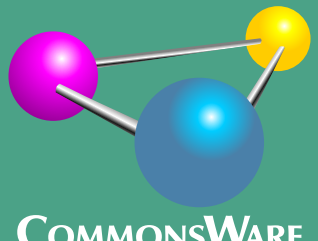
Android's Architecture Components

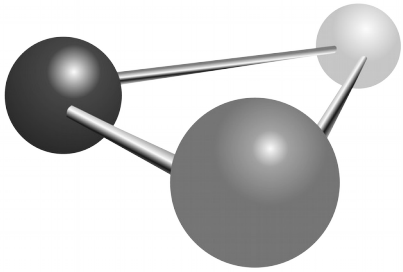
Mark L. Murphy



CommonsWare

Android's Architecture Components

by Mark L. Murphy



COMMONSWARE

Android's Architecture Components
by Mark L. Murphy

Printed in the United States of America.

Printing History:
January 2019: FINAL

Table of Contents

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

- Preface
  - First-Generation Book vii
  - Prerequisites viii
  - Source Code and Its License viii
  - Creative Commons and the Four-to-Free (42F) Guarantee viii
  - Acknowledgments ix
- Room Basics
  - Wrenching Relations Into Objects 1
  - Room Requirements 2
  - Room Furnishings 3
  - Get a Room 8
- Testing Room
  - Writing Instrumentation Tests 11
  - Writing Unit Tests via Mocks 14
- The Dao of Entities
  - Configuring Entities 19
  - DAOs and Queries 29
  - Dynamic Queries 35
  - Other DAO Operations 37
  - Transactions and Room 40
  - Threads and Room 42
- Room and Custom Types
  - Type Converters 45
  - Embedded Types 53
  - Updating the Trip Sample 56
- Room and Relations
  - The Classic ORM Approach 61
  - A History of Threading Mistakes 62
  - The Room Approach 63
  - Plans for Trips 64
  - Self-Referential Relations for Tree Structures 72
  - Using @Relation 74
  - @Relation and @Query 76
  - Representing No Relation 76
- The Support Database API

    - "Can't You See That This is a Facade?" 79
    - When Will We Use This? 81
    - Configuring Room's Database Access 81
- Room and Migrations
    - What's a Migration? 85
    - When Do We Migrate? 86
    - But First, a Word About Exporting Schemas 86
    - Writing Migrations 89
    - Employing Migrations 94
    - How Room Applies Migrations 96
    - Testing Migrations 96
- Lifecycles and Owners
    - A Tale of Terminology 103
    - Adding the Lifecycle Components 104
    - Getting a Lifecycle 105
    - Observing a Lifecycle 107
    - Legacy Options 108
    - So, What's the Point of This? 112
- LiveData
    - Observables Are the New Black 115
    - Yet More Terminology 116
    - Implementing LiveData 117
    - Other LiveData Examples 122
- ViewModel
    - Viewmodels, As Originally Envisioned 127
    - ViewModel Versus... 128
    - Dependencies 129
    - Mommy, Where Does a ViewModel Come From? 129
    - ViewModel In Action 130
- Other Lifecycle Owners
    - LifecycleService 135
    - ProcessLifecycleOwner 136
    - Wait... Where Are LifecycleProvider and LifecycleReceiver? 140
- LiveData and Data Binding
    - A Data Binding Recap 141
    - LiveData Updating Data Binding 144
    - Handling Changes to LiveData 150
    - The Saved Instance State Situation 152
- WorkManager
    - Where Should We Use WorkManager? 157
    - Where Should We Not Use WorkManager? 158

    - WorkManager Dependencies 158
    - Workers: They Do Work 159
    - Performing Simple Work 161
    - Work Inputs 162
    - Constrained Work 163
    - Tagged Work 164
    - Monitoring Work 165
    - Canceling Work 168
    - Delayed Work 169
    - Parallel Work 169
    - Chained Work 170
    - Periodic Work 176
    - Unique Work 177
    - Testing Work 177
    - WorkManager and Side Effects 180
- M:N Relations in Room
    - Implementing a Join Entity 185
    - Implementing DAO Methods 189
    - Where's That Good Ol' Object Feel? 191
- Polymorphic Room Relations
    - Polymorphism With Separate Tables 193
    - Polymorphism With a Single Table 199
    - Polymorphism With M:N Relations 203
- LiveData Transformations
    - The Bucket Brigade 205
    - Mapping Data to Data 206
    - Mapping Data to... LiveData? 207
    - Writing a Transformation 208
    - Do We Really Want This? 210
- RxJava and Room
    - Adding RxJava 213
    - Decisions, Decisions 214
    - The One-Time Option: Single 215
    - The One-Time 0-1 Option: Maybe 216
    - The Ongoing Option: Flowable 216
    - @RawQuery and Reactive Responses 217
- RxJava and Lifecycles
    - The Classic Approach 219
    - Bridging RxJava and LiveData 220
    - The Uber Solution: AutoDispose 223
- Packing Up a Room

    - The Problem 225
    - The Classic Solution: SQLiteAssetHelper 226
    - The New Problem 227
    - Merging SQLiteAssetHelper with Room 227
- Paging Room Data
    - The Problem: Too Much Data 231
    - Addressing the UX 232
    - Enter the Paging Library 232
    - Paging and Room 234
    - What About RxJava? 243
- LiveData and Bound Services
    - Old API, New Coat of Paint 245
    - Remote Sensors 246
- Immutability
    - The Benefits of Immutability 257
    - The Costs of Immutability 259
    - Immutability via AutoValue 262
- The Repository Pattern
    - What the Repository Does 267
    - High-Level Repository Strategies 270
    - Let's Roll the Dice 272
    - Blending Data Sources 285
- Introducing Model-View-Intent
    - GUI Architectures 293
    - Why People Worry About GUI Architectures 294
    - Why Others Ignore GUI Architectures 296
    - A Rough Comparison of GUI Architectures 298
    - The Basics of Model-View-Intent 302
    - Additional MVI Resources 305
- A Deep Dive Into MVI
    - What the Sample App Does 307
    - MVI and the Sample App 316
    - The Model 316
    - The View State 318
    - The View 320
    - The Actions 325
    - Publishing Actions 328
    - The Repositories 330
    - The Controller 335
    - About Those Results 339
    - The Reducer in the RosterViewModel 341

    - Examining the Other Fragments 346
    - Summary 347
- Backing Up a Room
    - What Do We Need to Back Up? 351
    - Beware of Open Rooms 352
    - When Do We Back Up the Database? 353
    - A Basic Backup Example 355
    - Backing Up Off-Device 362
- Room and Full-Text Searching
    - What Is FTS? 363
    - The Room 1.x FTS Recipe 364
    - Searching a Book 366
- Room and Conflict Resolution
    - Abort 388
    - Fail 390
    - Ignore 390
    - Replace 391
    - Rollback 392
    - What Should You Use with Room? 392
- Configuring SQLite Beyond Room
    - When To Make Changes 395
    - Example: Turbo Boost Mode 396

Preface

Thanks!

Thanks for your interest in Android app development, the world's most popular operating system! And, thanks for your interest in the Android Architecture Components, released by Google in 2017 to help address common "big-ticket" problems in Android app development.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful!

(OTOH, if you find it completely useless... um, don't tell anyone, OK?)

First-Generation Book

Android app development can be divided into two generations:

- First-generation app development uses Java as the programming language and leverages the Android Support Library and the `android.arch` edition of the Architecture Components
- Second-generation app development more often uses Kotlin as the programming language and leverages AndroidX and the rest of Jetpack (which includes an AndroidX edition of the Architecture Components)

This book is a first-generation book. It explores the `android.arch` edition of the Architecture Components and it uses Java for most of the examples.

Second-generation material can be found in CommonsWare's "Elements" book series. Of particular note, some introductory Architecture Components material can

be found in *Elements of Android Jetpack*.

Prerequisites

This book is targeted at:

- People who have read the core chapters of the companion volume, *The Busy Coder's Guide to Android Development*, or
- Intermediate Android app developers — those with some experience but not necessarily "experts" in the field

Source Code and Its License

The source code samples shown in this book are available for download from the book's GitHub repository. All of the Android projects are 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.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-ShareAlike 4.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 January 2023*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike

3.0 license, visit the Creative Commons Web site

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

The author would like to thank the Google team responsible for the Architecture Components for their work in making this library.

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 — undoubtedly you have encountered the "object-relational impedance mismatch". That is a very fancy way of saying "gosh, 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 Java 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, or ORMs, take. A

typical ORM works off of Java code and either generates a suitable database structure or works with you to identify how the Java 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 platfoms like Android devices. However, a vast roster of Android ORMs have been created over the years to try to fill that gap. Some of the more popular ones have been:

- 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 middle managers.

While this book is focused on the Architecture Components — and Room is part of those — you may wish to explore other ORMs if you are interested in using Java objects but saving the data in SQLite. Room is likely to become popular, but it is far from the only option.

Room Requirements

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

1. The runtime library version, using the standard `implementation` directive
2. An annotation processor, using the `annotationProcessor` directive

```
implementation "android.arch.persistence.room:runtime:1.1.1"
annotationProcessor "android.arch.persistence.room:compiler:1.1.1"
```

(from Trips/RoomBasics/app/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 divided into three sets of classes:

1. Entities, which are POJOs that model the data you are transferring into and out of the database
2. The data access object (DAO), that provides the description of the Java 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 (or whatever) to create based on the Web service response

In this chapter, we will look at the `Trips/RoomBasics` sample project. This app is the first of a linked series of apps that we will examine in this book, containing some code for a travel itinerary manager. Right now, though, we are settling for being able to see some *very* rudimentary trips get into and out of a database.

Entities

In many ORM systems, the entity (or that system's equivalent) is a POJO 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 POJOs representing:

- the data that you want to store into the database, 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 POJOs, 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 class marked with the `@Entity` annotation. For example, here is a `Trip` class that serves as a Room entity:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
import java.util.UUID;

@Entity(tableName = "trips")
class Trip {
  @PrimaryKey
  @NonNull
  public final String id;
  public final String title;
  final int duration;

  @Ignore
  Trip(String title, int duration) {
    this(UUID.randomUUID().toString(), title, duration);
  }

  Trip(@NonNull String id, String title, int duration) {
    this.id=id;
    this.title=title;
    this.duration=duration;
  }

  @Override
  public String toString() {
    return(title);
  }
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/Trip.java)

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

Sometimes, your fields 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 `Trip` objects by their primary key values. 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`.

Similarly, sometimes your methods will be marked with annotations. In this case, `Trip` has two constructors: one that generates the `id` from a `UUID`, and one that takes the `id` as a constructor parameter. Room needs to know which constructor(s) are eligible for its use; you mark the other constructors with the `@Ignore` annotation.

For Room to work with a field, it needs to be `public` or have JavaBean-style getter and setter methods, so Room can access them. If the fields are `final`, as they are on `Trip`, Room will try to find a constructor to use to populate the fields, as `final` fields will lack setters.

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, whatever.

In Room, the DAO is identified by the `@Dao` annotation, applied to either an `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 methods annotated `@GET` or `@POST` in Retrofit.

The sample app has a `TripStore` that is our DAO:

```
package com.commonsware.android.room;
```

```
import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

Besides the `@Dao` annotation on the `TripStore` interface, we have five methods, each with their own annotations. Your four main annotations for these methods are `@Query`, `@Insert`, `@Update`, and `@Delete`, which map to the corresponding database operations.

Two `TripStore` methods — `selectAll()` and `findById()` — have the `@Query` annotation. Principally, `@Query` will be used for SQL `SELECT` statements, where you put the actual SQL in the annotation itself. To a large extent, any valid SQLite query can be used here. However, instead of using `?` as placeholders for arguments, as we would in traditional SQLite, you use `:`-prefixed method parameter names. So, in `findById()`, we have a `String` parameter named `id`, so we can use `:id` in the SQL statement wherever we might have used `?` to indicate the value to bind in.

The remaining three methods use the `@Insert`, `@Update`, and `@Delete` annotations, mapped to methods of the same name. Here, the methods take a varargs of `Trip`, meaning that we can insert, update, or delete as many `Trip` objects as we want (passing in zero `Trip` objects works, though that would be rather odd).

If you want custom code on your DAO, beyond the code-generated implementations of your Room-annotated methods, use an `abstract` class and mark all the Room-annotated methods as `abstract`. If, on the other hand, all you need on the DAO are the Room-annotated methods, you can use an `interface` and skip all the `abstract` keywords, as we did with `TripStore`.

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 `TripDatabase` serving this role:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;

@Database(entities={Trip.class}, version=1)
abstract class TripDatabase extends RoomDatabase {
  abstract TripStore tripStore();

  private static final String DB_NAME="trips.db";
  private static volatile TripDatabase INSTANCE=null;

  synchronized static TripDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static TripDatabase create(Context ctxt, boolean memoryOnly) {
    RoomDatabase.Builder<TripDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        TripDatabase.class);
    }
```

```
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
        DB_NAME);
    }

    return(b.build());
  }
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

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)

```
@Database(entities={Trip.class}, version=1)
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

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

You also need `abstract` methods for each DAO class that return an instance of that class:

```
  abstract TripStore tripStore();
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

In this app, we have only one DAO (`TripStore`), so we have an `abstract` method to return an instance of `TripStore`.

Extending `RoomDatabase`, having the `@Database` annotation, and having the `abstract` method(s) for your DAOs are the requirements. Anything beyond that is up to you, and some apps may elect to have nothing more here.

In our case, we have a bit more logic.

Get a Room

In this example, the database is a singleton. `TripDatabase` has a `static` getter

method, cunningly named `get()`, that creates our singleton. `get()`, in turn, calls a `create()` method that is responsible for creating our `TripDatabase`:

```
static TripDatabase create(Context ctxt, boolean memoryOnly) {
  RoomDatabase.Builder<TripDatabase> b;

  if (memoryOnly) {
    b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
      TripDatabase.class);
  }
  else {
    b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
      DB_NAME);
  }

  return(b.build());
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

To create a `TripDatabase`, we use a `RoomDatabase.Builder`, which we get by calling one of two methods on the `Room` class:

- `databaseBuilder()` is what you will normally use
- `inMemoryDatabaseBuilder()` does what the method name suggests: it creates an in-memory SQLite database, useful for instrumentation tests where you do not necessarily need to persist the data for a user

Both of those methods take a `Context` and the Java `Class` object for the desired `RoomDatabase` subclass. `databaseBuilder()` also takes the filename of the SQLite database to use, much as `SQLiteOpenHelper` does in traditional Android SQLite development.

While there are some configuration methods that can be called on the `RoomDatabase.Builder`, we skip those here, simply calling `build()` to build the `TripDatabase`. The result is that when we call `get()`, we get a singleton lazy-initialized `TripDatabase`.

From there, we can:

- Call `tripStore()` on the `TripDatabase` to retrieve the `TripStore` DAO
- Call methods on the `TripStore` to query, insert, update, or delete `Trip` objects

We will see how to do that in the next chapter, where we look at how to write instrumentation tests for our Room-generated database code.

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.

Writing Instrumentation Tests

On the whole, writing instrumentation 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 instrumentation test case class to exercise the `TripDatabase` from the preceding chapter:

```
package com.commonsware.android.room;

import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import java.util.List;
import static junit.framework.Assert.assertNotNull;
import static junit.framework.Assert.assertTrue;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
```

```
@RunWith(AndroidJUnit4.class)
public class TripTests {
  TripDatabase db;
  TripStore store;

  @Before
  public void setUp() {
    db=TripDatabase.create(InstrumentationRegistry.getTargetContext(), true);
    store=db.tripStore();
  }

  @After
  public void tearDown() {
    db.close();
  }

  @Test
  public void basics() {
    assertEquals(0, store.selectAll().size());

    final Trip first=new Trip("Foo", 2880);

    assertNotNull(first.id);
    assertNotEquals(0, first.id.length());
    store.insert(first);

    assertTrip(store, first);

    final Trip updated=new Trip(first.id, "Foo!!!", 1440);

    store.update(updated);
    assertTrip(store, updated);

    store.delete(updated);
    assertEquals(0, store.selectAll().size());
  }

  private void assertTrip(TripStore store, Trip trip) {
    List<Trip> results=store.selectAll();

    assertNotNull(results);
    assertEquals(1, results.size());
    assertTrue(areIdentical(trip, results.get(0)));

    Trip result=store.findById(trip.id);

    assertNotNull(result);
    assertTrue(areIdentical(trip, result));
```

```
  }

  private boolean areIdentical(Trip one, Trip two) {
    return(one.id.equals(two.id) &&
      one.title.equals(two.title) &&
      one.duration==two.duration);
  }
}
```

(from Trips/RoomBasics/app/src/androidTest/java/com/commonsware/android/room/TripTests.java)

Here, we:

- Create an empty database
- Get the DAO (`TripStore`)
- Confirm that there are no trips in the database
- Create a `Trip` object and `insert()` it into the database, then confirm that the database was properly inserted
- Create a new `Trip` object with the same ID as the first, `update()` the database using it, then confirm that the database was properly modified
- Delete the `Trip` object, then confirm that the database has no trips once again

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 `TripTests` class — is to use an in-memory database. The static `create()` method on `TripDatabase`, if you pass `true` for the second parameter, creates a `TripDatabase` backed by memory, not disk.

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

1. It is intrinsically self-contained. Once the `TripDatabase` is closed, its memory is released, and if separate tests use separate `TripDatabase` 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 instrumentation 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.

Importing Starter Data

The one downside to having an empty starter database, such as a fresh in-memory database, is that you have no data. Eventually, you need some data to test.

That could come from test code, such as what `TripTests` does. In many cases, this is a necessary part of testing, to confirm that all of your DAO methods work as expected.

Alternatives include:

- Loading the data from some neutral format (e.g., JSON) via some utility method
- Packaging one or more starter database as assets in the instrumentation tests (e.g., `src/androidTest/assets/`), then using `ATTACH DATABASE ...` and `INSERT INTO ... SELECT FROM ...` SQLite code to copy from the starter database to the database to be used in testing

Writing Unit Tests via Mocks

Let's look again at the `TripStore` DAO:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();
```

```
  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

This is a pure interface. More importantly, other than annotations, its API is purely domain-specific. Everything revolves around our `Trip` entity and other business logic (e.g., `String` values as identifiers).

Room DAOs are designed to be mocked, using a mocking library like Mockito, so that you can write unit tests (tests that run on your development machine or CI server) in addition to — or perhaps instead of — instrumentation tests.

The primary advantage of unit tests is execution speed, as they do not have to be run on Android devices or emulators. On the other hand, setting up mocks can be tedious.

The `RoomBasics` project not only has the instrumentation tests shown earlier in this chapter, but an equivalent unit test in `test/`, embodied in a `TripUnitTests` class:

```
package com.commonsware.android.room;

import org.junit.Before;
import org.junit.Test;
import org.mockito.ArgumentMatchers;
import org.mockito.Mockito;
import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
```

```
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import static org.mockito.Matchers.any;
import static org.mockito.Mockito.doAnswer;

public class TripUnitTests {
  private TripStore store;

  @Before
  public void setUp() {
    store=Mockito.mock(TripStore.class);

    final HashMap<String, Trip> trips=new HashMap<>();

    doAnswer(new Answer() {
      @Override
      public Object answer(InvocationOnMock invocation) throws Throwable {
        ArrayList<Trip> result=new ArrayList<>(trips.values());

        Collections.sort(result, new Comparator<Trip>() {
          @Override
          public int compare(Trip left, Trip right) {
            return(left.title.compareTo(right.title));
          }
        });

        return(result);
      }
    }).when(store).selectAll();

    doAnswer(new Answer() {
      @Override
      public Object answer(InvocationOnMock invocation) throws Throwable {
        String id=(String)invocation.getArguments()[0];

        return(trips.get(id));
      }
    }).when(store).findById(any(String.class));

    doAnswer(new Answer() {
      @Override
      public Object answer(InvocationOnMock invocation) throws Throwable {
        for (Object rawTrip : invocation.getArguments()) {
          Trip trip=(Trip)rawTrip;

          trips.put(trip.id, trip);
        }
```

```
      return(null);
    }
  }).when(store).insert(ArgumentMatchers.any());

  doAnswer(new Answer() {
    @Override
    public Object answer(InvocationOnMock invocation) throws Throwable {
      for (Object rawTrip : invocation.getArguments()) {
        Trip trip=(Trip)rawTrip;

        trips.put(trip.id, trip);
      }

      return(null);
    }
  }).when(store).update(ArgumentMatchers.any());

  doAnswer(new Answer() {
    @Override
    public Object answer(InvocationOnMock invocation) throws Throwable {
      for (Object rawTrip : invocation.getArguments()) {
        Trip trip=(Trip)rawTrip;

        trips.remove(trip.id);
      }

      return(null);
    }
  }).when(store).delete(ArgumentMatchers.any());
}

@Test
public void basics() {
  assertEquals(0, store.selectAll().size());

  final Trip first=new Trip("Foo", 2880);

  assertNotNull(first.id);
  assertNotEquals(0, first.id.length());
  store.insert(first);

  assertTrip(store, first);

  final Trip updated=new Trip(first.id, "Foo!!!", 1440);

  store.update(updated);
  assertTrip(store, updated);
```

```
    store.delete(updated);
    assertEquals(0, store.selectAll().size());
  }

  private void assertTrip(TripStore store, Trip trip) {
    List<Trip> results=store.selectAll();

    assertNotNull(results);
    assertEquals(1, results.size());
    assertTrue(areIdentical(trip, results.get(0)));

    Trip result=store.findById(trip.id);

    assertNotNull(result);
    assertTrue(areIdentical(trip, result));
  }

  private boolean areIdentical(Trip one, Trip two) {
    return(one.id.equals(two.id) &&
      one.title.equals(two.title) &&
      one.duration==two.duration);
  }
}
```

(from Trips/RoomBasics/app/src/test/java/com/commonsware/android/room/TripUnitTests.java)

The `basics()` test method, and its supporting utility methods, are the same as in the instrumentation tests. What differs is where the `TripStore` comes from. In the instrumentation tests, we created an in-memory `TripDatabase` and retrieved a `TripStore` from it. In the unit tests, we use Mockito to create a mock `TripStore` (via `Mockito.mock(TripStore.class)`), then teach the mock how to respond to its methods. In this case, we mock a database with a simple `HashMap`, with a roster of the trips, keyed by their ID values. Each of the `doAnswer()` calls mocks a specific method on the `TripStore`, down to the details of having `selectAll()` return the trips ordered by title.

Whether this is worth the effort is for you to decide. For many projects, instrumentation tests will suffice. For larger projects, where the speed difference between unit tests and instrumentation tests is substantial, it might be worth the engineering time to create the mocks. While mocking is also useful for scenarios that are difficult to reproduce, it is unlikely that your DAO will have any of those scenarios.

The Dao of Entities

Two chapters ago, we went through the basic steps for setting up Room:

- Create and annotate your entity classes
- Create, annotate, and define operator methods 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, while you are safely on a background thread
- 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 — and the subsequent chapters on custom types and relations — we will explore the rest of the configuration for entities and DAOs.

Many of the code snippets shown in this chapter come from the `General/RoomDao` sample project. This contains a library module (`stuff`) with entity and DAO code along with instrumentation tests for bits of that code.

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 `int` or `Integer` field that is your `@PrimaryKey`, you can optionally apply `AUTOINCREMENT` to the corresponding column by adding `autoGenerate=true` to the annotation:

```
@Entity
public class Constant {
  @PrimaryKey(autoGenerate=true)
  @NonNull
  public int id;
  String title;
  double value;

  @Override
  public String toString() {
    return(title);
  }
}
```

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

```
CREATE TABLE IF NOT EXISTS Constant (id INTEGER PRIMARY KEY AUTOINCREMENT, title
TEXT, value REAL 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. That presents “trickle-down” complications — for example, you cannot make the primary key field `final`, as then you cannot create an instance of an entity that is not yet in the database. While you can try to work around this (e.g., default the `id` to `0`), then you have to keep

checking to see whether you have a valid identifier.

Most of the samples in this book will use a `UUID` instead. While these take up much more room than a simple `int`, 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.

Also, notice that the `value` column has `NOT NULL` applied to it. Room's rule is that primitive fields (`int`, `double`, etc.) will be `NOT NULL`, while their object equivalents (`Integer`, `Double`, etc.) will allow null values.

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 field, as we will see in the next chapter. However, it may be that they should be individual fields 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`.

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:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  @NonNull public final String id;
  public final int versionCode;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

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 VersionedThingy (id TEXT NOT NULL, versionCode INTEGER NOT NULL, PRIMARY KEY(id, versionCode))
```

Even though we are using `primaryKeys` rather than `@PrimaryKey`, the `@NonNull` requirement still holds. We need to add that to any of our `primaryKeys` fields that are of object types. Since `id` is a `String`, we need `@NonNull`. However, `versionCode` is an `int`, and an `int` cannot be `null`, so we do not need `@NonNull` (though having it will not cause a problem). If `versionCode` were an `Integer`, we would need `@NonNull`, as an `Integer` field could be `null`.

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, use the `indices` property on `@Entity`. This property takes a list of `@Index` annotations, each of which declares an index.

For example, as part of a `Customer` entity, you might have an address, which might contain a `postalCode`. You might be querying directly on `postalCode` as part of a search form, and so having an index on that would be useful. To do that, add the appropriate `@Index` to `indices`:

```
@Entity(indices={@Index("postalCode")})
class Customer {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String postalCode;
  public final String displayName;

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

Room will add the requested index:

```
CREATE INDEX index_Customer_postalCode ON Customer (postalCode)
```

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

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

```
@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String postalCode;
  public final String displayName;

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

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

```
CREATE UNIQUE INDEX index_Customer_postalCode ON Customer (postalCode)
```

Ignoring Fields

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

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  @NonNull public final String id;
  public final int versionCode;

  @Ignore
  private String something;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
```

```
  }
}
```

That annotation is required. For example, this does not work:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  @NonNull public final String id;
  public final int versionCode;

  private String something;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

You might think that since the field is private and has no setter, that Room would ignore it automatically. Room, instead, generates a build error, as it cannot tell if you want to ignore that field or if you simply forgot to add it properly.

With Room, `transient` fields are ignored automatically by default, so in the following code snippet, `something` will be ignored:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  @NonNull public final String id;
  public final int versionCode;

  public transient String something;

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

As seen earlier in the book, you can also `@Ignore` constructors. This may be required to clear up Room build errors, if the code generator cannot determine what constructor to use:

```
@Entity(primaryKeys={"id", "versionCode"})
class VersionedThingy {
  @NonNull public final String id;
```

```
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

NOT NULL Fields

As noted earlier, primitive fields get converted into `NOT NULL` columns in the table, while object fields allow null values.

If you want an object field to be `NOT NULL`, apply the `@NonNull` annotation:

```
@Entity(indices={@Index("postalCode")})
class Customer {
  @PrimaryKey
  @NonNull
  public final String id;

  @NonNull
  public final String postalCode;
  public final String displayName;

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

This will make the associated column have `NOT NULL` applied to it.

Custom Table and Column Names

By default, Room will generate names for your tables and columns based off of the

entity class names and field 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 methods.

To control the table name, use the `tableName` property on the `@Entity` attribute, and give it a valid SQLite table name. For example, while in Java we might want to call the class `VersionedThingy`, we might prefer the table to just be `thingy`:

```
@Entity(tableName="thingy", primaryKeys={"id", "versionCode"})
class VersionedThingy {
  @NonNull public final String id;
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

To rename a column, add the `@ColumnInfo` annotation to the field, with a `name` property that provides your desired name for the column:

```
@Entity(tableName="thingy", primaryKeys={"id", "versionCode"})
class VersionedThingy {
  @NonNull public final String id;

  @ColumnInfo(name="version_code")
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
```

```
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

Here, we changed the `versionCode` field's column to `version_code`, along with specifying the table name.

However, this fails. The values in the `primaryKeys` property are the *column names*, not the field names. Since we renamed the column, we need to update `primaryKeys` to match:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.ColumnInfo;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
import java.util.UUID;

@Entity(tableName="thingy", primaryKeys={"id", "version_code"})
class VersionedThingy {
  @NonNull public final String id;

  @ColumnInfo(name="version_code")
  @NonNull
  public final int versionCode;

  @Ignore
  private String something;

  @Ignore
  VersionedThingy() {
    this(UUID.randomUUID().toString(), 1);
  }

  VersionedThingy(String id, int versionCode) {
    this.id=id;
    this.versionCode=versionCode;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/VersionedThingy.java)

Also note that adding `@ColumnInfo` to a `transient` field means that this field will be included when creating the table structure. By default, `transient` fields 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.

Indexing

You can add an `index` property to indicate that you want to index the column, as an alternative to listing the column in the `indices` property of the `@Entity` annotation. For example, we could replace:

```
@Entity(indices={@Index("postalCode")})
class Customer {
  @PrimaryKey
  public final String id;

  public final String postalCode;
  public final String displayName;

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

with:

```
@Entity
class Customer {
  @PrimaryKey
  public final String id;

  @ColumnInfo(index=true)
  public final String postalCode;
  public final String displayName;
```

```
  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

and have the same result.

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 field (e.g., `int` or `Integer` turn into `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.

DAOs and Queries

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

Those do not have to be especially complicated, as we saw with the `TripStore`:

```
package com.commonsware.android.room;
```

```
import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

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` method's arguments and return type.

Adding Parameters

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

```
  @Query("SELECT * FROM thingy WHERE id=:id AND version_code=:versionCode")
  VersionedThingy findById(String id, int versionCode);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Bear in mind that the rest of the SQL statement is based on the *table*, not the *entity*.

Table names and column names will either be the code-generated names or your overridden names (via `tableName` and `@ColumnInfo`).

WHERE Clause

Principally, your method arguments will be injected into your `WHERE` clause, such as in the above examples.

Note that Room has special support for `IN` in a `WHERE` clause. So, while this works for a single `postalCode`:

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes)")
List<Customer> findByPostalCodes(String postalCodes);
```

...you can also do:

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes)")
List<Customer> findByPostalCodes(List<String> postalCodes);
```

...or even:

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes)")
List<Customer> findByPostalCodes(String... postalCodes);
```

Room will convert the collection argument into a comma-delimited list for use with the SQL query.

Other Clauses

If SQLite allows `?` placeholders, Room should allow method arguments to be used instead.

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

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes) LIMIT :max")
List<Customer> findByPostalCodes(int max, String... postalCodes);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Here, because Java needs the varargs to be the last parameter, we need to have `max` first.

What You Can Return

We have seen that a `@Query` can return a single entity (e.g., `findById()` returning a single `Trip`) or a collection of entity (e.g., `selectAll()` returning a `List` of `Trip` entities).

While those are simple, Room offers a fair bit more flexibility than that. In particular, not only does Room support reactive return values, but we can return objects that are not actually entities.

Specific Return Types

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

Beyond that, a `@Query` method can return:

- A `Flowable` or `Publisher` from RxJava2, a popular framework for reactive programming
- A `LiveData` object

NOTE: The upcoming Room 2.1.0 release will support other RxJava types, such as `Single`.

We will explore what a `LiveData` object is later in this book.

Breadth of Results

For small entities, like `Trip`, usually we will retrieve all columns in the query. However, the real rule is: the core return object of the `@Query` method 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 methods 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 method arguments, and the method return type, Room is perfectly happy.

So, for example, suppose that `Customer` not only tracks an `id` and a `postalCode`, but also has *many* other fields, including a `displayName`:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
import java.util.UUID;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String postalCode;
  public final String displayName;

  @Ignore
  Customer(String postalCode, String displayName) {
    this(UUID.randomUUID().toString(), postalCode, displayName);
  }

  Customer(String id, String postalCode, String displayName) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

Perhaps to show a list of customers, we need the `displayName` (to show in the list) and the `id` (to know which specific customer this is). But we do not need the `postalCode` or the rest of the fields in the `Customer` class.

We can still return a `Customer`:

```
@Query("SELECT id, displayName FROM Customer WHERE postalCode IN (:postalCodes) LIMIT
:max")
List<Customer> findByPostalCodes(List<String> postalCodes, int max);
```

The code that Room generates will simply fill in `null` for the `postalCode`, since that was not one of the returned columns. However, then it is not obvious whether a given instance of `Customer` is completely filled in from data in the table (and it is genuinely missing its `postalCode`) or whether this is a partially-populated `Customer` object.

However, we could also define a dedicated `CustomerDisplayTuple` class:

```
package com.commonsware.android.room.dao;

public class CustomerDisplayTuple {
  public final String id;
  public final String displayName;

  public CustomerDisplayTuple(String id, String displayName) {
    this.id=id;
    this.displayName=displayName;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/CustomerDisplayTuple.java)

Then, we can return a `List` of `CustomerDisplayTuple` from our DAO:

```
  @Query("SELECT id, displayName FROM Customer WHERE postalCode IN (:postalCodes) LIMIT :max")
  List<CustomerDisplayTuple> loadDisplayTuplesByPostalCodes(int max, String... postalCodes);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

This way, we get our subset of data, and we know by class whether we have the full `Customer` or just the subset for display purposes.

Note that `@ColumnInfo` annotations can be used on any class, not just entities. In particular, if you use `@ColumnInfo` on a field in an entity, you will need the same

`@ColumnInfo` on any "tuple"-style classes that represent subsets of data that include that same field.

Aggregate Functions

A `@Query` can also return an `int`, for simple aggregate functions:

```
  @Query("SELECT COUNT(*) FROM Customer")
  int getCustomerCount();
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

If you wish to compute several aggregate functions, create a "tuple"-style class to hold the values:

```
package com.commonsware.android.room.dao;

public class CustomerStats {
  public final int count;
  public final String max;

  public CustomerStats(int count, String max) {
    this.count=count;
    this.max=max;
  }
}
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/CustomerStats.java)

...and use `AS` to name the aggregate function "columns" to match the tuple:

```
  @Query("SELECT COUNT(*) AS count, MAX(postalCode) AS max FROM Customer")
  CustomerStats getCustomerStats();
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

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()` method, ones that you can map to `@Query` methods 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() method 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, and it works on all versions of Room. The downside is that you wind up with a Cursor, which is less convenient than the model objects that you get back from @Query methods on your @Dao.

@RawQuery

Room 1.1.0 added a new option for this: @RawQuery. Like @Query, this is an annotation that you can add to a method on your @Dao. And, like @Query, you can have that method 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 method:

```
@RawQuery
abstract List<Foo> _findMeSomething(SupportSQLiteQuery query);
```

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

```
@RawQuery
abstract List<Foo> _findMeSomething(SupportSQLiteQuery query);

List<Foo> findMeSomething(String value) {
  return _findMeSomething(new SimpleSQLiteQuery("SELECT some, columns FROM your_table
WHERE something=?",
```

```
    new Object[] {value}));
}
```

Here, `findMeSomething()` looks like a regular query method on the `@Dao`. Instead, it creates a `SimpleSQLiteQuery` for a SQL statement and a supplied `value`, then uses `_findMeSomething()` to execute that query and return a `List` of `Foo` objects.

In this particular case, `findMeSomething()` could have been written using a regular `@Query` annotation, as the SQL statement is known at compile time... assuming that `your_table` is associated with an `@Entity`. One scenario where `@RawQuery` comes into play is when you want to query a table using Room where the table is *not* associated with an `@Entity`. We will see an example of that much later in the book, when we examine full-text searching with Room.

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 methods on `TripStore`:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
  @Query("SELECT * FROM trips ORDER BY title")
  List<Trip> selectAll();

  @Query("SELECT * FROM trips WHERE id=:id")
  Trip findById(String id);

  @Insert
  void insert(Trip... trips);

  @Update
  void update(Trip... trips);

  @Delete
```

```
  void delete(Trip... trips);
}
```

(from Trips/RoomBasics/app/src/main/java/com/commonsware/android/room/TripStore.java)

Generally speaking, these scenarios are simpler than `@Query`. The `@Insert`, `@Update`, and `@Delete` set up simple methods for inserting, updating, or deleting entities passed to their methods... 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. `TripStore` uses varargs, so we can pass zero, one, or several `Trip` objects, though passing zero objects would be a waste of time.

However, in addition to varargs, you can have these methods accept:

- A single entity
- Individual entities as separate parameters (`void insert(Trip trip1, Trip trip2)`)
- A `List` of entities

Return Values

Frequently, you just have these methods return `void`.

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` method 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` method accepting multiple entities, you can have it return an array of `long` objects or a `List` of `Long` objects, 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`	Cancel this statement but preserve 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 wille 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 method 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 Customer")
void nukeCustomersFromOrbit();
```

...or:

```
  @Query("DELETE FROM Customer WHERE id IN (:ids)")
  int nukeCertainCustomersFromOrbit(String... ids);
```

(from General/RoomDao/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

...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 methods.

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 methods. For example, persisting an invoice might involve inserting an `Invoice` and all of its `InvoiceLineItem` objects, and that might require more than one DAO method to achieve.

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

Using @Transaction

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

There are two places to apply `@Transaction`: custom methods on an `abstract` DAO

class, or on `@Query` methods.

Custom Methods

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

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

```
@Dao
public abstract class InvoiceStore {
  @Insert
  public abstract void _insert(Invoice invoice);

  @Insert
  public abstract void insert(List<InvoiceLineItem> lineItems);

  @Transaction
  public void insert(Invoice invoice) {
    _insert(invoice);
    insert(invoice.getLineItems());
  }
}
```

Here, we still use an `insert()` method 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 you will need to use an `abstract` class, not an `interface`, as an `interface` cannot have arbitrary method implementations in them.

On @Query Methods

It may seem odd to have to specifically request a transaction on a `@Query`-annotated method. 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()` methods 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.

Threads and Room

`@Insert`, `@Update`, and `@Delete`-annotated methods are synchronous, performing their work on the current thread. Hence, they should only be called from a background thread.

`@Query` methods that return entities, `int`, tuples, etc. directly also are synchronous. However, `@Query` methods that return an RxJava type (e.g., `Flowable`) or a `LiveData` are *not* synchronous. Instead, the real work will be performed on a background

thread.

As noted earlier, we will explore what this "`LiveData`" is later in the book. For now, take it on faith that it is another piece of the Android Architecture Components, one that offers an alternative to RxJava for reactive programming.

Room and Custom Types

So far, all of our fields have been basic primitives (`int`, `float`, etc.) 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 field 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 field 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` or `Calendar` object? Do we want to store that as a milliseconds-since-the-Unix-epoch value as a Java `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 field? 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 methods, annotated with `@TypeConverter`, that map

the type for a single database column to a type for a Java field. So, for example, we can:

- Map a `Date` field to a `Long`, which can go in a SQLite `INTEGER` column
- Map a `Location` field 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
- And so forth

However, type converters offer only a 1:1 conversion: a single Java field to and from a single SQLite column. If you have a single Java field 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 Java class somewhere. The name, package, superclass, etc. do not matter.

Next, for each type to be converted, create two `public static` methods that convert from one type to the other. So for example, you would have one `public static` method that takes a `Date` and returns a `Long` (e.g., returning the milliseconds-since-the-Unix-epoch value), and a counterpart method that takes a `Long` and returns a `Date`. If the converter method 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 method, supplied as input to the other converter method, returns the original value.

Then, each of those methods get the `@TypeConverter` annotation. The method 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:

@TypeConverters Location	Affected Areas
Entity class	all fields in the entity
Entity field	that one field in the entity
DAO class	all methods in the DAO
DAO method	that one method in the DAO, for all parameters
DAO method parameter	that one parameter on that one method
POJO	all fields on the POJO

The General/RoomTypes sample project illustrates the use of type converters. As with the RoomDao project from the preceding chapter, this project contains a single library module with an associated instrumentation test case. In fact, it is a clone of the RoomDao project, just with some type converters.

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). Date has a getTime() method that returns this value.

So, the project has a TypeTransmogrifiers class that contains two methods, each annotated with @TypeConverter, for converting Date to and from a Long:

```
@TypeConverter
public static Long fromDate(Date date) {
  if (date==null) {
    return(null);
  }

  return(date.getTime());
}

@TypeConverter
public static Date toDate(Long millisSinceEpoch) {
  if (millisSinceEpoch==null) {
    return(null);
  }
```

```
    return(new Date(millisSinceEpoch));
  }
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/TypeTransmogrifier.java)

`StuffDatabase` then has the `@TypeConverters` annotation, listing `TypeTransmogrifier` as the one class that has type conversion methods:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.arch.persistence.room.TypeConverters;
import android.content.Context;

@Database(
  entities={Customer.class, VersionedThingy.class},
  version=1
)
@TypeConverters({TypeTransmogrifier.class})
abstract class StuffDatabase extends RoomDatabase {
  abstract StuffStore stuffStore();

  private static final String DB_NAME="stuff.db";
  private static volatile StuffDatabase INSTANCE=null;

  synchronized static StuffDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static StuffDatabase create(Context ctxt, boolean memoryOnly) {
    RoomDatabase.Builder<StuffDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        StuffDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), StuffDatabase.class,
        DB_NAME);
    }

    return(b.build());
```

```
  }
}
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/StuffDatabase.java)

Now, classes like `Customer` can use `Date` fields, which will be stored in `INTEGER` columns in the database.

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, PRIMARY KEY(`id`))
```

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. If we use `Locale.US` formatting for the latitude and longitude, so that the decimal place is denoted by a `.`, we could use a two-element comma-separated values list for the string.

That is what these two type converter methods on `TypeTransmogrifiers` do:

```
  @TypeConverter
  public static String fromLocation(Location location) {
    if (location==null) {
      return(null);
    }

    return(String.format(Locale.US, "%f,%f", location.getLatitude(),
      location.getLongitude()));
  }

  @TypeConverter
  public static Location toLocation(String latlon) {
    if (latlon==null) {
      return(null);
    }

    String[] pieces=latlon.split(",");
    Location result=new Location("");

    result.setLatitude(Double.parseDouble(pieces[0]));
    result.setLongitude(Double.parseDouble(pieces[1]));

    return(result);
```

```
  }
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/TypeTransmogrifier.java)

Since `TypeTransmogrifiers` is registered on the `StuffDatabase`, a `Customer` could have a `Location` field, which would be mapped to a `TEXT` column in the database:

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, officeLocation TEXT, PRIMARY KEY(`id`))
```

However, the downside of using this approach is that we cannot readily search based on location. If your location data is not a searchable field, 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 POJO in Java.

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
public static String fromStringSet(Set<String> strings) {
  if (strings==null) {
    return(null);
  }

  StringWriter result=new StringWriter();
  JsonWriter json=new JsonWriter(result);
```

```
    try {
      json.beginArray();

      for (String s : strings) {
        json.value(s);
      }

      json.endArray();
      json.close();
    }
    catch (IOException e) {
      Log.e(TAG, "Exception creating JSON", e);
    }

    return(result.toString());
  }

  @TypeConverter
  public static Set<String> toStringSet(String strings) {
    if (strings==null) {
      return(null);
    }

    StringReader reader=new StringReader(strings);
    JsonReader json=new JsonReader(reader);
    HashSet<String> result=new HashSet<>();

    try {
      json.beginArray();

      while (json.hasNext()) {
        result.add(json.nextString());
      }

      json.endArray();
    }
    catch (IOException e) {
      Log.e(TAG, "Exception parsing JSON", e);
    }

    return(result);
  }
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/TypeTransmogrifier.java)

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).

Note that type converter methods cannot throw checked exceptions, as the Room code generator does not wrap type converter calls in a `try/catch` block. Here, the `IOExceptions` should never happen, since we are working with strings, not files or other types of streams. In other cases, though, you may need to wrap the checked exception in some form of `RuntimeException` and throw that, to trigger your app's unhandled-exception logic, as it is unlikely that you can recover from within a type converter method.

Given these type conversion methods, we can now use a `Set` of `String` values in `Customer`:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.location.Location;
import android.support.annotation.NonNull;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String postalCode;
  public final String displayName;
  public final Date creationDate;
  public final Location officeLocation;
  public final Set<String> tags;

  @Ignore
  Customer(String postalCode, String displayName, Location officeLocation,
           Set<String> tags) {
    this(UUID.randomUUID().toString(), postalCode, displayName, new Date(),
      officeLocation, tags);
  }

  Customer(String id, String postalCode, String displayName, Date creationDate,
```

```
            Location officeLocation, Set<String> tags) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
    this.creationDate=creationDate;
    this.officeLocation=officeLocation;
    this.tags=tags;
  }
}
```

(from General/RoomTypes/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

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

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, officeLocation TEXT, tags TEXT, PRIMARY KEY(`id`))
```

Embedded Types

With type converters, we are teaching Room how to deal with custom types, but we are limited to mapping from one field to one column. That field 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 field?

In that case, we can use the `@Embedded` annotation on some POJO, then use that POJO as a type in an entity.

Example: Locations

For example, as was noted earlier in this chapter, cramming a location into a single `TEXT` field works, but we cannot readily query on the resulting field. 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 field.

With `@Embedded`, we can, as we can see in the `General/RoomEmbedded` sample project. This is a clone of the `RoomTypes` project from earlier in this chapter, where we have changed `Customer` to have the `officeLocation` be represented by a `LocationColumns` POJO:

```
package com.commonsware.android.room.dao;
```

```
import android.arch.persistence.room.Embedded;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.location.Location;
import android.support.annotation.NonNull;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String postalCode;
  public final String displayName;
  public final Date creationDate;

  @Embedded
  public final LocationColumns officeLocation;

  public final Set<String> tags;

  @Ignore
  Customer(String postalCode, String displayName, LocationColumns officeLocation,
           Set<String> tags) {
    this(UUID.randomUUID().toString(), postalCode, displayName, new Date(),
      officeLocation, tags);
  }

  Customer(String id, String postalCode, String displayName, Date creationDate,
           LocationColumns officeLocation, Set<String> tags) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
    this.creationDate=creationDate;
    this.officeLocation=officeLocation;
    this.tags=tags;
  }
}
```

(from General/RoomEmbedded/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

The `@Embedded` annotation tells Room to combine the columns from the annotated type into the table for this entity. In this case, `LocationColumns` has two fields, for latitude and longitude:

```
package com.commonsware.android.room.dao;

public class LocationColumns {
  public final double latitude;
  public final double longitude;
```

```
  public LocationColumns(double latitude, double longitude) {
    this.latitude=latitude;
    this.longitude=longitude;
  }
}
```

(from General/RoomEmbedded/stuff/src/main/java/com/commonsware/android/room/dao/LocationColumns.java)

`LocationColumns` itself is a POJO, not an entity, though you can use `@ColumnInfo` annotations if needed to rename the columns associated with the POJO's fields.

Now, Room will use individual `REAL` columns for our latitude and longitude:

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, tags TEXT, latitude REAL, longitude REAL, PRIMARY KEY(id))
```

...and we can query on those:

```
  @Query("SELECT * FROM Customer WHERE ABS(latitude-:lat)<.000001 AND ABS(longitude-:lon)<.000001")
  List<Customer> findCustomersAt(double lat, double lon);
```

(from General/RoomEmbedded/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

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` POJO's field names, perhaps modified by `@ColumnInfo` annotations on the POJO. In this case, though, if we have two `LocationColumns` fields in the `Customer` entity, 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_")
public final LocationColumns officeLocation;
```

The columns for that POJO will have the `prefix` added:

```
CREATE TABLE IF NOT EXISTS Customer (id TEXT, postalCode TEXT, displayName TEXT,
creationDate INTEGER, tags TEXT, office_latitude REAL, office_longitude REAL, PRIMARY
KEY(id))
```

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` method that references those names, so that you use the appropriate prefix.

Updating the Trip Sample

Back in the chapter on Room basics, we saw some rudimentary code to track upcoming travel. The `Trips/RoomConverters` sample project extends that code with four new fields on `Trip`:

- `priority`, representing how important the trip is to the user
- `startTime`, indicating when the trip is to begin
- `creationTime`, indicating when the `Trip` was first created... somewhere
- `updateTime`, indicating when the `Trip` was last changed... somewhere

Those latter two are largely ignored for the moment, though they will become more important later in the book.

The latter three are all `Date` fields, and so we need to have some code to support getting them into and out of our table. So, this project has a `TypeTransmogrifier` class, akin to the ones seen above, but right now only with the `Date` converters:

```
package com.commonsware.android.room;

import android.arch.persistence.room.TypeConverter;
import java.util.Date;

public class TypeTransmogrifier {
  @TypeConverter
  public static Long fromDate(Date date) {
    if (date==null) {
      return(null);
    }

    return(date.getTime());
  }

  @TypeConverter
  public static Date toDate(Long millisSinceEpoch) {
    if (millisSinceEpoch==null) {
```

```
      return(null);
    }

    return(new Date(millisSinceEpoch));
  }
}
```

(from Trips/RoomConverters/app/src/main/java/com/commonsware/android/room/TypeTransmogrifier.java)

`priority`, though, is an `enum`, as there is a list of valid values:

```
package com.commonsware.android.room;

import android.arch.persistence.room.TypeConverter;

enum Priority {
  LOW(0), MEDIUM(1), HIGH(2), OMG(3);

  private final int level;

  @TypeConverter
  public static Priority fromLevel(Integer level) {
    for (Priority p : values()) {
      if (p.level==level) {
        return(p);
      }
    }

    return(null);
  }

  @TypeConverter
  public static Integer fromPriority(Priority p) {
    return(p.level);
  }

  Priority(int level) {
    this.level=level;
  }
}
```

(from Trips/RoomConverters/app/src/main/java/com/commonsware/android/room/Priority.java)

Here, we implement the `@TypeConverter` methods right on `Priority`, as there is little value in having them elsewhere. Note that the `enum` assigns explicit numeric values to the priorities (`level`). That way, we are in control over the mapping between `Priority` values and their representation in the database.

Rather than apply these type converters on the `TripDatabase` (though we could), we instead apply them on the `Trip` model:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.TypeConverters;
import android.support.annotation.NonNull;
import java.util.Date;
import java.util.UUID;

@Entity(tableName = "trips")
@TypeConverters({TypeTransmogrifier.class})
class Trip {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String title;
  public final int duration;

  @TypeConverters({Priority.class})
  public final Priority priority;

  public final Date startTime;
  public final Date creationTime;
  public final Date updateTime;

  @Ignore
  Trip(String title, int duration, Priority priority, Date startTime) {
    this(UUID.randomUUID().toString(), title, duration, priority, startTime,
      null, null);
  }

  Trip(String id, String title, int duration, Priority priority,
       Date startTime, Date creationTime, Date updateTime) {
    this.id=id;
    this.title=title;
    this.duration=duration;
    this.priority=priority;
    this.startTime=startTime;
    this.creationTime=creationTime;
    this.updateTime=updateTime;
  }
```

```
  @Override
  public String toString() {
    return(title);
  }
}
```

(from Trips/RoomConverters/app/src/main/java/com/commonsware/android/room/Trip.java)

The `Priority` type converters are applied specific to the `priority` field, as this specific conversion is only needed here. The `TypeTransmogrifier` is registered on the `Trip` class, as there are multiple `Date` fields.

Room and Relations

SQLite is a relational database. At some point, Room should support relations. Right?

Right?!?

Well, actually, the story is a bit more complicated than that. Yes, 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".

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:

```
@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. Disk I/O — such as queries on complex structures – can easily take dozens or hundreds of milliseconds, particularly on older or low-end devices. As a result, we freeze our UI while that disk I/O is occurring, possibly resulting in visual "jank" for the user. Our objective is to move as much disk I/O as possible off the main application thread.

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.

No Direct Entity References

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

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.

Foreign Keys

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 does very little else with these keys.

Cascades on Updates and Deletes

Part of what you can place on a `@ForeignKey` annotation are `onUpdate` and `onDelete`

properties. 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 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.

Cascades on… Retrievals?

You cannot have an entity automatically retrieve related objects via a `@Query`.

You *can* have an arbitrary POJO automatically retrieve related objects via a `@Query`, by means of a `@Relation` annotation.

This seeming inconsistency will be explored later in this chapter.

Plans for Trips

Let's explore how `@ForeignKey` works by adding some more entities to the trip-tracking code, as seen in the `Trips/RoomRelations` sample project.

The Domain Model

In the beginning, we had just the `Trip` entity. However, a trip is made up of lots of pieces, so in this sample, we add two more: flights and lodgings. Not surprisingly, these come in the form of `Flight` and `Lodging` entity classes. A `Trip` can have zero or more related `Flight` instances and zero or more related `Lodging` instances.

However, many of the pieces of data that we need to track for these things – title, duration, start time, etc. — are in common. So, we will pull those things into an `abstract` base class named `Plan`, from which `Trip`, `Flight`, and `Lodging` will all inherit.

The New Entities

As a result, `Plan` itself is pretty much what `Trip` used to be:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.TypeConverters;
import android.support.annotation.NonNull;
import java.util.Date;
import java.util.UUID;

abstract class Plan {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String title;
  public final int duration;

  @TypeConverters({Priority.class})
  public final Priority priority;

  public final Date startTime;
  public final Date creationTime;
  public final Date updateTime;

  @Ignore
  Plan(String title, int duration, Priority priority, Date startTime) {
    this(UUID.randomUUID().toString(), title, duration, priority, startTime,
      null, null);
  }

  Plan(String id, String title, int duration, Priority priority,
       Date startTime, Date creationTime, Date updateTime) {
    this.id=id;
    this.title=title;
    this.duration=duration;
    this.priority=priority;
    this.startTime=startTime;
```

```
    this.creationTime=creationTime;
    this.updateTime=updateTime;
  }

  @Override
  public String toString() {
    return(title);
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Plan.java)

Note that while we have the `Priority` `TypeConverter` registered for the `Priority` field, we do not have the `TypeTransmogrifier` registered on the `Plan` class, the way we had it for `Trip`. That is due to a limitation in Room, whereby class-level `@TypeConverters` annotations are not inherited, though field-level ones are.

Instead, the `TypeTransmogrifier` `@TypeConverters` annotation appears on our rump `Trip` class:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import java.util.UUID;

@Entity(tableName = "trips")
@TypeConverters({TypeTransmogrifier.class})
class Trip extends Plan {
  @Ignore
  Trip(String title, int duration, Priority priority, Date startTime) {
    super(title, duration, priority, startTime);
  }

  Trip(String id, String title, int duration,
       Priority priority, Date startTime, Date creationTime,
       Date updateTime) {
    super(id, title, duration, priority, startTime, creationTime, updateTime);
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Trip.java)

The relations that we are setting up from `Trip` to `Flight` and `Lodging` are 1:N

relations. As such, the parent (`Trip`) does not need any foreign keys. Those are held by the children of the relation... such as `Lodging`:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="lodgings",
  foreignKeys=@ForeignKey(
    entity=Trip.class,
    parentColumns="id",
    childColumns="tripId",
    onDelete=CASCADE),
  indices=@Index("tripId"))
@TypeConverters({TypeTransmogrifier.class})
class Lodging extends Plan {
  public final String address;
  public final String tripId;

  @Ignore
  Lodging(String title, int duration, Priority priority, Date startTime,
          String address, String tripId) {
    super(title, duration, priority, startTime);
    this.address=address;
    this.tripId=tripId;
  }

  Lodging(String id, String title, int duration,
          Priority priority, Date startTime, Date creationTime,
          Date updateTime, String address, String tripId) {
    super(id, title, duration, priority, startTime, creationTime, updateTime);
    this.address=address;
    this.tripId=tripId;
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Lodging.java)

Here, `Lodging` also extends from `Plan`, adding two fields, one to track the address of the hotel (or whatever) and the `tripId` of the `Trip` that contains this `Lodging`. That

`tripId` field is then referenced in the `@ForeignKey` annotation,which:

- Sets up the relation as being with `Trip` (`entity=Trip.class`)
- Ties the `id` column on `Trip` (`parentColumns="id"`) to the `tripId` on `Lodging` (`childColumns="tripId"`)
- Indicates that if the `Trip` is deleted, associated `Lodging` instances should also be deleted (`onDelete=CASCADE`)

`Lodging` also sets up an index on `tripId` (`indices=@Index("tripId")`). Querying on `tripId` will be fairly common, as we look up the `Lodging` instances associated with a given `Trip`. Hence, typically you will want to set up an index on your foreign keys. Room will even warn you about this, if you examine the Gradle Console output from a build.

`Flight` works similarly:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.TypeConverters;
import java.util.Date;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="flights",
  foreignKeys=@ForeignKey(
    entity=Trip.class,
    parentColumns="id",
    childColumns="tripId",
    onDelete=CASCADE),
  indices=@Index("tripId"))
@TypeConverters({TypeTransmogrifier.class})
class Flight extends Plan {
  public final String departingAirport;
  public final String arrivingAirport;
  public final String airlineCode;
  public final String flightNumber;
  public final String seatNumber;
  public final String tripId;

  @Ignore
  Flight(String title, int duration, Priority priority, Date startTime,
```

```
        String departingAirport, String arrivingAirport, String airlineCode,
        String flightNumber, String seatNumber, String tripId) {
    super(title, duration, priority, startTime);
    this.departingAirport=departingAirport;
    this.arrivingAirport=arrivingAirport;
    this.airlineCode=airlineCode;
    this.flightNumber=flightNumber;
    this.seatNumber=seatNumber;
    this.tripId=tripId;
  }

  Flight(String id, String title, int duration,
        Priority priority, Date startTime, Date creationTime,
        Date updateTime, String departingAirport, String arrivingAirport,
        String airlineCode, String flightNumber, String seatNumber,
        String tripId) {
    super(id, title, duration, priority, startTime, creationTime, updateTime);
    this.departingAirport=departingAirport;
    this.arrivingAirport=arrivingAirport;
    this.airlineCode=airlineCode;
    this.flightNumber=flightNumber;
    this.seatNumber=seatNumber;
    this.tripId=tripId;
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/Flight.java)

The Updated DAO and Database

Since we added new entities, `TripDatabase` needs to know about them, via the `entities` property on the `@Database` annotation:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;

@Database(
  entities={Trip.class, Lodging.class, Flight.class},
  version=2
)
abstract class TripDatabase extends RoomDatabase {
  abstract TripStore tripStore();
```

```
  private static final String DB_NAME="trips.db";
  private static volatile TripDatabase INSTANCE=null;

  synchronized static TripDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static TripDatabase create(Context ctxt, boolean memoryOnly) {
    RoomDatabase.Builder<TripDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        TripDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
        DB_NAME);
    }

    return(b.build());
  }
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

Note that now we are on `version=2`. Ideally, this sort of change would involve updating an existing database in-place, so as not to disturb any existing data. Room calls these "migrations", and they are covered in an upcoming chapter.

`TripStore`, our DAO, now needs methods for `Lodging` and `Flight` as well:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.OnConflictStrategy;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import java.util.List;

@Dao
interface TripStore {
```

```
/*
  Trip
 */

@Query("SELECT * FROM trips ORDER BY title")
List<Trip> selectAllTrips();

@Query("SELECT * FROM trips WHERE id=:id")
Trip findTripById(String id);

@Insert
void insert(Trip... trips);

@Update
void update(Trip... trips);

@Delete
void delete(Trip... trips);

/*
  Lodging
 */

@Query("SELECT * FROM lodgings WHERE tripId=:tripId")
List<Lodging> findLodgingsForTrip(String tripId);

@Insert
void insert(Lodging... lodgings);

@Update
void update(Lodging... lodgings);

@Delete
void delete(Lodging... lodgings);

/*
  Flight
 */

@Query("SELECT * FROM flights WHERE tripId=:tripId")
List<Flight> findFlightsForTrip(String tripId);

@Insert
void insert(Flight... flights);

@Update
void update(Flight... flights);
```

```
  @Delete
  void delete(Flight... flights);
}
```

(from Trips/RoomRelations/app/src/main/java/com/commonsware/android/room/TripStore.java)

The `Lodging` and `Flight` `@Query` methods retrieve only those for a particular `Trip`, based on the ID. There is nothing stopping us from having other `@Query` methods (e.g., searching across all `Lodging`, regardless of `Trip`), but these will suffice for now.

We could elect to have separate DAO classes for each entity, or have nested `@Dao`-annotated classes inside the entity for these sorts of methods. In those cases, `TripDatabase` would have to be augmented with additional `abstract` methods to return instances of those classes, mirroring the existing `tripStore()` method.

Self-Referential Relations for Tree Structures

With care, you can use Room for self-referential relations: an entity having a foreign key back to itself. This is most commonly seen in tree structures:

- Categories having sub-categories
- Folders having folders and items
- And so on

The `General/RoomTree` sample project demonstrates the first of those examples: a `Category` entity that has an optional parent `Category`:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
import java.util.UUID;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="categories",
  foreignKeys=@ForeignKey(
    entity=Category.class,
    parentColumns="id",
    childColumns="parentId",
```

```
    onDelete=CASCADE),
  indices=@Index(value="parentId"))
class Category {
  @PrimaryKey
  @NonNull
  public final String id;
  public final String title;
  public final String parentId;

  @Ignore
  Category(String title) {
    this(title, null);
  }

  @Ignore
  public Category(String title, String parentId) {
    this(UUID.randomUUID().toString(), title, parentId);
  }

  public Category(@NonNull String id, String title, String parentId) {
    this.id=id;
    this.title=title;
    this.parentId=parentId;
  }
}
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/Category.java)

Here, `Category` has a `@ForeignKey` that points back to `Category` as the `entity`, with a `parentId` column holding the `id` of the parent `Category`. `onDelete` is set to `CASCADE`, so that when a parent `Category` is deleted, its children are deleted as well.

Now we can have DAO methods that work with the `Category` tree:

```
  @Query("SELECT * FROM categories")
  List<Category> selectAllCategories();

  @Query("SELECT * FROM categories WHERE parentId IS NULL")
  Category findRootCategory();

  @Query("SELECT * FROM categories WHERE parentId=:parentId")
  List<Category> findChildCategories(String parentId);

  @Insert
  void insert(Category... categories);

  @Delete
```

```
  void delete(Category... categories);
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Using @Relation

If you have a POJO class — one that does not *directly* have the `@Entity` annotation — you can use `@Relation` to automatically retrieve entities related to... something in the POJO.

For example, in other Android ORMs, one might expect that `Category` would have methods, fields, or something to get at the parent `Category` (where there is one) or the child `Category` instances (where there are some). However, that is not supported by Room and `@Entity`, but it is supported by separate POJO classes.

To that end, we can set up a `CategoryTuple`:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Relation;
import java.util.List;

public class CategoryTuple {
  public final String id;
  public final String title;
  public final String parentId;

  public CategoryTuple(String id, String title, String parentId) {
    this.id=id;
    this.title=title;
    this.parentId=parentId;
  }

  @Relation(parentColumn="id", entityColumn="parentId")
  public List<Category> children;

  @Relation(parentColumn="parentId", entityColumn="id")
  public List<Category> parents;
}
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/CategoryTuple.java)

Here, we have two `@Relation` annotations. These go on *fields*, not methods, and they indicate fields that Room should fill in when a `@Query` returns instances of this POJO. The field type needs to be a `List` or `Set` of the related *entity*, not the POJO.

Hence, `children` and `parents` are lists of `Category` instances, not `CategoryTuple`.

The two required properties on `@Relation` are `parentColumn` and `entityColumn`. `entityColumn` is the name of a column in the entity's table; `parentColumn` is the name of a *field* in the POJO representing the parent entity. In this case, the entity for both is `Category`, as we are working with a self-referential relation. In the generated code, Room is going to run a query that finds all objects whose `entityColumn` has the value pulled from this POJO's `parentColumn` field. More specifically:

- For the `children` field, Room will query the `categories` table to return all rows where the `parentId` column equals the `id` of this `CategoryTuple`
- For the `parent` field, Room will query the `categories` table to return all rows where the `id` column equals the `parentId` of this `CategoryTuple`

For a 1:N relation, Room's restriction on `@Relation` data types (must be `List` or `Set`) means that both the 1 side and the N side get represented by collection fields... even though one should only ever have at most one element.

If there are no matching entities (e.g., no parent for the root `Category`, no children for a leaf `Category`), the resulting field is either `null` or an empty collection.

But now, our DAO methods will not only set up the POJOs but all entities that are called for by the `@Relation` fields:

```
@Transaction
@Query("SELECT * FROM categories WHERE parentId IS NULL")
CategoryTuple findRootCategoryTuple();

@Transaction
@Query("SELECT * FROM categories WHERE parentId=:parentId")
List<CategoryTuple> findChildCategoryTuples(String parentId);
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

However, this involved a lot of copying. `CategoryTuple` has the same fields as `Category`. It would not *have* to have all of those fields, of course, as a POJO need not have fields for all columns in the table. But, still, it seems to be a bit wasteful.

Another related approach is to create a "POJO" *subclass* of the entity... such as this `CategoryShadow`:

```
package com.commonsware.android.room.dao;
```

```
import android.arch.persistence.room.Relation;
import java.util.List;

public class CategoryShadow extends Category {
  public CategoryShadow(String id, String title, String parentId) {
    super(id, title, parentId);
  }

  @Relation(parentColumn="id", entityColumn="parentId")
  public List<Category> children;
}
```

(from General/RoomTree/stuff/src/main/java/com/commonsware/android/room/dao/CategoryShadow.java)

Even though `CategoryShadow` inherits from `Category`, and even though `Category` is an entity, Room treats `CategoryShadow` as a POJO, and we can have `@Relation` fields, such as the `children` one shown. If you need most or all of the fields from the entity, this subclass approach involves less code duplication than does the standalone-POJO approach.

@Relation and @Query

If you use `@Relation`, it is a good idea to add the `@Transaction` annotation to any related `@Query`. That ensures that the initial query that populates the POJO, plus the query (or queries) necessary to resolve the `@Relation`, are all performed inside a transaction and therefore will have consistency.

By default, `@Transaction` logic is *not* applied to `@Query` methods returning POJOs with `@Relation`, and it is possible that you would wind up with inconsistent results, if the database was modified while the `@Query` was being processed. Unfortunately, this has been deemed as "working as intended".

This is why the `@Query` methods on `StuffStore` that return `CategoryTuple` or `CategoryShadow` also have the `@Transaction` annotation.

Representing No Relation

While much of this book will use `UUID` values for primary keys, plenty of other Room examples will use `int`, particularly with `autoGenerate` set to `true`, to have SQLite generate the keys.

However, this does not work well if those keys will be used as foreign key values, in

cases where there may be no value for the relation.

For example, `Category` uses `String` for its `id` (created from a `UUID`), and we represented a root category by means of having `null` for its `parentId` value. That works because `String` fields can be `null`.

If, however, we used `int`, we have no way of representing the no-relation scenario. You cannot assign `null` to an `int` field in Java.

Hence, if you want to support the no-relation scenario, your foreign key field needs to allow for `null` values. If you want to use auto-generated SQLite identifiers, use `Integer`, not `int`.

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.

Part of what we get with Room is a series of classes and interfaces in the `android.support.persistence.db` package. These classes come from a separate artifact (`android.arch.persistence.room:support-db`) and represent an abstraction for SQLite-style database access. This book will refer to this as "the support database API".

We *also* get implementations of that abstraction, in the form of the "framework" classes (from `android.arch.persistence.room:support-db-impl`). Those classes use the Android standard SQLite environment. Room's artifacts pull in these support artifacts by default, and 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.

CWAC-SafeRoom

One of the best-known alternative SQLite implementations for Android is Zetitec's SQLCipher for Android, which offers transparent encryption of database contents. However, Room knows nothing about SQLCipher for Android... which is why the author of this book wrote CWAC-SafeRoom. The bulk of CWAC-SafeRoom is an implementation of the support database API that completely replaces the use of the framework SQLite with SQLCipher for Android.

AssetRoom

Sometimes, you may want to package a database with the app, either as starter data or as a read-only data source. Jeff Gilfelt's `SQLiteAssetHelper` can handle this for you, but Room does not know about it.

The `General/AssetRoom` sample app profiled in another chapter contains a partial implementation of the support database API. Mostly, it is a pass-through to the stock implementation of that API that uses the framework's copy of SQLite. However, it uses a modified version of `SQLiteAssetHelper` to handle the timely unpacking of a database stored in `assets/`.

Other ORMs

Both of those use cases offer other implementations of the support database API. The idea is that not only can Room use that API, but so could other similar object-relational mapping (ORM) libraries. If those libraries write to the support database API and allow pluggable implementations the way that Room does, then solutions like CWAC-SafeRoom can work for those libraries in the same way that it works for Room itself.

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 (hopefully) work with it.

However, 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 create and manage tables that Room will not use
- 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 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 in the support database API is `SupportSQLiteOpenHelper`, and it fills the same basic role.

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 default Room implementation is `FrameworkSQLiteOpenHelperFactory`, from the `android.arch.persistence:db-framework` artifact
- CWAC-SafeRoom has `SafeHelperFactory`
- The `AssetRoom` sample app has `AssetSQLiteOpenHelperFactory`
- 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` and `AssetSQLiteOpenHelperFactory`, you just create an instance via a no-parameter constructor. CWAC-SafeRoom offers a one-parameter constructor on `SafeHelperFactory`, where you supply the passphrase as the parameter. `SafeHelperFactory` also has a static `fromUser()` method, for you to supply an `Editable` with the passphrase, such as from an `EditText` widget.

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. More often, though, 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:

```
// EditText passphraseField;
SafeHelperFactory factory=SafeHelperFactory.fromUser(passphraseField.getText());

StuffDatabase db=Room.databaseBuilder(ctxt, StuffDatabase.class, DB_NAME)
  .openHelperFactory(factory)
  .build();
```

Here, we are having Room use `SafeHelperFactory` from CWAC-SafeRoom, so Room will wind up interacting with SQLCipher for Android.

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

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.

For example, here we add a callback to manually create a table when the database is created:

```
RoomDatabase.Builder<BookDatabase> b=
  Room.databaseBuilder(ctxt.getApplicationContext(), BookDatabase.class,
    DB_NAME);

b.addCallback(new Callback() {
  @Override
  public void onCreate(@NonNull SupportSQLiteDatabase db) {
    super.onCreate(db);

    db.execSQL("CREATE VIRTUAL TABLE booksearch USING fts4(sequence, prose)");
  }
});

BookDatabase books=b.build();
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

Normally, creating tables is the job of an `@Entity` and its Room-generated code. In this case, we are creating an `fts4` virtual table, one used for full-text searching. Room does not know how to create those, so we have to create it ourselves.

We will see more about full-text searching and Room later in the book.

Room and 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, let's just say that users get somewhat irritated when you lose their data.

And that's where migrations come into play.

What's a Migration?

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` would do that 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()` rapidly grew 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 anonymous inner classes — 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 `YYYMMDD` (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,

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 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 the following `javaCompileOptions` closure:

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

(from Trips/RoomMigrations/app/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": "d46bfccddeca286f2948a702a4938d56",
    "entities": [
      {
        "tableName": "trips",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT NOT NULL, `title` TEXT,
`duration` INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime`
INTEGER, PRIMARY KEY(`id`))",
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "id",
```

```
          "affinity": "TEXT"
        },
        {
          "fieldPath": "title",
          "columnName": "title",
          "affinity": "TEXT"
        },
        {
          "fieldPath": "duration",
          "columnName": "duration",
          "affinity": "INTEGER"
        },
        {
          "fieldPath": "priority",
          "columnName": "priority",
          "affinity": "INTEGER"
        },
        {
          "fieldPath": "startTime",
          "columnName": "startTime",
          "affinity": "INTEGER"
        },
        {
          "fieldPath": "creationTime",
          "columnName": "creationTime",
          "affinity": "INTEGER"
        },
        {
          "fieldPath": "updateTime",
          "columnName": "updateTime",
          "affinity": "INTEGER"
        }
      ],
      "primaryKey": {
        "columnNames": [
          "id"
        ],
        "autoGenerate": false
      },
      "indices": [],
      "foreignKeys": []
    }
  ],
  "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,
\"d46bfccddeca286f2948a702a4938d56\")"
  ]
 }
}
```

(from Trips/RoomMigrations/app/schemas/com.commonsware.android.room.TripDatabase/1.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.

Writing Migrations

A `Migration` itself has only one required method: `migrate()`. You are given a `SupportSQLiteDatabase`, which we saw `SupportSQLiteDatabase` in the chapter on the support database API. 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. Hence, the recommended pattern is to use anonymous inner classes, where you can provide the `migrate()` method 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 `Trips/RoomMigrations` sample project has a `FROM_1_TO_2` migration:

```
  static final Migration FROM_1_TO_2=new Migration(1,2) {
    @Override
    public void migrate(SupportSQLiteDatabase db) {
      db.execSQL("CREATE TABLE IF NOT EXISTS `lodgings` (`id` TEXT NOT NULL, `title` TEXT, `duration`
INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER,
`address` TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`) ON UPDATE
NO ACTION ON DELETE CASCADE )");
      db.execSQL("CREATE  INDEX `index_lodgings_tripId` ON `lodgings` (`tripId`)");
      db.execSQL("CREATE TABLE IF NOT EXISTS `flights` (`id` TEXT NOT NULL, `title` TEXT, `duration`
INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime` INTEGER,
`departingAirport` TEXT, `arrivingAirport` TEXT, `airlineCode` TEXT, `flightNumber` TEXT, `seatNumber`
TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`) ON UPDATE NO ACTION
ON DELETE CASCADE )");
      db.execSQL("CREATE  INDEX `index_flights_tripId` ON `flights` (`tripId`)");
    }
  };
```

(from Trips/RoomMigrations/app/src/main/java/com/commonsware/android/room/Migrations.java)

Here, we create two tables and two indexes in `migrate()`. The SQL is mostly copied from the `2.json` file, representing the schema for version `2`:

```
{
  "formatVersion": 1,
  "database": {
    "version": 2,
```

```
    "identityHash": "69efe3a24b62764afa37e5eb0f162fd9",
    "entities": [
      {
        "tableName": "trips",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT NOT NULL, `title` TEXT,
`duration` INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime`
INTEGER, PRIMARY KEY(`id`))",
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "id",
            "affinity": "TEXT",
            "notNull": true
          },
          {
            "fieldPath": "title",
            "columnName": "title",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "duration",
            "columnName": "duration",
            "affinity": "INTEGER",
            "notNull": true
          },
          {
            "fieldPath": "priority",
            "columnName": "priority",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "startTime",
            "columnName": "startTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "creationTime",
            "columnName": "creationTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "updateTime",
            "columnName": "updateTime",
            "affinity": "INTEGER",
            "notNull": false
          }
        ],
        "primaryKey": {
          "columnNames": [
            "id"
          ],
          "autoGenerate": false
        },
        "indices": [],
        "foreignKeys": []
      },
```

```
      {
        "tableName": "lodgings",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT NOT NULL, `title` TEXT,
`duration` INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime`
INTEGER, `address` TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`)
ON UPDATE NO ACTION ON DELETE CASCADE )",
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "id",
            "affinity": "TEXT",
            "notNull": true
          },
          {
            "fieldPath": "title",
            "columnName": "title",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "duration",
            "columnName": "duration",
            "affinity": "INTEGER",
            "notNull": true
          },
          {
            "fieldPath": "priority",
            "columnName": "priority",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "startTime",
            "columnName": "startTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "creationTime",
            "columnName": "creationTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "updateTime",
            "columnName": "updateTime",
            "affinity": "INTEGER",
            "notNull": false
          },
          {
            "fieldPath": "address",
            "columnName": "address",
            "affinity": "TEXT",
            "notNull": false
          },
          {
            "fieldPath": "tripId",
            "columnName": "tripId",
            "affinity": "TEXT",
            "notNull": false
```

```
        }
      ],
      "primaryKey": {
        "columnNames": [
          "id"
        ],
        "autoGenerate": false
      },
      "indices": [
        {
          "name": "index_lodgings_tripId",
          "unique": false,
          "columnNames": [
            "tripId"
          ],
          "createSql": "CREATE  INDEX `index_lodgings_tripId` ON `${TABLE_NAME}` (`tripId`)"
        }
      ],
      "foreignKeys": [
        {
          "table": "trips",
          "onDelete": "CASCADE",
          "onUpdate": "NO ACTION",
          "columns": [
            "tripId"
          ],
          "referencedColumns": [
            "id"
          ]
        }
      ]
    },
    {
      "tableName": "flights",
      "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`id` TEXT NOT NULL, `title` TEXT,
`duration` INTEGER NOT NULL, `priority` INTEGER, `startTime` INTEGER, `creationTime` INTEGER, `updateTime`
INTEGER, `departingAirport` TEXT, `arrivingAirport` TEXT, `airlineCode` TEXT, `flightNumber` TEXT,
`seatNumber` TEXT, `tripId` TEXT, PRIMARY KEY(`id`), FOREIGN KEY(`tripId`) REFERENCES `trips`(`id`) ON
UPDATE NO ACTION ON DELETE CASCADE )",
      "fields": [
        {
          "fieldPath": "id",
          "columnName": "id",
          "affinity": "TEXT",
          "notNull": true
        },
        {
          "fieldPath": "title",
          "columnName": "title",
          "affinity": "TEXT",
          "notNull": false
        },
        {
          "fieldPath": "duration",
          "columnName": "duration",
          "affinity": "INTEGER",
          "notNull": true
        },
        {
          "fieldPath": "priority",
```

```
        "columnName": "priority",
        "affinity": "INTEGER",
        "notNull": false
      },
      {
        "fieldPath": "startTime",
        "columnName": "startTime",
        "affinity": "INTEGER",
        "notNull": false
      },
      {
        "fieldPath": "creationTime",
        "columnName": "creationTime",
        "affinity": "INTEGER",
        "notNull": false
      },
      {
        "fieldPath": "updateTime",
        "columnName": "updateTime",
        "affinity": "INTEGER",
        "notNull": false
      },
      {
        "fieldPath": "departingAirport",
        "columnName": "departingAirport",
        "affinity": "TEXT",
        "notNull": false
      },
      {
        "fieldPath": "arrivingAirport",
        "columnName": "arrivingAirport",
        "affinity": "TEXT",
        "notNull": false
      },
      {
        "fieldPath": "airlineCode",
        "columnName": "airlineCode",
        "affinity": "TEXT",
        "notNull": false
      },
      {
        "fieldPath": "flightNumber",
        "columnName": "flightNumber",
        "affinity": "TEXT",
        "notNull": false
      },
      {
        "fieldPath": "seatNumber",
        "columnName": "seatNumber",
        "affinity": "TEXT",
        "notNull": false
      },
      {
        "fieldPath": "tripId",
        "columnName": "tripId",
        "affinity": "TEXT",
        "notNull": false
      }
    ],
    "primaryKey": {
```

```
        "columnNames": [
          "id"
        ],
        "autoGenerate": false
      },
      "indices": [
        {
          "name": "index_flights_tripId",
          "unique": false,
          "columnNames": [
            "tripId"
          ],
          "createSql": "CREATE  INDEX `index_flights_tripId` ON `${TABLE_NAME}` (`tripId`)"
        }
      ],
      "foreignKeys": [
        {
          "table": "trips",
          "onDelete": "CASCADE",
          "onUpdate": "NO ACTION",
          "columns": [
            "tripId"
          ],
          "referencedColumns": [
            "id"
          ]
        }
      ]
    }
  ],
  "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,
\"69efe3a24b62764afa37e5eb0f162fd9\")"
  ]
 }
}
```

(from Trips/RoomMigrations/app/schemas/com.commonsware.android.room.TripDatabase/2.json)

In the JSON, the `createSql` properties have the table name as a template-style macro (`${TABLE_NAME}`), which you will need to replace with the actual table name. The backticks are supported in SQLite as they are in MySQL, and since they cause no harm here, usually it is simpler just to leave them in there.

Employing Migrations

Simply creating a `Migration` as a `static` field somewhere is necessary but not sufficient 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.commonsware.android.room;

import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;

@Database(
  entities={Trip.class, Lodging.class, Flight.class},
  version=2
)
abstract class TripDatabase extends RoomDatabase {
  abstract TripStore tripStore();

  private static final String DB_NAME="trips.db";
  private static volatile TripDatabase INSTANCE=null;

  synchronized static TripDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt, false);
    }

    return(INSTANCE);
  }

  static TripDatabase create(Context ctxt, boolean memoryOnly) {
    return(create(ctxt, DB_NAME, memoryOnly));
  }

  static TripDatabase create(Context ctxt, String name, boolean memoryOnly) {
    RoomDatabase.Builder<TripDatabase> b;

    if (memoryOnly) {
      b=Room.inMemoryDatabaseBuilder(ctxt.getApplicationContext(),
        TripDatabase.class);
    }
    else {
      b=Room.databaseBuilder(ctxt.getApplicationContext(), TripDatabase.class,
        name);
    }

    return(b.addMigrations(Migrations.FROM_1_TO_2).build());
  }
}
```

(from Trips/RoomMigrations/app/src/main/java/com/commonsware/android/room/TripDatabase.java)

Here, we teach the `RoomDatabase.Builder` about the `FROM_1_TO_2` `Migration`. In this

sample project, the migrations are implemented in a separate `Migrations` class, though you are welcome to have them directly in your `RoomDatabase` class or wherever makes sense for you.

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, it may be that for some device you need to skip a schema version, such as moving from version 1 to 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, 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 `android.arch.persistence.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 "com.android.support:recyclerview-v7:28.0.0"
  implementation 'com.android.support:support-fragment:28.0.0'
  androidTestImplementation 'com.android.support:support-compat:28.0.0'
  androidTestImplementation 'com.android.support:support-core-utils:28.0.0'
  implementation "android.arch.persistence.room:runtime:1.1.1"
  annotationProcessor "android.arch.persistence.room:compiler:1.1.1"
  androidTestImplementation "com.android.support:support-annotations:28.0.0"
  androidTestImplementation 'com.android.support.test:rules:1.0.2'
  androidTestImplementation "android.arch.persistence.room:testing:1.1.1"
}
```

(from Trips/RoomMigrations/app/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 Trips/RoomMigrations/app/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.commonsware.android.room.TripDatabase/`) 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 and Using a MigrationTestHelper

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

Adding the Rule

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

```
@Rule
public MigrationTestHelper helper;
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

Setting Up the Helper

You then need to create an instance of the `MigrationTestHelper`, such as in a `@Before`-annotated method:

```
@Before
public void setUp() {
  helper=new MigrationTestHelper(InstrumentationRegistry.getInstrumentation(),
    TripDatabase.class.getCanonicalName());
}
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

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

```
    SupportSQLiteDatabase db=helper.createDatabase(DB_NAME, 1);
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

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:

```
    SupportSQLiteDatabase db=helper.createDatabase(DB_NAME, 1);

    db.execSQL("INSERT INTO trips (id, title, duration) VALUES (1, NULL, 0)");

    final Cursor firstResults=db.query("SELECT COUNT(*) FROM trips");

    assertEquals(1, firstResults.getCount());
    firstResults.moveToFirst();
    assertEquals(1, firstResults.getInt(0));

    firstResults.close();
    db.close();
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

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:

```
    db=helper.runMigrationsAndValidate(DB_NAME, 2, true,
      Migrations.FROM_1_TO_2);
```

(from Trips/RoomMigrations/app/src/androidTest/java/com/commonsware/android/room/MigrationTests.java)

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.FROM_1_TO_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.

Lifecycle Components and ViewModels

Lifecycles and Owners

Programmers, in any environment, often encounter one or more topics that inspire the five stages of grief. It might be related to threads, to security, to UI implementation (e.g., how to deal with resizeable windows).

Android developers experience this sort of grief on all those topics.

Another one that triggers this sort of grief is the concept of lifecycles. On the surface, the concept seems unremarkable: objects are in use for a time and then become discarded, and along the way we receive callbacks regarding their state. However, dealing with the ramifications of those lifecycles — such as handling configuration changes, like screen rotation — vex even seasoned Android developers.

Part of the Architecture Components is a series of classes designed to help you deal with lifecycles in a more consistent fashion.

A Tale of Terminology

The Architecture Components have very specific definitions for certain terms, and these definitions affect the classes that we wind up using.

Lifecycle

A lifecycle is a series of states that an object can be in. Hence, a trivial lifecycle simply has “alive” and “dead” or similar states.

The eponymous `Lifecycle` class, however, models a *specific* lifecycle, that of activities and fragments.

Lifecycle Owner

A lifecycle owner is simply something that goes through a lifecycle. If the lifecycle is the state, the lifecycle owner is what has the trigger events for navigating through the state machine.

A `LifecycleOwner` is a Java interface, with a `getLifecycle()` method, that returns the `Lifecycle` for a given owner. As we will see, various classes already implement `LifecycleOwner`, and adding it to something else is not especially difficult.

Lifecycle Observers

A lifecycle observer is something that is notified about the change in state of some lifecycle. It finds out about those trigger events and the movement of the lifecycle from state to state.

There are two ways to do this, via annotations and via `DefaultLifecycleObserver`, as we will explore later in this chapter.

Adding the Lifecycle Components

You will need a runtime dependency and an annotation processor, akin to how Room is set up:

```
dependencies {
  implementation "com.android.support:recyclerview-v7:28.0.0"
  implementation "com.android.support:support-fragment:28.0.0"
  implementation "android.arch.lifecycle:runtime:1.1.1"
  implementation "android.arch.lifecycle:common-java8:1.1.1"
  implementation "android.arch.core:runtime:1.1.1"
}

android {
    compileSdkVersion 28


  defaultConfig {
      minSdkVersion 21
      targetSdkVersion 28
  }

  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
```

```
    targetCompatibility JavaVersion.VERSION_1_8
  }
}
```

(from General/Lifecycle/app/build.gradle)

However, making sense of the artifact versions — particularly when transitive dependencies come into play — will be difficult.

If your project directly or indirectly depends upon `support-compat` version `26.1.0` or higher, `support-compat` has a dependency on `android.arch.lifecycle:runtime`, for some version of that artifact. What version that is will depend on the version of `support-compat`:

Support Library Version	`android.arch.lifecycle:runtime` Version
`28.0.0`	`1.1.1`
`27.1.1`	`1.1.1`
`27.0.2`	`1.0.3`
`26.1.0`	`1.0.0`

In addition, we have a dependency on `android.arch.lifecycle:common-java8`, which provides better Java 8 support for the lifecycle system.

How these versions will work out in the future is anyone's guess right now.

Getting a Lifecycle

Everything dealing with `Lifecycle` comes down to a `LifecycleOwner`. You have several possibilities of where to get one of those.

…From a FragmentActivity or a Support Fragment

If you are using version `26.1.0` or higher of the Support Library artifacts, then `FragmentActivity` and the `android.support.v4.app.Fragment` class both implement `LifecycleOwner`.

If you are using an older version of the Support Library artifacts... you *really* should

upgrade to at least `26.1.0` to use the Architecture Components.

…From an AppCompatActivity

Perhaps you are using the `appcompat-v7` artifact. In that case, you are inheriting from `AppCompatActivity` instead of `FragmentActivity` or `Activity`.

However, since `AppCompatActivity` inherits from `FragmentActivity`, if you are using `26.1.0` or higher of `appcompat-v7`, your `AppCompatActivity` subclasses will also implement `LifecycleOwner`.

…From an Activity or Fragment

Perhaps you are using the classic `Activity` and `Fragment` classes, or from classes that extend those (e.g., `WearableActivity`). Those will never directly implement `LifecycleOwner`, as framework classes cannot depend upon libraries.

The simplest solution is to switch to inheriting from `FragmentActivity` and the corresponding backport of `Fragment`.

Otherwise, this means that we need to handle this in a more complex fashion, outlined later in the chapter.

…From Anything Else

In principle, you could have other objects that are themselves tied into the activity and fragment lifecycle. After all, the backport of fragments in the Support Library are just that sort of "other objects". It so happens that Google takes care of managing that backport. However, you might find other objects that, for whatever reason, are similar in concept to the fragments backport and therefore should be *suppliers* of lifecycle events.

In that case, you can implement `LifecycleOwner` on those classes. However, you will *also* need to call `handleLifecycleEvent()` method on the `LifecycleRegistry` at appropriate points.

This will be illustrated with support for ordinary activities, shown later in the chapter.

Observing a Lifecycle

Most likely, if you are interested in the Architecture Components, you are up to speed with Java 8 and are interested in using it in your project. In that case, you can go the preferred route and use `DefaultLifecycleObserver` as your observer implementation. This takes advantage of Java 8's ability to define methods on interfaces, so that you only need to override the particular lifecycle events that concern you.

So, for example, here is an observer that passes all events to a `RecyclerView.Adapter` named `EventLogAdapter`:

```
static class LObserver implements DefaultLifecycleObserver {
  private final EventLogAdapter adapter;

  LObserver(EventLogAdapter adapter) {
    this.adapter=adapter;
  }

  @Override
  public void onCreate(@NonNull LifecycleOwner owner) {
    adapter.add("ON_CREATE");
  }

  @Override
  public void onStart(@NonNull LifecycleOwner owner) {
    adapter.add("ON_START");
  }

  @Override
  public void onResume(@NonNull LifecycleOwner owner) {
    adapter.add("ON_RESUME");
  }

  @Override
  public void onPause(@NonNull LifecycleOwner owner) {
    adapter.add("ON_PAUSE");
  }

  @Override
  public void onStop(@NonNull LifecycleOwner owner) {
    adapter.add("ON_STOP");
  }

  @Override
```

```
    public void onDestroy(@NonNull LifecycleOwner owner) {
      adapter.add("ON_DESTROY");
    }
  }
```

(from General/Lifecycle/app/src/main/java/com/commonsware/android/lifecycle/MainActivity.java)

In our case, we happen to pay attention to all of the events; that is not required.

Then, you can register the observer, and it will start being called for the various events:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  setTitle(getString(R.string.title, hashCode()));

  RecyclerView rv=findViewById(R.id.transcript);

  adapter=new EventLogAdapter(getLastCustomNonConfigurationInstance());
  rv.setAdapter(adapter);

  getLifecycle().addObserver(new LObserver(adapter));
}
```

(from General/Lifecycle/app/src/main/java/com/commonsware/android/lifecycle/MainActivity.java)

All of this code is from the `General/Lifecycle` sample project, which shows you the events in a `RecyclerView` as they come in. The `MainActivity` handles configuration changes via `onRetainCustomNonConfigurationInstance()`, so you can see the lifecycle events across a configuration change. Through an overflow menu item, you can kick off another instance of `MainActivity`, then press BACK to see the flow of lifecycle events as the original instance comes and goes from the foreground.

Legacy Options

The Java 8 and `FragmentActivity` approach is the simplest way to work with lifecycles. However, sometimes, those are not an option, and for that, you will need workarounds.

Ordinary Activities and Fragments, and Other Objects

Sometimes, you have to use activities and fragments not rooted in the Support Library backport. For example, `WearableActivity` for Android Wear does not extend `FragmentActivity`. By default, you cannot use such activities with the lifecycle system. And, sometimes, you might have some other object need to be the source of lifecycle events, independent of activities and fragments.

For these scenarios:

- Have your class implement the `LifecycleOwner` interface
- Use `LifecycleRegistry` to track the registered observers
- Return that registry from `getLifecycle()`, the one method on `LifecycleOwner` that you need to implement
- From all of the lifecycle methods, call `handleLifecycleEvent()` on the registry, indicating what lifecycle event has just occurred

For example, here is a `SimpleLifecycleActivity` that handles the standard activity lifecycle events, forwarding them to the `LifecycleRegistry`:

```
package com.commonsware.android.lifecycle;

import android.app.Activity;
import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleOwner;
import android.arch.lifecycle.LifecycleRegistry;
import android.os.Bundle;
import android.support.annotation.Nullable;

public class SimpleLifecycleActivity extends Activity
  implements LifecycleOwner {
  private LifecycleRegistry registry=new LifecycleRegistry(this);

  @Override
  public Lifecycle getLifecycle() {
    return(registry);
  }

  @Override
  protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    registry.handleLifecycleEvent(Lifecycle.Event.ON_CREATE);
  }
```

```
  @Override
  protected void onStart() {
    super.onStart();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_START);
  }

  @Override
  protected void onResume() {
    super.onResume();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_RESUME);
  }

  @Override
  protected void onPause() {
    super.onPause();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_PAUSE);
  }

  @Override
  protected void onStop() {
    super.onStop();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_STOP);
  }

  @Override
  protected void onDestroy() {
    super.onDestroy();

    registry.handleLifecycleEvent(Lifecycle.Event.ON_DESTROY);
  }
}
```

(from General/LifecycleLegacy/app/src/main/java/com/commonsware/android/lifecycle/SimpleLifecycleActivity.java)

Pre-Java 8

Perhaps Java 8 is not an option for you, for whatever reason. `DefaultLifecycleObserver` will not work for you. Instead, you will need to:

- Remove the `android.arch.lifecycle:common-java8` dependency, as it will not be compatible with your app

- Add an `annotationProcessor` dependency on `android.arch.lifecycle:compiler`
- Have your observer implement `LifecycleObserver` instead of `DefaultLifecycleObserver`
- Implement one or more methods, annotated with `@OnLifecycleEvent`, to receive the lifecycle events of interest to you

So, for example, here is an observer that passes all events to a `RecyclerView.Adapter` named `EventLogAdapter`:

```
static class LObserver implements LifecycleObserver {
  private final EventLogAdapter adapter;

  LObserver(EventLogAdapter adapter) {
    this.adapter=adapter;
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
  void created() {
    adapter.add("ON_CREATE");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_START)
  void started() {
    adapter.add("ON_START");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
  void resumed() {
    adapter.add("ON_RESUME");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
  void paused() {
    adapter.add("ON_PAUSE");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
  void stopped() {
    adapter.add("ON_STOP");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
  void destroyed() {
    adapter.add("ON_DESTROY");
  }
```

```
  }
```

(from General/LifecycleLegacy/app/src/main/java/com/commonsware/android/lifecycle/MainActivity.java)

Note:

- There is also a `Lifecycle.Event.ON_ANY` event that you can request; this triggers your method to be called for any lifecycle event... though you have no way of knowing what event it was
- A single method can only have one `@OnLifecycleEvent` annotation, and that annotation accepts only a single `Lifecycle.Event` value (not a list)

As noted, you also need the annotation processor, so those `@OnLifecycleEvent` annotations can be interpreted and applied:

```
dependencies {
    implementation 'com.android.support:recyclerview-v7:28.0.0'
    implementation 'android.arch.lifecycle:runtime:1.1.1'
    annotationProcessor 'android.arch.lifecycle:compiler:1.1.0'
}
```

(from General/LifecycleLegacy/app/build.gradle)

So, What's the Point of This?

On the surface, this all seems fairly silly. One could just as easily override the lifecycle methods on `MainActivity` and log directly to the `RecyclerView`, bypassing all this `Lifecycle` and `LifecycleObserver` stuff.

The reason why `Lifecycle` and `LifecycleObserver` exist is to provide a standardized way of having *other* classes find out about lifecycle changes. Overriding the lifecycle methods on an activity or fragment tell that activity or fragment about the changes, but that's it.

So, for example, `LiveData` — the subject of the next chapter — is a `LifecycleObserver`, so it knows about lifecycle events and can activate/deactivate accordingly. Other libraries may implement `LifecycleObserver` so they can be plugged into your activities and fragments and find out about lifecycle events, without you having to manually dispatch those events to them.

In ordinary apps, though, most developers will not be creating their own `LifecycleObserver` classes, though anyone can, as the sample app demonstrates.

The focus for most app developers using the Architecture Components will be on `LiveData` and, later, `ViewModel`.

LiveData

`Lifecycle`, `LifecycleOwner`, and related classes mostly exist to provide the foundation for `LiveData`. `LiveData` is the next generation of various Android asynchronous solutions, such as `AsyncTask` and the `Loader` framework. `LiveData`, in particular, is modeled somewhat after RxJava, a popular reactive programming library.

All of this is to set up ways for you to be able to observe changes to data without having to worry as much about activity and fragment lifecycles... though, as it turns out, you cannot escape them entirely.

Observables Are the New Black

The observer pattern in software design has been around for decades. Yet, it has caught fire in the past few years, repackaged as "reactive programming". Reactive programming visualizes an app as a set of streams of data changes, whether from the user (e.g., UI widget interactions), from a server (e.g., updates to data from a sync operation), or from something else (e.g., GPS fixes). Developers set up observers to respond ("react") to these data changes and apply updates to the UI.

The centerpiece for reactive programming in Android is RxJava, typically combined with RxAndroid. RxJava provides the basic framework for observing streams of data changes, with RxAndroid primarily providing ways to route results of observations to the main application thread. This book is not going to go into details of how you use RxJava/RxAndroid in general — for that, see *The Busy Coder's Guide to Android Development* or other books.

One problem with RxJava, though, is that "it is difficult to get your head wrapped around it". Reactive programming works great in platforms that implemented

reactive programming from the outset. Reactive programming is more difficult to bolt onto an existing platform, both from a technical standpoint and from a documentation standpoint. RxJava is the sort of technology that is easy to illustrate in "hello, world"-level examples but gets difficult to explain for more practical scenarios. In part, that is because RxJava is extremely flexible, and with great flexibility comes great need for great documentation... which RxJava historically lacked.

`LiveData` is designed to be a much lighter-weight approach to reactive programming, designed to do one thing (deliver asynchronous data changes regardless of lifecycle events) and do it reasonably well.

Yet More Terminology

First, let's review some new and exciting terms that we need to understand in order to use `LiveData`.

`LiveData`

`LiveData` itself is a source of data, both for a point in time and (via an observer) for changes to that data over time. Something will create and hand you a `LiveData` object, where the work to get that data and update it over time is handled by some background thread coming from the `LiveData` supplier.

`Observer`

In principle, you can call `getValue()` on a `LiveData` to get the current value for whatever stream of data the `LiveData` is tracking. In practice, this will not be especially common.

Instead, you will register an `Observer` with the `LiveData`, usually via an `observe()` method. Your `Observer` will be called with `onChanged()` when:

- You start observing and there is already data in the `LiveData`, and
- When the `LiveData` finds out about a change in the data

Your `onChanged()` method is given the data (a `Location`, a `SensorEvent`, a Room entity, whatever) on the main application thread, with an eye towards you using it to update the UI by one means or another.

Active State

If a `LiveData` was instantiated in a forest, and nobody was there to observe data changes, does the `LiveData` really exist?

The answer is: yes, but it hopefully is not consuming any resources.

A `LiveData` implementation will be called with `onActive()` when it receives its first active observer. Here, "active" means that, if the observer is tied to a `LifecycleOwner`, the lifecycle is in the started or resumed state. Conversely, the `LiveData` will be called with `onInactive()` once it no longer has any active observers, either because all observers have been unregistered or none of them are active, as their lifecycles are all stopped or destroyed.

The idea is that a `LiveData` would only start consuming significant system resources — such as requesting GPS fixes — when there are active observers, releasing those resources when there are no more active observers. This works in many cases, though there are some that will require more finesse. For example, given that the GPS radio takes some time before it starts generating GPS fixes, a `LiveData` for GPS might want to wait some amount of time after losing its last active observer before releasing the GPS radio, in case a new observer pops up quickly, to avoid delays in getting those GPS fixes.

Implementing `LiveData`

With that as background, let's see `LiveData` in action. The `Sensor/LiveList` sample project implements `LiveData` for sensor readings coming from a `SensorManager`. We can use this to track the accelerometer, ambient light, and so on.

However, the technique shown here can be used for lots of different system-level data sources, such as:

- Other system services (e.g., `LocationManager`, `ClipboardManager`)
- System broadcasts, for cases where you want to dynamically register for the broadcast via `registerReceiver()`
- Local broadcasts, using `LocalBroadcastManager`
- Content changes in providers, via a `ContentObserver`

Dependencies

To use `Lifecycle` and `LifecycleOwner`, you needed two dependencies: the lifecycle `runtime` library and its `compiler` annotation processor.

`LiveData` has its own dependency: `android.arch.lifecycle:livedata`:

```
dependencies {
    implementation 'com.android.support:recyclerview-v7:28.0.0'
    implementation 'com.android.support:support-fragment:28.0.0'
    implementation 'android.arch.lifecycle:livedata:1.1.1'
}
```

(from Sensor/LiveList/app/build.gradle)

Of note:

- You do not need the `android.arch.lifecycle:runtime` dependency, as the `android.arch.lifecycle:livedata` dependency will pull that in for you
- You do not need the lifecycle annotation processor to use `LiveData`

State Transitions

We have a `SensorLiveData` class that extends the `LiveData` base class, offering to support a custom `Event` `static` nested class:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.LiveData;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import java.util.Date;

class SensorLiveData extends LiveData<SensorLiveData.Event> {
  final private SensorManager sensorManager;
  private final Sensor sensor;
  private final int delay;

  SensorLiveData(Context ctxt, int sensorType, int delay) {
    sensorManager=
      (SensorManager)ctxt.getApplicationContext()
        .getSystemService(Context.SENSOR_SERVICE);
```

```
    this.sensor=sensorManager.getDefaultSensor(sensorType);
    this.delay=delay;

    if (this.sensor==null) {
      throw new IllegalStateException("Cannot obtain the requested sensor");
    }
  }

  @Override
  protected void onActive() {
    super.onActive();

    sensorManager.registerListener(listener, sensor, delay);
  }

  @Override
  protected void onInactive() {
    sensorManager.unregisterListener(listener);

    super.onInactive();
  }

  final private SensorEventListener listener=new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
      setValue(new Event(event));
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
      // unused
    }
  };

  static class Event {
    final Date date=new Date();
    final float[] values;

    Event(SensorEvent event) {
      values=new float[event.values.length];

      System.arraycopy(event.values, 0, values, 0, event.values.length);
    }
  }
}
```

(from Sensor/LiveList/app/src/main/java/com/commonsware/android/livedata/SensorLiveData.java)

In the constructor, we hold onto configuration details, such as the particular sensor to monitor and how frequently we should ask for updates. We also obtain an instance of the `SensorManager` system service and try to find the actual requested `Sensor`, throwing a runtime exception if there is no matching sensor on this device.

However, we do not register for sensor events in the constructor. Until we have 1+ active observers, we do not need those events, and monitoring sensor events drains the battery. So, we postpone registering for events until `onActive()`, unregistering in the corresponding `onInactive()` callback.

Updating the Observers

The `SensorEventListener` that we use, in its `onSensorChanged()` method, creates a new instance of our `Event`, grabbing data from the `SensorEvent`. We use our own `Event` class for two reasons:

1. `SensorEvent` objects get recycled, and so it is not safe to hold onto one of those after the end of `onSensorChanged()`, so we copy the sensor results `float` values into our own object
2. While a `SensorEvent` has a timestamp, it is a pain to use, and this is a casual book sample, so we just track our own `Date` for simplicity

That `Event` is passed to `setValue()` on the `LiveData`, which in turn will pass the result to observers. Note that `setValue()` needs to be called on the main application thread — we will see how to handle events originating on background threads later in this chapter.

Retaining the LiveData

So, we have a `LiveData` for sensor readings. We can have an activity that displays those readings, by having it create a `SensorLiveData` instance and registering to observe those events. But now we run into a problem... what do we do with the `SensorLiveData` object after that?

One possibility is that we just hold onto it in a field, mostly to ensure that nothing gets garbage-collected that would interrupt the sensor readings. If we undergo a configuration change, we just create a new `SensorLiveData` objects and a fresh observer. While this is not completely ridiculous for this particular scenario, it is bad for cases where setting up the `LiveData` is expensive.

The most likely solution would be to hold it in a viewmodel — we will see that in an

upcoming chapter.

In this sample app, we take a third approach, using `onRetainCustomNonConfigurationInstance()` inside the activity that is going to use the sensor readings. Since the UI is going to be a `RecyclerView` of readings, we also need to hold onto past readings, so we do not lose them when we undergo the configuration change.

So, we have a `State` `static` nested class that holds onto the `SensorLiveData` and outstanding readings:

```
private static class State {
  final ArrayList<SensorLiveData.Event> events=new ArrayList<>();
  SensorLiveData sensorLiveData;
}
```

(from Sensor/LiveList/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

In `onCreate()`, we set up that `State` if we do not already have one, storing it in a `state` field. This includes setting up the `SensorLiveData`, in this case for the ambient light sensor:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  RecyclerView rv=findViewById(R.id.transcript);

  state=(State)getLastCustomNonConfigurationInstance();

  if (state==null) {
    state=new State();
    state.sensorLiveData=
      new SensorLiveData(this, Sensor.TYPE_LIGHT,
        SensorManager.SENSOR_DELAY_UI);
  }

  adapter=new EventLogAdapter();
  rv.setAdapter(adapter);

  state.sensorLiveData.observe(this, event -> adapter.add(event));
}
```

(from Sensor/LiveList/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

We also register our `Observer`, which will be called with `onChanged()` with a new `Event` as sensor readings come in. Our `EventLogAdapter` knows how to `add()` that to the list of historical readings and update the `RecyclerView`.

However, the `LiveData` will automatically deliver the last-received reading to our observer when we attach a fresh observer after a configuration change. That could result in `onChanged()` being given the same `Event` object as before, one that we already put into the `ArrayList`. So, the `EventLogAdapter` `add()` method checks that first, before actually adding it:

```
  void add(SensorLiveData.Event what) {
    if (!state.events.contains(what)) {
      state.events.add(what);
      notifyItemInserted(getItemCount()-1);
    }
  }
```

(from Sensor/LiveList/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

And we override `onRetainNonConfigurationInstance()` to return the `State` instance, so `onCreate()` can retrieve it after a configuration change:

```
@Override
public Object onRetainCustomNonConfigurationInstance() {
  return(state);
}
```

(from Sensor/LiveList/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

Other LiveData Examples

Let's take a look at a few more examples of using `LiveData`, to explore other facets of how this can be used.

Event Bus

`LocalBroadcastManager` implements an in-process event bus, where events are delivered to you on the main application thread, and where "events" are `Intent` objects.

You can accomplish the same thing, with greater flexibility, by means of a `LiveData` object, as can be seen in the `General/LiveBus` sample project.

This sample app is derived from one shown in *The Busy Coder's Guide to Android Development*, where we have `AlarmManager` triggering a service. In principle, that service should do some work, which we are skipping here because we are lazy. However, the fake work is something that the user might care about, and so we want to let the UI layer know about the event if we happen to be in the foreground. Otherwise, we want to raise a `Notification`. In *The Busy Coder's Guide to Android Development*, implementations of this sample are available for a few event buses, including `LocalBroadcastManager` and greenrobot's EventBus.

Here, though, we will use a `MutableLiveData` singleton:

```
  static final MutableLiveData<Intent> BUS=new MutableLiveData<>();
```

(from General/LiveBus/app/src/main/java/com/commonsware/android/livedata/bus/ScheduledService.java)

`MutableLiveData` is a subclass of `LiveData`, with one key feature: it offers a `postValue()` method that works like `setValue()` but can be called from a background thread. Here, our events are in the form of `Intent` objects, the way they would be for `LocalBroadcastManager`. However, you could create your own custom event objects if you prefer, and typically that would be a better idea. In this case, the sample is demonstrating a quick-and-dirty change from `LocalBroadcastManager`, so we are keeping the event objects the same to reduce the number of code changes.

The service, as part of its work, asks the `BUS` whether there are any active observers, by means of `hasActiveObservers()`. If `hasActiveObservers()` returns `true`, we use `postValue()` to post the event onto our `BUS`. Otherwise, we raise a `Notification`, as our UI is not in the foreground.

Our `EventLogFragment` registers an observer lambda on the `BUS`, adding the events to its `ArrayAdapter`:

```
    ScheduledService.BUS.observe(this, intent -> adapter.add(intent));
```

(from General/LiveBus/app/src/main/java/com/commonsware/android/livedata/bus/EventLogFragment.java)

Unlike `LocalBroadcastManager`, this approach performs no `Intent` filtering, and we can have as many `MutableLiveData` objects as needed. So, you can create custom buses for different event channels, instead of using action strings as you might with `LocalBroadcastManager`.

Room

Having DAO methods in Room return a `LiveData` is simply a matter of setting them up that way:

```
@Query("SELECT * FROM Customer WHERE postalCode IN (:postalCodes) LIMIT :max")
LiveData<List<Customer>> findByPostalCodes(int max, String... postalCodes);
```

(from General/LiveRoom/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Now, `findByPostalCodes()` will return a `LiveData`. Moreover, it will do so *immediately* when called, with the actual query being performed on a Room-supplied background thread. You can arrange to register an observer to find out when the results are ready. And, by using the same `LiveData` instance after a configuration change, you can get the last-loaded results without having to perform another round of disk I/O.

However, Room has an additional feature: if you make changes to the database through your DAO, Room will deliver fresh results to any registered observer of your `LiveData`. So, for example:

- You register an observer on a `LiveData`, returned by a Room `@Query`, that represents a list of your entities
- Shortly thereafter, you get the list of entities as they exist in the database at present, for you to fill into your `RecyclerView` (or whatever)
- Later on, as part of processing a request from the user, you invoke an `@Insert` method on your DAO to add a new entity to the database
- Your registered observer gets the updated list of entities as they exist in the database, for you to fill into your `RecyclerView` (or whatever)
- And so on

In effect, Room attempts to give you `ContentObserver` capabilities, for your own database, tied directly into the `LiveData` system.

Note, though, that these changes are tied in large part to your use of the DAO. For example, if you want to insert 100 entities, you could:

- Call a single `@Insert` method that takes a `List` of those entities, in which case you will get a single update from the `LiveData`
- Call a one-entity `@Insert` method 100 times, in which case you will get 100 updates from the `LiveData`

Doing things in batch form generally will be more efficient, both from a disk I/O standpoint and a `LiveData`-updating standpoint. On the other hand, this means that a `LiveData` update might represent several changes, and that may require additional smarts to handle properly in terms of updating the UI (e.g., use `DiffUtil` to efficiently update a `RecyclerView`).

We will see using `LiveData` with Room in the next chapter.

Many Android apps are trivial. The smaller the app, the less likely it is that you need much in the way of a true GUI architecture. Slapping together whatever you want wherever you want it most likely will suffice. Your average soundboard, flashlight, front-facing-camera "mirror", and similar apps just do what they do, and their developers do not need to worry about the alphabet soup of MVC, MVP, MVVM, MVI, and so on.

If you are reading this book, you may have an app in mind that is not so trivial.

The more complex the app, the more likely it is that you are going to want to think more seriously about the GUI architecture. The Architecture Components contribution to this is the `ViewModel`, which we will explore in this chapter.

Second-generation coverage of `ViewModel` can be found in the "Integrating ViewModel" chapter of *Elements of Android Jetpack*!

Viewmodels, As Originally Envisioned

Microsoft devised the model-view-viewmodel (MVVM) GUI architecture in 2005, and it has remained generally murky ever since. This is not terribly surprising, as many of the "alphabet soup" GUI architectures have malleable definitions which developers can twist and tweak to match what it is that they want to write.

Roughly speaking, in this GUI architecture, the "viewmodel" represents a collection of data and other state, necessary to render a view, derived from the underlying models. The viewmodel would be responsible for things like data formatting (e.g.,

converting the model's `long` Unix epoch time into something that the user will be able to read).

Ideally, the viewmodel knows nothing much about the view, but rather just exposes data and operations that the view needs.

The Architecture Components ships with a `ViewModel` class. This class does almost nothing on its own. Consider `ViewModel` to be a place to hold the data necessary to represent your views. For example, a `ViewModel` might hold a list of objects, obtained from Room, that are used to populate a `RecyclerView`.

`ViewModel` Versus...

The objective of `ViewModel`, in particular, is to be able to survive past configuration changes.

Of course, we have been dealing with configuration changes for years, before the Architecture Components were a glimmer in any Google engineer's eye.

So, when would we use a `ViewModel`, and when would we use other techniques?

...Saved Instance State

Saved instance state — what you put into the `Bundle` supplied to `onSaveInstanceState()` – survives process termination. A `ViewModel` does not. So while both can help deal with configuration changes, only saved instance state can help with the process termination scenario:

- User is in your app, in an activity
- User navigates to something else (e.g., presses HOME, switches to another task via the overview screen)
- A few minutes later, Android terminates your process to free up system RAM
- A few minutes after that — but within 30 minutes of the user navigating away – the user returns to your task
- Android recreates the activity atop your task's back stack as part of forking a fresh process for you, and Android hands you your saved instance state `Bundle` back

However, the saved instance state `Bundle` has size limits (should be well under 1MB) and type limits (only objects that can go into a `Parcel`).

As a result:

- Use the `ViewModel` for holding onto data in your process necessary to be able to rapidly repopulate the UI after a configuration change
- Use the saved instance state `Bundle` to hold identifiers and other data that will help you rebuild the UI after process termination, even if you wind up having to re-read from disk or the network as part of that work

…Retained Objects

In the end, the `ViewModelProviders` system supplied by the Architecture Components is a wrapper around retained fragments. As a result, there is nothing that you can do with a `ViewModel` that you could not do using retained objects, whether those are retained fragments or using `onRetainCustomNonConfigurationInstance()`.

Dependencies

In theory, you get `ViewModel` from the `android.arch.lifecycle:viewmodel` artifact.

In practice, you will want the `android.arch.lifecycle:extensions` artifact instead. This not only pulls in `android.arch.lifecycle:viewmodel` for you, but it provides the classes necessary to obtain a `ViewModel` associated with your activities and fragments.

Mommy, Where Does a `ViewModel` Come From?

You might think that you create a `ViewModel` via whatever constructor you set up for it.

Instead, the Architecture Components expect you to get a `ViewModel` instance by using `ViewModelProvider`. A `ViewModelProvider` instance is tied to either:

- A `FragmentActivity` (or a subclass, like `AppCompatActivity`), or
- A `Fragment`, from the fragments backport

If you do not have one of those, you cannot use `ViewModelProvider`.

If you *do* have one of those, call the `static of()` method on the `ViewModelProviders` class (note the plural) to get a `ViewModelProvider` (note the

singular) tied to your FragmentActivity or Fragment. This ViewModelProvider is tied to the *logical* instance of this activity or fragment, regardless of configuration changes. So, if the activity is destroyed and recreated as part of a configuration change, you will get the same ViewModelProvider instance in the new activity as you had in the old one.

Then, to get a ViewModel, call get() on the ViewModelProvider, passing in the Java class object for your subclass of ViewModel (e.g., MyViewModel.class). If there already is an instance of this ViewModel tied to this ViewModelProvider, you get that instance. Otherwise, a fresh instance will be created for you, from the zero-argument constructor. If using the zero-argument constructor is not what you want, you can:

- Create an implementation of the ViewModelProvider.Factory interface, implementing the create() method to create an instance of your ViewModel by whatever constructor you want
- Associate an instance of your ViewModelProvider.Factory with the ViewModelProvider by supplying it as a second parameter to the of() method on ViewModelProviders

So, in the typical case, you wind up with code like this:

```
    TripRosterViewModel vm=
      ViewModelProviders.of(this).get(TripRosterViewModel.class);
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

Here, this inherits from the Fragment backport, and we are retrieving a TripRosterViewModel to use in that fragment.

We will see this code snippet again in the next section.

ViewModel In Action

So, let's take a look at the Trips/ViewModels sample project. This adds a ViewModel to our app showing a roster of upcoming trips. More specifically, we will use ViewModelProvider, the way Google envisioned it.

Earlier editions of this sample used Android's native Activity and Fragment classes. Those do not work with ViewModelProviders. So, in this sample, MainActivity has been revised to extend from FragmentActivity and RecyclerViewFragment has been

revised to extend from the Support Library edition of `Fragment`.

Defining a ViewModel

The idea is that a `ViewModel` should hold the data necessary to render the UI. In our case, that is simply a roster of `Trip` objects, pulled in from Room.

For `ViewModelProvider` to work, the class must be `public`, even though your IDE might suggest otherwise. So, our `TripRosterViewModel` is `public`:

```
package com.commonsware.android.room;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.arch.lifecycle.LiveData;
import java.util.List;

public class TripRosterViewModel extends AndroidViewModel {
  final LiveData<List<Trip>> allTrips;

  public TripRosterViewModel(Application app) {
    super(app);

    allTrips=TripDatabase.get(app).tripStore().selectAllTrips();
  }
}
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripRosterViewModel.java)

Note that `TripRosterViewModel` extends from `AndroidViewModel`. `AndroidViewModel` itself extends `ViewModel`. The only difference between the two is the constructor: `ViewModel` has a zero-argument constructor, while `AndroidViewModel` has a one-argument constructor, supplying the `Application` instance. In our case, we need the `Application` instance to `get()` our `TripDatabase` (as Room needs a `Context` for this).

`TripRosterViewModel`, in its constructor, sets up an `allTrips` field that is a `LiveData` of our roster of `Trip` objects. Since this is `LiveData`, the actual work will not be done until we ask it to, by registering an observer to use the results.

Getting a ViewModel

Our `TripsFragment` needs access to the `TripRosterViewModel`, in order to be able to get to the `allTrips` data and request the roster of `Trip` objects.

However, now we have a decision to make: is the `TripRosterViewModel` tied to the fragment or to the activity?

Since a fragment can get to its hosting activity via `getActivity()`, a fragment can choose either scope:

- Pass `this` into `of()` to get the `ViewModelProvider` tied to the fragment, or
- Pass `getActivity()` into `of()` to get the `ViewModelProvider` tied to the activity

Either is perfectly legitimate. Frequently, it will boil down to who needs the data. Data that is only needed by a single fragment should be owned by a `ViewModel` tied to that fragment. Data needed by multiple fragments, or by a fragment and the activity, or just by the activity, should be owned by a `ViewModel` tied to the activity. A fragment can also elect to do both, using two `ViewModel` instances, one for its own data and one that it gets via the activity.

In this case, the only UI is the `TripsFragment`, so we can say that the `TripRosterViewModel` is owned by the fragment and retrieve it as part of our `onViewCreated()` work:

```
    TripRosterViewModel vm=
      ViewModelProviders.of(this).get(TripRosterViewModel.class);
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

The first time we run through these lines, we will get a fresh `TripRosterViewModel` instance. If we undergo a configuration change, when this fragment is recreated, the new fragment instance will get the same `TripRosterViewModel` as before.

Using the ViewModel

Given our `TripRosterViewModel`, our `TripsFragment` can now get at the roster of `Trip` objects, by registering an `Observer` (via a lambda expression):

```
    vm.allTrips.observe(this, trips -> {
      setAdapter(new TripsAdapter(trips, getActivity().getLayoutInflater()));

      if (trips==null || trips.size()==0) {
        final TripStore store=TripDatabase.get(getActivity()).tripStore();

        new Thread() {
          @Override
          public void run() {
            store.insert(new Trip("Vacation!", 10080, Priority.MEDIUM, new Date()),
```

```
            new Trip("Business Trip", 4320, Priority.OMG, new Date()));
        }
      }.start();
    }
  });
```

(from Trips/ViewModels/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

A typical app would just have the `setAdapter()` call, to pass the `Trip` roster over to the `TripsAdapter`, to show the roster in the `RecyclerView`. In this case, we want to lazy-create some trips, as otherwise we will have no data. So, if we have no trips, we insert some in a background thread.

However, there are two issues with that approach. One is the possible race condition, where the user rotates the screen while the background thread is going on, and so we fork a second thread. Since this code is not the sort of thing you would do in a production app, what we have here will suffice for now.

But, if you run the app, you will see that our data shows up in the `RecyclerView`, even after a fresh run of the app, when we did not have any data. Yet, our `Thread` is not doing anything to refresh the UI. So, the second issue is: how is this working?

The answer is that Room is monitoring our DAO for changes and is automatically updating the `LiveData` to reflect those changes, as was mentioned in the chapter on `LiveData`.

Getting Rid of the ViewModel

Ideally, you should not have to do anything to explicitly "get rid of" a `ViewModel`. If you are using `LiveData`, it is lifecycle-aware, and so it should clean up itself when the activity or fragment is destroyed. If you have anything else in the `ViewModel` that needs cleanup when the activity or fragment is destroyed, either:

- Use lifecycle-aware objects for that (e.g., `LiveData`), or
- Override `onCleared()` and clean up the objects at that point

When the `ViewModel` will no longer be used, the `ViewModel` will be called with `onCleared()`. This is an opportunity for you to release anything that needs to be released and will not just go away as part of normal garbage collection or lifecycle cleanup. So, for example, if you are holding an RxJava `Disposable` in a `ViewModel`, `onCleared()` is a good place to `dispose()` of it.

Other Lifecycle Owners

Activities and fragments are not the only things with lifecycles. The Architecture Components also support other forms of lifecycle owner:

- Services, and
- What the documentation will refer to as "the process"

LifecycleService

If you have a class that extends `Service`, you can replace it with `LifecycleService` and get a service that is a `LifecycleOwner`. Four of the six lifecycle events are honored:

This Lifecycle Event...	Is Triggered When...
`ON_CREATE`	the service is created
`ON_START`	when the service is first started or bound to
`ON_RESUME`	**unused**
`ON_PAUSE`	**unused**
`ON_STOP`	when the service is destroyed
`ON_DESTROY`	also when the service is destroyed

Of note, `LifecycleService` does not attempt to model binding/unbinding as a lifecycle (e.g., calling `ON_STOP` when the service is unbound and has no more active

bindings).

However, most services do not directly inherit from `Service`. Instead, they extend `IntentService` or `JobService` or any one of dozens of other specialized service implementations. Few, if any, of those will extend `LifecycleService`, as most of them come from the core framework, which cannot depend on libraries like the Architecture Components.

ProcessLifecycleOwner

With a name like `ProcessLifecycleOwner`, you might think that this modeled the lifecycle of a process. Then, you quickly realize that this makes little sense, as the only "lifecycle" that a process goes through is creation and termination, and we cannot get control in the latter event.

Instead, `ProcessLifecycleOwner` might better be named `ForegroundLifecycleOwner`. `ProcessLifecycleOwner` models the lifecycle of all activities combined:

- `ON_CREATE` is triggered when the process starts up
- `ON_START` and `ON_RESUME` are triggered when an activity goes through those lifecycle events, and no other activity had been started recently
- `ON_PAUSE` and `ON_STOP` are triggered, after a delay, when an activity goes through those lifecycle events, if another activity is not started and resumed by this time
- `ON_DESTROY` is never triggered

The delay period is 700ms (as of `1.1.1`), so as long as another activity is started and resumed after a prior activity was paused and stopped within 700ms, the *process* has not undergone a lifecycle change, even though those individual activities did.

So, imagine a single-activity app:

- `ON_CREATE` happens right away
- `ON_START` and `ON_RESUME` happen shortly thereafter, assuming that the process is starting because an activity is being displayed
- The user rotates the screen, causing the activity to be destroyed and recreated
- `ON_PAUSE` and `ON_STOP` *do not occur*, because a new activity was started and resumed before the `ProcessLifecycleOwner` delay period elapsed

- `ON_START` and `ON_RESUME` *do not occur*, because we did not move through the paused and stopped lifecycle states, even though the new activity instance did
- The user presses HOME, BACK, or otherwise leaves this activity for another task
- `ON_PAUSE` and `ON_STOP` happen after the delay period, since no activity from this process went through `ON_START` and `ON_RESUME` during that time

Note that this comes at a cost: the `extensions` artifact automatically adds a `<provider>` element to your manifest, one that initializes the `ProcessLifecycleOwner`... even if your app does not use `ProcessLifecycleOwner`. This is simply so `ProcessLifecycleOwner` code can be invoked as soon as your process is started.

The General/ProcessLifecycle sample project has a `LifecycleApplication` that registers itself as an observer of the singleton instance of `ProcessLifecycleOwner` and dumps all the events to Logcat:

```
package com.commonsware.android.recyclerview.videolist;

import android.app.Application;
import android.arch.lifecycle.DefaultLifecycleObserver;
import android.arch.lifecycle.Lifecycle;
import android.arch.lifecycle.LifecycleOwner;
import android.arch.lifecycle.OnLifecycleEvent;
import android.arch.lifecycle.ProcessLifecycleOwner;
import android.support.annotation.NonNull;
import android.util.Log;

public class LifecycleApplication extends Application
  implements DefaultLifecycleObserver {
  @Override
  public void onCreate() {
    super.onCreate();

    ProcessLifecycleOwner.get().getLifecycle().addObserver(this);
  }

  @Override
  public void onCreate(@NonNull LifecycleOwner owner) {
    Log.d(getClass().getSimpleName(), "ON_CREATE");
  }

  @Override
  public void onStart(@NonNull LifecycleOwner owner) {
```

```
    Log.d(getClass().getSimpleName(), "ON_START");
  }

  @Override
  public void onResume(@NonNull LifecycleOwner owner) {
    Log.d(getClass().getSimpleName(), "ON_RESUME");
  }

  @Override
  public void onPause(@NonNull LifecycleOwner owner) {
    Log.d(getClass().getSimpleName(), "ON_PAUSE");
  }

  @Override
  public void onStop(@NonNull LifecycleOwner owner) {
    Log.d(getClass().getSimpleName(), "ON_STOP");
  }

  @Override
  public void onDestroy(@NonNull LifecycleOwner owner) {
    Log.d(getClass().getSimpleName(), "ON_DESTROY");
  }
}
```

(from General/ProcessLifecycle/app/src/main/java/com/commonsware/android/recyclerview/videolist/LifecycleApplication.java)

That `LifecycleApplication` is then registered in the manifest via `android:name` on `<application>`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.recyclerview.videolist"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false" />

  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

  <application
    android:allowBackup="false"
    android:name=".LifecycleApplication"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
```

```
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
  <activity
    android:name=".VideoPlayerActivity"
    android:configChanges="screenSize|smallestScreenSize|screenLayout|orientation"
    android:launchMode="singleTask"
    android:supportsPictureInPicture="true"
    android:theme="@style/Theme.Apptheme.NoActionBar" />

  <receiver android:name=".RemoteActionReceiver" />

 </application>

</manifest>
```

(from General/ProcessLifecycle/app/src/main/AndroidManifest.xml)

The app itself is a clone of one from *The Busy Coder's Guide to Android Development*. It consists of two activities. One shows a list of all videos indexed by the `MediaStore`. The other plays back a selected video using a `VideoView`. And, on Android 8.0+ devices, the video player activity will have a FAB that switches that activity into picture-in-picture mode.

(NOTE: to run this sample, your test device will need 1+ videos)

If you run it, you will see the `ON_CREATE`, `ON_START`, and `ON_RESUME` events logged in rapid succession. And, if you do not press that enticing FAB, and just use the video player in normal mode, `ON_PAUSE` and `ON_STOP` get invoked at normal times, such as when the user navigates to some other task (e.g., presses HOME).

The FAB, though, changes things, as it moves the video player to a floating picture-in-picture (PiP) window.

If you tap the FAB, and do not touch anything else for a bit, you will see `ON_PAUSE`, then `ON_RESUME`, get logged. This is because:

- The PiP window never has the foreground from an input standpoint, and so its activity is paused, but not stopped (as it is still visible)
- The underlying activity is started and resumed, though with a few seconds' delay, for inexplicable reasons

Similarly, if you tap the PiP window, to bring up the controls, you will see `ON_PAUSE` logged, as the list-of-videos activity is paused (it no longer has the foreground input) but the PiP window is *not* resumed (the input is handled by the system UI, not the activity). After a few moments of inactivity, that PiP window will return to its regular

state, and `ON_RESUME` will be logged.

Playing around with the PiP further (e.g., closing it via the X in the corner) allows you to see how PiP mode ties into activity lifecycles.

Wait… Where Are LifecycleProvider and LifecycleReceiver?

A `ContentProvider` has no real "lifecycle". It is called with `onCreate()` when the process starts up… and that's about it. Similarly, a `BroadcastReceiver` is called with `onReceive()`… and that's about it.

As a result, the Architecture Components do not have lifecycle-aware editions of those components.

LiveData and Data Binding

Android's data binding framework offers a way for you to push data into your widgets with less code, by "binding" sources of data to those widgets via special XML attributes in layout resources. The data binding framework was created before the Architecture Components existed, and so originally the data binding framework had no support for `LiveData`. Now, though, it does, and so you can have your UI update itself automatically from `LiveData` sources, if you choose.

In this chapter, we will explore how to set this up.

A Data Binding Recap

Extensive coverage of data binding can be found in *The Busy Coder's Guide to Android Development*. However, here is a brief reminder of what data binding is and how it works.

Note that data binding is not automatically used in an Android project. You need to enable it via `dataBinding { enabled true }` in the `android` closure of your module's `build.gradle` file.

New Layout Resource Structure

Classic Android layout resources are purely a view hierarchy. Typically, the root XML element then is the outermost container class (e.g., a `ConstraintLayout`).

A layout resource that will participate in data binding will instead have a root `<layout>` element. That will have two child elements:

- A `<data>` element with metadata for the data binding

- The outermost container class — what would have been the root element before

Those need to appear in that order, with `<data>` as the first child.

A `<data>` element can have a variety of child elements to configure the data binding, but the most important ones are `<variable>` elements. These declare what objects are being bound to this layout, and they provide both the `name` and the Java/Kotlin `type` of the object:

```
<layout>

  <data>

    <variable
      name="viewModel"
      type="com.commonsware.android.livedata.SensorViewModel" />
  </data>

  <!-- view hierarchy goes here -->
</layout>
```

Binding Expressions

The XML attributes of the views can then reference those variables in binding expressions. Rather than having a simple value (e.g., `android:text="Hello, world!"`), a binding expression contains references to variables (plus `public` fields and methods on those variables) and other view attributes, with a simple expression syntax. The data binding framework identifies these expressions via `@{}` syntax (`android:text="@{viewModel.sensorReading}"`).

Adapters

Sometimes, the data types available to binding expressions do not quite match the data type needed by the XML attribute. For example, `android:text` maps to `setText()` on `TextView`, and that needs either an `int` resource ID or a `CharSequence` (e.g., `String`) value. If you have something else — such as a `SensorLiveData.Event` object — you will need to provide a "binding adapter" to help bridge the gap.

These adapters come in the form of `static` methods annotated with `@BindingAdapter`:

```
@BindingAdapter("android:text")
public static void setLightReading(TextView tv, SensorLiveData.Event event) {
  if (event==null) {
    tv.setText(null);
  }
  else {
    tv.setText(String.format("%f", event.values[0]));
  }
}
```

The annotation argument is the name of the XML attribute to which this adapter applies. So, in this case, if we try binding a `SensorLiveEvent.Data` object into a `TextView` via its `android:text` attribute, this method will be called. It is up to us then to do something useful to fulfill that binding — in this case, we call `setText()` on the `TextView` with a suitable value, based on the `event`.

Binding from Code

Somewhere, though, we need to supply the values for those variables. That comes from using a different way to set up this layout. Rather than using `setContentView()` or a `LayoutInflater` directly, we use a code-generated class, created by the tools associated with the data binding framework. This class not only takes care of inflating the layout for us, but it gives us setter methods to supply the variables.

The name of this code-generated class is based on the name of the layout resource:

- The filename, without the extension, is coverted into CamelCase
- `Binding` is appended

So, `main.xml` turns into `MainBinding`, `activity_main.xml` turns into `ActivityMainBinding`, and so on.

The binding class has a `static` method named `inflate()` that takes a `LayoutInflater` and returns an instance of the binding class (like a factory method would), inflating the underlying layout along the way:

```
MainBinding binding=MainBinding.inflate(getLayoutInflater());
```

It has a `getRoot()` method that returns the root `View` of your view hierarchy, for use with `setContentView()`, `onCreateViewHolder()`, etc. And it has methods for each one of your variables, where the method name is derived from the variable name:

- The variable name is converted to CamelCase
- `set` is appended to the front

So, a `viewModel` variable results in a `setViewModel()` method. Calling this on the binding triggers the evaluation of the binding expressions and populates the attributes of the affected widgets.

Observable Data Sources

If the variables implement the `Observable` interface, then once you have attached the variables' objects to the binding, changes to the variables' values *automatically* re-evaluate the binding expressions. You do not need to do anything else.

For simple primitives, there are a series of `Observable` implementations, such as `ObservableBoolean` and `ObservableInt`. For `String` and arbitrary other objects, there is a generic `ObservableField` that you can use.

We will see an example of this shortly, as this is where `LiveData` starts becoming important.

LiveData Updating Data Binding

Simply put: we want to be able to have `LiveData` update our data binding expressions. So, if we get new data from Room, or new data from a `Sensor`, or anything else that gives us a `LiveData`, we would like to have those changes be reflected in the UI, with as little effort as possible.

There are a few ways of going about this, outlined in the following sections. Each profiles a variation on the same sample app, which itself is a variation on the `Sensor/LiveList` sample from a previous chapter. In this case, the UI is a `TextView` showing the latest ambient light sensor reading. And, since we have covered `ViewModel`, we will use that for holding onto our `SensorLiveData` that is the source of those sensor readings. What varies between the three samples shown in this chapter is how those readings wind up affecting our UI via data binding.

Each of the three variations has the same "cast of characters":

- We have an activity named `MainActivity`
- It has a `main.xml` layout in which we use data binding
- It has the `SensorLiveData` from the `Sensor/LiveList` sample, except that we

no longer track the date of events, since we are not displaying that in the UI
- It has a `SensorViewModel` that holds that `SensorLiveData`

Updating Observables

The Sensor/SimpleBinding sample project uses an `ObservableField` to get the sensor readings into the layout. And, we have our own code to pipe events from the `SensorLiveData` into that `ObservableField`.

The `SensorViewModel` holds onto both the `ObservableField` and the `SensorLiveData`:

```
package com.commonsware.android.livedata;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.databinding.ObservableField;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.support.annotation.NonNull;

public class SensorViewModel extends AndroidViewModel {
  public final SensorLiveData sensorLiveData;
  public final ObservableField<String> sensorReading=new ObservableField<>();

  public SensorViewModel(@NonNull Application app) {
    super(app);

    sensorLiveData=new SensorLiveData(app, Sensor.TYPE_LIGHT,
      SensorManager.SENSOR_DELAY_UI);
  }
}
```

(from Sensor/SimpleBinding/app/src/main/java/com/commonsware/android/livedata/SensorViewModel.java)

We initialize the `SensorLiveData` in the constructor, using the `Application` supplied as an outcome of extending `AndroidViewModel`.

The `main` layout contains two `TextView` widgets: a label and our reading, wrapped in a `ConstraintLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>

  <data>
```

```
    <variable
      name="viewModel"
      type="com.commonsware.android.livedata.SensorViewModel" />
  </data>

  <android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginBottom="8dp"
      android:layout_marginEnd="8dp"
      android:layout_marginStart="8dp"
      android:layout_marginTop="8dp"
      android:text="@string/label_light"
      android:textAppearance="@android:style/TextAppearance.Material.Large"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      app:layout_constraintVertical_bias="0.25" />

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginBottom="8dp"
      android:layout_marginEnd="8dp"
      android:layout_marginStart="8dp"
      android:layout_marginTop="8dp"
      android:text="@{viewModel.sensorReading}"
      android:textAppearance="@android:style/TextAppearance.Material.Large"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      app:layout_constraintVertical_bias="0.75" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

(from Sensor/SimpleBinding/app/src/main/res/layout/main.xml)

We have one data binding variable, named `viewModel`, which is our `SensorViewModel` instance. And, we have one binding expression, populating

`android:text` of the second `TextView` with the `sensorReading` `ObservableField`.

`MainActivity` then glues the other two together:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.ViewModelProviders;
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import com.commonsware.android.livedata.databinding.MainBinding;

public class MainActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    MainBinding binding=MainBinding.inflate(getLayoutInflater());
    SensorViewModel vm=ViewModelProviders.of(this).get(SensorViewModel.class);

    binding.setViewModel(vm);
    setContentView(binding.getRoot());

    vm.sensorLiveData.observe(this, event ->
      vm.sensorReading.set(String.format("%f", event.values[0])));
  }
}
```

(from Sensor/SimpleBinding/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

Here, we:

- `inflate()` the `MainBinding`
- Obtain our `SensorViewModel` from the `ViewModelProviders`
- Attach the `SensorViewModel` to the `MainBinding` by calling the generated `setViewModel()` method
- Supply the root view of the `main` layout to `setContentView()`
- Observe the changes to the `SensorLiveData`, format each sensor reading into a `String` representation, and `set()` that value on the `ObservableField`

The result is that as the `SensorLiveData` reports new readings, they get piped into the `ObservableField`, which triggers an update to the `TextView`.

Binding to LiveData

However, if the point of `Observable` is to provide updates to data to the data binding

framework, and if the point of `LiveData` is to provide updates to data to observers... shouldn't there be a way to make a `LiveData` be `Observable`?

In short: no.

However, that is not needed, because as of 2018, the data binding framework can work with `LiveData` directly. If your binding expressions reference `LiveData` objects, the data binding framework knows to observe those objects and use any updates to re-evaluate the binding expressions.

The only requirement is that we now have to provide a `LifecycleOwner` to our binding. There is a `setLifecycleOwner()` for this. That `LifecycleOwner` is used for observing the `LiveData`, and it should be a `LifecycleOwner` of relevance to the views being managed by the data binding framework. So, for an activity's layout, you would use the activity as the `LifecycleOwner`.

The Sensor/LiveBinding sample project uses this approach.

`SensorViewModel` no longer has the `ObservableField`:

```
package com.commonsware.android.livedata;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.support.annotation.NonNull;

public class SensorViewModel extends AndroidViewModel {
  public final SensorLiveData sensorLiveData;

  public SensorViewModel(
    @NonNull Application app) {
    super(app);

    sensorLiveData=new SensorLiveData(app, Sensor.TYPE_LIGHT,
      SensorManager.SENSOR_DELAY_UI);
  }
}
```

(from Sensor/LiveBinding/app/src/main/java/com/commonsware/android/livedata/SensorViewModel.java)

The binding expression now refers to `sensorLiveData` directly:

```
    <TextView
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginBottom="8dp"
      android:layout_marginEnd="8dp"
      android:layout_marginStart="8dp"
      android:layout_marginTop="8dp"
      android:text="@{viewModel.sensorLiveData}"
      android:textAppearance="@android:style/TextAppearance.Material.Large"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      app:layout_constraintVertical_bias="0.75" />
```

(from Sensor/LiveBinding/app/src/main/res/layout/main.xml)

`onCreate()` of `MainActivity` no longer needs to `observe()` the `SensorLiveData` itself, as the data binding framework will handle that. It does, however, need to call `setLifecycleOwner()`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  MainBinding binding=MainBinding.inflate(getLayoutInflater());
  SensorViewModel vm=ViewModelProviders.of(this).get(SensorViewModel.class);

  binding.setViewModel(vm);
  binding.setLifecycleOwner(this);
  setContentView(binding.getRoot());
}
```

(from Sensor/LiveBinding/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

In some cases, that is all that you will need. In this case, though, there are a couple of additional changes from the previous sample that are needed to make it work.

Our binding expression is attempting to populate the text of a `TextView` with the objects emitted by a `SensorLiveData`. Those are `SensorLiveData.Event` objects. The data binding framework needs the `SensorLiveData` and the `SensorLiveData.Event` classes to be `public`, as otherwise the generated `MainBinding` code cannot compile, since that code resides in a different package than does `SensorLiveData` itself.

Also, the data binding framework has no idea how to take a `SensorLiveData.Event` and use it to populate the text of a `TextView`. That requires a `BindingAdapter`:

```
@BindingAdapter("android:text")
public static void setLightReading(TextView tv, SensorLiveData.Event event) {
  if (event==null) {
    tv.setText(null);
  }
  else {
    tv.setText(String.format("%f", event.values[0]));
  }
}
```

(from Sensor/LiveBinding/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

Simply having this annotated `static` method in the project is sufficient; the data binding framework can find it on its own and know to apply it as needed.

That required `BindingAdapter` means that this project is a bit more complex than the previous one. However, it is cleaner, in that we are no longer needing to manage observing the `LiveData` ourselves. There are fewer places where we can screw up, particularly since a `BindingAdapter` is a `static` method and therefore should not be touching any state beyond whatever parameters are passed in.

Handling Changes to LiveData

The approach in the preceding section works great, so long as your `LiveData` itself is unchanging. It might be emitting new results to the data binding observer, but the `LiveData` object is the same for the entire use of that layout and activity.

But what happens when you need to use a different `LiveData` object?

For example, perhaps you are showing search results for data stored in Room. So, you query your DAO and get a `LiveData` with those search results and show them via data binding. But the search results UI has its own search field, for making a new search. When the user searches, you want to re-query the DAO and show the new results in the same fragment or activity. But now you have a new `LiveData`, and you need to get that applied to the data binding.

One simple answer would be to put the `LiveData` directly in a data binding `<variable>`. This works, so long as you have a custom `LiveData` subclass, as data binding `<variable>` declarations do not support generics. However, you need to be in position to call your variable setter on the binding object each time you get a new `LiveData`. That may or may not be practical, depending on your architecture and where the `LiveData` is coming from.

What would be nice is if we could have data binding use a single `LiveData` object, as in our earlier examples, but have that `LiveData` object emit objects that come from other `LiveData` sources. This is a common technique in RxJava when you have the same sort of situation: some subscribers to an `Observable` happen to need a stable `Observable` for simpler subscription management, but you get new `Observable` objects from some API (e.g., new Retrofit calls). So, you use a `Subject`, such as a `BehaviorSubject` as the stable `Observable`, and you have the on-the-fly `Observables` feed their output into the `Subject` as input.

The equivalent approach with `LiveData` is `MediatorLiveData`. `MediatorLiveData` is designed to take output from another `LiveData` as input, emitting those same objects. Those `LiveData` inputs can come and go, and observers can just observe the `MediatorLiveData`.

The `Sensor/LiveMediator` sample project uses this approach.

The activity, layout, and `SensorLiveData` are the same as in the previous example. What differs is the `SensorViewModel`, and what sort of object is used for the `sensorLiveData` field. Previously, that was the `SensorLiveData` itself. Now, it is a `MediatorLiveData`:

```
package com.commonsware.android.livedata;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.arch.lifecycle.MediatorLiveData;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.support.annotation.NonNull;

public class SensorViewModel extends AndroidViewModel {
  public final MediatorLiveData<SensorLiveData.Event> sensorLiveData=
    new MediatorLiveData<>();

  public SensorViewModel(
    @NonNull Application app) {
    super(app);

    SensorLiveData opLiveData=
      new SensorLiveData(app, Sensor.TYPE_LIGHT, SensorManager.SENSOR_DELAY_UI);

    sensorLiveData.addSource(opLiveData, sensorLiveData::setValue);
  }
}
```

(from Sensor/LiveMediator/app/src/main/java/com/commonsware/android/livedata/SensorViewModel.java)

When we create the `SensorViewModel`, we still create an instance of `SensorLiveData`. But then we add it as a source to the `MediatorLiveData` via `addSource()`.

`addSource()` takes two parameters:

- The `LiveData` to be added as a source, and
- A lambda, method reference, or other `Observer` to say what should happen when events are emitted by that source

In this case, we use a method reference to feed the output from the `SensorLiveData` directly into the `MediatorLiveData`. In other circumstances, you might have a lambda that performs some sort of data conversion.

The behavior of the app overall does not change, because the events emitted by the `SensorLiveData` flow through the `MediatorLiveData` to our UI via data binding and the binding expression.

In this case, this adds no value, and we would be better served by using the previous example. However, suppose that our UI allowed the user to choose a different sensor. Now, we could tell our `SensorViewModel` to connect to a different `SensorLiveData` tied to the newly-selected sensor. We would want to disconnect the old `SensorLiveData`, which would require a call to `removeSource()` on the `MediatorLiveaData`.

We will see this technique used again in an upcoming chapter. There, we schedule multiple `WorkManager` pieces of work, and we want to keep track of the status of each of them, but using a single `LiveData` for our UI, instead of individual `LiveData` objects per piece of work.

The Saved Instance State Situation

Using `LiveData` and `ViewModel` exacerbates a problem with how the data binding framework interacts with the saved instance state `Bundle`.

In addition to data that you might put in that `Bundle`, the built-in `onSavedInstanceState()` logic saves obvious user-mutable state of widgets in the UI to the `Bundle`. So, for example, if the user types something into an `EditText`, or toggles the state of a `Switch`, that information goes in the `Bundle`. If you have matching widgets in the new configuration, the state is applied to those widgets automatically.

Except if you are using the data binding framework, in which case, things get complicated.

The General/DataBindingState sample project illustrates the problem and the workaround.

This app has a trivial UI, consisting mostly of an `EditText`. It uses the data binding framework, so the `main.xml` layout resource has a `<layout>` element and so forth:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <data>

    <variable
      name="model"
      type="com.commonsware.databindingstate.Model" />
  </data>

  <android.support.constraint.ConstraintLayout xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.commonsware.databindingstate.MainActivity">

    <EditText android:id="@+id/title"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:inputType="text"
      android:text="@{model.title}"
      app:layout_constraintLeft_toLeftOf="parent"
      app:layout_constraintRight_toRightOf="parent"
      app:layout_constraintTop_toTopOf="parent" />

  </android.support.constraint.ConstraintLayout>
</layout>
```

(from General/DataBindingState/app/src/main/res/layout/main.xml)

Of note, the `android:text` attribute of the `EditText` has a binding expression, pulling the `title` from a `Model` object. `Model` is as trivial of a model as you can imagine:

```
package com.commonsware.databindingstate;

public class Model {
  public String getTitle() {
    return("Title");
  }
}
```

(from General/DataBindingState/app/src/main/java/com/commonsware/databindingstate/Model.java)

However, we do not actually wind up using that `Model`. In fact, we never bind *anything* to the layout.

One might reasonably expect that this would result in the same flow as if we did not use data binding at all:

- The UI shows our layout
- The user types something into the `EditText`
- The user rotates the screen, and the saved instance state `Bundle` populates the newly-created replacement `EditText` for the new configuration

Instead, if you try it, you will find that what you type in gets lost on a configuration change.

However, you will notice that there is an action bar overflow menu, with an "Apply Workaround" checkable item in it. If you check that, what you type into the `EditText` is properly retained across the configuration change.

The difference is a call to `executePendingBindings()` in `onCreateView()` of the `FormFragment` that is showing our limited UI:

```
@Nullable
@Override
public View onCreateView(LayoutInflater inflater,
                         @Nullable ViewGroup container,
                         @Nullable Bundle savedInstanceState) {
  if (savedInstanceState!=null) {
    savedWorkaround=savedInstanceState.getBoolean(STATE_WORKAROUND);

    if (workaround!=null) {
      workaround.setChecked(savedWorkaround);
    }
  }

  MainBinding binding=MainBinding.inflate(inflater, container, false);

  if (savedWorkaround) {
    binding.executePendingBindings();
  }

  return(binding.getRoot());
}
```

(from General/DataBindingState/app/src/main/java/com/commonsware/databindingstate/FormFragment.java)

Our layout is `main.xml`, so our binding class is `MainBinding`. We call the static `inflate()` method on it to inflate our layout, and use `getRoot()` to return the `View` from `onCreateView()`.

If, however, our checkable `MenuItem` is checked (or, rather, *was* checked in the previous configuration), we call `executePendingBindings()`. Otherwise, we do not. That makes the difference.

(the author would like to thank Stack Overflow user Cheticamp for pointing out the workaround)

This sample is very artificial. Using data binding without binding any data would seem atypical. However, it does illustrate a general problem with data binding that using reactive UIs — `LiveData`, `ViewModel`, etc. — exacerbate. Simply put, when do we bind, to get the correct results?

Let's examine two common cases: the user is editing an existing model object, or the user is creating a new model object.

Existing Model

We inflate our layout using our binding object, plus we observe a `LiveData` to get the model object when it is ready. Once it is ready, we call the appropriate setter on the binding to populate the widgets.

Right?

Unfortunately, not always.

That is what we do at the outset, the first time our activity or fragment is displayed to edit this particular model. However, if we undergo a configuration change, we have a problem: we do not want to lose changes that the user made already. If the user started typing something into an `EditText`, then rotated the screen (e.g., to switch to the landscape keyboard for easier typing), we do not want to lose the changes already made in that `EditText`.

Ordinarily, the saved instance state `Bundle` would handle that... but if we turn around and call our binding setter in the new activity or fragment, we will replace what the user typed in with whatever is in the model object.

There are two main solutions here:

1. Use two-way binding, so the UI immediately updates the model object as the user makes changes to it. If the `ViewModel` holds onto that model object and can give it back to us after the configuration change, we can safely call the

setter method on the binding object, as our model has the appropriate data.

2. Only bind the model object on the initial creation of the activity or fragment, not on configuration changes. Instead, let the normal saved instance state logic handle the form contents. This requires you to call `executePendingBindings()` shortly after inflating the layout via the binding class, so that the saved instance state is applied properly.

New Model

In the new-model scenario, you have the additional question of: is there anything to bind, anyway?

If you are using two-way data binding, then you need to bind something to collect the input from the user. Similarly, if you are using a newly created model object to supply starter data to the form, you will need to create such a model object and bind it. In these cases, the new-model scenario is largely the same as the existing-model scenario, with the primary difference being where the model comes from.

If you have no need for a model at the outset, though, you could skip binding anything, much as the sample app skipped binding anything. Then, so long as you call `executePendingBindings()` shortly after inflating the layout via the binding class, you should be in fine shape.

WorkManager

Hardly a year goes by anymore without some new solution for doing background work becoming available for Android developers. In some cases, the new solution is designed to make things easier. In some cases, the new solution is designed to work around platform-imposed limitations (a.k.a., "The War on Background Processing").

The solution introduced in 2018 was `WorkManager`, and preliminary indications are that it will be Google's "go-to" solution for many background work scenarios.

`WorkManager` is considered to be part of the Architecture Components, despite having only loose connections to the rest of the Components. `WorkManager` does offer ways to monitor work via `LiveData`, for cases where the work happens to be going on while your UI is still visible.

In this chapter, we will explore `WorkManager`, its role, and how to employ it.

NOTE: At the time of this writing, `WorkManager` is in a beta state.

Where Should We Use `WorkManager`?

`WorkManager` is designed for "deferrable" work — work that you need to have done but does not have to happen right away. This includes the possibility that the work will be done sometime after your current process has terminated.

In this respect, `WorkManager` behaves akin to `JobScheduler`, which is the main engine behind `WorkManager` for API Level 23+ devices.

Where Should We *Not* Use `WorkManager`?

`WorkManager` is designed for discrete, "transactional" tasks, not ongoing work. So, for example, `WorkManager` is not designed to play music continuously in the background. A foreground service is the solution to use for that, with background threads as needed (e.g., for disk I/O to read in the playlist details).

`WorkManager` is designed for work that will happen sometime, but not at some specific time. If you need to get control at a specific time — such as to alert the user about an upcoming calendar event — use `setExactAndAllowWhileIdle()` on `AlarmManager`.

`WorkManager` is designed for work that will happen eventually, but perhaps not immediately. If you have background work that has to be done in real time in response to user input (e.g., download the video that they just purchased), use a foreground service.

`WorkManager` is designed for work that might happen completely asynchronously with respect to your current process. Hence, it is not useful for cases where the work that you are doing only affects the current process, particularly its UI. So, for example, downloading avatar icons to display in your app may not make sense once the UI is gone, as you may never need those icons. For that, use a thread pool, reactive solutions (e.g., RxJava), or libraries that in turn use those sorts of things.

`WorkManager` Dependencies

The main artifact that you will use for adding `WorkManager` to your project is `android.arch.work:work-runtime`. This contains `WorkManager` and its related classes.

There are three additional artifacts that you can elect to use:

- `work-runtime-ktx` provides a Kotlin-specific `WorkManager` API
- `work-testing` is available for testing your code that uses `WorkManager`
- `work-firebase` is if your `minSdkVersion` is below 23 and you wish to use Firebase JobDispatcher as the engine for `WorkManager`, as opposed `AlarmManager`, on older devices

Workers: They Do Work

The work that you want to have done in the background needs to be wrapped in a `Worker` subclass. This is an `abstract` class with one `abstract` method: `doWork()`. In that method, you put your work to be done in the background.

Note that:

- `doWork()` is called after the `WorkManager` engine has obtained a wakelock, so you do not need to acquire one yourself to ensure that the work can get done without the device falling asleep
- `doWork()` is called on a background thread, so you do not need to fork one yourself
- `doWork()` needs to return a `ListenableWorker.Result` object indicating if the work succeeded or failed, so `doWork()` should not be starting other background threads (directly or through libraries)
- `doWork()` cannot run forever — at best, it might run for 10 minutes before `JobScheduler` terminates it, and it is possible that it will have less time than that

As noted above, `doWork()` returns a `ListenableWorker.Result` object. There are static factory methods on `ListenableWorker.Result` that you use to create instances. Those factory methods represent three main result scenarios:

- `success()`, which is what you are hoping for
- `retry()`, which indicates that for one reason or another you could not do the work but would like `WorkManager` to retry the work in a little while
- `failure()`, which indicates that the work could not be done and a later retry is likely to fail as well, so you are giving up

Beyond the return value and the aforementioned limitations on what you can do in `doWork()`, the actual business logic is up to you.

For example, in the Work/Download sample project, there is a `DownloadWorker` class that downloads a file using OkHttp and Okio:

```
package com.commonsware.android.work.download;

import android.content.Context;
import android.support.annotation.NonNull;
import android.util.Log;
```

```
import java.io.File;
import java.io.IOException;
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okio.BufferedSink;
import okio.Okio;

public class DownloadWorker extends Worker {
  public static final String KEY_URL="url";
  public static final String KEY_FILENAME="filename";

  public DownloadWorker(@NonNull Context context,
                        @NonNull WorkerParameters workerParams) {
    super(context, workerParams);
  }

  @NonNull
  @Override
  public Result doWork() {
    OkHttpClient client=new OkHttpClient();
    Request request=new Request.Builder()
      .url(getInputData().getString(KEY_URL))
      .build();

    try (Response response=client.newCall(request).execute()) {
      File dir=getApplicationContext().getCacheDir();
      File downloadedFile=
        new File(dir, getInputData().getString(KEY_FILENAME));
      BufferedSink sink=Okio.buffer(Okio.sink(downloadedFile));

      sink.writeAll(response.body().source());
      sink.close();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception downloading file", e);

      return ListenableWorker.Result.failure();
    }

    return ListenableWorker.Result.success();
  }
}
```

(from Work/Download/app/src/main/java/com/commonsware/android/work/download/DownloadWorker.java)

`Worker` — from which `DownloadWorker` inherits — has a two-parameter constructor, taking a `Context` and a `WorkerParameters` object. In many cases, you can just chain to the superclass constructor, as `DownloadWorker` does.

Pretty much everything inside of `doWork()` is just application code that does the download and returns `success()` if the download succeeded or `failure()` if there was some exception during the download.

This `doWork()` method is using two methods that we get from `Worker`:

- `getApplicationContext()`, which works like the similarly-named method on `Context`, returning you the `Application` singleton, in case you need a `Context`
- `getInputData()`, which we will examine more closely later in the chapter

Performing Simple Work

Having a `Worker` is part of the puzzle. We still need to tell `WorkManager` to actually use that class to do our work.

If we want to perform the work once — perhaps in response to user input — we can create a `OneTimeWorkRequest` object to describe that work, then `enqueue()` it with `WorkManager`:

```
OneTimeWorkRequest downloadWork=
  new OneTimeWorkRequest.Builder(DownloadWorker.class)
    .build();

WorkManager.getInstance().enqueue(downloadWork);
```

We create a `OneTimeWorkRequest` via its associated `Builder`, which takes the Java `Class` object for our `Worker` subclass its constructor. We `build()` the `Builder` and pass the `OneTimeWorkRequest` to `enqueue()` on the `WorkManager` singleton, which we get by calling `getInstance()` on `WorkManager`.

After this code executes, at some point in time, an instance of `DownloadWorker` will be created and `doWork()` will be called. Exactly when that will be is indeterminate. In this particular case, *probably* it will be called fairly quickly, with `doWork()` being executed on a thread in a `WorkManager`-managed thread pool. However, it is entirely possible that the user leaves our app and our process is terminated before this work can begin. If so, `JobScheduler`, Firebase JobDispatcher, or `AlarmManager` will arrange

to get that work done later.

Work Inputs

However, `doWork()` would crash if we scheduled it this way. That comes back to those `getInputData()` calls from our `doWork()` method.

Often, our work needs data describing that work. In the case of `DownloadWorker`, we need to know:

- the HTTPS URL to download from
- the name of the file to download to (where the file will be placed in `getCacheDir()`)

`WorkManager` has a solution for this, via the `Data` class (perhaps named after the android character in *Star Trek* properties) (or perhaps not).

`Data` is a key-value store, one that very closely resembles `PersistableBundle`. The values are simple primitives plus arrays of simple primitives. We can package information into a `Data`, attach it to the work request, and then get that information from inside of `doWork()`.

Reading the `Data` is a matter of calling `getInputData()` inside of `doWork()`, then calling getter methods based on type (e.g., `getString()`). Those getter methods take the key under which the data is stored as a parameter. Getters for primitive types (e.g., `getInt()`, `getBoolean()`) also have a second parameter to use for the default response, if there is nothing associated with that key in the `Data`.

Putting `Data` into a request is a matter of creating a `Data` instance using a `Data.Builder`, which contains the corresponding setter methods (e.g., `putString()`):

```
OneTimeWorkRequest downloadWork=
  new OneTimeWorkRequest.Builder(DownloadWorker.class)
    .setInputData(new Data.Builder()
      .putString(DownloadWorker.KEY_URL,
        "https://commonsware.com/Android/Android-1_0-CC.pdf")
      .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
      .build())
    .build();

WorkManager.getInstance().enqueue(downloadWork);
```

Here, we fill in the two values that the `DownloadWorker` is expecting, using `setInputData()` to attach the `Data` to our `OneTimeWorkRequest`.

Note that there is a 10KB limit on the size of the `Data`. `Data` is there mostly to provide identifiers, such as the URL and filename that we are using here. Use the `Data` for unique information, stuff that the `Worker` subclass cannot obtain from other sources (e.g., your database, your `SharedPreferences`).

Constrained Work

Frequently, the work that we want to do has some requirements. For example, in the case of `DownloadWorker`, it helps to have an Internet connection, as otherwise we may not be able to download the content.

`WorkManager` exposes a similar set of constraints as you see with `JobScheduler`. You can constrain your work based on:

- Whether the device has an Internet connection, or perhaps a particular type of Internet connection (e.g., an unmetered connection)
- Whether the device has a decent amount of battery life remaining, or perhaps is on a charger
- Whether the device has a decent amount of storage space available
- Whether the device is idle (so your work is less likely to interfere with the user)

To configure these, we:

- Create a `Constraints.Builder`,
- Call setter methods on that `Builder` to specify our constraints,
- `build()` the `Builder`, and
- Call `setConstraints()` on the request `Builder` to attach the constraints

```
Constraints constraints=new Constraints.Builder()
  .setRequiredNetworkType(NetworkType.CONNECTED)
  .setRequiresBatteryNotLow(true)
  .build();
OneTimeWorkRequest downloadWork=
  new OneTimeWorkRequest.Builder(DownloadWorker.class)
    .setConstraints(constraints)
    .setInputData(new Data.Builder()
      .putString(DownloadWorker.KEY_URL,
        "https://commonsware.com/Android/Android-1_0-CC.pdf")
```

```
      .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
      .build())
    .build();

WorkManager.getInstance().enqueue(downloadWork);
```

Here, we say that we need a network connection (of any type) and that the battery should not be low.

Tagged Work

We can also associate one or more tags with our work requests. We can later get information about our outstanding work based on tags, or cancel work based on tags.

Tags are meant to be used as categories, to identify like pieces of work that we might want to operate on in unison:

- All downloads
- All work associated with some particular database table
- All work associated with some account
- And so on

To add tags, just call `addTag()` one or more times on the request `Builder`:

```
  public void doTheDownload() {
    Constraints constraints=new Constraints.Builder()
      .setRequiredNetworkType(NetworkType.CONNECTED)
      .setRequiresBatteryNotLow(true)
      .build();
    OneTimeWorkRequest downloadWork=
      new OneTimeWorkRequest.Builder(DownloadWorker.class)
        .setConstraints(constraints)
        .setInputData(new Data.Builder()
          .putString(DownloadWorker.KEY_URL,
            "https://commonsware.com/Android/Android-1_0-CC.pdf")
          .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
          .build())
        .addTag("download")
        .build();

    WorkManager.getInstance().enqueue(downloadWork);
```

(from Work/Download/app/src/main/java/com/commonsware/android/work/download/DownloadViewModel.java)

Here, we use `addTag()` to tag this work as `download`.

Monitoring Work

`WorkManager` does not provide a built-in means for you to monitor progress inside of an individual piece of work. It does, however, provide you with an API for monitoring the gross state changes of a piece of work: is it enqueued, is it running, is it completed, etc.

Getting the Status Updates

To find out about the general state changes in the life of a piece of work, you can use `getWorkInfoByIdLiveData()`, available on `WorkManager`. Each request has an ID, generated by the `WorkManager` system, which you get by calling `getId()` on the request:

```
    final LiveData<WorkInfo> liveOpStatus=
      WorkManager.getInstance().getWorkInfoByIdLiveData(downloadWork.getId());
```

(from Work/Download/app/src/main/java/com/commonsware/android/work/download/DownloadViewModel.java)

The `LiveData` that we get back will emit `WorkInfo` updates for the work identified by this ID. A `WorkInfo`, in turn, holds a `State` enum, that indicates what phase of the `WorkManager` process this piece of work is in:

- `ENQUEUED`
- `BLOCKED` (for use with chained work)
- `RUNNING`
- `SUCCEEDED`
- `FAILED`
- `CANCELED` (for use with canceling work)

You can then arrange to observe the `LiveData` or otherwise make use of its updates.

Consuming the Status Updates… In Code

The code shown in this chapter so far that created the `OneTimeWorkRequest` and enqueued the work is in a `DownloadViewModel`:

```
package com.commonsware.android.work.download;

import android.arch.lifecycle.LiveData;
```

```
import android.arch.lifecycle.MediatorLiveData;
import android.arch.lifecycle.ViewModel;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

public class DownloadViewModel extends ViewModel {
  public final MediatorLiveData<WorkInfo> liveWorkStatus=new MediatorLiveData<>();

  public void doTheDownload() {
    Constraints constraints=new Constraints.Builder()
      .setRequiredNetworkType(NetworkType.CONNECTED)
      .setRequiresBatteryNotLow(true)
      .build();
    OneTimeWorkRequest downloadWork=
      new OneTimeWorkRequest.Builder(DownloadWorker.class)
        .setConstraints(constraints)
        .setInputData(new Data.Builder()
          .putString(DownloadWorker.KEY_URL,
            "https://commonsware.com/Android/Android-1_0-CC.pdf")
          .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
          .build())
        .addTag("download")
        .build();

    WorkManager.getInstance().enqueue(downloadWork);

    final LiveData<WorkInfo> liveOpStatus=
      WorkManager.getInstance().getWorkInfoByIdLiveData(downloadWork.getId());

    liveWorkStatus.addSource(liveOpStatus, workStatus -> {
      liveWorkStatus.setValue(workStatus);

      if (workStatus.getState().isFinished()) {
        liveWorkStatus.removeSource(liveOpStatus);
      }
    });
  }
}
```

(from Work/Download/app/src/main/java/com/commonsware/android/work/download/DownloadViewModel.java)

The `doTheDownload()` method will be called when the user clicks a button in the UI of `MainActivity`. That triggers our creation of the work request.

`DownloadViewModel` takes the `MediatorLiveData` approach described in the chapter on `LiveData` and data binding. Consumers of the `DownloadViewModel`, such as our `MainActivity`, have access to a `liveWorkStatus` field that represents the outbound stream of work status updates. For each `doTheDownload()` call, we chain the `LiveData` for this individual download onto the `MediatorLiveData`, removing it as a source once the `State` reaches a terminal condition (`isFinished()`, which will be `true` for a `State` of `SUCCEEDED`, `FAILED`, or `CANCELED`).

The result is that our `MainActivity` can observe `liveWorkStatus`, without having to worry about individual `LiveData` objects from individual download requests.

`MainActivity` observes `liveWorkStatus` and uses it to display a `Toast` when the download is finished:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  final DownloadViewModel vm=ViewModelProviders.of(this).get(DownloadViewModel.class);

  binding=ActivityMainBinding.inflate(getLayoutInflater());
  binding.setViewModel(vm);
  binding.setLifecycleOwner(this);

  setContentView(binding.getRoot());

  vm.liveWorkStatus.observe(this, workStatus -> {
    if (workStatus!=null && workStatus.getState().isFinished()) {
      Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show();
    }
  });
}
```

(from Work/Download/app/src/main/java/com/commonsware/android/work/download/MainActivity.java)

Consuming the Status Updates… In Data Binding

`MainActivity` — and its `activity_main` layout resource — use data binding. Partially, this is to get control to `DownloadViewModel` when the user clicks a button. But we also want to disable the button while the download is going on, to reduce the likelihood of accidentally triggering multiple downloads.

To that end, we bind the `DownloadViewModel` into the binding, as was shown in the chapter on `LiveData` and data binding. The layout then has binding expressions both for `android:onClick` and `android:enabled` on its `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>

  <data>

    <variable
      name="viewModel"
      type="com.commonsware.android.work.download.DownloadViewModel" />
  </data>

  <android.support.constraint.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=".MainActivity">

    <Button
      android:id="@+id/download"
      android:layout_width="0dp"
      android:layout_height="0dp"
      android:text="@string/btn_title"
      android:onClick="@{() -> viewModel.doTheDownload()}"
      android:enabled="@{viewModel.liveWorkStatus }"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

(from Work/Download/app/src/main/res/layout/activity_main.xml)

`android:onClick` just calls `doTheDownload` on the `DownloadViewModel`. `android:enabled` takes advantage of the `LiveData` support in data binding, with the extra assistance of a `BindingAdapter`:

```
@BindingAdapter("android:enabled")
public static void setEnabled(View v, WorkInfo info) {
  if (info==null) {
    v.setEnabled(true);
  }
  else {
    v.setEnabled(info.getState().isFinished());
  }
}
```

(from Work/Download/app/src/main/java/com/commonsware/android/work/download/MainActivity.java)

Here, we map the `State` from a `WorkInfo` to the `boolean` value to use for the `android:enabled` attribute. Basically, if the `WorkInfo` is `null` or is finished, the button is enabled, otherwise it is disabled. So, as the `LiveData` emits new `WorkInfo` objects, data binding takes each, calls this `setEnabled()` method, and uses that to update the enabled state of the `Button`.

Canceling Work

Frequently, the work that we enqueue into the `WorkManager` is "fire and forget", where either the work succeeds or fails on its own. Occasionally, though, we may need to try to cancel a piece of enqueued work. For example, we might offer a cancel button in the UI to allow the user to abandon some enqueued request (e.g., do not download the thing that the user just requested to download).

For that, you can call `cancelWorkById()` or `cancelAllWorkByTag()` on `WorkManager`. The former takes the work's ID (from `getId()` on the `WorkRequest`), while the latter takes a tag. Since IDs are unique, `cancelWorkById()` will only try to cancel that once piece of work, while `cancelAllWorkByTag()` will try to cancel all enqueued work associated with that tag. So, for example, if you associate the `download` tag with your download work requests, `cancelAllWorkByTag("download")` will try to cancel all of those requests.

Note, though, that cancellation is "best effort". In particular, if the work has already begun, it might not be canceled. In some cases, this is fine. In other cases, you might want to both cancel the work in `WorkManager` and take steps to ensure that any affected running work finds out about the cancellation. For example, you might have some state field somewhere that the work can monitor to see if it should continue doing whatever it is doing.

Delayed Work

Typically, we are happy to have our work begin right away, if the conditions allow it. Occasionally, we may want to intentionally delay that work for a bit.

For this, you can call `setInitialDelay()` on the `OneTimeWorkRequest.Builder` as part of configuring the work. There are two flavors of `setInitialDelay()`:

- one takes a `long` value and a `TimeUnit`, where the `TimeUnit` indicates what unit of measure the `long` is in (e.g., `TimeUnit.SECONDS`)
- one takes a `Duration`, which is part of Java 8 and only available on API Level 26 and higher

In either case, the work will be delayed by at least that amount of time. However, depending on circumstances (constraints, Doze mode, etc.), the work might happen substantially later than the delay period. Do not assume that your work will start immediately after your delay period.

Parallel Work

Suppose you have N pieces of work to be done in (approximate) parallel. As it turns out, `enqueue()` on `WorkManager` takes either a varargs of `WorkRequest` or a `List` of `WorkRequest` objects. If you pass in more than one `WorkRequest`, all will be enqueued, and all will run when possible, based upon their constraints and the number of threads in the `WorkManager` thread pool.

WorkManager has a default thread pool, and in many cases it will be sufficient for your needs. If you wish to control that thread pool, call the static initialize() on WorkManager *once*, such as from onCreate() of a custom Application. initialize() takes a Context and a Configuration. You get a Configuration through the builder pattern:

- Create an instance of Configuration.Builder
- Call setExecutor() on that Builder with an Executor that will serve as the thread pool for the WorkManager
- Call build() on the Builder to get the Configuration

Chained Work

Where WorkManager shines in comparison to previous deferrable-task solutions is in its support for chained work. Chained work is where you set up work requests that in turn depend upon other work requests. Later work requests in the chain are only performed if the previous ones succeeded. And, work requests can supply data to the next request in the chain, akin to command-line pipelines or basic workflow systems.

Why?

On the one hand, chained work may not seem necessary. In principle, what you do as a series of work requests could be done in one large work request.

The big benefit of splitting the work into separate requests comes with the application of constraints. For example, the sample app that we will examine demonstrates chained work by downloading a ZIP file, then unZIPping it. Downloading a ZIP file requires an Internet connection, but unZIPping it does not. By providing separate constraints for each work request, you can require a network connection for the download, yet not require it for the unZIP task, thereby allowing that work to proceed even if Internet connectivity is lost.

Also, smaller Worker classes can be made more reusable. One can imagine a library of common Worker classes. Rather than having to write your own CompositeWorker that used several Worker classes, you can simply set up a chain using existing APIs.

Chained work also helps to address the delivery of status updates as a larger task is being processed. AsyncTask offers publishProgress() and onProgressUpdate() to inform users of the task about ongoing progress. WorkManager lacks that sort of

facility. However, each `WorkRequest` in the chain has its own `WorkStatus` that can be monitored via `LiveData`. This way, you can at least get coarse-grained information about how the chain overall is proceeding.

How Do We Chain Work?

To enqueue a `WorkRequest`, we used `enqueue()` on the `WorkManager` instance. In truth, that is a convenience method. This:

```
WorkManager.getInstance().enqueue(request);
```

is really this:

```
WorkManager.getInstance().beginWith(request).enqueue();
```

`beginWith()` returns a `WorkContinuation`. This is an object that knows a `WorkRequest` to process and knows how to be chained.

To have a follow-on `WorkRequest` in a simple two-element chain, call `then()` on the `WorkContinuation` before the terminal `enqueue()` call:

```
WorkManager.getInstance().beginWith(request).then(otherRequest).enqueue();
```

Now, `request` will be processed, and if it succeeds, then (and only then) will `otherRequest` be processed.

How Do We Pass Data Along the Chain?

We provide input to a `WorkRequest` via its `Builder` and `setInputData()`. However, this is input that is created outside the processing of any individual request; it is input that is defined when the chain is defined.

In addition, a `Worker` can provide *output* data to factory methods like `success()` on `ListenableWorker.Result`. Those factory methods take the same sort of `Data` object that `setInputData()` does. The output data can be used in two places:

- If this request has another request chained after it, that later request receives the earlier request's output data as input.
- The output data is available from the `WorkInfo` once the work is finished, so consumers of the `LiveData` status stream can also see the output data.

OK, Where's the Code?

The Work/UnZIP sample project is a variation on the previous example, this time where we have two requests in a chain.

`DownloadWorker` is largely the same as before, with two differences:

1. Rather than receiving a filename as input, it decides what the filename will be, as that will merely serve as a temporary file
2. It passes the path to that file to the next request in the chain via `setOutputData()`

```
package com.commonsware.android.work.download;

import android.content.Context;
import android.support.annotation.NonNull;
import android.util.Log;
import com.commonsware.cwac.security.ZipUtils;
import java.io.File;
import java.io.IOException;
import androidx.work.Data;
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okio.BufferedSink;
import okio.Okio;

public class DownloadWorker extends Worker {
  public static final String KEY_URL="url";
  public static final String KEY_RESULTDIR="resultDir";

  public DownloadWorker(@NonNull Context context,
                        @NonNull WorkerParameters workerParams) {
    super(context, workerParams);
  }

  @NonNull
  @Override
  public Result doWork() {
    OkHttpClient client=new OkHttpClient();
    Request request=new Request.Builder()
      .url(getInputData().getString(KEY_URL))
      .build();
```

```
    File dir=getApplicationContext().getCacheDir();
    File downloadedFile=new File(dir, "temp.zip");

    if (downloadedFile.exists()) {
      downloadedFile.delete();
    }

    try (Response response=client.newCall(request).execute()) {
      BufferedSink sink=Okio.buffer(Okio.sink(downloadedFile));

      sink.writeAll(response.body().source());
      sink.close();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception downloading file", e);

      return ListenableWorker.Result.failure();
    }

    return ListenableWorker.Result.success(new Data.Builder()
      .putString(UnZIPWorker.KEY_ZIPFILE, downloadedFile.getAbsolutePath())
      .build());
  }
}
```

(from Work/UnZIP/app/src/main/java/com/commonsware/android/work/download/DownloadWorker.java)

We now also have an `UnZIPWorker`. This expects two pieces of input: the file to unZIP and the directory to unZIP it into. It uses the CWAC-Security library's `ZipUtils.unzip()` method, as that safely handles possibly-malicious ZIP files (e.g., zip bombs):

```
package com.commonsware.android.work.download;

import android.content.Context;
import android.support.annotation.NonNull;
import android.util.Log;
import com.commonsware.cwac.security.ZipUtils;
import java.io.File;
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class UnZIPWorker extends Worker {
  public static final String KEY_ZIPFILE="zipFile";
  public static final String KEY_RESULTDIR="resultDir";

  public UnZIPWorker(@NonNull Context context,
                     @NonNull WorkerParameters workerParams) {
    super(context, workerParams);
```

```
  }

  @NonNull
  @Override
  public Result doWork() {
    File downloadedFile=new File(getInputData().getString(KEY_ZIPFILE));
    File dir=getApplicationContext().getCacheDir();
    String resultDirData=getInputData().getString(KEY_RESULTDIR);
    File resultDir=new File(dir, resultDirData==null ? "results" : resultDirData);

    try {
      ZipUtils.unzip(downloadedFile, resultDir, 2048, 1024*1024*16);
      downloadedFile.delete();
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception unZIPing file", e);

      return ListenableWorker.Result.failure();
    }

    return ListenableWorker.Result.success();
  }
}
```

(from Work/UnZIP/app/src/main/java/com/commonsware/android/work/download/UnZIPWorker.java)

`DownloadViewModel` now sets up a request chain using both worker classes:

```
package com.commonsware.android.work.download;

import android.arch.lifecycle.LiveData;
import android.arch.lifecycle.MediatorLiveData;
import android.arch.lifecycle.ViewModel;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

public class DownloadViewModel extends ViewModel {
  public final MediatorLiveData<WorkInfo> liveWorkStatus=new MediatorLiveData<>();

  public void doTheDownload() {
    OneTimeWorkRequest downloadWork=
      new OneTimeWorkRequest.Builder(DownloadWorker.class)
        .setConstraints(new Constraints.Builder()
          .setRequiredNetworkType(NetworkType.CONNECTED)
          .setRequiresBatteryNotLow(true)
          .build())
        .setInputData(new Data.Builder()
          .putString(DownloadWorker.KEY_URL,
            "https://commonsware.com/Android/source_1_0.zip")
          .build())
        .addTag("download")
        .build();
    OneTimeWorkRequest unZIPWork=
      new OneTimeWorkRequest.Builder(UnZIPWorker.class)
        .setConstraints(new Constraints.Builder()
```

```
          .setRequiresStorageNotLow(true)
          .setRequiresBatteryNotLow(true)
          .build())
        .setInputData(new Data.Builder()
          .putString(DownloadWorker.KEY_RESULTDIR, "unzipped")
          .build())
        .addTag("unZIP")
        .build();

    WorkManager.getInstance()
      .beginWith(downloadWork)
      .then(unZIPWork)
      .enqueue();

    final LiveData<WorkInfo> liveOpStatus=
      WorkManager.getInstance().getWorkInfoByIdLiveData(unZIPWork.getId());

    liveWorkStatus.addSource(liveOpStatus, workStatus -> {
      liveWorkStatus.setValue(workStatus);

      if (workStatus.getState().isFinished()) {
        liveWorkStatus.removeSource(liveOpStatus);
      }
    });
  }
}
```

(from Work/UnZIP/app/src/main/java/com/commonsware/android/work/download/DownloadViewModel.java)

Of note:

- `downloadWork` is defined the same as before, except that we skip supplying the filename, and the URL now points to a ZIP file instead of a PDF
- `unZIPWork` does not require an Internet connection, but it does require that we have a reasonable amount of storage available
- `unZIPWork` gets the name of a directory to create in `getCacheDir()` to hold the unZIPped results
- We use `beginWith()` and `then()` to set up the chain, using `enqueue()` to enqueue the results
- We monitor the `unZIPWork` status for the purposes of re-enabling the button and showing the `Toast`

In principle, we should be monitoring *both* requests' status updates. If the first request fails for some reason (e.g., HTTP 404 error), the second request will never run. We could do that by calling `getWorkInfosLiveData()` on the `WorkContinuation`, which returns a `LiveData` of a *list* of `WorkInfo` objects, one for each request in the chain. That significantly increases the complexity of the sample (e.g., what do we do for data binding in this case?), and so we cheat for the sake of brevity.

How Complex Can This Get?

It can get as complicated as you like:

- You can keep chaining work together by successive `then()` calls:

```
WorkManager.getIntstance().beginWith(lets).then(go).then(crazy).enqueue();
```

- You can have parallel requests as part of a chain, by passing multiple `WorkRequest` objects to `beginWith()` or `then()`
- You can chain a `WorkContinuation` onto another `WorkContinuation`
- You can create `InputMerger` implementations to help coordinate out the output data from previous steps in the chain are merged together to form the input data for successive steps in the chain
- And so on

However, while `WorkManager` is useful for deferrable tasks, it is not a full workflow system:

- There is limited ability to cancel work, as noted previously
- There is no ability to change enqueued work, except by trying to cancel it and then enqueuing its replacement
- There are no specifications for how long any individual request or an entire chain can take, in terms of time
- There are no specifications for how results are handled when it takes multiple process invocations to complete a chain (e.g., a long chain extending past the 10-minute `JobScheduler` limit)
- And so on

As a result, at least for the time being, be careful when trying to create complex `WorkRequest` chains.

Periodic Work

So far, all of the work has been for single tasks, using `OneTimeWorkRequest`. The other `WorkRequest` implementation is `PeriodicWorkRequest`, and as the name suggests, it is for work that should repeat with a given interval.

The interval is provided via the `PeriodicWorkRequest.Builder` constructor, either as a `Duration` or as a `long` and `TimeUnit` pair. There is a minimum allowed period,

defined as `PeriodicWorkRequest.MIN_PERIODIC_INTERVAL_MILLIS`, presently defined as 15 minutes.

However, bear in mind that `WorkManager` is implemented using `JobScheduler` on newer Android devices. On those devices, Doze mode and app standby will affect your periodic or delayed work. Hence, consider the supplied interval to be a suggestion, more than a requirement.

Unique Work

Sometimes, we want to avoid accidentally enqueuing the same work more than once.

For example, suppose that the app has a `Worker` that pulls data from a server. Normally, that is triggered by a push message (e.g., Firebase Cloud Messaging). However, you also have it set up that user actions can trigger that work to be done, such as via a manual refresh option (e.g., pull-to-refresh). What you want to avoid is trying to do two of this bit of work simultaneously, as that might confuse things on either the device side or the server side.

To handle this, instead of `enqueue()` on `WorkManager`, you can use `beginUniqueWork()`. This takes a "name" that identifies this logical unit of work. If you previously used `beginUniqueWork()`, and you later call `beginUniqueWork()` with the same name, and the earlier work is still ongoing, you can specify what should happen (e.g., ignore the new work request).

Note that this does not support periodic work: you can only coordinate `OneTimeWorkRequest`, not `PeriodicWorkRequest`.

Testing Work

The `work-testing` artifact offers a `WorkManagerTestInitHelper` utility class to help with instrumented testing.

First, it has `initializeTestWorkManager()`. This configures `WorkManager` to use a `SynchronousExecutor`. This amounts to a mock `Executor`, one that runs supplied `Runnable` objects immediately on the current thread. By using `SynchronousExecutor`, your `enqueue()` calls for `WorkManager` will happen immediately and synchronously, rather than asynchronously.

Also, `WorkManagerTestInitHelper` has a `getTestDriver()` method, which returns a `TestDriver`. This offers a `setAllConstraintsMet()` method, which takes a work request ID and tells `WorkManager` that all of the constraints for that work request are met. This makes your tests more deterministic, since constraints are normally there to test the environment, and that might change from run to run of your tests. However, it is very important to call `setAllConstraintsMet()` **after** you `enqueue()` the work:

```
WorkManager.getInstance().enqueue(work);
WorkManagerTestInitHelper.getTestDriver().setAllConstraintsMet(work.getId());
```

Note: calling `setAllConstraintsMet()` before calling `enqueue()` results in a crash.

The `Work/Download` sample app contains a `DownloadWorkerTest` class that shows the use of `WorkManagerTestInitHelper` and `TestDriver`:

```
package com.commonsware.android.work.download;

import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import java.io.File;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkManager;
import androidx.work.WorkRequest;
import androidx.work.testing.WorkManagerTestInitHelper;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

@RunWith(AndroidJUnit4.class)
public class DownloadWorkerTest {
  private File expected;

  @Before
  public void setUp() {
    WorkManagerTestInitHelper
      .initializeTestWorkManager(InstrumentationRegistry.getTargetContext());

    expected=
      new File(InstrumentationRegistry.getTargetContext().getCacheDir(),
        "oldbook.pdf");

    if (expected.exists()) {
      expected.delete();
    }
  }

  @Test
```

```
  public void download() {
    assertFalse(expected.exists());

    WorkManager.getInstance().enqueue(buildWorkRequest(null));

    assertTrue(expected.exists());
  }

  @Test
  public void downloadWithConstraints() {
    Constraints constraints=new Constraints.Builder()
      .setRequiredNetworkType(NetworkType.CONNECTED)
      .setRequiresBatteryNotLow(true)
      .build();
    WorkRequest work=buildWorkRequest(constraints);

    assertFalse(expected.exists());

    WorkManager.getInstance().enqueue(work);
    WorkManagerTestInitHelper.getTestDriver().setAllConstraintsMet(work.getId());

    assertTrue(expected.exists());
  }

  private WorkRequest buildWorkRequest(Constraints constraints) {
    OneTimeWorkRequest.Builder builder=
      new OneTimeWorkRequest.Builder(DownloadWorker.class)
        .setInputData(new Data.Builder()
          .putString(DownloadWorker.KEY_URL,
            "https://commonsware.com/Android/Android-1_0-CC.pdf")
          .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
          .build())
        .addTag("download");

    if (constraints!=null) {
      builder.setConstraints(constraints);
    }

    return builder.build();
  }
}
```

(from Work/Download/app/src/androidTest/java/com/commonsware/android/work/download/DownloadWorkerTest.java)

This class tests `DownloadWorker` both with and without constraints, validating that the output file exists after the work has been done. Since we are using the synchronous test configuration of `WorkManager`, we can test this work without having to resort to `CountDownLatch` or similar tricks for testing multithreaded code.

Independent of `work-testing`, note that `Worker` has some dependencies on `Context`, and it may be difficult to mock that `Context` since you are not the one providing it. It may be necessary to consider your `Worker` as something to be tested with instrumented tests. If you wish to have deferred tasks be unit tested outside of Android, consider isolating that logic in another class that your `Worker` then

happens to use.

`WorkManager` and Side Effects

`WorkManager` has a fairly clean and easy API and hides a lot of the complexity of scheduling background work that is not time-sensitive.

However, it has side effects.

To be able to restart your scheduled work after a reboot, `WorkManager` registers an `ACTION_BOOT_COMPLETED` receiver named `androidx.work.impl.background.systemalarm.RescheduleReceiver`. To be a good citizen, `WorkManager` only enables that receiver when you have relevant work and disables it otherwise. That way, your app does not unnecessarily slow down the boot process if there is no reason for your app to get control at boot time.

However, enabling and disabling a component, such as a receiver, triggers an `ACTION_PACKAGE_CHANGED` broadcast. Few apps directly have any code that watches for this broadcast, let alone would be harmed by having that broadcast be sent more times that might otherwise be necessary.

App widgets, though, *are* affected by `ACTION_PACKAGE_CHANGED`. Specifically, `ACTION_PACKAGE_CHANGED` triggers an `onUpdate()` call to your `AppWidgetProvider`.

That too may not be a problem for most app widgets. Ideally, your `AppWidgetProvider` makes no assumptions about when, or how frequently, it gets called with `onUpdate()`. However, there is one area where this *is* a problem: with an `AppWidgetProvider` scheduling work in `onUpdate()`. The flow then becomes:

- A "regular" `onUpdate()` call comes in
- Your `AppWidgetProvider` schedules some work with `WorkManager`
- `WorkManager` enables `RescheduleReceiver`
- That triggers `ACTION_PACKAGE_CHANGED`, which triggers an `onUpdate()` call
- Your `AppWidgetProvider` schedules some work with `WorkManager` again
- `WorkManager` eventually gets through those two pieces of work
- `WorkManager` disables `RescheduleReceiver`, since it is no longer needed
- That triggers `ACTION_PACKAGE_CHANGED`, which triggers an `onUpdate()` call
- Your `AppWidgetProvider` schedules some work with `WorkManager`
- `WorkManager` enables `RescheduleReceiver`
- And we're in an infinite loop

The recommendation from Google is to avoid unconditionally scheduling work with `WorkManager` from `onUpdate()`. Instead, only do it if you know that the work is needed and that it is safe to do so, meaning that you will not get into the infinite loop.

That advice may be difficult for some to implement.

This, and any other possible side-effects of `WorkManager`, are not documented. So, you need to be a bit careful about your use of `WorkManager`:

- If your app responds to `ACTION_PACKAGE_CHANGED` broadcasts, directly or indirectly, it may not be safe to schedule work there, lest you wind up in the infinite loop scenario described above.
- If your app responds to `ACTION_BATTERY_OK`, `ACTION_BATTERY_LOW`, `ACTION_POWER_CONNECTED`, `ACTION_POWER_DISCONNECTED`, `ACTION_DEVICE_STORAGE_LOW`, `ACTION_DEVICE_STORAGE_OK`, `CONNECTIVITY_CHANGE`, `ACTION_TIME_SET`, or `ACTION_TIMEZONE_CHANGED`, bear in mind that `WorkManager` has receivers for those broadcasts in your app. These are all disabled at the outset, but presumably `WorkManager` has code to enable them based on certain conditions, such as certain constraints that you set in your work requests. Be careful about scheduling work with `WorkManager` on those broadcasts as well.
- If your app responds to `ACTION_BOOT_COMPLETED` broadcasts, bear in mind that `WorkManager` also depends on this broadcast. Your respective receivers might be invoked in any order. It may not be safe to schedule work here, as `WorkManager` might assume that its own `ACTION_BOOT_COMPLETED` receiver has completed its work by the time you try scheduling new work. While I would not expect an infinite loop scenario, this is the sort of edge case that requires a lot of testing to ensure everything will work as expected.
- `WorkManager` has a `ContentProvider` that it bakes into your app as well. While scheduling your own work from `onCreate()` of a `ContentProvider` would be rather odd, it's possible that somebody might want to do that. Be careful, as `WorkManager` may not be fully ready for operation at that point. Note that, last time I tested it, all `ContentProvider` instances are created before `onCreate()` of `Application`, so *probably* it is safe to schedule work there.

There may be other edge and corner cases beyond these. So, while `WorkManager` is nice, make sure that you thoroughly test your use of it.

Intermediate Topics

M:N Relations in Room

For 1:1 relations, one entity has a foreign key back to the other entity.

For 1:N relations, one entity has a foreign key back to the other entity. In other words, 1:1 is simply 1:N for a specific small value of N.

In SQL, implementing M:N relations requires a join table of some form, where the join table has foreign keys back to the entities being related. Room, using SQL at its core, does not change this. And since Room does not model relations, but only foreign keys, to create an M:N relation, you have to create a "join entity" that winds up creating the associated join table.

In this chapter, we will take a look at how that is accomplished. Along the way, we will also look at other Room tidbits, such as how to use static classes as entities.

Implementing a Join Entity

The `General/RoomMN` sample project demonstrates an M:N relation. From earlier chapters, we have a `Customer` entity and a `Category` entity. Previously, those were unrelated. Now, let's implement an M:N relation between them, so a `Customer` can be a member of zero or more categories, and a `Category` can have zero or more customers.

Note that we are retaining the tree structure for `Category` used previously. For the purposes of this chapter, we are ignoring that, considering a customer to belong to a category only via a direct relationship. So for example, if customer `Foo` belongs to category `Child`, which has a parent category `Parent`, `Foo` is not a member of `Parent`. The tree structure simply organizes categories, without impacting customers.

Static Entity Classes

Much of the time, your entity classes will be standard, top-level Java classes. Sometimes, though, you might have some utility class that you would rather have as a static class, nested inside something else. For example, in the case of a join entity, perhaps you might want to tuck it inside of one of the entities being joined, just to reduce the clutter of your namespace.

Fortunately, this works, albeit with a wrinkle.

In the sample project, the `Customer` class — which itself is an entity — has a `static` class named `CategoryJoin` that will serve as the join entity:

```
package com.commonsware.android.room.dao;

import android.arch.persistence.room.Embedded;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
import java.util.Date;
import java.util.Set;
import java.util.UUID;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(indices={@Index(value="postalCode", unique=true)})
class Customer {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String postalCode;
  public final String displayName;
  public final Date creationDate;

  @Embedded
  public final LocationColumns officeLocation;

  public final Set<String> tags;

  @Ignore
  Customer(String postalCode, String displayName, LocationColumns officeLocation,
           Set<String> tags) {
    this(UUID.randomUUID().toString(), postalCode, displayName, new Date(),
      officeLocation, tags);
  }

  Customer(String id, String postalCode, String displayName, Date creationDate,
           LocationColumns officeLocation, Set<String> tags) {
    this.id=id;
    this.postalCode=postalCode;
    this.displayName=displayName;
```

```
    this.creationDate=creationDate;
    this.officeLocation=officeLocation;
    this.tags=tags;
  }

  @Entity(
    tableName="customer_category_join",
    primaryKeys={"categoryId", "customerId"},
    foreignKeys={
      @ForeignKey(
        entity=Category.class,
        parentColumns="id",
        childColumns="categoryId",
        onDelete=CASCADE),
      @ForeignKey(
        entity=Customer.class,
        parentColumns="id",
        childColumns="customerId",
        onDelete=CASCADE)},
    indices={
      @Index(value="categoryId"),
      @Index(value="customerId")
    }
  )
  public static class CategoryJoin {
    @NonNull public final String categoryId;
    @NonNull public final String customerId;

    public CategoryJoin(String categoryId, String customerId) {
      this.categoryId=categoryId;
      this.customerId=customerId;
    }
  }
}
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

Room is perfectly content to work with this class, so long as you also register it with your `RoomDatabase` via its `@Database` annotation:

```
@Database(
  entities={Customer.class, Category.class, Customer.CategoryJoin.class},
  version=1
)
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/StuffDatabase.java)

However, note that this is a `static` class. Room will not be able to work with a non-`static` nested class, as only instances of the outer class can create instances of the nested class.

Also, note that the default table name is based on the plain class name. In this case, the default table name is `CategoryJoin`. The outer class name (`Customer`) is not added into the table name. Normally, this will not be a problem, and you might be

renaming the table anyway. However, where you can get tripped up is if you decided to have two (or more) classes with the same name, such as having `CategoryJoin` inside both `Customer` and some other entity. Then, you would wind up with two entity classes both trying to define the same table name by default, and Room will not like that very much.

Foreign Keys and Indices

Let's take a closer look at the `@Entity` annotation on `Customer.CategoryJoin`:

```
@Entity(
  tableName="customer_category_join",
  primaryKeys={"categoryId", "customerId"},
  foreignKeys={
    @ForeignKey(
      entity=Category.class,
      parentColumns="id",
      childColumns="categoryId",
      onDelete=CASCADE),
    @ForeignKey(
      entity=Customer.class,
      parentColumns="id",
      childColumns="customerId",
      onDelete=CASCADE)},
  indices={
    @Index(value="categoryId"),
    @Index(value="customerId")
  }
)
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/Customer.java)

Here, we declare four properties.

`tableName` renames the table to something that is more unique to this situation, incorporating both "customer" and "category" in the name. That way, if we do wind up with `CategoryJoin` elsewhere, we can avoid table name collisions.

`primaryKeys` is used, instead of `@PrimaryKey`, because we need a composite key. The uniqueness is determined by the combination of the IDs of the `Customer` and `Category`, held in `customerId` and `categoryId` columns, respectively.

A join entity will need foreign keys back to both entities that it is joining. So, here, we have two `@ForeignKey` annotations for the `foreignKeys` property, connecting to

both `Customer` and `Category` by their respective IDs. We also use `onDelete=CASCADE`, so if the parent entity (`Customer` or `Category`) is deleted, we also delete all join entities associated with that parent.

And, since Room does not automatically add indices for foreign key columns, we add them ourselves, so we can rapidly find all of the join entity instances for a given `Customer` or `Category`.

Implementing DAO Methods

In addition to setting up the join entity, we need to leverage it in our DAO. Otherwise, the join entity is pointless.

Adding and Removing Relations

In many ORMs, where relations are directly implemented on model objects, you connect objects by direct manipulation. In our case, a `Customer` might have `addCategory()` and `removeCategory()` methods, and `Category` might have `addCustomer()` and `removeCustomer()`.

Since Room models foreign keys, not relations, that's not how we connect a `Customer` and a `Category`. Instead, we do it much the same way as you would with plain SQL: `@Insert` and `@Delete` `Customer.CategoryJoin` instances representing a particular customer-category connection.

And, to that end, we have suitable DAO methods for this:

```
@Insert
void insert(Customer.CategoryJoin... joins);

@Delete
void delete(Customer.CategoryJoin... joins);
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

And, to connect a specific `Customer` instance to a specific `Category` instance, we set up the `Customer.CategoryJoin` instance and `insert()` it:

```
  tags.add("sculpture");
  tags.add("bronze");
  tags.add("slow-pay");

  final LocationColumns loc=new LocationColumns(40.7047282, -74.0148544);
```

```
final Customer firstCustomer=new Customer("10001", "Fearless Girl", loc, tags);

tags.remove("slow-pay");
tags.add("large");

final Customer secondCustomer=new Customer("10002", "Charging Bull", loc, tags);

store.insert(firstCustomer, secondCustomer);

final Category root=new Category("Root!");
final Category child=new Category("Child!", root.id);

store.insert(root, child);

final Customer.CategoryJoin join=
  new Customer.CategoryJoin(root.id, secondCustomer.id);

store.insert(join);
```

(from General/RoomMN/stuff/src/androidTest/java/com/commonsware/android/room/dao/DaoTests.java)

Fetching Via the Join

If an ORM offers `addCategory()` and `removeCustomer()` methods, presumably that ORM also offers `getCategories()` on `Customer` and `getCustomers()` on `Category`, to identify the members of a relation with a specific entity.

Again, Room does not work that way.

Instead, we crack open our SQL syntax reference and craft an `INNER JOIN` ourselves, to use in a `@Query` method:

```
@Query("SELECT categories.* FROM categories\n"+
  "INNER JOIN customer_category_join ON categories.id=customer_category_join.categoryId\n"+
  "WHERE customer_category_join.customerId=:customerId")
List<Category> categoriesForCustomer(String customerId);

@Query("SELECT Customer.* FROM Customer\n"+
  "INNER JOIN customer_category_join ON Customer.id=customer_category_join.customerId\n"+
  "WHERE customer_category_join.categoryId=:categoryId")
List<Customer> customersForCategory(String categoryId);
```

(from General/RoomMN/stuff/src/main/java/com/commonsware/android/room/dao/StuffStore.java)

Here we have methods that return the members of a specific relation, so we can find the categories for a `Customer` or the customers for a `Category`. And the DAO methods return sensible data types. But, it is our job to set up the SQL.

So, in the case of `categoriesForCustomer()`, our SQL:

- Retrieves all columns from the `categories` table...
- ...where we `JOIN` with `customer_category_join` based on the IDs...
- ...and find all those where the join entity points to a specific customer ID

Where's That Good Ol' Object Feel?

By this point, some of you may be wanting to dismiss Room outright, as being too thin of a wrapper around the SQL. Certainly, Room has, um, room for improvement.

However, a lot of the pain may come from what you are thinking that entities represent. Many developers, particularly those using ORMs in other environments, will think of entities as being model objects.

That's not the best approach with Room.

Instead, consider entities to be more akin to data transfer objects (DTOs). They are a means of getting data from point (SQLite) to point (your application code), and not much more.

For example, pretend that the SQLite database was on a server somewhere, and you wrapped it in a Web service which you accessed from your Android app via Retrofit or some similar library. Developers are used to thinking of the POJOs that you might get back from a REST call to be DTOs, objects that model the Web service response, not necessarily modeling any business logic within the app.

Room is much the same. The entities are DTOs from the relational data store to your app, but may or may not line up with how you would want to represent that data in memory as "real" model objects. So, just as you sometimes convert the Retrofit response object graph into something more useful, you sometimes convert the Room response POJOs into something more useful.

Consider the DAO and the entities to be a low-level API, much as you might consider Retrofit or other REST access layers. If you need a richer object representation of your data, wrap the DAO and entities in some sort of repository object, one that knows more about your app's needs and can perform the conversions as needed. That repository can also handle details like transactions, to keep your business logic clean from any details about how the data storage is accomplished. The ultimate goal would be to replace one repository implementation (e.g., using Room) with another (e.g., using Realm or Couchbase Mobile or some non-SQL solution), without having to change anything related to the business logic

itself.

Polymorphic Room Relations

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 involved in 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 them, with an eye towards how they can be implemented with Room.

Polymorphism With Separate Tables

One approach has each concrete type be stored in its own table. So, for example if we have a `Comment` class and a `Link` class, and they both implement a common `Note` interface, we wind up with dedicated tables for `Comment` and `Link`. 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.

The `Trips/RoomPoly` sample project employs this strategy. This is another riff on the trip-tracking sample app shown elsewhere in this book.

As depicted in the preceding paragraph, we have a common `Note` interface:

```
package com.commonsware.android.room;

public interface Note {
  String tripId();
}
```

(from Trips/RoomPoly/app/src/main/java/com/commonsware/android/room/Note.java)

We also have `Comment` and `Link` classes that implement that interface and have slightly different contents:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
import java.util.UUID;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="comments",
  foreignKeys=@ForeignKey(
    entity=Trip.class,
    parentColumns="id",
    childColumns="tripId",
    onDelete=CASCADE),
  indices=@Index("tripId"))
public class Comment implements Note {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String text;

  @NonNull public final String tripId;

  public Comment(@NonNull String id, String text, @NonNull String tripId) {
    this.id=id;
    this.text=text;
    this.tripId=tripId;
  }

  @Ignore
  public Comment(String text, @NonNull Trip trip) {
    this(UUID.randomUUID().toString(), text, trip.id);
```

```
  }

  @Override
  public String tripId() {
    return tripId;
  }
}
```

(from Trips/RoomPoly/app/src/main/java/com/commonsware/android/room/Comment.java)

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Ignore;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
import android.support.annotation.NonNull;
import java.util.UUID;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="links",
  foreignKeys=@ForeignKey(
    entity=Trip.class,
    parentColumns="id",
    childColumns="tripId",
    onDelete=CASCADE),
  indices=@Index("tripId"))
public class Link implements Note {
  @PrimaryKey
  @NonNull
  public final String id;

  public final String title;

  @NonNull public final String url;
  @NonNull public final String tripId;

  public Link(@NonNull String id, String title, @NonNull String url, @NonNull
    String tripId) {
    this.id=id;
    this.title=title;
    this.url=url;
    this.tripId=tripId;
  }

  @Ignore
```

```
  public Link(String title, @NonNull String url, @NonNull Trip trip) {
    this(UUID.randomUUID().toString(), title, url, trip.id);
  }

  @Override
  public String tripId() {
    return tripId;
  }
}
```

(from Trips/RoomPoly/app/src/main/java/com/commonsware/android/room/Link.java)

So, from a database schema standpoint, we have a 1:N relation between `Trip` and `Comment`, and we have a separate 1:N relation between `Trip` and `Link`. However, some of the rest of our Java code might want to think of that as a unified 1:N relation between `Trip` and `Note`, not worrying about the differences between `Comment` and `Link`. After all, we implemented that interface for some good reason (though in this sample, the reason is "to have a common interface to use for illustration purposes").

The project still has a `TripStore` DAO, with a concrete implementation code-generated by Room. In this project, though, `TripStore` is an `abstract` class, not an interface, the way most versions of the "trips" sample are set up.

We can have the same sort of basic CRUD methods for `Comment` and `Link` as we would with any other 1:N Room relation:

```
  /*
    Comment
   */

  @Query("SELECT * FROM comments WHERE tripId=:tripId")
  abstract List<Comment> findCommentsForTrip(String tripId);

  @Insert
  abstract void insert(Comment... comments);

  @Update
  abstract void update(Comment... comments);

  @Delete
  abstract void delete(Comment... comments);

  /*
    Link
   */

  @Query("SELECT * FROM links WHERE tripId=:tripId")
  abstract List<Link> findLinksForTrip(String tripId);

  @Override
  @Query("SELECT * FROM links WHERE tripId=:tripId")
```

```
public abstract DataSource.Factory<Integer, Link> pagedStuffForTrip(String tripId);

@Insert
abstract void insert(Link... comments);

@Update
abstract void update(Link... comments);

@Delete
abstract void delete(Link... comments);
```

(from Trips/RoomPoly/app/src/main/java/com/commonsware/android/room/TripStore.java)

But... what if we want to work with `Note`, ignoring the differences between `Comment` and `Link`?

For that, we need to write some custom DAO methods, using `@Transaction` to ensure that they are transactional:

```
@Transaction
List<Note> findNotesForTrip(String tripId) {
  ArrayList<Note> result=new ArrayList<>();

  result.addAll(findCommentsForTrip(tripId));
  result.addAll(findLinksForTrip(tripId));

  return result;
}

@Transaction
void insert(Note... notes) {
  for (Note note : notes) {
    if (note instanceof Comment) {
      insert((Comment)note);
    }
    else if (note instanceof Link) {
      insert((Link)note);
    }
    else {
      throw new IllegalArgumentException("Um, wut dis? "+note.getClass().getCanonicalName());
    }
  }
}

@Transaction
void update(Note... notes) {
  for (Note note : notes) {
    if (note instanceof Comment) {
      update((Comment)note);
    }
    else if (note instanceof Link) {
      update((Link)note);
    }
    else {
      throw new IllegalArgumentException("Um, wut dis? "+note.getClass().getCanonicalName());
    }
  }
```

```
}

@Transaction
void delete(Note... notes) {
  for (Note note : notes) {
    if (note instanceof Comment) {
      delete((Comment)note);
    }
    else if (note instanceof Link) {
      delete((Link)note);
    }
    else {
      throw new IllegalArgumentException("Um, wut dis? "+note.getClass().getCanonicalName());
    }
  }
}
```

(from Trips/RoomPoly/app/src/main/java/com/commonsware/android/room/TripStore.java)

Retrieval — in the form of `findNotesForTrip()` — simply calls `findCommentsForTrip()` and `findLinksForTrip()` and concatenates their results.

Mutation — `insert()`, `update()`, and `delete()` — check each `Note` to see what type it is and casts to perform the "real" `insert()`, `update()`, or `delete()` for the concrete type.

So, this approach optimizes for the per-concrete-type behaviors, with wrappers to try to homogenize access when dealing with the `Note` abstraction.

Can I Join a `UNION`?

You might think that we could create `findNotesForTrip()` 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 WHERE tripId=:tripId UNION ALL SELECT * FROM comments
WHERE tripId=:tripId")
List<Note> findNotesForTrip(String tripId);
```

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 `Link` and `Comment` 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 `Link` objects and that rows that are comments should be turned into `Comment` 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, if having multiple tables makes you concerned.

The `Trips/RoomPolySingle` sample project employs this strategy. It has the same structure for `Comment`, `Link`, and `Note`, but this time `Note` is the `@Entity`:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;
```

```
import android.support.annotation.NonNull;
import static android.arch.persistence.room.ForeignKey.CASCADE;

@Entity(
  tableName="notes",
  foreignKeys=@ForeignKey(
    entity=Trip.class,
    parentColumns="id",
    childColumns="tripId",
    onDelete=CASCADE),
  indices=@Index("tripId"))
public class Note {
  public enum Type {
    COMMENT(0),
    LINK(1);

    private final int value;

    Type(int value) {
      this.value=value;
    }

    public int value() {
      return value;
    }
  }

  @PrimaryKey
  @NonNull
  public final String id;

  public final String title;
  public final String url;
  @NonNull public final String tripId;
  public final Type type;

  public Note(@NonNull String id, String title, @NonNull String url,
              @NonNull String tripId, Type type) {
    this.id=id;
    this.title=title;
    this.url=url;
    this.tripId=tripId;
    this.type=type;
  }
}
```

(from Trips/RoomPolySingle/app/src/main/java/com/commonsware/android/room/Note.java)

In addition to the fields from `Link` and `Comment`, we also have a `type` field, housing an `enum` that indicates whether this `Note` is a `LINK` or `COMMENT`. This requires `@TypeConverter` methods, in this case added to the existing `TypeTransmogrifier` class:

```
  @TypeConverter
  public static Integer fromType(Note.Type type) {
    return type.value();
  }

  @TypeConverter
  public static Note.Type toType(Integer value) {
    return value==0 ? Note.Type.COMMENT : Note.Type.LINK;
  }
```

(from Trips/RoomPolySingle/app/src/main/java/com/commonsware/android/room/TypeTransmogrifier.java)

`Comment` is a subclass of `Note`, using the `title` field to hold the comment text:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Ignore;
import android.support.annotation.NonNull;
import java.util.UUID;

public class Comment extends Note {
  public Comment(@NonNull String id, String title, String url,
                 @NonNull String tripId, Type type) {
    super(id, title, url, tripId, Type.COMMENT);
  }

  @Ignore
  public Comment(String text, @NonNull Trip trip) {
    this(UUID.randomUUID().toString(), text, null, trip.id, Type.COMMENT);
  }
}
```

(from Trips/RoomPolySingle/app/src/main/java/com/commonsware/android/room/Comment.java)

`Link` is another subclass of `Note`:

```
package com.commonsware.android.room;

import android.arch.persistence.room.Ignore;
import android.support.annotation.NonNull;
import java.util.UUID;
```

```
public class Link extends Note {
  public Link(@NonNull String id, String title, String url,
              @NonNull String tripId, Type type) {
    super(id, title, url, tripId, Type.LINK);
  }

  @Ignore
  public Link(String text, String url, @NonNull Trip trip) {
    this(UUID.randomUUID().toString(), text, url, trip.id, Type.LINK);
  }
}
```

(from Trips/RoomPolySingle/app/src/main/java/com/commonsware/android/room/Link.java)

This simplifies our `@Dao` class (`TripStore`). In effect, Room ignores the difference between `Link` and `Comment`, dealing only with the base `Note` class, since that is the `@Entity`. So for inserts, updates, and deletes, we can pass a `Link` or `Comment` to methods that take a `Note`, and it all works fine.

```
  /*
    Note
   */

  @Query("SELECT * FROM notes WHERE tripId=:tripId")
  abstract List<Note> findNotesForTrip(String tripId);

  @Insert
  abstract void insert(Note... comments);

  @Update
  abstract void update(Note... comments);

  @Delete
  abstract void delete(Note... comments);
```

(from Trips/RoomPolySingle/app/src/main/java/com/commonsware/android/room/TripStore.java)

Retrieval becomes a bit more interesting, though. `findNotesForTrip()`, shown above, nicely returns all links and comments... but as `Note` objects, not as `Link` and `Comment` objects. If we want those, we need to have dedicated retrieval methods by type:

```
  /*
    Comment
   */

  @Query("SELECT * FROM notes WHERE tripId=:tripId AND type=0")
```

```
abstract List<Comment> findCommentsForTrip(String tripId);

/*
  Link
 */

@Query("SELECT * FROM notes WHERE tripId=:tripId AND type=1")
abstract List<Link> findLinksForTrip(String tripId);
```

(from Trips/RoomPolySingle/app/src/main/java/com/commonsware/android/room/TripStore.java)

And, as a result, we do not have a single method to retrieve both links and comments as `Link` and `Comment` objects. We would need another `@Transaction` wrapper method as before.

Polymorphism With M:N Relations

The sort of discriminator column used above (the `type` field) also comes into play for M:N relations.

Suppose that links and comments could be associated with multiple trips. Somehow, given a trip, we need to know the links and comments. As we saw elsewhere, we can do this via a join entity mapping to a join table, with each entity instance representing one pairing of link/comment to a trip.

And, as with 1:N relations, we have two choices:

1. Put all of links and comments in a single table, as we did in the second example above. In that case, our type discriminator goes on that table and in its entity. The join table would only need the ID of the note.
2. Have links and comments in separate tables, as we did in the first example. In that case, our link IDs and comment IDs need to be of the same data type (e.g., strings), and we would need a type discriminator in the join table, to be able to distinguish whether a given join entity instance is connecting a trip to a link or to a comment.

LiveData Transformations

Sometimes, the data that you want is not the data that you get:

- You want to capitalize those names before showing them in a list
- You want to restrict the results to some subset of what you are receiving
- You do not need the data, but rather some calculation made upon batches of the data, grouped by some key
- And so on

The `LiveData` system has some limited support for "transformations", which help you adapt an existing `LiveData` into one that changes the data to better suit your needs. You can also create your own transformations, if desired. In this chapter, we will explore all of this.

The Bucket Brigade

`LiveData` is designed to be a simplified form of a reactive framework like RxJava.

Anyone who has looked at RxJava code knows that it has a tendency towards long chains of calls, to configure a stream of data, and sometimes to modify that stream along the way.

For example, you will find code like:

```
Observable<String> observable=Observable
  .create(new WordSource(getActivity()))
  .subscribeOn(Schedulers.io())
  .map(s -> (s.toUpperCase()))
  .observeOn(AndroidSchedulers.mainThread())
  .doOnComplete(() -> {
```

```
    Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
      .show();
  });
```

Here, we:

- Request a roster of words (`create(new WordSource(getActivity()))`)
- Ask to retrieve that roster on a background thread, as it involves disk I/O (`subscribeOn(Schedulers.io())`)
- Convert the words to uppercase (`map(s -> (s.toUpperCase()))`)
- Ask to get the results on the main application thread, so we can update our UI with these words (`observeOn(AndroidSchedulers.mainThread())`)
- Show a `Toast` when we are done processing the words (`doOnComplete() ...`)

In particular, `map()` is a transformation "operator", in Rx terms. `map()` takes an object from our stream of data (in this case, a word) and transforms it into something else, which flows downstream to the subsequent chained calls. In this case, `map()` transforms a `String` into a `String`, where the "transformation" is converting the input `String` to uppercase to use as the output `String`.

RxJava has a dozens of such operators. In contrast, `LiveData` has two, and we will implement a third ourselves to see how that is accomplished.

Mapping Data to Data

Both RxJava and `LiveData` offer a `map()` transformation. As seen in the preceding section, a `map()` converts an item of data from the stream (e.g., a `String`) to some other item of data to flow downstream (e.g., an uppercase `String`).

However, whereas `map()` is a method on RxJava's `Observable` and related classes, with `LiveData`, the transformations are held in a separate `Transformations` class.

For example, suppose we have a DAO method like:

```
@Query("SELECT * FROM Customer")
LiveData<List<Customer>> allCustomers();
```

Here, we are expecting a stream of results, where each item is the result of the query: a list of customers. Room will deliver the current customers to us quickly, then will deliver updated lists of customers as the `Customer` table changes, so long as we are observing the `LiveData`.

Suppose, though, we do not need the `Customer` objects, but instead need their IDs. The simplest and most performant solution would be to have a different DAO method:

```
@Query("SELECT id FROM Customer")
LiveData<List<String>> allCustomerIds();
```

However, that does not use `Transformations`, and so it is boring. Plus, not every possible transformation is simply cutting a POJO down to a single field from that POJO.

The `Transformations` equivalent would be something like this:

```
LiveData<List<String>> liveCustomerIds=
  Transformations.map(store.allCustomers(),
    new Function<List<Customer>, List<String>>() {
      @Override
      public List<String> apply(List<Customer> customers) {
        ArrayList<String> result=new ArrayList<>();

        for (Customer customer : customers) {
          result.add(customer.id);
        }

        return(result);
      }
    });
```

`map()` takes two parameters: a `LiveData` of the stream to manipulate, and a `Function` that converts items from that stream from one data type to another.

Here is where Room's insistence on a single-object response becomes a pain. If this were a stream of `Customer` objects, our `Function` could just get the `id` from the `Customer` and return it. But we do not have a stream of `Customer` objects — we have a stream of a list of `Customer` objects. That means we need to return a list of customer IDs, requiring allocating a new `ArrayList` and iterating over each `Customer` to add its `id` to that list.

Mapping Data to… LiveData?

So now we have a list of `Customer` IDs. Suppose that we now want to retrieve the categories associated with all of the `Customer` entities. That requires another database request via our DAO:

```
@Query("SELECT categories.* FROM categories\n"+
  "INNER JOIN customer_category_join ON
categories.id=customer_category_join.categoryId\n"+
  "WHERE customer_category_join.customerId IN (:customerIds)")
LiveData<List<Category>> categoriesForCustomers(List<String> customerIds);
```

And if we are on the main application thread — as is typical when working with `LiveData` results — we need the DAO to return another `LiveData`.

In principle, you could use `map()` for this. However, for this scenario, there is `switchMap()`. This says that the objects being created via the mapping are themselves `LiveData`. This helps the `LiveData` system keep everything in sync, particularly across lifecycle events.

So, given the `liveCustomerIds` from the preceding section, we can get the categories via:

```
final LiveData<List<Category>> liveCategories=
  Transformations.switchMap(liveCustomerIds,
    new Function<List<String>, LiveData<List<Category>>>() {
      @Override
      public LiveData<List<Category>> apply(List<String> customerIds) {
        return(store.categoriesForCustomers(customerIds));
      }
    });
```

And, if we arrange to `observe()` that `liveCategories` object, we will be called with `onChanged()` when the list of `Category` objects is available, after the initial database I/O to get the customers, then the secondary database I/O to get the categories for those customers.

Writing a Transformation

Another RxJava transformation operator is `filter()`. This takes a stream of objects and a function that tests each object and returns `true` for the ones to be sent downstream. The ones that test out to `false` are dropped. Hence, the stream becomes filtered by whatever rule is encoded in that function.

`Transformations` does not have a `filter()` method, but we can write one, to see what a transformation method looks like.

Earlier in the book, we had the `Sensor/LiveList` sample, where we had a `LiveData`

reporting sensor events, specifically the light level. The Sensor/LiveFilter sample project is a clone of that project, one that introduces a filter, to only report those readings that fall between 20 and 40 lux.

To that end, we have a `LiveTransmogrifiers` class that serves as a home for our transformation methods:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.LiveData;
import android.arch.lifecycle.MediatorLiveData;
import android.support.annotation.MainThread;
import android.support.annotation.NonNull;

class LiveTransmogrifiers {
  interface Confirmer<T> {
    boolean test(T thingy);
  }

  @MainThread
  static <X> LiveData<X> filter(@NonNull LiveData<X> source,
                                @NonNull final Confirmer<X> confirmer) {
    final MediatorLiveData<X> result=new MediatorLiveData<>();

    result.addSource(source, x -> {
      if (confirmer.test(x)) {
        result.setValue(x);
      }
    });

    return(result);
  }
}
```

(from Sensor/LiveFilter/app/src/main/java/com/commonsware/android/livedata/LiveTransmogrifiers.java)

The RxJava `filter()` operator uses a `Predicate` as the function for testing an object to determine if it should be passed or not. Unfortunately, `Predicate` is part of the Java 8 classes added in Android 7.0, and so it is unavailable for older devices. So, we have a `Confirmer` interface that fills that role. The `test()` method on a `Confirmer` needs to return `true` for objects that should pass the filter, `false` otherwise.

The `filter()` method on `LiveTransmogrifiers` takes a `LiveData` of some type and a `Confirmer` of that type. It then uses a `MediatorLiveData`, which is a `LiveData` object that can chain onto an existing `LiveData` and expose the `onChanged()` method for

outside parties to use. In this case, our lambda uses the `Confirmer` to see if the new object passes the `test()`, and if it does, we call `setValue()` on the `MediatorLiveData` to have that object flow along to anything that observes that `MediatorLiveData`. `filter()` then returns that `MediatorLiveData`. The net effect is as if `filter()` wraps the original `LiveData` in another `LiveData` that applies our filtering rule.

We can now use `filter()` to limit the readings that we get from the sensor:

```
    final LiveData<SensorLiveData.Event> filtered=
      LiveTransmogrifiers.filter(state.sensorLiveData,
        event -> (event.values[0]>20 && event.values[0]<40));

    filtered.observe(this, event -> adapter.add(event));
```

(from Sensor/LiveFilter/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

We pass our original `SensorLiveData` to `filter()`, along with a `Confirmer` (in the form of a lambda expression) that sees if the light level is between 20 and 40. Then, we observe the results of the `filter()` call and only add those objects — not every reading from the `SensorLiveData` — to the `EventLogAdapter`.

The net result, if you compare and contrast the output of this sample with the original, is that while the original reports everything, this new sample only reports a subset of the data.

Do We Really Want This?

`LiveData` was not set up to have a vast library of transformations, the way that RxJava has its vast library of operators. `map()` and `switchMap()` are almost afterthoughts. And while Google may not add many more transformations to the `Transformations` class, undoubtedly somebody will create a library with implementations of `filter()` and a handful of other RxJava-style operators.

However, those libraries will be limited, because `LiveData` itself is not as rich a framework as is RxJava. There is no notion in `LiveData` of propagating errors, or indicating that a stream is completed. Some RxJava operators will be difficult or impossible to implement as a result.

And this is by design.

`LiveData` is designed to be simple and lifecycle aware. That's it. If your needs

transcend what `LiveData` can handle well, consider migrating to RxJava. Conversely, if `LiveData` handles everything that you need, you can skip RxJava's complexity.

RxJava and Room

In a previous chapter, we saw how a Room DAO could return a `LiveData` object as a wrapper around the "real" data to be retrieved from the database. Then, the `@Query` method would no longer be a blocking call, but instead would return the `LiveData` immediately, with the actual query results being delivered to any observers of the `LiveData`.

Not surprisingly, you can do the same thing with RxJava types. Simply wrap the return type from the `@Query` method in a suitable RxJava type. You get the same results as you do with `LiveData`: the `@Query` method returns the RxJava object immediately, and you get the actual query results via RxJava's subscriber system.

However, RxJava is a much richer library, and it is commensurately more complex. In this chapter, we will explore what RxJava types can be returned by a `@Query`, how those work with respect to data changes, and how the underlying data type affects all of that.

Adding RxJava

RxJava has its own dependencies. In an Android app, typically you will use `io.reactivex.rxjava2:rxjava` and `io.reactivex.rxjava2:rxandroid`, where the latter provides utility classes like `AndroidSchedulers`.

However, Room itself does not have a transitive dependency upon RxJava. Otherwise, everybody using Room would need to pull in RxJava, and that would add unnecessary bloat.

So, in addition to the regular dependencies for RxJava and the regular dependencies for Room (e.g., `android.arch.persistence.room:runtime`), you also need the

`android.arch.persistence.room:rxjava2` dependency. This contains the glue code necessary to tie Room to RxJava:

```
implementation "android.arch.persistence.room:runtime:1.1.1"
implementation "android.arch.persistence.room:rxjava2:1.1.1"
annotationProcessor "android.arch.persistence.room:compiler:1.1.1"
```

(from Trips/RxRoom/app/build.gradle)

If you forget this dependency, but you have the regular RxJava and Room dependencies, you will not notice a problem right away. You will be able to use RxJava types in your DAO, because Android Studio knows about RxJava through your existing dependencies. However, when you go to build and run the project, your build will fail with:

```
Error:(41, 24) error: To use RxJava2 features, you must add `rxjava2` artifact from
Room as a dependency. android.arch.persistence.room:rxjava2:<version>
```

Decisions, Decisions

To determine what Rx type is appropriate for your case, you need to make two decisions:

1. Do you just want the data once, or do you want to be notified about updates over time?
2. How many objects might you get in your result set from your query? 0? 1? N?

One-Time or Ongoing?

In some cases, you are just performing a query to get some data out of the database, and that's it.

However, Room offers a powerful feature whereby it will re-execute your query and deliver you fresh results, if somewhere else Room sees that you manipulated the contents of the table(s) that your query uses. The result is a bit like a `ContentProvider` and `ContentObserver`, in that you get fresh results delivered "automatically". However, to make this work, you will need to use particular Rx types that support ongoing events, not just one-time events.

Zero, One, or N?

Sometimes, you are fairly certain how many rows will be returned by your query, and therefore how many entities or other POJOs will be returned by a DAO method. For example, if you are querying by primary key or some other unique column, you know that you will only ever get back 0 or 1 objects, as there cannot be more than one with the unique value. Similarly, if you are using simple aggregate functions (e.g., `SELECT COUNT(*) FROM table WHERE criteria...`), you know that you will get exactly one row back.

In other cases, you may have no idea how many rows you will get. Querying for all rows in a table might return 0, 1, or N rows, depending on the state of the table at the time. In these cases, you may want to have reactive types that return a `List` of objects, rather than just an individual object.

The One-Time Option: Single

In RxJava, a `Single` is a reactive type that is designed to deliver a single result, and that's it. You either get that result or you get an error, such as when the data source cannot actually deliver you the result (e.g., `IOException` on a network call).

`Single` is a good type to use in a Room DAO when you are looking to just get the current query results, without keeping an open subscription for ongoing changes.

Item Singles

For cases where you are fairly certain that you will get a single row in your query results, you could use a `Single` for the desired entity or POJO type:

```
@Query("SELECT * FROM trips WHERE id=:id")
Single<Trip> singleTripById(String id);
```

However, in this case, if your query returns no rows, you will get an `EmptyResultSetException` from Room. When you `subscribe()` to the `Single`, you should provide both a listener for a POJO and an error listener, in case your query returns no rows.

List Singles

Another option is to have your DAO method return a `Single` for a `List` of entities or

other POJOs:

```
@Query("SELECT * FROM trips ORDER BY title")
Single<List<Trip>> singleAllTrips();
```

This works for any number of rows, including the zero-row case (you just get an empty list).

The One-Time 0-1 Option: Maybe

`Maybe` is an RxJava type for cases where you may get a result, or you may not, or you may get an error.

For the case where you might get a row or you might not, and you would prefer not to have to look for a special exception to identify the zero-rows case, `Maybe` is a fine choice:

```
@Query("SELECT * FROM trips WHERE id=:id")
Maybe<Trip> maybeTripById(String id);
```

The Ongoing Option: Flowable

While we tend to focus on RxJava, bear in mind that RxJava 2 is itself based upon a broader Reactive Streams initiative, which has its own library. Effectively, Reactive Streams offers a common API for some high-level constructs, to help make it easier to have multiple disparate reactive libraries be able to inter-operate.

One key class from Reactive Streams is `Publisher`, which is the source of events that get "published". The primary implementation of `Publisher` in RxJava 2 is `Flowable`. In principle, you could use either type with Room, though you may be more comfortable with `Flowable`, so you are sticking with native RxJava types.

`Flowable` is reminiscent of `Observable`, in that it represents a stream of ongoing events. If you use `Flowable` as the return type of your DAO method, not only will you get the initial query results, but so long as you are subscribed to that `Flowable` you will receive *updated* query results if Room thinks that the data may have changed.

Note that what Room *thinks* might have changed may not actually have changed. It is entirely possible that you will get the exact same data multiple times. Room is

looking for SQL operations that modify the contents of a table; if your query does not happen to reference any of those modifications, the re-executed query will return the same data as you received in the previous results.

Item Flowables

If you have long-term interest in a single row, you could use a `Flowable` for a single entity or other POJO:

```
@Query("SELECT * FROM trips WHERE id=:id")
Flowable<Trip> flowTripById(String id);
```

However, if your query returns no rows, the `Flowable` does not deliver anything to you. This may be OK in your situation, or it may not. If you need a positive indication that no rows matched your query, a `Flowable` for an individual object is not a good choice.

List Flowables

You could also have a `Flowable` for a `List` of entities or other POJOs:

```
@Query("SELECT * FROM trips ORDER BY title")
Flowable<List<Trip>> flowAllTrips();
```

In this case, you will always get a result, which may be an empty `List` if no rows matched your query.

Plus, since this is a `Flowable`, so long as you have an active subscriber, you will get updates delivered to you if Room thinks that the table(s) in your query may have been modified.

@RawQuery and Reactive Responses

Earlier, we saw that you can use `@RawQuery` to execute queries where either:

- You do not know the SQL statement at compile time, as it needs to be assembled from pieces at runtime, or
- The table being queried is not associated with a Room `@Entity`

If you attempt to return a reactive result from the `@RawQuery` method (e.g., a `Flowable`, a `LiveData`), and the core object of the result is not an `@Entity`, you will

crash at build time, with:

```
Observable query return type (LiveData, Flowable, DataSource, DataSourceFactory etc)
can only be used with SELECT queries that directly or indirectly (via @Relation, for
example) access at least one table. For @RawQuery, you should specify the list of
tables to be observed via the observedEntities field.
```

Room wants to be able to deliver updates to your reactive object when the underlying table's content changes, and it can only do that if it knows the table.

For cases where there is a clear `@Entity` (or perhaps more than one) whose changes should trigger an updated result to be delivered to you reactively, add the `observedEntities` property to the `@RawQuery` annotation, with the `.class` reference(s) of the entities that should be observed. If anything associated with those entities changes, Room will re-deliver a fresh result to you.

For cases where the table being queried is not associated with a Room `@Entity`, either:

- Do not return a reactive type, or
- Lie to Room and supply some arbitrary entity in `observedEntities`, just to make the error go away

Hopefully, this requirement will be relaxed, as not every situation calls for this sort of automatic-update delivery. For example, in the chapter on full-text searching in Room, we will see an example of using `@RawQuery` with a reactive response type and having to provide a "fake" value for `observedEntities`, just to eliminate the error.

RxJava and Lifecycles

RxJava is cool, albeit confusing. But beyond that, it is a pure Java library. RxJava knows nothing about Android-specific concepts, as it is designed to be used on all sorts of Java projects.

Android developers using RxJava invariably also add RxAndroid, which gives us access to a `Scheduler` that knows about the Android main application thread. However, RxAndroid does not have anything that deals with activity or fragment lifecycles, leaving that up to you. With Android lifecycles, we want to create things as activities and fragments start up and clean up those things as the activities and fragments go away. In the case of RxJava, if we subscribe to some `Observable`, it would be nice to get rid of that subscription at an appropriate point.

In this chapter, we will explore a few options — including one from the Architecture Components — for dealing with lifecycles with RxJava.

The Classic Approach

The default way of handling this is the approach used in the chapter on RxJava and Room:

- Hold onto the `Disposable` that you get back from subscribing to an observable
- Clean up that `Disposable` in a suitable lifecycle method, such as `onDestroy()`, via a call to `dispose()`

If you have several subscriptions to track, `CompositeDisposable` lets you track all of them in one spot. `CompositeDisposable` has `add()` and `addAll()` methods to add subscriptions to it. And, as the name suggests, `CompositeDisposable` implements

the composite pattern, and so `CompositeDisposable` itself is a `Disposable`. Calling `dispose()` on the `CompositeDisposable` triggers calls to `dispose()` on all of the `Disposable` objects you added to the composite.

This works, but it does require you to remember to clean these things up, and it is easy to forget.

Bridging RxJava and LiveData

Of course, the Architecture Components have `Lifecycle`, `LifecycleOwner`, and related classes for performing operations when certain lifecycle events occur. `LiveData` — the Architecture Components' counterpart to RxJava — is intrinsically lifecycle-aware.

So, another option would be to have some sort of adapter that converts RxJava into `LiveData`. We could then observe the `LiveData`, knowing that our `Observer` would be cleaned up automatically as part of normal lifecycle management.

Fortunately, the Architecture Components has `LiveDataReactiveStreams`, for converting `LiveData` to and from RxJava structures, as is illustrated in the [`Trips/RxLifecycle`] sample project.

`LiveDataReactiveStreams` is in yet another artifact, `android.arch.lifecycle:reactivestreams`. So, you need to request that artifact with the others that you are using:

```
dependencies {
  implementation "com.android.support:recyclerview-v7:28.0.0"
  implementation "com.android.support:support-core-utils:28.0.0"
  implementation "com.android.support:support-fragment:28.0.0"
  implementation 'io.reactivex.rxjava2:rxjava:2.2.2'
  implementation 'io.reactivex.rxjava2:rxandroid:2.1.0'
  implementation 'android.arch.lifecycle:runtime:1.1.1'
  implementation 'android.arch.lifecycle:livedata:1.1.1'
  implementation 'android.arch.lifecycle:reactivestreams:1.1.1'
  implementation "android.arch.persistence.room:runtime:1.1.1"
  implementation "android.arch.persistence.room:rxjava2:1.1.1"
  annotationProcessor "android.arch.persistence.room:compiler:1.1.1"
  androidTestImplementation "com.android.support:support-annotations:28.0.0"
  androidTestImplementation 'com.android.support.test:rules:1.0.2'
  androidTestImplementation 'android.arch.core:core-testing:1.1.1'
  androidTestImplementation "com.android.support:support-core-utils:28.0.0"
  androidTestImplementation "com.android.support:support-compat:28.0.0"
```

```
  androidTestImplementation 'android.arch.lifecycle:runtime:1.1.1'
  androidTestImplementation 'android.arch.lifecycle:common:1.1.1'
}
```

(from Trips/RxLifecycle/app/build.gradle)

From RxJava to LiveData

To bridge from RxJava to `LiveData`, `LiveDataReactiveStreams` offers a `fromPublisher()` method. Here, "publisher" refers to `Publisher` from the Reactive Streams initiative. Most RxJava `Observable` types do not implement the `Publisher` interface, but `Flowable` does. And most RxJava `Observable` types can be converted to a `Flowable` via the `toFlowable()` method.

As a result, the recipe for using `LiveDataReactiveStreams` is:

- Create your RxJava `Observable` as normal
- Call `toFlowable()` on it to convert it into a `Flowable`
- Pass that `Flowable` to `fromPublisher()` to get a `LiveData`
- Observe that `LiveData` and consume the results, such as with a method reference

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  setLayoutManager(new LinearLayoutManager(getActivity()));
  getRecyclerView()
    .addItemDecoration(new DividerItemDecoration(getActivity(),
      LinearLayoutManager.VERTICAL));

  TripStore store=TripDatabase.get(getActivity()).tripStore();

  Flowable<List<Trip>> trips=store.maybeAllTrips()
    .subscribeOn(Schedulers.single())
    .observeOn(AndroidSchedulers.mainThread())
    .toFlowable();

  LiveDataReactiveStreams.fromPublisher(trips)
    .observe(this, this::setAdapter);
}
```

(from Trips/RxLifecycle/app/src/main/java/com/commonsware/android/room/TripsFragment.java)

Here, we do this work inside of `TripsFragment`, which is a `LifecycleOwner` courtesy

of the `support-fragment` implementation of `Fragment`. We do not have to worry about cleaning up the `LiveData` ourselves; the lifecycle system will handle this for us.

However, this comes with a significant side effect: on a configuration change, you may be handed data *twice*:

- Once from standard `LiveData` behavior, which automatically delivers the last-emitted object when there is a new observer, and
- Possibly once from your RxJava source, as `LiveDataReactiveStreams` re-subscribes to that source as part of that configuration change

For example, suppose that you have a method that returns a `Single` like this:

```
Single<Set<String>> loadStringSet() {
  return Single.create(emitter -> {
    SharedPreferences prefs=PreferenceManager.getDefaultSharedPreferences(ctxt);

    emitter.onSuccess(prefs.getStringSet(PREF_STUFF, Collections.emptySet()));
  });
}
```

Here, we are loading a `Set` of `String` objects from `SharedPreferences`. Since on the first load of data from `SharedPreferences`, `SharedPreferences` performs disk I/O, it is safest (in terms of avoiding jank) to always access `SharedPreferences` on a background thread.

If we use `LiveDataReactiveStreams.fromPublisher()` to wrap that `Single` in a `LiveData` object, the first observer of the `LiveData` will trigger the `Single` to load its data. When a configuration change occurs, the `LiveData` wrapper around the `Single` unsubscribes from the `Single`, then re-subscribes when the new activity/fragment instance starts observing the `LiveData` again. For a cold observable like this one, this causes our lambda expression to be re-evaluated. Both that newly-generated result *and* the previously-cached result from the `LiveData` will be delivered to our activity's (or fragment's) observer. The observer needs to be aware of this and take appropriate steps: it needs to *replace* any existing data, not add to it.

From LiveData to RxJava

If, for some reason, you need to convert a `LiveData` *to* something in the RxJava space, `toPublisher()` on `LiveDataReactiveStreams` can adapt a `LiveData` to a `Publisher`. On its own, `Publisher` only offers a `subscribe()` method. However,

`Observable.fromPublisher()` can adapt a `Publisher` into an `Observable`, and from there you can set up RxJava chains as needed.

The Uber Solution: AutoDispose

The downside of `LiveDataReactiveStreams`, to some, is that you wind up with a `LiveData` object. Some developers will prefer to stick with RxJava throughout, but still would like some measure of automatic lifecycle-based subscription cleanup.

For that, Uber offers AutoDispose.

It allows you to add lifecycle-based subscription cleanup with a single line added to your RxJava chain, akin to:

```
observable
  // subscribeOn(), observeOn(), map(), and so on go here
  .to(AutoDispose.with(scope).forObservable())
  .subscribe(/* good stuff here */);
```

Here, `scope` will be an object that provides lifecycle details to AutoDispose, so that it knows when to stop forwarding events on to your `Consumer` or other subscriber. There are two Android-specific classes for this:

- `AndroidLifecycleScopeProvider`, which uses `Lifecycle` and `LifecycleOwner`
- `ViewScopeProvider`, which (somehow) uses `View`

If you are comfortable with consuming events in your UI using `LiveData`, `LiveDataReactiveStreams` is likely to be the simpler choice. If, however, you are interested in avoiding the conversion to `LiveData`, AutoDispose is worth considering.

Packing Up a Room

A popular question over the years has been: how do I ship a pre-populated database with my app?

Android has never offered an "out of the box" solution for this, though there are third-party solutions that we can use. Room changes the problem space slightly, breaking the original solutions and requiring a fresh option. In this chapter, we will review that new solution and how you can use it to ship a database, packaged in your app, and used by Room.

The Problem

Roughly speaking, data for a SQLite database can come from one of three places:

- It can come from user input, through the UI of your app
- It can come from external sources, such as data that you synchronize with a Web service
- It can come with the app itself, when the app was installed

Most apps get by with the first two data sources. However, from time to time, there are situations where shipping a database with an app can prove useful.

Sometimes, that database represents starter data, that the user (or a server) will augment or modify over time. For example, you might be writing an app for documenting household goods and other items, to help a homeowner or renter with future insurance claims in case of a fire, natural disaster, etc. The app allows the user to take photos of items, provide notes about them (make, model, etc.), and categorize them. While the user can manage the list of categories, you might want to ship some pre-defined categories with the app, so that the user is not forced into

deciding on categories before the app can be used.

Sometimes, the packaged database is a read-only data repository. Frequently, this is for apps that want to offer offline access to a dataset that otherwise might be pulled from a Web service. Sometimes, the data simply is not meant to change frequently. For example, the APK edition of *The Busy Coder's Guide to Android Development* ships with a packaged database containing the prose of the book, indexed by SQLite's FTS3 engine for use with full-text searching. The database contents are updated when the app is updated, reflecting a new version of the book.

For tiny datasets, you can get away with populating the database yourself when you first create it, using ordinary Java code. This is inefficient for larger databases, though, as it forces us to execute a bunch of SQLite transactions on the user's device. It would be more efficient to ship an actual SQLite database file. And, since the developers of SQLite have done an admirable job of backwards and forwards compatibility with their database file structure, this works fairly well. You use other tools, such as DB Browser for SQLite, to create the database with your data. Then, you... do... something... to put that database in the APK and use it at runtime.

The Classic Solution: SQLiteAssetHelper

The recommended "something" for traditional SQLite work in Android has been Jeff Gilfelt's `SQLiteAssetHelper`. Basic use of `SQLiteAssetHelper` is fairly simple:

- Create an `assets/databases/` directory in your `main` sourceset of your app module
- Put your pre-populated database in that directory, with the same filename that you want to use at runtime
- Rather than using `SQLiteOpenHelper`, subclass `SQLiteAssetHelper` instead, supplying that filename:

```
public class YourDatabase extends SQLiteAssetHelper {
  private static final String DB_NAME="whatever.db";
  private static final int SCHEMA_VERSION=1;

  public YourDatabase(Context context) {
    super(context, DB_NAME, null, SCHEMA_VERSION);
  }
}
```

The rest of your code can use your `SQLiteAssetHelper` subclass just as it would

`SQLiteOpenHelper`, such as calling `getReadableDatabase()` or `getWritableDatabase()`. The first time one of those methods is called, `SQLiteAssetHelper` will notice that there is no database and will copy the database from assets into the proper location.

NOTE: Jeff Gilfelt is no longer maintaining `SQLiteAssetHelper`. Right now, it still works fine for this role. With luck, new actively-supported solutions will become available.

The New Problem

Room, however, does not use `SQLiteOpenHelper`... at least, not directly. There is no obvious place that you can use to put `SQLiteAssetHelper`.

There are two potential hooks that we saw in the chapter on the support database API.

One is with `addCallback()` on `RoomDatabase.Builder`. This allows you to register a `RoomDatabase.Callback` instance with the `RoomDatabase`. That callback will be called with `onCreate()` when the database is created for the first time. If you wanted to execute some SQL statements to populate the database, you could use this approach. However, Room stores its own metadata in the database in a private table. We have no great way of putting that metadata inside our pre-populated database, and by the time `onCreate()` is called, it is too late for us to try to swap in that pre-populated database.

The other hook is with `openHelperFactory()` on `RoomDatabase.Builder`. This is the entry point to replacing the standard SQLite access code with our own code. As it turns out, a slightly-modified version of `SQLiteAssetHelper` can be used in this fashion.

Merging SQLiteAssetHelper with Room

The `General/AssetRoom` sample project demonstrates that latter approach, in the form of `AssetSQLiteOpenHelper` and `AssetSQLiteOpenHelperFactory` classes.

As with the original `SQLiteAssetHelper`, you need to put your pre-populated database in `assets/databases/`, under whatever name you want to use for that database:

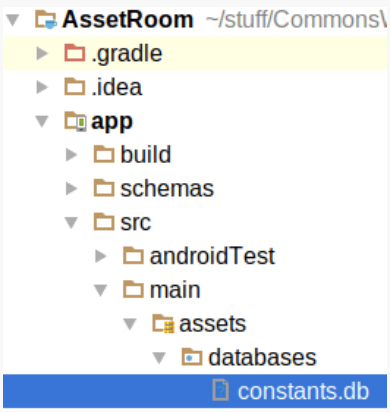


Figure 1: AssetRoom Project, Showing assets/databases/constants.db

Then, when creating your `RoomDatabase` subclass:

- Use that same database name, and
- Use `openHelperFactory(new AssetSQLiteOpenHelperFactory())` when creating the instance of your `RoomDatabase` via `RoomDatabase.Builder`

```
package com.commonsware.android.room;

import android.arch.persistence.db.framework.AssetSQLiteOpenHelperFactory;
import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Database;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;
import java.util.List;

@Database(entities={Constant.class}, version=1)
abstract class ConstantsDatabase extends RoomDatabase {
  public abstract Constant.Store constantsStore();

  static final String DB_NAME="constants.db";
  private static volatile ConstantsDatabase INSTANCE=null;

  synchronized static ConstantsDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt);
    }

    return(INSTANCE);
  }
```

```
  static ConstantsDatabase create(Context ctxt) {
    RoomDatabase.Builder<ConstantsDatabase> b=
      Room.databaseBuilder(ctxt.getApplicationContext(), ConstantsDatabase.class,
        DB_NAME);

    return(b.openHelperFactory(new AssetSQLiteOpenHelperFactory()).build());
  }
}
```

(from General/AssetRoom/app/src/main/java/com/commonsware/android/room/ConstantsDatabase.java)

Compared with some of the other Room samples, you will notice that `ConstantsDatabase` lacks any option to create an in-memory database. There is a good reason for that: `SQLiteAssetHelper` cannot support that, as we have no way of copying the database file into some place where SQLite itself will use it in memory.

This makes testing slightly more aggravating, as you will want to make sure that you delete your database file in an `@After` method, as otherwise future runs of your tests will encounter the existing database file:

```
package com.commonsware.android.room;

import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

@RunWith(AndroidJUnit4.class)
public class AssetTests {
  private ConstantsDatabase db;
  private Constant.Store store;

  @Before
  public void setUp() {
    db=ConstantsDatabase.get(InstrumentationRegistry.getTargetContext());
    store=db.constantsStore();
  }

  @After
  public void tearDown() {
    db.close();
    assertTrue(InstrumentationRegistry
      .getTargetContext()
      .getDatabasePath(ConstantsDatabase.DB_NAME)
```

```
      .delete());
  }

  @Test
  public void assets() {
    assertEquals(13, store.all().size());
    store.insert(new Constant("Pi", 3.1415926));
    assertEquals(14, store.all().size());
  }
}
```

(from General/AssetRoom/app/src/androidTest/java/com/commonsware/android/room/AssetTests.java)

However, otherwise, using the pre-populated database is no different than using a regular Room database. Whether the data can be modified is up to you, particularly with your `@Dao` class — if you do not write any DAO methods that modify the data, you should be safe. Note, though, that Room has its own metadata table, which might be modified by Room as it sees fit.

Paging Room Data

September 2017 brought a new addition to the Architecture Components: the Paging library. This library contains a series of classes designed to help you offer a browsable UI across a large data set, particularly where that data set comes from a Room-managed database.

In this chapter, we will explore the role of this library, some of the key classes, and the basic setup for use with Room and `RecyclerView`.

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.

There are a number of classes involved in the Paging library, but for basic scenarios, there are a few of significance: including `PagedList` and `PagedListAdapter`.

PagedList

`PagedList`, on the surface, is a `List`, not that dissimilar from an `ArrayList`. However, it is designed to handle very large collections using a time-honored technique: lying.

A `PagedList` may know how much data there can be — to be able to respond to methods like `size()` — but it does not actually hold all of that data. Instead, it holds onto a small amount of data and by default will return `null` for requests to get items from the `List` that have not been loaded.

A `loadAround()` method tells the `PagedList` a position of importance. Asynchronously, `PagedList` will work to load that data and be able to return non-`null` values for positions "around" the requested one. This may cause the `PagedList` to jettison previously-loaded data, to minimize the memory footprint that the `PagedList` takes up.

The idea is that `PagedList` should work the way that the UI does: showing a small amount of information at a time, but allowing for (theoretically) arbitrary navigation through a much larger set of information.

PagedListAdapter

A `PagedListAdapter` is a `RecyclerView.Adapter` that uses a `PagedList` as its source of items to render. It handles the details of calling `loadAround()` for you. All you need to do is handle standard `RecyclerView.Adapter` methods like `onCreateViewHolder()` and `onBindViewHolder()`.

DataSource.Factory

A `DataSource`, surprisingly enough, is a source of data. It is a wrapper around some data provider — a database, a Web service, etc. — and knows how to retrieve pages of data from it.

A `DataSource.Factory` follows a time-honored Java tradition, where we have factory classes to create instances of other things. In this case, a `DataSource.Factory` knows how to create certain types of `DataSource`. In particular, when working with Room, you can request that a `@Query` method return a `DataSource.Factory` as its data type, instead of a `List` of entities or a `LiveData` or other things.

LivePagedListProvider

`LivePagedListProvider` is a utility class that can create a `LiveData` object that delivers `PagedList` objects to observers, given a `DataSource.Factory`. So, if you have a Room `@Query` method that returns a `DataSource.Factory`, you can create a `LivePagedListProvider` to convert that into a `LiveData` for use with your UI layer. Room, `DataSource.Factory`, and `PagedList` will handle loading data asynchronously as you navigate through the list. If you attach the `PagedList` to a `RecyclerView` via `PagedListAdapter`, you get seamless data paging, with controllable memory consumption, with very little work on your part.

Paging and Room

The `CityPop/RoomPaging` sample project will illustrate the use of the Paging library in conjunction with Room.

As with the `PackRoom` sample shown earlier in the book, `RoomPaging` packages a database with the app. Specifically, it is list of 2015 city populations, culled from a United Nations data set. Not all cities are represented there, for unknown reasons, but there are over 1,000, and so it offers a chance to see how the Paging classes work in action.

The Dependency

To use those classes, we need another dependency, one for the Paging library. Paging is on its own separate release cycle from Room or the lifecycle classes.

So, we request the `android.arch.paging:runtime` library in our Gradle script, along

with other necessary dependencies:

```
dependencies {
  implementation "android.arch.persistence.room:runtime:1.1.1"
  annotationProcessor "android.arch.persistence.room:compiler:1.1.1"
  implementation "android.arch.paging:runtime:1.0.1"
  implementation "android.arch.lifecycle:extensions:1.1.1"
  implementation "com.android.support:support-annotations:28.0.0"
  implementation "com.android.support:recyclerview-v7:28.0.0"
  implementation 'com.android.support:support-fragment:28.0.0'
  androidTestImplementation 'com.android.support.test:rules:1.0.2'
  androidTestImplementation "com.android.support:support-annotations:28.0.0"
}
```

(from CityPop/RoomPaging/app/build.gradle)

The Entity, DAO, and Database

Our Room entity is a `City`. It has four fields:

- a unique ID in the form of a UUID (`id`)
- the name of the city (`city`)
- the name of the country or area in which the city is located (`country`)
- the population of the city (`population`)

In addition to sporting a suitable constructor for Room's use and a `toString()` that returns the `city` name, `City` also implements `equals()` and `hashCode()`, using the `id` as the discriminator.

```
package com.commonsware.android.citypop;

import android.arch.paging.DataSource;
import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.Query;
import android.support.annotation.NonNull;
import java.util.List;

@Entity(tableName = "cities")
class City {
  @PrimaryKey
  @NonNull
  final String id;
  final String country;
```

```
  final String city;
  final int population;

  City(@NonNull String id, String country, String city, int population) {
    this.id=id;
    this.country=country;
    this.city=city;
    this.population=population;
  }

  @Override
  public String toString() {
    return(city);
  }

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof City) {
      City other=(City)obj;

      return(id.equals(other.id));
    }

    return(false);
  }

  @Override
  public int hashCode() {
    return(id.hashCode());
  }

  @Dao
  interface Store {
    @Query("SELECT * FROM cities ORDER BY population DESC")
    List<City> allByPopulation();

    @Query("SELECT * FROM cities ORDER BY population DESC")
    DataSource.Factory<Integer, City> pagedByPopulation();
  }
}
```

(from CityPop/RoomPaging/app/src/main/java/com/commonsware/android/citypop/City.java)

`City` also has a nested `Store` interface that is our DAO, with two `@Query` methods. One (`allByPopulation()`) is a traditional synchronous "give me a list of all the cities" query. The other is `pagedByPopulation()`, and it returns a `DataSource.Factory`. `DataSource.Factory` takes two data types:

- The page identifier type
- The type of entity (or other POJO) that you want the underlying Room query to use

A "page identifier" is pretty much what it says: it identifies a page in a response. For a Room query, pages are numbered, and so you will use `Integer` as the page identifier type. The Paging library also supports "keyed" pages, where a page might be identified by something other than a simple number, but Room does not offer that at present.

Our `RoomDatabase` is `CityDatabase`, and it is set up akin to the one from `PackRoom`, using `AssetSQLiteOpenHelperFactory` to use a packaged `un.db` database as our initial data:

```
package com.commonsware.android.citypop;

import android.arch.persistence.db.framework.AssetSQLiteOpenHelperFactory;
import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;
import android.content.Context;

@Database(entities={City.class}, version=1)
abstract class CityDatabase extends RoomDatabase {
  public abstract City.Store cityStore();

  static final String DB_NAME="un.db";
  private static volatile CityDatabase INSTANCE=null;

  synchronized static CityDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt);
    }

    return(INSTANCE);
  }

  static CityDatabase create(Context ctxt) {
    RoomDatabase.Builder<CityDatabase> b=
      Room.databaseBuilder(ctxt.getApplicationContext(), CityDatabase.class,
        DB_NAME);

    return(b.openHelperFactory(new AssetSQLiteOpenHelperFactory()).build());
  }
}
```

(from CityPop/RoomPaging/app/src/main/java/com/commonsware/android/citypop/CityDatabase.java)

The ViewModel

This sample uses `ViewModelProviders`, and for that, we need a `ViewModel` that can get the `LiveData` from the `LivePagedListProvider`, so our UI can observe that data.

So, we have a `CitiesViewModel` serving that role:

```
package com.commonsware.android.citypop;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.arch.lifecycle.LiveData;
import android.arch.paging.DataSource;
import android.arch.paging.LivePagedListBuilder;
import android.arch.paging.PagedList;

public class CitiesViewModel extends AndroidViewModel {
  final LiveData<PagedList<City>> pagedCities;

  public CitiesViewModel(Application app) {
    super(app);

    DataSource.Factory<Integer, City> factory=
      CityDatabase.get(app).cityStore().pagedByPopulation();
    LivePagedListBuilder<Integer, City> pagedListBuilder=
      new LivePagedListBuilder<>(factory, 50);

    pagedCities=pagedListBuilder.build();
  }
}
```

(from CityPop/RoomPaging/app/src/main/java/com/commonsware/android/citypop/CitiesViewModel.java)

We first get our `DataSource.Factory` by asking the `CityDatabase` singleton for the `cityStore()`, and then ask it to get the cities `pagedByPopulation()`.

We then create a `LivePagedListBuilder`, supplying its constructor with the `DataSource.Factory` and how big of a page that we want (in this case, 50). Specifying the page size helps us manage how much heap space gets used by the `PagedList`. The number of rows you request should exceed the maximum number that you might display at once, but it should be small enough to not consume tons of heap space. For the purposes of this sample, 50 is plenty, though since our rows are fairly small, we could go higher if needed.

Finally, we can get a `LiveData` object by calling `build()` on the `LivePagedListBuilder`.

The PagedListAdapter

Our `LivePagedListProvider` will provide us with a `PagedList` of our `City` data, by way of the `CitiesViewModel`. To consume that, we can use a `PagedListAdapter` to show our cities in a `RecyclerView`. Ours is called `CityAdapter` and is a nested class inside of a `CitiesFragment`:

```
private static class CityAdapter extends PagedListAdapter<City, RowHolder> {
  private final LayoutInflater inflater;

  CityAdapter(LayoutInflater inflater) {
    super(CITIES_DIFF);
    this.inflater=inflater;
  }

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new RowHolder(inflater.inflate(R.layout.row, parent, false)));
  }

  @Override
  public void onBindViewHolder(RowHolder holder, int position) {
    City city=getItem(position);

    if (city==null) {
      holder.clear();
    }
    else {
      holder.bind(city);
    }
  }
}
```

(from CityPop/RoomPaging/app/src/main/java/com/commonsware/android/citypop/CitiesFragment.java)

To a large extent, you use `PagedListAdapter` as you would any other subclass of `RecyclerView.Adapter`, including needing to implement `onCreateViewHolder()` and `onBindViewHolder()`. `PagedListAdapter` manages the `PagedList` for us, and that in turn provides us with some methods that we will need along with opportunities to configure how the adapter works.

`PagedListAdapter` takes two data types: the type of data in the `PagedList` (here,

City) and a standard RecyclerView.ViewHolder as you would use with any other RecyclerView.Adapter (here, RowHolder).

PagedListAdapter needs you to pass a DiffUtil.ItemCallback object to the PagedListAdapter constructor. DiffUtil.ItemCallback is part of the RecyclerView family of classes. It can work with a RecyclerView to reflect changes made to the data behind the RecyclerView, ideally with the minimum amount of actual work required by the RecyclerView itself. For more details on DiffUtil.ItemCallback and its role, see *The Busy Coder's Guide to Android Development*.

Specifically, we need to supply a DiffUtil.ItemCallback for our model data type, City in this case. To that end, we have a static instance of DiffUtil.ItemCallback named CITIES_DIFF:

```
static final DiffUtil.ItemCallback<City> CITIES_DIFF=new DiffUtil.ItemCallback<City>() {
  @Override
  public boolean areItemsTheSame(@NonNull City oldItem,
                                 @NonNull City newItem) {
    return(oldItem.equals(newItem));
  }

  @Override
  public boolean areContentsTheSame(@NonNull City oldItem,
                                    @NonNull City newItem) {
    return(areItemsTheSame(oldItem, newItem));
  }
};
```

(from CityPop/RoomPaging/app/src/main/java/com/commonsware/android/citypop/CitiesFragment.java)

areItemsTheSame() takes advantage of that equals() method that we implemented on City, to determine if two City objects are the same by their id values. The data in the database is fairly unique — any two rows should have different content — so areContentsTheSame() simply delegates to areItemsTheSame().

PagedListAdapter offers a getItem() method. Given a position, it will give us the model object (City) for that position... if that model object is loaded. If not, it will return null. So, the onBindViewHolder() method of our CitiesAdapter uses getItem(), and either binds the City to the RowHolder or asks the RowHolder to clear() its contents.

RowHolder, in turn, does typical ViewHolder things: retrieving widgets out of the inflated layout and adjusting their contents as needed:

```
private static class RowHolder extends RecyclerView.ViewHolder {
  private final TextView cityLabel;
```

```
    private final TextView country;
    private final TextView population;

    RowHolder(View itemView) {
      super(itemView);

      cityLabel=itemView.findViewById(R.id.city);
      country=itemView.findViewById(R.id.country);
      population=itemView.findViewById(R.id.population);
    }

    void bind(City city) {
      cityLabel.setText(city.city);
      country.setText(city.country);
      population.setText(NumberFormat.getInstance().format(city.population));
    }

    void clear() {
      cityLabel.setText(null);
      country.setText(null);
      population.setText(null);
    }
  }
```

(from CityPop/RoomPaging/app/src/main/java/com/commonsware/android/citypop/CitiesFragment.java)

In a production app, we might put placeholder information in the rows for a `null` `City`, rather than clear the widgets.

The CitiesFragment

`CitiesFragment` inherits from a `RecyclerViewFragment` seen earlier in the book. That, plus our use of `LiveData` and view models, means the only method on `CitiesFragment` itself is `onViewCreated()`:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  setLayoutManager(new LinearLayoutManager(getActivity()));
  getRecyclerView()
    .addItemDecoration(new DividerItemDecoration(getActivity(),
      LinearLayoutManager.VERTICAL));

  CitiesViewModel vm=ViewModelProviders.of(this).get(CitiesViewModel.class);
  final CityAdapter adapter=new CityAdapter(getActivity().getLayoutInflater());

  vm.pagedCities.observe(this, adapter::submitList);
```

```
    setAdapter(adapter);
  }
```

(from CityPop/RoomPaging/app/src/main/java/com/commonsware/android/citypop/CitiesFragment.java)

In addition to basic setup of the `RecyclerView`, we:

- Obtain or create our `CitiesViewModel` by way of `ViewModelProviders`
- Create our `CitiesAdapter`
- Arrange to observe the `LiveData` and hand the resulting `PagedList` objects over to the `CitiesAdapter`, via its inherited `setList()` method
- Attach the `CitiesAdapter` to the `RecyclerView`

The Results

The UI is fairly straightforward: a scrolling list of rows that contains the city name, country or area the city resides in, and its 2015 population:

Figure 2: Room Paging Demo

If you scroll through the list of countries, even with a fairly aggressive fling operation, the list scrolls smoothly. On a well-equipped Android device there are no blank rows, as Room is able to load the 50-at-a-time pages fairly quickly and make

them available to the PagingListAdapter, so we do not see any gaps.

Obviously, not all sources of data will be that quick to load, and not all Android devices are powerful. You will need to run your own experiments with your own data and test devices to determine what the best thing to do is when PagingListAdapter lacks a model object for a particular position that has scrolled into view.

What About RxJava?

In that sample, CitiesViewModel holds a LiveData of our PagedList, obtained from a LivePagedListBuilder.

If you prefer to use RxJava throughout your app — rather than perhaps a mix of LiveData and RxJava — you can use RxPagedListBuilder. This is in the android.arch.paging:rxjava2 artifact, which is newer than the rest of Paging and therefore may not necessarily have the same artifact versions.

You create an instance of RxPagedListBuilder using the same parameters as you would for a LivePagedListBuilder, such as the DataSource.Factory and the page size. You can then call:

- buildObservable() to return an Observable for your PagedList, or
- buildFlowable(), to return a Flowable for your PagedList

Optionally, before the build...() method call, you can call setFetchScheduler() to provide the Scheduler to use for the data loading (e.g., Schedulers.single()) and/or setNotifyScheduler() to provide the Scheduler for delivering the results (e.g., AndroidSchedulers.mainThread()). The default notify scheduler is the main application thread, while the defualt fetch scheduler is one from a thread pool shared by many of the Architecture Components.

LiveData and Bound Services

Services have been an oft-overused bit of the Android SDK. Still, a foreground service is a key part of many apps, providing ongoing functionality to the user even though the app's UI is gone. For example, most audio player apps (music, audiobooks, etc.) rely on foreground services to play that audio.

One way to communicate with a service is to bind to it, using `bindService()`. This is a very flexible option, as you can define your own API that the service exposes and clients consume. However, binding to services can be tricky, particularly when configuration changes come into the picture.

And any time that configuration changes become a problem, `ViewModel` and `LiveData` should be a place to turn to come up with a solution.

So, in this chapter, we will explore how you can wrap your bound service in a Architecture Components-based API.

Old API, New Coat of Paint

Binding to a service sounds easy: call `bindService()` on a `Context`, supplying a `ServiceConnection` object. That `ServiceConnection` object is called with `onServiceConnected()` once the binding is ready, and you are passed an `IBinder` object that you can use to get at the API exposed by the service. Later, you can call `unbindService()`, passing in the same `ServiceConnection`, to drop the connection to the service.

However, there are three problems:

1. It is unsafe to use different `Context` objects for binding and unbinding

2. The `ServiceConnection` object is state — we need to hold onto that to be able to unbind
3. If the service has no other reason to be around (e.g., something called `startService()` on it), unbinding from the service immediately destroys it... even if you might bind again milliseconds later

This makes it tricky to bind from an activity, as configuration changes run right into all three of those problems.

The classic solution involved using the `Application` singleton for binding and a retained fragment for holding onto the `ServiceConnection` across configuation changes. While the `Application` singleton is still a good idea, nowadays we can replace the retained fragment with a `ViewModel`.

Another wrinkle comes with getting data from the service. We can pull such data by calling methods on its API. Or, the service could push data... somehow. For example, we could supply a callback object via the API, which the service can use to provide data updates. However, once again, this callback is part of our state, so we need to think about how we can manage it with our `ViewModel`. In addition, we have threading to consider, particularly if our service is in a separate process from our UI. This is a place where `LiveData` can shine.

Remote Sensors

The `Sensor/LiveService` sample project is another variation on some of the `SensorManager` samples shown elsewhere in the book. This one uses data binding to show the current ambient light reading in a simple UI. And in this case, the `SensorManager` is being used by a service, to which the activity is binding by way of a `LiveData` and `ViewModel` set up for that work.

The AIDL

The service (`LightSensorService`) is going to run in a separate process from the rest of the app. That is not required for services, but for the purposes of this example, it shows a slightly more complex scenario than having the service be in the same process.

Since we are going to use a remote bound service, we need to use AIDL to define the API between the service and its client. The project has two such AIDL files.

One is `ILightReporter`, which is the AIDL interface that the service will expose as its API:

```
package com.commonsware.android.livedata;

import com.commonsware.android.livedata.ILightCallback;

interface ILightReporter {
    void registerCallback(ILightCallback cb);
    void unregisterCallback(ILightCallback cb);
}
```

(from Sensor/LiveService/app/src/main/aidl/com/commonsware/android/livedata/ILightReporter.aidl)

This offers two methods, for registering and unregistering a callback to find out about new light readings. That callback is defined by the other AIDL interface, `ILightCallback`:

```
package com.commonsware.android.livedata;

interface ILightCallback {
    void onLightEvent(float value);
}
```

(from Sensor/LiveService/app/src/main/aidl/com/commonsware/android/livedata/ILightCallback.aidl)

Our client will implement this callback and it will receive light sensor readings (as a single `float` of the light level in lux) from the service.

Callbacks are not your only option for receiving asynchronous data updates from a service — you could use a `Messenger`, `ResultReceiver`, `PendingIntent`, etc. Most can follow the same basic pattern shown in this example for a `LiveData` wrapper.

The Service and the Process

`LightSensorService` is registered in the manifest with the `android:process` attribute, to have that service run in a separate private process from the rest of the app:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.android.livedata"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0">
```

```
  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <service
      android:name=".LightSensorService"
      android:exported="false"
      android:process=":light" />
  </application>

</manifest>
```

(from Sensor/LiveService/app/src/main/AndroidManifest.xml)

`LightSensorService` wraps a `SensorManager` and has a `Reporter` implementation of the `ILightReporter` interface to serve as its binder:

```
package com.commonsware.android.livedata;

import android.app.Service;
import android.content.Intent;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.IBinder;
import android.os.RemoteCallbackList;
import android.os.RemoteException;

public class LightSensorService extends Service {
  private SensorManager sensorManager;
  private Reporter reporter=new Reporter();

  @Override
  public void onCreate() {
    super.onCreate();

    sensorManager=(SensorManager)getSystemService(SENSOR_SERVICE);
  }

  @Override
  public IBinder onBind(Intent intent) {
```

```
    return reporter;
  }

  private class Reporter extends ILightReporter.Stub {
    private RemoteCallbackList<ILightCallback> callbacks=new RemoteCallbackList<>();

    @Override
    public void registerCallback(ILightCallback cb) {
      callbacks.register(cb);

      if (callbacks.getRegisteredCallbackCount()==1) {
        sensorManager.registerListener(listener,
          sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT),
          SensorManager.SENSOR_DELAY_UI);
      }
    }

    @Override
    public void unregisterCallback(ILightCallback cb) {
      callbacks.unregister(cb);

      if (callbacks.getRegisteredCallbackCount()==0) {
        sensorManager.unregisterListener(listener);
      }
    }

    final private SensorEventListener listener=new SensorEventListener() {
      @Override
      public void onSensorChanged(SensorEvent event) {
        callbacks.beginBroadcast();

        for (int i=0;i<callbacks.getRegisteredCallbackCount();i++) {
          ILightCallback cb=callbacks.getBroadcastItem(i);

          try {
            cb.onLightEvent(event.values[0]);
          }
          catch (RemoteException e) {
            // we tried!
          }
        }

        callbacks.finishBroadcast();
      }

      @Override
      public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // unused
      }
    };
  }
}
```

(from Sensor/LiveService/app/src/main/java/com/commonsware/android/livedata/LightSensorService.java)

`Reporter` uses `RemoteCallbackList` to keep track of the callback objects supplied by clients. `RemoteCallbackList` helps keep track of clients that crash, removing their registered callbacks from the list.

Once we have a registered callback, we begin requesting `TYPE_LIGHT` sensor readings from the `SensorManager`, routing those to a `SensorEventListener`. That listener iterates over the callbacks managed by that `RemoteCallbackList` and calls `onLightEvent()` on each. If we get a `RemoteException` when calling the callback, we just move on — `RemoteCallbackList` should detect that the client is no longer around and remove its callback from the list. Ideally, we would create a custom subclass of `RemoteCallbackList` and override `onCallbackDied()` to find out about it, so we can unregister our `SensorEventListener` if we have no more callbacks — this is left as an exercise for the reader.

The net result of this work is that our clients that register callbacks find out about light sensor events via those callbacks.

The LiveData and the ViewModel

We only need to receive light sensor readings in the activity when the activity is visible. Otherwise, such readings are just a waste of time, battery, etc. This is an ideal case for a lifecycle-aware component, and `LiveData` fits that bill nicely. So, we have a `ServiceLiveData` that wraps up the binding and callback work with the service and emits a stream of `Float` objects for the light sensor readings:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.LiveData;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;

public class ServiceLiveData extends LiveData<Float>
  implements ServiceConnection {
  private final Context app;
  private ILightReporter reporter;

  ServiceLiveData(Context ctxt) {
    app=ctxt.getApplicationContext();
  }

  @Override
  protected void onActive() {
    super.onActive();

    app.bindService(new Intent(app, LightSensorService.class), this, Context.BIND_AUTO_CREATE);
  }

  @Override
  protected void onInactive() {
```

```
    goAway();
    app.unbindService(this);

    super.onInactive();
  }

  @Override
  public void onServiceConnected(ComponentName name, IBinder service) {
    reporter=ILightReporter.Stub.asInterface(service);

    try {
      reporter.registerCallback(cb);
    }
    catch (RemoteException e) {
      Log.e(getClass().getSimpleName(), "Exception registering callback", e);
    }
  }

  @Override
  public void onServiceDisconnected(ComponentName name) {
    reporter=null;
  }

  private void goAway() {
    try {
      reporter.unregisterCallback(cb);
    }
    catch (RemoteException e) {
      Log.e(getClass().getSimpleName(), "Exception unregistering callback", e);
    }
    finally {
      reporter=null;
    }
  }

  private final ILightCallback cb=new ILightCallback.Stub() {
    @Override
    public void onLightEvent(float value) {
      postValue(value);
    }
  };
}
```

(from Sensor/LiveService/app/src/main/java/com/commonsware/android/livedata/ServiceLiveData.java)

We get the `Application` singleton in the constructor, then use that in `onActive()` and `onInactive()` to bind and unbind from the service. In the case of this app, binding will start the service, and unbinding will destroy it, as there is no other reason for the service to be around. In other scenarios — such as the audio player — the lifetime of the service will be managed by `startService()` and `stopService()` (or `stopSelf()`), and binding/unbinding merely is for the communications channel between the client and the service.

Once we are bound, in `onServiceConnected()`, we register our callback. Since we have to use AIDL here for cross-process service communication, the callback is a

subclass of `ILightCallback.Stub`. When it is called with `onLightEvent()`, it uses `postData()` to update the `LiveData` — `postData()` works from a background thread, and we are not in control over what thread is used for `onLightEvent()`.

Our `ServiceViewModel` simply exposes a `ServiceLiveData` named `sensorLiveData`:

```
package com.commonsware.android.livedata;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.support.annotation.NonNull;

public class ServiceViewModel extends AndroidViewModel {
  public final ServiceLiveData sensorLiveData;

  public ServiceViewModel(@NonNull Application app) {
    super(app);

    sensorLiveData=new ServiceLiveData(app);
  }
}
```

(from Sensor/LiveService/app/src/main/java/com/commonsware/android/livedata/ServiceViewModel.java)

The Activity and the Layout

`MainActivity` uses `ServiceViewModel` in `onCreate()`:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.ViewModelProviders;
import android.databinding.BindingAdapter;
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.widget.TextView;
import com.commonsware.android.livedata.databinding.MainBinding;

public class MainActivity extends FragmentActivity {
  @BindingAdapter("android:text")
  public static void setLightReading(TextView tv, Float value) {
    if (value==null) {
      tv.setText(null);
    }
    else {
      tv.setText(String.format("%f", value));
    }
  }
```

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    MainBinding binding=MainBinding.inflate(getLayoutInflater());
    ServiceViewModel vm=ViewModelProviders.of(this).get(ServiceViewModel.class);

    binding.setViewModel(vm);
    binding.setLifecycleOwner(this);
    setContentView(binding.getRoot());
  }
}
```

(from Sensor/LiveService/app/src/main/java/com/commonsware/android/livedata/MainActivity.java)

Principally, we want to bind it to our `main` layout resource:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>

  <data>

    <variable
      name="viewModel"
      type="com.commonsware.android.livedata.ServiceViewModel" />
  </data>

  <android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginBottom="8dp"
      android:layout_marginEnd="8dp"
      android:layout_marginStart="8dp"
      android:layout_marginTop="8dp"
      android:text="@string/label_light"
      android:textAppearance="@android:style/TextAppearance.Material.Large"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      app:layout_constraintVertical_bias="0.25" />

    <TextView
      android:layout_width="wrap_content"
```

```
      android:layout_height="wrap_content"
      android:layout_marginBottom="8dp"
      android:layout_marginEnd="8dp"
      android:layout_marginStart="8dp"
      android:layout_marginTop="8dp"
      android:text="@{viewModel.sensorLiveData}"
      android:textAppearance="@android:style/TextAppearance.Material.Large"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      app:layout_constraintVertical_bias="0.75" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

(from Sensor/LiveService/app/src/main/res/layout/main.xml)

The layout uses a binding expression to find out about changes in the light sensor, by tying our `ServiceLiveData` to a `TextView`. The `setLightReading()` `BindingAdapter` in `MainActivity` will wind up being used by the data binding framework to take the `Float` values from `ServiceLiveData` and pour them into the `TextView`.

And, so that the data binding framework can use the `LiveData` properly, not only do we call `setViewModel()` on the code-generated `MainBinding`, but we also call `setLifecycleOwner()` to give the data binding framework the `LifecycleOwner` to use.

Exploring Architecture

Immutability

Generally, developers like setters.

After all, it stands to reason that if you can get a piece of data from an object, you should be able to modify that piece of data in that same object. We are used to read-write data structures, whether in memory or persisted.

However, there are some costs to allowing objects to be mutable (in other words, able to be modified). In some cases, you can have a more robust app architecture if you consider objects to be immutable and unable to be modified. The objects get replaced outright, rather than changing their contents. This winds up being a bit more reminiscent of a transactional database, where changes are applied as a unit, rather than piecemeal.

In this chapter, we will explore the benefits (and costs) of immutability and how to create immutable (or mostly immutable) objects in Java.

The Benefits of Immutability

Having immutable objects — particularly for things like models — is not a new concept. Immutability has had its adherents for quite some time. It is what lead to libraries like AutoValue for Java, and immutability features in languages like Kotlin.

So, why go with immutability?

No Dirty Data

A perpetual problem with model objects (and related objects, such as view-models), is knowing what data changed, when, and by what. That comes from promiscuous

use of setters, blindly changing data that might be in use already.

For example:

- You implement a model cache, shared between your code that gets data from the server and your UI code, to minimize memory consumption
- You construct a view-model from a model object, representing data to be presented to the user in the UI
- You use two-way data binding, so user interactions with the UI directly update the view-model
- Your code for communicating with the server finds out about an update that happens to affect that same model object, and it updates the model object in the cache
- Your UI code, after the user clicks Save to commit the data changes, uses the view-model contents to update the model object

Now, we have possible data consistency issues:

- The model started in state A
- The view-model was created based on that state A
- The user mutates the view-model and moves it to state B
- The server-sync code mutates the model and moves it to state C
- The view-model updates the model and moves it to state B... potentially ignoring the changes the server-sync code made that resulted in state C

Immutable model and view-model objects do not prevent this sort of situation, but they help to make it a bit more obvious. Changing the state is a more obvious action; it is not merely a matter of calling some setters.

Thread Safety

If more than one thread has access to the same data, and that data can change, we wind up having to synchronize access to that data, so that changes can be made atomically. We do not want a thread to be part-way through updating the data when another thread tries reading it, as the partly-updated data may be in an inconsistent state at that moment. We wind up using `synchronized` and `CopyOnWriteArrayList` and all sorts of other constructs to allow mutable data to be shared between threads.

This goes away if the data is immutable. Multiple threads can read the same, unchanging data whenever they want without issue. Now, we limit our synchronization to more specific scenarios, such as updating shared caches: so long

as we are replacing a cache entry atomically, all consumers of the cache can run in parallel, if the cache entries themselves are immutable.

Functional Programming

One way to combat the complexities of multi-threaded development is to use functional programming. Functional programming is based on pure functions: methods (or the equivalent) that operate solely on input parameters, with no side-effects that affect the operation of the program.

RxJava is based on functional programming concepts. We build up chains of RxJava operators, where each operator applies some sort of function to the input, such as the `map()` operator applying a `Function` to convert objects from one type to another.

Immutability is one way of imposing a contract upon yourself, as a developer, to avoid side effects. Calling a setter is a very casual act in programming, even if calling that setter introduces a side effect. Immutability enforces the creation of new objects, ideal for use in pure functions, where the function can create objects to return but cannot change the parameters' contents and cause side effects.

The Costs of Immutability

Immutability is not without its downsides. Partly, that comes from its use in languages where immutability is not an integrated feature, such as Java. And partly, that comes from environments where mutability is the norm; you may be unable to impose immutability due to environmental restrictions.

Partial Immutability Problems

In a language like Java, where immutability is not a built-in feature, you need to implement it manually, ensuring that your to-be-immutable objects lack setters or other means of manipulating their contents. On the surface, this may not seem very difficult. After all, *not* writing code (e.g., setters) should take less time than would actually writing that code.

However, not everything can be made immutable just by avoiding setters.

For example, suppose we have:

```
class FooModel {
```

```
  final String bar;
  final List<GooModel> goos;

  FooModel(String bar, List<GooModel> goos) {
    this.bar=bar;
    this.goos=goos;
  }
}
```

There are no setters, and both fields are `final`. This is immutable, right?

Actually, no, because the `goos` `List` might be mutable. If this is just an ordinary `ArrayList`, for example, holders of a `FooModel` instance can call `add()` or `remove()` on the `goos` field, changing its contents.

This can be improved somewhat via the `Collections` class and its `unmodifiableList()` method:

```
class FooModel {
  final String bar;
  final List<GooModel> goos;

  FooModel(String bar, List<GooModel> goos) {
    this.bar=bar;
    this.goos=Collections.unmodifiableList(goos);
  }
}
```

Now, `goos` will fail if you attempt to call `add()`, `remove()`, etc. on it.

However, not all collection types have a corresponding `unmodifiable...()` method. Plus, you need to remember to use the `unmodifiable...()` method, such as we do here in the constructor. And, what if `GooModel` is mutable? Holders of a `FooModel` could reach into `goos`, pluck out a `GooModel`, and change it.

Creating surface-level immutability is not that hard, even in Java. The challenge is in having immutability "all the way down".

Some Things Want Setters

Unfortunately, some things really want setters or other forms of mutability:

- An interface might imply mutability. `Spannable`, for example, has `setSpan()`

and `removeSpan()` methods, implying mutability.

- Some frameworks might require mutability, at least for some features. Data binding, for example, works with immutable objects... except for two-way binding, which requires some means to modify the existing bound object. Even for one-way binding, `Observable` requires that the object support registering and removing listeners, which itself is an aspect of mutability.

In these and similar cases, avoiding mutability may be impossible, just because you are trying to use something that itself expects some degree of mutability.

Garbage, To Be Collected

One big problem with immutability is that it leads to lots of data copying. Instead of simply updating a field of a model object with a new value, we create a new instance of that model object. We cannot even use an object pool to help minimize the garbage that gets created, because typically we cannot reuse an existing object... because to reuse it, we often need to fill in different data, and that requires mutability.

The copies might be shallow copies, reducing the amount of garbage. Going back to the earlier example, we could have:

```
class FooModel {
  final String bar;
  final List<GooModel> goos;

  FooModel(String bar, List<GooModel> goos) {
    this.bar=bar;
    this.goos=Collections.unmodifiableList(goos));
  }

  FooModel withNewBar(String bar) {
    return(new FooModel(bar, this.goos));
  }
}
```

Here, we create a new `FooModel`, but both the old and the new instance of `FooModel` share the same `goos` collection. If `goos` is immutable, sharing it between two `FooModel` instances is not a problem. So, we consume the extra memory for an extra `FooModel`, but not an extra list of `GooModel` instances, keeping the memory consumption down.

However, there is little doubt that immutability leads to more garbage in Java. On

Android 5.0+, ART's garbage collector will help reduce the impact of this garbage, but it cannot completely eliminate its effects.

Immutability via AutoValue

Many developers who elect to make immutable classes in Java elect to use Google's AutoValue library. This library uses annotations and code generation to help enforce immutability, while also handling aggravating details like implementing `equals()`, `hashCode()`, and so forth.

For basic stuff, using AutoValue is fairly simple: implement an `abstract` class with `abstract` getter methods for the data that you want the immutable class to hold. Add the `@AutoValue` annotation — along with the dependency that supplies it — and AutoValue takes over from there.

Earlier in the book, we had the `Sensor/LiveList` sample app, where we wrapped the `SensorManager` in a `LiveData`. The `Sensor/AutoSensor` sample project is a clone of that one, where we use AutoValue for the event objects.

The original project had a simple `Event` static class inside of `SensorLiveData`, using `final` for its limited immutability:

```
  static class Event {
    final Date date=new Date();
    final float[] values;

    Event(SensorEvent event) {
      values=new float[event.values.length];

      System.arraycopy(event.values, 0, values, 0, event.values.length);
    }
  }
```

(from Sensor/LiveList/app/src/main/java/com/commonsware/android/livedata/SensorLiveData.java)

The revised project pulls that `Event` class out to a top-level `AutoSensorEvent` class and applies AutoValue to it:

```
package com.commonsware.android.livedata;

import android.hardware.SensorEvent;
import com.google.auto.value.AutoValue;
import java.util.ArrayList;
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Date;
import java.util.List;

@AutoValue
abstract class AutoSensorEvent {
  abstract long date();
  abstract List<Float> values();

  static AutoSensorEvent from(SensorEvent event) {
    ArrayList<Float> values=new ArrayList<>();

    for (float value : event.values) {
      values.add(value);
    }

    return(new AutoValue_AutoSensorEvent(System.currentTimeMillis(),
      Collections.unmodifiableList(values)));
  }
}
```

(from Sensor/AutoSensor/app/src/main/java/com/commonsware/android/livedata/AutoSensorEvent.java)

Annotating an `abstract` class with `@AutoValue` causes AutoValue to find all getter-style `abstract` methods — in this case, `date()` and `values()`. AutoValue then code-generates a shadow class, `AutoValue_AutoSensorEvent`, that is a concrete implementation of the `AutoSensorEvent` API. We use the concrete class constructor to make instances of an `AutoSensorEvent`-compatible class. Outside parties using `AutoSensorEvent` should neither know nor care that the actual implementation is actually `AutoValue_AutoSensorEvent`. The `AutoValue_AutoSensorEvent` class not only handles our two data values but also the `equals()`, `hashCode()`, and `toString()` methods as well.

Our `from()` factory method sets up the data to be passed to the `AutoValue_AutoSensorEvent` constructor. We use `unmodifiableList()` to ensure that nobody can modify the contents of the `values()` `List`, and since `Float` itself is immutable, that makes `values()` immutable "all the way down". Similarly, the `long` that is returned by `date()` is immutable, so nothing can be changed in the `AutoSensorEvent`.

All of this is possible because we are adding AutoValue's dependencies:

```
dependencies {
  implementation 'com.android.support:recyclerview-v7:28.0.0'
```

```
  implementation 'com.android.support:support-fragment:28.0.0'
  implementation 'android.arch.lifecycle:runtime:1.1.1'
  implementation 'android.arch.lifecycle:livedata:1.1.1'
  compileOnly 'com.google.auto.value:auto-value:1.5.2'
  annotationProcessor 'com.google.auto.value:auto-value:1.5.2'
}
```

(from Sensor/AutoSensor/app/build.gradle)

Here, the same dependency (`com.google.auto.value:auto-value`) is used twice. The `annotationProcessor` dependency enables the compile-time handling of `@AutoValue` and related annotations. The `provided` dependency adds in runtime support code that the generated code depends upon.

AutoValue itself has many more features, including:

- Generating an optional builder-style API for constructing instances of the `@AutoValue`-annotated class
- Support for `@Nullable` to indicate if a value can be `null` or not
- Memoization support for caching the results of derived values, which is particularly useful if those calculations are expensive

AutoValue and LiveData

`LiveData` and AutoValue work together nicely. The revised `SensorLiveData` simply uses the `from()` factory method to create `AutoSensorEvent` instances that wrap up the data we want to cache from a `SensorEvent`:

```
package com.commonsware.android.livedata;

import android.arch.lifecycle.LiveData;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;

class SensorLiveData extends LiveData<AutoSensorEvent> {
  final private SensorManager sensorManager;
  private final Sensor sensor;
  private final int delay;

  SensorLiveData(Context ctxt, int sensorType, int delay) {
    sensorManager=
      (SensorManager)ctxt.getApplicationContext()
```

```
        .getSystemService(Context.SENSOR_SERVICE);
    this.sensor=sensorManager.getDefaultSensor(sensorType);
    this.delay=delay;

    if (this.sensor==null) {
      throw new IllegalStateException("Cannot obtain the requested sensor");
    }
  }

  @Override
  protected void onActive() {
    super.onActive();

    sensorManager.registerListener(listener, sensor, delay);
  }

  @Override
  protected void onInactive() {
    sensorManager.unregisterListener(listener);

    super.onInactive();
  }

  final private SensorEventListener listener=new SensorEventListener() {
    @Override
    public void onSensorChanged(SensorEvent event) {
      setValue(AutoSensorEvent.from(event));
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
      // unused
    }
  };

}
```

(from Sensor/AutoSensor/app/src/main/java/com/commonsware/android/livedata/SensorLiveData.java)

AutoValue and Room

Unfortunately, AutoValue and Room 1.x do not work together, at least for `@Entity` classes:

- Room wants to work with a constructor, and an `@AutoValue` `abstract` class cannot have a constructor

- Annotations like `@PrimaryKey` and `@ColumnInfo` go on fields, and an `@AutoValue` class has no fields
- Room's documentation indicates that it wants setter methods or `public` fields, and an `@AutoValue` class has neither

This will be added in a future update to Room.

The Repository Pattern

There are lots of possible ways that your app's data can be stored. It could be local, remote, or both. The local copy could be in SQLite, XML files, JSON files, or other forms. The server could be using REST, GraphQL, gRPC, or something else.

And, on the whole, your UI should not care.

Your UI code has enough problems to deal with. Figuring out where the data comes from (to show the user) and where it goes (after getting input from the user) is more than it should have to bear.

That is where the repository pattern comes into play. In a nutshell: you design a single API that abstracts out all of the storage stuff. The repository implementation deals with all of the decision-making for where the data goes, what all has to get updated, what has to be refreshed from some remote source, and so on. The API just offers "give me X" and "here is an update to Y" and so on — the basic operations that the UI needs in order to function.

Therefore, in many respects, the repository pattern is not significantly different from any other abstraction that one might use. However, since data storage and retrieval is usually the reason why the app exists, it is important to give this pattern some thought.

What the Repository Does

A repository has a few key roles inside of your app.

Manages Data Storage

First and foremost, this is where you isolate all of the details of the data storage, including all the esoteric rules that your app may require (e.g., we have to update the catalog after midnight in the server's time zone).

The repository is responsible for:

- Making any real-time requests of a server that may be necessary, to retrieve data that is not yet available locally
- Managing or directing any in-memory cache of that data
- Saving the data in a local persistent store, whether on a temporary or long-term basis
- Orchestrating any background data transfers that may be necessary (e.g., responding to push requests, periodically synchronizing with a server)

The details of this will vary widely from app to app. Some of those details may be dictated by business requirements. Some of those details may be dictated by the server team. Some of those details might be under your control. As a result, there is no single recipe for implementing a repository — all books like this can do is explain the role, illustrate some implementations, and provide general guidance.

Normalizes Model Objects

Your UI code probably will work best with a nice clean object model representing the data that the app needs to allow the user to see and manipulate.

However, it is quite likely that you will not get a clean object model from the data storage code:

- Plain SQLite uses `Cursor` and `ContentValues`, which do not resemble business objects
- Object wrappers around SQLite like Room may impose their own limitations, such as Room's approach for relations
- Web service APIs cannot model some data structures at all (e.g., M:N relations), requiring some amount of data conversion to craft the desired object model
- Some Web service APIs will have further limitations, because the developers of the Web service had a different vision at the time they created the Web service (e.g., older approaches, targeting other platforms)
- And so on

Your UI code should not have to deal with any of that.

So, another part of the repository is to normalize the data from the data storage into clean model objects that the UI code can consume. So, the repository gets to convert those Retrofit POJOs and those Room POJOs (neither of which may resemble the other) into some consistent POJOs that form the object model that the rest of the app uses.

Provides a Clean Reactive API

The UI code needs to be able to make generic requests for normalized data, with the repository handling all of the "dirty details" for making that happen.

At the same time, the UI code needs to have the patience to allow the repository to do its work. The responsiveness of the repository could range from microseconds to seconds, depending on a lot of environmental factors:

- Is the data that the UI wants in a memory cache? A disk cache?
- Do we have to perform a SQL request? How about a network call?
- Do we have to do several of these things, because the UI is seeking a big object graph, and our data storage options deal in smaller slices?

Here, "reactive" *could* mean RxJava, or possibly `LiveData`. It could be some form of event bus. It could be a callback system. What it *has* to be is asynchronous — the API exposed by the repository has to force the UI code to receive the data at some time in the future, not immediately.

Isolates Rest of App from Strategy Changes

You might be tempted to cut corners on the previous point, and have some APIs exposed by the repository that return immediately. So long as those APIs are set up to gracefully fail — such as returning `null` if the data is not cached in memory — that can be fine. However, in general, that is still not a good idea, for one simple reason: things change. Today, your implementation might support those real-time APIs. Tomorrow, your implementation might not, for any number of reasons:

- You elect to switch to some newer approach that simply lacks an equivalent to the in-memory cache that you are using
- You elect to switch to some newer approach that does not offer its own real-time API, and you need to "pass along" the reactive approach
- You jettison this particular cache because you keep running out of memory

- And so on

If you design a reactive API around a generalized object model, you *should* be able to change the implementation of the repository without requiring changes in the UI code. The only time that the UI code would change is if the data structure itself changes (e.g., new fields or objects added to the object model).

High-Level Repository Strategies

There are many, many ways to implement a repository. How one app approaches it may differ significantly from how another app approaches it, and neither approach is necessarily wrong (or right).

That being said, there are a few commonalities among the approaches that will tend to arise, based on where the data is being stored.

Pure Network

Occasionally, you will have an app where the repository always makes network requests whenever the UI needs data. This is fairly uncommon, as it implies that caching is not an option, and usually there is *some* amount of caching that can be applied to the problem.

Network + Network API Caching

Sometimes, the caching can be provided by whatever API you are using to access that network:

- OkHttp offers integrated caching, assuming that the Web server uses appropriate cache-control headers
- Apollo-Android — a library for making GraphQL requests — offers its own caching, in addition to possibly using caching at the OkHttp layer
- Picasso, Glide, and other image-loading libraries often have their own caches
- And so on

In these cases, other than configuring the cache (e.g., specifying the directory to use for a disk cache), there is little cache-related code in the repository itself.

Network + External Caching

Sometimes, you may want more sophisticated caching than might be offered by the API that you are using to access the network, Or, perhaps the caching required by the app does not match what the libraries offer (e.g., the Web server does not use cache-control headers, as the caching is handled by client-side rules rather than server-side configuration). In those cases, you need to handle caching "above" the software layer represented by those networking APIs.

The Store library offers an "all-in-one" solution for this, though the learning curve is steep.

Network + First-Class Persistence

Sometimes — particularly for an app that offers rich offline functionality – you need to put the local storage first in your mind, with network functionality serving in a "sync" role.

While a robust caching system can help with "offline-first" apps, you are limited in how you can access that cached data to whatever APIs are offered by the caches. For caches integrated into the networking APIs, that means that all you can do is make a network request and deal with the failures when the device is offline and the data is not cached. Often you have no way of examining the cache to try to determine what you *can* do locally.

Also, most caching systems are designed for read operations, where the "system of record" is the server and the cache is a replica of data retrieved from that server. Many caches offer little to no support for buffering write operations while the device is offline. And caches, by definition, are never a "system of record", and sometimes the *device* is the primary storage location, with the network serving in the role as a backup.

For these, your repository will be built around a rich local storage API: SQLite, an object database, etc. The repository will be responsible for the network I/O as well, though that network I/O may be a "side" piece of functionality, not used in the direct fulfillment of requests from the UI.

Persistence-Only

Sometimes, there is no server. In those cases, the repository wraps around your local

storage API of choice, with the abstraction helping to isolate you from your *choice* of local storage API, in case you change your mind later.

Let's Roll the Dice

In *The Busy Coder's Guide to Android Development*, one of the sample apps is a "diceware" app, to help you generate a passphrase made up of a series of randomly-selected words, such as `correct horse battery staple`. However, that sample puts most of the work inside a single fragment, which is messy. So, let's rebuild that app, hiding all of the data-loading details in a repository, with a view-model to mediate communications between the fragment and the repository. The results can be found in the `Diceware/Repository` sample project.

The Repository

In our case, the words come from two locations: a "baked in" word list in `assets/` and a word list of the user's choosing, obtained via `ACTION_OPEN_DOCUMENT`. However, the data structure for each is the same: a list of words, one per line. Hence, we do not have a sophisticated data model, only a list of strings. So our repository does not need to worry about normalizing disparate model objects, though we might if we obtained words from some Web service. And, our repository does not need to worry about data modification, as the word lists are treated as read-only.

However, we still need a nice reactive API. The code for getting the words from a user-chosen document is a bit different from the code for getting the words from an asset. Moreover, if we want to cache the words, we need to handle the case where we have not yet loaded the words and the case where the words are cached.

API

In the end, what our UI needs is a set of randomly-selected words, with the UI providing the number of words and the source of those words.

To that end, `Repository` has a single instance method that is exposed to the rest of the app: `getWords()`. It takes the `Uri` representing the data source and the number of words to return. The words themselves will be a `List` of `String` objects. We wrap that in an RxJava `Single`, as we do not know how long it will take to come up with those words at compile time, since the word list from the data source may not be cached yet. However, we do know that this is a one-shot event, and so `Single` makes more sense than does a generic `Observable`.

The `Repository` is a singleton, so we will have a `static` method named `get()` to retrieve that singleton, given a `Context` to use for lazy initialization.

Implementation

`getWords()` breaks the problem down into two pieces: getting the full word list and then choosing a random subset of those words:

```
Single<List<String>> getWords(Uri source, final int count) {
  return(getWordsFromSource(source)
    .map(strings -> (randomSubset(strings, count))));
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/Repository.java)

The `map()` operator delegates the "choose a random subset" work to a `randomSubset()` method, which uses a `SecureRandom` instance to choose the words:

```
private List<String> randomSubset(List<String> words, int count) {
  List<String> result=new ArrayList<>();
  int size=words.size();

  for (int i=0;i<count;i++) {
    result.add(words.get(random.nextInt(size)));
  }

  return(result);
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/Repository.java)

`getWordsFromSource()` needs to look to see if we have a cached copy of the word list for the requested `Uri`. If not, we need to arrange to load and cache those words; otherwise, we can just use the cache. Our cache is `ConcurrentHashMap` mapping the `Uri` to the word list:

```
private final ConcurrentHashMap<Uri, List<String>> cache=new ConcurrentHashMap<>();
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/Repository.java)

`getWordsFromSource()` checks the cache and creates an `Single` chain based on whether or not the words are cached:

```
synchronized private Single<List<String>> getWordsFromSource(Uri source) {
  List<String> words=cache.get(source);
  final Single<List<String>> result;
```

```
    if (words==null) {
      result=Single.just(source)
        .subscribeOn(Schedulers.io())
        .map(uri -> (open(uri)))
        .map(in -> (readWords(in)))
        .doOnSuccess(strings -> cache.put(source, strings));
    }
    else {
      result=Single.just(words);
    }

    return(result);
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/Repository.java)

If the words are cached, our job is simple: just return that word list, wrapped in an `Single`, for `getWords()` to use to come up with the random subset.

If the words are not yet cached, we:

- Wrap the `Uri` in an `Single` to start a chain
- Use an `open()` method to get an `InputStream` on the contents identified by that `Uri` (or pulling in our one-and-only asset if the `Uri` seems to point to assets):

```
private InputStream open(Uri uri) throws IOException {
  String scheme=uri.getScheme();
  String path=uri.getPath();

  if ("file".equals(scheme) && path.startsWith("/android_asset")) {
    return(ctxt.getAssets().open(ASSET_FILENAME));
  }

  ContentResolver cr=ctxt.getContentResolver();

  cr.takePersistableUriPermission(uri, Intent.FLAG_GRANT_READ_URI_PERMISSION);

  return(cr.openInputStream(uri));
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/Repository.java)

- Use a `readWords()` method to convert that `InputStream` into a word list:

```
  private static List<String> readWords(InputStream in) throws IOException {
    InputStreamReader isr=new InputStreamReader(in);
    BufferedReader reader=new BufferedReader(isr);
```

```
    String line;
    List<String> result=new ArrayList<>();

    while ((line = reader.readLine())!=null) {
      String[] pieces=line.split("\s");

      if (pieces.length==2) {
        result.add(pieces[1]);
      }
    }

    return(result);
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/Repository.java)

- Arrange to do all that work on a background thread
- As a side effect, put the word list in the cache for later use, via `doOnSuccess()`

The ViewModel

The `Repository` API is fairly clean, isolating the caching and data loading and stuff behind a reactive response. However, there is one hiccup: each call to `getWords()` results in a new `Single`. This is somewhat of a headache for the UI, as we would need a fresh subscription — via a fresh `LiveData` – whenever the user asks for a new set of words, or changes the word count, or opens a new word list. That is on top of having to manage the subscriptions across lifecycle events.

What would be nice is if the UI could have a single `Observable`, on which all the words would come in, regardless of the trigger (including getting a set of words on first launch). We would still have to deal with lifecycle events, but we have `LiveData` for that.

So, this app has a `ViewModel` implementation — named `PassphraseViewModel` – that offers a `LiveData` of the incoming words that the UI can use, in addition to tracking our current word source and word count across configuration changes.

Repository Integration

The `PassphraseViewModel` constructor takes, among other things, a `Context` as a parameter, to use to retrieve the `Repository` singleton, held in a field named `repo`. We also have `source` and `count` fields to hold how many words we should retrieve

and where we should retrieve them from, initialized to some starter values:

```
private final Repository repo;
private Uri source=Uri.parse("file:///android_asset/eff_short_wordlist_2_0.txt");
private int count=6;
private Disposable sub=Disposables.empty();
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

Getting words to the UI is handled by a `words()` method, that returns a `LiveData` for random word list subsets. That `LiveData` is in the form of a `liveWords` field:

```
LiveData<List<String>> words() {
  return(liveWords);
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

As noted, though, we will get multiple `Single` instances from the `Repository`, one for each `getWords()` call that we need. If we want the UI to use this stable `LiveData` instance, we need a way to feed different `Single` results to it over time.

The approach that `PassphraseViewModel` takes is to have `liveWords` be a `MutableLiveData`:

```
private final MutableLiveData<List<String>> liveWords=new MutableLiveData<>();
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

The `PassphraseViewModel` has a `refresh()` method. Partly, this is used literally for a "refresh" operation, to load a fresh batch of words given the current `count` and `source` values. In fact, everything else that needs to trigger loading words routes through `refresh()`. `refresh()` calls the `getWords()` method that we have on `Repository` and forwards the events to the `liveWords` by using a Java 8 method reference to tie the `subscribe()` of the `Single` to `postValue()` of the `liveWords`:

```
void refresh() {
  sub.dispose();
  sub=repo.getWords(source, count)
    .observeOn(Schedulers.io())
    .subscribe(liveWords::postValue);
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

The net effect is that every time `refresh()` is called, the `liveWords` eventually will deliver a new random subset of the current word list.

Saving State

We need to get the `source` and the `count` from the UI, for use in `refresh()`. And, along the way, we can hold onto that information across configuration changes, since this is a `ViewModel`. Plus, we can also have the `PassphraseViewModel` store this information in the saved instance state `Bundle`, so the view-model is the single "source of truth" for the current `source` and `count`.

To that end, the constructor on `PassphraseViewModel` takes a saved instance state `Bundle` as input and — if the `Bundle` is not `null` — populates the `source` and `count` from its contents:

```
PassphraseViewModel(Context ctxt, Bundle state) {
  repo=Repository.get(ctxt);

  if (state!=null) {
    source=state.getParcelable(STATE_SOURCE);
    count=state.getInt(STATE_COUNT, 6);
  }

  refresh();
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

The constructor also calls `refresh()`, to queue up the first random set of words, so we can populate the UI as quickly as possible.

`PassphraseViewModel` then has its own `onSaveInstanceState()`, where it fills in the state `Bundle` using the same keys that its constructor uses to read the values out:

```
void onSaveInstanceState(Bundle state) {
  state.putParcelable(STATE_SOURCE, source);
  state.putInt(STATE_COUNT, count);
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

(hat tip to Danny Preussler for the idea of centralizing both view-model and saved instance state logic in the `ViewModel`)

The Factory

However, by default, the Architecture Components' `ViewModel` system has no way to

create an instance of `PassphraseViewModel`. After all, it has no idea what this `Bundle` is.

To help with that, `PassphraseViewModel` has a `Factory` nested class that implements `ViewModelProvider.Factory`. This provides the "glue" for tying the Architecture Components to `PassphraseViewModel`, by creating an instance of `PassphraseViewModel` as needed:

```
  static class Factory implements ViewModelProvider.Factory {
    private final Bundle state;
    private final Context ctxt;

    Factory(Context ctxt, Bundle state) {
      this.ctxt=ctxt.getApplicationContext();
      this.state=state;
    }

    @NonNull
    @Override
    public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {
      return((T)new PassphraseViewModel(ctxt, state));
    }
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

When we create an instance of the `Factory`, we need to provide a `Context` (such as the `Activity` hosting our UI) and the incoming saved instance state `Bundle`, for the `Factory` to pass along to the newly-created instance.

The Fragment

The launcher (and only) activity — `MainActivity` — simply sets up a `PassphraseFragment`:

```
package com.commonsware.android.diceware;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity {
  @Override
  protected void onCreate(Bundle state) {
    super.onCreate(state);

    if (getSupportFragmentManager().findFragmentById(android.R.id.content) == null) {
      getSupportFragmentManager().beginTransaction()
        .add(android.R.id.content,
```

```
          new PassphraseFragment()).commit();
    }
  }
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/MainActivity.java)

All of the real UI/UX work resides in the fragment.

The UI

The UI for `PassphraseFragment` consists of a `TextView` for the words, wrapped in a `CardView` to make it a bit more aesthetically interesting:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="8dp">

  <android.support.v7.widget.CardView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:padding="8dp">

    <TextView
      android:id="@+id/passphrase"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="20sp"
      android:typeface="monospace" />
  </android.support.v7.widget.CardView>

</FrameLayout>
```

(from Diceware/Repository/app/src/main/res/layout/activity_main.xml)

The core of our UI setup is in `onViewCreated()`:

```
  @Override
  public void onViewCreated(View view, Bundle state) {
    super.onViewCreated(view, state);

    passphrase=view.findViewById(R.id.passphrase);
    viewModel=ViewModelProviders
      .of(this, new PassphraseViewModel.Factory(getActivity(), state))
      .get(PassphraseViewModel.class);
    updateMenu();
```

```
    viewModel.words().observe(this,
      words -> passphrase.setText(TextUtils.join(" ", words)));
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

Here, we:

- Retrieve the `passphrase` `TextView` to hold our random word list subset
- Obtain our `PassphraseViewModel`, by way of `ViewModelProviders` and our `Factory`
- Update our menu, described in the next section
- `observe()` our `LiveData`, taking the list of words and populating the `TextView`, joining those words with spaces

For a newly-created `PassphraseViewModel`, the constructor's call to `refresh()` will give us some words to show automatically. On a configuration change, our `LiveData` will hand back our last set of words automatically. Plus, the `LiveData` handles the rest of the lifecycle work for us.

The Menu

The fragment also has a menu, with a drop-down for the word count, a "refresh" item to get a fresh random subset of words, and an "open" item to choose a word list document:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/word_count"
    android:showAsAction="ifRoom"
    android:title="@string/menu_words">
    <menu>
      <group android:checkableBehavior="single">
        <item
          android:id="@+id/word_count_4"
          android:title="4" />
        <item
          android:id="@+id/word_count_5"
          android:title="5" />
        <item
          android:id="@+id/word_count_6"
          android:checked="true"
          android:title="6" />
```

```
        <item
          android:id="@+id/word_count_7"
          android:title="7" />
        <item
          android:id="@+id/word_count_8"
          android:title="8" />
        <item
          android:id="@+id/word_count_9"
          android:title="9" />
        <item
          android:id="@+id/word_count_10"
          android:title="10" />
      </group>
    </menu>
  </item>
  <item
    android:id="@+id/refresh"
    android:icon="@drawable/ic_cached_white_24dp"
    android:showAsAction="ifRoom"
    android:title="@string/menu_refresh" />
  <item
    android:id="@+id/open"
    android:enabled="false"
    android:showAsAction="never"
    android:title="@string/open" />

</menu>
```

(from Diceware/Repository/app/src/main/res/menu/actions.xml)

The "refresh" item ties directly to the `refresh()` method on the view-model, while the word count items update their checked state and route to a `setCount()` method on the view-model:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
      case R.id.open:
        open();
        return(true);

      case R.id.refresh:
        viewModel.refresh();
        return(true);

      case R.id.word_count_4:
      case R.id.word_count_5:
```

```
      case R.id.word_count_6:
      case R.id.word_count_7:
      case R.id.word_count_8:
      case R.id.word_count_9:
      case R.id.word_count_10:
        item.setChecked(!item.isChecked());
        viewModel.setCount(Integer.parseInt(item.getTitle().toString()));

        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

The "open" item routes to an `open()` method, which brings up an `ACTION_OPEN_DOCUMENT` activity for the user to choose a word list:

```
  private void open() {
    Intent i=
      new Intent()
        .setType("text/plain")
        .setAction(Intent.ACTION_OPEN_DOCUMENT)
        .addCategory(Intent.CATEGORY_OPENABLE);

    startActivityForResult(i, REQUEST_OPEN);
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

If we get a document, that is passed over to a `setSource()` method on the view-model:

```
  @Override
  public void onActivityResult(int requestCode, int resultCode,
                               Intent resultData) {
    if (resultCode==Activity.RESULT_OK) {
      viewModel.setSource(resultData.getData());
    }
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

The `setCount()` and `setSource()` methods on `PassphraseViewModel` not only update their respective fields, but they also call `refresh()`, to deliver a fresh set of words based on the new count or new source of words:

```
void setSource(Uri source) {
  this.source=source;
  refresh();
}

void setCount(int count) {
  this.count=count;
  refresh();
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

As a result, when the user chooses any of those action bar items, if there is an actual state change (e.g., a new count), we get a new roster of words.

We also need to set the checked state of the word count items based on the `count`, either from the default value or from our state (view-model or saved instance state). Since we do not know whether the menu or the view-model will be set up first, we call a central `updateMenu()` method from a couple of places to check the right action bar item:

```
private void updateMenu() {
  if (menu!=null && viewModel!=null) {
    MenuItem checkable=menu.findItem(WORD_COUNT_MENU_IDS[viewModel.getCount()-4]);

    if (checkable!=null) {
      checkable.setChecked(true);
    }
  }
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

And, our fragment's `onSaveInstanceState()` forwards that `Bundle` to the `PassphraseViewModel` for saving the state:

```
@Override
public void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);
  viewModel.onSaveInstanceState(outState);
}
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

The Event Flow

When our fragment is created, we create our view-model. It, in turn, asks the repository to give us our initial random subset of words, triggering some background

file I/O to read in the initial word list. When that is done, the random words wind their way to the fragment, which pops them into the UI.

When the user requests different words — a different count, a different source, or just words that maybe they might like better — the fragment updates the view-model, which in turn asks the repository for words for the now-current word count and word source. Eventually, another random subset of words make its way to fragment, which displays them using the same code as before.

On a configuration change, our newly-recreated fragment winds up connecting to the same view-model as before, courtesy of the Architecture Components' `ViewModel` system. Our `LiveData` gives us back the words we were showing in the previous fragment, so we can show them again.

So, each layer has its role in the event flow:

- The fragment manages the UI, including the menu
- The repository manages the data loading and random-subset work
- The view-model mediates the communications between them, folding all of the disparate `Single` objects from the repository into a single stream of events for the fragment to consume

And, of course, the user gets a reasonably-secure passphrase to use for some app or site that needs it:

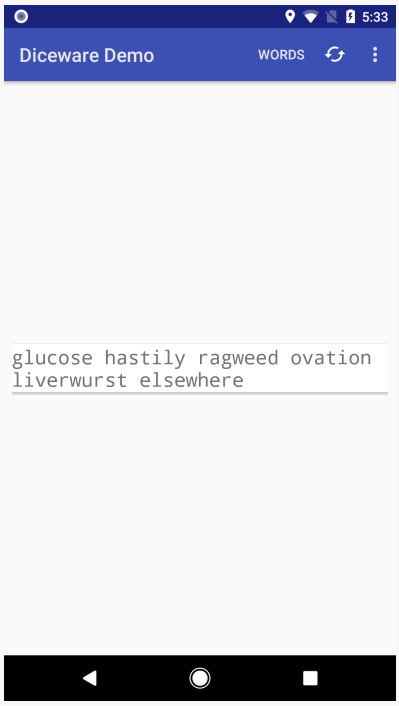


Figure 3: Diceware Demo, Showing Random Words

Blending Data Sources

A repository is not limited to only using a single data source.

Sometimes, the repository will use multiple data sources that are holding model objects. For example, the repository might implement a two-level cache (memory plus disk), with the "system of record" being a Web service. For some requests, the repository would need to check some or all of these locations for the data.

Sometimes, while the primary model data resides in one data source, ancillary data resides in another. For example, perhaps the model data contains a URL to an image, or an image URL can be derived from the model data (e.g., an avatar icon). A typical approach is to have the repository ignore that, allowing the UI layer to request the image from an image-loading library (e.g., Picasso, Glide), which serves as its own repository. But if you need to perform transformations on that image, you might find it better to handle that yourself in your own repository. In that case, while most of

the model data might come from one place (local database, Web service), other aspects of that model data might come from another place.

The `Diceware/Pwned` sample project will demonstrate this, by validating our randomly-generated passphrase to confirm that it has not already been "pwned".

Pwned Passwords

In the parlance of Troy Hunt, a password is "pwned" if it has appeared in a data breach. Sometimes, the breach is of a database where passwords were held in plaintext (inexplicably). Sometimes, the copied database used hashing, but the hash was weak and passwords could be obtained using rainbow tables and other attacks.

Mr. Hunt maintains a database of "pwned passwords". There are several ways of using this database, including a lightweight Web service API.

In particular, this API provides a way to check a password against the database without disclosing the password itself to the Web service. Instead, the client:

- Creates a SHA-1 hash of the password
- Submits the first five characters of that hash
- Receives in response the suffixes of all of the hashes of all of the passwords in the database that start with those five characters

That response will be several hundred entries, but that is still small enough to scan for matches against the rest of the password's hash. This way, all the Web service knows is that you have a password whose hash starts with those five characters; the password itself is never passed to the Web service. This is a nice way to validate a password against the database, as the candidate password is never disclosed.

In principle, a diceware app should not need to use this Web service. It is *very* unlikely that a randomly-generated set of words from decently-long word lists will have been used previously as a password. It is even less likely that it will have been used previously as a password that wound up in Mr. Hunt's database. However, "very unlikely" is not the same as "impossible", and since the Web service API is easy to use, it may be worthwhile to check the generated passphrase.

There are other cases where using this Web service API is more important, such as:

- User-supplied passwords for local data
- User-supplied passwords for an account to be created on your Web service

(where the Web service itself is not checking that password for pwnage)
- Pwnage checking integration into a password safe

PwnedCheck

The `PwnedCheck` class provides a simple RxJava/OkHttp wrapper around the Web service. The Web service does not use JSON, XML, or other conventional data structures for its response, and so a dedicated REST client API like Retrofit will not help much here.

`PwnedCheck` has a one-parameter constructor that takes an `OkHttpClient` instance. If you are already using OkHttp, pass in your existing `OkHttpClient` instance, so you can share the pools that OkHttp maintains (threads, connections, etc.). Or, just create a `new OkHttpClient()` and pass that in.

`PwnedCheck` exposes two methods that supply `Observable` responses related to passphrase validation: `score()` and `validate()`.

score()

The core one is `score()`. This will return an `Integer` — via an `Observable` — representing the number of times the supplied passphrase appears in the database, or `0` if it is not in the database at all:

```
Observable<Integer> score(String passphrase) {
  return Observable.just(passphrase)
    .map(PwnedCheck::getSha1Hex)
    .flatMap(this::fetchCandidates)
    .map(PwnedCheck::findCount);
}
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PwnedCheck.java)

Here, we start an `Observable` chain `just()` on the passphrase. Then, we use `map()` to convert that passphrase to its SHA-1 hash, using `getSha1Hex()`:

```
// based on https://stackoverflow.com/a/33260623/115145

private static String getSha1Hex(String original) throws Exception {
  MessageDigest messageDigest=MessageDigest.getInstance("SHA-1");

  messageDigest.update(original.getBytes("UTF-8"));
```

```
    byte[] bytes=messageDigest.digest();
    StringBuilder buffer=new StringBuilder();

    for (byte b : bytes) {
      buffer.append(Integer.toString((b & 0xff)+0x100, 16).substring(1));
    }

    return buffer.toString();
  }
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PwnedCheck.java)

Then, we `flatMap()` to get a new `Observable`, one that wraps around OkHttp to make the REST request of the Web service, via `fetchCandidates()`:

```
private Observable<FetchResult> fetchCandidates(String sha1) throws IOException {
  String url="https://api.pwnedpasswords.com/range/"+sha1.substring(0, 5);
  Request request=new Request.Builder().url(url).build();

  return Observable.fromCallable(
    () -> new FetchResult(okHttpClient.newCall(request).execute(), sha1));
}
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PwnedCheck.java)

The URL used for the Web service is simply a particular base URL with the first five characters of the SHA-1 has appended as a path segment.

`fetchCandidates()` returns a `FetchResult`, wrapped in an `Observable`. A `FetchResult` is a simple POJO wrapping our hash and the `Response` object from OkHttp:

```
private static class FetchResult {
  final Response response;
  final String sha1;

  private FetchResult(Response response, String sha1) {
    this.response=response;
    this.sha1=sha1;
  }
}
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PwnedCheck.java)

`score()` completes the chain by converting the `FetchResult` into the count of occurrences of the passphrase, using `findCount()`:

```
private static int findCount(FetchResult fetch) throws IOException {
```

```
    String candidates=fetch.response.body().string();
    String suffix=fetch.sha1.substring(5).toUpperCase();

    for (String line : candidates.split("\r\n")) {
      if (line.startsWith(suffix)) {
        return(Integer.parseInt(line.split(":")[1]));
      }
    }

    return 0;
  }
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PwnedCheck.java)

The response from the server is a series of lines. Each line contains the trailing 35 characters of the SHA-1 hash, after the common five-character prefix. Each line also has the number of occurrences of that hash in the database, separated from the hash suffix via a colon. So, `findCount()`:

- Retrieves the entire response as a `String`
- Splits that into lines
- Checks to see if the line starts with the trailing 35 characters of the candidate passphrase's SHA-1 hash
- If it is, pick out the count and return it

So, the chain set up by `score()` results in you getting that score: the number of times the passphrase appears in the database, or `0` if it is not in the database at all.

Note that the `Observable` chain set up by `score()` performs network I/O, so clients will want to use `subscribeOn()` or something to ensure that the work is performed on a background thread.

validate()

Most of the time, though, you do not need the score. You just need to know if the passphrase appears in the database.

One way to do that would be to have an `Observable` of `Boolean`, with `false` indicating that the passphrase is invalid (i.e., it is in the database and therefore is pwned). Another approach is to have an `Observable` chain that throws an exception for invalid passphrases. This approach can then be used as part of RxJava's "retry" options — in the case of this app, we can generate another random set of words and try again.

So, `validate()` wraps the `score()` `Observable` and yields one of two outcomes:

- You get an `Observable` that gives you your passphrase back, so you do not necessarily need to hold onto it elsewhere, or
- a `PwnedException`, if the `score()` is a positive number

```
Observable<String> validate(final String passphrase) {
  return score(passphrase).map(score -> {
    if (score>0) {
      throw new PwnedException();
    }

    return passphrase;
  });
}

private static class PwnedException extends RuntimeException {

}
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PwnedCheck.java)

Adding OkHttp and INTERNET

To make all of this work, the project has the OkHttp dependency, plus it has the `INTERNET` permission requested in the manifest.

Integrating PwnedCheck

Given the `PwnedCheck` class, our `Repository` can now blend it into its work for generating random passphrases based on word lists.

Modifying the Model

In the earlier sample, we considered the "model" to be a list of strings, of a UI-determined length. The UI was responsible for combining those into a passphrase using some delimiter (in this case, a space).

However, the Pwned Passwords API wants a simple passphrase as a `String`. So, we need to modify the app to have the model be a single `String`, not a list. And, we will need to have the `Repository` create the combined string.

This requires a few changes to consumers of the `Repository`.

`PassphraseViewModel` now has a `MutableLiveData` of `String`:

```
  private final MutableLiveData<String> livePassphrase=new MutableLiveData<>();
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

This also affects the method used to get that `LiveData`, now renamed to be `passphraseStream()`:

```
  LiveData<String> passphraseStream() {
    return(livePassphrase);
  }
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PassphraseViewModel.java)

When `PassphraseFragment` observes that `LiveData`, it no longer needs to use `TextUtils` to join the words into a single string. Instead, it gets the already-joined words and can just put them into the `TextView`:

```
  @Override
  public void onViewCreated(View view, Bundle state) {
    super.onViewCreated(view, state);

    passphrase=view.findViewById(R.id.passphrase);
    viewModel=ViewModelProviders
      .of(this, new PassphraseViewModel.Factory(getActivity(), state))
      .get(PassphraseViewModel.class);
    updateMenu();
    viewModel.passphraseStream().observe(this,
      newPhrase -> passphrase.setText(newPhrase));
  }
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/PassphraseFragment.java)

Validating the Passphrase

The original sample's `Repository` had a `getWords()` method that randomly selected the words out of the given source:

```
  Single<List<String>> getWords(Uri source, final int count) {
    return(getWordsFromSource(source)
      .map(strings -> (randomSubset(strings, count))));
  }
```

(from Diceware/Repository/app/src/main/java/com/commonsware/android/diceware/Repository.java)

That method is now `getPassphrase()`, and it involves a slightly longer `Observable` chain:

```
Observable<String> getPassphrase(Uri source, final int count) {
  return(getWordsFromSource(source)
    .map(strings -> (randomSubset(strings, count)))
    .map(pieces -> TextUtils.join(" ", pieces))
    .flatMap(checker::validate)
    .retryWhen(errors -> errors.retry(3)));
}
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/Repository.java)

The first two steps in the chain — `getWordsFromSource()` and the `map()` for `randomSubset()` — are what we had originally.

Next, we use another `map()` to `join()` the words here, rather than in the `PassphraseFragment` as before.

Then, we can use `flatMap()` to pull in our `PwnedCheck` instance, held in a field named `checker`:

```
private final PwnedCheck checker=new PwnedCheck(new OkHttpClient());
```

(from Diceware/Pwned/app/src/main/java/com/commonsware/android/diceware/Repository.java)

At this point, our stream is now an `Observable` of the passphrase itself... unless it fails validation, in which case a `PwnedException` is thrown. So, our final step uses RxJava's `retry()`, which will try the entire chain from the start if directed to, up to 3 times (as we are calling `retry(3)`).

So, if you get a passphrase from the `Observable` returned by `getPassphrase()`, it has been validated by the Pwned Passwords Web service and is guaranteed to be a fresh, un-pwned passphrase. More importantly, other than the change in data type from a list of strings to a single string, nothing outside of the repository knows or cares about the details of how the random passphrase is generated.

Introducing Model-View-Intent

MVC. MVP. MVVM. MVI. These abbreviations get tossed around a lot in app development discussions, and increasingly in Android app development discussions. Those using these abbreviations often think that:

- Everybody knows what they mean, and
- There is a single universal definition for each of those abbreviations, one that everybody holds

In reality, these MV* abbreviations are well-known in some circles and unknown in others. And, even among people who think they know these abbreviations, there is a fair bit of disagreement about what the abbreviations mean, particularly when it comes time to writing actual code. In this chapter, we will explore what these abbreviations mean, with a particular emphasis on the last of the four: MVI, which stands for Model-View-Intent. And, as an illustration of the problems inherent in applying these abbreviations:

- The "model" in Model-View-Intent may or may not be what you might think of as the model
- The "view" in Model-View-Intent is unlikely to be a `View`
- The "intent" in Model-View-Intent is *not* going to be an `Intent`

GUI Architectures

MVC, MVP, MVVM, MVI, and others are GUI architecture patterns. They describe different ways of organizing your code to update your UI based upon user input and other changes in data, such as the results of server requests or database operations. The abbreviations, as you might expect, abbreviate short phrases that are the formal names of these patterns:

- MVC = Model-View-Controller
- MVP = Model-View-Presenter
- MVVM = Model-View-Viewmodel
- MVI = Model-View-Intent

In these, "model" represents some data, and "view" represents some way of visualizing that data. The trailing portion of the GUI architecture name indicates another component that is involved in taking model changes and updating the view, and taking user input (e.g., button clicks, text entry) and updating the models... which in turn updates views.

Some people may find it surprising that there are so many different organization patterns for this work, and that developers spend time debating the merits of one pattern over another. Debates over GUI architectures are reminiscent of debates over text editors (`emacs` versus `vi`), in that they tend to be debates over minutiae and ignore other options (e.g., Sublime Text 3).

Why People Worry About GUI Architectures

Those participating in these debates over GUI architectures will tend to unite against architecture-agnostics, hammering home their belief that you should have *some* formal GUI architecture in your app — the debate merely is over which one. There are some reasons why formal GUI architecture adherents are strident in their beliefs that such architectures are universally beneficial.

Avoiding Known Problems

GUI development can be tricky. Android GUI development can be trickier than others, with things like configuration changes, tasks, and the like adding lots of edge and corner cases to worry about.

If a popular GUI architecture has been applied to Android — and if those results have been published in a blog post, conference presentation, etc. — part of the work will be to address those tricky bits. Some of the points of contention in the GUI architecture debates are how they handle things like configuration changes, where architectures that handle those things gracefully are considered to be better than those that ignore the problem.

This is not to say that using one of these formal GUI architectures is a requirement to address these problem areas in Android. However, *if* you implement an Android

version of one of these GUI architectures, there is a decent chance that these problem areas will be covered as part of that work.

Consistency Between Team Members

Many Android apps are developed by teams, rather than by solo developers. Team members may switch between working on different parts of the code base at different times, and so while certain areas may have a specific "owner", that owner may not be the only one to work in those areas. For example, if a critical bug is discovered while an "owner" is away from the office, somebody else may need to step in and fix the bug, so that users are not harmed any longer than is necessary.

Having team members be able to work in any portion of an app's code base is the reason for overall standards, such as:

- Tabs versus spaces for indents
- Line lengths
- Naming conventions (as opposed to `NamingConventions` or `namingConventions` or `naming_conventions`)
- And so on

At a higher level, teams may elect to standardize on a GUI architecture so that different developers writing different portions of a UI (e.g., different activities) will create similar code. It will be easier for others to adopt and modify that code if it is similar to other areas of the code that were seen previously.

Consistency Over Time

Teams are rarely stable for long. Team members come and go, within the overall organization or departing for greener pastures elsewhere. As such, "onboarding" of team members is important, and having a standardized approach to UI construction, in the form of a specific GUI architecture, can help here.

However, even solo developers are victims of time. Code that might make sense today may make less sense in a year and no sense at all in three years. Developer experience and expertise change, even when the code does not. Having consistent code within an app reduces this problem, in part by making it more likely that the developer will have seen similar code recently. And, adopting a specific GUI architecture means that all the world's prose on that GUI architecture act as documentation for the project that the developer does not have to write.

Why Others Ignore GUI Architectures

It is quite likely that the vast majority of Android apps do not use any of these GUI architectures.

In some cases, this is unintentional, insofar as the developer(s) of the app do not know about such architectures or have not considered them. Not everybody has equal education and experience, and that will be reflected in the tactical and strategic development decisions that they make or participate in making.

However, in some cases, there are clear reasons why a classic MV* GUI architecture is unsuitable.

Atypical Apps

These GUI architectures have been refined for "typical" app structures:

- You have some data (local or remote)
- You want to display that data to the user (in collections or for individual items)
- Often, the user can manipulate that data, adding to it, modifying it, or perhaps deleting it
- Sometimes, the data can change on the fly from outside of the app (e.g., push messages, other forms of server-centered updates)
- You want those data changes to flow back to the data source, plus update the UI of the app as needed

Lots of apps fit that general description. At the same time, lots of apps do not fit that general description, such as:

- Many types of games
- System-integrated tools, like soft keyboards and VPNs
- Musical instrument simulators
- Terminal emulators
- Camera apps
- Calculators
- And so on

Some of these will have their own architecture patterns, perhaps tied to libraries or frameworks. Game development, for example, has its own approaches, often

embodied in toolkits like Unity3D. Those approaches may not resemble the MV* architectures that you might use in, say, a social network client.

It is up to you to decide how well these sorts of GUI architectures fit your particular type of app.

YAGNI and Overhead

The bigger the app, the more likely it is that you will gain benefit from a formal GUI architecture. Such apps are more likely to have more developers and be used for more time, where a formal GUI architecture can yield benefits.

Conversely, the smaller the app, the less likely it is that a formal GUI architecture is necessarily worth it. Or, as the saying goes, "you aren't going to need it" (YAGNI).

Most of the sample apps in this book, like the sample apps from *The Busy Coder's Guide to Android Development*, skip the formalities. That is because they are samples of how to use particular APIs and usually have little code beyond that.

A well-architected app is likely to have more code than an equivalent app that "just gets the job done". The next chapter will profile a simple checklist-style "to-do" app built using the MVI architecture. As you will see, the architecture itself dictates that many more classes be created, above and beyond what might be needed for the core functionality.

No Obvious User Benefit

The counterpart to this issue is that the user rarely, if ever, benefits directly from the use of a formal GUI architecture. Developers may benefit, and their organization (where relevant) may benefit. However, the user is not necessarily going to see anything different. A note-taking app, or a chat client, or a video player, should look the same to the user whether the developer(s) used MVC, MVP, MVVM, MVI, some other GUI architecture... or no specific GUI architecture at all.

It is entirely possible that bugs will exist in an informally-developed app that would not exist had the developers chosen and implemented a formal GUI architecture. However, there is no guarantee that bugs will exist in an informally-developed app. Similarly, there is no guarantee that using an MV* architecture will eliminate all of your bugs by magic. A development team that doesn't pay much attention to GUI architecture — instead choosing to invest that time in an awesome test suite – may have a better app in the end than a team that focused heavily on the architecture

and did a sub-par job at testing.

Choosing a GUI architecture does not give the users any additional features. There are no new marketing buzzwords to tout. The media will not praise the app for its adherence to some GUI architecture. If management thinks that investing in a GUI architecture is slowing down "work that matters", management might steer developers towards other efforts and away from a formal architecture. Frequently, decisions made on this basis turn out to be bad ones... but management is free to make those decisions.

No Consistency In Definitions

Compounding that latter problem is that it may take a while to figure out how to implement a GUI architecture, as they are *very* ill-defined. We toss around terms like "model" and "view" without clear definitions, particularly with respect to concrete scenarios and Android apps. As a result, not only does the development team need to debate which architecture to use, but also what that architecture really means in practice.

A Rough Comparison of GUI Architectures

For the purposes of this book, the primary differences between the major GUI architectures is how data and control flow from the model data and the visual representation of that model data (the view).

The simplest, in many respects, is Model-View-Presenter:

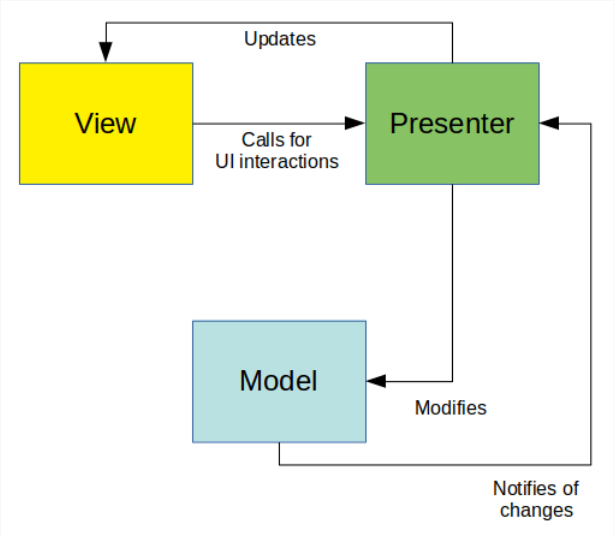


Figure 4: Model-View-Presenter

We have a UI, we have data, and we have glue code — in the form of a "presenter" — that ties the two together. The presenter is responsible for:

- Sinking events raised by the UI, such as form submissions, and using that information to update the stored representation of that data (the model), applying relevant business rules along the way
- Taking updates from the model — originating from this presenter, another presenter, asynchronous changes from a remote server, etc. — and updating its associated visual representation (the view)

It's nice and straightforward. On Android, things get a bit interesting with configuration changes, so while early Android apps might consider the activity or fragment to be the presenter, nowadays they are considered to be part of the view (along with the widgets). The presenter is a separate object, one that can be (carefully) reused across configuration changes, as a stable platform as our activities and fragments come and go.

Ironically, what today is called Model-View-Presenter originated as Model-View-Controller. Long-time MVC fans — those who worked on Smalltalk development in 1993, for example — need to deal with this name change. Nowadays, MVC is a slightly different approach than MVP:

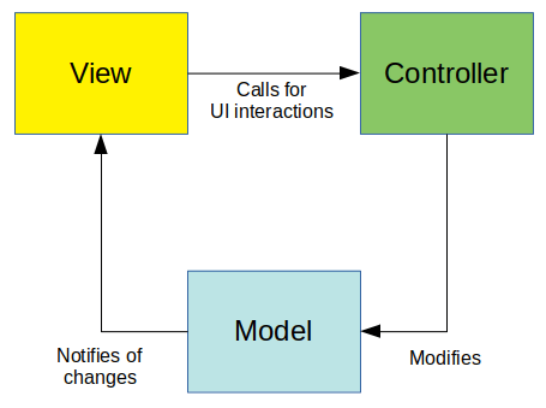


Figure 5: Model-View-Controller

The principal difference is that now data changes from the model flow directly to the view, bypassing any intermediary. In Android terms, if the model is publishing RxJava streams, for example, the view is the subscriber of those streams, whereas in MVP, the presenter would be the subscriber.

The benefit of this approach is the unidirectional data flow. A presenter serves as a common junction for data flows, which is OK in simple scenarios, but can get messy in larger use cases. In particular, the presenter needs to keep track not only of the data, but *who knows* about that particular piece of data:

- Is this some interim POJO, updated from the view, that needs to get reflected in the model?
- Is this some interim POJO, updated from the model, that needs to get reflected in the view?

If you use the same POJO class for each — and particularly if you are applying caching to hold onto that data in the presenter — there is a risk of bugs causing the data to stop flowing. For example, you get a revised POJO from the model, you update the presenter's cache... and fail to do anything to update the view. Or, you get data from the view, prepare changes for the model, then the model gives you new data from some other source, and you need to somehow merge this stuff together.

A unidirectional data flow, particularly with immutable objects, makes this simpler.

Now, each component is responsible for doing some work and passing the results along to a single destination, rather than having to remember which destination needs the results of the work.

Microsoft led the charge to convert Model-View-Presenter into Model-View-ViewModel (MVVM). From a diagram standpoint, MVVM looks strangely familiar:

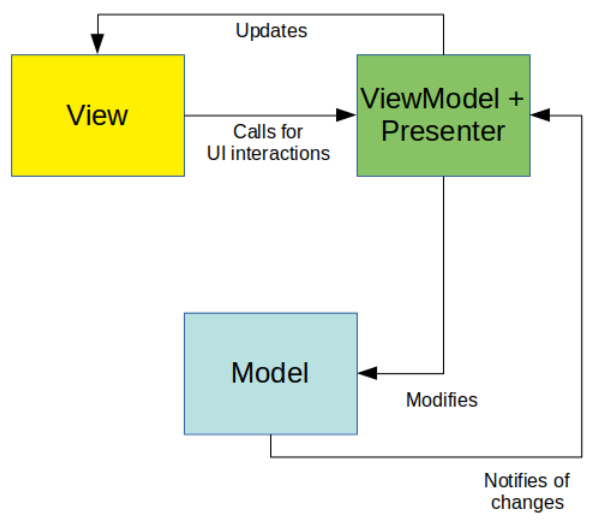


Figure 6: Model-View-ViewModel

In fact, typically, a MVVM app still has something called a presenter, which is responsible for preparing the view-model and connecting it with the view. MVVM implementations tend to emphasize two-way data binding, so the "business logic" for updating the view-model lies in a declarative UI (e.g., Android's layout resources, Microsoft's XAML). For major operations, such as form submission, the presenter takes the view-model and uses that information to update the "real" model.

This helps to clean up the communications somewhat. The presenter is mapping from the model to the view-model, applying appropriate changes along the way, such as:

- Data minimization: only exposing data in the view-model that the view needs
- Data transformation: taking the data in the form that the model stores it and changing it to something more amenable to the UI, such as converting currencies, converting time formats (e.g., Unix epoch times to richer date objects), and so on

The Basics of Model-View-Intent

While you have been working in Android, Web development has continued its own innovations. Redux has popularized a new approach to Web app development, and Redux in turn has led to interest in Model-View-Intent (MVI) as a GUI architecture.

However, as with the rest of the major GUI architectures, MVI is defined fairly loosely. However, at its core, it is a return to the unidirectional data flow that MVC offered... though with a few more parts:

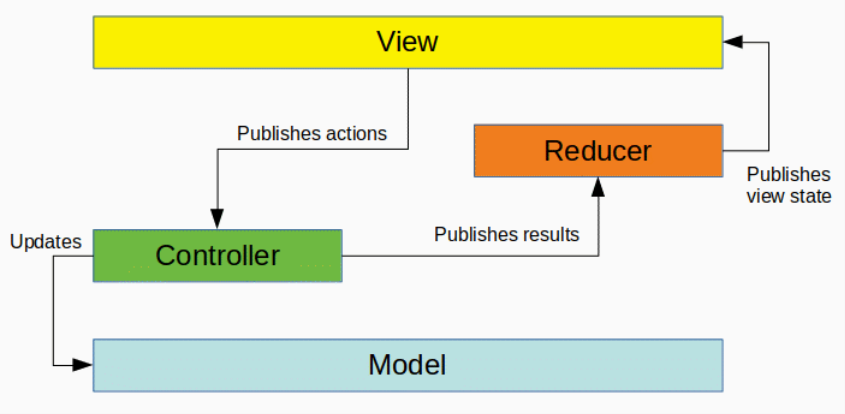


Figure 7: Model-View-Intent

Things start off a bit like MVC, where the view lets the controller know about actions that the user has taken, such as submitting a form or requesting a search. And, as with MVC, the controller is responsible for updating the model.

However, at that point, things start to differ, introducing a couple of new concepts: the view state and the reducer.

What's a View State?

The view state is somewhat reminiscent of the view-model in MVVM. It is the data necessary to render the view, by populating widgets and so forth.

The key behind many MVI implementations is that the view state is immutable. The view is handed a view state and needs to update the UI to reflect that new state. In many cases, that is simply filling in all of the widgets. In a few cases, that might get more elaborate, such as using `DiffUtil` to update the contents of a `RecyclerView`.

The Redux folks would phrase this something like "the view is a function applied to the state". The view layer does not care why the view state changed, just that it changed, and so it just updates the UI to match that state.

What's a Reducer?

The view state may be very complex, as it needs to be as complex as the UI that is being rendered. A single activity or fragment that has multiple tabs in a `ViewPager` might have several lists of material to go into those tabs, plus perhaps some additional data, all as part of the view state.

However, any individual action by the user is likely to only change a little bit of that view state. The user might mark some list item as a favorite, or add a new item, or swipe away an existing item. Most of the view state is stable.

In MVI, the controller is not directly responsible for maintaining that view state. It simply consumes actions from the view, updates the model, and publishes some sort of result to indicate that the work has been completed. Results do not have to map 1:1 to actions, though in many cases they will.

The "reducer" — whose name stems from the MapReduce model, presumably — is responsible for taking the result and crafting a new view state that reflects that incremental change in the data needed by the view. So, for example, if the user marks an item in the list as a favorite:

- The view emits an "mark-as-favorite" action
- The controller tells the model to persist that change and emits a "marked-as-favorite" result
- The reducer uses the result to update the in-memory representation of the list data to show that the favorite has been marked

The view does not update *itself* based upon the user's request. Instead, it emits the action, then renders the updated view state once it arrives.

Where Does the "Intent" Thing Show Up?

In the diagram shown above, there is a split between a result and a view state. In some MVI implementations, there is also a split between an "intent" and an "action". The intent is what the view publishes, but what the controller consumes is an action. There is a separate component that converts intents into actions.

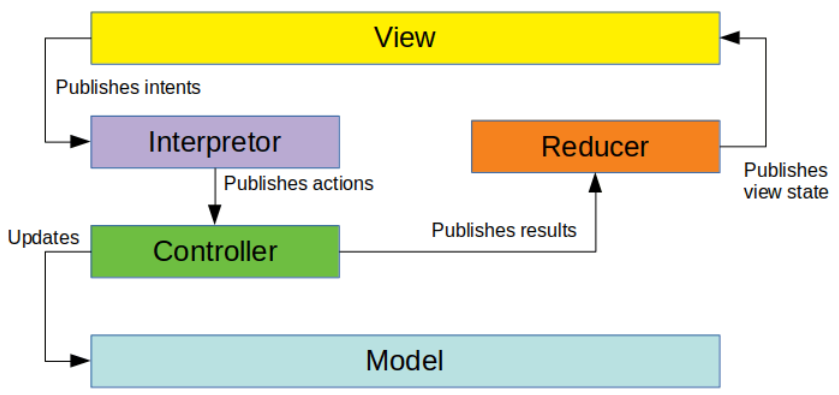


Figure 8: Model-View-Intent, With Actual Intents

Part of the rationale here is having a strictly layered separation of concerns:

- Intents and view state on the view side of the interpretor and reducer
- Actions and results on the controller side of the interpretor and reducer

For some UIs, it may be that the distinction between intents and actions may be fruitful over time. For example, perhaps you have a UI with a search option. On mobile devices, search is triggered by the typical sort of magnifying-glass icon in a toolbar. When the search is submitted, the UI wants to get a view state with search results. However, on Chromebooks or other devices with physical keyboards, you want to offer a direct typing approach, where the user can just start typing a search expression, and that automatically displays the `SearchView`, skipping the toolbar icon. The result is the same: conduct a search. But, perhaps you want to distinguish those as separate intents, with an eye towards perhaps offering different behaviors for searches triggered by each mechanism. However, the controller does not care whether the search was triggered by a toolbar icon click, just typing on the keyboard, selecting some saved search in a list or whatever. The controller just needs to know that a search is required. In this case, the view can publish different intents based upon how the search was requested, but the interpretor can normalize those into a smaller set of actions.

In practice, though, this approach can wind up with a lot of code duplication, as you mind-numbingly convert intents into actions on a 1:1 basis. For the sample MVI app profiled in the next chapter, the author originally wrote the app with intent/action separation... then got rid of the intents. If you feel that the intent/action separation is worthwhile, certainly use it. In the author's opinion, for many apps, the YAGNI principle applies: you aren't going to need it.

Additional MVI Resources

Here are some additional resources on MVI in Android that you may find useful:

- Jake Wharton's 2017 Devoxx US presentation – while Jake does not mention MVI, the architecture that he demonstrates is the Redux/MVI approach
- Yousuf Haque's droidcon NYC 2017 presentation on MVI
- Benoît Quenaudon's droidcon NYC 2017 presentation on MVI, and his associated sample app and blog post

A Deep Dive Into MVI

The preceding chapter introduced Model-View-Intent (MVI) as a GUI architecture pattern... without any code.

This chapter will look at a concrete implementation of MVI, so you can see how it works. Note, though, that all of these GUI architectures are fairly malleable, and so this chapter's approach may differ somewhat from other MVI implementations.

What the Sample App Does

A popular app category is a to-do list. These track outstanding tasks that need to be done, usually with some sort of checkbox or other indicator to denote which ones have been completed. Some offer due dates, recurring tasks, or other features to make it easier for you to set up a roster of tasks that match your needs. Some synchronize with a Web service, so that you can view your to-do list in multiple places, such as both on your phone and from a desktop Web browser.

Google has published a long list of sample apps that use a to-do list as a way of exploring various GUI architectures. The `ToDo/MVI` sample project is not a fork of those, but rather a "cleanroom" implementation of a to-do list with similar functionality. That functionality is tied into three fragments: the to-do list roster, the viewer, and the editor.

The Roster

When initially launched, the app will show a roster of the recorded to-do items, if there are any. Hence, on the first run, it will show just an "empty view", prompting the user to click the "add" action bar item to add a new item:

Figure 9: ToDo MVI App, As Initially Launched, with No Items

Once there are some items in the database, the roster will show those items, in alphabetical order by title, with a checkbox indicating whether or not they have been completed:

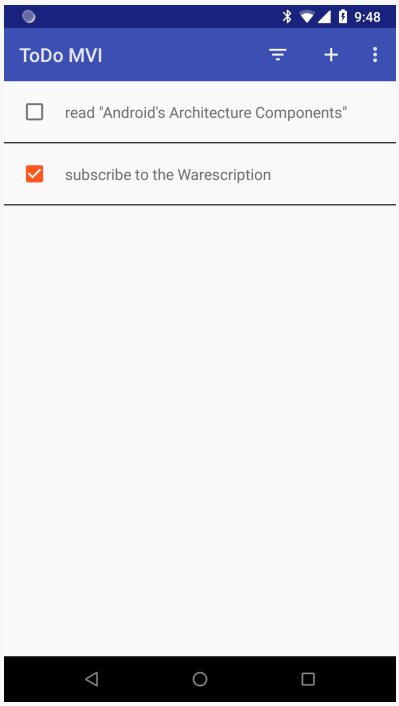


Figure 10: ToDo MVI App, Showing Two Items

From here, the user can tap the checkbox to quickly mark an item as completed (or un-mark it if needed). A filter drop-down allows the user to toggle the list from showing all items, only those marked as completed, or only those still outstanding (i.e., not yet checked as completed):

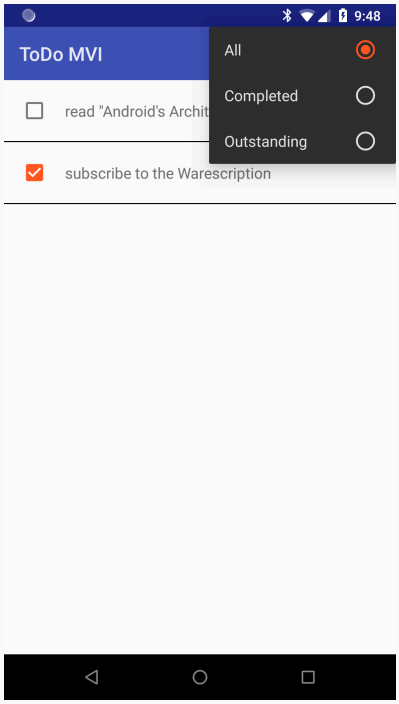


Figure 11: ToDo MVI App, Showing Filter Options

Long-pressing on an item switches the list into multiple-selection mode, where the user can then tap on items to build up a selection:

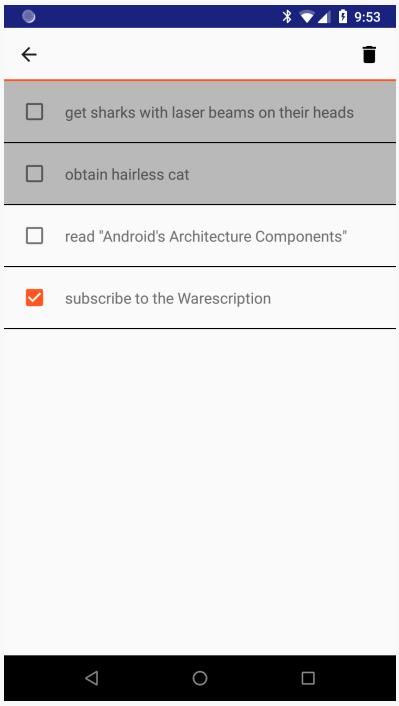


Figure 12: ToDo MVI App, Showing Multi-Select Mode with Two Selected Items

The “trash can” toolbar button will allow the user to delete the selected items, after confirmation:

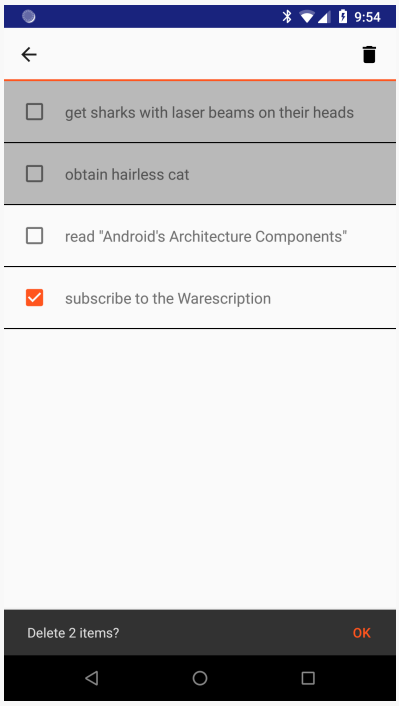


Figure 13: ToDo MVI App, Showing Delete Confirmation Snackbar

On a smaller-screen device, such as a phone, the roster will fill the screen. However, on larger-screen devices, the activity adopts the master-detail pattern and shows the viewer or editor fragment side-by-side with the roster:

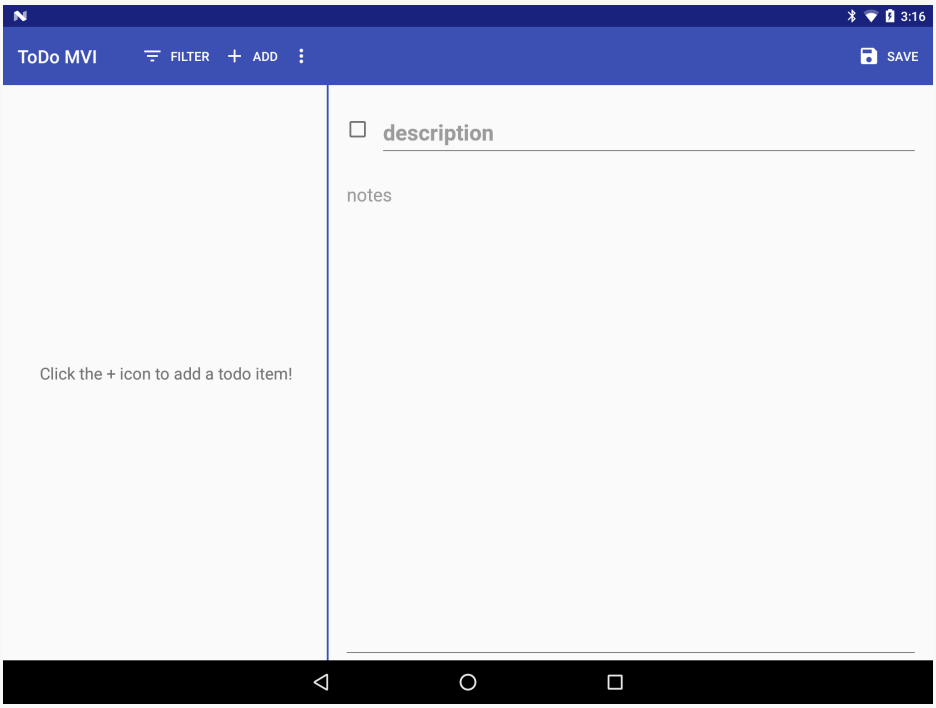


Figure 14: ToDo MVI App, Showing Master-Detail UI

The Viewer

A simple tap on an item in the roster brings up the viewer fragment, either alongside the roster on a larger screen or on its own on a smaller screen:

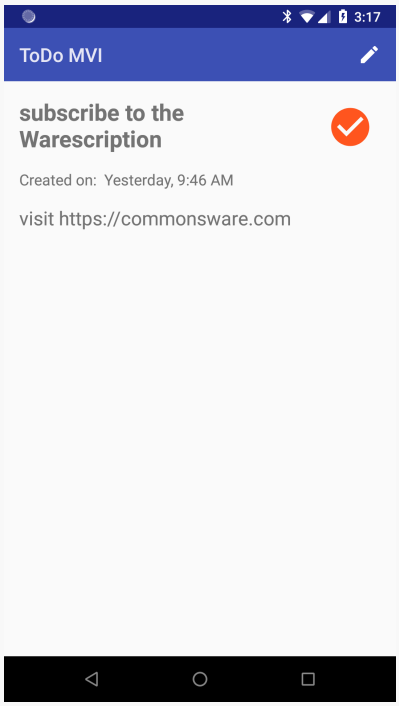


Figure 15: ToDo MVI App, Showing a Completed Item

This just shows additional information about the item, including any notes the user entered to provide more detail than the simple description that gets shown in the roster. The checkmark icon will appear for completed items.

From here, the user can edit this item (via the "pencil" icon). The user can also swipe left and right to traverse the roster of items — this is particularly useful on a phone, to avoid the "ping-pong" effect of having to navigate back to the roster fragment just to view details of the next item.

The Editor

The editor is a simple form, either to define a new to-do item or edit an existing one. If the user taps on the "add" action bar item from the roster, the editor will appear blank, and submitting the form will create a new to-do item. If the user taps on the "edit" (pencil) action bar item from the viewer, the editor will have the existing

item's data, which can be altered and saved:

Figure 16: ToDo MVI App, Editing a Completed Item

Clicking the "save" toolbar button will either add the new item or edit the item that the user requested to edit. For an edit, the "delete" toolbar button will be available and will allow the user to delete this specific item, after confirmation.

MVI and the Sample App

The sample app uses the simplified MVI approach outlined in the preceding chapter, skipping the intent/action separation and just having the view emit actions:

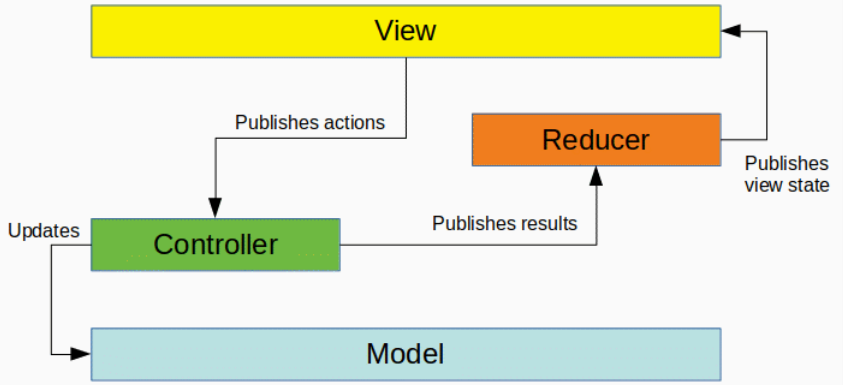


Figure 17: Model-View-Intent, As Used In This App

The view consists of our three fragments, as they each operate off of the same state: the roster of to-do items.

The model is made up of two parts:

- The to-do items themselves, stored in a Room-managed database, fronted by a repository, and converted into model objects used as part of our view state
- The filter mode, which we want to persist across runs of the app, stored in a `SharedPreferences` fronted by another repository

We also have actions, a controller, results, and a reducer as well, to mirror the MVI structure. Though, as you will see, the reducer is named something other than `Reducer`.

The Model

We need some in-memory representation of a to-do item. That comes in the form of a `ToDoModel` POJO class.

In addition to a unique ID, there are four pieces of additional data that we track about each to-do item in `ToDoModel`:

- The description, which is the text that the user sees in the roster
- Some notes, in case there are additional instructions that the user wants to track for this to-do item
- Whether or not this item is completed, in the form of a simple `boolean`
- The date and time on which this to-do item was created

We could track more information — last-updated timestamp, revision history, categories, etc. — but we are trying to keep this relatively simple.

`ToDoModel`, like the actions, uses AutoValue for immutability. So, our class has the `@AutoValue` annotation and abstract methods for each of the properties that we are tracking:

```
@AutoValue
public abstract class ToDoModel {
  public abstract String id();
  public abstract boolean isCompleted();
  public abstract String description();
  @Nullable public abstract String notes();
  public abstract Calendar createdOn();
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ToDoModel.java)

The `notes()` method is annotated with `@Nullable`, so AutoValue will allow `null` for that property. All other properties are required.

We are using the builder pattern, so we have an `@AutoValue.Builder`-annotated `static` class named `Builder`, with corresponding setter methods for the properties, plus `build()` to create a `ToDoModel` instance from the `Builder`:

```
  @AutoValue.Builder
  public abstract static class Builder {
    abstract Builder id(String id);
    public abstract Builder isCompleted(boolean isCompleted);
    public abstract Builder description(String desc);
    public abstract Builder notes(String notes);
    abstract Builder createdOn(Calendar date);
    public abstract ToDoModel build();
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ToDoModel.java)

We also have three methods for getting `Builder` instances:

- `builder()` is a `static` method that just returns an instance of the

AutoValue-generated `Builder` implementation

- `creator()` is a `static` method that takes the `builder()` and fills in some default information (a generated ID, the current timestamp, and `false` for `isCompleted()`)
- `toBuilder()` is an instance method that takes a `builder()` and fills in all of the current data

```
static Builder builder() {
  return(new AutoValue_ToDoModel.Builder());
}

public static Builder creator() {
  return(builder()
    .isCompleted(false)
    .id(UUID.randomUUID().toString())
    .createdOn(Calendar.getInstance()));
}

public Builder toBuilder() {
  return(builder()
    .id(id())
    .isCompleted(isCompleted())
    .description(description())
    .notes(notes())
    .createdOn(createdOn()));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ToDoModel.java)

So, creating a new `ToDoModel` instance is a matter of calling `creator()`, then whatever setters are necessary, then `build()` to create the instance. To modify an existing `ToDoModel`, call `toBuilder()` on it, then whatever setters are necessary, then `build()` to create the modified instance.

While the sample app uses Room to save the to-do items, `ToDoModel` is not a Room `@Entity`. There is a separate `ToDoEntity` class that models our database table, and the app maps between `ToDoModel` and `ToDoEntity` instances as needed.

The View State

Our view layer gets its `ToDoModel` instances from the `ViewState`. This class aggregates all of the data necessary to render our three fragments. It includes:

- The list of models to display (as this sample app makes the tremendously

simplifying assumption that the entire set of to-do items can be held in memory)
- A `boolean` indicating whether the data has been loaded or not, so we can distinguish whether an empty list of models means "we have no to-do items" or "we have not yet loaded the to-do items"
- A list of indices into the model data representing items that are selected, when the list fragment is in multi-select mode
- The current filter mode, indicating what subset of the list of models should be rendered
- The "current" model, for situations where our display or edit fragments are visible, to reflect the model that they are showing
- A `Throwable`, in case there was some exception coming from the repository that we need to show to the user

As with `ToDoModel`, `ViewState` uses AutoValue to create a more-or-less immutable object. So, we have `abstract` methods for each of those properties that we want to track:

```
@AutoValue
public abstract class ViewState {
  public abstract boolean isLoaded();
  public abstract List<ToDoModel> items();
  abstract Set<Integer> selections();
  @Nullable public abstract Throwable cause();
  public abstract FilterMode filterMode();
  @Nullable public abstract ToDoModel current();
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ViewState.java)

We also have an associated `Builder`:

```
  @AutoValue.Builder
  abstract static class Builder {
    abstract Builder isLoaded(boolean isLoaded);
    abstract Builder items(List<ToDoModel> items);
    abstract Builder selections(Set<Integer> positions);
    abstract Builder cause(Throwable cause);
    abstract Builder filterMode(FilterMode mode);
    abstract Builder current(ToDoModel current);
    abstract ViewState build();
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ViewState.java)

We have a `builder()` method that returns a `Builder` with a likely set of default

values (e.g., no current selections):

```
static Builder builder() {
  return(new AutoValue_ViewState.Builder()
    .isLoaded(false)
    .selections(new HashSet<>())
    .filterMode(FilterMode.ALL));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ViewState.java)

Beyond that, we have a series of helper methods for common scenarios, such as `empty()` for returning a `Builder` set up with no items:

```
static Builder empty() {
  return(builder().items(new ArrayList<>()));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ViewState.java)

...and a `toBuilder()` method, which creates a `Builder` pre-populated with the current values from a `ViewState`, to be able to revise those values and create a fresh `ViewState` from the `Builder`:

```
private Builder toBuilder() {
  return(builder()
    .isLoaded(isLoaded())
    .cause(cause())
    .items(items())
    .selections(selections())
    .current(current())
    .filterMode(filterMode()));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ViewState.java)

The View

The `MainActivity` and its fragments implement the "view" layer of the MVI architecture. These classes have two primary jobs:

1. Render the view state when it arrives
2. Create actions and get them over to the controller

Of course, this is Android, and so there are other details to be worried about.

Configuration changes are chief among those details.

Right now, we will focus on `RosterListFragment`, the fragment for displaying the list of to-do items. Later on, we will look briefly at the other two fragments. Note that `RosterListFragment` inherits from an `AbstractRosterFragment`, as some of its logic is shared with `DisplayFragment`.

Incorporating a ViewModel

The fragments use a `ViewModel`, named `RosterViewModel`, as the state to be retained across configuration changes. To that end, `MainActivity` is a `FragmentActivity` and the three fragments each extend from the support libraries' edition of `Fragment`. The fragments hold onto the `RosterViewModel` in a `viewModel` field and initialize it in `onCreate()`:

```
@Override
public void onCreate(@Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  viewModel=ViewModelProviders.of(getActivity()).get(RosterViewModel.class);
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/AbstractRosterFragment.java)

Notice that we are passing the activity into `of()`, not the fragment. As a result, all three fragments share a common `RosterViewModel`. For relatively tightly-coupled fragments, this likely will be a common pattern.

`RosterViewModel` has two key roles:

1. It holds onto the `Controller` to be used by the view and forwards actions from the view to that `Controller`
2. It serves as the reducer, receiving results from the `Controller` and converting them into updated view states, delivering those to the view as results arrive

Receiving View States

We need to get `ViewState` instances from the `RosterViewModel` to the fragments, when results arrive from the `Controller`.

To that end, `RosterViewModel` has a `LiveData` object, representing a stream of view

states. That is made available to the view layer via a simple `stateStream()` method:

```
public LiveData<ViewState> stateStream() {
  return(states);
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/RosterViewModel.java)

Our fragments then use that `LiveData` to subscribe to the stream and route the `ViewState` objects to a `render()` method:

```
    viewModel.stateStream().observe(this, this::render);
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/AbstractRosterFragment.java)

Rendering View States

Each of our three fragments has its own `render()` method, responsible for taking the data in our `ViewState` and updating its own bit of the UI to match.

In the case of the `RosterListFragment`, `render()` is responsible for populating the `RecyclerView`:

```
@Override
void render(ViewState state) {
  if (adapter!=null) {
    if (state.cause()==null) {
      adapter.setState(state);

      if (state.isLoaded() && state.filteredItems().size()==0) {
        getEmptyView().setVisibility(View.VISIBLE);

        if (state.items().size()>0) {
          getEmptyView().setText(R.string.msg_empty_filter);
        }
        else {
          getEmptyView().setText(R.string.msg_empty);
        }
      }
      else {
        getEmptyView().setVisibility(View.GONE);
      }

      if (state.getSelectionCount()==0 && snackbar!=null &&
        snackbar.isShown()) {
        snackbar.dismiss();
```

```
      }
    }
    else {
      Snackbar
        .make(getView(), R.string.msg_crash, Snackbar.LENGTH_LONG)
        .show();
      Log.e(getClass().getSimpleName(), "Exception in obtaining view state",
        state.cause());
    }
  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/RosterListFragment.java)

If we have no adapter, then our UI has not been set up just yet, and so we need to skip this rendering event.

If the `ViewState` contains a `cause()`, we show a `Snackbar` to alert the user to the problem, plus log the `Throwable` to Logcat for debugging purposes.

In the more normal case, where everything worked and we have our UI, we:

- Pass the `ViewState` along to the `RosterListAdapter`, to update the contents of the `RecyclerView`
- Hide, show, and update the prose for the empty view, as appropriate
- If we happen to have some other `Snackbar` showing (e.g., from a delete request), dismiss it

The empty view is a bit complicated, because we have three conditions:

1. There are items in the list, in which case we do not want to show the empty view, so we mark it `GONE`
2. There are no items in the list, because there are simply no items at all
3. There are items in the list, but the current filter mode that the user has chosen blocks all of them (e.g., the user chose outstanding items and all items are completed)

The `ViewState` has a helper method, `filteredItems()`, which returns only the subset of the `items()` list that apply for the currently-chosen filter:

```
@Memoized
public List<ToDoModel> filteredItems() {
  return(ToDoModel.filter(items(), filterMode()));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ViewState.java)

Here, `@Memoized` means that the `ViewState` will cache the results of computing this list, to save time on subsequent calls — this is a feature of AutoValue.

`ToDoModel.filter()`, in turn, does the actual filtering:

```
public static List<ToDoModel> filter(List<ToDoModel> models,
                                      FilterMode filterMode) {
  List<ToDoModel> result;

  if (filterMode==FilterMode.COMPLETED) {
    result=new ArrayList<>();

    for (ToDoModel model : models) {
      if (model.isCompleted()) {
        result.add(model);
      }
    }
  }
  else if (filterMode==FilterMode.OUTSTANDING) {
    result=new ArrayList<>();

    for (ToDoModel model : models) {
      if (!model.isCompleted()) {
        result.add(model);
      }
    }
  }
  else {
    result=new ArrayList<>(models);
  }

  return(result);
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ToDoModel.java)

The implementation of `setState()` on `RosterListAdapter` is responsible for updating the `RecyclerView` contents. That is fairly complicated and not particularly relevant for the discussion of MVI, so we will skip that here, other than to note that it uses `DiffUtil` to animate any relevant changes to the visible rows in the list, comparing the new view state with its predecessor.

The Actions

There are several possible actions that our view layer will be able to publish:

- The user might add a new to-do item
- The user might edit an existing to-do item, replacing some of its data with new values
- The user might delete one or more to-do items
- In the dual-pane master-detail mode, the user might select or unselect items – while this is not part of a persistent data model, it is information that we need to retain across configuration changes and therefore forms part of our view state
- The user might change the filter mode for controlling which set of to-do items appears in the roster

In addition, we need a "load" action to load our content when the UI first appears.

Some of these actions have associated data. For example, to add a new to-do item, we need some sort of model object describing the item. Some actions need no additional data, such as the "load" or "unselect-all" actions.

To that end, we have an `Action` class. This class is `abstract`, with concrete subclasses for each specific action. This way, some of our code can just deal with actions in general via the `Action` base type, while the rest of our code can work with the specific action types where needed. Each concrete class can hold whatever data that action needs.

The `Action` class:

- Defines those concrete subclasses, either directly (if the action has no associated data) or via AutoValue (if the action has data, for immutable action types)
- Defines helper static methods to create instances of the appropriate `Action` concrete type

```
package com.commonsware.android.todo.impl;

import com.google.auto.value.AutoValue;
import java.util.Collections;
import java.util.List;
```

```
public abstract class Action {
  public static Action add(ToDoModel model) {
    return(new AutoValue_Action_Add(model));
  }

  public static Action edit(ToDoModel model) {
    return(new AutoValue_Action_Edit(model));
  }

  public static Action delete(List<ToDoModel> models) {
    return(new AutoValue_Action_Delete(Collections.unmodifiableList(models)));
  }

  public static Action delete(ToDoModel model) {
    return(delete(Collections.singletonList(model)));
  }

  public static Action select(int position) {
    return(new AutoValue_Action_Select(position));
  }

  public static Action unselect(int position) {
    return(new AutoValue_Action_Unselect(position));
  }

  public static Action unselectAll() {
    return(new UnselectAll());
  }

  public static Action show(ToDoModel model) {
    return(new AutoValue_Action_Show(model));
  }

  public static Action filter(FilterMode mode) {
    return(new AutoValue_Action_Filter(mode));
  }

  public static Action load() {
    return(new Action.Load());
  }

  @AutoValue
  public static abstract class Add extends Action {
    public abstract ToDoModel model();
  }

  @AutoValue
  public static abstract class Edit extends Action {
```

```
    public abstract ToDoModel model();
  }

  @AutoValue
  public static abstract class Delete extends Action {
    public abstract List<ToDoModel> models();
  }

  @AutoValue
  static abstract class Select extends Action {
    public abstract int position();
  }

  @AutoValue
  static abstract class Unselect extends Action {
    public abstract int position();
  }

  static class UnselectAll extends Action {

  }

  @AutoValue
  static abstract class Show extends Action {
    public abstract ToDoModel current();
  }

  @AutoValue
  static abstract class Filter extends Action {
    public abstract FilterMode filterMode();
  }

  public static class Load extends Action {

  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Action.java)

This looks complicated, but it is just a number of occurrences of the same basic pattern. For example, for the "add" action we have an `Add` subclass of `Action`, set up via AutoValue:

```
  @AutoValue
  public static abstract class Add extends Action {
    public abstract ToDoModel model();
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Action.java)

We also have a `static` method named `add()` on `Action` to create an instance of an `Action.Add`:

```
public static Action add(ToDoModel model) {
  return(new AutoValue_Action_Add(model));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Action.java)

Publishing Actions

Our `RosterViewModel` has a `process()` method that our fragments call to publish actions. In the case of `RosterListFragment` and `RosterListAdapter`, this happens in a few places.

First, the user might modify a to-do item directly from the `RecyclerView` by clicking the item's `CheckBox`. This sample app uses the data binding framework, and so the `CheckBox` has a binding expression for the `onCheckedChanged` event, routing it to the view (by way of `RosterListAdapter` and its associated `RosterRowHolder` for each row). That, in turn, eventually triggers a `process()` call to publish an edit event:

```
public void edit(ToDoModel model, boolean isChecked) {
  process(Action.edit(model.toBuilder().isCompleted(isChecked).build()));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/RosterListFragment.java)

Here, we:

- Get a `Builder` with the current item's data from `toBuilder()`
- Call `isCompleted()` to mark the item as completed
- `build()` a new `ToDoModel` from that `Builder`
- Pass that `ToDoModel` to the `edit()` helper method on `Action` to create the proper `Action` instance
- Hand that `Action` over to `process()`

Similarly, if the user clicks the "delete" icon in multi-select mode, we want to get confirmation from the user, then delete those items. That eventually gets handled by a `requestDelete()` method on `RosterListFragment`, which shows a `Snackbar` and calls `process()` for `delete()` `Action` if the user clicks the `Snackbar` action:

```
public void requestDelete(int selectionCount) {
  Resources res=getResources();
  String msg=res.getQuantityString(R.plurals.snackbar_delete,
    selectionCount, selectionCount);

  snackbar=Snackbar.make(getView(), msg, Snackbar.LENGTH_LONG);

  snackbar
    .setAction(android.R.string.ok, view -> {
      process(Action.delete(adapter.getSelectedModels()));
      adapter.exitMultiSelectMode();
    })
    .show();
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/RosterListFragment.java)

Our filter mode is part of our view state, and the filter mode is something that we want to persist. That is why we have actions related to filter mode, and why when the user toggles those menu items we call `process()` to publish the associated actions:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.add:
      ((Contract)getActivity()).addModel();
      return(true);

    case R.id.all:
      filterAll.setChecked(true);
      process(Action.filter(FilterMode.ALL));
      return(true);

    case R.id.completed:
      filterCompleted.setChecked(true);
      process(Action.filter(FilterMode.COMPLETED));
      return(true);

    case R.id.outstanding:
      filterOutstanding.setChecked(true);
      process(Action.filter(FilterMode.OUTSTANDING));
      return(true);

    case R.id.export:
      export();
      return(true);
```

```
    case R.id.share:
      share();
      return(true);

    case R.id.backup:
      backup();
      return(true);

    case R.id.restore:
      restore();
      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/RosterListFragment.java)

Note that in none of these cases do we update the fragment's UI based on these events. Of course, in the case of the `CheckBox`, that happens automatically. But if the user deletes items or changes the filter mode, we do not update the `RecyclerView`. Instead, we just publish the actions, and we only update the UI in `render()`, when we get the updated view state.

One oddball action is the "load" action. Somewhere along the line, when we start up the app, we need to load our data — without the data, we have no view state and we have nothing to render. But, at the same time, we *only* want to load the data when we are starting out, not after a configuration change, so we cannot readily use lifecycle methods on the activity or fragments to trigger a load. Instead, `RosterViewModel`, as part of its constructor, publishes a `load() Action` on its own. Hence, when we create the view-model, we initiate loading of the data. Since a `ViewModel` is only created once per "logical" activity instance (taking into account configuration changes), we only load this data once for the entire activity. This might be insufficient in a more elaborate app, where multiple activities might share a common repository, but it will suffice here.

The Repositories

Eventually, our `Controller` will need to update backing stores, so that our to-do items and filter modes are persistent from run to run of our app. This sample app uses the repository pattern, with two repositories: `ToDoRepository` for to-do items and `FilterModeRepository` for filter modes.

The objective of each repository is simple:

- Modify the backing store as needed, where the caller is required to establish the appropriate thread to do this on
- Offer an observable API to load the data

Each repository is a singleton, so all logic routes through central points for each type of data.

ToDoRepository

The `ToDoRepository` wraps around a `ToDoDatabase` and `ToDoEntity` objects, where those have the appropriate Room annotations to set up a SQLite database.

`ToDoEntity` has fields that map to the same pieces of data that `ToDoModel` holds. In principle, we could have dispensed with the separation and passed around `ToDoEntity` objects where we are currently passing around `ToDoModel` objects. However, it is possible that some future edition of this sample might have multiple backing stores (e.g., local database and a server), in which case keeping some separation between the in-memory model and the persistent representations is worthwhile. In this case, converting between the two is relatively simple, and `ToDoEntity` has a constructor and `toModel()` methods that do just that:

```
package com.commonsware.android.todo.impl;

import android.arch.persistence.room.Dao;
import android.arch.persistence.room.Delete;
import android.arch.persistence.room.Entity;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.Insert;
import android.arch.persistence.room.PrimaryKey;
import android.arch.persistence.room.Query;
import android.arch.persistence.room.Update;
import android.support.annotation.NonNull;
import java.util.Calendar;
import java.util.List;
import io.reactivex.Single;

@Entity(tableName="todos", indices=@Index(value="id"))
public class ToDoEntity {
  @PrimaryKey
  @NonNull final String id;
  @NonNull final String description;
  final String notes;
  final boolean isCompleted;
  @NonNull final Calendar createdOn;

  public static ToDoEntity fromModel(ToDoModel model) {
    return(new ToDoEntity(model.id(), model.description(), model.isCompleted(),
```

```
      model.notes(), model.createdOn()));
  }

  ToDoEntity(@NonNull String id, @NonNull String description, boolean isCompleted,
             String notes, @NonNull Calendar createdOn) {
    this.id=id;
    this.description=description;
    this.isCompleted=isCompleted;
    this.notes=notes;
    this.createdOn=createdOn;
  }

  public ToDoModel toModel() {
    return(ToDoModel.builder()
      .id(id)
      .description(description)
      .isCompleted(isCompleted)
      .notes(notes)
      .createdOn(createdOn)
      .build());
  }

  @Dao
  public interface Store {
    @Query("SELECT * FROM todos ORDER BY description ASC")
    Single<List<ToDoEntity>> all();

    @Insert
    void insert(ToDoEntity... entities);

    @Update
    void update(ToDoEntity... entities);

    @Delete
    void delete(ToDoEntity... entities);

    @Delete
    void delete(List<ToDoEntity> entities);

    @Query("DELETE FROM todos")
    void deleteAll();
  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ToDoEntity.java)

`ToDoRepository` then maps between the models that it uses as its API and the entities that it uses with the `ToDoDatabase`:

```
package com.commonsware.android.todo.impl;

import android.content.Context;
import java.util.ArrayList;
import java.util.List;
import io.reactivex.Single;
import io.reactivex.annotations.NonNull;
```

```
import io.reactivex.functions.Function;

public class ToDoRepository {
  private static volatile ToDoRepository INSTANCE=null;
  private final ToDoDatabase db;

  public synchronized static ToDoRepository get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=new ToDoRepository(ctxt.getApplicationContext());
    }

    return(INSTANCE);
  }

  private ToDoRepository(Context ctxt) {
    db=ToDoDatabase.get(ctxt);
  }

  public Single<List<ToDoModel>> all() {
    return(db.todoStore().all().map(entities -> {
      ArrayList<ToDoModel> result=new ArrayList<>(entities.size());

      for (ToDoEntity entity : entities) {
        result.add(entity.toModel());
      }

      return(result);
    }));
  }

  public void add(ToDoModel model) {
    db.todoStore().insert(ToDoEntity.fromModel(model));
  }

  public void replace(ToDoModel model) {
    db.todoStore().update(ToDoEntity.fromModel(model));
  }

  public void delete(List<ToDoModel> models) {
    List<ToDoEntity> entities=new ArrayList<>();

    for (ToDoModel model : models) {
      entities.add(ToDoEntity.fromModel(model));
    }

    db.todoStore().delete(entities);
  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ToDoRepository.java)

Its `all()` method is a `Single`, used for the initial data load, which gets all the entities, then uses a `map()` operator to convert those entities into models.

FilterModeRepository

The filter mode is saved in `SharedPreferences`. The `FilterMode` itself is an `enum` that knows how to map between stable `int` values and the corresponding `enum` values:

```
package com.commonsware.android.todo.impl;

public enum FilterMode {
  ALL(0),
  COMPLETED(1),
  OUTSTANDING(2);

  private final int value;

  static FilterMode forValue(int value) {
    FilterMode result=ALL;

    if (value==1) {
      result=COMPLETED;
    }
    else if (value==2) {
      result=OUTSTANDING;
    }

    return(result);
  }

  FilterMode(int value) {
    this.value=value;
  }

  int getValue() {
    return(value);
  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/FilterMode.java)

`FilterModeRepository` hides the storage details, accepting in `FilterMode` objects and offering a `load()` method to obtain the current `FilterMode` via a `Single`:

```
package com.commonsware.android.todo.impl;

import android.annotation.SuppressLint;
import android.content.Context;
import android.content.SharedPreferences;
import io.reactivex.Single;

class FilterModeRepository {
  private static final String PREF_MODE="filterMode";
  private static volatile FilterModeRepository INSTANCE=null;
  private SharedPreferences prefs=null;

  synchronized static FilterModeRepository get() {
    if (INSTANCE==null) {
      INSTANCE=new FilterModeRepository();
    }

    return(INSTANCE);
  }

  Single<FilterMode> load(Context ctxt) {
    final Context app=ctxt.getApplicationContext();

    return(Single.create(e -> {
      synchronized(this) {
        if (prefs==null) {
          prefs=app.getSharedPreferences(getClass().getCanonicalName(),
            Context.MODE_PRIVATE);
        }
      }

      e.onSuccess(FilterMode.forValue(prefs.getInt(PREF_MODE, 0)));
    }));
  }

  @SuppressLint("ApplySharedPref")
  void save(FilterMode mode) {
    prefs.edit().putInt(PREF_MODE, mode.getValue()).commit();
  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/FilterModeRepository.java)

The Controller

Given the actions and the repositories, the controller is the glue code, updating the repositories based on the actions and emitting results to trigger updates to the view

state and, from there, the UI.

Subscribing to Actions

To get the actions over to the `Controller`, the `RosterViewModel` has an RxJava `PublishSubject` that serves as its `Observable` source of a stream of `Action` objects. Every time `process()` is called, the `RosterViewModel` emits that `Action` onto the `actionSubject`:

```
public void process(Action action) {
  actionSubject.onNext(action);
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/RosterViewModel.java)

As part of its constructor, the `RosterViewModel` creates a `Controller`. Once again, this may not be an appropriate approach for a more complex app, where there may be multiple view-models needing to work with a common `Controller`, but it suffices for here. Along the way, the `RosterViewModel` calls a `subscribeToActions()` method, so that the `Controller` can subscribe to those `Action` events:

```
public void subscribeToActions(Observable<Action> actionStream) {
  actionStream
    .observeOn(Schedulers.single())
    .subscribe(this::processImpl);
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Controller.java)

This particular subscription routes the work to a `single()` thread, to keep the repository work off of the main application thread. And, it passes the `Action` objects to a `processImpl()` method.

Doing the Work and Publishing Results

So far, the only place where we care about specific types of `Action` is when we publish them. In effect, we take several event types and combine them into a single type for convenience. However, at some point, we need to split them back out again, so we can handle specific logic for specific actions, and that occurs in `processImpl()`:

```
private void processImpl(Action action) {
  if (action instanceof Action.Add) {
```

```
      add(((Action.Add)action).model());
    }
    else if (action instanceof Action.Edit) {
      modify(((Action.Edit)action).model());
    }
    else if (action instanceof Action.Delete) {
      delete(((Action.Delete)action).models());
    }
    else if (action instanceof Action.Load) {
      load();
    }
    else if (action instanceof Action.Select) {
      select(((Action.Select)action).position());
    }
    else if (action instanceof Action.Unselect) {
      unselect(((Action.Unselect)action).position());
    }
    else if (action instanceof Action.UnselectAll) {
      unselectAll();
    }
    else if (action instanceof Action.Show) {
      show(((Action.Show)action).current());
    }
    else if (action instanceof Action.Filter) {
      filter(((Action.Filter)action).filterMode());
    }
    else {
      throw new IllegalStateException("Unexpected action: "+action.toString());
    }
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Controller.java)

This basically "unwraps" the action and invokes a dedicated method per action type. Most of those methods work with one of the repositories for the data associated with that action. All of these methods use a `BehaviorSubject` named `resultSubject` to publish the result, and we will examine that in detail a bit later.

The action method can be broken down into four groups, based on the data being operated on and the operation type.

add()/modify()/delete()

These three methods are fairly straightforward. They call the associated method on the `ToDoRepository` and publish their results:

```
private void add(ToDoModel model) {
  toDoRepo.add(model);
  resultSubject.onNext(Result.added(model));
}

private void modify(ToDoModel model) {
  toDoRepo.replace(model);
  resultSubject.onNext(Result.modified(model));
}

private void delete(List<ToDoModel> toDelete) {
  toDoRepo.delete(toDelete);
  resultSubject.onNext(Result.deleted(toDelete));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Controller.java)

filter()

`filter()` is similar, except that it works with the `FilterModeRepository`:

```
private void filter(FilterMode mode) {
  filterModeRepo.save(mode);
  resultSubject.onNext(Result.filter(mode));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Controller.java)

select()/unselect()/unselectAll()

The selections are not persistent — they are purely a UI contrivance. However, they are part of the view state, and the only way to update the view state is by going through the action-controller-reducer flow. So, these three methods just publish results to get their data updates over to the reducer:

```
private void select(int position) {
  resultSubject.onNext(Result.selected(position));
}

private void unselect(int position) {
  resultSubject.onNext(Result.unselected(position));
}

private void unselectAll() {
  resultSubject.onNext(Result.unselectedAll());
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Controller.java)

load()

`load()` is the quirky one, as this would not be traditional computer programming if *everything* were simple.

```
  private void load() {
    Single<Result> loader=
      Single.zip(toDoRepo.all(), filterModeRepo.load(ctxt),
        (models, mode) -> (Result.loaded(models, mode)));

    loader
      .subscribeOn(Schedulers.single())
      .subscribe(resultSubject::onNext);
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Controller.java)

The load action is to load our data. In our case, though, we have data from two repositories: the `ToDoRepository` and the `FilterModeRepository`. Each publishes a `Single` for loading their particular bits of data. We need to publish results once both of those `Single` objects have completed processing.

RxJava's `zip()` operator — which has nothing much to do with ZIP files, zip ties, or ziplines — is designed for this sort of scenario. You give `zip()` multiple observables, and it invokes your code for each set of results emitted by the source observables. Since our observables are `Single`, they only emit one object, and so we get control once both `Single` results are in. We then publish those as our own result.

About Those Results

Just as we have a common `Action` type that wraps up a bunch of disparate actions, we have a common `Result` type that wraps up a bunch of disparate results. The `Controller` uses the aforementioned `BehaviorSubject` to emit `Result` objects to interested parties.

`Result` is structured similarly to `Action`, using AutoValue for immutable objects, with `static` factory methods to create instances associated with each type:

```
package com.commonsware.android.todo.impl;

import com.google.auto.value.AutoValue;
import java.util.Collections;
```

```
import java.util.List;

public abstract class Result {
  public static Result added(ToDoModel model) {
    return(new AutoValue_Result_Added(model));
  }

  public static Result modified(ToDoModel model) {
    return(new AutoValue_Result_Modified(model));
  }

  static Result deleted(List<ToDoModel> models) {
    return(new AutoValue_Result_Deleted(Collections.unmodifiableList(models)));
  }

  static Result loaded(List<ToDoModel> models, FilterMode filterMode) {
    return(new AutoValue_Result_Loaded(Collections.unmodifiableList(models), filterMode));
  }

  static Result selected(int position) {
    return(new AutoValue_Result_Selected(position));
  }

  static Result unselected(int position) {
    return(new AutoValue_Result_Unselected(position));
  }

  static Result unselectedAll() {
    return(new AutoValue_Result_UnselectedAll());
  }

  static Result showed(ToDoModel current) {
    return(new AutoValue_Result_Showed(current));
  }

  static Result filter(FilterMode mode) {
    return(new AutoValue_Result_Filter(mode));
  }

  @AutoValue
  public static abstract class Added extends Result {
    public abstract ToDoModel model();
  }

  @AutoValue
  public static abstract class Modified extends Result {
    public abstract ToDoModel model();
  }

  @AutoValue
  public static abstract class Deleted extends Result {
    public abstract List<ToDoModel> models();
  }

  @AutoValue
  static abstract class Selected extends Result {
    public abstract int position();
  }

  @AutoValue
```

```
  static abstract class Unselected extends Result {
    public abstract int position();
  }

  @AutoValue
  static abstract class UnselectedAll extends Result {
  }

  @AutoValue
  static abstract class Showed extends Result {
    public abstract ToDoModel current();
  }

  @AutoValue
  static abstract class Filter extends Result {
    public abstract FilterMode filterMode();
  }

  @AutoValue
  public static abstract class Loaded extends Result {
    public abstract List<ToDoModel> models();
    public abstract FilterMode filterMode();
  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/Result.java)

The Reducer in the RosterViewModel

What remains is the reducer: accepting the results and updating the view state to match. In this sample app, that is part of the role of the `RosterViewModel`.

You might wonder why this is called "`RosterViewModel`", given that it has responsibilities that do not exactly line up with a classic view-model. The name comes from the base class: `AndroidViewModel`. We want to retain this object across configuration changes, and the way to do that with the Architecture Components is to use `ViewModel`, `AndroidViewModel`, `ViewModelProviders`, and so forth. In the author's opinion, Google would have been better served naming their system something that did not have "`ViewModel`" in it, just as Room does not have "`Model`" or "`Repository`" in it.

Subscribing to Results

Our `Controller` publishes `Result` objects via the `resultSubject` `Observable`, exposed via a `resultStream()` method. `RosterViewModel` needs to subscribe to that stream, take the results, fold them into the view state, and publish an updated view state.

That is wired together in the `RosterViewModel` constructor:

```
public RosterViewModel(Application ctxt) {
  super(ctxt);

  ObservableTransformer<Result, ViewState> toView=
    results -> (results.map(result -> {
      lastState=foldResultIntoState(lastState, result);

      return(lastState);
    }));

  Controller controller=new Controller(ctxt);

  states=LiveDataReactiveStreams
    .fromPublisher(controller.resultStream()
      .subscribeOn(Schedulers.single())
      .compose(toView)
      .cache()
      .toFlowable(BackpressureStrategy.LATEST)
      .share());
  controller.subscribeToActions(actionSubject);
  process(Action.load());
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/RosterViewModel.java)

The `stateStream()` method that our views use to get the updated live states is a `LiveData`, held onto as a `states` field and exposed via `stateStream()`. To create `states`, we:

- Get the `resultStream()` `Observable` from the `Controller`
- Arrange to process those `Result` objects on a background thread
- Use the `toView` `ObservableTransformer` to convert `Result` objects into a new `ViewState` — we will examine this part in greater detail shortly
- Cache the resulting `ViewState`
- Convert the `Observable` to a `Flowable`, only worrying about the latest `ViewState` that we receive
- `share()` that `Flowable` among multiple subscribers
- Convert that `Flowable` into a `LiveData` using `LiveDataReactiveStreams.fromPublisher()`

Merging Results Into the ViewState

An `ObservableTransformer` is simply a way of packaging an RxJava operator or chain of operators into a separate object. That can be useful in cases where:

- You might want to reuse the same operator(s) in multiple chains
- The operator might be fairly complex, and so you want to pull it out of the chain declaration to keep the chain itself more readable

So, let's look at that operator more closely:

```
    ObservableTransformer<Result, ViewState> toView=
      results -> (results.map(result -> {
        lastState=foldResultIntoState(lastState, result);

        return(lastState);
      }));
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/RosterViewModel.java)

We get in our `Result` stream, and our declaration says that we are emitting a `ViewState`. We are using the `map()` operator to make that conversion, where the bulk of the logic lies in a `foldResultIntoState()` method.

What we are trying to do is to mix a new `Result` with the previous `ViewState` to get a new `ViewState`. This implies that we *have* the previous `ViewState` somewhere. That is the `lastState` field, initialized to be an empty `ViewState` at the outset:

```
  private ViewState lastState=ViewState.empty().build();
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/RosterViewModel.java)

It is the job of `foldResultIntoState()` to create the new `ViewState`, which the `ObservableTransformer` both holds in `lastState` and returns to flow through the rest of the chain.

`foldResultIntoState()` needs to identify the specific type of `Result` (e.g., we added an item, we deleted an item) and update the `ViewState`. `foldResultIntoState()` mostly handles the first part: identifying the specific type of `Result`:

```
  private ViewState foldResultIntoState(@NonNull ViewState state,
    @NonNull Result result) throws Exception {
    if (result instanceof Result.Added) {
      return(state.add(((Result.Added)result).model()));
    }
```

```
    else if (result instanceof Result.Modified) {
      return(state.modify(((Result.Modified)result).model()));
    }
    else if (result instanceof Result.Deleted) {
      return(state.delete(((Result.Deleted)result).models()));
    }
    else if (result instanceof Result.Loaded) {
      List<ToDoModel> models=((Result.Loaded)result).models();

      return(ViewState.builder()
        .isLoaded(true)
        .items(models)
        .filterMode(((Result.Loaded)result).filterMode())
        .current(models.size()==0 ? null : models.get(0))
        .build());
    }
    else if (result instanceof Result.Selected) {
      return(state.selected(((Result.Selected)result).position()));
    }
    else if (result instanceof Result.Unselected) {
      return(state.unselected(((Result.Unselected)result).position()));
    }
    else if (result instanceof Result.UnselectedAll) {
      return(state.unselectedAll());
    }
    else if (result instanceof Result.Showed) {
      return(state.show(((Result.Showed)result).current()));
    }
    else if (result instanceof Result.Filter) {
      return(state.filtered(((Result.Filter)result).filterMode()));
    }
    else {
      throw new IllegalStateException("Unexpected result type: "+result.toString());
    }
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/RosterViewModel.java)

In the case of `Result.Loaded`, we are creating a brand-new `ViewState` from scratch. We only get this event when we first load the data, and so there is no meaningful prior state to use. In all the other scenarios, we call mutation methods on the existing `ViewState`, which turn around and create a new `ViewState` with the requested changes applied:

```
  ViewState add(ToDoModel model) {
    List<ToDoModel> models=new ArrayList<>(items());

    models.add(model);
    sort(models);

    return(toBuilder()
      .items(Collections.unmodifiableList(models))
      .current(model)
      .build());
  }

  ViewState modify(ToDoModel model) {
```

```
  List<ToDoModel> models=new ArrayList<>(items());
  ToDoModel original=find(models, model.id());

  if (original!=null) {
    int index=models.indexOf(original);
    models.set(index, model);
  }

  sort(models);

  return(toBuilder()
    .items(Collections.unmodifiableList(models))
    .build());
}

ViewState delete(List<ToDoModel> toDelete) {
  List<ToDoModel> models=new ArrayList<>(items());

  for (ToDoModel model : toDelete) {
    ToDoModel original=find(models, model.id());

    if (original==null) {
      throw new IllegalArgumentException("Cannot find model to delete: "+model.toString());
    }
    else {
      models.remove(original);
    }
  }

  sort(models);

  return(toBuilder()
    .items(Collections.unmodifiableList(models))
    .build());
}

ViewState selected(int position) {
  HashSet<Integer> selections=new HashSet<>(selections());

  selections.add(position);

  return(toBuilder()
    .selections(Collections.unmodifiableSet(selections))
    .build());
}

ViewState unselected(int position) {
  HashSet<Integer> selections=new HashSet<>(selections());

  selections.remove(position);

  return(toBuilder()
    .selections(Collections.unmodifiableSet(selections))
    .build());
}

ViewState unselectedAll() {
  return(toBuilder()
    .selections(Collections.unmodifiableSet(new HashSet<>()))
    .build());
```

```
}

ViewState show(ToDoModel current) {
  return(toBuilder()
    .current(current)
    .build());
}

ViewState filtered(FilterMode mode) {
  return(toBuilder()
    .filterMode(mode)
    .build());
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ViewState.java)

In all cases, we get a new immutable `ViewState`, which then flows out of the `RosterViewModel` to the view layer, so the fragments can update their UI as needed.

Examining the Other Fragments

While most of the code that we have reviewed so far was common for all three fragments, it is worth mentioning the other two fragments, as they have some different wrinkles.

DisplayFragment

`DisplayFragment` does not publish any actions itself. It is simply a passive consumer of view states.

However, its view state logic is a bit more complicated than is the `RosterListFragment`. While the user may think that the `DisplayFragment` is only displaying one to-do item, in reality it uses a `RecyclerView` to display the same roster that `RosterListFragment` does. While `RosterListFragment` displays the roster in a vertically-scrolling list, `DisplayFragment` displays the roster via horizontal swipes, akin to the behavior of a `ViewPager`. The `DisplayFragment` needs to know what particular to-do item to be displaying. That can be determined by:

- the `RosterListFragment` and `MainActivity`, if the user taps on an item in the list
- the user, via swipe gestures

As a result, `DisplayFragment` has *two* view-models. `AbstractRosterFragment` handles most of the logic for working with the `RosterViewModel`, as it does for `RosterListFragment`. `DisplayFragment` has its own `DisplayViewModel` that keeps

track of the currently-viewed page in the "pager", so we can restore this after a configuration change.

EditFragment

`EditFragment` publishes three actions:

- If the user saves the form, and this is a new to-do item, it publishes an "add" action
- If the user saves the form, and this is an existing to-do item, it publishes an "edit" action
- If the user taps on the "Delete" toolbar button and confirms this request, it publishes a "delete" action

Unlike `DisplayFragment`, `EditFragment` only ever works on a single to-do item. The ID of the `ToDoModel` which it is editing arrives via the `newInstance()` factory method and is saved in the arguments `Bundle`, so we do not need a separate view-model for it.

Summary

This sample app is *far* from perfect.

Strategically, this is quite a bit of code for a silly little to-do app. To an extent, the value of formal architectures increases with the size and scope of the app. The more "disposable" the app is, the more likely it is that you can skip some architectural complexity and just focus on writing a working app. As the preceding chapter noted, for smaller apps, YAGNI (You Aren't Going to Need It).

Tactically, the sample app makes a core simplifying assumption: that the entire set of to-do items fits comfortably in memory. Sometimes that will be the case with full production-grade apps. Sometimes that will not be the case, and the complexity of the app rises.

Advanced Topics

Backing Up a Room

In some apps, the local database is little more than a cache. Perhaps it is a true cache, where the app directly hits the network when some necessary data is not in the database. Perhaps the app uses a synchronization model, where the network is still the "system of record", but the local copy can be used offline and is periodically synchronized with the server.

But, in other apps, the local database itself is the "system of record". Android does not offer a real backup solution — the best it has amounts to a disaster recovery option, since users have little ability to restore a backup. Third-party apps cannot back up the internal storage of other apps (except perhaps on rooted devices), which is why there are few backup utilities for Android. Instead, if you want your app's local data to be backed up, you need to do that yourself.

So, with that in mind, let's explore backing up SQLite databases, with an eye towards Room.

What Do We Need to Back Up?

The first question whenever "backup" is discussed is that of scope: what files are we supposed to back up and be able to restore?

Plan A: The Whole Directory

The simple solution is to back up the entire directory that contains database files. That way, you do not care exactly what files are in there — you just back up everything.

To find out the default directory for Room database files, use:

```
ctxt.getDatabasePath("foo").getParentFile()
```

where `ctxt` is some `Context` and `"foo"` is, well, `"foo"`. In theory, you could replace `"foo"` with the name of a real database, but since you are not actually going to use that value, it just needs to be some valid string. This code snippet will return the directory where that database would be stored.

Given that directory, you could then just back up the entire directory's contents, knowing that your database and its files will be in there.

Plan B: Just SQLite's Files

However, that directory may contain databases other than yours. For example, if you are using `WebView`, it may have databases there for its own caches. Third-party libraries might have databases as well. It may make sense to back all of them up as well, in which case Plan A is ideal. But, perhaps you have reasons to only back up your own database.

You are in control over the name of the database file, as you provide it to your `RoomDatabase` as a part of configuring the `RoomDatabase.Builder`:

```
Room.databaseBuilder(ctxt.getApplicationContext(), ToDoDatabase.class,
"stuff.db").build();
```

Here, `stuff.db` is the name of the database.

It is possible that Room and SQLite will have more files than just this one, though. Ideally, that will not be the case. But, you may want to consider backing up all files that *begin* with your database name, as other files, such as ones with `-shm` or `-wal` suffixes, might be around.

One risk with this would be if Room started writing other files of its own, instead of just whatever SQLite uses. The same risk holds if you use other `SupportSQLiteDatabase` implementations.

Beware of Open Rooms

SQLite database files are just ordinary files. There is nothing particularly magic about actually making the backup: just make a copy of the file.

However, as with most files, it is important that nothing be trying to write to that

file while we are making a copy. Otherwise, we may have a mix of old and new data, and most likely that will result in a corrupted and unusable database.

So, we need to take steps to ensure that nothing is trying to use the database while the backup is going on (or while we are restoring a backup). We will explore specific ways of doing that here in this chapter.

When Do We Back Up the Database?

Databases do not get backed up automatically — we have to do that work. That in turn implies that we know *when* to do that work.

When the User Asks

The simple possibility is to back up the database when the user asks, via an action bar item or some similar option.

However, copying a database file will take some time. Exactly how long it will take depends on a lot of factors, particularly the size of the file and the speed of the device. We need to make sure that the user is not doing things in our app that need the database while the backup is going on. One approach is to have the backup be performed while some dedicated UI — such as an activity with a `ProgressBar` — is in the foreground.

When You Feel Like It

In principle, you could try backing up the database at odd times, when you think that the user is not going to be using the app. This has the advantage of being automatic, so you can offer to restore a backup that the user did not request manually. Plus, in theory, the backup will not disturb the user if you are doing it, say, in the middle of the night.

However:

- Some users might want to use your app in the middle of the night
- If you have your own scheduled tasks (e.g., `WorkManager` periodic work), that might run in the middle of your backup
- If the user tries to get into your app while the backup is running, you would need smarts in all of your app's entry points (e.g., launcher activity) to detect that a backup is going on and take steps to not use the database

You might think that your app has only one entry point: the launcher activity. However, suppose that the user had been using your app 10 minutes ago, and you decide to run the automatic backup. While that backup is ongoing, the user tries returning to your task (intentionally or accidentally) via the overview screen. The user will be returned to *whatever activity they were on last*, which may or may not be the launcher activity.

One way to help reduce the likelihood of problems is to have some sort of a usage timer, perhaps using `ProcessLifecycleOwner`, where you do not do the backup until at least 30 minutes has elapsed since your UI was last in the foreground. This should cause the overview screen to return the user to your launcher activity, rather than to some other point based on an outstanding task.

When You Think Room Is Not Using It

As mentioned previously, part of the struggle is in ensuring that Room is not using the database. Keeping the user out of database-driven UI and stopping any database-related background tasks are key steps... but that may not be quite enough.

A Simple Close

`RoomDatabase` has a `close()` method. In principle, it should close the underlying `SupportSQLiteDatabase` and prevent further use of the `RoomDatabase`.

However, Room itself has its own background threads, particularly in a class named `InvalidationTracker`. This code monitors your database I/O and arranges to re-deliver updated data to reactive queries (e.g., where your DAO returns an `Observable` or `LiveData`). It is unclear whether `InvalidationTracker` will shut down when you `close()` the `RoomDatabase`, and `InvalidationTracker` does not have any API of its own to shut it down.

The Nuclear Option

The safest way to ensure that nothing is using your Room database is a two-step process:

1. Put your backup and restore logic in a separate process
2. When it comes time to actually perform a backup or restore operation, you fork that separate process... then kill your main app's process

In the dedicated backup/restore process, you do not use Room and you do not open the SQLite database. You just copy files around.

Terminating your own process in Android is easy enough. The `android.os.Process` class has a `killProcess()` method that will kill a process given its process ID ("PID"). You get your own process' ID by calling `myPid()` on `android.os.Process`. Note that the class here is `android.os.Process` — be careful, as Java has its *own* `Process` class, one that is always visible owing to it being in the `java.lang` package.

(pro tip: do not name your own classes the same as classes in `java.lang`)

Terminating your own process itself is somewhat risky. For example, you should not assume that `onDestroy()` of anything gets called. This is why we rarely do this and certainly do not do it as a part of normal app operations. But, there are extraordinary cases where it may be useful: backup and restore operations are two such cases.

A Basic Backup Example

With all that as prologue, let's look at the backup and restore logic in the `ToDo/MVI` sample project, originally covered in a chapter on the Model-View-Intent architecture pattern.

Triggering the Operation

The `RosterListFragment` has a pair of action bar items for "Backup" and "Restore". These are tied to `backup()` and `restore()` methods, which in turn pass control to `launchAndGoAway()`:

```
private void backup() {
  launchAndGoAway(true);
}

private void restore() {
  launchAndGoAway(false);
}

private void launchAndGoAway(boolean isBackup) {
  ToDoDatabase.shutdown();
  startActivity(BackupRestoreActivity.newIntent(getActivity(), isBackup));
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/RosterListFragment.java)

`launchAndGoAway()` starts by calling a `shutdown()` method on `ToDoDatabase`, which simply calls `close()` on our singleton (if it exists) and sets that singleton field to `null`:

```
public synchronized static void shutdown() {
  if (INSTANCE!=null) {
    INSTANCE.close();
    INSTANCE=null;
  }
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/impl/ToDoDatabase.java)

In principle, this resets matters to where we were before we started using the database. Right now, `close()` does not appear to perform any disk I/O; if some future version of Room does perform I/O here, we would want to make our `shutdown()` call be asynchronous (perhaps using an RxJava `Completable`).

In addition to `ToDoDatabase.shutdown()`, this would be the spot where our code would need to suspend any other background work, particularly work that might be triggered even if our process goes away. This includes things like `WorkManager`, `JobScheduler`, and `AlarmManager`. In this case, the app has none of these things, so there is nothing that we need to worry about.

Finally, `launchAndGoAway()` retrieves an `Intent` from `BackupRestoreActivity`, then starts an activity based on that `Intent`. `BackupRestoreActivity` is where our actual backup and restore logic resides. That `newIntent()` method builds an `Intent` identifying `BackupRestoreActivity` and populates that `Intent` as `BackupRestoreActivity` needs:

```
static Intent newIntent(Context ctxt, boolean isBackup) {
  return new Intent(ctxt, BackupRestoreActivity.class)
    .addFlags(Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_CLEAR_TASK)
    .putExtra(EXTRA_IS_BACKUP, isBackup)
    .putExtra(EXTRA_MAIN_PID, Process.myPid());
}
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/BackupRestoreActivity.java)

Specifically, we:

- Use

`addFlags(Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_CLEAR_TASK)` to destroy any outstanding activities (in this case, it would be just `MainActivity`)
- Add the supplied boolean value as an extra, to denote whether this is a backup or a restore request
- Capture our PID and put that as an extra as well

Our UI

`BackupRestoreActivity` uses a dialog theme (`Theme.Apptheme.Dialog`), so it does not take up the entire screen:

```
<style name="Theme.Apptheme.Dialog" parent="@android:style/Theme.Material.Light.Dialog.NoActionBar">
  <item name="android:colorPrimary">@color/primary</item>
  <item name="android:colorPrimaryDark">@color/primary_dark</item>
  <item name="android:colorAccent">@color/accent</item>
  <item name="android:windowMinWidthMajor">@android:dimen/dialog_min_width_major</item>
  <item name="android:windowMinWidthMinor">@android:dimen/dialog_min_width_minor</item>
</style>
```

(from ToDo/MVI/app/src/main/res/values/styles.xml)

Our layout (`activity_progress`) is just a `ProgressBar` and `TextView`, wrapped in a `ConstraintLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="16dp">

  <ProgressBar
    android:id="@+id/progressBar"
    style="?android:attr/progressBarStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <TextView
    android:id="@+id/title"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@string/msg_wait"
```

```
    android:textAppearance="@android:style/TextAppearance.Material.Large"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/progressBar"
    app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

(from ToDo/MVI/app/src/main/res/layout/activity_progress.xml)

The net result is that we will show indefinite progress as our UI:

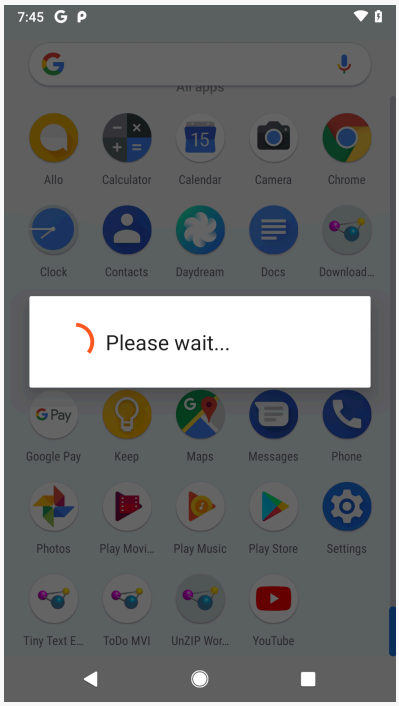


Figure 18: RoomBackup UI

In `onCreate()`, we load up that layout, plus we get an instance of our viewmodel (`VM`) via a `VMFactory`:

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_progress);

  VMFactory factory=
    new VMFactory(getApplication(),
      getIntent().getBooleanExtra(EXTRA_IS_BACKUP, true),
      getIntent().getIntExtra(EXTRA_MAIN_PID, -1));
```

```
    VM vm=ViewModelProviders.of(this, factory).get(VM.class);

    vm.results.observe(this, unused -> {
      startActivity(MainActivity.newIntent(this));
      finish();
    });
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/BackupRestoreActivity.java)

That `VMFactory` takes our extras and passes them into the viewmodel:

```
  private class VMFactory extends ViewModelProvider.AndroidViewModelFactory {
    private final boolean isBackup;
    private final int pid;

    public VMFactory(@NonNull Application app, boolean isBackup, int pid) {
      super(app);
      this.isBackup=isBackup;
      this.pid=pid;
    }

    @NonNull
    @Override
    public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {
      return (T)new VM(getApplication(), isBackup, pid);
    }
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/BackupRestoreActivity.java)

The `VM` viewmodel exposes a `LiveData` named `results`, which will emit an object when our backup or restore is completed. At that point, we start up `MainActivity` again and `finish()` the `BackupRestoreActivity`, so the user is taken right back to the main app UI.

Performing the Operation

The "heavy lifting" for the backup and restore operations is handled inside of `VM` for simplicity. One could argue that this sort of work belongs in a repository, and for mainstream functionality or a larger app that may make sense. However, the only place where such a repository would be used is right here in this viewmodel, so it is unclear how this book example would be improved by using a repository.

In `onCreate()`, we set up an RxJava `Single` that invokes a `process()` method that

will do the real work. The `Single` just allows us to do that work on a background thread, plus get a `LiveData` for our UI layer to use (via `LiveDataReactiveStreams.fromPublisher()`):

```
public static class VM extends AndroidViewModel {
  final LiveData<Boolean> results;

  public VM(@NonNull Application application, boolean isBackup, int pid) {
    super(application);

    Single<Boolean> backup=
      Single.create((SingleOnSubscribe<Boolean>)emitter -> {
        process(isBackup, pid);
        emitter.onSuccess(true);
      })
      .subscribeOn(Schedulers.single())
      .observeOn(AndroidSchedulers.mainThread());

    results=LiveDataReactiveStreams.fromPublisher(backup.toFlowable());
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/BackupRestoreActivity.java)

`process()` starts off by sleeping for one second, to give that main process extra time for any cleanup, particularly for any asynchronous work that we do not control. Then, it uses `Process.killProcess()` to terminate the main process, using the PID passed in via the extra:

```
  private void process(boolean isBackup, int pid) throws IOException {
    SystemClock.sleep(1000);  // wait for things to settle

    Process.killProcess(pid);

    File dbDir=getApplication().getDatabasePath("foo").getParentFile();
    File extDir=getApplication().getExternalFilesDir(null);
    File backupDir=new File(extDir, "db-backup");

    if (isBackup) {
      if (backupDir.exists()) {
        delete(backupDir);
      }

      backupDir.mkdirs();
      copy(dbDir, backupDir);
    }
    else {
      if (dbDir.exists()) {
```

```
          delete(dbDir);
        }

        dbDir.mkdirs();
        copy(backupDir, dbDir);
      }
    }
  }
```

(from ToDo/MVI/app/src/main/java/com/commonsware/android/todo/ui/BackupRestoreActivity.java)

What happens next depends a bit on the operation:

- If we are performing a backup, we make an empty directory for that backup, then copy the files from the database directory to the backup directory
- If we are performing a restore, we make an empty directory for the database, then copy the files from the backup directory to the database directory

You can see all of this in action by running the sample app:

- Create a couple of to-do items
- Run a backup
- Change your roster of to-do items (e.g., add some more)
- Run a restore
- See that your backed-up roster of to-do is restored

Areas for Improvement

There are many things that could be improved in this sample, things that a production app would need to have:

- We are doing no validation to confirm that the backed up data is a complete bit-for-bit copy of the original data. *Probably* it is, but random I/O hiccups could give us a corrupt backup, which is useless.
- We are deleting the "real" database prior to restoring the backup. If the restore process fails for some reason, we are now stuck. We could, instead, rename the database directory, restore to a new database directory, and confirm the restore result to ensure that it is a bit-for-bit copy of the backup *and* we can successfully open the database. If that succeeds, we can then delete the renamed older database directory. If the restore fails, we can delete the failed restored copy, rename the old database directory back to its original name, and be right back where we started prior to the restoration attempt.

- We are only allowing one backup, in a location chosen by us as developers. We could use `ACTION_OPEN_DOCUMENT_TREE` to allow the user to choose where to make the backup. In this case, though, we cannot be completely certain that the backup location will be on the device, as we get a `Uri` to a documents provider. It is possible that working with that provider will be slow, particularly if it is transferring the data in real time to a server. It may be better to make a local backup first, then copy the backup to the provider.
- We are not validating the backup. We assume that it exists, and that it contains a valid database. Clearly, this might not be the case.

Backing Up Off-Device

All of this has focused on making a backup to a location on the device. That is useful, but that usefulness fades if the device itself is lost, damaged, or destroyed. For that, we would need to backup to an off-device location.

However, that may be slow and may not be possible at the time of the backup, due to the nature of Internet connections. As with the document provider scenario mentioned in the preceding section, consider backing up locally first, then transferring the backup to the server as a separate second step. Once the local backup is complete, you can let the app return to normal operation, as transferring the backup to a server should not be affected by other work done on the live database.

Room and Full-Text Searching

SQLite supports FTS virtual tables for full-text searching of content.

Room does not.

NOTE: The upcoming 2.1.0 version of Room will support FTS. That edition of Room is part of AndroidX; this book focuses on the original `android.arch` edition of Room, which pre-dates this FTS support.

However, there are ways to get FTS support in a Room-managed database, but it requires you to do more of the work yourself. In this chapter, we will explore how to make this work, while also traveling in time. Or perhaps just reading about somebody who travels in time.

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.

There are a few full-text indexing options available in SQLite: FTS3, FTS4, and FTS5. Newer versions (higher numbers) are generally faster but take more disk space. FTS3 has been around since the beginning of Android. FTS4 arrived with the version of SQLite used in API Level 11. FTS5, by constrast, is only likely to be available on API Level 24 and higher devices.

This chapter does not cover all of the details of using FTS with SQLite. For that, please see the "Advanced Database Techniques" chapter in *The Busy Coder's Guide to Android Development* and the SQLite FTS documentation. This chapter is focused solely on enabling this stuff from Room, though you will see a bit of how FTS works along the way.

The Room 1.x FTS Recipe

Room 1.x cannot create an FTS table for us. However, Room *can* work with an FTS table, at least for data retrieval, using `@RawQuery`.

As a result, we can get partial support for FTS from Room, if we are willing to do the rest ourselves, using the support database API, such as what we use with migrations.

Manually Create the Table

First, we will need to manually create our FTS virtual table.

At minimum, we need to do this when we create the database. We can use a `RoomDatabase.Callback` object for that, as part of setting up our `RoomDatabase.Builder`:

```
RoomDatabase.Builder<BookDatabase> b=
  Room.databaseBuilder(ctxt.getApplicationContext(), BookDatabase.class,
    DB_NAME);

b.addCallback(new Callback() {
  @Override
  public void onCreate(@NonNull SupportSQLiteDatabase db) {
    super.onCreate(db);

    db.execSQL("CREATE VIRTUAL TABLE booksearch USING fts4(sequence, prose)");
  }
});

BookDatabase books=b.build();
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

`RoomDatabase.Callback` was covered in the chapter on the support database API.

Here, as part of setting up an instance of a `BookDatabase`, we request to get control when the database is created and add our own `booksearch` virtual table.

If you are adding the FTS table to an already-existing app, you will also need to create your FTS virtual table in the appropriate `Migration` object(s).

RawQuery the Table

To get data out of the virtual table, we can use a `@RawQuery`-annotated method on some `@Dao` class that references the table that we created manually. While `@RawQuery` can work with tables defined via Room entities, that is not a requirement. `@RawQuery` can work with *any* table, so long as Room can figure out how to map the columns that you request to the POJO that you want to return.

So, given some sort of `BookSearchResult` POJO that matches the `booksearch` table structure used above, we could have:

```
@RawQuery
protected abstract List<BookSearchResult> _searchSynchronous(SupportSQLiteQuery query);
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

Things get a bit tricky if we want a reactive response (e.g., `Observable`) or if we want to use paging, as we will see later in this chapter.

To put data into the virtual table, the official solution is to use a `SupportSQLiteDatabase`. You get one by calling `getOpenHelper().getWritableDatabase()` on your `RoomDatabase`. Then, just as we used one of those for a `CREATE VIRTUAL TABLE` statement (see above), we can use one for `INSERT`/`UPDATE`/`DELETE` statements or the corresponding `insert()`, `update()`, or `delete()` methods.

Unofficially, `@RawQuery` works for these as well:

```
@RawQuery
protected abstract long _insert(SupportSQLiteQuery queryish);
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

Searching a Book

The above code snippets come from the `TimeMachine/RoomFTS` sample project, demonstrating the use of FTS virtual tables for searching through large blocks of text. The techniques outlined in this chapter, via this sample, resemble the techniques used in the APK edition of *The Busy Coder's Guide to Android Development*, which offered a similar full-text search capability for a few years.

The Book

This sample app, though, does not allow searching through a book on Android app development. Instead, it allows the user to read and search a copy of the Project Gutenberg edition of H. G. Wells' "The Time Machine". This book is stored in a series of text files in the `assets/` directory. On first run, the app will pour that text into a SQLite database, both in a regular table and in an FTS4 virtual table. It then has two fragments, each with a `RecyclerView`:

- The first one shows the entire book, using the paging library to minimize the memory usage
- The second one shows FTS search results, again using the paging library, along with our semi-manual FTS support

The Data Model

The text files in `assets/` are subdivided into chapters, with the chapter title as the core of the filename:

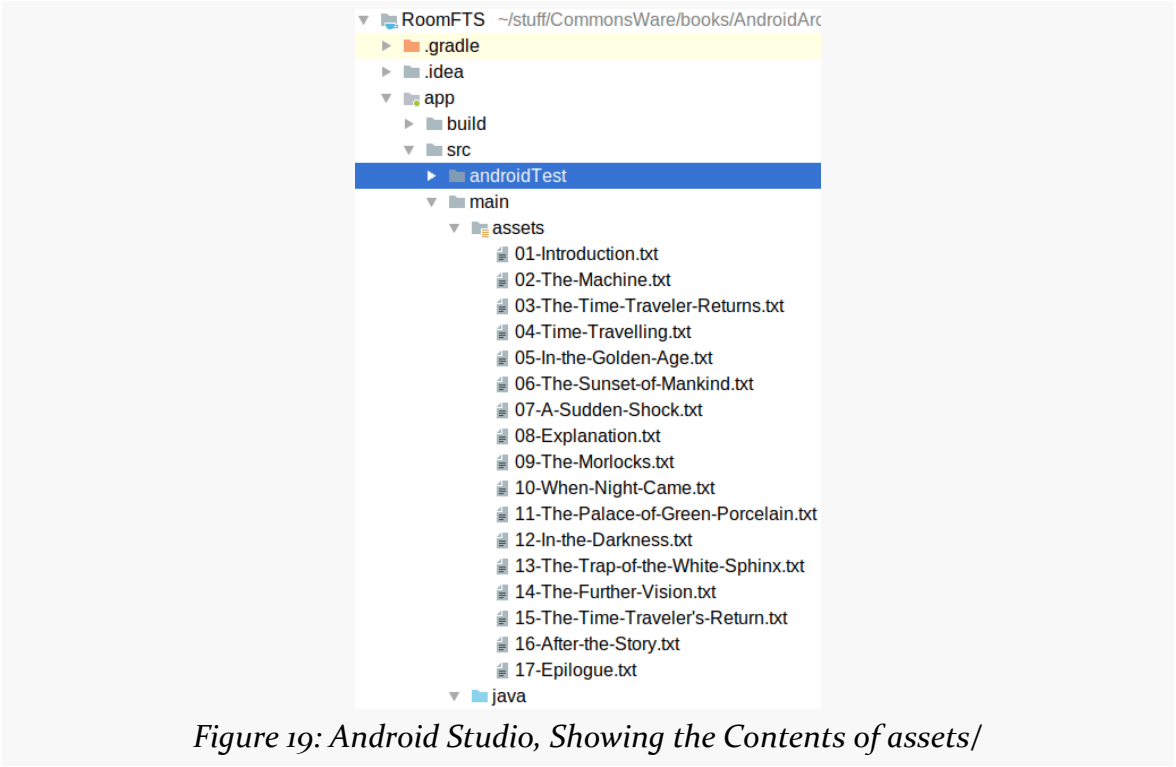


Figure 19: Android Studio, Showing the Contents of assets/

We want to be able to full-text search the prose of those chapters. For that, we would only need an FTS table, as we can get the text itself from the assets. However, to make this sample a bit more realistic, we will pour the book contents into regular SQLite tables, then add an FTS table for searching. That will more closely resemble common FTS scenarios, where the data to be searched is also in SQLite.

To that end, we have a `ChapterEntity` to model a chapter:

```
package com.commonsware.android.room.fts;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.PrimaryKey;

@Entity(tableName = "chapters")
public class ChapterEntity {
  @PrimaryKey(autoGenerate = true)
```

```
  long id;
  String title;

  ChapterEntity(String title) {
    this.title=title;
  }
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/ChapterEntity.java)

Notably, this has the chapter title, but it does not have the chapter prose. Instead, we have a 1:N relation with a `ParagraphEntity`, that will represent an individual paragraph from a chapter:

```
package com.commonsware.android.room.fts;

import android.arch.persistence.room.Entity;
import android.arch.persistence.room.ForeignKey;
import android.arch.persistence.room.Index;
import android.arch.persistence.room.PrimaryKey;

@Entity(tableName="paragraphs",
  foreignKeys=@ForeignKey(entity=ChapterEntity.class, parentColumns="id",
    childColumns="chapterId", onDelete=ForeignKey.CASCADE),
  indices={@Index(value="chapterId")})
public class ParagraphEntity {
  @PrimaryKey
  long sequence;
  String prose;
  long chapterId;

  ParagraphEntity(String prose) {
    this.prose=prose;
  }
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/ParagraphEntity.java)

Here, the `sequence` is the order in which this paragraph appears in the book overall. By dividing our prose into paragraphs, we can offer per-paragraph FTS searches. This app does not presently use the chapter information, though it could (e.g., section headings in the scrollable book, nav drawer to jump to a particular chapter).

The Database

Our database needs to hold the chapters, paragraphs, and full-text search index.

Moreover, when we create the database, we need to set up those chapters, paragraphs, and full-text search index.

This gets a bit tricky, particularly if we want to use Room for as much of this as possible.

The `RoomDatabase` subclass in this app is `BookDatabase`, with a declaration that it is tied to `ChapterEntity`, `ParagraphEntity`, and a `BookStore` that is our DAO:

```
@Database(entities = {ChapterEntity.class, ParagraphEntity.class}, version=1)
abstract public class BookDatabase extends RoomDatabase {
  static final String DB_NAME="time-machine.db";

  abstract public BookStore store();
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

As with many of the other samples in this book, the `BookDatabase` is a singleton:

```
  private static volatile BookDatabase INSTANCE=null;

  synchronized static BookDatabase get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=create(ctxt);
    }

    return INSTANCE;
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

And, as with many of the other samples, the `create()` method creates our database:

```
  private static BookDatabase create(Context ctxt) {
    RoomDatabase.Builder<BookDatabase> b=
      Room.databaseBuilder(ctxt.getApplicationContext(), BookDatabase.class,
        DB_NAME);

    b.addCallback(new Callback() {
      @Override
      public void onCreate(@NonNull SupportSQLiteDatabase db) {
        super.onCreate(db);

        db.execSQL("CREATE VIRTUAL TABLE booksearch USING fts4(sequence, prose)");
      }
    });

    BookDatabase books=b.build();

    populate(ctxt, books);
```

```
    return books;
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

Our `Callback` gives us control when the database is created, and we directly create a `booksearch` FTS table using the supplied `SupportSQLiteDatabase`. This bypasses the need for a Room entity for this table. If this app had more than one schema version, and `booksearch` was added in a later schema, we would also be registering `Migration` objects, where one or more of those would also create this `booksearch` FTS table.

The other departure is the `populate()` call. In that method, we load the `BookDatabase` with our chapters and paragraphs, if that data does not already exist. In principle, we could do this in `onCreate()` of the `Callback`. However, in that method, all we have is a `SupportSQLiteDatabase`, meaning that we cannot use our Room entities and DAO, but instead would roll all of the database code manually. If we do that, we might as well consider dumping Room altogether. So, instead, `populate()` will check the database to see if we already have our prose loaded, and if not, will load the prose into the database.

Our `BookStore` has a simple `chapterCount()` `@Query` method to return the number of chapters:

```
@Query("SELECT COUNT(*) FROM chapters")
abstract int chapterCount();
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

`populate()` starts by calling that method to determine whether we our database is empty:

```
private static void populate(Context ctxt, BookDatabase books) {
  if (books.store().chapterCount()==0) {
    try {
      AssetManager assets=ctxt.getAssets();
      int sequenceNumber=0;

      for (String path : assets.list("")) {
        List<String> paragraphs=paragraphs(assets.open(path));
        String title=title(path);
        ChapterEntity chapter=new ChapterEntity(title);
        List<ParagraphEntity> paragraphEntities=new ArrayList<>();
```

```
        for (String paragraph : paragraphs) {
          paragraphEntities.add(new ParagraphEntity(paragraph));
        }

        sequenceNumber=
          books.store().insert(chapter, paragraphEntities, sequenceNumber);
      }
    }
    catch (IOException e) {
      Log.e("BookDatabase", "Exception reading in assets", e);
    }
  }
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

If it is empty, we use `AssetManager` to iterate over those 17 text files in `assets/`. For each, we use a `paragraphs()` method to divide the text into paragraphs, with a blank line serving as the delimiter between paragraphs:

```
// inspired by https://stackoverflow.com/a/10065920/115145

private static List<String> paragraphs(InputStream is) throws IOException {
  BufferedReader in=new BufferedReader(new InputStreamReader(is));
  List<String> result=new ArrayList<>();

  try {
    StringBuilder paragraph=new StringBuilder();

    while (true) {
      String line=in.readLine();

      if (line==null) {
        break;
      }
      else if (TextUtils.isEmpty(line)) {
        if (!TextUtils.isEmpty(paragraph)) {
          result.add(paragraph.toString().trim());
          paragraph=new StringBuilder();
        }
      }
      else {
        paragraph.append(line);
        paragraph.append(' ');
      }
    }
```

```
      if (!TextUtils.isEmpty(paragraph)) {
        result.add(paragraph.toString().trim());
      }
    }
    finally {
      is.close();
    }

    return result;
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

We use a `title()` method to convert the "slug"-style filename into a normal title, using spaces instead of dashes for word separators:

```
  private static String title(String path) {
    String[] pieces=path.substring(0, path.length()-4).split("-");
    StringBuilder buf=new StringBuilder();

    for (int i=1;i<pieces.length;i++) {
      buf.append(pieces[i]);
      buf.append(' ');
    }

    return buf.toString().trim();
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookDatabase.java)

We convert the paragraphs into `ParagraphEntity` instances, then call an `insert()` method on our `BookStore`, passing it the `ChapterEntity` and each of the `ParagraphEntity` instances to be inserted into our database, plus the last-used sequence number (initialized at 0). `insert()` is supposed to put all of this data into the database and return the new last-used sequence number, to be applied in the next pass of the loop.

`insert()` uses a `@Transaction` annotation, so it can perform multiple database operations inside of a single transaction:

```
  @Transaction
  int insert(ChapterEntity chapter, List<ParagraphEntity> paragraphs,
             int startingSequenceNo) {
    long chapterId=insert(chapter);

    for (ParagraphEntity paragraph : paragraphs) {
```

```
      paragraph.chapterId=chapterId;
      paragraph.sequence=startingSequenceNo++;
      insertFTS(paragraph);
    }

    insert(paragraphs);

    return startingSequenceNo;
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

First, `insert()` calls another `insert()` method, to insert a `ChapterEntity`. That is just a standard `@Insert` method:

```
  @Insert
  abstract long insert(ChapterEntity chapter);
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

It returns the primary key used for this `ChapterEntity`, as the entity has `@PrimaryKey(autoGenerate = true)` on its `long id` field. We can then use that, plus an incremented sequence number, to fill in the missing details on the `ParagraphEntity`.

And, at this point, we cheat.

As was noted earlier in the chapter, we are supposed to use `SupportSQLiteDatabase` to work with `booksearch`. In reality, at least with Room `1.1.0`, we can use `@RawQuery` for this. However, `@RawQuery` requires that its method take a `SupportSQLiteQuery` as a parameter:

```
  @RawQuery
  protected abstract long _insert(SupportSQLiteQuery queryish);
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

That is inconvenient. So, we wrap that in an `insertFTS()` method that generates the `SupportSQLiteQuery`:

```
  void insertFTS(ParagraphEntity entity) {
    _insert(new SimpleSQLiteQuery("INSERT INTO booksearch (sequence, prose) VALUES (?, ?)",
      new Object[] {entity.sequence, entity.prose}));
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

The `_insert()` method has the leading underscore and is marked as `protected` to try to minimize the likelihood that anyone outside of `BookStore` would use that method. Unfortunately, we cannot make the method be `private`, because then the Room code generator cannot generate an implementation in a concrete subclass of `BookStore`.

Once we have iterated over all of the `ParagraphEntity` instances and added them to the FTS table, we insert them into their own table, using another `insert()` method:

```
@Insert
abstract void insert(List<ParagraphEntity> paragraphs);
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

Finally, we return the updated sequence number, for use in a future pass of this `insert()` method.

The net: when we open our `BookDatabase` and start working with it, we will lazy-populate the database, including the FTS table.

The `BookStore` also has a method for loading our paragraphs, in sequence order, using the paging library to maximize speed and minimize memory usage:

```
@Query("SELECT * FROM paragraphs ORDER BY sequence ASC")
abstract DataSource.Factory<Integer, ParagraphEntity> paragraphs();
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

The Searches

To search the `booksearch` FTS table, we turn once again to `@RawQuery` methods on the `BookStore`:

```
@RawQuery(observedEntities = ParagraphEntity.class)
protected abstract DataSource.Factory<Integer, BookSearchResult> _search(SupportSQLiteQuery query);

@RawQuery
protected abstract List<BookSearchResult> _searchSynchronous(SupportSQLiteQuery query);
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

There are two such methods, `_search` and `_searchSynchronous()`. The latter is for testing purposes, and it does the database I/O synchronously. The former is for the actual UI, and it uses the paging library.

However, `@RawQuery` insists upon having an `observedEntities` property, if your method returns an asynchronous type: `DataSource.Factory`, `LiveData`, `Observable`, etc. This is a fairly bizarre limitation, particularly given the fact that we are supposed to use `@RawQuery` for FTS tables, which have no associated entities. With luck, this will get addressed someday. In the meantime, we use `ParagraphEntity`, as being the closest entity that matches this table, and hope that this holds up.

Both of these methods return instances of a POJO named `BookSearchResult`:

```
package com.commonsware.android.room.fts;

public class BookSearchResult {
  long sequence;
  String snippet;
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookSearchResult.java)

The `sequence` field provides the primary key for the `ParagraphEntity`, should we want to find the full paragraph. However, mostly, we will use the `snippet` field, which will supply us with a SQLite-prepared bit of text highlighting our search term in the result.

As with `_insert()`, these two methods need a `SupportSQLiteQuery`, so we wrap them with methods that hide that detail:

```
  DataSource.Factory<Integer, BookSearchResult> search(String expr) {
    return _search(buildSearchQuery(expr));
  }

  List<BookSearchResult> searchSynchronous(String expr) {
    return _searchSynchronous(buildSearchQuery(expr));
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

```
  private SimpleSQLiteQuery buildSearchQuery(String expr) {
    return new SimpleSQLiteQuery("SELECT sequence, snippet(booksearch) AS snippet FROM booksearch WHERE
prose MATCH ? ORDER BY sequence ASC",
      new Object[] {expr});
  }
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookStore.java)

The Repository

Our `BookDatabase` and `BookStore` are wrapped in a `BookRepository`:

```
package com.commonsware.android.room.fts;

import android.arch.paging.DataSource;
import android.content.Context;
import io.reactivex.Single;

public class BookRepository {
  private static volatile BookRepository INSTANCE=null;
  private final Context ctxt;

  synchronized static BookRepository get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=new BookRepository(ctxt);
    }

    return INSTANCE;
  }

  private BookRepository(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
  }

  Single<DataSource.Factory<Integer, ParagraphEntity>> paragraphs() {
    return Single.create(emitter ->
      emitter.onSuccess(BookDatabase.get(ctxt).store().paragraphs()));
  }

  Single<DataSource.Factory<Integer, BookSearchResult>> search(String expr) {
    return Single.create(emitter ->
      emitter.onSuccess(BookDatabase.get(ctxt).store().search(expr)));
  }
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookRepository.java)

`DataSource.Factory` is asynchronous, in that our `paragraphs()` and `search()` methods do not perform any database I/O immediately. However, due to our lazy-loading of the book contents into the database, the `get()` method on `BookDatabase` *does* do database I/O, and potentially a fair bit of it (if this is the first run of the app). To avoid `StrictMode` violations and jank, we need to move `BookDatabase.get()` onto a background thread. So, the `BookRepository` wraps the `BookStore` edition of the `paragraphs()` and `search()` methods in its own, returning

a `Single`, so that we can perform the `BookDatabase.get()` call on a background thread.

The ViewModels

The `BookRepository` is used by two `AndroidViewModel` subclasses.

One — `BookViewModel` — will be used by the UI that displays the entire book contents:

```
package com.commonsware.android.room.fts;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.arch.lifecycle.LiveData;
import android.arch.lifecycle.LiveDataReactiveStreams;
import android.arch.lifecycle.Transformations;
import android.arch.paging.DataSource;
import android.arch.paging.LivePagedListBuilder;
import android.arch.paging.PagedList;
import io.reactivex.Single;
import io.reactivex.schedulers.Schedulers;

public class BookViewModel extends AndroidViewModel {
  final LiveData<PagedList<ParagraphEntity>> paragraphs;

  public BookViewModel(Application app) {
    super(app);

    Single<DataSource.Factory<Integer, ParagraphEntity>> liveParagraphs=
      BookRepository.get(app).paragraphs().subscribeOn(Schedulers.single());

    paragraphs=Transformations
      .switchMap(LiveDataReactiveStreams.fromPublisher(liveParagraphs.toFlowable().cache()),
        factory -> new LivePagedListBuilder<>(factory, 50).build());
  }
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookViewModel.java)

Here, we:

- Subscribe to the `Single` on a background thread (`subscribeOn(Schedulers.single())`)
- Convert that `Single` to a cached `Flowable`
- Wrap that in a `LiveData`, using `LiveDataReactiveStreams.fromPublisher()`
- Use `Transformations.switchMap()` to take the `DataSource.Factory` that we get from the `Single` and convert that into a `LivePagedListBuilder` for use in the UI

There is a similar `SearchViewModel` for processing the results of a search:

```
package com.commonsware.android.room.fts;

import android.app.Application;
import android.arch.lifecycle.AndroidViewModel;
import android.arch.lifecycle.LiveData;
import android.arch.lifecycle.LiveDataReactiveStreams;
import android.arch.lifecycle.Transformations;
import android.arch.paging.DataSource;
import android.arch.paging.LivePagedListBuilder;
import android.arch.paging.PagedList;
import io.reactivex.Single;
import io.reactivex.schedulers.Schedulers;

public class SearchViewModel extends AndroidViewModel {
  LiveData<PagedList<BookSearchResult>> results;

  public SearchViewModel(Application app) {
    super(app);
  }

  LiveData<PagedList<BookSearchResult>> search(String expr) {
    Single<DataSource.Factory<Integer, BookSearchResult>> liveSearch=
      BookRepository.get(getApplication()).search(expr).subscribeOn(Schedulers.single());

    results=Transformations
      .switchMap(LiveDataReactiveStreams.fromPublisher(liveSearch.toFlowable().cache()),
        factory -> new LivePagedListBuilder<>(factory, 50).build());

    return results;
  }
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/SearchViewModel.java)

The UI

Our `MainActivity` is pretty simple. It just loads up a `BookFragment` on startup. It also offers a `search()` method so that the `BookFragment` can request a search, which will result in a `SearchFragment` being placed on the back stack:

```
package com.commonsware.android.room.fts;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.text.TextUtils;

public class MainActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentById(android.R.id.content)==null) {
      getSupportFragmentManager().beginTransaction()
        .add(android.R.id.content, new BookFragment())
```

```
        .commit();
    }
  }

  public void search(String expr) {
    if (!TextUtils.isEmpty(expr)) {
      getSupportFragmentManager().beginTransaction()
        .replace(android.R.id.content, SearchFragment.newInstance(expr))
        .addToBackStack(null)
        .commit();
    }
  }
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/MainActivity.java)

`BookFragment` has an action bar with a `SearchView` in it:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/search"
    android:actionViewClass="android.widget.SearchView"
    android:icon="@drawable/ic_search_white_24dp"
    android:showAsAction="ifRoom|collapseActionView"
    android:title="@string/search">
  </item>

</menu>
```

(from TimeMachine/RoomFTS/app/src/main/res/menu/actions.xml)

`BookFragment` is that action bar with the `SearchView`, plus a `RecyclerView` to show the paragraphs of the book:

```
package com.commonsware.android.room.fts;

import android.arch.lifecycle.ViewModelProviders;
import android.arch.paging.PagedListAdapter;
import android.os.Bundle;
import android.support.annotation.Nullable;
import android.support.v7.util.DiffUtil;
import android.support.v7.widget.LinearLayoutManager;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.SearchView;
import android.widget.TextView;
```

```
public class BookFragment extends RecyclerViewFragment implements
  SearchView.OnQueryTextListener, SearchView.OnCloseListener {
  private SearchView sv=null;

  @Override
  public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setHasOptionsMenu(true);
  }

  @Override
  public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    setLayoutManager(new LinearLayoutManager(getActivity()));

    BookViewModel vm=ViewModelProviders.of(this).get(BookViewModel.class);
    final ParagraphAdapter adapter=new ParagraphAdapter(getActivity().getLayoutInflater());

    vm.paragraphs.observe(this, adapter::submitList);

    setAdapter(adapter);
  }

  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.actions, menu);

    configureSearchView(menu);

    super.onCreateOptionsMenu(menu, inflater);
  }

  @Override
  public boolean onQueryTextChange(String newText) {
    return false;
  }

  @Override
  public boolean onQueryTextSubmit(String query) {
    search(query);

    return true;
  }

  @Override
  public boolean onClose() {
    return true;
  }

  private void configureSearchView(Menu menu) {
    MenuItem search=menu.findItem(R.id.search);

    sv=(SearchView)search.getActionView();
    sv.setOnQueryTextListener(this);
    sv.setOnCloseListener(this);
    sv.setSubmitButtonEnabled(true);
    sv.setIconifiedByDefault(true);
  }
```

```
private void search(String expr) {
  ((MainActivity)getActivity()).search(expr);
}

private static class ParagraphAdapter extends PagedListAdapter<ParagraphEntity, RowHolder> {
  private final LayoutInflater inflater;

  ParagraphAdapter(LayoutInflater inflater) {
    super(PARA_DIFF);
    this.inflater=inflater;
  }

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new RowHolder(inflater.inflate(R.layout.row, parent, false)));
  }

  @Override
  public void onBindViewHolder(RowHolder holder, int position) {
    ParagraphEntity paragraph=getItem(position);

    if (paragraph==null) {
      holder.clear();
    }
    else {
      holder.bind(paragraph);
    }
  }
}

private static class RowHolder extends RecyclerView.ViewHolder {
  private final TextView prose;

  RowHolder(View itemView) {
    super(itemView);

    prose=itemView.findViewById(R.id.prose);
  }

  void bind(ParagraphEntity paragraph) {
    prose.setText(paragraph.prose);
  }

  void clear() {
    prose.setText(null);
  }
}

static final DiffUtil.ItemCallback<ParagraphEntity> PARA_DIFF=
  new DiffUtil.ItemCallback<ParagraphEntity>() {
    @Override
    public boolean areItemsTheSame(ParagraphEntity oldItem,
                                   ParagraphEntity newItem) {
      return oldItem.sequence==newItem.sequence;
    }

    @Override
    public boolean areContentsTheSame(ParagraphEntity oldItem,
                                      ParagraphEntity newItem) {
```

```
        return oldItem.prose.equals(newItem.prose);
      }
  };
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/BookFragment.java)

Of note:

- `onViewCreated()` observes the `LiveData` for our `PagedList` of paragraphs, routing the list to `submitList()` on the `PagedListAdapter`, to show those paragraphs as they become available
- When the user submits a search in the `SearchView`, `BookFragment` calls `search()` on the `MainActivity`, to bring up the `SearchFragment`

`SearchFragment` has a similar structure, just without the `SearchView`:

```
package com.commonsware.android.room.fts;

import android.arch.lifecycle.ViewModelProviders;
import android.arch.paging.PagedListAdapter;
import android.os.Bundle;
import android.support.v7.util.DiffUtil;
import android.support.v7.widget.DividerItemDecoration;
import android.support.v7.widget.LinearLayoutManager;
import android.support.v7.widget.RecyclerView;
import android.text.Html;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class SearchFragment extends RecyclerViewFragment {
  private static final String ARG_EXPR="expr";

  static SearchFragment newInstance(String expr) {
    SearchFragment result=new SearchFragment();
    Bundle args=new Bundle();

    args.putString(ARG_EXPR, expr);
    result.setArguments(args);

    return result;
  }

  @Override
  public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    setLayoutManager(new LinearLayoutManager(getActivity()));
    getRecyclerView()
      .addItemDecoration(new DividerItemDecoration(getActivity(),
        LinearLayoutManager.VERTICAL));
```

```
  SearchViewModel vm=ViewModelProviders.of(this).get(SearchViewModel.class);
  BookSearchAdapter adapter=
    new BookSearchAdapter(getActivity().getLayoutInflater());

  vm.search(getArguments().getString(ARG_EXPR))
    .observe(this, adapter::submitList);

  setAdapter(adapter);
}

private static class BookSearchAdapter extends PagedListAdapter<BookSearchResult, RowHolder> {
  private final LayoutInflater inflater;

  BookSearchAdapter(LayoutInflater inflater) {
    super(SEARCH_DIFF);
    this.inflater=inflater;
  }

  @Override
  public RowHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    return(new RowHolder(inflater.inflate(R.layout.row, parent, false)));
  }

  @Override
  public void onBindViewHolder(RowHolder holder, int position) {
    BookSearchResult result=getItem(position);

    if (result==null) {
      holder.clear();
    }
    else {
      holder.bind(result);
    }
  }
}

private static class RowHolder extends RecyclerView.ViewHolder {
  private final TextView prose;

  RowHolder(View itemView) {
    super(itemView);

    prose=itemView.findViewById(R.id.prose);
  }

  void bind(BookSearchResult result) {
    prose.setText(Html.fromHtml(result.snippet));
  }

  void clear() {
    prose.setText(null);
  }
}

static final DiffUtil.ItemCallback<BookSearchResult> SEARCH_DIFF=
  new DiffUtil.ItemCallback<BookSearchResult>() {
    @Override
    public boolean areItemsTheSame(BookSearchResult oldItem,
                                   BookSearchResult newItem) {
```

```
      return oldItem==newItem;
    }

    @Override
    public boolean areContentsTheSame(BookSearchResult oldItem,
                                      BookSearchResult newItem) {
      return oldItem.snippet.equals(newItem.snippet);
    }
  };
}
```

(from TimeMachine/RoomFTS/app/src/main/java/com/commonsware/android/room/fts/SearchFragment.java)

In its `onViewCreated()`, `SearchFragment` gets the search expression out of the arguments `Bundle` (placed there via `newInstance()`) and passes that to the `search()` method on the `SearchViewModel`, then observes the results and routes them to the `PagedListAdapter`.

The Results

When initially launched, you see the first few paragraphs of the book:

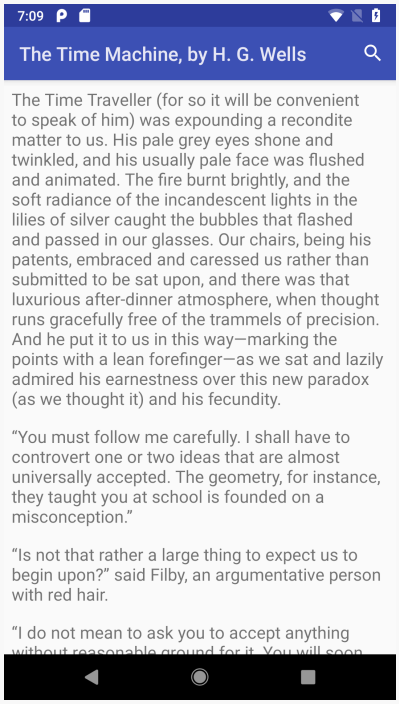


Figure 20: BookFragment, As Initially Launched

If the user taps on the search icon and types in a search:

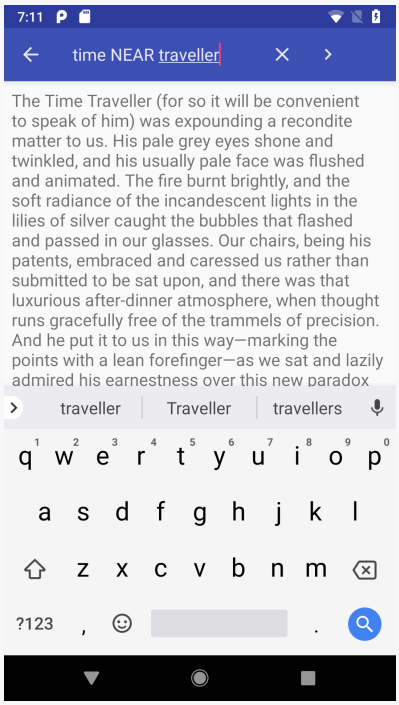


Figure 21: BookFragment, with Search Expression

...then submits it, the search results are shown in a `SearchFragment`:

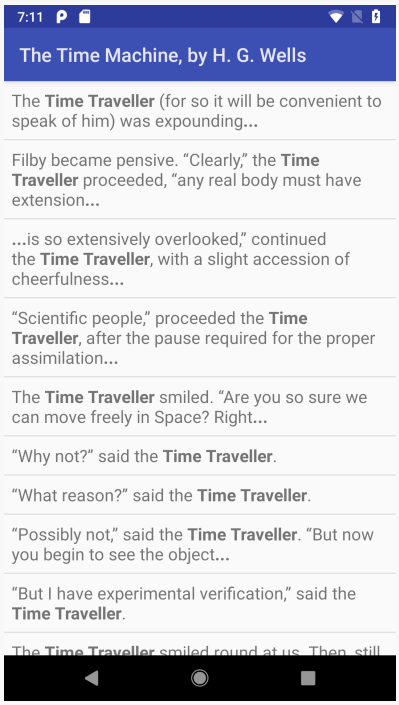


Figure 22: BookFragmSearchFragmentent, with Search Results

The search expressions can be simple words or anything supported by SQLite's FTS query syntax, including AND/OR/NOT/NEAR operators.

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 is different behavior in SQLite.

Value	Meaning
`OnConflictStrategy.ABORT`	Cancel this statement but preserve 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

However, they may *not* wind up with different behavior in Room, due to the way that Room works with SQLite.

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

Neither `@Insert` nor `@Update` will affect multiple rows in a single SQL statement. Even if your method accepts multiple entities (via varargs, `List`, etc.), they will each be processed in separate SQL statements, wrapped in a transaction. Room will roll back the transaction when it encounters the `SQLiteConstraintException`. As a result, `FAIL` will behave akin to `ABORT` when used with `@Insert` and `@Update`.

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.

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

Frankly, it does not work very well. The Room-generated code does not expect the transaction to be rolled back, and so its call to `endTransaction()` fails with an obscure `SQLiteException`.

Since `ABORT` and `FAIL` also have the net effect in Room of rolling back the transaction, they are better choices.

What Should You Use with Room?

`@Insert(onConflict = OnConflictStrategy.REPLACE)` may have its uses, particularly for cases where you are trying to use non-generated primary keys (e.g.,

"natural" keys).

Beyond that, `ABORT`, `FAIL`, and `ROLLBACK` all have the effect in Room of rolling back your requested insert or update. The first two throw a `SQLiteConstraintException`, which is a better exception than the `SQLiteException` that you get from `ROLLBACK`. And, since `ABORT` happens to be the default strategy, sticking with the default may well be your best option.

Configuring SQLite Beyond Room

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. For example, you cannot readily add full-text searching yourself, as that requires particular options in the `CREATE TABLE` statement.

Anything that lies outside of Room, though, is fair game, though you have to resort to classic SQLite approaches to make it work.

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 `CityPop/RoomPragma` sample project. `RoomDatabase.Callback` was introduced in the chapter on the support database API.

The `create()` 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:

```
static CityDatabase create(Context ctxt, final boolean applyPragmas) {
  RoomDatabase.Builder<CityDatabase> b=
    Room.databaseBuilder(ctxt.getApplicationContext(), CityDatabase.class,
      DB_NAME);

  if (applyPragmas) {
    b.addCallback(new Callback() {
      @Override
      public void onCreate(@NonNull SupportSQLiteDatabase db) {
        super.onCreate(db);
```

```
        db.query("PRAGMA journal_mode = MEMORY");
      }

      @Override
      public void onOpen(@NonNull SupportSQLiteDatabase db) {
        super.onOpen(db);

        db.query("PRAGMA synchronous = OFF");
      }
    });
  }

  return(b.build());
}
```

(from CityPop/RoomPragma/app/src/main/java/com/commonsware/android/citypop/CityDatabase.java)

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.

And, in truth, there does seem to be a significant performance gain:

Scenario	Use the `PRAGMA`s?	Time (milliseconds)
Inserting 1,063 cities via individual `insert()` calls	No	18,766
Inserting 1,063 cities via individual `insert()` calls	Yes	1,331
Inserting 1,063 cities in a single `insert()` call	No	402
Inserting 1,063 cities in a single `insert()` call	Yes	126

(tests conducted on a Google Pixel)

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.