PRANJAL BHARTI 2021080
ADITYA DIXIT2022030
KANAV MEENA 2023266

---

1. Introduction

This project is a **mini grocery delivery application**, inspired by platforms like Swiggy or BigBasket. It is built entirely with **Jetpack Compose** and uses an in-memory `AppRepository` as a fake backend.

Key features:

- Two roles: **Customer** and **Shop Owner**
- **5 grocery shops** in Noida, each with **30 items**
- **Cart + checkout** with:
    - Distance and ETA based on latitude/longitude
    - **Atomic ordering** (no partial orders if stock is insufficient)
- **Shop Owner Dashboard** with inventory management and recent orders
- **Item images** using **vector drawable icons** (no real photos)

---

2. Objectives & Requirements

Functional Objectives

1. Allow users to log in as:
    - **Customer** (`cust1/cust1`, `cust2/cust2`)

    - **Shop Owner** (`owner1/owner1` → Shop 1 … `owner5/owner5` → Shop 5)

2. As a **Customer**:
    - Browse shops and items
    - Add items to a cart (supports multiple shops)
    - Enter location (lat/lng)
    - View ETA per shop
    - Place an order with **atomic stock validation**
3. As a **Shop Owner**:
    - View and manage inventory (Add / Edit / Delete items)
    - See recent orders only for their own shop

Non-Functional Objectives

- Simple, clean Compose UI
- No external backend – **in-memory data only**
- Uses **vector drawables** for item icons
- Clear separation of concerns between:
    - UI (Compose screens)
    - Data (models + repository)

---

3. System Architecture

The app follows a **simple layered architecture**:

- **UI Layer (Compose)**
  Defined in `MainActivity.kt`, with composables for:
  - `RoleSelectionScreen`
  - `CustomerHomeScreen`
  - `ShopItemsScreen`
  - `CartScreen`
  - `OwnerDashboardScreen`
- **Data Layer**
  - `AppRepository` (singleton `object`)
  - In-memory lists for shops, items, cart, and orders
  - Handles business logic (login, cart, atomic ordering, ETA)
- **Model Layer**
  - Data classes in `model.kt`:
    `Item`, `Shop`, `CartItem`, `OrderItem`, `Order`

Navigation is implemented with a simple `sealed class Screen` and a `currentScreen` state in `GroceryAppRoot()`.

---

4. Data Model & Repository

4.1 Models (`model.kt`)

```
data class Item(
    val id: Int,
    val shopId: Int,
    var name: String,
```

```
    var price: Double,
    var quantity: Int,
    val iconResId: Int
)

data class Shop(
    val id: Int,
    val name: String,
    val ownerName: String,
    val city: String,
    val latitude: Double,
    val longitude: Double,
    val items: MutableList<Item> = mutableListOf()
)

data class CartItem(
    val shopId: Int,
    val itemId: Int,
    var quantity: Int
)

data class OrderItem(
    val itemName: String,
    val quantity: Int,
    val priceEach: Double
)

data class Order(
    val id: Int,
    val shopId: Int,
    val totalAmount: Double,
    val distanceKm: Double,
    val etaMinutes: Int,
    val items: List<OrderItem>,
    val customerLat: Double,
    val customerLng: Double,
    val timestampMillis: Long = System.currentTimeMillis()
)
```

4.2 Repository (`AppRepository.kt`)

Stored Data

- `shops: MutableList<Shop>`
- `cart: MutableList<CartItem>`
- `orders: MutableList<Order>`
- `users: MutableList<User>`
- `currentUser: User?`

Sample Data

- **5 shops** in **Noida** with slightly different coordinates
- Each shop is populated with **30 grocery items** built from a template list.
- Every item has:
    - Name (with shop suffix, e.g., `"Basmati Rice 1kg - GreenMart"`)
    - Price, quantity
    - `iconResId` linked to a vector drawable

User System

enum class UserRole { CUSTOMER, OWNER }

```
data class User(
    val username: String,
    val password: String,
    val role: UserRole,
    val shopId: Int? = null
)
```

Demo users:

- Customers: `cust1/cust1`, `cust2/cust2`

- Shop Owners:
    - `owner1/owner1` → Shop 1 (GreenMart)
    - `owner2/owner2` → Shop 2
    - `owner3/owner3` → Shop 3
    - `owner4/owner4` → Shop 4
    - `owner5/owner5` → Shop 5

Key Repository Functions

- **User/Login**
    - `login(username, password, role)`
    - `logout()`
    - `getCurrentUser()`
- **Shops & Items**
    - `getShops()`
    - `getShopById(id)`
    - `getAllItems()`
    - `addItem(shopId, name, price, quantity)`
    - `updateItem(shopId, itemId, newName, newPrice, newQuantity)`
    - `removeItem(shopId, itemId)`
    - `findItem(shopId, itemId)`
- **Cart**

    - `getCart()`
    - `getCartSize()`
    - `addToCart(shopId, itemId, quantity)`
    - `updateCartQuantity(shopId, itemId, quantity)`
    - `removeFromCart(shopId, itemId)`
    - `clearCart()`
- **Orders**
    - `placeOrder(customerLat, customerLng): PlaceOrderResult`
    - `getOrdersForShop(shopId)`

---

5. Core Logic

5.1 Atomic Stock Check

Before any stock is reduced, `placeOrder`:

1. Loops over every `CartItem`.
2. Fetches the corresponding `Item`.
3. If `requestedQuantity > item.quantity`, it:
    - Returns `success = false`
    - Sets an error message such as:
      `"Not enough stock for 'Item X'. Requested A, available B."`

    - **Does not change stock**
    - **Does not create any orders**

Only if **all** cart items pass this check does the function:

- Group cart items by `shopId`
- Create one `Order` per shop
- Deduct stock for all items
- Clear the cart

This makes the order **atomic with respect to stock** (no partial fulfillment).

5.2 Distance & ETA (Haversine)

For each shop involved in the cart, the app calculates distance with the **Haversine formula**:

val distanceKm = haversineKm(customerLat, customerLng, shop.latitude, shop.longitude)

ETA is computed as:

- Assume delivery speed = **25 km/h**
- `etaMinutes = distanceKm / 25 * 60 + 10`
- Minimum ETA = **10 minutes**

This is shown on the cart screen as:

```
ShopName: X.X km, ~Y min
```

5.3 Multi-Shop Orders

If items from multiple shops are in the cart:

- The cart is grouped by `shopId`
- One `Order` object is created **per shop**
- All orders are shown in the summary after checkout
- Owner dashboards only show orders for their shop

---

6. Image & Icon System

Items do **not** use real photos.
Instead, they use **vector drawable icons** stored in `res/drawable/`:

Required files:

- `ic_grains.xml`
- `ic_dairy.xml`
- `ic_veggies.xml`
- `ic_fruits.xml`
- `ic_snacks.xml`
- `ic_oil.xml`
- `ic_cleaning.xml`

Icon Logic

Each `Item` has an `iconResId: Int`.

When sample data is created, an icon is directly assigned from the template.
When an owner adds a new item, icons are auto-detected by item name:

- `"rice", "atta", "dal", "poha", "oats"` → `ic_grains`
- `"milk", "curd", "paneer", "cheese"` → `ic_dairy`
- `"potato", "onion", "tomato"` → `ic_veggies`
- `"apple", "banana", "orange"` → `ic_fruits`
- `"chips", "namkeen", "biscuit"` → `ic_snacks`
- `"oil", "mustard", "ghee"` → `ic_oil`
- `"cleaner", "dishwash", "detergent"` → `ic_cleaning`
- Otherwise → **default** `ic_grains`

In UI, icons are displayed using:

```
Image(
    painter = painterResource(id = item.iconResId),
    contentDescription = item.name,
    modifier = Modifier.size(40.dp).padding(end = 8.dp)
)
```

Used in:

- Customer item rows
- Shop items list
- Cart item rows
- Owner inventory list

---

7. UI Design & Navigation

Navigation is handled by:

```
sealed class Screen {
    object RoleSelection : Screen()
    object CustomerHome : Screen()
    object Cart : Screen()
    data class OwnerDashboard(val shopId: Int) : Screen()
    data class ShopItems(val shopId: Int) : Screen()
}
```

State:

```
var currentScreen by remember { mutableStateOf<Screen>(Screen.RoleSelection) }
```

7.1 Screens

1. RoleSelectionScreen

- Toggle between **Customer** and **Shop Owner** login.
- Username, password input.
- Uses `AppRepository.login`.
- On success:
    - Customer → `CustomerHome`
    - Owner → `OwnerDashboard(shopId)` from logged-in user
- Customer mode includes **"Continue as Guest"**.

2. CustomerHomeScreen

Two tabs:

1. **By Shop**
    - Shows list of shops (name, city, number of items).

    - Tap to open `ShopItems(shopId)`.

2. **All Items**
    - Shows all items from all shops.
    - Each row shows icon, name, price, stock, “Add” button.
    - Cart size displayed in top bar.

3. ShopItemsScreen

- Shows only items for a particular shop.
- Each row: icon, name, price, stock, “Add” to cart.
- Top bar: back button + Cart button with live count.

4. CartScreen

- Shows all `CartItem`s grouped visually in a list.
- Each row:
    - Icon, name, price, current stock
    - `-` and `+` buttons to change quantity
    - “Remove” button
- Shows:
    - **Current cart total**
    - Input fields: latitude, longitude
    - Estimated delivery per shop
    - **“Place Order (Atomic Check)”** button
- After placing an order:

    - Cart is cleared
    - Summary of created orders per shop is shown
    - Total bill displayed

5. OwnerDashboardScreen

- Shows inventory items with:
    - Icon, name, price, stock
    - Buttons: **Edit**, **Delete**
- “Add New Item” button opens a dialog:
    - Name, Price, Quantity
    - On save, `addItem` is called and UI reloads from repository.
- Shows **Recent Orders** for that shop:
    - Order ID
    - Total
    - Distance, ETA
    - A one-line summary of items

---

8. Build Instructions

8.1 Prerequisites

- **Android Studio** (Giraffe / Hedgehog / newer)
- Android SDK properly installed

- Gradle configured by Android Studio automatically

8.2 Steps to Build

1. **Open the project**
    - In Android Studio:
      `File → Open...` → select the project folder containing `app/` and `build.gradle`.
2. **Sync Gradle**
    - Android Studio will usually prompt for a **Gradle Sync**.
    - If not, click *"Sync Project with Gradle Files"* from the toolbar.
3. **Check vector drawables**
    - Make sure the following files exist in `app/src/main/res/drawable/`:
        - `ic_grains.xml`
        - `ic_dairy.xml`
        - `ic_veggies.xml`
        - `ic_fruits.xml`
        - `ic_snacks.xml`
        - `ic_oil.xml`
        - `ic_cleaning.xml`
    - If not, recreate them via:
      `res/drawable` → **Right click** → **New** → **Vector Asset** (choose any appropriate clipart).
4. **Select Run Configuration**

- Choose an emulator or connected physical device.

5. **Build & Run**
    - Click **Run** ▶ in Android Studio.

---

9. Usage Instructions

9.1 As a Customer

1. Launch the app.
2. On the **Role Selection** screen:
    - Either log in using:
        - `cust1 / cust1` or `cust2 / cust2`
    - Or click **“Continue as Guest Customer”**.
3. On **Customer Home**:
    - Tab **“By Shop”**:
        - Tap any shop card to view only that shop’s items.
    - Tab **“All Items”**:
        - Scroll through all items.
        - Tap “Add” to add items to your cart.
4. Click the **Cart** button in the top bar to go to the **CartScreen**.
5. On the **Cart** screen:
    - Increase/decrease quantities, or remove items.

- Enter **Customer Latitude** and **Customer Longitude** (e.g. some Noida coordinates).
- View **ETA per shop**.
- Click **“Place Order (Atomic Check)”**.
- If any item is over the available stock, you’ll see an error message and **no order will be created**.
- If all is valid, one order per shop is created and shown in the summary section.

9.2 As a Shop Owner

1. Launch the app.
2. On the Role Selection screen:
    - Switch to **“Shop Owner Login”**.
    - Log in with:
        - `owner1 / owner1` → GreenMart
        - `owner2 / owner2` → DailyNeeds
        - `owner3 / owner3` → Fresh Basket
        - `owner4 / owner4` → UrbanGrocer
        - `owner5 / owner5` → BudgetBazaar
3. You will be taken to **OwnerDashboardScreen(shopId)**.
4. Inventory section:
    - See all items with their icons, price, and stock.
    - Press **Edit** to modify an item’s name, price, or stock.
    - Press **Delete** to remove an item (this also cleans it from carts).

- Press **“Add New Item”**:
    - Enter Name, Price, Quantity.
    - Icon will be auto-assigned based on the name keywords.

5. Recent Orders:
    - Scroll down to “Recent Orders”.
    - See only orders for your shop, sorted by most recent.
