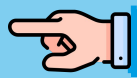
Acceso a datos en Android

La librería ROOM (Parte III)

{desafío} latam_

Implementar capa de acceso a datos en un aplicativo móvil utilizando la librería ROOM para otorgar persistencia de estados resolviendo el problema planteado

- Unidad 1:
  Acceso a datos en Android
- Unidad 2:
  Consumo de API REST
- Unidad 3:
  Testing
- Unidad 4:
  Distribución del aplicativo Android



{desafío} latam_

¿Qué aprenderás en esta sesión?

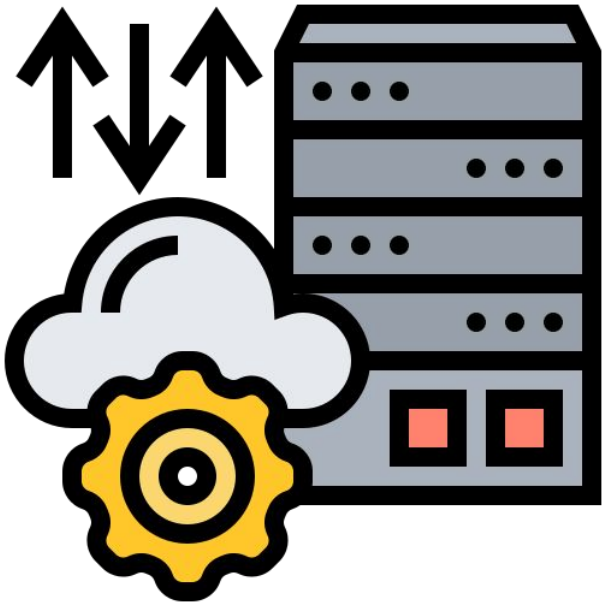
- *Describe el rol de la capa de abstracción Room y sus principales elementos tales como Entidad y DAO*

{desafío}
latam_

¿Sabes qué pasaría si modificamos una tabla en Room en una app que se encuentra en producción?

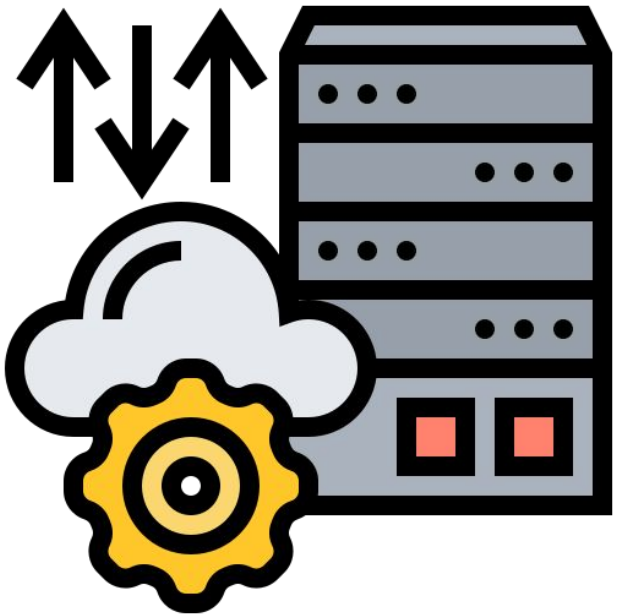
/* Migraciones en Room */

Migraciones en Room



- En Android Room, las migraciones se utilizan para controlar los cambios en el esquema de la base de datos a lo largo del tiempo. Cuando el esquema cambia, Room actualizará automáticamente el esquema de la base de datos en la próxima compilación, conservando los datos existentes si es posible.
- Las migraciones se definen mediante una serie de clases denominadas clases de migración. Cada clase de migración define los cambios que se realizarán en el esquema, como agregar o eliminar tablas, columnas o cambiar los tipos de columnas existentes.
- Las migraciones se definen utilizando la clase Migration, que toma dos enteros como parámetros. El primer entero es la versión "desde" y el segundo entero es la versión "hasta". Esto indicará la versión del esquema que va a cambiar la migración, a la versión a la que va a cambiar.

Migraciones en Room



- En la clase de migración, tendrá que usar el objeto SupportSQLiteDatabase de la base de datos Room y el método execSQL para ejecutar las instrucciones SQL para realizar los cambios necesarios en el esquema.
- Cuando se ejecuta la aplicación, Room comprueba la versión del esquema actual y la compara con la nueva versión del esquema. Si las versiones no coinciden, Room aplicará las migraciones necesarias para actualizar el esquema, en el orden correcto.
- Es importante tener en cuenta que las migraciones deben realizarse con cuidado, ya que si no se realizan correctamente, puede provocar la pérdida de datos. Además, Room solo aplicará las migraciones necesarias y, si hay varias migraciones, Room las aplicará una por una, en el orden correcto, por lo que es importante realizar un seguimiento de las versiones y los cambios realizados en el esquema.

Demostración "Room migration"

Room Migration

Digamos que tenemos una tabla User con el siguiente esquema:

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int,
    @ColumnInfo(name = "name") val name: String,
    @ColumnInfo(name = "email") val email: String
)
```

Ahora, digamos que queremos agregar una nueva columna llamada "phone_number" a la tabla Usuario. Crearemos una clase de Migración como esta:

```
class Migration1to2 : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE users ADD COLUMN phone_number TEXT NOT NULL DEFAULT ''")
    }
}
```

Esta migración cambiará el esquema de la versión 1 a la versión 2, y agregará una nueva columna llamada "phone_number" a la tabla Usuario, con un valor predeterminado de una cadena vacía.

/* Consultas SQL en Room */

Consultas SQL en Room

- En Android Room, una consulta SQL es una forma de interactuar con la base de datos escribiendo declaraciones SQL sin formato. Room te permite escribir estas declaraciones de una manera segura y conveniente al proporcionar un conjunto de anotaciones que se pueden usar para definir las consultas.

- En Room, puedes usar la anotación @Query para definir un método de consulta en una interfaz de objeto de acceso a datos (DAO). La anotación @Query toma un parámetro de cadena, que contiene la consulta SQL. La cadena de consulta puede incluir marcadores de posición para argumentos, que se pueden pasar al método de consulta como parámetros de método.

Consultas SQL en Room

```
@Dao
interface UserDao {

    @Query("SELECT * FROM users WHERE id =
:userId")
    fun getUserById(userId: Int): User

    @Query("SELECT * FROM users")
    fun getAllUsers(): List<User>
}
```

En este ejemplo, tenemos una interfaz UserDao que contiene dos métodos de consulta. El primer método, getUserById, toma un parámetro de ID de usuario y devuelve un objeto Usuario que corresponde al usuario con la identificación especificada. El segundo método, getAllUsers, devuelve una lista de todos los usuarios de la tabla de usuarios.

Consultas SQL en Room

```
@Dao
interface UserDao {

    @Query("SELECT * FROM users WHERE id =
:userId")
    fun getUserById(userId: Int): User

    @Query("SELECT * FROM users")
    fun getAllUsers(): List<User>
}
```

La anotación @Query también se puede utilizar para ejecutar declaraciones de actualización, inserción y eliminación. Room devolverá automáticamente el número de filas afectadas por la consulta.

Es importante tener en cuenta que Room validará la sintaxis de SQL en tiempo de compilación y también protegerá contra la inyección de SQL. Esto significa que no tiene que preocuparse por escapar de la entrada del usuario u otras preocupaciones de seguridad al escribir consultas SQL sin procesar en Room.

/* Room y Kotlin Coroutines */

Room y Kotlin Coroutines

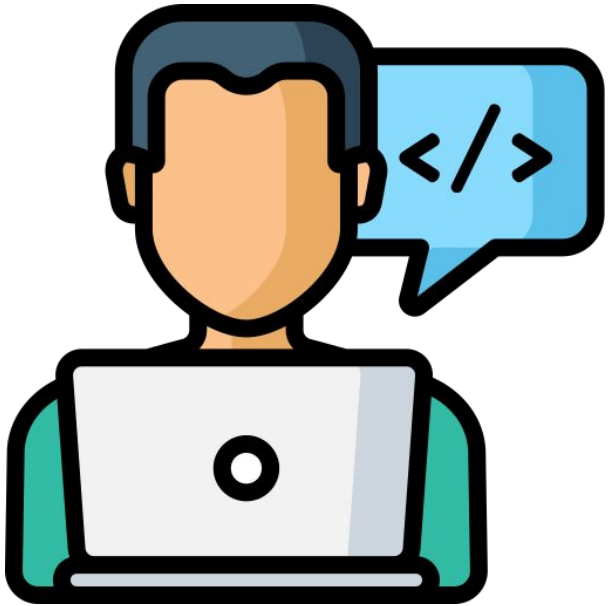
- **Coroutines:** Las coroutines son una forma liviana y eficiente de administrar la concurrencia y las operaciones asincrónicas en su código. Le permiten escribir código asíncrono que se ve y se comporta como código síncrono, lo que puede hacer que sea más fácil de entender y razonar. Las coroutines también pueden ayudar a evitar problemas comunes, como el infierno de devolución de llamada y las fugas de memoria.

- **Room:** Room es una biblioteca de base de datos que forma parte de los componentes de la arquitectura de Android. Proporciona una capa de abstracción sobre SQLite, lo que facilita el trabajo con bases de datos en su aplicación. Room puede ayudar a evitar problemas comunes, como el modelo manual de SQLite, subprocesos y fugas de memoria. Room también proporciona un poderoso conjunto de anotaciones que le permiten definir el esquema y las consultas de su base de datos, lo que hace que su código sea más legible y fácil de mantener.

Room y Kotlin Coroutines

Cuando se usan juntos, Coroutines y Room pueden facilitar el manejo de las operaciones de la base de datos de manera concurrente y eficiente. Las coroutines se pueden usar para ejecutar consultas y actualizaciones de bases de datos de forma asincrónica, sin bloquear el hilo principal. Esto puede mejorar el rendimiento y la capacidad de respuesta de su aplicación.

Room también brinda soporte para Coroutines a través de las funciones de suspensión, lo que le permite escribir operaciones de base de datos de una manera más natural y legible utilizando la API de flujo de coroutines.



Demostración
"Room y coroutines"

Room y coroutines

En la siguiente demostración seguiremos usando la clase User. Esta vez pon atención a la función insert en la DAO:

```
@Entity
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int,
    val name: String,
    val email: String
)

@Dao
interface UserDao {
    @Insert
    suspend fun insert(user: User)
}
```

Room y coroutines

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null
        fun getDatabase(context: Context): AppDatabase {
            val tempInstance = INSTANCE
            if (tempInstance != null) {
                return tempInstance
            }
            synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "user_database"
                ).build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

Room y coroutines

En tu ViewModel o Repository, puedes usar la DAO para insertar un usuario en la base de datos usando un Coroutine.

```
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val userDao = AppDatabase.getDatabase(application).userDao()
    private val scope = CoroutineScope(Dispatchers.IO)

    fun insertUser(user: User) {
        scope.launch {
            userDao.insert(user)
        }
    }
}
```

TIP: Es recomendable siempre usar “suspend” si vas a insertar, borrar o actualizar

Próxima sesión...

- *Implementa capa de acceso a datos en un aplicativo Android utilizando la librería ROOM como capa de abstracción para resolver un problema*

{desafío}
latam_

Academia de
talentos digitales