CMU 04-801: Advanced Mobile Application Development
Week 5: Room and SQLite

Room
https://developer.android.com/training/data-storage/room/index.html
Saving data to Android's SQLite database is ideal 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.

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.

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:

Database
Contains the database holder and 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)
  - The database version
  - The exportSchema argument tells Room where the database schema will be exported to
    - Optional but recommended for version history
    - Default is true
      - You will get a warning if you don't specify a location in your app's gradle file (details in example below)
    - Should set it to false before shipping your app
- Be an abstract class that extends RoomDatabase
- Create an instance of this class and a method that returns it
  - It is good practice to use singleton approach for the database so there's only one instance of the database in your app.
- You create an instance of Database by calling Room.databaseBuilder() or Room.inMemoryDatabaseBuilder() 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
- 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.

Entity
https://developer.android.com/training/data-storage/room/defining-data
An entity represents a table in the database and is often just your POJO 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 auto-generate 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

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.
- @ 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 doesn't support database access on the main UI thread so we must use a background thread for our database calls.

- calling allowMainThreadQueries() on the builder will allow you to make database calls on the main UI thread but it is not recommended because it might lock the UI for a long period of time.
- 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

The Room library requires these dependencies to be added to your app's gradle file.

```
implementation "androidx.room:room-runtime:2.2.3"
annotationProcessor "androidx.room:room-compiler:2.2.3"
```

List

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

Add the Room library dependencies to your app's gradle file.

```
implementation "androidx.room:room-runtime:2.2.3"
annotationProcessor "androidx.room:room-compiler:2.2.3"
```

Entity class

I already have a POJO 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.java model class with @Entity for the class name and added an int called id for my @PrimaryKey.
I also have one public constructor that takes in the String name. (id is not needed as it is set to autogenerate)

```
Entity public class Item {
    @PrimaryKey(autoGenerate = true)
    public int id;
    private String name;

    public Item(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Create DAO

Now we need to create a Data Access Object (DAO) class to define all our database operations.

In the data 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.

```
@Dao
public interface ItemDAO {
    @Query("SELECT * FROM Item")
    LiveData<List<Item>> getAllItems();

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insertItem(Item item);

    @Delete
    void deleteItem(Item item);

    @Query("DELETE FROM Item")
    int deleteAll();
}
```

SQL database

Now we need to create a RoomDatabase class.
In the data package add a new class called AppDatabase and set the superclass to android.room.RoomDatabase. Also select abstract as a modifier.

Above the class declaration, add the @RoomDatabase annotation. This takes entries which is an array of entity classes that you wrap in curly braces and use commas to separate them if you have multiple. It also takes a version which I'll set to 1.

You either need to set exportSchema to false or provide the directory to place the generated schema JSON files. As you update your schema, you'll end up with several JSON files, one for every version. The next time you increase your version number again, Room will be able to use the JSON file for testing. If you don't provide the directory you will receive the following warning -- Warning: Schema export directory is not provided to the annotation processor so we cannot export the schema. You can either provide `room.schemaLocation` annotation processor argument OR set exportSchema to false.

In the build.gradle file for your app module, add this javaCompileOptions section to the defaultConfig section (under the android section). This will write out the schema to a schemas subfolder of your project folder.

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

$projectDir is a variable name for your project directory, you cannot change it.
schemas is a string, you can change it to any you like such as "$projectDir/MyOwnSchemas".toString()

We'll create the database as a singleton so it can be referenced from anywhere in the application using the get instance method which returns the instance of Database.

I'm going to use synchronization when I create the instance of the class to make sure that different parts of the application don't try to create the database at the same time. When you want to synchronize in Java, you need some kind of object to synchronize on. This can be pretty much anything so I'll create and instantiate a new standard Java object.
First I'll check to see whether the instance is null. If it is, then I'll add a synchronize block using the Java object. Then I'll do a secondary check to make sure that that instance is still null and if it is I'll create my database instance.

We 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 = {Item.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    private static AppDatabase instance;

    public static AppDatabase getInstance(Context context) {
        if (instance == null){
            synchronized (new Object()){
                if (instance == null){
                    instance =
Room.databaseBuilder(context.getApplicationContext(), AppDatabase.class,
"AppDatabase.db").build();
                }
            }
        }
        return instance;
    }
    public abstract ItemDAO itemDAO();
}
```

Repository
I'm going to add a repository class for the logic between the database 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 data package I'll add a class called AppRepository. This will follow the singleton pattern so to save a little typing you can select "Singleton" under Kind so the singleton code structure is done for you. That creates a class with a private static field, which is an instance of the class. It also creates a getInstance method that's also static to access the instance, and a private constructor.
We'll also need an instance of our database and methods that call the methods in our ItemDAO class. For insert and delete we need to do it on a background thread so I'm creating an instance of the Executor interface. To use it you call .execute() and pass it a new instance of the Runnable class. This starts a new background thread and I can make my database call in the run() method.

```
public class AppRepository {
    private static AppRepository mInstance;

    private AppDatabase mAppDatabase;

    //for background thread
    private Executor executor = Executors.newSingleThreadExecutor();

    public static AppRepository getInstance(Context context) {
        if (mInstance == null){
            mInstance = new AppRepository(context);
        }
        return mInstance;
    }

    private AppRepository(Context context) {
        mAppDatabase = AppDatabase.getInstance(context);
    }

    public LiveData<List<Item>> getAllItems(){
        return mAppDatabase.itemDAO().getAllItems();
    }

    public void insertItem(final Item newItem){
        executor.execute(new Runnable() {
            @Override
            public void run() {
                mAppDatabase.itemDAO().insertItem(newItem);
            }
        });
    }

    public void deleteItem(final Item newItem){
        executor.execute(new Runnable() {
            @Override
            public void run() {
                mAppDatabase.itemDAO().deleteItem(newItem);
            }
        });
    }

    public void deleteAll(){
        executor.execute(new Runnable() {
            @Override
            public void run() {
                mAppDatabase.itemDAO().deleteAll();
            }
        });
    }
}
```

View Model

I had to make some updates to my ItemViewModel to work with the Room.

The main issue is that in the ItemDAO class Room can return LiveData objects but not MutableLiveData objects. So I had to change my itemList and everything associated with it to the LiveData type.
I still want my ViewModel to be the single datasource 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 ItemDAO class.

```
public class ItemViewModel extends AndroidViewModel {

    private LiveData<List<Item>> itemList;
    private Context context;
    private AppRepository appRepository;

    public ItemViewModel(@NonNull Application application) {
        super(application);
        context = application.getApplicationContext();
        appRepository = AppRepository.getInstance(context);
        //for debugging to clear out the database each time I run the app
        //appRepository.deleteAll();
        itemList = appRepository.getAllItems();
    }

    public LiveData<List<Item>> getItemList() {
        return itemList;
    }

    public void insertItem(Item newItem){
        itemList.getValue().add(newItem);
        appRepository.insertItem(newItem);
    }

    public void deleteItem(Item item){
        appRepository.deleteItem(item);
    }

    public void deleteAll(){
        appRepository.deleteAll();
    }
}
```

Main Activity

The LiveData class does not have the setValue() method as the MutableLiveData class does so I updated MainActivity.java to use the insert and delete methods that I added to my ItemViewModel class instead of using setValue(). The observer is still listening for changes to the item list.

```
public class MainActivity extends AppCompatActivity {
    private List<Item> mItemList = new ArrayList<>();
    private ItemViewModel itemViewModel;
    private MyListAdapter listAdapter;
    private RecyclerView recyclerView;

    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(final View view) {
                //create alert dialog
                AlertDialog.Builder dialog = new
AlertDialog.Builder(MainActivity.this);
                //create edit text
                final EditText edittext = new
EditText(getApplicationContext());
                //add edit text to dialog
                dialog.setView(edittext);
                //set dialog title
                dialog.setTitle("Add Item");
                //sets OK action
                dialog.setPositiveButton("Add", new
DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int
whichButton) {
                        //get item entered
                        String newName = edittext.getText().toString();
                        if (!newName.isEmpty()) {
                            Item newItem = new Item(newName);
                            // add item
                            mItemList.add(newItem);
                            //call new insert method
                            itemViewModel.insertItem(newItem);
                        }
                        Snackbar.make(view, "Item added",
Snackbar.LENGTH_LONG)
                                .setAction("Action", null).show();
                    }
                });
                //sets cancel action
                dialog.setNegativeButton("Cancel", new
DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int
whichButton) {
                        // cancel
                    }
                });
                //present alert dialog
                dialog.show();
            }
        });

        //create a viewmodel
```

```
        itemViewModel = new ViewModelProvider(this, new
ViewModelProvider.AndroidViewModelFactory(this.getApplication())).get(ItemVi
ewModel.class);

        //get the recycler view
        recyclerView = findViewById(R.id.recyclerView);

        //divider line between rows
        recyclerView.addItemDecoration(new DividerItemDecoration(this,
LinearLayoutManager.VERTICAL));

        //set a layout manager on the recycler view
        recyclerView.setLayoutManager(new LinearLayoutManager(this));

        //create the observer
        final Observer<List<Item>> itemObserver = new Observer<List<Item>>()
{
            @Override
            public void onChanged(List<Item> items) {
                mItemList.clear();
                mItemList.addAll(items);
                if (listAdapter == null) {
                    //define the adapter
                    listAdapter = new MyListAdapter(itemViewModel,
MainActivity.this);
                    //assign the adapter to the recycle view
                    recyclerView.setAdapter(listAdapter);
                } else {
                    listAdapter.setItems(mItemList);
                    listAdapter.notifyDataSetChanged();
                }
            }
        };

        //set the observer
        itemViewModel.getItemList().observe(this, itemObserver);
    }
}
```

RecyclerView Adapter

The list adapter had some changes as well. I added a method to update the item list from MainActivity.

```
public void setItems(List<Item> items){
    mItemList = items;
}
```

And I updated when an item is deleted to use the new delete method in the ItemViewModel class.

```
public void onBindViewHolder(@NonNull final MyListAdapter.ViewHolder holder, int position) {
. . .
                public boolean onMenuItemClick(MenuItem item) {
                    int position = holder.getAdapterPosition();
                    Item removeItem = mItemList.remove(position);
                    mItemViewModel.deleteItem(removeItem);
. . .
}
```

Delete all

Since I have a method to delete all the items from the database I used the options menu that was already set up and changed the Settings menu item to be Clear List instead.
I updated the strings.xml and menu_main.xml files and then implemented the onOptionsItemSelected() method in MainActivity.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_delete) {
        mItemList.clear();
        itemViewModel.deleteAll();
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```