Report

Lost & Found @Campus

Submitted by

Mr. Perawis	Buranasing	6688012
Mr. Phana	Mahachairachun	6688061
Mr. Pathompong	Prasitphol	6688088

Present to

Aj. Dr. Suradej Intagorn

This report is part of

ITCS259_Mobile Application Development, Semester 1 2025

The Faculty of Information and Communication Technology, Mahidol University

List of Contents

1. Introduction and Background

Losing personal belongings on campus is a common problem for university students and staff. Items such as student ID cards, laptops, iPads, AirPods, umbrellas, and water bottles are often misplaced in classrooms, libraries, cafeterias, and common areas. When this happens, owners usually rely on informal channels such as friends, LINE groups, or Facebook posts to ask whether someone has found their belongings. These methods are fragmented, hard to search, and often result in items never being returned.

To address this issue, our team developed **"Lost & Found @Campus"**, a mobile application built with Flutter and Firebase. The app provides a centralized, easy-to-use platform where users can post items they have lost or found, search existing posts, and directly contact the other party via in-app chat.

The app is designed specifically for university environments, where users share the same campus and are more likely to cross paths. By focusing on a single institution, we can apply simple, safe workflows (e.g., sign-in with university email, basic profile information) while keeping the interface clean and friendly. Our goal is to make returning items **fast, safe, and convenient**, and at the same time practice building a real-world mobile application using advanced Flutter libraries and Firebase services.

Target Users and Use Cases

- **Target Users:** Students, teachers, and staff in the university.
- **Main Use Cases:**
    - Post a "Lost" or "Found" item with image and details.
    - Search items by keyword or category.
    - Contact the post owner.
    - Mark item as "Resolved" when it's returned.

Motivation for Building the App

- Real problem happening in universities
- Need for a centralized tool
- Opportunity to practice Flutter + Firebase skills

Relevance of the project

- Users share the same campus
- Higher chance of successfully returning items
- University community is a controlled, trusted environment

2. Related Works

Before designing our solution, we looked at how lost-and-found problems are currently handled:

2.1. Facebook / LINE groups: Many campuses use social media groups where people post lost and found announcements.

- ***Strengths*:** Easy to start, almost everyone already has an account, supports images and comments.
- ***Weaknesses*:** Hard to search by item or location, posts quickly disappear under new posts, no clear status (whether an item has already been returned), and conversations are mixed with unrelated content.

2.2. General marketplace / community apps (e.g., marketplace-style apps and generic community boards)

- ***Strengths*:** Support rich posts with photos, categories, and messaging.
- ***Weaknesses*:** Not specialized for lost-and-found; items are mixed with selling/buying posts, and there is no campus-specific filtering. Privacy could also be a concern when dealing with strangers outside the university.

2.3. Physical lost-and-found counter (e.g., at the security office or faculty office)

- ***Strengths*:** Official and safe, staff can verify identity.
- ***Weaknesses*:** Limited open hours, people might not know where the office is, and there is no quick way to check whether someone already reported or found an item.

Compared with these options, **Lost & Found @Campus** provides:

- A **campus-focused** environment intended only for students, teachers, and staff.
- Clear separation of **“Lost”** and **“Found”** posts, with search and filter features.
- A simple **status update** mechanism where items can be marked as “Found” or “Lost”.
- A built-in **chat** feature to contact the item owner or finder without sharing personal phone numbers or social media accounts publicly.

Our app does not aim to replace official university procedures but to complement them by making it easier for community members to connect.

3. Methodology

3.1 System Architecture

The application follows a **client-backend-as-a-service** architecture:

- **Client (Mobile App)**
  - Developed using **Flutter** (Dart).
  - Runs on Android from a single codebase.
  - Handles UI rendering, input validation, navigation, and local state management.

- **Backend (Firebase)**
  - **Firebase Authentication**: Manages user registration, login, logout, password reset, and secure access to user data using email/password sign-in.
  - **Cloud Firestore**: Stores user profiles, lost/found items, chat rooms, and messages as collections and documents.
  - **Firebase Storage**: Stores uploaded item images; Firestore keeps only the URL.

3.2 Tools and Technologies

- **Programming Language:** Dart
- **Framework:** Flutter
- **Backend Services:** Firebase Authentication, Cloud Firestore, Firebase Storage
- **State Management:** Simple ChangeNotifier / Provider-style pattern integrated inside the Flutter widget tree (e.g., for current user, list of items, and chat messages).
- **IDE and Dev Tools:** Visual Studio Code, Flutter CLI, Android Studio Emulator / physical device.
- **Libraries:**
  - Image picker for selecting a single photo from a gallery or camera.
  - FlutterFire plugins for auth, Firestore, and storage.

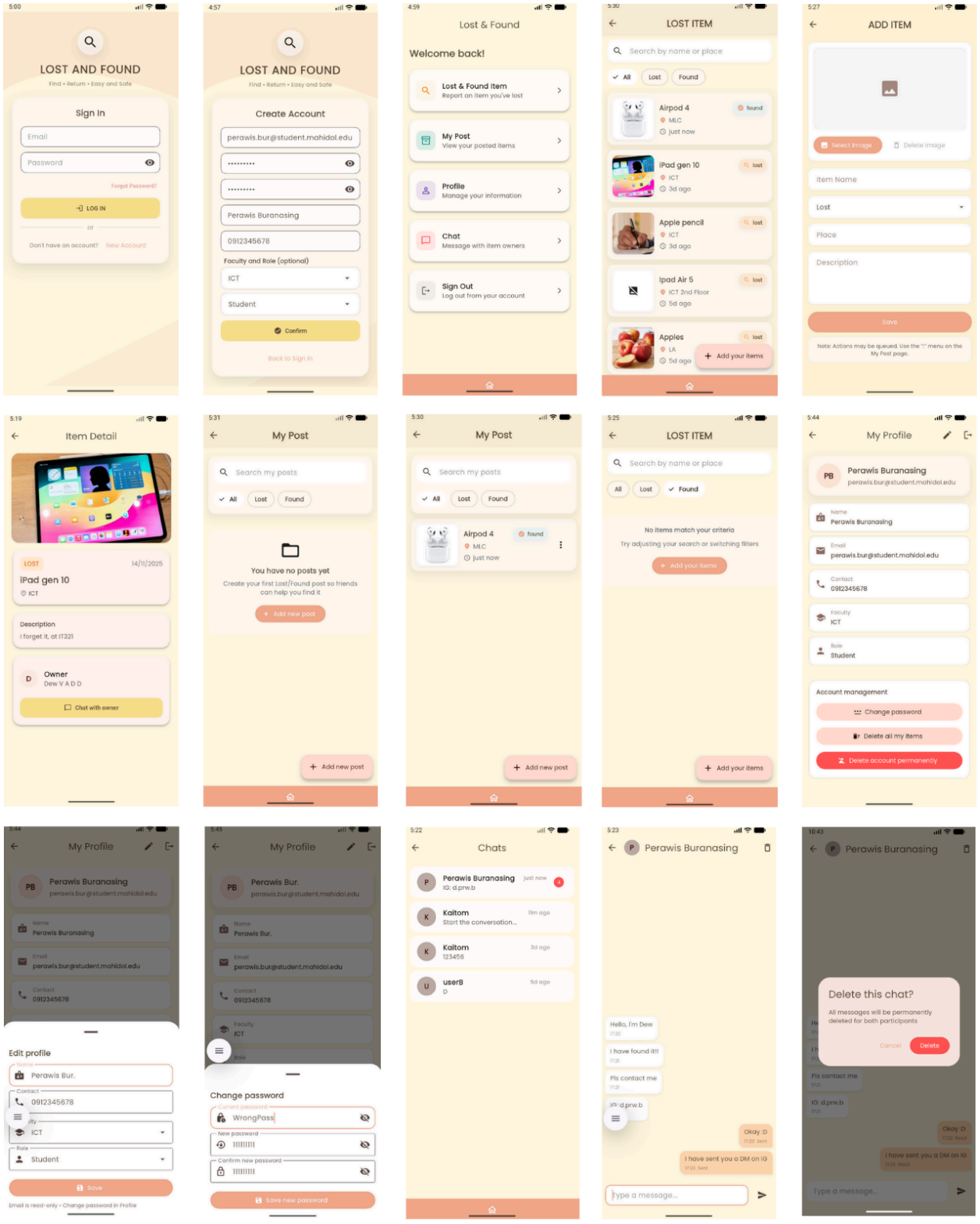
The project is structured as shown in the lib/ folder:

- **main.dart**: Entry point, app initialization, route setup, and Firebase initialization from firebase_options.dart.
- **theme.dart**: Centralizes colors, fonts, and theme configuration for consistent UI.
- **models/**: Data models (e.g., user, item, message) used across the app.
- **screens/**: Each major page in the app:
  - login_page.dart, add_item_page.dart, home_menu_page.dart, lost_list_page.dart, item_detail_page.dart, my_post_page.dart, profile_page.dart, other_profile_page.dart, chat_list_page.dart, chat_page.dart, etc.
- **services/**: e.g. chat_service.dart encapsulating logic to read/write messages in Firestore.
- **widgets/**: Reusable UI components such as app_button.dart, app_input.dart, bottom_home_bar.dart, and item_tile.dart.

This separation follows a simple MVVM style pattern to keep UI, business logic, and data models easier to maintain.

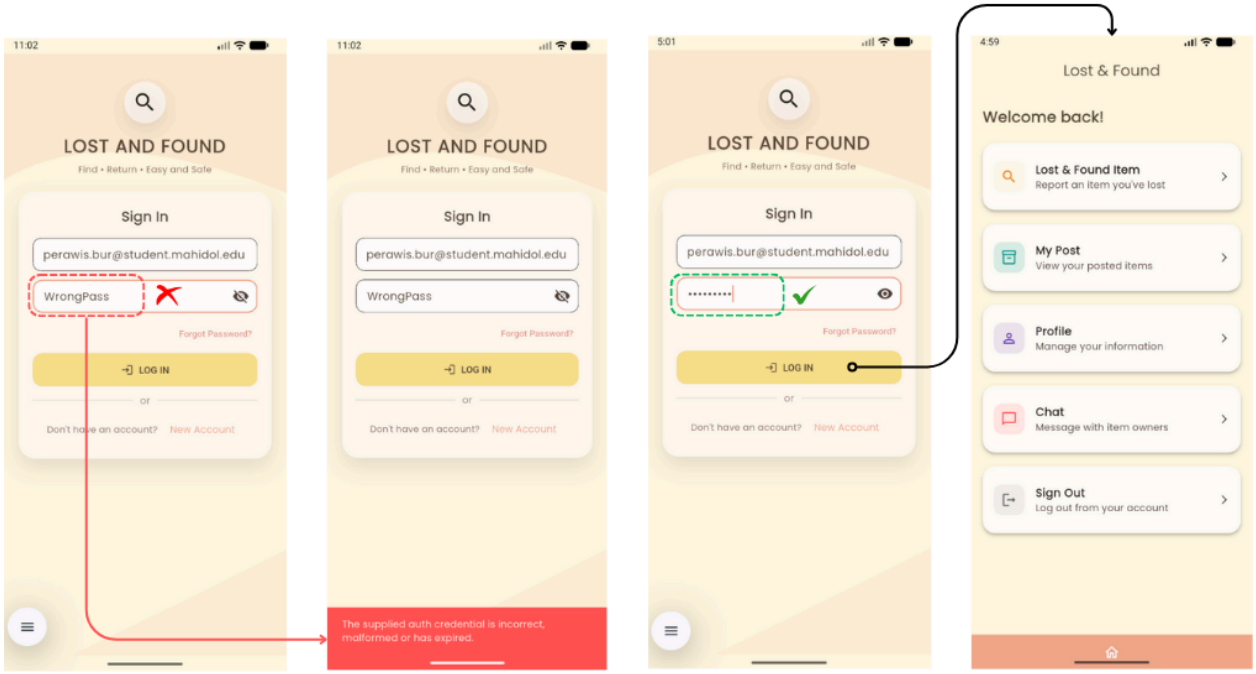
3.3 App Design and Workflow

All UI screens share a consistent pastel color palette with rounded cards, giving the app a friendly and modern look.



3.3.1 Authentication Flow (login_page.dart)

- **User Sign-In & Credential Validation Flow :** This flow illustrates how a user signs in to the Lost & Found mobile application, including incorrect and correct credential scenarios.



1) Accessing the Sign-In Screen

The user arrives on the Sign-In page containing fields for email and password, along with options for “Forgot Password?” and creating a new account. The user begins entering their login information.

2) Incorrect Credentials

In the first scenario, the user inputs an incorrect password. A visual red indicator shows that the password does not meet authentication requirements. After pressing **Log In**, the system validates the credentials with the backend and displays an error message: *“The supplied auth credential is incorrect, malformed or has expired.”* This clearly informs the user that login has failed.

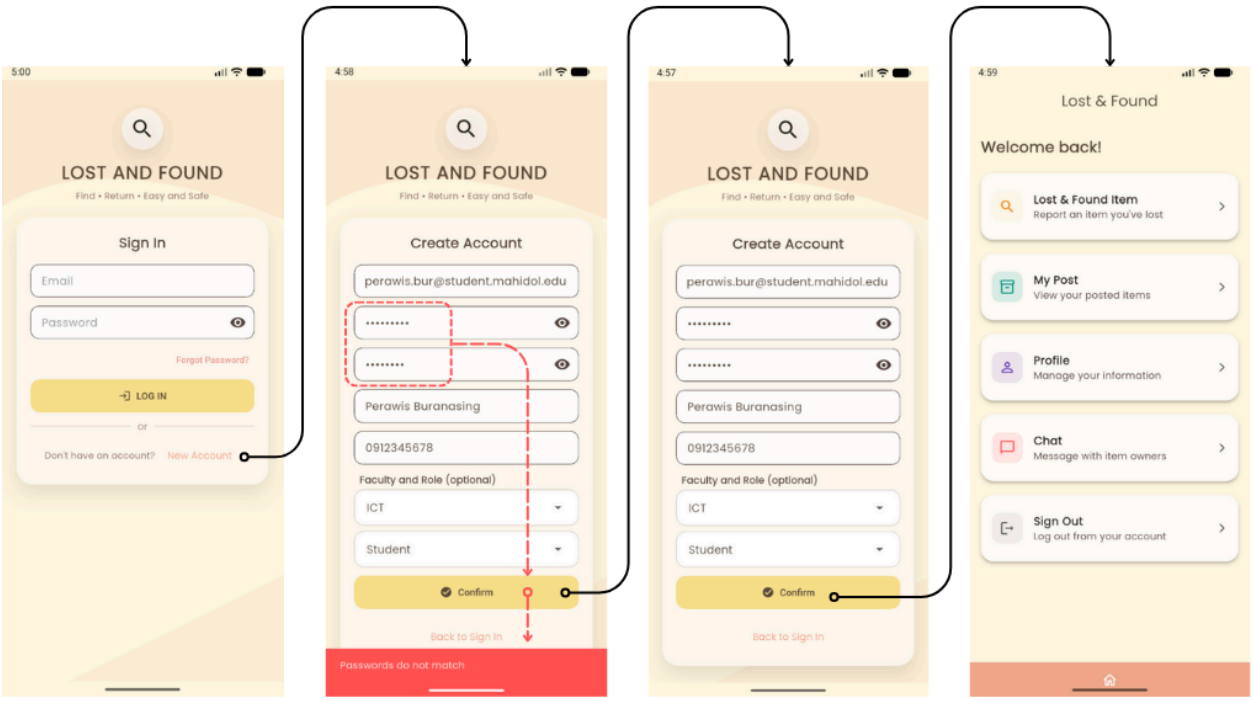
3) Correct Credentials

In the next scenario, the user enters a valid password, confirmed by a green checkmark beside the field. When pressing **Log In**, the authentication request succeeds, allowing the user to proceed.

4) Successful Login

Upon successful authentication, the system navigates the user to the **Welcome Screen**, where they can access key features such as Lost & Found Items, My Post, Profile, Chat, and Sign Out. This confirms that the login process is complete and the user has entered the main application.

- **Register screen:** This flow explains how a new user creates an account within the Lost & Found application, including validation steps and error handling.



1) Navigating to the Create Account Screen

From the Sign-In page, the user taps **"New Account"** to begin the registration process. They are redirected to the Create Account screen, which requires inputs such as email, password, confirm password, full name, phone number, and optional fields related to faculty and role.

2) Password Mismatch Error

In the first attempt, the user enters two passwords that do not match. When pressing **Confirm**, the system validates the input and detects the mismatch. A red error message appears at the bottom stating: *"Passwords do not match."* This prevents account creation until the user corrects the values.

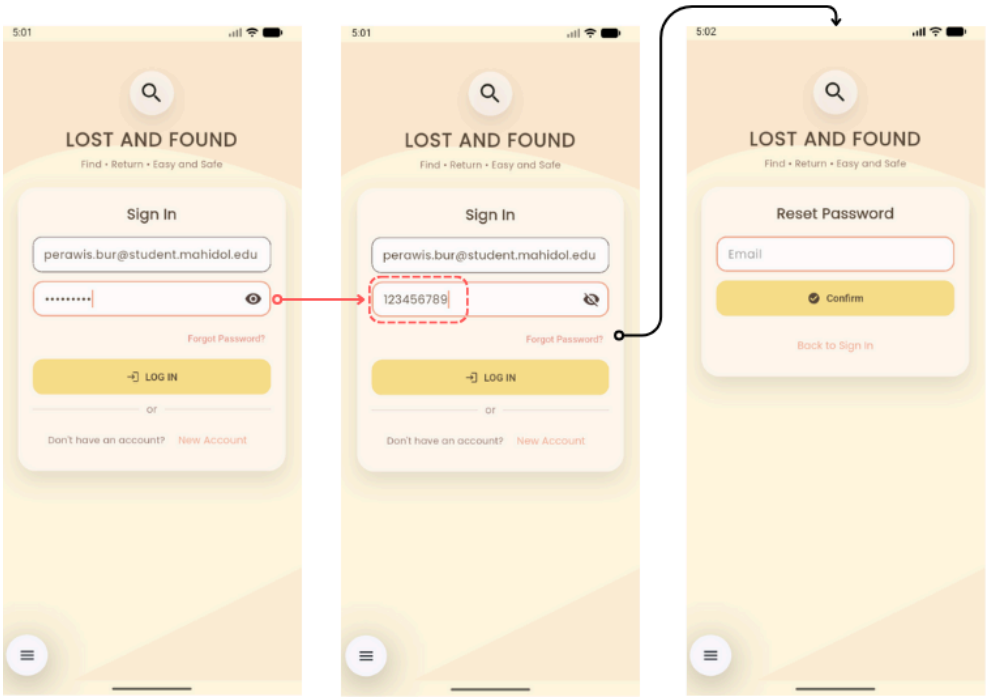
3) Correcting the Password and Reconfirming

The user updates the password fields to ensure both entries match. After filling in all required information accurately and tapping **Confirm**, the system processes the registration request successfully.

4) Successful Account Creation

Once the account is created, the user is automatically navigated to the **Welcome Screen**. Here, they gain access to the main app features, including Lost & Found Item, My Post, Profile, Chat, and Sign Out. This confirms that the registration flow is complete and the user is now logged into their new account.

- **Forgot Password & Password Reset Flow:** This flow demonstrates how a user initiates a password reset when they cannot remember their login password.



1) Starting from the Sign-In Screen

The user begins on the Sign-In page. After attempting to enter a password but realizing it is incorrect or forgotten, the user selects the **"Forgot Password?"** link. This indicates they need assistance recovering account access.

2) Navigating to the Reset Password Screen

Upon tapping the "Forgot Password?" link, the application redirects the user to the **Reset Password** page. This screen requires only the user's registered email address to initiate the reset process.

3) Submitting the Reset Request

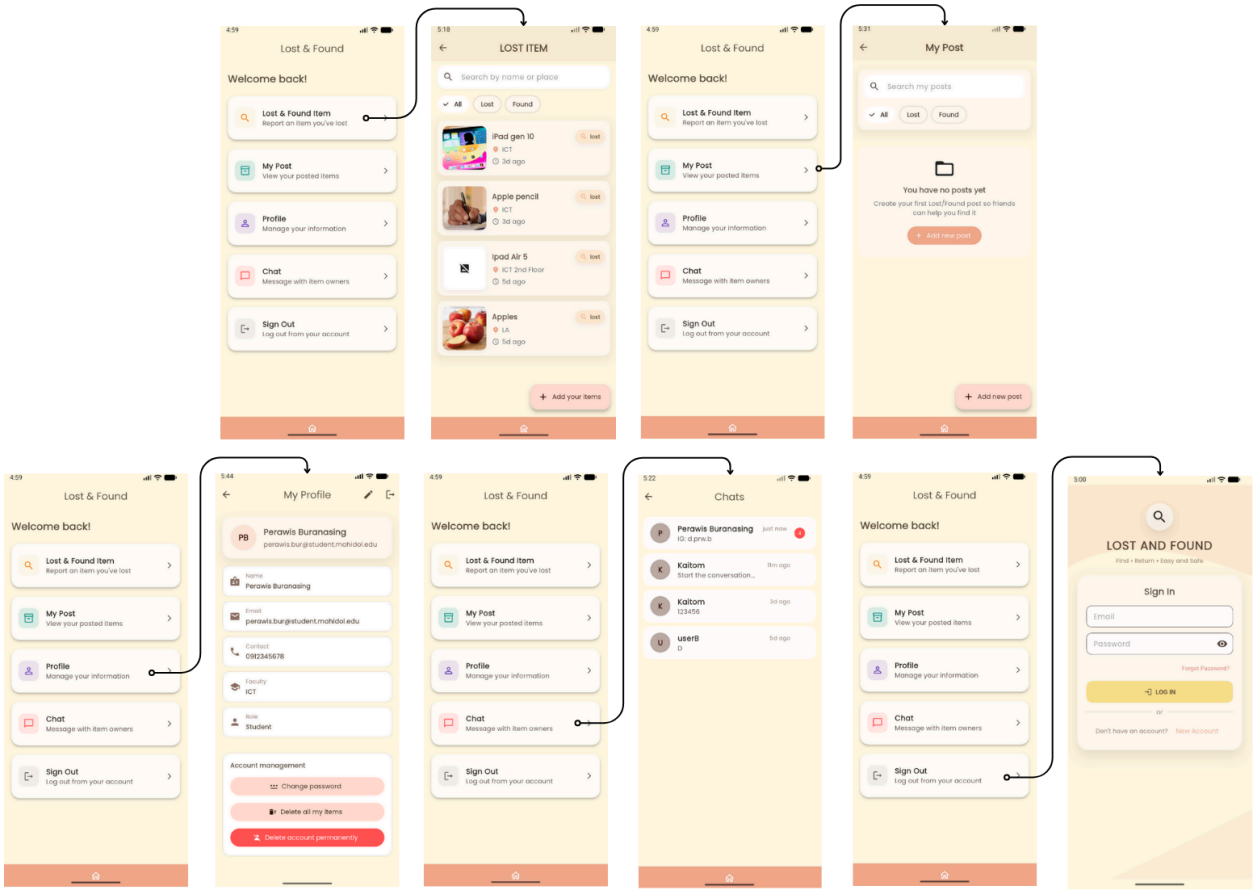
The user enters their email and presses **Confirm**. The system verifies whether the email corresponds to an existing account. Once validated, the backend triggers a password reset process (such as sending a reset link or temporary password to the user's email).

4) Completing the Flow

After submitting the email, the user is guided to check their inbox to complete the password reset steps. The **Back to Sign In** option allows them to return to the login page once the reset is done. This flow ensures that users can regain access securely and efficiently.

3.3.2 Home & Navigation

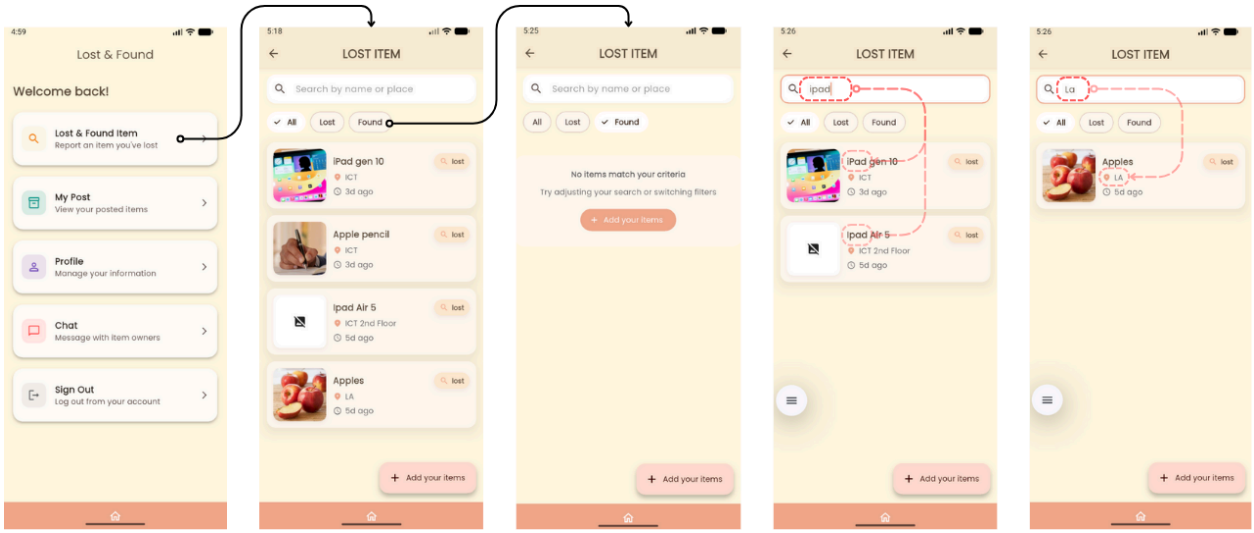
- After login, users see a **Home Menu** with cards such as:
  - “Lost & Found item”: Browse and search posts.
  - “My Post”: Manage items they created.
  - “Profile”: Edit personal information and manage accounts.
  - “Chat”: Open the chat list.
  - “Sign Out”.



A persistent bottom navigation bar (bottom_home_bar.dart) is used on main pages (Lost List, My Post, etc.) to switch quickly between sections.

3.3.3 Item Management Workflow

- **Lost Item Search & Filtering Flow** (lost_list_page.dart) This flow demonstrates how users browse, filter, and search for lost items within the Lost & Found application.



1) Accessing the Lost Item List

From the **Welcome Screen**, the user selects **"Lost & Found Item"**. This opens the Lost Item page, displaying all available items with details such as name, location, and posted date. Users can switch between **All**, **Lost**, and **Found** tabs to view different categories.

2) Filtering by Item Status

When the user taps the **Found** filter, the list updates accordingly. If no items match the selected filter, the system displays an empty-state message with an option to add new items. This helps users clearly understand when no relevant items are available.

3) Searching by Name or Location

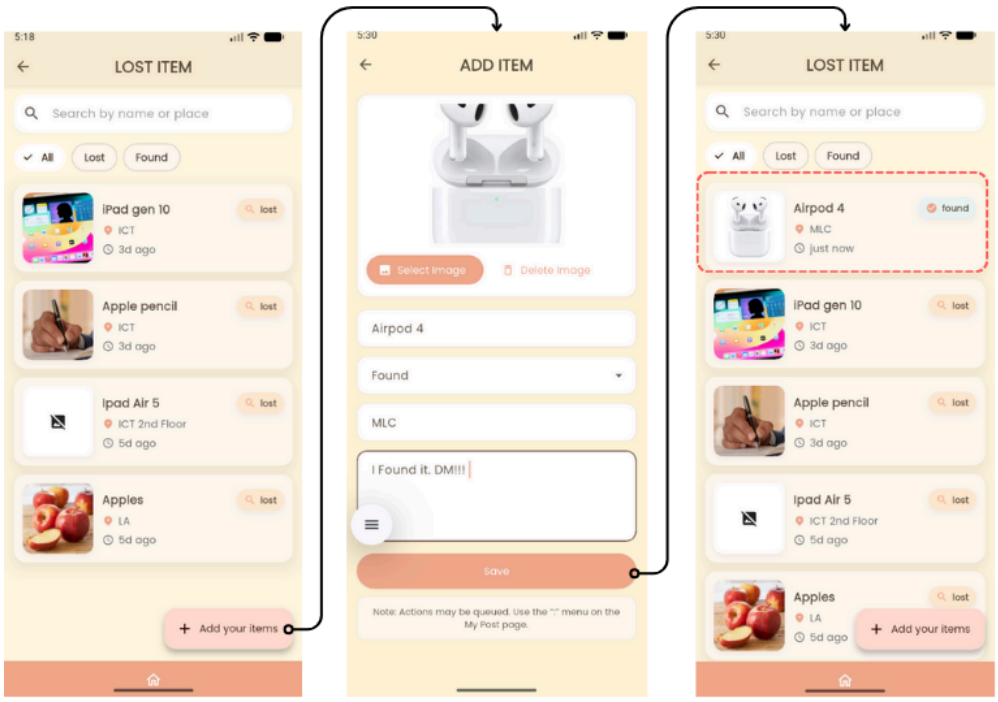
The user can also perform keyword searches using the search bar.

- When typing **"ipad"**, the system instantly filters and shows matching items such as "iPad Gen 10" and "iPad Air 5."
- When the user changes the search to **"La"**, the results update again, showing items associated with the LA location, such as "Apples."

4) Dynamic Result Updates

The search and filter mechanisms work together in real-time. The displayed items continuously adjust based on the user's entered text and selected category, ensuring a quick and intuitive search experience.

- **Add New Lost/Found Item Flow** (add_item_page.dart) This flow explains how a user adds a new lost or found item into the Lost & Found system.



1) Navigating to the Add Item Screen

From the **Lost Item** page, the user taps the **"Add your items"** button. This opens the Add Item screen, where the user can upload images and fill in item details.

2) Filling Item Information

On the Add Item page, the user selects or deletes an image, enters the item name (e.g., Airpod 4), chooses the item status (Lost or Found), specifies a location, and provides an optional description. These fields help categorize and display the item correctly within the system.

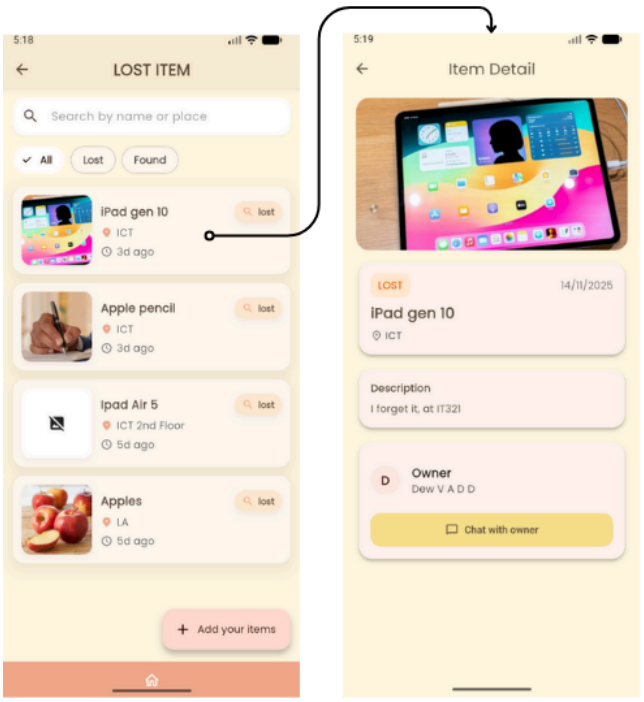
3) Saving the New Item

After completing the required information, the user taps **Save**. The system processes the submission and adds the item to the database. A note reminds users that some actions may be queued and viewable on the My Post page.

4) Item Successfully Added

Upon returning to the **Lost Item** list, the newly added item appears immediately, labeled with the selected status (e.g., Found) and showing its posting time as "just now." This confirms that the item is successfully uploaded and visible for other users to search, view, and contact if needed.

- **Item Detail Viewing Flow** (item_detail_page.dart) This flow describes how users view detailed information about a specific lost or found item in the Lost & Found application.



1) Selecting an Item from the List

From the Lost Item page, the user scrolls through the list of available posts. When the user taps on an item such as “iPad gen 10”, the system navigates to the Item Detail screen.

2) Displaying Item Information

The Item Detail screen shows complete information about the selected item. This includes a photo of the item, its name, status such as Lost, posting date, location category and a description written by the user who posted it. This allows the viewer to fully understand what the item is and where it was lost or found.

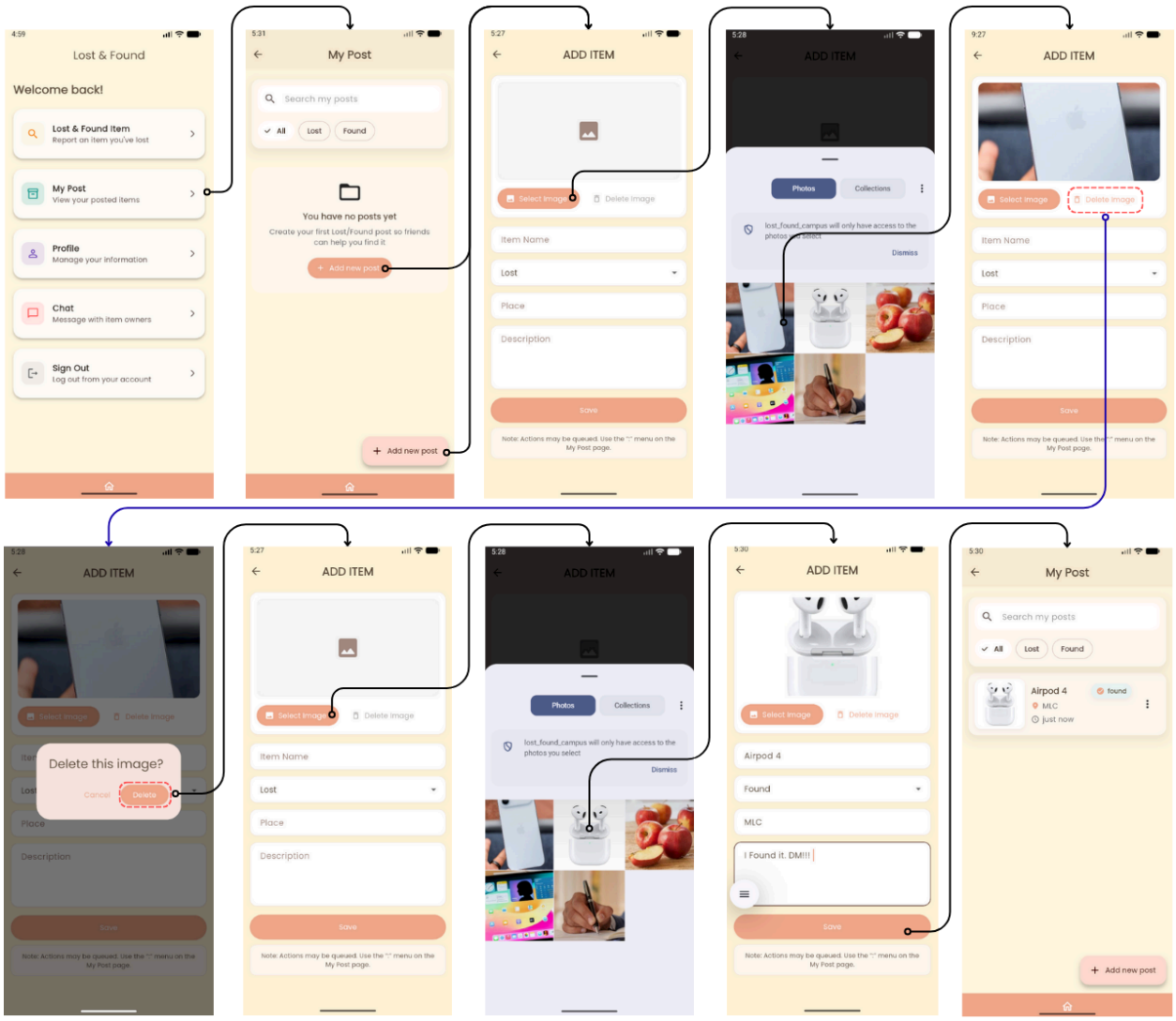
3) Viewing Owner Information

Below the description, the page displays the owner section. It includes the owner’s display name so that users know who posted the item and who they will be contacting.

4) Chat with Owner

At the bottom of the screen, there is a Chat with owner button. Selecting this button opens a chat where users can communicate with the owner, ask questions or arrange a return. This ensures smooth contact between users and supports the goal of helping lost items get returned.

- **My Post Screen, Add and Manage Item Flow** (my_post_page.dart, add_item_page.dart) This flow explains how users create and manage their own lost or found item posts within the My Post section of the application.



1) Navigating to My Post

From the Welcome screen, the user selects My Post. If this is their first time, the page displays an empty state with a prompt to create a new post. The user taps Add new post to begin posting an item.

2) Opening the Add Item Page

The system navigates to the Add Item page where the user can upload an image, enter the item name, choose Lost or Found, specify the location and write a description. These fields allow users to give clear details about the item they want to report.

3) Selecting an Image

When the user taps Select Image, the device's gallery opens. The user chooses an image and is returned to the Add Item page with the selected image displayed at the top.

4) Removing an Image

If the user wants to replace the image, they can tap Delete Image. A confirmation dialog appears asking whether they want to delete the current image. After confirming, the image is removed and the preview area becomes empty again.

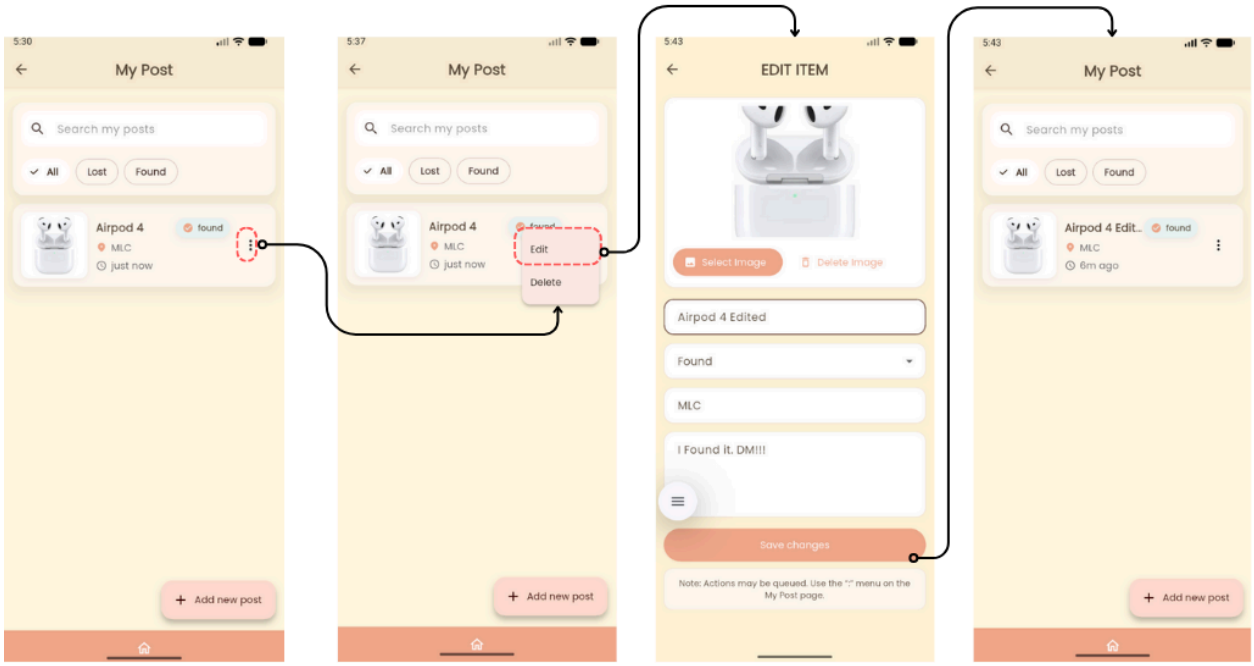
5) Re-selecting an Image and Completing the Form

The user selects a new image if needed, fills in all required fields and reviews the information. Once satisfied, the user taps Save to submit the post.

6) Viewing the Newly Added Post

After saving, the system returns the user to the My Post page. The new item appears immediately in the list with its status and posting time. The user can now manage or view details of their own posts.

- **Edit Posted Item Flow:** This flow shows how users edit an item they previously posted in the My Post section of the Lost & Found application.



1) Opening the Edit Options

From the My Post page, the user views their list of posted items. Each item has a small menu icon on the right side. When the user taps this icon, a menu appears with two options: Edit and Delete. The user selects Edit to update the item.

2) Navigating to the Edit Item Page

After choosing Edit, the system opens the Edit Item page. This screen displays the current image, item name, status, location and description. The user can modify any of these fields. They can also change the image or delete it if needed.

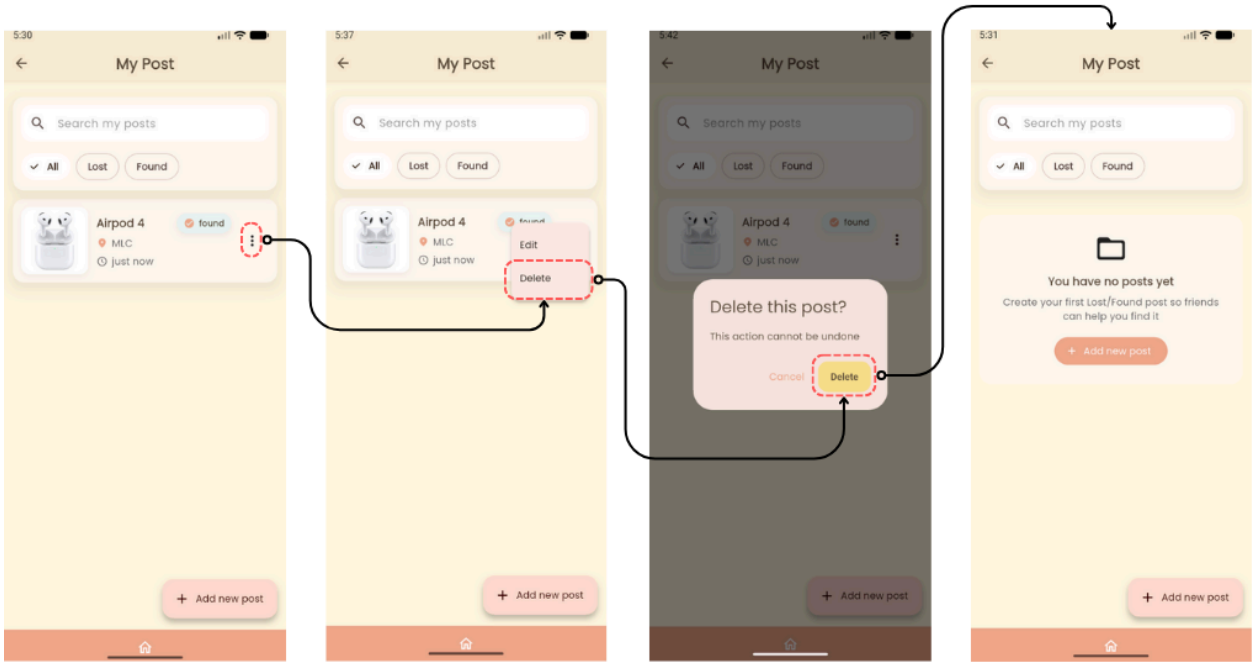
3) Updating Item Information

The user edits the item details, such as renaming the item to “Airpod 4 Edited” or adjusting the description. Once all changes are completed, the user taps Save changes. The system processes the update and applies the new information to the item.

4) Viewing the Updated Item

After saving, the user is taken back to the My Post page. The updated item appears in the list with the new details and a timestamp showing when it was last modified. This confirms that the item has been successfully updated and is now visible to other users in its revised form.

- **Delete Item screen:** This flow demonstrates how users remove an item they previously posted in the My Post section of the Lost & Found application.



1) Opening the Delete Menu

On the My Post page, the user sees a list of items they have created. Each item has a small menu icon on the right side. The user taps this icon to open the action menu. The menu presents two options: Edit and Delete. The user selects Delete to remove the item.

2) Confirming the Deletion

After choosing Delete, a confirmation dialog appears. This dialog informs the user that the deletion action cannot be undone. The user is prompted to either cancel the action or proceed. By tapping Delete in the dialog, the user confirms that they want to remove the item.

3) Removing the Item from the List

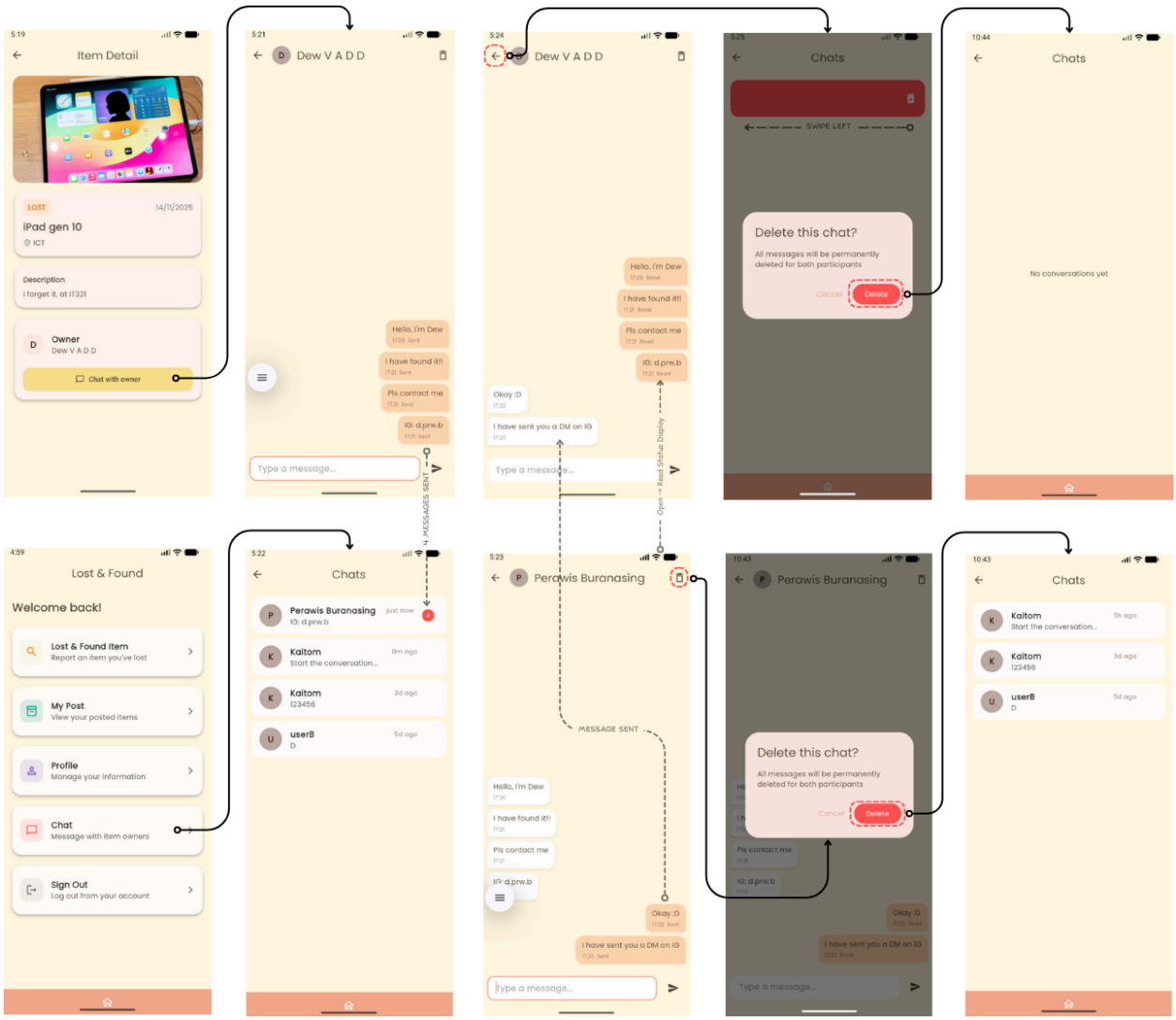
Once confirmed, the system deletes the selected item from the database. The user is returned to the My Post page, where the item no longer appears in the list.

4) Viewing the Empty State

If the deleted item was the only post, the page displays an empty state message indicating that no posts are available. A button labeled Add new post encourages the user to create a new item if needed.

3.3.4 Chat Workflow

- **Chat Interaction and Chat Management Flow** (chat_list_page.dart, chat_page.dart): This flow explains how users communicate with item owners through the chat system and how they manage or delete chat conversations.



1) Starting a Chat from an Item

From the Item Detail screen, the user can contact the owner by selecting the Chat with owner button. This opens a private chat room between the item owner and the viewer. The chat screen displays previous messages, if any, and includes a message input field at the bottom.

2) Sending Messages

The user types a message into the input field and sends it. The message appears immediately in the conversation. The system also updates the Chats list so users can quickly return to ongoing conversations from the main menu.

3) Accessing Chats from the Home Menu

From the Welcome screen, the user can select Chat to view all active conversations. The list displays each chat along with the latest message and timestamp. Tapping on a chat opens the conversation again, allowing users to continue messaging.

4) Deleting a Chat by Swiping

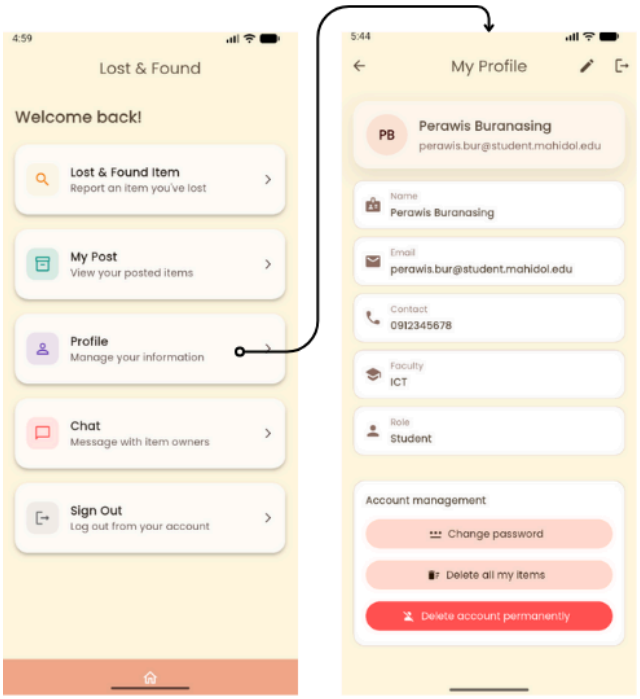
To remove a conversation, the user swipes left on any chat in the Chats list. A Delete option appears. Selecting it opens a confirmation dialog that warns the user that the chat will be permanently removed for both participants.

5) Confirming the Deletion

When the user confirms the deletion, the system removes the entire chat history. The Chats page refreshes to show the remaining conversations. If all chats are deleted, an empty state message appears indicating that there are no active conversations.

3.3.5 Profile & Account Management

- **Profile Viewing and Account Management Flow** (profile_page.dart): This flow explains how users view and manage their personal information within the My Profile section of the Lost & Found application.



1) Navigating to the Profile Page

From the Welcome screen, the user selects the Profile option. The system opens the My Profile page, where all stored account information is displayed in an organized layout.

2) Viewing Personal Information

The Profile page shows the user's name, email, contact number, faculty and role. This allows users to review the information they provided during registration. Each field is clearly presented so users can easily verify their details.

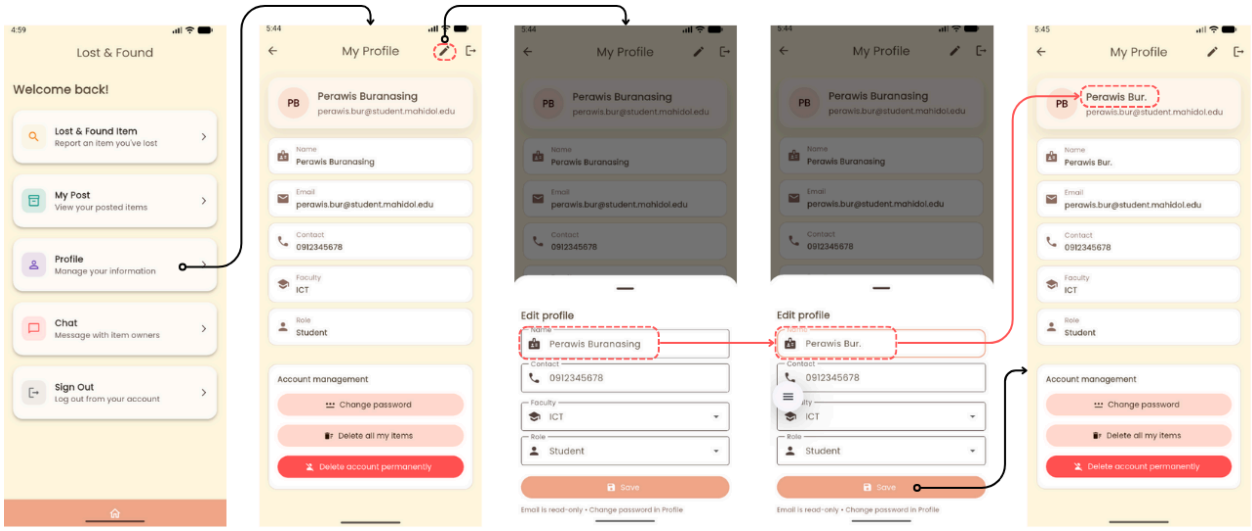
3) Accessing Account Management Tools

Below the personal information, the Account management section provides additional controls. These include a Change password button, a Delete all my items button and a Delete account permanently button. Each function offers users the ability to manage different aspects of their account.

4) Managing or Updating the Account

Selecting any of the options leads to the related action. For example, users can update their password for security purposes, remove all items they posted or permanently delete their entire account. This section ensures the user has full control over their profile and data within the system.

- **Edit Profile Flow** (profile_page.dart): This flow describes how users update their personal information within the My Profile section of the Lost & Found application.



1) Accessing the Profile Page

From the Welcome screen, the user selects the Profile option. The system opens the My Profile page, displaying the user's information such as name, email, contact number, faculty and role.

2) Entering Edit Mode

To modify their information, the user taps the edit icon at the top right of the Profile page. A bottom sheet appears containing editable fields including name, contact, faculty and role. The email field is displayed as read-only to protect account identity.

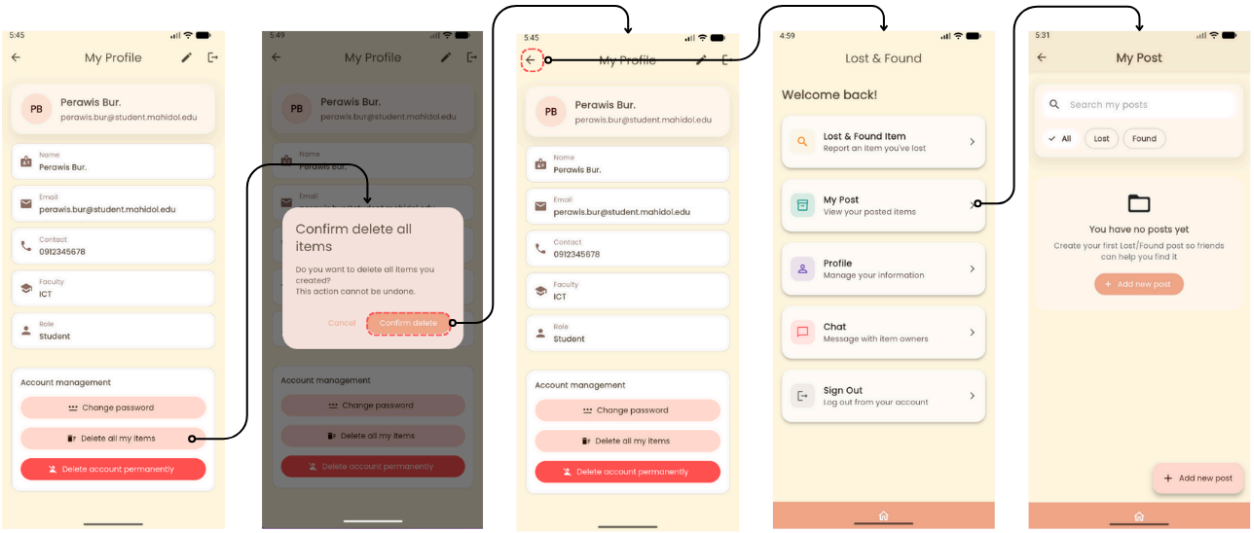
3) Editing Profile Information

The user makes changes to the fields, such as updating their name or adjusting their contact details. Once they finish editing, they tap the Save button at the bottom of the screen. The system processes the update and closes the edit panel.

4) Viewing Updated Information

After saving, the user returns to the My Profile page. The updated information appears immediately, confirming that the changes were successfully applied. The user can continue managing their account or return to the home screen.

- **Delete All Items Flow** (profile_page.dart): This flow describes how users delete all items they have created from within the My Profile section of the Lost & Found application.



1) Opening the Delete All Items Option

From the My Profile page, the user scrolls to the Account management section and selects the Delete all my items button. This action is designed for users who want to remove every item they have created.

2) Confirming the Bulk Deletion

A confirmation dialog appears, asking whether the user wants to delete all their items. The dialog clearly warns that this action cannot be undone. The user can choose to cancel or proceed. By selecting Confirm delete, the user approves the removal of all their posts.

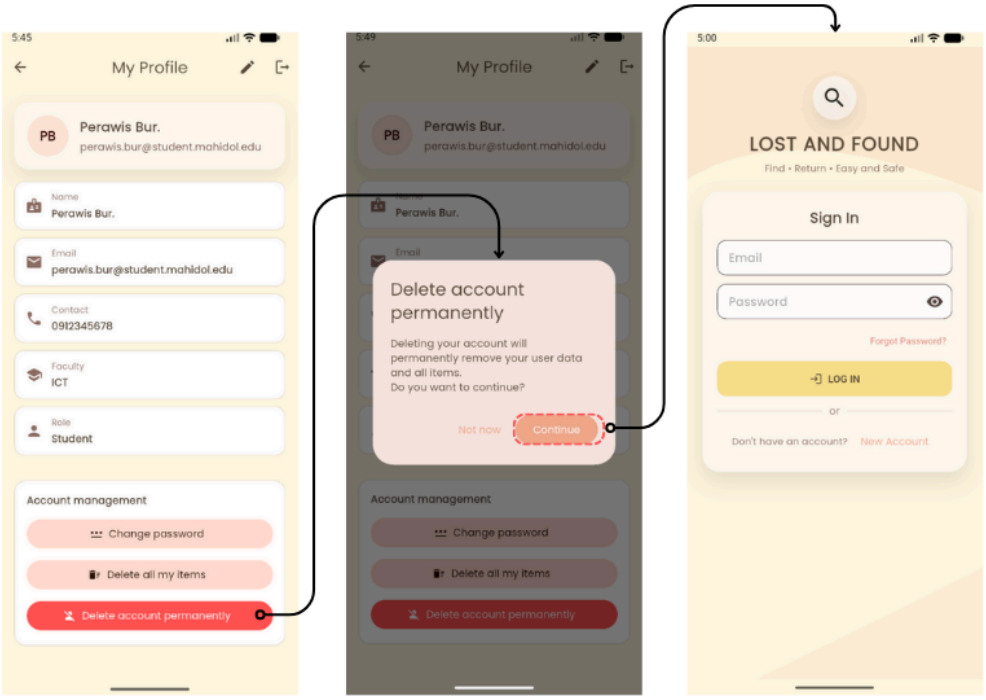
3) Completing the Deletion Process

After confirmation, the system deletes all items associated with the user's account. The user returns to the Profile page, where the Delete all my items option remains available for future use if needed.

4) Viewing the Result in My Post

To verify the deletion, the user navigates back to the Welcome screen and selects My Post. The My Post page now shows an empty state, indicating that all previously created posts have been removed. The page also provides an Add new post button for creating new entries.

- **Delete Account Permanently Flow** (profile_page.dart): This flow describes how users permanently remove their account and all associated data from the Lost & Found application.



1) Selecting the Delete Account Option

From the My Profile page, the user scrolls to the Account management section and chooses the Delete account permanently button. This option is intended for users who want to completely remove their presence from the system.

2) Confirming Permanent Deletion

A confirmation dialog appears, explaining that deleting the account will permanently erase all user data and all items created by the user. The dialog provides two choices: Not now or Continue. The user selects Continue to finalize the decision.

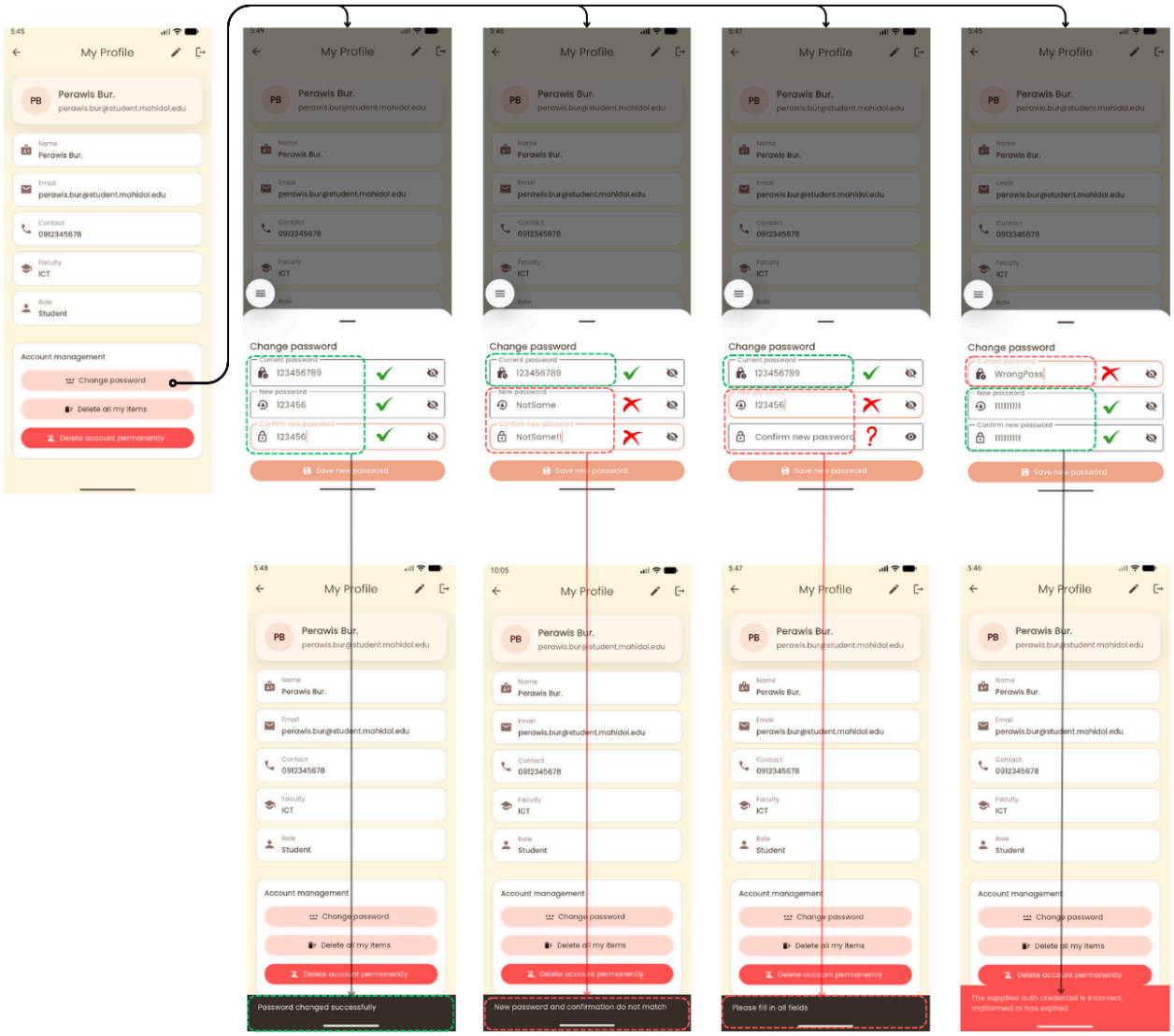
3) Completing the Deletion Process

Once confirmed, the system deletes the user's account along with all related data. After the deletion is processed, the user is automatically redirected to the Sign In screen.

4) Returning to the Sign In Page

The Sign In screen appears, confirming that the account no longer exists in the system. If the user wishes to access the application again, they must create a new account or log in with a different one.

- **Change Password Flow** (profile_page.dart): This flow explains how users update their account password and how the system handles incorrect or incomplete inputs.



1) Opening the Change Password Page

From the My Profile page, the user selects the Change password option located in the Account management section. The system opens a form that requires the current password, a new password and confirmation of the new password.

2) Entering Valid Passwords

When the user enters all three fields correctly, the interface shows green check marks indicating valid input. After selecting Save new password, the system updates the password and returns the user to the Profile page with a success message stating that the password has been changed successfully.

3) Handling Mismatched New Passwords

If the user enters a confirmed password that does not match the new password, the system highlights the mismatch. When attempting to save, the user receives an error message indicating that the new password and confirmation do not match. The user must correct the inputs to continue.

4) Handling Incorrect Current Password

If the current password entered by the user is wrong, the system detects it during the save attempt. The user is shown an error at the bottom of the screen stating that the supplied credentials are incorrect or malformed. The user must re-enter the correct current password before saving.

5) Handling Missing Fields

If the user leaves any field empty, the Save new password button results in an error. The system displays a message asking the user to fill in all fields before continuing. This ensures that the password cannot be updated with incomplete information.

After correcting errors and entering all required values properly, the user can successfully update their password and return to the Profile page.

3.4 Implementation Details

Below is a high level overview of key implementation aspects:

- **Authentication**
  - Email/password sign-up and login use FirebaseAuth.
  - Error codes from Firebase are mapped to user-friendly messages shown as banners at the bottom of the screen.
  - Password reset uses sendPasswordResetEmail with a simple form.

- **Firestore Data Model**
  - users collection: Stores profile information and metadata for each user.
  - items collection: Documents representing lost/found posts, including owner ID, image URL, status, location, and timestamps.
  - chats / messages collections: Store chat rooms and individual messages with sender ID, content, and timestamp.

- **Image Upload**
  - When creating or editing an item, the selected image is uploaded to Firebase Storage.
  - After upload, the download URL is stored in the corresponding item document.
  - Only **single-image upload** is currently supported; multi-image upload was planned but not completed.

- **Real-time Updates**
  - Item lists and chat messages listen to Firestore snapshots, so changes are reflected immediately without manual refresh.

- **Theming & Reusable Widgets**
  - Custom widgets (AppButton, AppInput, ItemTile) encapsulate styling and reduce duplicate code.
  - theme.dart defines the global color scheme and text styles, which are applied through ThemeData in main.dart.

3.5 Code Snippet and Explanation

3.5.1 Firebase Authentication: Sign In

```
Future<void> _signIn() async {
  if (loading) return;
  FocusScope.of(context).unfocus();

  final mail = email.text.trim();
  final pwd  = pass.text.trim();
  if (mail.isEmpty || pwd.isEmpty) {
    _showError('Please enter both email and password');
    return;
  }

  setState(() => loading = true);
  try {
    await FirebaseAuth.instance.signInWithEmailAndPassword(
      email: mail,
      password: pwd,
    );
    await _maybeReturnToAuthGate();
  } on FirebaseAuthException catch (e) {
    _showError(e.message ?? 'Sign in failed');
  } finally {
    if (mounted) setState(() => loading = false);
  }
}
```

Explanation:
This function validates the email and password fields, then uses **"`FirebaseAuth.signInWithEmailAndPassword`"** to authenticate the user. Errors from Firebase are mapped to a snackbar message, and a loading flag is used to disable the UI while the request is in progress.

3.5.2 Firebase Authentication: Password Reset

```
Future<void> _submitReset() async {
  if (loading) return;
  final mail = email.text.trim();
  if (mail.isEmpty) {
    _showError('Please enter your email first');
    return;
  }
  setState(() => loading = true);
  try {
    await FirebaseAuth.instance.sendPasswordResetEmail(email: mail);
    _showInfo('Password reset link sent to $mail');
    _switchMode(_AuthMode.signIn);
  } on FirebaseAuthException catch (e) {
    _showError(e.message ?? 'Failed to send password reset email');
  } finally {
    if (mounted) setState(() => loading = false);
  }
}
```

Explanation:
The reset flow sends a password reset email using **"sendPasswordResetEmail"**. When the request succeeds, the app displays a confirmation message and switches back to the sign in mode.

3.5.3 Firebase Authentication: Sign Up and Create User Profile

```
Future<void> _submitSignUp() async {
  if (loading) return;
  FocusScope.of(context).unfocus();

  final mail = email.text.trim();
  final pwd  = pass.text.trim();
  final cPwd = confirm.text.trim();

  if (mail.isEmpty || pwd.isEmpty || cPwd.isEmpty) {
    _showError('Please fill in Email, Password, and Confirm Password');
    return;
  }
  if (pwd != cPwd) {
    _showError('Passwords do not match');
    return;
  }

  setState(() => loading = true);
  try {
    final cred = await FirebaseAuth.instance.createUserWithEmailAndPassword(
      email: mail,
      password: pwd,
    );
```

```
    final uid = cred.user!.uid;
    final data = {
      'uid'      : uid,
      'email'    : mail,
      'name'     : displayName.text.trim().isEmpty
                    ? 'Unknown'
                    : displayName.text.trim(),
      'contact'  : contact.text.trim().isEmpty
                    ? 'None'
                    : contact.text.trim(),
      'faculty'  : faculty,
      'role'     : role,
      'createdAt': FieldValue.serverTimestamp(),
    };

    await FirebaseFirestore.instance
        .collection('users')
        .doc(uid)
        .set(data);

    await _maybeReturnToAuthGate();
  } on FirebaseAuthException catch (e) {
    _showError(e.message ?? 'Sign up failed');
  } catch (e) {
    _showError('Failed to save user data: $e');
  } finally {
    if (mounted) setState(() => loading = false);
  }
}
```

Explanation:
The sign up flow first validates and confirms the password, then creates a new account using Firebase Authentication. After registration, it initializes a corresponding **"users/{uid}"** document in Firestore with profile data such as name, contact, faculty, and role.

3.5.4 Add or Edit Lost Item: Firestore CRUD

```
Future<void> _save() async {
  if (!_formKey.currentState!.validate()) return;

  setState(() => _isSaving = true);
  try {
    final uid = FirebaseAuth.instance.currentUser?.uid;
    if (uid == null) throw Exception('Please sign in again.');

    final data = <String, dynamic>{
      'title'   : _titleCtrl.text.trim(),
      'status'  : _status,
      'place'   : _placeCtrl.text.trim(),
      'desc'    : _descCtrl.text.trim(),
      'imageUrl': _imageUrl ?? '',
      if (_imagePath != null) 'imagePath': _imagePath,
    };

    final col = FirebaseFirestore.instance.collection('items');

    if (_isEdit) {
      await col.doc(widget.docId).update(data);
      ScaffoldMessenger.of(context)
          .showSnackBar(const SnackBar(content: Text('Changes saved.')));
    } else {
      data['ownerUid']  = uid;
      data['createdAt'] = FieldValue.serverTimestamp();
      await col.add(data);
      ScaffoldMessenger.of(context)
          .showSnackBar(const SnackBar(content: Text('Post created.')));
    }

    Navigator.pop(context);
  } catch (e) {
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text('An error occurred. $e')));
  } finally {
    if (mounted) setState(() => _isSaving = false);
  }
}
```

Explanation:
This function is shared between "add item" and "edit item" modes. It collects form data, then either creates a new document or updates an existing document in the **"items"** collection. For new items it sets **"ownerUid"** and **"createdAt"** for later filtering.

3.5.5 Image Upload to Firebase Storage

```
Future<void> _pickImage() async {
  final picker = ImagePicker();
  final XFile? picked =
      await picker.pickImage(source: ImageSource.gallery, imageQuality: 88);
  if (picked == null) return;

  setState(() => _isUploadingImage = true);

  try {
    final uid = FirebaseAuth.instance.currentUser?.uid;
    if (uid == null) throw Exception('User not found.');

    final fileId = widget.docId??DateTime.now().millisecondsSinceEpoch.toString();
    final path   = 'items/$uid/$fileId.jpg';

    final ref = FirebaseStorage.instance.ref().child(path);
    await ref.putFile(File(picked.path));

    final url = await ref.getDownloadURL();

    if (_imagePath != null && _imagePath != path) {
      try {
        await FirebaseStorage.instance.ref().child(_imagePath!).delete();
      } catch (_) {}
    }

    setState(() {
      _imageUrl  = url;
      _imagePath = path;
    });
  } catch (e) {
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text('Image upload failed: $e')));
  } finally {
    if (mounted) setState(() => _isUploadingImage = false);
  }
}
```

Explanation:
This method allows the user to pick a photo from the gallery, upload it to Firebase Storage under a user specific path, and then store the download URL and storage path in state. When replacing an image, the previous file is deleted to avoid unused storage.

3.5.6 Lost Item List Page: Search and Filter Query

```
@override
Widget build(BuildContext context) {
  final query = FirebaseFirestore.instance
      .collection('items')
      .orderBy('createdAt', descending: true);

  return StreamBuilder<QuerySnapshot<Map<String, dynamic>>>(
    stream: query.snapshots(),
    builder: (context, snap) {
      // ...

      final s = searchCtrl.text.trim().toLowerCase();
      final docs = snap.data!.docs.where((e) {
        final d      = e.data();
        final status = (d['status'] ?? '').toString().toLowerCase();
        final title  = (d['title'] ?? '').toString().toLowerCase();
        final place  = (d['place'] ?? '').toString().toLowerCase();

        final matchFilter = (filter == 'All') ||
            (filter == 'Lost'  && status == 'lost') ||
            (filter == 'Found' && status == 'found');

        final matchSearch =
            s.isEmpty || title.contains(s) || place.contains(s);

        return matchFilter && matchSearch;
      }).toList();

      // build ListView with docs...
    },
  );
}
```

Explanation:
The lost item screen reads the **"items"** collection in real time using a Firestore stream. Additional filtering is done on the client to match the selected status filter and the search keyword typed by the user.

3.5.7 Chat Service: Send Message with Transaction

```
static Future<void> sendText({
  required String chatId,
  required String text,
  required String senderUid,
}) async {
  final chatRef = _db.collection('chats').doc(chatId);
  final msgRef  = chatRef.collection('messages').doc();

  await _db.runTransaction((tx) async {
    final snap = await tx.get(chatRef);
    final data = snap.data()!;
    final List parts = List.from(data['participants']);
    final other = parts.firstWhere((u) => u != senderUid);

    tx.set(msgRef, {
      'senderId' : senderUid,
      'text'     : text.trim(),
      'createdAt': FieldValue.serverTimestamp(),
      'readBy'   : [senderUid],
    });

    final Map<String, dynamic> unread =
        Map<String, dynamic>.from(data['unread'] ?? {});
    unread[other] = (unread[other] ?? 0) + 1;

    tx.update(chatRef, {
      'lastMessage': text.trim(),
      'lastAt'     : FieldValue.serverTimestamp(),
      'unread'     : unread,
    });
  });
}
```

Explanation:
Messages are stored under **"chats/{chatId}/messages"**. A Firestore transaction writes the new message and updates the parent chat document at the same time, including the last message preview and an unread counter for the other participant.

3.5.8 Chat Page: Real-time Messages and Send Function

```
Future<void> _send() async {
  final txt = _ctrl.text.trim();
  if (txt.isEmpty) return;
  await ChatService.sendText(
    chatId   : widget.chatId,
    text     : txt,
    senderUid: _uid,
  );
  _ctrl.clear();
```

```
}

@override
Widget build(BuildContext context) {
  final msgsQ = FirebaseFirestore.instance
      .collection('chats').doc(widget.chatId)
      .collection('messages')
      .orderBy('createdAt', descending: true);

  return Scaffold(
    // ...
    body: Column(
      children: [
        Expanded(
          child: StreamBuilder<QuerySnapshot>(
            stream: msgsQ.snapshots(),
            builder: (context, snap) {
              if (!snap.hasData) {
                return const Center(child: CircularProgressIndicator());
              }
              final docs = snap.data!.docs;

              return ListView.builder(
                reverse: true,
                padding: const EdgeInsets.all(12),
                itemCount: docs.length,
                itemBuilder: (_, i) {
                  final m     = docs[i].data() as Map<String, dynamic>;
                  final isMe  = m['senderId'] == _uid;
                  final readBy = List.from(m['readBy'] ?? []);
                  final createdAt =
                      (m['createdAt'] as Timestamp?)?.toDate();

                  // build chat bubble...
                },
              );
            },
          ),
        ),
        // input box and send button...
      ],
    ),
  );
}
```

Explanation:

The chat screen listens to the **"messages"** subcollection in real time and displays messages in a reversed **"ListView"**. When the user presses send, the text is passed to **"ChatService.sendText"**, then the text field is cleared.

3.5.9 Profile Page: Change Password with Re-authentication

```
Future<void> _handleChangePassword({required BuildContext sheetContext}) async {
  if (user == null) return;
  Navigator.of(sheetContext).pop();   // close bottom sheet
  final mail    = user!.email ?? '';
  final oldPwd  = _oldPwdCtrl.text;
  final newPwd1 = _newPwdCtrl.text;
  final newPwd2 = _newPwd2Ctrl.text;

  if (oldPwd.isEmpty || newPwd1.isEmpty || newPwd2.isEmpty) {
    ScaffoldMessenger.of(context)
        .showSnackBar(const SnackBar(content: Text('Please fill in all fields')));
    return; }
  if (newPwd1 != newPwd2) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('New password and confirmation do not match')),
    );
    return; }
  if (newPwd1.length < 6) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('New password must be at least 6 characters')),
    );
    return; }

  setState(() => working = true);
  try {
    final cred = EmailAuthProvider.credential(email: mail, password: oldPwd);
    await user!.reauthenticateWithCredential(cred);
    await user!.updatePassword(newPwd1);
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Password changed successfully')),
    );
  } on FirebaseAuthException catch (e) {
    final msg = switch (e.code) {
      'wrong-password'        => 'Current password is incorrect',
      'weak-password'         => 'New password is too weak',
      'requires-recent-login' => 'Please sign in again before changing password',
      _                       => e.message ?? 'Password change failed',
    };
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text(msg), backgroundColor: Colors.redAccent),
    );
  } finally {
    if (mounted) setState(() => working = false);
  }
}
```

Explanation:
To change the password securely, the app first validates all fields, then reauthenticates the user with the current password before calling **"updatePassword"**. Different Firebase error codes are converted to user friendly error messages.

3.5.10 Delete All Items and Delete Account

```
Future<void> _deleteAllMyItems() async {
  if (user == null) return;
  final uid   = user!.uid;
  final email = user!.email;

  setState(() => working = true);
  try {
    await _deleteByField('ownerUid', uid);
    await _deleteByField('uid', uid);
    await _deleteByField('ownerId', uid);

    if (email != null && email.isNotEmpty) {
      await _deleteByField('email', email);
      await _deleteByField('ownerEmail', email);
    }
  } finally {
    if (mounted) setState(() => working = false);
  }
}

Future<void> _deleteAccountCascade() async {
  if (user == null) return;
  final uid = user!.uid;

  setState(() => working = true);
  try {
    await _deleteAllMyItems();
    await fs.collection('users').doc(uid).delete();
    await user!.delete();

    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Account and all data deleted successfully')),
    );
    Navigator.of(context).popUntil((r) => r.isFirst);
  } on FirebaseAuthException catch (e) {
    final msg = (e.code == 'requires-recent-login')
        ? 'Please sign in again before deleting the account (security requirement)'
        : (e.message ?? 'Account deletion failed');
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text(msg), backgroundColor: Colors.redAccent),
    );
  } finally {
    if (mounted) setState(() => working = false);
  }
}
```

Explanation:
Before deleting the user account, the app removes all items created by that user and deletes the corresponding Firestore user document. The operation handles multiple possible field names for backward compatibility and also deletes attached images using storage paths.

3.5.11 Item Detail – Open Chat With Owner

```
ElevatedButton.icon(
  icon: const Icon(Icons.chat_bubble_outline_rounded),
  label: const Text('Chat with owner'),
  style: ElevatedButton.styleFrom(
    padding: const EdgeInsets.symmetric(vertical: 14),
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(14),
    ),
  ),
  onPressed: () async {
    final chatId = await ChatService.openOrCreateChat(
      uidA  : me,
      uidB  : ownerUid,
      itemId: itemId,
    );
    if (context.mounted) {
      Navigator.of(context).push(
        MaterialPageRoute(
          builder: (_) => ChatPage(
            chatId  : chatId,
            otherUid: ownerUid,
          ),
        ),
      );
    }
  },
)
```

Explanation:
From the item detail screen, the user can start a private chat with the owner. The app calls **"`openOrCreateChat`"** to either reuse an existing room or create a new one linked to the item, then navigates to **"`ChatPage`"**.

4. Results

4.1 Completed Features

According to the original project proposal, the MVP included the following core features: user login, registration, and password reset; creating and viewing posts; searching and filtering posts; updating post status; live chat with post owners (as an add-on feature); viewing, editing, and deleting user profiles; and viewing other users' profiles. As demonstrated in the workflow section, our application is also capable of handling additional scenarios such as input validation, error handling, and unexpected user or system behaviours.

In the final implementation, we successfully completed the following:

1. **User Authentication and Account Management**
   - ✔ Register with university-style email, password, and profile information.
   - ✔ Login, logout, and password reset via email.
   - ✔ Edit profile (name, contact, faculty, role).
   - ✔ Change password with validation on current and new passwords.
   - ✔ Delete all items created by the user.
   - ✔ Permanently delete account with confirmation.

2. **Lost & Found Posting**
   - ✔ Create "Lost" or "Found" posts with one image, location, and description.
   - ✔ View all posts in the Lost Item list with search and filter by Lost/Found.
   - ✔ View item details including owner information and description.
   - ✔ Edit or delete existing posts from the "My Post" page.

3. **Search and Filtering**
   - ✔ Text search by item name or place on Lost Item and My Post screens.
   - ✔ Filter switch between All / Lost / Found states.

4. **Chat System**
   - ✔ One-to-one chat between the finder and owner of an item.
   - ✔ Chat list displays all conversations with last message and time.
   - ✔ Chats can be deleted with confirmation.

5. **UI/UX**
   - ✔ Consistent design language across authentication, item, chat, and profile screens.
   - ✔ Clear error messages, confirmation dialogs, and empty states (e.g., "No items match your criteria").

4.2 Limitations

Some planned features from the proposal are **not implemented** in the current version:

- **Map pin / location on map**: The app currently stores location as plain text; there is no map view or geolocation pin.
- **Multi-image upload**: Each post supports only a single image, even though multi-image upload was originally planned as an add-on.
- **Push notifications**: Firebase Cloud Messaging integration for real-time notifications (e.g., new messages or updates) was not completed due to time constraints.

Other minor limitations:

- No dedicated admin or report-abuse system; moderation would require manual checking in Firestore.
- Testing was primarily manual; there are no automated unit tests or integration tests yet.

4.3 Testing Outcomes

We performed manual testing on both emulator and physical devices focusing on:

- Authentication scenarios:
    - Correct login, wrong email, wrong password, invalid email format, password mismatch in registration, and reset password flow.
- Item management:
    - Creating, editing, deleting posts; verifying that thumbnails and details match the database; checking that deleted posts disappear from the lists.
- Chat:
    - Sending messages between two test accounts; deletion of chat threads; checking that messages appear in real time.
- Profile and account management:
    - Editing profile fields, changing password with correct/incorrect current password, deleting all items, and deleting the whole account.

Overall, the core features worked as expected for normal usage. Errors were mainly related to network connectivity and Firebase permission rules during early development, which were fixed during debugging.

5. Conclusion

The **Lost & Found @Campus** application provides a practical solution to the everyday problem of missing items on campus. Users can create posts for lost or found items, search and filter existing posts, manage their own posts, and directly communicate with other users through the built-in chat feature. By building on Flutter and Firebase, we were able to implement authentication, real-time data, and image upload with relatively small code size and good developer productivity.

Although some planned add-on features such as map pins, multi-image upload, and push notifications are not yet implemented, the current version is already functional and usable as a minimum viable product (MVP). In the future, we plan to:

- Integrate **map pins** using geolocation and a map widget so users can see where items were found or lost.
- Add **multi-image upload** to allow multiple photos per item for better identification.
- Implement **push notifications** (Firebase Cloud Messaging) to notify users of new messages or updates to their posts.
- Improve moderation and safety with a reporting system and possibly an admin dashboard.

From a learning perspective, this project helped us practice the full development cycle: from proposal and UI design to implementation, testing, and documentation using advanced Flutter libraries.

6. Responsibilities of Each Team Member

All team members participated in discussions, design decisions, and testing. The main responsibilities were divided as follows:

1. **Perawis Buranasing (6688012)**
    - Project coordination and integration.
    - Implemented all application logic and backend-related functionality.
    - Developed focused on APIs part, data handling, and core system modules.
    - Managed necessary navigation links between screens.
    - Supported overall technical integration.
    - Prepared the written project report.

2. **Phana Mahachairachun (6688061)**
    - Responsible for UX/UI implementation across the entire project.
    - Designed and built all user interface screens and interaction flows.
    - Prepared the written project report.
    - Contributed to creating the presentation slides.

3. **Pathompong Prasitphol (6688088)**
    - Designed and implemented UX/UI components and layouts.
    - Created visual flow and improved user experience across screens.
    - Produced the project's video presentation.
    - Contributed to creating the presentation slides.

7. References

[1] **Google.** (2025). *Firebase Authentication documentation*. Firebase.google.com. https://firebase.google.com/docs/auth

[2] **Google.** (2025). *Cloud Firestore documentation*. Firebase.google.com. https://firebase.google.com/docs/firestore

[3] **Google.** (2025). *Firebase Storage documentation*. Firebase.google.com. https://firebase.google.com/docs/storage

[4] **Reso Coder.** (2020). *Flutter & Firebase tutorial series (Firestore CRUD)* [Video]. https://www.youtube.com/watch?v=8pG8CheUZOw

[5] **Flutter Community.** (2024). *Flutter state management, UI design, and Firebase guides*. Medium.com. https://medium.com/flutter

[6] **Fireship.** (2023). *Minimal chat app • Flutter x Firebase tutorial* [Video]. YouTube.https://www.youtube.com/watch?v=5xU5WH2kEc0

[7] **The Net Ninja.** (2020). *Getting started with Firebase* [YouTube Playlist]. YouTube. https://www.youtube.com/watch?v=-zHAVEoLkG0&list=PL4cUxeGkcC9jERUGvbudErNCeSZHWUVlb