# Data Persistence using Room

# Agenda

- Room
  - Entity
  - DAO
  - Database
- Android Jetpack Architecture Components
  - Repository
  - ViewModel
  - LiveData

# Room

Abstraction layer over SQLite to allow fluent database access for data stored locally

Part of Android Jetpack

Less boilerplate

Maps database values to objects - Object Relational Mapping (ORM)

Compile time validation on SQL queries

Support for observation – LiveData and RxJava

# Configure Room in App

- To use Room in your app, add the following dependencies to your app's build.gradle file:

```
implementation "androidx.room:room-runtime:2.3.0"
annotationProcessor "androidx.room:room-compiler:2.3.0"
```

# Room Architecture Diagram

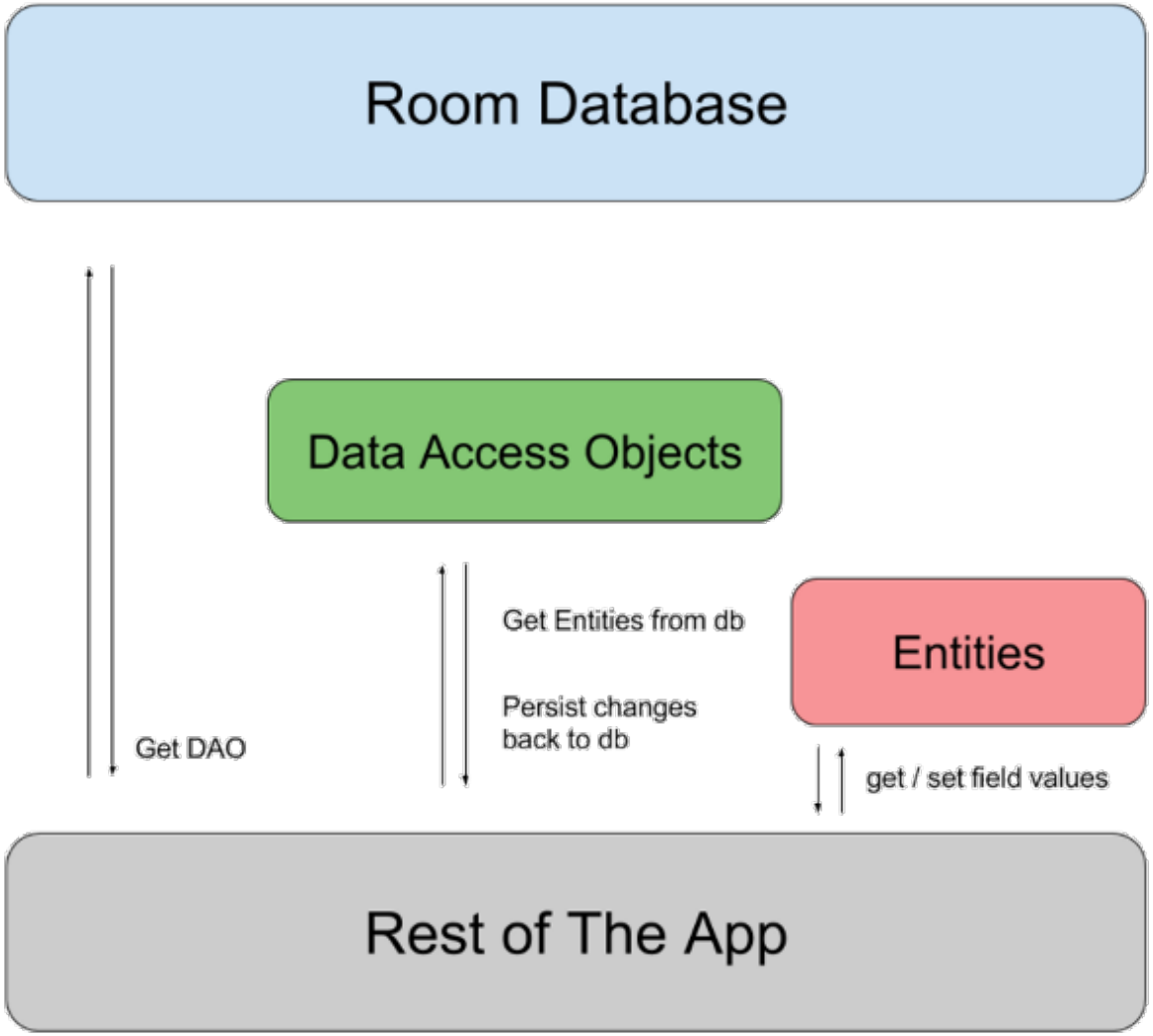



*Image source: https://developer.android.com/*

# Components of Room – The annotations

1. @Database:
   - Contains the database holder and serves as the main access point for the underlying connection to your app's persisted, relational data.
   - The class that's annotated with @Database should satisfy the following conditions:
     - Be an abstract class that extends RoomDatabase.
     - Include the list of entities associated with the database within the annotation.
     - Contain an abstract method that has 0 arguments and returns the class that is annotated with @Dao.
     - At runtime, you can acquire an instance of Database by calling Room.databaseBuilder() or Room.inMemoryDatabaseBuilder().

# Components of Room cont…

2. @Entity:
   - Represents a table within the database.
3. @Dao:
   - Contains the methods used for accessing the database.

- The app uses the Room database to get the data access objects, or DAOs, associated with that database.
- The app then uses each DAO to get entities from the database and save any changes to those entities back to the database.
- Finally, the app uses an entity to get and set values that correspond to table columns within the database.

# @Entity

- When using the Room persistence library, you define sets of related fields as entities.
- For each entity, a table is created within the associated Database object to hold the items.
- You must reference the entity class through the entities array in the Database class.

# @Entity – Primary keys

- Each entity must define at least 1 field as a primary key.
- Even when there is only 1 field, you still need to annotate the field with the @PrimaryKey annotation.
- Also, if you want Room to assign automatic IDs to entities, you can set the @PrimaryKey's autoGenerate property.
- If the entity has a composite primary key, you can use the primaryKeys property of the @Entity annotation, as shown in the following code snippet:

```
@Entity(primaryKeys = arrayOf("first_name", "last_name"))
public class User(
....
)
```

# @Entity cont...

- By default, Room uses the class name as the database table name. If you want the table to have a different name, set the tableName property of the @Entity annotation.

- Room uses the field names as the column names in the database. If you want a column to have a different name, add the @ColumnInfo annotation to a field.

- By default, Room creates a column for each field that's defined in the entity. If an entity has fields that you don't want to persist, you can annotate them using @Ignore.

# @Dao – Data Access Objects

- The set of Dao objects forms the main component of Room, as each DAO includes methods that offer abstract access to your app's database.

- A DAO can be either an interface or an abstract class.
- If it's an abstract class, it can optionally have a constructor that takes a RoomDatabase as its only parameter.
- Room creates each DAO implementation at compile time.

# @Dao cont...

- Room doesn't support database access on the main thread unless you've called allowMainThreadQueries() on the builder because it might lock the UI for a long period of time.

- Asynchronous queries that return instances of LiveData or Flowable are exempt from this rule because they asynchronously run the query on a background thread when needed.

# @Insert

- When you create a DAO method and annotate it with @Insert, Room generates an implementation that inserts all parameters into the database in a single transaction.

- If the @Insert method receives only 1 parameter, it can return a long, which is the new rowId for the inserted item. If the parameter is an array or a collection, it should return long[] or List<Long> instead.

# @Insert cont...

- The onConflict property of @Insert annotation indicates what to do if a conflict happens.
- There 3 different ways to handle conflict:
  - OnConflictStrategy.ABORT (default) : to roll back the transaction on conflict.
  - OnConflictStrategy.REPLACE : replace the existing rows with the new rows.
  - OnConflictStrategy.IGNORE : keep the existing rows.

# @Update

- The Update convenience method modifies a set of entities, given as parameters, in the database.
- It uses a query that matches against the primary key of each entity.
- You can have this method return an int value instead, indicating the number of rows updated in the database.

# @Delete

- The Delete convenience method removes a set of entities, given as parameters, from the database. It uses the primary keys to find the entities to delete.
- You can have this method return an int value instead, indicating the number of rows removed from the database.

# @Query

- @Query annotation allows you to perform read/write operations on a database.
- Each @Query method is verified at compile time, so if there is a problem with the query, a compilation error occurs instead of a runtime failure.
- Room also verifies the return value of the query such that if the name of the field in the returned object doesn't match the corresponding column names in the query response.
- Room gives a warning if only some field names match or an error if no field names match.

# Observable queries with LiveData

- When performing queries, you'll often want your app's UI to update automatically when the data changes.
- To achieve this, use a return value of type LiveData in your query method description.
- Room generates all necessary code to update the LiveData when the database is updated.

# @Database

- @Database annotation allows you annotate the class as the database holder.
- The class must be abstract and must inherit the RoomDatabase().
- The @Database annotation accepts set of entities that will create tables in the database and a database version number.
- The entities should be the class names in which @Entity has been defined.

- The RoomDatabase() instance defines distinct abstract getter method that returns an instance of DAOs for individual entities.

# Get a database connection

- The build() method of Room.databaseBuilder() will create the instance of database for given context, class name and database name.
- The fallbackToDestructiveMigration() indicates that if the database schema has changed delete everything from database. Alternatively, you can use Migration classes for migrating Room databases.

# Database - Singleton cont…

- If your app runs in a single process, you should follow the singleton design pattern when instantiating an AppDatabase object.

- Each RoomDatabase instance is fairly expensive, and you rarely need access to multiple instances within a single process.

- Kotlin companion object could be used for creating singleton instance of database that can be shared by entire app.

# Android Jetpack Architecture Components

# Android Jetpack Architecture Components

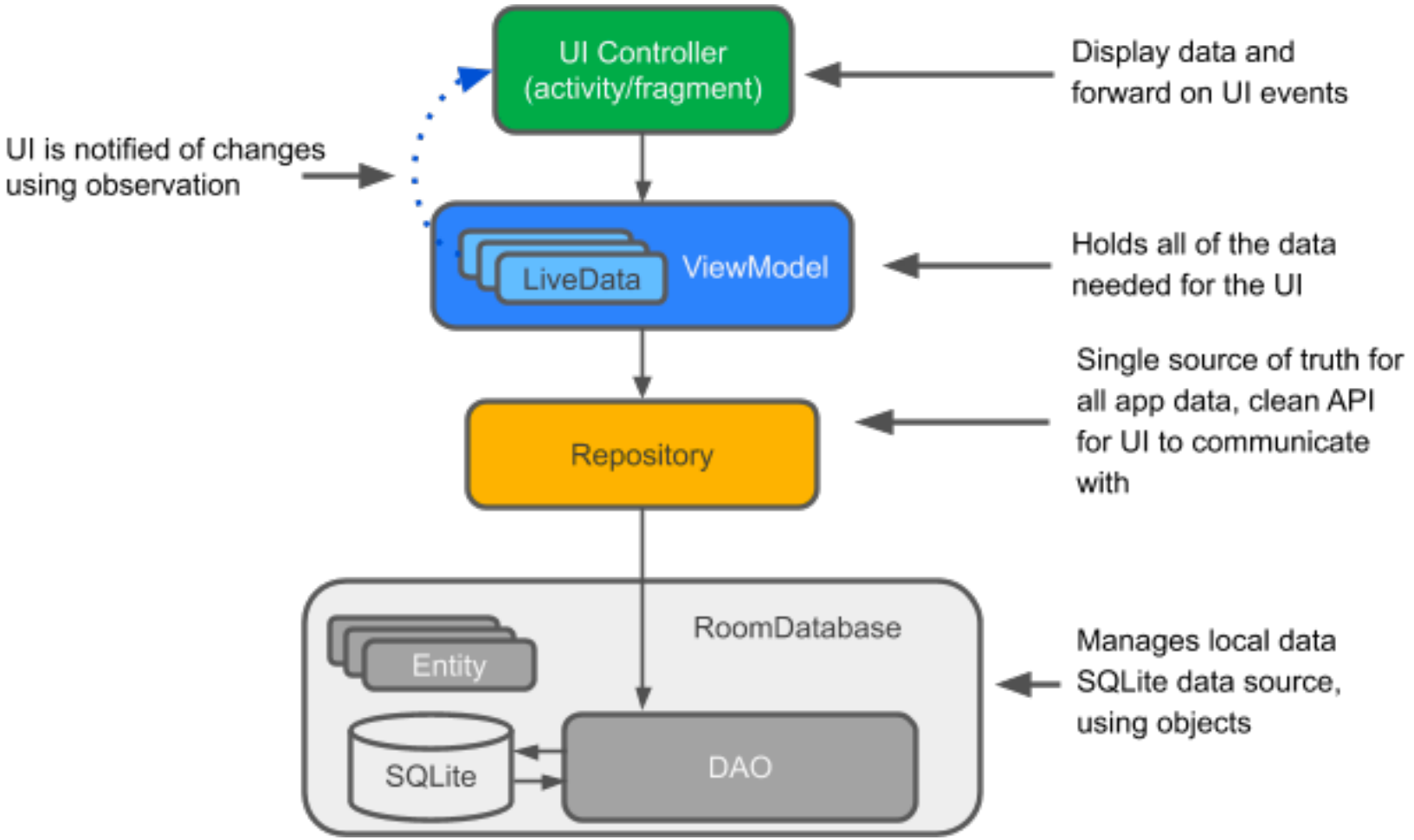



Image source: https://codelabs.developers.google.com/

# Repository – The mediator

- A repository class abstracts access to multiple data sources.
- The repository is not part of the Architecture Components libraries, but is a suggested best practice for code separation and architecture.
- A Repository class provides a clean API for data access to the rest of the application.
- In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in a local database.

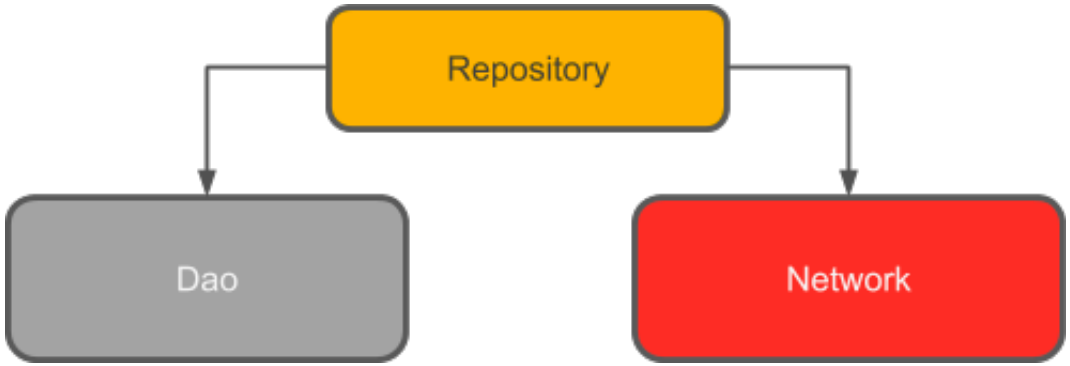
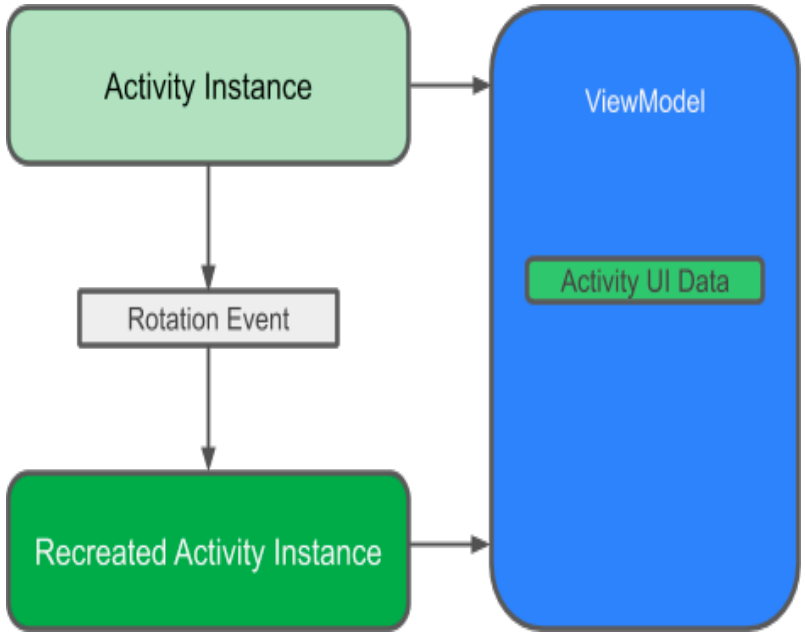



Image source: https://codelabs.developers.google.com/

# ViewModel

- The ViewModel's role is to provide data to the UI and survive configuration changes.
- A ViewModel acts as a communication center between the Repository and the UI.
- You can also use a ViewModel to share data between fragments.
- The ViewModel is part of the lifecycle library.



# ViewModel - Benefits

- A ViewModel holds your app's UI data in a lifecycle-conscious way that survives configuration changes.
- Separating your app's UI data from your Activity and Fragment classes lets you better follow the single responsibility principle:
  - Your activities and fragments are responsible for drawing data to the screen, while
  - your ViewModel can take care of holding and processing all the data needed for the UI.

- You should use LiveData for changeable data that the UI will use.

# ViewModel cont...

- You can create a viewmodel class that extends ViewModel class or AndroidViewModel class.
- If you need the application context (which has a lifecycle that lives as long as the application does), use AndroidViewModel.

- The Repository and the UI are completely separated by the ViewModel.

- There are no database calls from the ViewModel (this is all handled in the Repository), making the code more testable.

- You should not keep a reference to a context that has a shorter lifecycle than your ViewModel such as Activity, Fragment or View.
- ViewModels don't survive the app's process being killed in the background when the OS needs more resources.

# The LiveData class

- LiveData is an observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services.
- This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

- LiveData considers an observer, which is represented by the Observer class, to be in an active state if its lifecycle is in the STARTED or RESUMED state.
- LiveData only notifies active observers about updates.

# The LiveData class cont…

- Use a return value of type LiveData in your method description, Room generates all necessary code to update the LiveData when the database is updated.

- If you want to update data stored within LiveData, you must use MutableLiveData instead of LiveData.
- The MutableLiveData class has two public methods that allow you to set the value of a LiveData object, setValue(T) and postValue(T).
- Usually, MutableLiveData is used within the ViewModel, and then the ViewModel only exposes immutable LiveData objects to the observers.

# LiveData - Benefits

- Ensures your UI matches your data state
- No memory leaks
- No crashes due to stopped activities
- No more manual lifecycle handling
- Always up to date data
- Proper configuration changes
- Sharing resources

# Connect with the data from Activity

- Use ViewModelProvider to associate your ViewModel with your Activity.
- When your Activity first starts, the ViewModelProviders will create the ViewModel.
- When the activity is destroyed, for example through a configuration change, the ViewModel persists.
- When the activity is re-created, the ViewModelProviders return the existing ViewModel.

# References

- https://developer.android.com/training/data-storage/room
- https://developer.android.com/training/data-storage/room/defining-data
- https://developer.android.com/training/data-storage/room/accessing-data
- https://developer.android.com/jetpack/guide
- https://developer.android.com/topic/libraries/architecture/index.html
- https://developer.android.com/topic/libraries/architecture/livedata