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.

Au-delà de la simple réalisation fonctionnelle des exercices, ce travail se concentre sur la mise en place d'une architecture robuste et maintenable. Nous avons structuré ce rapport en suivant la progression logique des exercices demandés.

2 Architecture Globale (Exercices 1 & 2)

Avant d'aborder les applications spécifiques, il convient de définir le cadre technique global du projet.

2.1 Modèle de Conception (*MVC Android*)

L'architecture globale de nos applications suit le pattern *MVC* (*Model-View-Controller*) natif d'Android, mais avec une attention particulière portée à la séparation des responsabilités pour faciliter une future évolution :

- **La Vue (XML) :** Fichiers déclaratifs (`layout/*.xml`) définissant uniquement la structure graphique et les identifiants.
- **Le Contrôleur (Activity Kotlin) :** Classes (`.kt`) assurant le lien entre la vue et les données.
- **Le Modèle (Data & GSON) :** Nous avons isolé la logique de *parsing* (*GSON*) et les structures de données (`data classes`) hors des activités.

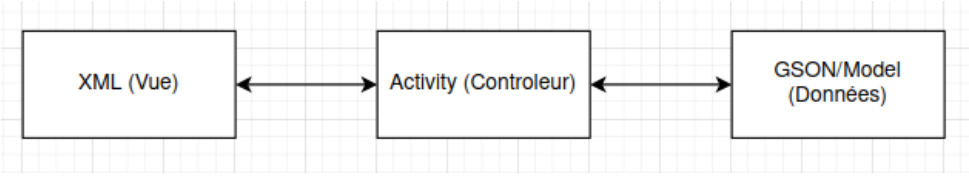

Fig. 1. – Schéma de l'architecture MVC mise en place : Séparation Vue / Activité / Données.

2.2 Stratégie de Navigation

Nous avons fait le choix d'une architecture *Multi-Activities*. Contrairement à une architecture *Single-Activity* (*Fragments*), ce choix permet de manipuler explicitement les *Intents* et de comprendre la gestion de la *Back Stack* (pile d'activités) par l'OS Android.

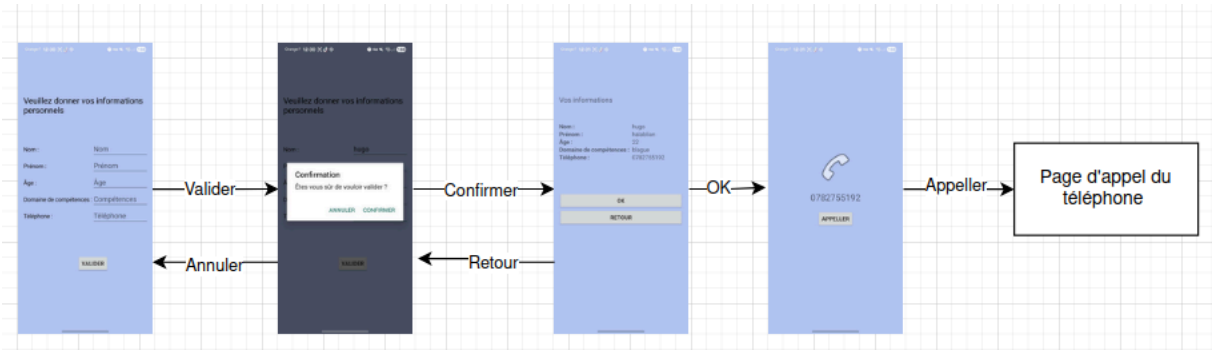

Fig. 2. – Graphe de navigation réel de l'Application 1 montrant l'enchaînement des Activités.

3 Exercices 3 à 7 : Formulaire, Cycle de Vie et Intents

Cette première application implémente un flux complet de saisie utilisateur, couvrant la création d'interface (Ex 3), l'internationalisation (Ex 4), les événements (Ex 5) et les *Intents* (Ex 6 & 7).

3.1 Choix Ergonomiques & UX (*Look & Feel*)

Pour l'exercice 3, nous ne nous sommes pas contentés de champs textes standards. Nous avons configuré les `inputType` XML pour améliorer l'expérience utilisateur :

- Le champ « Téléphone » ouvre automatiquement le clavier numérique.
- Le champ « Nom » met automatiquement la première lettre en majuscule.

C'est ce genre de détail qui différencie une application prototype d'une application utilisable.

3.2 Internationalisation (Exercice 4)

Conformément à la consigne, aucune chaîne de caractères n'est codée « en dur » (*hardcoded*). L'utilisation des ressources `res/values/strings.xml` et `res/values-en/strings.xml` permet une adaptation dynamique.

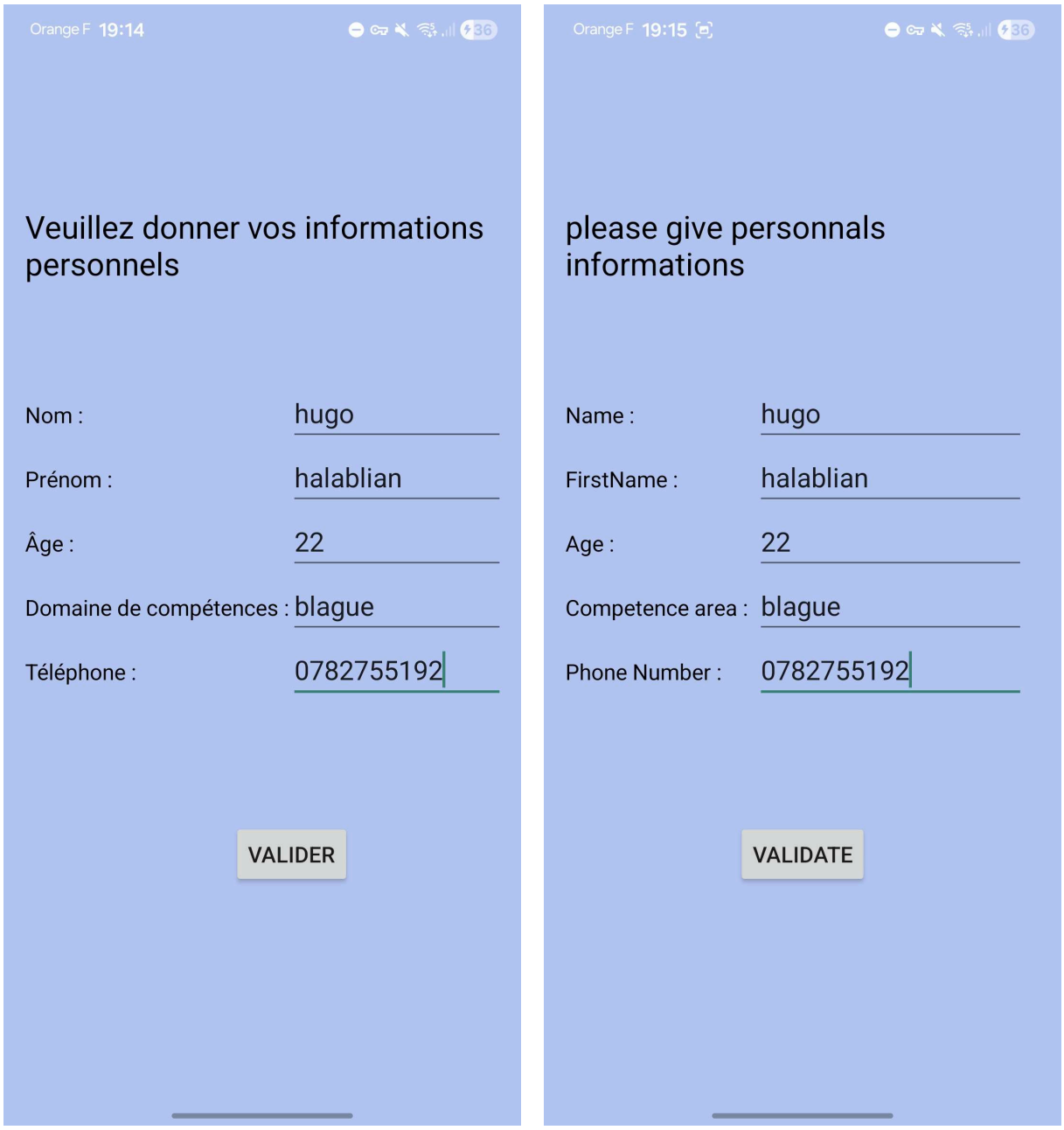

Fig. 3. – Preuve de l'internationalisation : Interface s'adaptant à la locale (FR/EN).

3.3 Robustesse et Cycle de Vie (Exercice 5)

Un défi classique est la perte de données lors de la rotation de l'écran. Grâce à l'utilisation correcte des `Bundles`, nous garantissons la continuité de l'expérience. De plus, une validation bloquante empêche la soumission de formulaires vides.

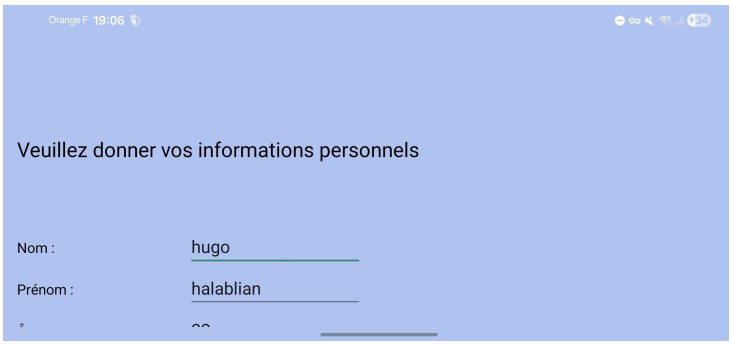

Fig. 4. – Preuve de robustesse : conservation des données en mode paysage.

3.4 Gestion des Erreurs et Validation (Exercice 3 & 5)

Afin de garantir l'intégrité des données saisies, nous avons implémenté une couche de validation stricte utilisant la méthode `setError()` native d'Android. Celle-ci fournit un retour visuel immédiat à l'utilisateur sans nécessiter de popup intrusive.

Nous avons identifié et traité plusieurs cas d'erreurs courants :

1. **Champs vides :** Empêche la validation si des informations obligatoires (Nom, Prénom) manquent.
2. **Format invalide :** Vérification de la cohérence de l'âge (entre 0 et 120 ans).
3. **Validation Regex :** Le numéro de téléphone est validé par une expression régulière stricte (`^0[1-9][0-9]{8}$`) pour garantir un format français valide.

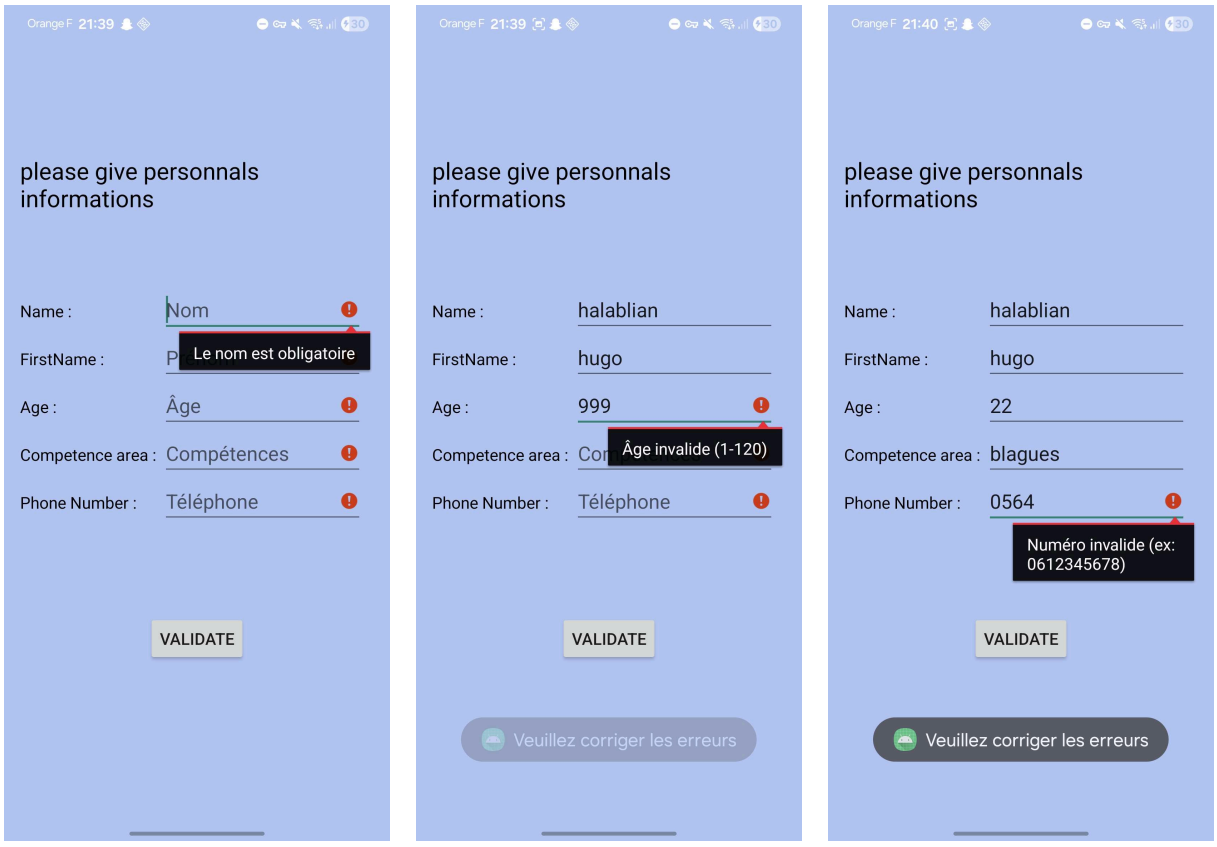

Fig. 5. – Gestion des cas d'erreurs : Champs vides, valeurs aberrantes et formatage incorrect.

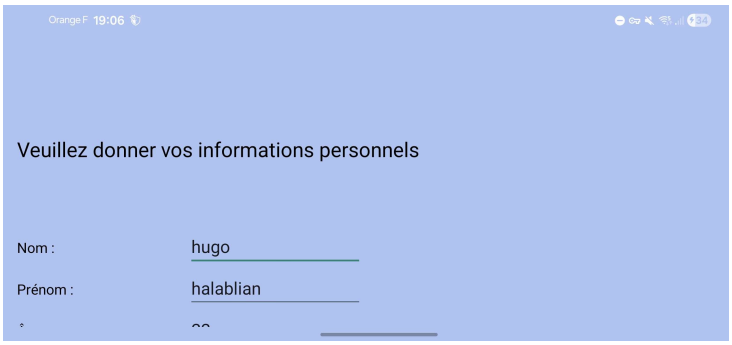

Fig. 6. – Preuve de robustesse : conservation des données en mode paysage.

3.5 Technique : Transfert de Données Complexes (Exercice 6)

Pour passer les données de l'activité de Saisie à l'activité de Résumé, nous n'avons pas seulement passé des types primitifs. Pour maintenir une architecture propre, nous avons opté pour la sérialisation de l'objet Utilisateur en chaîne JSON via *GSON*, passée ensuite dans l'*Intent*. Cela permet de transférer des structures de données complexes sans implémenter l'interface verbeuse `Parcelable` manuellement.

```
// Sérialisation avant envoi
val userJson = Gson().toJson(currentUser)
val intent = Intent(this, SummaryActivity::class.java)
intent.putExtra("USER_DATA", userJson)
startActivity(intent)
```

Liste 1. – Utilisation de GSON pour passer des objets complexes entre Activités.

4 Exercice 8 : Application Trains (Simulation Backend)

Pour l'application d'horaires de trains, l'enjeu était de proposer une interface de consultation efficace sans véritable API *backend*.

4.1 UX : Saisie Assistée et Liste

L'utilisation d'un champ texte libre pour une gare est source d'erreur. Nous avons justifié l'utilisation d'un `AutoCompleteTextView` : cela guide l'utilisateur et garantit que la ville saisie existe dans notre « base de données ». Pour l'affichage des résultats, nous avons opté pour une liste simple mais lisible, utilisant des `CardView` pour bien séparer chaque horaire, offrant un contraste visuel fort (Noir/Gris) adapté à une lecture rapide.

Fig. 7. – Détail UX : L'Auto-complétion prévient les erreurs de saisie de l'utilisateur.

4.2 Architecture des Données

L'utilisation de la librairie *GSON* nous a permis de mapper directement les fichiers JSON présents dans les `assets` vers des objets Kotlin. Cela simule parfaitement un appel réseau (qui renverrait du JSON) et rend le code prêt pour une future connexion API.

5 Exercice 9 : Agenda et Persistance Avancée

C'est l'application phare du projet, répondant aux exigences de l'exercice 9 sur la gestion d'événements.

5.1 Composants Avancés : *MaterialCalendarView & TimePicker*

Le composant natif `CalendarView` est limité visuellement. Pour offrir un *feedback* immédiat à l'utilisateur (pastilles de couleur sur les dates), nous avons intégré la librairie `MaterialCalendarView`.

Concernant la saisie de l'heure, nous avons fait le choix du `TimePickerDialog` natif d'Android (format horloge). Ce choix est justifié par l'habitude des utilisateurs Android : ils reconnaissent immédiatement ce composant et savent l'utiliser intuitivement, contrairement à des sélecteurs personnalisés « roue » (type iOS) qui seraient moins précis sur Android.

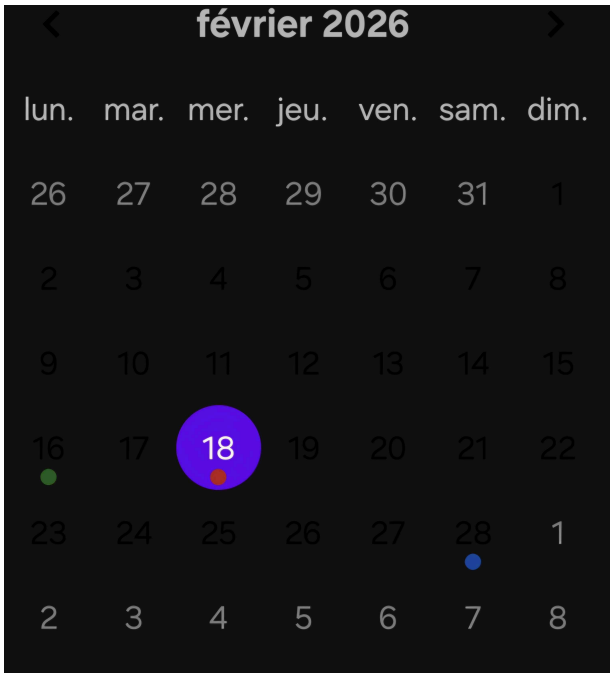

Fig. 8. – Indicateurs visuels (*Decorators*) : Une information immédiate sans ouvrir la date.

5.2 Persistance (JSON & *SharedPreferences*)

Pour conserver les événements, nous avons implémenté une persistance légère. 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.

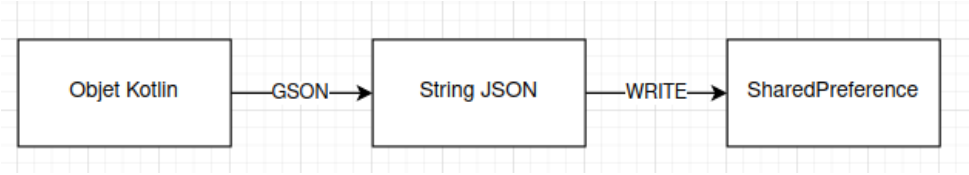

Fig. 9. – Flux de persistance : Objet Kotlin JSON string SharedPreferences.

6 Analyse Critique et Perspectives

Ce projet respecte les consignes fonctionnelles, mais peut être amélioré sur plusieurs axes architecturaux.

1. **Vers l'architecture *MVVM* :** Bien que ce TP soit réalisé en *MVC*, nous avons veillé à **découpler** la logique de données (*Parsing GSON*, Gestion des *SharedPreferences*) du code des Activités. Cette séparation prépare le terrain pour une migration vers *MVVM* (*Model-View-ViewModel*) : la logique de données actuelle serait déplacée dans des classes `Repository`, et les Activités ne s'occuperaient plus que de l'affichage, s'abonnant aux données via des `LiveData`.
2. **Limites du JSON Local :** Pour l'application « Trains », les données sont statiques. L'utilisation de *Retrofit* pour consommer une API REST (SNCF *Open Data*) serait l'étape suivante logique.
3. **Performance de la Persistance :** Sérialiser toute la liste d'événements en JSON à chaque ajout devient coûteux en performances. L'intégration de *Room (SQLite)* serait la solution technique appropriée pour un passage à l'échelle.

7 Bibliographie & Dépendances

Les bibliothèques suivantes ont été intégrées via *Gradle* pour enrichir les fonctionnalités natives :

- **Google Gson (v2.10.1)** : Sérialisation/Désérialisation JSON.
  - *Source : github.com/google/gson*
- **Material-CalendarView (v2.0.1)** : Composant de calendrier avancé permettant les *decorators*.
  - *Source : github.com/prolificinteractive/material-calendarview*
- **AndroidX Core & AppCompat** : Rétro-compatibilité et composants *Material Design*.