Sesión 3-05 Clase del 18 de diciembre

En esta sesión realizaremos una app que accede a datos de una base de datos embebida en la app. El acceso a la base de datos se realiza mediante una ORM llamada Room que es una librería ligera para acceder mediante mapeo objeto-relacional a una base de datos SQLite.

Para programar usando Room se ha de definir con una arquitectura en tres capas cuyas clases se marcan mediante anotaciones:

- **Entity**: Representa una tabla de la base de datos.
- **DAO (Data Access Object):** Interface que define los métodos para acceder a los datos de la base de datos.
- **RoomDatabase:** Clase principal de la arquitectura que define el conjunto de entidades y DAOs que se pueden usar en la base de datos y que sirve como punto de acceso principal.

1.- Para comenzar, en el archivo Gradle de módulo debemos agregar lo siguiente:

- En la sección de plugins, debes agregar esta referencia:

```
id("com.google.devtools.ksp") version "1.9.20-1.0.14"
```

- En la sección **dependencias** debes agregar las siguientes:

```
// Room runtime
implementation(libs.androidx.room.runtime)
ksp(libs.androidx.room.compiler)

// Room KTX
implementation(libs.androidx.room.ktx)

// Corroutinas
implementation(libs.kotlinx.coroutines.android)
```

2.- Vamos a crear las primeras entities. Declarar un entity con Room es bastante similar a hacerlo con JPA.

En un paquete **data.entities**, añadimos un data class Familia para una Entity que mapea una tabla **familias (de animales)**

```
@Entity(tableName = "familias")
data class Familia(

    @PrimaryKey
    @ColumnInfo(name = "id")
    val id: Int,

    @ColumnInfo(name = "nom_fam")
    val nombre: String,
```

```
    @ColumnInfo(name = "desc_fam")
    val descripcion: String =""
)
```

Ahora definimos otro data class Animal con información sobre animales y que incluye una clave ajena para indicar la familia a la que pertenece cada animal.

```
@Entity(tableName = "animales")
data class Animal(

    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    val id: Int = 0,

    @ColumnInfo(name = "nombre")
    val nombre: String,

    @ColumnInfo(name = "caracteristicas")
    val caracteristicas: String="",

    @ColumnInfo(name = "imagen_url")
    val imagenUrl: String="",

    @ColumnInfo(name = "familia_id")
    val familiaId: Int
)
```

Aunque no es necesario que lo hagamos, podríamos especificar que **familiaId** se comporte como restricción de **Foreign Key.** Para ello , en la anotación **@Entity** deberíamos tener lo siguiente:

```
@Entity(tableName = "animales",
    foreignKeys = [
        ForeignKey(
            entity = Familia::class,
            parentColumns = ["id"],
            childColumns = ["familia_id"],
            onDelete = ForeignKey.CASCADE
        )
    ])
```

3.- Creamos las interfaces DAO para las entities. Creamos los archivos kt correspondientes a las entities Alumno y Familia. Deberían estar en el package **data.daos.**

Interface FamiliaDao

```
@Dao
interface FamiliaDao {

    // Insertar una familia
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun insertFamilia(familia: Familia): Int

    // Obtener todas las familias
```

```
    @Query("SELECT * FROM familias")
    suspend fun getAllFamilias(): List<Familia>

    // Buscar familia por ID
    @Query("SELECT * FROM familias WHERE id = :familiaId")
    suspend fun getFamiliaById(familiaId: Int): Familia?

    // Actualizar una familia
    @Update
    suspend fun updateFamilia(familia: Familia)

    // Eliminar una familia
    @Delete
    suspend fun deleteFamilia(familia: Familia)
}
```

- Se han definido en el interface los métodos CRUD básicos para la Entity Familia y por tanto para la tabla familias.
- En el método para insertar se indica que si se inserta una familia con un ID existente, se aborte o rechace la inserción. El método devuelve el valor del id correspondiente a la fila insertada.
- Hay varios métodos en los que necesitamos especificar la sql de la consulta.
- Los métodos get, como vemos, devuelven un objeto Familia o una lista de objetos Familia según sea el caso.

Interface AnimalDao

```
@Dao
interface AnimalDao {

    // Insertar un animal
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAnimal(animal: Animal): Long

    // Obtener todos los animales
    @Query("SELECT * FROM animales")
    suspend fun getAllAnimales(): List<Animal>

    // Buscar animal por ID
    @Query("SELECT * FROM animales WHERE id = :animalId")
    suspend fun getAnimalById(animalId: Int): Animal?

    // Actualizar un animal
    @Update
    suspend fun updateAnimal(animal: Animal)

    //Actualización personalizada
    @Query("UPDATE animales SET familia_id = :nuevoFamiliaId WHERE fa-
milia_id = :anteriorFamiliaId")
    suspend fun updateFamiliaIdInAnimal(anteriorFamiliaId: Int, nuevo-
FamliaId: Int)
```

```
    // Eliminar un animal
    @Delete
    suspend fun deleteAnimal(animal: Animal)
}
```

- En el método para insertar se indica que si se inserta una familia con un ID existente, se reemplace el existente con los datos del nuevo.
- Se ha definido un método update personalizado para hacer una modificación en todos los animales que tengan un determinado id de familia para que tengan uno nuevo. Es necesario hacerla ya que el método **updateAnimal** es para poder modificar todos los datos de un solo animal.

4.- Creamos en al package data.database un archivo AppDatabase.kt para contener la clase **AppDatabase** usada para conectar las entities con los DAOs y con la base de datos subyacente. Esta clase hereda de la clase **RoomDatabase.**

```
@Database(
    // Se agregan la entidades
    entities = [Animal::class, Familia::class],
    // Cambiar este número si se modifica la estructura de la DB
    version = 1,
    // Cambiar a false si no se quiere exportar el esquema
    exportSchema = true
)
abstract class AppDatabase : RoomDatabase() {

    // DAOs
    abstract fun animalDao(): AnimalDao
    abstract fun familiaDao(): FamiliaDao

    // Configuraciones adicionales (opcional)
}
```

5.- Ya tenemos la arquitectura completa de nuestra base de datos. Vamos a probar una conexión a la base de datos y un uso de algunos de los métodos de acceso que hemos declarado en los DAOs.

Como es una prueba, todo esto lo vamos a realizar en **MainActivity.**

a) Crea un objeto que incluye una función que devuelve una conexión a la base de datos. Al crear la conexión, se crea el fichero de base de datos, si no existe aún

```
object DatabaseProvider {
    private var INSTANCE: AppDatabase? = null

    fun getDatabase(context: Context): AppDatabase {
        return INSTANCE ?: synchronized(this) {
            val instance = Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "my_database.db" // Nombre a elegir
```

```
            ).build()
            INSTANCE = instance
            instance
        }
    }
}
```

b) Dentro del SetContent de MainACtivity (no creamos una pantalla) cargamos este código para conectar con la base de datos y usar el DAO de familias para añadir o insertar varias familias en la base de datos y consultar después las familias cargadas.

```
 val database = DatabaseProvider.getDatabase(this)
val familiaDao = database.familiaDao()
//Insertar los datos de una lista de familiar
insertarFamilias(familiaDao)
//Ver los datos de las familias
verFamilias(familiaDao)
```

c) Creamos ahora la función **insertarFamilias** que insertará las familias de una lista usando el método **insertFamilia** de **FamiliaDao.** Los métodos de una DAO se deben ejecutar siempre dentro de una corrutina.

```
private fun insertarFamilias(familiaDao: FamiliaDao) {
    val familias = listOf(
        Familia(id=1,nombre = "Cérvidos", descripcion = "Familia de
los ciervos, renos y alces."),
        Familia(id=2,nombre = "Bóvidos", descripcion = "Familia de los
toros, búfalos y antílopes."),
        Familia(id=3,nombre = "Mustélidos", descripcion = "Familia de
los tejones, nutrias y hurones.")
    )
    CoroutineScope(Dispatchers.IO).launch {
        // Inserta cada familia en la base de datos
        familias.forEach { familia ->
            familiaDao.insertFamilia(familia)
            Log.i("ACCESO_ROOM", familia.nombre)
        }
    }
}
```

d) Creamos ahora la función **verFamilias** que obtendrá y mostrará las familias existentes usando el método **listAllFamiliasa** de **FamiliaDao.**

```
private fun verFamilias(familiaDao: FamiliaDao) {
    var familias= emptyList<Familia>()
    CoroutineScope(Dispatchers.IO).launch {
        familias = familiaDao.getAllFamilias()
    }
    familias.forEach { familia ->
        Log.i("ACCESO_ROOM", "ID=${familia.id}  Nom-
bre=${familia.nombre}")
    }
}
```

6.- Realiza el codelab **Como conservar datos con Room.**

https://developer.android.com/codelabs/basic-android-kotlin-compose-persisting-data-room?hl=es-419#0