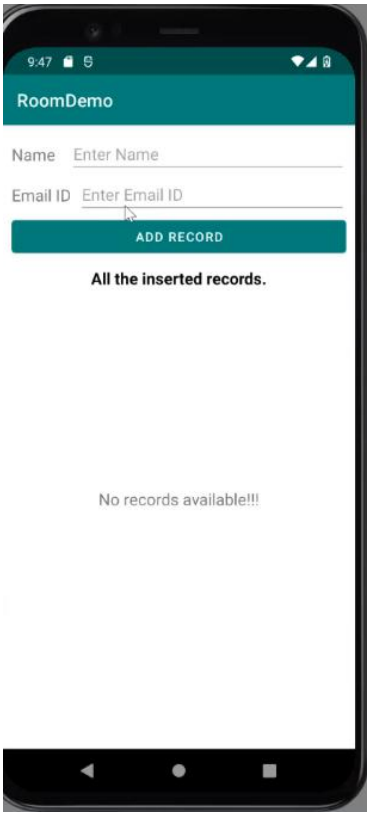
Roomdatabase

This is an abstraction on top of the SQLite that enables one to store data locally. It has a component known as **Entity** that is an equivalent of table, **DAO**(Data Access Object) is an interface that contains queries that will be used by the app to access data and **Database** serves as the main access point to the database. It contains all the necessary set up in order to create the database.

Setting up the data class



This is the main interface. The xml file is unnecessary since its setup is self explanatory at this point.
Before the database is set up there are dependencies that have to be downloaded.In the end the **Module: RoomDemo.app** should look as shown below:

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    //room
```

```
    id 'kotlin-kapt'
}
```

In the dependencies section:

```
dependencies {

    //room
    kapt 'androidx.room:room-compiler:2.4.3'
    implementation 'androidx.room:room-ktx:2.4.3'
    implementation 'androidx.activity:activity-ktx:1.5.1'
    implementation 'androidx.core:core-ktx:1.9.0'
    implementation 'androidx.room:room-runtime:2.4.3'
    //room
```

Now we will create an entity for employees that will create the table to store Employees as shown below:

```
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "employee-table")
data class EmployeeEntity(
    //Autogenerate ensures that every single entry made to the table becomes unique
    @PrimaryKey(autoGenerate = true)
    val id : Int = 0,
    val name : String = "",
    //Define a different name in the database
    @ColumnInfo(name = "email-id")
    val email : String = ""
)
```

DAO interface and database class

The DAO interface is a normal interface but it is annotaded with the **@Dao** key word to indicate that it is a Data Access Object as shown below:

```
@Dao
interface EmployeeDAO {
```

The operations that are done in the dao include data manipulation functions such as insert and fetch. These operations take a lot of time and are therefore not done in the main thread. They are done in the background via the use of coroutines. The

suspend key word indicated that a particular operation should be done in the background as shown in the code below:

```
//Co-routines are used since the operations take a lot of time to execute
@Insert
suspend fun insert(employeeEntity: EmployeeEntity)

@Update
suspend fun update(employeeEntity : EmployeeEntity)

@Delete
suspend fun delete(employeeEntity : EmployeeEntity)
```

Flow is a part of the Kotlin coroutines library and is designed to work well with suspending functions and coroutines. It can be used for tasks such as data streaming, handling asynchronous events, and performing long-running operations in a non-blocking way. Flow can be transformed, combined, and collected, making it a powerful tool for asynchronous programming in Kotlin.

The collect functionality of Flow enables real time data updates. This is very important since the query function that will result in fetching of data will use Flow as shown below:

```
@Query("SELECT * FROM `employee-table`")
//Flow is used to hold values that can change at runtime
fun fetchAllEmployees(): Flow<List<EmployeeEntity>>
```

In order to select or fetch the data for one employee the following code is used:

```
@Query("SELECT * FROM `employee-table` WHERE id=:id")
fun fetchEmployeeById(id : Int) : Flow<EmployeeEntity>
```

Now to create the database. The database is created inside an abstract class as shown below:

```
abstract class EmployeeDataBase : RoomDatabase(){
```

Next we need to define the version and the entities that are to be stored in the database. The entities in the database can be placed in an array or by themselves as shown below:

```
@Database(entities = [EmployeeEntity::class],version = 1)
```

Next we connect our database to the employeeDAO by creating an abstract method that returns the employeeDAO as shown below:

```
//Used to connect database to employeeDao
abstract fun employeeDao():EmployeeDAO
```

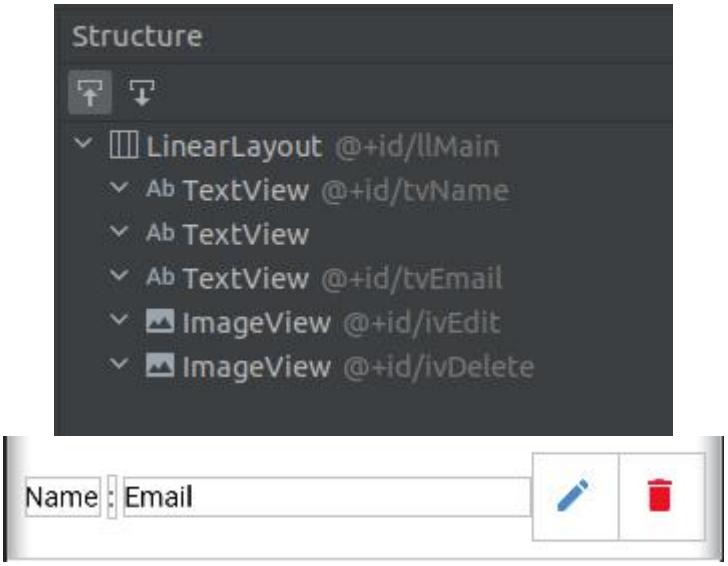
Now for the variables that are needed to create the database. These variables are kept in the companion object so that they can be accessed as class variables. The instance that is used is also declared as volatile so that changes made by one thread are visible to the other threads

```
companion object{
    @Volatile
    private var INSTANCE : EmployeeDataBase ? = null
    //Volatile means that all changes to data are from main memory so that the changes are seen by all threads
```

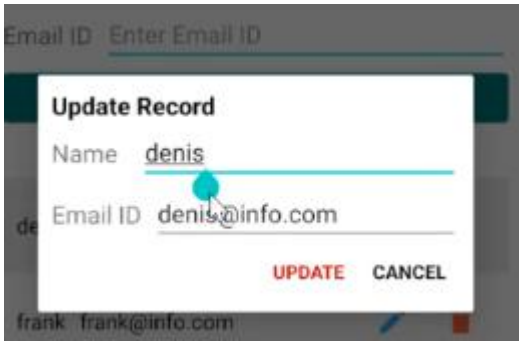
Now we create a function to create the database using the singleton pattern. This means that if you want to fetch the database and it already exists we destroy it and build a new one. The singleton pattern ensures that only one instance of an object exists at a time

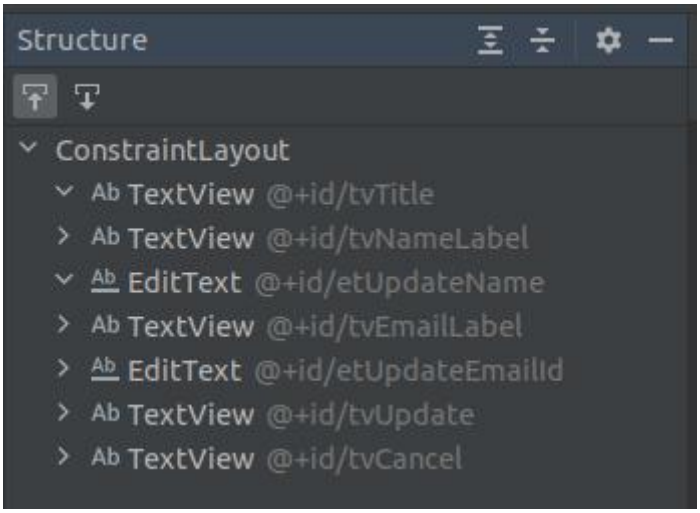
```
    //Singleton pattern is used here where only one instance of a class exists
    fun getInstance(context:Context) :EmployeeDataBase{
        //Synchronized means that only one thread may enter
        synchronized( lock: this){
            var instance = INSTANCE
            if(instance == null){
                //This method of creating the database is that it destroys and rebuilds the database when changes to the database structure occur
instead of migrating
                instance = Room.databaseBuilder(context.applicationContext,EmployeeDataBase::class.java, name: "employee_database"
                ).fallbackToDestructiveMigration().build()
                INSTANCE = instance
            }
            return instance
        }
    }
```

Adding clickable widgets to the items in the recycler view



There is also a dialog box that pops up when we want to edit a record as shown below:





Adding data to the database

We will create a method that will have as a parameter the DAO that we want to use as shown below:

```
private fun addRecord(employeeDAO : EmployeeDAO){...}
```

The process of adding data should be done in the background and therefore we will use co routines to achieve that functionality as shown below:

```
private fun addRecord(employeeDAO : EmployeeDAO){
    val name = binding?.etName?.text.toString()
    val email = binding?.etEmailId?.text.toString()
    if(name.isNotEmpty() && email.isNotEmpty()){
        //These tasks should be done in the background
        lifecycleScope.launch{ this: CoroutineScope
            //ID field is auto increment therefore no need to set it here
           employeeDAO.insert(EmployeeEntity(name=name,email=email))
            //ApplicationContext passed since were in the background
           Toast.makeText(applicationContext, text: "Record saved",Toast.LENGTH_LONG).show()
            //Clear fields to allow next input
            binding?.etName?.text?.clear()
            binding?.etEmailId?.text?.clear()
        }
    }else{
        Toast.makeText(applicationContext, text: "Name or email can not be blank",Toast.LENGTH_LONG).show()
    }
}
```

In order to use the EmployeeDao within a button we need to initialize the database using an application class. An application class is a class that inherits from application as shown below:

```
package com.example.roomdatabase

import android.app.Application

//This is meant to create the database
//The Application class must be defined in the android manifest under android:name in the <application></application>
//The Singleton pattern used implemented the getInstance() method which requires,as a parameter, a context
//The this application app is created to meet that requirement
class EmployeeApp : Application() {
    val db by lazy{
        EmployeeDataBase.getInstance( context: this)
    }
}
```

```
    <!-- The database is instantiated in the app successfully
because of the {android:name = ".EmployeeApp"} maybe its because
that class extends Application class -->
    <application
        android:allowBackup="true"
        android:name=".EmployeeApp"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="RoomDatabase"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
```

EmployeeApp declared in Manifest

We need to declare the class that instanciates the data base as an application class since it is going to be used throughout the application

Finally we can get the employeeDao that is needed in order to carry out data manipulations as shown below:

```
//Get database instance from EmployeeApp and use it to get the employeeDao
val employeeDao = (application as EmployeeApp).db.employeeDao()
```

Item adapter with listeners

This part is explained in the code comments. We first create an item adapter and pass in the necessary parameters to create a constructor. The item adapter must have the data structure that contains all of the data to be displayed. This one in particular also passes lambdas that will be used as onClick listeners as shown below:

```
//RecyclerView doesn't inherit from onClickListeners and that is why
  the listeners have to be passed as parameters
//This part{ private val updateListener:(id:Int)} is the method
 signature of the lambda; the parameters and the method body.
//The Unit part is describing what method body {Which is user
 designed} is supposed to return
class ItemAdapter(private val items : ArrayList<EmployeeEntity>,
                  private val updateListener:(id:Int) -> Unit,
                  private val deleteListener:(id:Int) -> Unit
                  ) : RecyclerView.Adapter<ItemAdapter
  .ItemViewHolder>()
{
    //The viewHolder
    class ItemViewHolder(binding: ItemsRowBinding) : RecyclerView
.ViewHolder(binding.root){
        //Setup the different item views
        val llMain = binding.llMain
        val tvName = binding.tvName
        val tvEmail = binding.tvEmail
        val ivEdit = binding.ivEdit
        val ivDelete = binding.ivDelete
    }
```

The onBind holder is where majority of the magic happens. This is the begining of the method as shown below:

```
    override fun onBindViewHolder(holder: ItemViewHolder, position:
Int) {
        //Item view is an implicit variable in the ViewHolder
(ItemViewHolder) class that represents a single row in the list
        val context = holder.itemView.context
        val item = items[position]

        holder.tvName.text = item.name
        holder.tvEmail.text = item.email

        //At every second object change background to light grey
        if(position % 2 == 0){
            holder.llMain.setBackgroundColor(ContextCompat.getColor
(holder.itemView.context,R.color.light_grey))
        }
        //Make the remaining to be white
        else{
            holder.llMain.setBackgroundColor(ContextCompat.getColor
(holder.itemView.context,R.color.white))
        }
```

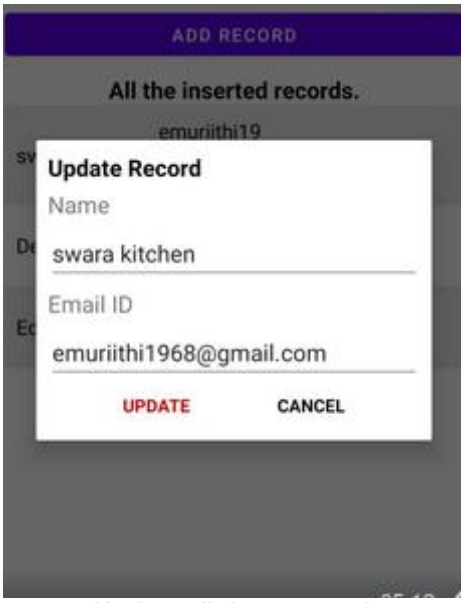
For the onclick listeners we create a lambda that will invoke the id of an entry. The id of the entry is what will be used in the **MainActivity** together with the EmployeeDao as a method parameter to manipulate data accordingly. The code below shows how the onClickListeners has been implemented:

```
    //ItemOnClickListeners
    holder.ivEdit.setOnClickListener{ it: View!
        // Get the user
        // Calling the lambda and passing the parameters required
        // Remember that invoking a lambda means that the passed variable is what will be used as a parameter
        updateListener.invoke(item.id)
    }
    holder.ivDelete.setOnClickListener{ it: View!
        // Get the user
        // Calling the lambda and passing the parameters required
        // Remember that invoking a lambda means that the passed variable is what will be used as a parameter
        deleteListener.invoke(item.id)
    }
}
```

Now we have to use this item adapter in the **MainActivity** in order to set up the data that will be posted to the recyclerView and finalize implementing the onClickListeners as shown below:

```
// Remember that the lambdas were part of the constructor parameters
val itemAdapter = ItemAdapter(employeeList,
{
    // When the edit button is pressed there is a lambda that invokes the item ID
    // That item ID is what will be used as the UpdateID
    updateID ->
    updateRecordDialog(updateID,employeeDAO)
},
    {
        deleteID ->
        deleteRecordDialog(deleteID,employeeDAO)
    }
)
```

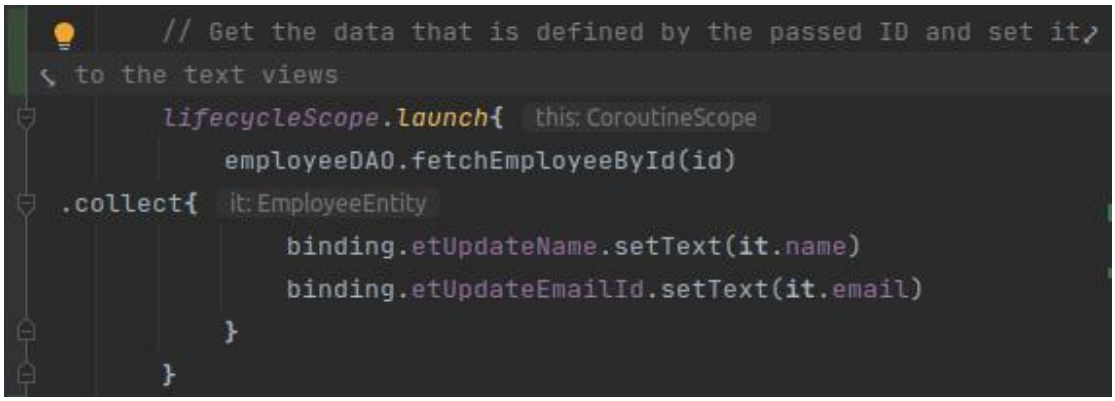
The methods that update and delete methods are as shown below. Remember that if the return type of a lambda is unit it means we are free to define the method body any way we want to.For the update implementation the method body creates a dialog that has the update button and the necessary implementations as described by the comments.



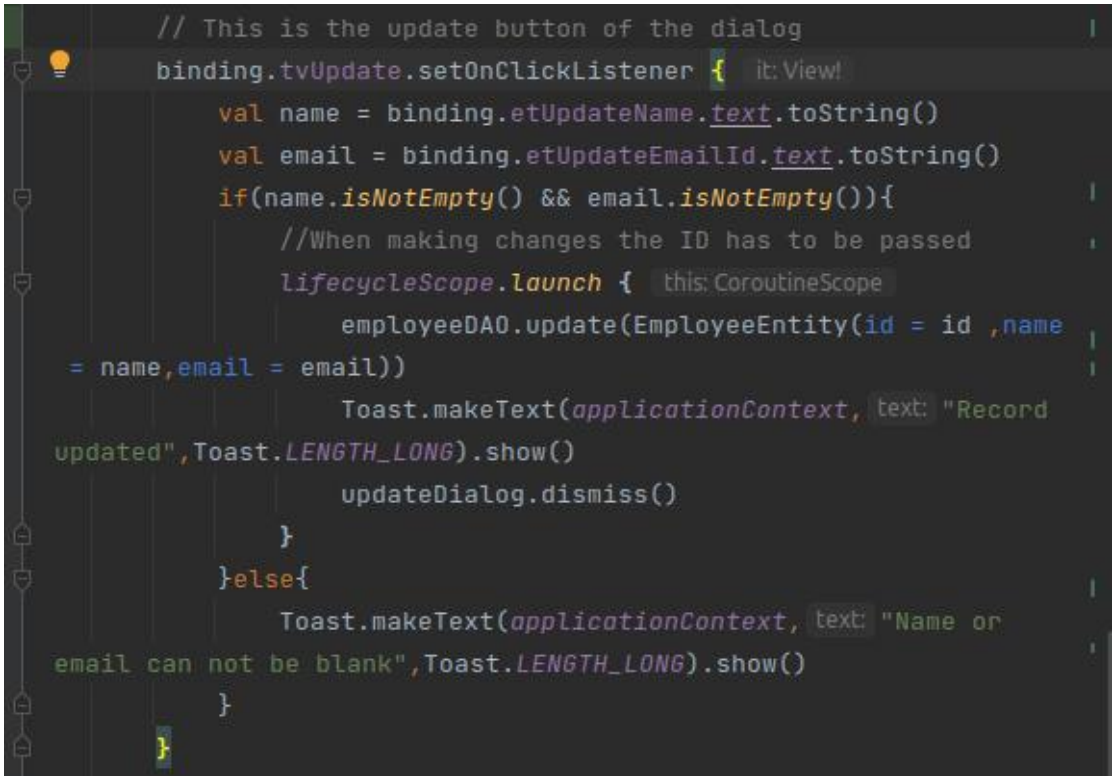
Update dialog created

```
private fun updateRecordDialog(id:Int,employeeDAO: EmployeeDAO){
    val updateDialog = Dialog( context: this,com.example
.roomdatabase.R.style.Theme_Dialog_Custom)
    updateDialog.setCancelable(false)
    //Inflate the layout file for the update dialog
    val binding = DialogUpdateBinding.inflate(layoutInflater)
    updateDialog.setContentView(binding.root)
```

Code for update dialog



```
        // Get the data that is defined by the passed ID and set it
    to the text views
        lifecycleScope.launch{ this: CoroutineScope
            employeeDAO.fetchEmployeeById(id)
    .collect{ it: EmployeeEntity
                binding.etUpdateName.setText(it.name)
                binding.etUpdateEmailId.setText(it.email)
            }
        }
```



```
        // This is the update button of the dialog
        binding.tvUpdate.setOnClickListener { it: View!
            val name = binding.etUpdateName.text.toString()
            val email = binding.etUpdateEmailId.text.toString()
            if(name.isNotEmpty() && email.isNotEmpty()){
                //When making changes the ID has to be passed
                lifecycleScope.launch { this: CoroutineScope
                    employeeDAO.update(EmployeeEntity(id = id ,name
    = name,email = email))
                    Toast.makeText(applicationContext, text: "Record
    updated",Toast.LENGTH_LONG).show()
                    updateDialog.dismiss()
                }
            }else{
                Toast.makeText(applicationContext, text: "Name or
    email can not be blank",Toast.LENGTH_LONG).show()
            }
        }
```

The delete functionality follows the same format. Create the delete dialog and implement the data manipulation methods when the buttons are clicked as shown below:

```
    private fun deleteRecordDialog(id:Int,employeeDAO: EmployeeDAO){
        // Creating the delete dialog
        val builder = AlertDialog.Builder( context: this)
        builder.setTitle("Delete Record")
        builder.setIcon(com.example.roomdatabase.R.drawable
.ic_baseline_delete)
        builder.setPositiveButton( text: "Yes"){
            dialogInterface,_ ->
            // Deleting the data
            lifecycleScope.launch{ this: CoroutineScope
                employeeDAO.delete(EmployeeEntity(id))
                Toast.makeText(applicationContext, text: "Record
deleted successfully",Toast.LENGTH_LONG).show()
            }
            dialogInterface.dismiss()
        }
        builder.setNegativeButton( text: "No"){
            dialogInterface,_ ->
            dialogInterface.dismiss()
        }
        val alertDialog : AlertDialog = builder.create()
        alertDialog.setCancelable(false)
        alertDialog.show()
    }
```

Finally we can finalize this process by using this method in the **onCreate** as shown below:

```
        //Loading data tasks should be done in the background
        lifecycleScope.launch{ this: CoroutineScope
            employeeDao.fetchAllEmployees()
.collect{ it: List<EmployeeEntity>
                // Create a list since the method
setUpRecyclerViewData has a list as a parameter
                val list = ArrayList(it)
                setUpRecyclerViewData(list,employeeDao)
                // Since a flow is used we do not need to inform
the recyclerview that data has been changed
            }
        }
```

```xml
<!-- Custom style that enables the update dialog to ake up 90% of screen -->
<style name="Theme_Dialog_Custom" parent="ThemeOverlay.AppCompat.Dialog">
    <!-- The Dialog width should take 90% of the screen -->
    <item name="android:windowMinWidthMajor">90%</item>
    <item name="android:windowMinWidthMinor">90%</item>
</style>
```