M1 IASD

HAI811I - Programmation Mobile

Rapport de Conception & Architecture - TP1

HALABLIAN Hugo

Faculté des Sciences
Université de Montpellier

Février 2026

Sommaire

1 Introduction et Contexte

Ce document détaille les choix d'architecture et d'implémentation réalisés pour le premier rendu de Programmation Mobile (HAI811I). Ce TP couvre les fondamentaux du développement Android natif : *lifecycle* (cycle de vie), gestion des événements, *Intents* et persistance.

L'objectif de ce rapport est de présenter non seulement le résultat visuel, mais aussi la structure interne du projet, la mécanique des composants (Adapters, Manifeste) et les choix d'ingénierie logicielle qui garantissent la robustesse des applications.

2 Environnement Technique et Configuration

Un projet Android moderne ne repose pas uniquement sur du code source, mais sur un écosystème de configuration rigoureux.

2.1 Gestion des Dépendances (*Gradle*)

L'outil *Gradle* joue un rôle central dans notre projet. Il permet d'automatiser la compilation et d'intégrer des bibliothèques tierces. Dans le fichier `build.gradle.kts`, nous avons défini :

- **minSdk (24) :** Pour garantir la compatibilité avec une large gamme de terminaux (Android 7.0+).
- **targetSdk (34) :** Pour bénéficier des dernières optimisations d'Android 14.
- **Dépendances :** Injection de `Gson` (pour le JSON) et `MaterialCalendarView` via `implementation`. Cela nous évite de gérer manuellement des fichiers `.jar` et assure la résolution automatique des conflits de versions.

2.2 Organisation des Ressources (`res/`)

Une bonne architecture Android sépare strictement le code Kotlin des ressources statiques. Notre projet respecte la structure standard :

- **`res/layout`** : Contient les fichiers XML décrivant l'interface graphique (Vue).
- **`res/values`** : Centralise les constantes. `strings.xml` pour les textes, `colors.xml` pour la charte graphique et `themes.xml` pour l'unité visuelle globale.

2.3 Gestion des Densités d'Écran (`dp` vs `sp`)

Un point crucial du développement mobile est l'adaptation aux différentes densités d'écrans (ldpi, mdpi, xhdpi). Dans nos fichiers XML, nous avons appliqué la règle d'or Android :

- **`dp` (Density-independent Pixels) :** Utilisé pour les dimensions de layout (marges, tailles de boutons). Cela garantit qu'un bouton de `60dp` aura la même taille physique sur une tablette ou un petit téléphone.
- **`sp` (Scale-independent Pixels) :** Utilisé exclusivement pour la taille des textes (`textSize`). Cela permet de respecter les préférences d'accessibilité de l'utilisateur (si l'utilisateur a configuré « Police Grande » dans ses paramètres système, notre application s'adapte automatiquement).

3 Architecture Globale (Exercices 1 & 2)

3.1 Le Cœur du Système : `AndroidManifest.xml`

Au-delà du code, le fichier `AndroidManifest.xml` agit comme le « passeport » de l'application auprès du système d'exploitation. Pour nos trois applications, nous avons dû configurer ce fichier pour :

1. **Déclarer les Activités :** Chaque classe `Activity` créée (`AgendaActivity`, `TrainsActivity`) a été enregistrée ici. Sans cette déclaration, toute tentative de navigation via un `Intent` provoquerait une `ActivityNotFoundException`.

2. **Gérer le Point d'Entrée :** Nous avons configuré l'`intent-filter` avec l'action `MAIN` et la catégorie `LAUNCHER` pour définir quelle activité se lance au démarrage.
3. **Permissions :** Pour la fonctionnalité d'appel (App 1), bien que l'intent `ACTION_DIAL` ne nécessite pas de permission critique, une implémentation plus pousée utilisant `ACTION_CALL` aurait nécessité la déclaration `<uses-permission android:name="android.permission.CALL_PHONE" />`.

3.2 Modèle de Conception (*MVC Android*)

L'architecture globale de nos applications suit le pattern *MVC* (*Model-View-Controller*) natif d'Android :

- **La Vue (XML) :** Fichiers déclaratifs définissant uniquement la structure graphique.
- **Le Contrôleur (Activity Kotlin) :** Classes assurant le lien entre la vue et les données.
- **Le Modèle (Data & GSON) :** Isolation de la logique de *parsing* et des structures de données.

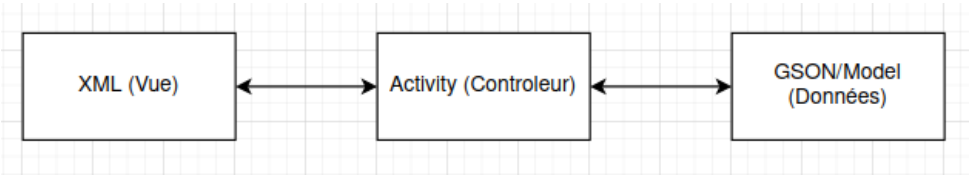

Fig. 1. – Schéma de l'architecture MVC mise en place : Séparation Vue / Activité / Données.

4 Exercices 3 à 7 : Formulaire et Cycle de Vie

Cette section couvre la création d'interface (Ex 3), l'internationalisation (Ex 4), les événements (Ex 5) et les *Intents* (Ex 6 & 7).

4.1 Comparaison : Interface Programmatique vs Déclarative (XML)

L'exercice 3 demandait de réaliser l'interface en code Kotlin (Java) et en XML. Bien que nous ayons implémenté la version programmatique (création de `LinearLayout` et `TextView` dynamiquement), nous avons privilégié l'approche XML pour la version finale.

Pourquoi ? Le XML offre une meilleure séparation des responsabilités (Vue vs Logique) et permet de prévisualiser le rendu dans Android Studio sans compiler (*Design Editor*). À l'inverse, l'approche programmatique est verbeuse et rend la maintenance de l'UI complexe (imbrication de `addView` difficile à lire).

4.2 Gestion des Erreurs et Validation (Robustesse)

Pour garantir l'intégrité des données, nous utilisons `setError()` sur les `EditText`. Cela fournit un retour visuel immédiat sans interrompre l'utilisateur avec des popups.

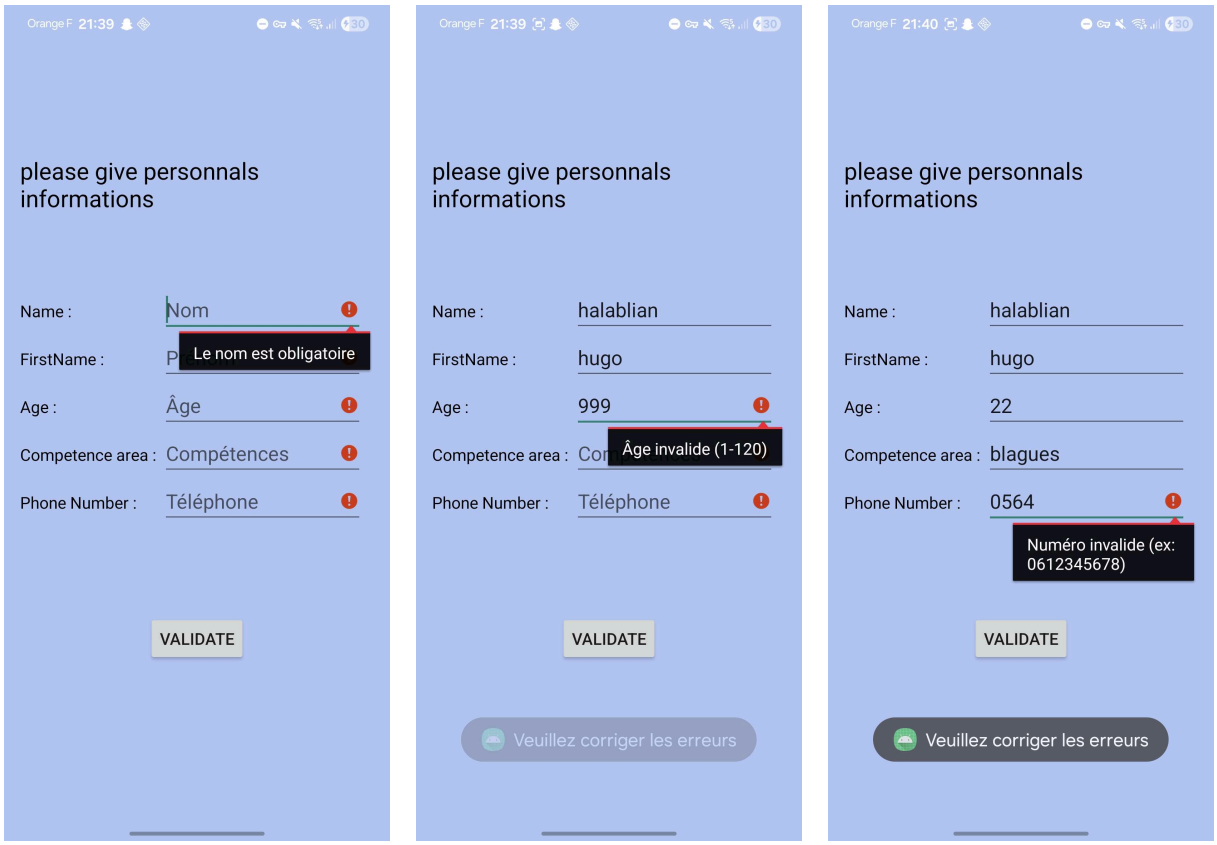

Fig. 2. – Gestion des cas d'erreurs : Champs vides, valeurs aberrantes et formatage incorrect.

4.3 Internationalisation et Cycle de Vie

L'application s'adapte dynamiquement à la langue du système (FR/EN) et résiste aux rotations d'écran grâce à la gestion du `Bundle`.

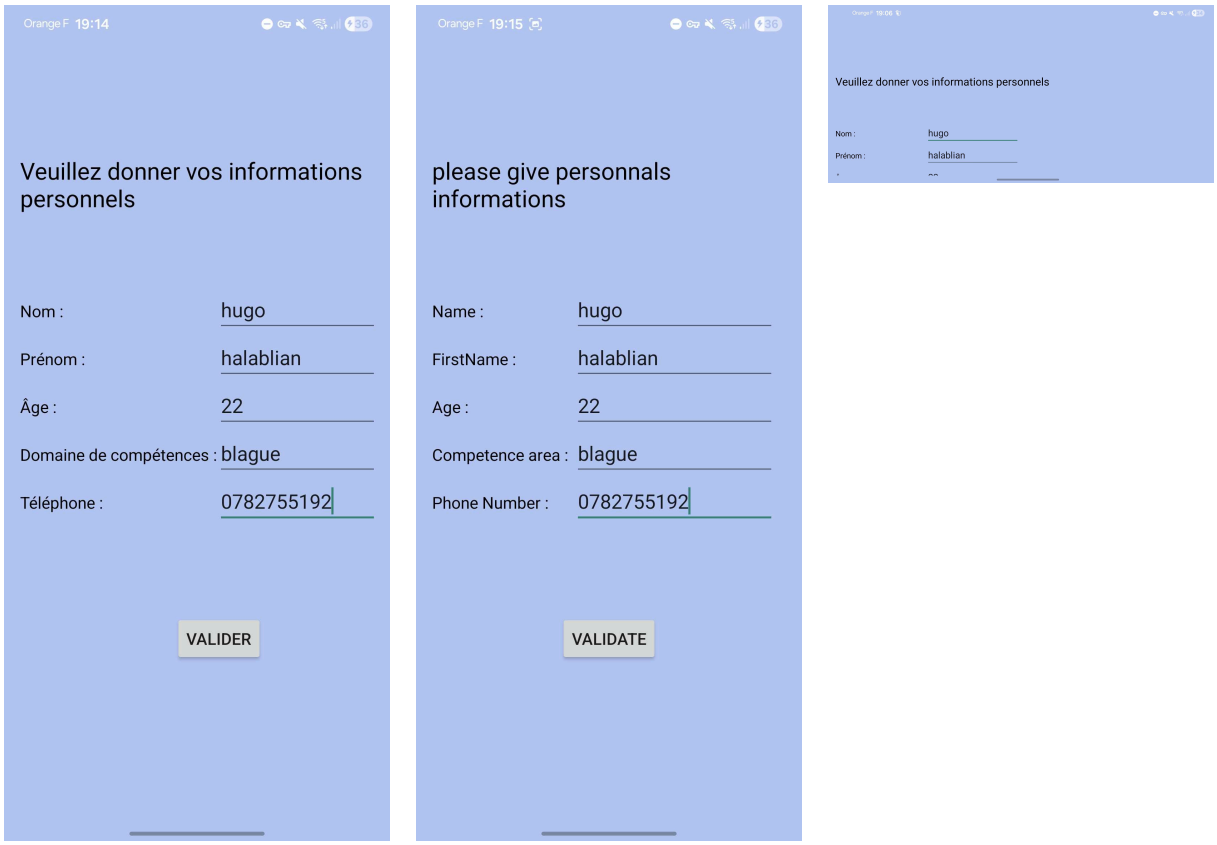

Fig. 3. – Internationalisation et conservation des données en mode paysage.

4.4 Syntaxe Kotlin : Lambdas et Null Safety

Le passage du Java au Kotlin pour ce TP a permis de simplifier drastiquement le code des *Listeners*. Au lieu d'utiliser des classes anonymes verbeuses (`new View.OnClickListener() { ... }`), nous avons utilisé la syntaxe des **Expressions Lambda** de Kotlin : `btn.setOnClickListener { view -> ... }`. De plus, la gestion des types nullables (`String?`) nous a permis d'éviter les célèbres `NullPointerException` lors de la récupération des données des *Intents* (`intent.getStringExtra(...)`).

5 Exercice 8 : Application Trains (Simulation Backend)

5.1 UX : Saisie Assistée

L'utilisation d'un champ texte libre pour une gare est source d'erreur. Nous avons implémenté un `AutoCompleteTextView` couplé à un `ArrayAdapter`. L'`ArrayAdapter` agit comme un pont : il prend la liste de villes (Données), crée une vue pour chaque item (Vue) et l'injecte dans le composant visuel.

Fig. 4. – Détail UX : L'Auto-complétion prévient les erreurs de saisie.

5.2 Architecture des Données

L'utilisation de la librairie *GSON* nous permet de mapper directement les fichiers JSON (`assets/trajets.json`) vers des objets Kotlin. Cela simule un appel API REST et prépare l'application pour une connexion réelle.

6 Exercice 9 : Agenda et Persistance Avancée

C'est l'application centrale du TP, nécessitant une interface complexe et une gestion d'état avancée.

6.1 Zoom Technique : Fonctionnement du Custom Adapter

L'affichage de la liste des événements utilise un `BaseAdapter` personnalisé. C'est un composant clé de l'architecture Android qui mérite une explication détaillée. Contrairement à l'appli « Trains » qui affichait du texte simple, l'Agenda doit afficher pour chaque ligne : un titre, une heure et une couleur de fond dynamique.

Pour réaliser cela, notre méthode `getView()` réalise une opération d'**Inflation** :

1. Elle vérifie si une vue recyclée est disponible (`convertView`) pour optimiser la mémoire.
2. Si non, elle utilise le `LayoutInflater` pour instancier le fichier XML `item_event.xml`.
3. Elle « bind » (lie) les données de l'objet `Evenement` courant (trouvé via `getItem(position)`) aux champs `TextView` de la vue.
4. Elle modifie dynamiquement la couleur de fond du layout via `setBackgroundColor`.

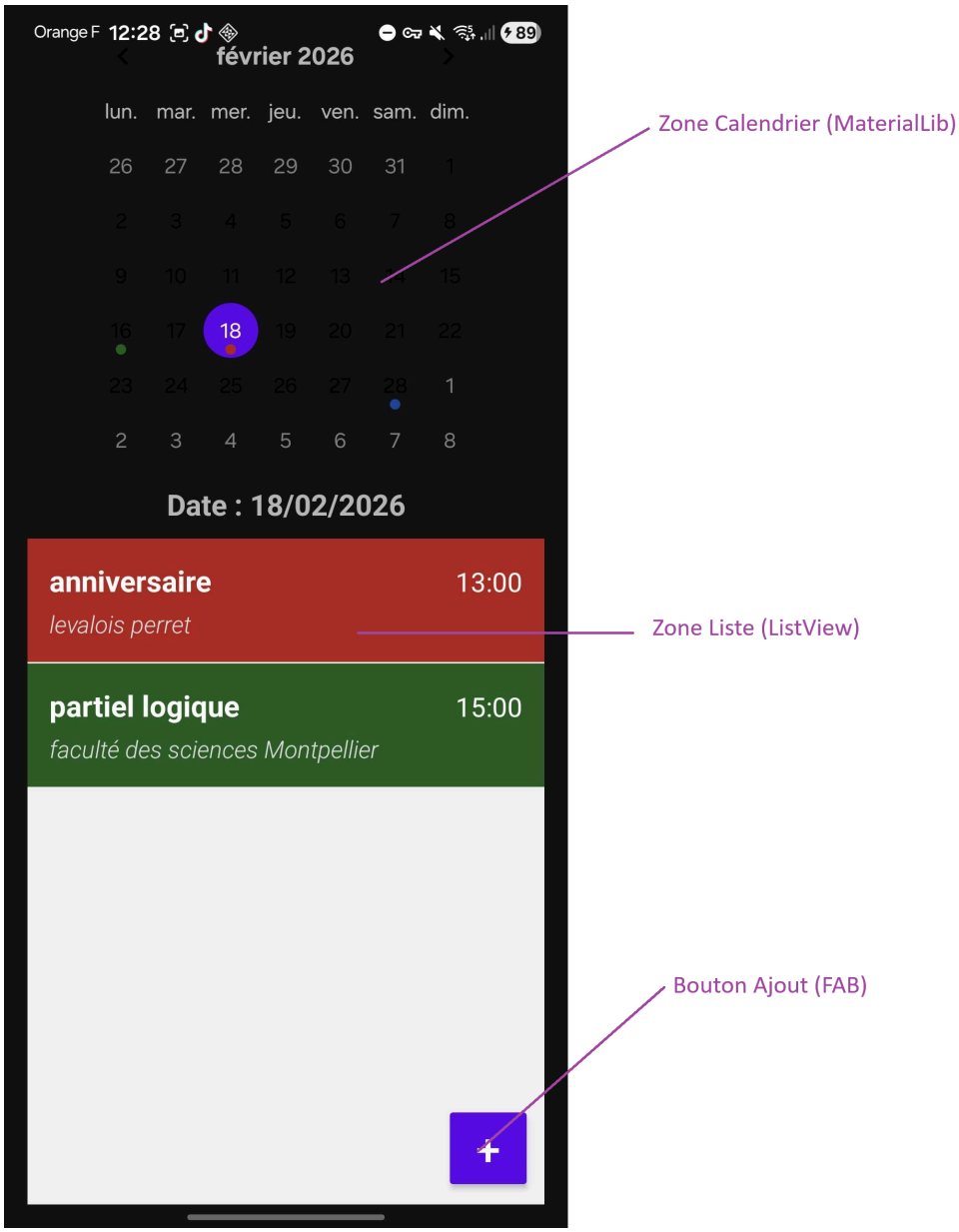

Fig. 5. – Décomposition structurelle de l'activité Agenda : Calendrier, Liste et FAB.

6.2 Workflow de Création et Suppression

L'expérience utilisateur a été pensée pour être fluide (« Storytelling »).

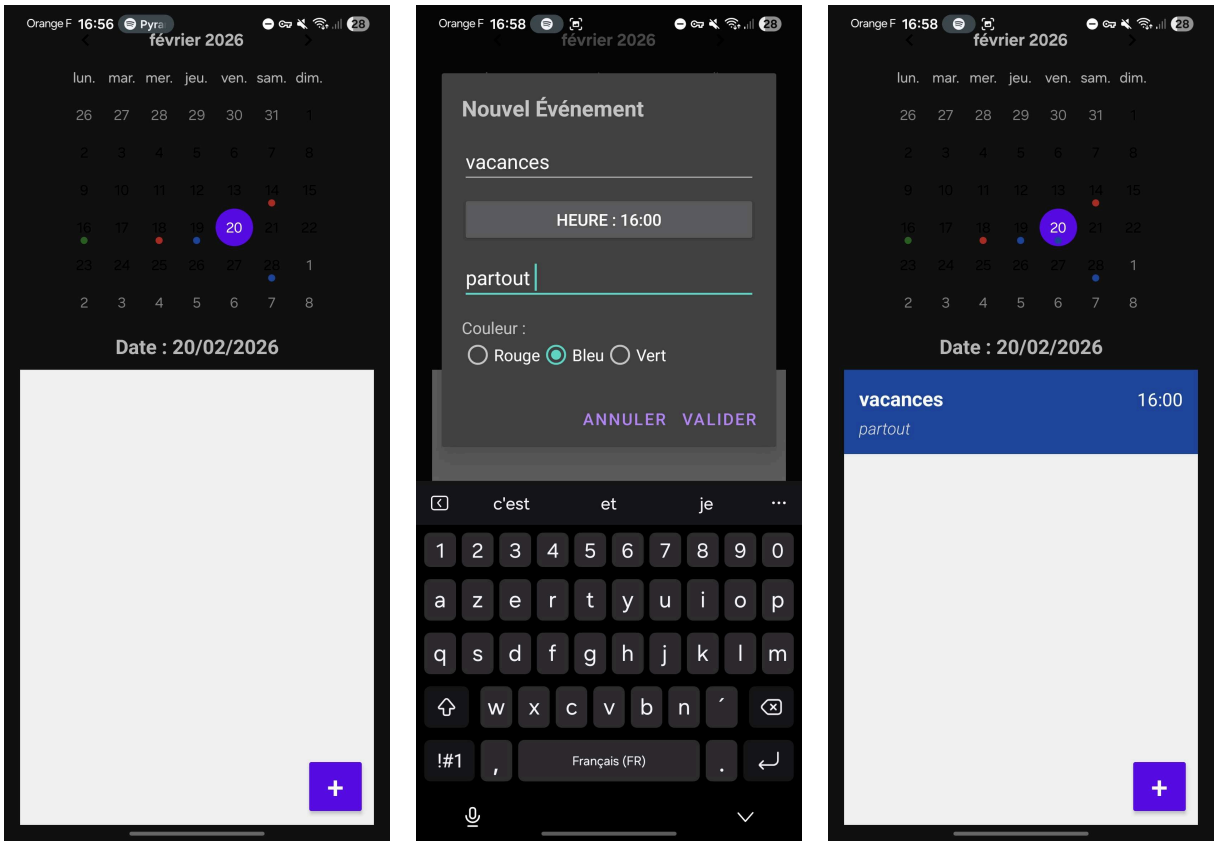

Fig. 6. – Séquence de création : Appel via le FAB, Saisie Dialog, Mise à jour ListView.

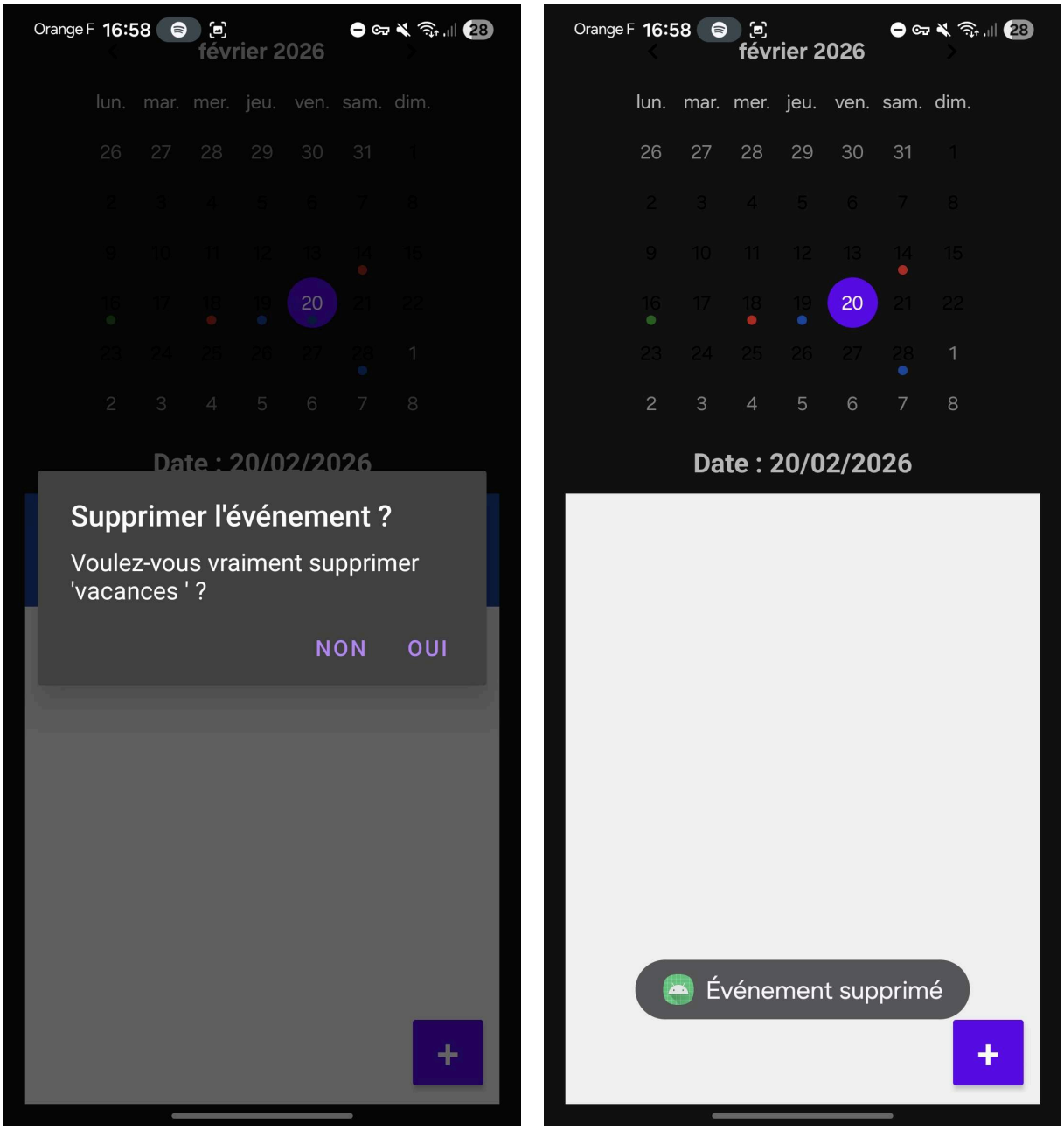

Fig. 7. – UX : Protection contre la suppression accidentelle via une boite de confirmation.

6.3 Persistance (JSON & *SharedPreferences*)

Le schéma ci-dessous illustre le flux de données : L'objet Kotlin est sérialisé en JSON, puis sauvegardé. Au démarrage, le processus inverse est exécuté.

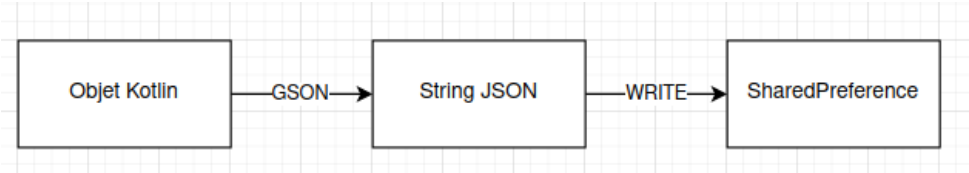

Fig. 8. – Flux de persistance : Objet Kotlin JSON string SharedPreferences.

7 Analyse Critique et Perspectives

Ce projet respecte les consignes fonctionnelles et va au-delà sur l'architecture. Cependant, des axes d'amélioration existent :

1. **Vers l'architecture *MVVM* :** La séparation actuelle des données prépare le terrain pour *MVVM*. Les appels GSON seraient déplacés dans un `Repository`, et les Activités s'abonneraient via `LiveData`.
2. **Base de Données :** Pour l'Agenda, sérialiser toute la liste en JSON est coûteux. L'intégration de *Room (SQLite)* serait la solution industrielle adaptée.
3. **API Distante :** Remplacer le fichier JSON local des trains par un appel *Retrofit* vers l'API SNCF Open Data.

8 Bibliographie & Dépendances

Les choix techniques s'appuient sur des standards industriels :

- **Gradle & SDK :** MinSDK 24, TargetSDK 34.
- **Google Gson (v2.10.1)** : Sérialisation JSON.
- **Material-CalendarView (v2.0.1)** : Calendrier avancé.
- **AndroidX Core & AppCompat** : Composants *Material Design*.