Chapitre 7

Persistance et partage des données

1 Introduction

Tous les exemples dans ce document ont été validés pour l'API 33.

Cependant, Google change à chaque nouvelle API les règles d'accès aux données!

Vérifier régulièrement la documentation d'Android, que la méthode valable à l'API 33 est toujours fonctionnelle dans la nouvelle API.

Google a regroupé ces tergiversations entre les différentes API, sur cette page :

Android storage use cases and best practices
https://developer.android.com/training/data-storage/use-cases

Ce lien résume les changements majeurs dans les versions API 29 et API 30 :

Android 10 & 11 storage cheat sheet
https://petrakeas.medium.com/android-10-11-storage-cheat-sheet-76866a989df4

Android offre plusieurs méthodes pour stocker les données d'une application.

La solution choisie va dépendre des besoins : données privées ou données publiques, un petit ensemble de données à préserver ou un large ensemble à préserver localement ou à travers le réseau.

Ces méthodes utilisent soit des éléments propres à l'API Java ou ceux associés à l'API d'Android.

Ces méthodes sont :

Persistance dans l'état d'une application

On utilise pour cela la notion de « Bundle » et les différents cycles de l'activité pour sauvegarder l'information utile à l'aide du « bundle » et récupérer cette information dans un autre état de l'activité.

On ne peut utiliser qu'un seul bundle, par ailleurs la donnée n'est pas persistante et n'est disponible que tant que l'application est utilisée.

Préférences partagées

Un ensemble de paires : clé et valeur. Clé est un « String » et Valeur est un type primitif (« boolean », « String », etc.).

Ces préférences sont gérées à travers un code Java ou bien à travers une activité.

Les données ne sont pas cryptées.

Les préférences sont adaptées pour des paires simples, mais dès qu'il est question de données plus complexes, il est préférable d'utiliser des fichiers.

Fichiers (création et sauvegarde)

Android permet la création, la sauvegarde et la lecture de fichiers à travers un média persistant (mémorisation et disponibilité).

Les fichiers peuvent être de n'importe quel type (image, XML, etc.).

Les fichiers peuvent être considérés pour une utilisation interne, donc local à l'application, ou bien externe, donc partagée avec plusieurs applications.

Base de données relationnelle, SQLite

Android offre aussi la possibilité d'utiliser toutes les propriétés d'une base de données relationnelle.

Android utilise pour cela une base de données basée sur « SQLite » (www.sqlite.org).

Android stocke la base de données localement à l'application.

Si l'on veut partager cette structure de données avec d'autres applications, il faudra utiliser dans ce cas un gestionnaire de contenu (content provider) configuré à cet effet.

Stockage réseau

Android permet de stocker des fichiers sur un serveur distant.

On peut utiliser pour cela les différentes techniques examinées dans le chapitre « Internet ».

2 Persistance dans l'état d'une application

Android peut arrêter l'activité et la redémarrer quand il y a :

- Rotation de l'écran.
- Changement de langue.
- L'application est en arrière-plan et le système a besoin de ressources.
- Et quand vous cliquez le bouton « retour » (« back »).

Ce redémarrage peut provoquer la perte des changements apportés à votre activité.

Une solution consiste à préserver les données dans un bundle à travers la méthode onSaveInstanceState puis récupérer cette information par la suite à l'aide de la méthode onRestoreInstanceState (ou bien dans la méthode onCreate).

Cette solution n'est pas appropriée dans le cas d'un clic sur le bouton « retour ». Dans ce cas, l'application démarre à partir de zéro et les données préservées sont détruites.

Généralement, l'orientation est gérée par la création d'une vue appropriée. Mais supposons que vous n'ayez pas eu le temps de le faire!

Nous pouvons fixer dans le fichier « AndroidManifest.xml » l'orientation supportée, comme suit :

```
<activity android:name=".YourActivity"
android:label="@string/app_name"
android:screenOrientation="portrait">
```

Nous allons examiner comment on préserve une donnée rattachée à une activité.

Examiner l'exemple « **C07 : Pers_Activity_1** ».

On déclare un widget « EditText », on insère une valeur quelconque, puis on change l'orientation de l'écran (CTRL-F11/CTRL-F12) :

```
<EditText android:layout_width="fill_parent"
  android:layout_height="wrap_content"/>
```

On constate que l'on a perdu le contenu du Widget en changeant l'orientation de l'écran.

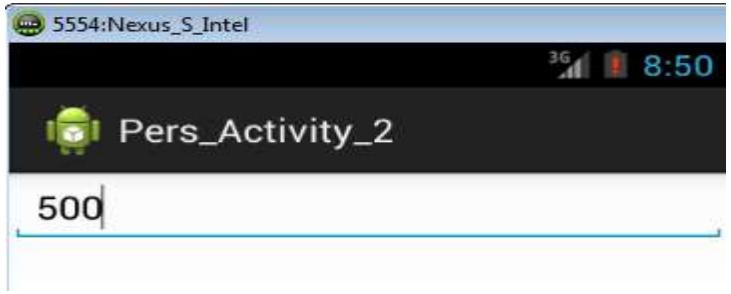
Si on avait ajouté dans le fichier manifeste cette ligne « `android:configChanges="orientation|screenSize"` », le contenu du Widget va être préservé. Cependant si une vue paysage a été définie avec l'application, Android ne va pas la prendre en considération.

Examiner l'exemple « C07 : Pers_Activity_2 ».

On reprend la même activité et on identifie maintenant le Widget, « EditText », avec un identificateur comme suit :

```
<EditText android:id="@+id/edittext1"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"/>
```

On teste de nouveau l'activité et on remarque maintenant que le contenu du Widget a été préservé.



Le fait, de passer d'une orientation à une autre, a fait en sorte que l'activité a été arrêtée puis démarrée de nouveau. Sauf que cette fois-ci, le contenu du Widget n'a pas été remis à zéro.

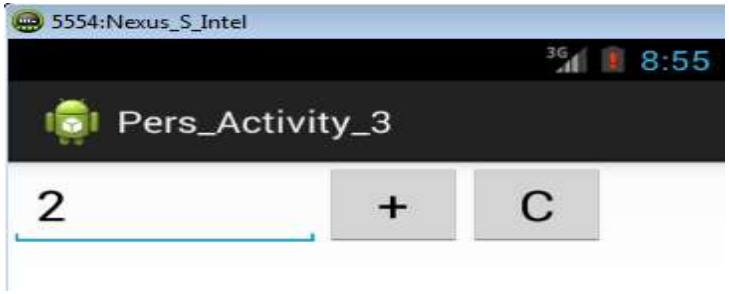
Android préserve automatiquement l'état d'un Widget quand ce dernier est identifié. Ce qui est le cas dans le second exemple.

Prenons un autre exemple, une simple calculatrice.

Examiner l'exemple « **C07 : Pers_Activity_3** ».

Faites 2 + 2 +, le résultat sera 4. Faites tourner l'orientation de l'écran, le résultat va rester 4. À noter que le bouton « C » permet de remettre à zéro la calculatrice.

Faites maintenant le test suivant : appuyez sur les touches 2 + 2, puis changez l'orientation de l'écran.



Appuyez maintenant sur la touche + et examinez le résultat :

Le résultat final est égal à 2 et non pas 4! Un bogue!

En changeant d'orientation, l'activité a été arrêtée puis démarrée de nouveau. Durant cette opération, nous avons perdu la valeur de la somme intermédiaire, ce qui explique pourquoi la somme finale affichée est erronée.

Dans le code Java associé à cette activité, la somme intermédiaire est représentée par la variable « total ». Il faudra donc préserver l'état de cette variable au moment de changer d'orientation.

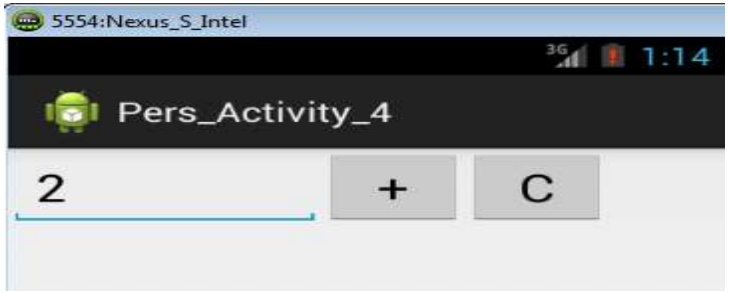
Pour ce faire, nous allons utiliser la méthode « onSaveInstanceState » pour préserver la valeur associée à cette variable au moment de l'arrêt de l'application.

On peut récupérer cette valeur lors du démarrage de l'activité à l'aide de la méthode « onRestoreInstanceState ».

http://developer.android.com/training/basics/activity-lifecycle/recreating.html

```
@Override
public void onSaveInstanceState(Bundle outState) {
     outState.putFloat("TOTAL", total);
     super.onSaveInstanceState(outState); // Il faut toujours le faire
}
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
     super.onRestoreInstanceState(savedInstanceState);
     total=savedInstanceState.getFloat("TOTAL");
}
```

Nous reprenons l'exemple « **C07 : Pers_Activity_3** » et on modifie son code Java. Le résultat est dans l'exemple « **C07 : Pers_Activity_4** ».



On clique sur le signe + :

Le total est maintenant correct.

Lecture supplémentaire :

Examiner le paragraphe « Saving and restoring transient UI state » et le test dans la méthode « onCreate ».

https://developer.android.com/guide/components/activities/activity-lifecycle#saras

3 Préférences partagées

Cette méthode permet de sauvegarder les préférences dans un fichier.

Ces préférences seront accessibles aux différentes activités associées à l'application.

Les préférences seront utilisées pour sauvegarder l'état de la configuration de l'application ou bien les données de session.

Les préférences sont comme les bundles sauf qu'elles sont persistantes ce qui n'est pas le cas des bundles.

Les préférences ne sont pas cryptées. Il faudra faire attention si l'intention est de préserver des données critiques.

Les préférences peuvent être effacées par l'utilisateur de l'application.

Chaque préférence a la forme d'une paire dont la clé est un élément du type String et dont la valeur est un des types primitifs (int, long, float, boolean) ou bien une collection de String (Set<String>).

Cette méthode n'est appropriée que pour une petite collection de paires.

Nous allons utiliser les méthodes de la classe « SharedPreferences » pour sauvegarder puis lire une préférence.

Une instance de la classe « SharedPreferences » est créée dans un mode prédéfini. Le mode le plus couramment utilisé est « MODE_PRIVATE » pour signifier que les préférences ne seront accessibles que par l'application.

Il existe d'autres modes :

- MODE_WORLD_READABLE : les autres applications peuvent lire l'ensemble,
- MODE_WORLD_WRITEABLE : les autres applications peuvent modifier l'ensemble,
- MODE_MULTI_PROCESS : plusieurs processus peuvent accéder à l'ensemble.

Les modes « WORLD_READABLE » et « WORLD_WRITEABLE » sont dépréciés depuis l'API 17. Ils provoqueront l'exception « SecurityException » depuis l'API 24 (Nougat).

Examiner l'exemple « **C07 : PrefsAvecJava** ».

Création :

Nous pouvons utiliser l'une des deux méthodes :

getPreference(int mode) : elle est associée à l'activité courante. Une procédure transparente est définie par défaut pour préserver les préférences désirées et associées à l'activité. Nous devons juste signifier l'argument qui représente l'un des modes d'accès précédemment mentionnés.

```
private SharedPreferences settings =
                getPreference(context.MODE_PRIVATE);
```

getSharedPreferences(String nom,int mode) : elle est utilisée si les préférences sont préservées dans plusieurs fichiers. Le premier argument représente le nom du fichier à utiliser et le second argument, le mode d'accès.

```
private SharedPreferences settings =
    getSharedPreference("nom_preferences",context.MODE_PRIVATE);
```

On édite la préférence:

```
SharedPreferences.Editor editor = settings.edit();
```

On définit la paire « string,string » et on valide l'inscription dans le conteneur des préférences :

```
editor.putString(NOM, PrefValeur);
editor.apply();
```

Pour récupérer la paire :

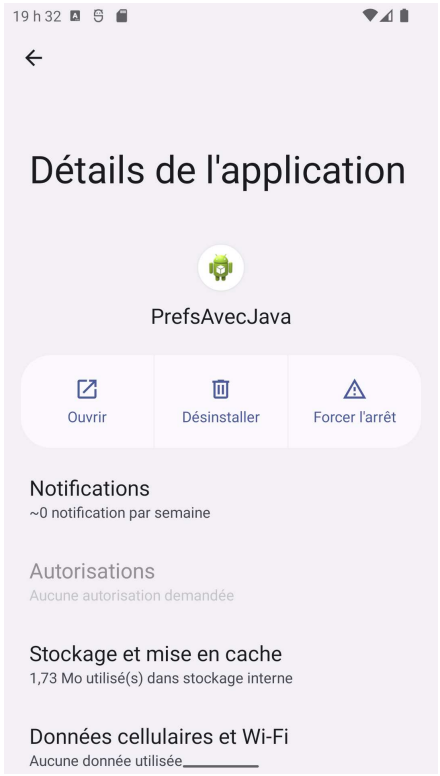
```
PrefValeur = parametres.getString(NOM, "Introuvable");
```

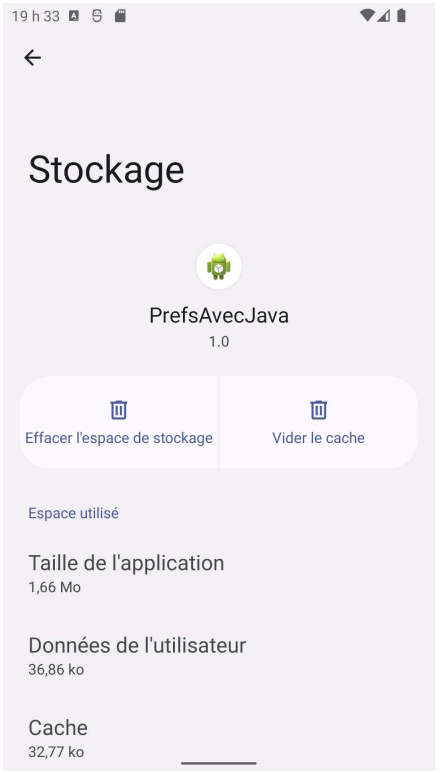
On peut détruire la préférence en détruisant la clé associée à l'aide de la méthode « removeString(String clé) ». On peut détruire toutes les préférences à l'aide de la méthode « clear » :

```
editor.removeString(Nom);
editor.clear();
```

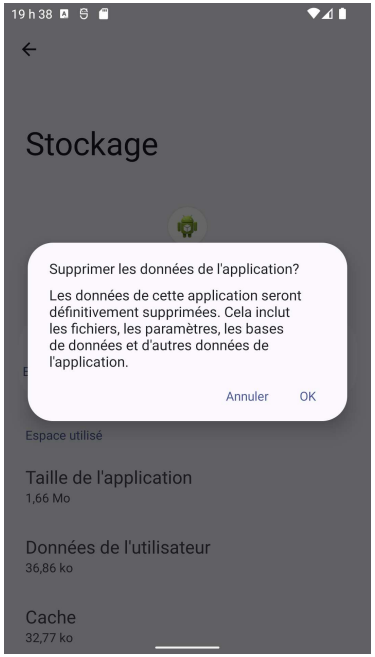
L'utilisateur peut aussi détruire les préférences sauvegardées par son application en procédant ainsi sous « Pie » :

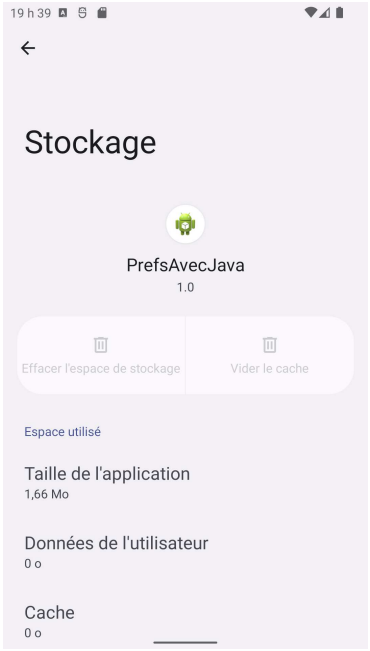
- On se positionne sur le menu des applications.
- On sélectionne notre application et on maintient le bouton gauche de la souris enfoncé. Une nouvelle fenêtre va apparaitre. Cliquer sur « App info ». Vous allez obtenir ce qui suit :





Vous cliquez par la suite sur « Stockage et mise en cache ». Vous pouvez effacer les préférences en cliquant sur le bouton « Effacer l'espace de stockage ».





Une autre manière de procéder est de passer par « settings », puis « Apps », puis sélectionnez votre application « PrefsAvecJava ».

Examiner l'exemple « **C07 : 14-PreferencesDemo0** ».

Les préférences sont sauvegardées dans un fichier. Ce fichier a par défaut le format « XML » et est localisé à :

/data/data/cis493.preferences/shared_prefs/MyPreferences_001.xml

Sur l'émulateur, vous pouvez voir l'existence du fichier grâce à la vue « Device File Explorer ». Vous pouvez aussi extraire le fichier à l'aide de « adb » et examiner son contenu. Il faut avoir les privilèges d'un utilisateur « root » pour pouvoir y avoir accès. Pour obtenir ces privilèges dans le cadre d'un émulateur, exécutez la commande :

```
adb root
restarting adbd as root
adb pull
 /data/data/cis493.preferences/shared_prefs/MyPreferences_001.xml .
```

Choisissez en premier « Pref Simple UI » et examinez par la suite le contenu du fichier « MyPreferences_001.xml » :

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<int name="textSize" value="15" />
<int name="backColor" value="-1" />
</map>
```

Choisissez par la suite « Pref Fancy UI » et examinez le contenu du fichier « MyPreferences_001.xml » :

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<int name="layoutColor" value="-16711936" />
<int name="textSize" value="20" />
<int name="backColor" value="-16776961" />
<string name="textStyle">bold</string>
</map>
```

Cliquer maintenait sur le bouton retour (« Back ») puis examinez le contenu du fichier « MyPreferences_001.xml » :

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <int name="layoutColor" value="-16711936" />
    <string name="DateLastExecution">3 févr. 2018</string>
    <int name="textSize" value="20" />
    <int name="backColor" value="-16776961" />
    <string name="textStyle">bold</string>
</map>
```

Sauvegarde des préférences à travers une activité

L'exemple crée une interface pour gérer les préférences de l'utilisateur.

La source de l'exemple est disponible à cette adresse :
http://androidresearch.wordpress.com/2012/03/09/creating-a-preference-activity-in-android/

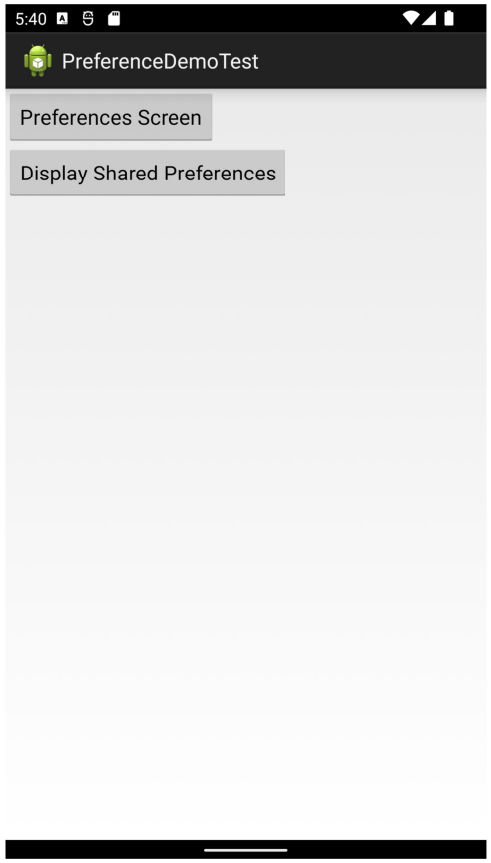
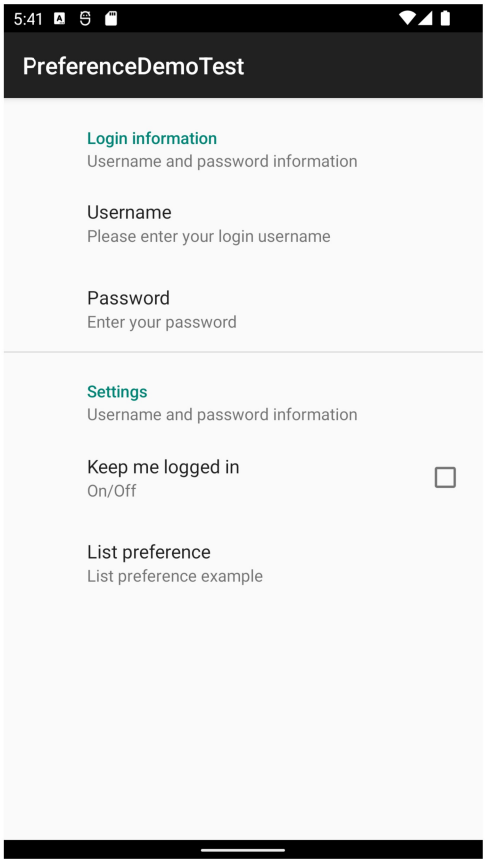
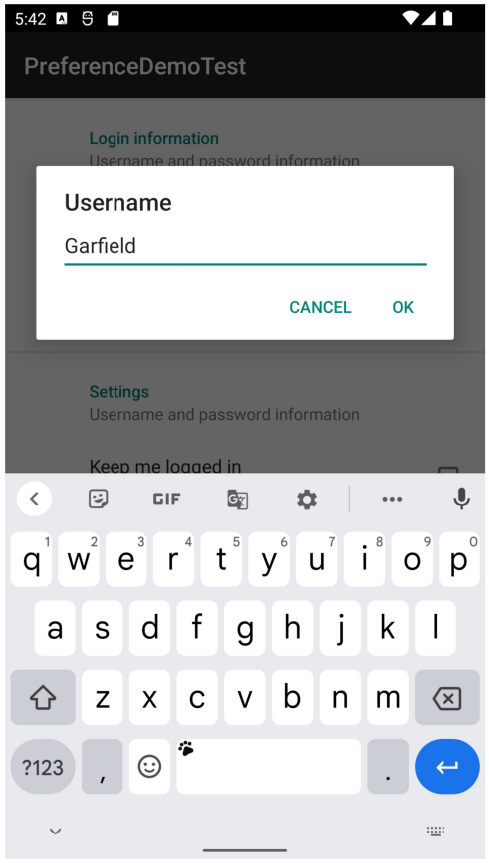
Examiner l'exemple « **C07 : PreferenceDemoTest** ».

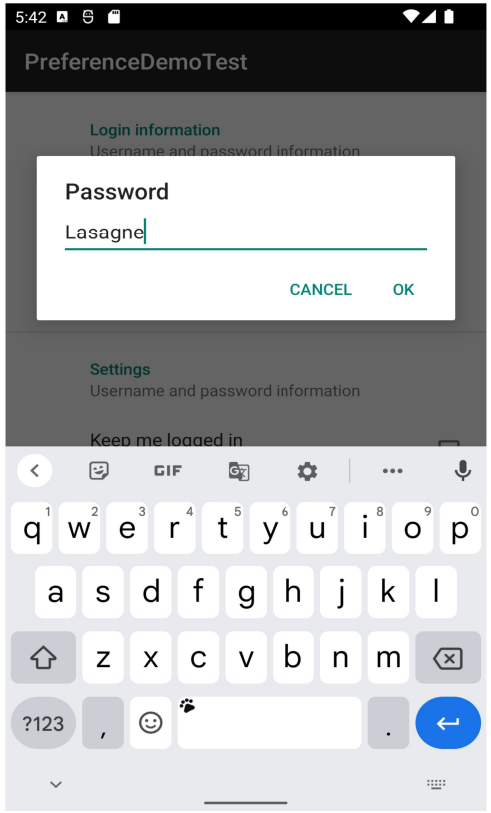
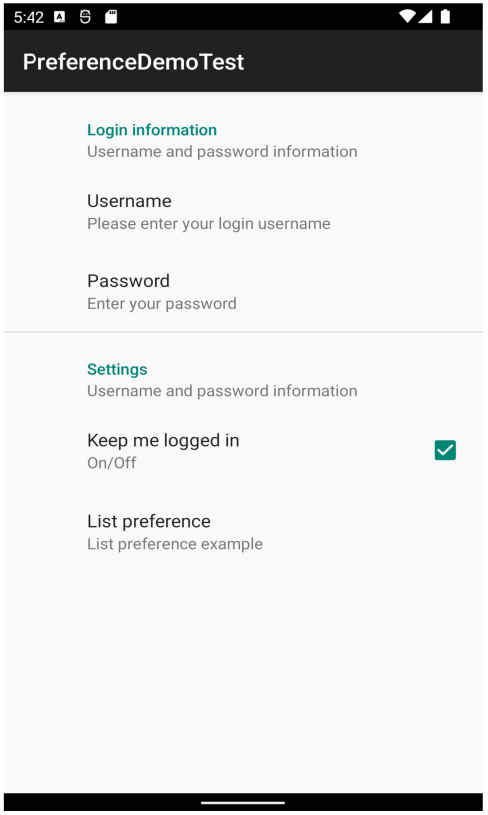
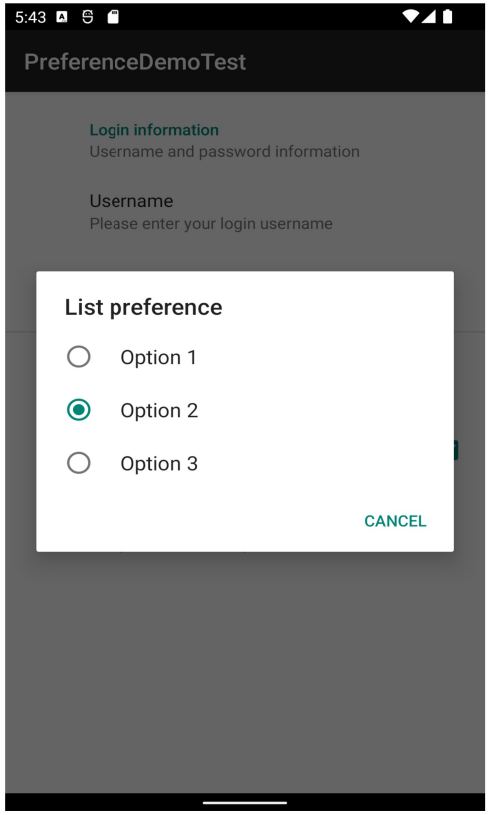
Nous avons dû donc apporter des modifications à cet exemple pour qu'il soit compatible avec l'API 33 (dernière API au moment de mettre à jour ce document).

Nous devons utiliser à la place les méthodes équivalentes définies dans les paquetages :

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.preference.PreferenceFragmentCompat;
```

Nous allons commencer par décrire brièvement l'exemple :

Écran de démarrage	Un clic sur « Preferences Screen »	Un clic sur « Username »
		

Un clic sur « Password »	On coche « Keep me logged in »	Un clic sur « List Preference »
		

L'interface de l'activité principale est définie dans le fichier « activity_preferencedemo.xml ».

Quand on clique sur le bouton « Preferences Screen », ce dernier lance une instance de la classe « PrefsActivity » définie dans le fichier « PrefsActivity.java ».

```
public class PrefsActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportFragmentManager().beginTransaction().\
            replace(android.R.id.content,\
            new PrefsActivity02()).commit();
    }
}
```

La méthode « onCreate » dans la classe « PrefsActivity » va remplacer le conteneur par la vue définie dans le fichier « PrefsActivity02 ».

Cette vue n'est autre que l'interface de la 2e activité. Elle est définie dans le fichier « prefs.xml », qui se trouve dans le répertoire « res/XML ».

Nous avons ajouté dans le fichier « PrefsActivity02.java » cet appel dans la méthode « onCreate » afin de réaliser la liaison avec le fichier « XML ».

```
public class PrefsActivity02 extends PreferenceFragmentCompat {
    @Override
    public void onCreatePreferences(@Nullable Bundle savedInstanceState,\
                                    @Nullable String rootKey) {
        setPreferencesFromResource(R.xml.prefs, rootKey);
    }
}
```

Cette méthode télécharge ainsi les préférences à partir d'une ressource, décrite dans un fichier « XML ». Ce fichier va définir la vue de l'activité.

Examiner le répertoire « xml » dans la hiérarchie du projet.

Dans le fichier « prefs.xml », le tag « PreferenceScreen » sera utilisé comme la racine du fichier. Quand une activité pointe ce fichier, le tag « PreferenceScreen » sera utilisé comme le point d'entrée.

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

Nous distinguons dans ce fichier plusieurs tags :

<PreferenceCategory>	Elle définit une catégorie de préférences. Pour l'exemple, nous avons défini deux catégories « Login Information » et « Settings ».
<EditTextPreference>	Elle définit un champ pour stocker de l'information.
<CheckBoxPreference>	Elle définit une boîte à cocher, « checkbox ».
<ListPreference>	Elle définit une liste d'éléments. La liste apparait comme des boutons radio.

Par la suite, l'activité principale récupère automatiquement les préférences préservées :

```
SharedPreferences prefs =
            PreferenceManager.getDefaultSharedPreferences(
                                        PreferenceDemoActivity.this);
```

4 Fichiers (création et sauvegarde)

Android s'appuie sur l'API Java pour réaliser la gestion de fichiers.

Préférences partagées

Une première utilisation des fichiers était en rapport avec les préférences partagées. Nous avons expliqué comment il était possible de paramétrer la méthode « `getSharedPreference` » pour préserver ces préférences dans un fichier.

Stockage interne

Comme nous l'avons fait depuis le début du cours, nous pouvons ajouter des fichiers supplémentaires à l'application. Ces fichiers peuvent être disposés dans divers endroits en fonction de leur utilisation : « assets », « res/raw », etc.

Ils feront partie du paquetage « apk » final.

Pour l'exemple « **C07 : 15-1-FileEmbeddedResources** », nous allons d'abord préserver le fichier « my_text_file.txt » dans le répertoire « res/raw » de l'application. Par la suite, l'application va se charger de lire le fichier en question et de l'afficher dans la vue principale.

Association d'un flux à la ressource :

```
int fileResourceId = R.raw.my_text_file;
InputStream is = this.getResources()
                        .openRawResource(fileResourceId);
```

Lecture du contenu de la ressource :

```
if (is!=null) {
    BufferedReader reader =
            new BufferedReader(new InputStreamReader(is));
    while ((str = reader.readLine()) != null) {
        buf.append(str).append("\n");
    }
}
```

Comme il s'agit d'une opération d'entrée/sortie (« I/O »), il faudra penser à capturer l'exception « IOException ».

Pour l'exemple « **C07 : 15-2-FileInternalWriteRead** », nous allons réaliser une lecture et une écriture à partir d'un fichier.

Quand l'application démarre pour la première fois, nous allons d'abord écrire un texte dans le fichier « notes.txt » et cliquer sur le bouton de sauvegarde. Ce bouton va terminer l'application, mais avant cela, la méthode « onPause » sera exécutée. Cette méthode aura la tâche de sauvegarder le texte dans le fichier « notes.txt ».

Examiner le contenu du fichier « notes.txt » qui se trouve dans le répertoire :

```
/data/data/cis470.matos.filewriteread/files
```

Par « adb », il faut être « root » pour y accéder au répertoire, sinon il faut utiliser le « Device File Explorer » dans Android Studio.

Quand l'activité est démarrée une seconde fois, la méthode « onStart » est exécutée. Cette méthode va se charger de lire puis d'afficher le contenu du fichier sur l'écran.

Stockage externe

Nous allons examiner comment effectuer des opérations de lecture et écriture à partir d'un média externe. Nous allons utiliser pour cela une carte mémoire SD pour effectuer ces opérations I/O.

Nous allons utiliser l'exemple disponible sur cette page :

http://hoodaandroid.blogspot.ca/2012/07/reading-and-writing-data-to-sdcard.html

Ainsi que, quelques bribes de code disponibles ici :

http://developer.android.com/training/basics/data-storage/files.html

Examiner l'exemple « **C07 : RWSdcard** ».

https://developer.android.com/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE

Depuis l'API 19 (Android KitKat), la permission « WRITE_EXTERNAL_STORAGE » n'est plus nécessaire pour lire et écrire dans des répertoires spécifiques à votre application.

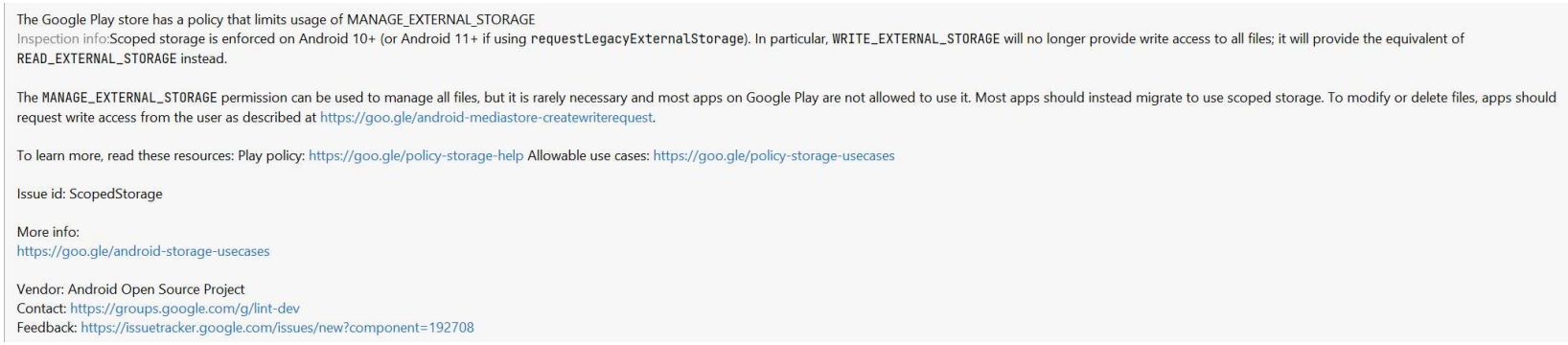
Par contre, il fallait la mentionner dans le manifeste de l'application.

Cependant, il n'est plus nécessaire de mentionner cette permission dans le manifeste des applications qui ciblent l'API 29+ (Android 10+).

La permission « MANAGE_EXTERNAL_STORAGE » permet à l'application de gérer les fichiers de l'utilisateur.

https://developer.android.com/reference/android/Manifest.permission.html#MANAGE_EXTERNAL_STORAGE

Si vous utilisez quand même cette permission, vous allez avoir cet avertissement dans le manifeste de l'application :



Mieux encore …

https://support.google.com/googleplay/android-developer/answer/10467955?hl=en

Use of All files access (MANAGE_EXTERNAL_STORAGE) permission

Google Play restricts the use of high risk or sensitive permissions, including a special app access called All files access. This is only applicable to apps that target Android 11 (API level 30) and declare the `MANAGE_EXTERNAL_STORAGE` permission, which is added in Android 11. Also, this policy does not impact the usage of the `READ_EXTERNAL_STORAGE` permission.

If your app does not require access to the `MANAGE_EXTERNAL_STORAGE` permission, you must remove it from your app's manifest in order to successfully publish your app. Details on policy-compliant alternative implementations are also detailed below.

If your app meets the policy requirements for acceptable use or is eligible for an exception, you will be required to declare this and any other high risk permissions using the Declaration Form in Play Console.

Apps that fail to meet policy requirements or do not submit a Declaration Form may be removed from Google Play.

Comme nous allons utiliser les répertoires par défaut disponibles sur n'importe quel appareil, il n'est pas nécessaire d'aller plus loin.

Nous évitons ainsi de passer par un filtre pour publier l'application.

Android est basé sur un noyau Linux, il faudra vérifier que l'unité externe est bien disponible (l'expression utilisée dans le jargon est « montée »).

```
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

La méthode « `Environment.getExternalStorageState()` » retourne un « MEDIA_MOUNTED » pour nous informer si le média est présent, monté, et dans quel mode, lecture/écriture.

Nous testons par la suite le mode qui a été autorisé avec la méthode « `Environment.MEDIA_MOUNTED.equals` ». Si elle retourne « true » alors l'écriture a été autorisée. Dans le cas contraire, il faudra vérifier si la lecture a été autorisée en procédant ainsi :

```
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Dans notre exemple, nous avons mis en cascade les différents appels comme suit :

```
if(!Environment.MEDIA_MOUNTED.equals
            (Environment.getExternalStorageState())){
    Toast.makeText(this, "External SD card not mounted",
                                    Toast.LENGTH_LONG).show();
}
```

Après avoir validé la disponibilité de l'unité externe, pour préciser le chemin à joindre au nom du fichier, nous allons faire appel à la méthode :

```
« Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS) »
```

On cible ainsi les répertoires publics créés par défaut sur chaque appareil Android.

```
File directory = new File(Environment.getExternalStoragePublicDirectory\
                         (Environment.DIRECTORY_DOCUMENTS) +
                         File.separator + "Data");
```

Le fichier « textfile.txt » va être disponible dans le répertoire :

« /sdcard/Documents/Data/textfile.txt »

Nous pouvons vérifier par la suite avec la commande « adb » la présence du fichier et son contenu :

```
C:\Users\lokbani>adb shell
emu64x:/ $ cd sdcard/Documents/Data/
emu64x:/sdcard/Documents/Data $ cat textfile.txt
Un exemple sans les permissions.
emu64x:/sdcard/Documents/Data $
```

On peut utiliser aussi le « Device File Explorer » pour visualiser le contenu du fichier.

5 SQLite

Pour avoir un aperçu sur les bases de données en général et SQL en particulier, consultez ce document :

http://www.iro.umontreal.ca/~lokbani/cours/ift1176/communs/Cours/PDF/jdbc.pdf

Android intègre le système de gestion de bases de données, SQLite.

Pour plus de détails, consultez ce lien : http://www.sqlite.org/

C'est un système compact, très efficace pour les systèmes embarqués. En effet, il utilise très peu de mémoire.

SQLite ne nécessite pas de serveur pour fonctionner, ce qui n'est pas le cas de MySQL par exemple.

Les opérations sur la base de données se feront donc dans le même processus que l'application. Il faudra faire attention aux opérations « lourdes », votre application va ressentir les contres coups. Il est conseillé dans ce cas d'utiliser les tâches asynchrones (ou threads).

Chaque application peut avoir donc ses propres bases.

Ces bases sont stockées dans le répertoire « databases » associé à l'application (`/data/data/APP_NAME/databases/nom_base`). Nous pouvons les stocker aussi sur une unité externe (sdcard).

Chaque base créée, elle le sera en mode « MODE_PRIVATE ». Aucune autre application ne peut y accéder que l'application qui l'a créée.

Pour y avoir accès, il faut que la base ait été sauvegardée sur un support externe, sinon utiliser le mécanisme d'échange de données fourni par Android (il sera développé plus tard dans ce chapitre).

L'accès par « adb » nécessite les privilèges « root ».

SQLite supporte les types : TEXT (chaîne de caractères), INTEGER (entiers), REAL (réels). Tous les types doivent être convertis pour être utilisés. SQLite ne vérifie pas le typage des éléments. À vous de vous en assurer que vous n'avez pas écrit un entier à la place d'une chaîne de caractères par exemple.

Création et mise à jour de la base

Nous avons adapté l'exemple développé sur cette page :

http://www.vogella.com/articles/AndroidSQLite/article.html

Examiner l'exemple « **C07 : Testbdd** ».

Vu que ListActivity est dépréciée depuis l'API 30, nous l'avons remplacée par RecyclerView.

L'exemple va créer une table de commentaires. Chaque commentaire est identifié par un identificateur unique.

Nom de la table : `comments`	
`_id`	`comments`

La base va porter le nom « `commments.db` ».

L'organisation des fichiers permet de faciliter l'organisation de la base de données et la compréhension de l'exemple.

- Le fichier « `Comment.java` » va contenir un enregistrement d'une table et les différentes méthodes qui gravitent autour.

La classe « Comment » décrite dans le fichier « Comment.java » contient deux attributs :

```
private long id;
private String comment;
```

La base de données doit utiliser un identifiant unique « _id » comme clé primaire de la table. Des méthodes d'Android se servent de ce standard.

- Le fichier « MySQLiteHelper.java » contient la classe qui dérive de « SQLiteOpenHelper ».

La classe « MySQLiteHelper » :

Créez une nouvelle classe qui va dériver de la classe « SQLiteOpenHelper » :

```
public class MySQLiteHelper extends SQLiteOpenHelper { …}
```

Dans le constructeur de la classe, faites appel à la méthode « super » de « SQLiteOpenHelper » et spécifiez le nom de la base et sa version.

```
public MySQLiteHelper(Context context) {
    super(context, "commments.db", null, 1);
}
```

Dans cette classe, vous devez redéfinir les méthodes « onCreate(SQLiteDatabase MaBase) » et « onUpgrade(SQLiteDatabase MaBase) ». L'argument représente votre base.

La méthode « onCreate » est appelée pour la création de la base si elle n'existe pas.

```
public void onCreate(SQLiteDatabase database) {
    database.execSQL(DATABASE_CREATE);
}
```

La variable « *DATABASE_CREATE* » va contenir la requête « SQL » qui permet de créer la base.

La méthode « onUpgrade » est appelée pour mettre à jour la version de votre base. Elle vous permet de mettre à jour le schéma de votre base.

```
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                                                    int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS " + "comments");
    onCreate(db);
}
```

Il est préférable de créer une classe par table. Cette classe va définir les méthodes « onCreate » et « onUpgrade ». Vous allégez ainsi le code de la classe qui dérive de « SQLiteOpenHelper ».

- Le fichier « CommentsDataSource.java » contient la classe contrôleur. Elle contient les différentes méthodes qui vont interagir avec la base de données. C'est le DAO (Data Access Object)

La classe « SQLiteOpenHelper » fournit les deux méthodes « getReadableDatabase() » et « getWriteableDatabase() » pour accéder à une instance « SQLiteDatabase » en mode de lecture ou écriture.

Ouverture de la base :

```
public void open() throws SQLException {
    database = dbHelper.getWritableDatabase();
}
```

Fermeture de la base :

```
public void close() {
    dbHelper.close();
}
```

Insérer un élément :

Pour insérer un élément dans la base, il faut d'abord former l'enregistrement. On utilise pour cela un objet du type « ContentValues » qui représente une collection de champs.

```
public long createComment(String comment) {
    ContentValues values = new ContentValues();
    values.put(MySQLiteHelper.COLUMN_COMMENT, comment);
    long insertId = database.insert(MySQLiteHelper.TABLE_COMMENTS,
                                                    null, values);
    return insertId;
}
```

Effacer un élément :

```
public void deleteComment(Comment comment) {
    long id = comment.getId();
    database.delete(MySQLiteHelper.TABLE_COMMENTS,
                MySQLiteHelper.COLUMN_ID     + " = " + id, null);
}
```

Faire une sélection :

```
    Cursor cursor = database.query(MySQLiteHelper.TABLE_COMMENTS,
            allColumns, MySQLiteHelper.COLUMN_ID + " = " + insertId,
                                                null, null, null, null);
```

La méthode « query » retourne une instance de « Cursor » qui représente un ensemble de résultats.

http://developer.android.com/reference/android/database/Cursor.html

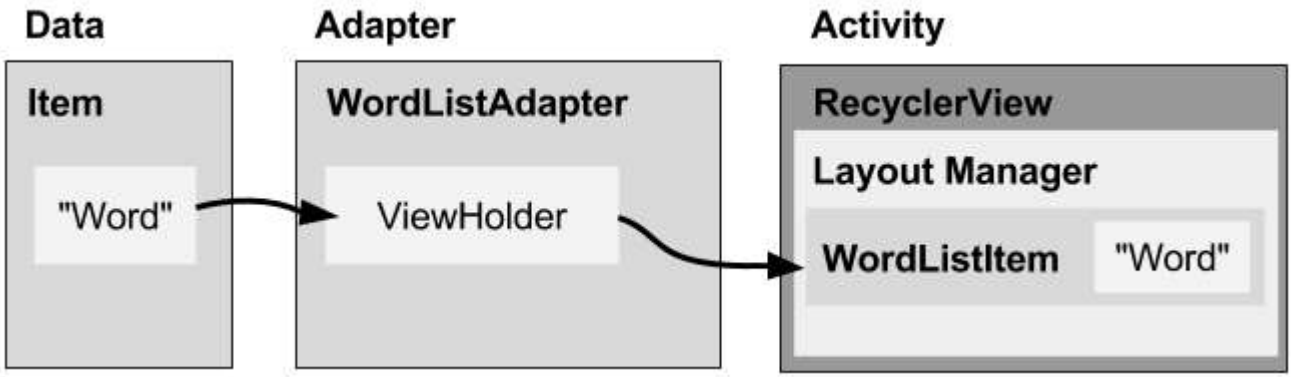
- Le fichier « `TestDatabaseActivity.java` » contient l'activité associée à notre application.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_testdatabase);

    datasource = new CommentsDataSource(this);
    datasource.open();

}

protected void onResume() {
    super.onResume();
    datasource.open();
}

protected void onPause() {
    super.onPause();
    datasource.close();
}
```

Le fichier « AdapterClass.java » représente la vue des informations à afficher sur l'interface utilisateur à l'aide d'un « RecyclerView »

Chaque élément est défini par son objet « view holder »

Au départ, l'objet est vide. Au moment de sa création, « RecyclerView » l'associe à la donnée.



Accès à la base de données SQLite

Le SDK d'Android inclut un programme permettant de lire une base de données SQLite.

Nous devons extraire le fichier de l'émulateur via la commande « adb/shell/pull » en tant que « root » ou bien en utilisant la vue « Device File Explorer », puis la commande « pull » :

```
adb pull
   /data/data/ca.umontreal.iro.ift1155.testbdd/databases/comments.db
```

```
C:> sqlite3 comments.db
SQLite version 3.40.0 2022-11-16 12:10:08
Enter ".help" for usage hints.
sqlite> .tables
android_metadata  comments
sqlite> select * from comments;
1|Hate it
2|Cool
3|Very nice
4|Hate it
5|Very nice
6|Cool
sqlite> .exit
```

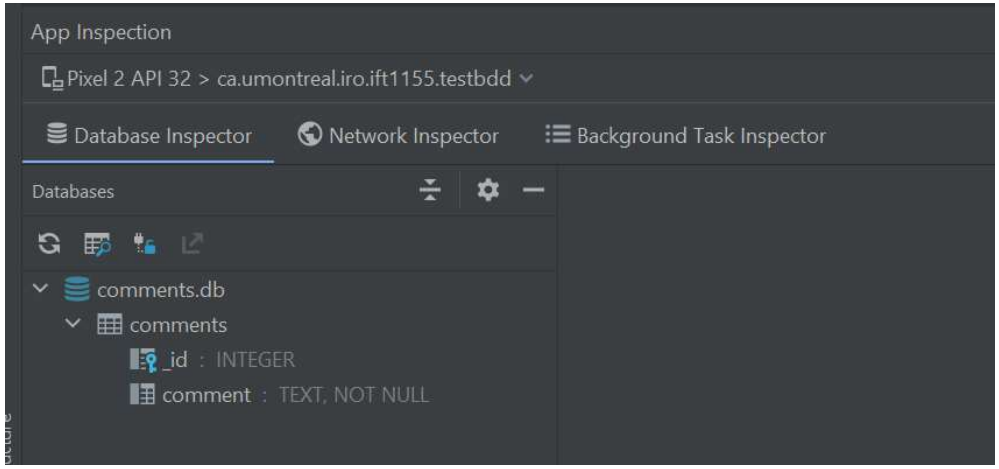
En tant que « root », il est possible aussi d'y avoir accès par la commande « adb » comme suit :

```
C:\Users>adb root
restarting adbd as root

C:\Users>adb shell
emu64x:/ # cd /data/data/ca.umontreal.iro.ift1155.testbdd/databases/
emu64x:/data/data/ca.umontreal.iro.ift1155.testbdd/databases # sqlite3 comments.db
SQLite version 3.32.2 2021-07-12 15:00:17
Enter ".help" for usage hints.
sqlite> .tables
android_metadata  comments
sqlite> select * from comments;
4|Very nice
5|Hate it
6|Hate it
7|Very nice
8|Cool
sqlite> .exit
emu64x:/data/data/ca.umontreal.iro.ift1155.testbdd/databases #
```

Android Studio contient un outil permettant de visualiser une base de données SQLite.

Commencer par lancer votre application dans un émulateur.

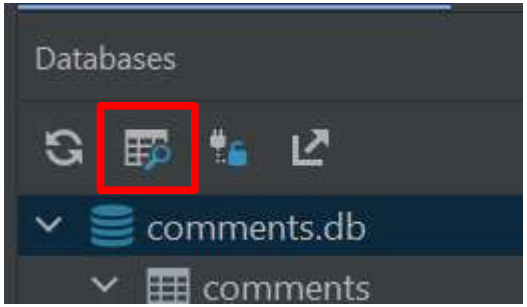
« View », « Tool Windows », « App Inspection » :

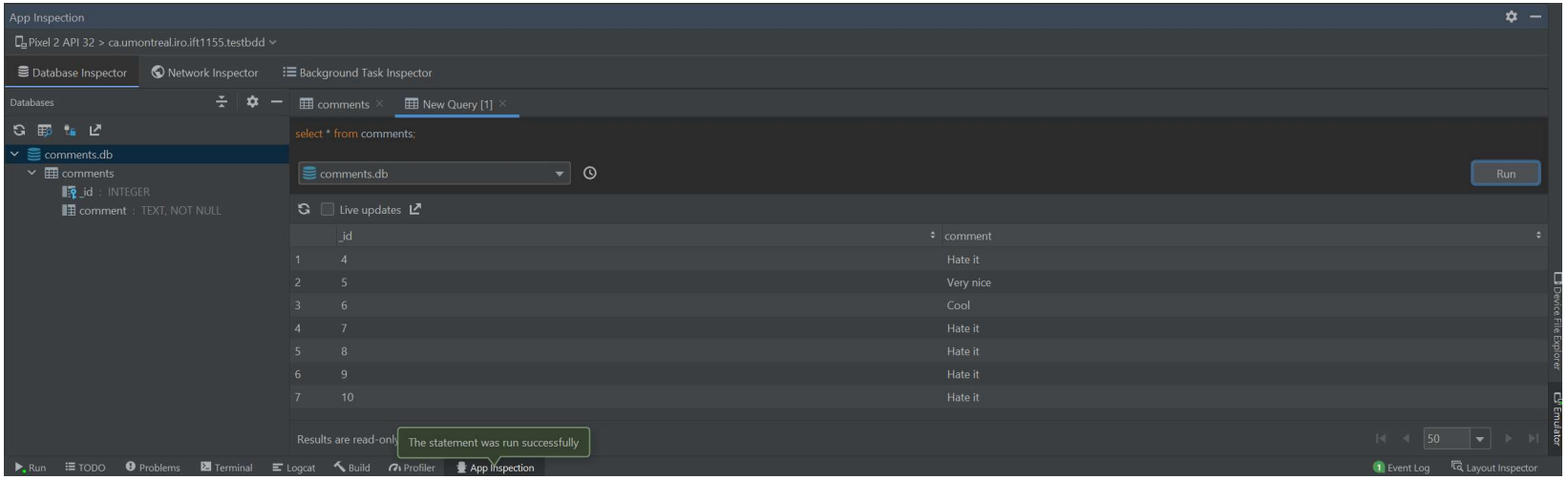


Double cliquer sur le nom de la base « comments » afin de visualiser son contenu :

On peut cocher la case « Live updates » afin de visualiser en direct les différents changements apportés à la base.

On peut exécuter une requête SQL en cliquant sur « Open New Query Tab » :





Il est possible aussi d'utiliser d'autres outils externes pour visualiser la base de données :

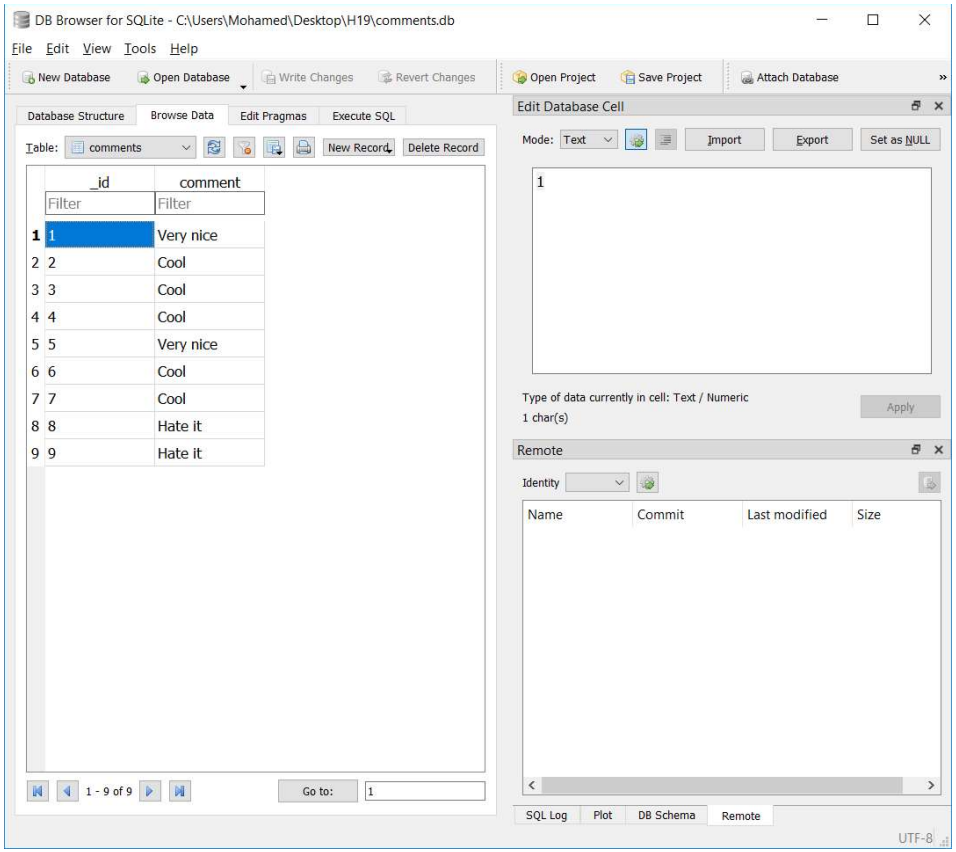
Un adds-on dans Firefox pour accéder à la base de données :

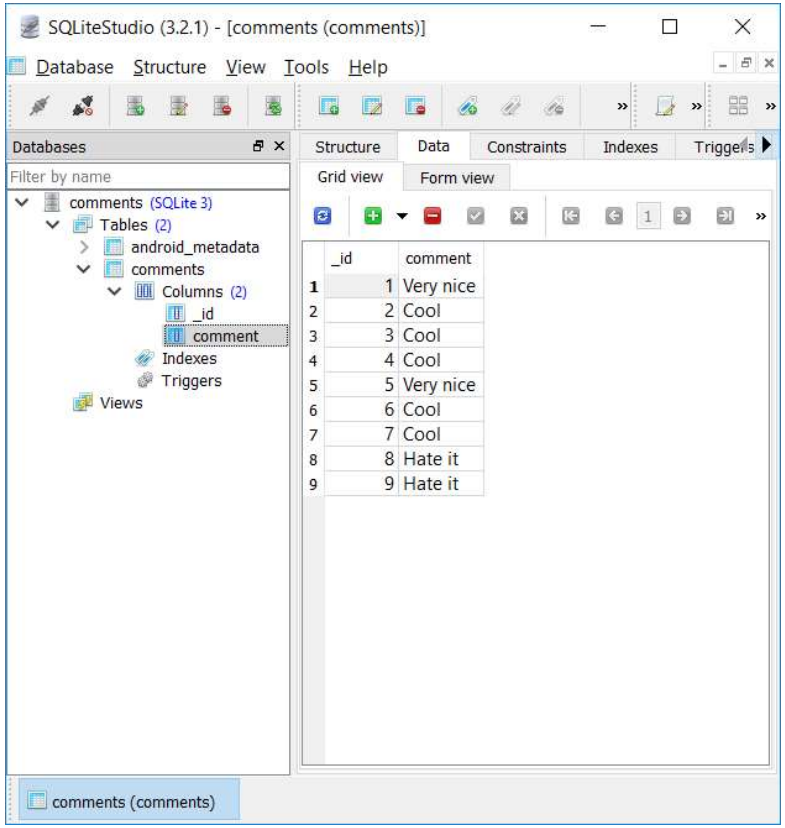
https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager-webext/

Comme il est possible aussi d'utiliser un de ces deux logiciels :

https://sqlitebrowser.org/

https://sqlitestudio.pl/


DB Browser for SQLite - C:\Users\Mohamed\Desktop\H19\comments.db
File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Table: comments
New Record
Delete Record
_id
comment
Filter
1 1 Very nice
2 2 Cool
3 3 Cool
4 4 Cool
5 5 Very nice
6 6 Cool
7 7 Cool
8 8 Hate it
9 9 Hate it
1 - 9 of 9
Go to: 1
Edit Database Cell
Mode: Text
Import
Export
Set as NULL
1
Type of data currently in cell: Text / Numeric
1 char(s)
Apply
Remote
Identity
Name
Commit
Last modified
Size
SQL Log
Plot
DB Schema
Remote
UTF-8


SQLiteStudio (3.2.1) - [comments (comments)]
Database Structure View Tools Help
Databases
Filter by name
comments (SQLite 3)
Tables (2)
android_metadata
comments
Columns (2)
_id
comment
Indexes
Triggers
Views
Structure
Data
Constraints
Indexes
Triggers
Grid view
Form view
_id
comment
1 1 Very nice
2 2 Cool
3 3 Cool
4 4 Cool
5 5 Very nice
6 6 Cool
7 7 Cool
8 8 Hate it
9 9 Hate it
comments (comments)

6 Fournisseur de contenu (Content Provider)

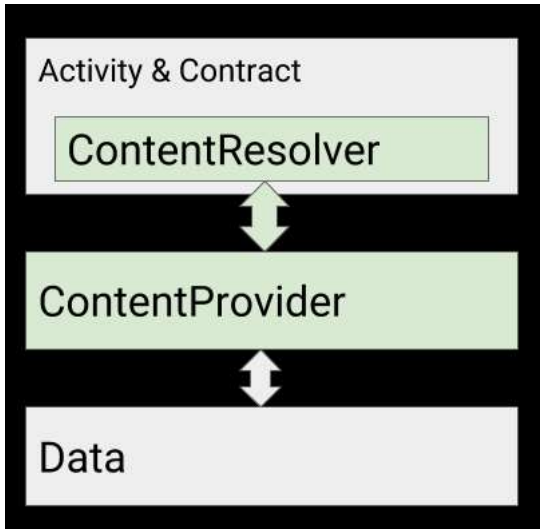
Dans les précédents exemples, nous avons mentionné que les données n'étaient pas accessibles aux autres applications.

Et si, nous aurions aimé partager quand même le carnet d'adresses entre plusieurs applications?

Android offre un mécanisme permettant à une application à accéder aux données d'une autre application.

Ce mécanisme porte le nom de « fournisseur de contenu » ou « Content Provider ».

C'est l'interface qui connecte les données associées à un processus avec le code exécuté par un autre processus.



Data / Données : elles sont générées par l'application.

ContentProvider / Fournisseur de contenu : une interface publique et sécuritaire pour accéder aux données.

ContentResolver / Résolveur de contenu : est utilisé par l'application pour établir des requêtes au « ContentProvider ». Il fournit les méthodes « query », « insert », « update » et « delete » pour la gestion et l'interaction avec le « ContentProvider ».

La réponse est sous la forme d'un objet du type « cursor ». On expose par la suite cette réponse à l'aide d'un adaptateur.

Peu importe comment les données sont stockées, un fournisseur de contenu fournit une interface uniforme pour accéder à ces données.

Les données sont exposées sous forme tabulaire.

Les lignes représentent les enregistrements, les colonnes représentent les attributs.

Chaque enregistrement est identifié par un identificateur unique et représente la clé d'entrée vers l'enregistrement.

Utiliser un fournisseur de contenu, consiste à se servir d'une boîte noire. Le plus important est de savoir comment récupérer l'information de la boîte et non pas comment la boîte a été fabriquée.

Je veux par exemple récupérer la liste des contacts, peu importe comment le système a fait pour regrouper et sauvegarder ces contacts.

Chaque fournisseur de contenu est identifié par un URI (Uniform Resource Identifier) unique.

URI

La forme de l'URI est comme suit : content://nompaquetage.provider/comments/2

« content » pour signifier qu'il s'agit d'un fournisseur de contenu, et non pas le protocole « https » par exemple.

« nompaquetage.provider » représente l'autorité. Elle permet d'identifier le fournisseur de contenu.

« comments », le nom de la table. Il n'y a pas de limite sur le nombre de tables utilisées.

« 2 », le 2e enregistrement dans la table.

Android contient un ensemble de fournisseurs de contenu natifs, destinés à gérer des données du type audio, vidéo, image, etc.

```
Browser
CallLog
Contacts
     People
     Phones
     Photos
     Groups
MediaStore
     Audio
          Albums
          Artists
          Genres
          Playlists
     Images
          Thumbnails
     Video
Settings
```

L'URI a été utilisé si on veut accéder aux fournisseurs de contenu natifs :

Pour les signets d'un navigateur …

```
android.provider.Browser.BOOKMARKS_URI
```

Pour les contacts d'un agenda …

```
ContactsContract.contacts.CONTENT_URI
```

Méthodes

Pour accéder à un fournisseur de contenu, nous utilisons la classe abstraite « android.content.Contentresolver ».

Une instance de cette classe peut-être obtenue par l'appel :

```
ContentResolver contentResolver=getContentResolver();
```

Les principales méthodes de la classe « ContentResolver » sont : « query » pour faire une requête à la base, « insert » pour insérer un élément, « update » pour mettre à jour un élément, « delete » pour effacer un élément et « getType » pour récupérer le type MIME de l'élément. On constate la similarité avec les méthodes de la base de données SQLite.

Examiner l'exemple « **C07 : ContactsView** ».

La source de l'exemple est disponible à cette adresse :

http://www.vogella.com/articles/AndroidSQLite/article.html

Pour ce premier exemple, nous allons interagir avec la base des contacts enregistrés sur notre appareil. Si elle ne contient aucun contact, pensez à en ajouter!

L'URI utilisé :

```
Uri uri = ContactsContract.Contacts.CONTENT_URI;
```

Il fallait permettre à notre application à avoir accès à la base des contacts dans le fichier « AndroidManifest.xml » :

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

Ne pas oublier d'autoriser la permission manuellement dans l'émulateur ou l'appareil.

```
private Cursor getContacts() {
     // L'URI pour le contenu à récupérer …
     Uri uri = ContactsContract.Contacts.CONTENT_URI;

     // La liste des colonnes à récupérer …
     String[] projection = new String[] {
          ContactsContract.Contacts._ID, ContactsContract.Contacts.DISPLAY_NAME };

     // La liste des lignes à récupérer …
     String selection = ContactsContract.Contacts.IN_VISIBLE_GROUP + " = '" +\
                                                          ("1") + "'";

     // Dans quel ordre, nous voulons récupérer les résultats.
     // Nous avons décidé de trier les résultats dans un ordre décroissant.
     String sortOrder = ContactsContract.Contacts.DISPLAY_NAME +\
                                                       " COLLATE LOCALIZED ASC";

     // On envoie la requête avec l'ensemble des paramètres.
     return getContentResolver().query(uri, projection, selection, null, sortOrder);
}
```

Dans le deuxième exemple, nous allons définir notre fournisseur d'accès.

`Examiner l'exemple « C07 : EssaiContentProvider ».`

La source de l'exemple (avec une légère modification pour corriger un avertissement relatif à une méthode obsolète depuis l'API 11).

http://www.tutorialspoint.com/android/android_content_providers.htm

- La classe « EssaiContentProvider » contient l'activité. Elle définit deux boutons : ajouter un élément dans la base, afficher le contenu de la base.

Ajouter un élément

```
public void onClickAddName(View view) {

   // On commence par définir le conteneur …
   ContentValues values = new ContentValues();

   // On ajoute le nom et le grade de la personne, en récupérant ces informations
   // à partir de l'interface de l'application …
   values.put(StudentsProvider.NAME,
       ((EditText)findViewById(R.id.txtName)).getText().toString());
   values.put(StudentsProvider.GRADE,
        ((EditText)findViewById(R.id.txtGrade)).getText().toString());

   // On ajoute l'élément …
   Uri uri = getContentResolver().insert(
        StudentsProvider.CONTENT_URI, values);

  // On affichje un toast si tout va bien …
  if (uri != null) Toast.makeText(getBaseContext(), uri.toString(),
                          Toast.LENGTH_LONG).show();
}
```

Afficher le contenu de la base

```
public void onClickRetrieveStudents(View view) {
   // URL vers la base …
   String URL = "content://com.example.provider.College/students";
   Uri students = Uri.parse(URL);

   // On fait la demande …
   try (Cursor c = getContentResolver().query(students, null, null, null,
                                                                "name")) {
     // On boucle sur le résultat et on l'affiche à travers un « toast » …
     if (c!= null && c.moveToFirst()) {
           do {
              String alpha =
                   c.getString(c.getColumnIndexOrThrow(StudentsProvider._ID))
                   + ", " +
                   c.getString(c.getColumnIndexOrThrow(StudentsProvider.NAME))
                   + ", " +
                   c.getString(c.getColumnIndexOrThrow(StudentsProvider.GRADE));

                   Toast.makeText(this, alpha,Toast.LENGTH_LONG).show();
           } while (c.moveToNext());
     }
   }
}
```

- La classe « StudentsProvider », c'est elle qui va définir notre fournisseur de contenu. Elle va dériver de la classe « ContentProvider » :

```
public class StudentsProvider extends ContentProvider { … }
```

Elle va définir les deux valeurs :

```
static final String PROVIDER_NAME = "com.example.provider.College";
static final String URL = "content://" + PROVIDER_NAME + "/students";
```

Nous allons redéfinir dans la classe « StudentsProvider », les deux méthodes « onCreate » et « getType ».

```
private DatabaseHelper dbHelper;


 void open() throws SQLException {
   db = dbHelper.getWritableDatabase();
 }

 public boolean onCreate() {
   Context context = getContext();
   dbHelper = new DatabaseHelper(context);
   /**
    * Create a write able database which will trigger its
    * creation if it doesn't already exist.
    */
   open();
   return (db!= null);
 }
```

Le programme utilise les propriétés « UriMatcher » :

http://developer.android.com/reference/android/content/UriMatcher.html

```
@Override
public String getType(Uri uri) {
   switch (uriMatcher.match(uri)){

   // Obtenir toutes les informations de « student »
   case STUDENTS:
      return "vnd.android.cursor.dir/vnd.example.students";

   // Obtenir juste l'ID
   case STUDENT_ID:
      return "vnd.android.cursor.item/vnd.example.students";
   default:
      throw new IllegalArgumentException("Unsupported URI: " + uri);
   }
}
```

Nous devons déclarer le fournisseur de contenu dans le fichier « AndroidManifest.xml » :

```
<provider android:name="StudentsProvider"
    android:authorities="com.example.provider.College">
</provider>
```

Suite à cette déclaration, vous avez cet avertissement :

[Exported content providers can provide access to potentially sensitive data. Content providers are exported by default and any application on the system can potentially use them to read and write data. If the content provider provides access to sensitive data, it should be protected by specifying export=false in the manifest or by protecting it with a permission that can be granted to other applications.]

```
<provider android:name="StudentsProvider"
    android:exported="false"
    android:authorities="com.example.provider.College">
</provider>
```

https://developer.android.com/training/articles/security-tips.html

7 Stockage réseau

Il y a plusieurs techniques :

- Utiliser la méthode « POST » pour pousser des données sur le réseau, nous avons mentionné cette technique dans le chapitre « Internet ».

- On peut utiliser l'API Google pour sauvegarder des préférences dans le dépôt Google (dans le nuage / in the Cloud).

- Pour cela, nous avons deux approches : l'utilisation d'une paire clé/valeur valable pour toutes les API ≥ 8 ou l'utilisation d'une sauvegarde automatique uniquement pour les API ≥ 23.

  https://developer.android.com/guide/topics/data/testingbackup.html

- Le lien suivant explique les différences entre les deux approches :

  https://developer.android.com/guide/topics/data/keyvaluebackup.html

- Nous n'avons pas besoin d'enregistrer la sauvegarde automatique dans le service de sauvegarde d'Android :

  https://developer.android.com/google/backup/index.html

L'exemple suivant « **`C07 : PreferencesBackupTut`** » utilise l'approche clé/valeur.

L'exemple est développé sur cette page :

http://blog.blundell-apps.com/backup-sharedpreferences-to-the-cloud/

Vous pouvez examiner en détail cet exemple dans la version « 1.09 » de ce chapitre.

Nous allons étudier dans ce qui suit, l'approche d'une sauvegarde automatique. Nous allons reprendre l'exemple « **`C07 : PreferenceDemoTest`** » que nous avons renommé pour les circonstances « **`C07 : Backupcloud`** ». Cet exemple sauvegarde des préférences dans le fichier par défaut :

« **`ca.umontreal.iro.ift1155.backupcloud_preferences.xml`** ».

Le scénario de sauvegarde est comme suit :

- Nous allons sauvegarder l'application dans « Google Drive ».
- Nous allons par la suite détruire le fichier de préférences.
- Finalement, restaurer ce fichier à partir de la sauvegarde.

Attention : il faut être d'abord connecté à votre compte Google avant de demander une sauvegarde.

Lors d'une sauvegarde automatique, cette opération est transparente pour le détenteur d'un appareil. Il suffit que l'appareil soit paramétré ainsi pour que la sauvegarde puisse avoir lieu.

Nous allons examiner la même opération de façon manuelle. Pour cela, nous allons utiliser la commande « bmgr » pour « Backup Manager ». On commence par autoriser la sauvegarde dans le manifeste de l'application :

```
android:allowBackup="true"
```

Par la suite, on active le service de sauvegarde sur l'appareil :

```
adb shell bmgr enable true
Backup Manager now enabled
```

On vérifie que les services nécessaires pour la sauvegarde sont actifs :

```
adb shell bmgr list transports
  android/com.android.internal.backup.LocalTransport
  com.google.android.gms/.backup.migrate.service.D2dTransport
  * com.google.android.gms/.backup.BackupTransportService
```

Dans le cas contraire, vous allez avoir ce message :

```
adb shell bmgr list transports
No transports available.
```

Nous allons sauvegarder l'application avec la commande :

```
adb shell bmgr fullbackup ca.umontreal.iro.ift1155.backupcloud
```

```
Performing full transport backup
```

On examine les logs avec la commande :

```
adb logcat|grep backup
```

S'il n'y a rien, voir plus loin dans ce document comment forcer une sauvegarde.

On obtient ce qui suit :

```
02-11 16:28:56.423 1382 1959 D BackupManagerService: fullTransportBackup()

02-11 16:28:56.432 1382 9332 I PFTBT : Initiating full-data transport backup of ca.umontreal.iro.ift1155.backupcloud
02-11 16:28:56.433 4248 4259 I Backup : [GmsBackupTransport] Attempt to do full backup on ca.umontreal.iro.ift1155.backupcloud

02-11 16:28:56.528 4248 4259 I Backup : [FullBackupWrapper] create full backup for : ca.umontreal.iro.ift1155.backupcloud

02-11 16:28:56.631 4614 4627 I file_backup_helper: Name: apps/ca.umontreal.iro.ift1155.backupcloud/sp/ca.umontreal.iro.ift1155.backupcloud_preferences.xml
```

Si la sauvegarde n'a pas été réalisée, c'est parce qu'en réalité, elle va avoir lieu plus tard dans la journée ou la nuit. Pour forcer la main à le faire « maintenant », exécuter cette commande :

```
adb shell bmgr backupnow ca.umontreal.iro.ift1155.backupcloud

Running incremental backup for 1 requested packages.
Package @pm@ with result: Success
Package ca.umontreal.iro.ift1155.backupcloud with progress: 512/1024
Package ca.umontreal.iro.ift1155.backupcloud with progress: 2560/1024
Package ca.umontreal.iro.ift1155.backupcloud with result: Success
Backup finished with result: Success
```

On efface le fichier de préférences qui se trouve dans le répertorie :

`/data/data/ca.umontreal.iro.ift1155.backupcloud/shared_prefs`

« **`ca.umontreal.iro.ift1155.backupcloud_preferences.xml`** ».

Il faut avoir les droits de « root » pour effacer le fichier. Si l'appareil le permet, utiliser la même approche que pour un émulateur. Dans le cas contraire, faire ce qui suit :

```
adb shell
run-as  ca.umontreal.iro.ift1155.backupcloud
cd /data/data/ca.umontreal.iro.ift1155.backupcloud/shared_prefs
rm  ca.umontreal.iro.ift1155.backupcloud_preferences.xml
```

On vérifier que le fichier a été effectivement effacé.

On restaure l'application avec la commande :

adb shell bmgr restore token ca.umontreal.iro.ift1155.backupcloud

Pour obtenir le token, faire :

```
adb shell dumpsys backup
```

Et chercher ces deux champs :

```
Ancestral: 0
Current:   44d4e642184f21gh (valeur fictive)
```

adb shell bmgr restore 44d4e642184f21gh ca.umontreal.iro.ift1155.backupcloud

```
Scheduling restore: sdk_gphone64_x86_64
restoreStarting: 1 packages
onUpdate: 0 = ca.umontreal.iro.ift1155.backupcloud
restoreFinished: 0
done
```

```
02-11 16:31:10.886 1382 1423 V BackupManagerService: beginRestoreSession:
pkg=ca.umontreal.iro.ift1155.backupcloud transport=null

02-11 16:31:10.891 1382 2194 V RestoreSession: restorePackage
pkg=ca.umontreal.iro.ift1155.backupcloud
obs=android.app.backup.IRestoreObserver$Stub$Proxy@11c876a

02-11 16:31:10.891 1382 2194 V RestoreSession: restorePackage
pkg=ca.umontreal.iro.ift1155.backupcloud token=3124dd633a9e564e
```

Vérifier que le fichier est réapparu avec les préférences sauvegardées.

Débogage :

On affiche tous les paramètres relatifs à la sauvegarde :

```
adb shell dumpsys backup
```

```
Backup Manager is enabled / provisioned / not pending init
Auto-restore is enabled
Last backup pass started: 1486843666300 (now = 1486843545202)
next scheduled: 1486858290199
```

Les temps indiqués sont en millisecondes et ils sont calculés par rapport à l'heure Unix :

https://fr.wikipedia.org/wiki/Heure_Unix

Le site web suivant permet de réaliser la conversion vers le temps « local » :

http://www.epochconverter.com/

Sinon, on peut utiliser la commande Linux suivante :

```
date -d @1486843489 <=> Sat Feb 11 15:04:49 EST 2017
```

Le temps fourni doit-être en secondes. Dans le nombre affiché, on enlève donc les 3 derniers chiffres : `1486843545202 => 1486843545`

```
1486843545 : Sat, Feb 11, 2017  3:05:45 PM
```

```
1486843666 : Sat, Feb 11, 2017  3:07:46 PM
```

La dernière fois que le paquetage a été sauvegardé :

```
1486843489300 : ca.umontreal.iro.ift1155.backupcloud
```

Il est possible d'exclure des fichiers de la sauvegarde :

```
android:fullBackupContent="@xml/fichier_exclusion"
```

Ce fichier doit-être défini dans le répertoire « res/XML » de l'application. Il contient par exemple ce qui suit :

```
<?xml version="1.0" encoding="utf-8"?>
<full-backup-content>
    <include domain="sharedpref" path="."/>
    <exclude domain="sharedpref" path="device.xml"/>
</full-backup-content>
```

L'exemple « backupcloud » inclut le fichier d'exclusion « backup_rulez.xml ».

Depuis API 31+, Android crée automatiquement un fichier de règles pour la sauvegarde et la restauration.

https://developer.android.com/guide/topics/data/autobackup#IncludingFiles

If your app targets Android 12 (API level 31) or higher, you must specify an additional set of XML backup rules to support the changes to backup restore that were introduced for devices running Android 12 or higher.

Rien ne vous empêche d'utiliser ce fichier pour ajouter vos règles ou bien d'en créer un autre.

Dans le fichier manifeste de l'application :

```
android:fullBackupContent="@xml/backup_rulez"
android:dataExtractionRules="@xml/data_extraction_rules"
```