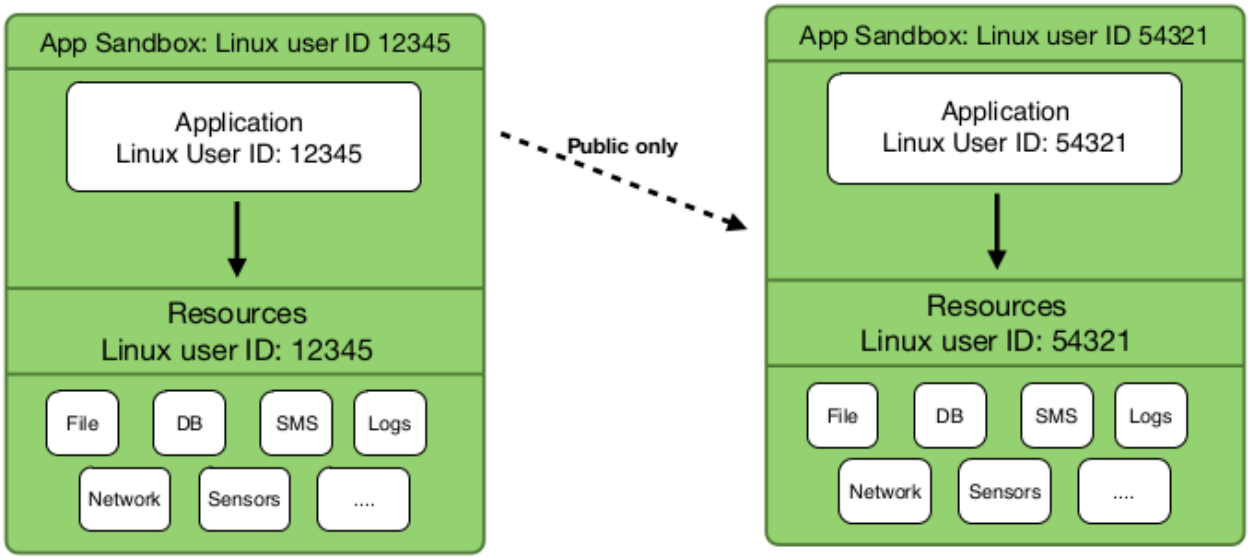
7-Data Persistence

Android & Data Persistence

La memorizzazione dei dati può avvenire in:

- Shared Preferences, memorizza dati primitivi in una struttura chiave-valore
- Internal Storage & External Storage, memorizza i dati su dispositivi di memoria (dati privati) o su memoria esterna (dati pubblici)
- SQLite Database, magazzino struttirato in un database privato
- Nerwork Conntection, memorizza i dati su un servere/servizio remoto

> Nota
> I dati privati sono accessibili solo all'interno dell'applicazione dallo stesso sviluppatore, i dati pubblici sono invece visibili dalle altre applciazioni



Shared Preferences

Molte applicazioni necessitano di un meccanismo di archiviazione dei dati leggero per memorizzare lo stato dell'applicazione, le opzioni di configurazione, le informazioni semplici dell'utente e altri dati dell'utente. Questo meccanismo sulla piattaforma Android è chiamato "Shared Preferences" (Preferenze Condivise) e fornisce un sistema di preferenze semplice per memorizzare dati primitivi dell'applicazione a livello di activity.

> Nota
> Le preferenze non sono condivise tra tutte le attività dell'applicazione e non è possibile condividere le preferenze al di fuori del pacchetto dell'applicazione

I tipi di dato supportati sono:

- `Boolean`
- `Float`
- `Integer`
- `Long`
- `String`

La classe `SharedPreferences` fornisce un framework generale che consente di salvare e recuperare coppie chiave-valore persistenti di tipi di dati primitivi, per prendere con un oggetto di questa classe nell'applicazione sono disponibili 2 metodi:

- `getSharedPreferences()`, se si ha bisogno di più file di preferenze identificati per nome (si specifica con il primo parametro)
- `getPreferences()`, se si ha bisogno di un solo file di preferenze per l'activity. Poiché questo sarà l'unico file di preferenze per l'activity, non è necessario fornire un nome

Per scrivere bisogna:

1. Chiamare `edit()` per ottenere un `SharedPreferences.Editor`
2. Aggiungere i/il valori/e con i metodi tipo `putBoolean()` e `putString()`
3. Memorizzare i nuovi valori con `commit()`

Per leggere i valori, si utilizzano i metodi di SharedPreferences come:

- `getBoolean()`
- `getString()`

```
public class Calc extends Activity {
        public static final String PREFS_NAME = "MyPrefsFile";
        @Override
        protected void onCreate(Bundle state){
                super.onCreate(state);
                // ...
                // Legge le SharedPreferences
                SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
                boolean silent = settings.getBoolean("silentMode", false);
                // Altri metodi sono getFloat(...), getInt(...), getLong(...), getString(...)
        }
        @Override
        protected void onStop(){
                super.onStop();
                SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
```

```
                SharedPreferences.Editor editor = settings.edit();
                editor.putBoolean("silentMode", mSilentMode);
                // Altri metodi sono methods sono putFloat(...), putInt(...), putLong(...),
putString(...)

                // Commit the edits!
                editor.commit();
        }
}
```

Internal Storage/Application Directories

I dati dell'applicazione Android vengono memorizzati nel file system nella directory `/data/data/<package_name>/`. In particolare, l'API consente di ottenere riferimenti per elencare, leggere e scrivere file nelle seguenti sottodirectory:

- `/data/data/<package_name>/files`
- `/data/data/<package_name>/cache`

Vengono creati diversi sottodirectory predefiniti per memorizzare database, preferenze e file secondo necessità.

> Nota
> È inoltre possibile creare una directory personalizzata secondo le esigenze

L'oggetto Context viene utilizzato per eseguire operazioni sui file e interagire con il file system per la gestione dei file dell'applicazione.

> Nota
> Si può anche utilizzare tutte le utilità standard del pacchetto `java.io` per lavorare con oggetti `FileStream`

Si possono salvare file direttamente nella memoria interna del dispositivo, di default sono salvati come file privati dell'applicazione e le altre applicazioni (nemmeno l'utente) non possono accedervi.

> Nota
> Quando l'applicazione viene disinstallata i file privati dell'applicazione vengono rimossi dalla memoria interna

Write to Internal Storage

Per creare e scrivere un file privato all'interno della memoria:

1. Bisogna chiamare il metodo `openFileOutput()` con il nome del file e la modalità di esecuzione, che restituirà un `FileOutputStream`
2. Bisogna scrivere sul file con il metodo `write()`
3. Bisogna chiudere lo stream con `close()`

```
String filename = "hello_file";
String string = "hello world!";
FileOutputStream fos = openFileOutput(filename, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

Il flag `MODE_PRIVATE` indica che il file creato (o sostituito se era presente un file con lo stesso nome) e lo renderà privato per l'applicazione. Altri modi disponibili sono:

- `MODE_APPEND`
- `MODE_WORLD_READABLE`
- `MODE_WORLD_WRITEABLE`

Read from Internal Storage

Per leggere un file dalla memoria interna:

1. Bisogna chiamare il metodo `openFileInput()` e passare il nome del file da leggere, che restituirà un `FileInputStream`
2. Leggi i byte dal file utilizzando il metodo`read()`
3. Chiudere lo stream con il metodo `close()`

```
String filename = "hello_file";
FileInputStream fis = openFileInput(filename);
StringBuffer sBuffer = new StringBuffer();
DataInputStream dataIO = new DataInputStream(fis);
String strLine = null;
while((strLine = dataIO.readLine()) != null){
```

```
        sBuffer.append(strLine+"\n");
}
dataIO.close();
fis.close();
```

C'è una scorciatoia per leggere i file memorizzati nella sottodirectory predefinita `/files` utilizzando il metodo `openFileInput` per ottenere un riferimento `FileInputStream` e leggere il contenuto del file utilizzando le API standard di Java. Ad esempio, si può utilizzare un oggetto `DataInputStream` per leggere un file riga per riga e memorizzarlo in un `StringBuffer`.

External Storage

Ogni dispositivo compatibile con Android supporta una memoria esterna condivisa chiamata "external storage" che si può utilizzare per salvare file, questa memoria può essere costituita da una scheda SD o da una memoria interna non rimovibile.
I file salvati nella memoria esterna sono leggibili da tutti gli utenti del dispositivo e possono essere modificati dall'utente quando abilitano la modalità di archiviazione di massa USB per trasferire file su un computer.

> Nota
> Bisogna fare attenzione, i file esterni possono scomparire se l'utente monta la memoria esterna su un computer o rimuove il supporto, e non vi è alcuna sicurezza imposta sui file salvati nella memoria esterna. Tutte le applicazioni possono leggere e scrivere file collocati nella memoria esterna e l'utente può eliminarli

Prima di effettuare qualsiasi operazione sulla memoria esterna, bisognerebbe sempre chiamare `getExternalStorageState()` per verificare se il supporto è disponibile, perché potrebbe non esserlo e risultare:

- Montato su un computer
- Mancante
- In sola lettura
- In altro stato

Il metodo `getExternalStorageState()` permette di verificare se la memoria esterna è disponibile per la lettura e la scrittura, restituisce anche altri stati che si potrebbe voler controllare:

- Il supporto è condiviso (connesso a un computer),
- È completamente assente
- È stato rimosso in modo improprio
- Ecc...

Si può utilizzare queste informazioni per notificare all'utente maggiori dettagli quando la tua applicazione ha bisogno di accedere alla memoria.

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(state)) {
        // We can read and write the media
        mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        // We can only read the media
        mExternalStorageAvailable = true;
        mExternalStorageWriteable = false;
} else {
        // Something else is wrong. It may be one of many other states, but all we need
        // to know is we can neither read nor write
        mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

Se si sta utilizzando API Level 8 o superiore, bisogna utilizzare `getExternalFilesDir()` per aprire un oggetto File che rappresenta la directory di archiviazione esterna dove si dovrebbe salvare i file. Questo metodo accetta un parametro `type` che specifica il tipo di sottodirectory desiderato:

- `DIRECTORY_MUSIC`
- `DIRECTORY_RINGTONES`
- `null`, per ottenere la radice della directory dei file dell'applicazione

> Nota
> Questo metodo creerà la directory appropriata se necessario.

Specificando il tipo di directory, si garantisce che il media scanner di Android categorizzi correttamente i file nel sistema (ad esempio, i suonerie verranno identificati come suonerie e non come musica).

> Nota
> Se l'utente disinstalla la tua applicazione, questa directory e tutti i suoi contenuti saranno eliminati

Se si sta utilizzando API Level 7 o inferiori, bisogna utilizzare `getExternalStorageDirectory()` per aprire un oggetto `File` che rappresenta la radice della memoria esterna, e quindi scrivere i dati nella directory `/Android/data/<package_name>/files/` dove `<package_name>` è il nome del pacchetto in stile Java (ad esempio `com.example.android.app`).

```
File dir = Environment.getExternalStorageDirectory();
String path = dir.getAbsolutePath();
String fileName = "mobdev_bookmarklist.txt";
File outputFile = new File(path + File.separator + fileName);
if(!outputFile.exists())
       outputFile.createNewFile();
FileOutputStream fos = new FileOutputStream(outputFile);

fos.write(...);
fos.close();
```

Per poter scrivere file nella memoria esterna, un'applicazione deve dichiarare un'autorizzazione specifica chiamata `WRITE_EXTERNAL_STORAGE` nel file `ApplicationManifest.xml`.

permesso pericoloso

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

External Storage Update

Category	Type of content	Access method	Permissions needed	Can other apps access?	Files removed on app uninstall?
App-specific files	Files meant for your app's use only	From internal storage, getFilesDir() or getCacheDir() From external storage, getExternalFilesDir() or getExternalCacheDir()	Never needed for internal storage Not needed for external storage when your app is used on devices that run Android 4.4 (API level 19) or higher	No, if files are in a directory within internal storage Yes, if files are in a directory within external storage	Yes
Media	Shareable media files (images, audio files, videos)	MediaStore API	READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE when accessing other apps' files on Android 10 (API level 29) or higher Permissions are required for all files on Android 9 (API level 28) or lower	Yes, though the other app needs the READ_EXTERNAL_STORAGE permission	No
Documents and other files	Other types of shareable content, including downloaded files	Storage Access Framework	None	Yes, through the system file picker Use startActivityForResult()	No

Reading XML Files

LaSDK include diverse utility per lavorare con i file XML, tra cui SAX e XML Pull Parser, e limited DOM, Livello 2 Core.

In particolare pacchetti utili sono:

Metodo	Scopo
`android.sax.*`	Framework per scrivere gestori SAX standard
`android.util.Xml.*`	Utilità XML che includono XMLPullParser
`org.xml.sax.*`	Funzionalità di base di SAX
`javax.xml.*`	Supporto per SAX e DOM limitato, Livello 2 Core
`org.w3c.dom`	Interfaccia per DOM, Livello 2 Core
`org.xmlpull.*`	Interfacce XmlPullParser e XMLSerializer, nonché una classe di driver SAX2

JSON

JSON (o JavaScript Object Notation), è uno standard aperto leggero basato su testo, progettato per uno scambio di dati leggibile dall'uomo. Deriva dal linguaggio di scripting JavaScript ed è utilizzato per rappresentare strutture dati semplici e array associativi, chiamati oggetti. Nonostante la sua relazione con JavaScript, è indipendente dal linguaggio, con parser disponibili per la maggior parte dei linguaggi.

```
public class BookmarkDescriptor {
	private String name = null;
	private String url = null;
	public BookmarkDescriptor(){}
	public BookmarkDescriptor(String name, String url) {
		this.name = name;
		this.url = url;
	}
}
```

```
{
	"name":"Unipr",
	"url":"http://www.unipr.it"
}
```

GSON Library

Gson è una libreria Java che può essere utilizzata per convertire gli oggetti Java nella loro rappresentazione JSON. Può anche essere utilizzata per convertire una stringa JSON in un oggetto Java equivalente.

> Nota
> Gson può lavorare con oggetti Java arbitrari, inclusi oggetti preesistenti per i quali non si dispone del codice sorgente

Si può dire che GSON:

- Fornire meccanismi facili da utilizzare come `toString()` e constructor (metodo di fabbrica) per convertire Java in JSON e viceversa
- Consentire la conversione in JSON e viceversa di oggetti preesistenti e non modificabili
- Consentire rappresentazioni personalizzate per gli oggetti
- Supportare oggetti arbitrariamente complessi
- Generare output JSON compatto e leggibile

dipendenze

```
dependencies{
	implementatiom 'com.google.code.gson:gson:2.8.6'
}
```

La classe principale da utilizzare è `Gson`, che puoi semplicemente creare chiamando `new Gson()`.

> Nota
> È disponibile anche una classe `GsonBuilder` che può essere utilizzata per creare un'istanza di Gson con varie impostazioni come il controllo della versione, e così via

Esempi dati primitivi

```
// serializzazione
Gson gson = new Gson();

gson.toJson(1); // ==> prints 1
gson.toJson("abcd"); // ==> prints "abcd"
gson.toJson(new Long(10)); // ==> prints 10
int[] values = { 1 };
gson.toJson(values); // ==> prints [1]

// deserializzazione

int one = gson.fromJson("1", int.class);
Integer one = gson.fromJson("1", Integer.class);
Long one = gson.fromJson("1", Long.class);
Boolean false = gson.fromJson("false", Boolean.class);
String str = gson.fromJson("\"abc\"", String.class);
String anotherStr = gson.fromJson("[\"abc\"]", String.class);
```

Esempi oggetti

```

// Classe esempio
class BagOfPrimitives {
	private int value1 = 1;
	private String value2 = "abc";
```

```
        private transient int value3 = 3;
        BagOfPrimitives() {
        // no-args constructor
        }
}

// Serializzazione
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj); // ==> {"value1":1,"value2":"abc"}

// Deserializzazione

BagOfPrimitives obj2 = gson.fromJson(json, BagOfPrimitives.class); // Uguale a obj
```

SQLite Database

Se l'applicazione Android richiede una soluzione di archiviazione dati più solida, la piattaforma Android include e fornisce il supporto per database relazionali specifici per l'applicazione utilizzando SQLite.

> Nota
> Qualsiasi database creato sarà accessibile per nome a qualsiasi classe nell'applicazione, ma non all'esterno dell'applicazione

Il database SQLite è una soluzione leggera basata su file che rimuove funzionalità non strettamente necessarie per ottenere un'occupazione ridotta di spazio, è stato progettato per gestire molte tipologie di guasti di sistema, come la scarsa memoria, gli errori del disco e le interruzioni di alimentazione.

> Nota
> Il Progetto SQLite non è un Progetto di Google, ma un progetto indipendente con un team internazionale di sviluppatori software

SQLite è basato sul linguaggio SQL (Structured Query Language), un linguaggio di programmazione progettato per gestire dati nei sistemi di gestione di database relazionali (RDBMS).

Per ulteriori informazioni dettagliate e la documentazione, puoi visitare i seguenti siti:

- sqlite (http://www.sqlite.org/).
- sqlite/lang (http://www.sqlite.org/lang.html)

Android Database Library

L'Android SDK include diversi utili classi di gestione del database SQLite, molte di queste classi si trovano nel pacchetto `android.database.sqlite`.
Le principali classi sono:

- `SQLiteDatabase`, un interfaccia Java per SQLite, che supporta un'implementazione SQL sufficientemente completa per qualsiasi necessità di un'applicazione mobile
- `Cursor`, che è un contenitore per i risultati di una query del database, che supporta un sistema di tipo MVC. Utilizzando un Cursor, è possibile navigare tra i risultati della query e accedere ad ogni riga quando necessario, i principali metodi sono:
  - `Cursor.getAs*(int columnNumber)` (ad esempio, `getAsString`) per accedere ai dati disponibili
  - `Cursor.moveToNext()` o `Cursor.moveToPrevious()` per cambiare l'indice del cursore corrente
- `SQLiteOpenHelper`, che fornisce un framework del ciclo di vita per la creazione e l'aggiornamento del database dell'applicazione
- `SQLiteQueryBuilder`, che fornisce un'astrazione di alto livello per creare query SQLite da utilizzare nelle applicazioni Android

> Nota
> Utilizzando la classe `SQLiteQueryBuilder` è possibile semplificare il compito di scrivere query

Database Creation

L'approccio consigliato per creare un nuovo database SQLite su Android è quello di creare una sottoclasse di `SQLiteOpenHelper` e sovrascrivere i metodi `onCreate()`, che viene chiamato automaticamente quando l'applicazione viene avviata per la prima volta per creare il database, e `onUpgrade()`.

Quando nuove versioni dell'applicazione vengono distribuite, potrebbe essere necessario aggiornare il database:

- Aggiungendo nuove tabelle
- Aggiungendo nuove colonne
- Modificando l'intero schema

Quando ciò è necessario, il compito è associato al metodo `onUpgrade()`, che viene chiamato ogni volta che il valore `DATABASE_VERSION` nella chiamata al costruttore è diverso da quello memorizzato nel database.

> Nota
> Quando viene distribuita una nuova versione del database, è necessario incrementare il numero di versione per consentire l'effettivo aggiornamento

```
public class LogDescriptorOpenHelper extends SQLiteOpenHelper {


        private static final int DATABASE_VERSION = 1;
        private static final String DATABASE_NAME = "log.db";
        public static final String TABLE_NAME = "log";

        public static final String TIMESTAMP_COL = "timestamp";
        public static final String LATITUDE_COL = "latitude";
        public static final String LONGITUDE_COL = "longitude";
        public static final String TYPE_COL = "type";
        public static final String DATA_COL = "data";
        public static final String ID_COL = "id";

        // Query per creare la tabella del database
        private static final String DATABASE_TABLE_CREATE =
                                        "CREATE TABLE " + TABLE_NAME + " (" +
                                        ID_COL + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                                        TIMESTAMP_COL + " TIMESTAMP, " +
                                        LATITUDE_COL + " DOUBLE,"+
                                        LONGITUDE_COL + " DOUBLE,"+
                                        TYPE_COL + " VARCHAR(50),"+
                                        DATA_COL + " TEXT"+
                                        ");";
        public LogDescriptorOpenHelper(Context context) {
                super(context, DATABASE_NAME, null, DATABASE_VERSION);
                Log.d(LoggerAppActivity.TAG, "LogOpenHelper Constructor !");
        }
                @Override
        public void onCreate(SQLiteDatabase db) {
                Log.d(LoggerAppActivity.TAG,"LogOpenHelper onCreate !");
                db.execSQL(DATABASE_TABLE_CREATE);
        }
        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
                Log.d(LoggerAppActivity.TAG,"LogOpenHelper onUpgrade !");
                db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
                db.execSQL(DATABASE_TABLE_CREATE);
        }
}
```

Gli elementi generali associati a un SQLiteOpenHelper sono:

- `DATABASE_NAME`, che contiene il nome del file del database che identifica il file effettivo del database SQLite. La piattaforma Android si occuperà di creare il file del database per te in `` `/data/data/<nome_pacchetto>/databases/DATABASE_NAME` ``.
- `DATABASE_VERSION` che definisce la versione del database, che si può scegliere arbitrariamente e si incrementa ogni volta che viene modificato lo schema del database
- `TABLE_NAME`, che è il nome della tabella che verrà creata
- `Constructor`, il costruttore per il database, che utilizza la funzione super per chiamare il costruttore del genitore che creerà l'oggetto del database
- `onCreate`, il metodo che il framework esegue per inizializzare il database. Questo metodo conterrà il codice per l'inizializzazione del database e la creazione delle tabelle

SQLite Database Interaction

Si può eseguire query SQLite utilizzando i metodi `query()` di `SQLiteDatabase`, che accettano vari parametri di query come

- La tabella su cui eseguire la query
- La proiezione
- La selezione
- Le colonne
- Il raggruppamento
- ecc...

Per query complesse, come quelle che richiedono alias di colonne, bisognerebbe utilizzare `SQLiteQueryBuilder`, che fornisce diversi metodi comodi per costruire le query.

Ogni query SQLite restituirà un Cursor che punta a tutte le righe trovate dalla query.

> Nota
> Il Cursor è sempre il meccanismo con cui si può navigare tra i risultati di una query del database e leggere righe e colonne.

Per aggiungere un nuovo record in una tabella, si pò utilizzare il metodo `insert(...)` specificando:

- La tabella in cui inserire la riga
- Una mappa con i valori delle colonne per la riga rappresentata da un oggetto chiamato ContentValues.

Per rimuovere record dal database si può utilizzare il metodo `remove(...)` che prende come argomenti:

- La tabella da cui eliminare il record
- Due argomenti che descrivono le condizioni `WHERE`

Database Connection

Per interagire con il database, bisognerebbe ottenere un'istanza di `SQLiteDatabase` attraverso una propria sottoclasse `OpenHelper`, il code seguente mostra un possibile flusso per recuperare l'oggetto lavorare con il DB e chiudere la connesione

```
{
        LogDescriptorOpenHelper dbHelper = new LogDescriptorOpenHelper(context);
        SQLiteDatabase database = dbHelper.getWritableDatabase();
        //... DB Operations ...
        dbHelper.close();
}
```

DB Query

```
public List<LogDescriptor> getAllLogsDescriptor() {
        String[] allColumns = {
                LogDescriptorOpenHelper.ID_COL,LogDescriptorOpenHelper.TIMESTAMP_COL,
                LogDescriptorOpenHelper.LATITUDE_COL, LogDescriptorOpenHelper.LONGITUDE_COL,
                LogDescriptorOpenHelper.TYPE_COL, LogDescriptorOpenHelper.DATA_COL
        };

        List<LogDescriptor> logList = new ArrayList<LogDescriptor>();

        Cursor cursor =
database.query(LogDescriptorOpenHelper.TABLE_NAME,allColumns,null,null,null,null,null,null);
        cursor.moveToFirst();

        while (!cursor.isAfterLast()) {
                LogDescriptor logDescr = cursorToLogDescriptor(cursor);
                comments.add(logDescr);
                cursor.moveToNext();
        }
        // Make sure to close the cursor
        cursor.close();
        return comments;
}
```

Leggi i valori dai Cursor (con `cursor.get_DataType*(<column*index>*)` `) per creare un oggetto` *LogDescriptor`*

```
private LogDescriptor cursorToLogDescriptor(Cursor cursor) {
        LogDescriptor logDescriptor = new LogDescriptor();
        logDescriptor.setId(cursor.getInt(0));
        logDescriptor.setTimestamp(cursor.getLong(1));
        logDescriptor.setLatitude(cursor.getLong(2));
        logDescriptor.setLongitude(cursor.getLong(3));
        logDescriptor.setType(cursor.getString(4));
        logDescriptor.setData(cursor.getString(5));
        return logDescriptor;
}
```

Insert & Query

```
// Crea un oggetto ContentValues per impostare i valori per ogni colonna della tabella
// (se un valore non è impostato, verrà automaticamente impostato a NULL)
ContentValues values = new ContentValues();
values.put(LogDescriptorOpenHelper.TIMESTAMP_COL, log.getTimestamp());
values.put(LogDescriptorOpenHelper.LATITUDE_COL, log.getLatitude());
values.put(LogDescriptorOpenHelper.LONGITUDE_COL, log.getLongitude());
values.put(LogDescriptorOpenHelper.TYPE_COL, log.getType());
```

```
values.put(LogDescriptorOpenHelper.DATA_COL, log.getData());

// Inserisci i valori configurati nella tabella desiderata.
// Il risultato sarà l'ID grezzo del nuovo record (-1 se si verifica un errore)
long insertId = database.insert(LogDescriptorOpenHelper.TABLE_NAME, null, values);

// Esegui una query al database per ottenere l'oggetto dell'ultimo record aggiunto.
// Utilizza il metodo query(...) specificando la condizione WHERE.
Cursor cursor = database.query(LogDescriptorOpenHelper.TABLE_NAME, allColumns,
LogDescriptorOpenHelper.ID_COL + " = " + insertId, null, null, null, null);


// Leggi i valori dal Cursor e crea l'oggetto.
cursor.moveToFirst();
LogDescriptor obj = cursorToLogDescriptor(cursor);
```

Delete

```
// Elimina tutti i record nella tabella specificata senza particolari condizioni WHERE.
public void deleteAllLogs() {
        database.delete(LogDescriptorOpenHelper.TABLE_NAME,null, null);
}

// Elimina tutti i record nella tabella specificata con una particolare condizione WHERE.
// In questo caso, verrà eliminato il record con un ID specifico.
public void deleteLog(LogDescriptor logDescr) {
        long id = logDescr.getId();
        database.delete(LogDescriptorOpenHelper.TABLE_NAME,LogDescriptorOpenHelper.ID_COL + "
= " + id, null);
}
```

Raw query

Si può eseguire query grezze (raw queries) utilizzando il linguaggio SQL e il metodo dedicato `rawQuery(...)`. In questo caso, la query viene costruita dallo sviluppatore come una stringa e passata alla funzione come parametro principale.

In questo esempio, la query restituisce il numero di record nella tabella:

```
int logCount = 0;
String query = "SELECT COUNT(*) FROM "+LogDescriptorOpenHelper.TABLE_NAME;
Cursor cursor = database.rawQuery(query, null);
cursor.moveToFirst();
// ...
cursor.close();
```

Room

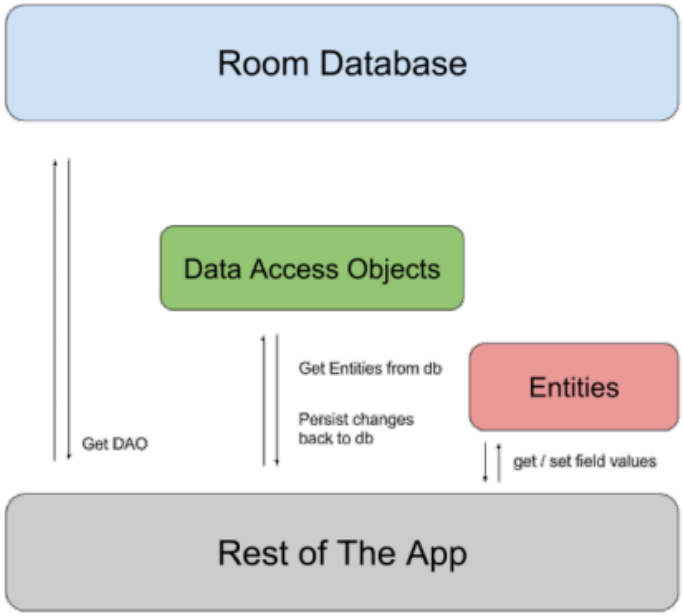
Ci sono 3 componenti principali in Room:

- Database, che contiene il gestore del database e funge da punto principale di accesso alla connessione sottostante per i dati persistiti e relazionali dell'applicazione. La classe annotata con `@Database` dovrebbe soddisfare le seguenti condizioni:
  - Essere una classe astratta che estende `RoomDatabase`
  - Includere nell'annotazione l'elenco delle entità associate al database
  - Contenere un metodo astratto senza argomenti che restituisce la classe annotata con `@Dao`
  - A runtime si può ottenere un'istanza del Database chiamando `Room.databaseBuilder()` o `Room.inMemoryDatabaseBuilder()`
- Entità, che rappresenta una tabella all'interno del database
- DAO, che contiene i metodi usati per accede al database

Room Architecture

L'appicazione utilizza il database Room per ottenere gli oggetti di accesso ai dati, o DAO (Data Access Objects), associati a quel database, successivamente, utilizza ciascun DAO per ottenere le entità dal database e salvare eventuali modifiche a tali entità nel suddetto.

Infine, l'applicazione utilizza un'entità per ottenere e impostare i valori corrispondenti alle colonne della tabella all'interno del database.



Import Room

```
dependencies {
        def room_version = "1.1.1"
        implementation "android.arch.persistence.room:runtime:$room_version"
        annotationProcessor "android.arch.persistence.room:compiler:$room_version" // use
kapt for Kotlin
        // optional - RxJava support for Room
        implementation "android.arch.persistence.room:rxjava2:$room_version"
        // optional - Guava support for Room, including Optional and ListenableFuture
        implementation "android.arch.persistence.room:guava:$room_version"
        // Test helpers
        testImplementation "android.arch.persistence.room:testing:$room_version"
}
```

Create a new Entity → User

```
@Entity
public class User {
        @PrimaryKey
        public int uid;
        @ColumnInfo(name = "first_name")
        public String firstName;
        @ColumnInfo(name = "last_name")
        public String lastName;
}
```

Create the DAO → UserDao

```
@Dao
public interface UserDao {
        @Query("SELECT * FROM user")
        List<User> getAll();

        @Query("SELECT * FROM user WHERE uid IN (:userIds)")
        List<User> loadAllByIds(int[] userIds);

        @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
        "last_name LIKE :last LIMIT 1")
        User findByName(String first, String last);

        @Insert
        void insertAll(User... users);
        @Delete
        void delete(User user);
}
```

Room App Database

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
        public abstract UserDao userDao();
}

AppDatabase db = Room.databaseBuilder(getApplicationContext(), AppDatabase.class, "database-
name").build();
```

Room & Primary Key

```
@Entity
public class User {
        @PrimaryKey
        public int id;
        public String firstName;
        public String lastName;
}
```

Per persistere un campo, Room deve avere accesso ad esso, si può rendere un campo pubblico, oppure si può fornire un getter e un setter per esso.

> Nota
> Se si utilizzano i metodi getter e setter, bisogna tener presente che devono seguire le convenzioni di JavaBeans in Room

```
@Entity(primaryKeys = {"firstName", "lastName"})
        public class User {
        public String firstName;
        public String lastName;
}
```

Room Table & Field Name

```
@Entity(tableName = "users")
public class User (
        // ...
)
```

Per impostazione predefinita, Room utilizza il nome della classe come nome della tabella del database. Se si desidera che la tabella abbia un nome diverso, può impostare la proprietà `tableName` dell'annotazione `@Entity` con il nome desiderato.

```
@Entity(tableName = "users")
public class User {
        @PrimaryKey
        public int id;
        @ColumnInfo(name = "first_name")
        public String firstName;
        @ColumnInfo(name = "last_name")
        public String lastName;
}
```

Ignore Fields

Per impostazione predefinita, Room crea una colonna per ogni campo definito nell'entità. Se un'entità ha campi che non si desidera far persistere, è possibile annotarli utilizzando `@Ignore`, così facendo Room ignorerà quei campi e non li tratterà come colonne nel database.

```
@Entity
public class User {
        @PrimaryKey
        public int id;
        public String firstName;
        public String lastName;
        @Ignore
        Bitmap picture;
}
```

Define relationships between objects

```
@Entity(foreignKeys = @ForeignKey(entity = User.class,
                                  parentColumns = "id",
                                  childColumns =
"user_id"))
public class Book {
        @PrimaryKey
        public int bookId;
        public String title;
        @ColumnInfo(name = "user_id")
        public int userId;
}
```

Ad esempio, se esiste un'altra entità chiamata "Book", si può definire la relazione con l'entità "User" utilizzando l'annotazione `@ForeignKey`, le chiavi esterne (Foreign keys) sono molto potenti, in quanto consentono di specificare cosa accade quando l'entità referenziata viene aggiornata. Ad esempio, si può indicare a SQLite di eliminare tutti i libri di un utente se l'istanza corrispondente di "User" viene eliminata includendo `onDelete = CASCADE` nell'annotazione `@ForeignKey`.

Nested Objects

A volte si vorrebbe esprimere un'entità o un semplice oggetto Java (plain old Java object, o POJO) come un insieme coeso nella logica del database, anche se l'oggetto contiene diversi campi, in queste situazioni si può utilizzare l'annotazione `@Embedded` per rappresentare un oggetto che si desidera decomporre nei suoi sotto-campi all'interno di una tabella.
Successivamente si può quindi interrogare i campi incorporati proprio come si farebbe per altre colonne individuali.

> Note
> La tabella che rappresenta un oggetto User contiene i i campi `id`, `firstName`, `street`, `state`, `city` e `post_code`

```
public class Address {
        public String street;
        public String state;
        public String city;
        @ColumnInfo(name = "post_code")
        public int postCode;
}

@Entity
public class User {
        @PrimaryKey
        public int id;
        public String firstName;
        @Embedded
        public Address address;
}
```

Room Notes

Se l'applicazione viene eseguita in un singolo processo, bisognerebbe seguire il pattern di progettazione singleton quando si istanza un oggetto AppDatabase.

> Nota
> Ogni sua istanza è piuttosto costosa e raramente si ha bisogno di accedere a più istanze all'interno di un singolo processo

Se l'applicazione viene eseguita in più processi bisogna includere `enableMultiInstanceInvalidation()` nell'invocazione del costruttore del database, se così facendo quando si ha un'istanza di AppDatabase in ciascun processo, si può invalidare il file del database condiviso in un processo e questa invalidazione si propaga automaticamente alle istanze di AppDatabase negli altri processi.

Room non supporta l'accesso al database nel thread principale a meno che tu non si abbia chiamato `allowMainThreadQueries()` nel costruttore, perché potrebbe bloccare l'interfaccia utente per un lungo periodo di tempo.
Le query asincrone, cioè le query che restituiscono istanze di LiveData o Flowable, sono esenti da questa regola poiché eseguono in modo asincrono la query su un thread in background quando necessario.

> Nota
> La combinazione perfetta è utilizzare Room con LiveData