Tabla de contenido

Creación y gestión básica de ROOM

1. Introducción

Android recomienda el uso de la Biblioteca de persistencias Room, como una capa de abstracción para acceder a la información de las bases de datos **`SQLite`** de tu app, por los siguientes motivos:

- **No existe verificación en tiempo de compilación** de las consultas **`SQL`** sin procesar. Aunque las API del paquete **`android.database.sqlite`** son potentes, requieren mucho tiempo y esfuerzo.
- **Es necesario mucho código estándar** para convertir entre consultas **`SQL`** y objetos de datos.

Room ofrece los siguientes beneficios:

- Verificación del tiempo de compilación de las consultas en **`SQL`**.
- Anotaciones de conveniencia que minimizan el código estándar repetitivo y propenso a errores.
- Rutas de migración de bases de datos optimizadas.

2. Componentes principales

Estos son los tres componentes principales de **`Room`**:

- 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

La clase de base de datos proporciona a tu app instancias de los **`DAOs`** asociados con esa base de datos. A su vez, la app puede usar los **`DAOs`** para recuperar datos de la base de datos como instancias de objetos de entidad de datos asociados. La app también puede usar las entidades de datos definidas para actualizar filas de las tablas correspondientes o crear filas nuevas para su inserción. En la figura 1, se muestran las relaciones entre los diferentes componentes de **`Room`**.

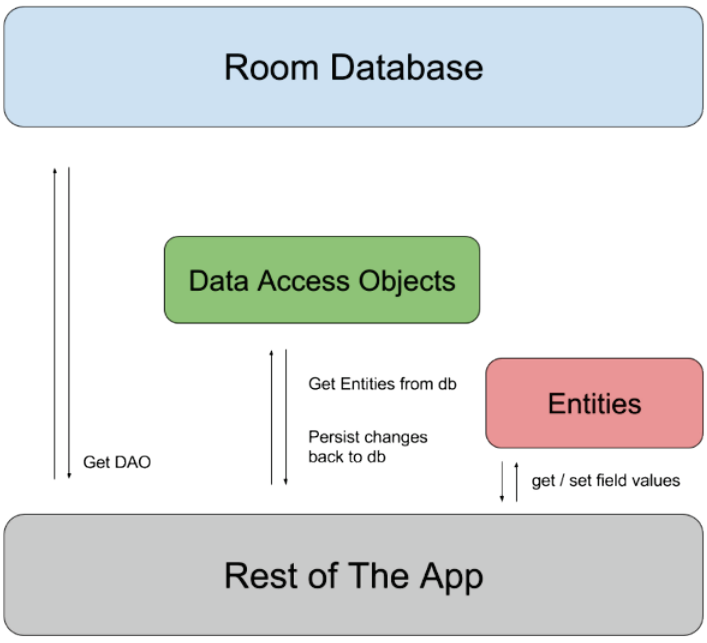


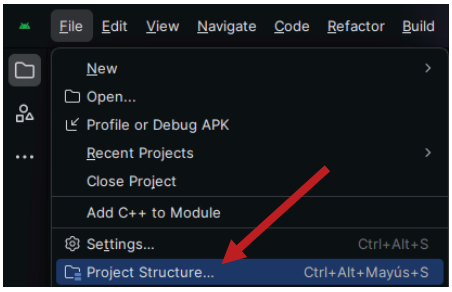
Figura 1. Diagrama de la arquitectura de la biblioteca de Room

3. Dependencias para Room

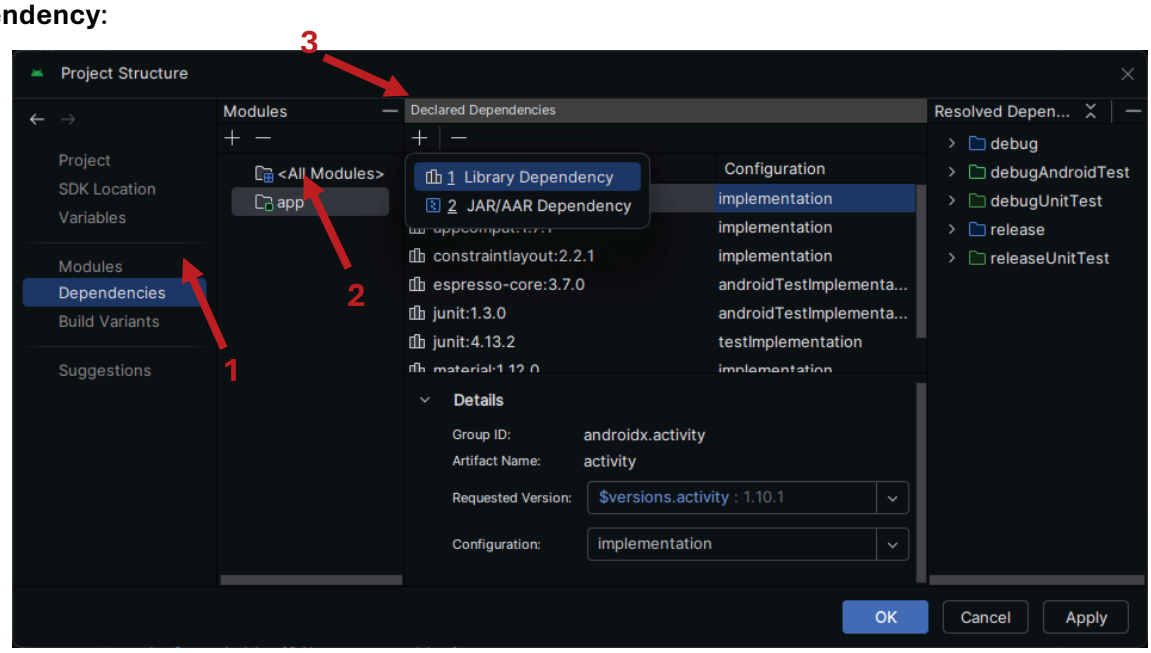
El primer paso consiste en añadir las librerías necesarias, y para un proyecto **Java** son:

- **`room-runtime`**: como `implementation`
- **`room-compiler`**: como `annotationProcessor`

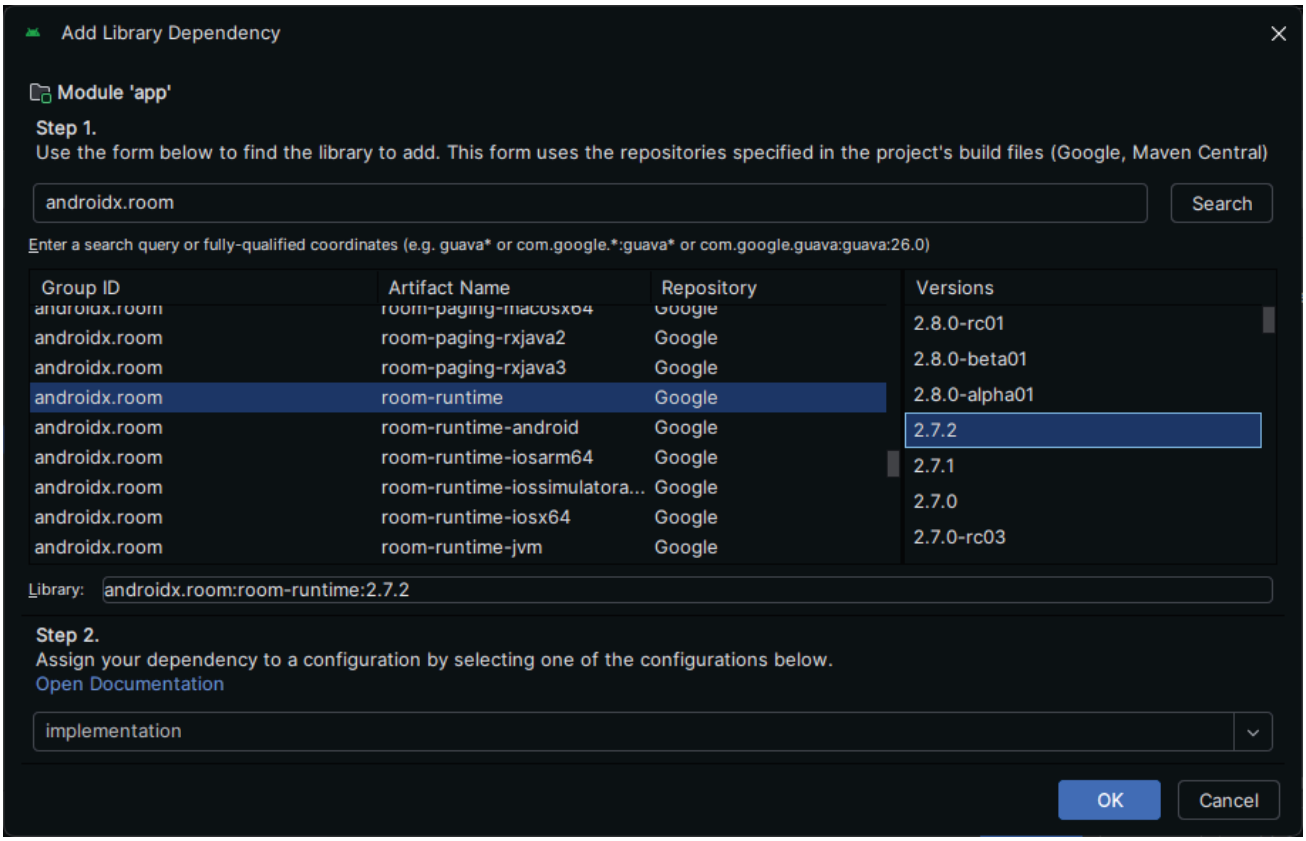
Para añadir las dependencias, podemos hacerlo a través del **IDE**. Vamos al menú **File → Project Structure...**



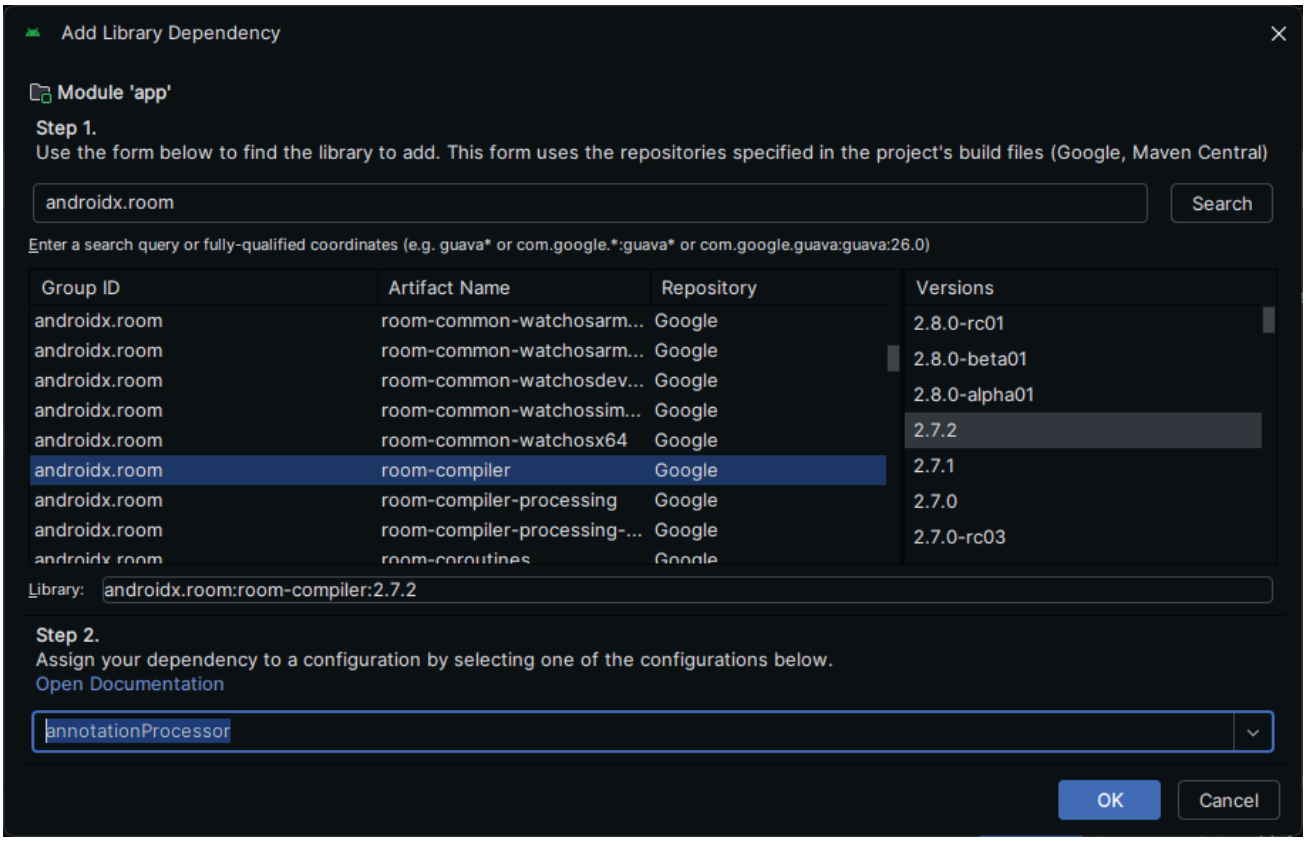
En la ventana, en la primera columna elegimos **Dependencies**, en la siguiente **app**. Por cada dependencia que necesitemos añadir será necesario hacer clic al signo **+** y seleccionar la opción **Library Dependency**:



En el **Step1** introducimos el nombre de la librería: **`androidx.room`**, y hacemos clic en **Search**. Seleccionamos el artefacto **`room-runtime`** y la versión. Para aceptar los cambios hacemos clic en **OK**.



Repetimos la acción anterior para la dependencia **`room-compiler`**. En este caso, debemos modificar el valor del **Step2**. Por defecto, Android Studio añade las librerías como **`implementation`** pero para este artefacto la configuración es **`annotationProcessor`**, y que especificamos en el campo del **Step2:**



La forma más inmediata de añadir las librerías es editar el archivo **`build.gradle`** a nivel de módulo con las siguientes declaraciones:

```
// Room para Java
val version_room = "2.7.2"
implementation("androidx.room:room-runtime:$version_room")
annotationProcessor("androidx.room:room-compiler:$version_room")
```

Si estamos usando un proyecto **`Maven`**, las dependencias serían:

```
<!-- https://mvnrepository.com/artifact/androidx.room/room-runtime -->
<dependency>
    <groupId>androidx.room</groupId>
    <artifactId>room-runtime</artifactId>
    <version>2.7.2</version>
    <scope>runtime</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/androidx.room/room-compiler -->
<dependency>
    <groupId>androidx.room</groupId>
    <artifactId>room-compiler</artifactId>
    <version>2.7.2</version>
</dependency>
```

4. Implementación

En esta sección, se presenta un ejemplo de implementación de una base de datos de **Room** con una sola entidad de datos y un **DAO** único.

4.1. Entidad de datos

Una clase java **Entidad** corresponde a una tabla en la base de datos de **Room** asociada y **cada instancia** de una entidad representa **una fila de datos** en la tabla correspondiente.

Eso significa que puedes usar las entidades de Room para definir tu esquema de base de datos sin escribir ningún código de **SQL**.

4.1.1. Anotación de clase

La anotación `@Entity` se declara antes de la firma de la clase, y de este modo, **Room** crea una tabla entidad de la base de datos.

```
@Entity(tableName = Constantes.TABLA_NOTAS)
public class Nota {    }
```

Por defecto, **Room** usa el *nombre de la clase* como el nombre de la tabla de la base de datos. Si quieres que la tabla tenga un nombre diferente, configura la propiedad `tableName` de la anotación `@Entity`.

Los **atributos** de esta etiqueta son:

- ✓ `tableName`: un `String` que define un nombre para la tabla.
- ✓ `índices`: un array de anotaciones `@Index` que indican las columnas a indizar. Ejemplo:

```
@Entity(indices = { @Index("nombre"),
                    @Index(value = {"apellidos", "direccion"})})
```

- ✓ `inheritSuperIndices`: un valor `boolean`. Si es `true`, los índices de las clases padre deben ser heredados automáticamente por esta Entidad, `false` en caso contrario. Por defecto es `false`.
- ✓ `primaryKeys`: un array de `String` con los nombres de las columnas que conforman la clave primaria compuesta. Ejemplo:

```
@Entity(primaryKeys = {"nombre", "apellidos"})
```

- ✓ `foreignKey`: un array de anotaciones `@ForeignKey` con los nombres de las restricciones **ForeignKey** de esta entidad. Ejemplo:

```
@Entity(foreignKeys = @ForeignKey(
        entity = Usuario.class,
        parentColumns = "id",
        childColumns = "id_usuario",
        onDelete = ForeignKey.CASCADE
))
```

- ✓ `ignoredColumns`: La lista de nombres de columna que deben ser ignorados por **Room**.

4.1.2. Anotación de atributos

Los **atributos de la clase** se corresponden con las *columnas en la tabla*, y se pueden emplear las siguientes anotaciones:

- `@ColumnInfo`

  Permite una personalización específica sobre la columna asociada a esta propiedad.

  Los atributos de configuración son:

  - ✓ `name`: un `String` que define un nombre para la columna.
  - ✓ `defaultValue`: un `String` que define el valor por defecto para esta columna.
  - ✓ `index`: un valor `boolean`. Si es `true` esta propiedad debe ser indexada, `false` en caso contrario. Por defecto es `false`.
  - ✓ `typeAffinity`: La equivalencia de tipo para la columna, que se utilizará para construir la base de datos.

    Los valores posibles son:

    ```
    ColumnInfo.{UNDEFINED|TEXT|INTEGER|REAL|BLOB}
    ```

  - ✓ `collate`: estable el modo de comparación de valores. Los valores posibles son:

    ```
    ColumnInfo.{UNSPECIFIED|BINARY|NOCASE|RTRIM|LOCALIZED|UNICODE}
    ```

    En el siguiente ejemplo, definimos la opción `NOCASE`:

    ```
    @ColumnInfo(name = "nombre", collate = ColumnInfo.NOCASE)
    public String nombre;
    ```

    En este caso, si haces la siguiente consulta:

    ```
    SELECT * FROM Usuario WHERE nombre = 'juan';
    ```

    SQLite encontrará tanto "Juan" como "JUAN" como "juan", porque la **`collation`** `NOCASE` hace la comparación **case-insensitive**.

- `@Ignore`

  Ignora el elemento marcado de la lógica de procesamiento de **`Room`**.

- `@PrimaryKey(autoGenerate = false [default])`

  Establece si la clave primaria debe ser autogenerada por **SQLite**. Por defecto, el valor es `false`. **`Room`** requiere que una clave primaria autogenerada sea de tipo `int` o `long`.

  Si necesitamos cambiar el nombre al campo debemos usarlo en combinación con `@ColumnInfo`:

  ```
  @PrimaryKey(autoGenerate = false)
  @ColumnInfo(name = "rowid")
  ```

4.1.3. Clave primaria compuesta

Si necesitas que las instancias de una entidad se identifiquen de forma única mediante una combinación de varias columnas, puedes definir una clave primaria compuesta si enumeras esas columnas en la propiedad **`primaryKeys`** de `@Entity`:

```
@Entity(primaryKeys = {"nombre", "apellidos"})
public class Usuario {
    public String nombre;
    public String apellidos;
}
```

4.1.4. Columnas indexadas

Para agregar índices a una entidad, incluye la propiedad `indices` dentro de la anotación `@Entity` y enumera los nombres de las columnas que quieras incluir en el índice o en el índice compuesto. En el siguiente fragmento de código, se muestra este proceso de anotación:

```
@Entity(indices = { @Index("nombre"),
                    @Index(value = {"apellidos", "direccion"})})
public class Usuario {
    @PrimaryKey
    public int id;

    public String nombre;
    public String direccion;

    @ColumnInfo(name = "apellidos")
    public String apellido;

    @Ignore
    Bitmap imagen;
}
```

A veces, ciertos campos o grupos de campos de una base de datos deben ser únicos. Puedes aplicar esta propiedad de exclusividad con la propiedad `unique` de una anotación `@Index` como `true`.

En la siguiente muestra de código, se evita que una tabla tenga dos filas con el mismo conjunto de valores para las columnas `nombre` y `apellido`:

```
@Entity(indices = { @Index(value = {"nombre", "apellidos"},
                    unique = true)
                  })
public class Usuario {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "nombre")
    public String nombre;

    @ColumnInfo(name = "apellidos")
    public String apellido;

    @Ignore
    Bitmap imagen;
}
```

4.2. Interfaz DAO

Las clases **DAO** son las responsables de definir los métodos de acceso a la base de datos. En **SQLite** empleamos los objetos **Cursor**. Con la **API Room**, no necesitamos todo el código relacionado con **Cursor**, y simplemente definimos nuestras consultas usando anotaciones en la clase **DAO**.

Puedes definir cada **DAO** como una interfaz o una clase abstracta. Por lo general, debes usar una interfaz para casos de uso básicos. En cualquier caso, siempre debes anotar tus **DAOs** con `@Dao`. Los **DAOs** no tienen propiedades, pero definen uno o más métodos para interactuar con los datos de la base de datos de tu app. Para más información: Cómo acceder a los datos con DAO de Room.

4.2.1. Insert

La anotación `@Insert` te permite definir métodos que insertan sus parámetros en la tabla adecuada en la base de datos. Cada parámetro del método `@Insert` debe ser una instancia de una clase de entidad de datos de Room anotada con `@Entity` o una colección de instancias de clase de entidad de datos:

```
@Dao
public interface UsuarioDao {

    // 1. INSERCIÓN INDIVIDUAL
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    public long insertarUsuario(Usuario usuario);

    // 2. INSERCIÓN MÚLTIPLE con varargs
    @Insert
    public long[] insertarUsuarios(Usuario ... usuarios);

    // 3. INSERCIÓN MÚLTIPLE args explicitos
    @Insert
    public long[] insertarAmbos(Usuario usuario1, Usuario usuario2);

    // 4. INSERCIÓN MÚLTIPLE con List
    @Insert
    public long[] insertarUsuarios(List<Usuario> usuarios);

    // 5. MÚLTIPLES PARÁMETROS DE DIFERENTES TIPOS
    @Insert
    public long[] insertarUsuariosConAmigos(Usuario usuario, List<Usuario> amigos);
}
```

Si el método `@Insert` recibe un único parámetro, puede devolver un valor `long` que corresponde al **rowId** del elemento insertado. Si se proporciona un array o una colección, en cambio, devuelve un array o colección de valores `long`, donde cada uno representa el **rowId** para cada uno de los elementos insertados.

4.2.2. Delete

La anotación `@Delete` te permite definir métodos que borran filas específicas de una tabla de base de datos. Al igual que los métodos `@Insert`, los métodos `@Delete` aceptan instancias de entidades de datos como parámetros.

```
@Dao
public interface UsuarioDao {

    @Delete
    public int borrarUsuarios(List<Usuario> usuarios);
}
```

4.2.3. Update

La anotación `@Update` te permite definir métodos que actualizan filas específicas en una tabla de base de datos. Al igual que los métodos `@Insert`, los métodos `@Update` aceptan instancias de entidades de datos como parámetros. El siguiente código muestra un ejemplo de un método `@Update` que intenta actualizar uno o más objetos `User` en la base de datos:

```
@Dao
public interface UsuarioDao {

    @Update
    public int actualizarUsuarios(List<Usuario> usuarios);
}
```

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. Un método `@Update` puede retornar un valor `int` que indica la cantidad de filas que se actualizaron de forma correcta.

4.2.4. Query

La anotación `@Query` te permite escribir instrucciones de **SQL** y exponerlas como métodos **DAO**. Usa estos métodos de búsqueda para consultar datos desde la base de datos de tu app o cuando necesites realizar inserciones, actualizaciones y eliminaciones más complejas. **Room valida las consultas en SQL en el tiempo de compilación**. Esto significa que, si hay un problema con tu búsqueda, se produce un error de compilación en lugar de una falla del tiempo de ejecución.

Búsquedas simples

El siguiente código define un método que usa una búsqueda `SELECT` simple para mostrar todos los objetos `Usuario` de la base de datos:

```
@Query("SELECT * FROM usuario")
public Usuario[] getUsuarios();
```

Búsquedas complejas

Algunas de tus búsquedas pueden requerir acceso a varias tablas para calcular el resultado. Puedes usar cláusulas `JOIN` en tus consultas en **SQL** para hacer referencia a más de una tabla.

El siguiente método une tres tablas para mostrar los libros prestados a un usuario:

```
@Query("SELECT * FROM libro " +
        "INNER JOIN prestamo ON prestamo.id_libro = id_libro " +
        "INNER JOIN usuario ON id_usuario = prestamo.id_usuario " +
        "WHERE usuario.nombre LIKE :nombreUsuario")
public List<Libro> buscarLibrosPrestadosPorNombre(String nombreUsuario);
```

Búsquedas parametrizadas

Room admite el uso de parámetros de métodos como parámetros de vinculación en tus búsquedas, usando la notación `:parámetro` en la definición de la consulta.

```
@Query("SELECT * FROM usuario WHERE edad BETWEEN :minEdad AND :maxEdad")
public Usuario[] getUsuariosRangoEdad(int minEdad, int maxEdad);
```

Es posible que necesitemos parametrizar una colección de datos para realizar subconsultas. En este caso, la notación empleada sería `(:coleccion)`.

```
@Query("SELECT * FROM usuario WHERE provincia IN (:provincias)")
public List<Usuario> getUsuariosProvincias(List<String> provincias);
```

Búsquedas con clase intermedia

Es habitual, que solo necesites mostrar un subconjunto de columnas de la tabla.

Room permite mostrar un objeto simple a partir de una consulta, siempre y cuando, pueda asignar el conjunto de columnas devueltos a la instancia de la clase definida.

Por ejemplo, podemos declarar una clase sencilla para recupera solo el `nombre` y `apellido` de un usuario:

```
public class NombreCompleto {
    @ColumnInfo(name = "nombre")
    public String nombre;

    @ColumnInfo(name = "apellidos")
    @NonNull
    public String apellido;
}
```

En el **DAO**, podríamos declarar el siguiente método:

```
@Query("SELECT nombre, apellidos FROM usuario")
public List<NombreCompleto> getNombreCompleto();
```

Si la búsqueda devuelve alguna columna que no se pueda asignar a un campo en el objeto, **Room** mostrará una advertencia.

Otro ejemplo:

```
@Dao
public interface UsuarioLibroDao {
    @Query("SELECT usuario.nombre AS nombreUsuario, libro.titulo AS tituloLibro " +
            "FROM usuario, libro " +
            "WHERE usuario.id = libro.idUsuario")
    public LiveData<List<UsuarioLibro>> getNombresUsuarioLibro();
}

public class UsuarioLibro {
    public String nombreUsuario;
    public String tituloLibro;
}
```

Búsquedas con multimapas

También es posible obtener un mapa de datos, en donde un objeto pueda tener asignados una colección de otros objetos:

```
@Query(
        "SELECT * FROM usuario" +
                "JOIN libro ON usuario.id = libro.id_usuario" +
                "GROUP BY usuario.nombre WHERE COUNT(libro.id) >= 3"
)
public Map<Usuario, List<Libro>> getUsuariosConLibros();
```

NOTA:

Cuando usas **LiveData** o **Flow** en una consulta de tu **DAO**, **Room** se encarga de:

1. Ejecutar la consulta en un hilo en segundo plano.
2. Notificar automáticamente a la **UI** cuando los datos cambian.

Por lo tanto, no necesitas manejar los hilos manualmente con **Executor** y **postValue**.

Si tu **DAO** devuelve una colección como **List<>**, la consulta debe ejecutarse a través de un hilo en segundo plano (Executors) antes de usarlo en **UI** (runOnUIThread).

4.3. Base de datos

Para definir la base de datos en **Room**, es necesario crear una clase que debe cumplir con las siguientes condiciones:

- La clase debe tener una anotación `@Database`. En su atributo `entities` se declara un array de anotaciones `@Entity` con aquellas entidades que conforman la base de datos.
- Debe ser una clase abstracta que extienda de `RoomDatabase`.
- Para cada clase **DAO**, hay que definir un método abstracto con cero argumentos y muestre una instancia de la clase **DAO**.

Un ejemplo sería:

```
@Database(entities = {Usuario.class}, version = 1)
public abstract class BaseDatosApp extends RoomDatabase {
    public abstract UsuarioDao usuarioDao();
}
```

Es habitual, implementar un patrón **Singleton** dentro de la clase que define la base de datos, además de incluir un **pool** de hilos:

```
@Database(
        entities = {Usuario.class},
        version = 1,
        exportSchema = false)
@TypeConverters({ConversorRoom.class})
public abstract class BasedatosUsuario extends RoomDatabase {

    public abstract UsuarioDAO usuarioDao();

    //Implementar un patrón Singleton
    private static volatile BasedatosUsuario INSTANCIA;
    private static final int NUMBER_OF_THREADS = 4;
    static final ExecutorService servicioExecutor =
            Executors.newFixedThreadPool(NUMBER_OF_THREADS);


    // Podemos añadir synchronized para que solo se pueda crear una instancia de la BD
    public static BasedatosUsuario getInstance(Context context) {

        if (INSTANCIA == null) {
            synchronized (BasedatosUsuario.class) {
                if (INSTANCIA == null) {
                    INSTANCIA = crearInstancia(context);
                }
            }
        }
        return INSTANCIA;
    }

    private static BasedatosUsuario crearInstancia(Context context) {
        return Room.databaseBuilder(context, BasedatosUsuario.class, Constantes.BBDD).
                build();
    }

    public static void anularInstancia() {
        INSTANCIA = null;

    }
}
```

4.4. Poblar base de datos

Por algunos motivos, comprobar el funcionamiento de la base de datos, es interesante añadir datos de prueba a la base de datos.

Es necesario crear un a instancia de **`RoomDatabase.Callback`** e incluirla en la creación de la base de datos.

En la instancia del **`callback`** podemos sobreescribir varios métodos asociados al ciclo de vida de **`Room`**, como `onCreate()`.

La implementación del **`callback`** se disparará cuando efectivamente se **cree la base de datos**, lo que ocurrirá cuando se ejecute la **primera operación de acceso** desde la capa de negocio.

Ejemplo:

```
private static RedSocial crearInstancia(Context context) {
    Log.d(TAG, "Creando instancia de la base de datos");
    return Room.databaseBuilder(context, RedSocial.class, BASEDATOS)
            .allowMainThreadQueries()
            .addCallback(accionesCicloVida)
            .build();
}

// Poblar BBDD mediante callback
static RoomDatabase.Callback accionesCicloVida = new RoomDatabase.Callback() {

    @Override
    public void onCreate(@NonNull SupportSQLiteDatabase db) {
        super.onCreate(db);
        Log.d(TAG, "Base de datos creada, poblando datos...");
        // Ejecutar en el executor para evitar bloquear el hilo principal
        servicioExecutor.execute(() -> {

            poblarBaseDatos(RedSocial.INSTANCIA);
        });
    }

    @Override
    public void onOpen(@NonNull SupportSQLiteDatabase db) {
        super.onOpen(db);
        Log.d(TAG, "Base de datos abierta");
    }
};
```

Para forzar la creación de la base de datos podemos usar este método en la clase que extiende de **`RoomDatabase`**:

```
/**
 * Fuerza la creación de la base de datos.
 * La base de datos no se crea hasta que la primera operación de acceso.
 */
private static void inicializar() {
    //Fuerza la creación de la base de datos
    servicioExecutor.execute(() -> {
        INSTANCIA.getOpenHelper().getWritableDatabase();
    });
}
```

4.5. Conversores de tipo (TypeConverters)

SQLite únicamente admite un conjunto limitado de tipos de datos (`INTEGER`, `REAL`, `TEXT` y `BLOB`). Sin embargo, en **Java** es frecuente usar tipos más ricos como `Date`, `UUID`, `List<String>` o incluso clases propias.

Para que **Room** pueda almacenar estos valores, debemos crear un **conversor de tipos** mediante la anotación `@TypeConverter`.

Un conversor define **cómo transformar un objeto complejo en un tipo primitivo soportado por SQLite** y viceversa.

Por ejemplo, si tenemos un campo de tipo `Date` en nuestra entidad, **Room** no puede guardarlo directamente. Necesitamos un conversor que transforme el objeto `Date` en un `Long` (**timestamp** en milisegundos):

```
public class ConversorRoom {

    @TypeConverter
    public static Date toDate(Long timestamp) {
        return timestamp == null ? null : new Date(timestamp);
    }

    @TypeConverter
    public static Long toTimestamp(Date date) {
        return date == null ? null : date.getTime();
    }
}
```

Para que **Room** utilice este conversor, debemos registrarlo en la clase que define la base de datos:

```
@Database(
        entities = {Nota.class},
        version = 1,
        exportSchema = false)
@TypeConverters({ConversorRoom.class})
public abstract class BasedatosNota extends RoomDatabase {

    public abstract NotaDAO notaDao();
}
```

De este modo, cada vez que **Room** lea o escriba un atributo `Date`, sabrá automáticamente cómo convertirlo hacia y desde la base de datos.

5. Uso de Room

Después de definir la entidad de datos, el **DAO** y el objeto de base de datos, puedes usar el siguiente código para crear una instancia de la base de datos:

```
BaseDatosApp db = Room.databaseBuilder(getApplicationContext(),
        BaseDatosApp.class, "nombre-basedatos").build();
```

Después, podemos usar los métodos abstractos de `BaseDatosApp` para obtener una instancia del **DAO**. A su vez, podemos invocar los métodos de la instancia del **DAO** para interactuar con la base de datos:

```
UsuarioDao usuarioDao = db.usuarioDao();
List<Usuario> usuarios = usuarioDao.getUsuarios();
```

Un ejemplo de uso sería:

```
nota = new Nota(titulo, contenido);
BasedatosNota.servicioExecutor.execute(() -> {
    long notaId = basedatosNota.notaDao().insertarNota(nota);
    // Asignar el id autogenerado por Room al objeto
    nota.setId_nota(notaId);

    //Volver a la actividad anterior
    runOnUiThread(() -> {
         // Asignamos el id dado por Room
        setResult(RESULT_CREATED, new Intent().putExtra(NOTA, nota));
        finish();
    });
});
```

6. Objetos incorporados (Embedded)

La anotación `@Embedded` se emplea para marcar atributos de tipo objeto. Esta anotación le comunica a Room que debe expandir los atributos del objeto marcado con `@Embedded` e integrarlos con el resto de atributos de la clase contenedora. De esta manera, **Room** considerará los campos del objeto como campos de la clase contenedora.

La anotación `@Embedded` puede usarse dentro de clases con anotación `@Entity` o en clases **POJO**. El resultado de la expansión de atributos tendrá diferentes connotaciones dependiendo del contexto de uso:

Clases @Entity

Los objetos marcados con la anotación `@Embedded` serán desnormalizados. Eso significa que sus atributos se usarán junto con los de la clase contenedora para que **Room** genere una tabla que incluya todos los campos. Por lo tanto, esta es una composición estática.

```
// Objeto para embeberse
class Direccion {
    public String calle;
    public String ciudad;
    public int codigoPostal;
}

// Entidad principal
@Entity
class Usuario {
    @PrimaryKey
    public int id;

    public String nombre;

    @Embedded
    public Direccion direccion;
    // Los campos de 'Direccion' se convertirán en columnas en la tabla 'Usuario'
}
```

La tabla **Usuario** en **Room** estará formada por los siguientes campos:

- id (`INTEGER PRIMARY KEY`)
- nombre (`TEXT`)
- calle (`TEXT`)
- ciudad (`TEXT`)
- codigoPostal (`INTEGER`)

En la interfaz **DAO**:

```
@Query("SELECT * FROM Usuario")
List<Usuario> getUsuarios();
```

Clase POJO

Permite combinar datos de múltiples tablas en un único objeto, mapeando directamente las columnas del resultado de la consulta a los campos incorporados.

```
@Entity
class Direccion {
    @PrimaryKey
    public int id;

    public String calle;
    public String ciudad;
}

// Entidad Usuario
@Entity
class Usuario {
    @PrimaryKey
    public int id;

    public String nombre;
    public int direccionId; // FK
}

// POJO para el resultado de una consulta que une ambas tablas
class UsuarioConDireccion {
    @Embedded
    public Usuario usuario;

    @Embedded(prefix = "dir_")
    public Direccion direccion;
}
```

En la interfaz **DAO**:

```
@Query("SELECT Usuario.*, "
          + "Direccion.id as dir_id, "
          + "Direccion.calle as dir_calle, "
          + "Direccion.ciudad as dir_ciudad "
        + "FROM Usuario "
        + "INNER JOIN Direccion ON Usuario.direccionId = Direccion.id")
List<UsuarioConDireccion> getUsuariosConDireccion();
```

De esta manera, evitamos una colision de nombres de los campos **id** de cada entidad, ya que los de **Direccion** se renombran con el prefijo **dir_**.

La anotación `@Embedded` combinada con **JOINs** manuales en la anotación `@Query`, ofrece un mecanismo **eficiente para consultar datos relacionados** entre múltiples tablas.

7. Relaciones

Como **SQLite** es una base de datos relacional, puedes definir relaciones entre entidades.

Es necesario explicar los casos de uso de la anotación `@Transaction` junto a consultas con `@Query`:

- **Consultas con relaciones** `@Relation`

  Los métodos que retornan instancias que modelan una relación hace obligatorio anotar con `@Transaction`, debido a que los campos anotados con `@Relation` se consultan por separado.

- **Consultas que devuelven resultados muy grandes**

7.1. Relaciones uno a uno

En las *relaciones de **uno a uno*** entre dos entidades, cada instancia de la entidad fuerte corresponde a una sola instancia de la entidad débil y viceversa.

Sigue estos pasos para definir y consultar relaciones ***uno a uno*** en tu base de datos:

1. Entidades

Para definir una *relación uno a uno*, construimos una clase por cada entidad. La entidad débil debe incluir una variable que haga referencia a la clave primaria de la entidad fuerte (clave foránea).

Perfil.java

```java
// Esta sería la entidad fuerte
@Entity(tableName = "perfil")
public class Perfil {
    @NonNull
    @PrimaryKey
    @ColumnInfo(name = "id_perfil")
    public Long idPerfil;

    public String bio;
    public String avatarUrl;

    public Perfil(Long idPerfil, String bio, String avatarUrl) {
        this.idPerfil = idPerfil;
        this.bio = bio;
        this.avatarUrl = avatarUrl;
    }
}
```

Usuario.java

```java
@Entity(tableName = "usuario")
public class Usuario {
    @PrimaryKey
    @ColumnInfo(name = "id_usuario")
    public Long idUsuario;

    public String nombre;

    @ColumnInfo(name = "id_perfil_fk", index = true)
    public Long idPerfil;  // FK

    public Usuario(Long idUsuario, String nombre, Long idPerfil) {
        this.idUsuario = idUsuario;
        this.nombre = nombre;
        this.idPerfil = idPerfil;
    }
}
```

Si queremos que la base de datos haga un **control de integridad referencial**, necesitamos modificar la anotación `@Entity` para especificar las claves foráneas que la componen:

```
@Entity(tableName = "usuario",
        foreignKeys = { @ForeignKey(
                entity = Perfil.class,
                parentColumns = "id_perfil",
                childColumns = "id_perfil_fk",
                onDelete = ForeignKey.CASCADE
        )}
)
public class Usuario { ... }
```

El atributo **`foreignKeys`** admite un array de anotaciones `@ForeignKey`. En cada anotación, se especifica:

- `entity`: entidad a la que se referencia con la clave foránea.
- `parentColumns`: columnas de la otra entidad con la que se relaciona.
- `childColumns`: columnas de la entidad actual que se corresponden con la clave foránea.
- `onDelete`: acción a realizar cuando la entidad referenciada ha sido borrada.
- `onUpdate`: acción a realizar cuando la entidad referenciada ha sido actualizada.
- `deferred`: Si la restricción de clave foránea debe aplazarse hasta que se complete la transacción. Por defecto es `false`.

Las acciones disponibles para `onDelete` y `onUpdate` son:

`{NO_ACTION|RESTRICT|SET_NULL|SET_DEFAULT|CASCADE}`

2. Relación

Para consultar los datos entre las dos entidades, hay que **modelar la relación** de *uno a uno*.

El modelado implica crear una clase que contenga una instancia de la clase padre anotada con `@Embedded` y otra instancia de la clase hija anotada con `@Relation`. Asigna a `parentColumn` el nombre de la columna de clave primaria de la entidad fuerte y a `entityColumn` el nombre de la columna de la entidad débil que hace referencia a la clave primaria de la entidad fuerte.

```
class RelacionUsuarioPerfil {
    @Embedded public Usuario usuario;

    @Relation(
            parentColumn = "idPerfil",
            entityColumn = "idPerfil"
    )
    public Perfil perfil;
}
```

3. Interfaz DAO

Por último, añada un método a la clase **`DAO`** que devuelva instancias de la clase de datos que modela la relación de entidades.

Este método requiere que **`Room`** ejecute dos consultas. Por lo tanto, debes añadir la anotación `@Transaction` a este método. Esto asegura que toda la operación se ejecute atómicamente.

```
interface UsuarioDAO{
    @Transaction
    @Query("SELECT * FROM Usuario")
    public List<RelationUsuarioPerfil> getUsuariosConPerfil();
}
```

7.2. Relaciones uno a varios

En las *relaciones de **uno a varios*** entre dos entidades, cada instancia de la entidad fuerte corresponde a **cero o más instancias** de la entidad débil. Además, cada instancia de la entidad débil solo puede corresponder una instancia de la entidad fuerte.

Sigue estos pasos para definir y consultar relaciones ***uno a varios*** en tu base de datos:

1. Entidades

Para definir una *relación uno a varios*, construimos una clase por cada entidad. Al igual que en una relación de uno a uno, la entidad débil debe incluir una variable que haga referencia a la clave primaria de la otra entidad (clave foránea).

Usuario.java (entidad fuerte)

```
@Entity(tableName = "usuario")
public class Usuario {
    @PrimaryKey
    @ColumnInfo(name = "id_usuario")
    public Long idUsuario;

    @ColumnInfo(name = "nombre")
    public String nombre;

    public Usuario(Long idUsuario, String nombre) {
        this.idUsuario = idUsuario;
        this.nombre = nombre;
    }
}
```

Email.java (entidad débil)

```
@Entity(tableName = "email",
        foreignKeys = {@ForeignKey(
                entity = Usuario.class,
                parentColumns = "id_usuario",
                childColumns = "id_usuario_fk",
                onDelete = ForeignKey.CASCADE )}
)
public class Email {
    @NonNull
    @PrimaryKey
    @ColumnInfo(name = "id_email")
    public Long idEmail;

    @ColumnInfo(name = "email")
    public String email;

    @ColumnInfo( name = "id_usuario_fk" , index = true)
    public Long idUsuario;

    public Email(@NonNull Long idEmail, String email, Long idUsuario) {
        this.idEmail = idEmail;
        this.email = email;
        this.idUsuario = idUsuario;
    }
}
```

En la entidad **`Email`**, la propiedad `idUsuario` se mapea a la columna `id_usuario_fk` mediante la anotación `@ColumnInfo`. Esta columna funciona como **clave foránea** que referencia la clave primaria `idUsuario` de la entidad **`Usuario`**, mapeada como `id_usuario`.

La configuración **`onDelete =`** `ForeignKey.CASCADE` garantiza la integridad referencial, eliminando automáticamente todos los emails asociados cuando se elimina un usuario.

2. Relación

Para consultar los datos entre las dos entidades, hay que **modelar la relación** de *uno a varios*.

El modelado implica crear una clase que contenga una instancia de la clase padre anotada con `@Embedded` y una lista de todas las instancias de la clase hija anotada con `@Relation`. Asigna a `parentColumn` el nombre de la columna de clave primaria de la entidad fuerte y a `entityColumn` el nombre de la columna de la entidad débil que hace referencia a la clave primaria de la entidad fuerte.

```
//No se marca como entidad
public class RelacionUsuarioEmail {
    @Embedded
    public Usuario usuario;

    @Relation(
            parentColumn = "id_usuario",
            entityColumn = "id_usuario_fk"
    )

    //Aqui se diferencia con la relacion uno a uno
    //Se declara una lista
    public List<Email> listaEmails;
}
```

3. Interfaz DAO

Por último, añada un método a la clase **`DAO`** que devuelva instancias de la clase de datos que modela la relación de entidades.

Este método requiere que **`Room`** ejecute dos consultas. Por lo tanto, debes añadir la anotación `@Transaction` a este método. Esto asegura que toda la operación se ejecute atómicamente.

```
interface UsuarioDAO{
    @Transaction
    @Query("SELECT * FROM Usuario")
    public List<RelacionUsuarioEmail> getUsuariosConEmails();
}
```

7.3. Relaciones varios a varios

En las *relaciones de varios a varios* entre dos entidades, cada instancia de una entidad puede estar asociada con cero o más instancias de la otra, y viceversa.

1. Entidades

En este tipo de relaciones, **ninguna** de las entidades **contiene una referencia** explícita **a la otra**. En su lugar, se **crea una tercera entidad de unión** (o tabla de referencias cruzadas) que actúa como enlace entre ambas.

La tabla de referencias cruzadas se caracteriza por:

1. Incluir una **columna por cada clave primaria** de las entidades relacionadas.
2. Estas columnas funcionan como **claves foráneas** que referencian a las entidades.
3. Definir una **clave primaria compuesta** formada por las claves primarias de las entidades relacionadas.

Alumno.java

```
@Entity(tableName = "alumnos")
public class Alumno {
    @PrimaryKey
    @ColumnInfo(name = "id_alumno")
    public Long idAlumno;

    @ColumnInfo
    public String nombre;

    @ColumnInfo(name = "correo")
    public String email;

    public Alumno(Long idAlumno, String nombre, String email) {
        this.idAlumno = idAlumno;
        this.nombre = nombre;
        this.email = email;
    }
}
```

Asignatura.java

```
@Entity(tableName = "asignaturas")
public class Asignatura {

    @PrimaryKey
    @ColumnInfo(name = "id_asignatura")
    public Long idAsignatura;

    @ColumnInfo(name = "nombre")
    public String nombre;

    @ColumnInfo(name = "curso")
    public String curso;

    public Asignatura(Long idAsignatura, String nombre, String curso) {
        this.idAsignatura = idAsignatura;
        this.nombre = nombre;
        this.curso = curso;
    }
}
```

Alumno_Asignatura.java

```
@Entity(
        tableName = "alumnos_asignaturas",
        primaryKeys = {"idAlumno", "idAsignatura"},
        foreignKeys = {
                @ForeignKey(
                        entity = Alumno.class,
                        parentColumns = "id_alumno",
                        childColumns = "idAlumno",
                        onDelete = ForeignKey.CASCADE
                ),
                @ForeignKey(
                        entity = Asignatura.class,
                        parentColumns = "id_asignatura",
                        childColumns = "idAsignatura",
                        onDelete = ForeignKey.CASCADE
                )
        }
)
public class Alumno_Asignatura {
    @NonNull
    @ColumnInfo(index = true)
    public Long idAlumno;

    @NonNull
    @ColumnInfo(index = true)
    public Long idAsignatura;

    public Alumno_Asignatura(Long idAlumno, Long idAsignatura) {
        this.idAlumno = idAlumno;
        this.idAsignatura = idAsignatura;
    }
}
```

En la tabla de unión, los atributos `idAlumno` e `idAsignatura` representan las claves primarias de las entidades **`Alumno`** y **`Asignatura`**, y deben ser anotadas como `@NonNull`, ya que no pueden tener valores nulos (al ser claves primarias).

Estas columnas suelen marcarse como índices para mejorar el rendimiento de la base de datos. Por lo tanto, si no añadimos el atributo index en la anotación `@ColumnInfo`, Room mostrará una advertencia en Android Studio.

Para crear la **clave primaria compuesta**, en la anotación `@Entity` debemos especificar los campos de la clase unión en el atributo **`primaryKeys`**:

```
...
primaryKeys = {"idAlumno", "idAsignatura"}
...
```

Para declarar las **claves foráneas**, en la anotación `@Entity` debemos especificarlas mediante el atributo **`foreignKeys.`**

```
@ForeignKey(
            entity = Alumno.class,
            parentColumns = "id_alumno",
            childColumns = "idAlumno",
            onDelete = ForeignKey.CASCADE
),
```

En el atributo `entity`, se indica la clase a la que se referencia. En `parentColumns`, se especifica el nombre de la clave primaria de esa entidad (`entity`), mientras que en `childColumns` se indica el campo de la tabla de unión que la referencia.

2. Relación

Para consultar los datos entre dos entidades en una relación *muchos a muchos*, se pueden modelar diferentes vistas según el sentido de la consulta. Por ejemplo, si tenemos una entidad **Alumno** y otra **Asignatura**, podemos obtener todas las asignaturas vinculadas a un alumno, o bien todos los alumnos asociados a una asignatura.

El modelado implica crear una clase para cada tipo de relación. Estas clases **no son entidades** de la base de datos, sino modelos que combinan:

- una instancia de la **clase principal**, anotada con `@Embedded`
- una lista de las instancias relacionadas, anotada con `@Relation`.

En la anotación `@Relation`:

- `parentColumn` indica el nombre de la columna de clave primaria de la **entidad principal**.
- `entityColumn` indica la columna de la otra entidad que hace referencia a esa clave primaria.
- `associateBy` define la entidad de unión (tabla de referencias cruzadas) que establece la relación entre ambas entidades.

AlumnoConAsignaturas.java

```
public class AlumnoConAsignaturas {
    @Embedded
    public Alumno alumno;

    @Relation(
            entity = Asignatura.class,
            parentColumn = "id_alumno",
            entityColumn = "id_asignatura",
            associateBy = @Junction(
                    value = Alumno_Asignatura.class,
                    parentColumn = "idAlumno",
                    entityColumn = "idAsignatura")
    )
    public List<Asignatura> asignaturas;
}
```

AsignaturaConAlumnos.java

```
public class AsignaturaConAlumnos {
    @Embedded
    public Asignatura asignatura;

    @Relation(
            entity = Alumno.class,
            parentColumn = "id_asignatura",
            entityColumn = "id_alumno",
            associateBy = @Junction(
                    value = Alumno_Asignatura.class,
                    parentColumn = "idAsignatura",
                    entityColumn = "idAlumno")
    )
    public List<Alumno> alumnos;
}
```

Se han utilizado nombres diferentes para las claves primarias, a fin de identificarlas claramente.

- En `entityColumn` se especifica la columna de la entidad indicada en `entity`.
- En `parentColumn` se define la clave primaria de la entidad anotada con `@Embedded`.
- En `associateBy` se referencian los **campos de la tabla de unión**.

3. Interfaz DAO

Por último, añade un método a cada clase **DAO** que devuelva **instancias de la clase** de datos **que modela la relación** de entidades.

Este método requiere que **Room** ejecute dos consultas. Por lo tanto, debes añadir la anotación `@Transaction` a este método. Esto asegura que toda la operación se ejecute atómicamente.

AlumnoDAO.java

```
@Dao
public interface AlumnoDAO {
    ...

    @Transaction
    @Query("SELECT * FROM alumnos")
    public LiveData<List<AlumnoConAsignaturas>> getAlumnosConAsignaturas();

}
```

AsignaturaDAO.java

```
@Dao
public interface AsignaturaDAO {
    ...

    @Transaction
    @Query("SELECT * FROM asignaturas")
    public LiveData<List<AsignaturaConAlumnos>> getAsignaturasConAlumnos();

}
```

AlumnoAsignaturaDAO.java

```
@Dao
public interface AlumnoAsignaturaDAO {

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    long[] insertarTodos(List<Alumno_Asignatura> alumnoAsignatura);
}
```

Por cada entidad, es habitual definir un **DAO** para especificar las operaciones básicas **CRUD**, y es aplicable a la **entidad de unión**.

Recuerda, que en la anotación `@Database` debemos especificar todas las entidades.

7.4. Relaciones anidadas

Es posible que, en ocasiones, debas buscar un conjunto de tres o más tablas relacionadas entre sí. En ese caso, definirías *relaciones anidadas* entre las tablas.

La búsqueda de datos con relaciones anidadas requiere que Room manipule un volumen de datos grande y puede afectar el rendimiento.

Para más información:

https://developer.android.com/training/data-storage/room/relationships/nested?hl=es-419

8. Vistas

Al igual que con las entidades, se pueden ejecutar instrucciones `SELECT` sobre las vistas. Sin embargo, no es posible realizar operaciones `INSERT`, `UPDATE` ni `DELETE` sobre ellas.

Las vistas se utilizan principalmente para:

- Obtener información con una estructura más amigable.
- Ocultar los detalles internos del esquema de la base de datos.
- Resumir consultas que impliquen el uso de la cláusula **`JOIN`**.

Para crear una vista en **`Room`**, se utiliza la anotación `@DatabaseView` en una clase que representará el esquema del resultado. El parámetro `value` define la sentencia `SELECT`, mientras que `viewName` permite asignar el nombre de la vista.

Ejemplo básico:

```
@DatabaseView(
    value = "SELECT t1.columna1, t1.columna2, t2.columna1, ... " +
            "FROM tabla1 as t1 " +
                "INNER JOIN tabla2 as t2 ON t1.columna1 = t2.columna1 " +
            "WHERE [condicion]",
    viewName = "vista_ejemplo"
)
public class VistaEjemplo {

    public tipo columna1;
    public tipo columna2;
    ...

}
```

Para que la vista se incluya en la base de datos, se debe agregar en la propiedad views de la anotación `@Database`, como se muestra a continuación:

```
@Database(
        entities = {Alumno.class, Asignatura.class, Alumno_Asignatura.class},
        views = {VistaEjemplo.class},
        version = 1,
        exportSchema = false)
public abstract class Colegio extends RoomDatabase { ... }
```

8.1. Definición de una vista

En el siguiente ejemplo, se crea una clase con la anotación `@DatabaseView` para crear una vista basada en una relación **`N:N`**, es decir, entre dos entidades unidas mediante una tabla de referencias cruzadas.

La consulta **`SQL`** permite obtener las asignaturas junto con sus alumnos:

```
SELECT asig.id_asignatura, asig.nombre AS nombre_asignatura,
       asig.curso, a.id_alumno, a.nombre AS nombre_alumno, a.correo
FROM asignaturas asig
INNER JOIN alumnos_asignaturas aa ON asig.id_asignatura = aa.idAsignatura
INNER JOIN alumnos a ON aa.idAlumno = a.id_alumno
```

El nombre asignado a esta vista será:

```
vista_asignaturas_alumnos
```

Por cada campo de la proyección de la consulta **`SQL`**, se debe definir un atributo en la clase correspondiente, de manera que esta actúe como modelo para mapear el resultado.

VistaAsignaturaAlumno.java

```
@DatabaseView(
        value = "SELECT " +
                "asig.id_asignatura, " +
                "asig.nombre AS nombre_asignatura, " +
                "asig.curso, " +
                "a.id_alumno, " +
                "a.nombre AS nombre_alumno, " +
                "a.correo " +
                "FROM asignaturas asig " +
                "INNER JOIN alumnos_asignaturas aa ON asig.id_asignatura = aa.idAsignatura " +
                "INNER JOIN alumnos a ON aa.idAlumno = a.id_alumno",
        viewName = "vista_asignaturas_alumnos"
)
public class VistaAsignaturaAlumno {
    @ColumnInfo(name = "id_asignatura")
    public Long idAsignatura;

    @ColumnInfo(name = "nombre_asignatura")
    public String nombreAsignatura;

    @ColumnInfo(name = "curso")
    public String curso;

    @ColumnInfo(name = "id_alumno")
    public Long idAlumno;

    @ColumnInfo(name = "nombre_alumno")
    public String nombreAlumno;

    @ColumnInfo(name = "correo")
    public String email;
}
```

8.2. Asociar vista a la BBDD

La forma de asociar una vista con la base de datos es mediante el atributo `views` de la anotación `@Database`. En el siguiente ejemplo, se vinculan las vistas creadas a la clase **`Colegio`**:

```
@Database(
        entities = {Alumno.class, Asignatura.class, Alumno_Asignatura.class},
        views = {VistaAsignaturaConteo.class, VistaAsignaturaAlumno.class},
        version = 1,
        exportSchema = false)
public abstract class Colegio extends RoomDatabase {
```

8.3. Interfaz DAO

Para exponer los resultados de ejecución de las vistas, es necesario que definir métodos de acceso en la interfaz **`DAO`**. Estos métodos pueden incluir parámetros para filtrar la información.

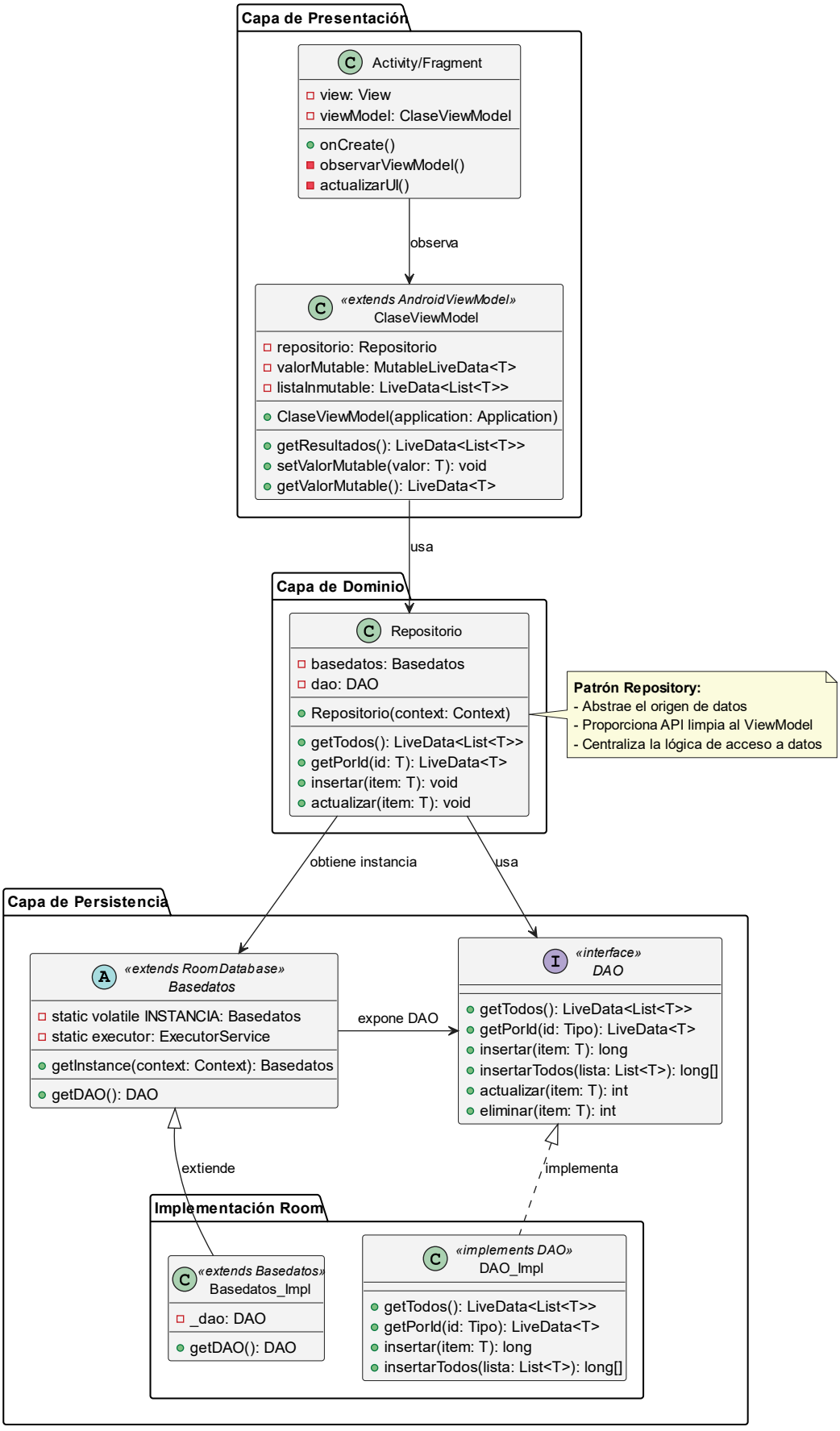
```
@Dao
public interface AsignaturaDAO {
    ...

    // CONSULTAR VISTAS
    @Query("SELECT * FROM vista_asignaturas_alumnos WHERE id_asignatura = :idAsignatura")
    public LiveData<List<VistaAsignaturaAlumno>> getVista_AsignaturasConAlumnos(Long
idAsignatura);

    @Query("SELECT * FROM vista_asignaturas_conteo ORDER BY total_alumnos DESC")
    public LiveData<List<VistaAsignaturaConteo>> getVista_AsignaturasPorPopularidad();
}
```

9. Patrón repositorio+LiveData

El siguiente diagrama muestra un patrón Repositorio recomendado por Android.



El diagrama deja patente que **Room**, en base a las interfaces y clases abstractas creadas, genera de forma interna la implementación de ellas. Room realiza consultas asíncronas añadiendo una capa de abstracción que nos evita tener que añadir más código para generar hilos de trabajo (**workers**) al realizar operaciones sobre la base de datos.

La biblioteca de **Room** incluye integraciones con varios frameworks diferentes a fin de permitir la ejecución de búsquedas asíncronas.

Las búsquedas **DAO** se dividen en tres categorías:

- Búsquedas de *escritura única* que insertan, actualizan o borran datos en la base de datos.
- Búsquedas de *lectura única* que leen datos de tu base de datos solo una vez y muestran un resultado con la instantánea de la base de datos en ese momento.
- Búsquedas de *lectura observable* que leen datos de tu base de datos cada vez que cambian las tablas de la base de datos subyacentes y emiten valores nuevos para reflejar esos cambios.

Para más información:

https://developer.android.com/training/data-storage/room/async-queries?hl=es-419#guava-livedata

9.1. Flujo de datos típico: Insertar un Nuevo Elemento

Flujo ascendente (Usuario → Room → SQLite)

1. **Usuario:** pulsa el botón "Añadir" en la pantalla.
2. **Activity/Fragment:** recoge datos de los `EditText`. Al pulsar un botón de **Guardar**, se llama a un método del **ViewModel**: `viewModel.insertar(dato)`.

```
botonGuardar.setOnClickListener(v -> {
    String campo1 = editText.getText().toString();
    String campo2 = editText2.getText().toString();
    ...

    viewModel.insertar(new Asignatura(campo1, campo2, campo3));
});
```

3. **ViewModel:** propaga el dato mediante una llamada al método del **Repositorio**: `repositorio.insertar(dato)`.

```
public void insertar(Asignatura dato) {
    repositorio.insertar(dato);
}
```

4. **Repositorio:** propaga el dato mediante una llamada al método del **DAO:** `dao.insertar(dato)`. Las *operaciones de acción* (**Create, Update y Delete**) requieren un *worker* (hilo secundario) para no bloquear el hilo principal. **Room** lanza una excepción si se usa el hilo principal.

```
public void insertar(Asignatura dato) {
    Colegio.servicioExecutor.execute(new Runnable() {
        @Override
        public void run() {
            asignaturaDAO.insertar(dato);
        }
    });
}
```

5. **DAO_Impl (Room):** traduce el método `insert()` a un `INSERT` en `SQL`, y lo ejecuta en la base de datos **SQLite**.

Flujo descendente (SQLite → Room → Usuario)

6. **Base de Datos (`Room+SQLite`):** el `INSERT` modifica la base de datos. **`Room`** detecta los cambios y vuelve a ejecutar la consulta de tipo **`Query`** que devolvía un `LiveData<List<T>>`, lo que provoca una actualización en los datos.

```
@Query("SELECT * FROM asignaturas ORDER BY nombre ASC")
LiveData<List<Asignatura>> getAsignaturas();
```

7. **ViewModel:** define un método que expone el `LiveData<List<T>>` almacenado en un miembro de la clase.

```
private LiveData<List<Asignatura>> listaAsignaturas;

public LiveData<List<Asignatura>> getAsignaturas() {
    return listaAsignaturas;
}
```

8. **Activity/Fragment:** se define un `observer` hacía `LiveData<List<T>>` del **`ViewModel`**, al invocar el método que expone la variable.

```
Observer<List<Asignatura>> observer = new Observer<List<Asignatura>>() {
    @Override
    public void onChanged(List<Asignatura> listaAsignaturas) {
        actualizarUI(listaAsignaturas);
    }
};

viewModel.getAsignaturas().observe(this, observer);
```

9. **Pantalla:** Se redibuja mostrando el nuevo elemento mediante la llamada al método `actualizarUI()`.

En **`ViewModel`** podemos usar `MutableLiveData` y `LiveData`. Su uso depende de:

- `MutableLiveData` es para **escribir y leer** (se usa **dentro** del **`ViewModel`**).
- `LiveData` es solo para **leer** (se expone **a la `Activity`**). Por eso los getters devuelven `LiveData<T>`, aunque internamente sea un `MutableLiveData<T>`. Esto protege los datos.

Las variables `MutableLiveData` almacenadas en el **`ViewModel`** modifican su valor mediante llamadas a los métodos `setValue()` o `postValue()`:

- `setValue()`: se debe emplear desde el **hilo principal**.

  Si se realiza una llamada a este método desde un hilo secundario (*worker*) saltará una *excepción de concurrencia*.

- `postValue()`: puede usarse desde **cualquier hilo**.

  Si se llama desde un *worker*, publicará (postea) el valor de *forma asíncrona* en el **hilo principal**.

  En arquitecturas típicas con **`Room`**, esta llamada suele encapsularse dentro de gestores de hilos como **`ExecutorService`** (o **`coroutines`** en Kotlin).