Book Exchange

Final Project Write-Up
Software Development (CS121)
Spring 2015

Development Team: Andrew Fishberg, Hannah Rose, & Hannah Young
Client Team: Sagar Batchu, Reid Callan, Won Kyoung Park, & Skyler WIlliams

I. INTRODUCTION AND MOTIVATION

Obtaining books is often expensive, especially when specialized texts are involved. While some books are worth the permanent place on a bookshelf and deserved to be purchased, many books are only needed temporarily. Libraries can fill this need, but they are designed to handle short-term borrowing and fall short when patrons wish to borrow a college textbook, since students often need a specific edition of a highly specialized textbook for several months. Since libraries may not stock enough copies of a textbook, house the right editions, or allow for long loans on popular books, the traditional library system is not always a satisfactory solution. Alternatively, purchasing books from a school bookstore or online vendor can accrue costs of several hundred dollars per semester. While these retailers often also provide a more cost-effective rental or buy-back system, they still cost a non-trivial amount of money for financially strained college students.

Many college students solve this problem by informally borrowing textbooks over the course of a semester from a friend or acquaintance who already owns the textbook. While this system helps to save money, it is limited by the student's contacts. It can also be difficult for the person lending out the book to keep track of who has it.

Our system, Book Exchange, seeks to improve this friendly-borrowing process by providing a platform to locate books that students want to borrow and to loan out books they have from previous semesters. It also enhances the current informal system by providing a record of who loaned a book to whom. Finally, although initially inspired by college textbook loaning, Book Exchange seeks to facilitate book loaning and borrowing beyond the college campus. Again, public libraries have certain limitations, and patrons may have to wait several months to check out new and popular books. Thus, the system has been generalized to allow any book, not just textbooks, to be loaned or borrowed.

II. CORE FUNCTIONALITY

The Book Exchange system achieves our goal of connecting book-sharing users through a Android application frontend and Java server backend. Users only directly interact with the app while the server supplies the app with all relevant data on users, books, and exchanges.

The Book Exchange system refers the process of a user loaning or borrowing a specific book as an ***exchange***. Each exchange has a lifecycle from its creation as an offer to loan or a request to borrow a book, to the physical exchange of the book between users, to its safe return (see Architecture for the technical details of the lifecycle). This allows the application to track the exchange process and to whom the user needs to return the book at the end of the exchange. An ***open exchange*** is an unaccepted offer or request, whereas a ***current exchange*** is one where the book is currently being borrowed by someone. A ***complete exchange*** is one where the book has been borrowed and returned to its owner. An exchange ***advances*** whenever the status of the exchange changes from `INITIAL` (unaccepted offer or request) to `RESPONSE` (current exchange pending user confirmation) to `ACCEPTED` (current, active exchange between two users) to `COMPLETE` (the loaner marks the exchange as finished).

When a user opens the app, they are shown the login screen. Here a user is required to identify themselves, either by choosing to login with an existing account or by registering a new

account. The user is then directed to the main screen, where they can view or access all relevant Exchanges. Screenshots of these activities are shown in Figure No. 1.1 and 1.2.

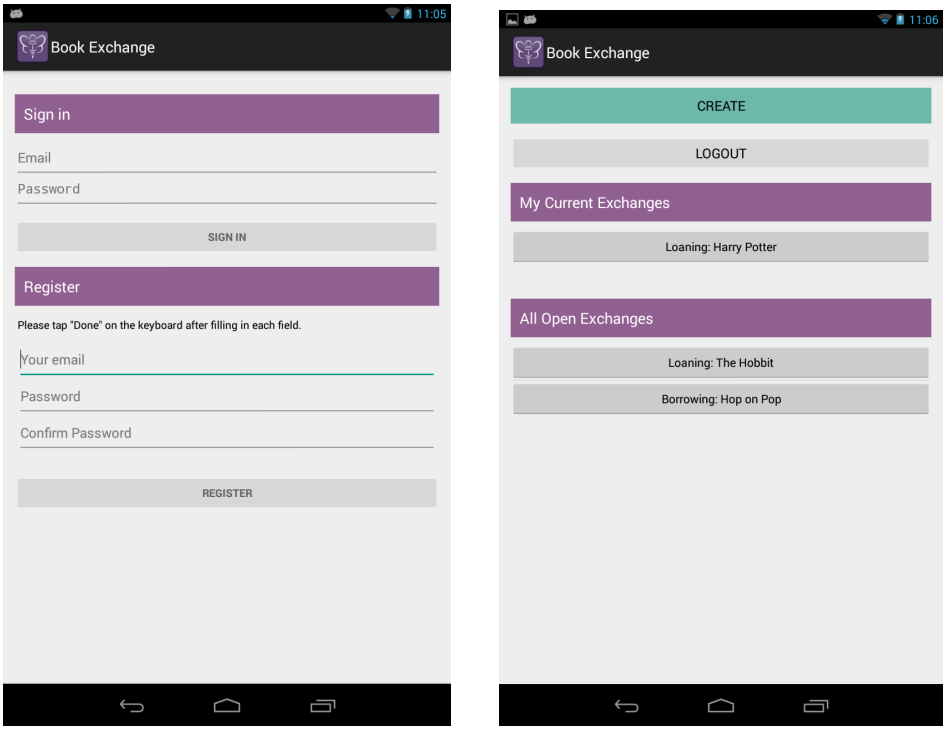

Figure No. 1.1 (left): Login/Registration, Figure No. 1.2 (right): Main Screen

The user can accomplish the following tasks while using the app:

- **Account Management:** The Login/Registration screen when the app opens. To create an account, the user must provide an email address and a password longer than four characters. Once the user is logged into the app, they can logout by clicking a button on the main screen.
- **View Open Exchanges:** From the main screen, the user can view exchanges (both offers and requests) from other users that have not yet received a response. On this page, they will see the book title, cover, author, ISBN, and publication year along with the owner of the book and the nature of the exchange (offer or request). If the exchange is a request, there is a button for the user to offer the book in response. Likewise, if the exchange is an offer, there is a button for the user to request the book in response. (See Figure No. 2.1)
- **View Current Exchanges:** From the main screen, the user can view exchanges (both offers and requests) that they are participating in. This includes books they have offered and loaned out, books they have requested and are borrowing, and their open exchanges. This activity displays the same book details as an open exchange. For current exchanges, there is button for the owner of the book to mark the exchange as complete.
- **Create an Open Exchange (Offer or Request):** From the main screen, the user can post a request or offer a book. The form requires the user to search for and choose a book, then mark the exchange as an offer or a request. Finally, they click the “CREATE EXCHANGE!” button to finalize exchange creation. (See Figure No. 2.2).

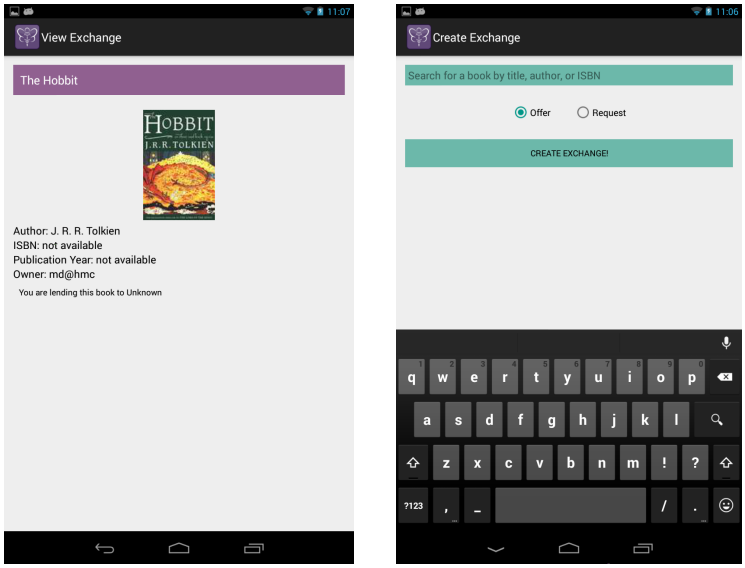

Figure No. 2.1 (left): View Exchange activity, Figure No. 2.2 (right): Create Exchange activity

While the app provides a clean user interface, the server is responsible for handling the data the app relies on. The server is used to:

- **Verify Account Credentials:** The server is responsible for tracking all account information, what accounts are already registered, and verifying or denying the user upon login.
- **Track Exchanges:** The server is responsible for tracking all exchanges that pass through the Book Exchange system. This includes tracking which users are loaning and borrowing a book in a given exchange, the status of the exchange (i.e. offer, borrowed, returned, etc.), and the book that was exchanged.
- **Fetch Book Information:** When a user creates a new Exchange, it is the server's responsibility to fetch all relevant information regarding the book (title, author, ISBN, cover image URL, etc.).
- **User Input:** When a user interacts with their Book Exchange app with the functionalities listed above, this information needs to be transmitted back to the server so other users can see the results. In other words, the server needs to accept the creation of new exchanges and the advancement and completion of others.

III. SYSTEM ARCHITECTURE

Our app is divided into two main sections: the frontend user experience and the server-side backend. The overview of how all these pieces fit together is shown in Figure No. 3.

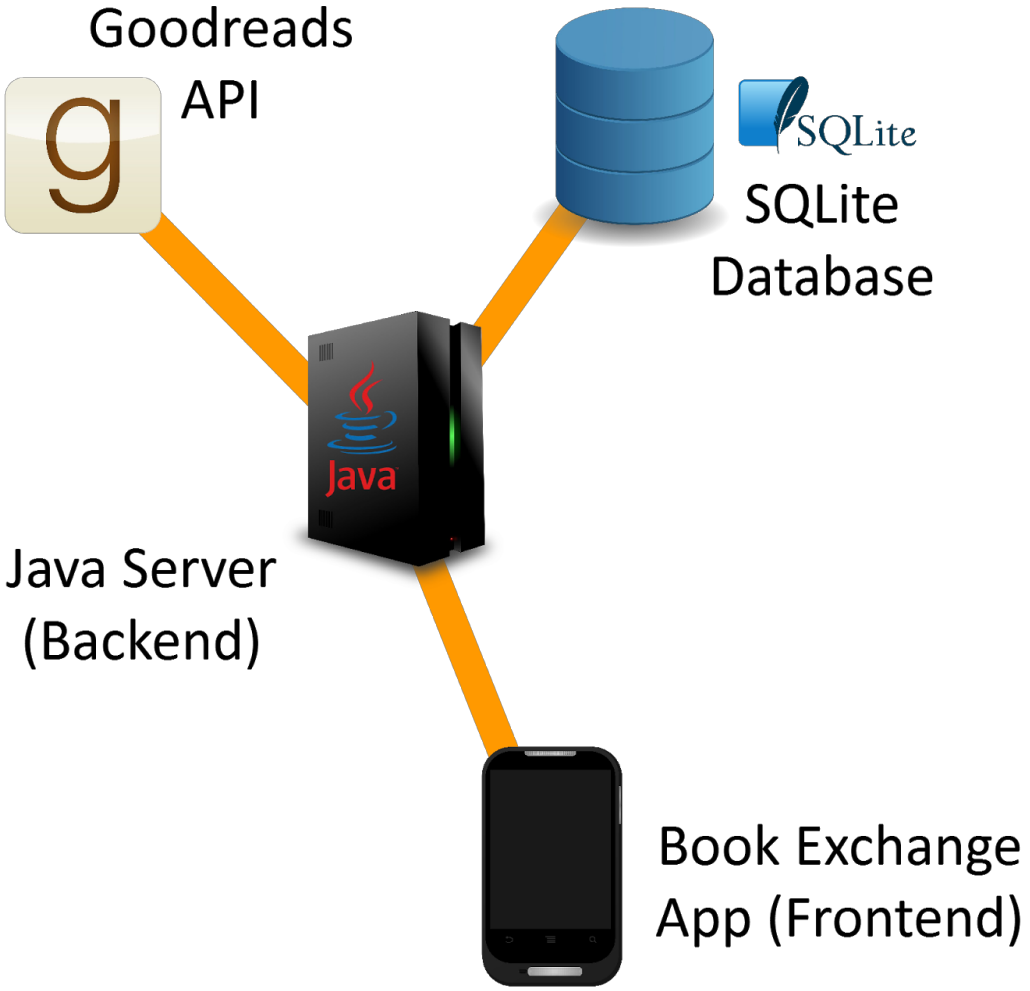

Figure No. 3: Grand scale view of large architecture components

THE ANDROID APPLICATION

The Book Exchange application itself follows the basic design patterns of a traditional Android application, with an `Activity` class corresponding to each user task. There are also custom views and adapters to facilitate the display of `Book` and `Exchange` objects. Figure No. 4 shows the different classes and their connections.

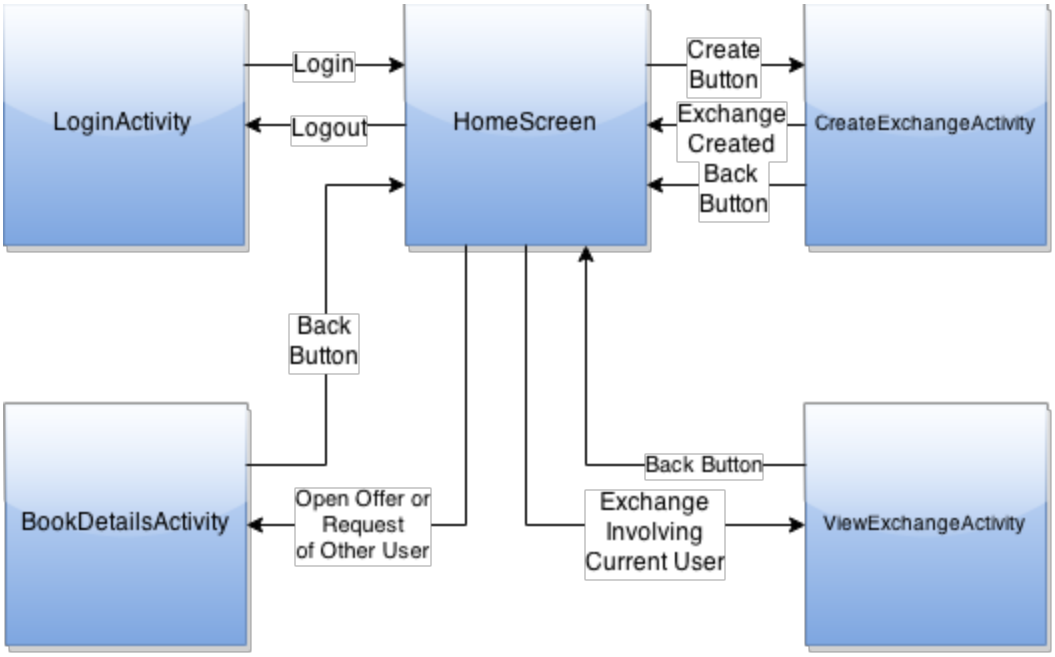

Figure No. 4: Activity and class interaction in Android Java code

HomeScreen:

- **Description:** This activity connects all the other activities and displays real-time information to the user about exchanges they are involved in, as well as offers and requests from other users.
- **Functionality:** Except for the list headers, everything on the `HomeScreen` is a button that directs the user to another activity.
- **Interactions:** Initially accessed from the `LoginActivity`. Connects to `CreateExchangeActivity`, `BookDetailsActivity`, and `ViewExchangeActivity`.

LoginActivity:

- **Description:** Entry point of the application.
- **Functionality:** Allows the application to uniquely identify users by requiring them to input an email address and password. Also allows user to create a new account, requiring a valid email and password confirmation.
- **Interactions:** Directs the user to the `HomeScreen`. Also communicates with the server to verify user credentials or create a new account.

CreateExchangeActivity:

- **Description:** Allows the user to create a new offer or request tied to a specific book.
- **Functionality:** Using the user input, searches for relevant related books on Goodreads and supplies those options for the user to choose from. Allows the user to choose to make either a request/offer and confirms that the user wishes to make this request.

- **Interactions:** Queries the server with the string the user types in the search box. When a book is selected, sends the `Book` object to the server to be added to the `books` table in the database. When the user confirms the creation of the exchange, sends the new Exchange object to the server to be added to the `exchanges` table in the database.

BookDetailsActivity:

- **Description:** Displays details of an exchange that the current user is not involved in.
- **Functionality:** The user can click a button to offer to loan the book (if the exchange is a request) or ask to borrow the book (if the exchange is an offer).
- **Interactions:** Queries the server to get the `Book` associated with the given exchange and creates a `GetImageTask` to fetch the book cover image from the url stored in the `Book` object. Contacts the server when the user submits an offer to loan or a request to borrow the book in question. This updates the Exchange Status in the database (and the app) to `REQUESTED` and sets the id of the current user to be the `borrower_id` or `loaner_id`, depending on the type of exchange.

ViewExchangeActivity:

- **Description:** This activity has a similar layout to the `BookDetailsActivity`, but it is designed to display additional information relevant to someone involved in the exchange in question.
- **Functionality:** Displays the current status of the exchange pertaining to the current user. For example, if a request to borrow has been accepted, the message “You may now borrow this book.” will be displayed. Allows the user to take action if another user has sent a request that requires a response. For example, the message “hrose@hmc.edu has offered to loan you this book. Would you like to accept the offer?" would be displayed along with Yes/No buttons that the user can select.
- **Interactions:** Queries the server to get the `Book` associated with the given exchange, and creates a `GetImageTask` to fetch the book cover image from the URL stored in the `Book` object. Informs the server of an update to the exchange when the user responds to a request.

ExchangeListAdapter:

- **Description:** Custom adapter to display a list of exchanges.
- **Functionality:** Extends the `ArrayAdapter` class to display a list of views formatted as described in the exchange_view xml file.
- **Interactions:** Used by the `HomeScreen` activity to display the lists under “My Current Exchanges” and “All Open Exchanges.”

BookListAdapter:

- **Description:** Custom adapter to display a list of books.
- **Functionality:** Extends the `ArrayAdapter` class to display a list of views formatted as described in the `book_view` xml file.
- **Interactions:** Used by `CreateExchangeActivity` to display the dropdown menu of books returned by the user query.

GetImageTask:

- **Description:** Creates a `Drawable` of the book cover image.
- **Functionality:** Extends the `AsyncTask` class and defines a method `doInBackground(URL... url)` which returns a `Drawable` created from the InputStream generated from the given URL (we always call with a single URL). Assumes that the given URL is valid.
- **Interactions:** Used by `BookDetailsActivity` and `ViewExchangeActivity` to fetch the book cover image from the URL stored in the associated `Book` object.

THE SERVER

The server is built out of several different modules, each which address a core part of the overall server functionality. While the actual code documentation will provide a better detailed description of the server on a class-by-class basis, here we overview the most important classes, their functionality, and how they interact with each other and the app. Figure No. 5 shows graphically how these pieces fit together.

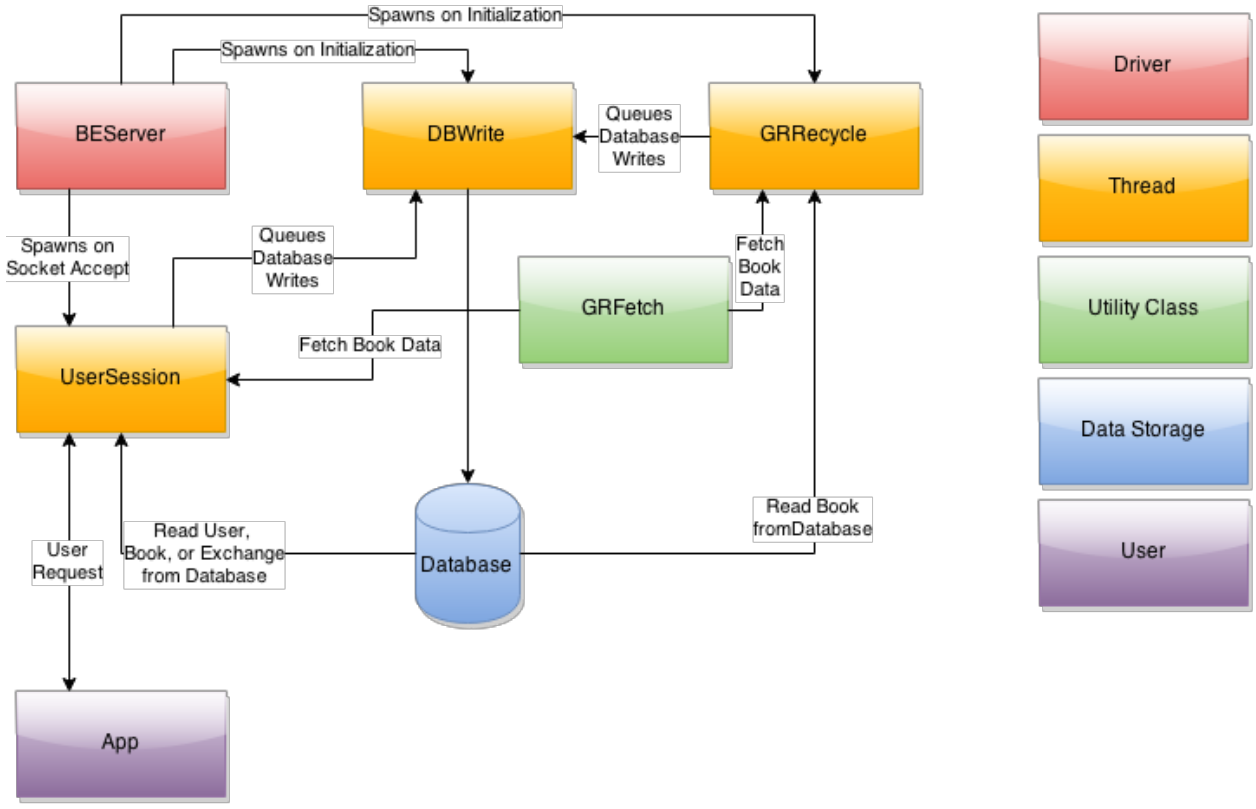

Figure No. 5: The different server modules and interactions

Below is a detailed description of each of the modules, their functionality, and how they interact with other modules.

Book Exchange App:

- **Description:** The user's Android app. Source of all user input.
- **Functionality:** Requests action from the server and awaits confirmation.
- **Interactions:** Does all communication with the server through the `Client` class in the app. Each call to `Client` starts by requesting a socket connection from the server's `BEServer` class. Once the `BEServer` class accepts the socket, a new `UserSession` thread is spawned. That `UserSession` thread handles all the remaining interaction with the Book Exchange app. The app sends its request by transferring a `Serializable` instance of the `Request` class over an `ObjectOutputStream`. It then awaits a confirmation `Request` object from the server over an `ObjectInputStream`. If the action requested of the server yields a response object, it is contained in the confirmation `Request` object.

BEServer:

- **Description:** The driver class for the server.
- **Functionality:** Handles initial startup of the server by spawning necessary threads. Afterwards, it listens to a `ServerSocket` for incoming connections from apps. If a connection is requested, it accepts the connection, spawns a thread to handle the request, and goes back to listening for incoming connections.
- **Interactions:** On startup, this class spawns the `DBWrite` and `GRRecycle` threads. Afterwards, it listens for incoming socket requests from apps. Each time an incoming socket is requested, BEServer will accept it and spawn a `UserSession` thread to handle the request.

UserSession:

- **Description:** The class for handling an accepted socket from an app.
- **Functionality:** Oversees the transfer of a `Request` object from an app via an `ObjectInputStream`. It then identifies what the `Request` object is requesting and handles it appropriately. Afterwards, it replies to the app by transferring the same `Request` object back as confirmation via an `ObjectOutputStream`. If the request yielded a return value, it is placed in the `Request.reply` field prior to responding.
- **Interactions:** After receiving the `Request` object from an app, the `UserSession` will handle the request accordingly. Depending on the request, this will mean it will either need to read information from the database in the form of `Book`, `Exchange`, or `User` object; fetch information in the form of `Book` objects from Goodreads via the `GRFetch` class; or queue something to be written to the database via the `DBWrite` thread.

GRFetch:

- **Description:** Single point for requesting any information from Goodreads via our developer key.
- **Functionality:** Synchronizes and accordingly paces Goodreads requests. This prevents violating the Goodreads API's "maximum 5 requests per second" clause. Also parses Goodreads's XML responses into a Book objects and and returns them.
- **Interactions:** Is called by the `UserSession` class to return `Book` objects. Directly interacts with the Goodreads API.

Database:

- **Description:** The database contains three tables: `users`, `books`, and `exchanges`.
- **Functionality:**
  - The `users` table is very simple, though it could be extended to store additional user information (e.g., separating username from email address). There are two fields: the unique integer `user_id` (primary key), and the text field `username` which currently stores the user email used for login. All valid user ids are strictly positive.
  - The `books` table contains key information about the books we parse from the Goodreads XML, including `book_title`, `author`, `isbn`, `pub_year`, and `image_url`. The primary key, `book_id`, is the unique id Goodreads uses to identify the book. Again, this table could also be extended to store more details, such as non-primary authors, number of pages, etc.
  - The `exchanges` table keeps track of all current and past exchanges. Each exchange has a unique `exchange_id` (the primary key), as well as the `book_id` and `book_title` of the associated book. The values of the fields `loaner_id` and `borrower_id` correspond to the user id of the appropriate user, or 0 if there is not yet a loaner/borrower. The creation of the exchange is recorded in `create_date`, and the `start_date` and `end_date` fields offer support for future implementation of due dates (see 'Limitations and Future Work' below).
- **Interactions:** Since SQLite allows for concurrent reads, both `UserSession` threads and `GRRecycle` threads directly request information from the database. Since SQLite does not support concurrent writes, only a single thread, `DBWrite`, writes to the database.

DBWrite:

- **Description:** Single point for writing any information to the database.
- **Functionality:** Synchronizes and ensures there are no concurrent database writes. This is needed because although SQLite databases allow for concurrent reads, as a file, it cannot handle concurrent writes. Other threads can queue data to be written by enqueuing the SQL command to a static `ConcurrentLinkedQueue`.
- **Interactions:** This class does not request any information from other threads. Rather, it simply waits for other threads, specifically `UserSession` threads and `GRRecycle` thread, to enqueue SQL commands to its static `ConcurrentLinkedQueue` for writing.

GRRecycle:

- **Description:** Watches over the `books` table of the database to ensure no content violates the Goodreads API requirements for maximum storage time.
- **Functionality:** Reads from the database and ensures no entry in the book table has existed for more than than 24 hours. If data has been in the database for at least 18 hours and is still needed, the `GRRecycle` thread re-requests the information from Goodreads and updates the necessary fields. If the data is no longer needed, it is simply deleted from the database.
- **Interactions:** When this thread needs to re-request information from Goodreads, it does so through the `GRFetch` class. It then writes the data to the database via the `DBWrite` thread.

IV. PRODUCT VERSUS PROPOSAL

The app that we produced met the key goals of our original proposal, though it lacked a few features that were difficult to implement or not well thought-out. Originally, we set three main tasks for our app: let users offer or request books, share contact information to facilitate the in-person loaning, and display exchange status as the exchange advances. All three of these tasks are present and functional in our final product, so our product achieves the core of our promised deliverables.

While our overall plan did not change, we did have to scale back the scope of this app. We did not account for how long it would take to properly set up the backend; there were three main features we had hoped to implement but had to leave out because of time constraints. The first such feature is the user profile, which was intended to display information about a specific user so that others could gain a sense of who they were borrowing from or loaning to. We also had planned to include due dates, but this ended up being slightly more complicated than anticipated. However, we did begin implementing this, so the basics of the code needed already exist in our repository. Finally, we had hoped to implement a feature to create and interact within user groups, so that people at a specific college or in a book group together could easily connect. However, this had always been a stretch goal, so its absence does not affect our promised product. All of these features are discussed in more detail in the 'Future Work' section below.

One feature that we added was error messages for user actions, which is an important principle of good UX. For example, the app alerts the user if the password they type is too short or if they forgot to complete a field, which enhances the user experience by letting the user know why the app is not behaving as they might expect. Figure No. 6 shows an example error message.

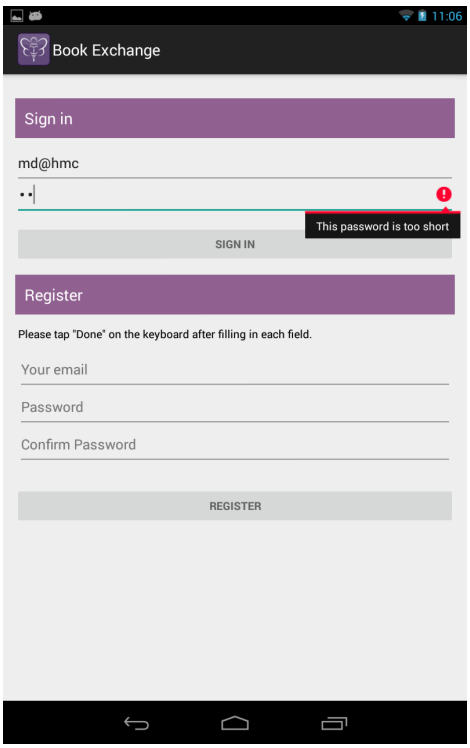

Figure No. 6: An error message informing the user that the password submitted is too short

V. PROJECT CHALLENGES

During this project, we faced challenges both in regards to team cohesion and technical issues. However, we worked through all of these to produce our final app. In regards to team dynamics, originally we had some problems with communication. This was detrimental as it affected our understanding of our goals, gave us unrealistic expectations of what had been accomplished, and generated confusion. To resolve this, we talked as a group and more explicitly set a standard for what level of communication was expected. For example, sometimes it is impossible to avoid being late for a meeting, or even being unable to attend altogether. We decided that in this cases, it is necessary to text or email the other team members and notify them of the lateness/non-attendance ASAP. We also were more intentional as the semester progressed about clearly deciding who was assigned to what specific task and regularly checking in on their progress.

Another difficulty we all faced at some point was getting through the learning curve of a project like this, either in Android Studio or in setting up the server. In regards to Android development, learning issues ranged from figuring out the quirks of the IDE to understanding how to effectively use all of the different XML tags. A deeper understanding of XML was also necessary to parse the Goodreads API. In regards to the server, that was a completely new venture, which required a level of technical expertise none of us had at the beginning of this project. To address our lack of prior knowledge, we used extensive googling and took advantage of grutoring hours. We also talked to various professors who had specialized knowledge relevant to our tasks. Finally, we often worked on our tasks in a group setting, so that we could ask each other questions and pair program when helpful.

Our two biggest technical challenges were server hosting and the login system. We originally hosted the server on a computer in one of our team member's houses. However, every time his mother cleaned his room, there was a possibility of her unplugging his computer and thus causing our system to go down. We then switched to a more reliable option and are now hosting our app on Knuth. For the login system, we had originally planned to use the Google+ API, allowing users to login with a Google+ account. However, we ran into some difficulties using Google+, and decided that it was pushing the scope of this project too far, so switched to our own login/registration system. (If we were to continue work on this project, we would like to re-introduce the Google+ API.) In the end, our own system was probably comparable work to implement, but it was a good learning experience for all of us, and adds polish to our final product.

The last major challenge we faced was integration of the app. The way our timeline worked out, we were only able to fully integrate UI with server and database two weeks before the code freeze. Part of why this took so long was because we were not specific enough about our design for connecting classes such as the Book and Exchange objects. If we had agreed on the design of those objects earlier, we could have prepped the UI more effectively for connection with the server. Thankfully, we were still able to integrate on time, so this challenge resolved itself. However, in future projects, we will be sure to define this data earlier to smooth and quicken the integration process.

VI. LIMITATIONS AND FUTURE WORK

The current version of our application is entirely functional and presents a positive user experience. However, there are some pieces of our design that could be improved or extended, many of which are are within our control and could be fixed in future iterations.

Our biggest limitation is that we are only allowed to issue one query per second to Goodreads. If we paid for the use of their API, we could have a much higher query rate, allowing us to search for more books in less time and improve the menu of book suggestions in the `CreateExchangeActivity`. It would also be possible (regardless of the query speed) to integrate the app further with Goodreads by allowing users to sign in with a Goodreads account and/or displaying book reviews from Goodreads.

Although the application lacks thorough unit tests, we developed the application with an awareness of user error, so many methods check for and handle edge cases, null values, etc. Each `Activity` handles only local views and objects, while classes like the `ExchangeListAdapter` and `GetImageTask` are abstracted out as appropriate.

Here we list the aspects of the application that could be improved in future iterations:

- **Testing:**
  - **Application:** Unfortunately, the nature of the Android `Activity` lifecycle means that implementing even basic UI tests often requires extensive set-up and mocking, since so many methods depend on the application Context and information from the previous `Activity`. As a result, most testing of the application has been manual thus far, though the `Client` class has been thoroughly unit tested as part of the server-side testing. We have already incorporated feedback from user testing into our design during Phase 2 and Phase 3; future iterations would require more user testing.
  - **Server**: Since the server was developed in modules, as described in the 'System Architecture' section, each section was run and tested independently. This was achieved through a series of rigorous JUnit tests and debugging `main()` methods. For instance, the `Client` class, although used in the app to connect to the server, can also be run with a temporary `main()` method that allows it to be an independent Java program rather than part of an entire Android app. Similarly, a JUnit test queues objects to write to the database and then reads them back to ensure they do not alter during this process.
- **Multithreading:** In order to optimize performance, it would be ideal for all requests to the server to be run on separate threads so the main UI thread would never be hampered by network issues or unexpected server-side errors (although the server should handle most errors).
- **Notification System:** There is currently a very basic notification system that displays a "response requested" message as part of the text in the list of user exchanges when the current user has not yet accepted or rejected a request from another user. With some changes to the `Exchange` class and `ViewExchangeActivity`, the application could instead determine if the status of an exchange has changed since the user last saw it and provide an appropriate visual cue.

- **Listener Improvement:** Currently, our `EditText` listeners (found when registering a new user or searching for a book) use an `IMEActionListener`. This requires the user to hit a "search" or "done" button in order for the app to record the text they have inputted. This could be better handled if we also added an `onFocusChange` listener (although that would require a different type of `View`), which would recognize when the user is done editing a field and has moved onto a new view, removing the need for a "done" or "search" button.
- **Refined Goodreads XML Parsing:** Our current strategy for parsing the Goodreads XML to gather book information relies somewhat on the consistency of the Goodreads XML formatting and could be made more robust. In addition, certain queries can sometimes return an author instead of a book, which we should filter out. Finally, we would like to offer better support for books with unusual characters in the title.
- **Implement Server-based Database:** For simplicity, we implemented our database using SQLite, which stores the database as a single file. In reality, it would be necessary to use a standard SQL database, which would be hosted on the server, since SQLite does not support concurrent writes.
- **Sanitize SQL Inputs:** It would be more robust to use the PreparedStatement class rather than the Statement class to generate SQL statements, ensuring that no SQL commands are accidentally injected into the database.
- **Improve Server-side Error Checking:** Currently, the server assumes most of the information it is provided with is valid. Although this can be assumed, since there are error checks performed app side, this is not the safest behavior for the server. If the app was to become bugged or a malicious user were to simulate an app connection, they could potentially provide invalid input to the server or database. A more rigorous server side input checking scheme would help protect against unintended behavior, such as writing null values into the database.
- **Google+ API:** The ability to login with a Google+ account could be reintroduced and debugged. The base code needed for this exists in our repository.
- **User Information Security:** The login information is currently passed as a string through a basic socket, so none of it is encrypted. A malicious user could hack the app to provide a false `user_id`.
- **Remember Logged-In Users:** Every time the user restarts the app, the user is required to log in again. We would like to be able to store some login information locally (likely in a `SharedPreferences` object) so that the user does not have to regularly re-login.

Here, we list the features that do not currently exist in the app but whose addition would improve overall user experience:

- **Due Date:** As mentioned above (see 'Product Versus Proposal'), we have some of the code to allow the owner of a book to specify a due date, such as the UI for a date picker and `start_date` and `end_date` fields in the `exchanges` table. With the addition of due dates, we could remind users when it is time to return the book. However, we ended up tabling this feature in order to focus on tasks with a higher priority.
- **User Groups:** As discussed earlier (see 'Product Versus Proposal'), user groups could help users connect with people in similar situations together. These groups could be based on

location, university, or any other organization affiliation. Currently no code exists to support this feature.

- **User Profile:** As previously discussed (see 'Product Versus Proposal'), a user profile would help other users gain a sense of who they are interacting with. Implementing a user profile would require extension of our user class along with our registration system.
- **Wishlist:** Given the request-based nature of this app, it would be useful for users to keep a running list of books they wish to borrow. Then, whenever one of those books is offered, the user would receive a notification so they could quickly request it. No code exists currently to support this feature, but it could be added as an extension of the Create Request/Offer feature. There would then need to be a new activity to display and edit the current wishlist.

APPENDIX A - PHASE #1 GOALS AND PROGRESS

Overall Goal: To complete the separate components of the application.

Priority level is rated 1 to 5, where 1 is unimportant and 5 is absolutely critical. Similarly, difficulty is rated 1 to 5, where 1 is trivial and 5 is very time intensive.

Completed Subgoals:

- Investigate Google+ API
  - Priority level: 5
  - Difficulty: 2
- Investigate Goodreads API
  - Priority level: 5
  - Difficulty: 2
- Develop user workflow
  - Priority level: 5
  - Difficulty: 3
- Create loan feature UI
  - Priority level: 5
  - Difficulty: 3
- Draft UI layout
  - Priority level: 3
  - Difficulty: 2
- Investigate database options
  - Priority level: 5
  - Difficulty: 2
- Create logo
  - Priority level: 1
  - Difficulty: 2

Backlogged Subgoals:

- Integrate Google+ API
  - Priority level: 5
  - Difficulty: 5
- Integrate Goodreads API
  - Priority level: 5
  - Difficulty: 5
- Implement user profile
  - Priority level: 5
  - Difficulty: 3
- Implement search feature
  - Priority level: 5
  - Difficulty: 4
- Create groups
  - Priority level: 3
  - Difficulty: 4
- Set up database
  - Priority level: 5
  - Difficulty: 5
- Add main activity UI to the app
  - Priority level: 5
  - Difficulty: 3

Summary of Progress: In-depth research regarding API usage, extensive flow planning, and first steps in UI implementation.

APPENDIX B - PHASE #2 GOALS AND PROGRESS

Overall Goal: To integrate the separate components of the application.

Priority level is rated 1 to 5, where 1 is unimportant and 5 is absolutely critical. Similarly, difficulty is rated 1 to 5, where 1 is trivial and 5 is very time intensive.

Completed Subgoals:

- Demonstrate a working example using the Google+ API to log in
  - Priority level: 5
  - Difficulty: 5
- Demonstrate using the Goodreads API to pull information regarding a book
  - Priority level: 5
  - Difficulty: 5
- Set up the database
  - Priority level: 5
  - Difficulty: 5
- Make the UI for the Book Details (Exchange) View
  - Priority level: 5
  - Difficulty: 3
- Add java functionality to XML buttons
  - Priority level: 5
  - Difficulty: 2
- Demonstrate that the server code is functional
  - Priority level: 5
  - Difficulty: 5
- Add UI for the main screen to the app
  - Priority level: 5
  - Difficulty: 3
- Make SQL statements
  - Priority level: 5
  - Difficulty: 3
- Parse Goodreads book data using XML

Backlogged Subgoals:

- Complete socket code
  - Priority level: 5
  - Difficulty: 4
- Implement search feature
  - Priority level: 5
  - Difficulty: 4
- Load an image in the app from a url
  - Priority level: 3
  - Difficulty: 3
- Check that user input is valid when searching for books
  - Priority level: 3
  - Difficulty: 1
- Fully integrate app with server
  - Priority level: 5
  - Difficulty: 5

○ Priority level: 5 ○ Difficulty: 4 ● Make confirmation pop-ups for important user actions ○ Priority level: 3 ○ Difficulty: 2	

Summary of Progress: Creation of all of the separate components (UI, Goodreads interaction, Server) and testing of the separate components.

APPENDIX C - PHASE #3 GOALS AND PROGRESS
Goal: Test
Actuality: Integrate, Test, and Polish

Overall Goal: Integrate, test, and polish the final application.

Priority level is rated 1 to 5, where 1 is unimportant and 5 is absolutely critical. Similarly, difficulty is rated 1 to 5, where 1 is trivial and 5 is very time intensive.

Completed Subgoals:

- Complete socket code
  - Priority level: 5
  - Difficulty: 4
- Implement search feature
  - Priority level: 5
  - Difficulty: 4
- Load an image in the app from a url
  - Priority level: 3
  - Difficulty: 3
- Check that user input is valid when searching for books
  - Priority level: 3
  - Difficulty: 1
- Fully integrate app with server
  - Priority level: 5
  - Difficulty: 5
- Add descriptive comments to codebase
  - Priority level: 5
  - Difficulty: 2
- Remove settings button and menu
  - Priority level: 2
  - Difficulty: 1
- Make a login and registration system
  - Priority level: 5
  - Difficulty: 4
- Add logout button
  - Priority level: 5
  - Difficulty: 1
- Unify color scheme and UI layout
  - Priority level: 3

    - Difficulty: 2
- JUnit Server Testing
    - Priority level: 5
    - Difficulty: 3
- JUnit Client Testing
    - Priority level: 5
    - Difficulty: 3
- Drop-down menu of book suggestions
    - Priority level: 4
    - Difficulty: 4
- Border on picture on Book Details page
    - Priority level: 1
    - Difficulty: 1

Summary of Progress: We integrated, tested, and polished the final app. Everything we did not accomplish has been recorded in the 'Limitations and Future Work' section.