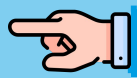
Acceso a datos en Android

Actualizar la UI con datos

{desafío} latam_

Implementar capa de acceso a datos en un aplicativo móvil utilizando la librería ROOM para otorgar persistencia de estados resolviendo el problema planteado

- Unidad 1:
  Acceso a datos en Android
- Unidad 2:
  Consumo de API REST
- Unidad 3:
  Testing
- Unidad 4:
  Distribución del aplicativo Android



{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Implementa capa de acceso a datos en un aplicativo Android utilizando la librería ROOM como capa de abstracción para resolver un problema*

Ya sabemos como guardar, leer y trabajar con datos en Room, pero ¿Cómo hacemos para mostrar esos datos al usuario?

{desafío} latam_

/* Diferencia LiveData y StateFlow */

Diferencia LiveData y StateFlow

Observabilidad	Seguridad de subprocesos	Inmutabilidad	Historial	Contrapresión
Tanto LiveData como StateFlow pueden ser observados por otras partes de la aplicación, como las vistas, para obtener actualizaciones cuando cambian los datos. Sin embargo, StateFlow tiene soporte de corrutinas incorporado, lo que permite que se use de una manera más fluida y eficiente con las API de Flow y Channels.	LiveData está diseñado para funcionar con el ciclo de vida de Android y maneja automáticamente los problemas de subprocesos y ciclos de vida. StateFlow, por otro lado, requiere que el desarrollador maneje los subprocesos y el contexto manualmente.	StateFlow es un contenedor de datos inmutable, lo que significa que una vez que se configuran los datos, no se pueden cambiar. LiveData, por otro lado, es mutable, por lo que los datos se pueden cambiar en cualquier momento.	StateFlow, de forma predeterminada, mantiene un historial de los cambios de datos, para que pueda acceder a los valores anteriores. LiveData, por otro lado, no proporciona esta característica, pero puede implementarla usando la clase MediatorLiveData.	StateFlow tiene soporte incorporado para el manejo de la contrapresión, lo que le permite manejar situaciones en las que los datos se producen más rápido de lo que se pueden consumir. LiveData no tiene esta característica, pero puede usar la clase MediatorLiveData para manejar la contrapresión.

Demostración: Integrar LiveData y Room

{desafío} latam_

Integrar LiveData y Room

Podemos integrar LiveData y Room si retornamos el tipo de LiveData para sus consultas en la DAO. Room convertirá automáticamente los resultados de la consulta en un objeto LiveData que otras partes de la aplicación pueden observar.

Aquí hay un ejemplo de cómo integrar LiveData y Room:

Define las entidades de su base de datos y la interfaz DAO

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

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

Crear una instancia de base de datos de ROOM

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null
        fun getDatabase(context: Context): AppDatabase {
            val tempInstance = INSTANCE
            if (tempInstance != null) {
                return tempInstance
            }
            synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "user_database"
                ).build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

Observa LiveData en tu Activity o Fragment

```
class MainActivity : AppCompatActivity() {
    private lateinit var userDao: UserDao
    private val usersObserver = Observer<List<User>> { users ->
        // Actualiza UI con nuevos datos
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        userDao = AppDatabase.getDatabase(this).userDao()
        userDao.getAllUsers().observe(this, usersObserver)
    }
}
```

- En este ejemplo, el método **`getAllUsers`** en **`UserDao`** devuelve un **`LiveData<List<User>>`**, el cual puede ser observado por **`usersObserver`** en **`MainActivity`**. Room ejecutará automáticamente la consulta y actualizará el objeto LiveData cuando cambien los datos.

- Es importante tener en cuenta que LiveData conoce el ciclo de vida de los componentes de la aplicación (como actividades y fragmentos) y automáticamente detendrá y reanudará la observación de acuerdo con el ciclo de vida. Esto significa que no causará pérdidas de memoria y también será eficiente ya que no consumirá recursos innecesarios.

Demostración: Integrar LiveData y Room usando MVVM

{desafío} latam_

Integrar StateFlow y Room usando MVVM

Comparando con la demostración anterior, la única diferencia es que esta vez agregaremos una clase ViewModel y reemplazamos LiveData con StateFlow, si bien el código anterior funciona, no se recomienda crear una instancia de la base de datos en un Activity o Fragment, eso debería hacerse en un ViewModel o idealmente en una clase repository

Integrar StateFlow y Room usando MVVM

Para integrar StateFlow, Room y MVVM, puede seguir estos pasos generales:

1. **Define las entidades y las DAO en Room.** Esto incluye crear su clase de base de datos, anotar sus entidades con @Entity y definir sus DAO con @Dao y las anotaciones de Room apropiadas.

2. **Crea una clase de Repositorio que manejará todas las operaciones de la base de datos.** La clase Repository usará los DAO para obtener y actualizar los datos de la base de datos. También puede usar StateFlow para emitir los datos actualizados al modelo de vista.

3. **Crea una clase ViewModel que manejará la lógica de negocio y la comunicación con la clase Repository.** La clase ViewModel tendrá una variable StateFlow que la vista puede observar para recibir actualizaciones.

4. **En la Vista (Actividad o Fragmento), observa la o las variables StateFlow del ViewModel y actualiza la interfaz de usuario en consecuencia.**

Integrar StateFlow y Room usando MVVM

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

@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getUsers(): Flow<List<User>>

    @Insert
    suspend fun insertUsers(users:
List<User>)
}
```

```
class UserRepository(private val userDao:
UserDao) {
    val users =
userDao.getUsers().asStateFlow()

    suspend fun insertUsers(users:
List<User>) {
        userDao.insertUsers(users)
    }
}

class UserViewModel(private val
userRepository: UserRepository) : ViewModel()
{
    val users = userRepository.users
}
```

Integrar StateFlow y Room usando MVVM

Puedes utilizar la arquitectura MVVM para separar la interfaz de usuario, la lógica de negocio y las capas de acceso a datos de tu aplicación.

StateFlow se puede usar para emitir actualizaciones desde el Repositorio al ViewModel y la vista puede observar los cambios y actualizar la interfaz de usuario en consecuencia. Room se puede utilizar para manejar todas las operaciones de la base de datos.

/* Algunas recomendaciones */

Al momento de usar StateFlow y Room…

- Usa Flow en lugar de LiveData: Room proporciona una versión LiveData de los métodos DAO, pero Flow es menos propenso a errores y brinda más flexibilidad.

- Use `asStateFlow()` para convertir Flow a StateFlow: StateFlow brinda la capacidad de mantener un estado y emitir solo el valor más reciente. Esto puede ser útil en situaciones en las que necesites realizar un seguimiento del estado actual de sus datos y solo desea emitir actualizaciones cuando cambia el estado.

Al momento de usar StateFlow y Room…

- Ten cuidado con las fugas de memoria (memory leaks): cuando utilices StateFlow, asegúrate de cancelar el flujo cuando ya no sea necesario para evitar fugas de memoria. Esto se puede hacer llamando a `cancel()` o usando las funciones `launchIn` o `collect`.

- Ten en cuenta el orden de las operaciones: asegúrate de realizar las operaciones de la base de datos en el orden correcto para evitar errores.

Al momento de usar StateFlow y Room…

- Ten cuidado con los subprocesos: Room ejecuta las operaciones de la base de datos en un subproceso en segundo plano de forma predeterminada, pero StateFlow se ejecuta en el subproceso principal. Por lo tanto, si necesita realizar una operación de base de datos y actualizar StateFlow, debe asegurarse de hacerlo en el subproceso correcto.

- Ten en cuenta el tamaño de los datos: no se recomienda utilizar StateFlow para emitir grandes conjuntos de datos. Esto puede hacer que su aplicación se bloquee debido a la falta de memoria o que la interfaz de usuario se congele.

- Ten en cuenta el manejo de errores: StateFlow no maneja los errores de forma predeterminada, por lo que debe manejarlos de manera adecuada.

Al momento de usar StateFlow y Room…

En resumen, la integración de StateFlow con Room puede ser una combinación poderosa, pero es importante tomar las precauciones necesarias para garantizar que su aplicación sea estable y eficiente.

Agregar soporte para base de datos en una app, es el primer gran paso para empezar a desarrollar apps a nivel profesional

Próxima sesión...

- *Construye una aplicación Android que integra LiveData, ROOM y el patrón MVVM para resolver un problema*

{desafío}
latam_

Academia de
talentos digitales