Universidad Nacional de La Matanza

Persistencia II

Bases de Datos

Agenda

- Introducción
- Estado del Arte
- SQLDelight
  - Integración
  - Creación de DB
  - Drivers
  - Operaciones
  - Transacciones
  - Plugin de Android Studio

Introducción

- Cuando necesitamos almacenar datos más complejos o en cantidad, el almacenamiento clave-valor no nos sirve
- Necesitamos hacer uso de Bases de Datos

Estado del arte

Estado del arte

- Los teléfonos móviles tienen instalado un motor de bases de datos llamado SQLite.
  - https://www.sqlite.org/index.html

Estado del arte

- Sqlite.lib es una librería **nativa**.
- Tiene limitaciones.
- Escrita en C.
- Distintas plataformas proveen distintas interfaces para acceder a la base de datos.

Estado del arte

¿Qué opciones tenemos para implementar bases de datos?

SQLite

Room

Realm (no relacional)

Estado del arte

¿Qué opciones tenemos para implementar bases de datos?

SQLite

CoreData

Realm (no relacional)

Estado del arte

android SQLite

Precaución: Si bien estas API son potentes, se caracterizan por ser bastante específicas y su uso requiere de mucho tiempo y esfuerzo.

- No hay verificación en tiempo de compilación de las consultas de SQL sin procesar. A medida que cambia tu grafo de datos, debes actualizar manualmente las consultas de SQL afectadas. Este proceso puede llevar mucho tiempo y causar errores.
- Debes usar mucho código estándar para convertir entre consultas de SQL y objetos de datos.

Por estos motivos, **recomendamos enfáticamente** usar la Biblioteca de persistencias Room como una capa de abstracción para acceder a la información de las bases de datos SQLite de tu app.

https://developer.android.com/training/data-storage/sqlite

Estado del arte

Room

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)

@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    // otras consultas
}
```

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}


val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "database-name"
).build()

val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```

SQLDelight

SQLDelight

- Es una librería/biblioteca para Kotlin Multiplatform.
- Genera una API Kotlin que es *type-safe* a partir de las consultas a la BBDD
- Soporte de SQLite para
  - Android
  - Plataformas “nativas” (iOS, macOS, Windows)
  - Javascript
- También soporta usar MySQL, PostgreSQL y HSQL/H2 como motor.

SQLDelight: otros features

- Verificación del *schema*.
- Verifica los *statements*.
- Verifica *migrations*.
- … en tiempo de **compilación!**

Integración

SQLDelight Plugin

- Además de la librería, nos va a hacer falta usar un plugin para el proyecto.
- Este plugin es el encargado de generar una API para que podamos crear una base de datos en nuestros módulos por medio de sus archivos *build.gradle*
- Además, generará el código Kotlin necesario para ejecutar nuestras queries.

build.gradle del proyecto

```
plugins {
    ...

    id("com.squareup.sqldelight").version("1.5.5").apply(false)

}
```

build.gradle del módulo compartido

```
plugins {
    ...

    id("com.squareup.sqldelight")

}
```

Dependencias por módulo

- Dependencia para **commonMain**
  `implementation("com.squareup.sqldelight:runtime:1.5.5")`
- Dependencia para **androidMain**
  `implementation("com.squareup.sqldelight:android-driver:1.5.5")`
- Dependencia para **iosMain**
  `implementation("com.squareup.sqldelight:native-driver:1.5.5")`

Creación de la DB

Configuración

- Una vez que se tiene el plugin y las librerías importadas, se puede configurar la base de datos.
- El plugin nos permite hacer uso del bloque `sqldelight` en el script de build.

```
sqldelight {
    database("AppDatabase") {
        packageName = "com.example.db"
    }
}
```

https://cashapp.github.io/sqldelight/multiplatform_sqlite/gradle/

Configuración

- **IMPORTANTE:** Hay que crear a mano la estructura de directorio para los archivos **.sq**, respetando el **packageName** que se utilizó para crear la DB. De otra forma, el plugin de Gradle no va a crear ***AppDatabase.Schema*** y vamos a empezar a recibir errores de compilación del proyecto.

Para nuestro ejemplo habría que crear:

*shared -> commonMain -> sqldelight -> com -> example -> db -> **Archivo**.sq*

- Luego, buildear el proyecto y ¡listo!
  `./gradlew build`, o desde el IDE.

Drivers

Drivers

- Para que SQLDelight sepa cómo conectarse al SQLite específico de cada plataforma, deben ser estas quienes provean una implementación correcta del *Driver* a utilizar.
- Otra vez la relevancia del patrón ***expect/actual***.

commonMain

```
expect class DatabaseDriverFactory {

    fun createDriver(): SqlDriver

}
```

androidMain

```
actual class DatabaseDriverFactory(private val context: Context) {

    actual fun createDriver(): SqlDriver {

        return AndroidSqliteDriver(AppDatabase.Schema, context,
"test.db")

    }

}
```

iosMain

```
actual class DatabaseDriverFactory {
    actual fun createDriver(): SqlDriver {
        return NativeSqliteDriver(AppDatabase.Schema, "test.db")
    }
}
```

¡Listo!

- Ahora alcanza con crear una instancia de la base de datos para ejecutar queries.

```
val database = AppDatabase(databaseDriverFactory.createDriver())
```

Operaciones

Operaciones

- Para poder ejecutar consultas/*statements* SQLite en nuestra aplicación hay que crear archivos con la extensión **.sq**.
- En esos archivos escribiremos el código SQL que queremos ejecutar.
- El plugin de SQLDelight hace la magia necesaria ✨
  - ¡Genera código Kotlin!
  - No es necesario generar *data classes* ni mapear nuestras entidades a la DB como en otros frameworks/ORM.

Crear tablas

- Es lo primero que tendría que haber en un archivo **.sq.**
- Se termina generando una data class en Kotlin con el nombre de la tabla, y properties con el tipo y nombre de las columnas.

```
CREATE TABLE Language (

    id INTEGER NOT NULL PRIMARY KEY,

    name TEXT NOT NULL

);
```

Borrar registros

- Incluir en el archivo **.sq**

```
deleteAllLanguages:
DELETE FROM Language;
```

- Esto genera una función deleteAllLanguages() que se usa de la siguiente manera:

```
val database = AppDatabase(sqlDriver)
val appDatabaseQueries: AppDatabaseQueries = database.appDatabaseQueries
appDatabaseQueries.deleteAllLanguages()
```

Insertar y actualizar registros

- Misma historia que con el **DELETE.**

```
insertLanguage:
INSERT INTO Language(id, name)
VALUES(?, ?);

updateLanguageName:
UPDATE Language
SET name = ?
WHERE id = ?;
```

Obteniendo datos de la DB

- Se utiliza la cláusula **SELECT**.

```
selectAllLanguages:
SELECT * FROM Language;
```

- En estos casos, SQLDelight genera dos funciones distintas, una para ejecutar la query y otra que nos va a permitir construir un objeto en base al resultado:

```
fun selectAllLanguages(): Query<Language>
fun <T : Any> selectAllLanguages(mapper: (id: Long, name: String) -> T):
Query<T>
```

Obteniendo datos de la DB

- Para poder ejecutar este código debemos hacer algo como lo siguiente:

```
data class SystemLanguage(val id: Long, val name: String)

val database = AppDatabase(sqlDriver)
val appDatabaseQueries: AppDatabaseQueries = database.appDatabaseQueries

val languages: List<SystemLanguage> = appDatabaseQueries.selectAllLanguages {
id: Long, name: String ->
        SystemLanguage(id, name)
    }.executeAsList()
```

Transacciones

Insertar y actualizar registros

- Junto con las queries se crea una función *transaction* que reciba una *lambda* en la que debemos ejecutar todas aquellas *queries* que queremos que se ejecuten en forma transaccional.

```
database.appDatabaseQueries.transaction {

    ...

}
```

Insertar y actualizar registros

```
database.appDatabaseQueries.transaction {

    appDatabaseQueries.insertLanguage(

        id = 1,
        name = “Español”
    )

}
```

Plugin de Android Studio

Plugin de Android Studio

Existe el SQLDelight plugin para Android Studio.

- Disponible desde el Marketplace del IDE.
- Le da formato al código de los archivos **.sq** mientras los editamos.
- Nos facilita escribir estas cláusulas gracias al *auto-complete*.
- Resalta errores en las *queries* en tiempo de compilación.

¿Preguntas?

Links

- Documentación oficial SQLDelight
- https://www.valueof.io/blog/installing-sqldelight-in-kotlin-multiplatform-project
- Charla KotlinConf 2018
- https://github.com/facundomr/EjemploSQLDelight
- Más recursos en documentación oficial

Codelab: Networking & Database en KMM

Fin