M1 IASD

HAI811I : Programmation mobile

TP1

CANI Nicolas

Faculté des Sciences
Université de Montpellier

14 Février 2026

Table des matières

1 Introduction

Ce TP se compose de trois applications distinctes dont l'objectif principal est l'initiation à l'environnement de développement Android Studio. À travers ces exercices, nous abordons les concepts fondamentaux de la programmation mobile sous Android.

- La première application se concentre sur la gestion d'un formulaire. Elle permet à un utilisateur de remplir des champs de saisie, de valider des données et d'utiliser la fonctionnalité d'appel téléphonique de l'OS, via le numéro saisi dans un champ. L'accent est mis sur la mise en application des concepts théoriques basiques vus en cours, et particulièrement sur les Intents et l'internationalisation de l'application via les fichiers de ressources `strings.xml`.
- La deuxième application approfondit la conception d'interfaces graphiques. En réutilisant les acquis précédents, le but est de proposer une interface de saisie d'itinéraires et de consultation d'horaires de train, mettant en avant l'ergonomie pour l'utilisateur.
- Enfin, la troisième application consiste en la réalisation d'un agenda classique. Elle offre à l'utilisateur la possibilité d'intégrer des événements à des dates précises et de les consulter ultérieurement, illustrant la gestion de listes et d'événements temporels.

Les technologies exploitées durant ce TP incluent le langage Kotlin, l'IDE Android Studio ainsi que diverses extensions de développement pour optimiser la production du code.

1.0.1 Lien du code source :

https://github.com/NicolasCani/TP1Mobile

1.0.2 Lien de la vidéo de présentation :

https://youtu.be/aJwSvntu7Ok

1.1 Architecture des applications (Commune)

Nos projets respectent la structure conventionnelle d'Android Studio. Pour illustrer cette architecture commune aux trois applications, nous détaillerons ici la plus aboutie et la plus complexe techniquement : l'application SNCF.

Voici l'architecture :

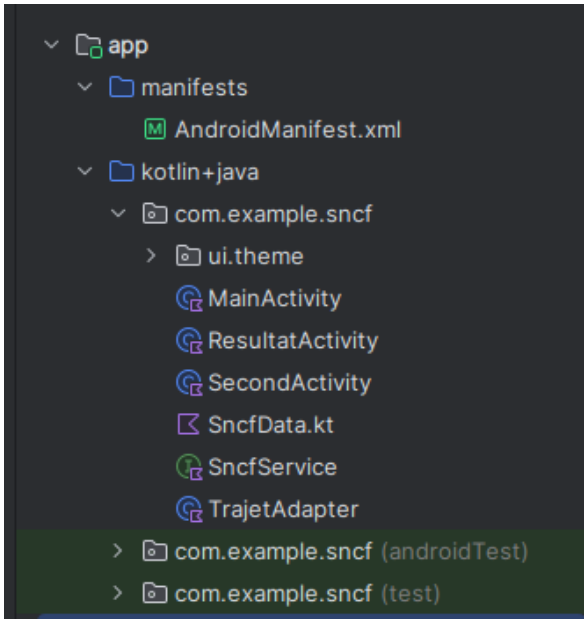

Fig. 1. – Architecture des projets (1)

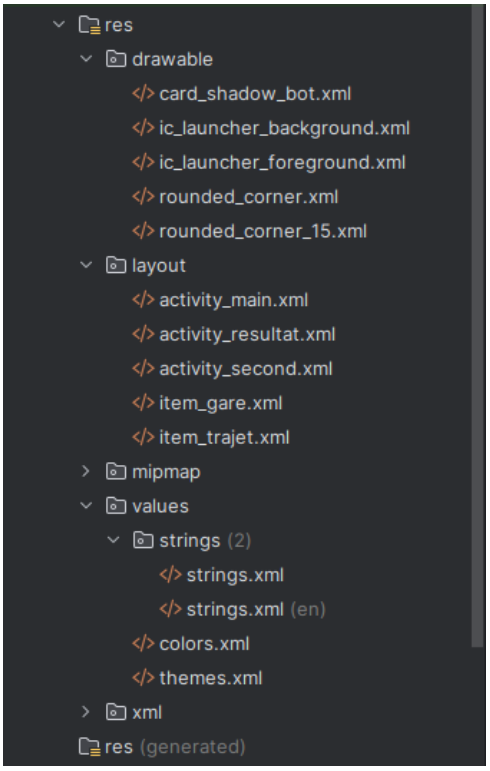

Fig. 2. – Architecture des projets (2)

L'arborescence du projet est générée automatiquement par Android Studio, à l'exception du répertoire « layout », crée manuellement.

1.1.1 Dossier kotlin+java

Ce dossier centralise la logique applicative, chaque fichier qui contient « Activity » représente un écran distinct de l'interface. Les éléments tels que SncfData, SncfService ou TrajetAdapter sont des classes et interfaces dédiées à l'utilisation de l'API SNCF et à la factorisation du code.

1.1.2 Dossier res

- Drawable :

Il définit les styles graphiques spécifiques. Par exemple, le fichier rounded_corner15 applique un border radius de 15dp aux vues, ce qui donne une esthétique plus moderne.

- Layout :

Ce dossier contient les structures XML des interfaces. notamment les fichiers liés aux Activity, donc chaque xml contenant « activity » est lié a une page de l'application. Et nous avons aussi les composants réutilisables, comme item_gare, qui sont injectés dans les XML d'activity principales pour éviter la duplication de code.

- values/strings :

Enfin, ce dossier est utilisé pour l'internationalisation de l'application.

2 Première application (Champs de saisis)

2.1 Logique de l'application

Dans cette applications nous cherchons a crée un formulaire classique comme mentionné lors de l'introduction, disponible en 2 langues, Français et Anglais.

Pour cette applications mon choix de layout XML fut TableLayout, car cela me sembait approprier et avoir un design épurer sans réel complication.

2.1.1 Première activité

L'interface de la première activité se présente ainsi :

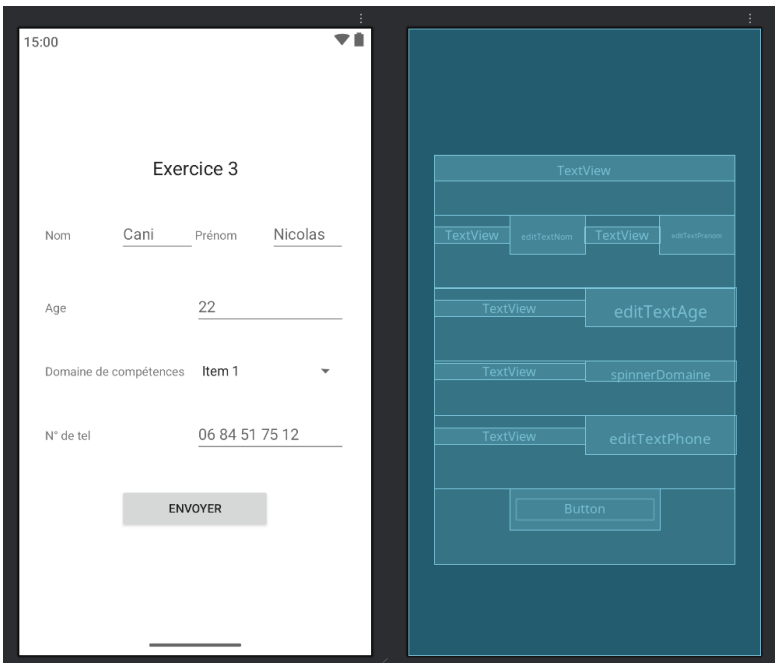

Fig. 3. – Squelette de la première activité

L'interface a également été générée par le code en Kotlin, conformément aux exigences du TP. Cette version, reproduit à l'identique le rendu du fichier XML, est disponible en commentaire dans le code source, MainActivity.kt

L'interface comporte des champs de saisie (EditText), chacun étant associé à un label descriptif afin de guider l'utilisateur. Les textes « Cani » et « Nicolas » correspondent à des suggestions de saisie (hints) configurés en XML via l'attribut suivant :

activity_main.xml

```
android:hint="@string/nameHint"
```

Code 1. – hint sur un champs

Une logique de validation a été implémentée pour chaque champ, une erreur de saisie déclenche une coloration rouge, tandis qu'une saisie valide est signalée en vert. Voici l'exemple d'un formulaire comportant des erreurs :

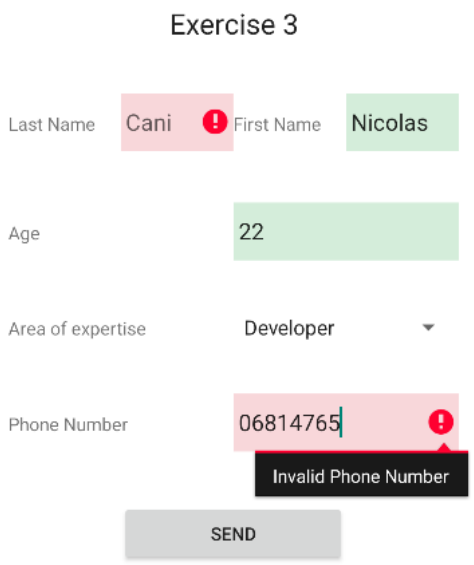

Fig. 4. – Exemple d'une saisie erronée

Sur cet exemple, l'application est configurée en anglais pour illustrer l'internationalisation. On remarque que le champ « Last Name » est resté vide seul le hint par défaut est visible et que le numéro de téléphone est incomplet (deux chiffres manquants).

Dès que le formulaire est intégralement valide, une boîte de dialogue de confirmation « AlertDialog » s'affiche avant l'envoi des données.

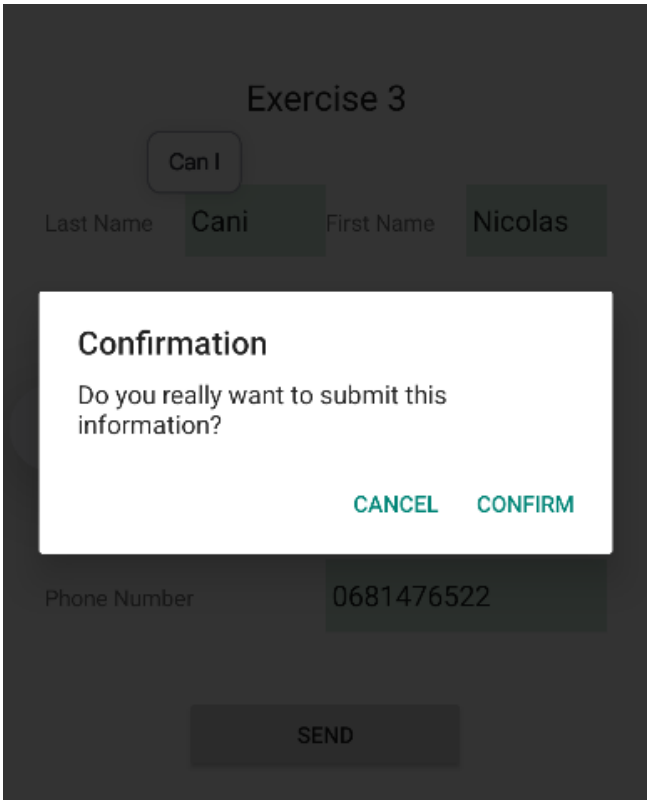

Fig. 5. – AlertDialog

À ce stade, deux actions sont possibles :

- Cliquer sur « Cancel » qui ferme simplement l'AlertDialog sans rien changer
- Cliquer sur « Confirm » qui crée un Intent avec les champs saisis et nous renvoi sur la seconde activité qui est le récapitulatif des données

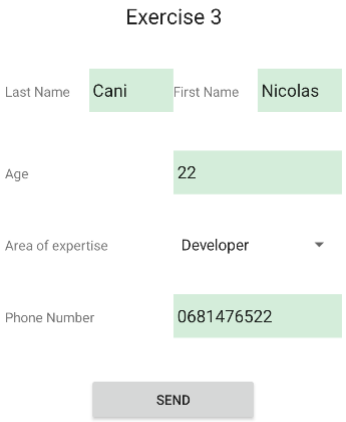

Fig. 6. – Le cas quand est cliqué « Cancel »

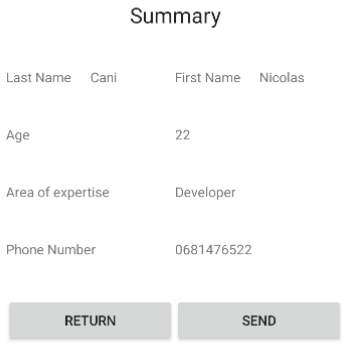

Fig. 7. – Le cas quand est cliqué « Confirm »

La gestion de ces deux interactions est implémentée dans l'AlertDialog :

MainActivity.kt

```
AlertDialog.Builder(this)
    .setTitle(R.string.dialog_title)
    .setMessage(R.string.dialog_message)
    .setPositiveButton(R.string.confirm) { _, _ ->
        val intent = Intent(this, ProfilActivity::class.java)
        intent.putExtra("nom", editNom.text.toString())
        //......
        intent.putExtra("tel", editPhone.text.toString())
        startActivity(intent) // Dans le cas où "Confirm est cliqué"
    }
    .setNegativeButton(R.string.cancel, null) // Dans le cas où "Cancel est
cliqué"
    .show()
```

Code 2. – AlertDialog

La validation finale permet ainsi de transiter vers la seconde activité.

2.1.2 Seconde activité

Voici comment se présente la seconde activité :

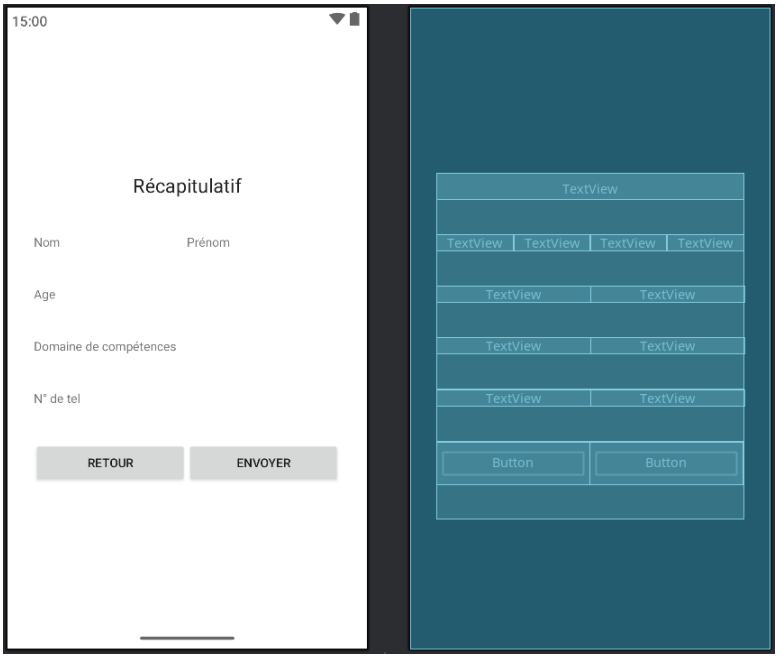

Fig. 8. – Squelette de a seconde activité

Sur cette interface, les TextView sont initialement vides. Cela est dû au fait qu'ils sont destinés à afficher les données transmises via l'Intent de la première activité, comme l'illustre la Fig. 7.

Ici, deux interactions sont possibles :

- Cliquer sur « Retour » qui va crée un intent explicite vers la première activité pour permettre de récupérer les données du formulaire sans que l'utilisateur ai a effectuer une nouvelle saisie. (Plus tard j'ai appris l'utilisation de la méthode finish() qui permet de faire le même résultat sans intent)
- Cliquer sur Envoyer, qui va crée un intent explicite aussi avec uniquement le numéro pour ensuite passer a la troisième activité.

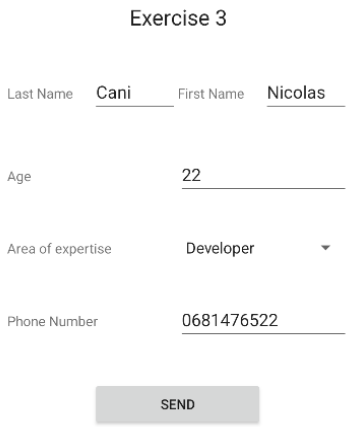

Fig. 9. – Le cas quand est cliqué « Return »

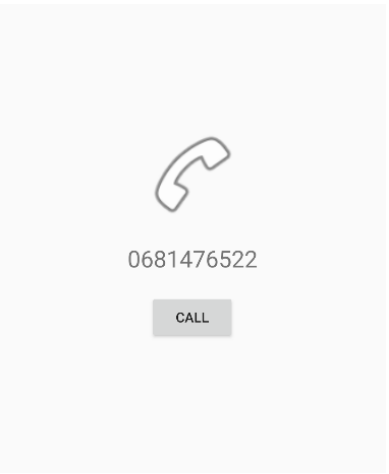

Fig. 10. – Le cas quand est cliqué « Send »

Le clique « Send » nous renvoi donc vers la troisième activité

2.1.3 Troisième activité

La troisième activité, plus concise, se présente comme ceci :

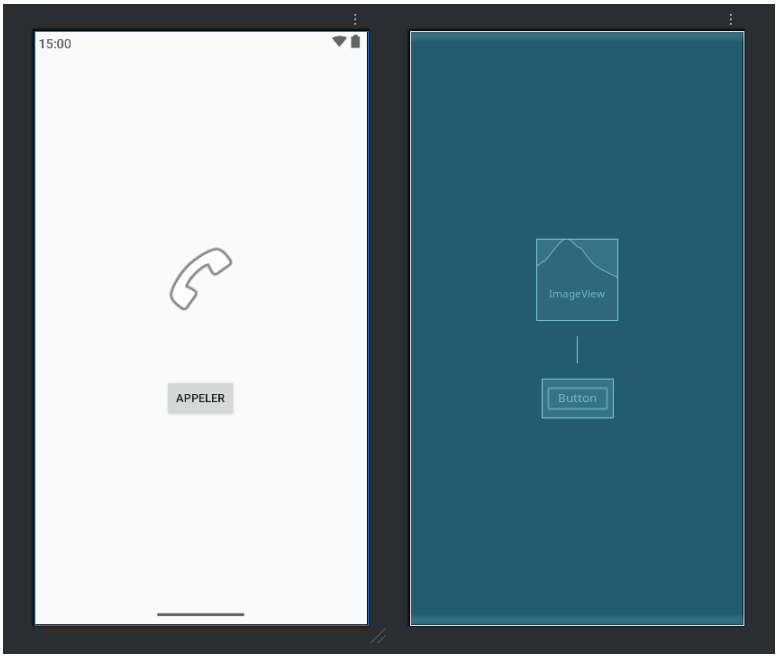

Fig. 11. – Squelette de a troisième activité

Cette interface affiche le numéro de téléphone récupéré grâce à l'Intent explicite de l'activité précédente. L'appui sur le bouton « Appeler » déclenche alors un Intent implicite ciblant l'application de numérotation native du système Android.

l'Intent implicite est géré par ce bloc de code :

MainActivity.kt

```
btnAppeler.setOnClickListener {
    val intentAppel = Intent(Intent.ACTION_DIAL)
    intentAppel.data = android.net.Uri.parse("tel:$numero")
    startActivity(intentAppel)
}
```

Code 3. – Intent Implicite

Il s'agit d'un Intent implicite car nous ne spécifions pas de classe cible précise. Nous utilisons l'action standard ACTION_DIAL, qui délègue au système Android de proposer les applications capables de gérer ce type de requête cela peut être l'application « Téléphone » par défaut, WhatsApp, ou autre. Le système trouve ces applications grâce aux Intent Filters déclarés dans leurs fichiers Manifest.

Dans notre cas précis, c'est l'application « Téléphone » par défaut, comme ceci :

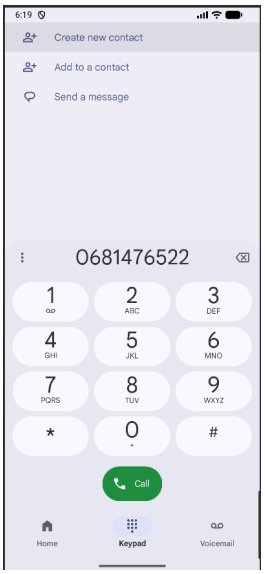

Fig. 12. – Application d'appel par défaut

2.2 Architecture Global de l'application

Notre application peut se présenter en une image comme ceci :

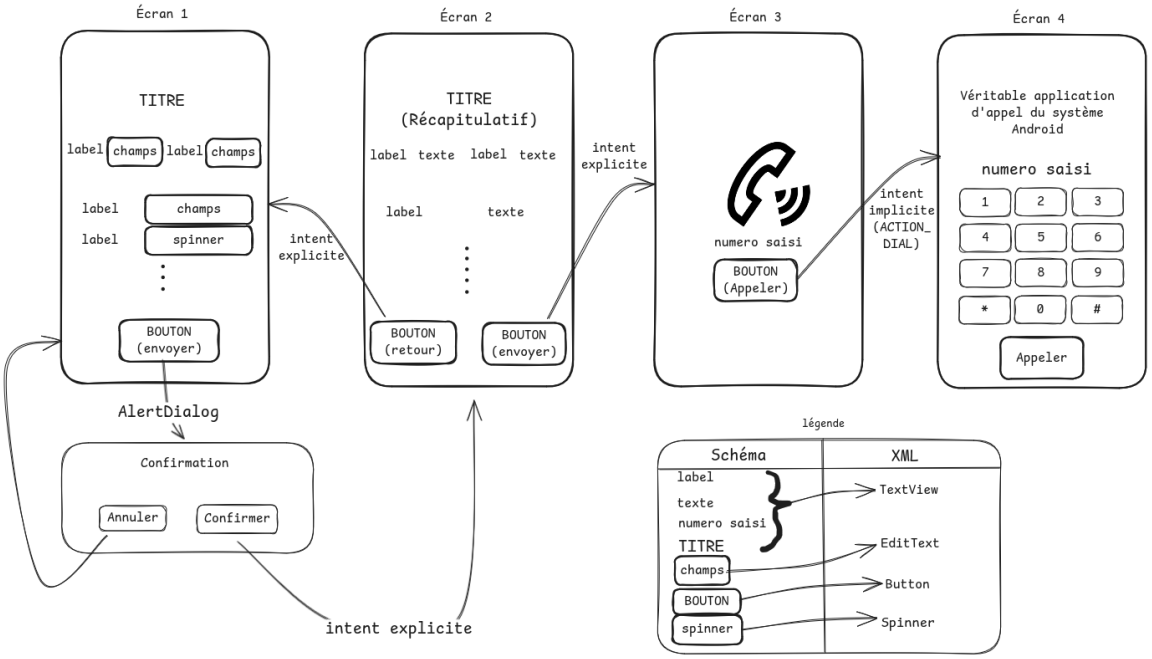

Fig. 13. – Schéma de l'application

2.3 Fonctionnalités - Application 1

Fonctionnalité	Statut
Remplir des champs	✓
Label à côté des champs	✓
Interface XML (TableLayout)	✓
Interface Kotlin	✓ Effectué pour la première page mis en commentaire dans le code source
Internationalisation (Plusieurs langues)	✓ (Français et Anglais)
Navigation	✘ (Pas de bouton retour a partir de l'activité 3)
Coloration visuelle des champs (Validation)	✓
Intent Explicite	✓
Intent Implicite	✓
Appel téléphonique	✓
Design	✘

Tableau 1. – Tableau des fonctionnalités implémentées pour la première application

L'absence d'un bouton de navigation de retour sur la troisième activité est un choix délibéré. En effet, la gestion du retour par le système Android permet déjà de revenir à l'état précédent de l'application. Pour un prototype, l'implémentation d'un bouton supplémentaire n'était donc pas techniquement nécessaire, bien qu'elle soit indispensable pour une interface utilisateur complète et standardisée.

Concernant l'aspect graphique, la priorité a été donnée à la maîtrise technique des concepts vu en cours plutôt qu'au design. L'objectif de cette première étape était de consolider les bases de la programmation mobile avant d'investir du temps dans l'esthétique de l'interface.

3 Deuxième application (SNCF)

3.1 Logique de l'application

L'objectif de cette application est de proposer un service de consultation d'itinéraires ferroviaires, permettant d'afficher les horaires correspondants sous forme de liste.

Pour la conception des interfaces XML, j'ai opté pour le ConstraintLayout. Après la réalisation du premier projet, mes recherches sur les standards actuels ont montré que ce layout est le plus utilisé dans les applications professionnelles. Sa souplesse d'utilisation permet de créer des interfaces complexes et modulables tout en optimisant les performances grâce à une hiérarchie de vues « à plat ».

J'ai apporté un soin particulier à l'ergonomie en m'inspirant de l'application réelle SNCF Connect. Mon objectif était de reprendre les codes visuels familiers des utilisateurs tout en me concentrant sur les fonctionnalités essentielles à l'exercice.

Afin de dépasser le cadre d'un simple design statique, j'ai choisi d'implémenter la véritable API SNCF et Retrofit. Cela permet de manipuler des données réelles, telles que les noms des gares et les horaires de train en temps réel. Par souci de lisibilité dans l'interface, l'affichage est actuellement limité aux 5 prochains départs, un paramètre que j'ai rendu facilement modulable dans le code source.

3.1.1 Première activité

L'interface de cette application dédiée à la SNCF s'ouvre sur l'écran suivant :

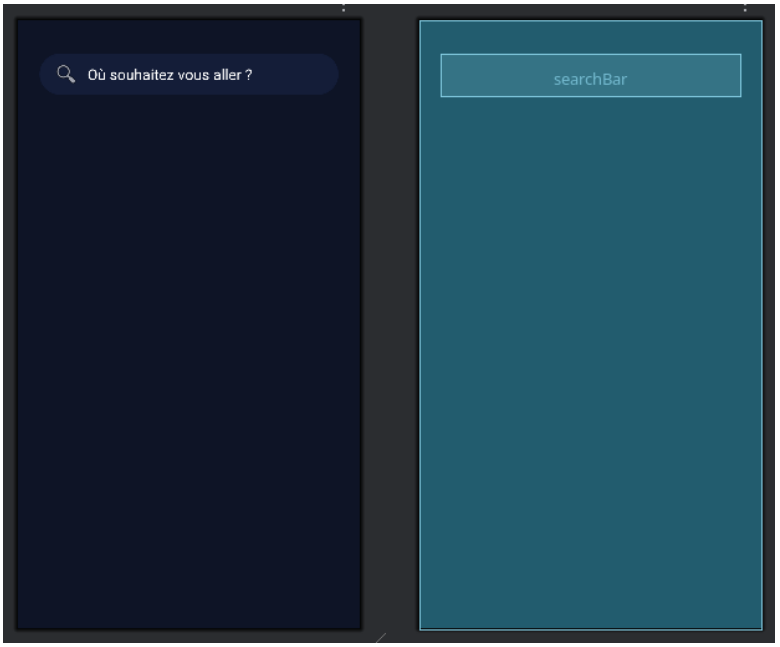

Fig. 14. – Squelette de la première activité

L'élément central est une barre de recherche (AutoCompleteTextView) qui permet de sélectionner la gare de destination. Les suggestions sont générées dynamiquement en interrogeant l'API SNCF à chaque saisie de l'utilisateur. Une fois la destination validée, l'application passe vers la seconde activité.

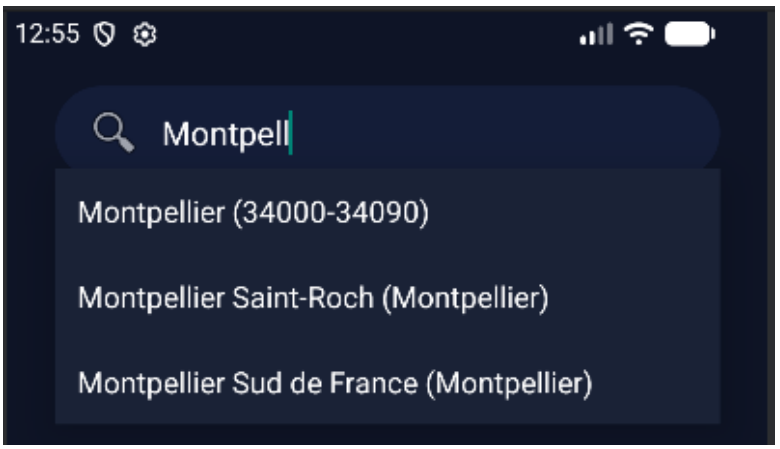

Fig. 15. – Recherche d'une ville

Comme illustré ci-dessus, une recherche sur « Montpell » retourne l'ensemble des points d'arrêt et gares correspondants fournis par l'API.

La sélection d'un résultat précis valide la destination et déclenche un Intent explicite vers la seconde activité, transmettant ainsi le nom de la gare et son identifiant technique indispensable pour la recherche d'itinéraires.

3.1.2 Seconde activité

L'interface de la seconde activité se structure ainsi :

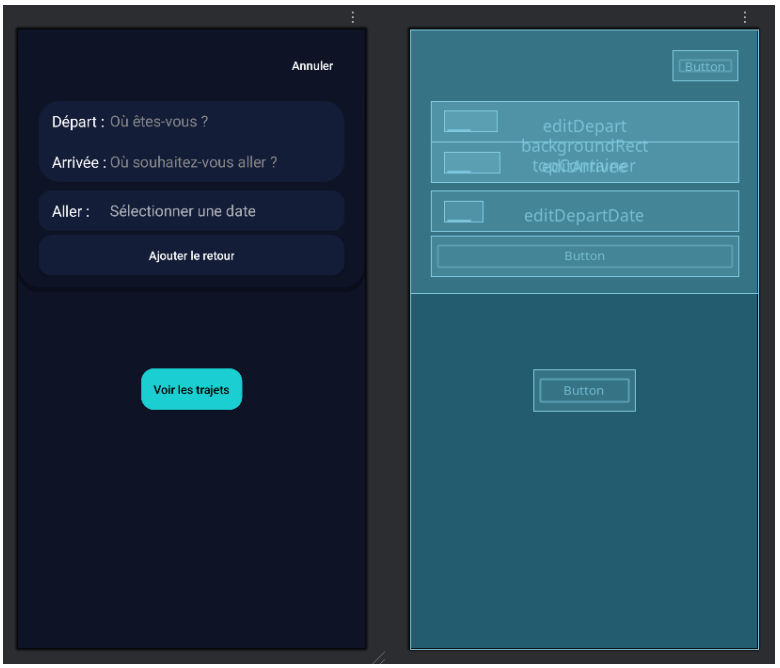

Fig. 16. – Squelette de la seconde activité

Cette page comporte deux barres de recherche dédiées au « Départ » et une « Arrivée ». Toutes deux exploitent l'API SNCF pour l'autocomplétion. Le champ d'arrivée est généralement pré-rempli grâce aux données transmises par l'Intent de l'activité précédente. L'utilisateur doit ensuite renseigner une date de départ (définie par défaut à la date du jour) et, s'il le souhaite, une date de retour optionnelle.

Cela ce passe comme ceci :

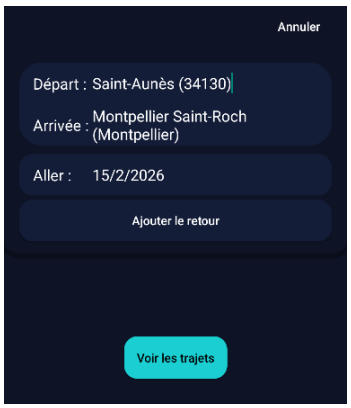

Fig. 17. – Ajout de la ville de départ

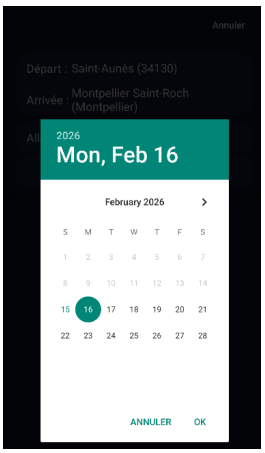

Fig. 18. – Ajout du retour

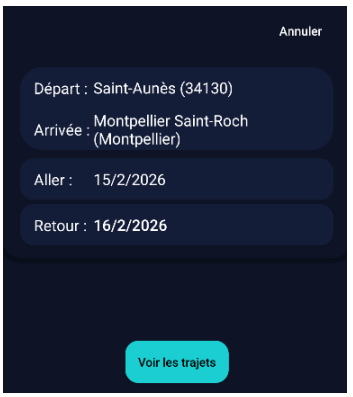

Fig. 19. – Final

Dans le cas de la Fig. 18, si nous cliquons sur « Annuler », cela retire la date, comme ceci :

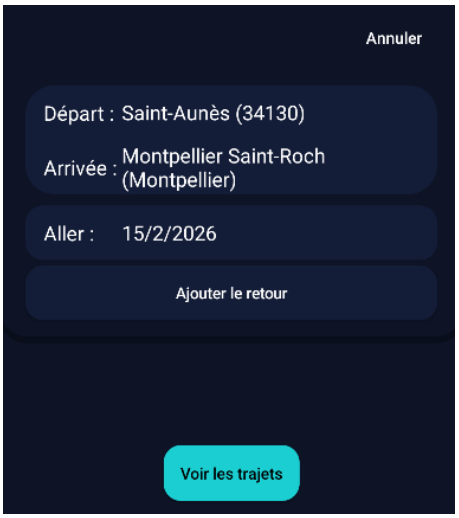

Fig. 20. – retrait de la date de retour

Un bouton « Annuler » situé en haut à droite permet de retourner à l'écran précédent. L'utilisation de la fonction `finish()` dans le code permet de fermer l'activité actuelle, ce qui restaure l'état de la première activité et conserve ainsi la saisie initiale de l'activité première.

3.1.2.1 Logique métier et gestion des données (API)

C'est au sein de cette seconde activité que réside l'essentiel de la logique métier. Pour interroger l'API sur les itinéraires, l'utilisation des noms de gares est insuffisante ; nous devons impérativement manipuler des identifiants techniques. Pour structurer ces échanges, j'ai créé le fichier `SncfData.kt` contenant les data classes nécessaires au parsing des réponses JSON (gérer pour Kotlin grâce a GSON):

SncfData.kt

```
  //...
data class Place(
    val id: String,
    val name: String
)
//...
data class Journey(
    val duration: Long, // Durée en secondes
    val departure_date_time: String, // Format: 20260213T183000
    val arrival_date_time: String
)
```

Code 4. – code de class

D'ailleurs, si l'utilisateur entre autrement une ville sans cliquer sur la ville de la liste, ce message s'installe (sinon il peut mettre un ville sans ID) :

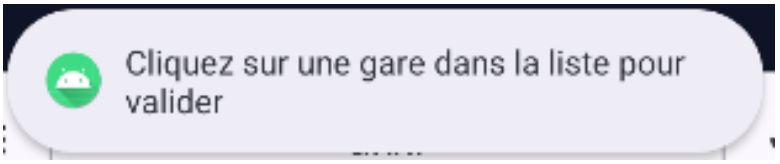

Fig. 21. – Cas si l'utilisateur rentre une ville sans cliquer sur la liste

Nous pouvons a présent cliquer sur « Voir les trajets » et avancer a la troisième et dernière activité.

3.1.3 Troisième activité

L'interface finale de cette application se présente comme suit :

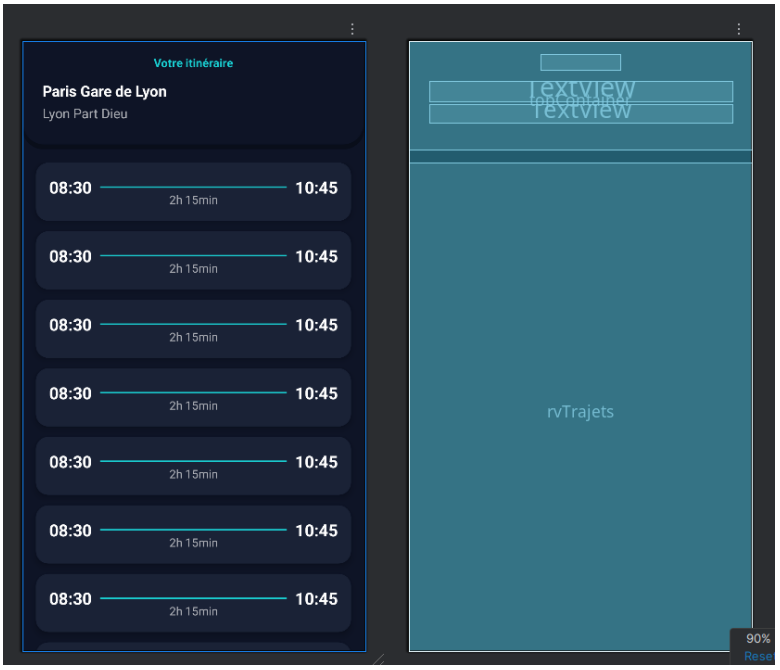

Fig. 22. – Squelette de la troisième activité

À la réception de l'Intent provenant de la seconde activité, l'application génère l'affichage suivant :

Fig. 23. – Résultat final

Cet écran permet de visualiser les 5 prochains itinéraires de la journée correspondant au trajet sélectionné par l'utilisateur.

3.2 Élément de Design

3.2.1 Drawable

Pour cette application, j'ai intégré plusieurs éléments graphiques personnalisés afin d'offrir une interface plus moderne et épurée, notamment via les ressources suivantes :

Fig. 24. – Corner 15dp

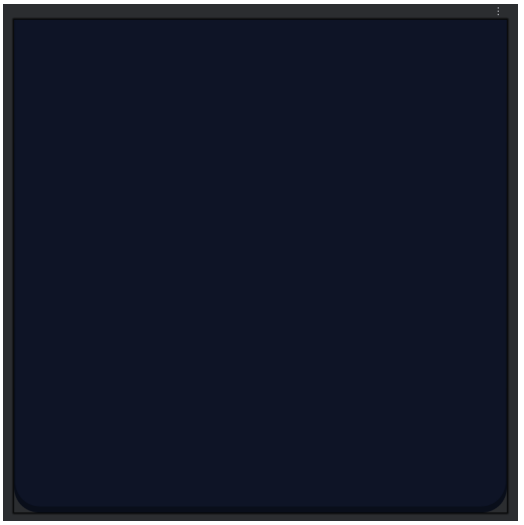

Fig. 25. – Corner uniquement en bas et ajout d'un shadow, effet « card »

Ces deux éléments, définis dans le dossier drawable, sont appliqués dans l'ensemble de l'application. Le premier permet d'arrondir les angles des vues à hauteur de 15dp, tandis que le second crée un effet de « card » grâce à un léger ombrage vers le bas.

3.2.2 Gestion de l'affichage dynamique : le RecyclerView

Comme illustré dans la Fig. 22, l'affichage des résultats repose sur un RecyclerView (rvTrajets). Ce composant est essentiel car il permet d'afficher une liste de trajets dont la longueur est indéfinie, tout en optimisant l'utilisation des ressources du système.

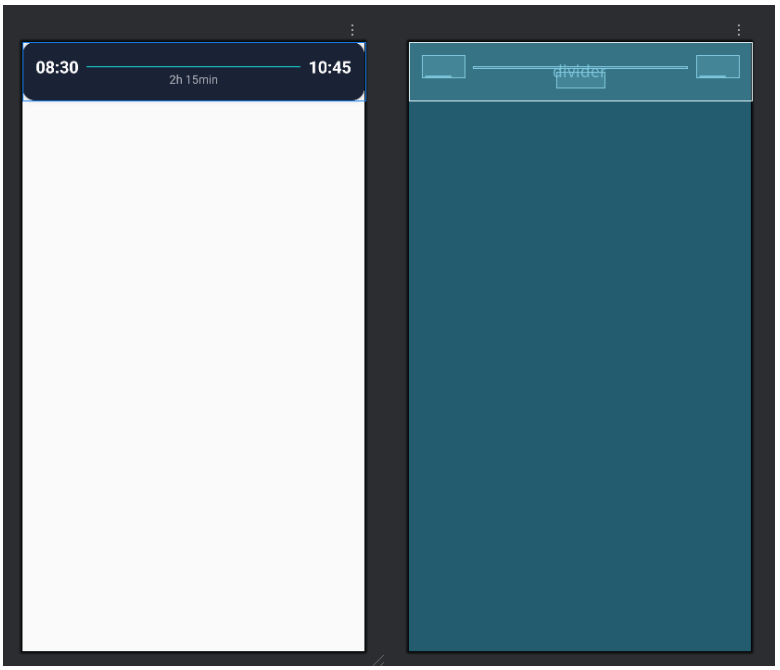

Fig. 26. – Squelette des items

L'utilisation d'un RecyclerView permet de recycler les vues qui sortent de l'écran pour les réutiliser avec de nouvelles données, ce qui limite considérablement la consommation de RAM. Ce fichier de mise en forme est stocké dans le dossier layout, ce qui permet de séparer la structure de la liste de celle des éléments qui la composent.

3.3 Architecture Global de l'application

Notre application peut se présenter en une image comme ceci :

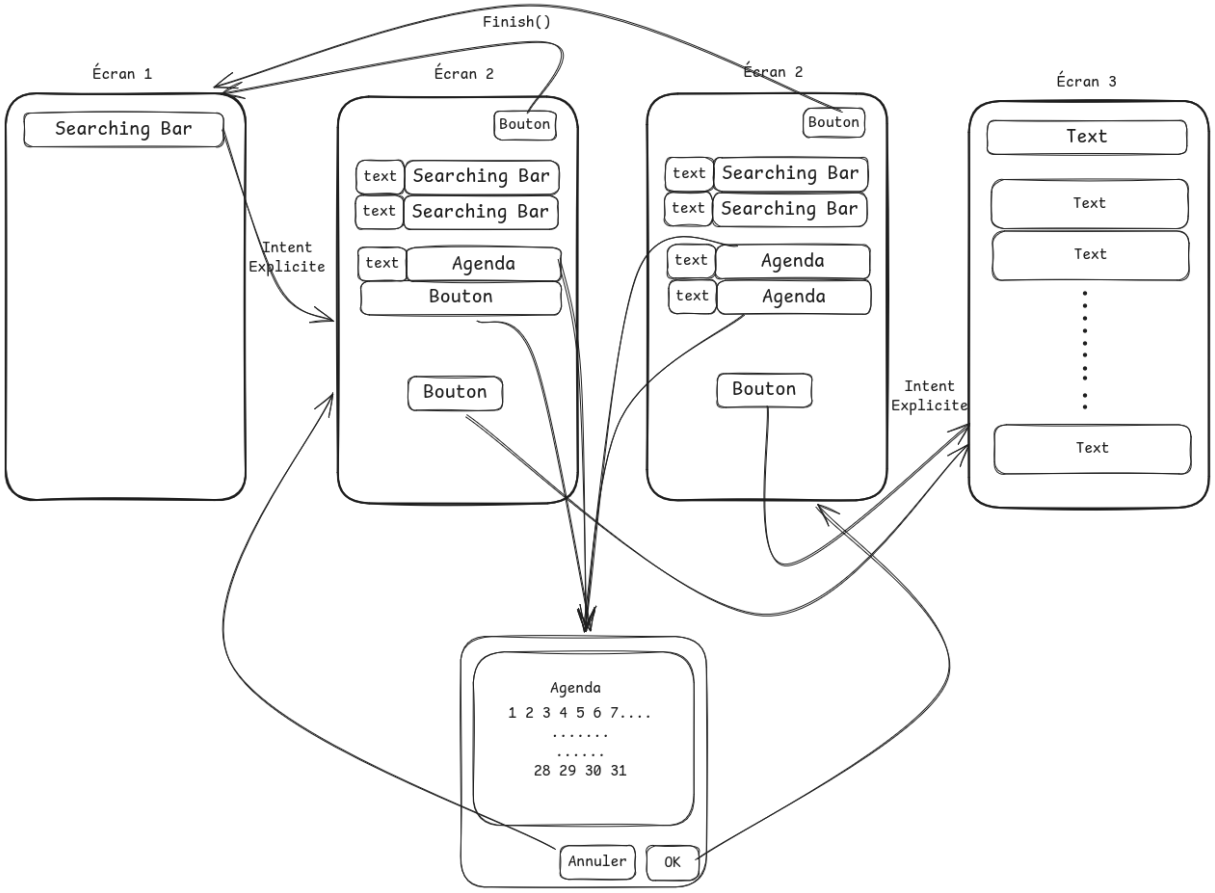

Fig. 27. – Schéma de l'application SNCF

3.4 Fonctionnalités - Application 2

Fonctionnalité	Statut
Visualiser des itinéraire	✓
Utilisation de l'API	✓
Internationalisation (Plusieurs langues)	✓ (Français et Anglais)
Navigation	✘ (Pas de bouton retour a partir de l'activité 3)
Vérification des champs	✓ (activité 2)
Intent Explicite	✓
Design	✓
Pouvoir voir un trajet spécifique aller-retour	✘ (Perspectives)
Choix de l'heure de l'itinéraire	✘ (Perspectives)

Tableau 2. – Tableau des fonctionnalités implémentées pour la deuxième application

Comme pour la première application, j'ai privilégié l'utilisation de la navigation native d'Android pour le retour depuis la troisième activité. Le système gérant efficacement la pile d'activités, l'ajout d'un bouton redondant n'a pas été jugé nécessaire pour ce prototype.

Par ailleurs, certaines fonctionnalités comme la sélection précise de l'horaire ou la gestion complète d'un trajet aller-retour n'ont pas été finalisées par manque de temps. Le bouton « Ajouter un retour » fait office de composant d'interface préparé pour une future implémentation de la logique métier. Quant au choix de l'heure, il constitue une évolution naturelle pour rendre l'outil parfaitement opérationnel.

4 Troisième application (Agenda)

4.1 Logique de l'application

Cette troisième réalisation consiste à concevoir un Agenda permettant l'insertion et la consultation d'événements tout au long de l'année. L'application est structurée autour d'une interface unique pour une navigation simplifiée.

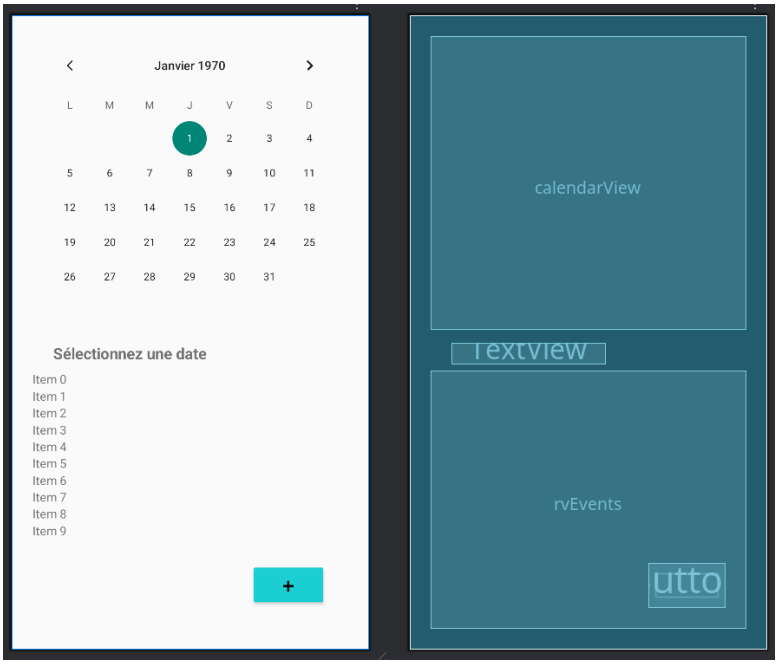

Fig. 28. – Squelette de la page

Pour ajouter un événement, l'utilisateur utilise le bouton « + », ce qui déclenche l'ouverture d'un AlertDialog :

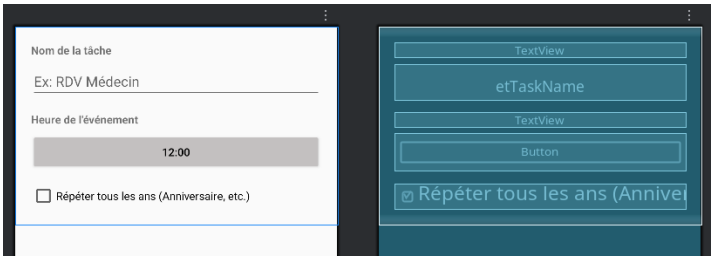

Fig. 29. – Squelette de l'AlertDialog

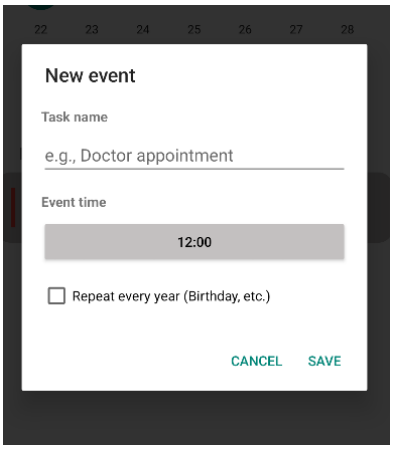

Fig. 30. – AlertDialog Réel

Cette interface permet plusieurs interactions : la saisie textuelle de l'activité, la sélection d'une heure précise via un TimePickerDialog, et l'activation d'une option de récurrence annuelle (Par exemple pour les anniversaire).

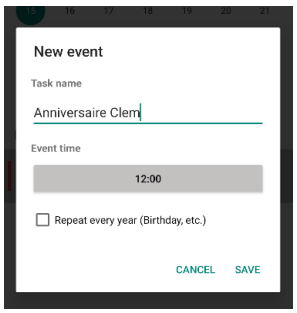

Fig. 31. – Ajout du texte

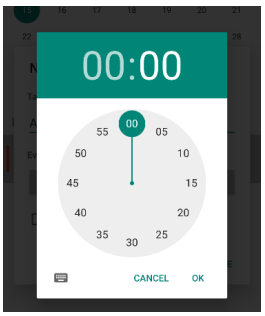

Fig. 32. – Ajout de l'heure

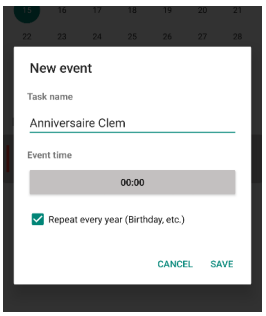

Fig. 33. – Cocher la case

L'utilisateur peut alors choisir d'annuler la saisie (ferme juste l'AlertDialog) ou de l'enregistrer. Les données sont ici sauvegardées de manière persistante. Cela signifie qu'en cas de fermeture ou de redémarrage de l'application, les événements sont automatiquement restaurés.

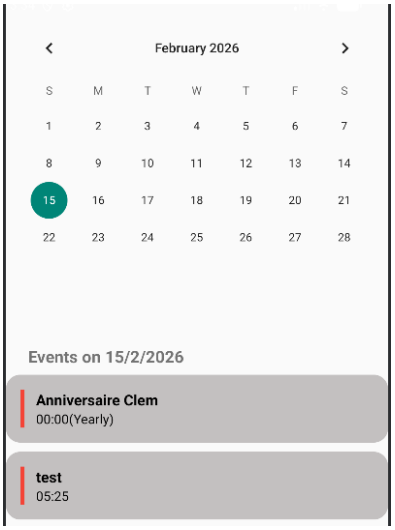

Fig. 34. – Résultat final

4.2 Architecture Global de l'application

Notre application peut se présenter en une image comme ceci :

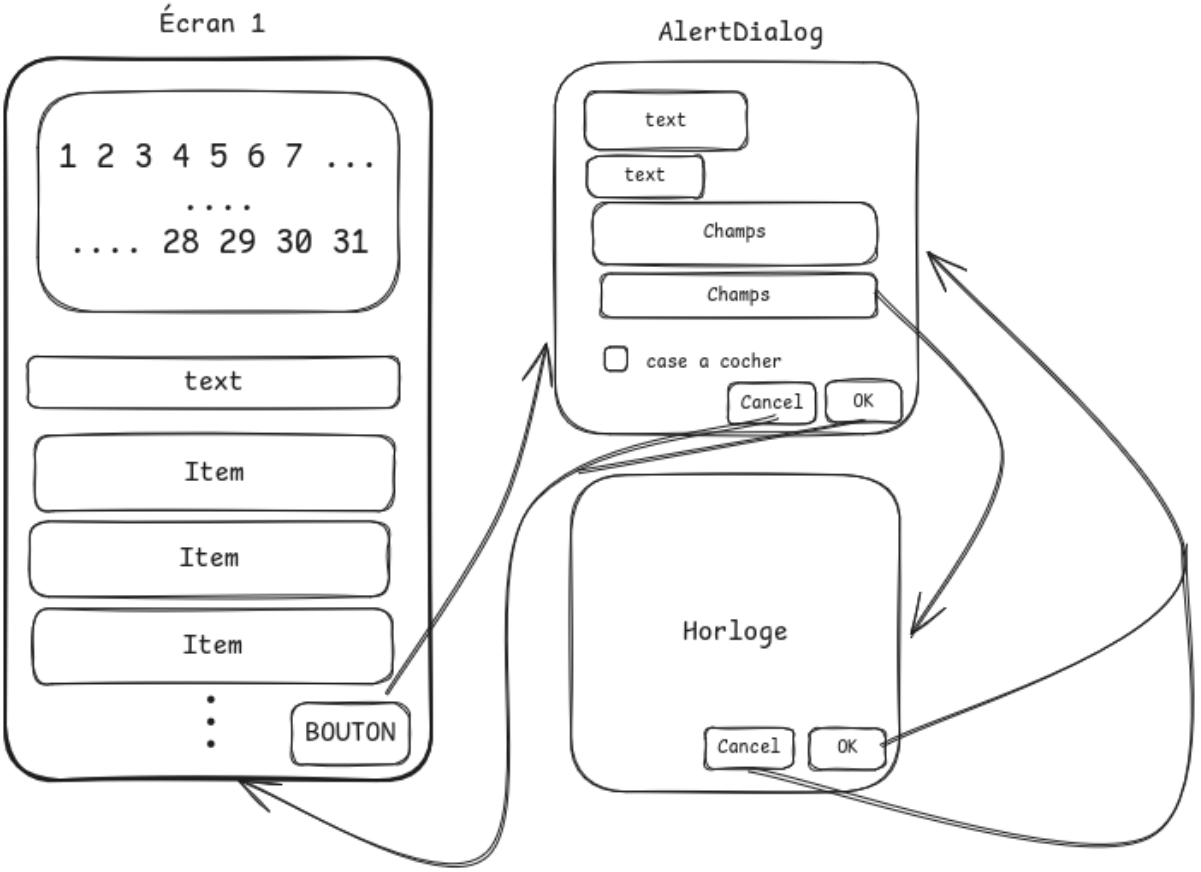

Fig. 35. – Schéma de l'application 3

4.3 Fonctionnalités - Application 3

Fonctionnalité	Statut
Sélection de date via CalendarView	✓
Ajout d'événements (Nom et Heure)	✓
Gestion de la récurrence annuelle	✓
Persistance des données	✓
Tri chronologique des tâches	✓
Internationalisation complète (FR / EN)	✓
Affichage dynamique par RecyclerView	✓
Édition ou suppression d'un événement	✘ (Perspective)
Notifications / Rappels	✘

Tableau 3. – Tableau des fonctionnalités implémentées pour la troisième application