**Team 6: Book Reader Documentation**

[**Section 1: Introduction**](#_he7znzwflv7h) **1**

[**Section 2: Architectural Documentation**](#_hljabjkqblpy) **3**

[Login Page](#_ral79sunj6ql) 4

[Register Page](#_u3z0xvkb8k3v) 6

[Search Page](#_zfcsn24ena7d) 8

[Library Functionality](#_4hdbwzfatnp7) 11

[Public and Private Wall Functionality](#_oxra5pu5b3w9) 14

[Profile Page](#_m615c5ddthrs) 17

[Friendship Functionality](#_nd1nx7l8lk5e) 21

[**Section 3: Context Diagram**](#_2cveio2x7b3x) **24**

[**Section 4: Variability Guide**](#_z4lftazcfequ) **25**

[**Section 5: Rationale**](#_pgzlmakoqtgl) **26**

[Strategy](#_v7y1k6pe1zka) 26

[Builder](#_js02pt9t0gwz) 27

[Template Method](#_lbnej0ebwce2) 27

[Chain of Responsibility – Authentication](#_z5d4ftxdfh1n) 28

[Chain of Responsibility – Create Posts](#_dg5o5p29jw4z) 28

[Composite](#_b23kaofdog8y) 28

[Decorator](#_htmhulcd3fu2) 29

[Bridge](#_av8n0asw71nr) 30

[Facade – formerly Factory Method](#_jjjvmu60n85q) 30

[Flyweight](#_lbs7xy5cx71b) 31

[Template](#_du2xo09ucmwz) 33

[Singleton](#_3hkmlov36w0s) [–](#_z5d4ftxdfh1n) Search Book Proxy 35

[Singleton](#_99lranfv96uk) [–](#_z5d4ftxdfh1n) Controller 35

[Proxy](#_pzkavovco778) 35

[**Section 6: Installation Guide**](#_rvnjs5kgwjtm) **37**

[**Section 7: Individual Contribution**](#_qzzcg6tzcoz8) **38**

# Section 1: Introduction

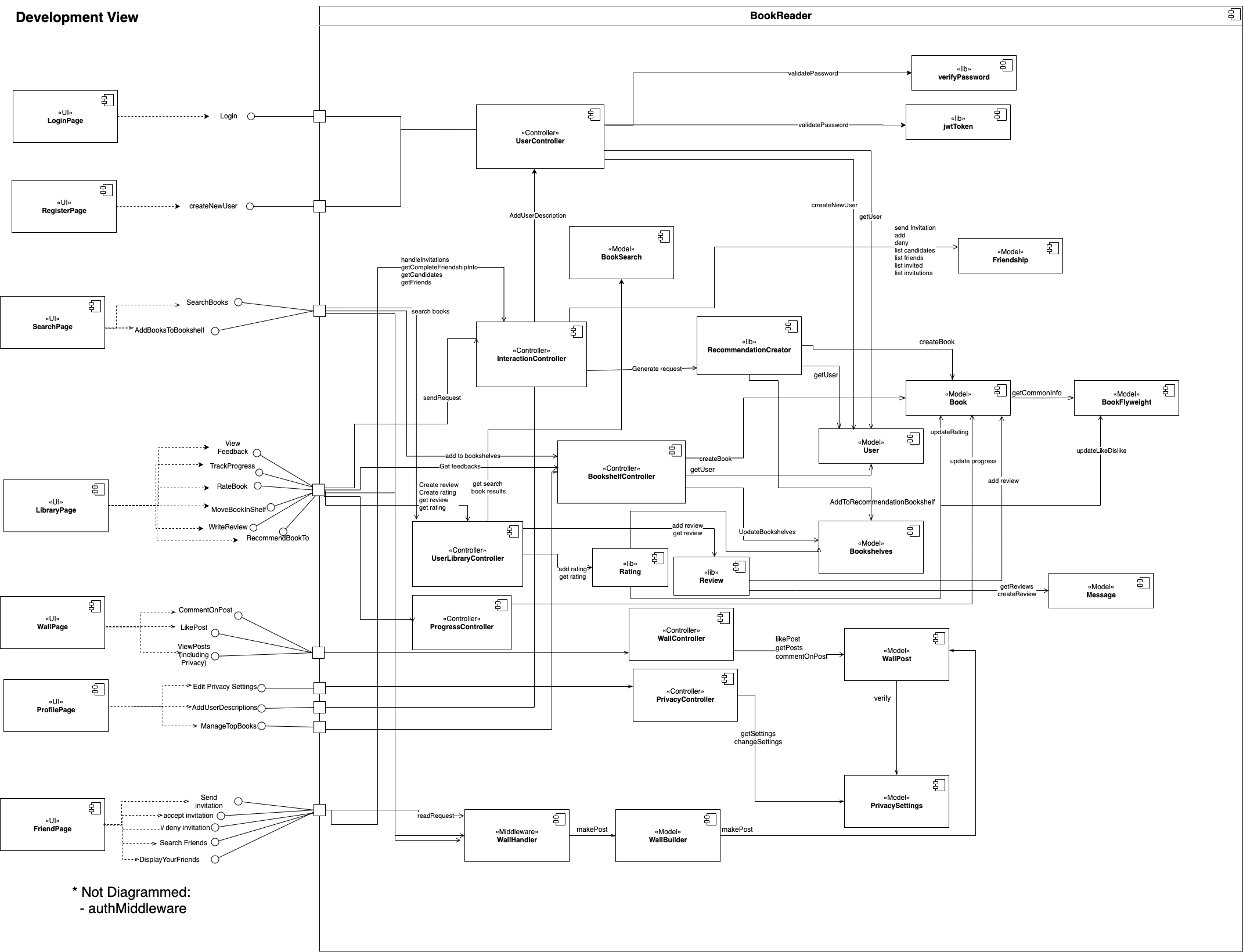
In this digital age, the availability of books are endless. Whether it’s historical autobiographies or dystopian fantasies, we nowadays can read almost any book from any genre with a touch of our fingertips.

This is where the problem comes in. There’s just so much choice out there that it’s almost impossible to determine what book that we would enjoy reading. Moreover, traditional forum websites like Reddit have a certain dominant preference for specific books. What if there was a way to connect like-minded book readers together? To meet other book readers who enjoy similar books as you? To be a part of a community of like-minded book readers where everyone can keep track of their favorite reads, share their favorite books on their feed, and more?

This is where our solution, BookReader, comes in. BookReader is a new, revolutionary, book-based social media application designed to give book readers a platform to share their love of reading. It is designed with user-experience in mind, maintaining an elegant, easy-to-use, and intuitive user interface. We wanted BookReader to be an open platform that enables readers to connect with other readers from across the globe. Unlike other social media applications like Twitter and Facebook, BookReader is designed as a book-specific application, giving readers the chance to connect with other readers and express their love of reading. Most importantly, we wanted to emphasize the social aspect of reading, which is why we designed many actions, such as adding books to bookshelves, switching books between bookshelves, adding new friends, updating book progress, submitting a new review/rating, and more, to automatically prompt a post to the public and private walls. Our hope is that with this platform, readers can connect with each other like never before, and build a community of like-minded book lovers.

Now what you’re probably thinking: “T6, there’s half a dozen different teams who are trying to solve the problem you’ve identified. What differentiates you?” We believe our superior user interface, responsive UI, incorporation of a user profile, privacy settings feature, and other team-specific functionalities sets our solution apart from our peers. In this report, we are going to describe our software architecture in detail. We will present the elements of our architecture, their properties, their relations to each other, their behavior, and more. We will then provide an overview of how our system relates to its environment. That will be followed by a variability guide, where we describe the changes we would make to our system if more time were permitted. We will end with a thorough discussion of the rationale behind our major design decisions. With this report, we hope you, the investor, will become convinced of our solution to revolutionize the book reader’s market with our superior architectural solution. At the end, you will find attached an installation guide in order to navigate our application yourself. Lastly, you will read an explanation from every individual member from our team detailing their contributions to the project.

# Section 2: Architectural Documentation

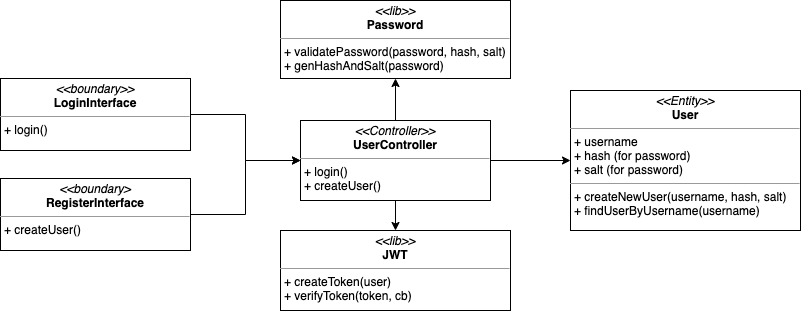
****

High Quality Link: <https://app.diagrams.net/#G1quXy72ACWO7xt79hpVeKBW5PxLFpznqL>

*Development View of the Book Reader App*

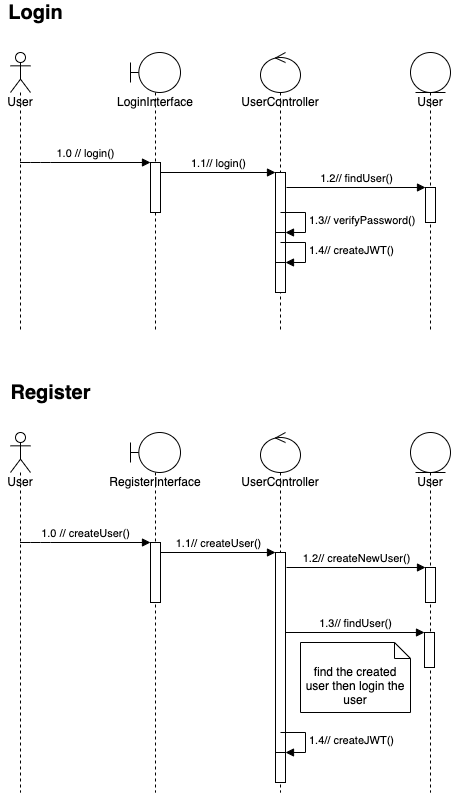
## Login Page

1. Elements and their properties
   * Login Interface: This is the login page that will allow users to input username and password. It will also handle the onclick event to submit the user information with HTTP post request to log in a user.
   * UserController: The user controller will handle the login request from the client. The controller will then connect with libraries and the user model to validate the request.
   * User Model: It defines the user schema and contains the business logic related to users.
   * Password: This library handles the logic related to password, such as verify the password and encrypt the password.
   * JWT: This library handles the logic related to JSON web tokens, such as creating new tokens and verifying tokens.
2. Relations and their properties
   * This page communicates with the user controller to validate the password and create a new JSON web token. The user controller will find the corresponding user by its username through the user model. It will then validate the user through the imported library password.js. It can check if the password provided by the current user is correct compared to the user record. Lastly, if the user is authenticated, the user controller will call supporting jwt.js from the lib folder to create a unique JSON web token for the logged-in user. In the following class diagram, you will see the relationship between the user controller, user model, and supported libraries.



*Class Diagram for Login and Register*

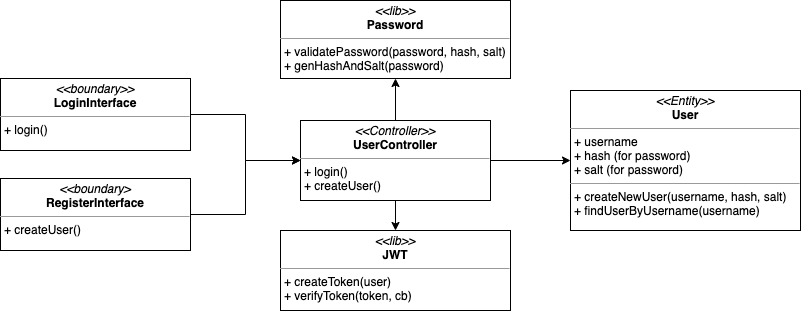
1. Elements Interface
   * Referring to the development view from Section 1, we will create a login page with the username and password fields. When the login action is initiated, it will connect to the user controller to validate the user.
2. Elements Behavior
   * As we can see in the following sequence diagram, the user will initiate the login action with the username and password. The login interface will then create an HTTP POST request, which will ask the user controller to implement the login request. To verify the user, the controller will first find the corresponding user by its username from the user entity. After successfully get the user, the controller will verify the password and create a unique JSON web token for the logged-in user.



*Sequence Diagram for User Login*

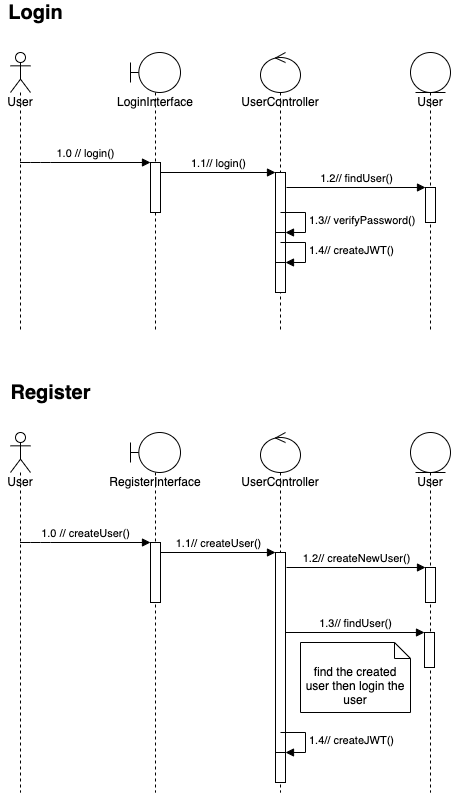
## Register Page

1. Elements and their properties
   * Register Interface: This is the register page that will allow users to input username, password, and confirmed password. It will also handle the onclick event to submit the user information with HTTP post request to create a new user.
   * User Controller: The user controller will handle the register request from the client. The controller will then connect with libraries and the user model to create a new user.
   * User Model: It defines the user schema and contains the business logic related to users.
   * Password: This library handles the logic related to password, such as verify the password and encrypt the password.
   * JWT: This library handles the logic related to JSON web tokens, such as creating new tokens and verifying tokens.
2. Relations and their properties
   * This page communicates with the user controller to validate the password and create a new JSON web token. The user controller will find the corresponding user by its username through the user model. It will then validate the user through the imported library password.js. It can check if the password provided by the current user is correct compared to the user record. Lastly, if the user is authenticated, the user controller will call supporting jwt.js from the lib folder to create a unique JSON web token for the logged-in user. In the following class diagram, you will see the relationship between the user controller, user model, and supported libraries.



*Class Diagram for Login and Register*

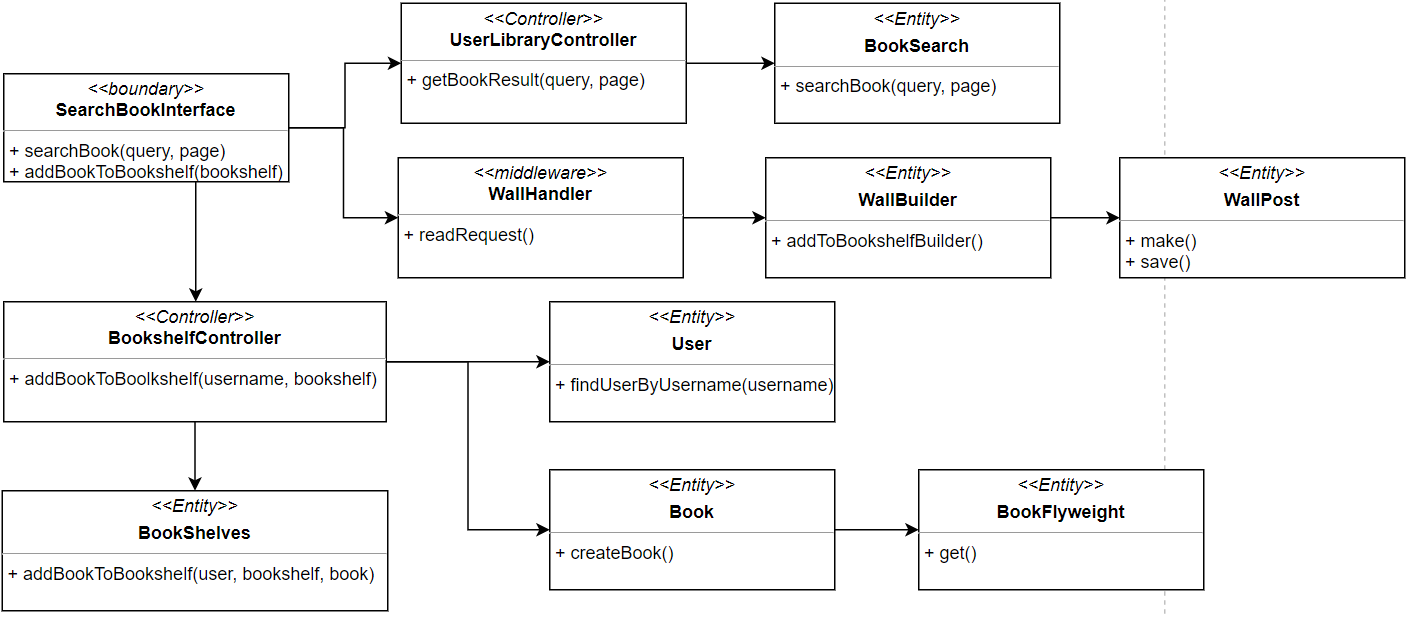
1. Elements Interface
   * Referring to the development view from section 1, the registration page will include username, password, and confirmed password fields. After submitting the form, the registration page will then connect to the user controller to create a new user.
2. Elements Behavior
   * As you see in the following sequence diagram, the new user will initiate the registration process with the username and password. The frontend of the registration page will then create an HTTP POST request to the user controller to create a new user. After creating the user, the controller will then find the newly created user through the user model and create a unique JSON web token for it.



*Sequence Diagram for User Registration*

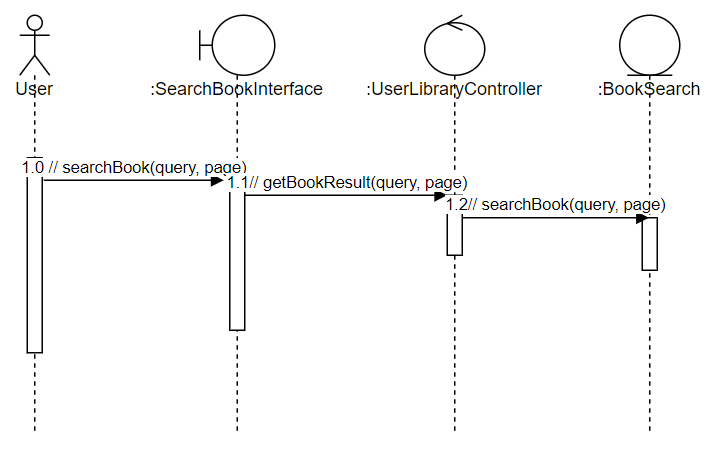
## Search Page

1. Elements and their properties
   * Search Page: This is the frontend display of the user search page. It initiates the user search book event, which includes searching books and adding the book to specific bookshelves
   * UserLibrary Controller: This handles request logic related to the user library including getting google book results in search book functionality
   * BookSearch Model: This model communicates with google search book engine to pre-handle the request parameters and connect to the google book API
   * Bookshelf Controller: This handles request logic related to the bookshelf including adding the book to specific bookshelves.
   * BookFlyweight Model: The stores the common information about a book including book isbn, title, authors, description, thumbnail, page count, likes and dislikes number
   * Book Model: This stores the information about a book with shared BookFlyweight information
   * Bookshelves Model: This stores the Book to the user's bookshelves: WantToRead, Reading, Read and Favorite
2. Relations and their properties
   * The SearchPage communicates with the UserLibraryController via HTTP requests. The UserLibraryController will get the book result through the BookSearch model using the search query and page index. The BookSearch model will connect to the GoogleBooks API to get the book results.
   * The SearchPage communicates with the BookshelfController via HTTP requests. The BookshelfController can add a book to a specified bookshelf. It first gets the corresponding user by its username through the User model. Then, it gets the Book and BookFlyweight information and adds it to the user’s bookshelves.
   * When the client wants to add the book to bookshelves, it also connects to the middleware WallHandler. The WallHandler will build addToBookshelfBuilder to make the post and save the post to the database. Then, it will post to private and public walls.



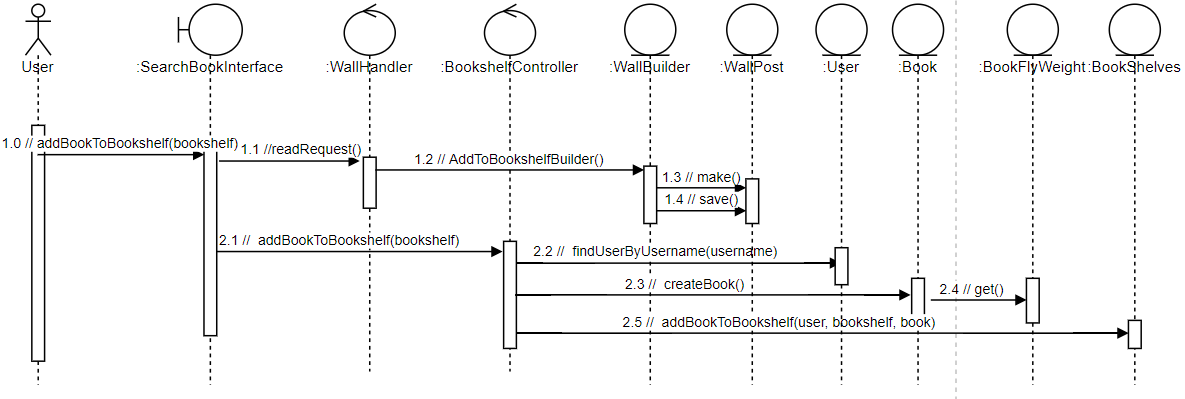
*Class Diagram for Search Page*

1. Elements Interface
   * Referring to the development view from section 1, the search page will include search book and search option fields. After submitting the search book query, the search page will then connect to the UserLibraryController to get book results from GoogleBookEngine. After getting back the book results, the client can click the button to add the book to bookshelves: WantToRead, Reading, Read, Favorite. The BookshelvesController will handle the behavior to add to specific bookshelves.
2. Elements Behavior
   * As shown in the diagram below, the user can go to the search page to search any books. It will connect with the UserLibraryController via HTTP request to get book results with the parameter search query and page index. From there, the UserLibraryController will interact with the BookSearch model to get the google book result.



*Sequence Diagram for Searching Book*

* + As shown in the diagram below, the user in the search page can add the book to bookshelves. The WallHandler will read the request sent to the BookshelfController and create a post on the wall. Afterwards, the request will go to the BookshelfController with the username, bookshelf name and book information. From there, the BookshelfController will interact with the User, Book, BookFlyweight and BookShelves model to add the book to specific user’s bookshelf. BookshelfController will interact with the User to find the user by the username. Then, it will interact with the Book and BookFlyweight model to create a book with shared bookFlyweight information including book isbn, title, thumbnail, authors, description, page count, like and dislike number. At the end, the controller will interact with the Bookshelves model to add the book to the bookshelf with the found user, book and chosen bookshelf.

****

*Sequence Diagram for Adding Book to Bookshelf*

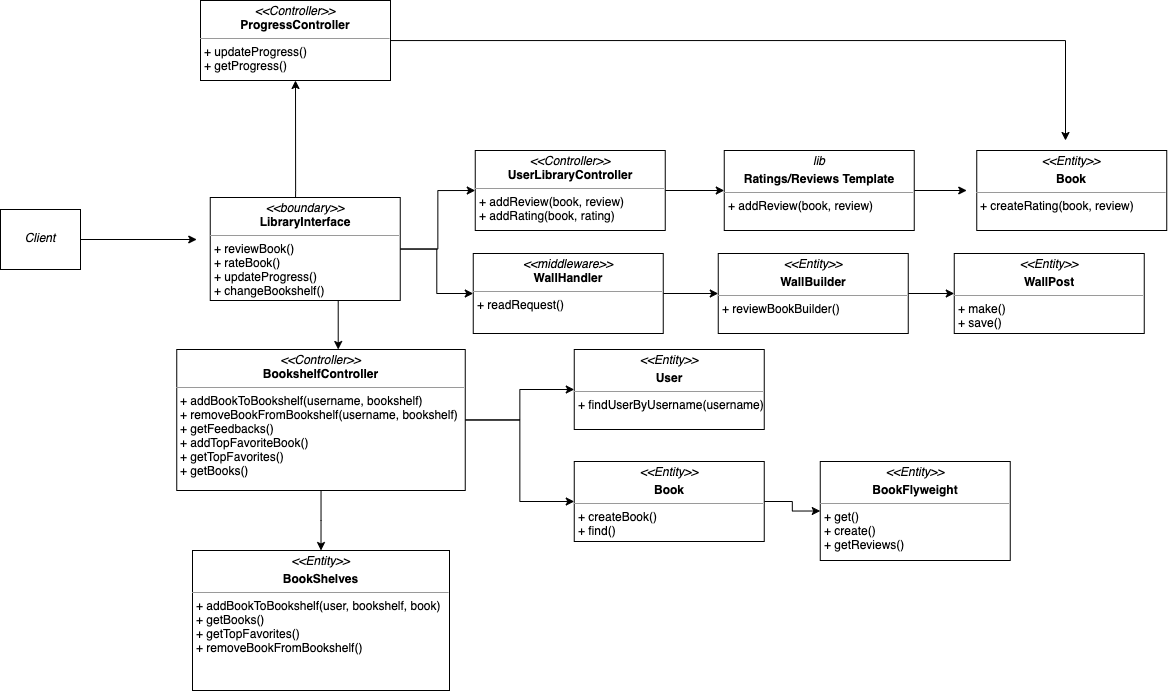
## Library Functionality

1. Elements and their properties
   * Bookshelf Controller: This handles the backend logic related to manipulating a user’s bookshelf.
   * UserLibrary Controller: This handles logic associated with the bookshelf, but not manipulating the bookshelf itself, such as adding and viewing reviews/ratings for different books
   * Progress Controller: This handles the logic associated with keeping track of and updating the progress a user made when reading a book.
   * Book Model: This stores all the information for a book, including a flyweight that has book-specific information. It also contains book-specific methods such as updating progress and ratings
   * Book Flyweight Model: This stores book-specific information such as author, title, page count, thumbnail, isbn, description, likes, and dislikes
   * Bookshelves Model: This is the entity that has all the information related to the different types of bookshelves (and the topFavorites shelf)
   * Ratings and reviews: This is the code component associated with adding ratings and reviews to books
2. Relations and their properties
   * The bookshelf controller communicates with the book, bookflyweights, and bookshelves model to manipulate logic related to the bookshelves and display them to the library page. For example, to add a book to a bookshelf, we have to get the flyweight associated with an isbn, find the book associated with the flyweight, and then add that book to the bookshelf model. In the bookshelf view page, we will have displayed all of the books associated with the bookshelf.

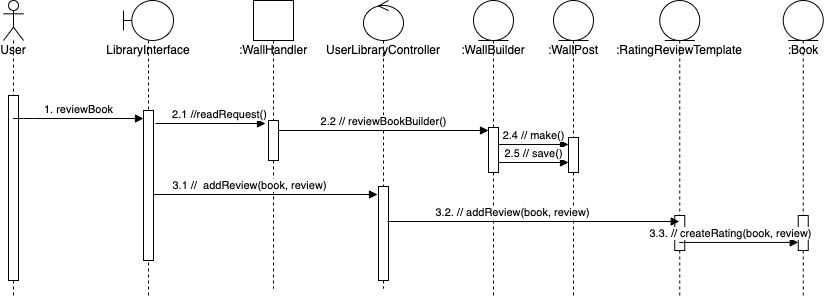
To manipulate logic related to reviews and ratings, the frontend display will show the review/rating information. Then, when the user changes it, this communicates with the userLibraryController, which handles the review/ratings logic. It updates the appropriate book based on the user input, and then displays the changes on the frontend.

*Class Diagram for Library Page*

1. Elements Interface
   * The two central communication controllers in this section are the bookshelf controller and the user library controller. This bookshelf manipulates most of the logic associated with bookshelves. It communicates with the necessary models, who themselves have logic associated with them. For example, this controller communicates with the book, book-flyweight, and bookshelves models to get books, get/add to top favorites, add books to bookshelves, remove books from bookshelves, and get feedback (likes/dislikes/reviews associated with the books). The user library controller communicates with the books in order to add reviews/ratings to each book. There is also one more (less major) controller responsible for updating user progress on particular books. It communicates with the book model to update reading progress.



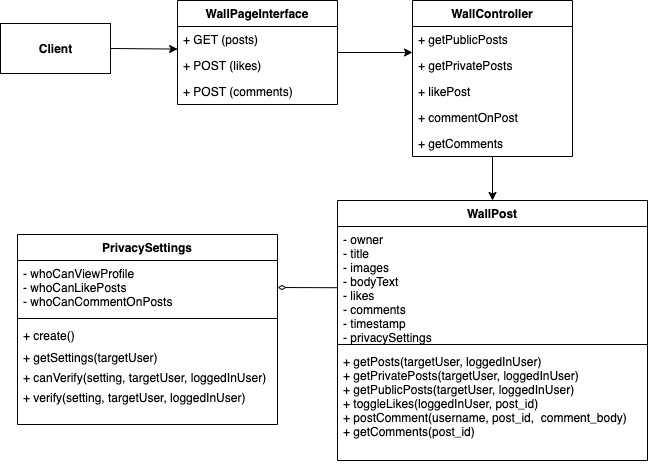
1. Elements Behavior
   * The Library page initiates a GET or POST request. Before going to the controller, the request is read from the WallHandler Chain of Responsibility, and creates a new post on the user’s wall. From there, the request goes into the UserLibraryController, and performs the functionality that is requested. For example, if the request is ReviewBook, the controller interacts with a helper module that creates a review for a book, and saves it to the database. This flow is the same for RateBook. For updating book progress, the flow is very similar. Instead of going to the UserLibraryController, the request goes into the ProgressController. From there, the controller communicates directly with the Book model to get and update the user’s progress.



*Sequence Diagram for Adding a Review to a Book*

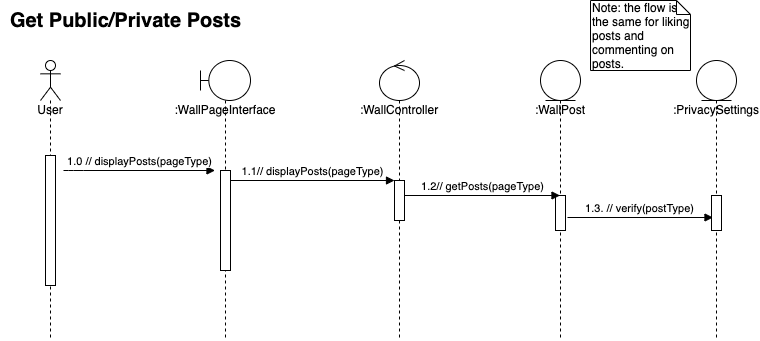
## Public and Private Wall Functionality

1. Elements and their properties
   * WallPage: This is the frontend display for the wall page. It displays the user’s public and private walls, and also enables the users to like and comment on a post.
   * WallController: This handles all of the logic related to posting on the public and private walls, including liking and commenting on posts.
   * WallPost: This component communicates with the database to get and edit posts. It also communicates with the privacy settings model to ensure the posts that are retrieved are within all users’ privacy settings.
   * WallHandler: This handles requests to post on the public and private wall
   * WallBuilder: This builds the needed post information to public and private wall
2. Relations and their properties
   * The WallPage communicates with the WallController via HTTP requests. From there, the WallController will call the function corresponding to the request. For example, if the WallPage is trying to get all public posts, the controller will call getPublicPosts within the WallPost model. For each action initiated within the WallPost model, the action is initiated depending on the privacy settings of the user. The WallPost will call PrivacySettings.verify, which is a strategy that can be extended later to include additional functionalities.
   * In addition, many external functionalities including reviewing a book, rating a book, and adding a user as friend communicates with the Wall Handler component. The WallHandler is responsible for making a post on the Public and Private Walls. The WallHandler instantiates a concrete wall builder, which makes a post on the wall. This can be exemplified by the class diagram for the library functionality (*Class Diagram for Library Page*).



*Class Diagram for Wall Page functionality*

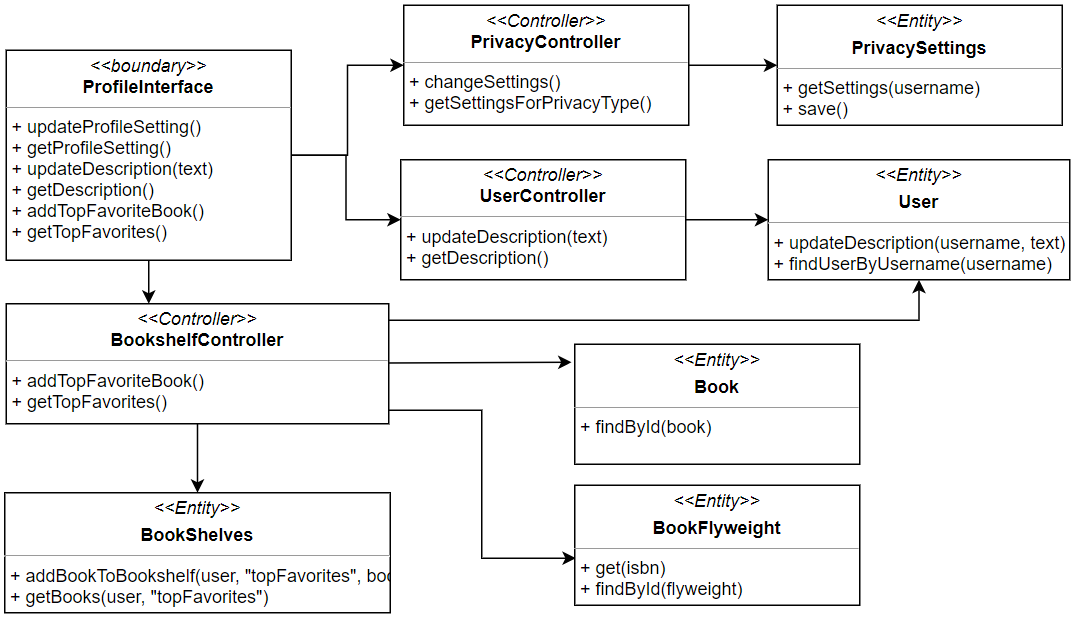
1. Elements Interface
   * Referring to the development view from section 1, the WallPage communicates with the WallController by initiating a request. The WallController communicates with the WallPost by calling getPosts, toggleLikes, and getComments. From there, the WallPost model communicates with the privacy settings model to ensure the action can be performed.
2. Elements Behavior
   * As shown in the sequence diagram below, the user can go to the public/private wall page to view all posts. The page will perform a GET request if the user is trying to view the wall, and a POST request if the user is attempting to like or comment on a post. From there, the controller will interact with the WallPost model to perform the requested action. Finally, the WallPost model will communicate with the PrivacySettings model to ensure that the action can be performed.



*Sequence Diagram for Performing an Action on the Wall*

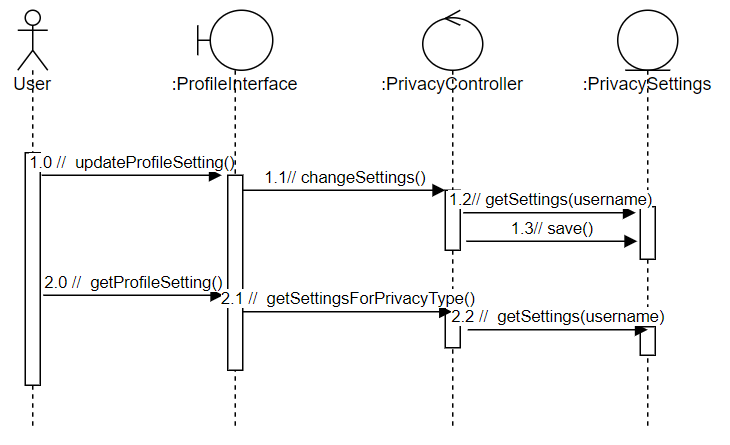
## Profile Page

1. Elements and their properties
   * Profile Page: This is the frontend display of the profile page. It initiates the user profile event, which includes editing privacy settings, adding user descriptions and managing top books
   * User Controller: This handles the request logic related to the user including getting description and updating user description in the profile page.
   * User Model: This stores the user information including privacy settings and description in the profile page.
   * Bookshelf Controller: This handles the request logic related to the bookshelf including adding the book to the top favorite bookshelf.
   * Book Model: This stores the information about a book with shared BookFlyweight information
   * BookFlyweight Model: The stores the common information about a book including book isbn, title, authors, description, thumbnail, page count, likes and dislikes number
   * Bookshelves Model: This stores the Book to the user's top favorite bookshelf.
   * Privacy Controller: This handles the request logic related to the privacy settings including changing settings and getting settings for privacy type.
   * PrivacySettings Model: The stores the user’s privacy settings including who can view profile, who can like posts and who can comment on posts.
2. Relations and their properties
   * The Profile page communicates with the PrivacyController via HTTP requests. The PrivacyController will update or get the user's profile settings: who can view profile, who can like posts and who can comment on posts. The PrivacySettings model will update the user profile settings or retrieve them from the PrivacySetting database.
   * The Profile page communicates with the UserController via HTTP requests. The UserController will update or get the user's description. The User model will update the user description or retrieve the user’s description from the User database.
   * The Profile Page communicates with the BookshelfController via HTTP requests. The BookshelfController can add the book to the top favorite bookshelf or retrieve all books information from the top favorite bookshelf. It first gets the corresponding user by its username through the User model. Then, it gets the Book and BookFlyweight information to add it to the user’s top favorite Bookshelf. Or, it finds all top favorite books from the bookshelves and for each book in the top favorite bookshelf, it finds Book and BookFlyweight information and sends them back to the client.



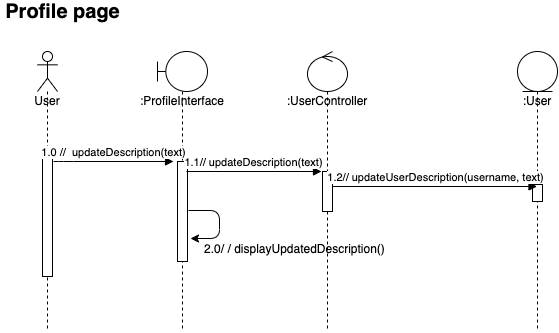
*Class Diagram for User Profiles*

1. Elements Interface
   * Referring to the development view from section 1, the profile page will include manage privacy settings, user description and top favorite books. The PrivacyController will handle the behavior of updating privacy settings and getting privacy settings. The UserController will handle the behavior of updating the user description and getting the user description. The BookShelf Controller will handle the behavior of adding or getting the top favorite books which will be shown in the privacy wall according to the user’s privacy settings.
2. Elements Behavior
   * As shown in the diagram below, the user can go to the profile page to update their profile setting including who can view profile, who can like post and who can comment on posts. It will connect with the PrivacyController via HTTP request to update and get the profile setting. From there, the PrivacyController will interact with the PrivacySetting model to get and save the setting by user’s username.



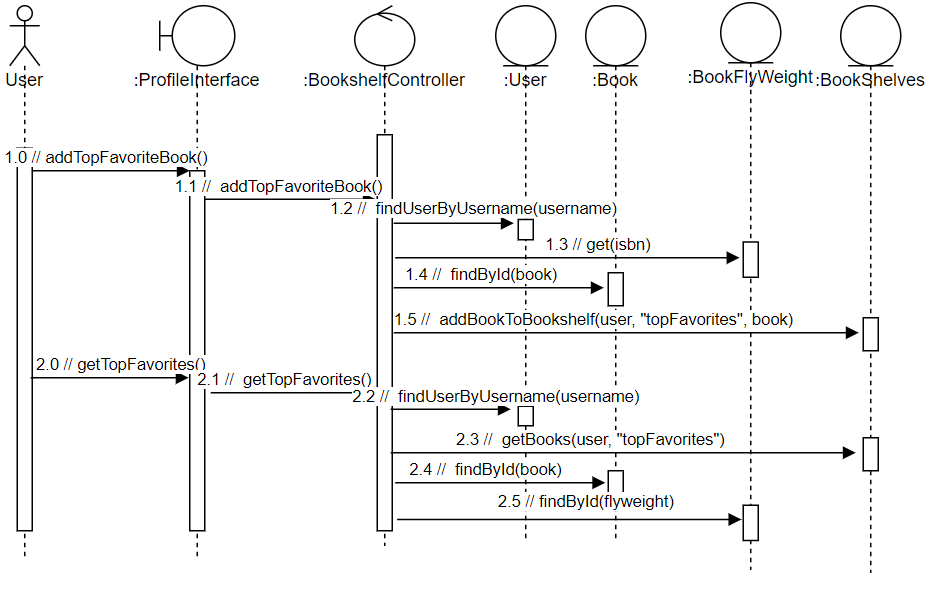
*Sequence Diagram for updating and getting profile settings*

* + As shown in the diagram below, the user can go to the profile page to update their description about themselves. It will connect with the UserController via HTTP request to update and get the description. From there, the UserController will interact with the User model to update and get the user’s description by user’s username.



*Sequence Diagram for updating and getting description*

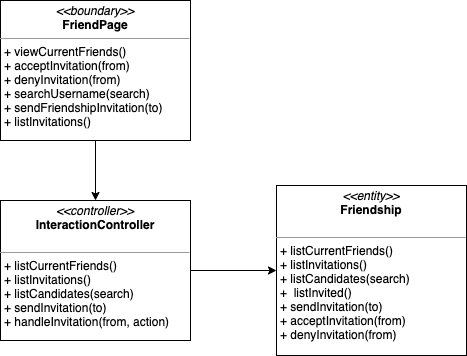
* + As shown in the diagram below, the user in the profile page can manage their top favorite bookshelf. It will connect with the BookShelfController via HTTP request to add and get the book to the top favorite bookshelf with the username and book information. From there, the BookshelfController will interact with the User, Book, BookFlyweight and BookShelves model to add the book to the user's top favorite bookshelf. BookShelfController will interact with the User to find the user by the username. Then, it will interact with the Book and BookFlyweight model to find the book and common book information in BookFlyweight. At the end, the controller will interact with the Bookshelves model to add the book to the bookshelf with the found user, book and top favorite bookshelf. When the user wants to get top favorite books about a user, the BookshelfController will interact with the user first to get the target user information. Then it interacts with the Bookshelves to retrieve all books in the top favorite bookshelf. For each book gotten from the bookshelf, the controller finds the book and shared book flyweight information and sends those book information to private wall interface



*Sequence Diagram for managing top books*

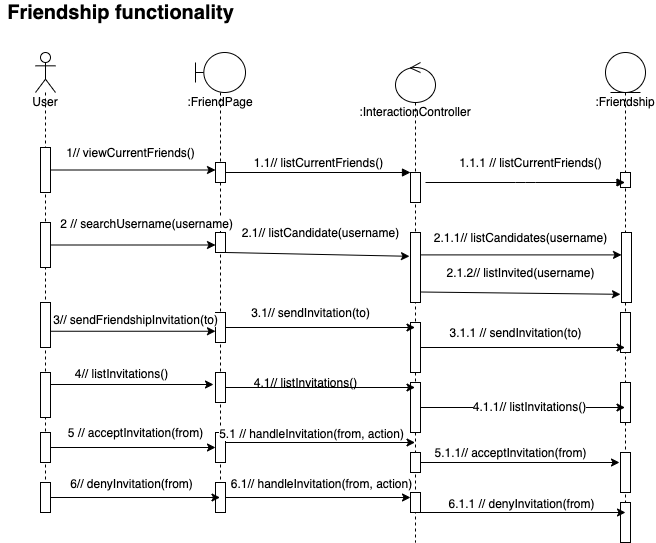
## Friendship Functionality

1. Elements and their properties
   * FriendPage: This is the UI interface where users can view and manipulate all friendship related information.
   * Interaction Controller: This element receives incoming requests related to displaying or modifying friendship data. It delegates the requests to appropriate elements with actual business logic. Finally, interactionController returns the result back to clients.
   * Friendship Model: This element stores a user’s friendship data and includes methods to modify these data. It directly manipulates the database documents.
2. Relations and their properties
   * FriendPage is the frontend client that makes different API requests based on the end user’s action. These requests are handled by InteractionController who calls appropriate methods in the Friendship model. After the Friendship model is done processing, it returns the result back to InteractionController. Finally, the InteractionController wraps the result in an JSON object and passes back to the FriendPage.



*Class Diagram for friendship functionality*

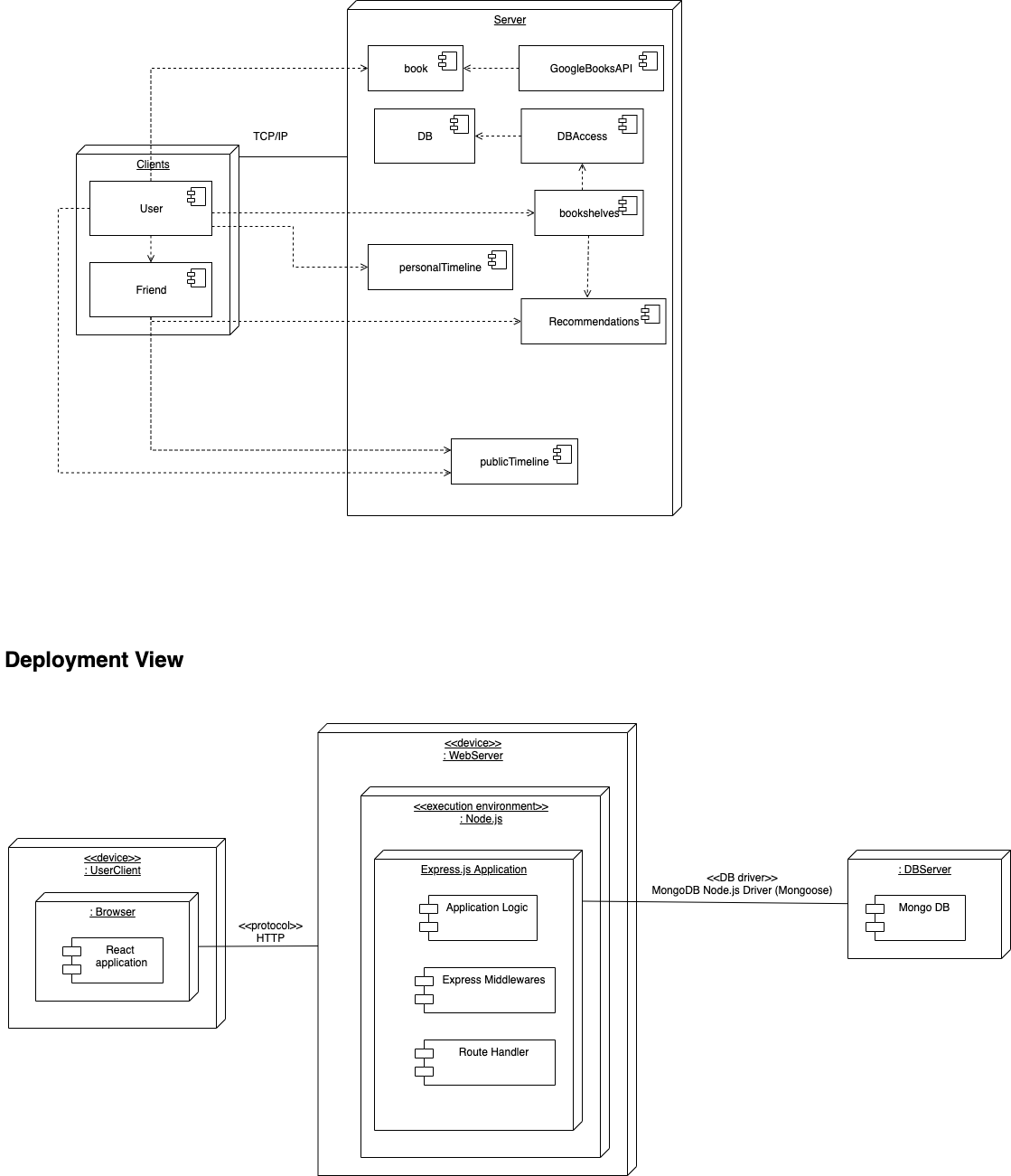
1. Elements Interface
   * The FriendPage interface includes the following main operations: see a user’s current friends, accept/deny an invitation, search for system users and send a friendship invitation.
   * The InteractionController interface includes the following main operations: list current friends, list invitations, list candidates based on search query, send invitations, and accept or deny an invitation.
   * The Friendship model interface includes the following main operations: list current friends, list invitations, list candidates based on search query, list invited users, send invitations, add invitations, and deny invitations.
2. Elements Behavior
   * The behaviors among the three elements related to friendship functionality is straightforward. The user can initiate different actions on FriendPage, where the actions are given to InteractionController and Friendship model to process. The sequence diagram depicts all major operations on FriendPage and the processing that follows.

*Sequence Diagram for friendship functionality*

# Section 3: Context Diagram

There are three major components in the BookReader app, which includes the client, web server, and database server. On the client-side, the browser will generate pages according to the requested components in the frontend. For instance, every page on the front end will include a navigation bar component.

In order to interact with the web server, the client can initiate the HTTP request to implement certain behavior, such as login, create posts, add friends, etc. On the web server side, our team decided to use Node.js to easily execute javascript code outside the web browser. We also utilized express.js as the backend web framework to manage routes, middlewares, and application logic. Lastly, we used MongoDB as our database server and utilized Mongoose to model our data. Below, you can see the structure of our program and how they are associated with each other.

****

*Deployment View of the Book Reader App*

# Section 4: Variability Guide

There are small variations we would do to our system had we had more time. One thing has to do with a specific design pattern we chose to implement for our privacy settings: the strategy. After consulting with our team of software architects, we concede that the State pattern would’ve also been an appropriate design pattern to implement the user privacy settings. In the state implementation, the different states would extend the state class, and the privacy settings would implement the settings based on the setting’s internal state. Nevertheless, we believe our current strategy implementation is architecturally sound, but felt the state decision was worth taking a second look at.

In terms of usability, we would make small improvements to make our application more intuitive and user-friendly. For example, currently in the library, when the user clicks on a bookshelf, there is no marker that indicates which bookshelf the user is currently using. We would like to fix this by having the clicked bookshelf highlight in green, so the user knows exactly which bookshelf they are looking at.

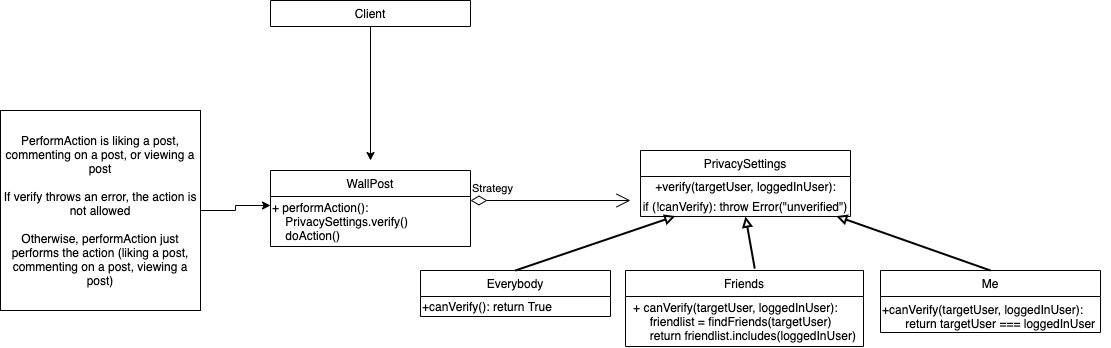
One last usability improvement we could have implemented given more time would be to hide the like/comment button if a particular user’s privacy settings prevent liking or commenting on their posts. Currently, we just send a message to say the feature is disabled if a user attempts to do so. It would be better from the user’s perspective that they know what they are and are not allowed to do immediately when the page loads up.

In general, we are very confident in our current architectural design decisions, but feel that with more time, these changes could be made to ensure our application remains competitive in the book social media application market.

# Section 5: Rationale

## Strategy

We incorporated a strategy pattern to implement the privacy settings feature in an extendable manner.

****

*Class Diagram for Strategy Pattern*

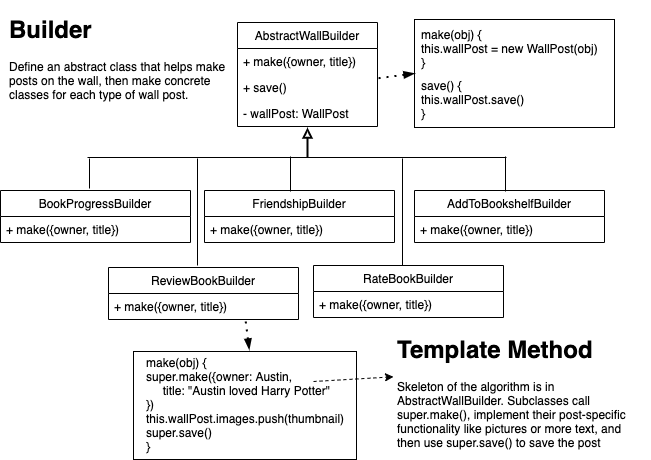
For implementation, WallPost is the WallPost model and PrivacySettings is the PrivacySettings model. Note for our implementation, we use case-switch statements to distinguish our strategy, but if we had sufficient time, we would refactor our implementation so that the canVerify() method is inherited by the Everybody, Friends, and Me strategy subclasses. That way, if we were to want to add more privacy settings (for example, friends of friends could perform an action), we would be following the open-closed principle, in which the class is closed to modification, but open to extension. With our current implementation, we violate the open-closed principle, because if we want to add a new strategy, we would have to edit the class directly. However, as a team, we decided that we needed to take on some small technical debt in order to finish our project before the deadline. Nevertheless, we maintain that our current implementation is architecturally sound, and a quick refactor would allow our architecture to continue to adhere to the SOLID principles.

## Builder

We make a builder as a part of what is needed to create a post on the wall. In the builder, we have an abstract class that helps make a post on the wall, then make concrete classes for each type of wall post (diagram below #3). The builder solves the problem of creating posts on the wall; now, specific concrete builders can extend the abstract builder and create the necessary post.

## Template Method

The second necessary design pattern to create a post on the wall. The skeleton of the algorithm is in AbstractWallBuilder. Subclasses call super.make(), implement their post-specific functionality, then call super.save(). This describes the template method.

****

*Class Diagram for Builder Pattern and Template Method*

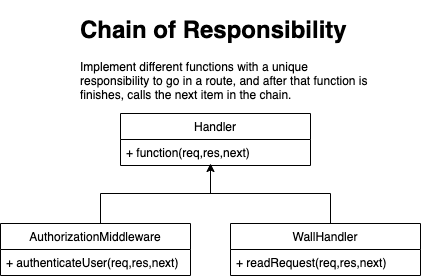
These design patterns solve the problem of creating posts on the wall. Besides developers worrying about creating posts, we separate the logic completely, so developers only have to be worried about the main functionality. Later, a developer can go in and extend the abstract builder if they want a particular action to create a post on the wall.

## Chain of Responsibility – Authentication

This pattern is designed to authenticate user requests by using a middleware, which could prevent users from viewing the page without authentication. In the middleware/authentication.js, it will handle the authentication process which includes verifying the JWT token. If the authentication passes, the current handler will transfer control to the next handler in the chain, if not, it will block the user and return an error. We have included the user authentication in most of our routes which can be viewed in routes/apis.js.

## Chain of Responsibility – Create Posts

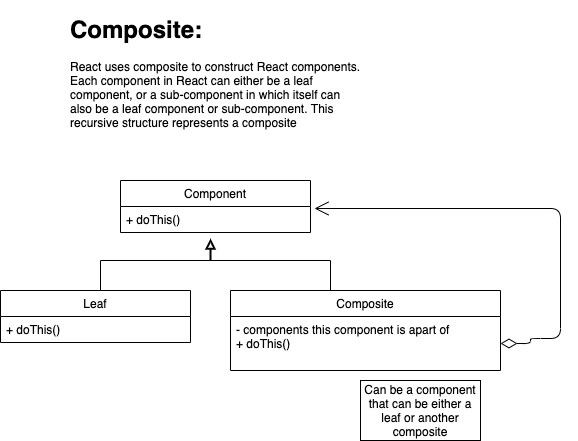
Used to create posts for public and private walls. In the chain, the request is read before it is passed to the controllers. If it is an action that should prompt a post on the wall, the observer creates and executes the template/builder algorithm described above.

****

*Class diagram for Chain of Responsibility*

## Composite

React uses the composite pattern to build complex user interfaces. For each component, each component can either be a leaf component or a composite that can either be a leaf or another composite. This recursive pattern represents a composite and represents the hierarchical structure for components in React.

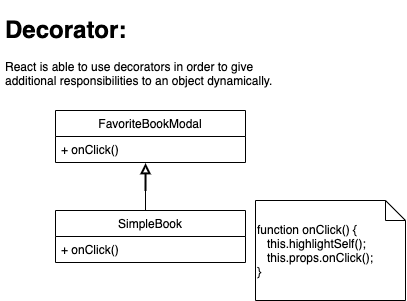
****

*Class Diagram for Composite*

## Decorator

React uses the Decorator pattern in order to assign additional responsibilities to a function after passing it as a prop from a parent component. This is best illustrated by an example from our source code. Let's say we are clicking on books to add to the top favorite display. Each time we click a book, we want to add that book to a list that will later be sent to the server.

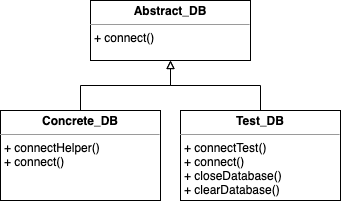
Let’s also say we want the book to highlight green when it is clicked. This is where the decorator pattern is exemplified. We can change the color of the background of the book by defining the onClick method in the subComponent. After changing the color, we can then call this.props.onClick() to execute the onClick function in the parent class. Theoretically, this can be repeated ad infinitum and can be used to define different behaviors to the onClick function.

****

*Class Diagram for Decorator Pattern*

## Bridge

We used the bridge pattern to implement our database. It enables us to manage different databases independently from our codebase. In the following class diagram, you can see that the Abstract\_DB class generalizes databases' concrete implementations. The concrete\_DB extends the Abstract\_DB to connect to two types of databases: local DB and cloud DB. The Test\_DB was also extended through the Abstract\_DB and can manage the database for testing purposes.

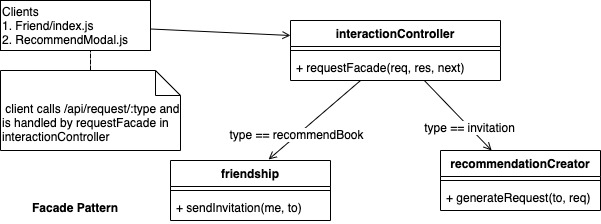


*Class Diagram for Bridge Pattern*

## Facade – formerly Factory Method

Clients could make two types of requests to a user’s friend: one is the request to recommend a book, and the other is the request to send an invitation. Each of the request types is handled by a dedicated module. We would like to provide a simplified interface, so the clients do not need to know about the different modules that handle the requests. The facade pattern helps us achieve this goal.

The clients make requests from two frontend component files: *Friend/index.js* and *RecommendModal.js.* They send requests to */api/request/:type*, where the path parameter *type* could be either *recommendBook* or *invitation*. The module *interactionController.js* has a method *requestFacade* that handles these api requests. For *type* equals *recommendBook*, the facade calls the *recommendationCreator.js* to handle the business logic related to adding a book to a friend’s recommendation bookshelf. For *type* equals *invitation*, the facade calls *friendship.js*, which handles the logic for sending an invitation in one of its methods *sendInvitation*.



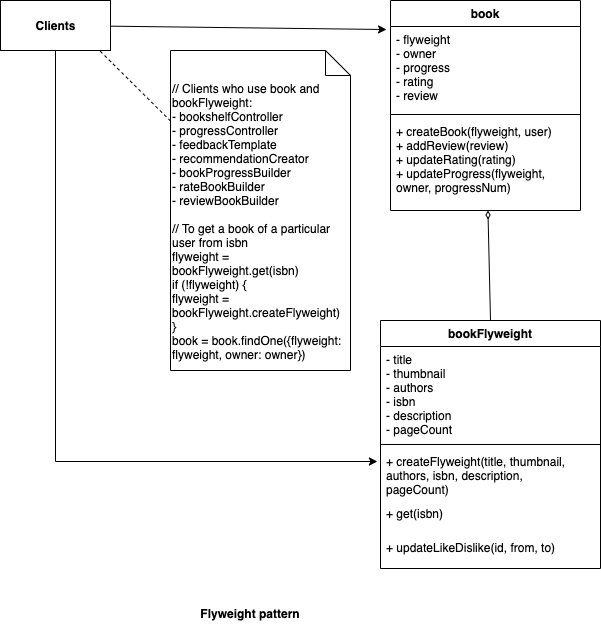
*Class Diagram for Facade Pattern*

## Flyweight

Whenever a user decides to add a new book to the bookshelf, the BookReader system would need to create a MongoDB documentation, which would record the information of that book, user rating, user review, reading progress and owner. However, we would like to have only one copy of immutable book data in our database, namely title, ISBN, thumbnail, authors, description and page count. We also want to store the total counts of likes and dislikes of a book in one place. This way, when we need to display the overall likes and dislikes of a book, we do not have to query for that book from every user who has it before adding up all individual ratings. So we leverage the Flyweight pattern based on these two considerations.

The common information of a book object is extracted into *bookFlyweight.js.* We also put the total likes and dislikes into schema definition. The *bookFlyweight.js* contains methods for creating flyweight, finding flyweight in db based on ISBN, and updating likes and dislikes. All our book flyweights are stored in MongoDB collection *bookFlyweights*, with each document including the following fields: *title, ISBN, thumbnail, authors, description, pageCount, like, dislike.* We store individual user’s information into *book.js*. The *book.js* contains methods for creating a book, adding review, updating rating and updating progress. So in the MongoDB collection *books*, each document includes the following fields: *flyweight* (a mongoDB document id pointing to a document in *bookFlyeights* collection), *owner*, *progress*, *rating*, and *review*.

When the clients in the frontend want to add a book, remove a book, add rating, add review, or update progress, they include the book’s ISBN. We would search for the bookFlyweight documentation that has the matching ISBN. Then we would find the correct book documentation based on the bookFlyweight id and name of owner. After that, we do manipulation on the book documentation and store the changes.

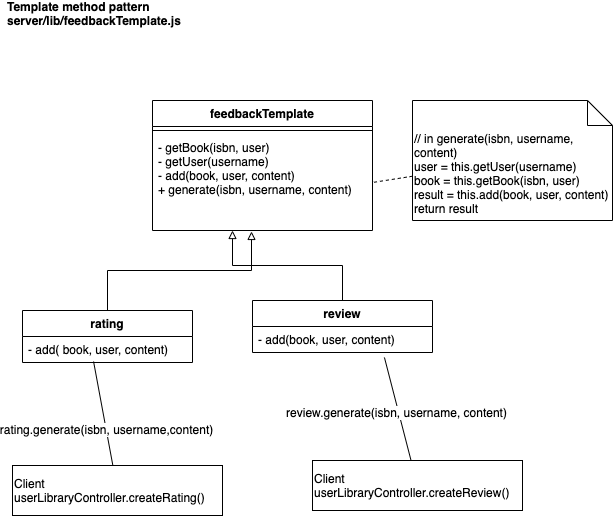


*Class Diagram for Flyweight Pattern*

## Template

When a review or rating is to be added to a book, there are some common preprocessing and post-processing steps. In the preprocessing steps, we need to get a user document from our db based on the username. Then we used the user document and isbn to get a bookFlyweight document, which is then used to get a book document. The common post-processing step is sending the success or fail result back to the frontend. The only different step is we are either adding a review or rating to a book. So we use the template pattern to let a base class handle the common steps. Two specific handler objects, *rating* and *review* in *feedbackTemplate.js,* implement the different add logic.

In the *review* handler object, the add logic involves creating a message document, adding it to db, then referencing the message document to the book document. In the *rating* handler object, the add logic involves updating the total like/dislike counts in bookFlyweight document, changing the rating in book document, and also moving to and from favorite bookshelf if necessary.

****

*Class Diagram for Template Method*

## Singleton - Search Book Proxy

We create a single instance of the bookSearch service. When the clients want to search books, we want to have a global book search instance with google search API instead of several objects. Class bookSearchProxy plays the role of Singleton and Controller userLibraryController will use the single instance of bookSearchProxy when the clients request to search books in the search page.

The Singleton getter is in server/models/bookSearch.js and the singleton object will be used in server/controllers/userLibraryController.js

## Singleton - Controller

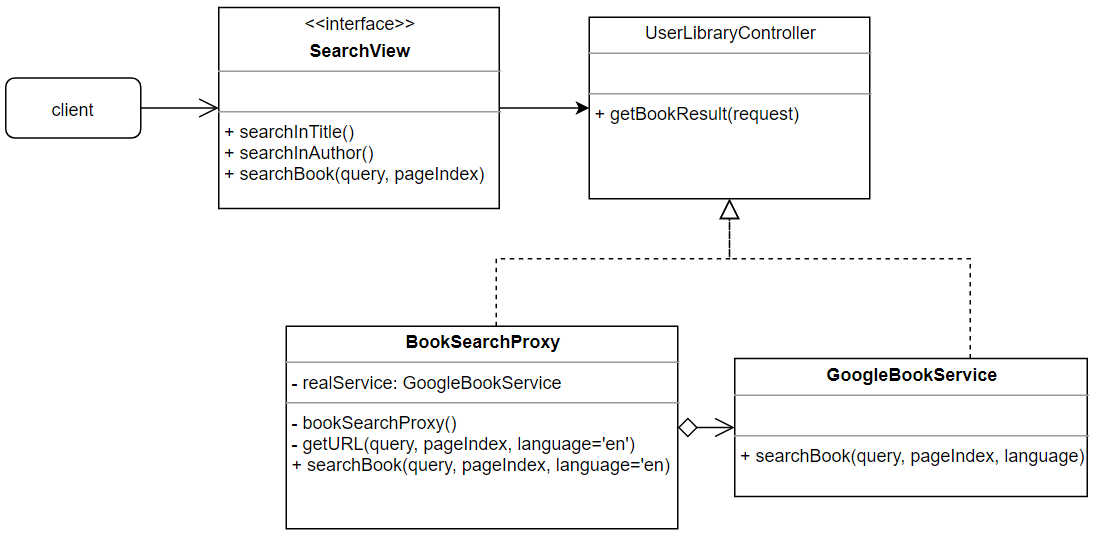
We use Singletons for each of our controllers. To represent a singleton in Javascript, we decided that the simplest way to do this was to use object literals. We simply define an object, and include all methods for the controller within the object. This also gives us the ability to define private functions within the controller when we need to; we simply just choose not to export functions that we want to keep private.

While it is true that most singletons are a class with a static getInstance function, we concluded that the overall complexity of implementation was not worth the benefits. The way we implemented it makes our controllers simpler and more maintainable, while still ensuring that there is only one instance of each controller.

## Proxy

We use Proxy patterns to control the access to the real google search book API. Instead of directly connecting to the google book API, we use Class bookSearchProxy to pre-deal with the request parameters and then request to the real google book API. In BookSearchProxy, it will handle the page index, search options (only search query in author/title, or all) and specific language (we set english as default). Then, the proxy will connect to the real google book API and get back the book result based on those request parameters setted by the clients.

When the clients in the search page enter the search query with page index and search filters, searchView will send the request to our controller. Then, userLibraryController(in server/controllers/userLibraryController.js) will ask the proxy to connect with the real google book API. BookSearchProxy(in server/models/bookSearch.js) will handle the request parameters and send the book results back.

****

*Class Diagram for Proxy*

# Section 6: Installation Guide

1. Install all packages npm install
2. Create keys for JWT node server/lib/generate-key-pair.js
3. Start server yarn server
4. Start client yarn client

# Section 7: Individual Contribution

**Mimi**

In this project, I contributed to the frontend and backend of login, registration, and authentication middleware. I also supported the top book functionality that will be displayed on the private walls. Additionally, I worked mostly on the frontend to refactor all pages in the book reader application to be consistent, helping our team achieve better usability. Besides developing the app, I also documented the meeting notes for our team to manage the schedule and task assignments better.

**Sabina**

I am in charge of creating functionalities related to friendship, including seeing all current friends, searching friends, handling invitations, and recommending books to friends. I also implemented the functionalities that allow users to add rating and review to a book and see a book’s rating/review from the search page.For each of the functionalities, I created backend business logic and the first version of the frontend UI (using simple bootstrap components, no additional styling). Some additional minor features that I am in charge of: adding pagination to search results, syncing like/dislike action with favorite bookshelf, separating bookFlyweight and book.

**Tracy**

In this project, I contributed to the frontend and backend of the searching books, removing the book, tracking book progress and managing the top book functionality. For each of the functionalities, I designed the original frontend with the react component, created the backend business logic and integrated them with my backend parts with design patterns. Additionally, I fixed some small bugs, such as adding some error handlers on login and registration pages to show some alert information.

**Austin**

In this project, I contributed to the frontend and backend for the library functionality. Additionally, I worked on the backend (especially the design patterns) for the functionality related to posting to the wall. I also implemented the privacy settings functionality, including getting posts, liking posts, and commenting on posts in accordance with a user’s privacy settings. Additionally, I refactored a lot of the React code to conform with proper React style and to reduce the number of bugs and increase the usability of the system. Finally, I edited the frontend to display helpful information for users if they are currently not logged into the system but are trying to access certain functionality.

**Justin**

In this project, I contributed to the frontend and backend of the library and public and private walls. For the walls, I made the original wall design (that Mimi would later modify to unify our application under one style). I incorporated the logic to add and remove likes, both using Component-level state in React and sending the request to the backend. Additionally, I worked on the comment functionality, including hiding and showing the comment section box on the frontend, and associating comments with particular posts on the backend.