

Persistencia de datos en Android con Room

Por Alejandro Platero Recio - 4 marzo, 2019

22849 | 5

[Twitter](#) | [Twitter](#) [LinkedIn](#) | [LinkedIn](#) [Facebook](#) | [Facebook](#) [Reddit](#)



Índice de contenidos

- [1. Introducción](#)
- [2. Entorno](#)
- [3. SQLite](#)
 - [3.1. ¿Qué es SQLite?](#)
 - [3.2. Inconvenientes de SQLite](#)
- [4. Room](#)
 - [4.1. ¿Qué es Room?](#)
 - [4.2. Arquitectura](#)
 - [4.3. Anotaciones](#)
- [5. Ejemplo de utilización de Room](#)
 - [5.1. Incorporando Room a nuestro proyecto](#)
 - [5.2. Creando la Entity](#)
 - [5.3. Creando el DAO](#)
 - [5.4. La RoomDatabase](#)
 - [5.5. La clase Repository](#)
 - [5.6. Accediendo a Room desde el resto de la app](#)
- [6. Referencias](#)

1. Introducción

Con este tutorial aprenderás a usar Room, una librería para manejar bases de datos SQLite en Android de una manera más segura. Además desarrollaremos un ejemplo utilizando Kotlin.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- **Hardware:** MacBook Pro 17' (2,66 GHz Intel Core i7, 8GB DDR3)
- **Sistema operativo:** macOS Sierra 10.13.6
- **Entorno de desarrollo:** Android Studio 3.3
- **Versión SDK mínima:** 16

3. SQLite

3.1. SQLite

SQLite es un sistema de dominio público de gestión de bases de datos relacionales. La principal ventaja que presenta es que no funciona como un proceso independiente, sino que forma parte de la aplicación que lo utiliza. Por tanto, no necesita ser instalado independientemente, ejecutado o detenido, ni tiene fichero de configuración. Además sigue los principios ACID.

En Android es bastante útil, ya que permite la utilización de una base de datos **local** en el dispositivo que utilice nuestra aplicación de una manera relativamente ligera y sencilla.

3.2. Inconvenientes

SQLite es relativamente de bajo nivel, por lo que presenta ciertos riesgos. Su implementación requiere tiempo y esfuerzo, ya que se deben escribir las sentencias SQL de la base de datos. Por ello, si el modelo de datos sufre cambios, tendremos que modificar estas sentencias manualmente, con el riesgo que ello implica. Por si esto no fuera poco, estas sentencias no se comprueban durante la compilación, por lo que hay un importante riesgo de errores en tiempo de ejecución.

4. Room

4.1. Room

Room es una librería que abstrae el uso de SQLite al implementar una capa intermedia entre



esta base de datos y el resto de la aplicación. De esta forma se evitan los problemas de SQLite sin perder las ventajas de su uso.

Room funciona con una arquitectura cuyas clases se marcan con anotaciones preestablecidas. Por otro lado, la mayoría de las consultas a la base de datos sí se comprueban en tiempo de compilación.

4.2. Arquitectura

Las partes de las que se compone Room son las siguientes:

- **Entity:** son clases que definen las tablas de la base de datos y de las entidades a utilizar.
- **DAO:** interfaces que definen los métodos utilizados para acceder a la base de datos.
- **RoomDatabase:** sirve de acceso a la base de datos SQLite a través de los DAOs definidos.

Además, es recomendable utilizar una clase intermedia a la cual denominamos Repository cuya finalidad es administrar las diferentes fuentes de datos.

4.3. Anotaciones

Para que se detecten qué clases tendrán que ser tratadas por esta librería y para indicar ciertas configuraciones debemos utilizar anotaciones. Las principales son las siguientes:

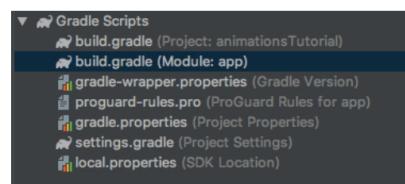
- **@Database:** para indicar que la clase será el Database. Además, dicha clase debería ser abstracta y heredar de RoomDatabase.
- **@Dao:** se utiliza para las interfaces de los DAOs.
- **@Entity:** indica que la clase es una entidad.
- **@PrimaryKey:** indica que el atributo al que acompaña será la clave primaria de la tabla. También podemos establecer que se asigne automáticamente si la incluimos así: `@PrimaryKey(autoGenerate = true)`.
- **@ColumnInfo:** sirve para personalizar la columna de la base de datos del atributo asociado. Podemos indicar, entre otras cosas, un nombre para la columna diferente al del atributo.
- **@Ignore:** previene que el atributo se almacene como campo en la base de datos.
- **@Index:** para indicar el índice de la entidad.
- **@ForeignKey:** indica que el atributo es una clave foránea relacionada con la clave primaria de otra entidad.
- **@Embedded:** para incluir una entidad dentro de otra.
- **@Insert:** anotación para los métodos de los DAOs que inserten en la base de datos.
- **@Delete:** anotación para los métodos de los DAOs que borren en la base de datos.
- **@Update:** anotación para los métodos de los DAOs que actualicen una entidad en la base de datos.
- **@Query:** anotación para un método del DAO que realice una consulta en la base de datos, la cual deberemos especificar.

5. Ejemplo de utilización de Room

Vamos a desarrollar un ejemplo para ver cómo utilizar Room. Para ello vamos a crear una agenda sencilla con contactos y su teléfono.

5.1. Incorporando Room a nuestro proyecto

Para seguir este tutorial, crea un nuevo proyecto de Android con una actividad vacía. Una vez tengamos nuestro proyecto, tenemos que añadir las dependencias de Room para usarlo.



Para ello, abrimos el fichero de propiedades de gradle y lo editamos. Este tutorial se desarrolla con Kotlin, por lo que tendremos que añadir la dependencia de kapt. Además, nuestro ejemplo seguirá la arquitectura MVVM por lo que vamos a añadir las dependencias para el ViewModel y LiveData, aunque no es necesario para utilizar Room. El resultado es el siguiente:

```
1 apply plugin: 'com.android.application'
2 apply plugin: 'kotlin-android'
3 apply plugin: 'kotlin-android-extensions'
4
5 apply plugin: 'kotlin-kapt'
6
7 android {
8     compileSdkVersion 28
9     defaultConfig {
10         applicationId "com.outentia.tutorialroom"
11         minSdkVersion 16
12         targetSdkVersion 28
13         versionCode 1
14         versionName "1.0"
15         testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
16     }
17     buildTypes {
18         release {
19             minifyEnabled false
20             proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
21         }
22     }
23 }
24
```

```

1 dependencies {
2     implementation "android.arch.persistence.room:runtime:1.1.1"
3     kapt "android.arch.persistence.room:compiler:1.1.1"
4
5     implementation "android.arch.lifecycle:extensions:1.1.1" //ViewModel and LiveData
6
7     implementation fileTree(dir: 'libs', include: ['*.jar'])
8     implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
9     implementation 'com.android.support:appcompat-v7:28.0.0'
10    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
11    testImplementation 'junit:junit:4.12'
12    androidTestImplementation 'com.android.support.test.runner:1.0.2'
13    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
14 }

```

5.2. Creando la Entity

Nuestra base de datos tendrá una tabla para los contactos. Por tanto, tenemos que crear una clase de tipo Entity.

```

1 @Entity(tableName = Contact.TABLE_NAME)
2 data class Contact(
3     @ColumnInfo(name = "phone_number") @NotNull val phoneNumber: String,
4     @ColumnInfo(name = "first_name") @NotNull val firstName: String,
5     @ColumnInfo(name = "last_name") val lastName: String? = null
6 ) {
7     companion object {
8         const val TABLE_NAME = "contact"
9     }
10    @PrimaryKey(autoGenerate = true)
11    @ColumnInfo(name = "contact_id")
12    var contactId: Int = 0
13 }
14

```

Como puedes ver, la entidad tiene cuatro atributos y, como ninguno tiene la etiqueta @Ignore, todos serán columnas de la tabla. La clave primaria se autogenerará y sólo el apellido permitirá valores nulos. Además, todos los nombres de las columnas están especificados con la anotación @ColumnInfo.

5.3. Creando el DAO

Como ya se ha indicado, el DAO será una interfaz que especifica los métodos con los que accederemos a la entidad en la base de datos. Aunque en nuestro caso sólo vamos a insertar y listar todos los elementos, también tienes las funciones para borrar y modificar. También hay que fijarse en que el método getOrderedAgenda() devuelve un objeto de tipo LiveData. Esto no es obligatorio, pudiendo devolver un Array o una Lista, pero como nuestro ejemplo seguirá una arquitectura MVVM vamos a hacerlo así.

```

1 @Dao
2 interface ContactDao {
3     @Insert
4     fun insert(contact: Contact)
5
6     @Update
7     fun update(vararg contact: Contact)
8
9     @Delete
10    fun delete(vararg contact: Contact)
11
12    @Query("SELECT * FROM " + Contact.TABLE_NAME + " ORDER BY last_name, first_name")
13    fun getOrderedAgenda(): LiveData<List<Contact>>
14 }

```

5.4. La RoomDatabase

Nuestra database será abstracta y seguirá el patrón singleton para que sea compartida por cualquier objeto que la utilice. Definimos una función que devolverá el DAO que queremos y en el método getInstance ordenaremos a Room que inicialice la instancia de la database si es null y luego la devolveremos.

```

1 @Database(entities = [Contact::class], version = 1)
2 abstract class ContactsDatabase : RoomDatabase() {
3     abstract fun contactDao(): ContactDao
4
5     companion object {
6         private const val DATABASE_NAME = "score_database"
7         @Volatile
8         private var INSTANCE: ContactsDatabase? = null
9
10        fun getInstance(context: Context): ContactsDatabase? {
11            INSTANCE ?: synchronized(this) {
12                INSTANCE = Room.databaseBuilder(
13                    context.applicationContext,
14                    ContactDatabase::class.java,
15                    DATABASE_NAME
16                ).build()
17            }
18            return INSTANCE
19        }
20    }
21 }
22

```

5.5. La clase Repository

Nuestro repositorio accederá a la base de datos para recuperar el DAO de los contactos y tendrá dos métodos, uno para insertar y otro para recuperar el LiveData. Además, implementaremos una clase privada para poder ejecutar la llamada de inserción en un hilo independiente, ya que no se permite realizarla en el hilo principal.

```

1 class ContactRepository(application: Application) {
2     private val contactDao: ContactDao = ContactsDatabase.getInstance(application)?.contact
3
4     fun insert(contact: Contact) {
5         if (contactDao != null) InsertAsyncTask(contactDao).execute(contact)
6     }
7
8     fun getContacts(): LiveData<List<Contact>> {
9
10    }
11 }
12

```

```

9     return contactDao?.getOrderedAgenda() ?: MutableLiveData<List<Contact>>()
10    }
11
12    private class InsertAsyncTask(private val contactDao: ContactDao) :
13        AsyncTask<Contact, Void, Void> {
14        override fun doInBackground(vararg contacts: Contact?): Void {
15            for (contact in contacts) {
16                if (contact != null) contactDao.insert(contact)
17            }
18            return null
19        }
20    }
21 }

```

5.6. Accediendo a Room desde el resto de la app

Llegados a este punto, ya tenemos Room implementado, por lo que ahora vamos a desarrollar el resto de la aplicación para acceder a la base de datos y manipular su contenido. Para empezar, vamos a desarrollar el View Model, que instanciará la clase ContactsRepository para recuperar el LiveData e insertar contactos.

```

1 class ContactsViewModel(application: Application) : AndroidViewModel(application) {
2     private val repository = ContactsRepository(application)
3     val contacts = repository.getContacts()
4
5     fun saveContact(contact: Contact) {
6         repository.insert(contact)
7     }
8 }

```

Para seguir, modificaremos el layout activity_main.xml que encontramos dentro de la carpeta res/layout, en el cual incluiremos tres campos de texto y un botón para añadir contactos. Por otro lado, tendremos un TextView para poder mostrar los contactos, aunque también podríamos hacerlo por ejemplo con un ListView. El contenido debe quedar así:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:app="http://schemas.android.com/apk/res-auto"
4     xmlns:tools="http://schemas.android.com/tools"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     tools:context=".MainActivity">
8
9     <EditText
10         android:id="@+id/fistName_editText"
11         android:layout_width="wrap_content"
12         android:layout_height="wrap_content"
13         android:ems="10"
14         android:inputType="textPersonName"
15         android:hint="Nombre"
16         app:layout_constraintLeft_toLeftOf="parent"
17         app:layout_constraintRight_toRightOf="parent"
18         app:layout_constraintTop_toTopOf="parent" />
19
20     <EditText
21         android:id="@+id/lastName_editText"
22         android:layout_width="wrap_content"
23         android:layout_height="wrap_content"
24         android:ems="10"
25         android:inputType="textPersonName"
26         android:hint="Apellido"
27         app:layout_constraintLeft_toLeftOf="parent"
28         app:layout_constraintRight_toRightOf="parent"
29         app:layout_constraintTop_toBottomOf="@+id/fistName_editText" />
30
31     <EditText
32         android:id="@+id/phone_editText"
33         android:layout_width="wrap_content"
34         android:layout_height="wrap_content"
35         android:ems="10"
36         android:inputType="textPersonName"
37         android:hint="Teléfono"
38         app:layout_constraintLeft_toLeftOf="parent"
39         app:layout_constraintRight_toRightOf="parent"
40         app:layout_constraintTop_toBottomOf="@+id/lastName_editText" />
41
42     <Button
43         android:id="@+id/addContact_button"
44         android:layout_width="wrap_content"
45         android:layout_height="wrap_content"
46         android:text="Añadir"
47         app:layout_constraintLeft_toLeftOf="parent"
48         app:layout_constraintRight_toRightOf="parent"
49         app:layout_constraintTop_toBottomOf="@+id/phone_editText" />
50
51     <TextView
52         android:id="@+id/contacts_textView"
53         android:layout_width="wrap_content"
54         android:layout_height="wrap_content"
55         app:layout_constraintLeft_toLeftOf="parent"
56         app:layout_constraintRight_toRightOf="parent"
57         app:layout_constraintTop_toBottomOf="@+id/addContact_button"
58         android:gravity="center" />
59 </android.support.constraint.ConstraintLayout>

```

Por último, vamos con la clase MainActivity. Esta clase debe observar el LiveData del ViewModel para mostrar los cambios y añadir un listener al botón para añadir el contacto cuando se pulse.

```

1 class MainActivity : AppCompatActivity() {
2     private lateinit var contactsViewModel: ContactsViewModel
3
4     override fun onCreate(savedInstanceState: Bundle?) {
5         super.onCreate(savedInstanceState)
6         setContentView(R.layout.activity_main)
7
8         contactsViewModel = run {
9             ViewModelProviders.of(this).get(ContactsViewModel::class.java)
10        }
11
12         addContact_button.setOnClickListener { addContact() }
13         addObserver()
14     }
15
16     private fun addObserver() {
17         val observer = Observer<List<Contact>> { contacts ->
18             if (contacts != null) {
19                 var text = ""
20                 for (contact in contacts) {
21                     text += contact.name + " "
22                 }
23                 contacts_textView.text = text
24             }
25         }
26         contactsViewModel.contacts.observe(this, observer)
27     }
28 }

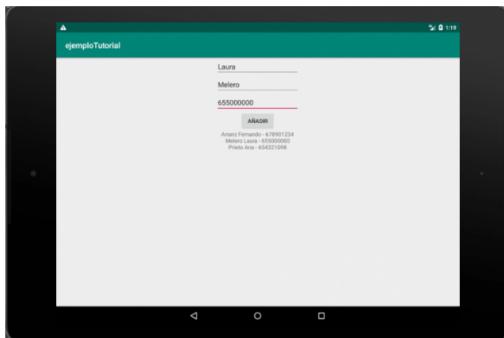
```

```

21         text += contact.lastName + " " + contact.firstName + " - " + contact.phone
22     }
23     contacts_textView.text = text
24   }
25 }
26 contactsViewModel.contacts.observe(this, observer)
27 }
28
29 private fun addContact() {
30     val phone = phone_editText.text.toString()
31     val name = firstName_editText.text.toString()
32     val lastName =
33         if (lastName_editText.text.toString() != "") lastName_editText.text.toString()
34         else null
35
36     if (name != "" && phone != "") contactsViewModel.saveContact(Contact(phone, name, la
37 }
38 }

```

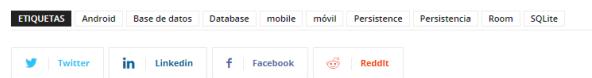
Con esto hemos terminado el tutorial, por lo que podemos ejecutar nuestra aplicación y añadir nuevos contactos que se mostrarán debajo del botón.



¡Muchas gracias por haber leído hasta aquí!

6. Referencias

<http://www.sqlitetutorial.net/what-is-sqlite/>
<https://developer.android.com/reference/androidx/room/Room>
<https://www.sqlite.org/index.html>
<https://developer.android.com/training/data-storage/room/>



Esta obra está licenciada bajo [licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)



Alejandro Platero Recio
Consultor tecnológico de desarrollo de proyectos informáticos
Graduado en Ingeniería del Software y Técnico superior en desarrollo de aplicaciones web
Puedes encontrarme en [Autentia](#): Ofrecemos servicios de soporte a desarrollo, factoría de formación.
Somos expertos en Java/Java EE.

[in](#) [e-mail](#) [Twitter](#)

Artículo relacionados

Más del autor

Tutoriales	Tutoriales	Tutoriales	Tutoriales	Tutoriales
10 octubre, 2022 Cobertura en un proyecto maven multimódulo con JaCoCo Dionisio Cortés Fernández	7 octubre, 2022 OKRs – From Zero To Hero Fernando Bogas Pomares	3 octubre, 2022 Como instalar una instancia de GitLab Andrés Urraca Barredo	27 septiembre, 2022 Despliegue de Aplicaciones sobre Kubernetes Nicolae Alexandru Molnar	27 septiembre, 2022 Event storming Tautvydas Bagocius
 				

Dejar respuesta

Comentario:

Nombre:^{*}Correo electrónico:^{*} He leído la [política de privacidad](#) y acepto que se almacenen mis datos para recibir respuestas por correo electrónico. He leído la [política de privacidad](#) y acepto recibir la newsletter con las últimas novedades vía email.**Publicar comentario****5 Comentarios****Brayán Calderón** 8 agosto, 2019 at 8:15 pm

Genial, voy a probarlo, muchas gracias!

[Responder](#)**Johnny Young** 13 septiembre, 2019 at 7:12 pm

Muy bien explicado, felicitaciones y gracias

[Responder](#)**Eduardo Mejía** 30 marzo, 2020 at 4:22 am

Excelente tutorial!!!

[Responder](#)**Leandro Flórez Aristizábal** 21 mayo, 2020 at 6:44 am

Hola, muy bueno el tutorial, bastante simple la explicación, esto es lo que uno espera. Tengo una duda, en el repositorio por qué no creaste métodos para borrar contactos? Sería bueno explicar cómo se haría la eliminación de un contacto o la actualización. Pero la duda principal es si no de debe incluir métodos de eliminar en el repositorio, he visto varios ejemplos que no lo hacen y no sé si es porque no se debe. Muchas gracias

[Responder](#)**Ismael Cruz** 28 mayo, 2020 at 3:13 am

Gracias por compartirlo, excelente trabajo me ayudo bastante

[Responder](#)

Portal de tutoriales de tecnología y programación donde escriben profesionales en activo.

Contáctanos:
adictos@adictosaltrabajo.com

**Menú**

[Home](#)
[Noticias](#)
[Tutoriales](#)
[Reviews](#)
[Autores](#)
[Participa](#)

Envíanos tu tutorial

En Adictosaltrabajo.com cualquier persona puede aportar conocimiento a la Comunidad tecnológica. Ya somos más de 150 autores compartiendo conocimiento.

¿Te animas?

PARTICIPA*Powered by autentia*

Este sitio web utiliza cookies para mejorar tu experiencia y analizar nuestro tráfico web, de forma anónima, sin compartir datos con terceros. Al utilizar Adictos al trabajo, aceptas nuestra [política de cookies](#).

[Rechazar](#)[Aceptar](#)