CRITERION A: PLANNING

SCENARIO:

My client is Timothy Edward Gray, and my advisor is Sandesh Tilapudi, who has 15+ years of experience as a Software Engineer. His expertise within Java Android Development will provide me with valuable guidance. My client has been an avid “betting man” for a great deal of time. He has made harmless wagers on pop culture phenomena but has yet to find a way to apply this passion in a tangible manner. In our initial consultation, he expressed desire in a mobile application that allows him to “invest” in upcoming books. In our first consultation, he requested these specific features:

1. Being able to invest in books using virtual currency.
2. Valuation of cultural products fluctuates based on product’s real world popularity and public interest, simulating a real market.

Evidence of this consultation can be found in the appendix (Appendix B)

Main Takeaway (Problem): Client is gifted in the art of predicting but has not found a way to gamify this. Absence of a competitive, measurable way to gauge and reward accurate predictions in the realm of literature success.

RATIONALE:

Solution: The proposed solution is a mobile application developed for Android, utilizing Java and Firebase. This platform will enable users to "purchase" shares in books using virtual currency.

The platform will be an Android application, developed in Java. The solution is appropriate, as my client expressed desire for an on-the-go app for accessibility. I will be using Android Studio and a Firebase backend. Firebase’s real-time database will be especially useful for the constant, frequent, fetching from APIs to evaluate the popularity (and hence price) of products. The real-time database allows for instant updates to investment values based on media performance metrics. This development process will make me more proficient in the Android development environment. Furthermore, I will learn how to use databases and practicing backend development are some things I have not explored before, allowing me to expand my computer science knowledge.

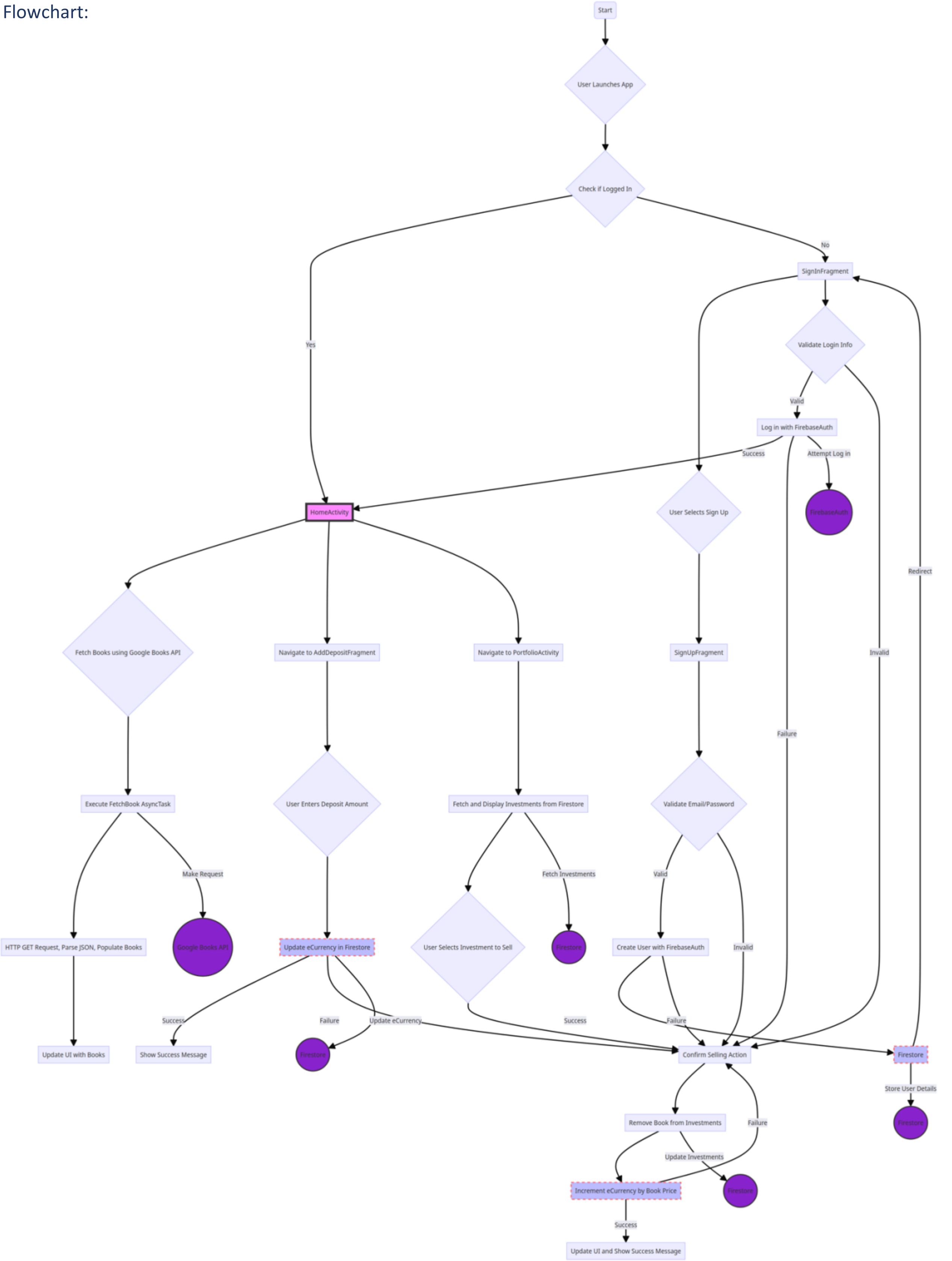
Word Count: 377

SUCCESS CRITERIA:

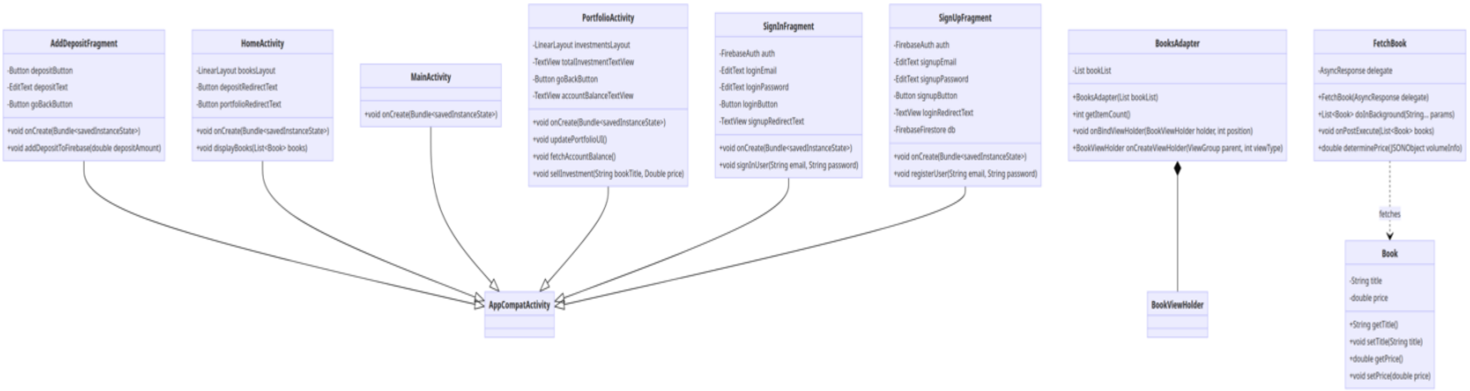
1. High Priority: Users *must* be able to create an account with a valid email and password with at least 8 characters and use it to sign in.
2. High Priority: Users *must* be able to browse through different books
3. High Priority: The system *must* allow users to make investments but validate each investment action to ensure the user has sufficient virtual currency after transaction arithmetic.
4. High Priority: The app *must* update investment values with data fetched from verified sources and APIs every instance.
5. Medium Priority: The app *should* allow users to add any amount of currency to their account.
6. Medium Priority: The app *should* allow users to view and manage their current investments, including amounts invested and current virtual currency value.
7. Low Priority: When user closes application, the system *should* log the user off, and prompt them to relog in for the next instance.

CRITERION B: DESIGN

Flowchart:

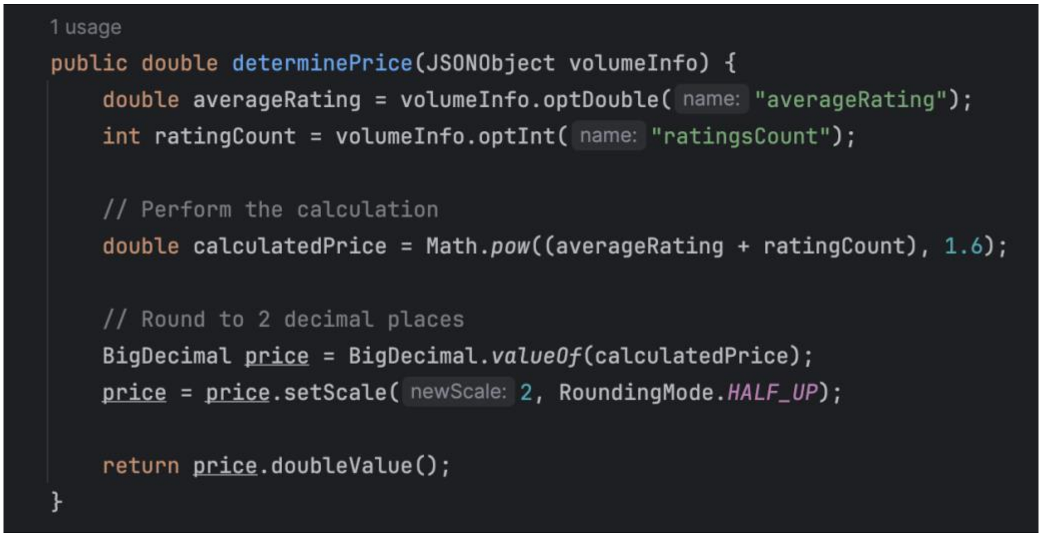


UML DIAGRAM


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
AppCompatActivity
BookViewHolder
Book
-String title
-double price
+String getTitle()
+void setTitle(String title)
+double getPrice()
+void setPrice(double price)

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.

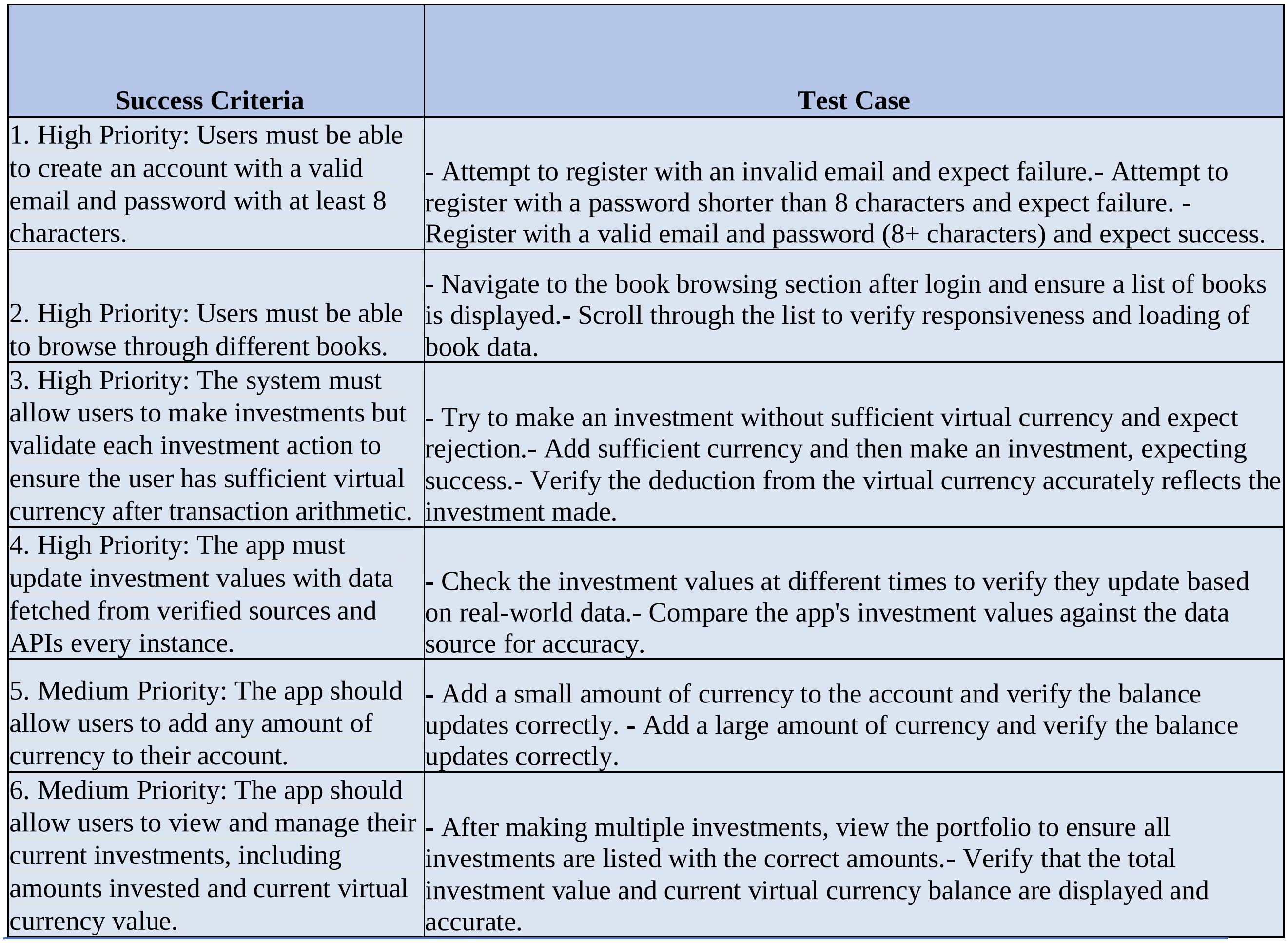


```
1 usage
public double determinePrice(JSONObject volumeInfo) {
    double averageRating = volumeInfo.optDouble( name: "averageRating");
    int ratingCount = volumeInfo.optInt( name: "ratingsCount");

    // 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.

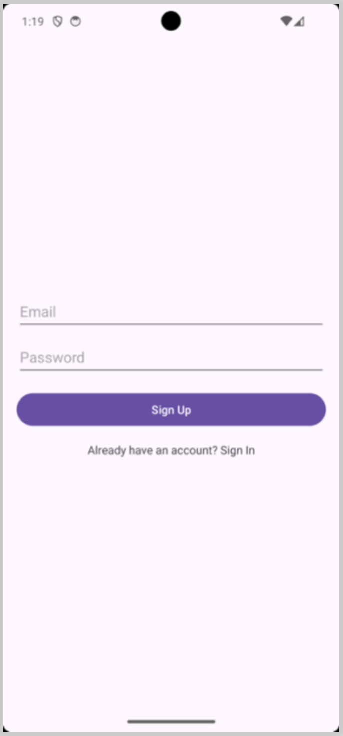
TEST PLAN



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.

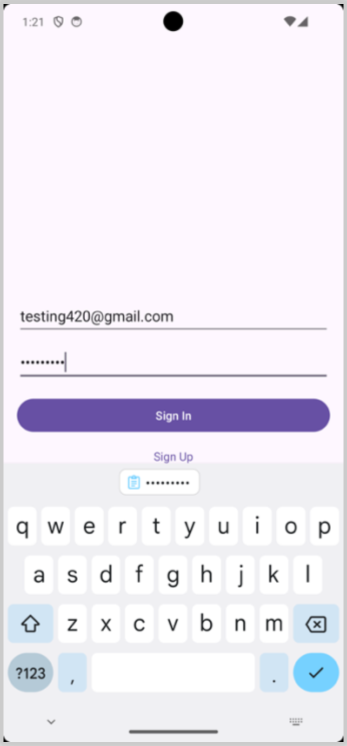
User Interface Flow

Sign Up Page



Already have an account? Sign In

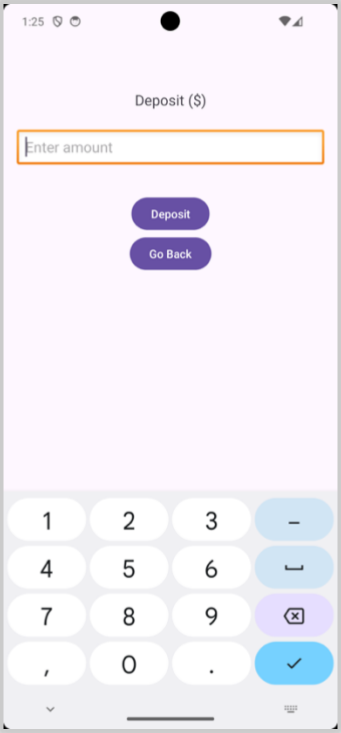
Sign In Page



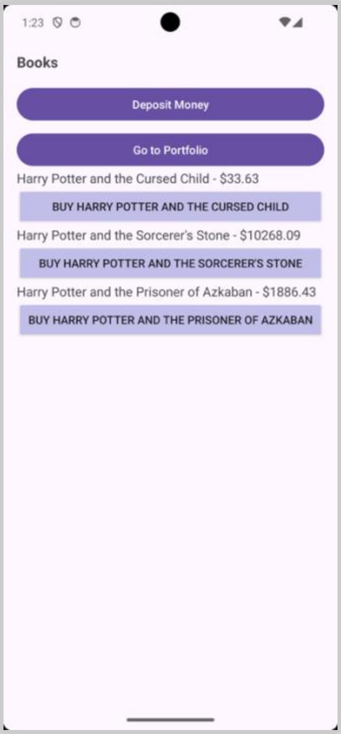
testing420@gmail.com

•••••••••

Deposit Page



MarketPlace Page

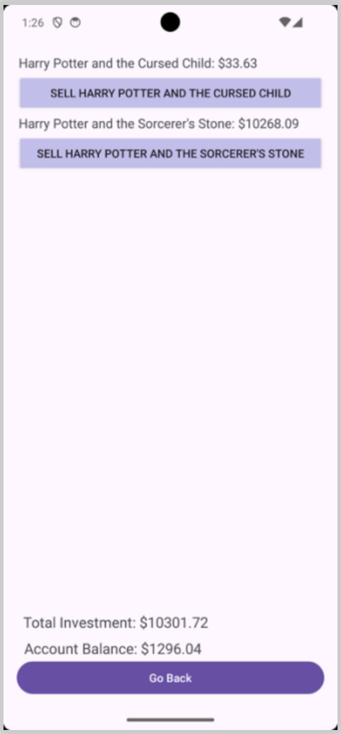

1:23
Books
Deposit Money
Go to Portfolio
BUY HARRY POTTER AND THE CURSED CHILD
BUY HARRY POTTER AND THE SORCERER'S STONE
BUY HARRY POTTER AND THE PRISONER OF AZKABAN

Harry Potter and the Cursed Child - $33.63

Harry Potter and the Sorcerer's Stone - $10268.09

Harry Potter and the Prisoner of Azkaban - $1886.43

Portfolio Page


1:26
SELL HARRY POTTER AND THE CURSED CHILD
SELL HARRY POTTER AND THE SORCERER'S STONE
Go Back

Harry Potter and the Cursed Child: $33.63

Harry Potter and the Sorcerer's Stone: $10268.09

Total Investment: $10301.72

Account Balance: $1296.04

Word Count: 197

CRITERION B: RECORD OF TASKS

Task Number	Planned Action	Planned Outcome	Time Estimated ()	Target Completion Date	Criterion
1	Initial consultation with client	Understand client requirements and desired features for the app	0.5 hours	2/2/24	A
2	Define project scope and objectives	A clear project scope and a list of objectives based on client's needs	3 hours	2/2/24	A
3	Select technology stack	Decide on Android (Java), Firebase, and APIs for development	0.5 hours	2/2/24	A
4	Create a project timeline and milestones	A detailed project plan with key milestones and deadlines	3 hours	2/3/24	A
5	Design database schema for user registration	A structured Firebase Firestore database schema	2 hours	2/3/24	B
6	Draft initial app design and layout concepts	Preliminary app designs for client review	2 hours	2/4/24	A
7	Implement FirebaseAuth user signup in SignUpFragment	Users can register with email and password	1 hours	2/4/24	B
8	Design SignUpFragment UI	A user-friendly registration interface	0.5 hours	2/5/24	B
9	Implement input validation for SignUpFragment	Secure and validated user input	0.5 hours	2/6/24	B
10	Test FirebaseAuth integration	Successful user registration in Firebase	2 hours	2/7/24	B

11	Design SignInFragment UI	A user-friendly login interface	3 hours	2/10/24	B
12	Implement input validation for SignInFragment	Secure and validated user login input	1 hours	2/11/24	B
13	1st Draft	Complete First Draft	0.5 hours	2/12/24	B
14	2nd draft	complete second draft	0.5 hours	2/12/24	B
15	Implement logout functionality	Users can securely log out from their account	0.5 hours	2/13/24	B
16	Implement AddDepositFragment functionality	Users can add virtual currency to their account	2 hours	2/15/24	B
17	Design AddDepositFragment UI	A user-friendly interface for adding virtual currency	3 hours	2/16/24	B
18	Test deposit functionality	Users can add virtual currency without issues	1 hours	2/17/24	B
19	Implement PortfolioActivity functionality	Users can view their investments	4 hours	2/19/24	B
20	Design PortfolioActivity UI	A user-friendly interface for viewing investments	3 hours	2/20/24	B
21	Test portfolio viewing functionality	Users can view their investments without issues	2 hours	2/21/24	B
22	Implement investment buying functionality	Users can invest in books using virtual currency	1 hours	2/23/24	B
23	Test investment buying functionality	Investment transactions are processed correctly	2 hours	2/24/24	B

24	Implement real-time investment value updates	Investment values update based on real-world data	1 hours	2/26/24	B
25	Test real-time investment value updates	Investment values accurately reflect real-world changes	0.5 hours	2/27/24	B
26	1st Draft	Complete First Draft	1 hours	3/1/24	C
27	Design HomeActivity UI	A user-friendly home interface with book investments	1 hours	3/1/24	B
28	Implement book search functionality in HomeActivity	Users can search for books to invest in	1 hours	3/3/24	B
29	Test book search functionality	Book search returns accurate results	0.5 hours	3/4/24	B
30	Implement Firebase Firestore security rules	Secure database interactions	1 hours	3/5/24	B
31	Test overall app security	App interactions with Firebase Firestore are secure	0.5 hours	3/6/24	B
32	Users can sell their book investments	A working PortfolioActivity.	2 hours	3/7/24	B
33	Test investment selling functionality	Investment selling transactions are processed correctly	2 hours	3/7/24	B
34	Implement error handling for transactions	Robust error handling for buying and selling investments	3 hours	3/7/24	C
35	Test error handling in transactions	Transactions gracefully handle errors without crashing the app	1 hours	3/7/24	C
36	Optimize Firestore queries	Improved efficiency and reduced latency in database interactions	0.5 hours	3/7/24	C

37	Test Firestore query optimizations	Firestore interactions are faster and consume less bandwidth	0.5 hours	3/8/24	C
38	Implement user session management	Secure management of user sessions with auto logout on app close	1 hours	3/9/24	C
39	Test user session management	User sessions are securely managed, with auto logout functioning	0.5 hours	3/9/24	C
40	Optimize UI/UX for SignIn and SignUp Fragments	Enhanced user experience for registration and login	0.5 hours	3/9/24	C
41	Test UI/UX optimizations	Users find the app more intuitive and easier to navigate	1 hours	3/10/24	C
42	Implement feedback mechanism for investment actions	Users receive feedback on the success or failure of investment actions	1 hours	3/10/24	C
43	Test feedback mechanisms	Feedback mechanisms for investments are informative and user-friendly	0.5 hours	3/12/24	C
44	Implement background task for updating book values	Book values are updated in the background without user intervention	1 hours	3/13/24	C
45	Test background task for book value updates	Background updates of book values are reliable and efficient	2 hours	3/14/24	C
46	Finalize all UI/UX elements across the app	A cohesive and engaging user interface and user experience	1 hours	3/14/24	C
47	Conduct comprehensive UI/UX testing	The app provides a seamless and intuitive user experience	0.5 hours	3/14/24	C
48	Implement and test data persistence	User data and state are persisted across app sessions	4 hours	3/14/24	C

49	Create demonstration video (Criterion D)	A video showcasing the app's functionality and features	2 hours	3/15/24	D
50	Make executible version	Complete Compliation	0.5 hours	3/16/24	NA
51	Evaluation	Use client feeback to complete evaluation	2 hours	3/16/24	E
52	Prepare product +documentation zip	Review project to ensure it fits requirements	1 hours	3/17/24	NA

Total Hours Spent 72.5

CRITERION C: DEVELOPMENT

TECHNIQUES:

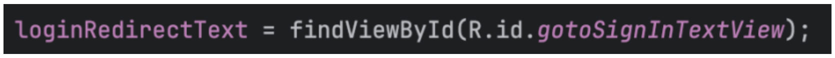
Some key techniques used in my project are:

- Dynamic UI Updates and Navigation
    - Use of listeners and navigation methods to facilitate data passing between activities and fragments
        - Inheritance
    - UpdatePortfolio Algorithm
        - Atomic Operations
        - Document Snapshots
- Firebase Firestore Integration
    - Firebase Authentication
    - Firebase Entries looped for entries using Map
    - Firebase Transactions
- Data Model and Google Books API Integration
    - Using AsyncTask for making non-blocking network requests.
    - List<Book> to hold Book objects received from API
        - Book Class
        - Encapsulation
    - Composite Data Structure in FetchBook: JSONObjects
    - Multi-Dimension JSONArrays
    - Fetching and parsing data from the Google Books API
        - Parsing text streams in JSONObjects
    - Algorithm for Valuation of Books

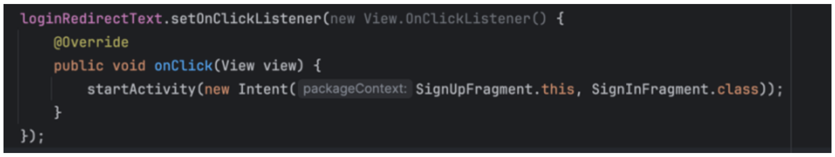
1. Dynamic UI Updates and Navigation

Utilizing a dynamic UI is essential as the price of books shares are constantly fluctuating. I accomplished this through the use of activities, fragments, and XML files in Android Studio. The purpose of using these over other methods are their seamless integration with Java, especially in Android development.

Navigation:



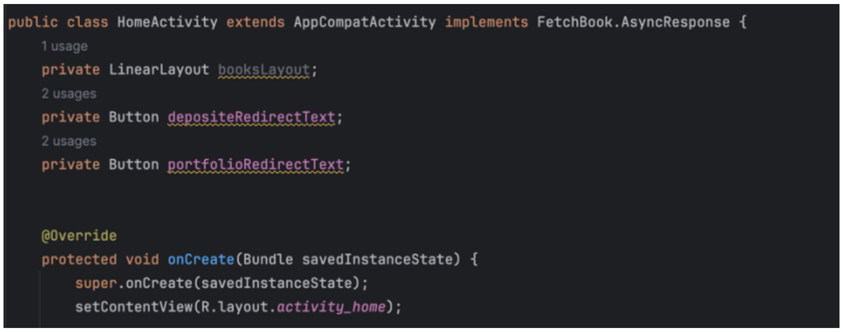
```java
loginRedirectText = findViewById(R.id.gotoSignInTextView);
```



```java
loginRedirectText.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        startActivity(new Intent( packageContext: SignUpFragment.this, SignInFragment.class));
    }
});
```

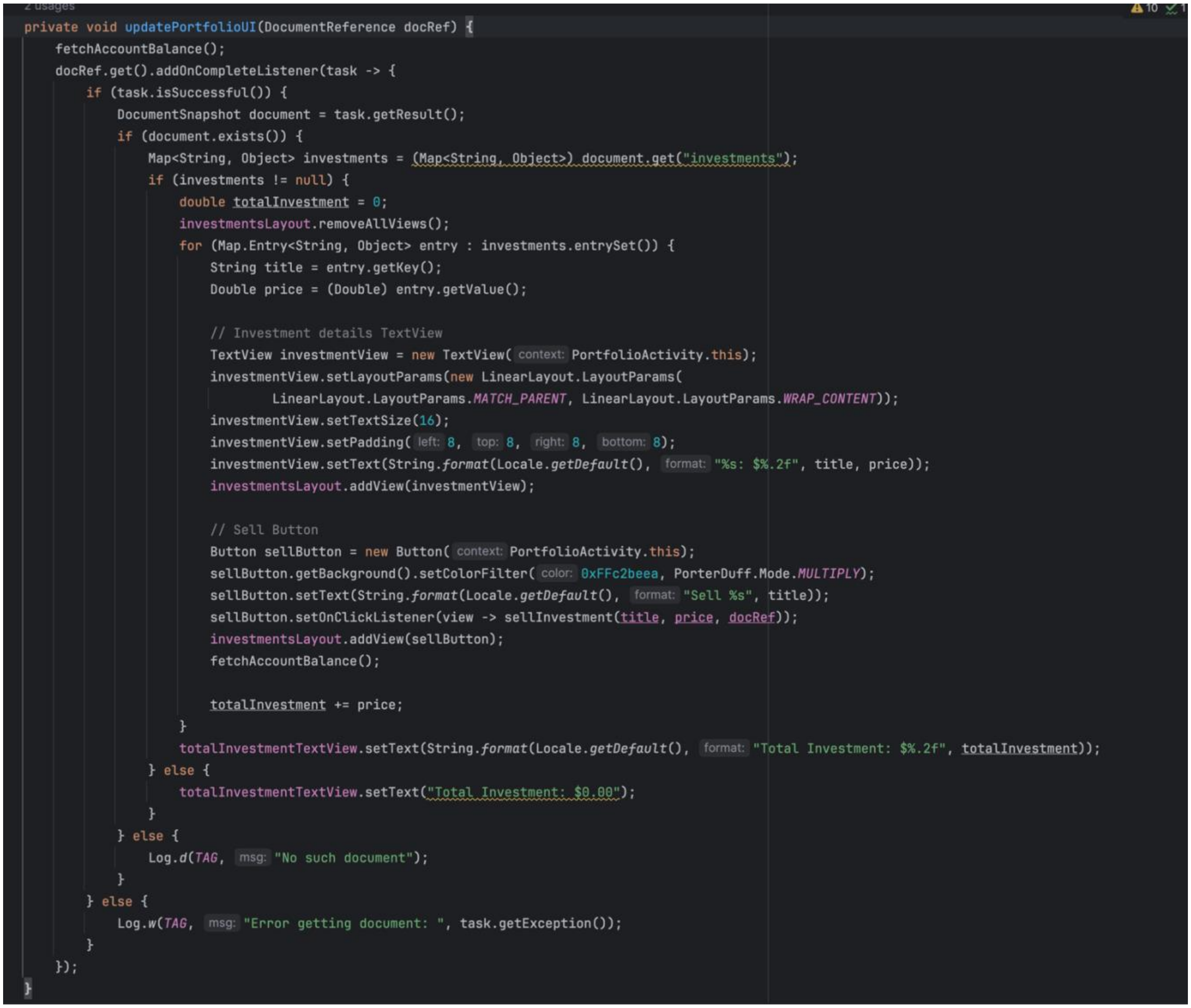
XML layouts were utilized, integrated with Java through the Android widget toolkit. Event listeners were set up, triggering actions such as transitioning to different screens in response to user interactions like button clicks.

Inheritance: Every activity class extends ‘AppCompatActivity’. It inherits methods like onCreate(), which is used to display the UI.



```
public class HomeActivity extends AppCompatActivity implements FetchBook.AsyncResponse {
    1 usage
    private LinearLayout booksLayout;
    2 usages
    private Button depositeRedirectText;
    2 usages
    private Button portfolioRedirectText;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_home);
```

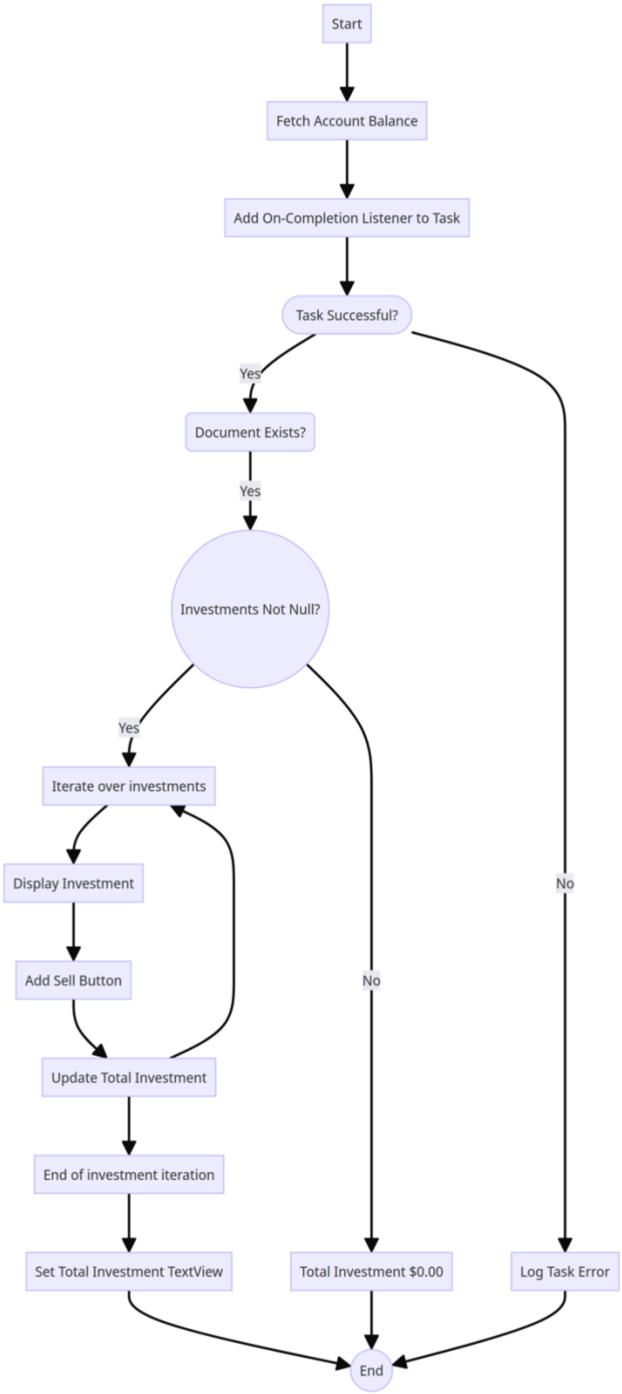
Algorithm for Updating Portfolio



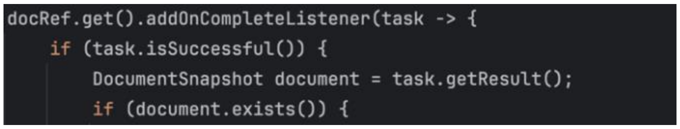
```
  2 usages
private void updatePortfolioUI(DocumentReference docRef) {
    fetchAccountBalance();
    docRef.get().addOnCompleteListener(task -> {
        if (task.isSuccessful()) {
            DocumentSnapshot document = task.getResult();
            if (document.exists()) {
                Map<String, Object> investments = (Map<String, Object>) document.get("investments");
                if (investments != null) {
                    double totalInvestment = 0;
                    investmentsLayout.removeAllViews();
                    for (Map.Entry<String, Object> entry : investments.entrySet()) {
                        String title = entry.getKey();
                        Double price = (Double) entry.getValue();

                        // Investment details TextView
                        TextView investmentView = new TextView( context: PortfolioActivity.this);
                        investmentView.setLayoutParams(new LinearLayout.LayoutParams(
                                LinearLayout.LayoutParams.MATCH_PARENT, LinearLayout.LayoutParams.WRAP_CONTENT));
                        investmentView.setTextSize(16);
                        investmentView.setPadding( left: 8,  top: 8,  right: 8,  bottom: 8);
                        investmentView.setText(String.format(Locale.getDefault(),  format: "%s: $%.2f", title, price));
                        investmentsLayout.addView(investmentView);

                        // Sell Button
                        Button sellButton = new Button( context: PortfolioActivity.this);
                        sellButton.getBackground().setColorFilter( color: 0xFFc2beea, PorterDuff.Mode.MULTIPLY);
                        sellButton.setText(String.format(Locale.getDefault(),  format: "Sell %s", title));
                        sellButton.setOnClickListener(view -> sellInvestment(title, price, docRef));
                        investmentsLayout.addView(sellButton);
                        fetchAccountBalance();

                        totalInvestment += price;
                    }
                    totalInvestmentTextView.setText(String.format(Locale.getDefault(),  format: "Total Investment: $%.2f", totalInvestment));
                } else {
                    totalInvestmentTextView.setText("Total Investment: $0.00");
                }
            } else {
                Log.d(TAG,  msg: "No such document");
            }
        } else {
            Log.w(TAG,  msg: "Error getting document: ", task.getException());
        }
    });
}
```

This is used in the PortfolioActivity class. It iteratively constructs a user interface for each investment, displaying the book title and investment value, and providing a sell button with an attached listener for further transactional operations. This approach was chosen to ensure that the UI reflects the current state of the user's investments in real-time, directly from the Firestore database. It employs a transactional model to handle user actions, which safeguards the integrity of the user's data, allowing for the sell operations to be **atomic** and consistent, thereby preventing data corruption or loss during the investment sell-off process.

Logic for Updating Portfolio Algorithm



Use of documentSnapshot

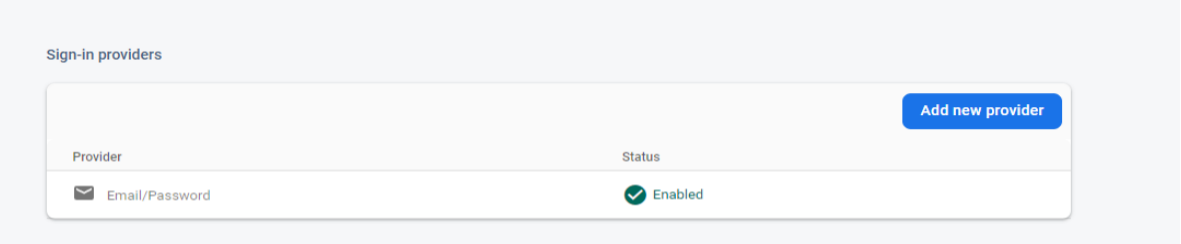


```
docRef.get().addOnCompleteListener(task -> {
    if (task.isSuccessful()) {
        DocumentSnapshot document = task.getResult();
        if (document.exists()) {
```

The **documentSnapshot** from Firestore captures the state of a document at a specific moment in time, providing a reliable view of its data for read operations without further database calls. Here, it ensures financial transactions, are based on the latest, consistent data.

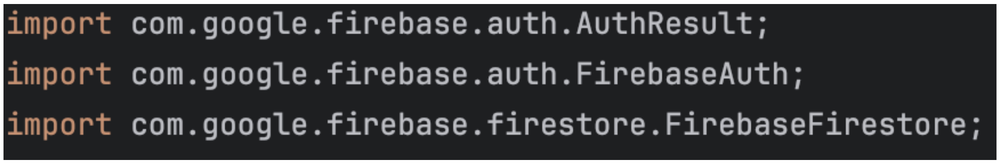
2. Firebase Firestore Integration

Compared to alternatives like MongoDB, Firebase is more straightforward usage especially when it comes to Java Android development. This simplicity is crucial for fulfilling Success Criteria 1



Sign-in providers

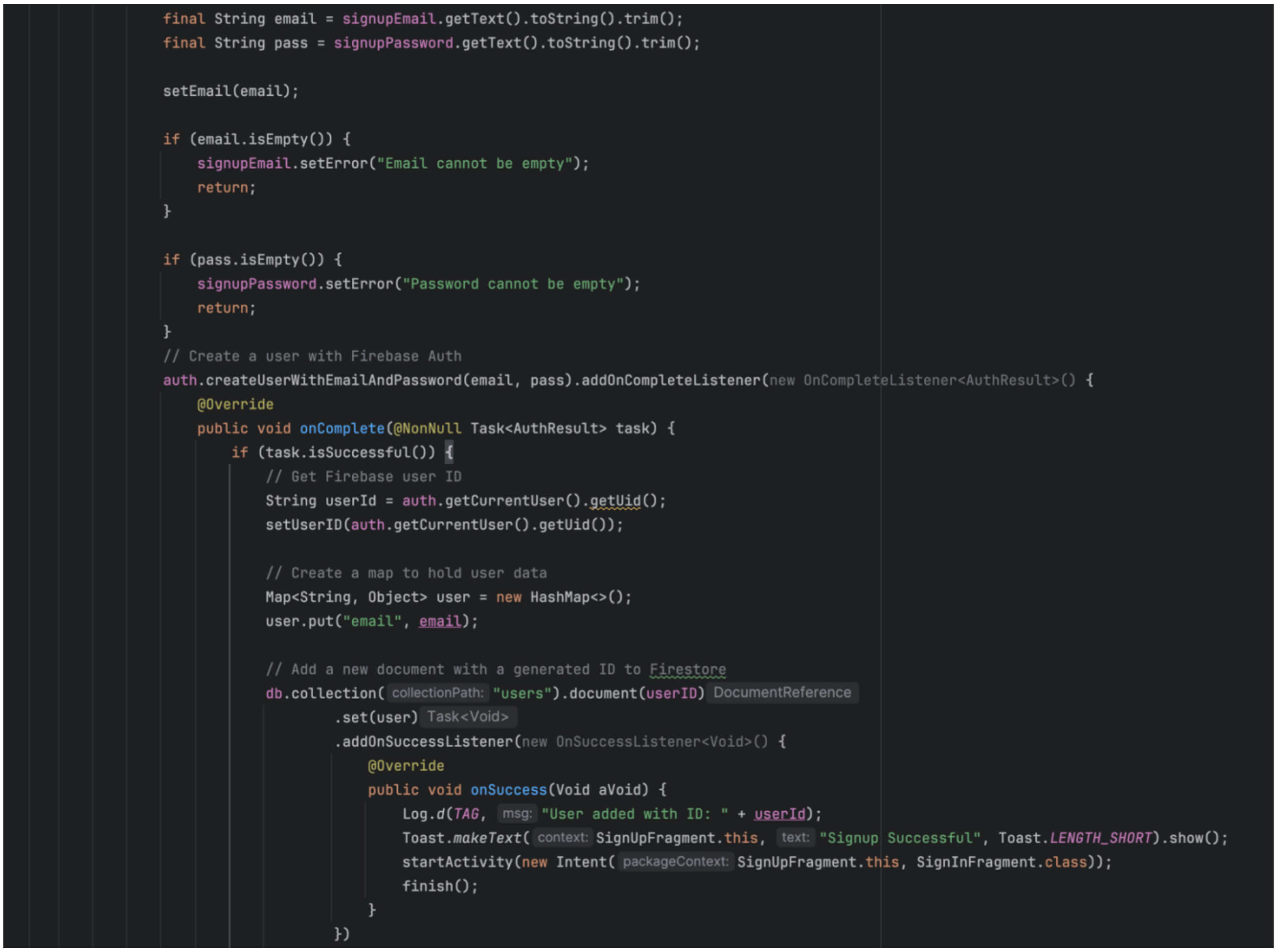
Provider	Status
✉ Email/Password	✓ Enabled



```java
import com.google.firebase.auth.AuthResult;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.firestore.FirebaseFirestore;
```

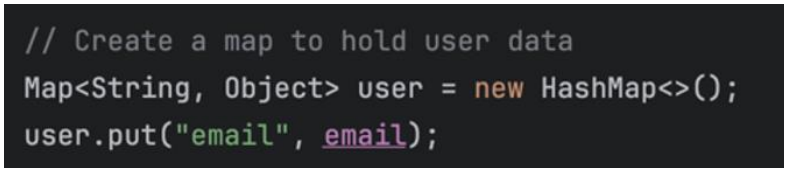
Below is the Sign Up and Sign In code:

Sign Up:



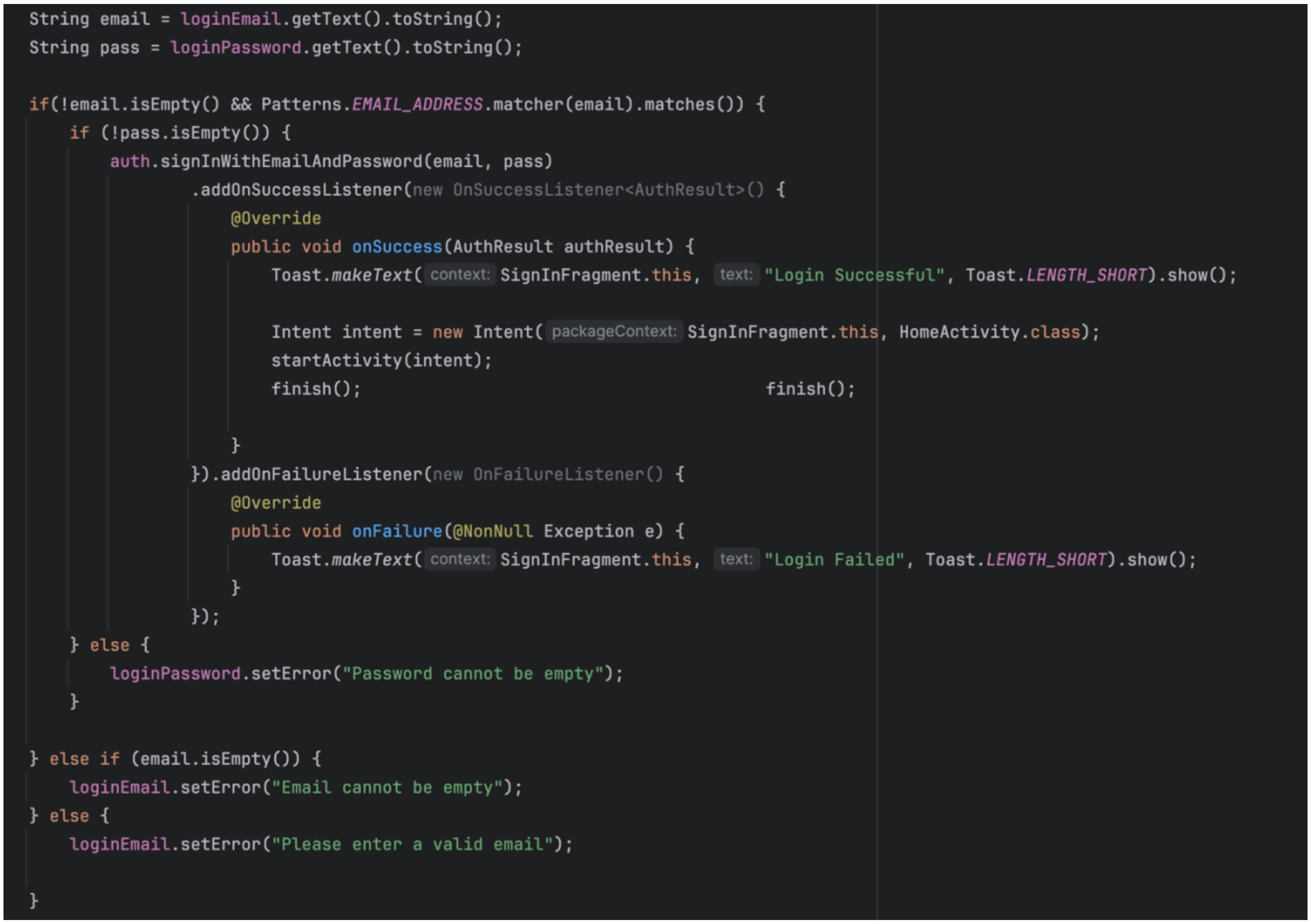
```java
final String email = signupEmail.getText().toString().trim();
final String pass = signupPassword.getText().toString().trim();

setEmail(email);

if (email.isEmpty()) {
    signupEmail.setError("Email cannot be empty");
    return;
}

if (pass.isEmpty()) {
    signupPassword.setError("Password cannot be empty");
    return;
}
// Create a user with Firebase Auth
auth.createUserWithEmailAndPassword(email, pass).addOnCompleteListener(new OnCompleteListener<AuthResult>() {
    @Override
    public void onComplete(@NonNull Task<AuthResult> task) {
        if (task.isSuccessful()) {
            // Get Firebase user ID
            String userId = auth.getCurrentUser().getUid();
            setUserID(auth.getCurrentUser().getUid());

            // Create a map to hold user data
            Map<String, Object> user = new HashMap<>();
            user.put("email", email);

            // Add a new document with a generated ID to Firestore
            db.collection( collectionPath: "users").document(userID) DocumentReference
                    .set(user) Task<Void>
                    .addOnSuccessListener(new OnSuccessListener<Void>() {
                        @Override
                        public void onSuccess(Void aVoid) {
                            Log.d(TAG,  msg: "User added with ID: " + userId);
                            Toast.makeText( context: SignUpFragment.this,  text: "Signup Successful", Toast.LENGTH_SHORT).show();
                            startActivity(new Intent( packageContext: SignUpFragment.this, SignInFragment.class));
                            finish();
                        }
                    })
```

Firebase is used for email authentication as seen above. The user's email is stored in a HashMap, as a value pair. User data is fetched by using the user's Firebase ID, which is given on registration.



```java
// Create a map to hold user data
Map<String, Object> user = new HashMap<>();
user.put("email", email);
```

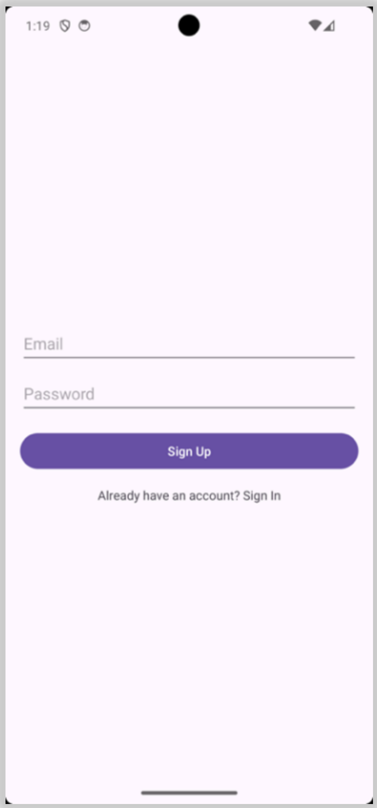
Log In:

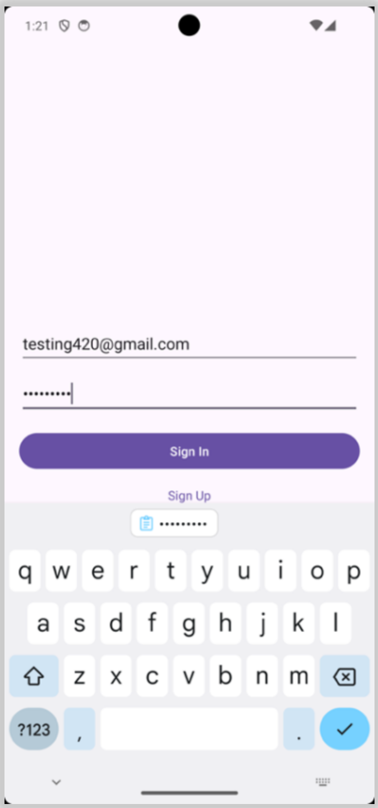


```java
String email = loginEmail.getText().toString();
String pass = loginPassword.getText().toString();

if(!email.isEmpty() && Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
    if (!pass.isEmpty()) {
        auth.signInWithEmailAndPassword(email, pass)
                .addOnSuccessListener(new OnSuccessListener<AuthResult>() {
                    @Override
                    public void onSuccess(AuthResult authResult) {
                        Toast.makeText( context: SignInFragment.this,  text: "Login Successful", Toast.LENGTH_SHORT).show();

                        Intent intent = new Intent( packageContext: SignInFragment.this, HomeActivity.class);
                        startActivity(intent);
                        finish();                                     finish();

                    }
                }).addOnFailureListener(new OnFailureListener() {
                    @Override
                    public void onFailure(@NonNull Exception e) {
                        Toast.makeText( context: SignInFragment.this,  text: "Login Failed", Toast.LENGTH_SHORT).show();
                    }
                });
    } else {
        loginPassword.setError("Password cannot be empty");
    }

} else if (email.isEmpty()) {
    loginEmail.setError("Email cannot be empty");
} else {
    loginEmail.setError("Please enter a valid email");

}
```

Note the usage of onSuccess and OnFailure listeners, which each, respectively handle success's and failures when the Sign In button is pressed. This ease of error handling was another reason for my use of Firebase.

UI Screenshots:

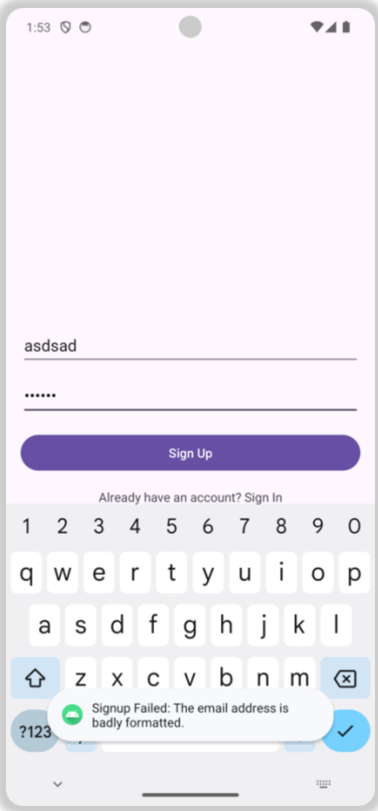



testing420@gmail.com

•••••••••

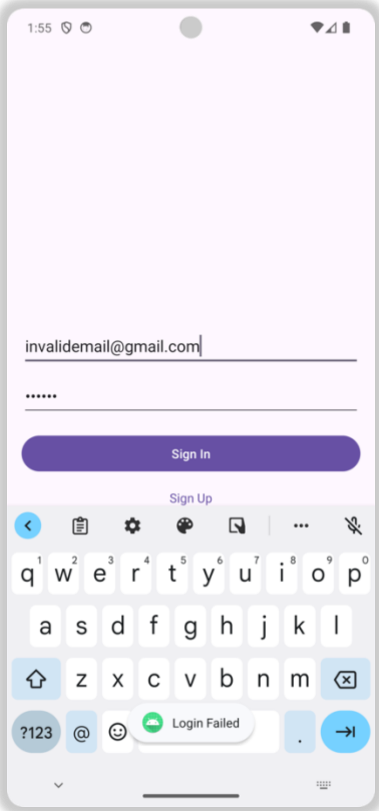
Error

Handling:


asdsad

••••••

Signup Failed: The email address is badly formatted.

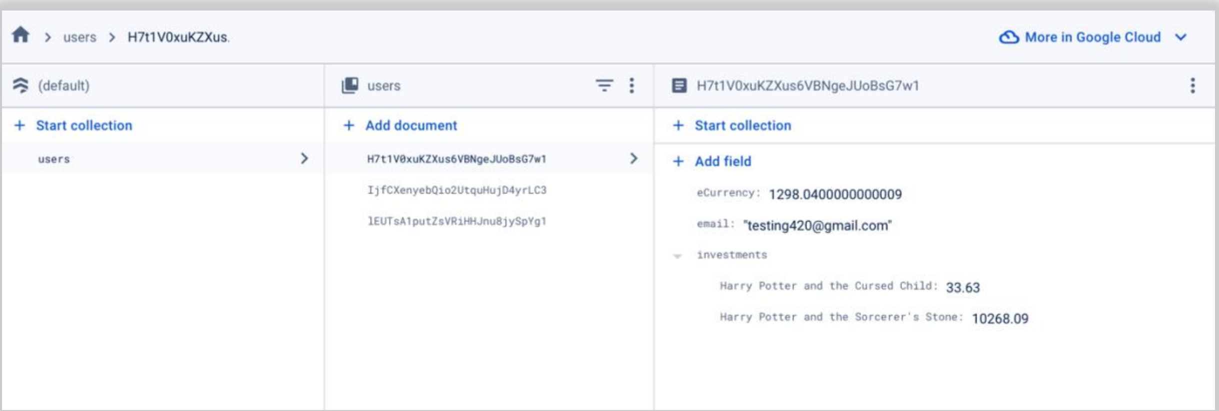

invalidemail@gmail.com

••••••

Login Failed

User-specific data storage was achieved by utilizing subcollections within a user's main collection with a userID based document in Firebase.

Data Structure:



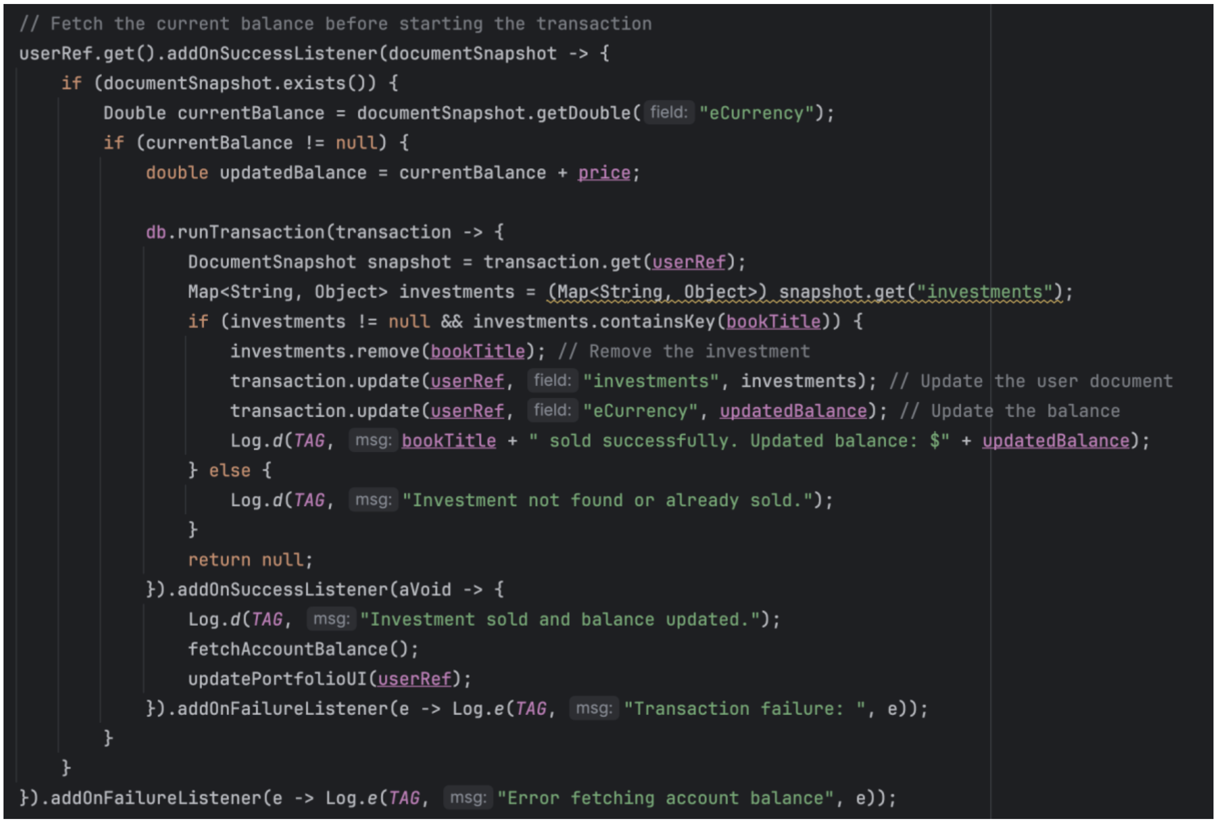
users

H7t1V0xuKZXus6VBNgeJUoBsG7w1
IjfCXenyebQio2UtquHujD4yrLC3
lEUTsA1putZsVRiHHJnu8jySpYg1

```
eCurrency: 1298.0400000000009
email: "testing420@gmail.com"
investments
    Harry Potter and the Cursed Child: 33.63
    Harry Potter and the Sorcerer's Stone: 10268.09
```

The next step in utilizing Firebase was to utilize data snapshots, and a Future<void> method for writing data (Appendix A, Build a Flutter Blog App from Scratch).

Transactions using Firebase:

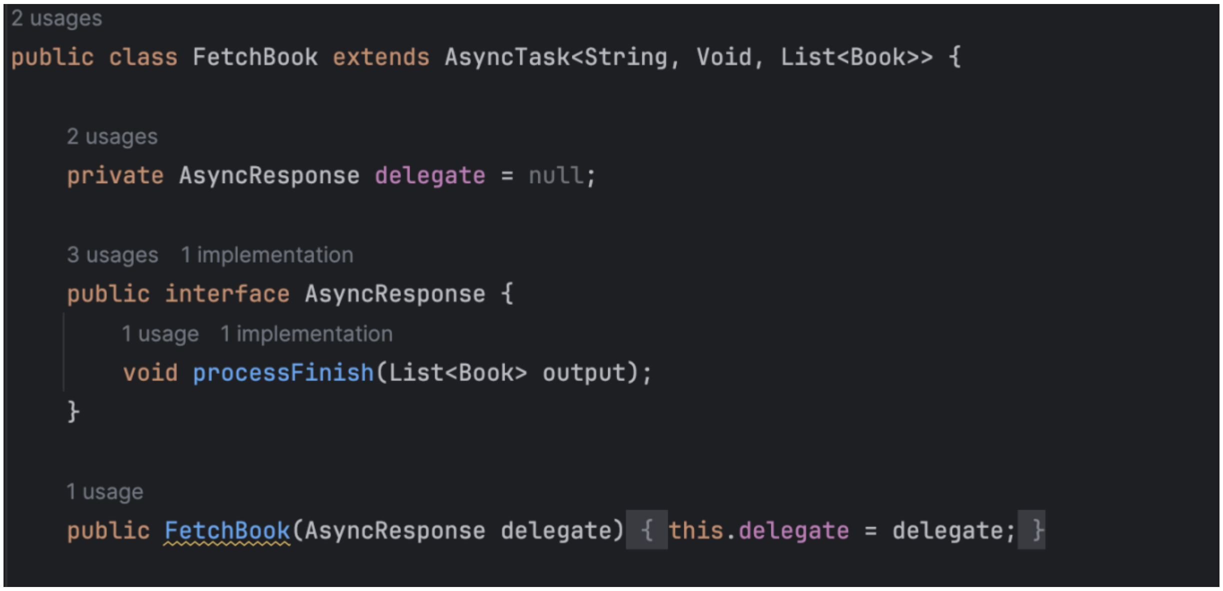


```
// Fetch the current balance before starting the transaction
userRef.get().addOnSuccessListener(documentSnapshot -> {
    if (documentSnapshot.exists()) {
        Double currentBalance = documentSnapshot.getDouble( field: "eCurrency");
        if (currentBalance != null) {
            double updatedBalance = currentBalance + price;

            db.runTransaction(transaction -> {
                DocumentSnapshot snapshot = transaction.get(userRef);
                Map<String, Object> investments = (Map<String, Object>) snapshot.get("investments");
                if (investments != null && investments.containsKey(bookTitle)) {
                    investments.remove(bookTitle); // Remove the investment
                    transaction.update(userRef,  field: "investments", investments); // Update the user document
                    transaction.update(userRef,  field: "eCurrency", updatedBalance); // Update the balance
                    Log.d(TAG,  msg: bookTitle + " sold successfully. Updated balance: $" + updatedBalance);
                } else {
                    Log.d(TAG,  msg: "Investment not found or already sold.");
                }
                return null;
            }).addOnSuccessListener(aVoid -> {
                Log.d(TAG,  msg: "Investment sold and balance updated.");
                fetchAccountBalance();
                updatePortfolioUI(userRef);
            }).addOnFailureListener(e -> Log.e(TAG,  msg: "Transaction failure: ", e));
        }
    }
}).addOnFailureListener(e -> Log.e(TAG,  msg: "Error fetching account balance", e));
```

Utilizing Firestore transactions is essential for maintaining data consistency within multiple classes, particularly for operations that must be atomic to prevent the app's state from becoming invalid. Transactions in Firestore ensure that all operations within the transaction block are executed and committed to the cloud database atomically. This ensures that changes to the user's balance during investment operations either fully complete or not at all, safeguarding against partial updates that can cause inaccurate financial data representation. This checks off the complexity requirements for sophisticated data management techniques.

3. Data Model and Google Books API Integration

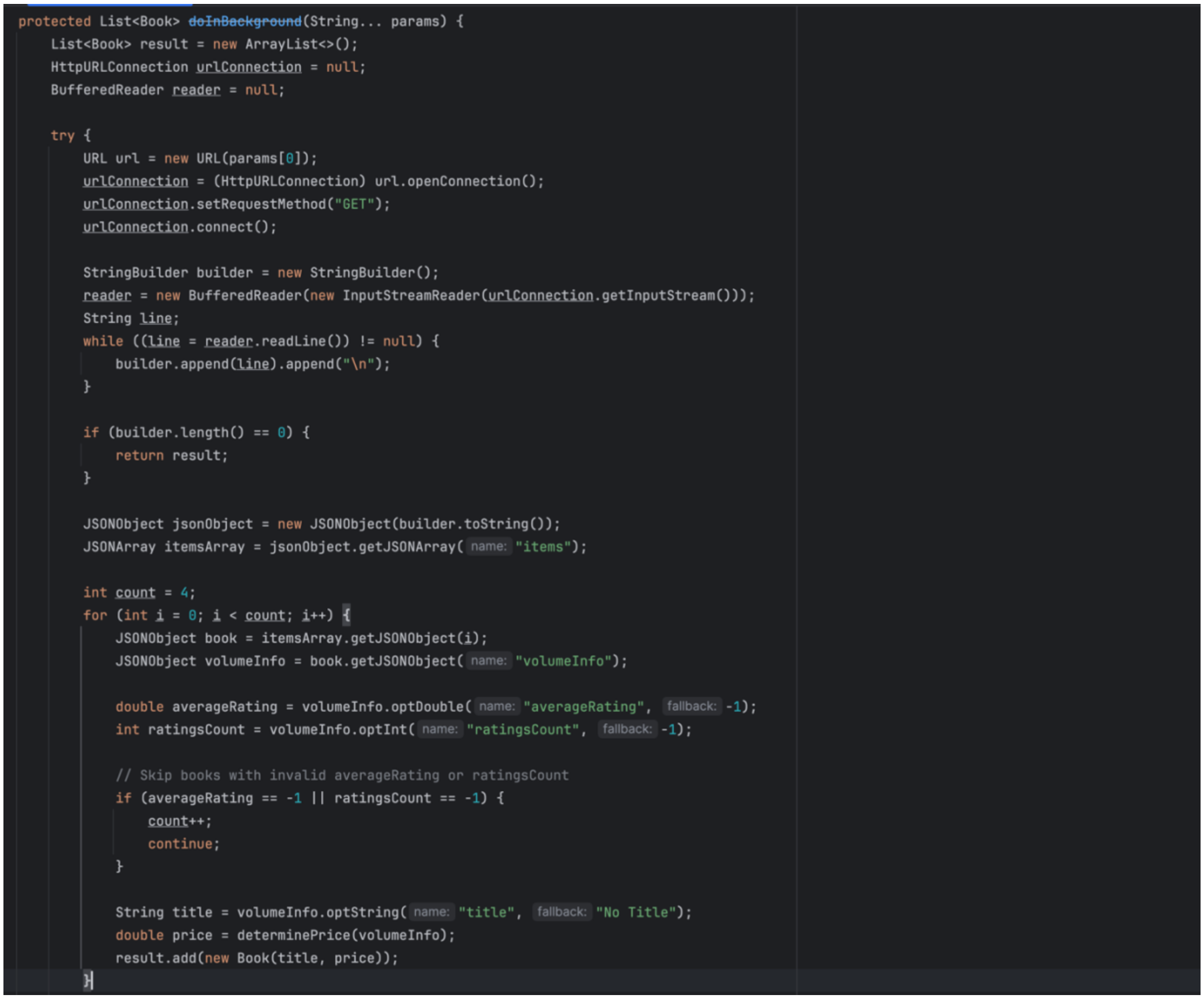
FetchBook Class:



```
2 usages
public class FetchBook extends AsyncTask<String, Void, List<Book>> {

    2 usages
    private AsyncResponse delegate = null;

    3 usages   1 implementation
    public interface AsyncResponse {
        1 usage   1 implementation
        void processFinish(List<Book> output);
    }

    1 usage
    public FetchBook(AsyncResponse delegate) { this.delegate = delegate; }
```

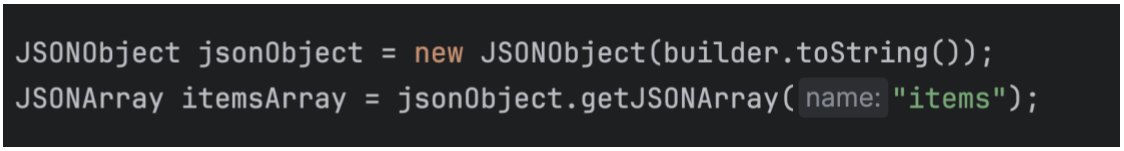
The purpose of FetchBook is to efficiently manage the retrieval and processing of data from the Google Books API. The doInBackground method operates asynchronously in the background, ensuring the application remains responsive. The AsyncResponse interface is a custom callback mechanism that enables communication back to the main thread upon task completion. FetchBook extends AsyncTask, utilizing **polymorphism** to execute operations that are inherently asynchronous and handle different types of execution tasks without blocking the user interface.

doInBackgroundIO method:



```java
protected List<Book> doInBackground(String... params) {
    List<Book> result = new ArrayList<>();
    HttpURLConnection urlConnection = null;
    BufferedReader reader = null;

    try {
        URL url = new URL(params[0]);
        urlConnection = (HttpURLConnection) url.openConnection();
        urlConnection.setRequestMethod("GET");
        urlConnection.connect();

        StringBuilder builder = new StringBuilder();
        reader = new BufferedReader(new InputStreamReader(urlConnection.getInputStream()));
        String line;
        while ((line = reader.readLine()) != null) {
            builder.append(line).append("\n");
        }

        if (builder.length() == 0) {
            return result;
        }

        JSONObject jsonObject = new JSONObject(builder.toString());
        JSONArray itemsArray = jsonObject.getJSONArray( name: "items");

        int count = 4;
        for (int i = 0; i < count; i++) {
            JSONObject book = itemsArray.getJSONObject(i);
            JSONObject volumeInfo = book.getJSONObject( name: "volumeInfo");

            double averageRating = volumeInfo.optDouble( name: "averageRating",  fallback: -1);
            int ratingsCount = volumeInfo.optInt( name: "ratingsCount",  fallback: -1);

            // Skip books with invalid averageRating or ratingsCount
            if (averageRating == -1 || ratingsCount == -1) {
                count++;
                continue;
            }

            String title = volumeInfo.optString( name: "title",  fallback: "No Title");
            double price = determinePrice(volumeInfo);
            result.add(new Book(title, price));
        }
```

The method performs a background API call to Google Books and processes the JSON response into a structured **list** of **Book** objects. This specific approach is justified as it allows the app to continue functioning normally while the data is being fetched and processed, which is critical for a smooth user experience.

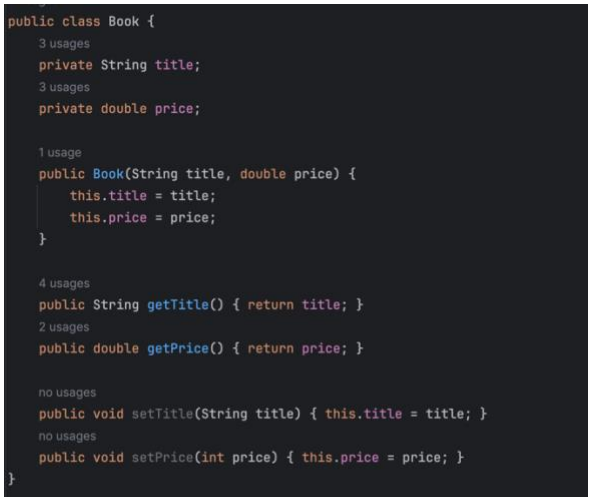


```java
JSONObject jsonObject = new JSONObject(builder.toString());
JSONArray itemsArray = jsonObject.getJSONArray( name: "items");
```

A **Composite Data Structure** is used to manage and access the complex, **multidimensional JSON array** effectively. After instantiation, the **text is parsed** through this hierarchical JSON object to meticulously search for specific book details, such as average ratings and ratings count.

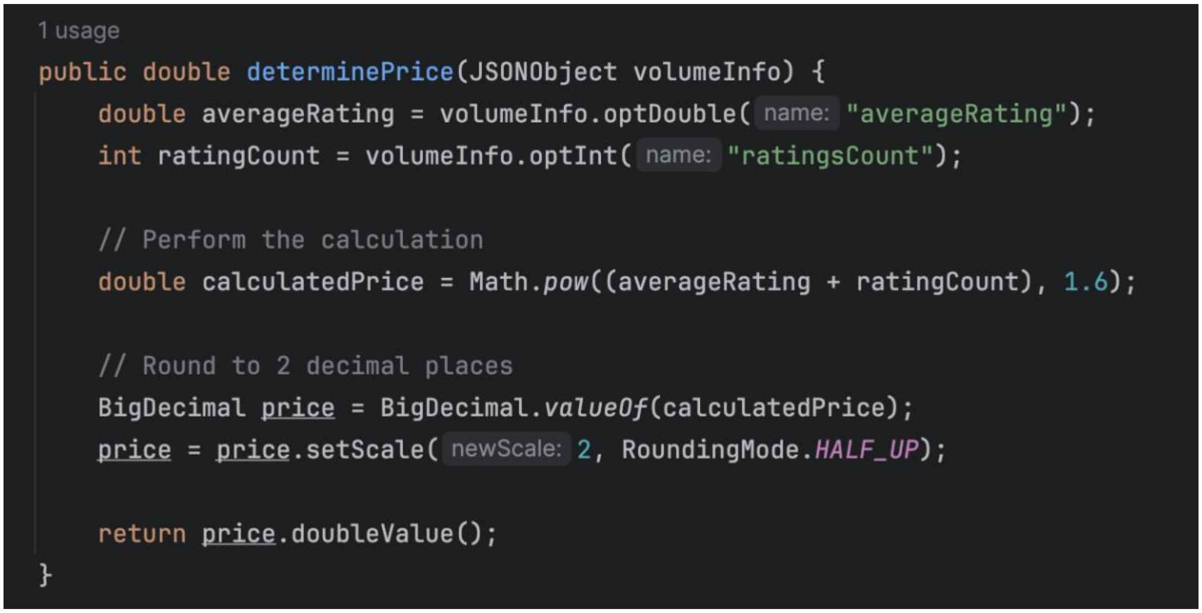
Book Class:

This class uses encapsulation with getters and setters. It used in the above method.



```
public class Book {
    3 usages
    private String title;
    3 usages
    private double price;

    1 usage
    public Book(String title, double price) {
        this.title = title;
        this.price = price;
    }

    4 usages
    public String getTitle() { return title; }
    2 usages
    public double getPrice() { return price; }

    no usages
    public void setTitle(String title) { this.title = title; }
    no usages
    public void setPrice(int price) { this.price = price; }
}
```

Algorithm for Valuating Books



```
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();
}
```

This deterimines book prices by combining average ratings and number of ratings, adjusting accordingly to reflect value from consumer feedback. After the calculation, the price is rounded to two decimal places for standardization and ease of display. This method is necessary part of the application.

Word Count: 876

CRITERION E: EVALUATION

EVALUATION:

Success Criteria	Progress
High Priority: Users must be able to create an account with a valid email and password with at least 8 characters.	I have successfully met this criterion. My client expressed satisfaction, particularly praising the ease of the process and the intuitive user interface for account setup. Security measures and database integration through Firebase was seamlessly integrated.
High Priority: Users must be able to browse through different books on the marketpage.	This criterion was achieved as demonstrated in the app's browsing functionality. The feedback from Timothy confirmed that the browsing experience was seamless and met his expectations.
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.	I believe this criterion has been effectively met. The investment functionality includes rigorous validation checks, as outlined in the project's demonstration. Timothy was particularly pleased with the clear feedback.
High Priority: The app must update investment values with data fetched from verified sources and APIs every instance.	For criterion 4, I believe I was successful. Although my client did not comment on this, it is clear from Criterion D’s implementation of the application that book prices were regularly updated through APIs.
Medium Priority: The app should allow users to add any amount of currency to their account.	This criterion has been met. As demonstrated in Criterion D, the app allows users to deposit any given amount of money.
Low Priority: When user closes application, the system should log the user off, and prompt them to relog in for the next instance.	This criterion has been met. When the emulator, and hence the app is restarted, the user is prompted with the sign-up screen. User can register a new account, or click the sign in button to log in. Regardless, user is logged out after closing the instance.

FUTURE EMPLOYMENTS:

In future renditions of this application, I would like to include:

1. Shares: I would like to implement a feature to buy multiple shares of a book. Thanks to extensibility of the HomeActivity class, I can easily add new buttons to buy shares and simply modify the investments map Java, and its corresponding collection in Firebase to include a “shares” value pair.

2. Search Bar: Implementing a search bar to search through books to invest in, could be done by leveraging Android’s RecyclerView and integrating it into the HomeActivity class, allowing for dynamic filtering of book titles as user input changes.

WordCount: 416

APPENDIX

APPENDIX A CITATIONS:

Works Cited

"Firebase Authentication." *Firebase Documentation*, Google, firebase.google.com/docs/auth. Accessed 2 Mar. 2024.

"Firebase Realtime Database." *Firebase Documentation*, Google, firebase.google.com/docs/database. Accessed 2 Mar. 2024.

"Google Books APIs." *Google Developers*, Google, developers.google.com/books. Accessed 3 Mar. 2024.

Griffith, Erin. "Start-Ups Aren't Cool Anymore." *The New York Times*, 27 Dec. 2018, www.nytimes.com/2018/12/27/technology/start-ups-cooling.html. Accessed 3 Mar. 2024.

Leach, Jim. "Developing Secure Mobile Applications for Android." *OWASP*, 2015, www.owasp.org/index.php/Developing_Secure_Mobile_Applications_for_Android. Accessed 3 Mar. 2024.

Morrison, Michael. *Head First Android Development: A Brain-Friendly Guide*. 2nd ed., O'Reilly Media, 2017.

Neilsen, Jakob. "Usability 101: Introduction to Usability." *Nielsen Norman Group*, 3 Jan. 2012, www.nngroup.com/articles/usability-101-introduction-to-usability/. Accessed 3 Mar. 2024.

Phillips, Bill, et al. *Android Programming: The Big Nerd Ranch Guide*. 4th ed., Big Nerd Ranch Guides, 2019.

"Using the Android Emulator." *Android Developers*, Google, developer.android.com/studio/run/emulator. Accessed 14 Mar. 2024.

APPENDIX B EVIDENCE OF CONSULTATION:

Initial Consultation with client 1/29/24:

Rohan (R): Hey, Timothy. Ready to discuss the my computer science IA?

Timothy (T): Yes, excited to share my thoughts. I envision a platform where I can invest in the success of books.

R: Interesting concept. Features like book browsing, investing, and portfolio tracking come to my mind.

T: Precisely. I would like a Android application.

R: Understood. For the backend, would Firebase sound good? I think this is a good combination with an Android application.

T: Yes, users should be able to sell of investments when needed.

R: Got it. I'll outline a project plan focusing on these key features. We'll refine the app based on user feedback and performance data.

T: Looking forward to it, Rohan.

Part 2:

Discussing the App with the Advisor 2/1/24:

R: Rohan (Me)

S: Sandesh

(Advisor)

R: Hey Sandesh, for my Computer Science IA I'm creating a book investment app for Timothy. Any ideas on how to implement it? It will be an Android application.

R: Sounds good. What about the IDE?

S: I would suggest using Android Studio with XMLs.

R: Alright. Sounds good.

Part 3:

Second Consultation with client 2/6/24:

R: Rohan (Me)

T: Timothy

(Client)

R: Do you know what kind of success

criteria you want?

T: Not really.

R: What about a working authentication

system? That seems vital for any app.

T: Oh yeah. That sounds good.

Part 4:

Check In with Client 3/2/24:

R: Rohan (Me)

T: Timothy

(Client)

R: Hey, Timothy, sorry for the delay, but I will show you what I have.

T: No worries! Go ahead.

T: Looks good! I like the UI layout.

R: Ok. Glad you like the UI. I'll continue to update you.

Part 7: Showing Final Product to Client (Virtual) 3/16/22

R: Rohan (Me)

T: Timothy

(Client)

R: Hey Timothy, the app is done.

T: I’m excited man!

Shows App

T: Wow Rohan that’s great!

R: Thank you!

T: This is amazing. I’m excited for the future of this app.

R: Thank you, thank you. It was great working with you.