# COM4510/6510
# Software Development for Mobile Devices

# Lab 6: MVVM, Live Data and Persistence

Temitope (Temi) Adeosun
The University of Sheffield
t.adeosun@sheffield.ac.uk

*Note: where the word "Update" appears in red font before a section, that section had been updated with information worth paying attention to since the last release of this document.*

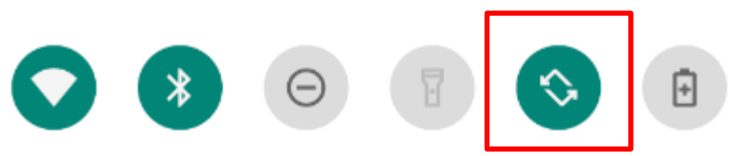
# Today's exercises

- Implement a simple but complete example using Jetpack Architectural components Room, ViewModel (for MVVM), LiveData
- Start working on your assignment
  - Extend the lab class of weeks 5 to use MVVM and LiveData (this will be left for you to practice towards getting ready for your assignment)


# MVVM, LiveData and Database

# But first…

- Let's check somethings from lab 4 and 5:
  - Run Lab4 and Lab5 solution codes. For each:
    - Add an image using the fab button as before.
    - Swipe down to open the notification draw and enable "Auto-rotate Screen"



- Then rotate the AVD by clicking one of the rotate buttons in the controls on the right

- Observe that the images you just added were removed for the lab 4 but not for lab 5
- Take a minute to consider why that might have happened (answer is on the next page, but try to answer it yourself before going to the next page)

- Handling configuration changes:
  Changes in screen orientation is one of the runtime configuration changes that Android handles by restarting the Activity.
    - This causes the onDestroy() and onCreate() to be called.
- In Lab 4, onCreate() calls initData(), which reloaded the drawable resources.
- In Lab 5, each time you add an image, the image's URI is saved to the database by the call to insertData() inside getImageData()
    - This allows the images to be reloaded when the initData is called in onCreate()

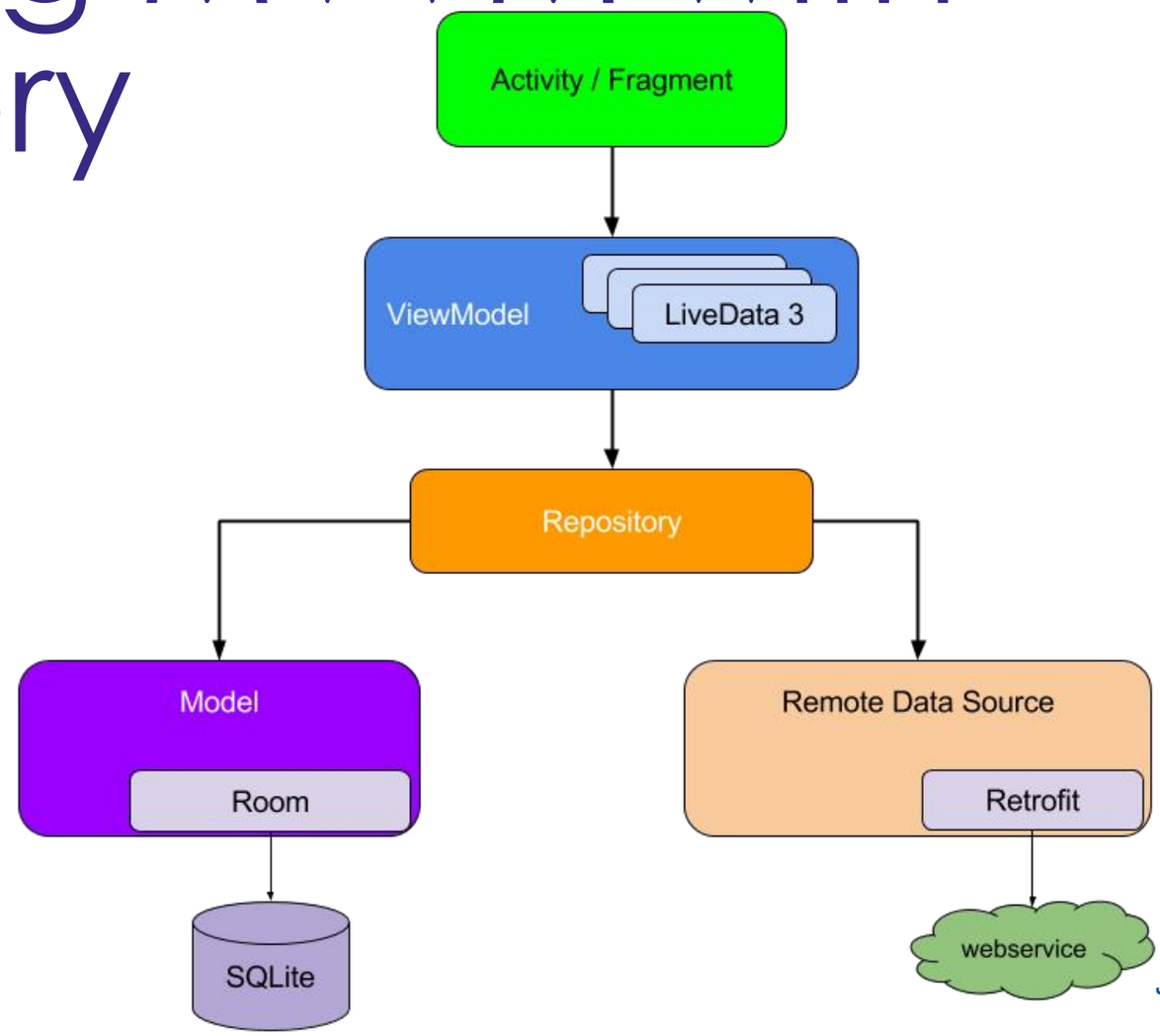
- Other events and configuration changes (phone calls, keyboard available) may similarly trigger the Android lifecycle events. These may happen at any point during your app's runtime.
- Lifecycle changes are managed by the framework and UI classes for Activities and Fragments are just glue classes that represent the contract between your apps and Android/framework.
  - The OS can destroy these glue classes at anytime (e.g. to save system resource) – it can be difficult to manage data loss when using resources you don't control (see some tips for doing so here - many of these tips however could result in larger UI classes)

- Week 4 discusses options for separation of concerns built into your app's architecture from ground up
- Another important design recommendation is to drive the UI element from models – models hold your app data and separate from the UI classes.
- Android Architecture component (Jetpack) provides strong support for this approach)
    - General principle best practice principles:
        - Separate concerns
        - Drive UI from a (persistent) model

- Neither Labs 4 or 5 do these – not best practice implementation!
  Other issues exist (can you identify them? mention in discussion board)
- Further, apps can get complex, interacting with REST services, processing large data, doing complex computations.

- What does Google recommend?
  - MVVM with Jetpack
    See: https://www.scalablepath.com/blog/recommended-architecture-for-android-apps/
  - Complete list of Jetpack libraries:
    *https://developer.android.com/jetpack/androidx/explorer*

# Extending MVVM with Repository




- Each component ONLY depends on the one below it
- Better single responsibility model, improved testability
- Repository just provide data for the ViewModel and can also help with caching data from the remote data source to the local data store.
  The Repository is an abstraction of the Model (in MVVM) as a data source (in this diagram Model and Remote data source) so the ViewModel doesn't know how data is retrieved. Possible to swapped data sources without changes to ViewModel

# Let's get started with the lab

- Download the starter code on Blackboard.
  - The app you are building will generate a random number for each button click and insert the number into the database. The intention is to show you MVVM + LiveData + Database with as little code as possible.
- Inspect the code carefully and make sure you understand what is happening.
  - Ask questions if you don't

# Add Room

- Room library

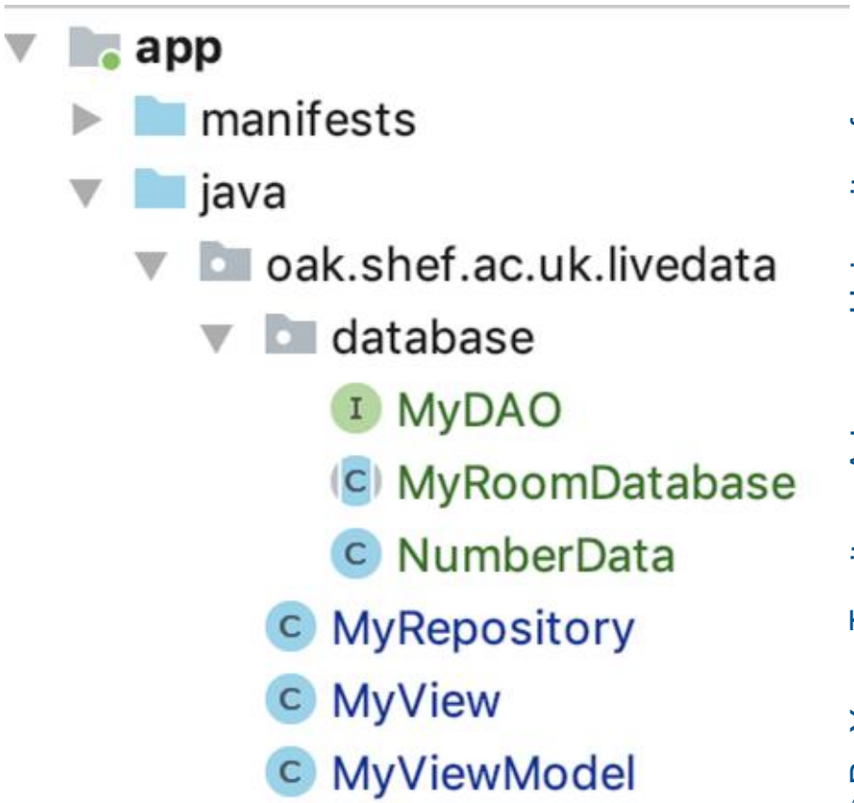
ADD TO THE GRADLE FILE the room library(Module: app)

```
// Room core library

implementation "androidx.room:room-runtime:2.3.0"
annotationProcessor "androidx.room:room-compiler:2.3.0“

// To use Kotlin Symbolic Processing (KSP)
// - will replace kapt,faster and preferred
ksp "androidx.room:room-compiler:2.3.0“

// To use Kotlin Annotation Processing, kapt
kapt "androidx.room:room-compiler:2.3.0“
```

**In plugin section of app gradle file:**

```
plugins {
…
// To use ksp
id "com.google.devtools.ksp"
version "1.5.31-1.0.0
// To use kapt
id 'kotlin-kapt’
…
}
```

- Same as last week, create the 3 Room component classes
  - RoomDatabase
  - DAO
  - Entity



# RoomDatabase

Our Entity Class

```
@Database(entities = [NumberData::class], exportSchema = false, version = 1)
abstract class MyRoomDatabase: RoomDatabase() {
    abstract fun myDao(): MyDAO

    companion object {
        // marking the instance as volatile to ensure atomic access to the variable
        private var INSTANCE: MyRoomDatabase? = null
        private val mutex = Mutex()

        fun getDatabase(context: Context): MyRoomDatabase? {
            if (INSTANCE == null) {
                runBlocking {
                    withContext(Dispatchers.Default) {
                        // add lock to MyRoomDatabase class
                        mutex.withLock(MyRoomDatabase::class) {
                            INSTANCE = databaseBuilder(
                                context.applicationContext,
                                MyRoomDatabase::class.java, "number_database"
                            )   // Wipes and rebuilds instead of migrating if no

                                // Migration is not part of this codelab.
                            .fallbackToDestructiveMigration()
                            .addCallback(sRoomDatabaseCallback)
                            .build()
                        }
                    }
                }
                return INSTANCE
            }
    ….
```

extends RoomDatabase

declares the DAO and the INSTANCE

Usual boilerplate code

Update: Note the use of Mutex, a Coroutine lock, which grants mutually exclusive access without blocking. runBlocking() blocks the calling thread until the current (coroutine) job is finished

Callback added to populate database

# RoonDatabase (cont.)

…

```
        /**
         * Override the onOpen method to populate the database.
         * For this sample, we clear the database every time it is created or opened.
         *
         * If you want to populate the database only when the database is created for the
1st time,
         * override RoomDatabase.Callback()#onCreate
         */
        private val sRoomDatabaseCallback: RoomDatabase.Callback = object : Callback() {
            override fun onOpen(db: SupportSQLiteDatabase) {
                super.onOpen(db)
                // do any init operation about any initialisation here
            }
        }
    }
}
```

Callback implementation for database population

# The Entity

It is an Entity

```
@Entity
class NumberData(

@PrimaryKey(autoGenerate
= true)
    val id: Int = 0,
    var number: Int) {
}
```

Only one field containing the number, other is automatic id

# The DAO

- It connects the database operations to the SQLite operations

```
@Dao
interface MyDAO {
    @Insert
    fun insertAll(vararg numberData: NumberData?)

    @Insert
    fun insert(numberData: NumberData?): Long

    @Delete
    fun delete(numberData: NumberData?)

    // it selects a random element
    @Query("SELECT * FROM numberData ORDER BY RANDOM() LIMIT 1")
    fun retrieveOneNumber(): LiveData<NumberData?>?

    @Delete
    fun deleteAll(vararg numberData: NumberData?)

    @Query("SELECT COUNT(*) FROM numberData")
    fun howManyElements(): Int
}
```

Specifying Long return type will cause the Id of the inserted row to be returned after insertion

It selects randomly one of the numbers in the DB

Note it returns LiveData – this is key!

# Update: The ViewModel

- MyViewModel extends androidx.lifecycle.AndroidViewModel
  - AndroidViewModel extends ViewModel, but exposes the application context - useful when you need context access.
- ViewModel (consequently AndroidViewModel) is lifecycle aware and exposes a CoroutinScope – viewModelScope
- Note we are no longer using GlobalScope (as found in Lab5). Avoid GlobalScope because it is not fully controllable – its lifecycle is not tied to an activity/ViewModel and may keep running well after an activity is destroyed.
  - Also, all context jobs created using GlobalScope share the same context, which isn't available for cancellation or destroy by your code.

# The Repository

- Update: The Repository is implemented as a simple Kotlin class that directly interacts with the RoomDatabase and DAO, makes the data available to the ViewModel, without exposing the detail of getting the data
- Repository establishes connection to the database instead of a derived Application class in previous lab.

```
class MyRepository(application: Application) : ViewModel() {
    private var mDBDao: MyDAO? = null

    init {
        val db: MyRoomDatabase? = MyRoomDatabase.getDatabase(application)
        if (db != null) { mDBDao = db.myDao() }
    }
```

- It must now insert the number into the database every time it generates it.
  - It will not assign it to a local variable
    - there are no local variable!

# The Repository (cont.)

- Add a methods:
  - getNumberData() to return a NumberData
  - it will just return whatever is returned to by the DAO
  - Note the DAO returns a LiveData<NumberData>
- LiveData makes the data observable
  - Note: in Kotlin, you also may also use Flow. But Flow is better suited for streaming or constantly changing data.

```
/**
 * it gets the data when changed in the db and returns it to the ViewModel
 * @return
 */
    fun getNumberData(): LiveData<NumberData?>? {
            return mDBDao?.retrieveOneNumber()
     }
```

# The Repository (cont.)

It inserts a new NumberData into the DB
NOTE: database tasks MUST run on a separate thread.

```
suspend fun generateNewNumber() {
    val r = Random()
    val i1 = r.nextInt(10000 - 1) + 1
    InsertAsyncTask(mDBDao).backgroundTask(NumberData(i1))
}
```

Updated: Provided for you already:
insertInBackground starts a Coroutine to Insert the data into DB. You may also use withContext() safely here, which doesn't start a coroutine but still lets you start the block in a new thread

```
companion object {
    private val scope = CoroutineScope(Dispatchers.IO)
    private class InsertAsyncTask(private val dao: MyDAO?) : ViewModel() {
        suspend fun insertInBackground(vararg params: NumberData) {
            scope.launch {
                for(param in params){
                    val insertedId: Int? = this@InsertAsyncTask.dao?.insert(param)?.toInt()
                    // you may want to check if insertedId is null to confirm successful
insertion
                    //Log.i("MyRepository", "number generated: " + param.number.toString()
                    //        + ", inserted with id: " + insertedId.toString() + "")
                }
            }
        }
    }
}
```

# The View - MyView

- In this example, vieBinding is used. Enabled by specifying in app gradle file:

```
buildFeatures {
    viewBinding true
}
```

This causes binding classes to be created to provide strongly typed access to view objects with using findViewById

- Binding classes are named after layout file. So, a layout main_activity.xml will have a binding class MainActivityBinding
- Binding class is instantiated by calling inflate() see example in class MyView.

# The View – MyView (cont)

- Instantiate ViewModel using ViewModelProvider:

```
this.myViewModel =
ViewModelProvider(this)[MyViewModel::class.java]
```

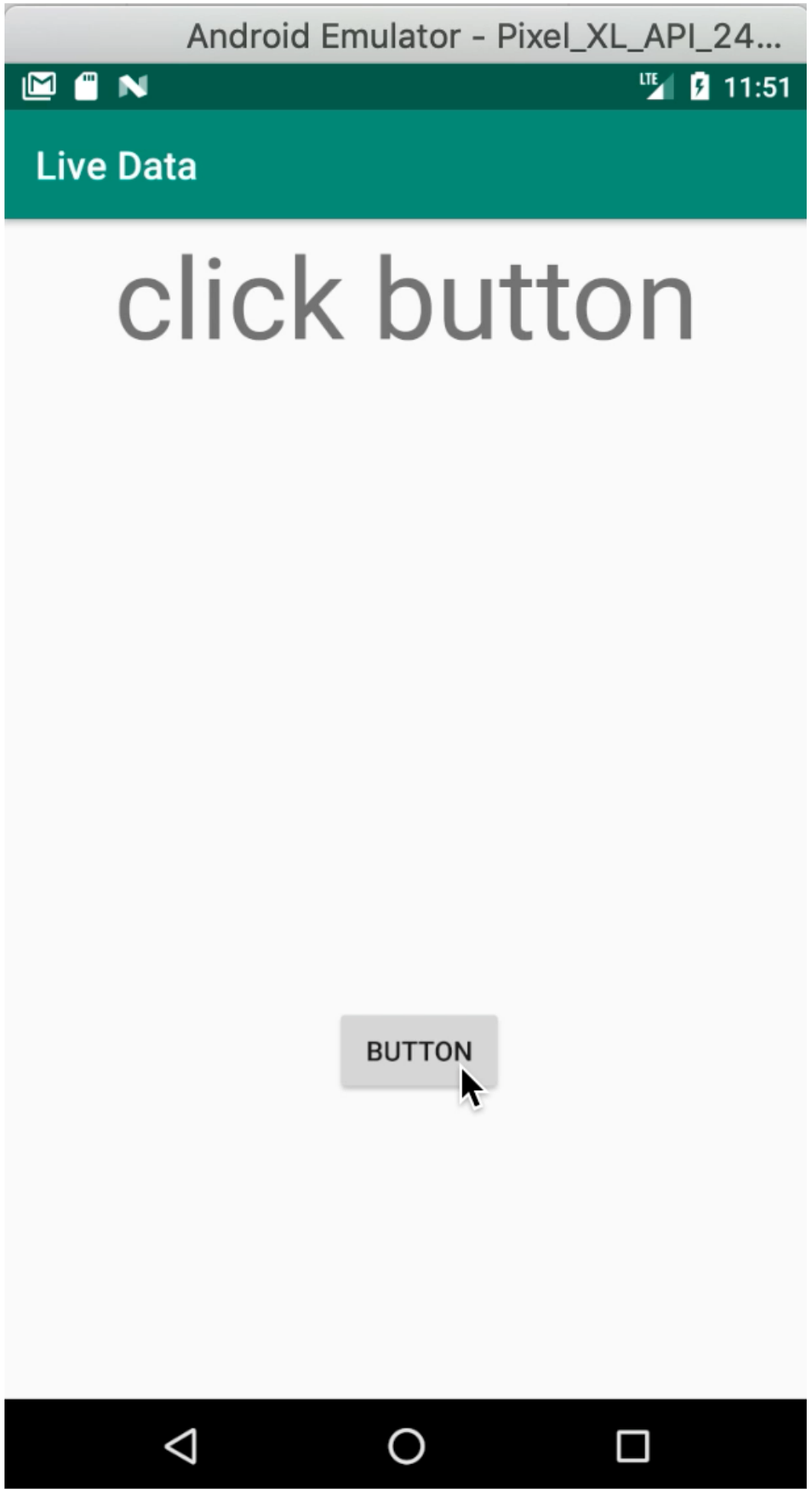
- Create an observer to observe the data published by the Repository
  Note: the first time you run the app, the database is empty and null will be returned, you need to decide what to do here for your app.

```
this.myViewModel!!.getNumberDataToDisplay()!!.observe(this,
    //  create observer, whenever the value is changed the function block
below will be called
    { newValue ->
        val tv: TextView = findViewById(R.id.textView)
        // if database is empty
        if (newValue == null) tv.text = "click button"
        else tv.text = newValue.number.toString()
    }
)
```

The University Of Sheffield.

# That is it!

App should run and look like this



# Run the app


Open the App inspector before run to see your database

Open a query tab and type *select * from NumberData.*

*Select the live update checkbox to see database update for each button click*

*Try the rotation exercise you did at the start with Labs 4 and 5. What did notice? Do you have an explanation why might have happened?*

*Note that with LiveData, there is no need to write boiler plate code to monitor changes to the data and propagate this changes to the view.*


# How to Debug

- When using JetPack and MVVMs, any Exception is likely return “cannot instantiate the ViewModel
- Also, coroutines can sometimes be diffifult to debug because
- Suggestions:
- For Coroutines, right click the breakpoint and select the ***All*** option button
- insert Log.i, Log.d, and Log.e traces to identify the area where the error is
- insert try/catches around the potential areas of error

```
try{
   /// code
   } catch (e: Exception){
      Log.e(“Class name”, “descriptive string “+ e.message)
      }
```