5. Persistance des données

5.1 Introduction

Android fournit plusieurs méthodes pour faire persister les données applicatives:

- la persistance des données de l'activité (cf Le SDK Android)
- un mécanisme de sauvegarde clé/valeur, utilisé pour les fichiers de préférences (appelé préférences partagées)
- des entrées sorties de type fichier
- une base de donnée basé sur SQLite

La persistance des données des activités est géré par un objet **Bundle** qui permet de restaurer les **View** qui possèdent un *id*. S'il est nécessaire de réaliser une sauvegarde plus personnalisée, il suffit de redéfinir les méthodes **onSaveInstanceState** et **onCreate** et d'utiliser les méthodes qui permettent de lire/écrire des données sur l'objet **Bundle**.

Android fournit aussi automatiquement, la persistance du chemin de navigation de l'utilisateur, ce qui le renvoie à la bonne activité lorsqu'il appuie sur la touche Retour. La navigation vers le parent (bouton "back") n'est pas automatique car c'est le concepteur qui doit décider vers quelle activité l'application doit retourner quand on appuie sur "back". Elle peut être programmée dans le Manifest avec l'attribut android:parentActivityName.

5.2 Préférences

La classe **SharedPreferences** permet de gérer des paires de clé/valeurs associées à une activité. On récupère un tel objet par l'appel à **getPreferences**:

```
SharedPreferences prefs =
getPreferences(Context.MODE_PRIVATE);
String nom = prefs.getString("login", null);
Long nom = prefs.getLong("taille", null);
```

La méthode **getPreferences(int)** appelle en fait **getPreferences(String, int)** à partir du nom de la classe de l'activité courante. Le mode **MODE_PRIVATE** restreint l'accès au fichier créé à l'application. Les modes d'accès **MODE_WORLD_READABLE** et **MODE_WORLD_WRITABLE**permettent aux autres applications de lire/écrire ce fichier.

L'intérêt d'utiliser le système de préférences prévu par Android réside dans le fait que l'interface graphique associé à la modification des préférences est déjà programmé: pas besoin de créer l'interface de l'activité pour cela.

Représentation XML d'une page de préférences

Une activité spécifique a été programmée pour réaliser un écran d'édition de préférences. Il s'agit de **PreferenceActivity**. A partir d'une description XML des préférences, la classe permet d'afficher un écran composé de modificateurs pour chaque type de préférences déclarées.

Voici un exemple de déclarations de préférences XML, à stocker dans *res/xml/preferences.xml*:

```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="first_preferencescreen">
  <CheckBoxPreference
        android:key="wifi_enabled"
        android:title="WiFi" />
  <PreferenceScreen
        android:key="second_preferencescreen"
        android:title="WiFi settings">
    <CheckBoxPreference
          android:key="prefer_wifi"
          android:title="Prefer WiFi" />
    ... other preferences here ...
  </PreferenceScreen>
</PreferenceScreen>
```

Activité/Fragment de préférences

Pour afficher l'écran d'édition des préférences correspondant à sa description XML, il faut créer une nouvelle activité qui hérite de **PreferenceActivity** et simplement appeler la méthode **addPreferencesFromResource** en donnant l'id de la description XML:

```
public class MyPrefs extends PreferenceActivity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.preferences);
}}
```

Pour lancer cette activité, on crée un bouton/menu et un **Intent** correspondant.

Attributs des préférences

Les attributs suivants sont utiles:

- **android:title**: La string apparaissant comme nom de la préférence
- **android:summary**: Une phrase permettant d'expliciter la préférence
- **android:key**: La clef pour l'enregistrement de la préférence

Pour accéder aux valeurs des préférences, on utiliser la méthode**getDefaultSharedPreferences** sur la classe **PreferenceManager**. C'est la clef spécifiée par l'attribut **android:key** qui est utilisée pour récupérer la valeur choisie par l'utilisateur.

```
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(
    getApplicationContext());
String login = prefs.getString("login","");
```

Des attributs spécifiques à certains types de préférences peuvent être utilisés, par exemple **android:summaryOn** pour les cases à cocher qui donne la chaîne à afficher lorsque la préférence est cochée. On peut faire dépendre une préférence d'une autre, à l'aide de l'attribut **android:dependency**. Par exemple, on peut spécifier dans cet attribut le nom de la clef d'une préférence de type case à cocher:

```
<CheckBoxPreference android:key="wifi" ... />
<EditTextPreference android:dependency="wifi" ... />
```

Exemples: Checkbox et Text

Une case à cocher se fait à l'aide de **CheckBoxPreference**:

```
<CheckBoxPreference android:key="wifi"
  android:title="Utiliser le wifi"
  android:summary="Synchronise l'application via le wifi."
  android:summaryOn="L'application se synchronise via le wifi."
  android:summaryOff="L'application ne se synchronise pas."
/>
```

Un champs texte est saisi via **EditTextPreference**:

```
<EditTextPreference android:key="login&"
  android:title="Login utilisateur"
  android:summary="Renseigner son login d'authentification."
  android:dialogTitle="Veuillez saisir votre login"
/>
```

Exemples: Listes/Options

Une entrée de préférence peut être liée à une liste de paires de clef-valeur dans les ressources:

```
<resources>
<array name="key"> <!-- Petite=1, Moyenne=5, Grande=20 -->
        <item>"Petite"</item>
        <item>"Moyenne"</item>
        <item>"Grande"</item>
    </array>
    <array name="value">
        <item>"1"</item>
        <item>"5"</item>
        <item>"20"</item>
    </array>
</resources>
```

qui se déclare dans le menu de préférences:

```
<ListPreference android:title="Vitesse"
  android:key="vitesse"
  android:entries="@array/key"
  android:entryValues="@array/value"
  android:dialogTitle="Choisir la vitesse:"
  android:persistent="true">
</ListPreference>
```

Lorsque l'on choisit la valeur "Petite", la préférence *vitesse* est associée à "1".

5.3 Les fichiers

Android fournit aussi un accès classique au système de fichier pour tous les cas qui ne sont pas couverts par les préférences ou la persistance des activités, c'est à dire de nombreux cas. Le choix de Google est de s'appuyer sur les classes classiques de Java EE tout en simplifiant la gestion des permissions et des fichiers embarqués dans l'application.

Pour la gestion des permissions, on retrouve comme pour les préférences les constantes **MODE__PRIVATE** et **MODE_WORLD_READABLE/WRITABLE** à passer en paramètre de l'ouverture du fichier. En utilisant ces constantes comme un masque, on peut ajouter l **MODE_APPEND** pour ajouter des données.

```
try {
  FileOutputStream out =
openFileOutputStream("fichier", MODE_PRIVATE);
  ...
} catch (FileNotFoundException e) { ... }
```

Les ressources permettent aussi de récupérer un fichier embarqué dans l'application:

```
Resources res = getResources();
InputStream is =
res.openRawResource(R.raw.fichier);
```

A noter: les fichier sont à éviter. Mieux vaut alléger l'application au maximum et prévoir le téléchargement des ressources nécessaires à partir d’un serveur.

Les fichiers : support de stockage externe

Permissions :

```
<uses-permission
android:name="android.permission.WRITE_EXTERNA
L_STORAGE" android:maxSdkVersion="18" />
```

Vérification :

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

chemin pour stocker des fichiers qui doivent être supprimés à la désinstallation de l'application : getExternalFilesDir()

chemin pour stocker des fichiers qui doivent rester après la désinstallation de l'application : getExternalStoragePublicDirectory()

accéder à un fichier dans un répertoire standard publique : (eg. photos)

```
File file = new
File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES),
"album/image.jpg");
```

5.4 BDD SQLite

Android dispose d'un SGBD relationnel appelé SQLite. Même si la base doit être utilisée avec modération, cela fournit un moyen efficace de gérer une petite quantité de données.

Ici un exemple classique de création de base de données, ce qui se fait en héritant de **SQLiteOpenHelper**:

```
public class TchatOpenHelper extends SQLiteOpenHelper {

  private static final String SQL_CREATE =
          "CREATE TABLE tchatcontact (
  nom_du_champ_1 INTEGER PRIMARY KEY,
  nom_du_champ_2 TEXT NOT NULL,
  nom_du_champ_3 REAL NOT NULL CHECK (nom_du_champ_3
> 0),
  nom_du_champ_4 INTEGER DEFAULT 10);";

  TchatOpenHelper(Context context) {
     super(context, "tchatcontacts", null, 2);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
     db.execSQL(SQL_CREATE);
  }
@Override
  public void onUpgrade(SQLiteDatabase db, int old, int new) {
     //politique de MAJ de la BDD
  }
}
```

Types de données

Pour SQLite, c'est simple, il n'existe que cinq types de données :

- NULL pour les données NULL.
- INTEGER pour les entiers (sans virgule).
- REAL pour les nombres réels (avec virgule).
- TEXT pour les chaînes de caractères.
- BLOB pour les données brutes, par exemple si vous voulez mettre une image dans votre base de

Il est aussi possible de déclarer des contraintes pour chaque attribut. On trouve comme principales contraintes :

- PRIMARY KEY, FOREIGN KEY pour désigner une clé primaire ou étrangère ;
- AUTOINCREMENT pour un attribut numérique auto-incrémentable;
- NOT NULL pour indiquer que cet attribut ne peut valoir NULL ;
- UNIQUE pour indiquer une valeur unique;
- CHECK afin de vérifier que la valeur de cet attribut est cohérente ;
- DEFAULT sert à préciser une valeur par défaut.

Lecture / Écriture dans une BDD: Sélection

Pour réaliser des écritures ou lectures, on utilise les méthodes **getWritableDatabase()** et **getReadableDatabase()** qui renvoient une instance de **SQLiteDatabase**. Sur cet objet, une requête peut être exécutée au travers de la méthode **query()**:

```
public Cursor query (boolean distinct,
String table, String[] columns,
String selection, String[] selectionArgs,
String groupBy, String having, String
orderBy, String limit)
```

Req: d'autres solutions existent comme:

```
public Cursor rawQuery(String sql,
String[] selectionArgs)
```

L'objet de type **Cursor** permet de traiter la réponse (en lecture ou écriture), par exemple:

- **getCount()**: nombre de lignes de la réponse
- **moveToFirst()**: déplace le curseur de réponse à la première ligne
- **getInt(int columnIndex)**: retourne la valeur (int) de la colonne passée en paramètre
- **getString(int columnIndex)**: retourne la valeur (String) de la colonne passée en paramètre
- **moveToNext()**: avance à la ligne suivante
- **getColumnName(int)**: donne le nom de la colonne désignée par l'index
- ...

Lecture / Écriture dans une BDD: Insertion

pour insérer une entrée, on utilisera la méthode :

`long insert(String table, String nullColumnHack, ContentValues values)`, qui renvoie le numéro de la ligne ajoutée où :

- table est l'identifiant de la table dans laquelle insérer l'entrée.
- nullColumnHack est le nom d'une colonne à utiliser au cas où vous souhaiteriez insérer une entrée vide (souvent mise à null).
- values est un objet qui représente l'entrée à insérer.

Les ContentValues sont utilisés pour insérer des données dans la base. Ainsi, on peut dire qu'ils fonctionnent un peu comme les Bundle par exemple, puisqu'on peut y insérer des couples identifiant-valeur, qui représenteront les attributs des objets à insérer dans la base.

Après avoir récupéré une instance de base de donnée en mode écriture, on peut faire:

```
ContentValues row = new
ContentValues();

row.put(“id”, 657882);

row.put(“nom”, “Hamdane”);

mDb.insert(TABLE_NAME, null, row);
```

Lecture / Écriture dans une BDD: Suppression

La méthode utilisée pour supprimer est quelque peu différente. Il s'agit de `int delete(String table, String whereClause, String[] whereArgs)`. L'entier renvoyé est le nombre de lignes supprimées. Dans cette méthode :

- `table` est l'identifiant de la table.
- `whereClause` correspond au WHERE en SQL. Par exemple, pour sélectionner la première valeur dans la table Metier, on mettra pour whereClause la chaîne « id = 1 ». En pratique, on préférera utiliser la chaîne « id = ? » et je vais vous expliquer pourquoi tout de suite.
- `whereArgs` est un tableau des valeurs qui remplaceront les « ? » dans whereClause. Ainsi, si whereClause vaut « LIKE ? AND salaire > ? » et qu'on cherche les métiers qui ressemblent à « ingénieur avec un salaire supérieur à 1000 € », il suffit d'insérer dans whereArgs un String[] du genre {"ingenieur", "1000"}.

Exemple :

```
public void supprimer(long id) {


getWritableDatabase().delete(TAB
LE_NAME, " id = ?", new String[]
{String.valueOf(id)});

}
```

Lecture / Écriture dans une BDD: Modification

La syntaxe est très similaire à la précédente :

int update(String table, ContentValues values, String whereClause, String[] whereArgs)

On ajoute juste le paramètre values pour représenter les changements à effectuer dans le ou les enregistrements cibles.

Exemple :

```
ContentValues value = new
ContentValues();

value.put(“nom”, “ABBASSI”);

mDb.update(TABLE_NAME, value, "id =
?", new String[]
{String.valueOf(657882)});
```

Exercice

- Dans le projet précédemment réalisé (téléchargement et affichage d'un profil d'étudiant), utilisez le bouton "enregistrer" pour enregistrer localement des remarques sur l'étudiant.
- Identifiez les remarques dans la base de données locale par l'identifiant de l'étudiant.
- Les remarques précédemment enregistrées (s'il y en a) doivent être affichées au moment de la récupération du profil depuis le serveur.