1. To **Change the Size of ImageView**:

Ex: in the XML File You Add Under the ImageView

<ImageView  
 android:id="@+id/imageView"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent"  
 app:srcCompat="@drawable/background"  
 android:scaleType="fitXY"/>

**Note**: This Will Make **the Image Expand** to **Take All the Screen**

EX: to make the image **Take All the Screen** **Without Expanding**

<ImageView  
 android:id="@+id/imageView"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:scaleType="fitXY"  
 android:adjustViewBounds="true"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent"  
 app:srcCompat="@drawable/art\_background" />

**Note**: For **More Ways** Visit this [WebSite](https://guides.codepath.com/android/Working-with-the-ImageView)

1. **Save in SQLite**:
2. Create **DBHelper class**:

class DBHelper(  
 context: Context?,  
 name: String?= "details.db",  
 factory: SQLiteDatabase.CursorFactory?= null,  
 version: Int= 1,  
 private val tableName: String= "notes"  
)

: SQLiteOpenHelper(context, name, factory, version) {  
 override fun onCreate(p0: SQLiteDatabase?) {  
 p0?.execSQL("create table $tableName (Note Text)")  
 }  
  
 override fun onUpgrade(p0: SQLiteDatabase?, p1: Int, p2: Int) {}  
  
 fun saveNotes(note: String): Long {  
 val cv= ContentValues()  
 cv.put("Note",note)  
 return *writableDatabase*.insert(tableName,null,cv)  
 }  
}

1. **In Main Class**:

val dbHelper= DBHelper(this)

val check= dbHelper.saveNotes(noteEntry.*text*.toString())  
val wrongCode: Long= -1  
if (check != wrongCode) {  
 StyleableToast.makeText(this, "Saved Successfully!!\n$check", R.style.*mytoast*).show()  
}  
else  
 StyleableToast.makeText(this,"Something Went Wrong!!\n$check",R.style.*mytoast*).show()

1. The **Deference** between (**this** and **applicationContext**) when we need to use Context:

If we are going to use it **inside the same activity** it’s ok to use (**this**)

**But** if we need to **send it to another function** or **another class**, better to use (**applicationContext**).

Because if we pass the activity context and that activity got destroyed the **app will crash**.

1. **Read from SQLite**
2. In DBHelper class

fun gettingNotes(): ArrayList<String>{  
 return try{  
 val notes= *arrayListOf*<String>()  
 val cursor =  
 sqLiteDatabase.query(tableName, *arrayOf*("Note"), "Note=?", *arrayOf*("Good"), null, null, null)  
 cursor.moveToFirst()  
 while (!cursor.*isAfterLast*){  
 notes.add(cursor.getString(0))  
 cursor.moveToNext()  
 }  
 notes  
 } catch (e:Exception){  
 *arrayListOf*("Error")  
 }  
}

**Note: SQL Query** = select Note from tablenName where Note= Good

Note: If you want to **select all the table** you can replace all with null (Except table name)

Ex: val cursor =  
 sqLiteDatabase.query(tableName, null ,null, null, null, null, null)

Note: You can Replace the index from **set number to index that the program locate**

EX: notes.add(cursor.getString(cursor.getColumnIndexOrThrow("Note")))

1. **In Main Class**

private lateinit var notes: ArrayList<String>

notes= dpHlpr.gettingNotes()

**Note: Another Way to Read The Entire Table**

val selectQuery = "SELECT \* FROM $tableName "var cursor: Cursor? = null  
try {  
 cursor = sqLiteDatabase.rawQuery(selectQuery, null)  
} catch (e: SQLiteException){  
 return ArrayList()  
}  
var noteText: String  
if(cursor.moveToFirst()){  
 do {  
 noteText = cursor.getString(cursor.getColumnIndexOrThrow("Note"))  
 noteList.add(noteText)  
 } while (cursor.moveToNext())  
}

1. **Delete From SQLite**

fun deleteNotes(pk: Int): Int{  
 return sqLiteDatabase.delete(tableName,"PK=?", *arrayOf*("$pk"))  
}

1. **Edit To SQLite**

fun updateNotes(pk:Int, newNote: String): Int{  
 val contentValue= ContentValues()  
 contentValue.put("Note",newNote)  
 return sqLiteDatabase.update(tableName,contentValue,"PK=?", *arrayOf*("$pk"))  
}

1. If we **Upgrade the version** to change the table columns or add new columns we make sure to **change onUpgrade:**

override fun onCreate(p0: SQLiteDatabase?) {  
 p0?.execSQL("create table $tableName (PK INTEGER PRIMARY KEY AUTOINCREMENT, Note Text)")  
}  
  
override fun onUpgrade(p0: SQLiteDatabase?, p1: Int, p2: Int) {  
 p0?.execSQL("DROP TABLE IF EXISTS $tableName")  
 onCreate(p0)  
}

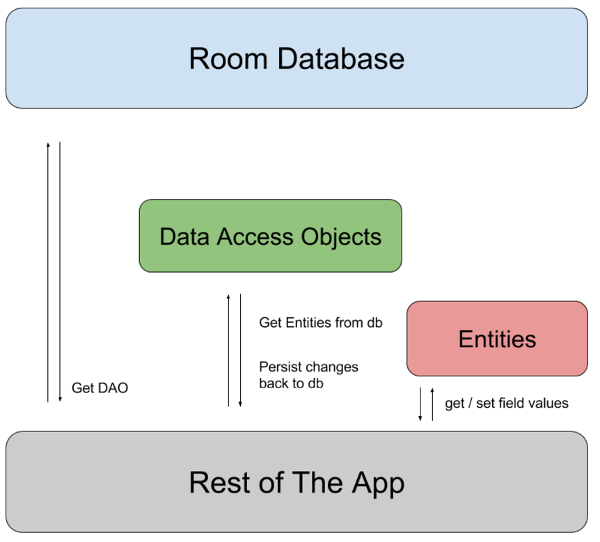
1. **Room**

Room is a persistence library and an abstraction layer over SQLite.

Room annotations (@Entity, @Dao, @Database ) make it easier to save data to SQLite.

Room verifies SQL queries during compile time, which means there will never be any runtime errors due to query syntax.

While it may take some time to get used to the setup, once you are familiar with it, you will notice that room makes using SQL much easier.  It also makes integration with other Android architectural components easier.  This will become more evident when we look at View Models and Live Data.



Source Code: <https://github.com/KPRanjithKumar/SQLiteRoomNameLocation>

Note: to Use Room We Need To Add To Our Gradle File

After this:

plugins **{** id 'com.android.application'  
 id 'kotlin-android'  
**}**apply plugin: 'kotlin-kapt'

and the implementations:

implementation 'androidx.room:room-common:2.3.0'  
implementation 'androidx.room:room-ktx:2.3.0'  
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.4.1"  
implementation "androidx.room:room-runtime:2.3.0"  
implementation 'android.arch.persistence.room:runtime:1.1.1'  
kapt "androidx.room:room-compiler:2.3.0"

# ViewModel

ViewModel is a part of Android Architecture Components, which make our code cleaner and easier to maintain. ViewModel holds our UI data while surviving configuration changes, such as device rotation and changing of activities. It allows us to separate our logic from the UI.

NOTE:

Never pass context into your ViewModel. Doing so, will lead to memory leaks as a ViewModel outlives the activity that is passed to it. If you need access to the activity, make sure to use AndroidViewModel, which includes a reference to the application.

In this demo, we will add a ViewModel to our Button Clicker App

For more information, refer to the [Official Documentation](https://developer.android.com/topic/libraries/architecture/viewmodel?gclid=CjwKCAjwq9mLBhB2EiwAuYdMtd_UTjQsSAYebwD-8ZQJrQMooCtDIP8UeKk4cRsI-w0GA8tcOrHvBxoCnS8QAvD_BwE&gclsrc=aw.ds)

# LiveData

LiveData works in combination with ViewModel to create a reactive UI. It uses observers to update UI elements based on changes in our data.

The following video takes the Button Clicker App a step further with LiveData

You can read more about LiveData in the [Official Documentation](https://developer.android.com/topic/libraries/architecture/livedata)

# Firebase Intro

Firebase by Google is a ready-made backend server.

Firebase offers lots of services that make the backend hassle-free. In this lesson, we will learn about Firestore, which is one of the database solutions Firebase offers.

Firestore is a cloud database that allows us to save and retrieve the data. As explained in the video, In Android Studio, there is an option to integrate Firestore into your Android app ( Tools → Firebase → Firestore )

First, sign up for Firebase with your Google account and connect your app to Firebase from Android Studio.

Watch the video and save some data to your first Firebase app

Source Code: <https://github.com/KPRanjithKumar/FirebaseFireStoreApp>

# Retrieving Data from Firebase

# Firebase Services

# Navigation Component

Up to this point, we have only used Intents to navigate between screens in our applications. Today, we will look at how the Navigation Component and Fragments can make moving between screens smoother.

The Navigation Component is a collection of libraries for making Android navigation simpler.

You can read more about Navigation Component in the [Official Documentation](https://developer.android.com/guide/navigation/navigation-getting-started)

**Notifications**

When there is an event that the user should be aware of, we need to notify the user.

One way of notifying the user in Android is by displaying a Notification.

A Notification appears from the top of the screen to notify the user of current or upcoming events.

Before we create our first notification, we need to understand that Google allows users to switch off notifications or control them from the settings menu of the phone. This affects the new Android versions (Oreo and higher).

Due to this restriction, we need to create a channel for our notifications.

With that in mind, let's create our first Notification.

First, we need to use the notification manager which controls all the notifications in the system.

val notificationManager = getSystemService(NOTIFICATION\_SERVICE) as NotificationManager

If we want our notification to start another activity when clicked, we need to create an intent for the other activity

val intent = Intent(this, NotificationActivity::class.java)

To transition to another activity, we usually use startActivity(intent), but with notifications we don’t know when the user is going to click the notification, so we need to make the intent *on-hold* (wait to get triggered).

To do that we use PendingIntent.

val pendingIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.FLAG\_UPDATE\_CURRENT)

PendingIntent takes 4 arguments:

1. The context which is the current activity.
2. The response code in case there is something to return.
3. The intent of the other activity
4. A flag to indicate what to be done with the old notification.

As we discussed before, in newer Android versions, we need to create a channel for the notification, but what about the older one?

In order to create a stable notification that works on newer and older versions we need to check the Android version running on the user's device.

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.O) {

    }

else{

    }

In this code we can check the build version and decide when to create an old notification or new notification.

To create the channel, we will use the Notification Channel class.

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.O) {

var notificationChannel = NotificationChannel(channelId,description,NotificationManager.IMPORTANCE\_HIGH)

notificationManager.createNotificationChannel(notificationChannel)

} else {}

The channel id and channel description is what the user will see in the settings when they check for the notifications.

And we can set them as normal strings.

private val channelId = "myapp.notifications"

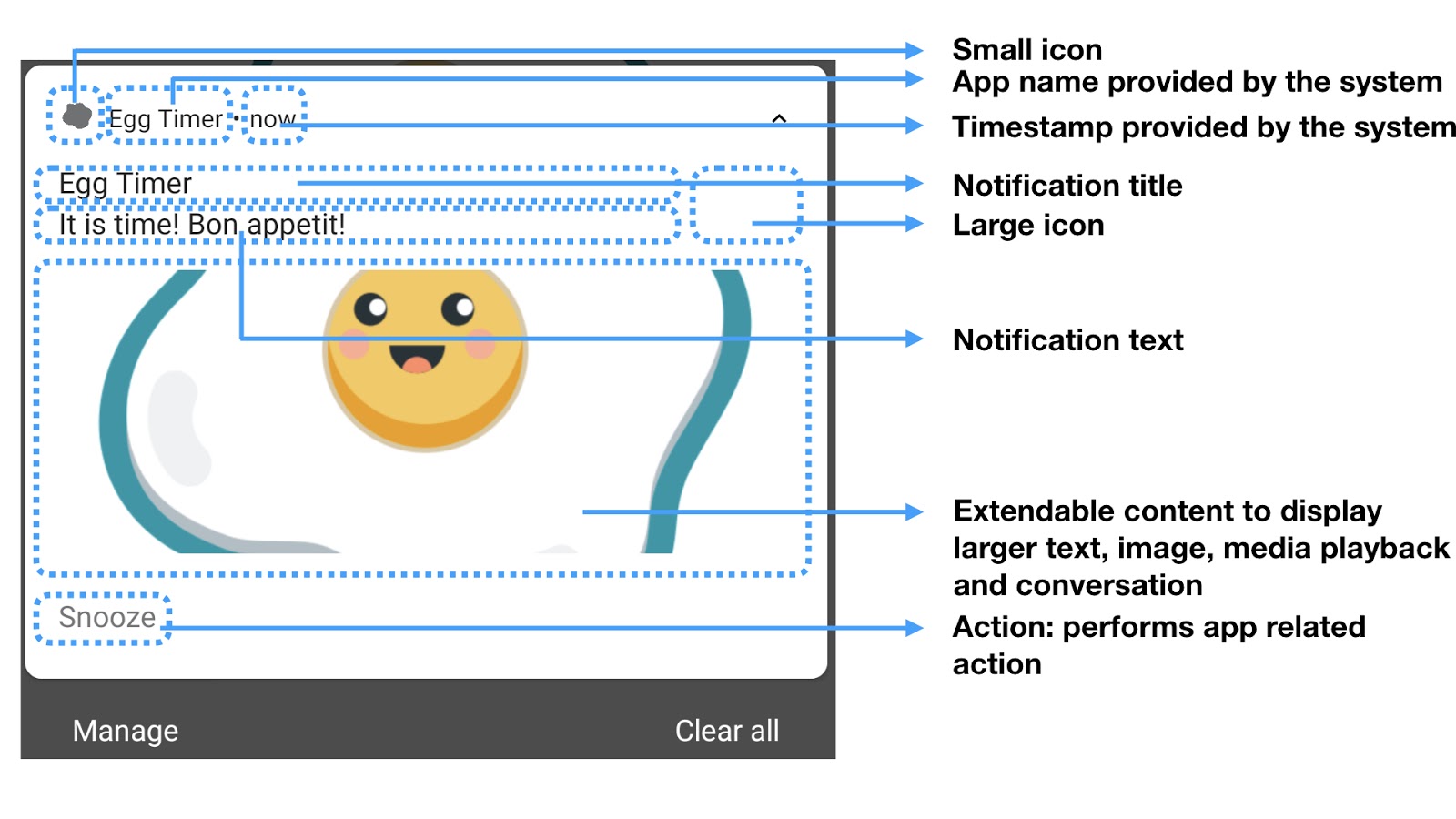
private val description = "Notification App Example"

The last argument in the notification channel is the importance of the notification, if it is high, it means that it will make a sound, vibrate, and notify the user even if the phone is silent in some cases.

After creating the channel and notification manager, building the notification is simple.

To create a notification, we need to define mandatory fields which are:

1. Small icon
2. Content title (the title of the notification)
3. Content text (the message in the notification)
4. Large icon (if it is going to expand)
5. Content intent (the pending intent of the other activity)



Var builder = Notification.Builder(this, channelId)

.setSmallIcon(R.drawable.ic\_notification)

.setLargeIcon(BitmapFactory.decodeResource(this.resources, R.drawable.ic\_notification))

.setContentIntent(pendingIntent)

.setContentTitle("My Notification")

.setContentText("Hello")

copy

This code shows the part that will be within the “if the current android version is newer or equal to Oreo” block.

For the older version we can use the same but without using the channel id in the builder.

Var builder = Notification.Builder(this)

.setSmallIcon(R.drawable.ic\_notification)

        .setLargeIcon(BitmapFactory.decodeResource(this.resources,R.drawable.ic\_notification))

.setContentIntent(pendingIntent)

.setContentTitle("My Notification")

.setContentText(("Hello")

The final step is to initiate the notification.

notificationManager.notify(notificationId, builder.build())

notificationId is an integer id that we can set for the notification, so the notification manager can know which notification we are referring to in case we have multiple notifications.

We can just use a number for the notification instead of creating a variable for it, but the best practice is to keep ids in variables.

notificationManager.notify(1234, builder.build())

Summary:

To Create a notification, we follow these steps:

1. Create notification manager
2. Create notification channel if android version >= Oreo
3. Create notification builder with the content of the notification
4. Call notification manager notify

Full Notification Code:

class MainActivity : AppCompatActivity() {  
  
 private val channelId = "myapp.notifications"  
 private val description = "Notification App Example"  
 private lateinit var builder: Notification.Builder  
 private val notificationId =123  
  
 @RequiresApi(Build.VERSION\_CODES.*O*)  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main*)  
  
 val notificationManager = getSystemService(*NOTIFICATION\_SERVICE*) as NotificationManager  
 val intent = Intent(this, NotificationActivity::class.*java*)  
 val pendingIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.*FLAG\_UPDATE\_CURRENT*)  
  
 builder = if (Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*O*) {  
 val notificationChannel = NotificationChannel(channelId,description,  
 NotificationManager.*IMPORTANCE\_HIGH*)  
 notificationManager.createNotificationChannel(notificationChannel)  
 Notification.Builder(this, channelId)  
 .setSmallIcon(R.drawable.*ic\_fastfood\_24*)  
 .setLargeIcon(BitmapFactory.decodeResource(this.*resources*,R.drawable.*ic\_fastfood\_24*))  
 .setContentIntent(pendingIntent)  
 .setContentTitle("Food Notification")  
 .setContentText("Food is Ready")  
} else {  
 Notification.Builder(this)  
 .setSmallIcon(R.drawable.*ic\_fastfood\_24*)  
 .setLargeIcon(BitmapFactory.decodeResource(this.*resources*,R.drawable.*ic\_fastfood\_24*))  
 .setContentIntent(pendingIntent)  
 .setContentTitle("Food Notification")  
 .setContentText("Food is Ready")  
}  
 notificationManager.notify(notificationId, builder.build())  
 }  
}

You can read more about Notifications in the [Official Documentation](https://developer.android.com/training/notify-user/build-notification)