The Android Prep Lab (Kotlin)

Part - 7 of 10
Android Data Persistence

What is data persistence in Android?

Data Persistence in Android refers to saving data in a way that it can be accessed and retrieved even after the app is closed or the device is restarted. It is crucial for retaining user data, settings, or any state information between app launches. There are several ways to persist data in Android:

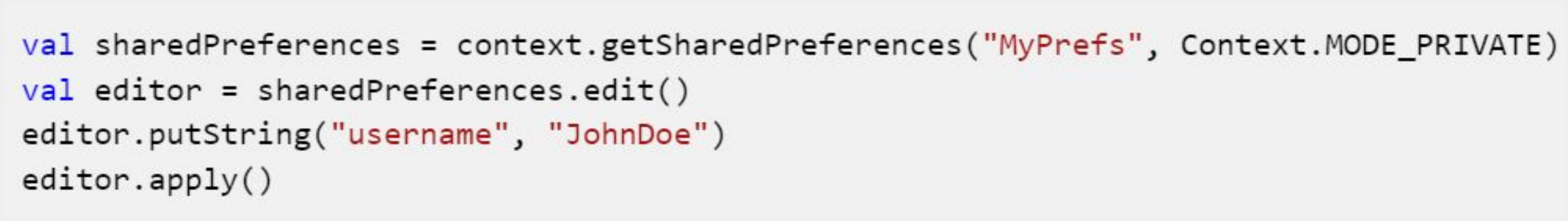
- **SharedPreferences**: For simple key-value pair storage (e.g., user preferences or app settings).
- **SQLite Database**: For structured data storage (e.g., lists, records).
- **Room Database**: A higher-level abstraction over SQLite for easier and more efficient database access.
- **Files**: For storing raw data or large objects like images or documents.
- **DataStore**: A modern, asynchronous data storage solution that replaces SharedPreferences.

What is SharedPreferences and when would you use it?

SharedPreferences is used for storing small amounts of primitive data (key-value pairs) in XML format. It's ideal for storing user preferences or settings like theme, login details, or other app configurations.

- **Use Cases**:
  - Storing user preferences (e.g., theme settings, language choice).
  - Storing authentication tokens or flags (e.g., if the user has logged in previously).

Example:



```
val sharedPreferences = context.getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)
val editor = sharedPreferences.edit()
editor.putString("username", "JohnDoe")
editor.apply()
```

What is SQLite in Android, and how does it work?

SQLite is a lightweight, relational database that is embedded within the Android device. It stores data in tables and supports SQL queries for data retrieval, insertion, update, and deletion.

- **Usage**: Suitable for storing structured data like user records, product catalogs, or transaction history.
- **Advantages**: It's fast, requires no server-side setup, and is a good option for apps that need to store large amounts of data with complex relationships.

Example:

```
val dbHelper = MyDatabaseHelper(context)
val db = dbHelper.writableDatabase

val values = ContentValues().apply {
    put("column_name", "value")
}
val newRowId = db.insert("table_name", null, values)
```

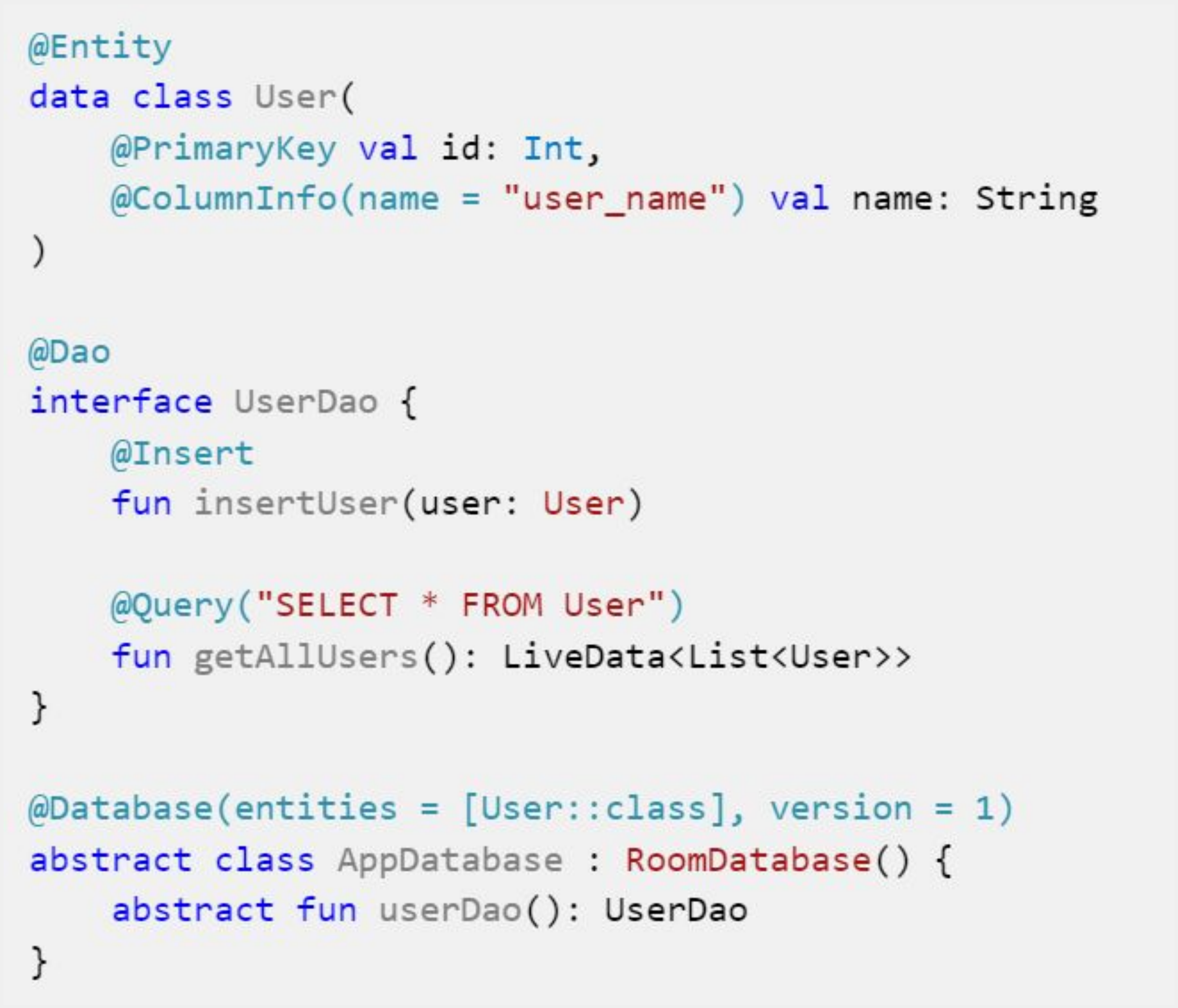
What is the difference between SQLite and Room Database?

Room Database is an abstraction layer over SQLite that simplifies database management and provides additional benefits:

- **Room** provides **annotations** to define database entities, making it easier to define the database schema compared to SQLite's raw SQL.
- **Room** supports **LiveData**, **Flow**, and **Coroutines**, making it more lifecycle-aware and suitable for modern Android development.
- **Room** offers compile-time validation of SQL queries and relationships between tables, which reduces errors and the need for boilerplate code.

Example of Room Database setup:

Example of Room Database setup:



```kotlin
@Entity
data class User(
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "user_name") val name: String
)

@Dao
interface UserDao {
    @Insert
    fun insertUser(user: User)

    @Query("SELECT * FROM User")
    fun getAllUsers(): LiveData<List<User>>
}

@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

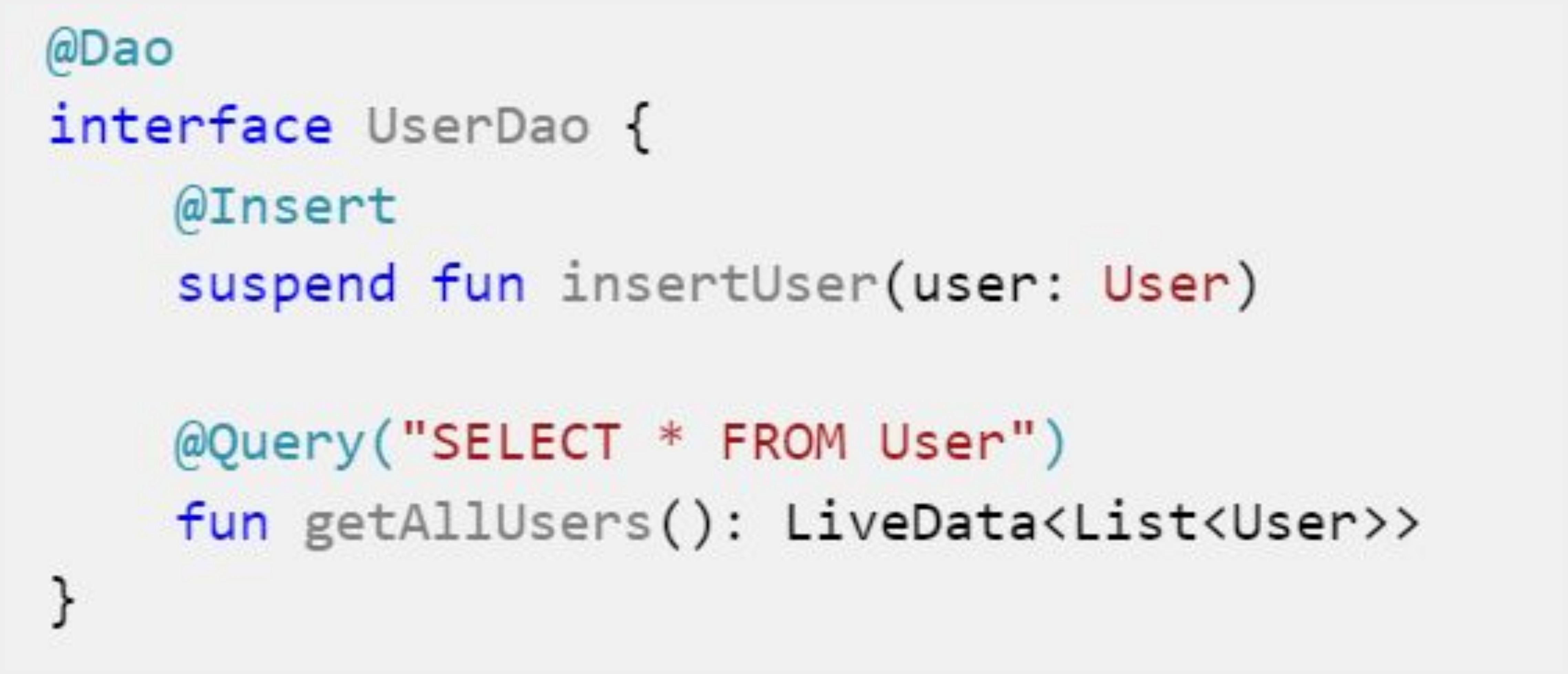
How do you perform database operations asynchronously in Android?

Database operations, especially long-running ones like reading and writing to the database, should be done asynchronously to avoid blocking the UI thread. This can be achieved using:

- **Coroutines**: Kotlin's coroutine system can be used with **Room** to run database queries asynchronously.
- **AsyncTask** (deprecated): Used to run database queries on background threads.
- **Executors**: Used to manage threads and execute tasks in the background.

Example with Room and Coroutines:

Example with Room and Coroutines:



```kotlin
@Dao
interface UserDao {
    @Insert
    suspend fun insertUser(user: User)

    @Query("SELECT * FROM User")
    fun getAllUsers(): LiveData<List<User>>
}
```

6

What is the use of @Entity, @Dao, and @Database annotations in Room?

- **@Entity**: Defines a table in the database. Each entity corresponds to a table and each field to a column.
- **@Dao (Data Access Object)**: Contains methods for interacting with the database (e.g., inserting, querying, updating).
- **@Database**: Defines the Room database and includes the entities and DAOs.

Example:

Example:

```kotlin
@Entity(tableName = "users")
data class User(
    @PrimaryKey val id: Int,
    val name: String
)

@Dao
interface UserDao {
    @Insert
    suspend fun insert(user: User)

    @Query("SELECT * FROM users")
    suspend fun getAllUsers(): List<User>
}

@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

What is DataStore, and how is it different from SharedPreferences?

DataStore is a modern data persistence library that provides a better, more scalable alternative to **SharedPreferences**. It offers two types of storage:

- **Preferences DataStore**: Similar to SharedPreferences, stores key-value pairs.
- **Proto DataStore**: Allows you to store typed data using protocol buffers, which is more flexible and safer for complex data models.

Advantages of DataStore over SharedPreferences:

- **Asynchronous**: DataStore uses coroutines to handle data operations asynchronously.
- **Strongly Typed**: Proto DataStore offers type safety, reducing the risk of runtime errors.
- **Data Validation**: Proto DataStore allows validation using schemas, preventing data corruption.

Example with Preferences DataStore:

Example with Preferences DataStore:

```kotlin
val dataStore: DataStore<Preferences> = context.createDataStore(name = "settings")

val key = stringPreferencesKey("user_name")

// Writing data
dataStore.edit { preferences ->
    preferences[key] = "JohnDoe"
}

// Reading data
val userNameFlow: Flow<String?> = dataStore.data
    .map { preferences -> preferences[key] }
```

8

How do you handle data migrations in Room Database?

Room provides an easy way to handle database migrations when the schema changes:

- **@Migration**: This annotation helps define changes between different versions of the database.
- **fallbackToDestructiveMigration()**: When a migration path isn't defined, this method allows Room to destroy and recreate the database.

Example of Room migration:

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE user ADD COLUMN email TEXT")
    }
}

val db = Room.databaseBuilder(context, AppDatabase::class.java, "app_database")
    .addMigrations(MIGRATION_1_2)
    .build()
```

What is the importance of using LiveData in Room?

- **LiveData** is a lifecycle-aware data holder that is used to automatically update the UI when data changes. It works well with **Room** to ensure that the UI stays updated with the latest data from the database, without requiring manual updates or callbacks.
- **Example**:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAllUsers(): LiveData<List<User>>
}
```

In your Activity or Fragment:

```
userDao.getAllUsers().observe(viewLifecycleOwner, Observer { users ->
    // Update the UI with the list of users
})
```

What are the best practices for managing data persistence in Android?

- **Use Room for complex data**: For structured and relational data, prefer **Room** over SQLite directly for better manageability and ease of use.
- **Avoid blocking the main thread**: Perform all database operations in the background (e.g., using Kotlin Coroutines) to prevent UI freezes.
- **Use LiveData for reactive UIs**: Leverage **LiveData** to update the UI automatically when the data changes.
- **Data Migrations**: Plan and handle database migrations carefully to avoid data loss when the schema changes.
- **Minimize SharedPreferences usage**: Use **SharedPreferences** for simple key-value pairs, and use **Room** or **DataStore** for more complex data needs.

https://www.linkedin.com/in/pavankreddy92/

Follow for More