Advanced Mobile Application Development
Week 15: Room and SQLite

RDBMS

Relational databases (RDBMS) are a good choice to store structured data. They handle complex data with relationships and enforce data integrity. They are also a good fit for larger amounts of data that will be accessed and manipulated often.
In a RDBMS database data is organized in a collection of tables with defined relationships. This structure and organization is referred to as the database schema.
A table is organized into columns that represent what is stored in the table. Each row in a table contains one record of data.

SQL

Structured query language (SQL) is the standard language for accessing and manipulating relational databases.

- INSERT creates a new record
- DELETE deletes a record
- UPDATE modifies a record
- SELECT performs a query that returns one or more records
    - SELECT column_name FROM table_name
- * is the wildcard in SQL
- WHERE adds a conditional
- ORDER BY column_name orders the results
    - ASC ascending
    - DESC descending

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

The Room library includes convenience methods for many SQL commands.

SQLite

Android's SQLite database is an RDBMS well suited for persisting large amounts of structured data locally. Similar to internal storage, Android stores your database in your app's private folder and therefore is not accessible to other apps or the user.

Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data. That way, when the device cannot access the network, the user can still browse that content while they are offline. Any user-initiated content changes are then synced to the server after the device is back online.

Room

https://developer.android.com/training/data-storage/room/index.html
The Room library is one of the new architecture components that is included in Jetpack. It provides an abstraction layer over SQLite that makes it much easier to work with SQLite and allow for more robust database access while harnessing the full power of SQLite.

Room lets you define your database structure and manage the data at runtime using annotations that are interpreted at compile time.

The library helps you create a cache of your app's data on a device that's running your app. This cache, which serves as your app's single source of truth, allows users to view a consistent copy of key information within your app, regardless of whether users have an internet connection.

SQLite has some significant drawbacks that Room improves upon.

- You have to write a lot of boilerplate code
- You have to implement object mapping for every query you write.
- Difficult to implement database migrations
- Database operation on the main thread

Because Room takes care of these concerns for you, Google highly recommends using Room over using SQLite APIs directly.

There are 3 major components in Room:

Entity
https://developer.android.com/training/data-storage/room/defining-data
An entity represents a table in the database and is often just your data class with the following requirements:

- Annotated with @Entity
    - the default name of the table will match the name of the class
    - optionally use the tableName attribute to define the SQLite database table name
- @PrimaryKey defines the primary key for the table
    - each entity must have at least 1 field as a primary key
        - You can use composites
    - the autoGenerate property will automatically generate the primary key
        - this works for integers but not for Strings
- The class data members are used as the column names in the database
    - the default name of the column will match the name of the data member
    - use the @ColumnInfo annotation with the name attribute to a field to change the name of the column
- Room only uses one constructor and it must accept parameters for all of the class' data members
- You can use Type Converters for properties that Room and SQL don't know how to serialize such as a custom class, a list, or a date type

Data Access Objects (DAOs)
https://developer.android.com/training/data-storage/room/accessing-data
To access your app's data using the Room persistence library, you work with *data access objects*, or DAOs. Each DAO class contains the methods used for accessing the database. By accessing a database using a DAO class instead of query builders or direct queries, you can separate different components of your database architecture.
A DAO can be either an interface or an abstract class. Room creates each DAO implementation at compile time. It's common to have one DAO class for each entity class.

There are multiple convenience queries that you can represent using a DAO class. Room generates an implementation for these at compile time.

- @Insert inserts all parameters into the database in a single transaction.
    - If the @Insert method receives only 1 parameter, it can return a long, which is the new rowId for the inserted item.
    - If the parameter is an array or a collection, it should return long[] or List<Long> instead.
- @ Update modifies a set of entities, given as parameters, in the database. It uses a query that matches against the primary key of each entity.
    - For insert or update operations the property *onConflict* can be used to indicate how to handle a conflict while performing the operation. The strategies available to use are: REPLACE, ABORT, FAIL, IGNORE, and ROLLBACK.
        - @Update(onConflict = OnConflictStrategy.IGNORE)
- @ Delete removes a set of entities, given as parameters, from the database. It uses the primary keys to find the entities to delete.
- @Query is the main annotation used in DAO classes. It allows you to perform read/write operations on a database.
    - Each @Query method is verified at compile time, so if there is a problem with the query, a compilation error occurs instead of a runtime failure.

Room prevents database access on the main UI thread so the UI is not blocked so the DAO queries must be asynchronous. https://developer.android.com/training/data-storage/room/async-queries

- Asynchronous queries—queries that return instances of LiveData or Flowable—are exempt from this rule because they handle running the query asynchronously on a background thread when needed.
    - this only applies to database calls that return LiveData objects like @Query. You will still need to use a background thread for other calls
- For our asynchronous calls we'll be using Kotlin coroutines which basically are light weight threads that can be suspended and resumed without blocking a thread.
    - Suspending vs blocking
        - If you make a blocking call on the main thread's execution, you effectively freeze the UI
        - If you call a suspending function in the main UI thread it can be run on a different thread so it won't block the main UI thread.
        - (images) https://www.raywenderlich.com/1423941-kotlin-coroutines-tutorial-for-android-getting-started#toc-anchor-004
    - A suspending function is simply a function that can be paused and resumed at a later time. They can execute a long running operation and wait for it to complete without blocking.
    - Suspending functions can only be called by another suspending function or within a coroutine.
        - CoroutineScopes define the scope when a coroutine runs and handles their lifecycle
    - Requires an additional dependency
        - `implementation "androidx.room:room-ktx:2.4.2 "`

Database

Serves as the main access point for the underlying connection to your app's persisted, relational data. The database class must satisfy the following conditions:

- Be annotated with @Database which takes two arguments:

    - An array of the Entity classes (the tables) associated with the database
    - The database version
    - The exportSchema parameter tells Room where the database schema will be exported to
        - Optional, default is true
            - Must be set to true if using automigration
        - Should set it to false before shipping your app
    - The ability to migrate version of the database using the autoMigration parameter (optional)
- Be an abstract class that extends RoomDatabase
- The database class must contain an abstract method that has 0 arguments and returns the class that is annotated with @Dao.
    - We have to create an abstract method for every DAO class that we create.

After you have defined the database class you'll need to create an instance of the database and a method that returns it.

- It is good practice to use singleton approach for the database instance so there's only one instance of the database in your app.
- You create an instance of Database by calling Room.databaseBuilder() with the following parameters
    - Application context
    - Database class name
    - Name for the database (can be anything you want)
- Call .build() to create an instance of the database

Dependencies

Apply the kapt annotation processor Kotlin plugin by adding it in the plugins section defined at the top of your app's gradle file.

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

These dependencies:

```
implementation "androidx.room:room-runtime:2.4.2"
implementation "androidx.room:room-ktx:2.4.2 "
kapt "androidx.room:room-compiler:2.4.2"
```

List

In the app's gradle file I added this plugin:

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

And the Room library dependencies:

```
implementation "androidx.room:room-runtime:2.4.2"
implementation "androidx.room:room-ktx:2.4.2 "
kapt "androidx.room:room-compiler:2.4.2"
```

I'm going to update my List app to use Room and SQLite for data persistence.
As with DataStore I'm going to create a separate class to work with Room.

Entity class

I already have a data class that defines the structure of an item object so I'm going to convert this to be my Entity class which will make it my database table. Now this class is so simple you don't really need a relational database for data persistence but it will give us the idea of how Room works.
I updated the Item.kt model class with @Entity for the class name.
I made id the primary key @PrimaryKey which has an autoGenerate property. When you set autogenerate to true the type has to be Int.
I don't need to use @ColumnInfo as the column and the property have the same name.

```
@Entity
data class Item(
        @PrimaryKey(autoGenerate = true) var id: Int,
        var name: String
        )
{}
```

This should import the following:

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

Create DAO

Now we need to create a Data Access Object (DAO) class to define all our database operations.
Create a new package called util in the java list folder.
In the util package add a new class called ItemDAO but for Kind pick Interface.
We use @Dao to tell Room that this is a DAO class so all the SQL code needed to interact with SQLite is generated at compile time.
The database commands are called using suspend so they are executed asynchronously so the main UI thread is not blocked waiting for these to finish executing.
Room generates all the necessary code to update the LiveData object when a database is updated. The generated code runs the query asynchronously on a background thread when needed so we don't need to specify suspend for LiveData objects as that's automatically handled.

```
@Dao
interface ItemDAO {
    @Query("SELECT * FROM Item ORDER BY name ASC")
    fun getAllItems(): LiveData<List<Item>>

    // Room executes all queries on a separate thread
    @Insert
    suspend fun insertItem(item: Item)

    @Delete
    suspend fun deleteItem(item: Item)

    @Query("DELETE FROM Item")
    suspend fun deleteAll()
}
```

SQL database

Now we need to create a RoomDatabase class.
In the util package add a new class called ItemDatabase.
Make its superclass RoomDatabase() and make the class abstract.

Above the class declaration, add the @Database annotation. This takes entries which is an array of entity classes (we only have one but you would use commas between them if you had multiple). It also takes a version which I'll set to 1 and I set exportSchema to false.

We'll create the database instance as a singleton so it can be referenced from anywhere in the application and define the getDatabase() method which returns the instance of our database. We define our singleton as a Kotlin companion object since these are initialized when the class is loaded (typically the first time it's referenced anywhere in the app). We'll be able to refer to this object simply using the class name so we don't need to give it its own name. (Companion objects are similar to static objects in other languages such as Java).
The @Volatile modifier is used for our database instance object to ensure that each thread will be able to access it in the main memory and that it will NOT be stored in the local cache of a thread.

Before creating the database instance we check to see whether the instance is null.
We use synchronized when creating our database instance to ensure that the build method is never called concurrently by two different threads trying to create the database at the same time.

We also have to create an abstract method for every DAO class that we create. It's abstract, because this method will never be called directly. Instead, there will be some generated code that's created by the Room database in the background and that's the version of the method that will be called.

```
@Database(entities = arrayOf(Item::class), version = 1)
abstract class ItemDatabase: RoomDatabase() {

    companion object {
        // Singleton prevents multiple instances of database opening at the
same time
        @Volatile
        private var dbInstance: ItemDatabase? = null

        fun getDatabase(context: Context): ItemDatabase {
            // if the database instance is null, create the database
            if (dbInstance == null) {
                synchronized(ItemDatabase::class) {
                    dbInstance = Room.databaseBuilder(context,
ItemDatabase::class.java, "item.db").build()
                }
            }
            //return the database instance
            return dbInstance!!
        }
    }

    abstract fun itemDAO(): ItemDAO
}
```

Repository
I'm going to add a repository class for the logic between the DAO and the view model. I could just put it all in the view model but this allows more flexibility in the future if I want to change where the data is coming from or perhaps support both local and remote storage.

In my util package I'll add a class called ItemRepository. The DAO is passed into the repository constructor as opposed to the whole database. This is because it only needs access to the DAO, since the DAO contains all the read/write methods for the database. There's no need to expose the entire database to the repository.

In this class I'll have a LiveData object initialized with the list of items using the getAllItems() method from my ItemDAO class.
I also create methods that call the methods in our ItemDAO class. For insert and delete we need to do it on a background thread so I use the keyword suspend to make them suspending functions.

```
class ItemRepository(private val itemDAO: ItemDAO) {

    // Observed LiveData will notify the observer when the data has changed
    val itemList: LiveData<List<Item>> = itemDAO.getAllItems()

    // Room executes all queries on a separate thread
    suspend fun insertItem(item: Item) {
        itemDAO.insertItem(item)
    }

    suspend fun deleteItem(item: Item) {
        itemDAO.deleteItem(item)
    }

    suspend fun deleteAll() {
        itemDAO.deleteAll()
    }
}
```

View Model
I had to make some updates to my ItemViewModel to work with the Room.
The main change is that Room can return LiveData objects but not MutableLiveData objects (you'll notice that in the ItemDAO class). So I had to change my itemList and everything associated with it to the LiveData type.

Because Room.databaseBuilder() needs the application context in ItemDatabase.getDatabase(), ItemViewModel needs the application context.
I'm going to change my view model to have a superclass of AndroidViewModel instead of ViewModel since AndroidViewModel takes in an application context and ViewModel does not.
Although our view model should not have any references to an activity or fragment context because it might outlive them in the lifecycle, application context is ok because view models have the same lifecycle scope as the application.

ItemViewModel will have these variables:

- context for the Application context
- itemDatabase for the database instance
    - initialized using the getDatabase(context) method in the ItemDatabase class
- itemRepository so we can access the methods in the ItemRepository class
    - initialized by passing the DAO to the ItemRepository constructor

- itemList which is a LiveData object that will be observed in MainActivity as before
    - initialized from the itemRepository itemList

I still want my ViewModel to be the single data source for the app so the lifecycle changes are handled during rotation so I added methods for insert and delete that will call the corresponding methods in the ItemRepository class. Note that the calls to the itemRepository methods are called within viewModelScope.launch{}. viewModelScope is a CoroutineScope provided as an extension function of the ViewModel class. A CoroutineScope defines the scope when a coroutine runs and handles their lifecycle.

```
class ItemViewModel( application: Application):
AndroidViewModel(application) {
    private val context = application.applicationContext
    private val itemDatabase = ItemDatabase.getDatabase(context)
    private val itemRepository = ItemRepository(itemDatabase.itemDAO())
    val itemList: LiveData<List<Item>> = itemRepository.itemList

    //uses the viewModelScope coroutine scope to launch the coroutine in a
worker thread
    fun add(item: Item) = viewModelScope.launch{
        itemRepository.insertItem(item)
    }

    fun delete(item: Item) = viewModelScope.launch{
        itemRepository.deleteItem(item)
    }

    fun deleteAll() = viewModelScope.launch{
        itemRepository.deleteAll()
    }
}
```

Main Activity

The only thing I had to change in MainActivity.kt is when I add an item the call to create a new instance of Item the id is now an Int instead of a String. I will pass in 0 but it doesn't really matter as the property is set up to be automatically generated.
The observer is still listening for changes to the item list.

```
viewModel.add(Item(0, newItem))
```

RecyclerView Adapter

No changes were required to the recyclerView adapter class.

Delete all

Since I have a method to delete all the items from the database I created an options menu with one item that will clear the list by calling out deleteAll() method.

In strings.xml add a string.

```
<string name="action_settings">Clear List</string>
```

In resources create a new directory called menu.

In menu create a new menu resource file.
Name: options_menu
Source set: main
Directory name: menu

In options_menu.xml add one item.

```
<item
    android:id="@+id/action_deleteAll"
    android:orderInCategory="100"
    android:title="@string/action_settings"
    app:showAsAction="never" />
```

MainActivity

In MainActivity add the onCreateOptionsMenu() method so the menu is inflated/added to the action bar.

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.options_menu, menu)
    return true
}
```

Then add the onOptionsItemSelected() method which is called when an item in the options menu is clicked.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_deleteAll -> {
            viewModel.deleteAll()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```