

# **La persistance de données Android**



**neopixl.**

A SMILE GROUP COMPANY

**Welcome on board**



# Sommaire

01

Base SQL

02

Persistence d'état

03

Shared Preferences

04

Preferences Menu

05

Fichiers

06

Room

07

ObjectBox

**1**

# Base SQL



# SQL

Le langage SQL (Structured Query Language) est un langage informatique utilisé pour exploiter des bases de données. Il permet de façon générale la définition, la manipulation et le contrôle de sécurité de données.

Dans la pratique, le langage SQL est utilisé pour créer des tables, ajouter des enregistrements sous forme de lignes, interroger une base de données, la mettre à jour, ou encore gérer les droits d'utilisateurs de cette base de données. Il est bien supporté par la très grande majorité des systèmes de gestion de base de données (SGBD). Créé au début des années 1970 par Donald D. Chamberlin et Raymond F. Boyce, tous deux chez IBM, le langage SQL est aujourd'hui reconnu comme une norme internationale.

Quelques moteur de base de données connus : Oracle, Microsoft SQL Server, MYSQL, SQLite, ...







# SQL

Une ligne représente un enregistrement.  
Ici, une ce sera une personne.

Ligne

Nom	Age	Ville



# SQL

Une colonne représente un élément précis, pour tous les enregistrements d'une table.  
Ici tous les ages.

## Colonne

Nom	Age	Ville





Une cellule représente une information précise d'un enregistrement.  
Ici le Nom d'une personne.

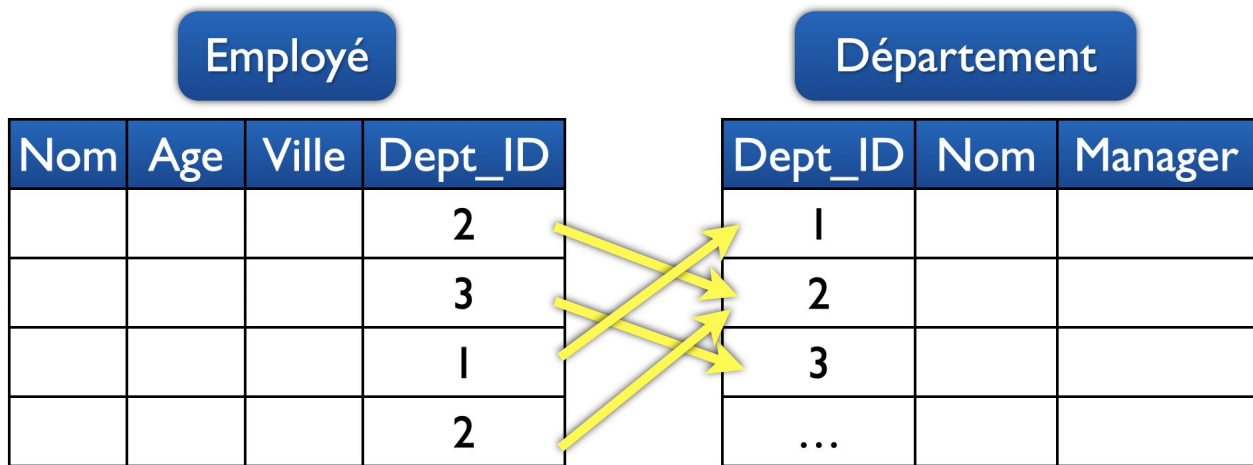
# Champ ou cellule

Nom	Age	Ville



# SQL

Plusieurs tables présentes dans une base de données relationnelle  
Relation entre plusieurs tables via une colonne "commune" (ex : identifiant)

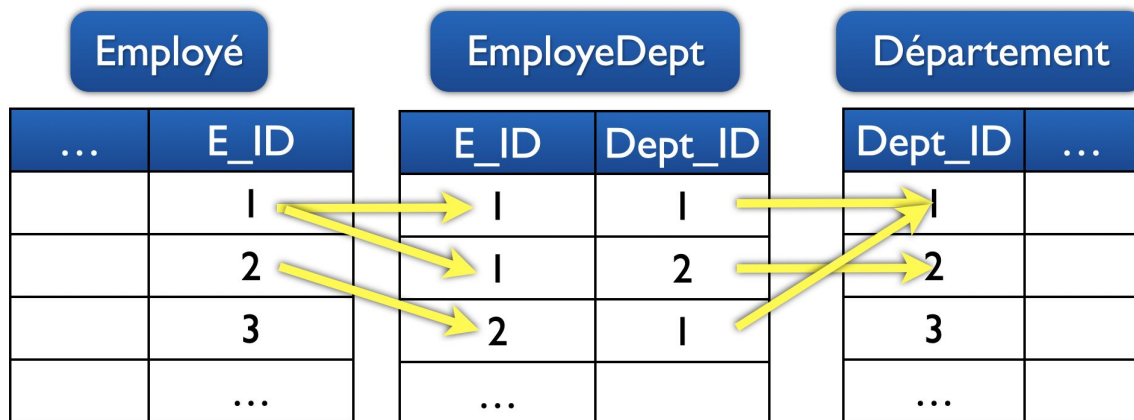


# SQL

Pour les relations 1 à n et n à 1, deux tables suffisent avec une clef primaire et une autre clef étrangère (ex : relation table employé <-> département).

Pour les relations n à n, il faut créer une table intermédiaire.

Une table intermédiaire est une table contenant une association clef primaire / clef étrangère





# SQL



## Les contraintes

- Non Nulle: la valeur doit être définie, et ne peut être nulle.
- Unique: la valeur doit être unique dans la table.
- Clef primaire: la valeur doit être unique dans la table, et est l'identifiant d'un enregistrement.
- Clef étrangère: la valeur référence une clé primaire.

La commande sqlite3 disponible avec le SDK Android ou Xcode.

Pour créer ou ouvrir un fichier : `sqlite3 nom_du_fichier` (ex : `sqlite3 movies.db`)

- `.tables` : affiche la liste des tables
- `.schema NOM_DE_LA_TABLE` : affiche le schéma de la table
- `.exit` : Quitter l'invite de commande sqlite3
- `.help` : affiche l'aide



# SQL

Les clefs primaires :

clef unique

peut être composée à partir d'autres clefs  
de préférence générée (non nulle)

Les clefs étrangères :

référence à partir d'une clef primaire (liaison)

Liste des attributs possible :

- INTEGER
- SMALLINT
- BIGINT
- CHARACTER(20)
- VARCHAR(255)
- BLOB
- REAL
- DOUBLE
- FLOAT
- DECIMAL(10,5)
- BOOLEAN
- DATE
- DATETIME



# SQL

Liste des opérations possible sur une Table :

- SELECT FROM (WHERE) : affiche les données d'une table
- CREATE TABLE : crée une nouvelle table
- INSERT INTO VALUES : ajoute des données à une table
- DELETE FROM : supprime des données d'une table
- UPDATE : met à jours des données d'une table
- ALTER TABLE : modifie le schéma d'une table

# SQL

Recherche dans une table

SELECT table1.champ(s) FROM table1, ...

Préciser des critères : WHERE champ (condition) valeur AND/OR ...

Ordonner les résultats : ORDER BY champ (DESC/ASC par défaut )

Regrouper les résultats : GROUP BY champ

SELECT \* FROM employe WHERE Dept\_ID = 1;

## Employé

Nom	Age	Ville	Dept ID
Jean	42	Liège	1
Paul	28	Liège	2
Alex	35	Mons	1
Albert	32	Liège	Null

Nom	Age	Ville	Dept_ID
Jean	42	Liège	1
Alex	35	Mons	1



# SQL

Récupérer le nombre d'éléments dans une table

```
SELECT COUNT (nom_table.nom_champ) FROM nom_table;
```

ex : `SELECT COUNT(*) FROM employe;`

Créer une nouvelle table

```
CREATE TABLE table_nom ( champ1 type(taille) contrainte, ... );
```

```
CREATE TABLE employe (id INTEGER PRIMARY KEY AUTOINCREMENT,nom VARCHAR(25) NOT NULL,age INTEGER NULL,ville VARCHAR(25) NULL,dept_id INTEGER, FOREIGN KEY(dept_id) REFERENCES DEPARTMENT(id));
```

Employé				
id	Nom	Age	Ville	Dept_ID





# SQL

Ajouter des enregistrements dans une table

```
INSERT INTO table_nom (champ1, ...) VALUES (valeur1, ...),(valeur11,...), ... ;
```

```
INSERT INTO employe(Nom, Age, Ville, Dept_ID) VALUES ("Jean", 42, "Liège",2);
```

Employé			
Nom	Age	Ville	Dept_ID
Jean	42	Liège	2



# SQL

Supprimer des enregistrements dans une table  
`DELETE FROM table_nom WHERE (condition);`

`DELETE FROM employe WHERE Dept_id=2;`

Employé			
Nom	Age	Ville	Dept_ID
Jean	42	Liège	2



# SQL

Modifier des enregistrements d'une table

```
UPDATE table_nom SET champ1 = valeur1 WHERE champ2 = valeur 2;
```

```
UPDATE employe SET dept_id = 1 WHERE nom = "Jean";
```

Employé			
Nom	Age	Ville	Dept_ID
Jean	42	Liège	2 => 1



# SQL

Modifier une table existante

ALTER TABLE table\_nom ADD/DROP champ

ALTER TABLE employe DROP ville; (non supporté via SQLITE)

Employé			
Nom	Age	Ville	Dept_ID



# SQL

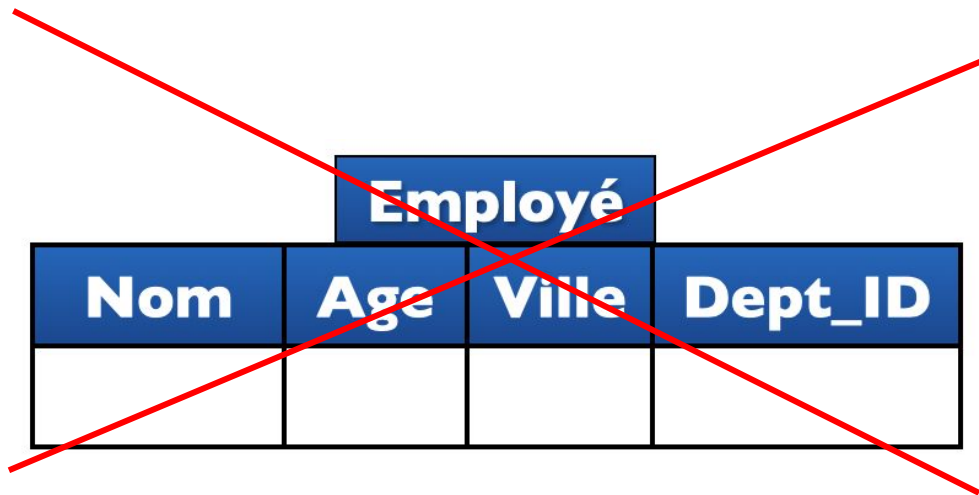
ALTER TABLE employe ADD marie INTEGER;

Employé				
Nom	Age	Ville	Dept_ID	Marié



# SQL

DROP TABLE Employé;



Employé			
Nom	Age	Ville	Dept_ID

Union simple = rassemblement de deux tables via un id en commun :  
`SELECT * FROM table1, table2 WHERE table1.foreign_key=table2.primary_key;`

CROSS JOIN : produit cartésien des données  
`SELECT * FROM table1 CROSS JOIN table2;`

Employé			
Nom	Age	Ville	Dept ID
Jean	42	Liège	1
Paul	28	Liège	2
Alex	35	Mons	1
Albert	32	Liège	Null

Département		
Dept ID	Nom	Manager
1	Java	Jean
2	Mobile	Paul
3	Microsoft	NULL

Nom	Age	Ville	Dept_ID	Nom	Manager	Dept_ID
Jean	42	Liège	1	Java	Jean	1
Paul	28	Liège	2	Java	Jean	1
Alex	35	Mons	1	Java	Jean	1
Jean	42	Liège	1	Mobile	Paul	2
...	...	...	...	...	...	...



# SQL

INNER JOIN : combine les valeurs communes de deux tables

```
SELECT * FROM table1 INNER JOIN table2 ON table1.champ1 = table2.champ1;
```

**Employé**

Nom	Age	Ville	Dept ID
Jean	42	Liège	1
Paul	28	Liège	2
Alex	35	Mons	1
Albert	32	Liège	Null

**Département**

Dept ID	Nom	Manager
1	Java	Jean
2	Mobile	Paul
3	Microsoft	NULL

Nom	Age	Ville	Dept_ID	Nom	Manager
Jean	42	Liège	1	Java	Jean
Paul	28	Liège	2	Mobile	Paul
Alex	35	Mons	1	Java	Jean





# SQL

LEFT OUTER JOIN : regroupe deux tables grâce à une condition mais force l'extraction de toutes les valeurs de la table de gauche

```
SELECT * FROM table1 LEFT OUTER JOIN table2 ON table1.champ1 = table2.champ1;
```

Employé			
Nom	Age	Ville	Dept ID
Jean	42	Liège	1
Paul	28	Liège	2
Alex	35	Mons	1
Albert	32	Liège	Null

Département		
Dept ID	Nom	Manager
1	Java	Jean
2	Mobile	Paul
3	Microsoft	NULL

Nom	Age	Ville	Dept_ID	Nom	Manager	Dept_ID
Jean	42	Liège	1	Java	Jean	1
Paul	28	Liège	2	Mobile	Paul	2
Alex	35	Mons	1	Java	Jean	1
Albert	32	Liège	NULL	NULL	NULL	NULL



# SQL

RIGHT OUTER JOIN : similaire au LEFT OUTER JOIN mais ce sont toutes les valeurs de la table de droite qui seront extraites  
SELECT \* FROM table1 RIGHT OUTER JOIN table2 ON table1.champ1 = table2.champ1

**Employé**

Nom	Age	Ville	Dept ID
Jean	42	Liège	1
Paul	28	Liège	2
Alex	35	Mons	1
Albert	32	Liège	Null

**Département**

Dept ID	Nom	Manager
1	Java	Jean
2	Mobile	Paul
3	Microsoft	NULL

Nom	Age	Ville	Dept_ID	Nom	Manager	Dept_ID
Jean	42	Liège	1	Java	Jean	1
Paul	28	Liège	2	Mobile	Paul	2
Alex	35	Mons	1	Java	Jean	1
NULL	NULL	NULL	NULL	Microsoft	NULL	3

FULL OUTER JOIN : Combine les résultats d'un LEFT OUTER JOIN et d'un RIGHT OUTER JOIN  
 SELECT \* FROM table1 FULL OUTER JOIN table2 ON table1.champ1 = table2.champ;

Employé			
Nom	Age	Ville	Dept ID
Jean	42	Liège	1
Paul	28	Liège	2
Alex	35	Mons	1
Albert	32	Liège	Null

Département		
Dept ID	Nom	Manager
1	Java	Jean
2	Mobile	Paul
3	Microsoft	NULL

Nom	Age	Ville	Dept ID	Nom	Manager	Dept ID
Jean	42	Liège	1	Java	Jean	1
Paul	28	Liège	2	Mobile	Paul	2
Alex	35	Mons	1	Java	Jean	1
Albert	32	Liège	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	Microsoft	NULL	3



# Les vues

agrégat d'une ou plusieurs tables (correspond à une "requête stockée")  
en général virtuel (les données ne sont pas stockées dans la DB)  
même requêtes que sur une table



# Les procédures stockées

Lorsqu'une requête est soumise au serveur, celui-ci doit analyser la syntaxe, puis l'interpréter et l'exécuter.

Opérations lourdes si la même requête doit être exécutée plusieurs fois de suite

Solution : la stocker sur le serveur

procédure stockée = requête analysée, interprétée, puis stockée sous forme compilée



# Les déclencheurs (triggers)

déclencheur associé à la mise à jour, l'insertion ou la suppression d'une donnée  
associé à une procédure stockée  
automatiser certains traitements pour répercuter des changements sur d'autres tables

ex1

# Exercise



# Exercice 1

Créez un nouveau fichier

Nom: Exo1

Placez le dans un dossier TECHNIFUTUR-AND14-EXO1







# Exercice 1

- 1 - Lister les tables dans le fichier sqlite grâce à la commande sqlite3.
- 2 - Récupérer le nombre de jeux vidéo contenus dans la base de données
- 3 - Récupérer la liste des meilleurs jeux vidéo (dont la note est supérieur à 90).
- 4 - Récupérer la liste des jeux vidéo dont la note est comprise entre 70 et 90.
- 5 - Récupérer la liste des jeux vidéo disponible sur Xbox 360.
- 6 - Récupérer la liste des jeux vidéo dont la note est supérieure à 80 et disponible sur Xbox 360.
- 7 - Récupérer la liste des jeux (nom du jeu + console) dont la note est supérieure à 85.
- 8 - Insérer les jeux suivants dans la base de données :
  - Crisis 3 (XBox 360), note : 72
  - DmC Devil May Cry (PS3), note : 80
- 9 - Ajouter une table Vente, qui contiendra un nom de client et un jeu vidéo.
- 10 - Insérer 5 Ventes de jeux, dont au moins deux fois le même.
- 11 - Pour Chaque jeu, affichez le nombre de vente.

2

# Persistence d'état



# Persistence d'état

Android permet d'enregistrer l'état des activités afin que l'utilisateur retrouve l'interface identique entre deux sessions

Deux mécanismes de sauvegarde de l'état d'une activité :

- gestion de la persistance grâce aux méthodes de cycle de vie de l'activité
- gestion manuelle grâce à des classes de persistance fournies par une API

# Persistence d'état

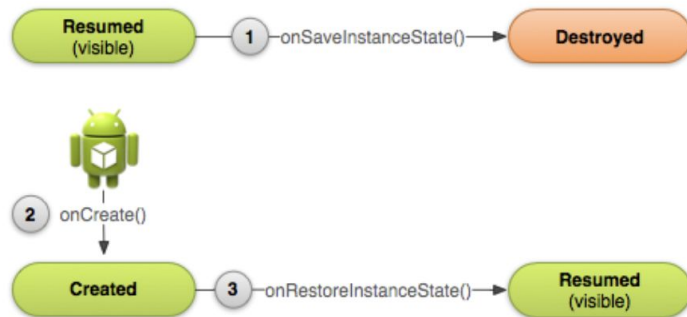
- gestion de la persistance grâce aux méthodes de cycle de vie de l'activité

la méthode `onSaveInstanceState` est appelée lorsque Android a besoin de libérer des ressources et de détruire l'activité. Attention l'appui sur Retour n'enregistre pas l'activité car celle-ci ne sera pas mise dans la pile de l'historique des activités. Objet de type `Bundle` utilisé pour stocker les données à sauvegarder (passé en paramètre aux méthodes `onCreate` et `onRestoreInstanceState`).

Par défaut les méthodes `onSaveInstanceState`, `onRestoreInstanceState` et `onCreate` enregistrent les valeurs de toutes les vues possédant un id puis les restaurent.

la méthode `onSaveInstanceState` enregistre l'état des vues "identifiées" dans un objet `Bundle`.

Cet objet `bundle` est alors sauvegardé et sera ensuite passé aux méthodes `onCreate` et `onRestoreInstanceState`



# Persistence d'état

Le mécanisme par défaut convient à la plupart des cas d'utilisation.

Cependant le développeur peut souhaiter enregistrer ses propres valeurs.

Pour une activité, il faut pour cela surcharger la méthode **onSaveInstanceState** pour enregistrer des valeurs supplémentaires

Pour restaurer les valeurs supplémentaires il faut surcharger les méthodes **onCreate** et **onRestoreInstanceState**

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    if (savedInstanceState != null && savedInstanceState.containsKey(GAME_STATE_KEY)) {
        textView.text = savedInstanceState?.getString(GAME_STATE_KEY)
    }
}

// invoked when the activity may be temporarily destroyed, save the instance state here
override fun onSaveInstanceState(outState: Bundle) {
    outState?.run {
        putString(GAME_STATE_KEY, gameState)
    }
    // call superclass to save any view hierarchy
    super.onSaveInstanceState(outState)
}

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    textView.text = savedInstanceState?.getString(GAME_STATE_KEY)
}
```

# Persistence d'état

Pour un fragment, Les mêmes méthode existent, mais en plus, il y a les **arguments**.

Les arguments sont sauvé en mémoire, donc conviennent pour des petit objet. Ils sont rapide à récupérer. Si les données sont volumineuse, préférez utiliser le **savedInstanceState**.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    arguments?.let {
        param1 = it.getString(ARG_PARAM1)
    }
}

companion object {
    @JvmStatic
    fun newInstance(param1: String) =
        BlankFragment().apply {
            arguments = Bundle().apply {
                putString(ARG_PARAM1, param1)
            }
        }
}
```

3

# Shared Preferences

# Shared Preferences

Possibilité d'enregistrer d'autres valeurs propres à toute l'application (ex : paramètres de l'utilisateur, configuration de l'application, ...)

Données accessibles à travers tous les composants de l'application

Mécanisme appelé préférences partagées

Permet la persistance des propriétés sous la forme d'une paire clef/valeur

Données stockables : boolean, int, long, float et String

Pour récupérer les préférences : `getPreferences(int)` (depuis l'activité courante)

Renvoie une instance d'une classe `SharedPreferences`

Possibilité de créer plusieurs ensembles de préférences partagées identifiés par un nom unique grâce à la méthode `getSharedPreferences(String, int)`

Lorsque vous nommez vos fichiers de préférences partagés, vous devez utiliser un nom identifiable de manière unique pour votre application. Un moyen simple de le faire est de préfixer le nom du fichier avec l'ID de votre application. Par exemple : "com.example.myapp.PREFERENCE\_FILE\_KEY"

```
val sharedPref = this?.getSharedPreferences(getString(R.string.preference_file_key),  
Context.MODE_PRIVATE)
```





# Shared Preferences

Méthodes disponibles dans la classe SharedPreferences :

- getBoolean: pour récupérer un booléen
- getFloat: pour récupérer un float
- getInt: pour récupérer un int
- getLong: pour récupérer un long
- getString: pour récupérer une string

Ces méthodes prennent en paramètres la clef + la valeur par défaut si aucune valeur n'est trouvée pour cette clef

```
val highScore = sharedPref.getInt(SCORE_STATE_KEY, 0)
```



# Shared Preferences

Enregistrement des valeurs dans les préférences grâce à l'interface Editor

Pour récupérer une instance de la classe Editor, il faut appeler la méthode edit sur l'objet de type SharedPreferences

Puis pour ajouter des valeurs, il faut recourir aux méthodes putBoolean, putFloat, putInt, putLong, putString avec une clef et la valeur associée

Si une valeur est déjà présente en mémoire pour une clef, la valeur sera mise à jour.

Pour valider l'enregistrement des données il faut appeler la méthode commit (ou apply => async)

```
with (sharedPref.edit()) {  
    putInt(SCORE_STATE_KEY, newHighScore)  
    apply()  
}
```



# Shared Preferences

Possibilité d'écouter les modifications effectuées dans les préférences (ex : répercuter une préférence modifiée dans l'interface utilisateur) grâce à la méthode `registerOnSharedPreferenceChangeListener`

A l'inverse pour ne plus être notifié des modifications faites dans les préférences il faut utiliser la méthode `unregisterOnSharedPreferenceChangeListener`

```
sharedPref.registerOnSharedPreferenceChangeListener { sharedPreferences, key ->
    if (key == SCORE_STATE_KEY) {
        Toast.makeText(this, "value changed", Toast.LENGTH_SHORT).show()
    }
}
```

4

# Preferences Menu

# Preferences Menu

Android permet la création d'écran de préférences grâce à une activité de présentation et de persistance des paramètres. Ces écrans sont similaires aux menus des paramètres systèmes d'Android



Pays

Veuillez choisir un pays

Activer le mode debug

Le mode debug est désactivé



Saisissez les paramètres de debug

Paramètres de debug





# Preferences Menu

Le fragment **PreferenceFragmentCompat** permet de construire l'interface de votre menu de plusieurs façons : depuis un fichier XML qui doit être placé dans les ressources du projet (ex : /res/xml/preferences.xml) et dans lequel sera spécifié la hiérarchie des préférences

Grâce à ce type de fragment, la gestion des paramètres est facilitée. Chaque paramètre est enregistré automatiquement et récupérable depuis la méthode getDefaultSharedPreferences de la classe PreferenceManager.

Recette pour implanter une activité de préférences en utilisant un fichier XML :

1. Créer un fichier XML avec la structure des paramètres
2. Créer un Fragment dérivant de **PreferenceFragmentCompat**
3. Charger la structure XML des paramètres
4. Récupérer les valeurs de chaque paramètre à l'aide de la méthode getDefaultSharedPreferences

Dépendance nécessaire :

```
implementation "androidx.preference:preference-ktx:1.1.1"
```

# Preferences Menu

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android" >
    <ListPreference
        android:dialogTitle="Choisissez votre pays"
        android:entries="@array/list_countries"
        android:entryValues="@array/list_countries_values"
        android:key="country"
        android:summary="Veuillez choisir un pays"
        android:title="Pays" />
    <CheckBoxPreference
        android:key="checkBoxDebug"
        android:summary="Mode debug"
        android:summaryOff="Le mode debug est désactivé"
        android:summaryOn="Le mode debug est activé"
        android:title="Activer le mode debug" />
    <EditTextPreference
        android:dialogTitle="Veuillez saisir les paramètres"
        android:key="editTextDebugSettings"
        android:summary="Paramètres de debug"
        android:title="Saisissez les paramètres de debug"
        android:dependency="checkBoxDebug" />
</PreferenceScreen>
```



# Preferences Menu

Différents types de préférence : CheckBoxPreference, EditTextPreference, ListPreference, RingTonePreference

Chaque préférence est définie par une clef (identifiant) grâce à l'attribut android:key

android:title correspond au titre affiché dans le menu

android:summary correspond à la description affichée dans le menu

Fragment de préférence :

```
class MySettingsFragment : PreferenceFragmentCompat() {  
    override fun onCreatePreferences(savedInstanceState: Bundle?, rootKey: String?) {  
        setPreferencesFromResource(R.xml.preferences, rootKey)  
    }  
}
```

la méthode **setPreferencesFromResource** ajoute le menu de préférences au fragment à partir du fichier XML stocké dans les ressources



# Preferences Menu

Préférence de type liste

élément ListPreference

propriétés :

dialogTitle : titre de la boîte de dialogue

entries : liste d'éléments présentés à l'utilisateur (ressource de type tableau)

entryValues : liste des valeurs enregistrées dans les préférences

```
<ListPreference
    android:dialogTitle="Choisissez votre pays"
    android:entries="@array/list_countries"

    android:entryValues="@array/list_countries_values"
    android:key="country"
    android:summary="Veuillez choisir un pays"
    android:title="Pays" />
```

```
<resources>
    <string-array name="list_countries">
        <item>Allemagne</item>
        <item>Belgique</item>
        <item>France</item>
        <item>USA</item>
    </string-array>
    <string-array
name="list_countries_values">
        <item>de</item>
        <item>be</item>
        <item>fr</item>
        <item>usa</item>
    </string-array>
</resources>
```



# Preferences Menu

Préférence de type case à cocher

Deux valeurs possibles : activée/désactivée

propriétés :

summaryOn : description affichée lorsque la case à cocher est activée

summaryOff : description affichée lorsque la case à cocher est désactivée

Possibilité de lier une case à cocher à une autre préférence grâce à la propriété dependency. Cette liaison activera ou désactivera la préférence dépendante.

```
<CheckBoxPreference
    android:key="checkBoxDebug"
    android:summary="Mode debug"
    android:summaryOff="Le mode debug est
désactivé"
    android:summaryOn="Le mode debug est activé"
    android:title="Activer le mode debug" />
```



# Preferences Menu

Préférence de type zone de texte

Ce type de paramètre permet à l'utilisateur de saisir une valeur de type chaîne de caractères. Une boîte de dialogue apparaît avec une zone de saisie de texte.

```
<EditTextPreference
    android:dialogTitle="Veuillez saisir les
paramètres"
    android:key="editTextDebugSettings"
    android:summary="Paramètres de debug"
    android:title="Saisissez les paramètres de
debug"
    android:dependency="checkBoxDebug" />
```

# Preferences Menu

Catégories de préférences

permet de regrouper des préférences entre elles afin de rendre l'organisation des menus plus logiques et ergonomiques. Les catégories de préférences seront caractérisées grâce à l'élément PreferenceCategory

```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android" >
  <PreferenceCategory android:title="Général" >
    <ListPreference
      android:dialogTitle="Choisissez votre pays"
      android:entries="@array/list_countries"
      android:entryValues="@array/list_countries_values"
      android:key="country"
      android:summary="Veuillez choisir un pays"
      android:title="Pays" />
    </PreferenceCategory>
  <PreferenceCategory android:title="Debug" >
    <CheckBoxPreference
      android:key="checkBoxDebug"
      android:summary="Mode debug"
      android:summaryOff="Le mode debug est désactivé"
      android:summaryOn="Le mode debug est activé"
      android:title="Activer le mode debug" />
    ...
  </PreferenceCategory>
</PreferenceScreen>
```

# Preferences Menu

Les écrans de préférences imbriqués

Si les préférences sont trop nombreuses sur le même écran malgré l'utilisation des catégories

Il est possible d'imbriquer des écrans de préférences afin d'obtenir des sous-écrans (similaire au menu paramètres du système Android)

```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android" >

    <PreferenceCategory android:title="Général" >
        <ListPreference .../>
    </PreferenceCategory>
    <PreferenceCategory android:title="Préférences avancées"
    >
        <PreferenceScreen
            android:key="DebugDetails"
            android:summary="Configuration debug"
            android:title="Debug" >
            <CheckBoxPreference ... />

            <EditTextPreference ... />
        </PreferenceScreen>
    </PreferenceCategory>
</PreferenceScreen>
```



# Preferences Menu

Lire une préférence sauvegarder depuis le menus de préférences :

```
val sharedPref: SharedPreferences = PreferenceManager.getDefaultSharedPreferences(this)

val debugSettings = sharedPref.getString("editTextDebugSettings", "default value")
Log.d("TEST", "${debugSettings}")
```

ex2

# Exercise



## Exercice 2

Créez un nouveau projet dans Android Studio

Nom: Exo2

Placez le dans un dossier TECHNIFUTUR-AND14-EXO2







## Exercice 2

Créez une application qui permettra de faire un login.

Vous devez pouvoir saisir un nom d'utilisateur, un mot de passe, et une case pour mémoriser le nom d'utilisateur et le mot de passe.

Ajouter un bouton login. Sur un tap, sauvegardez les informations dans les préférences partagées.

Ajoutez un bouton setting, qui permettra d'afficher un menu de préférences avec une liste de choix (pour sélectionner une couleur de fond de l'écran login) et un texte, qui sera affiché au-dessus des champs du login.

Les summaries des préférences doivent changer pour refléter la valeur choisie.

**5**

# Fichiers



# Fichiers

fichier = élément de base du système Android pour stocker n'importe quel type de données : applications, ressources, bases de données, ...

Par défaut toutes les ressources sont stockées et regroupées dans le dossier res/

Cependant il existe des cas de figure où l'application aura besoin d'accéder à d'autres fichiers

deux méthodes disponibles pour simplifier l'accès aux fichiers : `openFileOutput` et `openFileInput`

La méthode `openFileOutput` permet d'ouvrir un fichier en écriture ou de le créer s'il n'existe pas. Si le fichier existe déjà, le fichier sera écrasé par défaut. Pour ajouter du contenu il faut spécifier le mode `MODE_APPEND`.

Ouverture d'un fichier en écriture

⚠ IMPOSSIBLE DANS LE THREAD PRINCIPAL

```
try {
    val ous: FileOutputStream = context.openFileOutput("fichier.dat", MODE_PRIVATE)
    val bw = BufferedWriter(OutputStreamWriter(ous))
    bw.write("Hello world" + System.getProperty("line.separator"))
    bw.close()
} catch (e: Exception) {
    println("Exception : $e")
}
```



# Fichiers

Ouverture d'un fichier en lecture

⚠ IMPOSSIBLE DANS LE THREAD PRINCIPAL

```
try {  
    val ins: FileInputStream = context.openFileInput("fichier.dat")  
    val reader = BufferedReader(InputStreamReader(ins))  
    var line: String?  
    while (reader.readLine().also { line = it } != null) {  
        //display the line  
    }  
    reader.close()  
} catch (e: Exception) {  
    println("Exception : $e")  
}
```



# Fichiers

Les fichiers sont enregistrés dans le répertoire `/data/data/com.my.package/files`  
La méthode `deleteFile` permet de supprimer un fichier :

```
context.deleteFile("fichier.dat"); //suppression du fichier
```

Il est possible de modifier les permissions du fichier utilisé par exemple pour partager le fichier avec d'autres applications  
Modes d'accès disponibles :

- **MODE\_PRIVATE** : mode par défaut, fichier accessible uniquement à l'application
- **MODE\_APPEND** : permet d'ajouter des données en fin de fichier au lieu d'écraser tout le fichier



# Fichiers

Accéder aux ressources brutes

Le développeur peut intégrer des ressources brutes localisées (ex : sons, vidéos, ...) dans le dossier res/raw

Ces ressources sont accessibles grâce à la méthode openRawResource de la classe Resources

Ex :

```
val ins = context
    .resources
    .openRawResource(R.raw.game_over_sound)
```

Méthodes utilitaires pour manipuler les fichiers :

- fileList : permet de récupérer la liste des fichiers
- getFilesDir : récupère le chemin absolu du répertoire où sont stockés les fichiers créés avec openFileOutput
- getFilePath : retourne le chemin absolu du répertoire pour le fichier passé en paramètre.

# Exercise



## Exercice 3

Créez un nouveau projet dans Android Studio

Nom: Exo3

Placez le dans un dossier TECHNIFUTUR-AND14-EXO3







## Exercice 3

Reprenez l'exercice précédent.

Sur chaque tentative de login, écrivez une ligne de log

Ajoutez un bouton logs qui lancera une activité logs. Cette activité affichera le contenu du fichier log.

6

# Content Provider



# Content Provider

Quand utiliser un Content Provider :

pour fournir des données complexes ou des fichiers à d'autres applications

pour permettre à l'utilisateur de copier des données complexes de votre application vers d'autres applications

fournir des suggestions de recherche utilisant le search framework.

ContentURIs

URI identifiant les données d'un provider

via un argument permet de déterminer la table, la ligne ou le fichier à accéder

Créer une sous-classe de ContentProvider.

Méthodes à surcharger :

OnCreate

query : pour retourner les données

insert : pour insérer de nouvelles données

update

delete

getType : retourne le MIME type des données du content provider (ex :

vnd.android.cursor.dir/vnd.com.neopixl.myapplication.provider.book)



# Content Provider

```
<provider
    android:name=".contentprovider.MyContentProvider"
    android:authorities="com.neopixl.myapplication.provider"
    android:enabled="true"
    android:exported="true" >
</provider>
```

authorities : content uri pour identifier le content provider dans le système

enabled : active ou désactive le provider au niveau du système

exported : expose ou non le provider aux autres applications

<https://developer.android.com/guide/topics/providers/content-provider-creating>

# Content Provider

```
class MyContentProvider: ContentProvider() {  
    private val bookCode = 1  
    private val bookByIdCode = 2  
  
    private val uriContent = "com.neopixl.myapplication.provider"  
  
    private var mUriMatcher //uri matcher pour le mapping  
        : UriMatcher? = null  
  
    override fun onCreate(): Boolean {  
        mUriMatcher = UriMatcher(UriMatcher.NO_MATCH)  
  
        //associe 1'uri book -> 1  
        mUriMatcher!!.addURI(uriContent, "book", bookCode)  
  
        //associe book/# -> 2  
        mUriMatcher!!.addURI(uriContent, "book/#", bookByIdCode)  
        return true  
    }  
}
```



# Content Provider

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder,
CancellationSignal cancellationSignal) {

    int uriCode = mUriMatcher.match(uri);

    switch (uriCode) {
        case bookCode:
            if (TextUtils.isEmpty(sortOrder)) sortOrder = "_ID ASC";
            break;
        case bookByIdCode:
            selection = selection + "_ID = "+uri.getLastPathSegment();
            break;
    }

    ...
}
```



# Content Provider

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Uri uri = BooksContract.Contacts.CONTENT_URI;
    String[] projection = new String[] { BooksContract.Contacts._ID,
        BooksContract.Contacts.DISPLAY_TITLE };
    String selection = BooksContract.Contacts.READ + " = " +
        + ("0") + "";
    String[] selectionArgs = null;
    String sortOrder = BooksContract.Contacts.DISPLAY_TITLE
        + " COLLATE LOCALIZED ASC";

    Cursor unreadBooksCursor = managedQuery(uri, projection, selection, selectionArgs, sortOrder);

    ...
}
```

**6**



**Room**





# Room

Room est un ORM (object relational mapper) pour utiliser une DB SQLite en Android. Room fait partie des composants d'Architecture fournis par Google.

Room vous permet de manipuler des bases de données rapidement. Vous pouvez le voir comme une couche d'abstraction au-dessus de SQLite.



# Room

Pour utiliser room, ajoutez les dépendances suivantes dans votre projet.

```
dependencies {
    def room_version = "2.3.0"

    implementation("androidx.room:room-runtime$room_version")
    annotationProcessor "androidx.room:room-compiler$room_version"
    // To use Kotlin annotation processing tool (kapt)
    kapt("androidx.room:room-compiler$room_version")

    // To use Kotlin Symbolic Processing (KSP) (Jetpack Compose)
    ksp("androidx.room:room-compiler$room_version")

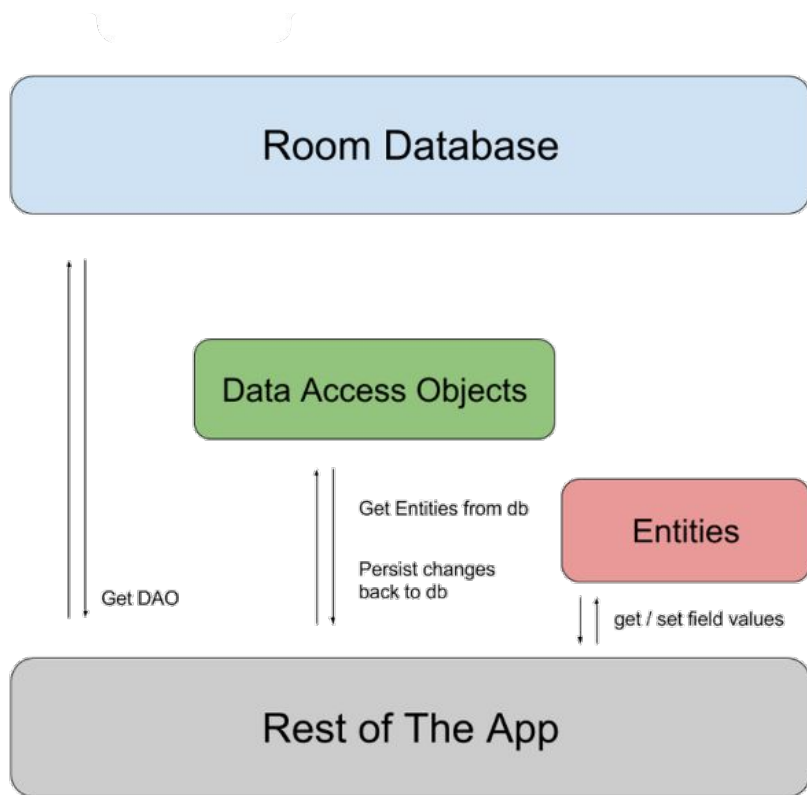
    // optional - Kotlin Extensions and Coroutines support for Room
    implementation("androidx.room:room-ktx$room_version")
    // optional - RxJava2 support for Room
    implementation "androidx.room:room-rxjava2$room_version"
    // optional - RxJava3 support for Room
    implementation "androidx.room:room-rxjava3$room_version"
    // optional - Guava support for Room, including Optional and ListenableFuture
    implementation "androidx.room:room-guava$room_version"
    // optional - Test helpers
    testImplementation("androidx.room:room-testing$room_version")
    // optional - Paging 3 Integration
    implementation("androidx.room:room-paging:2.4.0-alpha04")
}
```



# Room

Les trois principaux composants de Room sont :

- Base de données : Elle représente la base de données, c'est un objet qui détient une connexion à la base de données SQLite et toutes les opérations sont exécutées à travers elle. Il est annoté avec **@Database.**
- Entité : Représente une table dans la base de données. Chaque classe doit être annoté avec **@Entity.**
- DAO : Une interface qui contient les méthodes pour accéder à la base de données. Il est annoté avec **@Dao.**



# Room - Entity

Toutes les classes qui représentent une entité de la base de données doivent être annotées avec **@Entity**.

L'annotation **@PrimaryKey(autoGenerate = true)** nous indiquera que la variable est la clé primaire de l'entité et devrait être auto générée par le moteur de base de données.

L'annotation **@ColumnInfo(name = "your\_field\_name")** nous indiquera que le nom interne du champ en base de données sera **your\_field\_name**. Si cette information n'est pas présente, le nom sera déterminé en fonction du nom de la variable.

```
@Entity
data class User(
    @PrimaryKey(autoGenerate = true)
    val id: Long,
    @ColumnInfo(name = "first_name")
    val firstName: String?,
    @ColumnInfo(name = "last_name")
    val lastName: String?
)
```

# Room - DAO

Les classes qui gèrent l'accès aux données sont annotées avec **@Dao**.

Les annotations **@Insert**, **@Update** et **@Delete** nous indiqueront que les méthodes sont des méthodes d'ajout, de mise à jour et de suppression de données. **@Insert** peut prendre un **onConflict** pour spécifier l'action à prendre si un objet existe déjà en base de donnée.

L'annotation **@Query("YOUR CUSTOM REQUEST")** permet de spécifier une requête SQL custom pour votre méthode.

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user" )
    fun getAll(): List<User>
    @Query("SELECT * FROM user WHERE uid IN (:userIds)" )
    fun loadAllByIds (userIds: IntArray): List<User>
    @Query("SELECT * FROM user WHERE first_name LIKE :first AND last_name LIKE :last LIMIT
1")
    fun findByName (first: String, last: String): User
    @Insert (onConflict = OnConflictStrategy.REPLACE)
    fun insert (user: User)
    @Insert
    fun insertAll (vararg users: User)
    @Delete
    fun delete (user: User)
}
```



# Room - Database

Votre base de données doit hériter de **RoomDatabase**.

Elle doit aussi être annotée de l'annotation **@Database**.

**@Database** prend en paramètre une liste d'entités, qui compose votre base de données, et une version. Il est très important de mettre à jour le numéro de version si vous changez le format de la base de données entre deux mises en production de votre application.

Votre **RoomDatabase** définit la configuration de la base de données et sert de point d'accès principal de l'application aux données persistantes. La classe de base de données doit remplir les conditions suivantes :

Pour chaque classe DAO associée à la base de données, la classe de base de données doit définir une méthode abstraite qui n'a aucun argument et renvoie une instance de la classe DAO.

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

# Room - Utilisation

```
lateinit var db:AppDatabase
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    db = Room.databaseBuilder(this, AppDatabase::class.java, "Sample.db")
        .build()
}
fun buttonClick(view: android.view.View) {
    lifecycleScope.launch {
        dbActions()
    }
}
suspend fun dbActions() {
    withContext(Dispatchers.Default) {
        val userDao = db.userDao()
        var users = userDao.getAll()
        Log.d("TEST", "user count : ${users.size}")
        userDao.insert(User( firstName = "Axel", lastName = "Glibert"))
        users = userDao.getAll()
        Log.d("TEST", "users : ${users}")
    }
}
```



## Room - Relations

Il existe plusieurs types de relation.

La plus simple, c'est le cas où toutes les données sont dans une même table, mais où vous voulez représenter certaines données sous forme d'un objet à part entière.

L'annotation **@Embedded** nous permet dans ce cas-ci, de définir un sous objet.

```
data class Address(  
    val street: String?,  
    val state: String?,  
    val city: String?,  
    @ColumnInfo(name = "post_code") val postCode: Int  
)  
  
@Entity  
data class User(  
    @PrimaryKey val id: Long,  
    val firstName: String?,  
    @Embedded val address: Address?  
)
```



# Room - Relations

La relation One to One est une relation où on lie deux objets présents dans deux tables différentes.

Il vous faut ensuite créer un objet qui sera la relation avec vos entités.

Utilisez l'annotation **@Relation** pour spécifier les champs qui serviront à la relation.

Enfin, ajoutez à votre Dao une méthode pour récupérer votre nouvel objet. L'annotation **@Transaction** est présente pour s'assurer que la requête se fasse de manière atomique.

```
@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Pet(
    @PrimaryKey val petId: Long,
    val userOwnerId: Long
)
```

```
data class UserAndPet(
    @Embedded val user: User,
    @Relation(
        parentColumn = "userId",
        entityColumn = "userOwnerId"
    )
    val pet: Pet
)
```

```
@Transaction
@Query("SELECT * FROM User")
fun getUsersAndPets(): List<UserAndPet>
```

# Room - Relations

La relation One to Many est une relation ou on lie un objets à une liste d'autres objets.

Le principe est le même que pour les relation One to One, mais ici on reçoit une Liste d'entité au lieu d'une entité dans notre objet qui fait la relation.

```
data class UserWithPets(  
    @Embedded val user: User,  
    @Relation(  
        parentColumn = "userId",  
        entityColumn = "userOwnerId"  
    )  
    val pets: List<Pet>  
)
```

```
@Transaction  
@Query("SELECT * FROM User")  
fun getUsersWithPetss(): List<UserWithPets>
```

# Room - Relations

La relation Many to Many est une relation où on lie un objet à une liste d'autres objets.

Le principe est le même que pour les relations One to Many, mais doublé et où on précise une entité intermédiaire qui fera la relation.

Pour la partie entité, on a des entités classiques, et une entité intermédiaire.

Cette entité intermédiaire a une clé primaire composée, qui est spécifiée directement au niveau de l'entité via un paramètre.

```
@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val playlistName: String
)

@Entity
data class Song(
    @PrimaryKey val songId: Long,
    val songName: String,
    val artist: String
)

@Entity(primaryKeys = ["playlistId",
"songId"])
data class PlaylistSongCrossRef(
    val playlistId: Long,
    val songId: Long
)
```

# Room - Relations

Les Relations sont définies comme deux relations One to Many.

On précise cependant une entité intermédiaire, qui fait la relation.

```
data class PlaylistWithSongs(  
    @Embedded val playlist: Playlist,  
    @Relation(  
        parentColumn = "playlistId",  
        entityColumn = "songId",  
        associateBy = Junction(PlaylistSongCrossRef:: class)  
    )  
    val songs: List<Song>  
)  
  
data class SongWithPlaylists(  
    @Embedded val song: Song,  
    @Relation(  
        parentColumn = "songId",  
        entityColumn = "playlistId",  
        associateBy = Junction(PlaylistSongCrossRef:: class)  
    )  
    val playlists: List<Playlist>  
)
```



# Room - Relations

Finalement, nous avons notre Dao avec deux méthodes similaires à une relation One To Many.

```
@Transaction
@Query("SELECT * FROM Playlist" )
fun getPlaylistsWithSongs (): List<PlaylistWithSongs>

@Transaction
@Query("SELECT * FROM Song" )
fun getSongsWithPlaylists (): List<SongWithPlaylists>
```

# Room - Relations

Il est possible de faire des relations imbriquées.

Si je reprends les exemples précédents. Je peux ajouter une relation entre un utilisateur et une playlist. Pour récupérer la totalité des playlists et leur contenu, je peux faire la relation entre mon utilisateur et ma relation Many To Many. Je dois juste préciser que la relation se fait sur l'entité Playlist.

```
data class UserWithPlaylistsAndSongs(  
    @Embedded val user: User,  
    @Relation(  
        entity = Playlist::class,  
        parentColumn = "userId",  
        entityColumn = "userCreatorId"  
    )  
    val playlists:  
    List<PlaylistWithSongs>  
)
```

```
@Entity  
data class Playlist(  
    @PrimaryKey val playlistId: Long,  
    val userCreatorId: Long,  
    val playlistName: String  
)
```

```
@Transaction  
@Query("SELECT * FROM User")  
fun getUsersWithPlaylistsAndSongs():  
    List<UserWithPlaylistsAndSongs>
```



# Room - Converter

Parfois, vous avez besoin que votre application stocke un type de données personnalisé dans une seule colonne de base de données.

Vous devez donc prendre en charge les types personnalisés en fournissant des convertisseurs de type.

Ce sont des méthodes qui indiquent à Room comment convertir des types personnalisés vers et à partir de types connus que Room peut conserver.

Vous identifiez les convertisseurs de type à l'aide de l'annotation **@TypeConverter**.

```
class Converters {  
    @TypeConverter  
    fun fromTimestamp(value: Long?): Date? {  
        return value?.let { Date(it) }  
    }  
  
    @TypeConverter  
    fun dateToTimestamp(date: Date?): Long? {  
        return date?.time?.toLong()  
    }  
}
```



# Room - Converter

Vous devez ensuite passer vos convertisseurs à vos objets room qui vont les utiliser. Cela se fait via l'annotation **@TypeConverters**.

Il est possible de le faire à plusieurs niveau :

- Sur une base de données, tous les Daos et entités de cette base de données pourront l'utiliser.
- Sur un Dao, toutes les méthodes du Dao pourront l'utiliser.
- Sur une Entité, tous les champs de l'Entité pourront l'utiliser.
- Sur un POJO, tous les champs du POJO pourront l'utiliser.
- Sur un champ Entité, seul ce champ pourra l'utiliser.
- Sur une méthode Dao, tous les paramètres de la méthode pourront l'utiliser.
- Sur un paramètre de méthode Dao, seul ce champ pourra l'utiliser.

```
@Database(entities = arrayOf(User::class),  
version = 1)  
@TypeConverters(Converters::class)  
abstract class AppDatabase : RoomDatabase() {  
    abstract fun userDao(): UserDao  
}
```



# Room - Converter

Parfois, il est nécessaire de passer des informations pour initialiser un converteur. Il faut alors spécifier l'annotation **@ProvidedTypeConverter** et passer une instance du converteur lors de la création de votre instance de DB Room.

```
@ProvidedTypeConverter
class ExampleConverter( val someCustomData: String) {
    @TypeConverter
    fun StringToExample (string: String?): ExampleType?
    {
        ...
    }

    @TypeConverter
    fun ExampleToString (example: ExampleType?): String?
    {
        ...
    }
}
```

```
val exampleConverterInstance = ExampleConverter( "Test")

val db = Room.databaseBuilder( this, AppDatabase::class.java, "Sample.db")
    .addTypeConverter(exampleConverterInstance)
    .build()
```



## Room - Initial Data

Il est possible de charger la DB avec des données initiales. Pour ce faire, il y a plusieurs méthodes. La plus simple, si les données sont simples, est de les ajouter manuellement.

Il est aussi possible de charger des données initiales via des assets ou un fichier. Cela aura pour effet de créer la base de données initiale avec un fichier venant des assets, ou bien d'un autre fichier sélectionné par vous.

```
Room.databaseBuilder(applicationContext,
    AppDatabase::class.java, "Sample.db")
    .createFromAsset("database/myapp.db")
    .build()
```

```
Room.databaseBuilder(applicationContext,
    AppDatabase.class, "Sample.db")
    .createFromFile(File("mypath"))
    .build()
```



# Room - Migration

Quand vous mettez à jour votre application, il arrive que vous deviez modifier le schéma de données de votre base de données. Dans ce cas, l'utilisateur aura une application avec un ancien format de base de données. Il faudra donc l'adapter.

Il existe trois types de migration:

- Destructive : on reset la DB avec une db vierge ou une autre DB provenant des assets ou d'un fichier.
- Migration : on migre les données de l'ancien format vers le nouveau format.
- Automatique : la migration se fait de manière automatique.



# Room - Migration

En appelant **fallbackToDestructiveMigration** lors de la création de notre DB, on spécifie qu'en cas de nouvelle version de la DB, on la récrée.

Attention, l'utilisateur perdra donc ses données.

```
Room.databaseBuilder(applicationContext,  
    AppDatabase.class, "Sample.db")  
    .createFromAsset("database/myapp.db")  
    .fallbackToDestructiveMigration()  
    .build()
```

# Room - Migration

En mode Migration, on crée un objet Migration, qui va spécifier les actions nécessaires à faire pour passer d'une version de la DB à une autre.

On peut préciser plusieurs Migrations, pour différentes versions de DB.

On peut aussi combiner la migration avec le `fallbackToDestructiveMigration`, dans le cas où les migrations ne sont pas possibles.

```
// Migration path definition from version 2 to version 3.
val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL( "CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, PRIMARY
KEY(`id`))" )
        database.execSQL( "ALTER TABLE Book ADD COLUMN pub_year INTEGER" )
    }
}

Room.databaseBuilder( applicationContext, AppDatabase.class, "Sample.db" )
    .createFromAsset( "database/myapp.db" )
    .addMigrations(MIGRATION_2_3)
    .fallbackToDestructiveMigration()
    .build()
```



# Room - Migration

En mode Migration automatique, on laisse Room gérer la migration. Il va le faire au mieux.

```
// Database class before the version update.
@Database (
    version = 1,
    entities = [User::class]
)
abstract class AppDatabase : RoomDatabase() {
    ...
}

// Database class after the version update.
@Database (
    version = 2,
    entities = [User::class],
    autoMigrations = [
        AutoMigration ( from = 1, to = 2)
    ]
)
abstract class AppDatabase : RoomDatabase() {
    ...
}
```

# Room - Migration

Dans certains cas, il est impossible de faire la migration automatique. Il faut donc l'aider.

C'est souvent le cas quand on renomme une table ou qu'on supprime une colonne.

On doit donc créer une classe héritant de **AutoMigrationSpec** dans notre classe **RoomDatabase**.

On va lui passer des annotations pour l'aider

- @DeleteTable
- @RenameTable
- @DeleteColumn
- @RenameColumn

```
@Database (
    version = 2,
    entities = [User::class],
    autoMigrations = [
        AutoMigration (
            from = 1,
            to = 2,
            spec = AppDatabase.MyAutoMigration::class
        )
    ]
)

abstract class AppDatabase : RoomDatabase() {
    @RenameTable (fromTableName = "User", toTableName =
    "AppUser")
    class MyAutoMigration : AutoMigrationSpec() { }

    ...
}
```



## Room - Asynchrone

Jusqu'à maintenant, les appels étaient fait de manière synchrone.

Le temps d'accès à une base de données n'est pas null, donc il est préférable de faire les accès de manière asynchrone.

Room permet de faire les appels de manière asynchrone avec des coroutines, RxJava, ou encore des LiveData.





# Room - Asynchrone

Coroutines :

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUsers(vararg users: User)

    @Update
    suspend fun updateUsers(vararg users: User)

    @Delete
    suspend fun deleteUsers(vararg users: User)

    @Query("SELECT * FROM user WHERE id = :id")
    suspend fun loadUserById(id: Int): User

    @Query("SELECT * from user WHERE region IN (:regions)")
    suspend fun loadUsersByRegion(regions: List<String>): List<User>
}
```



# Room - Asynchrone

LiveData :

```
@Dao
interface UserDao {
    // Returns the number of users inserted.
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUsers(users: List<User>)

    // Returns the number of users updated.
    @Update
    suspend fun updateUsers(users: List<User>): Long

    // Returns the number of users deleted.
    @Delete
    suspend fun deleteUsers(vararg users: User): Long

    @Query("SELECT * FROM user WHERE id = :id" )
    fun loadUserById(id: Int): LiveData<User>

    @Query("SELECT * from user WHERE region IN (:regions)" )
    fun loadUsersByRegion(regions: List<String>): LiveData<List<User>>
}
```

# Room - Asynchrone

Exemple d'utilisation avec des LiveData :

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user" )
    fun getAll(): LiveData<List<User>>
    @Query("SELECT * FROM user WHERE uid IN (:userIds)" )
    fun loadAllByIds (userIds: IntArray): LiveData<List<User>>
    @Query("SELECT * FROM user WHERE first_name LIKE :first AND last_name LIKE :last LIMIT 1")
    fun findByName (first: String, last: String): LiveData<User>
}
```

```
fun buttonClick(view: android.view.View) {
    val userDao = db.userDao()
    userDao.getAll().observe(this) {
        Log.d("TEST", "users : ${it}")
    }
}
```

# Room - Asynchrone

Exemple d'utilisation avec des LiveData + ViewModel :

```
private lateinit var userViewModel: UserViewModel
```

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    binding = ActivityMainBinding.inflate(layoutInflater)  
    val view = binding.root  
    setContentView(view)  
    userViewModel = ViewModelProvider(this).get(UserViewModel::class.java)
```

```
fun buttonClick(view: android.view.View) {  
    userViewModel.getUserDetails(this, "Axel", "Glibert")?.observe(this) {  
        binding.mainTextView.text = "${it.firstName} ${it.lastName}"  
    }  
}
```



# Room - Asynchrone

Exemple d'utilisation avec des LiveData + ViewModel :

```
internal class UserViewModel: ViewModel() {
    var liveDataUser: LiveData<User>? = null

    fun insertData(context: Context, firstName: String, lastName: String) {
        UserRepository.insertData(context, firstName, lastName)
    }

    fun getUserDetails(context: Context, firstName: String, lastName: String): LiveData<User>? {
        liveDataUser = UserRepository.getUserDetails(context, firstName, lastName)
        return liveDataUser
    }
}
```



# Room - Asynchrone

Exemple d'utilisation avec des LiveData + ViewModel :

```
class UserRepository {
    companion object {
        var userDatabase: AppDatabase? = null
        var user: LiveData<User>? = null
        fun initializeDB(context: Context) : AppDatabase {
            return AppDatabase.getDB(context)
        }
        fun insertData(context: Context, firstName: String, lastName: String) {
            userDatabase = initializeDB(context)
            CoroutineScope(IO).launch {
                val user = User(firstName = firstName, lastName = lastName)
                userDatabase!!.userDao().insert(user)
            }
        }
        fun getUserDetails(context: Context, firstName: String, lastName: String) : LiveData<User>? {
            userDatabase = initializeDB(context)
            user = userDatabase!!.userDao().findByName(firstName, lastName)
            return user
        }
    }
}
```

# Room - Asynchrone

Exemple d'utilisation avec des LiveData + ViewModel :

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    companion object {

        @Volatile
        private var sharedInstance: AppDatabase? = null

        fun getDB(context: Context) : AppDatabase {
            if (sharedInstance != null) return sharedInstance!!
            synchronized(this) {
                sharedInstance = Room
                    .databaseBuilder(context, AppDatabase::class.java,
                        "Sample.db")
                    .fallbackToDestructiveMigration()
                    .build()
                return sharedInstance!!
            }
        }
    }
}
```

ex4

# Exercise





## Exercice 4

Créez un nouveau projet dans Android Studio

Nom: Exo4

Placez le dans un dossier TECHNIFUTUR-AND14-EXO4



## Exercice 4

Créer une application qui va gérer une liste de livres.

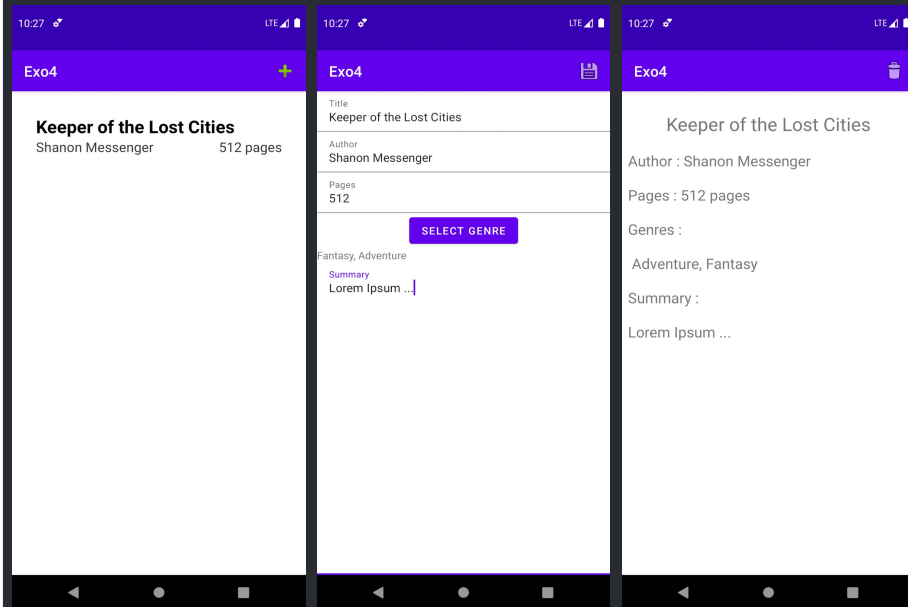
Un livre se compose d'un titre, un auteur, un nombre de pages, un ou plusieurs genres et un résumé.

Un genre est composé d'un nom.

Vous devez pouvoir ajouter un livre, afficher le détail d'un livre, et le supprimer.

Un livre peut avoir plusieurs genres.

Créez plusieurs genres dans la base de données la première fois que vous lancez l'application.



7

# Room + Flow



# Flow

Les Flows permettent de gérer des données mises à jour séquentiellement et en continu.

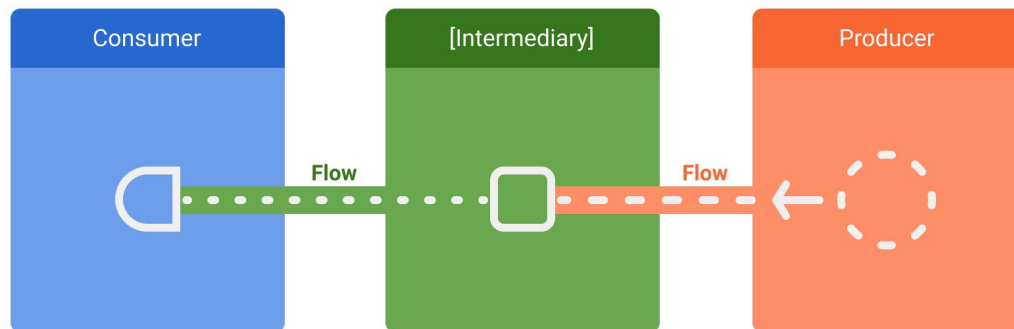
Pour par exemple recevoir des mises à jour en live d'une base de données.

Les Flows fonctionnent conjointement avec les coroutines et peuvent être représentées comme un tuyau faisant circuler les données.

# Flow

3 entités sont concernées lors de l'utilisation d'un Flow.

- Le **producer** : produit la donnée qui est ajoutée au flux (asynchrone grâce aux coroutines).
- Les **intermediaries** : qui peuvent modifier les valeurs émises.
- Le **consumer** : qui va consommer / utiliser les valeurs émises par le flux de données.



# Création d'un Flow

```
class NewsRemoteDataSource(  
    private val newsApi: NewsApi ,  
    private val refreshIntervalMs: Long = 5000  
) {  
    val latestNews: Flow<List<ArticleHeadline>> = flow {  
        while(true) {  
            val latestNews = newsApi.fetchLatestNews()  
            emit(latestNews) // Emits the result of the request to the flow  
            delay(refreshIntervalMs) // Suspends the coroutine for some time  
        }  
    }  
}  
  
// Interface that provides a way to make network requests with suspend functions  
interface NewsApi {  
    suspend fun fetchLatestNews(): List<ArticleHeadline>  
}
```

ex4b

# Exercise



## Exercice 4b

Sur base de l'exercice 4, remplacez les liveData par des Flow.

On va le faire ensemble !





8

# ObjectBox



# Object Box

Object box est similaire à Room

Pour utiliser ObjectBox, ajoutez dans le gradle du projet :

```
classpath 'io.objectbox:objectbox-gradle-plugin:2.9.1'
```

Et dans le gradle du module :

```
dependencies {  
    implementation "io.objectbox:objectbox-kotlin:2.9.1"  
}  
  
// Doit être à la fin du fichier sinon problème en debug avec le Browser  
apply plugin: 'io.objectbox'
```





# Object Box

@Entity : défini que c'est une entité ObjectBox

@Id : défini la clé primaire, toujours long

```
@Entity
data class Movie(
    @Id
    var id: Long = 0,
    var title: String? = null,
    var director: String? = null,
){
    lateinit var genre: ToOne<Genre>
}
```



# Object Box

Exemple de classe utilitaire pour gérer le boxStore, qui est le point d'entrée d'ObjectBox.

Il est idéalement initialisé dans votre classe Application

**AndroidObjectBrowser** : utilitaire permettant de voir la db depuis un navigateur ()

```
object ObjectBox {
    var boxStore: BoxStore? = null
    fun init(context: Context) {
        boxStore = MyObjectBox.builder()

        .androidContext(context.applicationContext)
            .build()

        if (BuildConfig.DEBUG) {
            val started =
                AndroidObjectBrowser(boxStore)
                .start(context.applicationContext)
                Log.i("ObjectBrowser", "Started:
                $started")
                Log.d(
                    "TAG",
                    String.format(
                        "Using ObjectBox %s (%s)",
                        BoxStore.getVersion(),
                        BoxStore.getVersionNative()
                    )
                )
        }
    }
}
```



# Object Box

Ajouter ceci dans les dépendances du projet pour utiliser le browser de debug

Attention, le apply plugin doit se trouver après le bloc de dépendances.

Astuce, utilisez “adb forward tcp:8090 tcp:8090” pour pouvoir y accéder avec le navigateur de votre mac (<http://localhost:8090>)

```
class App: Application() {  
    override fun onCreate() {  
        super.onCreate()  
        ObjectBox.init(this);  
    }  
}
```

```
dependencies {  
    //implementation "io.objectbox:objectbox-kotlin:2.9.1"  
    debugImplementation "io.objectbox:objectbox-android-objectbrowser:2.9.1"  
    releaseImplementation "io.objectbox:objectbox-android:2.9.1"  
}
```



# Object Box

Relation One To One

Permet de faire un lien vers un autre objet. Attention, cet autre objet doit déjà être en db pour que le lien soit possible.

```
@Entity
data class Movie(
    @Id
    var id: Long = 0,
    var title: String? = null,
    var director: String? = null,
){
    lateinit var genre: ToOne<Genre>
}
```

```
terminator.genre.target = sf
```



# Object Box

ToMany

@Backlink permet de définir une relation inverse

```
@Entity
data class Genre (
    @Id
    var id: Long = 0,
    var name: String? = null,
){
    @Backlink(to = "genre")
    lateinit var movies: ToMany<Movie>
}
```

```
sf.movies.add(terminator)
```



# Object Box

Query : permet de faire des recherche dans la DB sur certains critères

les requêtes peuvent être de type equal, greater, startWith, ...  
plusieurs conditions peuvent être mise sur une même requête

```
ObjectBox.boxStore?.boxFor(Genre::class.java)?.let {  
    val builder: QueryBuilder<Genre> = it.query()  
    builder.equal(Genre_.name, "Science Fiction")  
    val existingGenre = builder.build().findFirst()  
    Log.d("TEST", "foundGenre : ${existingGenre}")  
}
```





# Object Box

ObjectBox propose aussi des méthodes retournant des Live Data.

Pour ObjectBox, il faut utiliser la classe **ObjectBoxLiveData**.

```
movieBoxStore?.let {  
    val movieLiveData = ObjectBoxLiveData<Movie>(it.query().order(Movie_.title).build())  
    movieLiveData.observe(this) { movieList ->  
        Log.d("TEST2", "$movieList")  
    }  
}
```

ex5

# Exercise



## Exercice 5

Créez un nouveau projet dans Android Studio

Nom: Exo5

Placez le dans un dossier TECHNIFUTUR-AND14-EXO5





## Exercice 5

Reprenez l'exercice précédent, et remplacer Room par ObjectBox



# Fil rouge



# Intégration fil rouge

Depuis develop, créez une branche feature/FR010

Votre implémentation se fera sur cette branche

Une fois votre implémentation terminée, vous devrez “merge”  
votre branche sur **develop**

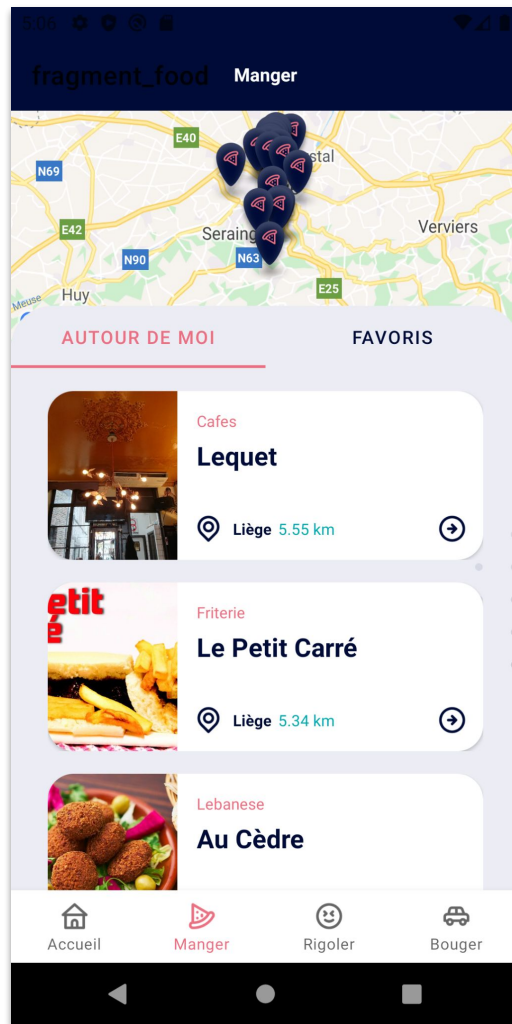


# Intégration fil rouge

Vous allez devoir ajouter des fonctionnalités à l'écran "Manger"

Vous allez ajouter la possibilité de mettre des favoris. Une liste de favoris sera disponible depuis un sélecteur sur cet écran.

Il sera possible d'ajouter ou de supprimer un élément des favoris depuis la vue de détail.



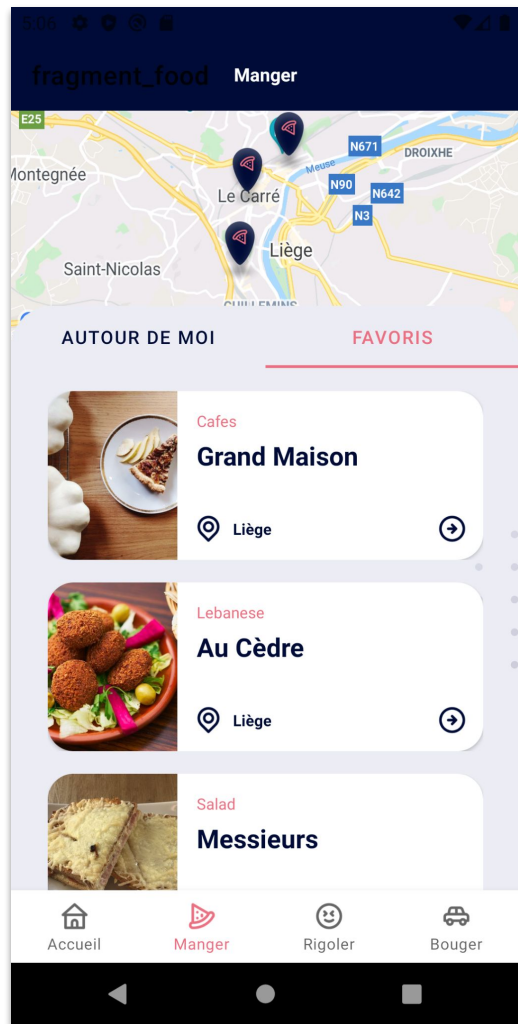
# Intégration fil rouge

Vous utiliserez un TabLayout avec deux “tabs” en haut de liste afin de recharger les données:

- Depuis le réseau pour l’option “Près de moi”
- Depuis votre base de données pour l’option “Favoris”

Le “tab” sélectionné sera affiché en rose et souligné

L’autre tab sera affiché en bleu





# Intégration fil rouge

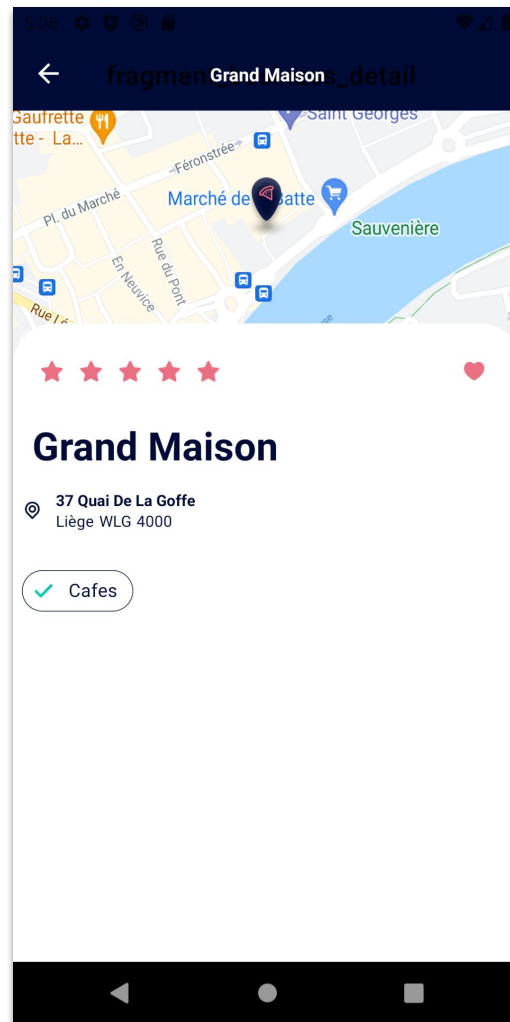
La vue de détail sera affichée suite à un clic sur une cellule de la liste (que ce soit en mode favoris ou proximité)

Cette vue affichera:

- Une carte
- Un rating sous forme d'étoiles
- Un picto (coeur) afin de mettre ou retirer le lieu des favoris  
Il sera rose si ajouté, blanc avec bordure bleue sinon
- Le titre
- L'adresse (rue, ville, état, code postal et picto)
- Les catégories sous forme de "chip"

Documentation sur le composant material chip et chip group  
<https://material.io/components/chips/android#using-chips>

Vous trouverez les ressources drawable dans le dossier partagé





## Restons en contact

**neopixl.**

A SMILE GROUP COMPANY

115A, Rue Emile Mark  
L-4620 Differdange

(+352) 26 58 06 03  
[contact@neopixl.com](mailto:contact@neopixl.com)