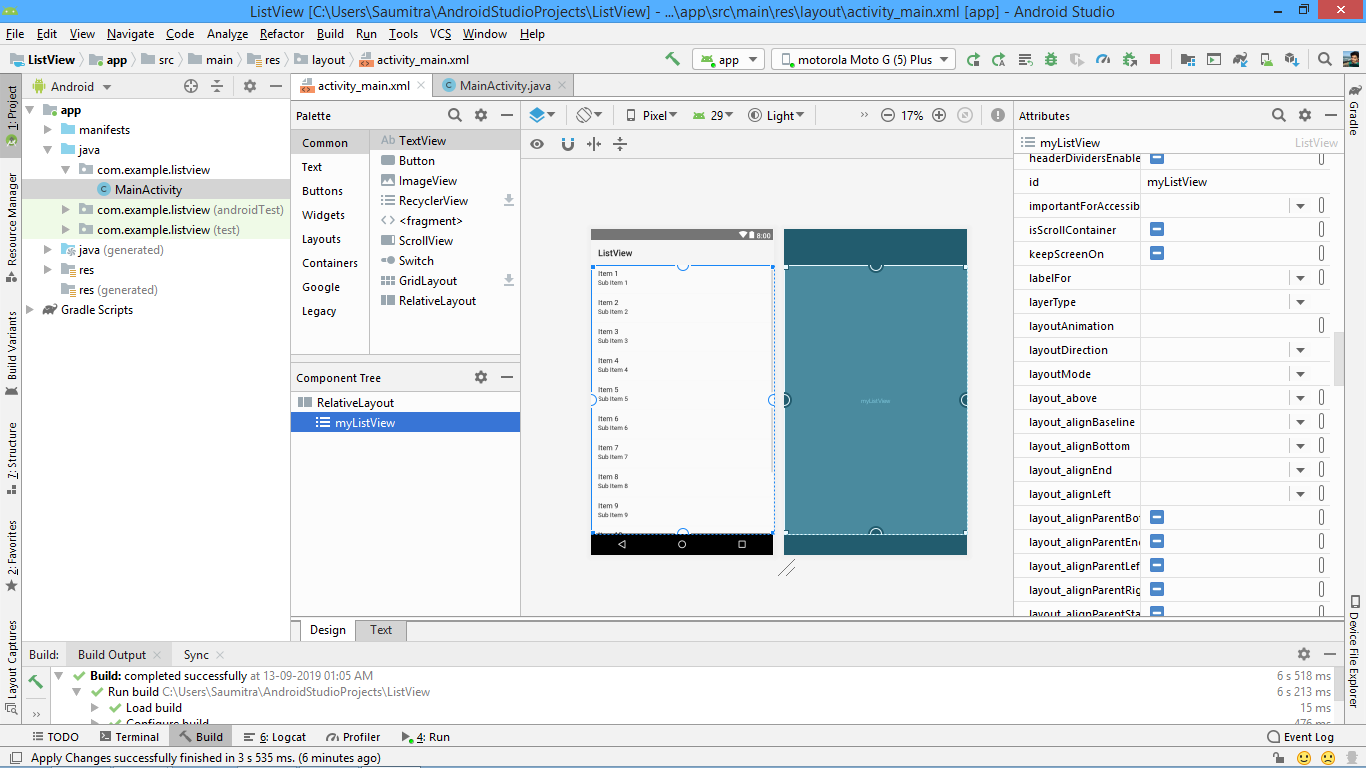
LIST VIEWS:

(Refer app: ListView)

ListViews are a very important concept in android UI design. All the reddit posts you see, all the different tweets that are laid out, the 9gag memes, the Spotify songs, etc are all laid out in a listview. For now, let us see how to create a simple text listview that will be used to display an array of strings. For this, we simply add a listview in our layout. For simplicity, we have chosen a relative layout.



For the java file, we shall import the following:

**import** android.view.View;  
**import** android.widget.AdapterView;  
**import** android.widget.ArrayAdapter;  
**import** android.widget.ListAdapter;  
**import** android.widget.ListView;

We shall create a simple array of strings in the onCreate method.

String dogs[]={**"Most aggressive dog breeds ranked:"**,**"14. You"**,**"13. cannot"**, **"12. rank"**,**"11. them"**,**"10. because"**,**"9. aggression"**,**"8. isn't"**,**"7. breed"**,**"6. specific"**,**"5. it's"**,**"4. a"**,**"3. learned"**,**"2. behaviour"**,**"1. Chihuahuas"**};

Now, for listviews, it is not possible to simply attach arrays directly into the listview. We have to set an adapter which will convert the array of strings into a form that is compatible with the listview. This is because in most practical examples a listview is not just a collection of strings, it can actually be a collection of text, images, videos,etc.

We create a reference of the ListAdapter class and set it to an object of the ArrayAdapter class. Its constructor will accept 3 parameters, first is the current context of the application (we pass “this” as the parameter).

The second is how you want the list to be formatted. When we are using custom listviews, we need not use the default values provided, but here we shall use the default format of the listview and set it equal to android.R.layout.simple\_list\_view\_1.

The 3rd parameter is the list of strings (the array of strings that we created).

ListAdapter dogadapter= **new** ArrayAdapter<String>(**this**,android.R.layout.***simple\_list\_item\_1***,dogs);

Then we create a reference to our listview that we have created in our UI xml file.

ListView myListView= (ListView)findViewById(R.id.***myListView***);

Then we set the adapter of the listview to the adapter that we just created.

myListView.setAdapter(dogadapter);

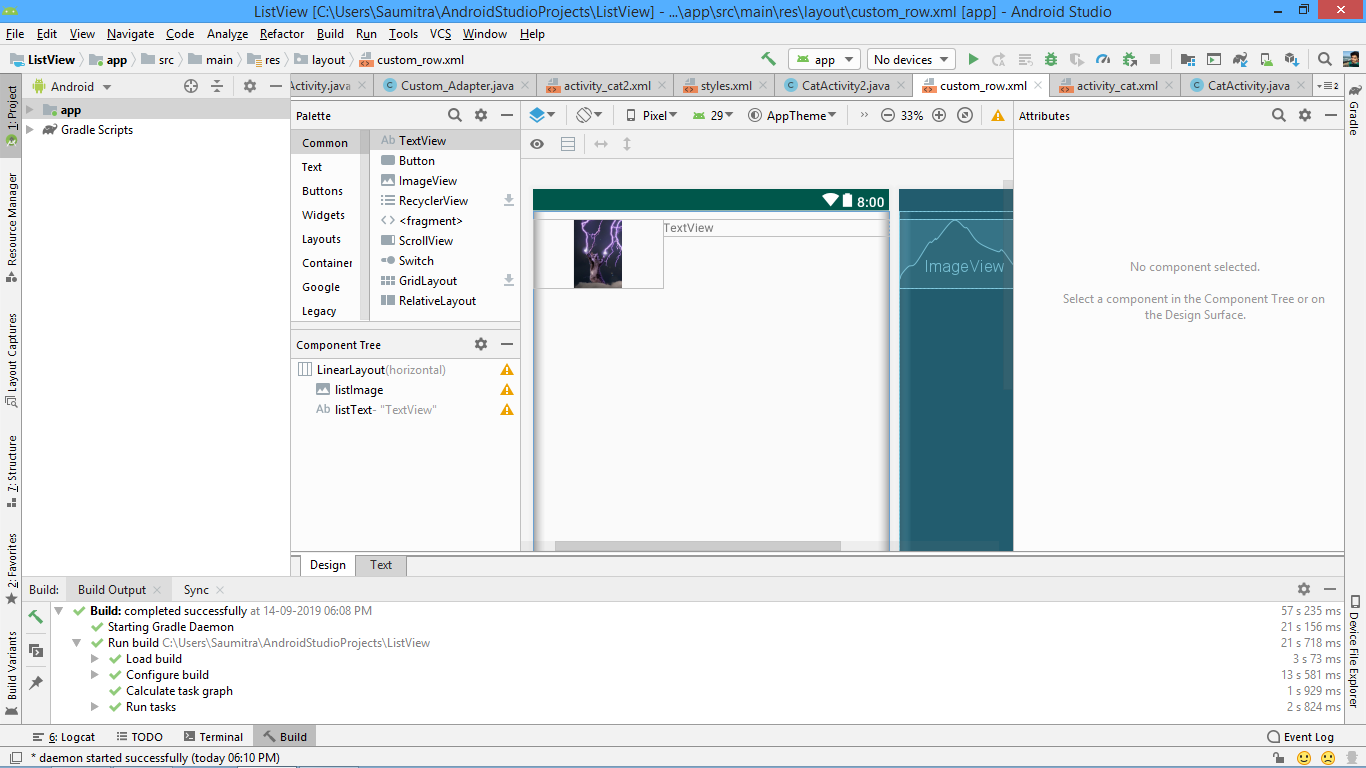
CUSTOM LIST VIEWS:

Till now we have seen the normal ListView where a bunch of text was just arranged in the form of a list. Now, however lets move on to the interesting part. We can create a custom row for our listview. This row can contain anything, from images, to videos, to buttons, you name it.

This can be done by simply creating an xml file which defines the structure of ONE row, and then creating a custom adapter that inflates the layout of the xml file so that it can be assigned to the ListView. So, crack your knuckles, and lets get on to it.

1. CREATING THE XML CUSTOM ROW:

We need to create a new xml file by right clicking the res->layout folder and clicking the new->android resource file.



For the purposes of this example, we’ve gone with a simple design for the custom row… With just an imageview and a textview. The imageview also contains only one image, but we can change it if we want for each row.

The actual xml code looks like:

*<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 android:orientation="horizontal" android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"**>  
  
 <**ImageView  
 android:id="@+id/listImage"  
 android:layout\_width="150dp"  
 android:layout\_height="80dp"  
 app:srcCompat="@drawable/catstorm2"  
 android:layout\_marginTop="10dp"**/>  
  
 <**TextView  
 android:id="@+id/listText"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_weight="1"  
 android:layout\_marginTop="10dp"  
 android:text="TextView"** />  
</**LinearLayout**>

1. CREATING THE CUSTOM ADAPTER FOR OUR CUSTOM ROW:

We now need to create our own custom adapter for our row. For this we need to create a new java file by right clicking the java-> (your\_package\_name) folder and selecting new->java class.

We copy all our imports from the earlier file. We need our main class to extend the ArrayAdapter class. This ArrayAdapter is actually a collection class (like arraylist, stack, queue, vector, hashmap,etc.), so we need to specify the datatype within angular brackets. This datatype is actually the datatype of the raw data that we want to use in our customised row. In this case, our array is a string, so we specify <String>.

**class** Custom\_Adapter **extends** ArrayAdapter <String> {

After we have extended the required ArrayAdapter class, it is compulsory for us to define the constructor. We also need to alter the list parameter of the constructor to a String array. REMEMBER, that we HAVEN’T PASSED THE ARRAY YET. So, we can have ANY NAME FOR THIS ARRAY as it is actually a parameter.

This constructor passes 3 key elements to ITS superclass.

1. The context i.e. the current state of the application. This is used for housekeeping purposes. This is present by default when we create the constructor using alt+insert and selecting constructor.
2. The resource file which we want our layout to look like (ie the xml file that we created just now.) R.Layout.custom\_row
3. The array that we will want to use for the layout. In this case because we have specified the ArrayAdapter class to be a string, we need an array of string type. This parameter is nothing but the array that we have specified in the constructor declaration.

**public** Custom\_Adapter(@NonNull Context context, String dogs[]) {  
  
  
 **super**(context,R.layout.***custom\_row***, dogs);  
  
  
}

We also need to override another method called getView. This method is used to tweak the finer aspects of the custom row of our listView.

We need to create an object of the LayoutInflater class. This class is used to convert the elements of the xml file into java objects (a process called “inflating” a view). We set it equal to LayoutInflater.from(getContext()). As discussed the context is used for housekeeping purposes.

We create a new object of the View class so that we can refer to our custom row. This view gets all its properties after the LayoutInflater has inflated the specified the required the view. The inflate() method returns the inflated view object.

We create a String variable called singleDogItem and set it equal to the String returned by the getItem method. This method is implicitly present in the ArrayAdapter class. It returns the item present at the position specified in the method. This position is present in the parameters of the getView method (the method we’re writing all this into).

The rest part is easy. We simply create objects for the textview and imageview. We then assign the text of the textView as the singleDogItem string. We also set the imageResource as the image that we want from the Res.Drawable folder.

**public** View getView(**int** position, @Nullable View convertView, @NonNull ViewGroup parent) {  
 LayoutInflater myInflater= LayoutInflater.*from*(getContext());  
 View customView= myInflater.inflate(R.layout.***custom\_row***,parent,**false**);  
  
 String singleDogItem=getItem(position);  
 TextView listText=customView.findViewById(R.id.***listText***);  
 ImageView listImage=customView.findViewById(R.id.***listImage***);  
 listImage.setImageResource(R.drawable.***catstorm2***);  
  
 listText.setText(singleDogItem);  
  
 **return** customView;  
  
 }  
}

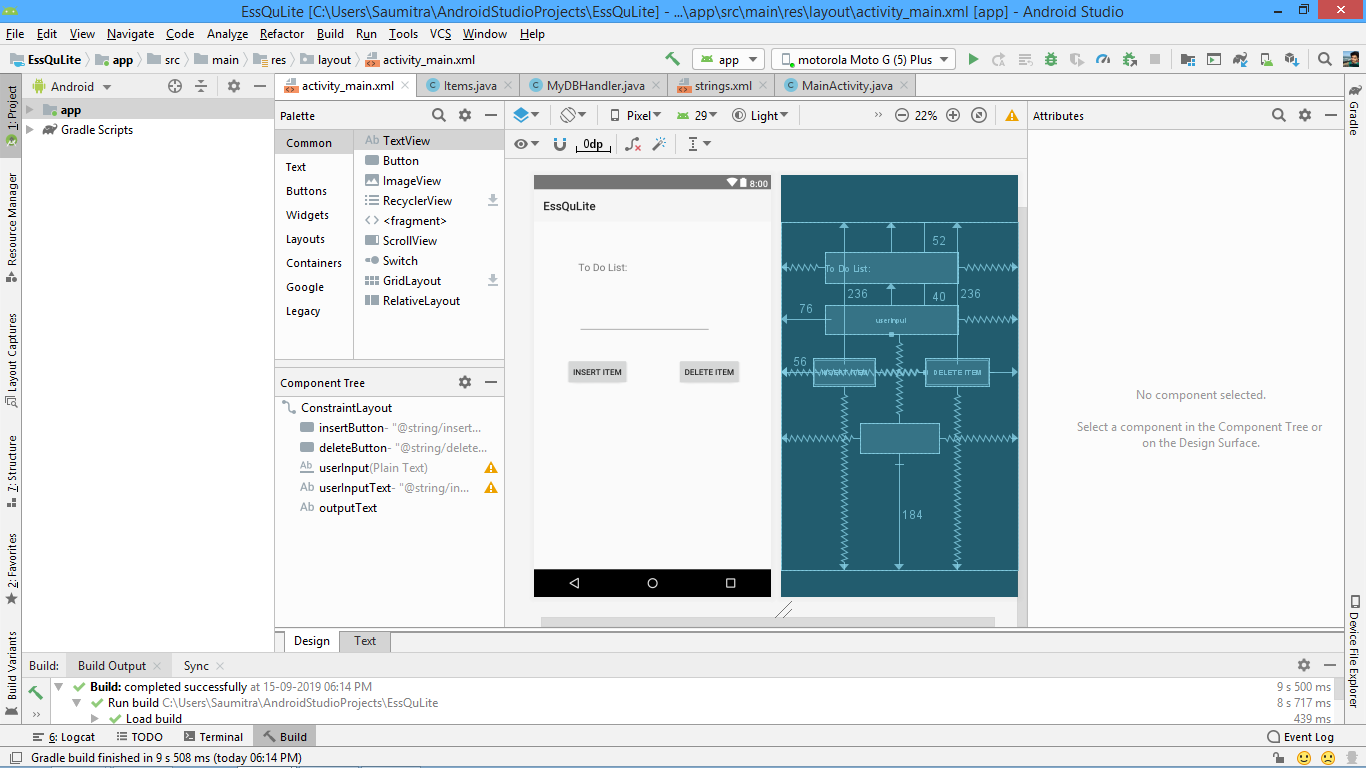
SAVING DATA WITH THE HELP OF SQLite DATABASE:

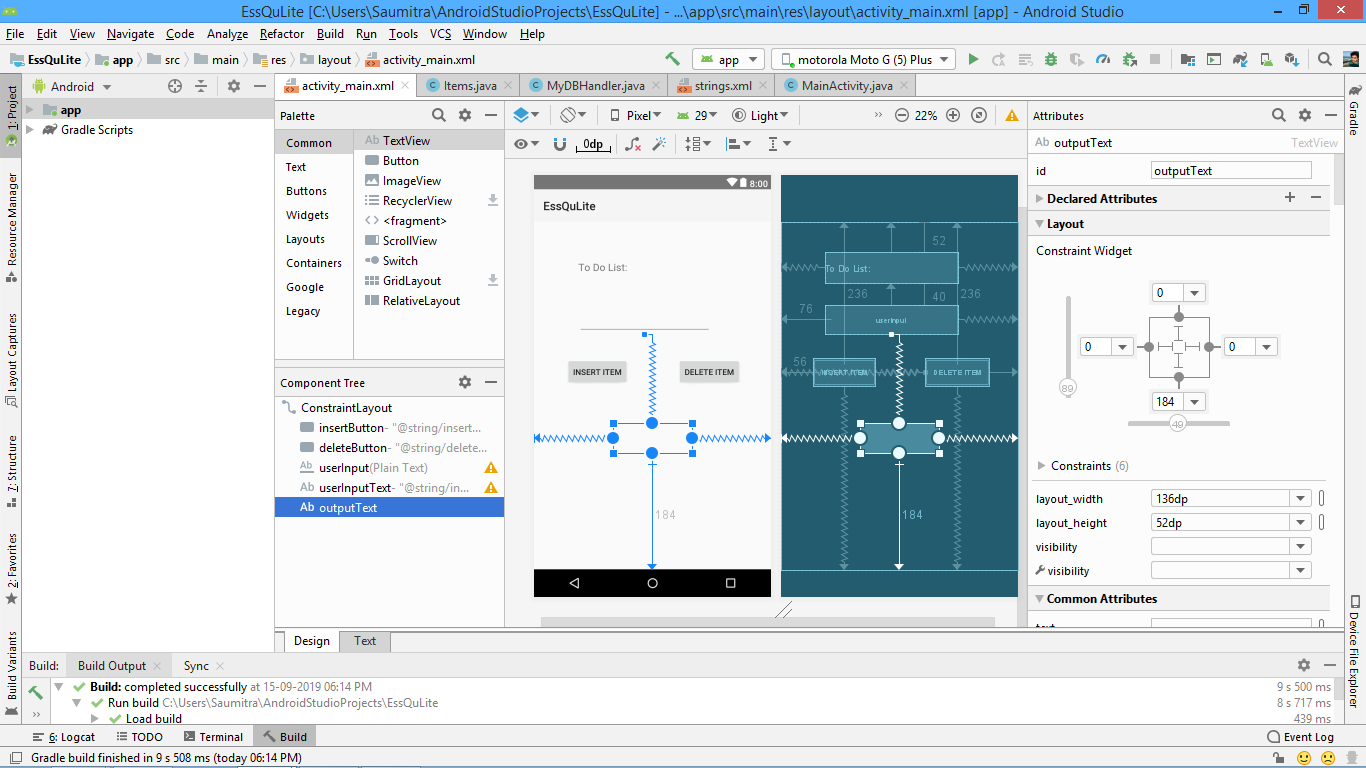
Till now we have seen applications where data was stored temporarily in the device and was lost when the app was restarted. This can be solved by permanently storing data in the secondary memory of the device. Using SQLite in Android is very similar to using JDBC in Java. We have an SQLiteOpenHelper which makes many of the actions easier.

We here are making a simple app where we have a simple table with just 2 columns.

1. The id of the item
2. The name of the item

For this we shall create a simple UI where we have an editText where the user inputs his data which will be inserted into the SQL table. An insert data button, and a delete data button. And a final TextView which will display the items inserted into the table.





Let’s now move on to the java files. We now make a new java file which contains the int and String variables for the id and the itemName.

NOTE: We could’ve simply declared an int and String variable in the mainActivity.java file…. But doing so just makes it look cleaner.

Now, we have these variables:

**public class** Items {  
  
 **private int \_id**;  
 **private** String **\_itemName**;

Now, in order to immediately assign values to the variables as soon as the object is created, we need to have a constructor.

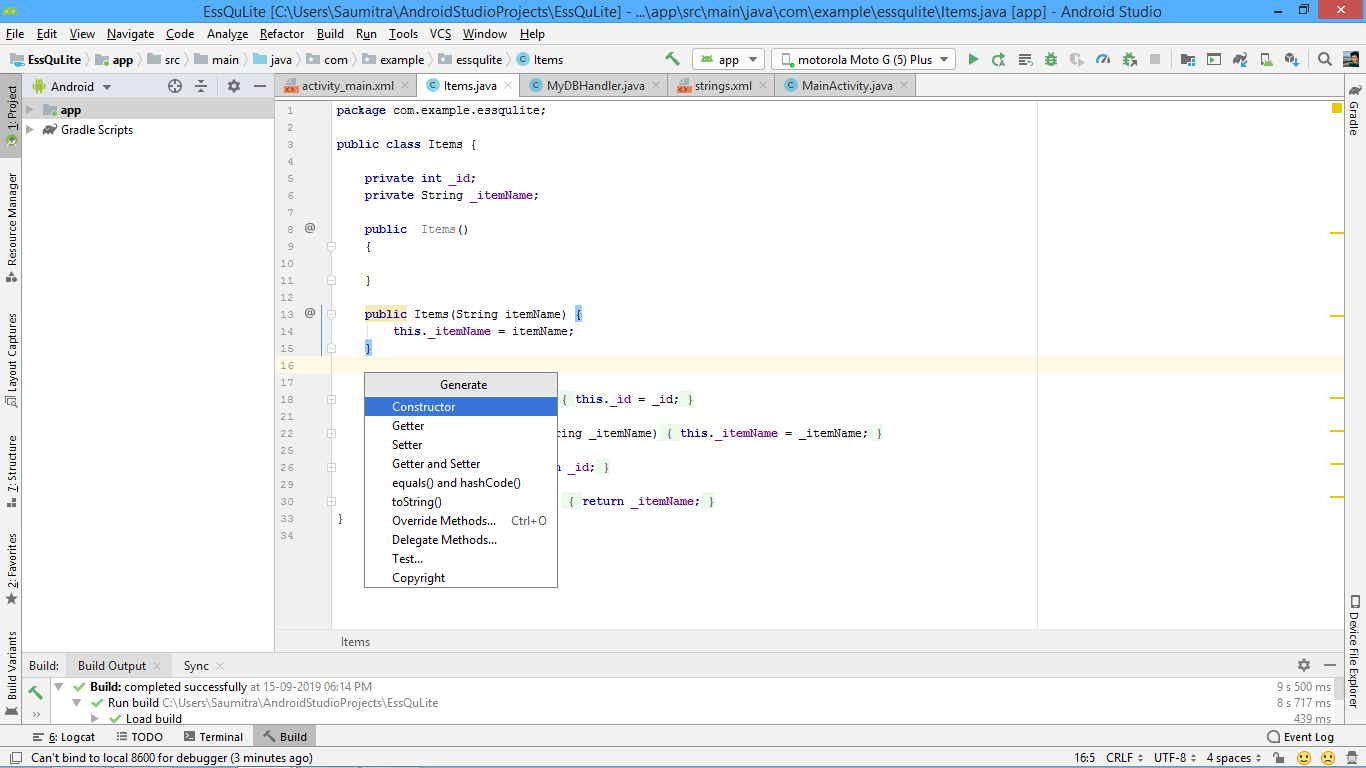
**public** Items(String itemName) {  
 **this**.**\_itemName** = itemName;  
}

In order to give the programmer flexibility to first create a blank object and then LATER assign values to the variables, we also create a blank constructor.

**public** Items()  
{  
  
}

NOW, because the variables \_id and \_itemName are declared PRIVATE, we need explicit getter and setter methods in order to access these variables from outside the class. This is done automatically by Android Studio. Just press alt+insert and then select getter and setter.

The getter methods will automatically return the variables while setter methods will set the values of the variables which are passed into the methods.



And that’s it for this Java file.

We now create a new Java class. This will handle all the database operations of the application. This class essentially acts as a database handler, for this reason it is called MyDBHandler.java

We import the following packages:

**import** android.database.sqlite.SQLiteDatabase;  
**import** android.database.sqlite.SQLiteOpenHelper;  
**import** android.database.Cursor;  
**import** android.content.Context;  
**import** android.content.ContentValues;

The SQLiteOpenHelper is a very useful class. This provides a JDBC like feeling whenever we are dealing with the database.

By default we just have the main class. In order to use the database operations, we need this class to extend the SQLiteOpenHelper class. On doing so, we get an error. This is because we need to perform these 3 actions:

1. Create a constructor
2. Provide implementation of the onCreate (database) method
3. Provide implementation of the onUpgrade(database) method

Before that, lets create some variables which will save us a ton of typing later on.

**private static final int *DATABASE\_VERSION***=1;  
**private static final** String ***DATABASE\_NAME***=**"items.db"**;  
**public static final** String ***TABLE\_ITEMS***=**"myitems"**;  
**public static final** String ***COLUMN\_ID***=**"\_id"**;  
**public static final** String ***COLUMN\_ITEMNAME***=**"ItemName"**;

The DATABASE\_VERSION is used to specify which revision of the database we are using. Suppose in the future, we add a column to the table, we then change the database version to 2. This becomes especially useful in the onUpgrade method. For the purposes of this application, we simply set it to 1.

The database name as the name suggests, is the name of the database. It is good programming practice to end the name of the database with a .db extension. This makes us easier to identify it in the system files. The table name within the database is called myitems and is stored in the variable TABLE\_ITEMS. By doing so, it saves us the hassle of typing the table name again and again.

Similarly we name the ID column as “\_id” and save it in the variable COLUMN\_ID and we name the Item name column as “ItemName” and save it in the variable COLUMN\_ITEMNAME.

We then create the required constructor. Using the alt+insert functionality and selecting the constructor option.

**public** MyDBHandler(@Nullable Context context, @Nullable String name, @Nullable SQLiteDatabase.CursorFactory factory, **int** version) {  
 **super**(context, ***DATABASE\_NAME***, factory, ***DATABASE\_VERSION***);  
}

It is important for the constructor to pass the name of the database and the version to its superclass. The context and factory variable is just background information. We pass the DATABASE\_NAME and DATABASE\_VERSION into the super keyword.

We then override 2 methods:

1. The onCreate method:

This method specifies what the system must do when the database is created for the first time.

And, what do we want to do after creating the database? Yes! We need to create the table. So we create a string variable and type the standard SQL create table query. And use the execSQL method provided by the SQLiteOpenHelper.

@Override  
**public void** onCreate(SQLiteDatabase db) {  
  
 String query=**"create table "**+***TABLE\_ITEMS***+**"("**+  
 ***COLUMN\_ID***+**" INTEGER PRIMARY KEY AUTOINCREMENT,"**+  
 ***COLUMN\_ITEMNAME***+**" TEXT "**+  
 **");"**;  
  
 db.execSQL(query);*//Much like statement.execute in JDBC*}

Notice that we have assigned the ID column as the primary key and have also set it as autoincrement. So, as soon as any new item is added to the list, SQL automatically takes care of the id column. For any future tasks, we needn’t worry about the ID column.

1. The onUpgrade method

This method comes into use when we make changes to the database structure( eg add columns, delete columns, etc). Essentially whenever we execute any DDL (Database Definition Language statement).

Whenever we update the table, we probably have some new code in the variable declarations (some new column names probably) and the onCreate method. We essentially drop the existing table, and then create a new one using the code inside the onCreate method. So, we execute an SQL drop table query and then simply call the onCreate method.

@Override  
**public void** onUpgrade(SQLiteDatabase db, **int** oldVersion, **int** newVersion) {  
  
 db.execSQL(**"DROP TABLE IF EXISTS "**+ ***TABLE\_ITEMS***); *//If our Database changes structure, we drop the existing one and create a new, updated one* onCreate(db);  
  
}

We now need to write the methods which will insert and delete records into our table. We create a method which takes in an object of the Items class that we created earlier. So, its essentially accepts a packet of values, opens up the packet and sets the values of the packet into the database.

We now need to learn about a new way of inserting values into the SQLite database. We take the help of an object of the ContentValues class. This works very similar to the prepared statement object.

It contains the put method which is used to put values into columns one by one, by using the COLUMN\_NAME as one parameter and then the value as another parameter.

**public void** addNewItems(Items items)  
{  
 ContentValues values=**new** ContentValues(); *//Much like using the preparedstatement in JDBC  
 //We are essentially creating a capsule of values and then inserting it into the database in bulk.  
 //As \_id is autoincrement, we need'nt worry about it.* values.put(***COLUMN\_ITEMNAME***,items.get\_itemName());  
 *//Smart! Takes in the COLUMN NAME as well as the data to be inserted into that COLUMN NAME*

Till now, we just have created a capsule with of values that we INTEND to insert into the table. We now need a reference to the database variable. So we create a new object of the SQLiteDatabase and set it equal to getReadableDatabase(). This will anchor the variable to the database that we are currently using.

We now have a new method of inserting values into the database. Instead of writing the entire query, we simply use the insert method of the db object. This method accepts 3 parameters: The table name, the nullcolumn hack(what to write for null column values), and the object of the ContentValues class which now contains the required information to be inserted.

SQLiteDatabase db=getReadableDatabase();  
 db.insert(***TABLE\_ITEMS***,**null**,values);  
 db.close(); *//Housekeeping stuff to return memory back to android.*}

To delete items from the table, we read the user input from the editText and then delete the row whose item column matches the entered string.

**public void** deleteItems(String itemName)  
{  
 SQLiteDatabase db=getReadableDatabase();  
 db.execSQL(**"DELETE FROM "**+ ***TABLE\_ITEMS*** +**" WHERE "**+ ***COLUMN\_ITEMNAME***+ **"= \""**+itemName+**"\";"**); *// '\' is used as an escape character*}

CONVERTING THE DATABASE VALUES INTO A STRING:

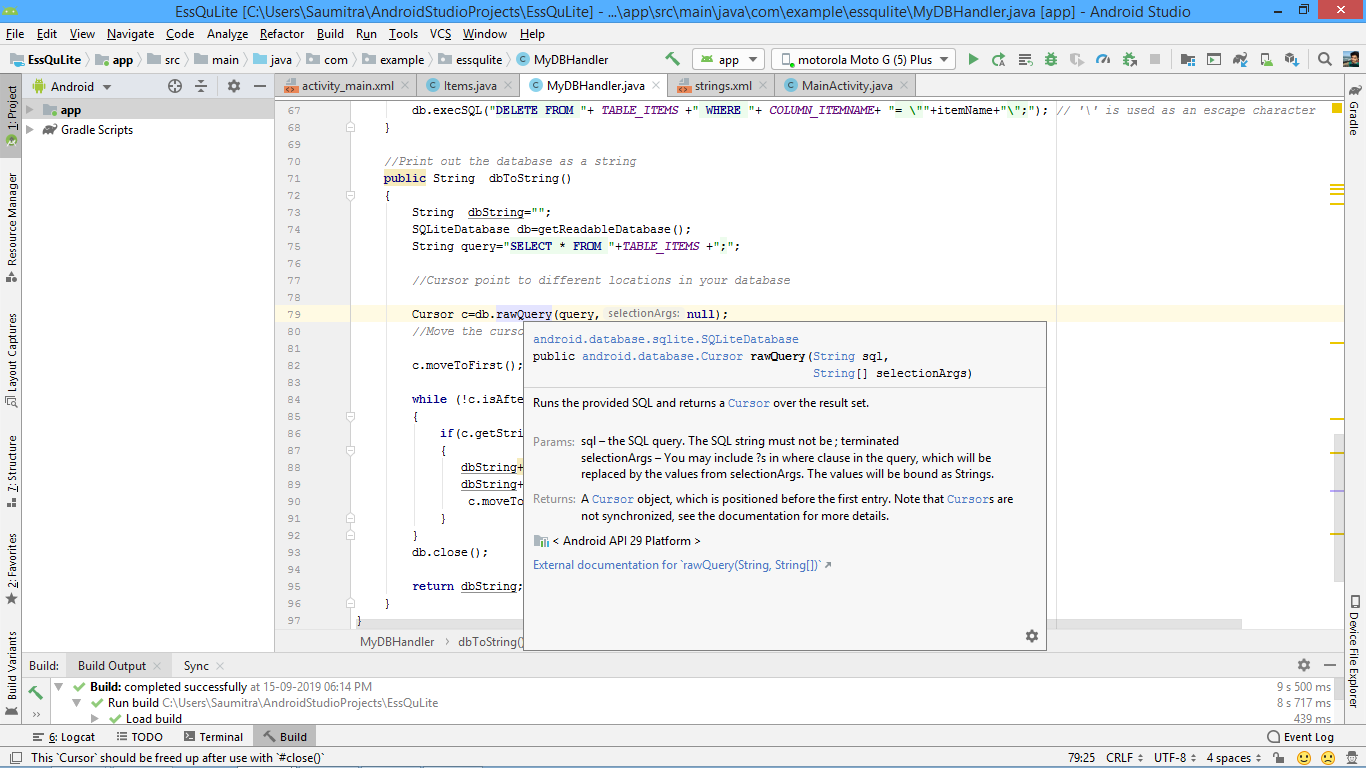
We need to use cursors for this. A cursor is nothing but a pointer that points to different elements of your mySQL table.

We first create an empty string that shall later store the value that we obtained from the table. We shall also get a reference to our database by using an object of the SQLiteDatabase class and setting it equal to getReadableDatabase method.We also create a string variable which is equal to the “SELECT \* FROM (TABLE\_NAME)” query.

**public** String dbToString()  
{  
 String dbString=**""**;  
 SQLiteDatabase db=getReadableDatabase();  
 String query=**"SELECT \* FROM "**+***TABLE\_ITEMS*** +**";"**;

*//Cursor point to different locations in your database*Cursor c=db.rawQuery(query,**null**);  
*//Move the cursor to the first row of the results*

We then create an object of the Cursor class. We set it equal to a method called db.rawQuery() method.



As described above, this method returns a cursor over the result set. The selection arguments are the values of the “?” elements which might be present in the query. (Much like preparedStatement). Because we don’t have any “?” in our query, we set it equal to null.

Cursor c=db.rawQuery(query,**null**);

We then move the cursor to the first element of the ResultSet, using the movetoFirst() method.

c.moveToFirst();

We set up a while loop which shall terminate when the cursor has reached the end of the resultSet. We append the string present at the columnIndex of the Item Name column of the table to the empty string variable called “dbString” that we had created earlier alomg with a new line (“\n”).

**while** (!c.isAfterLast())  
 {  
 **if**(c.getString(c.getColumnIndex(***COLUMN\_ITEMNAME***))!=**null**)  
 {  
 dbString+= c.getString(c.getColumnIndex(***COLUMN\_ITEMNAME***));  
 dbString+=**"\n"**;  
 c.moveToNext();  
 }  
 }  
 db.close();  
  
 **return** dbString;  
}

Finally we return the string that we have created here.

We now return to the MainActivity.java class where we shall integrate our database handler with the UI and the user input. We first create references to our editTexts, textViews and buttons.

EditText **userInput**;  
Button **insertButton**;  
Button **deleteButton**;  
TextView **outputText**;

We also create a reference variable for the database handler java class that we created just now.

MyDBHandler **dbHandler**;

We assign the ids to all the reference variables of the UI elements.

**userInput**=(EditText)findViewById(R.id.***userInput***);  
**insertButton**=(Button)findViewById(R.id.***insertButton***);  
**deleteButton**=(Button) findViewById(R.id.***deleteButton***);  
**outputText**=(TextView)findViewById(R.id.***outputText***);

Recall the constructor that we created in the DatabaseHandler java class. We had to specify the Database Version, the Database name, the context, and the factory cursor. So, while creating an object of the database handler class, we need to specify all these values here. BUT, HERE COMES THE CATCH. We have already passed the correct database name, and the database version INSIDE THE CONSTRUCTOR of the database handler class to the superclass (which is essentially the only thing the constructor does in this case).

So, we needn’t specify them again here. We can pass null values here as they are already taken care of inside the constructor.

**dbHandler**=**new** MyDBHandler(getApplicationContext(),**null**,**null**,1);

We now create a function called printDatabase() which essentially sets the text of the textView to the value returned by the database to string method (dbToString) in the database handler. It also resets the editText of the user Input to a blank value so that the user needn’t clear the text he has just entered manually.

**void** printDatabase()  
{  
 String dbString;  
 dbString=**dbHandler**.dbToString();  
 **outputText**.setText(dbString);  
 **userInput**.setText(**""**);  
}

We simply call this method in the onCreate method, as the first thing we want the app to do is to display the already previously entered database.

printDatabase();

We then write an onClickListener for the insert button. Here we create an empty object and pass the user input from the editText as the parameter into the constructor. We then call the addNewItems method of the database handler so that the user input is stored inside the database. It accepts an object of the Items class, which we just created and assigned the itemName in the constructor.

We finally call the printDatabase method so that the output is refreshed with the newly inserted values.

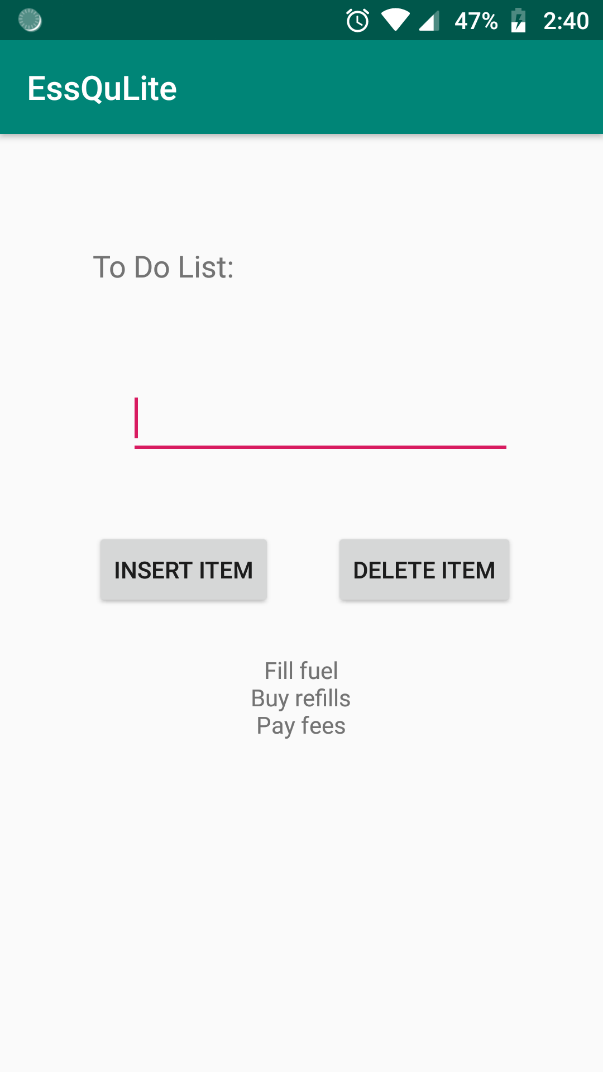
**insertButton**.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 Items item=**new** Items(**userInput**.getText().toString());  
 **dbHandler**.addNewItems(item);  
 printDatabase();  
 }  
});

We then write an onClickListener for the delete button. We simply accept the user input in a string and pass this string in the deleteItems method of the database handler. This will delete the required string from the database.

We finally call the printDatabase method so that the output is refreshed barring the newly deleted values.

**deleteButton**.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 String itemToDelete=**userInput**.getText().toString();  
 **dbHandler**.deleteItems(itemToDelete);  
 printDatabase();  
 }  
});

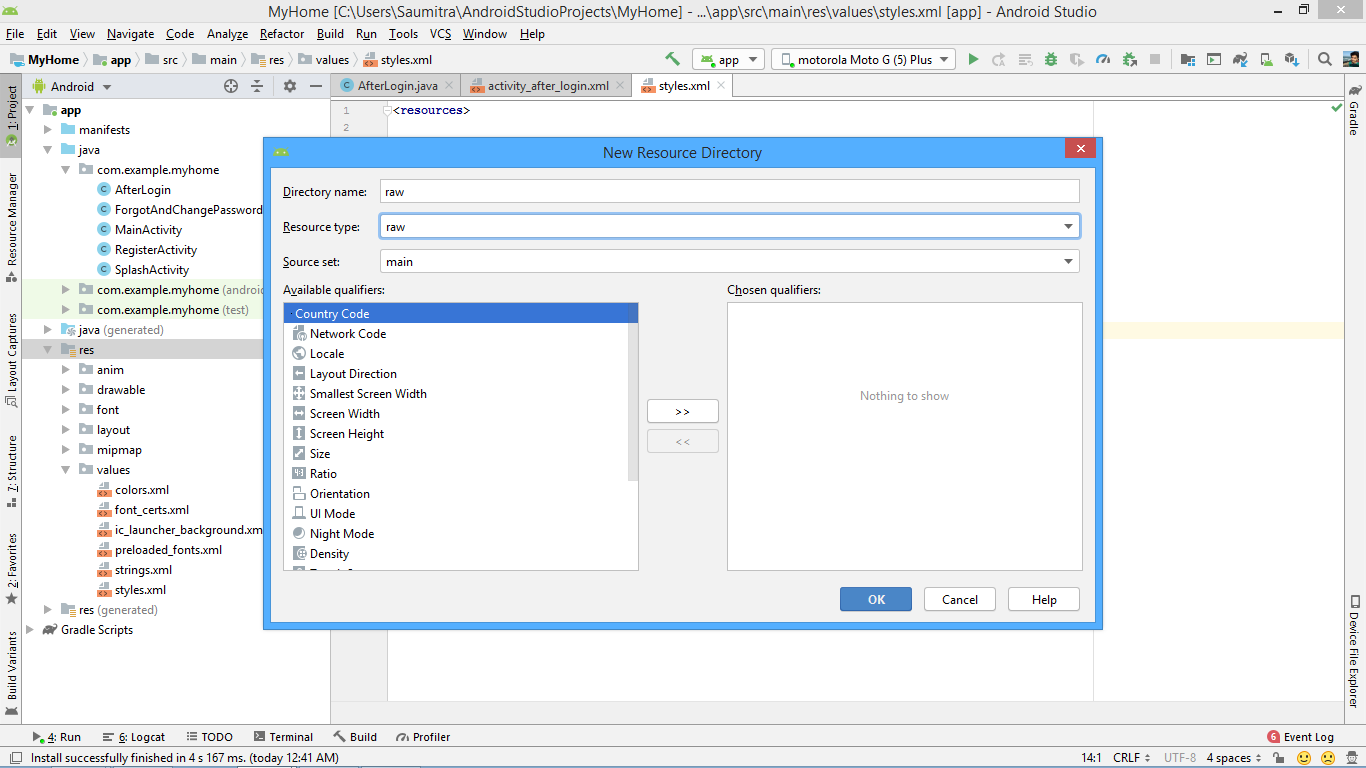
Aand that’s it! The program is ready for execution.



VIDEO PLAYER:

Most of this app is implemented by default in Android Studio. All you need to do is to create a new directory called “raw” under your resources directory.

We just copy our video file in this directory.



All we need to do is to create a new widget called “VideoView” in our layout file. This widget will have attributes match-parent for both width and height—indicating that it fills the entire screen.

THAT’S ALL FOR THE LAYOUT(Sweet isn’t it?)

We now turn to our java file. Here we create a new instance of our VideoView object and inflate it by using the findViewbyID method. We also need an object of the MediaController class. This object controls the playback of all media that is playing throughout the device.

**final** VideoView myVideoView= (VideoView)findViewById(R.id.***myvideoView***);  
MediaController mycontroller=**new** MediaController(**this**);

We also need an object of the URI(uniform resource identifier) class. It returns a string reference of any resource that we might need in our application.

We set the uri object equal to the following:

Uri uri=Uri.*parse*(**"android.resource://"**+getPackageName()+**"/"**+R.raw.***meme***);

This just specifies the path of the video file and parses the resource into a string value.

Then we set the videoURI of the VideoView to the uri object that we just created. It essentially “tells” the videoView that you have to play THIS particular video file.

myVideoView.setVideoURI(uri);

The mediacontroller now has the ability to play, pause, seek,fastforward, fastrewind,etc the video playback. In order for the mediacontroller to show all these functions, we need to “tell” it where to anchor the set of play/pause/seek buttons. This will of course be our videoView.

mycontroller.setAnchorView(myVideoView);

We however have not connected the video to be played to the mediacontroller. Thus, it currently doesn’t know which video to play/pause,etc. So, we need our videoView to set ITS mediacontroller to our mediacontroller object.

myVideoView.setMediaController(mycontroller);

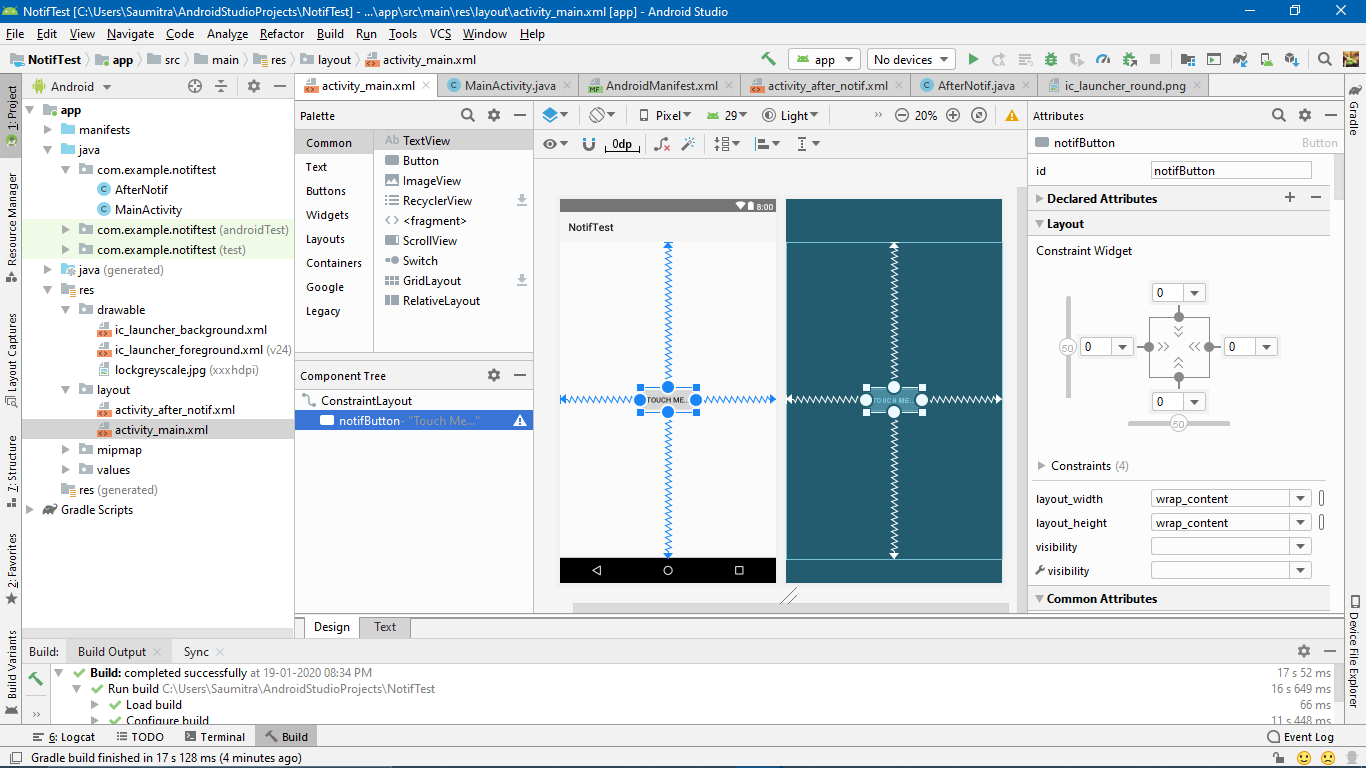
Finally, we start video playback using:

myVideoView.start();

NOTIFICATIONS: (REFER APP: NotifTest)

The manner of introducing android notifications is changed since android oreo. There are a few extra steps that we need to do, in order to ensure that the notifications are sent properly.

We have our usual stuff—an xml file, and a java file. The xml file is very simple. We just have the system issue a notification once the user clicks on a button. So, naturally, the xml will have only a single button.



The text will look like:

*<?***xml version="1.0" encoding="utf-8"***?>*<**androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 tools:context=".MainActivity"**>  
  
 <**Button  
 android:id="@+id/notifButton"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Touch Me..."  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent"** />  
</**androidx.constraintlayout.widget.ConstraintLayout**>

THE JAVA FILE:

First we shall define a unique id for our notification. This id can be anything, but must be unique for different notifications. We shall also have a reference to our button:

**public class** MainActivity **extends** AppCompatActivity {  
  
 **private static final int *uniqueID***=123456;  
 Button **notifbutton**;  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 **notifbutton**=(Button)findViewById(R.id.***notifButton***);

createNotificationChannel();//Explained below

}

}

Here we shall have 2 methods:

1. A method to create notification channels. This has been made compulsory for phones with OS greater than oreo.
2. A method to actually trigger the notification.

CREATE NOTIFICATION CHANNEL:

This is a filtering mechanism that allows us to segregate different notifications of our app into categories. There are a few steps to create a notification channel.

We first establish a check to see if the android version is oreo and above.

We then need an object of the notification channel, and need to pass parameters as the “id” of the channel, its name, and its importance (low, default, high, min, max). We then add a description of the channel. Then we also toggle the badge settings—whether to show a badge on the app icon whenever a new notification occurs. We then finally create the notification channel.

**private void** createNotificationChannel() {  
 **if**(Build.VERSION.***SDK\_INT***>=Build.VERSION\_CODES.***O***)  
 {  
 NotificationChannel notificationChannel= **new** NotificationChannel(**"Default"**,**"MyNotificationChannel"**,NotificationManager.***IMPORTANCE\_DEFAULT***);  
 notificationChannel.setDescription(**"MyDescription"**);  
 notificationChannel.setShowBadge(**true**);  
 NotificationManager notificationManager=getSystemService(NotificationManager.**class**);  
 notificationManager.createNotificationChannel(notificationChannel);  
 }  
}

TRIGGER NOTIFICATION:

We’ll first create an intent which will, specify which activity to launch after the user clicks the notification. We then pass this intent to a pending intent. This is because, a notification is a part of the OS and not limited to our app. Therefore we need to pass its handling to the OS, using a pending intent.

**private void** triggerNotification() {  
  
  
 Intent intent=**new** Intent(**this**,AfterNotif.**class**);  
 intent.setFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK***| Intent.***FLAG\_ACTIVITY\_CLEAR\_TASK***);  
 PendingIntent pendingIntent=PendingIntent.*getActivity*(**this**,0,intent,0);

Now we actually build the notification. We specify different aspects as to how the notification will actually look, like the small icon, the large icon, the title, the body, the time,etc… The following fields will be self explanatory…

NotificationCompat.Builder notification;  
notification=**new** NotificationCompat.Builder(**this**, **"Default"**);  
notification.setSmallIcon(R.drawable.***ic\_launcher\_foreground***);  
notification.setContentTitle(**"Hey Man!"**);  
notification.setContentText(**"Wear your mom..."**);  
notification.setPriority(NotificationCompat.***PRIORITY\_DEFAULT***);  
notification.setContentIntent(pendingIntent);  
notification.setChannelId(**"Default"**);  
notification.setOngoing(**true**);*//Notification cannot be swiped unless you click on it*notification.setAutoCancel(**true**); *//Notification cancels when you click it or it redirects you to the app*notification.setWhen(System.*currentTimeMillis*());  
  
  
NotificationManagerCompat notificationManagerCompat=NotificationManagerCompat.*from*(**this**);  
notificationManagerCompat.notify(***uniqueID***,notification.build());

Our entire method looks like:

**private void** triggerNotification() {  
  
  
 Intent intent=**new** Intent(**this**,AfterNotif.**class**);  
 intent.setFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK***| Intent.***FLAG\_ACTIVITY\_CLEAR\_TASK***);  
 PendingIntent pendingIntent=PendingIntent.*getActivity*(**this**,0,intent,0);  
  
 NotificationCompat.Builder notification;  
 notification=**new** NotificationCompat.Builder(**this**, **"Default"**);  
 notification.setSmallIcon(R.drawable.***ic\_launcher\_foreground***);  
 notification.setContentTitle(**"Hey Man!"**);  
 notification.setContentText(**"Wear your mom..."**);  
 notification.setPriority(NotificationCompat.***PRIORITY\_DEFAULT***);  
 notification.setContentIntent(pendingIntent);  
 notification.setChannelId(**"Default"**);  
 notification.setOngoing(**true**);*//Notification cannot be swiped unless you click on it* notification.setAutoCancel(**true**); *//Notification cancels when you click it or it redirects you to the app* notification.setWhen(System.*currentTimeMillis*());  
  
  
 NotificationManagerCompat notificationManagerCompat=NotificationManagerCompat.*from*(**this**);  
 notificationManagerCompat.notify(***uniqueID***,notification.build());  
}

Then, we shall call this method in the onClickListener for our button:

**notifbutton**.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
  
 triggerNotification();  
  
  
  
 }  
  
  
});