Annexe : Persistance de données

Persistance des données avec Room

Pourquoi persister des données

- Besoin de stocker des données persistantes qui survivent à une fermeture de l'app
- Structures des données très variés
  - Paramètres d'une application (thème sombre ou clair, réglages de notifications, règles d'affichage)
  - Mails
  - Articles de journaux
  - Photos
  - Caractérisques des personnages d'un jeu vidéo

SQLite

- Bibliothèque qui implémente un moteur de base de données SQL cross-platform
- Léger et rapide
- Intégré à Android
- Requêtes en SQL pour interagir avec
- Pour les données relationnels

Room

- Bibliothèque Android faisaint partie de Jetpack
- Couche d'abstraction au dessus de SQLite
- Simplifie la configuration et les opération avec la base de donnée
- Vérifie les requêtes SQL

Composants Room

- Les entities, représentent les tables dans la base de données
- Les DAO (Data Access Object) permet de récupérer, mettre à jour, insérer et supprimer des données dans la base
- La classe Database, donne accès aux DAO et gère la base de données

Entity

- Une classe d'entité est une table
- Chaque instance de cette classe est une ligne dans la table
- Les propriétés de la classe sont les colonnes
- Défini avec l'annotation **@Entity**
- **@PrimaryKey** pour définir la clé primaire de la table

```
@Entity(tableName = "items")
data class Item(
    @PrimaryKey
    val id: Int,
    ...
)
```

DAO

- Interface qui abstrait les opérations sur la base de données
- Annotation **@Dao**
- **@Insert** pour insérer, **@Delete** pour supprimer et **@Update** pour mettre à jour une donnée
- **@Query** pour une requête SQL personnalisé ou opérations plus complexes
- Peut retourner un Flow pour être notifié quand les données changent

```kotlin
@Dao
interface ItemDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(item: Item)

    @Update
    suspend fun update(item: Item)

    @Delete
    suspend fun delete(item: Item)

    @Query("SELECT * from items WHERE id = :id")
    fun getItem(id: Int): Flow<Item>

    @Query("SELECT * from items ORDER BY name ASC")
    fun getAllItems(): Flow<List<Item>>
}
```

RoomDatabase

- Classe abstraite qui représente la base de données
- Définit les Entités et DAO utilisés par l'app
- Les rend accessibles par l'app
- Singleton pour accéder à la base et les DAO
- Annotation @Database

RoomDatabase

```
@Database(entities = [Item::class], version = 1, exportSchema = false)
abstract class InventoryDatabase : RoomDatabase() {

    abstract fun itemDao(): ItemDao

    companion object {
        @Volatile
        private var Instance: InventoryDatabase? = null

        fun getDatabase(context: Context): InventoryDatabase {
            // if the Instance is not null, return it, otherwise create a new database instance.
            return Instance ?: synchronized(this) {
                Room.databaseBuilder(context, InventoryDatabase::class.java, "item_database")
                    .build()
                    .also { Instance = it }
            }
        }
    }
}
```

Utilisation dans Data source

```
class LocalItemsDataSource(private val itemDao: ItemDao) {
    override fun getAllItemsStream(): Flow<List<Item>> = itemDao.getAllItems()

    override fun getItemStream(id: Int): Flow<Item?> = itemDao.getItem(id)

    override suspend fun insertItem(item: Item) = itemDao.insert(item)

    override suspend fun deleteItem(item: Item) = itemDao.delete(item)

    override suspend fun updateItem(item: Item) = itemDao.update(item)
}
```

Utilisation dans Data source

```
override val localItemsDataSource: LocalItemsDataSource by lazy {
    LocalItemsDataSource(InventoryDatabase.getDatabase(context).itemDao())
}

override val itemsRepository: ItemsRepository by lazy {
    ItemsRepository(localItemsDataSource)
}
```

Ateliers supplémentaires sur Room

https://developer.android.com/codelabs/basic-android-kotlin-compose-persisting-data-room?hl=fr

https://developer.android.com/codelabs/basic-android-kotlin-compose-update-data-room?hl=fr

DataStore

- Bibliothèque JetPack
- Pour les donnés plus simples, non relationnelles
- String, Boolean, Integer
- 2 variantes : Preferences et Protobuf
- Protobuf permet de stocker des classes customs
- Preferences utilise des clés/valeurs est idéal pour stocker les paramètres utilisateurs

DataStore

```
class UserPreferencesRepository(
    private val dataStore: DataStore<Preferences>
) {
    private companion object {
        val IS_LINEAR_LAYOUT = booleanPreferencesKey("is_linear_layout")
    }

    // Read from DataStore
    val isLinearLayout: Flow<Boolean> = dataStore.data
        .map { preferences ->
            preferences[IS_LINEAR_LAYOUT] ?: true
        }

    // Write to DataStore
    suspend fun saveLayoutPreference(isLinearLayout: Boolean) {
        dataStore.edit { preferences ->
            preferences[IS_LINEAR_LAYOUT] = isLinearLayout
        }
    }
}
```

Atelier supplémentaires DataStore

https://developer.android.com/codelabs/basic-android-kotlin-compose-datastore

NAVHOST ?? Pour l'atelier