CRITERION C: DEVELOPMENT

TECHNIQUES:

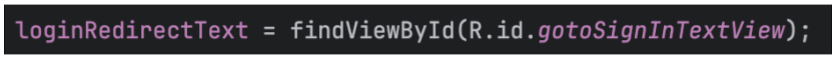
Some key techniques used in my project are:

- 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

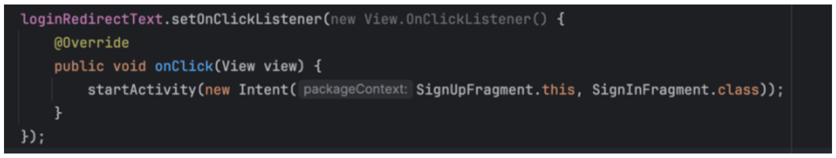
1. Dynamic UI Updates and Navigation

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:



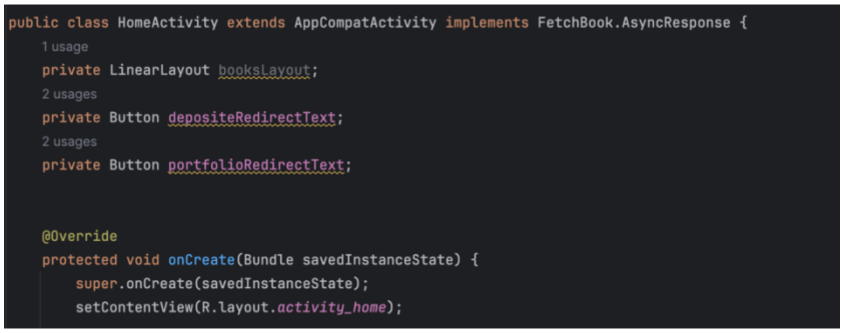
```java
loginRedirectText = findViewById(R.id.gotoSignInTextView);
```



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

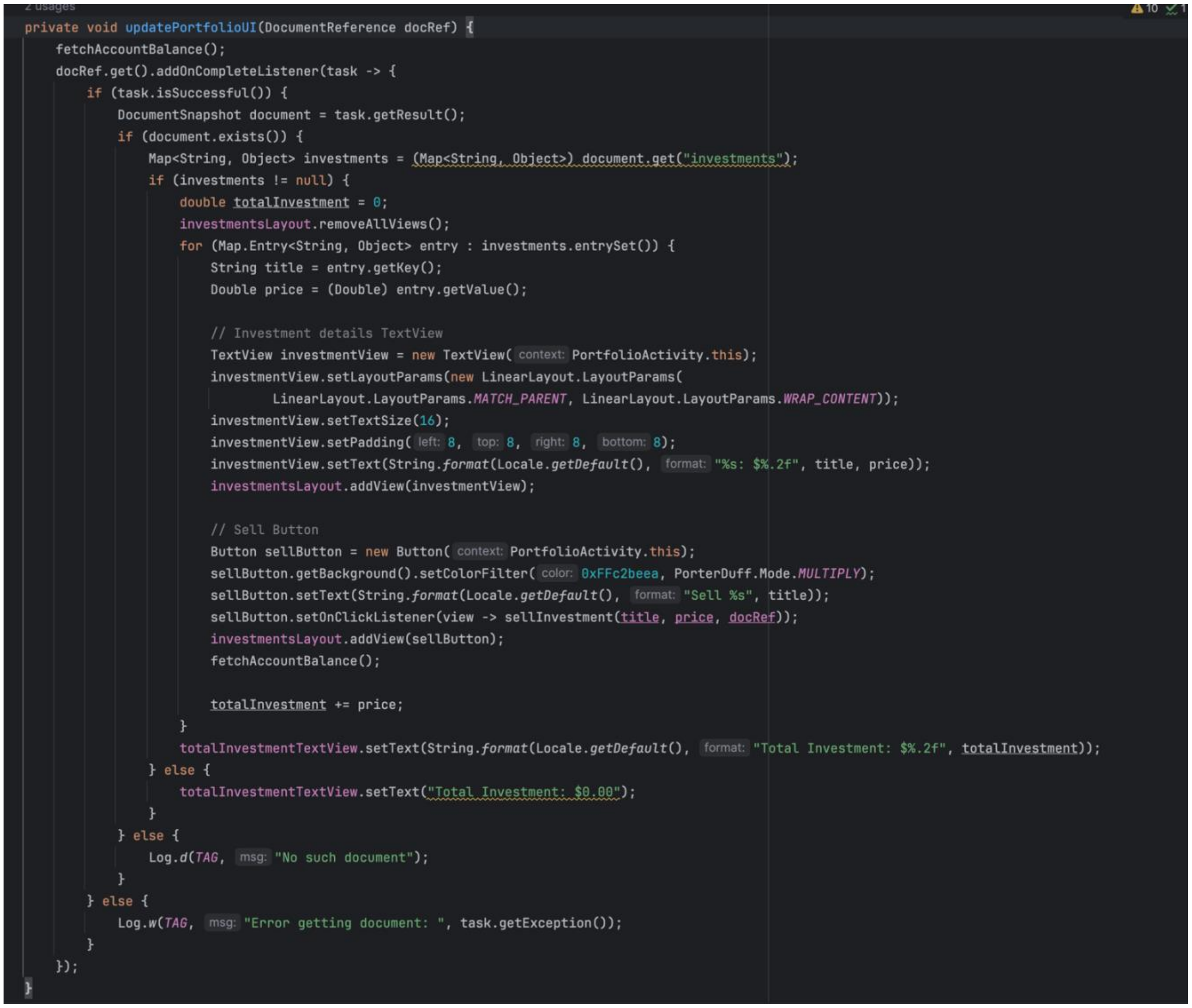
XML layouts were utilized, integrated with Java through the Android widget toolkit. Event listeners were set up, triggering actions such as transitioning to different screens in response to user interactions like button clicks.

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



```java
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);
```

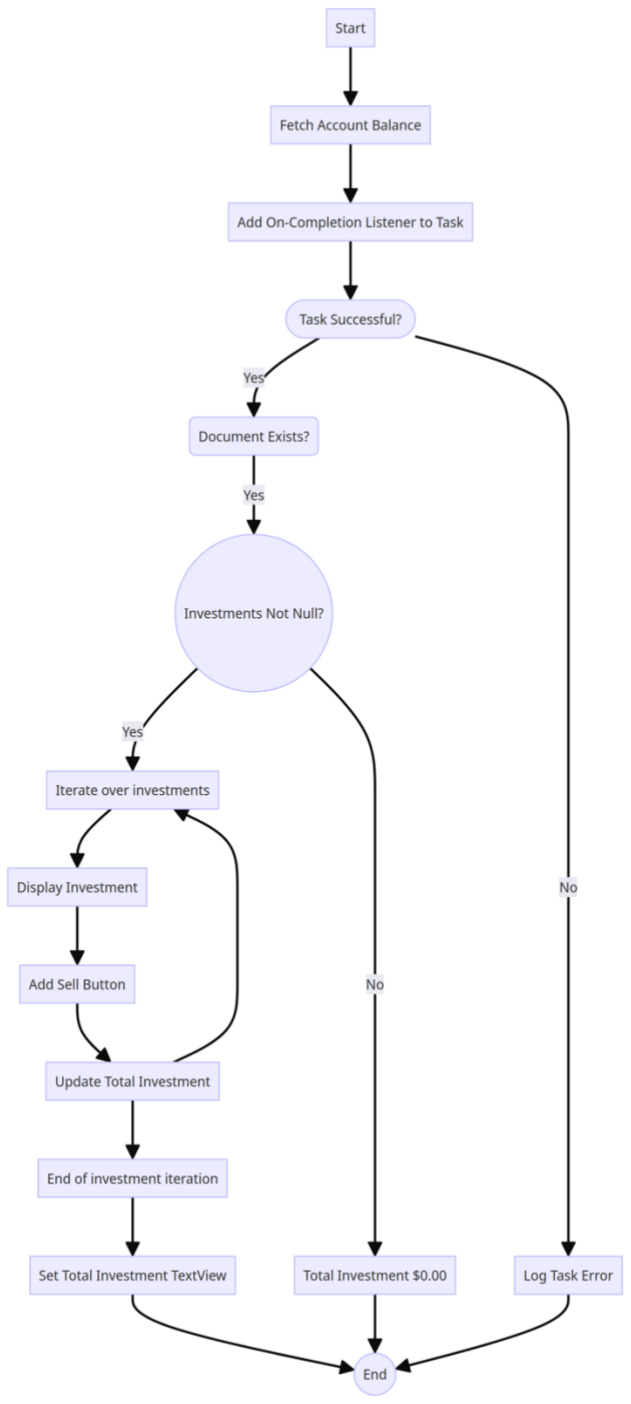
Algorithm for Updating Portfolio



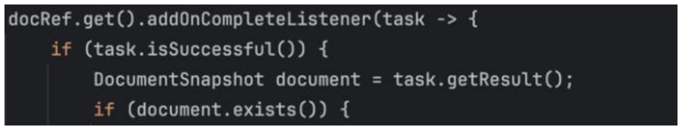
```java
  2 usages
  private void updatePortfolioUI(DocumentReference docRef) {
      fetchAccountBalance();
      docRef.get().addOnCompleteListener(task -> {
          if (task.isSuccessful()) {
              DocumentSnapshot document = task.getResult();
              if (document.exists()) {
                  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

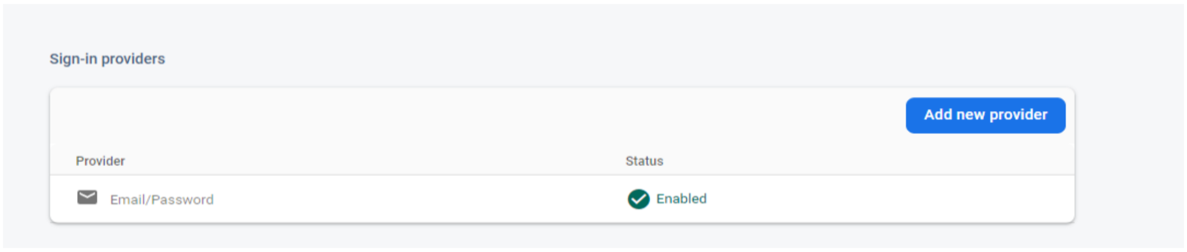


```
docRef.get().addOnCompleteListener(task -> {
    if (task.isSuccessful()) {
        DocumentSnapshot document = task.getResult();
        if (document.exists()) {
```

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.

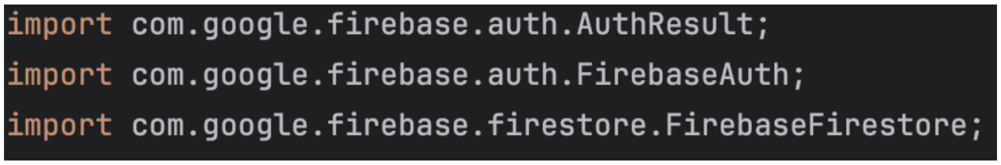
2. Firebase Firestore Integration

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

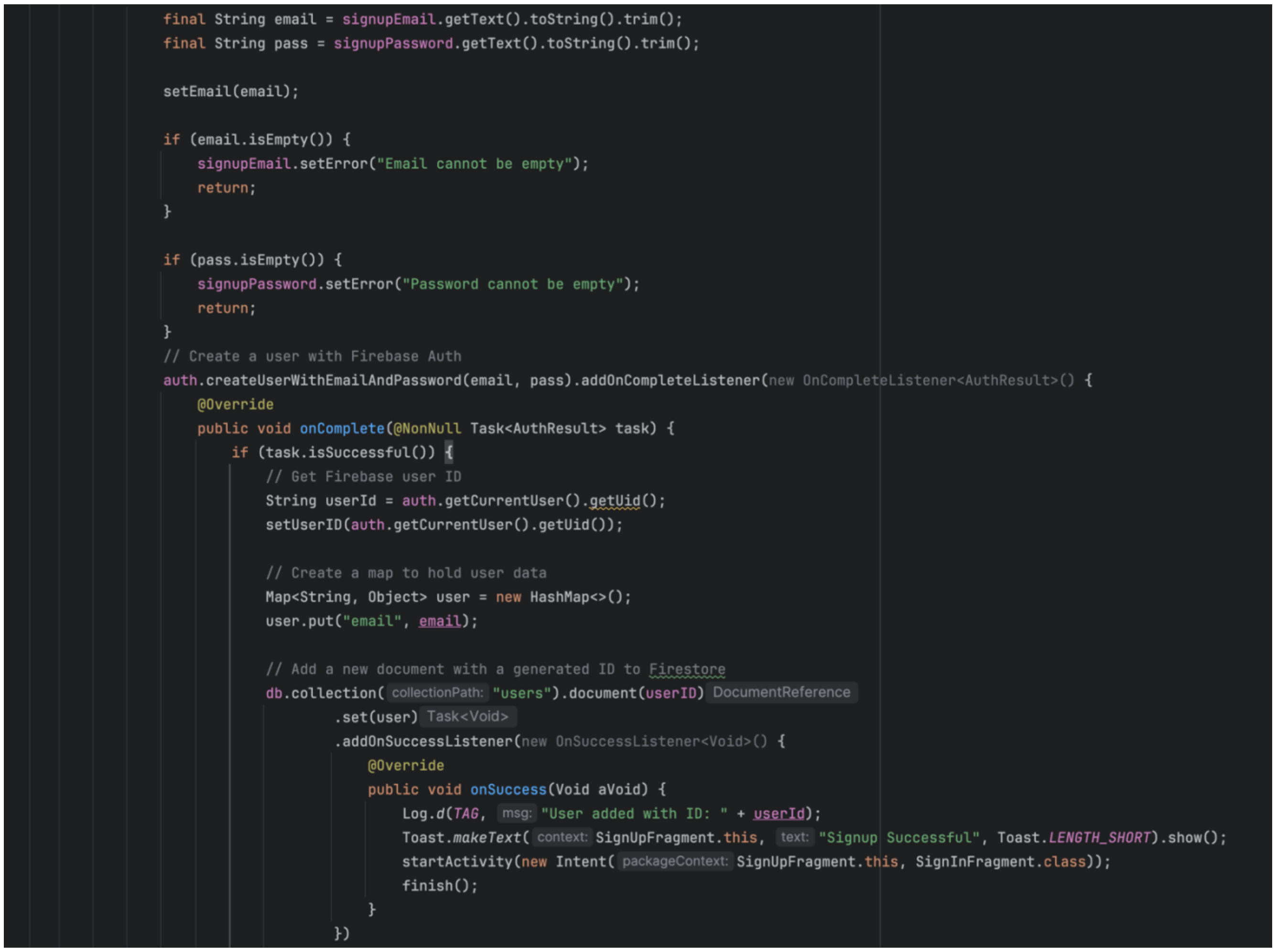
Provider	Status
Email/Password	✓ Enabled



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

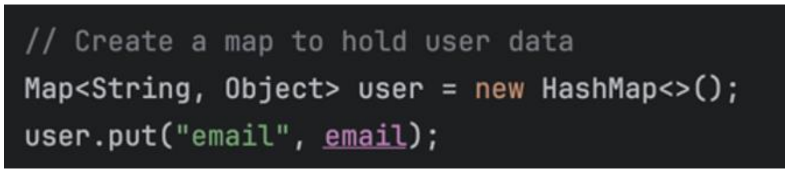
Below is the Sign Up and Sign In code:

Sign Up:



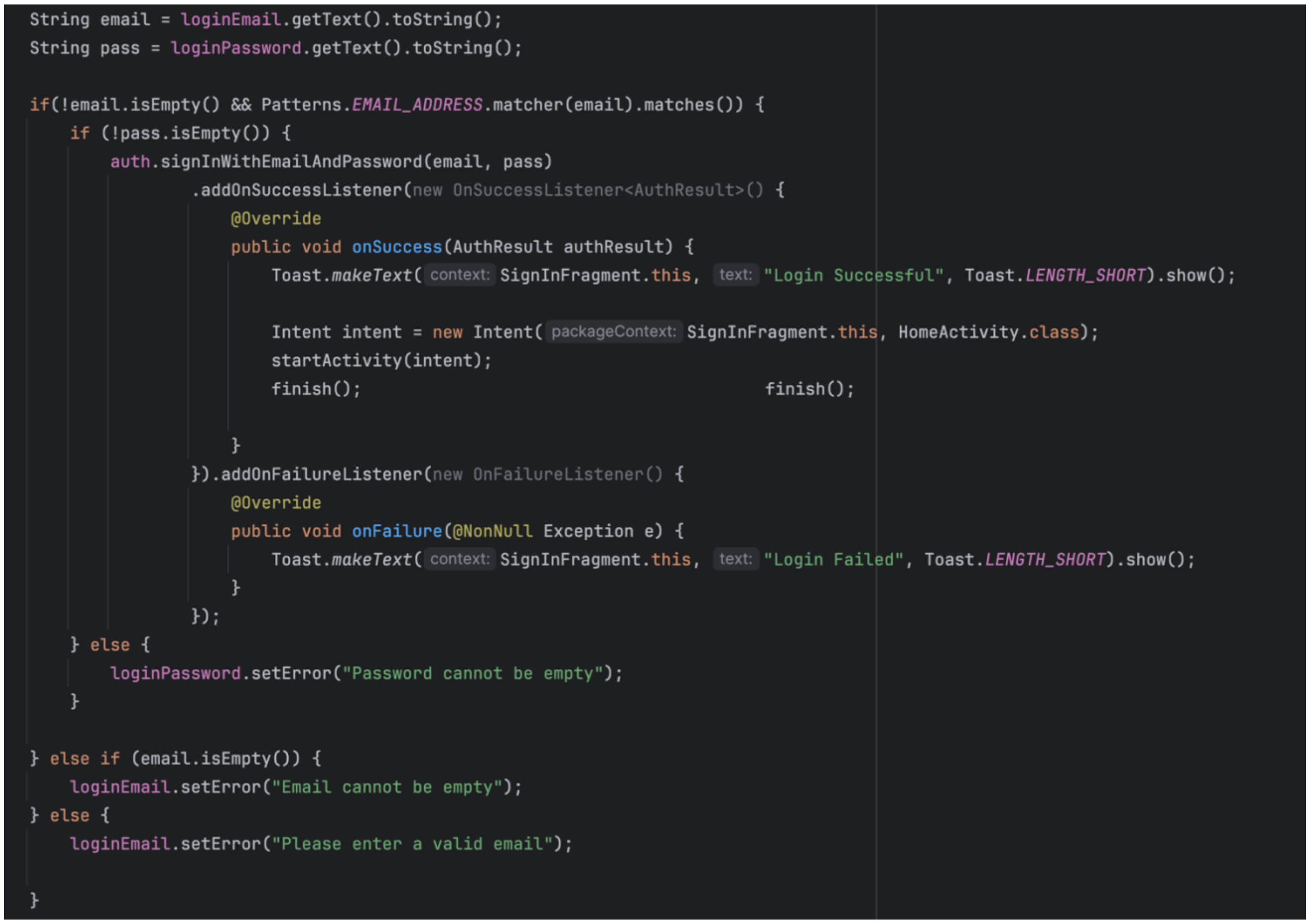
```
final String email = signupEmail.getText().toString().trim();
final String pass = signupPassword.getText().toString().trim();

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);
```

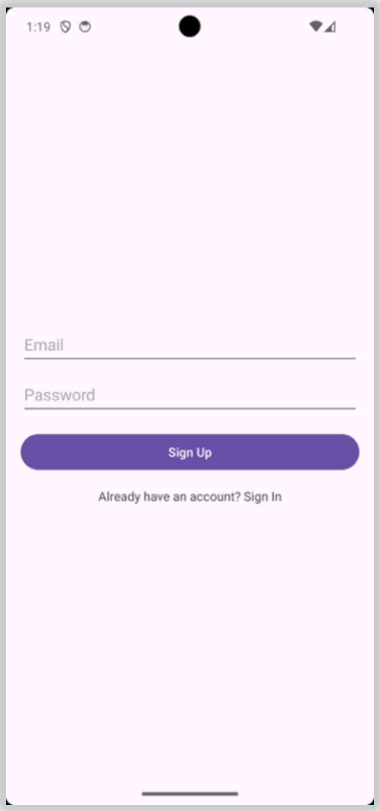
Log In:

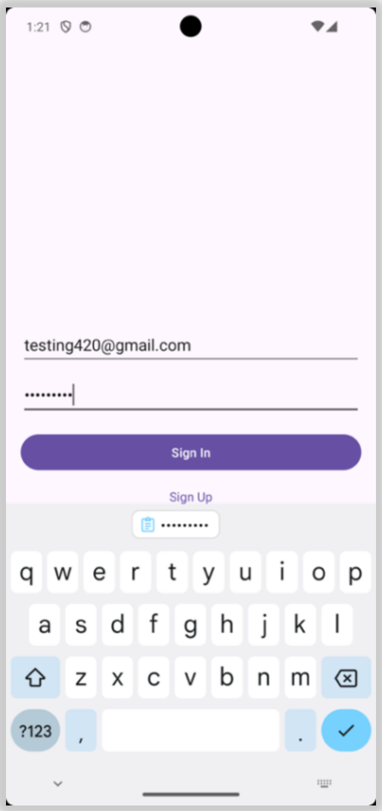


```java
String email = loginEmail.getText().toString();
String pass = loginPassword.getText().toString();

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:

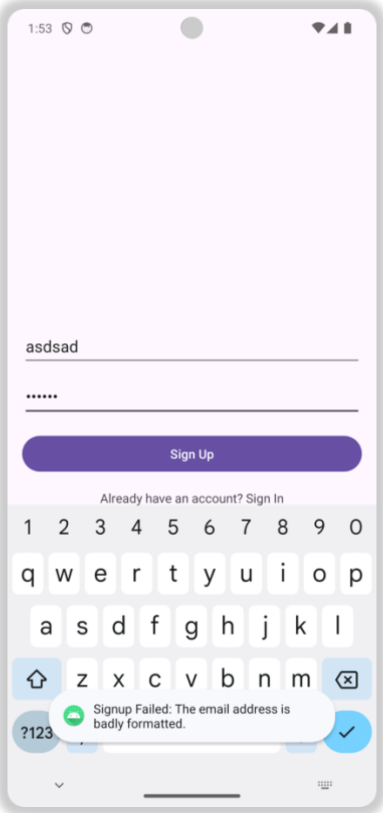




testing420@gmail.com

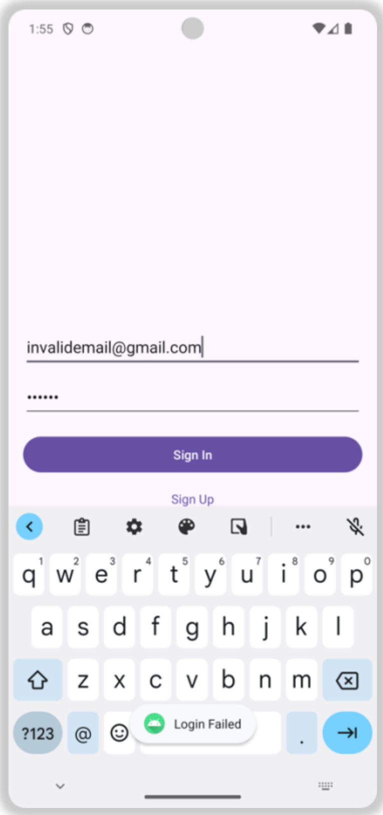
Error

Handling:



asdsad

Signup Failed: The email address is badly formatted.

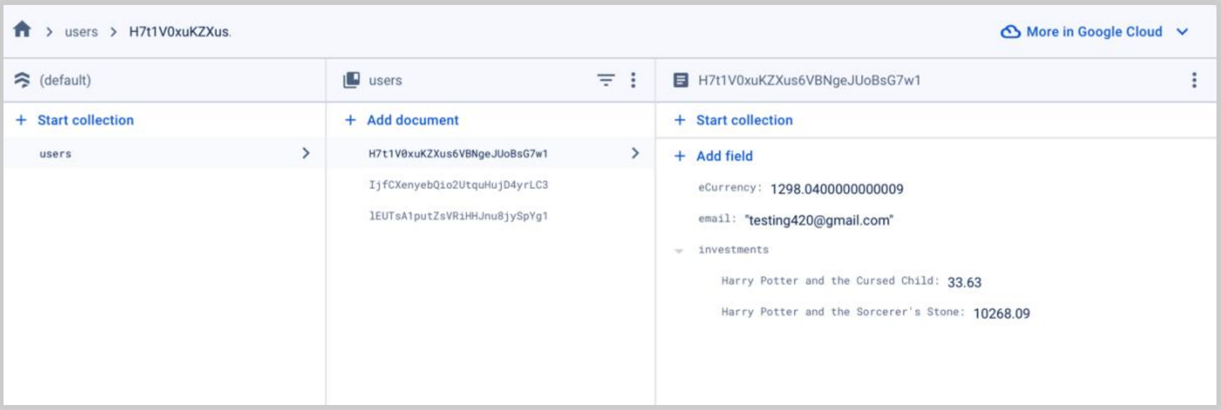


invalidemail@gmail.com

Login Failed

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

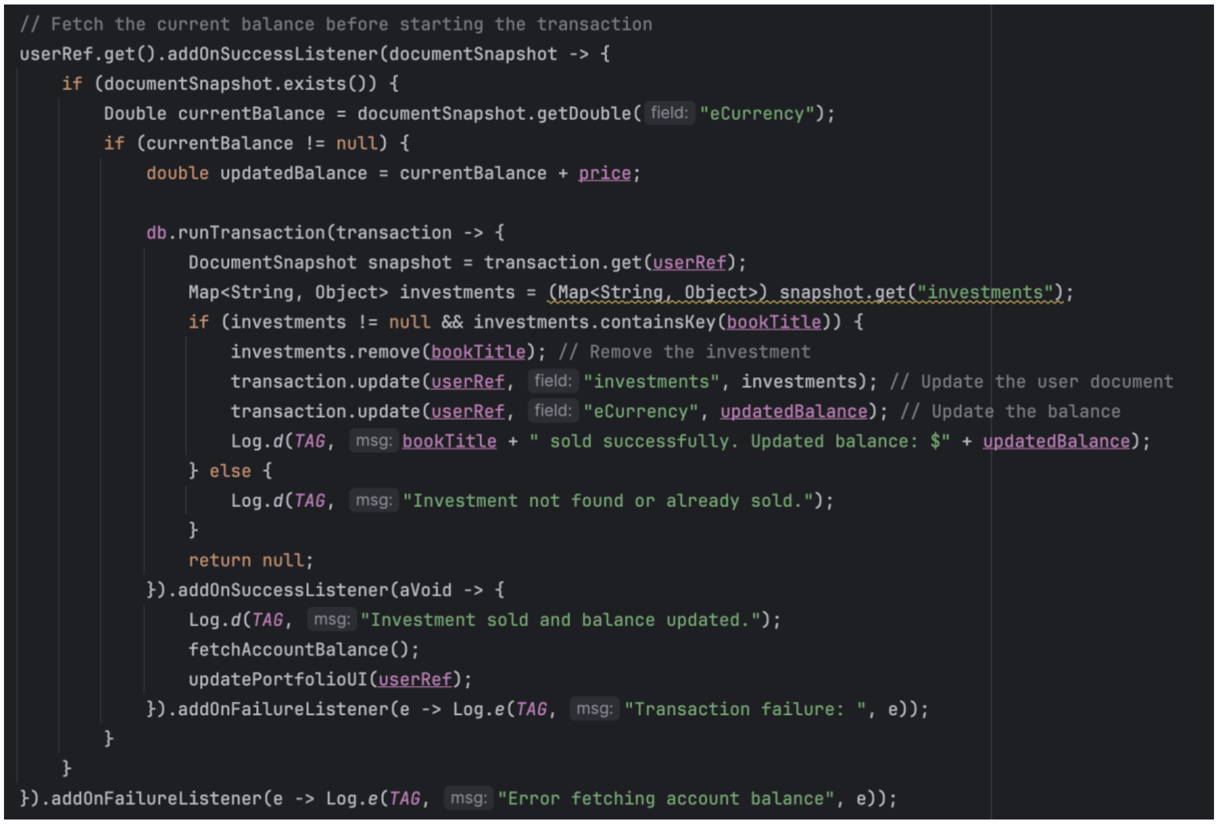
Data Structure:



```
users > H7t1V0xuKZXus.                                         More in Google Cloud

(default)                 | users                                   | H7t1V0xuKZXus6VBNgeJUoBsG7w1
+ Start collection        | + Add document                          | + Start collection
users                     | H7t1V0xuKZXus6VBNgeJUoBsG7w1            | + Add field
                          | IjfCXenyebQio2UtquHujD4yrLC3            | eCurrency: 1298.0400000000009
                          | lEUTsA1putZsVRiHHJnu8jySpYg1            | 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:

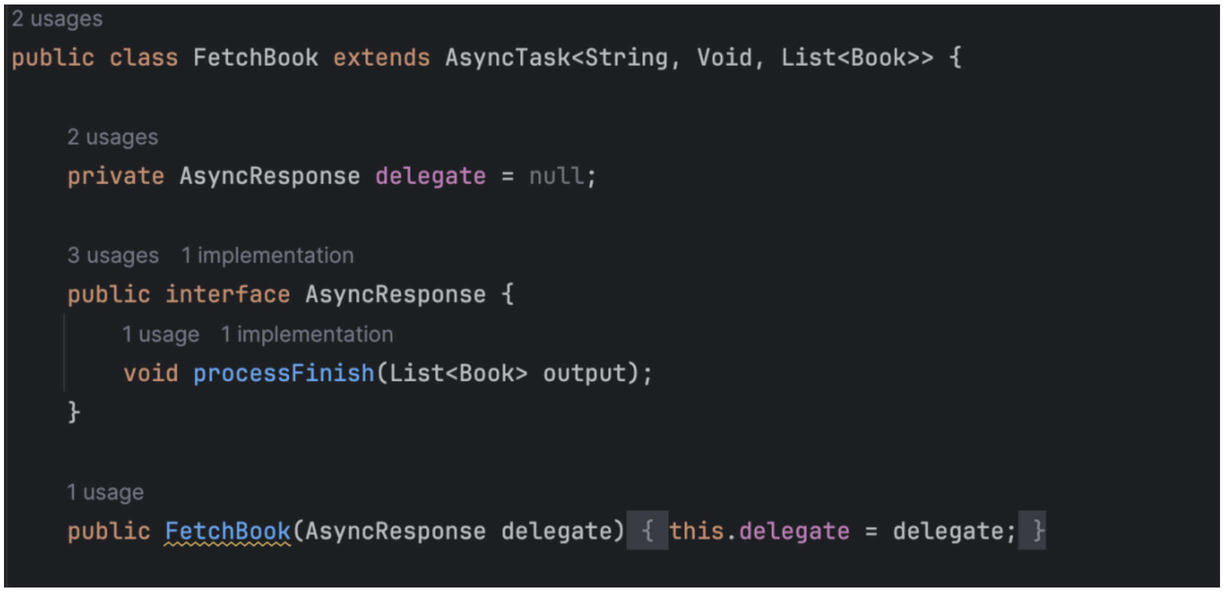


```java
// Fetch the current balance before starting the transaction
userRef.get().addOnSuccessListener(documentSnapshot -> {
    if (documentSnapshot.exists()) {
        Double currentBalance = documentSnapshot.getDouble( field: "eCurrency");
        if (currentBalance != null) {
            double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

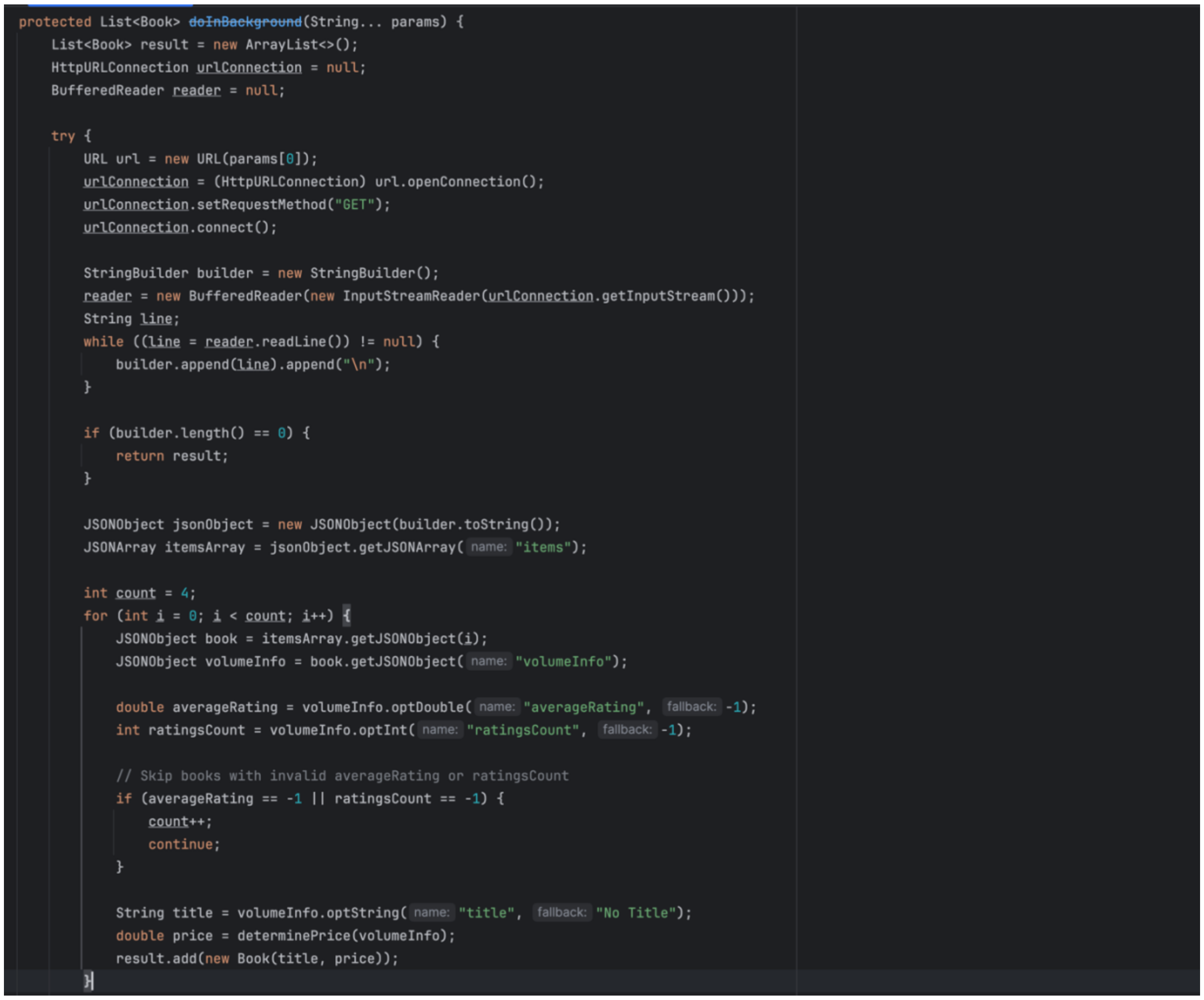
FetchBook Class:



```
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; }
```

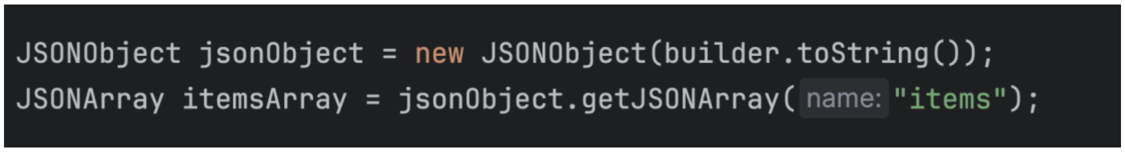
The purpose of FetchBook is to efficiently manage the retrieval and processing of data from the Google Books API. The doInBackground method operates asynchronously in the background, ensuring the application remains responsive. The AsyncResponse interface is a custom callback mechanism that enables communication 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:



```
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.

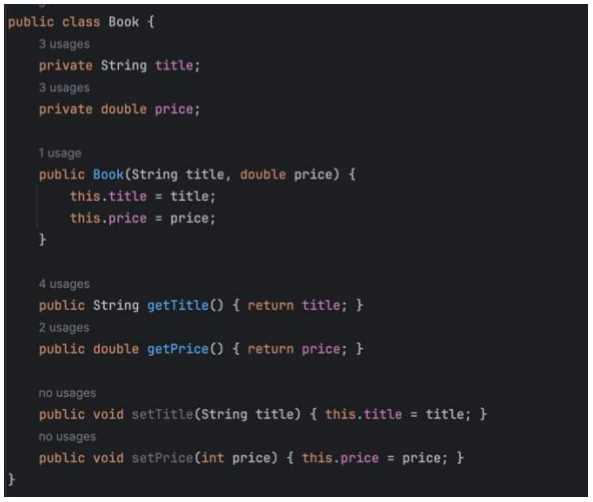


```
JSONObject jsonObject = new JSONObject(builder.toString());
JSONArray itemsArray = jsonObject.getJSONArray( name: "items");
```

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.

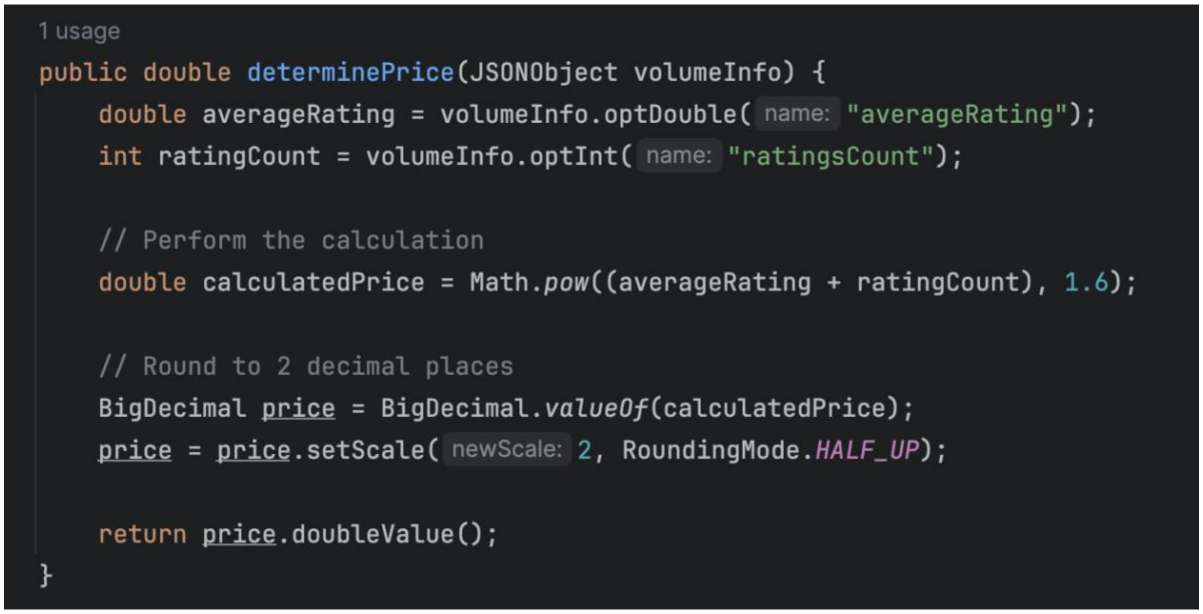
Book Class:

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