# SQLite - An Embedded Database Guide

## This document is a comprehensive guide to creating an embedded SQLite database for an Android Studio application. We will cover table creation, inserting, displaying data using trivial and non-trivial queries, and how to delete data. The example MyFace Day Spa business is used to create the database.

Contents

[MyFace Day Spa Entity Relationship Diagram 3](#_Toc10380031)

[SQLite Need to Know: 4](#_Toc10380032)

[Features to note: 4](#_Toc10380033)

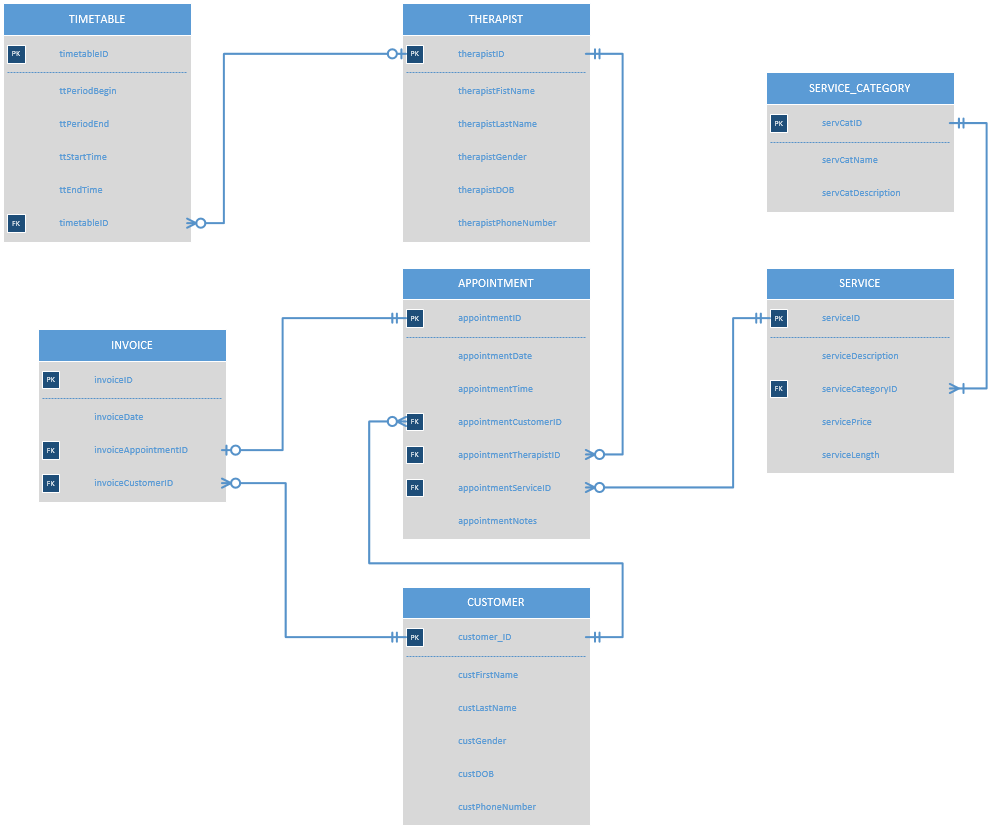
[Android Studio setup: 4](#_Toc10380034)

[Steps: 5](#_Toc10380035)

[Non-Trivial Queries 36](#_Toc10380036)

[References 38](#_Toc10380037)

# MyFace Day Spa Entity Relationship Diagram



# SQLite Need to Know:

SQLite is an embedded database environment. It is used by Android, Apple’s IOS and other systems with low memory consumption requirements. Its popularity comes from its low memory consumption, ease of use, and free availability. Rittmeyer (2013) describes SQLite as being different from conventional database systems.

## Features to note:

* SQLite is serverless. Meaning there is no SQLite process running. It acts like a library, allowing access to the database contents. There is no configuration, no adding users, no managing access levels, no tablespaces etc.
* SQLite stores all the contents of the database into one single database file.
* SQLite supports limited datatypes; NULL, INTEGER, REAL, TEXT, and BLOB. During this assignment we will be using dates for the Timetable and Appointment tables. SQLite does not support DATE and DATETIME datatypes. We will write our dates in TEXT columns as strings in the format “2019-05-30”. Storing dates in this format offer some date functionality.
* SQLite uses manifest typing. Type information is dependent on the value inserted, not the column’s data definition, set in the CREATE TABLE statement. This means that it is allowed in SQLite to create an INTEGER type column and still insert TEXT entries. We will be using matching datatypes.
* SQLite has no fixed column length. Strings and numbers can be any length. SQLite will not throw any exceptions (SQLExceptions).

# Android Studio setup:

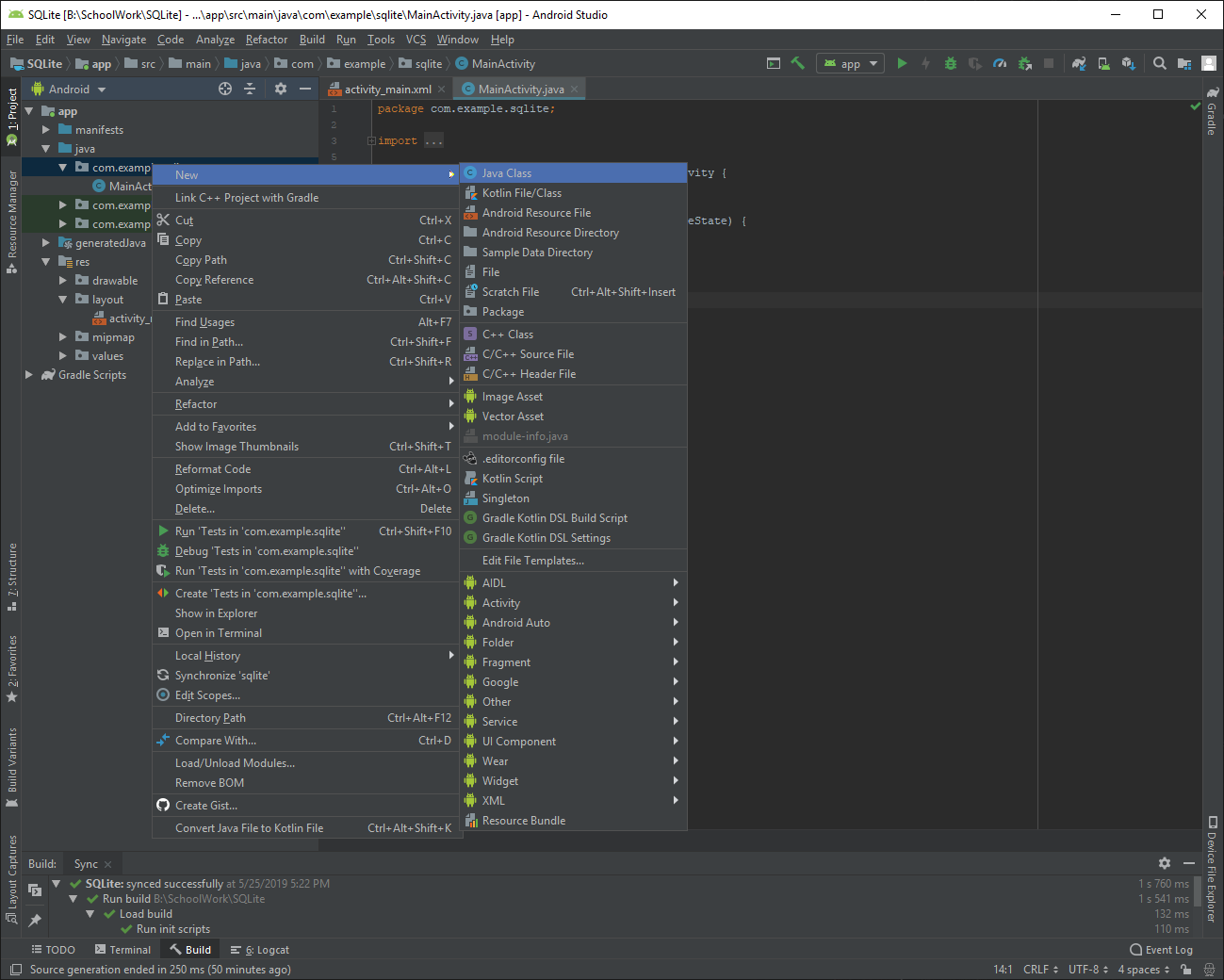
1. Start a new project in Android Studio.
2. Select ‘Empty Activity’. Next.
3. Name the application “SQLite”. Set the save location. Finish.

Android Studio navigation:

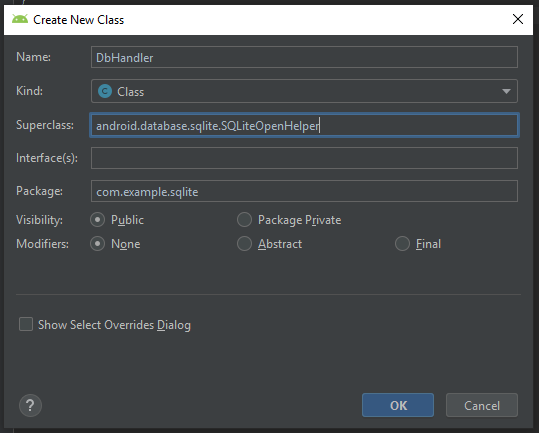
1. The project view is on the left-hand side of Android Studio. It contains the directory where you can navigate between activities. Activities are where we will use Java and SQL to create the database.
   * If missing, project view can be accessed via View -> Tool Windows -> Project.

# Steps:

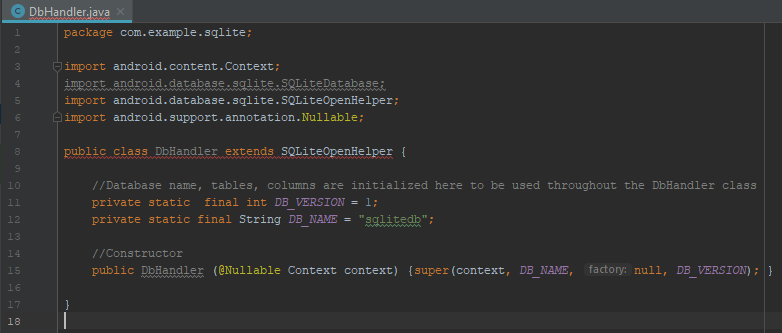
1. Create a subclass of the SQLiteOpenHelper class. The subclass named “DbHandler” will have access to SQLite functionality.



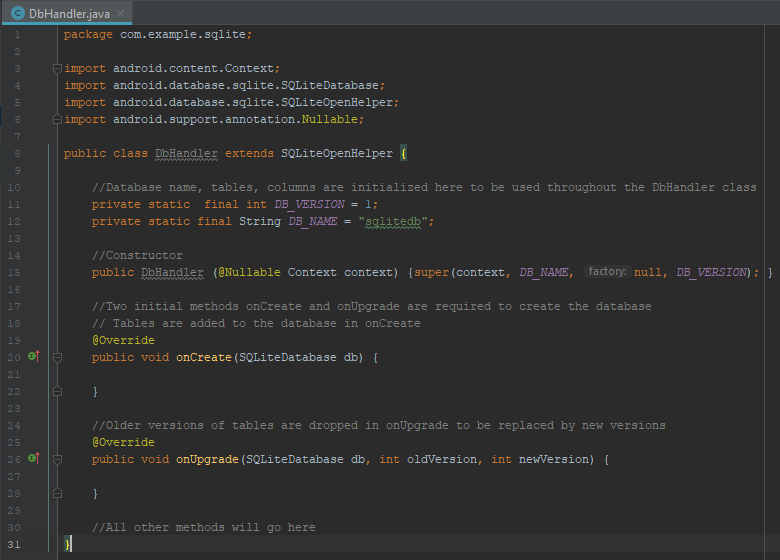
* 1. In project view, right click the package ‘com.example.sqlite’. It currently contains MainActivity. Select New -> Java Class.
  2. Name the class “DbHandler”. Under ‘Superclass’ enter “SQLiteOpenHelper” and select the auto-completion that android studio provides. Ok.



* 1. DbHandler can now be seen in the same package as MainActivity. If DbHandler.java is not open, double click on it in project view.
  2. DbHandler contains errors. The class declaration is underlined red. We will add a constructor and two methods to remove the errors.
  3. Add a constructor to DbHandler class. The constructor requires a database name and version. DbHandler class should look the same as the below screen shot.



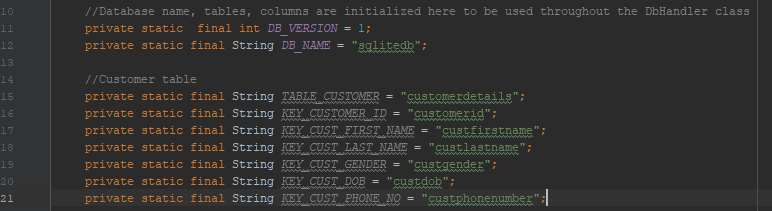
* 1. Add two methods ‘onCreate’ and ‘onUpgrade’ to the DbHandler class. Note that a double forward slash // is a comment and ignored by the compiler. It is just for us to read and doesn’t affect code.



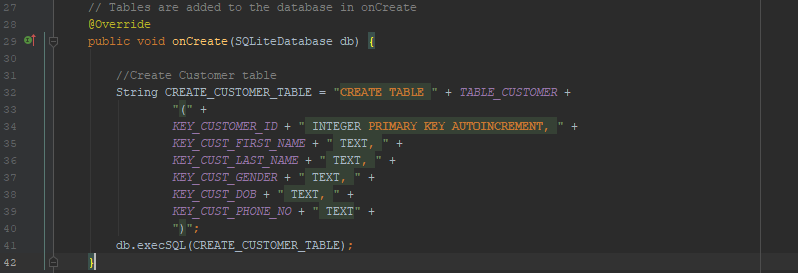
* 1. The errors are now resolved. DbHandler class is set up and the database is ready for tables to be added. Tables are created inside the onCreate method. The onUpgrade method will ensure tables stay up to date. Note that all future methods will be placed within the DbHandler class; Ensure the methods are inside the most outer layer of curly braces.

I will demonstrate creating the Customer table. Repeat steps 2, 3, and 4 to create additional tables.

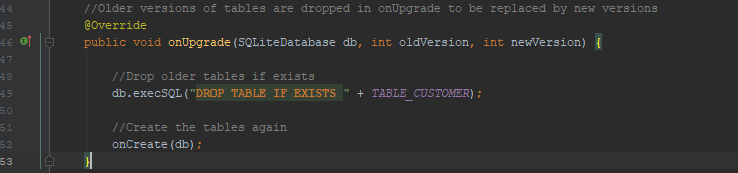
1. Before creating our tables, the table and column names must be initialized as string variables. The strings will be used in our SQL statements. Add the strings below the database version and name strings added in step 1 e.
   1. Add a table name, a primary key, first name, last name, gender, date of birth, and phone number columns.



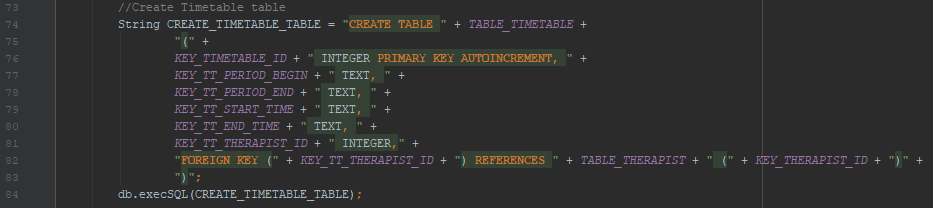
1. Inside the onCreate method; We will now use the strings we just created to write an SQL statement that will be passed into an embedded SQLite function. When creating the table, we must assign datatypes to each column. SQLite has limited datatypes as described in the start of this document under the ‘SQLite Need to Know’ heading.



1. Add our created Customer table to our onUpgrade method. Again, we will write an SQL statement. We will then call the onCreate method from within this onUpgrade method. Note that we only call the onCreate method once here; Do not call onCreate for each additional table added.

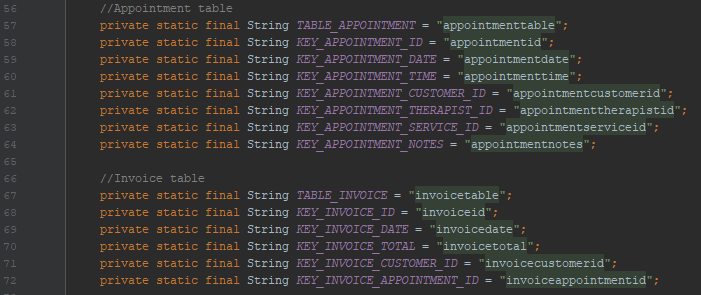


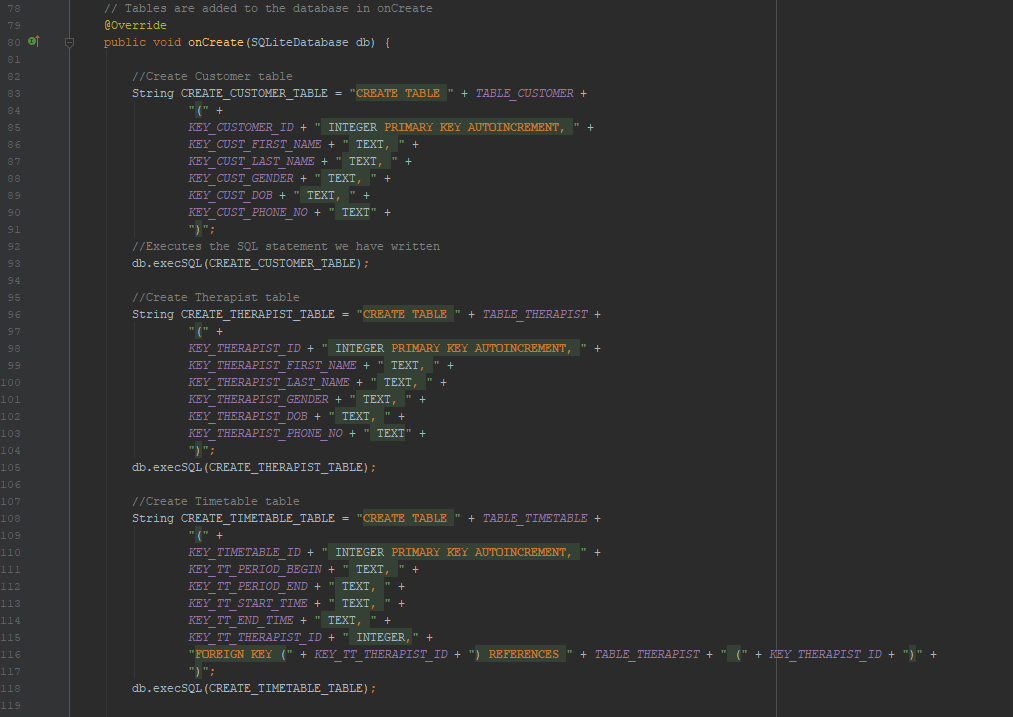
1. Continue to add all tables required for the database.
   1. Certain tables will use foreign keys. To create a table with a foreign key, refer to the image below. In this example KEY\_TT\_THERAPIST\_ID is the foreign key. It is referencing KEY\_THERAPIST\_ID, a primary key, from the Therapist table.

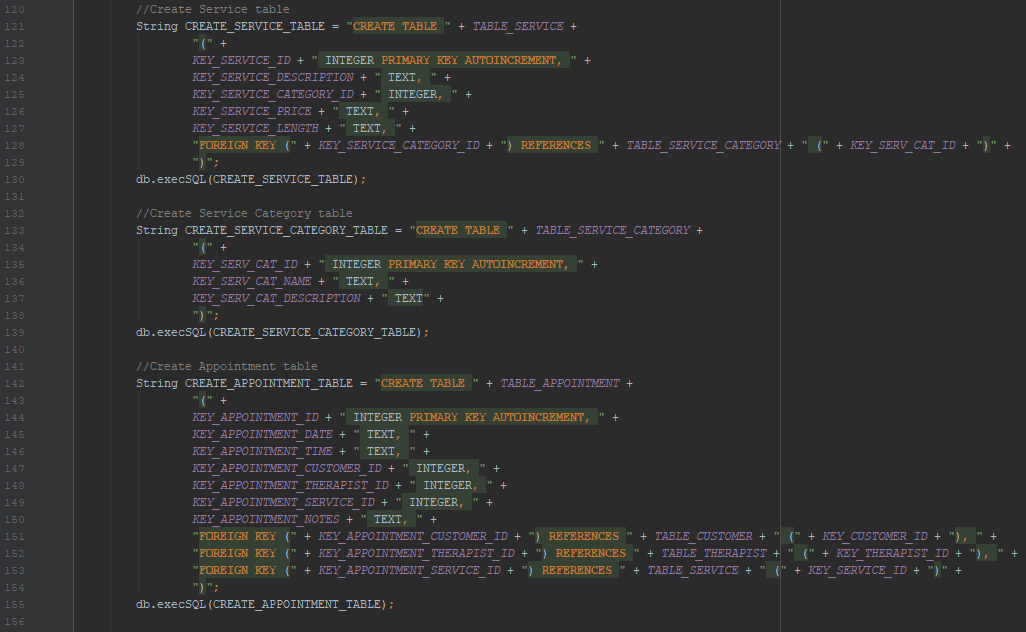


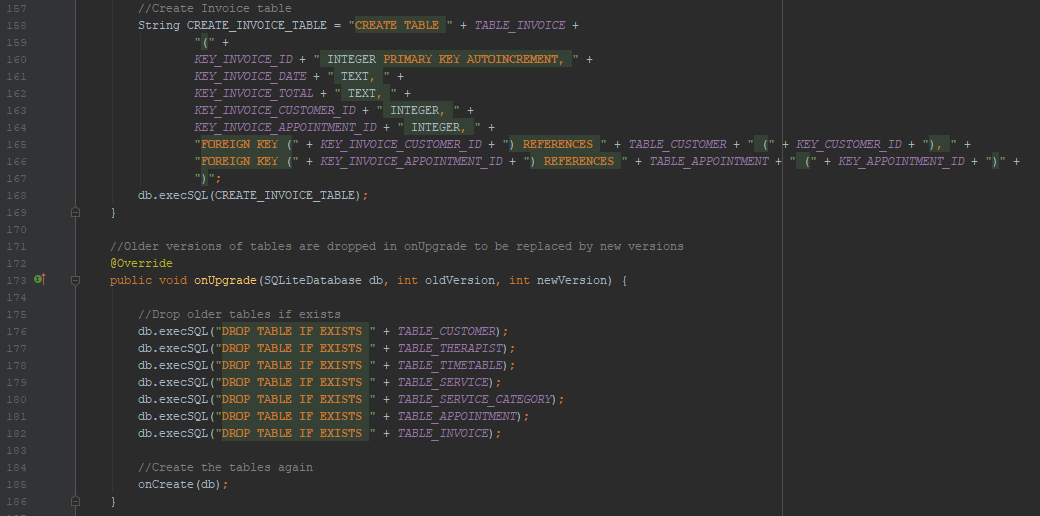
* 1. Seven tables have now been added to create a simple database for MyFace Beauty Therapy Day Spa. The tables are: Customer, Therapist, Timetable, Service, Service Category, Appointment, and Invoice. Ensure your DbHandler matches the below images.



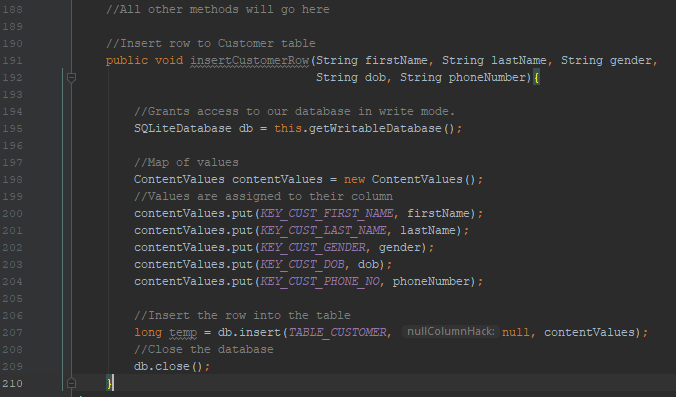






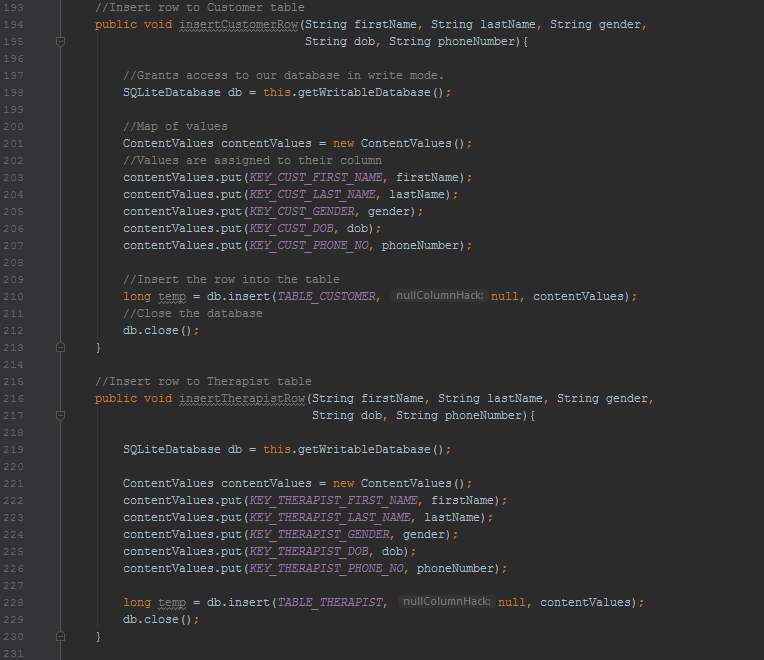


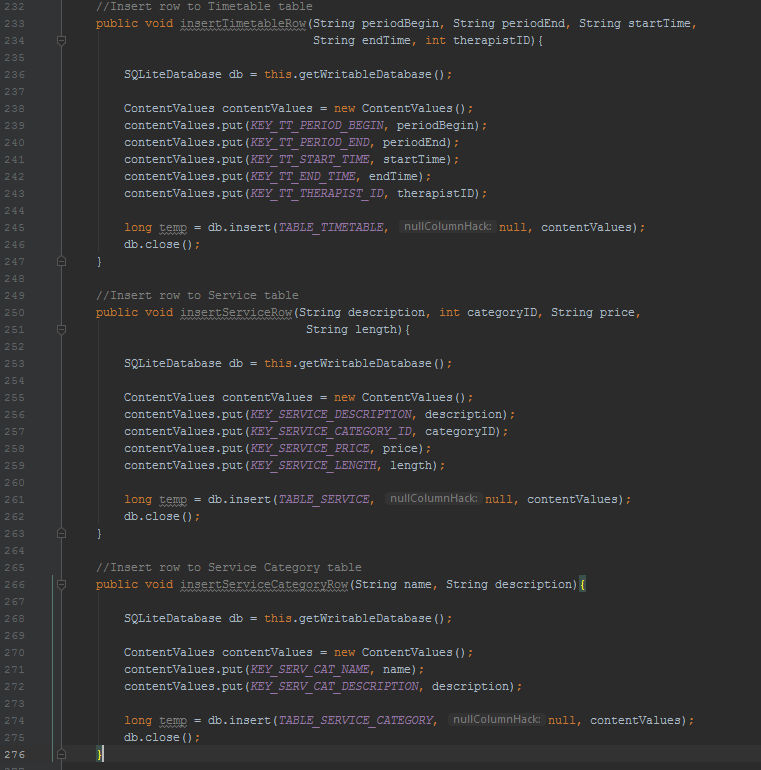
1. We will now create a method that allows for data to be written into specific tables. The method will take arguments for each column of the table that are not an auto created primary key.
   1. Using the Customer table as the example, our method will take in a value for the customer’s first name, last name, gender, date of birth, and phone number. The customer id is automatically assigned for us. Ensure the arguments’ datatypes are compatible with the columns’ datatypes (TEXT accepts String, INTEGER accepts int).
   2. To write (add) data into the database, we will create a reference to our database in write mode.
   3. The values that this method takes in will be now assigned to their respective columns, the columns that we created for the Customer table. To do this we create a map of values.
   4. Using our reference for the database in write mode, we call the SQLite method ‘insert’. We pass the name of the table and the map of values into this method. The insert method returns a ‘long’ number; We don’t do anything with this number. We must also close our database as we have finished writing to this table.

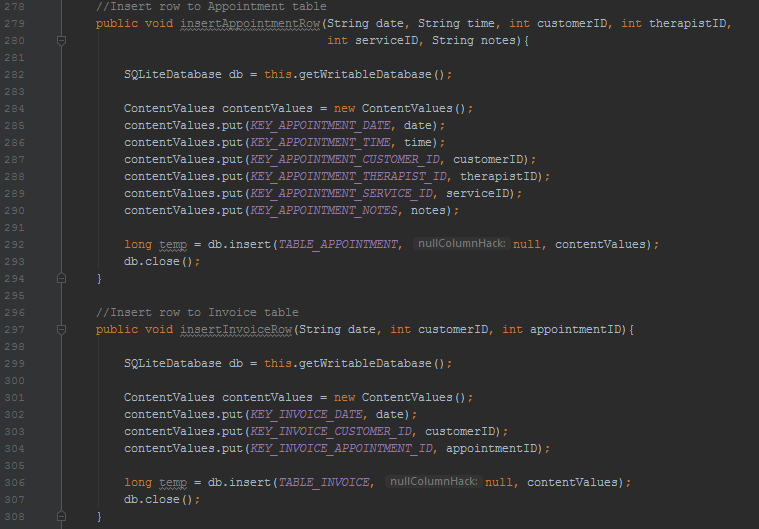


* 1. Repeat step 6 for each table in the database.

1. Quick recap:
   1. We have now created a subclass of SQLiteOpenHelper named DbHandler. This is where we are creating our Android Studio embedded SQLite database.
   2. We initialized the seven table names and their individual column names as string variables. These variables are used to write our SQL statements.
   3. In the onCreate method, we created the seven tables.
   4. In the onUpgrade method, we drop the seven tables if they exist. We then call onCreate once only.
   5. We have created an insert row (data) method for each table allowing us to populate the tables in step 8. Ensure your insert row methods match the below code.
   6. Pay attention to syntax while we create the database. We are writing our SQLite database in Java. Ensure parentheses ‘()’, curly braces ‘{}’, and semicolons ‘;’ are in the correct places.







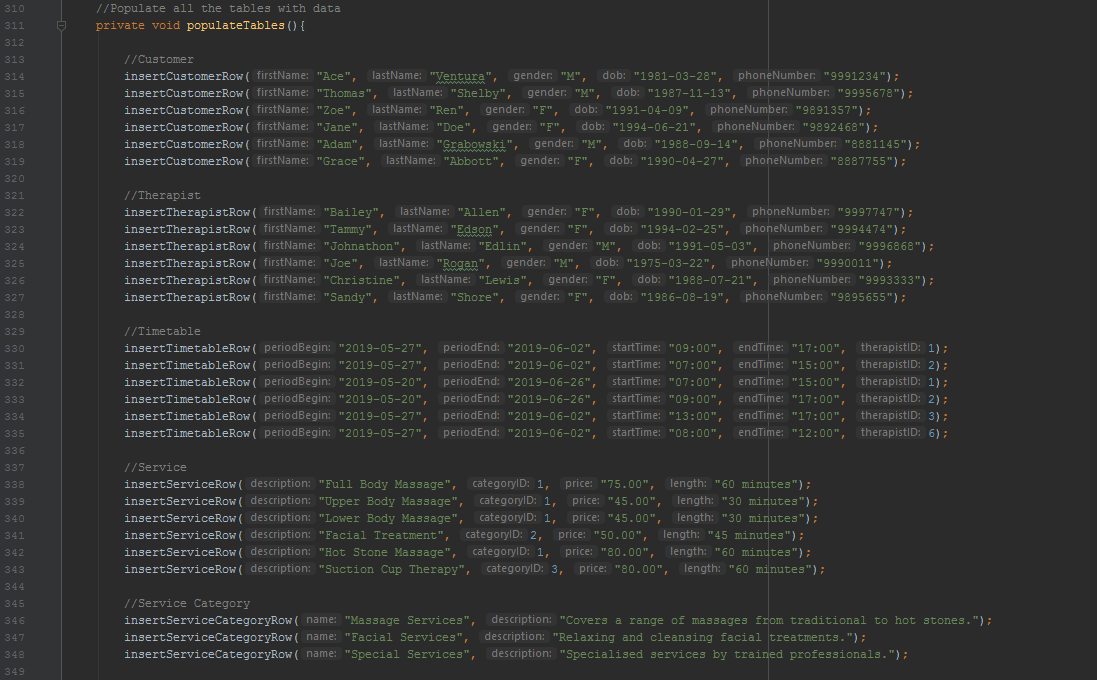
1. It is now time to use everything we have previously created, and add data to our database. We will add six rows of data to each table. Meaning, we will call each insert row method, created in step 6, six times each.
   1. Create a method “populateTables”. All insert row methods are called from within this method.
   2. We can start with the customer table. Call insertCustomerRow and within the method’s parentheses, enter data for a single customer. We do not enter primary key values because during table creation, we set them to be automatically assigned. The information that you enter must be in the same order as the columns i.e. customer first name, last name, gender, date of birth, and phone number.



* 1. Columns with a ‘text’ datatype will accept string values. Any column with an integer datatype will accept an integer value (a number with no quotation marks).

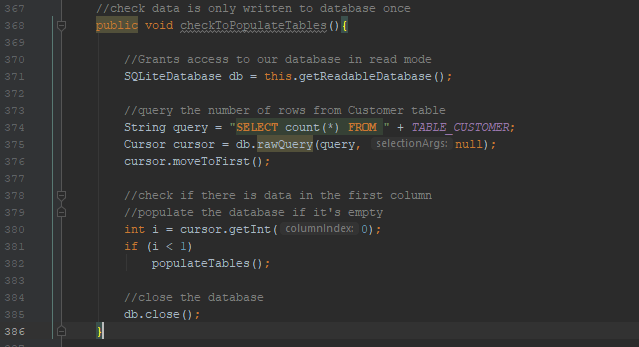


* 1. Continue adding data for each table. There needs to be six rows for each table.



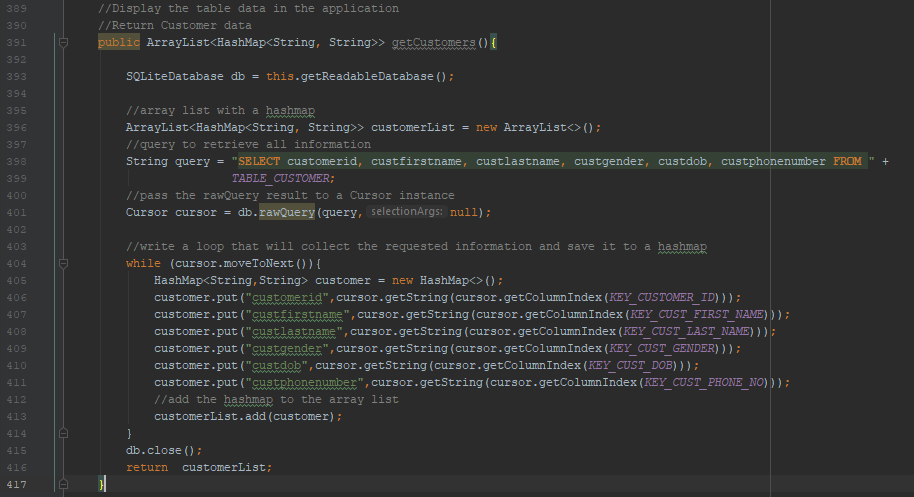


1. The database is embedded within the mobile application. We want to populate the tables with our data the first time we run the application on a device (or emulator). We do not want to keep adding the same rows each time we access the app. To achieve this, we will write a quick check method.
   1. To read data from the database, we will create a reference to our database in read mode.
   2. We query the count (number of rows) in the Customer table. It is our first table created and populated. If there is no data in Customer table, there is no data in any table.
   3. Next we look at the first column. If there are zero rows, meaning no data, we call populateTables method, created step 8.

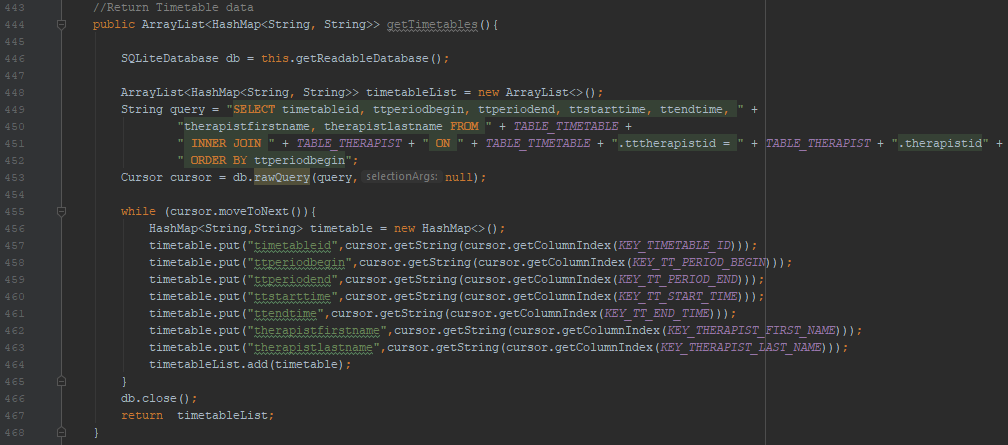


We will use the checkToPopulateTables method shortly. Our database is embedded into our application. To view the data that populates our tables we must retrieve it from the database (the backend) using queries and display it in the application (the frontend). This is the purpose of SQLite in Android Studio.

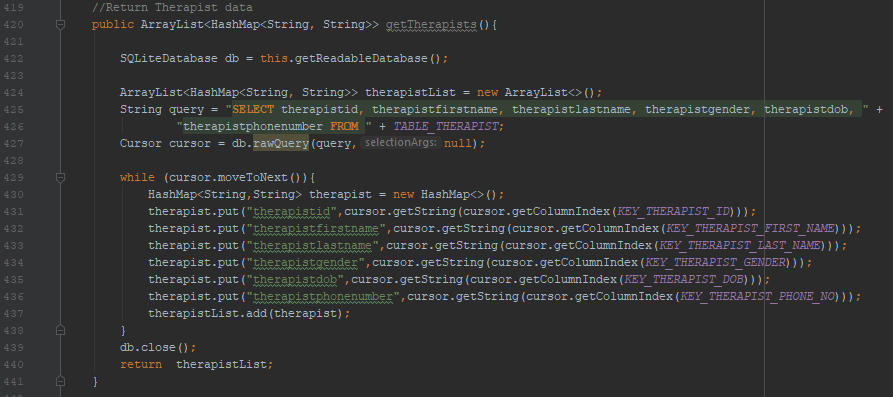
1. We are going write a method for each table that returns all data. Here we will utilize the relationships we created, that exist between the tables, using the foreign and primary keys. The purpose is to show information that is relevant for humans. For example, our Appointment table stores a therapist id, customer id, and service id; All foreign keys. If a staff member wants to review an appointment Therapist 7, Customer 85, and Service 1 is not helpful information. Instead we will show Therapist Nadia Glover, Customer Ashleigh Green, and Service Full Body Massage.
   1. I will demonstrate returning the Customer table data by writing the getCustomers method. Create a reference to our database in read mode.
   2. Create an Array List of HashMap called customerList. This will contain the customer data that we return.
   3. Write our query to select all the columns from Customer table. We are using the string column names we initialized in step 2 a.
   4. The results of the query are stored in a cursor.
   5. We iterate (loop) through the cursor, adding the information to a HashMap. A HashMap requires two arguments, a key and a value. The key is our column name and the value is the information stored there at that row. Note that the column names match the column names used in our query.
   6. The HashMap, a row of customer information, is added to customerList. This is done for each row stored in the cursor.
   7. Close the database.
   8. Return customerList.

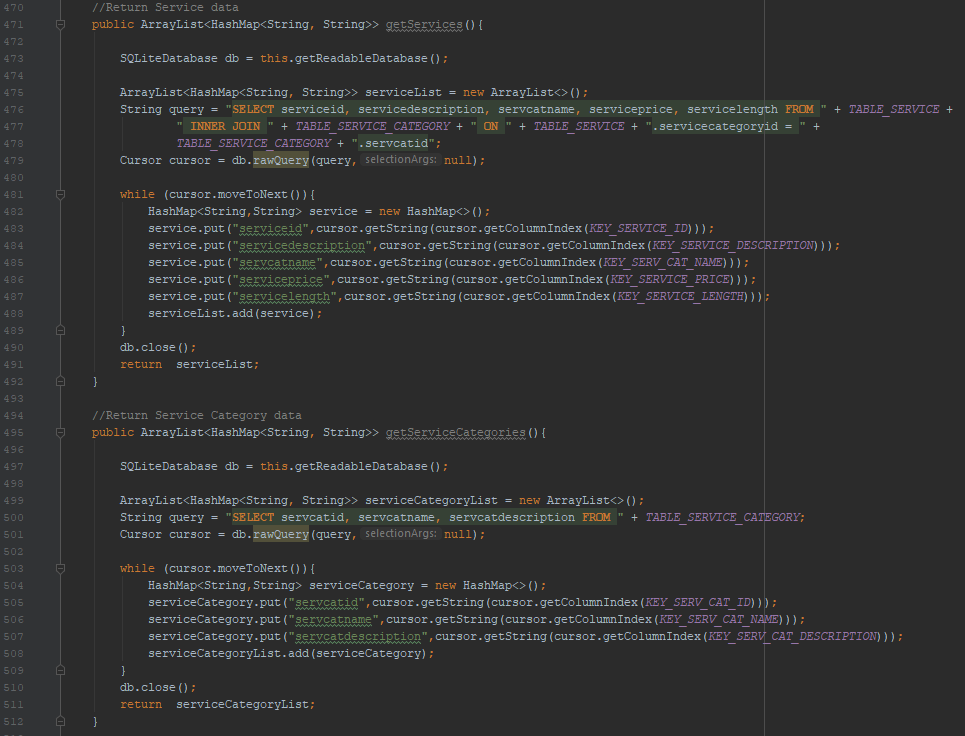


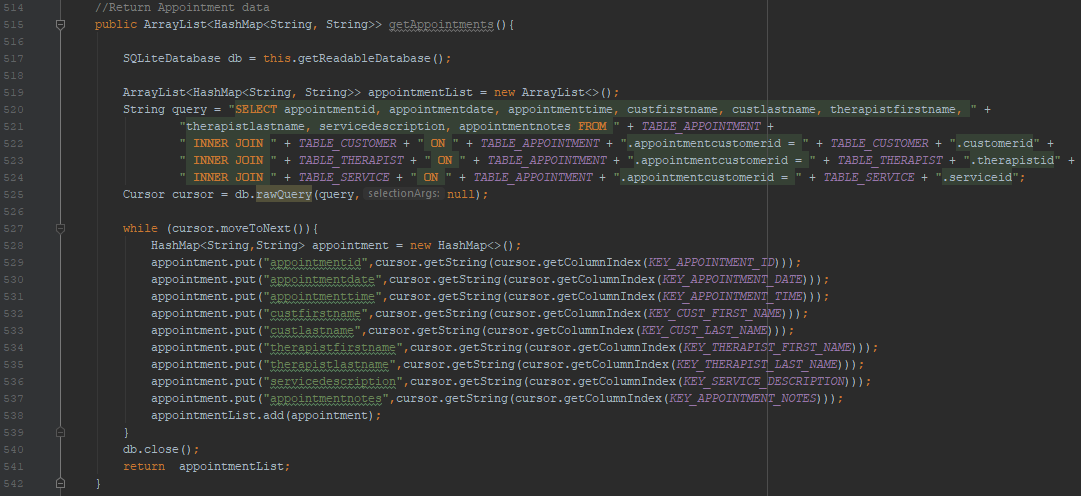
1. Write a return table data method for each remaining table. When returning table data containing foreign keys, the queries become more complex. Everything else remains the same.
   1. Timetable contains therapist id foreign key. We want to select the therapist first name and therapist last name columns instead of the timetable therapist id from the Timetable table.
   2. We join Therapist table on the Timetable table’s therapist id that has the same value as Therapist table’s therapist id.
   3. For display purposes we will order the rows by the timetable’s begin date from oldest to newest.



* 1. Compare your methods with the images below. Take your time to understand how the relationships are working amongst the tables.

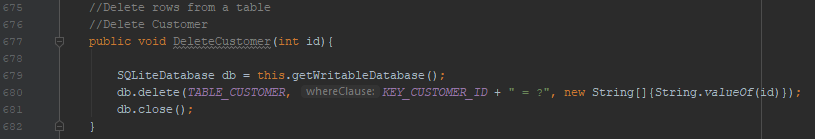








1. Deleting a row from a table requires a small method.
   1. Create a DeleteCustomer method.
   2. Access the database in write mode.
   3. Delete a row from Customer table where the customer id matches the id we are searching for.
   4. Close the database.



* 1. Create this method for each table.

The database at this stage is setup. However, note that our populateTables method is called from within our checkToPopulateTables method. This method has not been called. Our tables are not yet populated. We must run the application. The following steps are not SQLite steps. They are Android Studio steps. SQLite is an embedded database, it is built into Android Studio. To learn and use SQLite properly in this example, we must know the Android Studio steps.

1. Rename MainActivity to CustomerActivity. This is where we will display our Customer data.
   1. In the project view find MainActivity under app -> java -> com.example.sqlite -> MainActivity. Right click MainActivity -> Refactor -> Rename.
   2. Enter “CustomerActivity”, ensure all tick boxes are selected. Refactor.
   3. In the project view find activity\_main.xml under app -> res -> layout -> activity\_main.xml. Right click activity\_main.xml -> Refactor -> Rename.
   4. Enter “activity\_customer.xml”. Refactor.
2. We will assign one activity (an application page) per table. To navigate between the activities while the application is running, we will create an action/toolbar containing a menu and links to each activity.
   1. Open ‘AndroidManifest.xml’. In project view under app -> manifests -> AndroidManifest.xml.
   2. Locate this line of code:
      1. android:theme=”@style/AppTheme”.



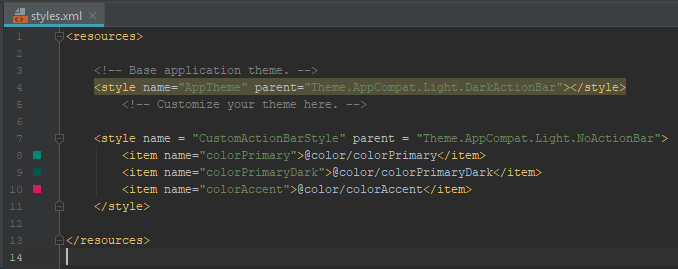
* 1. Replace with:
     1. android:them=”@style/Theme.AppCompat.Light.NoActionBar”. Close the manifest for now.



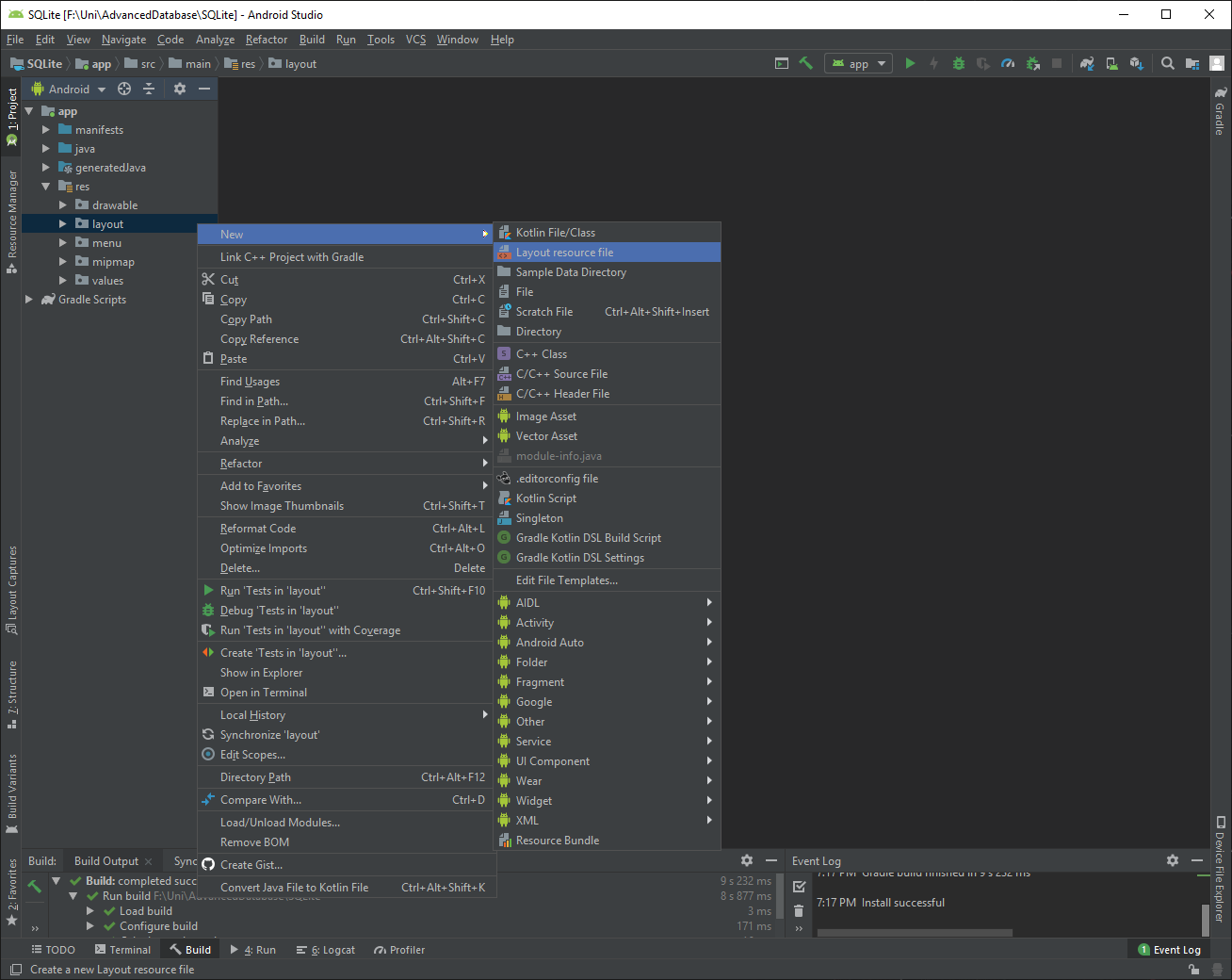
* 1. Open ‘styles.xml’. In project view under app -> res -> values -> styles.xml.



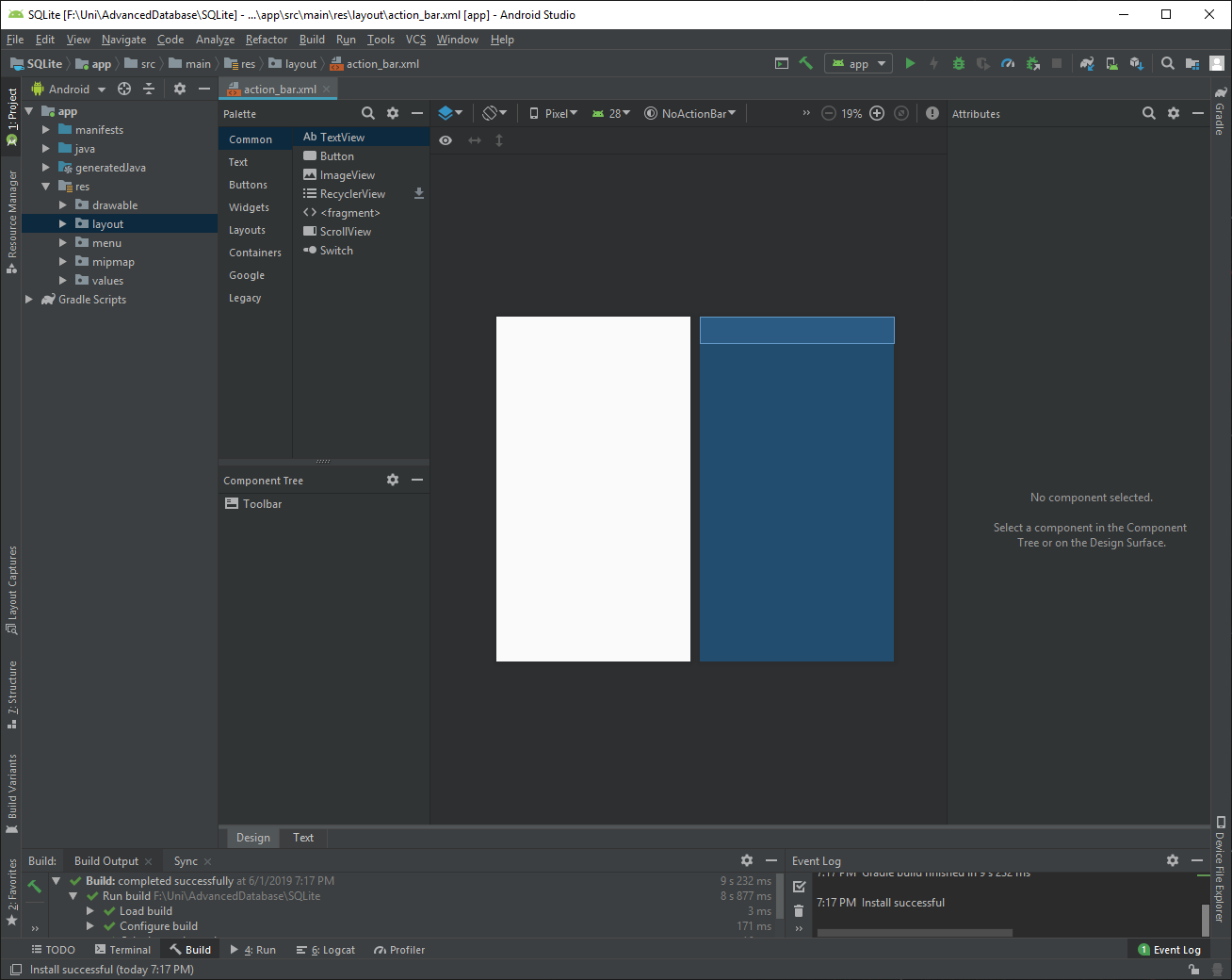
* 1. We will now reference the theme we added in step 13 c. Modify styles.xml to match the below image. Close styles.xml



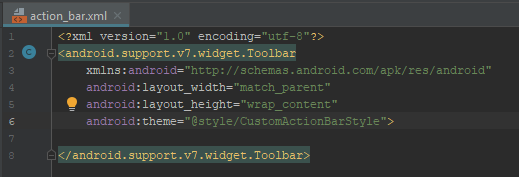
* 1. Create a new layout resource file for our action bar. In project view under app -> res -> layout, right click package ‘layout’. Select New -> Layout resource file.



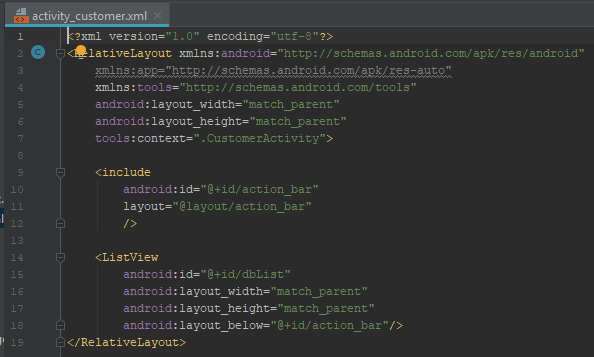
* 1. Name the resource file “action\_bar”. OK.
  2. Open action\_bar.xml. We are currently in design view. In the bottom left corner, setect the ‘Text’ tab for text view.



* 1. Change android:layout\_height from “match\_parent” to “wrap\_content”. Add a theme. We will reference ‘CustomActionBarSylte’ from styles.xml that we modified in step 13 e. Ensure action\_bar.xml matches the below image. Close action\_bar.xml.



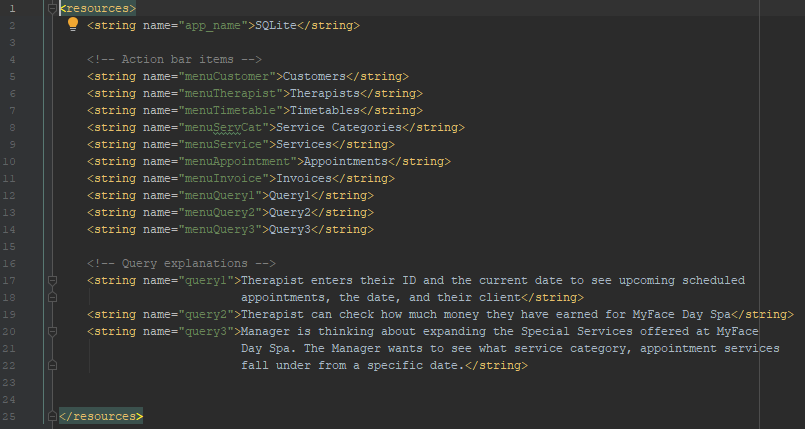
1. CustomerActivity contains Java code. It is the functionality of our application page. The layout of CustomerActivity takes place in activity\_customer.xml. Every activity we add has a corresponding layout file.
   1. Open activity\_customer.xml.
   2. Include our action bar layout and add a ListView (where our database information is displayed).



The code of the above image can be copied and pasted in each activities’ corresponding layout file that we will create. Just be sure that the context refers to the correct activity. This is activity\_customer.xml. The layout resource file for CustomerActivity. Take note of line 7.

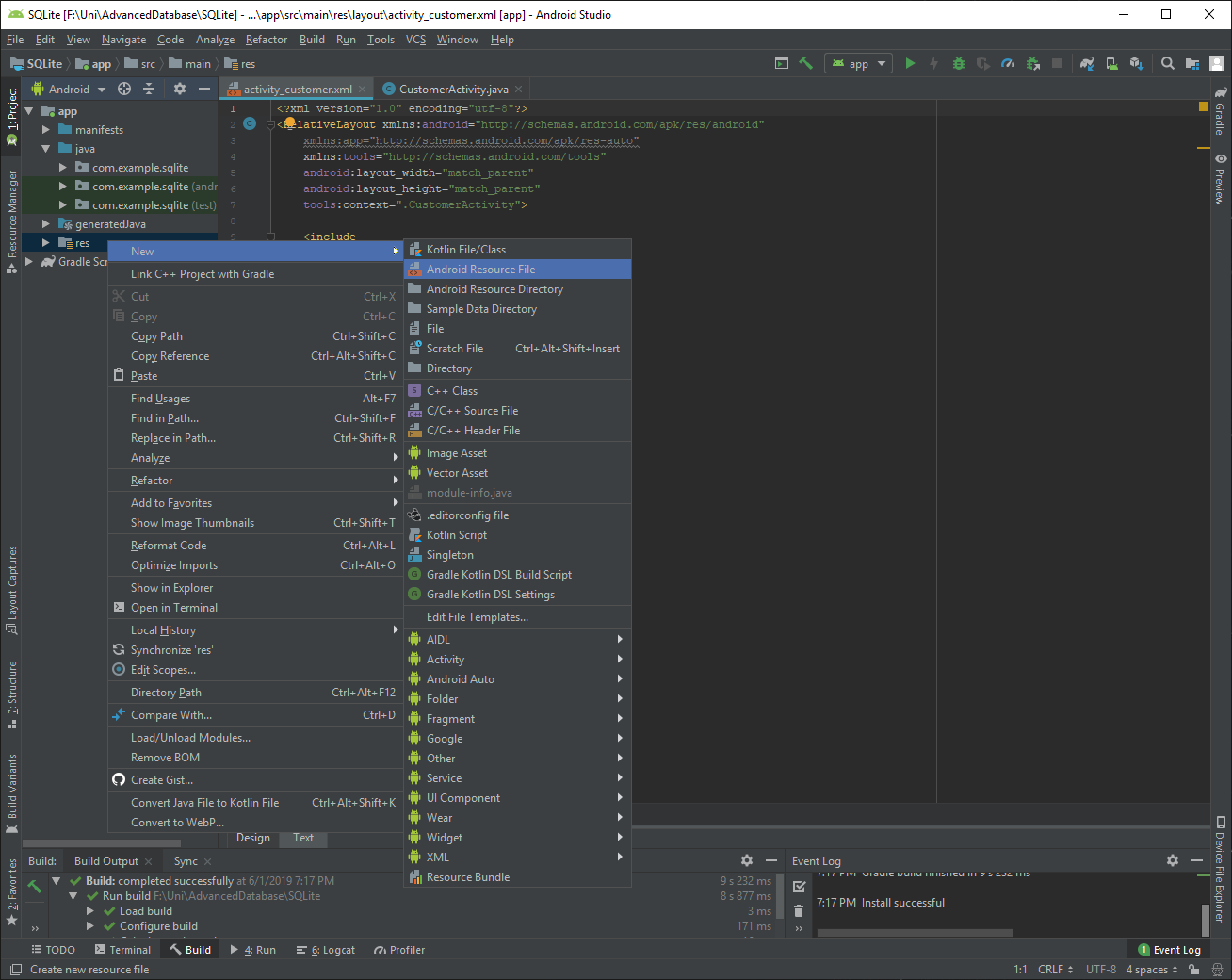


1. We will now add some string resources to our application. Text that we want to display in our app is written as a resource which we will then reference.
   1. Open strings.xml resource file. Under app -> res -> values -> strings.xml.
   2. Add the following string items.

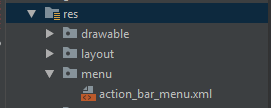


* 1. The key we will reference is “menuCustomer”. The value returned will be “Customers”. Close strings.xml.

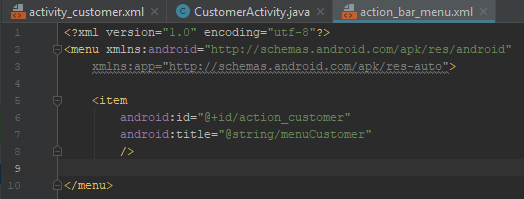
1. It is time to create the menu for our action bar. Eventually we will have an item in the menu for every activity of the application. For now, we will create a single item for our customer activity.
   1. First, we create a menu resource file. In the project view under app -> res, right click res. Select New -> Android Resource File.



* 1. Name the resource file “action\_bar\_menu”. Change the resource type from ‘Values’ to ‘Menu’. OK.
  2. Under res, the menu package now exists and contains action\_bar\_menu.xml. Open the resource file.



* 1. Add an item for the customer menu option.



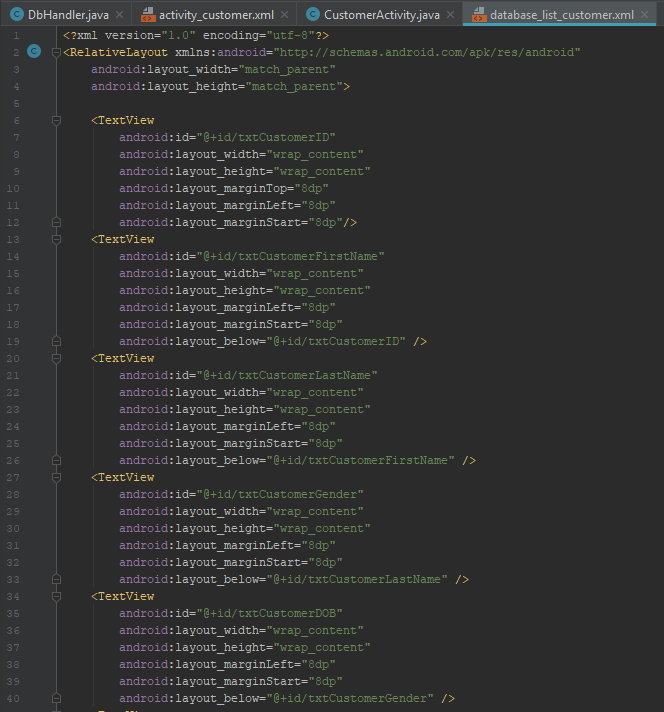
* 1. Note that we are referencing the string resource ‘menuCustomer’. Our menu item will display “Customers”.

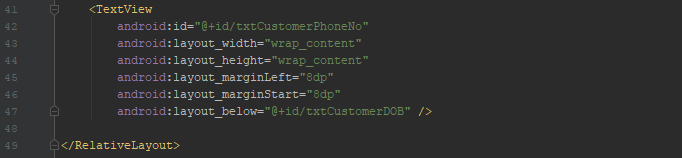
1. Open AndroidManifest.xml again. We will add a label to the action bar so we now what activity we are viewing when the app is running.
   1. Open ‘AndroidManifest.xml’. In project view under app -> manifests -> AndroidManifest.xml.
   2. Find CustomerActivity and add a label that references the string resource ‘menuCustomer’.



* 1. We will add a label for each activity we create. Close AndroidManifest.xml.

1. The information we will display is retrieved by the return data methods we wrote in step 10. Each method returns multiple values per row. We will create a layout resource file for each activity page we create, that will organize the information returned. For example, getCustomers method returns the customer id, first name, last name, gender, date of birth, and phone number.
   1. Create a new layout resource file like we did in step 13 f. Name the resource file “database\_list\_customer”.
   2. Create a TextView for each returned item.
   3. Each TextView has a unique id. Give the id a name that corresponds to the returned information.
   4. Display the TextView items below each other.

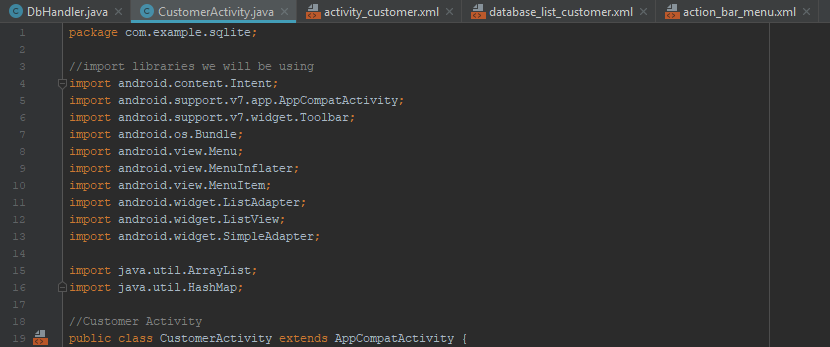




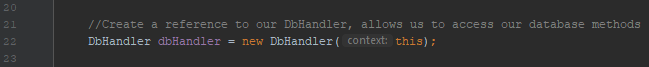
1. It is time to write the functionality (Java code) of CustomerActivity. All files we have will be brought together in this step. We will be using:
   * DbHandler methods:
     + checkToPopulateTables
     + getCustomers
   * activity\_customer.xml
   * action\_bar.xml
   * database\_list\_customer.xml
   * action\_bar\_menu.xml

Have open CustomerActivity.java, DbHandler.java, activity\_customer.xml, database\_list\_customer.xml, and action\_bar\_menu.xml. The following actions of step 18 all take place within CustomerActivity.

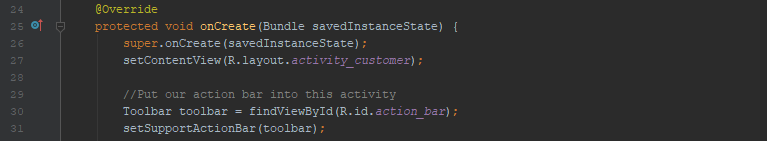
* 1. We will be accessing some code and functionality provided by Android Studio. We will import the corresponding libraries.



* 1. We will instantiate an object of DbHandler. We gain access to DbHandler methods from CustomerActivity.



* 1. Inside onCreate (the method that came with the activity), add our action bar to this activity.



* 1. Inside onCreate, call the DbHandler method, checkToPopulateTables. We will need to use the object we created in step 18 b. Note that this method only needs to be called once from one activity.



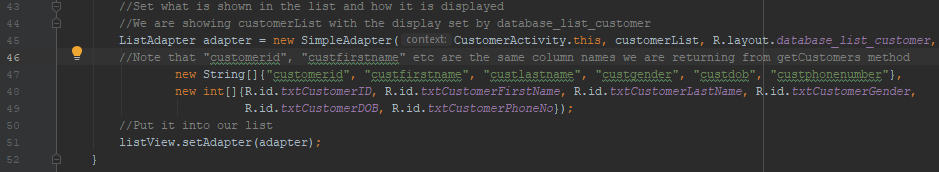
* 1. Inside onCreate, call the DbHandler method, getCustomers. Store the returned customer information in an ArrayList of HasMaps called customerList.



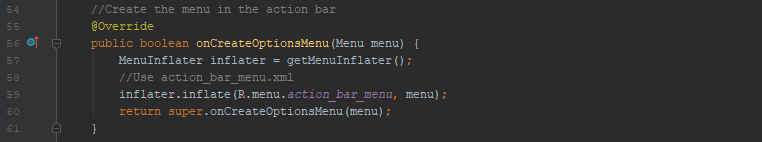
* 1. Inside onCreate, reference the ListView item from activity\_customer.xml.



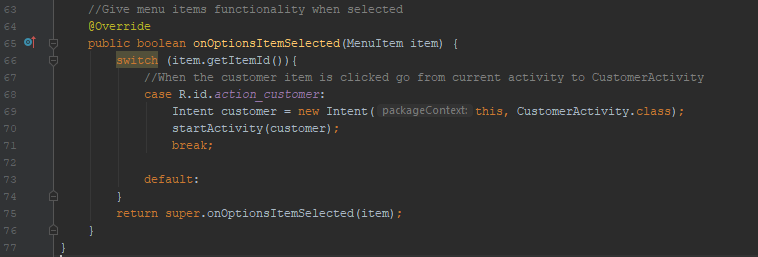
* 1. Inside onCreate, set what information is shown in the list and how it is displayed.



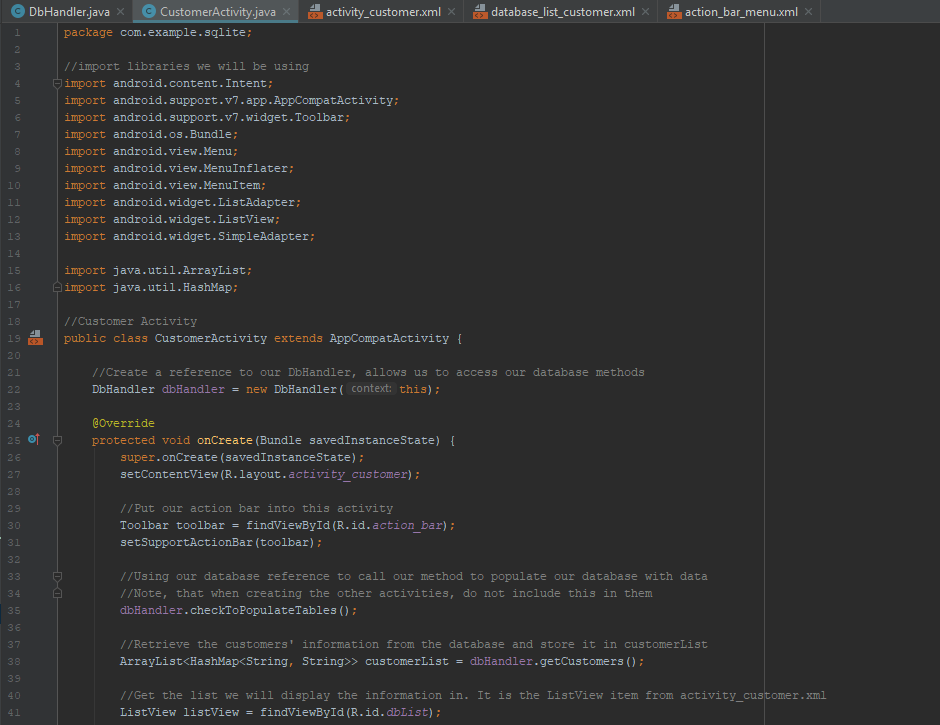
* 1. For our menu items to be displayed and work correctly we must add two additional methods under onCreate (still within CustomerActivity class). Create onCreateOptionsMenu to create the menu in the action bar using action\_bar\_menu.xml.



* 1. Create onOptionsItemSelected to give functionality to menu items when they are selected. For each additional activity created you will add a menu item in action\_bar\_menu.xml. Be sure to return to this method and add functionality to each menu item added.



* 1. Ensure your CustomerActivity matches the below image.

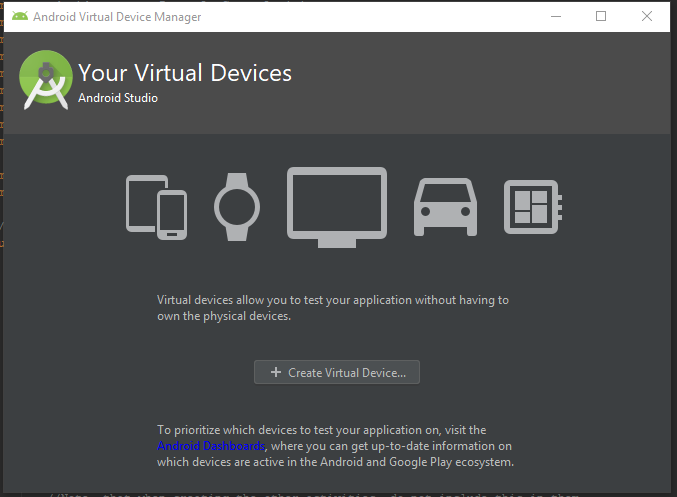




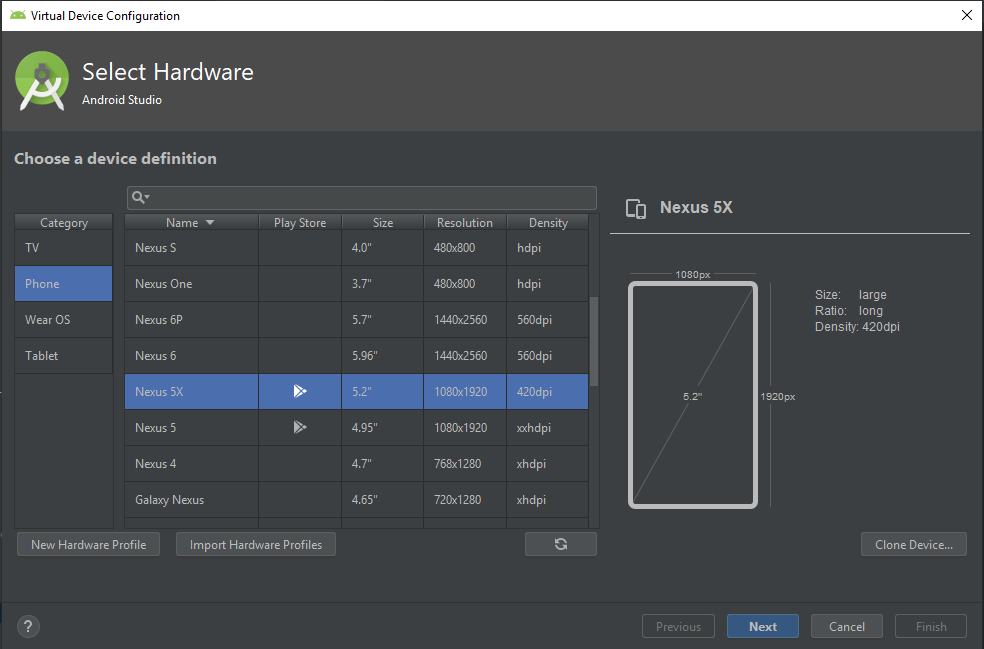
1. To test our application is working, we will create an emulator.
   1. Tools -> AVD Manager.



* 1. Select ‘Create Virtual Device’.



* 1. Select the Nexus 5X phone. Next.



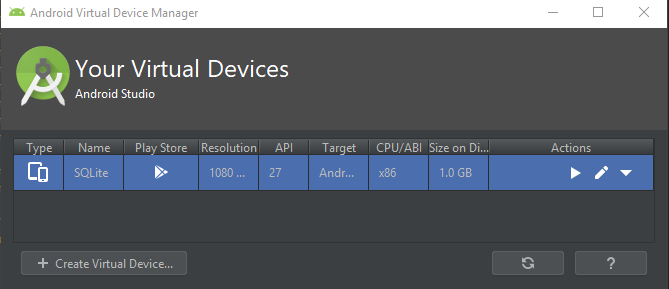
* 1. Select Oreo, API Level 27. Next. You may need to download the image. Just follow the prompts.



* 1. Call the AVD “SQLite”. Finish.



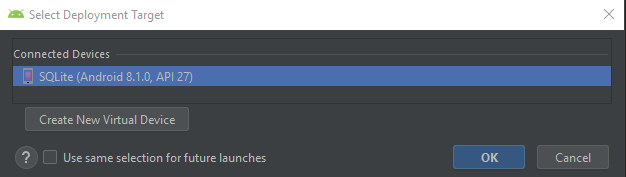
1. Our emulator is created. Press the play button under ‘Actions’ to launch the emulator. Starting it for the first time can take a few minutes. If the emulator fails to start and returns errors, there are configuration settings that need to modified. They usually relate to the BIOS and virtualization. Google Android Studio emulator errors and follow the steps. They are well documented. Close the Android Virtual Device Manager.



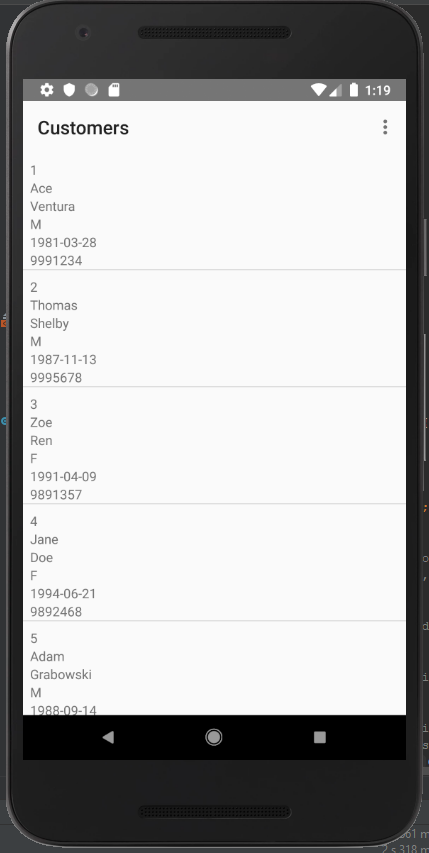
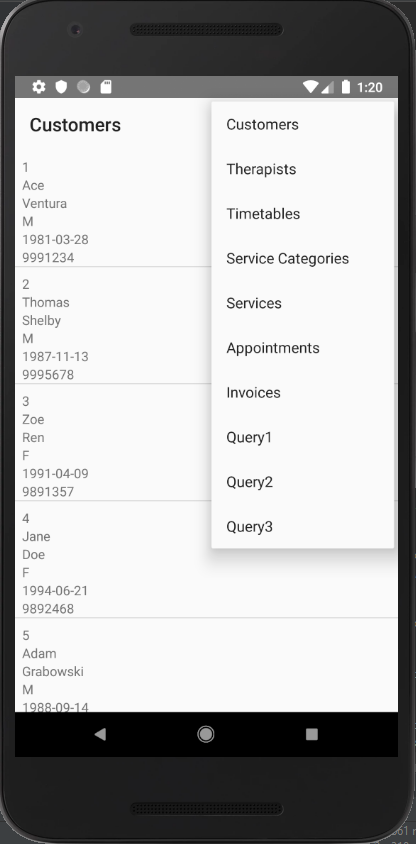
* 1. Once the emulator is open, return to Android Studio. Run -> Run ‘app’.



* 1. Our SQLite emulator should be highlighted under ‘Connected Devices’. OK.

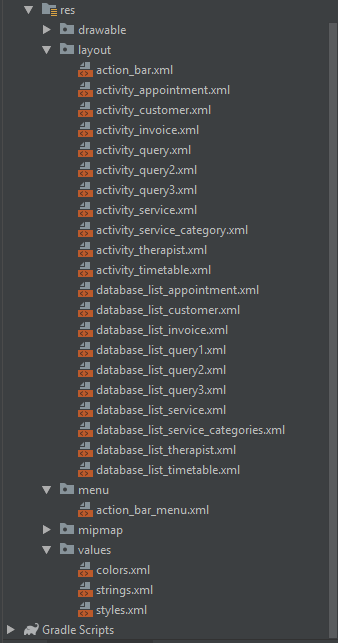
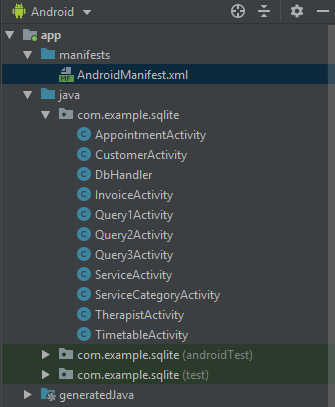


* 1. We see our customer information. There are six entries. The label we added in the manifest reads ‘Customers’. We can see the menu (three dots) in the top right corner. Select the menu to view the items of our action\_bar\_menu.xml.

* 1. Note my menu contains an item for each table and three additional queries.

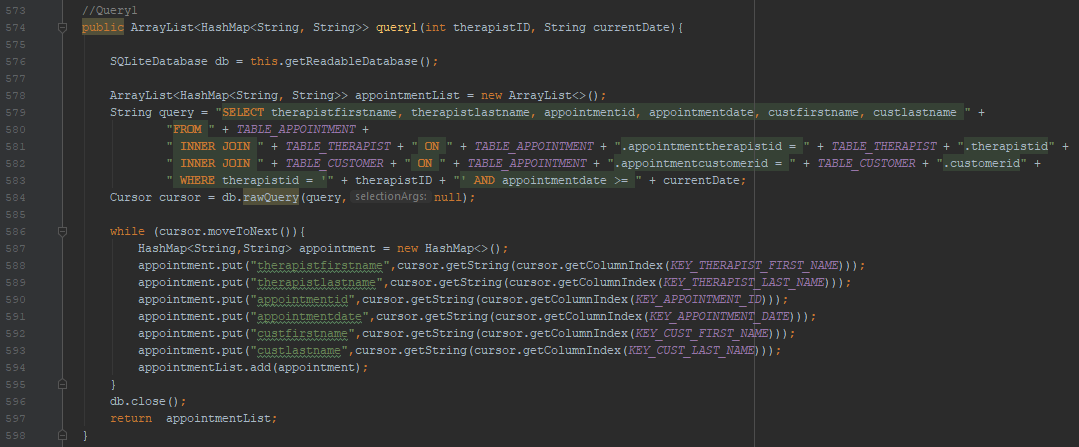
1. Take what we have learnt and create an activity with all supporting resource files for each table.



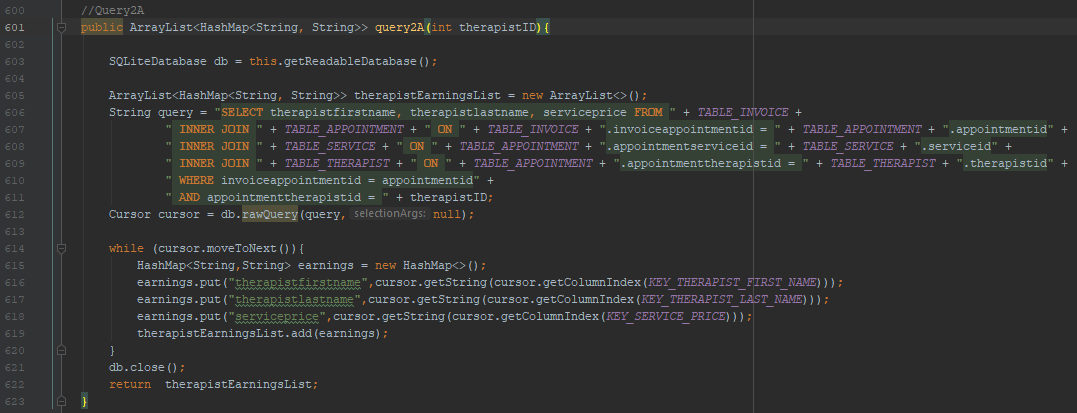
Refer to the completed SQLite Android Studio application that I have included in this GitHub repository, for the code of each activity and resource file.

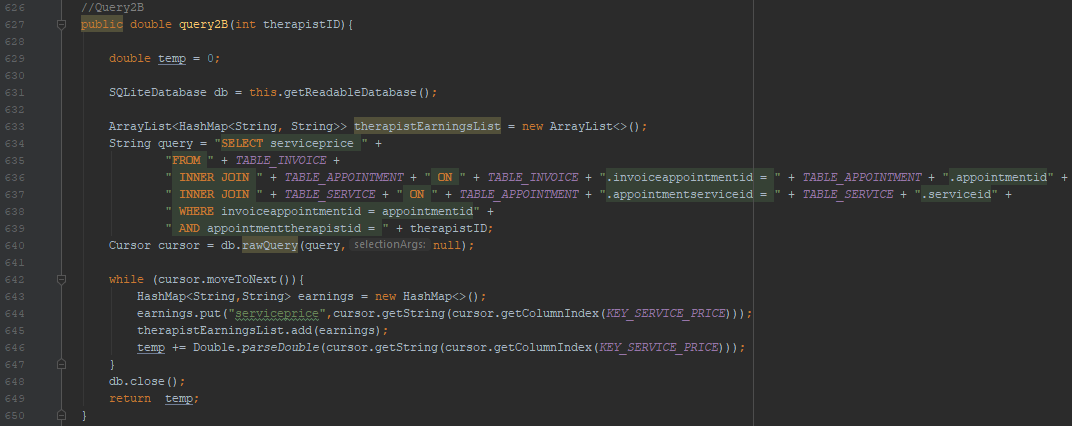
# Non-Trivial Queries

1. What upcoming scheduled appointments does a therapist have. Display to the therapist their name, appointment id, appointment date, and the customer’s name. A therapist inputs their id and the system inputs the current date.

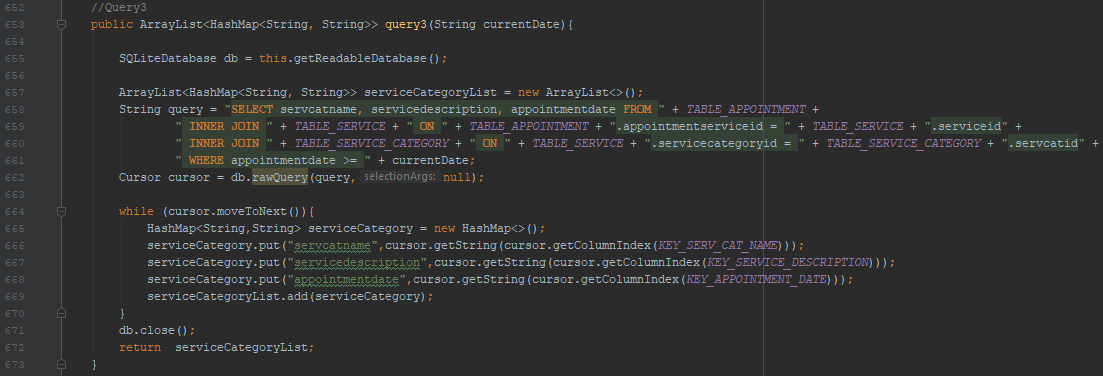


1. A therapist can view how much money their invoiced appointments have earnt for MyFace Day Spa. A second query returns the totaled sum of the appointments.





1. MyFace management is thinking of introducing a new service. They want to know what the most popular service category is currently. Input a date to see the services performed from the date. Returns service category, the service, and when it was performed.



# References

* Rittmeyer, W. (2013, March 28). *SQLite in Android*. Grokking Android. Retrieved from <https://www.grokkingandroid.com/sqlite-in-android/>
* Tutorials Point. (n.d.). *SQLite*. Retrieved from <https://www.tutorialspoint.com/sqlite/index.htm>
* Tutlane. (n.d.). *SQLite Tutorial*. Retrieved from <https://www.tutlane.com/tutorial/sqlite>