ROOM DB

Review Notes App with the following concept

- ➔ Various Dependencies
- ➔ How to work with Room DB
- ➔ Room DB Annotations
- ➔ Navigation Component
- ➔ Room DB functionalities cannot run on Main Thread
- ➔ To Manage on separate thread using Kotlin Coroutines
- ➔ Suspend function
- ➔ Kotlin higher order scope functions (let)
- ➔ Kotlin Extension function

Ref : NotesAppRoomDB demo code

Room is a persistence library, part of the Android Architecture Components.
The Room persistence library provides an abstraction layer over SQLite.

Android architecture components are a collection of libraries that help you design robust, testable, and maintainable apps. It's a part of Android Jetpack.

Android Jetpack contains the following libraries

- Guide to develop a robust app architecture
- Manage app lifecycle using lifecycle-aware components
- Use LiveData to build data objects that notify views when the underlying database changes.
- ViewModel stores UI-related data that isn't destroyed on app rotations.
- Room is a SQLite object mapping library.

Advantages of Room Library

- Use it to avoid boilerplate code and easily convert SQLite table data to Java objects.
- Room provides compile time checks of SQLite statements.

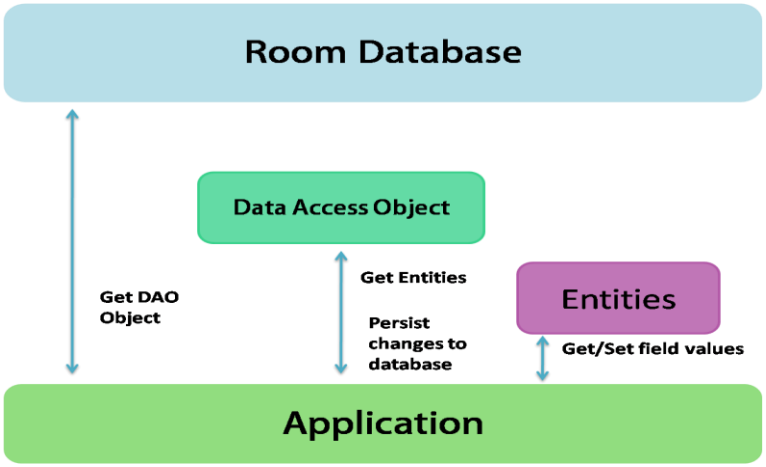
Components of Room database

Entity: Annotated class that describes a database table when working with Room.

DAO: Data access object. A mapping of SQL queries to functions. When you use a DAO, you call the methods, and Room takes care of the rest.

Room database: Simplifies database work and serves as an access point to the underlying SQLite database (hides SQLiteOpenHelper). The Room database uses the DAO to issue queries

to the SQLite database. An abstract class annotated with @Database and inherits from RoomDatabase class.



Step 1 : Dependencies need to include build.gradle

It should be at the top of build.gradle

apply **plugin**: **"androidx.navigation.safeargs"**

apply **plugin**: **'kotlin-kapt'**

dependencies{

// version of the room Link : https://developer.android.com/topic/libraries/architecture/room

def room_version = **"2.2.4"**
// Room Library
implementation **"androidx.room:room-runtime:**$room_version**"**
kapt **"androidx.room:room-compiler:**$room_version**"**

// Navigation Library
// Version of Navigation- //https://developer.android.com/guide/navigation/navigation-getting-started

def nav_version = **"2.3.0-alpha02"**
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"

// Material design : https://material.io/develop/android/docs/getting-//started/

// **To use** the new components and styles available in the **material design** support library

implementation **'com.google.android.material:material:1.1.0'**

}

To add Safe Args to your project, include the following `classpath` in your top level `build.gradle` file:

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"

apply plugin: "androidx.navigation.safeargs"
```

Navigation Component – Jetpack library

Note: If you want to use Navigation with Android Studio, you must use Android Studio 3.3 or higher.

A Collection of libraries, a plugin and tooling for unifying and simplifying Android Navigation.

Navigation Component Benefits

- Simplified setup for common navigation benefits
- Handles back stack
- Automates fragment transaction
- Type safe argument passing
- Handles transition animations
- Simplified deep linking
- Centralizes and visualizes navigation

Build in support for

- Fragments
- Activities
- Can also with custom destinations

Three major Navigation Components

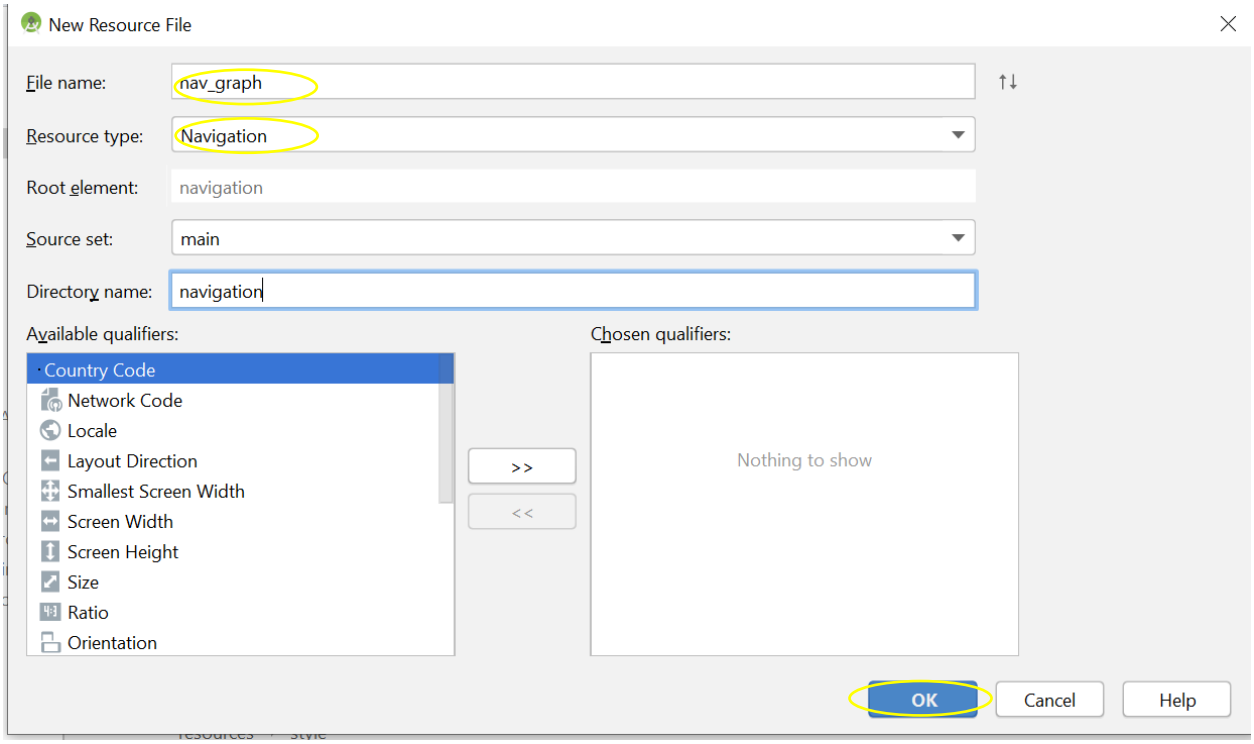
- Navigation graph: An XML resource that contains all navigation-related information in one centralized location. This includes all of the individual content areas within your app, called *destinations*, as well as the possible paths that a user can take through your app.
- `NavHost`: An empty container that displays destinations from your navigation graph. The Navigation component contains a default `NavHost` implementation, `NavHostFragment`, that displays fragment destinations.
- `NavController`: An object that manages app navigation within a `NavHost`. The `NavController` orchestrates the swapping of destination content in the `NavHost` as users move throughout your app.

Step 2 : change the style of your theme to MaterialComponents under res→values→styles.xml

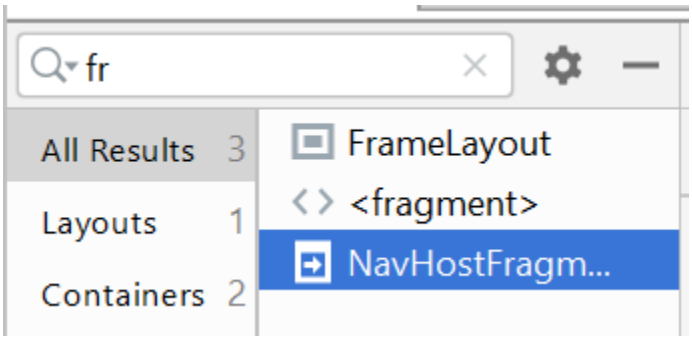
<**style name="AppTheme" parent="Theme.MaterialComponents.Light.DarkActionBar"**>

This app is designed with Activity and Fragments.

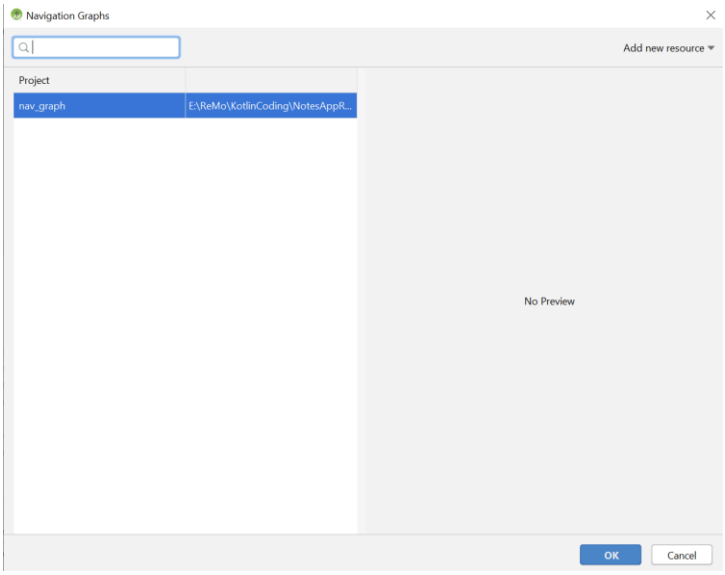
Step 3: Creation of Navigation Graph, need to add navigation resource file. Right click res→New→Android Resource File. You will get the below screen shot. In this give the name and select Resource Type as Navigation then click ok.



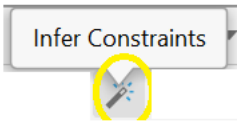
Step 4: Add NavigationHostFragment on activity_main.xml, type fragment on the search and you will NavHostFragment, drag into your activity.



a) Once you drag will get the given screen shot. Select the created navigation graph and click ok.



b) To get rid of constraints warning, select the infer constraints icon at the top



Step 5 : Create two packages for UI and DB, put MainActivity under UI

Step 6: Right click on UI package, New→Fragment→Fragment(Blank) and name it as HomeFrgment.

Step7: Use the given code to design the home fragment layout with FloatActionButton

```
<RelativeLayout 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=".ui.HomeFragment">
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view_notes"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/button_add"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:clickable="true"
        android:layout_alignParentRight="true"
```

```
        android:layout_alignParentBottom="true"
        android:src="@drawable/addbutton" />
</RelativeLayout>
```

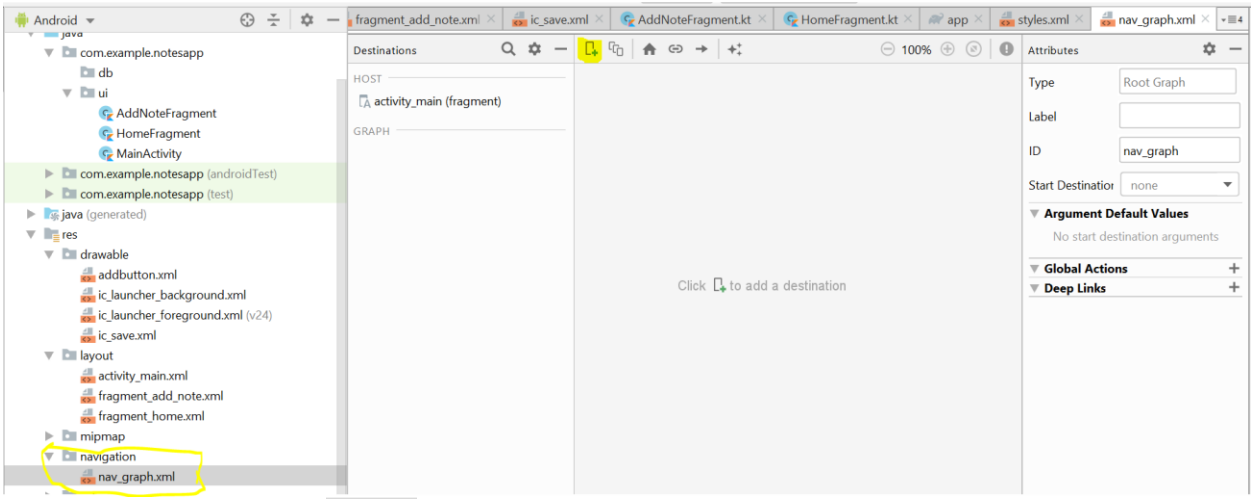
Step 8: Right click on UI package, New→Fragment→Fragment(Blank) and name it as AddNoteFragment

Design the fragment_add_note with the below xml code, which must get input for title and note. Save FAB to save note.

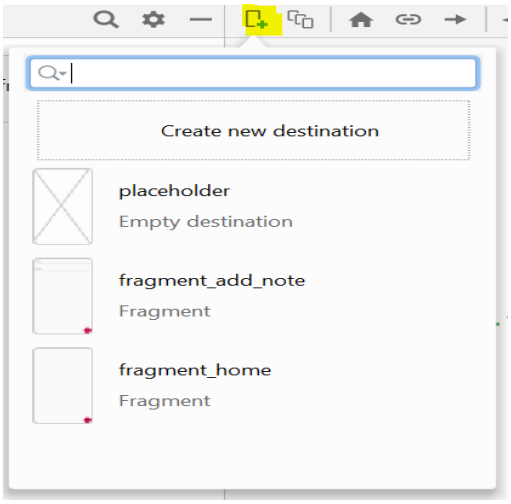
```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".ui.AddNoteFragment">

  <EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="text"
    android:id="@+id/title"
    android:hint="Title"
    android:textAppearance="@style/TextAppearance.AppCompat.Large">
  </EditText>

  <EditText
    android:id="@+id/editNote"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@+id/title"
    android:hint="Note"
    android:inputType="text"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    android:layout_marginTop="20dp"
    android:gravity="top"/>

  <com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/button_save"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:clickable="true"
    android:layout_alignParentRight="true"
    android:layout_alignParentBottom="true"
    android:src="@drawable/ic_save" />

</RelativeLayout>
```

Step 9: Using navigation graph need to make navigation between the two fragments created, by doing the following steps.

a) Click res→navigation→nav_grapg.xml, you will get a window on right side, by clicking the highlighted button and add the two created fragments.

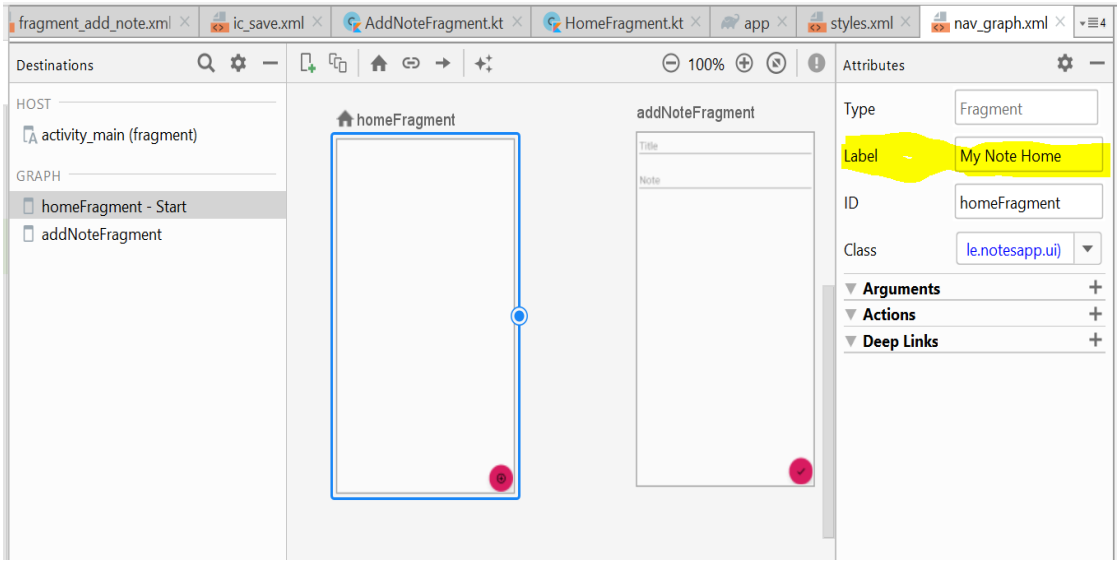


b) After clicking the button  you will get the screen as below



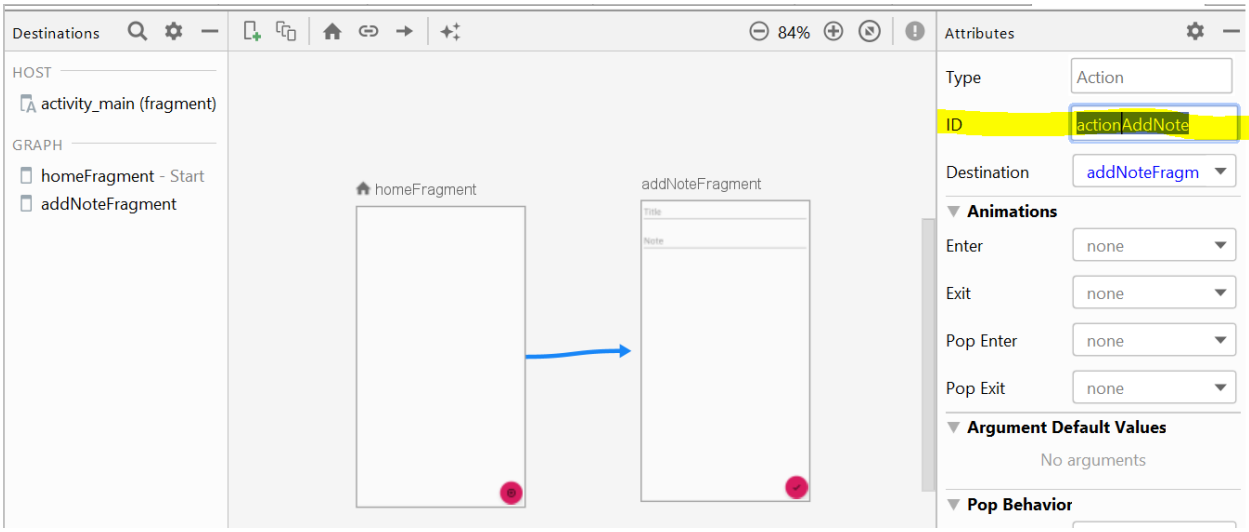
c) I changed the label name of both fragments using the attributes column, fragment_home labelled as My Note Home
addNoteFragment labelled as Add Note as

See the screen shot

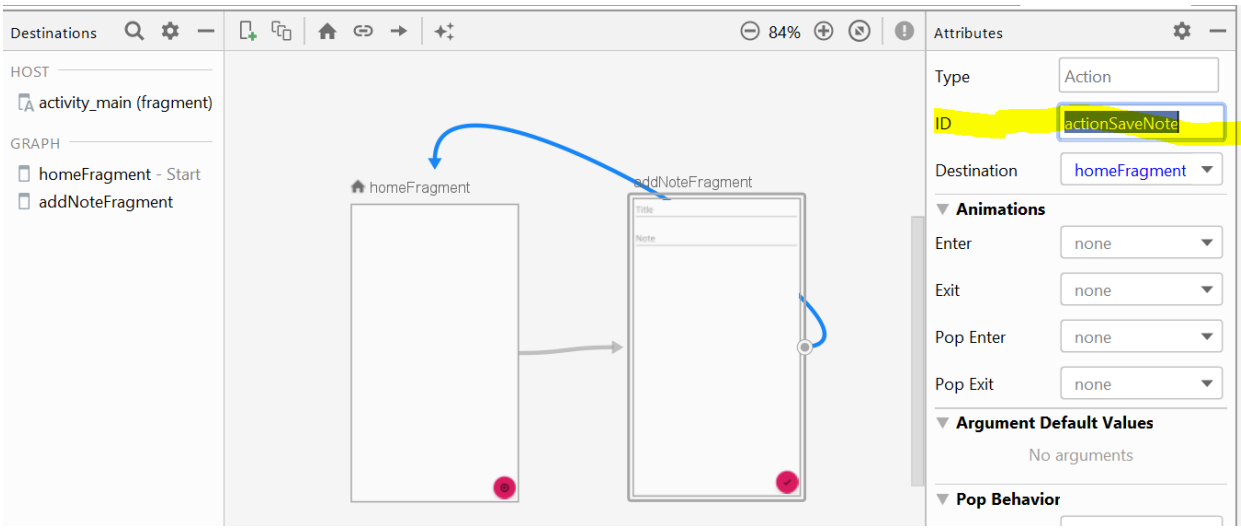


d) We want to navigate at the run time from homeFragment to addNoteFragmet and vice versa. For the just drag the dot blue button from Middle right of homeFragment to addNoteFragment(give ID as actionAddNote) and vice versa as per the below two screenshots. This connection are called Actions.

From homeFragment to addNoteFragment Navigation screenshot(Action 1)



From addNoteFragment to homeFragment Navigation screenshot(Action 2)



Step 10 : Open your MainActivity.kt, set the Naviagtioncontroller and override the onSupportNavigateUp()method. Add the highlighted code as below

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        // Setting Navigation to our Application, by passing current context and navigation host
//fragment id
        val navController = Navigation.findNavController(this,R.id.fragment)
       /*By calling this method, the title in the action bar will automatically
         be updated when the destination changes*/
        NavigationUI.setupActionBarWithNavController(this,navController)
    }

    override fun onSupportNavigateUp(): Boolean {
        return NavigationUI.navigateUp(Navigation.findNavController(this,R.id.fragment),null)
    }
}
```

Step 11: Click Build→ Rebuild Project, to get the Navigation Direction classes for the fragments generated automatically by the system.

Step 12: Implement action for button_add in the HomeFragment by adding the highlighted code

```
class HomeFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
```

```
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_home, container, false)
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        button_add.setOnClickListener {
            // After Rebuild you will get HomeFragmentDirections automatically,
            // call the navigation action id given in the Navigation graph
        val action = HomeFragmentDirections.actionAddNote()
            // Navigate to the action by passing view and call navigate by passing action
            Navigation.findNavController(it).navigate(action)
        }
    }
```

Step 13: Adding Important components for Room DB like Entity, DAO and DB

a) Entity class defines the table structure – Note.kt

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

/* Annotate as Entity and define columns of the table,
it will define the same name in the table column, if you want to provide the different name use as
@ColumnInfo(name = "note_title") before each column*/
@Entity
data class Note(
@PrimaryKey(autoGenerate = true)
val id:Int,
val title :String,
val Note :String
)
```

b) DAO class needed for each Entity includes what you want to do with your entity - NoteDao.kt

```
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query

// Create DAO for each entity, we have one entity. create one NoteDao
// Provide the functionality expects from the DB
@Dao
interface NoteDao {
    @Insert
    fun addNote(note:Note)
    @Query("SELECT * FROM NOTE")
```

```
    fun getAllNotes():List<Note>
    @Insert
    fun addMultipleNotes(vararg note: Note)
}
```

c) Build Database – NoteDatabase.kt

Operator fun invoke() An interesting feature of the Kotlin language is the ability to define an "invoke operator". When you specify an invoke operator on a class, it can be called on any instances of the class without a method name!

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

/* Database class should be abstract and annotated with @Database and pass all entities as an
array  so use [] brackets and specify the version number, in future if you want to change the DB
schema  and change the version number and create migration class to change database structure
*/
@Database(
    entities = [Note::class],
    version = 1
)
abstract class NoteDatabase():RoomDatabase() { // Must Inherit from RoomDatabase
    abstract fun getNoteDao() : NoteDao
    // Build RoomDB
    companion object {
        //  meaning that writes to this field
        // * are immediately made visible to other threads
        @Volatile private var instance : NoteDatabase? = null
        private val LOCK = Any() // The root of the Kotlin class hierarchy. Every Kotlin class has
[Any] as a superclass.
        //  Invoke check if the instance is not null return the instance, if it is null
        //  synchronized block work, inside this also check null or not and call the function
buildDatabase
        operator fun invoke(context: Context) = instance ?: synchronized(LOCK){
            instance ?: buildDatabase(context).also {
                instance = it
            }
        }
        // Function to build database
        private fun buildDatabase(context: Context) = Room.databaseBuilder(
            context.applicationContext,
            NoteDatabase::class.java,
            "notedatabase"
```

```
        ).build()

    }
}
```

Step – 14

Code to add Notes in the database by clicking Float Action Button inside AddNoteFragment.kt class

Kotlin Coroutines are introduced

Kotlin Coroutines: a new way of managing background threads that can simplify code by reducing the need for callbacks. Coroutines are a Kotlin feature that convert async callbacks for long-running tasks, such as database or network access, into *sequential* code.

Network request cannot run through Main thread, we need to run it on a separate thread using Coroutines.

The keyword suspend is Kotlin's way of marking a function, or function type, available to coroutines. When a coroutine calls a function marked suspend, instead of blocking until that function returns like a normal function call, it **suspends** execution until the result is ready then it **resumes** where it left off with the result. While it's suspended waiting for a result, **it unblocks the thread that it's running on** so other functions or coroutines can run.

Make a change in the NoteDao, add suspend keyword in the functions

```
@Dao
interface NoteDao {
    @Insert
    suspend fun addNote(note:Note)
    @Query("SELECT * FROM NOTE")
    suspend fun getAllNotes():List<Note>
    @Insert.
    suspend fun addMultipleNotes(vararg note: Note)
}
```

Step 15

To implement Coroutine in all fragments, new abstract class BaseFragment is introduced and which inherits from Fragment and implements CoroutineScope.

To make use of Kotlin Coroutines add the below dependency in build.gradle

```
// Kotlin Coroutines
implementation "androidx.room:room-ktx:$room_version"
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.0'
```

In Kotlin, all coroutines run inside a CoroutineScope. A scope controls the lifetime of coroutines through its job. When you cancel the job of a scope, it cancels all coroutines started in that scope.

On Android, you can use a scope to cancel all running coroutines when, for example, the user navigates away from an Activity or Fragment.

To specify where the coroutines should run, Kotlin provides three dispatchers that you can use:

- **Dispatchers.Main** - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling suspend functions, running Android UI framework operations, and updating LiveData objects.
- **Dispatchers.IO** - This dispatcher is optimized to perform disk or network I/O outside of the main thread. (Example : Retrofit)
- **Dispatchers.Default** - This dispatcher is optimized to perform CPU-intensive work outside of the main thread. Example use cases include sorting a list and parsing JSON.

BaseFragment.kt

```
package com.example.notesapp.ui

import android.os.Bundle
import androidx.fragment.app.Fragment
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlin.coroutines.CoroutineContext

abstract class BaseFragment : Fragment(),CoroutineScope{
    // Instance in the Co routine Context
    private lateinit var job: Job

    override val coroutineContext: CoroutineContext
        // To perform the Job, Displatchers.Main is used for CoroutineContext
        get() = job + Dispatchers.Main

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Create an Instance for the Job()
        job = Job()
    }

    override fun onDestroy() {
        super.onDestroy()
        // Cancel the Job
        job.cancel()
    }
}
```

AddNoteFragment.kt

```
package com.example.notesapp.ui
import android.os.AsyncTask
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import com.example.notesapp.R
import com.example.notesapp.db.Note
import com.example.notesapp.db.NoteDatabase
import kotlinx.android.synthetic.main.fragment_add_note.*
import kotlinx.coroutines.launch

/**
 * A simple [Fragment] subclass.
 */
class AddNoteFragment :BaseFragment() {
    private var note: Note? = null
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
       // setHasOptionsMenu(true)
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_add_note, container, false)
    }
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        // Set the listener for the FAB
        button_save.setOnClickListener {
            // Retrieve the values from the EditText fields
            val noteTitle = title.text.toString()
            val noteBody = editNote.text.toString()
            // Check the input values are empty, then set the error message and give the focus
            if(noteTitle.isEmpty()){
                title.error = "Title Required"
                title.requestFocus()
                return@setOnClickListener // stop further execution ie returning at the end of the
setOnClickListener
            }

            if(noteBody.isEmpty()){
                editNote.error = "Title Required"
```

```
            editNote.requestFocus()
            return@setOnClickListener // stop further execution ie returning at the end of the
setOnClickListener
        }
        launch {
            val note = Note(noteTitle,noteBody)
            context?.let {
                NoteDatabase(it).getNoteDao().addNote(note)
                it.toast("Note Saved")
            }
        }

    }
    }
}
```

Step 16: implementation for getting the saved note data and show it on the RecyclerView, create it's layout note_layout.xml

To display the latest entry in the recyclerview, NoteDao Query has changed as below

```
@Query("SELECT * FROM NOTE ORDER BY id DESC")
suspend fun getAllNotes():List<Note>
```

Note_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">

    <androidx.cardview.widget.CardView
        android:layout_margin="4dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">

        <RelativeLayout
            android:padding="8dp"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content">

            <TextView
```

```
android:textAppearance="@style/Base.TextAppearance.AppCompat.Headline"
            tools:text="My First Note"
            android:layout_marginBottom="10dp"
            android:id="@+id/text_view_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>

        <TextView

android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium"
            tools:text="Note Description"
            android:layout_below="@id/text_view_title"
            android:id="@+id/text_view_note"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>

    </RelativeLayout>

  </androidx.cardview.widget.CardView>

</LinearLayout>
```

Step 17: Implement adapter for the RecyclerView

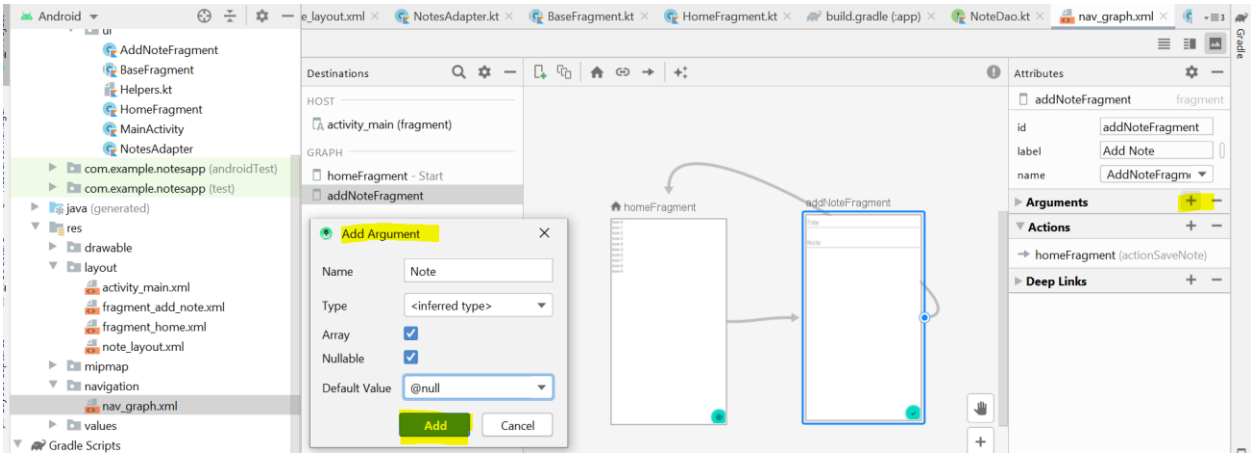
```
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import com.example.notesapp.R
import com.example.notesapp.db.Note
import kotlinx.android.synthetic.main.note_layout.view.*

class NotesAdapter(private val notes: List<Note>) :
RecyclerView.Adapter<NotesAdapter.NoteViewHolder>() {
  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
NotesAdapter.NoteViewHolder {
    return NoteViewHolder(
      LayoutInflater.from(parent.context)
        .inflate(R.layout.note_layout, parent, false)
    )
  }

  override fun getItemCount() =notes.size

  override fun onBindViewHolder(holder: NotesAdapter.NoteViewHolder, position: Int) {
    holder.view.text_view_title.text = notes[position].title
```

```
        holder.view.text_view_note.text = notes[position].note

    }
    class NoteViewHolder(val view: View) : RecyclerView.ViewHolder(view)
}


// Add the following code in  HomeFragment

override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)
    recycler_view_notes.setHasFixedSize(true)
    recycler_view_notes.layoutManager = StaggeredGridLayoutManager(2,
StaggeredGridLayoutManager.VERTICAL)

    // Retrieve all notes from database to RecyclerView using Coroutines
    launch {
        context?.let{
            val notes = NoteDatabase(it).getNoteDao().getAllNotes()
            recycler_view_notes.adapter = NotesAdapter(notes)
        }
    }
```

Step 18: Update Notes by clicking items from the Recyclerview list

Implement the click listener in NotesAdapter.kt and we are using AddNoteFragment.kt to update the data. Need to pass arguments using Navigation in nav_graph.xml.

Select nav_graph.xml, select + icon from Arguments on the right side, you will get Add Argument window give Name, select Nullable and default value as @null as per the screen shot.



Add the highlighted line of code in the nav_graph.xml. Add the path of Note.kt with full package name.

```
<fragment
   android:id="@+id/addNoteFragment"
   android:name="com.example.notesapp.ui.AddNoteFragment"
   android:label="Add Note"
   tools:layout="@layout/fragment_add_note" >
   <action
      android:id="@+id/actionSaveNote"
      app:destination="@id/homeFragment" />
   <argument
      app:argType="com.example.notesapp.db.Note"
      android:name="Note"
      app:nullable="true"
      android:defaultValue="@null" />
</fragment>
```

Step 19: After the above step rebuild your application

Step 20: Need to implement the delete logic, add button at fragment_add_note.xml

```
<Button
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:text="Delete"
   android:id="@+id/button_delete"
   android:textSize="20sp"
   android:layout_alignParentTop="true"
   android:layout_marginTop="151dp">
</Button>
```

Step 21: Complete code after adding delete logic in addNoteFragment.kt

```
package com.example.notesapp.ui
import android.app.AlertDialog
import android.os.AsyncTask
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.navigation.Navigation
import com.example.notesapp.R
import com.example.notesapp.db.Note
import com.example.notesapp.db.NoteDatabase
import kotlinx.android.synthetic.main.fragment_add_note.*
```

```
import kotlinx.coroutines.launch


/**
 * A simple [Fragment] subclass.
 */
class AddNoteFragment :BaseFragment() {
    private var note: Note? = null
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
      // setHasOptionsMenu(true)
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_add_note, container, false)
    }
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        // Receive the note arguments
        arguments?.let {
            // get the note value from the HomeFragment using Bundle instance
            note = AddNoteFragmentArgs.fromBundle(it).note
            title.setText(note?.title)
            editNote.setText(note?.note)
        }
        // deletion logic
        button_delete.setOnClickListener {
            if (note != null) deleteNote() else context?.toast("Cannot Delete")
        }
        // Set the listener for the FAB
        button_save.setOnClickListener {view ->
            // Retrieve the values from the EditText fields
            val noteTitle = title.text.toString()
            val noteBody = editNote.text.toString()
            // Check the input values are empty, then set the error message and give the focus
            if(noteTitle.isEmpty()){
                title.error = "Title Required"
                title.requestFocus()
                return@setOnClickListener // stop further execution ie returning at the end of the
setOnClickListener
            }

            if(noteBody.isEmpty()){
                editNote.error = "Title Required"
                editNote.requestFocus()
                return@setOnClickListener // stop further execution ie returning at the end of the
```

```
setOnClickListener
        }
        launch {
            context?.let {
                val mNote = Note(noteTitle, noteBody)
                // note == null means Inserting a new Note
                if (note == null) {
                    NoteDatabase(it).getNoteDao().addNote(mNote)
                    it.toast("Note Saved")
                } else {
                    // Update the note
                    mNote.id = note!!.id
                    NoteDatabase(it).getNoteDao().updateNote(mNote)
                    it.toast("Note Updated")
                }

                // after adding a note need to return to Home_Fragment as per the navigation
directions
                val action = AddNoteFragmentDirections.actionSaveNote()
                Navigation.findNavController(view).navigate(action)

            }
        }

    }
    }

    private fun deleteNote() {
        AlertDialog.Builder(context).apply {
            setTitle("Are you sure?")
            setMessage("You cannot undo this operation")
            setPositiveButton("Yes") {dialog, which ->
                launch {
                    NoteDatabase(context).getNoteDao().deleteNote(note!!)
                    val action = AddNoteFragmentDirections.actionSaveNote()
                    Navigation.findNavController(view!!).navigate(action)
                }
            }
            setNegativeButton("No") {dialog, which->

            }
        }.create().show()
    }
}
```