Room

Cómo guardar contenido en una base de datos local con Room

Room proporciona una capa de abstracción sobre SQLite que permite acceder a la base de datos sin problemas y, al mismo tiempo, aprovechar toda la potencia de SQLite.

Las apps que controlan grandes cantidades de datos estructurados pueden beneficiarse con la posibilidad de conservar esos datos localmente. El caso práctico más común es almacenar en caché datos relevantes. De esa manera, cuando el dispositivo no puede acceder a la red, el usuario de todos modos puede explorar ese contenido mientras está desconectado. Cualquier cambio de contenido iniciado por el usuario se sincroniza con el servidor una vez que el dispositivo vuelve a estar en línea.

Como Room se ocupa de estas inquietudes por ti, te **recomendamos** utilizar Room en lugar de SQLite. Sin embargo, si prefieres usar las API de SQLite directamente, lee [Cómo guardar datos mediante SQLite](https://developer.android.com/training/data-storage/sqlite.html).

**Nota**: Para usar Room en tu app, [declara las dependencias de Room](https://developer.android.com/jetpack/androidx/releases/room#declaring_dependencies) en el archivo **build.gradle**.

Estos son los 3 componentes principales de Room:

* [**Base de datos**](https://developer.android.com/reference/androidx/room/Database.html): Contiene el titular de la base de datos y sirve como punto de acceso principal para la conexión subyacente a los datos persistentes y relacionales de tu app.

La clase anotada con [@Database](https://developer.android.com/reference/androidx/room/Database.html) debería cumplir las siguientes condiciones:

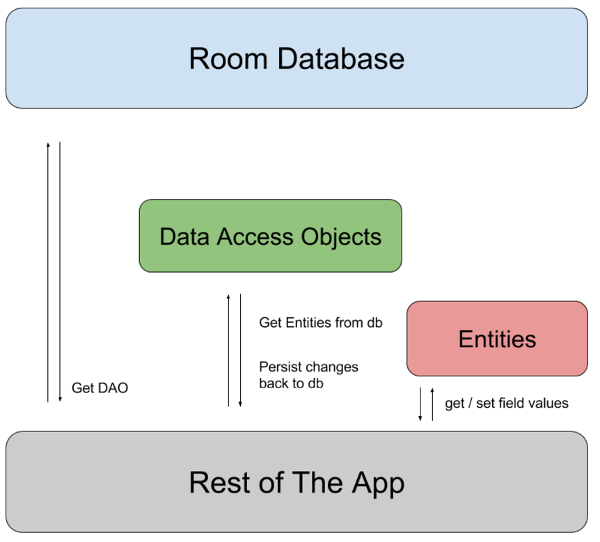
* + Ser una clase abstracta que extienda [RoomDatabase](https://developer.android.com/reference/androidx/room/RoomDatabase.html)
  + Incluir la lista de entidades asociadas con la base de datos dentro de la anotación
  + Contener un método abstracto que tenga 0 argumentos y muestre la clase anotada con [@Dao](https://developer.android.com/reference/androidx/room/Dao.html)

Durante el tiempo de ejecución, puedes adquirir una instancia de [Database](https://developer.android.com/reference/androidx/room/Database.html) llamando a [Room.databaseBuilder()](https://developer.android.com/reference/androidx/room/Room.html#databaseBuilder(android.content.Context,%20java.lang.Class%3CT%3E,%20java.lang.String)) o [Room.inMemoryDatabaseBuilder()](https://developer.android.com/reference/androidx/room/Room.html#inMemoryDatabaseBuilder(android.content.Context,%20java.lang.Class%3CT%3E)).

* [**Entidad**](https://developer.android.com/training/data-storage/room/defining-data.html): Representa una tabla dentro de la base de datos.
* [**DAO**](https://developer.android.com/training/data-storage/room/accessing-data.html): Contiene los métodos utilizados para acceder a la base de datos.

La app usa la base de datos de Room para obtener los objetos de acceso a los datos (DAO) asociados con esa base de datos. Luego, la app usa cada DAO para obtener entidades de la base de datos y guardar los cambios realizados en esas entidades en la base de datos. Por último, la app usa una entidad para obtener y configurar valores que corresponden a columnas de tabla dentro de la base de datos.

Esta relación entre los diferentes componentes de Room aparece en la figura 1:

**Figura 1:** Diagrama de arquitectura de Room

En el siguiente fragmento de código, se muestra una configuración de base de datos de ejemplo con una entidad y un DAO:

User

[KOTLIN](https://developer.android.com/training/data-storage/room#kotlin)[JAVA](https://developer.android.com/training/data-storage/room#java)

    @Entity  
    public class User {  
        @PrimaryKey  
        public int uid;  
  
        @ColumnInfo(name = "first\_name")  
        public String firstName;  
  
        @ColumnInfo(name = "last\_name")  
        public String lastName;  
    }

UserDao

[KOTLIN](https://developer.android.com/training/data-storage/room#kotlin)[JAVA](https://developer.android.com/training/data-storage/room#java)

    @Dao  
    public interface UserDao {  
        @Query("SELECT \* FROM user")  
        List<User> getAll();  
  
        @Query("SELECT \* FROM user WHERE uid IN (:userIds)")  
        List<User> loadAllByIds(int[] userIds);  
  
        @Query("SELECT \* FROM user WHERE first\_name LIKE :first AND " +  
               "last\_name LIKE :last LIMIT 1")  
        User findByName(String first, String last);  
  
        @Insert  
        void insertAll(User... users);  
  
        @Delete  
        void delete(User user);  
    }

AppDatabase

[KOTLIN](https://developer.android.com/training/data-storage/room#kotlin)[JAVA](https://developer.android.com/training/data-storage/room#java)

    @Database(entities = {User.class}, version = 1)  
    public abstract class AppDatabase extends RoomDatabase {  
        public abstract UserDao userDao();  
    }

Después de crear los archivos anteriores, obtendrás una instancia de la base de datos creada con el siguiente código:

[KOTLIN](https://developer.android.com/training/data-storage/room#kotlin)[JAVA](https://developer.android.com/training/data-storage/room#java)

    AppDatabase db = Room.databaseBuilder(getApplicationContext(),  
            AppDatabase.class, "database-name").build();

**Nota**: Si tu app se ejecuta en un solo proceso, debes seguir el patrón de diseño único cuando creas instancias de un objeto **AppDatabase**. Cada instancia de [**RoomDatabase**](https://developer.android.com/reference/androidx/room/RoomDatabase.html) es bastante costosa, pero rara vez necesitas acceso a varias instancias dentro de un solo proceso.

Si tu app se ejecuta en múltiples procesos, incluye **enableMultiInstanceInvalidation()** en tu invocación del generador de bases de datos. De esta manera, cuando tienes una instancia de **AppDatabase** en cada proceso, puedes invalidar el archivo de base de datos compartida en un proceso y esta invalidación se propaga automáticamente a las instancias de **AppDatabase** dentro de los otros procesos.

# Cómo definir datos con entidades Room

Cuando usas la [biblioteca de persistencias Room](https://developer.android.com/training/data-storage/room/index.html?hl=es-419), defines conjuntos de campos relacionados como entidades. Para cada entidad, se crea una tabla dentro del objeto [Database](https://developer.android.com/reference/androidx/room/Database.html?hl=es-419) asociado con el fin de retener los elementos. Debes hacer referencia a la clase de entidad mediante el arreglo de [entities](https://developer.android.com/reference/androidx/room/Database.html?hl=es-419#entities()) en la clase [Database](https://developer.android.com/reference/androidx/room/Database.html?hl=es-419).

**Nota**: Para usar entidades en tu app, [agrega los artefactos de los componentes de la arquitectura](https://developer.android.com/topic/libraries/architecture/adding-components.html?hl=es-419) al archivo **build.gradle** de tu app.

En el siguiente fragmento de código, se muestra cómo definir una entidad:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

**@Entity**  
    public class User {  
        @PrimaryKey  
        public int id;  
  
        public String firstName;  
        public String lastName;  
    }

Para conservar un campo, Room debe tener acceso a él. Puedes hacer que un campo sea público o puedes proporcionarle un captador y un colocador. Si usas los métodos de captador y colocador, ten en cuenta que están basados en las convenciones de JavaBeans en Room.

**Nota**: Las entidades pueden tener un constructor vacío (si la clase [DAO](https://developer.android.com/training/data-storage/room/accessing-data.html?hl=es-419) correspondiente puede acceder a cada campo persistente) o un constructor cuyos parámetros contengan tipos y nombres que coincidan con los de los campos de la entidad.Room también puede usar constructores completos o parciales, como un constructor que recibe solo algunos de los campos.

## **Cómo usar una clave primaria**

Cada entidad debe definir al menos 1 campo como clave primaria. Incluso cuando solo hay 1 campo, aún necesitas usar la anotación [@PrimaryKey](https://developer.android.com/reference/androidx/room/PrimaryKey.html?hl=es-419). Además, si quieres que Room asigne ID automáticos a entidades, puedes configurar la propiedad [autoGenerate](https://developer.android.com/reference/androidx/room/PrimaryKey.html?hl=es-419#autoGenerate()) de @PrimaryKey. Si la entidad tiene una clave primaria compuesta, puedes usar la propiedad [primaryKeys](https://developer.android.com/reference/androidx/room/Entity.html?hl=es-419#primaryKeys()) de la anotación [@Entity](https://developer.android.com/reference/androidx/room/Entity.html?hl=es-419), como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Entity(**primaryKeys = {"firstName", "lastName"}**)  
    public class User {  
        public String firstName;  
        public String lastName;  
    }

De manera predeterminada, 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](https://developer.android.com/reference/androidx/room/Entity.html?hl=es-419#tableName()) de la anotación [@Entity](https://developer.android.com/reference/androidx/room/Entity.html?hl=es-419), como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Entity(**tableName = "users"**)  
    public class User {  
        // ...  
    }

**Precaución**: En SQLite, los nombres de tabla no distinguen entre mayúsculas y minúsculas.

Al igual que la propiedad [tableName](https://developer.android.com/reference/androidx/room/Entity.html?hl=es-419#tableName()), Room usa los nombres de campo como nombres de columna en la base de datos. Si quieres que una columna tenga un nombre diferente, agrega la anotación [@ColumnInfo](https://developer.android.com/reference/androidx/room/ColumnInfo.html?hl=es-419) a un campo, como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Entity(tableName = "users")  
    public class User {  
        @PrimaryKey  
        public int id;  
  
        **@ColumnInfo(name = "first\_name")  
        public String firstName;  
  
        @ColumnInfo(name = "last\_name")  
        public String lastName;**  
    }

## **Cómo ignorar campos**

De manera predeterminada, Room crea una columna para cada campo que se define en la entidad. Si una entidad tiene campos que no deseas conservar, puedes usar la anotación [@Ignore](https://developer.android.com/reference/androidx/room/Ignore.html?hl=es-419) en ellos, como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Entity  
    public class User {  
        @PrimaryKey  
        public int id;  
  
        public String firstName;  
        public String lastName;  
  
        **@Ignore  
        Bitmap picture;**  
    }

En los casos en que una entidad hereda campos de una entidad principal, suele ser más fácil usar la propiedad [ignoredColumns](https://developer.android.com/reference/androidx/room/Entity?hl=es-419#ignoredcolumns) del atributo @Entity:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Entity(**ignoredColumns = "picture"**)  
    public class RemoteUser extends User {  
        @PrimaryKey  
        public int id;  
  
        public boolean hasVpn;  
    }

## **Cómo proporcionar compatibilidad con la búsqueda de tablas**

Room admite varios tipos de anotaciones que facilitan la búsqueda de detalles en las tablas de la base de datos. Usa la búsqueda en el texto completo a menos que la versión de minSdkVersion de tu app sea inferior a 16.

### **Cómo admitir la búsqueda en el texto completo**

Si tu app requiere un acceso muy rápido a la información de la base de datos mediante la búsqueda en el texto completo (FTS), respalda tus entidades mediante una tabla virtual que use el [módulo de extensión SQLite](https://www.sqlite.org/fts3.html) FTS3 o FTS4. Para usar esta función, disponible en Room 2.1.0 y versiones posteriores, agrega la anotación [@Fts3](https://developer.android.com/reference/androidx/room/Fts3?hl=es-419) o [@Fts4](https://developer.android.com/reference/androidx/room/Fts4?hl=es-419) a una entidad dada, como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    // Use `@Fts3` only if your app has strict disk space requirements or if you  
    // require compatibility with an older SQLite version.  
    **@Fts4**  
    @Entity(tableName = "users")  
    public class User {  
        // Specifying a primary key for an FTS-table-backed entity is optional, but  
        // if you include one, it **must** use this type and column name.  
        **@PrimaryKey  
        @ColumnInfo(name = "rowid")**  
        public int id;  
  
        @ColumnInfo(name = "first\_name")  
        public String firstName;  
    }

**Nota**: Las tablas habilitadas para FTS siempre usan una clave primaria de tipo **INTEGER** y el nombre de columna "rowid".Si la entidad respaldada por tablas FTS define una clave primaria, **debe** usar ese tipo y nombre de columna.

En los casos en que una tabla admita contenido en varios idiomas, usa la opción languageId para especificar la columna que almacena la información de idioma de cada fila:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Fts4**(languageId = "lid")**  
    @Entity(tableName = "users")  
    public class User {  
        // ...  
  
        @ColumnInfo(name = "lid")  
        int languageId;  
    }

Room ofrece varias opciones para definir entidades respaldadas por FTS, como el orden de los resultados, los tipos de tokenizadores y las tablas administradas como contenido externo. Para obtener más información sobre estas opciones, consulta la referencia de [FtsOptions](https://developer.android.com/reference/androidx/room/FtsOptions?hl=es-419).

### **Columnas específicas del índice**

Si tu app debe admitir versiones de SDK que no permiten usar entidades respaldadas por tablas FTS3 o FTS4, puedes indexar ciertas columnas en la base de datos para agilizar tus consultas. Para agregar índices a una entidad, incluye la propiedad [indices](https://developer.android.com/reference/androidx/room/Entity.html?hl=es-419#indices()) dentro de la anotación [@Entity](https://developer.android.com/reference/androidx/room/Entity.html?hl=es-419), enumerando los nombres de las columnas que deseas incluir en el índice o en el índice compuesto. En el siguiente fragmento de código, se muestra este proceso de anotación:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Entity(**indices = {@Index("name"),  
            @Index(value = {"last\_name", "address"})}**)  
    public class User {  
        @PrimaryKey  
        public int id;  
  
        public String firstName;  
        public String address;  
  
        @ColumnInfo(name = "last\_name")  
        public String lastName;  
  
        @Ignore  
        Bitmap picture;  
    }

A veces, ciertos campos o grupos de campos de una base de datos deben ser únicos. Puedes aplicar esta propiedad de unicidad mediante la propiedad [unique](https://developer.android.com/reference/androidx/room/Index.html?hl=es-419#unique()) de una anotación [@Index](https://developer.android.com/reference/androidx/room/Index.html?hl=es-419) en true. En el siguiente ejemplo de código, se evita que una tabla tenga dos filas con el mismo conjunto de valores para las columnas firstName y lastName:

[KOTLIN](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419#java)

    @Entity(indices = {@Index(value = {"first\_name", "last\_name"},  
            **unique = true**)})  
    public class User {  
        @PrimaryKey  
        public int id;  
  
        @ColumnInfo(name = "first\_name")  
        public String firstName;  
  
        @ColumnInfo(name = "last\_name")  
        public String lastName;  
  
        @Ignore  
        Bitmap picture;  
    }

## **Cómo incluir objetos basados en AutoValue**

**Nota**: Esta función está diseñada para usarse solo en entidades basadas en Java. Para lograr la misma funcionalidad en las entidades basadas en Kotlin, es mejor usar [clases de datos](https://kotlinlang.org/docs/reference/data-classes.html).

En la Room 2.1.0 y versiones posteriores, puedes usar [clases de valores inmutables](https://github.com/google/auto/blob/master/value/userguide/index.md) basadas en Java, que puedes anotar mediante @AutoValue, como entidades en la base de datos de tu app. Esta compatibilidad es particularmente útil cuando dos instancias de una entidad se consideran iguales si sus columnas contienen valores idénticos.

Cuando se usan clases anotadas con @AutoValue como entidades, puedes anotar los métodos abstractos de la clase mediante @PrimaryKey, @ColumnInfo, @Embedded y @Relation. Sin embargo, si usas estas anotaciones, debes incluir la anotación @CopyAnnotations cada vez, a fin de que Room pueda interpretar las implementaciones autogeneradas de los métodos correctamente.

En el siguiente fragmento de código, se muestra un ejemplo de una clase anotada con @AutoValue que Room reconoce como una entidad:

User.java

**@AutoValue**  
    @Entity  
    public abstract class User {  
        // Supported annotations must include `@CopyAnnotations`.  
        **@CopyAnnotations**  
        @PrimaryKey  
        public abstract long getId();  
  
        public abstract String getFirstName();  
        public abstract String getLastName();  
  
        // Room uses this factory method to create User objects.  
        public static User create(long id, String firstName, String lastName) {  
            return new AutoValue\_User(id, firstName, lastName);  
        }  
    }

# Cómo definir relaciones entre objetos

Como SQLite es una base de datos relacional, puedes especificar relaciones entre objetos. Aunque la mayoría de las bibliotecas de asignación relacional de objetos permiten que los objetos de entidad se hagan referencia entre sí, Room lo prohíbe explícitamente. Para obtener más información sobre el razonamiento técnico que respalda esta decisión, consulta [Por qué Room no permite referencias a objetos](https://developer.android.com/training/data-storage/room/referencing-data.html?hl=es-419#understand-no-object-references).

## **Cómo definir relaciones de uno a varios**

Aunque no puedes usar relaciones directas, Room te permite definir restricciones de clave externa entre entidades.

Por ejemplo, si hay otra entidad llamada Book, puedes definir su relación con la entidad User mediante la anotación [@ForeignKey](https://developer.android.com/reference/androidx/room/ForeignKey.html?hl=es-419), como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#java)

    @Entity(**foreignKeys = @ForeignKey(entity = User.class,  
                                      parentColumns = "id",  
                                      childColumns = "user\_id")**)  
    public class Book {  
        @PrimaryKey public int bookId;  
  
        public String title;  
  
        @ColumnInfo(name = "user\_id") public int userId;  
    }

Como pueden vincularse cero o más instancias de Book a una sola instancia de User mediante la clave externa user\_id, se genera una relación de uno a varios entre User y Book.

Las claves externas son muy potentes, ya que te permiten especificar qué ocurre cuando se actualiza la entidad referenciada. Por ejemplo, puedes decirle a SQLite que borre todos los libros de un usuario si se borra la instancia correspondiente de User que incluye [onDelete = CASCADE](https://developer.android.com/reference/androidx/room/ForeignKey.html?hl=es-419#onDelete()) en la anotación [@ForeignKey](https://developer.android.com/reference/androidx/room/ForeignKey.html?hl=es-419).

**Nota**: SQLite controla [**@Insert(onConflict = REPLACE)**](https://developer.android.com/reference/androidx/room/OnConflictStrategy.html?hl=es-419#REPLACE) como un conjunto de operaciones **REMOVE** y **REPLACE**, en lugar de una sola operación **UPDATE**.Este método de reemplazo de valores en conflicto podría afectar tus restricciones de clave externa. Para obtener más información, consulta la [documentación de SQLite](https://sqlite.org/lang_conflict.html) sobre la cláusula **ON\_CONFLICT**.

## **Cómo crear objetos anidados**

Es posible que a veces quieras expresar una entidad o un objeto de datos como un solo elemento integral en la lógica de tu base de datos, incluso si el objeto contiene varios campos. En estas situaciones, puedes usar la anotación [@Embedded](https://developer.android.com/reference/androidx/room/Embedded.html?hl=es-419) para representar un objeto cuyos subcampos quieres desglosar en una tabla. Luego, puedes consultar los campos integrados tal como lo harías para otras columnas individuales.

Por ejemplo, tu clase User puede incluir un campo del 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 que esté anotado con [@Embedded](https://developer.android.com/reference/androidx/room/Embedded.html?hl=es-419), como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#java)

    public class Address {  
        public String street;  
        public String state;  
        public String city;  
  
        @ColumnInfo(name = "post\_code") public int postCode;  
    }  
  
    @Entity  
    public class User {  
        @PrimaryKey public int id;  
  
        public String firstName;  
  
        **@Embedded public Address address;**  
    }

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

**Nota**: Los campos incorporados también pueden incluir otros campos incorporados.

Si una entidad tiene varios campos incorporados del mismo tipo, puedes establecer cada columna como única mediante la configuración de la propiedad [prefix](https://developer.android.com/reference/androidx/room/Embedded.html?hl=es-419#prefix()). Luego, Room agrega el valor proporcionado al comienzo de cada nombre de columna en el objeto incorporado.

## **Cómo definir relaciones de varios a varios**

Existe otro tipo de relación que es posible que quieras modelar en una base de datos relacional: una relación de varios a varios entre dos entidades, donde cada entidad se puede vincular a cero o más instancias de la otra. Por ejemplo, imagina una app de streaming de música en la que los usuarios pueden organizar sus canciones favoritas en listas de reproducción. Cada lista de reproducción puede tener cualquier cantidad de canciones, y cada canción se puede incluir en cualquier cantidad de listas de reproducción.

Para modelar esta relación, deberás crear tres objetos:

1. Una clase de entidad para las listas de reproducción
2. Una clase de entidad para las canciones
3. Una clase intermedia para contener la información sobre las canciones que están en cada lista de reproducción

Puedes definir las clases de entidad como unidades independientes de la siguiente manera:

[KOTLIN](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#java)

    @Entity  
    public class Playlist {  
        @PrimaryKey public int id;  
  
        public String name;  
        public String description;  
    }  
  
    @Entity  
    public class Song {  
        @PrimaryKey public int id;  
  
        public String songName;  
        public String artistName;  
    }

Luego, puedes definir la clase intermedia como una entidad que contiene referencias de clave externa tanto para Song como para Playlist de la siguiente manera:

[KOTLIN](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#java)

    @Entity(tableName = "playlist\_song\_join",  
            primaryKeys = { "playlistId", "songId" },  
            foreignKeys = {  
                    @ForeignKey(entity = Playlist.class,  
                                parentColumns = "id",  
                                childColumns = "playlistId"),  
                    @ForeignKey(entity = Song.class,  
                                parentColumns = "id",  
                                childColumns = "songId")  
                    })  
    public class PlaylistSongJoin {  
        public int playlistId;  
        public int songId;  
    }

Así se produce un modelo de relación de varios a varios que te permite usar un DAO para consultar tanto las listas de reproducción por canción como las canciones por lista de reproducción:

[KOTLIN](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/relationships?hl=es-419#java)

    @Dao  
    public interface PlaylistSongJoinDao {  
        @Insert  
        void insert(PlaylistSongJoin playlistSongJoin);  
  
        @Query("SELECT \* FROM playlist " +  
               "INNER JOIN playlist\_song\_join " +  
               "ON playlist.id=playlist\_song\_join.playlistId " +  
               "WHERE playlist\_song\_join.songId=:songId")  
        List<Playlist> getPlaylistsForSong(final int songId);  
  
        @Query("SELECT \* FROM song " +  
               "INNER JOIN playlist\_song\_join " +  
               "ON song.id=playlist\_song\_join.songId " +  
               "WHERE playlist\_song\_join.playlistId=:playlistId")  
        List<Song> getSongsForPlaylist(final int playlistId);  
    }

# Cómo acceder a los datos con DAO de Room

Para acceder a los datos de tu app con la [biblioteca de persistencias Room](https://developer.android.com/training/data-storage/room/index.html?hl=es-419), trabaja con objetos de acceso a datos o DAO. Este conjunto de objetos [Dao](https://developer.android.com/reference/androidx/room/Dao.html?hl=es-419) forma el componente principal de Room, ya que cada DAO incluye métodos que ofrecen acceso abstracto a la base de datos de tu app.

Al acceder a una base de datos con una clase DAO, en lugar de compiladores de consultas o consultas directas, puedes separar diferentes componentes de la arquitectura de tu base de datos. Además, las clases DAO te permiten simular fácilmente el acceso a la base de datos mientras [pruebas tu app](https://developer.android.com/training/data-storage/room/testing-db.html?hl=es-419).

**Nota**: Antes de incluir clases DAO en tu app, [agrega los artefactos de componentes de arquitectura](https://developer.android.com/topic/libraries/architecture/adding-components.html?hl=es-419) al archivo **build.gradle** de tu app.

Una clase DAO puede ser una interfaz o una clase abstracta. Si es una clase abstracta, opcionalmente, puede tener un constructor que tome un elemento [RoomDatabase](https://developer.android.com/reference/androidx/room/RoomDatabase.html?hl=es-419) como único parámetro. Room crea cada implementación de DAO en el momento de la compilación.

**Nota**: Room no admite el acceso a la base de datos en el subproceso principal a menos que hayas llamado a [**allowMainThreadQueries()**](https://developer.android.com/reference/androidx/room/RoomDatabase.Builder.html?hl=es-419#allowMainThreadQueries()) en el compilador, ya que podría bloquear la IU durante un largo período. Las consultas asíncronas, que son aquellas que muestran instancias de [**LiveData**](https://developer.android.com/reference/androidx/lifecycle/LiveData.html?hl=es-419) o [**Flowable**](http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Flowable.html), están exentas de esta regla, ya que ejecutan la consulta de manera asíncrona en un subproceso en segundo plano cuando es necesario.

## **Cómo definir métodos para mayor comodidad**

Hay múltiples consultas de conveniencia que puedes representar mediante una clase DAO. En este documento, se incluyen varios ejemplos comunes.

### **Insert**

Cuando creas un método DAO y lo anotas con [@Insert](https://developer.android.com/reference/androidx/room/Insert.html?hl=es-419), Room genera una implementación que inserta todos los parámetros en la base de datos con una sola transacción.

En el siguiente fragmento de código, se muestran varias consultas de ejemplo:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Insert(onConflict = OnConflictStrategy.REPLACE)  
        public void insertUsers(User... users);  
  
        @Insert  
        public void insertBothUsers(User user1, User user2);  
  
        @Insert  
        public void insertUsersAndFriends(User user, List<User> friends);  
    }

Si el método [@Insert](https://developer.android.com/reference/androidx/room/Insert.html?hl=es-419) recibe solo 1 parámetro, puede mostrar un objeto long, que es el nuevo objeto rowId para el elemento insertado. Si el parámetro es un arreglo o una colección, debería mostrar long[] o List<Long> en su lugar.

Para obtener más información, consulta la documentación de referencia sobre la anotación [@Insert](https://developer.android.com/reference/androidx/room/Insert.html?hl=es-419), así como la [documentación de SQLite para las tablas de rowid](https://www.sqlite.org/rowidtable.html).

### **Update**

El método de conveniencia [Update](https://developer.android.com/reference/androidx/room/Update.html?hl=es-419) modifica un conjunto de entidades, proporcionadas como parámetros, en la base de datos. Utiliza una consulta que coincide con la clave primaria de cada entidad.

En el siguiente fragmento de código, se muestra cómo definir este método:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Update  
        public void updateUsers(User... users);  
    }

Si bien no suele ser necesario, puedes hacer que este método muestre un valor int, que indica la cantidad de filas actualizadas en la base de datos.

### **Delete**

El método de conveniencia [Delete](https://developer.android.com/reference/androidx/room/Delete.html?hl=es-419) quita un conjunto de entidades, proporcionadas como parámetros, de la base de datos. Utiliza las claves primarias para encontrar las entidades que se borrarán.

En el siguiente fragmento de código, se muestra cómo definir este método:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Delete  
        public void deleteUsers(User... users);  
    }

Si bien no suele ser necesario, puedes hacer que este método muestre un valor int, que indica la cantidad de filas que se quitaron de la base de datos.

## **Consulta de información**

[@Query](https://developer.android.com/reference/androidx/room/Query.html?hl=es-419) es la anotación principal utilizada en las clases DAO. Te permite realizar operaciones de lectura/escritura en una base de datos. Cada método [@Query](https://developer.android.com/reference/androidx/room/Query.html?hl=es-419) se verifica en el momento de la compilación, por lo que si hay un problema con la consulta, se produce un error de compilación, y no uno de tiempo de ejecución.

Room también verifica el valor que se muestra de la consulta, de modo que si el nombre del campo en el objeto que se muestra no coincide con los nombres de las columnas correspondientes en la respuesta de la consulta, Room te alerta de una de estas dos maneras:

* Muestra una advertencia si solo coinciden algunos nombres de campos.
* Muestra un error si no coinciden los nombres de los campos.

### **Consultas simples**

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT \* FROM user")  
        public User[] loadAllUsers();  
    }

Esta es una consulta muy simple que carga todos los usuarios. En el momento de compilación, Room sabe que está consultando todas las columnas de la tabla de usuarios. Si la consulta contiene un error de sintaxis o si la tabla de usuarios no existe en la base de datos, Room muestra un error con el mensaje apropiado a medida que se compila tu app.

### **Cómo pasar parámetros a la consulta**

La mayoría de las veces, debes pasar parámetros a consultas con el objetivo de realizar operaciones de filtrado, como mostrar solo usuarios mayores de cierta edad. Para llevar a cabo esta tarea, usa los parámetros del método en tu anotación Room, como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT \* FROM user WHERE age > :minAge")  
        public User[] loadAllUsersOlderThan(int minAge);  
    }

Cuando esta consulta se procesa en el momento de la compilación, Room hace coincidir el parámetro de vinculación :minAge con el parámetro del método minAge. Room realiza la coincidencia con los nombres de los parámetros. Si hay una falta de coincidencia, se produce un error al compilar tu app.

También puedes pasar varios parámetros o hacer referencia a ellos varias veces en una consulta, como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT \* FROM user WHERE age BETWEEN :minAge AND :maxAge")  
        public User[] loadAllUsersBetweenAges(int minAge, int maxAge);  
  
        @Query("SELECT \* FROM user WHERE first\_name LIKE :search " +  
               "OR last\_name LIKE :search")  
        public List<User> findUserWithName(String search);  
    }

### **Cómo mostrar subconjuntos de columnas**

En la mayoría de los casos, solo necesitas obtener algunos campos de una entidad. Por ejemplo, tu IU puede mostrar solo el nombre y el apellido de un usuario, en lugar de mostrar todos sus detalles. Gracias a que obtienes solo las columnas que aparecen en la IU de tu app, ahorras recursos valiosos y la consulta se completa más rápido.

Room te permite mostrar cualquier objeto basado en Java de tus consultas siempre que el conjunto de columnas de resultados se pueda asignar al objeto que se muestra. Por ejemplo, puedes crear el siguiente objeto antiguo y sin formato basado en Java (POJO) para obtener el nombre y apellido del usuario:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    public class NameTuple {  
        @ColumnInfo(name = "first\_name")  
        public String firstName;  
  
        @ColumnInfo(name = "last\_name")  
        @NonNull  
        public String lastName;  
    }

Ahora, puedes usar este POJO en tu método de consulta de la siguiente manera:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT first\_name, last\_name FROM user")  
        public List<NameTuple> loadFullName();  
    }

Room entiende que la consulta muestra valores en las columnas first\_name y last\_name, y que esos valores pueden asignarse a los campos de la clase NameTuple. Por lo tanto, Room puede generar el código correcto. Si la consulta muestra demasiadas columnas o una columna que no existe en la clase NameTuple, Room enviará una advertencia.

**Nota**: Estos objetos POJO también pueden usar la anotación [**@Embedded**](https://developer.android.com/reference/androidx/room/Embedded.html?hl=es-419).

### **Cómo pasar una colección de argumentos**

Es posible que algunas de las consultas requieran que se pase una cantidad variable de parámetros y que no se conozca la cantidad exacta de parámetros hasta el tiempo de ejecución. Por ejemplo, es posible que quieras recuperar información sobre todos los usuarios de un subconjunto de regiones. 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.

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT first\_name, last\_name FROM user WHERE region IN (:regions)")  
        public List<NameTuple> loadUsersFromRegions(List<String> regions);  
    }

### **Consultas observables**

Al realizar consultas, a menudo desearás que la IU de tu app se actualice automáticamente cuando cambien los datos. Para lograrlo, usa un valor de muestra de tipo [LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData.html?hl=es-419) en la descripción del método de consulta. Room genera todo el código necesario para actualizar [LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData.html?hl=es-419) cuando se actualiza la base de datos.

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT first\_name, last\_name FROM user WHERE region IN (:regions)")  
        public LiveData<List<User>> loadUsersFromRegionsSync(List<String> regions);  
    }

**Nota**: A partir de la versión 1.0, Room usa la lista de tablas a las que se accede en la consulta para decidir si se deben actualizar instancias de [**LiveData**](https://developer.android.com/reference/androidx/lifecycle/LiveData.html?hl=es-419).

### **Consultas reactivas con RxJava**

Room proporciona la siguiente compatibilidad con valores de muestra de tipos RxJava2:

* Métodos @Query: Room admite valores de muestra de tipo [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.1-javadoc/org/reactivestreams/Publisher.html), [Flowable](http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Flowable.html) y [Observable](http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Observable.html).
* Métodos @Insert, @Update y @Delete: Room 2.1.0 y versiones posteriores admite valores de muestra del tipo [Completable](http://reactivex.io/RxJava/javadoc/io/reactivex/Completable.html), [Single<T>](http://reactivex.io/RxJava/javadoc/io/reactivex/Single.html) y [Maybe<T>](http://reactivex.io/RxJava/javadoc/io/reactivex/Maybe.html).

Para usar esta funcionalidad, incluye la versión más reciente del artefacto **rxjava2** en el archivo build.gradle de tu app:

app/build.gradle

    dependencies {  
        def room\_version = "2.1.0"  
        implementation 'androidx.room:room-rxjava2:$room\_version'  
    }

Para ver las versiones actuales de esta biblioteca, consulta la información sobre Room en la página de [versiones](https://developer.android.com/jetpack/androidx/versions?hl=es-419).

En el siguiente fragmento de código, se muestran varios ejemplos de cómo podrías usar estos tipos de datos que se muestran:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT \* from user where id = :id LIMIT 1")  
        public Flowable<User> loadUserById(int id);  
  
        // Emits the number of users added to the database.  
        @Insert  
        public Maybe<Integer> insertLargeNumberOfUsers(List<User> users);  
  
        // Makes sure that the operation finishes successfully.  
        @Insert  
        public Completable insertLargeNumberOfUsers(User... users);  
  
        /\* Emits the number of users removed from the database. Always emits at  
           least one user. \*/  
        @Delete  
        public Single<Integer> deleteUsers(List<User> users);  
    }

Para obtener información detallada, consulta el artículo de Google Developers [Room y RxJava](https://medium.com/google-developers/room-rxjava-acb0cd4f3757).

### **Acceso directo al cursor**

Si la lógica de tu app requiere acceso directo a las filas que se muestran, puedes mostrar un objeto [Cursor](https://developer.android.com/reference/android/database/Cursor.html?hl=es-419) de tus consultas, como se muestra en el siguiente fragmento de código:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @Query("SELECT \* FROM user WHERE age > :minAge LIMIT 5")  
        public Cursor loadRawUsersOlderThan(int minAge);  
    }

**Precaución**: No se recomienda trabajar con la API del cursor porque no garantiza que las filas existen o qué valores contienen. Usa esta funcionalidad solo si ya tienes un código que espera un cursor y que no se puede refactorizar fácilmente.

### **Cómo hacer consultas en varias tablas**

Algunas de tus consultas pueden requerir acceso a varias tablas para calcular el resultado. Room te permite escribir cualquier consulta, por lo que también puedes unir tablas. Por otra parte, si la respuesta es un tipo de datos observables, como [Flowable](http://reactivex.io/RxJava/2.x/javadoc/io/reactivex/Flowable.html) o [LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData.html?hl=es-419), Room observa todas las tablas de referencia de la consulta en busca de invalidación.

En el siguiente fragmento de código, se muestra cómo realizar una combinación de tablas para consolidar información entre una tabla que contiene usuarios que toman prestados libros y una tabla que contiene datos sobre libros prestados actualmente:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
        @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")  
       public List<Book> findBooksBorrowedByNameSync(String userName);  
    }

También puedes mostrar objetos POJO desde estas consultas. Por ejemplo, puedes escribir una consulta que cargue a un usuario y el nombre de su mascota de la siguiente manera:

[KOTLIN](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#kotlin)[JAVA](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419#java)

    @Dao  
    public interface MyDao {  
       @Query("SELECT user.name AS userName, pet.name AS petName " +  
              "FROM user, pet " +  
              "WHERE user.id = pet.user\_id")  
       public LiveData<List<UserPet>> loadUserAndPetNames();  
  
       // You can also define this class in a separate file, as long as you add the  
       // "public" access modifier.  
       static class UserPet {  
           public String userName;  
           public String petName;  
       }  
    }

### **Cómo escribir métodos asincrónicos con corrutinas de Kotlin**

Puedes agregar la palabra clave de Kotlin suspend a tus métodos DAO para hacerlos asíncronos con la funcionalidad de las rutinas de Kotlin. De esta manera, te aseguras de que no se puedan ejecutar en el subproceso principal.

@Dao  
    interface MyDao {  
        @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")  
        suspend fun loadAllUsers(): Array<User>  
    }

**Nota**: Para usar Room con las corrutinas de Kotlin, se requiere Room 2.1.0, Kotlin 1.3.0 y Coroutines 1.0.0 o versiones posteriores.Para obtener más información, consulta [Cómo declarar dependencias](https://developer.android.com/jetpack/androidx/releases/room?hl=es-419#declaring_dependencies).

Esta guía también se aplica a los métodos DAO anotados con [@Transaction](https://developer.android.com/reference/androidx/room/Transaction?hl=es-419). Puedes usar esta función para compilar métodos de bases de datos de suspensión a partir de otros métodos DAO. Estos métodos se ejecutan en una sola transacción de base de datos.

@Dao  
    abstract class UsersDao {  
        @Transaction  
        open suspend fun setLoggedInUser(loggedInUser: User) {  
            deleteUser(loggedInUser)  
            insertUser(loggedInUser)  
        }  
  
        @Query("DELETE FROM users")  
        abstract fun deleteUser(user: User)  
  
        @Insert  
        abstract suspend fun insertUser(user: User)  
    }

# Room

Referencia de la API:  
[androidx.room](https://developer.android.com/reference/androidx/room/package-summary) [androidx.room.migration](https://developer.android.com/reference/androidx/room/migration/package-summary) [androidx.room.testing](https://developer.android.com/reference/androidx/room/testing/package-summary)

La biblioteca de persistencias [Room](https://developer.android.com/training/data-storage/room/index.html) brinda una capa de abstracción para SQLite que permite acceder a la base de datos sin problemas y, al mismo tiempo, aprovechar toda la potencia de SQLite.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Actualización más reciente | Versión estable actual | Próxima versión potencial | Versión Beta | Versión Alfa |
| 20 de noviembre de 2019 | [2.2.2](https://developer.android.com/jetpack/androidx/releases/room#2.2.2) | - | - | - |

## **Cómo declarar dependencias**

Para agregar una dependencia en Room, debes agregar el repositorio Maven de Google a tu proyecto. Lee el [repositorio Maven de Google](https://developer.android.com/studio/build/dependencies#google-maven) para obtener más información.

Las dependencias de Room incluyen [pruebas de migraciones de Room](https://developer.android.com/training/data-storage/room.html#db-migration-testing) y [Room RxJava](https://developer.android.com/training/data-storage/room/accessing-data.html#query-rxjava).

Agrega las dependencias de los artefactos que necesites en el archivo build.gradle de tu app o módulo:

    dependencies {  
      def room\_version = "2.2.2"  
  
      implementation "androidx.room:room-runtime:$room\_version"  
      annotationProcessor "androidx.room:room-compiler:$room\_version" // For Kotlin use kapt instead of annotationProcessor  
  
      // optional - Kotlin Extensions and Coroutines support for Room  
      implementation "androidx.room:room-ktx:$room\_version"  
  
      // optional - RxJava support for Room  
      implementation "androidx.room:room-rxjava2:$room\_version"  
  
      // optional - Guava support for Room, including Optional and ListenableFuture  
      implementation "androidx.room:room-guava:$room\_version"  
  
      // Test helpers  
      testImplementation "androidx.room:room-testing:$room\_version"  
    }

**Nota:** En el caso de las apps basadas en Kotlin, asegúrate de usar **kapt**, en lugar de **annotationProcessor**. También debes agregar el complemento **kotlin-kapt**.

Para obtener información acerca del uso de extensiones de Kotlin, consulta la [documentación de ktx](https://developer.android.com/kotlin/ktx).

Para obtener más información sobre las dependencias, consulta [Cómo agregar dependencias de compilación](https://developer.android.com/studio/build/dependencies.html).

## **Cómo configurar las opciones del compilador**

Room tiene las siguientes opciones de procesador de anotaciones:

* room.schemaLocation: Configura y permite exportar esquemas de bases de datos a archivos JSON en el directorio determinado. Para obtener más información, consulta [Migraciones de Room](https://developer.android.com/training/data-storage/room/migrating-db-versions#export-schema).
* room.incremental: Habilita el procesador de anotaciones incrementales de Gradle.
* room.expandProjection: Configura Room para reescribir búsquedas, de modo que se expanda su proyección superior destacada para contener solo las columnas definidas en el tipo de datos que se muestra del método DAO.

En el siguiente fragmento de código, se muestra un ejemplo de cómo se pueden configurar estas opciones:

    android {  
        ...  
        defaultConfig {  
            ...  
            javaCompileOptions {  
                annotationProcessorOptions {  
                    arguments = [  
                        "room.schemaLocation":"$projectDir/schemas".toString(),  
                        "room.incremental":"true",  
                        "room.expandProjection":"true"]  
                }  
            }  
        }  
    }

## **Versión 2.2.2**

### **Versión 2.2.2**

20 de noviembre de 2019

Lanzamiento de androidx.room:room-\*:2.2.2. [La versión 2.2.2 contiene estas confirmaciones](https://android.googlesource.com/platform/frameworks/support/+log/33f1d55259b216abd7f9f5a8bcb2a363f8a0070c..a04145cf715ff88c1aa0dd36f3447be35a5c3e62/room).

**Correcciones de errores**

* Se corrigió un error en el que recopilar una relación de uno a uno con más de 999 filas provocaba que Room muestre elementos relacionados nulos. ([b/143105450](http://issuetracker.google.com/143105450))

## **Versión 2.2.1**

### **Versión 2.2.1**

23 de octubre de 2019

Lanzamiento de androidx.room:room-\*:2.2.1. [La versión 2.2.1 contiene estas confirmaciones](https://android.googlesource.com/platform/frameworks/support/+log/31d127394de2acb29cd81204796b413f5a1db8b2..33f1d55259b216abd7f9f5a8bcb2a363f8a0070c/room).

**Correcciones de errores**

* Se corrigió un error por el que Room intentaba incorrectamente advertir sobre CURSOR\_MISMATCH con la opción de compilador expandProjection activada. ([b/140759491](http://issuetracker.google.com/140759491))
* Se agregó un mecanismo de reintento para administrar la biblioteca nativa que falta utilizado para verificar búsquedas durante el tiempo de compilación.

## **Versión 2.2.0**

### **Versión 2.2.0**

9 de octubre de 2019

Lanzamiento de androidx.room:room-\*:2.2.0. [La versión 2.2.0 contiene estas confirmaciones](https://android.googlesource.com/platform/frameworks/support/+log/aa36228c061a81036f6b98fd4fe2e0814415303d..31d127394de2acb29cd81204796b413f5a1db8b2/room).

**Cambios importantes desde la versión 2.1.0**

* **Base de datos preempaquetada**: Hay dos nuevas API disponibles en RoomDatabase.Builder para crear un RoomDatabase si ya hay un archivo de base de datos propagado. createFromAsset() sirve cuando el archivo de la base de datos propagado previamente está en la carpeta de elementos del APK, mientras que createFromFile() sirve cuando el archivo está en una ubicación arbitraria. El uso de estas API cambia el comportamiento de las migraciones destructivas de manera que, durante una migración alternativa, Room intentará volver a copiar la base de datos prepropagada, si está disponible. De lo contrario, solo resguardará y volverá a crear todas las tablas. [b/62185732](https://issuetracker.google.com/issues/62185732)
* **Valores predeterminados del esquema**: @ColumnInfo ahora tiene una nueva propiedad defaultValue que se puede usar para especificar el valor predeterminado de una columna. Los valores predeterminados son parte de un esquema de base de datos y, si se lo especifica, se validarán durante las migraciones. [b/64088772](https://issuetracker.google.com/issues/64088772)
* **Relaciones de varios a varios**: @Relation ahora tiene una nueva propiedad associateBy, que toma una nueva anotación @Junction, usada para declarar una relación que debe satisfacerse mediante una tabla de conjunción (también conocida como tabla de unión). [b/69201917](https://issuetracker.google.com/issues/69201917)
* **Relaciones de uno a uno**: Se levantó la restricción en los campos POJO con anotaciones @Relation para que sean de tipo List o Set, lo que permite representar relaciones de valor único con eficacia. [b/62905145](https://issuetracker.google.com/issues/69201917)
* **Entidad de destino**: Las anotaciones de DAO @Insert, @Update y @Delete ahora tienen una nueva propiedad targetEntity, que permite especificar la tabla de destino sobre la que tiene efecto el método DAO. Esto permite que los parámetros de esos métodos DAO sean POJO arbitrarios que se interpretarán como entidades parciales. En la práctica, esto permite inserciones, eliminaciones y actualizaciones parciales. [b/127549506](https://issuetracker.google.com/issues/127549506)
* **Flujo de corrutinas**: Los métodos DAO @Query ahora pueden ser Flow<T> de tipo de datos que se muestra. El flujo que se muestre reemitirá un nuevo conjunto de valores si se invalidan las tablas de observación en la búsqueda. Declarar una función DAO con un tipo de datos que se muestra Channel<T> es un error. En cambio, Room te incentiva a usar Flow y, luego, utilizar las funciones cercanas para convertir el Flow en un Channel. [b/130428884](https://issuetracker.google.com/issues/130428884)
* **Procesador de anotaciones incrementales de Gradle**: Room ahora es un procesador de anotaciones de aislamiento de Gradle y la incrementabilidad se puede habilitar mediante la opción room.incremental del procesador. Para obtener más información, consulta [Opciones del compilador de Room](https://developer.android.com/jetpack/androidx/releases/room.md#compiler-options). Si encuentras algún problema, [infórmanos](https://issuetracker.google.com/issues/new?component=413107&template=1096568). Planeamos habilitar la incrementabilidad predeterminada en una versión futura y estable. [b/112110217](https://issuetracker.google.com/issues/112110217)
* **Proyecciones desplegables**: Se agregó una nueva opción experimental de compilador room.expandProjection, que hace que Room reescriba una búsqueda con una proyección destacada para contener solo las columnas en el tipo de datos que se muestra POJO. Por ejemplo, para un método DAO con @Query("SELECT \* FROM Song") que muestra un POJO llamado SongIdAndTitle con solo dos campos. Luego, Room reescribe la búsqueda a SELECT id, title FROM Song, de modo de obtener el conjunto mínimo de columnas para satisfacer el tipo de datos que se muestra. Básicamente, esto elimina la advertencia CURSOR\_MISMATCH que se presenta cuando la búsqueda muestra columnas adicionales que no coinciden con ningún campo en el tipo de POJO que se muestra.

### **Versión 2.2.0-rc01**

5 de septiembre de 2019

Lanzamiento de androidx.room:room:2.2.0-rc01. Para conocer las confirmaciones incluidas en esta versión, consulta [este artículo](https://android.googlesource.com/platform%2Fframeworks%2Fsupport/+log/603df7ced432238f83d1a244f10bfc1968d466ff..aa36228c061a81036f6b98fd4fe2e0814415303d/room).

No hay cambios públicos desde Room 2.2.0-beta01.

### **Versión 2.2.0-beta01**

22 de agosto de 2019

Lanzamiento de androidx.room:room-\*:2.2.0-beta01. Para conocer las confirmaciones incluidas en esta versión, consulta [este artículo](https://android.googlesource.com/platform%2Fframeworks%2Fsupport/+log/45e343f257a0efd3f0f69546ee9923a9ceb0a8cb..603df7ced432238f83d1a244f10bfc1968d466ff/room).

**Correcciones de errores**

* Se corrigió un error relacionado con una búsqueda de flujo de corrutina que dejaba de reemitir nuevos valores después de cierto tiempo. ([b/139175786](https://issuetracker.google.com/issues/139175786))
* Se corrigió un error por el que Room no aceptaba un código hash de esquema heredado al abrir una base de datos que no se había migrado desde Room 1.0, lo que provocaba una falla de tiempo de ejecución debido a un esquema no válido. ([b/139306173](https://issuetracker.google.com/issues/139306173))

### **Versión 2.2.0-alpha02**

7 de agosto de 2019

Lanzamiento de androidx.room:room-\*:2.2.0-alpha02. Para conocer las confirmaciones incluidas en esta versión, consulta [este artículo](https://android.googlesource.com/platform%2Fframeworks%2Fsupport/+log/f3fa99365aa7db9e882cb3f91fc486b1d9445814..45e343f257a0efd3f0f69546ee9923a9ceb0a8cb/room).

**Nuevas funciones**

* **Flujo de corrutinas**: Los métodos DAO @Query ahora pueden ser Flow<T> de tipo de datos que se muestra. El flujo que se muestre reemitirá un nuevo conjunto de valores si se invalidan las tablas de observación en la búsqueda. Declarar una función DAO con un tipo de datos que se muestra Channel<T> es un error. En cambio, Room te incentiva a usar Flow y, luego, utilizar las funciones cercanas para convertir el Flow en un Channel. [b/130428884](https://issuetracker.google.com/issues/130428884)
* **Proyecciones desplegables**: Se agregó una nueva opción experimental de compilador room.expandProjection, que hace que Room reescriba una búsqueda con una proyección destacada para contener solo las columnas en el tipo de datos que se muestra POJO. Por ejemplo, para un método DAO con @Query("SELECT \* FROM Song") que muestra un POJO llamado SongIdAndTitle con solo dos campos. Luego, Room reescribe la búsqueda a SELECT id, title FROM Song, de modo de obtener el conjunto mínimo de columnas para satisfacer el tipo de datos que se muestra. Básicamente, esto elimina la advertencia CURSOR\_MISMATCH que se presenta cuando la búsqueda muestra columnas adicionales que no coinciden con ningún campo en el tipo de POJO que se muestra.
* onDestructiveMigrate es una API de devolución de llamada nueva que se agregó a RoomDatabase.Callback para cuando Room migra destructivamente una base de datos. [b/79962330](https://issuetracker.google.com/issues/79962330)

**Correcciones de errores**

* Se corrigió un error relacionado con Room por el que se generaba un código incorrecto usando un método como configurador de campo cuando el campo estaba protegido. [b/136194628](https://issuetracker.google.com/issues/136194628)
* Se corrigió un error que hacía que el InvalidationTracker lanzara una NPE en un segundo proceso cuando se habilitaba la invalidación de varias instancias y se cancelaba el servicio de invalidación. [b/137454915](https://issuetracker.google.com/issues/137454915)
* Se corrigió un error por el cual Room no identificaba correctamente el tipo de datos que se muestra de una función de suspensión heredada con anotación @RawQuery. [b/137878827](https://issuetracker.google.com/issues/137878827)
* Se actualizó el código generado para @Relation cuando la clave relacionada es de tipo BLOB a fin de usar un ByteBuffer que sea comparable. [b/137881998](https://issuetracker.google.com/issues/137881998)
* Se corrigió un error por el que Room reclamaba la falta de establecedores en los POJO utilizados como parámetros de entidad parciales de @Insert, @Update, y @Delete. [b/138664463](https://issuetracker.google.com/issues/138664463)
* Se corrigió un error por el que Room reclamaba la falta de capturadores y establecedores para una columna ignorada a través de @Entity cuando se utilizaba la clase de entidad en algunos métodos DAO. [b/138238182](https://issuetracker.google.com/issues/138238182)
* Se corrigió un error por el que Room no convertía correctamente los argumentos de vinculación con nombre en argumentos posicionales, lo que causaba una excepción de tiempo de ejecución cuando se ejecutaba una búsqueda con parámetros reutilizados. [b/137254857](https://issuetracker.google.com/issues/137254857)

### **Versión 2.2.0-alpha01**

10 de julio de 2019

**Nuevas funciones**

* **Base de datos preempaquetada**: Hay dos nuevas API disponibles en RoomDatabase.Builder para crear un RoomDatabase si ya hay un archivo de base de datos propagado. createFromAsset() sirve cuando el archivo de la base de datos propagado previamente está en la carpeta de elementos del APK, mientras que createFromFile() sirve cuando el archivo está en una ubicación arbitraria. El uso de estas API cambia el comportamiento de las migraciones destructivas de manera que, durante una migración alternativa, Room intentará volver a copiar la base de datos prepropagada, si está disponible. De lo contrario, solo resguardará y volverá a crear todas las tablas. [b/62185732](https://issuetracker.google.com/issues/62185732)
* **Valores predeterminados del esquema**: @ColumnInfo ahora tiene una nueva propiedad defaultValue que se puede usar para especificar el valor predeterminado de una columna. Los valores predeterminados son parte de un esquema de base de datos y, si se lo especifica, se validarán durante las migraciones. [b/64088772](https://issuetracker.google.com/issues/64088772)

**Nota:** Si el esquema de tu base de datos ya tiene valores predeterminados, como los que se agregaron mediante **ALTER TABLE x ADD COLUMN y INTEGER NOTNULL DEFAULT z**, y decides definir los valores predeterminados a través de **@ColumnInfo** en las mismas columnas, es posible que debas proporcionar una migración para validar los valores predeterminados no contabilizados. Para obtener más información, consulta [Migraciones de Room](https://developer.android.com/training/data-storage/room/migrating-db-versions.md#handle-default-values-migrations).

* **Relaciones de varios a varios**: @Relation ahora tiene una nueva propiedad associateBy, que toma una nueva anotación @Junction, usada para declarar una relación que debe satisfacerse mediante una tabla de conjunción (también conocida como tabla de unión). [b/69201917](https://issuetracker.google.com/issues/69201917)
* **Relaciones de uno a uno**: Se levantó la restricción en los campos POJO con anotaciones @Relation para que sean de tipo List o Set, lo que permite representar relaciones de valor único con eficacia. [b/62905145](https://issuetracker.google.com/issues/69201917)
* **Entidad de destino**: Las anotaciones de DAO @Insert, @Update y @Delete ahora tienen una nueva propiedad targetEntity, que permite especificar la tabla de destino sobre la que tiene efecto el método DAO. Esto permite que los parámetros de esos métodos DAO sean POJO arbitrarios que se interpretarán como entidades parciales. En la práctica, esto permite inserciones, eliminaciones y actualizaciones parciales. [b/127549506](https://issuetracker.google.com/issues/127549506)
* **Procesador de anotaciones incrementales de Gradle**: Room ahora es un procesador de anotaciones de aislamiento de Gradle y la incrementabilidad se puede habilitar mediante la opción room.incremental del procesador. Para obtener más información, consulta [Opciones del compilador de Room](https://developer.android.com/jetpack/androidx/releases/room.md#compiler-options). Si encuentras algún problema, [infórmanos](https://issuetracker.google.com/issues/new?component=413107&template=1096568). Planeamos habilitar la incrementabilidad predeterminada en una versión futura y estable. [b/112110217](https://issuetracker.google.com/issues/112110217)

**Correcciones de errores**

* Room ya no propagará EmptySetResultException al controlador de errores globales cuando se haya borrado la secuencia Rx de una búsqueda antes de que se complete la búsqueda. [b/130257475](https://issuetracker.google.com/issues/130257475)
* Se corrigió un error por el que Room mostraba un mensaje de error incorrecto cuando una función de suspensión de DAO con anotación @RawQuery no tenía un tipo de datos que se muestra. [b/134303897](https://issuetracker.google.com/issues/134303897)
* Room ya no generará adaptadores de DAO con tipos sin formato. [b/135747255](https://issuetracker.google.com/issues/135747255)

## **Versión 2.1.0**

### **Versión 2.1.0**

13 junio de 2019

Lanzamiento de Room 2.1.0 sin modificaciones desde 2.1.0-rc01. Para conocer las confirmaciones incluidas en la versión, consulta [este artículo](https://android.googlesource.com/platform/frameworks/support/+log/c335f0223fafd3f8ce8658a472bc249d25992ff1..a0ad74d00bf98712b6d07b70214cac23154ff49c/room).

**Cambios importantes desde 2.0.0**

* **FTS**: Ahora Room admite entidades con una tabla de asignación [FTS3 o FTS4](https://www.sqlite.org/fts3.html). Ahora las clases con anotaciones @Entity también pueden tener anotaciones @Fts3 o @Fts4 a fin de declarar una clase con una tabla de asignación de búsqueda de texto completo. Para lograr una mayor personalización, hay opciones de FTS disponibles a través de los métodos de anotación.
* **Vistas**: Ahora Room permite declarar una clase como una búsqueda almacenada, también conocida como [vista](https://www.sqlite.org/lang_createview.html), mediante la anotación @DatabaseView.
* **Corrutinas**: Ahora los métodos DAO pueden ser funciones de suspensión. Incluye room-ktx en tus dependencias para aprovechar esta función. El artefacto ktx también proporciona la función de extensión RoomDatabase.withTransaction para realizar transacciones de bases de datos dentro de una corrutina.
* **Valor automático**: Ahora Room permite declarar clases con anotaciones [AutoValue](https://github.com/google/auto/blob/master/value/userguide/index.md) como entidades y POJO. Ahora puedes declarar las anotaciones @PrimaryKey, @ColumnInfo, @Embedded y @Relation de Room en métodos abstractos de la clase con anotaciones de valor. Ten en cuenta que estas anotaciones también deben ir acompañadas de @CopyAnnotations para que Room pueda comprenderlas correctamente.
* **Compatibilidad con datos asíncronos adicionales**: Los métodos DAO con anotaciones @Insert, @Delete o @Update, junto con @Query que contengan instrucciones INSERT, DELETE o UPDATE, ahora admiten tipos de resultado Rx Completable, Single, Maybe, y el tipo de datos que se muestra ListenableFuture de Guava, y pueden ser funciones de suspensión.
* enableMultiInstanceInvalidation es una API nueva en RoomDatabase.Builder que sirve para habilitar la invalidación en varias instancias de RoomDatabase con el mismo archivo de base de datos.
* fallbackToDestructiveMigrationOnDowngrade es una API nueva en RoomDatabase.Builder que sirve para recrear automáticamente la base de datos si se cambia a una versión inferior.
* ignoredColumns es una API nueva en la anotación @Entity que se puede usar para crear una lista de campos ignorados según su nombre.
* Room usará correctamente el constructor primario de Kotlin en las clases de datos, lo que evitará la necesidad de declarar las propiedades como vars.

### **Versión 2.1.0-rc01**

29 de mayo de 2019

**Correcciones de errores**

* Se corrigió un error de inicialización de Room que posiblemente se producía debido a una configuración temp\_store ya configurada. [b/132602198](https://issuetracker.google.com/issues/132602198)
* Se corrigió una advertencia de uso de comillas dobles para usuarios con SQLite 3.27.0 o versiones posteriores. [b/131712640](https://issuetracker.google.com/issues/131712640)
* Se corrigió un error en el que InvalidationTracker causaba una falla cuando se realizaban varias verificaciones de invalidación en paralelo. [b/133457594](https://issuetracker.google.com/issues/133457594)

### **Versión 2.1.0-beta01**

7 de mayo de 2019

Lanzamiento de androidx.room 2.1.0-beta01 sin modificaciones desde 2.1.0-alpha07. Para conocer las confirmaciones incluidas en esta versión, consulta [este artículo](https://android.googlesource.com/platform/frameworks/support/+log/9a00dbb5a21ad46c78b5ce525c761542f95d7cc3..92174716bb8c7d3088badc0b72b4e83dc8bf2c1b/room).

### **Versión 2.1.0-alpha07**

25 de abril de 2019

**Cambios de comportamiento/API**

* Se modificó la función de extensión RoomDatabase.withTransaction de modo que ya no tome un bloqueo de funciones con un CoroutineScope como receptor. De esta manera, se evita omitir el wrapper coroutineScope { } adicional necesario para ejecutar las acciones en el bloqueo de transacciones de manera simultánea.

**Correcciones de errores**

* Se corrigió un error por el que Room no podía encontrar una coincidencia con un TypeConverter para una función de DAO de Kotlin que contenía un parámetro de tipo Collection. [b/122066791](https://issuetracker.google.com/issues/122066791)

### **Versión 2.1.0-alpha06**

22 de marzo de 2019

**Cambios de comportamiento/API**

* Ahora se serializan las búsquedas de transacciones asíncronas, de modo que Room no usará más de un subproceso para ejecutar transacciones de la base de datos. Se agregó RoomDatabase.Builder.setTransactionExecutor(Executor) para permitir la configuración del ejecutor que se debe utilizar en las transacciones.
* RoomDatabase.runInTransaction(Callable) ya no unirá las excepciones verificadas en RuntimeExceptions. [b/128623748](https://issuetracker.google.com/issues/128623748)

**Correcciones de errores**

* Se corrigió un error en el que el seguimiento de invalidaciones dejaba de observar una tabla de contenido si se agregaban observadores tanto para la tabla de contenido como para una tabla FTS de contenido externo. [b/128508917](https://issuetracker.google.com/issues/128508917)
* Se actualizó la gramática de SQLite de Room para que coincida con SQLite 3.24.0. [b/110883668](https://issuetracker.google.com/issues/110883668)

### **Versión 2.1.0-alpha05**

13 de marzo de 2019

**Nuevas funciones**

* La función de extensión RoomDatabase.withTransaction te permite realizar transacciones de bases de datos de forma segura dentro de una corrutina. Las funciones de extensiones de Room, junto con la compatibilidad con corrutinas, están disponibles en el artefacto room-ktx.
* Los métodos DAO no abstractos con anotaciones @Transaction ahora pueden ser funciones de suspensión. [b/120241587](https://issuetracker.google.com/issues/120241587)

**Cambios de comportamiento/API**

* Se cambió el nombre del artefacto room-coroutines por room-ktx para seguir la misma nomenclatura que otros artefactos androidx.
* Se dieron de baja beginTransaction, setTransactionSuccessful y endTransaction en RoomDatabase, y se los reemplazó por runInTransaction y la función de extensión room-ktx withTransaction.

**Correcciones de errores**

* Se corrigió un error por el que se soltaban los argumentos del tokenizador si el que se usaba era SIMPLE. [b/125427014](https://issuetracker.google.com/issues/125427014)
* Se corrigió un error por el que Room no identificaba correctamente las funciones de suspensión con parámetros cuyo tipo era una clase interna. [b/123767877](https://issuetracker.google.com/issues/123767877)
* Se corrigió un error por el que el método @Query diferido de DAO con instrucciones INSERT, UPDATE o DELETE preparaba con anticipación la búsqueda en el subproceso principal. [b/123695593](https://issuetracker.google.com/issues/123695593)
* Se corrigieron varios errores por los que Room generaba códigos incorrectos para algunas funciones de suspensión. [b/123466702](https://issuetracker.google.com/issues/123466702) y [b/123457323](https://issuetracker.google.com/issues/123466702)
* Se corrigió un error por el que no se suprimía correctamente el uso obsoleto de los métodos en el código generado. [b/117602586](https://issuetracker.google.com/issues/117602586)
* Se actualizó la dependencia de Room de androidx.sqlite a 1.0.2, que contiene correcciones a fin de controlar de forma correcta las bases de datos dañadas. [b/124476912](https://issuetracker.google.com/issues/124476912)

**Problemas conocidos**

* Room 2.1.0-alpha05 depende del artefacto kotlinx-metadata-jvm que no está disponible actualmente en Maven Central ([KT-27991](https://youtrack.jetbrains.com/issue/KT-27991)). Puedes resolver esta dependencia si agregas maven { url "https://kotlin.bintray.com/kotlinx/" } a los repositorios de tu proyecto.

### **Versión 2.1.0-alpha04**

25 de enero de 2019

**Nuevas funciones**

* Los métodos DAO con anotaciones @Query que contienen instrucciones INSERT, UPDATE o DELETE ahora pueden mostrar tipos asíncronos Single, Mayble, Completable y ListenableFuture. Además, también pueden ser funciones de suspensión. [b/120227284](https://issuetracker.google.com/120227284)

**Cambios de comportamiento/API**

* Room ahora mostrará un error si un método DAO no abstracto con anotaciones @Transaction muestra un tipo asíncrono como Single, Mayble, Completable, LiveData o ListenableFuture. Dado que las transacciones están confinadas en subprocesos, por el momento, es imposible que Room comience y finalice una transacción en torno a una función que puede realizar búsquedas en diferentes subprocesos. [b/120109336](https://issuetracker.google.com/120109336)
* OnConflictStrategy.FAIL y OnConflictStrategy.ROLLBACK ahora son @Deprecated ya que no se comportan según lo esperado con las actuales vinculaciones SQLite de Android. [b/117266738](https://issuetracker.google.com/117266738)

**Correcciones de errores**

* Se corrigió un error por el que Room no usaba correctamente el TypeConverter de un tipo de datos que se muestra si el método DAO era una función de suspensión. [b/122988159](https://issuetracker.google.com/122988159)
* Se corrigió un error por el que Room identificaba de manera incorrecta las funciones de suspensión heredadas como de no suspensión. [b/122902595](https://issuetracker.google.com/122902595)
* Se corrigió un error por el que Room generaba códigos incorrectos cuando un campo @Embedded estaba en una clase superior y se usaba en varias clases secundarias. [b/121099048](https://issuetracker.google.com/121099048)
* Se solucionó un problema por el que se bloqueaba la base de datos al invocar funciones de suspensión de DAO entre beginTransaction() y endTransaction(). [b/120854786](https://issuetracker.google.com/120854786)

### **Versión 2.1.0-alpha03**

4 de diciembre de 2018

**Cambios en la API**

* Ahora el FTS tokenizer en @Fts3/@Fts4 lleva una cadena en lugar de una enumeración. Este cambio permite que Room use tokenizadores personalizados. Los tokenizadores integrados todavía están definidos en FtsOptions como constantes de la cadena. [b/119234881](https://issuetracker.google.com/119234881)

**Nuevas funciones**

* **Corrutinas**: Ahora los métodos DAO pueden ser funciones de suspensión. Para admitir las funciones de suspensión en Room, se lanzó un nuevo artefacto, room-coroutines. [b/69474692](https://issuetracker.google.com/69474692)
* Los métodos DAO con anotaciones @Insert, @Delete o @Update ahora admiten ListenableFuture como tipo de datos que se muestra. [b/119418331](https://issuetracker.google.com/119418331)

**Correcciones de errores**

* Se solucionó un error por el que Room intentaba incorrectamente encontrar un constructor con columnas en la propiedad ignoredColumns de @Entity. [b/119830714](https://issuetracker.google.com/119830714)
* Se solucionó un error en el que Room no identificaba parámetros del método DAO como finales en su implementación generada. [b/118015483](https://issuetracker.google.com/118015483)
* Se solucionó un error en el que el procesador de Room fallaba al informar un error en una búsqueda con símbolos especiales. [b/119520136](https://issuetracker.google.com/119520136)
* Se corrigió un error por el que Room rechazaba varias implementaciones de Collection como argumentos de una expresión IN. [b/119884035](https://issuetracker.google.com/119884035)
* Se corrigió un error en el que los LiveData mostrados por Room recibían residuos recopilados cuando se los observaba de manera indefinida, lo que ocasionaba que ya no emitieran datos nuevos. [b/74477406](https://issuetracker.google.com/74477406)
* Se actualizó el bloqueo de cierre de RoomDatabase a fin de reducir la competencia de bloqueo. [b/117900450](https://issuetracker.google.com/117900450)

### **Versión 2.1.0-alpha02**

30 de octubre de 2018

**Nuevas funciones**

* Se agregó compatibilidad para hacer referencia a @DatabaseView en @Relation. [b/117680932](https://issuetracker.google.com/117680932)

**Correcciones de errores**

* Se corrigió un error en el que Room realizaba operaciones de I/O en el disco en el proceso principal al suscribirse y desechar un tipo de datos que se muestra Rx [b/117201279](https://issuetracker.google.com/117201279)
* Se corrigió un error en el que Room no encontraba un convertidor de tipo apropiado para un campo en una clase de entidad de Kotlin. [b/111404868](https://issuetracker.google.com/111404868)
* Se corrigió un error en el que Room generaba un código incorrecto para una implementación de interfaz DAO que incluye un método predeterminado de Kotlin sin argumentos. [b/117527454](https://issuetracker.google.com/117527454)
* Se actualizó el analizador de gramática de SQLite de Room y se corrigió un error de rendimiento que prolongaba los tiempos de compilación. [b/117401230](https://issuetracker.google.com/117401230)

### **Versión 2.1.0-alpha01**

8 de octubre de 2018

**Nuevas funciones**

* **FTS**: Ahora Room admite entidades con una tabla de mapeo [FTS3 o FTS4](https://www.sqlite.org/fts3.html). Ahora las clases con anotaciones @Entity también pueden tener anotaciones @Fts3 o @Fts4 a fin de declarar una clase con una tabla de asignación de búsqueda en el texto completo. Las opciones de FTS para mayor personalización están disponibles mediante los métodos de la anotación. [b/62356416](https://issuetracker.google.com/62356416)
* **Vistas**: Ahora Room permite declarar una clase como una búsqueda almacenada, también conocida como [vista](https://www.sqlite.org/lang_createview.html) mediante la anotación @DatabaseView. [b/67033276](https://issuetracker.google.com/67033276)
* **Valor automático**: Ahora Room permite declarar clases con anotaciones [AutoValue](https://github.com/google/auto/blob/master/value/userguide/index.md) como entidades y POJO. Ahora puedes declarar las anotaciones @PrimaryKey, @ColumnInfo, @Embedded y @Relation de Room en métodos abstractos de la clase con anotaciones de valor. Ten en cuenta que estas anotaciones deben estar acompañadas de @CopyAnnotations para que Room pueda entenderlas correctamente. [b/62408420](https://issuetracker.google.com/62408420)
* **Compatibilidad con tipos de resultado Rx adicionales**: Los métodos DAO con anotaciones @Insert, @Delete o @Update ahora admiten tipos de resultado Rx Completable, Single<T> y Maybe<T>. [b/63317956](https://issuetracker.google.com/63317956)
* **Tipos con @Relation que no cambian**: Antes, Room requería que los campos con anotaciones @Relation pudieran configurarse, pero ahora pueden ser parámetros de constructor.
* enableMultiInstanceInvalidation: Es una API nueva en RoomDatabase.Builder que sirve para habilitar la invalidación en varias instancias de RoomDatabase con el mismo archivo de base de datos. Este mecanismo de invalidación de varias instancias también funciona con distintos procesos. [b/62334005](https://issuetracker.google.com/62334005)
* fallbackToDestructiveMigrationOnDowngrade: Es una API nueva en RoomDatabase.Builder que sirve para recrear automáticamente la base de datos si se cambia a una versión inferior. [b/110416954](https://issuetracker.google.com/110416954)
* ignoredColumns: Es una API nueva en la anotación @Entity que se puede usar para crear una lista de campos ignorados según su nombre. Útil para ignorar campos heredados en una entidad. [b/63522075](https://issuetracker.google.com/63522075)

**Cambios de comportamiento/API**

* Ahora mCallback y mDatabase en RoomDatabase están @Deprecated y se quitarán en la próxima versión de Room. [b/76109329](https://issuetracker.google.com/76109329)

**Correcciones de errores**

* Se corrigieron dos errores en los que Room no se recuperaba correctamente de una base de datos dañada o una migración incorrecta durante la inicialización. [b/111504749](https://issuetracker.google.com/111504749) y [b/111519144](https://issuetracker.google.com/111519144)
* Ahora Room usará de forma correcta el constructor principal de Kotlin en clases de datos, lo que evita la necesidad de declarar los campos como vars. [b/105769985](https://issuetracker.google.com/105769985)

## **Versión 2.0.0**

### **Versión 2.0.0**

1 de octubre de 2018

Lanzamiento de androidx.room 2.0.0 sin cambios desde la versión 2.0.0-rc01.

### **Versión 2.0.0-rc01**

20 de septiembre de 2018

Lanzamiento de androidx.room 2.0.0-rc01 sin modificaciones desde la versión 2.0.0-beta01.

### **Versión 2.0.0-beta01**

2 de julio de 2018

**Cambios de comportamiento/API**

* Se agregó RoomDatabase.Builder.setQueryExecutor() para permitir la personalización del lugar en el que se ejecutan las búsquedas.
* Se agregó compatibilidad con RxJava2 Observable.
* Ahora las implementaciones generadas de DAO y Database son definitivas.

**Correcciones de errores**

* Especifica el nombre de clase/campo en el error "no se puede encontrar un captador para el campo". [b/73334503](https://issuetracker.google.com/issues/73334503)
* Se corrigió la compatibilidad de versiones anteriores de RoomOpenHelper con versiones anteriores de Room. [b/110197391](https://issuetracker.google.com/issues/110197391)

## **Dependencias anteriores a AndroidX**

Para las versiones de Room anteriores a AndroidX, incluye estas dependencias:

dependencies {  
        def room\_version = "1.1.1"  
  
        implementation "android.arch.persistence.room:runtime:$room\_version"  
        annotationProcessor "android.arch.persistence.room:compiler:$room\_version" // For Kotlin use kapt instead of annotationProcessor  
  
        // optional - RxJava support for Room  
        implementation "android.arch.persistence.room:rxjava2:$room\_version"  
  
        // optional - Guava support for Room, including Optional and ListenableFuture  
        implementation "android.arch.persistence.room:guava:$room\_version"  
  
        // Test helpers  
        testImplementation "android.arch.persistence.room:testing:$room\_version"  
    }