Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

Data Management

Federico Montori
federico.montori2@unibo.it

Table of Contents

SQLite for Android

- General purpose solution
- Lightweight database based on SQL
- Standard SQL syntax

```
SELECT name FROM table WHERE name = “Federico”
```

- Android gives a standard interface to SQL tables of other apps
- For application tables no content providers are needed

Why a local database? Since you can’t assume connectivity

Single Source of Truth: SSOT refers to the concept where certain data has only one official source to be used by data consumers (i.e. humans and software) for the true current version of that data (more on that later).

SQLite for Android

- A DBMS to store information
  - Useful for structured informations
- Create a **DBHelper** that extends **SQLiteOpenHelper**
- Fill it with methods for managing the database
- Better to use constants like
  - TABLE_GRADES
  - COLUMN_NAME
  - …

This is an overview of how it is done underneath for simple projects.

For a structured approach, use **Room**

SQLite for Android

Our database will look like this:

grade table:

- id: integer, primary key, auto increment
- name: text, not null
- class: text, not null
- grade: integer, not null

Remember that database operations are **potentially blocking**!
Remember to **always use threads** for them.

SQLite for Android

A best practice is to define constants that identify column titles…

Interface “BaseColumns” provides the field _ID required by CursorAdapter (see later)

```
companion object StudentContract {
   // Table contents are grouped together in an anonymous object.
   object StudentEntry : BaseColumns {
      const val TABLE_NAME = "students"
      const val COLUMN_NAME = "name"
      const val COLUMN_CLASS = "class"
      const val COLUMN_GRADE = "grade"
   }
}
```

SQLite for Android

Create a class that extends SQLiteOpenHelper and implement its onCreate method...

```
val dbHelper = DbHelper(context)
```

```
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 }
}
```

SQLite for Android

Insert information into a database

```
val db = dbHelper.writableDatabase

val values = ContentValues().apply {
        put(StudentEntry.COLUMN_NAME, "Mario Rossi")
        put(StudentEntry.COLUMN_CLASS, "LAM")
        put(StudentEntry.COLUMN_GRADE, "30")
}

// Insert the new row, returning the primary key value of the new row
// Params are: the table, what to do in case of empty content values, the values to insert
val newRowId = db?.insert(StudentEntry.TABLE_NAME, null, values)
```

SQLite for Android

Update information in a database

```
val db = dbHelper.writableDatabase

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

val selection = "${StudentEntry.COLUMN_NAME} LIKE ?" // arg injection (replacing the ?)
val selectionArgs = arrayOf("Mario Rossi")
val count = db.update(
    StudentEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs)
```

SQLite for Android

Delete information from a database

```
val db = dbHelper.readableDatabase

// Define 'where' part of a query.
val selection = "${StudentEntry.COLUMN_NAME} LIKE ?"
// Specify arguments in placeholder order.
val selectionArgs = arrayOf("Mario Rossi")
// Issue SQL statement.
val deletedRows = db.delete(StudentEntry.TABLE_NAME, selection, selectionArgs)
```

SQLite for Android

Query information from a database

```
val db = dbHelper.readableDatabase

// Define a projection: the SELECT part of a query
val projection = arrayOf(BaseColumns._ID, StudentEntry.COLUMN_NAME)

val cursor = db.query(                                  // Returns a Cursor
        StudentEntry.TABLE_NAME,                        // The table to query
        projection,                                     // The array of columns to return (pass null to get all)
        "${StudentEntry.COLUMN_GRADE} = ?",             // The columns for the WHERE clause
        arrayOf("30"),                                  // The values for the WHERE clause (injected args)
        null,                                           // GROUP BY
        null,                                           // FILTER BY
        null                                            // SORT
)
```

SQLite for Android

Parse a Cursor (a pointer to the obtained columns that starts from index -1)

```
val items = mutableListOf<String>()
with(cursor) {
   while (moveToNext()) {
      val item = getString(getColumnIndexOrThrow(StudentEntry.COLUMN_NAME))
      items.add(item)
   }
}
cursor.close()
```

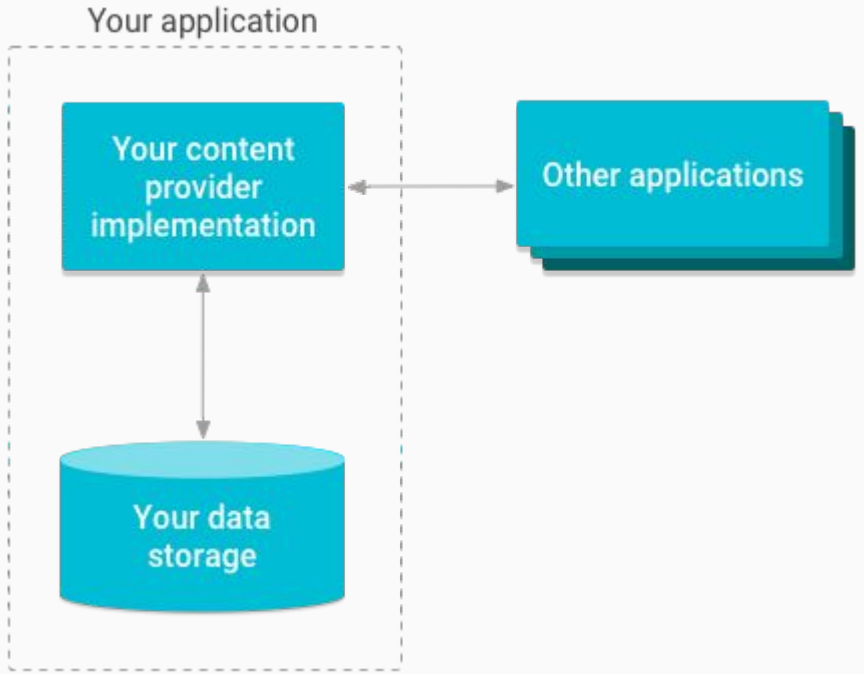
You should leave the db connection open for as long as you need to access it

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

Content Providers

A system to access shared data
Similar to a REST web service
For each Content Provider, one or more URIs are assigned in the form:

```
content://<authority>/path
```

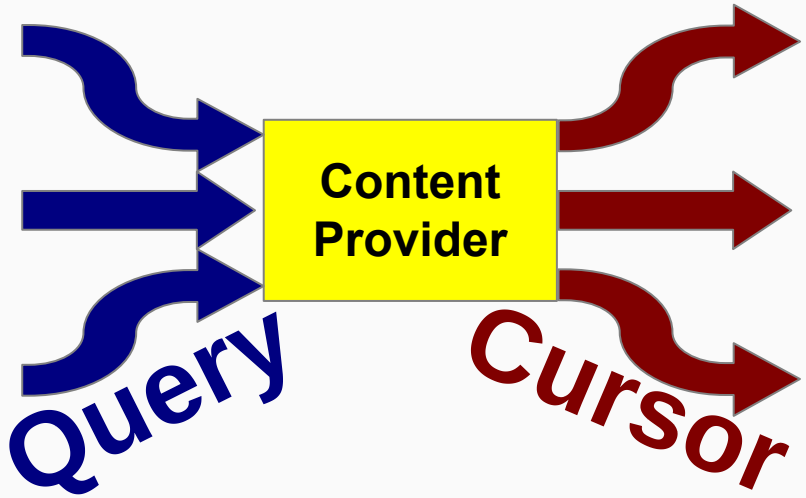


Be aware that some ContentProviders may request permissions

A Content Provider is seen by other applications as a DB interface that they can query.

Content Providers

- You need to get the URI
  - Usually this is declared as public inside the content provider class
  - URI = Table in the provider (authority = DB, path = table in the DB)
- Make a query, maybe adding some where clauses
  - You'll get a Cursor after that
  - Navigate the Cursor



Content Providers

Class	Description
AlarmClock	To interact with the alarm
BlockedNumberContract	To get blocked numbers
Browser	To perform commands on the browser
CalendarContract	To handle calendar information
CallLog	Log of past calls
ContactsContract	Get and add contacts
DocumentsContract	Interact with documents
DocumentsProvider	Interact with documents
MediaStore	Access Video, Pictures, Audio and more
Settings	Inquiry system settings

As you can see, Content Providers allow the access to Public Files in the **Scoped Storage**
Find them all at https://developer.android.com/reference/android/provider/package-summary.html

Content Providers

Example: **Contacts** are exposed via a Content Provider, for which we need permission:

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

```
// Does a query against the remote table and returns a Cursor object, this query is a “select all”
cursor = contentResolver.query(
        ContactsContract.Contacts.CONTENT_URI,   // The content URI of the words table
        null,                                    // The columns to return for each row [SELECT]
        null,                                    // Either null or the clause [WHERE]
        null,                                    // Either empty or the selection args
        null                                     // The sort order for the returned rows
)

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

Content Providers

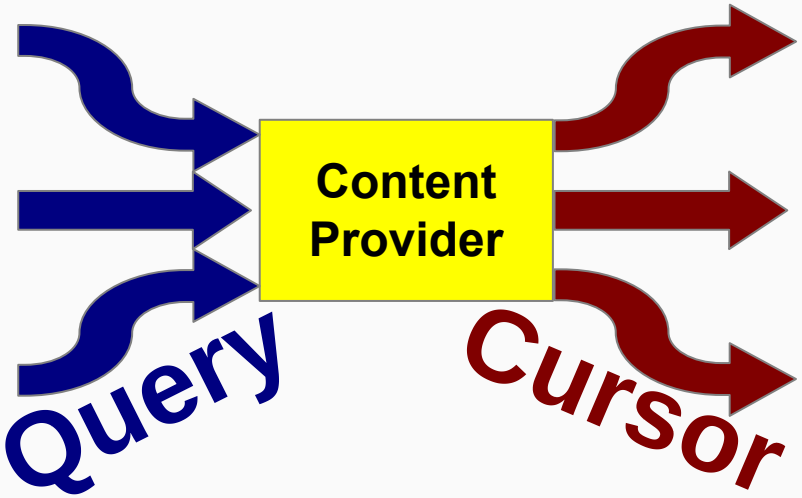
To create a Content Provider: create a class that extends android.content.ContentProvider and pick the URIs for your resources

```
class ExampleProvider : ContentProvider() {
        private val sUriMatcher = UriMatcher(UriMatcher.NO_MATCH).apply {
                addURI("com.example.app.provider", "table3", 1)
                addURI("com.example.app.provider", "table3/#", 2)
        }

        override fun query(uri: Uri?, projection: Array<out String>?, selection: String?,
        selectionArgs: Array<out String>?, sortOrder: String?): Cursor? {

                when (sUriMatcher.match(uri)) {
                        1 -> { … }
                        2 -> { … }
```

Content Providers

Register the ContentProvider in the **manifest** using the **<provider>** tag and:

- android:authorities (unique name of the provider)
- android:name (Class that implements it)
- various permissions...



Content Providers

You can easily share files using **FileProvider** (a special Content Provider)

```
<provider
        android:name="androidx.core.content.FileProvider"
        android:authorities="com.example.myapp.fileprovider"
        android:grantUriPermissions="true"
        android:exported="false">
        <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/filepaths" />
</provider>
```

Create res/xml/filepaths.xml

```
<paths>
        <files-path path="images/" name="myimages" />
</paths>
```

Now other apps can access your file using URI like

```
content://com.example.myapp.fileprovider/myimages
/default_image.jpg
```

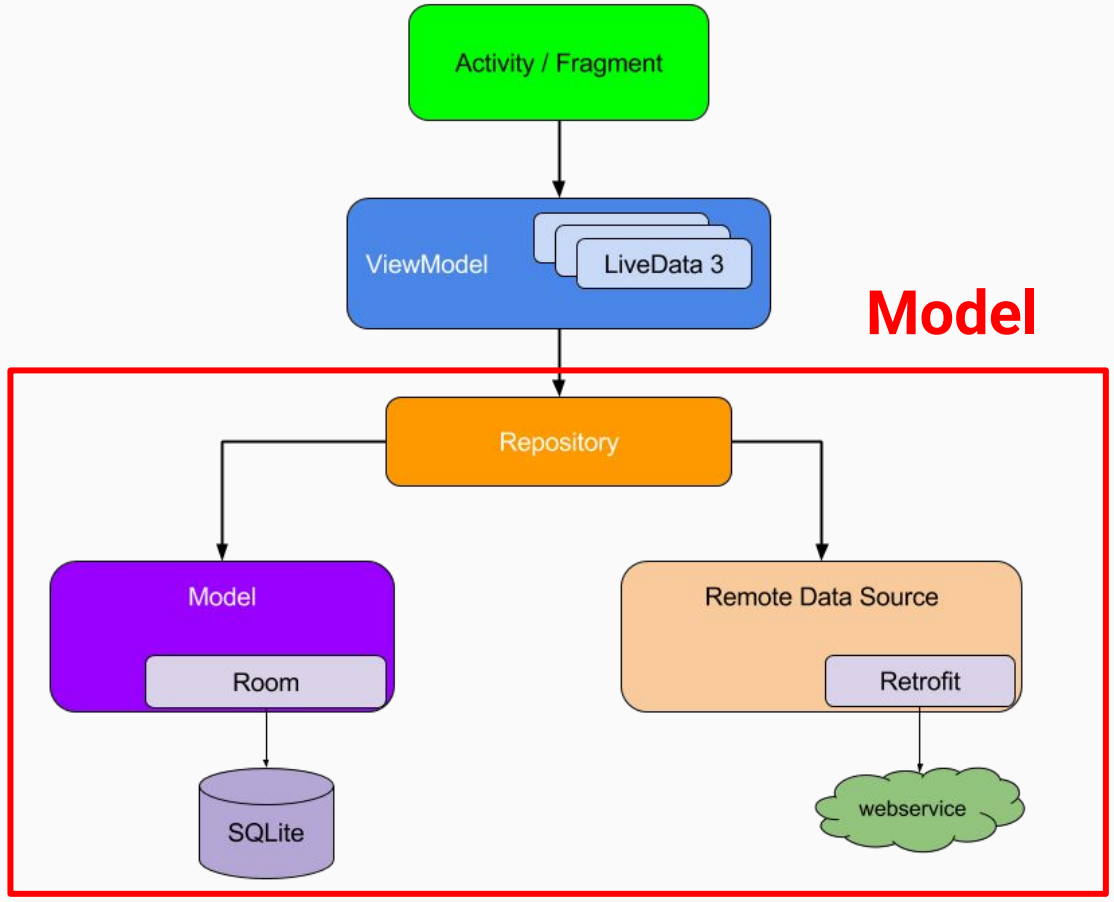
Room

Remember MVVM?

Let us talk about the **Model**

The Model is whatever is persistent in out macro system, which includes persistent data in the local database and remote data.

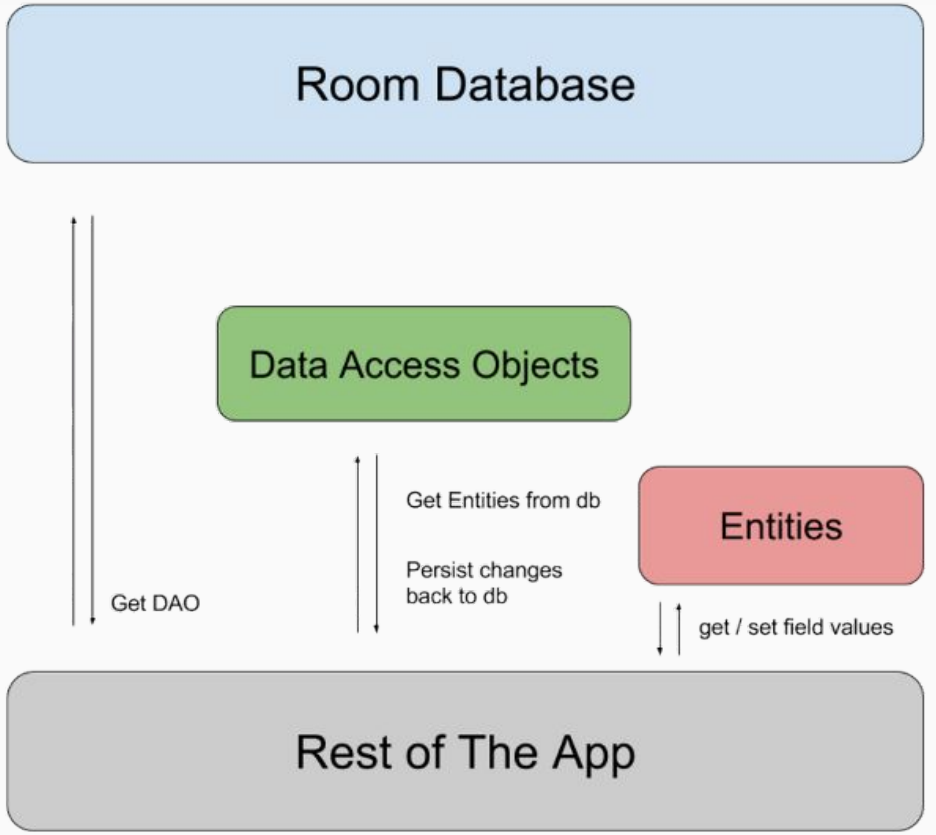
For the local database, the framework recommended by Android is **Room**



Room

Room provides an abstraction layer over SQLite. You should always use Room if your project is sufficiently complex…

- Database
  - Contains the database holder
  - Main access point
- Data Access Objects (DAOs)
  - Interface with methods to access the database
- Entities
  - Database tables



Room

Room is a generative library that generates code according to annotations.

Most probably, in order to use it properly, you will need to use KAPT (Kotlin Annotation Processing Tool) to be able to do it.

```
plugins {                                   // Module-level configuration file
   kotlin("kapt")
}

dependencies {
   implementation("androidx.room:room-ktx:2.6.1")
   kapt("androidx.room:room-compiler:2.6.1")
}
```

Room

It has to be an abstract class extending 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
}
```

Why abstract?

Room implements the conversion from the database interactions to your app classes automatically. Just tell what you want in and out (DAOs are in fact abstract).

- Room in fact sticks to the concept of **marshaling** and **unmarshaling** (like **serialization**):
  - *“transforming the memory representation of an object into a data format suitable for storage or transmission and vice versa”*

Room

For each Entity, Room creates a database Table

Each field references a column, except for those marked with @**Ignore**

```
@Entity(tableName = "my_entity")       /* A table called  my_entity with two columns: index and field1*/
class Entity1(
   @PrimaryKey
   @NonNull
   @ColumnInfo(name = "index")
   var id: String,

   var field1: String,
   @Ignore
   var temp: String
)
```

Tables and columns can have custom names by using tableName and ColumnInfo

Room

Entities fields needs to public - you have to provide getters and setters

Each entity needs at least one @PrimaryKey

- Primary keys can be defined with more than one field

```
@Entity(primaryKeys = {"firstName", "lastName"})
```

- The autoGenerate property automatically assigns IDs

```
@PrimaryKey(autoGenerate = true)
var id: String
```

- Speed up queries with indices

```
@Entity(indices = {@Index("name"), @Index(value = {”first_name", ”last_name"})})
```

Room

- Defining uniqueness:

```
@Entity(indices = {@Index(value = {"first_name", "last_name"}, unique = true)})
```

- Defining Relationships:

```
@Entity(foreignKeys = @ForeignKey(entity = User::class, parentColumns = "id", childColumns = "user_id"))
```

- Nested Objects

```
data class Material (val name : String, val weight: String)
@Entity
class myEntity (
        ...
        @Embedded
        var objectMaterial: Material,
)
```

Room

- Defining relations:

```
class Entity1AndEntity2 (
   @Embedded
   var e1: Entity1,
   @Relation(
       parentColumn = "id",
       entityColumn = "user_id"
   )
   var e2: Entity2
)
```

If it's one-to-many then you need to put a list of Entity2 here instead of only one.

Same as ForeignKey, but lets you make atomic queries (will see how)

If many-to-many relationship, then specify two one-to-many relations

Room

DAOs (Data Access Objects) embed the calls to the database.

- They are abstract classes or interfaces
  - Just specify what you want in and out and the query
  - Room generated an implementation at compile time, automatically handling the marshaling between entities and cursors.

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

// @QueryType can be: @Insert, @Update, @Delete, @Query
```

Room

- A DAO can be either an interface or an abstract class
  - If Abstract class, it takes the DB as input in the constructor.
- DO NOT perform DAO operations in the main thread,
  - this is btw forbidden unless you specify it
  - Typically use Worker Threads (coroutines or a thread pool)
- DO NOT implement it

Room

@Insert	@**Insert**(onConflict = OnConflictStrategy.REPLACE) **fun** insertUser(user: User): **Unit** @**Insert** **fun** insertBothUsers(user1: User, user2: User) @**Insert** **fun** insertUsersAndFriends(user: User, friends: List<User>)
@Update	@**Update** **fun** updateUser(user: User): **Unit**
@Delete	@**Delete** **fun** deleteUser(user: User): **Unit**

Room

@Query

```
@Query("SELECT * FROM user")
fun loadAllUsers(): List<User>
```

@Query + parameters

```
@Query("SELECT * FROM user WHERE age > :minAge")
fun loadAllUsersOlderThan(minAge: Int): List<User>
```

@Query + LiveData

```
@Query("SELECT * FROM user")
fun loadAllUsersObservable(): LiveData<List<User>>
```

The Room persistence library supports observable queries, which return LiveData objects.

- Observable queries are written as part of a DAO.
- Do not explicitly run them into a separate Thread (it is done by default).
- Changes in the Database are immediately notified to the LiveData.

Room

- Query on multiple tables

```
@Query("SELECT * FROM book " + "INNER JOIN loan ON loan.book_id = book.id " +
        "INNER JOIN user ON user.id = loan.user_id " + "WHERE user.name LIKE :userName")
        fun findBooksBorrowedByNameSync(userName: String): List<Book>
}
```

- Query a relation

```
@Transaction
@Query("SELECT * FROM Entity1")
fun getRelations(): List<Entity1AndEntity2>
```

Filters only the object of **Entity1** that have a respective on **Entity2**. The

@**Transaction** ensures that this is atomic as it would be 2 queries.

Room

- Updating APP's features may require updating the database
  - You add a UI field and need to add a DB field
  - You change the type of a field
  - You don't need anymore a field
- Room handles it providing the Migration environment
  - Remember:

```
@Database(entities = [Entity1::class, Entity2::class], version = 1, exportSchema = false)
abstract class myDatabase : RoomDatabase() {
    …
}
```

Room

Each Migration class defines a startVersion and endVersion

- At runtime, Room runs each migrate method in order

```
Room.databaseBuilder(context.applicationContext, myDatabase::class.java, "database-name")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build();

companion object {
        val MIGRATION_1_2 = object: Migration(1, 2) {
                override fun migrate(database: SupportSQLiteDatabase) {
                        database.execSQL("CREATE TABLE ‘Fruit’ (‘id’ INTEGER, ‘name’ TEXT, PRIMARY KEY(‘id’))")
        }

        val MIGRATION_2_3 = …
}
```

HTTP

HTTP (HyperText Tranfer Protocol): Network protocol for exchange/transfer data (hypertext).

Request/Response Communication Model

- MAIN METHODS:
  - HEAD
  - GET
  - POST
  - PUT
  - DELETE
  - TRACE
  - CONNECT

HTTP

Two implementations of HTTP Clients for Android historically:

- HTTPClient → Complete extendable HTTP Client suitable for web browser (not supported starting from 6.0)
- HTTPUrlConnection → Light-weight implementation, suitable for client-server networking applications (recommended by Google, starting from 2.3)

In both cases, HTTP connections must be managed on a separate thread, e.g. using Thread Pool (not the UI thread!).

HTTPUrlConnection

HTTPUrlConnection → HTTP component to send and receive streaming data over the web.

1. Obtain a new HttpURLConnection by calling the URL.openConnection()

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

2. Prepare the request, set the options:

- session cookies
- credentials
- preferred content type
  (e.g. *setRequestProperty("Content-Type", "text/plain")*)

HTTPUrlConnection

HTTPUrlConnection → HTTP component to send and receive streaming data over the web.

3. For POST commands, invoke setDoOutput(true). Transmit data by writing to the stream returned by getOutputStream().

```
urlConnection.doOutput = true
urlConnection.requestMethod = "POST"
urlConnection.setChunkedStreamingMode(0) // use setFixedLengthStreamingMode if size is known
val out: OutputStream = BufferedOutputStream(urlConnection.outputStream)
out.write("YourPostInput".toByteArray())
```

HTTPUrlConnection

HTTPUrlConnection → HTTP component to send and receive streaming data over the web.

4. Read the response (data+header). The response body may be read from the stream returned by getInputStream().

```
val inStream: InputStream = BufferedInputStream(urlConnection.inputStream);
// Do what you want with the InputStream
```

5. Close the session when ending reading the stream through disconnect().

```
urlConnection.disconnect()
```

HTTPUrlConnection

HTTPUrlConnection → HTTP component to send and receive streaming data over the web.

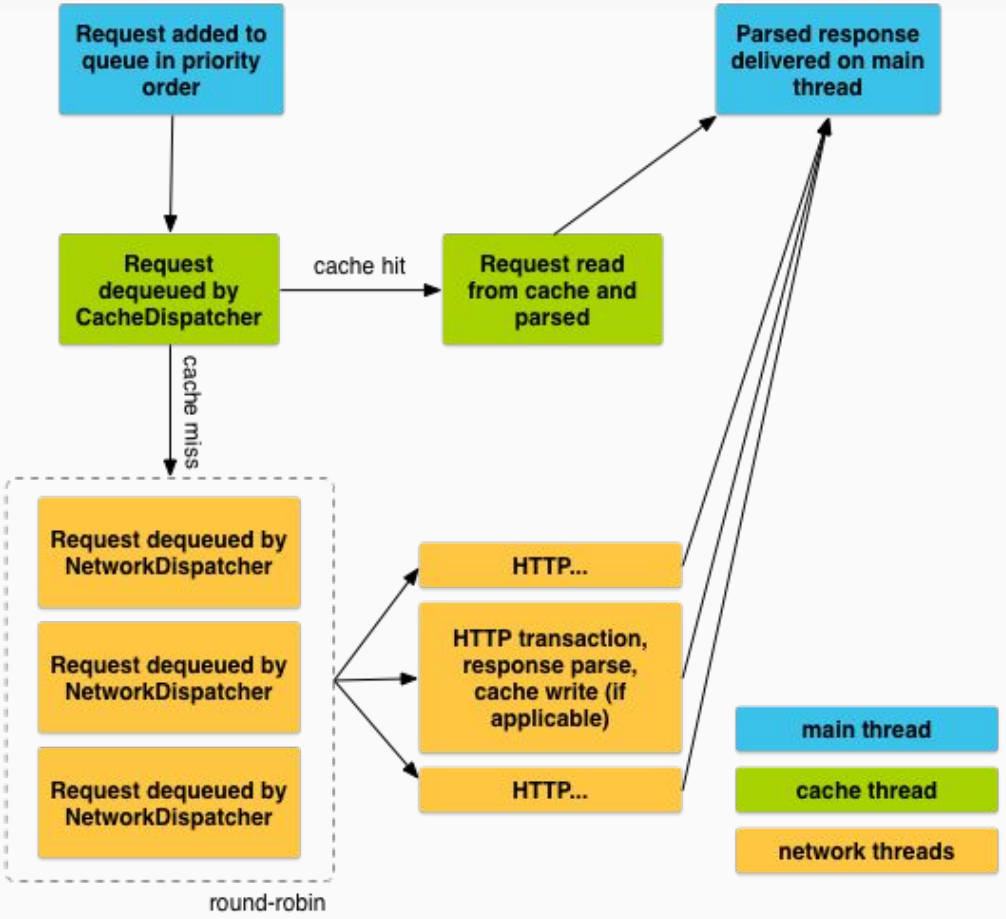
- use getErrorStream() in case of errors
  - use the HttpsURLConnection in case of HTTPS URLs
- Can override the default HostnameVerifier
- Can override the SSLSocketFactory
- Can define a custom X509TrustManager to verify certificate chains
- use HttpResponseCache if you need to cache replies in order not to waste resources

Volley

Volley → HTTP library with caching mechanism and async calls

- Volley is an HTTP library
- Supports scheduling of network requests
- Can have concurrent connections and handles priorities
- Caching mechanism
- Can cancel requests
- Heavily customizable
- Request ordering
- Not suited for long download operations (keeps in memory all streaming content)

Volley



- Make a request and add it.
- Then it moves through the pipeline
- Cache triages it
- If not found it’s transferred to a network thread
- Response is sent back

Add the dependency in Gradle:

```
implementation("com.android.volley:volley:1.2.1")
```

Volley

Volley → HTTP library with caching mechanism and async calls
Make a request (verbose syntax):

```
val queue: RequestQueue = Volley.newRequestQueue(this)
val stringRequest: StringRequest = StringRequest(
        Request.Method.GET,
        MY_URL,
        Response.Listener<String>() {
                response -> // Do stuff to handle the response
        },
        Response.ErrorListener() {
                error -> // Do stuff in case of error
        }
)
queue.add(stringRequest)
```

Volley

Volley → HTTP library with caching mechanism and async calls

Custom headers can be added by overriding the getHeaders method like so:

```
{ /* It is important to use the keyword object on StringRequest to be able to override this */
      override fun getHeaders(): MutableMap<String, String> {
            val headers = HashMap<String, String>()
            headers["Authorization"] = "Basic <<YOUR BASE64 USER:PASS>>"
            return headers
      } }
```

Retrofit

Retrofit → HTTP library for automatic Marshaling/Unmarshalling content

- Retrofit is a type-safe HTTP client for Java (yet another one)
  - full doc https://square.github.io/retrofit/
- It translates automatically XML and JSON objects into POJO (Plain-Old Java Objects)
- It is very similar to Room, indeed it can use the same Entities
  - We can say it is its dual for remote resources
- Here we will just see some basic functionalities, you can then explore further...
- Import the necessary dependencies (for JSON in this example):

```
implementation("com.squareup.retrofit2:retrofit:2.3.0")
implementation("com.squareup.retrofit2:converter-gson:2.3.0")
```

Retrofit

Retrofit → HTTP library for automatic Marshaling/Unmarshalling content

Just design a normal data class
Use the **SerializedName** to specify what name it has in the JSON/XML data frame.

```
data class RetroPhoto(
    @SerializedName("albumId")
    var AlbumId: Int,

    @SerializedName("id")
    var id: Int
)
```

Ideally I want to convert remote resources into RetroPhoto objects. In our case I am using https://jsonplaceholder.typicode.com/photos

Retrofit

Retrofit → HTTP library for automatic Marshaling/Unmarshalling content

Then set up the Retrofit client → this one translates JSON

```
class RetrofitClientInstance {          /* Singleton with lazy loading */
   companion object {
      private lateinit var retrofit: Retrofit
      const val BASE_URL = "https://jsonplaceholder.typicode.com"
      fun getRetrofitInstance(): Retrofit {
         if (!this::retrofit.isInitialized) {     /* Kotlin 2.1 improvement */
            retrofit = Retrofit.Builder().baseUrl(BASE_URL)
                  .addConverterFactory(GsonConverterFactory.create()).build()
         }
         return retrofit
      } }}
```

Retrofit

Retrofit → HTTP library for automatic Marshaling/Unmarshalling content

- Then, just like with the DAOs, create an interface for each remote call
  - Just like for the DAOs, they will be automatically implemented for you…

```
interface GetDataService {
   @GET("/photos")
   fun getAllPhotos(): Call<List<RetroPhoto>>
}
```

This will return a **Call** object: an instance of an interaction with the remote server. The Call needs to be effectively issued (asynchronously maybe) in order to be effective...

Retrofit

Retrofit → HTTP library for automatic Marshaling/Unmarshalling content

- Then, enqueue the call

```
val service = RetrofitClientInstance.getRetrofitInstance().create(GetDataService::class.java)
val call: Call<List<Todo>> = service.getAllPhotos()
call.enqueue(object : Callback<List<RetroPhoto>> {
        override fun onResponse
                ( call: Call<List<RetroPhoto>>, response: Response<List<RetroPhoto>> ) {
                        val myList = response.body()          /* Do your stuff with the result */
                }
        override fun onFailure(call: Call<List<RetroPhoto>>, t: Throwable?) {
                Log.e("RETROFIT", "something went wrong... but life goes on")
        }
})
```

Firebase

Firebase is a Google app development platform that gives you an easy-to use and reactive backend for your app.

- Realtime Database:
  - The original database, a simple JSON tree, supporting easy queries and an easier startup.
  - Made for performance, low latency, few data
- Cloud Firestore:
  - JSON-like documents organized into collections, supporting more advanced queries and a lot more scalability.

IN BOTH CASES YOU CAN PERFORM QUERIES AND OBSERVE THEM AS THE DATABASE IS **REACTIVE**

Firebase

Firebase

AHTWP4Demo

Vai alla documentazione

Panoramica del progetto

Creazione

- Authentication
- App Check
- Firestore Database
- Realtime Database
- Extensions
- Storage
- Hosting
- Functions
- Machine Learning

Rilascio e monitoraggio

Crashlytics, Performance, Test La...

Analisi

- Dashboard
- Realtime
- Events
- Conversions
- Audiences

Spark

Nessun costo 0 $/mese

Esegui l'upgrad

Realtime Database

Dati | Regole | Backup | Utilizzo

Proteggi le tue risorse di Realtime Database da comportamenti illeciti, come fatturazione fraudolenta o phishing

Configura App Check

https://ahtwp4demo-default-rtdb.firebaseio.com

```
https://ahtwp4demo-default-rtdb.firebaseio.com/
Temperature
  -Mlt2wnDJn1T5-vxMw9r
    timestamp: "2021-10-13T11:50:48.507454"
    value: 26.525516422207232
  -Mlt2znBGEpL4h63-B8B
    timestamp: "2021-10-13T11:50:53.507569"
    value: 23.36199306879651
  -Mlt2ztq2viotGJwdjXB
  -Mlt323hg0BoIEEQiVUd
  -Mlt32ARUsHhqGj8Xw0B
  -Mlt32Gu0M-7kwnIc3l0
  -Mlt35PyX0lwEmmYdswW
  -Mlt35WAIVNC_cE7iKB3
  -Mlt35bP0M3tkTn4bzpG
```

Località del database: (us-central1) - Stati Uniti

Firebase

Add the dependency: `implementation("com.google.firebase:firebase-database-ktx:20.3.1")`

Observe the query from your ViewModel (the result get passed to a LiveData so we have two nested listeners).

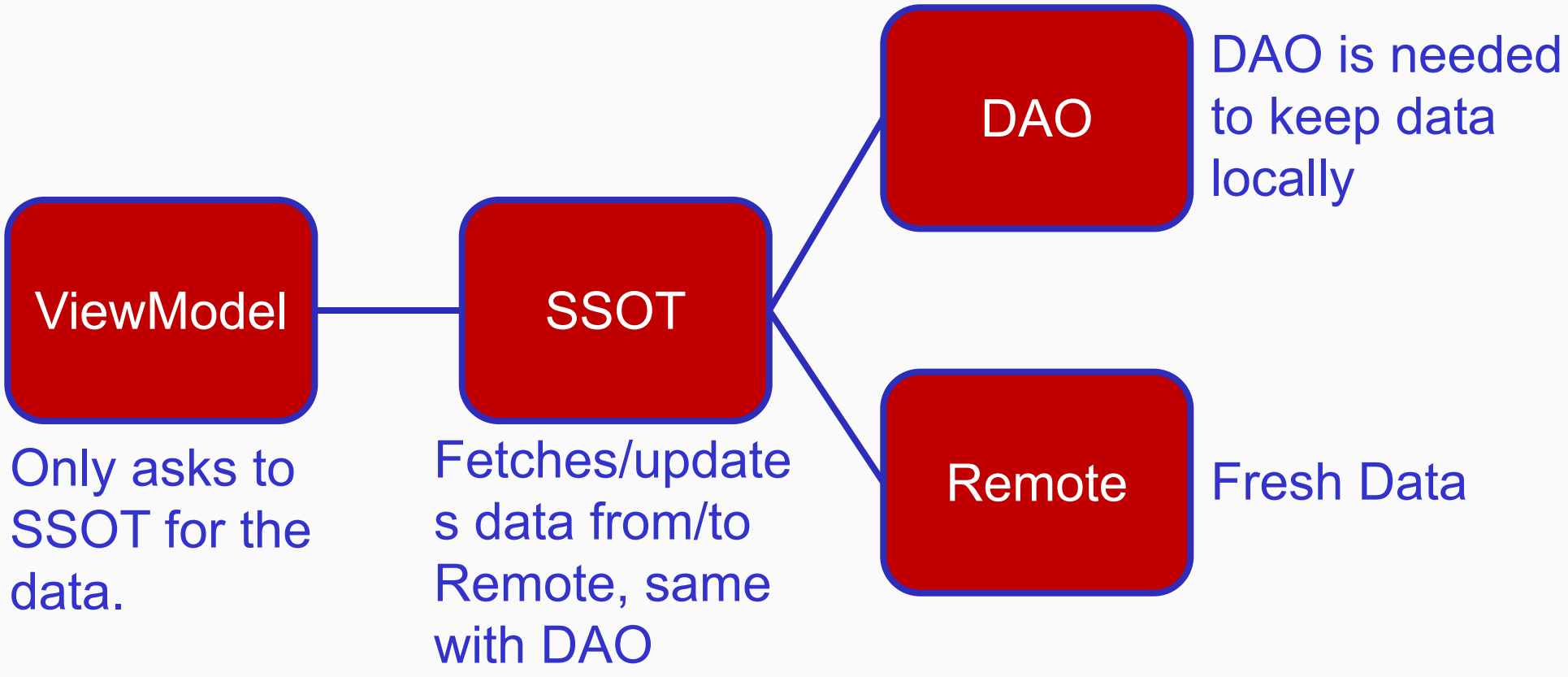
```
val mDatabase: FirebaseDatabase  =
FirebaseDatabase.getInstance("https://wp4demo-default-rtdb.firebaseio.com")
/* Let's assume we have a simple data class  TemperatureDataPoint */
val tempPoint = MutableLiveData<TemperatureDataPoint>()
mDatabase.getReference("Temperature").addChildEventListener(object: ChildEventListener {
      override fun onChildAdded(snapshot: DataSnapshot, previousChildName: String?) {
            tempPoint.postValue(snapshot.getValue(TemperatureDataPoint::class.java))
      }
      /* TODO implement other members ...  */
})
```

SSOT

SSOT model (**Single Source of Truth**) ensures that the request for the data is **ALWAYS** made against a single source
→ With Room and LiveData, your single source may be the Room Database

- IDEA: when requesting remote data, ALWAYS save it to your database and provide the LiveData returned by the database, so the ViewModel does not know who updated it.
- This is why you need an intermediate Repository class that handles all the different calls to data sources.

SSOT


ViewModel
SSOT
DAO
Remote
DAO is needed to keep data locally
Fresh Data
Only asks to SSOT for the data.
Fetches/updates data from/to Remote, same with DAO

SSOT

SSOT model →Let's get back to the retrofit call

When you ask for all the photos from the ViewModel:

```
val service = RetrofitClientInstance.getRetrofitInstance().create(GetDataService::class.java)
val call: Call<List<Todo>> = service.getAllPhotos()
call.enqueue(object : Callback<List<RetroPhoto>> {
        override fun onResponse
                ( call: Call<List<RetroPhoto>>, response: Response<List<RetroPhoto>> ) {
                        val myList = response.body()          /* INSERT THIS LIST INTO YOUR LOCAL DB! */
                }
        override fun onFailure(call: Call<List<RetroPhoto>>, t: Throwable?) {
                Log.e("RETROFIT", "something went wrong... but life goes on")
        }
})
```

Questions?

federico.montori2@unibo.it