NomNom Safe — Architecture Overview

Concise design notes for engineers: system components, data model, DI/testing, deployment, and extension points.

Goals

- Provide reliable allergen-aware restaurant discovery.
- Keep UI responsive with local filtering and background data fetches.
- Enable testable business logic via dependency injection.
- Support secure, auditable user profiles.

High-level Components

- Flutter app (mobile + desktop/web targets)
- State: Provider (ChangeNotifier) for `AuthStateProvider` and controllers
- Services: `AuthService`, `FirestoreAdapter` wrappers, `AllergenService`, `RestaurantService`
- Back end: Firebase (Auth + Firestore) via adapter interfaces
- Tests: unit, widget, integration, acceptance, regression, e2e — use fakes/adapters

Key Design Patterns

- Adapter pattern: `AuthAdapter` & `FirestoreAdapter` abstract SDK usage for test doubles.
- Singleton factory with test reset: `AuthService.clearInstanceForTests()`.
- Provider-based DI: inject services via `Provider` or constructor params for controllers/screens.
- Controller objects (e.g., `EditProfileController`) encapsulate view logic and notify UI.

Data Model (core entities)

- User: id, firstName, lastName, email, allergies[]
- Restaurant: id, name, addressId, cuisine, menu reference
- Menu: items[]
- MenuItem: id, name, allergens[]
- Allergen: id, label

Store layout (Firestore): collections `users`, `restaurants`, `menus`, `menu_items`, `allergens`.

Service Responsibilities

- AuthService: create/sign-in/update/delete users; manage `currentUser` cache.
- FirestoreAdapter: thin wrapper for testing (mockable `collection().doc()` API).
- AllergenService: load/cached allergen maps and ID/label helpers.
- RestaurantService: fetch restaurants, filter by allergen/cuisine, query menus.

Dependency Injection & Testing

- Prefer constructor injection for services used directly by controllers/screens.
- Use typed fake providers (e.g., `_FakeAuthProvider extends AuthStateProvider`) for widget tests.
- Reset singletons in `setUp()` with `AuthService.clearInstanceForTests()`.
- Keep production-only Firebase initialization out of tests by providing fake adapters.

UI → Business Flow Examples

- HomeScreen: on init, fetch allergens and restaurants via services; show spinner until ready.
- Filter application: controller updates selected allergen IDs → `RestaurantService.filterRestaurantsFromList()`
- EditProfile: controller writes updates through `AuthStateProvider.updateProfile()` → `AuthService` updates Firestore.

CI / Quality

- Run: unit + widget + integration tests on PRs.
- Use headless Flutter test runner on CI; ensure no Firebase initialization in tests.
- Include regression tests covering adapter behavior and controller logic.

Security & Privacy

- Authenticate with Firebase Auth; never log raw passwords.
- Store minimal PII (name, email, allergies). Follow org deletion/retention policies.
- Use Firestore rules to limit reads/writes per authenticated user.

Extensibility Notes

- Add new data sources by implementing `FirestoreAdapter` or a new adapter type.
- Add caching layers (local DB) behind service interfaces to reduce reads.
- Extract shared test fakes to `test/test_helpers/fakes.dart`.

Developer Setup (quick)

1. Install Flutter SDK matching project.
2. Copy `serviceAccountKey.json` or use test fakes for local dev.
3. Run: `flutter pub get` then `flutter run` or `flutter test`.

References

- `lib/services/*` — service implementations
- `lib/providers/*` — state providers
- `test/` — examples of fakes and harnesses