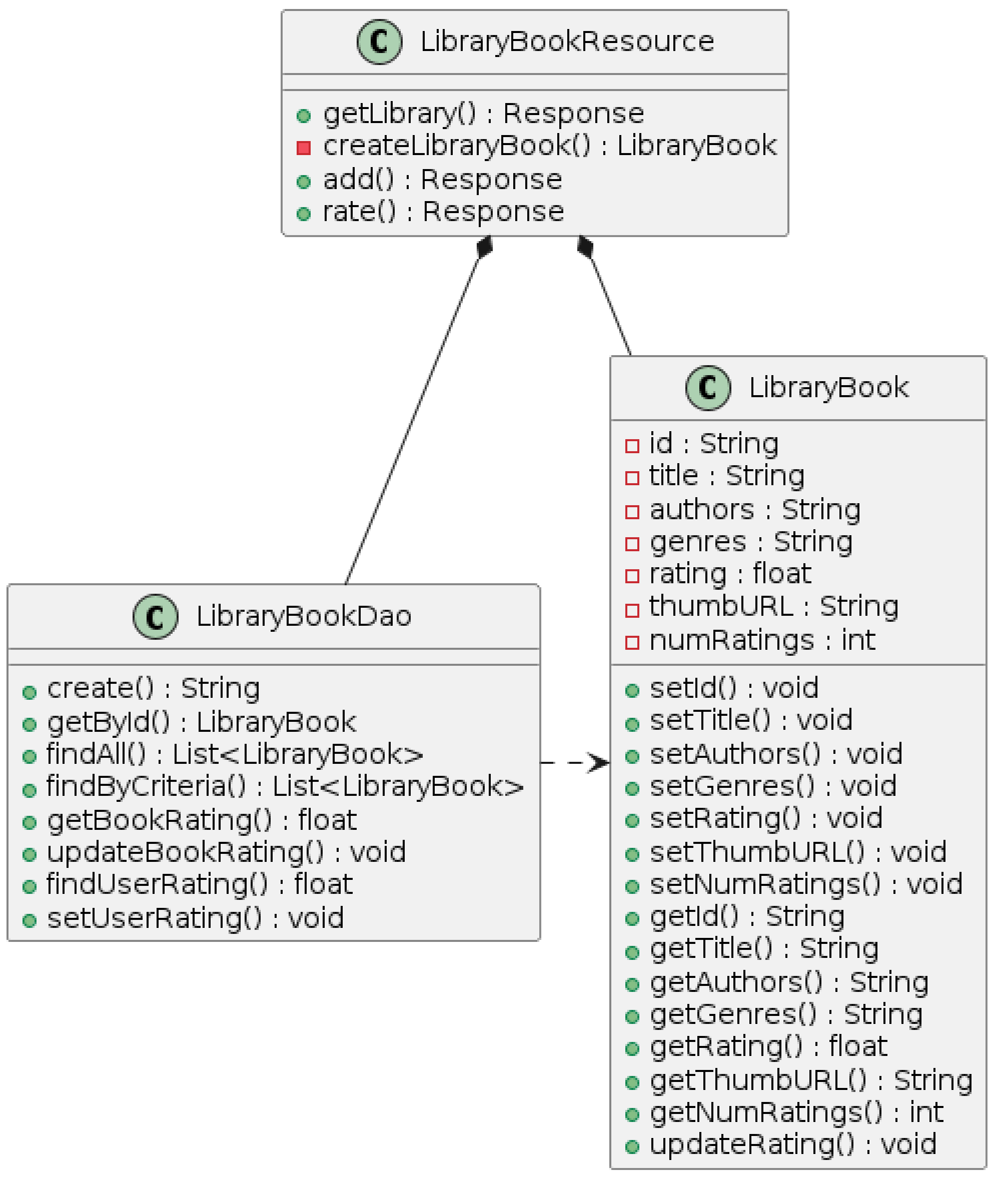
Software Engineering | Project 2 | Team 15

Common Library

Features: Adding Book Details, Implementing User Management Functionalities, Book Ranking and Filtering

Core class:

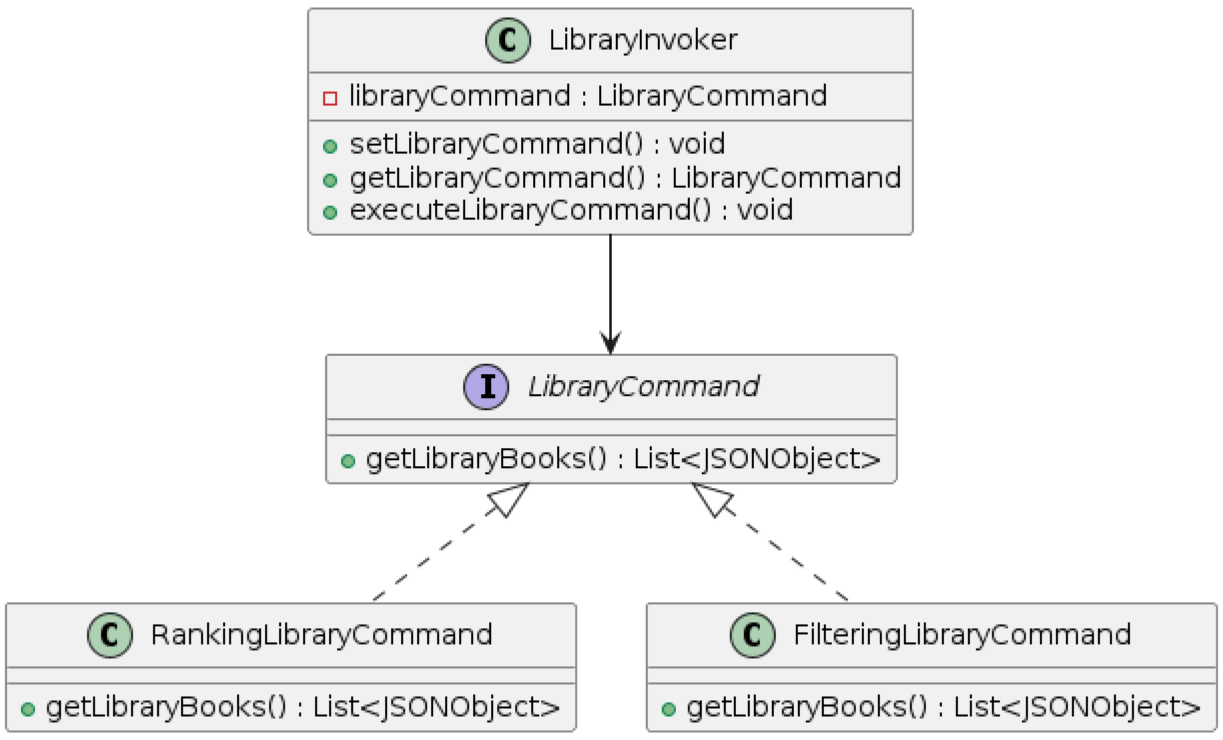


- `LibraryBook` is the class representing the entity which the entire system is centered around.
- `LibraryDao` provides the interface to the database.
- `LibraryBookResource` acts as the "client" for most operations, and provides an endpoint for the RESTful API to the frontend.

Other classes and interfaces are for better readability, flexibility, understandability, and extensibility. The core functionality revolves around the above 3 classes.

Design Patterns Used:

1. Command Pattern


LibraryInvoker
libraryCommand : LibraryCommand
setLibraryCommand() : void
getLibraryCommand() : LibraryCommand
executeLibraryCommand() : void
LibraryCommand
getLibraryBooks() : List<JSONObject>
RankingLibraryCommand
getLibraryBooks() : List<JSONObject>
FilteringLibraryCommand
getLibraryBooks() : List<JSONObject>

Participants

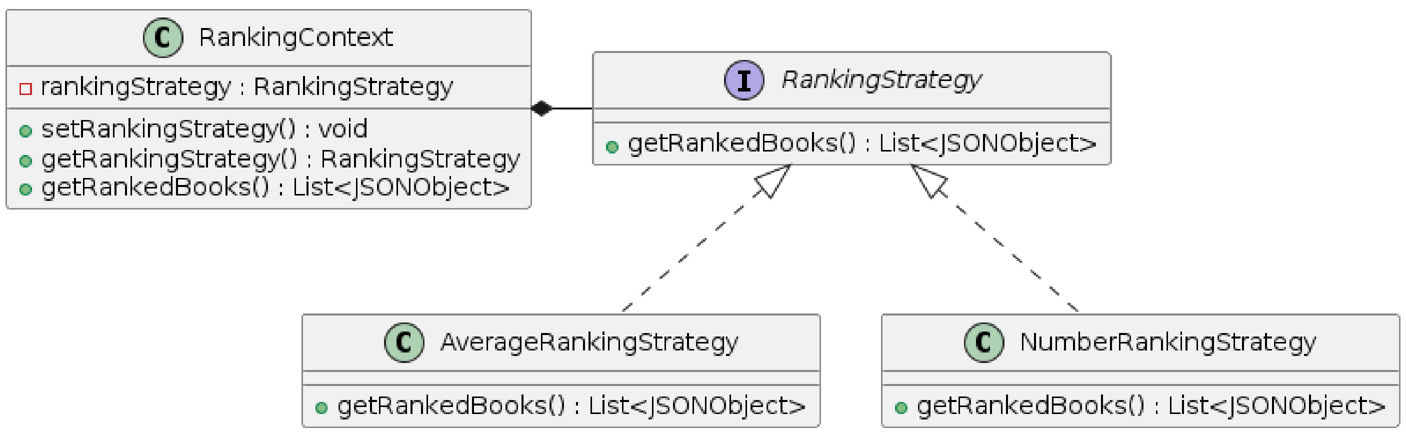
1. Invoker — `LibraryInvoker` — Maintains a reference to one of the concrete commands and invokes the referenced command
2. Command Interface — `LibraryCommand` — It is the interface common to all the different commands. It declares a method which the invoker calls.
3. Concrete Commands — `RankingLibraryCommand, FilteringLibraryCommand` — Implement different functionalities that can be called from the same place.
4. Client — `LibraryBookResource` — The Client here creates a specific command object and passes it to the invoker. The invoker exposes a setter which lets clients choose which command they want to execute. This choice is made by the request coming from the front-end.

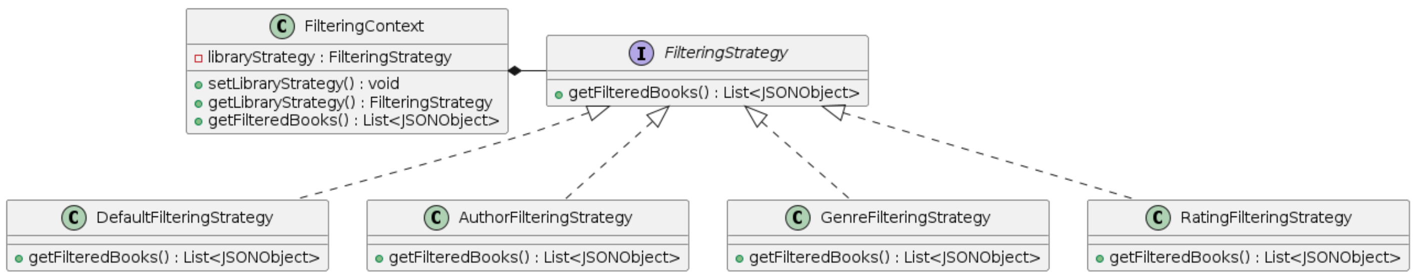
Motivation behind various design decisions

- **Why Command:** Implementing the Command design pattern for our repository offers several benefits:
  - **Separation of Concerns:** By encapsulating requests as objects, the Command pattern separates the requester (Invoker) from the object that performs the action (Command). This separation simplifies the architecture by decoupling the components, making it easier to maintain and extend.
  - **Flexibility and Extensibility:** With Command objects representing different actions (i.e., ranking and filtering), we can easily add new commands without modifying existing code. This flexibility allows you to introduce new features or variations in behavior without disrupting the existing system.

2. Strategy Pattern

RankingStrategy & FilteringStrategy


RankingContext
rankingStrategy : RankingStrategy
setRankingStrategy() : void
getRankingStrategy() : RankingStrategy
getRankedBooks() : List<JSONObject>
RankingStrategy
getRankedBooks() : List<JSONObject>
AverageRankingStrategy
getRankedBooks() : List<JSONObject>
NumberRankingStrategy
getRankedBooks() : List<JSONObject>


FilteringContext
libraryStrategy : FilteringStrategy
setLibraryStrategy() : void
getLibraryStrategy() : FilteringStrategy
getFilteredBooks() : List<JSONObject>
FilteringStrategy
getFilteredBooks() : List<JSONObject>
DefaultFilteringStrategy
getFilteredBooks() : List<JSONObject>
AuthorFilteringStrategy
getFilteredBooks() : List<JSONObject>
GenreFilteringStrategy
getFilteredBooks() : List<JSONObject>
RatingFilteringStrategy
getFilteredBooks() : List<JSONObject>

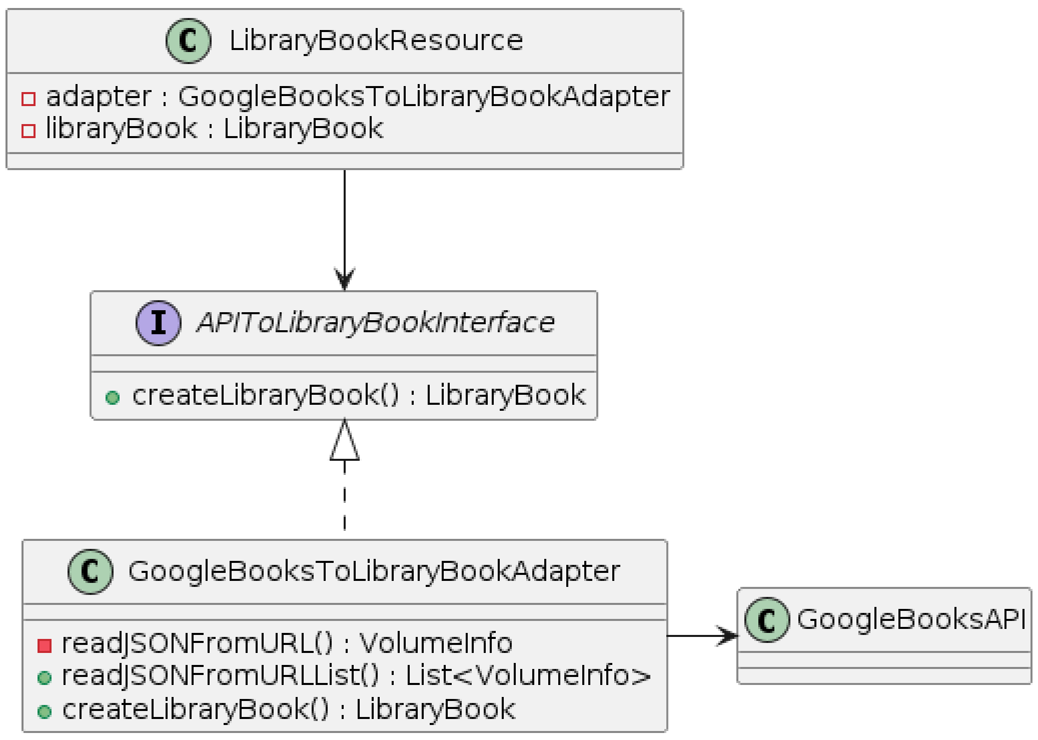
Participants

1. Context — `RankingContext` and `FilteringContext` — Maintains a reference to one of the concrete ranking strategies and allows switching between different strategies.
2. Strategy Interface — `RankingStrategy` and `FilteringStrategy` — It is the interface common to all the different ranking strategies. It declares a method the context uses to execute a strategy.
3. Concrete Strategies — `AverageRankingStrategy, NumberRankingStrategy` and `DefaultFilteringStrategy, AuthorFilteringStrategy, GenreFilteringStrategy, RatingFilteringStrategy` — Implement variations of the algorithms used for achieving the desired task. Are called by the respective context.
4. Client — `LibraryCommand` implementing classes, which in turn are called by `LibraryBookResource` — The Client creates specific strategy objects and puts them in the Context based on the user's choice. The context exposes a setter which lets clients replace the strategy.

Motivation behind various design decisions

- **Why Strategy:** The strategy pattern allows us to encapsulate different algorithms (for ranking and filtering) into separate classes. This makes it easy to switch between strategies dynamically without modifying the client code. Having all the different strategies in one class would lead to a **long method** and **violation of the Single Responsible Principle (SRP)**.
    - By encapsulating each strategy into its own class, it achieves a separation of concerns. Each strategy focuses on a specific way of importing media, making the code to easier to understand and maintain.
    - The pattern also enables us to add new import strategies in the future without disrupting the existing codebase. We can simply create new strategies by implementing the `RankingStrategy` or `FilteringStrategy` classes. This ensures the scalability of the system.

3. Adapter Pattern


LibraryBookResource
adapter : GoogleBooksToLibraryBookAdapter
libraryBook : LibraryBook
APIToLibraryBookInterface
createLibraryBook() : LibraryBook
GoogleBooksToLibraryBookAdapter
readJSONFromURL() : VolumeInfo
readJSONFromURLList() : List<VolumeInfo>
createLibraryBook() : LibraryBook
GoogleBooksAPI

Participants

1. Service — *Google Books API* — provides the data in a way that the Client is not expecting and cannot directly be accepted.
2. Client Interface — `APIToLibraryBookInterface` — describes a protocol that other classes must follow to be able to collaborate with the client code.

3. Adapter — `GoogleBooksToLibraryBookInterface` — implements the client interface, while wrapping the service object. Takes the Service and transforms it into a format that the Client can understand.
4. Client — `LibraryBookResource` — contains the existing business logic. It calls the adapter, and uses the received data for its operations.

Motivation behind this design pattern

- **Why Adapter:** The API returns a JSON object that cannot be stored directly, it needed to be stored in a nested object that mirrored the way the JSON dict was structured. This required the creation of a lot of classes and functions that are not relevant to any other functionality in `LibraryBookResource` , our Client class. It was expecting a `LibraryBook` so that it could create the entry in the database using the DAO, which was incompatible with the interface provided by Google Books. Hence, we went with this design pattern
- **Why many classes and functions within the Adapter that are not declared in the interface:** Our idea is to have each Adapter class be a self-sufficient entity that takes in a `Book` and returns a `LibraryBook`. In this case, the classes such as `VolumeInfo` are relevant only for the Google Books API i.e. they would be useless for e.g. the WorldCat API. So now, we can define many different adapters for each API that we might want to use, and the final product is that the `createLibraryBook` function does not have to worry about the internals. This achieves modularization.

Implementation Details

Preliminary steps

1. Create a LibraryBook class and a LibraryBookRating class.
2. Modify `db-update-000-0.sql` to create tables for the same. Introduce foreign keys where necessary.
3. Connect them to SQL by using @ tags — @Entity, @Table, @Column — in the Java code.
4. Add the Java filepath to `persistence.xml` .

Steps for implementing a functionality

1. Create a `DAO` class (or use an existing one) to interact with the SQL database.
2. Create a `Resource` class. This will furnish endpoints for the RESTful APIs. Use @Path, @GET, @POST, @QueryParam, etc. It uses the DAO to perform actions on the database.
3. Add an HTML page.
4. Add a controller for the same in Angular. Make necessary additions to files such as `app.js` and `index.html`.
5. Write functions using the endpoints provided by the `Resource` file.

LibraryBook

This class contains all the essential details for a book that has to be displayed in the library. It had to be made `Serializable` in order for it to be stored in the database.

```
public class LibraryBook implements Serializable {
    @Id
    @Column(name="L_BOOK_ID", length = Constants.DEFAULT_ID_LEN)
    private String id;

    @Column(name="L_BOOK_T", length = Constants.MAX_TITLE_LEN)
    private String title;

    @Column(name = "L_BOOK_AUTH", length = Constants.MAX_STRING_LEN)
    private String authors;

    @Column(name = "L_BOOK_G", length = Constants.MAX_STRING_LEN)
    private String genres;

    @Column(name = "L_BOOK_R")
    private float rating;
    @Column(name = "L_BOOK_NR")
    private int numRatings;

    @Column(name = "L_BOOK_THUMB", length = Constants.MAX_URL_LEN)
    private String thumbURL;
```

LibraryBookResource

```
    @GET
    @Path("/list")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getLibrary(
        @QueryParam("input") String input,
        @QueryParam("type") String type
    ) throws Exception {
        if (!authenticate()) {
            return Response.status(Response.Status.UNAUTHORIZED).build();
        }

        JSONObject response = new JSONObject();
        String commandType = ((Arrays.asList("default", "rating", "genre", "author").indexOf
(type) != -1) ? "filter" : "rank");

        List<JSONObject> books = new ArrayList<>();
        LibraryInvoker libraryInvoker = new LibraryInvoker();
        switch (commandType) {
            case "filter":
                libraryInvoker.setLibraryCommand(new FilteringLibraryCommand());
                break;
            case "rank":
                libraryInvoker.setLibraryCommand(new RankingLibraryCommand());
                break;
            default:
                break;
        }

        books = libraryInvoker.getLibraryCommand().getLibraryBooks(type, input, principal.get
Id());
        response.put("books", books);

        return Response.ok().entity(response).build();
    }
```

The Command Design Pattern

This is a concrete command that implements `LibraryCommand` interface for ranking.

```
public class RankingLibraryCommand implements LibraryCommand {

    @Override
    public List<JSONObject> getLibraryBooks(String type, String arg, String userId) throws Ex
ception {
        RankingContext rankingContext = new RankingContext();
        switch (type) {
            case "average":
                rankingContext.setRankingStrategy(new AverageRankingStrategy());
                break;
            case "number":
                rankingContext.setRankingStrategy(new NumberRankingStrategy());
                break;
            default:
                rankingContext.setRankingStrategy(new AverageRankingStrategy());
                break;
        }
        return rankingContext.getRankedBooks(arg, userId);
    }
}
```

The Strategy Design Pattern

This class is a concrete Strategy of the `RankingStrategy` pattern

```
public class AverageRankingStrategy implements RankingStrategy {

    @Override
    public List<JSONObject> getRankedBooks(String arg, String userId) throws JSONException {
        System.out.println("AverageRankingStrategy.getRankingBooks");
        String customQuery = "select l from LibraryBook l order by l.rating desc";
        LibraryBookDao libraryBookDao = new LibraryBookDao();
        List<LibraryBook> libraryBooks = libraryBookDao.findByCriteria(customQuery, "");

        List<JSONObject> books = new ArrayList<>();

        for (LibraryBook libraryBook : libraryBooks) {
            JSONObject book = new JSONObject();
            book.put("id", libraryBook.getId());
            book.put("title", libraryBook.getTitle());
            book.put("authors", libraryBook.getAuthors());
            book.put("genres", libraryBook.getGenres());
            book.put("rating", libraryBook.getRating());
            book.put("thumbURL", libraryBook.getThumbURL());
            book.put("numRatings", libraryBook.getNumRatings());
            book.put("userRating", libraryBookDao.findUserRating(libraryBook.getId(), userI
d));
            books.add(book);
        }

        return books;
    }
}
```

This is the context for it

```
public class RankingContext {
    private RankingStrategy rankingStrategy;

    public RankingContext () {
    }

    public void setRankingStrategy(RankingStrategy rankingStrategy) {
        this.rankingStrategy = rankingStrategy;
    }

    public RankingStrategy getRankingStrategy() {
        return rankingStrategy;
    }

    public List<JSONObject> getRankedBooks(String arg, String userId) throws Exception {
        return rankingStrategy.getRankedBooks(arg, userId);
    }
}
```

The Adapter Design Pattern

The

```
public class GoogleBooksToLibraryBookAdapter implements APIToLibraryBookInterface {
    private class urlDto {
        int totalItems = 0;
        List<Item> items;
    }

    private class Item {
        VolumeInfo volumeInfo;
    }

    private class VolumeInfo {
        // String title;
        List<String> authors;
        List<String> categories;
        ImageLinks imageLinks;
    }

    private class ImageLinks {
        // String smallThumbnail;
        String thumbnail;
    }

    private static String baseURL = "https://www.googleapis.com/books/v1/volumes?q=isbn:";

    public GoogleBooksToLibraryBookAdapter() {
    }

    private VolumeInfo readJSONFromURL(String url) throws Exception {
        urlDto dto;
        do {
            URL url_object = new URL(url);
            InputStreamReader reader = new InputStreamReader(url_object.openStream());
            dto = new Gson().fromJson(reader, urlDto.class);
        } while (dto.totalItems == 0);

        VolumeInfo volumeInfo = new VolumeInfo();
        volumeInfo = dto.items.get(0).volumeInfo;
        return volumeInfo;
    }

    public List<VolumeInfo> readJSONFromURLList(String url) throws Exception {
        urlDto dto;
        do {
            URL url_object = new URL(url);
            InputStreamReader reader = new InputStreamReader(url_object.openStream());
            dto = new Gson().fromJson(reader, urlDto.class);
        } while (dto.totalItems == 0);

        List<VolumeInfo> volumeInfoList = new ArrayList<>();
        for (Item item : dto.items) {
            volumeInfoList.add(item.volumeInfo);
        }
        return volumeInfoList;
    }

        @Override
    public LibraryBook createLibraryBook(Book book) {
        LibraryBook libraryBook;
        libraryBook = new LibraryBook();
        libraryBook.setId(book.getId());
        libraryBook.setTitle(book.getTitle());
        libraryBook.setRating(0);
        libraryBook.setNumRatings(0);

        String url = baseURL + book.getIsbn13();
        VolumeInfo volumeInfo;
        try {
            volumeInfo = readJSONFromURL(url);
        } catch (Exception e) {
            volumeInfo = new VolumeInfo();
        }

        if (volumeInfo.authors == null) {
            volumeInfo.authors = new ArrayList<>();
            volumeInfo.authors.add(book.getAuthor());
        }

        if (volumeInfo.categories == null) {
            volumeInfo.categories = new ArrayList<>();
            volumeInfo.categories.add("N/A");
        }

        libraryBook.setAuthors(StringUtil.ListToString(volumeInfo.authors, ","));
        libraryBook.setGenres(StringUtil.ListToString(volumeInfo.categories, ","));
        libraryBook.setThumbURL(volumeInfo.imageLinks.thumbnail);

        return libraryBook;
    }
}
```

This contains a lot of functions and classes that are not defined in the interface. Our idea is to have each Adapter class be a self-sufficient entity that takes in a `Book` and returns a `LibraryBook`. In this case, the classes such as `VolumeInfo` are relevant only for the Google Books API i.e. they would be useless for e.g. the WorldCat API.

This is the SQL code for making the tables

```
CREATE CACHED TABLE L_BOOK (L_BOOK_ID VARCHAR(36) PRIMARY KEY not null, L_BOOK_T VARCHAR(256)
not null, L_BOOK_AUTH VARCHAR(2048) not null, L_BOOK_G VARCHAR(2048) not null, L_BOOK_R FLOAT
not null, L_BOOK_NR INT not null, L_BOOK_THUMB VARCHAR(2048) not null);
create cached table L_BOOK_RATING (L_BOOK_R_ID VARCHAR(36) PRIMARY KEY not null, L_BOOK_ID VA
RCHAR(36) not null, L_USER_ID VARCHAR(36) not null, L_BOOK_R FLOAT not null);

alter table L_BOOK add constraint FK_L_BOOK_ID foreign key (L_BOOK_ID) references T_BOOK (BOK
_ID_C) on delete restrict on update restrict;
```

```
alter table L_BOOK_RATING add constraint FK_LR_BOOK_ID foreign key (L_BOOK_ID) references T_B
OOK (BOK_ID_C) on delete restrict on update restrict;
alter table L_BOOK_RATING add constraint FK_LR_USER_ID foreign key (L_USER_ID) references T_U
SER (USE_ID_C) on delete restrict on update restrict;
```

This is the entry in the `app.js` file

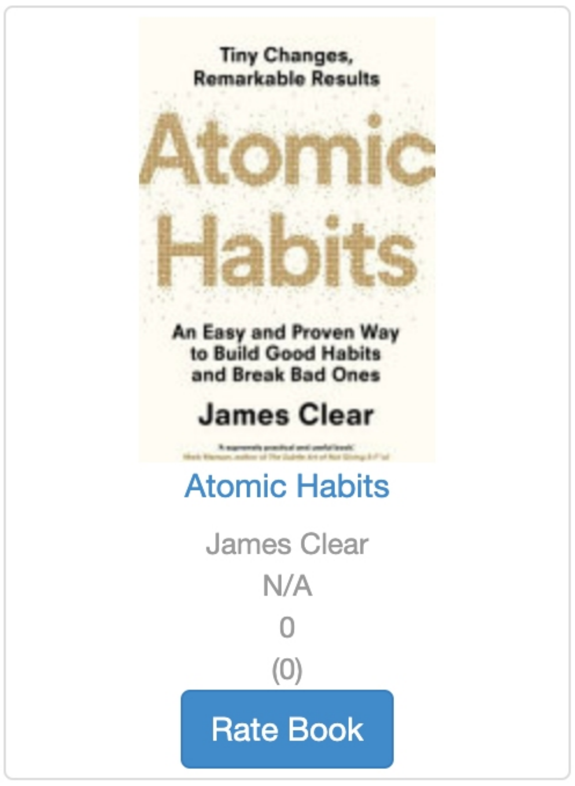
```
.state('library', {
    url: '/library',
    views: {
      'page': {
        templateUrl: 'partial/library.html',
        controller: 'Library'
      }
    }
  })
```

Features of the System

1. Book Details

Each Book in the library has the following information:

- Title
- Author(s)
- Genre(s) (supporting multiple genres per book)
- Rating (numerical value, calculated and averaged from user ratings)
- Thumbnail Image URL
- Number of ratings

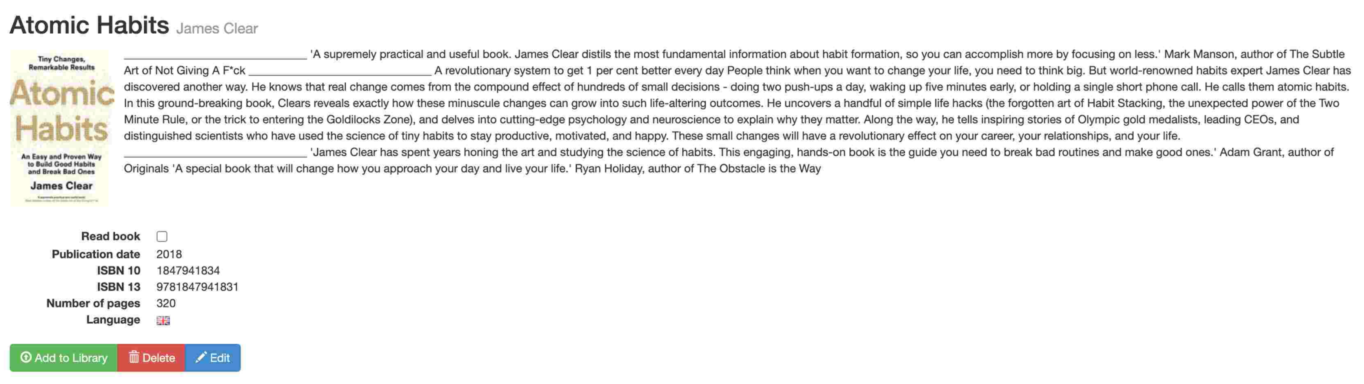

Tiny Changes, Remarkable Results
Atomic Habits
An Easy and Proven Way to Build Good Habits and Break Bad Ones
James Clear
Rate Book

Atomic Habits
James Clear
N/A
0
(0)

2. User Management

Users can register and interact with the system, including:

- Adding books to the library with the required information

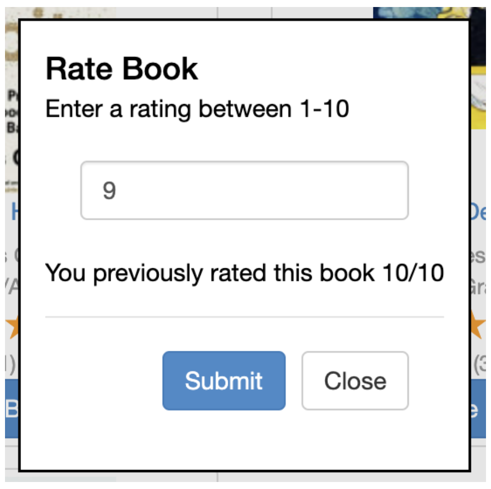


Atomic Habits James Clear

___________________________ 'A supremely practical and useful book. James Clear distils the most fundamental information about habit formation, so you can accomplish more by focusing on less.' Mark Manson, author of The Subtle Art of Not Giving A F*ck ___________________________ A revolutionary system to get 1 per cent better every day People think when you want to change your life, you need to think big. But world-renowned habits expert James Clear has discovered another way. He knows that real change comes from the compound effect of hundreds of small decisions - doing two push-ups a day, waking up five minutes early, or holding a single short phone call. He calls them atomic habits. In this ground-breaking book, Clears reveals exactly how these minuscule changes can grow into such life-altering outcomes. He uncovers a handful of simple life hacks (the forgotten art of Habit Stacking, the unexpected power of the Two Minute Rule, or the trick to entering the Goldilocks Zone), and delves into cutting-edge psychology and neuroscience to explain why they matter. Along the way, he tells inspiring stories of Olympic gold medalists, leading CEOs, and distinguished scientists who have used the science of tiny habits to stay productive, motivated, and happy. These small changes will have a revolutionary effect on your career, your relationships, and your life. ___________________________ 'James Clear has spent years honing the art and studying the science of habits. This engaging, hands-on book is the guide you need to break bad routines and make good ones.' Adam Grant, author of Originals 'A special book that will change how you approach your day and live your life.' Ryan Holiday, author of The Obstacle is the Way

Read book	☐
Publication date	2018
ISBN 10	1847941834
ISBN 13	9781847941831
Number of pages	320
Language	🇬🇧

- Rating existing books on a defined scale (1-10) (also shows previous rating of user)



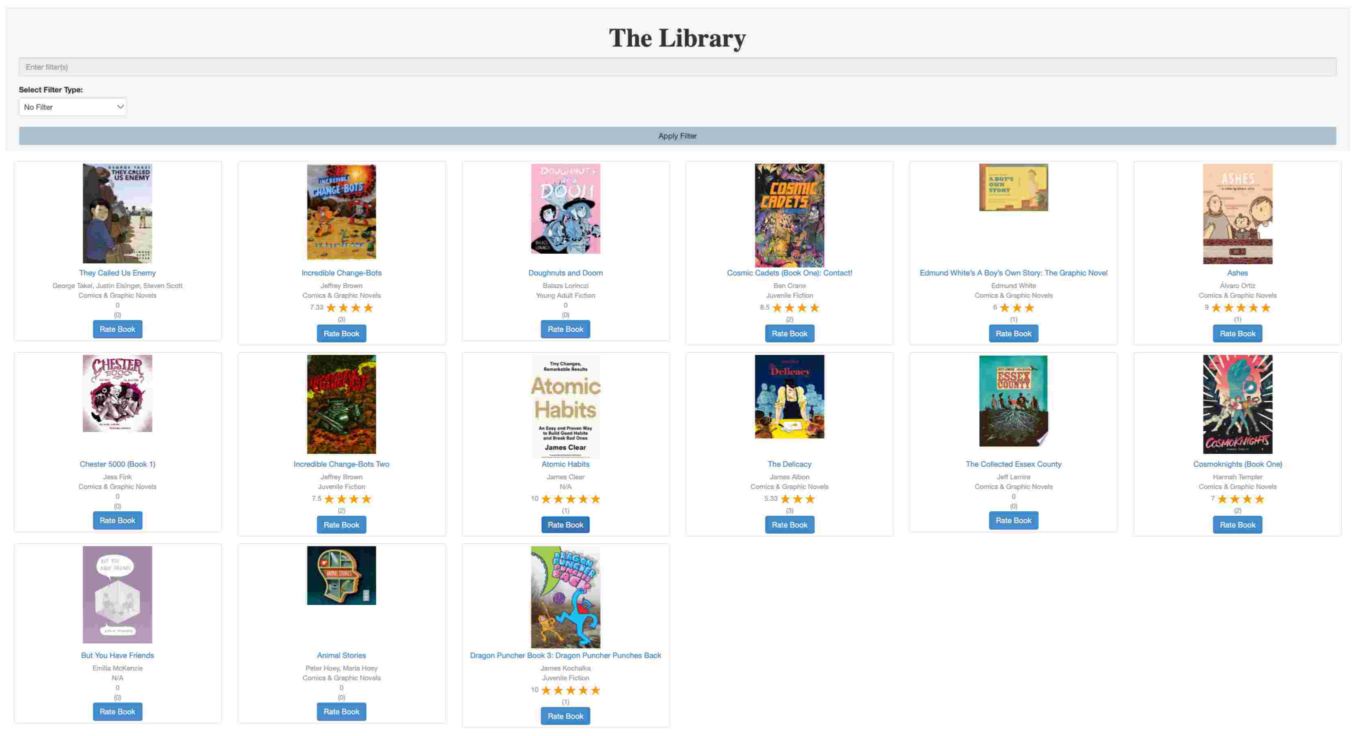
Rate Book

Enter a rating between 1-10

9

You previously rated this book 10/10

- Viewing all the books in the library



The Library

Enter filter(s)

Select Filter Type: No Filter

They Called Us Enemy
George Takei, Justin Eisinger, Steven Scott
Comics & Graphic Novels
0
(0)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33 ★★★★
(3)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

Cosmic Cadets (Book One): Contact!
Ben Crane
Juvenile Fiction
8.5 ★★★★
(2)

Edmund White's A Boy's Own Story: The Graphic Novel
Edmund White
Comics & Graphic Novels
6 ★★★
(1)

Ashes
Álvaro Ortiz
Comics & Graphic Novels
9 ★★★★★
(1)

Chester 5000 (Book 1)
Jess Fink
Comics & Graphic Novels
0
(0)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5 ★★★★
(2)

Atomic Habits
James Clear
N/A
10 ★★★★★
(1)

The Delicacy
James Albon
Comics & Graphic Novels
5.33 ★★★
(3)

The Collected Essex County
Jeff Lemire
Comics & Graphic Novels
0
(0)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7 ★★★★
(2)

But You Have Friends
Emilia McKenzie
N/A
0
(0)

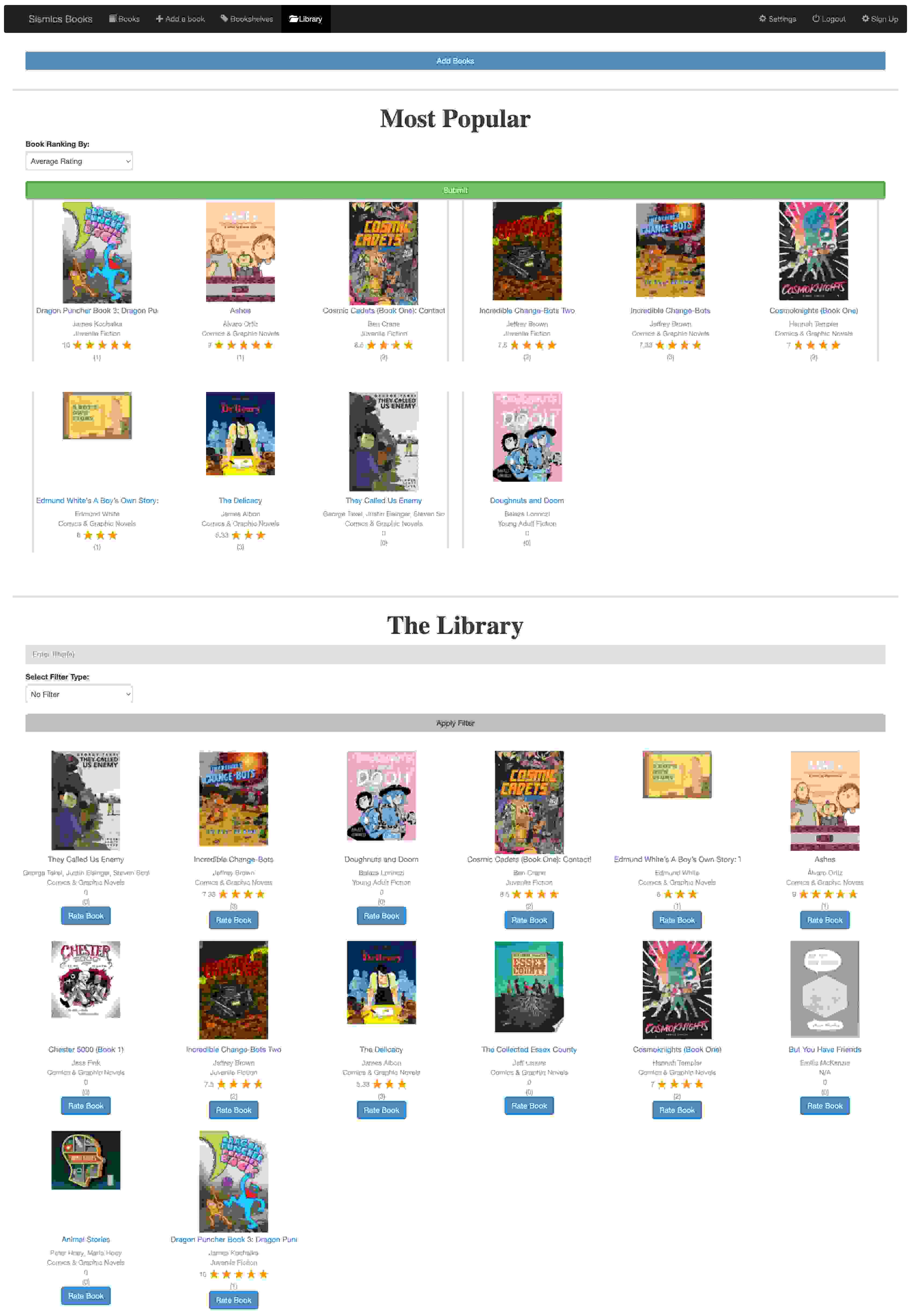
Animal Stories
Peter Hoey, Maria Hoey
Comics & Graphic Novels
0
(0)

Dragon Puncher Book 3: Dragon Puncher Punches Back
James Kochalka
Juvenile Fiction
10 ★★★★★
(1)

3. Book Ranking

A dynamic list displays the top 10 books based on the following two criterias (will be selected by the user).

- Average Rating



Most Popular

Book Ranking By:
Average Rating

Dragon Puncher Book 3: Dragon Pu
James Kochalka
Juvenile Fiction
10
(1)

Ashes
Alvaro Ortiz
Comics & Graphic Novels
9
(1)

Cosmic Cadets (Book One): Contact
Ben Crane
Juvenile Fiction
8.5
(2)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33
(3)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7
(2)

Edmund White's A Boy's Own Story:
Edmund White
Comics & Graphic Novels
6
(1)

The Delicacy
James Albon
Comics & Graphic Novels
5.33
(3)

They Called Us Enemy
George Takei, Justin Eisinger, Steven Sc
Comics & Graphic Novels
0
(0)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

The Library

Enter filter(s)

Select Filter Type:
No Filter

They Called Us Enemy
George Takei, Justin Eisinger, Steven Scot
Comics & Graphic Novels
0
(0)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33
(3)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

Cosmic Cadets (Book One): Contact!
Ben Crane
Juvenile Fiction
8.5
(2)

Edmund White's A Boy's Own Story: 1
Edmund White
Comics & Graphic Novels
6
(1)

Ashes
Álvaro Ortiz
Comics & Graphic Novels
9
(1)

Chester 5000 (Book 1)
Jess Fink
Comics & Graphic Novels
0
(0)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)

The Delicacy
James Albon
Comics & Graphic Novels
5.33
(3)

The Collected Essex County
Jeff Lemire
Comics & Graphic Novels
0
(0)

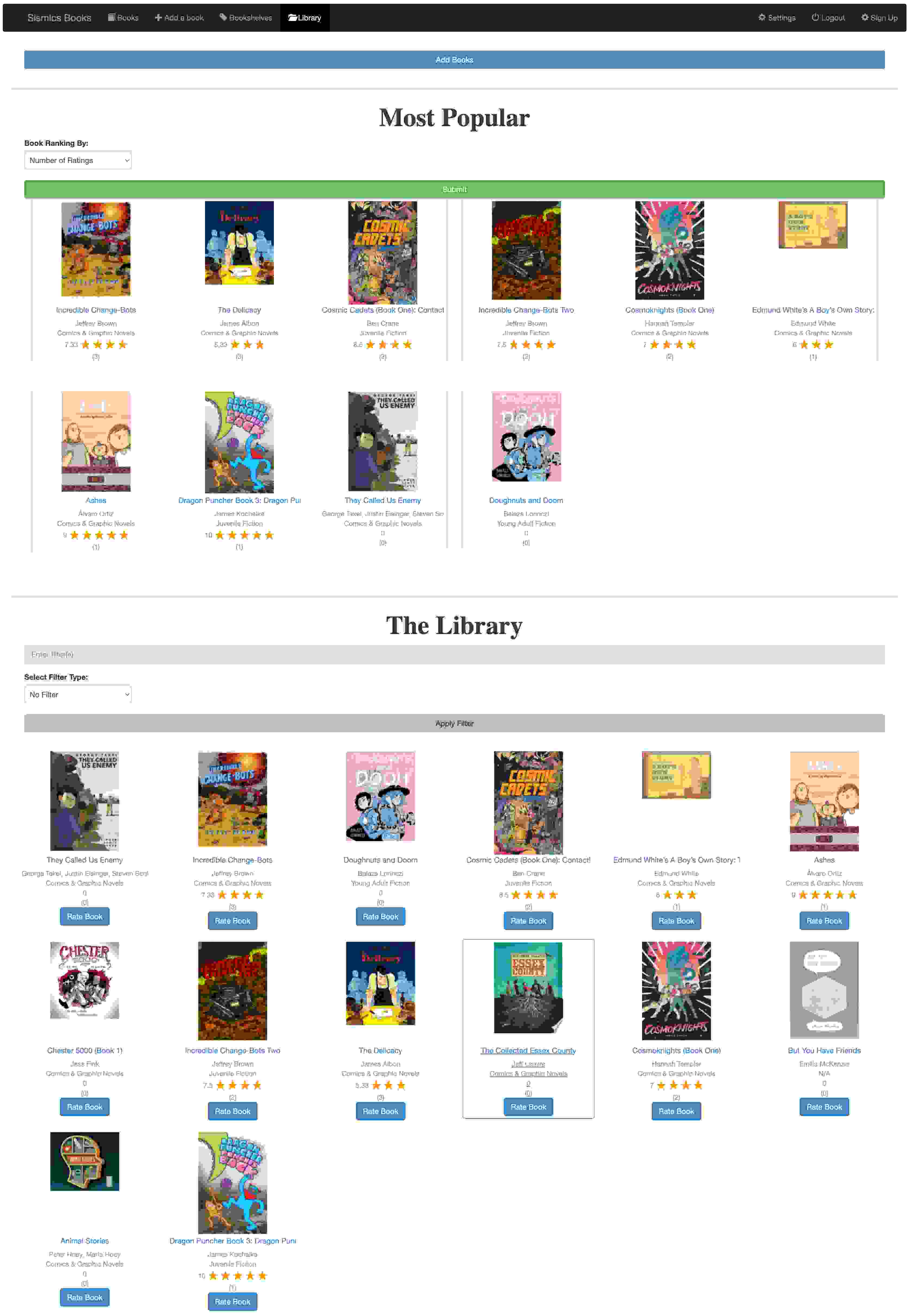
Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7
(2)

But You Have Friends
Emilia McKenzie
N/A
0
(0)

Animal Stories
Peter Hoey, Maria Hoey
Comics & Graphic Novels
0
(0)

Dragon Puncher Book 3: Dragon Punc
James Kochalka
Juvenile Fiction
10
(1)

- Number of Ratings



Most Popular

Book Ranking By:

Number of Ratings

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33
(3)

The Delicacy
James Albon
Comics & Graphic Novels
5.33
(3)

Cosmic Cadets (Book One): Contact
Ben Crane
Juvenile Fiction
8.5
(2)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7
(2)

Edmund White's A Boy's Own Story:
Edmund White
Comics & Graphic Novels
6
(1)

Ashes
Álvaro Ortiz
Comics & Graphic Novels
9
(1)

Dragon Puncher Book 3: Dragon Pu
James Kochalka
Juvenile Fiction
10
(1)

They Called Us Enemy
George Takei, Justin Eisinger, Steven Sc
Comics & Graphic Novels
0
(0)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

The Library

Enter filter(s)

Select Filter Type:

No Filter

They Called Us Enemy
George Takei, Justin Eisinger, Steven Scott
Comics & Graphic Novels
0
(0)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33
(3)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

Cosmic Cadets (Book One): Contact!
Ben Crane
Juvenile Fiction
8.5
(2)

Edmund White's A Boy's Own Story: 1
Edmund White
Comics & Graphic Novels
6
(1)

Ashes
Álvaro Ortiz
Comics & Graphic Novels
9
(1)

Chester 5000 (Book 1)
Jess Fink
Comics & Graphic Novels
0
(0)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)

The Delicacy
James Albon
Comics & Graphic Novels
5.33
(3)

The Collected Essex County
Jeff Lemire
Comics & Graphic Novels
0
(0)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7
(2)

But You Have Friends
Emilia McKenzie
N/A
0
(0)

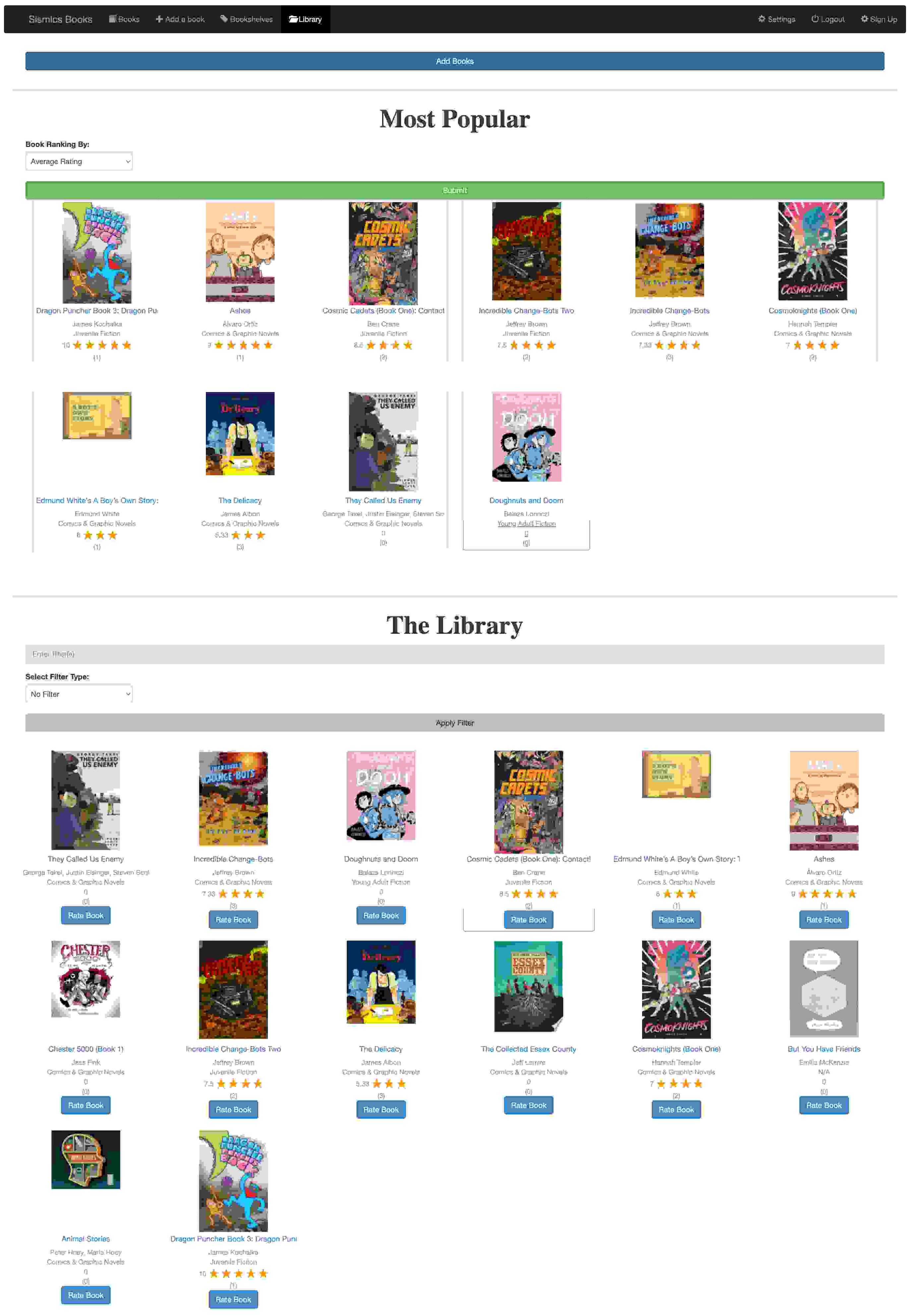
Animal Stories
Peter Hoey, Maria Hoey
Comics & Graphic Novels
0
(0)

Dragon Puncher Book 3: Dragon Pun
James Kochalka
Juvenile Fiction
10
(1)

4. Filtering

Users can filter displayed books by:

- No Filter



Most Popular

Book Ranking By: Average Rating

Dragon Puncher Book 3: Dragon Pu
James Kochalka
Juvenile Fiction
10 ★★★★★
(1)

Ashes
Alvaro Ortiz
Comics & Graphic Novels
9 ★★★★★
(1)

Cosmic Cadets (Book One): Contact
Ben Crane
Juvenile Fiction
8.5 ★★★★
(2)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5 ★★★★
(2)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33 ★★★★
(3)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7 ★★★★
(2)

Edmund White's A Boy's Own Story:
Edmund White
Comics & Graphic Novels
6 ★★★
(1)

The Delicacy
James Albon
Comics & Graphic Novels
5.33 ★★★
(3)

They Called Us Enemy
George Takei, Justin Eisinger, Steven Sc
Comics & Graphic Novels
0
(0)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

The Library

Enter filter(s)

Select Filter Type: No Filter

They Called Us Enemy
George Takei, Justin Eisinger, Steven Scott
Comics & Graphic Novels
0
(0)
Rate Book

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33 ★★★★
(3)
Rate Book

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)
Rate Book

Cosmic Cadets (Book One): Contact!
Ben Crane
Juvenile Fiction
8.5 ★★★★
(2)
Rate Book

Edmund White's A Boy's Own Story: 1
Edmund White
Comics & Graphic Novels
6 ★★★
(1)
Rate Book

Ashes
Álvaro Ortiz
Comics & Graphic Novels
9 ★★★★★
(1)
Rate Book

Chester 5000 (Book 1)
Jess Fink
Comics & Graphic Novels
0
(0)
Rate Book

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5 ★★★★
(2)
Rate Book

The Delicacy
James Albon
Comics & Graphic Novels
5.33 ★★★
(3)
Rate Book

The Collected Essex County
Jeff Lemire
Comics & Graphic Novels
0
(0)
Rate Book

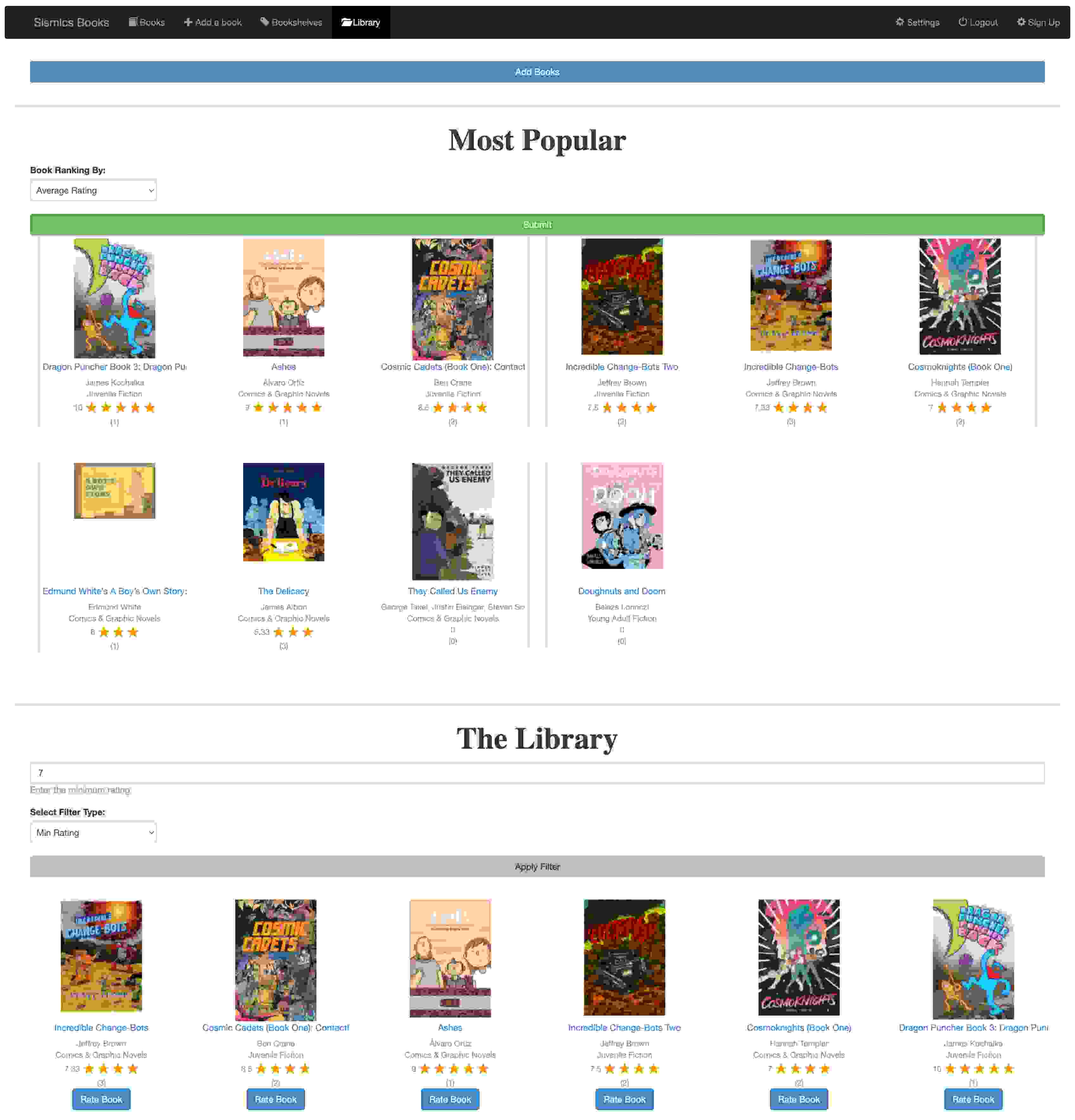
Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7 ★★★★
(2)
Rate Book

But You Have Friends
Emilia McKenzie
N/A
0
(0)
Rate Book

Animal Stories
Peter Hoey, Maria Hoey
Comics & Graphic Novels
0
(0)
Rate Book

Dragon Puncher Book 3: Dragon Puni
James Kochalka
Juvenile Fiction
10 ★★★★★
(1)
Rate Book

- Minimum Ratings



Most Popular

Book Ranking By: Average Rating

Dragon Puncher Book 3: Dragon Pu
James Kochalka
Juvenile Fiction
10
(1)

Ashes
Alvaro Ortiz
Comics & Graphic Novels
9
(1)

Cosmic Cadets (Book One): Contact
Ben Crane
Juvenile Fiction
8.5
(2)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33
(3)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7
(2)

Edmund White's A Boy's Own Story:
Edmund White
Comics & Graphic Novels
6
(1)

The Delicacy
James Albon
Comics & Graphic Novels
5.33
(3)

They Called Us Enemy
George Takei, Justin Eisinger, Steven Sc
Comics & Graphic Novels
0
(0)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

The Library

7
Enter the minimum rating

Select Filter Type: Min Rating

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.83
(3)

Cosmic Cadets (Book One): Contact!
Ben Crane
Juvenile Fiction
8.5
(2)

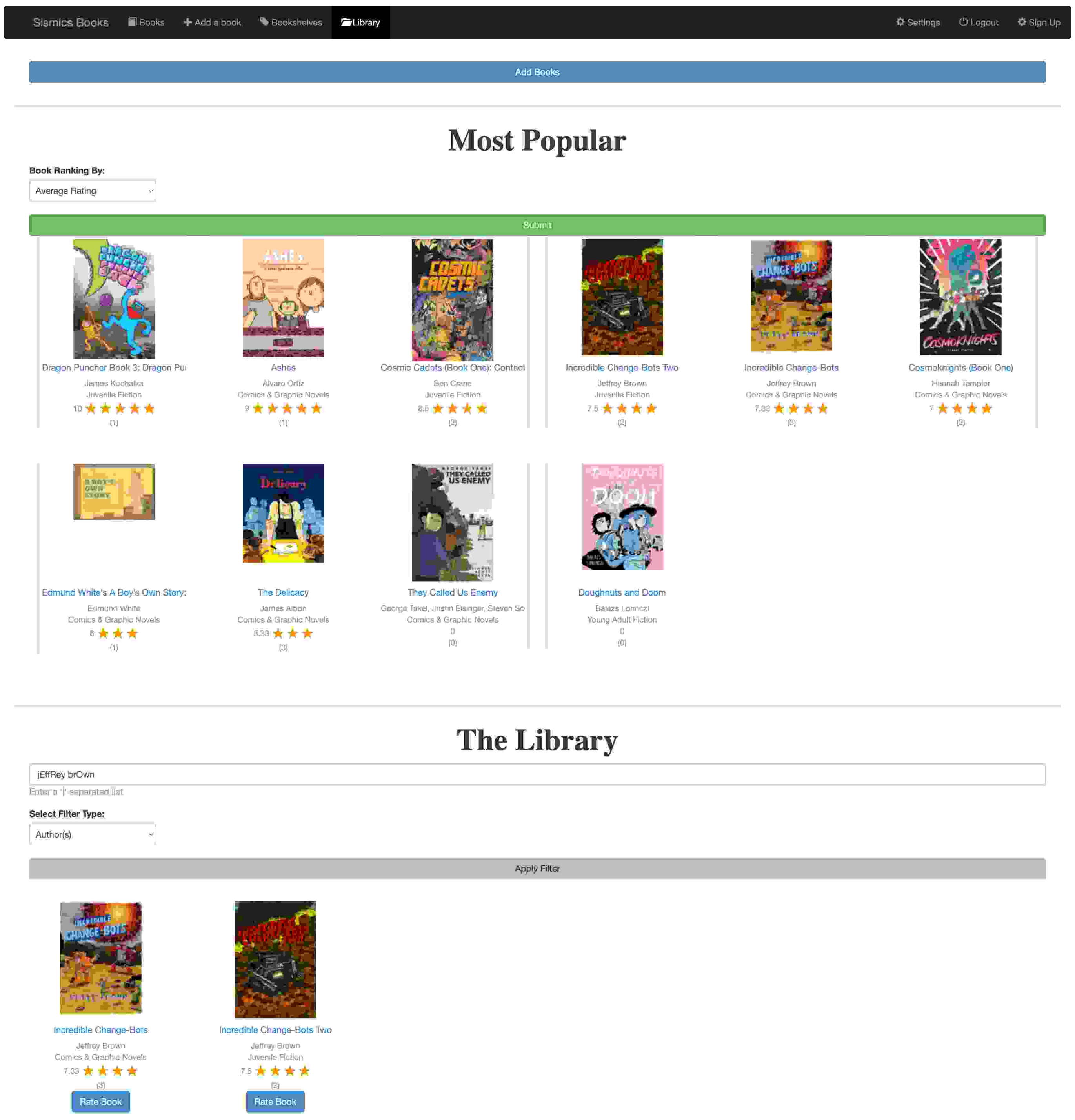
Ashes
Álvaro Ortiz
Comics & Graphic Novels
9
(1)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7
(2)

Dragon Puncher Book 3: Dragon Pun
James Kochalka
Juvenile Fiction
10
(1)

- Author(s)



Most Popular

Book Ranking By: Average Rating

Submit

Dragon Puncher Book 3: Dragon Pu
James Kochalka
Juvenile Fiction
10
(1)

Ashes
Alvaro Ortiz
Comics & Graphic Novels
9
(1)

Cosmic Cadets (Book One): Contact
Ben Crane
Juvenile Fiction
8.5
(2)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33
(3)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7
(2)

Edmund White's A Boy's Own Story:
Edmund White
Comics & Graphic Novels
6
(1)

The Delicacy
James Albon
Comics & Graphic Novels
5.33
(3)

They Called Us Enemy
George Takei, Justin Eisinger, Steven Sc
Comics & Graphic Novels
0
(0)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

The Library

jEffRey brOwn
Enter a '|' separated list

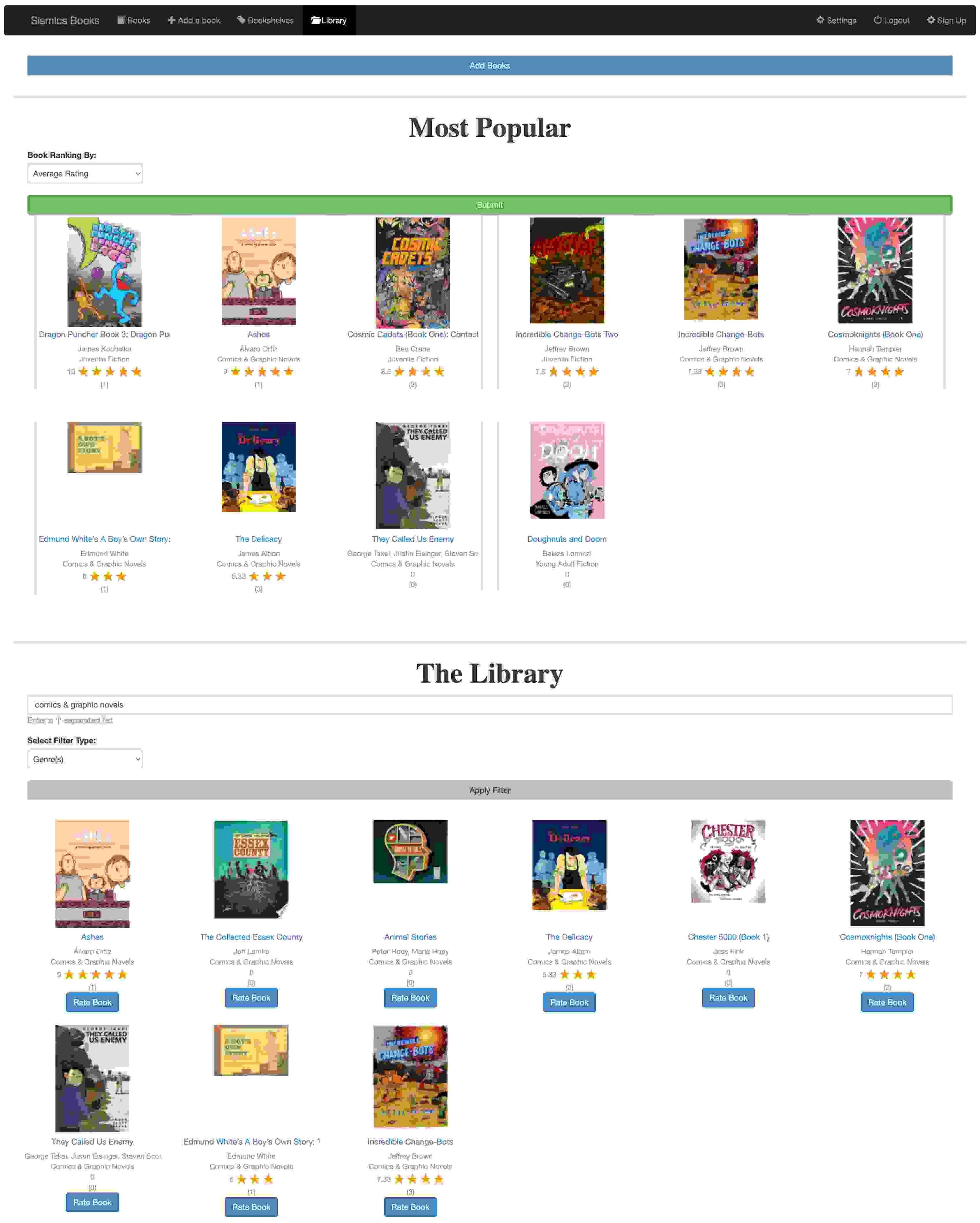
Select Filter Type: Author(s)

Apply Filter

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33
(3)
Rate Book

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5
(2)
Rate Book

- Genre(s)



Most Popular

Book Ranking By:
Average Rating

Dragon Puncher Book 3: Dragon Pu
James Kochalka
Juvenile Fiction
10 ★★★★★
(1)

Ashes
Alvaro Ortiz
Comics & Graphic Novels
9 ★★★★★
(1)

Cosmic Cadets (Book One): Contact
Ben Crane
Juvenile Fiction
8.5 ★★★★
(2)

Incredible Change-Bots Two
Jeffrey Brown
Juvenile Fiction
7.5 ★★★★
(2)

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33 ★★★★
(3)

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7 ★★★★
(2)

Edmund White's A Boy's Own Story:
Edmund White
Comics & Graphic Novels
6 ★★★
(1)

The Delicacy
James Albon
Comics & Graphic Novels
5.33 ★★★
(3)

They Called Us Enemy
George Takei, Justin Eisinger, Steven Sc
Comics & Graphic Novels
0
(0)

Doughnuts and Doom
Balazs Lorinczi
Young Adult Fiction
0
(0)

The Library

comics & graphic novels
Enter a '|'-separated list

Select Filter Type:
Genre(s)

Ashes
Álvaro Ortiz
Comics & Graphic Novels
9 ★★★★★
(1)
Rate Book

The Collected Essex County
Jeff Lemire
Comics & Graphic Novels
0
(0)
Rate Book

Animal Stories
Peter Hoey, Maria Hoey
Comics & Graphic Novels
0
(0)
Rate Book

The Delicacy
James Albon
Comics & Graphic Novels
5.33 ★★★
(3)
Rate Book

Chester 5000 (Book 1)
Jess Fink
Comics & Graphic Novels
0
(0)
Rate Book

Cosmoknights (Book One)
Hannah Templer
Comics & Graphic Novels
7 ★★★★
(2)
Rate Book

They Called Us Enemy
George Takei, Justin Eisinger, Steven Scot
Comics & Graphic Novels
0
(0)
Rate Book

Edmund White's A Boy's Own Story: 1
Edmund White
Comics & Graphic Novels
6 ★★★
(1)
Rate Book

Incredible Change-Bots
Jeffrey Brown
Comics & Graphic Novels
7.33 ★★★★
(3)
Rate Book

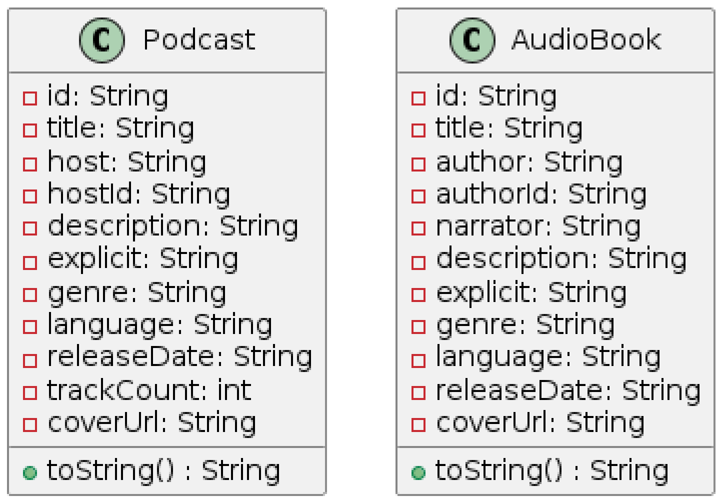
5. Additional Features Implemented

- `Stars` to shows book rating out of 5 stars.
- Show what the user `previously rated` a particular book in the library.

Online Integration

Features: Selecting Providers, Content type, Search Functionality, Display functionality, Favorites functionality

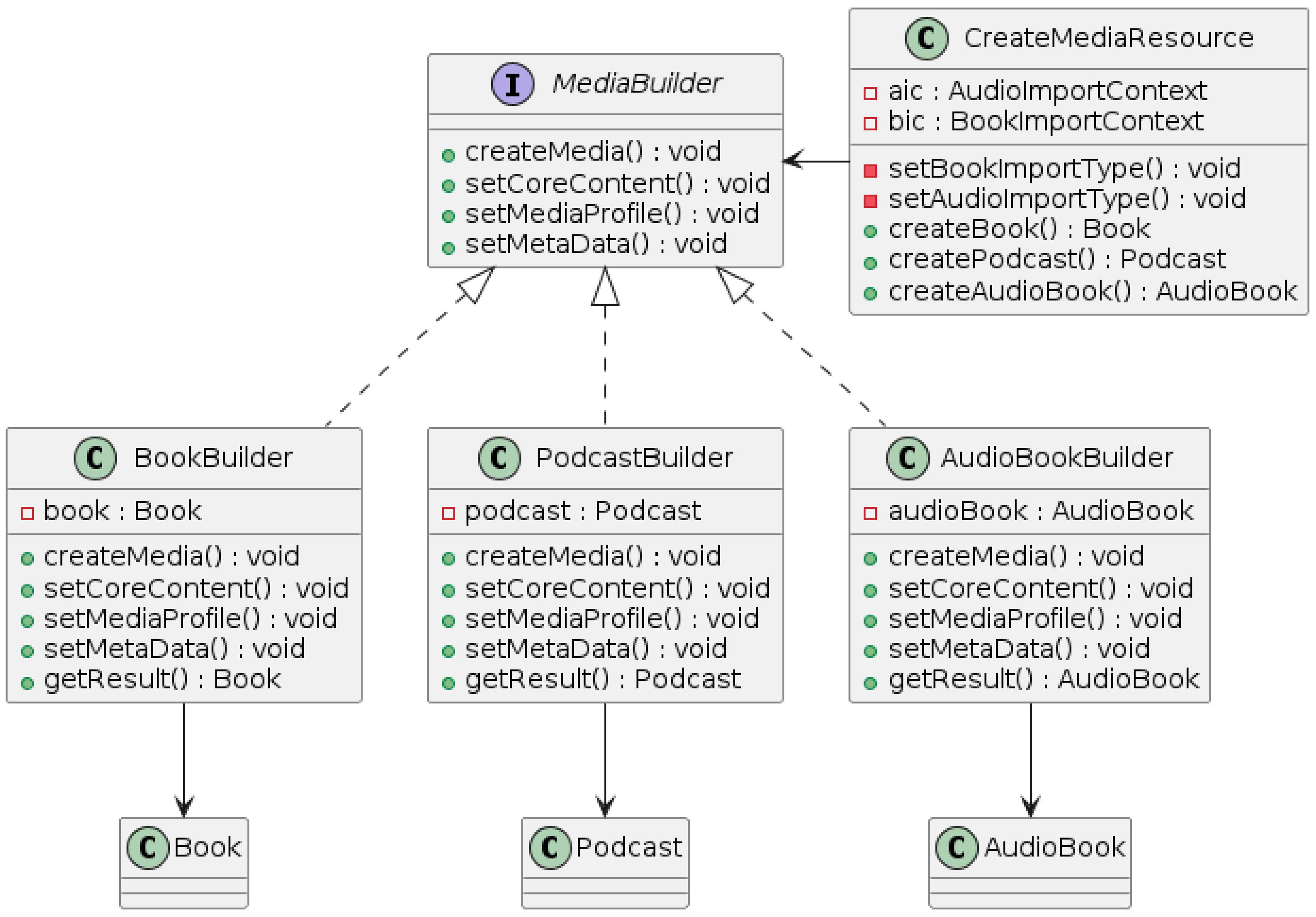
Core Classes



- These classes correspond to the entities for which all other functionalities mentioned below revolves around.
- These two classes collectively are called audio, and they are individually audio types
- The fields present in both these classes were decided on based on the data being returned by the various providers - Spotify and iTunes.
- SQL tables for these classes were created, and sql field decorators were also added.
- The toString function proved to be useful when we were passing data from the backend to the frontend.

Design Patterns Used :

1. Builder Pattern



Participants

1. Builder - `MediaBuilder` : Defines the interface for product construction steps that are common to all types of builders. The functions here correspond to 4 logical partitions of data in the products
    a. `createMedia` - creates/instantiates the product (audiobook, podcast etc)
    b. `setCoreContent` - responsible for setting absolutely necessary content in the product, essentially the content without which the product would make no sense/would be useless.
        i. title, author, host etc

c. `setMediaProfile` - responsible for setting content which describes the product a little more, basically fields/data that a person would want/need to see to decide whether to favorite the product or not. For example, before listening to the Joe Rogan podcast, you would like to know what its about (description), explicit, and its genre.

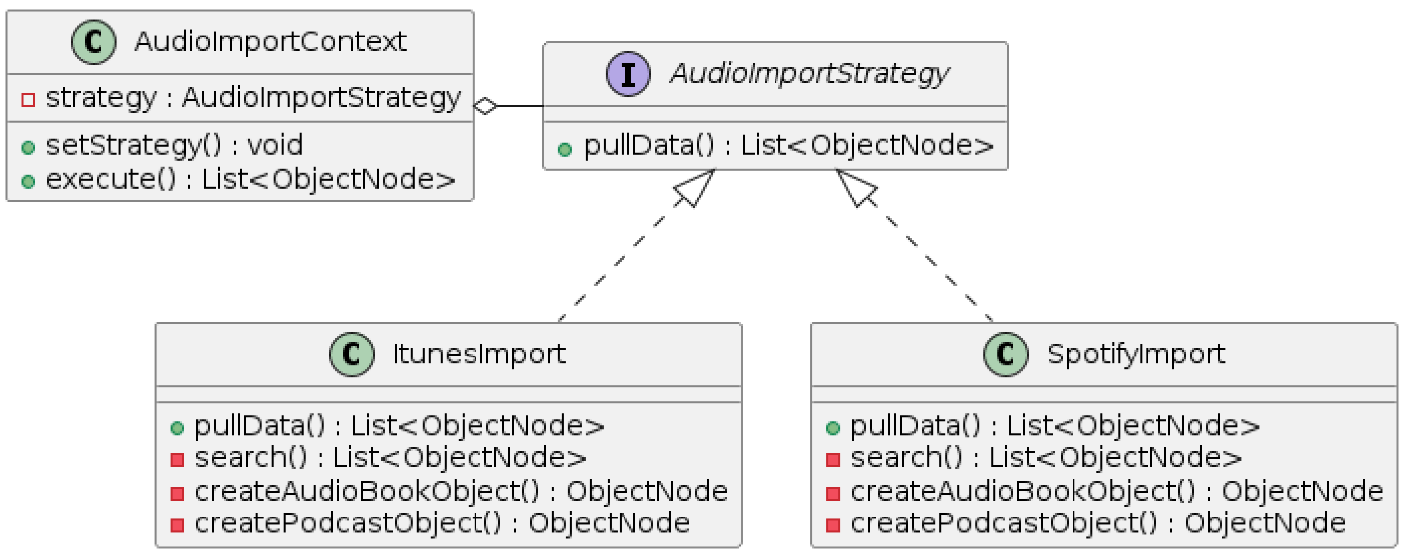
d. `setMetaData` - responsible for setting all the meta data i.e data that doesn't fall into the other categories mentioned above.

2. ConcreteBuilders - `BookBuilder`, `PodcastBuilder`, `AudioBookBuilder` : Implement the `MediaBuilder` interface and assemble the actual parts to create the product. Here they receive data in the form of ObjectNode.
    a. The builders have validation checks integrated into it. Before any field in the product is set by the builder, the data is first validated for null values and then also validated against SQL length constraints.
3. Director - `CreateMediaResource` : Takes in the type of media (AudioBook / Podcast) to be created and the provider (Itunes / Spotify). Sets a particular strategy based on the provider and constructs objects using the relevant Builder interface.
4. Client - User : Based on the user's selection in the frontend, parameters are sent to the director which finally gives a concrete product.

Motivation behind various design decisions

- **Why Builder** The reason for making a builder in this situation was necessitated by the need to break up setting fields in the concrete products, long parameter list smell that would arise, long method smell, and excessive conditional complexity.
    - We needed to break up this field setting process since if all the fields were set in one go it was causing a lot of null values in the parameter list, so we decided to break it down into logical data partitions (coreContent, mediaProfile, and metaData). Hence due to this most of the time the fields corresponding to a logical data partition are ALL set or NONE are set.
    - This gives rise to flexibility in setting various parts of the product (audiobook or podcast) based on the data we get from various media providers.
    - This improves the maintainability and flexibility
- **Why ObjectNode as Param** The builder interface have various functions, which take ObjectNode as a parameter.
    - This was needed since the functions werent setting the same fields in all concrete builders, hence they needed different parameters. We got around this challenge by encapsulating all the parameters into an ObjectNode object.
- **Why Validation in Builder** Validation checks were performed in builder since the validation step should ideally be performed **immediately** before the setting step. Or in other words the entity performing the setting of fields should also perform validation before setting, entity should encapsulate validation and setting.
    - This is required, since all the classes which use the concrete builder don't need to validate the data themselves (individually and independently) before passing it to builder.
    - It will reduce duplicate code, and will enforce the Single Responsibility Principle (SRP) and increases cohesion.

2. Strategy Pattern


C AudioImportContext
strategy : AudioImportStrategy
setStrategy() : void
execute() : List<ObjectNode>
I AudioImportStrategy
pullData() : List<ObjectNode>
C ItunesImport
pullData() : List<ObjectNode>
search() : List<ObjectNode>
createAudioBookObject() : ObjectNode
createPodcastObject() : ObjectNode
C SpotifyImport
pullData() : List<ObjectNode>
search() : List<ObjectNode>
createAudioBookObject() : ObjectNode
createPodcastObject() : ObjectNode

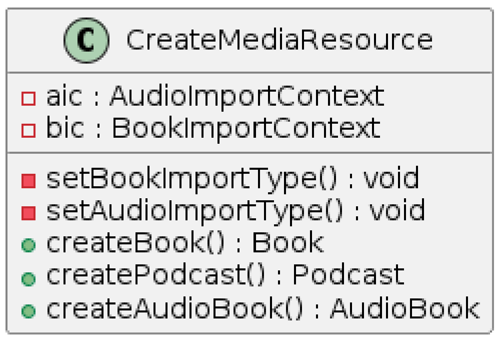
Participants

1. Context - `AudioImportContext` : Maintains a reference to one of the concrete strategies and allows switching between different strategies.
2. Strategy - `AudioImportStrategy` : It is the interface common to all the different strategies (import types). It declares a method the context uses to execute a strategy.
    a. `pullData` : gets data by making Spotify/ Itunes API calls based on the search parameters.
3. Concrete Strategies - `ItunesImport` , `SpotifyImport` : Implement variations of the algorithms (ways of importing media) that the context uses.
    a. The import strategies have validation checks integrated into it. Before making the API calls, we check for valid search types and only then make the call.
4. Client - `CreateMediaResource` : The Client here creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy. This strategy is set based on the choice of the User.

Motivation behind various design decisions

- **Why Strategy** The strategy pattern allows us to encapsulate different algorithms (import strategies) into separate classes. This makes it easy to switch between strategies dynamically without modifying the client code. Having all the different strategies in one class would lead to long method and violation of the Single Responsible Principle (SRP).
    - By encapsulating each strategy into its own class, it achieves a separation of concerns. Each strategy focuses on a specific way of importing media, making the code to easier to understand and maintain.
    - The pattern also enables us to add new import strategies in the future without disrupting the existing codebase. We can simply create new strategies by implementing the `AudioImportStrategy` class. This ensures scalability of the system.
- **Exclusion of Non-Essential Methods from Interface** We decided to exclude certain functions from the interface (like `createPodcastObject`) even though they are implemented by both the concrete strategies.
    - This was done to ensure flexibility while scaling the system in the future. This allows each concrete strategy being added later to focus on implementing only the functionality it requires. The new strategies can now be added without being burdened by unnecessary methods from the interface.
    - This also reduces the dependency between the client code and strategies. Clients interacting with interface do not need to be aware of any methods that are not relevant to their use case.

Tying together Strategy and Builder - `CreateMediaResource`


C CreateMediaResource
aic : AudioImportContext
bic : BookImportContext
setBookImportType() : void
setAudioImportType() : void
createBook() : Book
createPodcast() : Podcast
createAudioBook() : AudioBook

Description

`CreateMediaResource` : Takes in the type of media (AudioBook / Podcast) to be created and the provider (Itunes / Spotify). Sets a particular strategy based on the provider and constructs objects using the relevant Builder interface.

1. strategies are set through the `audioImportContext` (for audiobooks and podcasts) and `bookImportContext` (for books)
2. `createPodcast` , `createAudioBook` , `createBook` construct the objects based on type of media and provider
3. stuff related to frontend interaction, how we are passing it to frontend, endpoint urls etc

Why

`CreateMediaResource` was needed since we needed a single class to serve the frontend the required media based on different variables like - media type, provider.

- This meant that `CreateMediaResource` consolidated the strategy pattern and the builder pattern, and used them in tandem to create the required product.
- So for example to create a `podcast` from `spotify` , it would use the `SpotifyImport` from `AudioImportContext` to get data regrading the podcast from spotify, along with the `PodcastBuilder` to actually create the object in steps.

Implementation Details

`toString` from `AudioBook` - this helped in formatting the content from backend to frontend

```
public String toString() {
        return new JsonStringer()
                .object()
                .key("id")
                .value(id)
                .key("title")
                .value(title)
                .key("author")
                .value(author)
                .key("authorId")
                .value(authorId)
                .key("narrator")
                .value(narrator)
                .key("description")
                .value(description)
                .key("explicit")
                .value(explicit)
                .key("genre")
                .value(genre)
                .key("language")
                .value(language)
                .key("releaseDate")
                .value(releaseDate)
                .key("coverUrl")
                .value(coverUrl)
                .endObject()
                .toString();
    }
```

`setCoreContent` from `PodcastBuilder` - validates SQL related checks and sets the id, title and host for podcasts

```
/**
 * Sets the core content of the podcast based on the provided data.
 *
 * @param data The ObjectNode containing the podcast data.
 */
public void setCoreContent(ObjectNode data) {
    // Extract and set the ID from the provided data
    String id = data.get("id").getTextValue();
    if (id != null && id.length() <= Constants.MAX_ID_LEN) {
        this.pod.setId(id);
    }

    // Extract and set the title from the provided data
    String title = data.get("title").getTextValue();
    if (title != null && title.length() <= Constants.MAX_TITLE_LEN) {
        this.pod.setTitle(title);
    }

    // Extract and set the host from the provided data
    String host = data.get("host").getTextValue();
    if (host != null && host.length() <= Constants.MAX_TITLE_LEN) {
        this.pod.setHost(host);
    }
}
```

`createPodcast` from `CreateMediaResource` - takes in searchField (host, title), query (what to search for) and importType (spotify or itunes). Uses `PodcastBuilder` and `AudioImportStrategy`

```
/**
 * Endpoint to create a podcast based on the provided search parameters.
 *
 * @param searchField The field to search in (e.g., "host", "author", "title").
 * @param query The search query entered by the user.
 * @param importType The type of import (e.g., "spotify", "itunes").
 * @return A Response containing the created podcast data in JSON format.
 * @throws Exception If there is an authentication error or an error during the creation proc
ess.
 */
@GET
@Path("podcast")
@Produces(MediaType.APPLICATION_JSON)
public Response createPodcast(@QueryParam("searchField") String searchField,
                              @QueryParam("query") String query,
                              @QueryParam("importType") String importType) throws Exception {
    // Check if the client is authenticated
    if (!authenticate()) {
        throw new ForbiddenClientException();
    }

    // Initialize the response JSON object
    JSONObject response = new JSONObject();

    // Set the import type based on the query parameter
    setAudioImportType(importType);

    // Initialize the PodcastBuilder
    PodcastBuilder podcastBuilder = new PodcastBuilder();

    // Execute the audio import context to get data based on search field and query
    ArrayList<ObjectNode> dataList = this.audioImportContext.execute("podcast", searchField,
query);
    List<Podcast> podcastList = new ArrayList<Podcast>();

    // Iterate over the retrieved data and construct Podcast objects
    for (ObjectNode data : dataList) {
        // Create a new Podcast object using the builder pattern
        podcastBuilder.createMedia();
        podcastBuilder.setCoreContent(data);
        podcastBuilder.setMediaProfile(data);
        podcastBuilder.setMetaData(data);
        // Add the constructed Podcast object to the list
        podcastList.add(podcastBuilder.getResult());
    }

    // Populate the response JSON object
    response.put("type", "podcast");
    response.put("audio", podcastList);

    // Return the response
    return Response.ok(response).build();
}
```

`pullData` from `ItunesImport` - gets the data of all audio from the API in list format.

```
/**
 * Fetches data based on the search parameters using the appropriate search strategy.
 *
 * @param type The type of media (e.g., "podcast", "audiobook").
 * @param searchField The field to search in (e.g., "title", "author").
 * @param query The search query entered by the user.
 * @return An ArrayList of ObjectNode containing the retrieved data.
 * @throws Exception If an error occurs during the data retrieval process.
 */
public ArrayList<ObjectNode> pullData(String type, String searchField, String query) throws E
xception {
    // Validate the search data and get the validated search parameters
    HashMap<String, String> searchData = validateSearchData(type, searchField, query);

    // Execute the search based on the validated search parameters
    return search(searchData.get("type"), searchData.get("searchField"), searchData.get("quer
y"));
}
```

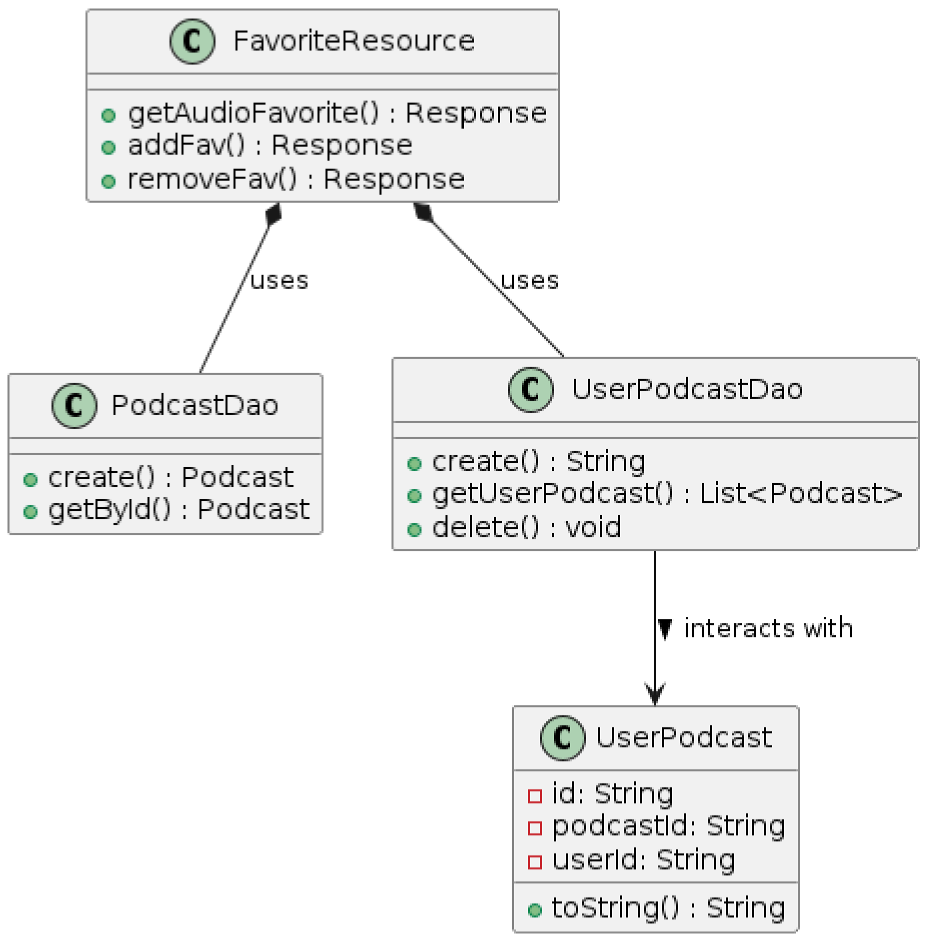
`validateSearchData` from `ItunesImport` - validates the search parameters before sending to API

```
/**
 * Validates the search parameters before executing the search.
 *
 * @param type The type of media (e.g., "audiobook", "podcast").
 * @param searchField The field to search in (e.g., "host", "author", "title").
 * @param query The search query entered by the user.
 * @return A HashMap containing the validated search parameters.
 * @throws Exception If the provided search parameters are invalid or not supported.
 */

private HashMap<String, String> validateSearchData(String type, String searchField, String qu
ery) throws Exception {
    // Check if the provided audio type is valid
    if (!type.equalsIgnoreCase("audiobook") && !type.equalsIgnoreCase("podcast")) {
        throw new Exception("Audio type is not valid");
    }

    // Check if the requested search field is supported
    if (!searchField.equalsIgnoreCase("host") && !searchField.equalsIgnoreCase("author") && !
searchField.equalsIgnoreCase("title")) {
        throw new Exception("Searching using the requested field is not supported");
    }
    // Map the provided search field to the corresponding search term used in the API
    HashMap<String, String> searchData = new HashMap<>();
    searchData.put("type", type);
    if (searchField.equalsIgnoreCase("host")) {
        searchData.put("searchField", "artistTerm");
    } else if (searchField.equalsIgnoreCase("author")) {
        searchData.put("searchField", "authorTerm");
    } else if (searchField.equalsIgnoreCase("title")) {
        searchData.put("searchField", "titleTerm");
    }
    searchData.put("query", query);

    return searchData;
}
```

`search` from `ItunesImport` - gets data from the API and converts them into ObjectNodes consisting of relevant data for each audio type.

```
/**
 * Searches for audio data based on the provided parameters using the iTunes API.
 *
 * @param type The type of media to search for (e.g., "audiobook", "podcast").
 * @param searchField The field to search in (e.g., "host", "author", "title").
 * @param query The search query entered by the user.
 * @return An ArrayList of ObjectNode containing the search results.
 * @throws Exception If no audio data is found or if there is an error in the search process.
 */
private ArrayList<ObjectNode> search(String type, String searchField, String query) throws Ex
ception {
    // Acquire the rate limiter permit before making the API call
    itunesRateLimiter.acquire();

    // Get the input stream from the iTunes API search endpoint
    InputStream inputStream = getStream(String.format(Locale.ENGLISH, itunes_api_search_endpo
int, query, type, searchField));

    // Parse the JSON response from the API
    ObjectMapper mapper = new ObjectMapper();
    JsonNode rootNode = mapper.readValue(inputStream, JsonNode.class);

    // Check if any audio data is found
    if (rootNode.get("resultCount").getIntValue() <= 0) {
        throw new Exception("No audio found");
    }

    // Retrieve the list of audio data from the JSON response
    ArrayNode dataList = (ArrayNode) rootNode.get("results");
    ArrayList<ObjectNode> audio = new ArrayList<>();

    // Iterate through each audio data and create corresponding ObjectNode
    for (JsonNode jsonNode : dataList) {
        if (type.equalsIgnoreCase("audiobook")) {
            audio.add(createAudioBookObject((ObjectNode) jsonNode));
        } else if (type.equalsIgnoreCase("podcast")) {
            audio.add(createPodcastObject((ObjectNode) jsonNode));
        }
    }
    return audio;
}
```

Favorites Functionality



- Our favorite subsystem supports **adding** favorites and also **removing** favorites
- The favorites functionality is handled primarily using 5 classes - `Podcast` , `UserPodcast` `PodcastDao` , `UserPodcastDao` , and `FavoriteResource`.
- `UserPodcast` - Contains the `id` , `userId` and the `podcastId`
  - The `UserPodcast` class essentially in this case represents a **favorite** podcast. Hence if a `UserPodcast` is created that podcast has been favorited. We did this since the only time a `Podcast` is associated with a User is when it has been favorited by a User. We refrained from naming it FavoritePodcast since this would affect the reusability of the code. Now the `UserPodcast` can be used for other use cases too.
- The `DAO` s handle saving and pulling data from the SQL database

  ```
  create cached table T_PODCAST
  create cached table T_USER_PODCAST
  ```

  - We have two SQL tables - `T_PODCAST`, and `T_USER_PODCAST`
  - This ensures that there is less data redundancy. The podcastID in `T_USER_PODCAST` is the foreign key corresponding to `T_PODCAST` .
  - This ensures that when another user favorites the same podcast, all data related to the podcast is already there in `T_PODCAST`.
- `FavoriteResource` interacts with our frontend when it comes to favoriting.
  - `getAudioFavorite` takes in the audio type (podcast or audiobook) and returns all the favorites for that particular user and that audio type. This essentially returns the entries in `T_USER_PODCAST` SQL table corresponding to that user. These podcastID in these entries are again used to query `T_PODCAST` table to get podcast details, which are then sent to frontend.
  - `addFav` adds a podcast as favorite for that user. This uses the `create` functions in `UserPodcastDao` and the `create` function in `PodcastDao` to create an entry in `T_USER_PODCAST` and `T_PODCAST`.

    - `removeFav` removes a podcast as favorite, by fremoving the entry for that specific podcast and user from the `T_USER_PODCAST`. **NOTE :** the original data for the podcast is NOT deleted from `T_PODCAST`.
- **This functionality is identical for audiobooks too.**

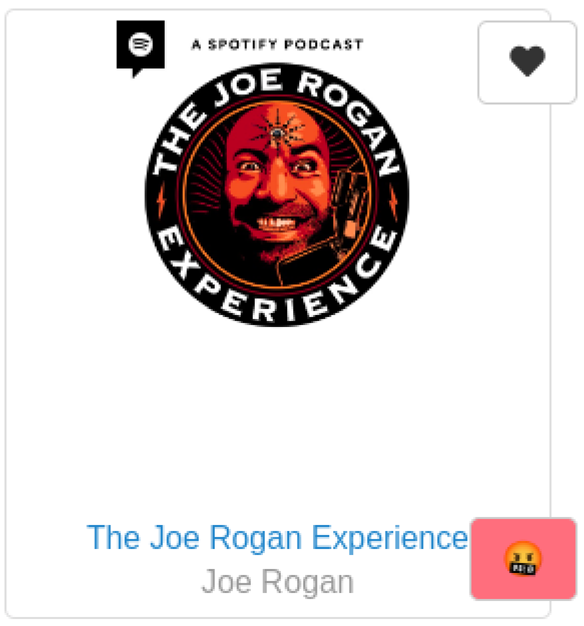
Features of the system

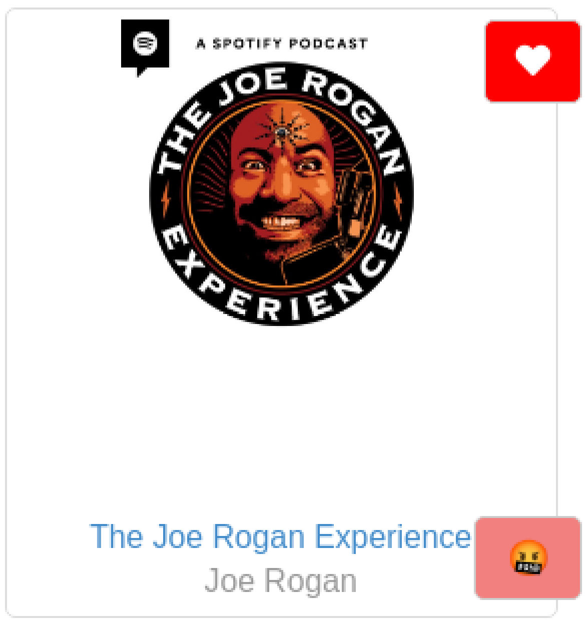
1. Audiobook / Podcast Details

Each AudioBook / Podcast in the library has the following information:

- Title
- Author(s)
- Description
- Genre(s)
- Thumbnail Image URL
- Explicitness - Represented by the angry emoji on bottom right

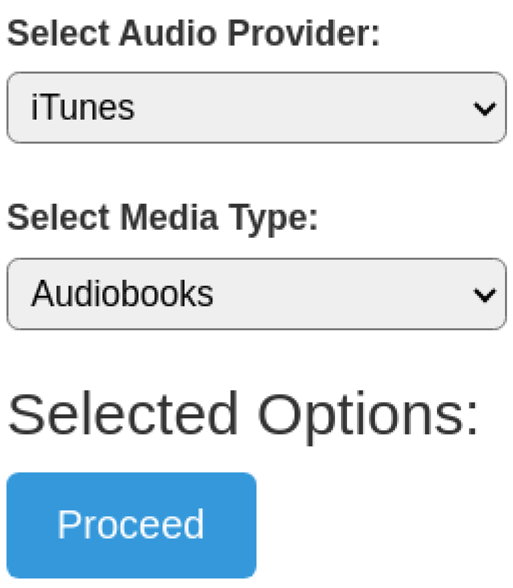
Clicking on the heart top right adds the audiobook/ podcast to favorites





2. Search Functionality

Users need to first select the media type (audiobooks or podcasts) and media provider (itunes or spotify)



Select Audio Provider:

iTunes

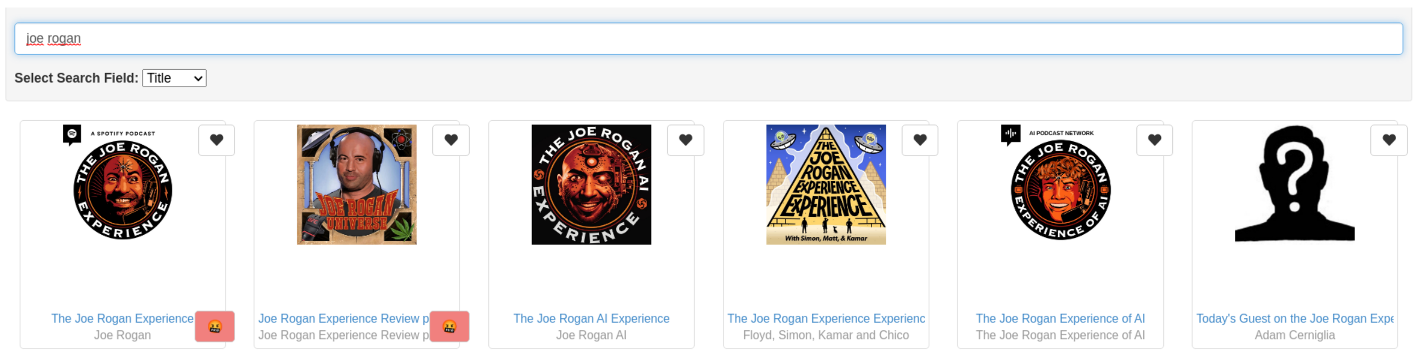
Select Media Type:

Audiobooks

Selected Options:

After selecting, users can choose to search based on the author or title (on itunes) and additionally do a blanket search on spotify.

a. By title


joe rogan

Select Search Field: Title

The Joe Rogan Experience
Joe Rogan

Joe Rogan Experience Review p
Joe Rogan Experience Review p

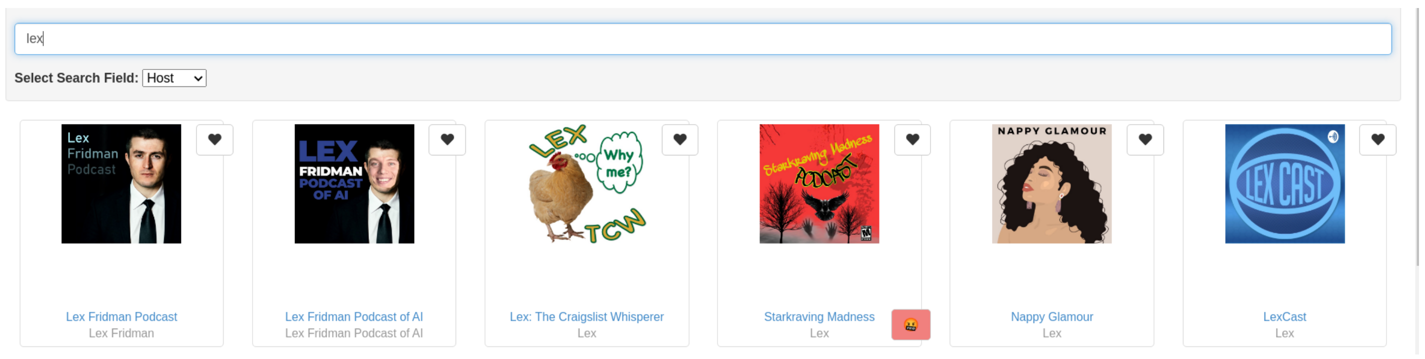
The Joe Rogan AI Experience
Joe Rogan AI

The Joe Rogan Experience Experienc
Floyd, Simon, Kamar and Chico

The Joe Rogan Experience of AI
The Joe Rogan Experience of AI

Today's Guest on the Joe Rogan Expe
Adam Cerniglia

b. By host/ author


lex

Select Search Field: Host

Lex Fridman Podcast
Lex Fridman

Lex Fridman Podcast of AI
Lex Fridman Podcast of AI

Lex: The Craigslist Whisperer
Lex

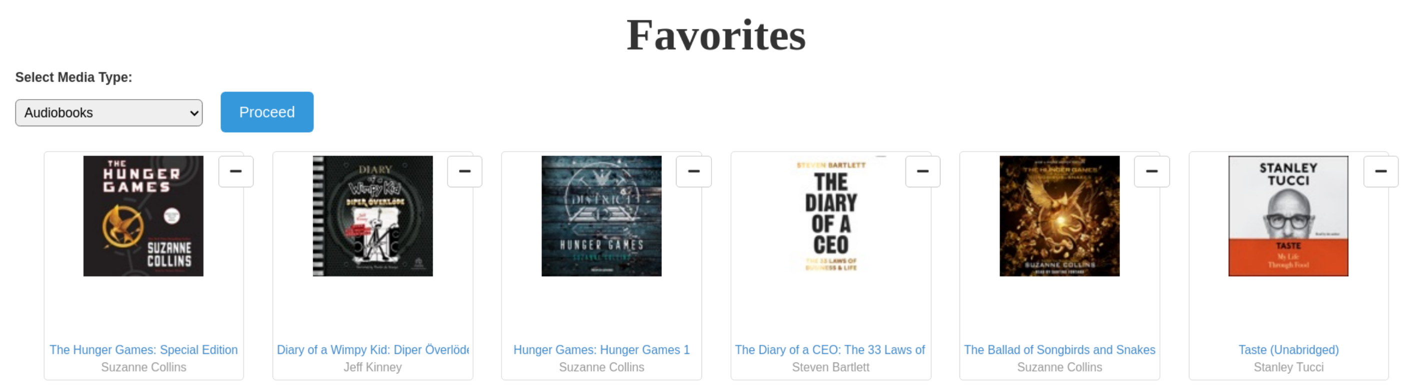
Starkraving Madness
Lex

Nappy Glamour
Lex

LexCast
Lex

3. Favorites Section

Users can choose to view their favorite podcasts and audiobooks by selecting media type. Clicking the minus icon removes the podcast from their favorites.


Favorites

Select Media Type: Audiobooks

Proceed

The Hunger Games: Special Edition
Suzanne Collins

Diary of a Wimpy Kid: Diper Överlöde
Jeff Kinney

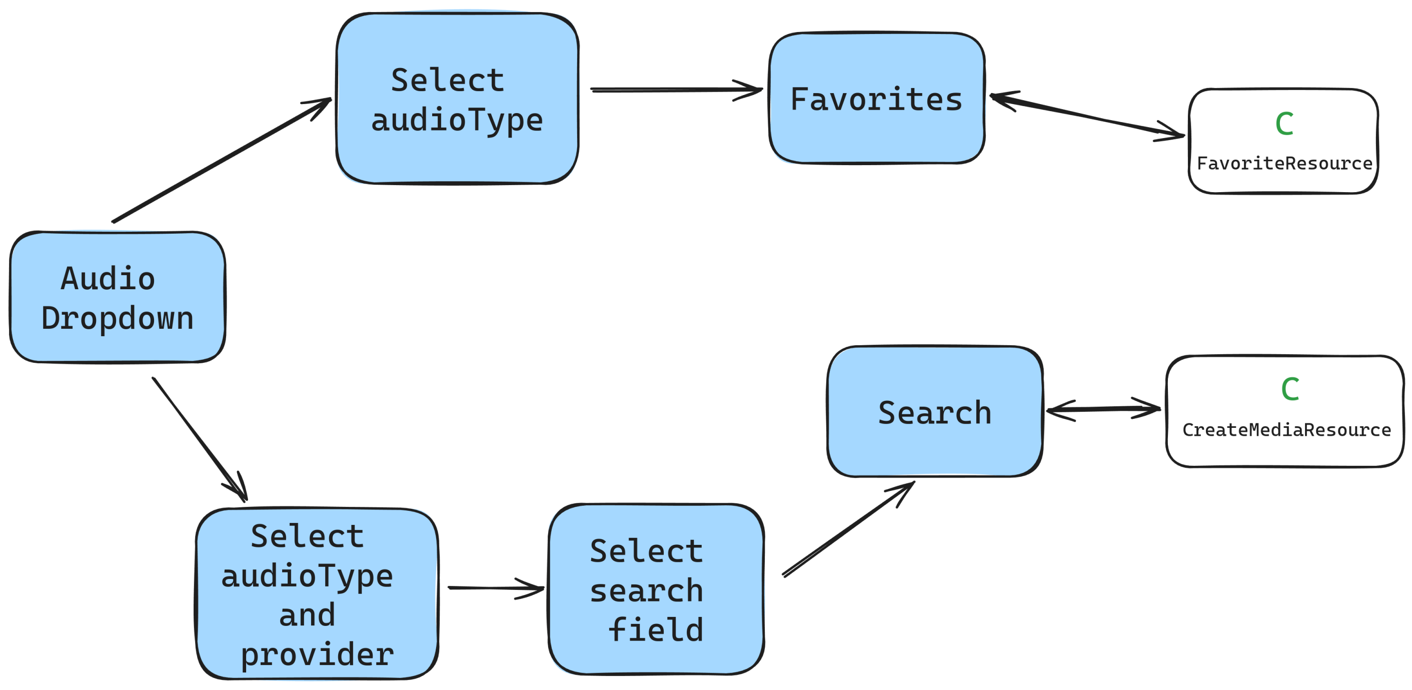
Hunger Games: Hunger Games 1
Suzanne Collins

The Diary of a CEO: The 33 Laws of
Steven Bartlett

The Ballad of Songbirds and Snakes
Suzanne Collins

Taste (Unabridged)
Stanley Tucci

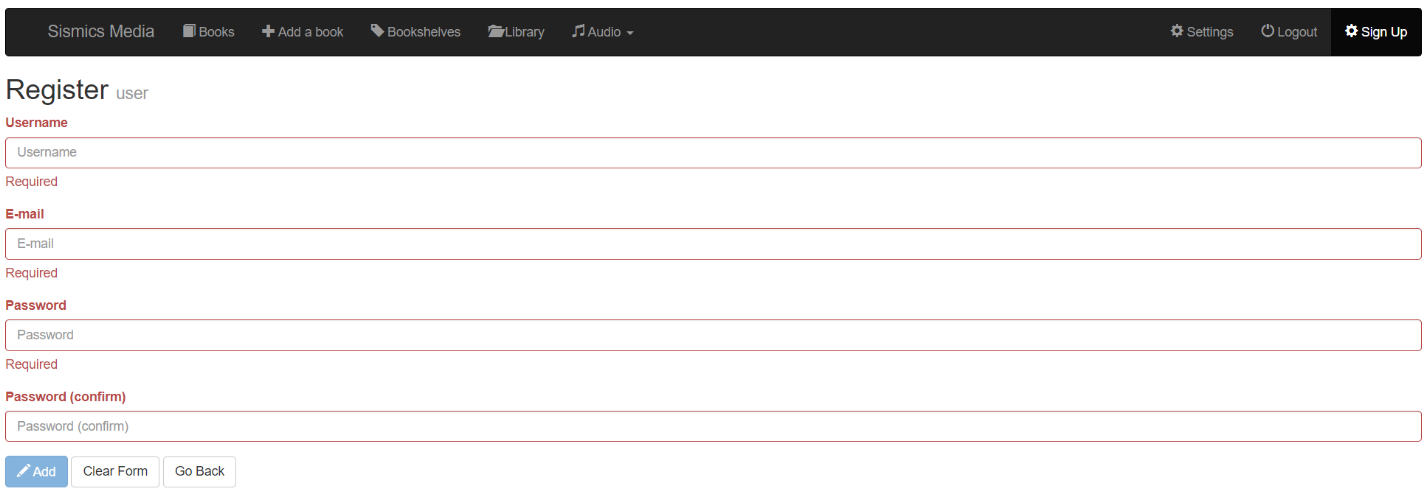
System flow diagram



Blue Boxes are frontend pages

Better user management

Phase 1 - Creation of sign up page



Register user

Username
Username
Required

E-mail
E-mail
Required

Password
Password
Required

Password (confirm)
Password (confirm)

Existing files modified:

`app.js`

`$stateProvider` in `.config` in `app.js` is used to configure states or routes for the application. Each state has an associated URL, an HTML template which loads on that particular URL, and an AngularJS form controller that handles the logic related to that view.

In the `$stateProvider` in this file, we add a state for the newly created signup page.

```
  .state('signup', {
   url: '/signup',
   views: {
      'page': {
        templateUrl: 'partial/signup.html',
        controller: 'SignUp'
      }
    }
   })
```

`index.html`

A button redirecting to the signup page is added to the existing navbar.

```
 <li ng-class="{active: $state.current.url == '/signup'}"><a href="#/signup"><span class="gly
phicon glyphicon-cog"></span> Sign Up</a></li>
```

The JavaScript file containing the controller (`SignUp.js`) is also included.

```
<script src="app/controller/SignUp.js" type="text/javascript"></script>
```

`UserResource.java`

In the `register` method in `UserResource` , before the user is created, some authentication checks are made to ensure that only admins can add new users.

```
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        checkBaseFunction(BaseFunction.ADMIN);
```

As we wish to implement a sign-up feature for non-admin users and users not logged in, we remove this snippet - this lets us use this method to enable non-admin users to sign up to the website.

New files created

`signup.html`

This file consists of an input form with 4 parameters: **username, email address, password and confirm password**. This form uses AngularJS directives (`required`, `ng-minlength`, `ng-maxlength`, `ui-validate`), and `ng-model` is used to bind the form data to the AngularJS controller scope. Validation of input in the form is done via AngularJS.
The main part of this form was taken from the form to add user/edit user information in `settings.user.edit.html`.

```
    <form role="form" name="editUserForm" novalidate>
        <div class="form-group" ng-class="{ 'has-error': !editUserForm.username.$valid, 'has-
success': editUserForm.username.$valid }">
            <label class="control-label" for="inputUsername">Username</label>

            <div class="controls">
                <input name="username" class="form-control" type="text" id="inputUsername" re
quired ng-disabled="isEdit()"
                       ng-minlength="3" ng-maxlength="50" placeholder="Username" ng-model="us
er.username"/>
                <span class="help-block" ng-show="editUserForm.username.$error.required">Requ
ired</span>
                <span class="help-block" ng-show="editUserForm.username.$error.minlength">Too
short</span>
                <span class="help-block" ng-show="editUserForm.username.$error.maxlength">Too
long</span>
            </div>
        </div>
        <div class="form-group" ng-class="{ 'has-error': !editUserForm.email.$valid, 'has-suc
cess': editUserForm.email.$valid }">
            <label class="control-label" for="inputEmail">E-mail</label>

            <div class="controls">
                <input name="email" class="form-control" type="email" id="inputEmail" require
d
                       ng-minlength="3" ng-maxlength="50" placeholder="E-mail" ng-model="use
r.email"/>
                <span class="help-block" ng-show="editUserForm.email.$error.required">Require
d</span>
                <span class="help-block" ng-show="editUserForm.email.$error.email">Must be a
valid e-mail</span>
                <span class="help-block" ng-show="editUserForm.email.$error.minlength">Too sh
ort</span>
                <span class="help-block" ng-show="editUserForm.email.$error.maxlength">Too lo
ng</span>
            </div>
        </div>
        <div class="form-group" ng-class="{ 'has-error': !editUserForm.password.$valid, 'has-
success': editUserForm.password.$valid }">
            <label class="control-label" for="inputPassword">Password</label>

            <div class="controls">
                <input name="password" class="form-control" type="password" id="inputPasswor
d" ng-required="!isEdit()"
                       ng-minlength="8" ng-maxlength="50" placeholder="Password" ng-model="us
er.password"/>
                <span class="help-block" ng-show="editUserForm.password.$error.required">Requ
ired</span>
                <span class="help-block" ng-show="editUserForm.password.$error.minlength">Too
short</span>
                <span class="help-block" ng-show="editUserForm.password.$error.maxlength">Too
long</span>
```

```
            </div>
        </div>
        <div class="form-group" ng-class="{ 'has-error': !editUserForm.passwordconfirm.$vali
d, 'has-success': editUserForm.passwordconfirm.$valid }">
            <label class="control-label" for="inputPasswordConfirm">Password (confirm)</label
>

            <div class="controls">
                <input name="passwordconfirm" class="form-control" type="password" id="inputP
asswordConfirm" ng-required="!isEdit()"
                       ui-validate="'$value == user.password'" ui-validate-watch="'user.passw
ord'"
                       placeholder="Password (confirm)" ng-model="user.passwordconfirm"/>
                <span class="help-block" ng-show="editUserForm.passwordconfirm.$error.validat
or">Password and password confirmation must match</span>
            </div>
        </div>
        <button type="submit" class="btn btn-primary" ng-click="signup()" ng-disabled="!editU
serForm.$valid">
            <span class="glyphicon glyphicon-pencil"></span> Add
        </button>
```

The form also contains 2 other buttons - `Clear Form` and `Go Back` .

- `Clear Form` refreshes the page, clearing all the data that is currently added to the form.
- `Go Back` returns you to the current homepage/login page, depending on your login status.

```
        <button type="button" class="btn btn-default" ng-click="clearForm()">Clear Form</butt
on>
        <button type="button" class="btn btn-default" ng-click="redirectToHomePageOrLogin()">
Go Back</button>
```

`SignUp.js`

This file is an AngularJS controller that is responsible for handling functionalities related to the `signup.html` page. These functionalities include user registration, form clearing and page redirection.

- User registration is done using the `signup` function. The `PUT` request is sent to the `register` method in `UserResource` with the form details.

```
    /**
     * Register a new user.
     */
    $scope.signup = function() {
        var promise = null;

        promise = Restangular
            .one('user')
            .put($scope.user);

        promise.then(function() {
            window.alert("created user");
            $state.transitionTo('main');
        }).catch(function (error) {
            console.error("Error encountered during signup:", error);
            window.alert("An error occurred while creating the user: " + error.data.message);
        });
    };
```

- Form clearance is done by the `clearForm` function. Reloading the signup page through `location.reload()` clears the form by resetting all the inputs.

```
    /**
     * Clear form by reloading page
     */
    $scope.clearForm = function() {
        console.log("clearing form");
        location.reload();
    };
```

- Page redirection is done using the `redirectToHomePageOrLogin` function. Redirection is achieved using a transition to the `main` state, which is either the homepage (`book`) or `login` depending on the user's login state.

```
    /**
     * Redirect to home page or login based on user authentication status.
     */
    $scope.redirectToHomePageOrLogin = function() {
        $state.transitionTo('main');
    }
```

```
/**
* Main controller
*/
App.controller('Main', function($scope, $rootScope, $state, User) {
  User.userInfo().then(function(data) {
    if (data.anonymous) {
      $state.transitionTo('login');
    } else {
      $state.transitionTo('book');
    }
  });
});
```

Phase 2 - Implementing email uniqueness checker

In `UserDao.java`, username uniqueness is checked by using a database query using the below code snippet:

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

A similar logic was used to check for email uniqueness. A database query was made using the provided email, and the query results were used to determine whether the provided email already existed in the database or not.

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

Phase 3 - Implementing design pattern for validation

After a request with registration form details is sent to the `register` method in `UserResource`, the user details are checked across 2 stages.

- **Stage 1 (Validation)** - in `UserResource.java` , the user details are verified using methods from `ValidationUtil`.
    - The username length and password length were validated using `validateLength` .
    - Email was additionally validated using `validateEmail` using regex.

`UserResource.java`

```
        // Validate the input data
        username = ValidationUtil.validateLength(username, "username", Constants.MIN_USERNAME
_LEN, Constants.MAX_USERNAME_LEN);
        ValidationUtil.validateAlphanumeric(username, "username");
        password = ValidationUtil.validateLength(password, "password", Constants.MIN_PWD_LEN,
Constants.MAX_PWD_LEN);
        email = ValidationUtil.validateLength(email, "email", Constants.MIN_EMAIL_LEN, Consta
nts.MAX_EMAIL_LEN);
        ValidationUtil.validateEmail(email, "email");

        // Create the user
        User user = UserResourceHelper.createUser(new UserDto(null, password, username, emai
l, null));

        // Create the user
        UserDao userDao = new UserDao();

        try {
            userDao.create(user);
        } catch (Exception e) {
            if ("AlreadyExistingUsername".equals(e.getMessage())) {
                throw new ServerException("AlreadyExistingUsername", "Username already used",
e);
@@ -102,6 +99,38 @@ public Response register(
                throw new ServerException("UnknownError", "Unknown Server Error", e);
            }
        }
```

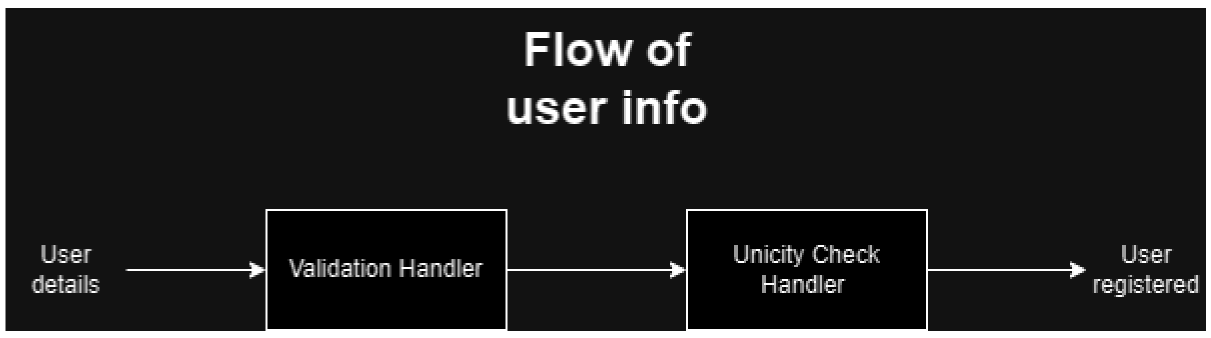
- **Stage 2 (Uniqueness check)**

The user data is sent to the `createDao` method in `UserDao.java` where the uniqueness of the username and email provided is checked using a database query.

`UserDao.java`

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

As these validation-related processes take place sequentially, we can apply the **Chain of Responsibility** design pattern to it. The 2 steps, `Validation` and `UnicityCheck` are linked together in a chain, and the flow of user information can be represented as follows:



Implementation

`RegisterHandler` is the base interface created for all the other handlers.

`interface RegisterHandler`

```
public interface RegisterHandler {
    void setNextHandler(RegisterHandler nextHandler);
    void handleRequest(User user) throws Exception;
}
```

`ValidationHandler` consists of the input validation checks that were earlier a part of `UserResource`. It handles requests related to validation of the form input.

`ValidationHandler`

```
public class ValidationHandler implements RegisterHandler {
    private RegisterHandler nextHandler;

    public void setNextHandler(RegisterHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(User user) throws Exception {
        String username = ValidationUtil.validateLength(user.getUsername(), "username", Const
ants.MIN_USERNAME_LEN, Constants.MAX_USERNAME_LEN);
        ValidationUtil.validateAlphanumeric(username, "username");
        String password = ValidationUtil.validateLength(user.getPassword(), "password", Const
ants.MIN_PWD_LEN, Constants.MAX_PWD_LEN);
        String email = ValidationUtil.validateLength(user.getEmail(), "email", Constants.MIN_
EMAIL_LEN, Constants.MAX_EMAIL_LEN);
        ValidationUtil.validateEmail(email, "email");

        if (nextHandler != null) {
            nextHandler.handleRequest(user);
        }
    }
}
```

`UnicityCheckHandler` consists of the database queries for checking unique username and email, which was earlier a part of `UserDao`. This handler handles requests related to checking uniqueness of the username and email provided in the form input.

`UnicityCheckHandler`

```
public class UnicityCheckHandler implements RegisterHandler {
    private RegisterHandler nextHandler;

    public void setNextHandler(RegisterHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(User user) throws Exception {
        EntityManager em = ThreadLocalContext.get().getEntityManager();

        Query q = em.createQuery("select u from User u where u.email = :email and u.deleteDat
e is null");
        q.setParameter("email", user.getEmail());
        List<?> emailResult = q.getResultList();
        if (emailResult.size() > 0) {
            throw new Exception("AlreadyExistingEmail");
        }

        q = em.createQuery("select u from User u where u.username = :username and u.deleteDat
e is null");
        q.setParameter("username", user.getUsername());
        List<?> usernameResult = q.getResultList();
        if (usernameResult.size() > 0) {
            throw new Exception("AlreadyExistingUsername");
        }

        if (nextHandler != null) {
            nextHandler.handleRequest(user);
        }
    }
}
```

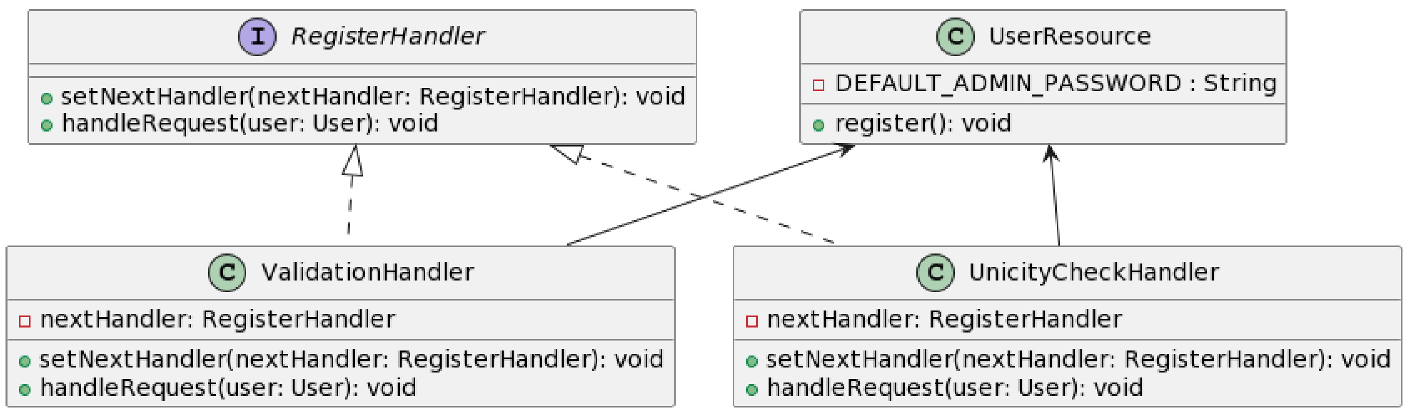
The `register` method in `UserResource` is modified to accommodate the handlers created.

As the `handleRequest` method takes `User` as input, we create the `User` before passing the details into the chain. The handlers and the chain are then initialized, after which the user details are sent to the first component of the chain, `validationHandler`. If both checks are passed without any exceptions thrown, we create the `UserDao` for the User.

modified `register` in `UserResource`

```
        // Create the user
        User user = UserResourceHelper.createUser(new UserDto(null, password, username, emai
l, null));

        // Create chain of responsibility
        RegisterHandler validationHandler = new ValidationHandler();
        RegisterHandler unicityCheckHandler = new UnicityCheckHandler();

        validationHandler.setNextHandler(unicityCheckHandler);

        try {
            validationHandler.handleRequest(user);
        } 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", "Email already used", e);
            } else {
                throw new ServerException("UnknownError", "Unknown Server Error", e);
            }
        }

        // Both validation and unicity checks passed, now create the user and DAO
        UserDao userDao = new UserDao();
```

UML Diagram for Chain of Responsibility



RegisterHandler
setNextHandler(nextHandler: RegisterHandler): void
handleRequest(user: User): void
UserResource
DEFAULT_ADMIN_PASSWORD : String
register(): void
ValidationHandler
nextHandler: RegisterHandler
setNextHandler(nextHandler: RegisterHandler): void
handleRequest(user: User): void
UnicityCheckHandler
nextHandler: RegisterHandler
setNextHandler(nextHandler: RegisterHandler): void
handleRequest(user: User): void

NOTE: **ASSUMPTIONS**

- You can access the sign up page whether you are logged in or not.
- You are also not automatically logged in with your sign up details, you would have to log in again with your newly created details.