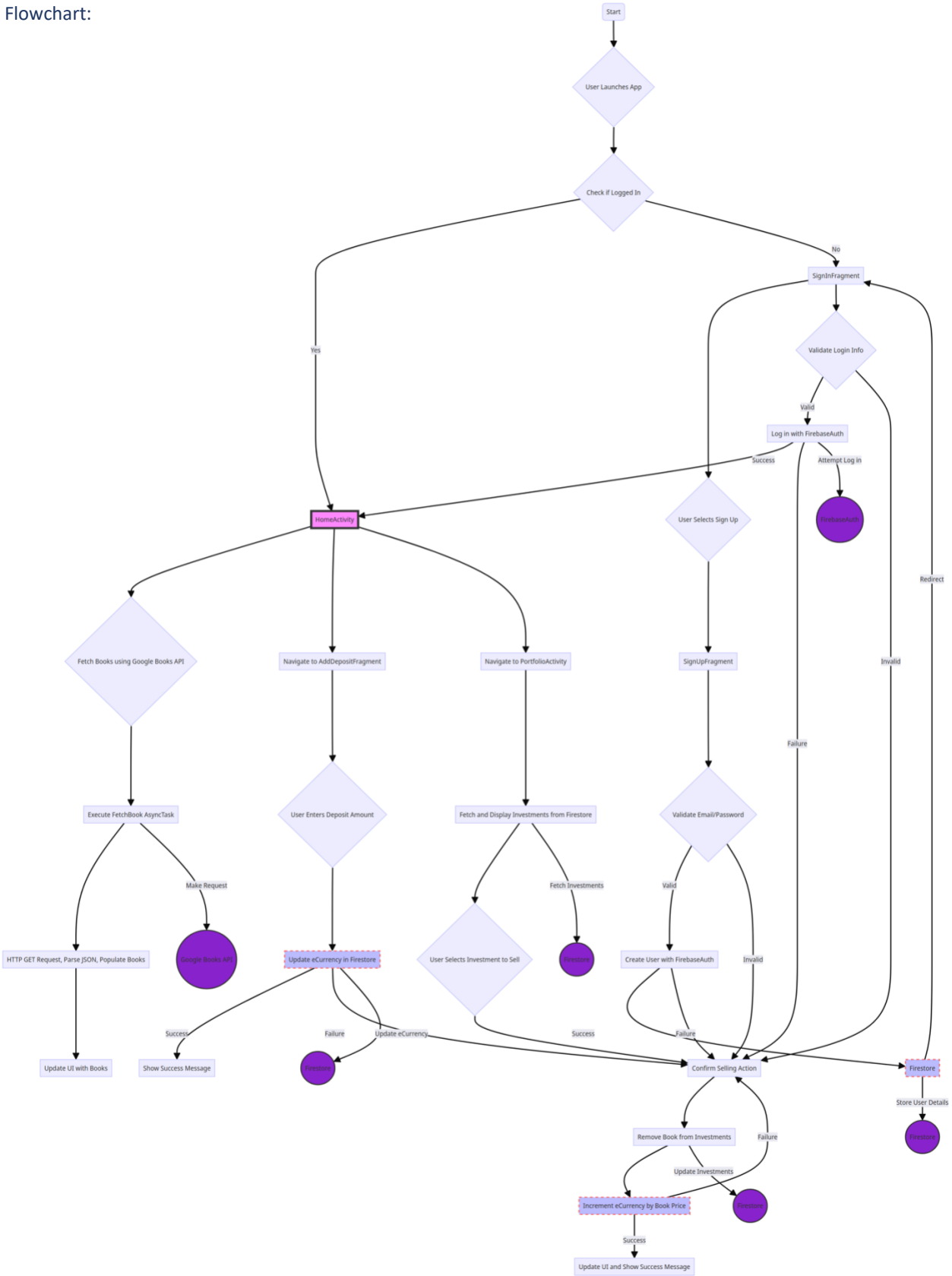
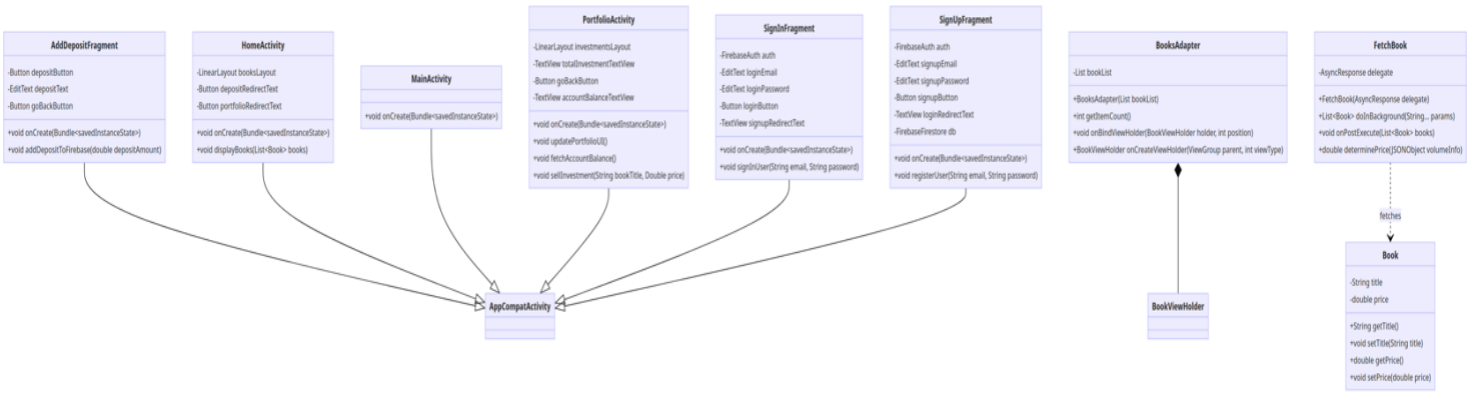
CRITERION B: DESIGN

Flowchart:



UML DIAGRAM


AddDepositFragment
-Button depositButton
-EditText depositText
-Button goBackButton
+void onCreate(Bundle<savedInstanceState>)
+void addDepositToFirebase(double depositAmount)
HomeActivity
-LinearLayout booksLayout
-Button depositRedirectText
-Button portfolioRedirectText
+void onCreate(Bundle<savedInstanceState>)
+void displayBooks(List<Book> books)
MainActivity
+void onCreate(Bundle<savedInstanceState>)
PortfolioActivity
-LinearLayout investmentsLayout
-TextView totalInvestmentTextView
-Button goBackButton
-TextView accountBalanceTextView
+void onCreate(Bundle<savedInstanceState>)
+void updatePortfolioUI()
+void fetchAccountBalance()
+void sellInvestment(String bookTitle, Double price)
SignInFragment
-FirebaseAuth auth
-EditText loginEmail
-EditText loginPassword
-Button loginButton
-TextView signupRedirectText
+void onCreate(Bundle<savedInstanceState>)
+void signInUser(String email, String password)
SignUpFragment
-FirebaseAuth auth
-EditText signupEmail
-EditText signupPassword
-Button signupButton
-TextView loginRedirectText
-FirebaseFirestore db
+void onCreate(Bundle<savedInstanceState>)
+void registerUser(String email, String password)
BooksAdapter
-List bookList
+BooksAdapter(List bookList)
+int getItemCount()
+void onBindViewHolder(BookViewHolder holder, int position)
+BookViewHolder onCreateViewHolder(ViewGroup parent, int viewType)
FetchBook
-AsyncResponse delegate
+FetchBook(AsyncResponse delegate)
+List<Book> doInBackground(String... params)
+void onPostExecute(List<Book> books)
+double determinePrice(JSONObject volumeInfo)
fetches
Book
-String title
-double price
+String getTitle()
+void setTitle(String title)
+double getPrice()
+void setPrice(double price)
AppCompatActivity
BookViewHolder

ALGORTHIMS

In my application, various algorithms are deployed for executing smaller tasks. For a detailed explanation of these tasks, you can refer to Criterion C where each function is elucidated. Presented below is a code snippet that was used to determine the initial price/value of a given book on the exchange. It uses the Google Books API to retrieve data.

```
1 usage
public double determinePrice(JSONObject volumeInfo) {
    double averageRating = volumeInfo.optDouble( name: "averageRating");
    int ratingCount = volumeInfo.optInt( name: "ratingsCount");

    // Perform the calculation
    double calculatedPrice = Math.pow((averageRating + ratingCount), 1.6);

    // Round to 2 decimal places
    BigDecimal price = BigDecimal.valueOf(calculatedPrice);
    price = price.setScale( newScale: 2, RoundingMode.HALF_UP);

    return price.doubleValue();
}
```

DATA STRUCTURES

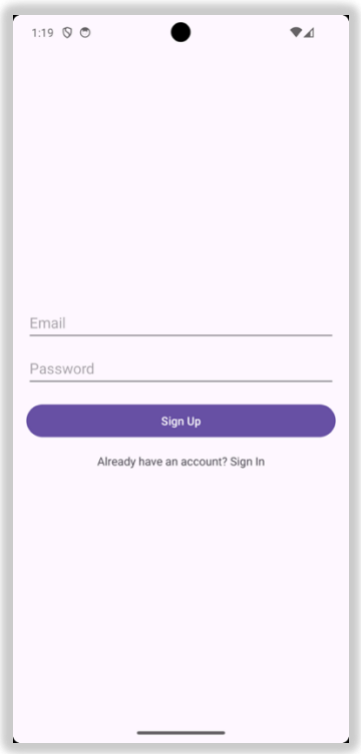
The core of my application's data handling revolves around Google's Cloud Firestore. I employ Firestore's structured approach to store data unique to each user, which is organized within a primary 'users' collection. Each user then has dedicated sub-collections containing their personal information regarding investments. The decision to utilize Cloud Firestore was influenced by popularity (therefore extensive documentation) and seamless integration with Java. This choice proved to be particularly prudent given our project's timeline and the nature of the application. The UI of the app functions by retrieving data from Firestore, encapsulating it into a list of maps that are subsequently sorted.

TEST PLAN

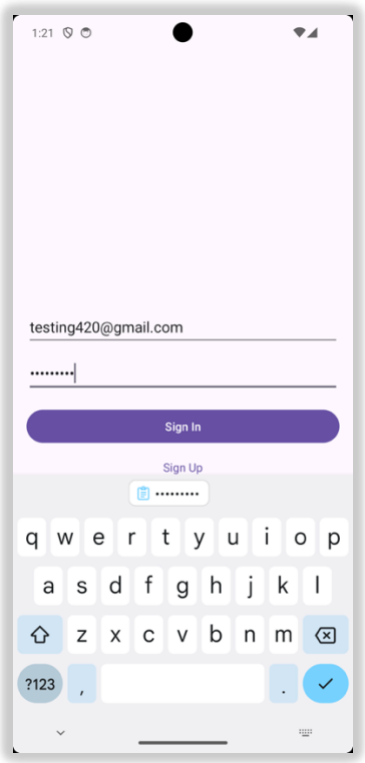
Success Criteria	Test Case
1. High Priority: Users must be able to create an account with a valid email and password with at least 8 characters.	- Attempt to register with an invalid email and expect failure.- Attempt to register with a password shorter than 8 characters and expect failure. - Register with a valid email and password (8+ characters) and expect success.
2. High Priority: Users must be able to browse through different books.	- Navigate to the book browsing section after login and ensure a list of books is displayed.- Scroll through the list to verify responsiveness and loading of book data.
3. High Priority: The system must allow users to make investments but validate each investment action to ensure the user has sufficient virtual currency after transaction arithmetic.	- Try to make an investment without sufficient virtual currency and expect rejection.- Add sufficient currency and then make an investment, expecting success.- Verify the deduction from the virtual currency accurately reflects the investment made.
4. High Priority: The app must update investment values with data fetched from verified sources and APIs every instance.	- Check the investment values at different times to verify they update based on real-world data.- Compare the app's investment values against the data source for accuracy.
5. Medium Priority: The app should allow users to add any amount of currency to their account.	- Add a small amount of currency to the account and verify the balance updates correctly. - Add a large amount of currency and verify the balance updates correctly.
6. Medium Priority: The app should allow users to view and manage their current investments, including amounts invested and current virtual currency value.	- After making multiple investments, view the portfolio to ensure all investments are listed with the correct amounts.- Verify that the total investment value and current virtual currency balance are displayed and accurate.

User Interface Flow

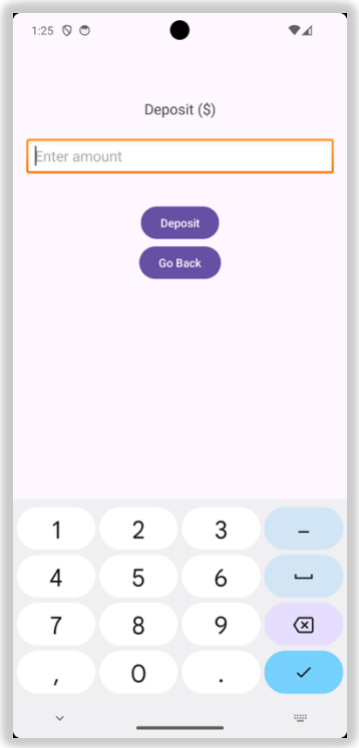
Sign Up Page



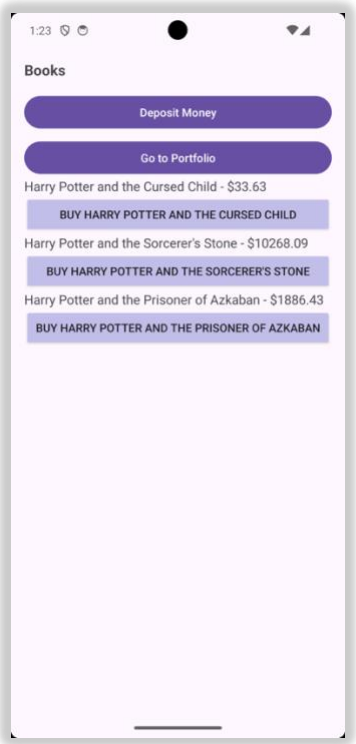
Sign In Page



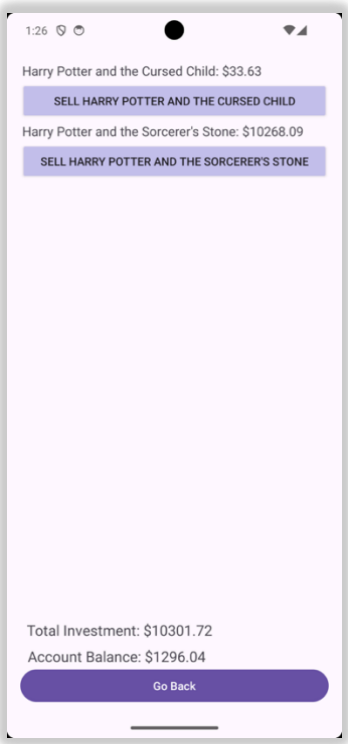
Deposit Page



MarketPlace Page



Portfolio Page



Word Count: 197