TP 3+ : Master/Detail

Objectif

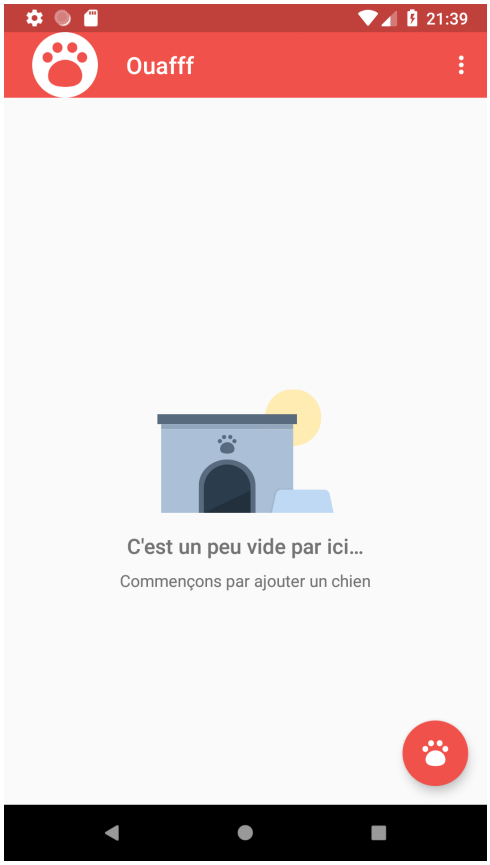
Le but de ces prochains TP (6h) est de développer une petite application de type Master/Detail sous Android en partant de zéro, afin de bien comprendre comment utiliser certains éléments plutôt indispensables du framework (`Fragment`, `RecyclerView`, `ViewPager2`, …). Évidemment, l'application devra être déployable aussi bien sur téléphone que sur tablette, donc vous n'oublierez pas de prendre cela en considération dans vos développements.

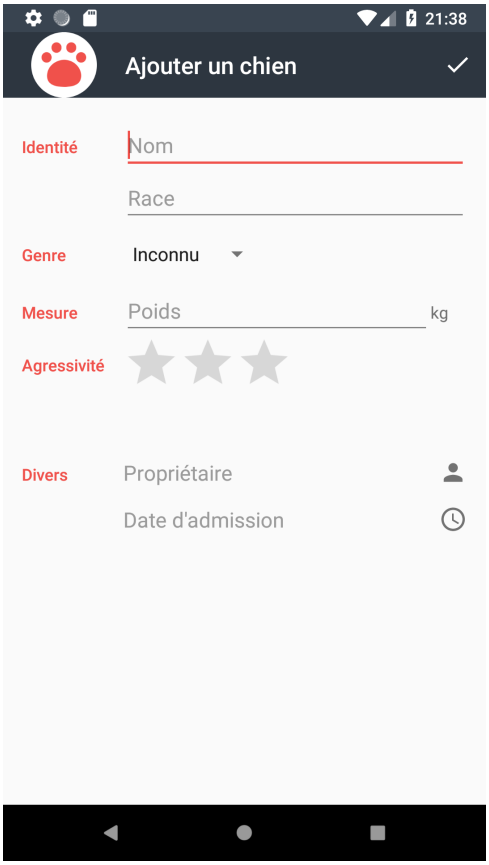
Contexte

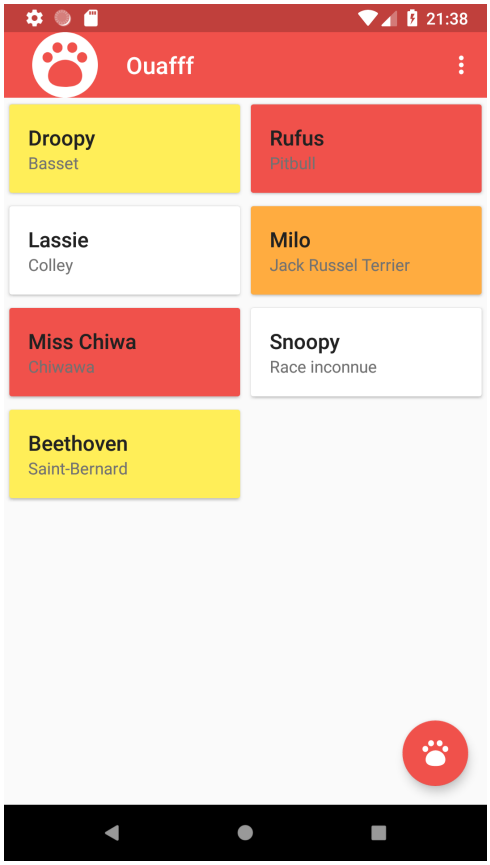
Vous travaillez en tant que bénévole dans un chenil et êtes en contact avec beaucoup de chiens dont vous avez du mal à vous rappeler le nom. De plus, vous devez vous méfier de certains d'entre eux car ils peuvent être assez agressifs. Comme vous avez de bonnes bases en informatique, vous décidez de mettre au point « *Ouafff* », une petite application mobile permettant de conserver des informations relatives à ces chiens et vous aider dans votre tâche.

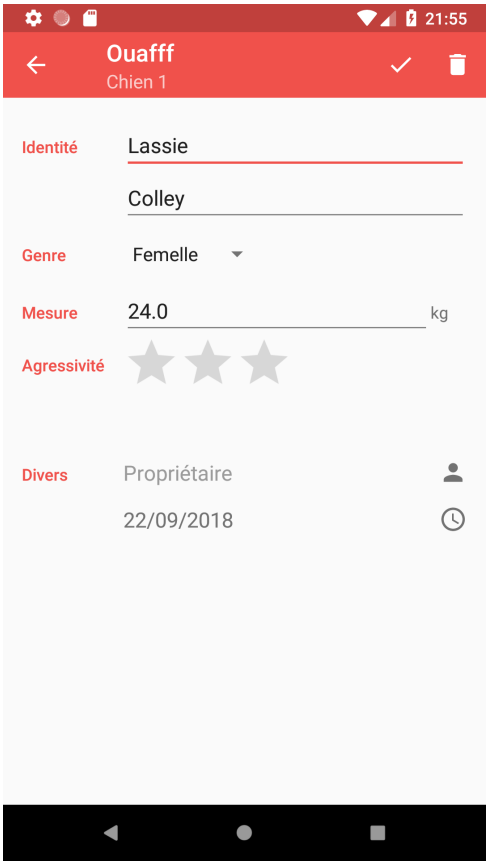
Travail demandé

Vous devez mettre au point une application Android de type Master/Detail dont les éléments devront ressembler aux captures ci-dessous (ou au moins s'en approcher) :



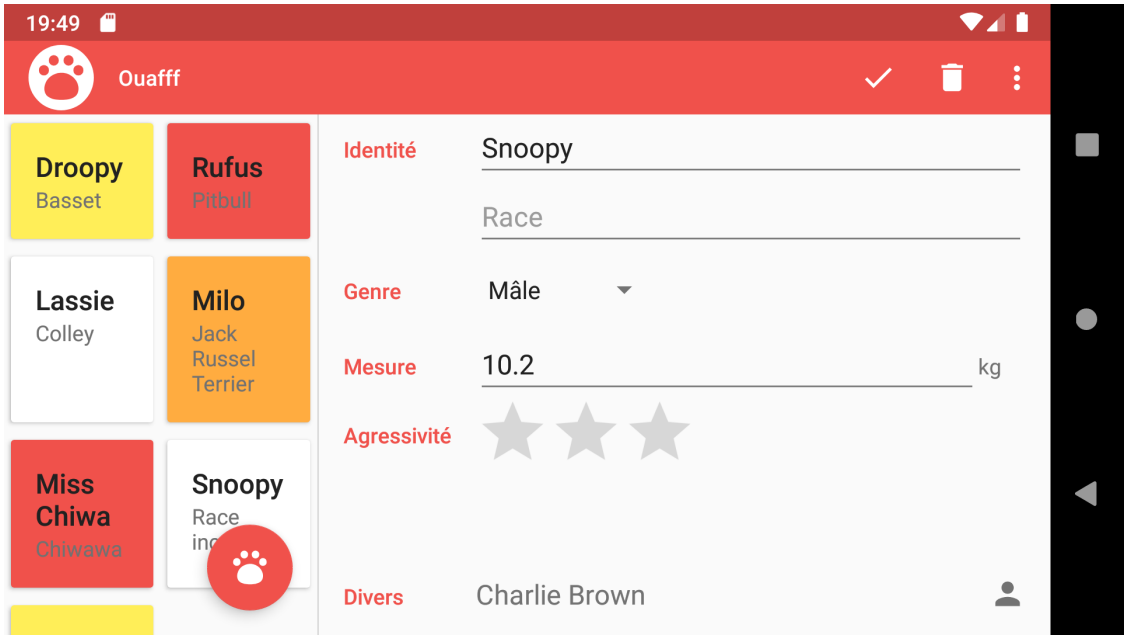






Instructions

Pour ce premier TP, vous vous occuperez de mettre en place le visuel de l'application ainsi que la dynamique du Master/Detail. Il est **obligatoire** d'utiliser des fragments. En mode portrait, un fragment occupera tout l'écran et vous utiliserez le mode `paysage` du téléphone pour mettre les 2 fragments côte à côte.



Les fonctionnalités à mettre en place devront respecter les contraintes suivantes :

1. Le détail permettra d'entrer les informations suivantes pour un chien :
   — son nom ;
   — sa race (optionnelle) ;
   — son genre (à choisir dans un `Spinner` parmi : Male, Femelle, Inconnu) ;
   — son poids (en kg) ;
   — son degré d'agressivité (de 0 à 3 étoiles) ;
   — son propriétaire (optionnel — à choisir parmi les contact du téléphone) ;
   — la date d'entrée du chien au chenil (optionnelle — à choisir dans un dialog contenant un `DatePicker`)
2. Le master devra contenir une `RecyclerView` gérant une grille de deux colonnes avec des petites cartes résumant les infos du chien :
   — le nom, en noir, assez gros ;
   — la race, en gris, plus petit (ou bien la mention « Race inconnue » si l'information n'avait pas été précisée) ;
   — la couleur de fond d'une carte sera définie en fonction de l'agressivité du chien (blanc, jaune, orange ou rouge pour des agressivités de 0, 1, 2 ou 3 étoiles respectivement).
3. Un clic sur un item de la liste affichera le détail du chien.
4. Une navigation entre les éléments du détail sera possible à l'aide d'un swipe (vous utiliserez pour cela un `ViewPager2`).
5. Il devra être possible d'ajouter un nouveau chien (grâce à un Floating Action Button)
6. L'app bar de l'application devra permettre une « Up navigation » depuis le détail.

Vous commencerez par mettre au point le fragment d'édition d'un chien (le Detail) en vous assurant que tout fonctionne correctement quand vous l'intégrez à une activité. Ne perdez pas trop de temps sur le design (utilisez un `ConstraintLayout` et une `Barrier` pour aligner les éléments). Ne vous occupez pas tout de suite des interactions « Propriétaire » et « Date d'admission », concentrez-vous sur la mise en place du Master/Detail.

Ainsi, vous créerez ensuite le fragment contenant la liste de chiens (le Master). Vide, il affichera l'image du chenil, puis dès qu'un chien sera inséré, il affichera la `RecyclerView`. Mettez cette dernière en place et codez l'adapter permettant d'obtenir un resultat semblable à l'exemple (une `CardView` pour les items fera l'affaire). Vous utiliserez un stub pour tester votre application — nous nous occuperons de la persistance des données dans un autre TP.

Vous définirez l'activité contenant ce fragment comme activité de démarrage de l'application (mettre à jour le `Manifest`) puis vous finirez en mettant en place la dynamique du clic sur un item de la `RecyclerView` pour que celle-ci affiche le détail d'un chien. Attention à ne pas dupliquer inutilement du code entre les versions portrait et paysage !

Quand ceci sera fonctionnel, vous pourrez ajouter l'activité contenant le détail d'un chien qui utilisera le `ViewPager2` pour gérer le swipe.

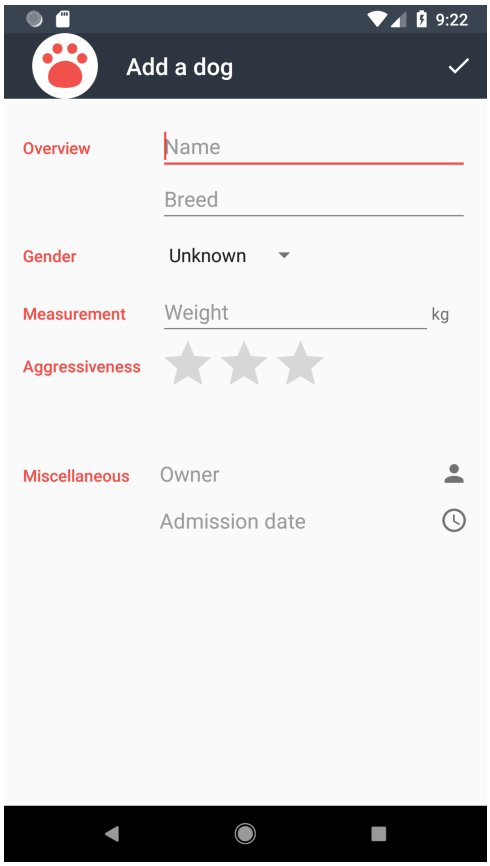
Vous pourrez aussi vous attaquer aux interactions « Propriétaire » et « Date d'admission ».
En cliquant sur la vue (icone ou champ texte) « Propriétaire » vous lancerez une intention implicite permettant de choisir un contact du téléphone. Seul son nom complet sera conservé et affiché dans le champ.
En cliquant sur la vue (icone ou champ texte) « Date d'admission » vous afficherez une boîte de dialogue avec un calendrier permettant de saisir une date (utilisez un `DialogFragment` sinon le dialog ne sera pas conservé lors des rotations ; vous mettrez uniquement un `DatePicker` à l'intérieur). Évidemment le format d'affichage dépendera de la langue configurée, donc pensez à choisir une donnée métier en conséquence.

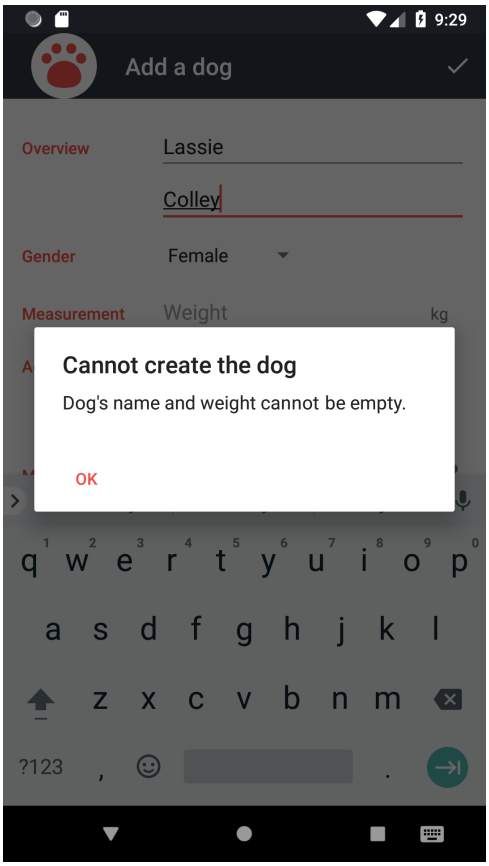
L'icône en SVG de l'application ainsi que d'autres ressources utilisées dans l'application sont disponibles dans le même dossier que cet énoncé (fichier ressourcesOuafff.zip).

i18n

L'application devra être disponible par défaut en langue anglaise, mais traduite en français sur les téléphones configurés ainsi. Voici une capture de la version anglaise qui vous permettra d'avoir le vocabulaire à utiliser.

J'en profite pour préciser qu'une omission des informations requises à l'ajout d'un chien amènera à l'affichage d'un dialog d'erreur. L'utilisateur devra alors remplir les champs obligatoires avant de pouvoir ajouter le chien.





Persistance

Pour cette dernière partie du TP, codez la persistance de vos données en utilisant *Room*.

Commencez par définir votre (ou vos) entité(s) et réfléchissez aux différentes méthodes dont vous aurez besoin dans votre application. Ceci vous permettra de mettre au point votre DAO. Une fois ceci fait, codez votre *Database* en faisant de cette dernière un singleton.

Attention, *Room* utilise les annotations des différentes classes pour avoir les informations nécessaires à la génération du code. N'oubliez pas de spécifier dans le `build.gradle` de votre module d'utiliser `kapt`, l'« annotation processor » de Kotlin, et non pas celui de Java. Ceci nécessitera d'utiliser le plugin `kotlin-kapt` pour Gradle.

La persistance en elle-même ne devrait pas poser de problème (c'est un cas d'école et tous les éléments à utiliser se trouvent dans le cours). Par exemple si vous avez utilisé une énumération pour le genre, il vous faudra un *Converter* au niveau de la base de données. Pareil pour la date, ces éléments n'ont pas de type direct approprié au niveau SQLite.

Un point plus ennuyant est celui de l'accès à la base de données depuis le thread principal. Essayez normalement puis voyez le problème. Un pis-aller est d'utiliser la méthode `allowMainThreadQueries()` du builder de votre base de données. Pour l'éviter vous pourrez utiliser les composants du package `java.util.concurrent` de Java. Notamment l'`ExecutorService` pourra vous aider ici. Vous essayerez de conserver une architecture propre en intégrant une couche d'abstraction supplémentaire (communément appelé le `Repository` dans Jetpack) dans laquelle vous placerez les méthodes d'accès à la persistance.

Nous verrons dans la deuxième partie du module comment les coroutines en Kotlin peuvent avantageusement remplacer cette dernière partie.

Quelques indices pour les retardataires de la première partie

Menus

1. Créez des nouvelles ressources décrivant le menu des fragments. Par exemple, pour le master, un menu permettant d'ajouter un nouveau chien à la liste que vous mettrez dans le fichier `fragment_list_dogs.xml` du dossier de ressources `menu`.
2. Si vous utilisez la Fragment API pour les menus (récemment DEPRECATED)
   — Redéfinissez les méthodes `onCreateOptionsMenu(…)` et `onOptionsItemSelected(…)` directement au sein des fragments pour rendre leur menu fonctionnel.
   — N'oubliez pas de spécifier que votre fragment utilise un menu grâce à un appel de la méthode `setHasOptionsMenu(…)`.
3. Si vous utilisez la nouvelle MenuProvider API (`androidx.fragment` $\geq$ v1.5.0)
   — L'activité implémente l'interface `MenuHost` et utilise des `MenuProvider` pour peupler le contenu de l'app bar.
   — Enregistrez un `MenuProvider` au sein de vos fragments grâce à la méthode `addMenuProvider(…)`.
   — Vous y implémenterez les méthodes `onCreateMenu(…)` et `onMenuItemSelected(…)` pour rendre fonctionnel vos menus.
   — N'oubliez pas de passer le fragment en deuxième paramètre de `addMenuProvider(…)` pour rendre le tout actif.

Changement de la date

1. Pour gérer le changement de date d'admission au sein de votre `DogFragment` il est conseillé d'utiliser un `DialogFragment`. À l'inverse d'un `Dialog` simple, celui-ci est restauré en cas de changement de configuration.
2. Ajoutez donc une classe `DatePickerFragment` qui dérive d'`AppCompatDialogFragment`. Le visuel de celui-ci ne sera composé que d'un unique `DatePicker`.
3. Redéfinissez la méthode `onCreateDialog(…)` du `DatePickerFragment` afin de créer la vue. Vous utiliserez un `AlertDialog.Builer` pour construire le dialog que vous retournerez.
4. Gérez l'échange des données (c.-à-d. la date) entre les deux fragments grâce aux Arguments (pour la communication `DogFragment` $\rightarrow$ `DatePickerFragment`) avec comme d'habitude la classique factory `newInstance(…)`. Pour la communication `DatePickerFragment` $\rightarrow$ `DogFragment` regardez du côté des méthodes `setTargetFragment(…)` et `onActivityResult(…)` (qui est le target fragment ?).