{desafío} latam_

Persistencia y bases de datos - Parte II

Room

Competencias:

- Entender que es un ORM, y cómo interactúa con una base de datos
- Definir que es Room, y qué papel juega en el paquete de Architecture Components
- Utilizando Room para construir una base de datos, tablas y DAO para persistir datos en una aplicación.
- Describir las ventajas de utilizar Room sobre SQL en una aplicación.

Motivación

Una de las necesidades más recurrentes de una aplicación móvil es el almacenamiento de datos estructurados. La opción más común para manejar datos estructurados en Android es SQLite. El principal problema con SQLite es la generación de consultas. Lamentablemente no podemos saber si una de estas consultas presenta algún error hasta que la ocupemos en nuestro código, es decir hasta que corramos la aplicación en un dispositivo y utilicemos la consulta. Estos errores son conocidos como Run Time Error, lo cuales sólo se presentan en tiempo de ejecución. Cuando utilizamos SQLite directamente debemos manejar estos errores considerando los distintos escenarios, o simplemente hacer prueba y error para corregir, lo que hace el proceso de desarrollo lento y tedioso. Para evitar estos errores, y simplificar la interacción con SQLite, Google crea Room, una librería que permite generar una capa de interacción con la base de datos, que además identifica los errores en tiempo de compilación, evitando los errores en tiempo de ejecución. Room es parte de Jetpack. Veremos toda la teoría detrás de Room, con un ejemplo donde construiremos las bases de un aplicación y posteriormente completamos la aplicación y veremos ejemplos más avanzados del uso de Room en Android.

¿Por qué Room?

Room nos permite crear una capa de abstracción sobre SQLite para facilitar el acceso a los datos, y la interacción con dicha Base de Datos. Una aplicación que maneja grandes cantidades de data estructurada se benefician directamente con la implementación de room.

Para entender bien los beneficios veamos un ejemplo teórico de cómo podríamos modelar una libreta de contactos.

Libreta de Contactos

Necesito crear una aplicación que maneje contactos telefónicos, normalmente esto lo podemos modelar en una aplicación orientada a objetos con una clase que represente a la Entidad Persona, esta clase será la base de los datos que maneja nuestra aplicación. Nuestros contactos tendrán las siguientes características:

- Nombre y Apellido
- Número telefónico
- Dirección, donde almacenaremos los siguientes datos:
  - Nombre calle
  - Número de la calle
  - Comuna
  - Región
- Correo electrónico

Las clases necesarias para este modelo de datos se pueden definir de la siguiente manera en Kotlin:

```
class Person(var name:String,
      var surname:String,
      var phonenumber:String,
      var address: Address,
      var email:String)
```

```
class Address(var streetName:String,
       var streetNumber:Int,
       var district: String,
       var region:String)
```

Tenemos dos clases, una representa a una persona, que contiene un conjunto de datos, además de una segunda clase que representa una dirección. Podemos discutir cómo modelar esto de diferentes manera, pero por ahora esta será nuestra definición.

La pregunta es: ¿cómo almacenamos esto en nuestro dispositivo?.

Respuesta: SQLite

Debemos pensar cómo son las tablas que necesitamos, cuáles son las id primarias, cómo se modela, conecta e interpreta la información entre las tablas y los objetos que definimos anteriormente, etc. Lo anterior es bastante trabajo y genera una gran cantidad de código. Necesitamos crear todo el código que crea la base de datos, las migraciones, las tablas, un DAO para interactuar con la base de datos, etc. A todo este código necesario, pero que genera mucho trabajo extra para un beneficio pequeño, se le conoce como boilerplate code.

Normalmente este boilerplate code, es bastante trabajo y genera muchos problemas de mantenimiento, cada vez que hacemos un cambio necesitamos actualizar todas las clases involucradas, lo cual puede generar más de algún error.Normalmente para evitar tener que escribir todo este código, se puede utilizar una herramienta. En este caso un ORM.

Room como ORM

Un Object Relational Mapper, ORM, es una herramienta que genera todo este código intermedio, boilerplate code, que une o pega nuestra aplicación con su base de datos.

Los ORM no son invención de Android, son ocupados hace mucho tiempo, y tenemos a nuestra disposición muchas opciones. La cantidad de opciones generó mucha fragmentación y limitaciones. Además, las aplicaciones tienen cada vez ciclos de vida más complejos, lo que hace difícil tomar una decisión de cuál elegir. Si elegimos de mala manera, y la arquitectura de nuestra aplicación no es la adecuada, cambiar de ORM puede ser un dolor de cabeza mayor.

Como respuesta a la fragmentación, errores comunes, y como parte de su cruzada por mejores aplicaciones con mejor arquitectura, Google crea Room. Room es parte de Jetpack, y se recomienda como uno de los componentes de una arquitectura por capas tipo MVVM o MVP. Room es parte de la capa de datos, que se recomienda implementar como un repositorio. Conceptualmente podemos entender Room como el mediador entre la Base de datos SQLite y la capa de acceso de datos de la aplicación, pudiendo esta ser implementada como repositorio u otro tipo de patrón arquitectónico.

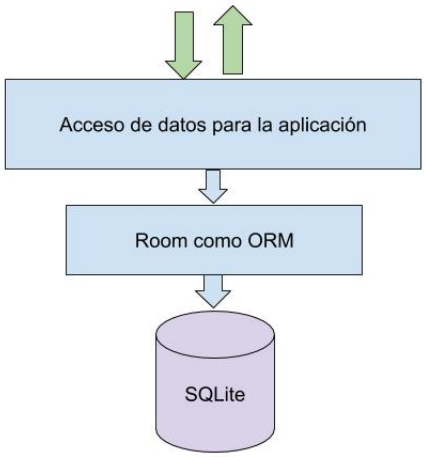

Imagen 1. Room actúa como intermediario entre SQLite y la capa de acceso a datos

Componentes de Room

Para utilizar Room debemos generar sus tres componentes principales:

- **Entity**: Representa un data model, o modelo de datos, que se mapea directamente a las tablas de la base de datos.
- **DAO**: Data Access Object, este objeto nos permite interactuar con la base de datos. Normalmente la interacción es del tipo CRUD (Create, Read, Update and Delete)
- **Database**: Un contenedor que mantiene una referencia y actúa como único punto de acceso a la base de datos.

Para explicar cómo funcionan, y qué papel juegan, cada uno de estos componentes en Room, definamos un problema a resolver. Necesitamos almacenar en un dispositivo un conjunto de recomendaciones, en una especie de libro de apuntes o notas. Pensemos en la libreta que tenía el capitán américa en Winter Soldier, veamos una imagen:

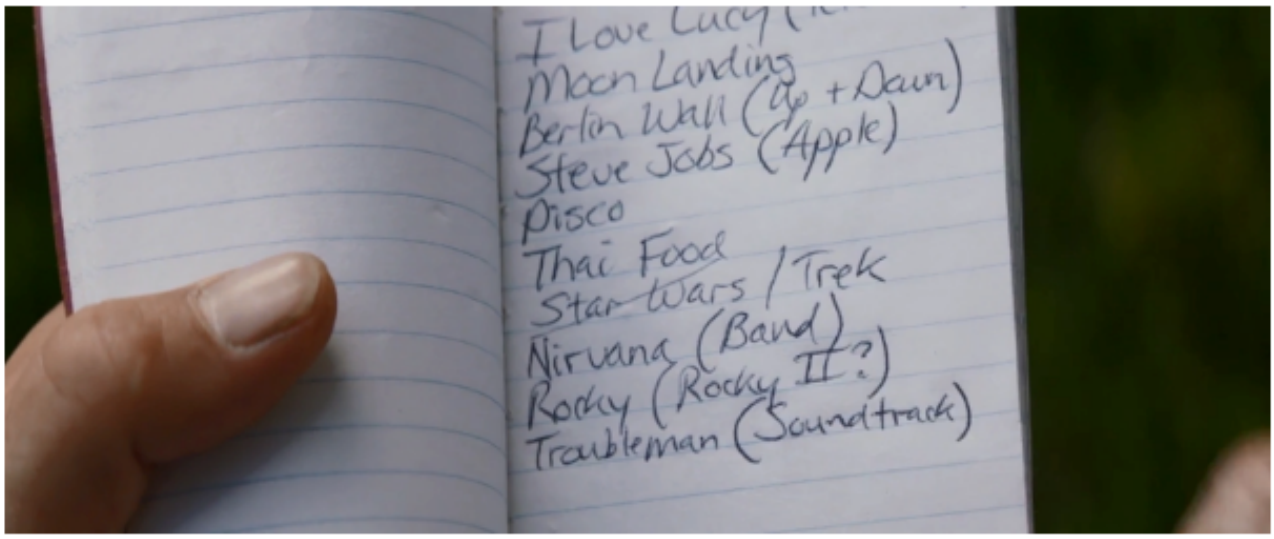

Imagen 2. Libreta.

Esta aplicación ficticia nos permitirá agregar información, o recomendaciones para ponernos al día, y guardarlas persistentemente en nuestro dispositivo. La aplicación tiene una pantalla donde vemos la lista, con un botón que nos permite agregar información. Un ejemplo de la pantalla en la siguiente imagen:

Imagen 3. Hola Mundo.

El objetivo de este ejercicio teórico es crear un aplicación, siguiendo los diferentes pasos del documento, pero en este capítulo veremos toda la parte teórica de Room, sus anotaciones y componentes. En el siguiente capítulo completamos la aplicación aquí descrita. Utilizaremos este ejemplo para definir los diferentes componentes de Room, cómo interactúan y que necesitamos en el código para cada uno. Vamos a partir configurando Room en un proyecto, para este paso se puede utilizar cualquier proyecto inicial, para crear un proyecto inicial puede seguir los pasos descritos en el capítulo dos para la aplicación de ejemplo de SharedPreferences. Veamos cómo se agrega Room a un proyecto.

Configuración de Room en un proyecto Android

Para tener acceso a la librería y todas sus herramientas, debemos configurar las dependencias necesarias en nuestro Gradle file del módulo. Cómo estamos usando Kotlin, debemos cerciorarnos de las siguientes cosas:

En el gradle principal del proyecto debemos agregar las dependencias de kotlin

```
buildscript {
    ext.kotlin_version = '1.3.40' //versión de kotlin definida como variable
    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.4.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

En el gradle file del módulo debemos tener las siguientes configuraciones:

- En el tope del archivo debemos agregar los siguientes plugin

```
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'
```

- Y en la parte de dependencias debemos agregar lo siguiente

```
def room_version = "2.1.0"
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
```

Notar que como estamos usando Kotlin, debemos utilizar kapt para la dependencia de room, en vez de annotationProcessor que es lo normal en java. Con estas configuraciones agregadas a nuestros gradle files, estamos listos para utilizar Room en nuestro proyecto. Comencemos por definir la primera parte de nuestro puzzle, definamos nuestro Entity.

Entity

Es el primer componente que debemos definir para nuestra aplicación. Un Entity, es un objeto que nos ayuda a mantener y modelar datos dentro de nuestra aplicación. Estos datos pueden estar almacenados o ser manejados en memoria. Un Entity modela con un objeto el tipo de datos que luego almacenaremos en la base de datos. Además, a este “modelo” le agregamos información para describir cómo construir la tabla en la base de datos. Para nuestro ejemplo definimos un set de datos y una tabla con columnas, considerando lo siguiente:

- Almacenaremos un texto con la recomendación
- Almacenaremos un id para mantener ordenados nuestros datos, este id será numérico y auto incrementado por la base de datos.
- Nuestra base de datos sólo tendrá una tabla, con dos columnas, por lo que hemos definido anteriormente.

Resolver esto en SQLite no es trivial, pero gracias a las anotaciones y la simplicidad de Room, podemos implementar un Entity que cumple con todos estos requerimientos de la siguiente manera:

```
@Entity(tableName = "recommendations_list")
data class RecommendationEntity(@ColumnInfo(name = "id")
                @PrimaryKey(autoGenerate = true)
                val id:Long = 0,
                @ColumnInfo(name = "recommendation_text")
                val recommendation:String)
```

Varias cosas debemos entender de este código. Primero, la clase es una data class, algo particular de Kotlin y que no es parte de Java. Este tipo de clases nos permiten definir modelos o POJOS de datos estructurados, también conocidos como objetos de datos. Estas clases son utilizadas como contenedores de información, y nos permiten mover la misma a través de los diferentes componentes de la aplicación, cuando los objetos transportan la información se conocen como DTO o Data Transfer Object.

Nuestro Entity está definido en una data class que contiene la información que nuestra aplicación necesita y almacenará en la base de datos. Entrando en el detalle de esta clase, tenemos los siguientes componentes:

- **Entity**: es una anotación de Room. Nos permite definir que la data class anotada es un Entity. Dentro de la anotación definimos que Entity almacenará la información en una tabla llamada “recommendations_list”. Está tabla tiene dos columnas. Para señalar que es una columna ocupamos la anotación que sigue. Cabe destacar que Entity puede inferir los nombres de las columnas, ocupando los nombres de las variables en la clase, pero cuando utilizamos camelCase, o deseamos un nombre específico, podemos definir el mismo dentro de la anotación

- **ColumnInfo.ColumnInfo**: Esta anotación nos permite definir que la variable anotada es una columna de la tabla antes nombrada. Dentro de la anotación definimos el nombre de la columna en la tabla. En este caso tenemos dos variables anotadas como columnas:
    - id: es el identificador de la fila en la tabla, definimos un valor inicial de 0 para el id.
    - recommendation_text: la columna que contiene texto de la recomendación.
- **PrimaryKey**: Esta anotación nos permite definir la clave primaria de cada fila en nuestra tabla. En este caso hemos definido id como clave primaria. También definimos que se autoincrementa dentro de la anotación.

De esta forma hemos definido todo lo que necesitamos en nuestra Entity. Notemos lo simple que ha resultado crear una tabla, con sus columnas, y los ids autoincrementales. Sigamos definiendo los otros dos componentes que nos faltan para completar este puzzle.

DAO

Para que nuestra aplicación ficticia pueda interactuar con la base de datos necesitamos un objeto que nos de acceso, y actúe como intermediario entre las tablas y los objetos. Este objeto, o clase, se llama DAO, o DataBase Access Object.

Este objeto se implementa como una interfaz, que luego es creado por Room, con el patrón Fachada, se explicará más adelante. Un DAO nos permite ejecutar diferentes interacciones contra nuestra base de datos. Normalmente nos permite realizar las operaciones CRUD sobre nuestra base de datos. Podemos Crear, escribir, actualizar y borrar una fila/tabla en la base de datos. También nos permite consultar por los datos almacenados, generar otras consultas, etc.

Para nuestro ejemplo necesitamos almacenar recomendaciones, insertar en la base de datos las que ingresamos manualmente. Además, debemos poder leer todas las recomendaciones almacenadas en la tabla. Para realizar estas operaciones en Room necesitamos un DAO. Implementar esto con Room es bastante sencillo. Para generar un DAO debemos utilizar una serie de anotaciones, veamos como quedaría un DAO en nuestro ejemplo:

```
@Dao
interface RecommendationsDAO {
 @Query("SELECT * FROM recommendations_list")
 fun getAllRecommendations(): List<RecommendationEntity>

 @Insert
 fun insertRecommendations(var recommendations:List<RecommendationEntity>)
}
```

Lo primero que se debe notar es que DAO es una interfaz y no una clase. Esto se debe a que Room ocupa un interesante patrón llamado Fachada. Este patrón nos permite definir la interfaz y dejar que la librería construya todo el código necesario para hacer funcionar esa interfaz, en este caso un DAO. Este comportamiento de Interfaz/Fachada es usado en muchas librerías populares, como Dagger o Retrofit, por lo cual Google decidió aprovechar la familiaridad que tiene la comunidad con el patrón y lo utilizó para construir Room.

Volvamos al código.

Comenzaremos mirando las anotaciones de la clase, la primera que vemos sobre la definición de la Interfaz es la anotación `@Dao`. Esta anotación le dice a Room que esta interfaz es un DAO a una base de datos. Además, le señala que debe implementar los métodos definidos en ella.

La segunda anotación que vemos en este código es `@Query`, la cual nos permite definir una query SQL que se ejecuta en la base de datos. Aquí podemos definir operaciones tan complejas como deseemos. La principal ventaja de esta anotación por sobre SQL puro es: las queries SQL en Room son revisadas en tiempo de compilación. Al realizar la revisión en la compilación, podemos saber de manera inmediata donde nos estamos equivocando, antes de correr la aplicación, o si nos falta algún parámetro en la consulta. Room construye la base de datos y cuando compila sabe de manera inmediata si la operación, o query, compila correctamente contra la base de datos definida.

Esto nos ahorra muchos dolores de cabeza, con SQL uno corre la aplicación y ejecutar la query en Runtime, esperando que estuviera correcta. Si la query falla, sólo lo sabremos después de correr la aplicación, con Room lo sabemos antes de correr la aplicación.

Dentro de la anotación Query en la primera función de este DAO definimos una consulta, en este caso la consulta es un *SELECT * FROM recommendations_list,* que nos permite obtener todas las filas almacenadas en la tabla recommendations list. Con esta lista podremos desplegar todas las recomendaciones al usuario.Este método no recibe parámetros, después de todo es un select *, nos retorna una lista de entidades RecommendationEntity, la cual puede ser utilizada directamente para alimentar la lista que implementaremos en el siguiente capítulo con un RecyclerView y un Adaptador. Por una decisión de diseño, no queremos exponer un Entity a la lista, por lo que definiremos otra clase para transportar esos datos, más sobre esto en el próximo capítulo.

Además Room cuando no pueda mapear directamente la data a una clase despliega un warning CURSOR_MISMATCH, y hace un mapeo parcial de la data. Es decir, incluso cuando no funciona 100%, Room hace su mejor esfuerzo para cumplir con nuestro requerimiento. En el siguiente capítulo aprenderemos a lidiar con un problema común en Room relacionado a los tipos de datos.

Siguiendo en el código, la segunda función es para insertar datos a la base de datos, en este caso debemos utilizar la anotación @Insert. Esta anotación le dice a Room que este método inserta data en la base de datos. Los parámetros de este tipo de métodos pueden ser sólo clases anotadas con `@Entity` o colecciones de dichas clases, en este caso tenemos una List de entities. Cómo Room conoce la base de datos, generará todo el SQL necesario para realizar este Insert en la base de datos.Sólo nos falta nuestro tercer componente para completar el puzzle, veamos cómo definir un Database.

Database

Ya definimos dos de nuestros tres componentes, tenemos un DAO y un Entity, necesitamos algo que las conecte. El componente que une a los otros dos es un Database. En este caso no necesitamos definir mucho, pero como referencia es interesante saber que podemos tener más de un DAO asociado a un database, lo cual nos permite granularizar más nuestro código y ordenar de manera más consciente. Para definir un database en Room necesitamos usar una anotación que es clave, además de extender una Clase de Room, veamos el código:

```
@Database(entities = [(RecommendationEntity::class)], version = 1)
abstract class RecommendationsDatabase: RoomDatabase() {
  abstract fun getRecommendationsDAO(): RecommendationsDAO
}
```

La anotación `@Database`, nos permite decirle a Room que esta clase es una base de datos. Dentro de la anotación definimos un arreglo de entities asociadas a esta base de datos. En la anotación también definimos qué versión de la base de datos estamos ocupando en esta base de datos, este n{umero está asociado a las diferentes migraciones que debemos hacer cuando modificamos nuestra base de datos, agregando entities o modificando las mismas. Veremos un ejemplo de migración en el próximo capítulo.

La clase y los métodos que definimos deben ser tipo abstract. nuestra clase debe heredar de RoomDatabase, de esta forma Room sabe que este es la clase que nos permite acceder a los DAOs disponibles en el database.

Cada método que se define retorna un DAO, en este caso sólo uno, por lo que definimos sólo un método.

Ya tenemos todo lo necesario para utilizar Room en nuestra aplicación, Antes de concluir este capítulo, veremos otras anotaciones importantes en Room. Estas anotaciones son las más comunes, pero no son todas las que uno encuentra en Room. Además, veremos un ejemplo simple de como migrar una base de datos con Room.

Otras Anotaciones y Migraciones en Room

Lo que hemos explorado hasta ahora es un set acotado de funcionalidades de Room. En esta sección revisaremos otras anotaciones, además del proceso de migración, que se vuelve útil cuando comenzamos a agregar más entidades y filas a la base de datos.

Primero veremos las anotaciones más comunes, el resto de las anotaciones pueden ser revisadas en la guía oficial de google disponible en https://developer.android.com/reference/android/arch/persistence/room/package-summary. En esta guía no sólo se encuentran la documentación, también tenemos ejemplos, lamentablemente la mayoría de ellos están en Java, esperemos que en un futuro cercano Google actualice todo a Kotlin.

@Update

Muchas veces no necesitamos agregar nuevos elementos a nuestra tabla, queremos actualizar lo que ya está en ella, para eso existe la anotación update. El siguiente método actualiza una entidad Address que recibe como parámetro. Si la entidad existe en la base de datos, se realiza la actualización de todos sus campos, salvo el la clave primaria. Si la entidad no existe, no se hace nada, la función se puede definir de la siguiente forma:

```
@Update
fun updateAddress(Address address)
```

Un problema típico con esta anotación es cuando una entidad tiene muchos campos y necesitamos actualizar uno sólo. En ese caso es más conveniente usar `@Query` con un Update en SQL, que una función con la anotación Update.

@Delete

Esta anotación nos permite borrar de la base de datos lo que recibe como parámetro, los parámetros siempre son entidades o colecciones de entidades. Una implementación de delete puede ser la siguente:

```
@Dao
interface UserDao {
     @Delete
     fun deleteUsers(vararg users:User)
     @Delete
     fun deleteWithFriends(user:User, friends:List<User> )
 }
```

Un problema que podemos tener con esta anotación es que no podemos borrar toda la tabla a menos que tengamos todas las entidades para pasar como parámetro. En el caso que queramos hacer un borrado total, se recomienda hacer una query SQL con un método anotado con `@SQL`.

@ForeignKey

Esta anotación nos permite utilizar una clave foránea desde otra entidad, por ejemplo si tenemos una entidad Persona y una entidad Mascota, sabemos que la mascota tiene dueño, podemos definir la relación entre mascota y persona de la siguiente forma:

```
@Entity
data class Person(
        @PrimaryKey(autoGenerate = true)
        val id: Long,
        val name: String = "",
        val sureName: String = "",
        val phone: String = ""
)

@Entity(foreignKeys = arrayOf(ForeignKey(entity = Person::class,
        parentColumns = arrayOf("id"),
        childColumns = arrayOf("ownerId"),
        onDelete = ForeignKey.CASCADE)))
data class Pet(
        @PrimaryKey(autoGenerate = true)
        val id: Long,
        val ownerId: Long,
        val name: String
)
```

Un Pet pertenece a una persona, el Entity Pet contiene una propiedad ownerId, que será asignada de la siguiente manera´en la anotación `@Entity`:

- Utilizamos foreignKeys para definir un arreglo de claves foráneas, en este caso definimos sólo una clave foránea.

- En el arreglo definimos ForeignKey con los siguientes parámetros
    - **entity**: la entidad de la cual viene la clave foránea, en este caso Person.
    - **parentColumn**: la columna que contiene el valor de la clave foránea en la entidad de origen, en este caso es el id de Person.
    - **childColumn**: la columna que almacena el valor de la clave foránea en la entidad de destino, en este caso ownerId.
    - **onDelete**: aquí definimos el comportamiento de la base de datos con respecto a la entidad de destino, cuando eliminamos la entidad de origen. Al definir ForeignKey.CASCADE la base de datos borrará en cascada las entidades, es decir si borramos al dueño se borran las mascotas. También podemos definir el comportamiento para onUpdate, que define qué se hace en caso de una actualización.

      Hay que tener cuidado con esta anotación, al igual que cuando se ocupa SQL. El comportamiento al eliminar puede ser desastroso para nuestra base de datos.Esta anotación nos permite generar relaciones entre entidades, al igual que Embedded y Relation.

@Embedded

Muchas veces tenemos objetos que contienen otros objetos, esto es la norma en OOP, pero no necesitamos almacenar ese objeto en la base de datos como entidad. Y cuando recuperamos esta información también queremos tener nuevamente ese objeto disponible para ocuparlo, como objeto. Para estos casos es útil la anotación `@Embeded`.

También es importante hacer notar que esta es una de las anotaciones que nos permite hacer relaciones en la base de datos, en este caso puede ser 1-1 o 1 - N, dependiendo cuantas direcciones tenga una persona.

Recordando nuestro caso de una lista de contactos, donde teníamos una persona y una dirección, podemos embeber la dirección como objeto dentro de persona, la anotamos con `@Embedded` y dejamos que Room maneje la referencia y el almacenamiento, veamos como queda esto en código:

```
@Entity
data class Person(
        ...// other fields
        @Embedded
        val address: Address
)

data class Address(val postcode: String,
                   val streetName: String,
                   val streetNumber:Long,
                   val city: String)
)
```

En este caso el objeto dirección se almacenará como campos que son parte de Person, no se crea una Entity Address. Pero cuando se rescate desde la base de datos un Person, se mapeará el objeto Address a un Objeto independiente, manteniendo la intención original del diseño, muy conveniente y útil para manejar datos en nuestra aplicación.

@Relation

Esta anotación también nos permite generar relaciones entre entidades, 1-1, 1-N, etc. Al utilizar esta anotación en un entity, Room restaura automáticamente todas las entidades involucradas, sin la necesidad de utilizar consultas adicionales, ahorrandonos tiempo y código.

Cosas importantes que debemos saber de Relation:

- Esta anotación se puede utilizar sólo en campos o propiedades que sean: Set o List. Esta anotación no se puede utilizar para propiedades que nos sean una lista o un set de datos
- Cada entidad involucrada debe tener declarada una clave primaria.
- Las claves primarias definidas en objetos embebidos no son consideradas primarias en el objeto que lo contiene.

Para entender mejor cómo funciona esta anotación veamos el siguiente ejemplo:

- Queremos definir la relación entre una Persona y todas sus mascotas.
- Podemos definir esta relación en una nueva Entity, llamada PersonAndAllHisPets.
- Esta entity mantiene la relación de un Person con la lista de todos sus Pets. Al hacer una consulta sobre esta nueva entidad en un DAO, podemos obtener una lista de personas, con todas sus mascotas en una lista.

Para hacer esto debemos utilizar la anotación @Relation, definir la nueva entidad y DAO. En el ejemplo la nueva entidad contiene un Person embebido, además tiene una lista de pets, veamos el código:

```
@Entity
class Person(
    @PrimaryKey
     val id:Int,
     // otros campos, esta entidad es la dueña de las mascotas)

@Entity
class Pet (
     @ PrimaryKey
     val id:Int,
     val userId:Int,
     val name:String,
     // otros campos, esta entidad pertenece a un dueño Person
)
```

```
@Entity //define la relación entre un Person y sus Pets
class PersonAndAllHisPets(
     @Embedded
     val person:Person,
     @Relation(parentColumn = “id”, entityColumn = “userId”)
     val pets:List<Pet>
 )

@Dao //nos permite consultar por todas las personas y sus mascotas
 interface PersonPetDao {
     @Query("SELECT * FROM person_table WHERE id=:personId")
     fun loadPersoAndPets(personId:Int):List<PersonAndAllHisPets>
 }
```

Como podemos ver en el ejemplo, hemos creado un nuevo Entity, PersonAndAllHisPets. Lo interesante es que este nuevo entity nos permite rescatar directamente una lista mascotas asociadas a una persona en particular. En la anotación @Relation definimos la columna del dueño o padre que se relaciona con la columna de la entidad de destino, en este caso Pet. Esta relación nos permite generar el DAO con la función `loadPersonsAndPets()`, gracias a este dao podemos obtener todos las mascotas de la persona con el id = personId. Esto es realmente poderoso si pensamos lo simple que es generar esta nueva entidad y nuevo dao, para generar una consulta que normalmente se hace con un join en SQL. Podemos modelar todo tipo de relaciones con esta entidad, siempre y cuando tengamos una lista o set asociado, recordado la restricción para su uso.

@Ignore

Esta anotación nos permite definir campos en un objeto que no necesitamos guardar en la base de datos, por ejemplo un bitmap que representa un avatar o foto de identificación. Lo ocupamos de la siguiente manera

```
@Ignore
val avatar:Bitmap
```

Hay muchas más anotaciones disponibles para diferentes operaciones, pero estas son las más habituales y comunes al utilizar Room. Revisar la documentación oficial para otra anotaciones y propiedades https://developer.android.com/reference/android/arch/persistence/room/package-summary.

Migración de una base de datos con Room

A medida que agregamos nuevas funciones y componentes a una aplicación se nos presenta el siguiente problema:

Agregué nuevas entidades y campos a mi base de datos por lo que necesito migrar el esquema a una versión más nueva. Para realizar esto debemos escribir clases de migración, para decirle explícitamente a Room que hacer con cada versión de nuestro esquema.

La migración permite que los usuario actuales de la aplicación no pierdan sus datos cuando actualicen a la nueva versión. Antes de que migremos una base de datos necesitamos una versión del esquema de la base de datos exportada en nuestra aplicación. Para hacer esto necesitamos agregar el siguiente código a nuestro gradle file del módulo, en la configuración por defecto.

```
defaultConfig {
    applicationId "cl.desafiolatam.roomexample"
    minSdkVersion 23
    targetSdkVersion 29
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    javaCompileOptions { //esto exporta el esquema
        annotationProcessorOptions {
            arguments = ["room.schemaLocation":
                                "$projectDir/schemas".toString()]
        }
    }
}
```

Ya tenemos nuestro esquema, para migrar una base de datos Room necesita clases que extienden la clase migración. Estas clases debe tener una versión de inicio y una versión final. Para poder ejecutar las migraciones necesarias, en el orden correspondiente. Las clases deben ejecutarse en orden consecutivo, de esta forma se lleva la base de datos a su versión más nueva sin perder los datos almacenados.

Migration es el nombre de la clase que Room ofrece para generar una migración, normalmente estas clases migración se implementan como clases estáticas. Para generar una migración en Kotlin debemos hacer lo siguiente:

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, "
                        + "PRIMARY KEY(`id`))")
    }
}
val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE Book ADD COLUMN pub_year INTEGER")
    }
}
Room.databaseBuilder(applicationContext, MyDb::class.java, "database-name")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build()
```

En este ejemplo se definen dos migraciones, la primera migración lleva la base de datos desde la versión 1 a la versión 2, inicializando un objeto de la clase Migration con los parámetros 1 y 2.

```
 val MIGRATION_1_2 = object : Migration(1, 2)
```

dentro del cuerpo de esta clase se debe sobreescribir la función migrate, que recibe como parámetro un database

```
override fun migrate(database: SupportSQLiteDatabase)
```

Este database, que recibe la función migrate, debe ejecutar el SQL necesario para modificar el esquema actual y llevarlo al nuevo esquema, en este caso se ejecuta un Create table:

```
database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, " +
"PRIMARY KEY(`id`))")
```

En este método podemos ejecutar todas las queries y sentencias SQL que sean necesarias para actualizar la base de datos a la versión de destino.

Finamente todas las migraciones se ejecutan al construir la base de datos con el databaseBuilder de Room. La operación en este caso es la siguiente:

```
Room.databaseBuilder(applicationContext,
MyDb::class.java,
"database-name")
.addMigrations(MIGRATION_1_2, MIGRATION_2_3).build()
```

Así aseguramos la completa compatibilidad de nuestra nueva base de datos con las versiones más antiguas, y el usuario no pierde datos al actualizar la aplicación.

Creando una aplicación con Android Room

Competencias

- Desarrollar una aplicación completa en Android con Room, creando todo lo necesario para que funcione.
- Entender cómo se convierten los datos cuando debemos lidiar con campos que no son compatibles con SQL, por ejemplo un Date.
- Entender por qué es necesario ejecutar las operaciones con Room en un Thread distinto al principal en Android.

Motivación

Ya conocemos la teoría de que necesitamos y cómo funciona Room. En este capítulo completamos una aplicación simple, que nos permite ver en acción la librería. Además, veremos como solucionar algunos problemas que se pueden presentar al utilizar Room, pondremos especial énfasis en TypeConverter y cómo convertir de un tipo a otro, y devuelta.

Continuando con la aplicación del capítulo anterior

En este capítulo completamos la aplicación que empezamos a realizar en el capítulo anterior. En este capítulo crearemos rápidamente le proyecto con lo necesario para desplegar la lista, agregar tareas, y completar las mismas.

Creación del Proyecto

Seguiremos los mismos pasos que hicimos en el capítulo 2 para crear el proyecto de SharedPreferences, pero esta vez seleccionamos Basic Activity como proyecto inicial. Luego de que se ejecuten todos las sincronizaciones, configuramos todo lo descrito en el capítulo anterior. Incluimos la librería al proyecto, los plugins de kotlin, y el código para exportar el esquema. Sincronizamos gradle y estamos listos para continuar.

Creamos un paquete en el código con el nombre room, y creamos las clases Entity, DAO y Database que describimos en el capítulo anterior. Se puede copiar y pegar directamente el código. El resultado final de estos pasos se debe ver así en el proyecto:

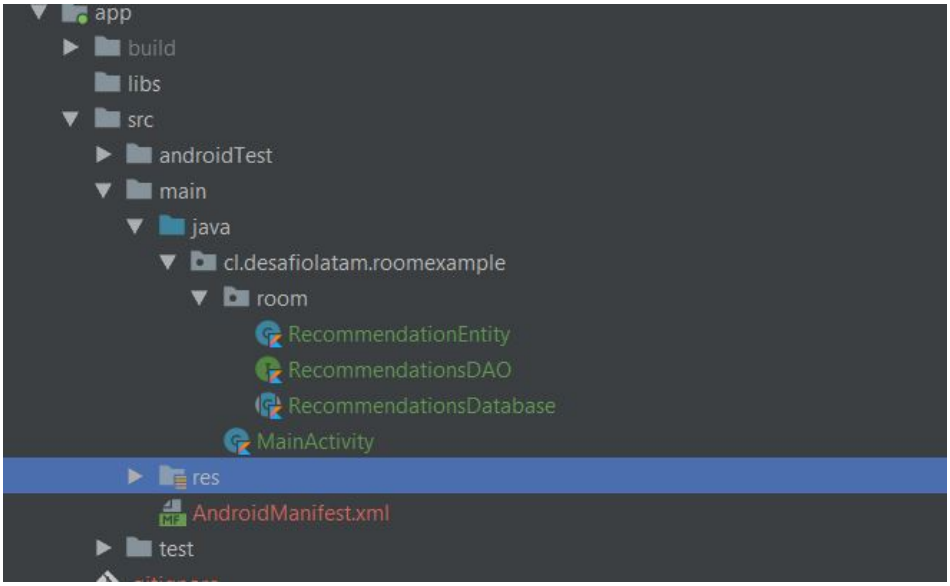

Imagen 4. Creamos los paquetes para Entity, DAO y Database.

Necesitamos configurar algunas otras cosas para poder utilizar completamente Room en el proyecto. Debemos crear una clase Application.

Clase Application y como crear una instancia de nuestra Base de Datos

Para utilizar la base de datos debemos crear una instancia de la misma, en la aplicación. Para crear la instancia debemos considerar que este proceso consume muchos recursos, por lo cual se recomienda utilizar un Singleton. Un Singleton nos permite tener una instancia global que se comparte a través de la aplicación, de esa forma no tenemos que crear una nueva instancia cada vez que necesitamos usarla. Esta instancia Singleton debe ser creada, o inicializada, en una clase que extienda Application, o alguna de sus variantes en Android. En este caso utilizaremos Application para crear lo que necesitamos para inicializar nuestra base de datos. La clase la creamos en el paquete principal del proyecto, y agregamos el siguiente código:

```
class RoomApplication: Application() {
    companion object {
        var recommendationsDatabase:RecommendationsDatabase? = null
    }

    override fun onCreate() {
        super.onCreate()
        recommendationsDatabase = Room
            .databaseBuilder(this,
                RecommendationsDatabase::class.java,
                "recommendations-master-db").build()
    }
}
```

Esta clase extiende o hereda de Application, que representa la clase aplicación en Android. Si nos falta alguna referencia podemos presionar alt + Enter en windows o option + Enter en Mac, para que Android Studio agregue la dependencia faltante.Debemos definir una instancia única, un Singleton de la base de datos. Para hacer esto en Kotlin debemos utilizar un companion object. Este companion object es un objeto que será compartido por todas las posibles instancias de esta clase, en este caso hay sólo una. Todo lo que se encuentra dentro de un companion object es compartido entre las diferentes instancias y puede ser accedido como los métodos o propiedades estáticas en Java. En este caso definimos nuestra variable database, y la inicializamos en el método onCreate de la clase. La variable database es de tipo RecommendationsDatabase?, el modificador ? significa que puede ser null, por eso la inicializamos null, para luego inicializarla una sóla vez. El método onCreate es llamado sólo cuando la clase Application es creada, esto ocurre una sóla vez al correr nuestra Aplicación. Tener una variable dentro de un companion object es el equivalente a declarar una variable static en Java.

Al crear la base de datos utilizamos el databaseBuilder de Room con tres parámetros:

- un Context, en este caso usamos this para pasar la instancia de Application.
- la clase que representa y está anotada con @Database, en este caso RecommendationsDatabase::class.java.
- un nombre en formato String, este es el nombre del archivo que será almacenado en la aplicación con nuestra base de datos y los datos almacenados en ella.

  Nos falta sólo un paso para terminar con la configuración, debemos agregar la clase aplicación al manifest.

Agregando Application al Manifest

Para configurar la aplicación que hemos creado en el manifest, lo hacemos con la siguiente modificación:

- agregamos la línea android:name, dentro del tag application, como se ve a continuación.

```
<application
        android:name=".RoomApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
```

Luego de esto estamos listos para construir esta aplicación. Veremos una pequeña explicación para entender que necesitamos a nivel de Interfaz de Usuario.

Aplicación Libreta de Recomendaciones

Para utilizar la base de datos que definimos anteriormente, construimos una sencilla aplicación que permite almacenar recomendaciones, que son desplegadas en una lista y almacenadas en una base de datos. En este diagrama podemos observar como está diseñada la interacción de los distintos componentes de la aplicación.

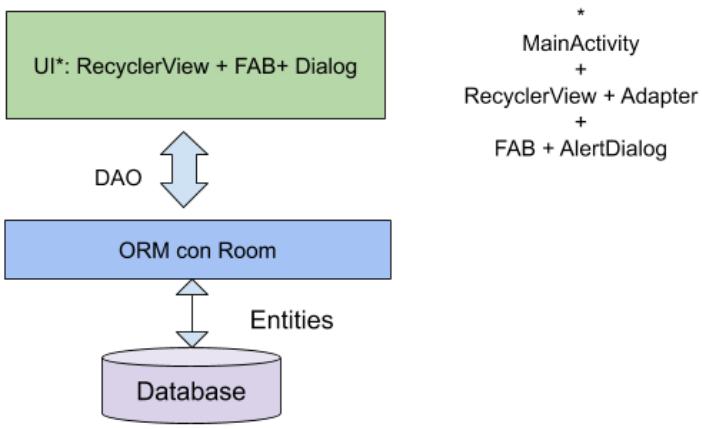

Imagen 5. Integración de los distintos componenentes en la aplicación.

Los componentes principales de la aplicación son:

- **ORM**: definido en los componentes de Room, y que ya hemos construido
- **MainActivity**: la pantalla principal que tiene una lista implementada con un RecyclerView más un Adaptador. Se crea también una dataclass que contiene los datos obtenidos de la base de dato. Este contenedor específico es utilizado con el adaptador del recyclerview. La idea es no exponer a capas superiores las entities, esto es una buena práctica. Por ejemplo si decidimos cambiar las entities, pero no cambiamos la información que se muestra, sólo debemos cambiar la función o método que mapea la información de las entities a los dataclass de pantalla, evitando tener que cambiar mucho código, facilitando la extensión y mantención de la aplicación.
- Cuatro **layouts**, en xml, que representan:
    - el AlertDialog que se utiliza para ingresar recomendaciones de a una.
    - Un layout que contiene la pantalla principal, que a su vez contiene otro layout.
    - Un layout con el RecyclerView que representa la lista que desplegamos, además de contener el FAB
    - Un layout que representa una fila de la lista

Comenzaremos construyendo los xml que necesitamos.

Creación de XML de la aplicación

El XML que representa la pantalla principal es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">
    <include layout="@layout/content_main"/>
    <com.google.android.material.floatingactionbutton.FloatingActionButton
            android:id="@+id/fab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="bottom|end"
            android:layout_margin="@dimen/fab_margin"
            app:srcCompat="@drawable/add_plus_white"/>
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Podemos ver en este xml:

- Un CoordinatorLayout, que coordina las interacciones en la pantalla, este fue seleccionado por defecto, y se utiliza para generar animaciones en la interfaz. Además de coordinar otras interacciones, no necesitamos saber mucho más sobre este componente.
- Tenemos un include, que nos permite reutilizar otros xml, al incluirlos como parte de este. El include en este caso agrega content_main, que es la pieza que contiene el recyclerview que mostrará la lista.
- Finalmente tenemos un FloatingActionButton que será el nos permite agregar recomendaciones a nuestra lista.
- Además, definimos un drawable que representa el símbolo + en el botón fab.

El drawable se define así

```
<?xml version="1.0" encoding="utf-8"?>
<vector xmlns:android="http://schemas.android.com/apk/res/android"
                android:width="24dp"
                android:height="24dp"
                android:viewportWidth="24.0"
                android:viewportHeight="24.0">
<path android:fillColor="#FFFFFFFF"
      android:pathData="M19,13h-6v6h-2v-6H5v-2h6V5h2v6h6v2z"/>
</vector>
```

El segundo Xml que debemos definir es content_main, lo definimos de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:tools="http://schemas.android.com/tools"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       app:layout_behavior="@string/appbar_scrolling_view_behavior"
       tools:showIn="@layout/activity_main"
       tools:context=".MainActivity">
   <androidx.recyclerview.widget.RecyclerView
           android:layout_width="match_parent"
           android:layout_height="match_parent"
           app:layout_constraintTop_toTopOf="parent"
           app:layout_constraintEnd_toEndOf="parent"
           app:layout_constraintBottom_toBottomOf="parent"
           app:layout_constraintStart_toStartOf="parent"
           android:id="@+id/recommendations_list"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Lo más importante de este xml es el RecyclerView que nos permitirá mostrar la lista de recomendaciones.

Además debemos crear un xml que represente en pantalla el texto, para esto creamos un xml llamado recommendation_item, este xml queda de la siguiente manera

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:orientation="vertical"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">
   <TextView
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:id="@+id/recommendationBox"
           app:layout_constraintEnd_toEndOf="parent"
           android:layout_marginEnd="8dp"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           android:layout_marginTop="8dp"
           app:layout_constraintTop_toTopOf="parent"
           android:textSize="18sp"
           app:layout_constraintBottom_toBottomOf="parent"
           android:layout_marginBottom="8dp"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Para agregar las diferentes recomendaciones utilizaremos un AlertDialog. Debemos crear un xml custom para este caso. El xml que utilizaremos es el siguiente, add_recommendation_layout.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       android:orientation="vertical"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">

   <com.google.android.material.textfield.TextInputLayout
           android:layout_width="0dp"
           android:layout_height="wrap_content"
           android:layout_marginTop="8dp"
           app:layout_constraintTop_toTopOf="parent"
           android:layout_marginBottom="8dp"
           app:layout_constraintBottom_toBottomOf="parent"
           app:layout_constraintStart_toStartOf="parent"
           android:layout_marginStart="8dp"
           app:layout_constraintEnd_toEndOf="parent"
           android:layout_marginEnd="8dp"
           android:id="@+id/recommendation_input_layout">

       <com.google.android.material.textfield.TextInputEditText
               android:layout_width="match_parent"
               android:layout_height="wrap_content"
               android:hint="@string/agrega_hint"
               android:id="@+id/recommendation_input"/>
   </com.google.android.material.textfield.TextInputLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

con este xml completamos todo lo necesario para mostrar y agregar recomendaciones a nivel visual, veamos ahora cómo pegamos la base de datos y la interfaz.

Creando el código para unir la interfaz con los datos

Como dijimos anteriormente, ya tenemos toda la base de nuestra base de datos y UI, ahora debemos construir el código para unir las dos capas, creamos un nuevo paquete llamado list, para mantener el orden. Lo primero que construiremos en este nuevo paquete es un contenedor de los datos que rescatamos de la base de datos. Esto es opcional, pero como no queremos exponer los Entities, creamos este pequeño data class llamado RecommendationDataView, el código a continuación:

```
data class RecommendationDataView(
var id:Long,
var text:String)
```

Para usar un recyclerview debemos construir dos componentes claves, su descripción a continuación:

- Un Adapter: que nos permite crear las vistas del RecyclerView, mantiene una lista de los datos y genera las vistas de cada fila.
- Un ViewHolder, que es el contenedor visual de los datos. Este viewholder representa visualmente una fila de la lista y está asociado directamente al adaptador.

La forma en que RecyclerView implementa las listas es eficiente y práctica, reduciendo el uso de memoria y facilitando nuestro trabajo. Lo primero que definiremos es el ViewHolder. El código a continuación:

```
class RecommendationViewHolder(view: View): RecyclerView.ViewHolder(view) {
val recommendationCheck = view.findViewById<TextView>
(R.id.recommendation_text)
}
```

El adaptador se define los métodos por defecto que debe definir cada clase que extiende RecyclerView.Adapter. Además, se agrega un método para actualizar la data de manera general, reemplazando la data presente en el adaptador y notificar que el set de datos ha cambiado, para refrescar la lista en pantalla, el código que define este adaptador es el siguiente:

```
class RecommendationsAdapter(
   private var recommendationsList: MutableList<RecommendationDataView>,
   var context: Context
): RecyclerView.Adapter<RecommendationViewHolder>() {

   override fun onCreateViewHolder(parent: ViewGroup, viewType:Int)
           :RecommendationViewHolder {
       return RecommendationViewHolder(
           LayoutInflater.from(context)
               .inflate(R.layout.recommendation_item, parent, false))
   }
   override fun getItemCount(): Int = recommendationsList.size

   override fun onBindViewHolder(holder: RecommendationViewHolder, position:
Int) {
       val data = recommendationsList[position]
       holder.recommendationText.text = data.text
   }

   fun updateData(items:List<RecommendationDataView>) {
//hace update de la data cuando el usuario agrega nuevas recomendaciones
       recommendationsList.clear()
       recommendationsList.addAll(items)
       notifyDataSetChanged()
   }
}
```

Para finalizar, la aplicación despliega su MainActivity, en la cual se definen las interacciones y las diferentes variables. La Actividad es muy grande para incluirla completamente en este documento, pero se incluyen los métodos principalesLo primero es definir MainActivity como una clase que extiende AppCompatActivity, y sus propiedades:

```
class MainActivity : AppCompatActivity() {
    private lateinit var list: RecyclerView
    private lateinit var adapter:RecommendationsAdapter
    private lateinit var dataBase:RecommendationsDatabase
    private lateinit var dao: RecommendationsDAO
    private lateinit var addButton:FloatingActionButton
```

En el método onCreate de la actividad inicializamos todas las propiedades de esta Actividad, el método onCreate queda definido de la siguiente manera, todo lo creado antes de setUpViews se crea automáticamente al crear el main activity. El modificador !! es para asegurar que la variable no es nunca null en este momento de la ejecución de la aplicación:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    setSupportActionBar(toolbar)
    setUpViews()
    dataBase = RoomApplication.recommendationsDatabase!!
    dao = dataBase.getRecommendationsDAO()
    setUpAddButton()
}
```

El método setUpViews está definido de la siguiente forma, tener claro que mutableListOf() crea una lista modificable vacía, es como hacer new ArrayList() en Java:

```
private fun setUpViews() {
    list = recommendations_list
    list.layoutManager = LinearLayoutManager(this)
    addButton = fab
    adapter = RecommendationsAdapter( mutableListOf(), this)
    list.adapter = adapter //el adapter de la derecha es la propiedad
//de la actividad, el de la izquierda es la propiedad del adapter
}
```

El método setUpAddButton está definido de la siguiente manera:

```
private fun setUpAddButton() {
   addButton.setOnClickListener {
   val dialogView = layoutInflater
                    .inflate(R.layout.add_recommendation_layout, null)
   val recommendationText = dialogView.recommendation_input
   val dialogBuilder = AlertDialog
           .Builder(this)
           .setTitle("Agrega una recomendación")
           .setView(dialogView)
           .setNegativeButton("Cerrar") {
                   dialog:DialogInterface, _: Int -> dialog.dismiss()}
      .setPositiveButton("Agregar") {dialog:DialogInterface, _: Int ->
      if (recommendationText.text?.isNotEmpty()!!) {
        AsyncTask.execute {
    dao.insertRecommendations(createEntity(recommendationText.text.toString()))
    val newItems = createEntityListFromDatabase(dao.getAllRecommendations())
    runOnUiThread {
                    adapter.updateData(newItems)
                    dialog.dismiss()
                    }
                }
            }
        }
        dialogBuilder.create().show()
   }
}
```

Acá se define la interacción que nos permite agregar recomendaciones a la base de datos. El addButton despliega un AlertDialog cuando es presionado.

Dentro del lambda que define el click listener del botón pasan varias cosas:

- Creamos la vista con el xml definido, add_recommendation_layout
- Creamos un TextInput para obtener el texto ingresado por el usuario
- Creamos el AlertDialog con un builder
    - Agregamos el título
    - Agregamos la vista que creamos
    - Agregamos la acción del botón negativo
    - Agregamos la acción del botón positivo, este botón ejecuta un AsycnTask en background. Los AsyncTask se deben ejecutar en un thread diferente al de la interfaz. En Android está prohibido ejecutar cálculos o tareas pesadas en el Thread principal, este thread maneja la interfaz. Si ejecutamos tareas en este thread, bloqueamos la interfaz de usuario y eso es un pecado mortal en Android. El AsyncTask ejecuta las siguientes operaciones
        - Inserta la nueva recomendación en la base de datos con el Dao
        - Rescata todos los elementos de la base de datos con el Dao
        - Finalmente vuelve al thread de UI para ejecutar un update de los datos en el adaptador y cerrar el dialog
- La última línea crea y muestra el AlertDialog cuando el addButton es presionado.

Tenemos dos pequeñas clases que nos ayudan a mapear los datos desde la entidad a el dataclass que se usa para desplegar información en la pantalla, y otra para crear la entidad a almacenar en la base de datos. Ambas funciones se ven a continuación:

```
private fun createEntity(text:String):RecommendationEntity {
   return RecommendationEntity(recommendation = text, isCheck = false)
}

private fun createEntityListFromDatabase(entities:List<RecommendationEntity>):
MutableList<RecommendationDataView> {
   val dataList = mutableListOf<RecommendationDataView>()
   if (entities.isNotEmpty()) {
       for (entity in entities) {
           val dataView = RecommendationDataView(
                               entity.id,
                               entity.recommendation,
                               entity.isCheck)
           dataList.add(dataView)
       }
   }
   return dataList
}
```

En la función createEntity creamos un Entity con parámetros nombrados, así le decimos explícitamente qué parámetro estamos ocupando. Uniendo todos estos componentes nuestra aplicación funciona correctamente, persistiendo los datos entre instancias y desplegando todo lo que se ha agregado en la lista principal. Algunas imágenes de la aplicación

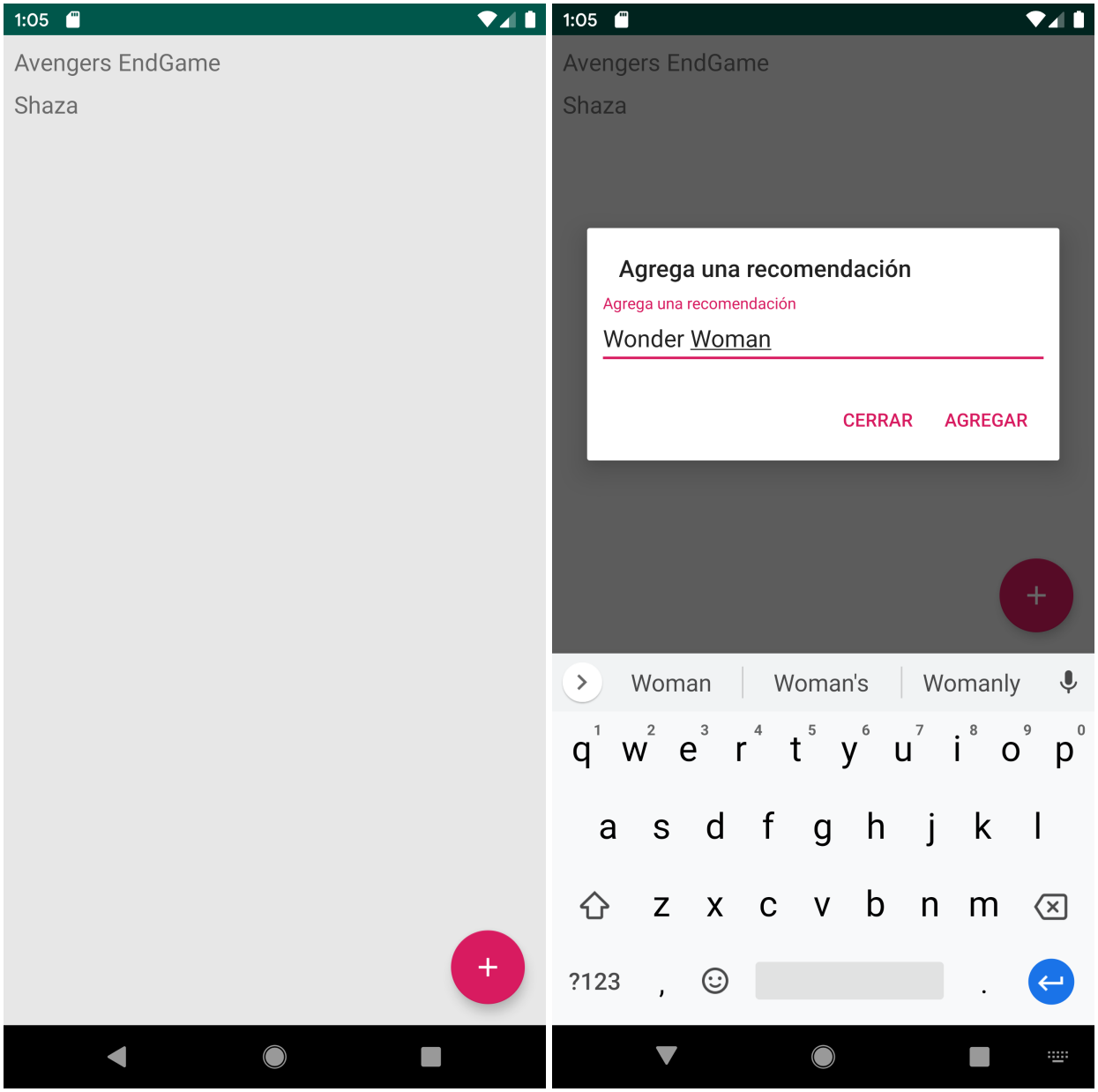

Imagen 6. Imágenes de la aplicación.

Imagen 7

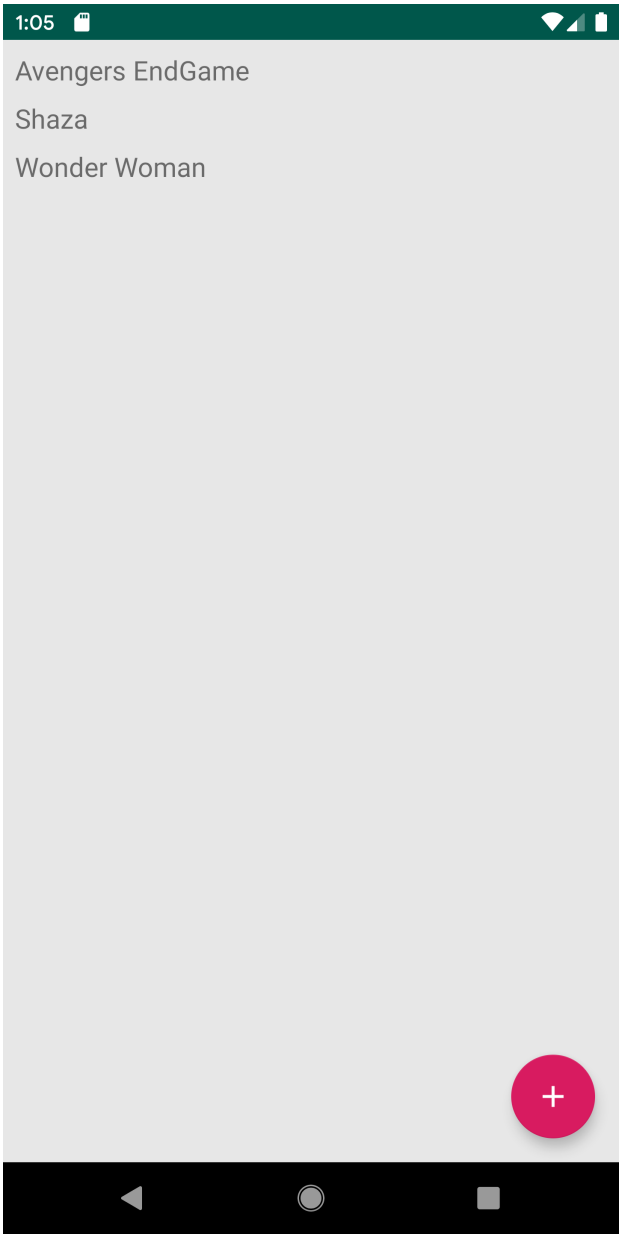

Imagen 8. Imágenes de la aplicación.

Nuestra aplicación funciona, pero qué ocurre si quiero agregar la fecha en que agregué las recomendaciones y mostrarlas. Esto nos crea dos problemas que resolveremos a continuación.

Manejo de datos no compatibles con SQL, TypeConverters

Queremos agregar la fecha en que se agrega cada recomendación y mostrarla en pantalla. Esto debería ser simple, ¿cierto?. Dos problemas ocurren cuando queremos hacer esto, y ya hemos corrido nuestra aplicación.

- El primer problema es que nuestro esquema ya existe, y si queremos agregar un campo a la tabla, necesitamos hacer una migración. Ya veremos como se soluciona ese problema.
- El segundo problema es que SQLite no puede almacenar Date, u otro objeto complejo, no es un tipo de dato compatible. Entonces, ¿cómo solucionamos este problema?, con un TypeConverter.

TypeConverter

Room nos permite almacenar tipos de datos primitivos como enteros, decimales, longs, texto, etc. Para poder guardar tipos de datos más complejos, debemos realizar una transformación de ese dato cuando lo almacenamos, y cuando lo rescatamos de la base de datos. Para este propósito tenemos disponible la anotación @TypeConverter. Esta anotación nos permite señalar que hemos definido una función que transformará de un tipo a otro, esto se hace ejecutando el código en la función.

En nuestro caso queremos agregar la fecha en que se agrega cada recomendación para lo cual haremos la siguiente modificación a nuestro Entity:

```
@Entity(tableName = "recommendations_list")
data class RecommendationEntity(@ColumnInfo(name = "id")
                                @PrimaryKey(autoGenerate = true)
                                val id:Long = 0,
                                @ColumnInfo(name = "recommendation_text")
                                val recommendation:String,
                                @ColumnInfo(name = "created_at")
                                val createdAt: Date?)
```

Agregamos la propiedad createAt, de tipo Date. Para que Room pueda persistir este nuevo campo debemos crear un TypeConverter. Para hacer esto debemos crear una clase que contenga las funciones para convertir los tipos de datos. Convertiremos el tipo Date en un UnitTimeStamp, esto se representa con un Long. Desde este Long crearemos un date cuando necesitemos hacer la conversión contraria. Definimos una clase llamada RecommendationsConverter, el código para esta clase es el siguiente:

```
class RecommendationsConverter {
    @TypeConverter
    fun fechaDesdeTimestampLong(value: Long?): Date? {
        return value?.let { Date(it) }
    }

    @TypeConverter
    fun longTimeStampDesdeDate(date: Date?): Long? {
        return date?.time
    }
}
```

Hemos creado dos funciones, estas funciones hacen lo siguiente:

- **fechaDesdeTimeStampLong**: crea un Date desde el Long que tenemos almacenado en la base de datos. Esta función ocupa let para crear el objeto Date, siempre y cuando exista el Long. De esta forma no debemos preocuparnos de posibles NullPointers u otra excepción.
- **longTimeStampDesdeDate**: crea un Long desde un Date, que se almacena en la base de datos. En este caso esto es simple debido a que Date de java tiene un método time que nos devuelve la representación en Long de la fecha en cuestión.Para que esta clase sea utilizada por nuestra base de datos debemos configurar la clase que convierte data en nuestra clase que contiene la base de datos. Para hacer esto, debemos agregar la anotación TypeConverte a la base de datos de la siguiente manera:

```
@Database(entities = [(RecommendationEntity::class)], version = 1)
@TypeConverters(RecommendationsConverter::class)
abstract class RecommendationsDatabase: RoomDatabase() {
    abstract fun getRecommendationsDAO(): RecommendationsDAO
}
```

Al agregar la anotación @TypeConverters, le decimos a este database que todos sus componentes pueden ocupar esta clase TypeConverter. Podemos definir un alcance menor a quién puede ocupar el TypeConverter de la siguiente forma:

- Si le agregamos la anotación a un Dao, todas las funciones de ese Dao pueden ocupar la clase.
- Si le agregamos la anotación a un método del Dao, sólo ese método puede ocupar la clase.
- Si le agregamos la anotación a un Entity, todos los campos de ese entity pueden ocupar la clase.
- Si le agregamos la anotación a un campo del Entity, sólo ese campo puede ocupar la clase.

Con esto ya hemos solucionado nuestro primer problema, podemos convertir nuestro nuevo campo para almacenarlo en la base de datos, y lo podemos convertir devuelta cuando lo necesitamos. Veamos cómo resolver nuestro problema de migración.

Migrando la base de datos

Necesitamos migrar nuestra base de datos para que acepte este nuevo campo. Debemos implementar la clase que conocimos en el capítulo anterior, Migration. Debemos agregar una nueva columna a nuestra tabla. En este caso utilizaremos el siguiente código, que se encuentra definido en un archivo llamado RecommendationsMigration.kt:

```
val MIGRATION_1_2:Migration = object : Migration(1,2) {
   override fun migrate(database: SupportSQLiteDatabase) {
       database.execSQL("ALTER TABLE recommendations_list " +
               "ADD COLUMN created_at INTEGER NOT NULL")
   }
}
```

Primero, notar que no es necesario definir una clase para definir nuestra clase Migration, lo podemos hacer en un Archivo .kt, esta es una de las ventajas de Kotlin.

Segundo, para definir la clase anónima utilizamos object:Migration(1,2). Al hacer esto le estamos diciendo a Room que esta clase Migration lleva la base de datos de la versión 1 a la 2.

Tercero, ejecutamos una sola query SQL, en este caso un Alter table, con el que agregamos a la tabla recommendations_list una nueva columna llamada created_at de tipo INTEGER, ya que SQLite no maneja LONGs. También le decimos a la base de datos que esta nueva columna es NOT NULL. Con esto logramos agregar lo que necesitamos para migrar la base de datos a la versión 2.

Debemos actualizar la versión en la base de datos por lo que modificamos el código de la siguiente manera:

```
@Database(entities = [(RecommendationEntity::class)], version = 2, exportSchema
= true)
@TypeConverters(RecommendationsConverter::class)
abstract class RecommendationsDatabase: RoomDatabase() {
   abstract fun getRecommendationsDAO(): RecommendationsDAO
}
```

Cambiamos la versión a 2, y agregamos el exportSchema a la anotación, de esta forma nuestro esquema aparecerá en la carpeta schemas de nuestro proyecto.

Para utilizar esta clase Migration la debemos agregar cuando inicializamos la base de datos. Recordemos que la inicialización de la base de datos se hace el método onCreate de la clase que extiende de Application. Debemos modificar el código de la siguiente manera:

```
override fun onCreate() {
   super.onCreate()
   recommendationsDatabase = Room
       .databaseBuilder(this,
           RecommendationsDatabase::class.java,
           "recommendations-master-db")
       .addMigrations(MIGRATION_1_2) // agregamos esta nueva operación para
migrar
       .build()
}
```

Al agregar la operación addMigrations le decimos a Room que actualice la base de datos al inicializarla. Con esto hemos migrado correctamente nuestra base de datos. Agreguemos lo que nos falta para poder mostrar la fecha de creación en Pantalla.

Modificando nuestro Adapter, ViewHolder y layout

Necesitamos hacer algunas modificaciones para mostrar la fecha en nuestra lista, lo primero que modificaremos es recommendation_item. Agregamos un nuevo textview con id creation_date. El layout, queda de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
    <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:id="@+id/recommendation_text"
            app:layout_constraintEnd_toEndOf="parent"
            android:layout_marginEnd="8dp"
            app:layout_constraintStart_toStartOf="parent"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            app:layout_constraintTop_toTopOf="parent"
            android:textSize="18sp"
    />
    <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:id="@+id/creation_date"
            android:layout_marginTop="10dp"
            app:layout_constraintTop_toBottomOf="@+id/recommendation_text"
            app:layout_constraintStart_toStartOf="parent"
            android:layout_marginStart="8dp"
            app:layout_constraintEnd_toEndOf="parent"
            android:layout_marginBottom="8dp"
            app:layout_constraintBottom_toBottomOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

La siguiente modificación que necesitamos hacer es en nuestro ViewHolder. Agregamos el textview nuevo que mostrará la fecha, para agregarlo modificamos nuestro ViewHolder de la siguiente manera:

```
class RecommendationViewHolder(view: View): RecyclerView.ViewHolder(view) {
   val recommendationText: TextView =
view.findViewById(R.id.recommendation_text)
   val dateCreation: TextView = view.findViewById(R.id.creation_date)
}
```

Debemos agregar la nueva propiedad a nuestro DataView, lo hacemos de la siguiente manera:

```
data class RecommendationDataView(
var id:Long,
var text:String,
var creationDate: Date
)
```

Modificamos nuestro adapter agregando lo necesario para mostrar la fecha en el método onBindViewHolder, el código queda de la siguiente manera:

```
override fun onBindViewHolder(holder: RecommendationViewHolder, position: Int)
{
   val data = recommendationsList[position]
   holder.recommendationText.text = data.text
   //Formato para parsear date de creación
 val dateFormater = SimpleDateFormat("EEE, d MMM yyyy HH:mm",
Locale.getDefault())
   val dateString = dateFormater.format(data.creationDate)
   holder.dateCreation.text = dateString
}
```

Sólo nos queda modificar dos métodos de MainActivity, los métodos que debemos modificar son createEntity, agregando createdAt con un Date que será la fecha y hora actual. El código queda de la siguiente forma:

```
private fun createEntity(text:String):List<RecommendationEntity> {
   val listEntities = mutableListOf<RecommendationEntity>()
   listEntities.add(RecommendationEntity(recommendation = text, createdAt =
Date()))
   return listEntities
}
```

También modificamos createEntityListFromDatabase, debemos agregar un método que maneje la fecha de creación, sobretodo en las primeras recomendaciones, ellas tienen created_at = null, ya que la columna no existía cuando fueron creadas. Las nuevas entradas tendrán la fecha de creación gracias a la modificación que hicimos en createEntity. El método createEntityListFromDatabase queda de la siguiente forma:

```
private fun createEntityListFromDatabase(entities:List<RecommendationEntity>):
MutableList<RecommendationDataView> {
    val dataList = mutableListOf<RecommendationDataView>()
    if (entities.isNotEmpty()) {
        for (entity in entities) {
            val date = entity.createdAt ?: Date()
            val dataView = RecommendationDataView(
                entity.id,
                entity.recommendation,
                date)
            dataList.add(dataView)
        }
    }
    return dataList
}
```

Después de estas modificaciones estamos listos para correr la aplicación. Si implementamos todo correctamente, la migración será sin problemas y veremos las siguientes pantallas:

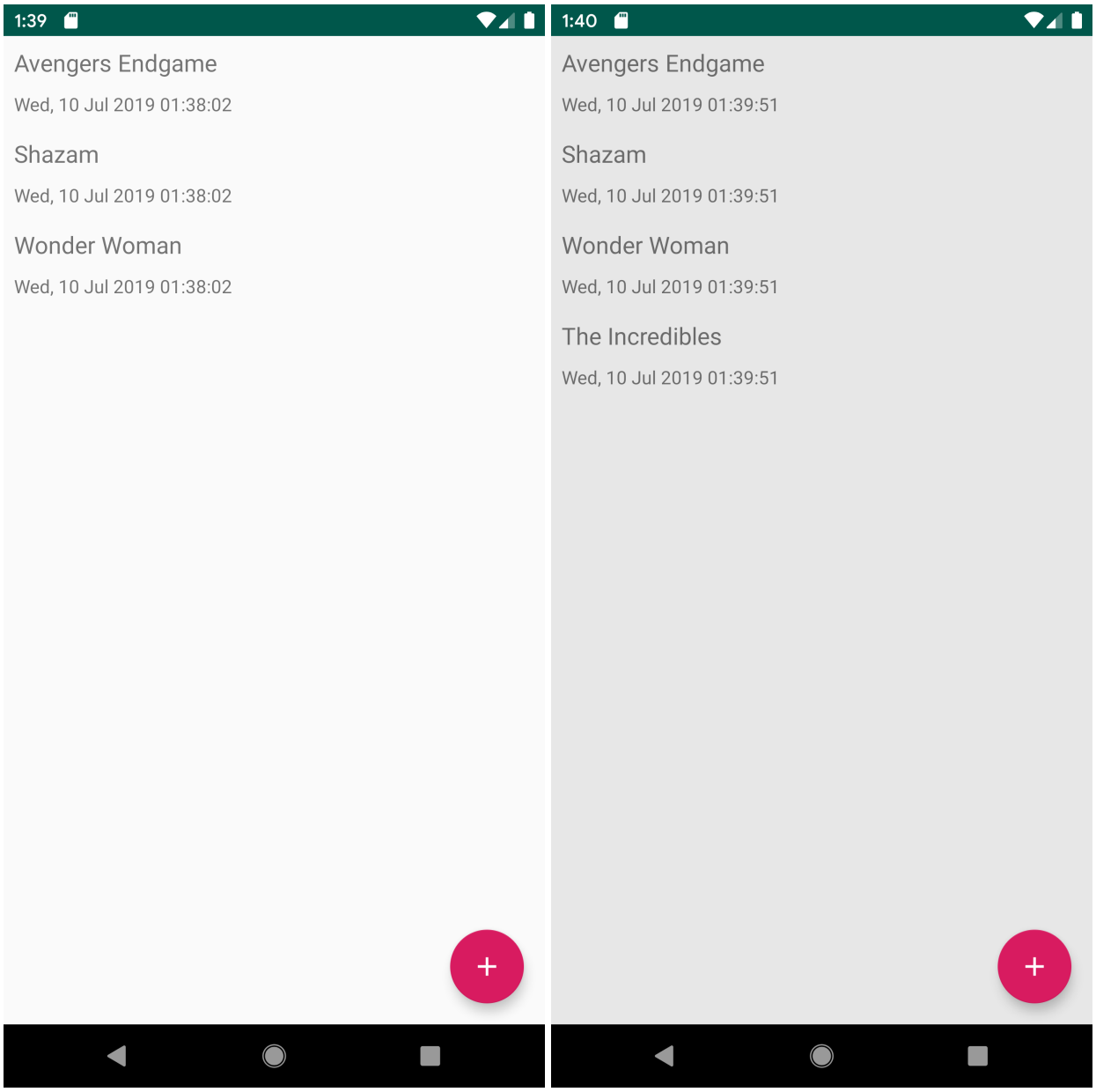

Imagen 9 y 10. Pantallas de la aplicación.

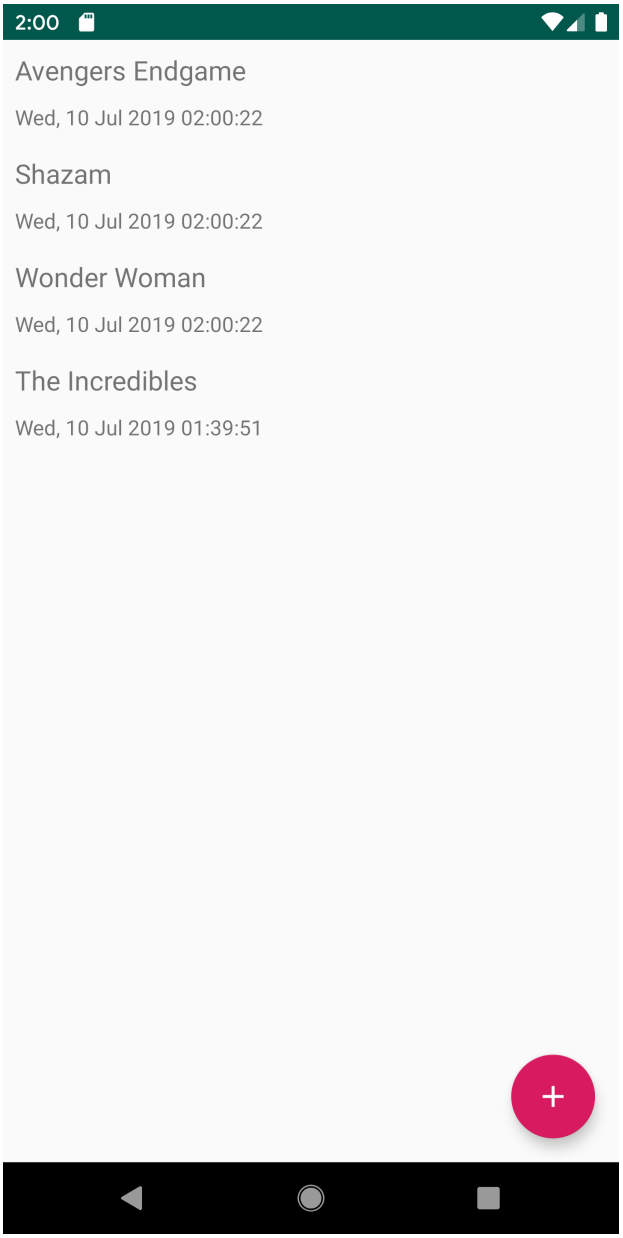

Imagen 11. Pantallas de la aplicación.

Cómo vemos las primeras recomendaciones van a mostrar siempre la última fecha, ya que su fecha es null. Pero las recomendaciones que ingresamos después de crear la nueva columna, veremos la fecha real de cuando se guardaron.Con esto concluimos el capítulo de Room. Ahora tenemos un capítulo donde veremos algunas de las opciones que existen aparte de Room en Android.

Realm y ObjectBox

Competencias:

- Conocer opciones a SQLite y Room para almacenar datos en Android
- Entender las principales ventajas y desventajas de Realm y ObjectBox
- Entender qué opción es la mejor para almacenar datos en una aplicación, dependiendo las necesidades de dicha aplicación.

Motivación

Conocer dos nuevas opciones para tener una capa de persistencia en nuestra aplicación. Si bien Android nos ofrece una librería oficial, muchas veces hay soluciones que son más fáciles de implementar, ocupan menos espacio o tienen mejor desempeño. Veremos dos de esas opciones, la madura Realm y la novata ObjectBox. Este capítulo no es una guía completa de ambas opciones, está más bien orientado a presentar y describir las ventajas, mostrar algunos ejemplos e incentivar la curiosidad para su uso.

Realm

Comenzaremos hablando de Realm. Se define como una base de datos no relacional o NoSQL. Este tipo de base de datos están diseñadas para tener un gran desempeño al leer, las consultas son casi inmediatas. Realm está específicamente diseñada para las plataformas móviles, originalmente estaba diseñada sólo para iOS, pero dado su éxito la versión de Android se puso a disposición de la comunidad en poco tiempo.

¿Cuáles son las ventajas y desventajas de Realm?. Veamos la respuesta:

Ventajas y Desventajas Realm

Las **ventajas** son las siguientes:

- Simplicidad: A diferencia de una base de datos SQL, SQLite para ser exactos, el código para realizar la mayoría de las operaciones en Realm es corto y conciso. Debemos lidiar con objetos y árboles de objetos, no tablas ni filas.
- Velocidad: como todo el sistema trabaja en base a objetos, de manera no relacional, las operaciones CRUD son muy rápidas. Además, la librería completa está escrita en c++, asegurando un desempeño a toda prueba.
- Live Objects: Realm no copia datos, todos los accesos a los datos son cargados de forma lazy, generando referencias, sin copiar los datos. Esto, en la práctica, significa que cada vez que accedemos a un objeto obtenemos su última versión.
- Documentación de primer nivel, una gran comunidad: todo lo que rodea a Realm es muy completo, una gran comunidad, buena tecnología y buenísima documentación.
- Soporte para Json: Realm incluye soporte para Json, esto es muy útil cuando estamos utilizando servicios REST o utilizando archivos Json para configuraciones u otros casos en la aplicación. Normalmente las base de dato NoSQL construyen documentos para almacenar la data, algunas ocupan json para lograr eso. Nos ahorra la necesidad de crear data classes para el parseo de datos de servicios web o archivos.
- Seguridad: Se puede utilizar encriptación para la base de datos y los datos.
- Programación Reactiva: Los cambios en los objetos de datos pueden ser “observados” y de esa forma mantener la UI actualizada, con cada cambio. Compatible con RxJava, la librería de programación reactiva más popular en Android.
- Adapters para la UI de Android: Realm trae adaptadores listos para ser utilizados con la UI de Android, por ejemplo en una lista de sugerencias.

Las principales **desventajas** de Realm, que si bien no son muchas pueden resultar relevantes dependiendo el caso, son:

- No autoincrementa valores: por lo que debemos definir funciones para hacer el trabajo de incrementar por ejemplo ids de los objetos.
- Model classes limitados: si bien podemos definir clases que modelan los datos, estos no se pueden personalizar mucho, normalmente se puede modificar los getters y setters, pero nada más.
- No es multi-hilo: Android es un sistema multi-hilo, la interfaz corre en un hilo principal, y no se debe bloquear nunca. Las entidades, model classes, no pueden pasar de un thread a otro en Realm. Por ejemplo si estamos modificando un dato fuera del hilo que corre la interfaz esa modificación no se puede pasar al hilo principal. Para poder leer datos modificados fuera del hilo principal, debemos hacer una consulta en el hilo principal, después de terminar de modificar los datos.
- Problemas con el tamaño: Las base de datos pueden ser de gran tamaño, debido a cómo está construída en Realm.
- Se necesita un binario para cada arquitectura soportada, recordemos que el código está en c++. Esto implica un APK universal más grande con cada arquitectura que agreguemos. Normalmente se soportan 4: armeabi-v7a, arm64-v8a, x86 y x86_64.
- Puede tener problemas de performance, dependiendo que tan compleja y grande sea la base de datos.

Después de ver las ventajas y desventajas de Realm, veremos algunos ejemplos. Para más información revisar https://realm.io/docs/java/latest, no hay documentación en Kotlin por ahora en la documentación oficial, pero está 100% soportado.

Ejemplos de Realm

Así podemos definir un modelo:

```
open class Person(
@PrimaryKey var id: Long = 0,
var firstName: String = "",
var lastName: String = "",
var age: Int= 0,
@Ignore var campoCalculado: Long= 0L) : RealmObject() {}
```

Definimos un objeto, como una clase, anotamos la clave primaria, y extendemos de RealmObject. Con esto tenemos todo lo necesario para operar. Notar que también podemos anotar que campos se ignoran. En Kotlin debemos poner valores por defecto obligatoriamente, Realm no tiene valores por defecto para los tipos.

Para inicializar la base de datos simplemente hacemos lo siguiente:

```
class ExampleApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        //al igual que Room sólo inicializamos una vez cuando
        //parte la aplicación
        Realm.init(this)
    }
}
```

La inicialización se debe hacer una sola vez, después de esto podemos utilizar la base de datos llamando la misma de la siguiente manera:

```
realm = Realm.getDefaultInstance()
```

Teniendo la referencia a la base de datos podemos ejecutar transacciones y queries, por ejemplo si quiero buscar a una persona con cierta edad en mi base de datos de personas podemos hacer lo siguiente:

```
val results = realm.where<Person>().equalTo("age", 55).findAll()
```

En este caso results es una lista de todas las coincidencias de esa búsqueda, ya que usamos findAll(). muchos más métodos están disponibles, todos bien explicados y con ejemplos en la documentación oficial. Para buscar por un campo ocupamos el método where, al igual que SQL en un select where. Le decimos que el tipo de objeto que vamos a buscar es de tipo Person y que necesitamos que nos encuentre a todos los que tienen el campo “age” igual a 55. Podemos ocupar otros comparadores para números, como por ejemplo:

- between, busca en un rango, incluidos los extremos
- lessThan: buscando el menor que el número definido

Hay más métodos disponibles en la documentación.

Cuando queremos escribir a la base de datos debemos ejecutar una transacción, para ejecutar transacciones lo debemos hacer en un bloque de código dentro de la función executeTransaction de la siguiente manera:

```
// buscamos a la primera persona de apellido bond, le
  //cambiaremos el apellido y la edad
  val person = realm.where<Person>()
                .equalTo(“lastName”,”Bond”)
                .findFirst()!!
  realm.executeTransaction { _ ->
          person.lastName = "DoubleOSeven"
          person.age = 50
      }
 //Hemos actualizado a la James Bond, ahora James DoubleOSeven
```

Las transacciones se pueden ejecutar en un thread distinto al de la UI, que es el que se ocupa cuando hacemos un executeTransaction, para hacer una ejecución en el background utilizamos executeTransactionAsync y Realm se encargará de ejecutar la transacción en un thread diferente, para no bloquear la UI.

Podemos ejecutar transacciones más complejas, pero como vemos en los ejemplos siempre utilizamos métodos de realm y las propiedades de los objetos. Esto simplifica mucho el trabajo y también nos permite utilizar la base de datos como si estuviéramos usando objetos.

Por ejemplo si queremos encontrar a todas las personas entre 25 y 35, que tienen apellido Perez o Soto, ordenados alfabéticamente, podemos hacer lo siguiente:

```
val results = realm.where<Person>()
            .between("age", 25, 35)
           .equalsTo("lastName", "Perez")
        .or()
        .equalsTo("lastName", "Soto")
         .sort(Person::age.name, Sort.DESCENDING)
            .findAll()
```

Realm es una tremenda herramienta, revisemos ObjectBox

ObjectBox

Una nueva opción que aparece hace muy poco, y que se presenta como una base de datos hecha para la IoT. Con esto en mente se diseñó para que fuera muy rápida. Sus principales ventajas son

- Rápida: diseñada para tener una performance sobresaliente, ganando en casi todos los benchmarks contra Room y Realm
- API basada en Objetos: ObjectBox no es un ORM, está construida en base a objetos 100%, no tiene columnas ni filas, sólo objetos, ni sql.
- QueryBuilder: Permite preguntar por objetos, hacer queries 100% basadas en objetos con verificación en tiempo de compilación, como lo hace Room con las queries de SQL.
- No es necesario migrar la base de datos, ObjectBox lo hace por uno, agrega automáticamente todos los cambios.

Parece no tener muchas desventajas a primera vista, salvo lo nueva que es, pero al parecer no tiene desventajas que la hagan un problema. Quizás en el futuro y con más uso, se encontrarán más desventajas.

Ejemplos de OpenBox

Para definir un Entity hacemos lo siguiente, los id se anotan con la anotación @Id

```
   @Entity
   data class Note(
   @Id var id: Long = 0,
   val text: String? )
```

Similar a lo que ya hemos visto en Room y Realm.

Para construir entidades ObjectBox ocupa una de estas dos opciones en Kotlin:

- El constructor por defecto, es decir el que tiene todos los argumentos asignables. Siempre prefiere este constructor.
- Un constructor sin todos los argumentos, esto lo podemos construir con un constructor custom o agregando valores por defecto a los argumentos del constructor por defecto. Este se elige después que falló llamar al constructor con todos los argumentos.

Por ejemplo esta clase permite ocupar la segunda opción

```
@Entity
data class Note(
        @Id var id: Long = 0,
        var text: String? = null,
        var comment: String? = null,
        var date: Date? = null
)
```

Para inicializar la base de datos hacemos uso de la clase MyObjectBox, de la siguiente manera, nuevamente esto se recomienda hacer una vez y en la clase Application de nuestra aplicación.

```
class ExampleApplication : Application() {

    lateinit var boxStore: BoxStore
    override fun onCreate() {
      super.onCreate()
      boxStore = MyObjectBox
                .builder().androidContext(this).build()
    }
}
```

Hemos creado nuestra base de datos, queda referenciada en boxStore, para interactuar con los otros objetos o entidades debemos crear un box. Eso lo podemos hacer de la siguiente manera:

```
var notesBox = ExampleApplication.boxStore.boxFor()
```

Con este box podemos ejecutar Queries, por ejemplo ordenar todas las notas del sistema alfabéticamente

```
var notesQuery = notesBox.query {
order(Note_.text)
}
```

Para obtener todas las notas del sistema

```
val notes = notesQuery.find()
```

Para crear una nueva nota y agregarla a nuestra base de datos

```
val noteText = "texto de la nota"
val df = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM)
val comment = "Agregada el " + df.format(Date())
//creamos la nota igual que como creamos un objeto
val note = Note(text = noteText, comment = comment, date = Date())
// para agregar la nueva nota a la base de dato hacemos put
notesBox.put(note)
```

Como se puede ver la simplicidad es la regla en esta herramienta, entonces la pregunta que nos debemos hacer ahora es:

¿Cuál ocupo en mi proyectos?

Room vs Realm vs ObjectBox, ¿Cúal es la mejor para mi proyecto?

Esta pregunta no tiene una respuesta simple ni obvia, porque va a depender de muchos factores. Si bien uno puede tener preferencias por razones personales, hay que aplicar una mezcla entre facilidad de uso, rendimiento y mantenimiento. Hay que tener claro que las herramientas de terceros pueden quedarse sin soporte, tener errores que nos son corregidos, etc. Al final depender de un tercero me facilita el trabajo, pero no me asegura fiabilidad todo el tiempo.

Es normal encontrar errores, incluso de Google, que no tienen solución oficial, y se termina “hackeando” la solución, o se busca un camino alternativo.Veamos algunos criterios para decidir, revisaremos algunos test realizados por terceros, para tener un punto de partida en esta discusión. Las comparativas completas se referencian en la bibliografía.

Desempeño y Tamaño

Si es por velocidad pura de operaciones CRUD, podemos mirar estos gráficos de una comparativa de AndroidProDev, donde el eje Y muestra el tiempo en milisegundos y el eje X muestra la cantidad de elementos en la base de datos, de 10000 a 50000. Los tests se corrieron 10 veces para cada librería, por cada tramo 10k, 20k, 30k, 40k y 50k. En los gráficos se muestran los promedios. El código de los tests se encuentra disponible en https://github.com/mac229/Databases/blob/master/app/src/androidTest/java/com/maciejkozlowski/databases/DatabasesTest.kt

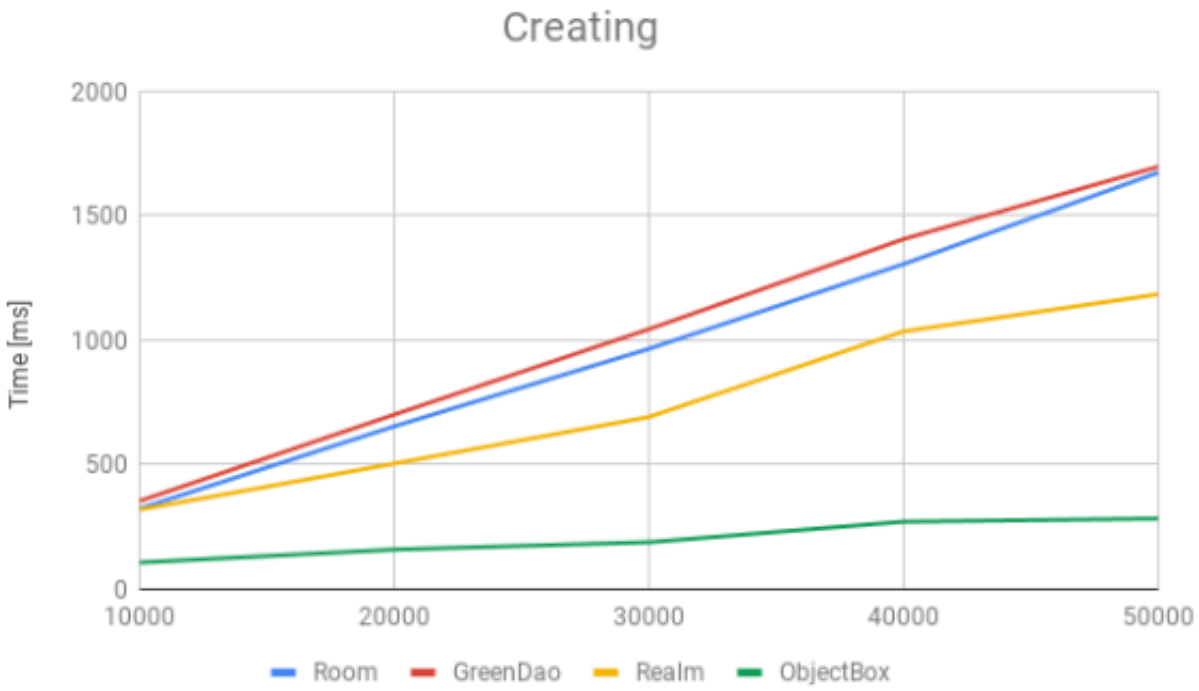

Imagen 12. Elementos creados en las bases de datos en el tiempo.

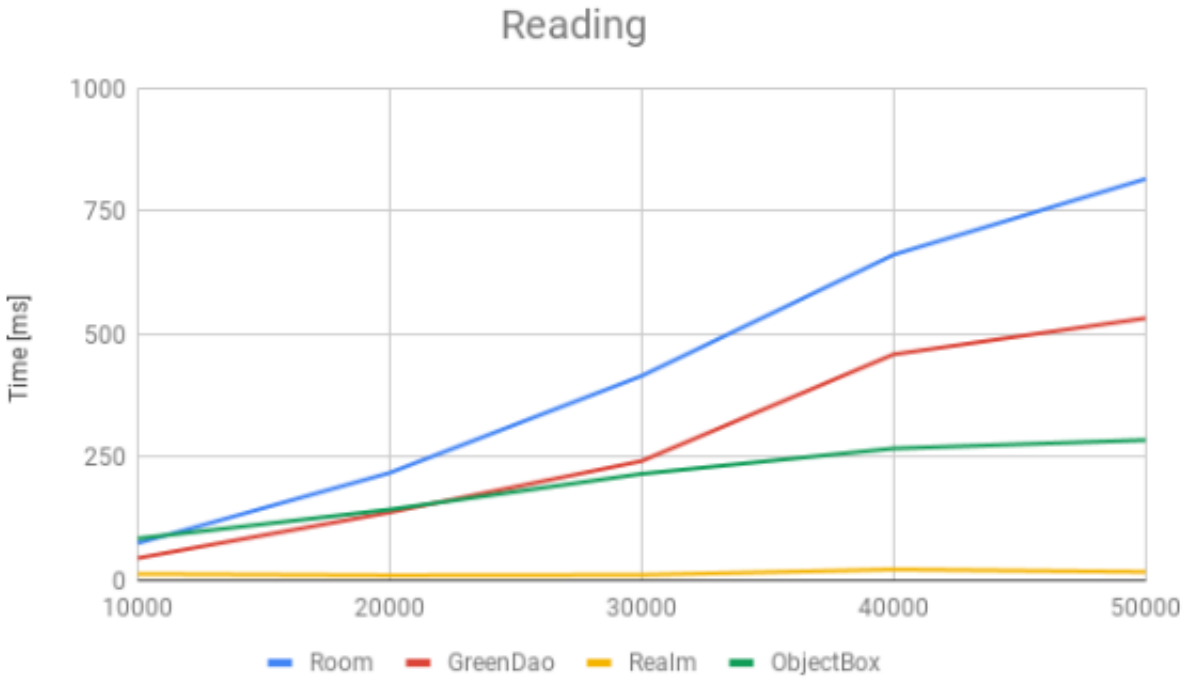

Imagen 13. Elementos leido en las bases de datos en el tiempo.

Imagen 14. Elementos actualizados en las bases de datos en el tiempo.

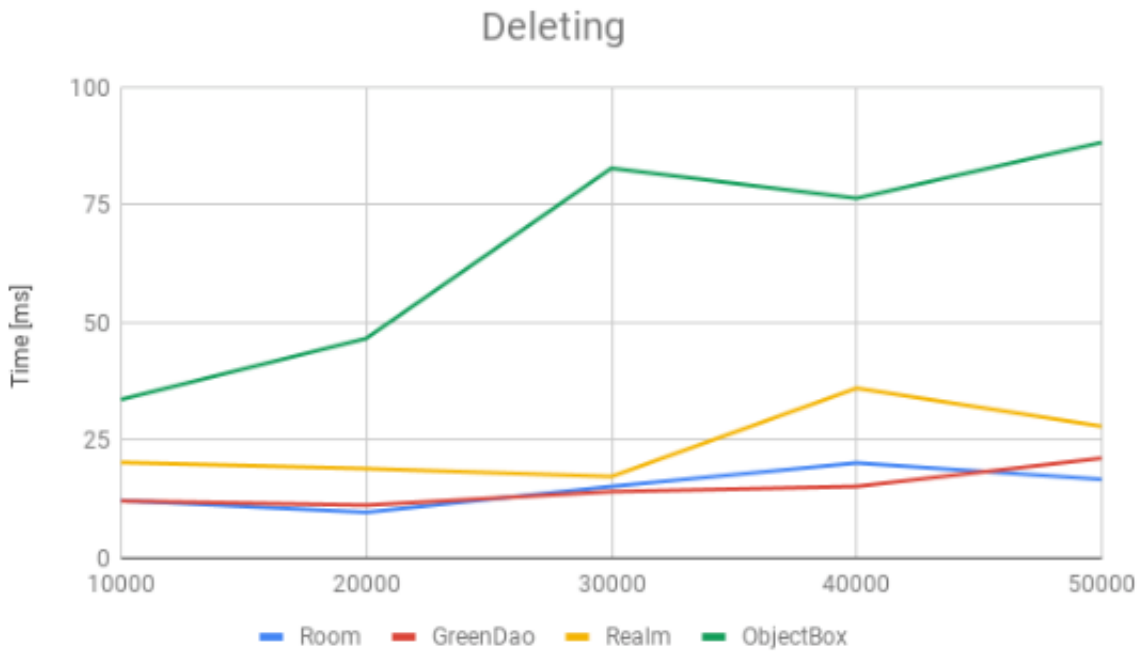

Imagen 15. Elementos eliminados en las bases de datos en el tiempo.

En estos gráficos aparece una cuarta opción, en rojo llamada GreenDao, que es de los mismos que crearon ObjectBox, la gente de GreenRobot, un gran aporte a Android en general con tremendas librerías. GreenDao, ORM SQL, no es parte de nuestra comparativa por lo que la ignoramos.

Como vemos ObjectBox tiene un tremendo desempeño en casi todas las categorías, salvo borrar donde se dispara su tiempo. Realm tiene un desempeño notable en la lectura, donde el tiempo es prácticamente constante. Room tiene un desempeño casi lineal con respecto a la cantidad de data en la base de datos, y se destaca en el borrado de datos. Todo esto hace mucho sentido con el tipo de implementación de las librerías:

- ObjectBox está orientada a desempeño en las operaciones comunes Crear, Leer y Actualizar.
- Realm está orientada a ser rápida sobretodo en lectura y actualizaciones, recordando la simplicidad de sus queries y transacciones. El generar las relaciones de objetos, en NoSQL se generan “documentos”, es caro, pero leer y actualizar es muy veloz.
- Room está diseñada con SQL en mente, trabaja sobre SQLite, por lo que se explica muy bien su desempeño lineal en todas las operaciones. Borrar data en SQL es muy veloz, mucho más que en NoSQL.

En cuanto a tamaño podemos mirar el siguiente gráfico, que nos muestra cual el resultado final de cada base de datos, y aquí también se ve la relación entre el diseño de la librería, su velocidad y que podemos esperar en casos extremos:

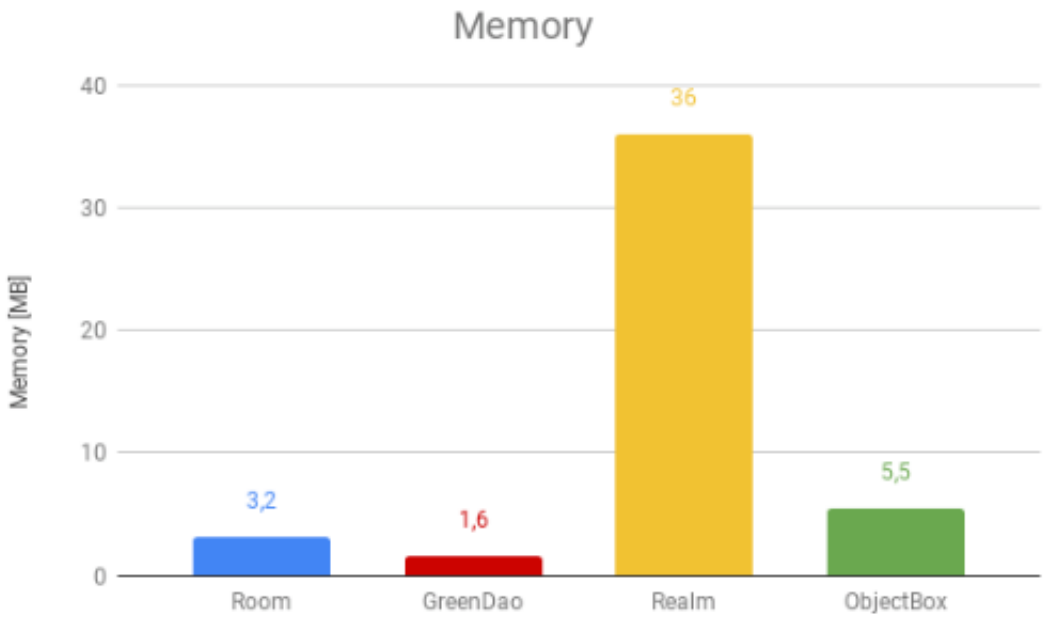

Imagen 16. Uso de memoria de una base de datos.

Hace sentido que Realm se la que ocupa más espacio, su tiempo de lectura era constante, independiente del tamaño de los datos, eso implica que la Base de Datos NoSQL está almacenando todos las relaciones y los resultados posibles de sus entidades, esto también se ve reflejado en ObjectBox, pero su implementación de objetos le da un mucho mejor resultado que Realm. Pero ObjectBox es casi el doble que la base de datos de Room, que se puede dar el lujo de depender de sus keys foráneas para reconstruir datos, por lo que no debe guardar ninguna relación extra, como sí lo tienen que hacer las NoSQL.

Entonces, ¿cuál me conviene?

La respuesta no es sencilla, pero depende directamente de las necesidades del projecto. Si necesitamos desempeño por sobre cualquier otra variable, tenemos Realm y ObjectBox. Si tenemos limitaciones de espacio, cosa habitual en los dispositivos móviles, ObjectBox es una buena opción, también lo es Room. Normalmente las base de datos SQL se utilizan para almacenar data que debe ser confiable y recuperable, no importando la lentitud del acceso. Cuando necesitamos velocidad, tenemos data que no cambia muy seguido, es media estática, y podemos perderla sin problemas, se recomienda NoSQL. Caso emblema de NoSQL, una memoria caché, que es volátil y podemos perder.Lo importante es tener claro que requerimientos tiene la aplicación, que dispositivos debemos soportar y si es necesario una librería compleja para resolver el problema.