



Rapport du TP1 de l'UE HAI811I - Programmation Mobile

Thibaud PAULIN

Faculté des Sciences
Université de Montpellier

16 February 2026

Table des matières

1	Exercice 3 - Informations personnelles	3
1.1	Introduction	3
1.2	Architecture	3
1.3	Fonctionnalités	4
1.3.1	Interface simple du formulaire	4
1.3.2	Internationalisation des interfaces	6
1.3.3	Événements associés aux objets graphiques d'une vue et intents explicites	6
1.3.4	Intent implicite	7
2	Exercice 8 - Consultation d'horaires de trains (TrainApp)	8
2.1	Introduction	8
2.2	Architecture	8
2.2.1	Le package model	9
2.2.2	Le package view	9
2.2.3	Le package.viewmodel	9
2.3	Fonctionnalités	9
2.3.1	Utilisation de l'IA pour TrainApp	13
3	Exercice 9 - Agenda et ajout d'événements	13
3.1	Introduction	13
3.2	Architecture	14
3.2.1	Model	14
3.2.2	ViewModel	14
3.3	Fonctionnalités	14
3.3.1	Ajout d'événements et validation	15
3.3.2	Affichage en timeline	16
3.3.3	Suppression d'un événement	17
4	Conclusion	18

1 Exercice 3 - Informations personnelles

1.1 Introduction

Le but de cette application est d'implémenter un formulaire demandant à l'utilisateur de renseigner des informations personnelles. Deux versions de cette application ont été demandées, l'une utilisant des layouts xml et l'autre n'utilisant que Kotlin pour la réalisation de l'interface. Pour cet exercice mais aussi pour tous ceux qui suivront, j'ai décidé d'utiliser le langage Kotlin.

1.2 Architecture

Pour réaliser cette application, j'ai créé trois activités Kotlin : `Main.kt`, qui contient le formulaire avec les différents champs du formulaire, `Recap.kt`, qui résume les informations fournies, et `Profile.kt` qui affiche les informations de l'utilisateur si ces dernières ont été acceptées dans le récapitulatif et permet d'appeler le numéro renseigné. De plus, j'ai créé une data class `CountryItem.kt` qui sera utile pour la logique concernant les numéros de téléphones, que j'expliquerai plus tard. Voici une capture de l'arborescence de ce projet dans Android Studio :

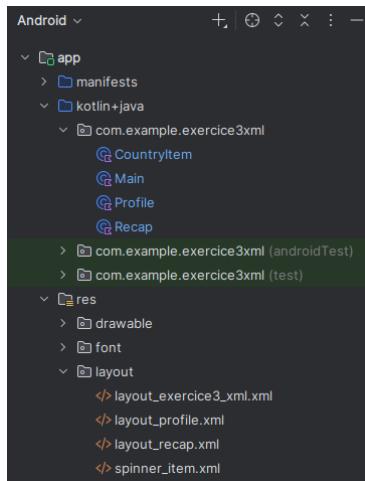


Fig. 1. – Arborescence des fichiers de l'application

Afin de représenter la navigation entre les différentes activités de l'application, voici un diagramme d'états détaillant comment l'on passe d'une activité à l'autre avec pour chaque transition l'intent utilisé, implicite ou explicite. De plus, si l'utilisateur se trouve dans `Recap.kt` mais qu'il veut revenir en arrière, j'ai préféré utiliser `finish()` qui permet de détruire l'instance de `Recap` qui ne nous intéresse plus, de passer dans un étage inférieur de la pile (l'instance de `Main` précédente) et ainsi économiser les ressources.

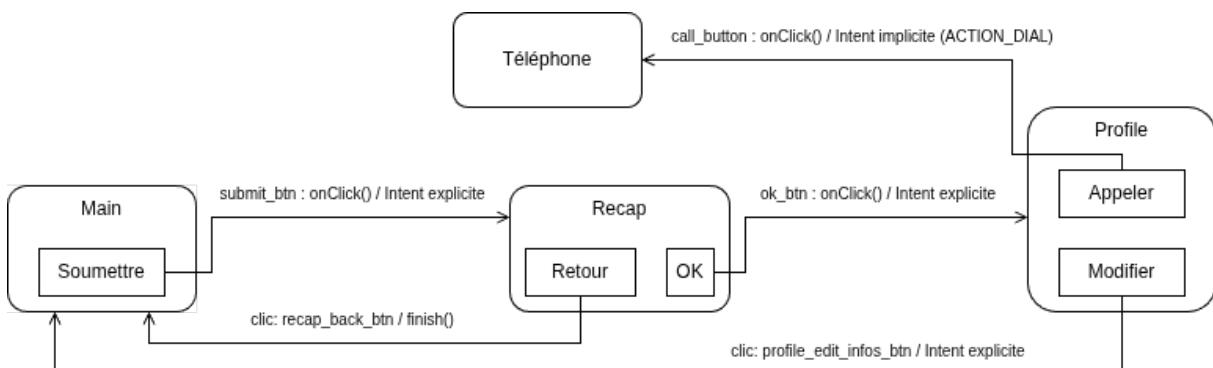


Fig. 2. – Diagramme d'états

1.3 Fonctionnalités

1.3.1 Interface simple du formulaire

Voici comment le formulaire vierge se présente lorsqu'on lance l'application.

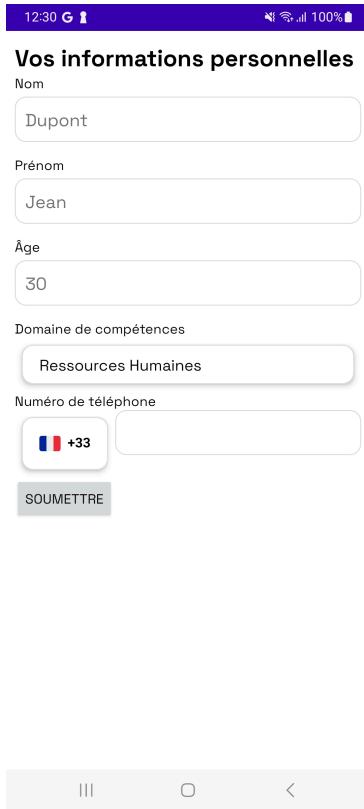


Fig. 3. – Formulaire vierge dans Main.kt

Tous les champs demandés dans l'énoncé du TP ont été inclus. Pour organiser les différentes vues, j'ai utilisé un `LinearLayout` avec une direction verticale. Pour les attributs Nom, Prénom, Âge et Numéro de téléphone, j'ai utilisé des vues `EditText` tout en me servant de la propriété `android:inputType` qui permet de restreindre l'utilisateur à n'utiliser le clavier que d'une certaine manière. Voici un exemple XML et Kotlin, comme demandé dans l'énoncé :

XML

```
<EditText  
    style="@style/RoundedEditText"  
    android:hint="@string/age_ph"  
    android:id="@+id/edit_text_age"  
    android:inputType="number"/>
```

Kotlin

```
val etAge =  
    createStyledEditText(getString(R.string.age_ph),  
        InputType.TYPE_CLASS_NUMBER)  
    rootLayout.addView(etAge)
```

Par ailleurs, je me suis servi de `TextView` pour créer des labels au dessus des champs et je leur ai renseigné un indice (`hint`). Dans cet exercice, il était demandé de réaliser l'interface à la fois en Kotlin et en XML. Pour passer d'une version à l'autre, j'ai commenté soit la version XML (utilisant `setContentViewR.layout. ...`) soit la version Kotlin.

Dans ce type de formulaire, les domaines de compétences sont souvent prédéfinis afin de faciliter le traitement des informations des utilisateurs. C'est la raison pour laquelle j'ai utilisé une vue `Spinner` qui permet de choisir son domaine parmi ceux prévus.

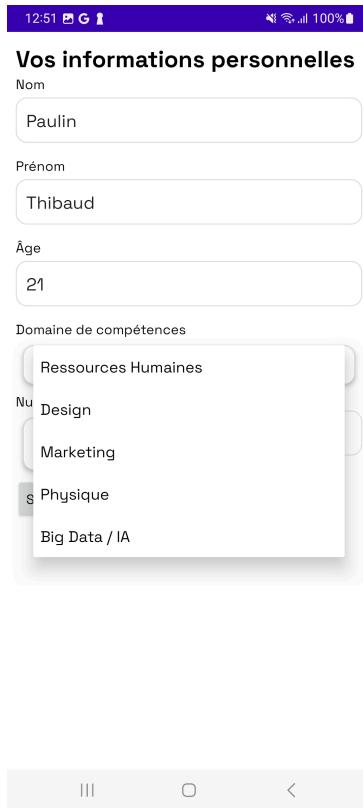
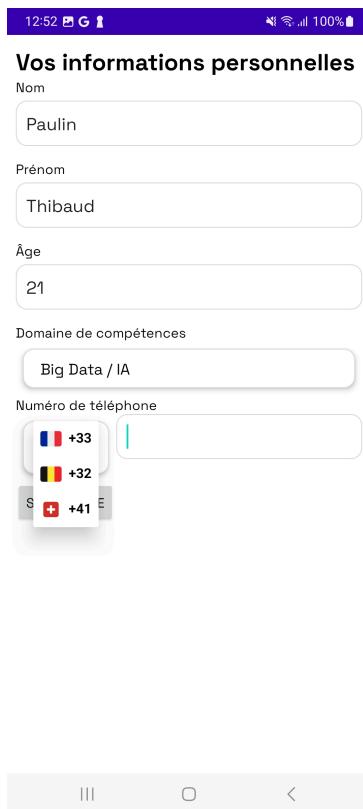


Fig. 4. – Spinner des domaines de compétences

Un autre Spinner est utilisé pour la sélection du pays du numéro de téléphone (+33 pour la France, +32 pour la Belgique et +41 pour la Suisse) le préfixe sélectionné sera concaténé au numéro renseigné pour passer l'appel.



1.3.2 Internationalisation des interfaces

Grâce à l'éditeur de traductions, il est possible, pour chaque valeur dans la ressource `string`, de fournir une valeur par défaut (en français ici) et une pour chaque pays, une langue. Dans ce cas je me suis concentré sur l'anglais. Voici l'activité une fois le téléphone passé en anglais :

15:11 100%
 Your personal data
Last Name
Doe
First Name
John
Age
30
Area of expertise
Big Data / IA
Phone number
+33
SUBMIT

Fig. 6. – Interface en anglais

1.3.3 Événements associés aux objets graphiques d'une vue et intents explicites

Enfin, au moment de soumettre le formulaire, les champs sont vérifiés pour qu'aucun d'entre eux ne soit vide.

12:52 100%
 Vos informations personnelles
Nom
Paulin
Prénom
Jean
Âge
21
Domaine de compétences
Big Data / IA
Numéro de téléphone
+33 783918884
SOUMETTRE
Un ou plusieurs champs sont vides. Veuillez les remplir

Fig. 7. – Coloration en rouge des champs manquants

12:52 100%
 Vos informations personnelles
Nom
Paulin
Prénom
Thibaud
Âge
21
Domaine de compétences
Big Data / IA
Numéro de téléphone
+33 783918884
SOUMETTRE
Etes-vous sûr de vouloir envoyer le formulaire ?
NON OUI, ENVOYER

Fig. 8. – Demande d'envoi du formulaire si les champs sont valides

Si l'on accepte d'envoyer le formulaire, un intent explicite est réalisé. Voici comment il se présente :

```

builder.setPositiveButton(getString(R.string.validate_dialog)) { _, _ ->
    val intent = Intent(this, Recap::class.java)
    intent.putExtra("USER_NAME", etName.text.toString())
    intent.putExtra("USER_FIRSTNAME", etFirstname.text.toString())
    intent.putExtra("USER_AGE", etAge.text.toString())
    intent.putExtra("USER_EXPERTISE", areaOfExpertise)
    intent.putExtra("USER_PHONE", fullPhoneNumber)
    startActivity(intent)
}

```

On désigne la classe `Recap` comme destination de l'intent en lui donnant des paramètres grâce à la méthode `putExtra`. J'ai de la même manière créé les intents pour passer de `Recap.kt` vers `Profile.kt` pour consulter les informations et celui de `Profile.kt` vers `Main.kt` pour l'envoi d'un nouveau formulaire. Comme mentionné plus haut, je me suis servi de la méthode `finish()` pour dépiler la pile d'activités, au lieu de construire une nouvelle instance de la classe `Main` quand on vient de `Recap`. Voici donc les trois activités côté-à-côte :



Fig. 9. – `Main.kt` : Les champs sont pré-remplis

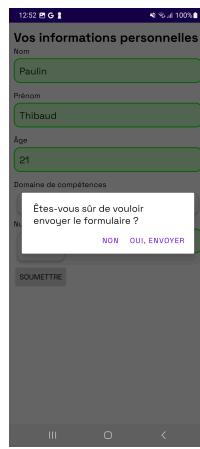


Fig. 10. – `Recap.kt` : Récapitulatif des informations

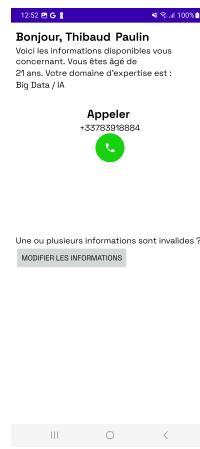


Fig. 11. – `Profile.kt` : Affichage du profil

1.3.4 Intent implicite

Même s'il était simplement demandé de pouvoir appeler le numéro de téléphone dans la dernière activité de l'application, j'ai rajouté la possibilité de revenir au point de départ pour refaire le formulaire si un ou plusieurs informations sont incorrectes. J'ai par ailleurs rajouté un descriptif sous forme de phrase qui résume les informations renseignées, en anglais et en français, plutôt qu'une simple liste des attributs comme dans `Recap.kt`. Enfin, lorsque l'on appuie sur la vue `ImageButton` vert, représenté par un symbole de téléphone, l'application téléphone se lance avec le bon numéro de téléphone et son préfixe (+33) :

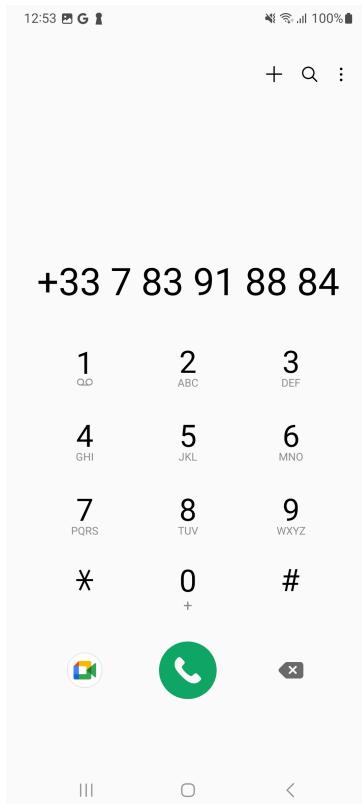


Fig. 12. – Ouverture de l’application téléphone avec le numéro donné

2 Exercice 8 - Consultation d’horaires de trains (TrainApp)

2.1 Introduction

Pour cette application, il nous était demandé de réaliser une interface dans laquelle l’utilisateur peut choisir un départ et une arrivée, pour se voir proposer les horaires de trains. Pour ce faire, j’ai décidé d’utiliser l’API de la SNCF, basée sur Navitia, dont la documentation est disponible ici : <https://doc.navitia.io/#getting-started>.

2.2 Architecture

Afin de prendre connaissances des bonnes pratiques de programmation sur mobile, je me suis renseigné sur l’architecture MVVM (Model-View-ViewModel) et l’ai mise en place pour ce projet. Voici l’architecture de l’application TrainApp :

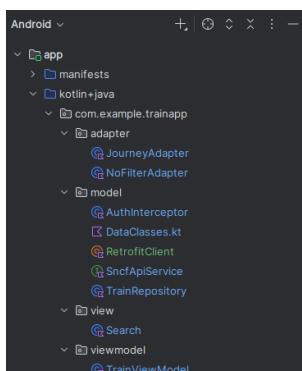


Fig. 13. – Architecture de TrainApp - fichiers

Kotlin

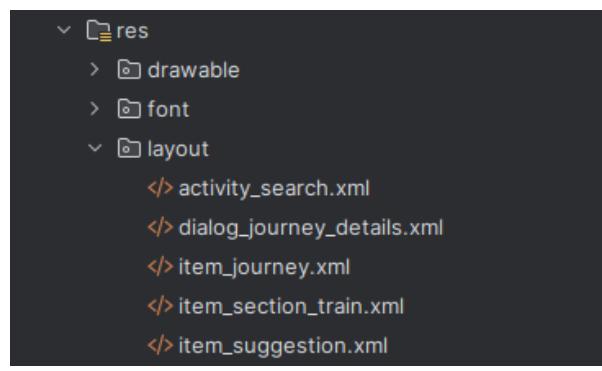


Fig. 14. – Architecture de TrainApp - layouts

2.2.1 Le package model

Dans l'architecture MVVM, il regroupe toutes les données qui seront à traiter pour les montrer à l'utilisateur.

`Sncf ApiService` accède aux deux endpoints de l'API de la SNCF dont j'aurai besoin pour obtenir un lieu à partir d'un texte et obtenir les itinéraires (ou « journeys » dans l'API).

`DataClasses.kt` contient les classes qui seront manipulées pour la création d'itinéraires (`Places`, `Section`, `Journey`, etc.) provenant de l'API.

Dans `TrainRepository.kt`, j'ai écrit les fonctions `searchPlaces` et `getJourneys` permettant respectivement d'obtenir tous les lieux correspondant à un nom et tous les itinéraires entre deux lieux, utilisant d'un côté les endpoints dans `Sncf ApiService` et de l'autre les objets définis dans `DataClasses.kt`. `AuthInterceptor` permet au moment où une requête est formulée, de lui insérer la clé API. En effet, l'authentification pour les requêtes à l'API SNCF fonctionne en « Basic Auth » où le nom d'utilisateur est la clé API et le mot de passe est vide.

Enfin, le fichier `RetrofitClient` permet de fournir une URL de base qui peut donc être facilement modifiée. Il construit la requête http voulue grâce à l'intercepteur défini plus haut.

Grâce à cette structure, le projet est plus modulable et utilise le principe de la séparation des rôles. La récupération, le traitement et l'affichage des données sont réalisés dans des fichiers bien distincts.

2.2.2 Le package view

Ce package va contenir l'activité principale qui affiche les informations déjà traitées et envoie les données renseignées par l'utilisateur au ViewModel. D'ailleurs, afin de rendre l'expérience de navigation plus simple, j'ai pris la décision de ne faire qu'une seule activité `Search.kt` regroupant la recherche et l'affichage des itinéraires. Dès que les informations minimales pour demander les itinéraires seront données, la requête sera automatiquement envoyée.

2.2.3 Le package.viewmodel

C'est dans ce package que le traitement des données sera réalisé. Dans `TrainViewModel`, Les informations renseignées (Villes, date et heure) dans la partie View seront récupérées et envoyées au Model. Quand ce dernier lui répondra avec les informations qu'il souhaite, le ViewModel va opérer les différents traitements (tri, filtrage, etc.) pour les envoyer à la View.

2.3 Fonctionnalités

Voici comment se présente l'application. Il y a deux barres de recherche, une pour le départ et l'autre pour la destination.

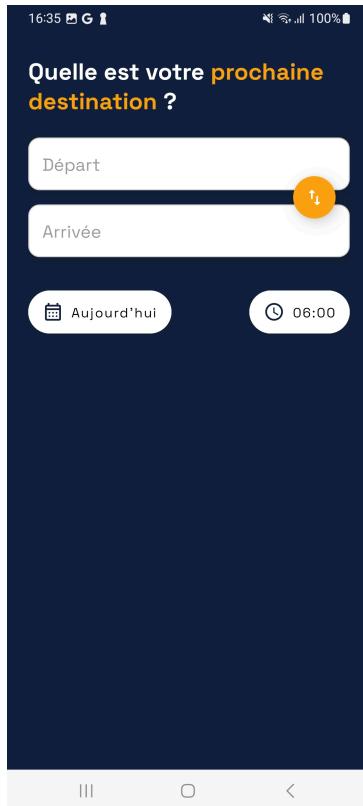


Fig. 15. – Ecran d'accueil de TrainApp

Jusque là, nous n'avions vu que les vues `EditText` pour un champ de texte. Cependant, l'objectif ici est d'obtenir une liste de suggestions de gares. C'est la raison pour laquelle j'ai choisi d'utiliser un `AutoCompleteTextView`, qui gère nativement une liste déroulante, basée sur une liste de valeurs qu'il faudra lui fournir. Il permet à l'utilisateur, même s'il a fait une faute de frappe, de sélectionner une valeur existante dans la base de données de la SNCF.

En utilisant un `Adapter` personnalisé (`NoFilterAdapter.kt` dans l'arborescence), j'ai pu lier les résultats de l'API à l'interface utilisateur. Il fallait un `Adapter` personnalisé et non un `ArrayAdapter` de base car ce dernier impose un filtrage qui n'affiche que les chaînes de caractères qui ne commencent que par ce que l'utilisateur écrit. En redéfinissant la méthode `getFilter()` pour donner tous les résultats de l'API sans filtre, j'ai empêché qu'il masque par exemple la gare « Paris Gare de Lyon » si l'utilisateur écrit « Lyon ».

Cet adaptateur est essentiel car il ne traite pas que des noms, mais des objets `Place`. Chaque objet `Place` contient le nom de la gare pour l'affichage, mais surtout un identifiant unique. C'est cet identifiant qui est indispensable pour effectuer les requêtes de recherche d'itinéraires (`Journeys`), car le nom seul ne suffit pas à l'API pour localiser précisément une station. Lorsqu'un utilisateur clique sur une suggestion, l'adaptateur permet de récupérer l'objet `Place` correspondant à la position cliquée pour extraire l'ID.

Pour envoyer dynamiquement les villes et récupérer les suggestions de l'API, j'ai utilisé la coroutine Kotlin `lifecycleScope.launch` qui permet de lancer dans un autre thread l'appel à l'API, qui est asynchrone. Grâce à la méthode `collect`, le thread écoute le flux de suggestions du `ViewModel` et met à jour dynamiquement la liste d'`item_suggestion`.

Voici ce qui se passe quand l'utilisateur écrit du texte :

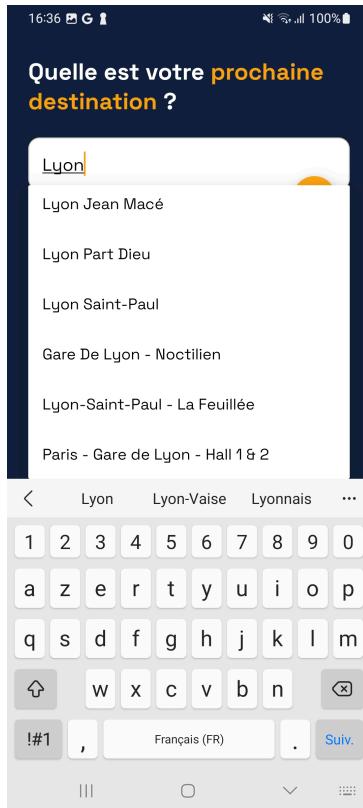


Fig. 16. – Suggestion de lieux dans la barre de recherche

Dès que les deux champs de départ et arrivée sont remplis, une requête s'envoie automatiquement, sans action supplémentaire de l'utilisateur pour obtenir les trajets entre ces deux lieux. De plus, j'ai ajouté un bouton permettant d'intervertir ces deux lieux, pouvant devenir soit le départ, soit l'arrivée.



Fig. 17. – Architecture de TrainApp - fichiers Kotlin

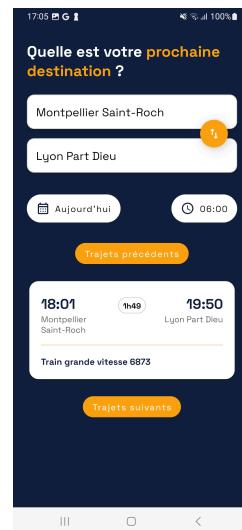


Fig. 18. – Architecture de TrainApp - layouts

Les résultats obtenus sont stockés et mis en forme dans le `JourneyAdapter`. Dès que l'un des lieux change, la méthode `clearHistory()` du `ViewModel` est appelée pour réinitialiser la liste et vider le contenu de la `RecyclerView`, un composant que j'utilise car il permet de générer autant de `journey_item` que nécessaire, sans avoir à prévoir leur nombre.

Afin de pouvoir raffiner la sélection, il est possible de choisir la date et l'heure du trajet. Par défaut, la date est aujourd'hui et l'heure est 06:00 pour être sûr que l'utilisateur ne rate pas le prochain train de la journée. Pour créer ces boutons, je me suis respectivement servi de `calendar.getInstance`, et de `MaterialTimePicker` :

```
val calendar = Calendar
    .getInstance(TimeZone.getTimeZone("UTC"))
    calendar.timeInMillis = today
    calendar.add(Calendar.DAY_OF_MONTH,
23) // 23 jours de prévision de l'API
disponibles uniquement
```

```
val picker = MaterialTimePicker.Builder()
    .setTimeFormat(TimeFormat.CLOCK_24H)
    .setHour(viewModel.selectedHour)
    .setMinute(viewModel.selectedMinute)
    .setTitleText("Heure de
départ")
    .build()
```

De la même manière que pour les changements de lieux, chaque modification de la date ou de l'heure déclenche instantanément une nouvelle requête. L'historique des trajets stockés est vidé afin de se renouveler avec les nouvelles données.

Chaque `journey_item` contient toutes les informations du voyage : Heure et lieu du départ et de l'arrivée, durée du trajet, numéro du train et le nombre de correspondances. Toutes ces informations sont disponibles dans le retour de l'API SNCF dans la liste des itinéraires reçue. Si l'on veut connaître les détails d'un trajet avec une ou plusieurs correspondances, il suffit de cliquer dessus, un `AlertDialog` apparaîtra avec toutes les informations qui se ferme en cliquant autre part sur l'écran.



Fig. 19. – Affichage d'un trajet avec une correspondance

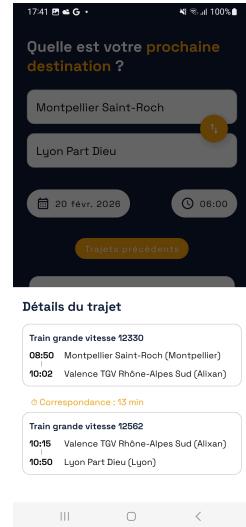


Fig. 20. – Détails de la correspondance d'un trajet

Enfin, pour ne pas surcharger l'API, la SNCF ne donne que 2 ou 3 trajets correspondants à la date et l'heure demandée. Dans le corps de la réponse, il existe cependant des liens (attribut nommé « `links` » dans le JSON de retour) appelés « `next` » et « `pred` » qui contiennent chacun l'URL exacte pour obtenir les trajets juste après ou juste avant. C'est de ces deux structures de données dont je me suis servi pour ajouter les boutons « `Trajets suivants` » et « `Trajets précédents` » qui ne font rien d'autre qu'appeler ces URLs. Il aura tout de même fallu gérer le cas limite où le train qui vient après est programmé au lendemain, auquel cas, comme sur l'application SNCF Connect, une `AlertDialog` apparaît pour prévenir du changement de date. L'utilisateur peut soit décliner, soit se rendre sur la page du jour suivant ou précédent.



Fig. 21. – Alerte en cas de changement de jour après appui sur « Trajets suivants »

2.3.1 Utilisation de l'IA pour TrainApp

Pour le développement de la partie « TrainApp », j'ai utilisé Gemini sur trois aspects.

Dans un premier temps, l'agencement des `item_journey` (Fig. 19) utilise une vue de `MaterialCardView` intégrant plusieurs niveaux de `LinearLayout` et de `TextView` pour afficher les horaires, les stations et les correspondances. Gemini m'a également aidé à comprendre l'utilisation de `drawables` et comment les composer et les utiliser pour personnaliser des vues plus élaborées.

Ensuite, comme je ne connaissais pas le pattern MVVM, j'ai associé mes recherches sur le web avec des questions d'un point de vue structurel à Gemini pour être sûr de bien comprendre ce nouveau patron de conception et m'assurer qu'aucun fichier ne fasse quelque chose dont il n'est pas responsable.

Enfin, le retour de l'API pour les itinéraires est extrêmement volumineux, contenant de nombreuses sections. L'IA m'a expliqué la structure et les différentes parties intéressantes pour mon application, notamment pour les différentes sections du trajet et les correspondances. Par ailleurs, elle m'a aussi aidé à générer des expressions régulières pour filtrer les données reçues, par exemple pour ôter le code postal dans : « Montpellier (34000 - 34090) ». Utiliser Gemini pour ces parties plus fastidieuses m'a permis de plus me concentrer sur la partie logique de l'application et m'a, en ce sens, fait gagner du temps.

3 Exercice 9 - Agenda et ajout d'événements

3.1 Introduction

L'objectif de ce dernier exercice était de concevoir une application d'agenda permettant d'associer des événements à des dates et de les consulter. J'ai renouvelé l'utilisation du pattern MVVM, des coroutines et je me suis servi du cache, géré par les `SharedPreferences`. En effet, afin de pouvoir réaliser un

stockage des événements et de pouvoir les consulter même après avoir redémarré l’application, j’ai décidé d’utiliser le cache du téléphone pour assurer cette fonctionnalité.

3.2 Architecture

Comme pour l’exercice précédent, j’ai séparé le code en trois packages distincts : `model`, `view` et `viewmodel`.

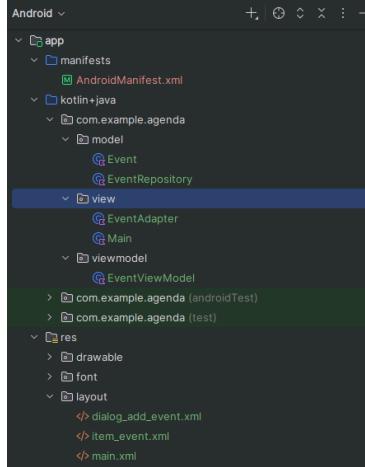


Fig. 22. – Arborescence du projet Agenda

3.2.1 Model

Contrairement à l’application TrainApp qui appelait une API externe, l’Agenda doit stocker ses données localement. J’ai créé la classe `EventRepository` qui utilise les `SharedPreferences` pour sauvegarder la liste des événements. Afin de stocker la liste des événements dans un fichier XML, j’ai utilisé la librairie `Gson` pour sérialiser les objets en JSON.

```
fun saveEvents(events: List<Event>) {
    val json = gson.toJson(events)
    prefs.edit { putString("events_list", json) }
}
```

3.2.2 ViewModel

C’est dans `EventViewModel` que la préparation des données à afficher est effectuée. J’ai utilisé l’opérateur `combine` des `StateFlow` qui permet de fusionner la liste complète des événements (`_allEvents`) et la date actuellement sélectionnée dans le calendrier (`_selectedDate`). Dès que l’utilisateur clique sur une date ou ajoute un événement, la liste est automatiquement filtrée et triée pour n’afficher que les événements de la journée choisie.

3.3 Fonctionnalités

Voici comment se présente l’application au lancement. En utilisant la vue `<CalendarView>` en XML, j’ai pu ajouter en haut de l’activité un calendrier où l’on peut sélectionner la date voulue. Par défaut, l’application affiche une `TextView` indiquant qu’aucun événement n’est prévu.



Fig. 23. – Accueil de l'Agenda sans événement

3.3.1 Ajout d'événements et validation

Pour ajouter un événement, il faut d'abord sélectionner la date de l'événement dans le calendrier. Ensuite, j'ai créé un `FloatingActionButton` pour déclencher l'ajout d'un événement. Cette vue permet de toujours rester fixée au même endroit de l'écran pour pouvoir ajouter même si on défile tous les événements vers le bas.

L'ajout se fait via une vue `AlertDialog`. Il est possible de choisir le titre, dire si l'événement va durer toute la journée, choisir l'heure si ce n'est pas le cas et renseigner une description facultative. Au moment de la validation, une vérification est faite pour s'assurer que le titre n'est pas vide. L'événement est ensuite créé avec un `UUID` et envoyé au `ViewModel`. De plus, si l'option «Toute la journée» est cochée, il est impossible de définir une heure.



Fig. 24. – Création d'un événement



Fig. 25. – Choix de l'heure



Fig. 26. – Si un événement dure toute la journée, impossible de choisir une heure

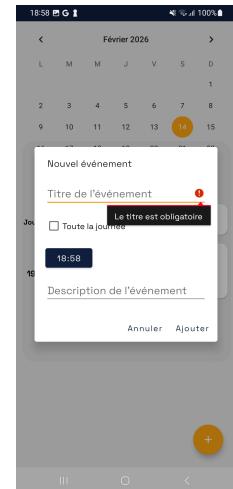


Fig. 27. – Le titre de l'événement ne peut pas être vide

3.3.2 Affichage en timeline

De la même manière que pour l'affichage des itinéraires de la SNCF, pour gérer un affichage de différentes vues en colonne dont on ne connaît le nombre à l'avance, j'ai réutilisé une RecyclerView, gérée par l'EventAdapter pour mettre à jour dynamiquement la liste des événements.

L'affichage suit un ordre logique géré directement dans le EventViewModel : grâce à l'opérateur combine, les données sont filtrées pour le jour sélectionné, puis triées pour que les événements « Toute la journée » apparaissent en tête de liste, suivis des autres par ordre chronologique. Voici comment la logique a été implémentée (isSameDay est une méthode qui indique si deux timestamps correspondent à la même journée et isAllDay est vrai si l'option « Toute la journée » a été cochée à la création) :

```
val events: StateFlow<List<Event>> = combine(_allEvents, _selectedDate) { list,
dateTimestamp ->
    list.filter { event ->
        isSameDay(event.timestamp, dateTimestamp)
    }.sortedWith(
        compareByDescending<Event> { it.isAllDay }
            .thenBy { it.timestamp }
    )
}
```

Chaque item a à sa gauche l'heure de l'événement et à droite le titre avec ou non une description. De plus, le fait d'encapsuler la RecyclerView dans un FrameLayout permet d'avoir une zone fixe de défilement, pour toujours avoir en vue le calendrier plus haut et pouvoir rapidement changer de date. Enfin, si l'on choisit un autre jour, la RecyclerView s'actualise et affiche uniquement les événements du jour sélectionné



Fig. 28. – Affichage en timeline des événements



Fig. 29. – Défilement dans un FrameLayout



Fig. 30. – La RecyclerView s'adapte au jour choisi

3.3.3 Suppression d'un événement

Pour supprimer un événement, j'ai implémenté un `OnLongClickListener` sur les éléments de la liste. J'ai trouvé cette solution plus agréable visuellement qu'un petit icône de corbeille répété sur chacun d'entre eux. Une alerte de confirmation demande validation avant de supprimer définitivement l'objet via le repository, qui gère l'enregistrement et le chargement des données depuis le cache (`SharedPreferences`).



Fig. 31. – Suppression d'un événement

4 Conclusion

Pour conclure, ces premières applications Android m'ont permis de découvrir plusieurs aspects majeurs du développement mobile. J'ai vraiment pu comprendre comment utiliser des layouts, que l'on soit en XML ou en Kotlin mais surtout la possibilité qui nous est laissée de choisir l'un ou l'autre.

De plus, j'ai pu apprécier comment passer d'une activité à une autre via les intents, qu'ils soient explicites ou implicites ou encore via la méthode `finish()` et la facilité de pouvoir manipuler le contexte d'une activité, et de rediriger vers une autre page avec des paramètres (`putExtra`) pour justement garder un contexte d'exécution.

Enfin, ce premier TP m'aura initié au pattern MVVM pour structurer proprement des applications qui manipulent des types précis de données, pouvant provenir de différentes sources. La distinction entre les différents packages permet également de bien comprendre la séparation des responsabilités des fichiers du projet.