Data Management

SQLite

Android permette ad una qualsiasi applicazione sviluppata di usufruire di molteplici *database*, adoperando il linguaggio *SQL* per formulare interrogazioni alle collezioni di dati.

Rispetto a quanto riportato, il framework in questione mette a disposizione specifiche interfacce affinchè sia possibile interagire con i dati mantenuti all'interno delle tabelle. L'utilizzo di database relazionali è fortemente consigliato qualora non sia sempre possibile garantire connettività; pertanto, mediante le collezioni di dati, è mantenuta una copia dell'informazioni ritenute fondamentali per il conseguimento dell'obiettivo dell'applicazione sviluppata.

Tipicamente è attuato un certo paradigma per estrapolare informazioni da un database; innanzitutto, occorre creare una classe *DBHelper* che estenda l'interfaccia *SQLiteHelper*, la quale svilupperà l'insieme dei metodi necessari per manipolare la base di dati.

Una buona pratica, qualora si interagisca con un *database*, consiste nella definizione di un *companion object* che identifica tutte le colonne della collezione circoscritta.

```
companion object StudentContract {
  object StudentEntry : BaseColumns {
    const val TABLE_NAME = "students"
    const val COLUMN_NAME = "name"
    const val COLUMN_CLASS = "class"
    const val COLUMN_GRADE = "grade"
  }
}
```

Successivamente avviene l'implementazione della classe *SQLiteHelper*, agendo, mediante *override*, sulla funzione *onCreate()*.

```
class DbHelper(context: Context) :
  SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {

  override fun onCreate(db: SQLiteDatabase) {
    db.execSQL(
      "CREATE TABLE ${StudentEntry.TABLE_NAME} (${BaseColumns._ID} INTEGER
          PRIMARY KEY, " +
      "${StudentEntry.COLUMN_NAME} TEXT, ${StudentEntry.COLUMN_CLASS} TEXT," +
      "${StudentEntry.COLUMN_GRADE} TEXT)"
    )
  }
  companion object { const val DATABASE_NAME = "Students.db", const val
      DATABASE_VERSION = 1 }
}
```

Inoltre, proseguendo, possono essere adeguate le tipiche operazioni, attuando qualche accorgimento a seconda dell'azione che voglia essere conseguita, definite come segue:

- **Insert**

```
val db = dbHelper.writableDatabase // riferimento "modificabile" del database

/* sintassi propria per l'inserimento di record */
val values = ContentValues().apply {
```

```
  put(StudentEntry.COLUMN_NAME, "Mario Rossi")
  put(StudentEntry.COLUMN_CLASS, "LAM")
  put(StudentEntry.COLUMN_GRADE, "30")
}

val newRowId = db?.insert(StudentEntry.TABLE_NAME, null, values)
```

- **Update**

```
val db = dbHelper.writableDatabase

val values = ContentValues().put(StudentEntry.COLUMN_GRADE, "29")

/* definita dipendency injection, data dal carattere "?" */
val selection = "${StudentEntry.COLUMN_NAME} LIKE ?"
val selectionArgs = arrayOf("Mario Rossi") // bind dei params
val count = db.update(
    StudentEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs)
```

- **Delete**

```
val db = dbHelper.readableDatabase
val selection = "${StudentEntry.COLUMN_NAME} LIKE ?"
val selectionArgs = arrayOf("Mario Rossi")

val deletedRows = db.delete(StudentEntry.TABLE_NAME, selection, selectionArgs)
```

- **Query**

```
val db = dbHelper.readableDatabase

val projection = arrayOf(BaseColumns._ID, StudentEntry.COLUMN_NAME)

val cursor = db.query(
  StudentEntry.TABLE_NAME,
  projection,
  "${StudentEntry.COLUMN_GRADE} = ?",
  arrayOf("30"),
  null,
  null,
  null
)
```

Nota: qualsiasi operazione che richieda l'interazione con un database, può essere potenzialmente bloccante per l'intero sistema ideato, pertanto è necessario delegarne l'esecuzione a *threads* oppure *coroutine*.

Solitamente è restituito un **Cursor** ad ogni query effettuata, il quale adotta un meccanismo simile ad un *iteratore*; permette di scorrere il risultato ottenuto per ogni record che componga la tabella risultante dall'interrogazione attuata.

```
val items = mutableListOf<String>()
with(cursor) {
  while(moveToNext()) {
```

```
    val item = getString(getColumnOrThorw(StudentEntry.COLUMN_NAME))
    items.add(item)
  }
}
cursor.close() // molto simile alla sintassi di un iterator in Java
```

Concludendo, onde evitare errori, occorre chiudere la connessione con il *database* di riferimento, affinchè sia possibile accedere nuovamente all'informazioni stilate al suo interno.

```
override fun onDestroy() {
  deHelper.close()
  super.onDestroy()
}
```

Content Providers

Un **content provider** definisce un sistema per accedere a dati condivisi, attuato principalmente per rendere disponibili informazioni ad altre applicazioni. Il proprio comportamento è simile ad un servizio *Web REST*, pertanto adopera *Uniform Resource Identificator* per definire il metodo di acquisizione delle risorse; infatti, lo stesso, è visualizzato da componenti esterni come un *database* a cui dettare *interrogazioni*.

Consuetudine principe di un *content provider* promuove l'acquisizione di dati da parte di ulteriori applicazioni installate nel dispositivo, tipicamente ritraggono servizi già presenti al suo interno; esempi consistono nell'elenco delle sveglie impostate dall'utente oppure l'accesso ai contatti memorizzati in rubrica. In queste casistiche è obbligatorio adoperare un *provider*, publicizzato mediante *URI* definiti, a loro volta, all'interno del *Manifest*.

Di seguito, a titolo esemplificativo, è sviluppato uno dei differenti approcci per acquisire i contatti memorizzati in rubrica; tuttavia, la realtà descritta, richiede che sia formulato un permesso all'interno del *Manifest*, dato che si manipolano dati sensibili dell'utente, definito sia a livello di lettura che di scrittura.

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

```
cursor = contentResolver.query(
  ContactsContract.Contacts.CONTENT_URI,
  null,
  null,
  null,
  null
)

while (cursor.moveToNext()) {
  val item = cursor.getString(
      cursor.getColumnIndexOrThrow(ContactsContract.Contacts.DISPLAY_NAME))
}
```

Come da sketch di codice, si visualizza la terminologia descritta nella parte introduttiva della sezione; infatti, è formulata una query affinchè sia possibile acquisire i contatti memorizzati in rubrica ed è restituito un *cursore* per scorrere il risultato ottenuto. A tutti gli effetti è riportata la manipolazione di un *database*.

Room

Room simboleggia un insieme di librerie attuate per strutturare interazioni con *database relazionali*. Definisce un *layer* astratto posto al di sopra di *SQLite*; tendenzialmente attuato qualora il progetto sviluppato sia sufficientemente complesso.

È caratterizzato da tre componenti principali, contraddistinti in:

- **Entities**, tabelle che compongono il database
- **DAOs**, acronimo di *Data Access Objects*, interfaccia contenente l'elenco dei metodi necessari per interrogare e manipolare il database
- **Database**, punto principale di accesso alle risorse contenute

Lo scopo di *Room* consiste nella previa definizione del codice di accesso e di sviluppo in completa autonomia, consecutivo a specifiche direttive, in modo tale che le librerie siano in grado di codificare il comportamento richiesto durante la fase di compilazione.

Per usufruire dello strumento è necessario dichiarare e stabilire una *classe astratta* che estenda la classe *RoomDatabase*.

```
@Database(entities = [Entity1::class, Entity2::class], version = 1, exportSchema
    = false)
abstract class myDatabase : RoomDatabase() {
  abstract fun entity1Dao() : Entity1Dao
  abstract fun entity2Dao() : Entity2Dao
  abstract fun twoEntitiesDao() : TwoEntitiesDao
}
```

Mediante i *DAOs*, le librerie di *Room* sono in grado di convertire le interazioni con il *database* in vere e proprie classi attinenti al progetto sviluppato, semplicemente dichiarando i parametri di *input* e di *output*.

```
@Dao
interface MyDao {
  @QueryType(params...)
  fun dbMethod(params): ReturnType
}
```

Proseguendo, per ciascuna *Entity* è creata ed associata una tabella del database, in cui ogni *field* referenzia ad un dominio della collezione, ad eccezione delle colonne demarcate con la key-word *@ignore*.

```
@Entity(tablename = "my_entity")
class Entity1 (
  @PrimaryKey
  @NonNull
  @columnInfo(name = "index")
  var id: String,
  var field1: String,
  @Ignore
  var temp: String
)
```

Concludendo, anche in questa casistica, è bene adeguare *coroutine* oppure *thread pool* per l'implementazione e sviluppo mediante *DAOs*.

HTTP

HTTP è il protocollo di rete per il trasferimento di dati, il quale si avvale di numerosi modelli di comunicazione. Storicamente *Android* supporta due librerie implementative principali, contraddistinte in:

- **HTTPClient**, estensione completa del protocollo *HTTP* designata per attività di navigazione sul Web
- **HTTPUrlConnection**, implementazione adeguata per il trasferimento di dati mediante un'architettura *client-server*

In entrambe le casistiche occorre gestire le connessioni attraverso *flussi di esecuzione* separati.

A titolo esemplificativo sono riportati gli step necessari per imbastire correttamente una connessione *HTTP*, suddivisi come segue:

- Definire una nuova connessione *HTTPUrlConnection* richiamando la funzione *openConnection()*

```
val url: URL = URL("https://www.android.com/")
val urlConnection: HttpURLConnection = url.openConnection() as
    HttpURLConnection
```

- Definire le caratteristiche della *request*; qualora debba essere utilizzato il verbo *POST* occorre invocare il metodo *setDoOutput()*

```
urlConnection.doOutput = true
urlConnection.requestMethod = "POST"
urlConnection.setChunkedStreamingMode(0)
val out: OutputStream = BufferedOutputStream(urlConnection.outputStream)
out.write("YourPostInput".toByteArray())
```

- Lettura della *response* corrisposta alla richiesta precedente; tipicamente l'oggetto risposta contiene *data* e *header*, mentre per accedere al *body* è necessario manipolare lo *stream* mediante *getInputStream()*

```
inStream: InputStream = BufferedInputStream(urlConnection.inputStream);
```

- Conclusa la lettura della *response* è norma chiudere la connessione

```
urlConnection.disconnect()
```

Volley

Volley è una libreria attuata per elaborare azioni *HTTP*. Possiede differenti *features*, in grado di gestire autonomamente meccanismi di *caching* e *chiamate asincrone*.

Le operazioni di rete sono molto onerose per i dispositivi, in termini di tempo e batteria, causate dal continuo traffico dati. Tali motivazioni richiedono una serie di attività mirate ad ottimizzare le prestazioni del device, come stabilire una cache delle richieste effettuate oppure adeguare le operazioni *HTTP* in determinate circostanze.

Volley promuove un atteggiamento simile a quanto riportato; quando il *client* formula una

request, il sistema di librerie si accerta che la stessa sia stata già effettuata, supervisionando la *queue* di richieste. In caso affermativo sarà restituita la *response* in locale, altrimenti verrà effettivamente effettuata la richiesta.

Per usufruire dei meccanismi di *Volley* occorre aggiungere le proprie librerie a *gradle*.

```
implementation 'com.android.volley:volley:1.1.1'
```

Installate le librerie, affinchè sia possibile formulare una *request* ad un determinato servizio *HTTP*, occorre definire i seguenti comandi

```
RequestQueue queue = Volley.newRequestQueue(this);
StringRequest stringRequest = new StringRequest(Request.Method.GET, baseUrl,
  new Response.Listener<String>() {
    @Override
    public void onResponse(String response) { /* do your stuff */ }
  }, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) { /* do your stuff */ }
  });
queue.add(stringRequest);
```

Inoltre, mediante le stesse librerie, è possibile personalizzare gli *header* attuando l'*override* della funzione *getHeaders()*.

```
{
  @Override
  public Map<String, String> getHeaders() {
    Map<String, String> params = new HashMap<String, String>();
    params.put("x-vacationtoken", "secre_token");
    params.put("content-type", "application/json");
    return params;
  }
}
```