Practice Guide – 06 – Database Handling with Room

- **Storing the data retrieved from the API in a table called "Photo".**
- **This is executed only once until the database table is filled with data. Because due to the primary key we used in the Entity class, another execution of the app will crash the app. Due to the primary key integration violation.**
- **Therefore, this guide is only to summarize the lecture video and to save data in the database with Room just once.**

❖ **Open the project created for Practice Guide 04. You can download the project from the link here.**

❖ **Add dependencies**

➢ In the build.gradle file,

- within **plugins** put, (just copy this & paste there)

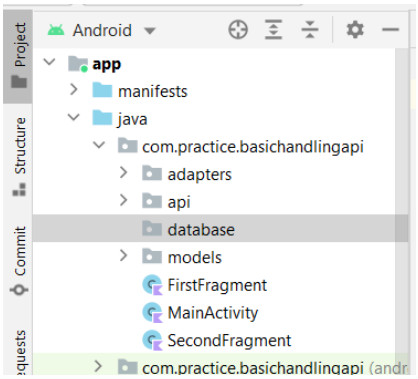
```
id 'kotlin-kapt'
```

- within **dependencies** put, (just copy this & paste there)

```
//Room
var room_version = '2.4.2'
implementation("androidx.room:room-runtime:$room_version")
annotationProcessor("androidx.room:room-compiler:$room_version")
kapt("androidx.room:room-compiler:$room_version")
```

❖ **Creating the database and relevant environment**

➢ Create a package named "database" in the project.



➢ Create Entity classes

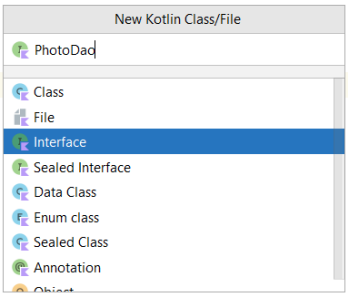
- Since we are going to save Photo objects, we need to have a Photo Entity class. However, since we are using the same project of Practice Guide 04, we can modify the **data class Photo** so that it becomes a Photo Entity.

NOTE: This will not affect to the data class used for the previous project. It will work as the same for it without any error.

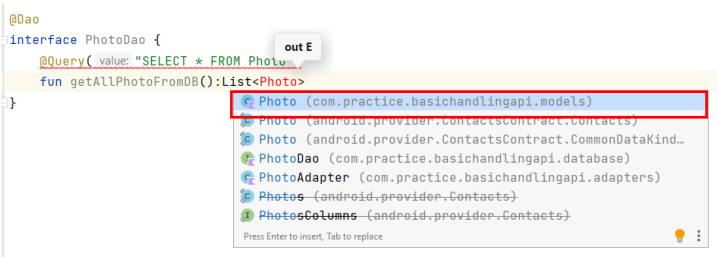
- Modify the **data class Photo** so that it looks like below.

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

@Entity
data class Photo(
    var albumId: Int,
    @PrimaryKey var id: Int,
    var title: String,
    var url: String,
    var thumbnailUrl: String
)
```

➢ Create DAO for the Entity created.



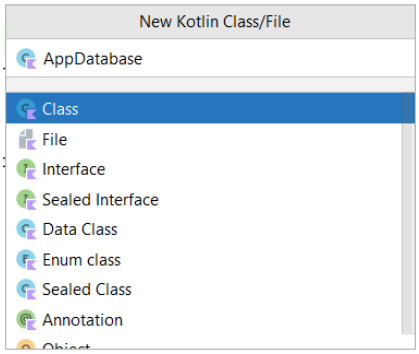
- Be careful to import the suitable **Query** Annotation and **Photo** Entity here.



- Finally, your **PhotoDao** will look like this.

```
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query
import com.practice.basichandlingapi.models.Photo

@Dao
interface PhotoDao {

    @Query( value: "SELECT * FROM Photo")
    fun getAllPhotoFromDB(): List<Photo>

    @Insert
    fun insertPhoto(vararg photo: Photo)

}
```

➢ Create the database kotlin file. Here we named it as "**AppDatabase**".



- Here a common design pattern called **Singleton Design Pattern** is used to instantiate database.
- Remember to declare all the abstract Dao classes created.
- And also, the entities as an array to the parameters of the **@Database** Annotation.
- Remember to give the which version of database that you are using.

- Finally, the code will look like below.

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.practice.basichandlingapi.models.Photo

@Database(entities = [Photo::class], version = 1)
abstract class AppDatabase: RoomDatabase() {
    abstract fun photoDao():PhotoDao

    companion object{
        //Singleton Instantiation
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context):AppDatabase{
            //If the INSTANCE is not null then, return it
            //If it is then, create the database
            return INSTANCE?: synchronized( lock: this){
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    name: "api-db"
                )
                    .fallbackToDestructiveMigration()
                    .allowMainThreadQueries()
                    .build()
                INSTANCE = instance
                instance ^synchronized
            }
        }
    }
}
```

If the version of the database is changed this helps to make the flow stabilize.

Disables main thread query check for Room

- Now database and its relations have been created. Let's use this database for functionalities.

❖ Storing the retrieved data from the API to the database

- Now go to the FirstFragment.kt (here only for this example) where we retrieved the photos from the API.
- There we are doing,
    - Call the database instance to the current view context.
    - Call the insertPhoto function to insert the retrieved Photos to the database.

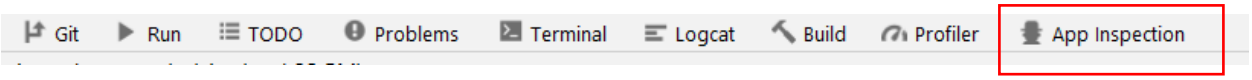
```
//save data in the database
val db = AppDatabase.getDatabase(view.context)
photosBody?.forEach { it: Photo
    db.photoDao().insertPhoto(it)
}
```

- Final code will look like this.

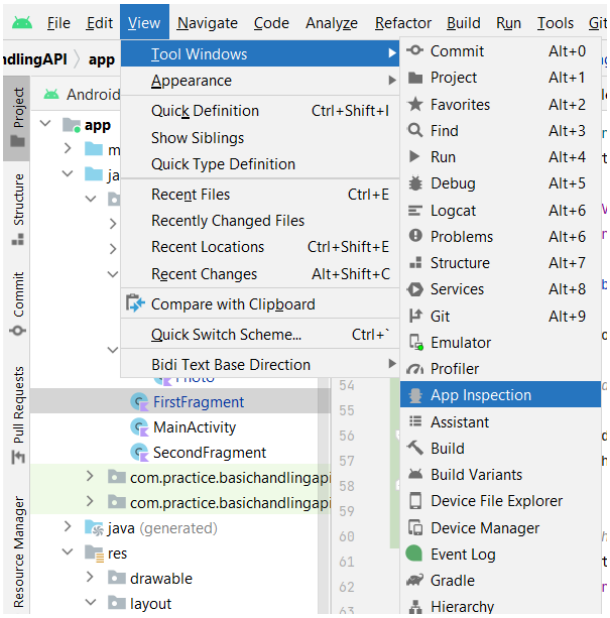
```
class FirstFragment : Fragment() {

    private var _binding: FragmentFirstBinding? = null

    // This property is only valid between onCreateView and
    // onDestroyView.
    private val binding get() = _binding!!

    private val retrofitAPIHandler = RetrofitAPIHandler.create()

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

        _binding = FragmentFirstBinding.inflate(inflater, container, attachToParent: false)
        return binding.root

    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.recyclerView.layoutManager = LinearLayoutManager(view.context)
        val photos = retrofitAPIHandler.getPhotos()

        photos.enqueue(object : Callback<List<Photo>>{
            override fun onResponse(call: Call<List<Photo>>, response: Response<List<Photo>>) {
                val photosBody = response.body()

                //save data in the database
                val db = AppDatabase.getDatabase(view.context)
                photosBody?.forEach { it: Photo
                    db.photoDao().insertPhoto(it)
                }

                //pass the data to the recyclerView via adapter
                val adapter = PhotoAdapter(photosBody!!, context: this,{position->onListItemClick(position)})
                binding.recyclerView.adapter = adapter

            }

            override fun onFailure(call: Call<List<Photo>>, t: Throwable) {
                Snackbar.make(view, text: "Failure in Callback", Snackbar.LENGTH_LONG)
                    .setAction( text: "Action", listener: null).show()
                Log.i( tag: "TAG", msg: "onFailure: Callback failed")
            }

        })
        binding.buttonFirst.setOnClickListener { it: View!
            findNavController().navigate(R.id.action_FirstFragment_to_SecondFragment)
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    private fun onListItemClick(position:Int){
        /*
        * In the next guide we will modify this code to trigger next Fragment
        * */
        Snackbar.make(requireView(), text: "Clicked on item ${position+1}", Snackbar.LENGTH_LONG)
            .setAction( text: "Action", listener: null).show()
        Log.i( tag: "TAG", msg: "onListItemClick: $position clicked")
    }
}
```

❖ **View data just stored in the database**

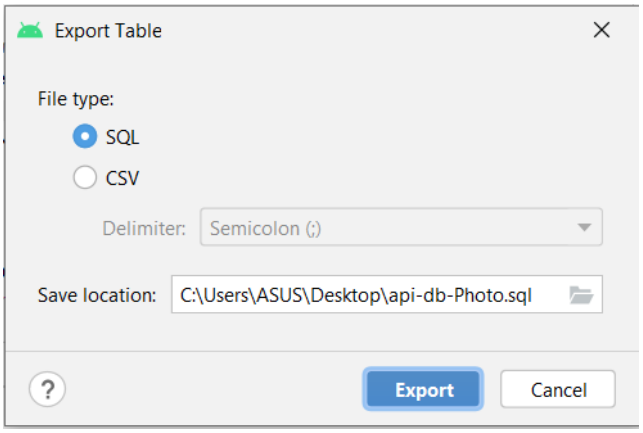
- Run the project successfully.
- Go to App Inspection tab in the bottom ribbon of Android Studio.



- OR otherwise **View->Tool Windows->App Inspection**

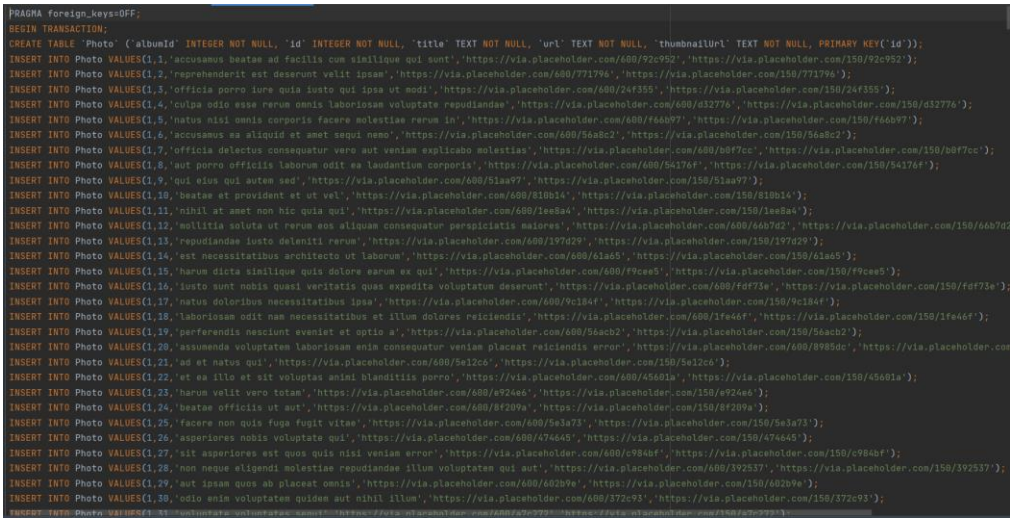


- Now expand the relevant database and look for your table.
- Right Click on the table name and select **export as file..**
- Then you will be prompt here.



- Select SQL and click **Export**.
- Then you can open the file with any editor and check for the data in it.

- If the data has been successfully stored then, you can see something like this with an editor software.



For your awareness,

- While the project is now running, if you again loaded the same fragment that storing the data in the database then, primary key integrity violation will happen.
- Then app will be crashed unless you have not handled it.

Optional Solution:

- You can handle the execution of the same data insertion part more than once.
- No need to worry for this. Try it after the exam.

❖ Summary

- Add plugin and dependencies
- Create the database package
- Create Entity classes
- Create DAO interfaces
- Create Database abstract class
- Call the database instance where you want.

❖ Follow these articles for your extra knowledge and FEAR

- Room Database with Kotlin. What is Room ? | by Çağnur Hacımahmutoğlu Parçal | Huawei Developers | Medium