Team 6 – Architecture Haiku

Mimi Shih, Sabina Chang, Justin Starks, Austin Starks, Jiaxin Du

**Brief summary of the overall solution**

The BookReader is a social network with four main functionalities for registered users. First, users can manage individual libraries by adding and organizing books. Second, users can update the reading progress for each book. Third, users can interact with friends on a public wall. Lastly, users can update their profiles to control the information they want to display.

**Design decisions and rationale**

* **MongoDB** to store user information and bookshelves information
* **React** to better extending our front-end
* **Bootstrap** to get clean and professional front-end looks
* **Axios** to make HTTP requests from node.js
* **MVC** pattern to easily model the behavior of our system

**Quality attributes**

Priority: Maintainability > Usability > Extensibility

* Maintainability
  + Source: Developers
  + Stimulus: Developers strive to keep their code organized and structure easily manageable
  + Environment: Design time, Development time
  + Artifacts: Codebase
  + Response: The application will be organized and easy to read. We use React to enhance this by making new components simpler.
  + Response measure: Effort to modify existing code
* Usability
  + Source: Users
  + Stimulus: Users try to find a book, manage their bookshelves, or interact with their friends.
  + Environment: Runtime
  + Artifacts: BookReader UI
  + Response: BookReader app provides a clean and intuitive UI for searching a book, managing books, and interacting with friends. The app will notify the user and his connection if there's any update.
  + Response measure: Number of errors
* Extensibility
  + Source: Developers
  + Stimulus: Developers try to add new functionality or components without touching other parts of the BookReader app
  + Environment: Design time, development time, compile time, and build time
  + Artifacts: The portion of the BookReader app being extended
  + Response: BookReader app provide low coupling and evolution of friendly interfaces to let developers easily add new functionalities and features
  + Response Measure: Efforts to extend the code

**Architectural styles and patterns**

* Design Patterns
  + **Singleton Pattern** for all controllers (will implement)
  + **Singleton Pattern** on our wrapper for the Google Search API
  + **Bridge Pattern** to implement database (will implement)
  + **Factory Method** for creating requests for adding friends or recommending books
  + **Facade Pattern** to provide a single endpoint for tracking book progress and hiding complex code algorithm (ask TA)
  + **Flyweight pattern** for storing the immutable information of a book (title, authors, cover image, description, ISBN, etc.)
  + **Composite Pattern** for when we utilize nested components in React to represent the hierarchical tree structure of the Component
  + **Chain of Responsibility pattern**. Authentication middleware for backend and helper function on the frontend
  + **Chain of Responsibility pattern**. When an action happens (like when a friend recommends a book, adds a user as a friend, etc), the request is first sent to a Chain of Responsibility handler that reads the request and makes a post on the public wall (depending on the request). After reading the request, the next item in the chain (which is often the controller) is called and the request is passed through there.
  + **Builder pattern and Template Method:** After the chain of responsibility handler reads a target request, the Builder Pattern helps make posts for the public/private wall. Also, the Template Method is used in the abstract post builder class, and is used to begin and end the post-building process.
  + **Strategy Pattern**: Is used to implement privacy settings. The client calls the function getPosts() to get all of the posts from the database. Let’s say user1 is trying to look at user2’s posts. The strategy pattern is used to determine whether user1 is within the range of user2’s privacy settings. The client is unaware of the getPosts implementation, and only interacts with the interface.
* Architectural Styles
  + **Layered Pattern** Model-View-Controller to efficiently model the behavior of our system
  + **Client-Server Pattern** to allow the communication between the server and multiple clients

**Description of the fourth functionality**

For our 4th functionality, we plan to expand the user profile to be more fully developed. Users should be able to upload profile pictures, write descriptions of themselves, and manipulate their own privacy settings. For privacy, users should be able to have their content viewable by everyone, their friends only, or just themselves.