SQLite-14

In this assignment, you will build the necessary components to incorporate an SQLite database into the Soo Greyhounds app. The database will store notes that are attached to individual photos.

1. Create a new repository on GitHub and add your instructor as a collaborator.

2. Open your Soo Greyhounds app in Android Studio. Ensure you have submitted your previous lab. Remove the remote repository that is pointing to your previous lab and add the remote to your new repository. Perform a push to ensure everything is setup properly.

3. Before we get into the database work, let's update our app to remove our open photo detail button and add a RecyclerView that will list the data in our database. Go to your activity_main.xml layout file and delete your photoDetailButton.

4. Delete the code from MainActivity.java that is attached to photoDetailButton.

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

    Button photoDetailButton = findViewById(R.id.photoDetailButton);
    photoDetailButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(getBaseContext(), PhotoDetailActivity.class);
            startActivity(intent);
        }
    });
}
```

5. We now need to create a data class that will represent a photo where we can load data into from the database. Create a new java class called Photo and define the following fields/properties within your class:

   a.) id as an int
   b.) uuid as a String
   c.) title as a String
   d.) url as a String
   e.) note as a String

6. In your Photo.java file, generate setters and getters for your fields/properties (if you do not remember how to do this, look it up in your book).

7. Now, let's create a singleton class that will serve as our photo storage manager. Create a new class called PhotoStorage and setup the main components of our singleton class.

```
public class PhotoStorage {
   private static PhotoStorage sPhotoStorage;
   private Context mContext;
   private List<Photo> mPhotos;

   public static PhotoStorage get(Context context) {
      if (sPhotoStorage == null) {
         sPhotoStorage = new PhotoStorage(context);
      }
      return sPhotoStorage;
   }

   private PhotoStorage(Context context) {
      mContext = context;
   }

   public void addPhoto(Photo p) {
   }

   public List<Photo> getPhotos() {
      return new ArrayList<>();
   }

   public Photo getPhoto(String uuid) {
      return null;
   }
}
```

8. Next we will setup a RecyclerView. Go back to your activity_main.xml layout file and add a RecyclerView (add it any way you want, using the text/XML view or by using the design/graphic view). Make sure you give your RecyclerView the id photo_recycler_view.

9. Now, let's get our RecyclerView hooked up to a variable so we can manage it's data.

```
public class MainActivity extends AppCompatActivity {
  private RecyclerView mPhotoRecyclerView;
  …

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mPhotoRecyclerView = findViewById(R.id.photo_recycler_view);
    mPhotoRecyclerView.setLayoutManager(new LinearLayoutManager(this));
  }
  …
}
```

10. As we know, we need a ViewHolder to setup the UI of the RecyclerView and we also need an Adapter to handle the data to be shown in the RecyclerView. First implement the ViewHolder code. Given we are going to eventually load images in our RecyclerView, we will use the View of the ViewHolder to load some data in instead of create a layout for our ViewHolder.

```
public class MainActivity extends AppCompatActivity {
  …
  private class PhotoHolder extends RecyclerView.ViewHolder {
    private TextView mTitleTextView;

    public PhotoHolder(View itemView) {
      super(itemView);
      mTitleTextView = (TextView) itemView;
    }

    public void bindPhoto(Photo photo) {
      mTitleTextView.setText(photo.getTitle());
    }
  }
  …
}
```

11. Let's add our Adapter that will manage the data to display in the RecyclerView (we will change this in a future lab to load photos from a web service).

```
public class MainActivity extends AppCompatActivity {
  …
  private class PhotoAdapter extends RecyclerView.Adapter<PhotoHolder> {
    private List<Photo> mPhotos;

    public PhotoAdapter(List<Photo> photos) {
      mPhotos = photos;
    }

    @Override
    public PhotoHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
      TextView textView = new TextView(getBaseContext());
      return new PhotoHolder(textView);
    }

    @Override
    public void onBindViewHolder(PhotoHolder photoHolder, int position) {
      Photo photo = mPhotos.get(position);
      photoHolder.bindPhoto(photo);
    }

    @Override
    public int getItemCount() {
      return mPhotos.size();
    }
  }
  …
}
```

12. Now that we have our RecyclerView setup, let's create a way we can refresh it, and also refresh it when our MainActivity is loaded.

```
public class MainActivity extends AppCompatActivity {
   private List<Photo> mPhotos;
   …

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      mPhotoRecyclerView = findViewById(R.id.photo_recycler_view);
      mPhotoRecyclerView.setLayoutManager(new LinearLayoutManager(this));

      updateList();
   }

   private void updateList() {
      mPhotoRecyclerView.setAdapter(new PhotoAdapter(PhotoStorage.get(this).getPhotos()));
   }
   …
}
```

13. We can now setup our database and connect it to our RecyclerView. The first thing we will do is create a class to put our schema in. You will call this class SooGreyhoundsDBSchema, but in the Create New Class dialog, enter database.SooGreyhoundsDBSchema. This will put the SooGreyhoundsDBSchema.java file in its own database package, which you will use to organize all your database-related code. Inside SooGreyhoundsDBSchema, define an inner class called PhotoTable to describe your table.

```
public class SooGreyhoundsDBSchema {
   public static final class PhotoTable {
      public static final String NAME = "photos";
   }
}
```

14. The PhotoTable class only exists to define the String constants needed to describe the moving pieces of your table definition. The first piece of that definition is the name of the table in your database, PhotoTable.NAME. Next, describe the columns.

```
public class SooGreyhoundsDBSchema {
   public static final class PhotoTable {
      public static final String NAME = "photos";

      public static final class Cols {
         public static final String UUID = "uuid";
         public static final String TITLE = "title";
         public static final String URL = "url";
         public static final String NOTE = "note";
      }
   }
}
```

15. Create a class called SooGreyhoundsDBHelper in your database package

```
public class SooGreyhoundsDBHelper extends SQLiteOpenHelper {
   private static final int VERSION = 1;
   private static final String DATABASE_NAME = "SooGreyhounds.db";

   public SooGreyhoundsDBHelper(Context context) {
      super(context, DATABASE_NAME, null, VERSION);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
   }

   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
   }
}
```

16. For now, Soo Greyhounds will only have one version, so you can ignore onUpgrade(…). You only need to create your database tables in onCreate(SQLiteDatabase). To do that, you will refer to the PhotoTable inner class of SooGreyhoundsDBSchema. The import is a two-step process. First, write the initial part of your SQL creation code. Notice the SQL command in the execSQL() method. Using the NAME variable from the PhotoTable inner class, it will execute the command "create table photos (_id integer primary key autoincrement, uuid, title, url, note)" which is the command to create your table called photos and all the columns within your table.

```
@Override
public void onCreate(SQLiteDatabase db) {
   db.execSQL("create table " + SooGreyhoundsDBSchema.PhotoTable.NAME + "(" +
         " _id integer primary key autoincrement, " +
         SooGreyhoundsDBSchema.PhotoTable.Cols.UUID + ", " +
         SooGreyhoundsDBSchema.PhotoTable.Cols.TITLE + ", " +
         SooGreyhoundsDBSchema.PhotoTable.Cols.URL + ", " +
         SooGreyhoundsDBSchema.PhotoTable.Cols.NOTE +
         ")"
   );
}
```

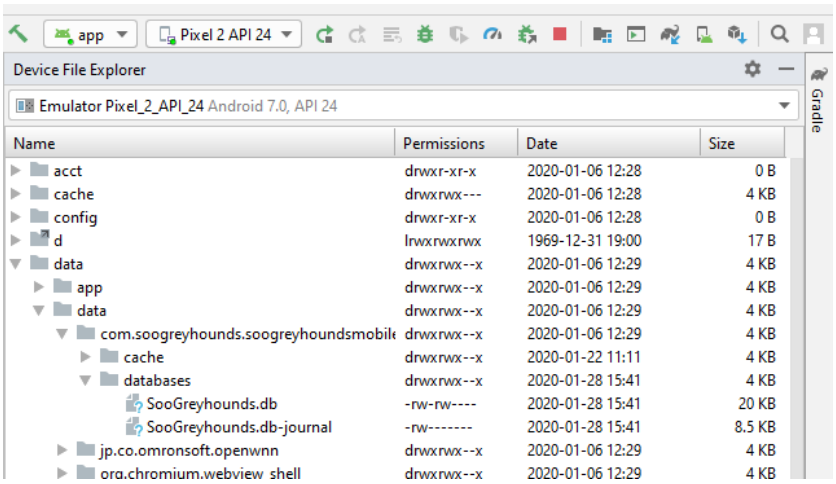
17. Update your PhotoStorage class to load the database.

```
public class PhotoStorage {
   …
   private SQLiteDatabase mDatabase;
   …
   private PhotoStorage(Context context) {
      mContext = context.getApplicationContext();
      mDatabase = new SooGreyhoundsDBHelper(mContext).getWritableDatabase();
   }
   …
}
```

18. Run Soo Greyhounds, and your database will be created.

19. Open the Device File Explorer. Select View → Tool Windows → Device File Explorer from the main menu. If you see a dialog that asks to disable ADB integration, click Yes.

20. When the Device File Explorer screen appears, click on the File Explorer tab. To see the database files Soo Greyhounds created, look in /data/data/com.soogreyhounds.soogreyhoundsmobile /databases/



21. You will be creating ContentValues instances from Photos a few times in PhotoStorage. Add a private method to take care of shuttling a Photo into a ContentValues.

```
public class PhotoStorage {
   …
   private static ContentValues getContentValues(Photo photo) {
      ContentValues values = new ContentValues();
      values.put(SooGreyhoundsDBSchema.PhotoTable.Cols.UUID, photo.getUUID());
      values.put(SooGreyhoundsDBSchema.PhotoTable.Cols.TITLE, photo.getTitle());
      values.put(SooGreyhoundsDBSchema.PhotoTable.Cols.URL, photo.getURL());
      values.put(SooGreyhoundsDBSchema.PhotoTable.Cols.NOTE, photo.getNote());
      return values;
   }
}
```

22. Now that you have a ContentValues, it is time to add rows to the database. Fill out addPhoto(Photo) with a new implementation.

```
public void addPhoto(Photo p) {
   ContentValues values = getContentValues(p);
   mDatabase.insert(SooGreyhoundsDBSchema.PhotoTable.NAME, null, values);
}
```

23. Continue applying ContentValues by writing a method to update rows in the database.

```
public Photo getPhoto(UUID id) {
   return null;
}

public void updatePhoto(Photo photo) {
   String uuid = photo.getUUID();
   ContentValues values = getContentValues(photo);

   mDatabase.update(SooGreyhoundsDBSchema.PhotoTable.NAME, values,
         SooGreyhoundsDBSchema.PhotoTable.Cols.UUID + " = ?",
         new String[] { uuid });
}

private static ContentValues getContentValues(Photo photo) {
  …
```

24. Let's implement the reading of data. Use query(…) in a convenience method to call this on your PhotoTable. Add this to your PhotoStorage.java file.

```
private Cursor queryPhotos(String whereClause, String[] whereArgs) {
   Cursor cursor = mDatabase.query(
         SooGreyhoundsDBSchema.PhotoTable.NAME,
         null, // columns - null selects all columns
         whereClause,
         whereArgs,
         null, // groupBy
         null, // having
         null  // orderBy
   );

   return cursor;
}
```

25. Create a new class in the database package called PhotoCursorWrapper.

```
public class PhotoCursorWrapper extends CursorWrapper {
   public PhotoCursorWrapper(Cursor cursor) {
      super(cursor);
   }
}
```

26. That creates a thin wrapper around a Cursor. It has all the same methods as the Cursor it wraps, and calling those methods does the exact same thing. This would be pointless, except that it makes it possible to add new methods that operate on the underlying Cursor. Add a getPhoto() method that pulls out relevant column data.

```
public class PhotoCursorWrapper extends CursorWrapper {
   public PhotoCursorWrapper(Cursor cursor) {
      super(cursor);
   }
   public Photo getPhoto() {
      String uuid =
getString(getColumnIndex(SooGreyhoundsDBSchema.PhotoTable.Cols.UUID));
      String title = getString(getColumnIndex(SooGreyhoundsDBSchema.PhotoTable.Cols.TITLE));
      String url = getString(getColumnIndex(SooGreyhoundsDBSchema.PhotoTable.Cols.URL));
      String note =
getString(getColumnIndex(SooGreyhoundsDBSchema.PhotoTable.Cols.NOTE));

      Photo photo = new Photo();
      photo.setUUID(uuid);
      photo.setTitle(title);
      photo.setURL(url);
      photo.setNote(note);

      return photo;
   }
}
```

27. With PhotoCursorWrapper, vending out a List<Photo> from PhotoStorage will be straightforward. You need to wrap the cursor you get back from your query in a PhotoCursorWrapper, then iterate over it calling getPhoto() to pull out its Photos. For the first part, in PhotoStorage.java, update queryPhotos(…) to use PhotoCursorWrapper.

```
~~private Cursor queryPhotos(String whereClause, String[] whereArgs) {~~
private PhotoCursorWrapper queryPhotos(String whereClause, String[] whereArgs) {
   Cursor cursor = mDatabase.query(
         SooGreyhoundsDBSchema.PhotoTable.NAME,
         null, // columns - null selects all columns
         whereClause,
         whereArgs,
         null, // groupBy
         null, // having
         null  // orderBy
   );
   ~~return cursor;~~
   return new PhotoCursorWrapper(cursor);
}
```

28. Then get getPhotos() into shape. Add code to query for all photos, walk the cursor, and populate a Photo list.

```
public List<Photo> getPhotos() {
   return new ArrayList<>();
   List<Photo> photos = new ArrayList<>();

   PhotoCursorWrapper cursor = queryPhotos(null, null);

   try {
      cursor.moveToFirst();
      while (!cursor.isAfterLast()) {
         photos.add(cursor.getPhoto());
         cursor.moveToNext();
      }
   } finally {
      cursor.close();
   }

   return photos;
}
```

29. PhotoStorage.getPhoto(String) will look similar to getPhotos(), except it will only need to pull the first item, if it is there.

```
public Photo getPhoto(String uuid) {
   return null;
   PhotoCursorWrapper cursor = queryPhotos(
         SooGreyhoundsDBSchema.PhotoTable.Cols.UUID + " = ?",
         new String[] { uuid }
   );

   try {
      if (cursor.getCount() == 0) {
         return null;
      }

      cursor.moveToFirst();
      return cursor.getPhoto();
   } finally {
      cursor.close();
   }
}
```

30. The last thing we will do is add a temporary menu item to add a record in our photos table and refresh the RecyclerView to show the data was added. First, add a menu item in your menu by opening your activity_main.xml file in your **res > menu** folder and add a new item (you will also have to add a string for the menu item). Make sure you give your menu item an id add_photo.

31. Now let's add some code to create a photo record in our database when the menu item is pressed. In MainActivity.java, add the following to your menu function.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      …
      case R.id.add_photo:
         Photo photo = new Photo();
         photo.setUUID("234243-dsfsa-23sdf");
         photo.setTitle("A photo");
         photo.setURL("https://www.test.com/Test.jpg");
         photo.setNote("This is a note");

         PhotoStorage.get(getBaseContext()).addPhoto(photo);
         updateList();

         return true;
      …
   }
}
```

32. Run your app, press the three dots to open your menu, then click your add photo item in the menu. You should see your photo in the RecyclerView list.

Final App