# A Final Report for Huddle: School Communities App

Selin Uygun

December 24, 2025

## Contents

## 1 What is Huddle?

Huddle is a mobile-first platform for school communities that makes it easy for students and staff to form clubs, manage membership and roles, publish events, and stay informed. It combines a lightweight social feed with structured club administration tools so communities can both promote activity and keep operations manageable.

Core aims:

- Connect students with interest-based clubs and events quickly and discoverably.
- Reduce administrative overhead for club officers via role-based tools and simple workflows.
- Provide a reliable, real-time feed of events and announcements with clear moderation controls.
- Respect privacy and safety through scoped permissions and consultative notification flows.

Key features and workflows:

- **Club management:** Create clubs with metadata (name, avatar, bio); maintain member lists and officer roles (President, Co-President, Board); promote/demote members with controlled permissions; and manage visibility and basic club settings.
- **Events:** Board members can create event posts (date, location, assets), submit them for approval where required, and publish to the club's profile and the global feed. RSVP and save interactions are supported to measure interest.
- **Membership and discovery:** Users can search for clubs and people, follow clubs to surface them in feeds, and request or be granted membership. Discovery is supported by search, profiles, and curated showcases.
- **Moderation and notifications:** Administrative actions (approve/reject posts, change ranks, revoke membership) produce notification events for transparency. The UI provides snackbars and in-app notifications to give immediate user feedback while writes persist to the backend.
- **Design principles:** Prioritise fast, responsive UI (optimistic updates), clear affordances for administrators vs. ordinary members, and accessibility across device sizes.

Technical foundation:

- Built with Flutter for a cross-platform mobile experience.
- Uses Firebase (Firestore, Authentication, Storage) for real-time data, auth, and media hosting.
- Architectural focus on small, testable services (e.g., `PostService`, `UserService`) so UI widgets can remain lightweight and composable.

## 2 Pages Overview

### 2.1 Home Page

The Home Page serves as the primary entry point for the user, acting as a dynamic feed for event discovery. It is implemented as a `StatefulWidget` to manage the asynchronous lifecycle of data fetching from Firebase.

### 2.1.1 Core Contents and Components

The UI is divided into two distinct sections to prioritize immediate engagement:

- **Today's Events (Showcase):** A horizontal carousel implemented via the `PostShowcase` widget. It highlights events occurring on the current calendar date.
- **Popular Events Feed:** A vertical list of `PostCard` widgets displaying all approved upcoming events within the community.
- **Refresh Mechanism:** Wrapped in a `RefreshIndicator`, allowing users to trigger the `_fetchData()` method manually to update the feed.

### 2.1.2 Logic and Functional Workflows

The logic within `home.dart` focuses on data synchronization and state consistency:

**Asynchronous Data Fetching:** Upon initialization (`initState`), the page invokes a private method `_fetchData()`. This method coordinates multiple calls to the `PostService` and `UserService` to retrieve:

1. A list of all approved posts.
2. The current user's set of liked post IDs.
3. The list of clubs/communities the user is currently following.

**Date-Based Filtering:** To populate the Showcase section, the page applies a filter on the retrieved posts: posts.where((p) => isSameDay(p.eventDate, DateTime.now())).toList(). This ensures that only relevant, time-sensitive content appears at the top of the feed.

**Optimistic UI State Management:** For interactions like "Favoriting" a post or "Following" a club, the page utilizes optimistic updates. When a user toggles a favorite:

- The local `likedPosts` set is updated immediately.
- `setState()` is called to reflect the change in the UI.
- An asynchronous call to `PostService.toggleLike()` is made in the background to persist the change in Firestore.

**Navigation Logic:** The page handles deep navigation to post details and community profiles, passing the necessary `PostData` objects to maintain context without redundant API calls.

## 2.2 Search Page

The Search Page provides a unified interface for locating both user profiles and community organizations. It leverages a tabbed navigation system to separate different search domains while maintaining a consistent query interface.

### 2.2.1 Core Contents and Components

The Search Page structure is designed around a `DefaultTabController` and consists of:

- **Search Header:** A persistent top section containing a `TextField` for input, stylized with a custom `InputDecoration` and a search icon.
- **Categorical Tabs:** A `TabBar` with two primary sections: "Communities" and "Users". This provides a clear semantic separation for the user's intent.

- **Dynamic Results View:** A `TabBarView` that swaps between the `ClubSearch` and `UserSearch` widgets. These specialized widgets handle the rendering logic for their respective data types.

#### 2.2.2 Logic and Functional Workflows

The logic in `search.dart` emphasizes efficient query propagation and UI responsiveness:

**Query State Management:** The page utilizes a `TextEditingController` to track user input. To ensure that child widgets remain synchronized with the query, the input is passed down as a parameter or managed through state updates whenever the text changes.

**Delegated Searching:** Rather than performing all search logic in a single file, `search.dart` acts as a router or coordinator.

- When the "Communities" tab is active, the `ClubSearch` widget is responsible for querying the `CommunityService` or Firestore collection.
- When the "Users" tab is active, the `UserSearch` widget interfaces with the `UserService`.

**Performance Considerations:** The implementation focuses on reducing unnecessary builds. By separating the search views into their own widgets (`ClubSearch` and `UserSearch`), only the relevant portion of the tree is rebuilt as the user types or switches tabs.

**UI Aesthetics:** The page maintains design consistency with the rest of the application, utilizing custom colors for tab indicators and ensuring a responsive layout that scales across different screen dimensions.

### 2.3 Favorite Page

The Saved Page acts as a personal archive for users, displaying a curated list of events they have previously marked with a "like" or "save" interaction.

#### 2.3.1 Core Contents and Components

The interface is designed for simplicity and focuses on high-intent content:

- **Saved Events List:** A vertical `ListView` populated by `PostCard` widgets. This maintains visual parity with the Home Page feed to ensure a consistent user experience.
- **Empty State Handler:** A conditional rendering block that displays a friendly message (e.g., "No saved events yet") when the user's liked collection is empty, preventing a blank screen.
- **Pull-to-Refresh:** Integrated via `RefreshIndicator` to allow users to synchronize their saved list with the latest database state manually.

#### 2.3.2 Logic and Functional Workflows

The logic in `saved.dart` revolves around user-specific data filtering and real-time synchronization:

**User-Centric Data Retrieval:** Unlike the Home Page, which fetches all approved posts, `saved.dart` performs a targeted query. It typically interfaces with `PostService.getLikedPosts(userId` to retrieve only the documents that the current authenticated user has interacted with.

**Post State Synchronization:** Because a user can "unlike" a post directly from the Saved Page, the logic must handle list modification. When a post is unliked, the page should remove the item from the local list and trigger `setState()` to reflect the change immediately.

**Dependency on Auth State:** The page is heavily dependent on a valid `userId`. It includes logic to ensure that data is only fetched once the Firebase Authentication state is confirmed, preventing null pointer exceptions during the initial build cycle.

## 2.4 Profile Page

The Profile Page is a comprehensive module that manages user identity, social statistics, and the display of user-contributed content. It is designed to be polymorphic, handling both the current user's own profile and public views of other users' profiles.

### 2.4.1 Core Contents and Components

The layout follows a standard social media architecture:

- **Header Section:** Displays the user's `CircleAvatar`, display name, and a short bio. It also includes a statistics row showing counts for "Posts", "Followers", and "Following".
- **Action Buttons:** Dynamically switches between an "Edit Profile" or "Logout" button (for the owner) and a "Follow/Unfollow" toggle (for visitors).
- **Tabbed Content View:** Utilizes a `TabBar` to toggle between:
  1. **My Posts:** A grid or list showing events created or managed by the user.
  2. **My Communities:** A list of clubs and organizations the user is a member of or follows.

### 2.4.2 Logic and Functional Workflows

The `profile.dart` logic coordinates complex state transitions and external service calls:

**Polymorphic Profile Loading:** The widget often accepts an optional `targetUserId`. If null, it defaults to `auth.currentUser.uid`. The `_fetchUserData()` method then branches its logic to fetch specific profile details and permissions based on this ID.

**Profile Editing Logic:** When in "Edit Mode", text fields replace static labels. The page manages the state of these inputs and coordinates with `UserService.updateProfile()` to persist changes. It also handles image picking logic for updating profile pictures via the `image_picker` plugin.

**Social Interaction Management:** The follow/unfollow logic utilizes an optimistic update pattern similar to the Home Page. It modifies the local follower count and button state immediately while sending a request to the `UserService` to update the Firestore relationship documents.

**Post Management:** For the owner, this page serves as a management hub. It provides entry points to edit or delete existing posts, leveraging the `PostService` to handle deletions and ensuring the UI is refreshed upon completion.

# 3 Main Widgets

## 3.1 Management Tab

The `tab_manage.dart` widget implements the administrative backbone of the community module. From an architectural perspective, it serves as a Role-Based Access Control (RBAC) hub, where the application's logic branches significantly depending on the user's specific rank within a club's hierarchy.

### 3.1.1 Hierarchical Rank Definitions

The system distinguishes between three primary administrative tiers: President, Co-President, and Board. These ranks are not merely labels but are tied to specific permission sets defined in the backend schema:

- **President:** The supreme administrative authority. This rank possesses destructive permissions (e.g., deleting the community) and exclusive rights to modify the highest-level metadata and executive appointments.
- **Co-President:** A secondary executive tier. While possessing nearly identical functional capabilities to the President regarding content and member management, the Co-President is restricted from modifying the status of the President or dissolving the organization.
- **Board Member:** The foundational administrative tier. Board members typically handle operational tasks, such as moderating pending posts or viewing member lists, but lack the authority to modify organizational structures or executive ranks.

### 3.1.2 Differential User Experiences and Functional Logic

The technical implementation of `tab_manage.dart` ensures that the "experience" of the application is contextually adapted to the user's authority level through conditional widget rendering and state gating:

**The Management Dashboard:** For **Presidents** and **Co-Presidents**, the interface exposes sensitive settings. This includes the "Edit Community" view, where they can update logos, descriptions, and visibility. The code utilizes conditional logic to verify rank before displaying the `ListTile` associated with executive settings.

**Promotion and Demotion Logic:** A critical differentiator exists in member management. The logic within the member list view allows a **President** to promote a member to **Co-President**. However, if the current user is a **Co-President**, the UI logic restricts them from promoting others to their own level or demoting the President, effectively enforcing a "ceiling" of authority.

**Content Moderation Workflows:** All three ranks share the "Pending Posts" experience. The widget fetches a stream of posts with a status of `pending`. Technically, this is implemented via a `StreamBuilder` that listens to a specific Firestore query. Any administrative user can approve or reject these posts, triggering the `PostService.updateStatus()` method.

**Data Integrity and Security:** From a security standpoint, `tab_manage.dart` acts as a front-end guard. While the UI hides buttons based on rank, the underlying services (such as `UserService`) require the rank to be passed as a parameter to ensure that the backend validates the request against the user's actual document in the `members` sub-collection.

### 3.1.3 Technical Implementation of Rank Checks

The widget maintains a `String currentUserRank` state variable. Upon loading, it synchronizes with the database:

```
// Pseudocode of the rank-checking logic
if (currentUserRank == 'President') {
  _showExecutiveSettings();
} else if (currentUserRank == 'Co-President' || currentUserRank == 'Board') {
  _showOperationalTools();
}
```

This categorical check ensures that the cognitive load on lower-ranked administrators is reduced by hiding irrelevant executive tools, while maintaining strict security for high-level operations.

## 3.2 Club Profile

The `club_profile.dart` module functions as the public-facing landing page for individual communities. It is technically distinct from the management tab as it focuses on social engagement, public metadata display, and membership conversion.

### 3.2.1 Architectural Components

The widget is structured as a `Sliver`-based layout to provide a professional, collapsing header effect common in modern mobile interfaces:

- **SliverAppBar:** Contains the community's banner image and logo, which remains visible while scrolling to maintain brand identity.
- **Engagement Metadata:** A statistics block displaying the member count and total events, acting as social proof for prospective members.
- **Membership Toggle:** A contextual primary action button. If the user is a non-member, it displays "Join"; if they are already a member, it provides access to the management dashboard or an "Exit Community" option.

### 3.2.2 Technical Logic and State Transitions

The complexity of `club_profile.dart` lies in its real-time reactivity to membership changes:

**Reactive Membership Streams:** The widget utilizes a `StreamBuilder` to listen to the specific membership sub-collection in Firestore. This ensures that when a user taps "Join", the UI updates the member count and button state instantly across all instances of the application.

**Rank-Based Action Routing:** The logic for the primary action button is branched. If the `UserService` identifies the user as an administrator (`President`, `Co-President`, or `Board`), the button navigates to the `TabManage` widget. For standard members, it handles general interaction.

**Content Partitioning:** The feed displayed on the profile is restricted to posts where the `communityId` matches the current profile. This involves a filtered Firestore query:

```
// Logic for club-specific feed retrieval
Query clubPosts = firestore.collection('posts')
                           .where('communityId', isEqualTo: targetClubId)
                           .orderBy('timestamp', descending: true);
```

**Experience Differentiators:** The "Experience Gating" architecture fundamentally alters user interaction based on membership state. A non-member's experience is restricted to content consumption and metadata observation. Upon joining, the transition to a member experience is managed by a transactional Firestore update, ensuring atomic synchronization between the user profile and the club document to maintain data integrity.

## 3.3 Notifications

The notification system provides a critical feedback loop within the application, ensuring that users remain informed of relevant organizational updates and social interactions. The system is implemented through a centralized `Notification` model and the `notification.dart` widget.

### 3.3.1 Categorization of Notifications

The system distinguishes between several objective notification types, each mapped to specific data schemas in the `notifications` collection:

- **Event Approval/Rejection:** Sent to users after a board member or president moderates a submitted post.
- **Rank Promotion:** Alerts a member when their administrative status is escalated (e.g., from Member to Board).
- **Community Updates:** General broadcasts regarding changes to a club's metadata or important announcements.
- **Social Interactions:** Notifications triggered by new followers or interactions with saved events.

### 3.3.2 Administrative Triggers in `tab_manage.dart`

The `tab_manage.dart` widget serves as the primary origin for administrative notifications. When high-level actions are performed, notification calls are integrated directly into the transaction logic:

**Moderation Feedback:** Upon approving or rejecting a pending post, `tab_manage.dart` invokes a notification trigger. This informs the original poster of the outcome. Objectively, this involves an `addDoc` operation to the recipient user's `notifications` sub-collection containing the `postId` and `status`.

**Rank Escalation Alerts:** When a President or Co-President promotes a member, the logic calls a utility function to generate a promotion notification. This notification is essential for informing the user that their UI has transitioned from a standard member view to the administrative `TabManage` view.

**Technical Workflow:** These calls are structured as asynchronous background tasks. The UI in `tab_manage.dart` provides immediate feedback to the administrator, while the notification document is propagated to the target user's feed.

### 3.3.3 UI Presentation and State Handling

In `notification.dart`, the system utilizes a `FirestoreListView` (or `StreamBuilder`) to display alerts in reverse-chronological order. Each notification item is technically a stateless widget that renders an icon based on the `type` field, a timestamp, and a deep-link capability that allows users to navigate directly to the relevant event or profile upon interaction.

## 3.4 Users and Clubs in Search

This subsection summarises the two compact row widgets used by the Search Page.

- **User results (`user_search.dart`):** A simple row showing a user's avatar and name, with an optional overflow menu. The menu can expose admin actions (revoke/grant membership) that perform small Firestore updates and show quick feedback; report/block currently show a message. The menu hides for the current user and relies on the parent to supply rank/club context.

- **Club results (`club_search.dart`):** A row displaying the club avatar, name, a short bio, and a follow toggle. Tapping navigates to the club profile; the follow button updates local UI state immediately and calls a parent callback to persist the change (optimistic update).

Both widgets keep data-heavy logic out of their builds: UI is kept local and quick, while persistence and permission checks are delegated to the surrounding page or services for clarity and testability.

## 4 API & Stress Testing

### 4.1 API Unit Tests

The repository contains a focused suite of unit tests that validate the behavior of core service classes without requiring live Firebase services. The unit test file `test/api_test_mockup.dart` instantiates service objects using lightweight fakes and mocks:

- Uses `fake_cloud_firestore` to provide an in-memory Firestore implementation for deterministic reads/writes.
- Uses a small fake for `FirebaseStorage` (a stubbed `Reference`) so storage-related code can return predictable URLs.
- Uses `firebase_auth_mocks` (`MockUser`) to simulate authenticated user data in tests.

Key behaviors covered:

- `PostService.getPost(postId)`: verifies that posts are read and parsed into the domain `Post` object with correct fields (club id, caption, photo URL, date).
- `PostService.getClub(clubId)`: validates retrieval and mapping of club documents to a `Club` model.
- `PostService.getAllPosts(state: 'approved')`: integrates posts and their parent club data, asserting the returned combined structure.
- `PostService.toggleLikePost(userId, postId, currentlyLiked)`: confirms that liking and unliking modify the user's `liked_posts` array as expected.
- `UserService.createUserDocument(user)`: ensures new user documents are created, and existing documents are updated while preserving certain fields (e.g., existing `club_rank`).

Notes and recommendations:

- These tests are fast and reliable because they avoid network I/O; they exercise service logic and Firestore mappings.
- Consider adding negative and edge-case tests (missing fields, malformed URLs, permission-denied scenarios) and tests that simulate storage URL rewriting behavior.
- To run: `flutter test test/api_test_mockup.dart`.

### 4.2 Stress and Microbenchmarks

The file `test/stress_test.dart` implements three microbenchmarks to capture performance characteristics of key operations. The tests are intended as developer-facing reports rather than strict unit tests; they print a human-readable report to stdout.

Scenarios implemented:

- **JSON Parsing (10k items):** Generates synthetic raw JSON maps and maps them to `Post` objects using `Post.fromFirestore`. A `Stopwatch` measures the elapsed time and compares it to an "ideal" threshold (300ms by default).
- **List Processing (20k items):** Builds 20k `Post` objects, filters by club id, and sorts the filtered list by date. This measures CPU cost for common collection operations (filter + sort).
- **Widget Instantiation (50k items):** Allocates many `PostCard` widgets to measure object creation cost and memory churn. Note: these widgets are created as Dart objects but not mounted into a Flutter widget tree, so this scenario measures allocation cost rather than full render performance.

Output and scoring:

- Each scenario prints elapsed time, a performance index (ideal/actual), and a colored PASS/WARN status using ANSI escape sequences for quick visual scanning in terminals.
- The tests are useful as quick local benchmarks, but results will vary by machine, VM, and Dart/Flutter runtime configuration.

Limitations and suggestions:

- The widget-instantiation scenario does not exercise rendering, layout, or rasterization costs. For frame-time and UI performance profiling, prefer instrumented widget tests (`WidgetTester`) or integration benchmarks (`flutter_driver / integration_test`).
- For more robust performance tracking, consider converting these microbenchmarks into standalone Dart scripts (outside the test harness) and collecting time-series data, or use the Dart Observatory / DevTools for CPU and memory profiling.
- Large counts (20k–50k) may be heavy for CI runners; reduce counts or gate these tests behind a manual flag or an environment variable when running in constrained environments.

How to run these benchmarks:

```
flutter test test/api_test_mockup.dart
flutter test test/stress_test.dart
```

Together, these files provide both unit-level correctness checks for API-layer logic and practical developer-facing benchmarks to spot regressions in parsing, collection processing, and allocation patterns.

### 4.3 Real Firebase API Tests (`test/api_test_real.dart`)

For completeness, the repository includes `test/api_test_real.dart`, which attempts to exercise the real Firebase Cloud Firestore and Storage APIs from a test harness. This file mirrors the unit-test scenarios (connectivity checks, CRUD on posts/clubs, likes/follows, query filters, stress/bulk operations) but targets the actual backend instead of fakes.

**Platform constraint.** Running `flutter test` on the Dart VM does not provide a device or platform channels required by Firebase plugins. As a result, real API tests will typically fail with errors such as:

- `[core/no-app] No Firebase App '[DEFAULT]' has been created`

- `PlatformException(channel-error, Unable to establish connection ...)`

These errors stem from plugin initialization not being available in a pure VM context. Real API tests should run on a device/emulator using the integration test framework.

**Recommended approach.** We provide a device-based counterpart at `integration_test/api_integration` It initializes Firebase via `IntegrationTestWidgetsFlutterBinding` and `Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform)`, then executes the same scenarios against the live backend, cleaning up documents created in `users`, `clubs`, and `posts`.

## 4.4 Test README Summary

The test folder includes a `README.md` that distinguishes unit vs. integration tests and provides run instructions:

- **Prerequisites:** Flutter SDK; Firebase configured (`lib/firebase_options.dart`, Android `google-services.json`); network connectivity; an Android emulator or device for integration tests.
- **Run unit tests (VM):**

  ```
  flutter test test
  flutter test test/api_test_mockup.dart
  flutter test test/stress_test.dart
  ```

- **Run real Firebase integration tests (device/emulator):**

  ```
  flutter pub get
  flutter devices
  flutter test integration_test/api_integration_test.dart -d <deviceId>
  ```

- **Optional emulator:** To avoid touching production data, point Firestore to the emulator in test setup with `FirebaseFirestore.instance.useFirestoreEmulator(host, port)`.
- **Cleanup and troubleshooting:** Integration tests delete created documents; if a run aborts, leftover test docs can be removed safely. Errors like `channel-error` or `no-app` indicate tests were run in the VM instead of on a device; use the integration test.

This structure ensures fast, deterministic unit-level coverage (using fakes) and faithful end-to-end validation (using device-based integration tests) while making the operational differences explicit for developers.