

Project 2 Documentation

Team 20

1 Feature 1: Better user management

In the existing system, there was no way for a user to register by himself. The admin had to add user and also set his/her password. This is neither intuitive nor convenient. Hence we enhanced the the user management system by allowing the user to register himself.

A user can click the *Register* button in the login/starting page and go to the registration page. In the Registration page the user can enter his username, mailId, and password, and register himself.

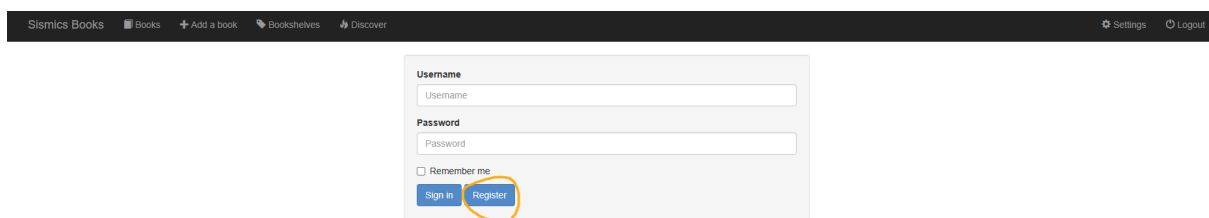
A screenshot of the application's login/starting page. The page has a dark header with navigation links: 'Sismics Books', 'Books', 'Add a book', 'Bookshelves', and 'Discover'. On the right side of the header are 'Settings' and 'Logout' links. The main content area is a light gray box containing a login form. The form has two input fields: 'Username' and 'Password'. Below these fields is a checkbox labeled 'Remember me'. At the bottom of the form are two buttons: 'Sign in' and 'Register'. The 'Register' button is circled in orange.

Figure 1: Register button in the login/starting page

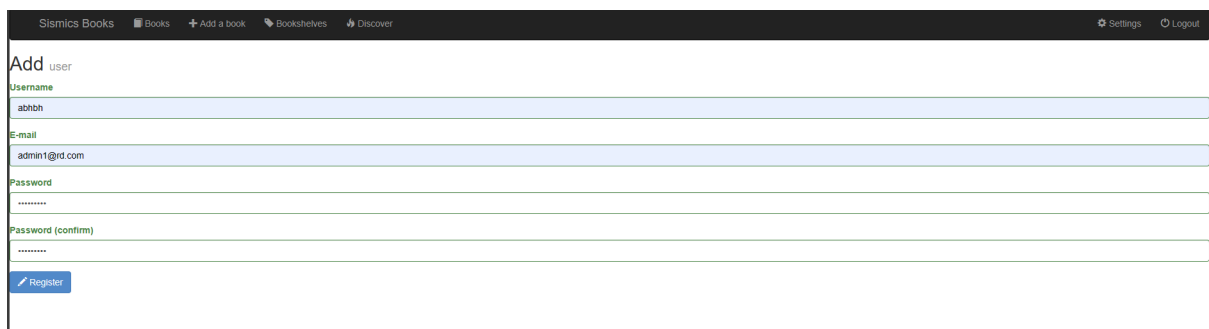
A screenshot of the registration page. The page has a dark header with navigation links: 'Sismics Books', 'Books', 'Add a book', 'Bookshelves', and 'Discover'. On the right side of the header are 'Settings' and 'Logout' links. The main content area is a light gray box with the title 'Add user'. Below the title are four input fields: 'Username' (with the value 'abbbh'), 'E-mail' (with the value 'admin1@rd.com'), 'Password' (with masked characters '*****'), and 'Password (confirm)' (with masked characters '*****'). At the bottom of the form is a blue button with a checkmark icon and the text 'Register'.

Figure 2: Registration page

1.1 Frontend Changes

- Modified login.html to display the *Register button*.
- Updated Login.js controller to support the *Register button* functionality.
- Added registration.html as a new partial for the registration page.
- Introduced Registration.js controller for handling registration logic.

1.2 Backend Changes

Added register_new() function in UserResource.java for this functionality.

2 Feature 2: Common Library

Assumption: The three criteria for filtering $\{authors, genres \text{ and } rating\}$ operate independently of each other. This means that the application of one criterion does not influence or alter the application of the other two.

2.1 Overview

A *library*, which is different from the existing bookshelves is a place for people to discover different tastes. A book can be added by anyone who has the corresponding book in one of their bookshelves. They choose the corresponding genres the book belongs to when adding it to the library. Users can rate books in the common library (once) and can get the top 10 books by average rating or number of ratings. Additionally, they can filter for books based on genre/author and search by title.

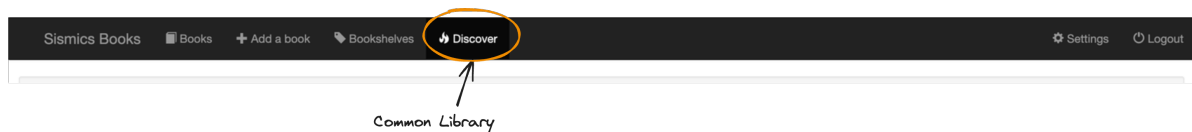


Figure 3: Common Library in NavBar

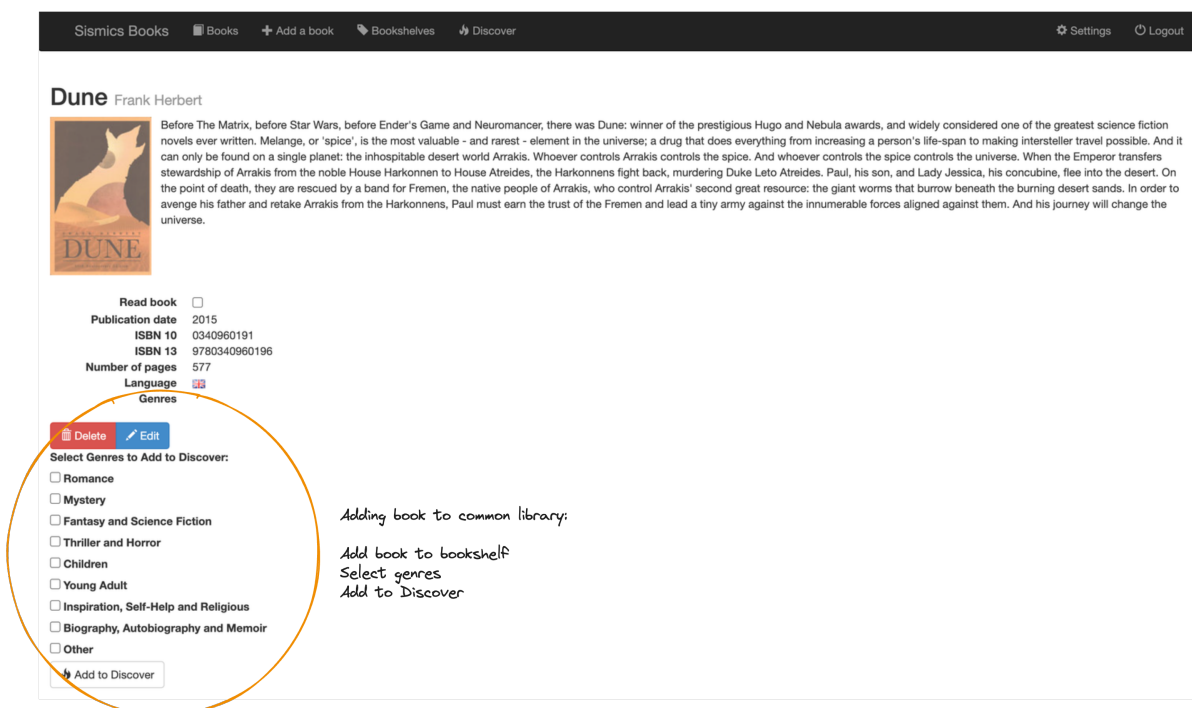


Figure 4: Adding a book to common library

2.2 Database Schema

2.2.1 T_GENRE Table

- **Purpose:** Stores the name of the genre and the corresponding id.
- **Schema:**

```
CREATE MEMORY TABLE T_GENRE (  
  GNR_ID_C VARCHAR(36) NOT NULL,  
  GNR_NAME_C VARCHAR(255) NOT NULL,
```

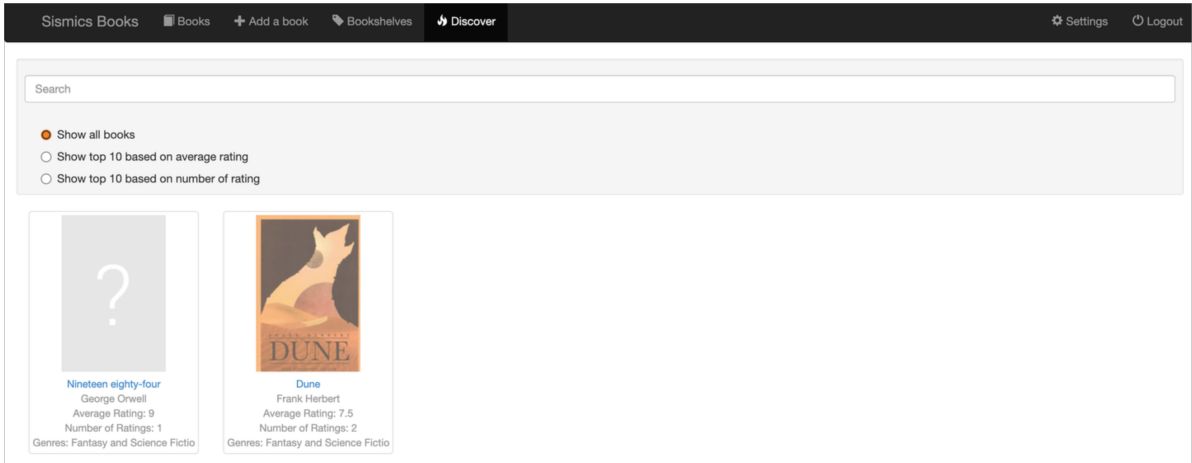


Figure 5: All books in common library

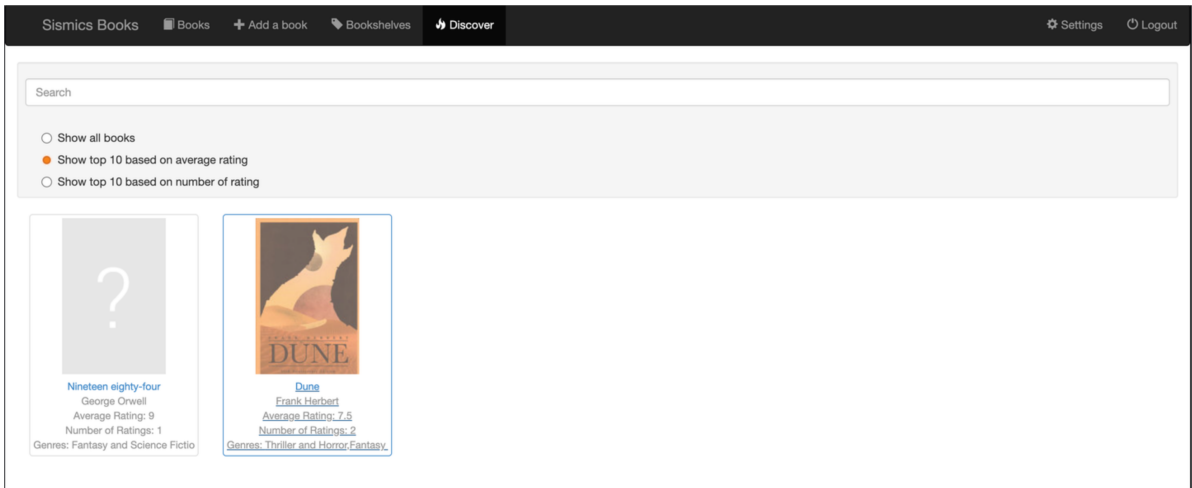


Figure 6: Top 10 books based on average rating

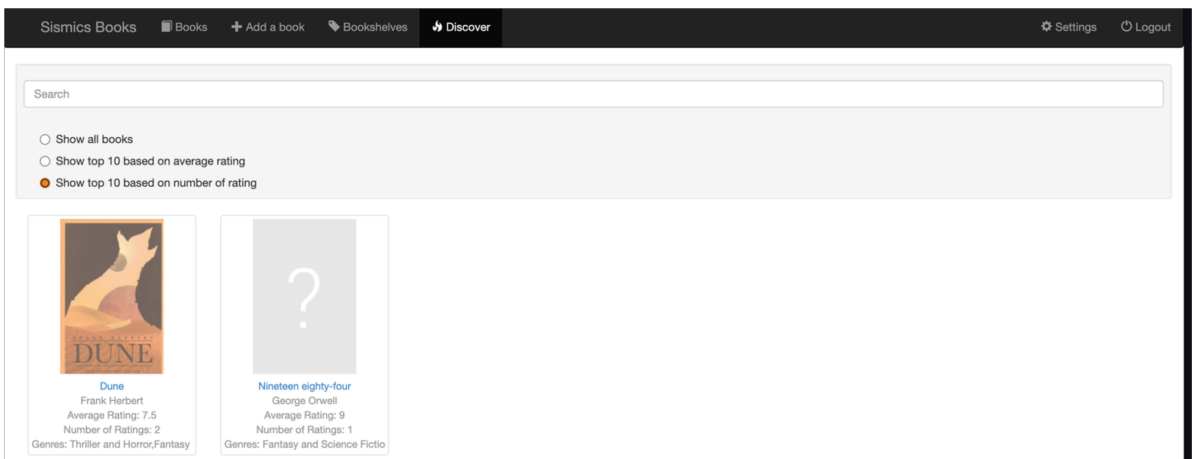


Figure 7: Top 10 books based on number of ratings

```
PRIMARY KEY (GNR_ID_C)
);
```

- **Additional Information:** New genres can be added by modifying the initial state of T_GENRE

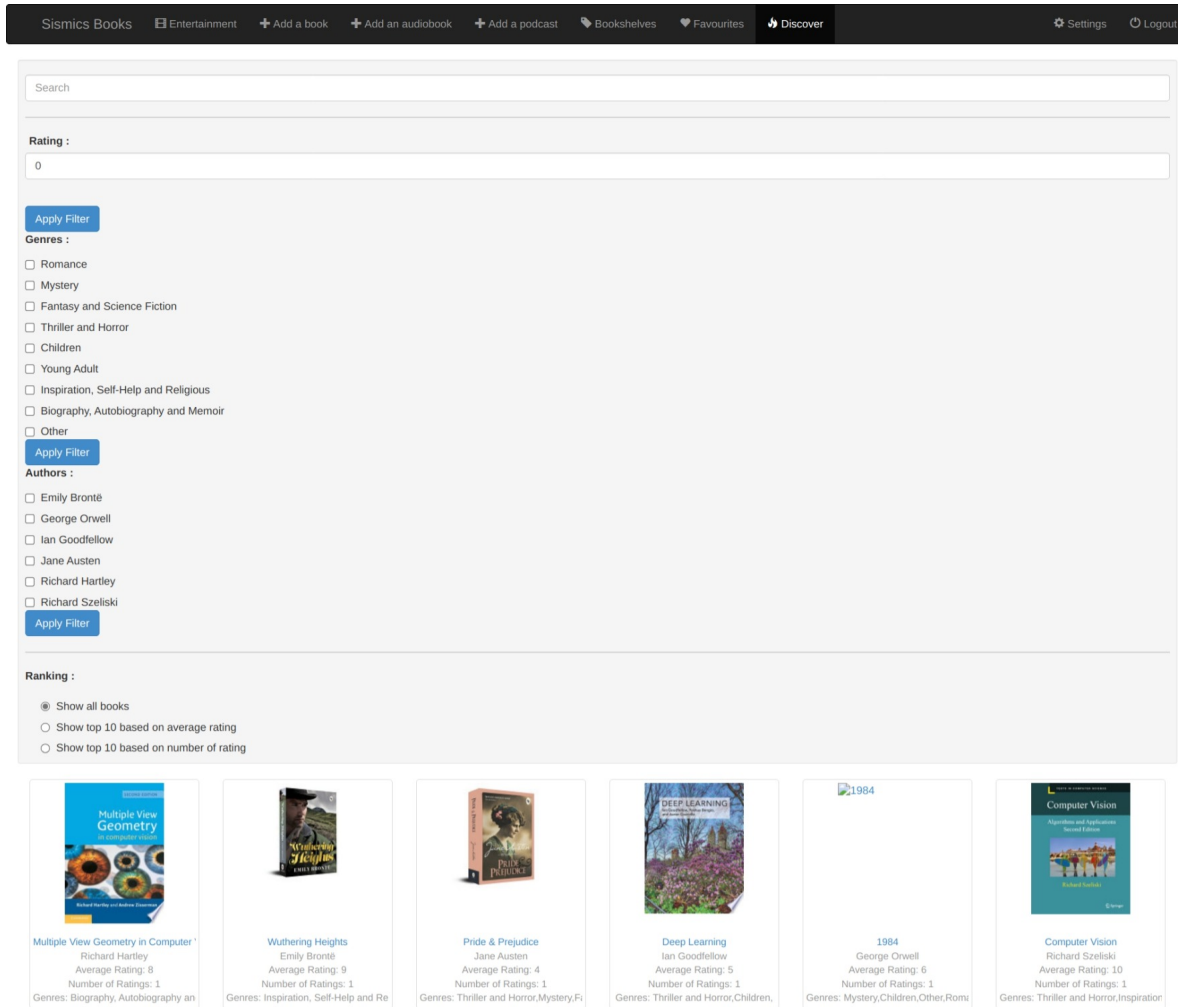


Figure 8: Common library without any active filters

or through `GenreDao`. Default values of genres are inserted into the table at startup.

2.2.2 T_BOOK_GENRE Table

- **Purpose:** Used for the many-to-many mapping between a book's set of genres and the genre table.
- **Schema:**

```
CREATE MEMORY TABLE T_BOOK_GENRE (
  BOK_ID_C VARCHAR(36) NOT NULL,
  GNR_ID_C VARCHAR(36) NOT NULL,
  CONSTRAINT PK_BOOK_GENRE PRIMARY KEY (BOK_ID_C, GNR_ID_C),
  CONSTRAINT FK_BOOK_GENRE_BOOK_ID FOREIGN KEY (BOK_ID_C)
  REFERENCES T_BOOK (BOK_ID_C),
  CONSTRAINT FK_BOOK_GENRE_GENRE_ID FOREIGN KEY (GNR_ID_C)
  REFERENCES T_GENRE (GNR_ID_C)
);
```

2.2.3 T_LIBRARY_BOOK Table

- **Purpose:** Stores the bookid of the books that are in the common library.
- **Schema:**

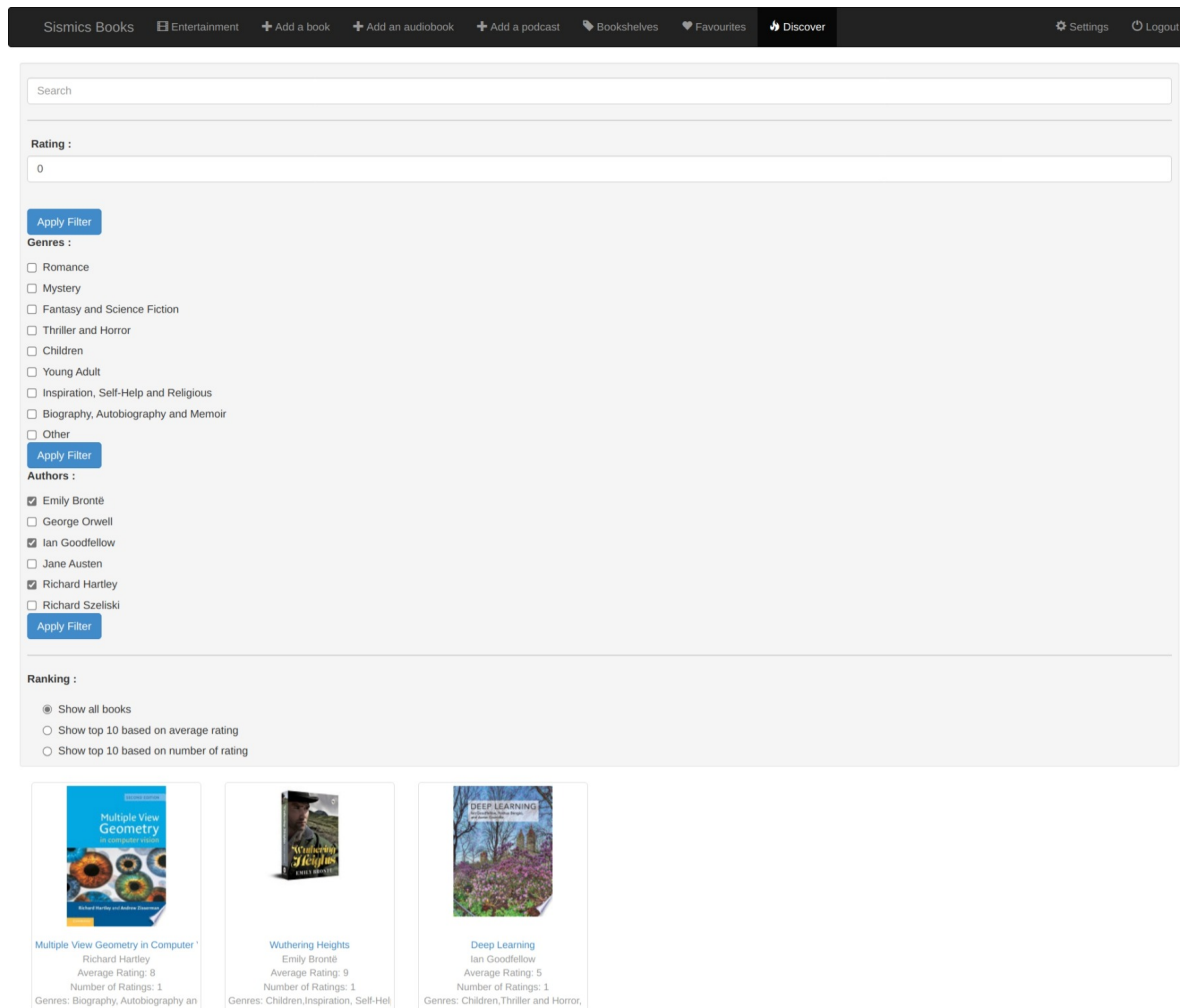


Figure 9: Common library with author filter active

```
CREATE MEMORY TABLE T_LIBRARY_BOOK (
  LBK_ID_C VARCHAR(36) NOT NULL,
  LBK_IDBOOK_C VARCHAR(36) NOT NULL,
  LBK_CREATEDATE_D DATETIME NOT NULL,
  LBK_DELETEDATE_D DATETIME,
  LBK_NUMRATINGS_I INTEGER NOT NULL,
  LBK_AVGRATING_F FLOAT NOT NULL,
  PRIMARY KEY (LBK_ID_C)
);
```

2.2.4 T_LIBRARY_BOOK_RATING Table

- **Purpose:** Stores the ratings for the books in the common library.
- **Schema:**

```
CREATE MEMORY TABLE T_LIBRARY_BOOK_RATING (
  LBR_ID_C VARCHAR(36) NOT NULL,
  LBR_IDBOOK_C VARCHAR(36) NOT NULL,
  LBR_IDUSER_C VARCHAR(36) NOT NULL,
  LBR_RATING_I INTEGER NOT NULL,
  LBR_CREATEDATE_D DATETIME NOT NULL,
  LBR_DELETEDATE_D DATETIME,
  PRIMARY KEY (LBR_ID_C)
```

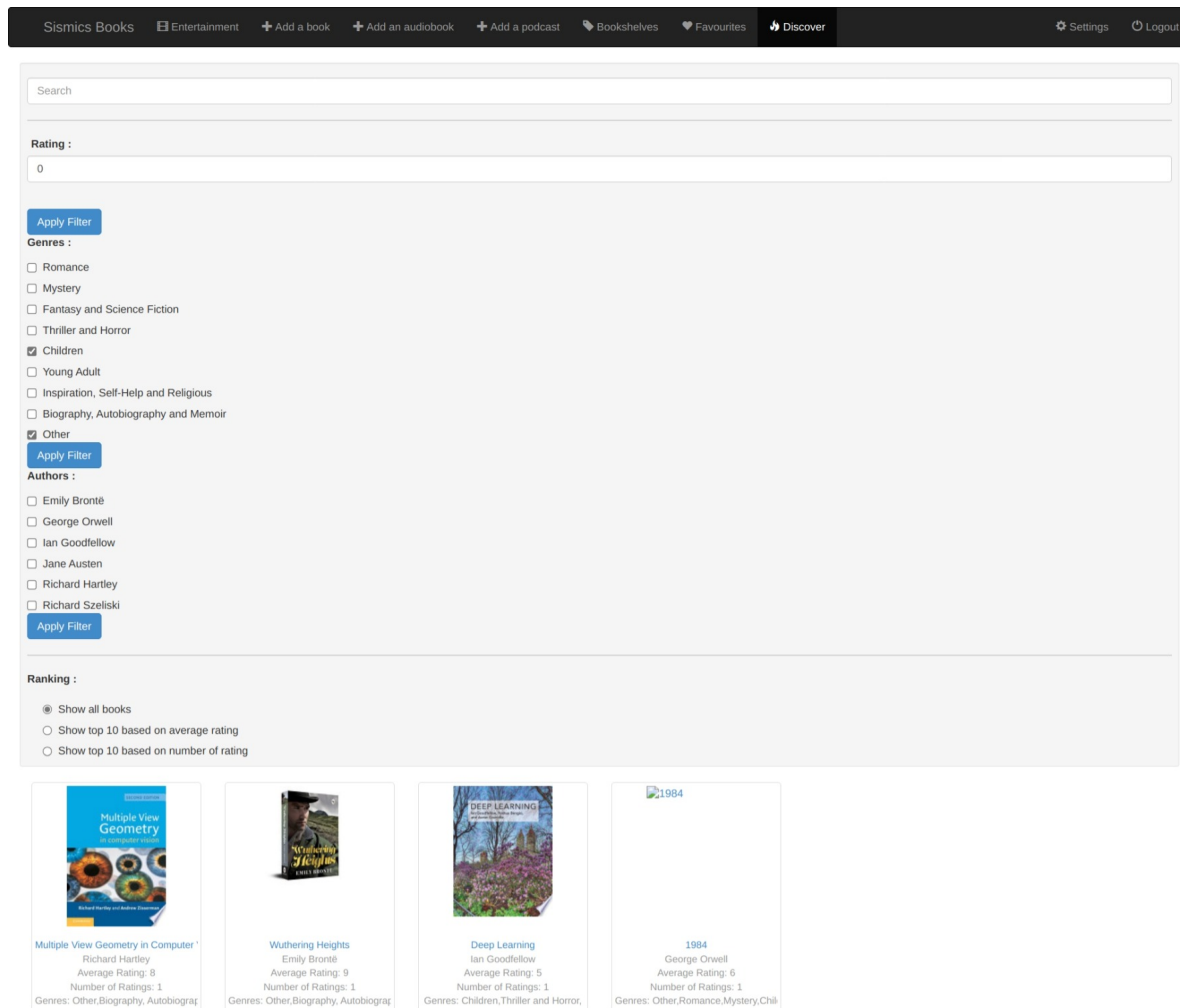


Figure 10: Common library with author genre filters active

);

2.3 Java Classes and Interfaces

2.3.1 GenreDao

- **Purpose:** Manages operations related to genres.
- **Methods:** create, getById, getName.

2.4 LibraryBookDao

- **Purpose:** Manages operations related to library books.
- **Methods:** create, getById, addRatingForBook, getBookId, getAllBooks, getTop10Books, findByCriteria, getQueryByCriteria, assembleResults, getAuthors.

2.4.1 LibraryBookRatingDao

- **Purpose:** Manages operations related to library book ratings.
- **Methods:** create, delete, getById, getUserIdAndBookId.

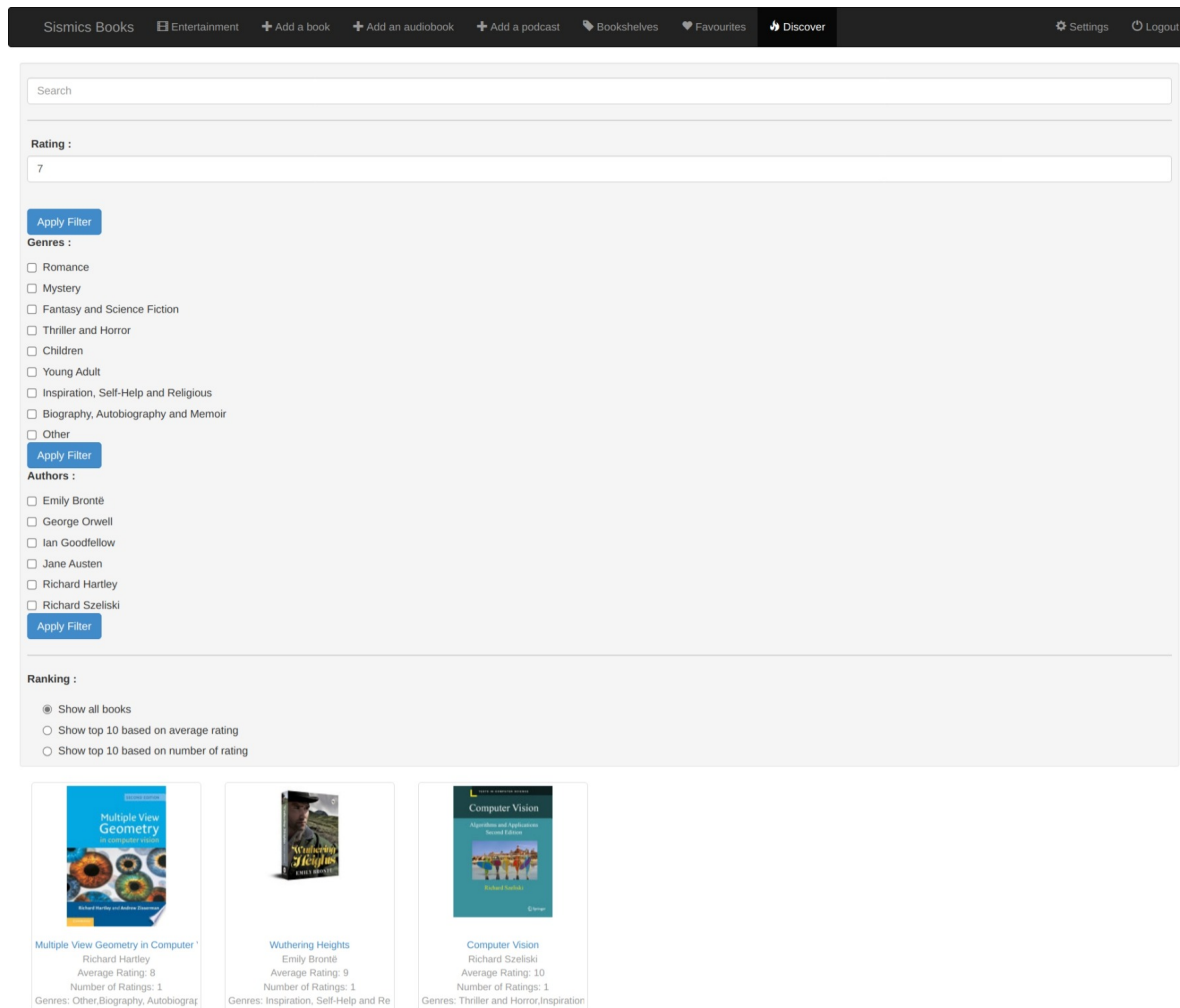


Figure 11: Common library with rating filters active

2.4.2 BaseDao Interface

- **Purpose:** Defines common database operations.
- **Methods:** create, getById.

2.4.3 LibraryBookDto

- **Purpose:** Represents a data transfer object for library books.

2.4.4 LibraryBookResource

- **Purpose:** Handles requests related to library books.
- **Endpoints:** add, list, getLibraryBook, rateBook, getCover, getTop10Books, getBooksBasedOnRating, getBooksBasedOnGenres, getBooksBasedOnAuthors

2.4.5 UserBookDto

- **Purpose:** Represents a data transfer object for user books.
- **Additional Information:** bookId attribute added.

2.5 Frontend Changes

- New controllers `Library` and `LibraryBook` created.
- New partials `library.html` and `library.book.html` added.
- Modified `book.view.html` to display genres.

2.6 Additional Notes

- An interface `BaseDao` with methods `create` and `getById` was introduced to be implemented by various DAOs.
- List of genres was added to `Book`.

2.7 Design Patterns

Strategy

1. **BookCriteriaInterface:** *Strategy* pattern is utilized to define and manage various criteria for searching and filtering books in a library system. An interface named `BookCriteriaInterface` serves as the strategy interface, outlining methods that represent different search and filtering options, such as by search term, read status, user ID, tags, favorites, minimum rating, author names, and genre list.

Concrete strategy classes, namely `LibraryBookCriteria` and `UserBookCriteria`, implement this interface to offer specific behaviors for different filtering scenarios. `LibraryBookCriteria` focuses on library-specific criteria like genre, author, and rating, while `UserBookCriteria` caters to user-specific filters such as read status, favorite status, and tags.

The *Strategy* pattern here allows for the dynamic selection and application of different search and filtering strategies based on the context (e.g., whether the search is more aligned with library-wide criteria or user-specific preferences), promoting flexibility and extensibility in the codebase.

2. **BookCoverable :** *Strategy* pattern is utilized to manage various book cover tasks within a library system. It defines an interface, `BookCoverable`, which outlines methods for *adding books*, *updating covers*, and *getting book information*, including *cover images*. Two primary classes, `LibraryBookResource` and `UserBookResource`, implement this interface, each offering specific ways to handle book covers.

`LibraryBookResource` is focused on managing cover images for the library's book collection. In contrast, `UserBookResource` is tailored towards individual users' needs, allowing them to customize or manage their book covers. This *Strategy* pattern facilitates easy switching between different cover management methods, depending on whether the focus is on the library as a whole or on individual users. This makes the system more flexible and easily adaptable.

3 Feature 3: Online Integration

3.1 Overview

Feature 3 introduces the integration of Spotify and iTunes into the platform, allowing users to access audiobooks and podcasts seamlessly. Users can select their preferred service provider and content type within the platform, enhancing their overall experience and expanding the platform's functionality.

3.2 Detailed Description

1. **Content Addition:** Users have the option to add audiobooks or podcasts to the application. They can select between Spotify and iTunes as service providers for accessing the desired content. Utilizing a simple search bar, users input strings to find specific content within the application.
2. **Search Functionality:** Upon entering search queries, the application retrieves relevant results based on the user's input. Top 10 results matching the search strings with available content is displayed.

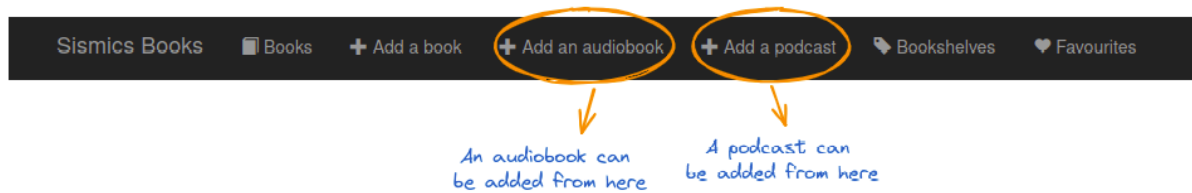


Figure 12: Add Audiobook and Podcast in NavBar

3. **Content Selection:** Users can select their preferred option from the displayed results, prompting the application to add the chosen content to the database for future access.

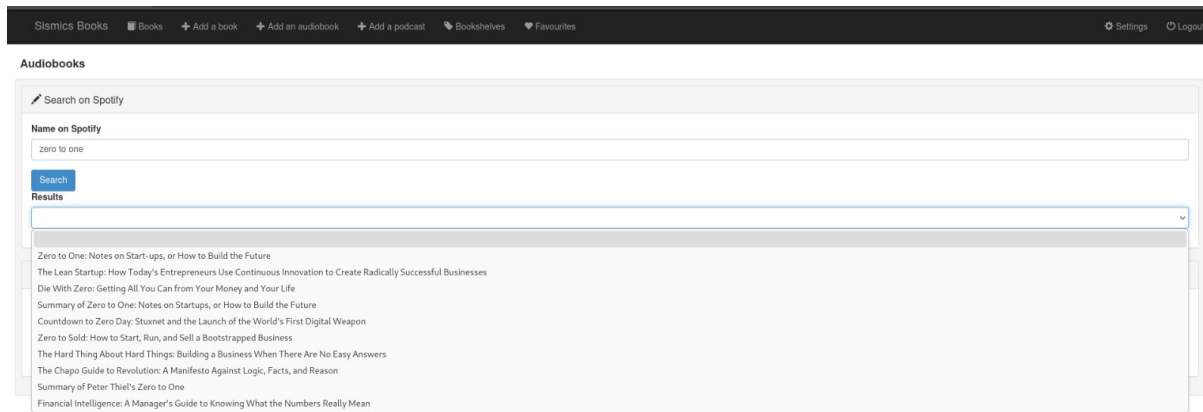


Figure 13: Drop down to choose among the top 10 search results

4. **Content Details:** Upon successful addition, users can view details of each content item by clicking on it. This feature enables users to explore additional information such as title, author, and description.
5. **Favorites Management:** Users can mark any content item, including books, audiobooks, or podcasts, as favorites. Once added to favorites, these items are accessible via a dedicated favorites tab, allowing users to conveniently access and manage their preferred content selections.

3.3 Design Patterns

1. Factory Pattern

We can use the Factory Method pattern to create instances of *Audiobooks* or *Podcasts* based on user selection. This pattern will allow us to encapsulate the object creation logic and provide a way to extend the creation process without modifying the existing code. For example, we can have *SpotifyAudiobookService*, *ItunesAudiobookService* and *SpotifyPodcastService*, *ItunesPodcastService* classes implementing the factory method to create instances of *Audiobook* and *Podcast* respectively.

2. Strategy Pattern

We can use the Strategy pattern to encapsulate the behavior of accessing content from different service providers (Spotify and iTunes). By defining an interface *ServiceStrategy* and implementing concrete strategies like *SpotifyAudiobookService* and *ItunesAudiobookService*, we can easily switch between providers to get the data related to audiobooks without affecting the rest of the codebase.

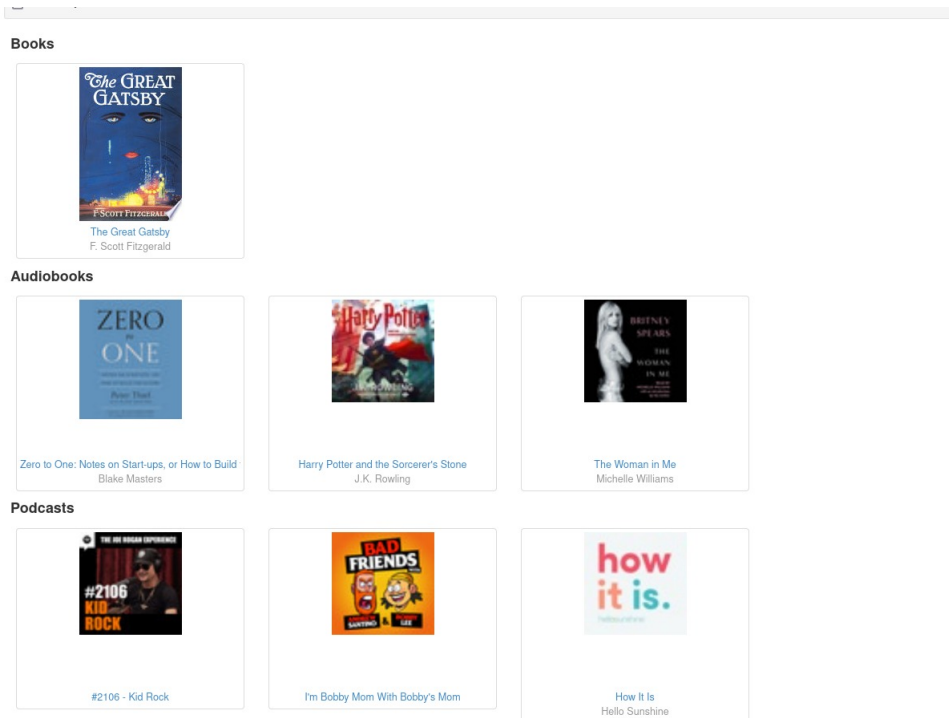


Figure 14: View section once content is added

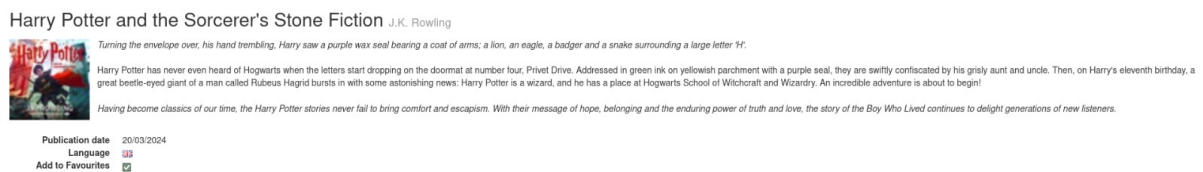


Figure 15: Option to add to favourites

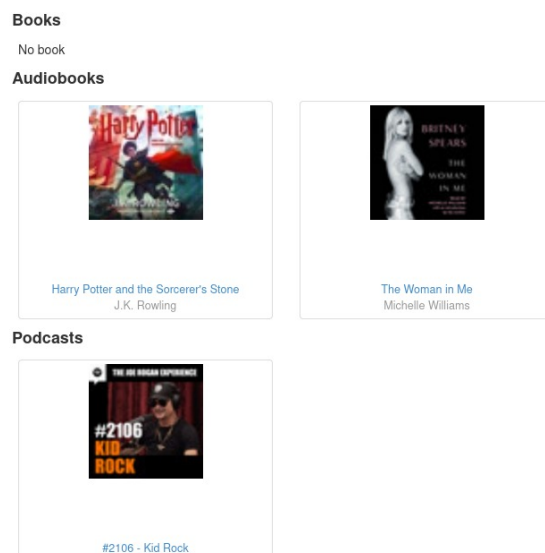


Figure 16: Favourites section

Similarly, for podcasts, concrete strategies like *SpotifyPodcastService* and *ItunesPodcastService* get

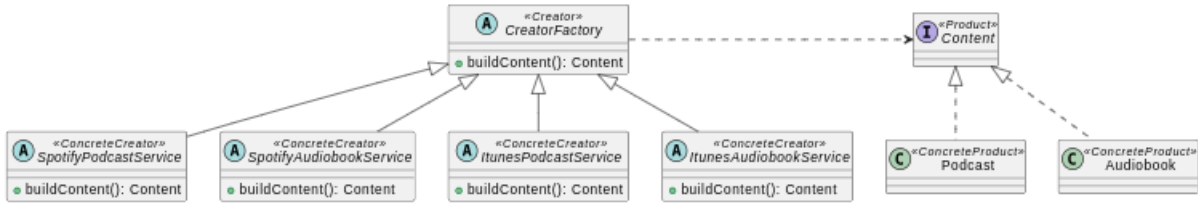


Figure 17: Factory Pattern

data related to podcasts.

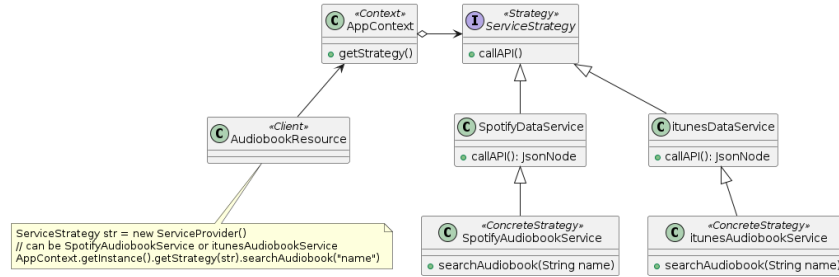


Figure 18: Strategy Pattern for audiobooks

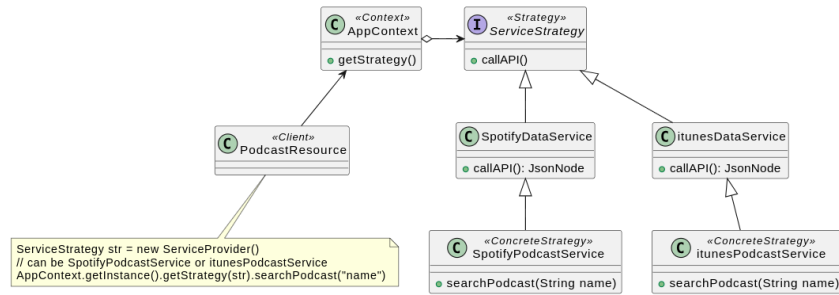


Figure 19: Strategy Pattern for podcasts

3. Adapter Pattern

The Adapter pattern is employed to ensure uniformity in handling various types of content, such as audiobooks and podcasts, within the platform. By adapting audiobooks and podcasts to the same format as books, we maintain consistency in database structure, minimizing changes required, such as introducing a new column called "type" to identify whether an entry represents a book, audiobook, or podcast.

By utilizing this pattern, we can maintain a unified approach across data access and transfer objects, such as *BookDao*, *BookDto*, *UserBookDao*, and *UserBookDto*, without the need for separate implementations. This design choice enhances extensibility, allowing for easier integration of additional content types in the future.

The Adapter pattern facilitates the extension of functions to include audiobooks and podcasts seamlessly, without the need to create separate implementations for each content type. By adapting audiobooks and podcasts to the same format as books, functions that originally handled only books can be extended to handle all content types uniformly. This approach streamlines development and maintenance efforts, as it eliminates the need to duplicate code for handling different content types, leading to a more maintainable and scalable.

4. Singleton Pattern

Singleton is a creational design pattern that lets you ensure that a class has only one instance while providing a global access point to this instance, typically achieved by declaring a static member variable to hold the instance and a private constructor to prevent external instantiation.

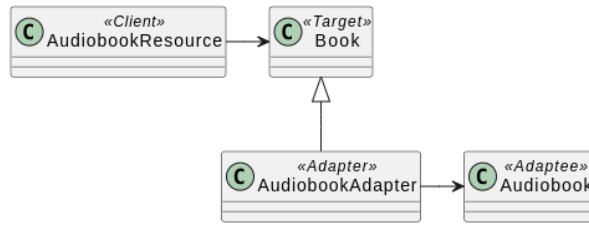


Figure 20: Adapter Pattern for audiobooks

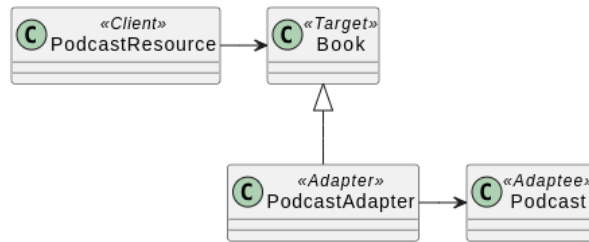


Figure 21: Adapter Pattern for podcasts

The *AppContext* class in our application uses the Singleton pattern. It serves as a global context for accessing various services and resources. By enforcing a single instance of *AppContext*, we ensure that all parts of the application interact with the same context, promoting consistency and centralized management.

The constructor of *AppContext* is private, preventing external classes from creating instances of *AppContext* directly. The class contains a static variable named *instance*, which holds the single instance of *AppContext*.

The *getInstance()* method is used to obtain the singleton instance of *AppContext*. It first checks if the *instance* variable is null, indicating that no instance has been created yet. If so, it creates a new instance of *AppContext*. Subsequent calls to *getInstance()* return the same instance.

In the context of integrating Spotify and iTunes services for accessing audiobooks and podcasts, the Singleton pattern in *AppContext* plays a crucial role. It ensures that there is only one instance of *AppContext* throughout the application, allowing consistent management of service instances. For instance, the *SpotifyAudiobookService*, *SpotifyPodcastService*, *ItunesAudiobookService*, and *ItunesPodcastService* instances are all managed within the same *AppContext* instance. This design choice streamlines the integration process by providing a centralized location to access and configure service instances, enhancing maintainability and extensibility.

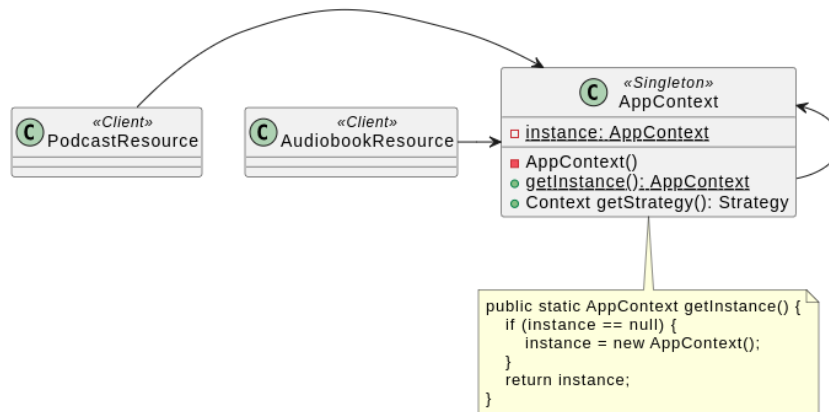


Figure 22: Singleton Pattern

3.4 Challenges

1. Search Format: Designing search functionality for audiobooks and podcasts proved challenging due to differences in their formats. Ensuring compatibility across various search parameters and formats required careful consideration and testing to achieve accurate and relevant results.
2. Access Token Management: Integrating with Spotify posed challenges due to the requirement for access tokens, which expire hourly. Managing token expiration and renewal while maintaining uninterrupted service delivery demanded careful implementation to prevent disruptions in user experience.
3. Adapter Pattern Implementation: Adopting the adapter pattern necessitated accommodating AudiobookAdapter and PodcastAdapter alongside regular books in the same database table. Configuring entity inheritance types and discriminator values became essential to distinguish between different content types accurately.
4. SQL Query Handling: The database structure made it difficult to create SQL queries since it only used discriminator variables to distinguish between different categories of material. Precise query building and optimisation were necessary to provide effective and efficient querying while utilising discriminator values in order to preserve data integrity and system performance.

4 Contribution

- Feature 1 - Rudra Dhar
- Feature 2 - Shrikara A, Rudra Dhar and Prakhar Jain
- Feature 3 - Meghana Tedla and Adyansh Kakran