Android Room con una Vista - Java

1.- Antes de que comience

El propósito de los componentes de arquitectura es brindarle una guía sobre la arquitectura de aplicaciones, con bibliotecas para tareas comunes como la gestión del ciclo de vida y la persistencia de datos. Los Componentes de Arquitectura lo ayudan a estructurar su aplicación de una manera sólida, comprobable y mantenible con menos código repetitivo. Las bibliotecas de Componentes de Arquitectura son parte de Android Jetpack .

Esta es la versión en el lenguaje de programación Java del Codelab. La versión en el lenguaje Kotlin la puede encontrarla aquí .

Si encuentra algún problema (errores de código, errores gramaticales, redacción poco clara, etc.) mientras trabaja en este Codelab, por favor informe el problema a través del enlace Informar un error en la esquina inferior izquierda del Codelab.

Prerequisitos

Debe estar familiarizado con Java, conceptos de diseño orientado a objetos y conceptos básicos de desarrollo Android. En particular:

- **RecyclerView** y adaptadores
- Base de datos SQLite y el lenguaje de consulta SQLite
- Threading (manejo de hilos) y **ExecutorService**
- Es útil estar familiarizado con los patrones arquitectónicos del software que separan los datos de la interfaz de usuario, como MVP o MVC. Este Codelab implementa la arquitectura definida en la Guide to App Architecture .

Este Codelab se centra en los Componentes de Arquitectura de Android. Se proporcionan conceptos off the topic y códigos para que usted simplemente los copie y pegue.

Este Codelab proporciona todo el código que necesita para crear la aplicación completa.

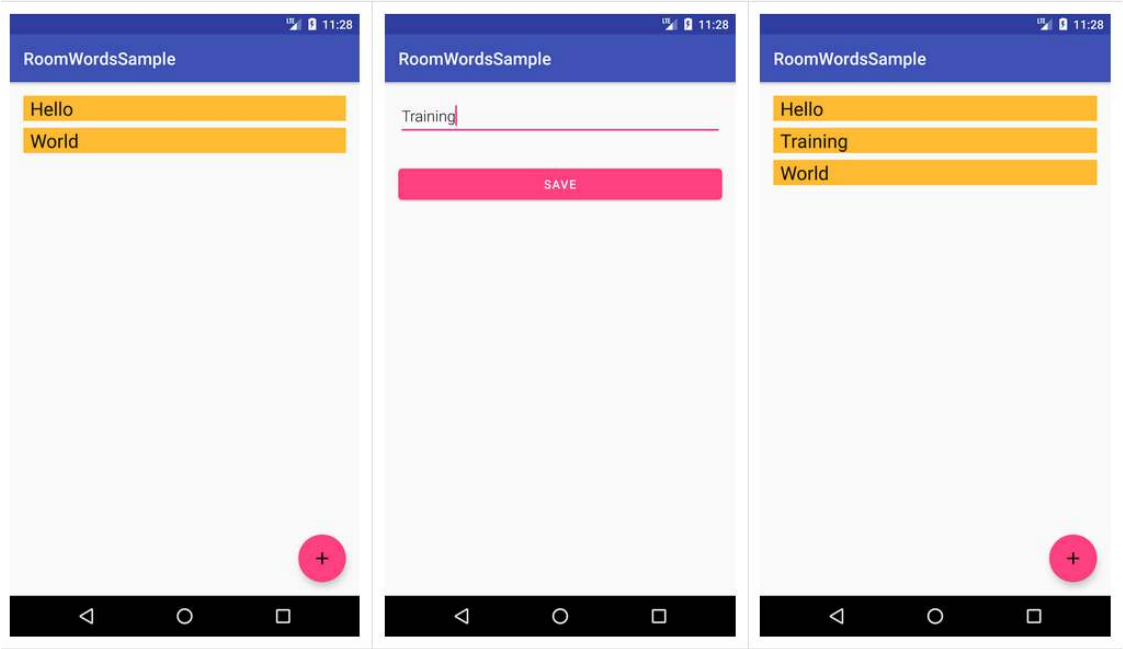
Qué es lo que hará

En este Codelab, aprenderá como diseñar y construir una aplicación utilizando los Componentes de Arquitectura Room, ViewModel y LiveData, y a crear una aplicación que haga lo siguiente:

- Implementar nuestra arquitectura recomendada utilizando los Componentes de la Arquitectura de Android.
- Trabajar con una base de datos para obtener y guardar los datos, y también pre poblar la base de datos con algunas palabras.

- Mostrar todas las palabras en un **RecyclerView** dentro de una **MainActivity**.
- Abrir una segunda actividad cuando el usuario toca el botón +. Cuando el usuario ingresa una palabra, la agrega a la base de datos y a la lista.

La aplicación es sencilla, pero lo suficientemente compleja como para que puedas usarla como plantilla sobre la que desarrollar. Aquí hay una vista previa:



Qué es lo que necesita

- La última versión estable de Android Studio y conocimiento de cómo utilizarlo. Asegúrese de que Android Studio esté actualizado, así como su SDK y Gradle. De lo contrario, es posible que tengas que esperar hasta que se realicen todas las actualizaciones.
- Un dispositivo o emulador de Android.

Tenga en cuenta que el código de la solución está disponible como zip y en github. Le recomendamos que cree la aplicación desde cero y mire este código si se queda atascado.

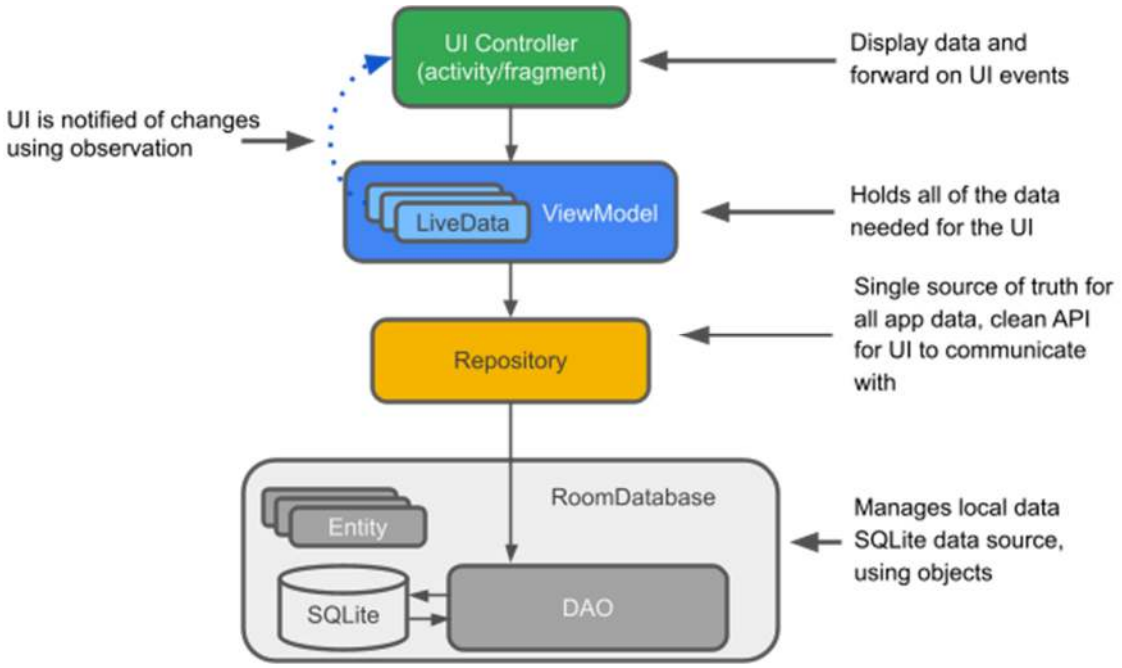
2. Usando los Componentes de la Arquitectura

Hay muchos pasos para utilizar los Componentes de la Arquitectura e implementar la arquitectura recomendada. Lo más importante es crear un modelo mental de lo que está pasando, entendiendo cómo encajan las piezas y cómo fluyen los datos. Mientras trabaja en este Codelab, no se limite a copiar y pegar el código, sino que intente comenzar a construir esa comprensión interna.

Cuales son los Componentes de Arquitectura recomendados?

A continuación, se ofrece una breve introducción a los Componentes de la Arquitectura y cómo funcionan juntos. Tenga en cuenta que este Codelab se centra en un subconjunto de componentes, a saber, LiveData, ViewModel y Room. Cada componente se explica más a medida que lo usa.

Este diagrama muestra una forma básica de esta arquitectura:



Entidad : Clase anotada que describe una tabla de base de datos cuando se trabaja con Room .

Base de datos SQLite: almacenamiento en el dispositivo. La biblioteca de persistencia Room crea y mantiene esta base de datos por usted.

DAO : Objeto de acceso a datos. Un mapeo de consultas SQL a funciones. Cuando usa un DAO, llama a los métodos y Room se encarga del resto.

Base de datos Room : simplifica el trabajo de la base de datos y sirve como punto de acceso a la base de datos SQLite subyacente (oculta **SQLiteOpenHelper**). La base de datos Room utiliza el DAO para realizar las consultas a la base de datos SQLite.

Repositorio: Se utiliza para gestionar múltiples fuentes de datos.

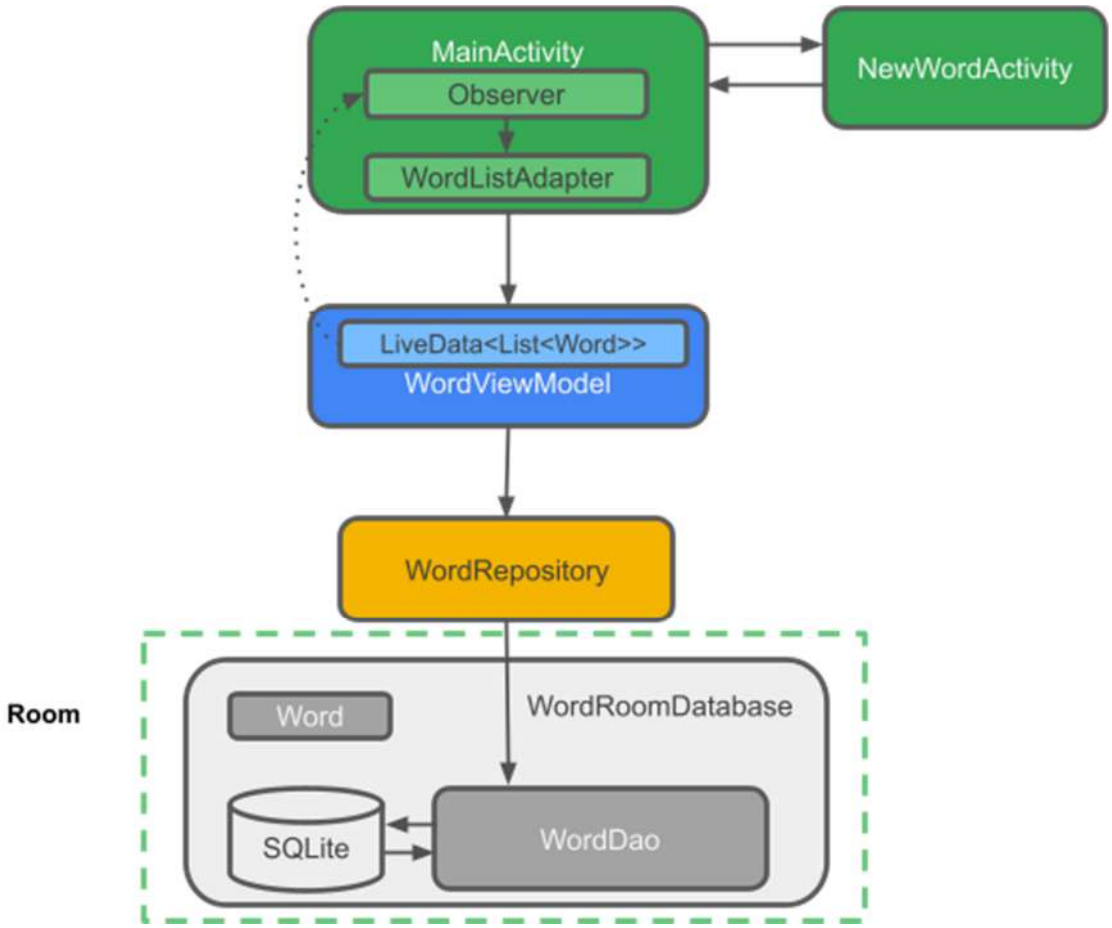
ViewModel : actúa como un centro de comunicación entre el Repositorio (datos) y la UI. La interfaz de usuario ya no necesita preocuparse por el origen de los datos. Las instancias de ViewModel sobreviven a la recreación de la Actividad/Fragmento.

LiveData : una clase contenedora de datos que puede ser observada . Siempre mantiene/almacena en caché la última versión de los datos y notifica a sus observadores cuando los datos han cambiado. **LiveData** es consciente del ciclo de vida. Los componentes de la UI simplemente observan datos relevantes y no detienen ni reanudan la observación. LiveData gestiona todo esto automáticamente, ya que es consciente de los cambios relevantes en el estado del ciclo de vida mientras observa.

¿Buscando más información? Consulte la Guide to App Architecture.

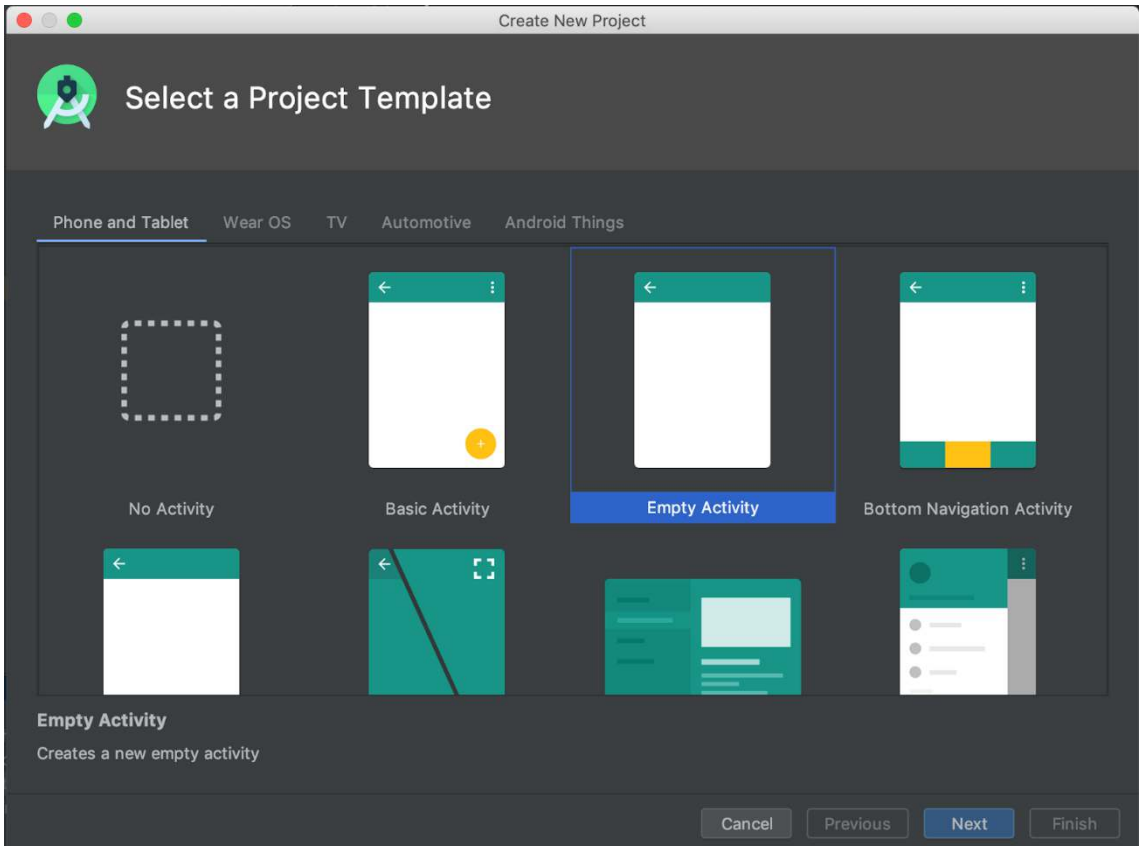
Descripción general de la arquitectura de RoomWordSample

El siguiente diagrama muestra todas las piezas de la aplicación. Cada uno de los cuadros adjuntos (excepto la base de datos SQLite) representa una clase que creará.



3. Crear su app

1. Abra Android Studio y haga clic en **New Project**.
2. En la ventana **New Project**, elija **Empty Activity** y haga clic en **Next.**
3. En la siguiente pantalla, asigne a la aplicación el nombre RoomWordSample y haga clic en **Finish**.



4. Actualizar sus archivos Gradle

Luego, tiene que agregar las librerías de los componentes en sus archivos Gradle.

1. In Android Studio, click en el tab Projects y expanda la carpeta Gradle Scripts.
2. Abra **build.gradle** (**Module: app**).
3. Agregue el siguiente bloque de **compileOptions** dentro del bloque **android** para establecer el target y el source compatibility a 1.8, lo que le permitirá usar las lambdas del JDK 8 posteriormente:

```
compileOptions {
    sourceCompatibility = 1.8
    targetCompatibility = 1.8
}
```

4. Reemplace el bloque de **dependencies** con:

```
dependencies {
    implementation "androidx.appcompat:appcompat:$rootProject.appCompatVersion"

    // Dependencies for working with Architecture components
    // You'll probably have to update the version numbers in build.gradle
(Project)

    // Room components
    implementation "androidx.room:room-runtime:$rootProject.roomVersion"
    annotationProcessor "androidx.room:room-compiler:$rootProject.roomVersion"
    androidTestImplementation "androidx.room:room-
testing:$rootProject.roomVersion"

    // Lifecycle components
    implementation "androidx.lifecycle:lifecycle-
viewmodel:$rootProject.lifecycleVersion"
    implementation "androidx.lifecycle:lifecycle-
livedata:$rootProject.lifecycleVersion"
    implementation "androidx.lifecycle:lifecycle-common-
java8:$rootProject.lifecycleVersion"

    // UI
    implementation
"androidx.constraintlayout:constraintlayout:$rootProject.constraintLayoutVersion
"
    implementation
"com.google.android.material:material:$rootProject.materialVersion"

    // Testing
    testImplementation "junit:junit:$rootProject.junitVersion"
    androidTestImplementation "androidx.arch.core:core-
testing:$rootProject.coreTestingVersion"
    androidTestImplementation ("androidx.test.espresso:espresso-
core:$rootProject.espressoVersion", {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    androidTestImplementation
"androidx.test.ext:junit:$rootProject.androidxJunitVersion"
}
```

En su archivo **build.gradle** (**Project: RoomWordsSample**), agregue los números de versión al final del archivo, como se muestra en el código de abajo:

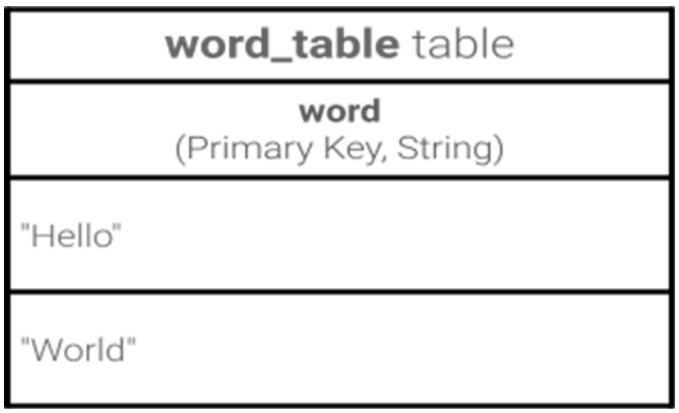
Obtenga los números de versiones más actuales desde la página Adding Components to your Project.

```
ext {
    appCompatVersion = '1.5.1'
    constraintLayoutVersion = '2.1.4'
    coreTestingVersion = '2.1.0'
    lifecycleVersion = '2.3.1'
    materialVersion = '1.3.0'
    roomVersion = '2.3.0'
    // testing
    junitVersion = '4.13.2'
    espressoVersion = '3.4.0'
    androidxJunitVersion = '1.1.2'
}
```

6. Realice un Sync de su proyecto.

5. Crear una Entidad

Los datos de esta aplicación son palabras y necesitarás una tabla simple para contener esos valores:



Los componentes de la arquitectura le permiten crear una a través de una Entidad . Hagamos esto ahora.

1. Cree un nuevo archivo de clase llamada **Word**. Esta clase describirá la Entidad (que representa la tabla SQLite) para sus palabras. Cada propiedad de la clase representa una columna de la tabla. Room utilizará estas propiedades para crear la tabla y crear instancias de objetos a partir de filas en la base de datos. Aquí está el código:

```
public class Word {

    private String mWord;

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

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

Para que la clase **Word** sea significativa para una base de datos Room, debe anotarla. Las anotaciones identifican cómo cada parte de esta clase se relaciona con una entrada en la base de datos. Room utiliza esta información para generar código.

2. Actualice su clase **Word** con anotaciones como se muestra en este código:

```
@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;}
}
```

Veamos qué hacen estas anotaciones:

- **@Entity(tableName = "word_table")** Cada clase **@Entity** representa una tabla SQLite. Anota tu declaración de clase para indicar que es una entidad. Puede especificar el nombre de la tabla si desea que sea diferente del nombre de la clase. Esta nombra la tabla "word_table".
- **@PrimaryKey** Toda entidad necesita una clave primaria. Para simplificar las cosas, cada palabra actúa como su propia clave principal.
- **@NonNull** Indica que un parámetro, campo o el valor de retorno de un método nunca puede ser nulo.
- **@ColumnInfo(name = "word")** Especifique el nombre de la columna en la tabla si desea que sea diferente del nombre de la variable miembro.
- Cada campo almacenado en la base de datos debe ser público o tener un método "getter". Este ejemplo proporciona un método **getWord().**

Puede encontrar una lista completa de las anotaciones en la Room package summary.

Consulte Defining data using Room entities.

Consejo: Puede generar automáticamente claves únicas anotando la clave principal de la siguiente manera:

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

    @PrimaryKey(autoGenerate = true)
    private int id;

    @NonNull
    private String word;
    //..other fields, getters, setters
}
```

6. Crear el DAO

¿Qué es DAO?

Un DAO (objeto de acceso a datos) valida su SQL en tiempo de compilación y lo asocia con un método. En su Room DAO, utiliza anotaciones útiles, como @Insert, para representar las operaciones de base de datos más comunes. Room utiliza DAO para crear una API limpia para su código.

El DAO debe ser una interface o una clase abstracta. De forma predeterminada, todas las consultas deben ejecutarse en un hilo separado.

Implementar el DAO

Escribamos un DAO que proporcione consultas para:

- Ordenar todas las palabras alfabéticamente
- Insertar una palabra
- Eliminar todas las palabras

1. Cree un nuevo archivo de clase llamado **WordDao**.
2. Copie y pegue el siguiente código en **WordDao** y corrija las importaciones según sea necesario para que compile:

```
@Dao
public interface WordDao {

    // allowing the insert of the same word multiple times by passing a
    // conflict resolution strategy
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insert(Word word);

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

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

Recorramos nuestro código:

- **WordDao** es una interface. Los DAO deben ser interfaces o clases abstractas.
- La anotación **@Dao** la identifica como una clase DAO para Room.
- **void insert(Word word)** declara un método para insertar una palabra:
- La **anotación @Insert** es una anotación especial de método DAO en la que no es necesario proporcionar ningún SQL. (También hay anotaciones **@Delete** y **@Update** para eliminar y actualizar filas, pero no se usan en esta aplicación).
- **onConflict = OnConflictStrategy.IGNORE**: la estrategia de conflicto seleccionada ignora una nueva palabra si es exactamente igual a una que ya está en la lista. Para saber más sobre las estrategias de conflicto disponibles, consulte la **documentación**.

- **deleteAll()**: declara un método para eliminar todas las palabras.
- No existe una anotación conveniente para eliminar varias entidades, por lo que se anota con la genérica **@Query**.
- **@Query("DELETE FROM word_table")**: **@Query** requiere que se le proporcione una consulta SQL como parámetro de tipo string para la anotación.
- **List<Word> getAlphabetizedWords()**: un método para obtener todas las palabras y hacer que devuelva una **List** de **Word**s.
- **@Query("SELECT * FROM word_table ORDER BY word ASC")**: Devuelve una lista de palabras ordenadas en orden ascendente.

Obtenga más información sobre **Room DAOs.**

7. La clase LiveData

Cuando los datos cambian, normalmente desea realizar alguna acción, como mostrar los datos actualizados en la UI. Esto significa que tienes que observar los datos para que, cuando cambien, puedas reaccionar.

Dependiendo de cómo se almacenen los datos, esto puede resultar complicado. Observar los cambios en los datos en múltiples componentes de una aplicación puede crear rutas de dependencia rígidas y explícitas entre los componentes. Esto dificulta las pruebas y la depuración entre otras cosas.

LiveData, es una clase de la **biblioteca lifecycle** para la observación de datos, que resuelve este problema. Utilice un valor de retorno de tipo **LiveData** en la descripción de su método y Room generará todo el código necesario para actualizar **LiveData** cuando se actualice la base de datos.

Nota: Si utilizas **LiveData** de forma independiente de Room, tendrás que gestionar la actualización de los datos. **LiveData** no tiene métodos disponibles públicamente para actualizar los datos almacenados.

Si desea actualizar los datos almacenados dentro de un **LiveData**, debe usar **MutableLiveData** en lugar de **LiveData**. La clase **MutableLiveData** tiene dos métodos públicos que le permiten establecer el valor de un objeto **LiveData**, **setValue(T)** y **postValue(T)**. Por lo general, **MutableLiveData** es usado dentro del **ViewModel**, y luego el **ViewModel** solo expone objetos **LiveData** immutables a los observadores.

En **WordDao**, cambie la firma del método **getAlphabetizedWords()** de tal manera que **List<Word>** este recubierto con **LiveData**:

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

Posteriormente en este Codelab, seguirá los cambios de los datos con un **Observer** en la **MainActivity**.

Vea la documentación de **LiveData** para aprender más acerca de otras maneras de usar **LiveData**, o vea el video **Architecture Components: LiveData and Lifecycle.**

8. Agregar una Base de datos Room

¿Qué es una base de datos Room?

- Room es una capa de base de datos sobre una base de datos SQLite.
- Room se encarga de las tareas mundanas que solías realizar con **SQLiteOpenHelper**.
- Room utiliza el DAO para realizar las consultas a su base de datos.
- De forma predeterminada, para evitar un rendimiento deficiente de la UI, Room no permite realizar consultas en el hilo principal. Cuando las consultas de Room regresan un **LiveData**, las consultas se ejecutan automáticamente de forma asincrónica en un hilo en segundo plano.
- Room proporciona comprobaciones en tiempo de compilación de las sentencias SQLite.

Implementar la base de datos Room

La clase de base de datos Room debe ser abstracta y extender **RoomDatabase**. Por lo general, solo necesitas una instancia de una base de datos de Room para toda la aplicación.

Hagamos una ahora. Cree un archivo de clase llamado **WordRoomDatabase** y agréguele este código:

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

   public abstract WordDao wordDao();

   private static volatile WordRoomDatabase INSTANCE;
   private static final int NUMBER_OF_THREADS = 4;
   static final ExecutorService databaseWriteExecutor =
        Executors.newFixedThreadPool(NUMBER_OF_THREADS);

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

Repasemos el código:

- La clase de base de datos Room debe ser **abstract** y extender **RoomDatabase**
- Usted anota la clase para que sea una base de datos Room con **@Database** y utiliza los parámetros de la anotación para declarar las entidades que pertenecen a la base de datos y establecer el número de versión. Cada entidad corresponde a una tabla que se creará en la base de datos. Las migraciones de bases de datos están fuera del alcance de este Codelab, por lo que aquí configuramos **exportSchema** en falso para evitar una advertencia de compilación. En una aplicación real, debería considerar configurar un directorio

para que Room lo use para exportar el esquema, de modo que pueda verificar el esquema actual en su sistema de control de versiones.

- La base de datos expone los DAO a través de un método "getter" abstracto para cada @Dao.
- Hemos definido un **singleton** para **WordRoomDatabase**, y así evitar que se abran varias instancias de la base de datos al mismo tiempo.
- **getDatabase** devuelve el singleton. Creará la base de datos la primera vez que se acceda a ella, utilizando el builder de Room para crear un objeto **RoomDatabase** en el contexto de la aplicación a partir de la clase **WordRoomDatabase** y le asignará el nombre **"word_database"**.
- Hemos creado un grupo de subprocesos fijo **ExecutorService** que se utilizará para ejecutar operaciones de base de datos de forma asincrónica en un subproceso en segundo plano.

Nota: Cuando modifica el esquema de la base de datos, deberá actualizar el número de versión y definir una estrategia de migración.

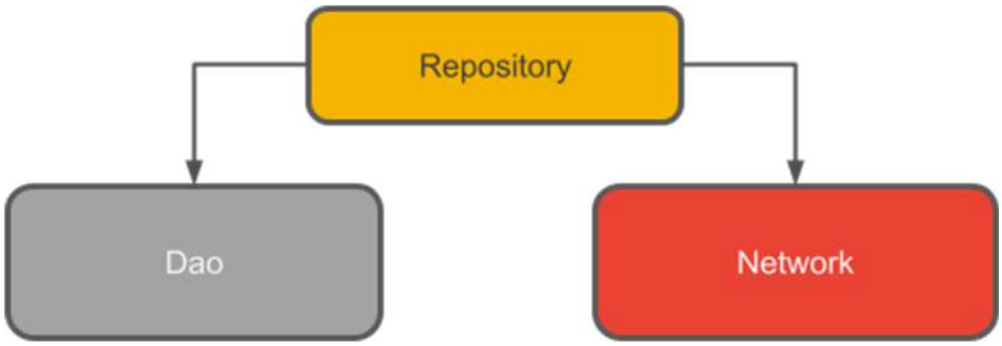
Para este ejemplo, la estrategia de destrucción y recreación puede ser suficiente. Pero, para una aplicación real, debes implementar una estrategia de migración. Consulte **Understanding migrations with Room.**

En Android Studio, si recibe errores al pegar código o durante el proceso de compilación, seleccione **Build > Clean Project**. Luego seleccione **Build > Rebuild Project** y luego vuelva a compilar. Si utiliza el código proporcionado, no debería haber errores cada vez que se le indique que cree la aplicación.

9. Crear el repositorio

¿Qué es un repositorio?

Una clase **Repository** abstrae el acceso a múltiples fuentes de datos. El Repositorio no forma parte de las bibliotecas de Componentes de Arquitectura, pero es una práctica recomendada sugerida para la separación de código y la arquitectura. Una clase **Repository** proporciona una API limpia para acceder a datos del resto de la aplicación.



¿Por qué utilizar un Repositorio?

Un repositorio gestiona las consultas y le permite utilizar múltiples backends. En el ejemplo más común, el Repositorio implementa la lógica para decidir si se obtienen datos de una red o se utilizan los resultados almacenados en caché en una base de datos local.

Implementando el repositorio

Cree un archivo de clase llamada **WordRepository** y pegue el siguiente código en él:

```
class WordRepository {

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

    // Note that in order to unit test the WordRepository, you have to remove
the Application
    // dependency. This adds complexity and much more code, and this sample is
not about testing.
    // See the BasicSample in the android-architecture-components repository at
    // https://github.com/googlesamples
    WordRepository(Application application) {
        WordRoomDatabase db = WordRoomDatabase.getDatabase(application);
        mWordDao = db.wordDao();
        mAllWords = mWordDao.getAlphabetizedWords();
    }

    // Room executes all queries on a separate thread.
    // Observed LiveData will notify the observer when the data has changed.
    LiveData<List<Word>> getAllWords() {
        return mAllWords;
    }
```

```
    // You must call this on a non-UI thread or your app will throw an
exception. Room ensures
    // that you're not doing any long running operations on the main thread,
blocking the UI.
    void insert(Word word) {
        WordRoomDatabase.databaseWriteExecutor.execute(() -> {
            mWordDao.insert(word);
        });
    }
}
```

Las principales conclusiones son:

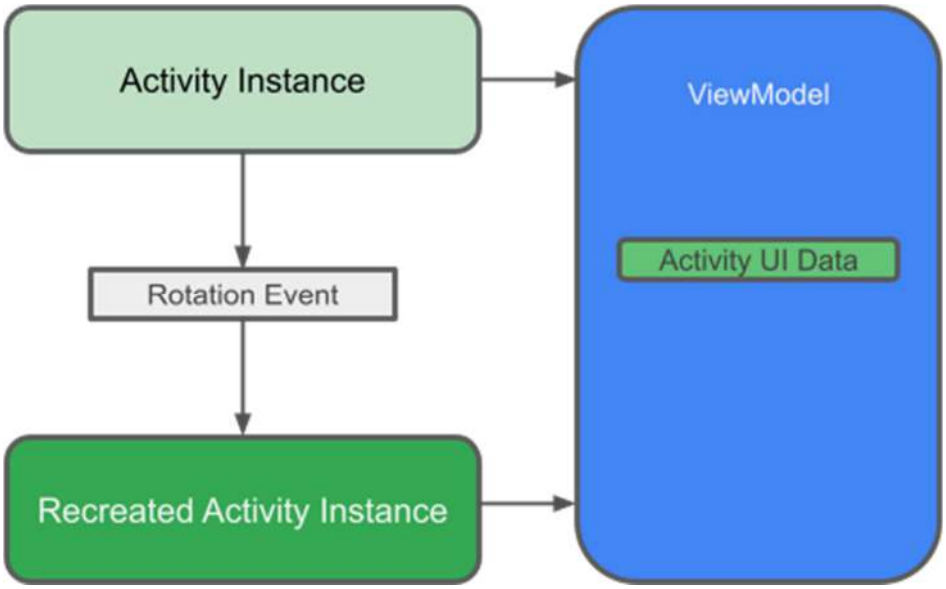
- El DAO se pasa al constructor del repositorio en lugar de toda la base de datos. Esto se debe a que solo necesita acceso al DAO, ya que contiene todos los métodos de lectura/escritura para la base de datos. No es necesario exponer toda la base de datos al repositorio.
- El método **getAllWords** devuelve la lista de palabras de Room como un **LiveData**. Podemos hacerlo debido a la manera en que definimos el método **getAlphabetizedWords** el cual devuelve un LiveData en la etapa "La clase LiveData". Room ejecuta todas las consultas en un hilo separado. Luego, el **LiveData** observado notificará al observador en el hilo principal cuando los datos hayan cambiado.
- No necesitamos ejecutar la inserción en el hilo principal, de modo que usamos el **ExecutorService** que creamos en **WordRoomDatabase** para realizar la inserción en un hilo en segundo plano.

Los repositorios están destinados a mediar entre diferentes fuentes de datos. En este ejemplo sencillo, sólo tienes una fuente de datos, por lo que el Repositorio no hace mucho. Consulte **BasicSample** para ver una implementación más compleja.

10. Crear el ViewModel

¿Qué es un ViewModel?

La función del **ViewModel** es proporcionar datos a la interfaz de usuario y sobrevivir a los cambios de configuración. Un **ViewModel** actúa como un centro de comunicación entre el Repositorio y la UI. También puede utilizar un **ViewModel** para compartir datos entre fragmentos. **ViewModel** es parte de la **lifecycle library**.



Para obtener una guía introductoria a este tema, consulte **ViewModel Overview** o la publicación de blog **ViewModels: A Simple Example.**

¿Por qué utilizar un ViewModel?

Un **ViewModel** mantiene los datos de la UI de su aplicación de una manera consciente del ciclo de vida de tal manera que sobrevive a los cambios de configuración. Separar los datos de la UI de la aplicación de las clases de los Activity y de los Fragment permiten cumplir mejor el principio de responsabilidad única: las actividades y fragmentos son responsables de dibujar datos en la pantalla, mientras que el **ViewModel** pueden encargarse de almacenar y procesar todos los datos necesarios para la UI.

En el **ViewModel**, utilíce los **LiveData** para aquellos datos modificables que la UI usará o mostrará. El uso de **LiveData** tiene varios beneficios:

- Puede poner un observador en los datos (en lugar de sondear los cambios) y actualizar la interfaz de usuario solo cuando los datos realmente cambien.
- El repositorio y la UI están completamente separadas por el **ViewModel**.
- No hay llamadas a la base de datos desde el **ViewModel**(todo esto se maneja en el Repositorio), lo que hace que el código sea más comprobable.

Implementar un ViewModel

Cree un archivo de clase **WordViewModel** y agréguele este código:

```
public class WordViewModel extends AndroidViewModel {

   private WordRepository mRepository;

   private final 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); }
}
```

Esto es lo que tenemos:

- Creó una clase llamada **WordViewModel** que obtiene una Application como parámetro y extiende **AndroidViewModel**.
- Se agregó una variable miembro privada para contener una referencia al repositorio.
- Se agregó un método **getAllWords()** para devolver una lista de palabras almacenadas en caché.
- Implemento un constructor que crea el **WordRepository**.
- En el constructor, inicializó el **LiveData** allWords usando el repositorio.
- Creó un método **insert()** el cual es un wrapper que llama al método insert() del Repositorio. De esta manera, la implementación de insert() se encapsula de la UI.

Advertencia : ¡No mantengas una referencia a un contexto que tenga un ciclo de vida más corto que tu ViewModel! Ejemplos son:

- Activity
- Fragment
- View

Mantener una referencia puede causar una pérdida de memoria, como ser que el ViewModel tenga una referencia a una actividad destruida! Todos estos objetos pueden ser destruidos por el sistema operativo y recreados cuando hay un cambio de configuración, y esto puede suceder muchas veces durante el ciclo de vida de un ViewModel.

Si necesita el contexto de la aplicación (que tiene un ciclo de vida que dura tanto como la aplicación), use **AndroidViewModel**, como se muestra en este codelab.

Importante: los **ViewModel** no sobreviven a que el proceso de la aplicación se cierre en segundo plano cuando el sistema operativo necesita más recursos. Para los datos de la UI que necesitan sobrevivir a la muerte del proceso debido a la falta de recursos, puede utilizar el Saved State module for ViewModels . Obtenga más información aquí .

Para obtener más información sobre las clases ViewModel, mire el video Componentes de arquitectura: ViewModel .