Sismics Books
Project-2
Team_11

Introduction

- **Objective:** Elevate functionality and user experience in the Sismics Books repository.
- **Design and OOP Principles:** Implementing key features and refining existing ones using best practices in design patterns and Object-Oriented Programming.
- **User Management:** Enhance user account features for a smoother experience.
- **Common Library System:** Introduce a unified system for seamless access to resources.
- **Integration with Audio Services:**
  - Integrate Spotify and iTunes for a diverse range of audiobooks and podcasts.
  - Provide seamless service integration for users.
- **Privacy Customization:** Enable users to set privacy levels for their bookshelves, ensuring a secure and personalized reading environment

Functional & Non-Functional Requirements

Functional Requirements:

- **User Management:** Users should be able to register, log in, and manage their profiles.
- **Book Management:** Users should be able to add, edit, view, and delete books, including managing book metadata and tags.
- **Tag Management:** Users should be able to create, edit, and delete bookshelves or tags to categorize their books.
- **Audio Management:** Users should be able to link their audio content (e.g., from Spotify or iTunes) to their bookshelves.
- **Settings Management:** Admin/Users should be able to manage their account settings, session settings, and preferences.

Non-Functional Requirements:

- **Performance:** The application should load quickly and handle a reasonable number of simultaneous users without significant degradation in performance.
- **Security:** User data should be securely stored, and user authentication should be robust to prevent unauthorized access.
- **Usability:** The application should be easy to use, with a clear and intuitive user interface.
- **Scalability:** The application should be able to scale to accommodate an increasing number of users and data.
- **Reliability:** The application should be reliable, with minimal downtime

Components and Subsystem

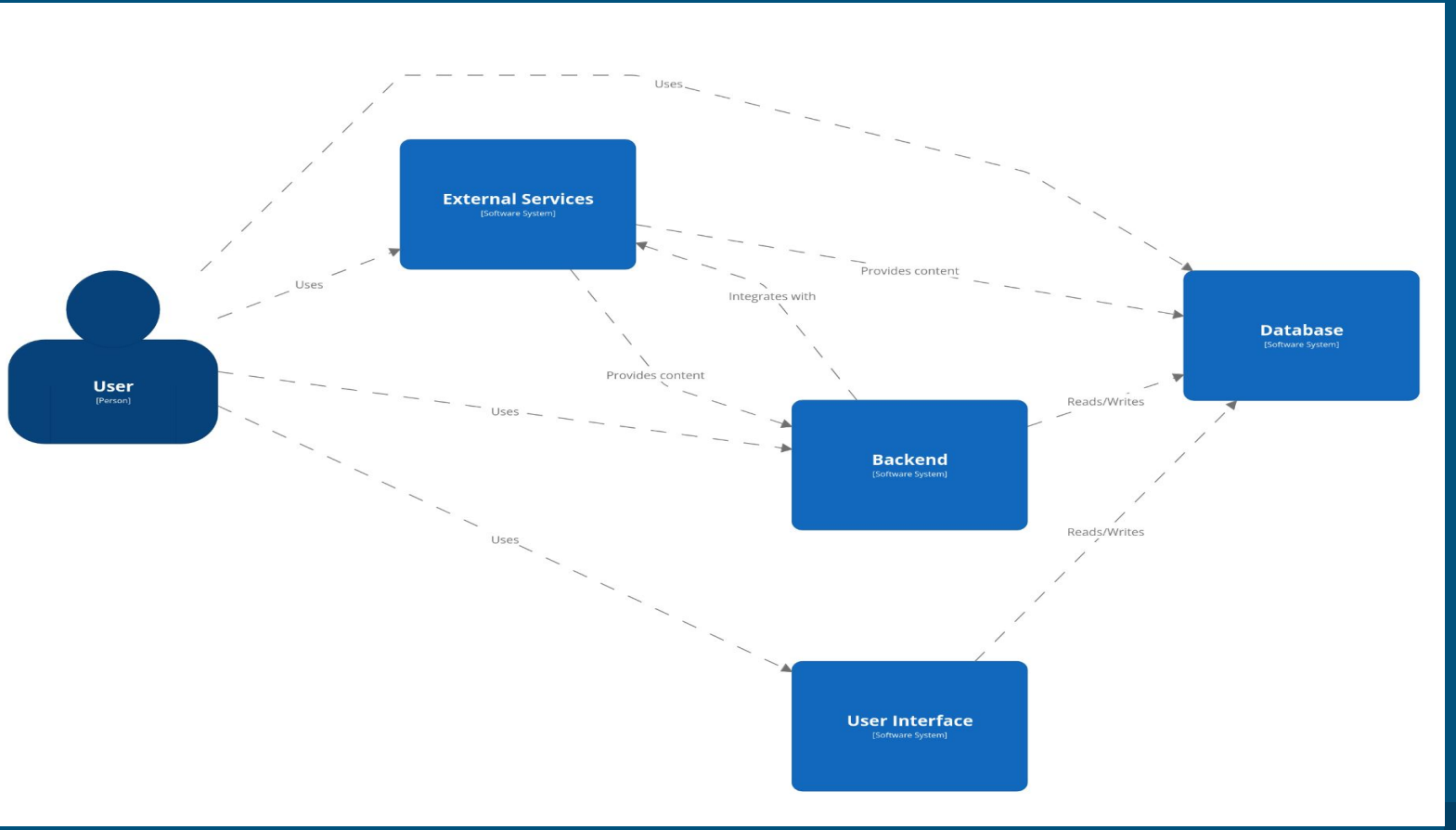
Components and Connectors:

- **Frontend:** AngularJS-based, handling user interactions and displaying data.
- **Backend:** Java-based, processing requests from the frontend, interacting with the database, and handling business logic.
- **Database:** Relational database, storing user data, book information, and settings.
- **External Services:** Integration with Spotify or iTunes for audio content.

Subsystems:

- **User Management Subsystem**
- **Book Management Subsystem**
- **Bookshelves Management Subsystem**
- **Audio Management Subsystem**
- **Settings Management Subsystem**

System Context Diagram



Feature 1 - Better User Management(Objective)

User Registration Page:

- Users can register by providing a username, email, and password.
- Accessible directly from the login page for seamless transition.

Email Uniqueness Check:

- System verifies that entered email addresses are unique.
- Prevents duplicate registrations and ensures data integrity.

Password Confirmation:

- Users must enter their password twice for security.
- System compares entries to ensure they match, reducing typos.

Validation Checks:

- **Valid Email ID:**
  - Ensures correct email format and absence of invalid characters.
- **Password Requirements:**
  - Passwords must be at least 8 characters long.
  - Enhances data validity and security, reducing errors.

Feature 1 ....
(Implementation of Builder Pattern)

Builder Pattern:

- In the above feature, we often encounter the need to create complex User objects with numerous optional parameters. To simplify this process and enhance code readability and maintainability, we have decided to implement the Builder Pattern for constructing User objects. The Builder Pattern separates the construction of an object from its representation, allowing flexible creation processes for different representations of the same object.
  - To implement the Builder Pattern for the User class, We would create a new file named UserBuilder.java in the same package as our User class, which is likely com.sismics.books.core.model.jpa. This keeps the code organized and consistent with the existing structure.

```
// UserBuilder.java
package com.sismics.books.core.model.jpa;
public class UserBuilder {
        private String username;
        private String password;
        private String email;
        public UserBuilder setUsername(String username) {
                this.username = username;
                return this;
        }
        public UserBuilder setPassword(String password) {
                this.password = password;
                return this;
        }
        public UserBuilder setEmail(String email) {
                this.email = email;
                return this;
        }
        // ... other setter methods ...
        public User build() {
                User user = new User();
                user.setUsername(username);
                user.setPassword(password);
                user.setEmail(email);
                // ... set other fields ...
                return user;
        }
}
```

Feature 1 …
(Implementation of Builder pattern)

- Using the UserBuilder: In our UserResource.java class, where we often create User objects, we import UserBuilder and utilize it to construct User instances.

Change in Code Base

- Added File:
  books-web/src/main/webapp/app/controller/Signup.js
  books-core/src/main/java/com/sismics/books/core/model/jpa/UserBuilder.java
- Modified File:
  books-web/src/main/java/com/sismics/books/rest/resource/UserResource.java
  books-web/src/main/webapp/app/app.js

```
// UserResource.java
package com.sismics.books.rest.resource;
import com.sismics.books.core.model.jpa.UserBuilder;
public class UserResource extends BaseResource {
        // ... existing code …
        @PUT
        @Path("/signup")
        @Produces(MediaType.APPLICATION_JSON)
        public Response signup(
                @FormParam("username") String username,
                @FormParam("password") String password,
                @FormParam("locale") String localeId,
                @FormParam("email") String email) throws JSONException {
        User user = new UserBuilder()
                .setUsername(username)
                .setPassword(password)
                .setEmail(email) // ... set other fields ...
                .build(); // ... existing code …
        }// ... existing code ...
}
```

Feature 2 - Common Library (Objective)

- Enable all users to view and rate books.
- Book Details
- User Management Features
- Filtering as per Genre,Authors and ratings
- Book Ranking

Feature 2 - Common Library (Ranking Implementation using Strategy Pattern)

- New class Genre added with relationship with Book Class (Association).
- Authors are saved with comma separated format and increased to 1000 char.
- Separate class BookRating for ranking created. BookRating class will store the combination of user and book rated by user.
- Ranking logic implemented using strategy design pattern.
- filtering logic is handled in the backend java code based on object oriented principles . (Done in PR #50)

Feature 2 - Common Library (Ranking Implementation using Strategy Pattern)

For getting the data based on average rating and number of ratings, we have to use strategy pattern.

BookRankingStrategy.java - Strategy interface to which all other concrete strategies must implement rank books.

AverageRatingRankingStrategy.java - concrete strategy for Average rating ranking criterion.

NumberOfRatingsRankingStrategy.java - concrete strategy for Number of ratings ranking criterion.

BookRankingResource.java - This is the controller class.

BookRankingDto.java - This is Dto class to store the ranking information.

Feature 2 - Common Library (Ranking Implementation using Strategy Pattern)

```java
package com.sismics.books.core.strategy;

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

import java.util.List;

public interface BookRankingStrategy {
    List<BookRankingDto> rankBooks();
}
```

BookRankingStrategy.java class

Feature 2 - Common Library (Ranking Implementation using Strategy Pattern)

```java
public class AverageRatingRankingStrategy implements BookRankingStrategy {
    private final BookRatingDao bookRatingDao;

    public AverageRatingRankingStrategy(BookRatingDao bookRatingDao) {
        this.bookRatingDao = bookRatingDao;
    }

    @Override
    public List<BookRankingDto> rankBooks() {
        return bookRatingDao.rankBooksByAverageRating();
    }
}
```

AverageRatingRankingStrategy.java

Feature 2 - Common Library (Ranking Implementation using Strategy Pattern)

```java
public class NumberOfRatingsRankingStrategy implements BookRankingStrategy {
    private final BookRatingDao bookRatingDao;

    public NumberOfRatingsRankingStrategy(BookRatingDao bookRatingDao) {
        this.bookRatingDao = bookRatingDao;
    }

    @Override
    public List<BookRankingDto> rankBooks() {
        return bookRatingDao.rankBooksByNumberOfRatings();
    }
}
```

NumberOfRatingsRankingStrategy.java

Feature 3: Online Integration (Objective)

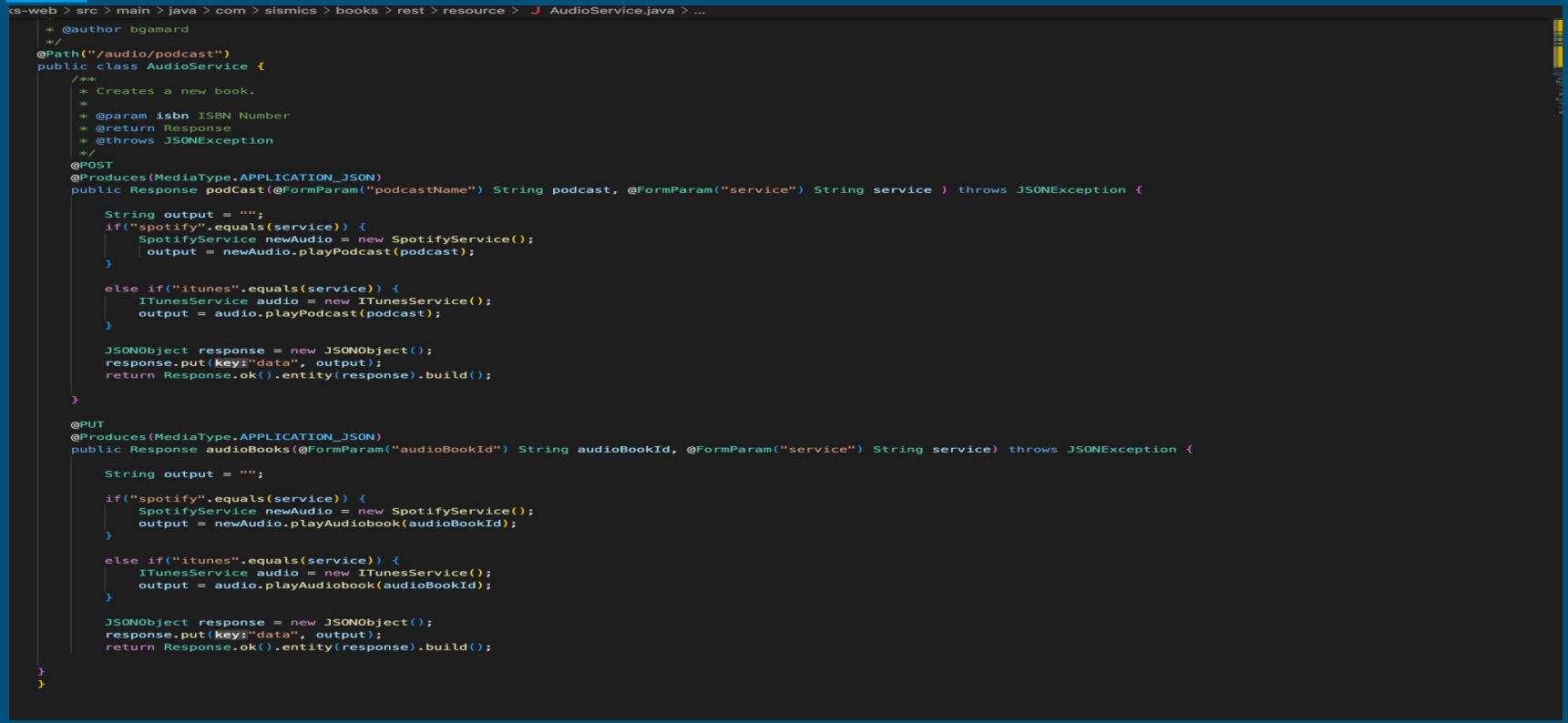
Our main goal is to elevate the user experience by incorporating a choice of audio content providers—Spotify and iTunes—directly into our platform. This seamless integration will allow users to select their favored service and type of content, such as audiobooks or podcasts, enhancing engagement and broadening the scope of our platform’s capabilities.

Feature 3: Online Integration (Implementation using strategy pattern)

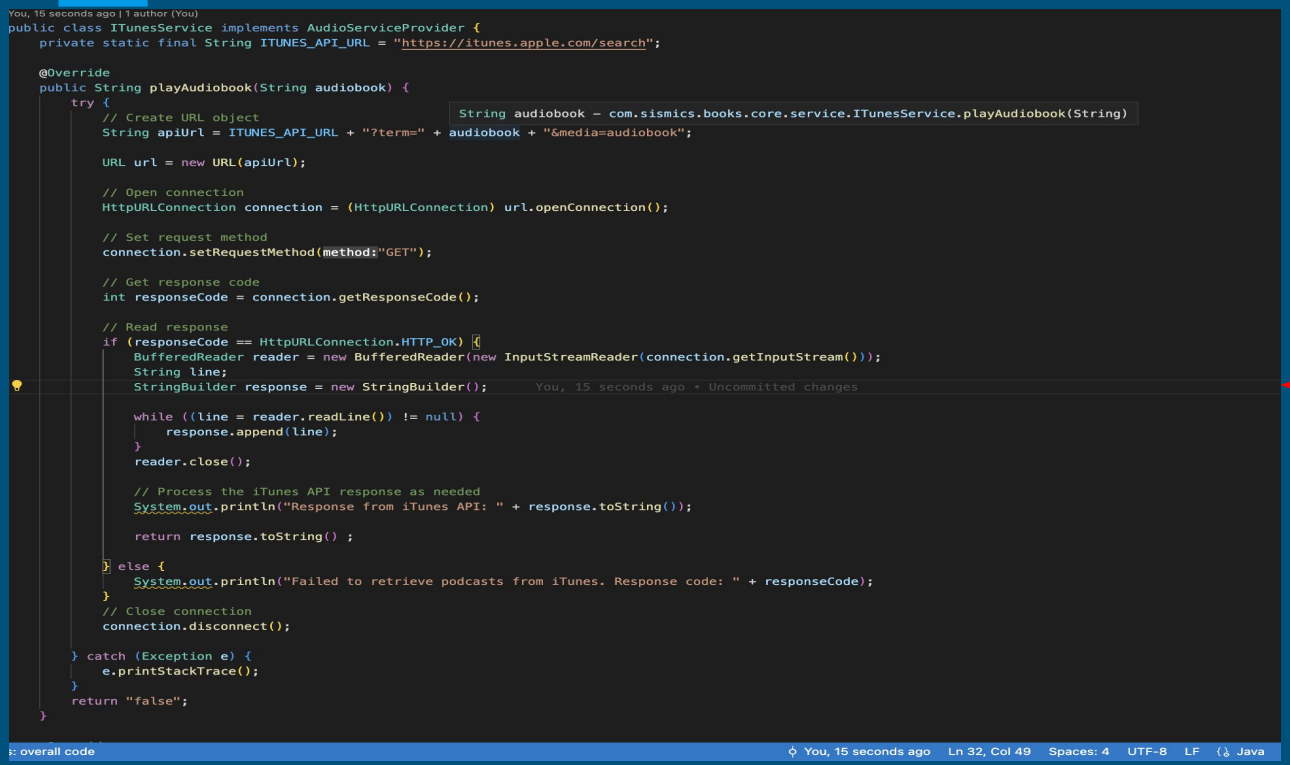
The Strategy Pattern is implemented through the AudioServiceProvider interface, which defines two methods: playAudiobook and playPodcast. This interface acts as a common interface for all audio service providers, allowing them to be used interchangeably.

- AudioServiceProvider.java: Main interface class
- SpotifyService.java and ITunesService.java: Implements the AudioServiceProvider interface, providing the specific implementation for Spotify and iTunes services.
- AudioService.java : This acts as context for strategy pattern
- Front End changes were done in index.html,app.js and AudioService.js.

Feature 3: Online Integration (Implementation using strategy pattern)



Feature 3: Online Integration (Implementation using strategy pattern)



```
You, 15 seconds ago | 1 author (You)
public class ITunesService implements AudioServiceProvider {
    private static final String ITUNES_API_URL = "https://itunes.apple.com/search";

    @Override
    public String playAudiobook(String audiobook) {
        try {
            // Create URL object
            String apiUrl = ITUNES_API_URL + "?term=" + audiobook + "&media=audiobook";

            URL url = new URL(apiUrl);

            // Open connection
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();

            // Set request method
            connection.setRequestMethod(method:"GET");

            // Get response code
            int responseCode = connection.getResponseCode();

            // Read response
            if (responseCode == HttpURLConnection.HTTP_OK) {
                BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
                String line;
                StringBuilder response = new StringBuilder();      You, 15 seconds ago • Uncommitted changes

                while ((line = reader.readLine()) != null) {
                    response.append(line);
                }
                reader.close();

                // Process the iTunes API response as needed
                System.out.println("Response from iTunes API: " + response.toString());

                return response.toString() ;

            } else {
                System.out.println("Failed to retrieve podcasts from iTunes. Response code: " + responseCode);
            }
            // Close connection
            connection.disconnect();

        } catch (Exception e) {
            e.printStackTrace();
        }
        return "false";
    }
```

String audiobook – com.sismics.books.core.service.ITunesService.playAudiobook(String)

: overall code — You, 15 seconds ago — Ln 32, Col 49 — Spaces: 4 — UTF-8 — LF — Java

```
package com.sismics.books.core.service;

You, 1 second ago | 1 author (You)
public interface AudioServiceProvider {
    String playAudiobook(String audiobook);
    String playPodcast(String podcast);
}      You, 1 second ago • Uncommitted changes
```

Bonus Task (Objective)

The objective of this task is to introduce the capability for users to adjust the privacy settings of their bookshelves. This enhancement empowers users to mark their bookshelves as either private or public.

Bonus Task
(Implementation using strategy pattern)

Our approach to access control within Sismics Books employs distinct strategies to manage private and public bookshelves. The PrivateAccessStrategy ensures exclusive access for creators to their private bookshelves. In contrast, the PublicAccessStrategy allows universal, read-only access to public bookshelves. This encapsulation not only enhances flexibility but also simplifies the process of extending the application in the future.

Design pattern

- Utilize the Strategy Pattern to encapsulate access control logic into separate classes (strategies).
- Strategies represent different approaches to access control (e.g., PrivateAccessStrategy, PublicAccessStrategy).
- Strategies can be selected at runtime based on context or configuration.

- **Example**:
  - `PrivateAccessStrategy`: Restricts access to private bookshelves to their creators.
  - `PublicAccessStrategy`: Allows read-only access to public bookshelves for all users.
- **Advantages**:
  - Encapsulation of access control rules enhances flexibility and extensibility of the application.
  - Simplifies maintenance by separating concerns and promoting a modular design.

Files changed

Process

- Tag.java
- TagResource.java
- TagDao.java

Front-end

- tag/html
- Tag.js

Functionality updates

```java
    /**
     * Private flag.
     */
    @Column(name = "TAG_PRIVATE_B", nullable = false)
    private boolean isPrivate;
```

```java
public Response update(
        @PathParam("id") String id,
        @FormParam("name") String name,
        @FormParam("color") String color,
        @FormParam("isPrivate") boolean isPrivate) throws JSONException {
```

```
                <label class="checkbox-inline">
                  <input type="checkbox" ng-model="tag.isPrivate"> Private
                </label>
              </td>
              <td class="col-md-1"><button class="btn btn-danger pull-right" ng-click="deleteTag(tag)"><spa
            </tr>
          </tbody>
```

Front end updates

Database update

Added the 'TAG_PRIVATE_B' column to the 'T_TAG' table in the database schema.

Ensured it is of type BOOLEAN and disallows null values.

Project Contributions

S. No.	Task Name	Implemented Design Pattern	Assignee Name	PR	PR Reviewed By	Contribution
1	Feature 1: Better user management	Builder Pattern	Vilal	Feature 1_PR-43	Shriom	Vilal 70%, Shriom 10%, Team 20%
2	Feature 2: Common Library_Part-1	##	Madan	Feature_2_PR_47	Vilal	Madan 60% Vilal 10% Team 30%
3	Feature 2: Common Library_Part-2	##		Feature_2_PR_47	Vilal	
4	Feature 2: Common Library_Part-3	Strategy Pattern		Feature_2_PR_48	Vilal	
5	Feature 2: Common Library_Part-4	##		Feature_2_PR_50	Vilal	
6	Feature 3: Online Integration	Strategy Pattern	Shriom	Feature_3_PR_45	Vilal	Shriom 60%, Vilal 10%, Team 30%
7	Bonus Feature: Public & Private Bookshelves	Strategy Pattern	Hanuma	Bonus_PR_46	Vilal	Hanuma 70%, Vilal 10%, Team 20%
8	Project Report Writing		Team	Doc_Report_PR_44	Vilal	Vilal 40%, Shriom 20%, Hanuma 20%, Madan 20%

thank you!

Team_11