Foddie.

A Food delivery app

|  |  |
| --- | --- |
|  | From Rohal jamal (22sw031) and malhar (22sw104) |

|  |
| --- |
|  |

# Overview

|  |
| --- |
|  |

Foodie is a Flutter mobile app for browsing and ordering food — it provides a menu, cart, checkout, payment flow and user profile/auth screens.

## Problem Identification

|  |
| --- |
|  |

* Provide a cross-platform, responsive MVP for browsing and purchasing food items.
* Enable simple onboarding and session handling with local-only authentication.
* Maintain consistent state (cart, auth) via providers with local persistence.
* Offer a streamlined cart-to-checkout flow with simulated payment.
* Prioritize speed and simplicity over backend complexity, suitable for demos/learning.

## Proposed solution

|  |
| --- |
| For the problem which is identified above, we have developed a mobile app which is far better but simple and easy to use. Main Objectives: |

* Deliver a cross-platform, user-friendly food shopping experience from discovery to checkout.
* Provide reliable session and cart persistence without relying on a backend.
* Ensure a consistent, responsive interface across mobile and desktop.
* Keep the system simple yet extensible for future integration with real services.

## Key Functionalities:

* Onboarding and identity: Guided splash, login, and signup with clear session handling.
* Product discovery: Curated catalog with images, descriptions, and pricing for quick decisions.
* Cart and order flow: Add/remove/update items, view totals, proceed through checkout steps.
* Payment simulation: Clear confirmation, success/failure states, and user feedback.
* Profile and preferences: Basic user info, logout, and placeholders for future order history.
* Persistence and offline readiness: Retain cart and session across app restarts.
* Responsive experience: Layouts and interactions that adapt to varying screen sizes.

# Project features

|  |
| --- |
|  |

*Foodie is a full-featured Flutter food-ordering app that lets users browse a curated menu with images, add items to a persistent cart, and complete orders through a checkout and payment flow; it includes authentication (signup/login), user profile, and order-related screens (cart, checkout, payment). The app persists data locally using an SQLite-backed service so users can keep carts and preferences across sessions and works offline for basic interactions. State is managed with Provider for predictable UI updates, and the UI is componentized into screens and reusable widgets for easy extension. Developer conveniences include a small test suite, static analysis configuration, and tooling to validate or replace asset images, so the app remains robust against corrupted image files.*

# Data storage (SQLite)

|  |
| --- |
|  |

I chose SQLite (via a local DB) because Foodie needs fast, reliable, offline-first persistence for core UX — carts, user preferences, and seeded menu data — without depending on network latency, auth, or cloud costs. A local SQL store is lightweight, deterministic, easy to debug and test, and gives instant reads/writes for UI responsiveness (important for cart/checkout flows). It also simplifies privacy and security (sensitive user/order drafts stay on-device until the user submits), removes the need for a backend during early development, and makes seeding/migrations straightforward with minimal infra.

* SQLite file (on-device): stored at <app\_documents>/foodie.db (sqflite). Tables: users, foods, orders. Passwords are salted & hashed with bcrypt before insert; timestamps are stored as integer milliseconds since epoch.
* Table users — purpose: auth & profile storage
  + id (INTEGER, PK AUTOINCREMENT) — local user id
  + email (TEXT, UNIQUE) — login identifier
  + passwordHash (TEXT) — bcrypt hash of password
  + firstName, lastName, address, city, phone (TEXT) — optional profile/contact fields
* Table foods — purpose: menu/catalog (seeded from packaged assets)
  + id (TEXT, PK) — string id (seeded as "1","2",...)
  + name (TEXT) — display name
  + description (TEXT) — short description
  + price (REAL) — numeric price (double)
  + imageUrl (TEXT) — asset path or mapped asset (e.g., "assets/images/cheese\_pizza.png"); legacy numeric ids are mapped to packaged filenames
* Serialization & validation notes
  + Orders.items uses JSON (string) to store the ordered items array inside orders.items.
  + Price fields are REAL; code defensively parses numeric/string cases.
  + Image references are stored as strings (asset paths); code normalizes numeric legacy ids to asset filenames and falls back to a guaranteed asset if missing.

# Project Design and Interface

## High-level architecture:

*Foodie follows a simple layered architecture: a presentation layer (Flutter widgets/screens under*[*screens*](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)*and*[*widgets*](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)*), a state layer using Provider (*[*providers*](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)*), and a data layer with a local persistence implementation ([local\_db.dart](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ")) that either uses SQLite (sqflite) or an in-memory fallback for web. Business logic (authentication, cart management, order placement) lives in providers/services so UI code stays thin and focused on rendering. The app seeds a local catalog of foods from packaged assets, which makes it self-contained and offline-capable.*

## UI & Navigation:

*The UI is organized into feature screens: splash, login/signup, home (catalog), product details, cart, checkout/payment, and profile. Navigation uses a central navigator (likely via the app\_navigator.dart) and follows standard mobile patterns: persistent top/bottom app chrome where needed, grids or lists for menus, and modal/dedicated flows for checkout. Widgets are componentized into reusable building blocks (cards, buttons, image loaders) so presentation is consistent across screens.*

## State management & data flow:

*Provider is used to expose state to the widget tree (authentication state, cart contents, user profile). UI listens to providers and reacts to changes; actions dispatch to providers which call the LocalStorage implementation to persist changes. The LocalDB wraps access to SQLite and provides deterministic methods: getFoods(), createUser(), authenticate(), placeOrder(), getUserOrders(). Orders serialize item lists as JSON strings in the DB; images are referenced by asset paths. This flow enables quick local reads/writes for responsive UI and defers any network sync to a future sync layer.*

## UI Images:

## Splash Screen:

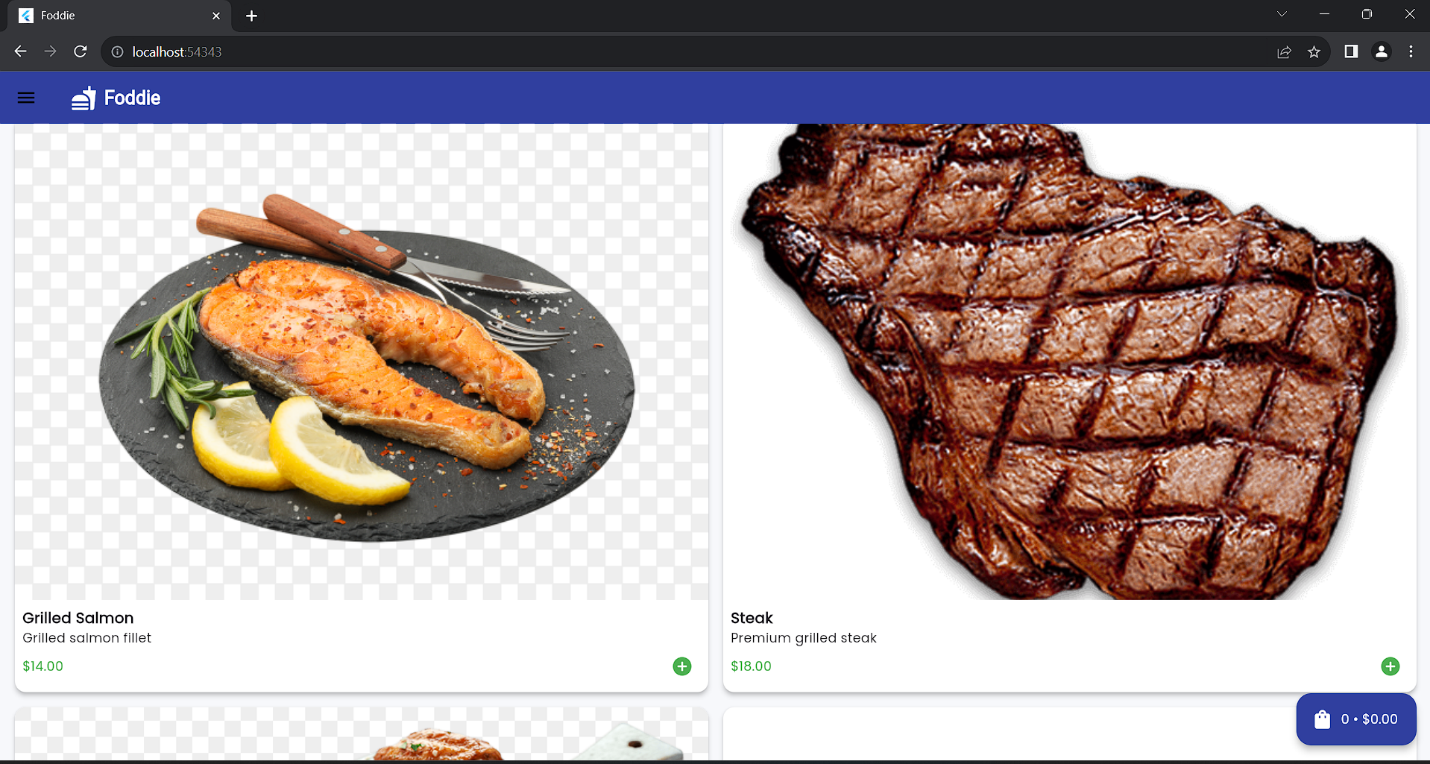
## sign-up screen:A screenshot of a computer AI-generated content may be incorrect.

## login screen:

A screenshot of a computer

AI-generated content may be incorrect.

## Home screen:



## cart screen:

A screenshot of a computer

AI-generated content may be incorrect.

## details screen:

A screenshot of a computer

AI-generated content may be incorrect.

## Payment screen:

A blue and white screen

AI-generated content may be incorrect.

## Drawer screen:

A screenshot of a food menu

AI-generated content may be incorrect.

## profile screen:

A screenshot of a computer

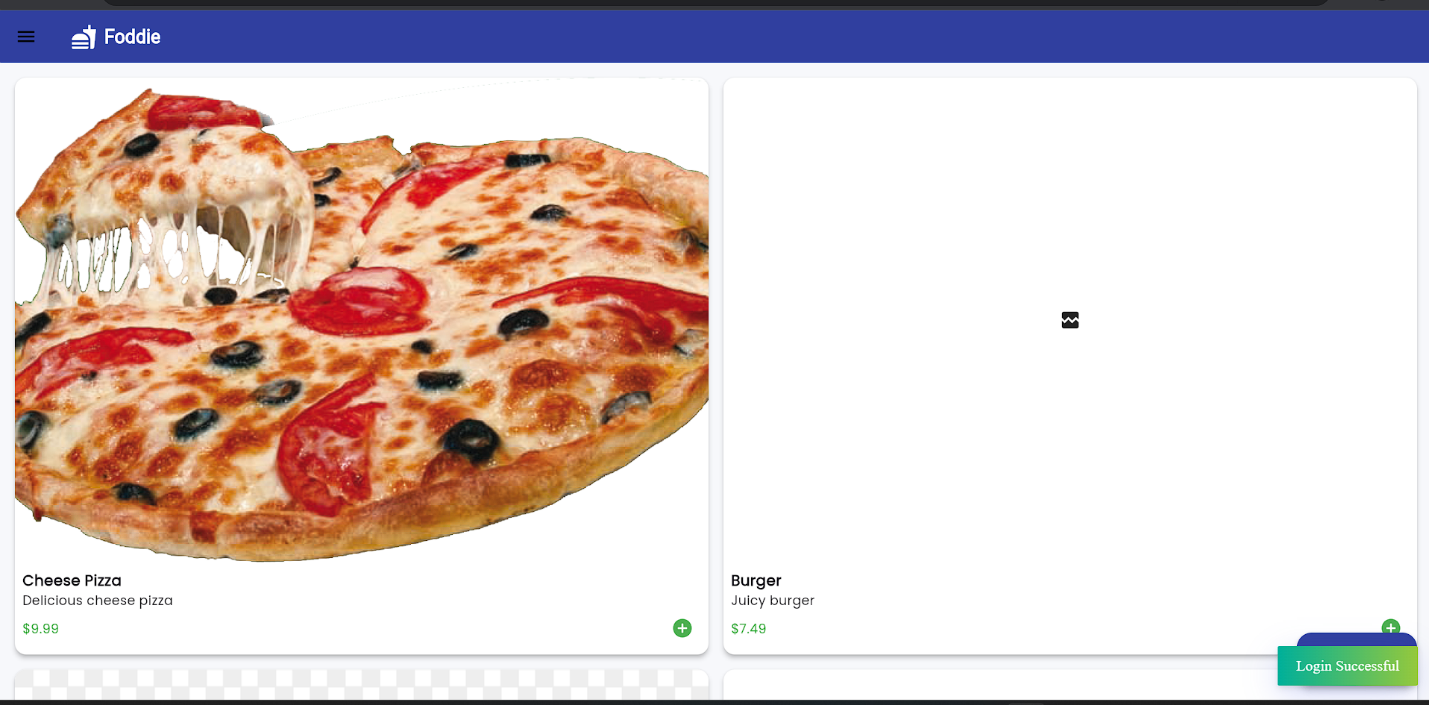
AI-generated content may be incorrect.

## logout:

A screenshot of a phone

AI-generated content may be incorrect.

## login successful:



# Error & bugs

|  |
| --- |
| Describe what sets your company apart from your competition (your unique selling proposition). Missing plugin exception:  What caused the error MissingPluginException(No implementation found for method getApplicationDocumentsDirectory on channel plugins.flutter.io/path\_provider) happens when the app calls the path\_provider platform plugin but the plugin is not available/registered on the current platform. This commonly appears when: Running on web: sqflite/path\_provider's platform implementations are not available on the web (sqflite is mobile/desktop only). A plugin is invoked before Flutter binding/plugin registration happens.  In this case the app was calling getApplicationDocumentsDirectory (from LocalDB.init) on a platform where the plugin was not available (web), causing the MissingPluginException during sign-up / login. What I changed (code edits): Added an in-memory fallback storage to local\_db.dart:  New class InMemoryLocalStorage implements LocalStorage that stores users, foods and orders in memory (no platform plugins). Keeps a small sample food dataset, allows createUser/authenticate/getFoods/placeOrder/getUserOrders. HomeScreen can't find a Provider<CartProvider>:   That means at runtime the widget tree that built HomeScreen didn't include a CartProvider ancestor.  The code fix was to add the CartProvider to the app root MultiProvider in main.dart (I added it). That resolves the code-level cause.  However if the running app was started before that change and only hot-reloaded, Provider additions at the app root may not be visible to already-instantiated routes — a hot-restart (or stop & relaunch) is required to rebuild the root widget tree and make the new provider available to all routes. What I changed (confirming): main.dart now registers both:  AuthProvider  CartProvider  So HomeScreen (and other routes) will be able to access context.watch<CartProvider>() or Provider.of<CartProvider>(context) once the app is restarted error while loading menu on the home page:  Root cause (why the menu failed): HomeScreen loads its menu with LocalDBService().getFoods() (a FutureBuilder).  LocalDBService previously instantiated LocalDB() directly. LocalDB.init() calls getApplicationDocumentsDirectory() and sqflite APIs.  On platforms that don't provide those platform plugins (notably web), calling getApplicationDocumentsDirectory() throws a MissingPluginException or otherwise prevents the SQL DB from opening — so the FutureBuilder failed and HomeScreen couldn't load the menu.  In short: HomeScreen used a service that invoked platform-only plugins on web. What I changed (concrete): 1.Made LocalDBService platform-aware and use the LocalStorage abstraction  File: local\_db\_service.dart  Change: instead of directly using LocalDB(), the service now chooses  between:  LocalDB() (sqflite-backed) on mobile/desktop, and  InMemoryLocalStorage() on web (no platform plugins).  Implementation uses kIsWeb to detect web and the LocalStorage interface already added earlier. This prevents web builds from calling path\_provider/sqflite.  2.Kept single responsibility / backward compatibility   * LocalDBService still exposes the same methods (getFoods, placeOrder, getUserOrders) and still accepts string user IDs like before.   Under the hood it converts the string userId to int and delegates to the chosen storage.  3.Small cross-checks / safety measures   * AuthProvider had already been updated earlier to default to InMemoryLocalStorage() on web — the missing piece was that LocalDBService (used by HomeScreen) still used LocalDB(). I changed the service to match and use the same fallback. * I ran the analyzer and made small fixes elsewhere earlier; flutter analyze now reports no issues.  after loging the logout functionality not working:  What I changed (concrete): Edited custom\_drawer.dart:  On "Logout" tap we now:  Await AuthProvider.signOut().  Use the global appNavigatorKey to push the LoginScreen and clear the navigation stack (pushAndRemoveUntil(..., (route) => false)), so the user always returns to the login screen and can't press Back to return to an authenticated page.  Why this fixes the problem Why this fixes the problem: Previously logout only cleared auth state and popped the drawer. That left the user on the same route (HomeScreen). Depending on the UI, HomeScreen might still render or require user data, making it look like logout didn't work. Replacing the navigator stack with LoginScreen ensures the UI is reset to the unauthenticated state.  Using the global navigator avoids BuildContext issues in drawer callbacks and guarantees the navigation uses the app-level navigator. Image Codec Exception:What I changed:Problem: Asset files currently contain base64 text (not decoded binary). Flutter's image decoder failed with "ImageCodecException: Failed to detect image file format using the file header". fix: I updated home\_screen.dart so asset images are loaded robustly by reading the raw asset bytes using rootBundle.load(...). The loader:  Detects PNG/JPEG magic headers and uses the bytes directly with Image.memory.  If the asset contains base64 text, it decodes it and then uses Image.memory.  Falls back to a friendly placeholder UI on error.  This avoids the exception whether the asset files are:  real binary PNG/JPEG files, or text files that hold base64-encoded image data (like the ones currently present). |
|  |
|  |

# Technologies and tools used

## flutter & dart:

Used as the core framework and language for the app — UI, navigation, and stateful widgets are written in Dart and rendered with Flutter. All screens, widgets, theming, and app entrypoint live under [lib](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html).

## Sqlite:

Local on-device relational store used for persistent data: users, foods, and orders tables live in foodie.db (see [local\_db.dart](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ")). Chosen for fast, offline-first reads/writes and deterministic local persistence.

## path\_provider / path:

Platform-safe file paths and locations (app documents directory) are provided by path\_provider; path is used to construct portable paths. These enable locating the SQLite file and any local backups.

## bycypt / crypto:

Password hashing (bcrypt) is used for local user credential hashing before storage (passwordHash column). crypto is available for any additional hashing or crypto utilities

## provider:

State management library used to expose app-wide state (auth, cart, profile) to the widget tree. Providers mediate UI actions and call the LocalStorage interface.

## http (dart http client) & dart httpclient (tools):

HTTP is used by repository tooling and optional networking code — tools in [tools](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) use the Dart HTTP client to fetch replacement images (Picsum/Unsplash sources). In-app networking (if added) would also use http.

## tools & helper scripts(tools):

Utility scripts under [tools](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html):

* [replace\_images.dart](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) — writes placeholder PNGs into asset filenames.
* fetch\_images.dart / [fetch\_picsum.dart](vscode-file://vscode-app/c:/Users/ZASS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ") — download real images (Picsum/Unsplash) for assets, with validation, backup, and retry/backoff logic.  
  These are developer utilities (not shipped) for maintaining/repairing asset images.

## ui & theming utilities:

* google\_fonts — for custom fonts usage across the app theme.
* fluttertoast — for toast notifications used by quick user feedback.
* cupertino\_icons — platform iconography compatibility.

## platform considerations:

* Cross-platform Flutter targets: Android, iOS, web, macOS, Windows, Linux (project contains platform folders). Web falls back to InMemoryLocalStorage because sqflite is not available on web.
* Path- and permission-aware file I/O handled via path\_provider and runtime checks (kIsWeb).