INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: PROGRAMAÇÃO PARA DISPOSITIVOS MÓVEIS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: BANCO DE DADOS LOCAL COM ROOM

Prática 11

Obs.: Esta prática é continuação da prática 10; use controle de versões.

Parte 1: Configuração das Dependências do Room

Passo 1: No arquivo **build.gralde.kts** do projeto, adicione o *puglin*:

```
puglins {
    ...
    id("com.google.devtools.ksp") version "1.9.10-1.0.13" apply false
}
```

Passo 2: No arquivo **build.gradle.kts** do módulo app, adicione as seguintes configurações:

```
plugins {
    ...
    id("com.google.devtools.ksp") version "1.9.10-1.0.13"
}

android {
    ...

    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.3"
    }
    ...
}

dependencies {
    val room_version = "2.6.1"
    implementation("androidx.room:room-runtime:$room_version")
    implementation("androidx.room:room-ktx:$room_version")
    ksp("androidx.room:room-compiler:$room_version")

   ...
}
```

Passo 3: Mande dar o "Sync now" e faça um *rebuild* completo do projeto.

Se estiver tudo OK, faça um novo commit.

Em caso de erros, mude a versão do JDK usada pelo Gradle para Java 18:

File > Settings > Build, ... > Build Tools > Gradle > Gradle JDK.

Parte 2: Criando os componentes do Room

Passo 1: Crie a classe `db.local.LocalCity`:

```
@Entity
data class LocalCity (
    @PrimaryKey
    var name: String,
    var latitude : Double,
    var longitude : Double,
    var isMonitored : Boolean
)

fun LocalCity.toCity() = City(
    name = this.name,
    location = LatLng(this.latitude, this.longitude),
    isMonitored = this.isMonitored
)

fun City.toLocalCity() = LocalCity(
    name = this.name,
    latitude = this.location?.latitude?:0.0,
    longitude = this.location?.longitude?:0.0,
    isMonitored = this.isMonitored
)
```

A classe `LocalCity` é a entidade que será salva na tabela local. Nesse caso só teremos uma tabela contendo as cidades favoritas do usuário. A prática recomendada (pelo Google) é ter uma classe a parte para o armazenamento local, diferente da classe que modela os objetos exibidos na UI, por isso a redundância e as transformações entre si (`toCity()` e `toLocalCity()`).

Passo 2: Crie a interface `db.local.LocalCityDAO`:

```
@Dao
interface LocalCityDAO {

    @Upsert
    suspend fun upsert(city : LocalCity)

    @Delete
    suspend fun delete(city : LocalCity)

    @Query("SELECT * FROM LocalCity")
    fun getCities() : Flow<List<LocalCity>>
}
```

Essa interface oferece os métodos do *Data Access Object* (DAO) que usaremos para acessar as entidades. A biblioteca Room vai fornecer as implementações dos métodos automaticamente, por isso as anotações. Repare que as funções são do tipo *suspend*, o que será explicado a seguir. Nessa interface podemos definir consultas SQL para acessar e alterar nossos dados. (*Upsert* é uma combinação de *Insert* e *Update*.)

Passo 3: Crie a classe abstrata `db.local.LocalCityDatabase`:

```
@Database ( entities = [LocalCity::class], version = 1, exportSchema = false)
abstract class LocalCityDatabase : RoomDatabase() {
    abstract fun favCityDao() : LocalCityDAO
}
```

A implementação dessa classe é gerada pela biblioteca Room, retornando os objetos DAO que usados para acessar os dados. (Nesse caso, apelas `LocalCityDAO`).

Passo 4: Crie `db.local.LocalDB`:

```
class LocalDB (context : Context, databaseName : String) {
    private var roomDB : LocalCityDatabase = Room.databaseBuilder(
        context = context,
        klass = LocalCityDatabase::class.java,
        name = databaseName
    ).build()

    private var scope : CoroutineScope = CoroutineScope(Dispatchers.IO)

    fun insert(city: City) =  scope.launch {
        roomDB.localCityDao().upsert(city.toLocalCity())
    }

    fun update(city: City) = scope.launch {
        roomDB.localCityDao().upsert(city.toLocalCity())
    }

    fun delete(city: City) =  scope.launch {
        roomDB.localCityDao().delete(city.toLocalCity())
    }

    fun getCities(doSomething : (City) -> Unit) = scope.launch {
        roomDB.localCityDao().getCities().collect { list ->
            val mappedList = list.map { it.toCity() }
            mappedList.forEach { doSomething(it) }
        }
    }
}
```

Essa classe é a responsável em nossa arquitetura por disparar as mudanças no banco de dados local. Para isso, pegamos um objeto do tipo DAO e chamamos o método adequado. Como os métodos do DAO são anotados com *suspend* (isto é, podem ficar bloqueados por algum tempo), é preciso chamá-los de dentro de uma corrotina, que são um mecanismo de paralelismo leve oferecido pela linguagem Kotlin (mais leves comparados a *Threads* de Java, por exemplo). Fazemos isso no código acima usando `scope.launch { ... }`, porém existem outras formas de lançar corrotinas. O objeto `scope` agrupa corrotinas relacionadas, o que permite cancelá-las facilmente (chamando `scope.cancel()` ou quando o escopo é destruído).

A função `getCities()` processa as cidades carregadas do banco via um lambda. O lambda é aplicado ao `Flow` retornado do DAO, o qual modela um *stream* de dados atualizado assincronamente. Usamos a função `map` para transformar a lista de cidades do tipo `LocalCity` em uma lista de `City`. Essa função será usada mais a frente para carregar as cidades inicialmente.

Passo 5: Compile o código para ver se não há erros. Faça um novo *commit*.

Parte 3: Conectando o Banco de Dados a nossa Arquitetura

Passo 1: Modifique `repo.Repository`, adicionando o banco de dados local.

```
class Repository (context : Context, private var listener : Listener):
 FBDatabase.Listener {
    private var fbDb = FBDatabase (this)
    private var weatherService = WeatherService()
    private var localDB: LocalDB = LocalDB(context, databaseName = "local.db")

    ...

    init {
        localDB.getCities {
            fbDb.add(it)
        }
    }

    fun addCity(name: String) {
        weatherService.getLocation(name) { lat, lng ->
            val city = City(name = name, location = LatLng(lat?:0.0, lng?:0.0))
            localDB.insert(city)
            fbDb.add(city)

        }
    }

    fun addCity(lat: Double, lng: Double) {
        weatherService.getName(lat, lng) { name ->
            val  city = City( name = name?:"NOT_FOUND", location = LatLng(lat, lng))
            localDB.insert(city)
            fbDb.add(city)
        }
    }

    fun remove(city: City) {
        localDB.delete(city)
        fbDb.remove(city)
    }

    fun update(city: City) {
        localDB.update(city)
        fbDb.update(city)
    }
    ...
}
```

O bloco `init {...}` carrega as cidades do banco local e salva no Firebase na criação do Repository. Isso é normalmente desnecessário, já que o Firebase tem um *cache* local, mas aqui está usamos o banco local como *backup* adicional. As demais alterações fazem com que a cidade seja atualizada/removida tanto no banco de dados local quanto na nuvem.

Passo 2: Altere `MainActivity` e `WeatherApp` de forma a passar um contexto (a própria *activity* ou *application*) como parâmetro para o repositório.

Passo 3: Compile e teste a aplicação.

Atenção: Recomenda-se apagar as cidades do Firebase antes de testar.

Não deve haver mudanças siginificativas no funcionamento do App, mas é possível ver se os dados estão sendo salvos no banco de dados local usando o *App Inspection* do Android Studio.

Faça um novo *commit* se estiver tudo certo.