{desafío}
latam_

Persistencia y bases de datos _

Parte II

Room

Objetos vs Tablas

- Quiero modelar una agenda de contactos con dos objetos
- Debo persistir estos objetos en una base de datos, crear tablas y columnas
- Debo generar mucho código intermedio o boilerplate, para hacer este mapeo entre objetos y tablas
- Mucho código, más problemas de mantención, menos código que realice las tareas que quiero en mi aplicación
- Podemos solucionar estos problemas con un ORM

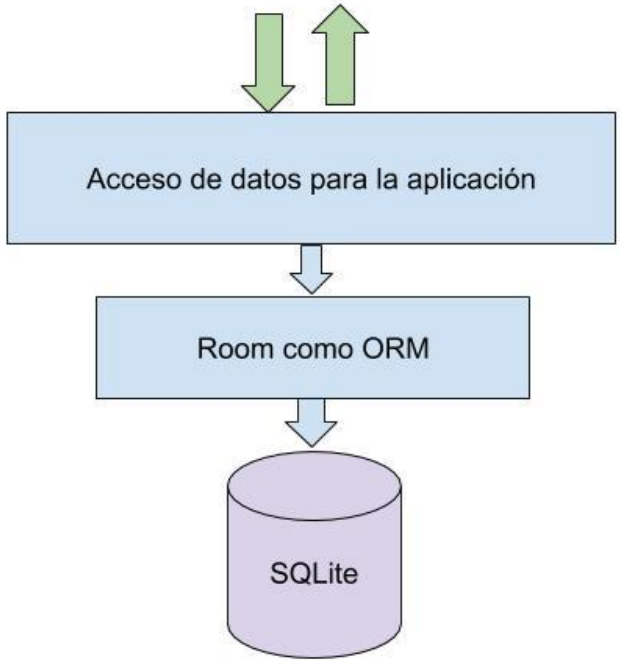
```
class Person(var name:String,
             var surname:String,
             var phonenumber:String,
             var address: Address,
             var email:String)


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

ORM

- Object Relational Mapper
- Herramienta que nos permite generar código boilerplate entre nuestra base de datos y nuestros objetos del modelo
- Nos permite traducir de objetos a tablas y de tablas a objetos de manera más simple
- Menos código intermedio, más código que ejecuta tareas valiosas

Room como ORM

- Android no inventó los ORM
- Room nace debido a una iniciativa de Google para tener mejores aplicaciones
- Room nos permite generar el código intermedio
- Nos permite mapear tablas y objetos
- Es bastante sencillo de utilizar y sigue las convenciones establecidas en la comunidad Android, por ejemplo ocupa Fachada como patrón de implementación



Componentes de Room

Room cuenta con 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.
- Database: Un contenedor que mantiene una referencia y actúa como único punto de acceso a la base de datos.

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)
```

- @Entity le dice a room que esta clase es un entity, tableName le dice el nombre de la tabla que almacena este entity
- @ColumnInfo le dice a room que esto es una columna, con name “id”, de la tabla
- @PrimaryKey le dice a room que es una clave primaria, y autoGenerate = true que se debe autoincrementar
- Veremos otras anotaciones más adelante

DAO

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

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

- Un dao es una interfaz no una clase, debido al patrón fachad que utiliza Room
- @Dao le dice a room que esto es un DAO, Room implementará todos sus métodos
- @Query nos permite ejecutar queries SQL con chequeo de errores en Compile Time. Con SQLite puro sólo sabremos si algo está mal en RunTime
- @Insert le dice a Room que este método hará un insert a la base de datos

Database

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

- Este componente une las entities con los daos en Room
- @Database le dice a Room que esta clase es un database, en la anotación entregamos un arreglo de entities que son parte de la base de datos en la propiedad entities = . Además, le decimos que versión de la base de datos está utilizando, en este caso version = 1
- Los métodos definidos son del tipo abstract, nuestra clase hereda de RoomDatabase

Crear una instancia de la base de datos

- Se recomienda inicializar sólo una vez la base de datos, Singleton
- En este caso lo mejor es hacerlo en una clase que extienda de Application, en su método onCreate, que se ejecuta sólo una vez por instancia de la aplicación.
- databaseBuilder recibe el Context, la clase anotada con @Database y el nombre del archivo de la base de datos

```
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()
   }
}
```

Otras anotaciones Room

- @Update nos permite generar métodos para actualizar objetos que ya existen en la base de datos
- @Ignore nos permite anotar campos que no queremos que se almacenen en la base de datos
- @Delete nos permite anotar métodos para eliminar objetos o colecciones completas de la base e datos

```
@Update
fun updateAddress(Address address)

@Ignore
val avatar:Bitmap

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

Otras anotaciones Room

- @ForeignKey o ForeignKey sin @, nos permite decir que keys son foráneas en nuestro Entity
- Nos permite definir el comportamiento cuando se elimina el objeto padre, y podemos mapear la columna hijo o destino, y la columna padre u origen. En este caso

```
@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
)
```

Otras anotaciones Room

- @Relation nos permite generar las relaciones comunes en SQL, por ejemplo 1:1 o 1:N, etc
- En este ejemplo vemos que Relation tiene definida la relación entre la columna padre y la columna de la entidad que resulta de esta relación, en este caso tenemos un usuario con todas sus mascotas.
- Room es capaz de identificar que tipo de entidad se retorna, en este caso un Pet, pero si queremos retornar otra le podemos decir cual con la propiedad entity = dentro de la anotación.

```
@Entity
class User(@PrimaryKey
           val id:Int,
          // other fields)
@Entity
class Pet (@PrimaryKey
           val id:Int,
           val userId:Int,
           val name:String,
           // other fields)
@Entity
class UserNameAndAllPets(
  val id:Int,
  val name:String,
  @Relation(parentColumn = “id”, entityColumn = “userId”)
  val pets:List<Pet>)
```

Migraciones

- Las migraciones en Room se deben ejecutar a mano y en secuencia
- Debemos definir una subclase de la clase Migration, con la versión inicial y final como parámetros
- Luego debemos agregar todas estas migraciones al construir la base de datos con dataBaseBuilder
- Así nos aseguramos que cuando un usuario actualice la aplicación, no se pieda la data que ya tiene

```
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()
```

Realm y ObjectBox

Realm - Ventajas

- Base de datos NoSql, diseñada para tener un gran desempeño en lectura de la base de datos
- Simple con código corto y conciso
- Veloz, escrita en C++ tiene operaciones CRUD muy veloces
- Live Objects, los objetos se acceden por referencia, de forma lazy, esto asegura que no copiamos datos, y tenemos siempre la última versión de la información
- Documentación y comunidad de primer nivel
- Soporte out of the box para Json
- Segura, permite encriptación en la base de datos y los datos
- Reactiva, juega muy bien con RxJava
- Incluye adapters para las listas más comunes en la UI de Android

Realm - Desventajas

- No autoincrementa valores, por lo que debemos definir funciones que lo hagan
- Los model classes están limitados a una pequeña personalización
- No es multi-thread
- Problemas con el tamaño, puede alcanzar un gran tamaño con muchos datos
- Es necesario un binario por arquitectura ABI soportada, aumentando el tamaño del APK
- Tiene problemas de performance en ciertas operaciones con base de datos muy complejas

Ejemplos Realm

- Para crear un Model Class o Entity debemos definir un open class y utilizar las anotaciones definidas en Realm, en este caso es necesario agregar valores por defecto siempre
- La inicialización se hace una sóla vez
- Obtener una instancia de la base de datos es tan fácil como ejecutar la última línea

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

//Inicializando la base de datos
class ExampleApplication : Application() {
 override fun onCreate() {
  super.onCreate()
  //al igual que Room sólo inicializamos una vez cuando
  //parte la aplicación
    Realm.init(this)
  }
}

//para obtener una instancia de la base de datos
realm = Realm.getDefaultInstance()
```

Ejemplos Realm

- Los queries de búsqueda son sencillos y se ejecutan como si uno estuviera interactuando con objetos, hay muchos métodos para utilizar. En este caso usamos equalTo
- Para escribir en la DB hacemos transacciones, de esta forma aseguramos que se ejecuten de manera segura y confiable. Hay del tipo síncronas, como la que está aquí, y asíncronas.

```
//un query sencillo, buscando por edad
val results = realm
              .where<Person>()
              .equalTo("age", 55).findAll()

 // 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
```

Ejemplos Realm

- Podemos ejecutar queries tan complejas como queramos, encadenando diferentes métodos y ocupando distintos criterios.
- Para más información https://realm.io/docs

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

Ejemplos Realm

- Para crear un Model Class o Entity debemos definir un open class y utilizar las anotaciones definidas en Realm, en este caso es necesario agregar valores por defecto siempre
- La inicialización se hace una sóla vez
- Obtener una instancia de la base de datos es tan fácil como ejecutar la última línea

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

//Inicializando la base de datos
class ExampleApplication : Application() {
 override fun onCreate() {
  super.onCreate()
  //al igual que Room sólo inicializamos una vez cuando
  //parte la aplicación
    Realm.init(this)
  }
}

//para obtener una instancia de la base de datos
realm = Realm.getDefaultInstance()
```

ObjectBox - Ventajas y Desventajas

Creada con IoT en mente, es la opción más novata y moderna de las tres

- Rápida, diseñada para tener performance sobretodo, gana en casi todos los benchmarks
- API 100% basada en objetos, no es un ORM, no hay SQL sólo objetos
- QueryBuilder nos permite interactuar con la base de datos, las consultas son lo mismo que interactuar con objetos
- No es necesario migrar la base de datos, ObjectBox lo hace automáticamente
- No tiene desventajas conocidas, salvo lo nueva que es
- más información en https://objectbox.io/mobile

Ejemplos ObjectBox

- Para crear entidades utilizamos @Entity
- Puede utilizar el constructor por defecto con todos los argumentos, o un constructor secundario, con menos argumentos, para construir un entity
- El segundo constructor se ocupa como fallback, si falla el primero va al segundo

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

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

Ejemplos ObjectBox

- La base de datos se inicializa una sóla vez como Room y Realm.
- Para obtener una instancia de la DB usamos el método boxFor() de la variable inicializada en onCreate
- Las queries se pueden ejecutar en bloques tipo transacción o como métodos definidos, dependiendo qué operación estemos realizando

```
class ExampleApplication : Application() {

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

var notesBox = ExampleApplication.boxStore.boxFor()

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

val notes = notesQuery.find()
```

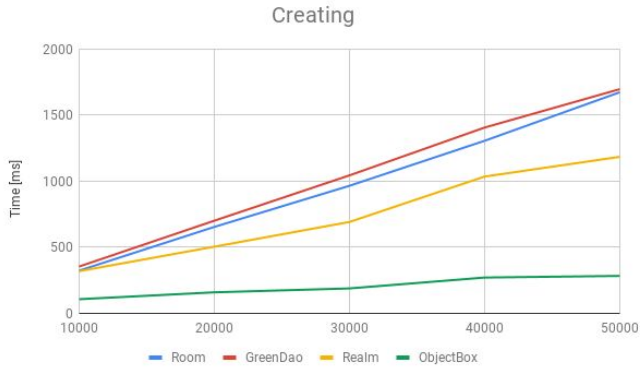
Ejemplos ObjectBox

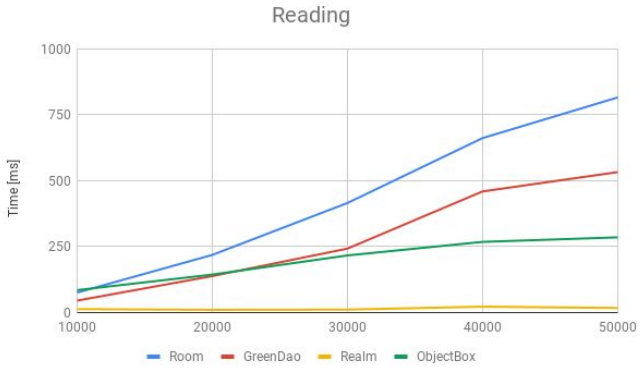
- Crear y agregar datos a la base de datos es muy simple
- Las entidades se crean igual que un objeto, incluso podemos ocupar parámetros con nombre
- Para agregar a la base de datos utilizamos el método put
- Más información en https://objectbox.io/mobile

```
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, con
//parámetros nombrados
val note = Note(text = noteText,
                comment = comment,
                date = Date())
// para agregar la nueva nota a la base de dato hacemos
//put
notesBox.put(note)
```

Comparativas de desempeño

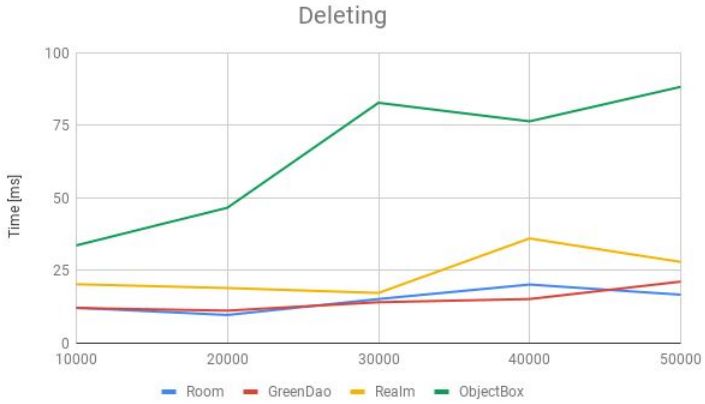
- Comparativa realizada por: https://proandroiddev.com/android-databases-performance-crud-a963dd7bb0eb
- Se corrió un test 10 veces por librería , para 10k, 20k, 30k, 40k y 50k, 50 tests por librería
- Se comparó Room, Realm, ObjectBox y GreenDao
- GreenDao y Room son SQL
- Realm y ObjectBox son NoSQL
- En los distintos gráficos podemos ver el desempeño en operaciones CRUD de las librerías

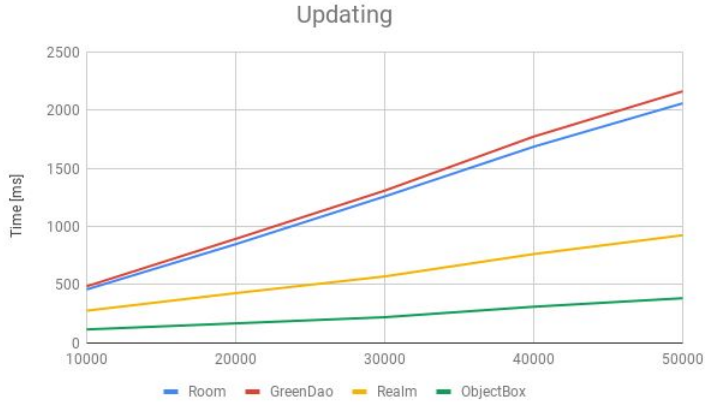




Comparativas de desempeño

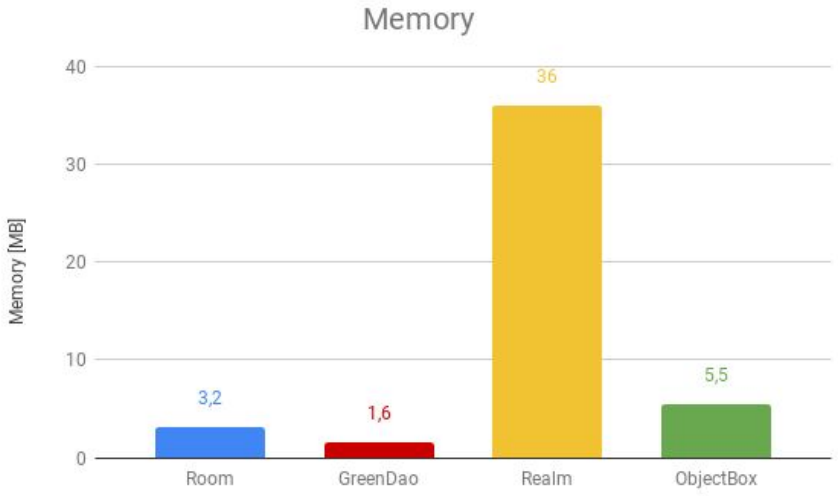
- Las base de datos SQL tienen tiempos lineales ascendentes en todas las categorías
- Las NoSql destacan en lectura, Realm tiene casi tiempo constante independiente la cantidad de datos
- El desempeño en Delete de objectBox es terrible, es su peor categoría
- Las base de datos SQL y NoSql son bastante predecibles en su comportamiento





Comparativas de desempeño

- El gran problema de las base de dato NoSQL, el tamaño. Esto se debe a su naturaleza de como almacenan los datos, generando todas las conexiones posibles para asegurar desempeño en la lectura
- ObjectBox tiene un tremendo desempeño general, pero su tamaño sigue siendo aproximadamente el doble de room.
- En conclusión, el uso de una base de datos depende de las necesidades de la aplicación y no la preferencia de una tecnología. No hay balas de plata.



{desafío} latam_

Academia de talentos digitales

www.desafiolatam.com