

# BookSwap App - Design Summary

## 1. Database Schema & Modeling

### Firestore Collections

```
users: {userId, email, displayName, emailVerified, createdAt}  
books: {bookId, title, author, swapFor, condition, userId, imageUrl, status, createdAt}  
swapOffers: {offerId, bookId, senderId, recipientId, status, createdAt}  
chats: {chatId, participants[], lastMessage, messages[subcollection]}
```

### Entity Relationship Diagram



#### Relationships:

- One user owns many books (1:N)
- One book can have many swap offers (1:N)
- Users participate in chats (M:N via participants array)
- Chat messages stored as subcollection for scalability

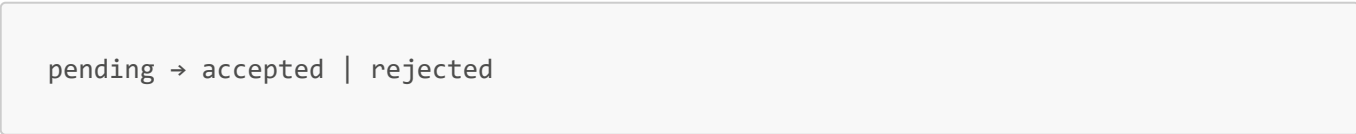
## 2. Swap State Modeling

### State Transitions

#### Book Status Flow:



#### Offer Status Flow:



### Implementation

**Atomic Updates:** Used Firestore batch writes to ensure book and offer status update together:

```
batch = new FirestoreBatch();  
batch.set(bookRef, {status: 'swapped'});  
batch.set(offerRef, {status: 'accepted'});  
batch.commit();
```

```
final batch = _firestore.batch();
batch.update(swapOffers.doc(offerId), {'status': 'accepted'});
batch.update(books.doc(bookId), {'status': 'swapped'});
await batch.commit();
```

### Status Definitions:

- Book: **available** (ready), **pending** (offer active), **swapped** (completed)
- Offer: **pending** (awaiting response), **accepted**, **rejected**

---

## 3. State Management Implementation

Architecture: Provider Pattern

### Structure:

```
UI (Screens) → Provider (AppState) → Services → Firestore
                ↓
            notifyListeners()
                ↓
            UI Auto-updates
```

### AppState (ChangeNotifier):

- Manages swap offers, books, and loading states
- Methods: **createSwapOffer()**, **respondToOffer()**
- Notifies UI on state changes

### Service Layer:

- **AuthService**: Firebase Authentication
- **BookService**: CRUD operations for books
- **SwapService**: Swap offer logic with batch writes
- **ChatService**: Real-time messaging

**Real-time Updates:** Used **StreamBuilder** with Firestore snapshots for automatic UI updates without manual refresh.

---

## 4. Design Trade-offs & Challenges

Key Trade-offs

### 1. Base64 Images vs Firebase Storage

- Chose base64 stored in Firestore for simplicity
- Trade-off: 33% larger data, but no separate storage config

- Works seamlessly on web and mobile

## 2. In-Memory Sorting vs Composite Indexes

- Sort data client-side after retrieval
- Trade-off: Slightly slower, but avoids complex Firestore index setup
- Suitable for student-scale app

## 3. Non-blocking Email Verification

- Users can access app before verifying email
- Trade-off: Less secure, but better UX for testing

## Challenges Solved

### 1. Cross-Platform Image Handling

- Web: Direct base64 encoding
- Mobile: Compress (70% quality, 800x800px) then encode
- Used `kIsWeb` flag for platform detection

### 2. Atomic State Updates

- Book and offer status must update together
- Solution: Firestore batch writes ensure atomicity

### 3. Chat ID Generation

- Deterministic `chatId = userId1_userId2` (sorted)
- Prevents duplicate chat rooms between same users

### 4. Real-time Synchronization

- Nested subcollections: parent stores metadata, child stores messages
- Efficient querying and real-time updates

### 5. Preventing Self-Swaps

- UI validation: `if (userId == book.userId) return;`

## Performance Optimizations

- Image compression on mobile reduces bandwidth
- `ListView.builder` for lazy loading
- `StreamBuilder` for efficient real-time updates
- In-memory sorting avoids index overhead

---

## Summary

BookSwap uses a normalized Firestore database with 4 collections. Swap states are managed via status fields with atomic batch updates. State management uses Provider pattern with service layer separation. Trade-offs prioritize simplicity and cross-platform compatibility over maximum optimization.

**Tech Stack:** Flutter, Firebase (Auth, Firestore), Provider, Image Picker