Programming with Android: App Guidelines part 1: Components

Federico Montori

Dipartimento di Informatica: Scienza e Ingegneria

Università di Bologna

Outline

- MVVM Design Pattern
- ViewModel
- LiveData
- MVVM vs. MVC
- Room
- Single Source of Truth
- An example with Retrofit

Architectural Components

- In time, the development in Android has changed quickly
  - Lack of architectural design patterns
  - Different native languages
  - Hybrid technologies
  - Handling bindings between views and controllers is tedious.
  - A lot of boilerplate code...

Architectural Components

Furthermore, well, you're on a smartphone, which means a lot more hassle:

- For example, you share a photo in your favorite social networking app
  - The app triggers a camera intent. The Android OS then launches a camera app to handle the request. So you leave the first app...
  - The camera app might trigger other intents, like launching the file chooser, which may launch yet another app.
  - Eventually, the user returns to the social networking app and shares the photo.
- At any point, the user could be interrupted by a phone call or notification. After acting this, the user should resume the photo sharing process...
- Keep in mind that the OS might kill some processes when needed

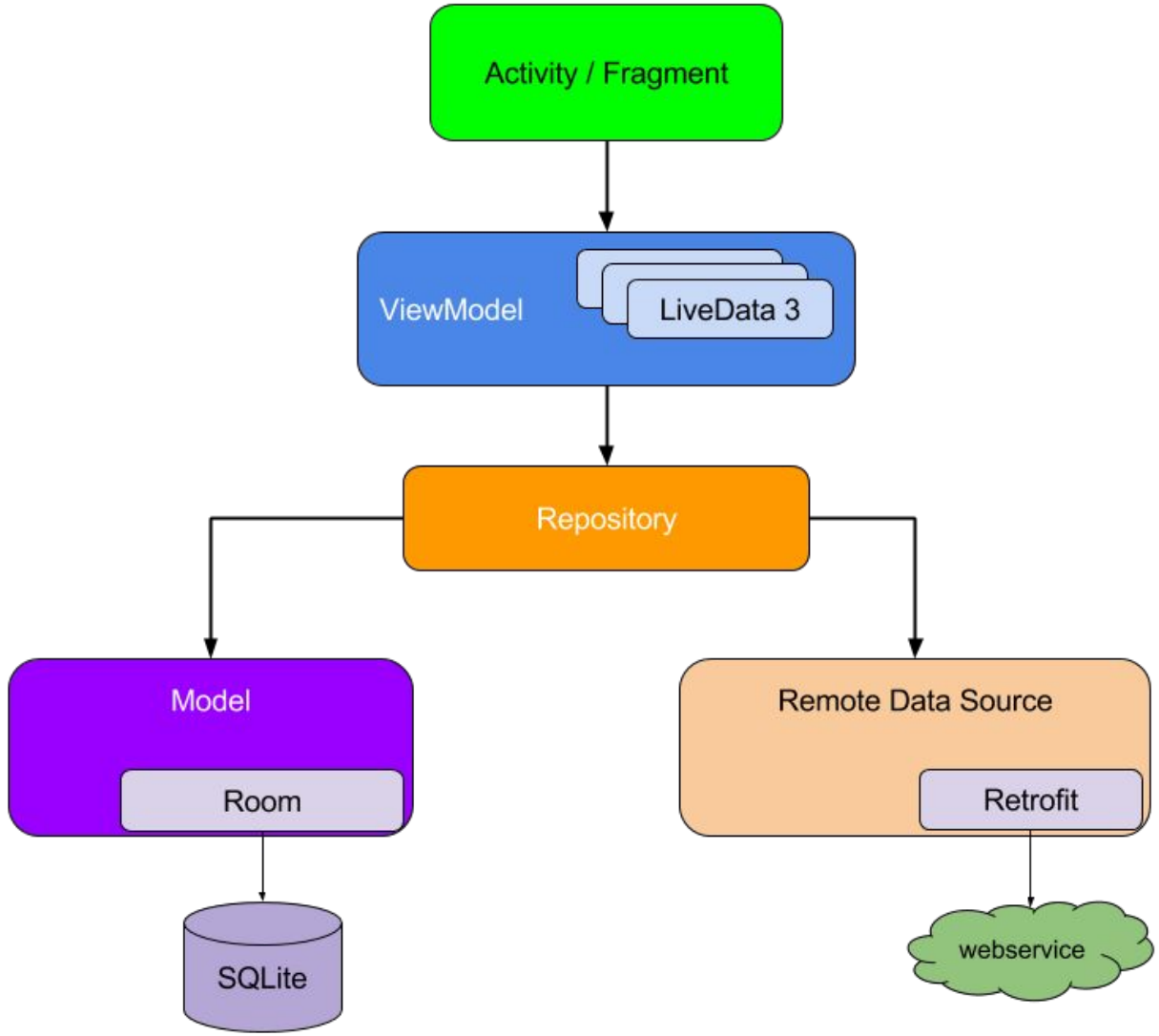
Given such condition, we need a solid architectural decoupling that ensures component are not depending on each other.

Model View ViewModel (MVVM)

In this lesson we will explore:

- ViewModel
- LiveData
- Room
- Retrofit

This is a good starting point for many apps but obviously it changes for other situations.



Architectural Components

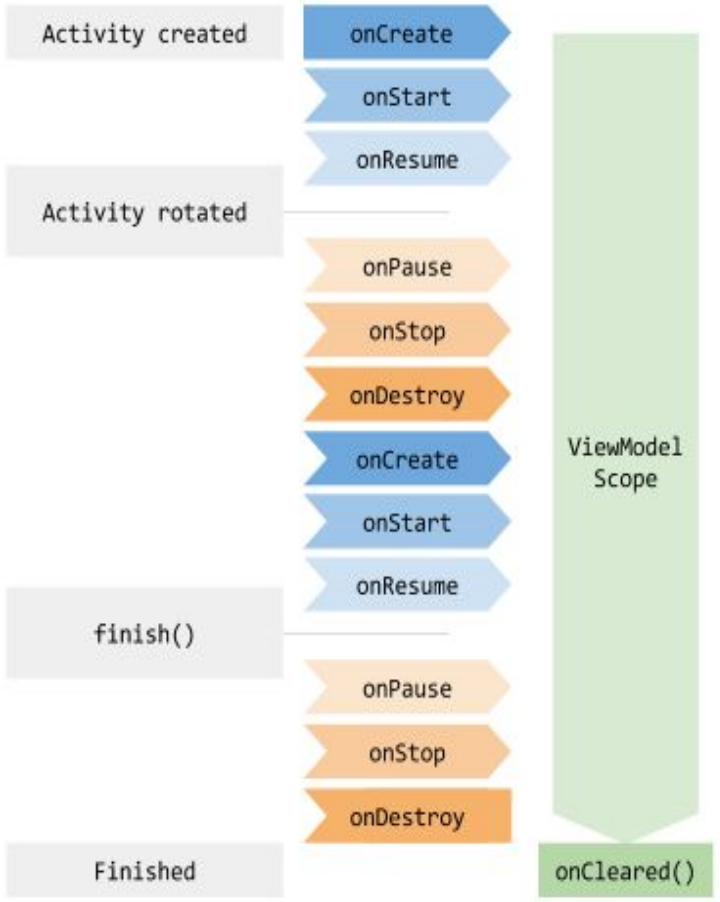
- Here we are delving into Android Jetpack Dependencies.
  - “suite of libraries, tools, and guidance to help developers write high-quality apps easier and following best practices”
  - Uses androidx.* stuff
- We will use something called “Android Components and we need to add all of these dependencies”

```
implementation "androidx.lifecycle:lifecycle-viewmodel:2.2.0"
implementation "androidx.lifecycle:lifecycle-livedata:2.2.0"
implementation "androidx.lifecycle:lifecycle-common-java8:2.2.0"
```

ViewModel

A ViewModel is a component that stores UI-related data in a Lifecycle-aware way.

- It helps surviving seamlessly configuration changes
- If the activity or the Fragment is destroyed and re-created there is no need for saving instance state every time (which is instead suitable only for small data).
- Separates view data ownership from UI controller logic.
  - One ViewModel per UI controller



Create a ViewModel

To create a ViewModel, first extend the ViewModel helper class:

```
public class MyViewModel extends ViewModel {
    private List<User> users;
    public List<User> getUsers() {
        // Do an asynchronous operation to fetch users.
        return users;
    }
}
```

Get the singleton from the Activity:

```
MyViewModel model = new ViewModelProvider(this).get(MyViewModel.class);
List<User> users = model.getUsers();
```

Create a ViewModel

ViewModel specifications:

- A ViewModel is scoped to the lifecycle of the object passed to the **ViewModelProvider** (this request makes it sort of singleton).
- A ViewModel never references elements of the View, the reference should be one-way only.
- Multiple Fragments can share the same ViewModel by passing **requireActivity**() to the ViewModelProvider.
- You also have application context-aware ViewModel, called AndroidViewModel (if you need reference to the application):

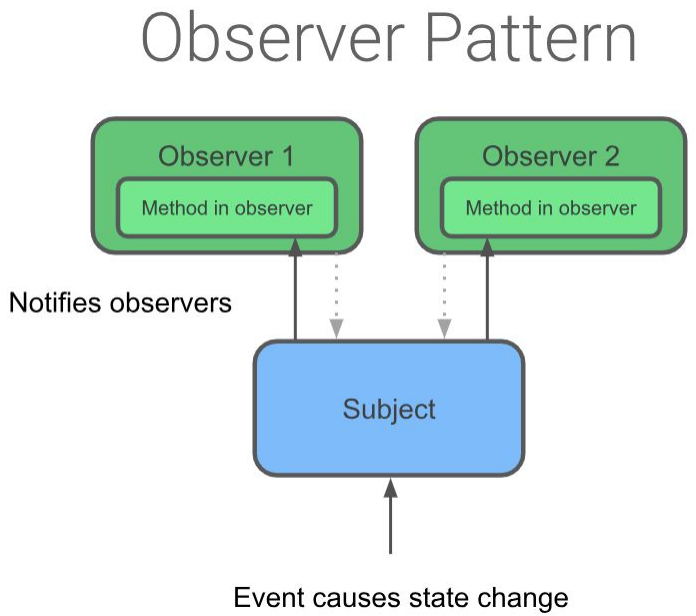
```
MyAndroidViewModel model = ViewModelProvider.AndroidViewModelFactory.
    getInstance(this.getApplication()).create(MyAndroidViewModel.class);
```

Observables

LiveData are based on the concept of Observables

- Observables are data classes that notify when changes on the observed data occur.
  - they wrap existing data types

```
public final ObservableField<String> name =
    new ObservableField<>();
public final ObservableInt age =
    new ObservableInt();
public final ObservableArrayList<String> users =
    new ObservableArrayList<>();
```



Life Cycle Awareness

For observables:

- can easily set/get their values
- need to subscribe to changes and design a callback function
- Part of RxJava (not only Android)...
- Cannot interact with the life cycle

LiveData are also based on the concept of LifeCycle Awareness

- Let’s leave observables for a second and see what these are

Life Cycle Awareness

You can implement LifeCycle awareness by implementing an Observer to the LifeCycle:

Useful when the component needs to react to lifecycle changes

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void function1() { … }
    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void function2() { … }
    }
}
myLifecycleOwner.getLifecycle().addObserver(new MyObserver());
```

Life Cycle Awareness

The function getLifecycle() can be called by a LifeCycleOwner

- an object implementing the LifeCycleOwner interface, i.e. it has a Lifecycle (Activities, Services, Fragments...)
- You can use powerful calls such as **lifecycle.getCurrentState().isAtLeast(STARTED))**
- You can create a class that implements the LifeCycleOwner interface

LiveData

LiveData are lifecycle-aware observable components that notify subscribers only when they are in active state (i.e. RESUMED or STARTED).

- Useful for activities and fragments because they can observe data and not worry about their state.
- First of all, design your Live Data to contain the actual data (just like the observer, it is a wrapper.
- MutableLiveData can change (it has a setter), LiveData cannot
- Instantiate them in your ViewModel

```
private MutableLiveData<String> currentName;
```

Creating LiveData

LiveData are typically instantiated in your ViewModel, which means that the observer is located elsewhere (i.e. the Activity). It is typically good practice to return an immutable or a mutable LiveData to the class that observes:

```
public MutableLiveData<String> getCurrentName() {
        if (currentName == null) {
            currentName = new MutableLiveData<String>();
        }
        return currentName;          // The observer can modify currentName
    }

public LiveData<String> getCurrentName() {
        if (currentName == null) {
            currentName = new MutableLiveData<String>();
        }
        return currentName;          // The observer cannot modify currentName
    }
```

Observing LiveData

You may want to start observe your LiveData in the Activity onCreate().

- LiveData delivers updates to active observers when data changes

```
model = new ViewModelProvider(this).get(NameViewModel.class);

final Observer<String> nameObserver = new Observer<String>() {
    @Override
    public void onChanged(@Nullable final String newName) {
        myTextView.setText(newName);
    }
};
model.getCurrentName().observe(this, nameObserver);
```

LifeCycleOwner

onChanged() is called every time currentName changes and as soon as observe is called if there is a value already.

Changing LiveData

LiveData values are updated by using:

- setValue() if called from the main thread
- postValue() if called from a worker thread

```
model.getCurrentName().postValue("New Name");
```

Remember that setValue() and postValue() are only callable against a MutableLiveData.

- If you want to pass LiveData to a class not in charge of modifying it, then only pass LiveData type.
- Typically ViewModel updates LiveData, Activity only observes
  - or calls a method in the ViewModel to update the LiveData

Other Components

LiveData and ViewModel are part of a bigger chunk of novelties that we will not explore. Here are the pointers:

For a tighter coupling between View elements and the UI controller we can also use:

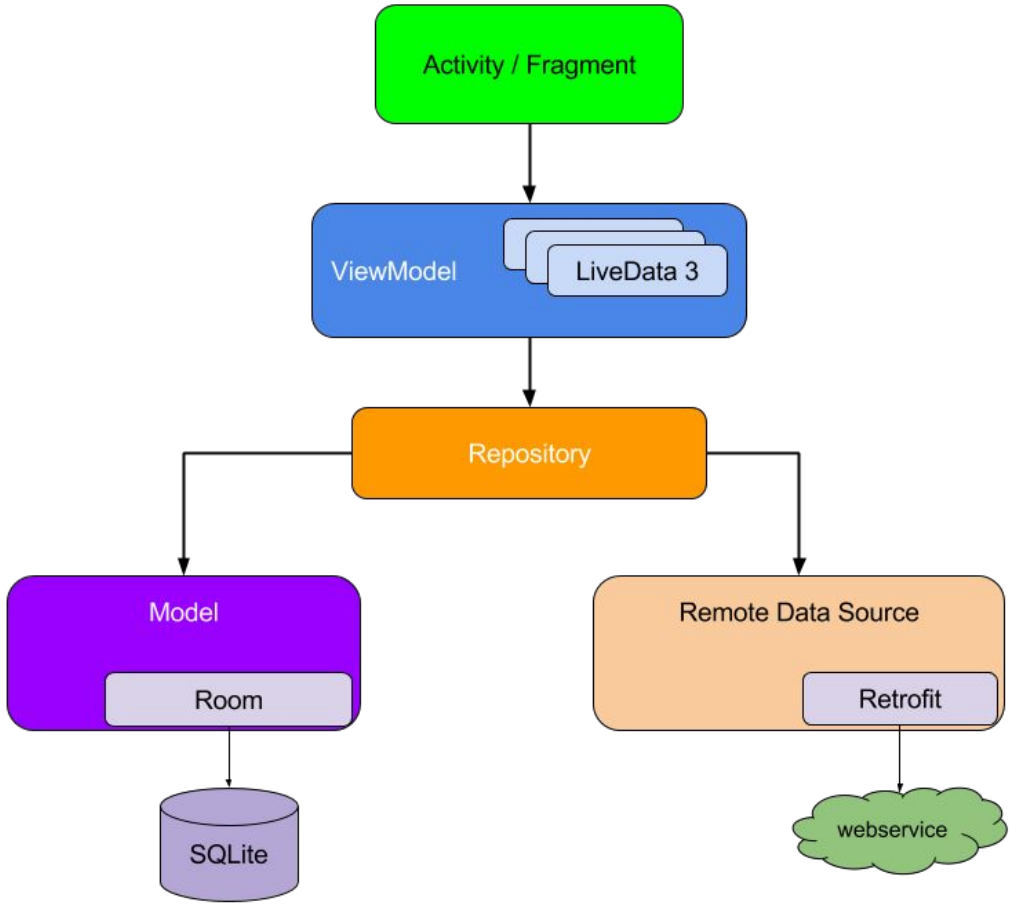
- Data Binding
  - https://developer.android.com/topic/libraries/data-binding
- View Binding
  - https://developer.android.com/topic/libraries/view-binding
- They both help in interacting declaratively with views (eliminating findViewById).

MVVM and MVC

Pit stop:
why then MVVM is different from MVC?

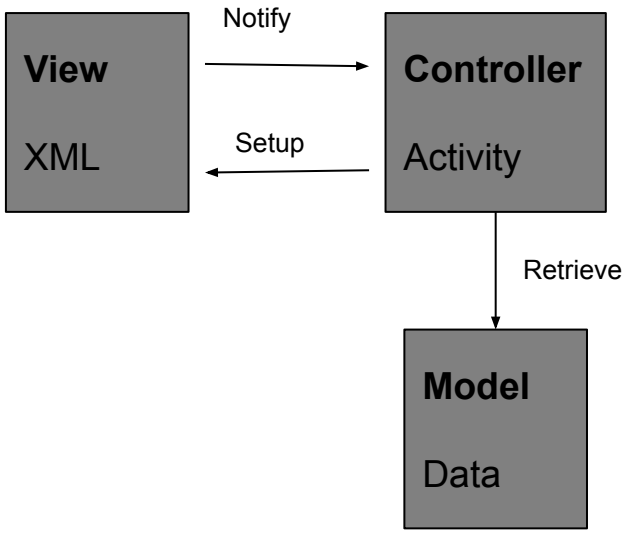
Layouts and static data is the **View**
Activities and ViewModel are the **Controller**
Persistence is the **Model**

… right?

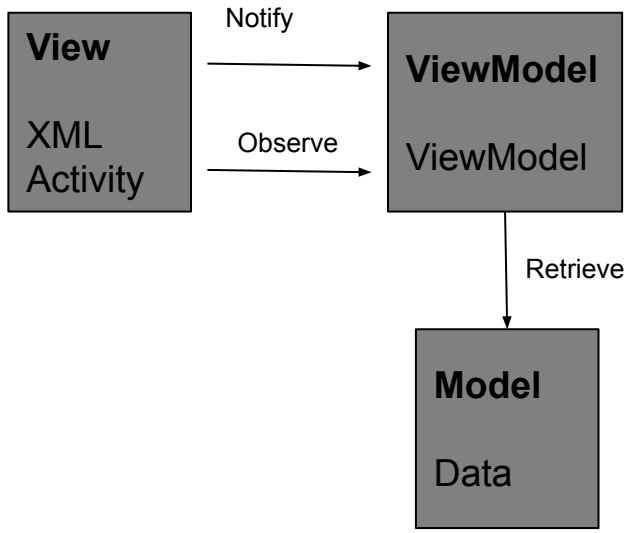


MVVM and MVC

Key differences are in different separation of concerns.



- Controller is the Active Part
- Easy to test Model
- Uneasy to test the Controller because is tied heavily to the API and the View.
- If we change the View, we change the controller



- View is the Active Part
- Business Logic separated from UI
- ViewModel prepares observable data
- Easier to test components separately.
- Need DataBinding to fully unleash...

Databases with Room

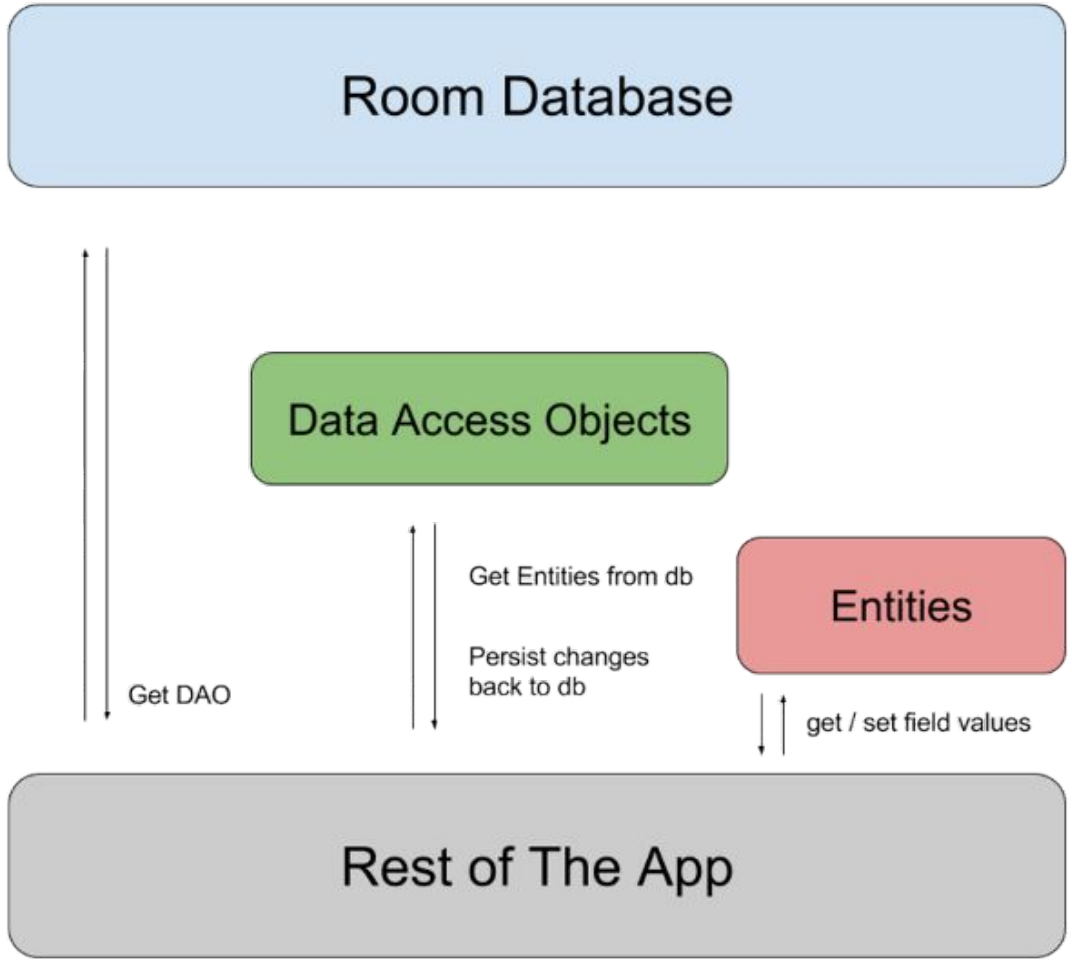
Let's talk about the **Model**

- Room provides an abstraction layer over SQLite
  - You should always use Room from now on
- Add it to your APP by adding in build.gradle

```
dependencies {
  def room_version = "2.2.5"
  implementation "androidx.room:room-runtime:$room_version"
  annotationProcessor "androidx.room:room-compiler:$room_version" // optional - Kotlin, RxJava and Guava Extensions and
Coroutines support for Room
  implementation "androidx.room:room-ktx:$room_version"
  implementation "androidx.room:room-rxjava2:$room_version"
  implementation "androidx.room:room-guava:$room_version"
  // Test helpers
  testImplementation "androidx.room:room-testing:$room_version"
}
```

Room architecture

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



Room components: Database

It has to be an abstract class extending RoomDatabase

```
@Database(version = 1, entities = {Entity1.class, Entity2.class})
abstract class myDatabase extends RoomDatabase {
    abstract public Entity1Dao entity1Dao();
    abstract public Entity2Dao entity2Dao();
    abstract public TwoEntitiesDao twoEntitiesDao();
}
```

It handles automatically the conversion from a Cursor to your APP classes

Room components: Entity

For each Entity, Room creates a database Table

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

```
@Entity
public class Entity1 {
    @PrimaryKey
    public int myId;

    public String firstField;

    public String secondField;
    @Ignore
    String tmp;
}
```

Room components: Entity

- Entities fields needs to be either public or 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)
private int uid;
```

Room components: Entity

- Room creates a table with the Entity name
  - Change it with

```
@Entity(tableName = "users")
```

  - Same goes for the columns

```
@ColumnInfo(name = "first_name")
    public String firstName;

@ColumnInfo(name = "last_name")
    public String lastName;
```

- Speed up queries with Indices

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

Room components: Entity

- 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

```
Class Material {
        public String name;
        public String weight;
}

@Entity
Class myEntity {
        …
        @Embedded
        public Material objectMaterial;
}
```

Room components: Relationships

- Defining relations in a more complex way

```
public class Entity1AndEntity2 {
    @Embedded public Entity1 e1;
    @Relation(
        parentColumn = "id",
        entityColumn = "user_id"
    )
    public Entity2 e2;
}
```

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 components: DAO

- You need DAOs to access data
- A DAO can be either an interface or an abstract class
- Room creates DAO implementations at compile time

- Syntax

```
@Dao
public interface MyDao {
    @QueryType(params..)
    public void method(method parameters);
}
```

- @QueryType can be:
  - @Insert, @Update, @Delete, @Query

Room components: DAO

- 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
- DO NOT implement it

DAO: Query examples

- @Insert

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
public void insertUsers(User... users);

@Insert
public void insertBothUsers(User user1, User user2);

@Insert
public void insertUsersAndFriends(User user, List<User> friends);
```

- @Update

```
@Update
public void updateUsers(User... users);
```

- @Delete

```
@Delete
public void deleteUsers(User... users);
```

- @Query

```
@Query("SELECT * FROM user")
public User[] loadAllUsers();
```

- @Query + parameters

```
@Query("SELECT * FROM user WHERE age > :minAge")
public User[] loadAllUsersOlderThan(int minAge);
```

Room components: DAO

- Query on multiple tables:

```
@Dao
public interface MyDao {
    @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")
   public List<Book> findBooksBorrowedByNameSync(String userName);
}
```

Query a relation

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

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: migrating databases

- 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(version = 1, entities = {Entity1.class, Entity2.class})
abstract class myDatabase extends RoomDatabase {
…
}
```

Room: migrating databases

- Each Migration class defines a startVersion and endVersion
  - At runtime, Room runs each migrate method in order

```
Room.databaseBuilder(getApplicationContext(), MyDb.class, "database-name")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build();

static final Migration MIGRATION_1_2 = new Migration(1, 2) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, "
                + "`name` TEXT, PRIMARY KEY(`id`))");
    }
};

static final Migration MIGRATION_2_3 = new Migration(2, 3) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("ALTER TABLE Book "
                + " ADD COLUMN pub_year INTEGER");
    }
};
```

To build a Content Provider with Room

- Define your resources (let's say it's a db)
- Implement the CRUD operations

```
public class ExampleProvider extends ContentProvider {
    private AppDatabase appDatabase;
    private UserDao userDao;
    private static final String DBNAME = "mydb";
    public boolean onCreate() {
        appDatabase = Room.databaseBuilder(getContext(), AppDatabase.class, DBNAME).build();
        userDao = appDatabase.getUserDao();
        return true; }
    public Cursor insert ( Uri uri, ContentValues values) {
        // Here do your ops against the DB….
    }
```

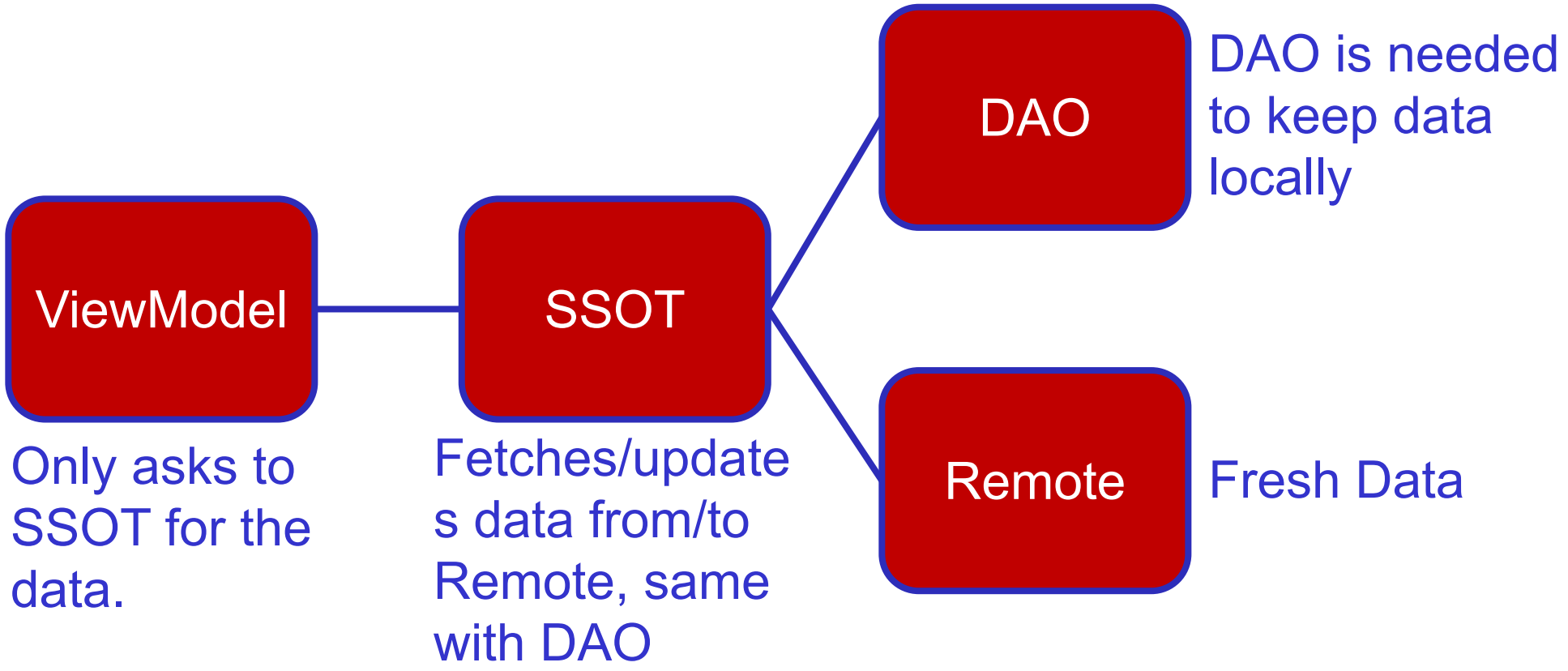
Room and LiveData

- The Room persistence library supports observable queries, which return LiveData objects.
- Observable queries are written as part of a DAO
- Do not need to explicitly run them into a separate Thread (it is done by default).
- Changes in the Database are immediately notified to the LiveData

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


// Meanwhile in your ViewModel (or Repository)
private LiveData<List<User>> myList;
myList = userDao.loadAllUsersObservable();
```

SSOT model


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 model and Repository

- SSOT model 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.
- You may need an intermediate **Repository** class that handles all the different calls to data sources.

SSOT model and Repository

- BASIC idea (you can implement with whatever HTTP client you want)

```
public LiveData<List<User>> loadAllUsersSSOT() {
        RequestQueue queue = Volley.newRequestQueue(this);
        StringRequest stringRequest = new StringRequest(Request.Method.GET,
        "http://fakedata.io/getUsers",
        new Response.Listener<String>() {
            @Override
                public void onResponse(String response) {  INSERT USERS INTO LOCAL DATABASE   }
            }, new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError error) {  // do nothing   }
        });
        queue.add(stringRequest);
    return loadAllUsersObservable;
  }
```

Retrofit

- 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
- 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 entities

- Just design a normal data class with setters and getters
  - Use the SerializedName to specify what name it has in the JSON/XML data frame.

```
public class RetroPhoto {
   @SerializedName("albumId")
   private Integer albumId;
   @SerializedName("id")
   private Integer id;
   public RetroPhoto(Integer albumId, Integer id) {
      this.albumId = albumId;
      this.id = id;
   }
//Setters and getters here…
}
```

Retrofit unique client

- Then set up the Retrofit client
  - Better to do it in a singleton-like fashion (this one translates JSON)

```
public class RetrofitClientInstance {
   private static Retrofit retrofit;
   private static final String BASE_URL = "https://jsonplaceholder.typicode.com";
   public static Retrofit getRetrofitInstance() {
      if (retrofit == null) {
         retrofit = new retrofit2.Retrofit.Builder().baseUrl(BASE_URL).addConverterFactory(GsonConverterFactory.create())
               .build();
      }
      return retrofit;
   }
}
```

Retrofit Interfaces

- 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...

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

- 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 Calls

- Just like with other HTTP clients, such as Volley, enqueue the call:

```
GetDataService service = RetrofitClientInstance.getRetrofitInstance().create(GetDataService.class);
    Call<List<RetroPhoto>> call = service.getAllPhotos();
    call.enqueue(new Callback<List<RetroPhoto>>() {
        @Override
        public void onResponse(Call<List<RetroPhoto>> call, Response<List<RetroPhoto>> response) {
            myList = Response.body();  // In ssot here we should also update the db...
        }
        @Override
        public void onFailure(Call<List<RetroPhoto>> call, Throwable t) {
            // Handle Errors...
        }
    });
```

- This is basically it, with the advantage that retrofit Entities could also be Room entities
  - It does not have to be like it always, it really depends...

Firebase

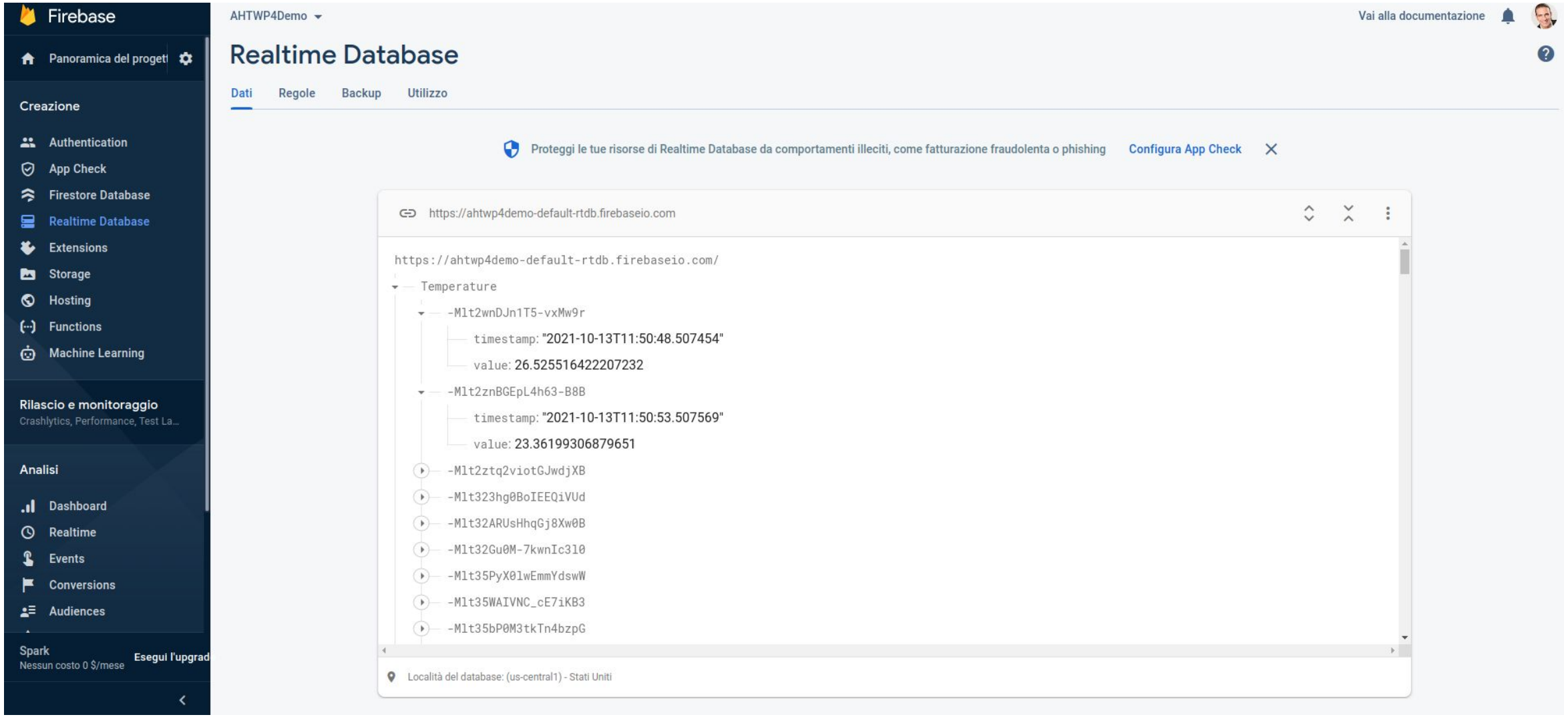
- Firebase is a Google app development platform that gives you an easy-to use and reactive backend for your app.
  - Realtime Database:

    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 Console



Example: Firebase calls

- Here we can see how the result of the Firebase query gets passed to a LiveData, so we have two nested listeners:

```
FirebaseDatabase mDatabase = FirebaseDatabase.getInstance("https://wp4demo-default-rtdb.firebaseio.com");
MutableLiveData<TemperatureDataPoint> tempPoint = new MutableLiveData<>();
        mDatabase.getReference("Temperature")
                .addChildEventListener(new ChildEventListener() {
                    @Override
                    public void onChildAdded(@NonNull DataSnapshot snapshot,
                                             @Nullable String previousChildName) {
                        tempPoint.postValue(snapshot.getValue(TemperatureDataPoint.class));
                    }
      [...]
});
```

- Here I am just interested in data when it gets added, but I can also use a generic call like **onDataChanged()**