101. Persistencia

Programación didáctica: •UT 9: Persistencia.

101.1 Formas de conseguir persistencia de datos en Android

- **Shared Preferences** (Preferencias compartidas): Ideal para guardar pequeñas cantidades de datos en forma de pares clave-valor. Es perfecto para preferencias de usuario, configuraciones, y datos sencillos.
- **Data Store** Desde la versión Jetpack 11 se recomienda usar DataStore mejor que Shared Preferences
- **File Storage**: Puedes almacenar datos directamente en el sistema de archivos del dispositivo, ya sea en *memoria interna* o *externa*. Es útil para guardar cosas como imágenes, archivos de audio, o documentos grandes.
- **SQLite Database**: Cuando necesitas almacenar una cantidad más grande de datos estructurados, SQLite es una gran opción. Es una base de datos relacional ligera que te permite realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) de manera eficiente.
- **Room Database**: Es una capa de abstracción sobre SQLite, parte del conjunto de librerías Android Jetpack. Room simplifica mucho el trabajo con bases de datos SQLite, manejando gran parte del código repetitivo y ofreciendo una forma más sencilla de interactuar con la base de datos.
- **Firebase Realtime Database y Firestore**: Si prefieres una solución basada en la nube, Firebase ofrece dos bases de datos potentes y fáciles de integrar. Permiten sincronizar datos en tiempo real entre diferentes dispositivos y manejar la persistencia de datos sin tener que preocuparte por la infraestructura.
- **DataStore**: Es una solución más moderna y robusta que Shared Preferences, también parte de Android Jetpack. DataStore usa coroutines y Flow para almacenar datos de manera asíncrona y transaccional.

101.2 Preferencias compartidas.

Es cmo un pequeño fichero con pares clave-valor que se conserva cuando se cierra la app y se recupera al volver a iniciar

Uso:
1. Guardamos las preferencias

```
// Obtén el SharedPreferences
val sharedPref = getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE)

// Usa un editor para guardar tus datos
with (sharedPref.edit()) {
    putString("nombreUsuario", "JetpackGenius")
```

```
    apply()
}
```

Donde `Context.MODE_PRIVATE` significa que solo tu app puede acceder a estas preferencias.
Nota: `with` es una forma de kotlin para "abrir" un objeto. Es lo mismo que ed.putString() seguido de ed.apply()

Para recuperar los datos:

```
// Obtén el SharedPreferences
val sharedPref = getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE)

// Usa getString para obtener el valor, el segundo parámetro es un valor por defecto
val nombreUsuario = sharedPref.getString("nombreUsuario", "UsuarioDesconocido")
```

Uso por ejemplo en una Activity en el método `onDestroy` de la actividad guardamos los datos y los recuperamos en `onCreate` cuando se produce una reconfiguración (rotación)

Se puede usar en:
* **Preferencias de Usuario**: Digamos que tienes una app con opciones de personalización, como un tema oscuro o claro. Puedes guardar la selección del usuario en SharedPreferences para que, cuando vuelva a abrir la app, se mantenga su elección.

- **Memoria interna del dispositivo**
- **Información de Autenticación Ligera**: Si necesitas recordar si un usuario ha iniciado sesión o guardar un token de autenticación simple, SharedPreferences es un buen lugar. Pero ten cuidado con la seguridad aquí, ¡no es el lugar para guardar datos sensibles como contraseñas!
- **Pequeñas configuraciones de la App**: Cosas como si el usuario activó notificaciones, configuró algún filtro específico en la visualización de una lista, o cualquier pequeña preferencia que mejore la experiencia del usuario sin necesitar una base de datos completa.
- **Datos de Estado de la Sesión**: Si tu app necesita recordar ciertos estados entre sesiones, como la última página visitada en un libro o una ubicación en un mapa, SharedPreferences puede ser un lugar práctico para guardar esa información.

101.3 Data Store

DataStore proporciona dos tipos de implementaciones: `Preferences DataStore` y `Proto DataStore`.

Preferences DataStore
* Uso: Ideal para almacenar pequeñas cantidades de datos, como ajustes y preferencias del usuario. * Funcionamiento: Almacena datos en pares clave-valor, similar a SharedPreferences, pero con mejoras. * Ventajas: Proporciona soporte para coroutines y Flow de Kotlin, lo que facilita la lectura y escritura de datos de manera asincrónica y segura en términos de hilos.

Proto DataStore * Uso: Adecuado para datos tipados más complejos. * Funcionamiento: Usa Protocol Buffers (formato de serialización binario ligero de Google) para almacenar datos de manera eficiente. * Ventajas: Permite definir esquemas de datos con tipos y estructuras claras, lo que facilita la validación y manipulación de datos complejos.

Características Clave de DataStore * Seguridad en Hilos: Realiza operaciones de lectura y escritura en un hilo seguro por defecto, eliminando errores comunes de concurrencia. * Transacciones Atómicas: Asegura la consistencia de los datos, incluso si ocurre una excepción o cierre inesperado de la app. * Uso de Flows: Permite a las apps observar cambios en los datos en tiempo real, facilitando la actualización de la UI en respuesta a cambios de datos.

101.4 Almacenamiento memoria interna del dispositivo. File Storage

En Android, el almacenamiento en la memoria interna se refiere a un espacio privado para tu app, donde puedes guardar archivos y datos. Estos archivos son accesibles solo por tu app, lo que añade una capa de seguridad ya que otras apps no pueden acceder a ellos. Aquí tienes algunos puntos clave:

Características:
* Espacio privado , sólo para tus aplicaciones.
* Limitado . Cada vez menos, pero es la memoria del teléfono compartido con el espacio de aplicaciones.
* Usamos `FileInputStream`, `FileOutputStream` para leer y escribir en memoria interna

Ejemplo, guardamos un texto:

```
fun guardarArchivo(context: Context, nombreArchivo: String, datos: String) {
    context.openFileOutput(nombreArchivo, Context.MODE_PRIVATE).use {
        it.write(datos.toByteArray())
    }
}

// Llamada a la función
guardarArchivo(context, "miArchivo.txt", "Hola Jetpack Genius!")
```

`use{}` es una función de extensión Kotlin que permite ejecutar un bloque de código con un recurso como FileOutput y cerrar el recurso al terminar el bloque
Es equivalente a este código:

```
val outputStream = FileOutputStream("miArchivo.txt")
try {
    outputStream.write("Hola Jetpack Genius!".toByteArray())
} catch (e: IOException) {
    // Manejo de excepciones
} finally {
    outputStream.close()
}
```

Lectura de archivos de memoria interna:

```
fun leerArchivo(context: Context, nombreArchivo: String): String {
    return context.openFileInput(nombreArchivo).bufferedReader().useLines { lines ->
        lines.fold("") { some, text ->
            "$some\n$text"
        }
    }
}

// Llamada a la función
val datos = leerArchivo(context, "miArchivo.txt")
```

Usamos `bufferedReader` y `useLines` para leer el archivo línea por línea y concatenarlas en un solo String.

101.5 Lectura de recursos de la aplicación

Cuando hablamos de "lectura de un recurso de la aplicación" en el desarrollo de Android, nos referimos a acceder y usar archivos y otros recursos que has incluido en tu proyecto de Android, como imágenes, archivos de texto, archivos XML para configuración, sonidos, etc.

Estos recursos suelen estar almacenados en la carpeta res de tu proyecto Android. Por ejemplo, podrías tener:

- Imágenes en res/drawable
- Archivos de sonido en res/raw
- Archivos XML para diseño o configuración en res/layout y res/values

Ejemplo Supongamos que tienes un archivo de texto (por ejemplo, mi_archivo.txt) en la carpeta res/raw. Aquí te muestro cómo podrías leerlo en tu aplicación:

```
fun leerRecursoTexto(context: Context): String {
    val inputStream = context.resources.openRawResource(R.raw.mi_archivo)
    return inputStream.bufferedReader().use(BufferedReader::readText)
}

// Llamada a la función
val miTexto = leerRecursoTexto(context)
```

Aquí Aquí, `R.raw.mi_archivo` se refiere al ID del recurso que Android genera automáticamente para cada archivo en tu carpeta res.

Utilizamos bufferedReader().use(BufferedReader::readText) para leer todo el contenido del archivo. Aquí, use es una función de Kotlin que automáticamente cierra el recurso una vez que termina de leerlo, evitando así fugas de memoria.

101.6 Almacenamiento en red

101.6.1 HTTP y HTTPS

Ejemplo: * Paso 1. Permisos

```
<uses-permission android:name="android.permission.INTERNET" />
```

* Paso 2. Elegir una librería HTTP

Kotlin no tiene una librería integrada para manejar HTTP, pero puedes usar librerías de terceros populares como:

- **Retrofit**: Una librería de tipo-safe para realizar llamadas HTTP. Es ampliamente utilizada por su eficiencia y facilidad de uso.
- **OkHttp**: Un cliente HTTP eficiente, que también puede ser utilizado con Retrofit.

- **Ktor**: Un framework de cliente y servidor web desarrollado por JetBrains, creadores de Kotlin

Seguimos el ejemplo con **Retrofit**

- Incluir dependencias

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

- Crear un interface de Retrofit

```
interface ApiService {
    @GET("endpoint")
    suspend fun getData(): Response<MyDataModel>
}
```

* Crear una instancia de Retrofit

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://tuapi.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

val apiService = retrofit.create(ApiService::class.java)
```

- Realizar la llamada

```
viewModelScope.launch {
    try {
        val response = apiService.getData()
        if (response.isSuccessful) {
            // Manejo de la respuesta
        } else {
            // Manejo de errores
        }
    } catch (e: Exception) {
        // Manejo de excepciones
    }
}
```

101.6.2 Sockets

Socket: Un socket es una interfaz de programación de red que permite la comunicación entre dos máquinas a través de una red.
TCP y UDP: Hay dos tipos principales de protocolos de sockets: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). TCP es orientado a la conexión y garantiza la entrega de los paquetes de datos, mientras que UDP es más simple y rápido, pero sin garantías de entrega.

101.6.3 WebSockets

En GPT

¿Fue útil esta página?