6

Using the Room Database and Testing

Android applications can benefit significantly from storing data locally. The Room persistence library harnesses the power of SQLite. In particular, Room offers excellent benefits for Android developers. Furthermore, Room offers offline support, and the data is stored locally. In this chapter, we will learn how to implement Room, a Jetpack Library.

In this chapter, we'll be covering the following recipes:

- Implementing Room in your applications
- Implementing Dependency Injection in Room
- Supporting multiple entities
- Migrating existing SQL database to Room
- Testing your local database

It is also important to mention there are a couple more libraries are used with Room – for example, RxJava and Paging integration. In this chapter, we will not focus on them but instead on how you can utilize Room to build modern Android apps.

Technical requirements

The complete source code for this chapter can be found at **https://github.com/PacktPublishing/Modern-Android-13-Development-Cookbook/tree/main/chapter_six**.

Implementing Room in your applications

Room is an object-relational mapping library used in Android data persistence and is the recommended data persistence in

Modern Android Development. In addition, it is effortless to use, understand and maintain, and harnesses the powers of **`SQLiteDatabase`**, it also helps reduce boilerplate code, an issue many developers experience when using SQLite. Writing tests is also very straightforward and easy to understand.

The most notable advantage of Room is that it is easy to integrate with other architecture components and gives developers runtime compile checks – that is, Room will complain if you make an error or change your schema without migrating, which is practical and helps reduce crashes.

How to do it...

Let's go ahead and create a new empty compose project and call it **`RoomExample`**. In our example project, we will create a form intake from users; this is where users can save their first and last names, date of birth, gender, the city they live in, and their profession.

We will save our user data in our Room database, and then later inspect whether the elements we inserted were saved in our database and display the data on the screen:

1. In our newly created project, let's go ahead and delete the unnecessary wanted code – that is, **`Greeting(name: String)`**, which comes with all empty Compose projects. Keep the preview function, since we will use it to view the screen we create.
2. Now, let's go on and add the needed dependencies for Room and sync the project. We will touch on dependency management using **`buildSrc`** in *Chapter 12*, *Android Studio Tips and Tricks to Help You during Development*. You can find the latest version of Room at **https://developer.android.com/jetpack/androidx/releases/room**; we will add **`kapt`**, which stands for **Kotlin Annotation Processing Tool**, to enable us to use the Java annotation processor with the Kotlin code:

```
dependencies {
implementation "androidx.Room:Room-runtime:2.x.x"
kapt "androidx.Room:Room-compiler:2.x.x"
}
//include kapt on your plugins
plugins {
```

```
    id 'kotlin-kapt'
}
```

3. Create a new package and call it **`data`**. Inside **`data`**, create a new Kotlin class and call it **`UserInformationModel()`**. A data class is used to hold data only – in our case, the type of data that we will collect from users will be the first name, last name, date of birth, and so on.
4. By using Room, we use the **`@Entity`** annotation to give our model a table name; hence, in our newly created **`UserInformation`** class, let's go ahead and add the **`@Entity`** annotation and call our table user information:

```
@Entity(tableName = "user_information")
data class UserInformationModel(
    val id: Int = 0,
    val firstName: String,
    val lastName: String,
    val dateOfBirth: Int,
    val gender: String,
    val city: String,
    val profession: String
)
```

5. Next, as in all databases, we need to define a primary key for our database. Hence, in our ID, we will add the **`@PrimaryKey`** annotation to tell Room that this is our primary key, and it should be autogenerated. If you don't wish to autogenerate, you can set the Boolean to **`false`**, but this might not be a good idea, due to conflicts that might arise later in your database:

```
@PrimaryKey(autoGenerate = true)
```

6. Now, you should have an entity with a table name, a primary key, and your data types:

```
import androidx.Room.Entity
import androidx.Room.PrimaryKey
@Entity(tableName = "user_information")
data class UserInformationModel(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    ...
)
```

Inside our data package, let us go ahead and create a new package and call it **`DAO`**, which means **data accessible object**. Once that is done, create a new interface and call it **`UserInformationDao`**; this interface will hold the **create**, **read**,

update, and **delete** (**CRUD**) functionality – that is, **update**, **insert**, **delete**, and **query**.

We must also annotate our interface with **`@Dao`** to tell Room that this is our DAO. We use **`OnConflictStrategy.REPLACE`** on the update and **`Insert`** functions to help us with a case where we might encounter conflicts in our database. **`OnConflictStrategy`**, in this case, means that if **`Insert`** has the same ID, it will replace that data with a particular ID:

```
private const val DEFAULT_USER_ID = 0
@Dao
interface UserInformationDao {
    @Query("SELECT * FROM user_information")
    fun getUsersInformation():
        Flow<List<UserInformationModel>>
    @Query("SELECT * FROM user_information WHERE id =
        :userId")
    fun loadAllUserInformation(userId: Int =
        DEFAULT_USER_ID): Flow<UserInformationModel>
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUserInformation(userInformation:
        UserInformationModel)
    @Update(onConflict = OnConflictStrategy.REPLACE)
    suspend fun updateUserInformation(userInformation:
        UserInformationModel)
    @Delete
    suspend fun deleteUserInformation(userInformation:
        UserInformationModel)
}
```

7. Now that we have our entity and DAO, we will finally create the **`Database`** class, which extends **`RoomDatabase()`**. In this class, we will use the **`@Database`** annotation, pass in the entity that we created, which is the **`UserInformation`** entity, and give our database a version name, which currently is **`one`**. We will also specify whether our database schema should be exported or not. So, let's go ahead and create the **`Database`** abstract class:

```
@Database(entities = [UserInformation::class], version = 1, exportSchema = false)
abstract class UserInformationDatabase : RoomDatabase() {
    abstract fun userInformationDao():
        UserInformationDao
}
```

8. Finally, we have **`Room`** set up and ready. Now, we need to add Dependency Injection and our user interface; you can

find the code in the *Technical requirements* section. Also, the UI is quite basic at this stage; you can make it a challenge to improve it, as this sample project is just for demonstration purposes.

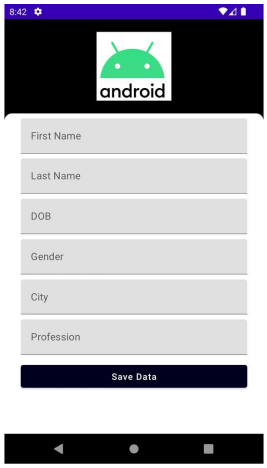

Figure 6.1 – The app's UI

How it works...

The Modern Android Development Room library has three significant components of the Room database:

- The entity
- The DAO
- The database

`Entity` is a table within the database. Room generates a table for each class that has the **`@Entity`** annotation; if you have used Java before, you can think of the entity as a **plain old Java object** (**POJO**). The entity classes tend to be minor, don't contain any logic, and only hold the data type for the object.

Some significant annotations that map the tables in the database are the foreign keys, indices, primary keys, and table names. There are other essential annotations, such as **`ColumnInfo`**, which gives column information, and **`Ignore`**, which, if used, whichever data you wish to ignore will not be persisted by Room.

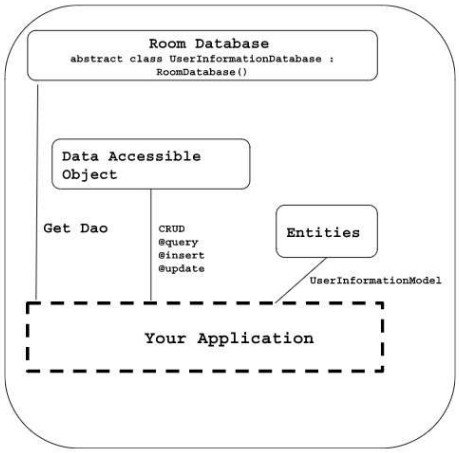

Figure 6.2 – Room DAO

@DAO defines the functions that access the database. Think of it like CRUD; if you used SQLite before Room, this is similar to using the cursor objects. Finally, **@Database** contains the database functions and serves as the main entry point for any underlying connection to our application's relational data.

If you need to use this, you annotate with **@Database**, as we did in our database class. In addition, this class extends **RoomDatabase** and includes the list of entities we create. It also contains the abstract method that we create, has no arguments, and returns the class that we annotated with **@Dao**. We run the database by calling **Room.databaseBuilder()**.

Implementing Dependency Injection in Room

As with other recipes, Dependency Injection is vital, and in this recipe, we will walk through how we can inject our **DatabaseModule** and provide the Room database where it is needed.

Getting ready

You will need to have prior knowledge of how Hilt works to be able to follow this recipe step by step.

How to do it...

Open the **RoomExample** project and add Hilt, which is what we will use for Dependency Injection. In *Chapter 3*, *Handling the*

UI State in Jetpack Compose and Using Hilt, we covered Hilt, so we will not discuss it here but just show you how you can use it with Room:

1. Open your project and add the necessary Hilt dependency. See *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*, if you need help setting up Hilt or visit **https://dagger.dev/hilt/**.
2. Next, let's go ahead and add our **`@HiltAndroidApp`** class, and in the **`Manifest`** folder, add the name of our **`HiltAndroidApp`**, in our case, **`UserInformation`**:

```
@HiltAndroidApp
class UserInformation : Application()
<application
    android:allowBackup="true"
    android:name=".UserInformation"
    tools:targetApi="33">
...
```

3. Now that we have Dependency Injection, let's go ahead and add **`@AndroidEntryPoint`** in the **`MainActivity`** class, and in our project, let's create a new package and call it **`di`**. Inside, we will create a new class, **`DatabaseModule`**, and add our functionalities.
4. In **`DatabaseModule`**, let's go ahead and create a **`provide-Database()`** function, where we will return the **`Room`** object, add the database name, and ensure we build our database:

```
@Module
@InstallIn(SingletonComponent::class)
class DataBaseModule {
    @Singleton
    @Provides
    fun provideDatabase(@ApplicationContext context:
    Context): UserInformationDatabase {
        return Room.databaseBuilder(
            context,
            UserInformationDatabase::class.java,
            "user_information.db"
        ).build()
    }
    @Singleton
    @Provides
    fun provideUserInformationDao(
    userInformationDatabase: UserInformationDatabase):
    UserInformationDao {
        return
```

```
            userInformationDatabase.userInformationDao()
        }
    }
```

5. Now that we have our Dependency Injection database module set up, we can now start adding the service, which are functions that will help us add user information to the database and get user information from the database. So, let us go ahead and create a new package called **`service`**. Inside the package, create a new interface, **`UserInfoService`**, and add the two aforementioned functions:

```
interface UserInfoService {
    fun getUserInformationFromDB():
        Flow<UserInformation>
    suspend fun addUserInformationInDB(
        userInformation: UserInformation)
}
```

6. Since **`UserInfoService`** is an interface, we will need to implement the functionalities in our **`Impl`** class, so let us now go ahead and create a new class called **`UserInfoServiceImpl`** and a singleton class, and then implement the interface:

```
@Singleton
class UserInfoServiceImpl() : UserInfoService {
    override fun getUserInformationFromDB():
        Flow<UserInformation> {
            TODO("Not yet implemented")
    }
    override suspend fun addUserInformationInDB(
        userInformation: UserInformation) {
            TODO("Not yet implemented")
    }
}
```

7. We will need to inject our constructor and pass **`UserInformationDao()`**, since we will use the insert function to insert the user data:

```
class UserInfoServiceImpl @Inject constructor(
private val userInformationDao: UserInformationDao
): UserInfoService
```

8. Now, we need to add code in our functions that have the TODO in them. Let's go ahead and see the user information first. Using **`userInformationDao`**, we will call the insert function to tell Room that we want to insert this user information:

```
override suspend fun addUserInformationInDB(
userInformation: UserInformation) {
    userInformationDao.insertUserInformation(
        UserInformation(
            firstName = userInformation.firstName,
            lastName = userInformation.lastName,
            dateOfBirth = userInformation.dateOfBirth,
            gender = userInformation.gender,
            city = userInformation.city,
            profession = userInformation.profession
        )
    )
}
```

9. Then, we need to get the user information from the database; this will visualize a user's data on the screen:

```
override fun getUserInformationFromDB() =
    userInformationDao.getUsersInformation().filter {
        information -> information.isNotEmpty()
    }.flatMapConcat {
        userInformationDao.loadAllUserInformation()
            .map { userInfo ->
                UserInfo(
                    id = userInfo.id,
                    firstName = userInfo.firstName,
                    lastName = userInfo.lastName,
                    dateOfBirth =
                        userInfo.dateOfBirth,
                    gender = userInfo.gender,
                    city = userInfo.city,
                    profession = userInfo.profession
                )
            }
    }
```

10. Finally, we need to ensure that we provide the implementation through Dependency Injection, so let's now go ahead and add the preceding code, then clean the project, run it, and ensure that everything works as expected:

```
@Module
@InstallIn(SingletonComponent::class)
abstract class UserInfoServiceModule {
    @Singleton
    @Binds
    abstract fun bindUserService(
        userInfoServiceImpl: UserInfoServiceImpl):
            UserInfoService
}
```

11. Once you run the project, you should be able to see it launch without issue. We will go ahead and add a function in our **`ViewModel`** to insert the data in our database; the **`ViewModel`** will be used in the views that we created:

```
@HiltViewModel
class UserInfoViewModel @Inject constructor(
    private val userInfoService: UserInfoService
) : ViewModel() {
    fun saveUserInformationData(userInfo: UserInfo) {
        viewModelScope.launch {
            userInfoService.addUserInformationInDB(
                userInfo)
        }
    }
}
```

12. We can now inspect the database and see whether it was created correctly. Run the app, and once it's ready in the IDE, click **App Inspection**, as shown in *Figure 6.3*. You should be able to open the Database Inspector.

Figure 6.3 – App Inspection

13. Once the Database Inspector is loaded, you should be able to select the currently running Android Emulator, as shown in *Figure 6.4*:

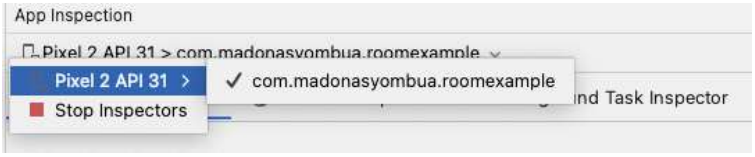

Figure 6.4 – The selected app for app inspection

14. In *Figure 6.5*, you can see the Database Inspector open and our database.

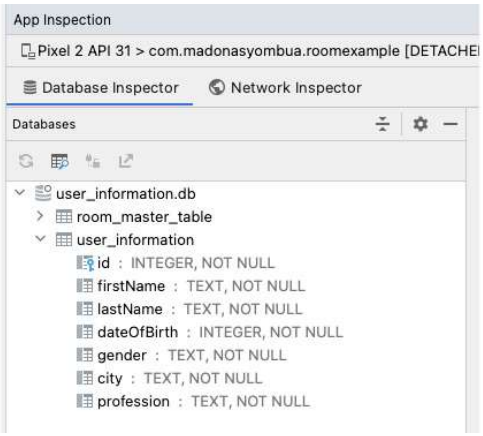

Figure 6.5 – Our user_information database

15. In *Figure 6.6*, you can see that the data we inserted is displayed, which means our insert function works as expected.

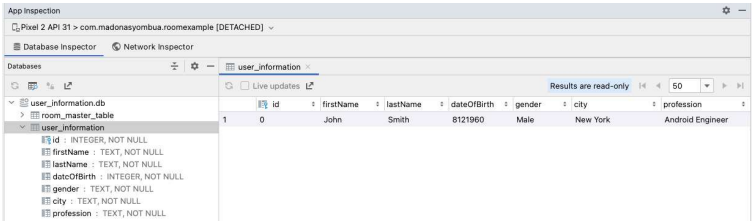

Figure 6.6 – Our database

How it works...

In this recipe, we opted to use Dependency Injection to supply the needed dependencies to specific classes. We covered in depth what Dependency Injection is in previous chapters, so we will not explain it again in this recipe but, instead, talk about the modules we created.

We used the **`@Singleton`** annotation in Hilt to indicate that **`provideDatabase`**, which provides an instance of Room, should be created only once during the lifetime of our application, and that this instance should be shared across all the components that depend on it. In addition, when you annotate a class or a binding method with **`@Singleton`**, Hilt ensures that only one instance of that class or object is created and all the components that need that object will receive the same instance.

It's important to also know that when we use **`@Singleton`** in Hilt, it is not the same as the Singleton pattern in software design, which can easily be a source of confusion. Hilt's

@Singleton only guarantees that one instance of a class will be created within the context of a particular component hierarchy.

In our project, we created **DatabaseModule()** and **UserInfoServiceModule()**. In the **DatabaseModule()** class, we have two functions, **provideDatabase** and **provideUserInformationDao**. The first function, **provideDatabase**, returns the **UserInformationDatabase** Room instance, where we get to create the database and build it:

```
fun provideDatabase(@ApplicationContext context: Context): UserInformationDatabase {
    return Room
        .databaseBuilder(context,
            UserInformationDatabase::class.java,
            "user_information.db")
        .build()
}
```

In **provideUserInformationDao**, we pass **UserInformationDatabase** in the constructor and return the **UserInformationDao** abstract class:

```
fun provideUserInformationDao(userInformationDatabase: UserInformationDatabase): UserInformationDao
    return userInformationDatabase.userInformationDao()
}
```

IMPORTANT NOTE

If you want to lose existing data when you are migrating or if your migration path is missing, you can use the **.fallbackToDestructiveMigration()** *function when creating the database.*

See also

There is more to learn in Room, and this recipe has only given you a brief overview of what you can do with it. You can learn more by following the link at **https://developer.android.com/reference/androidx/room/package-summary**.

Supporting multiple entities in Room

In this recipe, you will learn how to handle multiple entities in Room. This is useful whenever you have a big project that needs a different data input. An excellent example that we can work with is a budgeting app.

To support multiple entities in Room, you need to define multiple classes that represent your database tables. Each class should have its own annotations and fields that correspond to columns in a table. For instance, a budgeting app might need different types of models, such as the following:

- **`BudgetData`**
- **`ExpenseData`**
- **`ExpenseItem`**

Hence, having multiple entities is sometimes necessary, and knowing how to handle that comes in handy.

Getting ready

To follow along with this recipe, you must have completed the previous recipe.

How to do it ...

You can use any project of your choosing to implement the topics discussed in this recipe. In addition, you can use this example in your pre-existing project to implement the topic.

1. In **`RoomExample`**, you can add more functionality to the app and try to add more entities, but for this project, let's go ahead and show how you can handle multiple entities in Room.
2. For this example, we will use the sample budgeting App we introduced in an earlier chapter, and since we are working with entities, this will be easier to follow. Let's create a new entity and call it **`BudgetData`**; the budget data class might have several fields, such as **`budgetName`**, **`budgetAmount`**, **`expenses`**, **`startDate`**, **`endDate`**, **`notify`**, **`currency`**, and **`totalExpenses`**; hence, our **`BudgetData`** data class will look like this:

```
@Entity(tableName = "budgets")
data class BudgetData(
    @PrimaryKey(autoGenerate = true)
```

```
    var id: Int = 0,
    var budgetName: String = "",
    var budgetAmount: Double = 0.0,
    var expenses: String = "",
    var startDate: String = "",
    var endDate: String = "",
    var notify: Int = 0,
    var currency: String = "",
    var totalExpenses: Double
)
```

3. Let's go ahead and add two more entities. First, we will add **`ExpenseData`**, which might have the following fields and types:

```
@Entity(tableName = "expenses")
data class ExpenseData(
    @PrimaryKey(autoGenerate = true)
    var id: Int = 0,
    var expenseName: String = "",
    var expenseType: String = "",
    var expenseAmount: Double = 0.0,
    @ColumnInfo(name = "updated_at")
    var expenseDate: String = "",
    var note: String = "",
    var currency: String = ""
)
```

4. Then, let's add **`ExpenseItem`**, which might consist of the following fields:

```
@Entity(tableName = "items")
Data class ExpenseItem(
    @PrimaryKey(autoGenerate = true)
    private var _id: Int
    val name: String
    var type: String?
    val imageContentId: Int
    val colorContentId: Int)
```

5. As you can see, we have three entities; based on these entities, you should create different DAOs for each one:

```
abstract class AppDatabase : RoomDatabase() {
    abstract fun budgetDao(): BudgetDao
    abstract fun itemDao(): ItemDao
    abstract fun expenseDao(): ExpenseDao
}
```

6. At the top of the **`AppDatabase`** abstract class, we will annotate it with **`@Database`** and then pass it to all our entities:

```
@Database(
    entities = [ExpenseItem::class, BudgetData::class,
```

```
            ExpenseData::class],
        version = 1
    )
    @TypeConverters(DateConverter::class)
    abstract class AppDatabase : RoomDatabase() {
        abstract fun budgetDao(): BudgetDao
        abstract fun itemDao(): ItemDao
        abstract fun expenseDao(): ExpenseDao
    }
```

7. You can also use embedded objects; the **`@Embedded`** annotation includes nested or related entities within an entity. It allows you to represent the relationship between entities by embedding one or more related entities in the parent entity:

```
data class ExpenseItem(
...
    @Embedded val tasks: Tasks
  )
data class Tasks(...)
```

In our preceding example, we have annotated the tasks property in the **`ExpenseItem`** entity with the **`@Embedded`** annotation. This tells Room to include the fields of the **`Tasks`** data class within the **`ExpenseItem`** table, rather than creating a separate table for our **`ExpenseItem`** entity.

8. Then, the Tasks data class can have the **`description`**, **`priority`**, **`updatedAt`**, and ID:

```
data class Tasks (
    @PrimaryKey(autoGenerate = true)
    var id = 0
    var description: String
    var priority: Int
    @ColumnInfo(name = "updated_at")
    var updatedAt: Date)
```

Hence, the table representing the **`ExpenseItem`** object will contain additional columns with the newly added fields.

That's it; once you declare the entities in the database and pass them as required, you will have supported multiples entities in your **`Database`**.

IMPORTANT NOTE

If your entity has multiple embedded fields of the same type, you can keep each column unique by setting the **`Prefix`** *property; then, Room will add the provided values to the beginning of each column name in the embedded object. Find out more at* **https://developer.android.com/**.

How it works...

According to the rules in Room, you can define an entity relationship in three different ways.

- One-to-many relationships or many-to-one relationships
- One-to-one relationships
- Many-to-many relationships

As you have already seen, using entities in one class makes it manageable and easily trackable; hence, this is an excellent solution for Android engineers. A notable annotation is **`@Relation`**, which specifies where you create an object that shows the relationship between your entities.

There's more...

There is more to learn in Room – for instance, defining relationships between objects, writing asynchronous data accessible object queries, and referencing complex data. It is fair to say we cannot cover everything in just one chapter but offer some guidance to help you navigate building Modern Android applications. For more on Room, visit **https://developer.android.com/training/data-storage/room**.

Migrating an existing SQL database to room

As we mentioned earlier, Room does harness the power of SQLite, and because many applications still use legacy, you might find applications still using SQL and be wondering how you can migrate to Room and utilize the latest Room features.

In this recipe, we will cover the migration of an existing SQL database to Room with step-by-step examples. Furthermore,

Room offers an abstraction layer to help with SQLite migrations – that is, by offering the **`Migration`** class to developers.

How to do it...

Because we did not create a new SQLite database example, since that is not necessary, we will try to emulate a scenario with a dummy sample SQLite database and showcase how you can migrate your existing SQLite database to Room:

1. Since we will be adding Room in an existing SQLite project, you will need to ensure you add the required dependencies. To set this up, refer to the *Implementing Room in your applications* recipe.
2. Next, you will need to go ahead and create a new DAO and entity, since Room requires it. Hence, in this set following the first Room recipe, you can update the model classes to entities. This is pretty straightforward, since mostly what you will do is annotate the classes with **`@Entity`** and use the table **`Names`** property to set the name of the table.
3. You must also add **`@PrimaryKey`** and **`@ColumnInfo`** annotations for your entity classes. Here is a sample SQLite database:

```
fun onCreate(db: SQLiteDatabase) {
    // Create a String that contains the SQL statement
       to create the items table
    val SQL_CREATE_ITEMS_TABLE =(
            "CREATE TABLE " + ItemsContract.ItemsEntry
                    .TABLE_NAME.toString() + " ("
                    + ItemsContract.ItemsEntry.
                    _Id.toString()
                    + " INTEGER PRIMARY KEY
                    AUTOINCREMENT, "
                    + ItemsContract.ItemsEntry
                    .COLUMN_ITEM_NAME.toString()
                    + " TEXT NOT NULL, "
                    + ItemsContract.ItemsEntry
                    .COLUMN_ITEM_TYPE.toString()
                    + " TEXT NOT NULL, "
                    + ItemsContract.ItemsEntry
                    .COLUMN_ITEM_LOGO.toString()
                    + " INTEGER NOT NULL DEFAULT 0, "
                    + ItemsContract.ItemsEntry
                    .COLUMN_ITEM_COLOR.toString()
                    + " INTEGER NOT NULL DEFAULT 0, "
                    + ItemsContract.ItemsEntry
                    .COLUMN_ITEM_CREATED_DATE
```

```
                        .toString() + " DATE NOT NULL
                        DEFAULT CURRENT_TIMESTAMP);")
        // Execute the SQL statement
        db.execSQL(SQL_CREATE_ITEMS_TABLE)
    }
```

However, Room has simplified the process, and we no longer need to create **`Contracts`**. **`Contracts`** in Android are a way for developers to define and enforce a set of rules for accessing data within an application. These contracts typically define the structure and schema of the database tables and the expected data types and formats for the data within them. In the case of SQLite on Android, contracts are often used to define the tables and columns of the database, as well as any constraints or relationships between them.

4. Once we have created all our needed entities and DAOs, we can go ahead and create the database. As we saw in the *Implementing Room in your Applications* recipe, we can add all our entities in the **`@Database`** annotation, and since we are in the first **`(1)`** version, we can increment the version to (**`2`**):

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database:
    SupportSQLiteDatabase) {
        //alter items table
        database.execSQL("CREATE TABLE new_items (_id
            INTEGER PRIMARY KEY AUTOINCREMENT NOT
            NULL, name TEXT NOT NULL, type TEXT,
            imageContentId INTEGER NOT NULL,
            colorContentId INTEGER NOT NULL)")
        database.execSQL("INSERT INTO new_items
            (_id,name,type,imageContentId,
            colorContentId)Select_id,name,type,
            imageContentId, colorContentId FROM
            items")
        database.execSQL("DROP TABLE items")
        database.execSQL("ALTER TABLE new_items RENAME
            TO items")
    }
```

5. Then, the important part is ensuring we call **`build()`** to the Room database:

```
Room.databaseBuilder(
    androidContext(),
    AppDatabase::class.java, "budget.db"
)
```

```
        .addCallback(object : RoomDatabase.Callback() {
            override fun
            onCreate(db:SupportSQLiteDatabase){
                super.onCreate(db)
            }
        })
        .addMigrations(MIGRATION_1_2)
        .build()
```

6. Once your data layer starts using Room, you can officially replace all the **`Cursor`** and **`ContentValue`** code with the DAO calls. In our **`AppDatabase`** class, we have our entities, and our class extends **`RoomDatabase()`**:

```
@Database(
    entities = [<List of entities>],
    version = 2
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun itemDao(): ItemDao
}
```

Because Room offers runtime errors, if any error occurs, you will be notified in Logcat.

7. It is fair to say that not everything can be covered in one recipe because SQLite does require a lot of code to set up – for instance, to create queries and handle the cursors – but Room helps to speed these processes up, which is why it is highly recommended.

How it works...

As recommended earlier, migrating a complex database might be hectic and require caution, since it can affect users if pushed to production without thorough testing. It is also highly recommended to use **`OpenHelper`**, exposed by **`RoomDatabase`**, for more straightforward or minimal changes to your database. Furthermore, it is worth mentioning that if you have any legacy code using SQLite, it will be written at a high level in Java, so working with a team to find a better solution for the migration is needed.

In your project, you must update the class that extends **`SQLiteOpenHelper`**. We use **`SupportSQLiteDatabase`** because we need to update the calls to get the writable and readable

database. This is a cleaner database abstraction class to insert and query the database.

IMPORTANT NOTE

It is important to note that it might be complicated to migrate to a complex database that has many tables and complex queries. However, if your database has minimal tables and no complex queries, migration can be done quickly with relatively small incremental changes in a feature branch. It might be helpful to download the app's database, and you can do so by visiting the following link: **https://developer.android.com/training/data-storage/room/testing-db#command-line**.

Testing your local database

So far, we have ensured that we write tests whenever necessary for our projects. We will now need to go ahead and write tests for our **`RoomExample`** project, since this is crucial, and you might be required to do so in a real-world scenario. Hence, in this recipe, we will look at a step-by-step guide on writing CRUD tests for our database.

Getting ready

You will need to open the **`RoomExample`** project to get started with this recipe.

How to do it...

Let's go ahead and first add all the needed Room testing dependencies, and then start writing our tests. For the Hilt test setup, refer to the *Technical requirements* section, where you can find all the required code:

1. You will need to add the following to your **`build.gradle`**:

```
androidTestImplementation "com.google.truth:truth:1.1.3"
androidTestImplementation "android.arch.core:core-testing:1.1.1"
```

2. After you have added the required dependencies inside the Android test, go ahead and create a new class, calling it **`UserInformationDBTest`**:

```
class UserInformationDBTest {...}
```

3. Before we can set up our **`@Before`** function, we will need to create two **`lateinit var`** instances, which we will initialize in our **`@Before`** function:

```
private lateinit var database: UserInformationDatabase
private lateinit var userInformationDao: UserInformationDao
```

4. Now, let us go ahead and set up our **`@Before`** function and create our database, using the in-memory database for testing purposes:

```
@Before
fun databaseCreated() {
    database = Room.inMemoryDatabaseBuilder(
        ApplicationProvider.getApplicationContext(),
            UserInformationDatabase::class.java
    )
        .allowMainThreadQueries()
        .build()
    userInformationDao = database.userInformationDao()
}
```

5. Since we are running and creating the database in memory, we will need to close it after it is done; hence, in our **`@After`** call, we will need to call **`close()`** on our database:

```
@After
fun closeDatabase() {
    database.close()
}
```

6. Now that our setup is complete, we will go ahead and start testing our CRUD – that is, inserting, deleting, and updating. Let's go ahead and create an insert test first:

```
@Test
fun insertUserInformationReturnsTrue() = runBlocking {
    val userOne = UserInformationModel(
        id = 1,
        firstName = "Michelle",
        lastName = "Smith",
        dateOfBirth = 9121990,
        gender = "Male",
        city = "New york",
        profession = "Software Engineer"
    )
    userInformationDao.insertUserInformation(userOne)
    val latch = CountDownLatch(1)
    val job = async(Dispatchers.IO) {
        userInformationDao.getUsersInformation()
            .collect {
```

```
                assertThat(it).contains(userOne)
                latch.countDown()
            }
        }
        latch.await()
        job.cancelAndJoin()
    }
```

7. Finally, let us add the **`delete`** function, and that will wrap up our testing Room for now:

```
    @Test
    fun deleteUserInformation() = runBlocking {
        val userOne = UserInformationModel(
            id = 1,
            firstName = "Michelle",
            lastName = "Smith",
            dateOfBirth = 9121990,
            gender = "Male",
            city = "New york",
            profession = "Software Engineer"
        )
        val userTwo = UserInformationModel(
            id = 2,
            firstName = "Mary",
            lastName = "Simba",
            dateOfBirth = 9121989,
            gender = "Female",
            city = "New york",
            profession = "Senior Android Engineer"
        )
        userInformationDao.insertUserInformation(userOne)
        userInformationDao.insertUserInformation(userTwo)
        userInformationDao.deleteUserInformation(userTwo)
        val latch = CountDownLatch(1)
        val job = async(Dispatchers.IO) {
            userInformationDao.loadAllUserInformation()
                .collect {
                    assertThat(it).doesNotContain(userTwo)
                    latch.countDown()
                }
        }
        latch.await()
        job.cancelAndJoin()
    }
```

8. When you run the test, they should all pass with a green check mark:

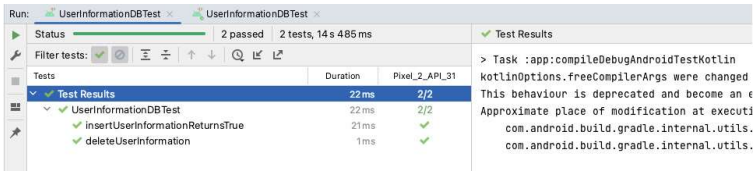

Figure 6.7 – Our tests passing

How it works...

You might have noticed we have used **`Truth`**, which is a testing framework that provides a fluent and expressive API to write assertions in tests. It is developed by Google, and some of the advantages of using **`Truth`** include readability, flexibility, and clear error messages. We can easily use a more like natural language constructs – for example, **`isEqualTo`** and **`shouldBe`** – which makes the test assertions more intuitive and readable for us developers.

When using the framework, you get a wide range of assertion methods that allow you to test a variety of conditions, including equality, order, and containment. It also allows you to define custom assertion methods, giving you more control over the behavior of your tests.

The **`@Before`** annotation ensures our **`databaseCreated()`** function is executed before each class. Our function then creates a database using **`Room.inMemoryDatabaseBuilder`**, which creates a database in **Random Access Memory** (**RAM**) instead of persistence storage. This means our database will be cleared once the process is killed; hence, in our **`@After`** call, we close the database:

```
@After
fun closeDatabase() {
    database.close()
}
```

As you might have seen, our tests are in **`AndroidTest`**, since we launch Room in the main thread and close it after we finish it. The test classes just test the DAO functions – that is, **`Update`**, **`Insert`**, **`Delete`**, and **`Query`**.