Reflection Report

Submission Info

- Name: Levis Ishimwe
- Email: i.levis@alustudent.com
- GitHub: https://github.com/levishimwe/bookswap
- Demo Video: https://www.youtube.com/watch?v=Fb6FazMZJpw
- Date: November 9, 2025

BookSwap Firebase Integration - Reflection Document

Student Information

- **Project**: BookSwap - Book Exchange Platform
- **Course**: Individual Assignment 2
- **Date**: November 9, 2025

Firebase Integration Experience

Overview

This document reflects on the experience of connecting the BookSwap Flutter application to Firebase, including challenges encountered, error messages, and solutions implemented.

---

1. Initial Firebase Setup

Challenge 1: Firebase CLI Configuration

Error Encountered:

```
FirebaseCommandException: An error occurred on the Firebase CLI when
attempting to run a command.
COMMAND: firebase --version
ERROR: 'firebase' is not recognized as an internal or external
command
```

Screenshot Location: `docs/errors/firebase-cli-error.png`

Solution: 1. Installed Firebase CLI globally using npm: `bash npm install -g firebase-tools` 2. Verified installation: `bash firebase --version` 3. Logged in to Firebase: `bash firebase login` 4. Initialized Firebase in the Flutter project: `bash flutterfire configure`

Lesson Learned: Always ensure Firebase CLI is installed before running `flutterfire configure`. The Firebase CLI is essential for connecting Flutter projects to Firebase services.

---

2. Firebase Authentication Implementation

Challenge 2: Email Verification Not Working

Error Encountered:

```
PlatformException(ERROR_INVALID_CUSTOM_TOKEN, The custom token
format is incorrect. Please check the documentation., null, null)
```

Screenshot Location: `docs/errors/auth-verification-error.png`

Root Cause: - Initially tried to implement email verification without proper Firebase Auth configuration - Missing email verification template in Firebase Console

Solution: 1. Configured email verification in Firebase Console: - Navigated to Authentication > Templates - Enabled and customized “Email verification” template - Set proper from address and subject line

2. Implemented proper verification flow in code:

```dart
Future<void> sendEmailVerification() async {
  try {
    final user = _auth.currentUser;
    if (user != null && !user.emailVerified) {
      await user.sendEmailVerification();
    }
  } catch (e) {
    _errorMessage = 'Failed to send verification email';
  }
}
```

3. Created dedicated `VerifyEmailScreen` to guide users through verification process

Lesson Learned: Email verification requires both backend configuration (Firebase Console) and frontend implementation (Flutter code). Always test the entire flow end-to-end.

---

3. Firestore Database Operations

Challenge 3: Permission Denied on Firestore Writes

Error Encountered:

```
[cloud_firestore/permission-denied] The caller does not have
permission to execute the specified operation.
```

Screenshot Location: `docs/errors/firestore-permission-error.png`

Root Cause: - Initial Firestore security rules were too restrictive - Rules didn't allow authenticated users to create documents

Solution: 1. Updated Firestore security rules:

```
        rules_version = '2';
        service cloud.firestore {
          match /databases/{database}/documents {
            // Books collection
            match /books/{bookId} {
              allow read: if true; // Anyone can browse books
              allow create: if request.auth != null; // Authenticated users
can post
              allow update, delete: if request.auth.uid ==
resource.data.ownerId;
            }

            // Swaps collection
            match /swaps/{swapId} {
              allow read: if request.auth != null;
              allow create: if request.auth != null;
              allow update: if request.auth.uid == resource.data.senderId
                            || request.auth.uid == resource.data.receiverId;
            }

            // Chats collection
            match /chats/{chatId} {
              allow read, write: if request.auth != null &&
                request.auth.uid in resource.data.participantIds;

              match /messages/{messageId} {
                allow read, write: if request.auth != null;
              }
            }

            // Users collection
            match /users/{userId} {
              allow read: if request.auth != null;
              allow write: if request.auth.uid == userId;
            }
          }
        }
```

2. Tested rules using Firebase Console Rules Playground

Lesson Learned: Firestore security rules are critical. Start with restrictive rules and gradually open permissions as needed. Always use `request.auth.uid` to verify ownership.

---

4. Real-time Data Synchronization

Challenge 4: StreamBuilder Not Updating

Error Encountered: - No error message, but UI wasn't updating when Firestore data changed - Books added by other users didn't appear without app restart

Screenshot Location: `docs/errors/stream-not-updating.png`

Root Cause: - Used `Future` instead of `Stream` in some Firestore queries - Provider wasn't properly listening to Firestore streams

Solution: 1. Changed all list queries to use Firestore streams:

```
        Stream<List<BookModel>> getAllBooks() {
          return _firestore
              .collection('books')
              .where('isAvailable', isEqualTo: true)
              .orderBy('postedAt', descending: true)
              .snapshots()
              .map((snapshot) =>
                  snapshot.docs.map((doc) =>
BookModel.fromFirestore(doc)).toList());
        }
```

2. Used `StreamBuilder` in UI:

```
        StreamBuilder<List<BookModel>>(
          stream: bookProvider.getAllBooksStream(),
          builder: (context, snapshot) {
            if (!snapshot.hasData) return LoadingIndicator();
            return ListView.builder(...);
          },
        )
```

3. Initialized streams in provider's `initialize()` method:

```
        void initialize() {
          _firestoreService.getAllBooks().listen((books) {
            _allBooks = books;
            notifyListeners();
          });
        }
```

Lesson Learned: For real-time updates, always use Firestore `.snapshots()` which returns a `Stream`, not `.get()` which returns a one-time `Future`.

---

5. Image Storage Strategy

Challenge 5: Firebase Storage Quota and Academic Requirements

Error Encountered:

```
Teacher requirement: "must not use any URL on uploading book cover
image"
```

Screenshot Location: `docs/errors/storage-requirement.png`

Initial Approach: - Tried using Firebase Storage with `uploadBytes()` and download URLs - Teacher rejected this approach due to academic integrity concerns

Solution: 1. Pivoted to base64 inline storage:

```
// Compress image before encoding
Future<Uint8List> _compressBytes(Uint8List input) async {
  final image = img.decodeImage(input);
  if (image == null) return input;

  const maxWidth = 800;
  final resized = image.width > maxWidth
      ? img.copyResize(image, width: maxWidth)
      : image;

  return Uint8List.fromList(img.encodeJpg(resized, quality: 80));
}

// Store in Firestore
final imageBase64 = base64Encode(compressedBytes);
await _firestore.collection('books').add({
  'imageBase64': imageBase64,
  // ... other fields
});
```

2. Display using `Image.memory()`:

```
if (book.imageBase64 != null) {
  final bytes = base64Decode(book.imageBase64!);
  return Image.memory(bytes, fit: BoxFit.cover);
}
```

Trade-offs: - **Pro**: No external storage service needed, images always available - **Con**: Firestore document size limit (1MB), requires compression - **Pro**: Simpler security model (no Storage rules needed) - **Con**: Slightly slower queries for large datasets

Lesson Learned: Sometimes requirements force creative solutions. Base64 storage works well for moderate-sized images with proper compression.

---

6. State Management with Provider

Challenge 6: Widget Not Rebuilding After State Change

Error Encountered: - Buttons remained “loading” even after operation completed - Delete/Edit actions didn’t reflect immediately in UI

Screenshot Location: `docs/errors/state-not-updating.png`

Root Cause: - Forgot to call `notifyListeners()` after state changes - Used `Provider.of<T>(context, listen: false)` when should have listened

Solution: 1. Always call `notifyListeners()` after state mutations:

```
Future<bool> deleteBook(String bookId) async {
  try {
    _setLoading(true); // This calls notifyListeners()
    await _firestoreService.deleteBook(bookId);
    _setLoading(false); // This also calls notifyListeners()
    return true;
  } catch (e) {
    _errorMessage = e.toString();
    _setLoading(false);
    return false;
  }
}

void _setLoading(bool value) {
  _isLoading = value;
  notifyListeners(); // Key!
}
```

2. Use `Consumer` widget for automatic rebuilds:

```
Consumer<BookProvider>(
  builder: (context, bookProvider, _) {
    if (bookProvider.isLoading) return LoadingIndicator();
    return ListView(...);
  },
)
```

Lesson Learned: Provider pattern requires discipline. Always notify listeners after state changes, and use `Consumer` or `context.watch()` for reactive updates.

---

7. Chat Real-time Messaging

Challenge 7: Messages Not Appearing for Receiver

Error Encountered: - Sender could see messages immediately - Receiver had to refresh app to see new messages

Screenshot Location: `docs/errors/chat-sync-error.png`

Root Cause: - Chat messages subcollection not properly queried with `.snapshots()` - Provider wasn't streaming messages in real-time

Solution: 1. Implemented streaming messages:

```
Stream<List<MessageModel>> getMessages(String chatId) {
  return _firestore
      .collection('chats')
      .doc(chatId)
      .collection('messages')
      .orderBy('timestamp', descending: true)
      .snapshots()
      .map((snapshot) => snapshot.docs
          .map((doc) => MessageModel.fromFirestore(doc))
```

```
            .toList());
    }
```

2. Used StreamBuilder in chat screen:

```
StreamBuilder(
  stream: chatProvider.getMessages(widget.chatId),
  builder: (context, snapshot) {
    if (!snapshot.hasData) return LoadingIndicator();
    final messages = snapshot.data!;
    return ListView.builder(...);
  },
)
```

Lesson Learned: Subcollections work exactly like root collections - use `.snapshots()` for real-time updates.

---

8. Email Notifications (Mailto Approach)

Challenge 8: Server-side Functions Not Allowed

Requirement: Teacher prohibited JavaScript code in repository

Initial Approach: - Built Cloud Functions with Nodemailer for automatic emails - Functions folder with Node.js code

Solution: 1. Removed Cloud Functions completely 2. Implemented client-side `mailto:` links:

```
try {
  final owner = await _firestoreService.getUser(book.ownerId);
  if (owner != null && owner.email.isNotEmpty) {
    final subject = Uri.encodeComponent('Swap offer for
"${book.title}"');
    final body = Uri.encodeComponent(
        'Hello ${book.ownerName},\n\n'
        '$senderName has sent you a swap offer...');
    final mailto = Uri.parse('mailto:${owner.email}?
subject=$subject&body=$body');
    await launchUrl(mailto, mode: LaunchMode.externalApplication);
  }
} catch (_) {
  // Ignore email errors
}
```

Trade-offs: - **Pro**: No backend deployment needed, complies with requirements - **Con**: User must manually send email (not automatic) - **Pro**: Works on all platforms (mobile, web, desktop) - **Con**: Receiver must still accept/reject in-app

Lesson Learned: When external services are restricted, leverage platform capabilities. `mailto:` is a simple, universal solution for email notifications.

Summary of Key Learnings

1. **Firebase Setup**: Always install Firebase CLI first, then use `flutterfire configure`
2. **Authentication**: Email verification requires both Console configuration and code implementation
3. **Security Rules**: Start restrictive, test thoroughly, and use `request.auth.uid` for ownership checks
4. **Real-time Sync**: Use `.snapshots()` streams, not `.get()` futures, for live data
5. **State Management**: Always call `notifyListeners()` after mutations; use `Consumer` for reactive UI
6. **Image Storage**: Base64 inline storage is viable with proper compression (800px, 80% quality)
7. **Error Handling**: Catch exceptions gracefully and show user-friendly messages
8. **Testing**: Test on actual devices/emulators, not just web, to catch platform-specific issues
9. **Documentation**: Screenshot every error and solution for future reference
10. **Requirement Adaptation**: Be flexible - when one approach is blocked, find creative alternatives

---

Dart Analyzer Results

Command Run: `dart analyze`

Results: 13 info-level warnings (mostly deprecated API usage with `withOpacity`) - 0 errors - 0 warnings - 13 info messages

Screenshot: See attached `dart_analyzer_report.png`

All info messages are related to Flutter SDK deprecations (using `.withOpacity()` instead of `.withValues()`). These are non-breaking and can be addressed in future refactoring.

Conclusion

Integrating Firebase with Flutter required overcoming several technical challenges, from CLI setup to real-time data synchronization. Each error encountered provided valuable learning experiences about Firebase

architecture, security, and best practices.

The most significant learning was adapting to constraints - when Firebase Storage and Cloud Functions were restricted, we found alternative solutions (base64 storage, mailto links) that still met all functional requirements.

This project demonstrates full-stack mobile development skills: authentication, database operations, real-time updates, state management, and adaptive problem-solving under constraints.

Total Development Time: ~20 hours **Lines of Code**: ~4,500 (Dart) **Firebase Collections**: 6 (users, books, swaps, chats, access_requests, action_tokens) **Screens Implemented**: 15+ **State Providers**: 7 (Auth, Book, Swap, Chat, Settings, AccessRequest, Users)