# Distribooked

## Distributed Library Management Platform

Project Report

Gabriele Giudici
Alessandro Ciarniello
Gabriele Billi Ciani

University of Pisa

Academic Year 2024–2025

Scuola di Ingegneria

# Contents

# Application Overview

Imagine a world where you have free access to books, as simple as a few clicks on your device. Our application, *Distribooked*, reimagines how users interact with libraries, bringing a seamless, user-friendly experience to both avid readers and library administrators.

As a reader, you can easily explore a vast collection of books, searching by title, author, or category. Want to find the most popular reads in your favourite genres? The system highlights the most popular books in various categories. Once you've found the perfect book, our system helps you locate the nearest libraries where it's available. If it's a must-read, you can reserve it right away and pick it up at your convenience.

But it doesn't stop there. The system keeps track of your reading history, helps you manage reservations, and provides an easy way to check the status of your loans. For those who love planning their next read, you can save books to your favourites and keep track of what you've borrowed.

For library administrators, our application simplifies everyday operations. They can manage inventory, track overdue returns, and get insights into reading trends, helping them make informed decisions to serve their communities better.

*Distribooked* is more than a tool; it's a bridge connecting readers and libraries, fostering a culture of learning and exploration in a way that's accessible and engaging for everyone.

# GitHub Repository

The GitHub repository containing all the project files is available at:

`https://github.com/gabrielebilliciani/Distribooked`

*Note: Although the repository pertains to the implementation phase of the project, its link is provided here at the beginning of the documentation for ease of reference and accessibility.*

# Chapter 1

# System Design

This chapter outlines the complete system design, from requirements analysis to the planning of a distributed architecture. It begins with a detailed analysis of the requirements, including the identification of main actors, functional and non-functional requirements, and application mock-ups. The system's structure is then illustrated through UML class diagrams. Subsequently, the chapter delves into database organisation, encompassing both document and key-value database designs. Finally, it concludes with the design of a distributed database architecture.

## 1.1 Requirements Analysis

The analysis phase aims to establish a clear understanding of the system's scope and constraints through the identification of key actors and the definition of both functional and non-functional requirements. This systematic approach ensures that all essential aspects of the system are properly addressed in the subsequent design phases.

### 1.1.1 Main Actors

The system interacts with three types of actors:

- **Unregistered User**: a visitor who can access public features without authentication;
- **Registered User**: an authenticated user with standard privileges;
- **Administrator** (referred to as Admin): a user with administrative privileges and system management capabilities.

### 1.1.2 Functional Requirements

This section outlines the system's functional capabilities, categorised by the roles of its main actors.

#### Unregistered User

The unregistered user can:

- Register an account;
- Browse the book catalogue;
- Search books by title, author, category and most popular;
- Search authors by name;
- Access authors details and works;
- Access book details;
- Access book availability across libraries;
- Locate nearby libraries holding a specific book by entering a location;
- Check book availability in a specific library;
- Access library details.

**Registered User**

The registered user can:

- Access functionalities available to unregistered users;
- Authenticate into and out of the system;
- Reserve a book (maximum five concurrent reservations and loans);
- Canel a book's reservation;
- Save a book (for future reference);
- Unsave a previously saved book;
- Access saved books;
- Access loaned books;
- Access reserved books;
- Access read books;
- Access personal details.

**Administrator**

The administrator can:

- Access functionalities available to unregistered users (excluding account registration);
- Authenticate into and out of the system;
- Monitor library reservations;

- Monitor overdue loans in a library;
- Add a book to the catalogue;
- Add a new library to the system;
- Add a new library to the ones holding a specific book;
- Remove a book from the catalogue;
- Modify book copies available in a library;
- Complete a loan (book return procedure);
- Access reading statistics, including:
  - Most read books categorised by age group in a given period;
  - Average age of active users[1] borrowing books per city;
  - Identification of books with high or low utilisation relative to their availability.

### 1.1.3 Non-Functional Requirements

The system must satisfy the following quality attributes:

- Access to authorised functionalities must be protected by a secure authentication mechanism.
- User credentials must be securely stored using encryption.
- The system must be developed using object-oriented programming languages.
- The system must prioritise high availability without compromising data integrity for critical operations, such as book reservations and loan management.
- The system must minimise data loss to preserve records of past readings and ensure consistency in book availabilities and loan management.
- The system must support a throughput of at least 50 operations per second, considering a typical workload distribution that includes authentication, catalogue browsing, and book management activities.

### 1.1.4 Design Context for Requirements

This section expands on how the functional requirements have been interpreted and translated into concrete design principles that shape the system's behaviour. The following considerations both complement and elaborate on the previously stated requirements, clarifying key aspects of the implementation and ensuring that the system aligns with its intended use and operational needs, also providing additional context that defines how information is structured and presented to best support the expected workflows.

[1]An active user is defined as one who has borrowed at least one book within a given time period.

The design decisions are directly informed by the system's core purpose: providing a unified access point to a distributed library network. The key design choices include:

- Book details and library availability are presented together by default, as locating a book is considered an integral part of the book discovery process. This reflects the assumption that users will frequently require both types of information in a single interaction.
- Author detail pages automatically include a list of their available works, based on the expectation that users searching for an author are also interested in exploring their bibliography.
- The system treats all library branches as components of a single distributed library rather than independent entities. As a result, book searches are always performed across the entire system, without an option to restrict searches to specific branches. This aligns with the need for a seamless discovery experience that does not require prior knowledge of specific library holdings.
- The interface prioritises book and author discovery, treating branch locations as complementary information rather than a primary search criterion. This choice reflects the assumption that users generally search for content first and only afterwards refine their queries based on location constraints.

These considerations establish the foundation for the mock-ups presented in the next section, ensuring that the system's structure effectively supports its functional objectives.

### 1.1.5 Expected Workload Distribution

Based on anticipated usage patterns, the system is expected to handle a mix of operations with the following approximate distribution under normal conditions:

- Read operations (75%): These include catalogue browsing, book searches, author lookups, and viewing of book and library details. These operations are expected to constitute the majority of system usage.
- Authentication operations (15%): Login, logout, and registration processes.
- Write operations (10%): Book reservations, cancellations, and administrative actions.

While actual usage patterns may evolve after deployment, this distribution serves as a baseline for system design and performance evaluation. The non-functional requirement of supporting at least 50 operations per second is derived from this expected workload mix, with the understanding that the ratio between different operation types may shift based on real-world usage.

Initial capacity planning assumes a user base of approximately 1,000 registered users and 500 unregistered users, with concurrent usage peaking at around 20% of registered users during high-demand periods. These estimations are to be refined based on actual usage metrics once the system enters production.

## 1.2 Application Mock-ups

The front-end of the application is not implemented in this project. However, the following mock-ups illustrate the expected views and screens that would be present in a complete front-end implementation. These mock-ups serve as a visual representation of the user interface, outlining the layout and functionality of key components that actors would interact with in the final system.[2]

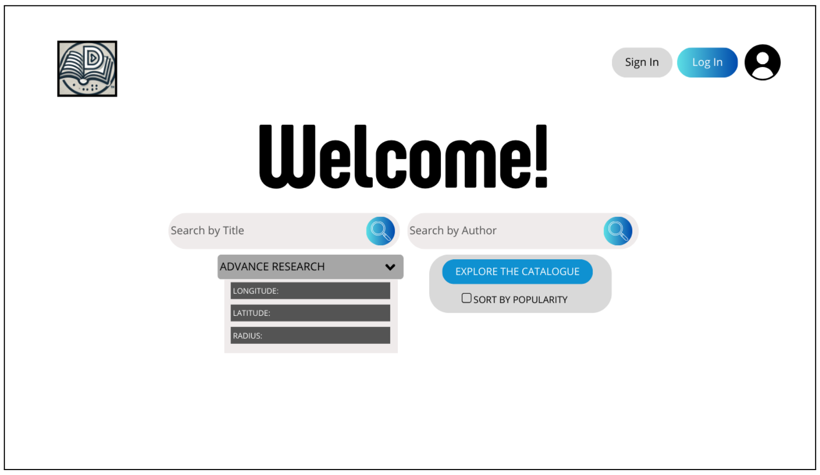


**Figure 1.1:** Mockup of the user starting page, offering navigation to various sections of the system.

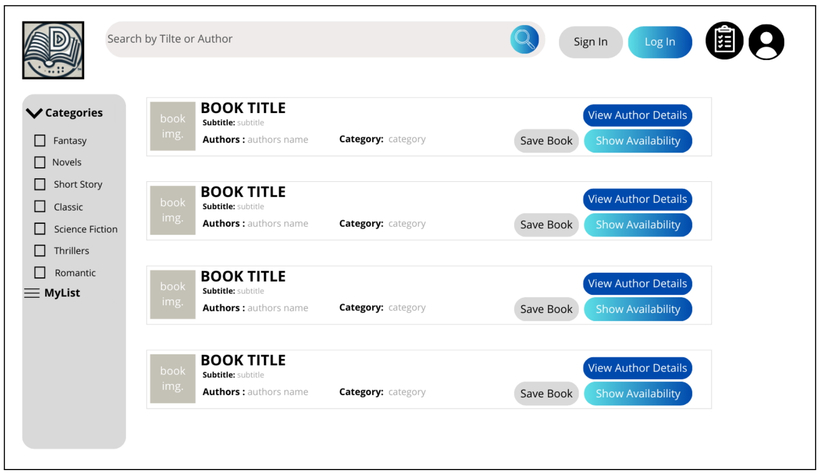
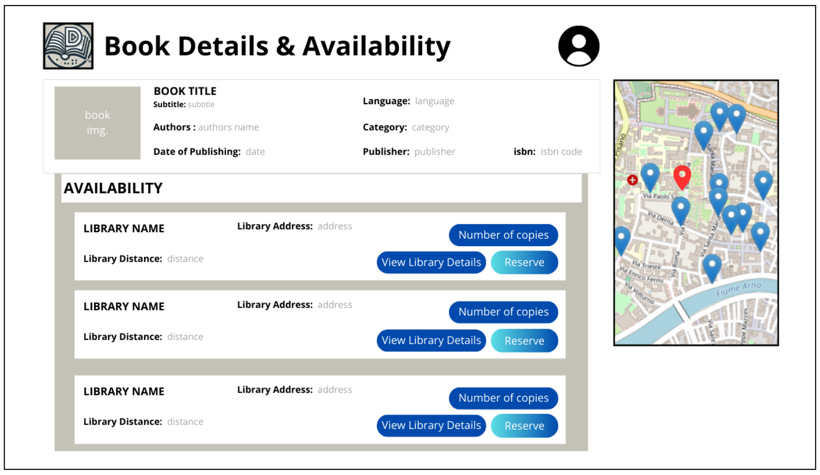


**Figure 1.2:** Mockup of the book research page, showcasing catalogue search results. Saving a book is only possible if the user is authenticated.

[2]It is assumed that all interactive actions trigger appropriate pop-ups or feedback messages to indicate whether the operation was successful or not.



## Book Details & Availability

book img.

BOOK TITLE
Subtitle: subtitle
Authors : authors name
Date of Publishing: date
Language: language
Category: category
Publisher: publisher
isbn: isbn code

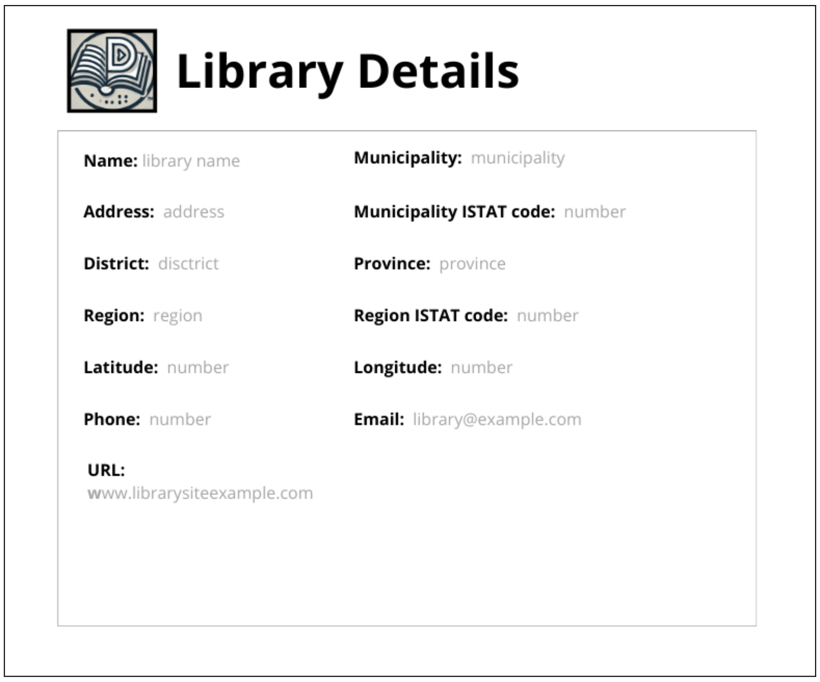
AVAILABILITY

LIBRARY NAME
Library Address: address
Library Distance: distance

LIBRARY NAME
Library Address: address
Library Distance: distance

LIBRARY NAME
Library Address: address
Library Distance: distance

**Figure 1.3:** Mockup of the book details page, where details of a selected book are shown together with all the branches which hold at least a copy of that book (cf. Section 1.1.4 on this). Reserving a book is only possible if the user is authenticated.



## Library Details

| | |
|---|---|
| **Name:** library name | **Municipality:** municipality |
| **Address:** address | **Municipality ISTAT code:** number |
| **District:** disctrict | **Province:** province |
| **Region:** region | **Region ISTAT code:** number |
| **Latitude:** number | **Longitude:** number |
| **Phone:** number | **Email:** library@example.com |
| **URL:** www.librarysiteexample.com | |

**Figure 1.4:** Mockup of the library page, providing information about a specific library.

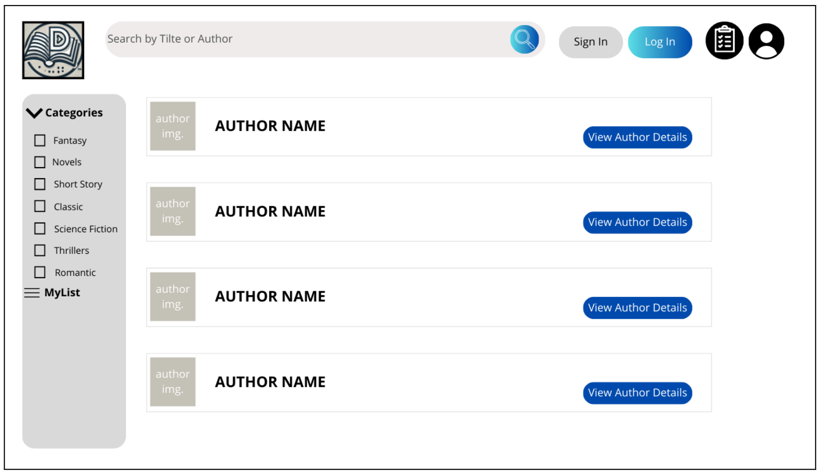


**Figure 1.5:** Mockup of the author research page, allowing users to search for authors by name.

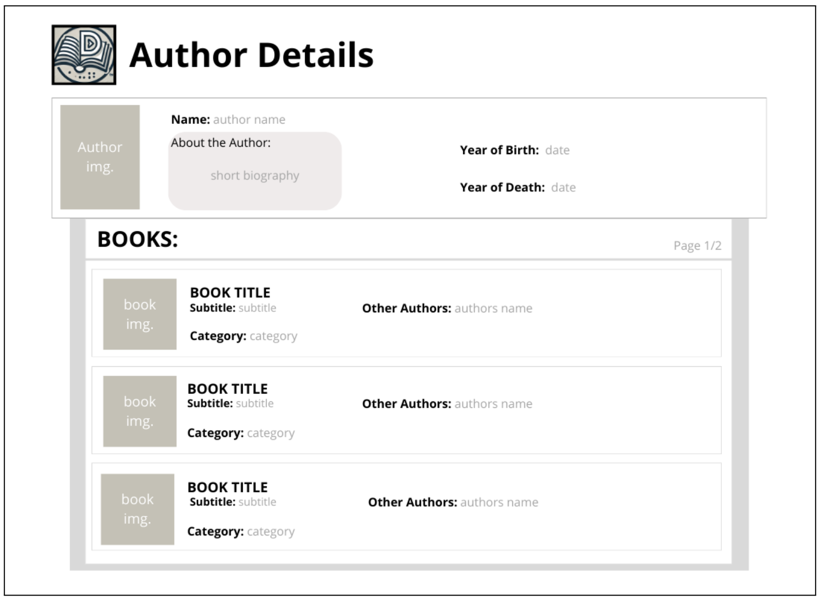


**Figure 1.6:** Mockup of the author page, displaying details about a selected author and their works.

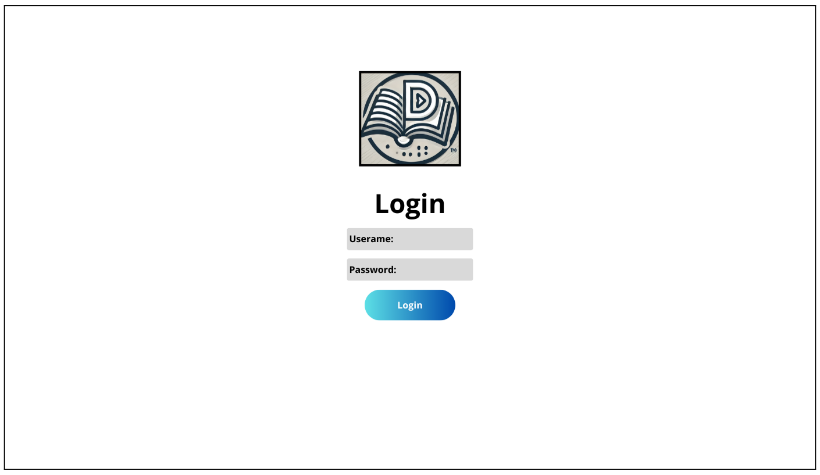


**Figure 1.7:** Mockup of the login page, allowing users to authenticate into the system.

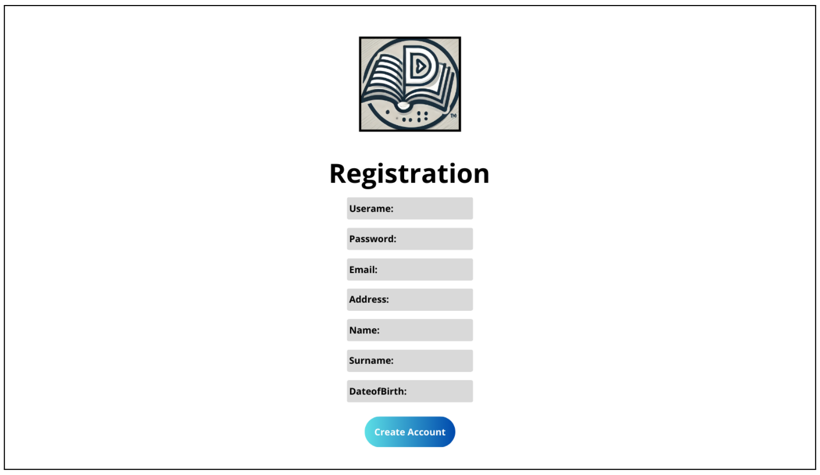


**Figure 1.8:** Mockup of the registration page, enabling new users to create an account.

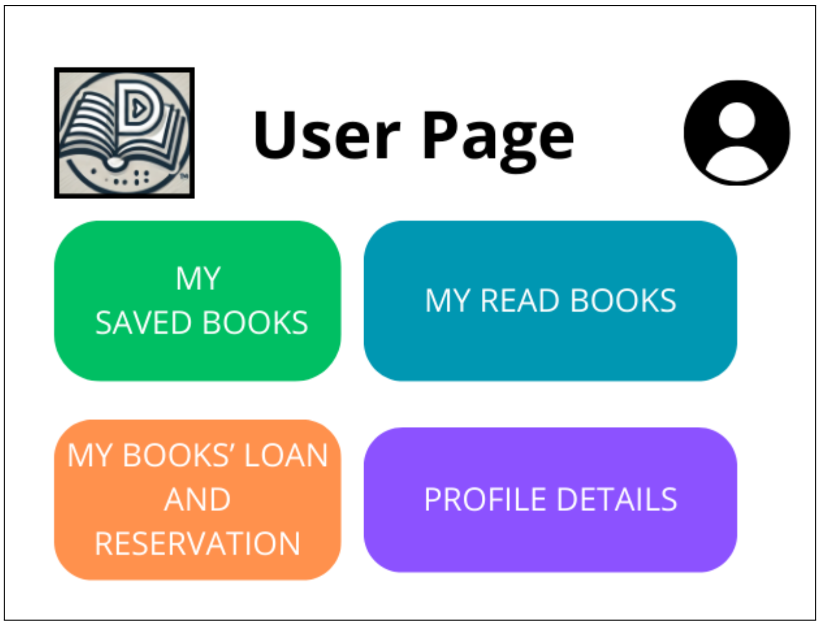


**Figure 1.9:** Mockup of the user profile page, displaying user-specific details and actions.

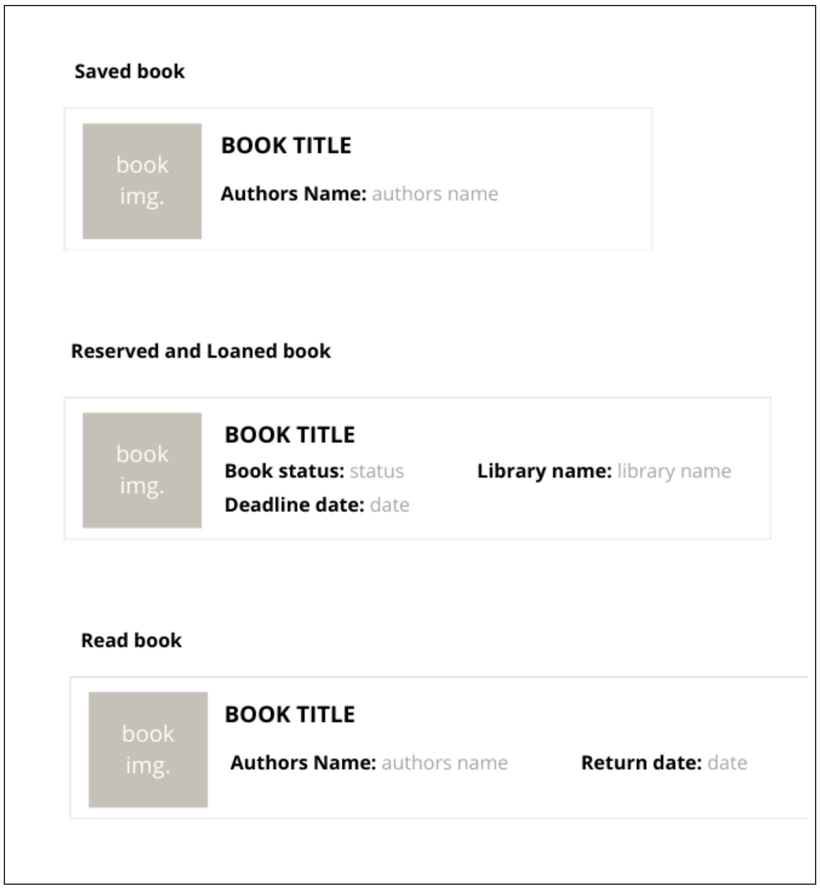


**Figure 1.10:** Examples of what book details in each of the user's pages look like.

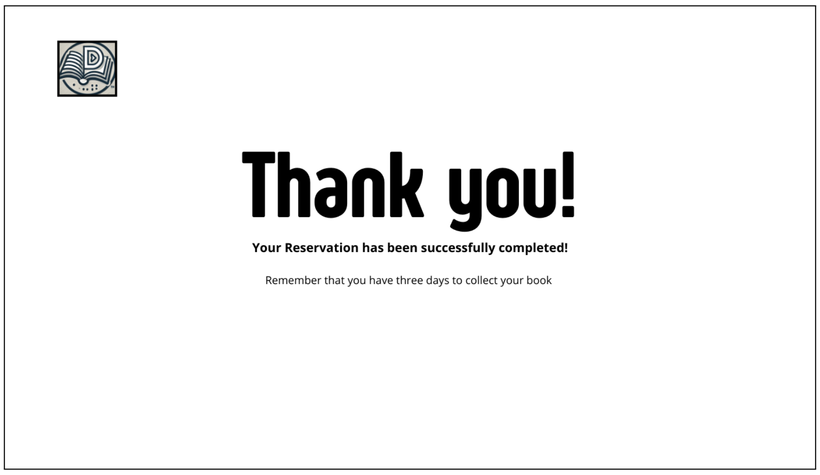

Thank you!
Your Reservation has been successfully completed!
Remember that you have three days to collect your book

**Figure 1.11:** Mockup of the book reservation confirmation page.

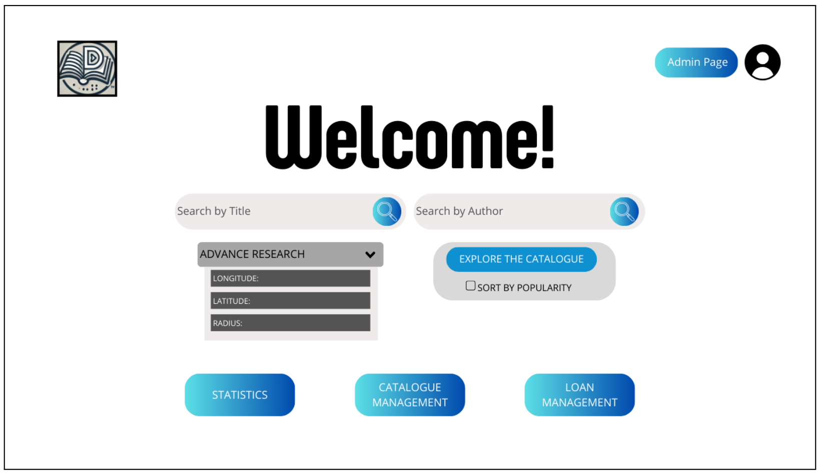


**Figure 1.12:** Mockup of the admin starting page, providing access to administrative functions.

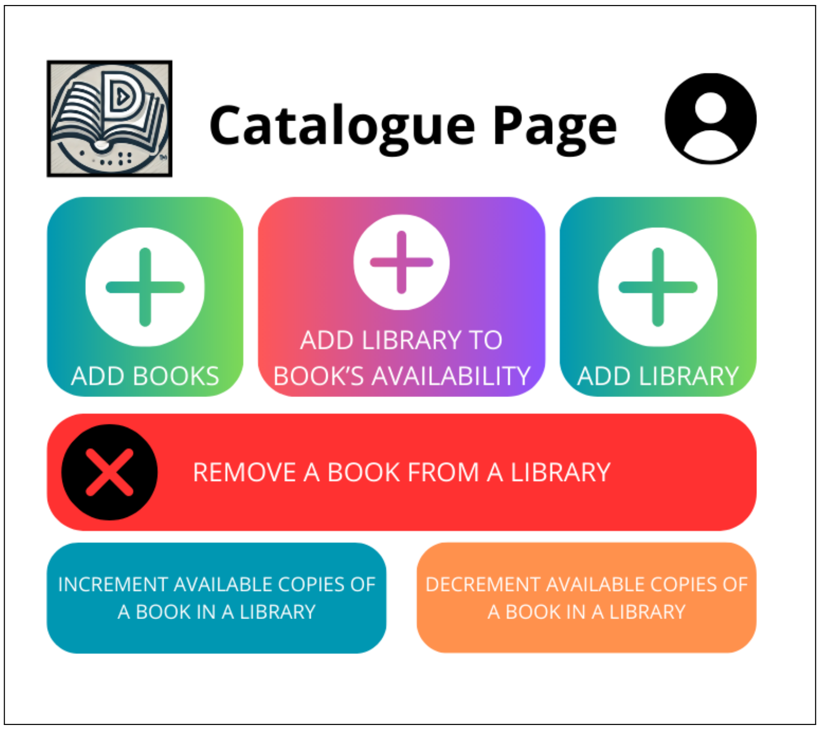

Catalogue Page
ADD BOOKS
ADD LIBRARY TO BOOK'S AVAILABILITY
ADD LIBRARY
REMOVE A BOOK FROM A LIBRARY
INCREMENT AVAILABLE COPIES OF A BOOK IN A LIBRARY
DECREMENT AVAILABLE COPIES OF A BOOK IN A LIBRARY

**Figure 1.13:** Mockup of the book catalogue management page, displaying a set of options admins can perform on the catalogue. For each of the possible actions, a pop-up for entering details is expected to appear.

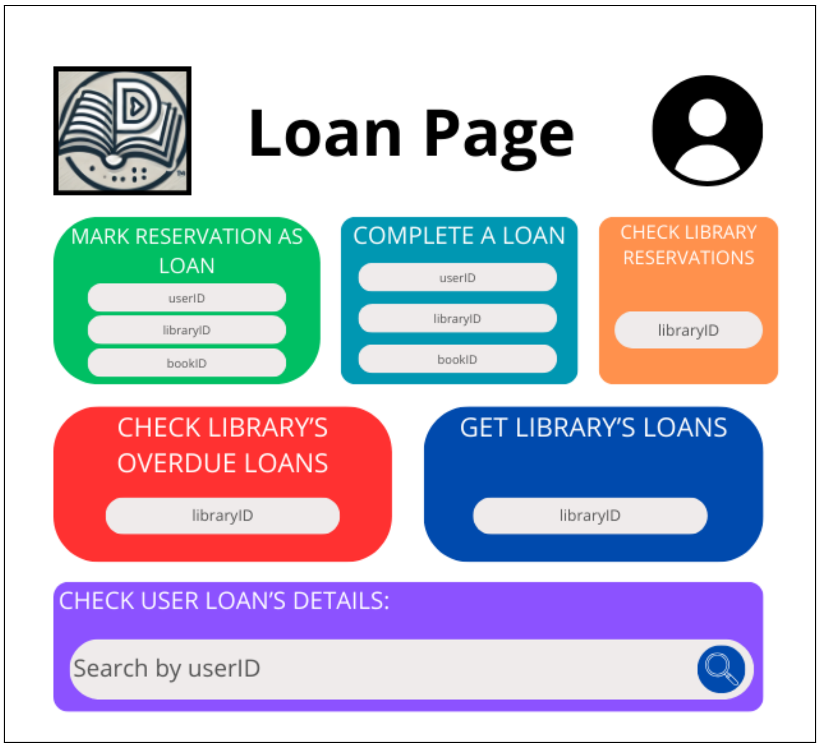


**Figure 1.14:** Mockup of the loan management page, where admins manage library's reservations and loans. In a real implementation, ID input could be automated in various ways. For example, once a list of reservations or loans is retrieved, the admin could simply click on an entry to auto-fill the relevant fields (e.g. userID, bookID, libraryID) for actions like marking a reservation as a loan or completing a loan. Other mechanisms, such as dropdown menus, search-based filtering, or barcode scanning, could further streamline the process.

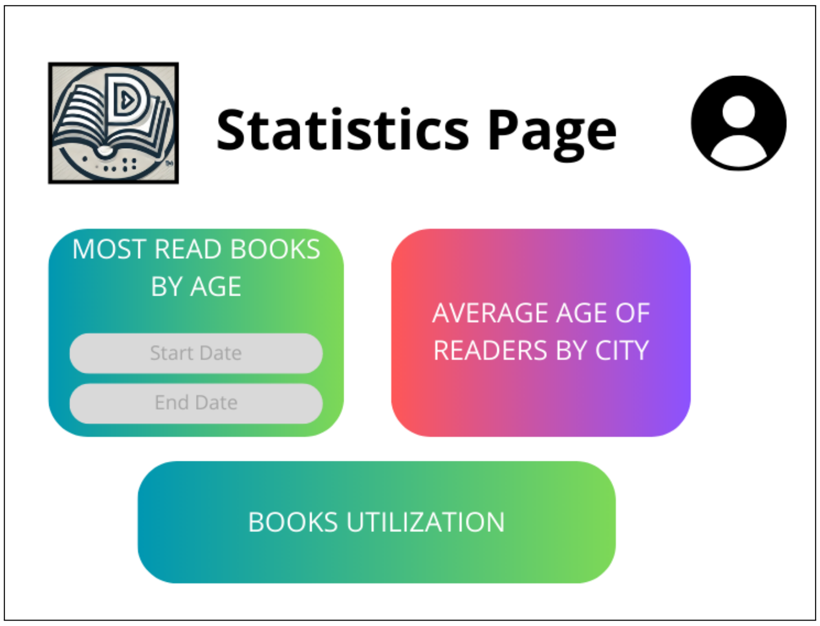
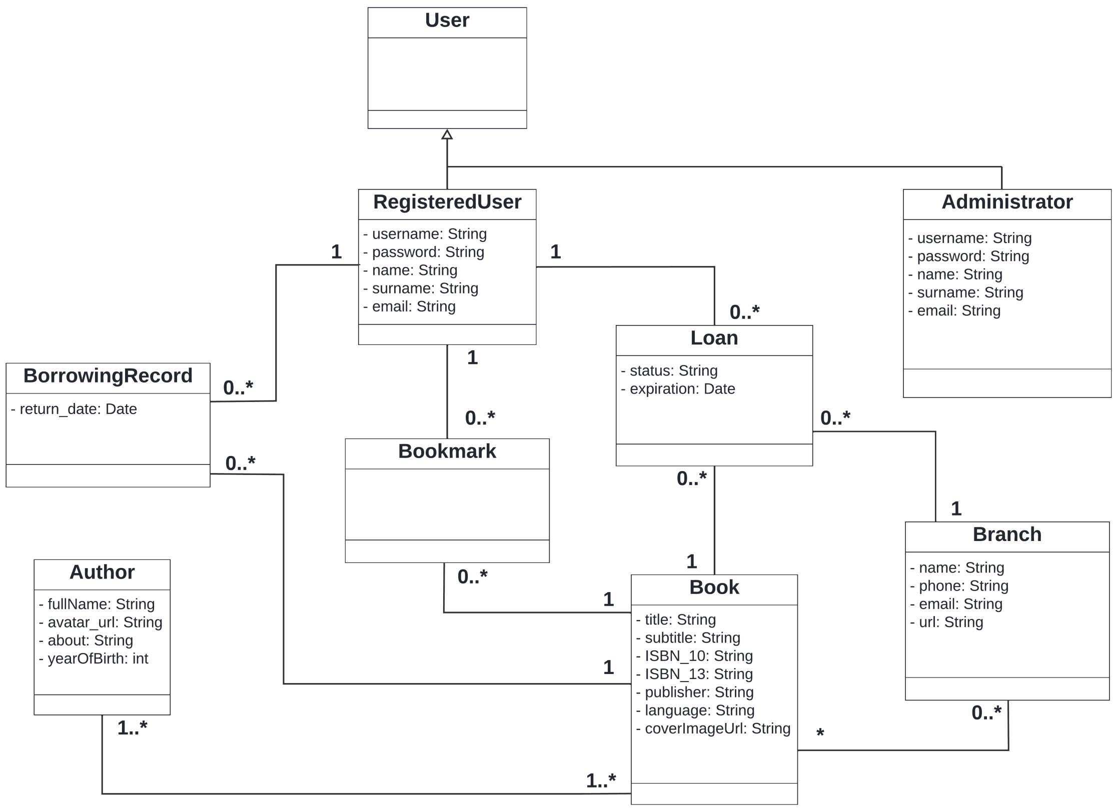


**Figure 1.15:** Mockup of the statistics page, presenting reading trends and user activity data.

## 1.3 UML Class Diagram



## 1.4 Database Organisation

The system's data layer employs a hybrid database architecture that combines MongoDB as a document database with Redis as a key-value store, each serving as the source of truth for different aspects of the system. This approach leverages MongoDB's flexible schema and rich querying capabilities for managing persistent domain entities, whilst Redis handles both real-time data (such as current book availability) and synchronisation-critical operations where concurrent access must be carefully controlled.

This section details the organisation of the system's data layer. It begins by describing the data sources used to populate the system, followed by an in-depth examination of the MongoDB implementation, including its collections and indexing strategies. The section concludes with a detailed overview of the Redis implementation, focusing on its namespace organisation and key structures.

### 1.4.1 Data Set Collection and Description

The system utilises multiple real-world data sets to construct a comprehensive library management environment, integrating information on books, libraries, and users. The primary sources include the *Amazon Review* data set [2], the *Anagrafe delle Biblioteche Italiane* [3], and the *Project Gutenberg Metadata* [1] for additional bibliographic records. Additionally, a synthetic user data set was generated to simulate a realistic user base.

The first stage of data preparation involved the integration of book-related records. The Amazon Reviews data set includes various product categories, requiring a filtering step to retain only records associated with books. Essential fields such as ISBN-10, ISBN-13, and title were extracted, and records containing null values in critical fields were removed to ensure data consistency. The resulting data set was then merged with the Project Gutenberg Metadata, which, despite its smaller size (19MB compared to Amazon's 13GB), provided valuable bibliographic details. Since Gutenberg records lack ISBN codes due to the historical nature of the collection, ISBNs from Amazon were used to supplement these entries. The merging process, conducted based on book titles, resulted in an expanded version of the Gutenberg data set, as Amazon Reviews contained multiple editions of the same books. This combination also allowed for the enrichment of missing attributes, where values absent in one source were filled using data from the other. The final book data set, formatted in JSON to maintain compatibility with the document database, has a total size of 88MB.

Library data were sourced from the *Anagrafe delle Biblioteche Italiane*, and a preprocessing phase was performed to remove irrelevant attributes and records containing excessive null values. The selection was restricted to library branches located in Pisa, ensuring a focused data set for the geographic scope of the system. This cleaning process resulted in a refined data set consisting of 26 library branches, formatted in JSON and structured to align with the fields present in the document database. The final library data set has a total size of 13KB.

Since real-world user data were not available, a synthetic user data set was generated using the *Faker* library in Python. This allowed for the creation of realistic user records, including names, addresses, email addresses, and phone numbers, while ensuring the protection of sensitive information. To maintain data coherence and facilitate the generation of meaningful analytics, user locations were limited to four cities: Pisa (70%), Livorno (10%), Lucca (10%), and Florence (10%). User passwords were hashed using the same encryption function (bcrypt) employed by the system's authentication mechanism. This ensured that artificially generated users could interact with the system as real users, enabling realistic testing scenarios. A separate file containing the plaintext credentials was maintained for simulation purposes, allowing the execution of automated login operations to generate system load.

Following data preparation, the database was populated with embedded documents and relationships reflecting real-world constraints. Books were assigned to library branches using a round-robin mechanism with additional randomisation. Each book was initially distributed sequentially across the 26 branches, ensuring an even spread. To simulate the availability of multiple copies across locations, an additional randomisation step was introduced, where each book was also assigned to three randomly selected branches, excluding the primary branch. The number of copies in each branch was determined by a random value between one and five, ensuring diversity in availability.

Historical reading data were generated to simulate past borrowings. Readings were limited to the years 2022, 2023, and 2024, with each user being assigned a random number of books per year. The maximum number of readings was capped at 80 per year, reflecting the estimated reading frequency of an avid reader. Random book selections were then assigned to each user based on this distribution.

The key-value database was populated by extracting book availability data from the document database. This involved iterating through book records to retrieve book IDs, corresponding branch IDs, and the number of available copies. These values were then structured as key-value pairs, where the key consisted of the book ID and library ID, while the value stored the number of available copies. This process ensured consistency between the document and key-value databases, supporting real-time availability tracking within the system.

The complete dataset preparation pipeline, including all preprocessing scripts used to filter, clean, and structure the data before generating the final database dumps, is available in the repository under the `dataset` directory. This directory also contains the database dumps for both MongoDB and Redis, allowing for a direct restoration of the system's initial state.

### 1.4.2 Document Database: MongoDB

MongoDB is used as document-oriented database for the system, offering high flexibility and scalability for handling structured and semi-structured data. Its schema-less nature allows efficient modelling of domain entities while enabling rapid iteration over data structures as requirements evolve. Additionally, MongoDB's indexing capabilities and support for geospatial queries make it well-suited for functionalities such as searching for libraries and books based on location.

#### 1.4.2.1 Collections

The system relies on the following MongoDB collections to manage its core data entities: `authors`, `books`, `branches`, `outbox`, `users`.

In the following section, further details on these collections are provided, including document examples to illustrate their structure and explanations of the design choices made in their modelling.

The `outbox` collection will be discussed in Section 2.1.1, as its inclusion is not derived from initial system design requirements. Instead, it was introduced as part of an architectural decision to ensure eventual consistency in the system and its role is more clearly defined when discussing the Outbox Pattern which was implemented.

**Authors**

The `authors` collection stores biographical details of authors along with a list of their published works. The following example illustrates the structure of an author document.

```
{
    "_id": ObjectId("679cb31ab477993c5cdc08da"),
    "fullName": "Thomas Mann",
    "yearOfBirth": "1875",
    "yearOfDeath": "1955",
    "avatarUrl":
        "https://m.media-amazon.com/images/I/91bDRRiA2fL._SY600_
        .jpg",
    "about": "Paul Thomas Mann was a German novelist...",
    "books": [
        {
            "_id": ObjectId("679cb33db477993c5cdcdf5c"),
            "title": "Gladius Dei; Schwere Stunde",
            "authors": [
                {
                    "id": ObjectId("679cb31ab477993c5cdc08da"),
                    "fullName": "Thomas Mann"
                }
            ]
        },
        ...
        {
            "_id": ObjectId("679cb34cb477993c5cdd5e5c"),
            "title": "Royal Highness",
            "subtitle": "Paperback - January 27, 2019",
            "categories": ["Books", "Literature & Fiction",
                "Literary"],
            "coverImageUrl":
                "https://m.media-amazon.com/images/I/51NSftca6AL.
                _SX348_BO1,204,203,200_.jpg",
            "authors": [
                {
                    "id": ObjectId("679cb31ab477993c5cdc08da"),
                    "fullName": "Thomas Mann"
                },
                {
                    "id": ObjectId("679cb31ab477993c5cdc3af0"),
                    "fullName": "Curtis A. Cecil [Translator]"
                },
                ...
            ]
        }
    ]
}
```

**Code snippet 1.1:** Example document from the `authors` collection.

The `authors` collection stores information about book authors, including their full name, birth and death years, a brief biography, and an avatar URL. Additionally, each author document embeds a list of books associated with them, represented as an array of objects. Each embedded book includes its unique identifier, title, and a list of authors, allowing for efficient retrieval of other author's details.

The document structure reflects the system's core design principle that author exploration and book discovery are intrinsically linked operations. As established in the design context (cf. Section 1.1.4), when users access author details, they expect to see the author's bibliography as an integral part of the information. This requirement directly influences the database design: book details are embedded within the author documents to ensure that all relevant information is available in a single query, supporting the seamless author-to-books discovery workflow that characterises the system.

Given the inherently finite nature of an author's bibliography and the selective subset of book information stored within each author document, the embedded structure poses no risk of exceeding MongoDB's document size limits, even for the most prolific authors.

### Books

The `books` collection stores information about individual books, including metadata such as title, publication details, language, and associated authors. Additionally, each book document contains an embedded list of branches, representing the libraries where copies of the book are available, along with their locations and availability.

```
{
    "_id": ObjectId("679cb33db477993c5cdcdf9c"),
    "title": "Der Tod in Venedig",
    "publicationDate": "2004-04-01",
    "language": "German",
    "authors": [
        {
            "_id": ObjectId("679cb31ab477993c5cdc08da"),
            "fullName": "Thomas Mann"
        }
    ],
    "readingsCount": 0,
    "branches": [
        {
            "_id": ObjectId("679cb364d125ba32463b9746"),
            "libraryName": "Biblioteca dell'Istituto Domus
               Mazziniana",
            "location": { "type": "Point", "coordinates":
               [10.3980345, 43.7114097] },
            "address": {
                "street": "Via Giuseppe Mazzini 71",
                "city": "Pisa",
                "province": "Pisa",
                "postalCode": "56125",
                "country": "Italia"
            },
            "numberOfCopies": 5
        },
        ...
    ]
}
```

**Code snippet 1.2:** Example document from the `books` collection.

Each book document may include additional metadata depending on the available information. These attributes include a subtitle, if applicable; publisher details, indicating the publishing house; categories that classify the book into relevant genres or topics; ISBN codes (`isbn10` and `isbn13`) for unique identification; and a cover image URL linking to an online image representation.

The `branches` array is essential for managing book availability across different library locations. Each embedded branch document contains both structured address information and a geographic coordinate point (`location`). The latter is particularly relevant for queries that retrieve books based on proximity, enabling users to search for available copies near a specific location.

Given the inherently limited number of library branches in the system, the embedded structure does not pose a risk of exceeding MongoDB's document size limits. Additionally, while books may have multiple authors, the number remains naturally constrained, thus not posing a risk of excessive document growth.

### Branches

Each branch document represents a physical library where books are available for loan. The document contains essential details such as the library's name, address, geographic coordinates, and contact information.

```
{
  "_id": ObjectId("679cb364d125ba32463b9744"),
  "isilCode": "IT-PI0112",
  "name": "Biblioteca Universitaria",
  "address": {
    "street": "Piazza San Matteo in Soarta 2",
    "city": "Pisa",
    "province": "Pisa",
    "postalCode": "56126",
    "country": "Italia"
  },
  "location": { "type": "Point", "coordinates": [ 10.4074101,
     43.7146166 ] },
  "phone": "+39 050573749",
  "email": "bu-pi@cultura.gov.it",
  "url": "https://www.bibliotecauniversitaria.pi.it/it/index.html"
}
```

**Code snippet 1.3:** Example of a branch document in MongoDB.

This document structure provides a comprehensive representation of a library branch. The `isilCode` is an identifier for the library within the international ISIL system. The `address` field contains detailed location information, while the `location` field uses GeoJSON format for spatial queries. The document also includes contact details such as phone number, email, and website URL, ensuring accessibility for users seeking further information about the library and its services.

Branch documents do not store a list of books available at each location. This design choice aligns with the system's approach of treating all library branches as components of a single distributed library rather than independent entities. Consequently, book searches are always performed across the entire system, without the option to restrict searches to specific branches. This ensures a seamless discovery experience that does not require prior knowledge of individual library holdings, as discussed in Section 1.1.4.

### Users

Each user document represents a registered individual in the system, containing personal details, login credentials, address, geolocation data, and lists of books the user has read or saved for future reference.

```
{
  "_id": ObjectId("679cb391d5e81efe4645569e"),
  "username": "victo13",
  "name": "Victoria",
  "surname": "Disdero",
  "dateOfBirth": "1983-10-13",
  "password": "8274699902124007b5fd493834602ec",
  "userType": "USER",
  "email": "disderovictoria@libero.it",
  "address": {
    "street": "Stretto Livio, 765 Piano 3",
    "city": "Pisa",
    "postalCode": "56126",
    "province": "Pisa",
    "country": "Italy"
  },
  "location": {
    "type": "Point",
    "coordinates": [10.410596867140791, 43.70232171686015]
  },
  "readings": [
    {
      "id": "679cb354b477993c5cdda049",
      "title": "The Remedy for Unemployment",
      "authors": [
        {
          "_id": ObjectId("679cb31ab477993c5cdbfa9e"),
          "fullName": "Alfred Russel Wallace"
        }
      ],
      "returnDate": "2024-10-03",
      "branch": { "type": "Point", "coordinates": [10.3975611,
         43.7193769] }
    },
    ...
  ],
  "savedBooks": [
    {
      "id": "679cb362b477993c5cde074e",
      "title": "The truth about Ireland",
      "authors": [
        {
          "_id": ObjectId("679cb31ab477993c5cdc8652"),
          "fullName": "Alexander Corkey"
        }
      ]
    },
    ...
  ]
}
```

**Code snippet 1.4:** Example of a user document in MongoDB.

This document structure stores essential user information, including personal details, email, and user type. The `address` field contains structured address information, while the `location` field uses GeoJSON format for spatial queries. The `readings` array lists books the user has borrowed, including metadata such as title, author(s), return date, and the branch where the book was borrowed. Similarly, the `savedBooks` array stores books that the user has saved for future reference.

The `password` field stores a hashed representation of the user's password to ensure security and protect sensitive credentials.

Each user document includes a `userType` field, which can be either `USER` or `ADMIN`. In the case of administrators, the `savedBooks` and `readings` arrays are not present, as these functionalities are not available to admin users.

A limit of 50 books is imposed on the `savedBooks` array to ensure controlled document growth. Regarding the `readings` array, statistical data suggests that even “super readers” – individuals who read extensively – typically consume around 50 books per year. While this may lead to document growth over time, such high-volume readers represent a minority of users, making it unlikely that a document would approach MongoDB’s maximum size limit. Nevertheless, in rare cases of exceptionally active users, storage implications may warrant further evaluation.

#### 1.4.2.2 Indices

To optimise query performance, various indices are planned to be introduced in the system’s MongoDB collections. These indices aim to improve efficiency for frequently executed queries, such as searching for books by title, filtering by category, retrieving all works by a specific author, identifying the most popular books, and performing proximity-based searches for book availability in nearby libraries.

In particular, we consider defining an index on the `title` field to optimise book searches by title and avoid full collection scans. A multikey index on the `categories` field may help improve filtering efficiency for genre-based searches. Similarly, an index on the `fullName` field in the `authors` collection could be useful for quickly retrieving an author’s works. Additionally, an index on the `readingsCount` field might be beneficial for ranking books based on their popularity by enabling more efficient sorting operations.

A geospatial index (`2dsphere`) is also under consideration for the `branches.location` field in the `books` collection. This would support location-based queries, allowing users to find books available in nearby libraries.

The effectiveness of these indices will be validated through performance testing, comparing query execution before and after their introduction. A detailed analysis of indexing performance and experimental results can be found in Section 3.2.2.

### 1.4.3 Key-Value Database: Redis

Redis has been chosen for certain aspects of the application where its speed and synchronisation capabilities provide significant advantages. The system leverages Redis for managing time-sensitive data and ensuring atomic operations that prevent race conditions in concurrent environments.

#### 1.4.3.1 Namespaces and Keys

Redis does not provide built-in namespace support, but logical namespaces can be defined by convention[3]. This is particularly useful when multiple applications share the same Redis instance, as it allows keys to be grouped using common prefixes. However, since our system is the only application using Redis in this context, we do not employ a global prefixing convention. Instead, we directly define key structures relevant to our application’s needs, ensuring clarity and consistency in key naming.

Throughout the Redis key structure, a consistent naming convention is followed: all keys adhere to the schema `EntityName:⟨EntityId⟩:EntityAttribute`, which provides clarity and predictability in key naming and organization.

We have defined multiple Redis keys that fall into four categories:

– Key-Value Pairs: Used for storing simple data structures.

[3]See Redis best practices: `https://redis.io/blog/5-key-takeaways-for-developing-with-redis/`.

– Hashes: Used for storing structured data where each hash contains multiple fields related to a common entity.

– Sorted Sets (ZSets): Used for managing expiration-based operations, particularly for tracking reservation and loan deadlines.

– Streams: Used to support event-driven architecture, allowing asynchronous processing of key changes.

### Key-Value Pairs

Simple key-value pairs are used for storing atomic values that need frequent updates and fast access.

– `book:⟨bookId⟩:lib:⟨libraryId⟩:avail`: Stores the current number of available copies of a book in a specific library. This value is atomically incremented when a book is returned and decremented when it is loaned.

### Hashes

Redis hashes are used for their efficiency in retrieving both entire hashes and individual fields. The system leverages Redis 7.0's feature of setting TTL on individual hash fields, which is particularly useful for managing loan and reservation expirations. This allows the system to directly query the remaining time until a loan or reservation expires.

The system uses the following hashes:

– `lib:⟨libraryId⟩:loans`: Stores active loans in a library. Each field follows the pattern `user:⟨userId⟩:book:⟨bookId⟩:start` with the loan timestamp as value. A TTL is set on individual fields to represent the loan duration (30 days).

– `lib:⟨libraryId⟩:res`: Stores book reservations per library. Each field follows the pattern `user:⟨userId⟩:book:⟨bookId⟩:start` with the reservation timestamp as value. The TTL is set on individual fields to represent the reservation duration (3 days).

– `lib:⟨libraryId⟩:overdue`: Stores overdue loans for a specific library. Each field follows the pattern `user:⟨userId⟩:book:⟨bookId⟩:start`, where the value is the timestamp (in milliseconds) marking when the loan became overdue.

– `user:⟨userId⟩:active`: Stores active loans and reservations for a user. Each field follows the pattern `lib:⟨libraryId⟩:book:⟨bookId⟩:info` and the value is a structured JSON object containing the status (reservation or loan), the expiration timestamp (representing the return date for a loan or the final date for pick-up for a reservation), the unique book identifier, the library where the book is borrowed or reserved, and the timestamp indicating when the reservation or loan was created. An example of this structure:

```
{
    "status": "LOANED",
    "deadlineDate": 1738272760031,
    "bookId": "679a76b930f95ec8c6e7c608",
    "libraryId": "679a76df5cf745180198d6a1",
    "timestamp": 1738272760031
}
```

**Code snippet 1.5:** Example of a user activity entry

The TTL functionality on hash fields enables efficient tracking of expiration times and can be queried to determine the remaining time for a loan or reservation.

### Sorted Sets

Sorted sets are used to track expiration-based events, such as due dates for loans or reservation deadlines (cf. Section 2.1.1 and 2.2.3.1 for more details on the usage of sorted sets).

- `zset:res-exp`: Stores reservation expirations, where the member is `user:⟨userId⟩:book:⟨bookId⟩:lib:⟨libraryId⟩:exp` and the score is the expiration timestamp.
- `zset:loan-exp`: Stores loan expirations, using the same format as the reservation expiration set.

### Streams

Streams allow the system to asynchronously process key changes, enabling event-driven updates and ensuring consistency between Redis and MongoDB (cf. Sections 2.1.1 and 2.2.3.1 for more details on the usage of streams).

- `stream:decrement-copies`: Logs book copy decrements in a library.
- `stream:increment-copies`: Logs book copy increments in a library.
- `stream:remove-library`: Logs the removal of book from a library.
- `stream:add-library`: Logs the addition of a book to a library.
- `stream:completed-loans`: Tracks successfully completed loans.

## 1.5 Distributed Database Design

The system employs a distributed database architecture to ensure redundancy, scalability, and operational continuity. It is deployed in a three-node configuration, enabling efficient load distribution and fault tolerance. A detailed description of deployment strategies and replica set configuration are provided in Section 3.1.2.

To manage consistency and availability trade-offs, the system employs a hybrid database approach, integrating MongoDB and Redis. Their cooperation ensures efficient query handling and strict synchronisation where necessary. The rationale behind this design and its impact on system behaviour are further analysed in Section 3.1.3.

### 1.5.1 Cooperation between different database management systems

MongoDB serves as the primary data store, managing books, authors, libraries and user data. It provides flexible querying, indexing, and aggregation capabilities, optimised for read-intensive operations such as catalogue searches and user activity tracking.

Redis is employed as a synchronisation layer for real-time operations requiring immediate consistency, including book reservations and loan management. By leveraging atomic operations and expiration-based mechanisms, it prevents race conditions and ensures transactional integrity. Time-sensitive operations are processed in Redis first, with subsequent persistence to MongoDB to maintain a durable audit trail. The implementation details of this coordination, including Redis Streams and Sorted Sets, are elaborated in Section 2.2.3.1.

To maintain consistency across the distributed system, the architecture implements an asynchronous update propagation mechanism based on the outbox pattern. This design element serves two crucial purposes: it ensures the eventual propagation of state changes between Redis and MongoDB, and it maintains internal consistency within MongoDB's document relationships. This architectural choice allows the system to handle both cross-database synchronisation and document embedding updates in a reliable and scalable way, while remaining resilient to temporary failures. The technical implementation details of this consistency mechanism are discussed in Section 2.2.3.1.

This division of responsibilities allows the system to balance high availability for read-heavy operations with strict consistency for transactional workflows, ensuring both responsiveness and data integrity.

### 1.5.2 Scalability Considerations: Potential Sharding Strategies

While the current implementation does not incorporate sharding, this section examines potential sharding strategies that could be employed should the system require horizontal scalability due to increased data volumes or workload demands.

#### Redis Sharding Strategy

For Redis, an effective sharding approach would involve distributing data based on library IDs, ensuring that all hash structures and availability keys related to a specific library reside on the same shard. This strategy is particularly advantageous as it preserves atomic operations for book reservations and loans, which involve multiple Redis structures such as availability counters, loan hashes, and reservation hashes that must be modified atomically within the same transaction.[4]

However, in a sharded environment, the user activity hash structure, which tracks active reservations and loans, could no longer be guaranteed to reside on a single shard, especially when users interact with multiple libraries. To ensure consistency

[4]Redis guarantees atomic execution of commands within a single transaction when using the `MULTI/EXEC` mechanism, and `WATCH` for optimistic locking, as long as all involved keys are located on the same Redis instance.

in a sharded Redis environment, the system could persist user activity to MongoDB asynchronously. This approach leverages an event-driven architecture where updates are processed in a strictly ordered manner, preventing inconsistencies that may arise due to out-of-order execution. For instance, if a user books a book and immediately cancels the reservation, it is crucial to ensure that, on Mongo, the cancellation event does not get processed before the initial reservation is recorded. By implementing idempotent processing and event ordering guarantees, the system ensures that updates are applied in the correct sequence, even if events are handled asynchronously.

Regarding Redis Streams, while they can function in a sharded environment, they require careful partitioning to ensure efficient event processing. An alternative approach could involve replacing streams with sorted sets, maintaining separate event queues per shard. This approach would enable scalable event-driven processing without introducing inter-shard dependencies that could degrade performance.

### MongoDB Sharding Strategy

Given the read-heavy nature of the system, sharding MongoDB is not only unnecessary but could also introduce inefficiencies. Sharding is typically required when write throughput becomes a bottleneck, as it allows distributing the write load across multiple nodes. However, since MongoDb in our system primarily serves read operations, replica sets provide a more suitable approach, offering high availability and load balancing for read queries without the added complexity of data partitioning. Given the system's read-heavy nature, scaling can be more efficiently achieved by adding additional replica nodes that store the full dataset, rather than distributing data across shards. This approach ensures that more queries can be handled simultaneously while maintaining data consistency and minimising architectural complexity.

Furthermore, sharding could degrade performance for certain read queries. For instance, a title-based search that returns multiple books could require fetching data from multiple shards, significantly increasing query latency.[5] Instead of retrieving results from a single, well-indexed replica set, the database engine would need to query multiple shards, merge partial results, and reassemble the dataset, introducing unnecessary overhead. As a result, unless dataset size exceeds the storage capacity of a single server, maintaining a replicated rather than sharded architecture remains the optimal approach.

However, should the dataset grow beyond the capacity of a single server, sharding could be implemented based on a hashed field, such as book titles or other indexed attributes, to achieve even data distribution across shards. While this would enhance scalability, it would also introduce challenges, particularly for analytical queries that involve unwind operations on embedded documents. These operations would require cross-shard aggregations, potentially leading to significant performance overhead.

To address this, the system could adopt a hybrid approach where OLAP (On-

[5]This limitation could only be overcome if we could accurately predict which records would be returned together for any given query, allowing them to be placed on the same shard. However, achieving such predictive co-location is impractical for text-based searches on book titles, as it's virtually impossible to anticipate all potential search patterns and their corresponding result sets. This challenge becomes even more pronounced if an external search engine like Elasticsearch were to be integrated, as it would operate with entirely separate indexing and partitioning mechanisms.

line Analytical Processing) servers are introduced for complex analytical queries. This approach would allow pre-aggregating analytical data in advance, reducing the complexity of querying fragmented datasets across multiple shards. By preparing dedicated data structures optimised for analytical processing, the system can efficiently handle large-scale statistical queries without requiring expensive real-time cross-shard operations. This ensures that analytics can be performed on consolidated datasets, avoiding the performance degradation that would arise from having data scattered across multiple servers.

The trade-off would be a slight delay in analytical results, as they would be computed asynchronously rather than in real time. However, given that most analytical use cases do not require immediate results, this compromise would be acceptable in exchange for significantly improved system efficiency.

# Chapter 2

# Implementation

This chapter details the technical implementation of Distribooked, outlining the core software modules and architectural organisation of the system.

The chapter begins by presenting the high-level software modules that form the foundation of the system, followed by an in-depth examination of the package structure and implementation details.

The GitHub repository containing all the project files can be found here.

## 2.1 Technical Implementation

The system was implemented as a Spring Boot application using Java 17 and Maven for dependency management and build automation. It integrates MongoDB as a Document Database for storing structured entities and Redis as a Key-Value Store to handle temporary data and fast retrieval operations. The service exposes functionalities through RESTful APIs and ensures modularity through a layered architecture based on the Model-View-Controller (MVC) pattern, which separates concerns between data management, business logic, and request handling.

### 2.1.1 Outbox Pattern Architecture

In many scenarios in which consistency between different components of the distributed data layer must be maintained, the outbox pattern is implemented to ensure reliable processing of asynchronous tasks and maintain eventual consistency in the system. Its usage is particularly beneficial in scenarios where an operation involves multiple updates across different MongoDB collections or when there is a need to propagate changes from Redis to MongoDB efficiently.

The Outbox pattern is a distributed systems design technique that addresses the challenges of maintaining data consistency across multiple services or databases. In essence, it works by first storing planned database operations in a separate collection before actually executing them. This approach ensures that even if a system component fails, the pending operations can be reliably tracked, retried, and eventually completed, preventing data inconsistencies and providing a robust mechanism for asynchronous processing.

When an operation requires modifications in multiple MongoDB collections, the initial update is applied immediately, while subsequent updates – such as ensuring consistency between embedded documents – are scheduled as outbox tasks. This allows for eventual consistency in cases where a slight delay in synchronisation is acceptable. Similarly, when updates are first performed on Redis for performance reasons or because Redis is source of truth for the involved piece of data, the outbox pattern ensures that these changes are later propagated to MongoDB without requiring immediate dual-access to both databases, thereby preserving Redis's speed advantage. In such cases, data modifications are initially recorded in Redis streams or sorted sets (ZSets), from which specialised workers retrieve the relevant information and generate corresponding outbox tasks in MongoDB (cf. Section 2.2.3.1).

The `OutboxWorker` (cf. Section 2.2.3.1) is responsible for periodically retrieving and processing these pending tasks, ensuring their execution even in the event of failures. If a task encounters an error, it is retried with an exponential backoff strategy to prevent unnecessary system load. Once successfully completed, the task status is updated to `COMPLETED`. Additionally, any tasks that remain in progress for an unusually long time are automatically reset and reprocessed.

Each outbox task represents a distinct operation that needs to be processed asynchronously. Below is an example of a document stored in the `outbox` collection, illustrating the structure of a task related to book availability updates:

```
{
    "_id": "679bb09e4a78c01e693af661",
    "type": "DECREMENT_BOOK_COPIES",
    "payload": {
        "libraryId": "679a76df5cf745180198d68e",
        "bookId": "679bafe94a78c01e693af65e"
    },
    "status": "COMPLETED",
    "retryCount": 0,
    "createdAt": "2025-01-30T17:02:22.532Z",
    "updatedAt": "2025-01-30T17:02:22.532Z",
    "nextRetryAt": "2025-01-30T17:02:22.532Z",

}
```

**Code snippet 2.1:** Example of an Outbox Document.

This document includes a unique identifier (`_id`) for the task, a task type (`type`) indicating the nature of the operation, and a payload containing relevant parameters needed for processing. Each type of outbox task has its own payload structure, tailored to the specific requirements of executing the associated task.

It also features a status field (`status`) to track the progress of the task. The possible values for this field are: `PENDING` (waiting to be processed), `IN_PROGRESS` (currently being executed), `COMPLETED` (successfully finished), `FAILED` (execution failed five times), and `RETRY_SCHEDULED` (queued for a retry after failure).

Additionally, timestamps are maintained for tracking creation, updates, and retry attempts. In case of failures, the document also records an error message and stack trace, providing useful diagnostic information.

This architecture enhances system reliability by ensuring that all scheduled operations are eventually executed, even in case of unexpected failures. It also enables robust failure recovery through structured retries while preserving database consistency, preventing discrepancies between persisted data and asynchronously processed events.

A brief overview of the key components was provided here, but detailed explanations are available in their respective sections. The primary components involved in this mechanism include the `OutboxService` (cf. Section 2.2.3), which handles the creation and management of outbox tasks, and the `OutboxWorker` (cf. Section 2.2.3.1), responsible for their periodic retrieval and execution. Each task is categorised by type, and the worker invokes the corresponding `OutboxTaskProcessor` (cf. Section 2.2.3.2), which implements the specific logic required for handling that particular task type.

Furthermore, in the already mentioned Section 2.2.3, a practical example of the Outbox pattern can be found, illustrating the complete workflow of a typical use case: the case of a loan request, where we can observe the process from updating book availability in Redis to creating and processing the corresponding outbox task in the MongoDB collection.

## 2.2 Package and Class Organisation

The application's codebase follows a structured package organisation under the root package `it.unipi.distribooked`, following a layered architecture to ensure modularity and maintainability. Each subpackage encapsulates a specific aspect of the system, with distinct sets of functionalities. The following subsections describe the key packages and their primary responsibilities.

### 2.2.1 Configuration (`.config`)

The `config` package (`it.unipi.distribooked.config`) contains configuration classes for system-wide settings, such as security, caching, and API documentation. The main configuration classes are:

— `AppConfig`: Provides the base configuration of the Spring Boot application. It enables component scanning for the `mapper, service, and repository` layers, ensuring proper dependency injection.

— `MongoConfig`: Configures the connection and interactions with `MongoDB`. This class reads the database URI from the application properties, defines a `MongoClient` bean to establish a connection, and provides a `MongoTemplate` bean for performing operations on the database.

— `RedisConfig`: Sets up `Redis` as a Key-Value store. It reads connection properties (host, port, password, database index), defines a connection pool with optimised settings, configures a `RedisTemplate` for structured operations on Redis, and enables transaction support and key serialisation for better data consistency.

— `SecurityConfig`: Implements authentication and authorisation using `Spring Security`. This class defines security policies for JWT-based authentication, enables method-level security, configures public and protected API endpoints, and implements a custom JWT encoder/decoder for token management.

  JWT tokens are generated upon successful login and returned to the client. Each token includes several claims: the subject (`sub`), which corresponds to the username of the authenticated user; the user ID (`user_id`), representing the unique identifier of the user; and the roles (`roles`), which define the user's access permissions. Additionally, each token contains an issued-at timestamp (`iat`) indicating when the token was generated and an expiration timestamp (`exp`) that defines its validity period, which is set to one hour by default.

These tokens are self-contained and must be included as a `Bearer` token in the `Authorization` header of each request to access protected endpoints.

– `SwaggerConfig`: Configures `Swagger/OpenAPI` for API documentation. It defines a bearer authentication scheme for secured endpoints, customises API metadata (title, description, version), and ensures Swagger UI resources are properly served.

### 2.2.2 Controller

The `controller` package (`it.unipi.distribooked.controller`) contains classes responsible for handling HTTP requests and providing endpoints for different functionalities of the system. Further details on the exposed endpoints can be found in Section 2.6.

The `AuthController` class is located directly under the `controller` package and is responsible for handling authentication-related endpoints. It provides functionalities for user registration, login, and logout.

#### 2.2.2.1 Open Controllers (`controller.open`)

The `open` subpackage contains controllers that provide publicly accessible endpoints without authentication requirements.

– `AuthorController`: Manages author-related endpoints, including searching for authors by name and retrieving detailed author information.

– `BookController`: Provides endpoints for browsing the book catalogue, searching for books by title and author, and checking book availability in libraries.

– `LibraryController`: Handles endpoints for retrieving details of specific libraries.

The following code snippet provides an example of a RESTful API endpoint implemented within the `BookController` class. This method retrieves a paginated list of books from the catalogue and returns it as a structured response. The implementation follows standard REST conventions, leverages Spring Boot features for request handling, and applies pagination to optimise data retrieval.

```java
@RestController
@RequestMapping("/api/books")
@Tag(name = "Books", description = "Endpoints for browsing and
    retrieving book details.")
public class BookController {

    @Autowired
    private BookService bookService;

    @Operation(summary = "Browse book catalog", description =
        "Retrieve a paginated list of books.")
    @GetMapping
    public ResponseEntity<Map<String, Object>> getBooks(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "20") @Max(100) int size) {

        Pageable pageable = PageRequest.of(page, size);
        Page<BookCatalogueDTO> books =
            bookService.getAllBooks(pageable);

        return ApiResponseUtil.ok("Books retrieved successfully",
            books);
    }
}
```

**Code snippet 2.2:** Example of a RESTful API endpoint implementing pagination. Note that the code shown here is a simplified version of the actual implementation: extensive Swagger annotations for API documentation, logging statements, additional input validation, and several Spring framework annotations have been omitted to enhance readability whilst preserving the core architectural patterns. The complete implementation can be found in the project repository.

This example demonstrates the core structure of a Spring Boot controller responsible for handling book-related requests. The class is annotated with `@RestController`, which declares it as a Spring MVC controller, enabling automatic serialisation of returned objects into JSON. The `@RequestMapping("/api/books")` annotation defines the base URL for all endpoints within this controller.

To improve API documentation, the class includes Swagger annotations: `@Tag(name = "Books", description = "...")` groups API endpoints within the generated documentation, while `@Operation(...)` provides metadata describing the purpose of the method. The `@GetMapping` annotation maps HTTP `GET` requests to the `getBooks` method.

The method parameters are extracted from query parameters using `@RequestParam`. The `page` parameter defaults to `0`, meaning the first page is retrieved if no value is provided, whereas the `size` parameter defaults to `20` and is limited to a maximum of `100` using the `@Max(100)` annotation, preventing excessive load.

The actual data retrieval logic is delegated to the service layer via the `bookService.getAllBooks(pageable)` call. The method then returns a standardised response using `ApiResponseUtil.ok(...)`, ensuring uniform response formatting across the application (cf. section 2.2.9 for further details on the `ApiResponseUtil` class.).

Further details on the API design, including a comprehensive list of available endpoints, can be found in Section 2.6.

The subsequent sections regarding controllers do not include additional code examples for other endpoint implementations, as the overall architecture remains consistent across controllers. Variations primarily concern the type of operation performed – whether it is a retrieval (`@GetMapping`), creation (`@PostMapping`), update (`@PutMapping`), partial update (`@PatchMapping`), or deletion (`@DeleteMapping`)– as well as the way parameters are passed, which may involve query parameters (`@RequestParam`), request payloads (`@RequestBody`), or path variables (`@PathVariable`).

#### 2.2.2.2 Secured Controllers (`controller.secured`)

The `secured` subpackage contains controllers that provide endpoints requiring user authentication.

- `ReservationController`: Manages book reservations, allowing users to reserve, cancel, and manage their book reservations.
- `UserController`: Handles user-specific functionalities, such as retrieving reserved, loaned, saved, and read books.

#### 2.2.2.3 Restricted Controllers (`controller.restricted`)

The `restricted` subpackage contains controllers that manage administrative operations, requiring administrative privileges for access.

- `CatalogueManagementController`: Manages the administration of the book catalogue, allowing books to be added, updated, or removed.
- `LoanManagementController`: Provides administrative endpoints for loan management, such as converting reservations into loans, completing loans, and retrieving overdue loans.
- `StatisticsController`: Exposes endpoints through which administrators can access statistics and analytics regarding libraries, book readings, users, and other relevant metrics. More details on the available statistics are provided in section 2.3.

### 2.2.3 Service

The `service` package (`it.unipi.distribooked.service`) implements the business logic and orchestrates interactions between controllers and repositories, delegating database access to the repository layer. It contains services responsible for user authentication, book and author management, library handling, and various other functionalities.

- `AuthorService`: Handles operations related to authors, such as searching for authors by name, retrieving author details, and fetching books written by a specific author.
- `AuthService`: Manages user authentication and registration. It provides functionalities for validating login credentials and generating JWT tokens for authenticated users.
- `BookService`: Implements the business logic for book-related operations, including searching books by title or author, retrieving book details, and checking book availability in libraries.
- `CatalogueManagementService`: Responsible for managing the book catalogue, allowing administrators to add, update, and remove books while ensuring data consistency.

- `LibraryService`: Handles library-related operations such as retrieving library details and managing library records in the system.
- `LoanManagementService`: Provides functionalities for managing book loans and reservations, including converting reservations into loans and handling overdue loans.
- `OutboxService`: Manages asynchronous outbox tasks, ensuring reliability in handling delayed operations and system-wide event processing (cf. Section 2.2.3.2 for further details).
- `ReservationService`: Implements logic for book reservations and saved book lists, allowing users to reserve books and manage their saved collections.
- `TokenService`: Handles JWT token generation, ensuring secure authentication and authorization across the system.
- `UserService`: Provides user-related operations, including retrieving user details, fetching reserved and loaned books, and managing saved and read books.

In addition to the main service classes described in this section and the subpackages detailed in the following sections, the `service` package also includes the `service.notification` subpackage. The `service.notification` subpackage is included as a logical placeholder for future implementations of notification-related services. Since the system is not intended for production deployment at this stage, no concrete notification mechanism has been implemented.

#### 2.2.3.1 Worker (`service.worker`)

The `service.worker` subpackage contains background worker components that process asynchronous tasks using Redis streams, Redis sorted sets (ZSets), and MongoDB outbox collection. These workers ensure that system updates and events are handled reliably, even in case of failures.

**Workers processing Redis Streams**

These workers listen to Redis streams, process incoming messages, and generate corresponding outbox tasks to ensure the consistency of operations. Redis Streams were chosen for their persistence guarantees, their ability to ensure messages are consumed in order, and their built-in consumer group functionality, which allows reliable message delivery.

Once an outbox task is successfully created based on a stream message, the worker sends an acknowledgment to the Redis stream to mark the message as processed. This approach ensures asynchronous execution while maintaining robustness. In the event of an application crash before the acknowledgment is sent, the message remains in the stream and will be reprocessed upon recovery.[1]

[1]Reprocessing the same message may lead to duplicate outbox tasks, but this is not an issue as the task processors (cf. Section 2.2.3.2) are implemented to be idempotent, ensuring that duplicate processing does not cause inconsistencies.

- `AddLibraryToBookWorker`: Processes messages related to adding library entries for books and creates corresponding outbox tasks.
- `CompletedLoanWorker`: Detects completed book loans and triggers the update of user reading history via an outbox task.
- `DecrementCopiesWorker`: Handles book loan operations by decrementing the available copies in Redis and ensuring persistence in MongoDB.
- `IncrementCopiesWorker`: Listens for book copy increment requests and ensures the changes are propagated correctly.
- `RemoveLibraryWorker`: Processes requests to remove library entries from books and schedules the necessary updates.

### Workers processing Redis Sorted Sets

These workers periodically scan Redis sorted sets to detect expired events and schedule the appropriate actions. Sorted sets provide the advantage of allowing efficient range queries based on their scores (see Redis Streams Documentation). In this implementation, the score associated with each entry represents the expiration timestamp. Workers can efficiently retrieve only the elements whose expiration timestamp is lower than the current time, ensuring that only relevant expired entries are processed.

This approach is particularly useful in scenarios where business logic requires specific actions to be taken after a set time has elapsed. For example, when a user's reservation period for a book (three days) expires, the system must increment the book's availability count at the respective library. Similarly, when a book loan period (thirty days) expires, the system triggers further actions, such as marking the loan as overdue[2] .

- `ExpiredLoanWorker`: Identifies overdue loans and marks them as such by creating an outbox task.
- `ExpiredReservationWorker`: Detects expired reservations and updates book availability accordingly.

### Worker processing MongoDB Outbox Collection

This worker periodically processes tasks stored in the MongoDB outbox collection, ensuring their execution and retrying failed operations when necessary.

- `OutboxWorker`: Handles pending, scheduled retries, and stuck tasks in the outbox collection, ensuring all background tasks are properly processed.

The following code snippet illustrates the implementation of the `OutboxWorker`, which retrieves pending tasks, processes them, and handles retry mechanisms for failed or stuck tasks.

[2]The system does not rely on Redis notifications triggered when a key's TTL expires, as these messages are not guaranteed to be delivered. If Redis crashes near the expiration time, the notification may never be sent, potentially leading to missed updates.

```java
@Component
@RequiredArgsConstructor
@Slf4j // lombok's logger tag
public class OutboxWorker {

    private final OutboxService outboxService;
    private final OutboxRepository outboxRepository;

    @Scheduled(fixedDelay = 5000) // Runs every 5 seconds
    public void processOutboxTasks() {
        LocalDateTime now = LocalDateTime.now();

        // Process pending tasks
        List<OutboxTask> pendingTasks =
            outboxRepository.findProcessableTasks(TaskStatus.
            PENDING, now);
        pendingTasks.forEach(outboxService::processTask);

        // Process retry-scheduled tasks
        List<OutboxTask> retryScheduledTasks =
            outboxRepository.findProcessableTasks(TaskStatus.
            RETRY_SCHEDULED, now);
        retryScheduledTasks.forEach(outboxService::processTask);

        // Identify and reset stuck tasks
        LocalDateTime stuckThreshold = now.minusMinutes(5);
        List<OutboxTask> stuckTasks =
            outboxRepository.findStuckTasks(stuckThreshold);
        stuckTasks.forEach(this::resetStuckTask);
    }

    private void resetStuckTask(OutboxTask task) {
        log.warn("Found stuck task {}, resetting status to
            RETRY_SCHEDULED", task.getId());
        task.setStatus(TaskStatus.RETRY_SCHEDULED);
        task.setNextRetryAt(LocalDateTime.now().plusMinutes(1));
        outboxRepository.save(task);
    }
}
```

**Code snippet 2.3:** Implementation of the `OutboxWorker`, responsible for processing pending and stuck outbox tasks.

This implementation ensures that all tasks stored in the outbox collection are processed reliably. If a task fails, it is scheduled for retry using an exponential backoff strategy. Additionally, tasks that remain in progress for too long (stuck tasks) are automatically reset and reprocessed.

In the previously shown code, methods of the `OutboxService` class (already mentioned in Section 2.2.3) are frequently invoked to handle task execution, retries, and status updates. In the following, further details on its key functionalities are provided.

Before processing a task, its status is updated to `IN_PROGRESS` to prevent concurrent execution:

```java
public void processTask(OutboxTask task) {
    task.setStatus(TaskStatus.IN_PROGRESS);
    outboxRepository.save(task);

    // Retrieve processor and execute task
    OutboxTaskProcessor processor =
        processors.get(task.getType().name());
    processor.process(task);
}
```

**Code snippet 2.4:** Marking a task as `IN_PROGRESS` before execution.

If a failure occurs, the task is updated with the error message, and the retry logic schedules its next attempt:

```
task.setRetryCount(task.getRetryCount() + 1);
task.setNextRetryAt(LocalDateTime.now().plus(calculateBackoff(task.
    getRetryCount())));
task.setStatus(TaskStatus.RETRY_SCHEDULED);
outboxRepository.save(task);
```

**Code snippet 2.5:** Scheduling a retry with exponential backoff.

The `calculateBackoff` method implements an exponential backoff strategy with jitter, ensuring that retry intervals increase progressively to avoid excessive system load. The retry delay doubles with each attempt, preventing frequent retries from overwhelming the system, but is capped at a predefined maximum to avoid excessively long wait times. A jitter component is introduced by adding a small random variation to the computed delay, which prevents multiple tasks from retrying simultaneously, reducing the risk of synchronised retry spikes. If a task exceeds the maximum number of allowed retries, it is marked as `FAILED`, ensuring that persistent failures do not remain indefinitely in the system.[3]

Further details on the `OutboxService` implementation, including complete retry logic and error handling, can be found in the source repository.

#### 2.2.3.2 Outbox Processors (`service.outbox`)

The `service.outbox` subpackage contains the implementations of the outbox pattern processors. These processors are responsible for executing the tasks stored in the `OutboxTask` collection, ensuring reliable and asynchronous execution of database updates.

At the core of this mechanism lies the `OutboxTaskProcessor` interface, which defines the contract for all outbox processors. Each processor implements this interface and overrides the `process` method, which is responsible for executing the corresponding update operation.

```
public interface OutboxTaskProcessor {
```

[3]Under normal conditions, outbox tasks are not expected to fail. The retry mechanism primarily addresses temporary failures, such as a crash of the primary MongoDB or Redis instance, allowing sufficient time for a secondary instance to take over as the new primary. However, if a task consistently fails after multiple retries, this suggests a more serious issue that requires manual intervention. Persistent failures may indicate underlying data consistency problems that cannot be resolved automatically, necessitating further investigation to prevent potential system inconsistencies.

```java
    void process(OutboxTask task) throws TaskExecutionException;
}
```

**Code snippet 2.6:** Definition of the OutboxTaskProcessor interface.

For each type of update operation, there is a dedicated class that implements `OutboxTaskProcessor`. These processors handle a variety of tasks, such as updating book availability, managing overdue loans, propagating library changes, and synchronising embedded documents. The `OutboxWorker` (cf. Section 2.2.3.1) invokes the appropriate processor based on the task type, ensuring the correct execution of the pending operations.

A full list of processors is omitted here, however, their implementations can be found in the source repository. As with all service-layer classes, these processors delegate database access to the repository layer, ensuring a clear separation of concerns.

### 2.2.4 Repository

The `repository` package (`it.unipi.distribooked.repository`) encapsulates database access logic, providing interfaces for CRUD operations using Spring Data.

The package is structured into two subpackages: `repository.mongo` and `repository.redis`. The former contains all classes required for interacting with MongoDB, the latter provides components for managing Redis-based operations.

#### 2.2.4.1 Mongo Repository (`repository.mongo`)

The `repository.mongo` package provides repository interfaces that manage interactions with MongoDB using Spring Data Mongo. Database access is performed using the `MongoTemplate` defined in the `MongoConfig` class (cf. Section 2.2.1).

For simple queries that can be automatically derived by Spring Data MongoDB from method names or require minimal annotations, interfaces extending `MongoRepository` are used. The primary repository interfaces in this package include `AuthorRepository`, `BookRepository`, `LibraryRepository`, `OutboxRepository` and `UserRepository`. These repository interfaces specify the model class they manage (e.g. `Book` in `BookRepository`), allowing Spring Data MongoDB to infer both the target collection and the document structure for query execution (cf. Section 2.2.5).

An example of a repository method that is fully inferred by Spring Data MongoDB based on its method name is:

```java
@Repository
public interface BookRepository extends MongoRepository<Book,
    ObjectId> {

// Spring will automatically generate the query based on method
    name
Page<BookCatalogueView> findByTitle(String title, Pageable
    pageable);
}
```

**Code snippet 2.7:** Spring Data MongoDB automatically derived query. Notice how the repository interface specifies the model class (`Book`) and the type of the primary key of the documents (`ObjectId`) as generic parameters of `MongoRepository`, enabling Spring to determine the corresponding MongoDB collection and document structure (cf. Section 2.2.5).

In some cases, as illustrated in code snippet 2.7, projection interfaces were utilised to reduce the amount of data retrieved from MongoDB. These projection interfaces, located in `repository.mongo.views`, define getter methods corresponding to the fields of MongoDB documents as inferred from the model classes (cf. Section 2.2.5). This approach limits the transferred data.

An example of projection interface is provided in code snippet 2.8:

```java
public interface BookCatalogueView {
    ObjectId getId();
    String getTitle();
    String getSubtitle();
    List<EmbeddedAuthor> getAuthors();
    List<String> getCategories();
    String getCoverImageUrl();
}
```

**Code snippet 2.8:** Projection interface for author search.

For more complex queries that cannot be inferred directly by Spring from method names or require specialised logic, custom repository interfaces were defined in `repository.mongo.custom`. These interfaces declare advanced query methods, then the actual implementations of these methods are provided in `repository.mongo.custom.impl`. These implementations use `MongoTemplate` to construct and execute custom queries, ensuring flexibility in handling complex operations.

An example of a custom repository implementation using `MongoTemplate` is presented below. This approach consists of three main components. First, a custom repository interface is defined to specify the method signature (code snippet 2.9). Then, a custom repository implementation leverages `MongoTemplate` to execute the query and perform database operations (code snippet 2.11). Finally, the primary repository interface extends the custom repository interface (code snippet 2.10), ensuring seamless integration with standard repository operations provided by Spring Data MongoDB.

```java
public interface CustomAuthorRepository {
    boolean updateAuthorWithBook(ObjectId authorId,
        EmbeddedBookAuthor book);
}
```

**Code snippet 2.9:** Custom repository interface for complex MongoDB operations.

```java
@Repository
public interface AuthorRepository extends MongoRepository<Author,
    ObjectId>, CustomAuthorRepository {

    // Other queries automatically derived by Spring Data MongoDB
    // ...
}
```

**Code snippet 2.10:** Extension of custom repository interface in the main repository.

```java
@Repository
public class CustomAuthorRepositoryImpl implements
    CustomAuthorRepository {

    @Autowired
    private MongoTemplate mongoTemplate;

    @Override
    public boolean updateAuthorWithBook(ObjectId authorId,
        EmbeddedBookAuthor book) {
        // Specific implementation
        // ...
    }
}
```

**Code snippet 2.11:** Custom repository implementation using MongoTemplate.

This layered approach ensures flexibility and efficiency in handling MongoDB queries. Simple queries can be managed directly within the main repository interface, whereas more complex operations leverage custom implementations to achieve atomic updates while maintaining data integrity.

#### 2.2.4.2 Redis Repository (`repository.redis`)

The `repository.redis` package provides repository classes that handle interactions with Redis.

The `LoanRepository` is responsible for managing loan-related data in Redis, while the `OverdueLoanRepository` specifically tracks overdue loans. The `RedisBookRepository` handles book-related operations, and the `RedisUserRepository` manages user-related loan and reservation information, enabling fast access to frequently queried user data. Finally, the `ReservationRepository` maintains reservation records, ensuring that book reservations are efficiently processed and tracked within the Redis store.

A key feature of this implementation is the use of Lua scripts to enforce atomicity and transactional-like operations in Redis. These scripts allow multiple operations to be executed as a single, indivisible transaction, preventing race conditions that could arise in a highly concurrent environment. In this system, concurrent access must be carefully managed to prevent inconsistencies when users simultaneously reserve books, loan them, and update their availability. For instance, without atomic operations, two users could attempt to reserve the last available copy of a book at the same time, leading to incorrect availability counts.[4]

To improve efficiency, Lua scripts are loaded into Redis at application startup using the `SCRIPT LOAD` command, which compiles the script and stores it in Redis's internal script cache, indexed by its SHA-1 hash. Once loaded, the script can be executed multiple times using `EVALSHA`, which references the cached version by its SHA-1 digest instead of resending the entire script. This approach reduces network overhead, eliminates the need for recompilation on every execution, and ensures

[4]For more details on the role of Lua scripting in Redis and its atomic execution properties, refer to the official Redis documentation: `https://redis.io/docs/latest/develop/interact/programmability/eval-intro/`.

consistent performance. By keeping scripts generic and parameterised through arguments, the system avoids excessive memory usage that could arise from dynamically generating scripts at runtime.

Lua scripts are stored in the Maven project under `resources/scripts`, as it is the standard directory for static resources.

### Best Practice in Lua Scripting

Redis enforces specific best practices when using Lua scripts to ensure correct execution. A key recommendation is:

> *Important: to ensure the correct execution of scripts, both in standalone and clustered deployments, all names of keys that a script accesses must be explicitly provided as input key arguments. The script should only access keys whose names are given as input arguments. Scripts should never access keys with programmatically-generated names or based on the contents of data structures stored in the database.*[5]

This ensures that Redis can properly track key access patterns.

### Example of a Lua Script

The following Lua script demonstrates a key advantage of Lua scripting in Redis: the ability to perform conditional operations based on database state within an atomic context. This pattern is particularly important when business logic requires reading values before deciding whether to proceed with modifications:

```lua
-- Keys: user activity key, library loans key, library overdue key
local userResLoansKey = KEYS[1]
local libraryLoansKey = KEYS[2]
local libraryOverdueKey = KEYS[3]
-- Check if loan exists in either loans or overdue hashes
local inLoans = redis.call('HEXISTS', libraryLoansKey,
    libraryField)
local inOverdue = redis.call('HEXISTS', libraryOverdueKey,
    libraryField)
if inLoans == 0 and inOverdue == 0 then
    return cjson.encode({["err"] = "Loan not found"})
end
-- Conditional write operations based on read results
if inLoans == 1 then
    redis.call('HDEL', libraryLoansKey, libraryField)
else
    redis.call('HDEL', libraryOverdueKey, libraryField)
end
```

**Code snippet 2.12:** Excerpt from the loan completion script: demonstrating atomic read-then-write pattern when an administrator marks a book as returned. The script first determines whether the loan is active or overdue before performing the appropriate deletion.

While `MULTI/EXEC` transactions with `WATCH` could theoretically achieve similar results in some cases, Lua scripts were chosen as the consistent approach throughout

[5]Redis scripting best practices: from the same Redis Lua scripts documentation previously cited.

the project. This decision was made for several reasons: firstly, the operations in our system are typically brief and well-defined, making them ideal candidates for Lua scripting; secondly, Lua scripts provide a more straightforward implementation pattern compared to managing `WATCH` commands and their potential retries. Moreover, Lua scripts enable complex conditional logic based on read operations within the same atomic context – a capability not possible with `MULTI/EXEC` transactions, where decisions cannot be made based on values read within the transaction block. Additionally, maintaining a consistent approach across the codebase enhances maintainability and reduces cognitive overhead for developers working with the system.

Once a Lua script is loaded into Redis (as previously described), it can be executed efficiently using its SHA-1 hash via the `EVALSHA` command. The following Java method demonstrates how this is done in the system when completing a loan operation.

```java
Object result = jedis.evalsha(
    completeLoanScriptSha,
    List.of(
        RedisKey.USER_ACTIVITY.getKey(userId),
        RedisKey.LIBRARY_LOANS.getKey(libraryId),
        RedisKey.LIBRARY_OVERDUE.getKey(libraryId),
        RedisKey.LOAN_ZSET_EXPIRATION.getKey(),
        RedisKey.BOOK_AVAILABILITY.getKey(bookId, libraryId),
        RedisKey.USER_HASH_ENTRY.getKey(libraryId, bookId),
        RedisKey.LIBRARY_HASH_ENTRY.getKey(userId, bookId),
        RedisKey.COMPLETED_LOANS_STREAM.getKey()
    ),
    List.of(userId, bookId, libraryId)
);
```

**Code snippet 2.13:** Executing a Lua script with `EVALSHA` in Redis. The first list contains the Redis keys accessed by the script, while the second list contains additional arguments passed to the script. The structure of the Redis keys follows the patterns defined in the `RedisKey` class (cf. Section 2.2.9).

### Redis Hash Field TTLs

Another notable feature in this implementation is the use of a new Redis capability introduced very recently, which allows setting per-field TTLs in Redis hashes[6]. This functionality is particularly useful for book reservations, which automatically expire after three days if the user does not collect the book.

### Transactional Guarantees of Lua Scripts

Lua scripts in Redis provide strong transactional guarantees, ensuring consistency and durability even in case of server failures. This is achieved through *script effects replication*, a mechanism introduced in Redis 3.2 and enabled by default from Redis 5.0:

> *Starting with Redis 5.0, script effects replication is the default mode and does not need to be explicitly enabled.*

[6]This implementation makes use of `HPEXPIRE`. More details can be found in the official Redis documentation: `https://redis.io/docs/latest/commands/hpttl/`.

> *In this replication mode, while Lua scripts are executed, Redis collects all the commands executed by the Lua scripting engine that actually modify the dataset. When the script execution finishes, the sequence of commands that the script generated are wrapped into a MULTI/EXEC transaction and are sent to the replicas and AOF.* [7]

This mechanism ensures that if a script fails, no modifications are applied, preserving atomicity. Write commands are persisted to the Append-Only File (AOF) only if the script completes successfully, preventing partial updates in case of a crash. Replicas receive only finalised write commands, guaranteeing consistency across the system.

If Redis crashes during script execution, incomplete modifications are never propagated, and upon restart, the AOF restores only fully committed changes. If the script completes, all modifications are safely persisted and replicated. This approach ensures that Redis maintains a consistent state, even in failure scenarios, making Lua scripts a reliable solution for transactional operations in a highly concurrent environment.

#### Examples of Critical Redis Workflows

To illustrate the interaction between Redis data structures and Lua scripting, this section presents two fundamental operations: *book reservation* and *reservation conversion to loan*.

When a user reserves a book, a Lua script ensures atomic execution of all required updates. The system first verifies whether the user has already an active reservation/loan of the same book at the given library, rejecting the request if such a reservation exists. It then checks whether the user has reached the maximum allowed concurrent reservations/loans and confirms whether the book is available at the specified library. If all conditions are met, the script proceeds with the reservation: the book's availability counter is decremented, a structured JSON entry representing the reservation is added to the user's activity hash, and a corresponding entry is inserted into the library's reservation hash. Additionally, expiration times are set on the reservation entries using Redis hash field TTLs (cf. Section 2.2.4.2), ensuring automatic cleanup upon expiration. Finally, the reservation's expiration timestamp is recorded in a sorted set to allow background processes to track and handle expired reservations.

Cancelling a reservation follows a similar atomic process. The system verifies whether a valid reservation exists for the user and removes it from both the user's activity hash and the library's reservation hash. The book's availability counter is then incremented, and the corresponding expiration entry is deleted from the sorted set tracking reservations. Since the Lua script executes atomically, if the cancellation request occurs concurrently with the TTL expiration, and the TTL has already expired when the script runs, no action is taken – the hash entries will be removed by the TTL mechanism itself, and the availability update will be handled by the worker process that processes expired entries from the sorted set. Conversely, if the cancellation succeeds, the sorted set entry is also removed atomically, ensuring that the worker does not mistakenly increment the availability twice.

[7] Redis scripting behavior in replication: `https://redis.io/docs/latest/develop/interact/programmability/eval-intro/`.

The transition from a reservation to a loan is also managed atomically. When a user checks out a reserved book, the Lua script first confirms that the reservation exists. It then updates the JSON metadata within the user's activity hash, changing the status from `RESERVED` to `LOANED` and setting a new expiration deadline that corresponds to the loan duration. The system also removes the reservation entry from the sorted set tracking pending reservations and adds a new entry to the sorted set tracking active loans. The reservation is removed from the library's reservations hash and added to the library's active loans hash. As with reservations, Redis hash field TTLs are applied to loan entries to ensure automatic cleanup when the loan period expires.

By leveraging Redis hash field TTLs, sorted sets, and Lua scripting, the system maintains strong consistency guarantees while allowing efficient management of book reservations and loans. Atomic execution ensures that no inconsistencies arise due to concurrent requests, preventing issues such as duplicate reservations or incorrect book availability counts.

Although multiple operations are performed to meet these business requirements, their number remains limited, and their execution is highly efficient due to Redis's in-memory nature and the atomicity of Lua scripts. Furthermore, these are among the most critical operations in the entire system, as they require strict race condition prevention, making this trade-off in operation count a natural and necessary design choice.

All other Redis operations in the system follow this same structured approach, leveraging atomic execution and efficient data structures to ensure consistency and correctness across every interaction managed through the database.

### 2.2.5 Model

The `model` package (`it.unipi.distribooked.model`) defines the domain entities. The majority of these classes map the documents stored in MongoDB collections, as described in Section 1.4.2.1. Each class represents a structured entity within the database and is managed using Spring Data MongoDB to facilitate persistence operations. The `@Document(collection = "...")` annotation is applied at the class level to explicitly define the corresponding MongoDB collection. Thus, when a repository declares a model class as a generic parameter, Spring Data MongoDB infers both the structure of the documents and the target collection, enabling seamless database interactions.

For Redis, no dedicated model classes were created except for `UserBookActivity`, which is used in a Redis hash that tracks a user's active books (i.e. reservations and loans; cf. Section 1.4.3.1). In all other Redis entries used throughout the application, values are simple primitives or strings, eliminating the need for additional model classes.

In addition to these main entity classes, the package also contains several subpackages:

– `model.embedded`: Contains embedded value objects stored within MongoDB documents, such as `EmbeddedAuthor`, `EmbeddedLibrary`, and `EmbeddedBookRead`.

- `model.enums`: Defines enumerations used throughout the application, including `OutboxTaskType`, `TaskStatus`, and `UserType`, to ensure type safety and maintain consistency.

- `model.security`: Includes security-related models, such as `CustomUserDetails`, which extends Spring Security's user details abstraction to store authentication-related attributes.

### 2.2.6 DTO

The `dto` package (`it.unipi.distribooked.dto`) defines Data Transfer Objects (DTOs) used to structure data exchanged between different layers of the application. DTOs provide a means of decoupling internal data models from API responses and requests, ensuring that only relevant fields are exposed to clients.

Unlike model classes, which fully represent MongoDB documents, DTOs are designed to be more lightweight and optimised for specific use cases. For instance, `BookCatalogueDTO` includes only a subset of fields from the corresponding `Book` entity, as it is tailored to present book details efficiently when browsing the catalogue. Additionally, all DTOs include Swagger annotations to facilitate the automatic generation of API documentation.

The package also contains the `dto.swagger` subpackage, which defines standardised response wrappers used in API documentation. These classes, such as `SuccessResponse`, `CreatedResponse`, and `ErrorResponse`, ensure consistency in API responses and enhance Swagger documentation by explicitly describing the format of HTTP responses for different status codes.

### 2.2.7 Mapper

The `mapper` package (`it.unipi.distribooked.mapper`) contains mapping utilities that facilitate the conversion between domain entities and DTOs. This conversion ensures a clear separation between internal data representations and the data structures exposed via the API, enhancing modularity and maintainability.

Mapping operations are implemented using MapStruct, a compile-time code generation framework that automatically produces efficient mapping implementations. Unlike manual mapping approaches or reflection-based libraries, MapStruct generates type-safe, performant mapping methods at build time, reducing runtime overhead.

Each mapper interface is annotated with `@Mapper(componentModel = "spring")`, allowing Spring to manage and inject the generated mapping implementations as beans. Within these interfaces, mappings between entity fields and DTO fields are explicitly defined using `@Mapping` annotations. When necessary, custom transformation logic is applied through annotated methods using `@Named`.

The package contains multiple mappers, including:

- `AuthorMapper`: Converts `Author` entities and search views to their corresponding DTO representations.

- `BookMapper`: Handles transformations between `Book`, `EmbeddedBook`, and various book-related DTOs.

- `LibraryMapper`: Converts `Library` entities to DTOs while handling transformations of geospatial data such as `GeoJsonPoint`.
- `ReservationMapper`: Maps `Reservation` entities to DTOs and vice versa, ensuring proper conversion of MongoDB `ObjectId` fields.
- `UserMapper`: Handles conversions between user-related models, including security-related transformations for authentication and authorisation.

### 2.2.8 Exceptions

The `exceptions` package (`it.unipi.distribooked.exceptions`) centralises custom exception handling, ensuring uniform error management throughout the system. This package contains both domain-specific business exceptions and general error-handling mechanisms.

The `GlobalExceptionHandler` class serves as the central component for exception handling across the application. By extending Spring's `ResponseEntityExceptionHandler`, it intercepts exceptions at the controller level before they propagate to the client, eliminating the need for repetitive try-catch blocks in individual controllers. This approach significantly improves code readability and maintainability by moving error handling logic out of the business code. The handler leverages `ProblemDetail` to generate structured, standardised error responses that integrate seamlessly with OpenAPI documentation tools and include metadata such as error type, instance, and timestamp.

When an exception occurs during request processing, Spring automatically routes it to the appropriate handler method within `GlobalExceptionHandler`. The class provides specialised handling for different error categories: validation failures that occur when request parameters or bodies fail constraint checks, security-related issues such as authentication and authorisation failures, business logic violations like attempting to reserve an unavailable book, and database access problems including MongoDB connection issues or constraint violations. A fallback handler manages any unexpected exceptions not covered by specific handlers, ensuring all errors receive appropriate treatment.

Business-specific errors are encapsulated in custom exceptions that extend the `BusinessException` abstract class. This hierarchy creates a clear separation between domain-specific failures and technical errors, facilitating precise error handling and consistent response formatting.

### 2.2.9 Utilities (`.utils`)

The `utils` package (`it.unipi.distribooked.utils`) contains utility classes that provide reusable functionalities across different components of the system. These utilities standardise API responses, facilitate interaction with external resources, handle Redis key generation, enforce security constraints, and support various auxiliary operations.

- `ApiResponseUtil`: Provides a standard structure for all successful HTTP responses, ensuring uniformity across the system. The `ok` method formats responses with status 200, while the `created` method is used for status 201 responses.

— `LuaScriptLoader`: Loads Lua scripts stored in the `resources/scripts` directory, enabling execution of predefined Redis scripts from within the application.

— `ObjectIdConverter`: Provides a utility method to safely convert string representations of MongoDB `ObjectId`s into actual `ObjectId` instances, ensuring input validation and preventing malformed identifiers from being processed.

— `RedisKey`: Defines the set of Redis key patterns used throughout the system. Each key supports parameterisation via the `getKey` method, allowing dynamic generation of keys based on contextual parameters. For example, the `BOOK_AVAILABILITY` key pattern is defined as `"book:%s:lib:%s:avail"`, where `%s` placeholders are replaced with actual book and library identifiers at runtime. A call such as `RedisKey.BOOK_AVAILABILITY.getKey(bookId, libraryId)` dynamically generates a key like `"book:12345:lib:6789:avail"`, uniquely identifying the availability of a specific book in a given library.

— `SecurityUtils`: Extracts the authenticated user's ID from the JWT token provided in each request's `Authorization` header. This allows protected endpoints to infer the requesting user's identity without requiring them to explicitly pass their user ID as a parameter. The token is issued and signed upon successful authentication (cf. `SecurityConfig` in Section 2.2.1), and the user ID is extracted from the `user_id` claim within the token. This mechanism enables secure user identification while maintaining a stateless authentication model.

### 2.2.10 Validation

The `validation` package (`it.unipi.distribooked.validation`) implements custom validation logic through Jakarta Validation (JSR 380) integration in Spring Boot. This enables automatic validation of user inputs against defined business rules.

While Spring Boot provides standard validation annotations such as `@NotBlank`, `@Email`, and `@Size`, custom validation was required for MongoDB-specific constraints. The package introduces `@ValidObjectId`, a custom annotation that works with both fields and parameters. This annotation is backed by `ObjectIdValidator`, which implements `ConstraintValidator` to verify that strings match MongoDB's `ObjectId` format of 24 hexadecimal characters through regular expression validation.

The validation occurs automatically whenever the `@ValidObjectId` annotation is applied, ensuring invalid formats are rejected early in the request processing pipeline.

The following code snippet illustrates how `@ValidObjectId` is applied alongside Spring's built-in validation annotations in a REST endpoint.

```
@GetMapping("/{id}")
public ResponseEntity<Map<String, Object>>
    getBookByIdWithLibraries(
        @Valid @ValidObjectId @PathVariable String id,  // Custom
            validation for ObjectId
        @RequestParam(required = false) Double latitude,
        @RequestParam(required = false) Double longitude,
        @RequestParam(required = false) Integer radius) {
            // ...
}
```

**Code snippet 2.14:** Example of combined validation using Spring and a custom annotation in an endpoint that retrieves book details by ID, with optional location-based filtering of nearby libraries. The `@Valid` annotation ensures validation is triggered, while `@ValidObjectId` enforces the correct MongoDB `ObjectId` format. This triggers the `ObjectIdValidator` class, which implements the validation logic, ensuring malformed IDs are rejected before further processing.

## 2.3 System Statistics

The application provides three key statistical aggregations to help administrators manage library resources more effectively. These analyses leverage MongoDB's powerful aggregation framework to extract meaningful insights from the dataset.

- Average Age of Active Users by City: This statistic calculates the average age of users who have borrowed books in the last year, grouped by city. It provides demographic insights to help libraries understand their user base and tailor their services accordingly.
- Book Utilisation Analysis: This aggregation identifies books that are underutilised (low number of readings relative to available copies) and overutilised (high number of readings relative to available copies). By highlighting inefficiencies in book allocation, this analysis assists administrators in redistributing copies or acquiring additional ones where needed.
- Most Read Books by Age Group: This statistic determines the most-read books for different user age groups over a given period. It helps libraries curate collections that better align with the reading preferences of different demographics.

The implementation details of these statistical aggregations, including the full MongoDB aggregation pipelines and their corresponding Java implementations, are provided in Section 2.4.1.

While many other useful statistics could be implemented, we have deliberately limited our analysis to these three, each showcasing different capabilities of MongoDB's aggregation framework. For instance, once user reading records are unwound, numerous additional insights could be derived using similar aggregation patterns. However, these would largely involve variations of the same fundamental operations rather than introducing distinct analytical approaches.

## 2.4 Database Queries

This section presents the key database queries implemented in the system, covering both MongoDB aggregation pipelines and operations within Redis. These queries are designed to support core functionalities, such as retrieving analytical insights from stored data and ensuring efficient data access patterns.

### 2.4.1 MongoDB Aggregations

MongoDB aggregation pipelines are extensively used to perform complex data transformations and analytics. Given the system's reliance on structured document storage, these queries enable efficient computation of statistical insights while leveraging MongoDB's built-in optimisations for distributed query execution.

#### 2.4.1.1 Average Age of Active Users by City

This aggregation computes the average age of users who have borrowed books within the last year, grouped by city. Additionally, it counts the total number of such active users per city. This query is useful for understanding the demographics of the library system's user base.

```
db.users.aggregate([
    {
    $match: {
        "readings.returnDate": {
        $gte: new Date(new Date().setFullYear(new
            Date().getFullYear() - 1))
            .toISOString()
            .split('T')[0]
        }
    }
    },
    {
    $group: {
        _id: "$address.city",
        average_age: {
        $avg: {
            $dateDiff: {
            startDate: { $toDate: "$dateOfBirth" },
            endDate: "$$NOW",
            unit: "year"
            }
        }
        },
        total_users: { $sum: 1 }
    }
    }
]);
```

**Code snippet 2.15:** MongoDB Aggregation: Average Age of Active Users by City

This aggregation pipeline processes the `users` collection to compute the average age of users who have borrowed books in the last year, grouped by city. The pipeline consists of two main stages:

- `$match` (Filtering users with recent borrowings): This stage filters documents to include only users who have returned a book within the last year. The field `readings.returnDate` is compared against a dynamically computed date that represents one year before the current date. This ensures that only active users are considered in the calculation.

- `$group` (Grouping by city and calculating statistics): The documents are then grouped by the field `address.city`. Within each group:

- `average_age`: The average age of users in each city is calculated using `$avg`. The age of each user is determined dynamically by computing the difference between their `dateOfBirth` and the current date using `$dateDiff`.
- `total_users`: The total number of users in each city that have borrowed books in the last year is counted using `$sum: 1`.

The final output of this query is a list of cities, each with its corresponding average user age and the total number of active users in the last year. This statistical insight helps administrators understand the demographics of library users and their engagement levels across different locations.

```
"Pisa": {
    "total_users": 697,
    "average_age": 59.76
},
"Livorno": {
    "total_users": 98,
    "average_age": 58.38
},
"Firenze": {
    "total_users": 93,
    "average_age": 56.98
},
"Lucca": {
    "total_users": 100,
    "average_age": 63.24
}
```

**Code snippet 2.16:** Example Output: Average Age of Active Users by City

In Java, the aggregation pipeline is constructed using the `Aggregation` framework provided by Spring Data MongoDB. The following code snippet defines the equivalent aggregation pipeline to compute the average age of active users by city. The query is executed using `mongoTemplate.aggregate`, which processes the pipeline on the `users` collection and retrieves the results.

```
// Compute the date one year ago from today
String oneYearAgo =
    java.time.LocalDate.now().minusDays(365).toString();

Aggregation aggregation = newAggregation(
    // Stage 1: Filter users who have borrowed books in the last
        year
    match(Criteria.where("readings.returnDate").gte(oneYearAgo)),

    // Stage 2: Compute user's age
    addFields().addField("age").withValue(
        new Document("$dateDiff", new Document("startDate", new
            Document("$toDate", "$dateOfBirth"))
            .append("endDate", "$$NOW")
            .append("unit", "year")
        )
    ).build(),

```

```java
    // Stage 3: Group by city and calculate the average age and
        total users
    group("$address.city")
        .avg("$age").as("averageAge")
        .count().as("totalUsers"),

    // Stage 4: Restructure the output
    project("averageAge", "totalUsers")
        .and("_id").as("city") // Rename _id (group key) to "city"
);

// Execute the aggregation on the "users" collection
AggregationResults<Document> results =
    mongoTemplate.aggregate(aggregation, "users", Document.class);
```

**Code snippet 2.17:** Java Aggregation: Average Age of Active Users by City

The aggregation follows a structured sequence of operations:

– `$match` (Filtering users with recent borrowings): This stage selects only users who have returned a book within the last year by filtering the `readings.returnDate` field. The date comparison dynamically calculates the threshold as 365 days before the current date.

– `$addFields` (Computing user age): The `$dateDiff` operator calculates the user's age by finding the difference in years between their `dateOfBirth` and the current date.[8]

– `$group` (Grouping by city and calculating statistics): Users are grouped by `address.city`, which becomes the `_id` field in the grouped documents. The pipeline then:

  ◦ Computes `averageAge` using `$avg` on the dynamically calculated user age.

  ◦ Computes `totalUsers` using `$sum: 1` to count the number of users per city.

– `$project` (Formatting the final output): The `_id` field from the grouping stage is explicitly renamed to `city` for clarity in the final output.

#### 2.4.1.2 Book Utilisation Analysis

This aggregation identifies underutilised and overutilised books based on the ratio of total readings to the number of available copies. It categorises books into two lists:

– Underutilised books: Books with at least 10 copies but a low utilisation ratio (i.e., low number of readings relative to available copies).

[8]Unlike the JavaScript implementation, where the `$dateDiff` calculation is embedded directly within the `$group` stage, Spring Data MongoDB requires an explicit `$addFields` stage beforehand to define the `age` field. This is due to the structured nature of the framework, which enforces a step-by-step approach to aggregation, ensuring clarity and compatibility with its API.

– Overutilised books: Books with at least one copy but a high utilisation ratio (i.e., high number of readings relative to available copies).

This analysis provides valuable insights for administrators responsible for managing the library catalogue. By identifying underutilised and overutilised books, they can make informed decisions on whether to redistribute, remove, or acquire additional copies in specific libraries, ensuring a more efficient allocation of resources.

The following aggregation pipeline computes these statistics.

```
db.books.aggregate([
  {
    $project: {
      _id: 1,
      title: 1,
      total_readings: "$readingsCount",
      total_copies: { $sum: "$branches.numberOfCopies" },
      total_branches: { $size: "$branches" },
      usage_ratio: {
        $cond: {
          if: { $gt: [{ $sum: "$branches.numberOfCopies" }, 0] },
          then: { $divide: ["$readingsCount", { $sum:
             "$branches.numberOfCopies" }] },
          else: 0
        }
      }
    }
  },
  {
    $project: {
      _id: 1,
      title: 1,
      total_readings: 1,
      total_copies: 1,
      total_branches: 1,
      avg_copies_per_branch: {
        $cond: {
          if: { $gt: ["$total_branches", 0] },
          then: { $divide: ["$total_copies", "$total_branches"] },
          else: 0
        }
      },
      usage_ratio: 1
    }
  },
  {
    $facet: {
      "underutilised_books": [
        { $match: { total_copies: { $gte: 10 } } },
        { $sort: { usage_ratio: 1 } },
        { $limit: 10 }
      ],
      "overutilised_books": [
        { $match: { total_copies: { $gte: 1 } } },
        { $sort: { usage_ratio: -1 } },
        { $limit: 10 }
      ]
    }
  }
]);
```

**Code snippet 2.18:** MongoDB Aggregation: Book Utilisation Analysis

The aggregation consists of three main stages:

– `$project` (Computing key metrics): The first stage calculates the total number of copies and branches for each book and derives the `usage_ratio` as the number of readings divided by total copies.

– `$project` (Refining fields): A second projection adds `avg_copies_per_branch` to normalise the number of copies per library branch.

– `$facet` (Categorisation into two lists): This final stage splits books into two lists:

  ◦ `underutilised_books`: Books with at least 10 copies, sorted by increasing usage ratio.

  ◦ `overutilised_books`: Books with at least 1 copy, sorted by decreasing usage ratio.

```
{
    "underutilised_books": [
      {
        "bookId": "679cb334b477993c5cdc8b4f",
        "title": "Alice's Adventures in Wonderland",
        "total_readings": 0,
        "total_copies": 13,
        "total_branches": 4,
        "avg_copies_per_branch": 3.25,
        "usage_ratio": 0
      },
      ...
    ],
    "overutilised_books": [
      {
        "bookId": "679cb339b477993c5cdcbc25",
        "title": "An Outback Marriage: A Story of Australian Life",
        "total_readings": 5,
        "total_copies": 4,
        "total_branches": 4,
        "avg_copies_per_branch": 1,
        "usage_ratio": 1.25
      },
      ...
    ]
}
```

**Code snippet 2.19:** Example Output: Book Utilisation Analysis

In Java, the equivalent aggregation pipeline is constructed using Spring Data MongoDB's `Aggregation` framework. The following code snippet defines the pipeline and executes it via `mongoTemplate.aggregate`.

```java
Aggregation aggregation = newAggregation(
    // Stage 1: Compute book statistics
    project()
        .and("_id").as("bookId")
        .and("title").as("title")
        .and("readingsCount").as("totalReadings")
        .and(AccumulatorOperators
            .Sum.sumOf("branches.numberOfCopies"))
            .as("totalCopies")
        .and(ArrayOperators.Size.lengthOfArray("branches"))
            .as("totalBranches")
        .and(ConditionalOperators.when(
                ComparisonOperators.Gt.valueOf(
                    AccumulatorOperators
                        .Sum.sumOf("branches.numberOfCopies")
                ).greaterThanValue(0))
            .then(new Document("$divide",
               Arrays.asList("$readingsCount",
                new Document("$sum", "$branches.numberOfCopies"))))
            .otherwise(0)
        ).as("usageRatio"),

    // Stage 2: Compute per-branch averages
    project()
        .andInclude("bookId", "title", "totalReadings",
            "totalCopies", "totalBranches", "usageRatio")
        .and(ConditionalOperators.when(
                ComparisonOperators.Gt.valueOf("totalBranches")
                    .greaterThanValue(0))
            .then(new Document("$divide",
               Arrays.asList("$totalCopies", "$totalBranches")))
            .otherwise(0)
        ).as("avgCopiesPerBranch"),

    // Stage 3: Categorise books using facets
    facet()
        .and(
            match(Criteria.where("totalCopies").gte(10)),
            sort(Sort.Direction.ASC, "usageRatio"),
            limit(10)
        ).as("underutilised_books")
        .and(
            match(Criteria.where("totalCopies").gte(1)),
            sort(Sort.Direction.DESC, "usageRatio"),
            limit(10)
        ).as("overutilised_books")
);

// Execute the aggregation on the "books" collection
AggregationResults<Document> results =
    mongoTemplate.aggregate(aggregation, "books", Document.class);
```

**Code snippet 2.20:** Java Aggregation: Book Utilisation Analysis

The aggregation follows the same logical steps as the JavaScript implementation:

- `$project` (Book statistics): Calculates key metrics including total copies, branches, and usage ratio.

- `$project` (Average copies per branch): Computes normalised values for copies per branch.
- `$facet` (Categorisation): Splits books into underutilised and overutilised categories.

#### 2.4.1.3 Most Read Books by Age Group

This aggregation identifies the most-read books for different user age groups over a specified time period. It provides insights into reading preferences across demographics, allowing administrators to tailor library collections and acquisition strategies to better serve different audience segments.

```
db.users.aggregate([
  { $unwind: "$readings" },
  { $addFields: { age: { $dateDiff: {
      startDate: { $toDate: "$dateOfBirth" },
      endDate: "$$NOW",
      unit: "year"
    }}}},
  { $addFields: { age_group: {
      $switch: {
        branches: [
          { case: { $and: [{ $gte: ["$age", 18] }, { $lt: ["$age",
             30] }] }, then: "18-29" },
          { case: { $and: [{ $gte: ["$age", 30] }, { $lt: ["$age",
             50] }] }, then: "30-49" },
          { case: { $gte: ["$age", 50] }, then: "50+" }
        ],
        default: "Unknown"
      }}}},
  { $match: { "readings.returnDate": { $gte: "2024-01-01", $lte:
     "2024-12-30" }}},
  { $group: {
      _id: { age_group: "$age_group", book_id: "$readings.id",
         book_title: "$readings.title" },
      read_count: { $sum: 1 }
    }},
  { $sort: { read_count: -1 }},
  { $group: {
      _id: "$_id.age_group",
      most_read_books: { $push: {
        book_id: "$_id.book_id",
        book_title: "$_id.book_title",
        read_count: "$read_count"
      }},
      total_readings: { $sum: "$read_count" }
    }},
  { $project: { _id: 0, age_group: "$_id", total_readings: 1,
     most_read_books: { $slice: ["$most_read_books", 10] }}},
  { $sort: { age_group: 1 }}
]).pretty();
```

**Code snippet 2.21:** MongoDB Aggregation: Most Read Books by Age Group

The aggregation consists of several stages:

- `$unwind` (Expanding the readings array): Each reading entry is treated as a separate document to allow per-book analysis.
- `$addFields` (Computing user age and age group): The user's age is calculated using `$dateDiff`, and then mapped to a predefined age group.
- `$match` (Filtering relevant readings): Only readings that occurred within the specified date range are retained.
- `$group` (Aggregating read counts per age group and book): Books are grouped by both age group and book ID, summing the number of times each book was read.
- `$sort` (Ordering by popularity): Books within each age group are sorted by total readings.
- `$group` (Regrouping by age group): The most-read books for each age group are stored in an array.[9]
- `$project` (Limiting the results): The output is formatted, keeping only the top 10 most-read books per age group.

```
[
    {
      "ageGroup": "50+",
      "totalReadings": 3967,
      "mostReadBooks": [
        {
          "bookId": "679cb35fb477993c5cddede2",
          "bookTitle": "De Napoléon",
          "readCount": 3
        },
        ...
      ]
    },
    {
      "ageGroup": "30-49",
      "totalReadings": 1519,
      "mostReadBooks": [
        {
          "bookId": "679cb338b477993c5cdcb595",
          "bookTitle": "Riley Farm-Rhymes",
          "readCount": 2
        },
        ...
      ]
    },
    {
      "ageGroup": "18-29",
      "totalReadings": 962,
```

[9]Although books are already grouped by age in the first `$group` stage, MongoDB does not support applying a `$limit` directly to each subgroup after `$sort`. The second `$group` is necessary to collect the books into an array per age group, allowing the use of `$slice` to retain only the top 10 most-read books per group. Without this step, each book would remain as a separate document, preventing per-group filtering.

```
      "mostReadBooks": [
        {
          "bookId": "679cb363b477993c5cde0fb0",
          "bookTitle": "A vision of life",
          "readCount": 2
        },
        ...
      ]
    }
]
```

**Code snippet 2.22:** Example Output: Most Read Books by Age Group

In Java, the aggregation pipeline is built using the `Aggregation` framework in Spring Data MongoDB. To improve readability, only the core steps of the pipeline are shown below, while verbose methods handling field computations are omitted.

```java
Aggregation aggregation = newAggregation(

    // Stage 1: Filter users who have readings in the given period
    Aggregation.match(
        Criteria.where("readings").elemMatch(
            Criteria.where("returnDate").gte(startDate).lte(endDate)
        )
    ),

    // Stage 2: Compute the age group of each user
    calculateAgeGroupField(), // Age computation logic is defined
        separately for readability

    // Stage 3: Project necessary fields before unwinding
    Aggregation.project()
        .andInclude("age_group")
        .andInclude("readings"),

    // Stage 4: Unwind the readings array to analyse individual
        book readings
    Aggregation.unwind("readings"),

    // Stage 5: Retain only readings within the given period
    Aggregation.match(Criteria.where("readings.returnDate").
        gte(startDate).lte(endDate)),

    // Stage 6: Group by age group and book ID, counting the
        number of times each book was read
    group("age_group", "readings.id", "readings.title")
        .count().as("readCount"),

    // Stage 7: Sort books in descending order by read count
    Aggregation.sort(Sort.Direction.DESC, "readCount"),

    // Stage 8: Regroup books by age group and aggregate the top
        books
    groupByAgeWithTopBooks(), // Defined separately for clarity

    // Stage 9: Format the final output
    projectFinalResults()
);
```

```

// Execute the aggregation query on the "users" collection
AggregationResults<BooksByAgeGroupDTO> results =
    mongoTemplate.aggregate(aggregation, "users",
    BooksByAgeGroupDTO.class);
```

**Code snippet 2.23:** Java Aggregation: Most Read Books by Age Group

The Java version follows the same logic as the JavaScript implementation but adheres to the structure enforced by Spring Data MongoDB. Some stages require explicit function calls instead of inline transformations:

- `calculateAgeGroupField()` computes the user's age and assigns them to an age group.
- `groupByAgeWithTopBooks()` performs the secondary grouping to organise the top books per age group.
- `projectFinalResults()` reformats the output to limit results and rename fields.

These steps are defined in separate methods to enhance readability and maintainability. The detailed implementation of these methods is omitted, as they follow the same structure as those shown in the previous statistical queries.

#### 2.4.1.4 Finding a Book with Nearby Libraries

This query retrieves details of a specific book, filtering the embedded `branches` array to include only libraries within a given radius from a specified geographical location. This feature allows users to check availability not just across all libraries but specifically in those nearby, enhancing the efficiency of book searches.
The aggregation pipeline used for this query is as follows:

```
db.books.aggregate([
  { $match: { _id: ObjectId("BOOK_ID_HERE") } },

  { $unwind: { path: "$branches", preserveNullAndEmptyArrays: true
     } },

  { $match: { "branches.location": {
      $geoWithin: {
        $centerSphere: [[LONGITUDE_HERE, LATITUDE_HERE],
            MAX_DISTANCE_HERE / 6378100]
      }
  }}},

  { $group: {
      _id: "$_id",
      title: { $first: "$title" },
      subtitle: { $first: "$subtitle" },
      publicationDate: { $first: "$publicationDate" },
      publisher: { $first: "$publisher" },
      language: { $first: "$language" },
      categories: { $first: "$categories" },
      isbn10: { $first: "$isbn10" },
      isbn13: { $first: "$isbn13" },
      coverImageUrl: { $first: "$coverImageUrl" },
      authors: { $first: "$authors" },
      branches: { $push: "$branches" }
  }}
]);
```

**Code snippet 2.24:** MongoDB Aggregation: Finding a Book with Nearby Libraries

The query consists of the following stages:

- `$match` (Selecting the book): Filters the `books` collection to retrieve only the document corresponding to the specified book ID.
- `$unwind` (Expanding the branches array): Deconstructs the `branches` array so that each library holding the book is processed as an independent document, allowing per-library filtering.
- `$match` (Filtering nearby libraries): Uses the `$geoWithin` operator with `$centerSphere` to retain only libraries within a given maximum distance from the specified coordinates. The maximum distance is expressed in radians by dividing the value in meters by MongoDB's Earth radius approximation (6378100 meters).
- `$group` (Reconstructing the book document): After filtering, the book details are restored, and only the nearby libraries are reinserted into the `branches` array.

The equivalent query is implemented in Java using Spring Data MongoDB. The aggregation pipeline is built dynamically and executed via `mongoTemplate.aggregate`, as shown below:

```
double searchRadius = (maxDistance != null) ? maxDistance : 50000;
Point point = new Point(longitude, latitude);
Circle circle = new Circle(point, new Distance(searchRadius,
    Metrics.METERS));

Aggregation aggregation = Aggregation.newAggregation(
    match(Criteria.where("_id").is(bookId)),

    Aggregation.unwind("branches", true),

    match(Criteria.where("branches.location").withinSphere(circle)),

    Aggregation.group("_id")
        .first("title").as("title")
        .first("subtitle").as("subtitle")
        .first("publicationDate").as("publicationDate")
        .first("publisher").as("publisher")
        .first("language").as("language")
        .first("categories").as("categories")
        .first("isbn10").as("isbn10")
        .first("isbn13").as("isbn13")
        .first("coverImageUrl").as("coverImageUrl")
        .first("authors").as("authors")
        .push("branches").as("branches")
);

// Execute the aggregation query
List<Book> results = mongoTemplate.aggregate(aggregation, "books",
    Book.class).getMappedResults();
```

**Code snippet 2.25:** Java Aggregation: Finding a Book with Nearby Libraries

This implementation follows the same logical steps as the JavaScript version. The key differences in Java are:

- The geographical filtering is handled using Spring Data's `withinSphere` method, which abstracts the `$geoWithin` logic.
- The `$unwind` operation includes the `preserveNullAndEmptyArrays` flag to avoid dropping books with no library branches.
- The book document is reconstructed using `$group`, ensuring that only filtered branches are included in the output.

Unlike statistical aggregations, this query does not generate a new dataset but instead modifies the structure of the book document dynamically based on the user's location. If no nearby libraries match the criteria, an alternative query retrieves the book details without any branches.

## 2.5 Redis Queries

Redis is utilised in this system to efficiently manage frequently accessed data and enforce atomic operations. Given the high concurrency of interactions within the system, ensuring consistency is crucial. The implementation details of Redis queries, including the use of Lua scripting for atomic operations, have been discussed in Section 2.2.4.2[10], while in the current section we only provide a high-level overview of the key operations performed using Redis.

### Book Reservation

When a user reserves a book, the system ensures that all necessary checks are performed, such as verifying book availability and ensuring that the user has not exceeded the maximum allowed reservations. The reservation process updates relevant Redis structures atomically.

### Cancelling a Reservation

If a user cancels a reservation, the system removes it from the appropriate data structures and restores the book's availability, following again an atomic execution model.

[10]In particular, examples of Redis workflows are illustrated in Paragraph *Examples of Critical Redis Workflows.*

#### Marking a Reservation as a Loan

When a user borrows a reserved book, the reservation is transitioned into an active loan. This update ensures that the system correctly tracks loan durations and maintains consistency across Redis structures.

#### Completing a Loan

When a book is returned, the system removes the loan entry, updates book availability, and processes related tracking information. This operation ensures consistency and prevents discrepancies in book availability.

#### Tracking Expirations and Overdue Loans

The system automatically tracks and processes expirations of reservations and loans. When a reservation expires, it is removed, and the book is made available again. Overdue loans are identified and handled accordingly.

## 2.6 REST API Documentation

This section provides a structured overview of the REST API, which serves as the primary interface for client applications to interact with the system. The API follows the OpenAPI standard, ensuring a well-defined and interoperable specification that facilitates integration and maintainability. The documentation is automatically generated from annotations within the Java codebase using `springdoc-openapi-starter-webmvc-ui`, ensuring consistency between implementation and documentation.

The API design prioritises clarity, scalability, and adherence to RESTful principles. Error handling is structured to provide standardised and meaningful responses, allowing clients to effectively interpret and resolve issues such as invalid input, authentication failures, and resource unavailability. All endpoints enforce proper validation and return appropriate HTTP status codes along with descriptive messages.

This section is divided into two parts. The first one provides a high-level overview of the available API endpoints, categorising them based on access control policies, distinguishing between public endpoints, user-restricted endpoints, and administrator-only operations. Each endpoint is presented with a brief description of its functionality, while the full specification, including request parameters, response schemas, and authentication mechanisms, is available in the OpenAPI JSON file.

The second part of this section presents concrete usage examples, demonstrating how to interact with the API in practice. This includes sample requests and responses for selected endpoints, along with visual representations of the API documentation as rendered in Swagger. These examples illustrate how client applications can interact with the system and how responses are structured.

### 2.6.1 API Endpoints

This section provides an overview of the API endpoints available in the system. Each endpoint is presented with a brief description of its functionality. However, detailed information regarding request parameters (path variables, query parameters, and payloads), authentication requirements, and possible responses is precisely documented in the OpenAPI JSON file included in the GitHub repository.

The OpenAPI JSON file can be viewed using tools such as Postman[11] or Swagger[12], allowing for a more interactive and structured visualisation of the APIs.

#### 2.6.1.1 Open Endpoints

These endpoints are publicly accessible and do not require authentication.

**Library-related Searches**

Base endpoint: `/api/v1/libraries/`
Operations related to retrieving library details.

- `GET {id}`

  Get library details.

**Author-related Searches**

Base endpoint: `/api/v1/authors/`
Operations related to retrieving author details and books.

- `GET {authorId}`

  Get author details together with their books (only the first 20).

- `GET {authorId}/books`

  Get paginated books by author in case there are more than 20.

- `GET search`

  Search authors by full name.

**Book-related Searches**

Base endpoint: `/api/v1/books/`
Operations related to book searches and availability.

- `GET`

  Browse the book catalogue retrieving paginated results.

- `GET {id}`

  Get book details with optional library proximity search.

- `GET {bookId}/availability/{libraryId}`

  Check the current availability of a book in a given library.

- `GET search`

  Search books by title and/or author.

[11]`https://www.postman.com/`
[12]`https://editor.swagger.io/`

— `GET filter`

  Browse the book catalogue with optional filters (category and/or popularity, i.e. most read books).

#### 2.6.1.2 Authentication Endpoints

Base endpoint: `/api/v1/auth/`
Operations related to user and admin authentication.

— `POST register`

  Register a new user.

— `POST logout`

  Log out a user.

— `POST login`

  Log in a user.

#### 2.6.1.3 Registered User Endpoints

These endpoints handle operations related to user data and book reservations for registered users.

**User Information**

Base endpoint: `/api/v1/users/`
Operations related to retrieving user data.

— `GET saved`

  Get saved books.

— `GET reserved-loaned`

  Get reserved and loaned books.

— `GET read`

  Get read books.

— `GET details`

  Get user details.

**Reservations**

Base endpoint: `/api/v1/reservations/`
Operations related to book reservations for registered users.

— `POST save`

  Save a book.

- `POST reserve`

  Reserve a book.

- `DELETE {bookId}/{libraryId}`

  Cancel a book reservation.

- `DELETE unsave/{bookId}`

  Unsave a book.

#### 2.6.1.4 Admin-Only Endpoints

These endpoints require administrator-level access.

**Loans**

Base endpoint: `/api/v1/admin/loans/`
Operations related to loan management.

- `POST {libraryId}/{userId}/{bookId}/mark-as-loan`

  Mark reservation as a loan (when a user picks up the reserved book).

- `POST {libraryId}/{userId}/{bookId}/complete`

  Complete a loan (when a user returns the book).

- `GET {libraryId}/reservations`

  Get all current reservations for a library.

- `GET {libraryId}/overdue`

  Retrieve overdue loans in a library.

- `GET {libraryId}/loans`

  Get all current loans for a library.

- `GET {userId}/reserved-loaned`

  Get a user's reserved and loaned books.

- `GET {userId}/details`

  Get user details.

**Usage Statistics**

Base endpoint: `/api/v1/admin/statistics/`
Operations related to statistics and analytics.

- `GET most-read-by-age`

  Get most read books by age group in a given time interval.

- `GET books-utilization`

  Provides book utilisation statistics, categorising books as *underutilised*, i.e. books with a high number of available copies but very few or no readings, and *overutilised*, i.e. books that are frequently read but have limited copies available.

- `GET average-age-readers`

  Get the average age of active readers by geographical area.

**Catalogue Management**

Base endpoint: `/api/v1/admin/catalogue/`
Operations related to managing the book catalogue.

- `POST libraries`

  Add a new library.

- `POST books`

  Add a book to the catalogue.

- `POST books/{bookId}/libraries/{libraryId}`

  Add a library to a book's availability.

- `DELETE books/{bookId}/libraries/{libraryId}`

  Remove a book from a library.

- `PATCH books/{bookId}/libraries/{libraryId}/increment`

  Increment available copies of a book in a library.

- `PATCH books/{bookId}/libraries/{libraryId}/decrement`

  Decrement available copies of a book in a library.

### 2.6.2 API Usage Examples

This subsection provides concrete examples of how to interact with the API, demonstrating request execution and response handling. The examples illustrate typical use cases, showcasing different types of operations such as retrieving data, creating resources, and deleting entries. Each example includes a visual representation of the request and response, highlighting key parameters and expected outcomes.

All successful API responses adhere to a standardised format. Each response contains a `data` field, which varies depending on the specific endpoint, along with a `message` field providing a human-readable confirmation of the operation's success. Additionally, a `timestamp` is included to indicate the exact moment the response was generated, and a `status` field reflects the corresponding HTTP status code.

The following listing presents the general structure of all "`200 OK`" and "`201 Created`" responses returned by the API.

```
{
  "data": {
    // Response payload, varies depending on the endpoint
  },
  "message": "Operation completed successfully",
  "timestamp": "YYYY-MM-DDTHH:MM:SS.ssssss",
  "status": 200 // or 201
}
```

**Code snippet 2.26:** General structure of a successful API response.

Similarly, all error responses follow a structured format that provides clear and meaningful feedback to the client. Each error response includes the request `path`, an `error` field specifying the type of error, a `message` field with a human-readable explanation, a `timestamp` indicating when the error occurred, and a `status` field containing the corresponding HTTP status code.

The following listing presents the general structure of all error responses returned by the API.

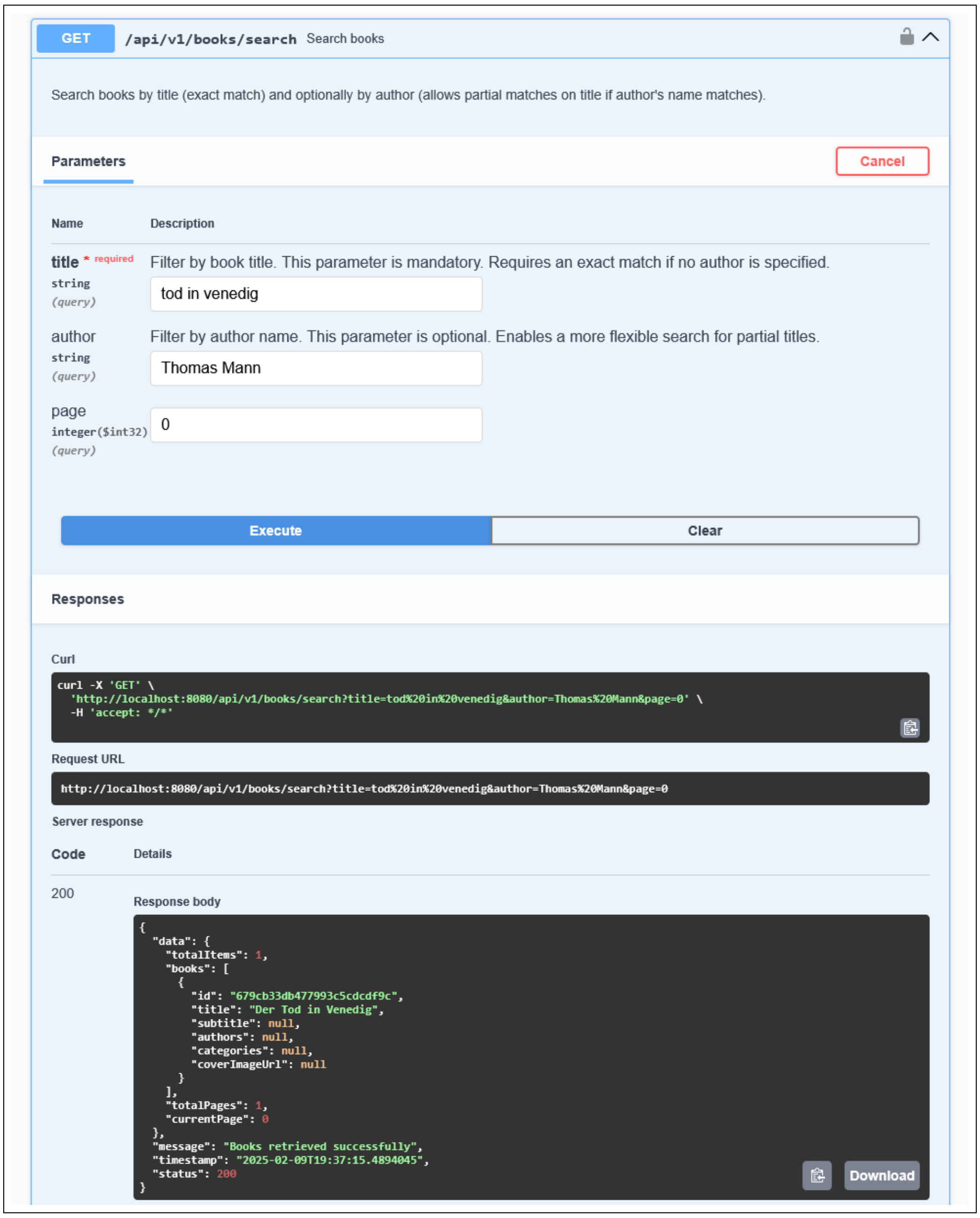
```
{
  "path": "/requested/endpoint",
  "error": "Error type",
  "message": "Detailed error description",
  "timestamp": 1739127427431,
  "status": 400 // or another HTTP error code
}
```

**Code snippet 2.27:** General structure of an API error response.

The following figure depicts a first API interactions within Swagger, offering a structured view of request construction and server responses. These examples serve as a reference for developers, complementing the OpenAPI documentation by demonstrating real-world API usage.

The previous example illustrated a simple `GET` request to retrieve book information. In contrast, the following example demonstrates a `POST` request that modifies the system state by creating a new resource. Specifically, it showcases how a user can reserve a book at a selected library using the `/api/v1/reservations/reserve` endpoint.

This request requires authentication, meaning the user must include a valid Bearer token in the `Authorization` header. The response confirms that the reservation has been successfully registered, returning a `201 Created` status along with relevant details such as the reservation timestamp and expiration time.



Search books by title (exact match) and optionally by author (allows partial matches on title if author's name matches).

| Name | Description |
|---|---|
| title * required string (query) | Filter by book title. This parameter is mandatory. Requires an exact match if no author is specified. tod in venedig |
| author string (query) | Filter by author name. This parameter is optional. Enables a more flexible search for partial titles. Thomas Mann |
| page integer($int32) (query) | 0 |

Curl

```
curl -X 'GET' \
  'http://localhost:8080/api/v1/books/search?title=tod%20in%20venedig&author=Thomas%20Mann&page=0' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/api/v1/books/search?title=tod%20in%20venedig&author=Thomas%20Mann&page=0
```

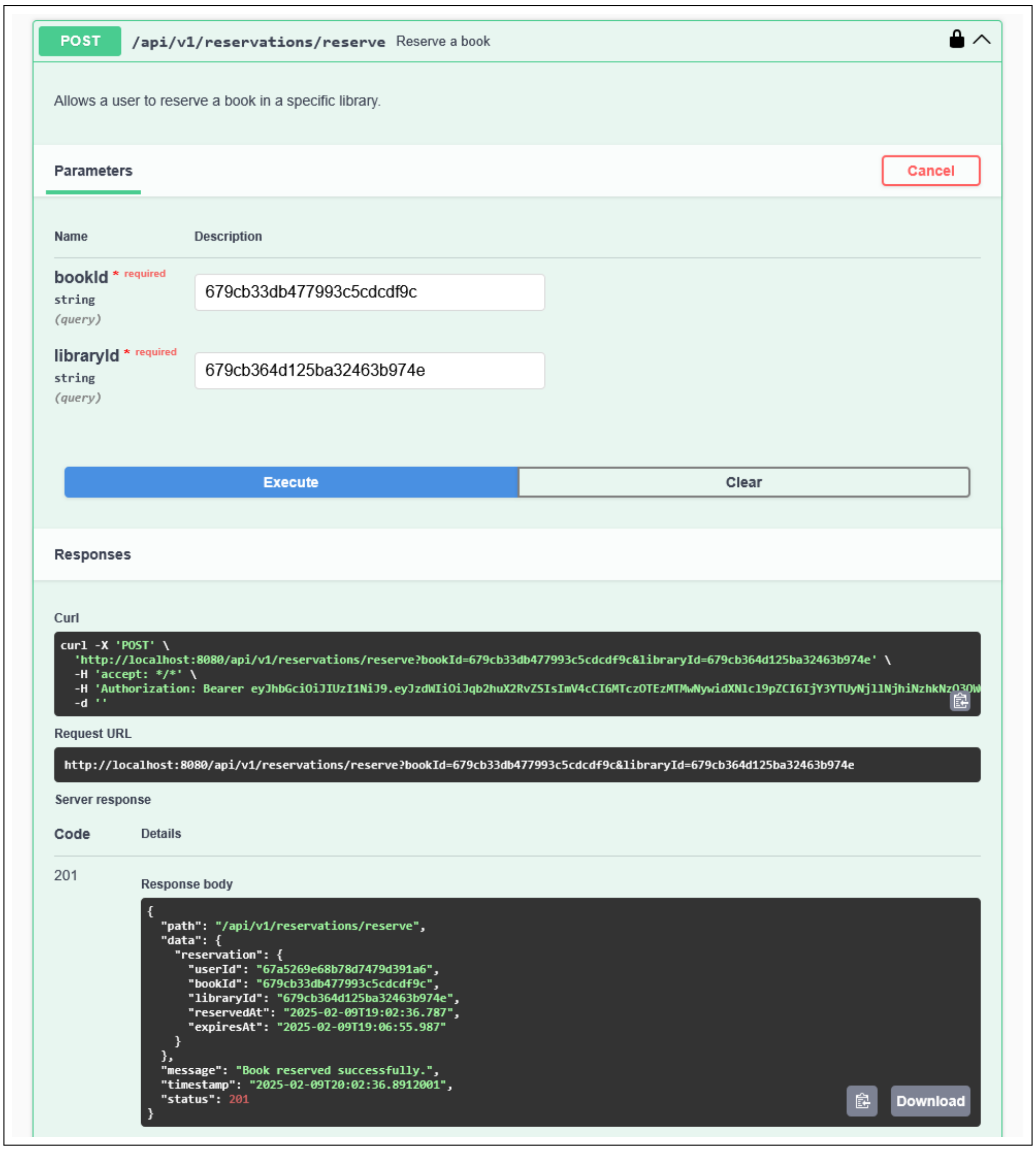
Server response

Code Details

200 Response body

```
{
  "data": {
    "totalItems": 1,
    "books": [
      {
        "id": "679cb33db477993c5cdcdf9c",
        "title": "Der Tod in Venedig",
        "subtitle": null,
        "authors": null,
        "categories": null,
        "coverImageUrl": null
      }
    ],
    "totalPages": 1,
    "currentPage": 0
  },
  "message": "Books retrieved successfully",
  "timestamp": "2025-02-09T19:37:15.4894045",
  "status": 200
}
```

**Figure 2.1:** Execution of a book search request using the `GET /api/v1/books/search` endpoint in Swagger. The request filters books by title and author, displaying the required and optional query parameters, the generated request URL, and the formatted server response.



Allows a user to reserve a book in a specific library.

bookId * required
string
(query)
679cb33db477993c5cdcdf9c

libraryId * required
string
(query)
679cb364d125ba32463b974e

```
curl -X 'POST' \
  'http://localhost:8080/api/v1/reservations/reserve?bookId=679cb33db477993c5cdcdf9c&libraryId=679cb364d125ba32463b974e' \
  -H 'accept: */*' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqb2huX2RvZSIsImV4cCI6MTczOTEzMTMwNywidXNlcl9pZCI6IjY3YTUyNjllNjhiNzhkNzQ3OW
  -d ''
```

```
http://localhost:8080/api/v1/reservations/reserve?bookId=679cb33db477993c5cdcdf9c&libraryId=679cb364d125ba32463b974e
```

201

```
{
  "path": "/api/v1/reservations/reserve",
  "data": {
    "reservation": {
      "userId": "67a5269e68b78d7479d391a6",
      "bookId": "679cb33db477993c5cdcdf9c",
      "libraryId": "679cb364d125ba32463b974e",
      "reservedAt": "2025-02-09T19:02:36.787",
      "expiresAt": "2025-02-09T19:06:55.987"
    }
  },
  "message": "Book reserved successfully.",
  "timestamp": "2025-02-09T20:02:36.8912001",
  "status": 201
}
```

**Figure 2.2:** Execution of a book reservation request using the `POST /api/v1/reservations/reserve` endpoint in Swagger. This request allows a user to reserve a book in a specific library. Notice that this endpoint requires authentication, which is why the request includes a Bearer token in the `Authorization` header, which can be obtained during login. The generated cURL command reflects this requirement. The response confirms the successful reservation with an HTTP `201 Created` status.

# Chapter 3

# Deployment and Testing

This chapter details the deployment strategies and testing methodologies employed for the system. The deployment process involved both a local cluster setup for initial development and testing, as well as a virtualised cluster deployment on the UniPi Virtual Lab infrastructure for a production-like environment. The local deployment was utilised primarily during the development phase to verify application functionality and database interactions within a controlled setting. In contrast, the virtual cluster deployment allowed for the evaluation of the system under realistic conditions, including replica set management, fault tolerance, and distributed operation.

Beyond deployment, this chapter also presents an in-depth analysis of the system's performance through structured testing. Various test scenarios were designed to assess the efficiency of read and write operations, evaluate the system's response under different workloads, and simulate failure scenarios to analyse system resilience. Load testing was conducted to measure system scalability and identify potential bottlenecks, while unit tests ensured the correctness and robustness of individual software components.

## 3.1 Deployment

This section outlines the deployment strategies adopted for the system, covering both local and virtualised environments.

### 3.1.1 Local Cluster Setup

The local deployment configuration was primarily utilised for initial testing and debugging purposes. It provided a simplified environment in which to validate application behaviour before transitioning to a distributed setup. The system was deployed with a single instance of both MongoDB and Redis, running on the local machine.

Configuration files played a crucial role in managing the transition between local and cluster environments. The application employed distinct profiles, `application-local.properties` and `application-cluster.properties`, which determined the system's operational mode. The configuration logic was embedded within the `MongoConfig` and `RedisConfig` classes (cf. Section 2.2.1), where connection settings were dynamically adjusted based on the active profile.

For MongoDB, the local configuration connected to a standalone instance using the default local address, as specified in `application-local.properties`:

```
spring.data.mongodb.uri=mongodb://localhost:27017/library
spring.data.mongodb.transaction.enabled=true
```

**Code snippet 3.1:** MongoDB Local Configuration (`application-local.properties`)

Redis followed a similar configuration, where a standalone instance was used without additional failover mechanisms:

```
spring.data.redis.host=localhost
spring.data.redis.port=6379
spring.data.redis.database=0
```

**Code snippet 3.2:** Redis Local Configuration (`application-local.properties`)

### 3.1.2 Virtual Cluster Deployment

The system was deployed in a virtualised environment within the UniPi Virtual Lab, where it operated across three distinct virtual machines. The deployment process involved configuring MongoDB as a replica set and setting up Redis with Sentinel-based failover mechanisms to ensure high availability and fault tolerance.

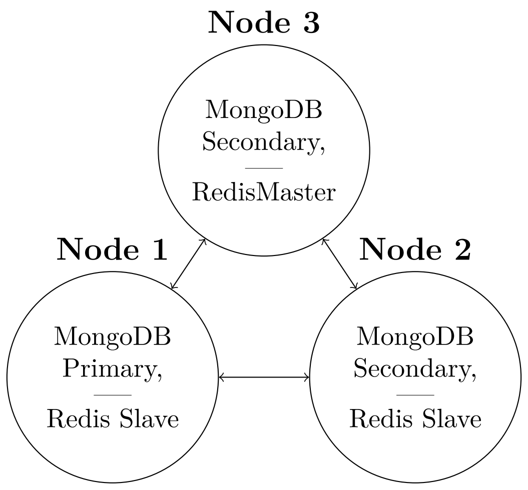


**Figure 3.1:** Representation of the three-node cluster configuration. Redis Sentinel is present on all nodes for monitoring and failover. The Redis Slave in the node with MongoDB Primary has lower priority, making it less likely to be promoted as the new master in case of failure.

#### 3.1.2.1 MongoDB Replica Set

MongoDB was configured as a replica set to enhance fault tolerance and ensure data consistency across nodes. The connection string used in the cluster environment is specified in `application-cluster.properties`:

```
spring.data.mongodb.uri=mongodb://10.1.1.42:27017,
    10.1.1.40:27017,10.1.1.39:27017/library?replicaSet=lsmdb
    &readPreference=secondaryPreferred&retryWrites=true
    &w=majority&wtimeoutMS=10000
spring.data.mongodb.transaction.enabled=true
```

**Code snippet 3.3:** MongoDB Cluster Configuration (`application-cluster.properties`)

This configuration specifies multiple database nodes to support replica sets. The parameter `readPreference=secondaryPreferred` directs read operations towards secondary nodes whenever possible, reducing the load on the primary. This approach is suitable for scenarios where eventual consistency is acceptable, as MongoDB does not enforce strict read-after-write consistency for secondary reads. However, if no secondary node is available, reads will automatically be served by the primary.

The option `retryWrites=true` ensures that failed write operations due to transient network issues or primary step-down events are automatically retried, improving reliability. The parameter `w=majority` guarantees that a write operation is acknowledged only after being successfully replicated to the majority of replica set members, enhancing data durability.

The timeout setting `wtimeoutMS=10000` specifies that if the write concern condition (e.g., acknowledgement by the majority of replica set members) is not met within 10 seconds, MongoDB will return a write concern error. However, any writes that were successfully applied before reaching the timeout will not be undone[1]. This prevents write operations from blocking indefinitely in case the required write concern is unachievable. If no `wtimeout` is set, the operation could theoretically block indefinitely while waiting for acknowledgements.

Once configured, the replica set was initiated from one of the virtual machines, specifying the cluster topology:

```
rs.initiate({
    _id: "lsmdb",
    members: [
        { _id: 0, host: "10.1.1.39", priority: 1 },
        { _id: 1, host: "10.1.1.40", priority: 2 },
        { _id: 2, host: "10.1.1.42", priority: 5 }
    ]
});
```

**Code snippet 3.4:** MongoDB Replica Set Initialisation

#### 3.1.2.2 Redis Replica Set with Sentinel

Redis was configured to operate with a single master and two replicas. The Sentinel mechanism was deployed to handle automatic failover. The application does not connect directly to a Redis instance but instead queries Sentinel to obtain the current master node:

```
spring.data.redis.sentinel.master=mymaster
```

[1]This implies that idempotency checks should be implemented when performing write operations in MongoDB.

```
spring.data.redis.sentinel.nodes=10.1.1.39:26379,
    10.1.1.40:26379,10.1.1.42:26379
spring.data.redis.database=0
```

**Code snippet 3.5:** Redis Sentinel Configuration (`application-cluster.properties`)

This ensures that if the master node becomes unavailable, one of the slave nodes is promoted to master, and the application automatically redirects operations accordingly.

The Sentinel configuration file, deployed on all three virtual machines, includes the following:

```
sentinel monitor mymaster 10.1.1.39 6379 2
```

**Code snippet 3.6:** Redis Sentinel Monitoring Configuration (`sentinel.conf`). Sentinel automatically discovers the master node, eliminating the need for manual configuration updates. The number 2 at the end of the line specifies the quorum required for failover.

This configuration ensures that if the master node becomes unavailable, Sentinel will initiate an automatic failover process, electing a new master from the available replicas based on the predefined quorum.

To reduce the risk of data loss, write operations were configured to propagate to at least one replica:

```
min-replicas-to-write 1
min-replicas-max-lag 10
```

**Code snippet 3.7:** Redis Write Propagation Configuration (`redis.conf`)

Redis operates asynchronously, meaning that this setting does not strictly guarantee that a write is committed on a replica. However, it ensures that the master will wait up to 10 seconds before proceeding, significantly improving the likelihood of data durability under normal conditions.

To prevent failover scenarios where a single machine could become a bottleneck, the priority of Redis replicas was adjusted:

```
replica-priority 50
```

**Code snippet 3.8:** Replica Priority Configuration (`redis.conf`)

By default, Redis assigns a priority value of 100. Lowering it to 50 ensures that the instance running on the same VM as the MongoDB primary is less likely to be promoted as a new master in case of failure.

Redis was configured to use the Append-Only File (AOF) mechanism to enhance durability:

```
appendonly yes
appendfsync always
```

**Code snippet 3.9:** Redis Append-Only File Configuration (redis.conf)

The `appendfsync always` setting ensures that every write operation is immediately recorded in the Append-Only File (AOF) before confirming the write to the client. This configuration prioritises durability, as no data is lost even in the event of a system crash. Since AOF writes are performed in an append-only manner, they remain efficient despite frequent disk I/O, as they avoid costly disk seek operations.[2]

Periodic RDB snapshots provide an additional backup mechanism by capturing full database snapshots at predefined intervals. The configuration ensures that a snapshot is created if at least one change occurs within 900 seconds (15 minutes), acting as a fallback in periods of low activity. During moderate activity, a snapshot is taken if at least 100 changes occur within 300 seconds (5 minutes). In high-activity periods, snapshots are generated more frequently, with a new snapshot created if at least 5000 changes occur within 60 seconds (1 minute).

By leveraging the latest RDB snapshot along with the AOF log, the system can efficiently restore its state to the most recent consistent version, ensuring minimal data loss and quick recovery.

To ensure data consistency during failover, the setting `replica-serve-stale-data no` is used. By default, Redis only allows the master to serve both read and write requests. However, in the event of a master failure, this setting ensures that a slave does not become the new master until it has fully synchronised, including replaying the AOF from the previous master. This guarantees that the newly promoted master serves only up-to-date and consistent data, preventing clients from working with incomplete or stale records during failover.

By implementing these configurations, the deployment achieved a highly available, fault-tolerant architecture capable of handling system failures without disrupting operations.

### 3.1.3 CAP Theorem Considerations

The distributed database design of the system exhibits distinct CAP characteristics across its two main database management systems, MongoDB and Redis, each optimised for different operational requirements.

MongoDB, as a document-oriented database, primarily adheres to an availability-partition tolerance (AP) model. The eventual consistency inherent in its distributed architecture is mitigated by the adoption of a `w=majority` write concern, which ensures that writes are acknowledged by a majority of replica set members before being committed. This choice is aimed at reducing the risk of data loss in the event of a primary node failure, ensuring that committed writes persist across the majority of the replica set. Although temporary inconsistencies may occur across nodes, the replication mechanism ensures that the distributed system converges towards a consistent state without significantly compromising responsiveness. In the context of the data stored in MongoDB, eventual consistency does not introduce issues, as operations such as the addition of new books do not require immediate visibility in user searches, making stale reads perfectly acceptable.

Redis serves as the synchronisation backbone, where strong write consistency is essential to prevent race conditions in reservation and loan transactions. The implementation relies on atomic operations, transactional commands, and expiration mechanisms to enforce strict write synchronisation. This positions the system

[2]Given that the system already replicates data across multiple Redis nodes, a more balanced approach such as `appendfsync everysec` could be considered. This would improve write performance by batching AOF writes every second, reducing the frequency of disk I/O operations.

predominantly in the consistency-partition tolerance (CP) domain, though it also achieves a degree of availability through replication, with slaves synchronising with the latest available changes before becoming available after a master failure. This approach to failover guarantees that any promoted slave maintains transaction integrity, though at the cost of temporary unavailability during synchronisation. The system accepts this trade-off as momentary unavailability is preferable to inconsistent states that could allow multiple users to claim the same book.

This distinction in CAP properties reflects the differing responsibilities of the two databases. MongoDB, while primarily availability-oriented, implements enhanced consistency mechanisms through majority write concerns. Similarly, Redis, though leaning towards CP, maintains a balanced approach through its replication strategy, achieving meaningful availability despite prioritising consistency. Neither system adheres strictly to one side of the CAP triangle, instead adopting intermediate positions that best serve their specific roles: MongoDB facilitating efficient catalogue access with controlled eventual consistency, and Redis ensuring transactional integrity while preserving availability through careful failover management.

## 3.2 Performance and Load Testing

Evaluating the system's performance is essential to ensure its scalability, efficiency, and reliability in a distributed environment. This section analyses the effectiveness of indexing strategies in MongoDB and the system's behaviour under different levels of concurrent load. The results provide insights into query optimisation, response times, and the overall robustness of the application under real-world conditions.

### 3.2.1 Load Testing

To evaluate the system's scalability and robustness, a comprehensive set of stress tests was conducted. The testing strategy was based on the Pareto Principle (80–20 rule), which states that 20% of users generate 80% of the system's load. Additionally, unregistered users were factored into the simulation, as they contribute to the load by browsing the catalogue without requiring authentication.

The distributed library system currently includes approximately 1,000 registered users, with an estimated 500 unregistered users. Given these numbers, 200 registered users were expected to generate 80% of the system's load. The load was categorised into three levels: low stress, with up to 100 registered users and 50 unregistered users issuing concurrent requests; medium stress, with up to 250 registered users and 125 unregistered users; and high stress, with up to 500 registered users and 250 unregistered users. Unregistered users were restricted to a subset of operations, such as browsing the book catalogue, while registered users could perform additional operations, including authentication and book reservations.

These stress levels were defined based on the expected workload distribution outlined in Section 1.1.5 and align with the system's non-functional requirement of supporting at least 50 operations per second, as discussed in Section 1.1.3. This ensures that the performance evaluation is grounded in realistic usage scenarios derived from the system's design considerations.

### Testing Environment and Tools

The testing methodology involved the use of Bash scripts that generated concurrent requests to the Java server through the `curl` command. This approach allowed the simulation of multiple concurrent users and the evaluation of the system's behaviour under increasing loads.

### Login Performance

The first test assessed the system's ability to handle concurrent login requests. Only registered users were considered for this scenario. In each stress level, a batch of concurrent login requests was issued five times, and the average response time was computed.

For low-stress conditions, where 100 concurrent login requests were sent, the average response time was 1.17948 seconds. Under medium-stress conditions, with 250 concurrent login attempts, the response time increased to 2.82811 seconds. In the high-stress scenario, where 500 concurrent login requests were processed, the response time reached 3.14643 seconds. These results indicate a degradation in authentication response times as concurrency increases, highlighting potential areas for optimisation in session handling and authentication workflows. This performance degradation is expected, as the login process involves computationally intensive operations such as credential verification through password hashing and the generation of authentication tokens, both of which add overhead under high concurrency.

### Catalogue Browsing Performance

The second test evaluated the system's ability to process concurrent book search queries. Unlike the login test, both registered and unregistered users participated in this scenario. As with the previous test, each stress level was executed five times, and the average response time was recorded.

With 150 concurrent catalogue search requests, the average response time was 0.770803 seconds. When the number of concurrent search queries increased to 375, the response time was 0.758047 seconds. At the highest level of stress, with 750 concurrent requests, the response time slightly decreased to 0.745574 seconds. These results suggest that the book search functionality remains efficient even under significant concurrent load, likely due to the effectiveness of indexing strategies and query optimisations implemented in MongoDB. Additionally, the presence of three MongoDB instances handling read requests, with read preferences prioritising secondary nodes, further contributes to load distribution and improved query performance under high concurrency.

### Book Reservation Performance

The third test measured the system's ability to handle concurrent book reservation requests, which are operations requiring transactional consistency. The performance was assessed using the same three stress levels.

When 100 concurrent book reservations were requested, the average response time was 0.324825 seconds. Increasing the load to 250 concurrent reservations resulted in an average response time of 0.42052 seconds. Under high-stress conditions, with 500 concurrent reservation requests, the response time further increased to 0.45545 seconds. Although the response time exhibited a slight increase as the load intensified, the system demonstrated stable performance, indicating that the database transactions handling book reservations were efficiently managed.

### Mixed Workload Performance

The final test aimed to simulate a real-world scenario where users performed multiple types of operations in parallel. This test combined login requests, book searches, and book reservations, executed concurrently at varying levels of stress.

At 150 parallel requests, consisting of 50 logins, 50 book searches, and 50 book reservations, the average response time was 0.732445 seconds. Increasing the workload to 375 concurrent requests, comprising 100 logins, 175 book searches, and 100 book reservations, resulted in an average response time of 1.16401 seconds. For 750 parallel requests, which included 200 logins, 300 book searches, and 250 book reservations, the response time increased to 1.42038 seconds. At the highest level of stress, where 1,000 concurrent requests were issued, with 250 logins, 400 book searches, and 350 book reservations, the average response time was 1.44818 seconds.

This test highlighted how the system scales under a diverse workload. While performance degradation was observed with increasing concurrency, response times remained within acceptable limits, demonstrating the system's ability to handle simultaneous user interactions.

### Summary of Findings

The performance evaluation revealed that the system efficiently handles concurrent read and write operations, with indexing strategies significantly improving query response times. Login operations exhibited the most pronounced increase in response times under higher concurrency, suggesting potential areas for optimisation. Catalogue browsing remained stable even under heavy load, indicating that the underlying query optimisations were effective. Book reservation performance showed a controlled increase in response times, highlighting the robustness of transactional handling.

The mixed workload test demonstrated the system's ability to process simultaneous operations efficiently, maintaining response times within an acceptable range. These findings validate the system's scalability and effectiveness in a distributed library environment, providing a strong foundation for future enhancements aimed at further optimising performance under high-concurrency scenarios.

While the system demonstrates efficient scalability in handling concurrent read and write operations, an additional factor influencing performance is the deployment architecture of the application server. Unlike MongoDB and Redis, which are distributed across three nodes, the server is currently running as a single instance without replication or a load-balancing layer. Introducing multiple instances of the application server with a load balancer could further enhance scalability by distributing incoming requests more effectively, reducing bottlenecks, and improving overall system responsiveness under high concurrency.

Furthermore, the results confirm that the system meets and exceeds the defined non-functional requirement of supporting a minimum throughput of 50 operations per second. The observed response times across different stress levels indicate that the system maintains efficient processing even under high concurrency, ensuring reliable performance for a large number of simultaneous users.

### 3.2.2 MongoDB Indices Performance and Effectiveness

Since some operations can be very frequent – such as searching for a specific book, retrieving books in a certain category, listing all works by a given author, or identifying the most popular books (i.e., most read ones) – it was beneficial to define appropriate indices in MongoDB to improve query efficiency. To assess the impact of these indices, queries were first executed without indexing, and their execution statistics were recorded. The same queries were then repeated after creating the corresponding indexes, and the performance improvements were analysed.

#### Indexing Strategy and Performance Evaluation

To compare query execution before and after indexing, MongoDB's built-in query profiler was used, specifically the `.explain("executionStats")` command, which provides insights into execution time, the number of documents examined, and whether an index was utilised. The following tests demonstrate the effect of indexing across various frequent operations.

These experiments were repeated multiple times in a controlled environment using `mongosh` to ensure the validity of the results, particularly for execution times, which can be subject to variations due to system load and background processes.

#### Searching for a Book by Title

Initially, searching for a book by its exact title resulted in a full collection scan (`COLLSCAN`), meaning that MongoDB examined all documents in the collection to find a match. This led to inefficient query execution, with every document being scanned even when only a single result was returned.

After defining an index on the `title` field, the query planner switched to an index scan (`IXSCAN`), significantly reducing the number of examined documents and lowering execution time to nearly zero.

```
db.books.createIndex(
    { title: 1 },
    { collation: { locale: "en", strength: 2, alternate: "shifted" } }
)
```

**Code snippet 3.10:** Index Definition: Title Index (Collation settings enable case-insensitive searches and ignore punctuation/whitespace differences)

With the index in place, searching for a single book title reduced the number of examined documents from over 100,000 (entire collection) to just one, and execution time dropped from around 30ms to near-instantaneous performance.

#### Searching for Books in a Category

Queries retrieving books belonging to a specific category initially required scanning the entire collection, leading to unnecessary overhead as the number of documents increased.

To optimise this operation, an index was created on the `categories` field. Since this field is an array, MongoDB automatically generated a multikey index, allowing efficient filtering based on category membership.

```
db.books.createIndex({ categories: 1 })
```

**Code snippet 3.11:** Index Definition: Category Index

This indexing strategy significantly improved query performance. The number of examined documents dropped from over 100,000 to just the relevant entries, and execution time decreased from 30ms to 2ms.

### Retrieving Books by Author

Since one of the most frequent queries in the system involves retrieving all books written by a specific author, the performance of this operation was also analysed. The `authors` collection contains an array of books associated with each author.

Without indexing, the database performed a full collection scan, examining every document in the `authors` collection. To improve efficiency, an index was created on the `fullName` field.

```
db.authors.createIndex({ fullName: 1 })
```

**Code snippet 3.12:** Index Definition: Author Name Index

Following this optimisation, queries for a specific author's books no longer required scanning the entire collection. The number of examined documents dropped from tens of thousands to just one, and execution time was reduced from 10ms to near-zero.

### Sorting by Read Count (Most Popular Books)

Determining the most-read books required sorting documents based on the `readingsCount` field. Initially, this operation triggered a full collection scan combined with an in-memory sort, leading to inefficiencies as the dataset grew.

To optimise this process, an index was created on `readingsCount` in descending order to support efficient sorting.

```
db.books.createIndex({ readingsCount: -1 })
```

**Code snippet 3.13:** Index Definition: Read Count Index

After implementing this index, MongoDB switched from a collection scan to an index-based sorting strategy. As a result, the number of examined documents decreased from the full collection size to just the required number (e.g. top 30 results), and execution time improved from 28ms to nearly instant sorting.

### Geospatial Indexing for Proximity Search

To support location-based queries that retrieve books available in nearby libraries, we adopted a `2dsphere` index on the latitude and longitude of the embedded library branches. This indexing strategy enables efficient execution of geospatial queries using MongoDB's `$geoWithin` operator.

```
db.books.createIndex({ "branches.location": "2dsphere" })
```

**Code snippet 3.14:** Index Definition: Geospatial Index on Library Branches

This geospatial index was actually defined directly in the application code using Spring Data MongoDB annotations. Specifically, the `@CompoundIndex` annotation was applied to the `branches.location` field within the `Book` model class, ensuring that the index is automatically created when the database schema is initialised. The same approach could also be applied to define other indices declaratively within the model classes, providing a more maintainable and programmatic way of managing them.

### Summary of Findings

The results of these indexing optimisations demonstrate significant performance improvements across all tested queries. Before indexing, MongoDB relied on full collection scans (`COLLSCAN`), leading to high query execution times and unnecessary processing overhead. After implementing indexes, queries leveraged efficient index scans (`IXSCAN`), drastically reducing the number of examined documents and lowering execution times to near-zero.

These findings validate the indexing strategy, ensuring that frequently executed operations, such as book searches, category filtering, author queries, and sorting popular books, are optimised for performance. As a result, the system achieves improved responsiveness and scalability, particularly under high-concurrency scenarios.

## 3.3 Unit Testing

This project implements a comprehensive suite of test classes to validate the behaviour and functionality of the system. The testing framework utilises JUnit 5 and Spring Boot Test, supplemented by Testcontainers to simulate MongoDB and Redis services in a controlled environment.

The test suite covers all controller methods linked to endpoints, excluding those used for statistics extraction, as they are not critical for the core functionality of the system. The implementation addresses the three distinct controller categories previously described in Section 2.6: Open Controllers (accessible by all users), Restricted Controllers (admin-only access), and Secured Controllers (registered users only).

### Testing Technologies

The primary technologies and tools employed in the testing framework include:

– Spring Boot Test: Provides an isolated and consistent test context. Key components include `@SpringBootTest` for application context initialisation and `MockMvc` for HTTP request simulation and response verification.

– Testcontainers: Ensures tests run in a production-like environment using Docker containers for MongoDB and Redis, initialised through `MongoDBContainer` and Redis `GenericContainer` classes.

– ObjectMapper: Implements Jackson library for Java-JSON conversion to facilitate data exchange in REST API testing.

**Security Bypass Mechanisms**

To enable test execution without complex authentication and authorisation management, two distinct approaches were implemented:

– Profile-based Security Deactivation: Most test classes exclude security filters by activating a specific test profile via `System.setProperty("spring.profiles.active", "test")`, allowing direct controller interaction without security restrictions.

– Mocked JWT Tokens: For secured controller tests (`UserControllerTest` and `ReservationControllerTest`), a simulated JWT token is employed using Spring Security Test's `with(jwt())` method to authenticate requests without generating actual tokens, thereby reducing test complexity.

**Test Methodology**

Each test class follows a structured approach:

– Pre-defined test data is created with deterministic IDs to ensure consistent test scenarios;

– JUnit lifecycle annotations (`@BeforeAll`, `@BeforeEach`, `@AfterEach`, `@AfterAll`) manage the test environment;

– Tests verify both HTTP response codes and response content structure using MockMvc and jsonPath assertions;

– Each endpoint is tested with appropriate parameters to validate expected behaviour.

# Conclusions and Future Developments

This project successfully demonstrated the feasibility of a distributed library management system, integrating multiple database technologies to achieve efficient and scalable operations. The implementation leveraged MongoDB for document-oriented storage and Redis for real-time availability tracking, ensuring a balance between consistency and performance. The system's architecture was designed to handle concurrent operations effectively, as confirmed by performance evaluations under varying levels of load.

Future improvements could focus on enhancing scalability by deploying multiple instances of the application server and introducing a load-balancing mechanism to distribute incoming requests more efficiently. Further optimisations could be explored by incorporating a graph database to enable advanced recommendation features, allowing the system to suggest books based on user reading patterns and relationships between authors and works. Enhancing search capabilities by integrating a dedicated search engine such as Elasticsearch would enable more efficient and flexible book retrieval, particularly for title-based queries. Expanding the system's functionality to support additional user interactions and administrative capabilities would further increase its usability and adaptability in real-world library networks.

While the current implementation provides a robust foundation, real-world deployment would offer deeper insights into usage trends and potential refinements. Continuous monitoring and iterative improvements would be essential to maintaining system efficiency and meeting evolving user needs.

# Appendix A

# Application of LLM-based Conversational Agents

This appendix details how LLM-based conversational agents were utilised to support various aspects of the project.

## A.1 Motivation and Methodology

Given the significant capabilities of modern LLM-based conversational agents, integrating them into development workflows represents a valuable opportunity that would be shortsighted to ignore. As these tools are rapidly becoming standard practice in professional environments, their use in academic projects also provides valuable experience with what is emerging as an essential skill in modern software development.

However, these tools were employed as assistive resources rather than automated solutions, with every output being critically evaluated and refined through human expertise. For instance, in documentation, the agents primarily assisted with English language refinement and vocabulary suggestions, as we maintained full control over content and structure.

Similarly, during system design and development phases, the agents served as interactive brainstorming tools, providing insights and alternatives that were then carefully assessed and adapted to meet our specific requirements. This supervised and iterative approach ensured that while we leveraged the capabilities of LLMs, the final decisions and implementations remained firmly under human control.

## A.2 Areas of Application

The following parts of the project benefited from LLM assistance:

- Documentation: LLM-based conversational agents were used to streamline the writing process and enhance the quality of technical English. This assistance proved particularly useful for refining sentence structure, improving clarity, and expanding vocabulary. However, all generated content was meticulously reviewed and edited to ensure coherence, academic tone, and technical accuracy. In many cases, adjustments were necessary to align with the intended style and to ensure precision in terminology and content.

- Code Assistance: Chatbots facilitated the development process by expediting the generation of repetitive code segments and providing quick feedback on implementation details. They were particularly useful for debugging, suggesting alternative approaches, and improving code readability, for example by providing comments.

- Search Engine Augmentation: Chatbots served as a complementary tool alongside traditional online searches. It was particularly effective for quickly summarising best practices, explaining technical concepts, and providing alternative solutions based on established development patterns. Rather than replacing manual research, it enhanced efficiency by offering concise, structured insights that were then verified against official documentation and other authoritative sources.

# Bibliography

[1] Project Gutenberg. Project gutenberg – free ebooks, 2025. Accessed: January 12, 2025.

[2] Yupeng Hou, Jiacheng Li, Zhankui He, An Yan, Xiusi Chen, and Julian McAuley. Bridging language and items for retrieval and recommendation. *arXiv preprint arXiv:2403.03952*, 2024. Dataset available at: `https://amazon-reviews-2023.github.io/main.html`.

[3] Istituto Centrale per il Catalogo Unico delle Biblioteche Italiane. Anagrafe delle biblioteche italiane – open data. Official website: `https://anagrafe.iccu.sbn.it/`, 2025. Dataset page: `https://anagrafe.iccu.sbn.it/it/open-data/`. Accessed: January 12, 2025.