Usando room en la Agenda

Descargar estos apuntes

🎓 Caso de Estudio

Vamos a partir de nuestra Agenda de Contactos con navegación y vamos a añadir a añadir como fuente de datos una BD local de SQLite a la que accederemos usando el ORM room proporcionado por Jetpack Compose.

> **Solución**
>
> Si te surge alguna duda o tienes dificultades para completar este caso de estudio. Puedes descargar la solución de este caso de estudio del siguiente enlace: **propuesta de solución**

Paso 1: Añadir dependencias y paquetes

1. Para usar room en nuestra aplicación debemos añadir las siguientes dependencias en el fichero `build.gradle.kts` del módulo `app`:

```
dependencies {
    ...
    // Room
    val roomVersion = "2.6.1"
    implementation("androidx.room:room-runtime:$roomVersion")
    annotationProcessor("androidx.room:room-compiler:$roomVersion")
    ksp("androidx.room:room-compiler:$roomVersion")
    implementation("androidx.room:room-ktx:$roomVersion")
    // optional - Paging 3 Integration
    implementation("androidx.room:room-paging:$roomVersion")
}
```

2. Crearemos el paquete `.data.room` dentro del cual definiremos las clases que nos permitirán acceder a la BD.

Paso 2: Definir la entidad Contacto

Dentro del paquete `.data.room` definiremos la entidad `Contacto` que será la clase que represente sus datos en la BD.

```
    @Entity(tableName = "contactos")
    data class ContactoEntity(
        @PrimaryKey(autoGenerate = true)
        @ColumnInfo(name = "id")
        val id: Int,
        @ColumnInfo(name = "nombre")
        val nombre: String,
        @ColumnInfo(name = "apellidos")
        val apellidos: String,
        @ColumnInfo(name = "telefono")
        val telefono: String,
        @ColumnInfo(name = "email")
        val email: String,
14      @ColumnInfo(name = "foto",  typeAffinity = ColumnInfo.BLOB)
        val foto: String?,
        @ColumnInfo(name = "categorias")
        val categorias: String
    )
```

Paso 3: Definir conversores de tipos

Fíjate en el código anterior que la **imagen del contacto**, la gradaremos en la BD como un tipo `BLOB` que es un **array de bytes**. Sin embargo, en la entidad `ContactoEntity` la imagen la tenemos como un **String** que será una **cadena en base64** tal y como la usamos en nuestro modelo. Para ello debemos definir un conversor de tipos que nos permita convertir de un tipo a otro. Es por eso que crearemos la clase `RoomConverters` dentro del paquete `.data.room`.

```
class RoomConverters {
    @TypeConverter
    fun toBlob(value: ByteArray?): String? = Imagenes.blobToBase64(value)

    @TypeConverter
    fun fromBlob(value: String?): ByteArray? = Imagenes.base64ToBlob(value)
}
```

Paso 4: Definir nuestro DAO sobre la entidad Contacto

Vamos a definir el DAO sobre la entidad `ContactoEntity`. Para ello crearemos la **interfaz** `ContactoDao` dentro del paquete `.data.room`. En el, definiremos los métodos que nos permitirán realizar las operaciones **CRUD** sobre dicha entidad.

> 📌 **Nota** Fíjate que todos los métodos son **suspend**. De esta manera estamos indicando que son métodos que se ejecutarán en un hilo secundario (`Dispatchers.IO`) y que por tanto no bloquearán el hilo principal.

```
@Dao
interface ContactoDao {
    @Insert
    suspend fun insert(contacto : ContactoEntity)

    @Delete
    suspend fun delete(contacto : ContactoEntity)

    @Update(onConflict = OnConflictStrategy.ABORT)
    suspend fun update(contacto : ContactoEntity)

    @Query("SELECT COUNT(*) FROM contactos")
    suspend fun count(): Int

    @Query("SELECT * FROM contactos")
    suspend fun get(): List<ContactoEntity>

    @Query("SELECT * FROM contactos WHERE id = :id")
    suspend fun get(id: Int): ContactoEntity
}
```

Paso 5: Definición de la BD

Definimos la clase abstracta `AgendaDb` que extiende de `RoomDatabase` y que nos permitirá acceder a la BD. En ella definiremos:

1. La anotación `@Database` que nos permitirá indicar el nombre de la BD, la versión y las entidades que contiene.
2. La anoación `@TypeConverters` que nos permitirá indicar los conversores de tipos que usaremos.
3. El método de clase `getDatabase()` que nos devolverá una instancia de la BD.
4. El método `contactoDao()` que nos devolverá el DAO sobre la entidad `ContactoEntity`.

```
@Database(
    entities = [ContactoEntity::class],
    exportSchema = false,
    version = 1
)
@TypeConverters(RoomConverters::class)
abstract class AgendaDb : RoomDatabase() {
    abstract fun contactoDao(): ContactoDao

    companion object {
        @Volatile
        private var db: AgendaDb? = null

        fun getDatabase(context: Context) = Room.databaseBuilder(
            context,
            AgendaDb::class.java, "agenda"
        )
        .allowMainThreadQueries()
        .fallbackToDestructiveMigration()
        .build()
    }
}
```

Paso 6: Preparar para inyectar con Hilt los módulos

En el fichero `AppModule.kt` dentro del paquete `.di` definiremos como inyectar la BD `provideAgendaDatabase` y el DAO `provideContactoDao` en nuestra aplicación.

> 📌 **Nota**: Fíjate que la anotación `@ApplicationContext` inyecta el contexto de la aplicación. De esta manera podrá crear el fichero de la BD en el sistema de ficheros accesible por el contexto de la misma.

```
@Module
@InstallIn(SingletonComponent::class)
class AppModule {

    @Provides
    @Singleton
    fun provideAgendaDatabase(
        @ApplicationContext context: Context
    ) : AgendaDb = AgendaDb.getDatabase(context)

    @Provides
    @Singleton
    fun provideContactoDao(
        db: AgendaDb
    ) : ContactoDao = db.contactoDao()

    // En el proveedor del repositorio sustituimos DaoMock por el Dao
    @Provides
    @Singleton
    fun provideContactoRepository(
        contactoDao: ContactoDao
    ) : ContactoRepository = ContactoRepository(contactoDao)
}
```

Paso 7: Añadir los conversores de ContactoEntity a Contacto y viceversa

Necesitamos **mapear las entidades de la BD a nuestro modelo** y viceversa.
Lo normal es que sea inmediato pero no tiene que ser así. Por ejemplo, si te fijas las **categorias** dentro de nuestro `ContactoEntity` es de tipo `string` por lo cual la forma de guardarlas será seguramente el nombre de las mismas separador por comas ***"Amigos,Trabajo,Familia"*** , pero en la clase `Contacto` de nuestro modelo era un array de tipo enumerado `EnumSet<Categorias>`.

Es por esto que en `RepositoryConverters.kt` dentro del paquete `.data` definiremos los métodos que nos permitirán convertir de un tipo a otro como hicimos a al principio con las clases de Mock y el modelo.

```
fun EnumSet<Contacto.Categorias>.toCategoriaEntity() =
    joinToString(separator = ",") { it.name }

fun Contacto.toContactoEntity() = ContactoEntity(
    id = id,
    nombre = nombre,
    apellidos = apellidos,
    foto = foto,
    email = correo,
    telefono = telefono,
    categorias = categorias.toCategoriaEntity()
)

fun String.toEnumSetCategorias(): EnumSet<Contacto.Categorias> {
    val categorias = EnumSet.noneOf(Contacto.Categorias::class.java)
    val textos = this.split(",")
    textos.forEach { categoria ->
        if (!categoria.isNullOrEmpty())
            categorias.add(Contacto.Categorias.valueOf(categoria))
    }
    return categorias
}

fun ContactoEntity.toContacto() = Contacto(
    id = id,
    nombre = nombre,
    apellidos = apellidos,
    foto = foto,
    correo = email,
    telefono = telefono,
    categorias = categorias.toEnumSetCategorias()
)
```

Paso 8: Generando el nuevo repositorio

Vamos a reescribir el código del repositorio `ContactoRepository` dentro del paquete `.data` . En el, reescribiremos los métodos para usar `ContactoDao.kt` en lugar de `ContactosMock.kt` .

> 📌 **Nota**: En la gran mayoría de ejemplos de Internet este paso lo hacen combinando el **Facade Pattern** con el **Repository Pattern** manteniendo así ambos repositorios (Repository en el fondo es una concreción de Facade). Sin embargo, en nuestro ejemplo no lo vamos a hacer así, ya que supondría un mayor nivel de complejidad en la inyección de dependencias, teniendo que definir nuestras propias anotaciones para saber que concreción de la abstracción del repositorio vamos a inyectar en el **ViewModel**.

Básicamente la implementación será igual a la que teníamos con el DaoMock pero usando el Dao de Room....

```
class ContactoRepository @Inject constructor(
    private val dao: ContactoDao
) {
    suspend fun get(): List<Contacto> = withContext(Dispatchers.IO) {
        dao.get().map { it.toContacto() }.toList()
    }
    suspend fun get(id: Int): Contacto  = withContext(Dispatchers.IO) {
        val dato = dao.get(id)
        dato!!.toContacto()
    }
    suspend fun insert(contacto: Contacto)  = withContext(Dispatchers.IO) {
        dao.insert(contacto.toContactoEntity())
    }
    suspend fun update(contacto: Contacto) = withContext(Dispatchers.IO) {
        dao.update(contacto.toContactoEntity())
    }
    suspend fun delete(id: Int)  = withContext(Dispatchers.IO) {
        dao.delete(dao.get(id))
    }
}
```

Paso 10: Carga inicial de datos en la BD con los datos de Mock

Vamos a cargar los datos de Mock en la BD. Para ello, en el fichero `AgendaApplication.kt` dentro del paquete `com.pmdm.agenda` invalidaremos un método `onCreate()` del ciclo de vida de la aplicación y que se ejecutará al iniciar la misma. En él, comprobaremos si la BD está vacía y en caso afirmativo cargaremos los datos de Mock en la BD.

```
@HiltAndroidApp
class AgendaApplication : Application()  {
    @Inject
    lateinit var daoMock: ContactoDaoMock
    @Inject
    lateinit var daoEntity: ContactoDao

    override fun onCreate() {
        super.onCreate()

        runBlocking {
            if (daoEntity.count() == 0)
                daoMock.get().forEach { daoEntity.insert(it.toContacto().toContactoEntity())
        }
    }
}
```

En la primera ejecución creará la BD si no existe y cargará todos los datos. Recuerda que la DB se encuentra en la ruta de nuestro dispositivo `/data/data/com.pmdm.agenda/databases` y por tanto si borramos su contenido o la vaciamos del todo se volverá a cargar.