Persisting Your Data with the Room Persistence Solution

Omri Erez

SOFTWARE ENGINEER

@innovationMaze | https://www.linkedin.com/in/omrierez/

Android Applications

- Handle quite a lot of data

- Usually using SQLite database or an equivalent solution

- Involves quite an implementation effort and mostly boring details

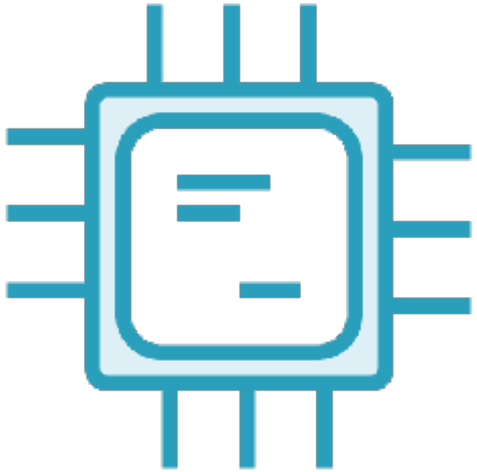
Advantages of Persistence

- If device has no network access the data will be available

- Faster loading of data

- Can save a huge amount of network data if the data is static

Different Types of Caches



Memory Cache

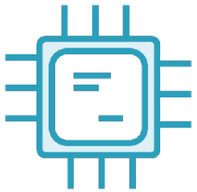
Disk Cache

Comparison

Memory Cache	Disk Cache
More expensive	Less expensive
Uses RAM	Uses local storage or a Database
Limited to RAM available	Limited to storage size
Faster access	Slower access
Used for objects which are often used	Used for the majority of the app's data

Memory Cache Example

Our app is using Glide for image loading

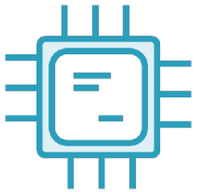


Saves last shown images in a memory cache for fast loading

Persist all loaded photos on disk to prevent undesired network activity

Disk Cache Example

Our app is loading data from an API



It saves the last fetched data on a LiveData object in memory

It persists the data on the device's local storage

Write Data

```
private void writeDataToInternalStorage(JSONArray data) {
    FileOutputStream fos = null;
    try {
        fos = mAppContext.openFileOutput(DATA_FILE_NAME,
Context.MODE_PRIVATE);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    try {
        fos.write(data.toString().getBytes());
        fos.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Read Data

```
private JSONArray readDataFromStorage() throws JSONException {
    FileInputStream fis = null;
    try {
        fis = mAppContext.openFileInput(DATA_FILE_NAME);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    InputStreamReader isr = new InputStreamReader(fis);
    BufferedReader bufferedReader = new BufferedReader(isr);
    StringBuilder sb = new StringBuilder();
    String line;
    try {
        while ((line = bufferedReader.readLine()) != null) {
            sb.append(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return new JSONArray(sb.toString());
}
```

Demo

Crypto Boom App

- Create a new module
- Move data related classes into it
- Add abstraction layer:
  - Create data sources
  - Create a repository

- Same functionality

- Separation of the data layer

- Short and clear classes

- Hide implementation details

Traditional SQLite Implementation

Create a Database

```
private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;

private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
            FeedEntry._ID + " INTEGER PRIMARY KEY," +
            FeedEntry.COLUMN_NAME_TITLE + " TEXT," +
            FeedEntry.COLUMN_NAME_SUBTITLE + " TEXT)";
```

Create a SQL Helper

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";
    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
       db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

In Addition

Mapping between cursors and Java objects

Custom ContentProviders

Migration if data changes

- Low level API’s which are time consuming to implement

- No compile time verification of raw SQLite queries

- Can cause problems and runtime errors after the data is changed

- Fairly a high amount of boilerplate “boring” code

- Implement over and over again the same functionalities

Solution
android.arch.persistence.room

- Lite persistence library

- Uses powerful annotations to define our database scheme and objects

- Uses compile time verification to show syntax errors

- Write less code for core functionality

- Entity centric approach

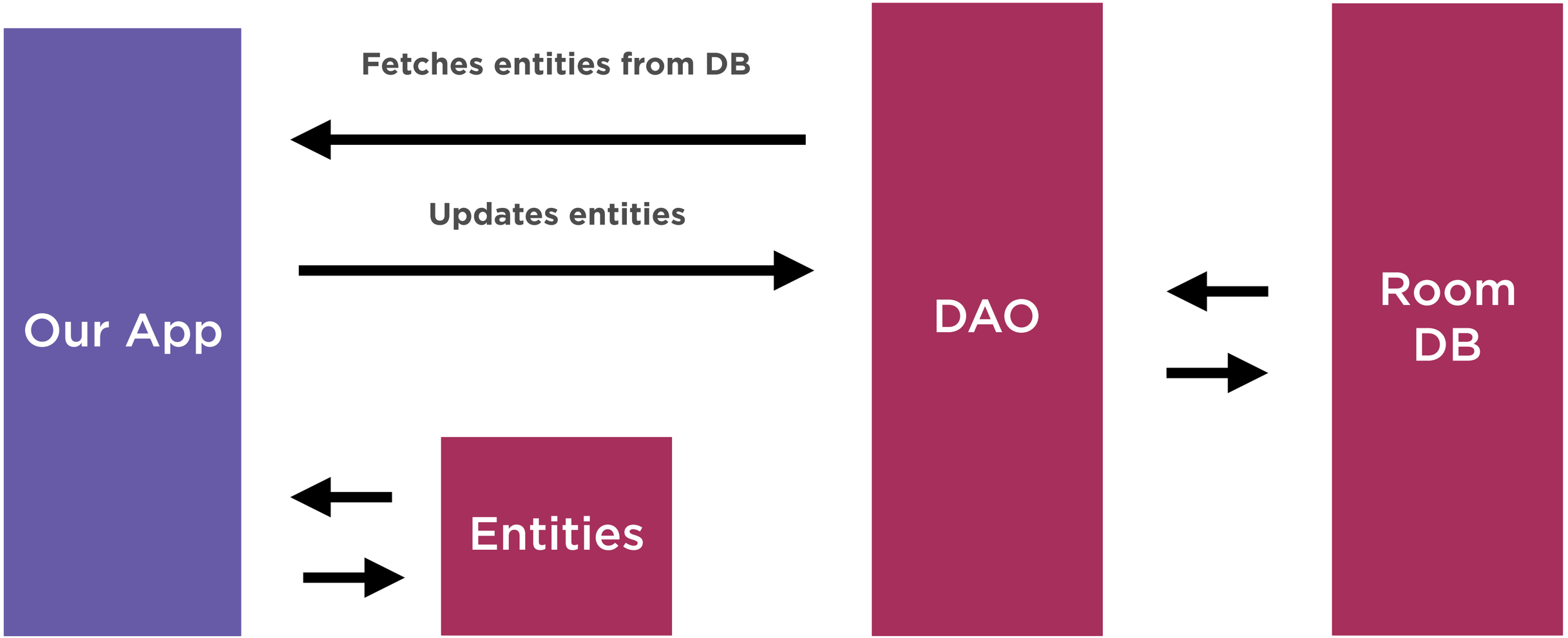
- No low level components like Cursors

- Strict access to DB from the UI thread

- Supports async queries with LiveData and RxJava

- Easy database migration in case of schema changes

Room Components



Entities

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

- @Entity annotation
- Primary key
- We can ignore fields from being stored in the DB

```
@Entity(tableName=“players",primaryKeys = {"firstName", "lastName"})
public class Player {
    public int id;

    @ColumnInfo(name = "f_n")
    public String firstName;
    public String lastName;
    @Ignore
    Bitmap picture;
}
```

- Define table name
- Define columns names
- Combine more than one field as a primary key

```
@Entity(indices = {@Index("first_name"),
        @Index(value = {"first_name", "last_name"})}
        ,foreignKeys = @ForeignKey(entity = CryptoCoinEntity.class,
        parentColumns = "id",
        childColumns = “player_id"))
public class Player {
    public int player_id;
    public String firstName;
    public String lastName;
    Bitmap picture;
}
```

- Define indices from one field or more
- Define relationships between entities
- More annotations available

Demo

Crypto Boom App

- Define our entity:
  - Define table
  - Define indices
  - Define column names

Data Access Objects

- This is how we access our database

- Uses simple annotations

- Here we define the functionality of our database

```java
@Dao
public interface CoinDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insertCoins(CryptoCoinEntity... coins);

    @Update
    void updateCoins(CryptoCoinEntity... coins);

    @Delete
    void deleteCoin(CryptoCoinEntity coin);
}
```

- Define a Dao object using the @Dao annotation
- Support insert, update and delete
- Define a conflict strategy

```java
@Dao
public interface CoinDao {

    @Query("SELECT * FROM coins")
    List<CryptoCoinEntity> loadCoins();

    @Query("SELECT * FROM coins LIMIT :limit")
    List<CryptoCoinEntity> loadCoins(int limit);

    @Query("SELECTT * FROM coins  WHERE " +
            "CAST(market_cap_usd as decimel) > :marketCap")
    List<CryptoCoinEntity> loadCoinsAboveMarketCap(long marketCap);
```

- Define raw queries using @Query annotation
- Let us pass parameters to queries
- Runs real time syntax checks

Database Definition

```
@Database(entities = {CryptoCoinEntity.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    static final String DATABASE_NAME = "market_data";
    private static AppDatabase INSTANCE;
    public abstract CoinDao coinDao();
    public static AppDatabase getDatabase(Context context) {
        if (INSTANCE == null) {
            INSTANCE= Room.databaseBuilder(context.getApplicationContext(),
                    AppDatabase.class, DATABASE_NAME).build();
        }
        return INSTANCE;
    }
}
```

- Define Database using the @Database annotation
- Specify supported entities and version
- Declare abstract method to get our DAO’s reference

Demo

Crypto Boom App

- Define our CoinDAO:
  - Define the way we access the data in our database
  - Define Conflict strategy
- Define our database and connect all components together

Data Migrations with Room

- Offers a simple mechanism to deal with data scheme changes

- Will throw an exception if something is wrong

- Define a migration for each database version

Room Will Throw an Exception

java.lang.IllegalStateException: Room cannot verify the data integrity. Looks like you've changed schema but forgot to update the version number. You can simply fix this by increasing the version number.

Room Will Throw an Exception

java.lang.IllegalStateException: A migration from 1 to 2 is necessary. Please provide a Migration in the builder or call fallbackToDestructiveMigration in the builder in which case Room will re-create all of the tables.

Summary

Data Persistence

- Advantages of persistence

- Memory and disk caches

Summary

Data Persistence

- Structure a proper data layer

- Traditional SQLite implementation

Summary

Room Persistence Library Components

- Entities

- DAOs

- Database definition

Summary

Room Persistence Library

- Leverage annotations

- Create common database functionality in two lines of code

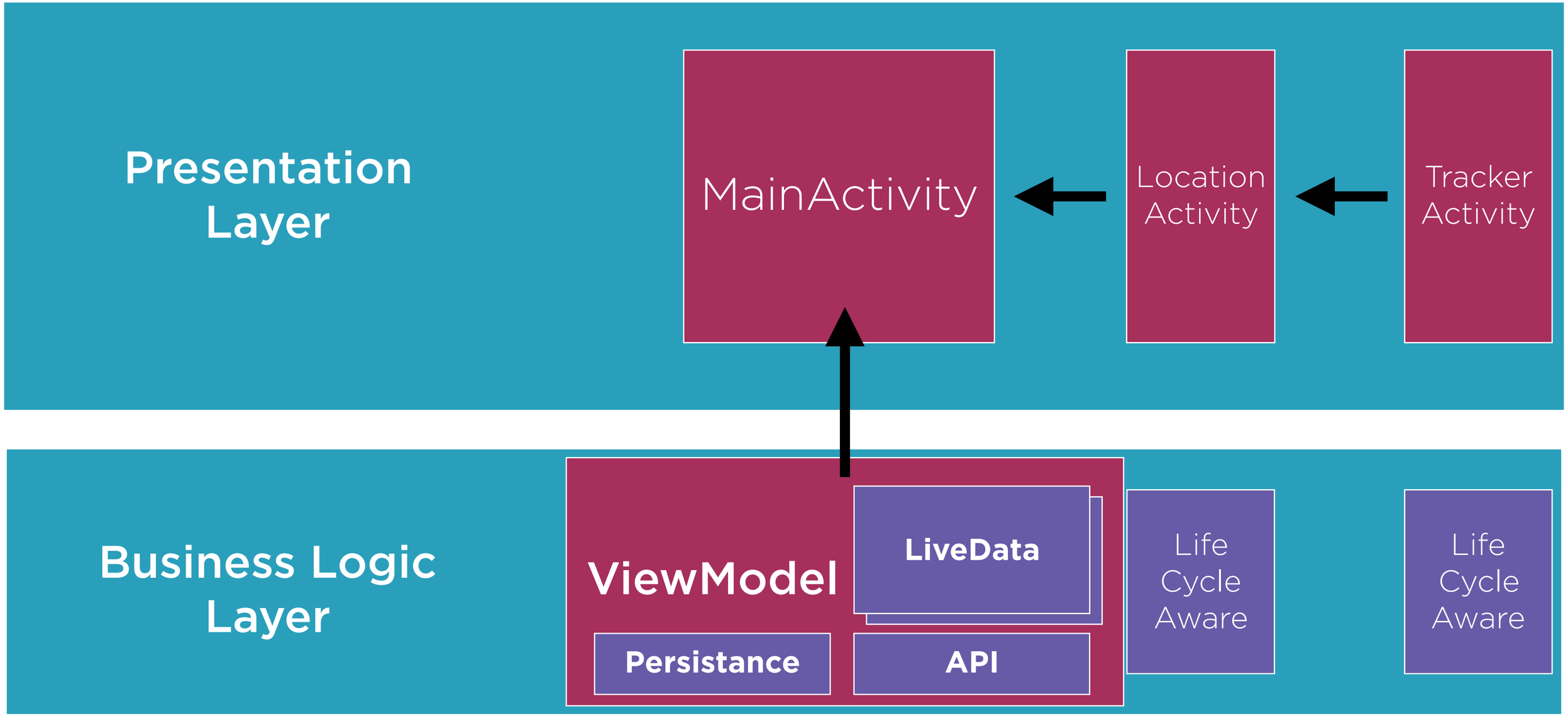
Summary

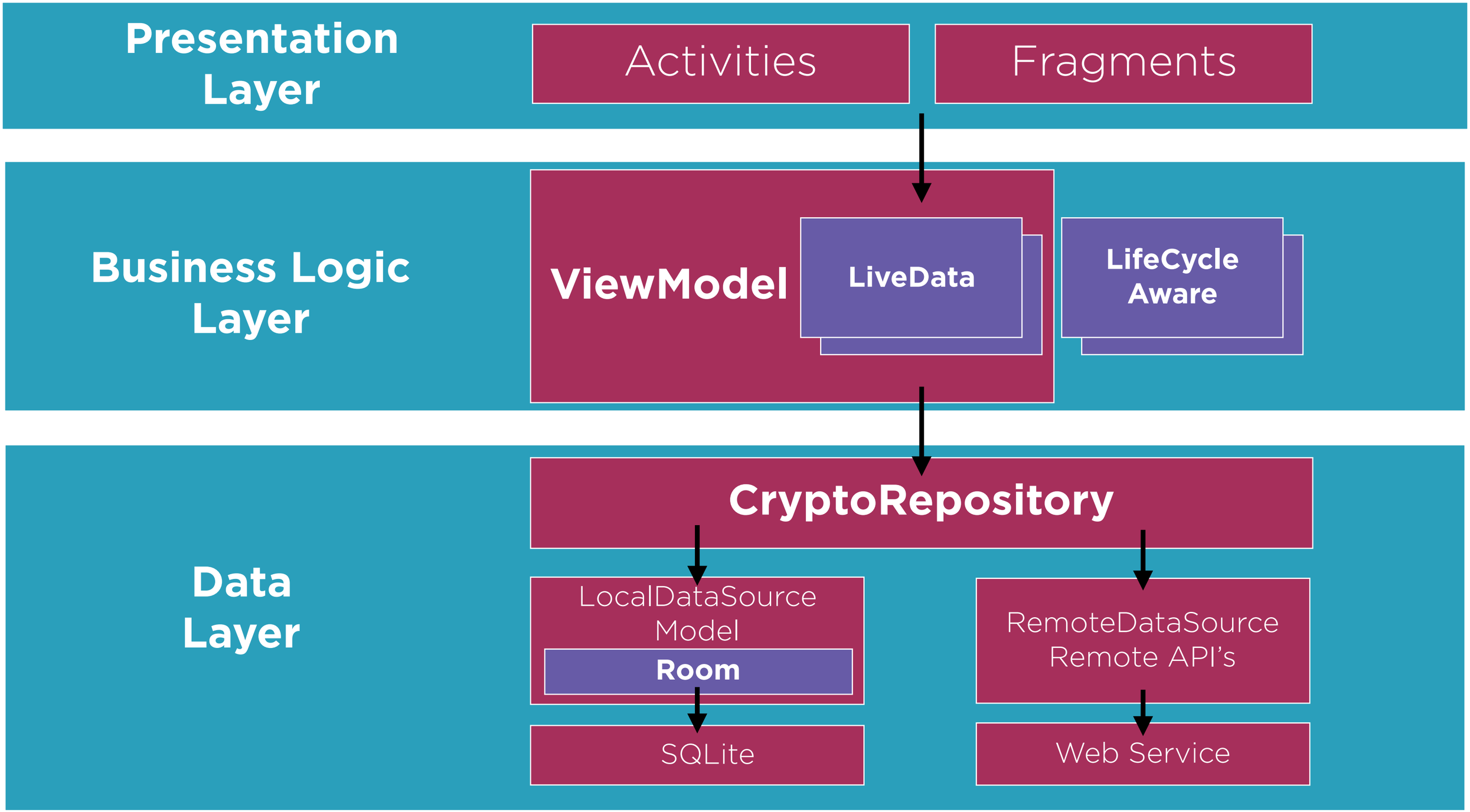
Room Persistence Library

- Utilize Room with the LiveData apis

- Data migrations

Before


Presentation Layer
MainActivity
Location Activity
Tracker Activity
Business Logic Layer
ViewModel
LiveData
Persistance
API
Life Cycle Aware
Life Cycle Aware


Presentation Layer
Activities
Fragments
Business Logic Layer
ViewModel
LiveData
LifeCycle Aware
Data Layer
CryptoRepository
LocalDataSource
Model
Room
SQLite
RemoteDataSource
Remote API's
Web Service