FINAL PROJECT REPORT

SEMESTER 2, ACADEMIC YEAR: 2024-2025

CT312H: MOBILE PROGRAMMING

- **Project/Application name: Mini supermarket application**
- **GitHub link:** https://github.com/24-25Sem2-Courses/ct312hm01-project-ngothuythanhtam.git
- **Student ID 1: B2111949**
- **Student Name 1: Ngô Thụy Thanh Tâm**
- **Student ID 2: B2111959**
- **Student Name 2: Nguyễn Thị Hoài Thương**
- **Class/Group Number: CT312HM01**

I. Introduction

- *Project/application description:* The Mini Supermarket Application is a mobile app designed to facilitate supermarket transactions. It allows users to browse products, place orders, and manage their accounts, while administrators can handle inventory and order processing.

- *A task assignment sheet for each member if working in groups.*

Team Member	Tasks Assigned
Ngô Thụy Thanh Tâm	- Administrator-side features: sign up, sign in, admin's home page, CRUD products, view and update status orders, view customers, log out. - Database management using Pocketbase.
Nguyễn Thị Hoài Thương	- Customer-side features: CRUD Carts, Update-Status/Create/View Orders, update/view/delete information user, Home Page Customer, Detail Product Screen. - Database management using Pocketbase.

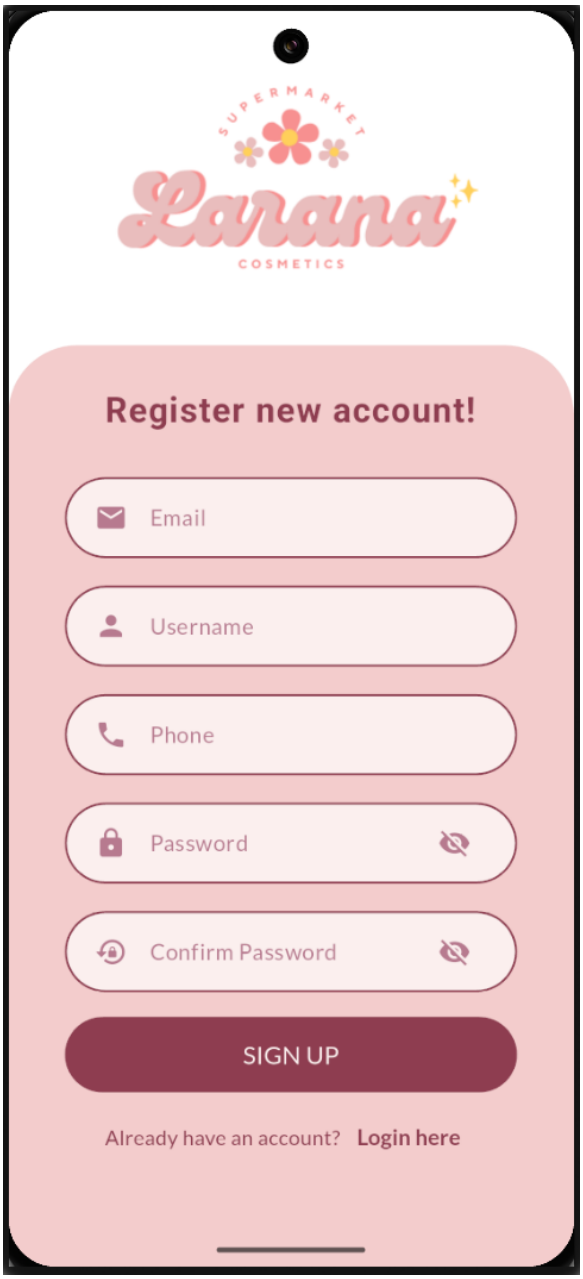
II. Details of implemented features

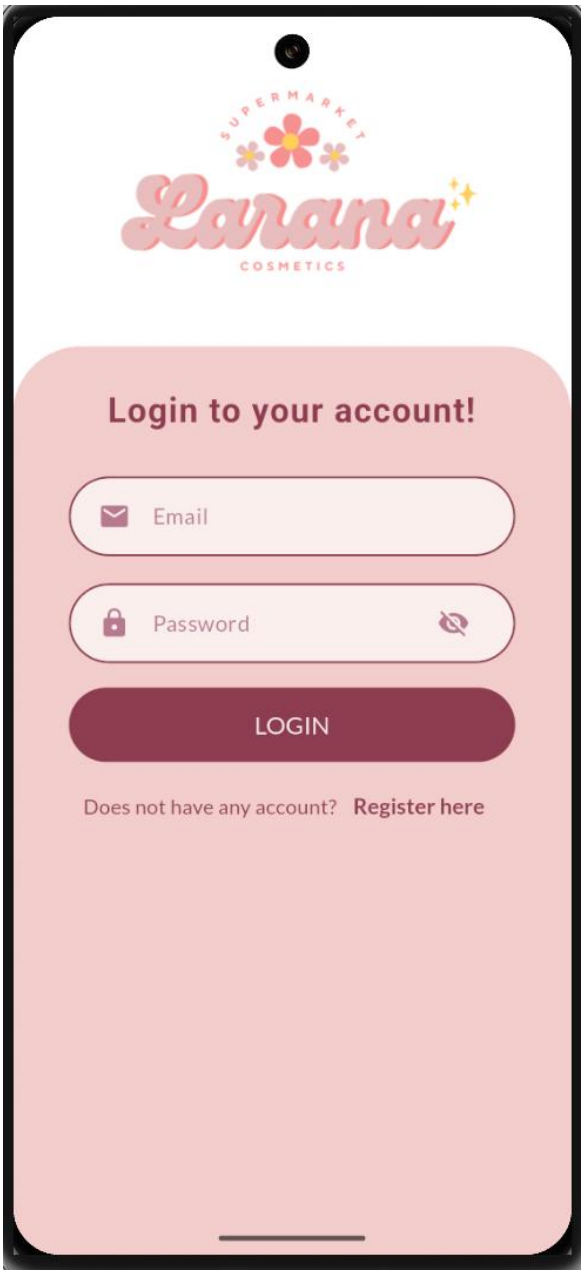
1. Feature / Application page 1: Sign Up – Sign In

Description: This feature handles user login and signup functionalities. The page allows users to either log in to an existing account or register a new account by providing credentials such as email, password, username, and phone number. It is a common entry point for applications requiring user authentication, ensuring that only authorized users can access the app's features. The implementation integrates with a backend service (PocketBase) for user management and authentication.

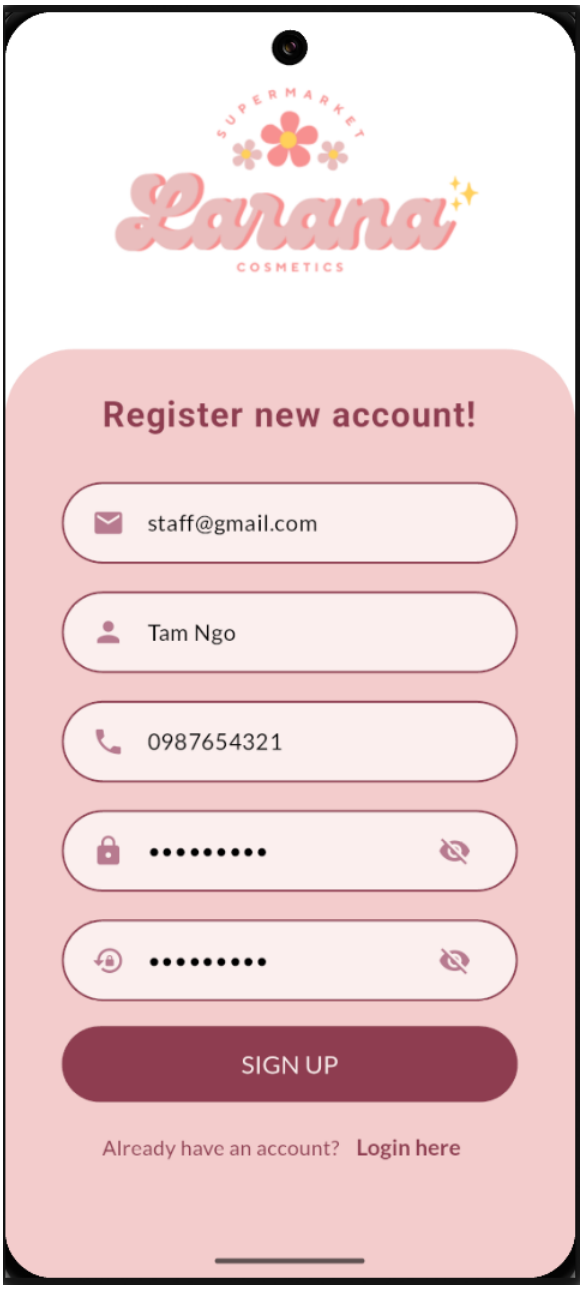
Screenshots:

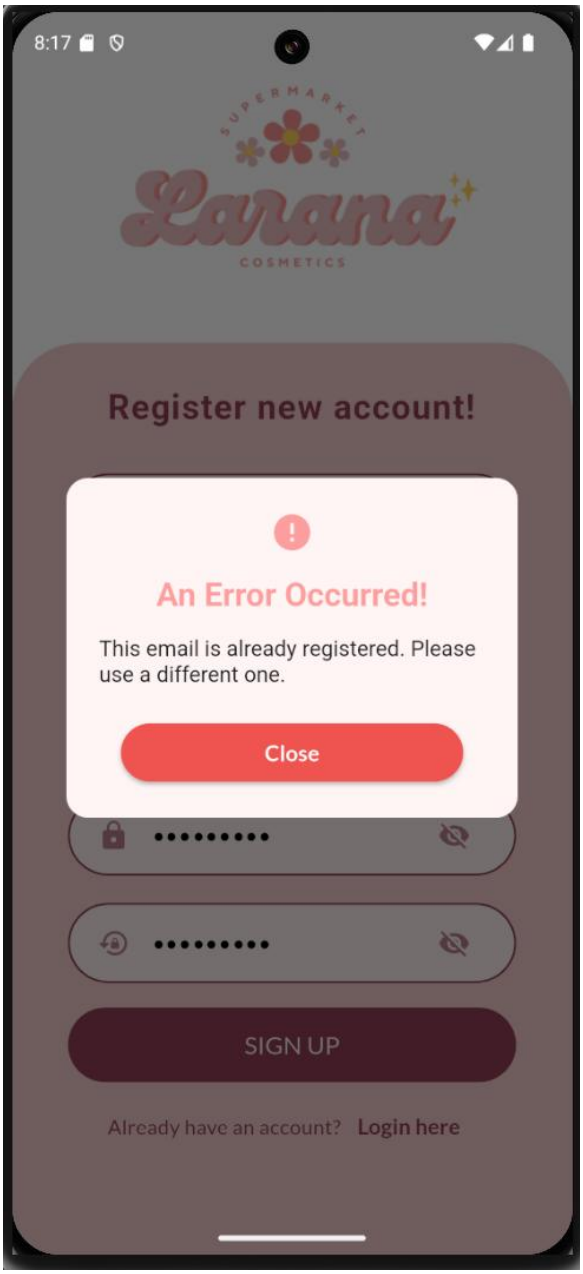
Interface when opening app

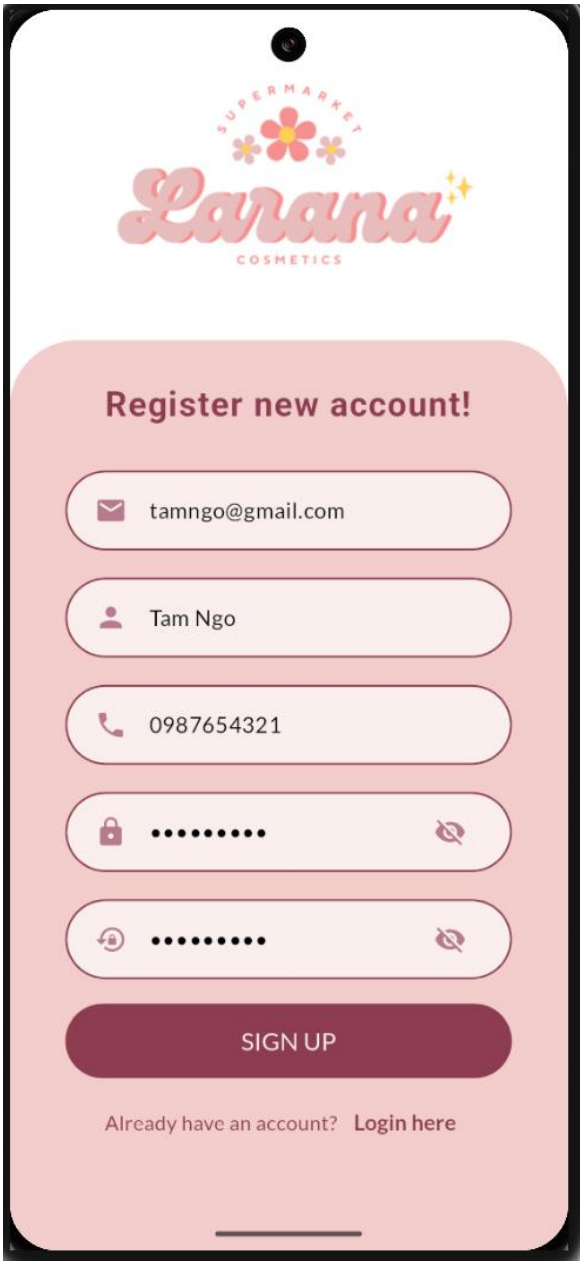


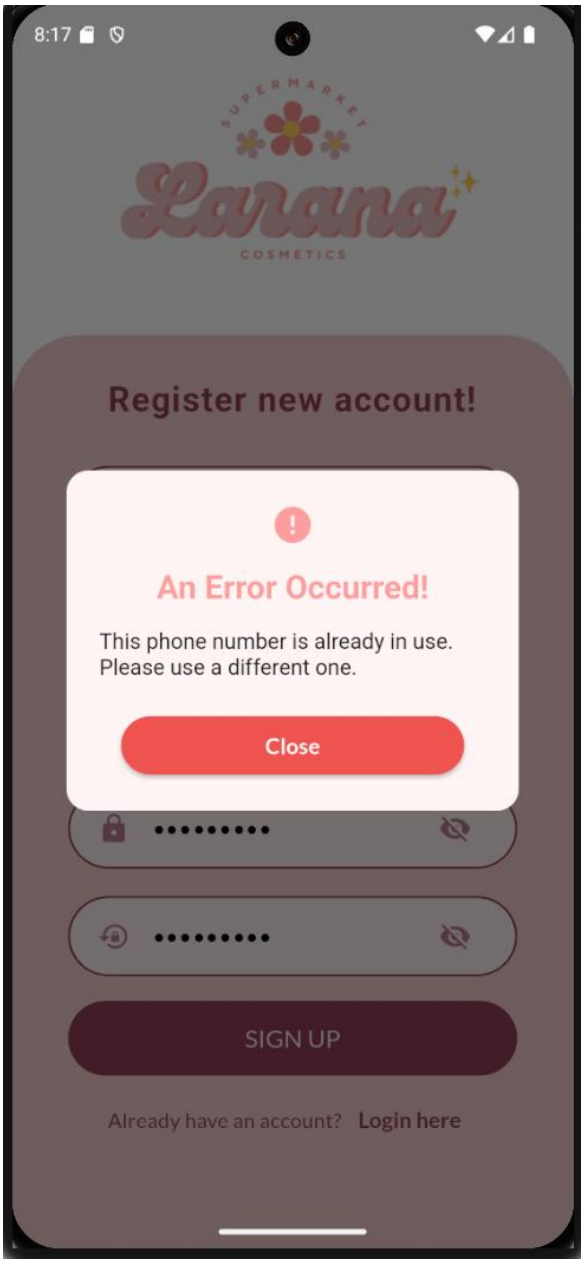


If you register with email and the phone number is already in use, a message will be displayed.

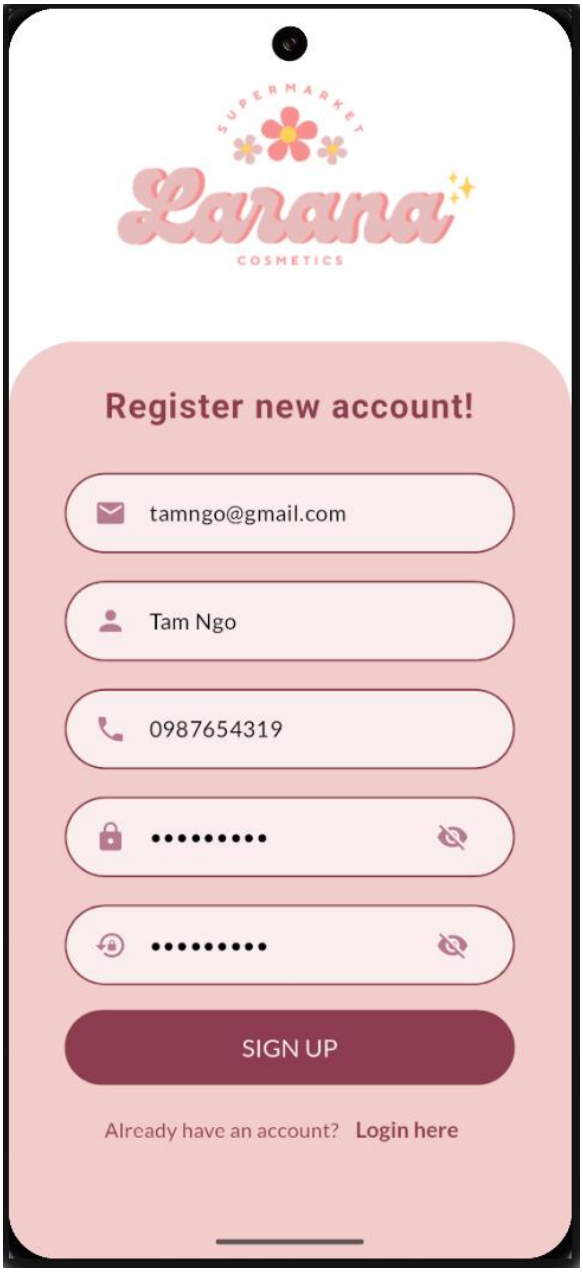


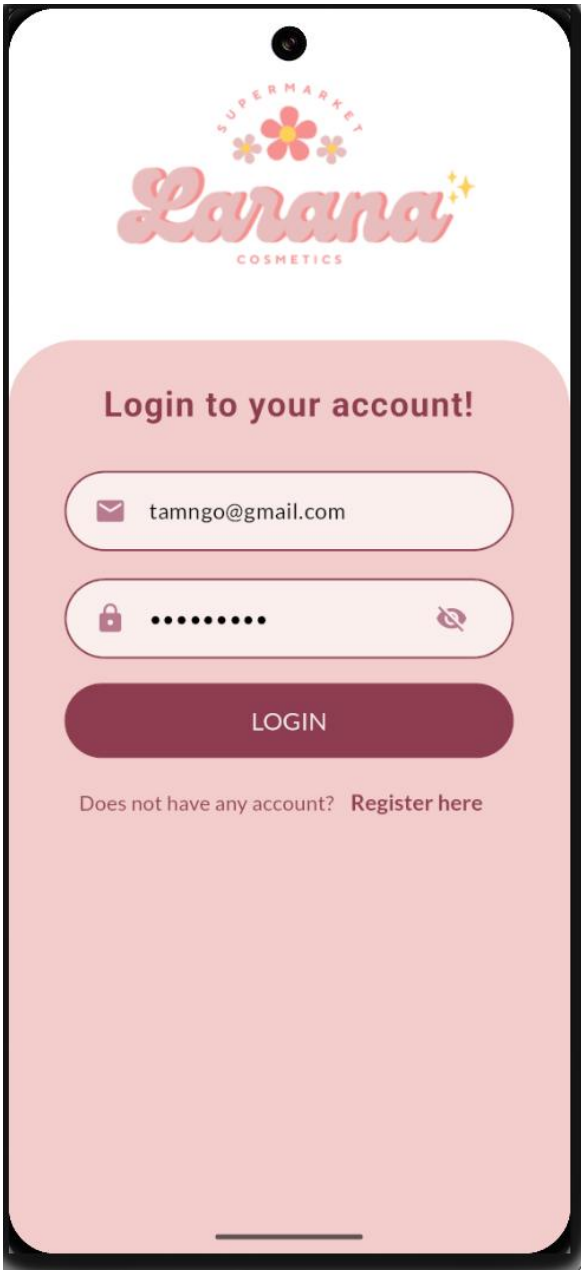



SUPERMARKET
Larana
COSMETICS
Register new account!
tamngo@gmail.com
Tam Ngo
0987654321
SIGN UP
Already have an account? Login here

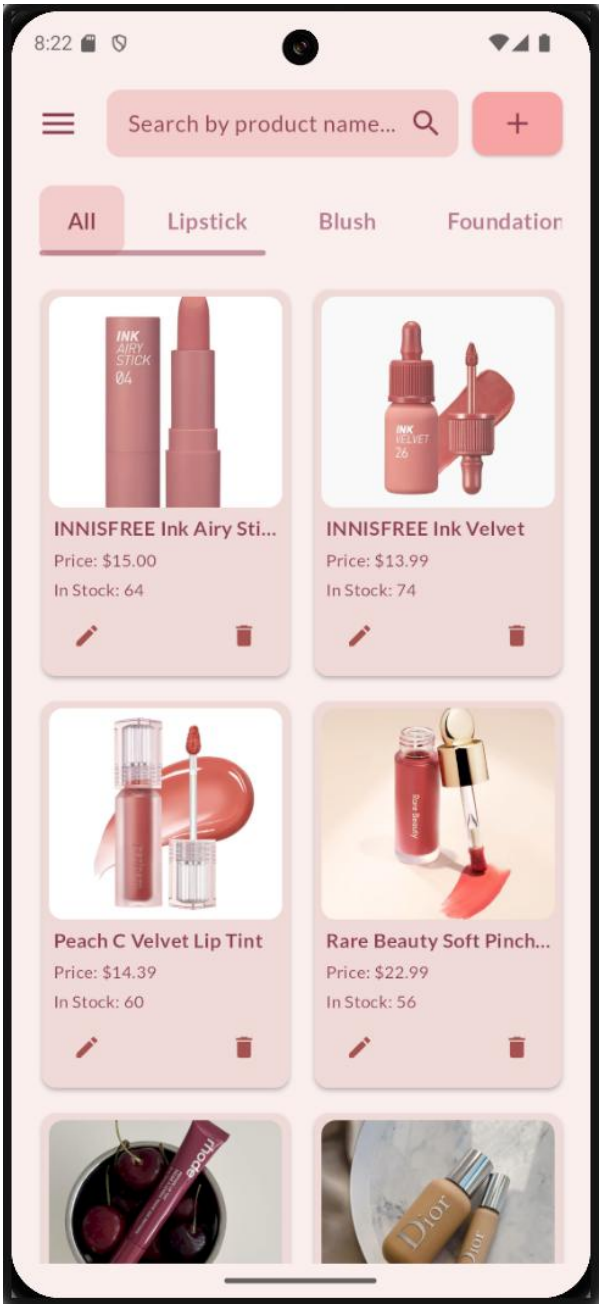

8:17
SUPERMARKET
Larana
COSMETICS
Register new account!
An Error Occurred!
This phone number is already in use.
Please use a different one.
Close
SIGN UP
Already have an account? Login here

When registration is successful, the app will automatically redirect to the login page.

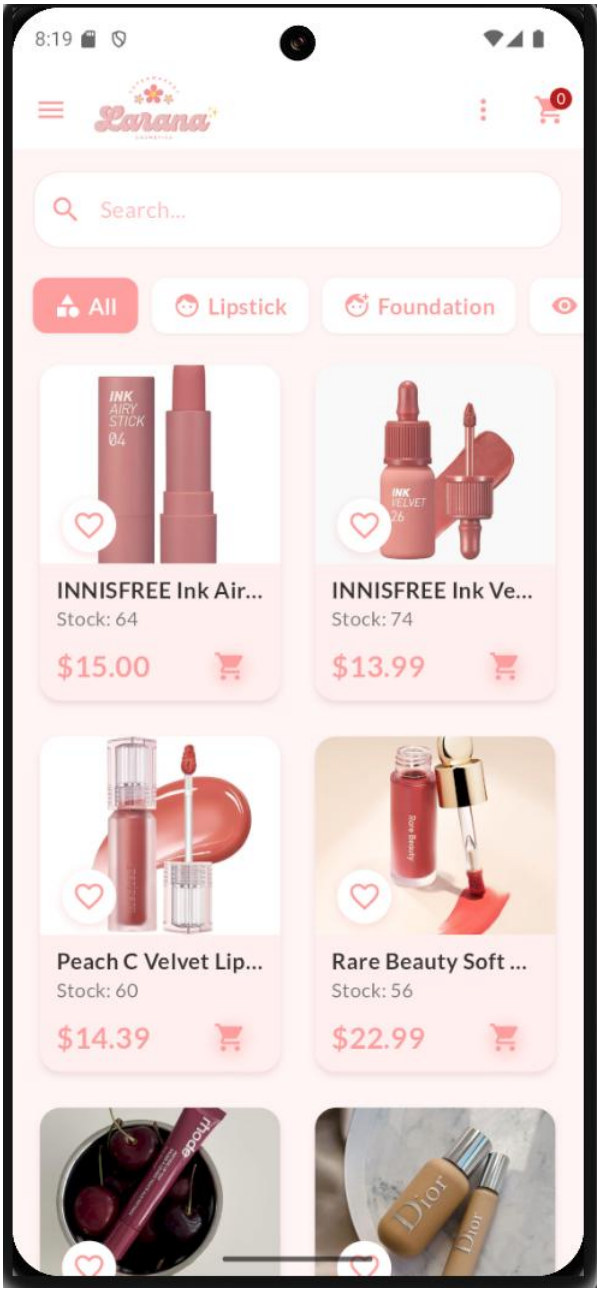




When logging in, the login function will check if the user is in the customer or admin role. If it is a customer, it will be redirected to the home page, and if it is an admin, it will be redirected to the dashboard page.



Admin Page



Customer Page

Implementation details:

- ***Widgets Used in this feature:***
  - Scaffold: Provides the basic structure of the page (background, body).
  - SingleChildScrollView: Ensures the content is scrollable if it exceeds the screen size.
  - Column: Arranges child widgets vertically (e.g., logo and form).
  - Stack: Overlays the logo image on the screen.
  - Padding: Adds spacing around widgets.
  - SizedBox: Controls the size of the logo image.
  - Image.asset: Displays the app logo from the assets folder.
  - Container: Used for styling (e.g., background color, border radius).
  - Text: Displays static text (e.g., "Login to your account!").
  - Form: Manages form input validation and saving.
  - TextFormField: Captures user input (e.g., email, password).
  - TextButton: Handles actions like submitting the form or switching auth modes.
  - Icon: Displays icons next to input fields (e.g., email, lock).
  - CircularProgressIndicator: Shows a loading state during submission.
  - Expanded: Ensures the form takes up available space.
  - Row: Aligns widgets horizontally (e.g., auth mode switch text and button).
  - IconButton: Toggles password visibility.
  - ValueListenableBuilder: A widget that rebuilds based on changes to a ValueNotifier (e.g., _isSubmitting). This is used to dynamically show a loading indicator or submit button during form submission. It’s not a basic widget and is typically introduced in more advanced state management lessons.
  - TextFieldContainer: A custom widget defined in the code to encapsulate and style TextFormField widgets. It’s not a built-in Flutter widget but a reusable component created for this app.

Libraries and Plugins Used

- flutter/material.dart: Provides the Material Design widgets and framework for building the UI.
- provider: A state management library used to manage and share the AuthManager state across the app. It allows the AuthCard to access and call methods on AuthManager (e.g., context.read<AuthManager>().login()).
- pocketbase: A plugin for interacting with the PocketBase backend, which handles user authentication and data storage remotely. It provides methods like authWithPassword and create for login and signup.
- dart:developer: Used for logging (log) to debug errors during development.
- flutter/foundation.dart: Provides ChangeNotifier for state management in AuthManager.

State Management

This feature uses a shared state management solution with the Provider package and a ChangeNotifier-based AuthManager class.

Code Architecture:

- Model Layer: User class defines the data structure for a user (e.g., id, email, username).
- Service Layer: AuthService handles communication with PocketBase (e.g., signup, login, logout) and notifies AuthManager of changes via the onAuthChange callback.
- Manager Layer: AuthManager acts as a bridge between the service and UI, managing state and business logic.
- UI Layer: AuthScreen and AuthCard render the UI and interact with AuthManager to perform actions.

How It Works:

- AuthManager is a class that extends ChangeNotifier, allowing it to notify listeners (e.g., UI widgets) when its state changes (e.g., user logs in or out).

- It maintains the current user state (_loggedInUser) and exposes properties like isAuth, isStaff, and isCustomer to check authentication status and roles.
- The Provider package makes AuthManager available globally, so widgets like AuthCard can access it via context.read<AuthManager>() or context.watch<AuthManager>().
- When authentication events occur (e.g., login, signup, logout), AuthManager updates _loggedInUser and calls notifyListeners(), triggering UI updates.

Data Reading and Storage

This feature reads and stores data remotely using the PocketBase backend.

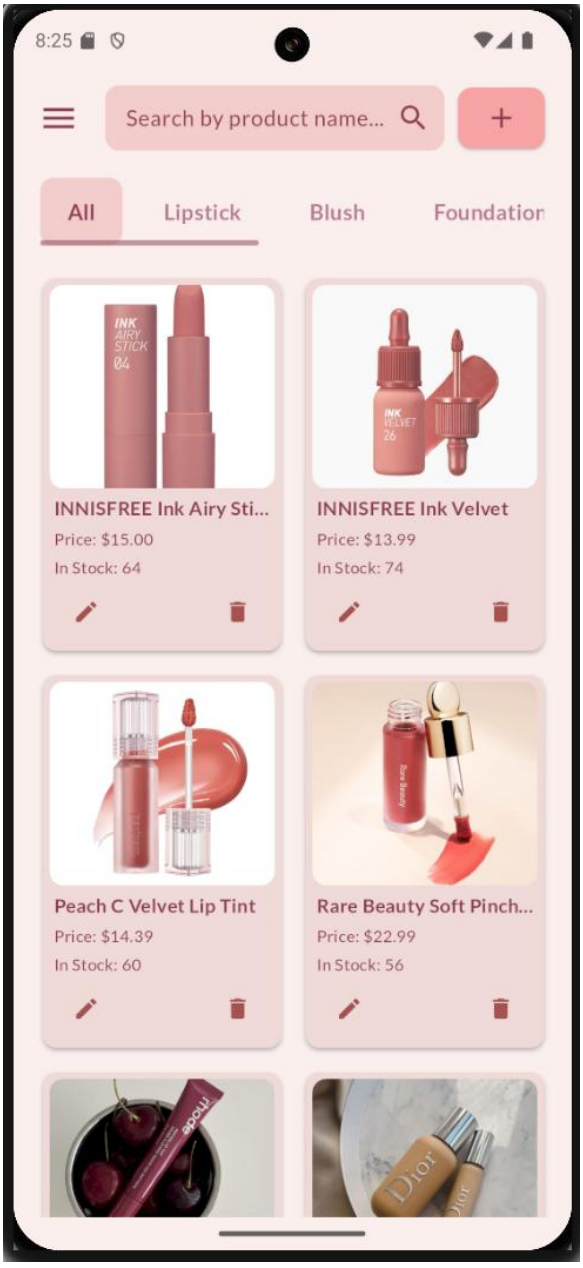
- Data Table Structure (PocketBase users Collection):
  - id (String): Unique identifier for the user (auto-generated by PocketBase).
  - email (String): User's email address (required).
  - username (String): User's chosen username (required).
  - phone (String): User's phone number (required).
  - password (String): Encrypted password (required, managed by PocketBase).
  - passwordConfirm (String): Password confirmation (required for signup).
  - urole (String): User role (e.g., "customer" or "staff", defaults to "customer").
  - address (String, optional): User's address.
  - avatar (String, optional): URL or reference to the user's avatar image.
  - emailVisibility (Boolean): Whether the email is publicly visible (set to true in signup).
- PocketBase API Interaction:
  - Signup:
    - Call: pb.collection('users').create(body: {...})
    - Input: A map with username, email, phone, password, passwordConfirm, urole, and emailVisibility.

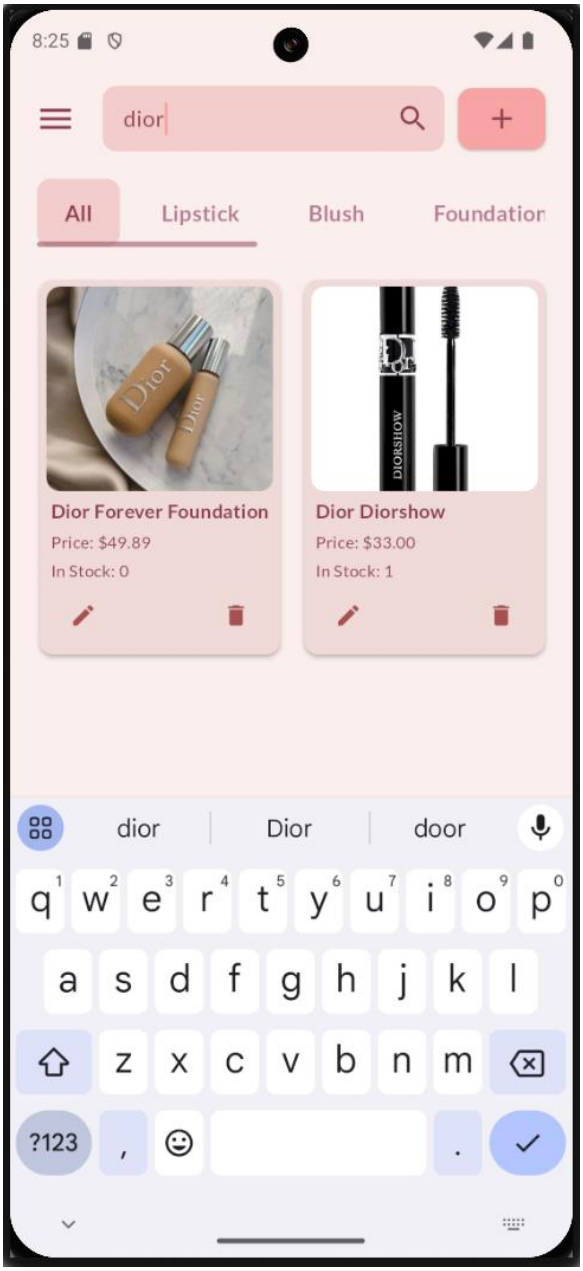
        - Output: A RecordModel object, converted to a User instance via User.fromJson().
    - Login:
        - Call: pb.collection('users').authWithPassword(email, password)
        - Input: email and password.
        - Output: An AuthRecord with user data, converted to a User instance.
    - Logout:
        - Call: pb.authStore.clear()
        - Input: None.
        - Output: Clears the auth store, effectively logging out the user.
    - Get User:
        - Call: pb.authStore.record
        - Input: None (reads from local auth store).
        - Output: A RecordModel or null, converted to a User instance if present.

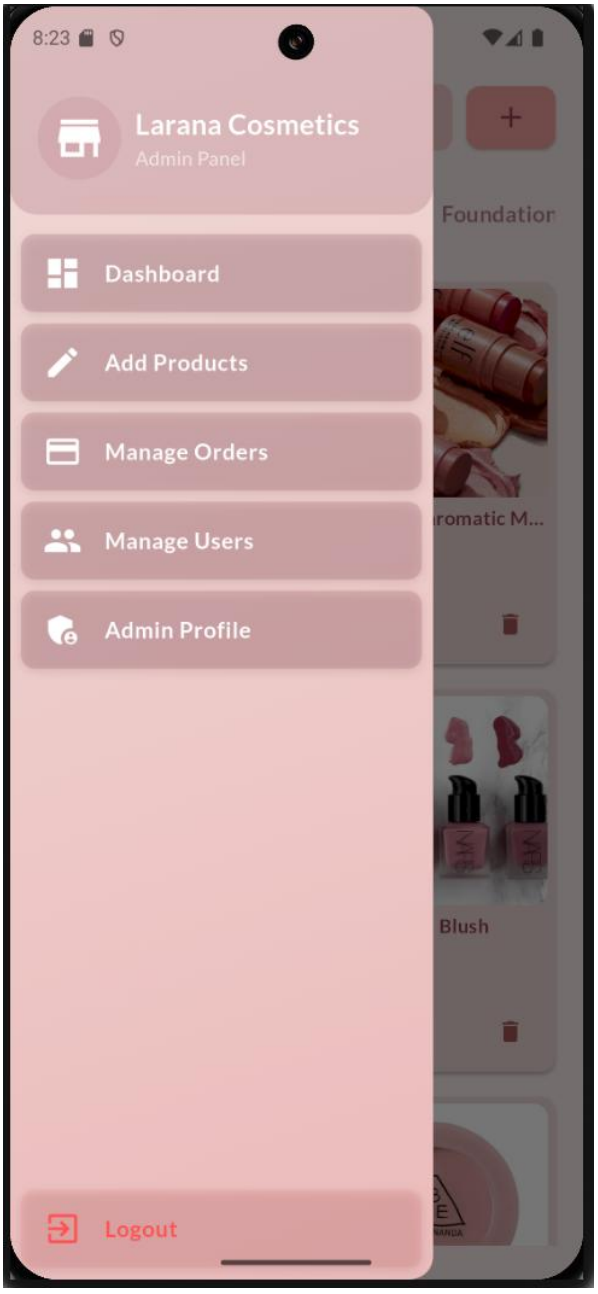
2. Feature / Application page 2: CRUD products with admin role

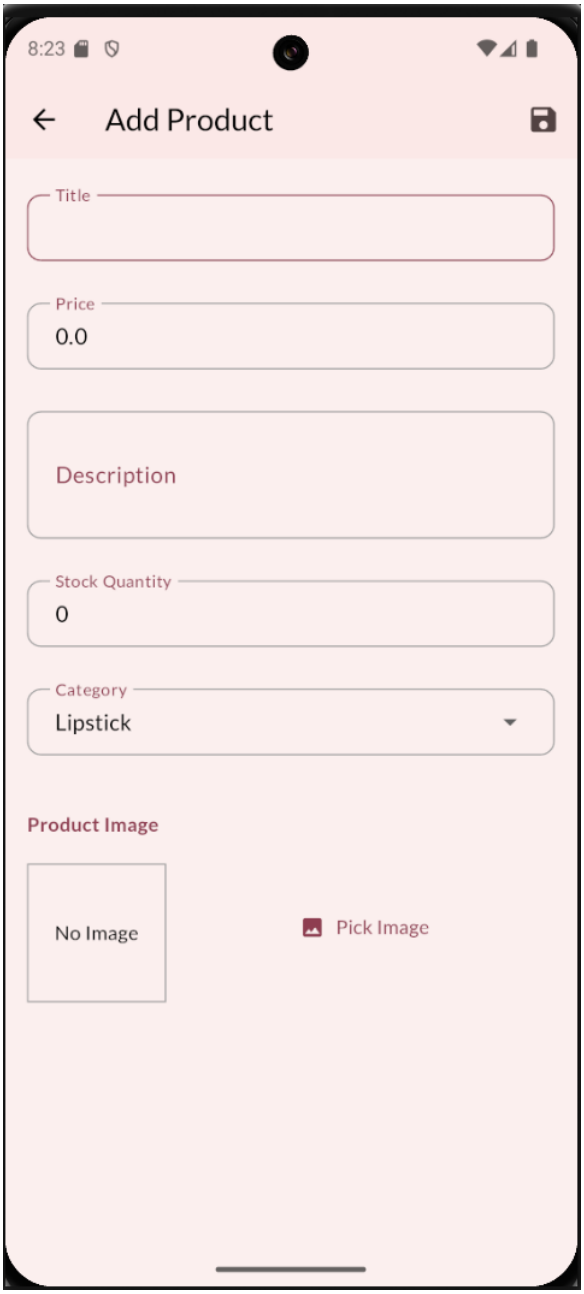
Description: This feature is used to manage product inventory, enabling admins to add, edit, delete, search and browse products stored in a backend system (PocketBase).

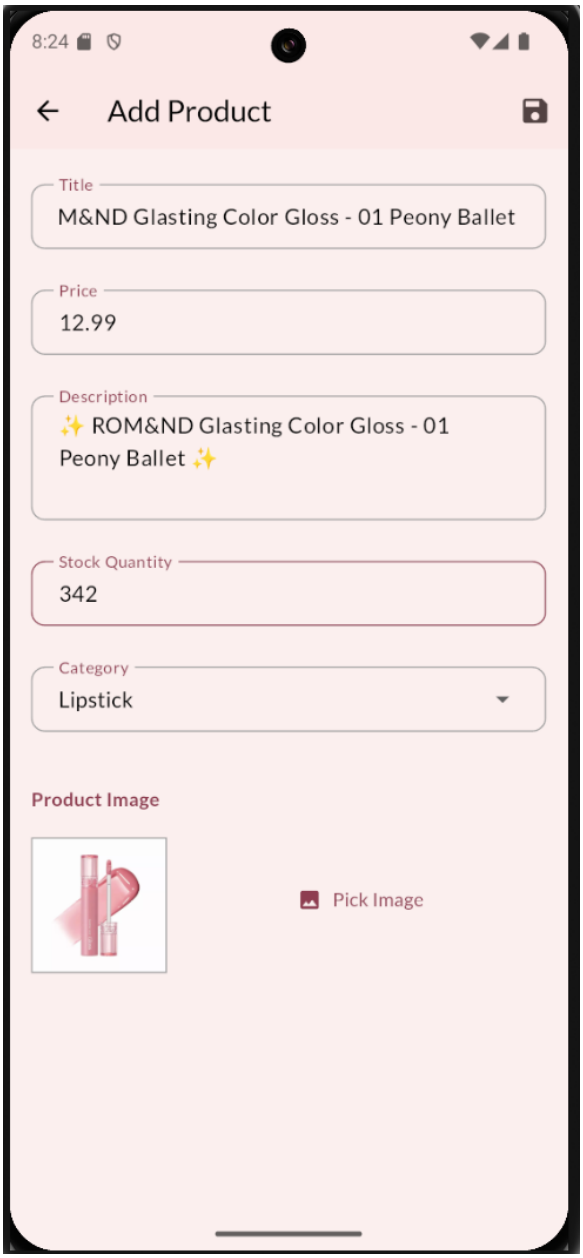
Screenshots:

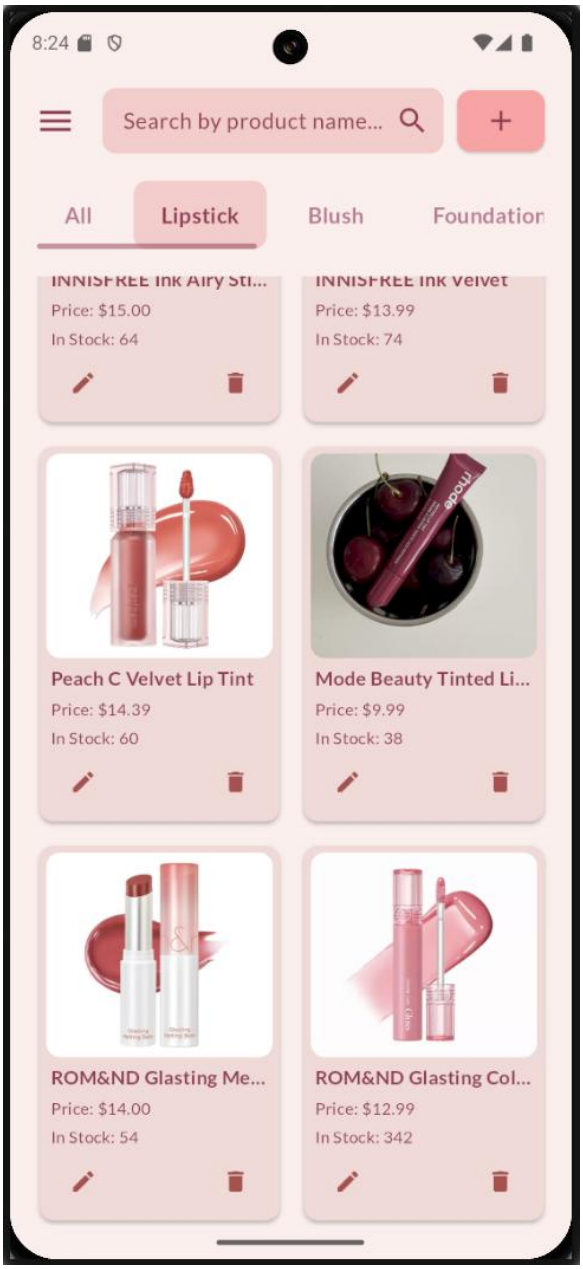




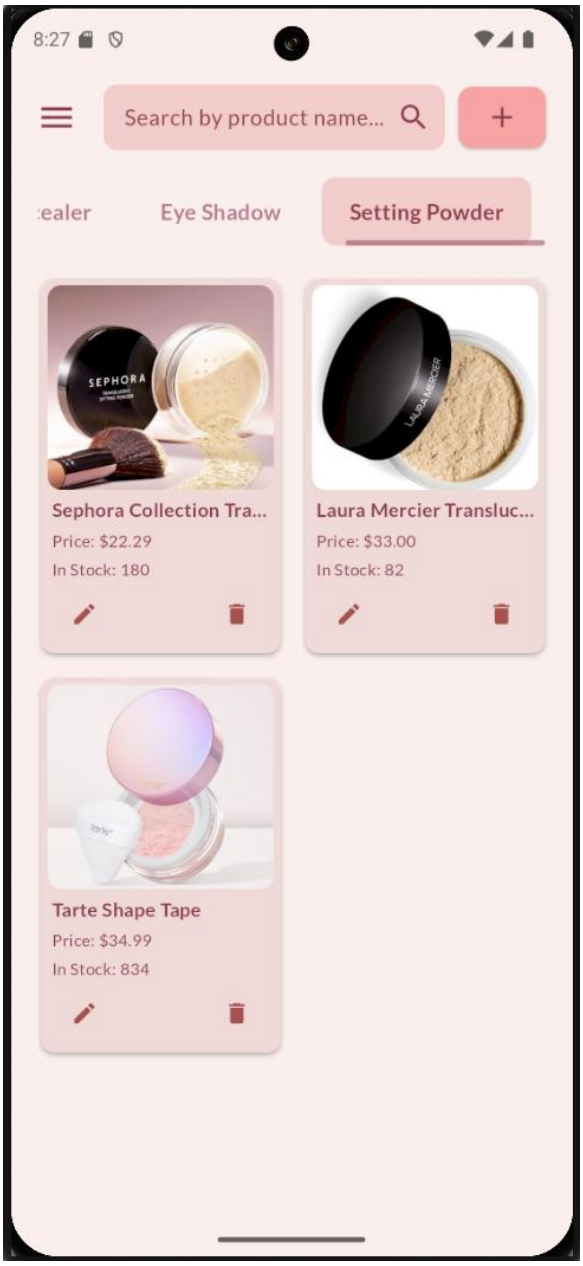

8:23
Larana Cosmetics
Admin Panel
Dashboard
Add Products
Manage Orders
Manage Users
Admin Profile
Logout
Foundatior
romatic M...
Blush

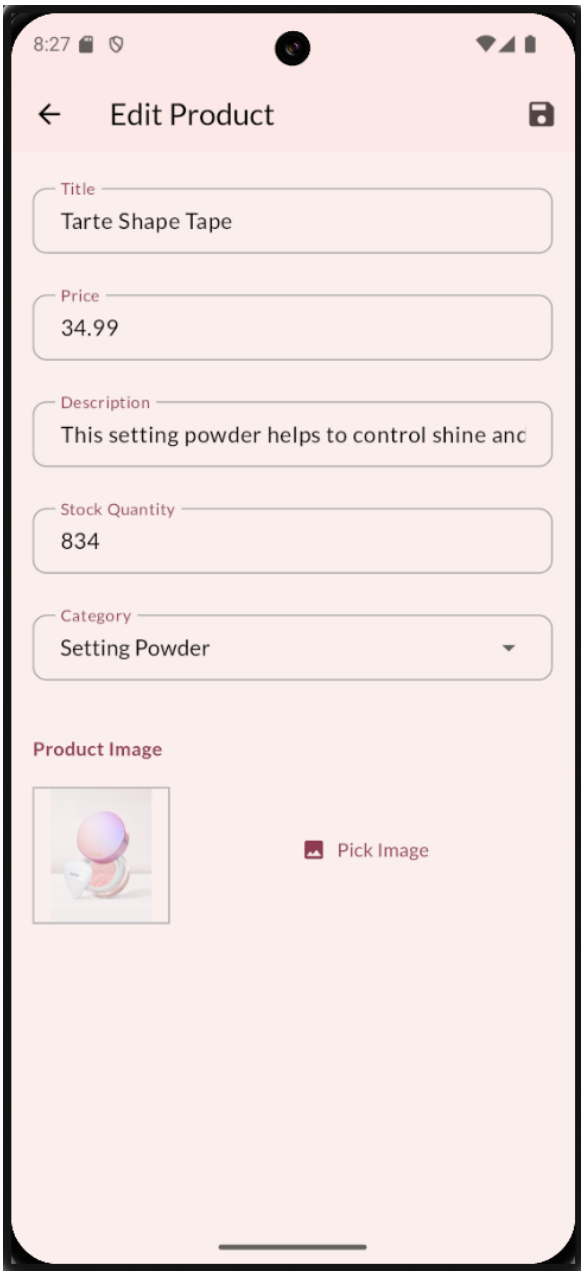

8:23
Add Product
Title
Price
0.0
Description
Stock Quantity
0
Category
Lipstick
Product Image
No Image
Pick Image

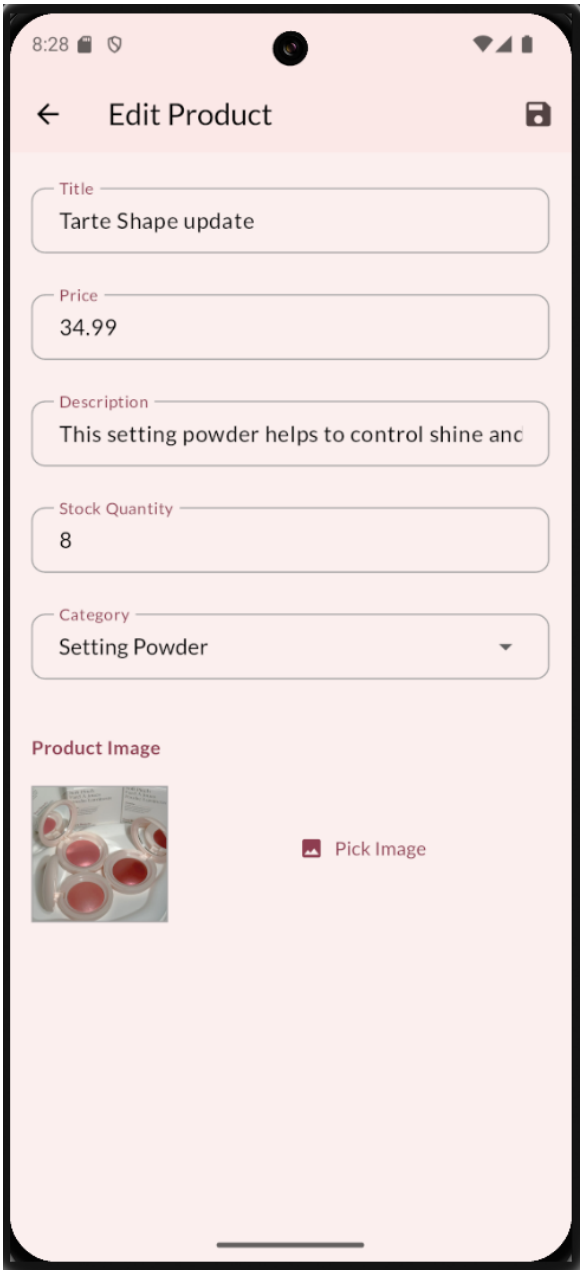

8:24
Add Product
Title
M&ND Glasting Color Gloss - 01 Peony Ballet
Price
12.99
Description
✨ ROM&ND Glasting Color Gloss - 01 Peony Ballet ✨
Stock Quantity
342
Category
Lipstick
Product Image
Pick Image

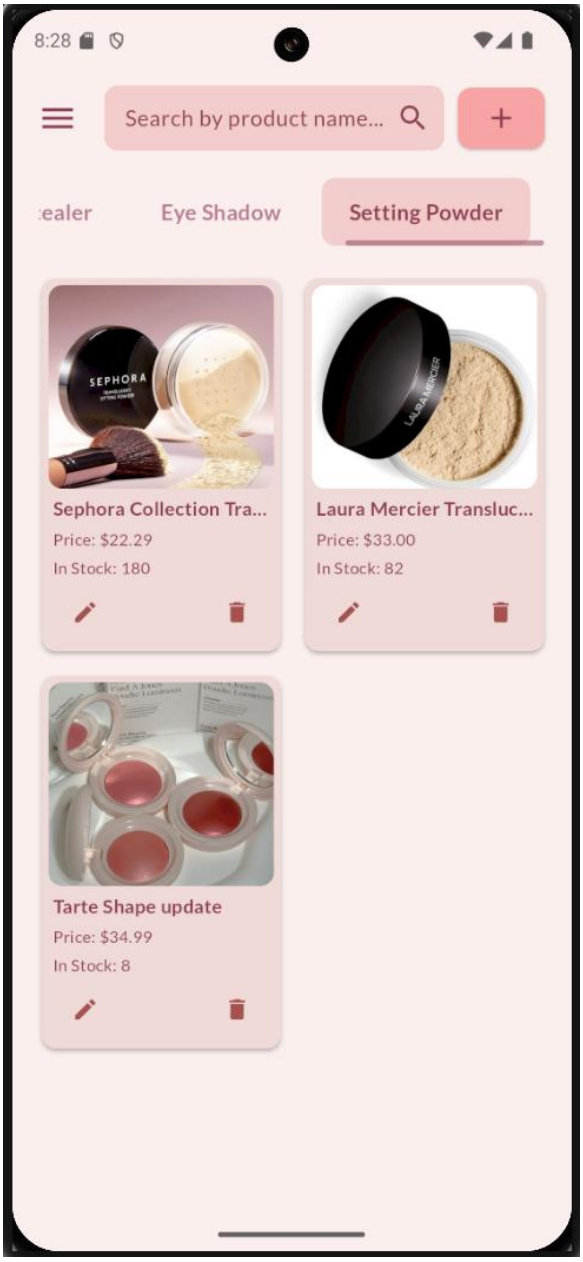

8:24
Search by product name...
All
Lipstick
Blush
Foundation
Price: $15.00
In Stock: 64
Price: $13.99
In Stock: 74
Peach C Velvet Lip Tint
Price: $14.39
In Stock: 60
Mode Beauty Tinted Li...
Price: $9.99
In Stock: 38
ROM&ND Glasting Me...
Price: $14.00
In Stock: 54
ROM&ND Glasting Col...
Price: $12.99
In Stock: 342

Edit

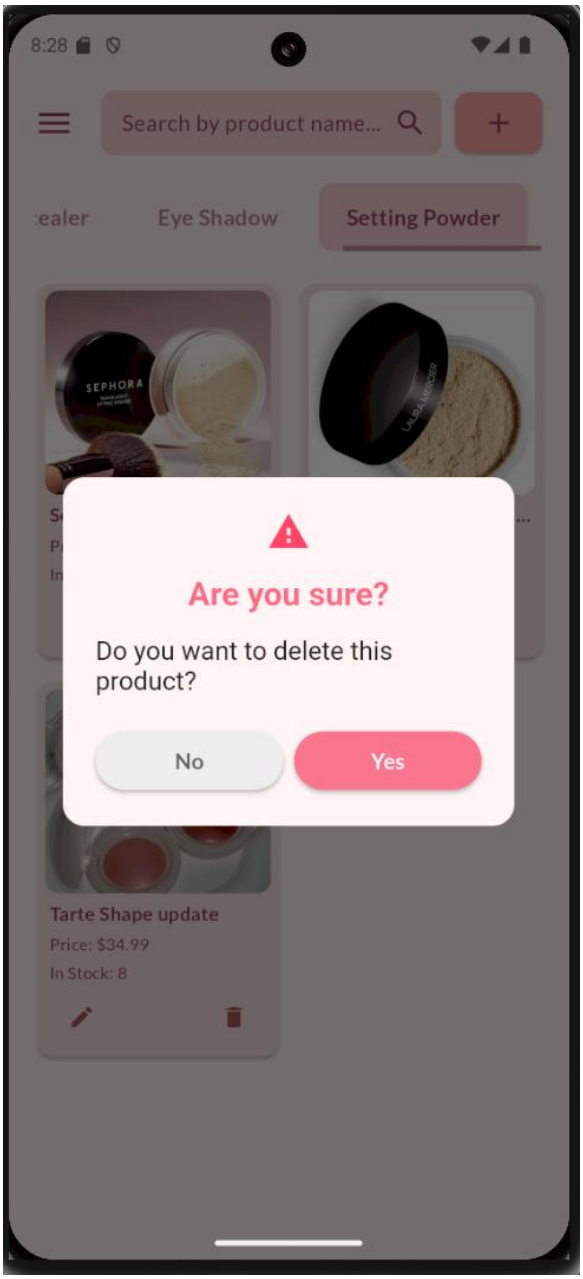

8:27
Search by product name...
ealer
Eye Shadow
Setting Powder
SEPHORA
Sephora Collection Tra...
Price: $22.29
In Stock: 180
Laura Mercier Transluc...
Price: $33.00
In Stock: 82
Tarte Shape Tape
Price: $34.99
In Stock: 834

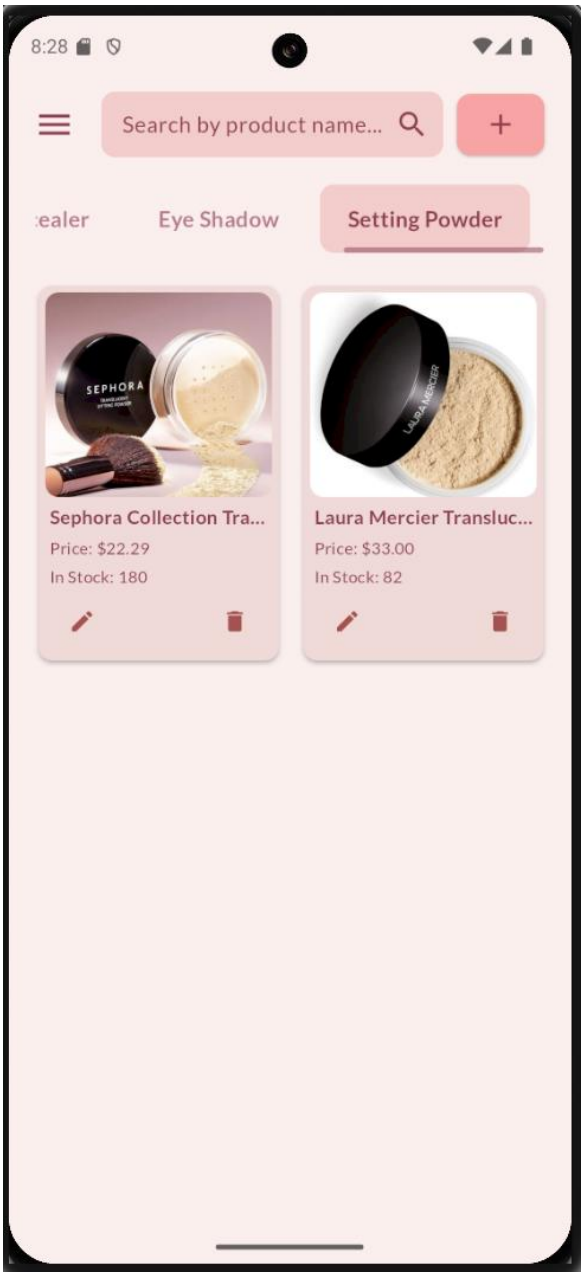

8:27
Edit Product
Title
Tarte Shape Tape
Price
34.99
Description
This setting powder helps to control shine anc
Stock Quantity
834
Category
Setting Powder
Product Image
Pick Image


8:28
Edit Product
Title
Tarte Shape update
Price
34.99
Description
This setting powder helps to control shine anc
Stock Quantity
8
Category
Setting Powder
Product Image
Pick Image


8:28
Search by product name...
ealer
Eye Shadow
Setting Powder
SEPHORA
Sephora Collection Tra...
Price: $22.29
In Stock: 180
Laura Mercier Transluc...
Price: $33.00
In Stock: 82
Tarte Shape update
Price: $34.99
In Stock: 8

Delete


8:28
Search by product name...
ealer
Eye Shadow
Setting Powder
Are you sure?
Do you want to delete this product?
No
Yes
Tarte Shape update
Price: $34.99
In Stock: 8


8:28
Search by product name...
ealer
Eye Shadow
Setting Powder
Sephora Collection Tra...
Price: $22.29
In Stock: 180
Laura Mercier Transluc...
Price: $33.00
In Stock: 82

Implementation details:

- ***Widgets Used in this feature:***
  - Scaffold: Provides the basic structure (app bar, body, floating action button, drawer).
  - AppBar: Displays the title and action buttons (e.g., save).
  - Container: Used for styling and layout (e.g., background color, borders).
  - Column: Arranges widgets vertically (e.g., form fields, product list).
  - Row: Arranges widgets horizontally (e.g., image preview and pick button).
  - Text: Displays static text (e.g., labels, product details).
  - TextFormField: Captures user input (e.g., title, price).
  - Form: Manages form validation and saving.
  - IconButton: Handles actions (e.g., save, edit, delete).
  - TextButton: Triggers actions (e.g., picking an image).
  - CircularProgressIndicator: Shows loading states.
  - SingleChildScrollView: Ensures scrollable content in forms.
  - ListView: Displays categories horizontally in ProductsScreen.
  - GridView: Displays products in a grid layout in ProductsScreen.
  - Card: Wraps product items for a card-like appearance.
  - Image.network / Image.file: Displays product images (remote or local).
  - Padding: Adds spacing around widgets.
  - SizedBox: Controls spacing or widget sizes.
  - Center: Centers content (e.g., loading indicator).
  - DropdownButtonFormField: Allows category selection in forms.
  - SafeArea: Ensures content fits within the device's safe area.
  - FloatingActionButton: Navigates to the add product screen.

- AlertDialog: Confirms deletion of products.
- ScaffoldMessenger / SnackBar: Shows validation messages (e.g., "Please select an image").
- FittedBox: Scales images to fit within their containers.
- ClipRRect: Rounds image corners.
- Expanded: Ensures widgets take available space.
- Placeholder: Displays a placeholder when no image is available.
- RawScrollbar: Used in ProductsScreen to add a custom scrollbar to the horizontal category list. This is less common in introductory lessons and provides fine-grained control over scrollbar appearance (e.g., thumb color, thickness).
- SliverGridDelegateWithFixedCrossAxisCount: Used with GridView in ProductsScreen to define the grid layout (cross-axis count, spacing, aspect ratio). While not rare, it's more advanced than basic ListView or Column layouts.

Libraries and Plugins Used

- flutter/material.dart: Provides Material Design widgets and framework for building the UI.
- provider: A state management library used to manage and share the ProductsManager state across the app. It allows screens to access and call methods like addProduct or deleteProduct.
- pocketbase: A plugin for interacting with the PocketBase backend, handling product data storage and retrieval.
- http: Used with PocketBase to upload files (e.g., product images) as MultipartFile objects.
- image_picker: Enables picking images from the gallery for product featured images.
- flutter/foundation.dart: Provides ChangeNotifier for state management in ProductsManager.
- dart:io: Handles file operations (e.g., File for picked images).

State Management

This feature uses a shared state management solution with the Provider package and a ChangeNotifier-based ProductsManager class.

Code Architecture:

- Model Layer: Product class defines the data structure for a product (e.g., pid, title, price).
- Service Layer: ProductsService handles communication with PocketBase (e.g., addProduct, fetchProducts), abstracting backend interactions.
- Manager Layer: ProductsManager bridges the service and UI, managing state and business logic (e.g., updating the product list).
- UI Layer: ProductsScreen, AddProductScreen, and EditProductScreen render the UI and interact with ProductsManager for data operations.

How It Works:

- ProductsManager extends ChangeNotifier, managing a list of products (_items) and categories (_categories). It notifies listeners (e.g., UI widgets) when the state changes (e.g., a product is added or deleted).
- The Provider package makes ProductsManager available globally, allowing widgets like ProductsScreen, AddProductScreen, and EditProductScreen to access it via Provider.of<ProductsManager>(context) or context.read<ProductsManager>().
- When data operations occur (e.g., fetching products, adding a product), ProductsManager updates its internal state and calls notifyListeners(), triggering UI rebuilds.

Data Reading and Storage

This feature reads and stores data remotely using the PocketBase backend.

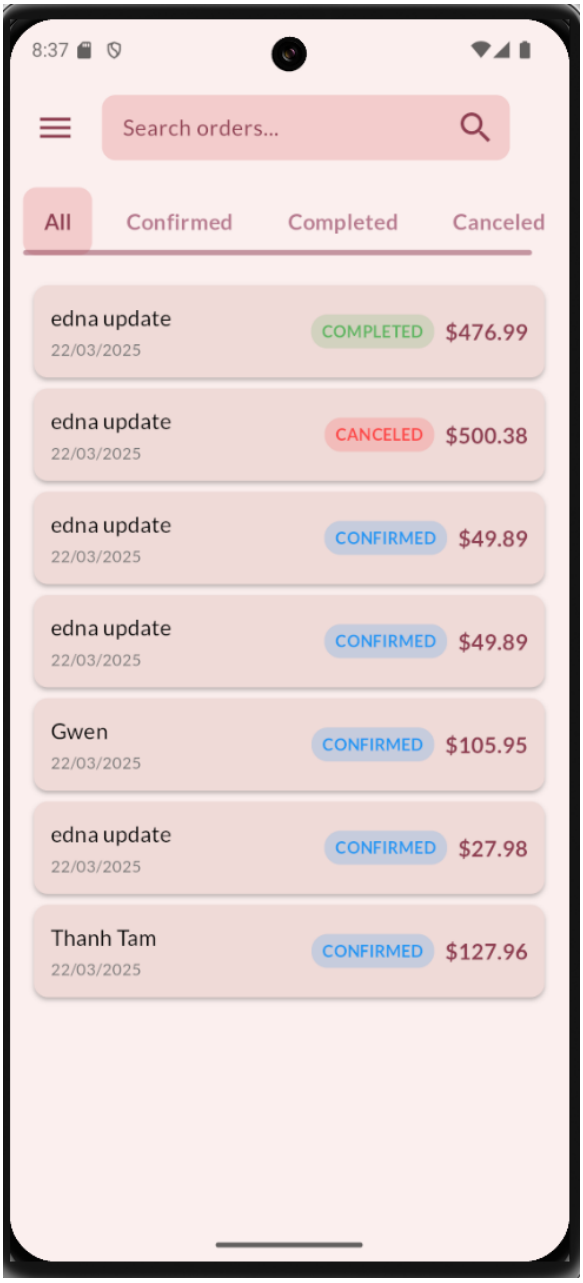
- Data Table Structure (PocketBase products Collection):
    - id (String): Unique identifier for the product (auto-generated by PocketBase, mapped to pid).
    - title (String): Product name (required).
    - description (String): Product description (required).
    - price (Double): Product price (required).
    - stockQuantity (Integer): Number of items in stock (required).
    - category (String): Product category (required).
    - userId (String): ID of the user who created the product (linked to the users collection).
    - featuredImage (File/String): Filename of the uploaded image (stored in PocketBase's file storage, with a URL generated via getUrl).
    - isFavorite (Boolean): Indicates if the product is favorited (defaults to false, though not used in UI here).
- PocketBase API Interaction:
    - Add Product:
        - Call: pb.collection('products').create(body: {...}, files: [...])
        - Input: A map with title, description, price, stockQuantity, category, userId, and an optional MultipartFile for featuredImage.
        - Output: A RecordModel object, converted to a Product with pid and imageUrl.
    - Fetch Products:
        - Call: pb.collection('products').getFullList(filter: ...)
        - Input: Optional filter string (e.g., "userId='userId' && category='category'").
        - Output: A list of RecordModel objects, converted to Product instances with imageUrl added.
    - Fetch Categories:
        - Call: pb.collection('products').getFullList()
        - Input: None.
        - Output: A list of unique category strings extracted from products.
    - Update Product:

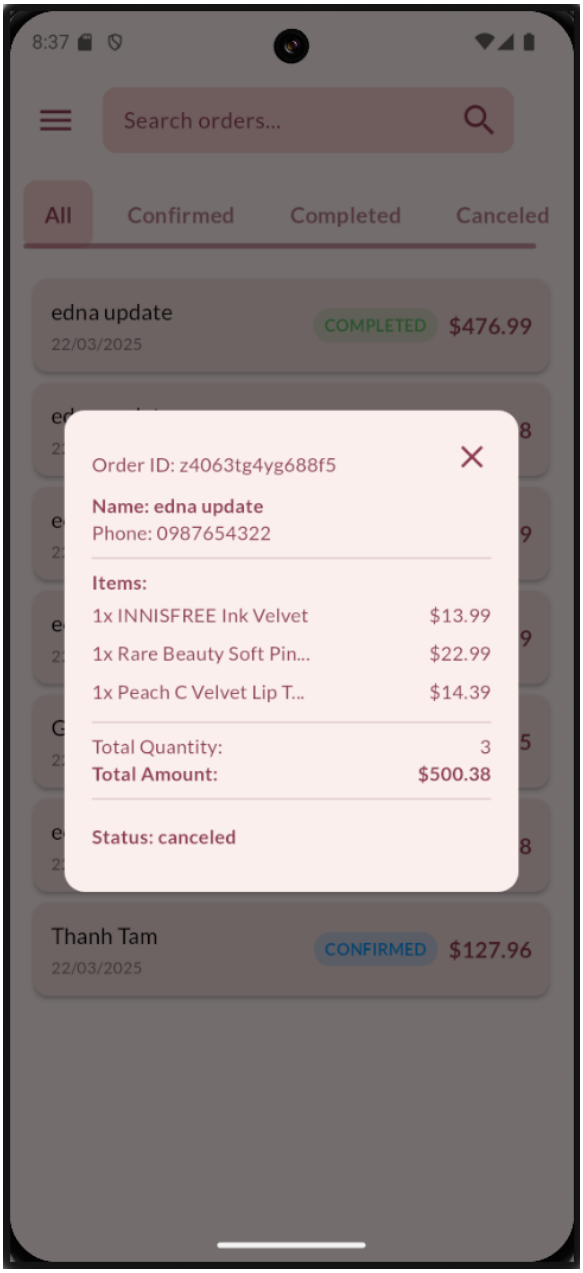
        - Call: pb.collection('products').update(id, body: {...}, files: [...])
        - Input: Product id, updated fields in a map, and an optional MultipartFile for featuredImage.
        - Output: A RecordModel object, converted to an updated Product.
    - Delete Product:
        - Call: pb.collection('products').delete(id)
        - Input: Product id.
        - Output: None (returns true on success).

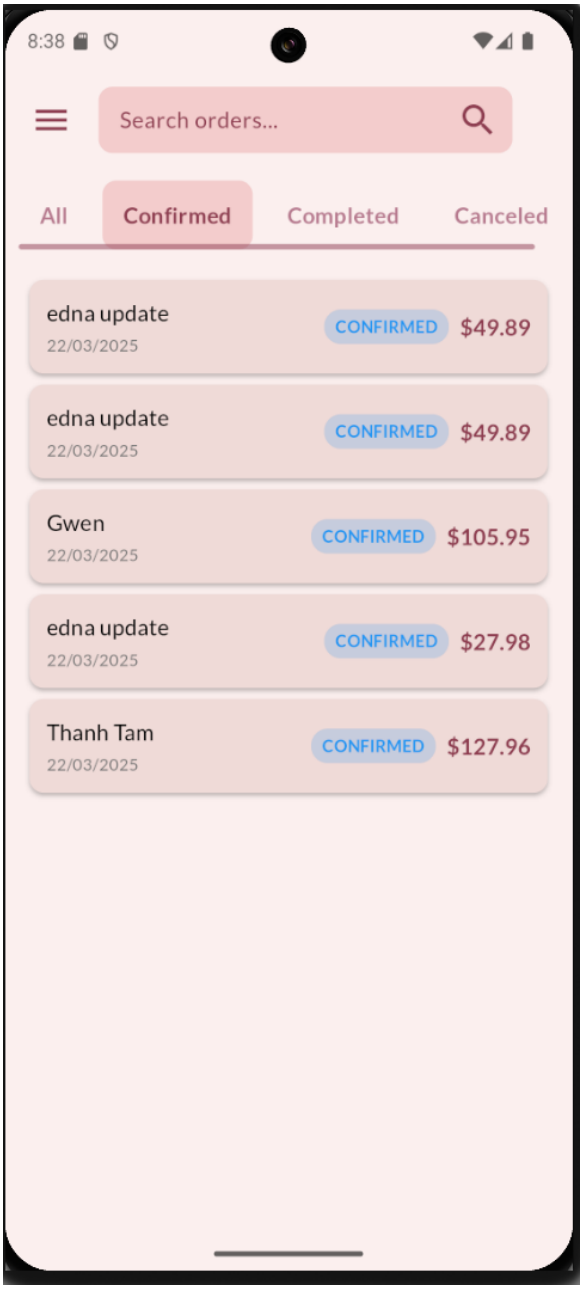
3. Feature / Application page 3: View and update orders with admin role

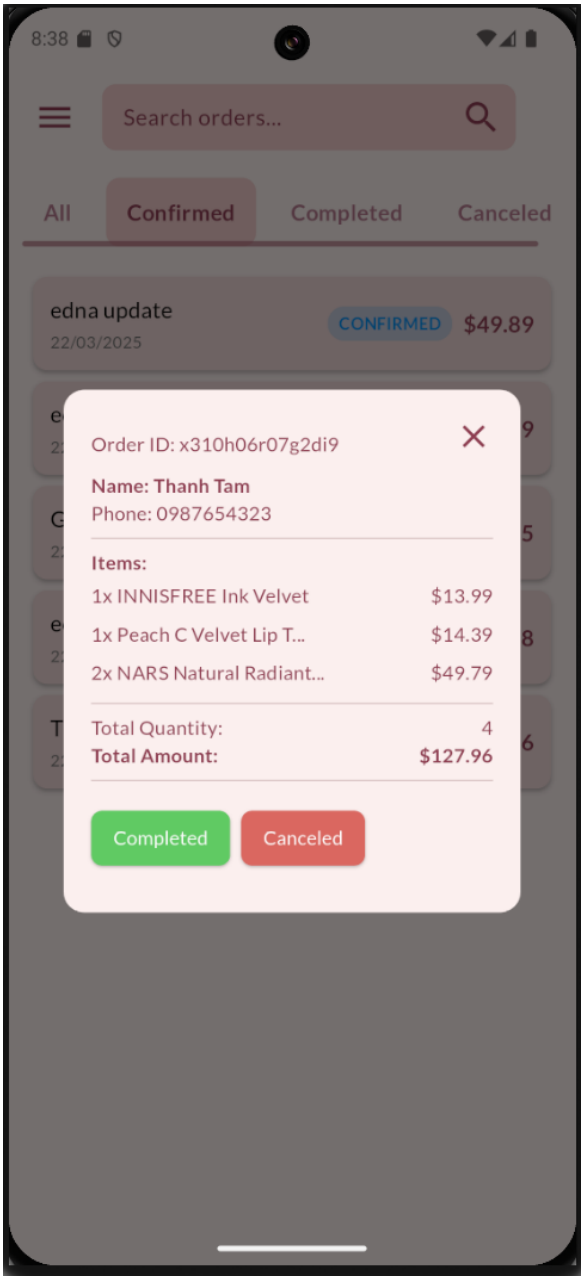
Description: This feature is used to track and manage orders, providing an interface for administrators to monitor order statuses, update them as needed.

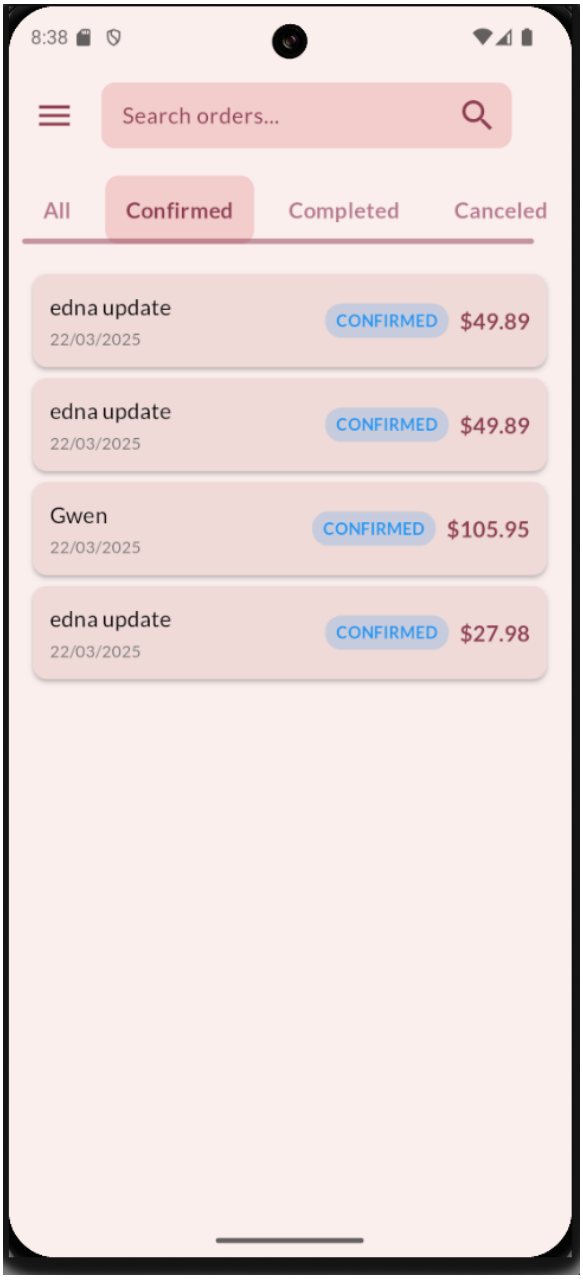
Screenshots:

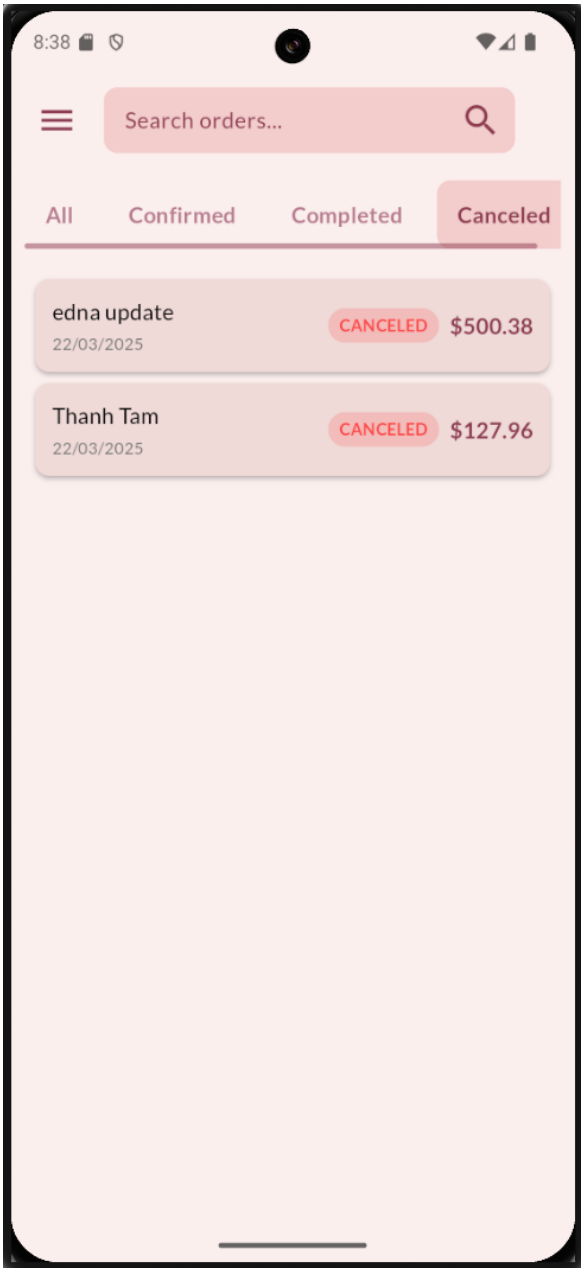


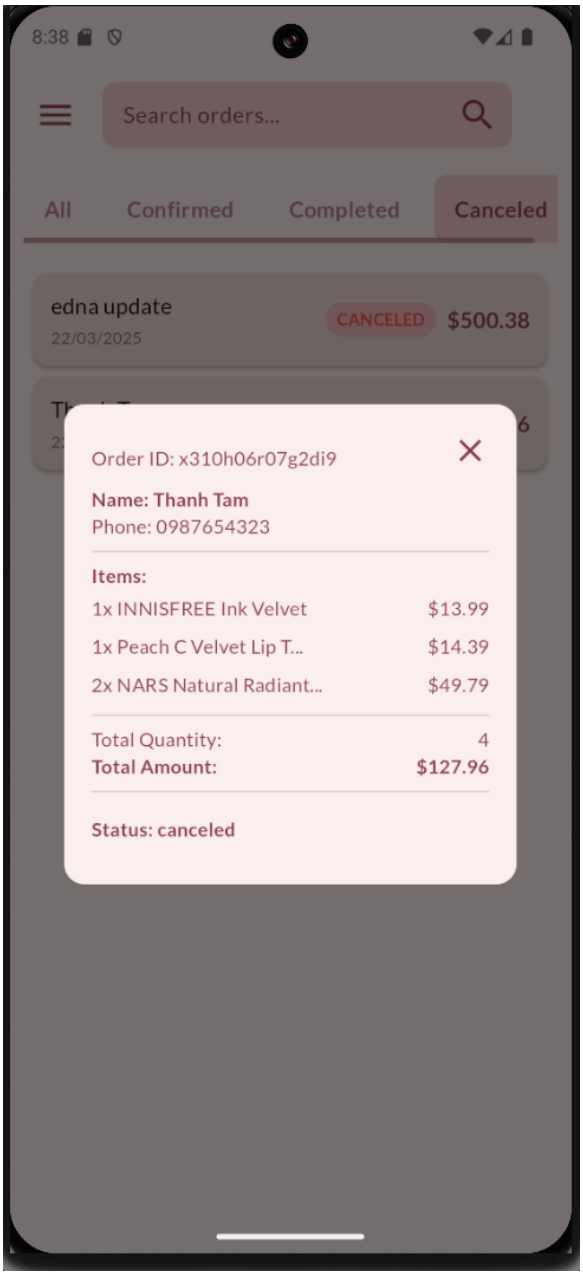



8:38
Search orders...
All
Confirmed
Completed
Canceled
edna update
22/03/2025
CONFIRMED
$49.89
edna update
22/03/2025
CONFIRMED
$49.89
Gwen
22/03/2025
CONFIRMED
$105.95
edna update
22/03/2025
CONFIRMED
$27.98
Thanh Tam
22/03/2025
CONFIRMED
$127.96


8:38
Search orders...
All
Confirmed
Completed
Canceled
edna update
22/03/2025
CONFIRMED
$49.89
Order ID: x310h06r07g2di9
Name: Thanh Tam
Phone: 0987654323
Items:
1x INNISFREE Ink Velvet
$13.99
1x Peach C Velvet Lip T...
$14.39
2x NARS Natural Radiant...
$49.79
Total Quantity:
4
Total Amount:
$127.96
Completed
Canceled


8:38
Search orders...
All
Confirmed
Completed
Canceled
edna update
22/03/2025
CONFIRMED
$49.89
edna update
22/03/2025
CONFIRMED
$49.89
Gwen
22/03/2025
CONFIRMED
$105.95
edna update
22/03/2025
CONFIRMED
$27.98


8:38
Search orders...
All
Confirmed
Completed
Canceled
edna update
22/03/2025
CANCELED
$500.38
Thanh Tam
22/03/2025
CANCELED
$127.96


8:38
Search orders...
All
Confirmed
Completed
Canceled
edna update
22/03/2025
CANCELED
$500.38
Order ID: x310h06r07g2di9
Name: Thanh Tam
Phone: 0987654323
Items:
1x INNISFREE Ink Velvet
$13.99
1x Peach C Velvet Lip T...
$14.39
2x NARS Natural Radiant...
$49.79
Total Quantity:
4
Total Amount:
$127.96
Status: canceled

Implementation details:

- ***Widgets Used in this feature***
  - Scaffold: Provides the basic structure (body, drawer).
  - SafeArea: Ensures content fits within the device's safe area.
  - Container: Used for styling and layout (e.g., background color, borders).
  - Column: Arranges widgets vertically (e.g., search bar, filters, order list).
  - Row: Arranges widgets horizontally (e.g., search bar with icon, status and amount).
  - Text: Displays static text (e.g., order details, status).
  - TextField: Captures search input.
  - IconButton: Opens the drawer or closes the dialog.
  - Icon: Displays icons (e.g., menu, search, close).
  - ListView: Displays orders vertically and statuses horizontally.
  - Card: Wraps each order item for a card-like appearance.
  - Padding: Adds spacing around widgets.
  - SizedBox: Controls spacing between widgets.
  - Center: Centers content (e.g., loading indicator).
  - FutureBuilder: Manages asynchronous order fetching and displays loading/error states.
  - Consumer: Rebuilds the UI based on OrdersManager state changes.
  - InkWell: Adds tap functionality to order cards.
  - Dialog: Shows order details in a popup.
  - SingleChildScrollView: Ensures scrollable content in the dialog or screen.
  - Divider: Separates sections in the dialog.
  - DropdownButton: Allows status updates in the dialog.
  - Stack: Positions the close button in the dialog.

- Positioned: Places the close button at the top-right of the dialog.
- CircularProgressIndicator: Shows loading states.
- Expanded: Ensures the order list takes available space.
- GestureDetector: Handles taps on status filter buttons.
- RawScrollbar: Used in OrdersScreen to add a custom scrollbar to the horizontal status filter list. This is less common in introductory lessons and allows customization (e.g., thumb color, thickness).
- Scrollbar: Used in the dialog to add a scrollbar to the product list when it exceeds a certain height (more than 3 products). This is an advanced widget for managing scrollable content visibility.

Libraries and Plugins Used

- flutter/material.dart: Provides Material Design widgets and framework for building the UI.
- provider: A state management library used to manage and share the OrdersManager state across the app.
- pocketbase: A plugin for interacting with the PocketBase backend, handling order data retrieval and updates.
- intl: Used for formatting dates (e.g., DateFormat in OrderItemCard).
- flutter/foundation.dart: Provides ChangeNotifier for state management in OrdersManager.

State Management

This feature uses a shared state management solution with the Provider package and a ChangeNotifier-based OrdersManager class.

Code Architecture:

- Model Layer: CartItem and OrderItem define data structures for cart items and orders, respectively. User (imported) represents the customer.
- Service Layer: OrderService handles communication with PocketBase (e.g., fetchAllOrders, updateOrder), abstracting backend interactions.
- Manager Layer: OrdersManager bridges the service and UI, managing state and business logic (e.g., updating order status).
- UI Layer: OrdersScreen and OrderItemCard render the UI and interact with OrdersManager for data operations.

How It Works:

- OrdersManager extends ChangeNotifier, managing a list of orders (_orders). It notifies listeners (e.g., UI widgets) when the state changes (e.g., orders are fetched or updated).
- The Provider package makes OrdersManager available globally, allowing OrdersScreen to access it via Provider.of<OrdersManager>(context) or context.read<OrdersManager>().
- When data operations occur (e.g., fetching orders, updating status), OrdersManager updates _orders and calls notifyListeners(), triggering UI rebuilds. The Consumer widget in OrdersScreen listens to these changes to rebuild the order list.

Data Reading and Storage

This feature reads and stores data remotely using the PocketBase backend.

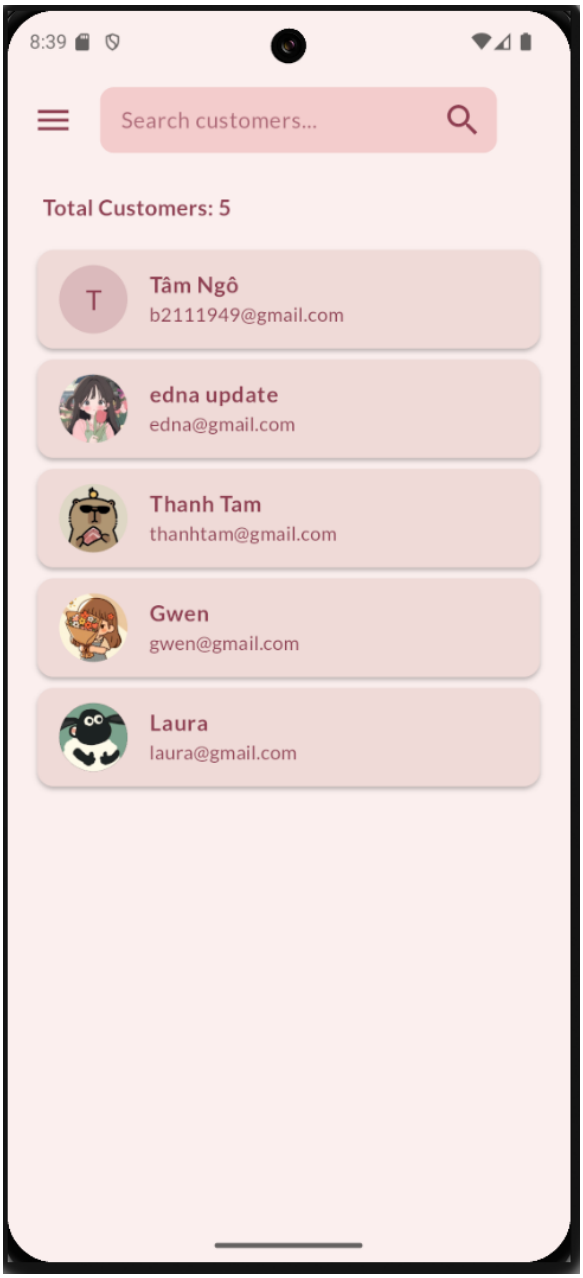
- **Data Table Structure (PocketBase Collections):**
  - orders Collection:
    - id (String): Unique identifier for the order (auto-generated by PocketBase).
    - amount (Double): Total order amount (required).
    - dateTime (String): ISO 8601 timestamp of the order (required).
    - products (List<String>): List of cart item IDs (references to carts collection).
    - status (String): Order status (e.g., "pending," "confirmed," "completed," "canceled").
    - userId (String): ID of the user who placed the order (references users collection).
  - carts Collection (Referenced by orders):
    - id (String): Unique identifier for the cart item.
    - productId (String): ID of the product (references products collection).
    - title (String): Product title (likely copied from products for convenience).
    - price (Double): Product price at the time of order.
    - quantity (Integer): Number of items ordered.
    - status (String): Cart item status (e.g., "pending," "error").
    - imageUrl (String, optional): URL of the product image.
  - products Collection (Referenced by carts):
    - id (String): Unique identifier for the product.
    - title (String): Product name.

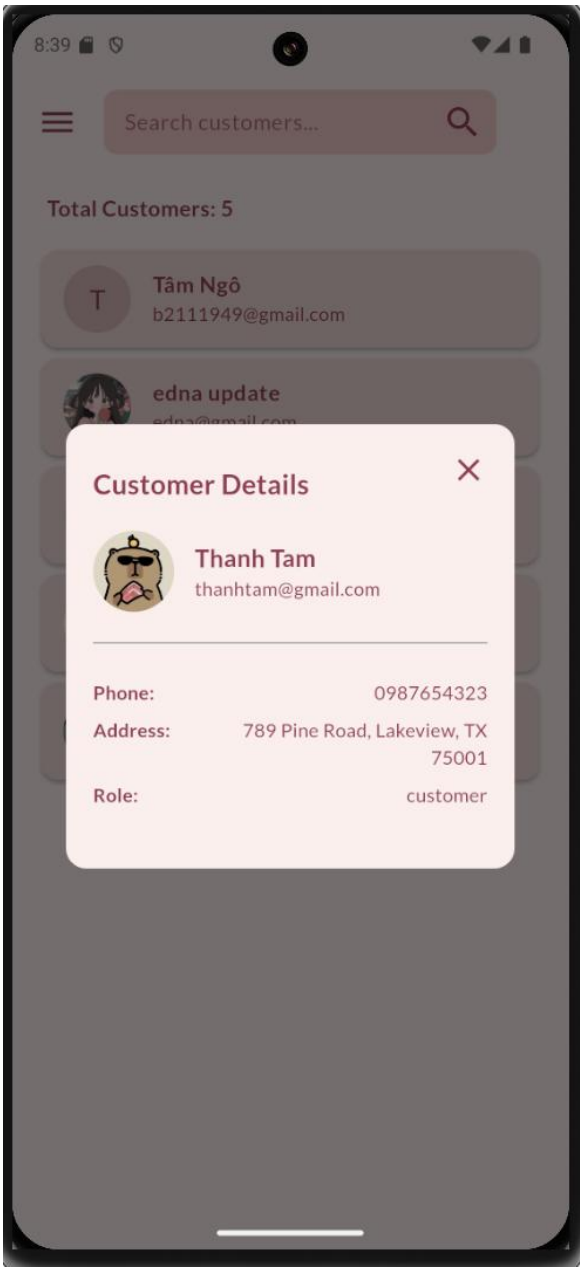
        - price (Double): Product price.
        - (Other fields like description, stockQuantity, etc., as seen in previous code.)
    - users Collection (Referenced by orders):
        - id (String): Unique identifier for the user.
        - username (String): User’s username.
        - phone (String): User’s phone number.
        - (Other fields like email, urole, etc., as seen in previous code.)
- PocketBase API Interaction:
    - Fetch All Orders:
        - Call: pb.collection('orders').getFullList()
        - Input: None.
        - Output: A list of RecordModel objects for orders. Additional calls to carts, products, and users collections fetch related data (cart items, products, users), which are combined into OrderItem instances.
    - Update Order:
        - Call: pb.collection('orders').update(id, body: {...})
        - Input: Order id and a map with updated fields (e.g., amount, dateTime, status).
        - Output: A RecordModel object, converted to an updated OrderItem.

4. Feature / Application page 4: View Customer with admin role

Description: This feature handles user login and signup functionalities. The page allows users to either log in to an existing account or register a new account by providing credentials such as email, password, username, and phone number. It is a common entry point for applications requiring user authentication, ensuring that only authorized users can access the app's features. The implementation integrates with a backend service (PocketBase) for user management and authentication.

Screenshots:





Implementation details:

- ***Widgets Used in this feature***
  - Scaffold: Provides the basic structure (body, drawer).
  - SafeArea: Ensures content fits within the device's safe area.
  - Container: Used for styling and layout (e.g., search bar background, borders).
  - Column: Arranges widgets vertically (e.g., search bar, customer list).
  - Row: Arranges widgets horizontally (e.g., search bar with icon, avatar with details).
  - Text: Displays static text (e.g., customer details, labels).
  - TextField: Captures search input.
  - IconButton: Opens the drawer or closes the dialog.
  - Icon: Displays icons (e.g., menu, search, close).
  - ListView: Displays customers vertically.
  - Card: Wraps each customer entry for a card-like appearance.
  - Padding: Adds spacing around widgets.
  - SizedBox: Controls spacing between widgets.
  - Center: Centers content (e.g., loading indicator or "No customers found" message).
  - FutureBuilder: Manages asynchronous user fetching and displays loading/error states.
  - Consumer: Rebuilds the UI based on UserManager state changes.
  - GestureDetector: Handles taps on customer cards to show details.
  - Dialog: Shows customer details in a popup.
  - SingleChildScrollView: Ensures scrollable content in the dialog.
  - Divider: Separates sections in the dialog.
  - CircleAvatar: Displays user avatars or initials.
  - Expanded: Ensures flexible layout (e.g., search field, customer list).

- ListTile: Structures customer entries with leading avatar, title, and subtitle.
- Stack: Positions the close button in the dialog.
- Positioned: Places the close button at the top-right of the dialog.
- CircularProgressIndicator: Shows loading states.

Libraries and Plugins Used

- flutter/material.dart: Provides Material Design widgets and framework for building the UI.
- provider: A state management library used to manage and share the UserManager state across the app.
- pocketbase: A plugin for interacting with the PocketBase backend, handling user data retrieval.
- flutter/foundation.dart: Provides ChangeNotifier for state management in UserManager.

State Management

This feature uses a shared state management solution with the Provider package and a ChangeNotifier-based UserManager class.

Code Architecture:

- Model Layer: User class defines the data structure for a user (e.g., id, username, email).
- Service Layer: UserService handles communication with PocketBase (e.g., fetchUsers), abstracting backend interactions.
- Manager Layer: UserManager bridges the service and UI, managing state and business logic (e.g., updating the user list).
- UI Layer: UsersScreen renders the UI and interacts with UserManager for data operations.

How It Works:

- UserManager extends ChangeNotifier, managing a list of users (_users). It notifies listeners (e.g., UI widgets) when the state changes (e.g., users are fetched).
- The Provider package makes UserManager available globally, allowing UsersScreen to access it via Provider.of<UserManager>(context) or context.read<UserManager>().
- When data operations occur (e.g., fetching users), UserManager updates _users and calls notifyListeners(), triggering UI rebuilds. The Consumer widget in UsersScreen listens to these changes to rebuild the customer list.

Data Reading and Storage

This feature reads data remotely using the PocketBase backend. No local storage or data writing (e.g., updating users) is implemented in the provided code.

- Data Table Structure (PocketBase users Collection):
  - id (String): Unique identifier for the user (auto-generated by PocketBase, required).
  - email (String): User's email address (required).
  - username (String): User's chosen username (required).
  - phone (String): User's phone number (required).
  - urole (String): User role (e.g., "customer" or "staff", defaults to "customer").
  - address (String, optional): User's address.
  - avatar (String, optional): Filename of the uploaded avatar image (stored in PocketBase's file storage, with a URL generated via getUrl).
- PocketBase API Interaction (Not a traditional REST API but an SDK):
  - Fetch Users:
    - Call: pb.collection('users').getFullList(filter: "urole='customer'")

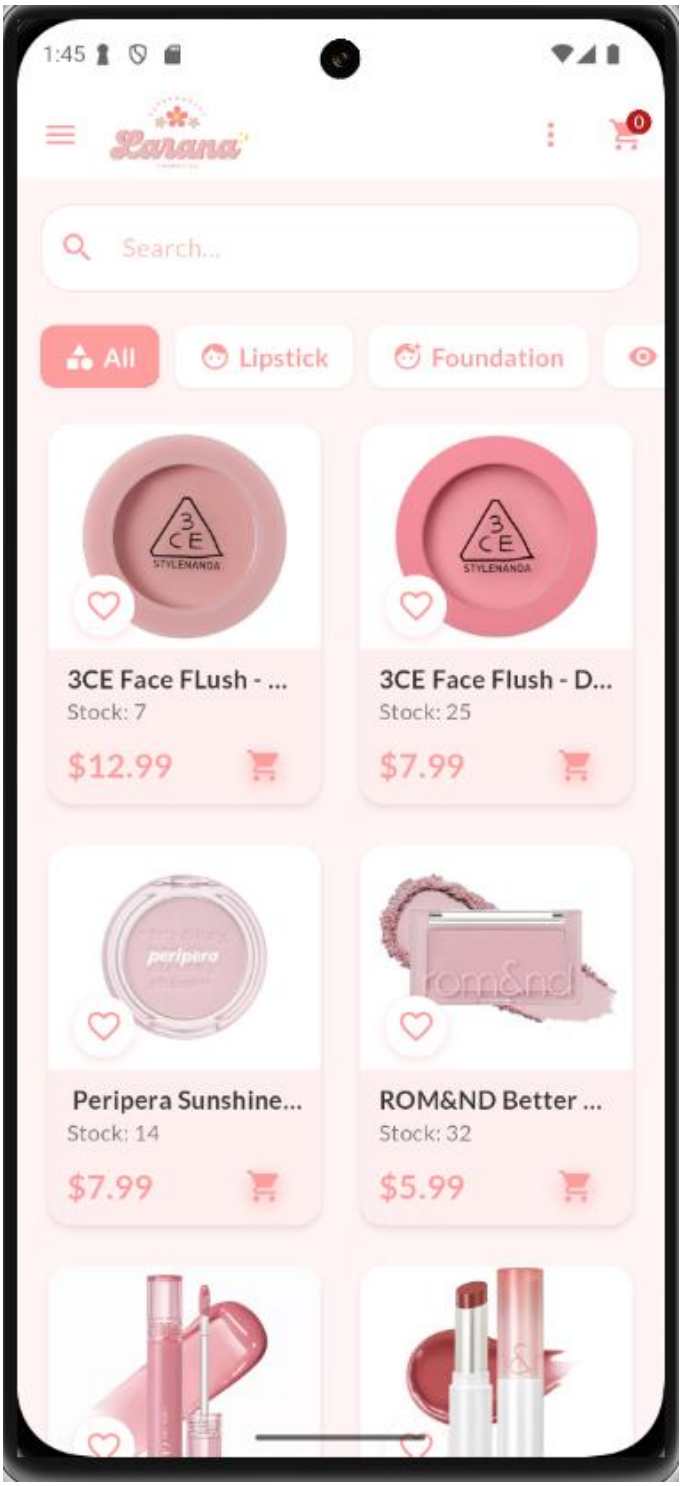
- Input: Optional filter string (e.g., "urole='customer'" or "id='userId' && urole='customer'" if filteredByUser is true).
- Output: A list of RecordModel objects, converted to User instances with avatar URLs added via getUrl.

5. Feature / Application page 5: Home Page – Customer Role

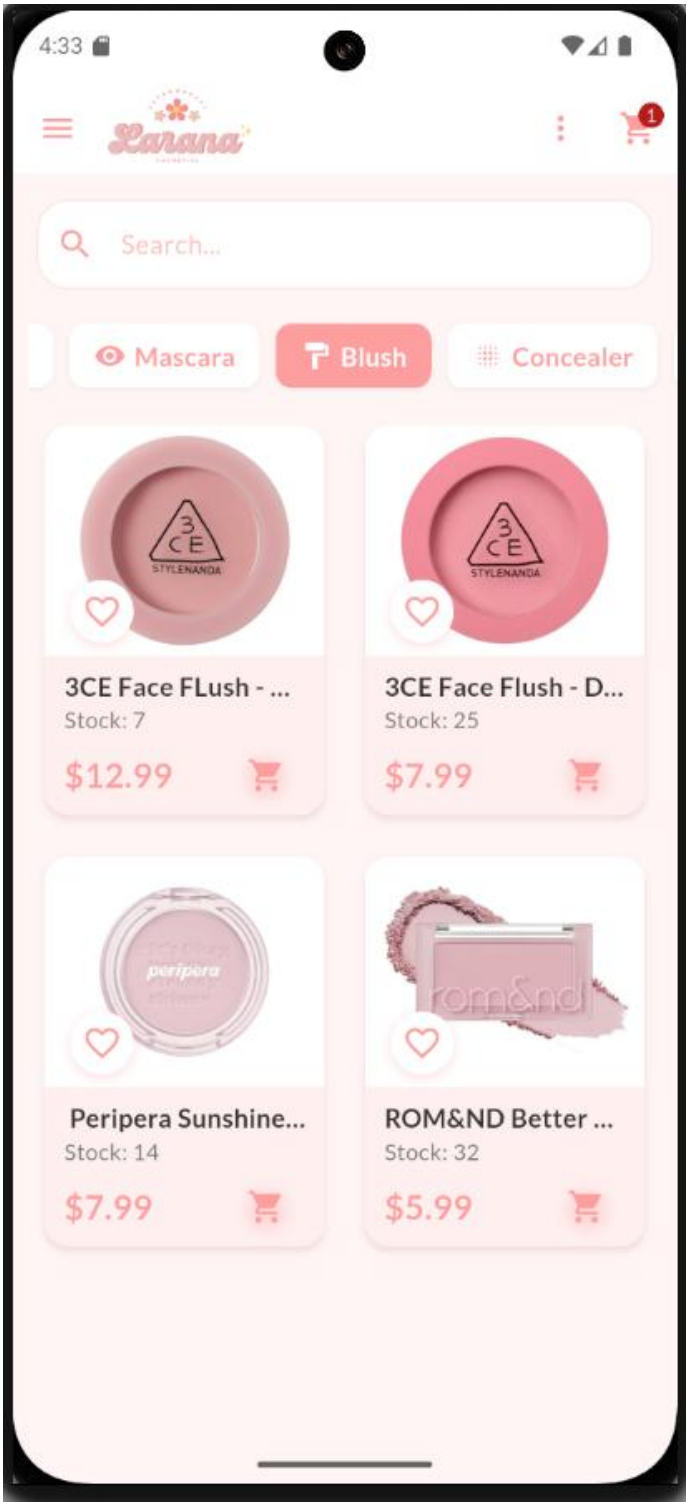
Description: This feature allows users to browse, filter, and interact with a product catalog in an e-commerce application. The UserProductsScreen serves as the main interface where users can view a grid of products, search by product title, filter by categories (e.g., Lipstick, Foundation), toggle between all products and favorite products, and add items to their shopping cart. It integrates with a backend service (PocketBase) for product data management and provides a visually appealing UI with animations for enhanced user experience. The feature supports infinite scrolling to load more products dynamically and includes real-time cart updates.

Screenshots

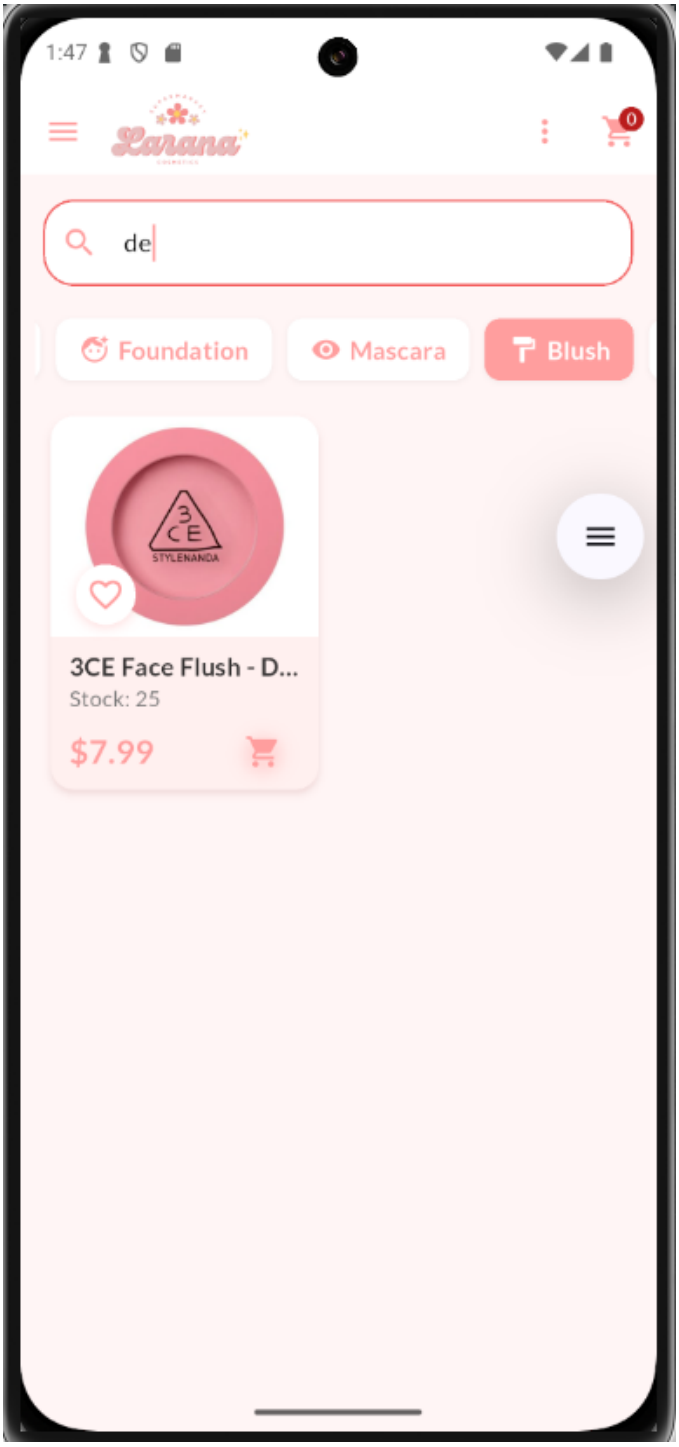
When the user successfully logs in as a customer, the user will be redirected to the home page displaying the product list.



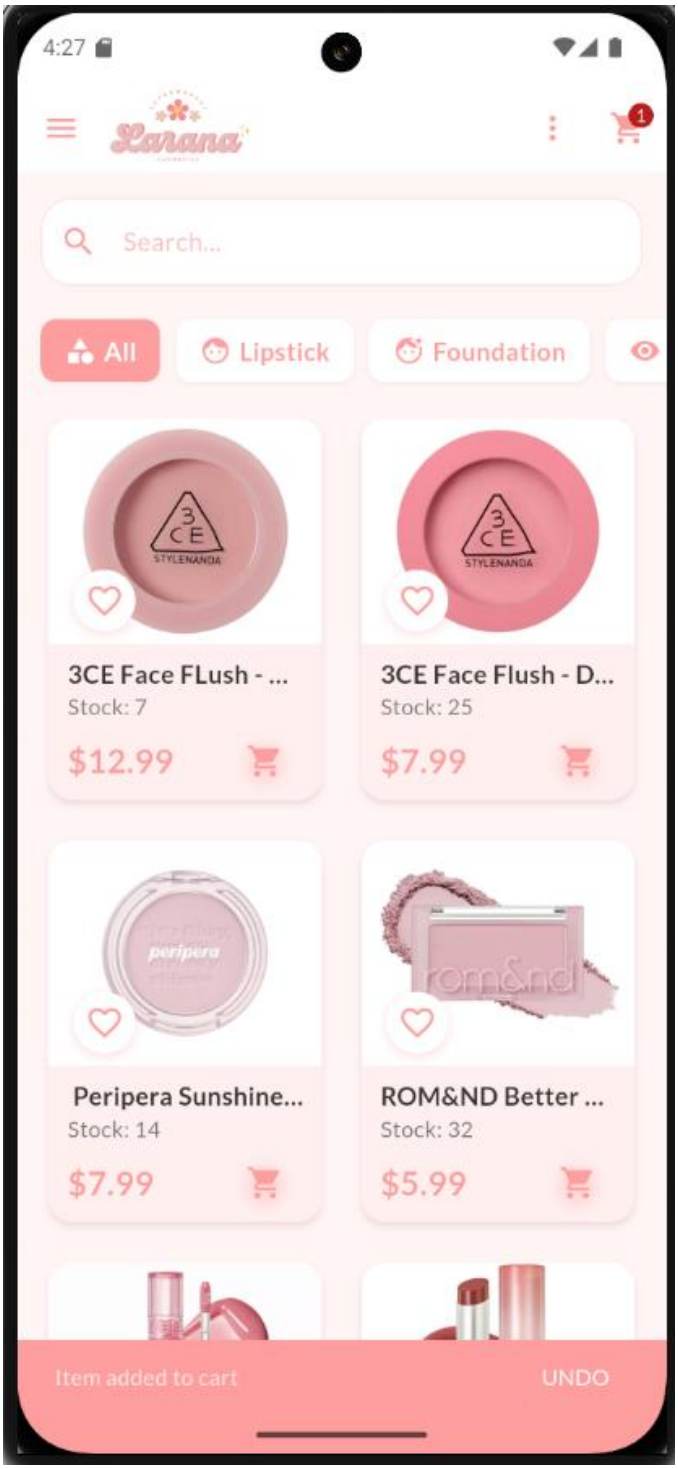
Users can filter the product list by category on the home page.



Users can search for products by product name.



Users can add products to the cart on the home page by clicking the cart button on each product. After successfully adding to the cart, a message "Item added to cart" will appear.



Implementation details:

- ***Widgets Used in this feature***
  - **Scaffold:** Provides the basic app structure with an app bar, body, and drawer.
  - **AppBar:** Displays the app logo, filter menu, and cart button with animations.
  - **SafeArea:** Ensures content fits within the device's safe area boundaries.
  - **Container:** Styles elements like the search bar and product cards.
  - **Column:** Arranges widgets vertically (e.g., search bar, category list, product grid).
  - **Row:** Arranges widgets horizontally (e.g., category items, price, and cart button).
  - **Text:** Displays static text (e.g., product titles, prices, stock quantities).
  - **TextField:** Captures user input for searching products.
  - **IconButton:** Triggers actions like opening the drawer or adding to cart.
  - **Icon:** Displays icons (e.g., search, favorite, cart).
  - **GridView:** Displays products in a 2-column grid layout with infinite scrolling.
  - **Card:** Wraps each product entry for a card-like appearance.
  - **Padding:** Adds spacing around widgets for better layout.
  - **SizedBox:** Controls spacing between widgets (e.g., in category list).
  - **Center:** Centers content (e.g., loading indicator).
  - **FutureBuilder:** Manages asynchronous product fetching and displays loading states.
  - **Consumer:** Rebuilds the UI based on ProductsManager or CartManager state changes.
  - **GestureDetector:** Handles taps on product cards to navigate to details.
  - **AnimatedBuilder:** Animates UI elements (e.g., filter button, cart button).
  - **FadeTransition / ScaleTransition:** Adds fade and scale animations to buttons and products.
  - **ListView:** Displays categories horizontally.
  - **CircularProgressIndicator:** Shows loading states during data fetch or cart actions.

- **Image:** Loads product images from URLs with loading and error handling.
- **Stack:** Positions elements like the favorite button over product images.
- **Positioned:** Places the favorite button at the bottom-left of product images.
- **InkWell:** Provides tappable product cards with ripple effects.
- **Badge:** Displays the cart item count on the cart button.
- **ClipRRect**: Applies rounded corners to product tiles for a polished look
- **GridTile**: Structures each product tile with a child (image) and footer
- **GridTileBar**: Implements the footer bar for each product tile, displaying the title and action buttons
- **ScaffoldMessenger**: Displays a snack bar with an "UNDO" action when adding a product to the cart

Libraries and Plugins Used

- **flutter/material.dart:** Provides Material Design widgets and framework for building the UI.
- **provider:** Manages state across the app using ProductsManager and CartManager.
- **pocketbase:** Interacts with the PocketBase backend for product data retrieval and updates.
- **flutter/foundation.dart:** Provides ChangeNotifier for state management.
- **http:** Supports file uploads for product images in ProductsService.

State Management

This feature uses the Provider package with a ChangeNotifier-based ProductsManager and CartManager for shared state management.

Code Architecture:

- **Model Layer**: Product class defines the product data structure (e.g., pid, title, price).
- **Service Layer:** ProductsService handles communication with PocketBase (e.g., fetchProducts, updateProduct).
- **Manager Layer:** ProductsManager manages product state and business logic (e.g., filtering favorites, updating products).
- **UI Layer:** UserProductsScreen and UserProduct render the UI and interact with managers for data operations.

How It Works:

- ProductsManager extends ChangeNotifier, maintaining a list of products (_items) and notifying listeners when the state changes (e.g., products fetched or updated).
- Provider makes ProductsManager and CartManager globally accessible, allowing widgets like UserProductList and ShoppingCartButton to react to state changes via Consumer.
- FutureBuilder handles the initial product fetch, while Consumer updates the UI dynamically as the product or cart state changes.

Data Reading and Storage

This feature reads data remotely using the PocketBase backend. No local storage or data writing (e.g., updating users) is implemented in the provided code.

- Data Table Structure (PocketBase users Collection):
    - id (String): Unique identifier for the product (auto-generated by PocketBase, required).
    - title (String): Product name (required).
    - description (String): Product description (required).
    - price (Double): Product price (required).
    - featuredImage (String): Filename of the uploaded product image (stored in PocketBase file storage, optional).
    - imageUrl (String): URL of the product image (generated via getUrl, optional).
    - isFavorite (Bool): Indicates if the product is favorited (defaults to false).
    - stockQuantity (Int): Available stock (required).
    - category (String): Product category (e.g., "Lipstick", required).
    - userId (String): ID of the user who created the product (required).
    - locked (Bool): Indicates if the product is locked (defaults to false).

PocketBase API Interaction:

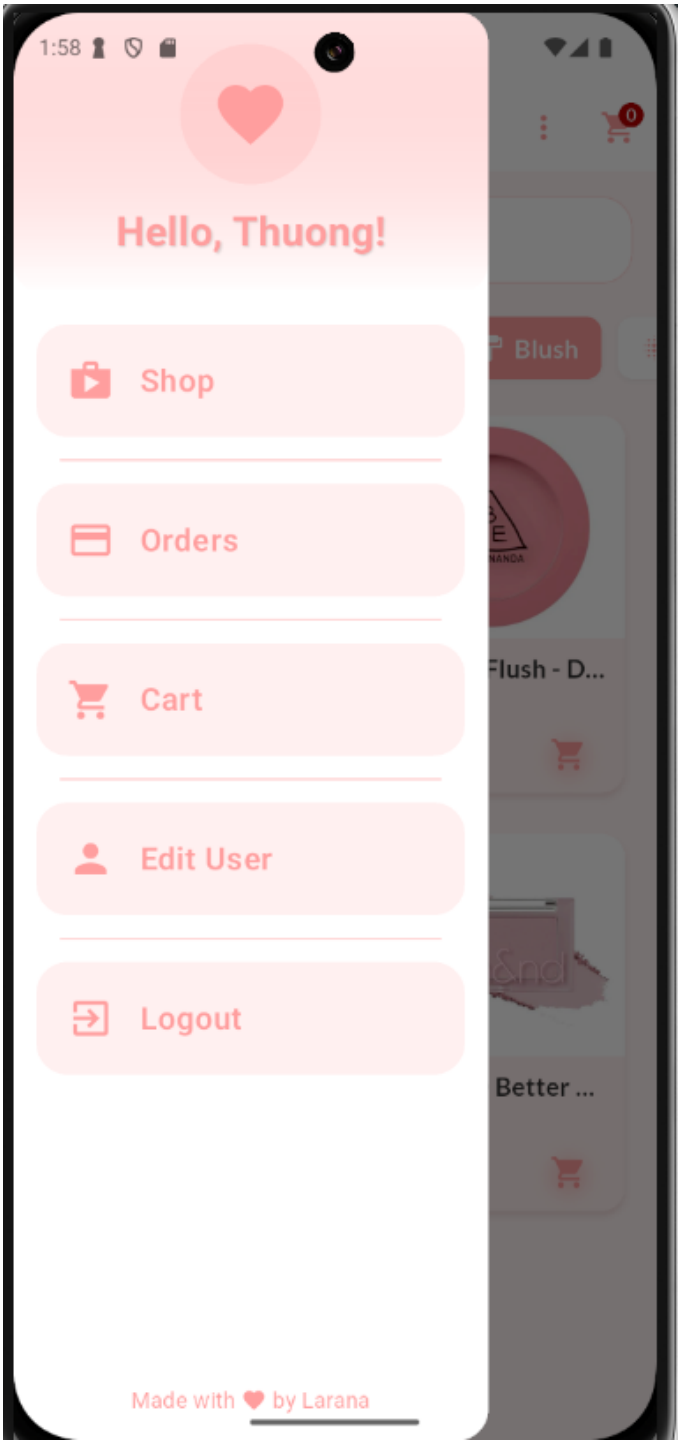
- **Fetch Products:**
    - Call: pb.collection('products').getFullList(filter: "userId='$userId'")
    - Input: Optional filter string (e.g., userId for user-specific products or category for filtering).
    - Output: List of RecordModel objects converted to Product instances with image URLs.
- **Update Product:**
    - Call: pb.collection('products').update(product.pid, body: {...})
    - Input: Product data and optional image file.
    - Output: Updated Product instance with new image URL if applicable.

6. Feature / Application page 6: View/Update/Delete Information – Customer Role

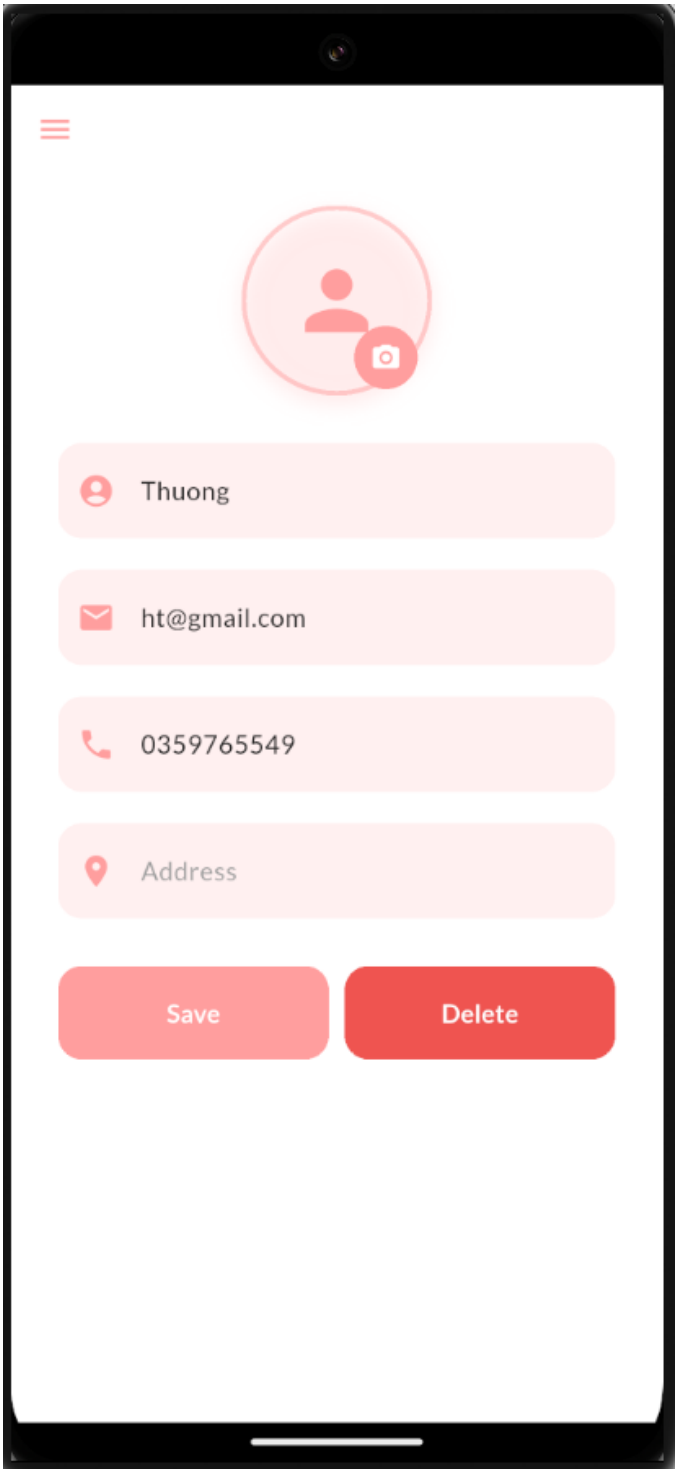
Description: This feature allows users with the "customer" role to view, update, and delete their personal information in the application. The EditUserScreen provides a form where users can edit their profile details such as username, phone number, and address, as well as upload or update their avatar. The AppDrawer displays the user's current information (e.g., username, avatar) and provides navigation options to other screens like Shop, Orders, Cart, and Logout. Additionally, the feature includes custom dialog utilities (dialog_utils.dart) to display confirmation, success, and error messages, enhancing user feedback during operations like saving or deleting user data. The feature integrates with the PocketBase backend for user data management, ensuring secure and efficient handling of user information, with validation, error handling, and user feedback through dialogs.

Screenshots

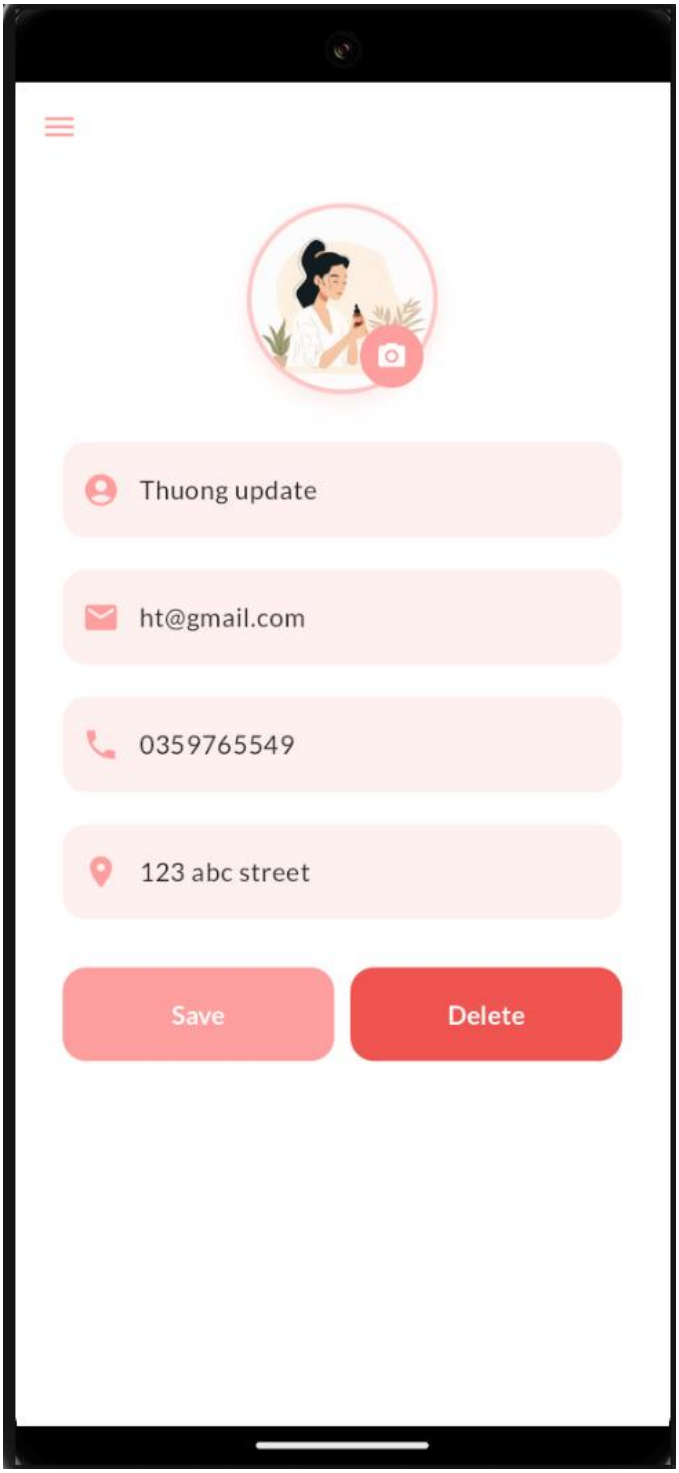
This is the navigation bar interface when users use it to go to the page to view and update user information, in addition, they can go to the interface of other functions.



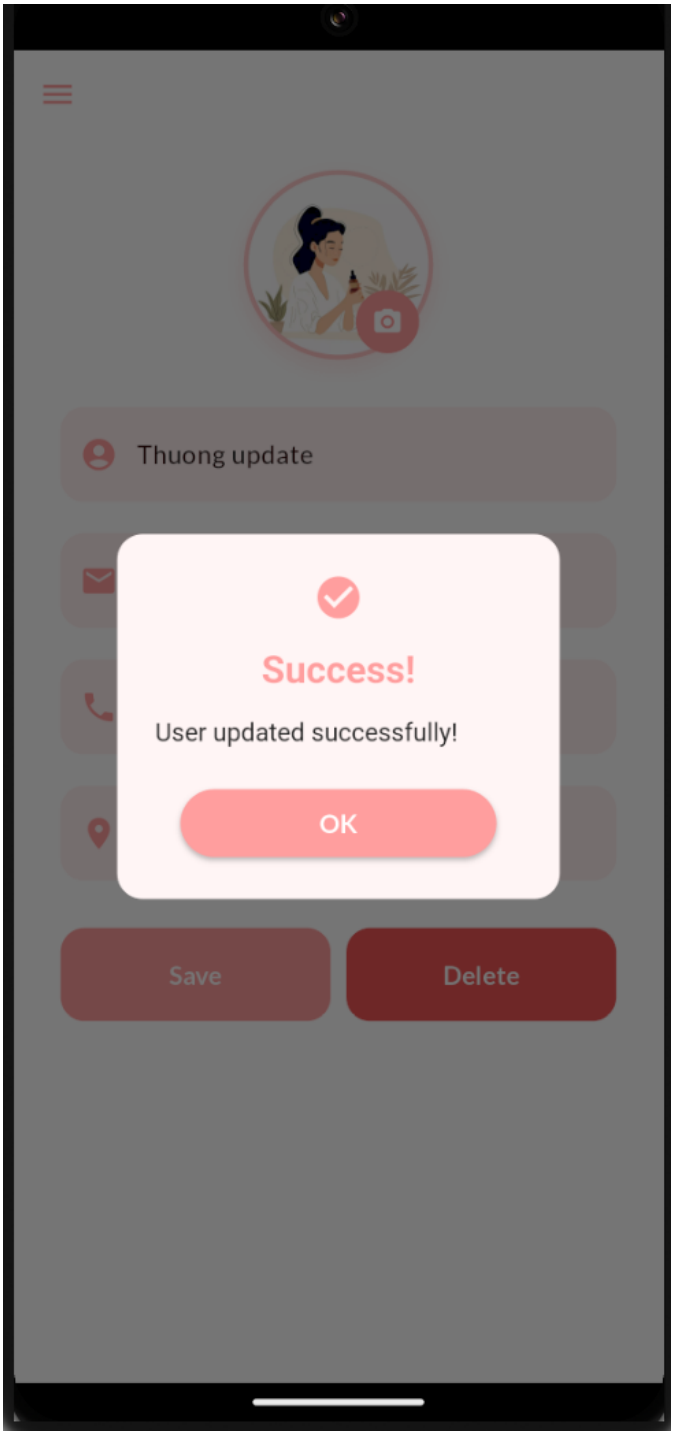
This is the interface when the user views the current information.



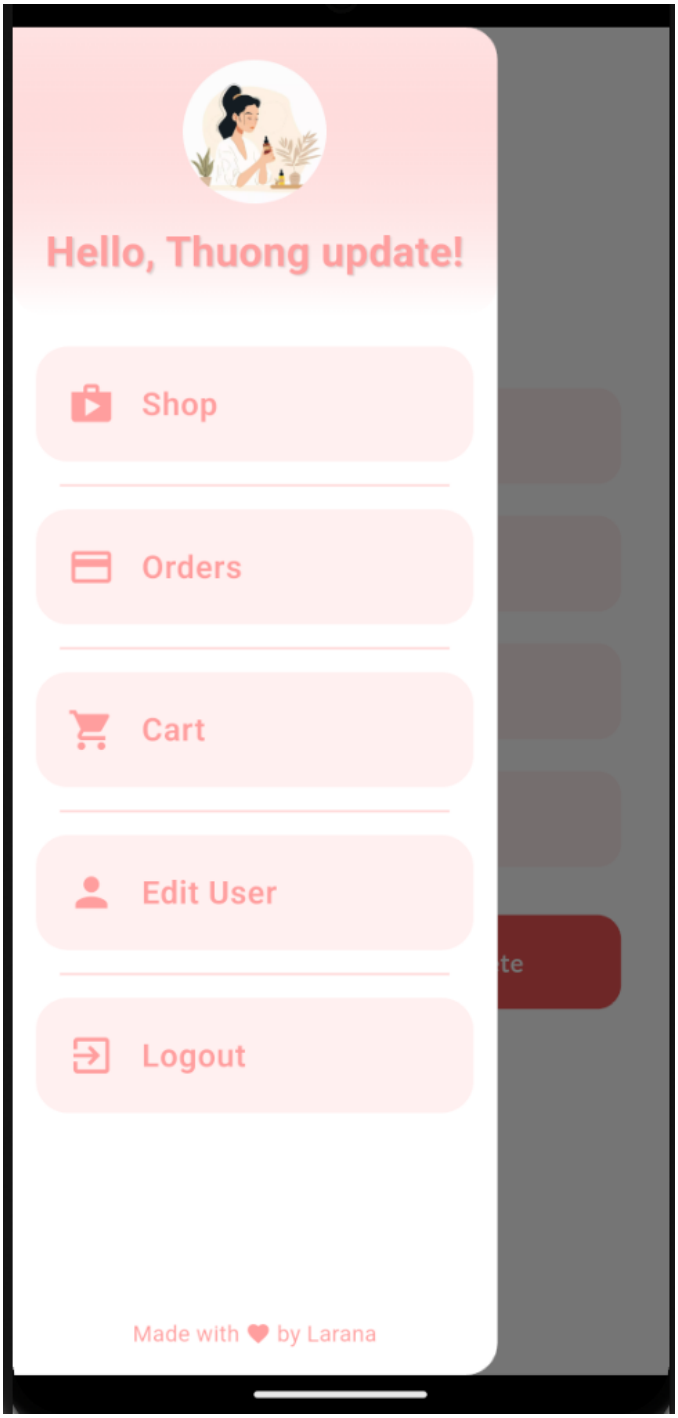
This is the interface when the user filled information.



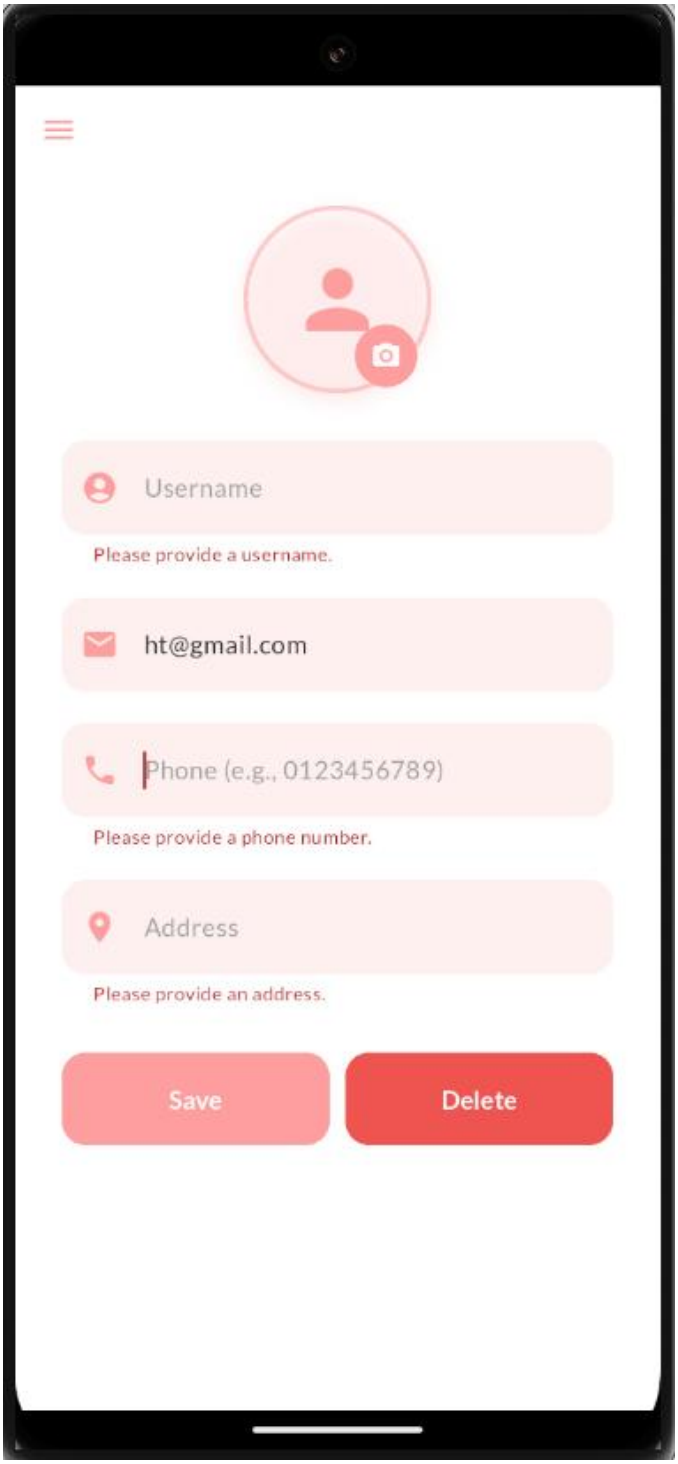
This is the interface when the user update information successfully.



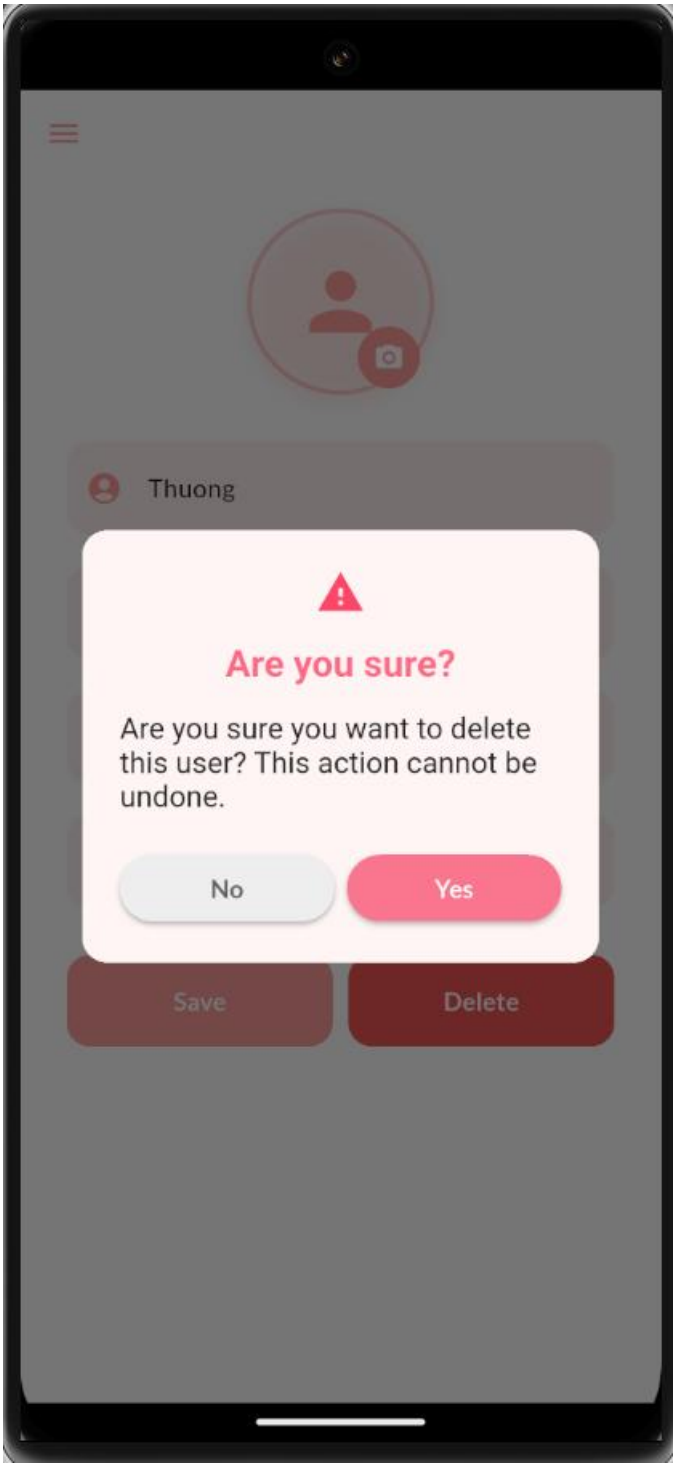
Username and Avatar are updated in navigation bar.



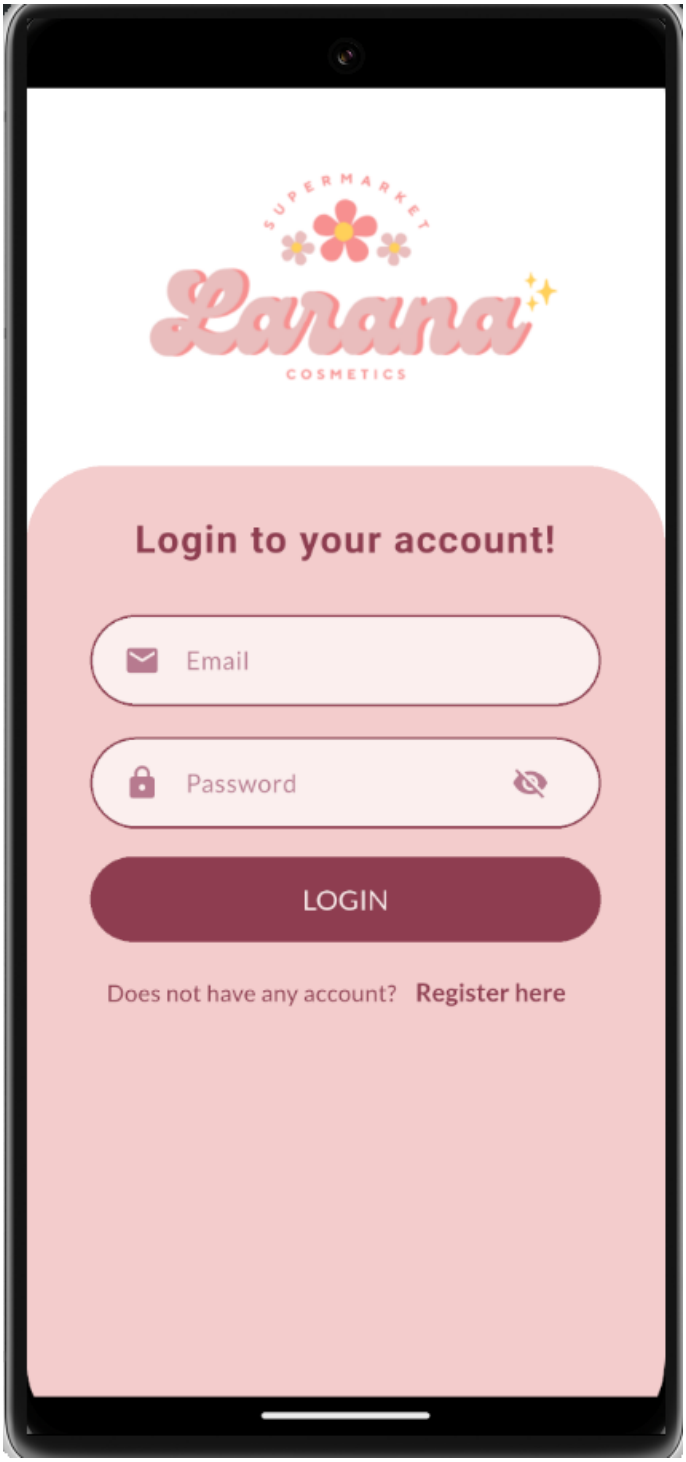
This is the interface when the user enters incorrect or missing information to update.



When clicking delete, there will be a notification for the user.



Upon successful deletion, the user will be returned to the registration and login page.



Implementation details:

- ***Widgets Used in this feature***
  - **Scaffold:** Provides the basic app structure with an app bar, body, and drawer.
  - **AppBar:** Displays a back button for navigation in the EditUserScreen.
  - **Drawer:** Implements the AppDrawer for navigation to other screens.
  - **Container:** Styles elements like the avatar preview, form fields, and dialog content.
  - **Column:** Arranges widgets vertically (e.g., avatar preview, form fields, dialog actions).
  - **Row:** Arranges widgets horizontally (e.g., action buttons in dialogs, footer text in drawer).
  - **Text:** Displays static text (e.g., labels, user info, dialog messages).
  - **TextFormField:** Captures user input for fields like username, phone, and address.
  - **IconButton:** Triggers actions like picking an avatar image or navigating.
  - **Icon:** Displays icons (e.g., camera, person, navigation icons in drawer, dialog icons like warning, error, success).
  - **ListView:** Displays form fields in a scrollable list in EditUserScreen.
  - **Padding:** Adds spacing around widgets for better layout.
  - **SizedBox**: Controls spacing between widgets (e.g., in form fields, dialog actions).
  - **Center:** Centers content (e.g., avatar preview).
  - **CircleAvatar:** Displays the user's avatar or a placeholder icon.
  - **ClipOval:** Ensures the avatar image is circular.
  - **Image:** Loads the user's avatar from a URL or local file with loading/error handling.
  - **Stack:** Positions the camera button over the avatar preview.
  - **Positioned:** Places the camera button at the bottom-right of the avatar.
  - **Form:** Manages form validation and submission in EditUserScreen.

- **ElevatedButton:** Provides buttons for saving, deleting user data, and dialog actions (e.g., "Yes", "No", "Close", "OK").
- **CircularProgressIndicator:** Shows loading states during data fetch or updates.
- **Divider:** Separates menu items in the AppDrawer**.**
- **ListTile:** Structures navigation items in the AppDrawer.
- **ScaleTransition:** Animates the user info section in the AppDrawer.
- **AnimatedContainer:** Adds animations to menu items in the AppDrawer.
- **ScaffoldMessenger:** Displays success/error messages via snack bars.
- **AlertDialog:** Displays confirmation, success, and error dialogs with custom styling (via showConfirmDialog, showSuccessDialog, showErrorDialog).
- **Expanded:** Ensures flexible layout for dialog action buttons (e.g., "Yes" and "No" in confirmation dialog).

Libraries and Plugins Used

- **flutter/material.dart: Provides Material Design widgets and framework for building the UI.**
- **provider: Manages state across the app using UsersManager and AuthManager.**
- **pocketbase: Interacts with the PocketBase backend for user data operations.**
- **http: Supports file uploads for avatars in UsersService.**
- **image_picker: Allows users to pick an avatar image from the gallery.**
- **flutter/foundation.dart: Provides ChangeNotifier for state management.**
- **path: Extracts file names for avatar uploads.**

State Management

This feature uses the Provider package with a ChangeNotifier-based UsersManager for shared state management.

Code Architecture:

- **Model Layer:** User class defines the user data structure (e.g., id, email, username).
- **Service Layer:** UsersService handles communication with PocketBase (e.g., fetchUser, updateUser).
- **Manager Layer:** UsersManager manages user state and business logic (e.g., updating the current user).
- **UI Layer:** EditUserScreen and AppDrawer render the UI and interact with UsersManager for data operations. Dialog utilities (dialog_utils.dart) provide reusable feedback mechanisms.

How It Works:

- UsersManager extends ChangeNotifier, maintaining the current user (_currentUser) and notifying listeners when the state changes (e.g., user updated or deleted).
- Provider makes UsersManager globally accessible, allowing widgets like AppDrawer to react to state changes.
- When data operations occur (e.g., fetching or updating user data), UsersManager updates _currentUser and calls notifyListeners(), triggering UI rebuilds.
- Dialog utilities (showConfirmDialog, showSuccessDialog, showErrorDialog) are stateless and used to provide user feedback during operations like saving or deleting.

Data Reading and Storage

This feature reads and writes user data remotely using the PocketBase backend. No local storage is implemented in the provided code.

- Data Table Structure (PocketBase users Collection):
  - id (String): Unique identifier for the user (auto-generated by PocketBase, required, non-empty).
  - password (String): User’s password (required, hidden, non-empty).
  - tokenKey (String): Authentication token key (required, hidden, non-empty).
  - email (String): User’s email address (required, non-empty).
  - emailVisibility (Bool): Indicates if the email is visible (optional).
  - verified (Bool): Indicates if the user’s email is verified (optional).
  - avatar (File): User’s avatar image (optional, single file).
  - username (String): User’s chosen username (required).
  - phone (String): User’s phone number (required).
  - address (String): User’s address (optional).
  - urole (Select): User role (e.g., "customer" or "staff", required, single selection).

PocketBase API Interaction:

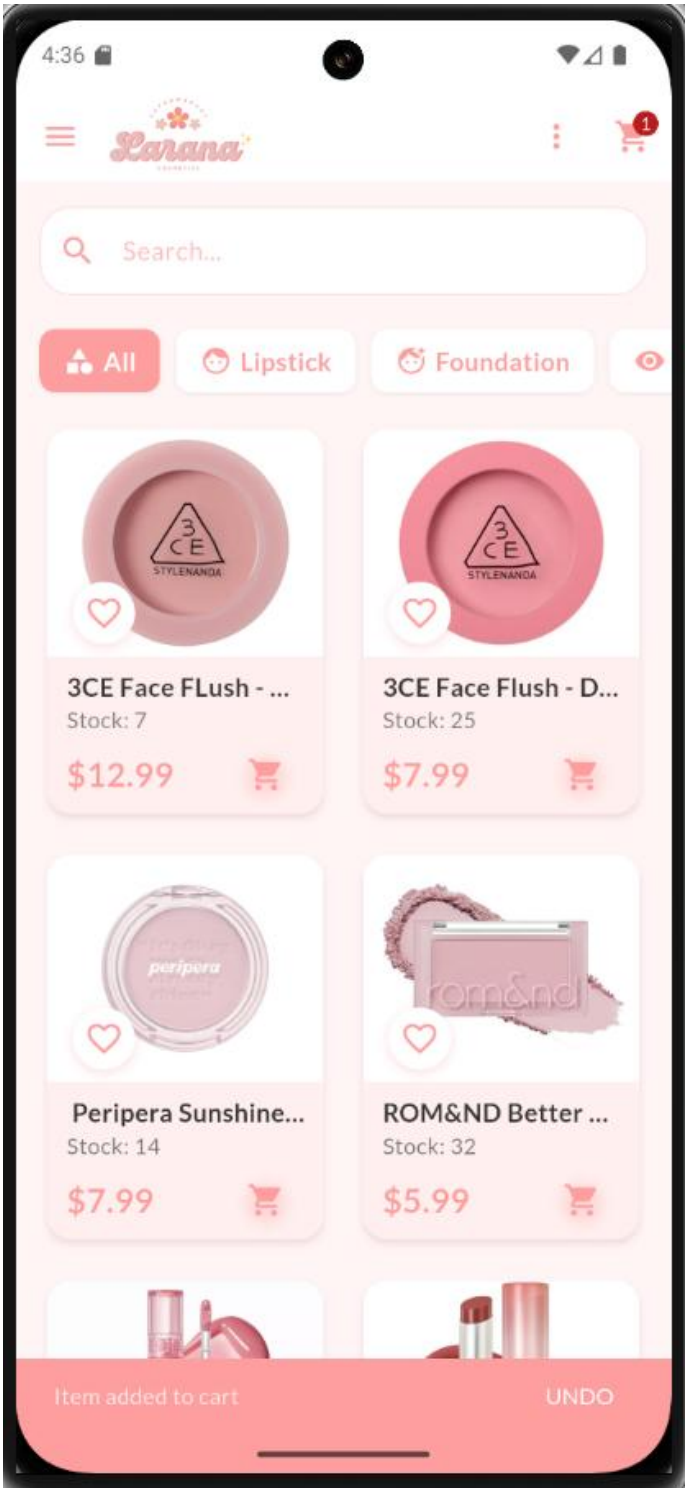
- **Fetch User:**
  - **Call:** pb.collection('users').getOne(userId)
  - **Input:** User ID from the authenticated session.
  - **Output:** A RecordModel object converted to a User instance with avatar URL.
- **Update User:**
  - **Call:** pb.collection('users').update(userId, body: {...}, files: [...])
  - **Input:** Updated user data and optional avatar file.
  - **Output:** Updated User instance with new avatar URL if applicable.
- **Add User:**
  - **Call:** pb.collection('users').create(body: {...}, files: [...])
  - **Input:** New user data and optional avatar file.
  - **Output:** Created User instance with avatar URL.
- **Delete User:**
  - **Call:** pb.collection('users').delete(userId)
  - **Input:** User ID from the authenticated session.
  - **Output:** Boolean indicating success or failure.

7. Feature / Application page 7: CRUD Carts - Manage Cart - Customer Role

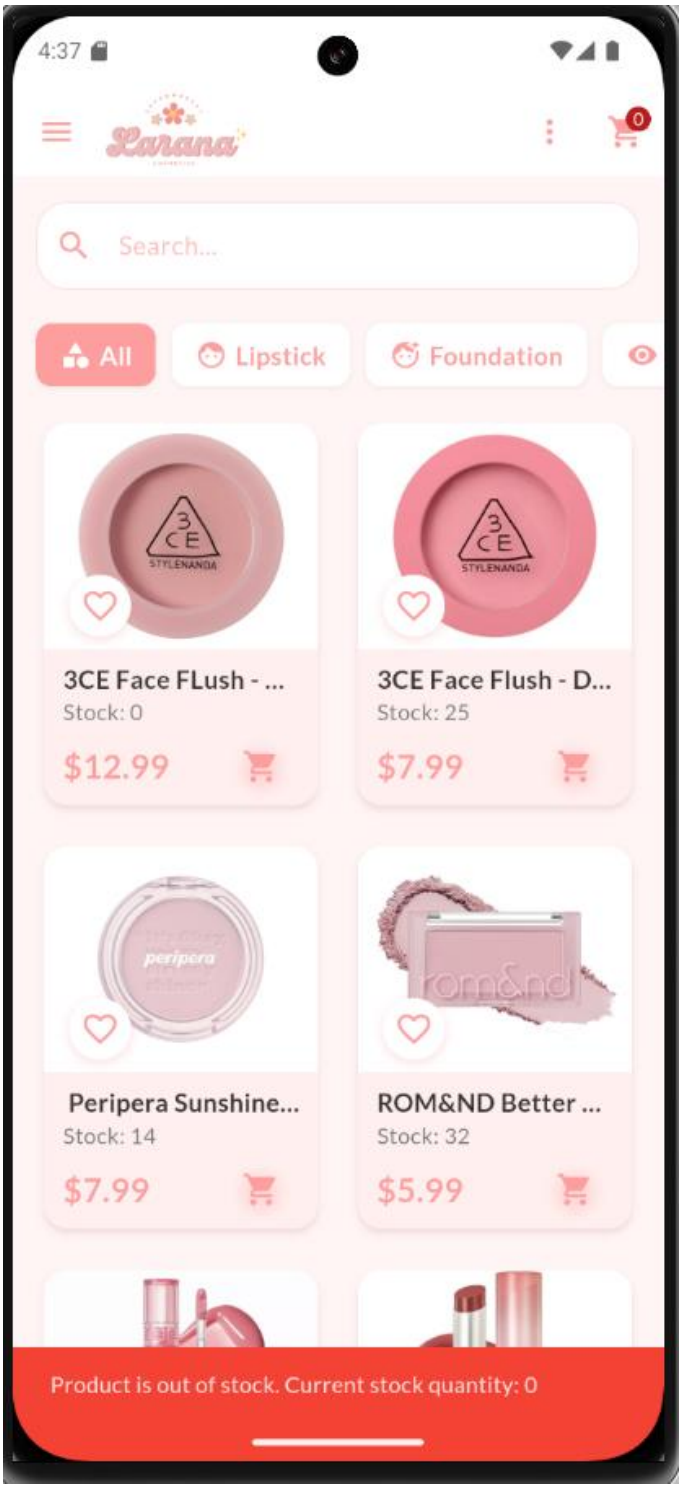
Description: This feature enables users with the "customer" role to perform CRUD operations on their cart. Users can add products to the cart (Create), view their cart items (Read), update the quantity of items in the cart (Update), and remove items from the cart (Delete). The cart page displays a list of cart items with details like product image, title, price, and quantity, along with a total amount and an option to place an order. The feature includes stock validation, user feedback through snack bars and dialogs, and smooth animations for navigation and interactions. It integrates with state management for cart and order data, using a backend for data operations.

Screenshots

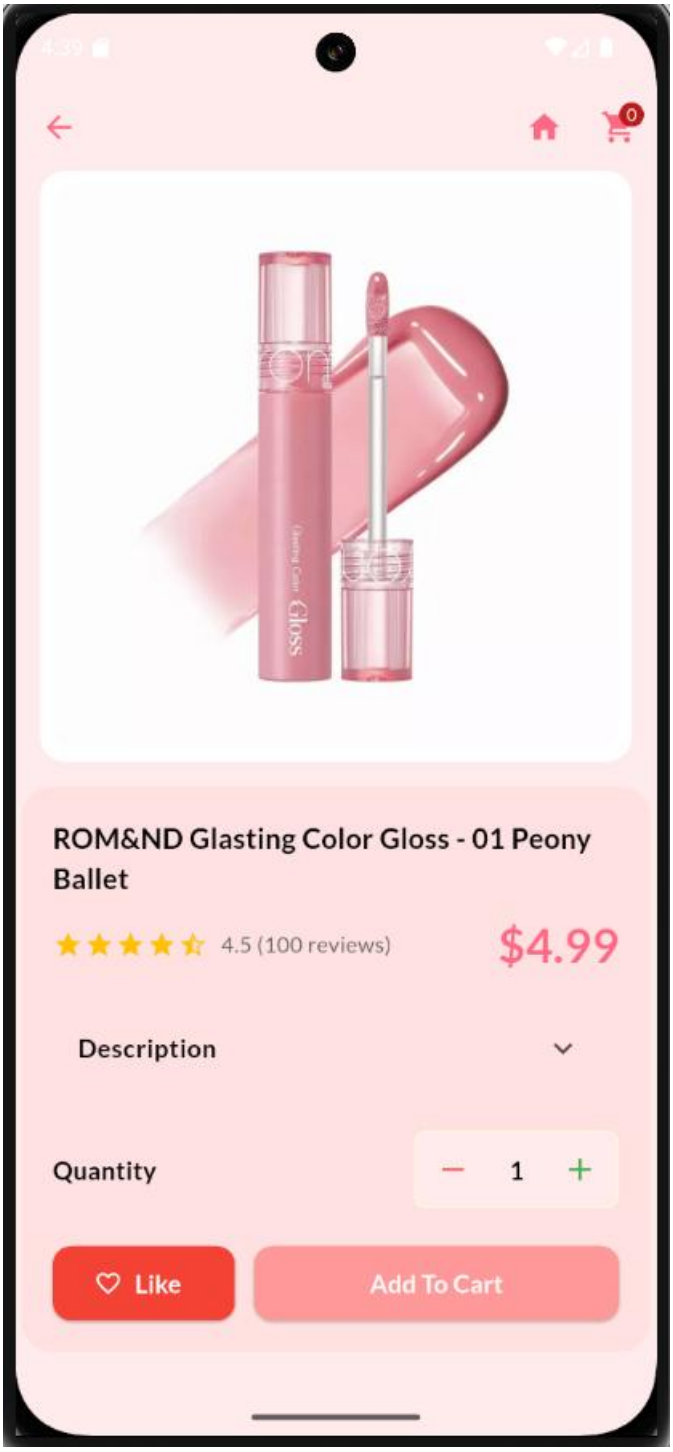
When a user successfully adds a product to the cart on the home page:



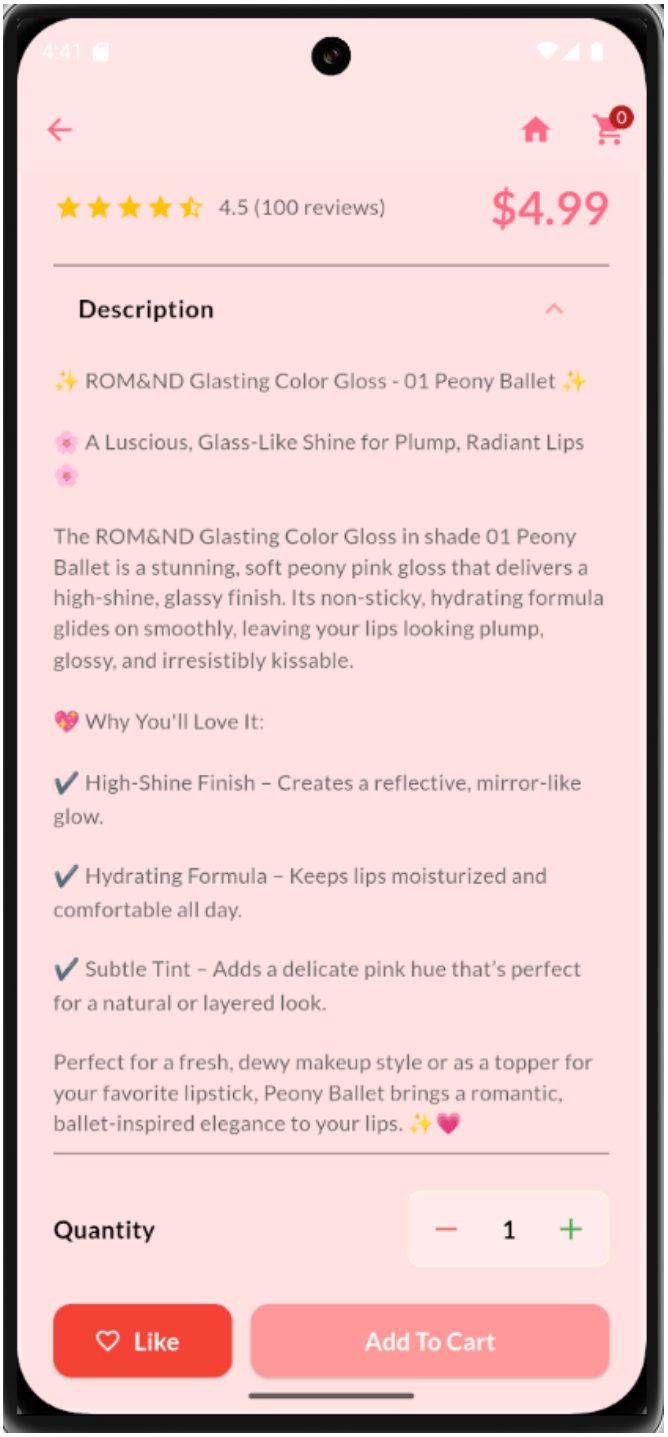
When the user adds an out of stock product to the shopping cart, the system will notify the user that the product is out of stock.



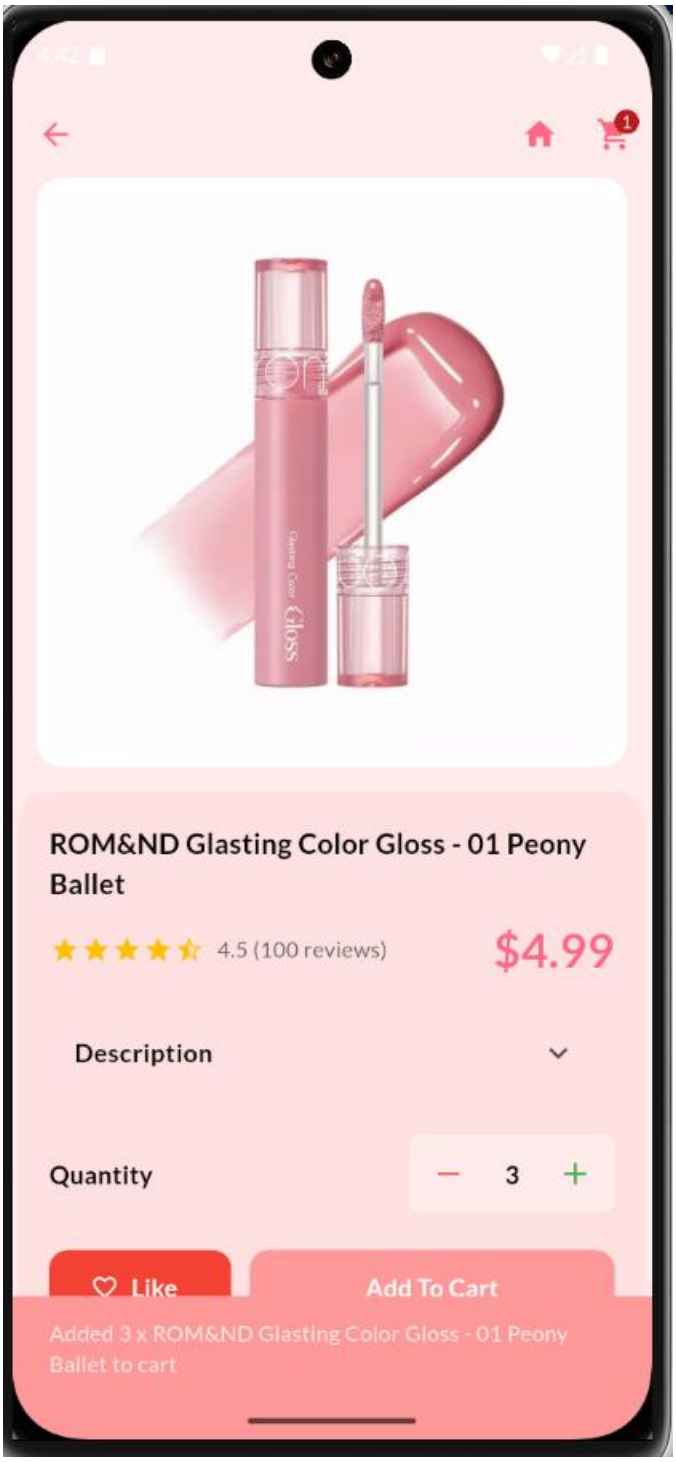
This is the product detail interface when the user clicks on any product on the home page.



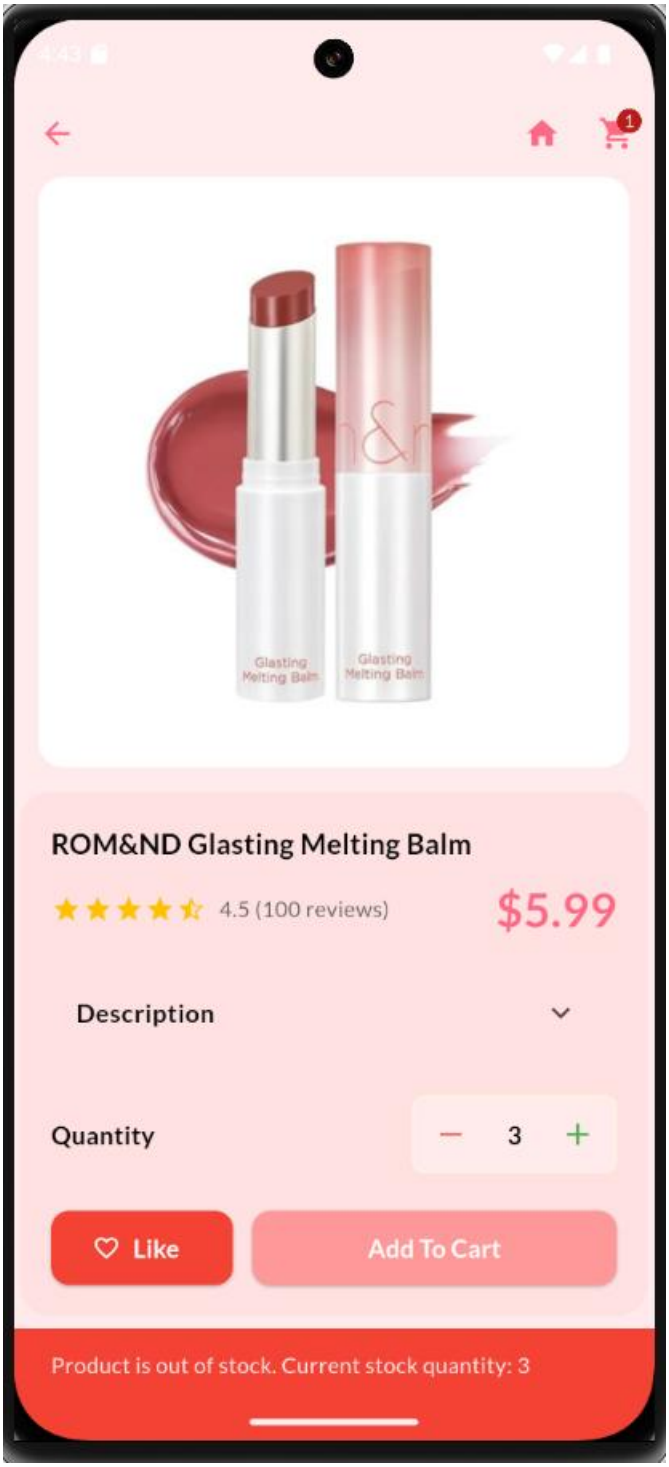
When the user clicks on the add info button in the description section



When user click add more products by clicking plus button and click add to cart successfully

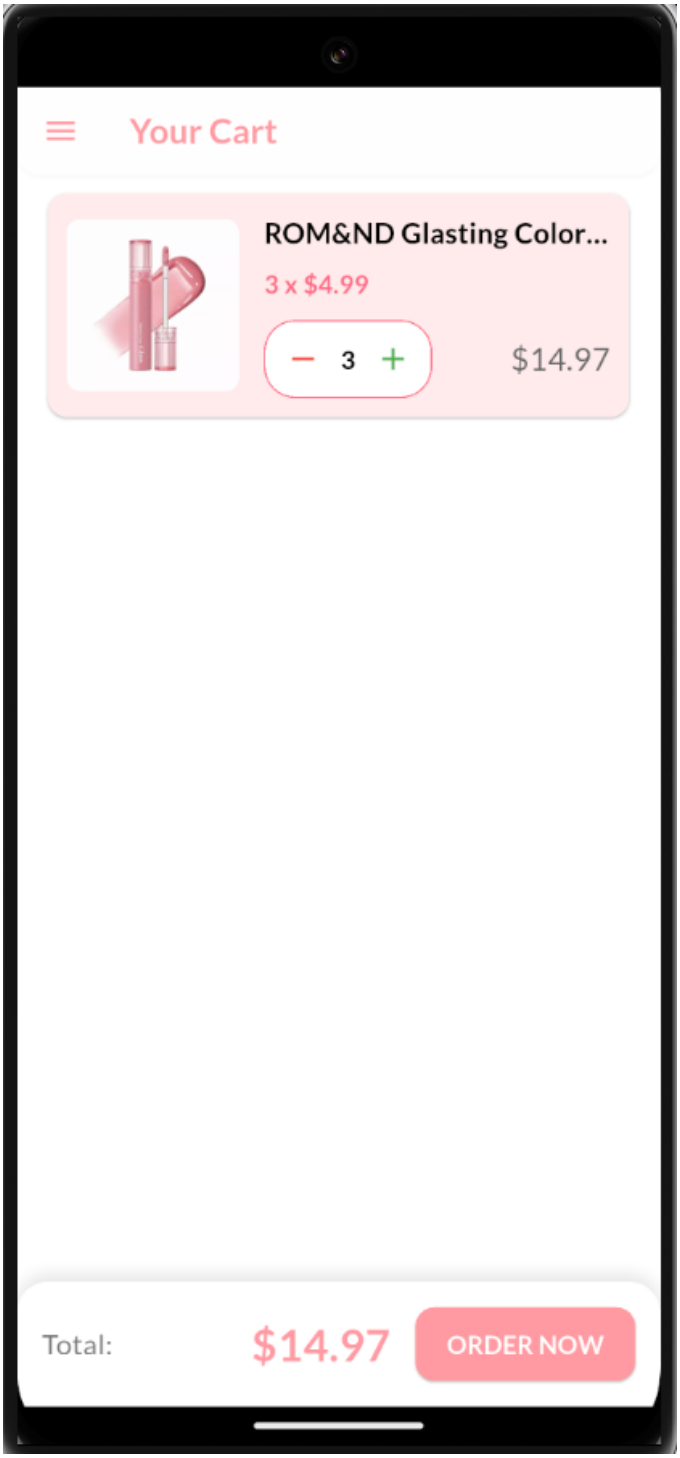


When the user clicks to add a product but exceeds the remaining quantity, the system will warn the user about the product quantity and will not increase the quantity when the user clicks the add product button multiple times.

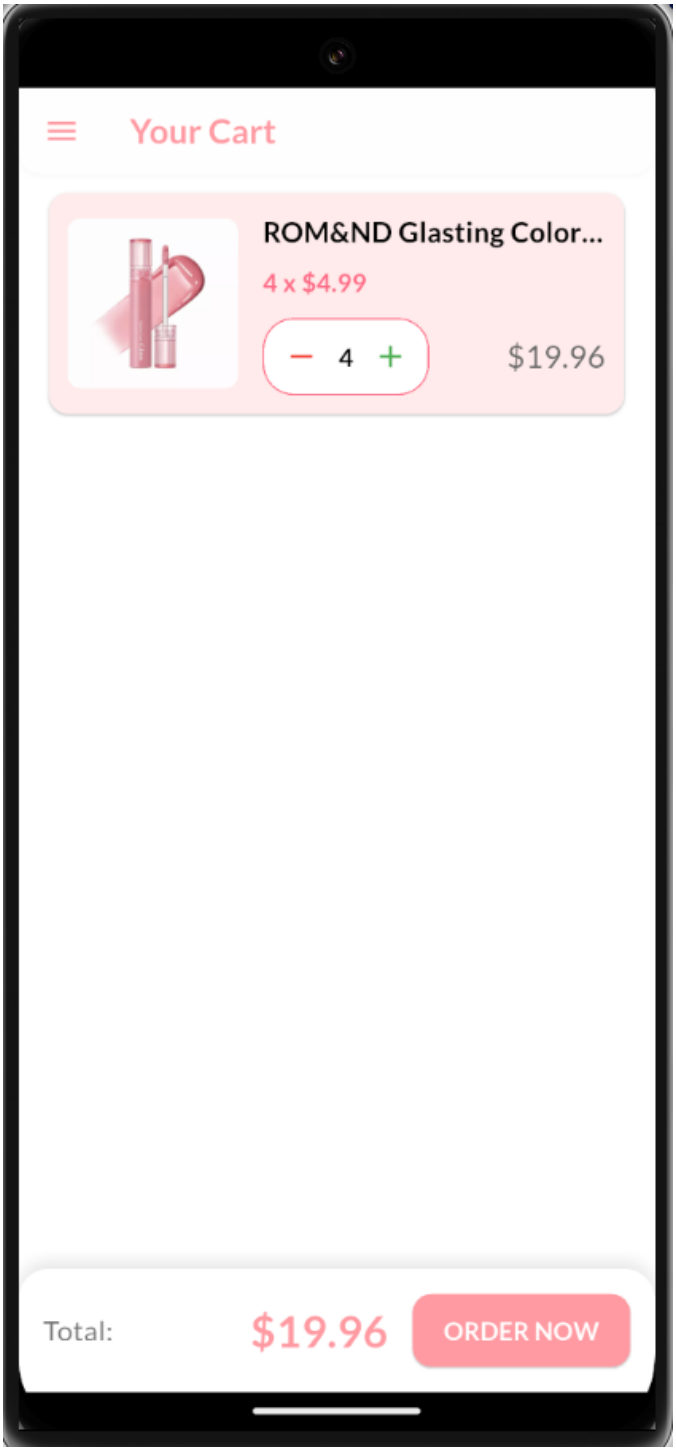


When the product is out of stock, the customer will not be able to add the product to the cart.

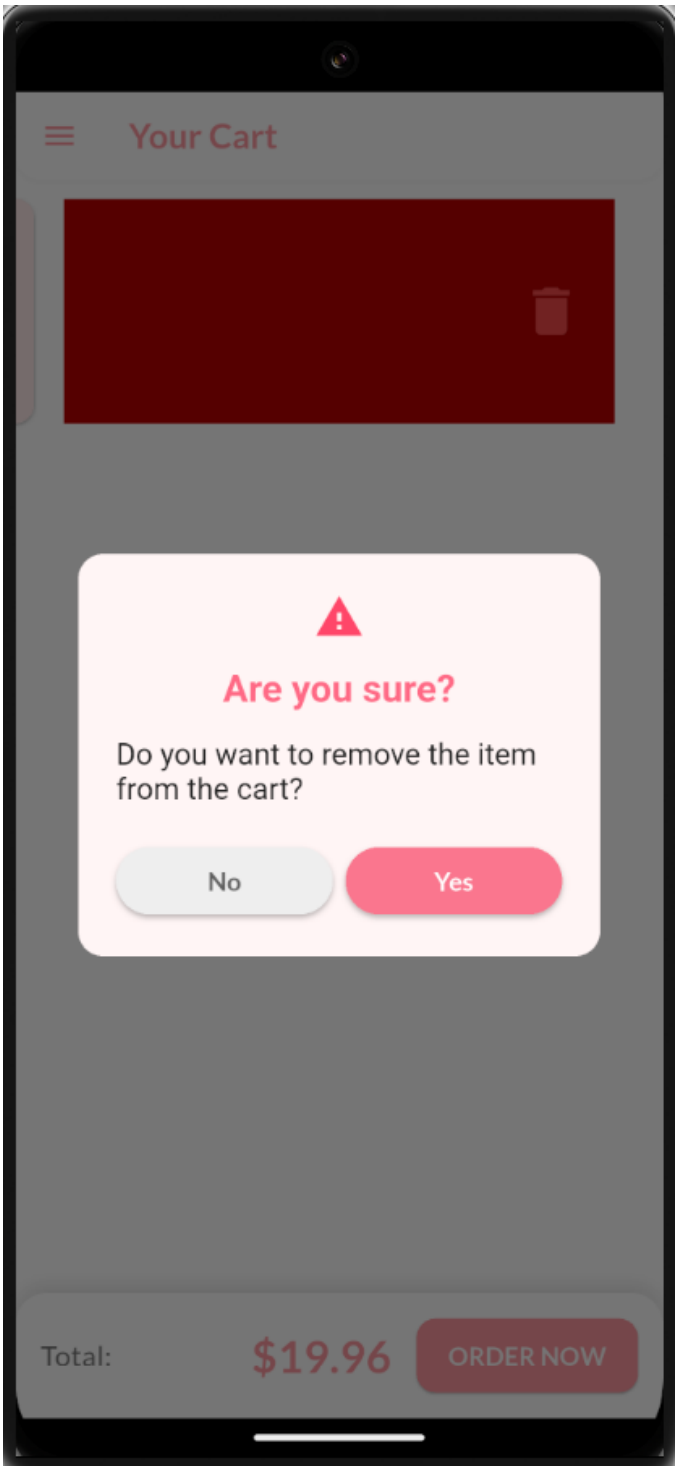
This is the shopping cart interface, we can increase or decrease the number of products added to the cart before.



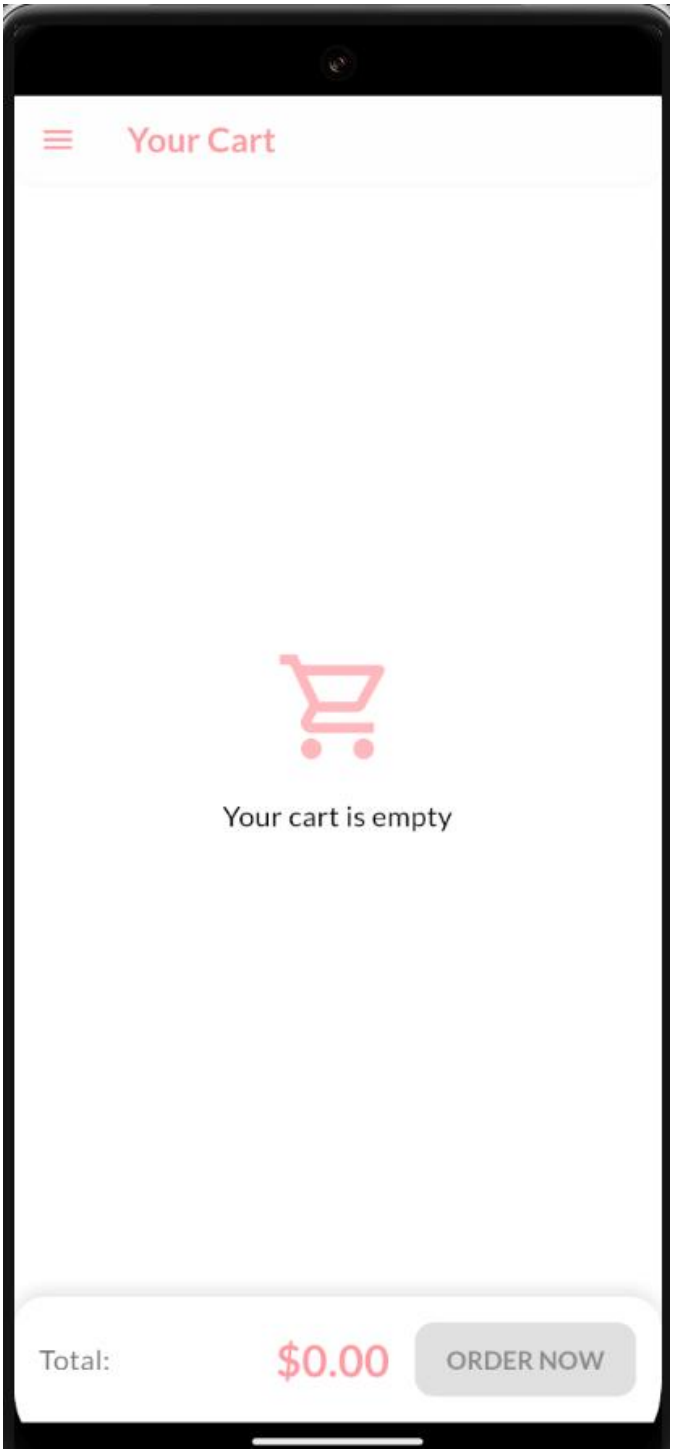
When increasing or decreasing the quantity of products, the quantity and total price will automatically update for us.



Interface when removing products from cart



When the product is successfully deleted, the shopping cart will be automatically updated and the order button will be disabled.



Implementation details:

- ***Widgets Used in this feature***
  - **Scaffold:** Provides the basic app structure with an app bar, body, and drawer.
  - **AppBar:** Displays the cart page title and navigation icons.
  - **Drawer:** Implements a navigation drawer for the cart page.
  - **Column:** Arranges widgets vertically (e.g., cart items list, total section).
  - **Row:** Arranges widgets horizontally (e.g., quantity controls, price, total amount, dialog actions).
  - **Text:** Displays static text (e.g., product title, price, quantity, total amount, dialog messages).
  - **IconButton:** Triggers actions like adjusting quantity or navigating.
  - **Icon:** Displays icons (e.g., add/remove for quantity, delete for removing items, warning/error/success icons in dialogs).
  - **Image:** Loads product images from URLs for cart items.
  - **ClipRRect:** Applies rounded corners to cart item images**.**
  - **Card:** Wraps cart items for a card-like appearance.
  - **Padding:** Adds spacing around widgets for better layout.
  - **SizedBox:** Controls spacing between widgets.
  - **Center:** Centers content (e.g., empty cart message, loading indicator).
  - **ListView:** Displays cart items in a scrollable list.
  - **Dismissible:** Allows users to swipe to remove cart items, with a confirmation dialog.
  - **ElevatedButton:** Provides buttons for placing an order and dialog actions (e.g., "Yes", "No", "Close", "OK").
  - **CircularProgressIndicator:** Shows loading states during data fetch, quantity updates, or order placement.
  - **ScaffoldMessenger:** Displays snack bars for user feedback (e.g., successful order placement, errors).

- **Expanded:** Ensures flexible layout for cart item content, total section, and dialog buttons.
- **Container:** Styles elements like the total section, quantity controls, and dialog content.
- **BoxDecoration:** Applies custom styling (e.g., borders, shadows) to containers.
- **AlertDialog:** Displays confirmation, error, and success dialogs with custom styling.
- **ActionButton (Custom Widget):** A reusable button for dialog actions with customizable text, colors, and actions.

Libraries and Plugins Used

- **flutter/material.dart:** Provides Material Design widgets and framework for building the UI.
- **provider:** Manages state across the app for cart and order data.
- **pocketbase:** Interacts with the backend for cart data operations.
- **flutter/foundation.dart:** Provides state management utilities.

State Management

This feature uses the Provider package with a ChangeNotifier-based UsersManager for shared state management.

Code Architecture:

- **Model Layer**: Defines the data structure for cart items (e.g., product ID, quantity, price).
- **Service Layer**: Handles communication with the backend for cart operations (e.g., adding, fetching, updating, deleting items).
- **Manager Layer:** Manages cart state (e.g., adding, updating, removing items) and order placement.

- **UI Layer:** Renders the UI and interacts with managers for data operations.

How It Works:

- Cart state management maintains a map of cart items and notifies listeners when the cart state changes (e.g., items added, updated, or removed).
- Order state management handles order placement and updates the cart state accordingly.
- Provider makes cart and order managers globally accessible, allowing widgets to react to state changes.
- Local state (e.g., quantity, loading states) is managed within stateful widgets for UI-specific updates.
- Dialogs are stateless and triggered by user actions (e.g., swiping to remove an item, placing an order).

Data Reading and Storage

This feature reads and writes data remotely using the PocketBase backend. No local storage is implemented in the provided code.

- **Data Table Structure (PocketBase carts Collection):**
  - id (String): Unique identifier for the cart item (auto-generated, required).
  - productId (String): ID of the associated product (required).
  - title (String): Product title (required).
  - price (Double): Product price (required).
  - quantity (Int): Quantity of the product in the cart (required).
  - imageUrl (String): URL of the product image (optional).
  - status (String): Status of the cart item (e.g., "pending", defaults to "pending").
  - userId (String): ID of the user who owns the cart item (required).
- **Data Table Structure (PocketBase products Collection - Referenced):**
  - id (String): Unique identifier for the product (auto-generated, required).
  - stockQuantity (Int): Available stock (required).

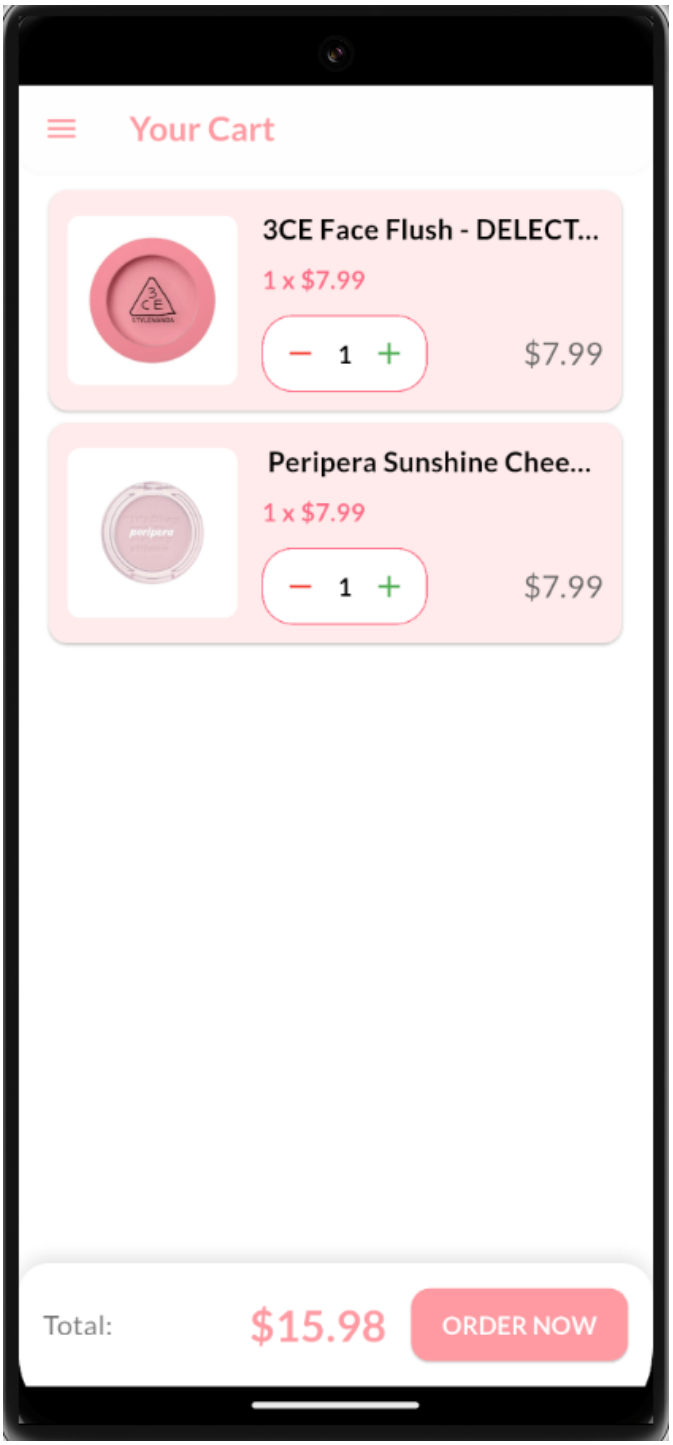
- **PocketBase API Interaction:**
    - **Create Cart Item (Add to Cart):**
        - Call: Creates a new cart item or updates an existing one if the product is already in the cart.
        - Input: Cart item data (e.g., product ID, quantity, user ID).
        - Output: Created or updated cart item.
    - **Read Cart Items (Fetch Cart):**
        - Call: Fetches user-specific pending cart items.
        - Input: Filter string for user-specific data.
        - Output: List of cart items with image URLs.
    - **Update Cart Item (Update Quantity):**
        - Call: Updates a cart item's quantity.
        - Input: Updated cart item data (e.g., quantity).
        - Output: Updated cart item.
    - **Delete Cart Item (Remove Item):**
        - Call: Deletes a cart item.
        - Input: Cart item ID.
        - Output: Boolean indicating success or failure.
    - **Check Stock Quantity (Validation):**
        - Call: Fetches a product's stock quantity.
        - Input: Product ID.
        - Output: Product record with stock quantity.

8. Feature / Application page 8: Update-Status/Create/View Orders – Customer Role

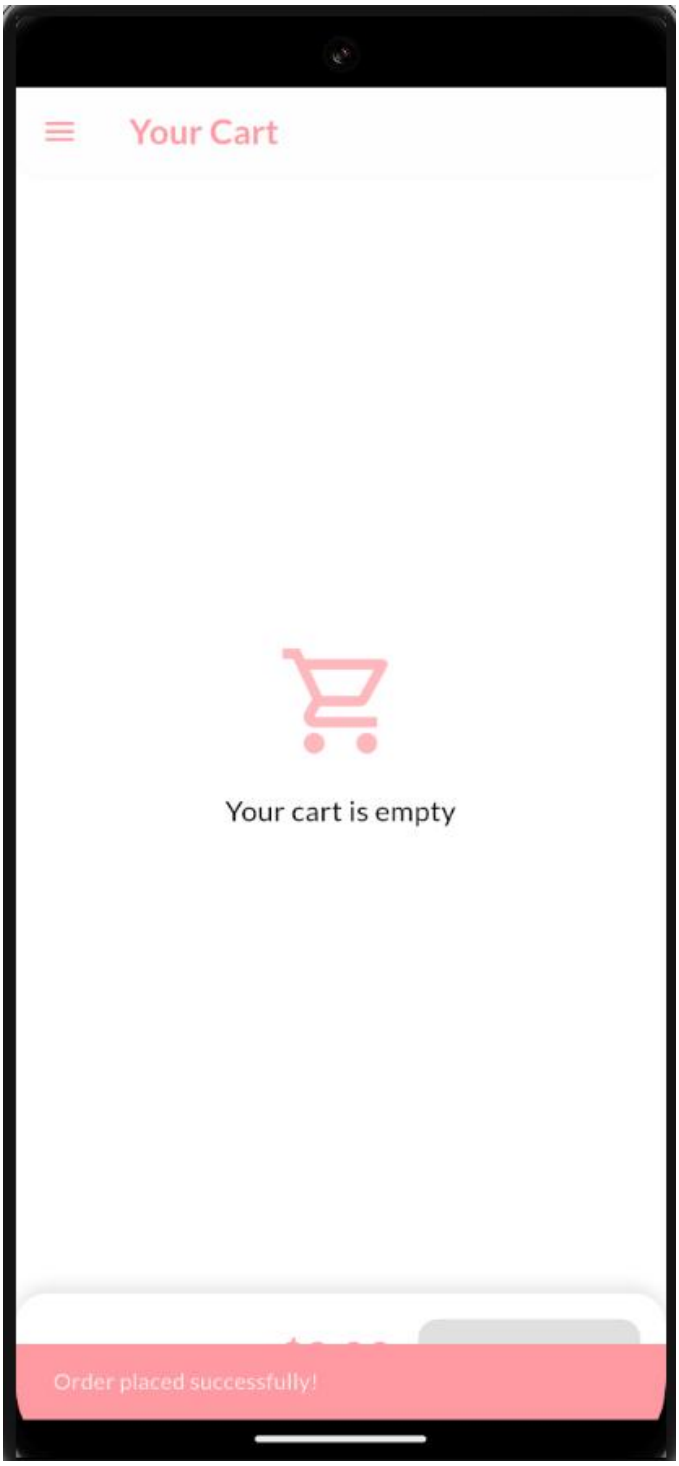
Description: This feature enables users with the "customer" role to perform CRUD operations on their orders. Users can place a new order from the cart (Create), view their order history with details like order amount, date, status, and items (Read), update the status of pending orders to "Completed" or "Canceled" (Update), and delete orders (Delete). The orders page displays a list of orders with expandable cards to show order details, and the cart page allows users to place orders. The feature includes user feedback through snack bars, color-coded status indicators, and smooth UI interactions. It integrates with state management for cart and order data, using a backend for data operations.

Screenshots

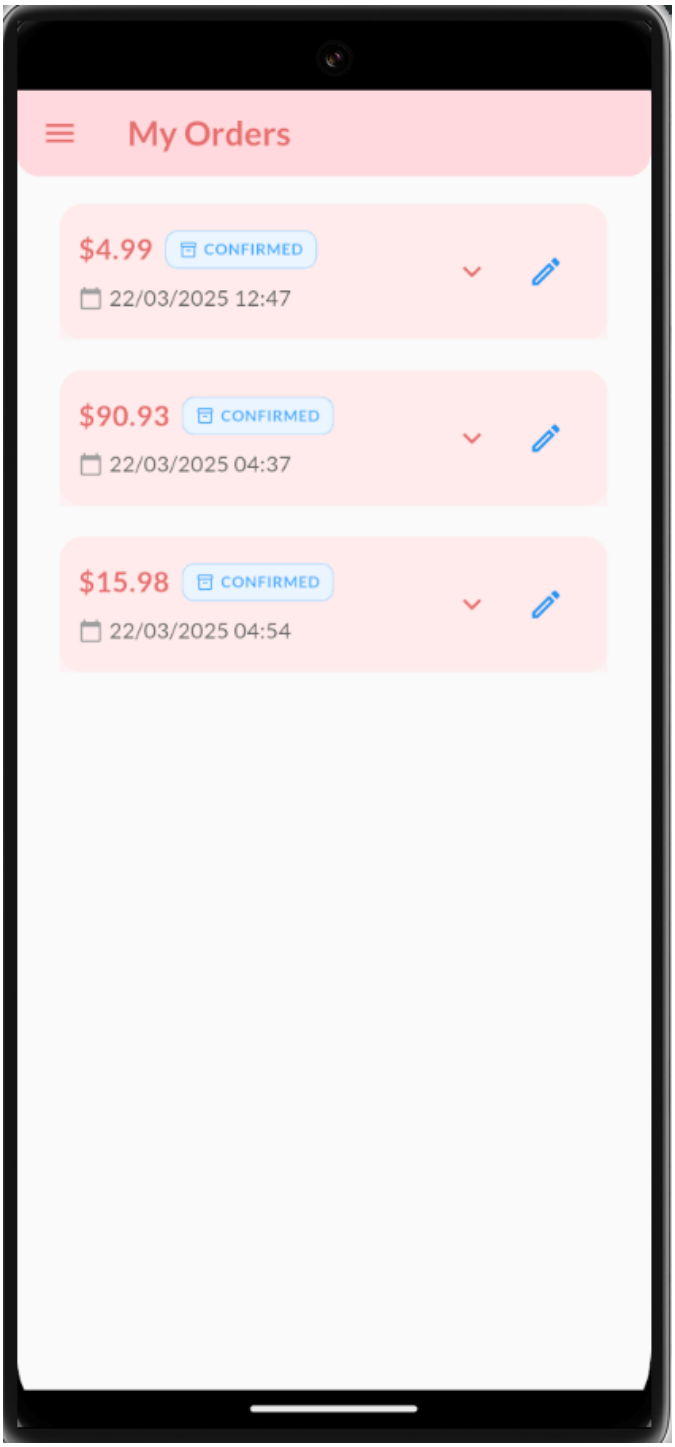
Cart is ready for ordering



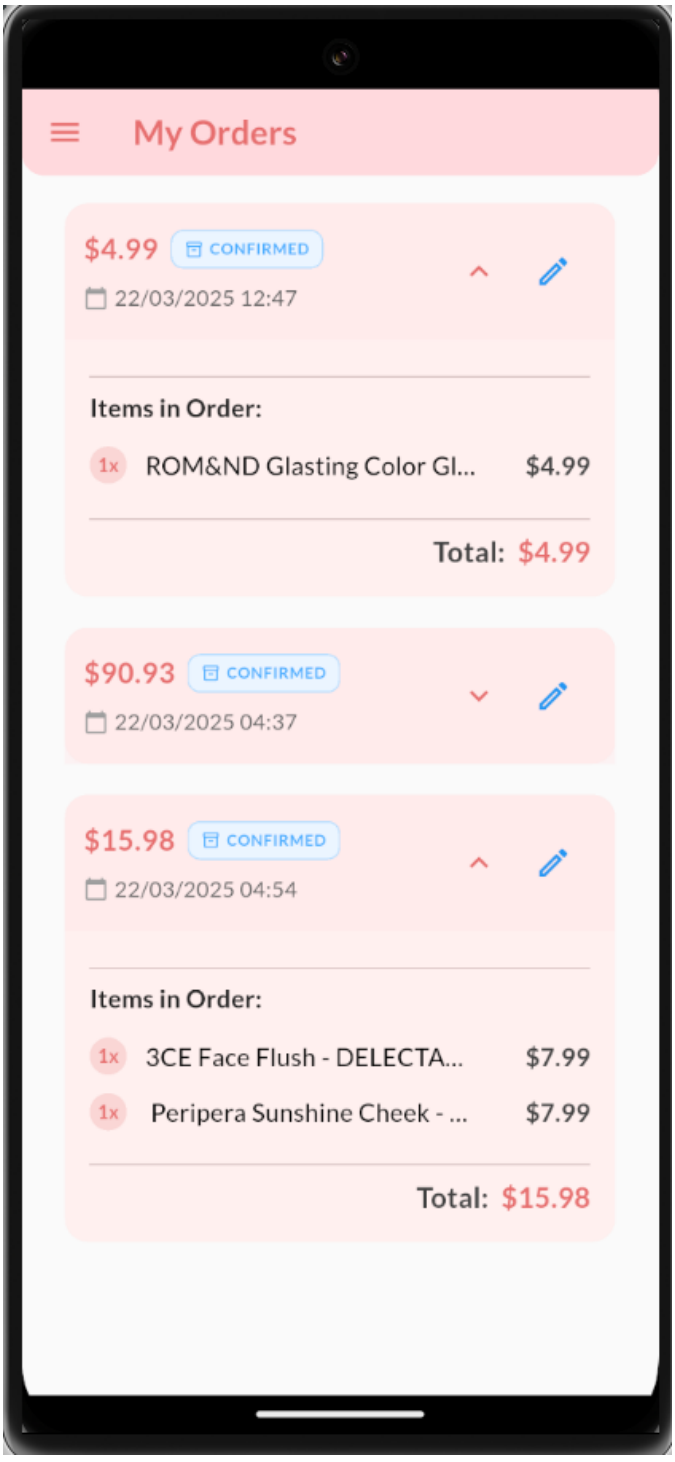
When the order is successful, there will be a notification that the order was successful, the shopping cart will be updated and the order button will be disabled.



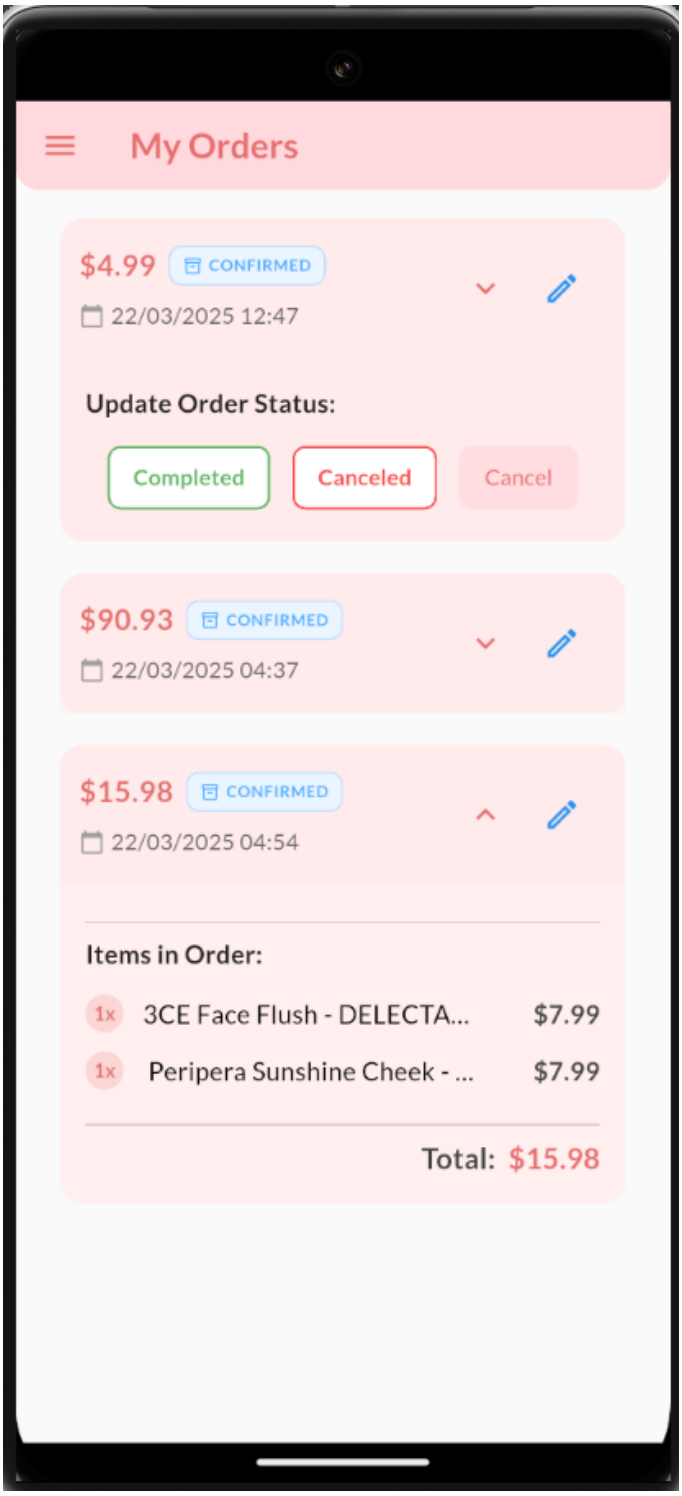
Transaction history interface



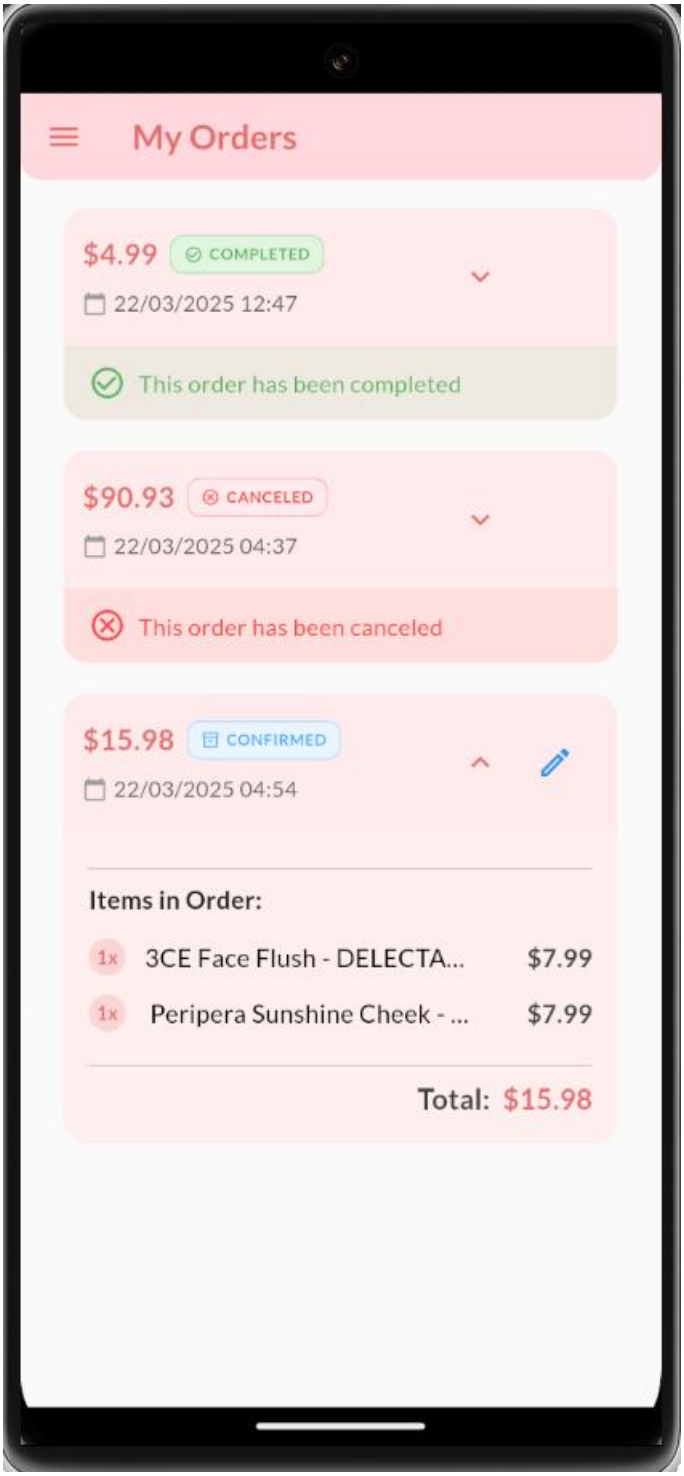
View order details



Click edit button to update order status



When updating the order status to completed or canceled. Once the user has updated one of these two statuses, the status cannot be updated again.



Implementation details:

- ***Widgets Used in this feature***
  - **Scaffold:** Provides the basic app structure with an app bar, body, and drawer.
  - **AppBar:** Displays the orders page title and navigation icons.
  - **Drawer:** Implements a navigation drawer for the orders page.
  - **FutureBuilder:** Manages asynchronous order fetching and displays loading or error states.
  - **Consumer:** Rebuilds the UI based on order state changes**.**
  - **ListView:** Displays orders and cart items in a scrollable list.
  - **Card:** Wraps order entries for a card-like appearance.
  - **Column:** Arranges widgets vertically (e.g., order details, cart items, total section).
  - **Row:** Arranges widgets horizontally (e.g., order amount and status, quantity and price)**.**
  - **Text:** Displays static text (e.g., order amount, date, status, product details).
  - **IconButton:** Triggers actions like expanding/collapsing order details or editing order status.
  - **Icon:** Displays icons (e.g., expand/collapse, status indicators, calendar).
  - **Padding:** Adds spacing around widgets for better layout.
  - **SizedBox:** Controls spacing between widgets.
  - **Center:** Centers content (e.g., loading indicator, empty orders message).
  - **Divider:** Separates sections within order details.
  - **Container:** Styles elements like order headers, status tags, and total section.
  - **BoxDecoration:** Applies custom styling (e.g., borders, shadows, rounded corners) to containers.
  - **OutlinedButton:** Provides buttons for updating order status (e.g., "Completed", "Canceled").
  - **ElevatedButton:** Provides buttons for placing an order or canceling edit mode.

- **CircularProgressIndicator:** Shows loading states during order fetching or placement.
- **ScaffoldMessenger**: Displays snack bars for user feedback (e.g., successful order placement, errors).

Libraries and Plugins Used

- **flutter/material.dart:** Provides Material Design widgets and framework for building the UI.
- **provider:** Manages state across the app for cart and order data**.**
- **pocketbase**: Interacts with the backend for order data operations (assumed via service layer).
- **flutter/foundation.dart**: Provides state management utilities.
- **intl:** Formats dates for display (e.g., order date).

State Management

This feature uses the Provider package with a ChangeNotifier-based UsersManager for shared state management.

Code Architecture:

- **Model Layer:** Defines the data structure for orders (e.g., ID, amount, products, status) and cart items (e.g., product ID, quantity).
- **Service Layer:** Handles communication with the backend for order operations (e.g., adding, fetching, updating, deleting orders).
- **Manager Layer:** Manages cart state (e.g., fetching cart items) and order state (e.g., adding, updating, deleting orders).
- **UI Layer:** Renders the UI and interacts with managers for data operations.

How It Works:

- Order state management maintains a list of orders and notifies listeners when the state changes (e.g., new order added, status updated, order deleted).
- Cart state management provides the data needed to create a new order (e.g., cart items, total amount).
- Provider makes cart and order managers globally accessible, allowing widgets to react to state changes.
- Local state (e.g., expanded state, editing mode) is managed within stateful widgets for UI-specific updates.

Data Reading and Storage

This feature reads and writes data remotely using the PocketBase backend. No local storage is implemented in the provided code.

- **Data Table Structure (PocketBase orders Collection):**
    - id (String): Unique identifier for the order (auto-generated, required).
    - amount (Double): Total order amount (required).
    - products (List): List of cart items in the order (required).
    - dateTime (DateTime): Order creation date and time (required).
    - status (String): Order status (e.g., "pending", "completed", "canceled", defaults to "confirmed").
    - userId (String): ID of the user who placed the order (required).
    - user (Relation): Reference to the user who placed the order (optional).
- **Data Table Structure (PocketBase carts Collection - Referenced):**
    - id (String): Unique identifier for the cart item (auto-generated, required).
    - productId (String): ID of the associated product (required).
    - quantity (Int): Quantity of the product in the cart (required).
- **PocketBase API Interaction:**
    - **Create Order (Add Order):**

        - Call: Creates a new order.
        - Input: Order data (e.g., amount, products, user ID).
        - Output: Created order.
    - **Read Orders (Fetch Orders):**
        - Call: Fetches user-specific orders.
        - Input: Filter string for user-specific data.
        - Output: List of orders.
    - **Update Order (Update Status):**
        - Call: Updates an order's status.
        - Input: Updated order data (e.g., new status).
        - Output: Updated order.
    - **Delete Order (Remove Order):**
        - Call: Deletes an order.
        - Input: Order ID.
        - Output: Boolean indicating success or failure.