10.1: Room, LiveData, and ViewModel

Contents:

- What are Android Architecture Components?
- Recommended Architecture Components
- Example app architecture
- Gradle files
- Entity
- The DAO (data access object)
- LiveData
- Room database
- Repository
- ViewModel
- Displaying LiveData
- Lifecycle-aware components
- Paging library
- Summary
- Related practicals
- Learn more

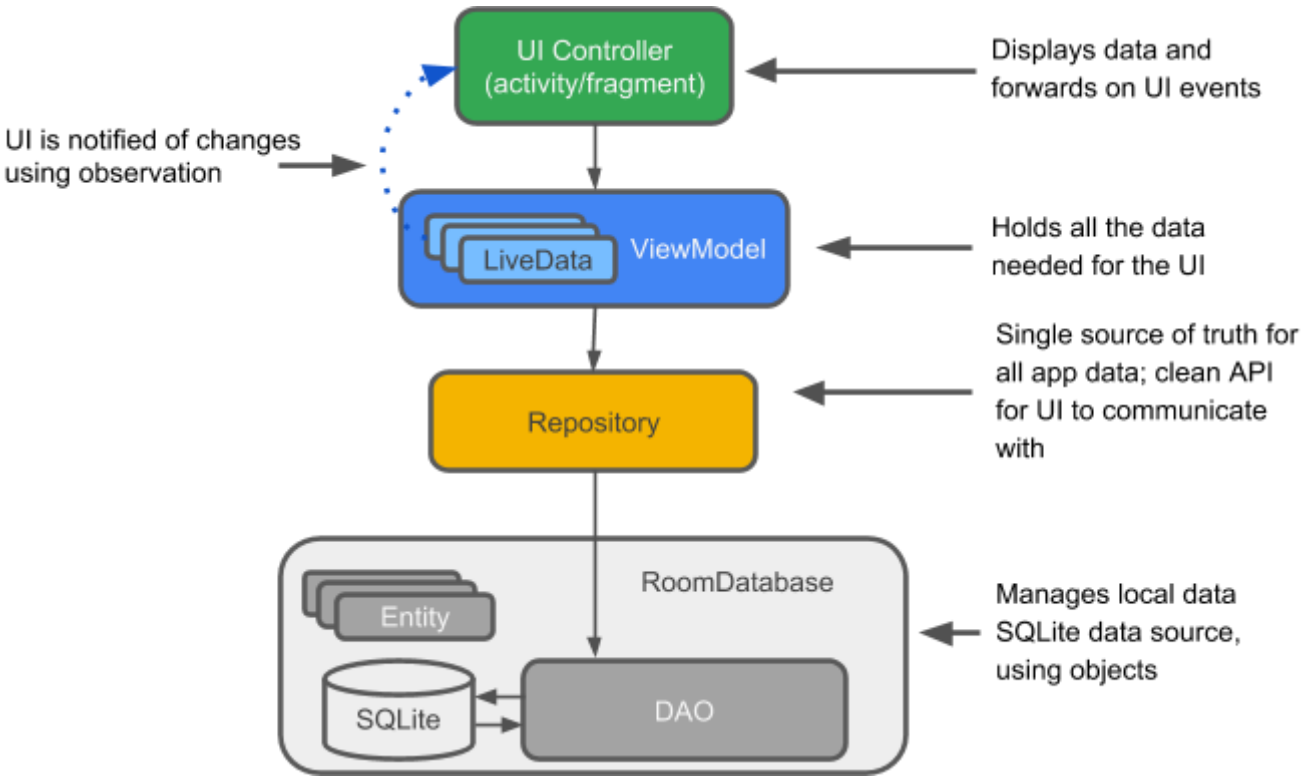
What are Android Architecture Components?

The Android OS manages resources aggressively to perform well on a huge range of devices, and sometimes that makes it challenging to build robust apps. Android Architecture Components provide guidance on app architecture, with libraries for common tasks like lifecycle management and data persistence.

Architecture Components help you structure your app in a way that is robust, testable, and maintainable with less boilerplate code. Architecture Components provide a simple, flexible, and practical approach that frees you from some common problems so you can focus on building great experiences.

Recommended Architecture Components

To introduce the terminology, here is a short introduction to the Architecture Components and how they work together. Each component is explained more in the following sections. This diagram shows a basic form of this architecture.



Entity: When working with Architecture Components, the entity is an annotated class that describes a database table.

SQLite database: On the device, data is stored in an SQLite database. For simplicity, additional storage options, such as a web server, are omitted from this chapter. The Room persistence library creates and maintains this database for you.

DAO: Data access object. A mapping of SQL queries to functions. You used to have to define these queries in a helper class, such as `SQLiteOpenHelper`. When you use a DAO, you call the methods, and the components take care of the rest.

Room database: Database layer on top of SQLite database that takes care of mundane tasks that you used to handle with a helper class, such as `SQLiteOpenHelper`. Provides easier local data storage. The Room database uses the DAO to issue queries to the SQLite database.

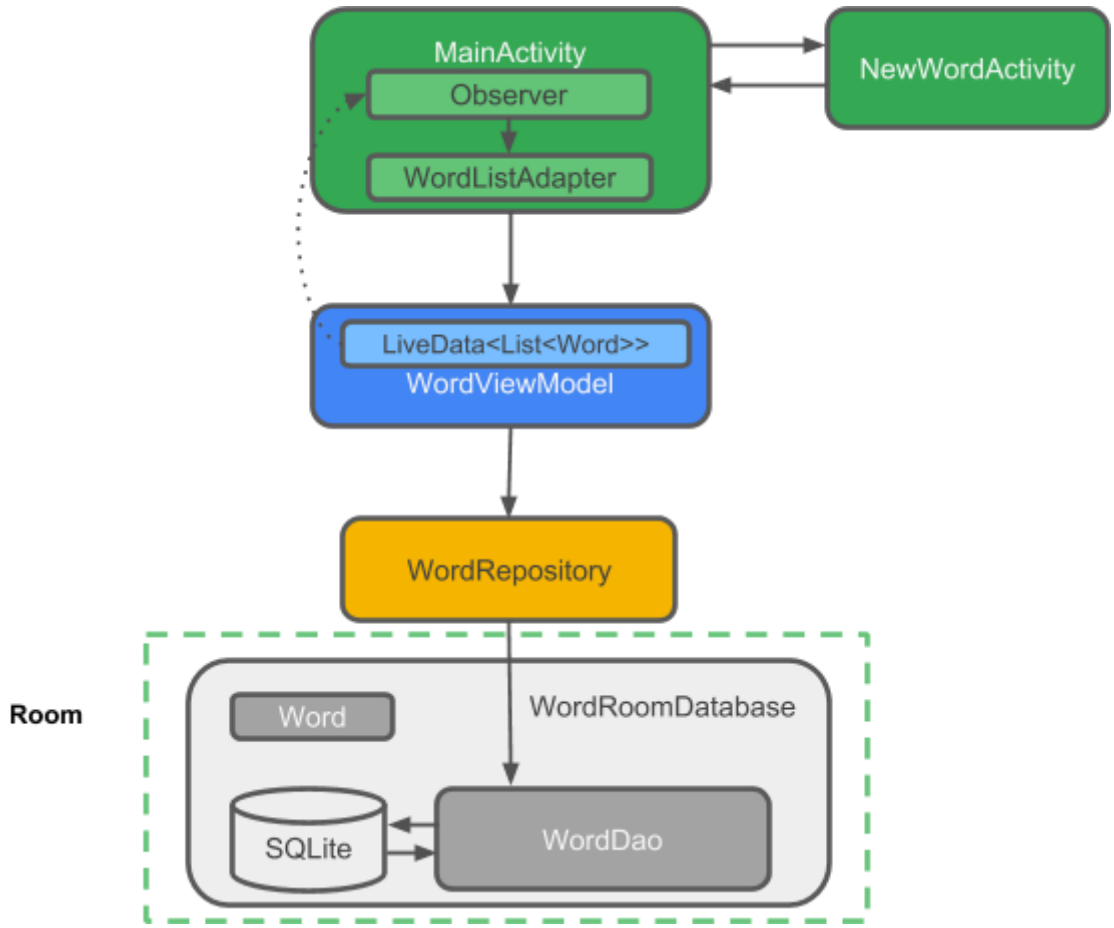
Repository: A class that you create for managing multiple data sources, for example using the `WordRepository` class.

`ViewModel`: Provides data to the UI and acts as a communication center between the Repository and the UI. Hides the backend from the UI. `ViewModel` instances survive device configuration changes.

`LiveData`: A data holder class that follows the observer pattern, which means that it can be observed. Always holds/caches latest version of data. Notifies its observers when the data has changed. `LiveData` is lifecycle aware. UI components observe relevant data. `LiveData` automatically manages stopping and resuming observation, because it's aware of the relevant lifecycle status changes.

Example app architecture

The following diagram shows the same basic architecture form as the diagram above, but in the context of an app. The following sections delve deeper into each component.



Gradle files

To use the Architecture Components libraries, you must manually add the latest version of the libraries to your Gradle files.

Add the following code to your `build.gradle (Module: app)` file, at the end of the dependencies block:

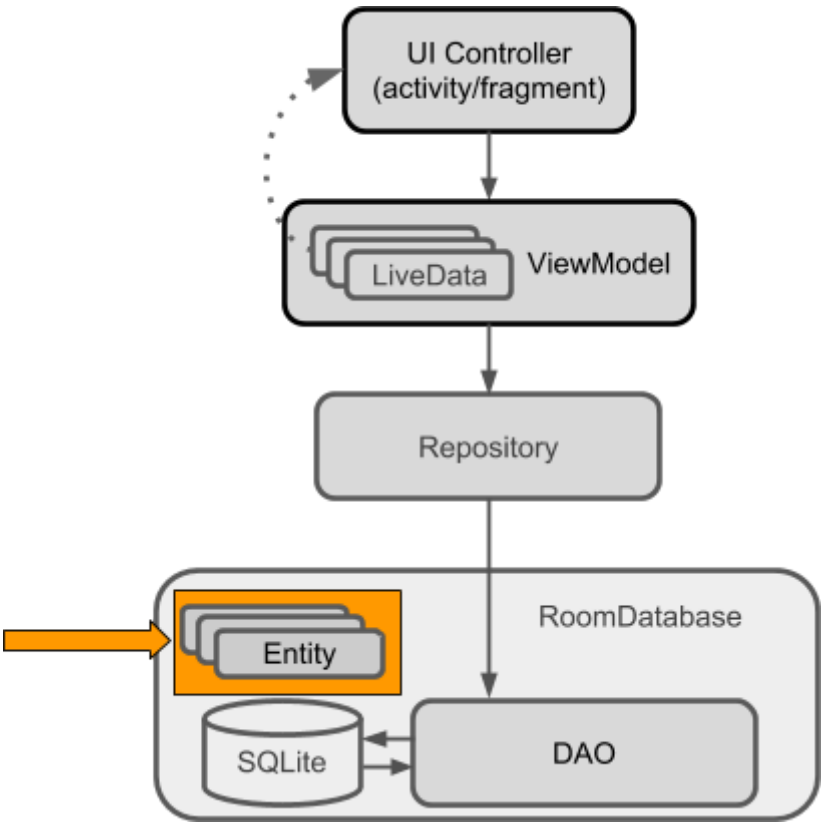
```
// Room components
implementation "android.arch.persistence.room:runtime:$rootProject.roomVersion"
annotationProcessor "android.arch.persistence.room:compiler:$rootProject.roomVersion"
androidTestImplementation "android.arch.persistence.room:testing:$rootProject.roomVersion"

// Lifecycle components
implementation "android.arch.lifecycle:extensions:$rootProject.archLifecycleVersion"
annotationProcessor "android.arch.lifecycle:compiler:$rootProject.archLifecycleVersion"
```

Get the current version numbers from the Adding Components to your Project page. In your `build.gradle (Project: RoomWordsSample)` file, add the version numbers to the end of the file, as shown in the code below:

```
ext {
   roomVersion = '1.0.0'
   archLifecycleVersion = '1.0.0'
}
```

Entity

The data for an app with a database centers around the data stored in the database. Of course, there can be other data from different sources, but for simplicity, this discussion ignores other data.



Room models an SQLite database, and is implemented with an SQLite database as its backend. SQLite databases store their data in tables of rows and columns. Each row represents one entity (or record), and each column represents a value in that entity. For example, an entity for a dictionary entry might include a unique ID, the word, a definition, grammatical information, and links to more info.

Another common entity is for a person, with a unique ID, first name, last name, email address, and demographic information. The entities in such a database might look like this:

ID	First name	Last name	...
12345	Aleks	Becker	...
12346	Jhansi	Kumar	...

When you write an app using Architecture Components (or an SQLite database), you define your entity or entities in classes. Each instance of the class represents a row, and each column is represented by a member variable.

Here is code representing a `Person` entity:

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

Entity annotations

For Room to work with an entity, you need to give Room information that relates the contents of the entity's Java class (for example `Person`) to what you want to represent in the database table. You do this using annotations.

Here are some commonly used annotations:

- `@Entity(tableName = "word_table" )` Each `@Entity` instance represents an entity in a table. Specify the name of the table if you want it to be different from the name of the class.

- `@PrimaryKey` Every entity needs a primary key. To auto-generate a unique key for each entity, add and annotate a primary integer key with `autoGenerate=true`, as shown in the code below. See Defining data using Room entities.
- `@NonNull` Denotes that a parameter, field, or method return value can never be `null`. The primary key should always use the `@NonNull` annotation. Use this annotation for mandatory fields in your rows.
- `@ColumnInfo(name = "word" )` Specify the name of the column in the table, if you want the column name to be different from the name of the member variable.

Every field that's stored in the database must either be public or have a "getter" method so that Room can access it. The code below provides a `getWord()` "getter" method rather than exposing member variables directly.

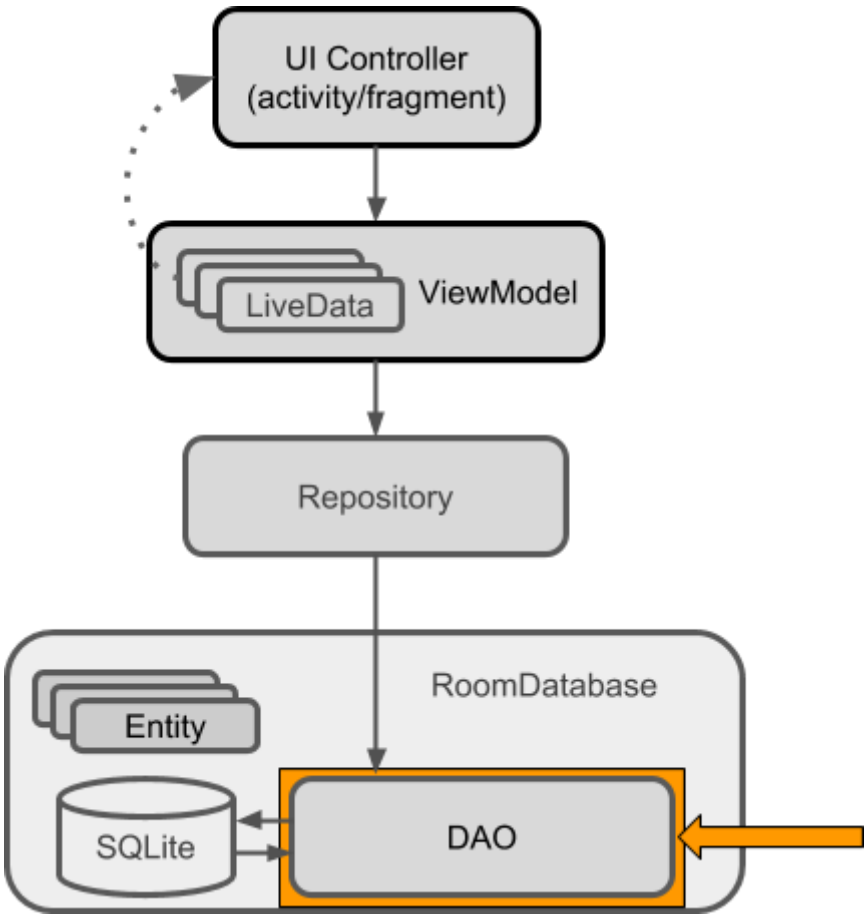
Here is annotated code representing a `Person` entity:

```
@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 are not shown for brevity,
    // but they're required for Room to work if variables are private.
}
```

You can also use annotations to define relationships between entities. For details and a complete list of annotations, see the Room package summary reference.

The DAO (data access object)

To access your app's data using the Room persistence library, you work with *data access objects*, or *DAO*s. A set of Dao objects (DAOs) forms the main component of a Room database. Each DAO includes methods that offer abstract access to your app's database.



You annotate the DAO to specify SQL queries and associate them with method calls. The compiler checks the SQL for errors, then generates queries from the annotations. For common queries, the libraries provide convenience annotations, such as `@Insert`, `@Delete`, and `@Update`.

Note that:

- The DAO must be an `interface` or `abstract` class.
- Room uses the DAO to create a clean API for your code.
- By default, queries (`@Query`) must be executed on a thread other than the main thread. For operations such as inserting or deleting, Room takes care of thread management for you if you use the appropriate annotations.
- The default action when you ask Room to insert an entity that is already in the database is to `ABORT`. You can specify a different conflict strategy, such as `REPLACE`.

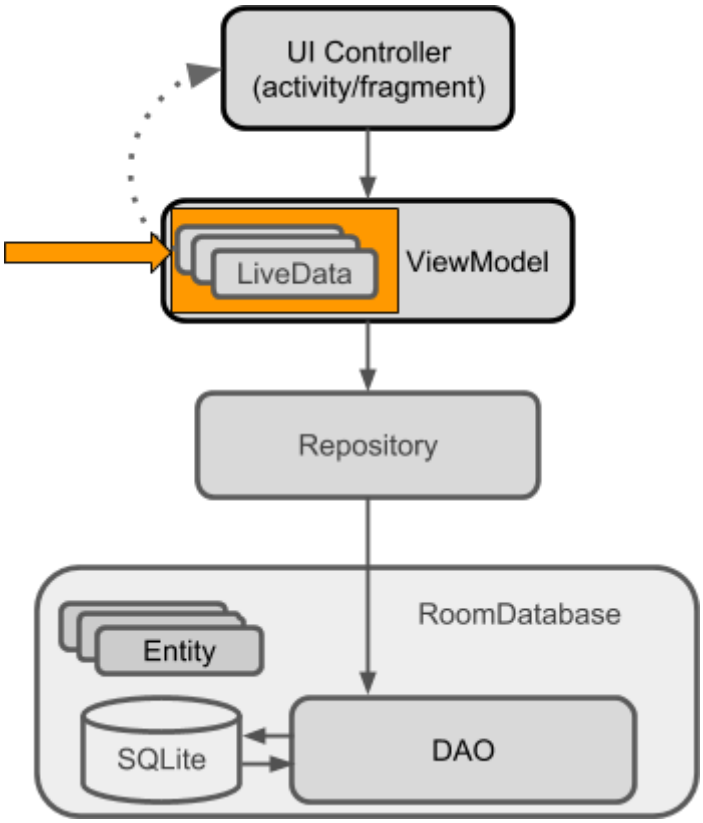
Here is code that demonstrates these annotations along with different types of queries:

```
@Dao
public interface WordDao {

    // The conflict strategy defines what happens,
    // if there is an existing entry.
    // The default action is ABORT.
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insert(Word word);

    // Update multiple entries with one call.
    @Update
    public void updateWords(Word... words);

    // Simple query that does not take parameters and returns nothing.
    @Query("DELETE FROM word_table")
    void deleteAll();

    // Simple query without parameters that returns values.
    @Query("SELECT * from word_table ORDER BY word ASC")
    List<Word> getAllWords();

    // Query with parameter that returns a specific word or words.
    @Query("SELECT * FROM word_table WHERE word LIKE :word ")
    public List<Word> findWord(String word);
}
```

Learn more about Room DAOs.

LiveData

When your app displays data or uses data in other ways, you usually want to take action when the data changes. This means your app has to observe the data so that when it changes, the app can react. Depending on how the data is stored, this can be tricky. Observing changes to data across multiple components of your app can create explicit, rigid dependency paths between the components. This makes testing and debugging, among other things, difficult.



To avoid these problems, use `LiveData`, which is a lifecycle library class for data observation. If you 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.

Benefits of using LiveData

Ensures that your UI matches your data state

`LiveData` follows the observer pattern, so it notifies `Observer` objects when the lifecycle state changes. You can consolidate your code to update the UI within these observer objects. Instead of updating the UI every time the app data changes, your observer can update the UI every time there's a state change.

No memory leaks

Observers are bound to `Lifecycle` objects, and the observers clean up after themselves when their associated lifecycle is destroyed.

No crashes due to stopped activities

If the observer's lifecycle is inactive, as in the case of an activity in the back stack, then it doesn't receive any `LiveData` events.

No more manual lifecycle handling

UI components just observe relevant data and don't stop or resume observation. `LiveData` is aware of relevant lifecycle status changes while observing, so `LiveData` automatically manages stopping and resuming observation.

Data is always up-to-date

If a lifecycle becomes inactive, it receives the latest data upon becoming active again. For example, an activity that was in the background receives the latest data right after it returns to the foreground.

Configuration changes handled properly

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

Resources can be shared

You can extend a `LiveData` object using the singleton pattern. Do this, 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. For more information, see Extend LiveData.

Using LiveData

Follow these general steps to work with `LiveData` objects:

1. Create an instance of `LiveData` to hold a certain type of data. This is usually done within your `ViewModel` class.
2. Create an `Observer` object that defines the `onChanged()` method, which controls what happens when the `LiveData` object's held data changes. You usually create an `Observer` object in a UI controller, such as an activity or fragment.
3. Attach the `Observer` object to the `LiveData` object using the `observe()` method. The `observe()` method takes a `LifecycleOwner` object. This subscribes the `Observer` object to the `LiveData` object so that the observer is notified of changes. You usually attach the `Observer` object in a UI controller, such as an activity or fragment.

Note: You can register an observer without an associated `LifecycleOwner` object using the `observeForever(Observer)` method. In this case, the observer is considered to be always active and is therefore always notified about modifications. To remove these observers, call the `removeObserver(Observer)` method.

When you update the value stored in the `LiveData` object, all registered observers are notified and will take specified actions, such as updating the UI, as long as the attached `LifecycleOwner` is in the active state.

`LiveData` allows UI-controller observers to subscribe to updates. When the data held by the `LiveData` object changes, the UI automatically updates in response.

These steps are explained in more detail in the rest of this chapter.

Making data observable with LiveData

To make data observable, wrap it with `LiveData`. That's it.

For the `Person` example, in the DAO you would wrap the returned data into `LiveData`, as shown in this code:

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

> **Important:** When you pass data through the layers of your app architecture from a Room database to your UI, that data has to be `LiveData` in all layers: All the data that Room returns to the Repository, and the Repository then passes to the `ViewModel`, must be `LiveData`. You can then create an observer in the activity that observes the data in the `ViewModel`.

MutableLiveData

You can use `LiveData` independently from Room, but to do so you must manage data updates. However, `LiveData` has no publicly available methods to update the stored data.

Therefore, if you want to update the stored data, you must use `MutableLiveData` instead of `LiveData`. The `MutableLiveData` class adds two public methods that allow you to set the value of a `LiveData` object: `setValue(T)` and `postValue(T)`.

`MutableLiveData` is usually used in the `ViewModel`, and then the `ViewModel` only exposes immutable `LiveData` objects to the observers.

See the Architecture Components' BasicSample code for examples of using `MutableLiveData`. The example in Guide to App Architecture also shows a use `MutableLiveData`.

Observing LiveData

To update the data that is shown to the user, create an observer of the data in the `onCreate()` method of `MainActivity` and override the observer's `onChanged()` method. When the `LiveData` changes, the observer is notified and `onChanged()` is executed. You then update the cached data, for example in the adapter, and the adapter updates what the user sees.

Usually you observe data in a `ViewModel`, not directly in the Repository or in Room. `ViewModel` is described in a later section.

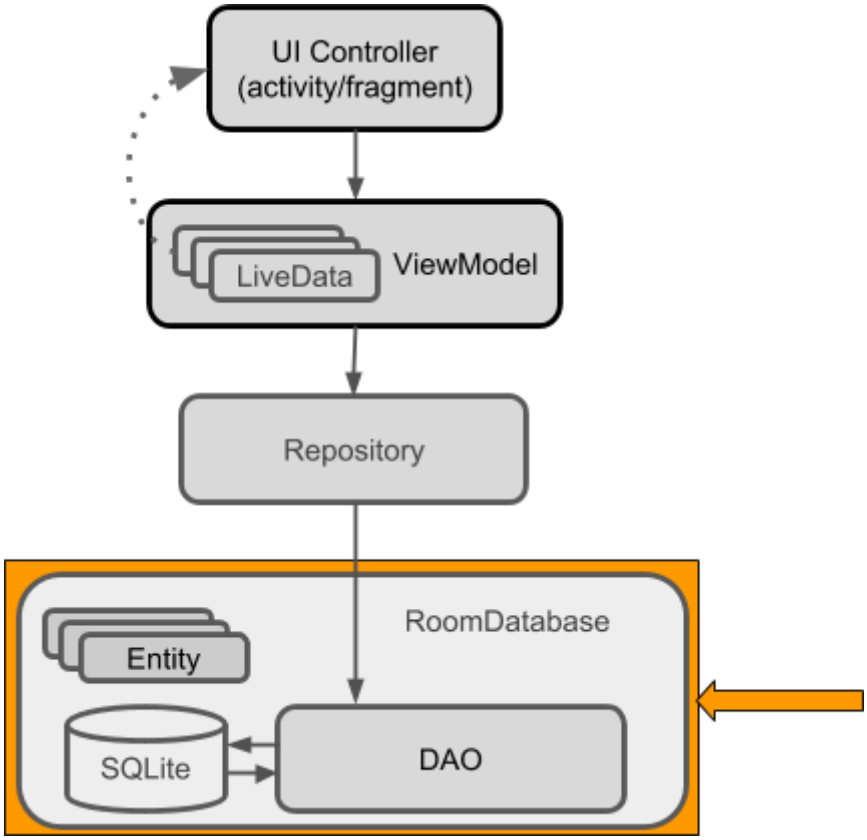
The following code shows how to attach an observer to `LiveData`:

```
// Create the observer which updates the UI.
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);
```

See the `LiveData` documentation to learn other ways to use `LiveData`, or watch this Architecture Components: LiveData and Lifecycle video.

Room database

Room is a database layer on top of an SQLite database. Room takes care of mundane tasks that you used to handle with an `SQLiteOpenHelper`.



To use Room:

1. Create a `public abstract` class that extends `RoomDatabase`.
2. Use annotations to declare the entities for the database and set the version number.
3. Use Room's database builder to create the database if it doesn't exist.
4. Add a migration strategy for the database. When you modify the database schema, you'll need to update the version number and define how to handle migrations. For a sample, destroying and re-creating the database is a fine migration strategy. For a real app, you must implement a migration strategy.
   See Understanding migrations with Room.

Note that:

- Room provides compile-time checks of SQLite statements.
- By default, to avoid poor UI performance, Room doesn't allow you to issue database queries on the main thread. `LiveData` applies this rule by automatically running the query asynchronously on a background thread, when needed.

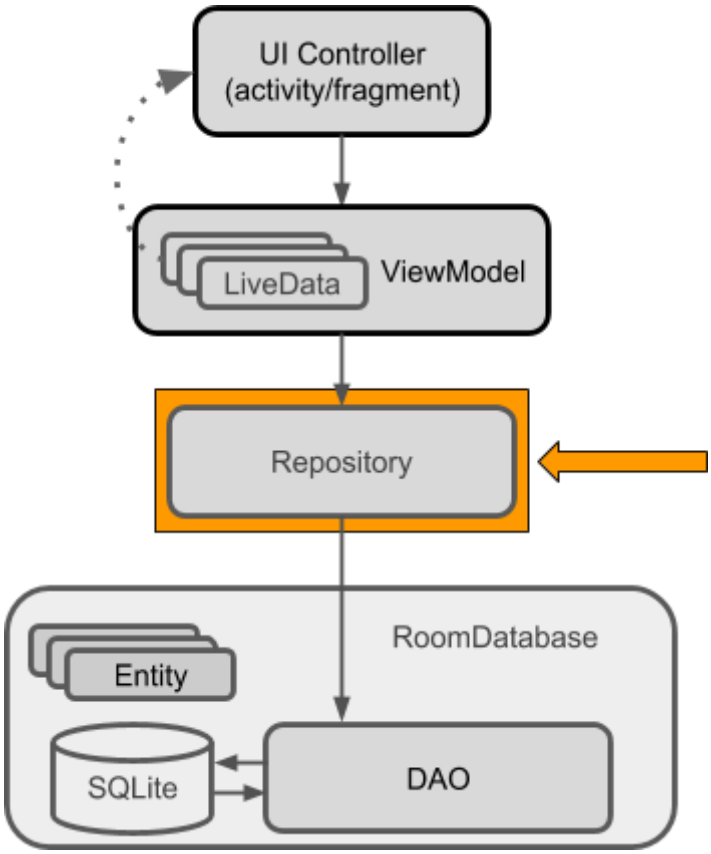
- Usually, you only need one instance of the Room database for the whole app. Make your `RoomDatabase` a singleton to prevent having multiple instances of the database opened at the same time, which would be a bad thing.

Here is a sample of a complete Room class:

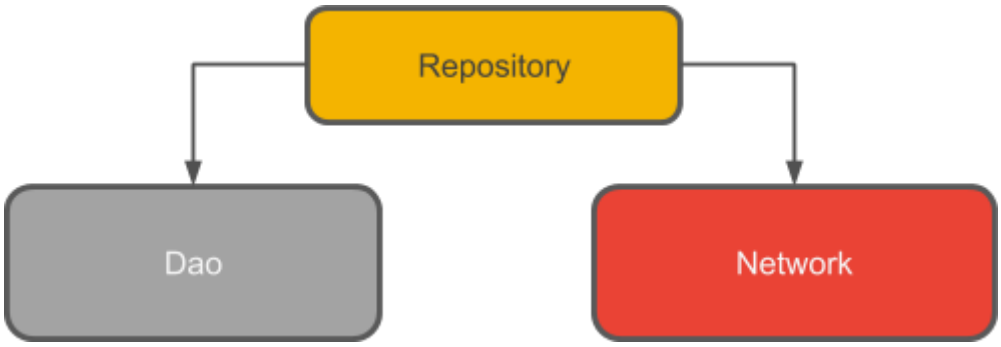
```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase extends RoomDatabase {

    public abstract WordDao wordDao();

    private static WordRoomDatabase INSTANCE;

    static WordRoomDatabase getDatabase(final Context context) {
        if (INSTANCE == null) {
            synchronized (WordRoomDatabase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                            WordRoomDatabase.class, "word_database")
                              // Wipes and rebuilds instead of migrating
                              // if no Migration object.
                            .fallbackToDestructiveMigration()
                            .build();
                }
            }
        }
        return INSTANCE;
    }
}
```

Repository

A *Repository* is a class that 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 handles data operations. It provides a clean API to the rest of the app for app data.



A Repository is where you would put the code to manage query threads and use multiple backends, if appropriate. Once common use for a Repository is to implement the logic for deciding whether to fetch data from a network or use results cached in the database.



Here is the complete code for a basic Repository:

```
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();
    }

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

    public void insert (Word word) {
        new insertAsyncTask(mWordDao).execute(word);
    }

    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;
        }
    }
}
```

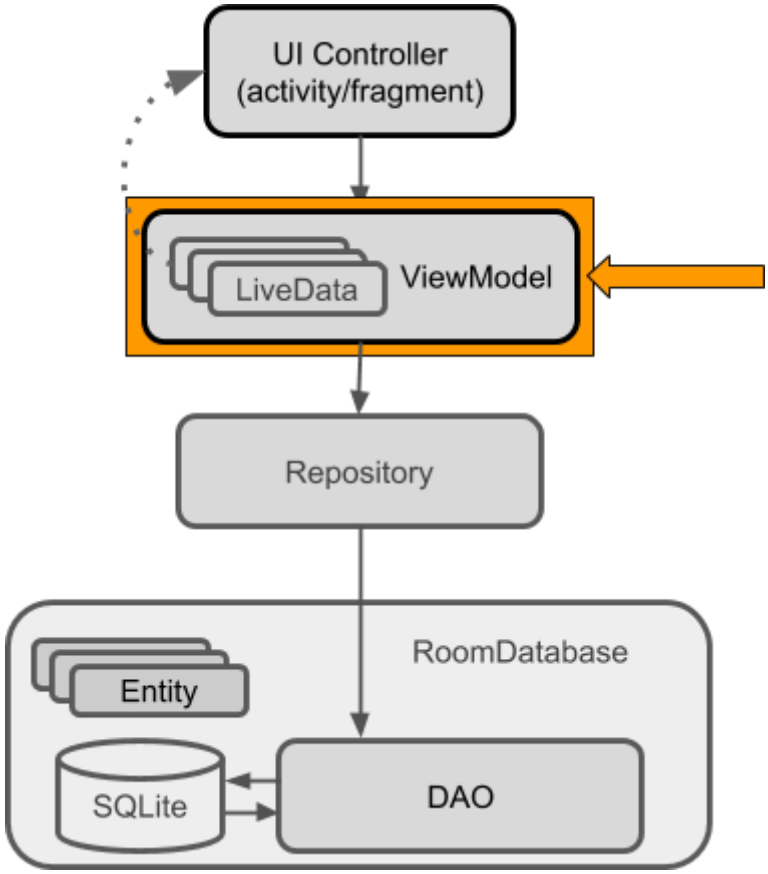
Note: In this example, the Repository doesn't do much. See the BasicSample for an applied implementation.

The Guide to App Architecture includes a more complex example that uses a web service to fetch data.

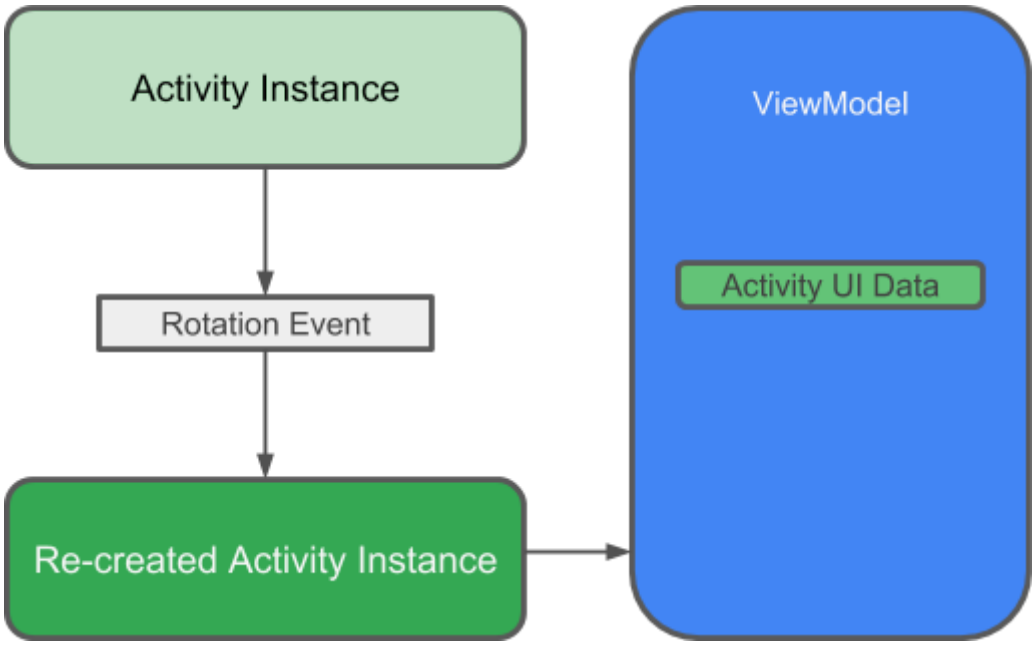
ViewModel

The `ViewModel` is a class whose 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. For an

introductory guide to this topic, see the `ViewModel` overview.



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` is responsible for holding and processing all the data needed for the UI.



Warning: 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`, such as when the device is rotated. If you store a reference to the `Activity` in the `ViewModel`, you end up with references that point to the destroyed `Activity`. This is a memory leak. If you need the application context, use `AndroidViewModel` instead of `ViewModel`.

In the `ViewModel`, use `LiveData` for changeable data that the UI will use or display, so that you can add an observer and respond to changes.

Here is the complete code for a sample `ViewModel`:

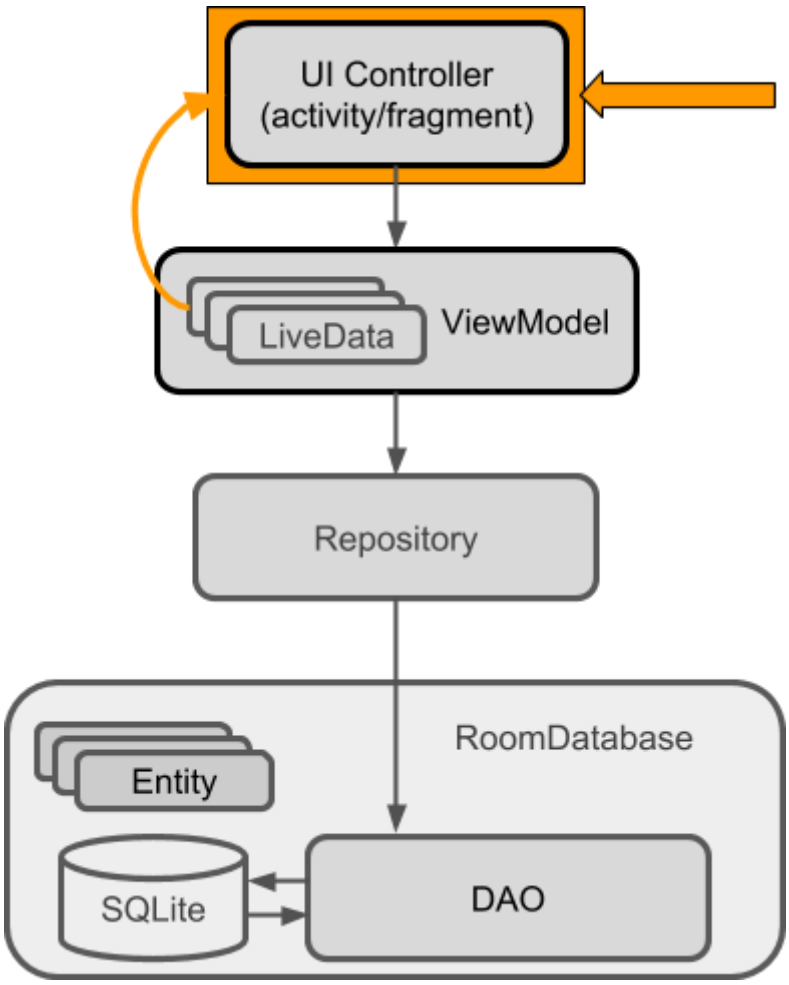
```
public class WordViewModel extends AndroidViewModel {

    private WordRepository mRepository;

    private LiveData<List<Word>> mAllWords;

    public WordViewModel (Application application) {
        super(application);
        mRepository = new WordRepository(application);
        mAllWords = mRepository.getAllWords();
    }

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

    public void insert(Word word) { mRepository.insert(word); }
}
```

> **Important:** `ViewModel` is not a replacement for `onSaveInstanceState()`, because the `ViewModel` does not survive a process shutdown. See Saving UI States.

To learn more, watch this Architecture Components: ViewModel video.

Displaying LiveData

Finally, you can display all this interesting data to the user.



Whenever the data changes, the `onChanged()` method of your observer is called.

In the most basic case, this can update the contents of a `TextView`, as shown in this code:

```
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);
    }
};
```

Another common case is to display data in a `View` that works with an adapter. For example, if you are showing data in a `RecyclerView`, the `onChanged()` method updates the data cached in the adapter:

```
mWordViewModel.getAllWords().observe(this, new Observer<List<Word>>() {
   @Override
   public void onChanged(@Nullable final List<Word> words) {
       // Update the cached copy of the words in the adapter.
       adapter.setWords(words);
   }
});
```

Tip: One way to get a reference to the application context in a data repository is to extend the `Application` class and add a member of type `Context` to that custom subclass.

Lifecycle-aware components

Most of the app components that are defined in the Android framework have lifecycles attached to them. Lifecycles are managed by the operating system or the framework code running in your process. They are core to how Android works and your app must respect them. Not doing so may trigger memory leaks or even app crashes. Activities and fragments are examples of lifecycle-aware components, and `LiveData` is lifecycle aware.

A common pattern is to implement the actions of the dependent components in the lifecycle methods of activities and fragments. For example, you might have a listener class that connects to a service when the activity starts, and disconnects when the activity is stopped. In the activity, you then override the `onStart()` and `onStop()` methods to start and stop the listener.

```
@Override
public void onStart() {
    super.onStart();
    myListener.start();
}
```

This code snippet looks innocent enough. However, once you have multiple components, using this pattern leads to poor code organization and a proliferation of errors, such as possible race conditions.

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. This results in code that is better-organized, usually shorter, and always easier to maintain.

The `android.arch.lifecycle` package provides classes and interfaces that let you build lifecycle-aware components that automatically adjust their behavior based on the lifecycle state of an activity or fragment. That is, you can make any class lifecycle aware.

- To import `android.arch.lifecycle` into your Android project, see Adding Components to your Project.
- See Handling Lifecycles with Lifecycle-Aware Components for more details on using these components.

Use cases for lifecycle-aware components

Lifecycle-aware components can make it much easier for you to manage lifecycles in a variety of cases. For example, you can use lifecycle-aware components to:

- *Switch between coarse and fine-grained location updates.*

  Use lifecycle-aware components to enable fine-grained location updates while your location app is visible, then switch to coarse-grained updates when the app is in the background. Add `LiveData` to automatically update the UI when your user changes locations.

- *Stop and start video buffering.*

  Use lifecycle-aware components to start video buffering as soon as possible, but defer playback until the app is fully started. You can also use lifecycle-aware components to end buffering when your app is destroyed.

- *Start and stop network connectivity.*

  Use lifecycle-aware components to enable live updating (streaming) of network data while an app is in the foreground, then automatically pause when the app moves into the background.

- *Pause and resume animated drawables.*

  Use lifecycle-aware components to pause animated drawables while the app is in the background, then resume drawables after the app returns to the foreground.

Lifecycle basic example

To use basic lifecycle observation, add an observer to the lifecycle that you want to observe, and specify what you want to happen for the lifecycle events you are interested in.

Below is the code for a very simple `LifecycleObserver` implementation. In a real project, you would want to do more useful things, such as connecting to a service or polling a sensor.

Note: You can display a toast on start and stop, because the method for starting is executed after the activity is started, while the method associated with stopping executes before the activity is stopped or destroyed.

```
public class MainActivity extends AppCompatActivity {

   private Aquarium myAquarium;

   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       // Create an aquarium. Pass in the context and lifecycle you
       // want to observe. In this case, the context is the application,
       // and the lifecycle of this activity.
       myAquarium = new Aquarium(this.getApplication(), getLifecycle());
   }
}


// Add lifecycle awareness to this POJO that could be doing much more
// complex things. For example, if you have fish, they could stop moving
// when the app is not in the foreground, to save resources.
public class Aquarium {

   // Constructor that receives the Application and a lifecycle.
   public Aquarium(final Application app, Lifecycle lifecycle) {

       // Create and add a new observer to the lifecycle.
       lifecycle.addObserver(new LifecycleObserver() {

           // Use annotation to indicate where in the lifecycle the
           // following method should run. In the method, do whatever
           // you want to happen on start.
           @OnLifecycleEvent(Lifecycle.Event.ON_START)
           public void start() {
               Log.d("LifecycleListener", "LIGHTS ON");
               Toast.makeText(app, "LIGHTS ON",
                   Toast.LENGTH_SHORT).show();
           }

           @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
           public void stop() {
               Log.d("LifecycleListener", "LIGHTS OFF");
               Toast.makeText(app, "LIGHTS OFF",
                   Toast.LENGTH_SHORT).show();
           }
       });

   }
}
```

Paging library

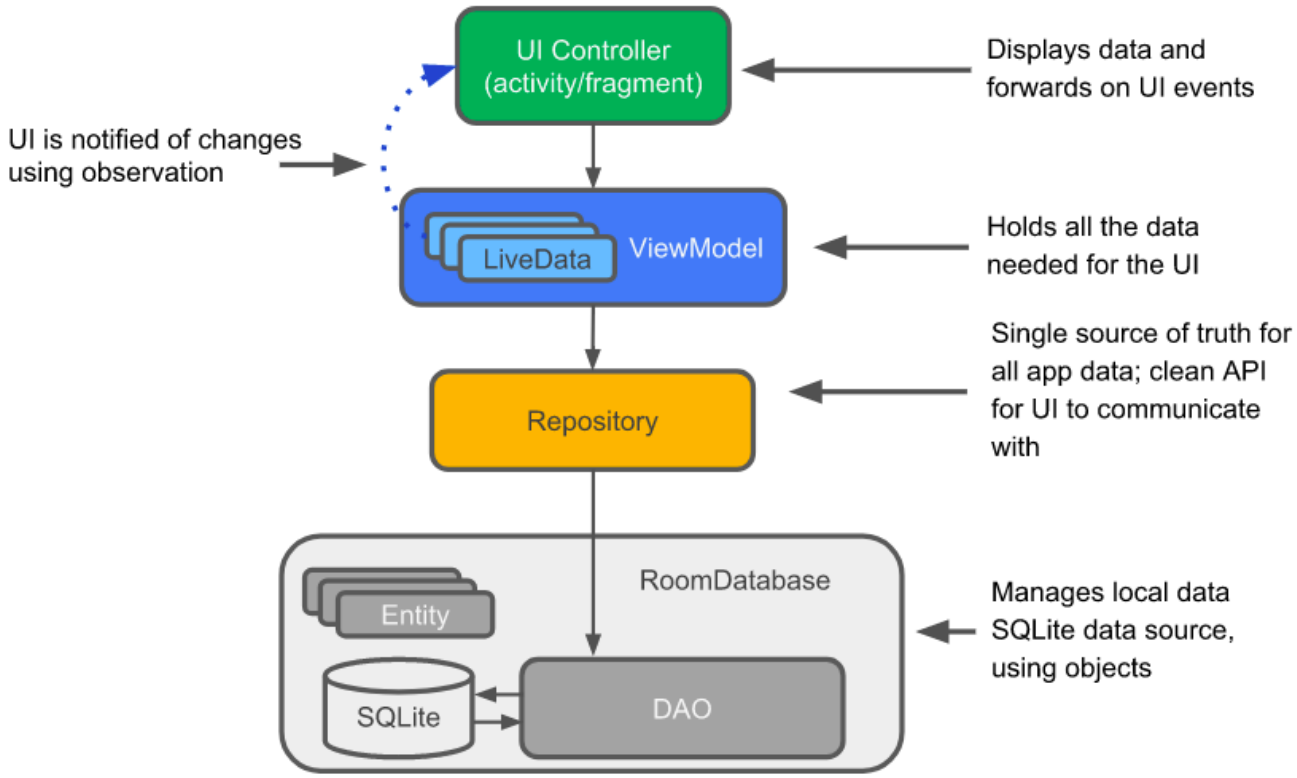
The paging library makes it easier for your app to gradually load information as needed from a data source, without overloading the device or waiting too long for a big database query.

Many apps work with large sets of data, but only need to load and display a small portion of that data at a time. An app might have thousands of items that it can display, but it might only need access to a few dozen of these items at once. If the app isn't designed with care, it can request data that it doesn't need, which places a performance burden on the device, and on the network. If the data is stored or synchronized with a remote database, this can also slow the app and waste the user's data plan.

The paging library addresses issues with existing solutions. This library contains several classes to streamline the process of requesting data as you need it. These classes also work seamlessly with existing Architecture Components like Room.

Learn more about the paging library in the official documentation.

Summary



Suppose you have an app that displays data, for example in a list in a `RecyclerView`, and you can add data to that list, or change data in the list. If the app implements the architecture shown in the diagram above, then the following are true:

- Some of that data is based on an instance of an `Entity` class, which represents an entity in a repository.
- The `Adapter` caches the list of data, and the list is automatically updated and redisplayed when the data changes.
- The automatic update happens because in the `MainActivity`, there is an `Observer` that observes the data and is notified when it changes. When there is a change, the observer's `onChange()` method is executed and updates the cached data in the `Adapter`.
- The data can be observed because it is `LiveData`. And what is observed is the `LiveData<List<Entity>>` that is returned by a method in the `ViewModel` object.
- The `ViewModel` hides everything about the backend from the user interface. It provides methods for accessing the UI data, and it returns `LiveData` so that `MainActivity` can set up the observer relationship. Views, activities, and fragments only interact with the data through the `ViewModel`. As such, it doesn't matter where the data comes from.
- In this case, the data comes from a Repository. The `ViewModel` does not need to know what that Repository interacts with. It just needs to know how to interact with the Repository, which is through the methods exposed by the Repository.
- The Repository manages one or more data sources. For this architecture, that backend is a Room database. Room is a wrapper around and implements an SQLite database. Room does a lot of work for you that you used to have to do yourself. For example, Room does everything that you used to use an `SQLiteOpenHelper` class to do.
- The DAO maps method calls to database queries, so that when the `Repository` calls a method such as `getAllEntities()`, Room can execute `SELECT * from table ORDER BY entity ASC.`
- The result returned from the query is observed `LiveData`. Therefore, every time the data in Room changes, the `Observer` interface's `onChanged()` method is executed and the UI is updated.