Advanced Android Development V2

Storing data with Room

Lesson 10

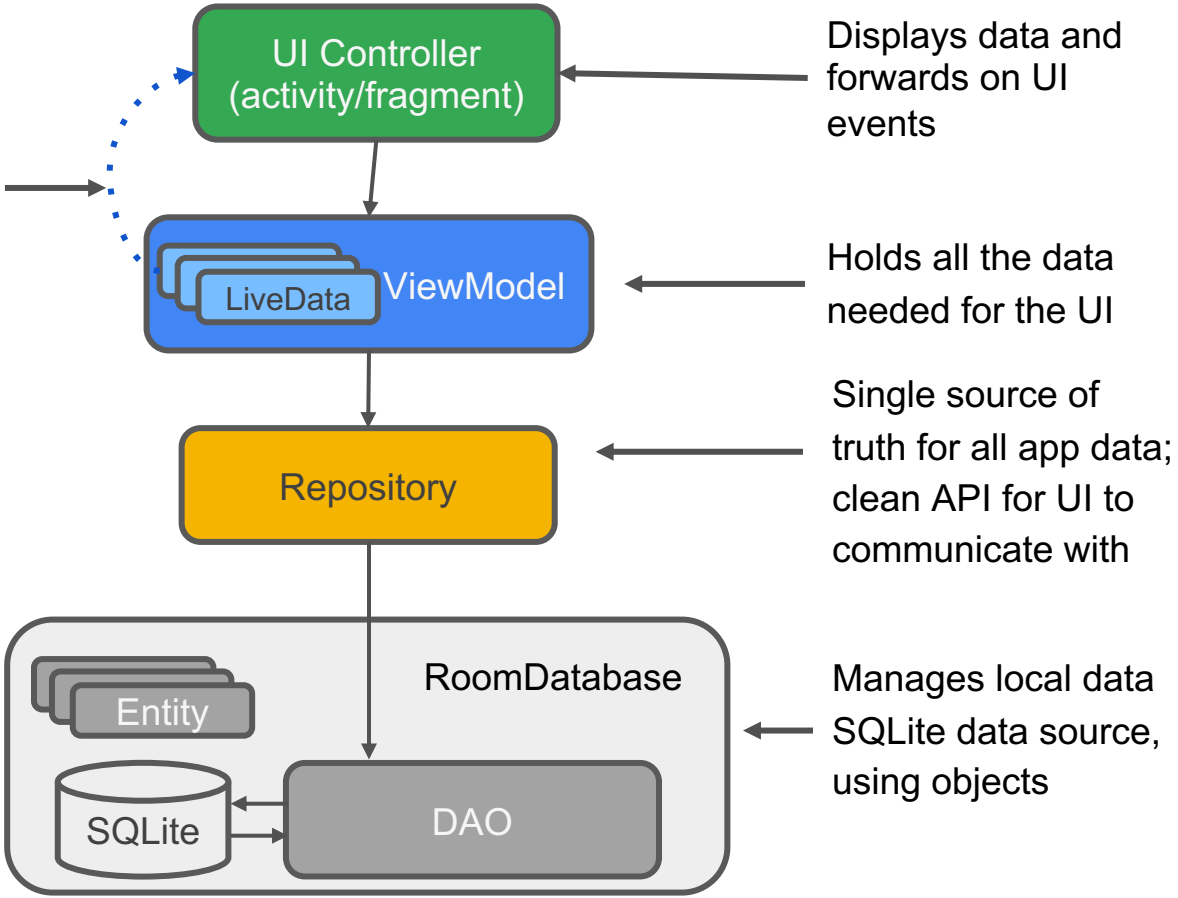
Contents

- Architecture Components
- Entity
- DAO
- Room database
- `LiveData`

Architecture Components

- Consist of best architecture practices + libraries
- Encourage recommended app architecture
- A LOT LESS boilerplate code
- Testable because of clear separation
- Fewer dependencies
- Easier to maintain

Overview



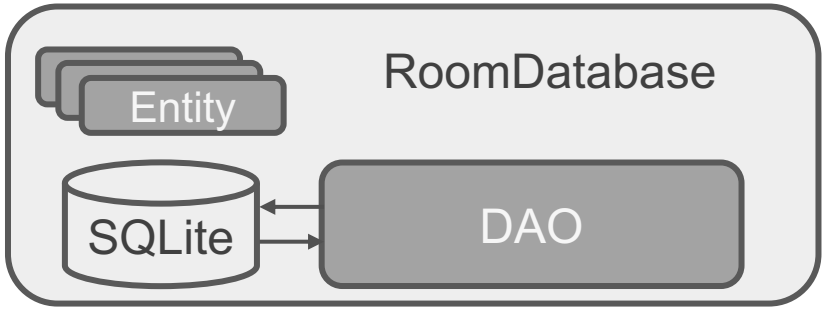
Components of Room

- **Entity**: Defines schema of database table.
- **DAO**: Database Access Object

  Defines read/write operations for database.
- **Database**:

  A database holder.

  Used to create or connect to database

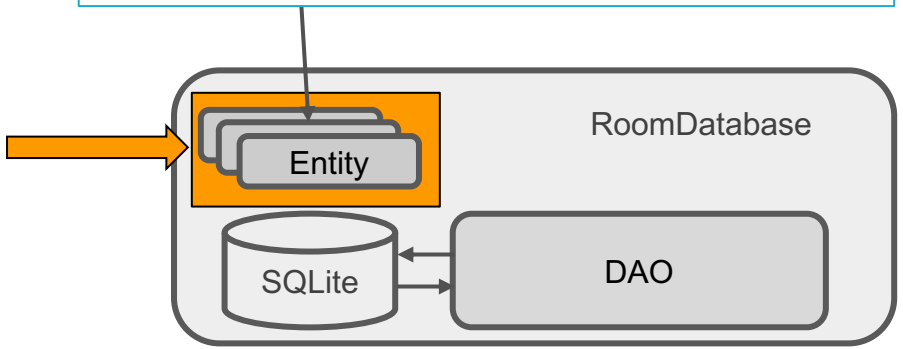


Entity

Entity

- Entity instance = row in a database table
- Define entities as POJO classes
- 1 instance = 1 row
- Member variable = column name

```
public class Person {
    private int uid;
    private String firstName;
    private String lastName;
}
```



Entity instance = row in a database table

```
public class Person {
    private int uid;
    private String firstName;
    private String lastName;
}
```

uid	firstName	lastName
12345	Aleks	Becker
12346	Jhansi	Kumar

Annotate entities

```
@Entity
public class Person {

    @PrimaryKey (autoGenerate=true)
    private int uid;

    @ColumnInfo(name = "first_name")
    private String firstName;

    @ColumnInfo(name = "last_name")
    private String lastName;

    // + getters and setters if variables are private.
}
```

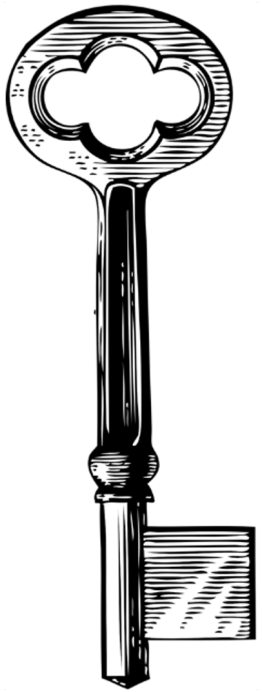
@Entity annotation

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

- Each `@Entity` instance represents an entity/row in a table
- Specify the name of the table if different from class name

@PrimaryKey annotation

```
@PrimaryKey (autoGenerate=true)
```



- **`Entity`** class must have a field annotated as primary key
- You can auto-generate unique key for each entity
- See Defining data using Room entities

@NonNull annotation

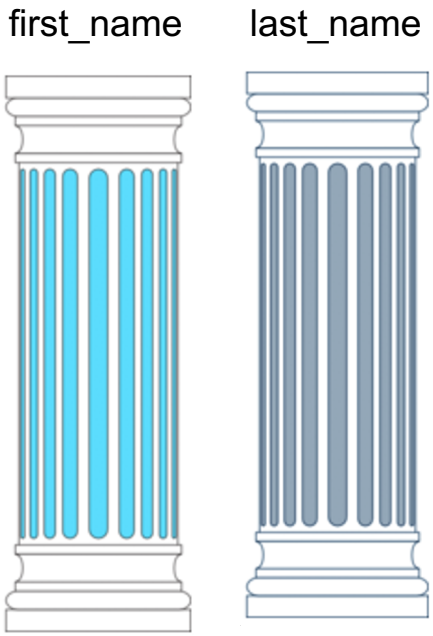
`@NonNull`

- Denotes that a parameter, field, or method return value can never be `null`
- Use for mandatory fields
- Primary key must use `@NonNull`

@ColumnInfo annotation

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

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

- Specify column name if different from member variable name



Getters, setters

Every field that's stored in the database must

- be public

OR

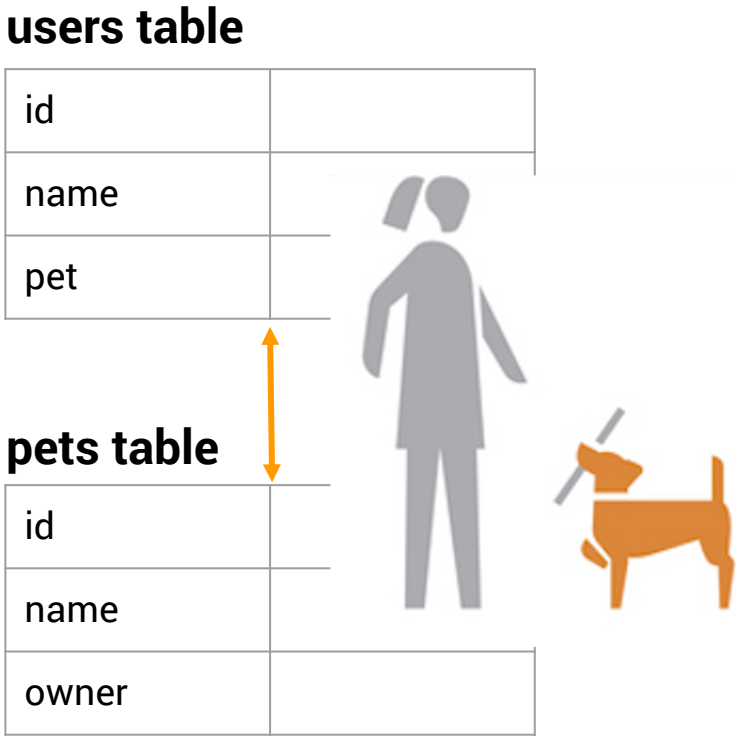
- have a "getter" method

  … so that Room can access it

Relationships

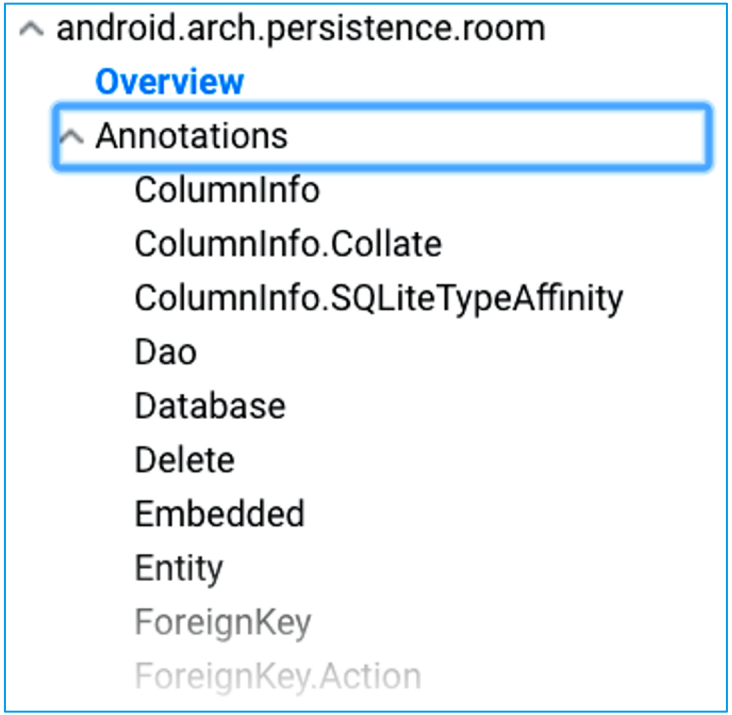
Use `@Relation` annotation to define related entities

Queries fetch all the returned object's relations



Many more annotations

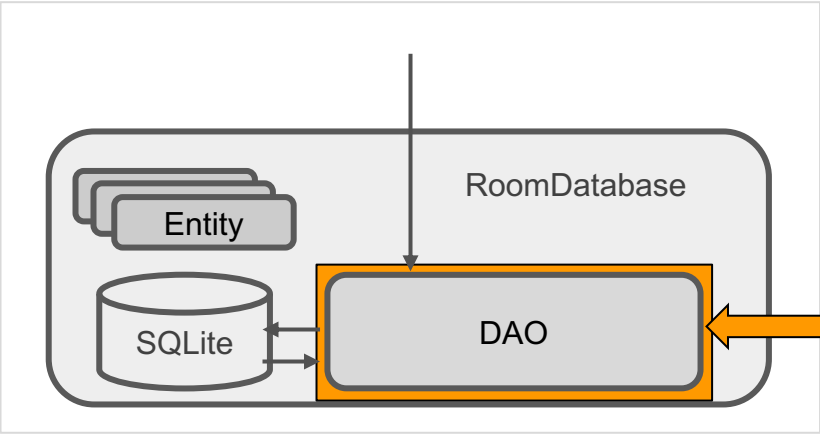
For more annotations, see [Room package summary reference]



Data access object (DAO)

Data access object

Use *data access objects*, or DAOs, to access app data using the Room persistence library



Data access object

- DAO methods provide abstract access to the app's database
- The data source for these methods are entity objects
- DAO must be `interface` or `abstract` class
- Room uses DAO to create a clean API for your code

Example DAO

```
@Dao
public interface WordDao {

    @Insert
    void insert(Word word);

    @Update
    public void updateWords(Word... words);

}
```

//... More queries on next slide...

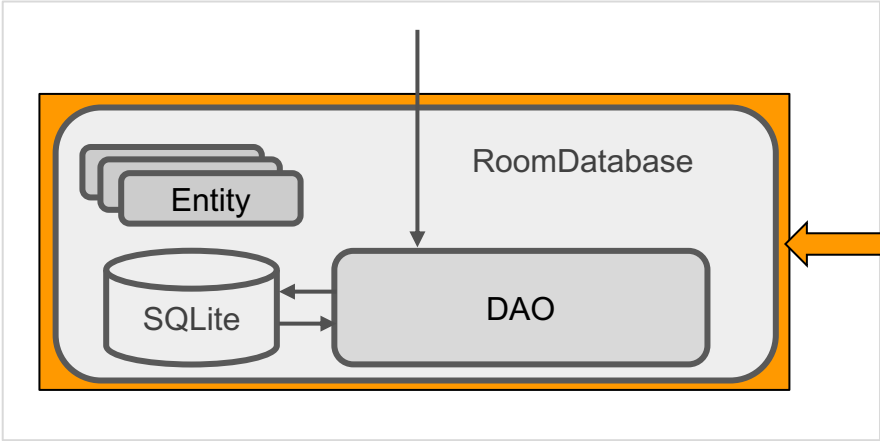
Example queries

```
@Query("DELETE FROM word_table")
void deleteAll();

@Query("SELECT * from word_table ORDER BY word ASC")
List<Word> getAllWords();

@Query("SELECT * FROM word_table WHERE word LIKE :word ")
public List<Word> findWord(String word);
```

Room database

Room

- Room works with DAO and Entities
- Entities define the database schema
- DAO provides methods to access database



Creating Room database

- Create public abstract class extending `RoomDatabase`
- Annotate as `@Database`
- Declare entities for database schema and set version number

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase extends RoomDatabase
```

Room class example

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase
                      extends RoomDatabase {

    public abstract WordDao wordDao();

    private static WordRoomDatabase INSTANCE;

    // ... create instance here
}
```

Entity defines DB schema

DAO for database

Create database as singleton instance

Use Database builder

- Use Room's database builder to create the database
- Create DB as singleton instance

```
private static WordRoomDatabase INSTANCE;

INSTANCE = Room.databaseBuilder(...)
    .build();
```

Specify database class and name

- Specify `Room` database class and database name

```
INSTANCE = Room.databaseBuilder(
        context,
        WordRoomDatabase.class, "word_database")
        //...
        .build();
```

Specify onOpen callback

- Specify onOpen callback

```
INSTANCE = Room.databaseBuilder(
        context,
        WordRoomDatabase.class, "word_database")
        .addCallback(sOnOpenCallback)
        //...
        .build();
```

Room database creation example

```
static WordRoomDatabase getDatabase(final Context context) {
  if (INSTANCE == null) {
    synchronized (WordRoomDatabase.class) {
      if (INSTANCE == null) {
        INSTANCE = Room.databaseBuilder(
           context.getApplicationContext(),
           WordRoomDatabase.class, "word_database")
           .addCallback(sOnOpenCallback)
           .build();
      }}}
  return INSTANCE;
}
```

Check if database exists before creating it

Initialize DB in onOpen callback

```
private static RoomDatabase.Callback sOnOpenCallback =
  new RoomDatabase.Callback(){
   @Override
   public void onOpen (@NonNull SupportSQLiteDatabase db){
      super.onOpen(db);
      initializeData();
   }};
```

Room caveats

- Compile-time checks of SQLite statements
- Do not run database operations on the main thread
- `LiveData` automatically runs query asynchronously on a background thread when needed
- Usually, make your `RoomDatabase` a singleton

Demo 1: Room DB

```
dependencies {
    implementation "androidx.room:room-runtime:2.4.0"
    annotationProcessor "androidx.room:room-compiler:2.4.0”
    ...
}
```

Demo 1: Entity class

```
@Entity(tableName = "users")
public class User {
    @PrimaryKey(autoGenerate = true)
    public int id;

    @ColumnInfo(name = "name")
    public String name;

    @ColumnInfo(name = "email")
    public String email;
}
```

Demo 1: DAO class

```
public interface UserDAO {
    @Query("SELECT * FROM users")
    List<User> getAll();

    @Insert
    void insert(User user);

    @Update
    void update(User user);

    @Delete
    void delete(User user);
}
```

Demo 1: RoomDatabase class

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

Demo 1: RoomDatabase class

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
UserDAO userDao = db.userDao();
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(() -> {
    List<User> users = userDao.getAll();
    userList.clear();
    userList.addAll(users);
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            adapter.notifyDataSetChanged();
        }
    });
});
```

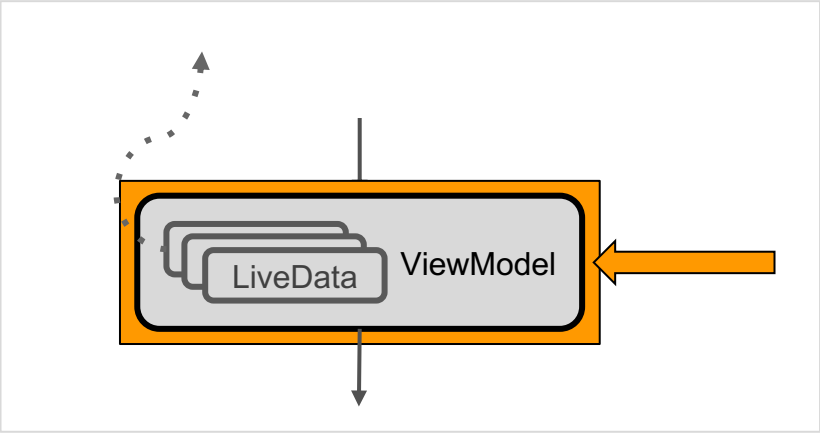
Demo 1: RoomDatabase class

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

User user = new User();
user.name = "John Doe";
user.email = "johndoe@example.com";

Executor executor = Executors.newSingleThreadExecutor();
executor.execute(() -> {
    userDao.insert(user);
});
```

ViewModel

ViewModel

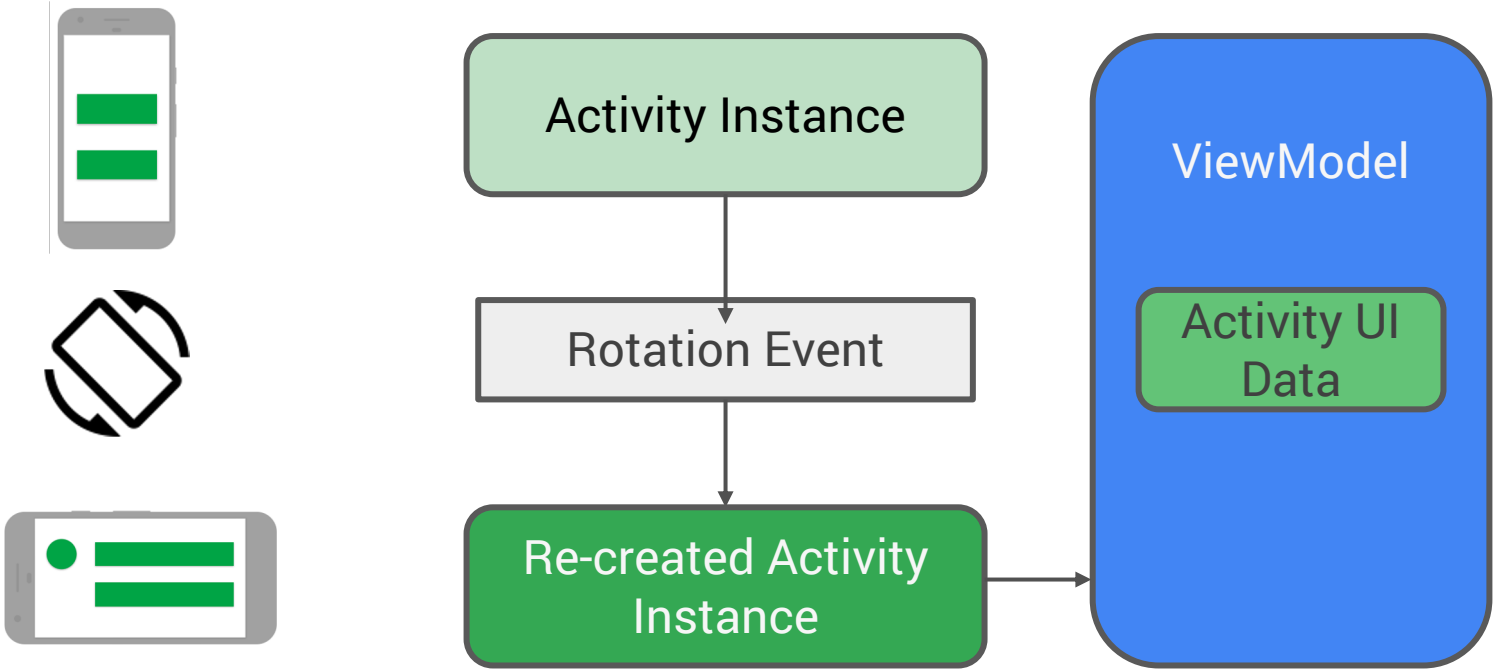
- View models are objects that provide data for UI components and **survive** configuration changes.



ViewModel

- Provides data to the UI
- Survives configuration changes
- You can also use a `ViewModel` to share data between fragments
- Part of the lifecycle library

Survives configuration changes

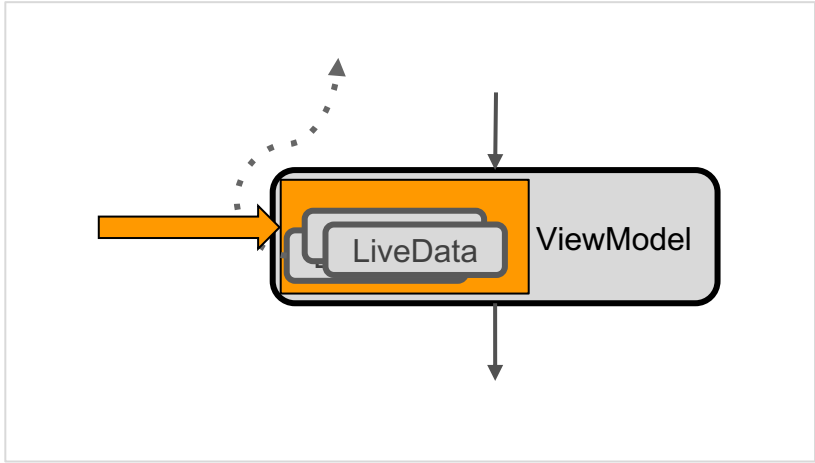


LiveData

LiveData

`LiveData` is a data holder class that is aware of lifecycle events. It keeps a value and allows this value to be observed.

Use `LiveData` to keep your UI up to date with the latest and greatest data.



Creating LiveData

To make data observable, return it as `LiveData`:

```
@Query("SELECT * from word_table")
LiveData<List<Word>> getAllWords();
```

Passing LiveData through layers

When you pass live data through the layers of your app architecture, from a Room database to your UI, that data must be `LiveData` in all layers:

- DAO
- `ViewModel`
- Repository

Passing LiveData through layers

- DAO:

```
@Query("SELECT * from word_table")
LiveData<List<Word>> getAllWords();
```

- Repository:

```
LiveData<List<Word>> mAllWords =
                    mWordDao.getAllWords();
```

- ViewModel:

```
LiveData<List<Word>> mAllWords =
                    mRepository.getAllWords();
```

Observing LiveData

- Create the observer in `onCreate()` in the `Activity`
- Override `onChanged()` in the observer to update the UI when the data changes

  When the LiveData changes, the observer is notified and its `onChanged()` is executed

Demo 2: Room DB with LiveData

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

    @Insert
    void insert(User user);

    @Update
    void update(User user);

    @Delete
    void delete(User user);
}
```

Demo 2: Room DB with LiveData

```
public class UserViewModel extends AndroidViewModel {
    private LiveData<List<User>> users;
    private UserDao userDao;

    public UserViewModel(Application application) {
        super(application);
        AppDatabase database = AppDatabase.getInstance(this.getApplication());
        userDao = database.userDao();
        users = userDao.getAll();
    }

    public LiveData<List<User>> getUsers() {}

    public void insert(User user) {}
}
```

Demo 2: Room DB with LiveData

```
public LiveData<List<User>> getUsers() {
    return users;
}

public void insert(User user) {
    Executor executor = Executors.newSingleThreadExecutor();
    executor.execute(new Runnable() {
        @Override
        public void run() {
            userDao.insert(user);
        }
    });
}
```

Demo 2: Room DB with LiveData

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

    public abstract UserDao userDao();

    public static synchronized AppDatabase getInstance(Context context) {
        if (INSTANCE == null) {
            INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                                AppDatabase.class, "app-database")
                    .build();
        }
        return INSTANCE;
    }
}
```

Demo 2: Room DB with LiveData

```
private UserViewModel userViewModel;
private List<User> userList = new ArrayList<>();
private ArrayAdapter<User> adapter;

userViewModel = new ViewModelProvider(this).get(UserViewModel.class);
userViewModel.getUsers().observe(this, new Observer<List<User>>() {
    @Override
    public void onChanged(@Nullable List<User> users) {
        userList.clear();
        userList.addAll(users);
        adapter.notifyDataSetChanged();
    }
});
```

Demo 2: Room DB with LiveData

```
User user = new User();
user.email = "tes@gmail.com";
user.name = "test";

userViewModel.insert(user);
```

What's next?

- Concept chapter: 10.1 Room, LiveData, and ViewModel
- Practical: 10.1A : Room, LiveData, and ViewModel
- Practical: 10.1B : Room, LiveData, and ViewModel

END