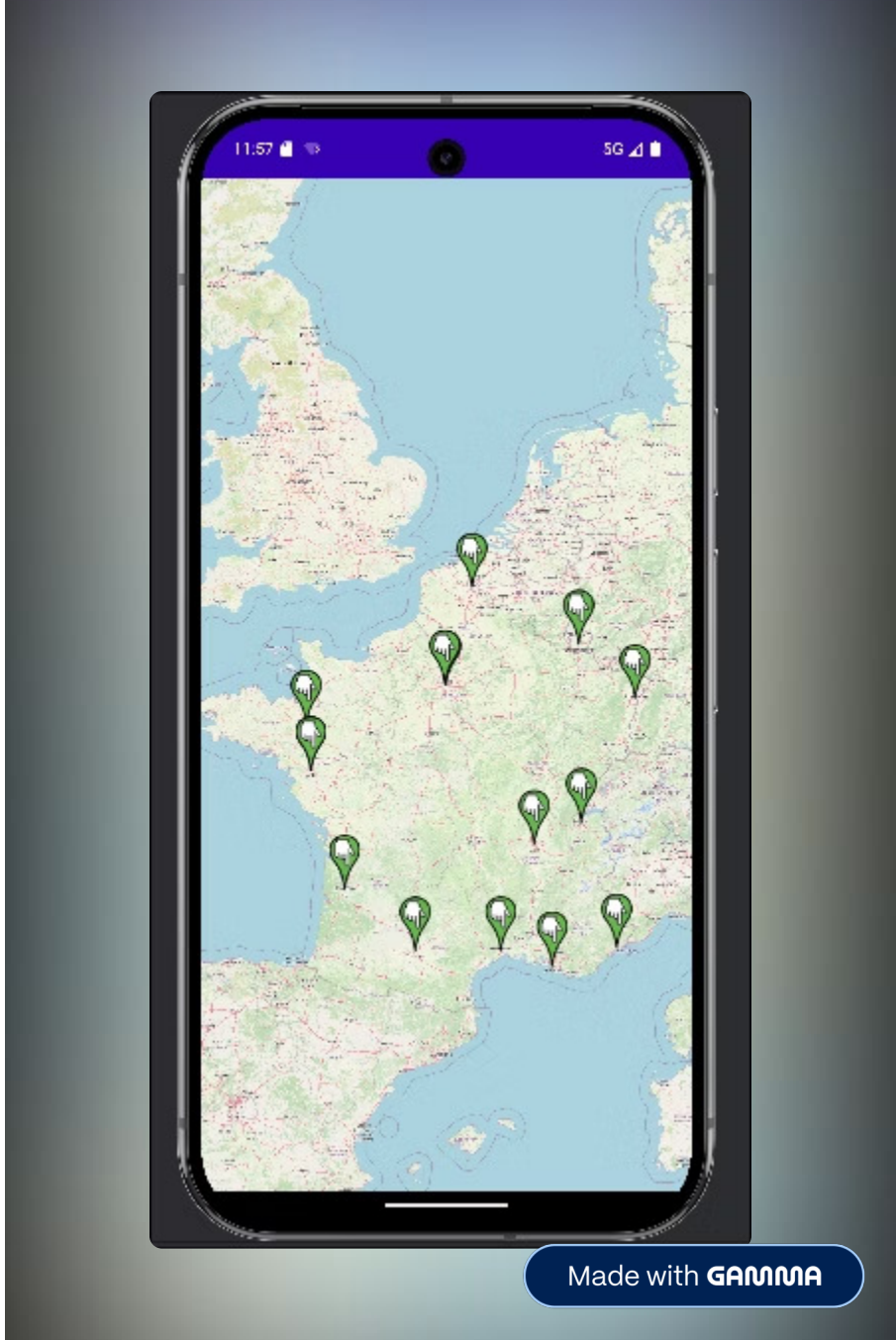
# Application Dawan Locations

Bienvenue à cette présentation de mon application Android développée pour afficher les centres Dawan sur une carte interactive.



# Objectifs Clés du Projet

## Affichage Cartographique

Présenter visuellement les centres Dawan sur une carte, chaque centre étant représenté par une épingle interactive.

## Persistance des Données

Stocker les localisations en local via SQLite pour garantir un accès en mode hors connexion, améliorant l'expérience utilisateur.

## Intégration REST

Consommer les données des centres Dawan depuis l'API publique ( **dawan.org/public/location/** ) via un client web service.

## Interaction Utilisateur

Permettre l'affichage des détails complets de chaque centre dans un fragment dédié au clic sur son épingle.

# Parcours Utilisateur : Une Expérience Fluide

01

## Démarrage de l'Application

L'application s'ouvre directement sur l'écran principal affichant la carte.

02

## Affichage des Centres

Les centres Dawan déjà stockés localement sont immédiatement visibles sous forme d'épingles sur la carte.

03

## Rafraîchissement des Données

Un mécanisme de rafraîchissement tente de synchroniser les données avec l'API REST distante.

04

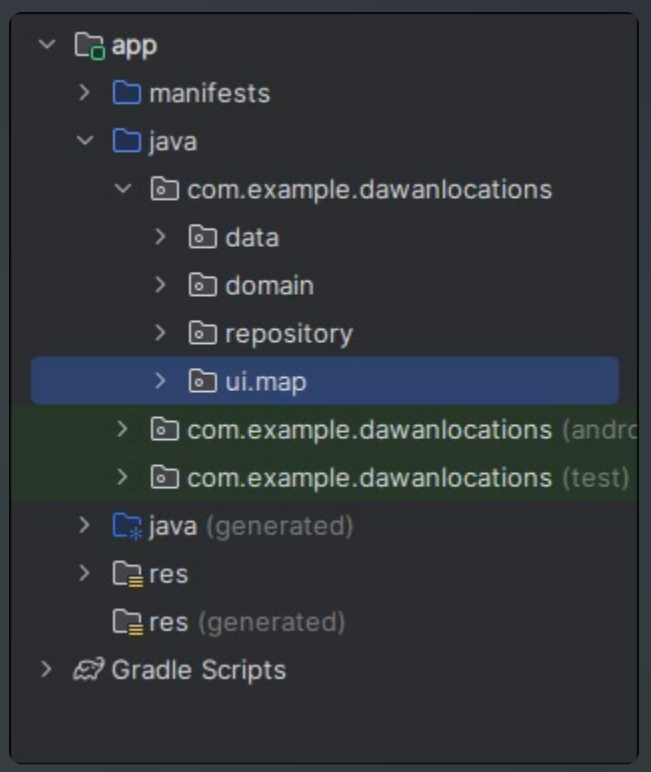
## Gestion du Mode Hors-Ligne

En cas d'échec de la connexion réseau, un bandeau informatif "mode hors-ligne" apparaît.

05

## Détails du Centre

Un clic sur une épingle ouvre une **BottomSheet** avec les détails complets du centre (nom, adresse).




# Architecture en Couches

L'application est structurée autour d'une architecture en couches pour garantir une séparation claire des responsabilités, facilitant la maintenabilité et la testabilité. Nous avons adopté le pattern **MVVM** (Model-View-ViewModel) en conjonction avec le pattern **Repository**.

## Couche UI

Gère l'affichage et les interactions utilisateur (ex: `ui/map/Activity`, `BottomSheet`).

## Couche Domain

Contient la logique métier et les modèles de données agnostiques à la source (`domain/model`, `domain/mapper`).

## Couche Data

Abstraie les sources de données, qu'elles soient distantes ( `data/remote/*` via Retrofit) ou locales ( `data/local/*` via Room).

## Couche Repository

Orchestre l'accès aux données, choisissant entre source locale ou distante (`repository/*`).

# Intégration du Web Service REST avec Retrofit

Pour communiquer avec l'API REST de Dawan, nous utilisons la bibliothèque Retrofit, un client HTTP sûr et efficace pour Android.

## Composants Clés

- **Interface API (`DawanApi`):** Définit les endpoints avec des annotations pour une typisation forte.
- **Client Retrofit (`RetrofitClient`):** Configure l'URL de base et le convertisseur JSON (Gson).

## Gestion des Réponses

- **Succès HTTP (2xx):** Le corps de la réponse est traité si non nul.
- **Erreurs HTTP:** Toute réponse avec un code d'erreur est propagée comme une exception métier.
- **Exceptions Réseau (`IOException`):** Capturées et converties en un état "offline" pour l'interface utilisateur.

```
/**
 * Retourne une instance unique de {@link Retrofit}.
 * <p>
 * L'appel est synchronisé pour être sûr qu'une seule instance
 * est créée même en cas d'accès concurrent.
 * </p>
 *
 * @return l'instance {@link Retrofit} configurée avec {@code BASE_URL}
 *         et le convertisseur Gson pour (dé)sérialiser les JSON.
 */
1 usage
public static synchronized Retrofit getInstance() {
    if (instance == null) {
        instance = new Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build();
    }
    return instance;
}
```

Retrofit simplifie grandement l'interaction avec les API REST, rendant le code plus lisible et moins sujet aux erreurs.

# Stockage Local avec Room

Pour assurer la persistance des données et le mode hors connexion, nous avons opté pour **Room Persistence Library**, la solution officielle de Google pour SQLite sur Android.

## Pourquoi Room ?

Room offre une abstraction au-dessus de SQLite, avec des annotations pour définir les entités, les DAO (Data Access Objects) et la base de données, simplifiant les interactions.

## Composants Implémentés

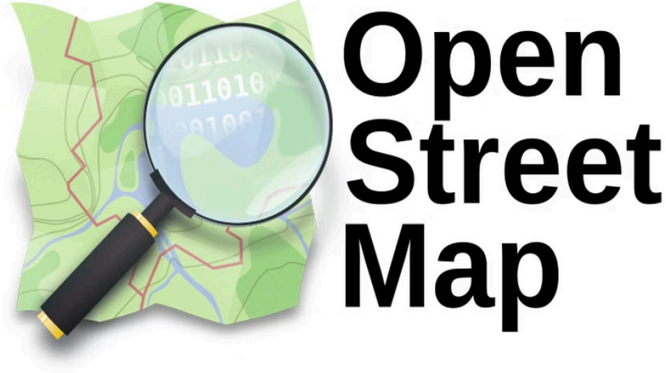
- `AppDatabase`**:** La classe abstraite principale de la base de données.
- `LocationDao`**:** Interface pour les opérations CRUD.
- `LocationEntity`**:** Représente un centre Dawan dans la base de données.

## Stratégie de Synchronisation

Lors d'un rafraîchissement, les données de l'API sont mappées, puis insérées dans une transaction qui vide et remplit la base de données locale. La source d'affichage est toujours la base de données locale.

# Affichage Cartographique avec OSMDroid

Plutôt que Google Maps, nous avons choisi **OSMDroid** pour sa flexibilité et son approche open-source, offrant une alternative robuste.




## Implémentation

- **MapView :** Intégrée directement dans le layout de l'activité.
- **Marqueurs Dynamiques :** Les épingles sont ajoutées à la carte dynamiquement à partir des coordonnées latitude/longitude des centres.
- **Gestion des Clics :** Un écouteur de clic sur chaque marqueur déclenche l'ouverture du BottomSheet de détails.

## Avantages d'OSMDroid

- **Open-source :** Aucune clé API nécessaire, réduction des coûts et dépendances.
- **Fonctionnalités Riches :** Supporte les marqueurs, le zoom, la rotation, et un cache pour l'offline.

# Détails du Centre : BottomSheet Intégrée

L'interaction avec les épingles de la carte ouvre une **BottomSheetDialogFragment**, un composant d'interface utilisateur familier sur Android.

## Composant Dédié

Nous utilisons un `LocationDetailsBottomSheet` pour présenter les informations de manière non intrusive.

## Déclenchement au Clic

Au clic sur une épingle, l'application récupère l'objet `Location` associé et le passe au BottomSheet.

## Contenu Affiché

Le BottomSheet affiche le nom et l'adresse complète du centre, tels que fournis par l'API et stockés en base de données.

```
 * <p>
 * On récupère les arguments (via {@link #getArguments()}),
 * puis on remplit les TextView du layout
 * {@code fragment_location_details.xml}.
 * </p>
 *
 * <ul>
 *     <li>{@code title} : affiche le nom du centre (par défaut "Centre")</li>
 *     <li>{@code addr} : affiche l'adresse, suivie du code postal et de la ville</li>
 * </ul>
 *
 * @param inflater  utilisé pour "gonfler" le layout XML
 * @param container parent du fragment
 * @param savedInstanceState état sauvegardé si recréation
 * @return la vue racine du fragment
 */
@Nullable
@Override
public View onCreateView(@NonNull LayoutInflater inflater,
                         @Nullable ViewGroup container,
                         @Nullable Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_location_details, container,  attachToRoot: false);

    Bundle args = getArguments();
    TextView title = v.findViewById(R.id.title);
    TextView addr = v.findViewById(R.id.address);

    if (args != null) {
        // Nom du centre
        title.setText(args.getString(ARG_NAME,  defaultValue: "Centre"));

        // Construction de l'adresse complète
        String a = args.getString(ARG_ADDR,  defaultValue: "");
        String pc = args.getString(ARG_PC,  defaultValue: "");
        String c = args.getString(ARG_CITY,  defaultValue: "");

        addr.setText(a + (pc.isEmpty() && c.isEmpty() ? "" : "\n" + pc + " " + c));
    }

    return v;
```

# Qualité et Bonnes Pratiques

La robustesse et la maintenabilité de l'application sont assurées par l'application rigoureuse des bonnes pratiques de développement.

### Séparation des Responsabilités (SRP)

Chaque couche (UI, Data, Domain, Repository) a une responsabilité unique, évitant les couplages forts.

### Mappers

Des mappers sont utilisés pour convertir les DTOs de l'API en entités Room, puis en modèles de domaine, assurant le découplage.

### Gestion des Exceptions

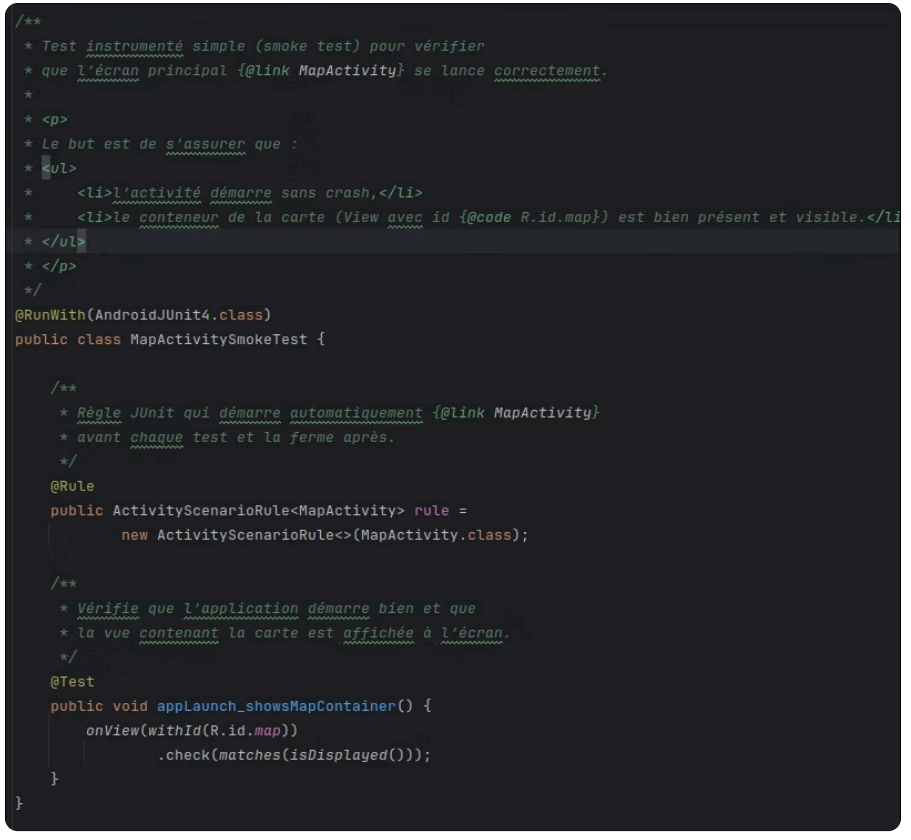
Les erreurs réseau, les échecs HTTP et les états hors ligne sont gérés proprement et remontés à l'UI pour informer l'utilisateur.

### Documentation Javadoc

Les classes et méthodes clés sont documentées avec Javadoc pour clarifier leur rôle, leurs paramètres et leur comportement.

# Tests

## Tests Instrumentés (Espresso)



```
/**
 * Test instrumenté simple (smoke test) pour vérifier
 * que l'écran principal {@link MapActivity} se lance correctement.
 *
 * <p>
 * Le but est de s'assurer que :
 * <ul>
 *     <li>l'activité démarre sans crash,</li>
 *     <li>le conteneur de la carte (View avec id {@code R.id.map}) est bien présent et visible.</li
 * </ul>
 * </p>
 */
@RunWith(AndroidJUnit4.class)
public class MapActivitySmokeTest {

    /**
     * Règle JUnit qui démarre automatiquement {@link MapActivity}
     * avant chaque test et la ferme après.
     */
    @Rule
    public ActivityScenarioRule<MapActivity> rule =
            new ActivityScenarioRule<>(MapActivity.class);

    /**
     * Vérifie que l'application démarre bien et que
     * la vue contenant la carte est affichée à l'écran.
     */
    @Test
    public void appLaunch_showsMapContainer() {
        onView(withId(R.id.map))
                .check(matches(isDisplayed()));
    }
}
```

- **Smoke Test :** Vérification du lancement de l'activité principale et de l'affichage correct de la vue carte.
- **Tests UI :** Possibilité d'ajouter des tests pour le bandeau hors ligne et les interactions des marqueurs.
- **Tests Unitaires :** Utilisation de mocks pour tester les mappers et la logique du Repository indépendamment.