102. Room Data y SQLite

Room es una capa de abstracción sobre SQLite que facilita el acceso y la manipulación de bases de datos en aplicaciones

102.1 Configuración

Añadimos plugin `id("com.google.devtools.ksp")`:

`xml`
`plugins {`
`id("com.android.application")`
`id("org.jetbrains.kotlin.android")`
`id("com.google.devtools.ksp") version "1.5.30-1.0.0"//version "1.8.21-1.0.11"`
`}` Y las dependencias

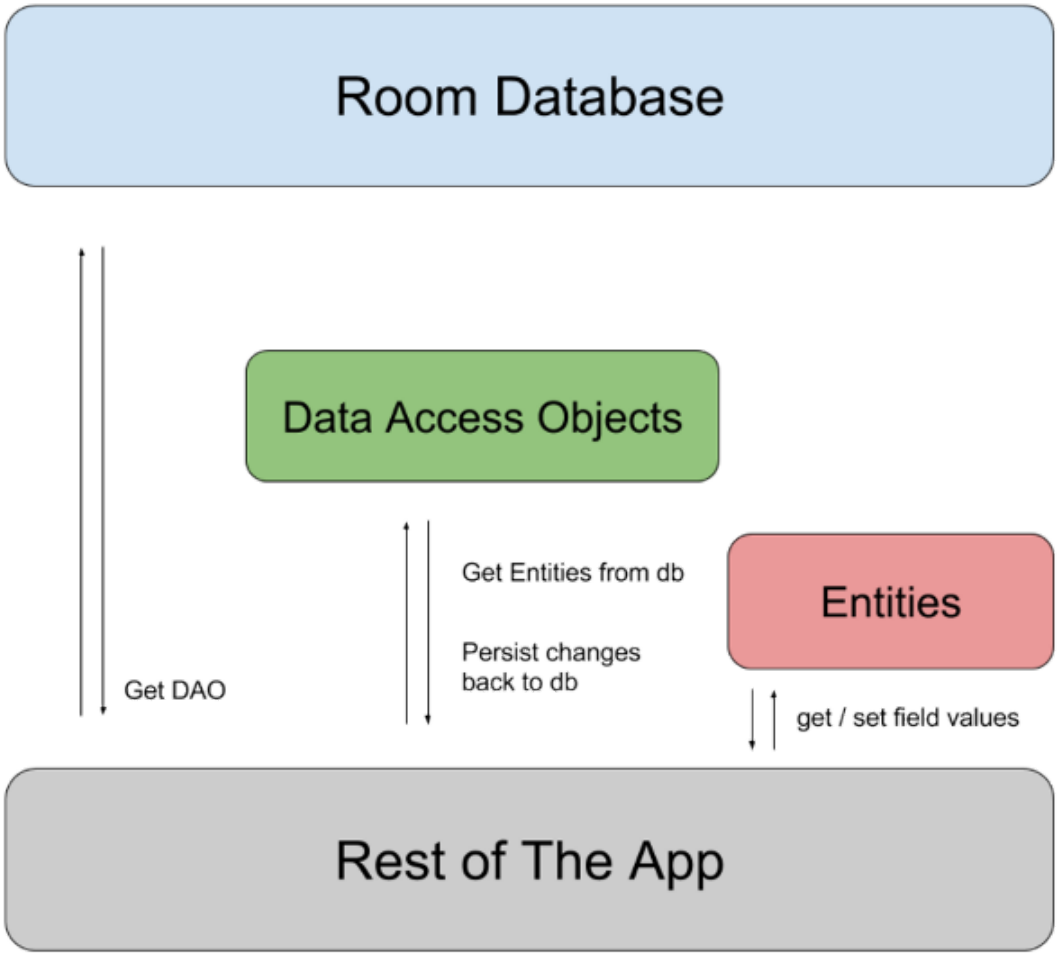
```
    // room
     implementation("androidx.room:room-runtime:2.6.0")
     implementation("androidx.room:room-ktx:2.6.0")
     annotationProcessor("androidx.room:room-compiler:2.6.0")
     ksp("androidx.room:room-compiler:2.6.0")
```

Además se suelen añadir las dependencias para corrutinas.

102.2 Introducción.

Tres compnentes principales

- La clase de la base de datos que contiene la base de datos y sirve como punto de acceso principal para la conexión subyacente a los datos persistentes de la app
- Las entidades de datos que representan tablas de la base de datos de tu app
- Los objetos de acceso a datos (DAOs) que proporcionan métodos que tu app puede usar para consultar, actualizar, insertar y borrar datos en la base de datos



102.2.1 Entidade

Ejemplo

Entidad User, filas en la tabla de bd:

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

102.2.2 Objeto de acceso DAO

Proporciona los métodos de acceso a la BD

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

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>
```

```
    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

102.2.3 Base de datos

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

La clase Base de datos debe cumplir:

- La clase debe tener una anotación `@Database` que incluya un **array entities** que enumere todas las entidades de datos asociados con la base de datos.
- Debe ser una clase **abstracta** que extienda `RoomDatabase`.
- Para cada clase DAO que se asoció con la base de datos, esta base de datos debe definir **un método abstracto que tenga cero argumentos y muestre una instancia de la clase DAO**.

Instanciamos una base de datos:

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

Si la aplicación se ejecuta en un proceso debe implementar el patrón `singleton`

Luego, puedes usar los métodos abstractos de AppDatabase para obtener una instancia del DAO. A su vez, puedes usar los métodos de la instancia del DAO para interactuar con la base de datos:

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

102.3 Entity, Base de datos y DAO en detalle

102.3.1 Base de datos

La clase de la base de datos debe cumplir con las siguientes condiciones:

- La clase debe tener una anotación `@Database` que incluya un array entities que enumere todas las entidades de datos asociados con la base de datos.
- Debe ser una clase abstracta que extienda RoomDatabase.

- Para cada clase DAO que se asoció con la base de datos, esta base de datos debe definir un método abstracto que tenga cero argumentos y muestre una instancia de la clase DAO.

Ejemplo:

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

Una vez creadas entidades y dao , se instancia AppDatabase

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

Y a continuación se usan los métodos abstractos de AppDatabase para obtener las instancias DAO y a su vez usar las intancias dao para operar con la base de datos:

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

102.3.2 Definir Entidades (Entity)

Definir entidades en Room, el ORM (Object-Relational Mapping) de Android, implica crear clases que representen las tablas de tu base de datos SQLite. Cada instancia de una entidad corresponde a una fila en la tabla. Aquí te explico con más detalle cómo definir entidades en Room utilizando Kotlin:

Sobre una Clase Kotlin donde tendremos **propiedades** (campos, atributos o variables de instancia) que anotamos

1. Anotaciones Básicas

`@Entity` : Esta anotación indica que la clase es una entidad en la base de datos. Puedes especificar detalles de la tabla, como el nombre de la tabla, índices, claves foráneas, etc.

`@PrimaryKey` : Cada entidad debe tener al menos un campo primario. Puedes configurarlo para que se auto-genere.

`@ColumnInfo` : Permite especificar el nombre de la columna en la base de datos. Si no se usa, el nombre del campo será usado como nombre de columna.

1. **Propiedades y Tipos de Datos**

Las propiedades de la clase (campos de la entidad) deben ser tipos de datos que Room pueda persistir. Estos incluyen tipos primitivos (int, long, boolean, etc.) y algunos tipos de objetos (String, ByteArrays). También puede almacenar datos personalizados a través de **typeConverter** como se describe más adelante.

```
@Entity
data class User(
```

```
    @PrimaryKey val id: Int,

    val firstName: String?,
    val lastName: String?
)
```

El nombre de la tabla será el mismo que el de la clase. Se modifica usando `@Entity(tableName = "usuarios")`

Clave Primaria

Cada entidad debe tener una clave primaria ver argumentos de @PriaryKey.
También pueden crearse claves primarias compuestas en la etiqueta `@Entity`:

```
@Entity(primaryKeys = ["firstName", "lastName"])
data class User(
    val firstName: String?,
    val lastName: String?
)
```

Ignorar campos

Se pueden ignorar campos de la clase que no se incluyen en las tablas:

```
@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    val lastName: String?,
    @Ignore val picture: Bitmap?
)
```

Si hay herencia por medio es más facil con la notación `@Entity`:

```
open class User {
    var picture: Bitmap? = null
}

@Entity(ignoredColumns = ["picture"])
data class RemoteUser(
    @PrimaryKey val id: Int,
    val hasVpn: Boolean
) : User()
```

1. **Relaciones**
   Room soporta relaciones como `@Embedded`, `@Relation`, `@ForeignKey`, etc., para representar relaciones entre tablas.
2. **Indices y Claves Foráneas**.
   Define índices en la tabla para mejorar el rendimiento de las consultas. **foreignKeys**: Define claves foráneas para mantener la integridad referencial entre tablas.

Ejemplo:

```
@Entity(
    tableName = "usuarios",
    foreignKeys = [
```

```
        ForeignKey(
            entity = Departamento::class,
            parentColumns = ["id"],
            childColumns = ["departamentoId"],
            onDelete = ForeignKey.CASCADE
        )
    ],
    indices = [Index(value = ["email"], unique = true)]
)
```

Type Converters

Si necesitas almacenar tipos de datos personalizados (como fechas o listas), debes definir un TypeConverter.

Un TypeConverter convierte un tipo de dato personalizado a un tipo que Room pueda entender y viceversa.

```
class Converters {
    @TypeConverter
    fun fromTimestamp(value: Long?): Date? {
        return value?.let { Date(it) }
    }

    @TypeConverter
    fun dateToTimestamp(date: Date?): Long? {
        return date?.time
    }
}
```

Registra tus TypeConverters en tu clase de base de datos RoomDatabase.

```
@Database(/* ... */)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    // ...
}
```

102.3.3 Clases DAO (Data Access Object)

El propósito principal es abstraer y encapsular todas las operaciones relacionadas con la base de datos, proporcionando una interfaz clara para interactuar con los datos.

Un DAO es una interfaz o clase abstracta en Kotlin que anota con `@Dao` .

102.3.3.1 Métodos de Acceso a Datos

Los métodos dentro de un DAO definen las operaciones de la base de datos, como inserciones, actualizaciones, eliminaciones y consultas.

Room permite definir estos métodos con anotaciones específicas:

`@Insert` : Para insertar registros.
`@Update` : Para actualizar registros.
`@Delete` : Para eliminar registros.
`@Query` : Para realizar cualquier consulta SQL

El método de insertar recibe un Entity definido con la anotación vista. Si el método `@Insert` recibe un solo parámetro, puede mostrar un valor `Long`, que es el nuevo `rowId` para el elemento insertado. Si el parámetro es un array o una colección, muestra un array o una colección de valores long en su lugar, donde cada valor debe ser el rowId de uno de los elementos insertados.

Room usa la clave primaria para hacer coincidir las instancias de entidades pasadas con las filas de la base de datos. Si no hay una fila con la misma clave primaria, Room no realiza cambios.

Con otras palabras cuando la clave primaria es un long autoincrementado, el rowId y la clave primaria coinciden cortesía de SQLite

De manera opcional, un método `@Update` y `@Delete` puede mostrar un valor int que indica la cantidad de filas que se actualizaron de forma correcta.

```
@Dao
interface UsuarioDao {
    @Insert
    suspend fun insertar(vararg usuario: Usuario)

    @Update
    suspend fun actualizar(vararg usuario: Usuario)

    @Delete
    suspend fun eliminar(vararg usuario: Usuario)

    @Query("SELECT * FROM usuarios")
    fun obtenerTodos(): Flow<List<Usuario>>

    @Query("SELECT * FROM usuarios WHERE id = :id")
    fun obtenerPorId(id: Int): Flow<Usuario>
}
```

102.3.3.1.1 BUSQUEDAS CON QUERY

Busqueda simple:

```
@Query("SELECT * FROM user")
fun loadAllUsers(): Array<User> // o List<User>
```

Busqueda con recuperación de un subconjunto de campos. Por motivos de optimización se recomienda recuperar solamente los campos que se usan en cada momemto.

Room permite mostrar un objeto simple de cualquiera de lA búsquedas, siempre *y cuando puedas asignar el conjunto de columnas de resultados al objeto que se muestra*. Por ejemplo, puedes definir el siguiente objeto para conservar el nombre y apellido de un usuario (que tendrá más campos que no necesitamos):

```
data class NameTuple(
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Y recuperamos con una query para esos dos campos:

```
@Query("SELECT first_name, last_name FROM user")
fun loadFullName(): List<NameTuple>
```

Para pasar parámetros a la busqueda. Las funciones DAO reciben argumentos que podemos usar en el Query utilizando dos puntos: (:minAge en el ejemplo)

```
@Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int): Array<User>

@Query("SELECT * FROM user WHERE first_name LIKE :search " +
       "OR last_name LIKE :search")
fun findUserWithName(search: String): List<User>
```

Como se ve en el ejemplo, se pueden usar varios parámetros y se pueden repetir en el Query.

Room entiende cuándo un parámetro representa una **colección** y lo expande automáticamente en el tiempo de ejecución en función de la cantidad de parámetros proporcionados.

```
@Query("SELECT * FROM user WHERE region IN (:regions)")
fun loadUsersFromRegions(regions: List<String>): List<User>
```

Para busquedas en varias tablas se puede usar `JOIN`

```
@Query(
    "SELECT * FROM book " +
    "INNER JOIN loan ON loan.book_id = book.id " +
    "INNER JOIN user ON user.id = loan.user_id " +
    "WHERE user.name LIKE :userName"
)
fun findBooksBorrowedByNameSync(userName: String): List<Book>
```

Tipos de datos especiales devueltos por ROOM

Room devuelve ciertos tipos de datos para usar con otras librerías.

Por ejemplo, con la libería de Paging:

```
@Dao
interface UserDao {
  @Query("SELECT * FROM users WHERE label LIKE :query")
  fun pagingSource(query: String): PagingSource<Int, User>
}
```

102.3.4 Relaciones entre objetos

Dos enfoques posibles: * Definir una clase de datos intermedia con objetos incorporados * un método de búsquedas relacionadas que muestre un tipo de datos que se devuelve de multimapa .

102.3.4.1 Clase de datos intermedia

Modelamos la relación con una clase intermedia que debe incluir las entidades relacionadas y los campos implicados.

Ejemplo si tenemos Users Y Books

```
@Dao
interface UserBookDao {
    @Query(
```

```
        "SELECT user.name AS userName, book.name AS bookName " +
        "FROM user, book " +
        "WHERE user.id = book.user_id"
    )
    fun loadUserAndBookNames(): Flow<List<UserBook>>
}

data class UserBook(val userName: String?, val bookName: String?)
```

Cómo crear objetos incorporados

Es posible que, a veces, quieras expresar una entidad o un objeto de datos **como un solo elemento** integral en la lógica de la base de datos, incluso si el objeto contiene varios campos. En esas situaciones, puedes usar la anotación @Embedded para representar un objeto cuyos subcampos quieras desglosar en una tabla. Luego, puedes buscar los campos integrados tal como lo harías con otras columnas individuales.

Por ejemplo, la clase User puede incluir un campo de tipo Address, que representa una composición de campos llamados street, city, state y postCode. Para almacenar las columnas compuestas por separado en la tabla, incluye un campo Address en la clase User con anotaciones @Embedded, como se muestra en el siguiente fragmento de código:

```
data class Address(
    val street: String?,
    val state: String?,
    val city: String?,
    @ColumnInfo(name = "post_code") val postCode: Int
)

@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    @Embedded val address: Address?
)
```

La tabla que representa un objeto User contiene columnas con los siguientes nombres: id, firstName, street, state, city y post_code.

102.3.4.1.1 RELACIÓN UNO A UNO

Ejemplo usuario y biblioteca

Primer paso: una de las entidades debe definir un variable que haga referencia a la clave primaria de la otra entidad.

```
@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Library(
    @PrimaryKey val libraryId: Long,
```

```
    val userOwnerId: Long
)
```

Segunda paso : modelamos la relación entre las entidades creando una data class que cumpla:

- incluye las dos entidades relacionadas de forma que cada nueva instancia de la clase-relación contenga una intancia de las clases relacionadas.
- Añadir @Relation
- y asigna a parentColum el nombre de la columna de clave primaria de la entidad principal y a entityColumn el nombre de la columna de la entidad secundaria que hace referencia a la clave primaria de la entidad principal.

```
data class UserAndLibrary(
    @Embedded val user: User,
    @Relation(
         parentColumn = "userId",
         entityColumn = "userOwnerId"
    )
    val library: Library
)
```

Con la notacion `@Embedded` los campos de User se incluyen en `UserAndLibrary`
Observar el uso de las comas en el código anterior

Tercera paso: Por último, agrega un método a la clase DAO que devuelve todas las instancias de la clase de datos que vincula la entidad principal con la secundaria. Este método requiere que Room ejecute dos búsquedas, así que agrega la anotación @Transaction al método para asegurarte de que toda la operación se realice automáticamente.

```
@Transaction
@Query("SELECT * FROM User")
fun getUsersAndLibraries(): List<UserAndLibrary>
```

102.3.4.1.2 RELACIÓN UNO A MUCHOS

En el ejemplo de la app de transmisión de música, supongamos que el usuario puede organizar las canciones en playlists. Cada usuario puede crear tantas playlists como desee, pero un solo usuario crea cada playlist. Por lo tanto, existe una relación de uno a varios entre la entidad User y la entidad Playlist.

Primer paso: crea una clase para las dos entidades. Al igual que en una relación de uno a uno, la entidad secundaria debe incluir una variable que haga referencia a la clave primaria de la entidad principal.

```
@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val userCreatorId: Long,
    val playlistName: String
)
```

En este ejemplo *userCreatorId*

Segundo paso : Modelamos la relación con una clase de datos intermedia en la que cada instancia tenga una instancia de la entidad principal y una lista de todas las instancias de entidades secundarias correspondientes. Agrega la anotación @Relation a la instancia de la entidad secundaria, y asigna a parentColumn el nombre de la columna de clave primaria de la entidad principal y a entityColumn el nombre de la columna de la entidad secundaria que hace referencia a la clave primaria de la entidad principal.

```
data class UserWithPlaylists(
    @Embedded val user: User,
    @Relation(
          parentColumn = "userId",
          entityColumn = "userCreatorId"
    )
    val playlists: List<Playlist>
)
```

Tercer paso agregar un nuevo metdo en la clase DAO que devuelva la lista de objetos

```
@Transaction
@Query("SELECT * FROM User")
fun getUsersWithPlaylists(): List<UserWithPlaylists>
```

102.3.4.1.3 RELACIÓN MUCHOS A MUCHOS

En el ejemplo de la app de transmisión de música, imagina las canciones en las playlists definidas por el usuario. Cada playlist puede incluir muchas canciones y cada canción puede ser parte de muchas playlists diferentes. Por lo tanto, existe una relación de varios a varios entre la entidad Playlist y la entidad Song.

Primer paso: crea una clase para cada una de las dos entidades. Las relaciones de varios a varios son diferentes de otros tipos de relación porque, por lo general, no hay una referencia a la entidad principal en la entidad secundaria. Entonces, **crea una tercera clase para representar una entidad asociativa (o tabla de referencias cruzadas)** entre las dos entidades.

La tabla de referencias cruzadas debe tener columnas para la clave primaria de cada entidad contemplada en la relación de varios a varios que se representa en la tabla.

En este ejemplo, cada fila de la tabla de referencias cruzadas corresponde a una vinculación de una instancia Playlist y una instancia Song donde la canción a la que se hace referencia se incluye en la playlist a la que se hace referencia.

```
@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val playlistName: String
)

@Entity
data class Song(
    @PrimaryKey val songId: Long,
    val songName: String,
    val artist: String
)
```

```
@Entity(primaryKeys = ["playlistId", "songId"])
data class PlaylistSongCrossRef(
    val playlistId: Long,
    val songId: Long
)
```

Segundo Paso El siguiente paso depende de cómo quieras consultar las entidades relacionadas.

Si quieres consultar playlists y un listado de las canciones correspondientes por cada playlist, crea una clase de datos nueva con un objeto Playlist único y un listado de todos los objetos Song que incluye la playlist. Si quieres consultar canciones y un listado de las playlists correspondientes por cada canción, crea una clase de datos nueva con un objeto Song único y un listado de los objetos Playlist en los que se incluye la canción.
En ambos casos, modela la relación entre las entidades mediante la propiedad associateBy en la anotación @Relation de cada una de las clases para identificar la entidad de la referencia cruzada que proporciona la relación entre las entidades Playlist y Song.

```
data class PlaylistWithSongs(
    @Embedded val playlist: Playlist,
    @Relation(
         parentColumn = "playlistId",
         entityColumn = "songId",
         associateBy = Junction(PlaylistSongCrossRef::class)
    )
    val songs: List<Song>
)

data class SongWithPlaylists(
    @Embedded val song: Song,
    @Relation(
         parentColumn = "songId",
         entityColumn = "playlistId",
         associateBy = Junction(PlaylistSongCrossRef::class)
    )
    val playlists: List<Playlist>
)
```

Tercer Paso: agrega un método a la clase DAO para exponer la funcionalidad de búsqueda que necesita la app.

getPlaylistsWithSongs: Este método busca la base de datos y devuelve todos los objetos PlaylistWithSongs resultantes. getSongsWithPlaylists: Este método busca la base de datos y devuelve todos los objetos SongWithPlaylists resultantes. Cada uno de los métodos requiere que Room ejecute dos búsquedas, así que agrega la anotación @Transaction a ambos para asegurarte de que toda la operación se realice automáticamente.

```
@Transaction
@Query("SELECT * FROM Playlist")
fun getPlaylistsWithSongs(): List<PlaylistWithSongs>

@Transaction
@Query("SELECT * FROM Song")
fun getSongsWithPlaylists(): List<SongWithPlaylists>
```

102.3.5 Uso de Corrutinas y Flow (Busquedas DAO asíncronas)

En Kotlin, es común usar **corrutinas** (suspend functions) para operaciones de base de datos que pueden bloquear el hilo principal. **Flow** se utiliza para consultas que observan cambios en la base de datos, proporcionando un flujo de datos reactivos (versión Room 2.2 y posteriores)

Busquedas únicas Se ejecutan una vez y usan una instantanea de los datos ```kotlin @Dao interface UserDao { @Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun insertUsers(vararg users: User)

```
@Update
suspend fun updateUsers(vararg users: User)

@Delete
suspend fun deleteUsers(vararg users: User)

@Query("SELECT * FROM user WHERE id = :id")
suspend fun loadUserById(id: Int): User

@Query("SELECT * from user WHERE region IN (:regions)")
suspend fun loadUsersByRegion(regions: List<String>): List<User>
```

}```

102.3.5.1 Busquedas observables

Cuando hay un cambio en los valoes de las tablas se emite los nuevos valores.

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user WHERE id = :id")
    fun loadUserById(id: Int): Flow<User>

    @Query("SELECT * from user WHERE region IN (:regions)")
    fun loadUsersByRegion(regions: List<String>): Flow<List<User>>
}
```

```
## Ejemplo


1. Definir entidades(tablas de BD) usando la notación **@Entity**
```kotlin

@Entity
data class Usuario(
    @PrimaryKey(autoGenerate = true) val id: Int,
    @ColumnInfo(name = "nombre") val nombre: String,
    // Otros campos...
)
```

1. DAO (Data Access Object): Crea una interfaz DAO para definir las operaciones de la base de datos

```
@Dao
interface UsuarioDao {
    @Query("SELECT * FROM usuario")
    fun obtenerTodos(): Flow<List<Usuario>>

    @Insert
    suspend fun insertar(usuario: Usuario)
```
```

```
    // Otros métodos como update, delete...
}
```

1. Base de Datos: Define una clase abstracta que extienda RoomDatabase e incluye tus entidades y DAOs:

```
@Database(entities = [Usuario::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun usuarioDao(): UsuarioDao
    // Otros DAOs...
}
```

1. Instancia de la Base de Datos: Crea una instancia de tu base de datos, generalmente en tu clase Application o de manera similar:

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

1. Uso en Compose. En tus composables, puedes usar ViewModels para manejar la lógica de negocio y acceder a la base de datos. Por ejemplo:

```
@Composable
fun MiComposable() {
    val viewModel: MiViewModel = viewModel()
    // Usa el viewModel para obtener datos y mostrarlos en la UI
}
```

1. ViewModel: En tu ViewModel, utiliza coroutines para realizar operaciones de base de datos de manera asíncrona:

```
class MiViewModel(private val db: AppDatabase) : ViewModel() {

    // Opción 1: Usar Flow directamente
    val usuarios: Flow<List<Usuario>> = db.usuarioDao().obtenerTodos()

    // Opción 2: Convertir Flow a LiveData
    val usuariosLiveData: LiveData<List<Usuario>> =
db.usuarioDao().obtenerTodos().asLiveData()
}
```

1. Uso en Composable

```
@Composable
fun MiComposable(viewModel: MiViewModel = viewModel()) {
    val usuarios by viewModel.usuarios.collectAsState(initial = listOf())

    // Ahora puedes usar 'usuarios' para mostrar los datos en la UI
}
```

Apendice

Enlaces:
* codelab Room y Flow * tutorial usando Vistas

* Relaciones en Room * Código ejemplo (cuidado tiene 4 años de antigüedad)

Versión 0.5, 12-12-23 Versión 0.8, 8-1-24

¿Fue útil esta página?