Practical 13: ROOM, LIVEDATA, AND VIEWMODEL

Introduction

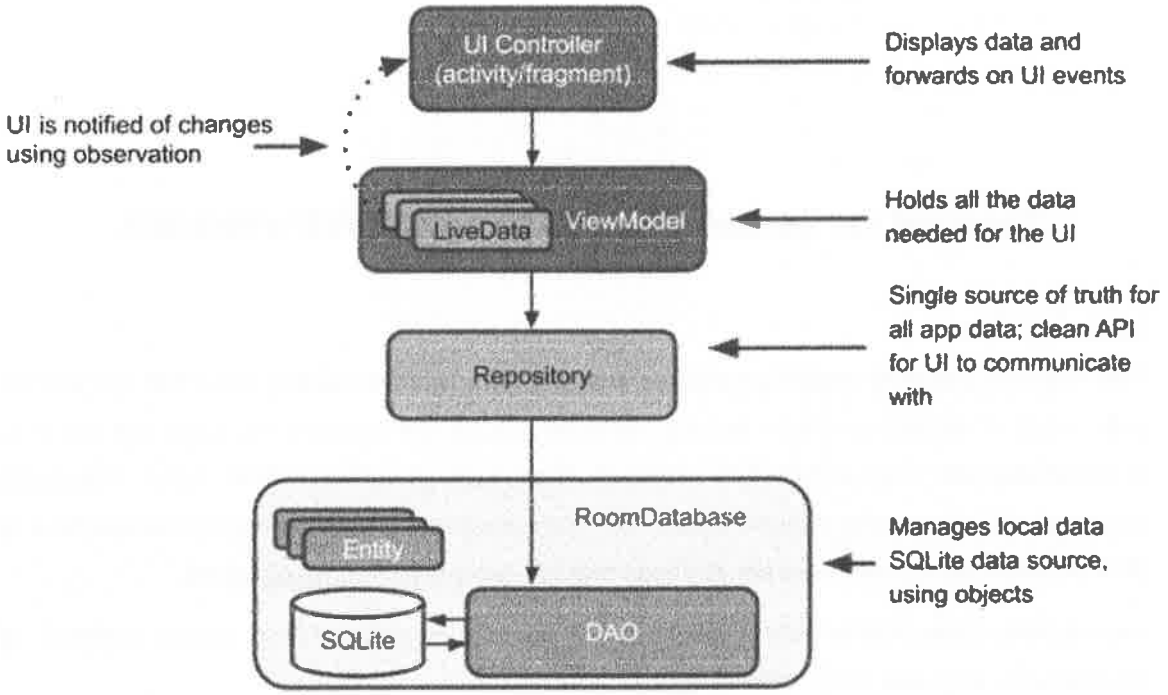
The Android operating system provides a strong foundation for building apps that run well on a wide range of devices and form factors. However, issues like complex lifecycles and the lack of a recommended app architecture make it challenging to write robust apps. The Android Architecture Components provide libraries for common tasks such as lifecycle management and data persistence to make it easier to implement the recommended architecture.

Architecture Components help you structure your app in a way that is robust, testable, and maintainable with less boilerplate code.

What are the recommended Architecture Components?

When it comes to architecture, it helps to see the big picture first. To introduce the terminology, here's a short overview of the Architecture Components and how they work together. Each component is explained more as you use it in this practical.

The diagram below shows a basic form of the recommended architecture for apps that use Architecture Components. The architecture consists of a UI controller, a ViewModel that serves LiveData, a Repository, and a Room database. The Room database is backed by an SQLite database and accessible through a data access object (DAO). Each component is described briefly below, and in detail in the Architecture Components concepts chapter, 10.1: Storing data with Room. You implement the components in this practical.



Because all the components interact, you will encounter references to these components throughout this practical, so here is a short explanation of each.

Entity: In the context of 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. The Room persistence library creates and maintains this database for you.

DAO: Short for *data access object*. A mapping of SQL queries to functions. You used to have to define these queries in a helper class. When you use a DAO, your code calls the functions, and the components take care of the rest.

Room database: Database layer on top of an SQLite database that takes care of mundane tasks that you used to handle with a helper class. The Room database uses the DAO to issue queries to the SQLite database based on functions called.

Repository: A class that you create for managing multiple data sources. In addition to a Room database, the Repository could manage remote data sources such as a web server.

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. Generally, UI components observe relevant data. LiveData is lifecycle-aware, so it automatically manages stopping and resuming observation based on the state of its observing activity or fragment.

App overview

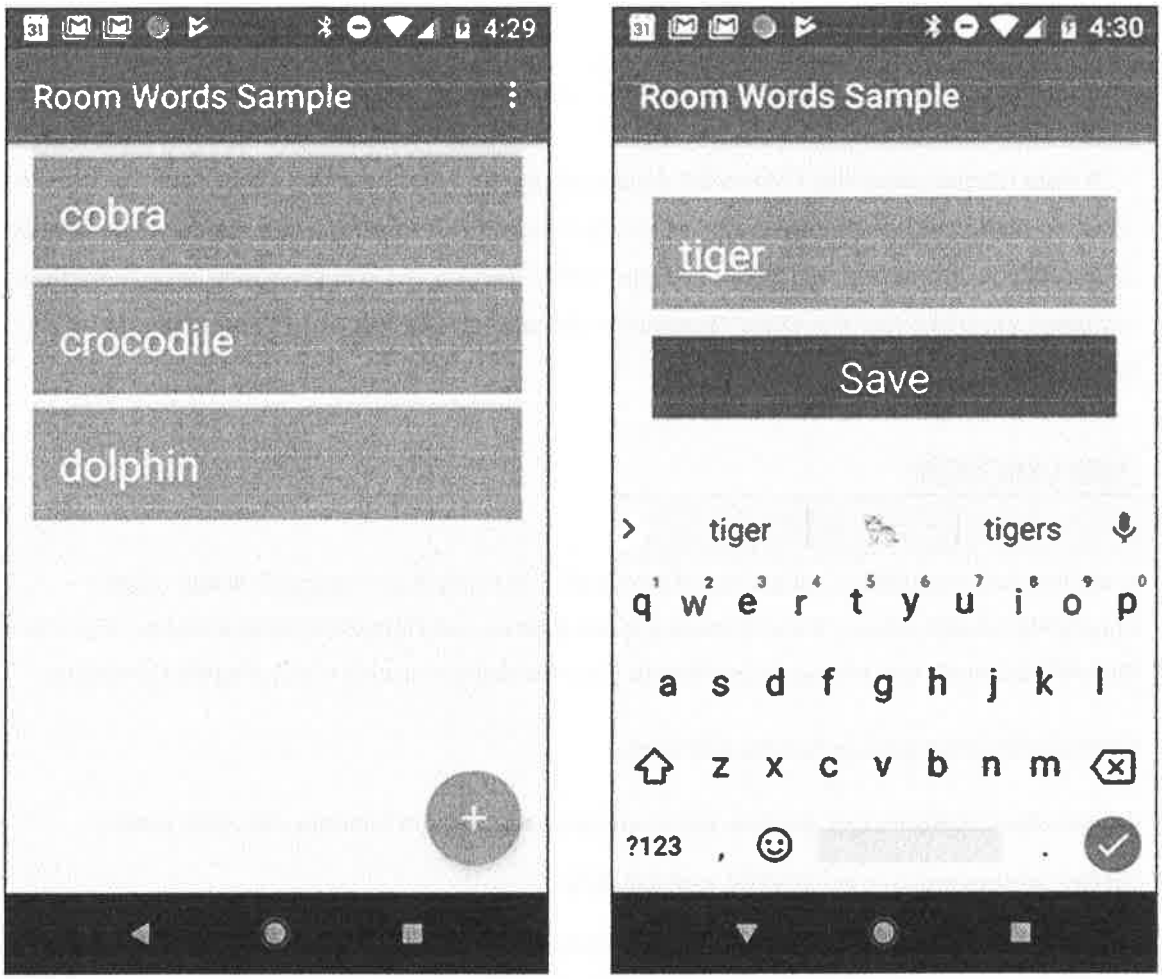
In this practical you build an app that uses the Android Architecture Components. The app, called RoomWordsSample, stores a list of words in a Room database and displays the list in a RecyclerView. The RoomWordsSample app is basic, but sufficiently complete that you can use it as a template to build on.

The RoomWordsSample app does the following:

- Works with a database to get and save words, and pre-populates the database with some words.
- Displays all the words in a RecyclerView in MainActivity.
- Opens a second Activity when the user taps the + FAB button. When the user enters a word, the app adds the word to the database and then the list updates automatically.

The screenshots below show the following:

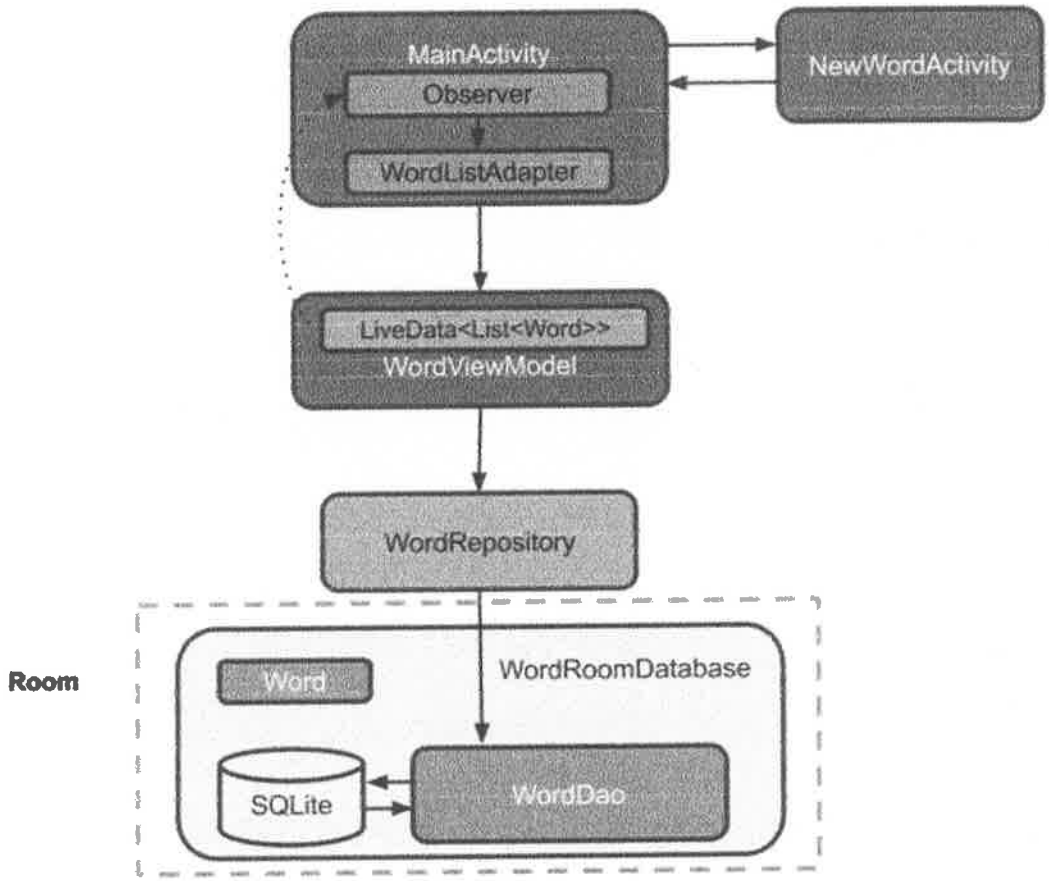
- The RoomWordsSample app as it starts, with the initial list of words
- The activity to add a word



RoomWordsSample architecture overview

The following diagram mirrors the overview diagram from the introduction and shows all the pieces of the RoomWordsSample app. Each of the enclosing boxes (except for the SQLite database) represents a class that you create.

Tip: Print or open this diagram in a separate tab so you can refer to it as you build the code.


MainActivity
Observer
WordListAdapter
NewWordActivity
LiveData<List<Word>>
WordViewModel
WordRepository
Room
Word
WordRoomDatabase
SQLite
WordDao

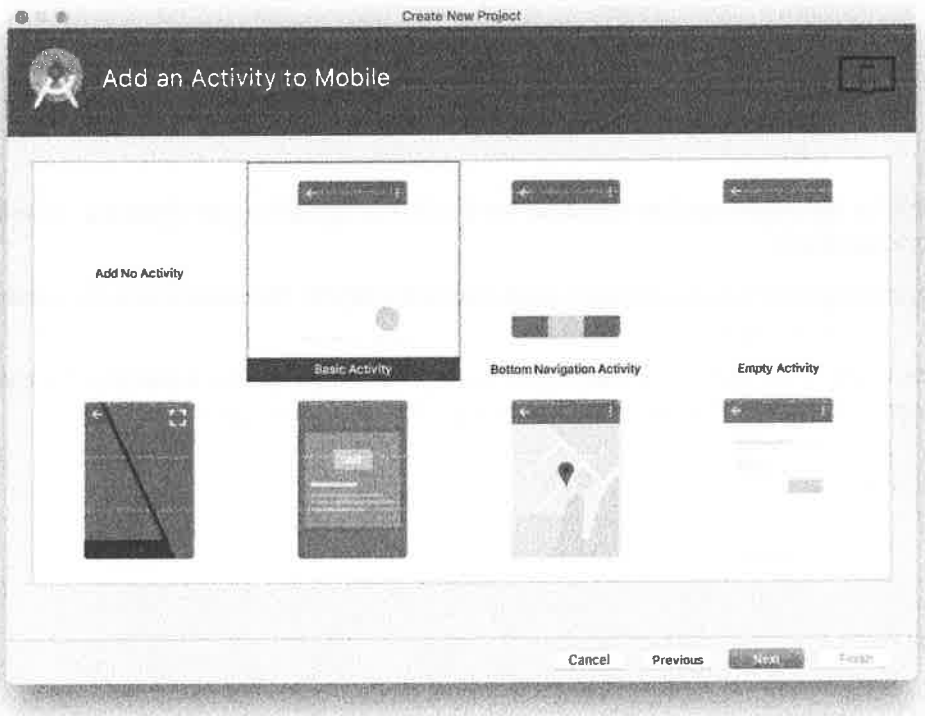
Task 1: Create the RoomWordsSample app

In this practical, you are expected to create member variables, import classes, and extract values as needed. Code that you are expected to be familiar with is provided but not explained.

1.1 Create an app with one Activity

Open Android Studio and create an app. On the setup screens, do the following:

- Name the app RoomWordsSample.
- If you see check boxes for **Include Kotlin support** and **Include C++ support**, uncheck both boxes.
- Select only the **Phone & Tablet** form factor, and set the minimum SDK to API 14 or higher.
- Select the **Basic Activity**.



1.2 Update Gradle files

In Android Studio, manually add the Architecture Component libraries to your Gradle files.

1. Add the following code to your **build.gradle (Module: app)** file, to the bottom of the dependencies block (but still inside it).

```
// 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"
```

2. In your **build.gradle (Project: RoomWordsSample)** file, add the version numbers at the end of the file.

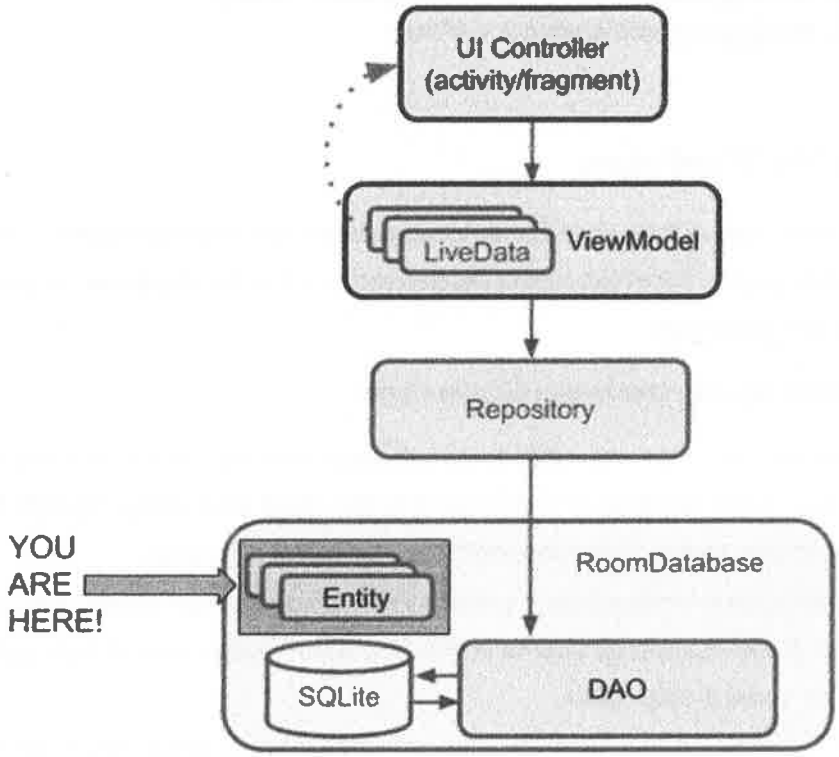
```
ext {
  roomVersion = '1.1.1'
  archLifecycleVersion = '1.1.1'
}
```

Important: Use the latest version numbers for the Room and lifecycle libraries. To find the latest version numbers:

1. On the Adding Components to your Project page, find the entry for the component, for example Room.
2. The version number is defined at the start of the component's dependencies definition. For example, the Room version number in the definition below is 1.1.1: `def room_version = "1.1.1"`

Task 2: Create the Word entity

The diagram below is the complete architecture diagram with the component that you are going to implement in this task highlighted. Every task will have such a diagram to help you understand where the current component fits into the overall structure of the app, and to see how the components are connected.



The data for this app is words, and each word is represented by an entity in the database. In this task you create the **Word** class and annotate it so Room can create a database table from it. The diagram below shows a **word_table** database table. The table has one **word** column, which also acts as the primary key, and two rows, one each for "Hello" and "World."

`word_table` table
`word` (primary key, string)
"Hello"
"World"

2.1 Create the Word class

1. Create a class called **Word.**
2. Add a constructor that takes a word string as an argument. Add the **@NonNull** annotation so that the parameter can never be **null.**
3. Add a "getter" method called **getWord()**that returns the word. Room requires "getter" methods on the entity classes so that it can instantiate your objects.

```
public class Word {
  private String mWord;
  public Word(@NonNull String word) {this.mWord = word;}
  public String getWord(){return this.mWord;}
}
```

2.2 Annotate the Word class

To make the Word class meaningful to a Room database, you must annotate it. Annotations identify how each part of the Word class relates to an entry in the database. Room uses this information to generate code.

You use the following annotations in the steps below:

- **@Entity(tableName = "word_table")** Each **@Entity** class represents an entity in a table. Annotate your class declaration to indicate that the class is an entity. 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 keep things simple, each word in the RoomWordsSample app acts as its own primary key. To learn how to auto-generate unique keys, see the tip below.
- **@NonNull** Denotes that a parameter, field, or method return value can never be null. The primary key should always use this annotation. Use this annotation for any mandatory fields in your rows.
- **@ColumnInfo(name = "word")** Specify the name of a 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. This app provides a **getWord()** "getter" method rather than exposing member variables directly.

For a complete list of annotations, see the Room package summary reference.

Update your **Word** class with annotations, as shown in the code below.

1. Add the **@Entity** notation to the class declaration and set the **tableName** to **"word_table"**.
2. Annotate the **mWord** member variable as the **@PrimaryKey**. Require **mWord** to be **@NonNull**, and name the column **"word"**.

Note: If you *type in* the annotations, Android Studio auto-imports everything you need.

Here is the complete code:

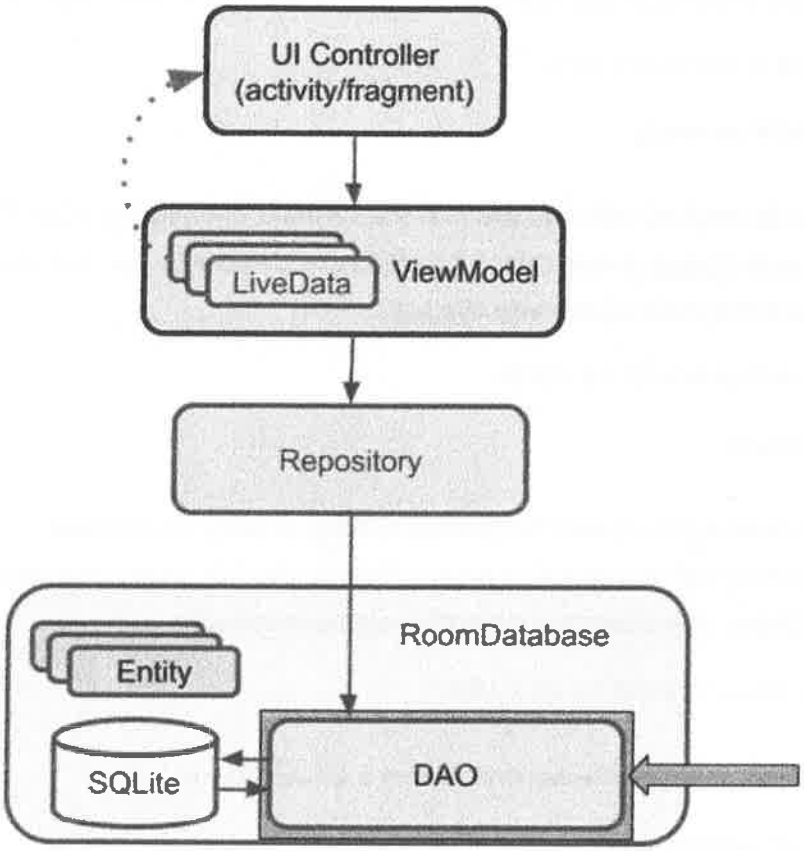
```
@Entity(tableName = "word_table")
public class Word {

  @PrimaryKey
  @NonNull
  @ColumnInfo(name = "word")
  private String mWord;

  public Word(@NonNull String word) {this.mWord = word;}

  public String getWord(){return this.mWord;}
}
```

If you get errors for the annotations, you can import them manually, as follows:

```
import android.arch.persistence.room.ColumnInfo;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
```

Tip on auto-generating keys: To auto-generate a unique key for each entity, you would add and annotate a primary integer key with autoGenerate=true. See Defining data using Room entities.

Task 3: Create the DAO



The data access object, or **Dao**, is an annotated class where you 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**.

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, all queries **(@Query)** must be executed on a thread other than the main thread. (You work on that later.) For operations such as inserting or deleting, if you use the provided convenience annotations, Room takes care of thread management for you.

3.1 Implement the DAO class

The DAO for this practical is basic and only provides queries for getting all the words, inserting words, and deleting all the words.

1. Create a new **interface** and call it **WordDao**.
2. Annotate the class declaration with **@Dao** to identify the class as a DAO class for Room.
3. Declare a method to insert one word:

```
void insert(Word word);
```

4. Annotate the **insert()** method with **@Insert.** You don't have to provide any SQL! (There are also **@ Delete** and **@ Update** annotations for deleting and updating a row, but you do not use these operations in the initial version of this app.)
5. Declare a method to delete all the words:

```
void deleteAll();
```

6. There is no convenience annotation for deleting multiple entities, so annotate the **deleteAll()** method with the generic **@Query**. Provide the SQL query as a string parameter to **@Query**. Annotate the **deleteAll()** method as follows:

```
@Query("DELETE FROM word_table")
```

7. Create a method called **getAllWords()** that returns a **List** of **Words**:

```
List<Word> getAllWords();
```

8. Annotate the **getAllWords()** method with an SQL query that gets all the words from the **word_table**, sorted alphabetically for convenience:

```
@Query("SELECT * from word_table ORDER BY word ASC")
```

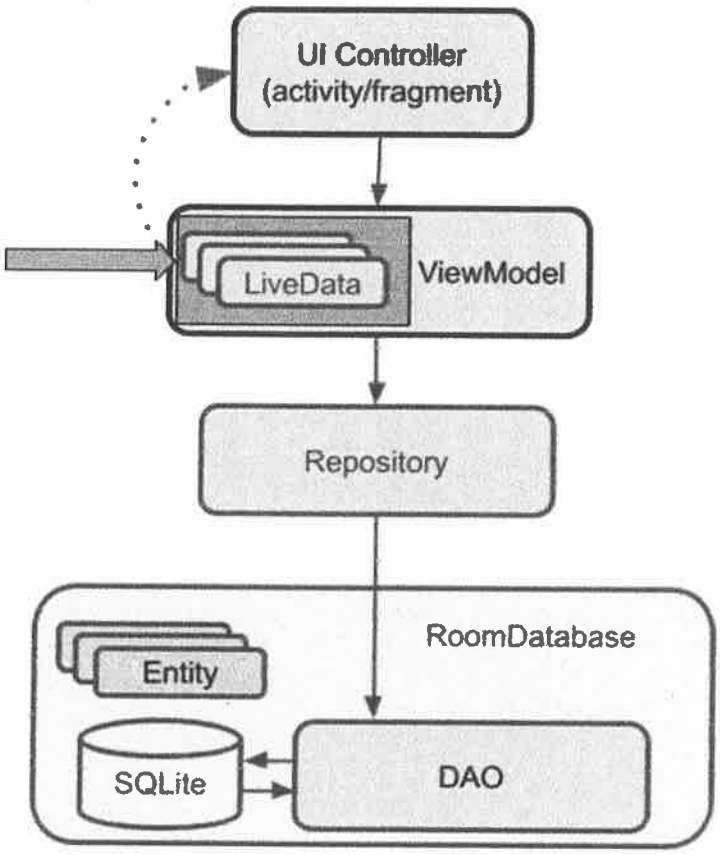
Here is the completed code for the **WordDao** class:

```
@Dao
public interface WordDao {

  @Insert
  void insert(Word word);

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

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

Tip: For this app, ordering the words is not strictly necessary. However, by default, return order is not guaranteed, and ordering makes testing straightforward.

To learn more about DAOs, see Accessing data using Room DAOs.

Task 4: Use LiveData



When you display data or use data in other ways, you usually want to take some action when the data changes. This means you have to observe the data so that when it changes, you can react.

LiveData, which is a lifecycle library class for data observation, can help your app respond to data changes. 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.

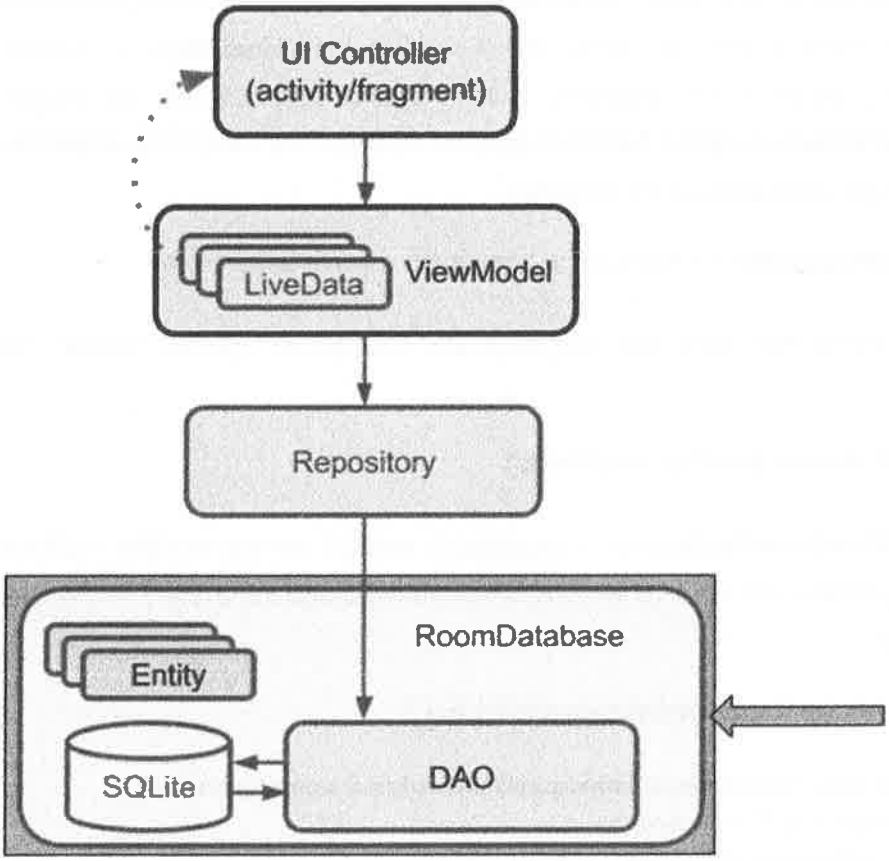
4.1 Return LiveData in WordDao

- In the **WordDao** interface, change the **getAllWords()** method signature so that the returned **List<Word>** is wrapped with **LiveData<>.**

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

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

Task 5: Add a 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 a database helper class such as **SQLiteOpenHelper**.

- Room uses the DAO to issue queries to its database.
- 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.
- Room provides compile-time checks of SQLite statements.
- Your **Room** class must be abstract and extend **RoomDatabase**.
- Usually, you only need one instance of the Room database for the whole app.

5.1 Implement a Room database

1. Create a **public abstract** class that extends **RoomDatabase** and call it **WordRoomDatabase**.

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

2. Annotate the class to be a Room database. Declare the entities that belong in the database—in this case there is only one entity, **Word**. (Listing the **entities** class or classes creates corresponding tables in the database.) Set the version number. Also set export schema to **false, exportSchema** keeps a history of schema versions. For this practical you can disable it, since you are not migrating the database.

```
@Database(entities = {Word.class}, version = 1, exportSchema = false)
```

3. Define the DAOs that work with the database. Provide an abstract "getter" method for each **@Dao**.

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

4. Create the **WordRoomDatabase** as a singleton to prevent having multiple instances of the database opened at the same time, which would be a bad thing. Here is the code to create the singleton:

```
private static WordRoomDatabase INSTANCE;

public static WordRoomDatabase getDatabase(final Context context) {
  if (INSTANCE == null) {
    synchronized (WordRoomDatabase.class) {
      if (INSTANCE == null) {
        // Create database here
      }
    }
  }
  return INSTANCE;
}
```

5. Add code to create a database where indicated by the **Create database here** comment in the code above.

   The following code uses Room's database builder to create a RoomDatabas``e object named **"word_database"** in the application context from the **WordRoomDatabase** class.

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

6. Add a migration strategy for the database.

   In this practical you don't update the entities and the version numbers. However, if you modify the database schema, you need to update the version number and define how to handle migrations. For a sample app such as the one you're creating, destroying and re-creating the database is a fine migration strategy. For a real app, you must implement a non-destructive migration strategy. See Understanding migrations with Room.

   Add the following code to the builder, before calling **build()**

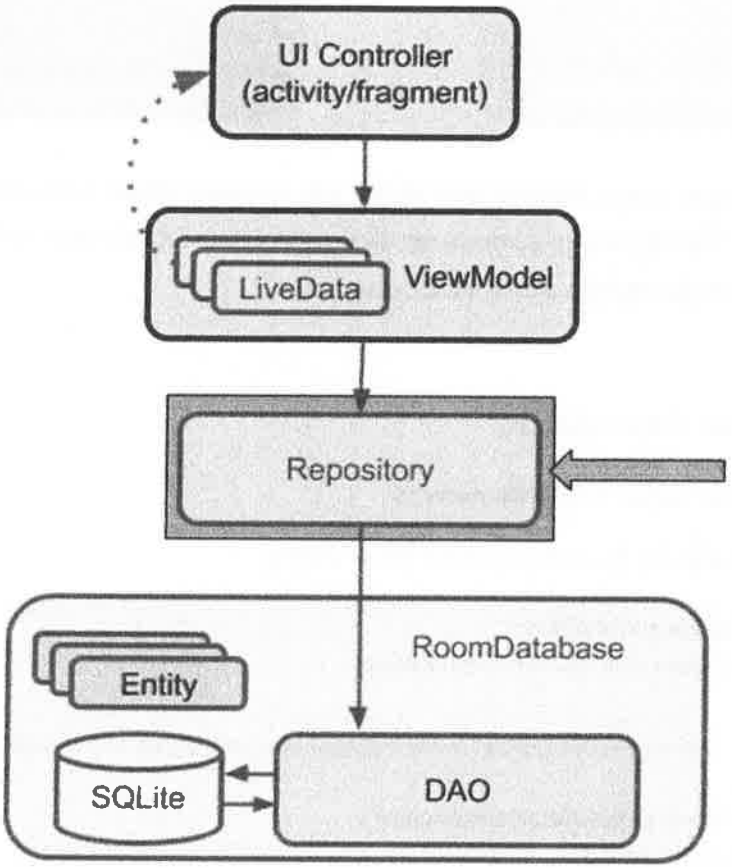
```
// Wipes and rebuilds instead of migrating
// if no Migration object.
// Migration is not part of this practical.
.fallbackToDestructiveMigration()
```

Here is the complete code for the whole **WordRoomDatabase** class:

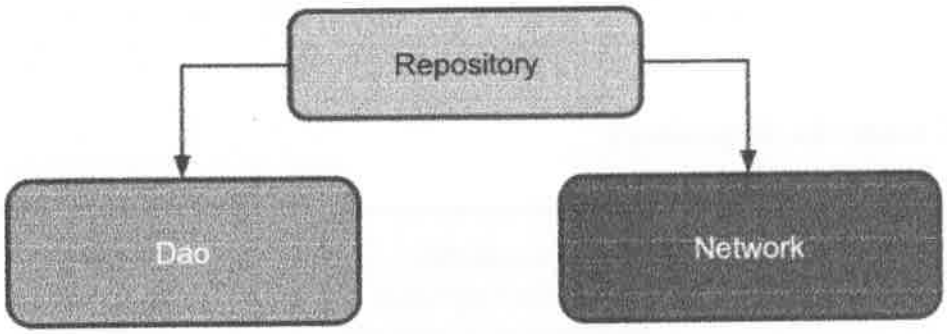
```
@Database(entities = {Word.class}, version = 1, exportSchema = false)
public abstract class WordRoomDatabase extends RoomDatabase {

  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.
               // Migration is not part of this practical.
              .fallbackToDestructiveMigration()
              .build();
        }
      }
    }
    return INSTANCE;
  }
}
```

Important: In Android Studio, if you get errors when you paste code or during the build process, make sure you are using the full package name for imports. See Adding Components to your Project. Then select **Build > Clean Project**. Then select **Build > Rebuild Project**, and build again.

Task 6: Create the 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 manages query threads and allows you to use multiple backends. In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in the local database.

6.1 Implement the Repository

1. Create a public class called **WordRepository.**
2. Add member variables for the DAO and the list of words.

```
private WordDao mWordDao;
private LiveData<List<Word>> mAllWords;
```

3. Add a constructor that gets a handle to the database and initializes the member variables.

```
WordRepository(Application application) {
  WordRoomDatabase db = WordRoomDatabase.getDatabase(application);
  mWordDao = db.wordDao();
  mAllWords = mWordDao.getAllWords();
}
```

4. Add a wrapper method called **getAllWords()** that returns the cached words as **LiveData**. Room executes all queries on a separate thread. Observed **LiveData** notifies the observer when the data changes.

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

5. Add a wrapper for the **insert()** method. Use an **AsyncTask** to call **insert()** on a non-UI thread, or your app will crash. Room ensures that you don't do any long-running operations on the main thread, which would block the UI.

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

6. Create the **insertAsyncTask** as an inner class. You should be familiar with **AsyncTask**, so here is the **insertAsyncTask** code for you to copy:

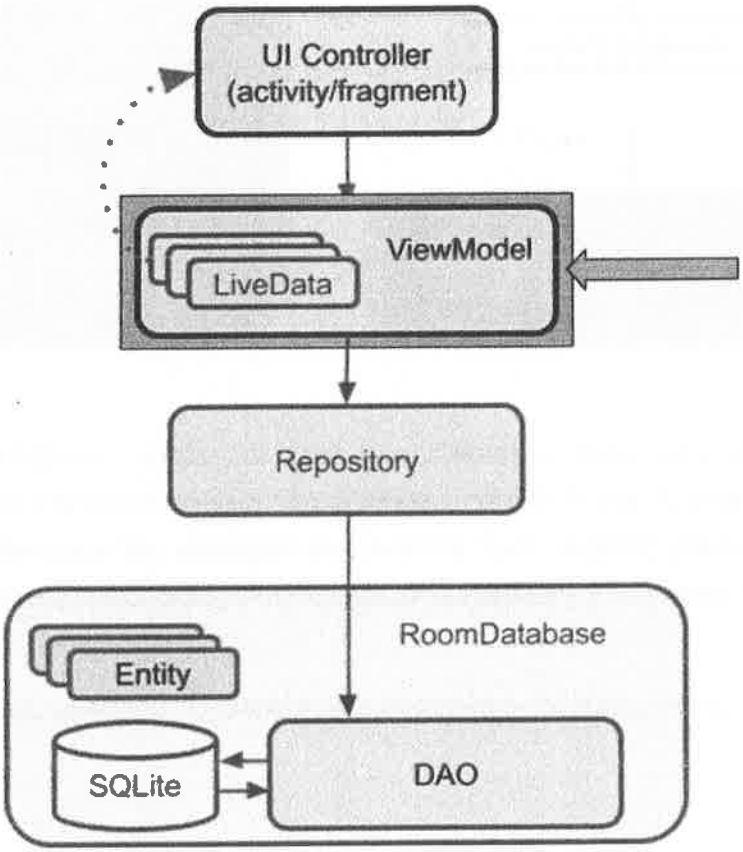
```
private static class insertAsyncTask extends AsyncTask<Word, Void, Void> {

   private WordDao mAsyncTaskDao;

   insertAsyncTask(WordDao dao) {
      mAsyncTaskDao = dao;
   }

   @Override
   protected Void doInBackground(final Word... params) {
      mAsyncTaskDao.insert(params[0]);
      return null;
   }
}
```

Here is the complete code for the **WordRepository** class:

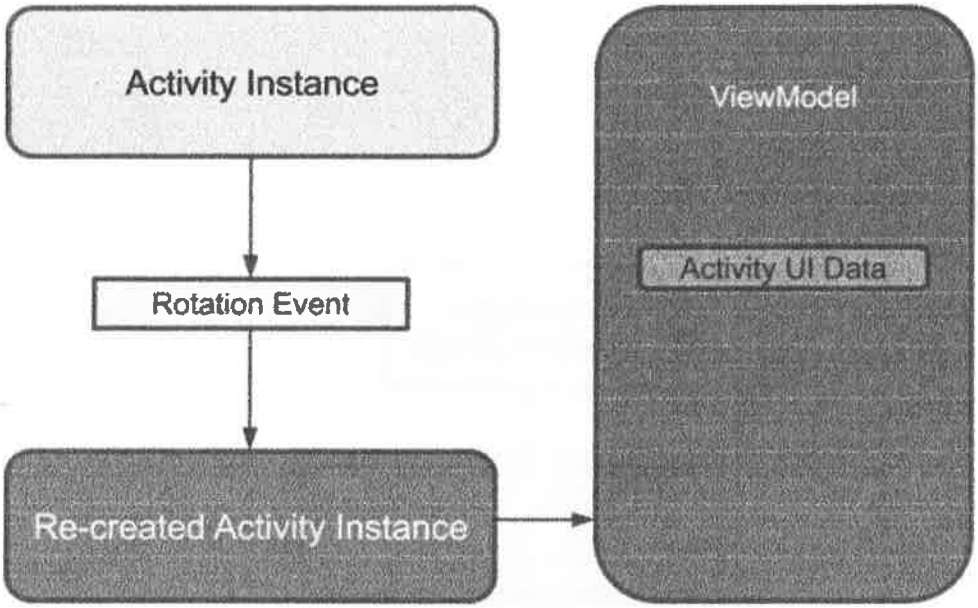
```
public class WordRepository {

  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: For this simple example, the Repository doesn't do much. For a more complex implementation, see the BasicSample code on GitHub.

Task 7: Create the 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. The **ViewModel** is part of the lifecycle library. For an introductory guide to this topic, see **ViewModel**.



A **ViewModel** holds your app's UI data in a 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.

In the **ViewModel,** use **LiveData** for changeable data that the UI will use or display.

7.1 Implement the WordViewModel

1. Create a class called **WordViewModel** that extends **AndroidViewModel**.

   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**, as shown in this practical.

   ```
   public class WordViewModel extends AndroidViewModel {}
   ```

2. Add a private member variable to hold a reference to the Repository.

   ```
   private WordRepository mRepository;
   ```

3. Add a private LiveData member variable to cache the list of words.

   ```
   private LiveData<List<Word>> mAllWords;
   ```

4. Add a constructor that gets a reference to the WordRepository and gets the list of all words from the WordRepository.

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

5. Add a "getter" method that gets all the words. This completely hides the implementation from the UI.

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

6. Create a wrapper insert() method that calls the Repository's insert() method. In this way, the implementation of insert() is completely hidden from the UI.

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

Here is the complete code for WordViewModel:

```
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); }
}
```

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

Task 8: Add XML layouts for the UI

Next, add the XML layout for the list and items to be displayed in the **RecyclerView**.

This practical assumes that you are familiar with creating layouts in XML, so the code is just provided.

8.1 Add styles

1. Change the colors in **colors.xml** to the following: (to use Material Design colors):

```
<resources>
  <color name="colorPrimary">#2196F3</color>
  <color name="colorPrimaryLight">#64b5f6</color>
  <color name="colorPrimaryDark">#1976D2</color>
  <color name="colorAccent">#FFFF9800</color>
  <color name="colorTextPrimary">@android:color/white</color>
  <color name="colorScreenBackground">#fff3e0</color>
  <color name="colorTextHint">#E0E0E0</color>
</resources>
```

2. Add a style for text views in the **values/styles.xml** file:

```
<style name="text_view_style">
  <item name="android:layout_width">match_parent</item>
  <item name="android:layout_height">wrap_content</item>
  <item name="android:textAppearance">
    @android:style/TextAppearance.Large</item>
  <item name="android:background">@color/colorPrimaryLight</item>
  <item name="android:layout_marginTop">8dp</item>
  <item name="android:layout_gravity">center</item>
  <item name="android:padding">16dp</item>
  <item name="android:textColor">@color/colorTextPrimary</item>
</style>
```

8.2 Add item layout

- Add a **layout/recyclerview_item.xml** layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="match_parent"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_height="wrap_content">
```

```
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="@style/text_view_style"
        tools:text="placeholder text" />
</LinearLayout>
```

8.3 Add the RecyclerView

1. In the **layout/content_main.xml** file, add a background color to the **ConstraintLayout**:

   ```
   android:background="@color/colorScreenBackground"
   ```

2. In **content_main.xml** file, replace the **TextView** element with a **RecyclerView** element:

   ```
   <android.support.v7.widget.RecyclerView
     android:id="@+id/recyclerview"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:layout_margin="16dp"
     tools:listitem="@layout/recyclerview_item"
   />
   ```

8.4 Fix the icon in the FAB

The icon in your floating action button (FAB) should correspond to the available action. In the **layout/activity_main.xml** file, give the **FloatingActionButton** a + symbol icon:

1. Select **File > New > Vector Asset**.
2. Select **Material Icon**.
3. Click the Android robot icon in the **Icon:** field, then select the + ("add") asset.
4. In the **layout/activity_main.xml** file, in the **FloatingActionButton,** change the **srcCompat** attribute to:

   ```
   android:src="@drawable/ic_add_black_24dp"
   ```

Task 9: Create an Adapter and adding the RecyclerView

You are going to display the data in a **RecyclerView**, which is a little nicer than just throwing the data in a **TextView**. This practical assumes that you know how **RecyclerView**, **RecyclerView.LayoutManager**, **RecyclerView.ViewHolder**, and **RecyclerView.Adapter** work.

9.1 Create the WordListAdapter class

- Add a class **WordListAdapter** that extends **RecyclerView.Adapter**. The adapter caches data and populates the **RecyclerView** with it. The inner class **WordViewHolder** holds and manages a view for one list item.

Here is the code:

```
public class WordListAdapter extends RecyclerView.Adapter<WordListAdapter.WordViewHolder>
{

  private final LayoutInflater mInflater;
  private List<Word> mWords; // Cached copy of words

  WordListAdapter(Context context) { mInflater = LayoutInflater.from(context); }

  @Override
  public WordViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View itemView = mInflater.inflate(R.layout.recyclerview_item, parent, false);
    return new WordViewHolder(itemView);
  }

  @Override
  public void onBindViewHolder(WordViewHolder holder, int position) {
    if (mWords != null) {
      Word current = mWords.get(position);
      holder.wordItemView.setText(current.getWord());
    } else {
      // Covers the case of data not being ready yet.
      holder.wordItemView.setText("No Word");
    }
  }
```

```
    void setWords(List<Word> words){
        mWords = words;
        notifyDataSetChanged();
    }

    // getItemCount() is called many times, and when it is first called,
    // mWords has not been updated (means initially, it's null, and we can't return null).
    @Override
    public int getItemCount() {
        if (mWords != null)
            return mWords.size();
        else return 0;
    }

    class WordViewHolder extends RecyclerView.ViewHolder {
        private final TextView wordItemView;

        private WordViewHolder(View itemView) {
            super(itemView);
            wordItemView = itemView.findViewById(R.id.textView);
        }
    }
}
```

Note: The **mWords** variable in the adapter caches the data. In the next task, you add the code that updates the data automatically.

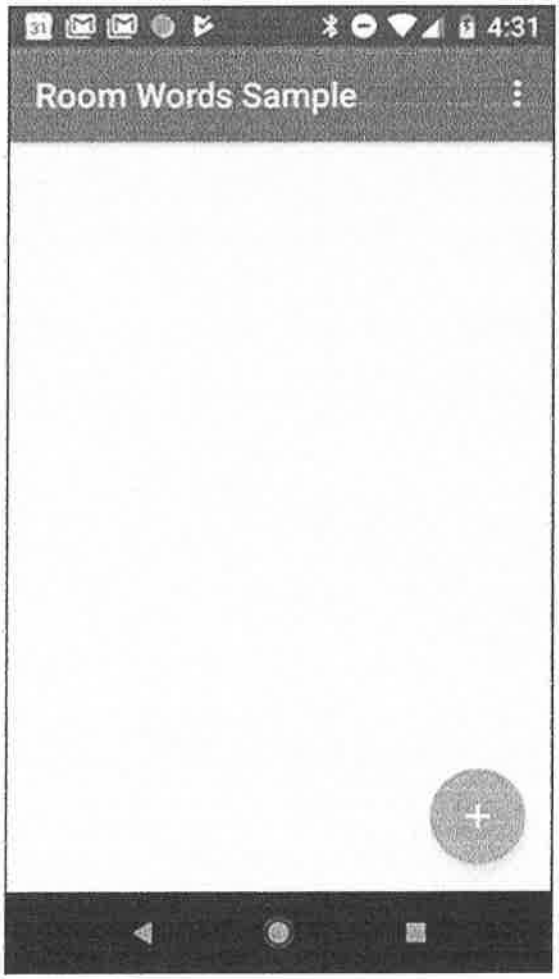
Note: The **getItemCount()** method needs to account gracefully for the possibility that the data is not yet ready and **mWords** is still null. In a more sophisticated app, you could display placeholder data or something else that would be meaningful to the user.

9.2 Add RecyclerView to MainActivity

1. Add the **RecyclerView** in the **onCreate()** method of **MainActivity**:

```
RecyclerView recyclerView = findViewById(R.id.recyclerview);
final WordListAdapter adapter = new WordListAdapter(this);
recyclerView.setAdapter(adapter);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

2. Run your app to make sure the app compiles and runs. There are no items, because you have not hooked up the data yet. The app should display the empty recycler view.



Task 10: Populate the database

There is no data in the database yet. You will add data in two ways: Add some data when the database is opened, and add an **Activity** for adding words. Every time the database is opened, all content is deleted and repopulated. This is a reasonable solution for a sample app, where you usually want to restart on a clean slate.

10.1 Create the callback for populating the database

To delete all content and repopulate the database whenever the app is started, you create a **RoomDatabase.Callback** and override the **onOpen()** method. Because you cannot do Room database operations on the UI thread, **onOpen()** creates and executes an **AsyncTask** to add content to the database.

1. Add the **onOpen()** callback in the **WordRoomDatabase** class:

```
private static RoomDatabase.Callback sRoomDatabaseCallback =
    new RoomDatabase.Callback(){

    @Override
    public void onOpen (@NonNull SupportSQLiteDatabase db){
        super.onOpen(db);
        new PopulateDbAsync(INSTANCE).execute();
    }
};
```

2. Create an inner class **PopulateDbAsync** that extends **AsycTask**. Implement the **doInBackground()** method to delete all words, then create new ones. Here is the code for the **AsyncTask** that deletes the contents of the database, then populates it with an initial list of words. Feel free to use your own words!

```
/**
* Populate the database in the background.
*/
private static class PopulateDbAsync extends AsyncTask<Void, Void, Void> {

    private final WordDao mDao;
    String[] words = {"dolphin", "crocodile", "cobra"};

    PopulateDbAsync(WordRoomDatabase db) {
        mDao = db.wordDao();
    }
```

```
    @Override
    protected Void doInBackground(final Void... params) {
        // Start the app with a clean database every time.
        // Not needed if you only populate the database
        // when it is first created
        mDao.deleteAll();

        for (int i = 0; i <= words.length - 1; i++) {
            Word word = new Word(words[i]);
            mDao.insert(word);
        }
        return null;
    }
}
```

3. Add the callback to the database build sequence in **WordRoomDatabase**, right before you call **.build()**:

```
.addCallback(sRoomDatabaseCallback)
```

Task 11: Connect the UI with the data

Now that you have created the method to populate the database with the initial set of words, the next step is to add the code to display those words in the **RecyclerView.**

To display the current contents of the database, you add an observer that observes the **LiveData** in the **ViewModel**. Whenever the data changes (including when it is initialized), the **onChanged()** callback is invoked. In this case, the **onChanged()** callback calls the adapter's **setWord()** method to update the adapter's cached data and refresh the displayed list.

11.1 Display the words

1. In **MainActivity,** create a member variable for the **ViewModel,** because all the activity's interactions are with the **WordViewModel** only.

```
private WordViewModel mWordViewModel;
```

2. In the **onCreate()** method, get a **ViewModel** from the **ViewModelProviders** class.

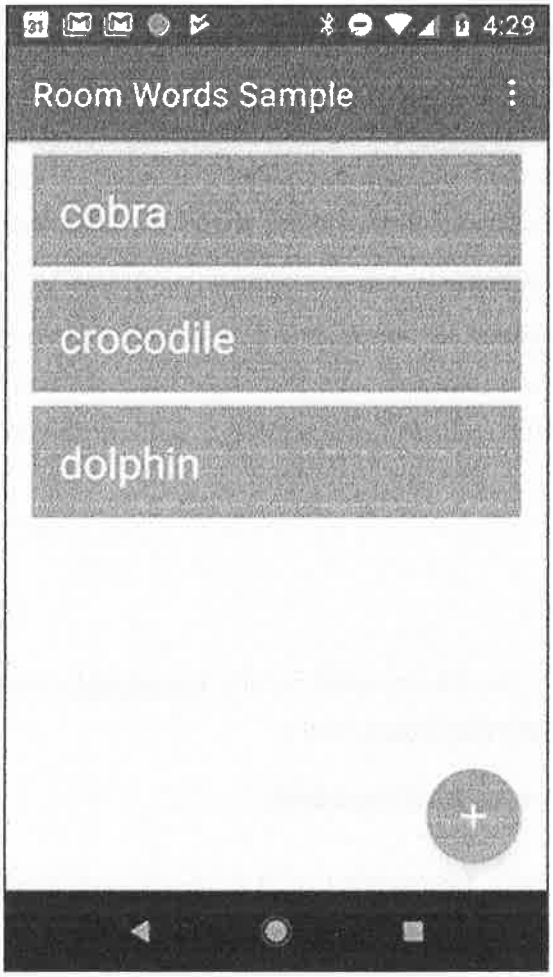
```
mWordViewModel = ViewModelProviders.of(this).get(WordViewModel.class);
```

Use **ViewModelProviders** to associate your **ViewModel** with your UI controller. When your app first starts, the **ViewModelProviders** class creates 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**. See **ViewModel.**

3. Also in **onCreate()**, add an observer for the **LiveData** returned by **getAllWords().** When the observed data changes while the activity is in the foreground, the **onChanged()** method is invoked and updates the data cached in the adapter. Note that in this case, when the app opens, the initial data is added, so **onChanged()** method is called.

```
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);
  }
});
```

4. Run the app. The initial set of words appears in the **RecyclerView**.



Task 12: Create an Activity for adding words

Now you will add an Activity that lets the user use the FAB to enter new words. This is what the interface for the new activity will look like:

12.1 Create the NewWordActivity

1. Add these string resources in the values/strings.xml file:

```
<string name="hint_word">Word...</string>
<string name="button_save">Save</string>
<string name="empty_not_saved">Word not saved because it is empty.</string>
```

3. Add a style for buttons in value/styles.xml:

```
<style name="button_style" parent="android:style/Widget.Material.Button">
  <item name="android:layout_width">match_parent</item>
  <item name="android:layout_height">wrap_content</item>
  <item name="android:background">@color/colorPrimaryDark</item>
  <item name="android:textAppearance">@android:style/TextAppearance.Large</item>
  <item name="android:layout_marginTop">16dp</item>
  <item name="android:textColor">@color/colorTextPrimary</item>
</style>
```

5. Use the Empty Activity template to create a new activity, NewWordActivity. Verify that the activity has been added to the Android Manifest.

```
<activity android:name=".NewWordActivity"></activity>
```

6. Update the activity_new_word.xml file in the layout folder:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@color/colorScreenBackground"
  android:orientation="vertical"
  android:padding="24dp">

  <EditText
    android:id="@+id/edit_word"
    style="@style/text_view_style"
    android:hint="@string/hint_word"
    android:inputType="textAutoComplete" />

  <Button
    android:id="@+id/button_save"
    style="@style/button_style"
    android:text="@string/button_save" />
</LinearLayout>
```

7. Implement the NewWordActivity class. The goal is that when the user presses the **Save** button, the new word is put in an Intent to be sent back to the parent Activity.

Here is the code for the NewWordActivity activity:

```
public class NewWordActivity extends AppCompatActivity {
  public static final String EXTRA_REPLY =
        "com.example.android.roomwordssample.REPLY";

  private  EditText mEditWordView;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_new_word);
    mEditWordView = findViewById(R.id.edit_word);

    final Button button = findViewById(R.id.button_save);
    button.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        Intent replyIntent = new Intent();
        if (TextUtils.isEmpty(mEditWordView.getText())) {
          setResult(RESULT_CANCELED, replyIntent);
        } else {
          String word = mEditWordView.getText().toString();
          replyIntent.putExtra(EXTRA_REPLY, word);
          setResult(RESULT_OK, replyIntent);
        }
        finish();
      }
    });
  }
}
```

12.2 Add code to insert a word into the database

1. In MainActivity, add the onActivityResult() callback for the NewWordActivity. If the activity returns with RESULT_OK, insert the returned word into the database by calling the insert() method of the WordViewModel.

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
  super.onActivityResult(requestCode, resultCode, data);

  if (requestCode == NEW_WORD_ACTIVITY_REQUEST_CODE && resultCode ==
RESULT_OK) {
    Word word = new Word(data.getStringExtra(NewWordActivity.EXTRA_REPLY));
    mWordViewModel.insert(word);
  } else {
    Toast.makeText(
        getApplicationContext(),
        R.string.empty_not_saved,
        Toast.LENGTH_LONG).show();
  }
}
```

2. Define the missing request code:

```
public static final int NEW_WORD_ACTIVITY_REQUEST_CODE = 1;
```

3. In MainActivity,start NewWordActivity when the user taps the FAB. Replace the code in the FAB's onClick() click handler with the following code:

```
Intent intent = new Intent(MainActivity.this, NewWordActivity.class);
startActivityForResult(intent, NEW_WORD_ACTIVITY_REQUEST_CODE);
```

4. Run your app. When you add a word to the database in NewWordActivity, the UI automatically updates.
5. Add a word that already exists in the list. What happens? Does your app crash? Your app uses the word itself as the primary key, and each primary key **must** be unique. You can specify a conflict strategy to tell your app what to do when the user tries to add an existing word.
6. In the WordDao interface, change the annotation for the insert() method to:

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
```

   To learn about other conflict strategies, see the OnConflictStrategy reference.

7. Run your app again and try adding a word that already exists. What happens now?