Lab 11 - Room Database

`COMP4107 - SDDT - HKBU - Spring2023`

Today we'll continue building our InfoDay app using Jetpack Compose. We'll discuss how to **work with an embedded database**.

Clone your project from GitHub https://classroom.github.com/a/t86LHCCf and let's keep developing.

Jetpack Room is a library provided by Android that makes it easier for developers to work with **SQLite databases** on Android devices. SQLite is a popular database engine that is widely used in mobile applications, and is the **default database engine used by Android**.

Room provides an **abstraction layer over SQLite**, allowing developers to create, read, update, and delete records in a database **using simple annotations and queries**.

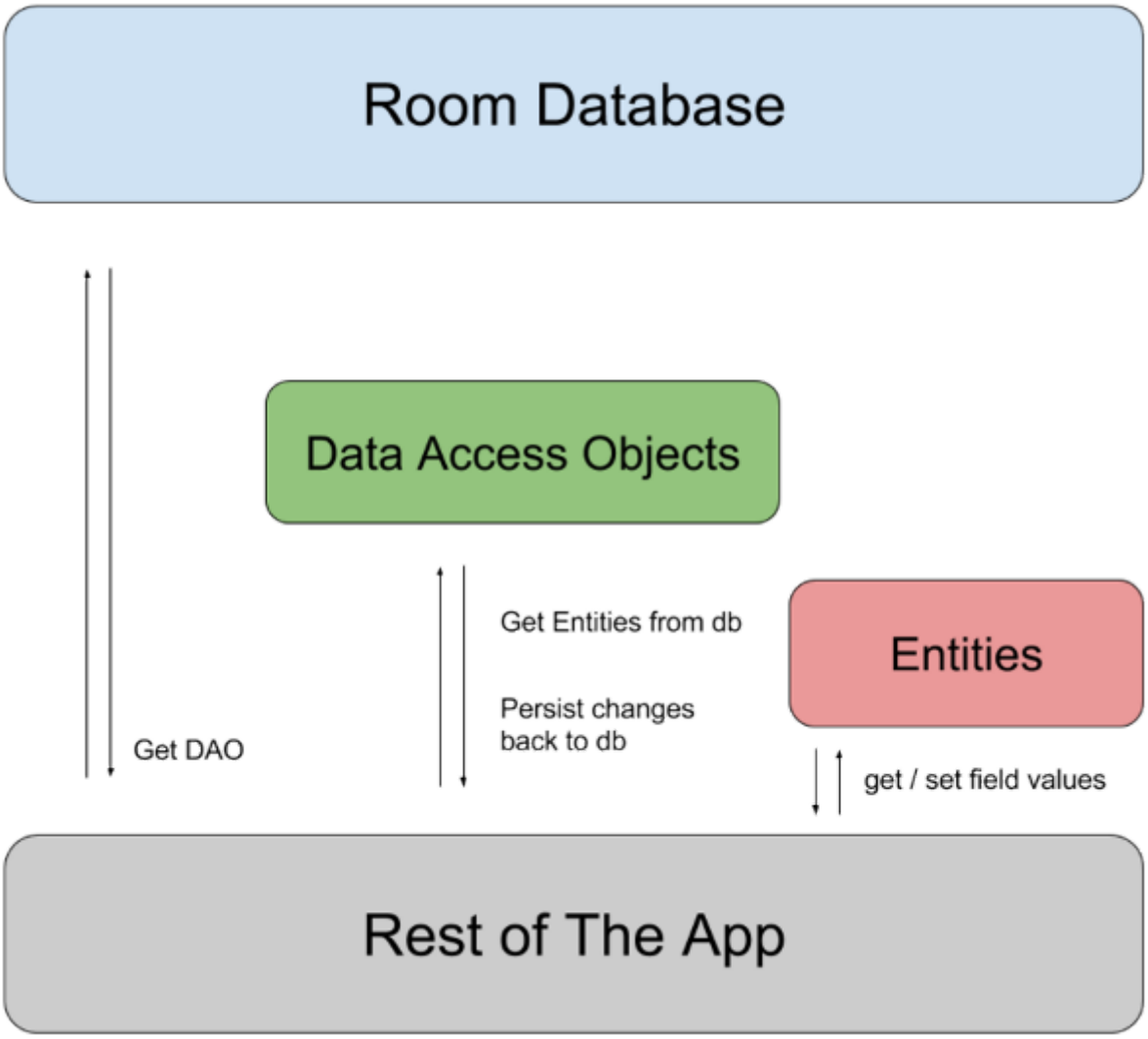
Room also provides **compile-time verification of SQL queries**, making it less error-prone than traditional SQLite approaches. Additionally, Room **supports the use of LiveData objects**, which enables developers to observe changes in the database and automatically update the user interface when the underlying data changes.

Overall, Room simplifies the process of working with SQLite databases in Android applications and provides a more robust and reliable solution for managing data.

There are **three main components** in Jetpack Room:

1. **Entity**: This **represents a table in the database** and is defined using a Java or Kotlin class. Each entity **represents a row in the table** and contains **fields** that map to columns in the table.
2. **DAO (Data Access Object)**: This is **an interface that defines the methods for accessing the database**. It contains methods for inserting, updating, deleting, and querying the data in the database.
3. **Database**: This is an **abstract class that provides the database instance** and serves as the main access point for the database. It contains methods for defining the entities and version of the database, and it also provides a way to create and access the DAOs.

Together, these three components form the backbone of Jetpack Room and make it easier for developers to work with SQLite databases in Android applications.



We aim to transform the SQLite file into a Room database.
First, we add the following to the `build.gradle` dependencies.

```
def room_version = "2.5.0"

implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"

// To use Kotlin annotation processing tool (kapt)
kapt "androidx.room:room-compiler:$room_version"
// Kotlin Extensions and Coroutines support for Room
implementation "androidx.room:room-ktx:$room_version"
```

Add the following line at the top of the file to apply the plugin.

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

Next, we prepare the three classes — Entity, DAO, and Database — separately.

Entity

We're converting the `Event` data class into an **entity** by **adding annotations**.

```
@Entity(tableName = "event")
data class Event(
    @PrimaryKey val id: Int, val title: String, val deptId: String, var saved:
Boolean
) {
    companion object {
        val data = listOf(
            Event(id = 1, title = "Career Talks", deptId = "COMS", saved = false),
            Event(id = 2, title = "Guided Tour", deptId = "COMS", saved = true),
            Event(id = 3, title = "MindDrive Demo", deptId = "COMP", saved =
false),
            Event(id = 4, title = "Project Demo", deptId = "COMP", saved = false)
        )
    }
}
```

Note that we add `@Entity` only to indicate this is an entity class and `@PrimaryKey` to show `id` is the primary key. The `tableName` specifies the SQLite table name.

DAO

Let's make a new Kotlin file called `EventViewModel.kt` and define a **Kotlin interface** called `EventDatabaseDao`.

```
import androidx.room.*

@Dao
interface EventDatabaseDao {
    @Query("SELECT * from event")
    fun getAll(): LiveData<List<Event>>

    @Query("SELECT * from event where deptId = :id")
    fun getByDeptId(id: String): LiveData<List<Event>>

    @Update
    suspend fun update(event:Event)
}
```

It's not hard to understand the overall concept. Each function (`getAll`, `getByDeptId`, etc.) queries the database. So in total there are three functions. Standard queries like insert, update, delete can use the query templates `@Insert`, `@Delete` and `@Update`. For queries that need more complex logic, we use `@Query`, which accepts an SQL statement.

At the start of each function, we add the modifier `suspend`. This forces the function to only be called in a **coroutine**. A coroutine works like a thread, where multiple coroutines can run at once. Unlike a thread, a coroutine is lightweight and multiple coroutines can run in the same thread. Typically, the UI and main logic run in the main thread, while IO-intensive operations like database or network queries run in IO coroutines.

LiveData

`LiveData` is an **observable data holder** class. Unlike a regular observable, LiveData is **lifecycle-aware**, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData **only updates observers that are in an active lifecycle state**.

`LiveData` is an **observable data holder** class in Android development that is **part of the Architecture Component Libraries**. LiveData is designed to help developers build reactive, data-driven user interfaces by **providing a way to observe changes to data in real-time**.

LiveData is used to hold data that can be observed by other parts of the application, such as the user interface. When the data changes, LiveData notifies the observers, which can then update the UI with the new data.

LiveData is particularly useful for handling asynchronous data updates, such as data from a network request or a database query. By using LiveData, developers can avoid the need for manual callbacks or complex event handling, and instead rely on a simple, reactive programming model.

Overall, LiveData is a powerful tool for building reactive, data-driven user interfaces in Android applications, and is widely used in modern Android development. To use LiveData, you need the following dependency:

```
implementation "androidx.compose.runtime:runtime-livedata:1.3.3"
```

Database

Add an abstract class `EventDatabase` in the same file:

```
@Database(entities = [Event::class], version = 1)
abstract class EventDatabase : RoomDatabase() {
    abstract fun eventDao(): EventDatabaseDao

    companion object {
        private var INSTANCE: EventDatabase? = null
        fun getInstance(context: Context): EventDatabase {
            synchronized(this) {
                var instance = INSTANCE
```

```
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        EventDatabase::class.java,
                        "event_database"
                    )
                        .createFromAsset("events.db")
                        .fallbackToDestructiveMigration()
                        .build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

We define the database class as an **abstract singleton class**. It needs **one abstract method that takes no parameters** and returns a DAO object. Depending on how we set up the data, the database can be loaded with the app, created by the user, or downloaded from a website. In this case, we load a SQLite database file named `events.db`.

Make a new `assets` directory (`/src/main/assets`) inside `app`. Download this file and name it `events.db`. Move the file into that folder.

Repository

In Android Room development, a **repository is an abstraction layer between the data source (i.e. the database) and the rest of the application**. A repository acts as a **mediator** between the application and the underlying data, providing a **clean and consistent API** for accessing and manipulating data.

Under the same file, let's develop the following **repository**. A repository class can provide access to multiple data sources. The repository is not part of the **Architecture Component Libraries** but is a **suggested best practice for separating code and architecture**.

```
class EventRepository(private val eventDatabaseDao: EventDatabaseDao) {

    val readAllData: LiveData<List<Event>> = eventDatabaseDao.getAll()

    suspend fun updateEvent(event: Event) {
        eventDatabaseDao.update(event)
    }
}
```

Keep in mind that **database queries cannot be executed on the main thread**. Therefore, the `updateEvent()` function is `suspend` and must be called in a coroutine.

ViewModel

The **ViewModel** component is designed to help developers **separate the UI-related data from the UI components**, making it easier to manage and maintain the code. By using ViewModel, developers can ensure that the **data is retained across configuration changes, such as screen rotations or device orientation changes**, without the need for complex handling by the UI components:

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

    val readAllData: LiveData<List<Event>>
    private val repository: EventRepository

    init {
        val eventDao = EventDatabase.getInstance(application).eventDao()
        repository = EventRepository(eventDao)
        readAllData = repository.readAllData
    }

    fun bookmarkEvent(event: Event) {
        viewModelScope.launch(Dispatchers.IO) {
            event.saved = true
            repository.updateEvent(event = event)
        }
    }

}

class EventViewModelFactory(
    private val application: Application
) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        @Suppress("UNCHECKED_CAST")
        if (modelClass.isAssignableFrom(EventViewModel::class.java)) {
            return EventViewModel(application) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

The `viewModelScope.launch(Dispatchers.IO)` allows a block of code to be launched in **another coroutine by the dispatchers IO** . This coroutine will be executed separately.

Access the Data

Our `EventScreen` composable is modified as follows:

```
import androidx.compose.runtime.*

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun EventScreen(deptId: String?) {

    val context = LocalContext.current
    val eventViewModel: EventViewModel = viewModel(
        factory = EventViewModelFactory(context.applicationContext as Application)
    )

    val events by eventViewModel.readAllData.observeAsState(listOf())

    LazyColumn {
        items(events.filter{it.deptId == deptId}) { event ->
            ListItem(
                headlineText = { Text(event.title) }
            )
            Divider()
        }
    }
}
```

The `readAllData` provides a `LiveData`, which is being **observed** as a **state** in this composable. `events` is a list of `Event` and its value is provided by the `LiveData`.

Bookmark an Event

When we **long-press** on an event, we want to change its `saved` status to **true** and update the database accordingly. First, we need to set up a **modifier** to **detect the tap and hold gesture** as follows:

```
ListItem(
    headlineText = { Text(event.title) },
    modifier = Modifier.pointerInput(Unit) {
        detectTapGestures(
```

```
            onLongPress = {
                eventViewModel.bookmarkEvent(event)
            }
        )
    }
)
```

The `bookmarkEvent()` function will set the `saved` value to true.
You can select `App Inspection` to see the current database contents.

Showing a Snackbar

We can set up a **snackbar** within the `ScaffoldScreen`:

```
snackbarHost = { SnackbarHost(snackbarHostState) },
```

The `SnackbarHost` requires a `SnackbarHostState` to be provided as follows:

```
val snackbarHostState = remember { SnackbarHostState() }
```

We will pass this `snackbarHost` down the screen hierarchy:

```
1 -> DeptNav(navController, snackbarHostState)
```

```
composable("event/{deptId}") { backStackEntry ->
    EventScreen(snackbarHostState, backStackEntry.arguments?.getString("deptId"))
}
```

We need to fix some constructor definitions and previews.

Finally, in the `onLongPress` lambda, add the following to display the **snackbar**:

```
coroutineScope.launch {
    snackbarHostState.showSnackbar(
        "Event has been added to itinerary."
    )
}
```

The `coroutineScope` is set up under `EventScreen` as follows:

```
val coroutineScope = rememberCoroutineScope()
```

Exercise: Itinerary

Our third tab displays our itinerary, listing all saved events. Create a new file, `ItineraryScreen.kt`, for this purpose.

To retrieve saved events, you should make changes to the data access object (DAO), repository, and view model. Finally, also implement a "Remove from itinerary" feature.

Commit and push your project to the private repository.

13/03/2023 11:58