

Advanced Android Development V2

Storing data with Room

Lesson 10



10.1 Room, LiveData, and ViewModel



Contents

- Architecture Components
- Entity
- DAO
- Room database
- ViewModel
- Repository
- LiveData
- Lifecycle

Architecture Components

Architecture Components

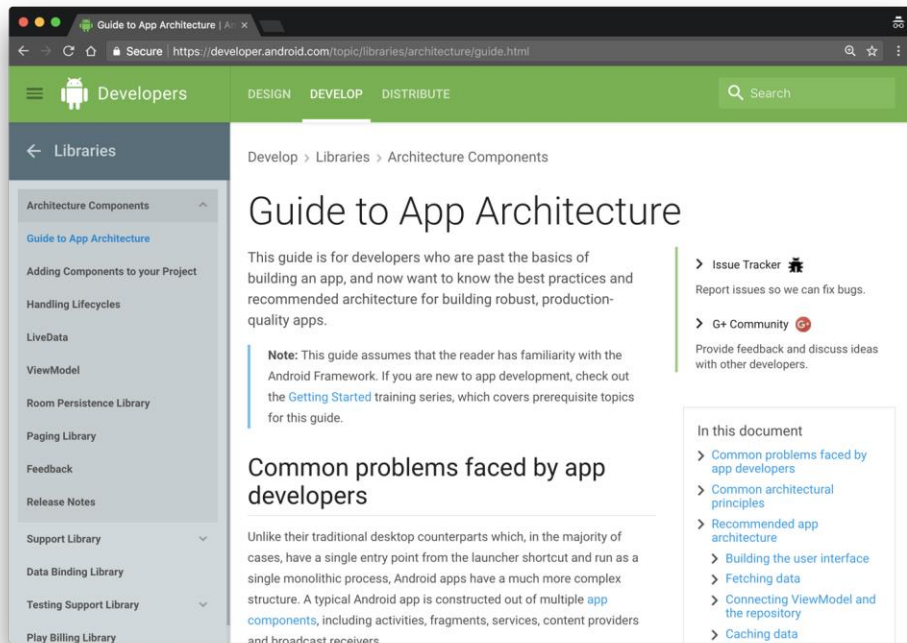
A set of Android libraries for structuring your app in a way that is robust, testable, and maintainable.



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

Guide to app architecture

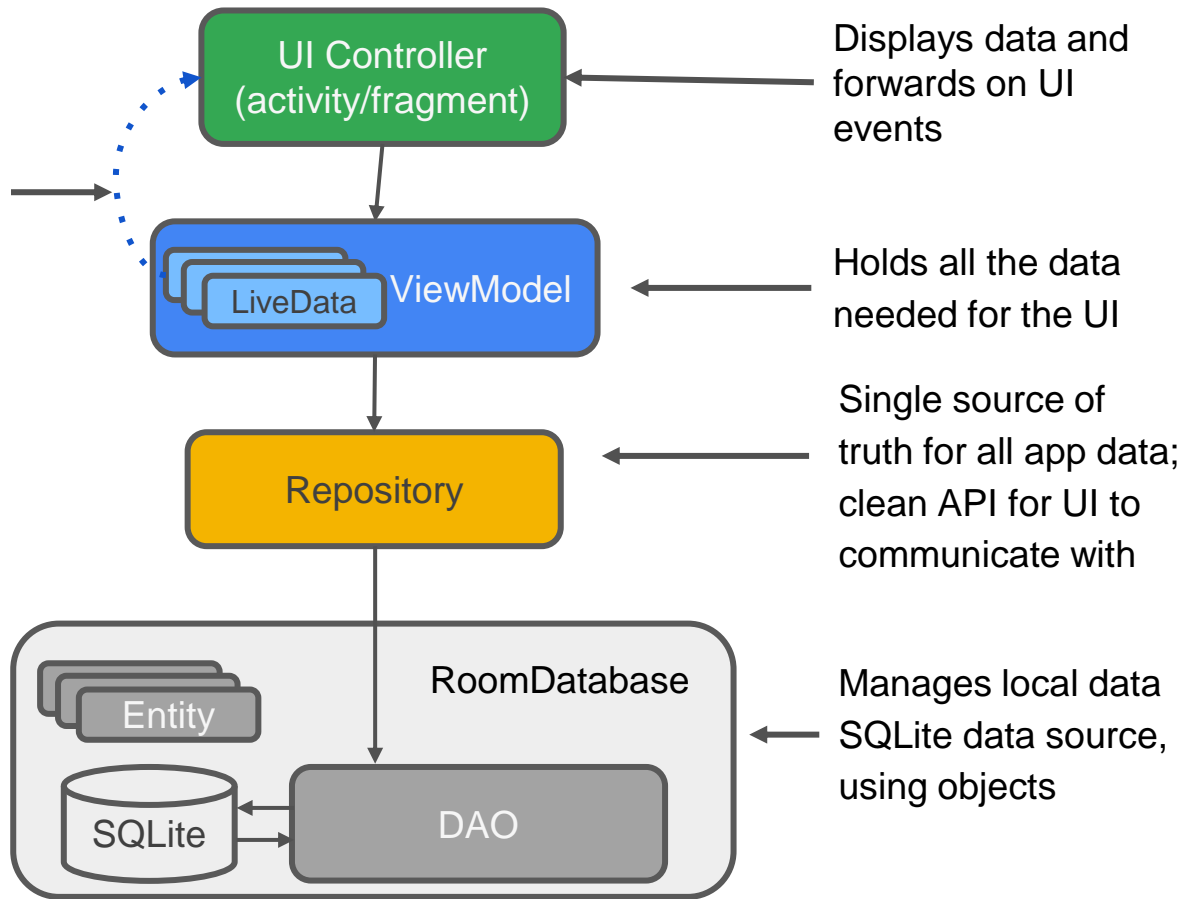


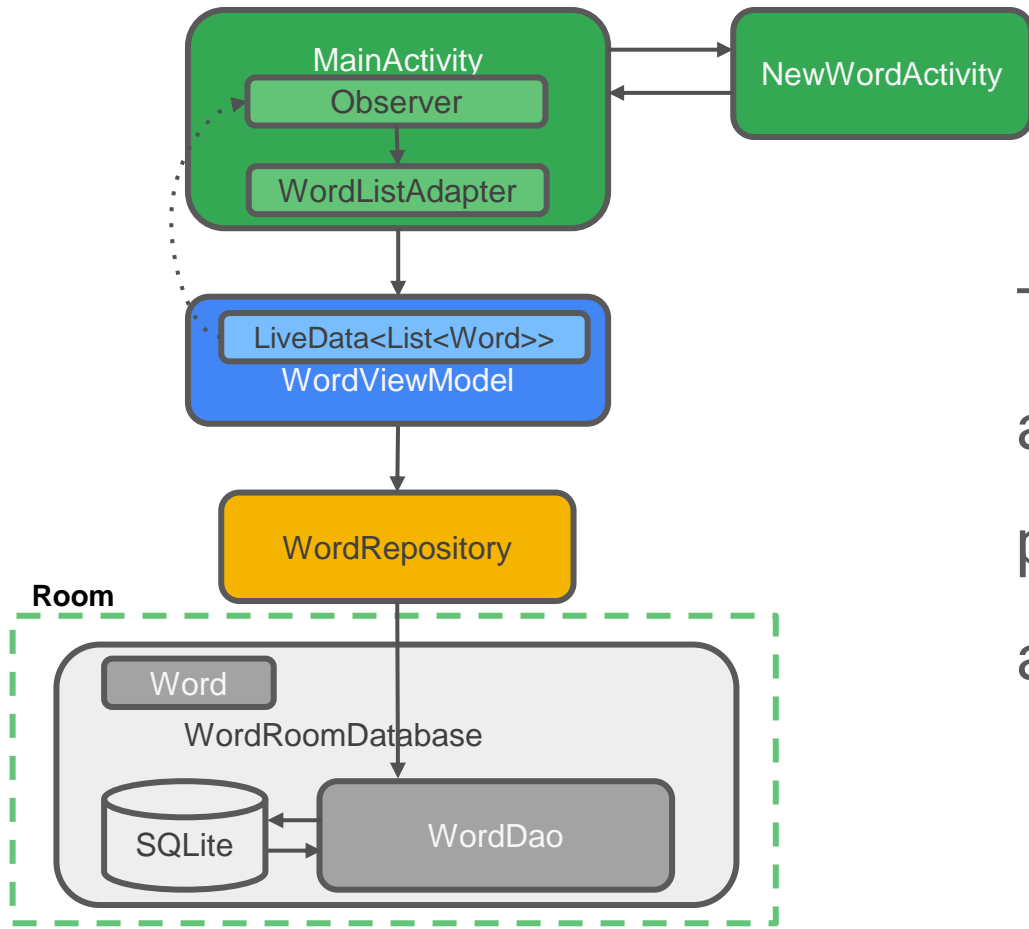
developer.android.com/architecture

[h](#)

Overview

UI is notified of changes using observation





The RoomWordsSample app that you build in the practical implements this architecture

Room overview

Room overview

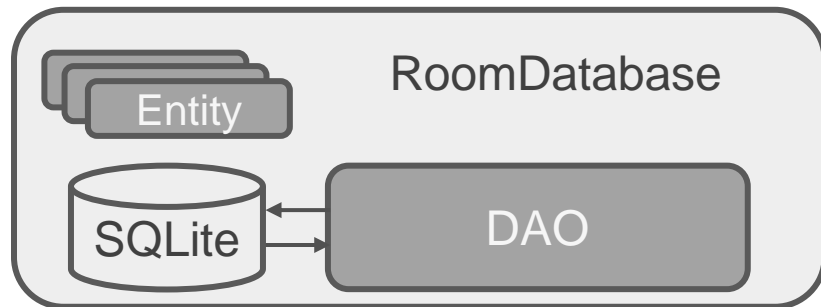
Room is a robust SQL object mapping library

- Generates SQLite Android code
- Provides a simple API for your database



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

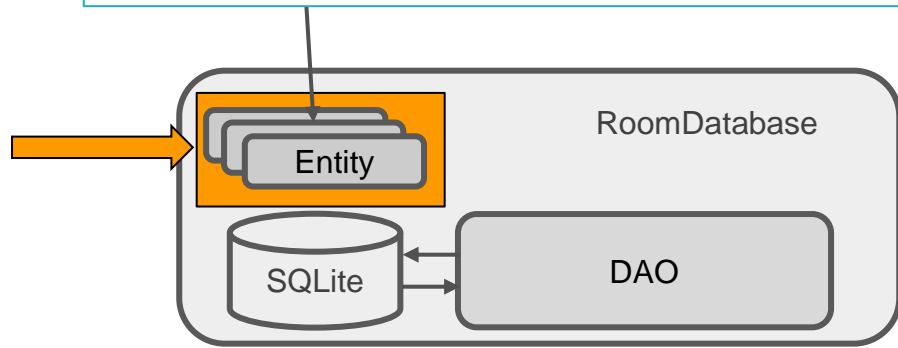


Entity

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

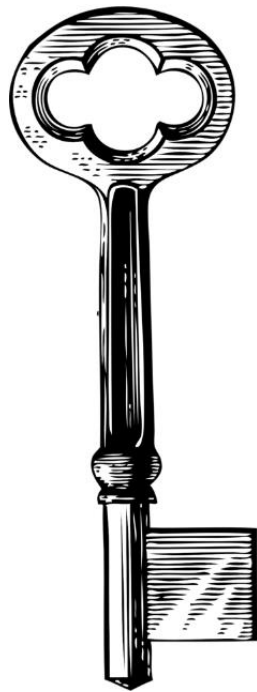
```
@Entity(tableName = "word_table")
```

- 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

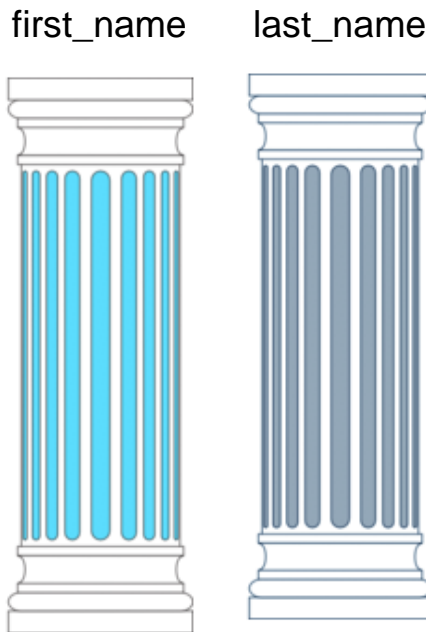


@ColumnInfo annotation

```
@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

users table

id	
name	
pet	

pets table

id	
name	
owner	



Many more annotations

For more annotations, see
[Room package summary
reference](#)

^ android.arch.persistence.room

Overview

^ Annotations

ColumnInfo

ColumnInfo.Collate

ColumnInfo.SQLiteTypeAffinity

Dao

Database

Delete

Embedded

Entity

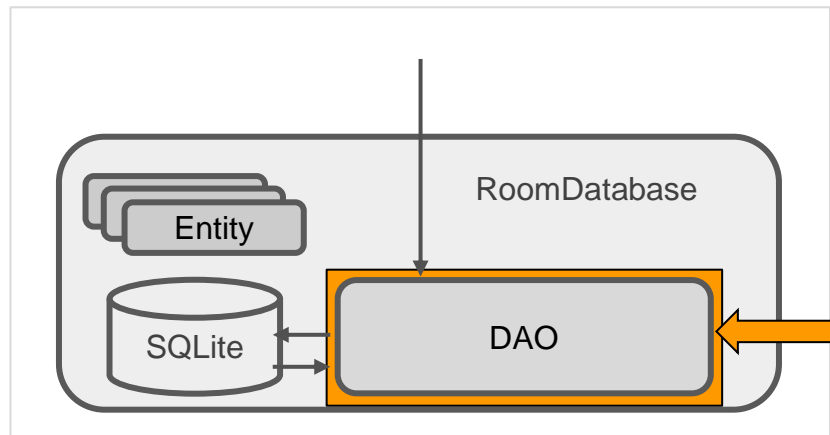
ForeignKey

ForeignKey.Action

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

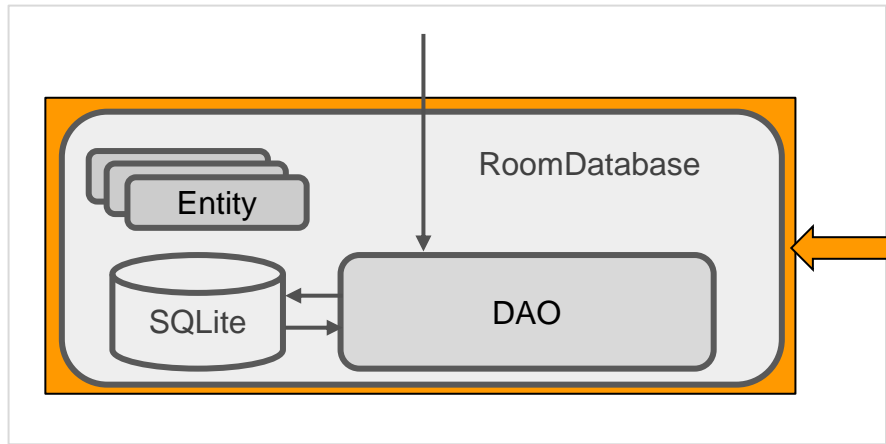
Room

- Room is a robust SQL object mapping library
- Generates SQLite Android code



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)
```

*Entity defines
DB schema*

```
public abstract class WordRoomDatabase  
    extends RoomDatabase {
```

*DAO for
database*

```
    public abstract WordDao wordDao();
```

```
    private static WordRoomDatabase INSTANCE;
```

*Create
database as
singleton
instance*

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

```
}
```



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();
```

Specify migration strategy

- Specify migration strategy callback

```
INSTANCE = Room.databaseBuilder(  
    context.getApplicationContext(),  
    WordRoomDatabase.class, "word_database")  
    .addCallback(sOnOpenCallback)  
    .fallbackToDestructiveMigration()  
    .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)  
                        .fallbackToDestructiveMigration()  
                        .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();  
        }  
    };
```

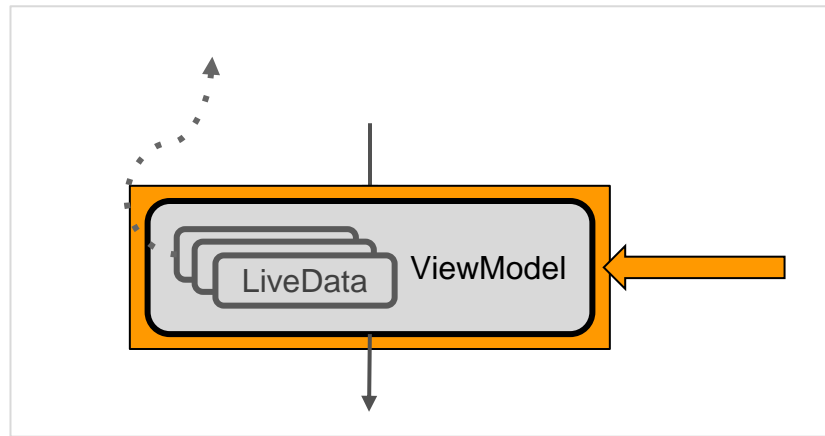
Room caveats

- 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](#)

ViewModel

ViewModel

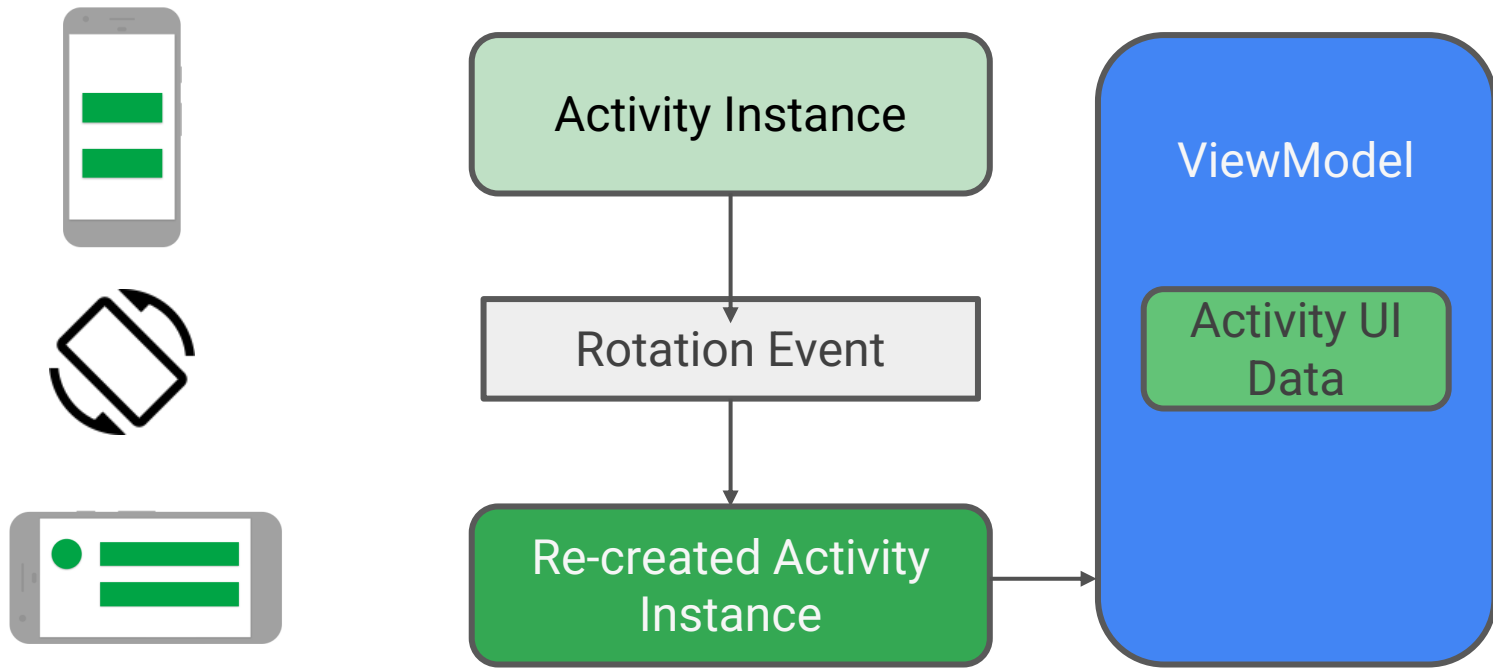
- View models are objects that provide data for UI components and survive configuration changes.



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



ViewModel serves data

- ViewModel serves data to the UI
- Data can come from Room database or other sources
- ViewModel's role is to return the data, it can get help to find or generate the data



Best practice to use repository





Recommended best practice:

- Use a repository to do the work to get the data
- Keeps `ViewModel` as clean interface between app and data



Repository is discussed in next section

Restaurant analogy

- Customer requests meal from server 
- Server takes order to chefs 
- Chefs prepare meal 
- Server delivers meal to customer 
- UI requests data from ViewModel
- ViewModel asks Repository for data
- Repository gets data
- ViewModel returns data to UI

ViewModel example using repository

```
public class WordViewModel extends AndroidViewModel {  
    private WordRepository mRepository;  
    private LiveData<List<Word>> mAllWords;  
  
    // Initialize the repository and the list of words  
    public WordViewModel (Application application) {  
        super(application);  
        mRepository = new WordRepository(application);  
        mAllWords = mRepository.getAllWords();  
    }  
}
```


ViewModel example continued

```
LiveData<List<Word>> getAllWords() {  
    return mAllWords;  
}  
  
public void insert(Word word) {  
    mRepository.insert(word);  
}  
  
public void deleteWord(Word word) {  
    mRepository.deleteWord(word);  
}
```

Do not pass context into ViewModel

- Never pass context into `ViewModel` instances
- Do not store `Activity`, `Fragment`, or `View` instances or their `Context` in the `ViewModel`
- An `Activity` can be destroyed and created many times during the lifecycle of a `ViewModel`
- If you need application context, inherit from [`AndroidViewModel`](#)

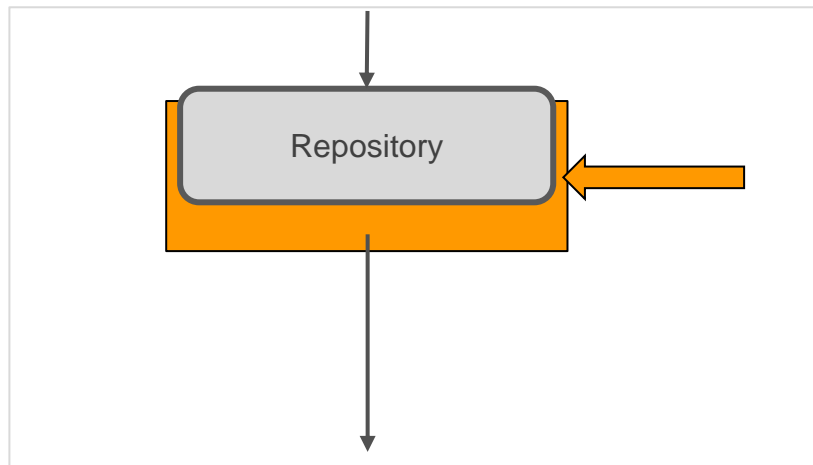
ViewModel does not survive app closure

- ViewModel survives configuration changes, **not** app shutdown
- ViewModel is **not** a replacement for `onSaveInstanceState()` (if you are not saving the data with Room)
- See [Saving UI States](#)

Repository

Repository

- Best practice, not part of Architecture Components libraries
- Implement repository to provide single, clean API to app data

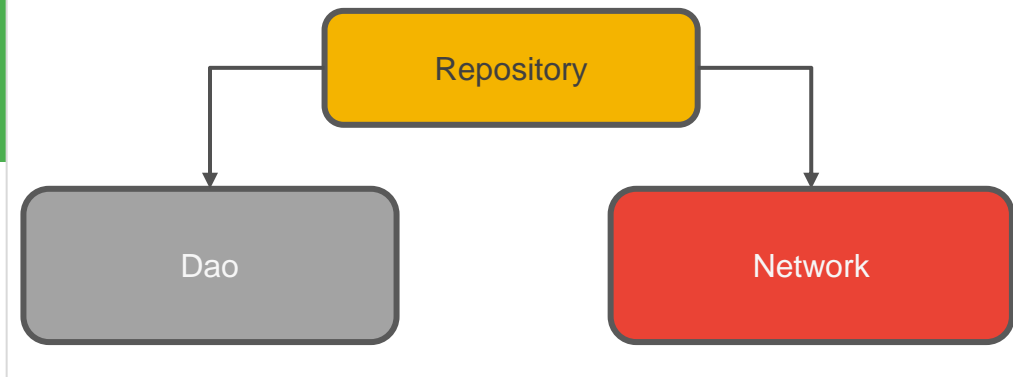


Repository fetches or generates data

- Use repository to fetch data in the background
- Analogy: chefs prepare meals behind the scenes



Multiple backends



- Potentially, repository could manage query threads and allow you to use multiple backends
- Example: in Repository, implement logic for deciding whether to fetch data from a network or use results cached in the database

Repository example

```
public class WordRepository {  
  
    private WordDao mWordDao;  
    private LiveData<List<Word>> mAllWords;  
  
    WordRepository(Application application) {  
        WordRoomDatabase db = WordRoomDatabase.getDatabase(application);  
        mWordDao = db.wordDao();  
        mAllWords = mWordDao.getAllWords();  
    }  
    [... more code...]  
}
```


Get and insert data

```
LiveData<List<Word>> getAllWords() {  
    return mAllWords;  
}
```

```
// Must insert data off the main thread  
public void insert (Word word) {  
    new insertAsyncTask(mWordDao).execute(word);  
}
```

```
[... more code...]  
}
```

Insert off main thread

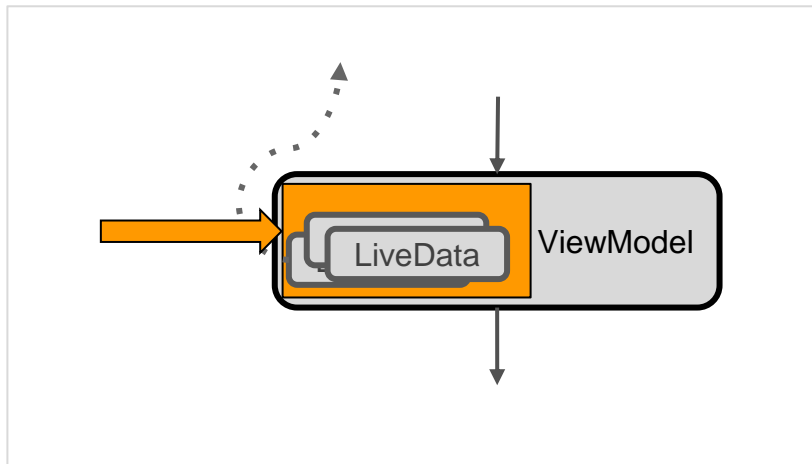
```
private static class insertAsyncTask extends AsyncTask
    <Word, Void, Void> {
    private WordDao mAsyncTaskDao;
    insertAsyncTask(WordDao dao) {
        mAsyncTaskDao = dao;
    }
    @Override
    protected Void doInBackground(final Word... params) {
        mAsyncTaskDao.insert(params[0]);
        return null;
    }
}
```

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.



LiveData

- LiveData is observable data
- Notifies observer when data changes
- Is lifecycle aware: knows when device rotates or app stops



Use LiveData to keep UI up to date

- Create an observer that observes the LiveData
- LiveData notifies Observer objects when the observed data changes
- Your observer can update the UI every time the data changes



Creating LiveData

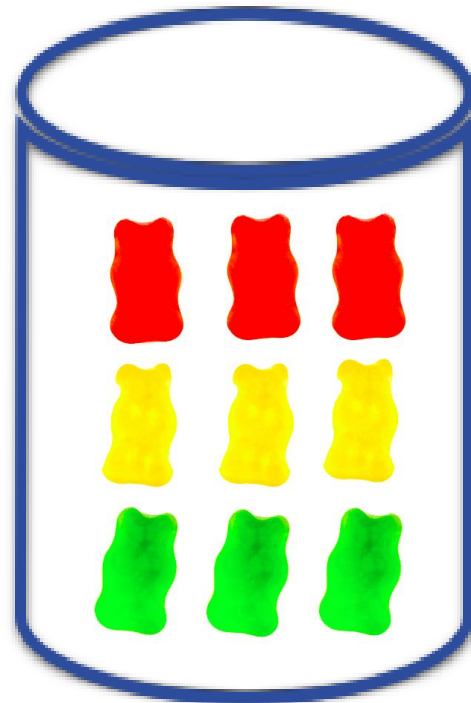
To make data observable, return it as LiveData:

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

Using LiveData with Room



Room generates all the code to update the LiveData when the database is updated



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

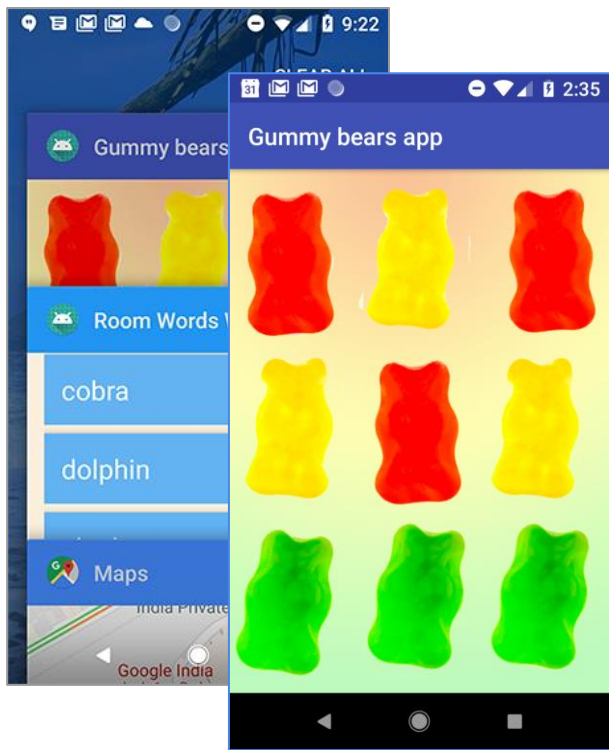
Observing LiveData: example

```
final Observer<String> nameObserver =  
    new Observer<String>() {  
        @Override  
        public void onChanged(@Nullable final String newName) {  
            // Update the UI, in this case, a TextView.  
            mNameTextView.setText(newName);  
        }  
    };  
  
mModel.getCurrentName().observe(this, nameObserver);
```

No memory leaks

- Observers are bound to [Lifecycle](#) objects which are objects that have an Android Lifecycle
- Observers clean up after themselves when their associated lifecycle is destroyed

LiveData is always up to date



- If a lifecycle object becomes inactive, it gets the latest data when it becomes active again
- Example: an activity in the background gets the latest data right after it returns to the foreground

LiveData handles configuration changes



If an activity or fragment is re-created due to a configuration change such as device rotation, the activity or fragment immediately receives the latest available data



Share resources

- You can extend a LiveData object using the [singleton](#) pattern, for example for services or a database
- The LiveData object connects to the system service once, and then any observer that needs the resource can just watch the LiveData object
- See [Extend LiveData](#)

Lifecycle

Lifecycle-aware components

Instead of managing lifecycle-dependent components in the activity's lifecycle methods, `onStart()`, `onStop()`, and so on, you can make any class react to lifecycle events

Lifecycle-aware components

- Lifecycle-aware components perform actions in response to a change in the lifecycle status of another component
- For example, a listener could start and stop itself in response to an activity starting and stopping

Use cases

- Switch between coarse and fine-grained location updates depending on app visibility
- Stop and start video buffering
- Stop network connectivity when app is in background
- Pause and resume animated drawables

Lifecycle library

- Import the [android.arch.lifecycle](#) package
- Provides classes and interfaces that let you build lifecycle-aware components that automatically adjust their behavior based on lifecycle state of activity or fragment
- See [Handling Lifecycles with Lifecycle-Aware Components](#)

LifecycleObserver interface

- [LifecycleObserver](#) has an Android lifecycle.
- It does not have any methods, instead, uses [OnLifecycleEvent](#) annotated methods.

@OnLifecycleEvent

@OnLifecycleEvent indicates life cycle methods

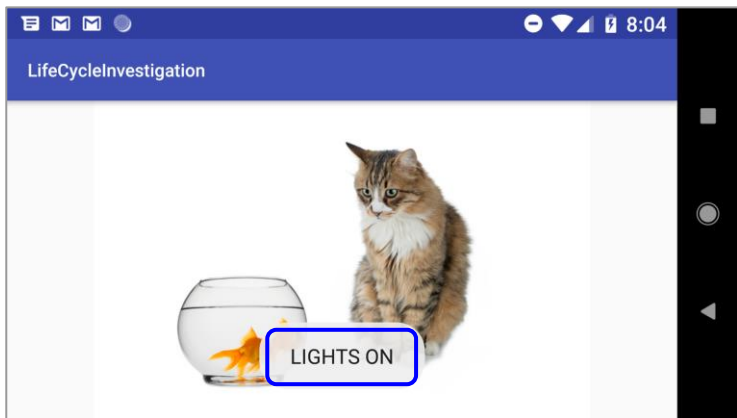
```
@OnLifecycleEvent(Lifecycle.Event.ON_START)  
public void start() {...}
```

```
@OnLifecycleEvent(Lifecycle.Event.ON_STOP)  
public void start() {...}
```

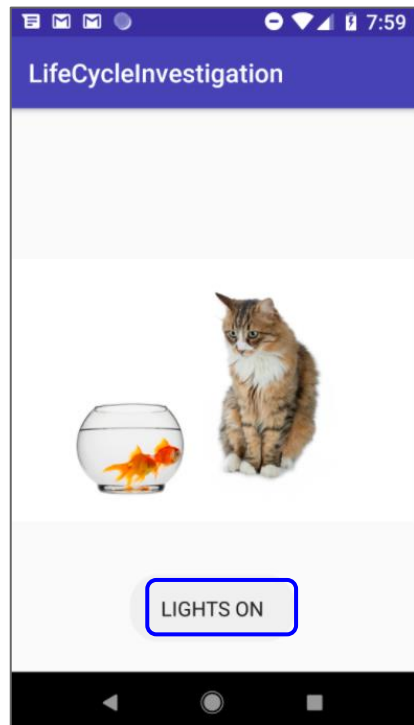
See [Lifecycle.event reference](#) for more lifecycle events

POJOs can be life cycle aware

You can make *any class* react to lifecycle events



*In these pictures,
the Toast is
created by a plain
old Java class
when app starts
or device rotates*



Adding lifecycle awareness to a POJO

```
public class Aquarium {  
  
    // Constructor takes Application and lifecycle  
    public Aquarium(final Application app,  
                    Lifecycle lifecycle) {  
  
        ...  
    }  
}
```

Constructor for lifecycle aware POJO

```
public Aquarium(final Application app, Lifecycle lifecycle) {  
    // Add a new observer to the lifecycle.  
    lifecycle.addObserver(new LifecycleObserver() {  
        @OnLifecycleEvent(Lifecycle.Event.ON_START)  
        public void start() {  
            Toast.makeText(app, "LIGHTS ON", Toast.LENGTH_SHORT).show();  
        }  
    });  
}
```

Creating an instance

```
public class MainActivity extends AppCompatActivity {  
    private Aquarium myAquarium;  
    protected void onCreate(...) {  
        ...  
        // Create aquarium.  
        // Pass context and this activity's lifecycle  
        myAquarium = new Aquarium(this.getApplication(),  
                                   getLifecycle());  
    }  
}
```



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