android

Laurent Provot <laurent.provot@uca.fr>

Octobre 2022

Stockage de données sous Android

Stockage temporaire

- Une activité peut être détruite :
  - En cas de changement de configuration
  - En cas de pénurie de mémoire
- Avant destruction : sauvegarde manuelle des structures de données temporaires utiles
  - `onSaveInstanceState(outState: Bundle?)`
- À la reprise de l'activité : restauration des données
  - `onCreate(state: Bundle?)` (ou `onRestoreInstanceState(state: Bundle?)`, …)
- Procédé de sauvegarde temporaire sans persistance au redémarrage de l'application

Persistance

- Besoin de sauvegarder des informations **entre 2 exécutions** d'une application
- Plusieurs *espaces* de stockage avec Android
  - Les préférences
  - Le stockage interne
  - Le stockage externe
  - Une base de données embarquée
  - Un espace de stockage distant
- Certaines zone sont privées, d'autres publiques

Les préférences (`SharedPreferences`)

- Stockage (normalement privé) de type paires clé/valeurs
  - Valeurs de types simples uniquement : `Boolean`, `Float`, `Int`, `Long`, `String`, `Set<String>`
- Obtenir (ou créer) un profil de préférences
  - `getSharedPreferences(name: String, mode: Int)` ou (pour une activité spécifique) `getPreferences(mode: Int)`
  - mode : `0` ou `MODE_PRIVATE`, ~~`MODE_WORLD_READABLE`~~, ~~`MODE_WORLD_WRITEABLE`~~, ~~`MODE_MULTI_PROCESS`~~
  - Obtention des infos stockées : `getXXX(key: String, defaultValue: XXX): XXX`

Les préférences (`SharedPreferences`) : modifications

- Modifier les préférences
  - `SharedPreferences.Editor` obtenu par un appel à `edit()`
  - Modifications des infos : `putXXX(key: String, value: XXX)`
  - `apply()` pour valider les modifications (asynchone)
  - `commit()` pour valider les modifications (synchone)
- Possibilité d'enregistrer un listener :
  - `registerOnSharedPreferenceChangeListener( lst: SharedPreferences.OnSharedPreferenceChangeListener)`
  - `onSharedPreferenceChanged( sharedPreferences: SharedPreferences, key: String)`

Les préférences (`SharedPreferences`) : exemple

```
private const val PREFS_NAME = "my_prefs_file"
private const val KEY_SILENT = "keySilent"

class ReminderActivity : Activity() {
    override fun onCreate(state: Bundle?) {
        super.onCreate(state)
        val settings = getSharedPreferences(PREFS_NAME, 0)
        val silent = settings.getBoolean(KEY_SILENT, false)
        setSilent(silent)
    }

    override fun onStop() {
        super.onStop()
        val settings = getSharedPreferences(PREFS_NAME, 0)
        with(settings.edit()) {
            putBoolean(KEY_SILENT, mSilentMode)
            apply()
        }
    }
}
```

Stockage interne

- Lecture/écriture de données dans des fichiers classiques
- Espace de stockage interne privé propre à l'application
- Suppression si désinstallation de l'application
- Généralement `/data/data/[package name]`
- Utilisation des classes Kotlin d'E/S habituelles
- Points d'entrée :
  - `fun openFileOutput(name: String, mode: Int): FileOutputStream`
  - `fun openFileInput(name: String, mode: Int): FileInputStream`
- Paramètres comme pour les `SharedPreferences`
- Ne pas oublier de libérer la ressource : `close()`

Stockage interne : exemple

```
private const val DEFAULT_BACKUP_FILENAME = "crimes.json"

class CrimesJSONSerializer(private val context: Context,
                           private val filename: String = DEFAULT_BACKUP_FILENAME) {

    fun saveCrimes(crimes: List<Crime>) {
        val array = JSONArray()
        for (c in crimes)
            array.put(c.toJSON())

        val out = context.openFileOutput(filename,
                                         Context.MODE_PRIVATE)
        OutputStreamWriter(out).use {
            it.write(array.toString())
        }
    }
```

Stockage interne : exemple

```
    fun loadCrimes(): List<Crime> {
        val crimes = ArrayList<Crime>()
        val input = context.openFileInput(filename)
        BufferedReader(InputStreamReader(input)).use {
            val jsonString = it.readText()
            val array = JSONTokener(jsonString).nextValue() as JSONArray
            for (i in 0 until array.length()) {
                crimes.add(Crime(array.getJSONObject(i)))
            }
        }
        return crimes;
    }
}
```

Stockage interne

- Autres méthodes utiles :
  - `getFilesDir(): File`,
    `getDir(name: String, mode: Int): File`,
    `deleteFile(name: String)`,
    `fileList(): Array<String>`
- Accès à une ressource static du projet (classe `Resources`)
  - Fichier `filename` dans `res/raw`
  - Ouverture en lecture seulement (`R.raw.filename` en paramètre)
    `openRawResource(id: Int): InputStream`
- Création de fichiers temporaires
  - `getCacheDir(): File`
  - Suppression auto en cas d'espace manquant
  - Se fixer une taille max totale (1Mo)
  - Suppression si désinstallation de l'application

Stockage externe

- Différents supports possibles
  - Carte SD externe
  - Mémoire interne
- Accès et lecture possibles par les utilisateurs
- Permission nécessaires pour lecture/écriture si Android < 4.4
  - `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`

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

Stockage externe

- Vérifier si support disponible
  - Carte SD potentiellement retirée
  - Support monté en USB sur ordinateur
- Classe `Environment` :
  `getExternalStorageState(): String`
- `Environment.MEDIA_UNKNOWN`,
  `Environment.MEDIA_MOUNTED`, ...

```
// Vérifie que le stockage externe est dispo et au moins accessible en lecture
fun isExternalStorageReadable() =
  with(Environment.getExternalStorageState()) {
    Environment.MEDIA_MOUNTED == this ||
    Environment.MEDIA_MOUNTED_READ_ONLY == this
  }
```

Stockage externe publique

- Accès à l'espace publique de stockage
- Généralement `/sdcard/`
- Différents répertoires standard :
  `Music/`, `Pictures/`, `Download/`, `Ringtones/`, …
- Accès :
  - `getExternalStoragePublicDirectory (type: String): File`
  - deprecated depuis API 29 (Android 10) : utiliser Storage Access Framework (SAF) ou `getExternalFilesDir(type: String): File`
  - type : `DIRECTORY_MUSIC`, `DIRECTORY_PICTURES`, `DIRECTORY_DOWNLOADS`, `DIRECTORY_RINGTONES`, …
- `File.mkdirs()` pour être sûr que le répertoire existe
- `MediaScannerConnection.scanFile(…)` pour màj du `MediaStore`

Stockage externe publique : exemple (deprecated)

```
fun createExternalStoragePublicPicture() {
    val pictDir = Environment.getExternalStoragePublicDirectory(
                                    Environment.DIRECTORY_PICTURES)
    val file = File(pictDir, "Balloons.jpg")
    pictDir?.mkdirs()

    context.resources.openRawResource(R.drawable.balloons).use { balloons ->
        FileOutputStream(file).use {
            val data = ByteArray(balloons.available())
            balloons.read(data)
            it.write(data)
        }
    }
    MediaScannerConnection.scanFile(context,
     arrayOf(file.toString()), null) { path, uri ->
        Log.i("ExternalStorage", "Scanned $path:")
        Log.i("ExternalStorage", "-> uri=$uri") // null si FAIL
    }
}
```

Stockage externe privé

- Accès à un espace privé de l'application
- Généralement `Android/data/[package name]/`
- Permission nécessaires pour lecture/écriture uniquement Android < 4.4

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

Stockage externe privé

- Fonctionnement similaire à espace publique
- Mais fichiers supprimés si l'application est désinstallée
- Méthodes différentes :
  - `getExternalFilesDir (String type): File` (espace primaire)
  - `getExternalFilesDirs (String type): Array<File>` (tous les espaces)
- Non scanné(s) par le `MediaStore` par défaut
- Fichiers temporaires :
  - `getExternalCacheDir(): File`
  - `getExternalCacheDirs(): Array<File>`
- Attention à bien gérer le cache et faire le ménage

Stockage dans base de données

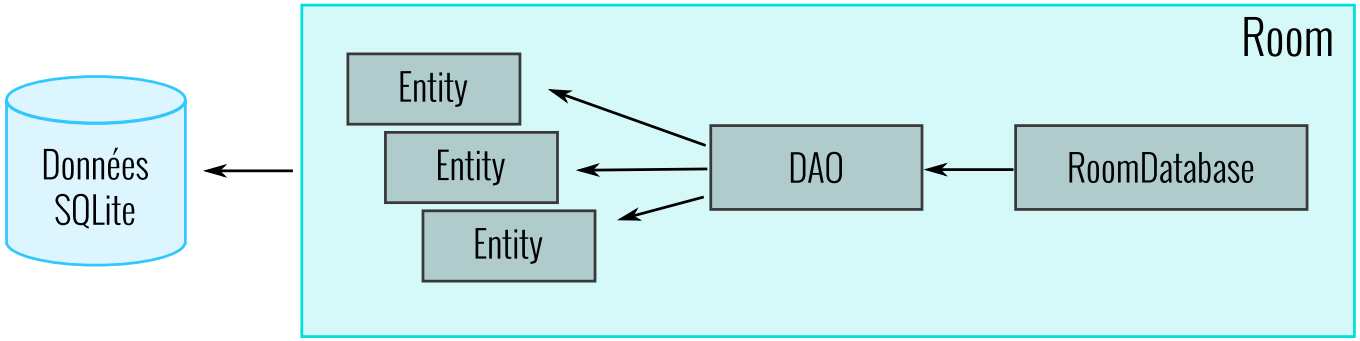
- Base de données embarquée SQLite
- Open source (http://www.sqlite.org)
- Empreinte mémoire faible ($\simeq$ 300kB)
- Support limité des types (TEXT, INTEGER, REAL)
- Examen facile avec `sqlite3`
- API pour la création de bases et le traitement de requêtes SQL
  - Par extension de la classe `SQLiteOpenHelper`
- Stocké dans l'espace privé interne (`/data/data/[pkg_name]/databases/`)
- Suppression si désinstallation de l'application

Persistance locale en SQLite

- Auparavant : lourd et répétitif
  1. Définition des `Contract`
  2. Définition du `DbHelper`
  3. Mise en place d'une classe implémentant les requêtes CRUD (ou `ContentProvider`)
  4. Uilisation des `Cursor` et des `ContentValue` (plutôt bas niveau)

- Avec Jetpack : `Room`
  - ORM (Object-Relational Mapping), tout simplement
  - Utilisation plus naturelle dans un cadre objet
  - Moins de boilerplate code : annotations => implémentation automatique des méthodes d'accès aux données
  - Vérification statique de la cohérence des requêtes SQL

Les éléments de `Room`



1. `Entity` : une table de la base de données
   - Représente une classe de votre modèle que vous voulez faire persister (avec des limitations tout de même)
   - Chaque instance est stockée dans une ligne de la table
   - Chaque colonne représente un attribut de la classe
2. `DAO` : une interface contenant les méthodes d'accès aux données
   - Mapping entre une requête SQL et une méthode
3. `RoomDatabase` : le point d'accès pour la persistance

Entity

- Une classe Kotlin de donnée annotée... et c'est tout !

```
@Entity(tableName = "books")
data class Book(@PrimaryKey(autoGenerate = true) val id: Long,
                var title: String,
                var author: String,
                @ColumnInfo(name = "num_pages") var nbPages: Int
                @Ignore var cover: Bitmap?)
```

- `@Entity` : marque la classe comme étant une entité
  - la table porte le nom de la classe par défaut
  - `tableName =` dans l'annotation pour définir un autre nom
- `@PrimaryKey` : chaque entité DOIT avoir au moins 1 champ qui définit sa clé primaire
  - `autoGenerate = true` laisse SQLite générer l'id unique
- `@Ignore` : champ qui ne sera pas persisté
- `@ColumnInfo` : permet de personnaliser l'association entre la colonne et l'attribut
  - par défaut la colonne porte le même nom que l'attribut

DAO

- Une interface qui contient la déclaration des méthodes d'accès aux données

```
@Dao
interface BookDao {
  @Query("SELECT * FROM books")
  fun loadAllBooks(): List<Book>

  @Query("SELECT * FROM books WHERE num_pages > :nbPages")
  fun loadThickBooks(nbPages: Int): Array<Book>

  @Insert(onConflict = OnConflictStrategy.REPLACE)
  fun insertBook(book: Book)

  @Update
  fun updateBook(book: Book)

  @Delete
  fun deleteBooks(vararg books: Book)
}
```

DAO

Bon à savoir :

- Les paramètres doivent tous être des entités (ou des collections d'entités)
- Les requêtes ne sont pas faites en asynchrone !!!
  - Exception : les requêtes qui retournent un objet de type `LiveData` ou `Flow`
- `@Delete` et `@Update` (et `@Insert` si conflit) utilisent la clé primaire pour retrouver les entités
- Pour effacer tous les éléments d'une table :

```
@Query("DELETE FROM books")
fun deleteAllBooks()
```

- Syntaxe pour les paramètres de type collection :

```
@Query("SELECT * FROM books WHERE author IN (:authors)")
fun loadBooksFromAuthors(authors: List<String>): List<Book>
```

RoomDatabase

- Une classe abstraite qui hérite de `RoomDatabase`

```
@Database(entities = [Book::class], version = 1)
abstract class BookDatabase : RoomDatabase() {
    abstract fun bookDAO() : BookDao
}
```

- `@Database` : permet de marquer la classe comme étant une `RoomDatabase`
  - on doit préciser toutes les entités utilisées
  - on spécifie la version (pour les migrations)
  - la classe doit contenir une méthode (getter) abstraite par DAO
- Une `RoomDatabase` est un objet plutôt coûteux, il est conseillé d'en faire un singleton

RoomDatabase

- Pour instancier la connexion à la base de données :

```
val bookDB = Room.databaseBuilder(
                    applicationContext,
                    BookDatabase::class.java,
                    "book-db"
                ).build()
```

- ou bien pour une BdD virtuelle (en mémoire)

```
val bookDB = Room.inMemoryDatabaseBuilder(
                    applicationContext,
                    BookDatabase::class.java
                ).build()
```

Relations entre les entités

- Room n'autorise pas les références entre les entités (cf. les raisons ici)

```
@Entity(tableName = "books")
data class Book(@PrimaryKey(autoGenerate = true) val id: Long,
                var title: String,
                var author: Author,    // <-- Pas possible
                @ColumnInfo(name = "num_pages") var nbPages: Int)
```

- Adopter une vision BdD des choses suivant le cas de la relation
  - 1-1 (one-to-one)
  - 1-N (one-to-many)
  - M-N (many-to-many)

Conversion de type (plutôt pour 1-1)

- Possibilité d'utiliser des `TypeConverters` pour transformer en un type que la BdD sait gérer

```
data class Author(var firstName: String, var lastName: String)

class AuthorStringConverter {
    @TypeConverter
    fun authorToString(author: Author) =
        author.firstName + "@" + author.lastName

    @TypeConverter
    fun authorFromString(str: String): Author {
      val indexToSplit = str.indexOf('@')
      return Author(str.substring(0, indexToSplit),
                    str.substring(indexToSplit + 1))
  }
}

@Database(entities = [Book::class], version = 1)
@TypeConverters(AuthorStringConverter::class)
abstract class BookDatabase : RoomDatabase() { … }
```

Intégration du type (plutôt pour 1-1)

- Permet d'intégrer tous les champs d'une classe au sein même de l'entité
- Par exemple tous les champs de `Author` seront directement mis dans la table `books`

```
@Entity(tableName = "books")
data class Book(@PrimaryKey(autoGenerate = true) val id: Long,
                var title: String,
                @Embedded var author: Author,
                @ColumnInfo(name = "num_pages") var nbPages: Int)
```

- Le nom des champs dans la table `books` sera le nom des attributs (ou du nom spécifié avec `@ColumnInfo` si entité)
- Annotation `@Embedded(prefix = "author_")` pour préfixer

Clé(s) étrangère(s) (relation 1-N)

- Pour les relations à cardinalité multiple on pourra utiliser l'annotation `@ForeignKey` pour définir une ou des contraintes entre les entités

```
@Entity(tableName = "books")
data class Book(@PrimaryKey(autoGenerate = true) val id: Long,
                var title: String,
                @ColumnInfo(name = "num_pages") var nbPages: Int) {
}

@Entity(tableName = "authors",
        foreignKeys = [ForeignKey(entity = Book::class,
                                  parentColumns = ["id"],
                                  childColumns = ["bookId"],
                                  onDelete = CASCADE)]
)
data class Author(@PrimaryKey(autoGenerate = true) val id: Long,
                  @ColumnInfo(name = "first_name") var firstName: String,
                  @ColumnInfo(name = "last_name") var lastName: String
                  val bookId: Long)
```

Clé(s) étrangère(s)

- L'entité `Book` ne possède plus l'information des auteurs
- On perd le côté naturel « Objet » : `book.authors[0]` impossible
- On doit rajouter des méthodes au DAO pour récupérer l'information

```
@Dao
interface AuthorDao {
    @Query("SELECT * FROM authors WHERE bookId = :bookId")
    fun getBookAuthors(bookId: Long) : List<Author>
    …
}
```

- Ne pas oublier de mettre à jour le paramètre `entities` de la `@Database`

Clé(s) étrangère(s)

- Autre possibilité pour avoir une entité plus « orientée objet »
- Utiliser l'annotation `@Relation`
- On ne définit plus de clé étrangère pour la classe `Author`
- On rajoute une classe POKO pour représenter l'association livre / auteurs

```
data class BookWithAuthors(@Embedded val book: Book,
                           @Relation(parentColumn = "id",
                                     entityColumn = "bookId")
                                     val authors: List<Author>)


@Dao
interface AuthorDao {
    @Transaction
    @Query("SELECT * FROM books")
    fun getBooksWithAuthors(): List<BookWithAuthors>
}
```

Clé(s) étrangère(s) (relation M-N)

- Même principe pour le cas d'une relation M-N : `@ForeignKey` mais x2

```
@Entity(tableName = "books")
data class Book(@PrimaryKey(autoGenerate = true) val id: Long,
                var title: String,
                @ColumnInfo(name = "num_pages") var nbPages: Int)

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

- Les entités n'ont plus directement l'information de leur relation

Clé(s) étrangère(s) (relation M-N)

- Par contre il faut une nouvelle table pour stocker chaque association livre/auteur

```
@Entity(tableName = "book_author",
        primaryKeys = ["bookId", "authorId"],
        foreignKeys = [ForeignKey(entity = Book::class,
                                  parentColumns = ["id"],
                                  childColumns = ["bookId"]),
                       ForeignKey(entity = Author::class,
                                  parentColumns = ["id"],
                                  childColumns = ["authorId"])]
)
data class BookAuthorJoin(val bookId: Long, val authorId: Long)
```

Clé(s) étrangère(s) (M-N)

- On fait des JOIN sur les tables pour récupérer les infos
- Comme avant : rajouter un DAO et mettre à jour `@Database`

```
@Dao
interface BookAuthorJoinDao {
    @Insert
    fun insert(bookAuthorJoin: BookAuthorJoin)

    @Query("SELECT * FROM book INNER JOIN book_author ON" +
           "book.id = book_author.bookId WHERE" +
           "book_author.authorId = :authorId")
    fun getBooksFromAuthor(authorId: Long): List<Book>

    @Query("SELECT * FROM author INNER JOIN book_author ON" +
           "author.id = book_author.authorId WHERE" +
           "user_repo.bookId = :bookId")
    fun getBookAuthors(bookId: Long): List<Author>
    …
}
```

`Relation` pour le cas M-N

- Avec l'annotation `@Relation` cela donne

```
@Entity(primaryKeys = ["bookId", "authorId"])
data class BookAuthorRel(val bookId: Long, val authorId: Long)

data class BookWithAuthors(
    @Embedded
    val book: Book,
    @Relation(parentColumn = "id",
              entityColumn = "authorId",
              associateBy = Junction(BookAuthorRel::class))
    val authors: List<Author>
)

@Dao
interface AuthorDao {
    @Transaction
    @Query("SELECT * FROM books")
    fun getBooksWithAuthors(): List<BookWithAuthors>
}
```

Éléments retournés

- DAO : pas obligatoire de retourner des entités
- On peut utiliser des classes non annotées du moment qu'un mapping est possible entre les attributs et les champs de la table
- Permet de ne pas gaspiller inutilement de la mémoire

```
data class BookPages(val id: Long, var nbPages: Int)

@Dao
interface BookDao {
    @Query("SELECT id, num_pages AS nbPages " +
           "FROM books")
    fun getBooksPages(): List<BookPages>
}
```

Content Provider

- Une application crée et gère des données (dans son espace privé)
- Une autre application aimerait disposer de ces données
- Pas possible directement
- C'est le rôle du `ContentProvider`
- Centralisation des données et « distribution » contrôlée
- L'application « serveur de données » implémente le `ContentProvider`
- Les application « clientes » utilisent un `ContentResolver`
- Accès aux données à travers des URIs

Content Provider

- Implémentation d'un Content Provider :
  - Choix du stockage (fichiers ? base de données ? service web)
  - Définition des URIs pour l'accès aux données : `content://<authority>/<path_to_data>/<id>`
  - Implémentation d'une classe dérivée de `ContentProvider`
  - Implémentation des opérations CRUD
- Si stockage = BdD, `ContentProvider` $\simeq$ wrapper autour de la BdD
- Doit être déclaré dans le manifest

```
<provider android:name=".MyContentProvider"
          android:authorities="myapp.provider.ContentData"
          android:readPermission="myapp.permission.READ"
          android:writePermission="myapp.permission.WRITE" />
```