Room

Download the full App Created in this Guide:

https://drive.google.com/file/d/1tVbM7C7kIt0rw3FcmelwZ1GON2GxJiGR/view?usp=sharing

Let's say we need to persist the data that added by the user. So we want a persistence library like *Room*. It is an Object-Mapping library that provides an abstraction layer over SQLite that doesn't try to hide SQLite but rather embrace it.

There are some reasons why we use room for data persistence:

- It's based on SQLite so we can write SQL queries. Don't forget that Android supports SQLite as a proven technology from the day one.
- With Room we can have observability, because it can return LiveData objects and does the object mapping for you.
- Room creates the database schema using your entity definitions and does the sql operations like *insert*, *update* and *delete* using annotation processing, resulting in lesser boilerplate code.
- Room speaks SQL. So it knows whether you made a typo or did something wrong in queries at compile time.
- You can create abstract suspended functions and let room create all the logic for you.

Other SQLite Android libraries:
1. Sugar ORM – This is an object relational mapper that wrap SQLite database. It map sqlite table to a java plain object.
2. Realm Database – It provides offline-first functionality & data persistence through an easy-to-use API.
3. SQLBrite – A lightweight wrapper around SQLiteOpenHelper which introduces reactive stream semantics to SQL operations

Primary components

There are three major components in Room:

- Data entities that represent tables in your app's database.
- Data access objects (DAOs) that provide methods that your app can use to query, update, insert, and delete data in the database.
- The database class that holds the database and serves as the main access point for the underlying connection to your app's persisted data.

First Step - Add the latest room library

Visit this page and import the room library to your app Gradle project file:

```
def room_version = "2.4.2"

  implementation "androidx.room:room-runtime:$room_version"
  kapt "androidx.room:room-compiler:$room_version"
```

Important Note:
In the app Gradle Add the plugin :
id 'kotlin-kapt'

Please note we replaced the room annotationProcessor to kapt - Kapt is the Kotlin Annotation Processing Tool. If you want to be able to reference generated code from Kotlin, you need to use kapt.

Second Step - Define your Entities
Use the **@Entity** annotation to define a new entity - this you will do for the basic Kotlin or Java class you want to save in your database - you can optionally give the table a name using (tableNmae = "[table name]") the default table name is the class name.
Define your primary key using @PrimaryKey next to the property. That will serve as your primary key. If you don't have a unique key to your objects like emails you can set it to be auto generated use (autoGenerate = true) next to it.

If you want a different column name in the data base from the property name use @ColumnInfo(name = ["Your name"])

It's recommended you always use the @ColumnInfo annotation as it gives you more flexibility to rename the members without having to change the database column names. Changing the column names leads to a change in the database schema and therefore you need to implement a migration or specific instructions not to implement migration.

For example:

```kotlin
@Entity(tableName = "items_table")
data class Item(
    @ColumnInfo(name = "content")
    val content:String,

    @ColumnInfo(name = "description")
    val description:String,

    @ColumnInfo(name = "image")
    val image:String?)
{

    @PrimaryKey(autoGenerate = true)
    var id : Int = 0
}
```

Third step - define you Dao classes

Dao classes will allow you to abstract the database communication in a more logical layer which will be much easier to mock in tests (compared to running direct sql queries). It also automatically does the conversion from Cursor to your application classes so you don't need to deal with lower level database APIs for most of your data access.

Room also verifies all of your queries in Dao classes while the application is being compiled so that if there is a problem in one of the queries, you will be notified instantly while you are writing it.

The class marked with @Dao should either be an interface or an abstract class. At compile time, Room will generate an implementation of this class when it is referenced by a Database.
*An abstract **@Dao** class can optionally have a constructor that takes a Database as its only parameter.*
It is recommended to have multiple Dao classes in your codebase depending on the tables they interact.

```
@Dao
interface ItemsDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun addItem(item: Item)

    @Delete
    fun deleteItem(vararg item: Item)

    @Update
    fun update(item: Item)

    @Query( value: "SELECT * from items_table ORDER BY content ASC")
    fun getItems() : LiveData<List<Item>>

    @Query( value: "SELECT * from items_table WHERE content LIKE :title")
    fun getItem(title:String) : Item
}
```

onConflict annotation parameter signifies what to do if a conflict happens on insertion. It can take the following values:

- **OnConflictStrategy.REPLACE** : To replace the old data and continue the transaction.
- **OnConflictStrategy.ABORT** : To abort the transaction. The transaction is rolled back.
- **OnConflictStrategy.NONE** : To ignore the conflict.

Third step - create you database

Now create the AppDatabase class to hold the database. AppDatabase defines the database configuration and serves as the app's main access point to the persisted data. The database class must satisfy the following conditions:

- The class must be annotated with a @Database annotation that includes an entities array that lists all of the data entities associated with the database.
- The class must be an abstract class that extends RoomDatabase.
- For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.

In order to make sure we don't have multiple database instances open at the same time we define a RoomDatabase instance in the companion object of our class. We need the application context to initialized the database. So the best way to handle this is to and a getDatabase function that receives the context

and builds the database.

We'll define an abstract method that returns the ItemsDao. Everything is abstract because Room is the one that generates the implementation for us.

exportSchema
You can set annotation processor argument to tell Room to export the schema into a folder. Even though it is not mandatory, it is a good practice to have version history in your codebase and you should commit that file into your version control system (but don't ship it with your app!).
So if you don't need to check the schema and you want to get rid of the warning, just add exportSchema = false to your RoomDatabase

@Volatile - Volatile means, it will not be stored in the local cache. Meaning: writes to this field are immediately made visible to other threads.

```
@Database(entities = arrayOf(Item::class),version = 1, exportSchema = false)
abstract class ItemsDatabase : RoomDatabase() {

    abstract fun itemsDao() : ItemsDao

    companion object {

        @Volatile
        private var instance:ItemsDatabase? = null

        fun getDatabase(context:Context) : ItemsDatabase? {

            if(instance == null) {
                synchronized(ItemsDatabase::class.java) {

                    instance = Room.databaseBuilder(context.applicationContext,
                        ItemsDatabase::class.java, name: "items_database").build()
                }
            }
            return instance
        }
    }
}
```

```
//or in more Kotlin way
fun getDatabase(context:Context) = instance ?: synchronized( lock: this) {
    Room.databaseBuilder(context.applicationContext,ItemsDatabase::class.java, name: "items_database")
        .allowMainThreadQueries().build().also { instance = it }
}
```

Forth step(optional)
create your helper class the gives a single access point to your database
You can think of repository as the single access point for getting the data. The

class will include all the functions from which we can get all the data.

```
class ItemsRepository(application: Application) {

    private var itemsDao:ItemsDao?

    init {
        val db = ItemsDatabase.getDatabase(applicat
        itemsDao = db?.itemsDao()
    }

    fun getItems() = itemsDao?.getItems()

    fun addItem(item: Item) {
            itemsDao?.addItem(item)
    }
}
```

Please note that while the fetching that return LiveData is done on a background thread automatically, adding the item is done on the application main thread! until we use coroutines you can use this but be sure to add the .allowMainThreadQueries() to your ROOM database builder. If you don't do this the app will crash and you will get an error message "Cannot access database on the main".

```
instance = Room.databaseBuilder(context.applicationContext,
    ItemsDatabase::class.java, name: "items_database").allowMainThreadQueries().build()
```

Later we will solve this by adding Coroutines to out project.

Fifth step
Update your view model
Here we need to extend the Android ViewModel because we need the application instance to give to our database

```
class ItemViewModel(application: Application) : AndroidViewModel(application) {

    private var repository = ItemsRepository(application)

    fun getItems() = repository.getItems()

    fun addItem(item : Item) {
        repository.addItem(item)
    }
}
```

And That's it you have integrated ROOM. Go ahead and check your implementation by creating the full project

Let's add our ItemViewModel

In our case we want to share one ViewModel for the whole activity because it makes sense. We have three Fragments all need to access the same information: one shows the list, one adds an item to it and one shows a single item from the list. The ViewModel will hold all the items as a LiveData property and the chosen item also as LiveData, it will have functions to add an item, delete an item, delete all items and set the chosen item. Please note that we will inherit from the **AndroidViewModel** because we need the Context in order to create the repository private instance who will serve as a single access point for our data.
Our final View Model will look like this (please note that while the chosen item must be a mutable live data we only expose it as Live Data in order to keep our data persistent):

```
class ItemsViewModel(application: Application)  : AndroidViewModel(application){

    private val repository = ItemRepository(application)

    val items : LiveData<List<Item>>? = repository.getItems()

    private val _chosenItem = MutableLiveData<Item>()
    val chosenItem : LiveData<Item> get() = _chosenItem

    fun setItem(item:Item) {
        _chosenItem.value = item
    }

    fun addItem(item: Item) {
        repository.addItem(item)
    }

    fun deleteItem(item:Item) {
        repository.deleteItem(item)
    }

    fun deleteAll() {
        repository.deleteAll()
    }
}
```

Note that we also added a delete all function to the repository and to the Dao:

```
fun deleteAll() {
    itemDao?.deleteAll()
}
```

```
@Query( value: "DELETE FROM items")
fun deleteAll()
```

Now let's go back the the UI - To all fragment get the view Model bounded to the activity scope and because all of them lives in the same activity there will be one instance of it that will be shared by all of them. So add this line to the top of **each fragment**:

```
private val viewModel : ItemsViewModel by activityViewModels()
```

Go to the **AllItemsFragment** In the **onViewCreated** get the viewModel's items LiveData and observe it. In the callback which is called with the updated list of items pass it to the adapter and implement the callback functions (remove the code with the ItemManager and replace it with this):

```
viewModel.items?.observe(viewLifecycleOwner) {  it: List<Item>!

    binding.recycler.adapter = ItemAdapter(it, object : ItemAdapter.ItemListener  {

        override fun onItemClicked(index: Int) {

            Toast.makeText(requireContext(),
                 text: "${it[index]}",Toast.LENGTH_SHORT).show()
        }

        override fun onItemLongClicked(index: Int) {
            viewModel.setItem(it[index])
            findNavController().navigate(R.id.action_allItemsFragment_to_detailItemFragment)
        }
    })
    binding.recycler.layoutManager = LinearLayoutManager(requireContext())

}
```

Before finishing the AllItemsFragment go ahead to your adapter and add a function that will return an Item according to the position, because in the fragment upon swiping we get the position but we need to pass an Item to the viewModel delete function. So we will add a function to get an item according to its position(in ItemAdapter):

```
fun itemAt(position:Int) = items[position]
```

Now on the onSwipe of the ItemTouchHelper use this function:

```
    override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {
        val item = (binding.recycler.adapter as ItemAdapter).itemAt(viewHolder.adapterPosition)
        viewModel.deleteItem(item)
       // ItemManager.remove(viewHolder.adapterPosition)
      //  binding.recycler.adapter!!.notifyItemRemoved(viewHolder.adapterPosition)
    }
}).attachToRecyclerView(binding.recycler)
```

In the **AddItemFragment** remove the ItemManger access and replace it with the viewModel call (remember you added it as a property before):

```
val item  = Item(binding.itemTitle.text.toString(),
                 binding.itemDescription.text.toString(),imageUri.toString())
// ItemManager.add(item)

viewModel.addItem(item)
```

And in the DetailItemFragment after the view had been created observe your chosen item live data and update your UI (what was previously located in the arguments let scope):

```
viewModel.chosenItem.observe(viewLifecycleOwner) { it: Item!
    binding.itemTitle.text = it.title
    binding.itemDesc.text = it.description
    Glide.with(requireContext()).load(it.photo).circleCrop()
        .into(binding.itemImage)
}
```

Adding a Menu

In order to add an action/option menu to the top of the AllItemsFragment we create the following xml file under the resource menu folder:


```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item android:id="@+id/action_delete"
        android:title="Delete all"
        android:icon="@drawable/ic_baseline_delete_outline_24"
        app:showAsAction="always"/>
</menu>
```

This menu item contains an id, a vector asset we added before (just right click on the res folder choose new and then choose vector asset, in the new window choose a trash bin from the clipart and click finish), and a showAsAction attribute set to always means it will be shown all the rime on the menu and not under the thee dots (try giving different values).

In the AllItemsFragment where the menu is shown override these two functions:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.main_menu,menu)
    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    if(item.itemId == R.id.action_delete) {
        val builder = AlertDialog.Builder(requireContext())
        builder.setTitle("Confirm delete")
            .setMessage("Are you sure you want to delete all items?")
            .setPositiveButton( text: "Yes")
            { p0, p1 ->
                viewModel.deleteAll()
                Toast.makeText(requireContext(), text: "Items deleted",Toast.LENGTH_SHORT).show()
            }.show()
    }
    return super.onOptionsItemSelected(item)
}
```

The first one will be called by the system when its time to create the option menu and we will use the given menu inflater to inflate our own xml menu file we just dud to the empty menu the os gives us.
The second function is called upon pressing the action menu items and after presenting a confirmation dialog we will use our viewModel deleteAll function.

Please note that in order for the fragment to show the Manu you must add this line in the onCreateView of the AllItemsFragment function:

```
setHasOptionsMenu(true)
```

this is only needed when presenting the menu in a fragment and not in the activity - if we would have presented the menu in the activity it would have existed throughout all go the fragments and we don't want that.

And that's it the project is finished run and test your app.

Please note that we still allow queries too run on the application main thread:

```
Room.databaseBuilder(context.applicationContext, ItemDataBase::class.java, name: "items_db")
    .allowMainThreadQueries().build()
```

Try to remove this line and test your app.
Please note the the getItems works just fine because it returns LiveData and LiveData by default is doing all of its work on the Dispatchers.IO group of threads which are background thread ads and not on the main thread but try to add an item and see what happens... YES the app crashed and ion the logical you can't find the following message:

```
    java.lang.IllegalStateException: Cannot access database on the main thread since it
may potentially lock the UI for a long period of time.
```

And that's why we need to study Coroutines (Come back to this tutorial afterwards).

Lets improve our background work(After the co-routine chapter):

The first solution is for repository to implement CoroutineScope and override CoroutineContext to operate in IO Thread.

```
class ItemsRepository(application: Application) : CoroutineScope {

    override val coroutineContext: CoroutineContext
        get() = Dispatchers.IO

    private var itemsDao:ItemsDao?

    init {
        val db = ItemsDatabase.getDatabase(application)
        itemsDao = db?.itemsDao()
    }

    fun getItems() = itemsDao?.getItems()

    fun addItem(item: Item) {
        launch { this: CoroutineScope

            itemsDao?.addItem(item)
        }
    }
}
```

Now you can remove the **allowMainThreadQueries**() from your database instance initialization and go ahead and run your app and try adding an item... No crash!
Try deleting and it carshare s again, so do the same for the delete functions they are not returning LiveData:

```
fun deleteItem(item: Item) {
    launch { this: CoroutineScope
        itemDao?.deleteItem(item)
    }
}

fun getItem(id:Int)   = itemDao?.getItem(id)

fun deleteAll() {
    launch { this: CoroutineScope
        itemDao?.deleteAll()
    }
}
```

The problem with this solution is that it is not subject to the principle of *structured concurrency* *meaning there is no actual scope confined to any lifecycle for these coroutines. So although it works we can make it better.*

The best option is use the ROOM coroutine KTX extensions and our view model scope.

Room nows comes with coroutine support. DAO methods can now be marked as suspended to ensure that they are not executed on the main thread.
We can make the Dao addItem() function to be suspended and then Room will generate the code using coroutines by himself but we must call it from another suspended function or a coroutine context so we will use the viewModel scope to execute it.

Just add the following dependency to your app grade file:
implementation("androidx.room:room-ktx:$room_version")

And make the DAO insert, update and delete functions suspended:

```
@Dao
interface ItemDao {

    @Insert(onConflict = OnConflictStrategy.RE
    suspend fun addItem(item:Item)

    @Delete
    suspend fun deleteItem(vararg  items:Item)

    @Update
    suspend fun updateItem(item: Item)

    @Query( value: "DELETE FROM items")
    suspend fun deleteAll()

    @Query( value: "SELECT * FROM items ORDER BY
    fun getItems() : LiveData<List<Item>>

    @Query( value: "SELECT * FROM items WHERE id
    fun getItem(id:Int) : Item
```

Thats it under the hood ROOM automatically replaces this auto implemented synchronous code:

```
@Override
public void insertUserSync(final User user) {
  __db.beginTransaction();
  try {
    __insertionAdapterOfUser.insert(user);
    __db.setTransactionSuccessful();
  } finally {
    __db.endTransaction();
  }
}
```

With this:

```
@Override
public Object insertUserSuspend(final User user,
    final Continuation<? super Unit> p1) {
  return CoroutinesRoom.execute(__db, new Callable<Unit>() {
    @Override
    public Unit call() throws Exception {
      __db.beginTransaction();
      try {
        __insertionAdapterOfUser.insert(user);
        __db.setTransactionSuccessful();
        return kotlin.Unit.INSTANCE;
      } finally {
        __db.endTransaction();
      }
    }
  }, p1);
}
```

The generated code ensures that the insert happens off of the UI thread. In our suspend function implementation, the same logic from the synchronous insert method is wrapped in a `Callable`. Room calls the `CoroutinesRoom.execute` suspend function, which switches to a background dispatcher, depending on whether the database is opened and we are in a transaction or not. If we check the CoroutinesRoom.execute() implementation, we see that Room moves callable.call() to a different CoroutineContext. This is derived from the executors you provide when building your database or by default will use the Architecture Components IO Executor.

So the actual changes in our code is making the Dao functions suspended and because it is called from the repository functions we should make them also suspended and execute it from the viewModelScope:

```
suspend fun addItem(item:Item) {
    //launch {
        itemDao?.addItem(item)
    //}
}

suspend fun deleteItem(item: Item) {
    // launch {
        itemDao?.deleteItem(item)
    // }
}

fun getItem(id:Int)  = itemDao?.getItem(id)

suspend fun deleteAll() {
    // launch {
        itemDao?.deleteAll()
    //}
}
```

```
fun addItem(item: Item) {
    viewModelScope.launch { this: CoroutineScope
        repository.addItem(item)
    }

}
fun deleteItem(item:Item) {
    viewModelScope.launch { this: CoroutineScope
        repository.deleteItem(item)
    }
}

fun deleteAll() {
    viewModelScope.launch { this: CoroutineScope
        repository.deleteAll()
    }
}
```

viewModelScope is a Kotlin extension property on the ViewModel class. It is a CoroutineScope that is cancelled once the ViewModel is destroyed (when onCleared() is called). Thus when you're using a ViewModel, you can start all of your coroutines using this scope.

Please note that @Transaction methods can also be suspended and they can call other suspended DAO functions:

```
@Dao
abstract class UsersDao {

    @Transaction
    open suspend fun setLoggedInUser(loggedInUser: User) {
        deleteUser(loggedInUser)
        insertUser(loggedInUser)
    }

    @Query("DELETE FROM users")
    abstract fun deleteUser(user: User)

    @Insert
    abstract suspend fun insertUser(user: User)
}
```

Room offers allot of functionality and flexibility than what we've covered — you can define how Room should handle database conflicts, you can store types that otherwise, natively with SQLite can't be stored, like Date, by creating TypeConverters, you can implement complex queries, using JOIN and other SQL functionality, create database views, pre-populate your database or trigger certain database actions whenever the database is created or opened.

For more reading please refer to
https://developer.android.com/training/data-storage/room