M1 IASD

HAI811I - Programmation Mobile

Rapport de Conception & Architecture - TP1

HALABLIAN Hugo

Faculté des Sciences
Université de Montpellier

Février 2026

Sommaire

1 Introduction

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 : cycle de vie, gestion des événements, *Intents* et persistance des données.

L'objectif de ce rapport est de présenter non seulement le résultat visuel, mais aussi la structure interne du projet, la maîtrise de l'écosystème Android (Gradle, Ressources) et les choix d'ingénierie logicielle qui garantissent la robustesse et la maintenabilité des applications.

2 1. Environnement Technique et Gestion de Projet

Un projet Android moderne ne repose pas uniquement sur du code source Kotlin, mais sur un écosystème de configuration rigoureux qui orchestre la compilation et les ressources.

2.1 1.1 Gestion des Dépendances avec *Gradle*

L'outil *Gradle* joue un rôle central dans notre projet. Contrairement à une gestion manuelle de bibliothèques (fichiers `.jar`), il permet d'automatiser la résolution de dépendances et la compilation.

Dans le fichier `build.gradle.kts` (niveau module), nous avons configuré les paramètres suivants :

- **minSdk (24) :** Nous ciblons Android 7.0 (Nougat) comme version minimale. Cela garantit que l'application fonctionne sur environ 94% des terminaux actifs dans le monde.
- **targetSdk (34) :** Nous ciblons la dernière version stable (Android 14) pour bénéficier des optimisations de performance et de sécurité les plus récentes.
- **Dépendances Tierces :** L'intégration de fonctionnalités complexes a été simplifiée par l'injection de dépendances :
  - `com.google.code.gson:gson`: Pour la sérialisation JSON.
  - `com.github.prolificinteractive:material-calendarview`: Pour l'affichage avancé du calendrier (nécessitant l'ajout du dépôt *JitPack*).

2.2 1.2 Organisation des Ressources (`res/`)

Une bonne architecture Android sépare strictement la logique (Kotlin) de la présentation (XML). Notre projet respecte la structure standardisée :

- **`res/layout`** : Contient les fichiers XML décrivant l'interface graphique. Chaque Activité possède son propre fichier de mise en page (ex: `activity_agenda.xml`), favorisant la modularité.
- **`res/values`** : Ce dossier est crucial pour la maintenabilité :
  - `strings.xml` : Centralise tous les textes. C'est ce mécanisme qui nous permet de gérer l'internationalisation (FR/EN) sans modifier le code.
  - `colors.xml` : Définit la palette de couleurs (ex: le violet `#6200EE` de la charte graphique).
  - `themes.xml` : Assure une cohérence visuelle globale (Thème Sombre/Clair) sur l'ensemble des trois applications.

2.3 1.3 Gestion des Densités d'Écran (`dp` vs `sp`)

Pour garantir que l'interface reste utilisable sur tous les appareils (de la petite montre à la grande tablette), nous avons appliqué la règle d'or des unités Android :

- **`dp` (Density-independent Pixels) :** Utilisé pour les marges et tailles de composants. Android redimensionne automatiquement ces valeurs selon la densité de pixels de l'écran (*dpi*).
- **`sp` (Scale-independent Pixels) :** Utilisé exclusivement pour la taille des textes. Cette unité respecte les préférences d'accessibilité de l'utilisateur (taille de police système).

3 2. Architecture Globale (Exercices 1 & 2)

3.1 2.1 Le Rôle du Manifeste (`AndroidManifest.xml`)

Le fichier `AndroidManifest.xml` est le point d'entrée névralgique. Il déclare les composants de l'application au système d'exploitation. Pour nos trois applications, nous avons configuré :

- Les déclarations d'`Activity` pour chaque écran.
- Les *Intent-Filters* (`ACTION_MAIN`, `CATEGORY_LAUNCHER`) pour définir l'écran d'accueil.
- Les permissions implicites nécessaires au bon fonctionnement des Intents système.

3.2 2.2 Modèle de Conception (*MVC Android*)

L'architecture suit le pattern *MVC* (*Model-View-Controller*) natif :

- **Vue (XML) :** Structure passive de l'interface.
- **Contrôleur (Activity) :** Gestion des interactions utilisateur et du cycle de vie.
- **Modèle (Data Classes) :** Structures de données (ex: `Evenement`, `Trajet`).

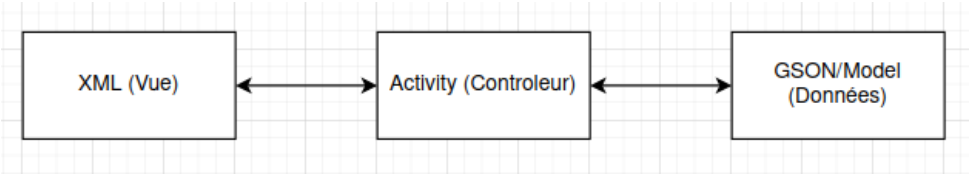

Fig. 1. – Schéma de l'architecture MVC mise en place : Séparation Vue / Activité / Données.

3.3 2.3 Stratégie de Navigation

Nous avons opté pour une architecture *Multi-Activities* pour manipuler explicitement les *Intents* et la *Back Stack*.

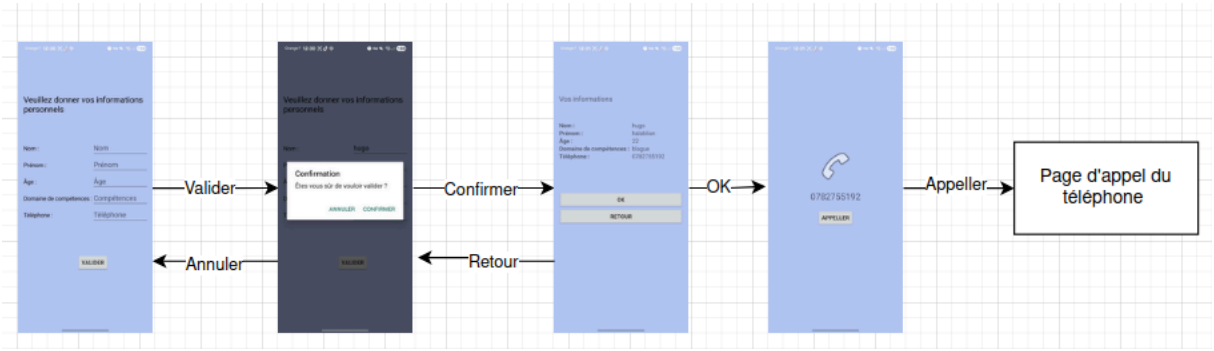

Fig. 2. – Graphe de navigation réel de l'Application 1 montrant le flux des Intents.

4 3. Exercices 3 à 7 : Formulaire et Cycle de Vie

4.1 3.1 Analyse Comparative : XML vs Programmatique

L'exercice 3 demandait de réaliser l'interface de deux manières. Voici notre analyse des deux approches :

1. **Approche Déclarative (XML) :**
   - **Avantages :** Séparation claire Vue/Logique, prévisualisation immédiate dans Android Studio, code plus lisible.
   - **Inconvénient :** Moins flexible si l'interface doit changer radicalement selon des données reçues au runtime.
2. **Approche Programmatique (Kotlin) :**

- **Avantages :** Création dynamique de vues (ex: ajouter N champs selon une réponse serveur).
- **Inconvénient :** Code verbeux, difficile à maintenir, pas de prévisualisation.

Choix Final : Pour la suite du TP, nous avons privilégié le XML pour sa robustesse et sa maintenabilité.

4.2 3.2 Robustesse et Validation (Exercice 5)

Pour garantir l'intégrité des données, nous utilisons `setError()` sur les `EditText`. Cela fournit un retour visuel immédiat (*Feedback*).

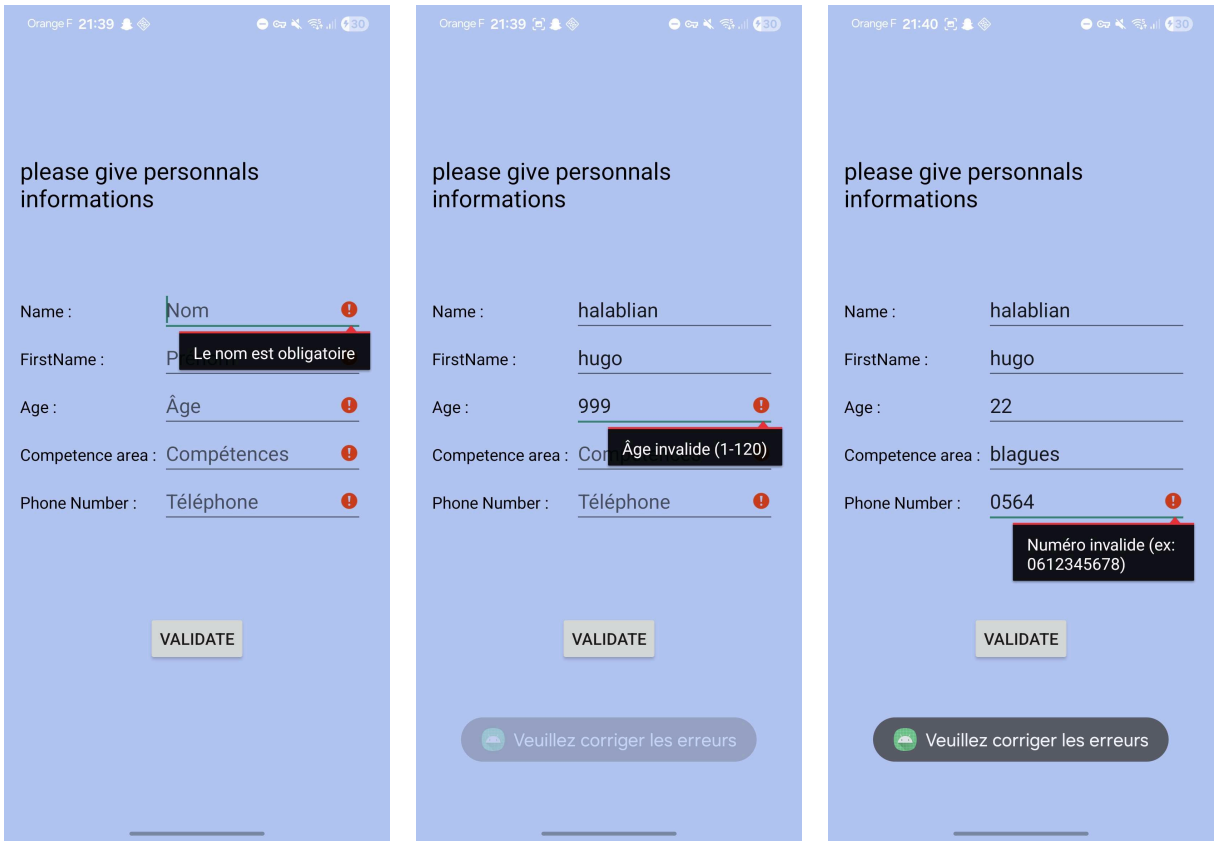

Fig. 3. – Gestion des cas d'erreurs : Champs vides, valeurs aberrantes et formatage incorrect.

4.3 3.3 Internationalisation et Rotation

L'application gère nativement le changement de langue et la rotation d'écran (conservation des données via `Bundle`).

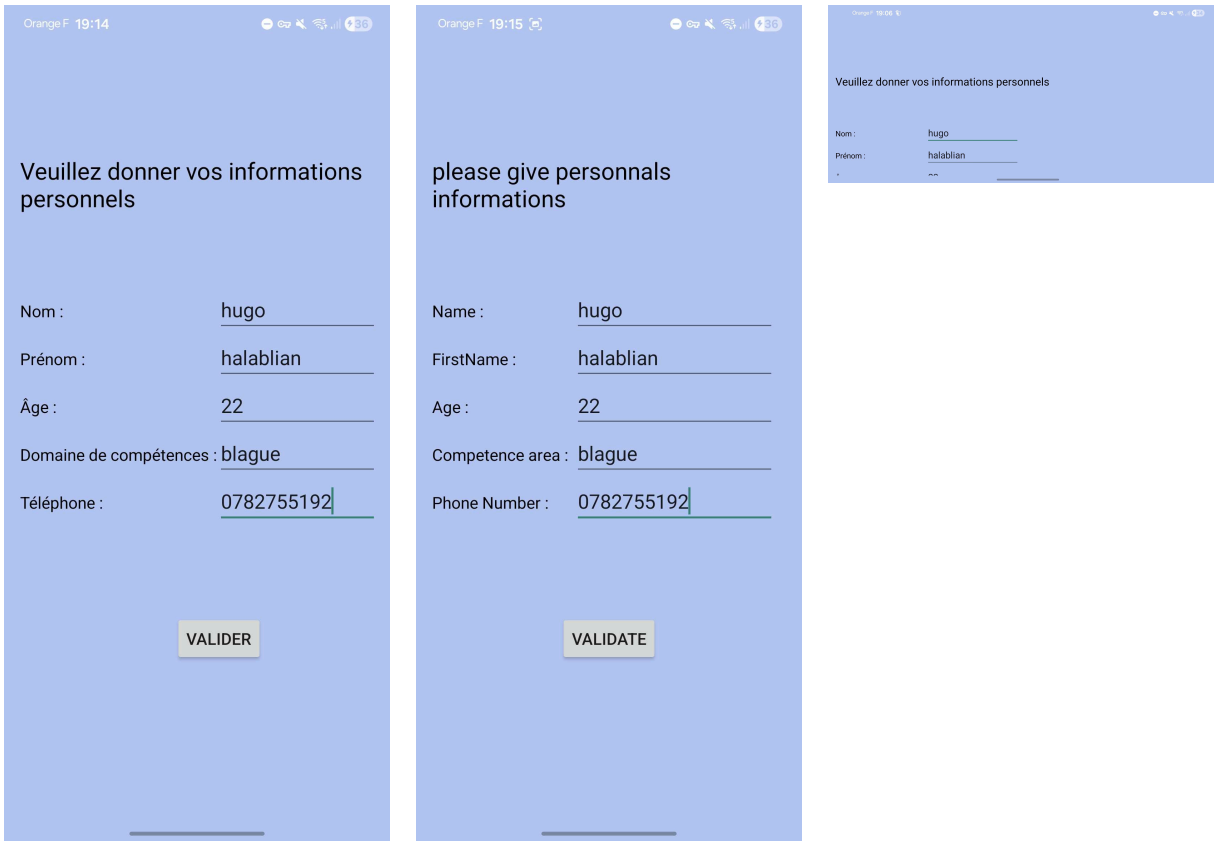

Fig. 4. – Internationalisation (FR/EN) et résilience au changement de configuration (Paysage).

5 4. Exercice 8 : Application Trains (Simulation Backend)

L'exercice 8 consistait à réaliser une application de consultation d'horaires. Nous avons conçu une interface en deux temps : recherche et résultats.

5.1 4.1 Interface et Choix Ergonomiques (UI/UX)

L'interface a été pensée pour être sobre et efficace, respectant le thème sombre global de nos applications.

Écran de Recherche : Nous avons centré les éléments dans un `ConstraintLayout` pour une mise en page fluide sur tous les écrans. Le choix majeur d'UX est l'utilisation de l'`AutoCompleteTextView`. Plutôt que de laisser l'utilisateur taper « Montpellier » avec le risque de fautes de frappe, nous filtrons dynamiquement une liste de villes valides. Cela garantit que la recherche aboutira toujours.

Écran de Résultats : La page de résultats affiche clairement le trajet sélectionné en rouge pour rappeler le contexte. La liste des horaires est présentée de manière épurée. Nous avons ajouté un bouton « Retour à la recherche » explicite, bien que le bouton « Retour » natif d'Android soit également géré par la fermeture de l'activité (`finish()`).

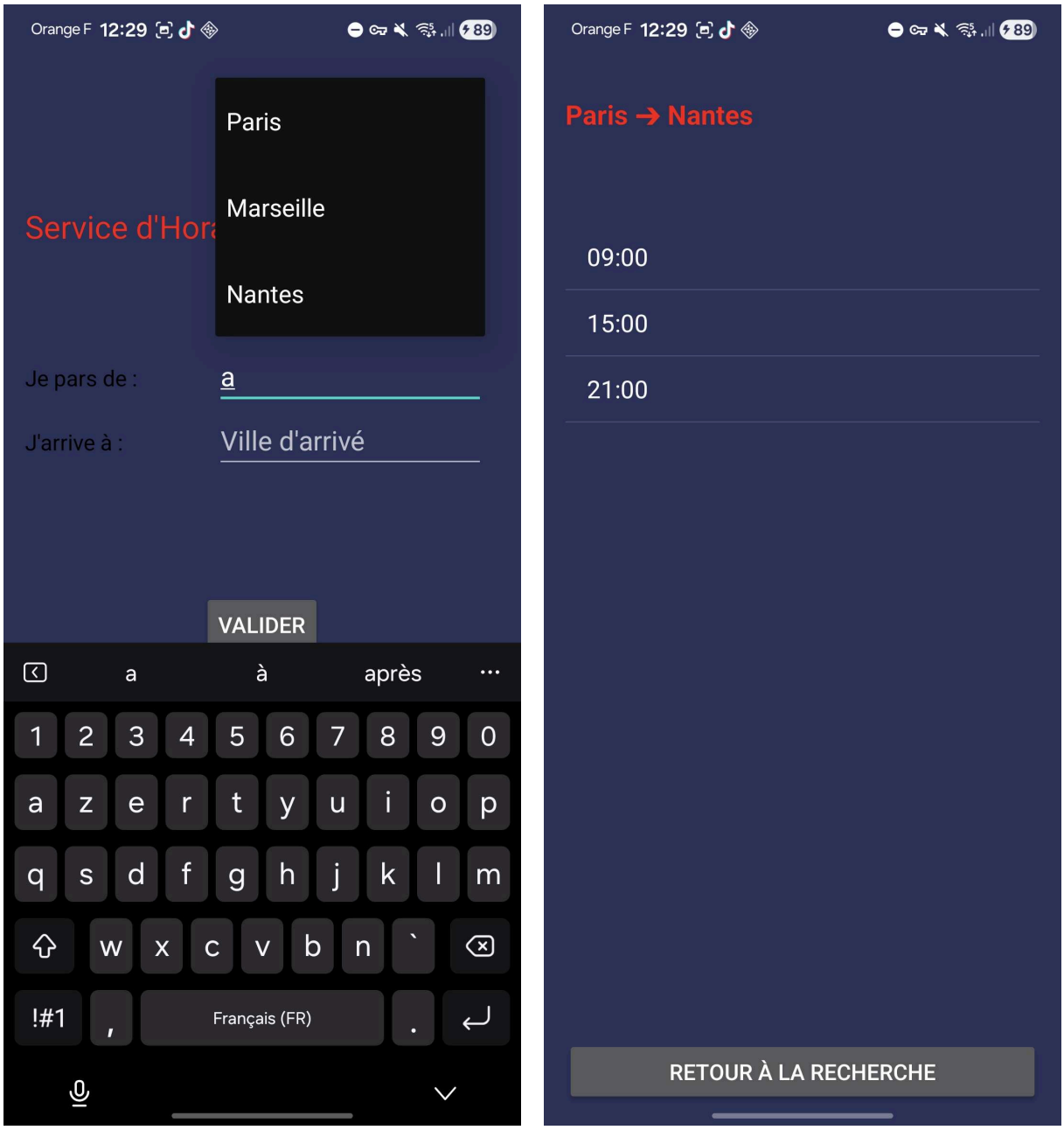

Fig. 5. – Flux complet de l'Application 2 : De la recherche assistée à l'affichage des horaires.

5.2 4.2 Architecture des Données

Pour simuler le backend, nous utilisons la librairie *GSON*. Au démarrage, l'application lit le fichier `assets/trajets.json`, le désérialise en objets Kotlin `Trajet` et alimente l'adapter d'autocomplétion. Lors de la validation, les critères (Ville Départ / Arrivée) sont passés via `Intent` à la seconde activité, qui filtre cette liste en mémoire pour n'afficher que les horaires correspondants.

6 5. 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 5.1 Structure de l'Interface (Wireframe)

L'écran principal s'articule autour de trois composants majeurs :

1. **Zone Calendrier :** Affiche le mois et les indicateurs (points).

2. **Zone Liste :** Affiche le détail des événements du jour sélectionné.
3. **Action (FAB) :** Permet l'ajout rapide.

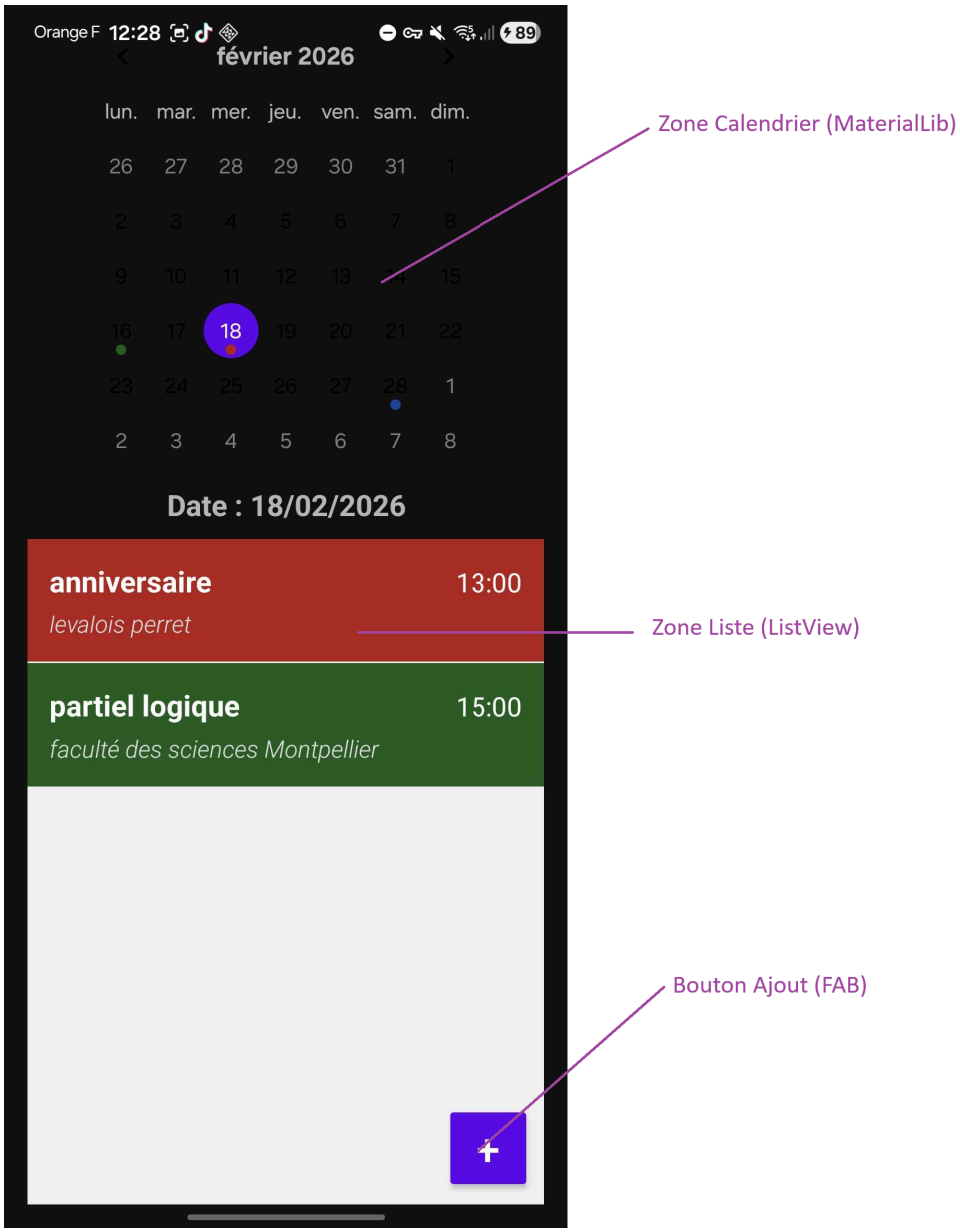

Fig. 6. – Structure de l'activité Agenda : MaterialCalendarView (Haut), ListView (Bas), FAB.

6.2 5.2 Zoom Technique : Fonctionnement du Custom Adapter

Pour afficher la liste des événements avec un design personnalisé (couleur de fond, heure alignée à droite), nous avons implémenté un `BaseAdapter`. La méthode clé est `getView()`. Elle utilise un `LayoutInflater` pour « gonfler » (instancier) le fichier XML `item_event.xml` pour chaque ligne de la liste, puis remplit les champs textuels avec les données de l'objet `Evenement`. C'est ce mécanisme qui permet une personnalisation totale des listes Android.

6.3 5.3 Workflow Utilisateur : Création d'un Événement

L'expérience utilisateur a été pensée pour être fluide et guidée. La séquence ci-dessous illustre le parcours complet :

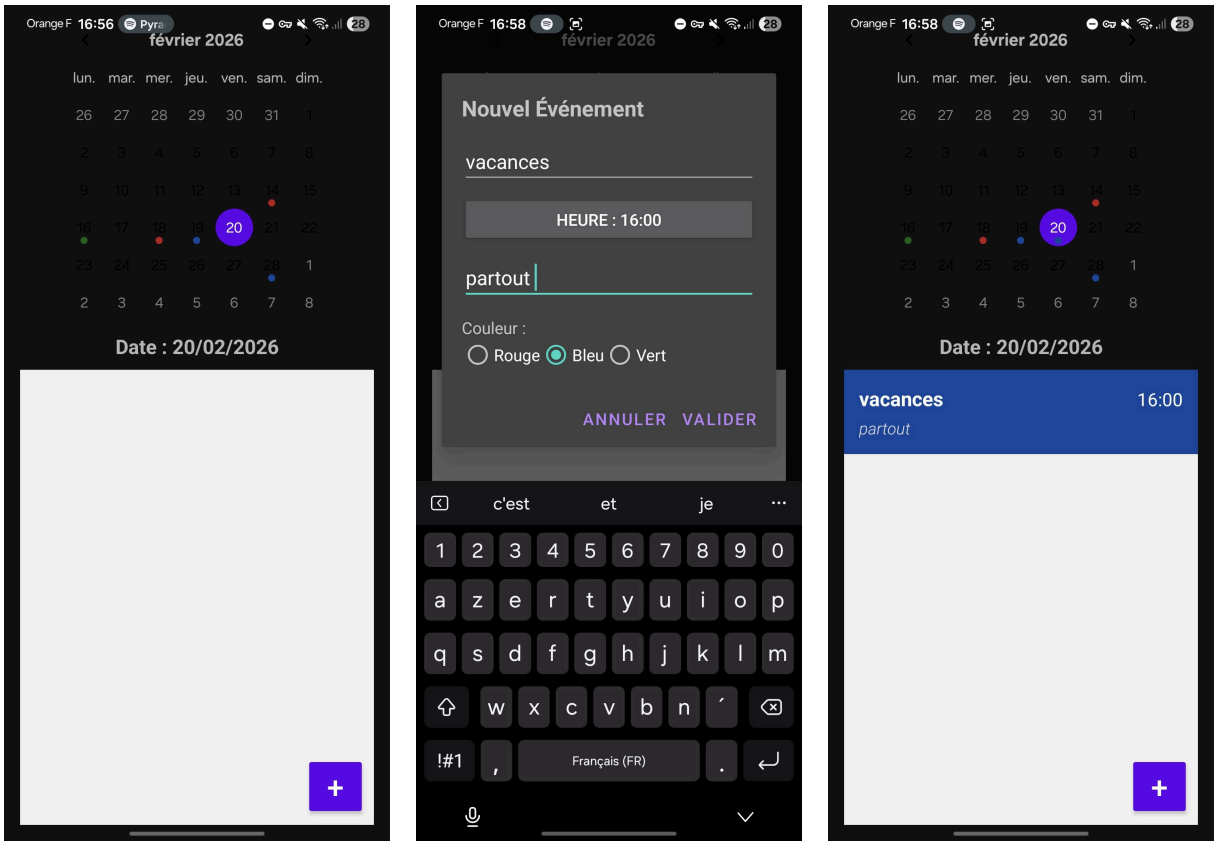

Fig. 7. – Scénario de création : 1. Clic sur le FAB, 2. Saisie dans la Dialog modale, 3. Mise à jour temps réel de la liste et du marqueur.

6.4 5.4 Feature Bonus : Suppression

Pour améliorer l'ergonomie, nous avons implémenté un *Long Click Listener* sur les éléments de la liste.

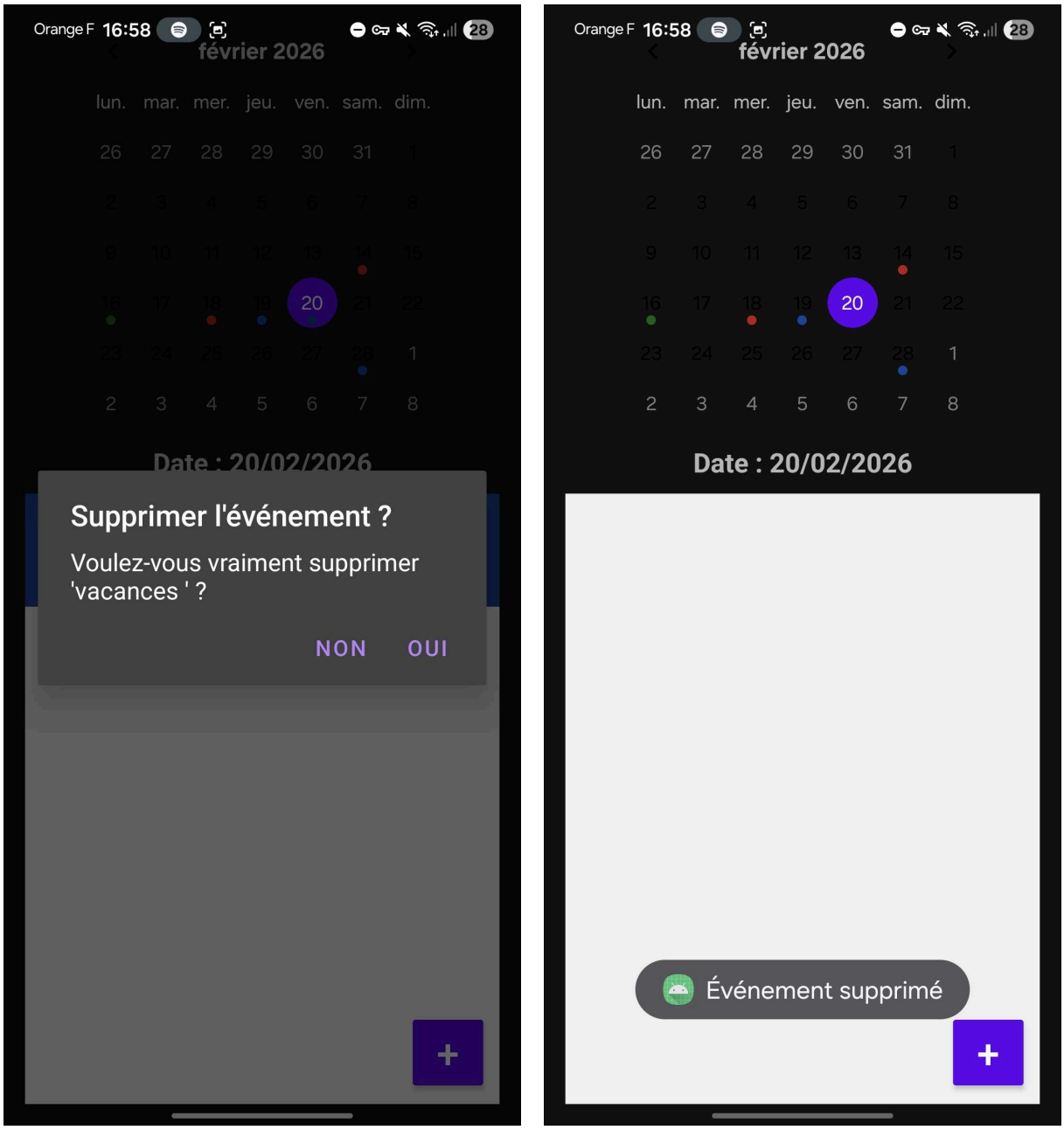

Fig. 8. – Sécurité UX : Demande de confirmation avant suppression définitive.

6.5 5.5 Persistance des Données

Le schéma ci-dessous illustre le cycle de vie des données : Objet Kotlin → Sérialisation JSON (GSON) → Stockage (SharedPreferences).

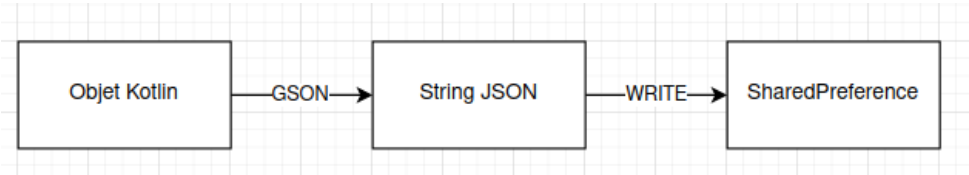

Fig. 9. – Flux de persistance : Sauvegarde locale via sérialisation JSON.

7 6. Analyse Critique et Perspectives

Ce projet respecte les consignes fonctionnelles et va au-delà sur l'architecture. Cependant, pour une mise en production industrielle, plusieurs évolutions seraient nécessaires :

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 Relationnelle :** Pour l'Agenda, sérialiser toute la liste en JSON est coûteux en performances. L'intégration de *Room (SQLite)* serait la solution adaptée pour gérer des milliers d'événements.
3. **Connectivité API :** Remplacer le fichier JSON local des trains par un appel *Retrofit* vers l'API SNCF Open Data pour obtenir des horaires en temps réel.

8 Bibliographie & Outils

Les choix techniques s'appuient sur des standards industriels :

- **Gradle & SDK :** MinSDK 24, TargetSDK 34.
- **Google Gson (v2.10.1)** : Sérialisation JSON performante.
- **Material-CalendarView (v2.0.1)** : Composant UI avancé (dépot JitPack).
- **AndroidX Core & AppCompat** : Composants de rétro-compatibilité.