AI Recipe App — Technical Report

October 21, 2025

Abstract

Abstract (English)

This technical report presents the architecture, design principles, and implementation details of the AI Recipe App, a two-tier system integrating AI-driven recipe generation with a mobile culinary assistant. The project consists of two complementary components that work in concert to deliver a seamless and intelligent cooking experience.

1. **AI Backend Service** — Built with FastAPI, this service transforms unstructured ingredient inputs from users into well-formed, TheMealDB-compatible recipes through a carefully engineered large language model (LLM) pipeline. The backend emphasizes reliability, determinism, and schema compliance, employing Pydantic-based data validation, structured JSON responses, and request caching to guarantee reproducible outputs. It exposes a clearly defined RESTful API that acts as an intelligent augmentation layer atop traditional recipe databases.
2. **Android Application** — Developed in Kotlin, the mobile client implements a Clean Architecture pattern supported by Retrofit, Room, and ViewModel/Flow components. The app consumes both TheMealDB API and the custom AI backend, enabling users to search, generate, and store recipes with offline-first capabilities. The design ensures a consistent, reactive, and user-friendly experience aligned with modern Android development standards.

The overarching objective is to establish a production-grade, scalable, and maintainable ecosystem that bridges the gap between structured culinary data and AI-generated creativity. Key architectural concerns include:

- Exact API contracts ensuring data integrity and forward compatibility
- Unified data flow and robust error propagation across backend and client layers
- Scalable deployment pipelines leveraging Docker, CI/CD automation, and container orchestration
- Operational observability via metrics, logging, and distributed tracing tools (e.g., Prometheus, Grafana, Sentry)
- Extensibility for future AI model integration and fine-tuning on domain-specific recipe datasets

By integrating deterministic LLM inference with traditional APIs, this system demonstrates a practical blueprint for hybrid AI–API architectures—where generative intelligence is operationalized in real-world software environments. The resulting platform offers not only an enhanced user experience but also a reproducible and maintainable foundation for future intelligent culinary applications.

Contents

1 Executive Summary **3**

1.1 How the system works (end to end) 4

1.2 Outcome 4

2 System Overview **4**

2.1 High-Level Data Flow 4

2.2 Why the Repository Pattern 4

3 Step 1 — Inventory **6**
3.1 Repository Inventory 6
3.2 API Configuration (File: `API.kt`) 7
3.3 AI Android Client (Request, Service, Network, Repository) 9
3.4 Local Persistence (Room — Database & DAO) 11

4 ViewModels **13**
4.1 Detail Screen ViewModel (HowToViewModel) — loading, favorites, and DI . . . 13
4.2 Home Screen ViewModel (HomeViewModel) — filters, banner, and favorites overlay 15
4.3 Favorites ViewModel (FavoritesViewModel) — streaming list & search 17
4.4 AI Generate ViewModel (AiRecipeViewModel) — request pipeline & UI state . . 18

5 Retrofit services and repositories — exact responsibilities (Android side) **19**

6 Network Response Models (Android) — `MealResponse` & `CategoriesResponse` **20**

7 AI Backend — responsibilities and pipeline **23**

8 FastAPI Reference Implementation (server-side Python) **25**

9 JSON Contracts **26**

10 Step-by-Step Build Order (reference) **26**

11 User Interface (Activities & Fragments) **27**
11.1 Home (Fragment) 27
11.2 Detail Screen (Activity) 31
11.3 Favorites (Fragment) 34
11.4 AI Generate (Fragment) — UI responsibilities & rendering 37
11.5 Bindings 43
11.6 Adapters — Code & responsibilities 43
11.7 Main Activity (Navigation shell & BottomNavigationView) 47
11.8 Handling UI XML Files 49

12 Wrap-up & Conclusion **49**

1 Executive Summary

This report provides a clear and practical guide for building and deploying a complete AI-powered recipe application. The system is made up of two main parts that work together seamlessly:

- an Android app that allows users to browse and search for meals, generate new recipes with AI, and save their favorites for later; and
- a lightweight AI backend that converts plain lists of ingredients into structured, ready-to-use recipes.

The project focuses on practical engineering rather than theory. It explains exactly how the architecture is organized — the Android client follows a clean MVVM (Model–View–ViewModel) design combined with Repository, Room, Retrofit, Coroutines, and Glide. On the server side, a FastAPI backend exposes a clear, versioned endpoint that follows strict input/output rules. Together, these components form a simple but production-ready system that can go from "runs on my laptop" to "ready for release." The Android app includes everything needed for modern, reliable mobile development. It defines Gradle setup, package structure, and uses ViewBinding for type-safe layouts. Offline favorites are handled using Room entities and DAOs, while Retrofit provides connections to both the public TheMealDB API and the custom AI backend. A single Repository keeps all data consistent across the app. The user interface uses Fragments and Adapters with clear roles, supporting smooth loading, empty, and error states. Actions like pressing the "favorite" heart button are instantly reflected thanks to reactive updates via Kotlin Flow. The backend is minimal yet powerful. It exposes one main API endpoint — `/v1/ai/recipe` — which accepts a simple list of ingredients (for example: "eggs, tomatoes, onion") and returns a structured JSON recipe object compatible with TheMealDB format. Each response includes a recipe title, up to 20 ingredient-measure pairs, cooking instructions, and optionally an image URL. The backend also includes important safety measures such as request timeouts, rate limits, and secure logging with request IDs. Sensitive data is hidden, so the system can be debugged safely without leaking any secrets. The way data moves through the system is simple and clear from beginning to end.

1. **Browsing and searching for meals:** When a user opens the app, they can browse or search for different meal ideas. The app's interface sends the request step by step — first through the ViewModel, then the Repository, and finally to Retrofit, which connects to TheMealDB API. The API sends back the list of meals, and the app shows them to the user.

2. **Viewing meal details:** When a user selects a specific meal, the app loads all of its details — including ingredients, instructions, and related images or videos — directly from TheMealDB and displays them neatly on the detail screen.

3. **Saving favorites:** Users can tap the heart icon to mark a recipe as a favorite or remove it. Behind the scenes, this action updates the local Room database through the Repository layer. Thanks to Kotlin Flow, the app updates instantly — the favorites list changes right away without the user needing to refresh anything.

4. **Generating recipes with AI:** When users want to create a new recipe, they simply enter a few ingredients (for example: "eggs, tomatoes, onion"). The app sends this input to the AI backend using a POST request to `/v1/ai/recipe`. The backend processes the request through its language model (LLM), creates a complete recipe, checks and corrects its format, and sends back a clean, structured JSON recipe to the app. The app then displays the generated recipe just like a normal one from TheMealDB.

1.1 How the system works (end to end)

1. **Browse & search meals:** UI → ViewModel → Repository → Retrofit → TheMealDB API.
2. **Open details:** UI loads by `mealId` and renders instructions/media.
3. **Toggle favorites:** UI → ViewModel → Repository → Room; UI stays in sync via Flow.
4. **Generate a recipe:** UI → ViewModel → Repository → POST `/v1/ai/recipe` → LLM pipeline → schema validate/repair → JSON back to UI.

1.2 Outcome

The result is a mobile app and backend that are easy to use, easy to maintain, and safe to deploy. The whole system works smoothly — responses are predictable, errors are handled clearly, and all data remains secure. In simple terms, this project shows how an AI feature can be added to a real mobile app in a way that is clean, reliable, and ready for real-world use.

2 System Overview

2.1 High-Level Data Flow

The overall data flow of the system is designed to be simple, consistent, and predictable. When a user interacts with the app — for example, by searching for a meal, viewing details, or generating a new recipe — the request follows a clear and structured path. The User Interface (UI) sends the user's action to the ViewModel, which then communicates with a central Repository. The repository decides whether to fetch the data from the network (via Retrofit and TheMealDB API or the AI backend) or from the local Room database if the data has already been stored. This layered design ensures that every piece of data — whether it comes from the internet or local storage — passes through a single, unified data channel. This not only keeps the flow consistent but also makes the app's behavior easier to understand, debug, and maintain. The same path is used when the AI backend generates new recipes: the request travels from the app to the server, is processed by the AI model, and the structured recipe is returned and displayed instantly on the user's screen.

2.2 Why the Repository Pattern

As the first step in implementing the mobile client, the system focuses on structuring the data layer using the Repository pattern. This pattern serves as a centralized point for handling all data operations—whether the source is remote (network) or local (on-device storage). It ensures a consistent and clean data flow throughout the app while keeping the UI layer decoupled from data management concerns.

At the center of this layer is the `MealRepository`, which acts as the single source of truth for all meal-related data. It integrates with multiple Retrofit-based services that connect to TheMealDB API—namely: `SearchAPIService`, `FilterAPIService`, `LookupAPIService`, and `RandomAPIService`. On the local side, it communicates with the `MealDao`, which provides access to the Room database for storing user favorites.

Responsibilities of MealRepository include:

- Fetching data from remote APIs (such as search, filter, or random meals).
- Managing CRUD operations for user favorites in the Room database.
- Mapping API response objects (`MealResponseItem`) to persistent entities (`FavoriteMealEntity`).

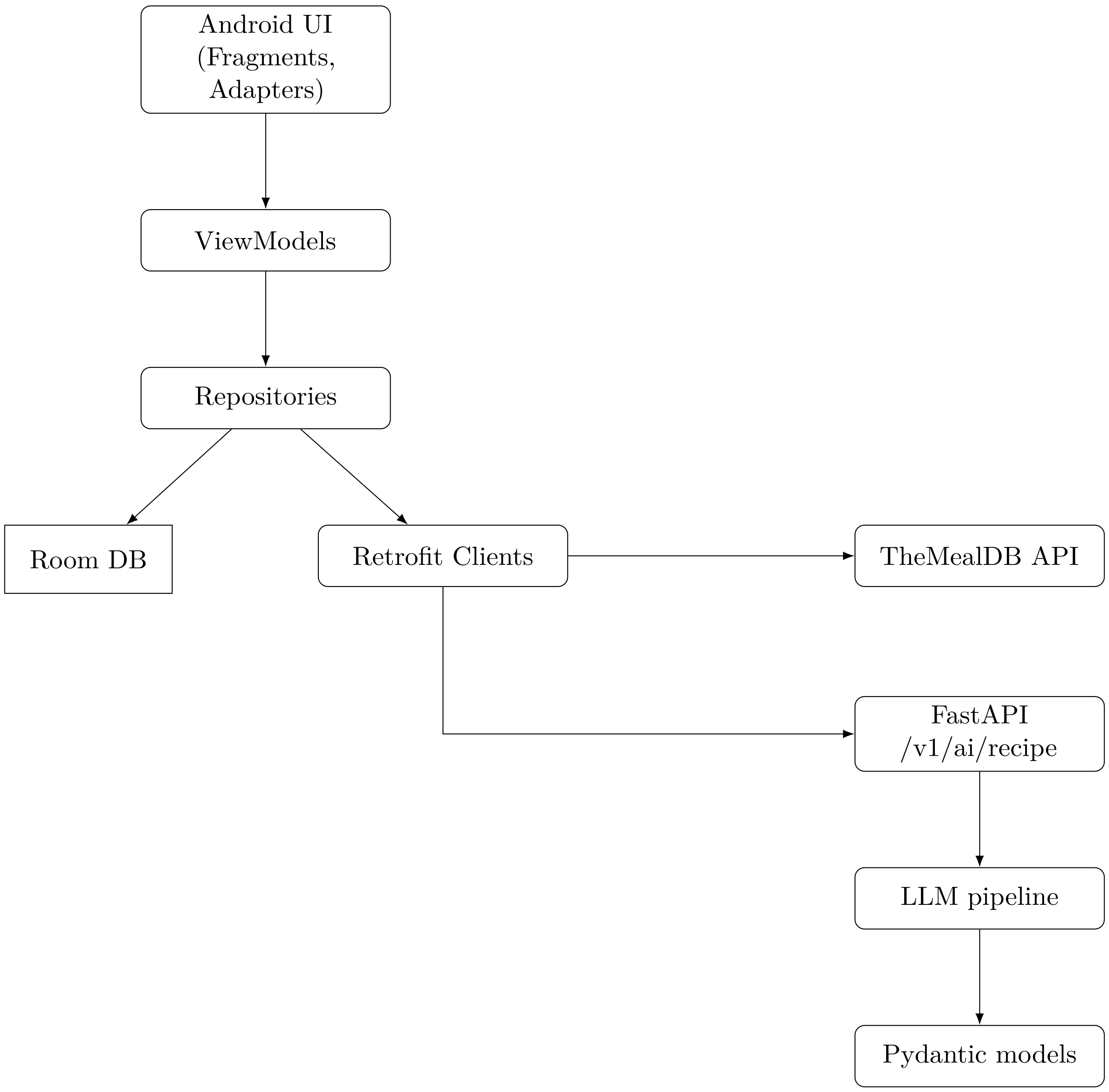
- Wrapping responses into `Result` types and transforming raw errors into structured failure modes—thus preventing direct exposure of Retrofit responses to upper layers.

From a functional perspective, the repository exposes a concise public API with methods like `searchMeals(query)`, `filterByCategory(category)`, `lookupMeal(id)`, and `getRandomMeal()`. It also provides a live data stream of saved favorites via `favoritesFlow()`, which is based on Kotlin Flow and keeps the UI automatically updated. Additionally, methods for saving and removing favorites are implemented as simple `Result<Unit>` actions.

Why this design works:

- The UI layer remains agnostic to the source of data—it does not need to know whether data came from the network or local cache.
- Testability improves dramatically, as the repository can be mocked or replaced during testing without affecting the rest of the system.
- Policies such as retry logic, data mapping (e.g., up to 20 ingredients per recipe), and throttling are all centralized in one location.
- Offline behavior is robust and consistent: the repository manages read-through and write-through strategies to Room, allowing users to access and update favorites even without an internet connection.

In summary, the `MealRepository` encapsulates the full lifecycle of meal data in the app. It provides a reliable and maintainable abstraction that unifies multiple data sources, enforces clean contracts, and supports both online and offline usage without complicating the rest of the application.



3 Step 1 — Inventory

3.1 Repository Inventory

MealRepository (TheMealDB + Room).

- **Depends on:** `SearchAPIService`, `FilterAPIService`, `LookupAPIService`, `RandomAPIService` (Retrofit); `MealDao` (Room).
- **Responsibilities:** remote reads; favorites CRUD; mapping `MealResponseItem` → `FavoriteMealEntity`; error shaping into sealed results (no raw Retrofit beyond repo).

Context: The repository is the single entry point for all meal data. It wires multiple Retrofit services (search/filter/lookup/random) to TheMealDB while persisting favorites via `MealDao`. Every call is wrapped in `Result<T>` so ViewModels never deal with raw Retrofit responses or unchecked exceptions. A reactive `favoritesFlow()` (Kotlin Flow) keeps the UI instantly in sync with the local database without manual refresh.
Public surface:

Method	Returns	Notes
`searchMeals(query)`	`Result<List<MealResponseItem»`	GET `search.php?s=`
`filterByCategory(cat)`	`Result<List<MealResponseItem»`	GET `filter.php?c=`
`lookupMeal(id)`	`Result<MealResponseItem>`	GET `lookup.php?i=`
`getRandomMeal()`	`Result<MealResponseItem>`	GET `random.php`
`favoritesFlow()`	`Flow<List<FavoriteMealEntity»`	Live Room stream for UI
`saveFavorite(entity)`	`Result<Unit>`	Insert/Upsert in Room
`removeFavorite(id)`	`Result<Unit>`	Delete in Room

CategoryRepository (TheMealDB categories).

- **Depends on:** `CategoriesAPIService`.
- **Responsibilities:** fetch categories; normalize HTTP/IO errors into `Result`; provide data for category grid and to drive `filter.php?c=`.

Context: This is a focused, lightweight repository for category data. It calls `categories.php` through `CategoriesAPIService` and returns `Result<CategoriesResponse>`. Null bodies or HTTP errors are converted to `Result.failure`, so ViewModels handle categories with the same consistent error model as meals.
Public surface:

Method	Returns	Notes
`getCategories()`	`Result<CategoriesResponse>`	Wraps `categories.php`; null body → `Result.failure`

Implementation (for reference):

```
package com.Mels_Proj.feature.category.domain.data.repository

import com.Mels_Proj.feature.category.domain.data.model.CategoriesResponse
import com.Mels_Proj.shared_component.CategoriesAPIService

class CategoryRepository(private val categoriesApi: CategoriesAPIService) {

    suspend fun getCategories(): Result<CategoriesResponse> {
        return try {
            val response = categoriesApi.getCategories()
            if (response.isSuccessful) {
                response.body()?.let { Result.success(it) } ?: Result.failure(Throwable("Response body is null"))
            } else {
                Result.failure(Throwable("API Error: ${response.code()}"))
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

3.2 API Configuration (File: `API.kt`)

Purpose The file `API.kt` defines the core networking configuration of the Android application. It centralizes how all HTTP requests are built, executed, and logged by combining Retrofit, OkHttp, and Gson into one reusable component. Every MealDB endpoint used by the app is accessed through this configuration, ensuring consistent timeouts, logging, and JSON parsing across all data sources.

Core Implementation The implementation is organized within a `companion object`, which allows all components and services to be accessed statically across the application. Below are the key code segments and their corresponding functions.

1. Logging Interceptor and HTTP Client

```
private val interceptor: HttpLoggingInterceptor
    get() = HttpLoggingInterceptor()
        .apply { level = HttpLoggingInterceptor.Level.BODY }

private val myAppClient: OkHttpClient
    get() = OkHttpClient.Builder()
        .addInterceptor(interceptor)
        .writeTimeout(Constants.connectionTime, TimeUnit.SECONDS)
        .readTimeout(Constants.connectionTime, TimeUnit.SECONDS)
        .connectTimeout(Constants.connectionTime, TimeUnit.SECONDS)
        .build()
```

The `interceptor` prints complete request and response data for each HTTP call, which is useful for debugging API traffic. The `myAppClient` then integrates this interceptor and enforces equal timeout durations for write, read, and connect operations—each set to 60 seconds.

2. Constants and Gson Configuration

```
private object Constants {
    const val baseURL = "https://www.themealdb.com/api/json/v1/1/"
    const val connectionTime: Long = 60
}

private val gson: Gson get() = GsonBuilder().setLenient().create()
```

The `Constants` object defines two immutable values:

- `baseURL` — the main endpoint for TheMealDB API.
- `connectionTime` — the timeout applied to every network call.

The `gson` instance provides lenient JSON parsing, allowing the app to handle slightly irregular responses from TheMealDB API without parsing errors.

3. Retrofit Builder (Shared Instance)

```
private val retrofit: Retrofit by lazy {
    Retrofit.Builder()
        .baseUrl(Constants.baseURL)
        .client(myAppClient)
        .addConverterFactory(GsonConverterFactory.create(gson))
        .build()
}
```

This block creates a single Retrofit instance shared across the application. It uses the configured `OkHttpClient` and the lenient Gson converter. Declaring it with `by lazy` ensures that the Retrofit object is initialized only once—on first use—and reused afterward.

4. API Service Interfaces

```
val searchApiService: SearchAPIService by lazy {
    retrofit.create(SearchAPIService::class.java)
}

val filterApiService: FilterAPIService by lazy {
    retrofit.create(FilterAPIService::class.java)
}

val lookupApiService: LookupAPIService by lazy {
    retrofit.create(LookupAPIService::class.java)
}

val randomApiService: RandomAPIService by lazy {
    retrofit.create(RandomAPIService::class.java)
}

val categoriesApiService: CategoriesAPIService by lazy {
    retrofit.create(CategoriesAPIService::class.java)
}
```

Functional Summary When a repository requests remote data, the service call is made via the shared Retrofit instance, executed by the shared `OkHttpClient`, parsed by the Gson converter, and returned to the repository for mapping and error shaping.

3.3 AI Android Client (Request, Service, Network, Repository)

What this layer does This Android-side client calls the AI backend to generate a MealDB-shaped recipe from free-form ingredients. It consists of:

1. **`AiNetwork`** — Retrofit + OkHttp configuration targeted at the AI base URL (emulator/local or deployed).
2. **`AiRecipeApi`** — Retrofit interface exposing `POST /v1/ai/recipe` (returns a single MealDB-like item).
3. **`AiRecipeRepository`** — convenience facade that cleans input and performs the network call.
4. **`AiGenerateRequest`** — request model sent to the backend.

Design notes

- **Base URL & environments:** use emulator loopback `http://10.0.2.2:8080/` for local, or your deployed HTTPS hostname for staging/prod. In the sample below the base URL is a LAN IP; prefer Build Flavors for this.
- **Timeouts & logging:** client uses 30s connect / 60s read with BASIC logging.
- **Path versioning:** endpoint is `/v1/ai/recipe` across client and server.
- **Error shaping:** wrap results in `Result` and map HTTP errors to meaningful messages.

Key Implementation Details

1. **Request Model — `AiGenerateRequest`.** Represents the payload sent to the backend: a list of ingredients, optional number of servings (default 2), and optional user preference (e.g., dietary restriction).
2. **Network Setup — `AiNetwork`.** Configures Retrofit with OkHttp, enables basic logging, applies timeouts suitable for AI processing, defines the backend base URL, and exposes the `AiRecipeApi`.
3. **Retrofit Interface — `AiRecipeApi`.** Declares the `POST /v1/ai/recipe` call that accepts `AiGenerateRequest` and returns a single `MealResponseItem`.
4. **Repository — `AiRecipeRepository`.** Trims and filters the ingredient list, builds the request, calls the API, and returns the generated recipe to the ViewModel/UI.

Implementation:

```
package com.Mels_Proj.feature.ai_recipe.data.model

data class AiGenerateRequest(
    val ingredients: List<String>,
    val servings: Int? = 2,
    val preference: String? = null
)
```

```
package com.Mels_Proj.feature.ai_recipe.data.remote

import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import java.util.concurrent.TimeUnit

object AiNetwork {
    // LOCAL on emulator:  "http://10.0.2.2:8080/"
    // Deployed (Render):  "https://<your-service>.onrender.com/"
    private const val BASE_URL = "http://192.168.15.66:8080/"

    private val logging = HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BASIC
    }
    private val client = OkHttpClient.Builder()
        .addInterceptor(logging)
        .connectTimeout(30, TimeUnit.SECONDS)
        .readTimeout(60, TimeUnit.SECONDS)
        .build()

    val api: AiRecipeApi by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .client(client)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(AiRecipeApi::class.java)
    }
}
```

```
package com.Mels_Proj.feature.ai_recipe.data.remote

import com.Mels_Proj.feature.ai_recipe.data.model.AiGenerateRequest
import com.Mels_Proj.feature.meal.domain.data.model.MealResponse
import retrofit2.http.Body
import retrofit2.http.POST

interface AiRecipeApi {
    @POST("/v1/ai/recipe")
    suspend fun generateRecipe(@Body body: AiGenerateRequest): MealResponse.MealResponseItem
}
```

```
package com.Mels_Proj.feature.ai_recipe.data

import com.Mels_Proj.feature.ai_recipe.data.model.AiGenerateRequest
import com.Mels_Proj.feature.ai_recipe.data.remote.AiNetwork
import com.Mels_Proj.feature.meal.domain.data.model.MealResponse

class AiRecipeRepository {
    private val api = AiNetwork.api

    suspend fun generate(ingredients: List<String>, servings: Int? = 2): MealResponse.MealResponseItem {
        val clean = ingredients.map { it.trim() }.filter { it.isNotEmpty() }
        return api.generateRecipe(AiGenerateRequest(clean, servings))
    }
}
```

How it works (step-by-step)

1. The user enters free-form ingredients (and optional servings) in the UI.
2. The ViewModel calls `AiRecipeRepository.generate(...)`.
3. The repository trims/removes blanks and builds `AiGenerateRequest`.
4. Retrofit posts to `/v1/ai/recipe` via `AiRecipeApi` and returns a `MealResponseItem`.
5. The ViewModel maps the result to UI state; on error, the UI shows a retry with a friendly message.

3.4 Local Persistence (Room — Database & DAO)

Purpose This layer stores and streams the user's favorite meals on-device. It provides: (1) a singleton `RoomDatabase` to access tables, and (2) a `Dao` with insert/delete and reactive queries via `Flow`.

Database (`MealDatabase`) — what it does

- Declares a Room database with the entity `FavoriteMealEntity` and schema version 1.
- Exposes a single entry point `mealDao()` to retrieve the DAO.
- Ensures a single process-wide instance via `@Volatile`+`synchronized` `getInstance(...)`.

```
package com.Mels_Proj.db

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.Mels_Proj.db.dao.MealDao
import com.Mels_Proj.feature.meal.domain.data.model.FavoriteMealEntity

@Database(entities = [FavoriteMealEntity::class], version = 1)
abstract class MealDatabase: RoomDatabase() {

    abstract fun mealDao(): MealDao

    companion object {
        @Volatile
        private var INSTANCE: MealDatabase? = null

        fun getInstance(context: Context): MealDatabase {

            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        MealDatabase::class.java,
                        "meal_database"
                    ).build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

DAO (`MealDao`) — what it does

- Inserts or replaces a favorite meal; deletes a favorite by entity.
- Streams a single favorite by `mealId` as `Flow<FavoriteMealEntity?>`.
- Streams the entire favorites list as `Flow<List<FavoriteMealEntity»`.
- Uses table/column names from `Constants` to keep SQL aligned with your model.

```
package com.Mels_Proj.db.dao

import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import com.Mels_Proj.feature.meal.domain.data.Constants
import com.Mels_Proj.feature.meal.domain.data.model.FavoriteMealEntity
import kotlinx.coroutines.flow.Flow

@Dao
interface MealDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertFavorite(favoriteMeal: FavoriteMealEntity)

    @Delete
    suspend fun deleteFavorite(favoriteMeal: FavoriteMealEntity)

    @Query("SELECT * FROM ${Constants.FAVORITE_MEALS_TABLE_NAME} WHERE ${Constants.MEAL_ID_COLUMN} = :mealId")
    fun getFavoriteById(mealId: String): Flow<FavoriteMealEntity?>

    @Query("SELECT * FROM ${Constants.FAVORITE_MEALS_TABLE_NAME}")
    fun getAllFavorites(): Flow<List<FavoriteMealEntity>>
}
```

How the app uses it

1. UI toggles a heart: ViewModel calls Repository, which calls `MealDao.insertFavorite(...)` or `deleteFavorite(...)`.
2. Favorites screen subscribes to `MealDao.getAllFavorites()`; any change emits a new list to the UI.
3. Detail screen can observe `getFavoriteById(mealId)` to keep the heart state in sync.

4 ViewModels

4.1 Detail Screen ViewModel (HowToViewModel) — loading, favorites, and DI

Purpose

The `HowToViewModel` owns the state and logic for the meal detail screen. It: (1) loads a meal by `mealId`, preferring local Room when available; (2) fetches from network as a fallback; (3) exposes reactive UI state (`meal`, `loadingState`, `toastMessage`, `isFavorite`); and (4) toggles the favorite status using the repository/DAO.

Dependencies

- **MealRepository** — orchestrates Retrofit calls (search/filter/lookup/random) and Room access (favorites).
- **MealDatabase → MealDao** — Room persistence of favorites and `Flow` queries.
- **Converters** — `toFavoriteEntity` and `toMealResponseItem` provide lossless mapping between network/DB shapes.

Public state (observables)

Exposed as `LiveData` so Fragments can observe safely across lifecycle events:

- `meal: LiveData<MealResponseItem?>` — the currently displayed meal.
- `loadingState: LiveData<Boolean>` — drives the progress indicator.
- `toastMessage: LiveData<String>` — single-shot feedback for errors/notifications (for strict one-off events, consider a `SingleLiveEvent` or exposing a `Channel` as `Flow`).
- `isFavorite: LiveData<Boolean>` — whether the current meal is saved locally.

Load flow: `loadMealDetails(mealId)`

1. Set `loadingState = true`.
2. Query Room via `mealRepository.getFavoriteMeal(mealId)?.first()`.
   - If found: map with `toMealResponseItem()`, publish to `meal`, set `isFavorite = true`, stop loading.
   - If not found: set `isFavorite = false` and call network `fetchMealFromApi(mealId)`.
3. Network path:
   - `mealRepository.getMealById(id)` returns `Result<MealResponse>`.

   - `onSuccess`: if response has items, publish `first()` into `meal`; else, publish a user-friendly toast.
   - `onFailure`: publish API error into `toastMessage`.

4. Set `loadingState = false`.

Favorite toggle: `onFavoriteClicked()`

1. Read current `meal`; map to `FavoriteMealEntity` via `toFavoriteEntity()`. If null, return.
2. If `isFavorite = true`: call `mealRepository.removeFavorite(entity)`, show "Removed from favorites", set `isFavorite = false`.
3. Else: call `mealRepository.addFavorite(entity)`, show "Added to favorites", set `isFavorite = true`.

Threading & lifecycle

All work runs in `viewModelScope`, ensuring cancellation on `ViewModel` clear. Room `Flow` is bridged using `first()` for a one-time snapshot.

Error handling

Network and parsing failures are mapped to user-friendly `toastMessage` text. For production, prefer a sealed UI state (`Success/Loading/Empty/Error`) and surface errors inline (snackbar/inline state) rather than only toasts.

Construction & DI (HowToModule, HowToViewModelFactory)

- **HowToModule** wires `MealRepository` with:
  - `MealDao` from `MealDatabase.getInstance(application).mealDao()`
  - The shared Retrofit services exposed by `API` (`searchApiService`, `filterApiService`, `lookupApiService`, `randomApiService`)
- **HowToViewModelFactory** constructs `HowToViewModel` with the module-provided repository to enable injection without a DI framework.

Why this design works

- **Fast path first:** Local DB is prioritized for instant rendering and offline support.
- **Single source of truth:** Repository centralizes mapping, error shaping, and data policy.
- **UI simplicity:** Fragment only observes `LiveData` and triggers intents (`load/favorite`), keeping view code dumb.
- **Upgradeable:** Module/Factory can be replaced by Hilt with minimal code churn later.

4.2 Home Screen ViewModel (HomeViewModel) — filters, banner, and favorites overlay

Purpose

Owns the Home screen's dynamic state and orchestration: (1) loads a curated set of random meals and cycles the banner image, (2) applies *three* user-driven filters (category, first letter, search term), (3) exposes a live favorites overlay sourced from Room, (4) fetches categories for the chip list, (5) toggles favorites (ensuring a complete object before persisting).

Public state (observables)

Name	**Type**	**What UI uses it for**
`meals`	`LiveData<List<MealResponseItem»`	Carousel data (snap list).
`allFavorites`	`LiveData<List<FavoriteMealEntity»`	Overlay heart state in the carousel.
`categories`	`LiveData<List<CategoryResponseItem»`	Category chip row.
`bannerImageUrl`	`LiveData<String>`	Slowly panning hero image.
`foodLoadingState`	`LiveData<Boolean>`	Meals shimmer/progress.
`catLoadingState`	`LiveData<Boolean>`	Categories shimmer/progress.
`toastMessage`	`LiveData<String>`	One-off toasts for user feedback.

Core behaviors (how it works)

Startup and favorites stream

- Subscribes to `MealDao.getAllFavorites()` and mirrors it into `allFavorites` so the heart overlay is always in sync.

`getRandomMeals(forceRefresh)`

- Parallelizes 10 `random.php` calls via `async/awaitAll`; takes the first 5 successes for a stable, small carousel.
- Starts *banner cycling* with a main-thread `Handler` that swaps `bannerImageUrl` every 10s.
- Memoizes the initial list in `initialMealsList` (fast return on subsequent entries unless `forceRefresh=true`).

Filtering model

- Three inputs: `currentFilter` (category), `currentLetter` (A–Z), *optional* search term.
- The method `applyFilters(searchTerm)` chooses the best upstream source:
  1. If no filters: return `initialMealsList`.
  2. If category set: call `filter.php?c=`.
  3. Else if letter set: call `search by first letter`.
  4. Else if search term: call `search.php?s=`.
- Then refines in-memory (e.g., apply letter+search on the category result). Empty results surface a friendly toast.

`toggleFavorite(meal)`

- Removes if already in `allFavorites`.
- For adds: if instructions are missing (common in `filter.php` responses), fetches the full meal via `lookup.php?i=` to persist a *complete* entity, then saves in Room.

`getCategories()`

- Wraps the categories call in a `Result`; drives `catLoadingState` and normalizes errors to toasts.

Lifecyle & cleanup Cancels banner cycling in `onCleared()` by removing callbacks from the `Handler`.

Construction & DI (HomeModule, HomeViewModelFactory)

- **HomeModule** wires a `MealRepository` (Retrofit + Room) and a `CategoryRepository`.
- **HomeViewModelFactory** creates `HomeViewModel` with those dependencies so the Fragment stays DI-framework-agnostic.

Abridged implementation (for reference)

```
class HomeViewModel(
  application: Application,
  private val mealRepository: MealRepository,
  private val categoryRepository: CategoryRepository
) : AndroidViewModel(application) {

  // LiveData (meals, allFavorites, categories, bannerImageUrl, ...)

  init { observeFavorites() }

  private fun observeFavorites() = viewModelScope.launch {
    mealRepository.getAllFavorites()?.collectLatest { _allFavorites.postValue(it) }
  }

  fun getRandomMeals(forceRefresh: Boolean) = viewModelScope.launch {
    if (initialMealsList.isNotEmpty() && !forceRefresh) { _meals.postValue(initialMealsList); return@launch }
    _foodLoadingState.postValue(true)
    val successful = (1..10).map { async { mealRepository.getRandomMeal() } }
      .awaitAll()
      .mapNotNull { it.getOrNull()?.mealList?.firstOrNull() }
      .take(5)
    if (successful.isNotEmpty()) { startBannerCycling(successful); initialMealsList = successful; _meals.
↪ postValue(successful) }
    else onSomeError("Failed to load random meals")
    _foodLoadingState.postValue(false)
  }

  fun filterByCategory(category: String) { currentFilter = if (category == currentFilter) null else category;
↪ applyFilters() }
  fun listMealsByFirstLetter(letter: String) { currentLetter = letter; applyFilters() }
  fun searchMeal(name: String) { applyFilters(searchTerm = name) }
  fun clearLetterFilter() { currentLetter = null; applyFilters() }

  private fun applyFilters(searchTerm: String? = null) = viewModelScope.launch {
    _foodLoadingState.postValue(true)
    // choose upstream source based on currentFilter/currentLetter/searchTerm ...
    _foodLoadingState.postValue(false)
  }

  fun toggleFavorite(meal: MealResponse.MealResponseItem) = viewModelScope.launch {
    val isFav = _allFavorites.value?.any { it.idMeal == meal.idMeal } == true
    if (isFav) meal.toFavoriteEntity()?.let { mealRepository.removeFavorite(it) }
    else {
      if (meal.strInstructions.isNullOrBlank())
        meal.idMeal?.let { id -> mealRepository.getMealById(id).onSuccess { it.mealList.firstOrNull()?.
↪ toFavoriteEntity()?.let(mealRepository::addFavorite) }
          .onFailure { onSomeError("Failed to fetch full meal details.") } }
      else meal.toFavoriteEntity()?.let(mealRepository::addFavorite)
    }
  }

  override fun onCleared() { super.onCleared(); if (::homeBannerRunnable.isInitialized) homeBannerHandler.
↪ removeCallbacks(homeBannerRunnable) }
}
```

Why this VM design works Single source of truth lives in repositories; the VM composes server + DB streams into simple UI observables and guarantees the Favorites overlay never drifts.

4.3 Favorites ViewModel (FavoritesViewModel) — streaming list & search

Purpose

Expose a reactive list of locally saved meals and provide a lightweight, client-side name filter.

Public state

`favoritesList: LiveData<List<FavoriteMealEntity>>` — the *currently displayed* list (either full stream or filtered).

Behavior

- **`loadFavorites()`** collects `MealDao.getAllFavorites()` into an internal `allFavorites` cache and mirrors it into `favoritesList`.
- **`searchFavorites(query)`** performs a case-insensitive filter on `strMeal`. Blank query restores the stream.

Abridged implementation (for reference)

```
class FavoritesViewModel(
  application: Application,
  private val mealRepository: MealRepository
) : AndroidViewModel(application) {

  private var allFavorites: List<FavoriteMealEntity> = emptyList()
  private val _favoritesList = MutableLiveData<List<FavoriteMealEntity>>()
  val favoritesList: LiveData<List<FavoriteMealEntity>> get() = _favoritesList

  fun loadFavorites() = viewModelScope.launch {
    mealRepository.getAllFavorites()?.collectLatest { favorites ->
      allFavorites = favorites
      _favoritesList.postValue(allFavorites)
    }
  }

  fun searchFavorites(query: String) {
    if (query.isBlank()) _favoritesList.postValue(allFavorites)
    else _favoritesList.postValue(allFavorites.filter { it.strMeal?.contains(query, true) == true })
  }
}
```

Construction & DI (FavoritesModule, FavoritesViewModelFactory)

The module builds a `MealRepository` with Room + Retrofit; the factory injects it into the VM. Keeping it manual makes migration to Hilt trivial.

4.4 AI Generate ViewModel (AiRecipeViewModel) — request pipeline & UI state

Purpose

Owns the AI generation flow: normalize the user's ingredient input, call the AI backend via `AiRecipeRepository`, and expose a simple UI state machine for the Fragment.

Public state

`state: StateFlow<AiUiState>` with four states:

- **Idle** — initial screen, nothing generated yet.
- **Loading** — network call in flight.
- **Success(meal)** — MealDB-shaped item ready to render.
- **Error(message)** — user-friendly failure reason.

Behavior

1. **Input parsing:** split on commas/semicolons/newlines, trim, drop blanks; error if empty.
2. **Async call:** switch to **Loading**, invoke `repo.generate(...)` in `viewModelScope`.
3. **Result mapping:** on success publish **Success(meal)**; on failure publish **Error(...)**.

Implementation (full code)

```kotlin
package com.Mels_Proj.feature.ai_recipe.presentation

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.Mels_Proj.feature.ai_recipe.data.AiRecipeRepository
import com.Mels_Proj.feature.meal.domain.data.model.MealResponse
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.launch

sealed class AiUiState {
    object Idle : AiUiState()
    object Loading : AiUiState()
    data class Success(val meal: MealResponse.MealResponseItem) : AiUiState()
    data class Error(val message: String) : AiUiState()
}

class AiRecipeViewModel(
    private val repo: AiRecipeRepository = AiRecipeRepository()
) : ViewModel() {
    private val _state = MutableStateFlow<AiUiState>(AiUiState.Idle)
    val state: StateFlow<AiUiState> = _state

    fun generateFromInput(input: String, servings: Int? = 2) {
        val ingredients = input.split(',', ';', '\n')
            .map { it.trim() }
            .filter { it.isNotEmpty() }
        if (ingredients.isEmpty()) {
            _state.value = AiUiState.Error("Please enter at least one ingredient.")
            return
        }
        _state.value = AiUiState.Loading
        viewModelScope.launch {
            try {
                val meal = repo.generate(ingredients, servings)
                _state.value = AiUiState.Success(meal)
            } catch (e: Exception) {
                _state.value = AiUiState.Error(e.message ?: "Something went wrong")
            }
        }
    }
}
```

ViewModel layer invariants (quick checklist)

- **No I/O in UI/adapters:** all network/DB work is inside repositories; VMs only orchestrate.
- **Main-safe posting:** `postValue` used from background, `setValue` from main; all adapter updates happen on the main thread.
- **Lifecycles respected:** long-running handlers/callbacks are removed in `onCleared()`.

5 Retrofit services and repositories — exact responsibilities (Android side)

`SearchAPIService` (Retrofit interface)

- **Declares:**

```
@GET("search.php")
suspend fun search(@Query("s") q: String): Response<MealListResponse>
```

- **Responsibility:** map HTTP responses to data classes; the repository handles Response → Result conversion.

`FilterAPIService`

- **Declares:**

```
@GET("filter.php")
suspend fun filter(@Query("c") cat: String): Response<FilterResponse>
```

- **Responsibility:** fast category-based lists; repository may call `lookup` for details when needed.

`LookupAPIService`

- **Declares:**

```
@GET("lookup.php")
suspend fun lookup(@Query("i") id: String): Response<MealLookupResponse>
```

- **Responsibility:** provide the full meal object including ingredient slots; repository validates and maps to domain.

`RandomAPIService`

- **Declares:**

```
@GET("random.php")
suspend fun random(): Response<MealListResponse>
```

- **Responsibility:** one-off random meal; repository caches/limits calls as needed.

`AiRecipeApi`

- **Declares:**

```
@POST("v1/ai/recipe")
suspend fun generate(@Body req: AiGenerateRequest): Response<MealDBItem>
```

- **Responsibility:** send normalized requests and receive MealDB-shaped responses; repository handles network errors and maps them to sealed results.

6 Network Response Models (Android) — `MealResponse` & `CategoriesResponse`

Purpose These models mirror TheMealDB JSON so Retrofit+Gson can deserialize responses into strongly-typed Kotlin. They are intentionally *thin*; mapping to Room/UI lives in repositories.

`MealResponse` (search/lookup/random)

- Top-level payload wraps a list under the `"meals"` key.
- Each `MealResponseItem` contains base metadata plus up to 20 ingredient/measure slots.
- Most fields are nullable because TheMealDB often omits keys.

Kotlin (abridged):

```
package com.Mels_Proj.feature.meal.domain.data.model
import com.google.gson.annotations.SerializedName

class MealResponse(@SerializedName("meals") val mealList: List<MealResponseItem>){
    data class MealResponseItem(
        val dateModified: Any? = null,
        val idMeal: String? = null,
        val strArea: String? = null,
        val strCategory: String? = null,
        val strCreativeCommonsConfirmed: Any? = null,
        val strImageSource: Any? = null,
        val strIngredient1: String? = null,
        val strIngredient10: String? = null,
        val strIngredient11: String? = null,
        val strIngredient12: String? = null,
        val strIngredient13: String? = null,
        val strIngredient14: String? = null,
        val strIngredient15: String? = null,
        val strIngredient16: String? = null,
        val strIngredient17: String? = null,
        val strIngredient18: String? = null,
        val strIngredient19: String? = null,
        val strIngredient2: String? = null,
        val strIngredient20: String? = null,
        val strIngredient3: String? = null,
        val strIngredient4: String? = null,
        val strIngredient5: String? = null,
        val strIngredient6: String? = null,
        val strIngredient7: String? = null,
        val strIngredient8: String? = null,
        val strIngredient9: String? = null,
        val strInstructions: String? = null,
        val strMeal: String? = null,
        val strMealAlternate: Any? = null,
        val strMealThumb: String? = null,
        val strMeasure1: String? = null,
        val strMeasure10: String? = null,
        val strMeasure11: String? = null,
        val strMeasure12: String? = null,
        val strMeasure13: String? = null,
        val strMeasure14: String? = null,
        val strMeasure15: String? = null,
        val strMeasure16: String? = null,
        val strMeasure17: String? = null,
        val strMeasure18: String? = null,
        val strMeasure19: String? = null,
        val strMeasure2: String? = null,
        val strMeasure20: String? = null,
        val strMeasure3: String? = null,
        val strMeasure4: String? = null,
        val strMeasure5: String? = null,
        val strMeasure6: String? = null,
        val strMeasure7: String? = null,
        val strMeasure8: String? = null,
        val strMeasure9: String? = null,
        val strSource: String? = null,
        val strTags: String? = null,
        val strYoutube: String? = null
    )
}
```

Field	**Type**	**Usage**
`idMeal`	`String?`	Primary identifier for detail/favorites.
`strMeal`	`String?`	Title in lists/details.
`strMealThumb`	`String?`	Image URL (Glide).
`strCategory`, `strArea`	`String?`	Metadata chips/badges in UI.
`strInstructions`	`String?`	Main instruction text.
`strIngredient1..20`/`strMeasure1..20`	`String?`	1..20 aligned pairs; ignore null/blank.

Key fields (how the app uses them)

Nullability and mapping notes

- Treat null/blank ingredient or measure as absent; never show empty rows.
- Pair by index: `(strIngredientN, strMeasureN)`.
- Persist only what's needed in Room for Favorites.

`CategoriesResponse` (category list)

- Top-level payload wraps a list under the `"categories"` key.
- Each item contains ID, display name, thumbnail, and long description.

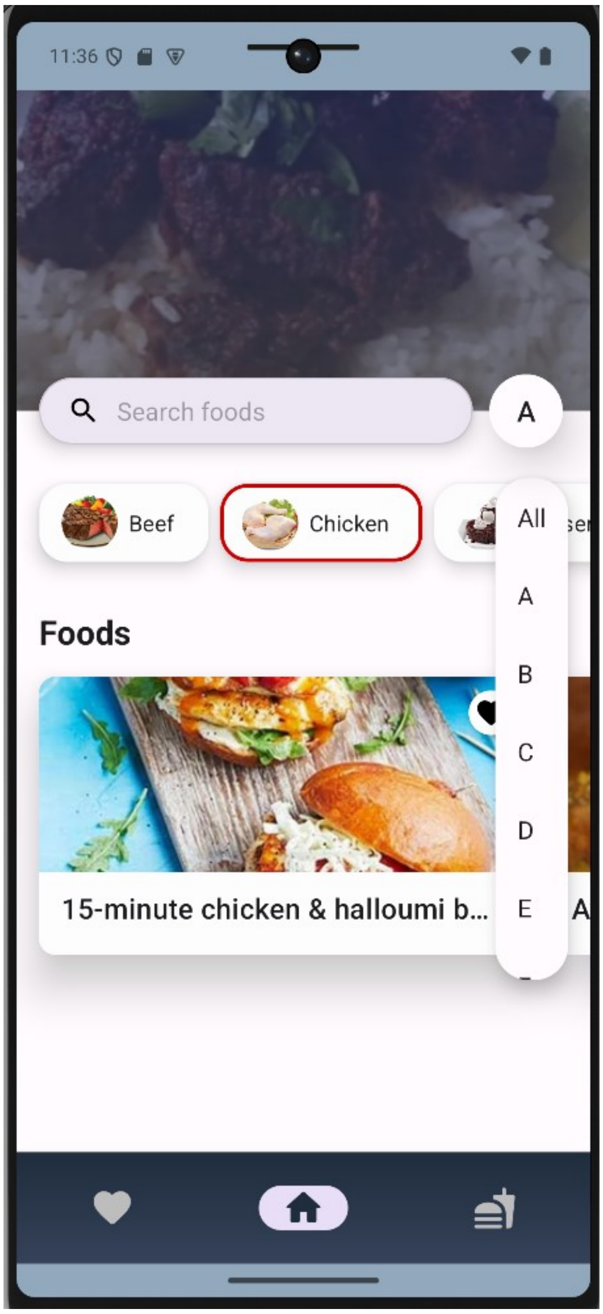


Figure 1: Home — category chips highlighted and used to filter the meal list.

Kotlin:

```kotlin
package com.Mels_Proj.feature.category.domain.data.model
import com.google.gson.annotations.SerializedName

class CategoriesResponse(@SerializedName("categories") val categoryList: List<CategoryResponseItem>) {
    data class CategoryResponseItem(
        val idCategory: String?,
               val strCategory: String?,
        val strCategoryThumb: String?,
        val strCategoryDescription: String?
    )
}
```

Field usage

Field	Type	Usage
`idCategory`	`String?`	Optional key; rarely needed in UI.
`strCategory`	`String?`	Display name; passed to `filter.php?c=`.
`strCategoryThumb`	`String?`	Grid/list thumbnail.
`strCategoryDescription`	`String?`	Optional long text on detail.

Repository → UI guidance

- **Meals:** Repositories map `MealResponseItem` → domain/`FavoriteMealEntity`; skip empty slots, trim whitespace.
- **Categories:** `CategoryRepository.getCategories()` returns `Result<CategoriesResponse>`; UI renders `strCategory` and thumb. Tapping triggers `filter.php?c={strCategory}`.
- **Errors:** Null bodies and HTTP codes are normalized to `Result.failure` in repositories.

7 AI Backend — responsibilities and pipeline

Purpose

- Convert free-form ingredient lists (e.g., `"eggs, tomatoes, onion"`) into a clean, validated MealDB-shaped JSON object the Android app can render with zero special-case logic.
- Keep the LLM and provider keys server-side; enforce deterministic prompt templates, unit conventions (metric), and strict JSON-only output to guarantee parseability.

LLM Provider (AvalAI.ir)

We use **AvalAI** as the LLM provider for the server–to–server completion call that powers `/v1/ai/recipe`. The API key is kept *only* on the backend (never in the Android client), and the FastAPI service invokes AvalAI's HTTP endpoint, parses the result, validates it against our schema, and then maps it into MealDB slots (1..20). For setup and request examples, see AvalAI's Quick Start documentation (FA): `https://docs.avalai.ir/fa/?id=%D8%B4%D8%B1%D9%88%D8%B9-%D8%B3%D8%B1%DB%8C%D8%B9`.

Backend wiring (where it fits) Below, `generate_mealdb_json(...)` delegates the text generation to a small helper that calls AvalAI. Replace the placeholder endpoint/model with the values from the docs.

```python
import os, httpx, json

def call_llm_via_avalai(prompt: str, timeout: float = 20.0) -> str:
    api_key = os.getenv("AVALAI_API_KEY")
    if not api_key:
        raise RuntimeError("Missing AVALAI_API_KEY")

    # TODO: set from AvalAI docs / dashboard:
    endpoint = os.getenv("AVALAI_ENDPOINT", "https://<avalai-endpoint>/v1/chat/completions")
    model = os.getenv("AVALAI_MODEL", "<model_name>")

    headers = {
        # Some providers use "Authorization: Bearer <key>", others "X-API-Key: <key>"  use AvalAI's spec.
        "Authorization": f"Bearer {api_key}",
        "Content-Type": "application/json",
    }
    payload = {
        "model": model,
        "temperature": 0.2,
        "messages": [
            {"role": "system", "content": "Return only JSON that matches the requested schema."},
            {"role": "user", "content": prompt}
        ]
    }

    with httpx.Client(timeout=timeout) as client:
        r = client.post(endpoint, headers=headers, json=payload)
        r.raise_for_status()
        data = r.json()
        # Adjust extraction to AvalAI's response shape per docs:
        content = data["choices"][0]["message"]["content"]
        return content

def generate_mealdb_json(req: GenerateRequest) -> dict:
    # Build deterministic prompt describing the MealDB schema and the user's ingredients/servings...
    prompt = build_prompt_from_request(req)
    raw = call_llm_via_avalai(prompt)
    obj = json.loads(raw)  # validate/repair if needed
    return map_core_recipe_to_mealdb(obj)  # fills strIngredient1..20 / strMeasure1..20
```

API contract (recap)

```
POST /v1/ai/recipe
Request: { ingredients: [..], servings: 1..20, preference?: "vegetarian"|"vegan"|... }
Response:
  MealDB-compatible JSON (
    idMeal, strMeal,
    strIngredient1..20, strMeasure1..20,
    strInstructions, strMealThumb, ...
  )
```

Server-side pipeline (detailed step-by-step)

1. **Input validation:** Validate the JSON body (1..20 ingredients, servings in range). Return `400 BAD_INPUT`.
2. **Normalization:** Trim/canonicalize ingredients; allow minimal pantry additions.
3. **Prompt construction:** Deterministic prompt; JSON-only; metric units; max 20 slots.
4. **LLM call (text):** Low temperature; handle timeouts; map provider errors to `502 LLM_ERROR`.
5. **Parse & schema validation:** Enforce schema; if invalid, run a targeted repair loop.
6. **Optional image:** Attempt image generation; if it fails, continue without blocking the recipe.
7. **Map to MealDB slots:** Flatten ingredients/measures to `strIngredient1..20/strMeasure1..20`.
8. **Finalize:** Unique `idMeal`, readable instructions, HTTPS image URL.
9. **Return 200** with a MealDB-compatible JSON object.

Error codes and HTTP status mapping

- **BAD_INPUT** (400): client must fix input.
- **LLM_ERROR** (502/503): provider failures; retry allowed.
- **IMAGE_ERROR** (200 with partial data): render without image.

- **INTERNAL** (500): unexpected server failure.
- **RATE_LIMIT** (429): respect `Retry-After`.

8 FastAPI Reference Implementation (server-side Python)

Purpose Runnable, minimal server that enforces the contract, validates inputs, calls the LLM layer (stubbed), maps to MealDB fields, and returns JSON. Replace the `generate_mealdb_json` stub with your real LLM pipeline.

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field, conlist, conint, HttpUrl
from uuid import uuid4
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI(title="AI Recipe API", version="1.0.0")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://yourapp.example", "http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

class GenerateRequest(BaseModel):
    ingredients: conlist(str, min_items=1, max_items=20)
    servings: conint(ge=1, le=20) = 2
    preference: str | None = None

class MealDBItem(BaseModel):
    idMeal: str
    strMeal: str
    strCategory: str | None = None
    strArea: str | None = None
    strInstructions: str
    strMealThumb: HttpUrl | None = None
    # up to 20 ingredient/measure slots
    # (define dynamically at runtime when building response)

def to_slots(ingredients: list[str]) -> dict:
    out = {}
    for i, ing in enumerate(ingredients[:20], start=1):
        out[f"strIngredient{i}"] = ing
        out[f"strMeasure{i}"] = ""  # or inferred measure
    return out

def generate_mealdb_json(req: GenerateRequest) -> dict:
    # --- STUB: replace with your LLM pipeline ---
    title = "AI Pantry Bowl"
    instructions = (
        "1) Prep ingredients.\n"
        "2) Cook aromatics.\n"
        "3) Add mains, season, and simmer.\n"
        "4) Taste and serve."
    )
    base = {
        "idMeal": f"ai-{uuid4()}",
        "strMeal": title,
        "strCategory": req.preference or "AI",
        "strArea": "Fusion",
        "strInstructions": instructions,
        "strMealThumb": None,
    }
    base.update(to_slots([s.strip().title() for s in req.ingredients if s.strip()]))
    return base

@app.post("/v1/ai/recipe")
def generate_recipe(req: GenerateRequest):
    try:
        data = generate_mealdb_json(req)
        return data
    except ValueError as e:
        raise HTTPException(status_code=400, detail={"code":"BAD_INPUT","message":str(e)})
    except TimeoutError:
        raise HTTPException(status_code=502, detail={"code":"LLM_ERROR","message":"Upstream timeout"})
    except Exception:
        raise HTTPException(status_code=500, detail={"code":"INTERNAL","message":"Unexpected error"})
```

9 JSON Contracts

Request: `POST /v1/ai/recipe`

```
{
  "ingredients": ["egg", "broccoli", "olive oil"],  // 1..20 items, trimmed
  "servings": 2,                                    // 1..20
  "preference": "vegetarian"                        // optional
}
```

Response: MealDB-compatible item

```
{
  "idMeal": "ai-7d1a0f94-2ea8-4c3e-9d4f-2b8e7a8c1a2a",
  "strMeal": "Broccoli & Egg Stir-Fry",
  "strCategory": "Vegetarian",
  "strArea": "Fusion",
  "strInstructions": "1) Blanch broccoli... 2) Scramble eggs... 3) Combine...",
  "strMealThumb": "https://cdn.example.com/ai/broccoli-egg.jpg",
  "strIngredient1": "Egg",              "strMeasure1": "2",
  "strIngredient2": "Broccoli florets", "strMeasure2": "200 g",
  "strIngredient3": "Olive oil",        "strMeasure3": "1 tbsp"
  // ... up to strIngredient20 / strMeasure20
}
```

Error envelope (stable, minimal)

```
{
  "code": "BAD_INPUT" | "LLM_ERROR" | "IMAGE_ERROR" | "INTERNAL",
  "message": "human-readable summary",
  "details": { "requestId": "1b2c..." }  // no stack traces in production
}
```

10 Step-by-Step Build Order (reference)

1. FastAPI scaffold, typed models, pinned deps, `/healthz`, CORS allowlist.
2. Prompt pipeline with deterministic template; schema validation/repair loop.
3. Mapping tests (1..20 slots, truncation, nulls).
4. Deploy with logging/metrics; redaction in logs.
5. Android flavors, centralized Retrofit, Hilt DI, AI repository.
6. UI wiring; loading/empty/error states; one instrumentation happy-path.

11 User Interface (Activities & Fragments)

Why Fragments (quick context) Modular screens, lifecycle-aware views, and Navigation integration. The app ships three surfaces: **Home** (fragment), **Detail** (activity), **Favorites** (fragment). Below—in that order.

11.1 Home (Fragment)

Purpose Landing screen with categories carousel, meals carousel (snap), alphabet scroller, banner image, search, and quick favorite toggle.

```
package com.Mels_Proj.feature.home_screen.presentation.ui

import android.content.Intent
import android.content.res.ColorStateList
import android.os.Bundle
import android.util.TypedValue
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.annotation.AttrRes
import androidx.appcompat.widget.SearchView
import androidx.core.content.ContextCompat
import androidx.fragment.app.Fragment
import androidx.lifecycle.ViewModelProvider
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.LinearSnapHelper
import androidx.recyclerview.widget.RecyclerView
import com.bumptech.glide.Glide
import com.bumptech.glide.load.resource.drawable.DrawableTransitionOptions
import com.Mels_Proj.R
import com.Mels_Proj.databinding.FragmentHomeBinding
import com.Mels_Proj.feature.home_screen.presentation.ui.adapter.CategoryAdapter
import com.Mels_Proj.feature.home_screen.presentation.ui.adapter.AlphabetAdapter
import com.Mels_Proj.feature.home_screen.presentation.viewmodel.HomeViewModel
import com.Mels_Proj.feature.home_screen.presentation.viewmodel.HomeViewModelFactory
import com.Mels_Proj.feature.detail_screen.presentation.ui.DetailScreen
import com.Mels_Proj.feature.home_screen.presentation.ui.adapter.MealAdapter
import kotlin.random.Random

fun View.setVisible(isVisible: Boolean) {
    visibility = if (isVisible) View.VISIBLE else View.INVISIBLE
}

class HomeFragment : Fragment() {

    private lateinit var binding: FragmentHomeBinding
    private lateinit var homeViewModel: HomeViewModel
    private lateinit var categoryAdapter: CategoryAdapter
    private lateinit var mealAdapter: MealAdapter
    private lateinit var alphabetAdapter: AlphabetAdapter

    private val maxHorizontalTranslation = 200f
    private val maxVerticalTranslation = 110f

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View
↪   {
        binding = FragmentHomeBinding.inflate(layoutInflater); return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupViewModel(); setupAdapters(); setupListeners(); setupObservers(); callAPI()
    }

    private fun getThemeColor(@AttrRes attrRes: Int): Int {
        val typedValue = TypedValue()
        requireContext().theme.resolveAttribute(attrRes, typedValue, true)
        return typedValue.data
    }

    private fun setupViewModel() {
        val factory = HomeViewModelFactory(requireActivity().application)
        homeViewModel = ViewModelProvider(this, factory)[HomeViewModel::class.java]
    }

    private fun setupAdapters() {
        categoryAdapter = CategoryAdapter { category ->
            category.strCategory?.let {
                homeViewModel.filterByCategory(it)
                binding.rvMeals.smoothScrollToPosition(0)
                binding.searchView.setQuery("", false)
            }
        }
        binding.rvCategories.adapter = categoryAdapter
        binding.rvCategories.layoutManager =
            LinearLayoutManager(requireContext(), LinearLayoutManager.HORIZONTAL, false)
        binding.rvCategories.setHasFixedSize(true)

        mealAdapter = MealAdapter(
            onItemClick = { meal ->
                Intent(requireContext(), DetailScreen::class.java).also {
                    it.putExtra("mealID", meal.idMeal); startActivity(it)
                }
            },
            onFavoriteClick = { meal -> homeViewModel.toggleFavorite(meal) }
        )
        mealAdapter.stateRestorationPolicy =
            RecyclerView.Adapter.StateRestorationPolicy.PREVENT_WHEN_EMPTY

        binding.rvMeals.adapter = mealAdapter
        binding.rvMeals.layoutManager =
            LinearLayoutManager(requireContext(), LinearLayoutManager.HORIZONTAL, false)
        binding.rvMeals.setHasFixedSize(true)
        LinearSnapHelper().attachToRecyclerView(binding.rvMeals)

        alphabetAdapter = AlphabetAdapter { letter ->
            binding.btnSearchLetter.animate().scaleX(1.2f).scaleY(1.2f).setDuration(150).withEndAction {
                binding.btnSearchLetter.animate().scaleX(1f).scaleY(1f).setDuration(150).start()
            }.start()

            val surfaceColor = getThemeColor(com.google.android.material.R.attr.colorSurface)
            if (letter == "All") {
                homeViewModel.clearLetterFilter()
                binding.btnSearchLetter.text = "A"
                binding.btnSearchLetter.backgroundTintList = ColorStateList.valueOf(surfaceColor)
            } else {
                binding.searchView.setQuery("", false)
                homeViewModel.listMealsByFirstLetter(letter)
                binding.btnSearchLetter.text = letter
                binding.btnSearchLetter.backgroundTintList =
                    ColorStateList.valueOf(ContextCompat.getColor(requireContext(), R.color.vibrant_purple))
            }
            binding.alphabetScrollerContainer.visibility = View.GONE
        }
        binding.rvAlphabetScroller.adapter = alphabetAdapter
    }

    private fun setupListeners() {
        binding.btnSearchLetter.setOnClickListener {
            val isVisible = binding.alphabetScrollerContainer.visibility == View.VISIBLE
            if (!isVisible) binding.rvAlphabetScroller.scheduleLayoutAnimation()
            binding.alphabetScrollerContainer.visibility = if (isVisible) View.GONE else View.VISIBLE
        }
        binding.searchView.setOnQueryTextListener(object : SearchView.OnQueryTextListener {
            override fun onQueryTextSubmit(query: String?): Boolean {
                binding.searchView.clearFocus(); homeViewModel.searchMeal(query.toString()); return true
            }
            override fun onQueryTextChange(newText: String?): Boolean {
                if (newText.isNullOrEmpty()) homeViewModel.searchMeal(newText.toString())
                return false
            }
        })
    }

    private fun setupObservers() {
        homeViewModel.bannerImageUrl.observe(viewLifecycleOwner) { url ->
            Glide.with(this).load(url).transition(DrawableTransitionOptions.withCrossFade()).into(binding.
↪ ivBanner)
            binding.ivBanner.animate().cancel()
            binding.ivBanner.translationX = 0f; binding.ivBanner.translationY = 0f
            binding.ivBanner.animate()?.apply {
                duration = 10000
                translationX(Random.nextDouble(-200.0, 200.0).toFloat())
                translationY(Random.nextDouble(-110.0, 110.0).toFloat())
            }?.start()
        }
        homeViewModel.categories.observe(viewLifecycleOwner) {
            categoryAdapter.updateData(it); binding.rvCategories.smoothScrollToPosition(0)
        }
        homeViewModel.meals.observe(viewLifecycleOwner) {
            mealAdapter.updateData(it); binding.rvMeals.smoothScrollToPosition(0)
        }
        homeViewModel.allFavorites.observe(viewLifecycleOwner) { favorites ->
            mealAdapter.updateFavorites(favorites.mapNotNull { it.idMeal }.toSet())
        }
        homeViewModel.catLoadingState.observe(viewLifecycleOwner) { isLoading ->
            binding.progCars.setVisible(isLoading); binding.rvCategories.setVisible(!isLoading)
        }
        homeViewModel.foodLoadingState.observe(viewLifecycleOwner) { isLoading ->
            binding.progFoods.setVisible(isLoading); binding.rvMeals.setVisible(!isLoading)
        }
        homeViewModel.toastMessage.observe(viewLifecycleOwner) {
            Toast.makeText(requireContext(), it, Toast.LENGTH_LONG).show()
        }
    }

    private fun callAPI(forceRefresh: Boolean = false) {
        binding.rvMeals.visibility = View.INVISIBLE
        binding.rvCategories.visibility = View.INVISIBLE
        homeViewModel.getCategories(forceRefresh)
        homeViewModel.getRandomMeals(forceRefresh)
    }
}
```

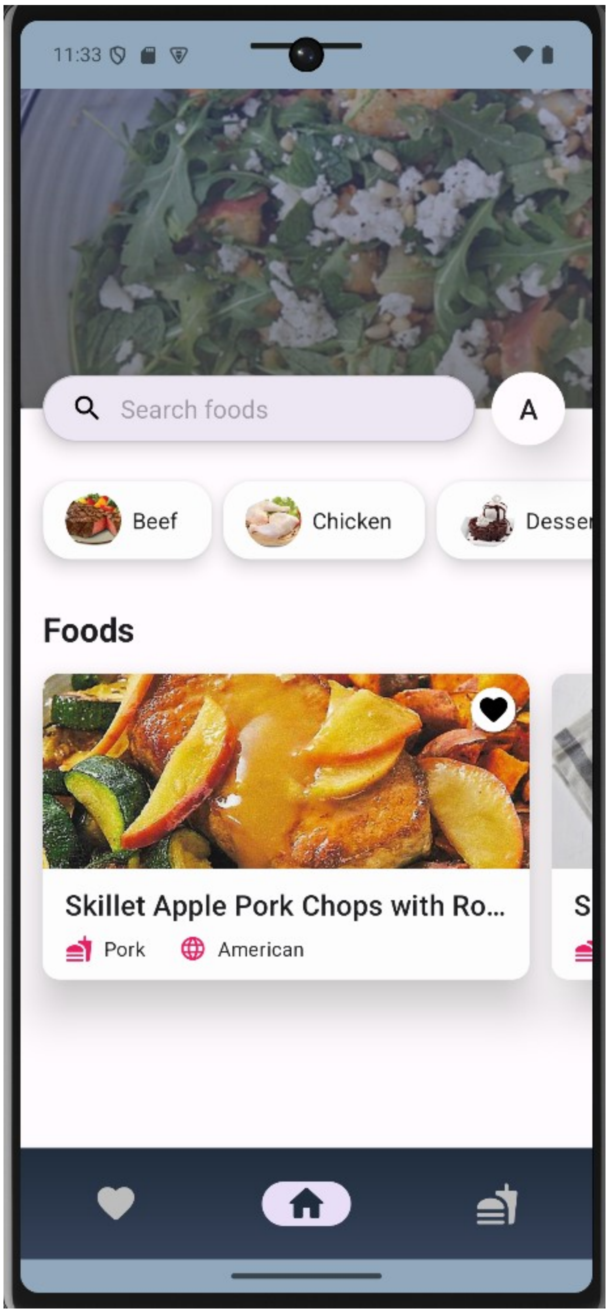


Figure 2: Home screen — categories, search bar, and meals carousel.

Home responsibilities Search feeds the Meals carousel; Categories and Alphabet are mutually exclusive filters that reset each other; Favorites overlay is derived from Room and pushed into the adapter via `updateFavorites`.

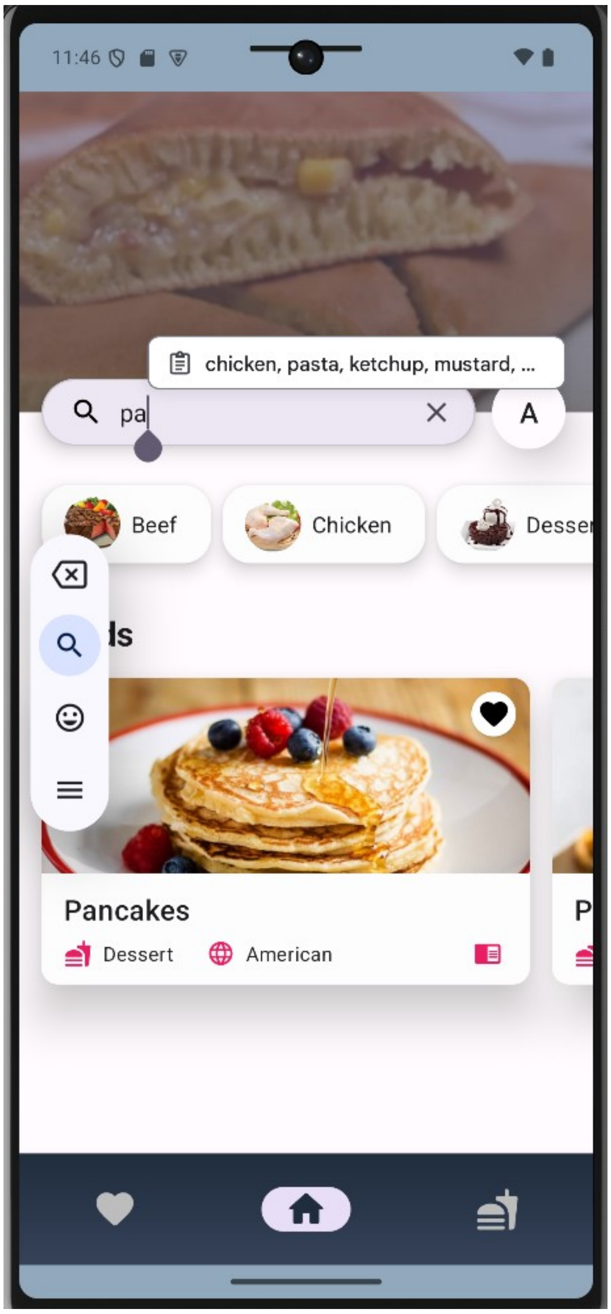
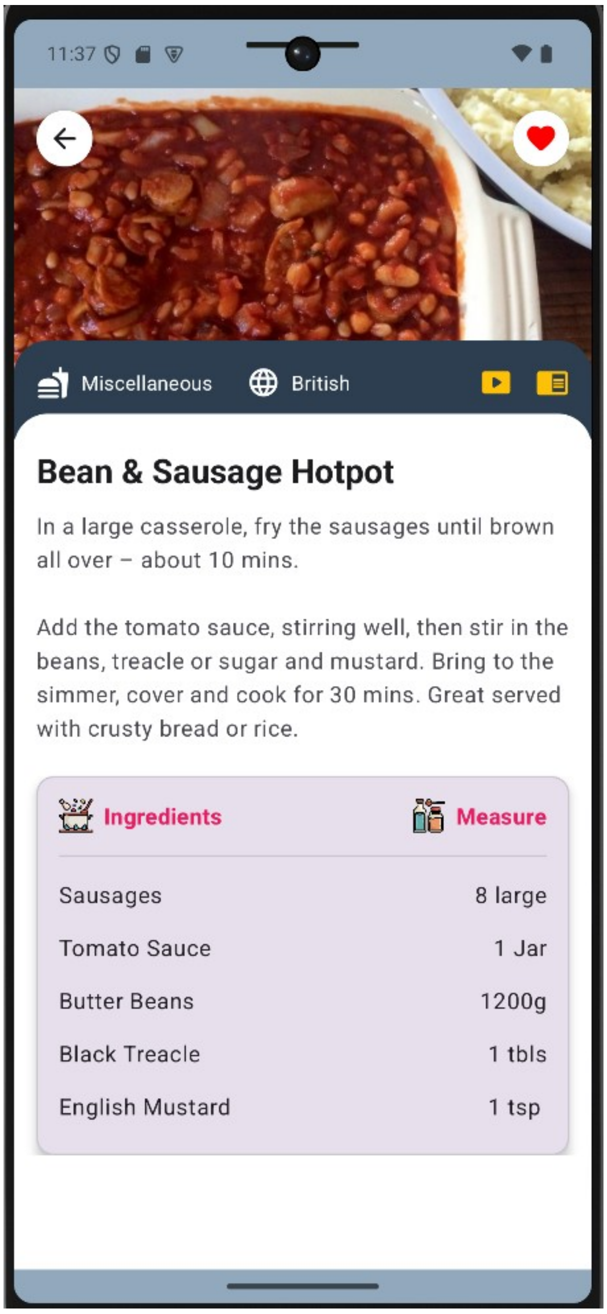


Figure 3: Home — search flow showing query input and filtered results.

11.2 Detail Screen (Activity)

Purpose Show full meal details (image, metadata, instructions, ingredients), toggle favorite, open external links (YouTube/source). Binds to `HowToViewModel`.

```
package com.Mels_Proj.feature.detail_screen.presentation.ui

import android.content.Intent
import android.graphics.Color
import android.net.Uri
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.ViewModelProvider
import com.bumptech.glide.Glide
import com.Mels_Proj.R
import com.Mels_Proj.databinding.ScreenDetailBinding
import com.Mels_Proj.feature.detail_screen.ui.HowToViewModel
import com.Mels_Proj.feature.detail_screen.ui.HowToViewModelFactory
import com.Mels_Proj.feature.meal.domain.data.model.MealResponse

class DetailScreen : AppCompatActivity() {

    private lateinit var binding: ScreenDetailBinding
    private lateinit var viewModel: HowToViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        initialBinding()
        initialViewModel()
        fetchData()
        configObservers()
    }

    private fun initialBinding() {
        binding = ScreenDetailBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }

    private fun initialViewModel() {
        val factory = HowToViewModelFactory(application)
        viewModel = ViewModelProvider(this, factory)[HowToViewModel::class.java]
    }

    private fun fetchData() {
        val mealId = intent.getStringExtra("mealID")
        if (mealId == null) {
            Toast.makeText(this, "Error: Meal ID not found", Toast.LENGTH_LONG).show()
            finish(); return
        }
        viewModel.loadMealDetails(mealId)
    }

    private fun configObservers() {
        viewModel.loadingState.observe(this) { isLoading ->
            binding.prog2.visibility = if (isLoading) View.VISIBLE else View.GONE
            val content = if (isLoading) View.INVISIBLE else View.VISIBLE
            binding.tvFoodArea.visibility = content
            binding.tvFoodCategory.visibility = content
            binding.tvFoodTitle.visibility = content
            binding.tvInstructions.visibility = content
        }
        viewModel.meal.observe(this) { meal -> meal?.let { bindMealData(it) } }
        viewModel.toastMessage.observe(this) { Toast.makeText(this, it, Toast.LENGTH_SHORT).show() }
        viewModel.isFavorite.observe(this) { isFav ->
            binding.btnFavorite.setColorFilter(if (isFav) Color.RED else Color.BLACK)
        }
    }

    private fun bindMealData(meal: MealResponse.MealResponseItem) = with(binding) {
        tvFoodTitle.text = meal.strMeal
        tvFoodCategory.text = meal.strCategory
        tvFoodArea.text = meal.strArea
        tvInstructions.text = meal.strInstructions?.replace("\r\n", "\n")
        Glide.with(this@DetailScreen).load(meal.strMealThumb).into(ivFoodImageDetail)
        setupClickListeners(meal); populateIngredients(meal)
    }

    private fun setupClickListeners(meal: MealResponse.MealResponseItem) {
        binding.btnBack.setOnClickListener { finish() }
        binding.ivYoutube.setOnClickListener {
            if (!meal.strYoutube.isNullOrBlank())
                startActivity(Intent(Intent.ACTION_VIEW, Uri.parse(meal.strYoutube)))
            else Toast.makeText(this, "No YouTube video available", Toast.LENGTH_SHORT).show()
        }
        binding.ivChrome.setOnClickListener {
            if (!meal.strSource.isNullOrBlank())
                startActivity(Intent(Intent.ACTION_VIEW, Uri.parse(meal.strSource)))
            else Toast.makeText(this, "No source link available", Toast.LENGTH_SHORT).show()
        }
        binding.btnFavorite.setOnClickListener {
            it.animate().scaleX(1.2f).scaleY(1.2f).setDuration(150).withEndAction {
                it.animate().scaleX(1f).scaleY(1f).setDuration(150).start()
            }.start()
            viewModel.onFavoriteClicked()
        }
    }

    private fun populateIngredients(meal: MealResponse.MealResponseItem) {
        binding.llIngredientsContainer.removeAllViews()
        val ingredients = createIngredientsList(meal)
        val inflater = LayoutInflater.from(this)
        for ((ing, mea) in ingredients) {
            val v = inflater.inflate(R.layout.item_ingredient, binding.llIngredientsContainer, false)
            v.findViewById<TextView>(R.id.tv_ingredient_name).text = ing
            v.findViewById<TextView>(R.id.tv_ingredient_measure).text = mea
            binding.llIngredientsContainer.addView(v)
        }
    }

    private fun createIngredientsList(meal: MealResponse.MealResponseItem): List<Pair<String,String>> =
        buildList {
            if (!meal.strIngredient1.isNullOrBlank()) add(meal.strIngredient1 to (meal.strMeasure1 ?: ""))
            if (!meal.strIngredient2.isNullOrBlank()) add(meal.strIngredient2 to (meal.strMeasure2 ?: ""))
            if (!meal.strIngredient3.isNullOrBlank()) add(meal.strIngredient3 to (meal.strMeasure3 ?: ""))
            if (!meal.strIngredient4.isNullOrBlank()) add(meal.strIngredient4 to (meal.strMeasure4 ?: ""))
            if (!meal.strIngredient5.isNullOrBlank()) add(meal.strIngredient5 to (meal.strMeasure5 ?: ""))
            if (!meal.strIngredient6.isNullOrBlank()) add(meal.strIngredient6 to (meal.strMeasure6 ?: ""))
            if (!meal.strIngredient7.isNullOrBlank()) add(meal.strIngredient7 to (meal.strMeasure7 ?: ""))
            if (!meal.strIngredient8.isNullOrBlank()) add(meal.strIngredient8 to (meal.strMeasure8 ?: ""))
            if (!meal.strIngredient9.isNullOrBlank()) add(meal.strIngredient9 to (meal.strMeasure9 ?: ""))
            if (!meal.strIngredient10.isNullOrBlank()) add(meal.strIngredient10 to (meal.strMeasure10 ?: ""))
            if (!meal.strIngredient11.isNullOrBlank()) add(meal.strIngredient11 to (meal.strMeasure11 ?: ""))
            if (!meal.strIngredient12.isNullOrBlank()) add(meal.strIngredient12 to (meal.strMeasure12 ?: ""))
            if (!meal.strIngredient13.isNullOrBlank()) add(meal.strIngredient13 to (meal.strMeasure13 ?: ""))
            if (!meal.strIngredient14.isNullOrBlank()) add(meal.strIngredient14 to (meal.strMeasure14 ?: ""))
            if (!meal.strIngredient15.isNullOrBlank()) add(meal.strIngredient15 to (meal.strMeasure15 ?: ""))
            if (!meal.strIngredient16.isNullOrBlank()) add(meal.strIngredient16 to (meal.strMeasure16 ?: ""))
            if (!meal.strIngredient17.isNullOrBlank()) add(meal.strIngredient17 to (meal.strMeasure17 ?: ""))
            if (!meal.strIngredient18.isNullOrBlank()) add(meal.strIngredient18 to (meal.strMeasure18 ?: ""))
            if (!meal.strIngredient19.isNullOrBlank()) add(meal.strIngredient19 to (meal.strMeasure19 ?: ""))
            if (!meal.strIngredient20.isNullOrBlank()) add(meal.strIngredient20 to (meal.strMeasure20 ?: ""))
        }
}
```



Bean & Sausage Hotpot

In a large casserole, fry the sausages until brown all over – about 10 mins.

Add the tomato sauce, stirring well, then stir in the beans, treacle or sugar and mustard. Bring to the simmer, cover and cook for 30 mins. Great served with crusty bread or rice.

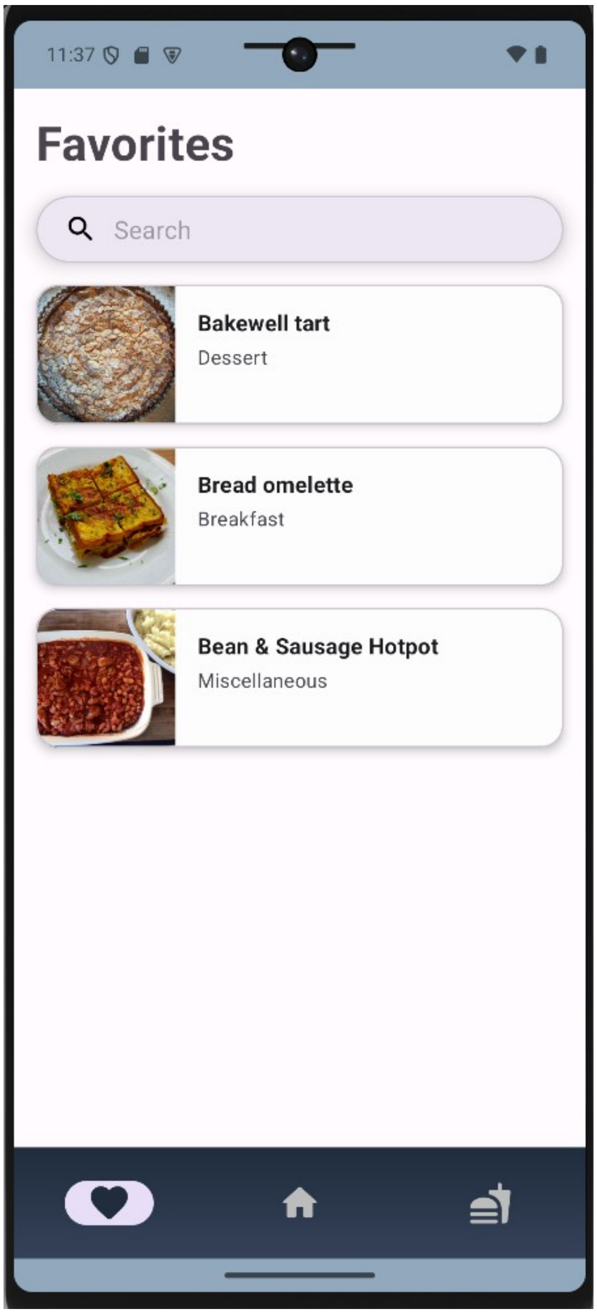
Ingredients	Measure
Sausages	8 large
Tomato Sauce	1 Jar
Butter Beans	1200g
Black Treacle	1 tbls
English Mustard	1 tsp

Figure 4: Detail screen — full recipe, metadata chips, and ingredients table.

11.3 Favorites (Fragment)

Purpose Show locally saved favorites with a text filter. Click opens `DetailScreen`.

```
package com.Mels_Proj.feature.favorites_screen.presentation.ui

import android.content.Intent
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.appcompat.widget.SearchView
import androidx.fragment.app.Fragment
import androidx.lifecycle.ViewModelProvider
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import com.Mels_Proj.databinding.FragmentFavoritesBinding
import com.Mels_Proj.feature.favorites_screen.presentation.ui.adapter.FavoritesAdapter
import com.Mels_Proj.feature.favorites_screen.presentation.viewmodel.FavoritesViewModel
import com.Mels_Proj.feature.favorites_screen.presentation.viewmodel.FavoritesViewModelFactory
import com.Mels_Proj.feature.detail_screen.presentation.ui.DetailScreen

class FavoritesFragment : Fragment() {

    private lateinit var binding: FragmentFavoritesBinding
    private lateinit var viewModel: FavoritesViewModel
    private lateinit var favoritesAdapter: FavoritesAdapter

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View
↪  {
        binding = FragmentFavoritesBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupViewModel()
        setupRecyclerView()
        setupSearch()
        setupObservers()
        viewModel.loadFavorites()
    }

    private fun setupViewModel() {
        val factory = FavoritesViewModelFactory(requireActivity().application)
        viewModel = ViewModelProvider(this, factory)[FavoritesViewModel::class.java]
    }

    private fun setupRecyclerView() {
        favoritesAdapter = FavoritesAdapter { meal ->
            Intent(requireContext(), DetailScreen::class.java).also {
                it.putExtra("mealID", meal.idMeal); startActivity(it)
            }
        }
        favoritesAdapter.stateRestorationPolicy =
            RecyclerView.Adapter.StateRestorationPolicy.PREVENT_WHEN_EMPTY

        binding.rvFavorites.apply {
            adapter = favoritesAdapter
            layoutManager = LinearLayoutManager(requireContext())
            setHasFixedSize(true)
        }
    }

    private fun setupSearch() {
        binding.searchViewFavorites.setOnQueryTextListener(object : SearchView.OnQueryTextListener {
            override fun onQueryTextSubmit(query: String?) = false
            override fun onQueryTextChange(newText: String?): Boolean {
                viewModel.searchFavorites(newText.orEmpty()); return true
            }
        })
    }

    private fun setupObservers() {
        viewModel.favoritesList.observe(viewLifecycleOwner) { favorites ->
            if (favorites.isNullOrEmpty()) {
                binding.rvFavorites.visibility = View.GONE
                binding.emptyStateView.visibility = View.VISIBLE
            } else {
                binding.emptyStateView.visibility = View.GONE
                binding.rvFavorites.visibility = View.VISIBLE
                favoritesAdapter.updateData(favorites)
            }
        }
    }
}
```



Bakewell tart
Dessert

Bread omelette
Breakfast

Bean & Sausage Hotpot
Miscellaneous

Figure 5: Favorites — saved meals list.

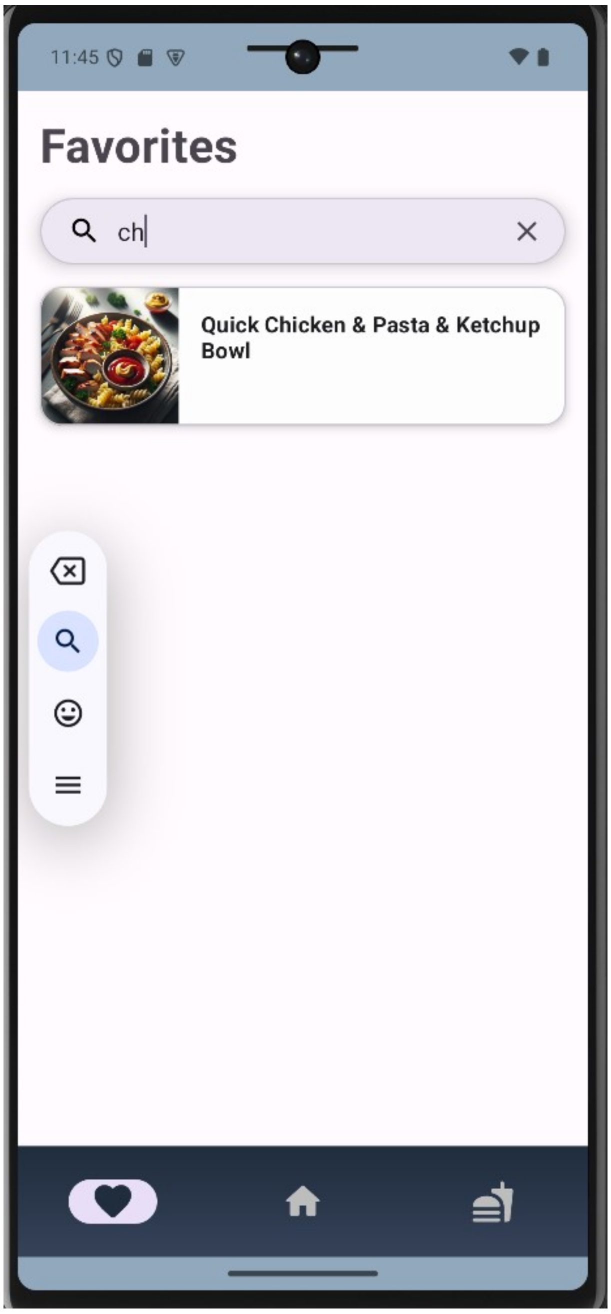


Figure 6: Favorites — in-list search filtering results.

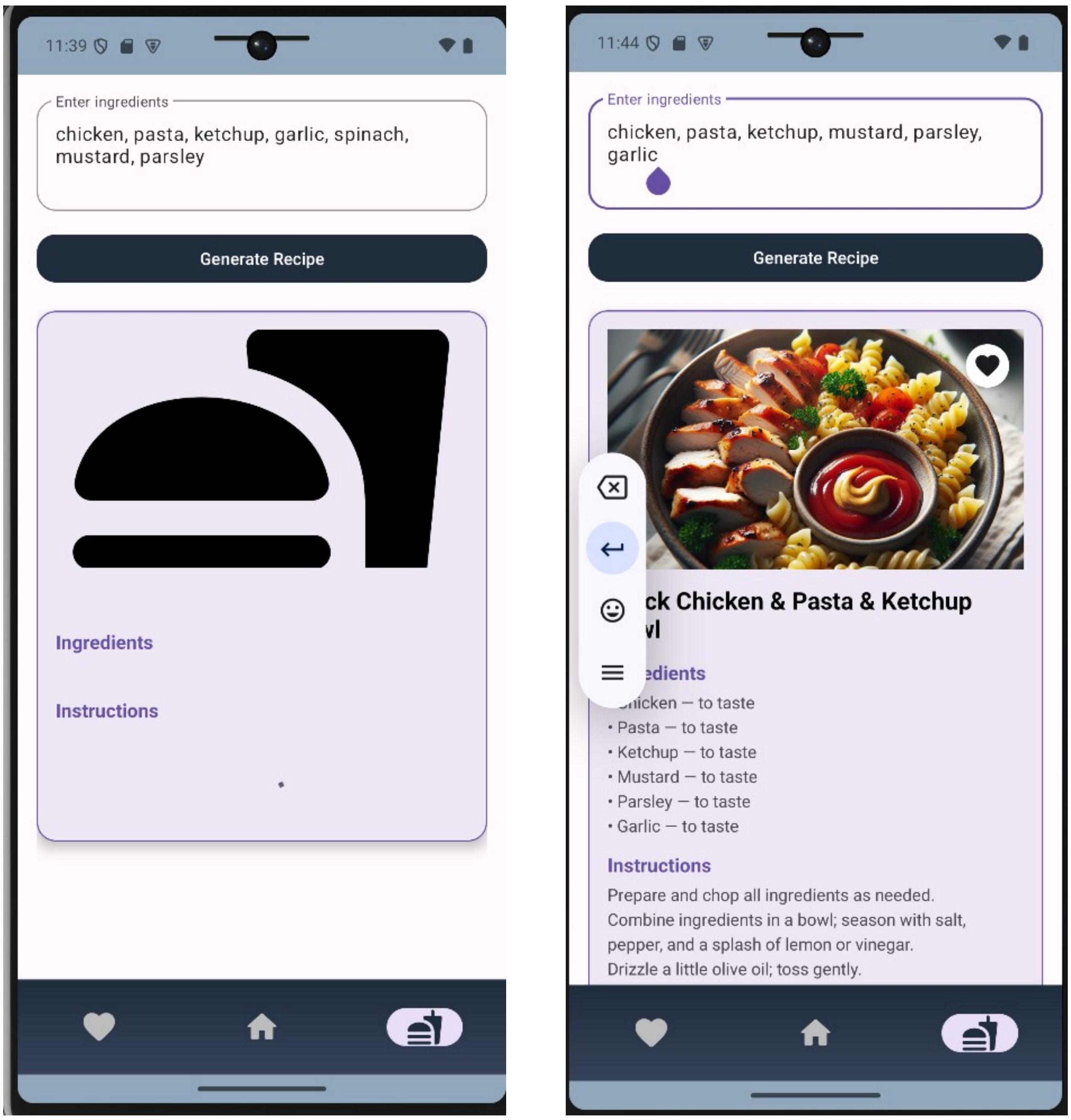
11.4 AI Generate (Fragment) — UI responsibilities & rendering

What this screen does Takes free-form ingredients, calls the AI backend, renders the MealDB-shaped result (title, image, 1..20 ingredient/measure pairs, instructions), and lets users *Favorite* the generated recipe using Room.

UI responsibilities

- Build a local `MealRepository` (Retrofit + Room) and mirror `getAllFavorites()` to know current favorite state.
- Collect `AiRecipeViewModel.state` lifecycle-safely and toggle progress/content/favorite button.
- Handle images from URL, Base64 data URI, or absent (fallback icon).

- Hide keyboard on Generate; render ingredients by pairing `strIngredientN` with `strMeasureN`.
- Toggle favorite with the same “complete entity” policy used elsewhere (fetch via `lookup` if instructions are missing).



chicken, pasta, ketchup, garlic, spinach, mustard, parsley

chicken, pasta, ketchup, mustard, parsley, garlic

ck Chicken & Pasta & Ketchup
vl

edients

- icken — to taste
- Pasta — to taste
- Ketchup — to taste
- Mustard — to taste
- Parsley — to taste
- Garlic — to taste

Instructions

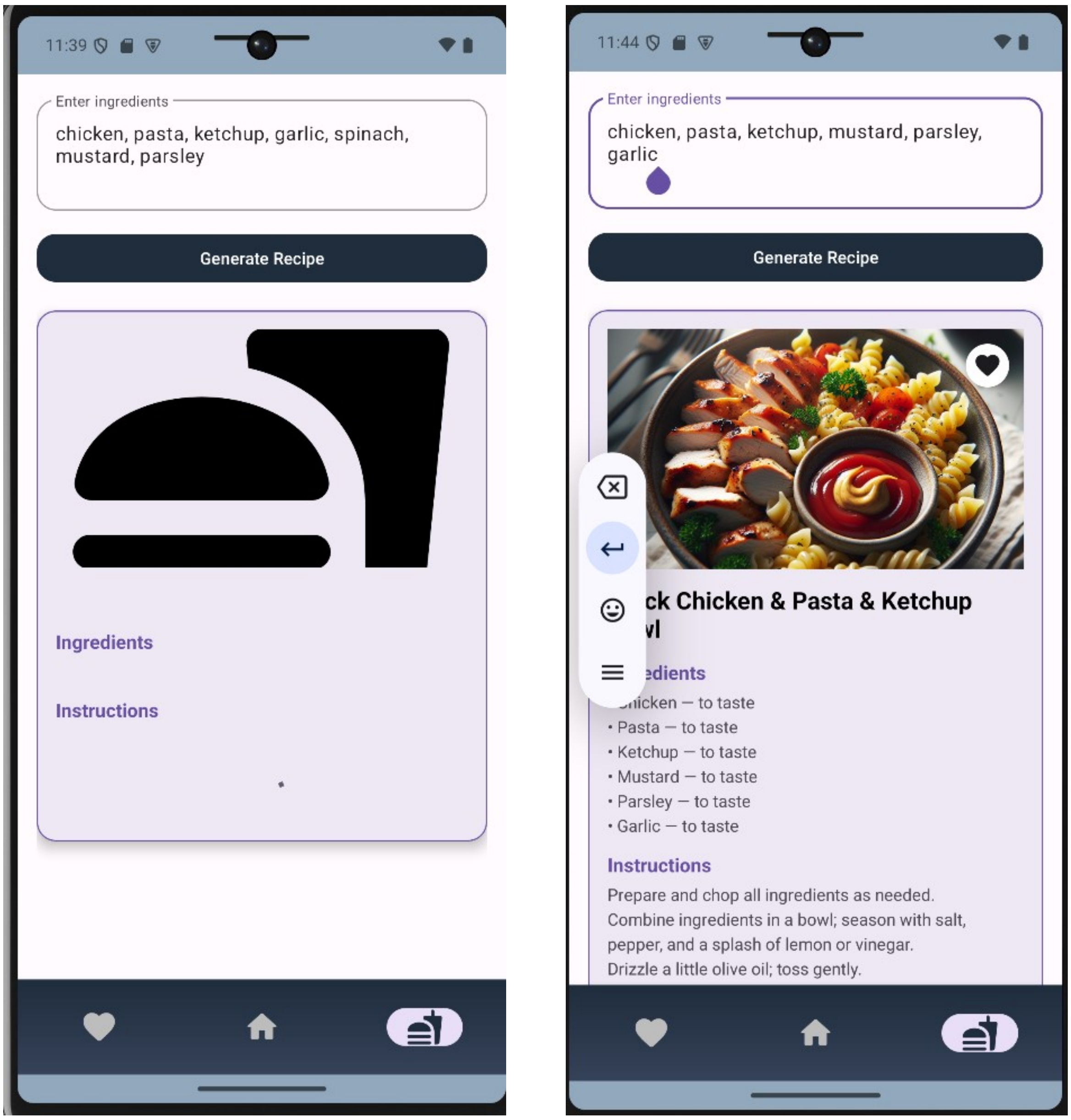
Prepare and chop all ingredients as needed.
Combine ingredients in a bowl; season with salt, pepper, and a splash of lemon or vinegar.
Drizzle a little olive oil; toss gently.

(a) Before generation (ingredients input). (b) After generation (rendered recipe).

Figure 7: AI Generate screen flow.

Implementation (full code — Fragment)

```
package com.Mels_Proj.feature.ai_recipe.presentation

import android.graphics.BitmapFactory
import android.os.Bundle
import android.util.Base64
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.view.inputmethod.InputMethodManager
import android.widget.Toast
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.lifecycleScope
import com.bumptech.glide.Glide
import com.bumptech.glide.load.resource.bitmap.RoundedCorners
import com.bumptech.glide.request.RequestOptions
import com.Mels_Proj.R
import com.Mels_Proj.databinding.FragmentAiRecipeBinding
import com.Mels_Proj.feature.meal.domain.data.model.FavoriteMealEntity
import com.Mels_Proj.feature.meal.domain.data.model.MealResponse
import com.Mels_Proj.feature.meal.domain.data.repository.MealRepository
import com.Mels_Proj.feature.meal.domain.data.toFavoriteEntity
import com.Mels_Proj.db.MealDatabase
import com.Mels_Proj.shared_component.API
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.launch

class AiRecipeFragment : Fragment() {

    private var _binding: FragmentAiRecipeBinding? = null
    private val binding get() = _binding!!
    private val vm: AiRecipeViewModel by viewModels()

    // --- Local repo + favorites (so we can run toggleFavorite here) ---
    private lateinit var mealRepository: MealRepository
    private val _allFavorites = MutableLiveData<List<FavoriteMealEntity>>(emptyList())

    // Keep last generated meal for the favorite action
    private var lastMeal: MealResponse.MealResponseItem? = null

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentAiRecipeBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // Build repository exactly like HomeViewModel does
        val mealDao = MealDatabase.getInstance(requireContext().applicationContext).mealDao()
        mealRepository = MealRepository(
            searchApi = API.searchApiService,
            filterApi = API.filterApiService,
            lookupApi = API.lookupApiService,
            randomApi = API.randomApiService,
            mealDao = mealDao
        )

        // Observe favorites so toggleFavorite can check current state
        viewLifecycleOwner.lifecycleScope.launch {
            mealRepository.getAllFavorites()?.collectLatest { favorites ->
                _allFavorites.postValue(favorites ?: emptyList())
            }
        }

        // Scroll bars in long instructions
        binding.tvInstructions.isVerticalScrollBarEnabled = true

        // Generate click
        binding.btnGenerate.setOnClickListener {
            val ingredients = binding.etIngredients.text?.toString().orEmpty().trim()
            if (ingredients.isEmpty()) {
                Toast.makeText(requireContext(), "Please enter some ingredients.", Toast.LENGTH_SHORT).show()
                return@setOnClickListener
            }

            // Hide keyboard & clear focus
            val imm = requireContext().getSystemService(InputMethodManager::class.java)
            imm.hideSoftInputFromWindow(binding.etIngredients.windowToken, 0)
            binding.etIngredients.clearFocus()

            vm.generateFromInput(ingredients, servings = 2)
        }

        // Favorite click (Fragment-local toggleFavorite)
        binding.btnFavorite.setOnClickListener {
            val meal = lastMeal
            if (meal == null) {
                Toast.makeText(requireContext(), "No recipe yet.", Toast.LENGTH_SHORT).show()
                return@setOnClickListener
            }
            toggleFavorite(meal)
        }

        // Observe AI state
        viewLifecycleOwner.lifecycleScope.launch {
            vm.state.collectLatest { state ->
                when (state) {
                    is AiUiState.Idle -> {
                        showLoading(false)
                        binding.btnFavorite.visibility = View.GONE
                    }
                    is AiUiState.Loading -> {
                        showLoading(true)
                        binding.btnFavorite.visibility = View.GONE
                    }
                    is AiUiState.Error -> {
                        showLoading(false)
                        binding.btnFavorite.visibility = View.GONE
                        Toast.makeText(requireContext(), state.message, Toast.LENGTH_SHORT).show()
                    }
                    is AiUiState.Success -> {
                        showLoading(false)
                        lastMeal = state.meal
                        renderMeal(state.meal)
                        binding.btnFavorite.visibility = View.VISIBLE
                    }
                }
            }
        }
    }

    private fun showLoading(show: Boolean) {
        binding.progress.visibility = if (show) View.VISIBLE else View.GONE
        binding.btnGenerate.isEnabled = !show
    }

    private fun renderMeal(item: MealResponse.MealResponseItem) {
        binding.tvTitle.text = item.strMeal ?: getString(R.string.app_name)

        val thumb = item.strMealThumb
        if (thumb.isNullOrBlank()) {
            binding.imgMeal.setImageResource(R.drawable.ic_round_fastfood_24)
        } else if (thumb.startsWith("data:image")) {
            try {
                val base64 = thumb.substringAfter("base64,", "")
                val bytes = Base64.decode(base64, Base64.DEFAULT)
                val bmp = BitmapFactory.decodeByteArray(bytes, 0, bytes.size)
                Glide.with(binding.imgMeal)
                    .asBitmap()
                    .load(bmp)
                    .apply(RequestOptions.bitmapTransform(RoundedCorners(24)))
                    .placeholder(R.drawable.ic_round_fastfood_24)
                    .error(R.drawable.ic_round_fastfood_24)
                    .into(binding.imgMeal)
            } catch (_: Throwable) {
                binding.imgMeal.setImageResource(R.drawable.ic_round_fastfood_24)
            }
        } else {
            Glide.with(binding.imgMeal)
                .load(thumb)
                .apply(RequestOptions.bitmapTransform(RoundedCorners(24)))
                .placeholder(R.drawable.ic_round_fastfood_24)
                .error(R.drawable.ic_round_fastfood_24)
                .into(binding.imgMeal)
        }

        // Ingredients 1..20
        val lines = buildList {
            fun addIf(name: String?, measure: String?) {
                if (!name.isNullOrBlank()) add(if (!measure.isNullOrBlank()) "$name  $measure" else name)
            }
            addIf(item.strIngredient1, item.strMeasure1)
            addIf(item.strIngredient2, item.strMeasure2)
            addIf(item.strIngredient3, item.strMeasure3)
            addIf(item.strIngredient4, item.strMeasure4)
            addIf(item.strIngredient5, item.strMeasure5)
            addIf(item.strIngredient6, item.strMeasure6)
            addIf(item.strIngredient7, item.strMeasure7)
            addIf(item.strIngredient8, item.strMeasure8)
            addIf(item.strIngredient9, item.strMeasure9)
            addIf(item.strIngredient10, item.strMeasure10)
            addIf(item.strIngredient11, item.strMeasure11)
            addIf(item.strIngredient12, item.strMeasure12)
            addIf(item.strIngredient13, item.strMeasure13)
            addIf(item.strIngredient14, item.strMeasure14)
            addIf(item.strIngredient15, item.strMeasure15)
            addIf(item.strIngredient16, item.strMeasure16)
            addIf(item.strIngredient17, item.strMeasure17)
            addIf(item.strIngredient18, item.strMeasure18)
            addIf(item.strIngredient19, item.strMeasure19)
            addIf(item.strIngredient20, item.strMeasure20)
        }
        binding.tvIngredients.text = lines.joinToString("\n") { " $it" }
        binding.tvInstructions.text = item.strInstructions.orEmpty()
    }

    /**
     * Fragment-local favorite toggle using Room via MealRepository.
     */
    private fun toggleFavorite(meal: MealResponse.MealResponseItem) {
        viewLifecycleOwner.lifecycleScope.launch {
            val isCurrentlyFavorite =
                _allFavorites.value?.any { it.idMeal == meal.idMeal } == true

            if (isCurrentlyFavorite) {
                val favoriteEntity = meal.toFavoriteEntity() ?: return@launch
                mealRepository.removeFavorite(favoriteEntity)
            } else {
                if (meal.strInstructions.isNullOrBlank()) {
                    meal.idMeal?.let { mealId ->
                        mealRepository.getMealById(mealId).onSuccess { response ->
                            response.mealList.firstOrNull()?.let { fullMeal ->
                                fullMeal.toFavoriteEntity()?.let {
                                    mealRepository.addFavorite(it)
                                }
                            }
                        }.onFailure {
                            onSomeError("Failed to fetch full meal details.")
                        }
                    }
                } else {
                    meal.toFavoriteEntity()?.let {
                        mealRepository.addFavorite(it)
                    }
                }
            }
        }
    }

    private fun onSomeError(message: String) {
        Toast.makeText(requireContext(), message, Toast.LENGTH_SHORT).show()
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```



chicken, pasta, ketchup, garlic, spinach, mustard, parsley

chicken, pasta, ketchup, mustard, parsley, garlic

ck Chicken & Pasta & Ketchup
wl

edients
hicken — to taste
• Pasta — to taste
• Ketchup — to taste
• Mustard — to taste
• Parsley — to taste
• Garlic — to taste

Instructions
Prepare and chop all ingredients as needed.
Combine ingredients in a bowl; season with salt, pepper, and a splash of lemon or vinegar.
Drizzle a little olive oil; toss gently.

(a) Ingredients entered before generation. (b) Generated recipe rendered in the UI.

Figure 8: AI recipe generation flow.

11.5 Bindings

What ViewBinding generates For each layout XML `<name>.xml`, Gradle generates a type-safe class `<Name>Binding` with direct references to its views. In this UI:

- `FragmentHomeBinding` → `fragment_home.xml`
- `ScreenDetailBinding` → `screen_detail.xml`
- `FragmentFavoritesBinding` → `fragment_favorites.xml`
- `ItemHomeMealBinding`, `ItemHomeCatBinding` → RecyclerView rows

Activity binding (lifetime = Activity) Inflate once in `onCreate`, setContentView to `binding.root`. No manual nulling. If you hold the binding past `onDestroy`, that's on you.

```
class DetailScreen : AppCompatActivity() {
  private lateinit var binding: ScreenDetailBinding
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ScreenDetailBinding.inflate(layoutInflater)
    setContentView(binding.root)
  }
}
```

Fragment binding (view lifetime ≠ fragment lifetime) The view dies in `onDestroyView`. If you keep a non-nullable binding field in a Fragment and never clear it, you will leak—no discussion.

```
class SafeFragment : Fragment() {
  private var _binding: FragmentHomeBinding? = null
  private val binding get() = _binding!!

  override fun onCreateView(inflater: LayoutInflater, c: ViewGroup?, s: Bundle?): View {
    _binding = FragmentHomeBinding.inflate(inflater, c, false)
    return binding.root
  }

  override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
  }
}
```

RecyclerView ViewHolder binding (row-scoped) Each ViewHolder owns one row binding. No nulling; GC handles it with the row view. Don't play smart—just use the generated binding.

```
class MealVH(private val binding: ItemHomeMealBinding)
  : RecyclerView.ViewHolder(binding.root) {
  fun bind(item: MealResponse.MealResponseItem) {
    binding.tvMealTitle.text = item.strMeal
    // ...
  }
}
```

11.6 Adapters — Code & responsibilities

MealAdapter (carousel with favorite overlay)

```
package com.Mels_Proj.feature.home_screen.presentation.ui.adapter

import android.annotation.SuppressLint
import android.content.Intent
import android.graphics.Color
import android.net.Uri
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import com.bumptech.glide.Glide
import com.Mels_Proj.databinding.ItemHomeMealBinding
import com.Mels_Proj.feature.meal.domain.data.model.MealResponse

class MealAdapter(
    private val onItemClick: (MealResponse.MealResponseItem) -> Unit,
    private val onFavoriteClick: (MealResponse.MealResponseItem) -> Unit
) : RecyclerView.Adapter<MealAdapter.ViewHolder>() {

    private var items: List<MealResponse.MealResponseItem> = emptyList()
    private var favoriteMealIds: Set<String> = emptySet()

    @SuppressLint("NotifyDataSetChanged")
    fun updateData(newItems: List<MealResponse.MealResponseItem>) {
        items = newItems
        notifyDataSetChanged()

    }

    fun updateFavorites(newFavoriteIds: Set<String>) {
        val oldFavoriteIds = favoriteMealIds
        favoriteMealIds = newFavoriteIds

        items.forEachIndexed { index, meal ->
            val mealId = meal.idMeal ?: return@forEachIndexed
            val wasFavorite = oldFavoriteIds.contains(mealId)
            val isNowFavorite = newFavoriteIds.contains(mealId)
            if (wasFavorite != isNowFavorite) {
                notifyItemChanged(index, PAYLOAD_FAVORITE_STATUS_CHANGED)
            }
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val binding = ItemHomeMealBinding.inflate(
            LayoutInflater.from(parent.context),
            parent,
            false
        )
        return ViewHolder(binding)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(items[position])
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int, payloads: MutableList<Any>) {
        if (payloads.contains(PAYLOAD_FAVORITE_STATUS_CHANGED)) {
            holder.updateFavoriteStatus(favoriteMealIds.contains(items[position].idMeal))
        } else {
            super.onBindViewHolder(holder, position, payloads)
        }
    }

    override fun getItemCount(): Int = items.size

    inner class ViewHolder(private val binding: ItemHomeMealBinding) :
        RecyclerView.ViewHolder(binding.root) {

        init {
            binding.root.setOnClickListener {
                if (adapterPosition != RecyclerView.NO_POSITION) {
                    onItemClick(items[adapterPosition])
                }
            }

            binding.btnFavorite.setOnClickListener {
                if (adapterPosition != RecyclerView.NO_POSITION) {
                    val meal = items[adapterPosition]
                    it.animate()
                        .scaleX(1.2f)
                        .scaleY(1.2f)
                        .setDuration(150)
                        .withEndAction {
                            it.animate()
                                .scaleX(1f)
                                .scaleY(1f)
                                .setDuration(150)
                                .start()
                        }.start()
                    onFavoriteClick(meal)
                }
            }
        }

        fun updateFavoriteStatus(isFavorite: Boolean) {
            binding.btnFavorite.setColorFilter(if (isFavorite) Color.RED else Color.BLACK)
        }

        fun bind(item: MealResponse.MealResponseItem) {
            binding.tvMealTitle.text = item.strMeal

            if (!item.strSource.isNullOrBlank()) {
                binding.ivChrome.visibility = View.VISIBLE
                binding.ivChrome.setOnClickListener {
                    val intent = Intent(Intent.ACTION_VIEW, Uri.parse(item.strSource))
                    binding.root.context.startActivity(intent)
                }
            } else {
                binding.ivChrome.visibility = View.INVISIBLE
            }

            item.strCategory?.let {
                binding.tvCategory.text = it
                binding.ivCategoryIcon.visibility = View.VISIBLE
                binding.tvCategory.visibility = View.VISIBLE
            } ?: run {
                binding.ivCategoryIcon.visibility = View.GONE
                binding.tvCategory.visibility = View.GONE
            }
            item.strArea?.let {
                binding.tvArea.text = it
                binding.ivAreaIcon.visibility = View.VISIBLE
                binding.tvArea.visibility = View.VISIBLE
            } ?: run {
                binding.ivAreaIcon.visibility = View.GONE
                binding.tvArea.visibility = View.GONE
            }
            Glide.with(binding.root.context)
                .load(item.strMealThumb)
                .into(binding.ivMealImage)

            updateFavoriteStatus(favoriteMealIds.contains(item.idMeal))
        }
    }

    companion object {
        private const val PAYLOAD_FAVORITE_STATUS_CHANGED = "PAYLOAD_FAVORITE_STATUS_CHANGED"
    }
}
```

Responsibilities Snapshot list (`updateData`); cheap favorite overlay via payload rebinding (`updateFavorites`); emits clean callbacks upward. No DB or network logic inside the adapter—good.

AlphabetAdapter (letter scroller)

```
package com.Mels_Proj.feature.home_screen.presentation.ui.adapter

import android.view.LayoutInflater
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.Mels_Proj.R

class AlphabetAdapter(
    private val onLetterClick: (String) -> Unit
) : RecyclerView.Adapter<AlphabetAdapter.ViewHolder>() {

    private val alphabet = listOf("All") + ('A'..'Z').map { it.toString() }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_alphabet, parent, false) as TextView
        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(alphabet[position])
    }

    override fun getItemCount(): Int = alphabet.size

    inner class ViewHolder(private val textView: TextView) : RecyclerView.ViewHolder(textView) {
        init {
            textView.setOnClickListener {
                if (adapterPosition != RecyclerView.NO_POSITION) {
                    onLetterClick(alphabet[adapterPosition])
                }
            }
        }

        fun bind(letter: String) {
            textView.text = letter
        }
    }
}
```

CategoryAdapter (category chips with selection)

```
package com.Mels_Proj.feature.home_screen.presentation.ui.adapter

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import com.bumptech.glide.Glide
import com.Mels_Proj.databinding.ItemHomeCatBinding
import com.Mels_Proj.feature.category.domain.data.model.CategoriesResponse

class CategoryAdapter(
    private val onItemClick: (CategoriesResponse.CategoryResponseItem) -> Unit
) : RecyclerView.Adapter<CategoryAdapter.ViewHolder>() {

    private var items: List<CategoriesResponse.CategoryResponseItem> = emptyList()
    private var selectedPosition = -1

    fun updateData(newItems: List<CategoriesResponse.CategoryResponseItem>) {
        items = newItems
        selectedPosition = -1
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val binding = ItemHomeCatBinding.inflate(
            LayoutInflater.from(parent.context),
            parent,
            false
        )
        return ViewHolder(binding)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val data = items[position]
        holder.bind(data)
    }

    override fun getItemCount(): Int = items.size

    inner class ViewHolder(private val binding: ItemHomeCatBinding) :
        RecyclerView.ViewHolder(binding.root) {

        init {
            binding.root.setOnClickListener {
                val clickedPosition = adapterPosition
                if (clickedPosition != RecyclerView.NO_POSITION) {
                    val previousPosition = selectedPosition

                    if (selectedPosition == clickedPosition) {
                        selectedPosition = -1
                        notifyItemChanged(previousPosition)
                    } else {
                        selectedPosition = clickedPosition
                        if (previousPosition != -1) {
                            notifyItemChanged(previousPosition)
                        }
                        notifyItemChanged(selectedPosition)
                    }
                    onItemClick(items[clickedPosition])
                }
            }
        }

        fun bind(item: CategoriesResponse.CategoryResponseItem) {
            binding.tvCategoryName.text = item.strCategory

            Glide.with(binding.root.context)
                .load(item.strCategoryThumb)
                .into(binding.ivCategoryImage)

            val cardView = binding.root
            if (adapterPosition == selectedPosition) {
                val strokeWidthPx = (3 * binding.root.context.resources.displayMetrics.density).toInt()
                cardView.strokeWidth = strokeWidthPx
            } else {
                cardView.strokeWidth = 0
            }
        }
    }
}
```

11.7 Main Activity (Navigation shell & BottomNavigationView)

Purpose `MainActivity` is the single-activity shell that hosts the app's three primary fragments (**Home**, **Favorites**, **AI Recipe**). It owns the `BottomNavigationView` and swaps fragments into a `FragmentContainerView`.

Responsibilities

- Inflate `ActivityMainBinding` and set the content view (*binding conventions in* §11.8).
- On first launch, load `HomeFragment` and set the selected tab.
- Handle tab selection and replace or show/hide the current fragment accordingly.
- Keep fragment transactions simple and lifecycle-safe.

Implementation

```
package com.Mels_Proj
import com.Mels_Proj.feature.ai_recipe.presentation.AiRecipeFragment
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.Fragment
import com.Mels_Proj.databinding.ActivityMainBinding
import com.Mels_Proj.feature.favorites_screen.presentation.ui.FavoritesFragment
import com.Mels_Proj.feature.home_screen.presentation.ui.HomeFragment

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        if (savedInstanceState == null) {
            loadFragment(HomeFragment())
            binding.bottomNavigation.selectedItemId = R.id.navigation_home
        }

        binding.bottomNavigation.setOnItemSelectedListener { item ->
            when (item.itemId) {
                R.id.navigation_home -> { loadFragment(HomeFragment()); true }
                R.id.navigation_favorites -> { loadFragment(FavoritesFragment()); true }
                R.id.navigation_ai -> { loadFragment(AiRecipeFragment()); true }
                else -> false
            }
        }
    }

    private fun loadFragment(fragment: Fragment) {
        supportFragmentManager.beginTransaction()
            .replace(R.id.fragment_container, fragment)
            .commit()
    }
}
```

Activity XML contract To ensure smooth navigation and consistent UI behavior, the activity layout must expose specific view IDs and components that the code relies on to swap content and handle user input.

Required views

`@+id/fragment_container` — a `FragmentContainerView` or `FrameLayout` that acts as the placeholder where fragments are loaded or replaced.

- All top-level screens (Home, Favorites, AI) appear inside this container.
- Navigation logic targets this ID to display the active fragment.
- *Example:* tapping the "Favorites" tab loads the favorites fragment into this container.

`@+id/bottomNavigation` — a `BottomNavigationView` that provides the bottom tab bar for app navigation.

- Contains menu items representing the main sections, with fixed IDs:
  - `navigation_home` — Home screen
  - `navigation_favorites` — Favorites screen
  - `navigation_ai` — AI-generated recipe screen
- These IDs are referenced in Kotlin to listen for tab selection and trigger the corresponding fragment transactions.

11.8 Handling UI XML Files

The user interface (UI) is built with **ConstraintLayout** and **Material Components**, and all layouts are wired to Kotlin via **ViewBinding** for type-safe view access (no `findViewById`).

Core layout concepts

- Each screen is defined in a separate XML under `res/layout`. The XML describes structure, styling, and constraints between views.
- **ConstraintLayout** is the root in most screens, enabling precise, responsive positioning relative to siblings or parent.
- Material theming (shapes, elevation, typography) is applied via Material components and theme attributes.
- ViewBinding generates a `<Name>Binding` class per layout for direct, null-safe property access.

Common components used

- **ConstraintLayout** — flexible root container for complex, responsive UIs.
- **MaterialCardView** — content blocks with rounded corners and elevation.
- **SearchView** — text input for query/submit interactions.
- **MaterialButton** — primary/secondary actions (e.g., filter by letter).
- **RecyclerView** — efficient scrolling lists (categories, meals, alphabet).
- **ProgressBar** — visible during loading states.
- **TextView** — titles, labels, and static text.
- **View** — utility/invisible views for transitions/overlays.

12 Wrap-up & Conclusion

This report delivered a complete, interoperable system: an Android client (MVVM + Repository + Room + Retrofit) and a FastAPI backend that turns free-form ingredients into MealDB-shaped recipes. The two pieces meet at a small, versioned contract (`POST /v1/ai/recipe`) so the UI renders AI results with zero special-casing. The result is production-minded: predictable models, clear failure modes, and a path to ship safely.

What’s solid now

- **Clean boundaries:** UI → ViewModel → Repository; Retrofit/Room hidden behind repositories.
- **Reliable shapes:** MealDB-compatible models end-to-end; up to 20 aligned ingredient/measure slots.
- **Offline-friendly:** Favorites persisted via Room with reactive flows.
- **Operational guardrails:** Timeouts, CORS allowlist, minimal error envelope, and key redaction.
- **Navigable UI:** BottomNavigationView hosting Home, Favorites, and AI Recipe surfaces.