**Sesión 3-07 Clase del 9 de enero**

En esta sesión seguimos trabajando con Room para desarrollar una app que incluye todas las funciones CRUD en sus funcionalidades.

Seguimos trabajando en el proyecto de la Sesion3\_07.

En **Archivos\_Sesion3\_07.rar** de la carpeta RECURSOS de Moodle tienes varios archivos fuente que puedes utilizar:

* El archivo **CreateAnimalView.kt** contine el código fuente base para una pantalla **CreateAnimalView** que cargar los datos de un animal.

1.- Añade en el proyecto un archivo para la pantalla **CreateAnimalView** copiando el código del archivo **CreateAnimalView.kt** proporcionado en RECURSOS. En la pantalla se incluyen TextFields para introducir nombre, características y url de la imagen de un animal, así como el número de id de la familia a la que pertenece un animal (esta parte debería hacerse con RadioButtons que permiten seleccionar la familia, pero no lo vamos a hacer así).

En la pantalla hay un botón para cancelar y otro para confirmar la operación. De momento esos dos botones están programados para navegar hacia atrás usando el **navController** que recibe **CreateAnimalView.**

Se tiene también programado que al editar la url de la imagen de un animal, se represente en un componente **GlideImage** la imagen correspondiente a la url. Debes agregar en el proyecto la dependencia de Glide.

Añade un componente de navegación en el proyecto para que inicie la navegación en **HomeView** y permita navegar a **CreateAnimalView.**

2.- Añade en **HomeView** un Botón de Acción Flotante **añadir** (puedes hacerlo usando un Scaffold dentro de **HomeView**.

Establece que **HomeView** reciba un segundo parámetro para el navegador (navController).

Establece que al clicar el botón **añadir** se navegue a **CreateAnimalView.**

Comprueba la funcionalidad inicial de la navegación y de **CreateAnimalView**.

3.- Añade en el ViewModel una función para insertar un animal.

fun insertarAnimal(animalEd: Animal) {  
 *viewModelScope*.*launch*(Dispatchers.IO) **{** appDatabase.animalDao().insertAnimal(animalEd)  
 **}**}

4.- Programa el botón **Añadir** de la pantalla **CreateAnimalView** para que al clicarlo, usando el método **insertarAnimal** del ViewModel, se inserte un animal en la base de datos con los datos editados en la pantalla.

val animalEd= Animal(  
 nombre = nombreEd,  
 imagenUrl = urlEd,  
 caracteristicas = caracteristicasEd,  
 familiaId = familiaIdEd.*toIntOrNull*() ?: 0  
 )  
homeViewModel.insertarAnimal(animalEd)  
navController.popBackStack()

Comprueba la funcionalidad.

Verás que, al volver a la pantalla principal, no aparece el animal añadido en la lista de animales (aunque realmente se ha añadido a la base de datos). Eso es porque en el ViewModel no se ha refrescado el contenido del LiveData de la lista de animales.

Una posible solución es modificar el método **insertarAnimales** para que, una vez insertado el animal, llamar al método **obtenerAnimales** para que así se recargue el LiveData de la lista de animales y lo publique en **HomeView.**

fun insertarAnimal(animalEd: Animal) {  
 *viewModelScope*.*launch*(Dispatchers.IO) **{** appDatabase.animalDao().insertAnimal(animalEd)  
 obtenerAnimales() // Recargar la lista de animales después de la eliminación  
 **}**}

5.- La solución anterior, aunque es funcional, no es muy adecuada ya que, siempre que hagamos en una aplicación operaciones de modificará datos, tendríamos que recargar las listas con un método del ViewModel.

Dentro de la librería de corrutinas se incluye el tipo de dato **Flow** en el que se registra el estado de una query sobre una tabla cuando se realiza cualquier actualización en la tabla.

Para “enganchar” un **Flow** a una query sobre una tabla, lo tenemos que realizar en un DAO. En nuestro caso, tenemos que modificar la función **getAllAnimales** de **AnimalDao** para que devuelva un **Flow<List<Animal>>** en lugar de **List<Animal>.**

@Query("SELECT \* FROM animales")  
fun getAllAnimales(): Flow<List<Animal>>

*Las funciones de un DAO que devuelven* ***Flow*** *no se definen con* ***suspend*** *(no se necesita que se ejecuten dentro de una corrutina).*

Ahora, en el **ViewModel**, tienes que cambiar las dos instrucciones de definición del **LiveData** para la lista de animales. En su lugar vamos a usar un **StateFlow**:

// LStateFlow para servir la lista de animales  
val animales: StateFlow<List<Animal>> = appDatabase.animalDao()  
 .getAllAnimales() // Esto devuelve un Flow<List<Animal>> desde el DAO  
 .*stateIn*(  
 scope = *viewModelScope*, // Vinculado al ciclo de vida del ViewModel  
 started = SharingStarted.WhileSubscribed(5000), // si no hay pantalla que consuma el Flow durante 5 segundos,

// Flow se desconecta y deja de consumir recursos.

// Aunque en el momento que una pantalla lo consume, Flow de conecta de nuevo

initialValue = *emptyList*() // Valor inicial para los observadores o consumidores del Flow  
 )

Ahora, puedes eliminar la función **obtenerAnimales** del **ViewModel**, no se necesita. También debes eliminar la llamada a esa función en las funciones **eliminarAnimal e insertarAnimal.** Y también debes dejar comentado todo el código de la función **iniciar** (más adelante lo usaremos).

Por último, hay que modificar **HomeView** para que la variable **animales** sea una lista de animales “observador” del **StateFlow** anterior.

// Observar la lista de animales desde el State del ViewModel  
val animales by homeViewModel.animales.collectAsState()

Todo deberá funcionar OK ahora.

6.- Pasamos a desarrollar la funcionalidad del botón **Editar** incluido en **AnimalBox** de **HomeView.**

6.1.- Primero debes crear una nueva pantalla **EditAnimalView** copia de la que tienes descargada e incluida en **Archivos\_Sesion3\_07.**

En la pantalla tienes:

Una variable observador del LiveData de un Animal del ViewModel.

Variables de estado asociadas a los TextField de la pantalla.

* Un componente **GlideImage** en el que tendrás que hacer la modificación oportuna para que muestre la imagen del animal observado,
* Un componente Text en el que tendrás que hacer la modificación para que muestre el id del animal observado.
* Componentes TextField que muestran inicialmente datos del animal observado. No hay que hacer nada sobre ellos, su contenido ya está asociado a las variables de estado.
* Un botón **cancelar** ya programado correctamente, navegará a pantalla principal.
* Un botón **confirmar** programado inicialmente para que navegue a pantalla principal.

6.2.- Añade al componente de navegación el elemento para navegar a **EditAnimalView.**

6.3.- Programa el botón **editar** de **AnimalBox** para que:

* Use la función del **ViewModel** para obtener un animal por su Id y lo asigne al **LiveData**.
* Navegue a la pantalla **EditAnimalView** (que tiene el observador del **LiveData** y que, por tanto, va a trabajar con el animal en el que se ha clicado el botón editar).

6.4. Añade una función **modificarAnimal** en el **ViewModel** para modificar un animal usando el método adecuado de **AnimalDao**.

6.5.- En el botón **Confirmar** de la pantalla **EditAnimalView** programa que se construya un **Animal** con los datos de la pantalla de edición y se haga la llamada para modificar a través del ViewModel.

val animalEd= Animal(  
 id=animal!!.id, //confiamos en que animal no es null  
 nombre = nombreEd,  
 caracteristicas = caracteristicasEd,  
 imagenUrl = animal.imagenUrl,  
 familiaId = familiaIdEd.*toIntOrNull*() ?: animal.familiaId  
)  
homeViewModel.modificarAnimal(animalEd)

navController.popBackStack()

Ahora ya deberá tener la pantalla la funcionalidad deseada.

7.- Pasamos a usar las **preferencias** en este proyecto. Las preferencias nada tienen que ver con almacenamiento en base de datos.

Las preferencias son datos que se almacenan en un dichero XML(de preferencias) dentro de la memoria interna del dispositivo. Se gestionan mediante la librería **DataStore.** Generalmente en las preferencias se almacenan:

* Datos de configuración de la aplicación en el dispositivo.

Pero también se pueden dedicar a almacenar otros datos como:

* Datos de uso de la aplicación.
* Control de primera ejecución de la aplicación.

Lo que vamos a hacer inicialmente es activar un archivo de preferencias y usarlo para almacenar:

* Un control de estado de primera ejecución para que, en caso ser la primera ejecución, se añadan unos datos iniciales a las tablas animales y familias de nuestra base de datos.
* Un control de la fecha y hora en que se realizó la instalación.
* Un control sobre el número de veces que se ha ejecutado la aplicación.

7.1.- Creamos una clase PreferencesManager (por ejemplo, en el package data.preferences) con este código:

// Extensión para DataStore. Nombre de archivo app\_mis\_preferencias  
private val Context.*dataStore* by *preferencesDataStore*("app\_mis\_preferencias")  
  
class PreferencesManager(context: Context) {  
 // Instancia de DataStore  
 private val dataStore = context.*dataStore* // Claves para los datos  
 companion object {  
 //preferencia de primera ejecución cuya key es primer\_ejecucion  
 val PRIMERA\_EJECUCION\_KEY = *booleanPreferencesKey*("primer\_ejecucion")  
 }  
  
 // Flow para observar si es la primera ejecución  
 val isFirstExecution: Flow<Boolean> = dataStore.data //lee los datos de preferencias  
 .*catch* **{** exception **->** if (exception is IOException) {  
 emit(*emptyPreferences*()) // Si no se puede leer, emite valores vacíos  
 } else {  
 throw exception  
 }  
 **}** .*map* **{** preferences **->** //si no existe la key devuelve true  
 preferences[PRIMERA\_EJECUCION\_KEY] ?: true  
 **}** // Función para registrar la primera ejecución  
 suspend fun setFirstExecution(value: Boolean) {  
 dataStore.edit **{** preferences **->** preferences[PRIMERA\_EJECUCION\_KEY] = value  
 **}** }  
  
}

7.2.- Modificamos el ViewModel para que reciba una instancia de PreferencesManager como parámetro:

class HomeViewModel(  
 private val appDatabase: AppDatabase,  
 private val preferencesManager: PreferencesManager  
) : ViewModel() {

Y en ese ViewModel cargamos el código para el inicio del ViewModel, para que cargue o inserte los animales y las familias si es la primera ejecución:

init{  
 *viewModelScope*.*launch* **{** // Verificar si es la primera ejecución  
 val isFirstExecution = preferencesManager.isFirstExecution.first()  
 if (isFirstExecution) {  
 iniciarAnimales()  
 iniciarFamilias()  
 // Marcar como no primera ejecución  
 preferencesManager.setFirstExecution(false)  
 }  
 **}**}

7.3.- Ahora en **MainActivity** hay que crear la instancia de **PreferencesManager** y pasarla al **ViewModel**:

val database = DatabaseProvider.getDatabase(this)  
val preferencesManager = PreferencesManager(*applicationContext*)  
val homeViewModel= HomeViewModel(database,preferencesManager)

8.- Vamos ahora a usar nuevas preferencias en la aplicación:

8.1.- Agregamos en **PreferencesManager** código para definir nuevas preferencias de fecha de instalación y contador de ejecuciones:

// Claves para los datos  
companion object {  
 //preferencia de primera ejecución cuya key es primer\_ejecucion  
 val PRIMERA\_EJECUCION\_KEY = *booleanPreferencesKey*("primera\_ejecucion")  
 val FECHA\_PRIMERA\_EJECUCION\_KEY = *stringPreferencesKey*("fecha\_primera\_ejecucion")  
 val CONTADOR\_EJECUCIONES\_KEY = *intPreferencesKey*("contador\_ejecuciones")  
}

8.2.- Agregamos el código para los Flows que controlan el estado de las nuevas preferencias:

// Flow para observar la fecha de la primera ejecución  
val firstExecutionDate: Flow<String?> = dataStore.data  
 .*catch* **{** exception **->** if (exception is IOException) {  
 emit(*emptyPreferences*())  
 } else {  
 throw exception  
 }  
 **}** .*map* **{** preferences **->** preferences[FECHA\_PRIMERA\_EJECUCION\_KEY]  
 **}**// Flow para observar el contador de ejecuciones  
val executionCount: Flow<Int> = dataStore.data  
 .*catch* **{** exception **->** if (exception is IOException) {  
 emit(*emptyPreferences*())  
 } else {  
 throw exception  
 }  
 **}** .*map* **{** preferences **->** preferences[CONTADOR\_EJECUCIONES\_KEY] ?: 0  
 **}**

8.3.- Agregamos en la clase una función para incrementar en uno el contador de ejecuciones:

// Función para incrementar el contador de ejecuciones  
suspend fun incrementContadorEjecuciones() {  
 dataStore.edit **{** preferences **->** val currentCount = preferences[CONTADOR\_EJECUCIONES\_KEY] ?: 0  
 preferences[CONTADOR\_EJECUCIONES\_KEY] = currentCount + 1  
 **}**}

8.4.- Y agregamos el código para que en la primera ejecución de la app se guarde la fecha y hora de la primera ejecución:

// Función para registrar la fecha de la primera ejecución  
suspend fun setFirstExecution(value: Boolean) {  
 dataStore.edit **{** preferences **->** preferences[PRIMERA\_EJECUCION\_KEY] = value  
 if (value) {  
 val currentDate = LocalDateTime.now()  
 val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")  
 preferences[FECHA\_PRIMERA\_EJECUCION\_KEY] = currentDate.format(formatter)  
 }  
 **}**}

8.5.- No vamos a hacer nada con estas nuevas preferencias en las pantallas de la aplicación. Para demostrar su funcionamiento, debes hacer en MainActivity para que:

* Se muestre un Log que indique cuanto tiempo ha transcurrido desde la primera ejecución hasta la actual.
* Se incremente el contador de ejecuciones y se muestre un Log con el valor actual del contador.

Necesariamente este código, ya que va a usar funciones suspend, debe ejecutarse dentro de una corrutina. En este caso, por estar en MainActivity y afectar a una ejecución completa, dentro de un bloque:

*lifecycleScope*.*launch* **{**   
**}**