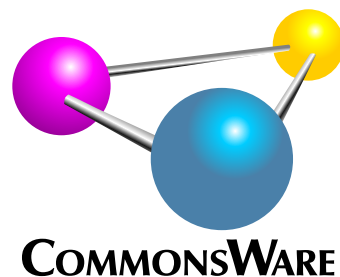


FINAL Version

Elements of Android Room

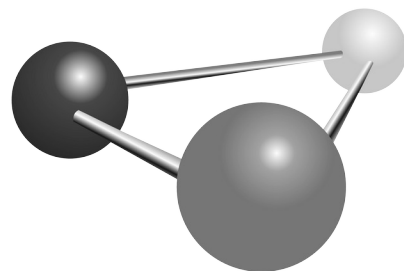


Mark L. Murphy



Elements of Android Room

by Mark L. Murphy



COMMONSWARE

Elements of Android Room
by Mark L. Murphy

Copyright © 2019-2021 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

Printing History:
December 2021: FINAL Version

The CommonsWare name and logo, “Busy Coder’s Guide”, and related trade dress are trademarks of CommonsWare, LLC.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

• Preface	
◦ The Book's Prerequisites	v
◦ Source Code and Its License	vi
◦ Acknowledgments	vi
• Room Basics	
◦ Wrenching Relations Into Objects	1
◦ Room Requirements	2
◦ Room Furnishings	3
◦ Get a Room	8
◦ Testing Room	8
• The Dao of Entities	
◦ Configuring Entities	13
◦ DAOs and Queries	29
◦ Dynamic Queries	37
◦ Other DAO Operations	39
◦ Transactions and Room	42
• Room and Custom Types	
◦ Type Converters	45
◦ Embedded Types	51
• Room and Reactive Frameworks	
◦ Room and the Main Application Thread	55
◦ Room and LiveData	57
◦ Room and Coroutines	59
◦ Room and RxJava	63
◦ Observable Queries	65
◦ Room and ListenableFuture	65
◦ Where Synchronous Room is Safe	66
◦ Being Evil	66
• Relations in Room	
◦ The Classic ORM Approach	67
◦ A History of Threading Mistakes	68
◦ The Room Approach	69
◦ One-to-Many Relations	69
◦ Many-to-Many Relations	75
◦ Room Entities as DTOs	79

• The Support Database API	
◦ “Can’t You See That This is a Facade?”	81
◦ When Will We Use This?	82
◦ Configuring Room’s Database Access	83
• Database Migrations	
◦ What’s a Migration?	87
◦ When Do We Migrate?	88
◦ But First, a Word About Exporting Schemas	88
◦ Writing Migrations	91
◦ Employing Migrations	91
◦ How Room Applies Migrations	93
◦ Testing Migrations	94
• Polymorphic Entities	
◦ Polymorphism With Separate Tables	103
◦ Polymorphism With a Single Table	109
• Default Values and Partial Entities	
◦ Default Values, and the Other Default Values	115
◦ Default Values and Inserts	116
◦ Partial Entities	117
• Room and Full-Text Search	
◦ What Is FTS?	121
◦ Applying FTS to Room	123
◦ Supported MATCH Syntax	132
◦ Migrating to FTS	132
• Room and Conflict Resolution	
◦ Abort	136
◦ Fail	138
◦ Ignore	138
◦ Replace	139
◦ Rollback	140
◦ What Should You Use with Room?	140
• A Room With a View	
◦ Defining a View	144
◦ Registering the View	145
◦ Querying a View	145
◦ OK, Why Bother?	146
• Room and PRAGMAs	
◦ When To Make Changes	149
◦ Example: Turbo Boost Mode	150
• Packaged Databases	
◦ Going Back In Time	153

◦ The Room Mechanics	154
◦ Creating the Database Asset	155
◦ Dealing With Metadata and Upgrades	157
◦ Hybrid Data	157
• Backing Up Your Room	
◦ Backup and Restore. Or, Import and Export.	159
◦ Choosing a Storage Target	160
◦ Thinking About Journal Modes	160
◦ Keeping It Closed	161
◦ Import and Export Mechanics	162
◦ The createFromFile() Alternative	165
• SQLite Clients	
◦ Database Inspector	167
◦ DB Browser for SQLite	172
◦ Flipper	174
• SQLCipher for Android	
◦ Introducing SQLCipher for Android	183
◦ But First, A To-Do Reminder	184
◦ The Basics of SQLCipher for Android	188
◦ The Costs of SQLCipher for Android	190
• SQLCipher and Passphrases	
◦ Generating a Passphrase	193
◦ Collecting a Passphrase	198
◦ Multi-Factor Authentication	206
◦ The Risks of String	207
• Managing SQLCipher	
◦ Backup and Restore	209
◦ Migrating to Encryption	221
• Paged Room Queries	
◦ The Problem: Too Much Data	227
◦ Addressing the UX	228
◦ Enter the Paging Library	229
◦ Paging and Room	230
• Room Across Processes	
◦ Room and Invalidation Tracking	237
◦ Invalidation Tracking and Processes	238
◦ Introducing enableMultiInstanceInvalidation()	238
• Triggers	
◦ Trigger Basics	243
◦ Room and Triggers	244
◦ Triggers the Hard Way	244

- [What's New in Room?](#)
 - Version 2.3.x 249

Preface

Thanks!

Thanks for your interest in Room! Room is Google's solution for a high-level database access API, for your local SQLite databases. As such, Room gets a lot of attention and is reasonably popular.

Thanks also for your broader interest in Android app development! Android is the most widely-used operating system on the planet, so we need to be able to rapidly develop high-quality Android apps. Room can help with that.

And thanks for your interest in this book! Here, you can learn more about how to work with Room, from the basics through more complex scenarios. And, along the way, there may be a joke or two.

The Book's Prerequisites

This book is designed for developers with at least a bit of Android app development experience. If you are fairly new to Android, please consider reading [Elements of Android Jetpack](#), [Exploring Android](#), or both, before continuing with this book.

Also note that this book's examples are written in Kotlin. If you are unfamiliar with Kotlin, you can still learn stuff about Room from this book, though it will be more difficult. You might consider reading [Elements of Kotlin](#), to familiarize yourself with Kotlin.

Source Code and Its License

The source code in this book is licensed under the [Apache 2.0 License](#), in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Acknowledgments

The author would like to thank Daniel Rivera, Yiğit Boyar, and the rest of the developers at Google responsible for Room.

The author would also like to thank Stephen Lombardo, Nick Parker, and the rest of Zetetic, plus Nathan Frietas, Hans-Cristoph Steiner, and the rest of the Guardian Project, for all their work on SQLCipher for Android and general Android app security.

Introductory Room

Room Basics

Google describes Room as providing “an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.”

In other words, Room aims to make your use of SQLite easier, through a lightweight annotation-based implementation of an [object-relational mapping \(ORM\) engine](#).

Wrenching Relations Into Objects

If you have ever worked with a relational database — like SQLite — from an object-oriented language — like Java or Kotlin — undoubtedly you have encountered [the “object-relational impedance mismatch”](#). That is a very fancy way of saying “it’s a pain getting stuff into and out of the database”.

In object-oriented programming, we are used to objects holding references to other objects, forming some sort of object graph. However, traditional SQL-style relational databases work off of tables of primitive data, using foreign keys and join tables to express relationships. Figuring out how to get our classes to map to relational tables is aggravating, and it usually results in a lot of boilerplate code.

Traditional Android development uses SQLiteDatabase for interacting with SQLite. That, in turn, uses Cursor objects to represent the results of queries and ContentValues objects to represent data to be inserted or updated. While Cursor and ContentValues are objects, they are fairly generic, much in the way that a HashMap or ArrayList is generic. In particular, neither Cursor nor ContentValues has any of our business logic. We have to somehow either wrap that around those objects or convert between those objects and some of ours.

That latter approach is what object-relational mapping engines (ORMs) take. A

ROOM BASICS

typical ORM works off of Java/Kotlin code and either generates a suitable database structure or works with you to identify how the classes should map to some existing table structure (e.g., a legacy one that you are stuck with). The ORM usually generates some code for you, and supplies a library, which in combination hide much of the database details from you.

The quintessential Java ORM is [Hibernate](#). However, Hibernate was developed with server-side Java in mind and is not well-suited for slim platforms like Android devices. However, [a vast roster of Android ORMs and similar libraries](#) have been created over the years to try to fill that gap. Some of the more popular ones have been:

- [SQLDelight](#)
- [DBFlow](#)
- [greenDAO](#)
- [OrmLite](#)
- [Sugar ORM](#)

Room also helps with the object-relational impedance mismatch. It is not as deep of an ORM as some of the others, as you will be dealing with SQL a fair bit. However, Room has one huge advantage: it is from Google, and therefore it will be deemed “official” in the eyes of many developers and managers.

While this book is focused on Room, you may wish to explore other ORMs if you are interested in using Java/Kotlin objects but saving the data in SQLite. Room is popular, but it is far from the only option. In particular, if you are interested in Kotlin/Multiplatform for cross-platform development, you will want to look at [SQLDelight](#), so your database operations can also be cross-platform.

Room Requirements

To use Room, you need two dependencies in your module’s `build.gradle` file:

1. The runtime library
2. An annotation processor

In a Kotlin project, those will be:

- `room-ktx`, to pull in the core Room runtime libraries plus some Kotlin-specific extensions

ROOM BASICS

- room-compiler, used with kapt

For example, in [the NoteBasics module](#) of [the book's primary sample project](#), we have a build.gradle file that pulls in those two artifacts:

```
apply plugin: 'com.android.library'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-kapt'

android {
    compileSdkVersion 31

    defaultConfig {
        minSdkVersion 21
        targetSdkVersion 30
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation "androidx.appcompat:appcompat:1.3.1"
    implementation "androidx.core:core-ktx:1.6.0"
    implementation "androidx.constraintlayout:constraintlayout:2.1.1"
    implementation "androidx.room:room-ktx:$room_version"
    kapt "androidx.room:room-compiler:$room_version"

    androidTestImplementation "androidx.test.ext:junit:1.1.3"
    androidTestImplementation "androidx.test:runner:1.4.0"
    androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
    androidTestImplementation "com.natpryce:hamkrest:1.7.0.0"
}
```

(from [NoteBasics/build.gradle](#))

Note that Room has a minSdkVersion requirement of API Level 15 or higher. If you attempt to build with a lower minSdkVersion, you will get a build error. If you try to override Room's minSdkVersion using manifest merger elements, while the project will build, expect Room to crash horribly.

Room Furnishings

Roughly speaking, your use of Room is dominated by three sets of classes:

1. Entities, which are simple classes that model the data you are transferring

ROOM BASICS

- into and out of the database
2. The data access object (DAO), that provides the description of the API that you want for working with certain entities
 3. The database, which ties together all of the entities and DAOs for a single SQLite database

If you have used Square's [Retrofit](#), some of this will seem familiar:

- The DAO is roughly analogous to your Retrofit interface on which you declare your Web service API
- Your entities are the POJOs that you are expecting Gson/Moshi/whatever to create based on the Web service response

The `NoteBasics` module mentioned above has a few classes related to a note-taking application, exercised via instrumented tests.

Entities

In many ORM systems, the entity (or that system's equivalent) is a simple class that you happen to want to store in the database. It usually represents some part of your overall domain model, so a payroll system might have entities representing departments, employees, and paychecks.

With Room, a better description of entities is that they are classes representing:

- the data that you want to store into a table, and
- a typical unit of a result set that you are trying to retrieve from the database

That difference may sound academic. It starts to come into play a bit more when we start thinking about [relations](#).

However, it also more closely matches the way Retrofit maps to Web services. With Retrofit, we are not describing the contents of the Web service's database. Rather, we are describing how we want to work with defined Web service endpoints. Those endpoints have a particular set of content that we can work with, courtesy of whoever developed the Web service. We are simply mapping those to methods and classes, both for input and output. Room is somewhere in between a Retrofit-style "we just take what the Web service gives us" approach and a full ORM-style "we control everything about the database" approach.

From a coding standpoint, an entity is a Java/Kotlin class marked with the `@Entity`

ROOM BASICS

annotation. For example, here is a `NoteEntity` class that serves as a Room entity:

```
package com.commonware.room.notes

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "notes")
data class NoteEntity(
    @PrimaryKey val id: String,
    val title: String,
    val text: String,
    val version: Int
)
```

(from [NoteBasics/src/main/java/com/commonware/room/notes/NoteEntity.kt](#))

There is no particular superclass required for entities, and the expectation is that often they will be simple data classes, as we see here.

The `@Entity` annotation can have properties customizing the behavior of your entity and how Room works with it. In this case, we have a `tableName` property. The default name of the SQLite table is the same as the entity class name, but `tableName` allows you to override that and supply your own table name. Here, we override the table name to be `notes`.

Sometimes, your properties will be marked with annotations describing their roles. In this example, the `id` field has the `@PrimaryKey` annotation, telling Room that this is the unique identifier for this entity. Room will use that to know how to update and delete `Note` objects by their primary key values. In Java, Room also requires that any `@PrimaryKey` field of an object type — like `String` — be annotated with `@NonNull`, as primary keys in SQLite cannot be null. In Kotlin, you can just use a non-nullable type, such as `String`.

We will explore entities in greater detail in [an upcoming chapter](#).

DAO

“Data access object” (DAO) is a fancy way of saying “the API into the data”. The idea is that you have a DAO that provides methods for the database operations that you need: queries, inserts, updates, deletes, and so on.

In Room, the DAO is identified by the `@Dao` annotation, applied to either an

ROOM BASICS

abstract class or an interface. The actual concrete implementation will be code-generated for you by the Room annotation processor.

The primary role of the @Dao-annotated abstract class or interface is to have one or more methods, with their own Room annotations, identifying what you want to do with the database and your entities. This serves the same role as the functions annotated @GET or @POST in a Retrofit interface.

The sample app has a NoteStore that is our DAO:

```
package com.commonware.room.notes

import androidx.room.*

@Dao
interface NoteStore {
    @Query("SELECT * FROM notes")
    fun loadAll(): List<NoteEntity>

    @Insert
    fun insert(note: NoteEntity)

    @Update
    fun update(note: NoteEntity)

    @Delete
    fun delete(vararg notes: NoteEntity)
}
```

(from [NoteBasics/src/main/java/com/commonware/room/notes/NoteStore.kt](#))

Besides the @Dao annotation on the NoteStore interface, we have four functions, each with their own annotations: @Query, @Insert, @Update, and @Delete, each which map to the corresponding database operations.

The loadAll() function has the @Query annotation. Principally, @Query will be used for SQL SELECT statements, where you put the actual SQL in the annotation itself. Here, we are retrieving everything from the notes table.

The remaining three functions use the @Insert, @Update, and @Delete annotations, mapped to functions of the same name. The actual function names do not matter — they could be larry(), curly(), and moe() and work just as well. As you might expect, @Insert inserts an entity into our table, @Update updates an existing table row to reflect the supplied entity's properties, and @Delete deletes table rows

ROOM BASICS

corresponding with the supplied entities' primary keys. In this sample, `insert()` and `update()` each take a single `NoteEntity`, while `delete()` takes a vararg of `NoteEntity`. Room supports either pattern, as well as others, such as a List of `NoteEntity` — choose what fits your needs.

We will explore the DAO in greater detail in [an upcoming chapter](#).

Database

In addition to entities and DAOs, you will have at least one `@Database`-annotated abstract class, extending a `RoomDatabase` base class. This class knits together the database file, the entities, and the DAOs.

In the sample project, we have a `NoteDatabase` serving this role:

```
package com.commonware.room.notes

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(entities = [NoteEntity::class], version = 1)
abstract class NoteDatabase : RoomDatabase() {
    abstract fun notes(): NoteStore
}
```

(from [NoteBasics/src/main/java/com/commonware/room/notes/NoteDatabase.kt](#))

The `@Database` annotation configures the code generation process, including:

- Identifying all of the entity classes that you care about in the entities collection
- Identifying the schema version of the database (as you see with `SQLiteOpenHelper` in conventional Android SQLite development)

Here, we are saying that we have just one entity class (`NoteEntity`), and that this is schema version 1.

You also need abstract functions for each DAO class that return an instance of that class. Here, we have a `notes()` function that returns `NoteStore`.

Get a Room

Our `NoteDatabase` is an abstract class. Somewhere, though, we need to get an instance of it, so we can call `notes()` and be able to start manipulating the database.

To create a `NoteDatabase`, you need a `RoomDatabase.Builder`. There are two functions on the `Room` class for getting one:

- `databaseBuilder()`, and
- `inMemoryDatabaseBuilder()`

`databaseBuilder()` will help you create a database backed by a traditional SQLite database file. `inMemoryDatabaseBuilder()` creates a SQLite database whose contents are only stored in memory — as soon as the database is closed, the memory holding the database contents gets freed.

Both functions take a `Context` and the Java Class object of your `RoomDatabase` subclass as parameters. `databaseBuilder()` also takes the name of the database file to use.

So, we could create a regular, file-backed `NoteDatabase` via:

```
private val db =  
    Room.databaseBuilder(context, NoteDatabase::class.java, "notes.db").build()
```

(where `context` is a suitable `Context`, such as the `Application` singleton)

While there are some configuration methods that can be called on the `RoomDatabase.Builder`, we skip those here, simply calling `build()` to build the `NoteDatabase`, assigning it to the `db` property.

From there, we can:

- Call `notes()` on the `NoteDatabase` to retrieve the `NoteStore` DAO
- Call methods on the `NoteStore` to query, insert, update, or delete `NoteEntity` objects

Testing Room

Once you have a `RoomDatabase` and its associated DAO(s) and entities set up, you

should start testing it.

The good news is that testing Room is not dramatically different than is testing anything else in Android. Room has a few characteristics that make it a bit easier than some things to test, as it turns out.



You can learn more about testing in the "Testing Your Changes" chapter of [Elements of Android Jetpack](#)!

Writing Instrumented Tests

On the whole, writing instrumented tests for Room — where the tests run on an Android device or emulator — is unremarkable. You get an instance of your RoomDatabase subclass and exercise it from there.

So, for example, here is an instrumented test case class to exercise the NoteDatabase:

```
package com.commonware.room.notes

import androidx.room.Room
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import com.natpryce.hamkrest.assertThat
import com.natpryce.hamkrest.equalTo
import com.natpryce.hamkrest.hasSize
import com.natpryce.hamkrest.isEmpty
import org.junit.Test
import org.junit.runner.RunWith
import java.util.*

@RunWith(AndroidJUnit4::class)
class NoteStoreTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        NoteDatabase::class.java
    )
        .build()
    private val underTest = db.notes()

    @Test
    fun insertAndDelete() {
        assertThat(underTest.loadAll(), isEmpty)
```

```
val entity = NoteEntity(
    id = UUID.randomUUID().toString(),
    title = "This is a title",
    text = "This is some text",
    version = 1
)

underTest.insert(entity)

underTest.loadAll().let {
    assertThat(it, hasSize(equalTo(1)))
    assertThat(it[0], equalTo(entity))
}

underTest.delete(entity)

assertThat(underTest.loadAll(), isEmpty)
}

@Test
fun update() {
    val entity = NoteEntity(
        id = UUID.randomUUID().toString(),
        title = "This is a title",
        text = "This is some text",
        version = 1
    )

    underTest.insert(entity)

    val updated = entity.copy(title = "This is new", text = "So is this")

    underTest.update(updated)

    underTest.loadAll().let {
        assertThat(it, hasSize(equalTo(1)))
        assertThat(it[0], equalTo(updated))
    }
}
```

(from [NoteBasics/src/androidTest/java/com/commonsware/room/notes/NoteStoreTest.kt](#))

Using In-Memory Databases

When testing a database, though, one of the challenges is in making those tests

“hermetic”, or self-contained. One test method should not depend upon another test method, and one test method should not affect the results of another test method accidentally. This means that we want to start with a known starting point before each test, and we have to consider how to do that.

One approach — the one taken in the above `NoteStoreTest` class — is to use an in-memory database. The `db` property is initialized using `Room.inMemoryDatabaseBuilder()`, so we get our fast, disposable in-memory database. For a context, we use `InstrumentationRegistry.getInstrumentation().targetContext`, a `Context` for the code being tested. We then set up `underTest` to be the object that we are testing: the `NoteStore` and its functions.

There are two key advantages for using an in-memory database for instrumented testing:

1. It is intrinsically self-contained. Once the `NoteDatabase` is closed (or garbage-collected), its memory is released, and if separate tests use separate `NoteDatabase` instances, one will not affect the other.
2. Reading and writing to and from memory is much faster than is reading and writing to and from disk, so the tests run much faster.

On the other hand, this means that the instrumented tests are useless for performance testing, as (presumably) your production app will actually store its database on disk. You could use Gradle command-line switches, custom build types and `buildConfigField`, or other means to decide when tests are run whether they should use memory or disk.

The Test Functions

Our test functions do things like:

- Creating `NoteEntity` instances, using a UUID for the `id`
- Calling `insert()`, `update()`, and `delete()` to manipulate the table contents
- Calling `loadAll()` to retrieve what is in the table

And, along the way, we use [Hamcrest matchers](#) to confirm that everything is working as we expect.

The Dao of Entities

[In the previous chapter](#), we went through the basic steps for setting up Room:

- Create and annotate your entity classes
- Create, annotate, and define operator functions on your DAO(s)
- Create a subclass of RoomDatabase to tie the entities and DAO(s) together
- Create an instance of that RoomDatabase at some likely point in time,
- Use the RoomDatabase instance to retrieve your DAO and from there work with your entities

However, we only scratched the surface of what can be configured on entities and DAOs. In this chapter, we will start to explore the rest of the configuration for entities and DAOs.

Many of the code snippets shown in this chapter come from the [the MiscSamples module](#) of [the book's primary sample project](#), sample project. This is a library module with a variety of entities and DAOs, all tied into a MiscDatabase, with instrumented tests for each of the entities.

Configuring Entities

The only absolute requirements for a Room entity class is that it be annotated with the `@Entity` annotation and have a field identified as the primary key, typically by way of a `@PrimaryKey` annotation. Anything above and beyond that is optional.

However, there is a fair bit that is “above and beyond that”. Some — though probably not all — of these features will be of interest in larger apps.

Primary Keys

If you have a single field that is the primary key for your entity, using the `@PrimaryKey` annotation is simple and helps you clearly identify that primary key at a later point.

However, you do have some other options.

Auto-Generated Primary Keys

In SQLite, if you have an `INTEGER` column identified as the `PRIMARY KEY`, you can optionally have SQLite assign unique values for that column, by way of the `AUTOINCREMENT` keyword.

In Room, if you have an `Long` property that is your `@PrimaryKey`, you can optionally apply `AUTOINCREMENT` to the corresponding column by adding `autoGenerate=true` to the annotation:

```
package com.commonware.room.misc

import androidx.room.*

@Entity(tableName = "autoGenerate")
data class AutoGenerateEntity(
    @PrimaryKey(autoGenerate = true)
    var id: Long,
    var text: String
) {
    @Dao
    abstract class Store {
        @Query("SELECT * FROM autoGenerate")
        abstract fun loadAll(): List<AutoGenerateEntity>

        @Query("SELECT * FROM autoGenerate WHERE id = :id")
        abstract fun findById(id: Int): AutoGenerateEntity

        fun insert(entity: AutoGenerateEntity): AutoGenerateEntity {
            entity.id = _insert(entity)

            return entity
        }
    }

    @Insert
```

THE DAO OF ENTITIES

```
    abstract fun _insert(entity: AutoGenerateEntity): Long
  }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/AutoGenerateEntity.kt](#))

By default, `autoGenerate` is false. Setting that property to true gives you `AUTOINCREMENT` in the generated `CREATE TABLE` statement:

```
CREATE TABLE IF NOT EXISTS autoGenerate (id INTEGER PRIMARY KEY AUTOINCREMENT NOT
NULL, text TEXT NOT NULL)
```

However, this starts to get complicated in the app. You do not know your primary key until you insert the entity into a database. Your `@Insert`-annotated functions can return a `Long` result, and that will be the primary key for that inserted entity. In the `AutoGenerateEntity` shown above, `_insert()` has the `@Insert` annotation, while `insert()` wraps `_insert()` and sets the inserted entity's `id` to be the `Long` returned by `_insert()`. Hence, `insert()` updates the entity to have its primary key:

```
package com.commonsware.room.misc

import androidx.room.Room
import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4
import com.natpryce.hamkrest.assertion.assertThat
import com.natpryce.hamkrest.equalTo
import com.natpryce.hamkrest.isEmpty

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

@RunWith(AndroidJUnit4::class)
class AutoGenerateEntityTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.autoGenerate()

    @Test
    fun autoGenerate() {
        assertThat(underTest.loadAll(), isEmpty)

        val original = AutoGenerateEntity(id = 0, text = "This will get its own ID")
        val inserted = underTest.insert(original)

        assertTrue(original === inserted)
        assertThat(inserted.id, !equalTo(0L))
    }
}
```

THE DAO OF ENTITIES

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/AutoGenerateEntityTest.kt](#))

This presents “trickle-down” complications — for example, you cannot make the primary key property be `val`, as then you cannot create an instance of an entity that is not yet in the database.

Some of the samples in this book will use a `UUID` instead. While these take up much more room than a simple `Long`, they can be uniquely generated outside of the database. For your production apps, you will need to decide if the headaches surrounding database-generated identifiers are worth their benefits.

Composite Primary Keys

In some cases, you may have a composite primary key, made up of two or more columns in the database. This is particularly true if you are trying to design your entities around an existing database structure, one that used a composite primary key for one of its tables (for whatever reason).

If, logically, those are all part of a single object, you could combine them into a single property, as we will see in [the next chapter](#). However, it may be that they should be individual properties in your entity, but they happen to combine to create the primary key. In that case, you can skip the `@PrimaryKey` annotation and use the `primaryKeys` property of the `@Entity` annotation.

One scenario for this is data versioning, where we are tracking changes to data over time, the way a version control system tracks changes to source code and other files over time. There are several ways of implementing data versioning. One approach has all versions of the same entity in the same table, with a version code attached to the “natural” primary key to identify a specific version of that content. In that case, you could have something like:

```
package com.commonsware.room.misc

import androidx.room.Dao
import androidx.room.Entity
import androidx.room.Insert
import androidx.room.Query

@Entity(tableName = "compositeKey", primaryKeys = ["id", "version"])
data class CompositeKeyEntity(
    val id: String,
    val title: String,
    val text: String = "",
```

THE DAO OF ENTITIES

```
    val version: Int = 1
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM compositeKey")
        fun loadAll(): List<CompositeKeyEntity>

        @Query("SELECT * FROM compositeKey where id = :id AND version = :version")
        fun findByPrimaryKey(id: String, version: Int): CompositeKeyEntity

        @Insert
        fun insert(entity: CompositeKeyEntity)
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/CompositeKeyEntity.kt](#))

Room will then use the PRIMARY KEY keyword in the CREATE TABLE statement to set up the composite primary key:

```
CREATE TABLE IF NOT EXISTS compositeKey (id TEXT NOT NULL, title TEXT NOT NULL, text
TEXT NOT NULL, version INTEGER NOT NULL, PRIMARY KEY(id, version))
```

In our case, we set version to have a default value of 1, so we can create a CompositeKeyEntity with just a string identifier, at least for its initial version:

```
package com.commonsware.room.misc

import android.database.sqlite.SQLiteConstraintException
import androidx.room.Room
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import com.natpryce.hamkrest.assertion.assertThat
import com.natpryce.hamkrest.equalTo
import com.natpryce.hamkrest.isEmpty
import org.junit.Test
import org.junit.runner.RunWith
import java.util.*

@RunWith(AndroidJUnit4::class)
class CompositeKeyEntityTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.compositeKey()
```

```
@Test
fun compositeKey() {
    assertThat(underTest.loadAll(), isEmpty)

    val original = CompositeKeyEntity(
        id = UUID.randomUUID().toString(),
        title = "A composite key entity"
    )

    underTest.insert(original)

    underTest.loadAll().let {
        assertThat(it.size, equalTo(1))
        assertThat(it[0], equalTo(original))
    }

    assertThat(
        underTest.findByPrimaryKey(
            id = original.id,
            version = original.version
        ), equalTo(original)
    )
}

@Test(expected = SQLiteConstraintException::class)
fun duplicateCompositeKey() {
    assertThat(underTest.loadAll(), isEmpty)

    val original = CompositeKeyEntity(
        id = UUID.randomUUID().toString(),
        title = "A composite key entity"
    )

    underTest.insert(original)

    val copy = original.copy(text = "This is different!")

    underTest.insert(copy)
}
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/CompositeKeyEntityTest.kt](https://github.com/commonsware/Commons-DB/blob/master/androidTest/java/com/commonsware/room/misc/CompositeKeyEntityTest.kt))

If we try to insert entities with the same key twice, our @Insert-annotated function will throw a `SQLiteConstraintException`.

Adding Indexes

Your primary key is indexed automatically by SQLite. However, you may wish to set up other indexes for other columns or collections of columns, to speed up queries. To do that, you have two choices:

1. Use the `indices` property on `@Entity`. This property takes a list of nested `Index` annotations, each of which declares an index.
2. Use the `index` property on `@ColumnInfo`, to add an index on a single property.

The latter is simpler; the former handles more complex scenarios (e.g., an index involving multiple properties).

Here, we have an entity with an index on a category property:

```
package com.commonware.room.misc

import androidx.room.*

@Entity(tableName = "indexified")
data class IndexedEntity(
    @PrimaryKey
    val id: String,
    val title: String,
    @ColumnInfo(index = true) val category: String,
    val text: String? = null,
    val version: Int = 1
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM indexified")
        fun loadAll(): List<IndexedEntity>

        @Query("SELECT * FROM indexified where category = :category")
        fun loadAllForCategory(category: String): List<IndexedEntity>

        @Insert
        fun insert(vararg entity: IndexedEntity)
    }
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/IndexedEntity.kt](#))

Room will add the requested index:

THE DAO OF ENTITIES

```
CREATE TABLE IF NOT EXISTS indexified (id TEXT NOT NULL, title TEXT NOT NULL,  
category TEXT NOT NULL, text TEXT, version INTEGER NOT NULL, PRIMARY KEY(id))  
CREATE INDEX IF NOT EXISTS index_indexified_category ON indexified (category)
```

Alternatively, we could have used indices on the @Entity annotation:

```
@Entity(tableName = "indexified", indices = [Index("category")])  
class IndexedEntity(  
    @PrimaryKey  
    val id: String,  
    val title: String,  
    val category: String,  
    val text: String? = null,  
    val version: Int = 1  
)
```

If you have a composite index, consisting of two or more fields, the Index nested annotation takes a comma-delimited list of column names and will generate the composite index.

The index will be used by SQLite automatically if you execute queries that involve the index. The loadAllForCategory() function queries on the indexed category property, and so our index should be used when we call that function:

```
package com.commonware.room.misc  
  
import androidx.room.Room  
import androidx.test.platform.app.InstrumentationRegistry  
import androidx.test.ext.junit.runners.AndroidJUnit4  
import com.natpryce.hamkrest.assertion.assertThat  
import com.natpryce.hamkrest.equalTo  
import com.natpryce.hamkrest.hasSize  
import com.natpryce.hamkrest.isEmpty  
  
import org.junit.Test  
import org.junit.runner.RunWith  
  
import java.util.*  
  
@RunWith(AndroidJUnit4::class)  
class IndexedEntityTest {  
    private val db = Room.inMemoryDatabaseBuilder(  
        InstrumentationRegistry.getInstrumentation().targetContext,  
        MiscDatabase::class.java  
    )  
        .build()
```

```
private val underTest = db.indexed()

@Test
fun queryByCategory() {
    assertThat(underTest.loadAll(), isEmpty)

    val funStuff = IndexedEntity(
        id = UUID.randomUUID().toString(),
        title = "This is fun!",
        category = "fun-stuff",
        text = "words words words"
    )
    val notAsFunStuff = IndexedEntity(
        id = UUID.randomUUID().toString(),
        title = "Gloom, despair, and agony on me",
        category = "un-fun-stuff"
    )

    underTest.insert(funStuff, notAsFunStuff)

    underTest.loadAllForCategory("fun-stuff").let {
        assertThat(it, hasSize(equalTo(1)))
        assert(it[0] == funStuff)
    }
}
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/IndexedEntityTest.kt](#))

If the index should also enforce uniqueness — only one entity can have the indexed value — add `unique = true` to the `Index` annotation. This requires you to assign the column(s) for the index to the `value` property, due to the way annotations work in Kotlin:

```
package com.commonsware.room.misc

import androidx.room.*

@Entity(
    tableName = "uniquelyIndexed",
    indices = [Index(value = ["title"], unique = true)]
)
data class UniqueIndexEntity(
    @PrimaryKey val id: String,
    val title: String,
    val text: String = "",
    val version: Int = 1
)
```


THE DAO OF ENTITIES

```
) {  
    @Dao  
    interface Store {  
        @Query("SELECT * FROM uniquelyIndexed")  
        fun loadAll(): List<UniqueIndexEntity>  
  
        @Insert  
        fun insert(entity: UniqueIndexEntity)  
  
        @Insert(onConflict = OnConflictStrategy.ABORT)  
        fun insertOrAbort(entity: UniqueIndexEntity)  
  
        @Insert(onConflict = OnConflictStrategy.IGNORE)  
        fun insertOrIgnore(entity: UniqueIndexEntity)  
  
        @Insert(onConflict = OnConflictStrategy.REPLACE)  
        fun insertOrReplace(entity: UniqueIndexEntity)  
    }  
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/UniqueIndexEntity.kt](#))

This causes Room to add the UNIQUE keyword to the CREATE INDEX statement:

```
CREATE TABLE IF NOT EXISTS uniquelyIndexed (id TEXT NOT NULL, title TEXT NOT NULL,  
text TEXT NOT NULL, version INTEGER NOT NULL, PRIMARY KEY(id))  
CREATE UNIQUE INDEX IF NOT EXISTS index_uniquelyIndexed_title ON uniquelyIndexed  
(title)
```

While a regular index supports multiple values, a unique index does not, leading once again to a `SQLiteConstraintException` if we try inserting a duplicate:

```
package com.commonsware.room.misc  
  
import android.database.sqlite.SQLiteConstraintException  
import androidx.room.Room  
import androidx.test.platform.app.InstrumentationRegistry  
import androidx.test.ext.junit.runners.AndroidJUnit4  
import com.natpryce.hamkrest.*  
import com.natpryce.hamkrest.assertion.assertThat  
  
import org.junit.Test  
import org.junit.runner.RunWith  
  
import java.util.*  
  
private const val TEST_TITLE = "A Tale of Two Entities"
```

```
@RunWith(AndroidJUnit4::class)
class UniqueIndexEntityTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.uniqueIndex()

    @Test
    fun singleInsert() {
        assertThat(underTest.loadAll(), isEmpty)

        val firstEntity = UniqueIndexEntity(
            id = UUID.randomUUID().toString(),
            title = TEST_TITLE,
            text = "This entity will get inserted successfully")

        underTest.insert(firstEntity)

        assertThat(
            underTest.loadAll(),
            allOf(hasSize(equalTo(1)), hasElement(firstEntity))
        )
    }

    @Test(expected = SQLiteConstraintException::class)
    fun duplicateFailure() {
        singleInsert()

        val secondEntity = UniqueIndexEntity(
            id = UUID.randomUUID().toString(),
            title = TEST_TITLE,
            text = "This entity is doomed")

        underTest.insert(secondEntity)
    }
}
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/UniqueIndexEntityTest.kt](#))

Ignoring Properties

If there are properties in the entity class that should not be persisted, you can annotate them with `@Ignore`:

THE DAO OF ENTITIES

```
package com.commonware.room.misc

import androidx.room.*

@Entity(tableName = "ignoredProperty")
data class IgnoredPropertyEntity(
    @PrimaryKey val id: String,
    val title: String,
    val version: Int = 1
) {
    @Ignore var text: String = ""
    var moreText: String = ""

    @Dao
    interface Store {
        @Query("SELECT * FROM ignoredProperty")
        fun loadAll(): List<IgnoredPropertyEntity>

        @Query("SELECT * FROM ignoredProperty where id = :id")
        fun findByPrimaryKey(id: String): IgnoredPropertyEntity

        @Insert
        fun insert(entity: IgnoredPropertyEntity)
    }
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/IgnoredPropertyEntity.kt](#))

You might think that you could skip that and use other techniques, such as by using a private val property:

```
@Entity(tableName = "ignoredProperty")
data class IgnoredPropertyEntity(
    @PrimaryKey val id: String,
    val title: String,
    val version: Int = 1
) {
    private val text: String = ""
    var moreText: String = ""
}
```

Since the text property is private and has no setter, one could argue that Room might ignore it automatically. Room, instead, generates a build error, as it cannot tell if you want to ignore that property or if you simply forgot to add it properly.

Another option, instead of @Ignore, is to use @Transient, if that annotation fits your

needs better:

```
@Entity(tableName = "ignoredProperty")
data class IgnoredPropertyEntity(
    @PrimaryKey val id: String,
    val title: String,
    val version: Int = 1
) {
    @Transient var text: String = ""
    var moreText: String = ""
}
```

A third option is to use `ignoredColumns`, a property in the `@Entity` annotation, that takes an array of column names that should be ignored:

```
@Entity(
    tableName = "ignoredProperty",
    ignoredColumns = ["text"]
)
data class IgnoredPropertyEntity(
    @PrimaryKey val id: String,
    val title: String,
    val version: Int = 1,
    var text: String = ""
) {
    var moreText: String = ""
}
```

Custom Column Names

By default, Room will generate names for your tables and columns based off of the entity class names and property names. In general, it does a respectable job of this, and so you may just leave them alone. However, you may find that you need to control these names, particularly if you are trying to match an existing database schema (e.g., you are migrating an existing Android app to use Room instead of using SQLite directly). And for table names in particular, setting your own name can simplify some of the SQL that you have to write for `@Query`-annotated functions.

As we have seen, to control the table name, use the `tableName` property on the `@Entity` attribute, and give it a valid SQLite table name. To rename a column, add the `@ColumnInfo` annotation to the property, with a `name` property that provides your desired name for the column:

THE DAO OF ENTITIES

```
package com.commonware.room.misc

import android.database.Cursor
import androidx.room.*

@Entity(tableName = "customColumn")
class CustomColumnNameEntity(
    @PrimaryKey
    val id: String,
    val title: String,
    @ColumnInfo(name = "words") val text: String? = null,
    val version: Int = 1
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM customColumn")
        fun loadAll(): List<CustomColumnNameEntity>

        @Query("SELECT * FROM customColumn where id = :id")
        fun findByPrimaryKey(id: String): CustomColumnNameEntity

        @Query("SELECT * FROM customColumn where id IN (:ids)")
        fun findByPrimaryKeys(vararg ids: String): List<CustomColumnNameEntity>

        @Query("SELECT * FROM customColumn where id IN (:ids)")
        fun findByPrimaryKeys(ids: List<String>): List<CustomColumnNameEntity>

        @Query("SELECT * FROM customColumn LIMIT :limit")
        fun loadFirst(limit: Int): List<CustomColumnNameEntity>

        @Query("SELECT * FROM customColumn")
        fun loadCursor(): Cursor

        @Insert
        fun insert(entity: CustomColumnNameEntity)
    }
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/CustomColumnNameEntity.kt](#))

Here, we changed the text property's column to words, along with specifying the table name. The SQL will reflect that change:

```
CREATE TABLE IF NOT EXISTS customColumn (id TEXT NOT NULL, title TEXT NOT NULL, words
TEXT, version INTEGER NOT NULL, PRIMARY KEY(id))
```

...even though we still refer to the property by its regular Kotlin name:

THE DAO OF ENTITIES

```
package com.commonware.room.misc

import androidx.room.Room
import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4
import com.natpryce.hamkrest.assertion.assertThat
import com.natpryce.hamkrest.equalTo
import com.natpryce.hamkrest.isEmpty

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*
import java.util.*

@RunWith(AndroidJUnit4::class)
class CustomColumnNameEntityTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.customColumn()

    @Test
    fun customColumnPersists() {
        assertThat(underTest.loadAll(), isEmpty)

        val original = CustomColumnNameEntity(
            id = UUID.randomUUID().toString(),
            title = "This space available for rent",
            text = "This will be stored as words. Well, it will be stored in a column named 'words'."
        )

        underTest.insert(original)

        val retrieved = underTest.findByPrimaryKey(original.id)

        assertThat(retrieved.id, equalTo(original.id))
        assertThat(retrieved.title, equalTo(original.title))
        assertThat(retrieved.text, equalTo(original.text))
    }
}
```

(from [MiscSamples/src/androidTest/java/com/commonware/room/misc/CustomColumnNameEntityTest.kt](#))

Note, though, that many of the annotation attributes that Room uses refer to column names, not property names. For example, suppose that instead of using `*` to indicate that our queries should return all columns, we list the ones that we want... and we use property names:

```
@Query("SELECT id, title, text, version FROM customColumn")
fun loadAll(): List<CustomColumnNameEntity>
```

Android Studio will be unhappy with you:

```
@Query( value: "SELECT id, title, text, version FROM customColumn")  
fun loadAll(): List<CustomColumnNameEntity>
```

Figure 1: Android Studio, Showing Syntax Error

And, if you try ignoring Android Studio and building the project anyway, you will get a build error:

```
error: There is a problem with the query: [SQLITE_ERROR] SQL error or missing  
database (no such column: text)  
    public abstract  
java.util.List<com.commonware.room.misc.CustomColumnNameEntity> loadAll();
```

You need to use the column name instead:

```
@Query("SELECT id, title, words, version FROM customColumn")  
fun loadAll(): List<CustomColumnNameEntity>
```

Also note that adding `@ColumnInfo` to a `@Transient` property means that this property will be included when creating the table structure, despite the `@Transient` annotation. By default, `@Transient` properties are ignored, but adding `@ColumnInfo` indicates that you want that default behavior to be overridden.

Other @ColumnInfo Options

Beyond specifying the column name to use, you can configure other options on a `@ColumnInfo` annotation. We saw using `index = true` earlier to add an index to a column, but we have options beyond that as well.

Collation

You can specify a `collate` property to indicate the collation sequence to apply to this column. Here, “collation sequence” is a fancy way of saying “comparison function for comparing two strings”.

There are four options:

- `BINARY` and `UNDEFINED`, which are equivalent, the default value, and indicate that case is sensitive
- `NOCASE`, which indicates that case is not sensitive (more accurately, that the

- 26 English letters are converted to uppercase)
- RTRIM, which indicates that trailing spaces should be ignored on a case-sensitive collation

There is no full-UTF equivalent of NOCASE in SQLite.

Type Affinity

Normally, Room will determine the type to use on the column in SQLite based upon the type of the property (e.g., Int properties create INTEGER columns). If, for some reason, you wish to try to override this behavior, you can use the typeAffinity property on @ColumnInfo to specify some other type to use.

Default Values

@ColumnInfo also has a defaultValue property. As you might guess from the name, it provides a default value for the column in the table definition.

However, “out of the box”, it may be less useful than you think. If you @Insert an entity, the value for this column from the entity will be used, not the default value.

We will explore defaultValue, and scenarios where it is useful, [later in the book](#).

DAOs and Queries

One popular thing to do with a database is to get data out of it. For that, we add @Query functions on our DAO.

Those do not have to be especially complex. The loadAll() functions in the samples shown in this chapter are delightfully simple:

```
@Query("SELECT * FROM customColumn")
fun loadAll(): List<CustomColumnNameEntity>
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/CustomColumnNameEntity.kt](#))

However, SQL queries with SQLite can get remarkably complicated. Room tries to support a lot of the standard SQL syntax, but Room adds its own complexity, in terms of trying to decipher how to interpret your @Query function’s arguments and return type.

Adding Parameters

As we saw with functions like `findByPrimaryKey()`, you can map function arguments to query parameters by using `:` syntax. Put `:` before the argument name and its value will be injected into the query:

```
@Query("SELECT * FROM customColumn where id = :id")
fun findByPrimaryKey(id: String): CustomColumnNameEntity
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/CustomColumnNameEntity.kt](#))

WHERE Clause

Principally, your function arguments will be injected into your WHERE clause, such as in the above example.

Note that Room has special support for IN in a WHERE clause, where you can query using a vararg or List parameter:

```
@Query("SELECT * FROM customColumn where id IN (:ids)")
fun findByPrimaryKeys(vararg ids: String): List<CustomColumnNameEntity>

@Query("SELECT * FROM customColumn where id IN (:ids)")
fun findByPrimaryKeys(ids: List<String>): List<CustomColumnNameEntity>
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/CustomColumnNameEntity.kt](#))

Here, the IN `(:ids)` SQL syntax will be expanded by Room to include all of the values that you supply in the argument. In this case, you would retrieve all of the entities matching any of those primary key values.

Other Clauses

Wherever SQLite allows `?` placeholders, Room should allow function arguments to be used instead.

So, for example, you can parameterize a LIMIT clause:

```
@Query("SELECT * FROM customColumn LIMIT :limit")
fun loadFirst(limit: Int): List<CustomColumnNameEntity>
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/CustomColumnNameEntity.kt](#))

What You Can Return

We have seen that a `@Query` can return a single entity (e.g., the single-ID `findByPrimaryKey()` functions) or a collection of entities (e.g., `loadAll()` returning a `List` of entities).

While those are simple, Room offers a fair bit more flexibility than that.

Returning Cursor

In addition to returning single objects or collections of objects, a Room `@Query` can return a good old-fashioned `Cursor`:

```
@Query("SELECT * FROM customColumn")
fun loadCursor(): Cursor
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/CustomColumnNameEntity.kt](https://github.com/commonsware/room/blob/master/misc/CustomColumnNameEntity.kt))

This is particularly useful if you are migrating legacy code that uses `CursorAdapter` or other `Cursor`-specific classes. Similarly, if you are looking to expose part of a Room-defined database via a `ContentProvider`, it may be more convenient for you to get your results in the form of a `Cursor`, so that you can just return that from the provider's `query()` function.

Note that, as with getting a `Cursor` from `SQLiteDatabase`, you are responsible for closing the `Cursor` when you are done with it.

Non-Entity Results

For small entities, like the ones shown so far in this chapter, usually we will retrieve all columns in the query. However, the real rule is: the core return object of the `@Query` function must be something that Room knows how to fill in from the columns that you request.

For wider tables with many columns, this is important. For example, perhaps for a `RecyclerView`, you only need a couple of columns, but for all entities in the table. In that case, it might be nice to only retrieve those specific columns. You have two ways to do that:

1. Have your `@Entity` support only a subset of columns, allowing the rest to be `null` or otherwise tracking the fact that we only retrieved a subset of

- columns from the table
- 2. Return something other than the entity that you have associated with this table

If you look at your @Dao-annotated interface, you will notice that while functions might refer to entities, its annotations do not. That is because the DAO is somewhat independent of the entities. The entities describe the table, but the DAO is not limited to using those entities. So long as the DAO can fulfill the contract stipulated by the SQL, the function arguments, and the function return type, Room is perfectly happy.

For example, suppose that you were making an app store client. While many developers think that the Play Store is the only app store client, there are lots of alternatives, such as [F-Droid](#), which specializes in open source apps.

In your data model, you will need some sort of Room entity representing apps in the store catalog. That may involve a lot of data, where you might need all of that data for the app “display” screen in your app. However, when you are presenting a list of apps — the whole catalog, some category of apps, or search results — you may need just a subset of that data. For that, you can take advantage of Room’s query flexibility and define a “list model” class that contains the subset of data that your lists need:

```
package com.commonware.room.misc

import androidx.room.*

@Entity(tableName = "apps")
data class AppEntity(
    @PrimaryKey
    val applicationId: String,
    val displayName: String,
    val shortDescription: String,
    val fullDescription: String,
    val latestVersionName: String,
    val lastUpdated: Long,
    val iconUrl: String,
    val packageUrl: String,
    val donationUrl: String
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM apps")
        fun loadAll(): List<AppEntity>

        @Query("SELECT applicationId, displayName, shortDescription, iconUrl FROM apps")
        fun loadListModel(): List<AppListModel>

        @Insert
        fun insert(entity: AppEntity)
    }
}
```

THE DAO OF ENTITIES

```
}  
  
data class AppListModel(  
    val applicationId: String,  
    val displayName: String,  
    val shortDescription: String,  
    val iconUrl: String  
)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/AppEntity.kt](#))

Here, AppEntity itself is unremarkable. However, we also define an AppListModel class, with a subset of the AppEntity properties. AppEntity.Store not only defines a loadAll() function that returns entities, but it has a loadListModels() function that returns AppListModel objects. loadListModels() has a @Query annotation that returns only the columns needed by AppListModel, and Room will happily pour that data into AppListModel objects for us. The only connection between AppEntity and AppListModel is that our loadListModels() query queries the apps table.

Now, app code that displays lists, or otherwise only needs this subset of the overall entity data, can work with the “list models” instead:

```
package com.commonsware.room.misc  
  
import androidx.room.Room  
import androidx.test.platform.app.InstrumentationRegistry  
import androidx.test.ext.junit.runners.AndroidJUnit4  
import com.natpryce.hamkrest.assertThat  
import com.natpryce.hamkrest.equalTo  
import com.natpryce.hamkrest.hasSize  
import com.natpryce.hamkrest.isEmpty  
  
import org.junit.Test  
import org.junit.runner.RunWith  
  
@RunWith(AndroidJUnit4::class)  
class AppEntityTest {  
    private val db = Room.inMemoryDatabaseBuilder(  
        InstrumentationRegistry.getInstrumentation().targetContext,  
        MiscDatabase::class.java  
    )  
        .build()  
    private val underTest = db.apps()  
  
    @Test  
    fun queryDisplayModels() {  
        assertThat(underTest.loadAll(), isEmpty)  
        assertThat(underTest.loadListModels(), isEmpty)  
  
        val fdroid = AppEntity(  
            applicationId = "org.fdroid.fdroid",  
            displayName = "F-Droid",  
            shortDescription = "An independent app store featuring open source Android apps",  
            fullDescription = "F-Droid is an installable catalogue of FOSS (Free and Open Source Software)  
applications for the Android platform. The client makes it easy to browse, install, and keep track of
```

THE DAO OF ENTITIES

```
updates on your device. Visit https://f-droid.org to learn more!",
    lastUpdated = 1566652015000,
    latestVersionName = "1.7.1",
    donationUrl = "https://flattr.com/thing/343053/F-Droid-Repository",
    packageUrl = "https://f-droid.org/FDroid.apk",
    iconUrl = "https://gitlab.com/fdroid/fdroidclient/raw/master/app/src/main/res/drawable-hdpi/
ic_launcher.png?inline=true"
)

underTest.insert(fdroid)

underTest.loadAll().let {
    assertThat(it, hasSize(equalTo(1)))
    assert(it[0] == fdroid)
}

underTest.loadListModels().let {
    assertThat(it, hasSize(equalTo(1)))

    val model = it[0]

    assertThat(model.applicationId, equalTo(fdroid.applicationId))
    assertThat(model.displayName, equalTo(fdroid.displayName))
    assertThat(model.shortDescription, equalTo(fdroid.shortDescription))
    assertThat(model.iconUrl, equalTo(fdroid.iconUrl))
}
}
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/AppEntityTest.kt](#))

Note that `@ColumnInfo` annotations can be used on any class, not just entities. Frequently, if you use `@ColumnInfo` on a property in an entity, you will wind up using that same `@ColumnInfo` on the corresponding property in any sort of “list model”-style object, so that the property names line up with the column in the table.

Reactive Return Types

All of our DAO functions have returned values directly, whether those values are entities, lists of entities, or something else (e.g., list of some “tuple” objects).

Those functions are synchronous. The query will be performed when we call the function, and we get the results of the query returned to us.

That is very simple. It is also very annoying, as it means that we have to deal with background threads ourselves. We do not want to be calling these DAO functions on the main application thread, as database I/O can be slow. As a result, we will be forced to use something else, such as a Java Executor, to be able to call the functions from a background thread.

However, if you look at a lot of Room sample code, you will see that the `@Query`-annotated functions often wrap the return values in... something else:

- `LiveData`
- `Flow`
- `Single`
- `Observable`
- `Flowable`
- and so on

These are classes from reactive frameworks. `LiveData` is part of the Android Jetpack, `Flow` is from Kotlin coroutines, and the others are from RxJava. These frameworks are designed to help simplify threading. They hide the complexity of doing the actual database I/O on a background thread while getting the results to you on the main application thread (or some other thread of interest to you).

We will explore these options, and threads with Room in general, [in an upcoming chapter](#).

Aggregate Functions

SQL supports aggregate functions, like `COUNT` and `SUM`. If you include these in a query, you get those calculated values back, instead of (or perhaps in addition to) actual data from a table.

Room also supports aggregate functions. However, by definition, there is no entity whose properties are counts or sums. We have two options for getting our results back for these calculations:

- If we need just a single value, we can have a `@Query`-annotated function return an `Int`, `Long`, or other basic type that represents the result from the aggregate function
- Otherwise, we use a variation on the approach from the preceding section, where we create a data class or similar structure that Room can use to return our results

```
package com.commonware.room.misc

import androidx.room.*
import kotlin.random.Random

data class CountAndSumResult(val count: Int, val sum: Long)
```

```
@Entity(tableName = "aggregate")
class AggregateEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0,
    val value: Long = Random.nextLong(1000000)
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM aggregate")
        fun loadAll(): List<AggregateEntity>

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

        @Query("SELECT COUNT(*) as count, SUM(value) as sum FROM aggregate")
        fun countAndSum(): CountAndSumResult

        @Insert
        fun insert(entities: List<AggregateEntity>)
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/AggregateEntity.kt](#))

Here, our entity just has an auto-generated primary key, plus some other randomly-generated Long value (cunningly stored in a value property). In its associated `AggregateEntity.Store`, in addition to our standard `loadAll()` function and an `insert()` that accepts a List of entities, we have:

- `count()`, which returns an `Int` representing the count of rows in the table
- `countAndSum()`, which returns the count of rows and the sum of the values, in the form of a `CountAndSumResult`.

Note that the SQL for `countAndSum()` uses the AS operator to put a specific name on the returned values. Those need to map to the property names in your result type.

We can then use those functions just like any other `@Dao` functions. And, in the case of `countAndSum()`, we can use Kotlin's destructuring declarations to avoid having to fuss with a `CountAndSumResult`:

```
package com.commonsware.room.misc

import androidx.room.Room
import androidx.test.platform.app.InstrumentationRegistry
```

```
import androidx.test.ext.junit.runners.AndroidJUnit4
import com.natpryce.hamkrest.assertion.assertThat
import com.natpryce.hamkrest.equalTo
import com.natpryce.hamkrest.greaterThan
import com.natpryce.hamkrest.hasSize
import com.natpryce.hamkrest.isEmpty

import org.junit.Test
import org.junit.runner.RunWith

import java.util.*

@RunWith(AndroidJUnit4::class)
class AggregateEntityTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.aggregate()

    @Test
    fun aggregateFunctions() {
        assertThat(underTest.loadAll(), isEmpty)

        underTest.insert(List(100) { AggregateEntity() })

        assertThat(underTest.count(), equalTo(100))

        val (count, sum) = underTest.countAndSum()

        assertThat(count, equalTo(100))
        assertThat(sum, greaterThan(0L))
    }
}
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/AggregateEntityTest.kt](#))

Dynamic Queries

Sometimes, you do not know the query at compile time.

One scenario for this is when you want to expose a Room-managed database via a ContentProvider to third-party apps. You could document that you support a limited set of options in your provider's query() function, ones that you can map to @Query functions on your DAO. Alternatively, you could generate a SQL statement

using `SQLiteQueryBuilder` that supports what your table offers, but then you need to somehow execute that statement and get a `Cursor` back.

You have a few options for handling this sort of situation.

query()

`RoomDatabase` has a `query()` function that is analogous to `rawQuery()` on a `SQLiteDatabase`. Pass it the SQL statement and an `Object` array of position parameters, and `RoomDatabase` will give you a `Cursor` back.

The benefit is that this is quick and easy. The downside is that you wind up with a `Cursor`, which is less convenient than the model objects that you get back from `@Query` functions on your `@Dao`.

@RawQuery

Another option is `@RawQuery`. Like `@Query`, this is an annotation that you can add to a function on your `@Dao`. And, like `@Query`, you can have that function return instances of an `@Entity` or other POJO.

However, rather than supplying a fixed SQL statement in the annotation, you provide a `SupportSQLiteQuery` object as a parameter to the `@RawQuery` function:

```
@RawQuery
fun _findMeSomething(query: SupportSQLiteQuery): List<Foo>
```

A `SupportSQLiteQuery` comes from [the support database API](#), which is how Room interacts with your `SQLite` database. Fortunately, for the purposes of using `@RawQuery`, the only thing that you need from that API is `SimpleSQLiteQuery`. Its constructor takes the same two parameters as does `rawQuery()` on a `SQLiteDatabase`:

- The SQL statement to execute, and
- An `Object` array of values to use to replace positional placeholders

So, you can wrap your query and placeholder values in a `SimpleSQLiteQuery`, pass that to your `@RawQuery`-annotated function, and Room will take care of the rest.

Other DAO Operations

To get data out of a database, generally it is useful to put data into it. We have seen basic @Insert, @Update, and @Delete DAO functions on NoteStore:

```
package com.commonware.room.notes

import androidx.room.*

@Dao
interface NoteStore {
    @Query("SELECT * FROM notes")
    fun loadAll(): List<NoteEntity>

    @Insert
    fun insert(note: NoteEntity)

    @Update
    fun update(note: NoteEntity)

    @Delete
    fun delete(vararg notes: NoteEntity)
}
```

(from [NoteBasics/src/main/java/com/commonware/room/notes/NoteStore.kt](#))

Generally speaking, these scenarios are simpler than @Query. The @Insert, @Update, and @Delete set up simple functions for inserting, updating, or deleting entities passed to their functions... and that is pretty much it. However, there are a few additional considerations that we should explore.

Parameters

@Insert, @Update, and @Delete work with entities. In the above code, insert() and update() each take a single entity. delete() takes a vararg of entities, so you can pass one or several as you see fit.

You can also have a List of entities, as we saw in the insert() function in AggregateEntity:

```
package com.commonware.room.misc

import androidx.room.*
import kotlin.random.Random
```

```
data class CountAndSumResult(val count: Int, val sum: Long)

@Entity(tableName = "aggregate")
class AggregateEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0,
    val value: Long = Random.nextLong(1000000)
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM aggregate")
        fun loadAll(): List<AggregateEntity>

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

        @Query("SELECT COUNT(*) as count, SUM(value) as sum FROM aggregate")
        fun countAndSum(): CountAndSumResult

        @Insert
        fun insert(entities: List<AggregateEntity>)
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/AggregateEntity.kt](#))

Return Values

Frequently, you just have these functions return nothing (technically, they return `Unit`, though we can drop that in Kotlin).

However:

- For `@Update` and `@Delete`, you can have them return an `Int`, which will be the number of rows affected by the update or delete operation
- For an `@Insert` function accepting a single entity, you can have it return a `Long` which will be the ROWID of the entity (and, if you are using an auto-increment int as your primary key, this will also be that key)
- For an `@Insert` function accepting multiple entities, you can have it return a `LongArray` or a `List` of `Long` values, being the corresponding ROWID values for those inserted entities

Conflict Resolution

@Insert and @Update support an optional onConflict property. This maps to [SQLite's ON CONFLICT clause](#) and indicates what should happen if there is either a uniqueness violation (e.g., duplicate primary keys) or a NOT NULL violation when the insert or update should occur.

The value of onConflict is an OnConflictStrategy enum:

Value	Meaning
OnConflictStrategy.ABORT	Cancels this statement but preserves prior results in the transaction and keeps the transaction alive
OnConflictStrategy.FAIL	Like ABORT, but accepts prior changes by this specific statement (e.g., if we fail on the 50th row to be updated, keep the changes to the preceding 49)
OnConflictStrategy.IGNORE	Like FAIL, but continues processing this statement (e.g., if we fail on the 50th row out of 100, keep the changes to the other 99)
OnConflictStrategy.REPLACE	For uniqueness violations, deletes other rows that would cause the violation before executing this statement
OnConflictStrategy.ROLLBACK	Rolls back the current transaction

The default strategy for @Insert and @Update is ABORT.

We will explore these conflict strategies in greater detail [much later in the book](#).

Other Operations

The primary problem with @Insert, @Update, and @Delete is that they need entities. In part, that is so the DAO function knows what table to work against.

For anything else, use @Query. @Query not only works with operations that return

result sets, but with *any* SQL that you wish to execute, even if that SQL does not return a result set.

So, for example, you could have:

```
@Query("DELETE FROM aliens")
fun nukeFromOrbit() // it's the only way to be sure
```

...or INSERT INTO ... SELECT FROM ... syntax, or pretty much any other combination that cannot be supported directly by @Insert, @Update, and @Delete annotations.

Consider @Insert, @Update, and @Delete to be “convenience annotations” for entity-based operations, where @Query is the backbone for your DAO functions.

Transactions and Room

By default, SQLite treats each individual SQL statement as an individual transaction. To the extent that Room winds up generating multiple SQL statements in response to our annotations, it is Room’s responsibility to wrap those statements in a suitable transaction.

However, sometimes, you have business logic that requires a transaction, for operations that require multiple DAO functions. For example, persisting an invoice might involve inserting an Invoice and all of its InvoiceLineItem objects, and that might require more than one DAO function to achieve.

Room offers two ways of setting up app-defined transactions: the @Transaction annotation and some functions on RoomDatabase.

Using @Transaction

Your DAO can have one or more functions that have the @Transaction annotation. Whatever a @Transaction-annotated function does is wrapped in a SQLite transaction. The transaction will be committed if the @Transaction-annotated function does not throw an exception. If it does, the transaction will be rolled back.

There are two places to apply @Transaction: custom open functions on an abstract DAO class, or on @Query functions.

Custom Functions

Here, the idea is that your @Transaction-annotated function would make multiple DAO calls to other functions (e.g., ones with @Insert or @Query annotations), so that the work performed in those other functions “succeed or fail as a whole”.

Given our fictitious Invoice example, we might have something like this:

```
@Dao
abstract class InvoiceStore {
    @Insert
    fun _insert(invoice: Invoice)

    @Insert
    fun _insertItems(lineItems: List<InvoiceLineItem>)

    @Transaction
    open fun insert(invoice: Invoice) {
        _insert(invoice)
        _insertItems(invoice.getLineItems())
    }
}
```

Here, we still use an insert() function to insert an Invoice, but we use that to wrap two DAO calls to insert the Invoice metadata and insert the InvoiceLineItem objects.

Note that the function with @Transaction needs to be open. Room will generate a concrete implementation of your DAO, either extending your abstract class or implementing your interface. To make @Transaction work, Room code-generates an overriding function that wraps a call to your implementation in a transaction. However, in Kotlin, concrete functions cannot be overridden without the open keyword. Leaving that keyword off [may result in strange compile error messages](#).

On @Query Functions

It may seem odd to have to specifically request a transaction on a @Query-annotated function. After all, the default behavior of SQLite is to have each individual SQL statement be in its own transaction.

However, there are two scenarios called out in [the documentation](#) where @Transaction would be a good idea. One is tied to @Relation, which we will cover

[later in the book](#).

The other is tied to a little-known issue with Android's SQLite support: things get weird when the result set of a query exceeds 1MB. In that case, using the regular Android SQLiteDatabase API, the Cursor that you get back does not contain the full result set. Instead, it contains a “window” of results, and if you position the Cursor after that window, the query is re-executed to load in the next window. This can lead to inconsistencies, if the database is changed in between those two database requests to populate the window. Room, by default, will load the entire result set into your entities, quickly moving through the windows as needed, but there is still a chance that a database modification occurs while this is going on. Using `@Transaction` would help ensure that this is not an issue, by having the entire query — including traversing the windows — occur inside a transaction.

Using RoomDatabase

Alternatively, RoomDatabase offers the same `beginTransaction()`, `endTransaction()`, and `setTransactionSuccessful()` functions that you see on SQLiteDatabase, and so you use the same basic algorithm:

```
roomDb.beginTransaction()

try {
    // bunch of DAO operations here
    roomDb.setTransactionSuccessful()
}
finally {
    roomDb.endTransaction()
}
```

The advantage to this approach is that you can put the transaction logic somewhere other than the DAO, if that would be more convenient or make more sense for your particular implementation. However, it is a bit more work.

Room and Custom Types

So far, all of our properties have been basic primitives (such as `Int`) or `String`. There is a good reason for that: those are all that Room understands “out of the box”. Everything else requires some amount of assistance on our part.

Sometimes, a property in an entity will be related to another entity. Those are relations, and we will consider those in [the next chapter](#).

However, other times, a property in an entity does not map directly to primitives and `String` types, or to another entity. For example:

- What do we do with a Java `Date`, `Calendar`, or `Instant` objects? Do we want to store that as a milliseconds-since-the-Unix-epoch value as a `Long`? Do we want to store a string representation in a standard format, for easier readability (at the cost of disk space and other issues)?
- What do we do with a `Location` object? Here, we have two pieces: a latitude and a longitude. Do we have two columns that combine into one property? Do we convert the `Location` to and from a `String` representation?
- What do we do with collections of strings, such as lists of tags?
- What do we do with enums?

And so on.

In this chapter, we will explore two approaches for handling these things without creating another entity class: type converters and embedded types.

Type Converters

Type converters are a pair of functions, annotated with `@TypeConverter`, that map

the type for a single database column to a type for a Kotlin property. So, for example, we can:

- Map an `Instant` property to a `Long`, which can go in a SQLite `INTEGER` column
- Map a `Location` property to a `String`, which can go in a SQLite `TEXT` column
- Map a collection of `String` values to a single `String` (e.g., comma-separated values), which can go in a SQLite `TEXT` column
- Etc.

However, type converters offer only a 1:1 conversion: a single property to and from a single SQLite column. If you have a single property that should map to several SQLite columns, the `@Embedded` approach can handle that, as we will see [later in this chapter](#).

Setting Up a Type Converter

First, define a Kotlin class somewhere. The name, package, superclass, etc. do not matter.

Next, for each type to be converted, create two functions that convert from one type to the other. So for example, you would have one function that takes an `Instant` and returns a `Long` (e.g., returning the milliseconds-since-the-Unix-epoch value), and a counterpart function that takes a `Long` and returns an `Instant`. If the converter function is passed `null`, the proper result is `null`. Otherwise, the conversion is whatever you want, so long as the “round trip” works, so that the output of one converter function, supplied as input to the other converter function, returns the original value.

Then, each of those functions get the `@TypeConverter` annotation. The function names do not matter, so pick a convention that works for you.

Finally, you add a `@TypeConverters` annotation, listing this and any other type converter classes, to... something. What the “something” is controls the scope of where that type converter can be used.

The simple solution is to add `@TypeConverters` to the `RoomDatabase`, which means that anything associated with that database can use those type converters. However, sometimes, you may have situations where you want different conversions between the same pair of types, for whatever reason. In that case, you can put the `@TypeConverters` annotations on narrower scopes:

ROOM AND CUSTOM TYPES

@TypeConverters Location	Affected Areas
Entity class	all properties in the entity
Entity property	that one property in the entity
DAO class	all functions in the DAO
DAO function	that one function in the DAO, for all parameters
DAO function parameter	that one parameter on that one function

For example, the `TransmogrifyingEntity` file in the [the MiscSamples module](#) of [the book's primary sample project](#) has not only `TransmogrifyingEntity` but also a `TypeTransmogrifier` class. A [transmogrifier](#) is a ~30-year-old piece of advanced technology that can convert one thing into another. `TypeTransmogrifier` has a set of functions that turn one type into another — we will examine those functions in upcoming sections. `TransmogrifyingEntity` itself has the `@TypeConverters` annotation, indicating that the type converters on `TypeTransmogrifier` should be used for that entity:

```
@Entity(tableName = "transmogrified")
@TypeConverters(TypeTransmogrifier::class)
data class TransmogrifyingEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0,
    val creationTime: Instant = Instant.now(),
    val location: Location = Location(null as String?).apply {
        latitude = 0.0
        longitude = 0.0
    },
    val tags: Set<String> = setOf()
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM transmogrified")
        fun loadAll(): List<TransmogrifyingEntity>

        @Insert
        fun insert(entity: TransmogrifyingEntity)
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/TransmogrifyingEntity.kt](#))

Example: Dates and Times

A typical way of storing a date/time value in a database is to use the number of milliseconds since the Unix epoch (i.e., the number of milliseconds since midnight, 1 January 1970). `Instant` has a `getEpochMillis()` function that returns this value.

The `TypeTransmogrifier` class has a pair of functions designed to convert between an `Instant` and a `Long`:

```
@TypeConverter
fun instantToLong(timestamp: Instant?) = timestamp?.toEpochMilli()

@TypeConverter
fun longToInstant(timestamp: Long?) =
    timestamp?.let { Instant.ofEpochMilli(it) }
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/TransmogrifyingEntity.kt](#))

Each has the `@TypeConverter` annotation, so Room knows to examine those functions when it has the need to convert between types, such as finding some Room-native type into which we can convert an `Instant`.

`TransmogrifyingEntity` has an `Instant` property named `creationTime`:

```
val creationTime: Instant = Instant.now(),
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/TransmogrifyingEntity.kt](#))

Given that `TransmogrifyingEntity` has the `@TypeConverters` annotation pointing to `TypeTransmogrifier`, Room will be able to find a way to convert that `Instant` to something that it knows how to handle: a `Long`. As a result, our timestamp will be stored in a SQLite `INTEGER` column.

Example: Locations

A `Location` object contains a latitude, longitude, and perhaps other values (e.g., altitude). If we only care about the latitude and longitude, we could save those in the database in a single `TEXT` column, so long as we can determine a good format to use for that string. One possibility is to have the two values separated by a semicolon.

That is what these two type converter functions on `TypeTransmogrifier` do:

```
@TypeConverter
fun locationToString(location: Location?) =
    location?.let { "${it.latitude};${it.longitude}" }

@TypeConverter
fun stringToLocation(location: String?) = location?.let {
    val pieces = location.split(';')

    if (pieces.size == 2) {
        try {
            Location(null as String?).apply {
                latitude = pieces[0].toDouble()
                longitude = pieces[1].toDouble()
            }
        } catch (e: Exception) {
            null
        }
    } else {
        null
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/TransmogrifyingEntity.kt](#))

Our entity class has a Location property:

```
val location: Location = Location(null as String?).apply {
    latitude = 0.0
    longitude = 0.0
},
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/TransmogrifyingEntity.kt](#))

Room will know how to convert a Location to and from a String, so our location will be stored in a SQLite TEXT column.

However, the downside of using this approach is that we cannot readily search based on location. If your location data is not a searchable property, and it merely needs to be available when you load your entities from the database, using a type converter like this is fine. [Later in this chapter](#), we will see another approach (@Embedded) that allows us to store the latitude and longitude as separate columns while still mapping them to a single data class in Kotlin.

Example: Simple Collections

TEXT and BLOB columns are very flexible. So long as you can marshal your data into a

String or byte array, you can save that data in TEXT and BLOB columns. As with the comma-separated values approach in the preceding section, though, columns used this way are poor for searching.

So, suppose that you have a Set of String values that you want to store, perhaps representing tags to associate with an entity. One approach is to have a separate Tag entity and [set up a relation](#). This is the best approach in many cases. But, perhaps you do not want to do that for some reason.

You can use a type converter, but you need to decide how to represent your data in a column. If you are certain that the tags will not contain some specific character (e.g., a comma), you can use the delimited-list approach demonstrated with locations in the preceding section. If you need more flexibility than that, you can always use JSON encoding, as these type converters do:

```
@TypeConverter
fun stringSetToString(list: Set<String>?) = list?.let {
    val sw = StringWriter()
    val json = JsonWriter(sw)

    json.beginArray()
    list.forEach { json.value(it) }
    json.endArray()
    json.close()

    sw.buffer.toString()
}

@TypeConverter
fun stringToStringSet(stringified: String?) = stringified?.let {
    val json = JsonReader(StringReader(it))
    val result = mutableSetOf<String>()

    json.beginArray()

    while (json.hasNext()) {
        result += json.nextString()
    }

    json.endArray()

    result.toSet()
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/TransmogrifyingEntity.kt](#))

Here, we use the `JsonReader` and `JsonWriter` classes that have been part of Android since API Level 11. Alternatively, you could use a third-party JSON library (e.g., Gson, Moshi).

Given these type conversion functions, we can use a `Set` of `String` values in `TransmogrifyingEntity`:

```
val tags: Set<String> = setOf()
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/TransmogrifyingEntity.kt](#))

...where the tags will be stored in a TEXT column.

Embedded Types

With type converters, we are teaching Room how to deal with custom types, but we are limited to mapping from one property to one column. That property might be complex, but it still goes into one column in the table.

What happens, though, when we have multiple columns that should combine to create a single property?

In that case, we can use the `@Embedded` annotation on some data class or other simple Kotlin class, then use that class as a type in an entity.

Example: Locations

For example, as was noted earlier in this chapter, cramming a location into a single TEXT column works, but we cannot readily query on the resulting column. If we want to query for locations near some location in the database, it would be much more convenient to have the latitude and longitude stored as individual REAL columns. But, using type converters, we cannot map two columns to one property.

With `@Embedded`, we can, as we can see in the `EmbeddedLocationEntity` class in the [the MiscSamples module](#):

```
package com.commonsware.room.misc

import android.location.Location
import android.util.JsonReader
import android.util.JsonWriter
import androidx.room.*
```

ROOM AND CUSTOM TYPES

```
import org.threeten.bp.Instant
import java.io.StringReader
import java.io.StringWriter

data class LocationColumns(
    val latitude: Double,
    val longitude: Double
) {
    constructor(loc: Location) : this(loc.latitude, loc.longitude)
}

@Entity(tableName = "embedded")
data class EmbeddedLocationEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0,
    val name: String,
    @Embedded
    val location: LocationColumns
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM embedded")
        fun loadAll(): List<EmbeddedLocationEntity>

        @Insert
        fun insert(entity: EmbeddedLocationEntity)
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/EmbeddedLocationEntity.kt](#))

Here, we have a `LocationColumns` class that wraps our latitude and longitude. The entity itself has a `LocationColumns` property, named `location`, marked with the `@Embedded` annotation. Now, Room will use individual REAL columns for our latitude and longitude:

```
CREATE TABLE IF NOT EXISTS embedded (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
name TEXT NOT NULL, latitude REAL NOT NULL, longitude REAL NOT NULL)
```

As a result, we could construct queries on those columns, if we wished.

Note that even though a class used in `@Embedded`, like `LocationColumns`, is not an entity, you can still use `@ColumnInfo` annotations on it to rename columns, if desired. Also, you are subject to the same rules for data types: the properties need to be types that Room understands, natively or via type converters that you create.

Simple vs. Prefixed

What happens if we need *two* locations, though? Perhaps we need `officeLocation` and `affiliateLocation`, or something like that.

By default, Room generates column names based on the `@Embedded` class' property names, perhaps modified by `@ColumnInfo` annotations on those properties. In this case, though, if we have two `LocationColumns` properties in the same entity class, we would wind up with two `latitude` and two `longitude` columns, which neither Room nor SQLite will support.

To address this, the `@Embedded` annotation accepts an optional `prefix` property:

```
@Embedded(prefix = "office_")  
val officeLocation: LocationColumns
```

The columns for that entity will have the prefix added:

```
CREATE TABLE IF NOT EXISTS embedded (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,  
name TEXT NOT NULL, office_latitude REAL NOT NULL, office_longitude REAL NOT NULL)
```

Hence, having two `LocationColumns` simply means that one or both need to use distinct prefix values.

However, bear in mind that this changes the column names, so you will also need to adjust any `@Query` function that references those names, so that you use the appropriate prefix.

Room and Reactive Frameworks

Database I/O can be slow, particularly for larger databases and unoptimized operations. As a result, we invariably want to do that database I/O on background threads.

There are many options for doing that, including simply spinning up your own Thread or Executor that you use to make your DAO calls. However, the more popular way of addressing this nowadays is to use “reactive frameworks”, which wrap up threading and results delivery for you. In this chapter, we will examine Room’s support for a few reactive options: LiveData, Kotlin coroutines, and RxJava.

But, first, a word from our UI.

Room and the Main Application Thread

You might not be worried so much about the speed of your database I/O. Maybe you think that your database will never get large. Maybe you think that your users will all be using expensive devices (and think that expensive devices means that they will have fast database I/O). Maybe threads make your head hurt.

Hence, you might think that you can just go ahead and use Room on the main application thread, despite the fact that it will freeze your UI while that database I/O is going on. After all, in the code that we have seen so far in the book, we have not used Thread, Executor, or any fancy reactive framework — we have just made the database calls.

But that is because test functions are called on a background thread automatically. We do not have to fork a background thread of our own.

ROOM AND REACTIVE FRAMEWORKS

To see how Room behaves on the main application thread, we have to write our test to use the main application thread, such as via `runOnMainSync()`:

```
package com.commonware.room.misc

import androidx.room.Room
import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4
import com.natpryce.hamkrest.assertion.assertThat
import com.natpryce.hamkrest.equalTo
import com.natpryce.hamkrest.isEmpty

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

@RunWith(AndroidJUnit4::class)
class MainAppThreadGoBoomTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.autoGenerate()

    @Test
    fun goBoom() {
        InstrumentationRegistry.getInstrumentation().runOnMainSync {
            assertThat(underTest.loadAll(), isEmpty)

            val original = AutoGenerateEntity(id = 0, text = "This will get its own ID")
            val inserted = underTest.insert(original)

            assertTrue(original === inserted)
            assertThat(inserted.id, !equalTo(0L))
        }
    }
}
```

(from [MiscSamples/src/androidTest/java/com/commonware/room/misc/MainAppThreadGoBoomTest.kt](#))

This is a clone of the test we had for using `@PrimaryKey(autoGenerate = true)` on an entity, except that the test code is wrapped in a call to `runOnMainSync()`, to force it to run on the main application thread. While the original test succeeds, this one crashes with:

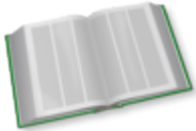
```
java.lang.IllegalStateException: Cannot access database on the main thread since it
may potentially lock the UI for a long period of time.
```

The developers who created Room block Room usage on the main application thread, as that is an anti-pattern.

So, we need to do something to avoid this sort of crash.

Room and LiveData

One “out of the box” option is to use LiveData.



You can learn more about LiveData in the "Thinking About Threads and LiveData" chapter of [Elements of Android Jetpack!](#)

No special dependencies are required, other than Room itself. You can simply wrap your DAO @Query return values in LiveData:

```
package com.commonware.todo.repo

import androidx.lifecycle.LiveData
import androidx.room.*
import kotlinx.coroutines.flow.Flow
import org.threeten.bp.Instant
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Instant = Instant.now(),
    val isCompleted: Boolean = false
) {
    constructor(model: ToDoModel) : this(
        id = model.id,
        description = model.description,
        isCompleted = model.isCompleted,
        notes = model.notes,
        createdOn = model.createdOn
    )

    fun toModel(): ToDoModel {
        return ToDoModel(
            id = id,
            description = description,
            isCompleted = isCompleted,
```

```
        notes = notes,
        createdOn = createdOn
    )
}

@Dao
interface Store {
    @Query("SELECT * FROM todos")
    fun all(): LiveData<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String): LiveData<ToDoEntity?>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun save(vararg entities: ToDoEntity)

    @Delete
    fun delete(vararg entities: ToDoEntity)
}
```

(from [LiveData/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

This ToDoEntity class is from [the LiveData module](#) of [the book's primary sample project](#).



You can learn more about the to-do application this module is based on in the "What We Are Building" chapter of [Exploring Android!](#)

We have a @Dao-annotated interface named ToDoEntity.Store. It has four functions, two of which have @Query annotations (all() and find()). Instead of returning entities directly, though, they return LiveData wrappers around those entities.

When we call all() or find(), we get a LiveData back immediately. The I/O will not be performed until we observe() that LiveData. At that point, the database I/O will be conducted on a Room-supplied background thread, but the results will be delivered to our Observer on the main application thread (as with all uses of LiveData).

Benefits of LiveData

As with all of our reactive options, the database I/O gets offloaded to a background

thread, so we do not need to fork such a thread ourselves. Yet, we still get the results delivered to us on the main application thread, so we can easily apply those results to our UI.

Also, LiveData does not require any additional dependencies, which can help to keep the app a bit smaller.

Issues with LiveData

LiveData is not an option for @Insert, @Update, or @Delete functions — you would still need to manage your own background threads for those.

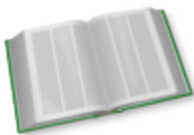
LiveData always delivers its results on the main application thread. This is great when we want those results on the main application thread. This is a problem when we do *not* want those results on the main application thread, such as performing some database I/O in preparation for making a Web service request.

LiveData needs to be observed, and the typical way of observing a LiveData requires that you pass a LifecycleOwner to observe(). This is annoying when you do not have a LifecycleOwner to use, such as when using LiveData in some types of services. You can use observeForever() to avoid the need for the LifecycleOwner, but then you need to remember to remove your Observer, lest you accidentally wind up with a memory leak.

And LiveData is lightweight by design. If you have complex background operations to perform, such as combining results from multiple sources and converting those results into specific object types, LiveData has limited facilities to help with that. And, what it does have can be somewhat arcane to use (e.g., MediatorLiveData).

Room and Coroutines

For Kotlin developers, the leading reactive solution is Kotlin's own coroutines system. This is a direct extension of the programming language and offers the most power with the least syntactic complexity.



You can learn more about coroutines in the "Introducing Coroutines" chapter of [Elements of Kotlin Coroutines](#)!

Room supports coroutines via the `androidx.room:room-ktx` dependency. Adding that will pull in a compatible version of coroutines in addition to Room's own code for supporting coroutines in `@Dao`-annotated interfaces.

suspend

One advantage that Room's coroutines support has over its LiveData support is that you can use coroutines with `@Insert`, `@Update`, and `@Delete`. Simply add the `suspend` keyword to the `@Dao`-annotated functions:

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun save(vararg entities: ToDoEntity)

@Delete
suspend fun delete(vararg entities: ToDoEntity)
```

(from [Coroutines/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](https://github.com/commonsware/todo-repo/blob/master/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt))

These functions are from a `ToDoEntity.Store` implementation in [the Coroutines module](#) of [the book's primary sample project](#). This module is functionally the same as the LiveData module, except that it uses coroutines with Room.

These suspend functions can then be called from any suitable CoroutineScope, such as the `viewModelScope` offered by Jetpack's ViewModel system.

Just as the LiveData `@Dao` functions use a Room-supplied background thread, so too do the suspend `@Dao` functions.

Flow

For `@Query`-annotated functions, you have two choices. You could use a suspend function, just as you can with `@Insert`, `@Update`, and `@Delete`. The closer equivalent of using LiveData, though, is to use Flow:

```
@Dao
interface Store {
    @Query("SELECT * FROM todos")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String): Flow<ToDoEntity?>

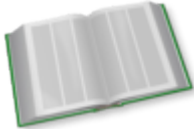
    @Insert(onConflict = OnConflictStrategy.REPLACE)
```

ROOM AND REACTIVE FRAMEWORKS

```
suspend fun save(vararg entities: ToDoEntity)

@Delete
suspend fun delete(vararg entities: ToDoEntity)
}
```

(from [Coroutines/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))



You can learn more about Flow in the "Introducing Flows and Channels" chapter of [Elements of Kotlin Coroutines](#)!

Flow is a bit like LiveData, in that you can observe it (via functions like `collect()`) to receive your results. Like the rest of coroutines, you can control the dispatcher that dictates what thread is used for receiving those results. Or, you can use the `asLiveData()` extension function supplied by the `androidx.lifecycle:lifecycle-livedata-ktx` artifact to convert a Flow into a LiveData for consumption by an activity or fragment:

```
package com.commonsware.todo.ui.roster

import androidx.lifecycle.LiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.asLiveData
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.launch

class RosterViewState(
    val items: List<ToDoModel> = listOf()
)

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    val states: LiveData<RosterViewState> =
        repo.items().map { RosterViewState(it) }.asLiveData()

    fun save(model: ToDoModel) {
        viewModelScope.launch {
            repo.save(model)
        }
    }
}
```


(from [Coroutines/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This ViewModel (RosterMotor) is not using the Flow from our `ToDoEntity.Store` directly. Rather, it is calling `all()` on a `ToDoRepository`, which in turn is using `ToDoEntity.Store`. The `all()` on `ToDoRepository` takes advantage of the rich set of operators on Flow to convert entities to model objects:

```
package com.commonsware.todo.repo

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.withContext

class ToDoRepository(
    private val store: ToDoEntity.Store,
    private val appScope: CoroutineScope
) {
    fun items(): Flow<List<ToDoModel>> =
        store.all().map { all -> all.map { it.toModel() } }

    fun find(id: String): Flow<ToDoModel?> = store.find(id).map { it?.toModel() }

    suspend fun save(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.save(ToDoEntity(model))
        }
    }

    suspend fun delete(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.delete(ToDoEntity(model))
        }
    }
}
```

(from [Coroutines/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

Benefits of Coroutines

Room's coroutines support covers all `@Dao` functions, whereas LiveData only works for `@Query`.

Coroutines provides flexibility for the thread that you use to receive the results. While often times you will use `Dispatchers.Main` to get the results on the main application thread (for UI use), you have the option of using other dispatchers for

other scenarios.

Coroutines overall are not tied to Android, the way that LiveData is. It will be more common to see libraries expose a coroutines-based API than one based on LiveData. As a result, for the overall Kotlin ecosystem, coroutines is likely to eclipse LiveData in popularity, if it has not done so already.

Issues with Coroutines

Kotlin's coroutines system is the youngest of the three main Room reactive frameworks. As such, it has not been "beaten up" as much as, say, RxJava has.

Coroutines are tied to Kotlin. While there are ways to get Java code to interoperate with Kotlin code that uses coroutines, it is rather clunky. If you expect to have a lot of Java code needing to work with your @Dao, you may be better off with LiveData or RxJava, until such time as you can move to coroutines.

Room and RxJava

The classic reactive solution for Java is RxJava. RxJava 2 is the most popular version, and it offers a rich-but-complex set of types for reactive results delivery. Room, via the `androidx.room:room-rxjava2` artifact, supports many of these, including Flowable, Observable, Single, Maybe, and Completable. Or, we can use an alpha edition of `androidx.room:room-rxjava3`, for RxJava 3, which debuted in 2020.

Your results-returning @Dao functions might use something like Observable or Maybe, while your other @Dao functions can use Completable:

```
@Dao
interface Store {
    @Query("SELECT * FROM todos")
    fun all(): Observable<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String): Maybe<ToDoEntity>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun save(vararg entities: ToDoEntity): Completable

    @Delete
    fun delete(vararg entities: ToDoEntity): Completable
}
```

(from [Rx/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

Unlike with coroutines and LiveData, Room has intermittent support for automatically putting your RxJava work on a scheduler with background threads. Room will put your Observable work on its own scheduler, but not Maybe or Completable, for example. For those, you will need to supply your own scheduler, such as `Schedulers.io()`:

```
package com.commonsware.todo.repo

import io.reactivex.Maybe
import io.reactivex.Observable
import io.reactivex.schedulers.Schedulers

class ToDoRepository(private val store: ToDoEntity.Store) {
    fun items(): Observable<List<ToDoModel>> = store.all()
        .map { all -> all.map { it.toModel() } }

    fun find(id: String): Maybe<ToDoModel> = store.find(id)
        .map { it.toModel() }
        .subscribeOn(Schedulers.io())

    fun save(model: ToDoModel) = store.save(ToDoEntity(model))
        .subscribeOn(Schedulers.io())

    fun delete(model: ToDoModel) = store.delete(ToDoEntity(model))
        .subscribeOn(Schedulers.io())
}
```

(from [Rx/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

Otherwise, you will get the “Cannot access database on the main thread” error.

Beyond that, though, you can then `subscribe()` to these RxJava types and consume their results as you see fit.

Benefits of RxJava

RxJava is the most seasoned of these reactive solutions, so it is more likely that you will find information about any problems that you encounter.

Like coroutines, RxJava gives you flexibility for supplying a scheduler on which to observe the results. This is in contrast to LiveData, which always delivers its results on the main application thread.

Issues with RxJava

RxJava has a very steep learning curve. Either LiveData or coroutines will be simpler for newcomers to pick up and use.

Observable Queries

For @Query-annotated functions returning LiveData, Flow, Observable, or Flowable, Room will deliver changes over time. You will get one result from the query initially. If you continue to observe the reactive data source, though, and you modify the database via Room, you will get *fresh results* delivered to you automatically via the reactive data source.

So, for example, if you have a fragment observing a query and using that data to populate a list, and other code in that fragment inserts a new row into the database, your fragment will get a fresh query result without having to manually re-request it.

This is very convenient in many cases. Bear in mind, though, that you just get a fresh result without any context. So, in the example from the previous paragraph, while you get a fresh query result, you are not told exactly what changed in that result... if anything. For example, you might be inserting a row that is not included in the query result because it failed to match a WHERE clause.

Conversely, you might get a new result delivered to you where nothing actually changed. Suppose you have an outstanding query with WHERE price > 10 in the SQL, and you insert a new row to that table where price is 5. Room will deliver you a fresh result representing the data change in the table... but since this new row does not match your WHERE clause, the “fresh” result will be the same as the last result. You could use distinctUntilChanged() on Flow or Observable to filter out those duplicate results, if desired.

Room and ListenableFuture

Room also supports [an AndroidX edition of Guava's ListenableFuture interface](#). You can have DAO functions return a ListenableFuture instead of an RxJava Single, for example. ListenableFuture allows you to register a callback to find out when the work is completed.

This option is relatively obscure and has very little documentation. Unless you are

integrating with some framework that itself is based on Future and ListenableFuture, the other reactive solutions presented in this chapter are substantially more popular.

Where Synchronous Room is Safe

So long as you do your Room database I/O on a background thread, Room does not care where that background thread came from. Room is not forcing you to use some Room-supplied thread — it is just forcing you to use some background thread.

Hence, you are welcome to use non-reactive Room @Dao options from places where you already have a background thread, such as a Worker for use with WorkManager or a JobIntentService.

Being Evil

Some developers do not like to take “no” for an answer. When they are told that they should not do database I/O on the main application thread — and that Room actively blocks that behavior — they get angry.

For those developers, there is `allowMainThreadQueries()`.

This is a method on `RoomDatabase.Builder` that you can call to turn off the ban on Room DAO calls on the main application thread:

```
private val db = Room.databaseBuilder(  
    myContext,  
    MiscDatabase::class.java,  
    DB_NAME  
)  
    .allowMainThreadQueries()  
    .build()
```

Now, `db` can be used on the main application thread, without complaints from Room. However, there *might* be complaints from users, or management, or the QA team, due to the app appearing to perform poorly.

Relations in Room

SQLite is a relational database. So far, we have not talked about that, focusing instead on standalone entities.

Room supports entities being related to other content in other tables. Room does *not* support entities being directly related to other entities, though.

And if that sounds strange, there is “a method to the madness”.

(or, in Kotlin, “a function to the foolishness”)

In this chapter, we will explore how you implement relational structures with Room and why Room has the restrictions that it does.

The Classic ORM Approach

Java ORMs have long supported entities having relations to other entities, though not every ORM uses the “entity” term.

One Android ORM that does is [greenDAO](#). It allows you to use annotations to indicate relations, such as in this Java snippet:

```
@Entity
public class Thingy {
    @Id private Long id;

    private long otherThingyId;

    @ToOne(joinProperty="otherThingyId")
    private OtherThingy otherThingy;
```

```
// other good stuff here
}  
  
@Entity  
public class OtherThingy {  
    @ID private Long id;  
}
```

These annotations result in `getOtherThingy()` and `setOtherThingy()` methods to be synthetically added to `Thingy` (or, more accurately, to a hidden subclass of `Thingy`, but for the purposes of this section, we will ignore that). Which `OtherThingy` our `Thingy` relates to is tied to that `otherThingyId` field, which is stored as a column in the table. When you call `getOtherThingy()`, `greenDAO` will query the database to load in the `OtherThingy` instance, assuming that it has not been cached already.

That is where the threading problem creeps in.

A History of Threading Mistakes

In Android app development, we are constantly having to fight to keep disk I/O off of the main application thread. Every millisecond that our code executes on the main application thread is a millisecond that the main application thread is not updating our UI. Our objective is to move as much disk I/O as possible off the main application thread. That is why we use all those nice reactive solutions from [the preceding chapter](#).

The problem is that the nice encapsulation that we get from object-oriented programming also encapsulates knowledge of whether disk I/O will be done when we call a particular method.

Classic use of `SQLiteDatabase` encounters this with the `rawQuery()/query()` family of methods. They return a `Cursor`. You might think — reasonably — that those methods execute the SQL query that you request. In truth, they do not. All they do is create a `SQLiteCursor` instance that holds onto the query and the `SQLiteDatabase`. Later, when you call a method that requires the actual query result (e.g., `getCount()`, to get the number of returned rows), *then* the query is executed against the database. As a result, all the work that you do to call `rawQuery()` or `query()` on a background thread gets wasted if you do not *also* do something to force the query to be executed on that same background thread. Otherwise, you may

wind up with the query being executed on the main application thread, with impacts on the UI.

greenDAO relations can work the same way. If you retrieve your Thingy on a background thread, then call `getOtherThingy()` on the main application thread, depending on what else has all occurred, `getOtherThingy()` might need to perform a database query... which you do not want on the main application thread.

The Room Approach

Room behaves a bit like other annotation-based Android ORMs, but when it comes to relations, Room departs from norms, in an effort to reduce the likelihood of threading problems.

Unlike the greenDAO example above, with Room, a Thingy cannot have a property for an OtherThingy that Room is expected to manage. You could have a property for an OtherThingy marked as `@Ignore`, but then you are on your own for dealing with that property.

The implication of an entity referencing another entity directly is that developers would expect that when Room retrieves the outer entity, that Room either will automatically retrieve the inner entity or will retrieve it lazily later on. The former approach avoids threading issues but runs the risk of loading more data than is necessary. The latter approach runs the risk of trying to do disk I/O on the main application thread.

This does not mean that you cannot have foreign keys. Room fully supports foreign key relationships, by way of a `@ForeignKey` annotation. This sets up the foreign keys in the appropriate tables... but that's about it. Room also has a `@Relation` annotation, to allow you to retrieve related data... but it does not involve entities as much as you might think.

And all of that will (hopefully) make more sense with some examples, from the [the MiscSamples module](#) of [the book's primary sample project](#).

One-to-Many Relations

Let's imagine that we are setting up an app for a book catalog. The catalog is divided into categories, and categories can have books. So, we need a Category entity to model the categories, and we need a Book entity to model the books. As part of this,

we also need to model the one-to-many relationship between Category and Book.

In this case, Category itself does not need anything special. It is just an ordinary Room entity:

```
package com.commonware.room.misc.onetomany

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "categories")
data class Category(
    @PrimaryKey
    val shortCode: String,
    val displayName: String
)
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/onetomany/Category.kt](#))

Note, though, that it does not have a List or Array or other collection of Book objects. You cannot ask a category for its books, at least not by asking the entity.

Configuring the Foreign Key

Book, and our DAO, are where things start to get interesting.

The Book class, in isolation, is about as plain as is Category:

```
data class Book(
    @PrimaryKey
    val isbn: String,
    val title: String,
    @ColumnInfo(index = true) var categoryShortCode: String
)
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/onetomany/Book.kt](#))

The only thing interesting here is that we declare an index on categoryShortCode. As the name suggests, this holds the shortCode primary key of the Category that is associated with this Book. Note that Book does not have an actual property for the Category, just its key.

When we scroll up the source code a bit and look at the @Entity annotation, we encounter a @ForeignKey:

RELATIONS IN ROOM

```
package com.commonware.room.misc.onetomany

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.ForeignKey.CASCADE
import androidx.room.PrimaryKey

@Entity(
    tableName = "books",
    foreignKeys = [ForeignKey(
        entity = Category::class,
        parentColumns = arrayOf("shortCode"),
        childColumns = arrayOf("categoryShortCode"),
        onDelete = CASCADE
    )]
)
data class Book(
    @PrimaryKey
    val isbn: String,
    val title: String,
    @ColumnInfo(index = true) var categoryShortCode: String
)
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/onetomany/Book.kt](#))

The `@Entity` annotation can have an array of `@ForeignKey` annotations (named `foreignKeys`). Each `@ForeignKey` annotation has at least three primary properties:

- `entity` points to the entity class that represents the “one” side of the one-to-many relation
- `parentColumns` identifies the column(s) in the parent table that represent the primary key
- `childColumns` identifies the column(s) in the child table that represent the parent’s primary key

In this case, `Category` has a simple single-property primary key, so `parentColumns` points to that (`shortCode`), while `childColumns` points to the corresponding column in the `Book` (`categoryShortCode`).

Cascades on Updates and Deletes

In addition, you can place `onUpdate` and `onDelete` properties on a `@ForeignKey` annotation. These indicate what actions should be taken on this entity when the parent of the foreign key relationship is updated or deleted. There are five

RELATIONS IN ROOM

possibilities, denoted by ForeignKey constants:

Constant Name	If the Parent Is Updated or Deleted...
NO_ACTION	...do nothing
CASCADE	...update or delete the child
RESTRICT	...fail the parent's update or delete operation, unless there are no children
SET_NULL	...set the foreign key value to null
SET_DEFAULT	...set the foreign key value to the column(s) default value

NO_ACTION is the default, though CASCADE will be a popular choice for onDelete. In fact, we use CASCADE for onDelete in the Book entity's @ForeignKey, so if the Category is deleted, all of its associated Book rows are deleted from the books table.

Retrieving the Related Entities

For many things, our DAO can be no different than any other one we have seen so far. We can insert, update, delete, and query our entities as we see fit. For example, our Bookstore DAO has two @Insert functions to save categories and books:

```
@Insert
suspend fun save(category: Category)

@Insert
suspend fun save(vararg books: Book)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/onetomany/Bookstore.kt](#))

However, from a property standpoint, books and categories only have keys, not references to other entities. So, by default, if we have a @Query function that returns a Book, we have to execute a separate @Query function to look up its Category via the shortCode. And if we have a @Query function that returns a Category, we have to have another @Query function to retrieve all books associated with that Category.

To work around this limitation to some extent, we can use @Relation.

RELATIONS IN ROOM

A `@Query` does not have to return entities. As we saw earlier, it can return other things, such as an `Int` result from an aggregate function. So long as Room can figure out how to map the columns in your query result to properties of some return type, Room is happy.

So, we can declare a custom data class for a `@Query` response, such as this `CategoryAndBooks` class:

```
data class CategoryAndBooks(  
    @Embedded  
    val category: Category,  
    @Relation(  
        parentColumn = "shortCode",  
        entityColumn = "categoryShortCode"  
    )  
    val books: List<Book>  
)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/onetomany/Bookstore.kt](https://github.com/CommonWare/room-misc/blob/master/src/main/java/com/commonsware/room/misc/onetomany/Bookstore.kt))

By using `@Embedded`, any columns that we return from the `@Query` that can go in a `Category` will go into the `category` property.

The `@Relation` annotation says that, in addition to processing our direct query (from a `@Query` annotation), Room should automatically make a second query to retrieve related entities. Since our `@Relation` is tied to a property that is based around `Book`, Room knows that it needs to query our `books` table. The `parentColumn` and `entityColumn` annotation properties teach Room how to map data from our direct query result to `Book`. Specifically, Room should:

- Get the value of `shortCode` for a `Category` returned by the `@Query`, and
- Query the `books` table to find all rows where `categoryShortCode` matches that `shortCode` value

We can then use `CategoryAndBooks` in `@Query` functions:

```
package com.commonsware.room.misc.onetomany  
  
import androidx.room.*  
  
data class CategoryAndBooks(  
    @Embedded  
    val category: Category,  
    @Relation(  
        parentColumn = "shortCode",  
        entityColumn = "categoryShortCode"  
    )  
    val books: List<Book>  
)
```

RELATIONS IN ROOM

```
        parentColumn = "shortCode",
        entityColumn = "categoryShortCode"
    )
    val books: List<Book>
}

@Dao
interface Bookstore {
    @Insert
    suspend fun save(category: Category)

    @Insert
    suspend fun save(vararg books: Book)

    @Transaction
    @Query("SELECT * FROM categories")
    suspend fun loadAll(): List<CategoryAndBooks>

    @Transaction
    @Query("SELECT * FROM categories WHERE shortCode = :shortCode")
    suspend fun loadByShortCode(shortCode: String): CategoryAndBooks
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/onetomany/Bookstore.kt](#))

The `loadAll()` and `loadByShortCode()` functions return `CategoryAndBooks` objects, so we get categories and their associated books. Because Room needs to perform multiple queries for this, we use the `@Transaction` annotation to ensure that Room does all of those SQL calls inside a transaction.

We can then do things like create categories and books:

```
val category =
    Category(shortCode = "stuff", displayName = "Books About Stuff")
val bookOne = Book(
    isbn = "035650056X",
    title = "Feed",
    categoryShortCode = category.shortCode
)
val bookTwo = Book(
    isbn = "0451459792",
    title = "Dies the Fire",
    categoryShortCode = category.shortCode
)
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/onetomany/OneToManyTest.kt](#))

...save those to the database:

```
underTest.save(category)
underTest.save(bookOne, bookTwo)
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/onetomany/OneToManyTest.kt](#))

...and retrieve them later:

```
val all = underTest.loadAll()

assertThat(all, hasSize(equalTo(1)))
assertThat(all[0].category, equalTo(category))
assertThat(
    all[0].books,
    allOf(hasSize(equalTo(2)), hasElement(bookOne), hasElement(bookTwo))
)

val loaded = underTest.loadByShortCode(category.shortCode)

assertThat(loaded.category, equalTo(category))
assertThat(
    loaded.books,
    allOf(hasSize(equalTo(2)), hasElement(bookOne), hasElement(bookTwo))
)
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/onetomany/OneToManyTest.kt](#))

Representing No Relation

Sometimes, with one-to-many relations, the more correct model is “zero/one-to-many”. For example, perhaps a Book has not yet been assigned to a Category.

For that, simply make the foreign key property (e.g., categoryShortCode) be nullable, and let null represent the lack of a relationship.

Many-to-Many Relations

We can model one-to-many relations by having the “many” side (e.g., Book) have a foreign key back to its corresponding “one” item (e.g., Category).

The traditional way to model many-to-many relations is through a “join table”, where rows in that table represent the pairs of entities that are related. To change the relationships, you add or remove join table rows.

That is how Room handles many-to-many relations, with the assistance of a `@Junction` annotation.

Declaring the Join Table

Suppose that we want to say that a Book can be in more than one Category, such as “Android Programming Books” and “Books Written By Balding Men”. That now means that Book and Category have a many-to-many relationship, as we still want a Category to have many Book objects.

Our Category does not need to change:

```
package com.commonware.room.misc.manytomany

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "categoriesManyToMany")
data class Category(
    @PrimaryKey
    val shortCode: String,
    val displayName: String
)
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/manytomany/Category.kt](#))

Our Book no longer needs `categoryShortCode`, as that can only model a one-to-many relationship:

```
package com.commonware.room.misc.manytomany

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "booksManyToMany")
data class Book(
    @PrimaryKey
    val isbn: String,
    val title: String
)
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/manytomany/Book.kt](#))

However, we now need a third entity, to model the join table (here called `BookCategoryJoin`):

```
package com.commonware.room.misc.manytomany

import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.ForeignKey.CASCADE
import androidx.room.Index
import androidx.room.OnConflictStrategy

@Entity(
    primaryKeys = ["isbn", "shortCode"],
    indices = [Index("isbn"), Index("shortCode")],
    foreignKeys = [
        ForeignKey(
            entity = Book::class,
            parentColumns = arrayOf("isbn"),
            childColumns = arrayOf("isbn"),
            onDelete = CASCADE
        ), ForeignKey(
            entity = Category::class,
            parentColumns = arrayOf("shortCode"),
            childColumns = arrayOf("shortCode"),
            onDelete = CASCADE
        )
    ]
)
data class BookCategoryJoin(
    val isbn: String,
    val shortCode: String
)
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/manytomany/BookCategoryJoin.kt](#))

The BookCategoryJoin class has our keys: isbn to point to a Book and shortCode to point to a Category. Hence, each BookCategoryJoin instance (or row in its table) represents one relationship between a Book and Category.

The @Entity annotation is more complex:

- We use the primaryKeys property to say that the combination of isbn and shortCode is the primary key for our table
- We set up indices on each of those columns, as we will be querying this table a lot to find all categories for a book or all books for a category
- We have two @ForeignKey annotations, tying this class to Book and Category, and using onDelete = CASCADE to ensure that when we delete a Book or Category that its corresponding join entry gets deleted

Retrieving the Related Entities

In Bookstore, the `@Relation` annotation in `CategoryAndBooks` has a new property, `associateBy`, that contains a `@Junction` annotation:

```
data class CategoryAndBooks(  
    @Embedded  
    val category: Category,  
    @Relation(  
        parentColumn = "shortCode",  
        entityColumn = "isbn",  
        associateBy = Junction(  
            value = BookCategoryJoin::class,  
            parentColumn = "shortCode",  
            entityColumn = "isbn"  
        )  
    )  
    val books: List<Book>  
)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/manytomany/Bookstore.kt](#))

This teaches the `@Relation` about our join table and how to map columns in the query's result set to columns in the join table. This allows Room to be able to retrieve the books for a category. And, if we wanted, we could create a `BookAndCategories` class that handled the opposite case, wrapping a `Book` and a list of its associated `Category` objects.

We can then use `CategoryAndBooks` in our DAO functions:

```
@Transaction  
@Query("SELECT * FROM categoriesManyToMany")  
abstract suspend fun loadAll(): List<CategoryAndBooks>  
  
@Transaction  
@Query("SELECT * FROM categoriesManyToMany WHERE shortCode = :shortCode")  
abstract suspend fun loadByShortCode(shortCode: String): CategoryAndBooks
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/manytomany/Bookstore.kt](#))

The `@Transaction` annotations are there because we will wind up with multiple queries to populate our `CategoryAndBooks` objects, and Room will not automatically set up a database transaction for us to ensure that those queries all work off of the same edition of the data.

Room Entities as DTOs

This works, and it is not that difficult to set up. However, the resulting classes are tightly coupled to the underlying SQLite table structure and give us a clunky object model. A traditional object-oriented design usually does not involve separate objects representing relations, the way `BookCategoryJoin` does, for example.

In addition, not only is our code a bit clunky today, but it might need to change in the future in ways that make it worse. That could be due to changes in Room or due to changes in the data model (e.g., adding a `Library` and `BookLibraryJoin`).

For simple apps, you might just live with the odd object model. For larger apps, it may make sense to treat Room entities as “data transfer objects” (DTOs) that you convert into a preferred object model. For example:

- We could have `BookEntity`, `CategoryEntity`, etc. that model our tables
- We could have `Book` and `Category` as our object model, where `Book` and `Category` actually refer to each other
- A repository could use Room as a local data source and map between the Room-structured entity classes and the “pure” object model represented by `Book` and `Category`

You might already be doing some of this sort of mapping in other areas. For example, larger apps often choose to use an object model that is separate from the objects used in Web service calls, as the way the Web service API is structured might be unnatural in the client app. Plus, the Web service API might be modified by the server team, and it may be useful to minimize the impact of those changes on the majority of your application code.

The Support Database API

So far, this book has portrayed Room as being an ORM-style bridge between your code and SQLite.

Technically, that is not accurate.

The Room artifacts, such as `androidx.room:room-runtime`, have transitive dependencies on `androidx.sqlite:sqlite` and `androidx.sqlite:sqlite-framework`. Room itself talks to an abstraction around SQLite provided by `androidx.sqlite:sqlite`. This book will refer to this as “the support database API”. `androidx.sqlite:sqlite-framework` provides the default implementation of that abstraction, one that works with the built-in copy of `SQLiteDatabase` (part of the “framework”). When we use `RoomDatabase.Builder` to set up our `RoomDatabase`, we are using those “framework” classes for the database access.

In this chapter, we will explore in greater detail why this support database API exists and how we can use it, because while most of the time we will be able to use Room-generated code to work with the database, sometimes we cannot.

“Can’t You See That This is a Facade?”

To many developers, SQLite “is what it is”. Android ships with a SQLite implementation, and we use it, either directly or via some form of wrapper library.

However, in truth, there are many SQLite implementations. After all, SQLite is a library, and so there is nothing stopping people from using a separate, independent copy of SQLite from what is in Android. Even in Android itself, what SQLite you get depends on what device you run on:

- Different API levels integrate [different versions of SQLite](#)
- Device manufacturers sometimes [replace the stock SQLite version with another](#)

And so, sometimes, we need a [facade](#): an API that we can code to that supports a pluggable implementation. The following sections outline some examples.

Requery

[Requery](#) is a Room-like object mapping library, one that works both on Android and on the regular Java JVM. For plain Java (or Kotlin), Requery uses JDBC. For Android, Requery integrates with the support database API. Beyond that, Requery offers [its own implementation](#) of that API, wrapped around a standalone copy of SQLite. This ensures that you are using a current version of SQLite, even on older devices.

SQLCipher for Android

One of the best-known alternative SQLite implementations for Android is Zetitec's [SQLCipher for Android](#), which offers transparent encryption of database contents. As of [version 4.3.0](#), SQLCipher for Android offers an implementation of the support database API. This allows you to use encrypted databases from Room. We will explore this in greater detail [later in the book](#).

SQLDelight

Just as Requery is an ORM that uses the support database API, so is [SQLDelight](#). As with Requery, SQLDelight works on the JVM and Android. However, SQLDelight is a Kotlin/Multiplatform library, and it also supports Kotlin/Native for iOS and Windows. So, SQLDelight lets you write your database access code once and use it in multiple environments, and SQLDelight for Android lets you use the support database API so you can use the framework database, Requery's standalone SQLite, or SQLCipher for Android as your actual database implementation.

When Will We Use This?

There are two broad categories of scenarios where the support database API comes into play.

First is when you want to use a different SQLite implementation, such as wanting to use SQLCipher for Android. Then, as part of setting up your RoomDatabase, you can

provide it with the details of how to use that SQLite implementation, and Room will work with it.

In addition, there are other places in the Room API where the Room abstractions break down and the support database API peeks through, such as:

- When you need to [migrate a database](#) from one schema to another
- When you need to configure your database in ways beyond what Room supports, such as [directly invoking PRAGMA statements](#)

Configuring Room's Database Access

We have used RoomDatabase to set up our database and get access to our DAO(s) for working with our entities. By default, RoomDatabase will use the “framework” implementation of the support database APIs. However:

- We can tell it to use something else
- We can get control as part of the database setup, to configure the database manually, regardless of what support database API implementation we use

Get a Factory

With the framework's Android SQLite API, many developers elect to use SQLiteOpenHelper as their entry point. This handles creating and upgrading the database in a decent structured fashion. However, SQLiteOpenHelper is not a requirement — developers could use static methods on SQLiteDatabase, such as openOrCreateDatabase(), to work with a SQLiteDatabase without an associated SQLiteOpenHelper.

The equivalent interface to SQLiteOpenHelper in the support database API is SupportSQLiteOpenHelper. However, with the support database API, working with a SupportSQLiteOpenHelper is unavoidable. Whether you use it, or Room uses it, *somebody* sets up one of these. SupportSQLiteOpenHelper fills a role similar to that of SQLiteOpenHelper, providing a single point of control for creating and upgrading a database.

However, you do not create a SupportSQLiteOpenHelper directly yourself. Instead, you ask a SupportSQLiteOpenHelper.Factory to do that for you. Each implementation of the support database API should have a class that implements the SupportSQLiteOpenHelper.Factory interface:

THE SUPPORT DATABASE API

- The default Room implementation is `FrameworkSQLiteOpenHelperFactory`, from the `androidx.sqlite:sqlite-framework` artifact
- SQLCipher for Android has `SupportFactory`
- Requery has `RequerySQLiteOpenHelperFactory`
- And so on

How you get an instance of that factory is up to the implementation of the support database API. In the case of `FrameworkSQLiteOpenHelperFactory`, you just create an instance via a no-parameter constructor. SQLCipher for Android offers three `SupportFactory` constructors, where the passphrase is among the various parameters.

Regardless, one way or another, you will need to get an instance of a factory.

You can use the factory directly, bypassing all of Room. Other times, you will want to use Room, but have Room use this support database API implementation.

For that, call `openHelperFactory()` on the `RoomDatabase.Builder` as part of setting it up:

```
val db = Room.databaseBuilder(ctxt, StuffDatabase.class, DB_NAME)
    .openHelperFactory(SupportFactory(passphrase))
    .build()
```

Here, we are having Room use `SupportFactory` from SQLCipher for Android. Room will now use SQLCipher for Android, via `SupportFactory`, for all of its actual database I/O.

We will examine SQLCipher for Android, and its use with Room, more [later in the book](#).

Add a Callback

Regardless of whether we use `openHelperFactory()` or not, we can also call `addCallback()` on the `RoomDatabase.Builder` to supply a `RoomDatabase.Callback` to use. This callback can get control at two points:

- When the database file is created, via an `onCreate()` function on the callback
- When the database file is opened, via an `onOpen()` function on the callback

THE SUPPORT DATABASE API

In each case, you get a `SupportSQLiteDatabase` object to use for manipulating the database. Room itself may not be completely ready for use — particularly in the `onCreate()` callback — which is why you are not passed your `RoomDatabase` subclass. Instead, you have to work with the database using the support database API directly.

We will see examples of this, in the context of running some PRAGMA statements, [later in the book](#).

Database Migrations

When you first ship your app, you think that your database schema is beautiful, a true work of art.

Then, you wake up the next morning and realize that you need to make changes to that schema.

During initial development — and for silly little book examples — you just go in and make changes to your entities, and Room will rebuild your database for you. However, it does so by dropping all of your existing tables, taking all the data with it. In development, that may not be so bad. In *production*... well, users get somewhat irritated when you lose their data.

And that is where migrations come into play.

What's a Migration?

As mentioned in [the preceding chapter](#), with traditional Android SQLite development, we typically use `SQLiteOpenHelper`. This utility class manages a `SQLiteDatabase` for us and addresses two key problems:

1. What happens when our app first runs on a device — or after the user has cleared our app's data — and we have no database at all?
2. What happens when we need to modify the database schema from what it was to some new structure?

`SQLiteOpenHelper` accomplishes this by calling `onCreate()` and `onUpgrade()` callbacks, where we could implement the logic to create the tables and adjust them as the schemas change.

While `onCreate()` worked reasonably well, `onUpgrade()` could rapidly grow out of control. Long-lived apps might have dozens of different schemas, evolving over time. Because users are not forced to take on app updates, our apps need to be able to transition from any prior schema to the latest-and-greatest one. This meant that `onUpgrade()` would need to identify exactly what bits of code are needed to migrate the database from the old to the new version, and this could get unwieldy.

Room addresses this somewhat through the `Migration` class. You create subclasses of `Migration` — typically as Kotlin object implementations — that handle the conversion from some older schema to a newer one. You pass a bunch of `Migration` instances to Room, representing different pair-wise schema upgrade paths. Room then determines which one(s) need to be used at any point in time, to update the schema from whatever it was to whatever it needs to be.

When Do We Migrate?

On our `RoomDatabase` subclass, we have a `@Database` annotation. One of the properties is `version`. This works like the version code that we would pass into the `SQLiteOpenHelper` constructor. It is a monotonically increasing integer, with higher numbers indicating newer schemas. The version in the code represents the schema version that this code is expecting.

Once your app ships, any time you change your schema — mostly in the form of modifying entity classes — you need to increment that version and create a `Migration` that knows how to convert from the prior version to this new one.

Note that there is no requirement that you increment the version by 1, though that is a common convention. If using a date-based format like `YYYYMMDD` (e.g., `20170627`) makes your life easier, you are welcome to do so!

But First, a Word About Exporting Schemas

One of the side-effects of using Room is that you do not write your own schema for the database. Room generates it, based on your entity definitions. During the ordinary course of programming, this is perfectly fine and saves you time and effort.

However, when it comes to migrations, now we have a problem. We cannot create code to migrate from an old to a new schema without knowing what those schemas are. And while schema information is baked into some code generated by Room's annotation processor, that is only for the current version of your entity classes (and,

DATABASE MIGRATIONS

hence, your current schema), not for any historical ones.

Fortunately, Room offers something that helps a bit: exported schemas. You can teach Room's annotation processor to not only generate Java code but also generate a JSON document describing the schema. Moreover, it will do that for each schema version, saving them to version-specific JSON files. If you hold onto these files — for example, if you save them in Git or some other form of version control — you will have a history of your schema and can use that information to write your migrations.

However, the real reason for those exported schemas is to help with testing your migrations. As a result, the JSON format is not designed for developers to read.

To set this up, in the `defaultConfig` closure of your module's `build.gradle` file, you can add a `javaCompileOptions` closure:

```
defaultConfig {
    minSdkVersion 21
    targetSdkVersion 30
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"

    javaCompileOptions {
        annotationProcessorOptions {
            arguments = ["room.schemaLocation": "$projectDir/schemas".toString()]
        }
    }
}
```

(from [Migration/build.gradle](#))

This teaches Room to save your schemas in a `schemas/` directory off of the module root directory. In principle, you could store them elsewhere by choosing a different value for the `room.schemaLocation` argument.

The next time you (re-)build your project, that directory will be created. Subdirectories with the fully-qualified class names of your `RoomDatabase` classes will go inside there, and inside each of those will be a JSON file named after your schema version (e.g., `1.json`):

```
{
  "formatVersion": 1,
  "database": {
    "version": 1,
    "identityHash": "051fc3ca1ecb3344055fd77365a9bf8e",
    "entities": [
      {
        "tableName": "notes",
```

DATABASE MIGRATIONS

```
"createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT NOT NULL, `title` TEXT NOT NULL, `text` TEXT NOT NULL, `version` INTEGER NOT NULL, PRIMARY KEY(`id`))",
"fields": [
  {
    "fieldPath": "id",
    "columnName": "id",
    "affinity": "TEXT",
    "notNull": true
  },
  {
    "fieldPath": "title",
    "columnName": "title",
    "affinity": "TEXT",
    "notNull": true
  },
  {
    "fieldPath": "text",
    "columnName": "text",
    "affinity": "TEXT",
    "notNull": true
  },
  {
    "fieldPath": "version",
    "columnName": "version",
    "affinity": "INTEGER",
    "notNull": true
  }
],
"primaryKey": {
  "columnNames": [
    "id"
  ],
  "autoGenerate": false
},
"indices": [],
"foreignKeys": []
}
],
"views": [],
"setupQueries": [
  "CREATE TABLE IF NOT EXISTS room_master_table (id INTEGER PRIMARY KEY,identity_hash TEXT)",
  "INSERT OR REPLACE INTO room_master_table (id,identity_hash) VALUES(42, '051fc3ca1ecb3344055fd77365a9bf8e')"
]
}
```

(from [Migration/schemas/com.commonware.room.migration.NoteDatabase/L.json](#))

The JSON properties that will matter to you will be the createSql ones. There are ones that create your tables and others that create your indexes. This is fairly normal SQL, except that:

- The table name is injected at runtime, replacing the `\${TABLE_NAME}` placeholder
- Backticks are wrapped around column names

Writing Migrations

A Migration itself has only one required method: `migrate()`. You are given a `SupportSQLiteDatabase`, from the support database API covered in [the preceding chapter](#). You can use the `SupportSQLiteDatabase` to execute whatever SQL statements you need to change the database schema to what you need.

The Migration constructor takes two parameters: the old schema version number and the new schema version number. You will only ever need one instance of a Migration for a given schema version pair, and usually the `migrate()` implementation for that schema version pair will be unique. Hence, the typical pattern is to implement each Migration as a Kotlin object, where you can provide the `migrate()` function to use for migrating the schema between that particular pair of schema versions.

To determine what needs to be done, you need to examine that schema JSON and determine what is different between the old and the new. Someday, we may get some tools to help with this. For now, you are largely stuck “eyeballing” the SQL. You can then craft the `ALTER TABLE` or other statements necessary to change the schema, much as you might have done in `onUpgrade()` of a `SQLiteOpenHelper`.

For example, [the Migration module](#) of [the book’s primary sample project](#) has a `NoteDatabase` akin to the `NoteBasics` module that we saw earlier in the book. One difference is that we have a `MIGRATION_1_2` object that implements a Migration:

```
@VisibleForTesting
internal val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE notes ADD COLUMN andNowForSomethingCompletelyDifferent TEXT")
    }
}
```

(from [Migration/src/main/java/com/commonsware/room/migration/NoteDatabase.kt](#))

Our `migrate()` function executes an `ALTER TABLE` statement, using `execSQL()` on the supplied `SupportSQLiteDatabase` object. The `MIGRATION_1_2` object supplies 1 and 2 as the schema version pair to the Migration constructor, to identify what version pair this `migrate()` will handle.

Employing Migrations

Simply creating a Migration as an object somewhere is necessary but not sufficient

DATABASE MIGRATIONS

to have Room know about performing the migration. Instead, you need to use the `addMigrations()` method on `RoomDatabase.Builder` to teach Room about your Migration objects. `addMigrations()` accepts a varargs, and so you can pass in one or several Migration objects as needed.

```
package com.commonware.room.migration

import android.content.Context
import androidx.annotation.VisibleForTesting
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.migration.Migration
import androidx.sqlite.db.SupportSQLiteDatabase

@Database(entities = [NoteEntity::class], version = 2)
abstract class NoteDatabase : RoomDatabase() {
    companion object {
        fun newTestDatabase(context: Context) = Room.inMemoryDatabaseBuilder(
            context,
            NoteDatabase::class.java
        )
            .addMigrations(MIGRATION_1_2)
            .build()
    }

    abstract fun notes(): NoteStore
}

@VisibleForTesting
internal val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE notes ADD COLUMN andNowForSomethingCompletelyDifferent TEXT")
    }
}
```

(from [Migration/src/main/java/com/commonware/room/migration/NoteDatabase.kt](#))

This version of `NoteDatabase` has a companion object with a `newTestDatabase()` function that our tests can use. As part of building the `NoteDatabase` instance, we use `addMigrations(MIGRATION_1_2)` to teach Room about our Migration.

Also, note that the version in the `@Database` annotation is 2. In theory, this module demonstrates an app that has been modified from its original. The code started with the `NoteBasics` edition of `NoteEntity`:

```
package com.commonware.room.notes

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "notes")
```

DATABASE MIGRATIONS

```
data class NoteEntity(  
    @PrimaryKey val id: String,  
    val title: String,  
    val text: String,  
    val version: Int  
)
```

(from [NoteBasics/src/main/java/com/commonsware/room/notes/NoteEntity.kt](#))

The Migration module added a new nullable property to NoteEntity:

```
package com.commonsware.room.migration  
  
import androidx.room.Entity  
import androidx.room.PrimaryKey  
  
@Entity(tableName = "notes")  
data class NoteEntity(  
    @PrimaryKey val id: String,  
    val title: String,  
    val text: String,  
    val version: Int,  
    val andNowForSomethingCompletelyDifferent: String?  
)
```

(from [Migration/src/main/java/com/commonsware/room/migration/NoteEntity.kt](#))

That property is the column that we are adding in MIGRATION_1_2, to handle a user that had been using our app with schema version 1 and now upgrades to a newer copy of our app that uses schema version 2.

In principle, MIGRATION_1_2 could be private to NoteDatabase. However, we want to be able to [test our migration](#), so we have it marked as internal instead, with the `@VisibleForTesting` annotation to help discourage unexpected use.

How Room Applies Migrations

When you create your RoomDatabase instance via the Migration-enhanced Builder, Room will use SQLiteOpenHelper semantics to see if the schema version in the existing database is older than the schema version that you declared in your `@Database` annotation. If it is, Room will try to find a suitable Migration to use, falling back to dropping all of your tables and rebuilding them from scratch, as happens during ordinary development.

Much of the time, the schema will jump from one version to the next. If you are using a simple numbering scheme starting at 1, the schema will then move to 2, then 3, then 4, and so on, for a given device. Hence, your primary Migration objects will be ones that implement these incremental migrations.

However, some user might have skipped some app updates, so you need to skip a schema version as part of an upgrade (e.g., go from schema version 1 to schema version 3). Room is smart enough to find a chain of Migration objects to use, and so if you have Migration objects for each incremental schema change, Room can handle any combination of changes. For example, to go from 1 to 3, Room might first use your (1, 2) migration, then the (2, 3) migration.

Sometimes, though, this can lead to unnecessary work. Suppose in schema version 2, you created a bunch of new tables and stuff... then reverted those changes in schema version 3. By using the incremental migrations, Room will create those tables and then turn around and drop them right away.

However, all else being equal, Room will try to use the shortest possible chain. Hence, you can create additional Migration objects where appropriate to streamline particular upgrades. You could create a (1, 3) migration that bypasses the obsolete schema version 2, for example. This is optional but may prove useful from time to time.

Testing Migrations

It would be nice if your migrations worked. Users, in particular, appreciate working code... or, perhaps more correctly, they get rather angry with non-working code.

Hence, you might want to test the migrations.

This gets a bit tricky, though. The code-generated Room classes are expecting the latest-and-greatest schema version, so you cannot use your DAO for testing older schemas. Besides, RoomDatabase.Builder wants to set up your database with that latest-and-greatest schema automatically.

Fortunately, Room ships with some testing code to help you test your schemas in isolation... though you bypass most of Room to do that.

Adding the Artifact

This testing code is in a separate `androidx.room:room-testing` artifact, one that you can add via `androidTestCompile` to put in your instrumentation tests but leave out of your production code:

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation "androidx.arch.core:core-runtime:2.1.0"
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
    androidTestImplementation 'androidx.test:runner:1.4.0'
    androidTestImplementation "androidx.test.ext:junit:1.1.3"
    androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
    androidTestImplementation "androidx.room:room-testing:$room_version"
    androidTestImplementation "com.natpryce:hamkrest:1.7.0.0"
}
```

(from [Migration/build.gradle](#))

Adding the Schemas

Remember those exported schemas? While we used them for helping us write the migrations, their primary use is for this testing support code.

By default, those schemas are stored outside of anything that goes into your app. After all, you do not need those JSON files cluttering up your production builds. However, this also means that those schemas are not available to your test code, by default.

However, we can fix that, by adding those schemas to the `assets/` used in the `androidTest` source set, by having this closure in your `android` closure of your module's `build.gradle` file:

```
sourceSets {
    androidTest.assets.srcDirs += files("$projectDir/schemas".toString())
}
```

(from [Migration/build.gradle](#))

Here, `"$projectDir/schemas".toString()` is the same value that we used for the `room.schemaLocation` annotation processor argument. This snippet tells Gradle to include the contents of that `schemas/` directory as part of our `assets/`.

The result is that our instrumentation test APK will have those directories named after our RoomDatabase classes (e.g., `com.commonware.room.migration.NoteDatabase/`) in the root of `assets/`. If you have code that uses `assets/`, make sure that you are taking steps to ignore these extra directories.

Creating a MigrationTestHelper

The testing support comes in the form of a `MigrationTestHelper` that you can employ in your instrumentation tests.

`MigrationTestHelper` is a JUnit4 rule, which you add to your test case class via the `@Rule` annotation:

```
@get:Rule
val migrationTestHelper = MigrationTestHelper(
    InstrumentationRegistry.getInstrumentation(),
    NoteDatabase::class.java.canonicalName
)
```

(from [Migration/src/androidTest/java/com/commonware/room/migration/MigrationTest.kt](#))

The `MigrationTestHelper` constructor takes two parameters, both of which are a bit unusual.

First, it takes an `Instrumentation` object. We use those in our test code, but it is rare that we pass them as a parameter. You get your `Instrumentation` by calling `getInstrumentation()` on the `InstrumentationRegistry`.

Then, it takes what appears to be the fully-qualified class name of the RoomDatabase whose migrations we wish to test. Technically speaking, this is actually the relative path, inside of `assets/`, where the schema JSON files are for this particular RoomDatabase. Given the above configuration, each database's schemas are put into a directory named after the fully-qualified class name of the RoomDatabase, which is why this works. However, if you change the configuration to put the schemas somewhere else in `assets/`, you would need to modify this parameter to match.

Creating a Database for a Schema Version

There are two main methods on `MigrationTestHelper` that we will use in testing. One is `createDatabase()`. This creates the database, as a specific database file, for a specific schema version... including any of our historical ones found in those schema

DATABASE MIGRATIONS

JSON files. Here, we ask the helper to create a database named `DB_NAME` for schema version 1:

```
val initialDb = migrationTestHelper.createDatabase(DB_NAME, 1)
```

(from [Migration/src/androidTest/java/com/commonsware/room/migration/MigrationTest.kt](#))

This gives you a `SupportSQLiteDatabase`, not a Room database (e.g., `NoteDatabase`). That is because you may be using a historical schema version, and the Room-generated code only exists for the most recent schema version.

As part of testing a migration, you will need to add some sample data to the database, using whatever schema you asked to be used, so that you can confirm that the migration worked as expected and did not wreck the existing data. This code will not be very Room-ish, but more like classic SQLite Android programming:

```
val initialDb = migrationTestHelper.createDatabase(DB_NAME, 1)

initialDb.execSQL(
    "INSERT INTO notes (id, title, text, version) VALUES (?, ?, ?, ?)",
    arrayOf(TEST_ID, TEST_TITLE, TEST_TEXT, TEST_VERSION)
)

initialDb.query("SELECT COUNT(*) FROM notes").use {
    assertThat(it.count, equalTo(1))
    it.moveToFirst()
    assertThat(it.getInt(0), equalTo(1))
}

initialDb.close()
```

(from [Migration/src/androidTest/java/com/commonsware/room/migration/MigrationTest.kt](#))

Testing a Migration

The other method of note on `MigrationTestHelper` is `runMigrationsAndValidate()`. After you have set up a database in its starting conditions via `createDatabase()` and CRUD operations, `runMigrationsAndValidate()` will migrate that database from its original schema version to the one that you specify:

```
val db = migrationTestHelper.runMigrationsAndValidate(
    DB_NAME,
    2,
```

DATABASE MIGRATIONS

```
true,  
MIGRATION_1_2  
)
```

(from [Migration/src/androidTest/java/com/commonsware/room/migration/MigrationTest.kt](#))

You need to supply the same database name (DB_NAME), a higher schema version (2), and the specific Migration that you want to use (MIGRATION_1_2).

Not only does this method perform the migration, but it validates the resulting schema against what the entities have set up for that schema version, based on the schema JSON files. If there is something wrong — your migration forgot a newly-added column, for example — your test will fail with an assertion violation. The `true` parameter shown above determines whether this schema validation will be checked for un-dropped tables. `true` means that if you have unnecessary tables in the database, the test fails; `false` means that unnecessary tables are fine and will be ignored.

However, all `MigrationTestHelper` can do is confirm that you set up the new schema properly and give you a `SupportSQLiteDatabase` representing the migrated database. It cannot determine whether the data is any good after the migration. That you would need to test yourself:

```
val db = migrationTestHelper.runMigrationsAndValidate(  
    DB_NAME,  
    2,  
    true,  
    MIGRATION_1_2  
)  
  
db.query("SELECT id, title, text, version, andNowForSomethingCompletelyDifferent FROM notes")  
    .use {  
        assertThat(it.count, equalTo(1))  
        it.moveToFirst()  
        assertThat(it.getInt(0), equalTo(TEST_ID))  
        assertThat(it.getString(1), equalTo(TEST_TITLE))  
        assertThat(it.getString(2), equalTo(TEST_TEXT))  
        assertThat(it.getInt(3), equalTo(TEST_VERSION))  
        assertThat(it.getString(4), absent())  
    }
```

(from [Migration/src/androidTest/java/com/commonsware/room/migration/MigrationTest.kt](#))

In many cases, there is little to test, particularly if you are just setting up empty tables as we are doing in this migration. However, if you had a complex table change, perhaps requiring a temp table and statements like `INSERT INTO ... SELECT FROM ...`, you could write test code that confirms the data is OK. However, as shown above, you cannot use the Room DAO for this either. Instead, you will use the

DATABASE MIGRATIONS

Support `SQLiteDatabase` and work with the tables “the old-fashioned way”, using `query()` and `Cursor` and similar constructs.

Intermediate Room

Polymorphic Entities

Java and Kotlin programmers are used to polymorphism, where you can treat objects as being of the same type, when in truth their concrete types differ. This could be based on a common interface or a common base class (abstract or otherwise).

Those used to putting data into SQL databases are used to the fact that polymorphism and a relational database do not work together naturally. This is just “one of those things” that developers have to deal with, as part of “object-relational impedance mismatch”.

There are a few strategies for dealing with polymorphic relations in relational databases. This chapter outlines two of them, with an eye towards how they can be implemented with Room.

Polymorphism With Separate Tables

One approach uses a separate table for instances of each concrete type. So, for example if we have a `CommentEntity` class and a `LinkEntity` class, and they both implement a common `Note` interface, we wind up with dedicated tables for `CommentEntity` and `LinkEntity`. This keeps the database structure simple, as we still have a 1:1 relationship between concrete class and table. However, it means that any persistence code that deals with `Note` objects needs to handle the fact that a `Note` is stored differently for different `Note` implementations, and that can add complexity.

The [the MiscSamples module](#) of [the book’s primary sample project](#) has a poly sub-package with classes that implement this strategy.

As depicted in the preceding paragraph, we have a `Note` interface:

POLYMORPHIC ENTITIES

```
package com.commonware.room.misc.poly

interface Note {
    val displayText: CharSequence
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/poly/Note.kt](#))

We also have CommentEntity and LinkEntity classes that implement that interface and have slightly different contents:

```
package com.commonware.room.misc.poly

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "comments")
data class CommentEntity(
    @PrimaryKey
    val id: Long,
    val text: String
) : Note {
    override val displayText: CharSequence
        get() = text
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/poly/CommentEntity.kt](#))

```
package com.commonware.room.misc.poly

import androidx.core.text.HtmlCompat
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "links")
data class LinkEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Long,
    val title: String,
    val url: String
) : Note {
    override val displayText: CharSequence
        get() = HtmlCompat.fromHtml(
            """<a href="$url">$title</a>""",
            HtmlCompat.FROM_HTML_MODE_COMPACT
        )
}
```

POLYMORPHIC ENTITIES

(from [MiscSamples/src/main/java/com/commonsware/room/misc/poly/LinkEntity.kt](#))

All Note wants is a `displayName` `CharSequence` with a display representation of the note. `Comment` just holds some text (`text`), so the `displayName` is just the text. `Link` has a URL and a title, so the `displayName` is a clickable rendition of that link, here generated by a snippet of HTML and `HtmlCompat`. A more efficient, but less readable, approach would be to use a `ClickableSpan` with a `SpannableStringBuilder` to create this clickable link.

`PolyStore` is our DAO for manipulating these objects. That needs to have functions for working with our two entity classes:

```
@Query("SELECT * FROM comments")
fun allComments(): List<CommentEntity>

@Insert
fun insert(vararg comments: CommentEntity)

@Query("SELECT * FROM links")
fun allLinks(): List<LinkEntity>

@Insert
fun insert(vararg links: LinkEntity)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/poly/PolyStore.kt](#))

However, consumers of `PolyStore` might prefer to work with `Note` objects, ignoring the details of whether those notes are comments or links. For that, we have to write our own concrete functions:

```
@Transaction
fun allNotes() = allComments() + allLinks()

@Transaction
fun insert(vararg notes: Note) {
    insert(*notes.filterIsInstance(CommentEntity::class.java).toTypedArray())
    insert(*notes.filterIsInstance(LinkEntity::class.java).toTypedArray())
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/poly/PolyStore.kt](#))

`allNotes()` simply combines the results of the `allComments()` and `allLinks()` functions. The `Note` variant of `insert()` turns around and calls the `CommentEntity` and `LinkEntity` variants of `insert()`, finding the `CommentEntity` and `LinkEntity` instances in our `vararg` of `Note` objects. The fact that we are using `vararg` makes

POLYMORPHIC ENTITIES

`insert()` a bit complicated, with spread operators (*) and `toTypedArray()` calls. If we were using `List` instead of `vararg`, we would have somewhat simpler code:

```
@Transaction
fun insert(notes: List<Note>) {
    insert(notes.filterIsInstance(CommentEntity::class.java))
    insert(notes.filterIsInstance(LinkEntity::class.java))
}
```

Having the `Note` functions on `PolyStore` allows us to work with the concrete types or the `Note` interface, as we see fit:

```
package com.commonware.room.misc.poly

import androidx.room.Room
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import com.commonware.room.misc.MiscDatabase
import com.natpryce.hamkrest.*
import com.natpryce.hamkrest.assertion.assertThat
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class PolyStoreTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.polyStore()

    @Test
    fun comments() {
        assertThat(underTest.allComments(), isEmpty)

        val firstComment = CommentEntity(1, "This is a comment")
        val secondComment = CommentEntity(2, "This is another comment")

        underTest.insert(firstComment, secondComment)

        assertThat(
            underTest.allComments(),
            allOf(
                hasSize(equalTo(2)),
                hasElement(firstComment),
                hasElement(secondComment)
            )
        )
    }
}
```

```
    )
  )

  assertThat(
    underTest.allNotes(),
    listOf(
      hasSize(equalTo(2)),
      hasElement(firstComment as Note),
      hasElement(secondComment as Note)
    )
  )
}

@Test
fun links() {
  assertThat(underTest.allLinks(), isEmpty)

  val firstLink = LinkEntity(1, "CommonsWare", "https://commonsware.com")
  val secondLink = LinkEntity(
    2,
    "Room Release Notes",
    "https://developer.android.com/jetpack/androidx/releases/room"
  )

  underTest.insert(firstLink, secondLink)

  assertThat(
    underTest.allLinks(),
    listOf(
      hasSize(equalTo(2)),
      hasElement(firstLink),
      hasElement(secondLink)
    )
  )

  assertThat(
    underTest.allNotes(),
    listOf(
      hasSize(equalTo(2)),
      hasElement(firstLink as Note),
      hasElement(secondLink as Note)
    )
  )
}

@Test
fun notes() {
  assertThat(underTest.allNotes(), isEmpty)
```

POLYMORPHIC ENTITIES

```
val firstComment = CommentEntity(1, "This is a comment")
val secondComment = CommentEntity(2, "This is another comment")
val firstLink = LinkEntity(1, "CommonsWare", "https://commonsware.com")
val secondLink = LinkEntity(
    2,
    "Room Release Notes",
    "https://developer.android.com/jetpack/androidx/releases/room"
)

underTest.insert(firstComment, secondComment, firstLink, secondLink)

assertThat(
    underTest.allNotes(), allOf(
        hasSize(equalTo(4)),
        hasElement(firstLink as Note),
        hasElement(secondLink as Note),
        hasElement(firstComment as Note),
        hasElement(secondComment as Note)
    )
)
}
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/poly/PolyStoreTest.kt](#))

Can I JOIN a UNION?

You might think that we could create `allNotes()` using the UNION support in SQLite. This basically allows you to concatenate two queries and combine their results.

The theory would be that you could do something like this:

```
@Query("SELECT * FROM links UNION ALL SELECT * FROM comments")
fun allNotes(): List<Note>
```

However, this will not work.

In this specific case, links and comments do not have the same columns, as our entities have different fields. This runs afoul of UNION regulations, as at minimum, both halves of the UNION have to return the same number of columns.

Beyond that, Room has no way to know which rows are links and which rows are comments, as there is nothing to distinguish them in the result set.

Finally, Room cannot create instances of Note, as that is an interface, and we want LinkEntity and CommentEntity objects anyway. That would require Room to not only know which rows are links and which are comments, but that rows that are links should be turned into LinkEntity objects and that rows that are comments should be turned into CommentEntity objects.

From a practical standpoint, both entities would need to have the same properties and resulting schema. The result set (embodied in a Cursor) has only one set of column names, based on the first query in the UNION. Room would need to be able to determine how to populate entities from the second query using the first query's column names. In all likelihood, that would require the names to be the same in both queries and in both entities.

Due to these limitations, it is unlikely that Room will get this capability, though it is not impossible.

Polymorphism With a Single Table

We could go the other route: have a single table for all note objects, regardless of whether they are a comment or a link. For small objects with few properties, with a lot of overlap between the properties of the concrete types, this is manageable. It becomes unwieldy for many concrete types with many disparate properties. It also puts limits on your SQL, as the only practical NOT NULL columns are ones for which you can supply values for every possible concrete type. You also need some way of determining what concrete type to use for any given table row, and often that requires yet *another* column.

But, it *is* an option.

The polysingle sub-package in [the MiscSamples module](#) demonstrates this approach. This time, the entity is NoteEntity:

```
package com.commonware.room.misc.polysingle

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "notes")
open class NoteEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Long,
```


POLYMORPHIC ENTITIES

```
    val title: String,  
    var url: String?  
)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/polysingle/NoteEntity.kt](#))

It contains a superset of all columns from both our comments and links. In this case, that just means that the URL is optional — a link has one, but a comment does not. This approach will get a lot more messy if you have lots of entities and lots of columns that exist only in a subset of those entity types, though.

Now, Comment and Link are subclasses of NoteEntity, implementing the Note interface from the poly package and overriding `displayText` as needed for their scenarios:

```
package com.commonsware.room.misc.polysingle  
  
import com.commonsware.room.misc.poly.Note  
  
class Comment(id: Long, title: String) : NoteEntity(id, title, null), Note {  
    override val displayText: CharSequence  
        get() = title  
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/polysingle/Comment.kt](#))

```
package com.commonsware.room.misc.polysingle  
  
import androidx.core.text.HtmlCompat  
import com.commonsware.room.misc.poly.Note  
  
class Link(  
    id: Long,  
    title: String,  
    url: String  
) : NoteEntity(id, title, url), Note {  
    override val displayText: CharSequence  
        get() = HtmlCompat.fromHtml(  
            ""<a href="$url">$title</a>"" ,  
            HtmlCompat.FROM_HTML_MODE_COMPACT  
        )  
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/polysingle/Link.kt](#))

You might argue that NoteEntity should be abstract and define `displayText` there. That could work, at the cost of not being able to load NoteEntity objects directly, as

POLYMORPHIC ENTITIES

Room cannot create instances of an abstract class.

PolySingleStore — the DAO for this scenario — is a bit simpler. We do not need dedicated `insert()` functions for `Link` and `Comment`, as an `insert()` function for `NoteEntity` covers both of those cases. `allLinks()` and `allComments()` can take advantage of Room's return type flexibility, having Room create `Link` and `Comment` objects directly, with our query returning the proper rows based on whether we have a url or not:

```
package com.commonware.room.misc.polysingle

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query

@Dao
interface PolySingleStore {
    @Query("SELECT * FROM notes")
    fun allNotes(): List<NoteEntity>

    @Insert
    fun insert(vararg notes: NoteEntity)

    @Query("SELECT * FROM notes WHERE url IS NOT NULL")
    fun allLinks(): List<Link>

    @Query("SELECT id, title FROM notes WHERE url IS NULL")
    fun allComments(): List<Comment>
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/polysingle/PolySingleStore.kt](#))

And, once again, we have sufficient CRUD operations now to be able to manipulate links and comments separately or treating them all as notes:

```
package com.commonware.room.misc.polysingle

import androidx.room.Room
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import com.commonware.room.misc.MiscDatabase
import com.natpryce.hamkrest.anyOf
import com.natpryce.hamkrest.assertion.assertThat
import com.natpryce.hamkrest.equalTo
import com.natpryce.hamkrest.hasSize
import com.natpryce.hamkrest.isEmpty
```

POLYMORPHIC ENTITIES

```
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class PolySingleStoreTest {
    private val db = Room.inMemoryDatabaseBuilder(
        InstrumentationRegistry.getInstrumentation().targetContext,
        MiscDatabase::class.java
    )
        .build()
    private val underTest = db.polySingleStore()

    @Test
    fun comments() {
        assertThat(underTest.allComments(), isEmpty())

        val firstComment = Comment(1, "This is a comment")
        val secondComment = Comment(2, "This is another comment")

        underTest.insert(firstComment, secondComment)

        val allComments = underTest.allComments()

        assertThat(allComments, hasSize(equalTo(2)))
        assertThat(
            allComments[0].title,
            anyOf(equalTo(firstComment.title), equalTo(secondComment.title))
        )
        assertThat(
            allComments[1].title,
            anyOf(equalTo(firstComment.title), equalTo(secondComment.title))
        )

        val allNotes = underTest.allNotes()

        assertThat(allNotes, hasSize(equalTo(2)))
        assertThat(
            allNotes[0].title,
            anyOf(equalTo(firstComment.title), equalTo(secondComment.title))
        )
        assertThat(
            allNotes[1].title,
            anyOf(equalTo(firstComment.title), equalTo(secondComment.title))
        )
    }

    @Test
    fun links() {
```

```
assertThat(underTest.allLinks(), isEmpty)

val firstLink = Link(1, "CommonsWare", "https://commonsware.com")
val secondLink = Link(
    2,
    "Room Release Notes",
    "https://developer.android.com/jetpack/androidx/releases/room"
)

underTest.insert(firstLink, secondLink)

val allLinks = underTest.allLinks()

assertThat(allLinks, hasSize(equalTo(2)))
assertThat(
    allLinks[0].title,
    anyOf(equalTo(firstLink.title), equalTo(secondLink.title))
)
assertThat(
    allLinks[1].title,
    anyOf(equalTo(firstLink.title), equalTo(secondLink.title))
)

val allNotes = underTest.allNotes()

assertThat(allNotes, hasSize(equalTo(2)))
assertThat(
    allNotes[0].title,
    anyOf(equalTo(firstLink.title), equalTo(secondLink.title))
)
assertThat(
    allNotes[1].title,
    anyOf(equalTo(firstLink.title), equalTo(secondLink.title))
)
}

@Test
fun notes() {
    assertThat(underTest.allNotes(), isEmpty)

    val firstComment = Comment(1, "This is a comment")
    val secondComment = Comment(2, "This is another comment")
    val firstLink = Link(3, "CommonsWare", "https://commonsware.com")
    val secondLink = Link(
        4,
        "Room Release Notes",
        "https://developer.android.com/jetpack/androidx/releases/room"
    )
}
```

```
underTest.insert(firstComment, secondComment, firstLink, secondLink)

val allNotes = underTest.allNotes()

assertThat(allNotes, hasSize(equalTo(4)))
assertThat(
    allNotes[0].title,
    anyOf(
        equalTo(firstComment.title),
        equalTo(secondComment.title),
        equalTo(firstLink.title),
        equalTo(secondLink.title)
    )
)
assertThat(
    allNotes[1].title,
    anyOf(
        equalTo(firstComment.title),
        equalTo(secondComment.title),
        equalTo(firstLink.title),
        equalTo(secondLink.title)
    )
)
assertThat(
    allNotes[2].title,
    anyOf(
        equalTo(firstComment.title),
        equalTo(secondComment.title),
        equalTo(firstLink.title),
        equalTo(secondLink.title)
    )
)
assertThat(
    allNotes[3].title,
    anyOf(
        equalTo(firstComment.title),
        equalTo(secondComment.title),
        equalTo(firstLink.title),
        equalTo(secondLink.title)
    )
)
}
```

(from [MiscSamples/src/androidTest/java/com/commonsware/room/misc/polysingle/PolySingleStoreTest.kt](https://github.com/CommonWare/room-misc/blob/master/src/androidTest/java/com/commonsware/room/misc/polysingle/PolySingleStoreTest.kt))

Default Values and Partial Entities

We have already seen how Room supports Kotlin default values on properties:

```
class CustomColumnNameEntity(  
    @PrimaryKey  
    val id: String,  
    val title: String,  
    @ColumnInfo(name = "words") val text: String? = null,  
    val version: Int = 1  
) {
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/CustomColumnNameEntity.kt](#))

Here, text and version both have default values. Really, these have nothing to do with Room — they are a pure Kotlin concept. However, in the end, if we create a CustomColumnNameEntity instance without supplying text or version, we get those defaults.

But, in Room 2.2.0, we got another option for default values... one that on the surface seems pointless, as it does not work with @Insert operations. Principally, this appears to be tied to another feature added in that same release: partial entity support.

In this chapter, we will explore both of these features.

Default Values, and the Other Default Values

The new option for default values comes in the form of a defaultValue property on the @ColumnInfo annotation:

DEFAULT VALUES AND PARTIAL ENTITIES

```
@Entity(tableName = "defaultValue")
class DefaultValueEntity(
    @PrimaryKey
    val id: String,
    val title: String,
    @ColumnInfo(defaultValue = "something")
    val text: String? = null,
    @ColumnInfo(defaultValue = "123")
    val version: Int = 1
) {
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/DefaultValueEntity.kt](#))

Here, as in the example shown earlier, we have text and version properties on an entity (DefaultValueEntity), and in Kotlin, we have those properties default to null and 1, respectively. However, we also have @ColumnInfo annotations on those, where each has a defaultValue property. The one on text says the default value is something, while the one on version says that the default value is 123. So, for each property, we have two separate default values declared: one via Kotlin and one via the defaultValue annotation property... and both are different.



The Kotlin default values are for Kotlin's use. You can create instances of DefaultValueEntity without supplying values for text and/or version.

The @ColumnInfo defaultValue properties are for SQLite's use. If you attempt to INSERT a value into the table without providing values for the text and/or version columns, the defaultValue properties take effect. Basically, the defaultValue properties are embedded directly in the CREATE TABLE SQL statement generated from our @Entity declaration:

```
CREATE TABLE IF NOT EXISTS `defaultValue` (`id` TEXT NOT NULL, `title` TEXT NOT NULL,
`text` TEXT DEFAULT 'something', `version` INTEGER NOT NULL DEFAULT 123,
PRIMARY KEY(`id`))
```

Default Values and Inserts

If we @Insert our entity, though, the values for all columns get specified via properties on that entity. After all, regardless of whether or not we provide text or

DEFAULT VALUES AND PARTIAL ENTITIES

version when we call the `DefaultValueEntity` constructor, the resulting object will have values for those properties, just via the Kotlin defaults. So, our resulting `INSERT` SQL statement will provide values for all of those properties, meaning that the SQL default values will not be used.

So, once again:



However, `@Insert` is not the only option for executing `INSERT` statements in Room. We can also do that using `@Query`:

```
@Insert
fun insert(entity: DefaultValueEntity)

@Query("INSERT INTO defaultValue (id, title) VALUES (:id, :title)")
fun insertByQuery(id: String, title: String)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/DefaultValueEntity.kt](https://github.com/CommonWare/room/blob/master/misc/DefaultValueEntity.kt))

Here we have two DAO functions for inserting into our `defaultValue` table. The first uses `@Insert`. The second uses `@Query`, with a SQL `INSERT` statement that does not provide values for `text` or `version`. So, when we call `insertByQuery()`, our resulting database table row will contain the supplied `id` and `title` values, plus the *SQL default values* for `text` and `version`.

Partial Entities

Prior to the introduction of support for SQL default values, we always had to provide values for all columns in our `INSERT` statements. If you inserted solely via `@Insert`, you might not notice this limitation. The fact that we now have SQL default value support means that the “insert-by-`@Query`” option has greater flexibility: we do not have to provide all values for all columns.

Related to that is partial entity support, where we can have ordinary `@Insert` DAO functions also only supply a subset of properties to go into the to-be-inserted row.

We covered earlier in the book how we can use arbitrary POJOs or Kotlin data classes for the output of `@Query` functions:

DEFAULT VALUES AND PARTIAL ENTITIES

```
package com.commonware.room.misc

import androidx.room.*
import kotlin.random.Random

data class CountAndSumResult(val count: Int, val sum: Long)

@Entity(tableName = "aggregate")
class AggregateEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0,
    val value: Long = Random.nextLong(1000000)
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM aggregate")
        fun loadAll(): List<AggregateEntity>

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

        @Query("SELECT COUNT(*) as count, SUM(value) as sum FROM aggregate")
        fun countAndSum(): CountAndSumResult

        @Insert
        fun insert(entities: List<AggregateEntity>)
    }
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/AggregateEntity.kt](#))

Here, we use `CountAndSumResult` to model the response from:

```
SELECT COUNT(*) as count, SUM(value) as sum FROM aggregate
```

Similarly, we can use a POJO or Kotlin data class to model partial input to an `@Insert`, `@Update`, or `@Delete` function:

```
package com.commonware.room.misc

import androidx.room.*

data class PartialDefaultValue(
    val id: String,
    val title: String
)
```

DEFAULT VALUES AND PARTIAL ENTITIES

```
@Entity(tableName = "defaultValue")
class DefaultValueEntity(
    @PrimaryKey
    val id: String,
    val title: String,
    @ColumnInfo(defaultValue = "something")
    val text: String? = null,
    @ColumnInfo(defaultValue = "123")
    val version: Int = 1
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM defaultValue")
        fun loadAll(): List<DefaultValueEntity>

        @Query("SELECT * FROM defaultValue where id = :id")
        fun findByPrimaryKey(id: String): DefaultValueEntity

        @Insert
        fun insert(entity: DefaultValueEntity)

        @Query("INSERT INTO defaultValue (id, title) VALUES (:id, :title)")
        fun insertByQuery(id: String, title: String)

        @Insert(entity = DefaultValueEntity::class)
        fun insertPartial(partial: PartialDefaultValue)
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/DefaultValueEntity.kt](#))

In addition to the `insert()` and `insertByQuery()` DAO functions, we have `insertPartial()`. This accepts a `PartialDefaultValue` object and uses that for an INSERT into our table.

By default, this would fail at compile time, with the Room annotation processor complaining that `PartialDefaultValue` is not an entity. However, on the `@Insert` annotation for `insertPartial()`, we use an entity property to teach Room what `@Entity` is associated with this `@Insert` operation:

```
@Insert(entity = DefaultValueEntity::class)
fun insertPartial(partial: PartialDefaultValue)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/DefaultValueEntity.kt](#))

The POJO or Kotlin data class that you supply as input (`PartialDataEntity`) will be

DEFAULT VALUES AND PARTIAL ENTITIES

used like the entity itself, based on individual property names. Room will look in the supplied object for properties matching those of the entity class and use the ones that it finds. For everything else — such as text and version in this case — Room will rely on the underlying table either supporting NULL for the column... or having a `defaultValue` supplied.

Many developers will not need these features. Kotlin property default values and an ordinary `@Insert` will suffice. But, these features make it a bit easier to model classic SQL database structures and may help in some scenarios.

Room and Full-Text Search

SQLite supports [FTS virtual tables](#) for full-text searching of content. As the SQLite documentation puts it:

The most common (and effective) way to describe full-text searches is “what Google, Yahoo, and Bing do with documents placed on the World Wide Web”.

Originally, Room did not have any support for FTS, but in 2019 Google added FTS support to Room 2.2.0. So, in current versions of Room, you can use FTS in your Android app, for rich queries of long pieces of text.

In this chapter, we will explore more about this capability.

What Is FTS?

Standard SQL databases are great for ordinary queries. In particular, when it comes to text, SQL databases are great for finding rows where a certain column value matches a particular string. They are usually pretty good about finding when a column value matches a particular string prefix, if there is an index on that column. Things start to break down when you want to search for an occurrence of a string in a column — “find all rows where the column prose contains the word vague” — as this usually requires a “table scan” (i.e., iteratively examining each row to see if this matches). And getting more complex than that is often impossible, or at least rather difficult.

SQLite, in its stock form, inherits all those capabilities and limitations. However, SQLite also offers full-text indexing, where we can search our database much like how we use a search engine (e.g., “find all rows where this column has both foo and

bar in it somewhere”). While a full-text index takes up additional disk space, the speed of the full-text searching is quite impressive.

A Word About SQLite Versions

SQLite has evolved since Android’s initial production release in 2008.

In many cases, Android does not incorporate updates to third-party code, for backwards-compatibility reasons (e.g., Apache’s HttpClient). In the case of SQLite, newer Android versions do take on newer versions of SQLite... but the exact version of SQLite that a given version of Android uses is undocumented. Worse, some device manufacturers replace the stock SQLite for a version of Android with a different one.

[This Stack Overflow answer](#) contains a mapping of Android OS releases to SQLite versions, including various “anomalies” where manufacturers have elected to ship something else.

In many cases, the SQLite version does not matter. Core SQLite capabilities will have existed since the earliest days of Android. However, full-text indexing did not exist in the first SQLite used by Android, meaning that you will have to pay attention to your `minSdkVersion` and aim high enough that devices should support the full-text indexing option you choose.

Note that you could use an external SQLite implementation, one that gives you a newer SQLite engine than what might be on the device. For example, [SQLCipher for Android](#) ships its own copy of SQLite (with the SQLCipher extensions compiled in), one that is often newer than the one that is baked into the firmware of any given device.

FTS3, FTS4, and FTS5

There are three full-text indexing options available in SQLite: FTS3, FTS4, and [FTS5](#). Google, with Room, offers support for FTS3 and FTS4, not FTS5. Presumably, Google has determined that few device manufacturers enable FTS5. If you [package your own SQLite version](#), you could ensure that FTS5 is available to you. Perhaps in the future, [Google will add FTS5 support](#).

Comparing FTS3 and FTS4, FTS4 can be much faster on certain queries, though overall the speed of the two implementations should be similar. FTS4 may take a bit more disk space for its indexes, though.

Applying FTS to Room

The [FTS module](#) of [the book's primary sample project](#) demonstrates the use of FTS4 in Room.

The app is a trivial e-book reader. In `assets/`, it contains a copy of [the Project Gutenberg edition of H. G. Wells' "The Invisible Man"](#). This is saved in plain text files, with blank lines serving as paragraph separators. The reader itself simply displays the book in its entirety, one paragraph per row in a RecyclerView:

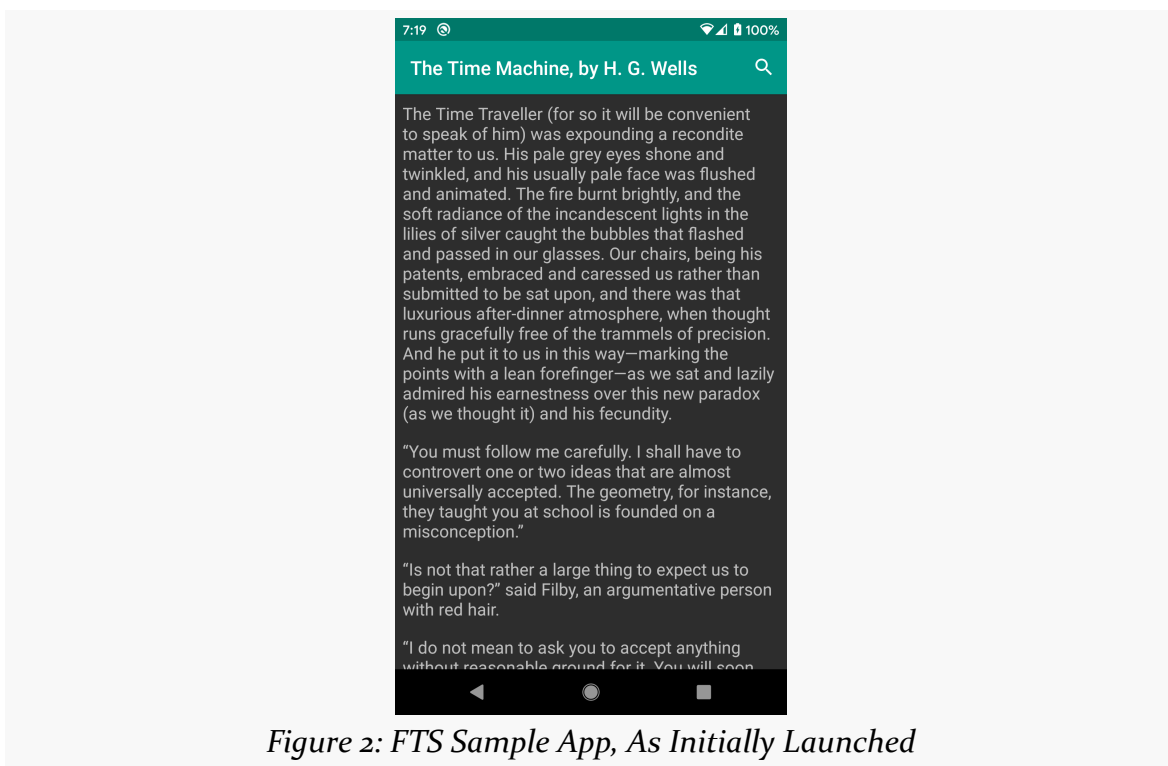


Figure 2: FTS Sample App, As Initially Launched

It also has a SearchView, and if you search on a term, you will be given snippets of the book showing where that word was used.

This chapter will focus on the BookRepository and its associated Room classes that handle storing the book content and conducting the searches.

Creating the FTS Table and Entity

While the book is packaged with the app in `assets/` as plain text files, SQLite and

ROOM AND FULL-TEXT SEARCH

FTS have no way of searching that content. So, the app has a SQLite table and associated Room entity for the text. Since the prose is divided into paragraphs, we have a ParagraphEntity that models those:

```
package com.commonware.room.fts

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "paragraphs")
data class ParagraphEntity(
    @PrimaryKey(autoGenerate = true) val id: Long,
    val sequence: Int,
    val prose: String
)
```

(from [FTS/src/main/java/com/commonware/room/fts/ParagraphEntity.kt](https://github.com/CommonWare/fts/blob/main/src/main/java/com/commonware/room/fts/ParagraphEntity.kt))

Here, sequence is an integer showing where the paragraph appears in the book, with lower numbers meaning earlier appearances. prose is the text of the paragraph itself.

There are a few SQLite strategies for creating and maintaining an FTS index. This book goes with a parallel-table approach, where we not only have an entity and table for the data, but a separate entity and table for the FTS index.

To that end, we not only have ParagraphEntity, but we have ParagraphFtsEntity:

```
package com.commonware.room.fts

import androidx.room.Entity
import androidx.room.Fts4

@Fts4(contentEntity = ParagraphEntity::class)
@Entity(tableName = "paragraphsFts")
data class ParagraphFtsEntity(val prose: String)
```

(from [FTS/src/main/java/com/commonware/room/fts/ParagraphFtsEntity.kt](https://github.com/CommonWare/fts/blob/main/src/main/java/com/commonware/room/fts/ParagraphFtsEntity.kt))

The FTS entity has the same properties as the main entity, for those columns that we want to be indexed by FTS. In this case, that is only the prose.

The @Fts4 annotation indicates that this is an entity representing an FTS4 “shadow table” for some other table. The contentEntity property of the annotation points Room to the main entity (ParagraphEntity in this case).

Under the covers, Room and SQLite's FTS4 will:

- Add a standard rowid column to the shadow table, matching the INTEGER ID of the main table
- Add triggers, such that whenever changes are made to the main table, Room can update the shadow table to match

As a result, while we will query using both tables, we will only insert content using ParagraphEntity — ParagraphFtsEntity will be handled for us.

Querying Using FTS

Our DAO is BookStore:

```
package com.commonware.room.fts

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query

@Dao
abstract class BookStore {
    @Insert
    abstract suspend fun insert(paragraphs: List<ParagraphEntity>)

    @Query("SELECT prose FROM paragraphs ORDER BY sequence")
    abstract suspend fun all(): List<String>

    @Query("SELECT snippet(paragraphsFts) FROM paragraphs JOIN paragraphsFts "+
        "ON paragraphs.id == paragraphsFts.rowid WHERE paragraphsFts.prose "+
        "MATCH :search ORDER BY sequence")
    abstract suspend fun filtered(search: String): List<String>
}
```

(from [FTS/src/main/java/com/commonware/room/fts/BookStore.kt](https://github.com/Commonware/room/blob/main/src/main/java/com/commonware/room/fts/BookStore.kt))

As noted above, our only data-manipulation function — insert() — works with ParagraphEntity rather than ParagraphFtsEntity. And, we are welcome to query ParagraphEntity as normal, without having to deal with FTS, as we do in the all() function to retrieve all the prose in sequential order.

However, we can also use the MATCH operator for queries against our FTS4 shadow table. MATCH takes [a search expression](#), which could be as simple as a word to find in the indexed content.

ROOM AND FULL-TEXT SEARCH

For the moment, let's pretend that the `filtered()` function on `BookStore` looked like this:

```
@Query("SELECT prose FROM paragraphs JOIN paragraphsFts "+
      "ON paragraphs.id == paragraphsFts.rowid WHERE paragraphsFts.prose "+
      "MATCH :search ORDER BY sequence")
abstract suspend fun filtered(search: String): List<String>
```

The `SELECT prose FROM paragraphs` and the `ORDER BY sequence` parts are simple SQL. Combined, they make up the query used in `all()`, to return all the prose, order by sequence.

However, `filtered()` also uses `JOIN` to connect the `paragraphsFts` table with the `paragraphs` table. The `id` of our `ParagraphEntity` will match the `rowid` of the corresponding `ParagraphsFtsEntity` in its shadow table.

Finally, we use `WHERE paragraphsFts.prose MATCH :search` to apply an FTS4 search against the FTS4-indexed prose in `paragraphsFts`. Through the `JOIN`, we can get columns back from `paragraphs` for whatever rows in `paragraphsFts` matched the FTS4 search expression.

As a result, calling this version of `filtered()` with an FTS4 search expression would give us all the complete text of all the paragraphs that matched that search expression.

Getting Snippets

However... the `filtered()` function shown above is not actually what `BookStore` has. Instead, it has:

```
@Query("SELECT snippet(paragraphsFts) FROM paragraphs JOIN paragraphsFts "+
      "ON paragraphs.id == paragraphsFts.rowid WHERE paragraphsFts.prose "+
      "MATCH :search ORDER BY sequence")
abstract suspend fun filtered(search: String): List<String>
```

(from [FTS/src/main/java/com/commonsware/room/fts/BookStore.kt](https://commonsware.com/licenses/room/fts/BookStore.kt))

Here, instead of retrieving prose from `paragraphs`, we have `snippet(paragraphFts)`.

ROOM AND FULL-TEXT SEARCH

When you use Internet search engines like Google, DuckDuckGo, etc., usually their search results are not just links, but snippets of text showing you the context around your search term, often with a search keyword highlighted:

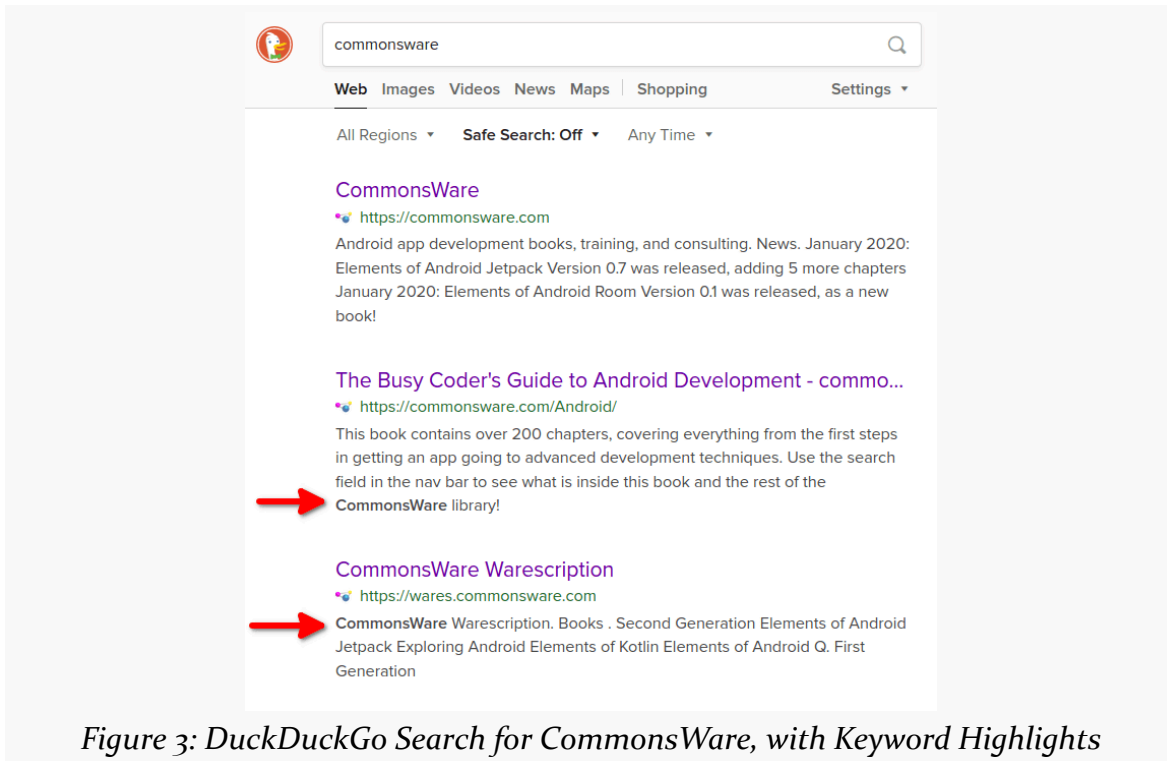


Figure 3: DuckDuckGo Search for CommonsWare, with Keyword Highlights

snippet() gives you the same effect. SQLite's FTS engine will return to you a portion of the text, with search keywords in boldface. The value passed to the snippet() function is the name of the FTS table (in this case, paragraphFts).

The RecyclerView.ViewHolder renders that HTML using HtmlCompat:

```
package com.commonsware.room.fts

import android.view.View
import android.widget.TextView
import androidx.core.text.HtmlCompat
import androidx.recyclerview.widget.RecyclerView

class RowHolder(root: View) : RecyclerView.ViewHolder(root) {
    private val prose = root.findViewById<TextView>(R.id.prose)

    fun bind(paragraph: String) {
        prose.text = HtmlCompat.fromHtml(paragraph, HtmlCompat.FROM_HTML_MODE_COMPACT)
    }
}
```

ROOM AND FULL-TEXT SEARCH

(from [FTS/src/main/java/com/commonsware/room/fts/RowHolder.kt](https://github.com/commonsware/room/fts/RowHolder.kt))

The effect is that the user's chosen search term shows up in bold in the search results:

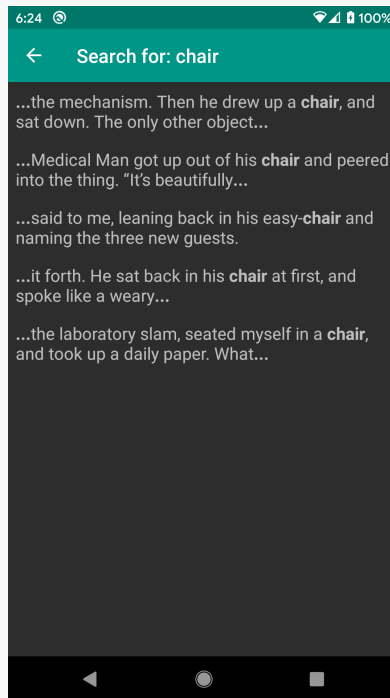


Figure 4: FTS Sample App, Showing Search Results for 'chair'

You do not have to use this feature, if you do not want, but it is available to you if it would be of use to your users.

Populating the Database

Our BookDatabase does not need anything special for having FTS be a part of its entity and DAO mix. However, ParagraphFtsEntity is an entity, so it needs to be listed in the @Database annotation:

```
package com.commonsware.room.fts

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
```

ROOM AND FULL-TEXT SEARCH

```
private const val DB_NAME = "book.db"

@Database(
    entities = [ParagraphEntity::class, ParagraphFtsEntity::class],
    version = 1
)
abstract class BookDatabase : RoomDatabase() {
    abstract fun bookStore(): BookStore

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, BookDatabase::class.java, DB_NAME).build()
    }
}
```

(from [FTS/src/main/java/com/commonsware/room/fts/BookDatabase.kt](#))

Technically, BookRepository needs to know even less about FTS. However, *somebody* has to have the job of adding the book to the database on first run, and in this case, that job falls on BookRepository. Its `all()` function will be called when the app launches, to populate the RecyclerView to show the book contents. So, we can see if `all()` on BookStore returns an empty list — if it does, we can surmise that the database is empty and we need to fill it:

```
suspend fun all(): List<String> {
    val result = db.bookStore().all()

    return if (result.isEmpty()) {
        import()
        db.bookStore().all()
    } else {
        result
    }
}
```

(from [FTS/src/main/java/com/commonsware/room/fts/BookRepository.kt](#))

The `import()` function iterates over all of the book chapters (which are numbered so their order can be maintained) and generates `ParagraphEntity` objects for each of their paragraphs. The resulting List of entities then gets inserted into the database via `insert()` on BookStore:

```
private suspend fun import() = withContext(Dispatchers.IO) {
    val assets = context.assets

    val entities = assets.list(BOOK_DIR).orEmpty()
        .sorted()
```

ROOM AND FULL-TEXT SEARCH

```
.map { paragraphs(assets.open("$BOOK_DIR/$it")) }
.flatten()
.mapIndexed { index, prose -> ParagraphEntity(0, index, prose) }

db.bookStore().insert(entities)
}
}

// inspired by https://stackoverflow.com/a/10065920/115145

private fun paragraphs(stream: InputStream) =
    paragraphs(stream.reader().readLines())

private fun paragraphs(lines: List<String>) =
    lines.fold(listOf<String>()) { roster, line ->
        when {
            line.isEmpty() -> roster + ""
            roster.isEmpty() -> listOf(line)
            else -> roster.take(roster.size - 1) + (roster.last() + " ${line.trim()}")
        }
    }
    .map { it.trim() }
    .filter { it.isNotEmpty() }
```

(from [FTS/src/main/java/com/commonsware/room/fts/BookRepository.kt](https://github.com/commonsware/room-fts/blob/master/BookRepository.kt))

Here, BOOK_DIR identifies the directory within assets/ where the chapters reside:

```
private const val BOOK_DIR = "TheTimeMachine"
```

(from [FTS/src/main/java/com/commonsware/room/fts/BookRepository.kt](https://github.com/commonsware/room-fts/blob/master/BookRepository.kt))

Dealing With Errors

It is possible that the user will type in an invalid search expression. SQLite has a specific search language, and the user's search expression might not adhere to that.

In this case, our `filtered()` DAO function, being a suspend function, will throw the exception. We can catch that and handle it how we see fit.

The app has a `SearchViewModel` that calls `filtered()` on `BookRepository` and surfaces the result via a `LiveData`:

```
package com.commonsware.room.fts

import android.content.Context
import android.util.Log
import androidx.lifecycle.LiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.liveData
```

```
class SearchViewModel(
    search: String,
    repo: BookRepository,
    private val context: Context
) : ViewModel() {
    val paragraphs: LiveData<List<String>> = liveData {
        try {
            emit(repo.filtered(search))
        } catch (t: Throwable) {
            Log.e("FTS", "Exception with search expression", t)
            emit(listOf(context.getString(R.string.search_error, search)))
        }
    }
}
```

(from [FTS/src/main/java/com/commonsware/room/fts/SearchViewModel.kt](#))

We wrap our `filtered()` call in a `try/catch`. If we get an exception, we log its details to Logcat, and we `emit()` a result with an error message for the user. In this case, we are taking advantage of the fact that our UI is already displaying text to the user — we just replace the normal search results with an error message in this case.

Alternatively, we could have `paragraphs` be a `LiveData` of some sealed class with `Content` and `Error` implementations. `Content` could hold the paragraphs, while `Error` could hold the error message. This would give the code that displays the results (`SearchFragment`) more flexibility in displaying the error, albeit with somewhat greater code complexity.

Why Are We Bothering?

Of course, you might wonder what all the fuss is about. In this sample, we are loading *the entire book* into memory, via the `all()` functions on our `BookStore` and `BookRepository`. If we want to search, we could do so purely in memory, without any FTS stuff.

There are three reasons why we are implementing FTS here:

1. While we could do a simple keyword search easily enough ourselves, FTS supports [richer search expressions](#) than that, and writing a whole search expression parser and executor is going to be a lot of work
2. We would not have to load the entire book into memory, if we did not want to — we will examine that more [in a later chapter](#)
3. It would be silly to have a chapter on FTS and then not show it

Supported MATCH Syntax

The MATCH operator supports a range of query structures, including:

- Keyword matches (e.g., Android)
- Prefix matches (e.g., SQL*)
- Phrase matches (e.g., "open source")
- NEAR, AND, OR, and NOT operators (e.g., sqlite AND database)

[The SQLite FTS3/FTS4 documentation](#) has more on its supported query syntax.

Migrating to FTS

If you have already shipped your app, and you want to add an FTS index to an existing table or add a new table with FTS, you will need to implement a migration, as we discussed in [a previous chapter](#).

In this case, your migration will be a bit more complicated. Not only will you need to create your FTS table, but you will also need to define some triggers to keep your main table and the FTS shadow table in sync.

If you set up Gradle to [export your schemas](#), you will have access to the required SQL. Partly, that will be in the createSql JSON property for your entity:

```
"createSql": "CREATE VIRTUAL TABLE IF NOT EXISTS `${TABLE_NAME}` USING FTS4(`prose`  
TEXT NOT NULL, content=`paragraphs`)"
```

And, partly, that will be in the contentSyncTriggers JSON property, which holds a list of SQL statements to be executed:

```
"contentSyncTriggers": [  
  "CREATE TRIGGER IF NOT EXISTS room_fts_content_sync_paragraphsFts_BEFORE_UPDATE  
  BEFORE UPDATE ON `paragraphs` BEGIN DELETE FROM `paragraphsFts` WHERE  
  `docid`=OLD.`rowid`; END",  
  "CREATE TRIGGER IF NOT EXISTS room_fts_content_sync_paragraphsFts_BEFORE_DELETE  
  BEFORE DELETE ON `paragraphs` BEGIN DELETE FROM `paragraphsFts` WHERE  
  `docid`=OLD.`rowid`; END",  
  "CREATE TRIGGER IF NOT EXISTS room_fts_content_sync_paragraphsFts_AFTER_UPDATE  
  AFTER UPDATE ON `paragraphs` BEGIN INSERT INTO `paragraphsFts`(`docid`, `prose`)  
  VALUES (NEW.`rowid`, NEW.`prose`); END",  
  "CREATE TRIGGER IF NOT EXISTS room_fts_content_sync_paragraphsFts_AFTER_INSERT
```

ROOM AND FULL-TEXT SEARCH

```
AFTER INSERT ON `paragraphs` BEGIN INSERT INTO `paragraphsFts`(`docid`, `prose`)
VALUES (NEW.`rowid`, NEW.`prose`); END"
]
```

Your migration will need to include all of these, so your migrated database schema matches the database schema that new users will use.

Room and Conflict Resolution

For `@Insert` and `@Update` methods in your `@Dao`, you can have `onConflict` properties in the annotations that stipulate what should happen if the insert or update results in a violation of a few types of constraints:

- A unique index, including a duplicate primary key
- A `NULL` value being put into a `NOT NULL` column
- A `CHECK` constraint (which Room does not support presently)

Room gives you five `OnConflictStrategy` enum values to choose from for your `onConflict` property. Each of those `OnConflictStrategy` values maps to an equivalent SQLite keyword, and each of those strategies results in different behavior in SQLite.

ROOM AND CONFLICT RESOLUTION

Value	Meaning
<code>OnConflictStrategy.ABORT</code>	Cancels this statement but preserves prior results in the transaction and keeps the transaction alive
<code>OnConflictStrategy.FAIL</code>	Like ABORT, but accepts prior changes by this specific statement (e.g., if we fail on the 50th row to be updated, keep the changes to the preceding 49)
<code>OnConflictStrategy.IGNORE</code>	Like FAIL, but continues processing this statement (e.g., if we fail on the 50th row out of 100, keep the changes to the other 99)
<code>OnConflictStrategy.REPLACE</code>	For uniqueness violations, deletes other rows that would cause the violation before executing this statement
<code>OnConflictStrategy.ROLLBACK</code>	Rolls back the current transaction

However, they may *not* wind up with different behavior in Room, due to the way that Room works with SQLite. As a result, two of these five (FAIL and ROLLBACK) have been deprecated.

In this chapter, we will examine those five options and see what SQLite does and what the resulting effects are in a Room-based app. As you will see, while there are five official options, fewer are practical.

Abort

```
@Insert(onConflict = OnConflictStrategy.ABORT)
```

```
@Update(onConflict = OnConflictStrategy.ABORT)
```

What SQLite Does

This strategy maps to INSERT OR ABORT or UPDATE OR ABORT statements. If a constraint violation would occur from this statement, the statement is skipped. SQLiteDatabase throws a SQLiteConstraintException. However, if you have started

a transaction, that transaction remains open, so further statements in the transaction can be executed.

Effects in Room

An individual `@Insert` or `@Update` method that uses `OnConflictStrategy.ABORT` will throw a `SQLiteConstraintException` if there is a constraint violation. In isolation, this fits with what you might expect.

The problem comes with `@Transaction`.

Every method that Room generates in response to your `@Dao`-annotated methods has the same basic structure:

```
@Override
public void whatever(SomeEntity... entities) {
    __db.beginTransaction();
    try {
        // the real work for whatever whatever() does
        __db.setTransactionSuccessful();
    } finally {
        __db.endTransaction();
    }
}
```

This includes `@Transaction`-annotated methods, which just wrap that template around a call to your real method:

```
@Override
public void whatever(SomeEntity... entities) {
    __db.beginTransaction();
    try {
        super.whatever(entities);
        __db.setTransactionSuccessful();
    } finally {
        __db.endTransaction();
    }
}
```

If anything in your `@Transaction` method throws an exception, of any kind, the entire transaction gets rolled back, courtesy of the `try/finally` structure.

So, even though `ABORT` is supposed to keep the transaction open, Room rolls back the transaction, so that your `@Transaction` is atomic.

Fail

```
@Insert(onConflict = OnConflictStrategy.FAIL)
```

```
@Update(onConflict = OnConflictStrategy.FAIL)
```

What SQLite Does

This strategy maps to INSERT OR FAIL or UPDATE OR FAIL statements. These work much like their ABORT counterparts, in that a SQLiteConstraintException is thrown, but the transaction remains open.

The difference is in what happens if your UPDATE statement affects several rows. In that case, rows that were changed *prior* to the constraint violation remain changed. The row with the constraint violation, and any others after it, are unchanged.

Frankly, this does not seem like a particularly good idea. At least with ABORT, you have consistent behavior. With FAIL, some arbitrary amount of data gets changed, and the rest is not, and without doing your own post-FAIL analysis, you have no idea what to expect.

Effects in Room

While Room used to support OnConflictStrategy.FAIL, it is now deprecated. Google recommends that you use [ABORT](#) instead.

Ignore

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
```

```
@Update(onConflict = OnConflictStrategy.IGNORE)
```

What SQLite Does

This strategy maps to INSERT OR IGNORE or UPDATE OR IGNORE statements. This has two key differences to how FAIL works:

- It does not throw any sort of exception.
- It tries to process everything that the statement should affect. So, if there are 100 rows to be updated, and the 50th row winds up with a constraint

violation, that row is skipped, but SQLite continues to try to process any not-yet-updated rows.

The result is that everything that can be inserted or updated is inserted or updated, with individual rows being skipped where they fail on constraint violations.

This is risky, in that you may not necessarily have a good way of knowing that some of your requested data manipulations did not take effect.

Effects in Room

Since IGNORE does not trigger an exception, Room will commit the transaction that contains your @Insert or @Update work. Hence, this works, insofar as Room does not reject changes that SQLite would otherwise accept because Room rolled back the transaction. You still suffer from not knowing what exactly was changed by your @Insert or @Update method, though.

However, for SQL statements that you know will only affect one row, IGNORE is a perfectly reasonable solution.

Replace

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
```

```
@Update(onConflict = OnConflictStrategy.REPLACE)
```

What SQLite Does

This strategy maps to INSERT OR REPLACE or UPDATE OR REPLACE statements.

If SQLite encounters a row where a UNIQUE or PRIMARY KEY constraint conflicts with the change requested via the INSERT OR REPLACE or UPDATE OR REPLACE statement, SQLite *deletes the existing data* and proceeds with the statement. The net effect is that you replace the old data with the new data.

If SQLite encounters a row where a NOT NULL constraint is violated, it will attempt to replace the null value with the default value for that column, if there is one defined in the table schema. If not, this strategy behaves like ABORT.

And for any other constraint violation, this strategy behaves like ABORT.

As a result, this is useful, but only in fairly controlled circumstances, and then mostly for `INSERT OR REPLACE`. This guarantees that your desired row will wind up in the table, either because it is new or because SQLite gets rid of the previous edition of that row.

Effects in Room

This strategy, like `IGNORE`, works pretty much as SQLite intends, for the most common use case: a duplicate on a `UNIQUE` or `PRIMARY KEY` constraint, resulting in data replacement.

This strategy will have the same problems as `ABORT` in the cases where it behaves like `ABORT` — Room will roll back the transaction once it sees the `SQLiteConstraintException`.

Rollback

```
@Insert(onConflict = OnConflictStrategy.ROLLBACK)
```

```
@Update(onConflict = OnConflictStrategy.ROLLBACK)
```

What SQLite Does

This strategy maps to `INSERT OR ROLLBACK` or `UPDATE OR ROLLBACK` statements.

It rolls back the transaction, as you might expect given the name.

Effects in Room

As with `FAIL`, while Room used to support `OnConflictStrategy.ROLLBACK`, it is now deprecated. Google recommends that you use [ABORT](#) instead.

What Should You Use with Room?

Given the deprecations, your options with Room now are very straightforward:

- If you want to replace the database contents with new data when there is a conflict, use `REPLACE`
- If you want to keep the existing database contents when there is a conflict, but not treat it as an error, use `IGNORE`

ROOM AND CONFLICT RESOLUTION

- If you want to keep the existing database contents when there is a conflict, and you want an exception to be raised so you know about the problem, use ABORT (or, do nothing, as ABORT is the default strategy)

A Room With a View

SQLite, like many SQL databases, not only supports CREATE TABLE for creating tables, but CREATE VIEW for creating views. In this case, a database view has nothing much to do with an Android View. Rather, a database view is a “view” onto other data in the database. It amounts to a SQL query that you can in turn query against as if it were a table.

For example, [earlier in the book](#), we saw AppEntity:

```
package com.commonware.room.misc

import androidx.room.*

@Entity(tableName = "apps")
data class AppEntity(
    @PrimaryKey
    val applicationId: String,
    val displayName: String,
    val shortDescription: String,
    val fullDescription: String,
    val latestVersionName: String,
    val lastUpdated: Long,
    val iconUrl: String,
    val packageUrl: String,
    val donationUrl: String
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM apps")
        fun loadAll(): List<AppEntity>

        @Query("SELECT applicationId, displayName, shortDescription, iconUrl FROM apps")
        fun loadListModel(): List<AppListModel>

        @Insert
        fun insert(entity: AppEntity)
    }
}
```

A ROOM WITH A VIEW

```
data class AppListModel(  
    val applicationId: String,  
    val displayName: String,  
    val shortDescription: String,  
    val iconUrl: String  
)
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/AppEntity.kt](#))

We used that to demonstrate having a DAO function return a type other than the entity itself, where `loadListModels()` returns a subset of the columns of the table, and those get mapped to an `AppListModel` instead of an `AppEntity`.

Another way to have implemented that would be to define a database view on the `apps` table that happens to return those four columns. We can then query that view as if it were an actual table, with separate `@Query` functions.

Defining a View

Just as an entity is declared via an `@Entity` annotation, a database view is declared via a `@DatabaseView` annotation. It takes two parameters:

- `value`, which is a SQL query like you might have in a `@Query` annotation, though no parameters are allowed; and
- `viewName`, which is the equivalent of `tableName` on `@Entity`, supplying the name to use for this view

`viewName` is optional; if you skip it, the view is named the same as the class on which you are applying the `@DatabaseView` annotation.

`AppView` is a view on the `AppEntity` table (`apps`) that returns the four columns that we were using with `loadListModels()` and `AppListModel` before:

```
@DatabaseView(  
    viewName = "appListView",  
    value = "SELECT applicationId, displayName, shortDescription, iconUrl FROM apps"  
)  
data class AppView(  
    val applicationId: String,  
    val displayName: String,  
    val shortDescription: String,  
    val iconUrl: String  
) {
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/AppView.kt](#))

Registering the View

Just as we have to teach our RoomDatabase about entities, we have to teach it about database views. For entities, we have an entities array that we supply to the @Database annotation. For database views, we have a view array that fills the same role:

```
@Database(  
    entities = [  
        AutoGenerateEntity::class,  
        CompositeKeyEntity::class,  
        UniqueIndexEntity::class,  
        IgnoredPropertyEntity::class,  
        NullablePropertyEntity::class,  
        CustomColumnNameEntity::class,  
        IndexedEntity::class,  
        AppEntity::class,  
        AggregateEntity::class,  
        TransmogrifyingEntity::class,  
        EmbeddedLocationEntity::class,  
        DefaultValueEntity::class,  
        com.commonware.room.misc.onetomany.Book::class,  
        com.commonware.room.misc.onetomany.Category::class,  
        com.commonware.room.misc.manytomany.Book::class,  
        com.commonware.room.misc.manytomany.Category::class,  
        com.commonware.room.misc.manytomany.BookCategoryJoin::class,  
        LinkEntity::class,  
        CommentEntity::class,  
        NoteEntity::class,  
        AutoEnumEntity::class  
    ],  
    views = [  
        AppView::class  
    ],  
    version = 1  
)
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/MiscDatabase.kt](https://github.com/CommonWare/misc-samples/blob/master/src/main/java/com/commonware/room/misc/MiscDatabase.kt))

So, here, we tell MiscDatabase that we have a single @DatabaseView, named AppView.

Querying a View

And, just as we can have a @Dao class that works with AppEntity and its table, we can

A ROOM WITH A VIEW

have a @Dao class that works with AppView and its database view:

```
package com.commonware.room.misc

import androidx.room.Dao
import androidx.room.DatabaseView
import androidx.room.Query

@DatabaseView(
    viewName = "appListView",
    value = "SELECT applicationId, displayName, shortDescription, iconUrl FROM apps"
)
data class AppView(
    val applicationId: String,
    val displayName: String,
    val shortDescription: String,
    val iconUrl: String
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM appListView")
        fun loadListView(): List<AppView>

        @Query("SELECT * FROM appListView WHERE applicationId = :applicationId")
        fun findById(applicationId: String): AppView?
    }
}
```

(from [MiscSamples/src/main/java/com/commonware/room/misc/AppView.kt](https://github.com/CommonWare/misc-samples/blob/main/src/main/java/com/commonware/room/misc/AppView.kt))

loadListView() on AppView.Store will return the exact same list as does listListModels() on AppEntity.Store, as they both refer to the same query. We reference appListView in @Query annotations just like we do tables like apps. The available columns are determined by our value query in the @DatabaseView and our properties on the class that we apply @DatabaseView to. Under the covers, when we query appListView, SQLite will really query apps based on the value query and then sub-select off of that based on the @Query against appListView.

We can also have queries against the view that apply constraints, as seen in findById(). In the end, we treat the database view much like a table... for read operations. We cannot @Insert, @Update, or @Delete data in a database view — for those operations, we have to manipulate the table that is used by the view.

OK, Why Bother?

This seems a bit silly, as we did not gain much by having the database view instead of just having the simple loadListModels() function on AppEntity.Store.

In a database server, database view implementations usually have two features:

A ROOM WITH A VIEW

- They are read-only, and
- They can be secured separately from the underlying tables

As a result, you might expose database views as being something like a read-only API to certain types of systems (e.g., report generators) that only need access to some of the data.

SQLite's views are read-only, but SQLite has no per-user security model the way that a database server might. This limits the usefulness of database views.

However, it may be that there are certain subsets of columns, or certain WHERE constraints, that you need to use a lot. Rather than duplicate them across many separate @Query functions, you could centralize them in a database view. That way, if the list of columns changes or the WHERE constraints need to be revised, you can do so in one spot (the view definition) instead of having to ensure that you update lots of individual queries.

Room and PRAGMAs

Room covers a lot of what you will need when interacting with SQLite from your app. Room might not cover *everything* of what you would like to use with SQLite, though.

Some things — particularly anything involving table definitions — pretty much requires Room itself to be upgraded in order to work. Anything that lies outside of Room, though, is fair game, though you have to resort to classic SQLite approaches to make it work.

One of those approaches is to execute a PRAGMA, as we will explore briefly in this chapter.

When To Make Changes

You have two main events for when to make changes outside of Room to the database: when it is created and when it is opened. Which you use depends on the nature of your changes.

Changes that are persistent would be applied when the database is created, or (eventually) via a Migration when the database schema is modified. For example, using `CREATE TRIGGER` to create a trigger results in a persistent change to the database, so you only need to do this when the database schema is created or modified.

However, some PRAGMA statements are transient, living for the life of our connection to the database. Once the connection is closed, the effects of those PRAGMA statements go away. As a result, we have to apply these every time that the database is opened.

Example: Turbo Boost Mode

Some developers are desperate to wring every last bit of performance out of their database, even to the point of risking data loss or corruption. Some PRAGMA statements tie into performance this way.

For example, normally, many times when SQLite writes data to disk, it will use `fsync()` or the equivalent to block until all of the bytes are confirmed to be written. This is important in operating systems with write-caching filesystems, as otherwise the data that you think that you wrote might actually just be in a buffer waiting to be written in the future. Android, when using the ext4 filesystem, is one such OS. However, `PRAGMA synchronous = OFF` tells SQLite to skip those `fsync()` calls. This speeds up I/O, with increased risk of the database becoming corrupted if there is a major system problem while that I/O is going on. This is a transient PRAGMA, only affecting the current connection.

Even riskier is `PRAGMA journal_mode = MEMORY`. In effect, this says to keep the transaction log of the database in memory, rather than writing it to disk. As [the documentation states](#), “if the application using SQLite crashes in the middle of a transaction when the MEMORY journaling mode is set, then the database file will very likely go corrupt”. But, some people would consider performance gains as being a valid trade-off here. This is a persistent setting, and so it only needs to be applied once.

The approach for both of these cases is to use a `RoomDatabase.Callback`, as seen in the [PragmaTest](#) sample module from [the book’s primary sample project](#). `RoomDatabase.Callback` was introduced in [the chapter on the support database API](#).

This is a tests-only module, one that tests importing a bunch of city population data from a JSON file into a Room-powered database called `CityDatabase`.

The `newInstance()` method that we use to create an instance of our `CityDatabase` uses a `RoomDatabase.Builder` as normal. However, based on a boolean parameter, it may also use `addCallback()` to add a `RoomDatabase.Callback` to the builder:

```
package com.commonware.room.pragmatetest

import android.content.Context
import androidx.annotation.VisibleForTesting
import androidx.room.Database
import androidx.room.Room
```

ROOM AND PRAGMAS

```
import androidx.room.RoomDatabase
import androidx.sqlite.db.SupportSQLiteDatabase

@Database(entities = [City::class], version = 1)
abstract class CityDatabase : RoomDatabase() {
    abstract fun cityStore(): City.Store

    companion object {
        @VisibleForTesting
        const val DB_NAME = "un.db"

        fun newInstance(ctxt: Context, applyPragmas: Boolean): CityDatabase {
            val builder = Room.databaseBuilder(
                ctxt,
                CityDatabase::class.java,
                DB_NAME
            )

            if (applyPragmas) {
                builder.addCallback(object : Callback() {
                    override fun onCreate(db: SupportSQLiteDatabase) {
                        super.onCreate(db)
                        db.query("PRAGMA journal_mode = MEMORY")
                    }

                    override fun onOpen(db: SupportSQLiteDatabase) {
                        super.onOpen(db)
                        db.query("PRAGMA synchronous = OFF")
                    }
                })
            }

            return builder.build()
        }
    }
}
```

(from [Pragmatest/src/main/java/com/commonsware/room/pragmatest/CityDatabase.kt](https://commonsware.com/room/pragmatest/CityDatabase.kt))

There are two methods that you can supply on a Callback implementation: `onCreate()` and `onOpen()`. As the names suggest, they are called when the database is created and opened, respectively. In each, you are handed a `SupportSQLiteDatabase` instance, which has an API reminiscent of the framework's `SQLiteDatabase`. It has a `query()` method that works like `rawQuery()`, taking a simple SQL statement (that might return a result set) and executing it. Since PRAGMA might return a result set, we have to use `query()` instead of `execSQL()`. Here, we invoke our PRAGMA statements at the appropriate times.

ROOM AND PRAGMAS

And, in truth, there does seem to be a performance gain:

Scenario	Use the PRAGMAS?	Time (milliseconds)
Inserting 1,063 cities via individual insert() calls	No	1,135
Inserting 1,063 cities via individual insert() calls	Yes	960
Inserting 1,063 cities in a single insert() call	No	393
Inserting 1,063 cities in a single insert() call	Yes	252

(tests conducted on a Google Pixel 4)

Proper use of transactions — such as doing all of the inserts at once rather than one at a time — has a much bigger impact, though. Using these two PRAGMA statements is a bit like [using a holodeck with the safeties off](#): you may have some casualties.

Packaged Databases

We often tend to think of databases being populated at runtime:

- From user input
- From data retrieved from a server
- From device sensors or other local dynamic data sources

However, another scenario is to have the database be populated already when the app is first run, with some initial data supplied by the app developer. We will explore this sort of “packaged” database in this chapter.

Going Back In Time

Back in the chapter on [full-text searching](#), we needed a database with something to search. There, we used the text of H. G. Wells’ “The Time Machine”. The way that we got the data into our database was:

- Read in text files that were packaged as assets
- Subdivide those files into paragraphs, based on blank lines as paragraph separators
- Insert those paragraphs into our FTS-enhanced table

We did this manually on the first run of the app. Our `BookRepository` would detect that the table was empty and arrange to import the paragraphs.

It works. It is clunky, but it works.

Room offers an alternative: packaging an entire SQLite database as an asset. We can teach Room to use that database as a starting point. Then, when we first run the

app, Room will copy the database from assets/ instead of creating an empty database with our schema. So, we could prepare a database that has our paragraphs already in it and use that, rather than do all the text-file import work at runtime.

This approach eliminates the time and code necessary to do the data import. In this particular case, the app actually gets *bigger*, because the SQLite database (with its FTS index) is larger than the text files.

The Room Mechanics

[The PackagedFTS module](#) of [the book's primary sample project](#) is a clone of the FTS module, except that we now switch to using a packaged database.

In the original FTS project, we had assets/TheTimeMachine/ as a directory, containing a series of text files representing chapters in the book. Now, we have assets/TheTimeMachine.db, a SQLite database containing our paragraphs. We will discuss [later in the chapter](#) where this database came from. For the moment, assume that it was created using [a magnetized needle and a steady hand](#).

When our BookDatabase newInstance() function uses Room.databaseBuilder() to set up the database, we have an extra configuration call: createFromAsset(). This provides a relative path within assets/ to the database that we want to use at the outset:

```
package com.commonware.room.fts

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

private const val DB_NAME = "book.db"

@Database(
    entities = [ParagraphEntity::class, ParagraphFtsEntity::class],
    version = 1
)
abstract class BookDatabase : RoomDatabase() {
    abstract fun bookStore(): BookStore

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, BookDatabase::class.java, DB_NAME)
```

PACKAGED DATABASES

```
        .createFromAsset("TheTimeMachine.db")
        .build()
    }
}
```

(from [PackagedFTS/src/main/java/com/commonsware/room/fts/BookDatabase.kt](#))

For simple cases, that is all that we need.

In particular, `BookRepository` no longer needs all that code that we had to import the text files. We can just have pass-through calls to the database:

```
package com.commonsware.room.fts

class BookRepository(private val db: BookDatabase) {
    suspend fun all() = db.bookStore().all()

    suspend fun filtered(search: String): List<String> =
        db.bookStore().filtered(search)
}
```

(from [PackagedFTS/src/main/java/com/commonsware/room/fts/BookRepository.kt](#))

Everything else works as it did before, and if you run the app, it should behave as does the original.

(Why did the FTS module bother with the whole import-the-text-files approach? That chapter, and its sample app, were written before `createFromAsset()` was added to Room.)

Creating the Database Asset

If you are going to use this approach in your app, you will need to get that packaged database from somewhere. And, perhaps you do not own a magnetized needle or a suitably-trained butterfly. Instead, you are going to need to use other approaches to create the database and fill in your starter data.

Build In Android

The original approach used for `PackagedFTS` was simple: the author ran the FTS app and used Device File Explorer to copy the populated database from that app over to `assets/` of the `PackagedFTS` project.

PACKAGED DATABASES

In other words, you can create your database using one app that you then package into another app.

For example, you could:

- Have your Room entity classes in a library module
- Have your main app reference that library module
- Have a utility app also reference that library module
- Have the utility app contain the code to populate the database from some data sources

You would then run the utility app and copy the database (using Device File Explorer, `adb pull`, etc.) to the main app's `assets/` directory.

Build By Hand

There are plenty of SQLite client programs available, such as [DB Browser for SQLite](#). You could use one to hand-populate your database.

In this case, you would also need to consider your database schema. Room defines what the tables are and what SQL statements are used to create them and related structures (e.g., indices). If you have enabled [automatic exporting of database schemas](#), then you will have the SQL statements to use.

Build By Script

Another possibility is to write some software that creates and populates your database, but have that software run on your development machine (or perhaps some server), rather than on an Android device. SQLite client APIs are available for many languages. You could:

- Create a standalone command-line program that you run manually
- Create a standalone command-line program that you run via a custom Gradle task
- Create a Gradle plugin that creates the database
- Create the database directly in a custom Gradle task
- Etc.

Dealing With Metadata and Upgrades

Eventually, though, you may need a different database, either for schema changes or content changes.

You can handle this via [migrations](#) as normal. However, not only are you responsible for changing the schema but also for adjusting the content to take that new schema into account. And, if you want new or modified content, you will need to handle that as well.

An alternative is to add `fallbackToDestructiveMigration()` to the `Room.databaseBuilder()` configuration. This says “if we cannot find migrations to handle version changes, start over from scratch”. And, if you are using `createFromAsset()`, this means that Room will delete the existing database and make a new copy from the now-current asset. This works, though it has problems if your database is not a simple copy of the asset, as we will see in the next section.

Room uses the `room_master_table` to track Room-specific metadata, along with `android_metadata` that is set up and managed by the framework copy of `SQLiteDatabase`. For an initial deployment, you do not need these tables — the copy in `PackagedFTS` does not have them, for example. Room will create them automatically.

Hybrid Data

Life is simple if all your data comes from packaged databases, or if all your data comes from dynamic data sources (users, servers, etc.). Things get messy when your data is a hybrid of both: partially packaged and partially dynamic.

One Database

You could put everything into one database. That database would start out being copied from an asset, with empty tables for the dynamic data. Your app would then add data to those tables using ordinary Room operations.

This works, but probably it eliminates the `fallbackToDestructiveMigration()` option for dealing with database schema and content upgrades. `fallbackToDestructiveMigration()` would start everything over from the now-current asset, which will once again have empty tables for the dynamic data. You would lose whatever is in the current database’s edition of those tables, and that

might make the user unhappy.

Instead, you would need to use migrations to carefully update your schema and formerly-packaged content while leaving the dynamic data alone.

Separate Databases

One workaround for this is to use separate databases: one for the packaged data and one for the dynamic data. You would have two `RoomDatabase` classes with entities, DAOs, and so on. This means that changes that you make to one database, such as replacing it with the current asset via `fallbackToDestructiveMigration()`, would not affect the other database.

The downside is that these databases are completely independent. You cannot have entities in one database have foreign keys to the other database, for example. Instead, anything like that would need to be implemented in application code. Depending on your database structure, this may be practical or insane.

Attached Databases

SQLite itself has a solution for this: attached databases. The `ATTACH DATABASE` command tells SQLite to work with multiple databases at once, and you can have SQL statements that pull from both of them. Used carefully, this could allow the seamless integration seen in the single-database solution while still making it easy to bulk-replace the packaged data using `fallbackToDestructiveMigration()`.

Room, however, [has no support for ATTACH DATABASE](#) at the present time.

Backing Up Your Room

Users do not like it when their data vanishes, gets damaged, or otherwise becomes unusable.

Fortunately, modern smartphone hardware is fairly reliable. Still, problems can happen. Sometimes those problems come from the user, such as accidentally deleting something that they would rather not have deleted.

While the OS offers “backups”, in reality Android’s “backup” mechanism is embarrassing:

- Neither users nor developers strictly control when backups are taking place
- Neither users nor developers control where backups are stored
- Neither users nor developers control when backups get restored

In short, Android’s “backup” mechanism is for disaster recovery (e.g., user dropped their phone in a toilet and replaced the phone). You may want to offer something else that is more user-controlled. This chapter will explore how to do that.

Backup and Restore. Or, Import and Export.

“Backup” and “restore” imply making a copy of the data, such that in case of a problem with the original data, we can replace the damaged original with the copy.

“Import” and “export” imply making a copy of the data, such that the user can perhaps manipulate that data in some third-party tool. Later, the “import” implies that we can take external data and add it to the app’s own data, or perhaps replace the app’s own data outright.

The line dividing “backup/restore” and “import/export” is rather blurry. If you back up data to a user-accessible spot, and the data is in an open format, “backup” and “export” become identical, in effect. You may not *intend* for the user to use the data in another piece of software, but you cannot stop it either.

Choosing a Storage Target

You could back up or export the data to some file on the filesystem. On newer versions of Android, though, your ability to work with external storage starts to become more restricted.

You could back up or export the data to some location that the user chooses through the Storage Access Framework, such as via `ACTION_CREATE_DOCUMENT`.

You could back up the data to some server. In this case, the safest thing to do is to back up the data to some file on the filesystem first, then upload the file. As we will see later in the chapter, it is important for the database to be closed at the time you are making the backup copy, so we want the backup process to be as fast as possible. Directly streaming the data to some server will be a lot slower than making a local copy, due to network speeds.

It may be that your repository can be oblivious (somewhat) to these decisions. If you tell the repository to back up to a `Uri`, then whether that `Uri` points to a local file or to a location from `ACTION_CREATE_DOCUMENT` is immaterial. Your UI and viewmodel can make the decision of exactly where the copy is made.

Thinking About Journal Modes

If you examine your Room-enabled app’s portion of internal storage in Device File Explorer, in the `databases/` subdirectory you will find your actual Room database. Frequently, though, it consists of three files:

- The actual database (whatever.db)
- A file with `-shm` appended (whatever.db-shm)
- A file with `-wal` appended (whatever.db-wal)

This indicates that the database was opened in a particular “journal mode” called [write-ahead logging \(WAL\)](#) and has not been closed. Roughly speaking, in WAL mode, database transactions primarily affect these secondary files and only get merged into the main database at particular “checkpoints”. Such checkpoints occur

automatically, and for most circumstances we do not really worry about them. However, when it comes to making a backup or exporting a database, it is best if everything gets merged into the main database file.

Primarily, the way we can do that is to `close()` our `RoomDatabase`. Once all connections to our SQLite database are closed, SQLite will checkpoint the database, merging everything into the main database file and removing the `-shm` and `-wal` files.

You can disable WAL outright by calling `setJournalMode()` on your `RoomDatabase.Builder` and opting into `TRUNCATE` as a journal mode.

Keeping It Closed

However, even if you were to opt out of WAL, it is important that your database be closed when we are making a backup or exporting the data. SQLite will know nothing about this backup/export operation. If part of your code is performing database operations while another part of your code is making a copy of the database file, you will wind up with a corrupted database copy. This is not unique to SQLite: making a copy of a file while that file is being written to is asking for trouble.

In principle, this is not too hard to manage:

- Before making the copy of the database, call `close()` on the `RoomDatabase`
- Make the copy
- Before trying to work with the database again, re-open it using a `RoomDatabase.Builder`

This makes a repository a bit more complicated, as now it will need to be able to close and open databases as needed. Plus, your repository will be your primary “line of defense” against bugs. For example, suppose that you have a `WorkManager` scheduled task to synchronize your app with a Web service. If that task gets executed while your backup is being made, you do not want to re-open the database at that time. Somehow, the worker will need to find out that the database is unavailable, so it can reschedule that work.

The sample app, being a simple book sample, does not get into how to manage this, as the details will vary widely by app.

Import and Export Mechanics

The [ImportExport module](#) of [the book's primary sample project](#) demonstrates copying databases for backup/restore or import/export purposes.

Its UI consists mostly of three big buttons:

- Add some random data to the database
- Export the database to a location that the user chooses via `ActivityResultContracts.CreateDocument`
- Import the database from a location that the user chooses via `ActivityResultContracts.OpenDocument`

Also, at the bottom, there is a `TextView` showing the number of rows in our one database table and how old the oldest row is.

`RandomRepository` is the repository that not only mediates conventional database operations but also handles import and export operations:

```
package com.commonware.room.importexport

import android.content.Context
import android.net.Uri
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.asCoroutineDispatcher
import kotlinx.coroutines.sync.Mutex
import kotlinx.coroutines.sync.withLock
import kotlinx.coroutines.withContext
import java.util.concurrent.Executors
import kotlin.random.Random

class RandomRepository(
    private val context: Context,
    private val appScope: CoroutineScope
) {
    private val mutex = Mutex()
    private val dispatcher =
        Executors.newSingleThreadExecutor().asCoroutineDispatcher()
    private var db: RandomDatabase? = null

    suspend fun summarize() =
        withContext(dispatcher) {
            mutex.withLock {
                if (RandomDatabase.exists(context)) {
```

```
        db().randomStore().summarize()
    } else {
        Summary(count = 0)
    }
}
}

suspend fun populate() {
    withContext(dispatcher + appScope.coroutineContext) {
        mutex.withLock {
            val count = Random.nextInt(100) + 1

            db().randomStore().insert((1..count).map { RandomEntity(0) })
        }
    }
}

suspend fun export(uri: Uri) {
    withContext(dispatcher + appScope.coroutineContext) {
        mutex.withLock {
            db?.close() // ensure no more access and single database file
            db = null

            context.contentResolver.openOutputStream(uri)?.use {
                RandomDatabase.copyTo(context, it)
            }
        }
    }
}

suspend fun import(uri: Uri) {
    withContext(dispatcher + appScope.coroutineContext) {
        mutex.withLock {
            db?.close() // ensure no more access and single database file
            db = null

            context.contentResolver.openInputStream(uri)?.use {
                RandomDatabase.copyFrom(context, it)
            }
        }
    }
}

private fun db(): RandomDatabase {
    if (db == null) {
        db = RandomDatabase.newInstance(context)
    }
}
```

BACKING UP YOUR ROOM

```
    return db!!  
  }  
}
```

(from [ImportExport/src/main/java/com/commonsware/room/importexport/RandomRepository.kt](https://github.com/commonsware/room-importexport/blob/master/RandomRepository.kt))

As with some of the other examples, this sample uses Kotlin coroutines. `RandomRepository` get a `Context` and a `CoroutineScope` injected via `Koin` — the `Context` is to open a `RandomDatabase`, while the latter is for ensuring that data modification operations do not get canceled based on our `MainActivity` getting destroyed.

Functions like `summarize()` (used to get the data for the `TextView`) and `populate()` (used to insert some random rows into the database table) are mostly normal. They have two differences compared to past samples:

1. Both wrap the database I/O in a `Mutex`. `Mutex` is the coroutines approach for mutual exclusion. Only one bit of code can be operating inside of the `Mutex` (and its `withLock()` function) at a time. We use this `Mutex` in our `import()` and `export()` functions as well, and we use this to ensure that nothing tries working with our database while the import or export operations are ongoing. This is a crude approach, offering limited concurrency, and a more sophisticated app might want to do something... well, more sophisticated.
2. Both call a `db()` function to get the `RandomDatabase` instance to use. `db()`, in turn, lazy-creates the `RandomDatabase`, if it presently is `null`.

`import()` and `export()` also use the `Mutex`. However, inside of `withLock()`, they:

- `close()` our `RandomDatabase`, so our WAL files get cleared and we have a single database file
- Set the `db` property to `null`, so `db()` knows to re-open the database on the next regular bit of database I/O
- Copies the database to or from a location specified by a `Uri`, using `copyFrom()` and `copyTo()` functions on `RandomDatabase` to handle the actual copy operations:

```
package com.commonsware.room.importexport  
  
import android.content.Context  
import androidx.room.Database  
import androidx.room.Room  
import androidx.room.RoomDatabase  
import androidx.room.TypeConverters  
import java.io.InputStream  
import java.io.OutputStream
```

```
private const val DB_NAME = "random.db"

@Database(entities = [RandomEntity::class], version = 1)
@TypeConverters(TypeTransmogrifier::class)
abstract class RandomDatabase : RoomDatabase() {
    abstract fun randomStore(): RandomStore

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, RandomDatabase::class.java, DB_NAME).build()

        fun exists(context: Context) = context.getDatabasePath(DB_NAME).exists()

        fun copyTo(context: Context, stream: OutputStream) {
            context.getDatabasePath(DB_NAME).inputStream().copyTo(stream)
        }

        fun copyFrom(context: Context, stream: InputStream) {
            val dbFile = context.getDatabasePath(DB_NAME)

            dbFile.delete()

            stream.copyTo(dbFile.outputStream())
        }
    }
}
```

(from [ImportExport/src/main/java/com/commonsware/room/importexport/RandomDatabase.kt](https://commonsware.com/licenses/room/importexport/RandomDatabase.kt))

The createFromFile() Alternative

In [an earlier chapter](#), we explored `createFromAsset()` as a way to set up a Room database from an existing database copy, in this case packaged as an asset.

There is also `createFromFile()`. This works just like `createFromAsset()`, but it takes a `File` parameter that should point to a readable file containing the database copy.

For restoring from a backup or importing from a database, this is more convenient than is the manual-copy-from-a-stream approach used in this chapter's sample. However, it only works with files, which is an unfortunate limitation. The sample uses `Uri`, not `File`, to identify the location of the backup, and so `createFromFile()` is not an available alternative.

SQLite Clients

Sometimes, it would be nice to look at what is in our Room-powered database. For example, it might simplify tracking down a bug if we could peek inside our tables and see what data is inside of them.

For that, you will need some form of SQLite client.

Fortunately, SQLite is open source and has been around for a long time. There are quite a few clients available to you to choose from, to find one that meets your needs and fits your workflow.

Database Inspector

A leading candidate, starting with Android Studio 4.1, is Android Studio itself, through its Database Inspector tool.

Getting To Your Database

How you get to the Database Inspector varies by Android Studio version:

SQLITE CLIENTS

- For 4.1.x and 4.2.x, look for the “Database Inspector” tool, by default docked on the bottom edge
- For Arctic Fox, look for the “App Inspection” tool, by default docked on the bottom edge — “Database Inspector” is a tab in the resulting UI

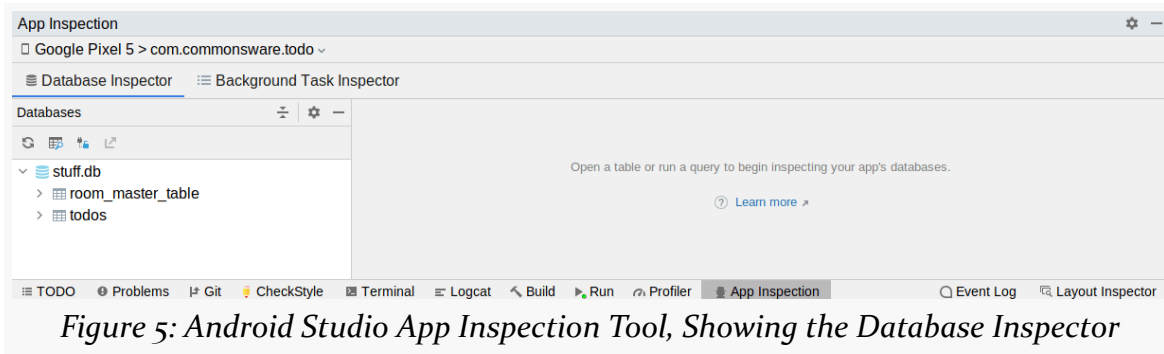


Figure 5: Android Studio App Inspection Tool, Showing the Database Inspector

In the strip just below the title, you will see the particular app that Database Inspector is offering to inspect, or “Select Process” if Database Inspector does not know of a suitable process (e.g., your phone is not plugged in). You can switch to something else by clicking on that entry and choosing the desired device (or emulator) and process from drop-down menus.

Note that loading the database details seems to take far longer than it should — be patient!

Database Inspector appears to look for databases in the stock location that Room and most other apps place them — in this case, it shows `stuff.db` in the tree on the left. Inside, it shows two tables:

- `todos`, akin to the ones in various samples in this book
- `room_master_table`, a table created by Room itself to help manage its work

Seeing the Schema

The tree goes beyond the database and the tables in that database. Folding open a table gives you the table's schema:

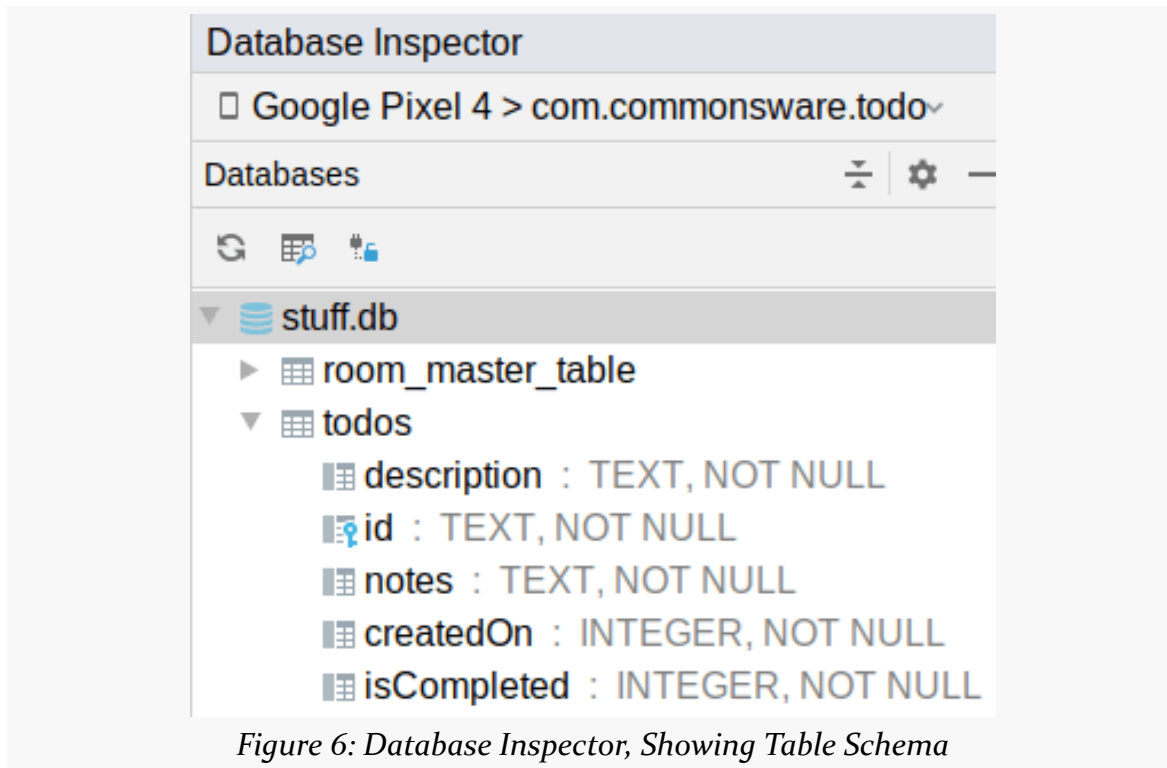


Figure 6: Database Inspector, Showing Table Schema

Performing Operations

In the toolbar above that tree, the toolbar button that looks like a grid with a magnifying glass will open a tab for you to be able to execute queries against the selected database:

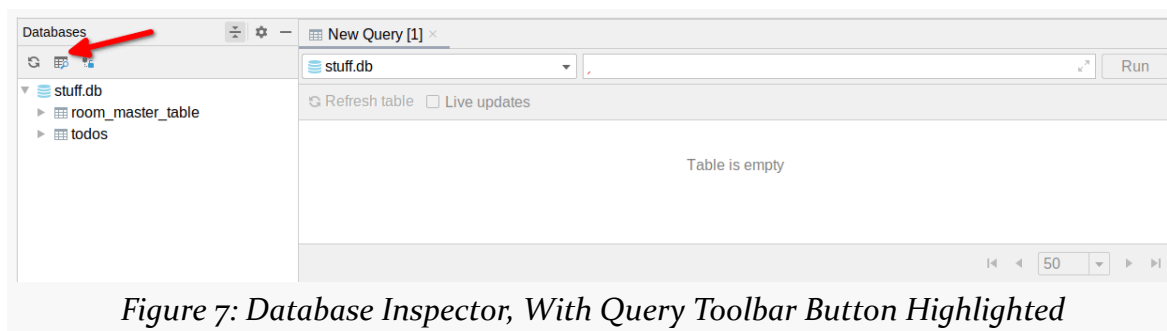


Figure 7: Database Inspector, With Query Toolbar Button Highlighted

SQLITE CLIENTS

The drop-down controls the database, and the field above it is where you can enter a SQL expression. Clicking the “Run” button then executes your SQL expression, with a grid showing you the results:

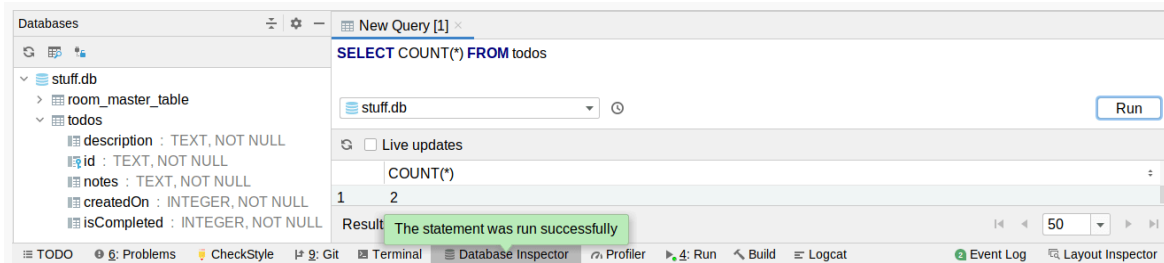


Figure 8: Database Inspector, Showing Query, Results, and Congratulatory Tooltip

Using Your DAO

The Database Inspector also ties into the Android source editor. For some queries on your DAO classes, you can use a gutter icon to trigger the same query to run in the Database Inspector:

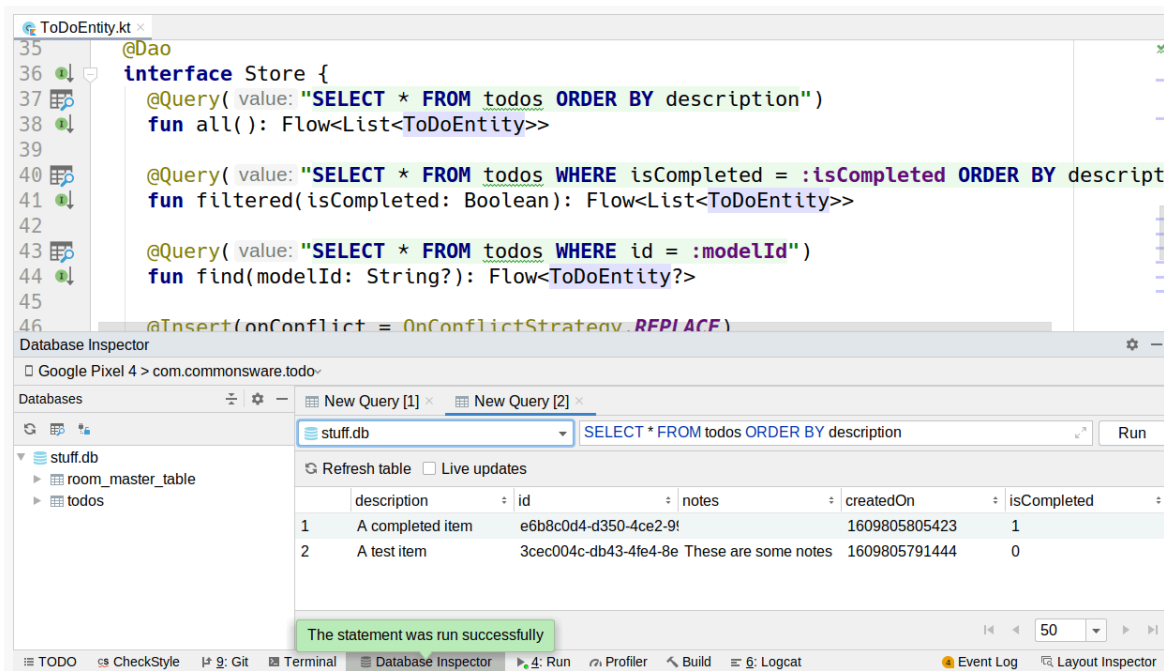


Figure 9: Android Studio Editor, Showing Query Gutter Icons, Plus `all()` Results in Database Inspector

Those icons will only appear while the Database Inspector is running and is inspecting your app's database.

Updating the Output

A popular thing to do with databases is to change the data.

If you change the data within the app, you have two ways of seeing those results reflected in the Database Inspector output.

The low-impact approach is simply to click the “Refresh table” button above your query output (note: the “table” referred to in button caption is the UI table, not a database table). This will re-query the database and update the output to match.

The alternative is to check the “Live updates” checkbox:

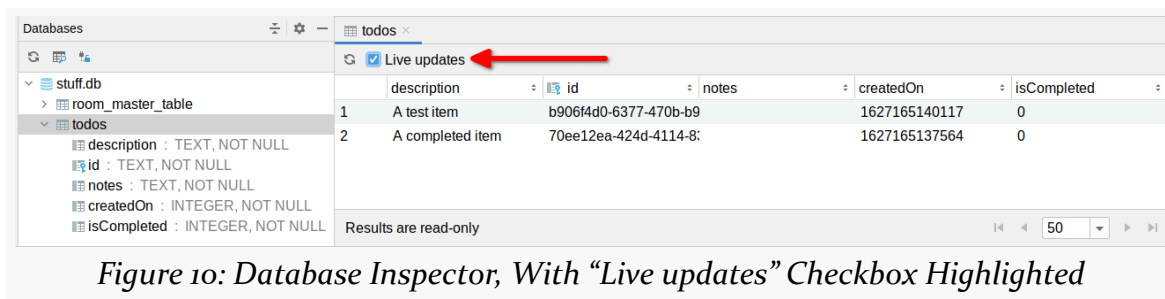


Figure 10: Database Inspector, With “Live updates” Checkbox Highlighted

This will hook into the same “invalidation tracker” that Room itself uses to know to deliver new updates to your app via LiveData, Flow, etc. The table showing the query results will update in near-real-time as your app makes changes to the data.

Updating the App

When “Live updates” is checked, a message appears below the output: “Results are read-only”. You will see the same message when running one of your DAO functions, or if you run a custom query yourself.

However, if you just double-click the table name in the tree on the left, that message does not appear. And, in that case, the UI table is itself live — any changes that you make there will be reflected, in near-real-time, in your app's UI. Just double-click on a cell, type in the revised value, and press or .

This too ties into Room's invalidation tracker. The data change that you make

triggers that tracker and in turn causes any of your reactive observers to get fresh data from Room, reflecting whatever change you make.

Note, though, that the Database Inspector does not perform much data validation. For example, a Boolean property maps to an INTEGER column in the table, with 0 meaning false and 1 meaning true... but Database Inspector will be happy to let you fill in 3 as the value. And, since in SQLite column types are hints, Database Inspector will be happy to let you fill in foo as the value of an INTEGER column. While Database Inspector is happy, Room might not be, so be careful when modifying your app's data via the Inspector this way.

DB Browser for SQLite

While there are a variety of standalone SQLite clients, in terms of desktop use, [DB Browser for SQLite](#) is arguably the most popular. It is open source, available for Linux/macOS/Windows, and is fairly easy to use.

Copying Your Database

It has no specific integration with Android or Android Studio, though, which means that you will need to copy your database off of your device or emulator and onto your development machine.

Ideally, your app's process is terminated when you do this, so your app does not attempt to use the database while the copy operation is ongoing.

If you are using ordinary Room, your database will be in the default location for SQLite databases for your app. From the standpoint of development tools, for the primary device user, that will be:

```
/data/data/.../databases/
```

...where ... is your application ID.

In there, you will find a database file, with the name that you gave it in your RoomDatabase (e.g., `stuff.db`). Particularly if the app opened the database and did not explicitly close it, you will also see two additional files, with the same name as the database plus `-shm` and `-wal` extensions. You will need to copy all of these files to your development machine, most likely using Device File Explorer from Android Studio.

SQLITE CLIENTS

You can then open it in DB Browser for SQLite using the “Open Database” toolbar button, selecting the database file itself (not the `-shm` or `-wal` files, if any).

Basic Database Operations

Like Database Inspector, DB Browser for SQLite gives you a tree of the various tables in the “Database Structure” tab, where you can see the schema for a table:

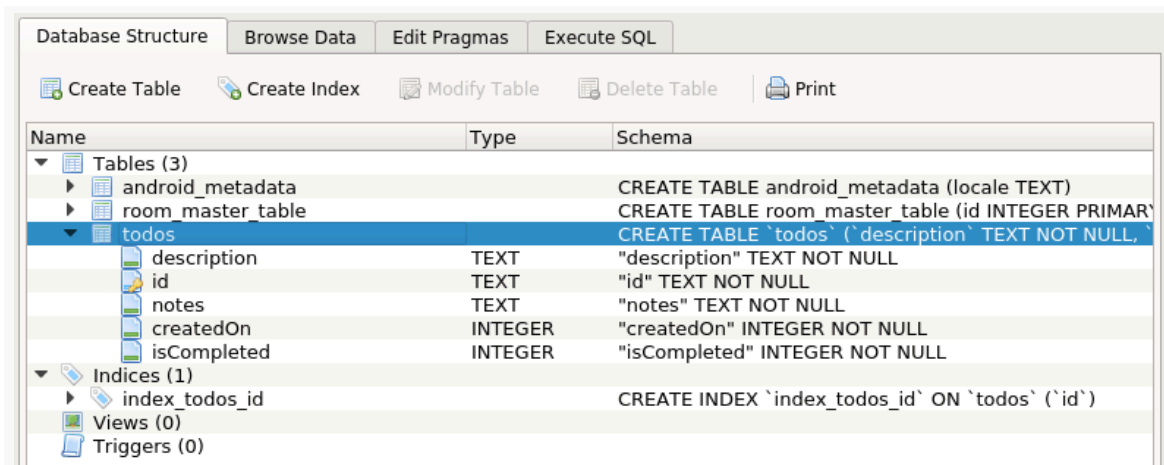


Figure 11: DB Browser for SQLite, Showing Table Schema

The “Browse Data” tab gives you a tabular view of the contents of a selected table, chosen via the drop-down in the tab’s own toolbar:

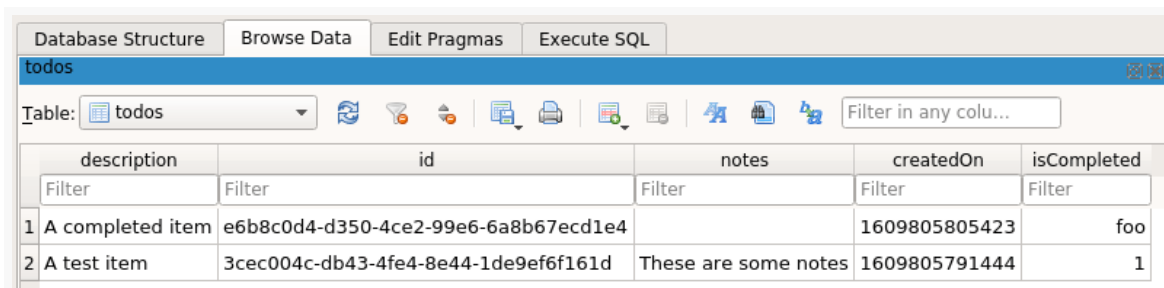


Figure 12: DB Browser for SQLite, Showing Table Contents

The “Execute SQL” tab lets you enter in your own queries or other operations (e.g., INSERT statements) and run them against your database. For queries or other statements that return results, you get a table showing those results:

Note, though, that if you modify the data and wish to persist those changes, you need to click the “Write Changes” toolbar button.

When you are done, if you click the “Close Database” button, the SQLite database will be closed cleanly, leaving you with just the database file and without any `-shm` or `-wal` file.

If you wish, you could then copy the database back to the device, using Device File Explorer. However:

- Be sure to terminate your app’s process before you do this, so you do not replace SQLite files behind Room’s back
- If your app’s data has `-shm` and/or `-wal` files, and you used “Close Database” to get a clean single-file copy of your database, in addition to copying that database to your device, you will need to remove the device’s `-shm` and `-wal` files to match

Flipper

Another possibility is to integrate [Flipper](#). Flipper is a library from Facebook that you can integrate into debug builds of your app. It opens a port that a Flipper-supplied desktop app can connect to. Depending on what Flipper plugins you have enabled, that desktop app can do different things... with one being giving you Database Inspector-style access to your app’s database.

[The PagedFTS module](#) of [the book’s primary sample project](#) — profiled [elsewhere in the book](#) — happens to integrate Flipper.

Adding Dependencies

Flipper takes an approach used by a few debugging-centric libraries: have some real dependencies to add to debug builds and a “no-op” dependency to add to release builds. The objective is to allow for configuration to be the same regardless of build type, while avoiding the risk of shipping debug-related code to your users.

PagedFTS, therefore, has two `debugImplementation` lines and one `releaseImplementation` line in its dependencies list:

```
debugImplementation 'com.facebook.flipper:flipper:0.96.1'
debugImplementation 'com.facebook.soloader:soloader:0.10.1'

releaseImplementation 'com.facebook.flipper:flipper-noop:0.96.1'
```

(from [PagedFTS/build.gradle](#))

Configuring Flipper for Database Debugging

Flipper is designed to be configured from `onCreate()` of a custom `Application` class. `PagedFTS` has such a class, named `KoinApp`:

```
package com.commonware.room.fts

import android.app.Application
import com.facebook.flipper.android.AndroidFlipperClient
import com.facebook.flipper.android.utils.FlipperUtils
import com.facebook.flipper.plugins.databases.DatabasesFlipperPlugin
import com.facebook.flipper.plugins.inspector.DescriptorMapping
import com.facebook.flipper.plugins.inspector.InspectorFlipperPlugin
import com.facebook.soloader.Soloader
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.dsl.module

class KoinApp : Application() {
    private val koinModule = module {
        single { BookDatabase.newInstance(androidContext()) }
        single { BookRepository(get()) }
        viewModel { BookViewModel(get()) }
        viewModel { (search: String) -> SearchViewModel(search, get()) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@KoinApp)
            modules(koinModule)
        }

        if (BuildConfig.DEBUG && FlipperUtils.shouldEnableFlipper(this)) {
            Soloader.init(this, false)
        }
    }
}
```

SQLITE CLIENTS

```
        AndroidFlipperClient.getInstance(this).also { client ->
            client.addPlugin(DatabasesFlipperPlugin(this))

            client.start()
        }
    }
}
```

(from [PagedFTS/src/main/java/com/commonsware/room/fts/KoinApp.kt](#))

Part of `onCreate()` is setting up Koin for dependency inversion. However, the last few lines of `onCreate()` are focused on Flipper initialization:

```
if (BuildConfig.DEBUG && FlipperUtils.shouldEnableFlipper(this)) {
    SoLoader.init(this, false)

    AndroidFlipperClient.getInstance(this).also { client ->
        client.addPlugin(DatabasesFlipperPlugin(this))

        client.start()
    }
}
```

(from [PagedFTS/src/main/java/com/commonsware/room/fts/KoinApp.kt](#))

The key, for being able to debug databases using Flipper, is adding the `DatabasesFlipperPlugin` to Flipper. Flipper operates based on a series of plugins, and this one provides access to SQLite databases, at least those in standard locations. [The documentation for that plugin](#) provides some other options, particularly for databases residing in non-standard locations.

See [the Flipper documentation](#) for more details about this initialization/configuration process.

Obtaining and Using the Desktop App

The Flipper site has [download options for Linux, macOS, and Windows](#) versions of the desktop app. Windows and Linux are bare ZIP files containing the React Native app; macOS gets a proper DMG file.

SQLITE CLIENTS

If you run your Flipper-enabled app, then run the Flipper desktop app, your app will appear in a section in the accordion on the left of the desktop app, and you can enable the Databases tool via a switch:

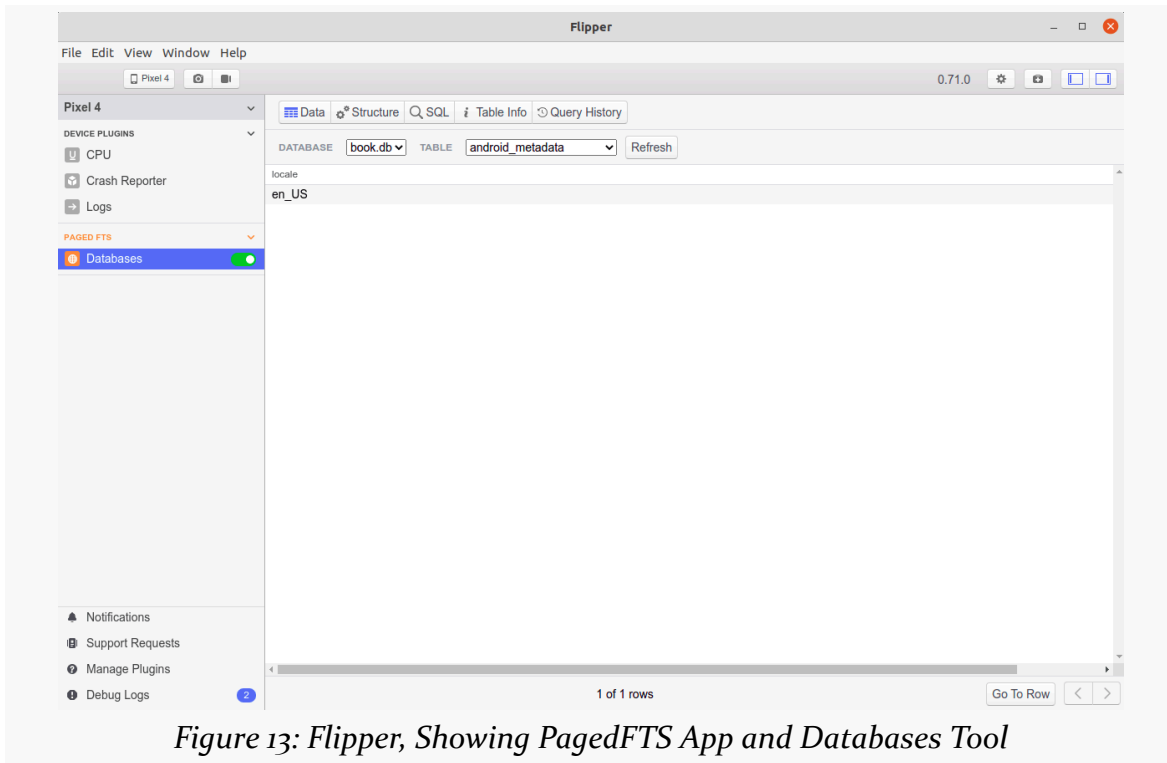


Figure 13: Flipper, Showing PagedFTS App and Databases Tool

SQLITE CLIENTS

You can choose the database and table via drop-downs towards the top of the Databases content pane. The default view is “Data”, showing you the contents of your selected table:

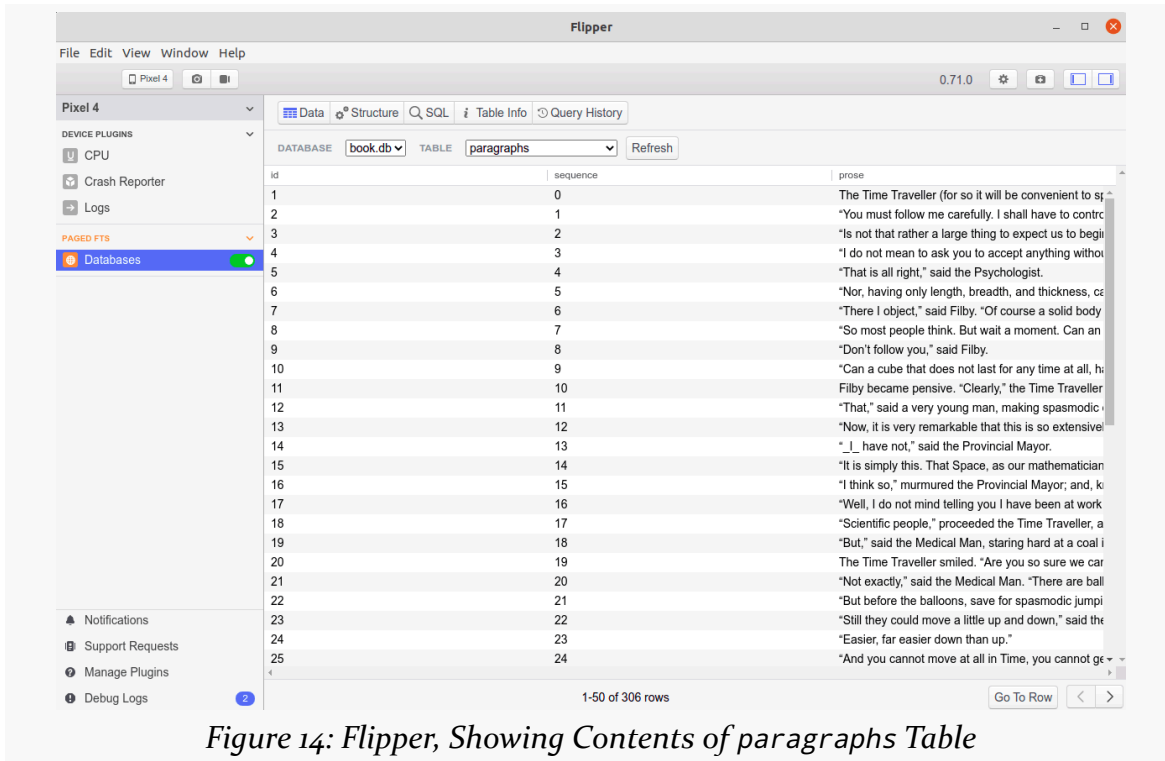


Figure 14: Flipper, Showing Contents of paragraphs Table

SQLITE CLIENTS

You can view your table schema either in the form of SQL via the “Table Info” tab or as a table in the “Structure” tab. And the “SQL” tab lets you execute arbitrary SQL statements, showing you query results below the text area:

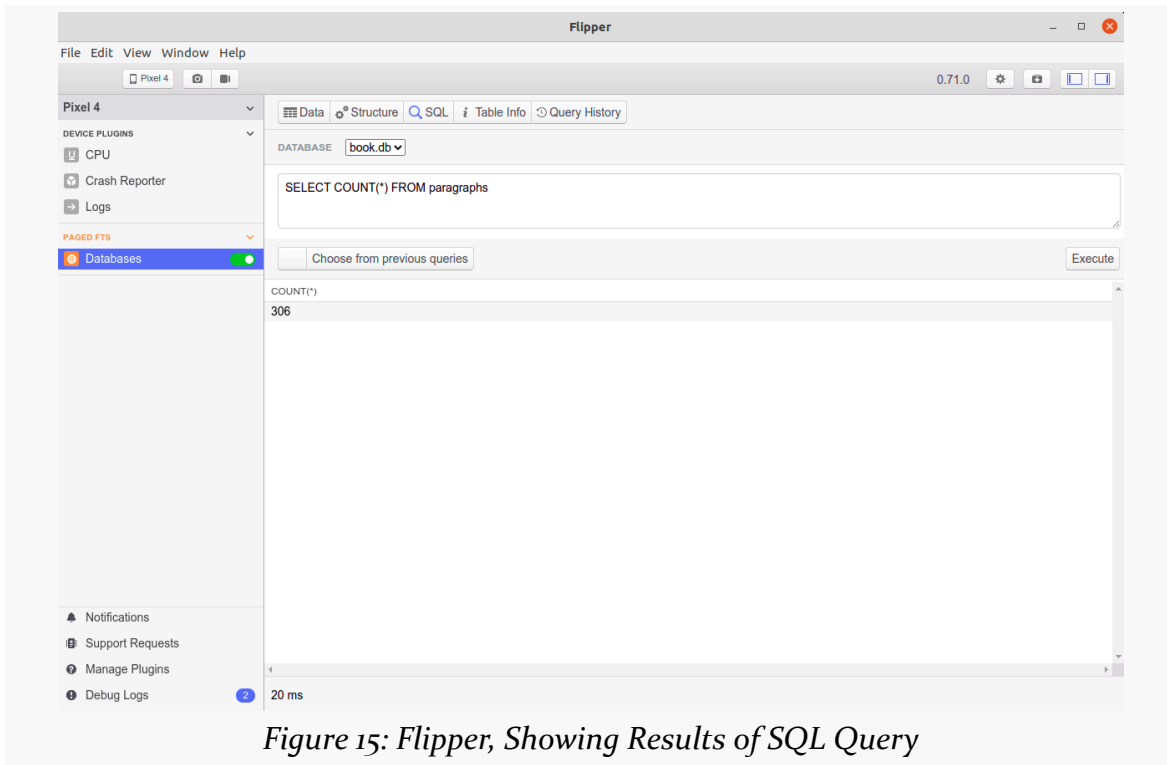


Figure 15: Flipper, Showing Results of SQL Query

Room Security

SQLCipher for Android

Room, by default, works with the device's stock copy of SQLite. This is fine, as far as it goes. However, from a security standpoint, SQLite stores its data unencrypted. Some apps should be considering encrypting their data “at rest”, when it is stored in a database, to protect their users.

Fortunately, as noted in [an earlier chapter](#), Room supports a pluggable SQLite implementation, and so we can plug in a SQLite edition that supports encryption, such as SQLCipher for Android. This chapter will outline how to do this.

Introducing SQLCipher for Android

Since SQLite is public domain, it is easy for people to grab the source code and hack on it. SQLite also offers an extension system, making it relatively easy for developers to add functionality with a minimal number of changes to SQLite's core code. As a result, a few encryption options for SQLite have been published.

One of these is [SQLCipher](#), whose development is overseen by [Zetetic](#). This offers transparent AES-256 encryption of everything in the database: data, schema, etc.

With the help of the Guardian Project, Zetetic released [SQLCipher for Android](#). This combines a pre-compiled version of SQLite with Java classes that mimic an old edition of Android's native SQLite classes (e.g., `SQLiteOpenHelper`). SQLCipher for Android is open source, and if you can live with the increase in app size due to the native binaries, it is an effective solution.

And, in 2019, Zetetic started offering support for the `SupportSQLite*` APIs that allow SQLCipher for Android to be plugged into Room.

But First, A To-Do Reminder

Back in the chapter on [reactive threading solutions](#), we looked at some code for tracking to-do items. This code was derived from the sample app built up in [Exploring Android](#). That chapter explored variations of this code that used LiveData, RxJava, and coroutines.

The SQLCipher material in this book continues its riff on that example, so let's quickly review the core Room elements of the to-do code, specifically the coroutines edition.

The Entity, the Model, and the Store

Our one Room entity is `ToDoEntity` implemented as a data class:

```
package com.commonware.todo.repo

import androidx.room.*
import kotlinx.coroutines.flow.Flow
import org.threeten.bp.Instant
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Instant = Instant.now(),
    val isCompleted: Boolean = false
) {
    constructor(model: ToDoModel) : this(
        id = model.id,
        description = model.description,
        isCompleted = model.isCompleted,
        notes = model.notes,
        createdOn = model.createdOn
    )

    fun toModel(): ToDoModel {
        return ToDoModel(
            id = id,
            description = description,
            isCompleted = isCompleted,
```

```
        notes = notes,
        createdOn = createdOn
    )
}

@Dao
interface Store {
    @Query("SELECT * FROM todos")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String): Flow<ToDoEntity?>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(vararg entities: ToDoEntity)

    @Delete
    suspend fun delete(vararg entities: ToDoEntity)
}
```

(from [Coroutines/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

To simulate a more complex scenario, an entity knows how to convert itself to and from a `ToDoModel`:

```
package com.commonsware.todo.repo

import org.threeten.bp.Instant
import java.util.*

data class ToDoModel(
    val description: String,
    val id: String = UUID.randomUUID().toString(),
    val isCompleted: Boolean = false,
    val notes: String = "",
    val createdOn: Instant = Instant.now()
) {
}
```

(from [Coroutines/src/main/java/com/commonsware/todo/repo/ToDoModel.kt](#))

In theory, that model might have a significantly different representation than does the entity, from data type conversions to having direct references to other models derived from other entities.

`ToDoEntity` contains a nested `Store @Dao` interface with functions for working with

entities. Two (`all()` and `find()`) are queries and return Flow objects, while the `save()` and `delete()` functions are marked with `suspend`. Hence, our DAO uses coroutines for read and write operations.

The Database and the Transmogrifier

Our `@Database` class is `ToDoDatabase`:

```
package com.commonware.todo.repo

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters

private const val DB_NAME = "stuff.db"

@Database(entities = [ToDoEntity::class], version = 1)
@TypeConverters(TypeTransmogrifier::class)
abstract class ToDoDatabase : RoomDatabase() {
    abstract fun todoStore(): ToDoEntity.Store

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME).build()

        fun newInstance(context: Context) =
            Room.inMemoryDatabaseBuilder(context, ToDoDatabase::class.java)
                .build()
    }
}
```

(from [Coroutines/src/main/java/com/commonware/todo/repo/ToDoDatabase.kt](#))

It ties in a `TypeTransmogrifier` class that offers type converters between Instant timestamps and Long objects for storage in Room:

```
package com.commonware.todo.repo

import androidx.room.TypeConverter
import org.threeten.bp.Instant
import java.util.*

class TypeTransmogrifier {
    @TypeConverter
```

```
fun instantToLong(timestamp: Instant?) = timestamp?.toEpochMilli()

@TypeConverter
fun longToInstant(timestamp: Long?) =
    timestamp?.let { Instant.ofEpochMilli(it) }
}
```

(from [Coroutines/src/main/java/com/commonsware/todo/repo/TypeTransmogifier.kt](https://github.com/commonsware/todo-repo/blob/master/src/main/java/com/commonsware/todo/repo/TypeTransmogifier.kt))

ToDoDatabase offers newInstance()-style factory functions both for normal use and for tests, with the latter being backed purely by memory instead of storing data on disk.

The Repository

ToDoRepository hides all of those details, exposing its own coroutine-based API that, in particular, uses a custom CoroutineScope to ensure that write operations are not canceled early due to user navigation and the resulting clearing of viewmodels:

```
package com.commonsware.todo.repo

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.withContext

class ToDoRepository(
    private val store: ToDoEntity.Store,
    private val appScope: CoroutineScope
) {
    fun items(): Flow<List<ToDoModel>> =
        store.all().map { all -> all.map { it.toModel() } }

    fun find(id: String): Flow<ToDoModel?> = store.find(id).map { it?.toModel() }

    suspend fun save(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.save(ToDoEntity(model))
        }
    }

    suspend fun delete(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.delete(ToDoEntity(model))
        }
    }
}
```

```
}  
}  
}
```

(from [Coroutines/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

The Basics of SQLCipher for Android

The [ToDoCrypt module](#) of [the book's primary sample project](#) contains its own edition of those classes, plus the whole to-do app UI that employs them. In addition, this app adds SQLCipher for Android, in case the user really wants to protect those to-do items.

Adding the Dependency

Zetetic maintains a standard Android AAR artifact, available in Maven Central and its mirrors (e.g., Bintray's JCenter), using `net.zetetic:android-database-sqlcipher` as the base Maven coordinates. So, `ToDoCrypt` adds that library to the roster of libraries that it pulls in via the dependencies closure in the module's `build.gradle` file:

```
implementation "net.zetetic:android-database-sqlcipher:4.4.2"
```

(from [ToDoCrypt/build.gradle](#))

Creating and Applying the Factory

That gives us access to a `SupportFactory` class. This is an implementation of `SupportSQLiteHelper.Factory` and serves as the “glue” between SQLCipher for Android and clients like Room.

The simplest `SupportFactory` constructor takes a `byte[]` that represents the passphrase for the database. This will be used in two cases:

- If the database does not yet exist, SQLCipher for Android will create one, and this passphrase will be used for encrypting the database
- If the database does exist, SQLCipher for Android will try to open it using this passphrase to decrypt the database

How you get that `byte[]` for the passphrase is up to you. In this sample, we take a very easy and very lousy approach: hardcoding it. So, we have a `PASSPHRASE` constant and use that in the `SupportFactory` constructor:

```
package com.commonware.todo.repo

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters
import net.sqlcipher.database.SupportFactory

private const val DB_NAME = "stuff.db"
private const val PASSPHRASE = "sekr1t"

@Database(entities = [ToDoEntity::class], version = 1)
@TypeConverters(TypeTransmogrifier::class)
abstract class ToDoDatabase : RoomDatabase() {
    abstract fun todoStore(): ToDoEntity.Store

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME)
                .openHelperFactory(SupportFactory(PASSPHRASE.toByteArray()))
                .build()
    }
}
```

(from [ToDoCrypt/src/main/java/com/commonware/todo/repo/ToDoDatabase.kt](#))

We pass that `SupportFactory` to `openHelperFactory()` on our `RoomDatabase.Builder`, and from there, Room will take over and integrate with SQLCipher for Android.

Using the Database

The beauty of the `SupportSQLite*` family of APIs is that, for the most part, Room clients neither know nor care about the actual SQLite implementation. `ToDoEntity` and `ToDoEntity.Store` do not need anything special for SQLCipher for Android. Even `ToDoDatabase` has just the change to add that one `openHelperFactory()` call — nothing else is affected. `ToDoRepository` and its clients (e.g., viewmodels) are also unaffected. So, everything that has been covered to date in the book just works, with the added improvement of encryption.

Using the Database... Outside the App

To work with a database encrypted by SQLCipher for Android, you will need a [client](#) that has SQLCipher compiled in. SQLCipher databases are portable across

SQLCIPHER FOR ANDROID

platforms, just as SQLite databases are, but plain SQLite clients will not know how to deal with SQLCipher's encryption scheme. So, for example, neither Android Studio's Database Inspector nor the `sqlite3` binary that is part of Android itself will be able to work with SQLCipher for Android databases.

[DB Browser for SQLite](#), however, does support SQLCipher.

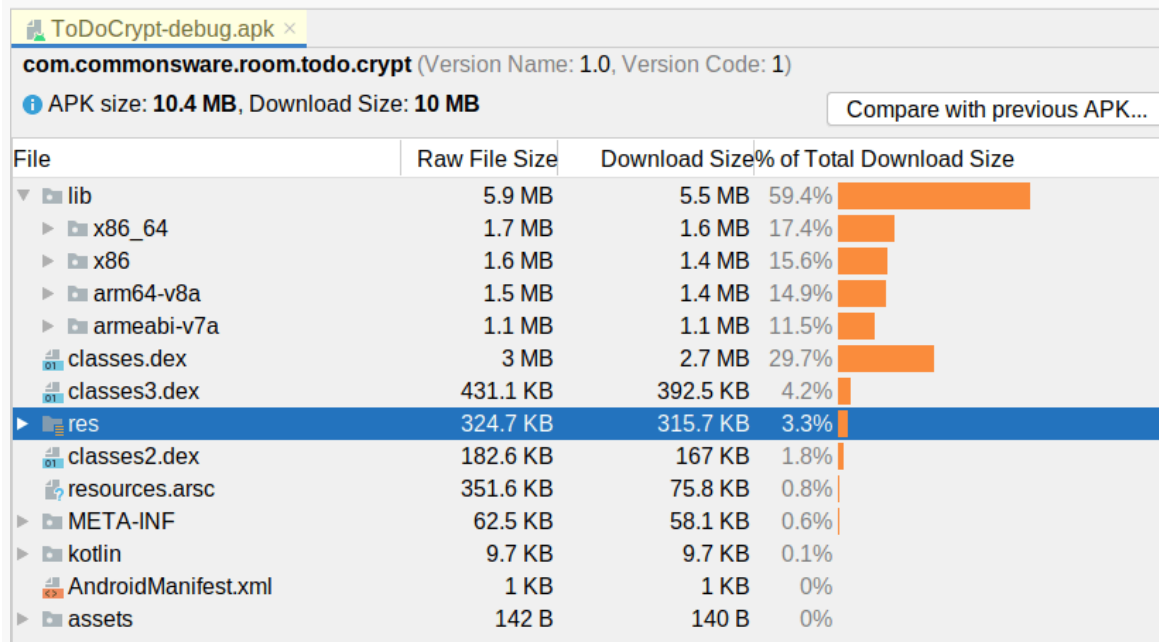
The Costs of SQLCipher for Android

Being able to get high-grade encryption for one dependency and (seemingly) one line of code in the app seems wonderful. And, in truth, in scenarios that need encryption, it is wonderful. It is even [both “free as in beer” and “free as in speech”](#).

However, there are costs... just not in terms of money or rights.

APK Size

The debug build of `ToDoCrypt` is 10MB. Of that, nearly 6MB comes from NDK binaries:



File	Raw File Size	Download Size	% of Total Download Size
lib	5.9 MB	5.5 MB	59.4%
x86_64	1.7 MB	1.6 MB	17.4%
x86	1.6 MB	1.4 MB	15.6%
arm64-v8a	1.5 MB	1.4 MB	14.9%
armeabi-v7a	1.1 MB	1.1 MB	11.5%
classes.dex	3 MB	2.7 MB	29.7%
classes3.dex	431.1 KB	392.5 KB	4.2%
res	324.7 KB	315.7 KB	3.3%
classes2.dex	182.6 KB	167 KB	1.8%
resources.arsc	351.6 KB	75.8 KB	0.8%
META-INF	62.5 KB	58.1 KB	0.6%
kotlin	9.7 KB	9.7 KB	0.1%
AndroidManifest.xml	1 KB	1 KB	0%
assets	142 B	140 B	0%

Figure 16: APK Analyzer, Analyzing `ToDoCrypt` Debug Build

SQLCipher for Android includes a full copy of SQLite (with the SQLCipher extensions installed), for four CPU architectures. Using ABI splits or App Bundles, the cost for individual users can be a lot less, dropping that 6MB to 1-2MB. For some apps, this will not be a big problem; for other apps (and other user bases), adding that kind of size to the APK could be a deal-breaker.

Runtime Performance

Considering that everything is being encrypted and decrypted, the performance of SQLCipher for Android is fairly reasonable.

The biggest expense is when you open the database. That usually is when those NDK libraries will be loaded. Also, SQLCipher uses “key stretching” (thousands of rounds of PBKDF2) to convert your supplied passphrase into the actual encryption key, and this takes a bit of time. So, if your app loads data out of the database when the app starts, as ToDoCrypt does, this make take longer than you are used to. This specific edition of ToDoCrypt does *not* have a loading state, and depending on your test device, you will see why you need one — the UI briefly shows an empty state before any to-do items get loaded. However, since Room and modern Android architecture tend to steer developers towards opening the database just once per process, this cost is only incurred once per process.

After that, individual database operations are more expensive, but usually not dramatically more expensive. The cost of the encryption and decryption will be roughly proportional to the amount of data that needs to be encrypted or decrypted. As a result:

- Small database operations will not be perceptively slower with SQLCipher for Android than with plain SQLite
- Large database operations — ones that were somewhat painful already — may be a substantially more painful

This just means that you will need to spend some extra time in optimizing your database access, such as adding indices to avoid SQLCipher for Android having to decrypt an entire table to walk through all rows in a “table scan”-style query.

Complexity

The code changes in the chapter were trivial. That is because our approach towards managing the passphrase was trivial. Unfortunately, that also means that the security benefit is trivial, as an attacker would not have a difficult time finding that

hard-coded passphrase.

In the end, the complexity of SQLCipher for Android comes not from the library, but rather from passphrase management. We will explore that more in [the next chapter](#).

SQLCipher and Passphrases

The fun with SQLCipher for Android — particularly when used with Room and dependency inversion frameworks — is in getting the passphrase to use for the database.

In [the preceding chapter](#), we hard-coded the passphrase. This is simple but insecure. Every user has the same passphrase, and that passphrase is fairly easy to find out via reverse-engineering the app's APK. At that point, SQLCipher for Android adds no real security over regular SQLite, so you have all the costs (e.g., APK size, runtime performance) with no benefits.

Instead, we need to have a passphrase that is unique to the app installation. Ideally, attackers would have no way to find out the passphrase for any user. But, even if they could get the passphrase for one user, that would only affect that user — it does not immediately compromise all other users.

So, in this chapter, we will explore alternative ways of setting up passphrases.

Generating a Passphrase

The classic solution for this problem is to have the user provide their own passphrase. We will explore that option [later in the chapter](#).

However, typically, that solution has issues:

- Users are far too likely to choose poor passphrases, [such as 12345](#)
- Users have to enter that passphrase every time, which makes using better passphrases more annoying

The nice thing about the hard-coded passphrase is that the user does not have to worry about it. They just use the app normally.

We could generate a per-installation passphrase and use that to encrypt the database. We then wind up with a “chicken and egg” problem: where do we store that generated passphrase, such that it cannot be accessed by attackers? We could store it in an ordinary file, but then any attacker that can get to the database file can also get to the passphrase file, and our security is blown.

We could store the generated passphrase in an encrypted file. This gives us another form of the “chicken and egg” problem: how are we going to encrypt it? After all, most encryption systems, like SQLCipher, require a passphrase, and so if encrypting a passphrase requires *another* passphrase, we seem to have gotten nowhere.

However, for plain files, Google has better options for encryption. In particular, we can use the Security library from the Jetpack. This encrypts data using encryption keys that (on most hardware) is stored in a hardware-backed “keystore”, one that is designed to be tamper-resistant. So, we wind up with:

- A database encrypted by a generated passphrase
- That generated passphrase encrypted using a key that is inaccessible except from our app

We still have a risk of an attacker accessing our generated passphrase, as we will see [later in the chapter](#), but it really starts to ramp up the difficulty, and with some careful work, we can reduce the risk even further.

So, with all of that in mind, let us look at [the ToDoGen module](#) of [the book’s primary sample project](#). This is a clone of ToDoCrypt that we saw in the previous chapter, except that we use the Security library and a generated passphrase, rather than a hard-coded one.

Creating the Passphrase

The passphrase that we generate will never be entered by a user — it is purely for internal use. Hence, we do not need to worry about how easy it is to type in. However, [Zetetic requires binary keys to not have zero byte values](#).

So, for the purposes of this sample, we will go with a 32-byte passphrase, rejecting any that contain zero as a value:

```
private fun generatePassphrase(): ByteArray {
    val random = SecureRandom.getInstanceStrong()
    val result = ByteArray(PASSPHRASE_LENGTH)

    random.nextBytes(result)

    // filter out zero byte values, as SQLCipher does not like them
    while (result.contains(0)) {
        random.nextBytes(result)
    }

    return result
}
```

(from [ToDoGen/src/main/java/com/commonsware/todo/PassphraseRepository.kt](https://github.com/commonsware/todo-repo/blob/master/src/main/java/com/commonsware/todo/PassphraseRepository.kt))

Safely Storing the Passphrase

That passphrase is generated by a `PassphraseRepository`, backed by a `EncryptedFile` instance. If we do not have a passphrase file, we generate a passphrase and save it in an encrypted form. If we do have a passphrase file, we decrypt it to get the passphrase to use:

```
package com.commonsware.todo.repo

import android.content.Context
import androidx.security.crypto.EncryptedFile
import androidx.security.crypto.MasterKeys
import java.io.File
import java.security.SecureRandom

private const val PASSPHRASE_LENGTH = 32

class PassphraseRepository(private val context: Context) {
    fun getPassphrase(): ByteArray {
        val file = File(context.filesDir, "passphrase.bin")
        val encryptedFile = EncryptedFile.Builder(
            file,
            context,
            MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC),
            EncryptedFile.FileEncryptionScheme.AES256_GCM_HKDF_4KB
        ).build()

        return if (file.exists()) {
            encryptedFile.openFileInput().use { it.readBytes() }
        } else {
            generatePassphrase().also { passphrase ->

```

```
        encryptedFile.openFileOutput().use { it.write(passphrase) }
    }
}
}

private fun generatePassphrase(): ByteArray {
    val random = SecureRandom.getInstanceStrong()
    val result = ByteArray(PASSPHRASE_LENGTH)

    random.nextBytes(result)

    // filter out zero byte values, as SQLCipher does not like them
    while (result.contains(0)) {
        random.nextBytes(result)
    }

    return result
}
```

(from [ToDoGen/src/main/java/com/commonsware/todo/repo/PassphraseRepository.kt](#))

The specific recipe used here, in terms of the MasterKeys and various Scheme objects, comes from Google and appears to be a reasonable set of defaults.

That EncryptedFile class comes from the `androidx.security:security-crypto` added to the module's `build.gradle` file:

```
implementation "androidx.security:security-crypto:1.0.0"
```

(from [ToDoGen/build.gradle](#))

The `PassphraseRepository` itself is created by Koin in `ToDoApp`, as part of a module:

```
single { PassphraseRepository(androidContext()) }
```

(from [ToDoGen/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

The net effect is that our `PrefsRepository` stores that generated passphrase in the encrypted `SharedPreferences`, and that repository is available for other objects to use via Koin.

Using the Generated Passphrase

Our `ToDoDatabase` factory function needs that generated passphrase. So, we add it to the function signature and remove the hard-coded `PASSPHRASE` that `ToDoCrypt` used:

SQLCIPHER AND PASSPHRASES

```
package com.commonware.todo.repo

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters
import net.sqlcipher.database.SupportFactory

private const val DB_NAME = "stuff.db"

@Database(entities = [ToDoEntity::class], version = 1)
@TypeConverters(TypeTransmogrifier::class)
abstract class ToDoDatabase : RoomDatabase() {
    abstract fun todoStore(): ToDoEntity.Store

    companion object {
        fun newInstance(context: Context, passphrase: ByteArray) =
            Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME)
                .openHelperFactory(SupportFactory(passphrase))
                .build()
    }
}
```

(from [ToDoGen/src/main/java/com/commonware/todo/repo/ToDoDatabase.kt](#))

When Koin creates our `ToDoDatabase`, we pull the passphrase from `PassphraseRepository`, causing it to be generated if needed:

```
single {
    val passRepo: PassphraseRepository = get()

    ToDoDatabase.newInstance(androidContext(), passRepo.getPassphrase())
}
```

(from [ToDoGen/src/main/java/com/commonware/todo/ToDoApp.kt](#))

The result is no change from the user's standpoint, but we have replaced the hard-coded passphrase with a generated passphrase, backed by Jetpack-managed device encryption.

Pros and Cons

This required relatively little in the way of code changes. It does not change the user experience. And, it is a lot better from a security standpoint than having a hard-coded passphrase.

However, it is not perfect:

- [EncryptedFile may have problems on some hardware.](#)
- The database is unusable, except via the app. For example, you cannot copy the database off the device and use it with another client, unless you also arrange to get a copy of the passphrase. In development, that might be a matter of logging it with Logcat... so long as you do not accidentally ship such code. In production, this will cause issues with backup solutions, if they back up the database but not the encryption key needed to actually use that database.
- Anyone who can get into the phone can get into the database via the app. “Social engineering” attacks can trick users into handing over their phones in an unlocked state. This can be improved somewhat by creating a custom MasterKeys that [includes options like setUserAuthenticationRequired\(true\).](#)
- The implementation here keeps things simple and does the disk I/O for the EncryptedFile on the current thread, which may well be the main application thread. This is not ideal, and a production-grade app ideally would do a better job.

Collecting a Passphrase

Another option is to have the passphrase be stored in the user’s memory. We have them choose a passphrase when we go to create the database, and we have them supply that passphrase again later when opening the database in a fresh app process.

Adding a Passphrase Field

One downside to this approach is that we have to add some UI to our app to collect that passphrase from the user.

To keep the example simple — if perhaps not very pretty — this example just shoves a password EditText and a Button into the layout that we use for the list of to-do items.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
tools:context=".ui.MainActivity">

<TextView
    android:id="@+id/empty"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/msg_empty"
    android:textAppearance="?android:attr/textAppearanceMedium"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/items"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<EditText
    android:id="@+id/passphrase"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:hint="@string/passphrase_hint"
    android:importantForAutofill="no"
    android:inputType="textPassword"
    android:maxLines="1"
    app:layout_constraintBottom_toTopOf="@id/open"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_chainStyle="packed" />

<Button
    android:id="@+id/open"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="@string/button_open"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/passphrase" />

<androidx.constraintlayout.widget.Group
```

```
android:id="@+id/authGroup"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:visibility="gone"
app:constraint_referenced_ids="passphrase,open"
tools:visibility="visible" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ToDoUser/src/main/res/layout/todo_roster.xml](#))

The objective is to then show those widgets at the outset, so we can collect the passphrase from the user:

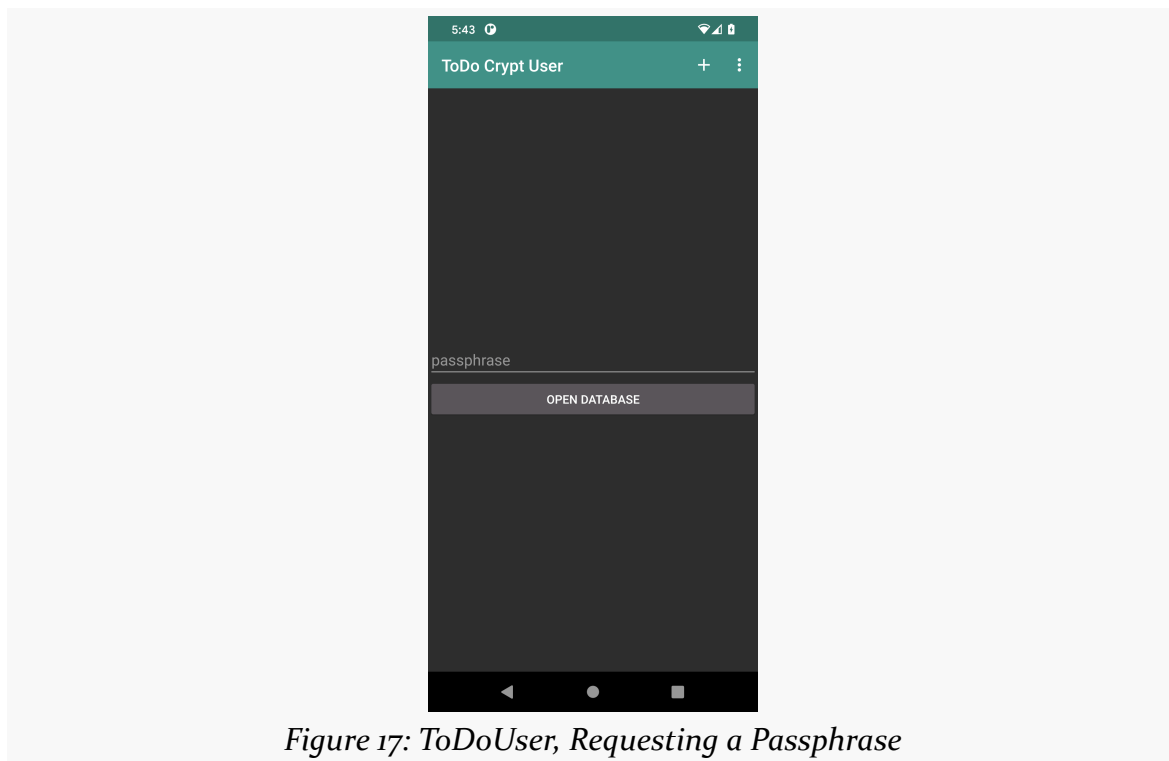


Figure 17: ToDoUser, Requesting a Passphrase

Detecting We Need a Passphrase

Our viewmodel will need to track whether or not we need to be asking for the passphrase and should be showing those new widgets. So, we have our view-state — here called `RosterViewState` — wrap both the list of to-do items and a flag indicating whether or not authentication is required:

SQLCIPHER AND PASSPHRASES

```
data class RosterViewState(  
    val items: List<ToDoModel> = listOf(),  
    val authRequired: Boolean = false  
)
```

(from [ToDoUser/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

We then have our LiveData from this viewmodel be for this RosterViewState. And, we can go ahead and initialize it to start with authRequired = true, since the flow of this app pretty much guarantees that if this viewmodel is first being created, we are going to need the passphrase:

```
private val _states =  
    MutableLiveData<RosterViewState>(RosterViewState(authRequired = true))  
val states: LiveData<RosterViewState> = _states
```

(from [ToDoUser/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Our view — RosterListFragment — can then observe that stream of view-states and update the visible set of widgets to match:

```
motor.states.observe(viewLifecycleOwner) { state ->  
    adapter.submitList(state.items)  
  
    when {  
        state.authRequired -> {  
            binding.authGroup.isVisible = true  
            binding.empty.isVisible = false  
        }  
        state.items.isEmpty() -> {  
            binding.authGroup.isVisible = false  
            binding.empty.isVisible = true  
            binding.empty.setText(R.string.msg_empty)  
        }  
        else -> {  
            binding.authGroup.isVisible = false  
            binding.empty.isVisible = false  
        }  
    }  
}
```

(from [ToDoUser/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

So, when we launch the app and we get to this screen, initially, we will ask for the passphrase.

Applying the Passphrase

When the user clicks the button, we call an `open()` function on the viewmodel (`RosterMotor`), supplying the contents of the `EditText`:

```
binding.open.setOnClickListener {  
    motor.open(binding.passphrase.text.toString())  
}
```

(from [ToDoUser/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

With the previous versions of these samples, `RosterMotor` could just start working with the database right away, as the passphrase was either hard-coded or app-generated. Now, since we need a user-supplied passphrase, we need to delay opening the database until we have that passphrase, and that is what `open()` does:

```
fun open(passphrase: String) {  
    viewModelScope.launch {  
        if (repo.openDatabase(context, passphrase)) {  
            repo.items().collect { _states.value = RosterViewState(items = it) }  
        } else {  
            _states.value = RosterViewState(authRequired = true)  
        }  
    }  
}
```

(from [ToDoUser/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

`openDatabase()` on `ToDoRepository` will open or create the database using the supplied passphrase. It will return `true` if that succeeds or `false` otherwise. In the former case, we load our to-do list items and emit a fresh view-state with that data (and with `authRequired` set to `false`, the default). In the latter case, we ensure that our `LiveData` has `authRequired = false`. The UX will be that if the user mis-types their passphrase, nothing really happens — this is not ideal, but it keeps the example simple.

Creating and Opening the Database

In earlier examples, `ToDoRepository` received a `ToDoEntity.Store` as a constructor parameter via dependency inversion. Now, it receives a `ToDoDatabase.Factory` instead:

SQLCIPHER AND PASSPHRASES

```
class ToDoRepository(  
    private val dbFactory: ToDoDatabase.Factory,  
    private val appScope: CoroutineScope  
) {
```

(from [ToDoUser/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

That is because while previously Koin could open the database on its own, we now need to wait until the passphrase is available.

ToDoDatabase.Factory knows how to open a database, given a passphrase:

```
package com.commonsware.todo.repo  
  
import android.content.Context  
import androidx.room.Database  
import androidx.room.Room  
import androidx.room.RoomDatabase  
import androidx.room.TypeConverters  
import net.sqlcipher.database.SupportFactory  
  
private const val DB_NAME = "stuff.db"  
  
@Database(entities = [ToDoEntity::class], version = 1)  
@TypeConverters(TypeTransmogrifier::class)  
abstract class ToDoDatabase : RoomDatabase() {  
    abstract fun todoStore(): ToDoEntity.Store  
  
    class Factory {  
        fun newInstance(context: Context, passphrase: ByteArray) =  
            Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME)  
                .openHelperFactory(SupportFactory(passphrase))  
                .build()  
    }  
}
```

(from [ToDoUser/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#))

The koinModule in KoinApp now sets up a singleton instance of ToDoDatabase.Factory, to satisfy the ToDoRepository requirement:

```
single { ToDoDatabase.Factory() }  
single {  
    ToDoRepository(  
        get(),  
        get(named("appScope"))  
    )  
}
```

SQLCIPHER AND PASSPHRASES

(from [ToDoUser/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

ToDoRepository now has a db property that holds the ToDoDatabase... if it has been opened. Otherwise, it remains null:

```
private var db: ToDoDatabase? = null

fun isReady() = db != null
```

(from [ToDoUser/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

The openDatabase() function that our viewmodel called will use that ToDoDatabase.Factory to open the database and populate that db property:

```
suspend fun openDatabase(context: Context, passphrase: String): Boolean {
    try {
        db = dbFactory.newInstance(context, passphrase.toByteArray())
        db?.todoStore()?.count()
    } catch (t: Throwable) {
        try { db?.close() } catch (t2: Throwable) { }
        db = null
        Log.e("ToDoUser", "Exception opening database", t)
    }

    return isReady()
}
```

(from [ToDoUser/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

openDatabase() uses the ToDoDatabase.Factory to get the ToDoDatabase instance, supplying the passphrase. However that does not trigger the database to be *opened*. Normally, having the database be opened lazily is a fine Room feature. In this case, though, we have to worry about the possibility that the passphrase is wrong. So, we now have a count() function on our DAO, that just returns a count of the to-do item rows:

```
@Dao
interface Store {
    @Query("SELECT * FROM todos ORDER BY description")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String?): Flow<ToDoEntity?>

    @Query("SELECT COUNT(*) FROM todos")
    suspend fun count(): Long
}
```

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun save(vararg entities: ToDoEntity)

@Delete
suspend fun delete(vararg entities: ToDoEntity)
}
```

(from [ToDoUser/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

This is designed to be fairly cheap to execute while ensuring that the passphrase works. So, `openDatabase()` calls `count()`. If that throws an exception, then presumably the passphrase was mis-entered, so we `close()` the database and set `db` back to null to indicate that we have not successfully opened the database. `openDatabase()` then returns the Boolean indicating success or failure. All of this is done in a suspend function, so it can be performed on a background thread.

The remaining functions on `ToDoRepository` now use `db` to access the database and throw an exception if the database is not currently open:

```
fun items(): Flow<List<ToDoModel>> =
    db?.todoStore()?.let { store ->
        store.all().map { all -> all.map { it.toModel() } }
    } ?: throw IllegalStateException("database is not open")

fun find(id: String?): Flow<ToDoModel?> =
    db?.todoStore()?.let { store ->
        store.find(id).map { it?.toModel() }
    } ?: throw IllegalStateException("database is not open")

suspend fun save(model: ToDoModel) {
    withContext(appScope.coroutineContext) {
        db?.todoStore()?.save(ToDoEntity(model))
        ?: throw IllegalStateException("database is not open")
    }
}

suspend fun delete(model: ToDoModel) {
    withContext(appScope.coroutineContext) {
        db?.todoStore()?.delete(ToDoEntity(model))
        ?: throw IllegalStateException("database is not open")
    }
}
```

(from [ToDoUser/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

Pros and Cons

The good news is that the user knows the passphrase. Backups of the database can be made and restored, and the user should retain access to them. The database could even be transferred to some other device or platform, and the user can retain access.

The bad news is that the user knows the passphrase. This means that [the user might be convinced to give up the passphrase](#) and thereby lose control over their data.

Multi-Factor Authentication

Multi-factor authentication is all about combining multiple data sources to validate identity. In classic two-factor authentication, the phrase “something you have and something you know” is often used to describe the factors. “Something you have” might be a hardware token, or a code generated by an authenticator app. “Something you know” is a passphrase.

Similarly, we could combine the two passphrase techniques shown in this chapter, where the actual passphrase given to SQLCipher for Android combines:

- A user-supplied passphrase, *and*
- A generated passphrase that effectively is tied to the user’s ability to authenticate against their device

Or, we could combine a user-supplied passphrase and an externally-generated token, such as via [NFC-capable hardware tokens](#).

Or, we could combine all three: a user-supplied passphrase, a generated on-device passphrase, and a hardware token.

Or, we could come up with yet other sources of passphrase material and offer them as options.

In the end, SQLCipher for Android does not know or care how you get the passphrase or how it got assembled from individual pieces. That is up to you, as you try to strike the balance between security and usability.

The Risks of String

One flaw in `ToDoUser` is that the passphrase is being passed around as a `String`.

While `String` is convenient, `String` is immutable. We have no way to get rid of a `String`, other than to let go of it, hope that it gets garbage-collected quickly, then hope that something else allocates that same bit of memory and overwrites it.

Passphrases in memory are like nuclear waste: they served a role and now are just disasters waiting to happen. Fortunately, usually, disasters do not happen, but for some people, “usually” is insufficient.

So long as passphrases remain in memory, it is possible, through advanced techniques, for them to be extracted from memory. That almost always requires somebody to have physical access to the device, be able to obtain superuser privileges (a.k.a., “root the device”), and be able to use specialized tools to save a snapshot of the app’s heap to disk. Those are significant barriers, but ones that are manageable by attackers who are skilled, wealthy, or both.

Ideally, once we use the passphrase to gain access to the encrypted database, we would clear the passphrase itself out of memory. That is not possible with a `String`.

This is why `ToDoGen` uses a `ByteArray`. `SQLCipher` for Android, by default, will “zero out” the `ByteArray`, replacing all its bytes with zeros, once the passphrase has been used. This ensures that the passphrase can no longer be retrieved via examining the application’s heap.

Managing SQLCipher

Just as we need to think about managing our ordinary SQLite databases, we need to think about doing the same for SQLCipher for Android databases. In this chapter, we will briefly explore some of those common concerns.

Backup and Restore

[Earlier in the book](#), we looked at how to back up and restore a SQLite database. The same mechanisms would be used for import/export operations, to make a copy of the database available for users to take to another machine, for example.

With an encrypted database, things get more complicated. Roughly speaking, there are two scenarios to consider:

1. You want to back up the database and keep it encrypted. This would be good for cases where the passphrase is user-supplied, or for cases where you are going to back up the passphrase as well (by some secure means). Similarly, you might be trying to restore the database that is already encrypted.
2. You want to export the database in a decrypted form, one that can be used by ordinary SQLite clients, not necessarily ones that support the SQLCipher format. Similarly, you might be trying to import a decrypted database and store that data in your encrypted database.

The first scenario works pretty much the same as with regular SQLite databases. If you are copying using filesystem APIs, the data in the database remains in its current form. And, with SQLCipher for Android, the “current form” is encrypted. So, if you copy the database file for an encrypted database, you wind up with a copy of that encrypted database.

The import/export to unencrypted (“plaintext”) databases gets more involved, as we will see in [the ImportExportCrypt module](#) of [the book’s primary sample project](#). That sample builds upon the ImportExport sample from [the chapter on backing up a database](#). However, this time, the database is encrypted, using a hard-coded passphrase for simplicity. And, our UI now has five buttons, including options for both plain and encrypted import and export:

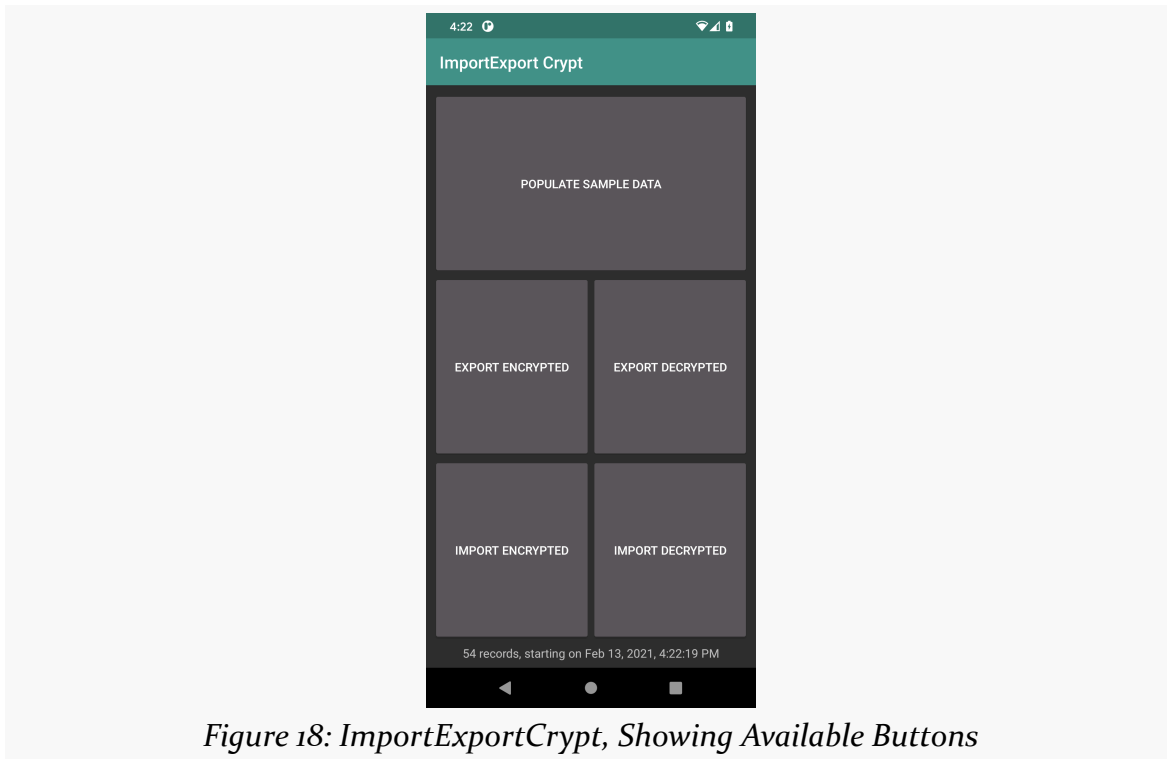


Figure 18: ImportExportCrypt, Showing Available Buttons

Exporting a Plaintext Database

Our app has a SQLCipher for Android database, encrypted by our hard-coded passphrase. We want to export a plaintext copy of that database: same schema, same data, just without the encryption.

SQLCipher for Android has a recipe for doing this. Since it involves a fair bit of manual database manipulation, rather than using Room or the `SupportSQLiteDatabase` API, we will go “old school” and work with the SQLCipher for Android version of `SQLiteDatabase` directly. In the end, we wind up with a `decryptTo()` function on a `SQLCipherUtils` object that takes:

- A path to our encrypted database file

- A path to the destination plaintext database file
- The passphrase for our encrypted database
- A Context, because this is Android, and you cannot get out of bed in the morning without a Context

After calling this blocking function, our plaintext database should reside at the requested path.

Examining the Utility Function

The function looks a bit nasty:

```
fun decryptTo(
    ctxt: Context,
    originalFile: File,
    targetFile: File,
    passphrase: ByteArray?
) {
    SQLiteDatabase.loadLibs(ctxt)

    if (originalFile.exists()) {
        val originalDb = SQLiteDatabase.openDatabase(
            originalFile.absolutePath,
            passphrase,
            null,
            SQLiteDatabase.OPEN_READWRITE,
            null,
            null
        )

        SQLiteDatabase.openOrCreateDatabase(
            targetFile.absolutePath,
            "",
            null
        ).close() // create an empty database

        //language=text
        val st =
            originalDb.compileStatement("ATTACH DATABASE ? AS plaintext KEY ''")

        st.bindString(1, targetFile.absolutePath)
        st.execute()
        originalDb.rawQuerySQL("SELECT sqlcipher_export('plaintext')")
        originalDb.rawQuerySQL("DETACH DATABASE plaintext")

        val version = originalDb.version
    }
```

```
st.close()
originalDb.close()

val db = SQLiteDatabase.openOrCreateDatabase(
    targetFile.absolutePath,
    "",
    null
)

db.version = version
db.close()
} else {
    throw FileNotFoundException(originalFile.absolutePath + " not found")
}
}
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

If you wish to take it on faith that the function works, feel free to skip ahead to the next section.

But, to explain what this function does, let's take it step by step.

```
SQLiteDatabase.loadLibs(ctxt)
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

SQLCipher for Android contains a complete copy of SQLite with the SQLCipher for Android extensions compiled in. This is native code, compiled via the NDK. When we use the SQLCipher for Android dependency, we get that compiled code in the form of .so files in the AAR. This statement loads the SQLCipher for Android library.

Then, after checking to see if the encrypted database file exists, we open it using the SQLCipher for Android edition of SQLiteDatabase:

```
val originalDb = SQLiteDatabase.openDatabase(
    originalFile.absolutePath,
    passphrase,
    null,
    SQLiteDatabase.OPEN_READWRITE,
    null,
    null
)
```

MANAGING SQLCIPHER

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

The primary difference from what you would do with the framework SQLiteDatabase is that you supply the passphrase to use to open the database.

We then need to set up the target plaintext database. Once again we use the SQLCipher for Android edition of SQLiteDatabase, but this time we use "" as the passphrase, which tells SQLCipher to leave this database decrypted:

```
SQLiteDatabase.openOrCreateDatabase(  
    targetFile.absolutePath,  
    "",  
    null  
).close() // create an empty database
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

You will notice that we then immediately close this database. We need the database to exist, but we do not need (or want) it to be open — we are going to use it in a slightly different fashion... via ATTACH DATABASE:

```
//language=text  
val st =  
    originalDb.compileStatement("ATTACH DATABASE ? AS plaintext KEY ''")  
  
st.bindString(1, targetFile.absolutePath)  
st.execute()
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

ATTACH DATABASE is [part of SQLite's SQL syntax](#). It allows you to open two databases at once. The database that you opened using SQLiteDatabase is identified normally; the database that you attach is identified by the name you give it via the AS keyword. So, in this case, we are attaching the empty plaintext database as plaintext. SQLCipher extends ATTACH DATABASE to take a KEY keyword that provides the passphrase to use — in this case, we are passing the empty string, to indicate that this database is not encrypted. The ? in the ATTACH DATABASE statement is the fully-qualified path to the database file, which we supply as a query parameter using bindString(). The net effect of executing this SQL statement is that we now have both databases open at once.

SQLCipher comes with a sqlcipher_export() function that we can invoke that copies the contents of the database from the original to the attached one:

MANAGING SQLCIPHER

```
originalDb.rawQuerySQL("SELECT sqlcipher_export('plaintext')")
originalDb.rawQuerySQL("DETACH DATABASE plaintext")
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

The effect of these two statements is to copy almost all of the encrypted data from the original database to the plaintext database, then to detach that plaintext database (thereby closing it).

One piece of data that `sqlcipher_export()` misses is the schema version. So, our final step is to get the schema version from the encrypted database and apply it to the plaintext database:

```
val version = originalDb.version

st.close()
originalDb.close()

val db = SQLiteDatabase.openOrCreateDatabase(
    targetFile.absolutePath,
    "",
    null
)

db.version = version
db.close()
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

The result is that all of our data resides in our new plaintext database, and both database files are closed when we are done.

This code is far from perfect. It does not have much in the way of recovery from problems, for example. For the purposes of this book example, it works and is reasonable. If you wish to do this sort of work in a production app, though, you should look to improve upon this function.

Using the Utility Function

One thing that `SQLCipherUtils.decryptTo()` requires is that the original and target both be files. That is because SQLite and SQLCipher for Android both need files. However, `RandomDatabase.copyTo()` function uses an `OutputStream` as a target, as the user is choosing the destination of the export via `ActivityResultContracts.CreateDocument`, so we do not have a file.

So, `RandomDatabase.decryptTo()` has `SQLCipherUtils.decryptTo()` decrypt the database to a temporary file, which we then copy to the desired `OutputStream`:

```
fun decryptTo(context: Context, stream: OutputStream) {
    val temp = File(context.cacheDir, "export.db")

    temp.delete()

    SQLCipherUtils.decryptTo(
        context,
        context.getDatabasePath(DB_NAME),
        temp,
        PASSPHRASE.toByteArray()
    )

    temp.inputStream().copyTo(stream)
    temp.delete()
}
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/RandomDatabase.kt](https://commonsware.com/licenses/room/importexport/RandomDatabase.kt))

That way, we can use `RandomDatabase.decryptTo()` the same way that we used `RandomDatabase.copyTo()`. The activity, viewmodel, and repository all follow the same pattern as in the original `ImportExport` app, just routing the export-plain case through new functions that, in the end, trigger a call to `RandomDatabase.decryptTo()`.

The result, if you run the app and populate the database, then opt to export the plaintext database, is that your exported copy can be opened in a SQLite client without any passphrase.

Importing a Plaintext Database

The other direction — importing a plaintext database — works much the same way. However, this time, we also add a new wrinkle: detecting whether the to-be-imported database really is a plaintext database or not.

Detecting Plaintext

Sometimes, you will be in a situation where you do not know if the database is already encrypted or not. The way to determine if the database is encrypted is to try opening it with the empty string as a passphrase. As we saw earlier, that is how you tell `SQLCipher` for Android to open an unencrypted database. If we can successfully

open the database with an empty passphrase, we know that the database is not encrypted. If we get some sort of problem, we know that either:

- The database is encrypted, or
- The database is unencrypted but the file has been corrupted somehow

For security reasons, SQLCipher for Android does not distinguish between those two cases. That is so attackers cannot learn from a failed open attempt (perhaps with a candidate passphrase, like 12345) whether the database is encrypted or not.

The SQLCipherUtils object has a `getDatabaseState()` function that applies this technique, returning a `State` object for three possibilities:

- The database is unencrypted (UNENCRYPTED)
- The database is encrypted (ENCRYPTED)
- The database is missing (DOES_NOT_EXIST)

Just remember that ENCRYPTED is short for ENCRYPTED_OR_POSSIBLY_CORRUPT.

To detect those cases, `getDatabaseState()` sees if your requested file exists and, if it does, tries opening it with the empty passphrase:

```
/**
 * The detected state of the database, based on whether we can open it
 * without a passphrase.
 */
enum class State {
    DOES_NOT_EXIST, UNENCRYPTED, ENCRYPTED
}

fun getDatabaseState(ctxt: Context, dbPath: File): State {
    SQLiteDatabase.loadLibs(ctxt)

    if (dbPath.exists()) {
        var db: SQLiteDatabase? = null

        return try {
            db = SQLiteDatabase.openDatabase(
                dbPath.absolutePath,
                "",
                null,
                SQLiteDatabase.OPEN_READONLY
            )
            db.version
        } catch (e: Exception) {
            State.DOES_NOT_EXIST
        }
    } else {
        State.DOES_NOT_EXIST
    }
}
```

```
        State.UNENCRYPTED
    } catch (e: Exception) {
        State.ENCRYPTED
    } finally {
        db?.close()
    }
}

return State.DOES_NOT_EXIST
}
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

Examining the Utility Function

Given a plaintext database, `SQLCipherUtils.encryptTo()` will encrypt it to a designated database file:

```
fun encryptTo(
    ctxt: Context,
    originalFile: File,
    targetFile: File,
    passphrase: ByteArray?
) {
    SQLiteDatabase.loadLibs(ctxt)

    if (originalFile.exists()) {
        val originalDb = SQLiteDatabase.openDatabase(
            originalFile.absolutePath,
            "",
            null,
            SQLiteDatabase.OPEN_READWRITE
        )
        val version = originalDb.version

        originalDb.close()

        val db = SQLiteDatabase.openOrCreateDatabase(
            targetFile.absolutePath,
            passphrase,
            null
        )

        //language=text
        val st = db.compileStatement("ATTACH DATABASE ? AS plaintext KEY ''")

        st.bindString(1, originalFile.absolutePath)
```

MANAGING SQLCIPHER

```
st.execute()
db.rawQuerySQL("SELECT sqlcipher_export('main', 'plaintext')")
db.rawQuerySQL("DETACH DATABASE plaintext")
db.version = version
st.close()
db.close()
} else {
    throw FileNotFoundException(originalFile.absolutePath + " not found")
}
}
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

As with `decryptTo()`, `encryptTo()` is a bit complex, but it has the same basic structure.

We start by loading SQLCipher for Android's native libraries using `SQLiteDatabase.loadLibs(ctxt)`. Then, assuming that the plaintext database exists, we open it using the empty string as a passphrase (indicating it is plaintext), get the database version, and close the database back up:

```
val originalDb = SQLiteDatabase.openDatabase(
    originalFile.absolutePath,
    "",
    null,
    SQLiteDatabase.OPEN_READWRITE
)
val version = originalDb.version

originalDb.close()
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

We then open or create the encrypted database, using the supplied passphrase:

```
val db = SQLiteDatabase.openOrCreateDatabase(
    targetFile.absolutePath,
    passphrase,
    null
)
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

Next, we attach the plaintext database using `ATTACH DATABASE`:

MANAGING SQLCIPHER

```
//language=text
val st = db.compileStatement("ATTACH DATABASE ? AS plaintext KEY ''")

st.bindString(1, originalFile.absolutePath)
st.execute()
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

We then use a two-parameter form of `sqlcipher_export()`, saying that we want to “export” the plaintext database into the encrypted one:

```
db.rawQuerySQL("SELECT sqlcipher_export('main', 'plaintext')")
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

Finally, we detach the plaintext database, set the version in the encrypted database, and close everything up:

```
db.rawQuerySQL("DETACH DATABASE plaintext")
db.version = version
st.close()
db.close()
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

Using the Utility Function

Just as `RandomDatabase` has its `decryptTo()` that wraps the `SQLCipherUtils` equivalent, `RandomDatabase` also has `encryptFrom()` that wraps `SQLCipherUtils.encryptFrom()`:

```
fun encryptFrom(context: Context, stream: InputStream) {
    val temp = File(context.cacheDir, "import.db")

    temp.delete()

    stream.copyTo(temp.outputStream())

    try {
        when (SQLCipherUtils.getDatabaseState(context, temp)) {
            SQLCipherUtils.State.UNENCRYPTED -> SQLCipherUtils.encryptTo(
                context,
                temp,
                context.getDatabasePath(DB_NAME),
                PASSPHRASE.toByteArray()
            )
        }
    }
}
```

```
SQLCipherUtils.State.DOES_NOT_EXIST ->
    throw IllegalStateException("Could not find $temp??")
SQLCipherUtils.State.ENCRYPTED ->
    throw IllegalStateException("Original database appears encrypted!")
}
} finally {
    temp.delete()
}
}
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/RandomDatabase.kt](#))

Both `getDatabaseState()` and `encryptTo()` on `SQLCipherUtils` work with a file, and we are starting with an `InputStream`. So, we start by copying the contents of that stream to a temporary database file.

We then call `getDatabaseState()`. In the expected outcome, we get `UNENCRYPTED` for the state and can then call `encryptTo()` to encrypt the database and put it in our desired location. If we find any other state, we throw an exception. All of that is wrapped in `try / finally`, so we can delete the temporary unencrypted database copy.

About That language Comment

In both `encryptTo()` and `decryptFrom()` on `SQLCipherUtils`, we have an odd comment:

```
//language=text
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

This appears before each of our `compileStatement()` calls:

```
//language=text
val st =
    originalDb.compileStatement("ATTACH DATABASE ? AS plaintext KEY '')
```

(from [ImportExportCrypt/src/main/java/com/commonsware/room/importexport/SQLCipherUtils.kt](#))

This stems from a feature of Android Studio called “language injection”. Basically, there are ways for the IDE to interpret a string as being associated with a programming language and to provide syntax validation *of the string* following the rules of that language.

In this case, by default, Studio thinks that the parameter to `compileStatement()` is a

SQL string, and so it wants to validate that SQL.

This sounds wonderful!

However, Studio's SQL syntax rules are based on SQLite, not SQLCipher for Android. As a result, Studio does not like our KEY clause:

```
val st : SQLiteStatement! =  
    originalDb.compileStatement( sql: "ATTACH DATABASE ? AS plaintext KEY ''")
```

Figure 19: Android Studio “Language Injection” SQL Validation

The language=text comment is [a hack to block that validation warning](#).

Migrating to Encryption

Normal Room migrations work with SQLCipher for Android, for adjusting your database schema to account for changes in your entities, DAOs, and so forth.

However, with SQLCipher for Android, you may have a special “migration” to consider: migrating from having an ordinary database to having an encrypted one. That would occur if you started with ordinary SQLite and only decided to add encryption later on.

So, suppose you initially shipped your app with ordinary SQLite as version 1.0.0. Later, in version 2.0.0 of your app, you shipped support for encrypted databases. Now your RoomDatabase will need to handle two cases:

1. Version 2.0.0 of your app is being installed by a new user. In that case, you have no existing database, and you can start with the encrypted database from the outset.
2. An existing 1.x.y user upgrades to 2.0.0. They already have an unencrypted database. Presumably, they would like to keep that data, and so you need to encrypt that database.

The SQLCipherUtils code that we saw earlier in this chapter can handle this scenario as well, as we can see in [the cryptMigrate pair of modules](#) of [the book's primary sample project](#).

That directory contains two modules. CMBefore is the same to-do app that we saw in

earlier SQLCipher for Android samples, just without any encryption. Theoretically, this represents version 1.0.0 of your app. CMAfter is mostly the same as the ToDoCrypt sample from previous chapters, where we use SQLCipher for Android with a hardcoded passphrase (to simplify the sample). However, this time, ToDoDatabase has a substantially more complex version of newInstance():

```
package com.commonware.todo.repo

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters
import net.sqlcipher.database.SupportFactory
import java.io.IOException

private const val DB_NAME = "stuff.db"
private const val PASSPHRASE = "sekr1t"

@Database(entities = [ToDoEntity::class], version = 1)
@TypeConverters(TypeTransmogrifier::class)
abstract class ToDoDatabase : RoomDatabase() {
    abstract fun todoStore(): ToDoEntity.Store

    companion object {
        fun newInstance(context: Context): ToDoDatabase {
            val dbFile = context.getDatabasePath(DB_NAME)
            val passphrase = PASSPHRASE.toByteArray()
            val state = SQLCipherUtils.getDatabaseState(context, dbFile)

            if (state == SQLCipherUtils.State.UNENCRYPTED) {
                val dbTemp = context.getDatabasePath("_temp.db")

                dbTemp.delete()

                SQLCipherUtils.encryptTo(context, dbFile, dbTemp, passphrase)

                val dbBackup = context.getDatabasePath("_backup.db")

                if (dbFile.renameTo(dbBackup)) {
                    if (dbTemp.renameTo(dbFile)) {
                        dbBackup.delete()
                    } else {
                        dbBackup.renameTo(dbFile)
                        throw IOException("Could not rename $dbTemp to $dbFile")
                    }
                } else {

```

MANAGING SQLCIPHER

```
        dbTemp.delete()
        throw IOException("Could not rename $dbFile to $dbBackup")
    }
}

return Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME)
    .openHelperFactory(SupportFactory(passphrase))
    .build()
}
}
```

(from [cryptMigrate/CMAfter/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](https://github.com/commonsware/todo-repo/blob/master/CMAfter/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt))

We start by checking the database state using `getDatabaseState()`. If the state is `ENCRYPTED`, we are set — this implies that the user has already run this version of the app before and we already have our encrypted database. If the state is `DOES_NOT_EXIST`, we also do not need to do anything special, as Room will lazy-create our encrypted database for us. In either of those cases, we proceed directly to using `Room.databaseBuilder()` with our `SupportFactory` to create and/or open the database.

The scenario where we need to do extra work is if our state is `UNENCRYPTED`. That means the user was using `CMBefore` and has an existing unencrypted database. We want to keep the data, but encrypt it. So, we:

- Use `SQLCipherUtils.encryptTo()` to encrypt the database to a new database file
- Rename the unencrypted database to a temporary name (`dbBackup`)
- Rename the newly-encrypted database to our desired name (`dbFile`)
- Delete the unencrypted database

If something goes wrong with one of our file I/O operations, we try to switch back to the unencrypted database and fail with an exception.

If you run `CMBefore`, fill in some to-do items, and then run `CMAfter`, you will find that your to-do items remain intact, but that the resulting database is encrypted.

Advanced Scenarios

Paged Room Queries

Sometimes, we simply have too much data.

It is very easy to write a DAO that returns all rows from a particular table. Whether doing that is sensible or not will depend on:

- The number of rows in that table
- The number, types, and contents of columns in that table
- The impacts of any JOINS or other constructs that might expand the result set

If we know that we might have a ridiculous amount of data, we can use Room's support for [the Jetpack Paging library](#). This teaches Room to retrieve rows a “page” at a time from the underlying table(s), instead of the full result set all at once. This can greatly reduce the amount of memory that is consumed at once, if we can organize our UI to only need a page's worth of data at once. The downside is that the only easy way to consume this paged data is via a RecyclerView — anything else is terribly complicated.

In this chapter, we will explore a bit about the Paging library in general and show how Room and Paging can populate a RecyclerView as the user scrolls.

The Problem: Too Much Data

One of the little-known issues with Android's SQLite API is how the Cursor works. We tend to just use that Cursor and ignore exactly how it is getting its data. The behavior of our database Cursor is normal for smaller data sets but possibly problematic for really large ones.

Cursor is an interface. The real Java class that we get back from SQLite is a `SQLiteCursor`. The Cursor API, and `SQLiteCursor` in particular, was developed well before Android 1.0 was released, and therefore has a fair share of “features” that seemed like good ideas at the time but did not hold up well as the years progressed. The one that everybody encounters is the fact that when you get a Cursor back from methods like `query()` or `rawQuery()` on a `SQLiteDatabase`, the query has not actually been done yet. Instead, it is lazy-executed when you ask the Cursor for something where the data is needed, such as `getCount()`. This is a pain, as we want to do the database I/O on a background thread, so we have to specifically do something while on that background thread (e.g., call `getCount()`) to ensure that the query really does get executed when we expect it to.

Another quirk with Cursor is that when the query is executed, it really populates a `CursorWindow`. For small queries, this will represent the entire result set. For larger queries, it is a portion of that result set. As we move through the Cursor, `SQLiteCursor` will load more relevant rows into the `CursorWindow`, around the new position. This exacerbates the threading problem, as we might wind up doing disk I/O at any point while working with the Cursor, if the window’s contents need to be adjusted.

Ideally, your queries are small, within the `CursorWindow` limits. And for apps where the data comes from the user, usually you can keep your queries small. Users are only going to enter in so much data on a small screen. Even if the user records some form of multimedia — such as taking a picture with the camera — large queries can be avoided by not storing the media in the database itself, but rather storing it in plain files referenced by the database.

However, in cases where the data comes from some server, sticking with small queries can get tricky.

Addressing the UX

Beyond the threading issues, there is another challenge with showing large result sets in a single UI (e.g., in a `RecyclerView`): it is a pain for users to navigate. Nobody is going to want to scroll through 10,000 rows in a vertically-scrolling list — their finger will develop a blister first.

If you anticipate having a large amount of data, your primary concern is to get the UX right. Focus on searching, filtering, and other means for the user to easily scope the required data to some subset of relevance. Do not have the primary UX be a

“scroll through the world” sort of experience, even if that is an available option for users who are gluttons for punishment or have steel-tipped fingers (or, perhaps, a stylus).

However, even with user-supplied constraints, you still might wind up with more data than can fit in a `CursorWindow`. And we have no direct control over that `CursorWindow` behavior, as it is hidden behind a few layers of abstraction.

Enter the Paging Library

The Paging library exists to provide greater developer control over exactly what gets loaded from a backing data store and when, handling things like:

- Performing smaller queries, to stay inside a `CursorWindow`’s bounds, so we can control the threads used for data loads
- Supporting multiple traversal options through a data set: not only classic position-based systems, but ones where you might be navigating a tree and need to retrieve related child objects as part of traversal
- Offering reactive approaches, based on `LiveData`, so we can ensure that our UI remains responsive.

The Paging library has evolved over the years. The current edition, known as Paging v3, has a powerful Kotlin-centric API, and Room has basic support for returning Paging v3-based responses from queries.

There are a number of classes involved in the Paging library, but for basic scenarios, there are a few of significance:

- A `PagingSource` represents a source of paged data. Room can be set up to return a `PagingSource` from a `@Query`-annotated `@Dao` function, for example.
- A `PagingData` represents a chunk of data retrieved from the underlying data source via the `PagingSource`
- A `Pager` represents a specific operation for getting paged data out of a `PagingSource`. We can teach a `Pager` how many items to retrieve in a “page”, and the `Pager` gives us a `Flow`, `LiveData`, or similar reactive API for getting `PagingData` objects as needed.
- A `PagingDataAdapter` is a `RecyclerView.Adapter` that knows how to work with a `PagingData`. As the user scrolls the `RecyclerView`, the `PagingDataAdapter` can signal that we need more data, triggering the `Pager` to fetch more data from the `PagingSource` and emit a fresh `PagingData` for

the `PagingDataAdapter` to use.

That still leaves a lot of moving parts, which we will examine in greater detail in this chapter.

Paging and Room

The [PagedFTS module](#) of [the book's primary sample project](#) demonstrates the use of the Paging library with Room.

This is our third generation of a full-text searching sample, originating with FTS and continuing with `PackagedFTS`. In those samples, we had an app that displayed the text of H. G. Wells' "The Time Machine", with individual paragraphs as rows in a `RecyclerView`. However, we loaded the entire book (or entire set of search results) at once. "The Time Machine" is short, but there are [much longer books](#), where loading the entire book into memory would be impractical.

So, `PagedFTS` replaces the load-everything logic of FTS with load-pages logic, courtesy of the Paging library.

The Dependency

To use those classes, we need another dependency, one for the Paging library:

```
implementation "androidx.paging:paging-runtime-ktx:3.0.1"
```

(from [PagedFTS/build.gradle](#))

`androidx.paging:paging-runtime-ktx` gives us the code that we need to tell Room to produce paged query results and to populate a `RecyclerView` with the results. As with other dependencies, the `-ktx` suffix indicates that there are Kotlin-specific functions, such as a `toLiveData()` extension function that we will examine shortly.

The DAO

The revised `BookStore` is a bit different than what we had before:

```
package com.commonware.room.fts

import androidx.paging.PagingSource
import androidx.room.Dao
import androidx.room.Insert
```

PAGED ROOM QUERIES

```
import androidx.room.Query

@Dao
abstract class BookStore {
    @Insert
    abstract suspend fun insert(paragraphs: List<ParagraphEntity>)

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

    @Query("SELECT prose FROM paragraphs ORDER BY sequence")
    abstract fun all(): PagingSource<Int, String>

    @Query("SELECT snippet(paragraphsFts) FROM paragraphs JOIN paragraphsFts "+
        "ON paragraphs.id == paragraphsFts.rowid WHERE paragraphsFts.prose "+
        "MATCH :search ORDER BY sequence")
    abstract fun filtered(search: String): PagingSource<Int, String>
}
```

(from [PagedFTS/src/main/java/com/commonsware/room/fts/BookStore.kt](#))

The `insert()` function is unchanged. There is a new `count()` function that returns the number of paragraphs in our database. But the more interesting changes affect `all()` and `filtered()`:

- They no longer are suspend functions
- They no longer return a `List` of paragraphs, but instead return a `PagingSource`

`PagingSource` takes two generic types. The second is the type of data that we are paging through. In this case, both queries return simple `String` objects, so our second data type to `PagingSource` is `String`. We could use `ParagraphEntity` or anything else Room knows how to handle.

The first data type in the `PagingSource` declaration indicates to the Paging library how we are identifying the contents of pages. Room will do this based on position within a result set, so the first page might be rows 0-49, the second page might be rows 50-99, and so on. With Room, positions are indicated by an `Int`, so the first type that we provide to `PagingSource` is an `Int`. The Paging library supports other strategies, mostly designed around developers creating custom data sources (e.g., wrapped around a REST-style Web service), but Room uses positions.

We skip the `suspend` keyword because `PagingSource` does not perform any I/O immediately. Instead, the actual I/O is delayed until something starts using the

factory. This is much like how LiveData, Single, and Flow work with [our classic reactive return types](#).

The ViewModels

BookViewModel and SearchViewModel need to expose the book contents to their respective UI layers. In the original project, this was via a LiveData. And, the good news is that we can get a LiveData from a PagingSource — though while the code is terse, it is a bit complicated:

```
package com.commonware.room.fts

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.paging.Pager
import androidx.paging.PagingConfig
import androidx.paging.cachedIn
import androidx.paging.liveData

class BookViewModel(repo: BookRepository) : ViewModel() {
    val paragraphs =
        Pager(PagingConfig(pageSize = 15)) { repo.all() }
            .liveData
            .cachedIn(viewModelScope)
}
```

(from [PagedFTS/src/main/java/com/commonware/room/fts/BookViewModel.kt](#))

```
package com.commonware.room.fts

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.paging.Pager
import androidx.paging.PagingConfig
import androidx.paging.cachedIn
import androidx.paging.liveData

class SearchViewModel(search: String, repo: BookRepository) : ViewModel() {
    val paragraphs =
        Pager(PagingConfig(pageSize = 15)) { repo.filtered(search) }
            .liveData
            .cachedIn(viewModelScope)
}
```

(from [PagedFTS/src/main/java/com/commonware/room/fts/SearchViewModel.kt](#))

As noted earlier, a Pager handles getting pages of data from our PagingSource. We can configure the Pager via the PagingConfig parameter — here, we say that we want to load 15 paragraphs at a time, via pageSize. We also provide the Pager with the PagingSource by way of a lambda expression. When we start trying to get data from the Pager, it will invoke that lambda expression, get the PagingSource, and start asking it for data.

That sets up the Pager itself. The LiveData extension property on Pager gives us a LiveData wrapper around the Pager... or, more accurately, a LiveData wrapper around a Flow that is wrapped around the Pager. This LiveData will be for a PagingData of our PagingSource data type — in this case, String. The Pager, when it is requested to load different pages, will emit a fresh PagingData holding the results.

Because this LiveData is backed by a Flow, we need to teach the LiveData about the CoroutineScope to use, so that the Flow collector that it uses handles lifecycles properly within the coroutine system. This is handled by calling cachedIn() and providing a suitable scope — in this case, viewModelScope. If you forget to do this, while the code will compile, [you will have a bad time when you try to run it](#).

The PagingDataAdapter

We are showing our book contents in a ParagraphAdapter. In the original project, this was a simple RecyclerView.Adapter. If you are using the Paging library, though, you will want to use PagingDataAdapter. You can hand a PagingDataAdapter a PagingData of data (e.g., a PagingData of paragraphs), and PagingDataAdapter knows how to use the PagingData API to handle loading additional pages as the user scrolls.

The good news is that with PagingDataAdapter, you do not need to bother with implementing getCount(), as PagingDataAdapter knows its PagingData and can get the overall size from it. The bad news is that you will need to provide a DiffUtil.ItemCallback to compare objects in the list. If you have used the RecyclerView version of ListAdapter, it uses the same DiffUtil.ItemCallback abstract class that PagedListAdapter does.

So, our revised ParagraphAdapter looks like:

```
package com.commonware.room.fts

import android.view.LayoutInflater
```

PAGED ROOM QUERIES

```
import android.view.ViewGroup
import androidx.paging.PagingDataAdapter
import androidx.recyclerview.widget.DiffUtil

class ParagraphAdapter(private val inflater: LayoutInflater) :
    PagingDataAdapter<String, RowHolder>(STRING_DIFFER) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        RowHolder(inflater.inflate(R.layout.row, parent, false))

    override fun onBindViewHolder(holder: RowHolder, position: Int) {
        holder.bind(getItem(position).orEmpty())
    }
}

private val STRING_DIFFER = object : DiffUtil.ItemCallback<String>() {
    override fun areItemsTheSame(oldItem: String, newItem: String) =
        oldItem === newItem

    override fun areContentsTheSame(oldItem: String, newItem: String) =
        oldItem == newItem
}
```

(from [PagedFTS/src/main/java/com/commonsware/room/fts/ParagraphAdapter.kt](#))

Here, `STRING_DIFFER` is a `DiffUtil.ItemCallback` that can handle comparing simple strings, using content equality for `areContentsTheSame()` and object equality (`===` in Kotlin) for `areItemsTheSame()`.

There is also one other subtle difference: the data that we bind into the `RowHolder`. In the original app, this was a `String`. In this app, though, it is a `String?`. The Paging library uses `null` as the default placeholder data, if we try accessing parts of the list that have not yet been loaded. We call `getItem()` from `PagingDataAdapter` to return the data for a given position, and it will return a nullable type. We have to be able to cope with a `null` value. Here, we just use `orEmpty()` to convert it into an empty string, but a more sophisticated app might use a different mechanism to indicate a `RecyclerView` item that has been loading. Once the data for that position is available, `onBindViewHolder()` will be called again, so you can repopulate the UI for that item with the now-available data.

The Fragments

The only significant difference in the fragments is in setting up the `ParagraphAdapter`. In the original project, we would pass in the `List<String>` to the `ParagraphAdapter` constructor, so we had to wait to create the `ParagraphAdapter`

PAGED ROOM QUERIES

until we were observing the LiveData that would provide our List. Now, we call a `submitData()` method on `ParagraphAdapter`, supplied by `PagingDataAdapter`, so we can set up the adapter ahead of time:

```
super.onViewCreated(view, savedInstanceState)

val rv = view as RecyclerView
val adapter = ParagraphAdapter(layoutInflater)

rv.adapter = adapter

vm.paragraphs.observe(viewLifecycleOwner) {
    adapter.submitData(viewLifecycleOwner.lifecycle, it)
}
}
```

(from [PagedFTS/src/main/java/com/commonsware/room/fts/BookFragment.kt](#))

```
super.onViewCreated(view, savedInstanceState)

val rv = view as RecyclerView
val adapter = ParagraphAdapter(layoutInflater)

rv.adapter = adapter

vm.paragraphs.observe(viewLifecycleOwner) {
    adapter.submitData(viewLifecycleOwner.lifecycle, it)
}
}
}
```

(from [PagedFTS/src/main/java/com/commonsware/room/fts/SearchFragment.kt](#))

From the user's standpoint, there is no real difference in behavior. The user can scroll through the list and read the book or the search results. In principle, on a slow device, the user might scroll fast enough that the data is not ready, and so a blank spot would appear at the bottom of the scrolled area (rows with the empty string) until the data got loaded. In the case of this app, the database I/O is small and fairly quick, so the user is unlikely to encounter any such visual hiccup with rapid scrolling. Yet, our app will be more efficient in its use of memory, by not necessarily loading the entire book at once.

Room Across Processes

Most Android apps run in a single process. However, Android does give developers the option to have their apps be split across multiple processes.

However, this can cause problems with stuff held in memory, as multiple processes will not share that memory. In the case of Room, one area where the problem crops up is with “invalidation tracking”: the piece of Room that knows to update your reactive query results (Flow, LiveData, Observable, etc.) when the data changes. By default, that does not work across process boundaries, but you can enable it via `enableMultiInstanceInvalidation()` on your `RoomDatabase.Builder`.

In this chapter, we will explore all of this in greater detail and demonstrate how it works.

Room and Invalidation Tracking

When you modify your Room database, active observers of reactive responses from `@Query`-annotated functions get fresh results delivered automatically.

Under the covers, that is powered by `InvalidationTracker` and related code. Room tracks:

- Which tables and views are referenced by active queries, and
- Which tables are affected by database modifications

When you modify the database, Room looks up the active queries that are tied to any tables that you affected. For those, Room re-executes the queries and emits new results via the Flow, LiveData, Observable, or whatever.

This is all done for you, almost by magic. But, in reality, it just sophisticated code from a sophisticated library.

Invalidation Tracking and Processes

InvalidationTracker is an ordinary Java class. So are RoomDatabase and the code-generated bits that we get from the Room compiler. They are part of the memory of whatever process that you used to create an instance of your RoomDatabase subclass.

If you have two processes, each working with the same RoomDatabase subclass, each process will have its own independent instance of that subclass, and each of those will be associated with its own InvalidationTracker. Each of those instances will know nothing about the other.

If you modify the database in one process, the InvalidationTracker of that process can notify observers in that process about updated results to their queries. However, by default, the InvalidationTracker of the other process will not find out about the database modifications and will not be able to update its own observers with fresh data.

Introducing enableMultiInstanceInvalidation()

Room now has an enableMultiInstanceInvalidation() function that you can call on RoomDatabase.Builder when you are setting up the database. This tells Room that you want to use it across processes. Room will then set up a MultiInstanceInvalidationService in your primary (default) process.

RoomDatabase objects in other processes will connect to that service, and Room will use IPC to allow database modification information to flow between the processes. The net effect is that each InvalidationTracker finds out about modifications happening in any of the app's processes.

[The CrossProcess module](#) of [the book's primary sample project](#) has a database that uses enableMultiInstanceInvalidation():

```
package com.commonware.room.process

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters
```

```
private const val DB_NAME = "random.db"

@Database(entities = [RandomEntity::class], version = 1)
@TypeConverters(TypeTransmogriifier::class)
abstract class RandomDatabase : RoomDatabase() {
    abstract fun randomStore(): RandomStore

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, RandomDatabase::class.java, DB_NAME)
                .enableMultiInstanceInvalidation()
                .build()
    }
}
```

(from [CrossProcess/src/main/java/com/commonsware/room/process/RandomDatabase.kt](#))

For that to be useful, though, we need more than one process, and we need for each process to be working with the same underlying SQLite database via the same Room-generated classes.

In One Process, an Activity

In the main application process, MainActivity has a really big “Populate Sample Data” button that, when clicked, calls a `populate()` function on `MainViewModel`. That in turn calls `populate()` on the `RandomRepository`. That generates a random number of `RandomEntity` instances and inserts them:

```
package com.commonsware.room.process

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.asCoroutineDispatcher
import kotlinx.coroutines.withContext
import java.util.concurrent.Executors
import kotlin.random.Random

class RandomRepository(
    private val db: RandomDatabase,
    private val appScope: CoroutineScope
) {
    private val dispatcher =
        Executors.newSingleThreadExecutor().asCoroutineDispatcher()

    fun summarize() = db.randomStore().summarize()
```

ROOM ACROSS PROCESSES

```
suspend fun populate() {
    withContext(dispatcher + appScope.coroutineContext) {
        val count = Random.nextInt(100) + 1

        db.randomStore().insert((1..count).map { RandomEntity(0) })
    }
}
```

(from [CrossProcess/src/main/java/com/commonsware/room/process/RandomRepository.kt](#))

RandomRepository also has a `summarize()` function that exposes a corresponding function on our DAO:

```
package com.commonsware.room.process

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query
import kotlinx.coroutines.flow.Flow
import java.time.Instant

data class Summary(
    val count: Int,
    val oldestTimestamp: Instant? = null
)

@Dao
interface RandomStore {
    @Insert
    suspend fun insert(entities: List<RandomEntity>)

    @Query("SELECT COUNT(*) as count, MIN(timestamp) as oldestTimestamp FROM randomStuff")
    fun summarize(): Flow<Summary>
}
```

(from [CrossProcess/src/main/java/com/commonsware/room/process/RandomStore.kt](#))

`summarize()` gets the count of entities and the oldest timestamp and emits them via a Flow. That Flow will emit new results as the database is modified and so long as something is observing the Flow. MainActivity gets that data via MainViewModel and shows the count and date on the screen.

In Another Process, a Service

Our manifest has a `<service>` entry for SomeService, placing it into another process via `android:process`:

```
<service android:name=".SomeService" android:process=":something" />
```

(from [CrossProcess/src/main/AndroidManifest.xml](#))

ROOM ACROSS PROCESSES

SomeService also uses `summarize()` on `RandomRepository`, dumping whatever it receives to Logcat:

```
package com.commonware.room.process

import android.content.Intent
import android.os.Process
import android.util.Log
import androidx.lifecycle.LifecycleService
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.launch
import org.koin.android.ext.android.inject
import org.koin.core.qualifier.named

class SomeService : LifecycleService() {
    private val repo: RandomRepository by inject()
    private val appScope: CoroutineScope by inject(named("appScope"))

    override fun onCreate() {
        super.onCreate()

        appScope.launch {
            repo.summarize().collect {
                Log.d("SomeService", "PID: ${Process.myPid()} summary: $it")
            }
        }
    }
}
```

(from [CrossProcess/src/main/java/com/commonware/room/process/SomeService.kt](#))

MainActivity starts that service when it is in the foreground via `onStart()` and stops that service when the UI returns to the background in `onStop()`. This is not a wise use of a service; the point behind a service is to run when the UI is *not* in the foreground. But, it helps illustrate the effects of `enableMultiInstanceInvalidation()`.

Results, Before and After

If we lacked `enableMultiInstanceInvalidation()` — such as if you comment out that line in `RandomDatabase` and run the app — you will find that `SomeService` logs the initial state of the database, but that is it. You can push the big button as much as you want, and the activity will display the current count of entities, but the service will not log new data. That is because our separate process does not know

ROOM ACROSS PROCESSES

that the database changed, so Room does not emit a new result on that process' `summarize()` Flow.

But, if we use `enableMultiInstanceInvalidation()`, now clicking the button causes both the activity and the service to get details of the updated database, so we see both the UI update and the `SomeService` Logcat entry.

Triggers

Triggers are a way to teach SQLite to manipulate Table X every time that you do something to Table Y. While to date Google has elected to not support triggers directly in Room, you can still set them up manually if needed.

In this chapter, we will examine how to do this.

Trigger Basics

Triggers are most often associated with server-side databases: Oracle, SQL Server, and so on. And [triggers have their downsides](#). However, they are an available option on SQLite, which supports [the CREATE TRIGGER statement](#) to define a trigger.

You can think of triggers as being an “if this, then that” sort of construct:

- If we insert a row in Table X, update some data in Table Y
- If we modify a row in Table Y, also modify related rows in Table Z
- If we delete data from Table Z, insert a row in Table Q
- And so on

At a high level, the syntax for creating a trigger is:

```
CREATE TRIGGER [name] [timing] [action] ON [table]
BEGIN
  [SQL statements]
END;
```

Each trigger has a name, the same way that tables and views have names. The timing usually is BEFORE or AFTER, and the action is INSERT, DELETE, or UPDATE OF [columns], with the latter representing modification of some column(s) on one or

more rows. So, you get combinations like:

- BEFORE INSERT
- AFTER UPDATE OF name
- BEFORE DELETE
- And so on

The SQL statements between BEGIN and END; will be executed whenever the action occurs on the specified table, either BEFORE or AFTER the action itself is performed. So, BEFORE executes the SQL statements before the action is applied to the table, while AFTER executes the SQL statements after the action is applied to the table.

Room and Triggers

Developers asked for Google to offer first-class support for triggers back in 2017; [this request was rejected](#).

However, [another request is still open](#). That could be a sign that Room might support triggers directly in the future. Or, it could be a sign that the issue tracker has stale issues.

Regardless, as of Room 2.4.0, there is no @Trigger annotation to teach Room about triggers.

Instead, we have to do it the hard way.

Triggers the Hard Way

So, back in 2017, [Adam McNeilly worked out the basic mechanics of setting up a trigger](#), by using the onCreate() callback function on a RoomDatabase.Callback.

[The Trigger module](#) of [the book's primary sample project](#) demonstrates this approach.

In this project, we have two Room-managed tables:

- randomStuff, containing some random entities that we insert
- countOfRandomStuff, which contains the count of the number of entities in randomStuff, that for some strange reason we want to store in a separate table

TRIGGERS

To keep `countOfRandomStuff` up to date with respect to entities being inserted into `randomStuff`, we could use the following trigger:

```
CREATE TRIGGER updateCount AFTER INSERT ON randomStuff
BEGIN
  UPDATE countOfRandomStuff SET count = (SELECT COUNT(*) FROM randomStuff);
END;
```

(in principle, we also need an `AFTER DELETE` trigger, but since this sample app never deletes data from `randomStuff`, we can skip it)

The `UPDATE countOfRandomStuff` statement will update after one or more rows are inserted into `randomStuff`. The `UPDATE countOfRandomStuff` statement updates *all* of its rows to have a count column reflect the count of rows in `randomStuff`. As it turns out, we will only have one row in that `countOfRandomStuff` table.

We set up `countOfRandomStuff` using a Room `@Entity`:

```
package com.commonware.room.trigger

import androidx.room.Dao
import androidx.room.Entity
import androidx.room.PrimaryKey
import androidx.room.Query

@Entity(tableName = "countOfRandomStuff")
data class CountEntity(
    @PrimaryKey(autoGenerate = true) val id: Long,
    val count: Int
) {
    @Dao
    interface Store {
        @Query("SELECT count FROM countOfRandomStuff LIMIT 1")
        suspend fun getCurrent(): Int
    }
}
```

(from [Trigger/src/main/java/com/commonware/room/trigger/CountEntity.kt](https://github.com/CommonWare/room-trigger/blob/main/src/main/java/com/commonware/room/trigger/CountEntity.kt))

That `CountEntity` also has a nested DAO interface, describing a `getCurrent()` function that will return the count value for the first row in the `countOfRandomStuff` table.

`RandomDatabase` not only hooks up both the `CountEntity` and the `RandomEntity` (that defines the `randomStuff` table), but it also does some extra work when the

TRIGGERS

database is created:

```
package com.commonware.room.trigger

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters
import androidx.sqlite.db.SupportSQLiteDatabase

private const val DB_NAME = "random.db"

@Database(entities = [RandomEntity::class, CountEntity::class], version = 1)
@TypeConverters(TypeTransmogriifier::class)
abstract class RandomDatabase : RoomDatabase() {
    abstract fun randomStore(): RandomStore
    abstract fun countStore(): CountEntity.Store

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, RandomDatabase::class.java, DB_NAME)
                .addCallback(object : RoomDatabase.Callback() {
                    override fun onCreate(db: SupportSQLiteDatabase) {
                        super.onCreate(db)

                        db.execSQL("INSERT INTO countOfRandomStuff (count) VALUES (0);")
                        db.execSQL(
                            """
CREATE TRIGGER updateCount AFTER INSERT ON randomStuff
BEGIN
    UPDATE countOfRandomStuff SET count = (SELECT COUNT(*) FROM randomStuff);
END;
""").trimIndent()
                    }
                })
                .build()
    }
}
```

(from [Trigger/src/main/java/com/commonware/room/trigger/RandomDatabase.kt](#))

We use `addCallback()` and a `RoomDatabase.Callback` object to get control when the database is created, via `onCreate()`. In there, we do two things:

1. Insert a single row into `countOfRandomStuff`, so our trigger always has a row

TRIGGERS

- to update; and
2. Executes the `CREATE TRIGGER` shown earlier in this chapter, to define our trigger

Now, every time that we insert rows into `randomStuff`, `countOfRandomStuff` will get updated. We can access the `countOfRandomStuff` data via `CountEntity` and its DAO, just like any other Room-managed table. That is because `countOfRandomStuff` is a Room-managed table, just one whose contents are set up via the `onCreate()` callback and the trigger, rather than via calls on the DAO itself.

What's New in Room?

Room keeps changing, adding in new features with every major and minor release. This chapter highlights some of those changes.

Version 2.3.x

April 2021 saw the release of 2.3.0. As of late October 2021, no new patch releases have been required.

2.3.0 gave us a bunch of improvements, particularly around type and column management.

Enum Support

Historically, if your `@Entity` had a property that was some `enum` class, you needed to write [a type converter](#) to convert that `enum` to and from something else, such as an `Int` or `String`. That is still an option.

However, Room 2.3.0 added automatic conversion of an `enum` class value to and from its `String` representation. So, now you can use an `enum` without a `@TypeConverter` pair:

```
package com.commonware.room.misc

import androidx.room.*

enum class Silly { First, Second, Third }

@Entity(tableName = "autoEnum")
data class AutoEnumEntity(
```

WHAT'S NEW IN ROOM?

```
@PrimaryKey(autoGenerate = true)
var id: Long,
var silly: Silly
) {
    @Dao
    abstract class Store {
        @Query("SELECT * FROM autoEnum")
        abstract fun loadAll(): List<AutoEnumEntity>

        @Query("SELECT * FROM autoEnum WHERE id = :id")
        abstract fun findById(id: Int): AutoEnumEntity

        fun insert(entity: AutoEnumEntity): AutoEnumEntity {
            entity.id = _insert(entity)

            return entity
        }

        @Insert
        abstract fun _insert(entity: AutoEnumEntity): Long
    }
}
```

(from [MiscSamples/src/main/java/com/commonsware/room/misc/AutoEnumEntity.kt](#))

In this case, our silly column is declared as TEXT in the generated SQL:

```
CREATE TABLE IF NOT EXISTS `autoEnum` (`id` INTEGER PRIMARY KEY AUTOINCREMENT NOT
NULL, `silly` TEXT NOT NULL)
```

Part of what Room code-generates for us are functions to convert the enum class values to String representations and back:

```
private String __Silly_enumToString(final Silly _value) {
    if (_value == null) {
        return null;
    } switch (_value) {
        case First: return "First";
        case Second: return "Second";
        case Third: return "Third";
        default: throw new IllegalArgumentException("Can't convert enum to string,
unknown enum value: " + _value);
    }
}

private Silly __Silly_stringToEnum(final String _value) {
    if (_value == null) {
```

```
    return null;
} switch (_value) {
    case "First": return Silly.First;
    case "Second": return Silly.Second;
    case "Third": return Silly.Third;
    default: throw new IllegalArgumentException("Can't convert value to enum,
unknown value: " + _value);
}
}
```

Room then uses those functions as if they were a custom `@TypeConverter`, to convert our enum class values to and from a String representation to put in the TEXT column.

Type Converter Improvements

The `@TypeConverter` annotation is a useful way to [get Room to recognize types](#) that it otherwise could not handle. However, `@TypeConverter` had been fairly inflexible: you had to implement the functions on a class, where Room would instantiate that class as needed.

Room 2.3.0 improves the flexibility in this area.

The simple improvement is that now you can have `@TypeConverter` annotations on an object, saving Room the need to instantiate the class.

The more complex improvement is in a new `@ProvidedTypeConverter` annotation. This is for cases where you are happy to have the `@TypeConverter` functions be on a class, but you want to be in charge of creating the object for that class. One popular case is where you want to use a dependency inversion framework (Dagger/Hilt, Koin, etc.) — for example, perhaps you are providing a JSON converter via DI and need to inject the converter into an object that will use that converter for `@TypeConverter` functions.

To make this work, you write a class to house your `@TypeConverter` functions as normal. However, now you can add a constructor on that class or otherwise configure it as you see fit. You do, however, also have to add `@ProvidedTypeConverter` as a class-level annotation:

```
@ProvidedTypeConverter
class SomeValueTypeConverter(private val adapter: JsonAdapter<SomeValueType>) {
    @TypeConverter
    fun someValueFromJson(json: String) = adapter.fromJson(json)
```

```
@TypeConverter
fun someValueToJson(value: SomeValueType) = adapter.toJson(value)
}
```

Here, we have a `SomeValue` type (not shown here) and an injected [Moshi](#) adapter for that type. The `@TypeConverter`-annotated functions simply delegate to Moshi.

`@ProvidedTypeConverter` tells Room to allow classes without a public zero-argument constructor. However, the cost is in the second step: you need to supply the instance of your `@ProvidedTypeConverter` class to the `RoomDatabase.Builder`, via an `addTypeConverter()` call:

```
class YourRepository(private val someValueTypeAdapter: JsonAdapter<SomeValueType>) {
    val db = Room.databaseBuilder(context, YourDatabase::class.java, DB_NAME)
        .addTypeConverter(SomeValueTypeConverter(someValueTypeAdapter))
        .build()
}
```

The name of the annotation explains the rule: by using `@ProvidedTypeConverter`, you have more flexibility, but you have the corresponding responsibility to provide the instance of the type converter to the `RoomDatabase`.

Packaged Databases Improvements

As we have seen, you can [package a database with your app](#), so you have starter data immediately when the app is first opened. Room 2.3.0 makes a couple of improvements in this area.

In addition to using databases packaged as assets — as [the chapter on packaged databases](#) focuses on — you can use the same technique to populate your database from a file, using `createFromFile()` instead of `createFromAsset()`. This, however, requires a `File` object, and those are more difficult to come by given the “scoped storage” added to Android 10. Room 2.3.0 now gives us `createFromInputStream()`, so we can get our starter database from a `InputStream`, such as one that we might get from `openInputStream()` on a `ContentResolver` using a `Uri`.

Also, these `create...()` functions now have a variant that takes an additional `RoomDatabase.PrepackagedDatabaseCallback` object (whose name makes the author *very* grateful for auto-completion in modern IDEs...). This will be called with `onOpenPrepackagedDatabase()` *after* the data source has been copied and the Room database is set up. That way, if you need to do some cleanup (e.g., delete a

downloaded file from `getCacheDir()`, you know it is safe to do so.

@RewriteQueriesToDropUnusedColumns

Using `*` in a `@Query` SQL `SELECT` statement is easy. However, by default, Room is not very sophisticated about using it. Room will happily retrieve *all* the columns available to it from your table, even if your entity or other output object only uses a few of those columns. Plus, you will get a compile-time warning from Room about the inefficiency. It is more efficient to have your `SELECT` list the exact columns that you need, but this gets tedious to keep in sync with your output object structures.

Adding `@RewriteQueriesToDropUnusedColumns` to your `@Query`-annotated function tells Room to try to handle this automatically. The Room compiler will replace your `*` with the column names that are needed in the SQL that it actually executes. That way, you get the convenience of `*` and the efficiency of only querying for the needed columns. You can also add `@RewriteQueriesToDropUnusedColumns` to the entire `@Dao` class or interface, to have it apply to all `@Query` functions.

Paging 3 Support

If you are using Paging 3 — the now-latest generation of the Jetpack Paging framework — Room will now let you set up DAO functions that support it. [The chapter on paging](#) has been updated to show Paging 3.

RoomDatabase.QueryCallback

For logging purposes, it might be useful to learn when Room executes some SQL. For that, you can call `setQueryCallback()` on your `RoomDatabase.Builder` when you are setting up Room. This function takes a `RoomDatabase.QueryCallback` object, which has a single `onQuery()` function — this makes it easy to be replaced by a lambda expression in Java or Kotlin. There, you get a `String` of the SQL to be executed and a `List` of the objects to be bound as replacements for `?`-style placeholders in that SQL.

Do be careful, though, about your logging, as the bind objects may contain data that should be considered private. Consider only logging in debug builds.

RxJava 3 Support

Not everybody realizes this, but RxJava 2.x is end-of-life: other than some bug fixes,

WHAT'S NEW IN ROOM?

no new updates are planned for that. The RxJava team has moved on to RxJava 3, which has a similar API to RxJava 2.x but not quite identical.

Previously, Room supported other reactive options — LiveData, RxJava 2.x, coroutines — but not RxJava 3. Now, using `androidx.room:room-rxjava3`, you can use RxJava 3 types in your Room DAO functions, if you so choose.