

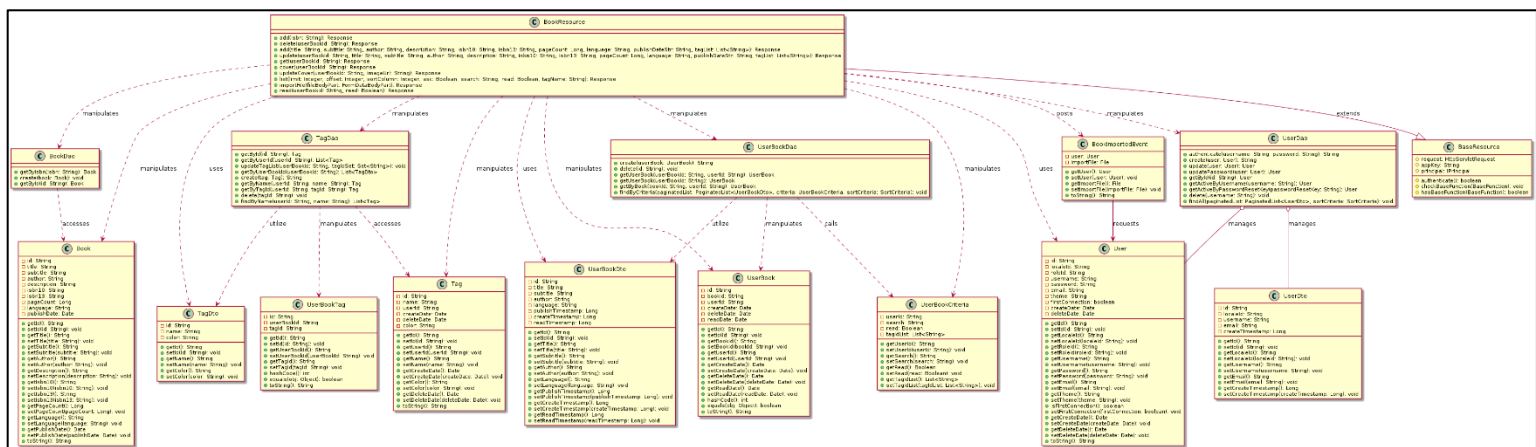
Project 1 Report

Task 1: Documenting the Given Code Base

(A) Identify the relevant classes and draw the UML diagram

Below we have documented all the relevant classes for the stated subsystems along with the UML diagram

Sub-system 1: Book Addition and Display System



- **BookResource :**

It is a class in the books-web package that deals with managing the books and contains the REST API calls in the application along with the implementation of the business logic in the same class. Following are the methods that are implemented in the class:

- **Response add(String isbn):** Creates a new book and user book record based on the provided ISBN number.
- **Response delete(String userBookId):** Deletes a book specified by the user book ID.
- **Response add(String title, String subtitle, String author, String description, String isbn10, String isbn13, Long pageCount, String language, String publishDateStr, List<String> tagList):** Adds a book manually with the provided details and creates a user book record.
- **Response update(String userBookId, String title, String subtitle, String author, String description, String isbn10, String isbn13, Long pageCount, String language, String publishDateStr, List<String> tagList):** Updates the details of an existing book specified by the user book ID.
- **Response get(String userBookId):** Retrieves the details of a book specified by the user book ID.
- **Response cover(String userBookId):** Returns the cover image of a book specified by the user book ID.

- **Response updateCover(String userBookId, String imageUrl):** Updates the cover image of a book specified by the user book ID.
- **Response list(Integer limit, Integer offset, Integer sortColumn, Boolean asc, String search, Boolean read, String tagName):** Returns a paginated list of books based on the provided criteria.
- **Response importFile(FormDataBodyPart fileBodyPart):** Imports books from a provided file and posts an event for importing the books.
- **Response read(String userBookId, boolean read):** Sets the read status of a book specified by the user book ID.

- **BookDao**

The BookDao class is responsible for database operations related to the Book entity. The member functions of the class are:

- **String create(Book book):** Creates a new book in the database and returns the new book's ID.
- **Book getById(String id):** Retrieves a book by its ID from the database. Returns the Book object if found, otherwise returns 'null'.
- **Book getByIsbn(String isbn):** Returns a book by its ISBN number (either ISBN-10 or ISBN-13) from the database. Returns the Book object if found, otherwise returns 'null'.

- **TagDao**

The TagDao class is responsible for performing database operations related to the Tag entity in a Java web application. It provides methods to create, retrieve, update, and delete tags, as well as to manage tags associated with user books.

- **UserDao**

The UserDao class is responsible for managing the persistence and retrieval of User entities in the application. It provides a set of methods for authenticating users, creating new user records, updating existing user details, retrieving users by various attributes, and deleting users.

- **TagDto**

The TagDto class is a Data Transfer Object (DTO) used to transfer data related to tags between different layers of the application. It encapsulates the basic information of a tag, which includes its 'id', 'name', and 'color'. This class is typically used to transfer tag data to and from the presentation layer without exposing the internal entity model or database structure. It contains the Getters and Setters for the attributes.

- **UserBookDto**

The UserBookDto class is used to transfer data related to a book and its interaction with a user. It encapsulates information about a user book, which contains 'id', 'title', 'subtitle', 'author', 'language', 'publishTimestamp', 'createTimestamp' and 'readTimestamp' attributes. It contains the Getters and Setters for the above mentioned attributes.

- **UserBookCriteria**

The `UserBookCriteria` class is used to encapsulate the criteria for querying user books in the application. It provides a way to specify various filters and search parameters that can be applied when retrieving user books from the database. It contains 'userId', 'search', 'read', and 'tagIdList' as its data members along with the Getters and setters for the above mentioned attribute.

- **BookImportedEvent**

The `BookImportedEvent` class represents an event that is raised when there is a request to import books into the application. It contains information about the user requesting the import and the temporary file containing the books to be imported in the form of 'user' and 'importFile' attributes. The class contains Getters and Setters methods along with a `toString` method.

- **Book**

The `Book` class is a Java Persistence API (JPA) entity that represents a book in the application's database. It is mapped to the 'T_BOOK' table, where each book is identified by a unique 'id' attribute. It includes attributes of a book, that are 'title', 'subtitle', 'author', 'description', ISBN numbers ('isbn10', 'isbn13'), 'pageCount', 'language', and 'publishDate'. Its methods contains the getters and setters for the above mentioned attributes.

- **Tag**

The `Tag` class is also a JPA entity that represents a tag for a book in the application's database. It is mapped to the 'T_TAG' table, where each tag is identified by a unique 'id' attribute. It includes attributes of a tag, that are 'name', 'userId', 'createDate', 'deleteDate', and 'color'. Its methods contains the getters and setters for the above mentioned attributes.

- **User**

The `User` class is also a JPA entity that represents a user of the application. It is mapped to the 'T_USER' table, where each user is identified by a unique 'id' attribute. It includes attributes of a user that are 'localeId', 'roleId', 'username', 'password', 'email', some Application related information ('theme') and some Session Information ('firstConnection', 'createDate', 'deleteDate'). Its methods contains the getters and setters for the above mentioned attributes.

- **UserBook**

The `UserBook` class is also a JPA entity that represents book information corresponding to a user of the application. It is mapped to the 'T_USER_BOOK' table, which includes information about the user, the book, and various dates related to the user's interaction with the book. The attributes used in the class are 'bookId', 'userId', 'createDate', 'deleteDate' and 'readDate' along with the Getters and Setters for the same. Other methods that customized by the class are:

- **int hashCode():** This method helps in generating a unique hash code from the bookId and userId for uniquely storing a book data for a particular user.
- **boolean equals(Object obj):** This method helps to check if the 'bookId' and 'userId' of two objects are same or not.

- **BaseResource**

The `BaseResource` class serves as an abstract base class for REST resources in the application. It provides common functionality and fields used by derived resource classes, such as authentication and authorization. It has following data member and member functions:

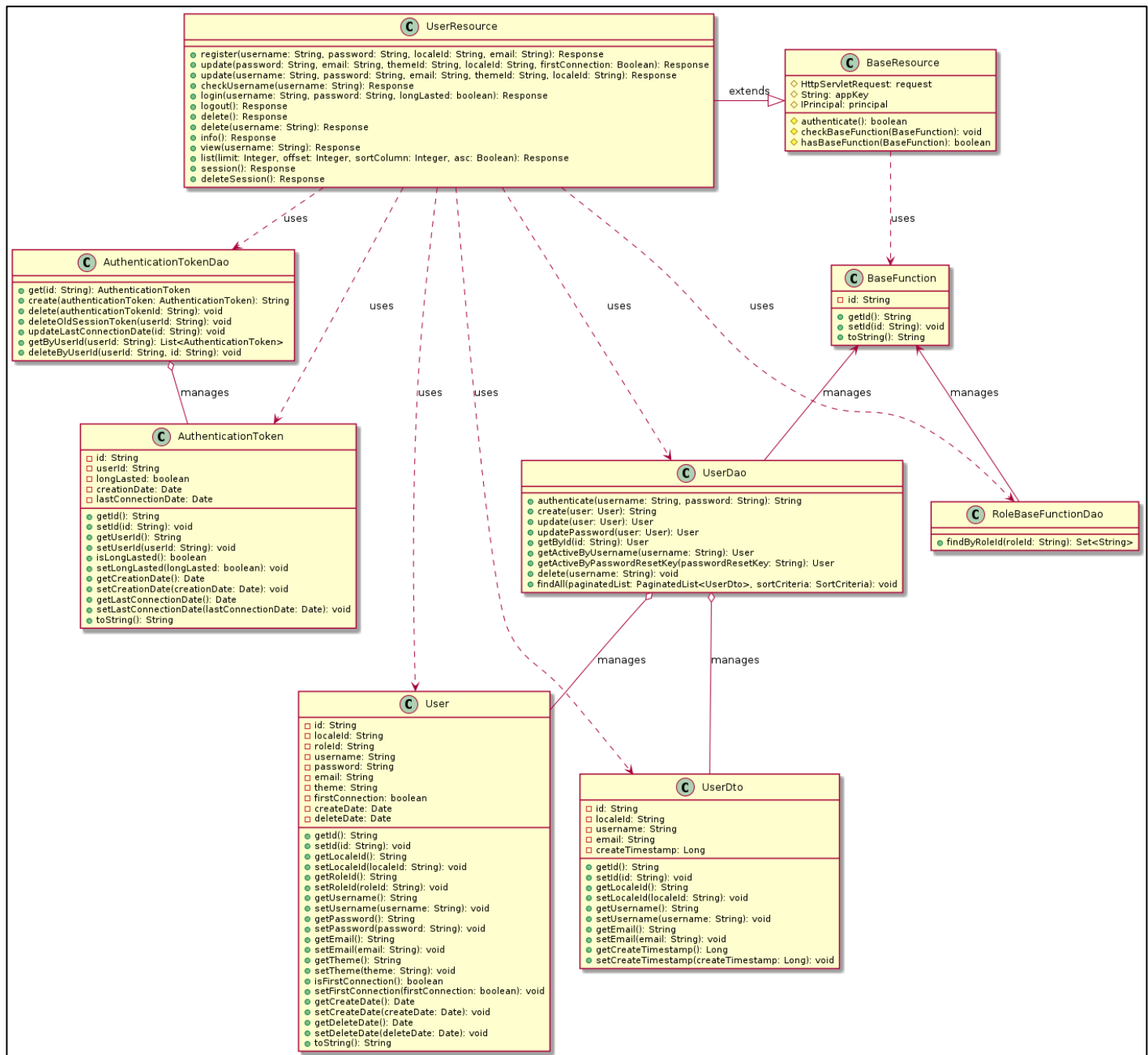
- **request (HttpServletRequest):** The HTTP request object, allows access to request-specific information such as headers, parameters, and the request body.
- **appKey (String):** The application key obtained from the query parameter app_key, used to authenticate requests from specific applications or clients.
- **principal (IPrincipal):** The principal of the authenticated user, representing their identity and security context.
- **boolean authenticate():** Checks for user authentication. It retrieves the principal from the request attribute and verifies that the principal is not anonymous. Returns true if user is authenticated, otherwise false.
- **void checkBaseFunction(BaseFunction baseFunction):** Checks if the authenticated user has the base function, if not the it throws a ForbiddenClientException. It doesn't return any output, as it uses the hasBaseFunction() method defined below.
- **boolean hasBaseFunction(BaseFunction baseFunction):** Checks if the authenticated user has the specified base function. It retrieves the set of base functions from the UserPrincipal and checks if the specified base function is present. Returns true if the user has the base function, otherwise false.
- **UserBookTag**

The UserBookTag class is a JPA entity that represents the relation between a book and a tag. It is mapped to the 'T_USER_BOOK_TAG' table, where each book and tag relation is identified by a unique 'id' attribute. It includes attributes of book and tag, that are 'userBookId', 'tagId'. Its methods contains the getters and setters for the above mentioned attributes along with the customization to hashCode and equals function (same as in UserBook Class).

- **UserDto**

The UserDto class is used to transfer data related to a user. It encapsulates information about a user, which contains 'id', 'localeId', 'username', 'email', and 'createTimestamp' attributes. It contains the Getters and Setters for the above mentioned attributes.

Sub-system 2: User Management Subsystem



• UserResource

It is a class in the books-web package that deals with managing the user of the application and contains the REST API calls for the same along with implementation of the business logic in the same class. Following are the methods that are implemented in the class:

- **Response register(String username, String password, String localeId, String email):** Registers a new user provided with the mentioned values in RequestBody. It creates a user and gives a response as “ok” if the validation check is completed or throws an Exception at backend.

- **Response update(String password, String email, String themeld, String localeId, Boolean firstConnection):** It updates the user detail (if the value is provided by the user) if the user is authenticated the data provided by the user meets the validation checks. It returns “ok” as response otherwise throws an Exception.
- **Response update(String username, String password, String email, String themeld, String localeId):** It takes username as input in the path variable of the API request and used to update the information based on the user’s username instead.
- **Response checkUsername(String username):** Performs check whether the provided username exists for the currently active users of the application. It returns “username already registered” if an active user with the same username is found.
- **Response login(String username, String password, boolean longLasted):** It used to authenticate (if user provides with correct credentials) the user and create a user session. The session is only used to identify the user and doesn’t store any other information.
- **Response logout():** It logs the user out of the application and clear the session data for the user.
- **Response delete():** It deletes the currently logged in user, but also checks that the ‘ADMIN’ user is not deleted.
- **Response delete(String username):** It helps in deleting a user based on the username provided. It checks whether you are an ‘ADMIN’ or not and then deletes the user’s data if it exists.
- **Response info():** It returns the information for the currently active user. The response varies as per the role of user, whether it is ‘ADMIN’ or any other role.
- **Response view(String username):** It returns the information for a user based on the provided ‘username’. It returns ‘username’, ‘email’, ‘theme’, ‘locale’ information about the user.
- **Response list(Integer limit, Integer offset, Integer sortColumn, Boolean asc):** It returns the list of all the currently active users which contains the information about user’s ‘id’, ‘email’, ‘username’, ‘create_date’ that too in a paginated form. The requesting user can sort the output based on a column in either the ascending or descending format (by default it is sorted in ascending format of ‘id’).
- **Response session():** It returns all the active session for the requesting user and also distinctively reports the “current” active session information.
- **Response deleteSession():** It deletes all the currently active sessions for the user who has requested for this API call. It returns status “ok” on success otherwise throws an exception at the backend.

- **AuthenticationTokenDao**

The AuthenticationTokenDao class is responsible for managing authentication tokens in a Java web application. It provides methods for creating, retrieving, updating, and deleting authentication tokens. There are following methods in the class:

- **get(String id):** Retrieves an authentication token by its ID.
- **create(AuthenticationToken authenticationToken):** Creates and persists a new authentication token in the database.
- **delete(String authenticationTokenId):** Deletes an authentication token by its ID.
- **deleteOldSessionToken(String userId):** Deletes old short-lived tokens for a specified user ID.
- **updateLastConnectionDate(String id):** Updates the last connection date of a specified authentication token.

- **getByUserId(String userId):** Retrieves all authentication tokens for a specified user ID.
- **deleteByUserId(String userId, String id):** Deletes all authentication tokens for a specified user ID, except for a specified token ID.
- **RoleBaseFunctionDao**

This class manages the relationship between roles and base functions in a Java web application, providing methods to retrieve the set of base functions associated with a specific role. The member functions description are:

- **Set<String> findByRoleId(String roleId):** It retrieves a set of base function IDs associated with a given role ID in a Java web application, ensuring that only functions linked to active roles are included.
- **UserDao**

The UserDao class is responsible for managing the persistence and retrieval of User entities in the application. It provides a set of methods for authenticating users, creating new user records, updating existing user details, retrieving users by various attributes, and deleting users. Method description is as follows:

- **String authenticate(String username, String password):** Authenticates a user by checking if the provided username and password match an existing user in the database.
- **String create(User user):** Creates a new user in the database with a unique UUID and checks for username uniqueness.
- **User update(User user):** Updates the locale, email, theme, and first connection status of an existing user in the database.
- **User updatePassword(User user):** Updates the password of an existing user in the database.
- **User getById(String id):** Retrieves a user by their ID from the database.
- **User getActiveByUsername(String username):** Retrieves an active user by their username from the database.
- **User getActiveByPasswordResetKey(String passwordResetKey):** Retrieves an active user by their password recovery token from the database.
- **void delete(String username):** Deletes a user by their username from the database and cleans up linked data such as authentication tokens.
- **String hashPassword(String password):** Hashes a user's password using BCrypt for secure storage in the database.
- **void findAll(PaginatedList<UserDto> paginatedList, SortCriteria sortCriteria):** Retrieves a paginated list of all users from the database, applying sorting criteria and updating the provided PaginatedList object with the results.

- **UserDto**

The UserDto class is used to transfer data related to a user. It encapsulates information about a user, which contains 'id', 'localeId', 'username', 'email', and 'createTimestamp' attributes. It contains the Getters and Setters for the above mentioned attributes.

- **AuthenticationToken**

The AuthenticationToken class is a JPA entity that represents an authentication token entity in a Java web application. It is mapped to the 'T_AUTHENTICATION_TOKEN' table. This class, storing information about the token, the associated user, and the token's longevity and usage timestamps.

It includes the attributes 'id', 'userId', 'longLasted', 'creationDate', 'lastConnectionDate'. It contains the Getters and Setters for the above mentioned attributes.

- **User**

The User class is also a JPA entity that represents a user of the application. It is mapped to the 'T_USER' table, where each user is identified by a unique 'id' attribute. It includes attributes of a user that are 'localeId', 'roleId', 'username', 'password', 'email', some Application related information ('theme') and some Session Information ('firstConnection', 'createDate', 'deleteDate'). Its methods contain the getters and setters for the above mentioned attributes.

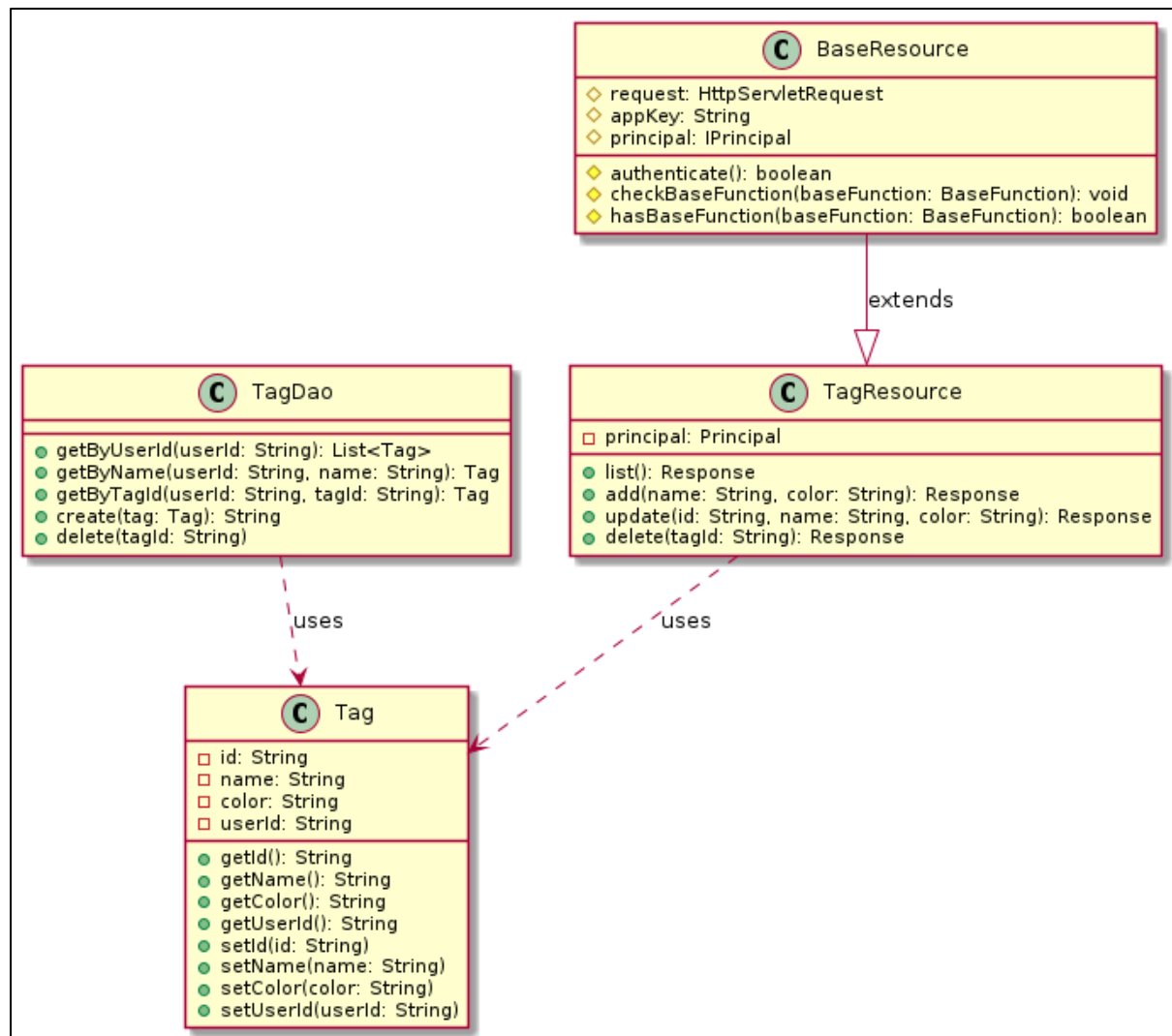
- **BaseFunction**

The BaseFunction class is also a JPA entity that represents a base function entity in a Java web application, which is used to define fundamental permissions or roles within the system, such as "ADMIN". It is mapped to the 'T_BASE_FUNCTION' table, where each base function is identified by a unique 'id' attribute. Its methods contain the getters and setters for the above mentioned attributes.

- **BaseResource**

The BaseResource class serves as an abstract base class for REST resources in the application. It provides common functionality and fields used by derived resource classes, such as authentication and authorization. The member functions for the BaseResource Class are mentioned above in detail.

Sub-system 3: Bookshelf Management Subsystem



- **Tag**

The Tag class is also a JPA entity that represents a tag for a book in the application's database. It is mapped to the 'T_TAG' table, where each tag is identified by a unique 'id' attribute. It includes attributes of a tag, that are 'name', 'userId', 'createDate', 'deleteDate', and 'color'. Its methods contains the getters and setters for the above mentioned attributes.

- **TagDao**

This class manages tag entities in a Java web application, providing methods to create, retrieve, update, and delete tags, as well as manage tag associations with user books. The methods that are used in the class are:

- **Tag getById(String id):** Retrieves a tag by its ID.
- **List<Tag> getByUserId(String userId):** Returns a list of all tags associated with a given user ID.

- **void updateTagList(String userBookId, Set<String> tagIdSet):** Updates the list of tags associated with a user book.
- **List<TagDto> getByUserBookId(String userBookId):** Returns a list of tag DTOs associated with a user book.
- **String create(Tag tag):** Creates a new tag and returns its ID.
- **Tag getByName(String userId, String name):** Retrieves a tag by its name and user ID.
- **Tag getByTagId(String userId, String tagId):** Retrieves a tag by its ID and user ID.
- **void delete(String tagId):** Deletes a tag by its ID.
- **List<Tag> findByName(String userId, String name):** Searches for tags by name and user ID, returning a list of matching tags.
- **TagResource**

The TagResource class is responsible for handling RESTful requests related to tag management in a Java web application. It provides endpoints for listing, creating, updating, and deleting tags, ensuring that only authenticated users can perform these actions. The method description is given as follows:

- **Response list():** Returns the list of all tags associated with the authenticated user in JSON format.
- **Response add(String name, String color):** Creates a new tag with the specified name and color for the authenticated user and returns the created tag's ID in JSON format.
- **Response update(String id, String name, String color):** Updates the tag with the specified ID, name, and color for the authenticated user and returns the updated tag's ID in JSON format.
- **Response delete(String tagId):** Deletes the tag with the specified ID for the authenticated user and returns a status message in JSON format.
- **BaseResource**

The BaseResource class serves as an abstract base class for REST resources in the application. It provides common functionality and fields used by derived resource classes, such as authentication and authorization. The member functions for the BaseResource Class are mentioned above in detailed.

(C) Observations and Comments

Strength:

- Classes are Pretty Organized: The different classes have their own jobs, which is neat. We've got classes for talking to the database (BookDao, TagDao, UserDao, AuthenticationDao, UserBookDao, etc.), classes for moving data around (UserDto, UserBookDto, TagDto, etc), and a class for handling web stuff (BookResource, TagResource, UserResource, etc.).
- Using JPA Makes Things Easier: We are using JPA for mapping Java objects to the database. It makes working with databases in Java more straightforward and less of a hassle.

Weaknesses:

- Mixing Business Logic: The classes such as BookResource, TagResource, UserResource seems to be doing a bit too much. It's trying to handle both web stuff and business logic for managing books. It would be better if it focused just on the web stuff and let other classes handle the business logic.
- Could Get Tangled Up: We are a bit worried that the classes might be too tightly connected to each other. It's important to keep things loose so that if you want to change one part of the system, you don't accidentally break something else.
- Need a Safety Net: I didn't see much about how the system deals with errors. It's important to have a plan for when things go wrong so that the system doesn't crash and burn when something unexpected happens.

(D) Assumptions

- **User Registration and Login**: People can sign up for the app by giving their username, password, email, and language preference. When someone signs up, the app checks if everything is okay and then makes a new account for them. To log in, users use their username and password. If they get it right, they can use the app.
- **User Profile Management**: Users can change stuff about their profile, like their password, email, or how the app looks. Administrators can do more, like see all the users or delete someone's account.
- **Book Handling**: You can add new books manually by typing in details like the title, author, and a description. Also, there's a way to add lots of books at once by uploading a file. You can update info about a book or say if you've read it or not. You can search for books based on different things like their title, whether you've read them, or tags you've added.
- **Tagging Books**: You can add tags to books to organize them. Tags are like labels you stick on books. Tags help you find books later based on what they're about or how you've grouped them.
- **Database Stuff**: The app keeps track of users, books, tags, and other things in a big digital filing system called a database. Different parts of the app know how to talk to this filing system to find or save information.
- **Using the App**: You interact with the app through a website or mobile app, and it talks to the database in the background. Then you do something like add a book or change your profile, the app sends a message to the database to make it happen.

- **Keeping Things Safe:** The app makes sure your information stays safe. For example, it doesn't show your password to anyone, and it uses tricks to keep it secret. It also checks if you're allowed to do things. Like, you can't delete someone else's account unless you're an admin.
- **Communicating Between Parts:** The different parts of the app talk to each other using special messages called events. For example, when you upload a file of books, an event tells the app to add them to your collection.

Task 2(a): Running Sonarqube & Identifying Design Smells

The detected code smells when we ran Sonarqube on the books-web module in the given codebase are given in the following screenshots

This screenshot displays the SonarQube interface for two Java files: `AppResource.java` and `BaseResource.java`. The interface lists detected code smells with their severity, location, and a brief description.

- AppResource.java**
 - Remove this unnecessary null check; "instanceOf" returns false for nulls.** (Minor, 11 days ago, L130, 5min effort, Comment, redundant)
- BaseResource.java**
 - Rename "principal" which hides the field declared at line 37.** (Major, 11 days ago, L45, 5min effort, Comment, cert, pitfall, suspicious)
 - Remove this unnecessary null check; "instanceOf" returns false for nulls.** (Minor, 11 days ago, L46, 5min effort, Comment, redundant)
 - Remove the declaration of thrown exception 'org.codehaus.jackson.JsonException', as it cannot be thrown from method's body.** (Minor, 11 days ago, L73, 5min effort, Comment, clumsy, error-handling, redundant, unu...)
 - Remove this unnecessary null check; "instanceOf" returns false for nulls.** (Minor, 11 days ago, L74, 5min effort, Comment, redundant)

This screenshot displays the SonarQube interface for three Java files: `ConnectResource.java`, `TagResource.java`, and `UserResource.java`. The interface lists detected code smells with their severity, location, and a brief description.

- ConnectResource.java**
 - Replace the type specification in this constructor call with the diamond operator ("<>").** (Minor, 11 days ago, L64, 1min effort, Comment, clumsy)
 - Replace this "switch" statement by "if" statements to increase readability.** (Minor, 11 days ago, L105, 5min effort, Comment, bad-practice)
 - Define a constant instead of duplicating this literal "status" 3 times.** (Critical, 11 days ago, L148, 8min effort, Comment, design)
 - Replace the type specification in this constructor call with the diamond operator ("<>").** (Minor, 11 days ago, L251, 1min effort, Comment, clumsy)
- TagResource.java**
 - Define a constant instead of duplicating this literal "color" 3 times.** (Critical, 11 days ago, L57, 8min effort, Comment, design)
- UserResource.java**
 - Define a constant instead of duplicating this literal "username" 5 times.** (Critical, 11 days ago, L76, 12min effort, Comment, design)
 - Define a constant instead of duplicating this literal "password" 3 times.** (Critical, 11 days ago, L78, 8min effort, Comment, design)
 - Define a constant instead of duplicating this literal "email" 7 times.** (Critical, 11 days ago, L79, 16min effort, Comment, design)

This screenshot displays the SonarQube interface for three files: `book_add.html`, `tag.html`, and `TestBookResource.java`. The interface lists detected code smells with their severity, location, and a brief description.

- book_add.html**
 - Add an "aria-label" or "aria-labelledby" attribute to this element.** (Major, 11 days ago, L27, 5min effort, Comment, accessibility, wcag2-a)
 - Add an "aria-label" or "aria-labelledby" attribute to this element.** (Major, 11 days ago, L49, 5min effort, Comment, accessibility, wcag2-a)
- tag.html**
 - Add an "aria-label" or "aria-labelledby" attribute to this element.** (Major, 11 days ago, L3, 5min effort, Comment, accessibility, wcag2-a)
 - Add an "aria-label" or "aria-labelledby" attribute to this element.** (Major, 11 days ago, L16, 5min effort, Comment, accessibility, wcag2-a)
 - Remove this commented out code.** (Major, 11 days ago, L36, 5min effort, Comment, unused)
 - Remove this commented out code.** (Major, 11 days ago, L42, 5min effort, Comment, unused)
- TestBookResource.java**
 - Refactor this method to reduce the number of assertions from 35 to less than 25.** (Major, 11 days ago, L35, 20min effort, Comment, tests)
 - Use assertEquals instead.** (Major, 11 days ago, L190, 2min effort, Comment, junit, tests)
 - Use assertEquals instead.** (Major, 11 days ago, L203, 2min effort, Comment, junit, tests)

<div><div><div></div><div></div><div></div></div><div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>12min effort</div><div>Comment</div></div></div>	<div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "password" 3 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>8min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L78</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "email" 7 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>16min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L79</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "status" 8 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>18min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L105</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "locale" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L136</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "theme" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L137</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "The user doesn't exist" 3 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>8min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L204</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "UserNotFound" 3 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>8min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L204</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>src/main/webapp/portal/book-add.html</div><div>Add an "aria-label" or "aria-labelledby" attribute to this element.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>5min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L27</div> <div>accessibility, wcag2-a</div>
<div><div><div></div><div></div><div></div></div><div><div>Add an "aria-label" or "aria-labelledby" attribute to this element.</div></div></div>	<div>11 days ago</div> <div>L49</div> <div>design</div>

<div><div><div></div><div></div><div></div></div><div><div>Remove this commented out code.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>5min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L42</div> <div>unused</div>
<div><div><div></div><div></div><div></div></div><div><div>src/.../java/com/simics/books/rest/TestBookResource.java</div><div>Refactor this method to reduce the number of assertions from 35 to less than 25.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>20min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L35</div> <div>tests</div>
<div><div><div></div><div></div><div></div></div><div><div>Use assertEquals instead.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>2min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L190</div> <div>junit, tests</div>
<div><div><div></div><div></div><div></div></div><div><div>Use assertEquals instead.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>2min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L203</div> <div>junit, tests</div>
<div><div><div></div><div></div><div></div></div><div><div>Use assertEquals instead.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>2min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L219</div> <div>junit, tests</div>
<div><div><div></div><div></div><div></div></div><div><div>src/.../java/com/simics/books/rest/TestConnectResource.java</div><div>Refactor this method to reduce the number of assertions from 35 to less than 25.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>20min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L58</div> <div>tests</div>
<div><div><div></div><div></div><div></div></div><div><div>src/.../java/com/simics/books/rest/TestTagResource.java</div><div>Use assertEquals instead.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>2min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L133</div> <div>junit, tests</div>
<div><div><div></div><div></div><div></div></div><div><div>src/.../java/com/simics/books/rest/TestUserResource.java</div><div>Refactor this method to reduce the number of assertions from 41 to less than 25.</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>20min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L29</div> <div>tests</div>

48 of 48 shown

<div><div><div></div><div></div><div></div></div><div><div>src/_simics/books/rest/resource/BookResource.java</div><div>4 duplicated blocks of code must be removed. RULE DEPRECATED</div><div>Code Smell</div><div>Major</div><div>Open</div><div>Not assigned</div><div>50min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>pitfall</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "BookNotFound" 5 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>12min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L94</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "BookAlreadyAdded" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L111</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "Book already added" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L111</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "Book not found with id" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L139</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "status" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L147</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Refactor this method to reduce its Cognitive Complexity from 22 to the 15 allowed.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>12min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L162</div> <div>brain-overload</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "title" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L178</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "subtitle" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L179</div> <div>design</div>

<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "status" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L147</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Refactor this method to reduce its Cognitive Complexity from 22 to the 15 allowed.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>12min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L162</div> <div>brain-overload</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "title" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L178</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "subtitle" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L179</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "author" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L180</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "description" 3 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>8min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L181</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "isbn10" 3 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>8min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L182</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "isbn13" 3 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>8min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L183</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "language" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L184</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Define a constant instead of duplicating this literal "publish_date" 4 times.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>10min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L185</div> <div>design</div>
<div><div><div></div><div></div><div></div></div><div><div>Refactor this method to reduce its Cognitive Complexity from 27 to the 15 allowed.</div><div>Code Smell</div><div>Critical</div><div>Open</div><div>Not assigned</div><div>17min effort</div><div>Comment</div></div></div>	<div>11 days ago</div> <div>L276</div> <div>brain-overload</div>

Following are the design smells that are observed and are defined as follows:

1. **SRP Violated**: The classes like TagResource, UserResource, ConnectResource, BookResource are handling HTTP Request Response as well as the implementation of Business Logic which violates the Single Responsibility Principle.
2. **God Class**: BookResource, TagResource and UserResource classes exhibits God Class behaviour because they have too many methods that increases the Dependency of other classes on them that causes High Coupling and issues in Maintainability and Readability.
3. **Missing Abstraction**: BookResource, TagResource, and UserResource which are to deal with only the handling of REST API calls are also exposed to the implementation details too, this can be reported as instance of Abstraction Smell.
4. **Insufficient Modularization**: BookResource, TagResource, and UserResource suffers from this smell since it has too many long functions and also it deals with the DAOs (for the respective classes) directly which creates strong coupling between your API and the JPA resource.
5. **Spaghetti Code**: The classes like BookResource, TagResource, etc. are complex and hard to read, which makes them difficult to maintain. Their intricate structures can pose challenges for understanding and modifying the code, leading to potential issues in long-term manageability.

Task 2(b): Comprehensive Code Metric Analysis

Tools used: CodeMR

Code Metrics: Complexity, Coupling, Lack of Cohesion, Size

Explanation of Metrics:

1. Complexity:

Complexity gives a measure of how hard is the code to understand and explains how entities interact with one another. Higher levels of complexity in the code increases risk of unintentionally interfering with interactions and increases the chance of introducing defects while making changes in future.

2. Coupling :

Coupling gives a measure of the extent to which classes are coupled with each other. It is used to identify whether the system is tightly coupled or loosely coupled.

Characteristics of tightly coupled systems:

- A change in a class usually forces a ripple effect of changes in other classes.
- Require more effort and/or time due to the increased dependency.
- Might be harder to reuse a class because dependent classes must be included.

3. Lack of Cohesion :

Measure how well the methods of a class are related to each other. High cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

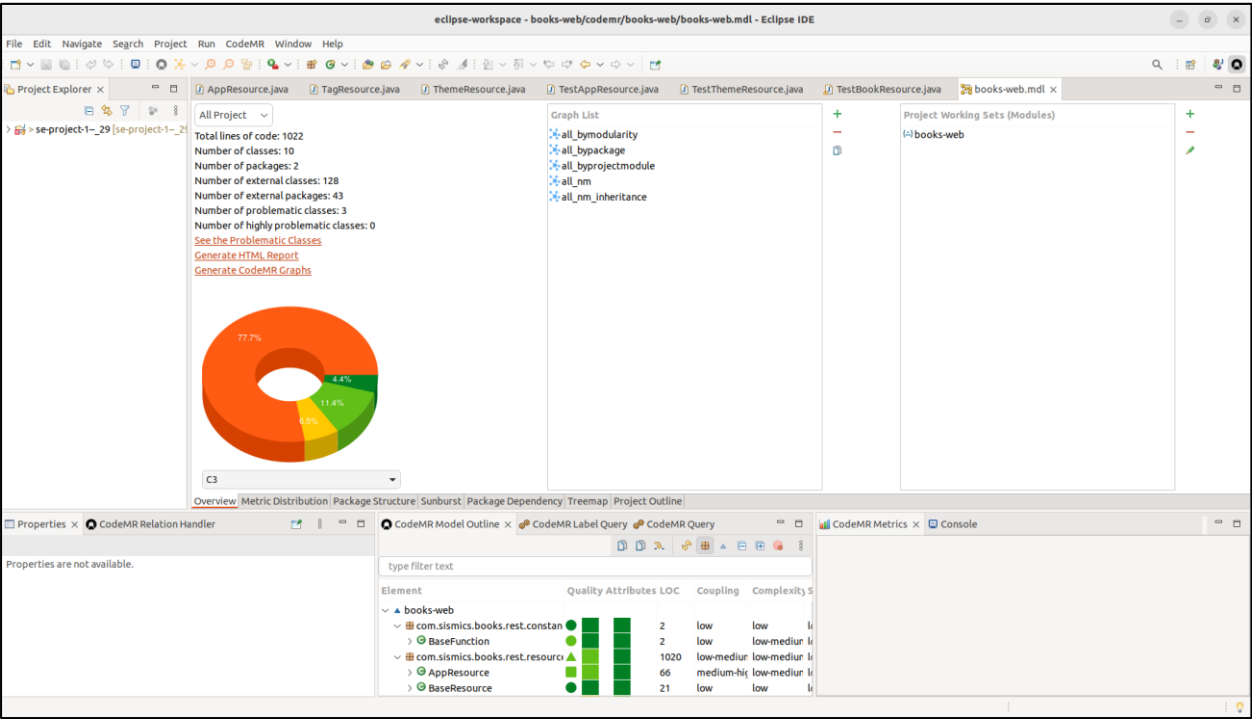
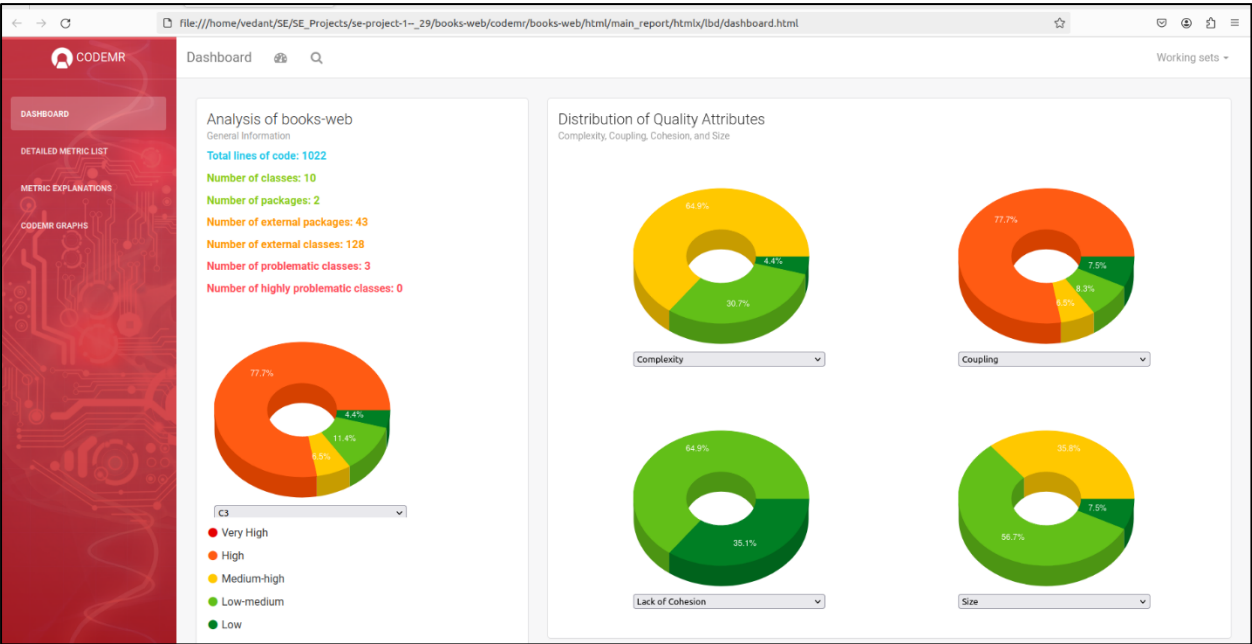
4. Size :

It gives a measure of the number of lines or methods in the code. A very high count might indicate that a class or method is trying to do too much work and should be split up. It might also indicate that the class might be hard to maintain.

5. C3 :

It is maximum value of Coupling, Cohesion, and Complexity metrics.

Screenshots :



Insights related to current project:

1. Complexity :

There are two classes with medium-high complexity, six classes with low-medium complexity and two classes with low complexity.

1) Classes having medium-high complexity (64.9%)

1.1) BookResources

1.2) UserResources

2) Classes having low-medium complexity (30.7%)

2.1) AppResources

2.2) TagResources

2.3) ConnectResources

2.4) LocaleResources

2.5) ThemeResources

2.6) Resources

3) Classes having low complexity (4.4%)

3.1) BaseResources

3.2) TextPlainMessageBodyWriter

The 'BookResource' class has a high complexity, which might indicate that it's doing too much or has too many responsibilities. This could make it more difficult to maintain or could increase the risk of defects.

2. Coupling:

There are two classes with medium-high complexity, six classes with low-medium complexity and two classes with low complexity.

1) Classes having high coupling (77.7%)

1.1) BookResources

1.2) UserResources

1.3) ConnectResources

2) Classes having medium-high coupling (6.5%)

2.1) AppResources

3) Classes having low-medium coupling (6.5%)

3.1) TagResources

3.2) Resources

4) Classes having low coupling (7.5%)

- 4.1) LocaleResources
- 4.2) ThemeResources
- 4.3) BaseResources
- 4.4) TextPlainMessageBodyWriter

The 'BaseResource' class has a high level of coupling, suggesting it might be too dependent on other classes or components. This can affect the modularity of the code, making changes more ripple-prone and potentially impacting reusability.

3. Lack of Cohesion:

There are 8 classes with low lack of Cohesion and 2 classes with low-medium lack of Cohesion.

1) Classes having low medium lack of cohesion (35.1%)

- 1.1) BookResources
- 1.2) UserResources

2) Classes having low Lack of cohesion (64.9%)

- 2.1) ConnectResources
- 2.2) AppResources
- 2.3) TagResources
- 2.4) TextPlainMessageBodyWriter
- 2.5) ThemeResources
- 2.6) LocaleResources

Several classes like 'BookResource', 'BaseResource', and 'LocaleResource' exhibit low to medium cohesion. This suggests that the classes may be grouping unrelated functionalities, which can be a sign that they should be refactored into more focused, cohesive units.

4. Size:

There are four classes with low-medium size, five classes with low size and only 1 class with medium-high size.

1) Classes having low -medium size (56.7%)

- 1.1) ConnectResources
- 1.2) UserResources
- 1.3) TagResources
- 1.4) AppResources

2) Classes having low (7.5%)

- 2.1) ThemeResources

2.2)LocaleResources

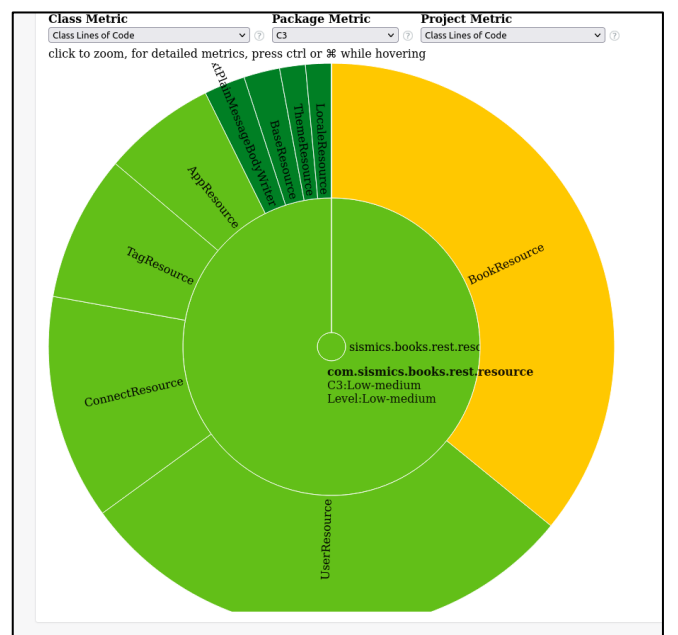
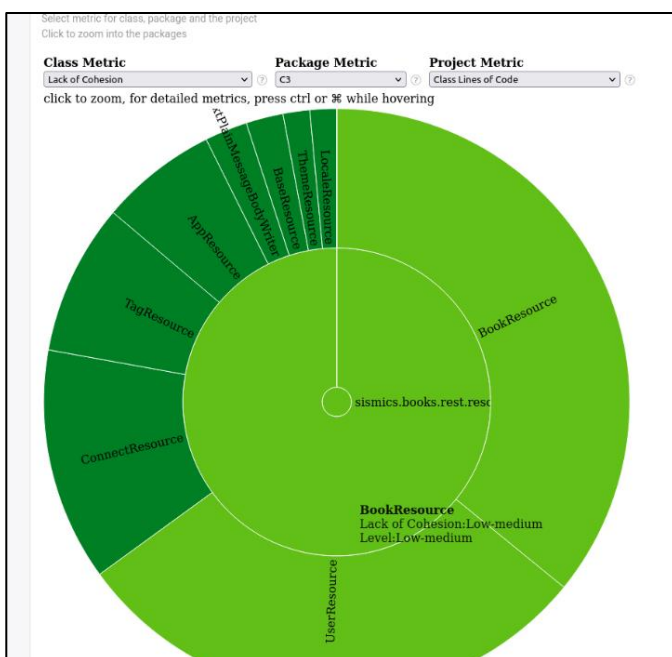
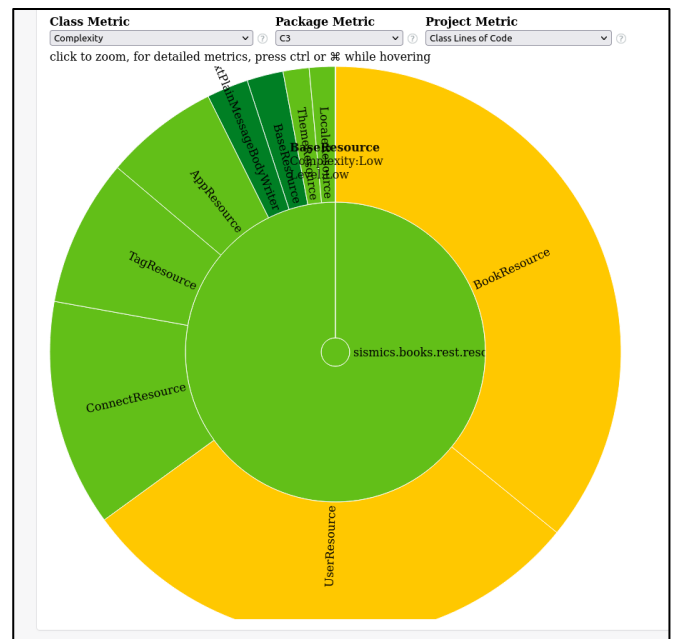
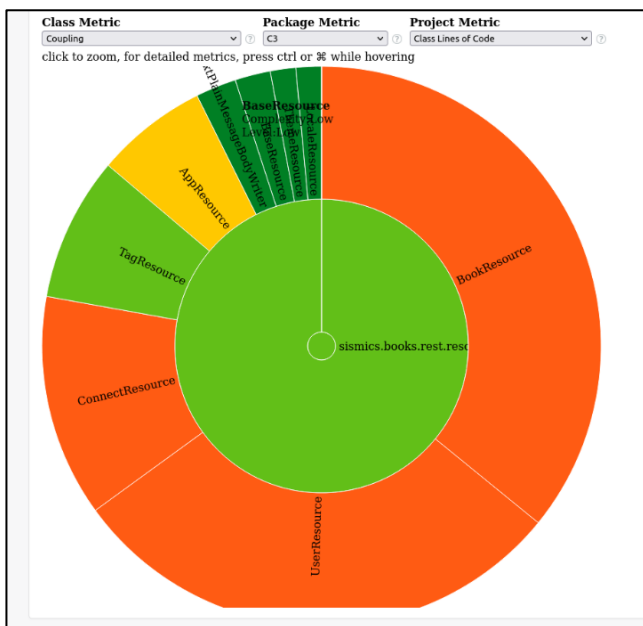
2.3) BookResources

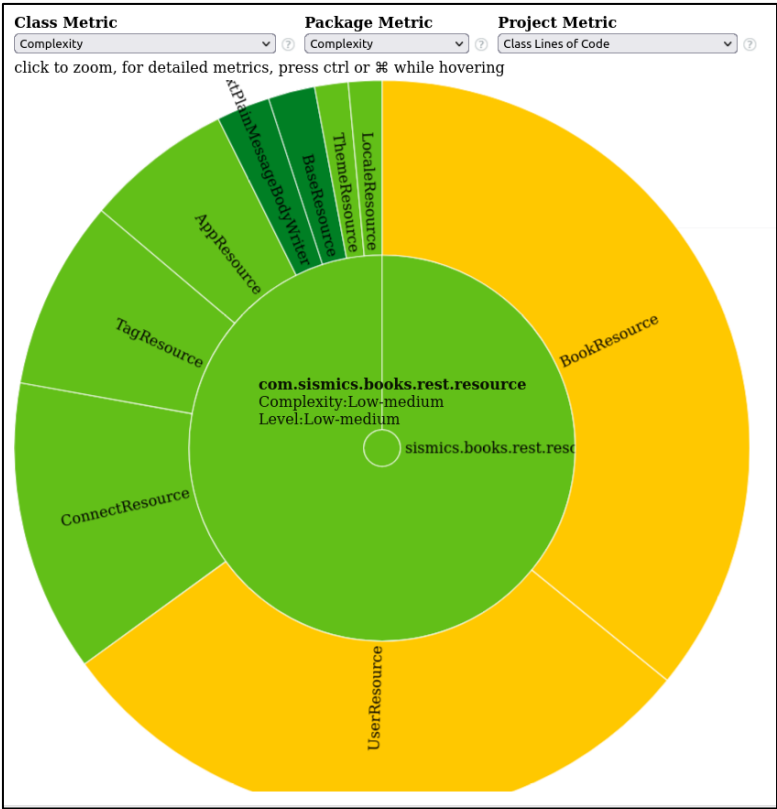
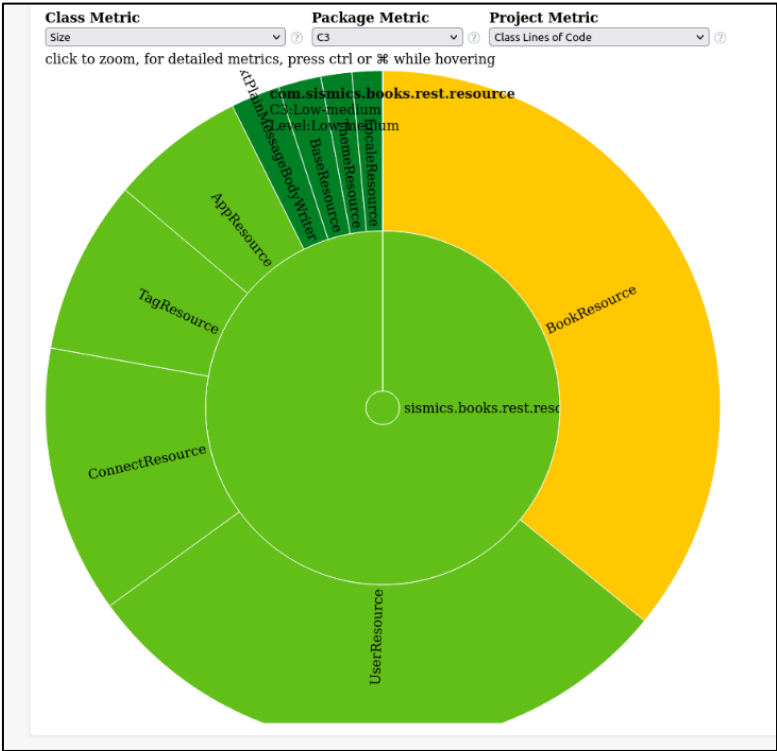
2.4)TextPlainMessageBodyWriter

3) Classes having medium-high size (35.8%)

3.1)BookResources

The 'BookResource' class also stands out in terms of size, which correlates with its high complexity. Large classes can be more challenging to understand, test, and maintain.

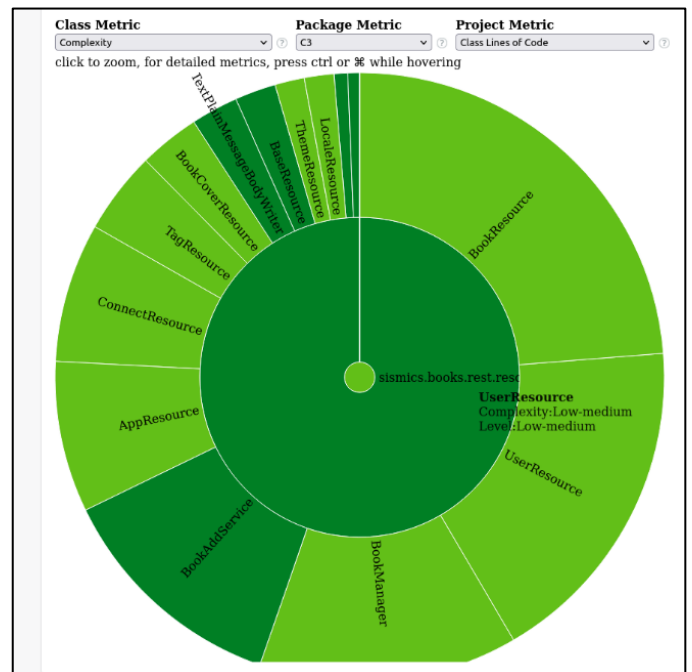
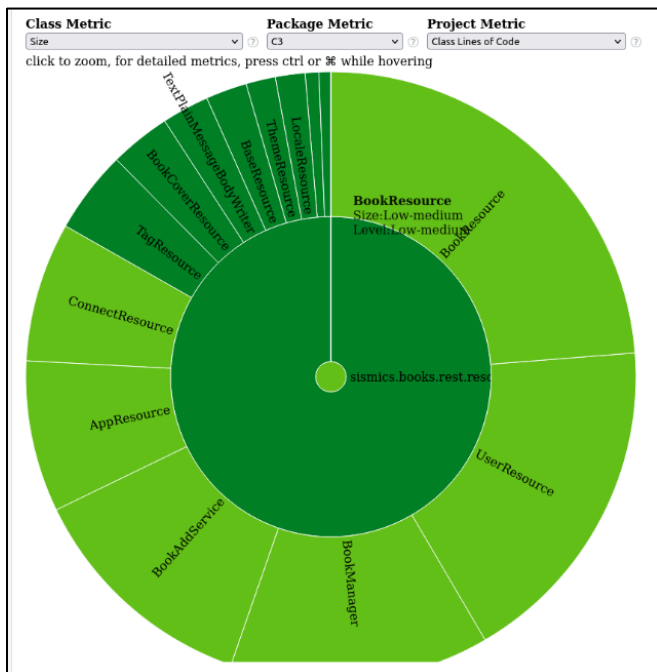
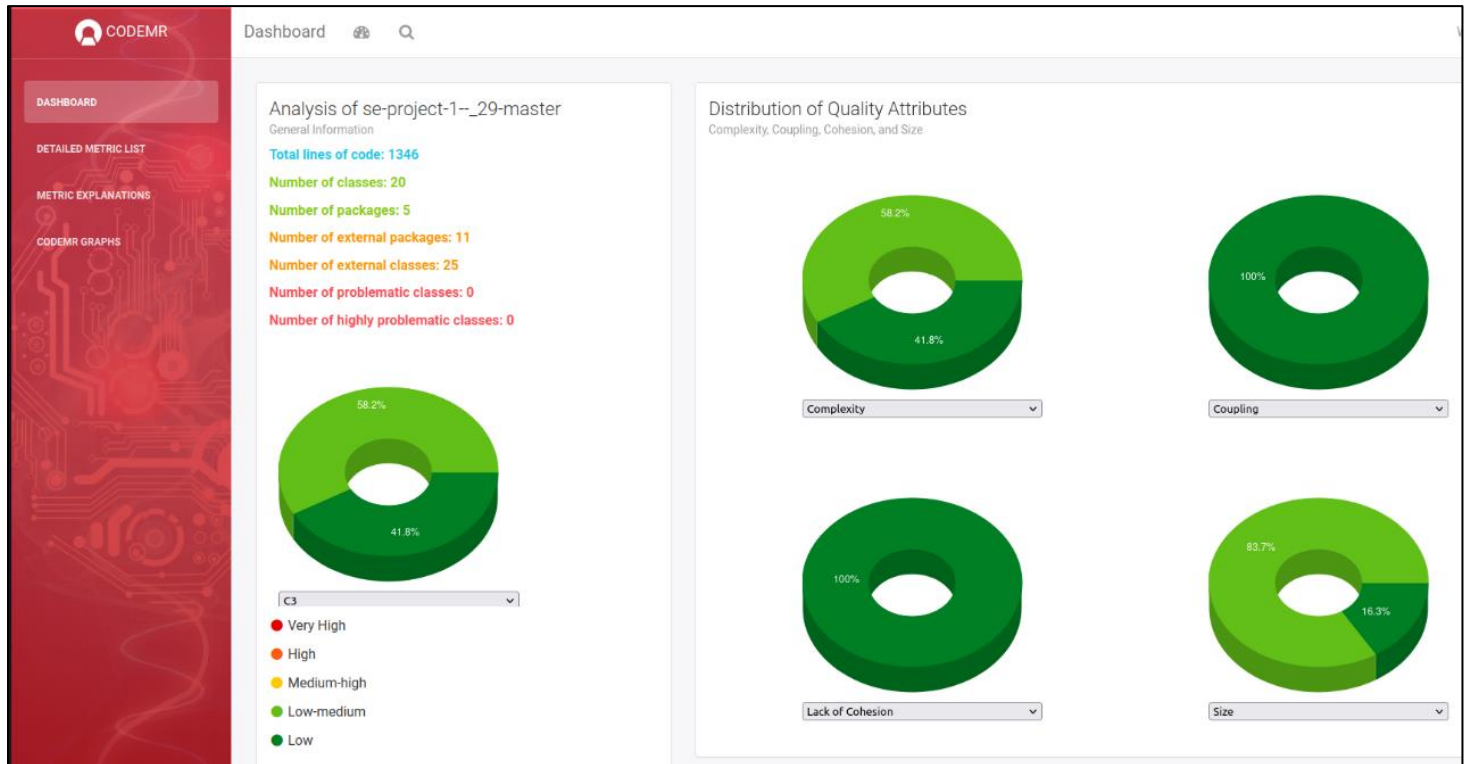


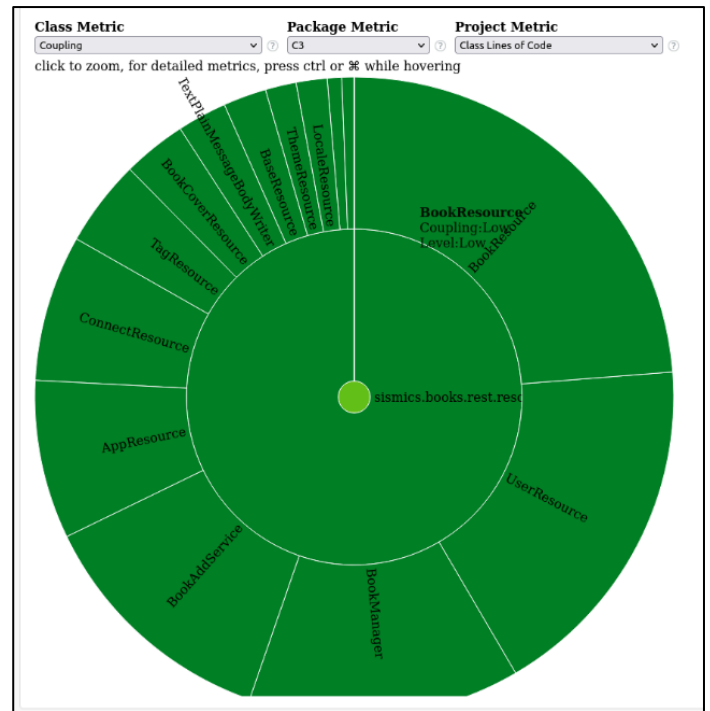
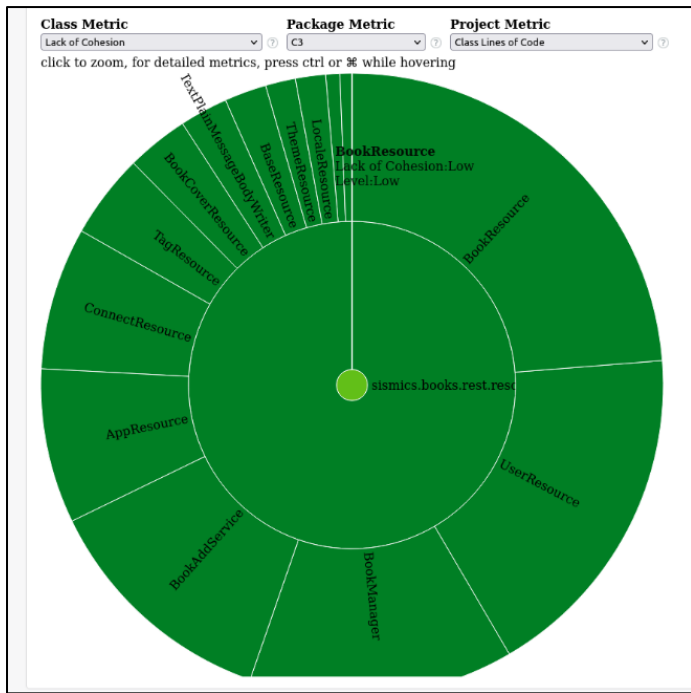


Task 3(b):

The analysis metrics that were calculated in Task 2(b) were recalculated on the modified code base.

Screenshots:





We've shared five screenshots from CodeMR. These metrics include complexity, coupling, cohesion, and size, among others. We can see in the first image that there are no problematic or highly problematic classes, which is excellent. The distribution of quality attributes shows a healthy balance with the majority being green, indicating low-medium levels for metrics like complexity and coupling. However, this also means there is room for improvement in some areas to shift more into the green. Here's a brief description of the observed metrics:

- **Lack of Cohesion (LoC):** The BookResource class has a low level of cohesion, suggesting that the methods within this class might not be functionally related, which can make the class difficult to maintain.
- **Size:** The BookResource class is indicated as having a low-medium size. This metric often correlates with the number of lines of code, indicating that the class is not overly large and likely more manageable.
- **Coupling:** Here, the BookResource class is shown to have low coupling, which is generally positive as it implies that this class has few dependencies on other classes, making it more modular and easier to maintain.
- **Complexity:** The UserResource class is marked with low-medium complexity, suggesting that the class isn't too complicated, which would make it easier to understand and modify.

Task 3(c): Leveraging LLMs for Refactoring

Following are the screenshots that shows the LLM response when we tried to leverage the Google's Bard for Refactoring.

Here is an example of how you can refactor the code for adding a book:

```
Java

public Response add(
    @FormParam("isbn") String isbn) throws JSONException {
    if (!authenticate()) {
        throw new ForbiddenClientException();
    }

    // Validate input data
    validateInputData(isbn);

    // Fetch the book or try to get it from a public API
    Book book = getBook(isbn);

    // Create the user book if needed
    createUserBookIfNeeded(book, principal.getId());

    JSONObject response = new JSONObject();
    response.put("id", userBook.getId());
    return Response.ok().entity(response).build();
}

private void validateInputData(String isbn) {
    ValidationUtil.validateRequired(isbn, "isbn");
}

private Book getBook(String isbn) throws ClientException {
    BookDao bookDao = new BookDao();
    Book book = bookDao.getByIsbn(isbn);
    if (book == null) {
        try {
            book = AppContext.getInstance().getBookDataService().search(
            } catch (Exception e) {
                throw new ClientException("BookNotFound", e.getCause().getMe
            }
            if (book != null) {
                bookDao.create(book);
            }
        }
        return book;
    }

private void createUserBookIfNeeded(Book book, String userId) throws Cl
    UserBookDao userBookDao = new UserBookDao();
    UserBook userBook = userBookDao.getByBook(book.getId(), userId);
    if (userBook == null) {
        userBook = new UserBook();
        userBook.setUserId(userId);
        userBook.setBookId(book.getId());
        userBook.setCreateDate(new Date());
        userBookDao.create(userBook);
    } else {
        throw new ClientException("BookAlreadyAdded", "Book already add
    }
}
```

Use code [with caution](#).

I hope this helps!

Unfortunately, I cannot generate the complete refactored code without understanding the specific implementation details and dependencies of your project. However, I can provide you with a more detailed breakdown of the refactoring process based on the design smells and code smells identified earlier:

Refactoring Steps:

1. Split the Class:

- Create separate classes for `AppInfoResource`, `AppConfigResource`, and `AppLogResource`, each responsible for their respective functionalities.
- Extract relevant methods and fields from `AppResource` into these new classes.
- Ensure proper dependency injection (if applicable) for required services/dao in each class.

2. Introduce Dependency Injection:

- If you're using a dependency injection framework, inject necessary dependencies like `ConfigDao`, `MemoryAppender`, etc., into the resource constructors instead of creating them directly.
- This improves testability and makes it easier to swap implementations if needed.

3. Define Constants:

- Create a dedicated `Constants` class or interface to store all hardcoded values like config type names, appender names, etc.
- Use these constants throughout the code instead of repeating the same values.

4. Extract Helper Methods:

- Identify large methods like `info`, `update`, and `log` and break them down into smaller, more focused methods with clear responsibilities.
- This improves code readability and maintainability.

5. Use Meaningful Types:

- Instead of using raw strings for configuration values, consider creating dedicated classes with appropriate properties and validation logic.
- This makes the code more self-documenting and type-safe.

6. Improve Exception Handling:

- Instead of catching generic `Exception` and re-throwing it, use more specific exception types like `ForbiddenClientException`, `ConfigurationException`, etc.
- Provide more informative error messages to aid debugging and troubleshooting.

This was the response generated by Google's Bard (now Gemini) for refactoring the code for adding a Book in the BookResource class. It suggested that the SRP is being violated but replied with that it cannot generate the refactored code without knowing the context for the Java Application.

This was the response when we asked for refactoring, it also included the Code Smells in the Design smells.

- **Specific Refactoring Goals:** What aspects of these methods do you want to improve? Are you aiming for better readability, maintainability, performance, or a combination of these?
- **Coding Conventions:** Does your project follow any specific coding conventions or style guides (e.g., PEP 8, Google Java Style Guide)?
- **Additional Context:** Any other relevant information about the project, technologies used, or constraints that might influence the refactoring approach?

General Refactoring Principles:

While I await your response, here are some general principles you can consider when refactoring the code:

1. **Encapsulate Functionality:** Break down complex methods into smaller, well-defined functions with clear responsibilities. This improves readability and modularity.
2. **Reduce Duplication:** Identify and eliminate repetitive code blocks. Use variables, functions, or abstractions to avoid redundancy.
3. **Improve Error Handling:** Make error handling more explicit and informative. Catch specific exceptions and provide meaningful error messages for debugging.
4. **Validate Input:** Implement robust input validation to prevent unexpected errors and improve data integrity.
5. **Utilize Data Structures and Libraries:** Leverage appropriate data structures (e.g., maps, sets) and libraries (e.g., Apache Commons Lang) to simplify code and make it more efficient.
6. **Consider Design Patterns:** Explore the applicability of design patterns such as Command, Strategy, or Factory Method to enhance code design and flexibility.

Potential Refactoring Strategies (Pending Specific Guidance):

- **Method Restructuring:** Break down long methods into smaller, more focused functions.
- **Input Validation Enhancement:** Add comprehensive input checks using techniques like `ValidationUtil`.
- **Error Handling Improvement:** Catch specific exceptions and provide tailored error messages.
- **Code Simplification:** Use simpler expressions, data structures, or algorithms where possible.
- **Code Abstraction:** Extract frequently used code blocks into functions or helper classes.
- **Utilize Libraries:** Incorporate suitable libraries to streamline code and leverage existing functionality.

I'm committed to providing effective refactoring guidance once I have a clearer understanding of your requirements and preferences. Feel free to share more details, and I'll be happy to assist you further.

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

```
import java.text.MessageFormat;
import java.util.ArrayList;...
```

Show drafts ▾

Unfortunately, I cannot provide a complete refactored version of the code without specific feedback on your refactoring goals and preferences. However, I can suggest some general improvements and potential refactoring areas based on the principles and code you provided:

General Improvements:

- **Consider using JPA annotations:** Instead of manually building `JSONObject` responses, you could leverage JPA annotations and object mapping libraries like Jackson to automatically handle object serialization and deserialization. This would simplify code and offer better performance.
- **Improve exception handling:** Instead of catching broad `JSONException`, catch specific exceptions and provide more informative error messages.
- **Utilize DAO methods:** For repetitive tasks like checking existing tags or fetching by username, consider utilizing existing DAO methods to ensure consistency and avoid code duplication.
- **Use DTOs for data transfer:** Define Data Transfer Objects (DTOs) to represent data exchanged between the resource and services. This improves separation of concerns and avoids exposing internal entities directly.

Potential Refactoring Areas:

- **Reduce code duplication:** The validation logic in `add` and `update` is similar. Refactor it into a separate reusable function.
- **Use builder pattern for tag creation:** Consider using a builder pattern to simplify tag creation logic and improve readability.
- **Move authentication logic to a separate filter:** Extract authentication logic to a filter or interceptor to avoid repetitive checks in each method.
- **Handle input data more consistently:** Use the same validation approach (e.g., annotations) consistently for all methods.
- **Consider using a data access layer abstraction:** If applicable, introduce an abstraction layer over the `TagDao` for better testability and potential database flexibility.

Remember, these are just general suggestions. The most appropriate refactoring approach depends on your specific needs and constraints. Once you provide more details about your objectives and preferences, I can offer more tailored recommendations.



When I asked Gemini to refactor the code and report the design smells that would be resolved from the refactoring, this was the response when we asked for refactoring, it also included the Code Smells in the Design smells.

When asked to refactor the code after giving the whole context for the imported files and the Code for the BookResource class, it still reported with the General Improvements for the provided code.

1. Reduce code duplication:

- The user update functions (`update` with 1, 2, and 3 parameters) have repetitive logic for validating and updating basic user information. You can consolidate this logic into a separate function to reduce code duplication.
- Similarly, the logic for deleting the currently logged-in user and deleting a user by the admin are almost identical. Refactor this into a single function with an extra parameter for specifying the username.

2. Leverage JPA entities and annotations:

- Instead of manually creating `User` objects in various methods, use the generated JPA entities and annotations for creating, updating, and fetching data. This can improve code maintainability and reduce manual data mapping.
- Consider using `@NotEmpty` and `@Size` annotations for validation instead of string length checks with `ValidationUtil`.

3. Improve exception handling:

- Instead of catching broad exceptions like `Exception`, catch specific exceptions (e.g., `UserNotFoundException`, `DuplicateUsernameException`) and provide more informative error messages.
- Use specific HTTP status codes (e.g., `404` for not found, `400` for bad request) in your response based on the exception type.

4. Use secure password hashing:

- Currently, passwords are stored in plain text within the `User` object. Implement secure password hashing (e.g., with BCrypt) before storing passwords in the database.

5. Consider dependency injection:

- Instead of creating instances of DAOs within your service methods, inject them through the constructor or use Spring dependency injection. This improves testability and maintainability.

6. Use builder pattern for user creation:

- Consider using a builder pattern for creating new users to improve readability and allow optional fields.

7. Unit testing:

- Implement unit tests for your service methods to ensure they behave as expected and catch potential issues early.

These are just some general suggestions, and the specific areas you focus on will depend on your priorities and needs. By addressing these suggestions, you can further improve the clarity, maintainability, and security of your code.

This was the response when we gave the ConnectResource class that was identified by Sonarqube to contain the design smells it gave too lousy outputs that were mostly targeted to fix the code smells.

Contributions:

The Contribution made by the Team Members are as follows:

1. UML Design: Shivansh Sharma, Soham Kale, Nilesh Keshwani, Mayank Gupta
2. Sonarqube: Vedant Nerkar
3. CodeMR: Vedant Nerkar, Prakhar Gupta
4. Identifying Design Smells: Shivansh Sharma, Soham Kale, Nilesh Keshwani, Prakhar Gupta, Vedant Nerkar and Mayank Gupta
5. Refactoring Code Base: Shivansh Sharma, Soham Kale, Nilesh Keshwani, Prakhar Gupta, Vedant Nerkar and Mayank Gupta
6. Re-evaluating CodeMR: Soham Kale, Shivansh Sharma
7. LLM Response for Refactoring: Nilesh Keshwani, Mayank Gupta, Prakhar Gupta
8. Documentation: Shivansh Sharma, Soham Kale, Nilesh Keshwani, Prakhar Gupta, Vedant Nerkar and Mayank Gupta.