CENTRE ESPECÍFIC
D´EDUCACIÓ A DISTÀNCIA DE
LA COMUNITAT VALENCIANA

UD 06

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES 24/25
CFGS DAM

PERSISTENCIA DE DATOS

CONCEPTOS BÁSICOS SOBRE EL TRATAMIENTO DE DATOS DE UNA APP

Autor: Mara Vañó
m.vanoalonso@edu.gva.es
Fecha: 2024/2025

Índice

1. Objetivos de la unidad 1

2. Introducción 1

3. Almacenamiento más simple: Parceables y Bundle 1

3.1 Cómo enviar datos entre actividades 1

4. GUARDAR Y CARGAR LAS PREFERENCIAS DEL USUARIO 2

4.1 Métodos SharedPreferences 3

5. USO DEL ALMACENAMIENTO INTERNO 4

6. ALMACENAMIENTO EN TARJETA SD 5

7. Elegir la opción de almacenamiento 7

8. CREAR Y USAR BASES DE DATOS 7

8.1 Creación de la clase auxiliar DBAdapter 8

8.2 Usar la base de datos mediante programación 8

8.3 Bases de datos en la nube 15

9. BIBLIOGRAFÍA 16

1. OBJETIVOS DE LA UNIDAD

- Intercambiar valores entre activities mediante bundle
- Conocer la persistencia de datos
- Conocer los diferentes modos para asegurar la persistencia de datos
- Seleccionar el mejor método para el almacenamiento de datos
- Revisar las tendencias del tratamiento de datos en las apps

2. INTRODUCCIÓN

La persistencia es la capacidad de una aplicación para que los datos que utilice perduren en el tiempo al ejecutar la misma aplicación, es decir, almacenamos la información para poder recuperarla posteriormente y utilizarla. La persistencia de datos es importante porque, generalmente, al programar querremos reutilizar datos en el futuro.

En Android existen diferentes formas de guardar estos datos y usaremos una u otra en función de nuestras necesidades. Las formas que utilizamos son:

● Sistemas de archivos tradicionales

● Gestión de bases de datos relacionales mediante apoyo de bases de datos en SQLite

● Shared Preferences.

3. ALMACENAMIENTO MÁS SIMPLE: PARCEABLES Y BUNDLE

Los objetos Parcelable y Bundle están diseñados para usarse entre límites de procesos como con transacciones de IPC y Binder, entre actividades con intents y para almacenar el estado transitorio en los cambios de configuración. En esta página, se proporcionan recomendaciones y prácticas recomendadas para usar los objetos Parcelable y Bundle.

3.1 Cómo enviar datos entre actividades

Cuando una app crea un objeto Intent para usar en startActivity(android.content.Intent) al iniciar una nueva Actividad, la app puede pasar parámetros por medio del método putExtra(java.lang.String, java.lang.String).

Es recomendable que utilices la clase Bundle para configurar primitivas que el SO conozca en objetos Intent. La clase Bundle está muy optimizada para asociar y desasociar el uso de parcelas.

En el siguiente fragmento de código, se muestra un ejemplo de cómo realizar esta operación:

Para el 1º Activity

```
// Crea la instancia del bundle
val bundle = Bundle()

// guarda el valor de tipo string en el bundle mapeado con una clave
bundle.putString("key1", "Gfg :- Main Activity")
```

```
// Crea un intent

intent = Intent(this@MainActivity, SecondActivity::class.java)


// Pasa el bundle al intent

intent.putExtras(bundle)


// Empieza la actividad pasando el intent como parámetro

 startActivity(intent)
```

En el 2º Activity

```
// Recupera el bunble mediante la memoria de Android

val bundle = intent.extras


// comprueba que no haya nulos

var s:String? = null


// recupera la clave con el valor

s = bundle!!.getString("key1", "Default"))
```

4. GUARDAR Y CARGAR LAS PREFERENCIAS DEL USUARIO

Android proporciona el objeto **SharedPreferences** para ayudarte a guardar datos de aplicaciones simples. Por ejemplo, la aplicación puede tener una opción que permite a los usuarios especificar el tamaño de fuente utilizado en la aplicación. En este caso, la aplicación debe recordar el tamaño establecido por el usuario para que el tamaño se establezca correctamente cada vez que se abre la aplicación. Tienes varias opciones para guardar este tipo de preferencia:

- **Guardar datos en un archivo:** puedes guardar los datos en un archivo, pero debes realizar algunas rutinas de administración de archivos, como escribir los datos en el archivo, indicar cuántos caracteres leer de él, etc. Además, si tienes varias piezas de información para guardar, como el tamaño del texto, el nombre de la fuente, el color de fondo preferido, etc., entonces la tarea de escribir en un archivo se vuelve más engorrosa.

- **Escribir texto en una base de datos:** una alternativa a escribir en un archivo de texto es usar una base de datos. Sin embargo, guardar datos simples en una base de datos es excesivo, tanto desde el punto de vista de un desarrollador como en términos del rendimiento en tiempo de ejecución de la aplicación.

- **Uso del objeto sharedPreferences:** guarda datos mediante el uso de pares nombre/valor. Por ejemplo, especificamos un nombre para los datos que deseamos guardar y, a continuación, tanto él como su valor se guardarán automáticamente en un archivo XML.

4.1 Métodos SharedPreferences

SharedPreferences es parte de la API de Android desde el nivel 1 de la API. Es una interfaz que nos permite almacenar/modificar/eliminar datos localmente. Generalmente, se utiliza para almacenar en caché los datos locales del usuario, como los formularios de inicio de sesión. Los datos se almacenan en forma de un par clave-valor. Puede crear varios archivos para almacenar los datos de SharedPreferences.

Estos son los más utilizados:

- **El método getSharedPreferences(String, int)** se usa para recuperar una instancia de SharedPreferences. Aquí String es el nombre del archivo SharedPreferences e int es el contexto pasado.
- **SharedPreferences.Editor()** se usa para editar valores en SharedPreferences.
- Podemos llamar a **commit() o apply()** para guardar los valores en el archivo SharedPreferences. El commit() guarda los valores inmediatamente, mientras que apply() guarda los valores de forma asíncrona.

Podemos establecer valores en nuestra instancia de SharedPreference usando Kotlin de la siguiente manera.

```
val sharedPreference = getSharedPreferences("PREFERENCE_NAME",Context.MODE_PRIVATE)
var editor = sharedPreference.edit()
editor.putString("username","Anupam")
editor.putLong("l",100L)
editor.commit()
```

Para recuperar un

valor:

```
sharedPreference.getString("username","defaultName")

sharedPreference.getLong("l",1L)
```

Los tipos permitidos en una instancia de SharedPreference son:

- putBoolean
- putFloat
- putInt
- putString
- putStringSet
-

También podemos borrar todos los valores o eliminar un valor en particular llamando clear() y remove(String key) con los siguientes métodos.

```
editor.clear()

editor.remove("username")
```

5. USO DEL ALMACENAMIENTO INTERNO

Es la forma más sencilla de guardar información en tu dispositivo Android. El sistema proporciona directorios dentro del almacenamiento interno donde las apps pueden organizar sus archivos. Un directorio está diseñado para los archivos persistentes de la app y otro contiene los archivos almacenados en caché de la app. La app no requiere ningún permiso del sistema para leer y escribir en los archivos de estos directorios.

Las otras apps no pueden acceder a los archivos almacenados en el almacenamiento interno. Esto hace que el almacenamiento interno sea un buen lugar para los datos de apps a los que otras no deberían acceder.

Sin embargo, recuerda que esos directorios suelen ser pequeños. Antes de escribir archivos específicos de una app en el almacenamiento interno, la app debería consultar el espacio libre del dispositivo.

Los archivos persistentes comunes de la app están en un directorio al que puedes acceder con la propiedad filesDir de un objeto de contexto. El framework proporciona varios métodos para ayudarte a acceder y almacenar archivos en este directorio. Puedes usar la API de File para acceder a los archivos y almacenarlos.

Para ayudar a mantener el rendimiento de la app, no abras ni cierres el mismo archivo varias veces.

En el siguiente fragmento de código, se muestra cómo usar la API de File:

val file = File(context.filesDir, filename)

Como alternativa al uso de la API de File, puedes llamar a openFileOutput() para obtener una FileOutputStream que escriba en un archivo dentro del directorio filesDir.

En el siguiente fragmento de código, se muestra cómo escribir texto en un archivo:

```
val filename = "myfile"
val fileContents = "Hello world!"
context.openFileOutput(filename, Context.MODE_PRIVATE).use {
    it.write(fileContents.toByteArray())
}
```

Precaución: En dispositivos que ejecutan Android 7.0 (API nivel 24) o versiones posteriores, a menos que pases el modo de archivo Context.MODE_PRIVATE a openFileOutput(), se generará una SecurityException.

Para permitir que otras apps accedan a los archivos almacenados en el directorio dentro del almacenamiento interno, usa un FileProvider con el atributo FLAG_GRANT_READ_URI_PERMISSION.

Para leer un archivo como una transmisión, usa openFileInput():

```
context.openFileInput(filename).bufferedReader().useLines { lines ->
    lines.fold("") { some, text ->
```

```
        "$some\n$text"
    }
}
```

Puedes obtener un array con los nombres de todos los archivos del directorio filesDir llamando a fileList(), como se muestra en el siguiente fragmento de código:

```
var files: Array<String> = context.fileList()
```

Más información aquí:

https://developer.android.com/training/data-storage/app-specific?hl=es-419

6. ALMACENAMIENTO EN TARJETA SD

Con el avance de la tecnología y la mejora de los dispositivos la memoria SD de los dispositivos se destina principalmente para el almacenamiento de contenido multimedia o archivos pesados y no para el uso de aplicaciones.

Aquí tenéis un repaso actualizado del uso de la memoria de los dispositivos actuales, aunque aún podéis encontrar información sobre el uso de SD en vuestra app:

https://developer.android.com/training/data-storage

Es recomendable que el uso de SD se utilice únicamente para aplicaciones muy específicas donde se necesite un gran volumen de memoria en caso contrario es mejor utilizar la interna para mejorar la eficiencia de nuestra app.

La solución que elijas dependerá de tus necesidades específicas:

¿Cuánto espacio requieren tus datos?

El almacenamiento interno tiene espacio limitado para los datos específicos de la app. Usa otros tipos de almacenamiento si necesitas guardar una cantidad considerable de datos.

¿Qué tan confiable debe ser el acceso a los datos?

Si la funcionalidad básica de tu app requiere ciertos datos, como cuando tu app se inicia, coloca los datos en un directorio del almacenamiento interno o en una base de datos. No siempre se puede acceder a los archivos específicos de la app que se encuentran en el almacenamiento externo porque algunos dispositivos permiten que los usuarios quiten los dispositivos físicos que cumplen esa función.

¿Qué tipo de datos necesitas almacenar?

Si tienes datos que solo son importantes para tu app, usa el almacenamiento específico de la app. Para el contenido multimedia que se puede compartir, usa el almacenamiento compartido a fin de que otras apps puedan acceder a él. En el caso de los datos estructurados, usa las preferencias (para datos de clave-valor) o una base de datos (para datos que contengan más de 2 columnas).

¿Los datos deben ser privados para tu app?

Para almacenar datos sensibles (datos que no deberían ser accesibles desde ninguna otra app), usa el almacenamiento interno, las preferencias o una base de datos. El almacenamiento interno tiene el beneficio adicional de ocultar los datos a los usuarios.

Permisos y acceso al almacenamiento externo

Android define los siguientes permisos relacionados con el almacenamiento: READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE y MANAGE_EXTERNAL_STORAGE.

En las versiones anteriores de Android, las apps debían declarar el permiso READ_EXTERNAL_STORAGE para acceder a cualquier archivo fuera de los directorios específicos de la app en el almacenamiento externo. Además, las apps debían declarar el permiso WRITE_EXTERNAL_STORAGE para escribir en cualquier archivo fuera del directorio específico de la app.

Las versiones más recientes de Android se guían en mayor medida por el propósito de un archivo que por su ubicación para determinar la capacidad de una app de acceder a un archivo determinado y escribir en él. En particular, si tu app está orientada a Android 11 (nivel de API 30) o versiones posteriores, el permiso WRITE_EXTERNAL_STORAGE no tiene ningún efecto en el acceso de tu app al almacenamiento. Este modelo de almacenamiento basado en el propósito aumenta la privacidad del usuario, ya que las apps tienen acceso solo a las áreas del sistema de archivos del dispositivo que

realmente usan.

Android 11 introduce el permiso MANAGE_EXTERNAL_STORAGE, que brinda acceso de escritura a archivos fuera del directorio específico de la app y MediaStore. Si quieres obtener información sobre este permiso y los motivos por los que la mayoría de las apps no necesitan declararlo a fin de cumplir

con sus casos de uso, consulta la guía para administrar todos los archivos en un dispositivo de almacenamiento.

7. ELEGIR LA OPCIÓN DE ALMACENAMIENTO

Hemos visto cómo hacerlo mediante el objeto SharedPreferences, el almacenamiento interno y el almacenamiento externo.

¿Cómo elegimos el más adecuado?

- SharedPreferences: para datos que se pueden representar mediante pares de nombre / valor.
- Almacenamiento interno: Si necesitas almacenar datos ad-hoc.
- Tarjeta SD: Si necesitamos compartir luego los datos en la tarjeta SD sólo para archivos de gran tamaño.

Cuando los dispositivos Android carecían de memoria interna suficiente se utilizaba mucho la memoria de la tarjeta SD que es mucho más lenta que la interna del propio teléfono. Con el avance

de la tecnología y la mejora de los dispositivos la memoria SD de los dispositivos se destina principalmente para el almacenamiento de contenido multimedia y no para el uso de aplicaciones.

8. CREAR Y USAR BASES DE DATOS

Hasta ahora, todas las técnicas que has visto son útiles para guardar conjuntos de datos simples. Para guardar datos relacionales, usar una base de datos es mucho más eficiente.

Por ejemplo, si deseas almacenar los resultados de las pruebas de todos los estudiantes en una escuela, es mucho más eficiente usar una base de datos para representarlos porque puedes usar la consulta de la base de datos para recuperar los resultados de estudiantes específicos.

Además, el uso de bases de datos te permite hacer cumplir la integridad de los datos especificando las relaciones entre diferentes conjuntos de datos. Android usa el sistema de base de datos SQLite.

La base de datos que crea para una aplicación sólo es accesible para ella; otras aplicaciones no podrán acceder a ella. Aquí veremos cómo crear mediante programación una base de datos SQLite en su aplicación de Android.

Para Android, la base de datos SQLite que se crea mediante programación en una aplicación siempre se almacena en la carpeta / data / data / /bases de datos.

Cuando creamos una base de datos desde nuestra aplicación, básicamente estamos creando un

archivo que el sistema guardará en la carpeta privada de la app, de la misma forma que con los archivos del almacenamiento interno. De este modo se mantiene la privacidad de los datos, pues el directorio de la aplicación no es accesible para otros usuarios o aplicaciones.

Desde las primeras versiones de iOS y Android, la mejor forma de mantener una base de datos local ha sido mediante SQLite. Aunque existen muchos otros, el sistema de gestión de base de datos SQLite está escrito en C, por lo que es muy eficiente, es simple pero completo y ocupa muy poco.

8.1 Creación de la clase auxiliar DBAdapter

Para tratar con bases de datos se puede crear una clase auxiliar para encapsular todas las complejidades de acceder a los datos para que sean transparentes para el código de llamada. Para hacerlo, crea una clase auxiliar llamada DBAdapter, que crea, abre, cierra y usa una base de datos SQLite. En este ejemplo, creará una base de datos llamada MyDB que contiene una tabla llamada contactos. Esta tabla tiene tres columnas: _id, nombre y correo electrónico. 1.

Con Android Studio, crea un proyecto de Android y asígnale el nombre Bases de datos. 2. Agrega un nuevo archivo Java Class al paquete y asígnale el nombre DBAdapter 3. Agrega al fichero DBAdapter.java

8.2 Usar la base de datos mediante programación

Vamos a explicarlo mediante el siguiente ejemplo de código SQL. Imaginemos que nuestra aplicación es un videojuego y necesitamos guardar los datos de los usuarios y los puntos de cada partida.

Con el DDL de SQL crearíamos dos tablas:

```
CREATE TABLE jugadores (
jugador_id INTEGER PRIMARY KEY,
nombre TEXT NOT NULL UNIQUE,
avatar TEXT,
cinturon INTEGER,
);

CREATE TABLE partidas (
partida_id INTEGER PRIMARY KEY,
jugador_id INTEGER, fecha INTEGER,
puntos INTEGER,
nivel INTEGER
);
```

La tabla jugadores almacenará a los jugadores que han echado una partida hasta ahora. Sus campos serán: jugador id, un identificador único para identificar al jugador; nombre, que será el alias del jugador en la interfaz gráfica del juego; avatar, que será una URL o ruta del sistema de archivos hacia una imagen que identifique gráficamente al jugador en el juego; y un cinturón, que no es más que el nivel que ha alcanzado el jugador en el juego por puntos o niveles superados.

Por otro lado, la tabla partidas guardará los datos de todas las partidas jugadas por los diferentes jugadores, y sus camposserán: partida_id, que es el identificador único de cada partida; jugador_id, que identifica qué jugador en la partida; fecha de la partida; puntos conseguidos por el jugador en la partida; y nivel máximo alcanzado en la partida.

Imaginemos que un nuevo jugador echa una partida: mediante el DML de SQL guardaríamos esa nueva información:

INSERT INTO jugadores (jugador_id, nombre, avatar, cinturon)

VALUES (68, "Ninja Rookie", "http://img.com/ninja", 0)

INSERT INTO partidas (partida_id, jugador_id, fecha, puntos, nivel)

VALUES (3, 68, strftime('%s','now'), 30, 2)

Con el comando INSERT INTO introducimos los datos del jugador Ninja Rookie con el código identificador 68 en la tabla jugadores, y la partida que ha jugado con el identificador 3 y los 30 puntos obtenidos hasta el nivel 2 del juego se guardan en partidas. Como SQLite no tiene un tipo de datos para fechas, usaremos un número entero, como el número de segundos desde 1970, una forma clásica de contar el tiempo en computación. De ahí la función strftime, que convierte la fecha now a segundos %s y lo pasa a entero.

Cuando después quisiéramos recuperar los datos del jugador 68, haríamos la consulta:

*SELECT * FROM jugadores WHERE jugador_id = 68*

Y si quisiéramos recuperar todas las partidas del jugador 68 haríamos:

*SELECT * FROM partidas WHERE jugador_id = 68*

Como puede observarse, SQL es sencillo y potente, solo requiere algo de estudio y práctica. Si tuviésemos todos los datos desperdigados en un archivo del disco sería muy costoso acceder a cualquiera de ellos sin tener que leerlo todo y decodificarlo antes. Con SQL pedimos lo que queremos

y el sistema gestor de BBDD nos lo devuelve.

Sin embargo, como programadores, no seremos nosotros los que hablemos con la BBDD, será nuestra aplicación. Veamos entonces cómo podríamos crear y usar una base de datos SQLite desde nuestra aplicación Android con las funciones integradas de bajo nivel, es decir, sin utilizar ningún ORM como Room que nos facilite las cosas.

Lo primero es definir las estructuras que almacenarán nuestros datos, las tablas. Para ello heredamos de la clase SQLiteOpenHelper, que será la responsable de mantener esas estructuras en el archivo de base de datos:

class JuegoDbHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {

Como vemos, necesita un contexto, el nombre del archivo donde se almacenará la BBDD y la versión actual, que como en nuestro caso será la primera, podríamos definir como 1.

Después sobrescribimos la función onCreate para definir las tablas que darán forma a nuestra base de datos:

```
override fun onCreate(db: SQLiteDatabase) {
db.execSQL(SQL_CREATE_JUGADORES)
db.execSQL(SQL_CREATE_PARTIDAS)
}
```

Se nos pasa un objeto db de base de datos y llamamos a su execSQL, una función que ejecutará el código SQL que se le pase. Las constantes string SQL_CREATE_JUGADORES y SQL_CREATE_PARTIDAS que más tarde mostraremos son el código SQL para crear las tablas que deseamos. Si mientras creamosla BBDD desde nuestra aplicación resulta que ya existía otra con una versión anterior, se llamará a onUpdate, que será la encargada de actualizar los datos antiguos a la nueva estructura:

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
db.execSQL(SQL_DELETE_JUGADORES)
db.execSQL(SQL_DELETE_PARTIDAS)
onCreate(db)
```

}

Para no complicar el código, y puesto que solo tenemos una versión de nuestras tablas, hacemos lo más sencillo, que es eliminar las tablas antiguas y volverlas a crear con la nueva estructura, sin tener en cuenta que perderemos los datos de la versión previa en caso de que los hubiera.

El código completo sería:

```
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper

class JuegoDbHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {

 // Se llama cuando la base de datos aún no existe y debe crearse

override fun onCreate(db: SQLiteDatabase) {

// Ejecutamos el SQL que crea las tablas

db.execSQL(SQL_CREATE_JUGADORES)
db.execSQL(SQL_CREATE_PARTIDAS) }
```

//Se llama cuando hemos modificado la estructura de las tablas y hemos incrementado el número de la versión, pero aún tenemos una BBDD antigua con una versión anterior, aquí se actualizará

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
```

// Ejecutamos el SQL que borra las tablas, después el que las crea con la nueva versión

```
 db.execSQL(SQL_DELETE_JUGADORES)
db.execSQL(SQL_DELETE_PARTIDAS)
 onCreate(db)
}
companion object {
```

// Si necesitamos cambiar las tablas una vez que la app está en producción, debemos incrementar este número

```
const val DATABASE_VERSION = 1
const val DATABASE_NAME = "juego.db"
const val TABLE_JUGADORES = "jugadores"
private const val SQL_CREATE_JUGADORES =
"CREATE TABLE $TABLE_JUGADORES (" +
```

```
 "jugador_id INTEGER PRIMARY KEY," +
 "nombre TEXT NOT NULL UNIQUE," +
 "avatar TEXT," +
 "cinturon INTEGER"

private const val SQL_DELETE_JUGADORES =
        "DROP TABLE IF EXISTS $TABLE_JUGADORES"
const val TABLE_PARTIDAS = "partidas"
private const val SQL_CREATE_PARTIDAS = "CREATE TABLE $TABLE_PARTIDAS (" +
 "partida_id INTEGER PRIMARY KEY," + "jugador_id INTEGER NOT NULL," +
 "fecha INTEGER NOT NULL," +
 "puntos INTEGER NOT NULL," +
 "nivel INTEGER NOT NULL"
private const val SQL_DELETE_PARTIDAS = "DROP TABLE IF EXISTS $TABLE_PARTIDAS }
}
```

Hemos definido los comandos SQL como constantes string para tenerlos bien definidos solo en un lugar y no desperdigados por el código. Ahora podríamos introducir datos en nuestra nueva base de datos. Como programamos con Kotlin, nuestros datos serán objetos de alguna clase.

Definamos nuestra clase Jugador, asociada a la tabla jugadores:

```
data class Jugador(
        val id: Int,
```

```
    val nombre: String,
    val avatar: String?,
    val cinturon: Int?
)
```

Nada más que un data class de Kotlin con los campos de nuestro jugador. Añadamos una función a la clase para insertar el propio objeto en la BBDD:

```
fun introducirEnBBDD(db: SQLiteDatabase) {
 // Juntamos todos los campos en una colección ContentValues

val values = ContentValues().apply {
put("nombre", jugador.nombre)
put("avatar", jugador.avatar)

put("cinturon", jugador.cinturon)
}

// Insertamos la fila de datos, y el sistema devuelve el id
id = db?.insert(JuegoDbHelper.TABLE_JUGADORES, null, values)
}
```

Esta nueva función introducirEnBBDD recibe por parámetro un objeto SQLiteDatabase que crearemos mediante nuestro JuegoDbHelper. Este objeto de base de datos nos permite insertar, seleccionar, actualizar y borrar. Introducimos los datos en un objeto ContentValues, que no es más que un objeto contenedor de pares clave-valor, y llamamos a la función insert, que devolverá el identificador de la fila insertada en la tabla.

Para introducir un nuevo jugador, el código sería:

```
val dbHelper = JuegoDbHelper(context)
val db = dbHelper.writableDatabase
//Creamos base de datos en modo escritura
val jugador = Jugador(68, "Ninja Rookie", "http://img.com/ninja.png", 0)
jugador.insertarEnBBDD(db)
```

Primero creamos nuestra clase de ayuda para BBDD. Con ese objeto creamos nuestra base de datos db. Luego creamos el Jugador, y llamamos a su método insertarEnBBDD con el objeto db.

Como podemos observar, utilizar las funciones de bajo nivel para manejar nuestra base de datos es posible, pero requiere mucho código para cada tabla. Pasar un objeto de Kotlin a una fila de base de datos SQL esincómodo y repetitivo, entre otras cosas porque para cada acción de BBDD necesitamos desgranar los campos del objeto en pares clavevalor.

Además, necesitamos conocer el lenguaje SQL para definir las tablas y sus relaciones, y para codificar las consultas de selección, actualización y borrado. Por fortuna, existe Room, una librería con la que crear y acceder a los datos de nuestra BBDD SQLite de forma fácil, eficiente y orientada a objetos. Veamos cómo utilizar este ORM.

Lo primero es añadir la librería en Gradle para que esté disponible en nuestro proyecto:

```
implementation "androidx.room:room-ktx:$room_version"
kapt "androidx.room:room-compiler:$room_version"
```

A continuación, definiríamos nuestras clases de Kotlin, al mismo tiempo que definimos las tablas de la base de datos:

```
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "jugadores")
data class JugadorEntity(
@PrimaryKey val id: Int,
 val nombre: String,
 val avatar: String?,
 val cinturon: Int?
 )
```

```
@Entity(tableName = "partidas")
data class PartidaEntity(
@PrimaryKey val id: Int,
  val jugador_id: Int,
 val fecha: Long,
 val puntos: Int?,
 val nivel: Int?
 )
```

Cuando el compilador ve las anotaciones de Room @Entity sobre nuestras clases, llamará a funciones internas de la librería Room que generarán código SQL para crear la base de datos con sus tablas jugadores y partidas.

De este modo, cuando hagamos operacionessobre la BBDD, podremos utilizar directamente nuestros objetos Kotlin y no tendremos que traducir parejas de clave-valor.

Así hemos definido las estructuras de datos; ahora definiremos las operaciones sobre los datos. Para

ello utilizamos un objeto DAO (Data Access Object) mediante la anotación de Room @Dao en nuestra interfaz JuegoDao:

```
import androidx.room.*
@Dao interface JuegoDao {
//SELECT
```

*@Query("SELECT * FROM jugadores ORDER BY nombre") suspend fun getJugadores(): List*

*@Query("SELECT * FROM partidas ORDER BY fecha DESC") suspend fun getPartidas(): List*

*@Query("SELECT * FROM jugadores WHERE id = :idJugador") suspend fun getJugadorById(idJugador: Int): JugadirEntity*

//INSERT

@Insert(onConflict = OnConflictStrategy.REPLACE) suspend fun addPartida(partida: PartidaEntity)

//DELETE

@Query("DELETE FROM partidas") suspend fun deletePartidas() }

Este es un ejemplo de DAO en el que hemos definido tres funciones de selección, una de inserción y otra de borrado sobre nuestras dos tablas, pero podemos crear fácilmente todas las que sean necesarias. Vemos que sobre cada función existe una anotación de Room que indica el tipo de comando y la orden SQL que ejecutar. También podemos preguntarnos qué significa la palabra clave suspend que hay delante de la definición de cada función. Esa palabra reservada indica que la función

puede suspender la ejecución del código. Entenderemos mejor qué significa en un capítulo posterior, cuando estudiemos el multiproceso.

Tengamos en cuenta, simplemente, que el acceso a la BBDD es muy lento, igual que el acceso a un archivo, a la red o a la cámara, por lo que tenemos que usar algún mecanismo por el que la interfaz gráfica de la aplicación siga respondiendo al usuario mientras otro código realiza alguna operación pesada como una consulta a base de datos o una llamada a un servicio web.

Ahora que hemos definido la estructura y la funcionalidad de la base de datos mediante nuestras entidades JugadorEntity y PartidaEntity y la interfaz DAO, solo nos queda crear la base de datos mediante la anotación @Database:

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

@Database(
entities = [JugadorEntity::class, PartidaEntity::class],
version = 1,
exportSchema = false)

abstract class JuegoDb : RoomDatabase() {
 abstract val dao: JuegoDao
    companion object {

        private const val NAME = "juego.db"
        fun buildDefault(context: Context) =
        Room.databaseBuilder(context,JuegoDb::class.java,NAME)
        .fallbackToDestructiveMigration()
        .build()
```

```
    }
}
```

Para crear nuestra clase de BBDD heredamos de RoomDatabase, que declara un objeto interno DAO y una función factoría que devuelve una instancia de la clase. Al crear la base de datos mediante las funciones de Room, podemos utilizar diferentes opciones.

Por ejemplo, en lugar de tener una BBDD permanente mediante un archivo, podemos crear una BBDD en memoria que existirá solo mientras la app esté activa. También podemos definir la forma en la que migrar los datos cuando cambie la versión, etcétera.

Más información sobre bases de datos, Room, sQLite:

https://developer.android.com/reference/kotlin/android/database/sqlite/SQLiteDatabase

https://developer.android.com/reference/kotlin/androidx/room/Room

https://developer.android.com/training/data-storage/sqlite

https://developer.android.com/training/data-storage/room

8.3 Bases de datos en la nube

En este módulo no trabajaremos con bases de datos en la nube pero de manera complementaria podéis encontrar aquí toda la información para ampliar conocimientos:

https://console.firebase.google.com/?pli=1

https://firebase.google.com/docs/database/android/start

9. BIBLIOGRAFÍA

i. Android. Programación Multimedia y de dispositivos móviles. Garceta.

ii. Programación Multimedia y Dispositivos Móviles. Editorial Síntesis.

iii. Android App Development. For Dummies.

iv. Beginning Android Programming with Android Studio. Wrox.

v. Java Programming for Android Developers. For Dummies.

vi. https://academiaandroid.com/

vii. https://developer.android.com/

viii. https://www.redhat.com

ix. https://desarrolloweb.com