Développement Android avec Kotlin

Cours - 07 - Bases de données

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com

Introduction aux bases de données avec Android

Bases de données SQLite

Bibliothèque de persistance Room

Qu'est-ce qu'une base de données ?

Une **base de données** est une collection de **données structurées**, conçue pour être :

- Facilement **consultée**
- Rapidement **recherchée**
- Efficacement **organisée**

user

id	name	age	city
1	Alice	25	New York
2	Bob	30	San Francisco
3	Charlie	22	Los Angeles

Structure :

- **Tables** : Conteneurs principaux des données
- **Lignes** : Représentent chaque **enregistrement**
- **Colonnes** : Définissent les **attributs** ou **champs**

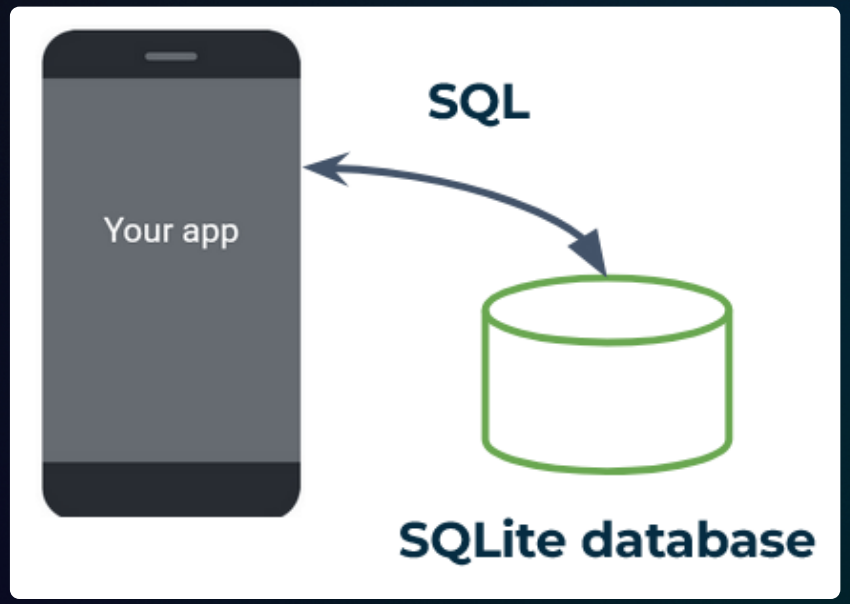
Communiquer avec une base de données

Langage utilisé : SQL (*Structured Query Language*).

- SQL permet de **lire** et de **manipuler** les données d'une base de données relationnelle.

Exemples d'opérations :

- Récupérer des lignes spécifiques (`SELECT`).
- Ajouter de nouvelles données (`INSERT`).
- Mettre à jour des données existantes (`UPDATE`).
- Supprimer des données (`DELETE`).



La commande SQL SELECT

- Utilisée pour **lire et récupérer des données** d'une ou plusieurs tables.
- Structure de base : **`SELECT colonne(s) FROM table [clauses];`**

```sql
SELECT * FROM User; -- Récupère toutes les colonnes de la table User

SELECT name, city FROM User; -- Récupère uniquement les colonnes 'name' et 'city'

SELECT * FROM User WHERE city = 'Paris';  -- Récupère toutes les colonnes des utilisateurs dont la ville est Paris

SELECT * FROM User WHERE age > 25;  -- Récupère tous les utilisateurs ayant un âge supérieur à 25

SELECT * FROM User ORDER BY age DESC;  -- Récupère tous les utilisateurs, triés par âge de manière décroissante

SELECT * FROM User LIMIT 5;  -- Récupère seulement les 5 premiers utilisateurs
```

La commande SQL INSERT

- Utilisée pour ajouter de nouvelles données dans une table.
- Structure de base : **`INSERT INTO table (colonne1, ...) VALUES (value1, ...);`**

```sql
INSERT INTO User (name, age, city) VALUES ('Alice', 25, 'Paris');  -- Ajoute un utilisateur nommé Alice

INSERT INTO User (name, age, city)
VALUES ('Bob', 30, 'Lyon'), ('Charlie', 22, 'Marseille');  -- Ajoute deux utilisateurs, Bob et Charlie

INSERT INTO User VALUES (NULL, 'David', 40, 'Lille');  -- NULL pour l'id la colonne est auto-incrémentée
```

La commande SQL UPDATE

- Utilisée pour modifier les données existantes dans une table.
- Structure de base : **`UPDATE table SET colonne = valeur WHERE condition`**

```sql
UPDATE User SET age = 26 WHERE name = 'Alice';  -- Met à jour l'âge de l'utilisateur Alice à 26 ans

UPDATE User SET age = 27, city = 'Lyon' WHERE name = 'Bob';  -- Met à jour l'âge et la ville de Bob

UPDATE User SET city = 'Paris';  -- Met à jour la ville de tous les utilisateurs en 'Paris'
```

La commande SQL DELETE

- Utilisée pour supprimer des données dans une ou plusieurs tables.
- Structure de base : **`DELETE FROM table WHERE condition`**

```sql
DELETE FROM User WHERE name = 'Alice';  -- Supprime l'utilisateur nommé Alice

DELETE FROM User WHERE age < 20;  -- Supprime tous les utilisateurs ayant moins de 20 ans

DELETE FROM User;  -- Supprime tous les utilisateurs de la table
```

Et avec Android ?

Avant l'arrivée de Room, on utilisait `rawQuery()` et un `Cursor` pour exécuter des requêtes SQL sur SQLite.

```
val cursor = db.rawQuery("SELECT * FROM User WHERE age > ?", arrayOf("30"))

while (cursor.moveToNext()) {
    val name = cursor.getString(cursor.getColumnIndexOrThrow("name"))
    // Traitement des données récupérées
}

cursor.close()
```

"Une époque où chaque requête semblait venir tout droit de l'âge de pierre du développement mobile."

Un développeur

Bibliothèque de persistance Room

Room est une bibliothèque d'abstraction sur SQLite qui simplifie l'accès à la base de données tout en offrant la puissance de SQLite. Elle présente plusieurs avantages :

- **Vérification des requêtes SQL au moment de la compilation**
- **Annotations pratiques** qui réduisent le code récurrent et minimisent les erreurs.
- **Migration simplifiée** pour gérer les mises à jour des bases de données.

Configuration de Room

Pour utiliser Room dans votre application, ajoutez les dépendances suivantes dans votre fichier `build.gradle` :

```
dependencies {
    val room_version = "2.6.1"

    // Dépendance principale pour Room
    implementation("androidx.room:room-runtime:$room_version")

    // KSP (Kotlin Symbol Processing) pour la génération du code Room
    ksp("androidx.room:room-compiler:$room_version")

    // Optionnel : Extensions Kotlin et support des coroutines pour Room
    implementation("androidx.room:room-ktx:$room_version")
}
```

Note : L'option `room-ktx` permet d'utiliser des coroutines et des extensions Kotlin pour simplifier l'intégration avec Room.

Composants de Room

Room repose sur trois composants principaux :

- **La classe de base de données** :

Contient la base de données et sert de point d'accès principal pour la connexion aux données persistantes de votre application.

- **Les entités de données** :

Représentent les tables de la base de données de votre application. Chaque entité est associée à une table dans la base de données.

- **Les objets d'accès aux données (DAO)** :

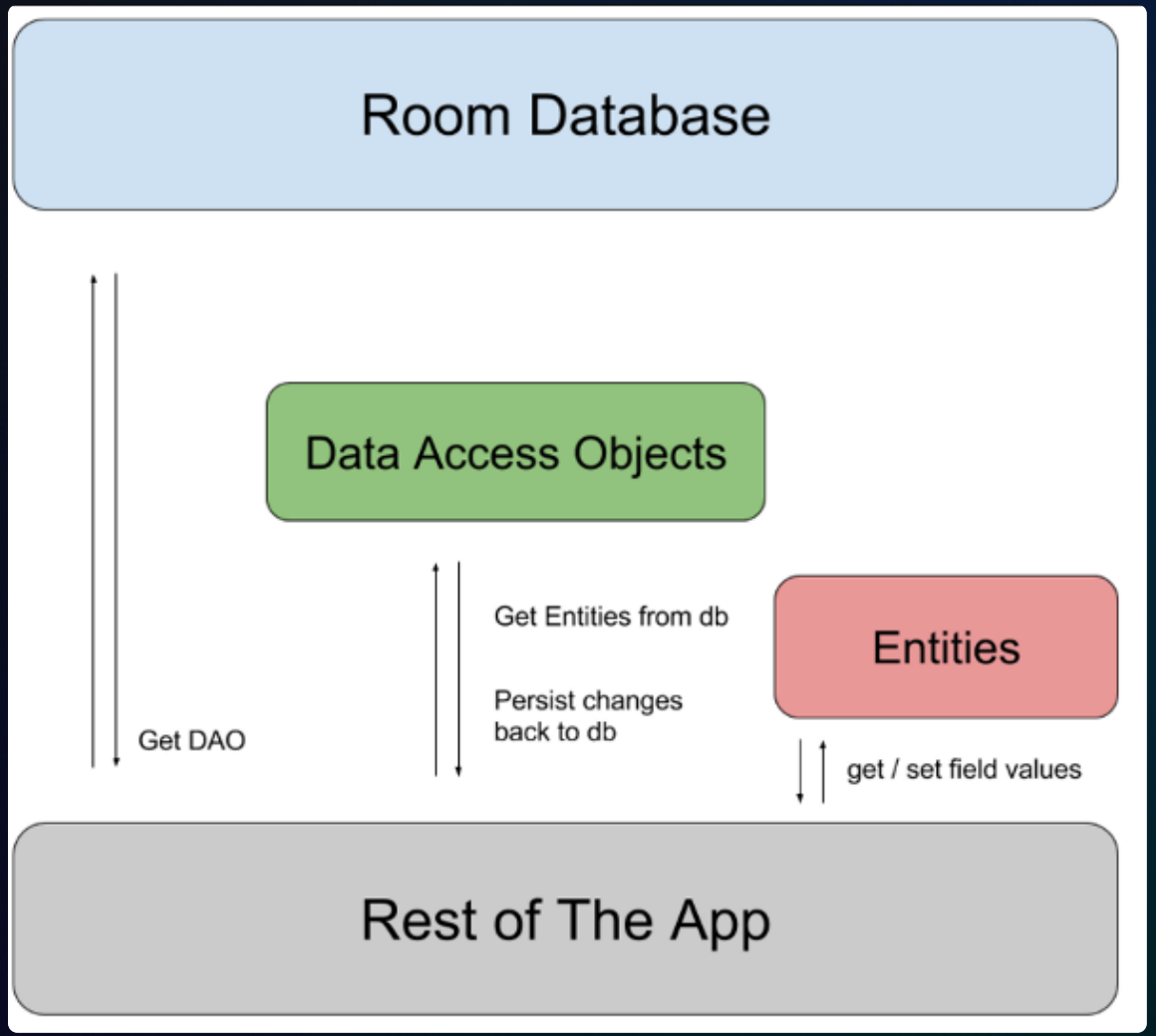
Fournissent des méthodes permettant de gérer les données (interroger, insérer, mettre à jour et supprimer) dans la base de données.

La relation entre les composants de Room

La classe de base de données fournit à votre application des instances des DAO associés à cette base de données.

L'application peut ensuite utiliser ces DAO pour :

- Récupérer des données sous forme d'objets d'entité.
- Mettre à jour des lignes dans les tables correspondantes.
- Insérer de nouvelles lignes dans les tables.



Les entités de données

Les entités de données sont des classes Kotlin annotées avec `@Entity`. Elles représentent les tables de la base de données, où chaque propriété devient une colonne.

```kotlin
@Entity(tableName = "user")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    @ColumnInfo(name = "name") val name: String,
    @ColumnInfo(name = "age") val age: Int,
    @ColumnInfo(name = "city") val city: String,

    @Ignore val isAdult: Boolean = age >= 18  // Ce champ ne sera pas stocké dans la base de données
)
```

Objets d'accès aux données (DAO)

Les DAO permettent d'interagir avec la base de données via des méthodes abstraites. Room génère automatiquement l'implémentation des DAO au moment de la compilation, facilitant l'accès aux données.

```
1  @Dao
2  interface UserDao {
3
4      @Query("SELECT * FROM user")
5      fun getAllUsers(): List<User>  // Récupérer tous les utilisateurs
6
7      @Insert
8      fun insertUser(user: User) // Insérer un nouvel utilisateur
9
10     @Update
11     fun updateUser(user: User) // Mettre à jour un utilisateur
12
13     @Delete
14     fun deleteUser(user: User) // Supprimer un utilisateur
15 }
```

Exemples de requêtes dans un DAO

Room permet d'utiliser des requêtes SQL plus complexes dans les DAO pour gérer des interactions avancées avec la base de données.

```
@Dao
interface UserDao {

    @Query("SELECT * FROM user WHERE age > :minAge")
    fun loadAllUsersOlderThan(minAge: Int): Array<User>

    @Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
    fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int): Array<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :search " +
       "OR last_name LIKE :search")
    fun findUserWithName(search: String): List<User>

}
```

💡 **Note** : Room vérifie les requêtes SQL lors de la compilation pour éviter les erreurs d'exécution liées à des requêtes incorrectes.

Base de données

Le code suivant définit une classe `AppDatabase` destinée à contenir la base de données. Elle configure la base de données et sert de point d'accès principal aux données persistantes.

- Elle doit être annotée avec `@Database`, listant toutes les entités de données associées à la base de données.
- Elle doit être abstraite et étendre `RoomDatabase`.
- Elle doit inclure une méthode abstraite pour chaque DAO associé à la base de données, renvoyant une instance du DAO.

```
1 @Database(entities = [User::class], version = 1)
2 abstract class AppDatabase : RoomDatabase() {
3
4     // Définir les DAO ici
5     abstract fun userDao(): UserDao
6 }
```

Instanciation de la base de données (Singleton)

Pour éviter plusieurs instances de la base de données et assurer une gestion centralisée, il est recommandé d'instancier `AppDatabase` via un singleton.

```
object DatabaseProvider {

    @Volatile
    private var INSTANCE: AppDatabase? = null

    fun getDatabase(context: Context): AppDatabase {
        return INSTANCE ?: synchronized(this) {
            val instance = Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "app_database"
            ).build()
            INSTANCE = instance
            instance
        }
    }
}
```

Relations entre objets dans Room

SQLite étant une base de données relationnelle, vous pouvez définir des relations entre les entités.

Types de relations supportées par Room :

- **Un à un :** représente une relation dans laquelle une entité est associée à une autre entité.
- **Un à plusieurs :** représente une relation dans laquelle une entité unique peut être associée à plusieurs entités d'un autre type.
- **Plusieurs à plusieurs :** représente une relation dans laquelle plusieurs entités d'un type peuvent être associées à plusieurs entités d'un autre type. Cela nécessite généralement une table de jointure.
- **Imbriquées :** Une entité contient une autre, via `@Embedded`.

Relation un à un

```kotlin
@Entity
data class User(
    @PrimaryKey val id: Long,
    val name: String,
    val age: Int
)
```

```kotlin
@Entity(tableName = "profile_table")
data class Profile(
    @PrimaryKey val userId: Int,
    val bio: String,
    val avatarUrl: String
)
```

```kotlin
data class UserWithProfile(
    @Embedded val user: User,

    @Relation(
        parentColumn = "id",
        entityColumn = "userId"
    )
    val profile: Profile
)
```

```kotlin
@Dao
interface UserDao {

    @Transaction
    @Query("SELECT * FROM user WHERE id = :userId")
    suspend fun getUserWithProfile(userId: Int): UserWithProfile
}
```

Relation un à plusieurs

```kotlin
@Entity
data class User(
    @PrimaryKey val id: Long,
    val name: String,
    val age: Int
)
```

```kotlin
@Entity
data class Post(
    @PrimaryKey val id: Int,
    val userId: Int,  // Référence à l'utilisateur
    val content: String
)
```

```kotlin
data class UserWithPosts(
    @Embedded val user: User,
    @Relation(
        parentColumn = "id",
        entityColumn = "userId"
    )
    val posts: List<Post>  // Liste des posts associés à l'utilisateur
)
```

```kotlin
@Dao
interface UserDao {

    @Transaction
    @Query("SELECT * FROM user WHERE id = :userId")
    suspend fun getUserWithPosts(userId: Int): UserWithPosts
}
```

Écrire des requêtes DAO asynchrones

Pour éviter de bloquer l'interface utilisateur, Room empêche l'accès direct à la base de données sur le thread principal. Cela signifie que vous devez rendre vos requêtes DAO asynchrones.

Solutions disponibles :

- **Coroutines Kotlin** *(Abordées en détail dans un cours séparé)*

  Permet une gestion fluide et efficace des requêtes sans bloquer l'interface utilisateur. Utilisation avec `suspend` et `async`.

- **ExecutorService**

  Permet d'exécuter des tâches asynchrones en utilisant un pool de threads. Facile à intégrer dans des projets existants.

Exemple : Utilisation d'ExecutorService avec Room

```
// Manager class
val executor = Executors.newSingleThreadExecutor()


fun getUserById(userId: Int, onSuccess: (User?) -> Unit) {
    executor.execute {
        val user = userDao.getUserById(userId)
        onSuccess(user)
    }
}
```

```
// Activity class

userManager.getUserById(1) { user ->
    runOnUiThread { // Important
        // Update UI with data
    }
}
```

Conclusion

Les bases de données sont un élément clé de nombreuses applications Android, permettant de stocker et de gérer des données de manière persistante.

- **SQLite** : Une base de données relationnelle légère intégrée à Android, idéale pour des besoins simples de stockage local.
- **Room** : Une bibliothèque de persistance moderne qui simplifie l'interaction avec SQLite, avec des avantages comme la vérification des requêtes SQL à la compilation, la réduction du code répétitif, et une gestion améliorée des migrations.
- **DAO et Entités** : Grâce aux annotations, Room facilite la gestion des entités et l'accès aux données via les DAO, tout en respectant les bonnes pratiques de séparation des préoccupations.