Programación multimedia y dispositivos móviles

Actividad 3.3. El componente Room

Francisco José García Cutillas | 2FPGS_DAM

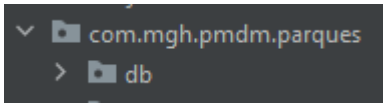
Índice

Ejercicio 1

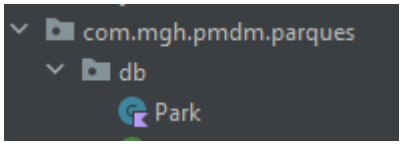
Siguiendo con el caso práctico anterior, y con el proyecto ya configurado para usar la librería Room, vamos a crear los diferentes componentes necesarios para dotar de persistencia a nuestra aplicación.

Para dotar de persistencia a nuestra aplicación, debemos de asociar la misma a una base de datos que sea capaz de almacenar todos los datos de la misma.

Por ello, para comenzar, vamos a crear un directorio dentro de nuestro paquete para tenerlo todo más organizado. Como ahí vamos a realizar todo lo relativo a la base de datos, lo llamamos "db".



Dentro de este paquete vamos ahora a introducir la clase Park.kt que teníamos anteriormente dentro del paquete "model".



Le vamos a añadir la anotación @Entity para determinar que ésta va a ser una entidad de nuestra base de datos. La clase quedaría de la siguiente manera:

```
package com.mgh.pmdm.parques.db

import androidx.room.Entity
import androidx.room.PrimaryKey
import java.io.Serializable

@Entity
data class Park(
    @PrimaryKey(autoGenerate = true)
    var id:Long = 0,
    var name: String,
    var desc: String?,
    var phone: String?,
    var webiste: String?,
    var openingTime: String?,
    var closingTime: String?,
    var sports: Boolean?,
    var children: Boolean?,
    var Bar: Boolean?,
    var Pets: Boolean?
): Serializable
```

Con respecto a la anterior clase, también se le ha añadido la anotación @PrimaryKey a una nueva variable que hemos creado con el nombre “id”, la cual se va a autogenerar para cada parque que guardemos en nuestra base de datos. Este dato lo proporcionará automáticamente la base de datos, por lo que no tendremos que preocuparnos nosotros por darle valor, aunque por defecto le pongamos el valor 0.

En este mismo paquete db, vamos a crear también una interfaz “ParkDAO” que va a ser la plantilla que va a tener que seguir cualquier clase que implemente la misma. En ella vamos a tener las operaciones básicas a realizar a la base de datos (consultar todos los parques, insertar un parque, actualizar un parque o borrar un parque).

Para todas estas operaciones vamos a utilizar las anotaciones @Dao para indicar que se trata de una interfaz de acceso a datos, @Query para consultar todos los parques, @Insert para insertar un parque, @Update para actualizar un parque y finalmente @Delete para borrar un parque.

Esta interfaz quedaría de la siguiente manera:

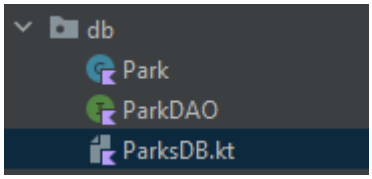
```
package com.mgh.pmdm.parques.db

import androidx.lifecycle.LiveData
import androidx.room.*

@Dao
interface ParkDAO {

    @Query("SELECT * from Park")
    fun getAll(): LiveData<List<Park>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun addPark(park: Park)

    @Update
    suspend fun updatePark(park: Park)

    @Delete
    suspend fun deletePark(park: Park): Int
}
```

Podemos observar que los métodos a los que hacen referencia las anotaciones, son métodos “suspend”. Esto quiere decir que dichos métodos se invocarán mediante corrutinas que se ejecutarán fuera del hilo principal de la aplicación.

Por último, vamos a añadir al paquete db la clase “ParksDB.kt”. Esta clase va a ser la encargada de realizar la conexión con la base de datos SQLite, en nuestro caso llamada “parks-db”.

```kotlin
package com.mgh.pmdm.parques.db

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [Park::class], version = 1)
abstract class ParksDB : RoomDatabase() {

    abstract fun parkDao(): ParkDAO
}

object DatabaseBuilder {

    private var INSTANCE: ParksDB? = null

    fun getInstance(context: Context): ParksDB {
        if (INSTANCE == null) {
            synchronized(ParksDB::class) {
                INSTANCE = buildRoomDB(context)
            }
        }
        return INSTANCE!!
    }

    private fun buildRoomDB(contexto: Context) =
        Room.databaseBuilder(
            contexto.applicationContext,
            ParksDB::class.java,
            name: "parks-db"
        ).build()

}
```

Finalmente, la estructura de nuestro paquete db quedaría de la siguiente manera:



Ahora nos vamos a cambiar al paquete “repository”, en el que se encuentra la clase “ParkRepository.kt”. Para nuestro caso actual, al utilizar la base de datos, ya no vamos a utilizar el json de parques ejemplo que teníamos en ejercicios anteriores. Ahora vamos a pedir los parques contenidos en dicha base de datos. La clase quedaría de la siguiente forma:

```
package com.mgh.pmdm.parques.repository

import android.content.Context
import androidx.lifecycle.LiveData
import com.mgh.pmdm.parques.db.DatabaseBuilder
import com.mgh.pmdm.parques.db.Park

class ParkRepository private constructor(
    private var context: Context
) {

    companion object {
        private var INSTANCE: ParkRepository? = null
        fun getInstance(context: Context): ParkRepository {
            if (INSTANCE == null) {
                INSTANCE = ParkRepository(context)
            }
            return INSTANCE!!
        }
    }

    fun getParks(): LiveData<List<Park>> {
        return DatabaseBuilder.getInstance(context).parkDao().getAll()
    }

    suspend fun removePark(park: Park): Int =
        DatabaseBuilder.getInstance(context).parkDao().deletePark(park)

    suspend fun update(park: Park) = DatabaseBuilder.getInstance(context).parkDao().updatePark(park)

    suspend fun add(park: Park) = DatabaseBuilder.getInstance(context).parkDao().addPark(park)

}
```

Resaltar que al igual que en el DAO, las funciones de borrar, actualizar o añadir parques deben declararse también como “suspend”, por el mismo motivo que hemos comentado anteriormente. Estas funciones sólo pueden ser invocadas desde funciones también suspend o corrutinas.

Una vez realizado lo anterior, nos vamos a ir a la clase “MainViewModel.kt” localizada dentro del paquete viewModel. Esta clase es la encargada de acceder desde el código a la base de datos, por lo que también será la encargada de comunicar a la interfaz todos aquellos posibles cambios.

Vamos a ir comentando la clase paso por paso:

Cabecera de la clase.

```
package com.mgh.pmdm.parques.viewmodel

import android.app.Application
import android.view.View
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import com.mgh.pmdm.parques.db.Park
import com.mgh.pmdm.parques.repository.ParkRepository
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainViewModel(application: Application) : AndroidViewModel(application) {

    /* Variable estática con la que nos vamos a ahorrar escribir código cada vez que
    *  vayamos a hacer referencia al repositorio de los parques */
    val repository = ParkRepository.getInstance(application.applicationContext)

    /* Vamos a crear una variable estática para usarla únicamente dentro de esta clase.
    *  Esto se hace así debido a que si queremos referenciar la clase dentro de un callback,
    * "this" haría referencia al propio callback y no a la clase */
    private val me = this
```

Ahora vamos a representar los LiveData que están observados desde la interfaz.

```
24
25      /* Parque que estamos editando actualmente. Atributo privado para acceder al mismo sólo
26      *  desde esta clase */
27      private var _currentPark = MutableLiveData<Park?>()
28      val currentPark: LiveData<Park?> = _currentPark
29
30      // Obtenemos los parques contenidos en el repositorio
31      val parkList: LiveData<List<Park>> by lazy {
32          repository.getParks()
33      }
34
35      /* Variables para notificar a la interfaz cuando un parque ha sido guardado
36      *  o actualizado en la base de datos */
37      var parkSaved: MutableLiveData<Boolean> = MutableLiveData()
38      var parkUpdated: MutableLiveData<Boolean> = MutableLiveData()
39
40      // Notificación a la interfaz la posición de un parque se ha borrado
41      var deletedPos: MutableLiveData<Int> = MutableLiveData()
42
43      // Notificación a la interfaz de que se ha hecho un clic largo sobre un parque
44      val parkLongClicked: MutableLiveData<Park> by lazy {
45          MutableLiveData<Park>()
46      }
47
48      // Notificación a la interfaz de que se ha hecho un clic sobre un parque
49      val parkClicked: MutableLiveData<Park> by lazy {
50          MutableLiveData<Park>()
51      }
52
53      /* El adaptador también va a ser observado desde la interfaz. Primero lo hemos creado
54      *  como un atributo privado, ya que la referencia al ViewModel la hacemos desde aquí con "me" */
55      private val _adaptador = MutableLiveData<AdaptadorParques>().apply { this: MutableLiveData<AdaptadorParques>
56          value = AdaptadorParques(
57              me,
58              { park: Park, v: View -> parkClickedManager(park, v) },
59              { park: Park, v: View -> ParkLongClickedManager(park, v) }
60          )
61      }
62
63      /* Al declarar el adaptador antes como privado, necesitamos un atributo público al que
64      *  si podamos acceder desde fuera de la clase. Lo realizamos de la siguiente manera: */
65      val adaptador: MutableLiveData<AdaptadorParques> = _adaptador
```

Finalmente vamos a acabar con la definición de esta clase con sus métodos para limpiar un parque en edición o establecerlo para editarlo, así como los métodos para saber si se ha hecho un clic corto o largo sobre un parque.

```
67      //Método para limpiar el parque actual en edición
68      fun cleanPark() {
69          _currentPark.value = null
70      }
71
72      //Método para establecer el valor del parque actual en edición
73      fun setCurrentPark(park: Park) {
74          _currentPark.value = park.copy()
75      }
76
77      //Método que recoge si se ha pulsado sobre un parque
78      private fun parkClickedManager(park: Park, v: View) {
79          parkClicked.value = park
80      }
81
82      //Método que recoge si se ha pulsado un clic largo sobre un parque
83      private fun ParkLongClickedManager(park: Park, v: View): Boolean {
84          parkLongClicked.value = park
85          return true
86      }
87
```

Los dos últimos métodos de esta clase son los que van a gestionar el borrado o guardado de un parque en la base de datos.

```
    //Método para borrar un parque y notificar al adaptador la posición en la que estaba
    fun removePark(park: Park) {

        viewModelScope.launch(Dispatchers.IO) { this: CoroutineScope
            repository.removePark(park)

            parkList.value?.indexOf(park)?.let { it: Int
                deletedPos.postValue(it);
            }
        }

    }

    /* Método para guardar un parque y notificar al adaptador si se ha actualizado un parque
     * o creado uno nuevo */
    fun savePark(park: Park) {

        viewModelScope.launch(Dispatchers.IO) { this: CoroutineScope

            if (_currentPark.value != null) {
                repository.update(_currentPark.value as Park)

                parkList.value?.indexOf(_currentPark.value)
                    ?.let { adaptador.value?.notifyItemChanged(it) }

                parkUpdated.postValue( value: true)
            } else {
                repository.add(park)

                _currentPark.postValue(park.copy())

                adaptador.value?.notifyDataSetChanged()

                parkSaved.postValue( value: true)
            }

        }

    }
}
```

Podemos observar que los Scope utilizados tienen asociados el dispatcher.IO, por lo que esto significa que el guardado o borrado se realizará en otro hilo distinto al de la interfaz.

Ahora nos pasamos al paquete view -> ui a su clase “FirstFragment.kt” y a ella vamos a añadir lo siguiente dentro del método onViewCreated:

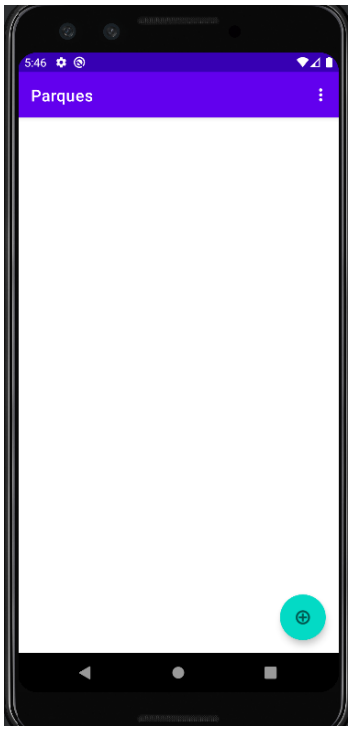
```
/* Observador para la lista de parques. Con esto se actualizará el adaptador,
 * dependiendo de si se ha actualizado la lista o borrado algún parque */
viewModel.parkList.observe(viewLifecycleOwner) { parks ->
    parks.let { it: List<Park>!
        if (viewModel.deletedPos.value == null) {
            // Actualización general
            viewModel.adaptador.value?.notifyDataSetChanged()
        } else {
            viewModel.adaptador.value?.notifyItemRemoved(viewModel.deletedPos.value!!)
            viewModel.deletedPos.value=null
        }
    }
}
```

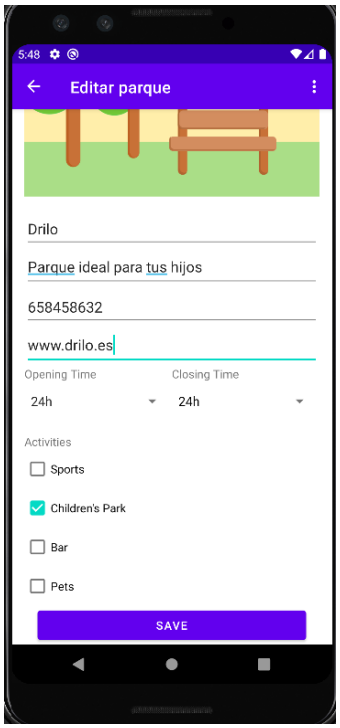
Finalmente, dentro del mismo paquete que anteriormente, pero en la clase “SecondFragment.kt”, vamos a añadir lo siguiente al método ya creado prepareObservers:

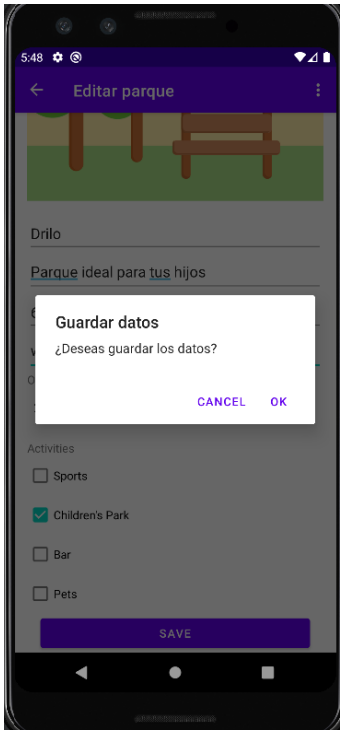
```
// Observamos para cuando se guarde un parque nuevo, se muestre un Snackbar
viewModel.parkSaved.observe(viewLifecycleOwner) { saved ->
    saved?.let { it: Boolean
        viewModel.parkSaved.value=null
        Snackbar.make(
            binding.root,
            resources.getString(R.string.dataSaved),
            Snackbar.LENGTH_LONG
        ).show()
    }
}

// Observamos para cuando se actualice un parque, se muestre un Snackbar
viewModel.parkUpdated.observe(viewLifecycleOwner) { saved ->
    saved?.let { it: Boolean
        viewModel.parkUpdated.value=null
        Snackbar.make(
            binding.root,
            resources.getString(R.string.updatedSaved),
            Snackbar.LENGTH_LONG
        ).show()
    }
}
```

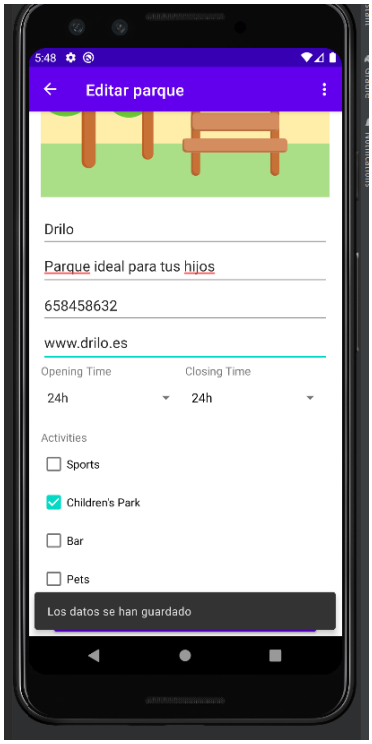
Para este último caso, hemos tenido que añadir también en res -> values -> strings.xml el valor que saldrá en el Snackbar al actualizar un parque.

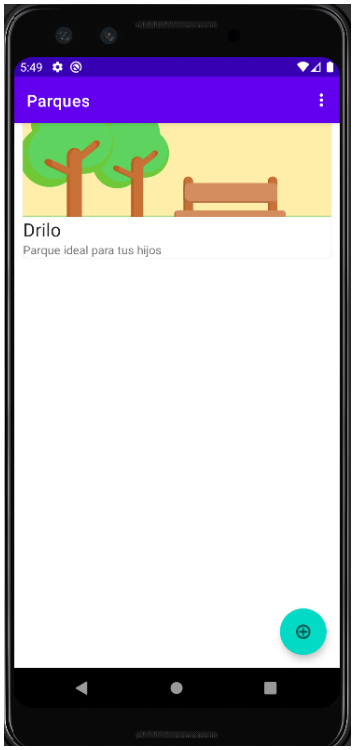
```
<string name="updatedSaved">Los datos del parque han sido actualizados</string>
```

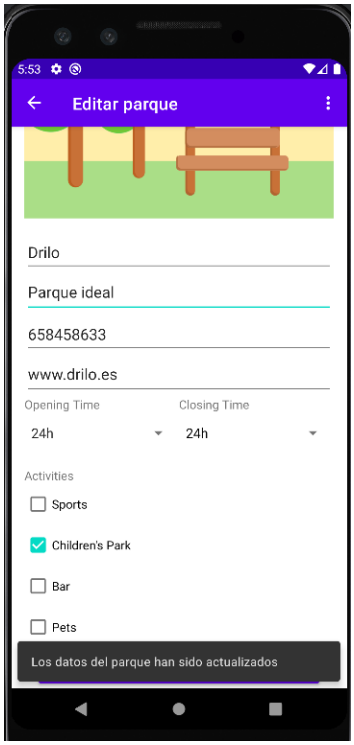

5:46
Parques

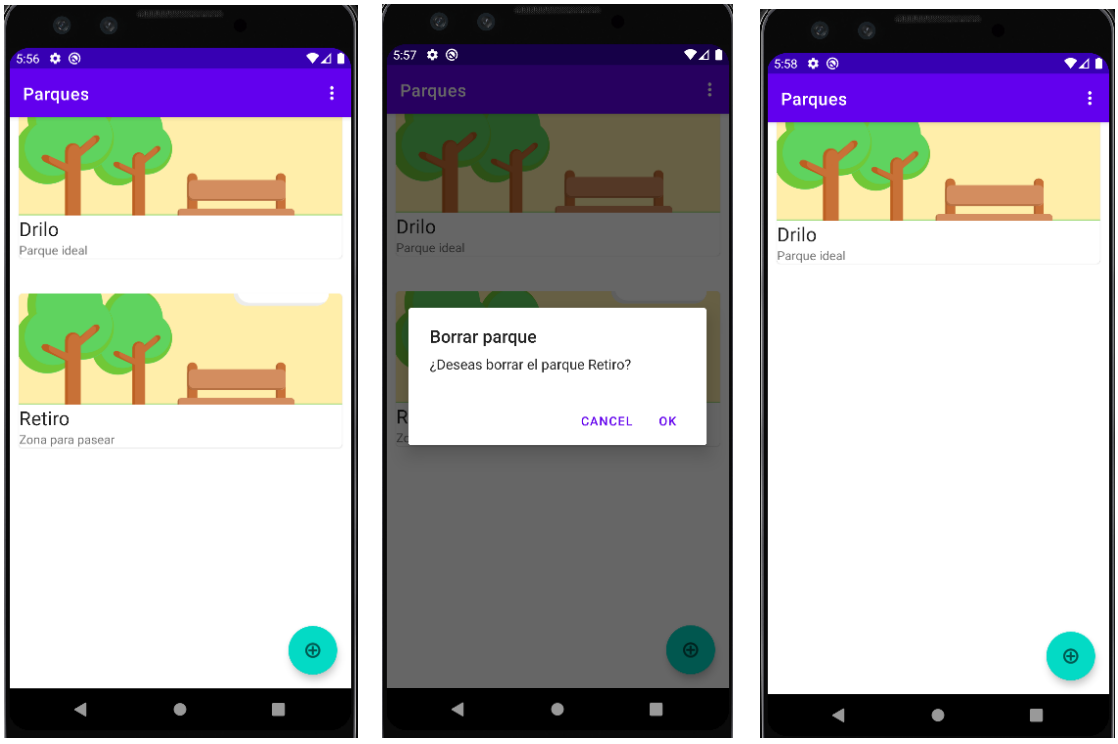

5:48
Editar parque
Drilo
Parque ideal para tus hijos
658458632
www.drilo.es
Opening Time
Closing Time
24h
24h
Activities
Sports
Children's Park
Bar
Pets
SAVE


5:48
Editar parque
Drilo
Parque ideal para tus hijos
Guardar datos
¿Deseas guardar los datos?
CANCEL
OK
Activities
Sports
Children's Park
Bar
Pets
SAVE


5:48
Editar parque
Drilo
Parque ideal para tus hijos
658458632
www.drilo.es
Opening Time
Closing Time
24h
24h
Activities
Sports
Children's Park
Bar
Pets
Los datos se han guardado


5:49
Parques
Drilo
Parque ideal para tus hijos


5:53
Editar parque
Drilo
Parque ideal
658458633
www.drilo.es
Opening Time
Closing Time
24h
24h
Activities
Sports
Children's Park
Bar
Pets
Los datos del parque han sido actualizados


5:56
Parques
Drilo
Parque ideal
Retiro
Zona para pasear
5:57
Parques
Drilo
Parque ideal
Borrar parque
¿Deseas borrar el parque Retiro?
CANCEL
OK
5:58
Parques
Drilo
Parque ideal