CS6.401 Software Engineering Spring 2024

Project 2

Team 23

Team Members:

Name	Roll Number
Karan Bhatt	2022202003
Madhusree Bera	2022202007
Vedashree Ranade	2022201073
Yash Maheshwari	2022201074
Piyush Rana	2022202012

Table of Contents

Feature 1: Better user management

Requirements

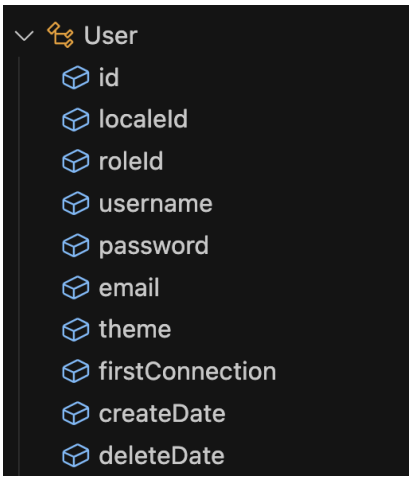
- **R1**: This feature enables a user to register for a user account from the login page itself.
- **R2**: User can enter his details like username, password and email in the registration page
- **R3**: User email should be checked for uniqueness
- **R4**: Password should be entered twice for confirmation
- **R5**: Validation checks should be performed such as valid email, minimum length of password should be 8 digit, etc.

Design Pattern:

● Builder Pattern - Creation of Users

Rationale:

The User object has various parameters as shown below. Creating a new user does not require all the constructor parameters. Hence a builder pattern is suitable to create User objects. The default User parameters required during registration are username, email and password. All these fields are mandatory. Hence, the director will build the User object using these 3 fields from the registration form and all the remaining fields will have default values.



Extensibility

The builder pattern will make the code extensible as new form parameters can be introduced which may or may not be optional. The director will use the appropriate construction when a new user with a separate set of attributes needs to be created.

Modularity

There has been separation of functionalities making the code modular as parameters are being populated in the User object using the Builder class methods and the Director class will construct the object using the object of Builder class. So the object itself is separated from the User class. Interaction with the database is done in UserDao class and API endpoint for registration is present in UserResource class. So the code is following a modular approach.

Maintainability

Using builder design pattern is making the code maintainable as standard classes and methods are used to perform the task of building a User object. For validating unicity of email, database query is written in UserDao class which is also following standard practice. If modifications are needed to User object creation, they are localized within the builder, improving overall code maintainability.

Code Snippets

- Interface created for builder class with all the methods to build the user parameters

UserBuilder.java

```
package com.sismics.books.rest.builder;
import java.util.Date;
public interface UserBuilder {
   public abstract void buildId(String id);
   public abstract void buildLocaleId(String localeId);
   public abstract void buildRoleId(String roleId);
   public abstract void buildUsername(String username);
   public abstract void buildPassword(String password);
   public abstract void buildEmail(String email);
   public abstract void buildTheme(String theme);
```

```
    public abstract void buildFirstConnection(boolean firstConnection);
    public abstract void buildCreateDate(Date createDate);
    public abstract void buildDeleteDate(Date deleteDate);
}
```

- ConcreteUserBuilder class created that implements the UserBuilder interface

ConcreteUserBuilder.java

```
package com.sismics.books.rest.builder;

import java.util.Date;

import com.sismics.books.core.model.jpa.User;

public class ConcreteUserBuilder implements UserBuilder{
    private User user;
    public ConcreteUserBuilder() {
        user = new User();
    }
    @Override
    public void buildId(String id) {
        user.setId(id);
    }

    @Override
    public void buildLocaleId(String localeId) {
        user.setLocaleId(localeId);
    }

    @Override
    public void buildRoleId(String roleId) {
        user.setRoleId(roleId);
    }

    @Override
    public void buildUsername(String username) {
        user.setUsername(username);
    }

    @Override
    public void buildPassword(String password) {
        user.setPassword(password);
    }
```

```java
    @Override
    public void buildEmail(String email) {
        user.setEmail(email);
    }

    @Override
    public void buildTheme(String theme) {
        user.setTheme(theme);
    }

    @Override
    public void buildFirstConnection(boolean firstConnection) {
        user.setFirstConnection(firstConnection);
    }

    @Override
    public void buildCreateDate(Date createDate) {
        user.setCreateDate(createDate);
    }

    @Override
    public void buildDeleteDate(Date deleteDate) {
        user.setDeleteDate(deleteDate);
    }
    public User getResult() {
        return user;
    }

}
```

- Created the director class which will use the builder object and construct a new user according to the parameters received from user input.

UserDirector.java

```java
package com.sismics.books.rest.builder;

import java.util.Date;
import java.util.UUID;

import org.mindrot.jbcrypt.BCrypt;
```

```java
import com.sismics.books.core.constant.Constants;

public class UserDirector {
   private UserBuilder userBuilder;

   public UserDirector(UserBuilder userBuilder) {
       this.userBuilder = userBuilder;
   }

   public void construct(String username, String password, String email){
       userBuilder.buildUsername(username);
       userBuilder.buildPassword(BCrypt.hashpw(password, BCrypt.gensalt()));
       userBuilder.buildEmail(email);
       userBuilder.buildLocaleId(Constants.DEFAULT_LOCALE_ID);
       userBuilder.buildRoleId(Constants.DEFAULT_USER_ROLE);
       userBuilder.buildCreateDate(new Date());
       userBuilder.buildId(UUID.randomUUID().toString());
       userBuilder.buildTheme(Constants.DEFAULT_THEME_ID);
   }
}
```

- Created a method in UserDao class that interacts with the database to check whether the username and email are not already registered previously and finally saves the User object in the database.

UserDao.java

```java
public String registerNewUser(User user) throws Exception {
            System.out.println("user: " + user.getUsername() + " email: " +
user.getEmail() + " password: " + user.getPassword() + " theme: " +
user.getTheme() + " locale: " + user.getLocaleId() + " firstConnection: " +
user.isFirstConnection() + " id: " + user.getId() + " createDate: " +
user.getCreateDate() + " deleteDate: " + user.getDeleteDate());

       // Checks for user unicity
       EntityManager em = ThreadLocalContext.get().getEntityManager();
           Query q = em.createQuery("select u from User u where u.username =
:username and u.deleteDate is null");
       q.setParameter("username", user.getUsername());
       List<?> l = q.getResultList();
       if (l.size() > 0) {
           throw new Exception("AlreadyExistingUsername");
       }
```

```
        Query q2 = em.createQuery("select u from User u where u.email = :email
and u.deleteDate is null");
        System.out.println("email: " + user.getEmail());
        q2.setParameter("email", user.getEmail());
        List<?> l2 = q2.getResultList();
        if (l2.size() > 0) {
            throw new Exception("AlreadyExistingEmail");
        }
        em.persist(user);
        return user.getId();
    }
```

- Created a new API endpoint for “/user/register” path in UserResource class that takes the form parameters from frontend, performs validation check, uses the Builder to build the user object and save it in the database.

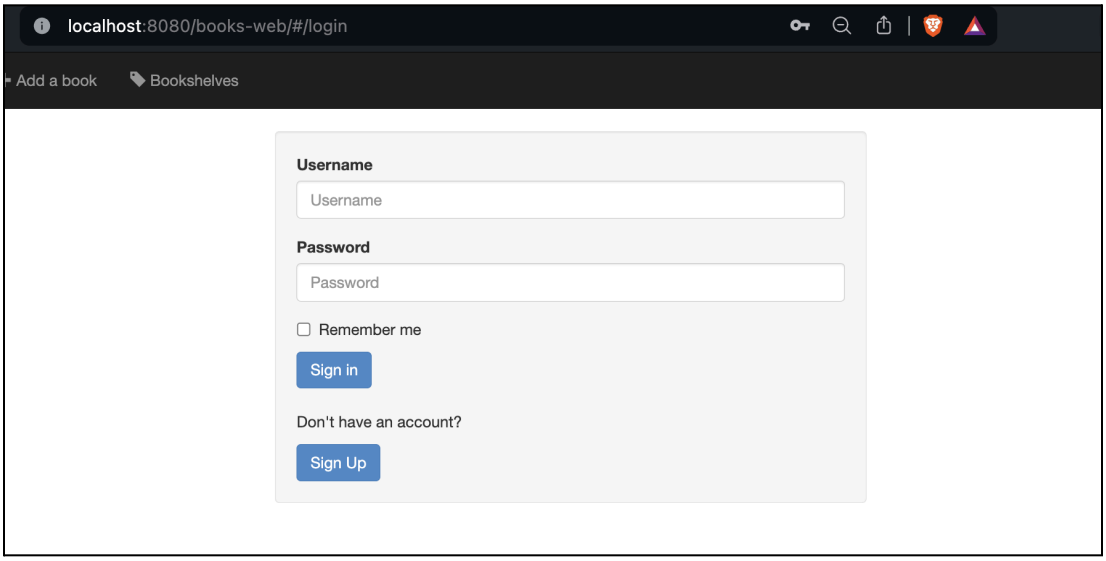
UserResource.java

```
@PUT
    @Path("register")
    @Produces(MediaType.APPLICATION_JSON)
    public Response register(
            @FormParam("username") String username,
            @FormParam("password") String password,
            @FormParam("passwordconfirm") String rePassword,
            @FormParam("email") String email) throws JSONException {
        System.out.println("register");

        // Validate the input data
        username = ValidationUtil.validateLength(username, "username", 3, 50);
        ValidationUtil.validateAlphanumeric(username, "username");
        password = ValidationUtil.validateLength(password, "password", 8, 50);
        email = ValidationUtil.validateLength(email, "email", 3, 50);
        ValidationUtil.validateEmail(email, "email");

        //Create the user using the default UserBuilder
        UserBuilder userBuilder = new ConcreteUserBuilder();
        UserDirector userDirector = new UserDirector(userBuilder);
        userDirector.construct(username, password, email);

        //Save the user
```

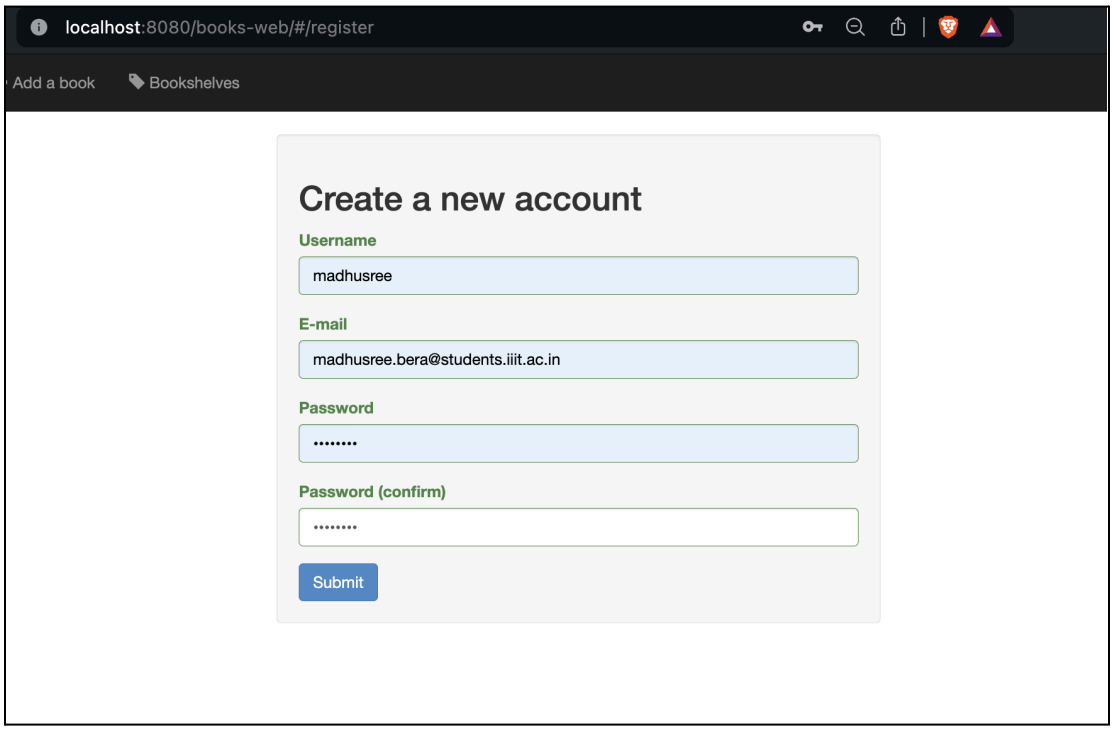
```
        UserDao userDao = new UserDao();
        try {

userDao.registerNewUser(((ConcreteUserBuilder)userBuilder).getResult());
        } catch (Exception e) {
            if ("AlreadyExistingUsername".equals(e.getMessage())) {
                    throw new ServerException("AlreadyExistingUsername", "Username
already used", e);
            } else if ("AlreadyExistingEmail".equals(e.getMessage())) {
                        throw new ServerException("AlreadyExistingEmail", "E-Mail
already used", e);
            } else {
                        throw new ServerException("UnknownError", "Unknown Server
Error", e);
            }
        }
        JSONObject response = new JSONObject();
        response.put("status", "ok");
        return Response.ok().entity(response).build();
    }
```

User Interface

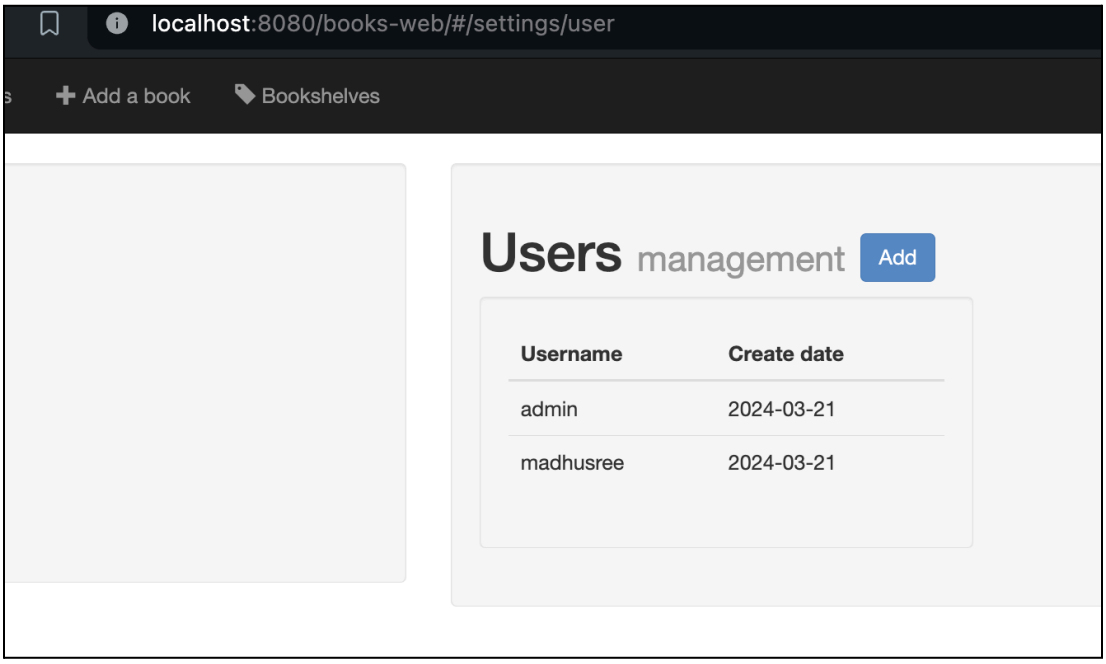
Sign Up button in Login page



User Registration Page

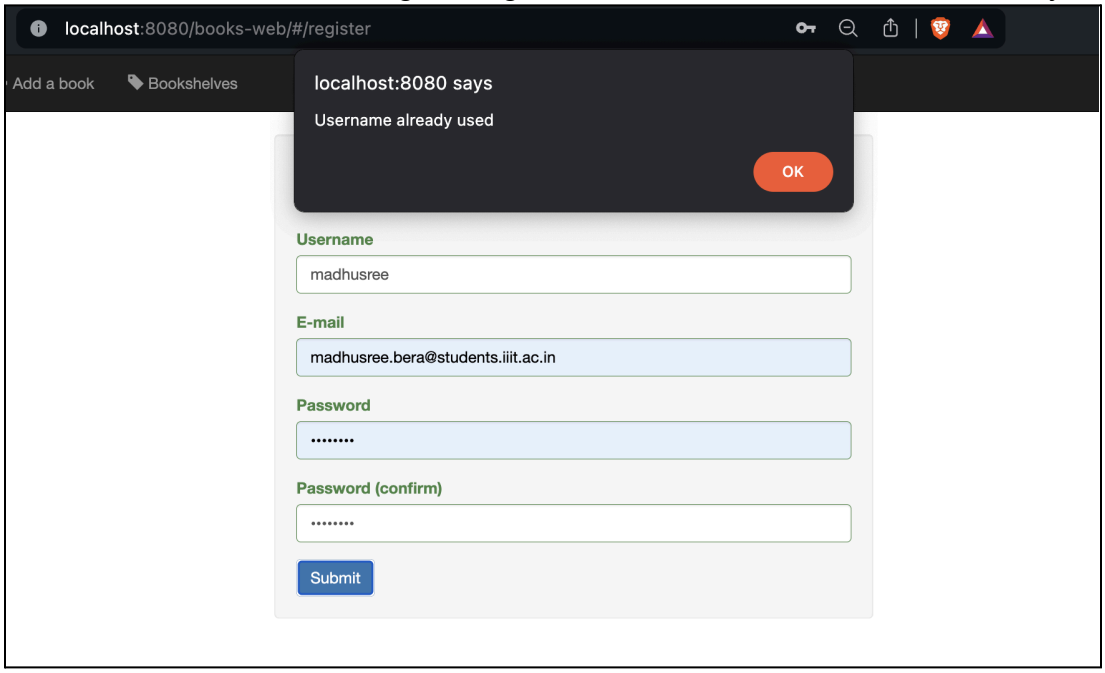


As we can see, a new user has been created and the admin can find it in the list of users.



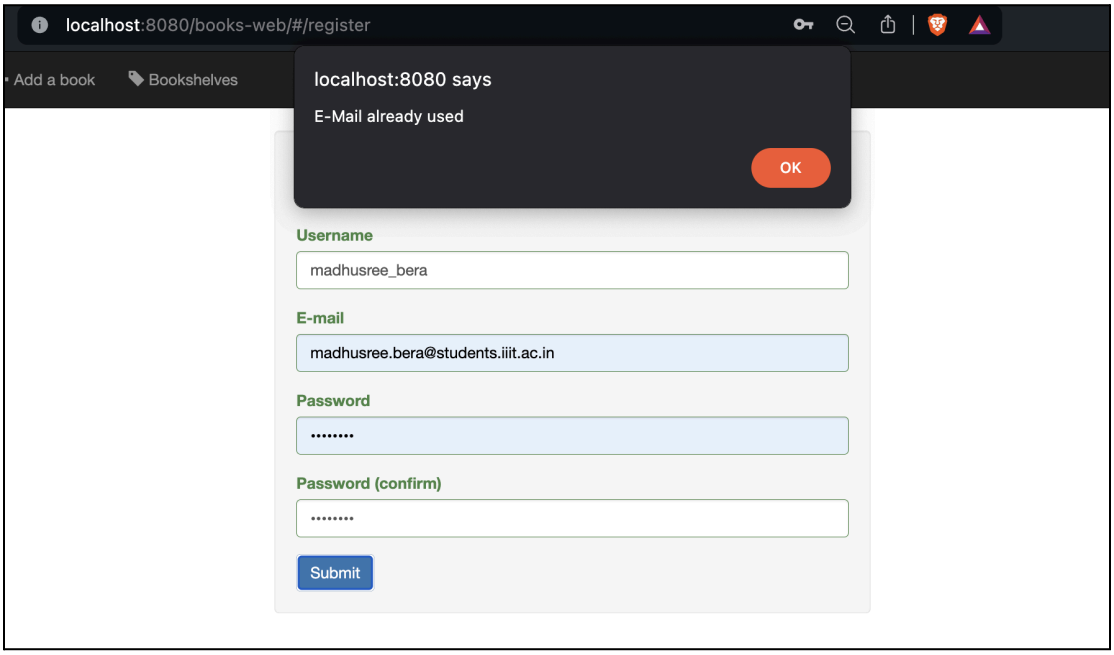
Username Already Used error

If same username tries to register again, it will alert that “Username already used”



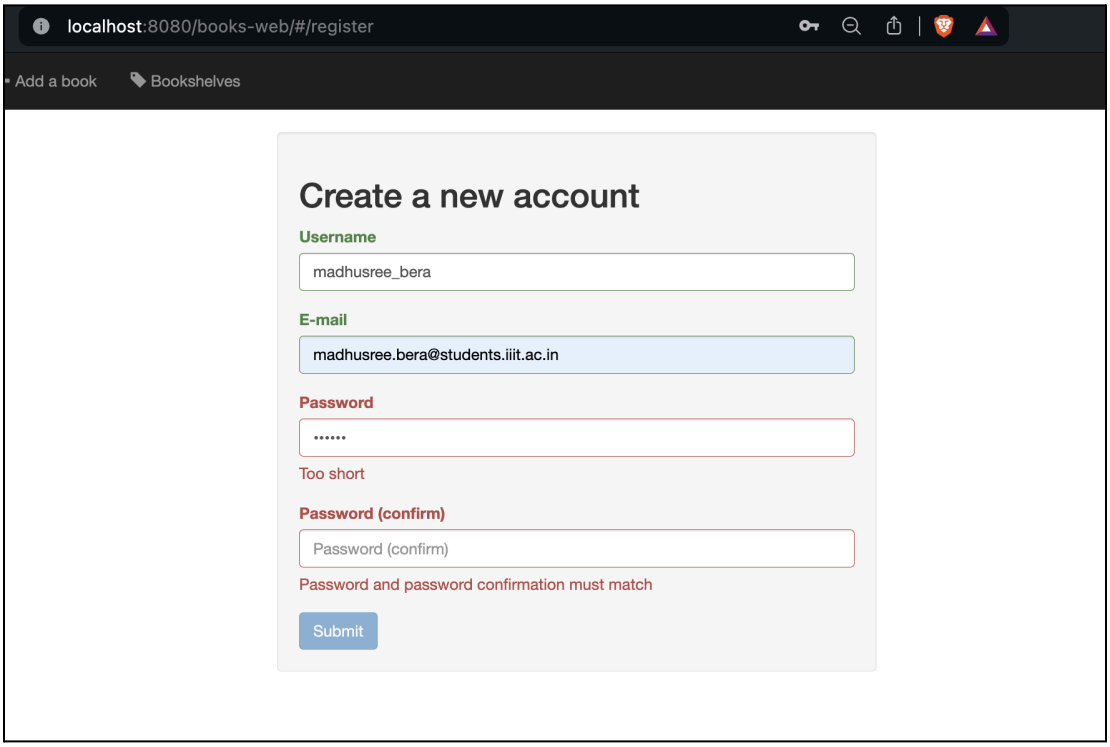
Checking unique E-mail

If same email id tries to register again, it will alert that “E-mail already used”



Validating password length and confirm password

Password length should be minimum 8 characters and maximum 50



Feature 2: Common Library

Requirements

- **R1:** A common library accessible to all users for contributing and exploring books
- **R2:** This feature allows users to add books in a common library with the required information (title, authors, genres, rating)
- **R3:** Users can rate the books on defined scale (0-10)
- **R4:** Users can view all books in the common library
- **R5:** A dynamic list displays the top 10 books based on the two criterias (average rating and number of ratings) as selected by user
- **R6:** Users can filter displayed books by authors, genres and ratings

Design Patterns:

● Singleton Pattern - Common Library

Rationale:

A single instance of the library is needed to ensure consistent data and access throughout the application. Both contribution and exploration modules would need to interact with the library's data (books). A singleton provides a central point of access. It controls library initialization (e.g., database connection) and avoids code duplication for creating multiple instances. Hence using the singleton pattern for the common library would be beneficial.

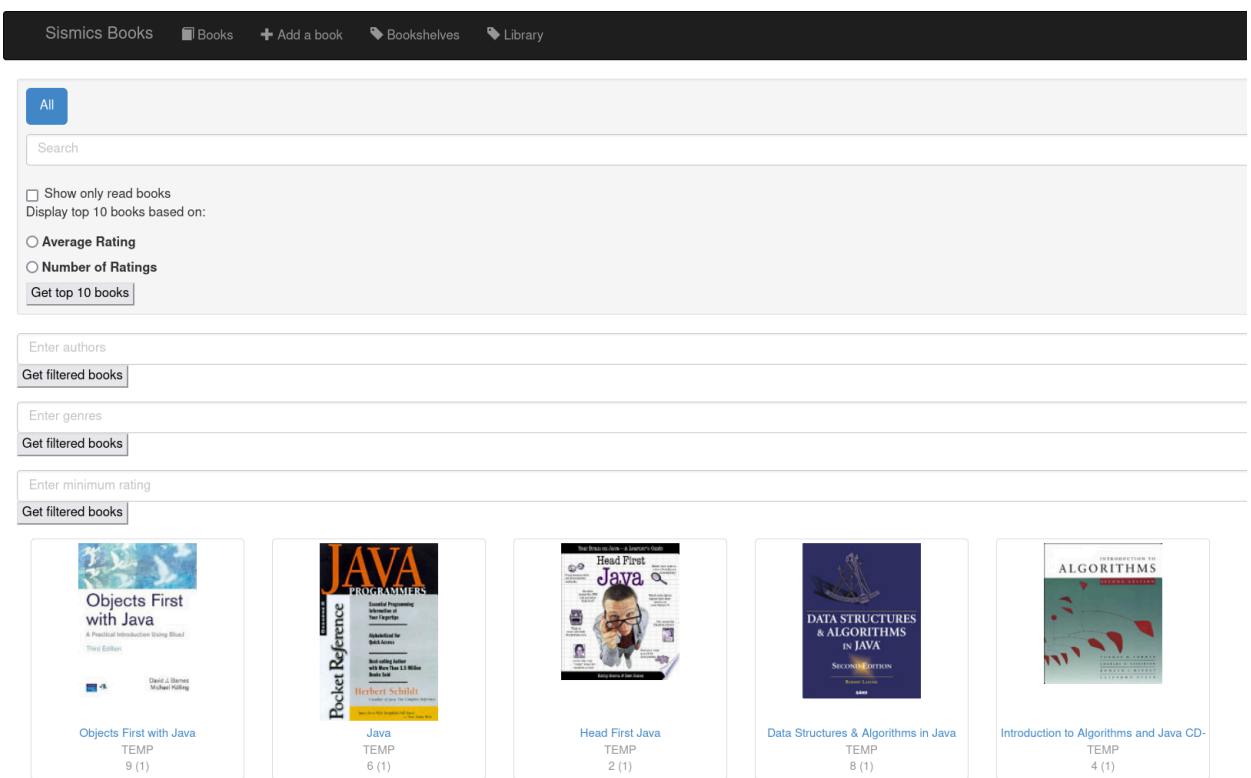
Code Snippets

LibraryResource.java

```java
package com.sismics.books.rest.resource;

import ...

@Path("/library")  Vedashree +1
public class LibraryResource extends BaseResource{

    @GET  Vedashree +1
    @Path("list")
    @Produces(MediaType.APPLICATION_JSON)
    public Response list(
            @QueryParam("limit") Integer limit,
            @QueryParam("offset") Integer offset,
            @QueryParam("sort_column") Integer sortColumn,
            @QueryParam("asc") Boolean asc,
            @QueryParam("tag") String tagName
            ) throws JSONException {...}

    @GET  Karan Bhatt +1
    @Path("top10")
    @Produces(MediaType.APPLICATION_JSON)
    public Response listTop10(
            @QueryParam("sort") String selectedCriteria
    ) throws JSONException {...}

    @GET  Vedashree +1
    @Path("filter")
    @Produces(MediaType.APPLICATION_JSON)
    public Response listFilter(
            @QueryParam("selectedCriteria") Integer selectedCriteria,
            @QueryParam("listCriteria") String listCriteria
    ) throws JSONException {...}
}
```

```java
Library.java ×
package com.sismics.books.core.model.jpa;

import ...

public class Library{  11 usages   Vedashree *

    private static Library library = new Library();  1 usage

    private List<UserBookDto> booksList;  2 usages

    public void setBooksList(List<UserBookDto> booksList) {...}

    private Library(){}

    public static Library getInstance() { return library; }

    public List<UserBookDto> getBooksList() { return booksList; }

}   Vedashree, 17/03/24, 7:09 pm • singleton pattern implemented
```

User Interface



- **Builder Pattern - Creation of Books**

Rationale:

The builder pattern is useful for creating Book objects in our library. The Book object has a long list of attributes already and we have new attributes like multiple authors and genres, and the rating.. The builder pattern allows us to construct Books in a step-by-step manner, handling optional fields gracefully. This keeps the code clean and readable, especially when dealing with potentially complex Book objects.

Extensibility:

Adding new attributes to Book objects in the future becomes straightforward. We can simply introduce new methods in the builder class without modifying the existing Book object creation logic.

Modularity:

The builder class encapsulates the Book object construction process, separating it from the Book class itself. This promotes better code organization and reduces the complexity within Book.

Maintainability:

The step-by-step construction and clear method names in the builder class make the code easier to understand and modify. If modifications are needed to Book object creation, they are localized within the builder, improving overall code maintainability.

Code Snippets

BookBuilder.java

```java
package com.sismics.books.core.model.jpa;

import ...

public abstract class BookBuilder {  15 usages  1 inheritor  Karan Bhatt *
    private String id, title, subtitle, author, genres, description, isbn10,  2 usages
    isbn13, language;  2 usages
    private Long pageCount;  2 usages
    private Date publishDate;  2 usages

    public String getId() { return id; }
    public String getTitle() { return title; }
    public String getSubtitle() { return subtitle; }
    public String getAuthor() { return author; }
    public String getGenres() { return genres; }
    public String getDescription() { return description; }
    public String getIsbn10() { return isbn10; }
    public String getIsbn13() { return isbn13; }
    public Long getPageCount() { return pageCount; }
    public String getLanguage() { return language; }
    public Date getPublishDate() { return publishDate; }
    public BookBuilder setId(String id) {...}
    public BookBuilder setTitle(String title) {...}
    public BookBuilder setSubtitle(String subtitle) {...}
    public BookBuilder setAuthor(String author) {...}
    public BookBuilder setDescription(String description) {...}
    public BookBuilder setGenres(String genres) {...}
    public BookBuilder setIsbn10(String isbn10) {...}
    public BookBuilder setIsbn13(String isbn13) {...}
    public BookBuilder setPageCount(Long pageCount) {...}
    public BookBuilder setLanguage(String language) {...}
    public BookBuilder setPublishDate(Date publishDate) {...}
    public Book build() { return new Book( bookBuilder: this); }
}
```

BookDirector.java

```java
package com.sismics.books.core.model.jpa;

import java.util.Date;

public class BookDirector {  7 usages  Karan Bhatt
    BookBuilder bookBuilder;  3 usages

    public BookDirector(BookBuilder bookBuilder) {  3 usages  Karan Bhatt
        this.bookBuilder = bookBuilder;
    }

    private Book createGenericBook(String id, String title, String subtitle,  1 usa
                                   String author, String genres, String description,
                                   String isbn10, String isbn13, Long pagecount,
                                   String language, Date publishDate) {...}

    public Book createBook(String id, String title, String subtitle,  3 usages  K
                           String author, String genres, String description,
                           String isbn10, String isbn13, Long pagecount,
                           String language, Date publishDate) {...}
}
```

Book.java

```java
public class Book {

    public Book() {  Karan Bhatt

    }

    public Book(BookBuilder bookBuilder) {  Karan Bhatt
        this.id = bookBuilder.getId();
        this.title = bookBuilder.getTitle();
        this.subtitle = bookBuilder.getSubtitle();
        this.author = bookBuilder.getAuthor();
        this.genres = bookBuilder.getGenres();
        this.description = bookBuilder.getDescription();
        this.isbn10 = bookBuilder.getIsbn10();
        this.isbn13 = bookBuilder.getIsbn13();
        this.pageCount = bookBuilder.getPageCount();
        this.language = bookBuilder.getLanguage();
        this.publishDate = bookBuilder.getPublishDate();
        //...
    }
```

User Interface

Sismics Books | Books | Add a book | Bookshelves | Library

New Book ABC

Title

New

Subtitle

Book

Author

ABC

Genre

Fiction

Description

new_book

ISBN 10

1234567890

ISBN 13

1234567890101

Page count

100

Language

English

Publication year

2024

Add | Cancel

- **Strategy Pattern - Book Ranking**

Rationale:

The strategy pattern is suitable for implementing the dynamic "BookRanking" feature in the common library. Users can choose between ranking books by average rating or number of ratings. The strategy pattern allows us to separate the ranking logic from the user interface. We can define separate strategy classes that encapsulate the logic for each ranking criterion (average rating, number of ratings). When a user selects a criterion, the library can dynamically switch between these strategies to determine the top 10 books. This provides a flexible and user-driven ranking experience.

Extensibility:

The strategy pattern provides extensibility for the Book Ranking feature. Adding new ranking criteria in the future becomes very convenient. We simply need to implement a new strategy class for the desired ranking logic (e.g., publication date, user reviews). This new strategy class would define how to sort and compare books based on the chosen criteria. The core ranking functionality in the Book Ranking component remains independent and modular. This allows us to easily incorporate new ranking options without modifying the existing logic.

Modularity:

By separating the ranking logic into distinct strategy classes, the code becomes more organized and easier to understand. Each strategy class encapsulates a specific ranking criterion, making the overall ranking functionality more modular.

Maintainability:

If modifications are needed to a specific ranking logic, changes are localized within the corresponding strategy class. This isolation improves code maintainability and reduces the risk of unintended side effects in other parts of the ranking system. Additionally, testing individual ranking strategies becomes simpler due to their separation from the core ranking component.

Code Snippets

SortingStrategy.java

```java
package com.sismics.books.core.model.jpa;

import com.sismics.books.core.dao.jpa.dto.UserBookDto;

import java.util.List;

public interface SortingStrategy {  4 usages  2 implementations  Vedashree +1
    public List<UserBookDto> sortBooks(List<UserBookDto> booksList);  1 usage
}
```

AverageRatingSortingStrategy.java

```java
package com.sismics.books.core.model.jpa;
import com.sismics.books.core.dao.jpa.UserBookRatingDao;
import com.sismics.books.core.dao.jpa.dto.UserBookDto;

import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class AverageRatingSortingStrategy implements SortingStrategy{  2 usages  Vedashree +1 *
    @Override  1 usage  Karan Bhatt +1 *
    public List<UserBookDto> sortBooks(List<UserBookDto> booksList) {
        List<UserBookDto> sortedBooksList = booksList;

        Collections.sort(sortedBooksList, new Comparator<UserBookDto>() {  Karan Bhatt +1 *
            @Override  Karan Bhatt
            public int compare(UserBookDto b1, UserBookDto b2) {
                return Double.compare(UserBookRatingDao.getAvgRatingByID(b1.getId()),
                        UserBookRatingDao.getAvgRatingByID(b2.getId()));
            }
        });
        return sortedBooksList;
    }
}
```

NumberOfRatingsSortingStrategy.java

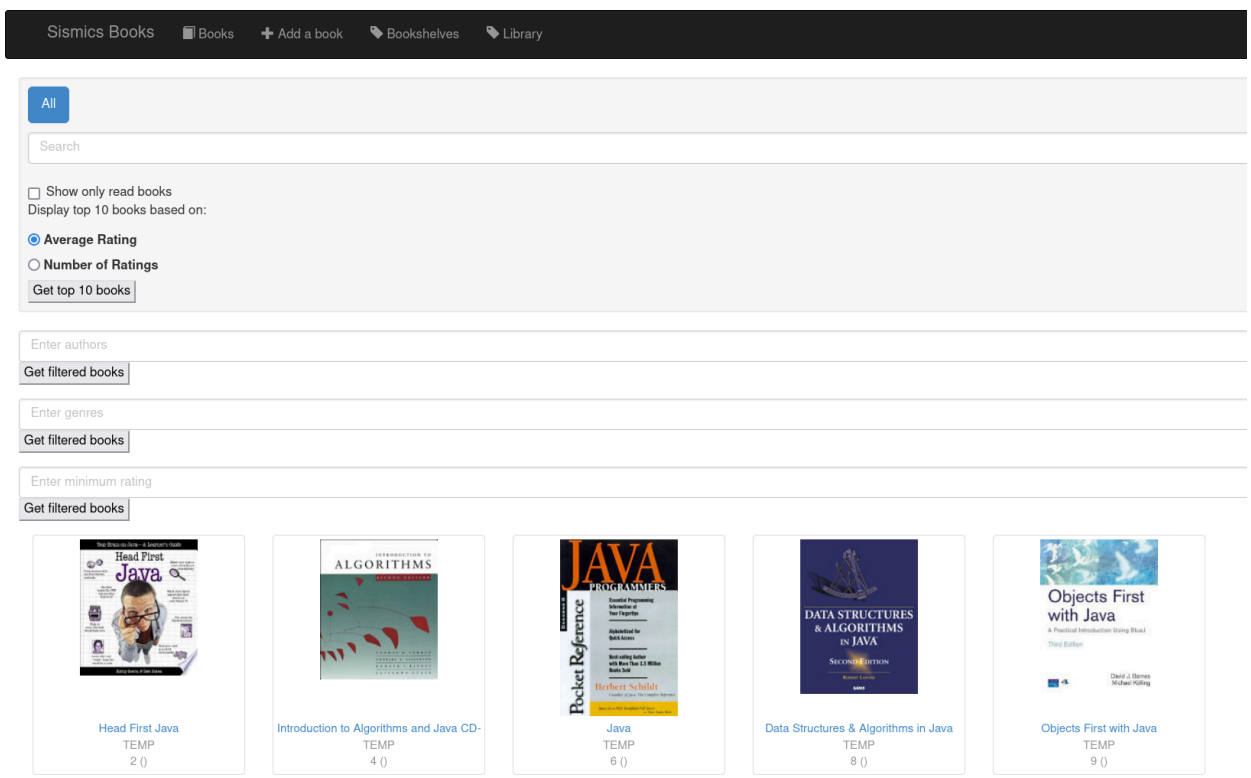
```java
package com.sismics.books.core.model.jpa;
import ...
public class NumberOfRatingsSortingStrategy implements SortingStrategy{  1 usage  Vedashree +1 *
    @Override  1 usage  Karan Bhatt +1 *
    public List<UserBookDto> sortBooks(List<UserBookDto> booksList) {
        List<UserBookDto> sortedBooksList = booksList;

        Collections.sort(sortedBooksList, new Comparator<UserBookDto>() {  Karan Bhatt +1 *
            @Override  Karan Bhatt
            public int compare(UserBookDto b1, UserBookDto b2) {
                return Long.compare(UserBookRatingDao.getNumRatingsByID(b1.getId()),
                        UserBookRatingDao.getNumRatingsByID(b2.getId()));
            }
        });
        return sortedBooksList;
    }
}
```

Top10List.java

```java
package com.sismics.books.core.model.jpa;

import ...

public class Top10List {  7 usages  Karan Bhatt *
    public List<UserBookDto> getTop10Books(List<UserBookDto> booksList,  1 usage
                                           SortingStrategy strategy){

        List<UserBookDto> sortedBooksList = strategy.sortBooks(booksList);
        return sortedBooksList.subList(i: 0, Math.min(sortedBooksList.size(), 10));
    }
}
```

```java
LibraryResource.java ×
29      public class LibraryResource extends BaseResource{
99          public Response listTop10(
126                 booksList = library.getBooksList();
127             } catch (Exception e) {
128                 throw new ServerException("Top10Error", "Error getting top 10 books", e);
129             }
130
131             // sort logic
132             SortingStrategy sortingStrategy;
133             if (selectedCriteria.equals("avg_rating")) {
134                 sortingStrategy = new AverageRatingSortingStrategy();
135             }
136             else {
137                 sortingStrategy = new NumberOfRatingsSortingStrategy();
138             }
139             Top10List top10List = new Top10List();
140
141             List<UserBookDto> top10books = top10List.getTop10Books(booksList, sortingStrategy);
```

User Interface



● Criteria / Filter Pattern - Filtering Books

Rationale:

The criteria pattern is a powerful approach for implementing user-driven book filtering in the common library. Users can filter books based on various criteria, including authors (multi-select), genres (multi-select), and ratings (single select with options like ">6", ">7", etc.). The criteria pattern allows us to encapsulate each filtering condition into separate criteria objects. When a user selects filters, the library can combine these criteria objects to determine the books that match all specified conditions. This approach offers a flexible and user-friendly filtering experience.

Extensibility:

Introducing new filtering options in the future becomes straightforward. We simply need to create a new criteria class for the desired condition. This new class would define how to check if a book meets the specific criteria. The core filtering functionality remains modular and can seamlessly integrate the new criteria without code modifications.Also the criteria objects can potentially be reused across different parts of the library if filtering needs arise in other contexts. This promotes code reusability and reduces redundancy.

Modularity:

Separating filtering logic into individual criteria classes enhances code organization. Each class represents a specific filtering condition, making the overall filtering functionality more modular and easier to understand.

Maintainability:

If modifications are needed to a particular filtering criteria, changes are localized within the corresponding criteria class. This isolation improves code maintainability and reduces the risk of unintended side effects in other parts of the filtering system. Additionally, testing individual criteria becomes simpler due to their separation from the core filtering logic.

Code Snippets

FilterCriteria.java

```java
package com.sismics.books.core.model.jpa;

import com.sismics.books.core.dao.jpa.dto.UserBookDto;

public interface FilterCriteria {  4 usages  3 implementations  Vedashree
    boolean meetsCriteria(UserBookDto book);  1 usage  3 implementations  Vedashree
}
```

AuthorFilterCriteria.java

```java
package com.sismics.books.core.model.jpa;

import ...

public class AuthorFilterCriteria implements FilterCriteria {  1 usage  Vedashree
    private List<String> authors;  2 usages

    public AuthorFilterCriteria(List<String> authors) { this.authors = authors; }

    @Override  1 usage  Vedashree
    public boolean meetsCriteria(UserBookDto book) {

        if (authors.contains(book.getAuthor())) {
            return true;
        }
        return false;
    }
}
```

GenreFilterCriteria.java

```java
package com.sismics.books.core.model.jpa;

import ...

public class GenreFilterCriteria implements FilterCriteria{  1 usage  Vedashree
    private List<String> genres;  2 usages

    public GenreFilterCriteria(List<String> genres) { this.genres = genres; }

    @Override  1 usage  Vedashree
    public boolean meetsCriteria(UserBookDto book) {
        String[] items = book.getGenres().split( regex: ",");
        List<String> bookGenres = Arrays.asList(items);
        for (String a: bookGenres) {
            if (!genres.contains(a)) {
                return false;
            }
        }
        return true;
    }
}
```

RatingFilterCriteria.java

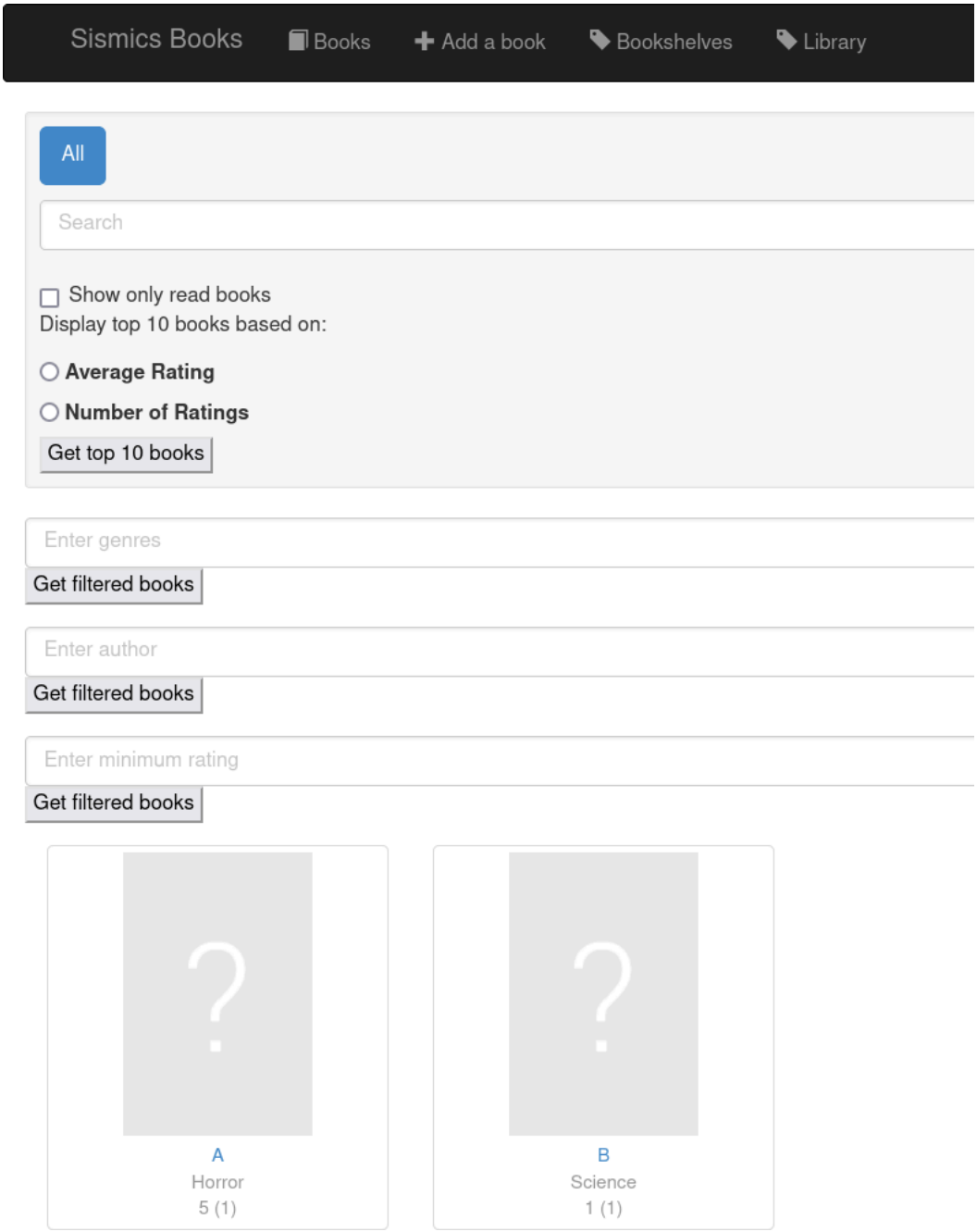
```java
package com.sismics.books.core.model.jpa;

import ...

public class RatingFilterCriteria implements FilterCriteria{  1 usage  Vedashree
    private int minRating;  2 usages

    public RatingFilterCriteria(int minRating) { this.minRating = minRating; }

    @Override  1 usage  Vedashree
    public boolean meetsCriteria(UserBookDto book) {
        return UserBookRatingDao.getAvgRatingByID(book.getId()) >= minRating;
    }
}
```

LibraryResource.java

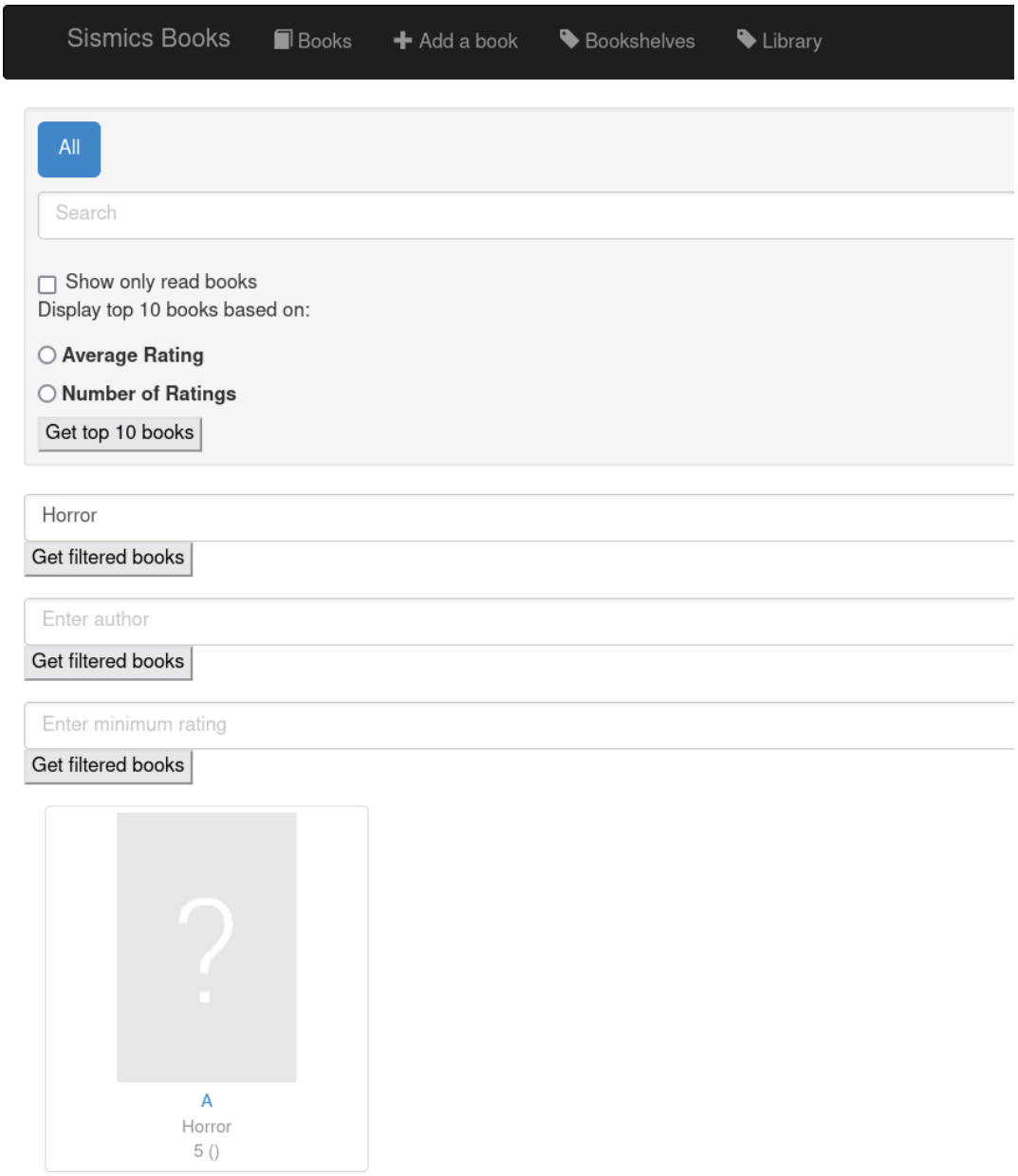
```java
public class LibraryResource extends BaseResource{
    public Response listFilter(
        }

        // filter logic
        String[] items = listCriteria.split(regex: ",");
        List<String> itemList = Arrays.asList(items);
        System.out.println("Items:" + items);
        FilterCriteria filterCriteria;
        if(selectedCriteria == 1){
            filterCriteria = new AuthorFilterCriteria(itemList);
        } else if (selectedCriteria == 2) {
            filterCriteria = new GenreFilterCriteria(itemList);
        }
        else{filterCriteria = new RatingFilterCriteria(Integer.parseInt(listCriteria));}

        List<UserBookDto> filteredBooks = new ArrayList<>();
        for (UserBookDto book: booksList) {
            if (filterCriteria.meetsCriteria(book)) {
                filteredBooks.add(book);
            }
        }
```

User Interface

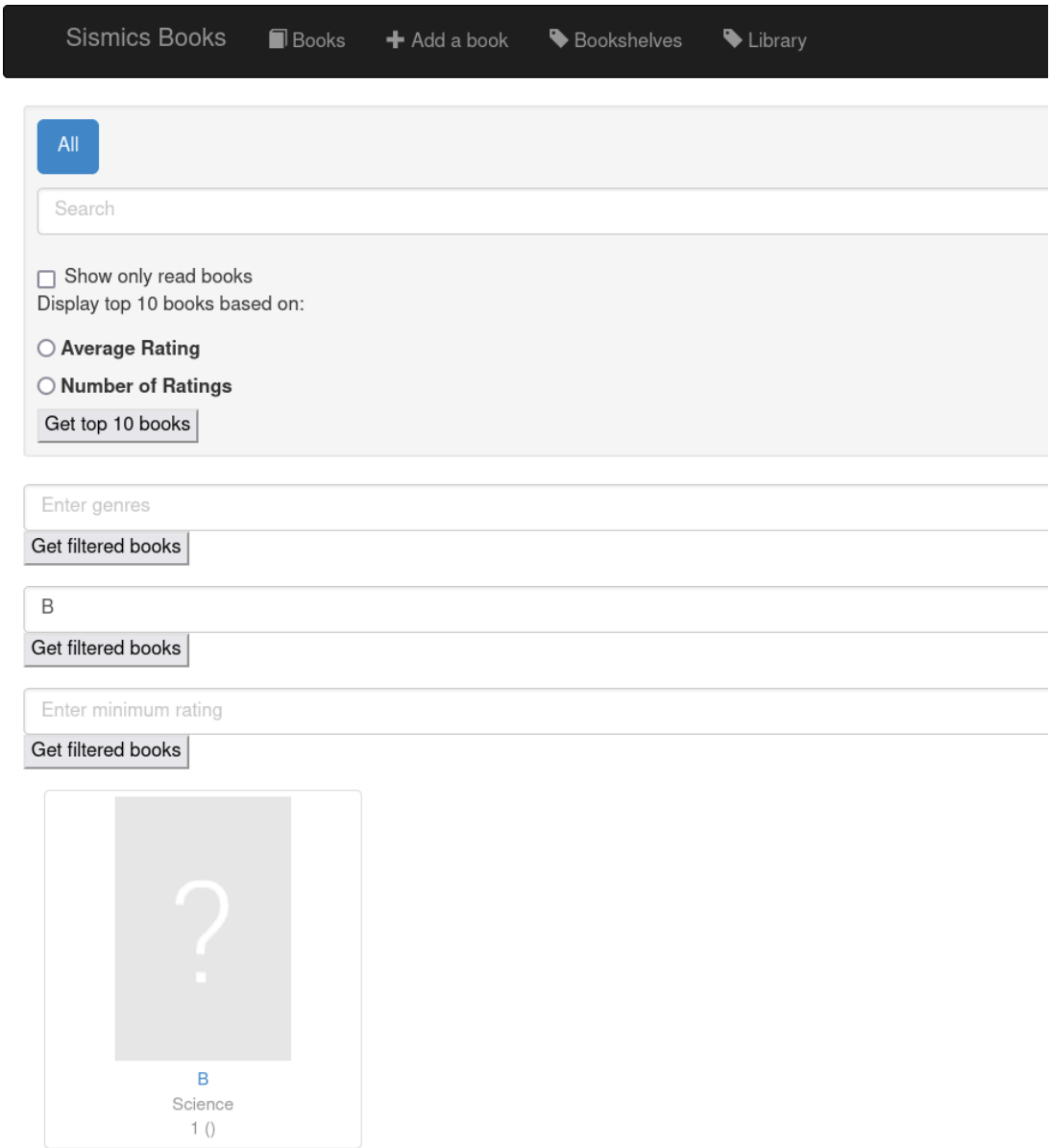
Books in the common library before applying filtering



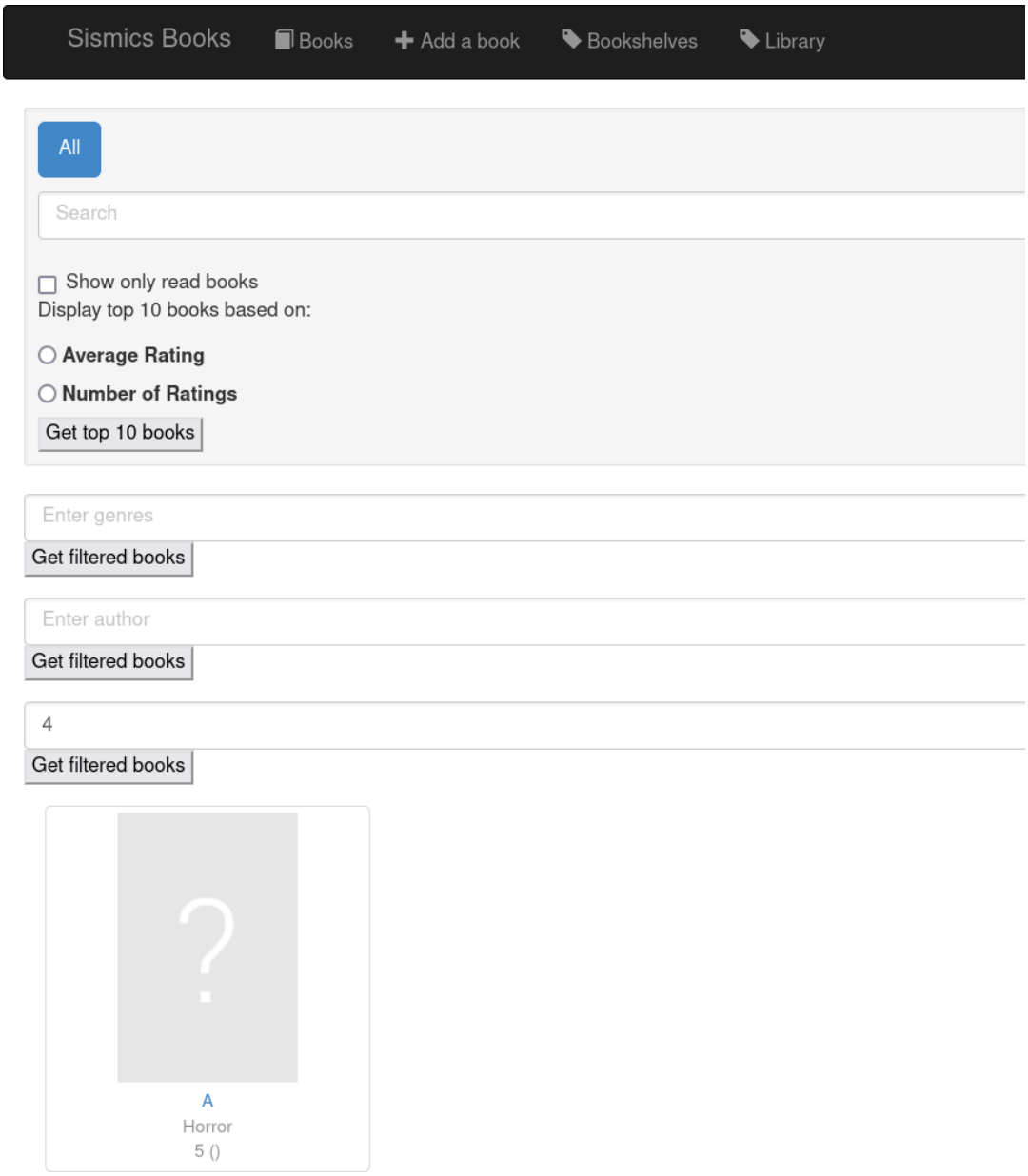
Applying filter by genre



Applying filter by author



Applying filter by rating



Feature 3: Online Integration

Requirements

- **R1:** Users can select either Spotify or iTunes as the service provider for accessing audiobooks and podcasts.
- **R2:** Users can choose between audiobooks and podcasts within the selected service provider.
- **R3:** Users can search using the search bar by providing a simple string and the results get displayed
- **R4:** Users can mark any audiobook or podcast as favorite
- **R5:** Users can access the books marked as favorites in a separate favorites section.

Design Patterns:

● Strategy - Selection of Content type selection

Rationale:

The strategy pattern allows us to define separate strategies for searching and retrieving audiobooks or podcasts based on the user's selection. Each strategy class would encapsulate the logic for crafting appropriate search queries or filtering retrieved results specific to the chosen content type. This ensures the library uses the correct parameters when interacting with the service provider's API.

Separating content type logic into distinct strategies promotes clean and maintainable code. The core library logic remains focused on handling user interactions and managing search results, while the strategies handle the nuances of searching for audiobooks or podcasts within the chosen service provider. This separation improves code readability and reduces the risk of introducing errors when handling different content types.

Extensibility:

If the service providers offer additional content types (e.g., music videos) in the future, we could potentially leverage the strategy pattern. We would need to develop a new strategy class (e.g., MusicVideoSearchStrategy) that understands how to search for that specific content type using the service provider's API. This modular approach allows for future expansion without modifying the core library logic.

Modularity:

By separating content type logic into distinct strategies, the code becomes more modular and easier to understand. Each strategy class focuses on a specific content type, improving code organization and clarity.

Maintainability:

If modifications are needed to how audiobooks or podcasts are searched within a service provider, changes are localized within the corresponding strategy class. This isolation improves code maintainability and reduces the risk of unintended side effects in other parts of the search functionality. Additionally, testing individual content type strategies becomes simpler due to their separation from the core library logic.

Code Snippets

ItunesController.js

```
class Strategy {
  search(searchString) {
    throw new Error("Method 'search' must be implemented.");
  }
}
class PodcastStrategy extends Strategy {
  search(searchString) {
    var baseUrl = "https://itunes.apple.com/search";
    var term = encodeURIComponent(searchString);
    var country = "US";
    var media = "podcast";
    var limit = 10;
    var url = `${baseUrl}?term=${term}&country=${country}&media=${media}&l
    $http
      .get(url)
      .then(function (response) {
        if (response.status !== 200) {
          alert("request failed with status " + response.status);
          throw new Error("Request failed with status " + response.status)
        }

        const adapter = new DataAdapter(response.data.results);
        const adaptedData = adapter.adapt();
        $scope.searchResults = createTableMarkup(adaptedData);
      })
      .catch(function (error) {
        alert("request failed");
        console.error("Error fetching data:", error);
      });
  }
}
```

```
class AudiobookStrategy extends Strategy {
  search(searchString) {
    var baseUrl = "https://itunes.apple.com/search";
    var term = encodeURIComponent(searchString);
    var country = "US";
    var media = "audiobook";
    var limit = 10;
    var url = `${baseUrl}?term=${term}&country=${country}&media=${media}&l
    $http
      .get(url)
      .then(function (response) {
        if (response.status !== 200) {
          alert("request failed with status " + response.status);
          throw new Error("Request failed with status " + response.status)
        }
        const adapter = new DataAdapter(response.data.results);
        const adaptedData = adapter.adapt();
        $scope.searchResults = createTableMarkup(adaptedData);
      })
      .catch(function (error) {
        alert("request failed");
        console.error("Error fetching data:", error);
      });
  }
}
```

```
function createTableMarkup(data) {
  var markup = "<table class='table'>";
  markup +=
    "<thead><tr><th>Track ID</th><th>Artist Name</th><th>Collection Name</
  markup += "<tbody>";
  data.forEach(function (item) {
    markup += "<tr>";
    markup += "<td>" + item.trackId + "</td>";
    markup += "<td>" + item.artistName + "</td>";
    markup += "<td>" + item.collectionName + "</td>";
    markup += "<td>" + item.trackName + "</td>";
    markup += "<td>" + item.collectionCensoredName + "</td>";
    markup += "</tr>";
  });
  markup += "</tbody></table>";
  return markup;
}

$scope.searchQuery = function () {
  let strategy;
  if ($scope.searchType === "podcast") {
    strategy = new PodcastStrategy();
  } else if ($scope.searchType === "audiobook") {
    strategy = new AudiobookStrategy();
  }

  strategy.search(
    $scope.searchKeywords,
    // $scope.searchType,
    $http,
    function (adaptedData) {
      $scope.searchResults = createTableMarkup(adaptedData);
      $scope.$apply();
    }
  );
```

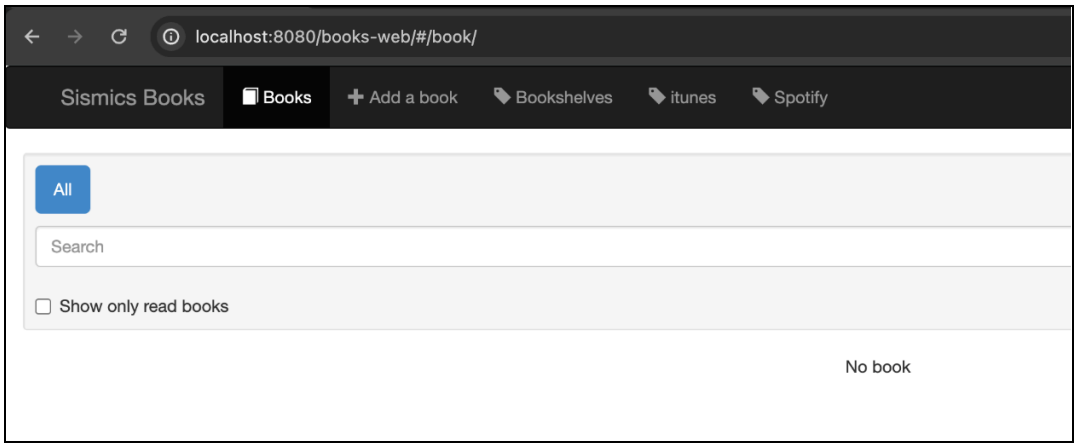
SpotifyController.js

```
class Strategy {
  search(searchString) {
    throw new Error("Method 'search' must be implemented.");
  }
}

// PodcastStrategy subclass
class PodcastStrategy extends Strategy {
  search(searchString) {
    var apiUrl = "https://api.spotify.com/v1/search";
    $http({
      method: "GET",
      url: apiUrl,
      headers: {
        Authorization: "Bearer " + accessToken,
      },
      params: {
        q: searchString,
        type: "show",
        market: "US",
        limit: 10,
      },
    })
      .then(function (response) {
        if (response.status === 200) {
          alert(JSON.stringify(response.data.shows.href));
          // alert(response.data);
          // var adaptedData = spotifyAdapter(response.data.shows.items);
          // alert(adaptedData);
          // $scope.searchResults = createTableMarkup(adaptedData);
        } else {
          console.error("Request failed with status: ", response.status);
        }
```

```
class AudiobookStrategy extends Strategy {
  search(searchString) {
    var apiUrl = "https://api.spotify.com/v1/search";
    $http({
      method: "GET",
      url: apiUrl,
      headers: {
        Authorization: "Bearer " + accessToken,
      },
      params: {
        q: searchString,
        type: "audiobook",
        market: "US",
        limit: 10,
      },
    })
      .then(function (response) {
        if (response.status === 200) {
          // alert(JSON.stringify(response.data.audiobooks.items));
          var adaptedData = spotifyAdapter(response.data.audiobooks.items)
          // alert(adaptedData);
          // const adapter = new DataAdapter(response.data.audiobooks.item
          // const adaptedData = adapter.adapt();
          $scope.searchResults = createTableMarkup(adaptedData);
        } else {
          console.error("Request failed with status: ", response.status);
        }
      })
      .catch(function (error) {
        console.error("Error fetching audiobooks: ", error);
      });
  }
}
```

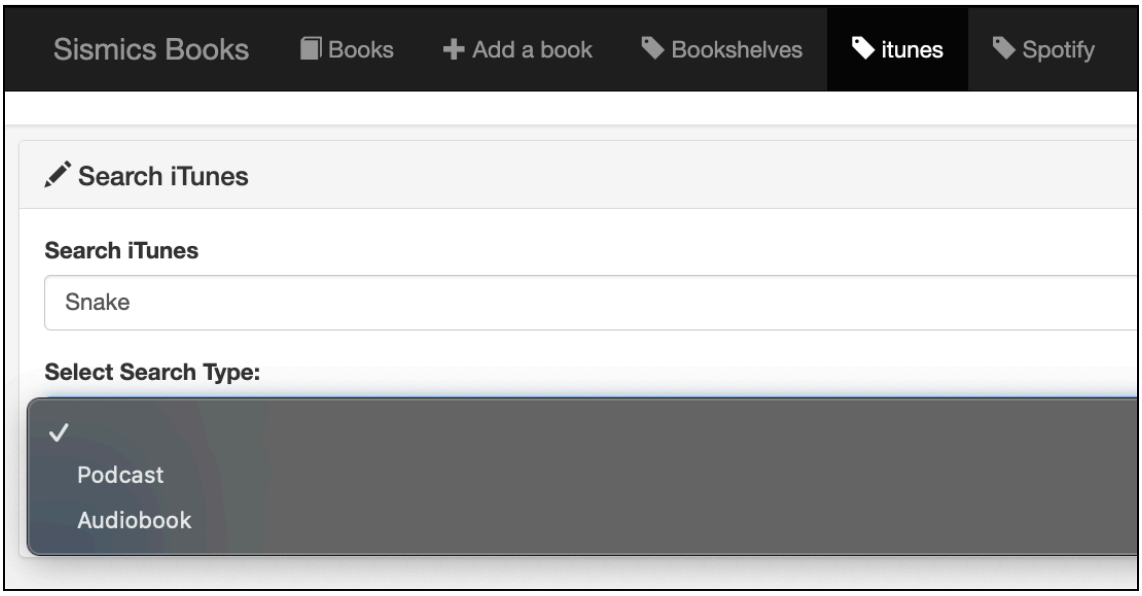
User Interface

Navbar containing ITunes and Spotify option



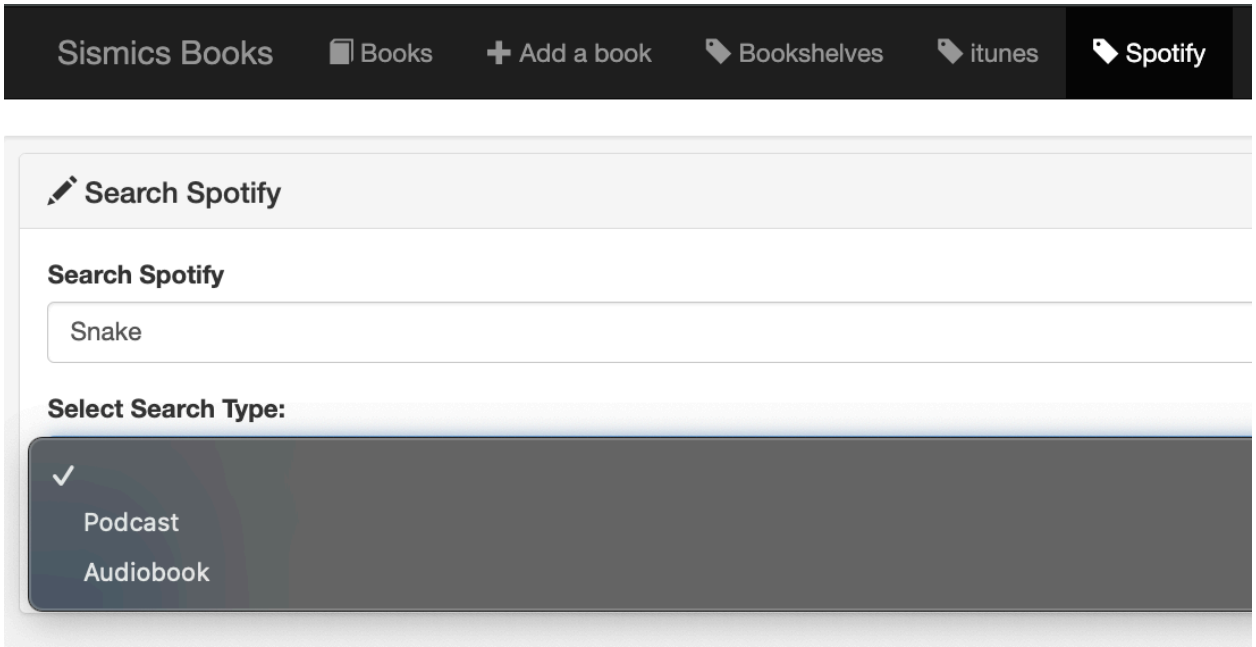
Dropdown to choose Podcasts or Audiobooks

When we need to search using these two, we can ask for both “Audiobooks” and “Podcasts” using Dropdown.



And depending on our search type, we will get our result.

Similarly, in the case of Spotify, based on the value in Dropdown we choose, we get our desired result.



Note that our result will require the Adapter pattern. Hence, our result will be shown in Adapter Pattern.

- **<u>Adapter Pattern - Representing the results as strings</u>**

Rationale:

The adapter pattern can be a well-suited approach for implementing online integration with Spotify and iTunes in the common library for audiobooks and podcasts. Spotify and iTunes have different APIs for accessing content. The adapter pattern allows us to create a unified interface for interacting with both services. We can develop separate adapters for each API, each translating the specific API calls into a common set of methods understood by the library. This allows the library to work seamlessly with either service without requiring code modifications for each API.
Using adapters simplifies the integration process by encapsulating the complexity of each API within its respective adapter. This makes the core library code cleaner and easier to understand, as it interacts only with the common interface provided by the adapters.

Extensibility:

If we want to integrate with additional audiobook or podcast providers in the future, the adapter pattern makes it straightforward. We simply need to develop a new adapter for the new service's API, following the same principles as the existing adapters. This modular approach allows us to expand the platform's functionality without modifying the core library logic.

Modularity:

By separating the logic for interacting with each service into distinct adapters, the code becomes more modular and easier to understand. Each adapter focuses on a specific API, promoting better code organization.

Maintainability:

If changes are needed to accommodate updates or modifications in either Spotify's API or iTunes's API, changes are localized within the corresponding adapter. This isolation improves code maintainability and reduces the risk of unintended side effects in other parts of the integration system. Additionally, testing individual adapters becomes simpler due to their separation from the core library logic.

Code Snippets

ItunesController.js

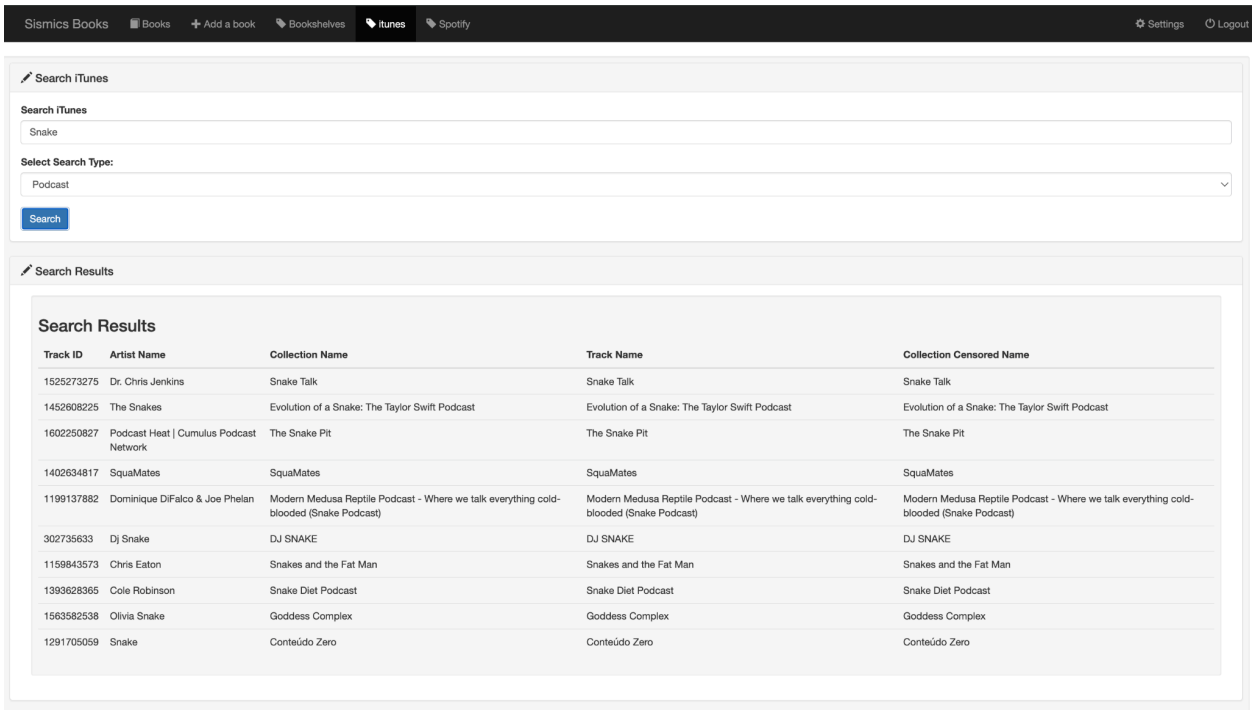
```
class DataAdapter {
  constructor(data) {
    this.data = data;
  }

  adapt() {
    return this.data.map((item) => ({
      trackId: item.trackId,
      artistName: item.artistName,
      collectionName: item.collectionName,
      trackName: item.trackName,
      collectionCensoredName: item.collectionCensoredName,
    }));
  }
}
```

SpotifyController.js

```
class SpotifyDataAdapter {
  constructor(data) {
    this.data = data;
  }
  adapt() {
    return data.map(function (item) {
      return {
        trackId: item.id,
        artistName: item.authors[0]?.name, // Using the first author's nam
        collectionName: item.name,
        trackName: item.name, // Assuming you want to use the audiobook's
      };
    });
  }
}
```

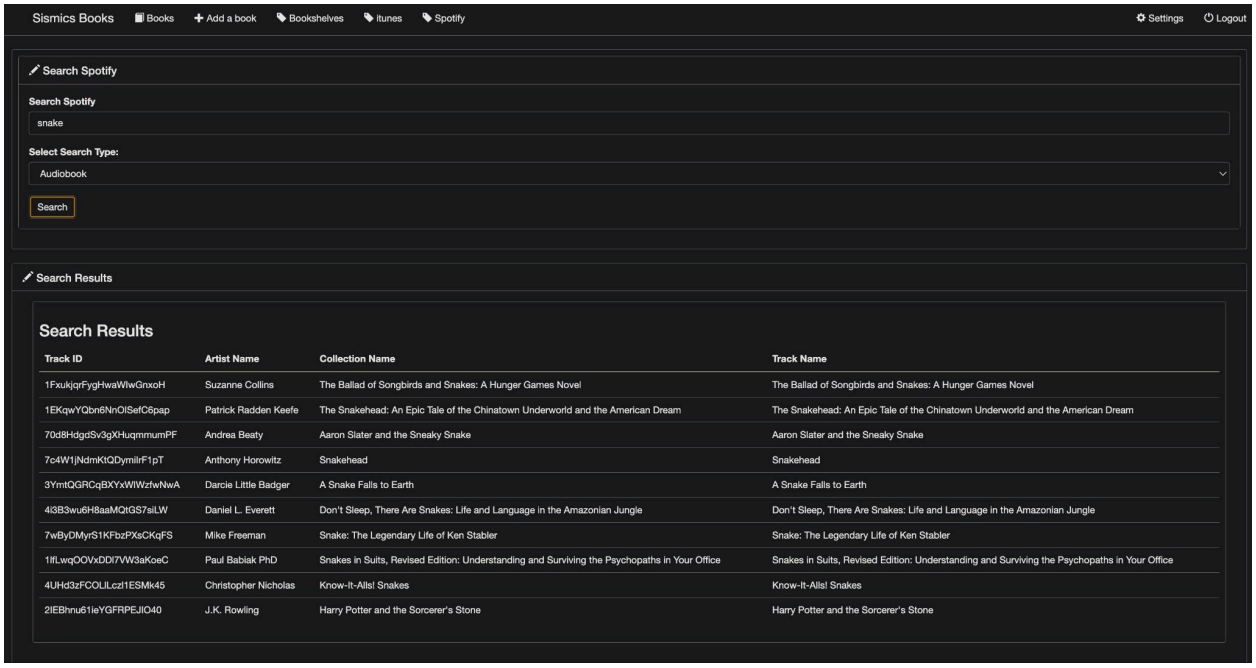
User Interface

ITunes Search Result



Track ID	Artist Name	Collection Name	Track Name	Collection Censored Name
1525273275	Dr. Chris Jenkins	Snake Talk	Snake Talk	Snake Talk
1452608225	The Snakes	Evolution of a Snake: The Taylor Swift Podcast	Evolution of a Snake: The Taylor Swift Podcast	Evolution of a Snake: The Taylor Swift Podcast
1602250827	Podcast Heat | Cumulus Podcast Network	The Snake Pit	The Snake Pit	The Snake Pit
1402634817	SquaMates	SquaMates	SquaMates	SquaMates
1199137882	Dominique DiFalco & Joe Phelan	Modern Medusa Reptile Podcast - Where we talk everything cold-blooded (Snake Podcast)	Modern Medusa Reptile Podcast - Where we talk everything cold-blooded (Snake Podcast)	Modern Medusa Reptile Podcast - Where we talk everything cold-blooded (Snake Podcast)
302735633	Dj Snake	DJ SNAKE	DJ SNAKE	DJ SNAKE
1159843573	Chris Eaton	Snakes and the Fat Man	Snakes and the Fat Man	Snakes and the Fat Man
1393628365	Cole Robinson	Snake Diet Podcast	Snake Diet Podcast	Snake Diet Podcast
1563582538	Olivia Snake	Goddess Complex	Goddess Complex	Goddess Complex
1291705059	Snake	Conteúdo Zero	Conteúdo Zero	Conteúdo Zero

Spotify Search Result



Track ID	Artist Name	Collection Name	Track Name
1FxukjqrFygHwaWlwGnxoH	Suzanne Collins	The Ballad of Songbirds and Snakes: A Hunger Games Novel	The Ballad of Songbirds and Snakes: A Hunger Games Novel
1EKqwYQbn6NnOlSefC6pap	Patrick Radden Keefe	The Snakehead: An Epic Tale of the Chinatown Underworld and the American Dream	The Snakehead: An Epic Tale of the Chinatown Underworld and the American Dream
70d8HdgdSv3gXHuqmmumPF	Andrea Beaty	Aaron Slater and the Sneaky Snake	Aaron Slater and the Sneaky Snake
7c4W1jNdmKtQDymilrF1pT	Anthony Horowitz	Snakehead	Snakehead
3YmtQGRCqBXYxWlWzfwNwA	Darcie Little Badger	A Snake Falls to Earth	A Snake Falls to Earth
4i3B3wu6H8aaMQtGS7siLW	Daniel L. Everett	Don't Sleep, There Are Snakes: Life and Language in the Amazonian Jungle	Don't Sleep, There Are Snakes: Life and Language in the Amazonian Jungle
7wByDMyrS1KFbzPXsCKqFS	Mike Freeman	Snake: The Legendary Life of Ken Stabler	Snake: The Legendary Life of Ken Stabler
1fILwqOOVxDDi7VW3aKoeC	Paul Babiak PhD	Snakes in Suits, Revised Edition: Understanding and Surviving the Psychopaths in Your Office	Snakes in Suits, Revised Edition: Understanding and Surviving the Psychopaths in Your Office
4UHd3zFCOLiLczl1ESMk45	Christopher Nicholas	Know-It-Alls! Snakes	Know-It-Alls! Snakes
2IEBhnu61ieYGFRPEJIO40	J.K. Rowling	Harry Potter and the Sorcerer's Stone	Harry Potter and the Sorcerer's Stone

Contribution

Name	Feature
Madhusree Bera	Feature 1 (Better user management)
Karan Bhatt	Feature 2 (Common Library)
Vedashree Ranade	Feature 2 (Common Library)
Yash Maheshwari	Feature 3 (Online Integration)
Piyush Rana	Feature 3 (Online Integration)