

Stockage de données sous Android

Master 2
informatique
2012-2013



micro



Données temporaires d'une activité

- Une activité peut être détruite :
 - En cas de changement de géométrie de l'écran
 - En cas de pénurie de mémoire centrale
- Avant destruction, sauvegarde manuelle des structures de données temporaires utiles --> restauration de l'état de l'activité lors de la réinstantiation
 - *onSaveInstanceState(Bundle outState)* : sauvegarde des données dans le Bundle
 - *onCreate(Bundle state)* : restauration des données
- Procédé de sauvegarde temporaire sans persistance au redémarrage du système

Stockage de données persistantes

- Stockage de fichiers
 - Sur le système de fichier principal de la mémoire flash interne
 - Sur une carte SD ou un périphérique de stockage USB connecté
 - En ligne sur une machine distante
- Stockage de données structurées
 - Dans une base de données SQLite3
- Stockage de préférences (couples de clé-valeur)
 - En utilisant l'interface *SharedPreferences*

Manipulation de fichiers internes

- Chaque application dispose d'un répertoire réservé pour stocker ses fichiers (noms de fichier en UTF-8) récupérable avec *File Context.GetFilesDir()* (ce répertoire est détruit lors de la désinstallation de l'application)
- Le système de fichiers interne peut être chiffré à l'aide du mot de passe de déverrouillage
- Opérations (chemins relatifs au répertoire de l'application) sur instance de *Context* :
 - *FileInputStream openFileInput(String name)*
 - *FileOutputStream openFileOutput(String name, int mode)*
 - *File getDir(String name, int mode)* : ouverture (création si nécessaire) d'un répertoire
 - *File deleteFile(String name)*
 - *String[] fileList()* : liste des fichiers privés sauvés par l'application
- Modes de création de fichier et répertoire (combinables par ou binaire) :
 - *MODE_PRIVATE* : accessibilité réservée à l'application (ou d'autres applications avec le même user ID)
 - *MODE_APPEND* : ajout en fin de fichier (par défaut écrasement du fichier)
 - *MODE_WORLD_{READABLE, WRITABLE}* : accessibilité en lecture, écriture pour toutes les applications. À bannir : si des données doivent être lisibles ou écrites depuis d'autres applications, elles doivent l'être depuis une API publique

Manipulation de fichiers externes

- Les fichiers sur support externes sont toujours publics et non chiffrés
- Obtention de répertoires racines externes :
 - *File Context.getExternalFilesDir(String type)* : répertoire racine réservé à l'application (détruit à la désinstallation), par exemple `/sdcard/Android/data/fr.upemlv.HelloWorld/files/` ; type peut être null
 - *File Environment.getExternalStorageDirectory()* : répertoire racine externe global
 - *File Environment.getExternalStoragePublicDirectory(String type)* : répertoire racine externe global pour un type de fichier spécifié
- Les fichiers de mêmes types doivent être regroupés dans des sous-répertoires :
 - *DIRECTORY_MUSIC, DIRECTORY_PODCASTS, DIRECTORY_RINGTONES, DIRECTORY_ALARMS, DIRECTORY_NOTIFICATIONS, DIRECTORY_PICTURES, DIRECTORY_MOVIES, DIRECTORY_DOWNLOADS, DIRECTORY_DCIM*

Répertoires cache

- Obtention des chemins vers les répertoires cache spécifiques à l'application courante :
 - *File getCacheDir()*
 - *File getExternalCacheDir()* (retourne null si le stockage externe n'est pas disponible)
- Utile pour y stocker des données temporaires (issues de calculs, de récupération de données sur Internet...)
- Les données stockées peuvent être effacées par le système :
 - En cas de désinstallation de l'application
 - En cas de pénurie de mémoire de stockage
- Nécessité pour chaque application d'être raisonnable pour l'espace utilisé par le cache (partage par toutes les applications)

Sauvegarde des fichiers d'application

- Propriétés de l'application
 - *android:allowBackup*
 - *android:backupAgent* : classe de backup
- Implantation d'une classe dérivée de BackupAgent :
 - *void onBackup(ParcelFileDescriptor oldState, BackupDataOutput data, ParcelFileDescriptor newState)* : réalise une sauvegarde incrémentale depuis oldState vers newState en écrivant des données binaires dans data
 - *void onRestore(BackupDataInput data, int appVersionCode, ParcelFileDescriptor newState)* : restaure une sauvegarde incrémentale ; appVersionCode est la version de l'application ayant réalisée le backup
 - Il existe des *BackupAgentHelper* pour aider à la sauvegarde/restauration de données courantes (fichiers, préférences...)
 - Attention à ne pas modifier des données concurremment à leur sauvegarde (utiliser un verrou)
- Lorsque des données utilisateur sont modifiées, l'application peut demander une sauvegarde incrémentale avec *BackupManager.dataChanged()*
- L'implantation du transport de backup dépend de la distribution Android (par exemple un système de backup utilisant le nuage de Google)

Préférences d'application (couples clé/valeur)

- *SharedPreferences Context.getSharedPreferences(String name, int mode)*
 - *getPreferences(int mode)* récupère les préférences du nom de l'activité courante
 - Plusieurs applications peuvent accéder aux mêmes préférences si mode = *MODE_WORLD_READABLE* ou *MODE_WORLD_WRITEABLE*
- Récupération d'une valeur avec *get{Boolean, Float, Int, Long, String}(String key, X defaultValue)*
- Modification d'entrées :
 - en obtenant l'éditeur (*edit()*) sur lequel on réalise des opérations *putX(String key, X value)*
 - en validant atomiquement les changements avec *commit()*
- Possibilité d'ajouter un listener appelé lors de la modification d'une entrée : *registerOnSharedPreferenceChangeListener()*

Base de données SQLite3

- SQLite3 : moteur de base de données sur fichiers sans support de concurrence ; supporte les requêtes SQL ; pas de typage fort
- Permet la persistance de données structurées en tables (définies par des colonnes)
- Relations possibles entre différentes tables (jointure de tables)
- Maintenance d'index de tri sur les enregistrements : requêtes de sélection rapides

Gestion de tables SQLite

- Création de table avec **CREATE TABLE** :
CREATE TABLE IF NOT EXISTS gpstrace(date INTEGER PRIMARY KEY, latitude REAL NOT NULL, longitude REAL NOT NULL, altitude REAL) ;
- Création d'index avec **CREATE INDEX** :
CREATE INDEX IF NOT EXISTS latitude ON gpstrace (latitude)
- Renommage de table avec **ALTER TABLE** :
ALTER TABLE gpstrace RENAME TO newgpstrace
- Effacement de table avec **DROP TABLE** :
DROP TABLE gpstrace

Requêtes SQL sur enregistrements

- Requêtes :
 - Sélection : *SELECT col1,col2,...,coln FROM table WHERE expr GROUP BY expr HAVING expr {UNION, INTERSECT, EXCEPT} SELECT ...*
 - Insertion : *INSERT INTO table (col1,col2,...,coln) VALUES (val1,val2,...,valn)*
 - Mise à jour : *UPDATE table SET col1=val1, col2=val2, ...,coln=valn WHERE expr*
 - Suppression : *DELETE FROM table WHERE expr*
- Expressions :
 - Opérateurs classiques : `|| * / % + - << >> & | < <= > >= == != NOT`
 - colname LIKE x : suit le format de la chaîne x (%: joker 0, 1 ou plusieurs caractères, _ : joker 1 caractère) ; exploite les index
 - colname REGEXP x : valide l'expression régulière x
 - Penser aux index lors de l'écriture d'expressions (pour ne pas parcourir tous les enregistrements)

SQLiteOpenHelper

- Classe à dériver pour l'ouverture de base SQLite
- Deux méthodes à redéfinir :
 - *onCreate(SQLiteDatabase)* : on y exécute le script de création de base (création des tables et index)
 - *onUpgrade(SQLiteDatabase, int, int)* : pour mettre à jour le schéma d'une base d'une version *i* à une version *j* > *i*
- *onOpen(SQLiteDatabase)* peut également être redéfini pour agir lors de l'ouverture de la base
- Il faut expliciter un constructeur, par exemple :

```
public static final String DB_NAME = "gpsLog";  
public static final int VERSION = 1;  
public GPSLogBaseHelper(Context context) { super(context, DB_NAME, null, VERSION); }
```

SQLiteDatabase

- SQLiteDatabase = Instance représentant une base SQLite
 - base ouvrable depuis un SQLiteOpenHelper
Exemple : *SQLiteDatabase base = new GPSLogBaseHelper(this).get{Readable, Writable}Database()*
 - base devant être fermée après utilisation : *base.close()*
- Requête avec méthode query() :
 - *Cursor query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)*
 - Exemple : récupération des 100 traces GPS les plus récentes de latitude supérieure à 45°
 - *Cursor c = db.query("gpstraces", new String[]{"date", "latitude", "longitude"}, "latitude > ?", new String[]{"45"}, null, null, "date DESC", 100)*
 - *Cursor c = db.rawQuery("SELECT date, latitude, longitude FROM gpstraces WHERE latitude > ? ORDER BY date DESC limit 100", new String[]{"45"})*
 - Ne jamais intégrer directement les arguments dans la requête ([faille d'injection de commandes SQL](#))

Cursor

- On parcourt les résultats d'une requête avec une instance de *Cursor* (thread unsafe)
- Méthode utiles :
 - *int getCount()* : nombre d'éléments dans le Cursor
 - *X get{Short, Int, Long, Float, Double, String, Blob}(int i)* : valeur de la colonne #i de l'enregistrement courant
 - *boolean moveToNext()* : déplacement sur le prochain enregistrement (retourne false si fin des enregistrements)

Ecriture d'enregistrements

- Insertion avec *SQLiteDatabase.insert(String table, String nullColumnHack, ContentValues values)*
 - ContentValues est une sorte de Map où les valeurs des colonnes sont insérées avec *put(String key, X value)*
- Mise à jour : *int SQLiteDatabase.update(String table, ContentValues values, String whereClause, String[] whereArgs)*
- Suppression : *SQLiteDatabase.delete(String table, String whereClause, String[] whereArgs)*

ContentProvider

- Fournit une interface d'accès aux fichiers ou données structurées d'une application
- Les données sont identifiées par une URI, e.g.
content://fr.upemlv.gpslogger.provider/log/ID
- Opérations CRUD disponibles (à implanter de façon thread-safe) :
 - Récupération de données avec *query(...)*
 - Insertion d'enregistrement avec *insert(...)*
 - Mise à jour d'enregistrements avec *update(...)*
 - Suppression d'enregistrements avec *delete(...)*
- Autres méthodes à redéfinir :
 - *onCreate()* appelé lors de la création du *ContentProvider*
 - *String getType(Uri uri)* retourne le type MIME d'un contenu identifié par son Uri
- Il est conseillé de créer une classe compagnon contenant des sous-classes pour chaque table avec le nom des colonnes en constantes ainsi que l'Uri vers la table.

Types MIME d'URI

- `ContentProvider.getType(Uri uri)` : retourne le type MIME d'une table ou d'un enregistrement
 - Table : *vnd.android.cursor.dir/vnd.fr.upemlv.gpslog.provider.gpstrace*
 - Enregistrement d'une table :
vnd.android.cursor.item/vnd.fr.upemlv.gpslog.provider.gpstrace
- `ContentProvider.getStreamTypes(Uri uri, String mimeTypeFilter)` : retourne les types MIME supportés pour un fichier dont l'URI est spécifié. Un filtre indique quels sont les types qui nous intéressent.
 - Par exemple, on peut retourner {"image/jpeg", "image/png"} pour une image

ContentProvider.query()

- *query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)*
 - Méthode implantée en appelant *SQLiteDatabase.query()*
 - Il faut auparavant reconnaître quelle table ou enregistrement est concerné : on analyse pour cela l'URI
 - ID indiqué en fin d'URI : on ne récupère que l'enregistrement demandé dans la table
 - Sinon on requête la table avec les paramètres de sélection et de tri
 - L'analyse de l'URI peut être aidée avec *UriMatcher* (on y enregistre les schémas des URIs)

ContentProvider.{insert(), update(), delete()}

- *Uri insert(Uri uri, ContentValues values)* : ajoute un enregistrement (values contient les couples nom de colonne, valeur) ; retourne une URI avec l'identificateur de l'enregistrement
- *int update(Uri uri, ContentValues values, String selection, String[] selectionArgs)* : met à jour les enregistrements suivant selection avec selectionArgs ; retourne le nombre d'enregistrements mis à jour
- *int delete(Uri uri, String selection, String[] selectionArgs)* : efface des enregistrements ; retourne le nombre d'enregistrements effacés

Fournitures de méthodes et fichiers par un `ContentProvider`

- Mécanisme de RPC intégré :
 - Redéfinition de *Bundle call(String method, String arg, Bundle extras)*
- Accès à des fichiers :
 - Redéfinition de *ParcelFileDescriptor openFile(Uri uri, String mode)*
 - Mode = "r", "rw", "rwt" (troncature de fichier existant)
 - Création d'un *ParcelFileDescriptor* à partir d'un fichier : *ParcelFileDescriptor.open(File file, int mode)*

Les permissions

- Modèle de sécurité Android : chaque application est exécutée dans son propre processus avec son propre user ID
 - Par défaut le ContentProvider peut être accédé uniquement par son application hôte
- Création de permissions dans le manifeste

```
<permission android:name="fr.upemlv.gpslog.GPS_TRACE_READ"
    android:label="@string/perm_label_GPSTraceRead"
    android:description="@string/perm_desc_GPSTraceRead"
    android:permissionGroup="android.permission-group.PERSONAL_INFO"
    android:protectionLevel="dangerous" />

<permission android:name="fr.upemlv.gpslog.GPS_TRACE_WRITE"
    android:label="@string/perm_label_GPSTraceWrite"
    android:description="@string/perm_desc_GPSTraceWrite"
    android:permissionGroup="android.permission-group.PERSONAL_INFO"
    android:protectionLevel="dangerous" />
```

Pour être accédé d'une autre application, le ContentProvider doit également posséder la propriété `android:exported="true"`

3 niveaux de protection :

- normal : pas d'alerte spécifique de l'utilisateur
- dangerous : l'utilisateur peut être informé et refuser
- signature : seule une application signée avec le même certificat peut obtenir la permission

Protection d'un composant par permission

- *<{application, activity, service, receiver, provider} android:permission="p" ...>*
 - Une application externe doit disposer de p pour interagir avec ce composant
- provider propose également deux propriétés plus précises :
 - *android:readPermission*
 - *android:writePermission*
- Permissions accordables avec une granularité fine sur un provider (en fonction de l'URI) :

```
<provider ...>  
  <path-permission android:path="string"  
    android:pathPrefix="string"  
    android:pathPattern="string"  
    android:permission="string"  
    android:readPermission="string"  
    android:writePermission="string" />  
</provider>
```

Demande de permission

- Un composant nécessitant une permission la demande dans le manifeste de l'application
 - `<uses-permission
 android:name="nom.de.la.permission" />`
- Les permissions demandées doivent être visualisées avant l'installation : elles sont regroupées en *PermissionGroup*

Permissions temporaires

- Autorisation de permissions temporaires avec `<provider android:grantUriPermissions="true" ...>`
 - Granularité sur URI avec `<grant-uri-permission android:path="string" android:pathPattern="string" android:pathPrefix="string" />`
- Permet à une application de déléguer temporairement une permission qu'elle possède (par exemple gpslog peut déléguer une permission de lecture à une application de cartographie pour visualiser les traces)

ContentResolver

- L'accès à un *ContentProvider* est réalisé depuis un *ContentResolver*. Le *ContentResolver* du *Context* est récupérable avec *getContentResolver()*
- Propose les méthodes CRUD *query()*, *insert()*, *update()* et *delete()*
- Propose des méthodes de requêtage et d'ajout en lots : *applyBatch()* et *bulkInsert()*

Élément dans le presse-papier

- *ClipData* représente des données dans le presse-papier
 - *ClipDescription ClipData.getDescription()* : métadonnées avec les types MIME représentés
 - Un *ClipData* contient généralement un *ClipData.Item* (*getItemAt(0)*) qui peut être :
 - Du texte pur (ou du texte HTML) : *ClipData.new{Plain, Html}Text(CharSequence label, CharSequence text)* ; *ClipData.Item.get{Plain, Html}Text()*
 - Une URI : *ClipData.newUri(ContentResolver resolver, CharSequence label, Uri uri)* ; existe aussi en version *newRawUri* sans spécifier le résolver ; *ClipData.Item.getUri()*
 - Un Intent : *ClipData.newIntent(CharSequence label, Intent intent)*

Fonctionnement du copier-coller

- Récupération du ClipboardManager :
ClipboardManager cbm = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE)
- On crée un nouveau *ClipData* (avec texte, URI ou Intent)
- On le copie dans le presse-papier :
cbm.setPrimaryClip(myClipData) ;
- ...
- On récupère l'item principal pour le coller ailleurs (on consulte ses getters) :
ClipData.Item item = myClipData.getItemAt(0)
 - Si une URI est présente, on utilise le *ContentResolver* pour récupérer les données et les mettre sous une forme adaptée
 - On peut essayer de forcer la conversion en texte de tout type de données avec *ClipData.Item.coerceToText(Context)*