN u1.favouriteGenres WHERE genre IN u2.favouriteGenres]) > 0
RETURN u2.id AS user2, u2.nickname AS Nickname, commonBooks, commonBookCount
ORDER BY commonBookCount DESC
LIMIT 10
```

Query Breakdown

- **MATCH**: Matches a user (u1) to books via RATES and another user (u2) to the same books;
- **WHERE**: Ensures users are different and ratings differ by less than 2;
- **WITH**: Collects books both users rated similarly and counts them;
- **FILTER**: Requires shared favourite genres between users;
- **RETURN**: Displays similar users, their nicknames, common books, and counts;
- **ORDER BY & LIMIT**: Sorts by common book count descending and limits to 10 results.

4.3.3 Get Recent Follows by Followed Users

This method retrieves a list of recent follow actions made by users who are followed by a specified user. The follows are ordered based on the follow relationship ID in descending order, and up to 6 recent follow actions are returned.

```
MATCH (user:User {id: '60de188000000000002392b'})-[:FOLLOWS]→(followed:User)-[f:FOLLOWS]→(other:User)
RETURN followed.id AS followedUserId, followed.nickname AS followedUserNickname,
other.id AS followedUserFollowId, other.nickname AS followedUserFollowNickname, id(f) AS followId
ORDER BY id(f) DESC
LIMIT 6
```

Query Breakdown

- **MATCH**: Connects a specified user to users they follow and who those followed users follow;
- **RETURN**: Retrieves followed user ID and nickname, user who initiated the follow ID and nickname, and the follow relationship ID;

- **ORDER BY & LIMIT**: Orders results by follow relationship ID in descending order and limits results to 6 entries.

4.3.4 Get Ratings of Followed Users

This query retrieves up to 6 recent ratings made by users followed by the specified user. It orders these ratings based on the rating ID in descending order.

```
MATCH (user:User {id: '60de1880000000000002392b'})-[:FOLLOWS]→(followed:User)-[r:RATES]→(book:Book)
RETURN followed.id AS followedUserId, followed.nickname AS followedUserNickname,
book.id AS bookId, book.title AS bookTitle, r.rating AS rating, id(r) AS ratingId
ORDER BY id(r) DESC
LIMIT 6
```

Query Breakdown

- **MATCH**: Finds a user (user) who follows other users (followed) and these followed users rate books;
- **RETURN**: Retrieves data including followed users' IDs and nicknames, along with details about the rated books such as IDs, titles, ratings given, and rating IDs;
- **ORDER BY**: Sorts the results based on the ID of the rating relationship in descending order to fetch the most recent ratings first;
- **LIMIT**: Restricts the output to a maximum of 6 records to ensure only the 6 most recent ratings are returned.

4.4 Indices

We considered implementing a couple of indices in our graph database to optimize the performance of our queries. These indices were created on the properties that our queries frequently filter or sort on.

4.4.1 Index on User ID

Our queries often involve matching a User node based on the id property. To speed up this operation, we created an index on User(id).
The Cypher command to create this index is:

```
neo4j$ CREATE INDEX FOR (n:User) ON (n.id)
```

We tested this index with all the queries we introduced, and these are our results:

	Time without index	*Time with index*
Book Recommendations	167ms	11ms
Users with Similar Tastes	24ms	3ms

Recent Follows by Followed Users	305ms	4ms
Ratings of Followed Users	32'491ms	321ms

4.4.2 Index on Book ID

We also tried to add this index since all the queries were using the Book id. Unfortunately, it didn't have the effect we hoped, especially compared to the index mentioned above. Since the speed of each query was high enough, we decided to not implement it.

5 Other Details

This section will discuss other implementation details i.e. an hypothetical sharding implementation and cross-database consistencies and how they are guaranteed.

5.1 Sharding

Sharding (or horizontal scaling) is a technique employed in distributed databases with the aim to better adapt to the different loads exploiting horizontal scaling; combined with replication this technology can boost availability and response time significantly.

For our collections there can be multiple ways to apply sharding, let us go through them one by one:

Books Collection: this collection should be the most shardable, and there are many different ways to achieve it; it is however also the collection which raises the most problems after sharding due to the possible queries, for example:

- **Range:** each book document contains the year of publication, so sharding per year is possible. In our opinion this is however the worst type of sharding for this collection, as every query made on the books collection would need to contact potentially all the shard servers.
- **Hash:** this type of sharding can be applied to the id field to divide books based on the hash of the field. This option presents the same problems as before.
- **List:** there are two options here: the first one is sharding based on **genres**; this option can improve the efficiency of the second analytic, so it has some advantages. The second possibility is sharding based on **authors**, i.e. dividing the authors in groups (name starting with A-D in

one machine, E-F in another and so on). This option would be beneficial for the first analytic, as each author corresponds to a specific shard.

Overall sharding on the books collection is not a very good idea, simple replication is advised; it is to remember that the home page allows for a search of book by title and such search would imply prompting potentially all shard machines, rendering sharding disadvantageous. Also, sharding by title ranges the same way it was hypothesized for author name is not viable since the search can also be made (and usually is) with words in the middle of the title, not at the beginning.

Users Collection: Users may be sharded way more easily than books:

- **Range:** users may be sharded by birthdate, this would have no particular disadvantage in terms of queries as each user profile is always opened individually; this is however the less advantageous option as well, as we think that other methods would be better.
- **Hash:** a standard hashing method on ids may prove very useful as it distributes the users equally between different shards, and each time a user profile is opened there is no need to access more than one shard.
- **List:** sharding based on nationality may also be a very good option depending on network activity; if for example the admins observe that users tend to search for other users of the same country, sharding by nationality may be better for latency if the servers are distributed around the globe in the respective countries.

Overall, we think that **sharding the users collection may always be a good idea when based on hashing**, as the only slow down would be in the user search but we also think that such search would not happen frequently and would happen by searching very specific usernames so that only one shard is accessed. **A list based sharding on nationality may also be good** based on users' activity.

Reviews Collection: sharding on reviews only makes sense if the reviews are sharded w.r.t. to the users in our opinion. Another option would be sharding them alongside the books but since we concluded to avoid sharding the books the only remaining option is the users.

It would be a **list based sharding based on user ids** and would allow for reduced latency when navigating a user's reviews after opening his profile.

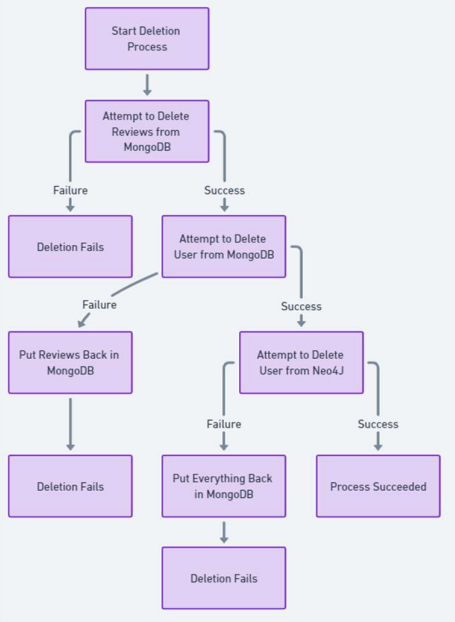
5.2 Document-Graph Consistency Checks

In Neo4J we have two types of node: Book and User; furthermore, we possess two types of relation: RATES and FOLLOWS, the first from User to Book and the second one from User to User. The information about User, Book and RATES are also contained in MongoDB, hence our need for consistency

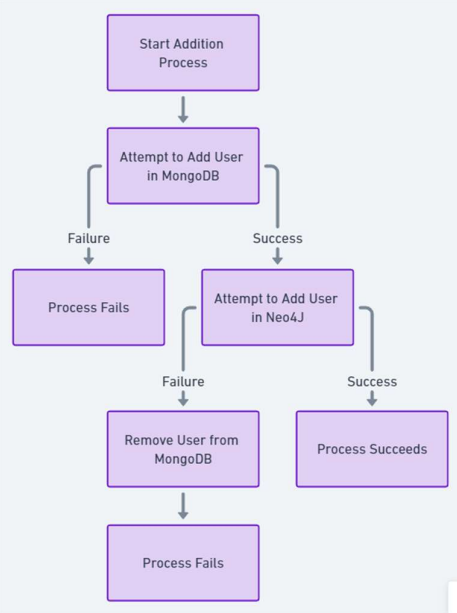
checks which make sure that if some kind of CUD operation fails at any stage and in any type of database, everything is aborted on all the others as well.

5.2.1 User Consistency Operations

Delete Account



Add User

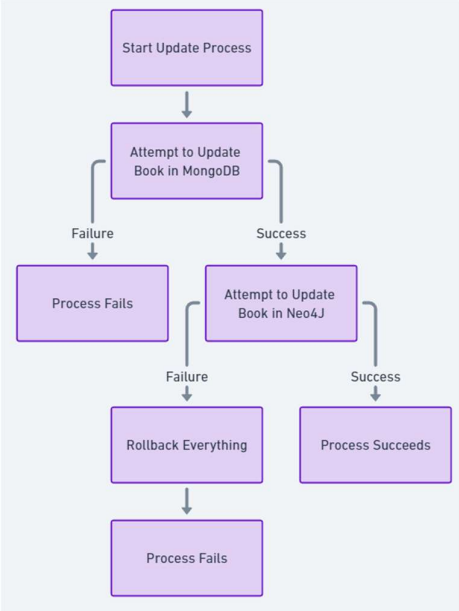


Update User

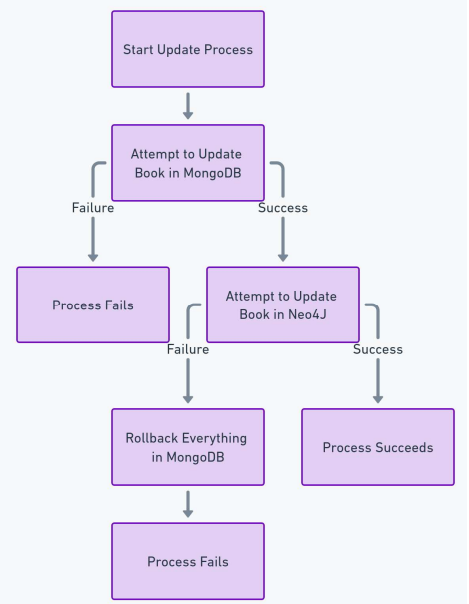
When updating a user information in MongoDB nothing changes in Neo4J.

5.2.2 Book Consistency Operations

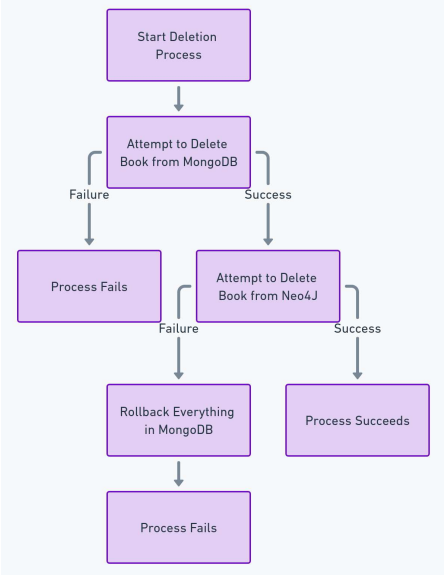
Add Book



Update Book

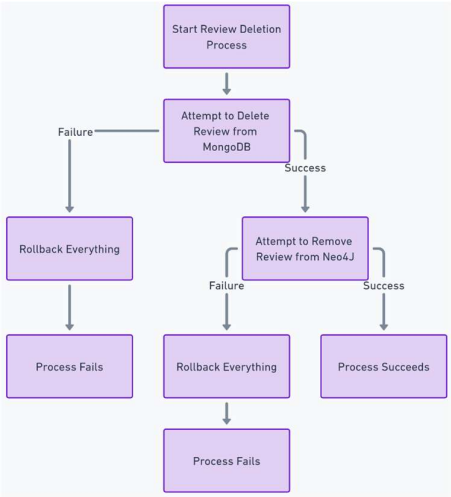


Delete Book

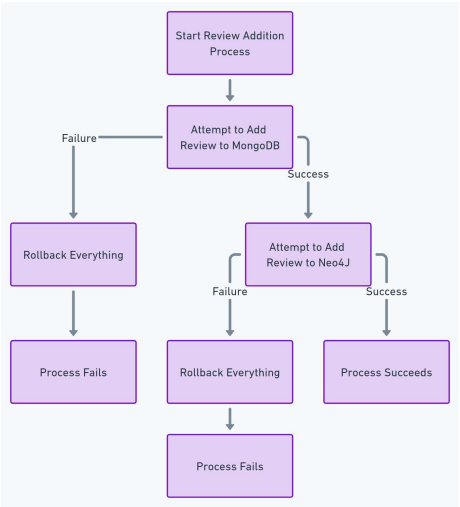


5.2.3 Review / RATES Consistency Operations

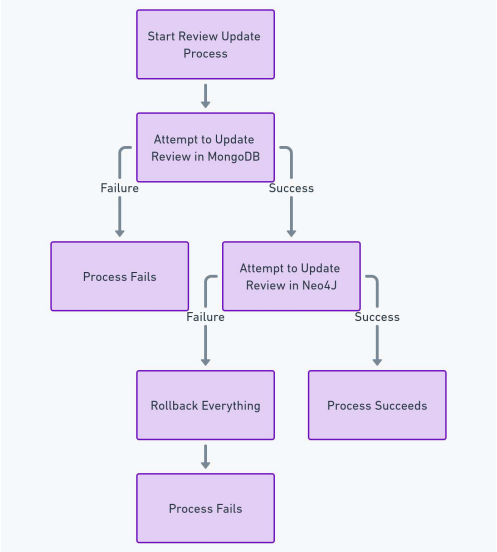
Delete Review



Add Review



Update Review



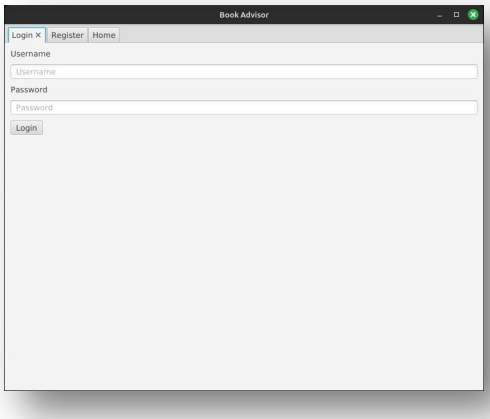
6 User Manual

All the functionalities of the GUI will be briefly explained along with images of the various tabs and windows.

6.1 StartUp Tabs

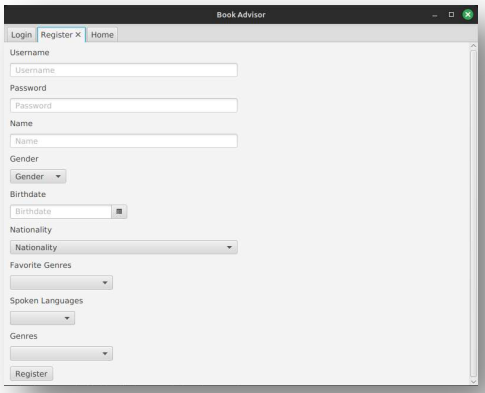
At the startup a user may decide to login, to register, or to use the app without creating a user profile (with limited social functionalities)

Login



A user who possesses an account may decide to login using his nickname and password.

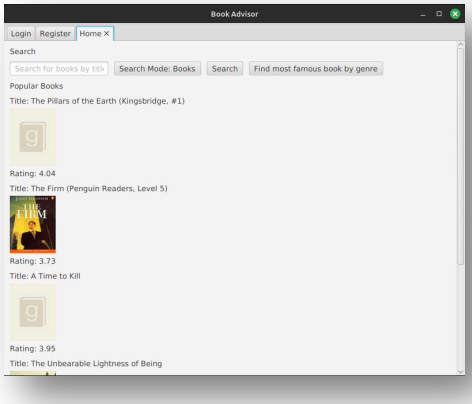
Register



A user who does not possess an account may decide to register.

The fields to fill up are username, full name, password, gender, birthdate, nationality, favourite genres and spoken languages. If an author wants to register, he also needs to specify generally the genres of his books.

Home

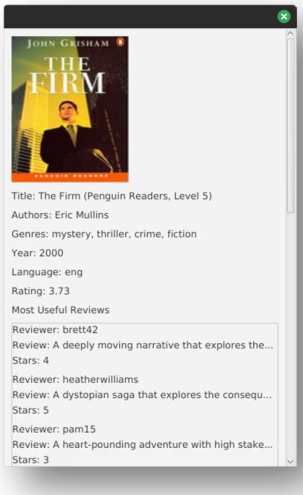


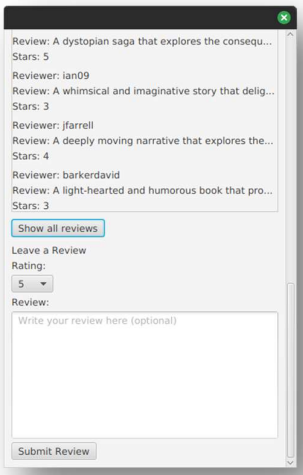
Even an unlogged user may access the home screen. It presents the five books with the most reviews, a search bar which allows to search for both books and users, and the button for one of the mongo analytics, the one which shows the most popular books given a genre for each country.

6.2 Tabs After Login

From now on every tab will be visualized from the point of view of a logged in author, as he is the one with the most functionalities; every other type of user (reviewer and unsigned) has a subset of the following features.

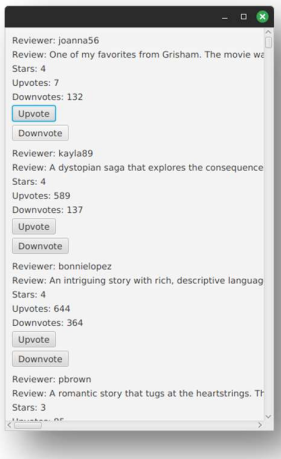
Book Info





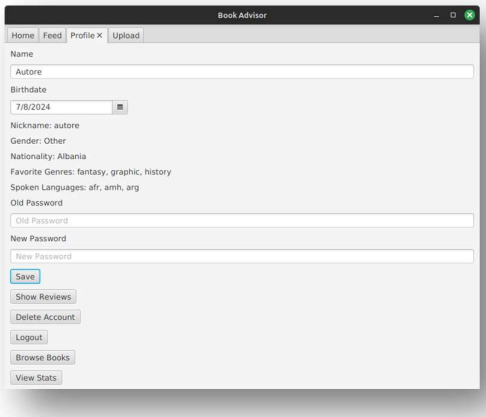
After clicking on a book or pressing the button "Go to book" (...) the book information window will pop up; such window contains all the information about the book along with the top 10 popular reviews, the possibility to leave a review and to browse them.

Review List



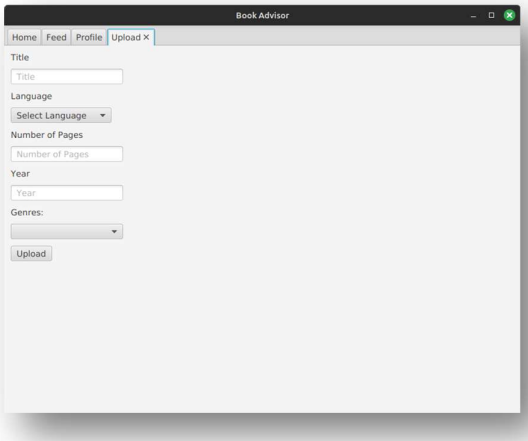
This window is accessible in multiple ways, e.g. by checking the reviews of a book or by checking the reviews of a specific user. It allows to read the reviews and to upvote and downvote them.

Profile



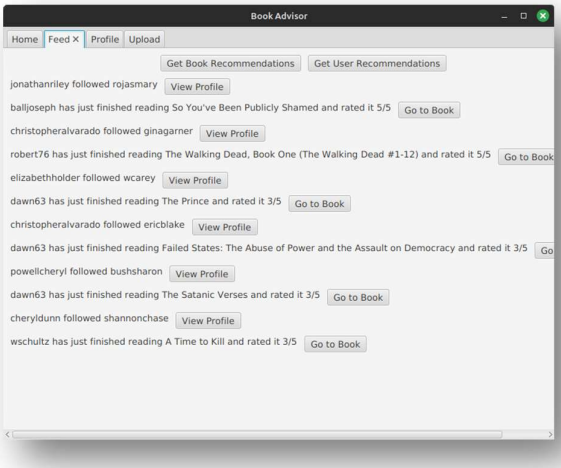
The profile tab shows all personal information and also allows to update them and to log out. A reviewer can browse his reviews and update them, and an author can also browse his books and view his statistics.

Upload a Book



An author has the possibility to upload a new book into the databases by filling the fields.

Feed



This is the most social part of the application; it allows any logged in user to view his feed, which contains recent activity about followed users, i.e. the users that have been recently followed by our followed users and the books recently rated by our followed users.

There are also two buttons, one produces books recommendations based on our followed users' taste, and the other produces users recommendations based on similar reviews in books compared to ours.

Other

There are several other windows which are technically different windows w.r.t the ones described above, but they are just result of different ways to access similar information. For example, the window that opens when looking for the most famous fiction books in each country is a list of books similar to the one that pops up when searching for a book; another example is opening another user's profile, which is similar to our own with the difference that there is no right to update.