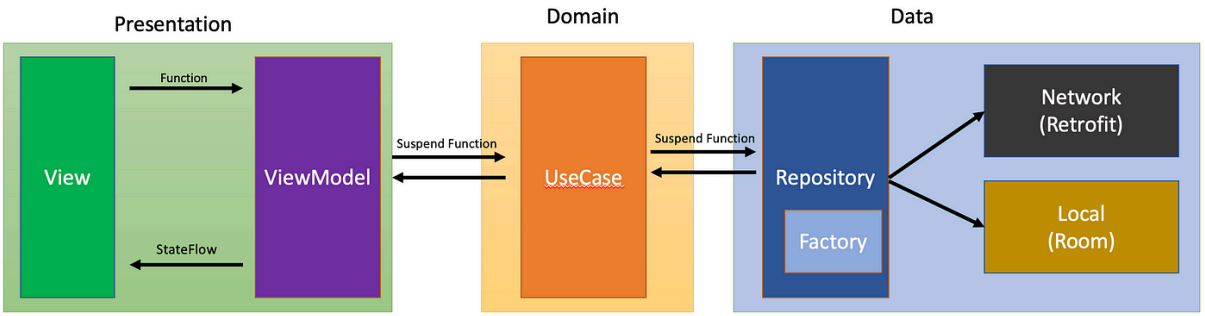
Project of mobile

1. Architecture Globale du Projet



architecture **MVVM Clean Architecture + Jetpack Compose + Hilt + Room + Firebase**.

C'est aujourd'hui **le standard recommandé par Google**, surtout avec **Jetpack Compose**. Voici **les raisons essentielles** pour lesquelles MVVM est utilisé.

Séparation claire des responsabilités

Dans MVVM :

- **Model** → données (Room, Firebase, API, modèles métier)
- **View** → interface utilisateur (Compose UI)
- **ViewModel** → logique de présentation

Extensible et évolutif (scalable)

Pour un projet comme un shop :

- gestion du panier
- authentification
- base locale

- synchronisation firebase
- écran produit/détail/checkout
- paramètres, favoris, orders...

MVVM permet :

✔️ d'ajouter des fonctionnalités sans casser les autres

✔️ de travailler en équipe sur différentes couches en parallèle

✔️ de garder une structure stable

Intégration parfaite avec Jetpack Compose

Compose fonctionne très bien avec **StateFlow**, **MutableState**, **LiveData**...

Et MVVM fournit exactement ça.

Le ViewModel expose un état :

```
val uiState = MutableStateFlow(ProductUiState())
```

Et la vue écoute automatiquement :

```
viewModel.uiState.collectAsState()
```

➡️ **Pas besoin de callbacks, ni d'observateurs manuels.**

➡️ **UI toujours synchronisée avec les données.**

Couche 1 — Presentation (UI + ViewModels)

📁 `presentation/`

- UI 100% Jetpack Compose
- Chaque écran a un **ViewModel**

- Le ViewModel communique avec le **Repository**
- Il expose des **StateFlow** / **MutableStateFlow**

Exemples de ViewModels :

- `ProductListViewModel.kt`
- `LoginViewModel.kt`
- `CartViewModel.kt`
- `SettingsViewModel.kt`

Couche 2 — Domain (Business Logic)

📁 `domain/model/` → Modèles "propres" indépendants d'Android

📁 `domain/repository/` → Interfaces des repositories

Exemples :

- `Product.kt`
- `User.kt`
- `Order.kt`

Couche 3 — Data (Sources locales et distantes)

📁 `data/local/` → Room Database

📁 `data/datafirebase/` → Firestore models

📁 `data/repository/` → Implémentations des repository

📁 `data/mapper/` → Conversions Entity ↔ Domain

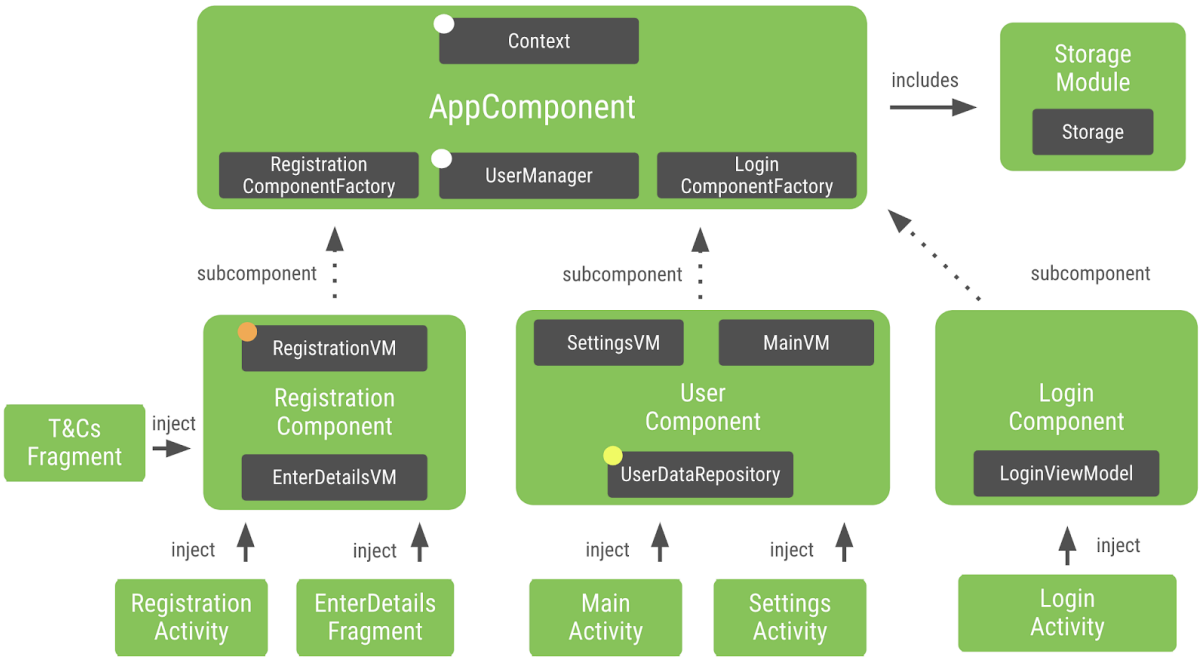
Sources de données utilisées :

- **Firebase Firestore**
- **Firebase Auth**
- **Firebase Storage**
- **Room Database**
- **Shared Preferences (DataStore)**

Couche 4 — DI (Injection de dépendances)

📁 `di/`

- `AppModule.kt` → Fournit DAO, Repository, Database
- `FirebaseModule.kt` → Fournit Firestore, Auth, Storage
- `MyApplication.kt` utilise `@HiltAndroidApp`



2. Les outils et technologies utilisés

Outil / Technologie	Usage
Jetpack Compose	UI déclarative
MVVM	Architecture UI
Room	Base de données locale
Firebase Auth	Authentification
Firebase Firestore	Produits & commandes
Firebase Storage	Images produits
Hilt	Injection des dépendances
DataStore Preferences	Stockage thème/langue
Kotlin Coroutines + Flow	

3. Méthodes importantes utilisées dans le projet

🔷 Mappers

Convertissent entités Room ↔ Domain.

Exemple :

```
fun ProductEntity.toDomain(): Product
```

🔷 Flows

Exemple :

```
val authState = MutableStateFlow(AuthState())
```

🔷 Coroutines

Les Repository utilisent :

```
withContext(Dispatchers.IO)
```

🔷 Firebase Batch Write

Dans MyApplication :

```
firestore.batch().set(...).commit()
```

🔷 DataStore pour préférences

Store du thème + langue :

```
themeOptionFlow
languageFlow
```

A Component in Java/Kotlin is

Declarative, XML is *Imperative*

No More “Two-Language Problem”

With Components (Compose):

- UI + logic in **one place**
- Everything written in **one language**

✔️ easier to read
✔️ faster to write
✔️ better for teams

Feature	XML Layouts	Java/Kotlin Components (Compose)
Declarative	❌ No	✔️ Yes
Reusable UI	🔶 Limited	✔️ Excellent
Performance	🔶 Medium	✔️ High
State handling	❌ Manual	✔️ Built-in
Testing	❌ Hard	✔️ Easy
Language	XML + Java	Kotlin only
Learning curve	Medium	Easy once learned