HAI811I – Programmation Mobile

RAPPORT TP1

SALHI Nina
22115492

Encadrant : Seriai Abdelhak Djamel

Master 1 Informatique - IASD

Université de Montpellier
Département d'Informatique

Table des figures

1 Introduction

Ce TP1 vise à nous introduire au développement d'applications Android avec la découverte de la structure de projet, la réalisation d'interfaces utilisateur et la gestion de l'internationalisation. Il permet de se former sur les fondamentaux d'Android Studio et de réaliser une première application de smartphone de manière très simple.

2 Hello world

Notre objectif pour ce premier exercicie est de créer une première application Android affichant "Hello World" et explorer sa structure. Pour ce faire, nous avons utilisé Android Studio pour générer automatiquement un projet Android contenant une activité par défaut, dans notre cas on a choisi "Empty activity". Ensuite, nous avons examiné les différents fichiers constituant ce projet :

2.1 Organisation des fichiers du projet

L'organisation des fichiers d'un projet Android suit une structure bien définie. Voici les principaux dossiers et leur utilité :

- **Dossier** `Manifests` : Contient le fichier `AndroidManifest.xml`, qui est essentiel à la configuration de l'application. Ce fichier déclare les composants de l'application (activités, services, etc.) ainsi que les permissions requises pour son bon fonctionnement.
- **Dossier** `Java + Kotlin` : Contient tous les fichiers sources de l'application. Dans notre cas, nous avons choisi de travailler avec Kotlin. Le fichier `MainActivity.kt` y est présent et contient la logique principale de l'application.
- **Dossier** `res` **(ressources)** : Ce dossier regroupe tous les fichiers nécessaires à l'interface graphique et au design de l'application. Il est subdivisé en plusieurs sous-dossiers :
  - `drawable/` : Contient les images et icônes utilisées dans l'application (ex. logos, boutons personnalisés, illustrations).
  - `layout/` : Stocke les fichiers XML définissant la structure visuelle de chaque écran ou interface utilisateur.
  - `values/` : Regroupe des fichiers XML contenant des valeurs réutilisables :
    - `strings.xml` : Définit les textes de l'application pour en faciliter la gestion et la traduction.
    - `colors.xml` : Contient les couleurs utilisées dans l'application.
- **Dossier** `Gradle Scripts` : Utilisé par Android Studio pour générer les fichiers nécessaires à la compilation de l'application et produire l'APK final.

Une fois l'application exécutée, nous avons pu afficher le texte "Hello World" à l'écran.

3 Une première application- Interface simple

Dans cette exercice nous devons développer une application type formulaire permettant à l'utilisateur de saisir des informations personnelles (nom, prénom, âge, domaine de compétences, numéro de téléphone) et de les valider via un bouton. On a 2 façons de faire :

3.1 La vue (interface) en XML

Dans cette approche, l'ensemble de l'interface utilisateur est défini dans un fichier XML. Ce fichier contient la structure de l'interface avec les éléments suivants :

- **Un `LinearLayout` vertical** : Permet d'organiser les éléments les uns en dessous des autres.
- **Un `TextView`** : Placé avant chaque champ pour afficher un libellé (ex. « Nom », « Prénom », etc.).
- **Un `EditText`** : Permet à l'utilisateur de saisir des informations.
- **Un `Button`** : Utilisé pour soumettre les données.

Dans le fichier Kotlin (`MainActivity.kt`), nous avons procédé comme suit :

- Récupération des valeurs saisies dans les champs `EditText` à l'aide de leurs références (`findViewById`) et ajout d'un écouteur d'événements sur le bouton pour afficher ou traiter les informations saisies.

Cette approche permet une séparation claire entre l'interface (**XML**) et la logique (**Kotlin**), ce qui facilite la maintenance et l'évolutivité du code.

3.2 la vue (interface) dans le code kotlin

L'autre approche consiste à générer dynamiquement l'interface utilisateur directement dans le code Kotlin, sans utiliser de fichier XML.

- Création d'un `LinearLayout` vertical dans le code.
- Instanciation manuelle des composants `TextView`, `EditText` et `Button`, en leur attribuant leurs propriétés (`hint`, `textSize`, etc.).
- Ajout de ces éléments au `LinearLayout` et définition de ce dernier comme contenu principal de l'activité.

De la même manière que dans la version XML, nous avons utilisé les identifiants pour récupérer les saisies et implémenté un écouteur sur le bouto et ici, toute l'interface est créée à la volée, ce qui peut être utile si l'interface doit être modifiée dynamiquement en fonction du contexte.

3.3 Conclusion sur les deux options

L'approche XML est la plus optimale pour plusieurs raisons :

- Elle sépare clairement l'interface de la logique métier, facilitant ainsi la maintenance et la réutilisation du code.

— Toute modification apportée au layout XML est automatiquement appliquée à toutes les activités qui l'utilisent, contrairement à la version Kotlin où chaque activité doit être modifiée individuellement.

— Android optimise mieux les layouts XML, ce qui permet d'améliorer les performances de l'application.

Ainsi, pour un développement professionnel et évolutif, l'utilisation des interfaces en XML est fortement recommandée.

3.4 Ajout de labels

Afin d'améliorer la lisibilité, nous avons ajouté un texte descriptif avant chaque champ, en indiquant par exemple "Entrez votre nom" ou "Entrez votre numéro de téléphone". Cela permet d'accompagner l'utilisateur dans la saisie des informations, pour ce faire on a :

3.4.1 Si l'interface est créée en XML

— Ajout manuel des labels (`TextView`) avant chaque champ `EditText` dans le fichier XML.

— Chaque champ est précédé d'un texte descriptif tel que « Entrez votre nom », « Entrez votre âge », etc.

— Cette approche assure une organisation claire et facile à comprendre pour tout développeur lisant le fichier XML.

3.4.2 Si l'interface est créée dynamiquement en Kotlin

— Automatisation avec une fonction `addLabeledField()` pour éviter la répétition du code.

— Cette fonction prend en paramètre :

  — Le texte du label (`TextView`) qui s'affiche avant le champ de saisie.

  — Le texte d'indication (`hint`) qui s'affiche à l'intérieur du champ (`EditText`).

  — Le type de saisie (`inputType`) pour garantir un format adapté à la donnée saisie (ex. : `number` pour l'âge, `phone` pour le numéro de téléphone).

— Ensuite, la fonction ajoute dynamiquement chaque champ et son label à l'interface utilisateur.

FIGURE 1 – Formulaire exericice 3

3.5 Remarque

Pour personnaliser la présentation de l'application et utiliser un fond de couleur rose, nous avons amendé le fichier colors.xml contenu dans la sous-racine res/values/. Cet fichier détient la couleur qui sera utilisée dans la présentation de l'interface et permet de la concentrer et de la modifier aisément sans avoir à la modifier un par un dans un composant après l'autre.

4 Internationalisation des interfaces

Dans cet exercice, on va essayer d'adapter l'application pour qu'elle puisse être affichée en français et en anglais en fonction de la langue par défaut du téléphone. L'objectif est d'éviter d'avoir des textes codés en dur dans l'application et de centraliser toutes les traductions dans des fichiers de ressources.

4.1 Utilisation de fichiers `strings.xml` dans la gestion de textes

Dans Android Studio, nous avons importé le fichier `strings.xml` situé dans `res/values/`. Nous y avons ajouté tous les libellés utilisés dans l'application, comme « Nom », « Prénom », « Âge », etc. Ensuite, nous avons utilisé l'outil de traduction intégré pour générer la version anglaise.

4.2 Création du fichier de traduction anglais (`strings.xml`)

— Dans l'éditeur de traduction, nous avons sélectionné l'option permettant d'ajouter une nouvelle langue.

— Nous avons choisi l'anglais, et Android Studio a automatiquement créé un fichier `strings.xml` dans `res/values-en/`.

— Nous avons ensuite traduit tous les libellés dans ce fichier.

4.3 Création du fichier `strings.xml` pour le français

Par défaut, Android utilise l'anglais comme langue principale. Nous avons donc ajouté un fichier `res/values-fr/strings.xml` contenant les mêmes clés, mais avec des textes en français. Ainsi, si l'appareil est configuré en français, Android utilisera automatiquement ce fichier.

4.4 Remplacement des textes en dur dans l'interface

Dans nos fichiers XML (`layout.xml`), nous avons remplacé les textes statiques (par exemple : « Nom ») par une référence aux fichiers `strings.xml` en utilisant `@string/nom`. Ainsi, lors de l'exécution, Android charge automatiquement la version correspondant à la langue du téléphone.

4.5 Détection automatique de la langue

Android sélectionne automatiquement la langue correspondant à celle du téléphone.

— Si l'appareil est en français, l'application utilisera le fichier `strings.xml` situé dans `res/values-fr/`.

— Si l'appareil est en anglais ou dans une autre langue, l'application utilisera la version par défaut
(`res/values/strings.xml`).

4.6 Remarque

— Facilité de mise à jour et de maintenance : Toutes les chaînes de caractères sont centralisées dans les fichiers `strings.xml`, ce qui facilite l'ajout de nouvelles langues sans modifier le code. Si un texte doit être modifié ou corrigé, il suffit de le mettre à jour dans `strings.xml`, et la modification sera automatiquement prise en compte dans toute l'application.

— Optimisation pour les évolutions futures : Cette approche permet d'ajouter facilement de nouvelles langues en créant un dossier `values-lang_code/` et en y ajoutant un fichier `strings.xml` traduit.

Par exemple, si nous voulons ajouter l'espagnol (`values-es/`) ou l'allemand (`values-de/`), il suffira simplement d'ajouter les traductions dans ces nouveaux fichiers.

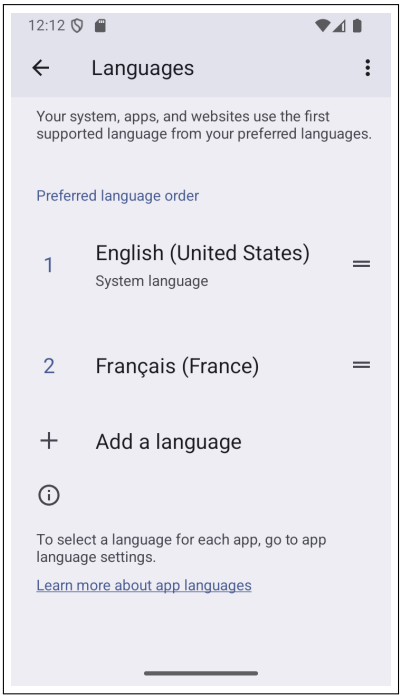

FIGURE 2 – Langue par default du Systeme est l'Anglais

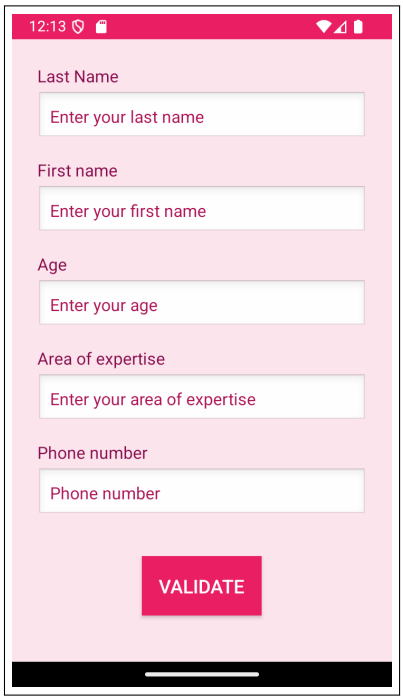

FIGURE 3 – Formulaire en Anglais

FIGURE 4 – Langue par default du Systeme est le Français

FIGURE 5 – Formulaire en Français

5 Événements associés aux objets graphiques d'une vue

Dans cet exemple, nous allons nous consacrer au traitement des événements utilisateur sur Android, avec un focus sur la manipulation de la prise de possession par un bouton de validation. L'objectif est de réaliser un système qui affiche un dialogue de confirmation lors de la validation de données et qui modifie la présentation de la saisie des champs automatiquement sur la base de la manipulation de l'utilisateur. Lorsqu'un utilisateur clique sur « Valider » :

— Une boîte de dialogue apparaît demandant la confirmation.

— Si l'utilisateur valide : les champs deviennent de couleur et la redirection vers une page différente est activée.

— Si l'utilisateur annule : aucun changement n'est appliqué.

La logique métier repose sur trois axes :

5.1 Gestion des événements (Event Handling)

Les interfaces utilisateur nécessitent des interactions. Nous allons donc utiliser `setOnClickListener` pour écouter l'événement déclenché lorsqu'un utilisateur appuie sur le bouton de validation. Exemple :

```
button.setOnClickListener {
    // Action déclenchée lorsque l'utilisateur clique sur le bouton
}
```

5.2 Affichage d'une boîte de dialogue de confirmation

Une boîte de dialogue (`AlertDialog`) interrompt l'action principale pour demander à l'utilisateur s'il souhaite confirmer ou annuler son action.

— « Oui » → Applique les modifications (ex : changement de couleur des champs).

— « Non » → Ferme simplement la boîte de dialogue.

5.3 Modification dynamique des éléments graphiques

Après validation, nous modifions les propriétés des champs de saisie, notamment leur couleur de fond, pour indiquer visuellement qu'ils sont validés. Exemple :

```
editNom.setBackgroundColor(Color.parseColor("#D1C4E9")) // Violet clair
```

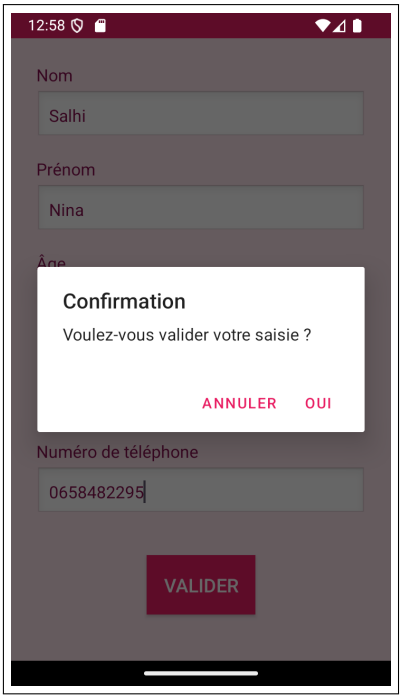

FIGURE 6 – Boite de dialogue de confirmations

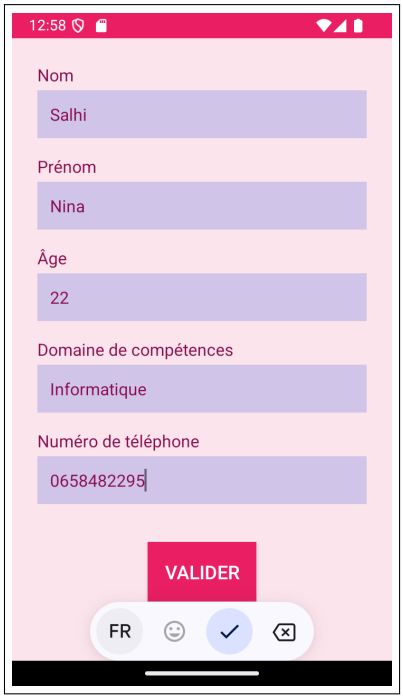

FIGURE 7 – Formulaire une fois cliquer sur OUI

6 Intent

Un Intent est un objet de communication utilisé dans les applications Android pour la transmission de données et la navigation entre des composants tels que les activités, les services ou les diffuseurs (BroadcastReceiver). Il permet d'exécuter des actions comme l'ouverture d'un nouvel écran, l'envoi de données ou encore la demande d'une action à une autre application. Il ya 2 types :

6.1 Intent explicite

Utilisé pour démarrer une activité ou un service spécifique dans une application. L'Intent cible une classe précise, ce qui signifie que la destination est connue à l'avance. Par exemple Passer d'un écran à un autre au sein d'une même application.

Dans cet exercice, nous avons employé un Intent explicite pour passer de données entre un certain nombre de fonctionnalités de l'application. Nous avons visé à recueillir les données saisies par l'utilisateur, les envoyer au sein de la seconde fonctionnalité et y ajouter des boutons de transition entre les écrans.

6.1.1 Gestion de la navigation et transmission des données entre activités

Tout d'abord, dans l'activité principale (`MainActivity`), nous avons récupéré les informations saisies par l'utilisateur dans les champs du formulaire, à savoir : nom, prénom, âge, domaine de compétences et numéro de téléphone. Ces données ont ensuite été envoyées à une seconde activité (`SecondActivity`) via un `Intent` explicite. Pour ce trans-

fert, nous avons utilisé la méthode `putExtra()`, qui permet d'attacher des informations supplémentaires avant d'appeler la seconde activité avec `startActivity()`.

Dans `SecondActivity`, nous avons extrait les données transmises par l'`Intent` et les avons affichées à l'écran à l'aide de `TextView`. Cette activité comprend également deux boutons :

— Un bouton « OK », qui permet à l'utilisateur de passer à une troisième activité (`ThirdActivity`).

— Un bouton « Retour », qui permet de revenir à `MainActivity` en fermant simplement `SecondActivity`.

L'accès à `ThirdActivity` se fait lorsque l'utilisateur appuie sur « OK » dans `SecondActivity`. Son contenu étant libre, nous avons choisi un écran minimaliste avec un bouton permettant de revenir directement à `MainActivity`. Afin d'optimiser la navigation, nous avons configuré l'`Intent` pour fermer toutes les activités intermédiaires, évitant ainsi d'accumuler des écrans inutiles.

FIGURE 8 – SecondActivity : les informations saisies

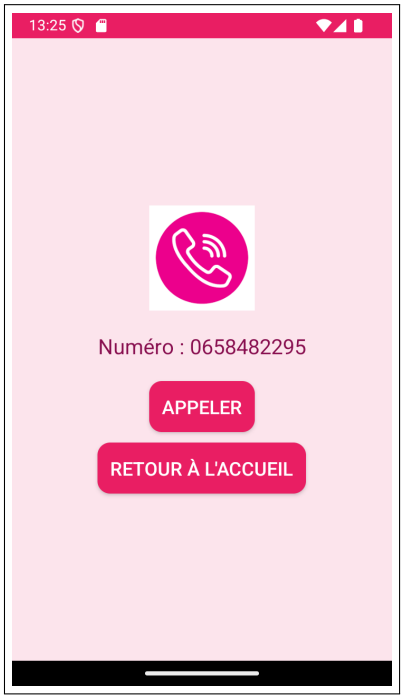

FIGURE 9 – ThirdActivity : nouvelle page

6.1.2 Création des fichiers XML pour chaque activité

Afin que chaque activité puisse disposer de son propre affichage, il est essentiel de créer un fichier XML de mise en page (`layout.xml`) correspondant à chaque activité dans le dossier `res/layout/`.

— `layout.xml` → Définit l'interface de `MainActivity` contenant le formulaire de saisie et le bouton de validation.

— `activity_second.xml` → Contient les `TextView` affichant les informations reçues ainsi que les boutons « OK » et « Retour ».

— `activity_third.xml` → Interface minimale avec un bouton permettant de retourner à `MainActivity` et un bouton appeler qu'on va expliquer dans la section suivante.

Cela permet d'avoir une séparation claire entre la logique métier (Kotlin) et l'interface utilisateur (XML), facilitant ainsi la maintenance et l'évolution de l'application.

6.1.3 Mise à jour du fichier `AndroidManifest.xml`

Enfin, nous avons mis à jour le fichier `AndroidManifest.xml` en déclarant les trois activités (`MainActivity`, `SecondActivity` et `ThirdActivity`). Cette étape est essentielle pour permettre à Android de reconnaître et gérer correctement ces activités.

Grâce à cette approche, nous avons appris à utiliser efficacement les `Intents` explicites pour transmettre des données et gérer la navigation entre plusieurs écrans, améliorant ainsi l'expérience utilisateur et la structure de notre application.

6.2 Intent implicite

Utilisé pour demander une action sans préciser une application spécifique. Android identifie automatiquement quelle application est capable de répondre à la requête. Exemple : Ouvrir un navigateur pour afficher une page web, lancer une application de messagerie pour envoyer un e-mail ou utiliser l'appareil photo pour prendre une photo.

Dans notre cas on va mettre en place une fonctionnalité d'appel téléphonique permettant à l'utilisateur de composer le numéro qu'il a saisi précédemment dans le formulaire. Pour cela, nous avons utilisé un Intent implicite afin d'ouvrir l'application téléphone du smartphone avec le numéro renseigné.

6.3 Transmission et affichage du numéro de téléphone

Dans `SecondActivity`, avant de naviguer vers `ThirdActivity`, nous avons veillé à transmettre la donnée « PHONE » via un **Intent explicite** en utilisant la méthode `putExtra()`. Cette méthode permet d'attacher une valeur (ici, le numéro de téléphone) à l'`Intent` afin qu'elle puisse être récupérée dans l'activité suivante. Dans `ThirdActivity`, au moment du chargement de l'écran (`onCreate()`), nous avons extrait cette valeur à l'aide de `getStringExtra("PHONE")`, ce qui nous a permis d'afficher dynamiquement le numéro récupéré dans un `TextView`.

6.4 Mise en place d'un Intent Implicite pour l'appel téléphonique

Pour permettre à l'utilisateur de composer un appel, nous avons implémenté un **Intent Implicite** avec l'action `ACTION_DIAL`, qui permet de préremplir le numéroteur téléphonique avec le numéro transmis.

— L'`Intent` est instancié dynamiquement avec `Intent(Intent.ACTION_DIAL)`.

— L'URI du numéro (`tel:+numéro`) est attachée à l'`Intent` via la méthode : `intent.data = Uri.parse("tel:$phoneNumber")`.

— Enfin, l'`Intent` est déclenché par `startActivity(intent)`, ce qui ouvre directement l'application Téléphone du système avec le numéro saisi.

Remarque :

Nous avons utilisé `ACTION_DIAL` plutôt que `ACTION_CALL`, car Android empêche les appels automatiques pour des raisons de sécurité et de protection des utilisateurs. Avec `ACTION_DIAL`, l'appel n'est pas lancé immédiatement, mais l'utilisateur doit confirmer manuellement l'appel dans l'application Téléphone. et bien-sùr afin d'améliorer l'expérience utilisateur, nous avons mis à jour le fichier activity_third.xml en y ajoutant : un TextView, une icône de téléphone, un bouton "Appeler", un bouton "Retour" (vers l'acceuil).

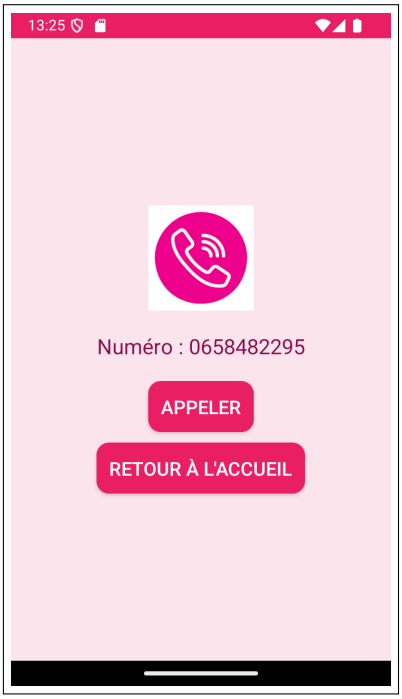

FIGURE 10 – ThirdActivity : page pour appeller

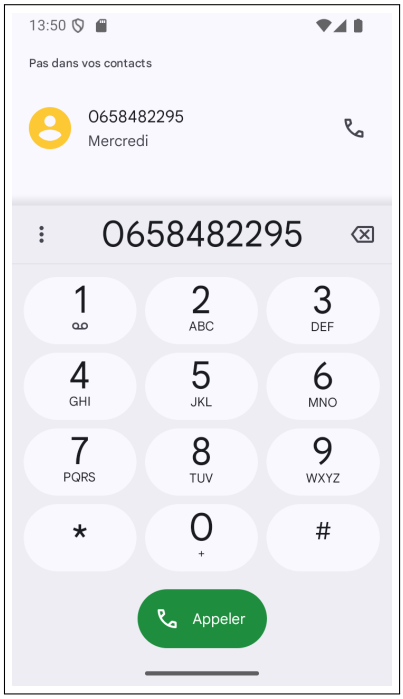

FIGURE 11 – Redirection vers l'application d'appel

7 Application simple pour consulter les horaires de trains

Dans cette section de rapport, nous décrivons le developpement d'une application de train par laquelle l'API SNCF et l'outil Material3 sont utilisés pour afficher les horaires de train. Nous présentons les étapes empruntées pour l'extraction des données des trains en temps réel et rendre les résultats accessibles sur une interface conviviale, en expliquant toutes l'application.

7.1 Présentation de l'API SNCF

L'API SNCF est une interface de programmation applicative permettant d'accéder aux données du réseau ferroviaire français. Elle offre une multitude de services, notamment :

— La consultation des horaires des trains en fonction des gares de départ et d'arrivée.

— Le suivi en temps réel des retards, annulations ou suppressions de trains.

— L'accès aux itinéraires alternatifs en cas de perturbation.

— La recherche des gares et leur identification via un code unique.

Cette API est conçue selon une architecture RESTful, ce qui signifie qu'elle est accessible via le protocole HTTP et retourne des réponses au format JSON.

7.2 Processus d'Accès et Authentification

7.2.1 Obtention des Accès

Pour utiliser l'API, il faut d'abord s'inscrire sur la plateforme SNCF API (`https://www.digital.sncf.com`) et générer une clé d'authentification. Cette clé doit être incluse dans chaque requête effectuée à l'API.

7.2.2 Méthodes d'Appel

L'API SNCF fonctionne sur un modèle d'appels `GET`, ce qui signifie qu'elle permet uniquement la lecture des informations. Les paramètres sont passés directement dans l'URL sous forme de *query parameters* (paramètres de requête), comme :

— La gare de départ et d'arrivée, identifiées par un code unique appelé **code UIC**.

— La date et l'heure du trajet souhaité.

— Les options de filtrage, comme le type de train (*TGV*, *TER*, *Intercités*...).

7.2.3 Sécurité et Limites d'Utilisation

L'API impose certaines restrictions, notamment :

— Un nombre limité de requêtes par minute et par jour, en fonction du type d'abonnement.

— Une obligation d'authentification pour éviter toute surcharge abusive.

— Des temps de latence dus à la récupération des données en temps réel.

7.3 Structure des Données Renvoyées par l'API

Lorsqu'une requête est envoyée à l'API SNCF, la réponse reçue est structurée en plusieurs niveaux d'informations :

— **Liste des trajets possibles**, chacun comprenant :

  — Horaires de départ et d'arrivée.

  — Numéro du train et type de service (*TGV*, *TER*, *OUIGO*, etc.).

  - Durée totale du trajet.
  - Gares desservies sur le parcours.
- **Informations sur chaque segment du trajet**, incluant :
  - Nom et code de chaque gare impliquée.
  - Détails sur les correspondances si nécessaires.
  - Perturbations éventuelles (retards, annulations).
- **Statut du train**, incluant :
  - Confirmation de la circulation du train à l'heure indiquée.
  - Signalement des retards en minutes.
  - Informations sur les changements de voie en gare.

Grâce à cette structure, il est possible de reconstituer un itinéraire détaillé, affiché ensuite sous forme de liste ou de tableau dans l'application.

7.4 Exploitation des Codes UIC pour Identifier les Gares

L'API SNCF utilise un identifiant unique pour chaque gare, appelé **code UIC** (*Union Internationale des Chemins de fer*). Ce code, qui est une suite de 7 chiffres, permet d'éviter les confusions entre les gares ayant des noms similaires.

7.4.0.1 Exemple de codes UIC de gares en France :

- **Gare de Lyon (Paris)** : 0087777001
- **Montpellier Saint-Roch** : 0087751008
- **Marseille Saint-Charles** : 0087730008

L'utilisateur ne saisit pas ces codes directement. L'application effectue d'abord une recherche automatique pour associer la ville ou le nom de la gare tapé par l'utilisateur au bon code UIC avant d'envoyer la requête à l'API.

7.5 Avantages de l'Intégration de l'API SNCF dans l'Application

L'utilisation de cette API dans notre projet apporte plusieurs bénéfices :

- **Données toujours à jour** : Les horaires et perturbations sont récupérés en temps réel.
- **Simplicité pour l'utilisateur** : Il suffit de saisir la ville de départ et d'arrivée pour obtenir tous les horaires disponibles.
- **Possibilité de gérer les imprévus** : En cas de retard ou d'annulation, des itinéraires alternatifs peuvent être proposés.
- **Accès à un grand nombre de données** : Au-delà des horaires, l'API permet aussi d'afficher des informations détaillées sur chaque train.

8 Explication Détaillée de l'Application de train

Cette application Android offre un utilisateur la possibilité de regarder les horaires de train entre deux stations de ville, de réserver un sièges, de valider son paiement, et de se munir de un ticket. L'application obéit à un modèle modulaire et se fonde sur la SNCF API pour charger les données en temps reel. Voici un chronologie détaillée de la expérience utilisateur et la description de chaque composant de système.

8.1 Déroulement Global de l'Application

L'application suit une séquence bien définie, partant de la saisie de l'itinéraire jusqu'à l'obtention du billet de train. Voici les étapes clés :

1. Saisie des informations du voyage (ville de départ, ville d'arrivée, date)
2. Vérification de la connectivité internet
3. Envoi de la requête à l'API SNCF pour récupérer les horaires disponibles
4. Affichage des trajets trouvés et sélection d'un train
5. Choix du siège dans le train sélectionné
6. Validation et résumé de la réservation
7. Paiement des billets
8. Affichage du billet sous forme de QR Code

Chacune de ces étapes est gérée par un fichier spécifique de l'application. Voici l'explication de chaque fichiers ou on a intègrer plusieurs concepts avancés d'Android :

— **RecyclerView** pour l'affichage des résultats.
— **GridLayout** pour la sélection des sièges.
— **Retrofit** pour la communication avec l'API.
— **Intents explicites** pour la navigation entre les activités.
— **Génération de QR Code** pour la billetterie.

8.2 MainActivity.kt - Interface Principale et Recherche de Trajets

Ce fichier représente l'écran d'accueil de l'application où l'utilisateur peut rechercher des trajets en train en saisissant la gare de départ, la gare d'arrivée et la date du voyage. Il envoie ensuite une requête à l'API SNCF pour obtenir les trajets disponibles. L'écran principal permet à l'utilisateur de :

— **Saisir la ville de départ et la ville d'arrivée** via des champs de texte.
— **Sélectionner la date du voyage** avec un `DatePickerDialog`.
— **Lancer la recherche** en cliquant sur un bouton.

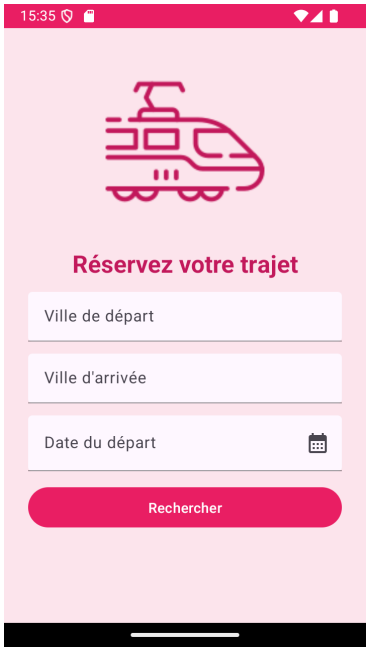

FIGURE 12 – Page d'acceuil de l'application de train

8.2.1 Traitement des entrées et communication avec l'API

Avant d'envoyer la requête, plusieurs vérifications sont effectuées :

— Vérification que tous les champs sont remplis, sinon un `Toast` affiche un message d'erreur.

— Vérification de la connexion Internet avant d'effectuer la requête. Cette vérification est réalisée à l'aide du `ConnectivityManager`.

— Conversion des valeurs saisies au format requis par l'API (dates et noms de ville en identifiants SNCF et l'identifiant de chaque gare est déterminé grâce à une fonction de correspondance).

— Envoi d'une requête HTTP avec `Retrofit` de manière asynchrone.

Si la requête réussit, les résultats sont envoyés à `ResultsActivity` pour être affichés.

8.3 ResultsActivity.kt - Affichage des Résultats

Ce fichier gère l'affichage des résultats de la recherche de trajets sous forme de liste. L'utilisateur peut sélectionner un trajet pour passer à l'étape suivante. Les résultats sont récupérés via un `Intent` depuis `MainActivity`.

8.3.1 Présentation des résultats

Les résultats sont affichés sous forme de liste avec un `RecyclerView`, optimisant le rendu des éléments affichés. Chaque trajet affiche :

— L'heure de départ et d'arrivée.

— La durée du trajet.

— Le nombre de correspondances.

L'utilisateur peut sélectionner un trajet pour accéder à la page de sélection des sièges. Un `OnClickListener` est attaché à chaque élément pour transmettre l'objet `Journey` à l'activité `SeatSelectionActivity`.

```
    val adapter = SNCFTrajetAdapter(trajetsList) { trajet ->
    val intent = Intent(this, SeatSelectionActivity::class.java).apply {
        putExtra("trajet", trajet)
    }
    startActivity(intent)
}
```

FIGURE 13 – Résultat de recherche entre Marseille et Paris

8.4 SeatSelectionActivity.kt - Sélection des Sièges

Cet écran permet à l'utilisateur de sélectionner un siège pour son trajet. Certains sièges sont déjà réservés et ne peuvent pas être choisis. Un `GridLayout` est utilisé pour représenter les sièges sous forme de boutons interactifs.

8.4.1 Affichage des sièges

— Les sièges **réservés** sont affichés en rouge.

— Les sièges **disponibles** en gris.

— Un siège sélectionné par l'utilisateur devient bleu.

8.4.2 Gestion de la sélection

— Le siège précédemment sélectionné est réinitialisé.

— Le nouveau siège sélectionné est mis en surbrillance.

```
seatImage.setOnClickListener {
    if (seatStates[i] == "reserved") return@setOnClickListener
    selectedSeat?.let { previousSeat ->
        seatStates[previousSeat] = "available"
        (seatGrid.getChildAt(previousSeat - 1) as ImageView).setColorFilter(getSeatCo
    }
    seatStates[i] = "selected"
    seatImage.setColorFilter(getSeatColor("selected"))
    selectedSeat = i
}
```

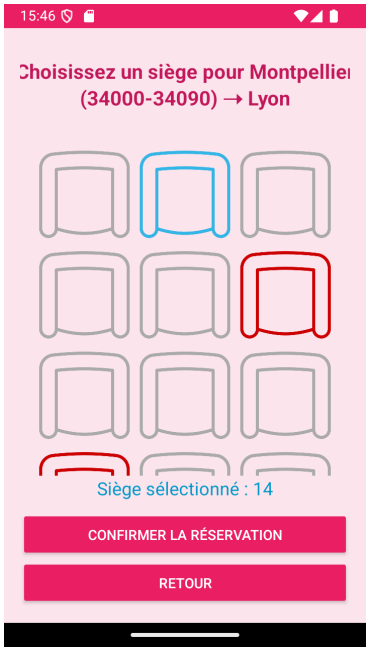

FIGURE 14 – Sélection des sièges pour un trajet Montpellier-Lyon

8.5 SummaryActivity.kt - Récapitulatif du Trajet

Cet écran affiche un résumé du voyage sélectionné par l'utilisateur, en récupérant les informations essentielles depuis l'Intent de l'écran précédent. Une des particularités est que l'API SNCF ne fournit pas directement le prix des billets, c'est pourquoi l'application génère un prix aléatoire entre 20 et 100 euros. Dans SummaryActivity, les informations du trajet sont récupérées depuis l'Intent qui a été envoyé depuis SeatSelectionActivity. Cela inclut : L'objet Journey, qui contient les détails du trajet (gare de départ, arrivée, heure de départ), le numéro du siège sélectionné et un prix généré aléatoirement.

L'API SNCF ne fournit pas directement les tarifs des billets dans les données retournées. Afin d'afficher un prix sur l'écran récapitulatif, l'application génère un prix aléatoire entre 20 et 100 euros. Une fois les données récupérées, elles sont affichées à l'écran via un TextView. Lorsque l'utilisateur appuie sur le bouton Confirmer la Réservation, il est redirigé vers PaymentActivity. Voici un extrait du code :

```
    confirmButton.setOnClickListener {
    val intent = Intent(this, PaymentActivity::class.java).apply {
        putExtra("trajet", trajet)
        putExtra("selectedSeat", selectedSeat)
        putExtra("prix", prix) //  Envoi du prix généré vers l'écran de paiement
    }
    startActivity(intent)
}
```

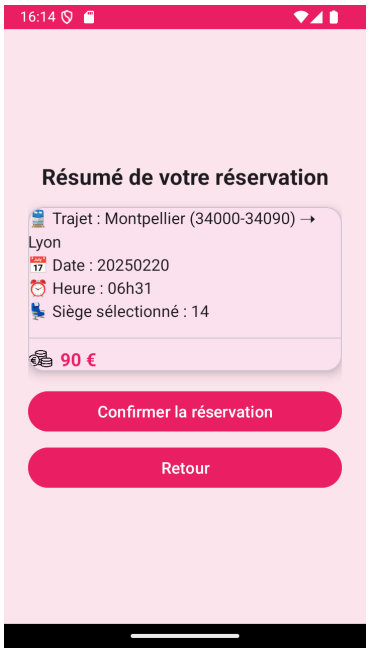

FIGURE 15 – Resume de la commande d'un trajet Montpellier-Lyon

8.6 PaymentActivity.kt - Paiement

Cet écran permet à l'utilisateur d'entrer ses informations de paiement pour finaliser la réservation.

8.6.1 Vérification des Champs

Avant de valider, l'application effectue plusieurs vérifications :

— Le numéro de carte doit contenir exactement 16 chiffres.

— La date d'expiration doit être dans un format valide.

— Le CVV doit contenir exactement 3 chiffres.

```
if (cardNumber.length != 16) {
    showToast("Le numéro de carte doit contenir 16 chiffres")
    return false
}
```

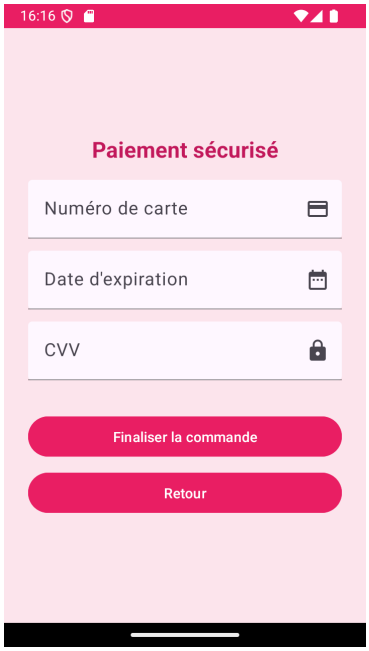

FIGURE 16 – Page de paiement pour la commande d'un trajet Montpellier-Lyon

8.7 Génération du Billet et QR Code

Dans `TicketQRCodeActivity`, un QR Code est généré dynamiquement à partir du numéro de billet.Une fois le paiement validé, l'application affiche ce QR Code contenant les informations du billet. Ce QR Code pourrait être scanné pour valider l'accès au train. Il permet : de stocker toutes les informations essentielles du billet de manière compacte et d'être scanné facilement par un contrôleur ou un terminal d'embarquement. Dans TicketQRCodeActivity, un QR Code est généré en utilisant une bibliothèque de génération de codes-barres (com.google.zxing). Le QR Code est une représentation graphique encodée des informations du billet, incluant :

- Le trajet (gare de départ et d'arrivée). - L'heure de départ. - Le numéro du siège. - Le prix du billet. Avant de générer le QR Code, l'application récupère les informations du trajet à partir de l'Intent provenant de PaymentActivity.

```
val trajet = intent.getParcelableExtra<Journey>("trajet")
val selectedSeat = intent.getIntExtra("selectedSeat", -1)
val prix = intent.getIntExtra("prix", 0)
```

Pour générer un QR Code, il faut d'abord convertir les informations du billet en une chaîne de texte formatée. Cette chaîne contient toutes les informations que l'on souhaite inclure dans le QR Code. Le format est simple et lisible, ce qui est pratique si on scanne

le QR Code avec une application de lecture. L'application utilise la bibliothèque ZXing pour convertir cette chaîne en QR Code.

```
val qrCodeWriter = QRCodeWriter()
try {
    val bitMatrix = qrCodeWriter.encode(qrData, BarcodeFormat.QR_CODE, 512, 512)
    val qrBitmap = Bitmap.createBitmap(512, 512, Bitmap.Config.RGB_565)

    for (x in 0 until 512) {
        for (y in 0 until 512) {
            qrBitmap.setPixel(x, y, if (bitMatrix[x, y]) Color.BLACK else Color.W
        }
    }

    val qrImageView = findViewById<ImageView>(R.id.qrCodeImageView)
    qrImageView.setImageBitmap(qrBitmap)

} catch (e: Exception) {
    e.printStackTrace()
}
```

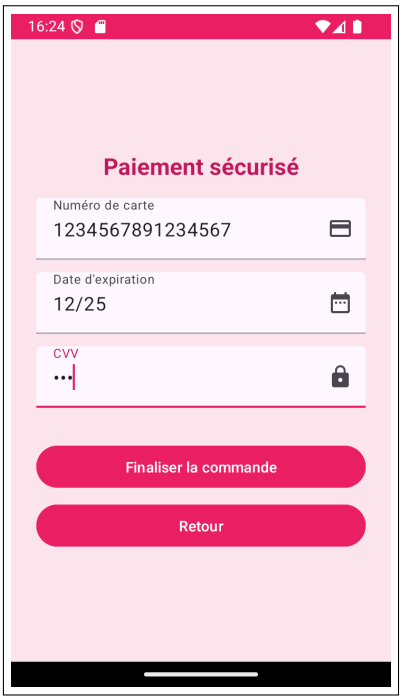

FIGURE 17 – Page de paiement

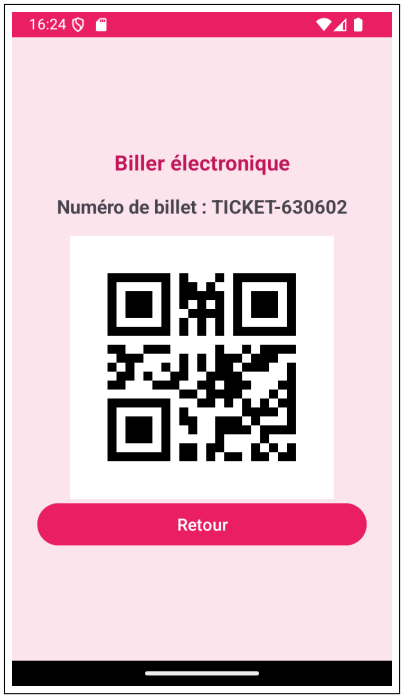

FIGURE 18 – Qr$_c$*odegenerer*

L'API est appelée de manière asynchrone (`CoroutineScope`) pour éviter de bloquer l'interface utilisateur.

8.8 SNCFService.kt - Communication avec l'API SNCF

Ce fichier contient la configuration pour interagir avec l'API SNCF via `Retrofit`.

8.8.1 Modèles de Données SNCF

— **SNCFResponse** : Stocke la liste des trajets récupérés.

— **Journey** : Contient les informations d'un trajet (heures, correspondances, durée).

— **Fare** : Gère les informations tarifaires (prix et devise).

— **Ticket** : Stocke les liens des billets électroniques.

8.9 Conclusion sur cette application de train

Cette application Android offre aux utilisateurs la possibilité de chercher des itinéraires de train, choisir un siège, confirmer un paiement et recevoir un ticket sous forme de QR Code. Elle propose une expérience de voyage fluide et pratique grâce à son interface conviviale et à une gestion efficace des données via l'API SNCF. L'architecture de l'application repose sur plusieurs activités qui interagissent entre elles en utilisant des Intents pour transférer les informations du trajet et du billet. Les données sont récupérées en temps réel depuis l'API SNCF, mais le prix du billet n'étant pas disponible, nous avons implémenté une génération aléatoire du tarif afin d'assurer une expérience réaliste.

L'un des aspects clés de cette application est la génération d'un QR Code après le paiement, permettant une gestion numérique des billets. Ce QR Code, encodé à l'aide de la bibliothèque ZXing, contient toutes les informations nécessaires au voyage et peut être scanné pour validation.

Pour le design et l'expérience utilisateur :nous avons utilisé Material3 dans les fichiers XML de l'application. Cette approche a permis d'obtenir : un design épuré et harmonieux, en respectant les guidelines Material Design, des composants modernes tels que TextInputEditText, MaterialButton et CardView pour améliorer l'interaction utilisateur et une cohérence visuelle sur tous les écrans, garantissant une navigation intuitive.

9 Application simple d'agenda

L'objectif de cette application est de fournir une interface intuitive permettant aux utilisateurs de :

— Afficher les événements associés à une date spécifique.

— Ajouter de nouveaux événements pour une date sélectionnée.

— Supprimer des événements avec une confirmation.

Ce projet met en œuvre plusieurs concepts clés du développement Android, notamment la gestion d'une interface graphique avec `RecyclerView`, `CalendarView`, et l'utilisation d'`Intent` pour la navigation entre les écrans.

9.1 Architecture de l'Application

L'application repose sur une architecture modulaire composée de trois composants principaux :

9.1.1 MainActivity (Écran principal)

L'activité principale gère l'affichage du calendrier et des événements pour une date sélectionnée. Elle inclut :

— Un `CalendarView` permettant de naviguer entre les dates.

— Un `RecyclerView` affichant la liste des événements du jour.

— Un `TextView` pour afficher la date sélectionnée.

— Un `Button` pour accéder à l'écran d'ajout d'événements.

9.1.2 EventsAdapter (Adaptateur pour RecyclerView)

L'adaptateur permet d'afficher dynamiquement les événements du jour dans un `RecyclerView`. Il gère :

— L'affichage des événements sous forme de liste.

— La mise à jour des événements après ajout ou suppression.

— La gestion des clics pour supprimer un événement après confirmation.

9.1.3 AddEventActivity (Écran d'ajout d'événements)

Cette activité secondaire permet aux utilisateurs d'ajouter de nouveaux événements à une date sélectionnée. Elle comprend :

— Un `EditText` pour saisir le titre de l'événement.

— Un `EditText` pour le lieu.

— Un `EditText` pour l'heure (avec un `TimePickerDialog`).

— Un `Spinner` pour sélectionner un emoji.

— Un `Button` pour confirmer l'ajout de l'événement.

Lors de la validation, l'événement est renvoyé à `MainActivity` via `setResult()`, et la date est marquée sur le calendrier.

9.2 Fonctionnalités Détaillées

9.2.1 Affichage et Sélection d'une Date

Lorsqu'un utilisateur sélectionne une date dans le `CalendarView`, l'application :

— Met à jour la variable `selectedDate` avec la date sélectionnée.

— Charge les événements associés à cette date à partir du `HashMap`.

— Met à jour l'interface en affichant la liste des événements ou un message indiquant qu'aucun événement n'existe.

9.2.2 Gestion des Dates avec Événements

— Si une date contient des événements, elle est marquée en rouge sur le calendrier grâce à `EventDay`.

— Si tous les événements d'une date sont supprimés, la couleur rouge est retirée.

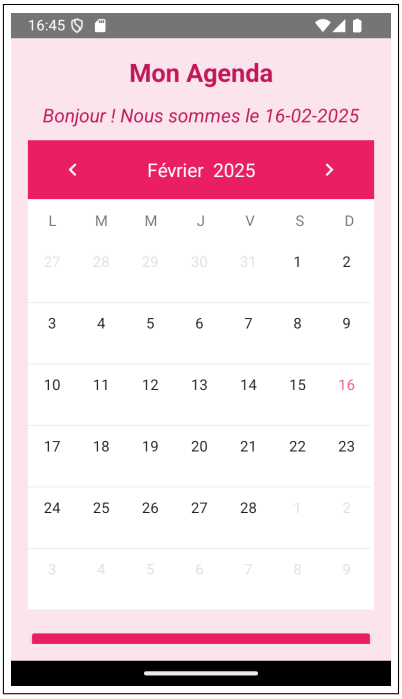

FIGURE 19 – Page du calendrier

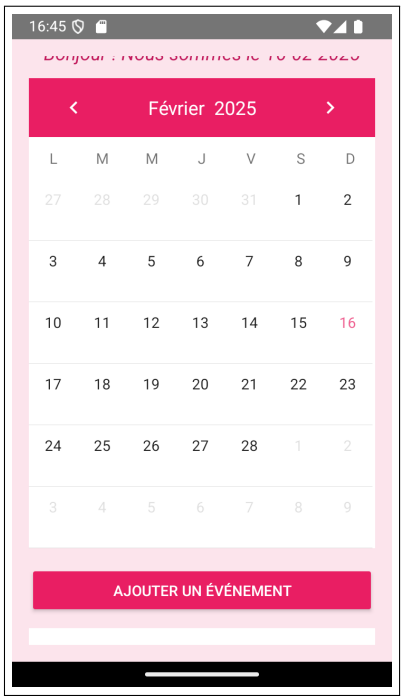

FIGURE 20 – Page du calendrier en scrollant vers le bas

9.3 Ajout d'un Événement

Lorsqu'un utilisateur appuie sur "Ajouter un événement" :

— L'application ouvre `AddEventActivity` en envoyant la date sélectionnée via un `Intent`.

— L'utilisateur saisit les informations de l'événement (titre, lieu, heure, emoji).

— À la validation, l'activité renvoie l'événement à `MainActivity` via `setResult()`.

— L'événement est ajouté à la liste et au `HashMap`, puis la date est marquée sur le calendrier.

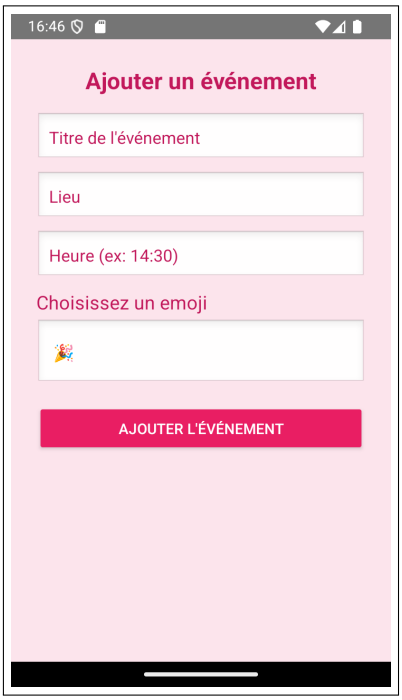

FIGURE 21 – Page du calendrier

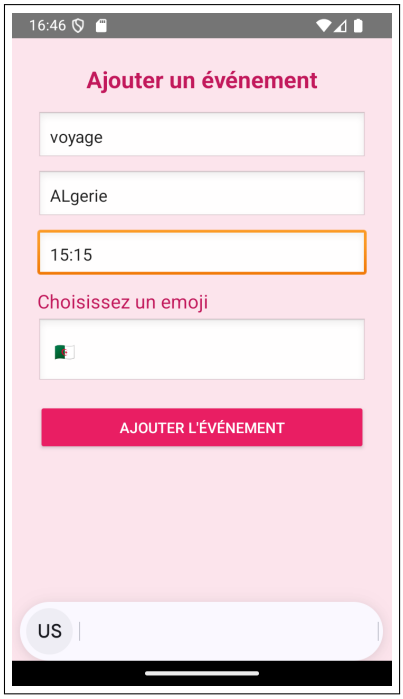

FIGURE 22 – Page du calendrier en scrollant vers le bas

L'application d'agenda implémente une indication visuelle pour signaler les dates contenant des événements. Cela est réalisé en ajoutant une icône rouge sur les dates concernées dans le CalendarView. Lorsque l'utilisateur ajoute un événement à une date, un marqueur rouge apparaît sur cette date, et lorsqu'il supprime le dernier événement de cette date, le marqueur disparaît. L'application utilise la classe EventDay de la bibliothèque MaterialCalendarView pour ajouter un indicateur visuel aux dates qui ont des événements.

Lorsqu'un événement est ajouté : La date concernée est extraite du calendrier, l'événement est ajouté à la liste des événements de cette date dans la HashMap et une icône rouge (ic_event_marker) est associée à cette date via EventDay.A la fin la liste des événements du calendrier est mise à jour pour afficher la marque rouge. Lorsqu'un événement est supprimé : Si l'événement supprimé était le dernier de la date sélectionnée : La date est retirée de la HashMap et l'icône rouge est retirée du CalendarView.

FIGURE 23 – Page du calendar avec le point rouge dans la date et la liste des evenement en bas du calendrier

9.4 Suppression d'un Événement

L'application d'agenda permet aux utilisateurs de supprimer un événement en effectuant un simple clic sur un événement listé pour une date sélectionnée. Cette action entraîne l'affichage d'une boîte de dialogue demandant une confirmation avant la suppression définitive de l'événement. L'interface principale de l'application affiche une liste d'événements pour la date sélectionnée, grâce à un RecyclerView. Chaque événement est représenté sous forme d'un élément interactif. Lorsqu'un utilisateur appuie sur un événement, l'application détecte cet appui via un gestionnaire de clics associé à l'adaptateur (EventsAdapter) et l'application déclenche alors l'affichage d'une boîte de dialogue pour confirmer l'intention de suppression. Si la suppression est confirmée : L'événement est retiré de la liste des événements affichés dans le RecyclerView, et est également supprimé du HashMap qui stocke les événements associés aux dates et l'adaptateur (EventsAdapter) est notifié du changement, ce qui met à jour l'affichage de la liste des événements.

FIGURE 24 – Affichage de la boite de dialogue pour supprimer l'evenement

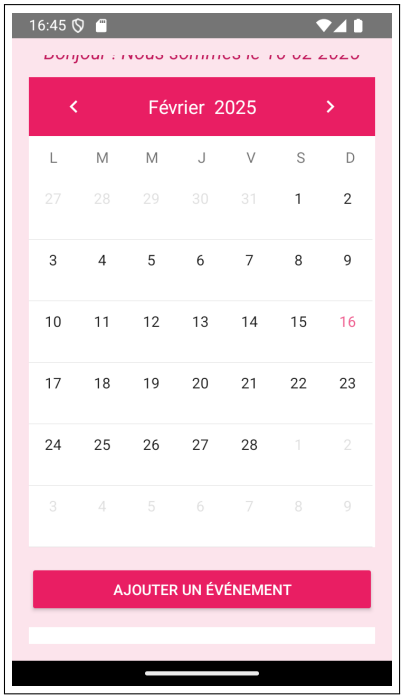

FIGURE 25 – Une fois appuyer sur oui l'evenement est supprimer

10 Probleme rencontrer au cours du developpement de ces applications

10.1 Problème de compatibilité avec le JDK

Lors du développement de l'application, un problème est survenu à l'ouverture du projet dans Android Studio. Le JDK installé sur la machine était en version 35, tandis qu'Android Studio et le Gradle Plugin Android (AGP) étaient configurés pour fonctionner avec la version 34. Cette incompatibilité empêchait la compilation et l'exécution de l'application, générant des erreurs liées à la version du JDK.

Pour résoudre ce problème, il a fallu modifier le fichier libs.version.toml en ajustant les versions des dépendances pour qu'elles soient compatibles avec JDK 34. Voici les valeurs mises à jour :

```
[versions]
agp = "8.5.0"
kotlin = "1.9.0"
coreKtx = "1.12.0"
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
lifecycleRuntimeKtx = "2.8.7"
activityCompose = "1.8.0"
composeBom = "2024.04.01"
```

```
material = "1.12.0"
```

10.2 Problème lié aux codes des villes dans l'API SNCF

Lors de l'intégration de l'API SNCF pour récupérer les horaires de train et les villes desservies, il a été constaté que les villes ne sont pas référencées par leur nom mais par des codes spécifiques. Ces codes ne sont pas toujours intuitifs et nécessitent une correspondance pour pouvoir être utilisés correctement dans les requêtes. l a fallu identifier et mapper les codes des villes avec leur nom respectif et cela a nécessité une vérification manuelle ou l'utilisation d'une source externe pour récupérer ces codes.

10.3 Absence des prix dans l'API SNCF

Un autre problème majeur rencontré avec l'API SNCF est que celle-ci ne fournit pas directement les prix des billets dans les résultats de requête. Cela posait un problème pour afficher une information tarifaire aux utilisateurs. Pour contourner cette limitation, une génération aléatoire de prix a été mise en place. Chaque trajet récupéré via l'API se voit attribuer un prix aléatoire compris entre 20 et 100 euros. Bien que cette approche ne reflète pas les prix réels, elle permet d'avoir une simulation fonctionnelle et un affichage cohérent dans l'application.

11 Conclusion general

Ce TP1 a permis d'explorer et de mettre en pratique plusieurs concepts du développement Android, notamment :

La gestion des interfaces graphiques avec RecyclerView et CalendarView. L'interaction entre activités avec Intent et la gestion des résultats d'activité. La gestion des événements utilisateur comme l'ajout et la suppression d'événements dans un agenda. L'adaptation aux contraintes techniques, comme les problèmes de compatibilité JDK et les limitations des API externes. L'amélioration de l'expérience utilisateur en mettant en place des marqueurs visuels et des dialogues de confirmation. Malgré certains défis techniques, les solutions mises en place ont permis de stabiliser et finaliser l'application. Cette première expérience pose une base solide pour la poursuite du développement mobile, avec des axes d'amélioration comme l'intégration d'une base de données SQLite, la synchronisation avec un serveur distant, et une meilleure gestion des données issues d'API externes.

Ce TP constitue donc une expérience enrichissante qui met en avant l'importance de l'adaptabilité et de la résolution de problèmes dans le développement mobile.

12 Lien GitHub

Le TP est disponible sur GitHub : Programmation-Mobile Repository