Programación multimedia y dispositivos móviles

Programación avanzada de aplicaciones móviles

Índice

Esquema 3

Material de estudio 4

5.1. Introducción y objetivos 4

5.2. Descubrimiento de servicios 4

5.3. Bases de datos y almacenamiento 10

5.4. Modelo de hilos 18

5.5. Búsqueda de dispositivos 22

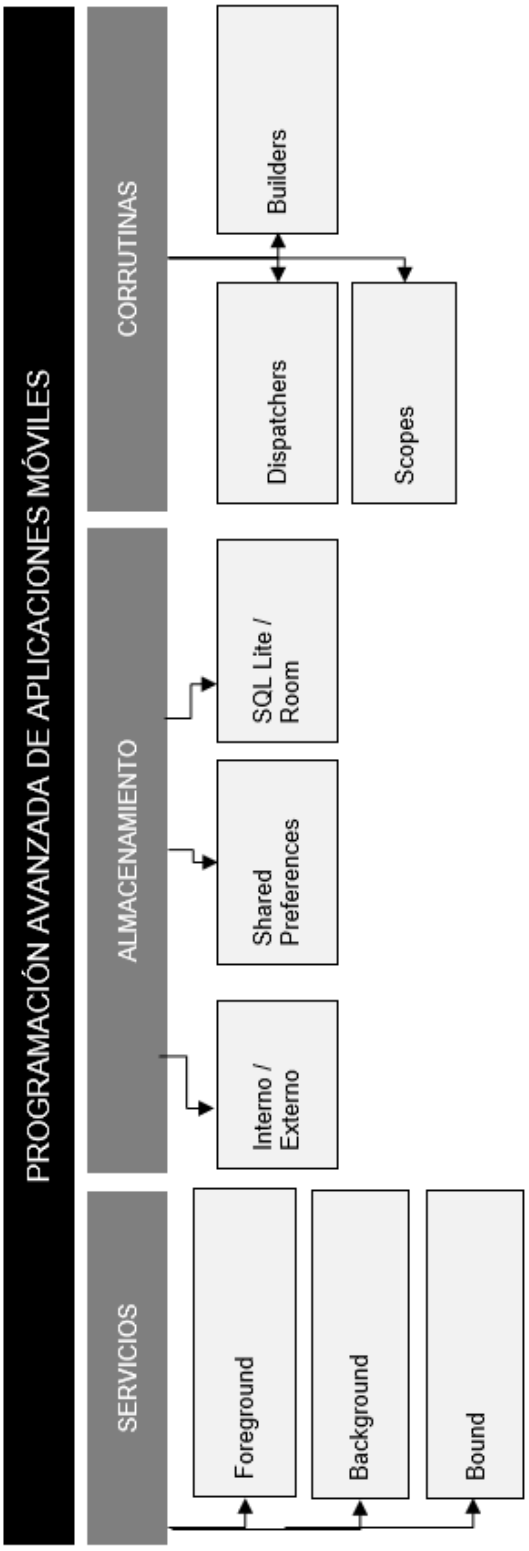
5.6. Búsqueda de servicios 24

A fondo 26

Entrenamientos 27

Test 29

Esquema


PROGRAMACIÓN AVANZADA DE APLICACIONES MÓVILES
SERVICIOS
Foreground
Background
Bound
ALMACENAMIENTO
Interno / Externo
Shared Preferences
SQL Lite / Room
CORRUTINAS
Dispatchers
Scopes
Builders

Material de estudio

5.1. Introducción y objetivos

Los objetivos que se persiguen en el siguiente tema son:

- Conocer los servicios, características, propiedades y tipos en Android Studio.
- Conocer los distintos tipos de almacenamiento disponible en Android Studio.
- Saber manejar SharedPreferences.
- Conocer SQL Lite y Room.
- Ser capaz de crear entidades, DAO y realizar consultas y operaciones.
- Conocer los protocolos y formas de conectar con otros dispositivos.

5.2. Descubrimiento de servicios

Un Service o servicio es un componente de una aplicación que realiza operaciones de larga ejecución en segundo plano. De este modo, la aplicación puede iniciar un servicio y continuar ejecutándose en segundo plano, aunque el usuario cambie a otra aplicación.

La misión de los servicios en Android es ejecutar operaciones de larga duración. Para ello, y en oposición a la misión de una Activity, un Service no depende de una interfaz de usuario.

Pero ¿qué operaciones de larga duración puede interesarnos realizar en un dispositivo móvil? Pensemos, por ejemplo, en una aplicación que permite acceder, gestionar y compartir ficheros en nuestra nube privada. Esta aplicación ejecutaría tareas que requieran descargar o subir ficheros a un servidor remoto.

La clase Activity no es apropiada, ya que está concebida para proporcionar una interfaz de usuario a la que está íntimamente ligada. Una instancia de Activity, de forma general, debería interrumpir o pausar cualquier trabajo que ejecute en segundo plano cuando el usuario la retire de la pantalla. Pero contamos con un Service para ayudarnos a implementar este tipo de trabajos.

La diferencia fundamental entre la Activity y el Service está en su ciclo de vida. Cuando queremos desarrollar un servicio, debemos escribir una clase que extienda a la clase Service o alguna de sus herederas. Al hacerlo, tendremos que sobrecargar algunos métodos específicos de Service, que serán llamados en nuestra aplicación en respuesta a las peticiones de los componentes que actúen como clientes del servicio. En dichos métodos, tendremos que iniciar las tareas a ejecutar. El framework se compromete, por medio de la especificación del ciclo de vida del Service, a no interrumpir la operación del servicio, aunque los clientes que inicien operaciones en él se retiren.

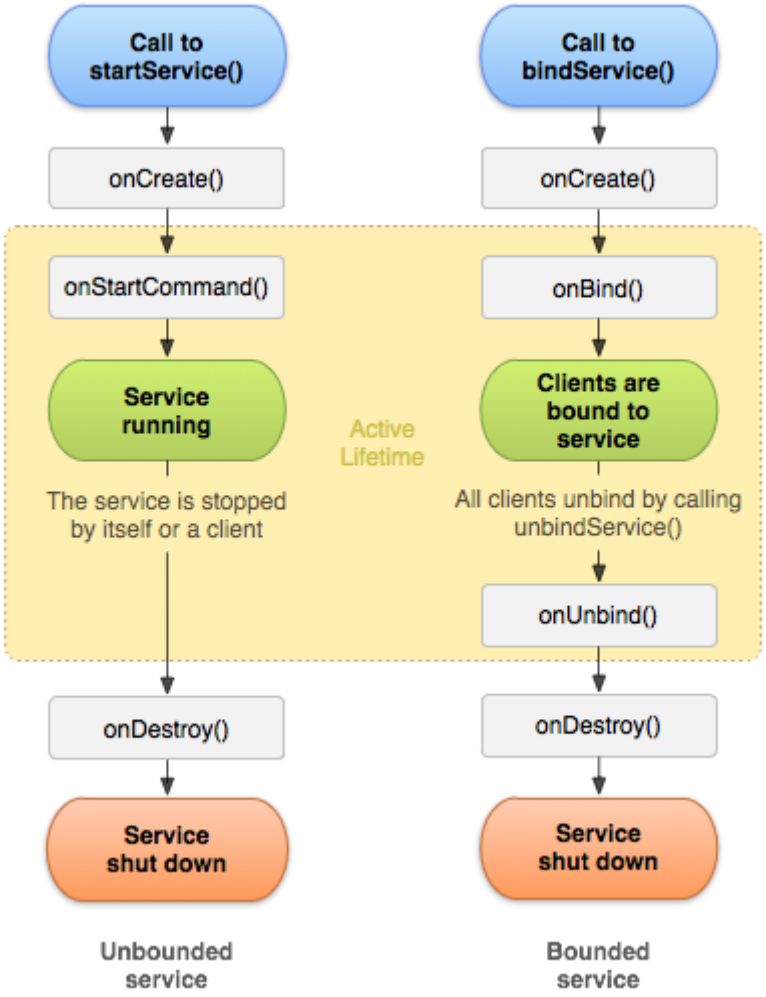


Figura 1: El ciclo de vida de un servicio. El diagrama de la izquierda muestra el ciclo de vida cuando se crea el servicio con startService() y en el diagrama de la derecha se muestra el ciclo de vida cuando se crea el servicio con bindService(). Fuente: https://developer.android.com/develop/background-work/services?hl=es-419

A pesar de que su misión es la ejecución de tareas en segundo plano, la gran mayoría de los métodos de entrada en los servicios son ejecutados por Android en el hilo principal de la aplicación.

Tipos de servicios

1. Foreground: Un servicio en primer plano realiza alguna operación que es perceptible para el usuario. Por ejemplo, una aplicación de audio utilizaría un servicio en primer plano para reproducir una pista de audio. Los servicios en primer plano deben mostrar una notificación. Los servicios en primer plano continúan ejecutándose incluso cuando el usuario no está interactuando con la aplicación. Cuando se utiliza un servicio en primer plano, se debe mostrar una

notificación para que los usuarios sean conscientes de que el servicio se está ejecutando. Esta notificación no puede cancelarse a menos que el servicio se detenga o se elimine del primer plano.

2. Background: Un servicio en segundo plano realiza una operación que el usuario no percibe directamente. Por ejemplo, si una aplicación utiliza un servicio para compactar su almacenamiento, normalmente se trataría de un servicio en segundo plano.

3. Bound: Un servicio está vinculado cuando un componente de la aplicación se vincula a él llamando a `bindService()`. Un servicio vinculado ofrece una interfaz cliente-servidor que permite a los componentes interactuar con el servicio, enviar peticiones, recibir resultados, e incluso hacerlo a través de procesos con comunicación entre procesos (IPC). Un servicio enlazado sólo se ejecuta mientras otro componente de la aplicación esté enlazado a él. Varios componentes pueden vincularse al servicio a la vez, pero cuando todos ellos se desvinculan, el servicio se destruye.

Creación de un Servicio

Para crear un servicio, debes crear una subclase de Service o utilizar una de sus subclases existentes. En su implementación, debe sobrescribir algunos métodos de llamada de retorno que manejan aspectos clave del ciclo de vida del servicio y proporcionan un mecanismo que permite a los componentes enlazarse al servicio, si procede. Éstos son los métodos de llamada de retorno más importantes que debes redefinir:

- onStartCommand(): El sistema invoca este método llamando a `startService()` cuando otro componente (como una actividad) solicita que se inicie el servicio. Cuando este método se ejecuta, el servicio se inicia y puede funcionar en segundo

plano indefinidamente. Si implementas esto, es tu responsabilidad detener el servicio cuando su trabajo haya terminado llamando a `stopSelf()` o `stopService()`. Si sólo quieres proporcionar binding, no necesitas implementar este método.

- onBind(): El sistema invoca este método llamando a `bindService()` cuando otro componente quiere enlazarse con el servicio (como para realizar RPC). En tu implementación de este método, debes proporcionar una interfaz que los clientes utilicen para comunicarse con el servicio devolviendo un `IBinder`. Siempre debes implementar este método; sin embargo, si no quieres permitir la vinculación, debes devolver null.
- onCreated(): El sistema invoca este método para realizar procedimientos de configuración únicos cuando el servicio se crea inicialmente (antes de llamar a `onStartCommand()` o a `onBind()`). Si el servicio ya se está ejecutando, no se llama a este método.
- onDestroy(): El sistema invoca este método cuando el servicio ya no se utiliza y se está destruyendo. Tu servicio debería implementar esto para limpiar cualquier recurso como hilos, oyentes registrados o receptores. Esta es la última llamada que recibe el servicio.

Si un componente inicia el servicio llamando a `startService()` (lo que resulta en una llamada a `onStartCommand()`), el servicio continúa ejecutándose hasta que se detiene a sí mismo con `stopSelf()` u otro componente lo detiene llamando a `stopService()`. Si un componente llama a `bindService()` para crear el servicio y no se llama a `onStartCommand()`, el servicio se ejecuta sólo mientras el componente esté vinculado a él. Una vez que el servicio se desvincula de todos sus clientes, el sistema lo destruye.

Declarar un servicio en el AndroidManifest

Los servicios deben aparecer en el AndroidManifest.XML. Deben declararse dentro del TAG application, independientemente del tipo de servicio que sea.

```
<application
      android:name="@string/app_name"
```

```
      android:theme="@android:style/Holo">
            <service name="com.myapp.id.MyService" label="@string/service_
label">
                  <intent-lter>
                        <action              android:name              =
                  "com.myapp.id.action.ACTION_OPEN_-
SERVICE"/>
                  </intent-lter>
            </service>
</applicaton>
```

Es importante destacar que el parámetro “name” debe ser la ruta en la que se encuentra el servicio dentro de la carpeta src de nuestro proyecto. Además, hemos establecido un intent-fillter, de manera que podemos iniciar el servicio de dos formas.

Cómo iniciar un servicio

Puedes iniciar un servicio desde una actividad u otro componente de la aplicación pasando un Intent a `startService()` o `startForegroundService()`. El sistema Android llama al método `onStartCommand()` del servicio y le pasa el Intent, que especifica qué servicio iniciar.

Por ejemplo, una actividad puede iniciar el servicio HelloService utilizando un intent explícito con startService(), como se muestra aquí:

```
      startService(Intent(this, HelloService::class.java))
```

El método `startService()` retorna inmediatamente, y el sistema Android llama al método `onStartCommand()` del servicio. Si el servicio aún no se está ejecutando, el sistema primero llama a `onCreate()` y luego llama a `onStartCommand()`.

Si el servicio no proporciona también binding, el intent que se entrega con `startService()` es el único modo de comunicación entre el componente de la aplicación y el servicio. Varias solicitudes para iniciar el servicio dan lugar a varias

llamadas a la función `onStartCommand()` del servicio. Sin embargo, solo se requiere una solicitud para detener el servicio (con `stopSelf()` o `stopService()`) para detenerlo.

Detener un servicio

Un servicio iniciado debe gestionar su propio ciclo de vida. Es decir, el sistema no detiene o destruye el servicio a menos que deba recuperar la memoria del sistema y el servicio continúe ejecutándose después de que regrese `onStartCommand()`. El servicio debe detenerse por sí mismo llamando a `stopSelf()`, u otro componente puede detenerlo llamando a `stopService()`. Una vez solicitada la detención con `stopSelf()` o `stopService()`, el sistema destruye el servicio lo antes posible.

5.3. Bases de datos y almacenamiento

Android ofrece varias opciones para que guardemos datos de aplicaciones persistentes. La solución que elijamos dependerá de nuestras necesidades específicas.

Opciones de almacenamiento

Las opciones de almacenamiento de datos son las siguientes:

- **Preferencias compartidas (SharedPreferences):** almacena datos primitivos privados en pares clave-valor.
- **Almacenamiento interno:** almacena datos privados en la memoria del dispositivo.
- **Almacenamiento externo:** almacena datos públicos en el almacenamiento externo compartido.
- **Bases de datos SQLite:** almacenan datos estructurados en una base de datos privada.
- **Conexión de red:** almacena datos en la web con su propio servidor de red.

- **Copia de seguridad en la nube:** copia de seguridad de la aplicación y los datos del usuario en la nube.
- **Proveedores de contenido:** almacenan los datos de forma privada y hacen que estén disponibles públicamente.
- **Base de datos en tiempo real de Firebase:** almacena y sincroniza datos con una base de datos en la nube NoSQL. Los datos se sincronizan en todos los clientes en tiempo real y permanecen disponibles cuando su aplicación se desconecta.

SharedPreferences

Si tiene una colección relativamente pequeña de clave-valor que desea guardar, puede utilizar las API de SharedPreferences. Debe ser información no comprometida puesto que Android generará un fichero XML donde almacenará toda esta información sin cifrar. Un objeto SharedPreferences apunta a un archivo que contiene pares clave-valor y proporciona métodos sencillos para leerlos y escribirlos. Cada archivo SharedPreferences es gestionado por el framework y puede ser privado o compartido.

Para crear un nuevo archivo de preferencias compartidas o acceder a uno existente se puede realizar llamando a uno de estos métodos:

- getSharedPreferences(): Utilízalo si necesitas varios archivos de preferencias compartidas identificados por su nombre, que especificas con el primer parámetro. Puede llamar esto desde cualquier Contexto en su aplicación.
- getPreferences(): Use esto desde una Actividad si necesita usar sólo un archivo de preferencias compartidas para la actividad. Debido a que esto recupera un archivo de preferencias compartidas predeterminado que pertenece a la actividad, no es necesario proporcionar un nombre.

Al nombrar tus archivos de preferencias compartidas, debes utilizar un nombre que identifique de forma única a tu aplicación. Una buena forma de hacerlo es anteponer al nombre del archivo el ID de tu aplicación. Por ejemplo: "`com.example.myapp.PREFERENCE_FILE_KEY`".

Para escribir en un archivo de preferencias compartidas, cree un `SharedPreferences.Editor` llamando a `edit()` en su SharedPreferences.
Pase las claves y valores que desea escribir con métodos como: `putInt()` y `putString()`. Luego llame a `apply()` o `commit()` para guardar los cambios. Por ejemplo:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
with (sharedPref.edit()) {
    putInt(getString(R.string.saved_high_score_key), newHighScore)
    apply()
}
```

Para recuperar valores de un archivo de preferencias compartido, llame a métodos como `getInt()` y `getString()`, proporcionando la clave para el valor que desea y, opcionalmente, un valor predeterminado para devolver si la clave no está presente. Por ejemplo:

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
val defaultValue =
resources.getInteger(R.integer.saved_high_score_default_key)
val highScore =
sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue)
```

Almacenamiento interno y externo

Android utiliza un sistema de archivos que es similar a los sistemas de archivos basados en disco en otras plataformas, como Linux. Las operaciones basadas en archivos son familiares para cualquiera que haya usado el uso de la E/S de archivos de Linux o el paquete java.io.

Todos los dispositivos Android tienen dos áreas de almacenamiento de archivos: almacenamiento "interno" y "externo". Estos nombres provienen de los primeros días de Android, cuando la mayoría de los dispositivos ofrecían memoria no volátil

incorporada (almacenamiento interno), más un medio de almacenamiento extraíble, como una tarjeta microSD (almacenamiento externo).

Hoy en día, algunos dispositivos dividen el espacio de almacenamiento permanente en particiones "internas" y "externas" por lo que, incluso sin un medio de almacenamiento extraíble, siempre hay dos espacios de almacenamiento y el comportamiento de la API es el mismo, sea el almacenamiento externo removible o no.

1. Almacenamiento interno

Siempre está disponible.

- Solo la aplicación puede acceder a los archivos. Específicamente, el directorio de almacenamiento interno de la aplicación se especifica por el nombre de su paquete en una ubicación especial del sistema de archivos de Android. Otras aplicaciones no pueden navegar por los directorios internos, y no tienen acceso de lectura o escritura, a menos que se establezca explícitamente que los archivos sean legibles o grabables.
- Cuando el usuario desinstala la aplicación, el sistema elimina todos los archivos de la aplicación del almacenamiento interno.
- El almacenamiento interno es mejor cuando se quiere estar seguro de que ni el usuario ni otras aplicaciones pueden acceder a los archivos.

2. Almacenamiento externo

- No siempre está disponible, porque el usuario puede montar el almacenamiento externo como almacenamiento USB y, en algunos casos, eliminarlo del dispositivo.
- Puede leer cualquier aplicación.
- Cuando el usuario desinstala la aplicación, el sistema elimina los archivos de la aplicación desde aquí solo si se guardan en el directorio desde getExternalFilesDir().
- El almacenamiento externo es el mejor lugar para los archivos que no requieren restricciones de acceso, que se desea compartir con otras aplicaciones o permitir que el usuario acceda con un ordenador.

Bases de datos locales

Las aplicaciones que manejan cantidades no triviales de datos estructurados pueden beneficiarse enormemente de la persistencia local de esos datos. El caso de uso más común es almacenar en caché fragmentos de datos relevantes para que, cuando el dispositivo no pueda acceder a la red, el usuario pueda seguir navegando por ese contenido mientras está desconectado.

La biblioteca de persistencia Room proporciona una capa de abstracción sobre SQLite para permitir un acceso fluido a la base de datos aprovechando toda la potencia de SQLite. En concreto, Room ofrece las siguientes ventajas:

- Verificación de las consultas SQL en tiempo de compilación.
- Anotaciones prácticas que minimizan el código repetitivo y propenso a errores.
- Rutas de migración de bases de datos simplificadas.

Para utilizar Room en tu aplicación, añade las siguientes dependencias al archivo build.gradle de tu aplicación y el plugin KSP.

```
dependencies {
    val room_version = "2.6.1"

    implementation("androidx.room:room-runtime:$room_version")

    // If this project uses any Kotlin source, use Kotlin Symbol Processing
(KSP)
    // See Add the KSP plugin to your project
    ksp("androidx.room:room-compiler:$room_version")


    // optional - Kotlin Extensions and Coroutines support for Room
    implementation("androidx.room:room-ktx:$room_version")

    // optional - RxJava2 support for Room
    implementation("androidx.room:room-rxjava2:$room_version")
```

```
    // optional - RxJava3 support for Room
    implementation("androidx.room:room-rxjava3:$room_version")

    // optional - Guava support for Room, including Optional and
ListenableFuture
    implementation("androidx.room:room-guava:$room_version")

    // optional - Test helpers
    testImplementation("androidx.room:room-testing:$room_version")

    // optional - Paging 3 Integration
    implementation("androidx.room:room-paging:$room_version")
}
```

Componentes principales.

Hay tres componentes principales en Room:

- La clase de base de datos que contiene la base de datos y sirve como punto de acceso principal para la conexión subyacente a los datos persistentes de tu app.
- Entidades de datos que representan tablas en la base de datos de tu app.
- Objetos de acceso a datos (DAOs) que proporcionan métodos que tu app puede utilizar para consultar, actualizar, insertar y borrar datos en la base de datos.

La clase de base de datos proporciona a tu app instancias de los DAOs asociados con esa base de datos. A su vez, la aplicación puede utilizar los DAO para recuperar datos de la base de datos como instancias de los objetos de entidad de datos asociados. La aplicación también puede utilizar las entidades de datos definidas para actualizar filas de las tablas correspondientes, o para crear nuevas filas para su inserción. La figura 2 ilustra la relación entre los distintos componentes de Room.

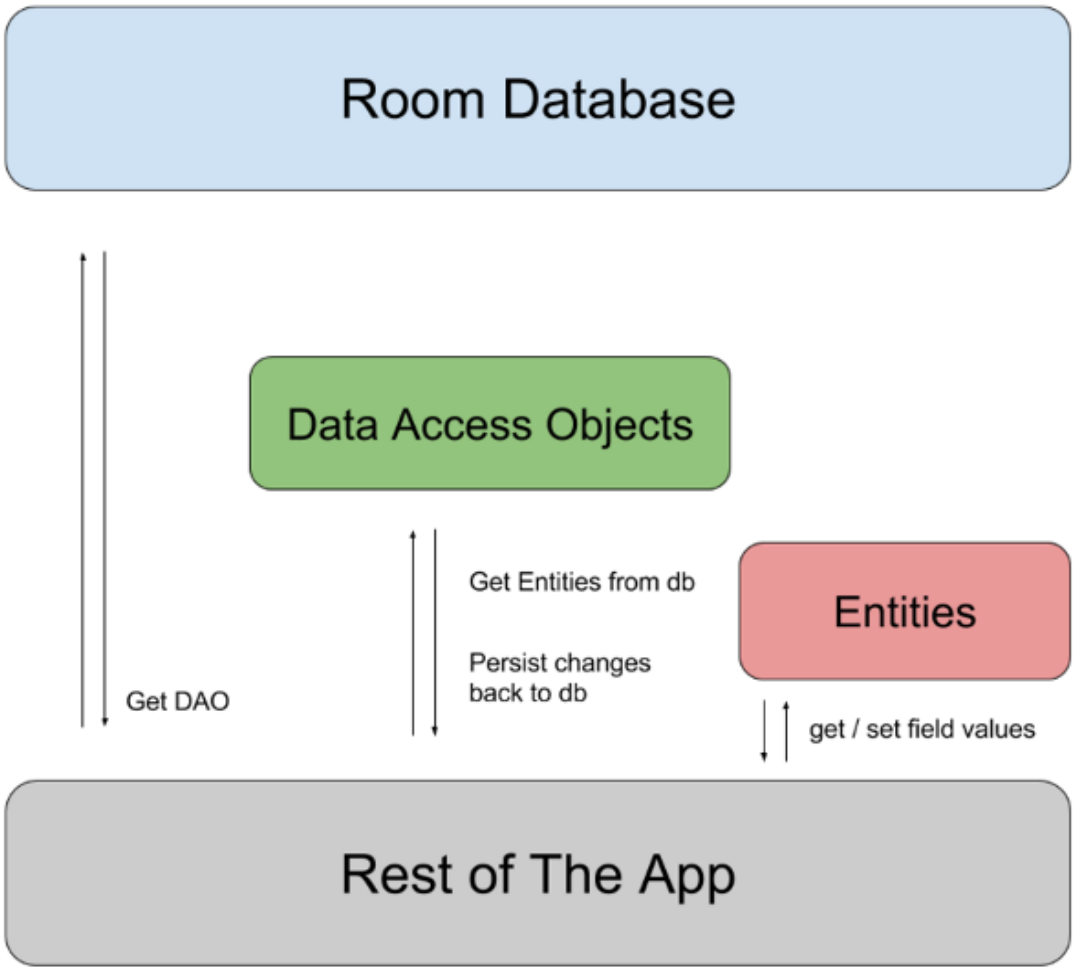


Figura 2: Diagrama de la arquitectura de la biblioteca Room.

Ejemplo de implementación de Room

Esta sección presenta un ejemplo de implementación de una base de datos Room con una única entidad de datos y una única DAO.

1. Data Entity

El siguiente código define una entidad de datos User. Cada instancia de User representa una fila en una tabla de user en la base de datos de la aplicación.

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String? )
```

2. Data access object (DAO)

El siguiente código define un DAO llamado UserDao. UserDao proporciona los métodos que el resto de la aplicación utiliza para interactuar con los datos de la tabla de usuarios.

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

3. Database

El siguiente código define una clase AppDatabase para contener la base de datos. AppDatabase define la configuración de la base de datos y sirve como punto de acceso principal de la aplicación a los datos persistentes. La clase de base de datos debe cumplir las siguientes condiciones:

- La clase debe estar anotada con una anotación @Database que incluya una matriz de entidades que enumere todas las entidades de datos asociadas con la base de datos.
- La clase debe ser una clase abstracta que extienda RoomDatabase.

- Para cada clase DAO que esté asociada con la base de datos, la clase de base de datos debe definir un método abstracto que tenga cero argumentos y devuelva una instancia de la clase DAO.

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

4. Utilización

Después de haber definido la entidad de datos, la DAO y el objeto de base de datos, puedes utilizar el siguiente código para crear una instancia de la base de datos:

```
val db = Room.databaseBuilder(
            applicationContext,
            AppDatabase::class.java, "database-name"
        ).build()
```

A continuación, puedes utilizar los métodos abstractos de la AppDatabase para obtener una instancia de la DAO. A su vez, puedes utilizar los métodos de la instancia DAO para interactuar con la base de datos:

```
val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```

5.4. Modelo de hilos

Hasta la entrada de Kotlin como lenguaje de programación en Android Studio (al usar Java) se utilizaban hilos o subprocesos de la forma clásica que permitía Java. Sin embargo, con la irrupción de Kotlin como nuevo lenguaje oficial para Android Studio todo el sistema de hilos o subprocesos se vio modificado por el uso de las corrutinas a partir de la versión 1.3 de Kotlin, convertiendose en la solución recomendada en Android Studio para la programación Asincrona. Una corrutina es un patrón de diseño

de simultaneidad que puedes usar en Android para simplificar el código que se ejecuta de forma asíncrona

Corrutinas

Una corrutina es una instancia de una computación suspendible. Es conceptualmente similar a un hilo, en el sentido de que toma un bloque de código para ejecutarse que trabaja concurrentemente con el resto del código. Sin embargo, una corrutina no está ligada a ningún hilo en particular. Puede suspender su ejecución en un subproceso y reanudarla en otro. Las corrutinas pueden considerarse como subprocesos ligeros, pero hay una serie de diferencias importantes que hacen que su uso en la vida real sea muy distinto del de los subprocesos.

Las corrutinas siguen un principio de concurrencia estructurada que significa que sólo se pueden lanzar nuevas corrutinas en un CoroutineScope específico que delimita el tiempo de vida de la corrutina. En una aplicación real, lanzarás muchas corrutinas. La concurrencia estructurada asegura que no se pierdan y no haya fugas. Un ámbito externo no puede completarse hasta que todas sus corrutinas hijas se completen. La concurrencia estructurada también garantiza que cualquier error en el código se notifique correctamente y nunca se pierda.

Para usar corrutinas en tu proyecto de Android, agrega la siguiente dependencia al archivo build.gradle de tu app:

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
android:1.3.9")
}
```

Dispatchers

Los dispatchers son un tipo de contextos de corrutina que especifican el hilo o hilos que pueden ser utilizados por la corrutina para ejecutar su código. Hay dispatchers que solo usan un hilo (como Main) y otros que definen un grupo de hilos que se optimizarán para ejecutar todas las corrutinas que reciben.

Es importante destacar que 1 hilo puede ejecutar muchas corrutinas, por lo que el sistema no creará 1 hilo por corrutina, sino que intentará reutilizar los que ya están vivos.

En Android Studio disponemos de 4 dispatchers principales:

- Default: Se usará cuando no se defina un dispatcher, pero también podemos configurarlo explícitamente. Este dispatcher se utiliza para ejecutar tareas que hacen un uso intensivo de la CPU, principalmente cálculos de la propia App, algoritmos, etc. Puede usar tantos subprocesos como cores tenga la CPU. Ya que estas son tareas intensivas, no tiene sentido tener más ejecuciones al mismo tiempo, porque la CPU estará ocupada.
- IO: Se utiliza para ejecutar operaciones de entrada/salida. En general, todas las tareas que bloquean el hilo mientras esperan la respuesta de otro sistema: peticiones al servidor, acceso a la base de datos, sistema de archivos, sensores, etc. Como no usan la CPU, se puede tener muchas en ejecución al mismo tiempo, por lo que el tamaño de este grupo de hilos es de 64. Las Apps de Android, lo que más hacen, es interactuar con el dispositivo y hacer peticiones de red, por lo que probablemente usarás este la mayoría del tiempo.
- Main: Este es un dispatcher especial que se incluye en las librerías de corrutinas relacionadas con interfaz de usuario. En particular, en Android, utilizará el hilo de UI.
- Unconfined: Si no te importa mucho qué hilo se utiliza, puedes usar este dispatcher. Es difícil predecir qué hilo se usará, así que no es muy recomendable su uso.

Builders de corrutinas

Para crear una nueva corrutina se debe de utilizar los builders. Disponemos de varios builders o incluso se puede crear uno propio personalizado. En general, con los que proporciona la API es suficiente.

- runBlocking: Este builder bloquea el hilo actual hasta que se terminen todas las tareas dentro de esa corrutina.

- launch: Este es el builder más usado. Se utiliza mucho porque es la forma más sencilla de crear corrutinas. A diferencia de runBlocking, no bloqueará el subproceso actual. launch devuelve un Job, que es otra clase que implementa CoroutineContext.
- async: permite ejecutar varias tareas en segundo plano en paralelo. No es una función de suspensión en sí misma, por lo que cuando se ejecuta async, el proceso en segundo plano se inicia, pero la siguiente línea se ejecuta de inmediato. async siempre debe llamarse dentro de otra corrutina, y devuelve un job especializado que se llama Deferred. Este objeto tiene una nueva función llamada await() que es la que bloquea. Se llamará a await() solo cuando se necesita el resultado. Si el resultado aún no está listo, la corrutina se suspende en ese punto. Si ya está disponible el resultado, simplemente lo devolverá y continuará. De esta manera, puedes ejecutar tantas tareas en segundo plano como necesites.

Scopes

Mediante los scopes se pueden definir los ámbitos de las corrutinas, en general trabajaremos con GlobalScope o implementaremos CoroutineScope.

- Global scope: Es un scope general que se puede usar para cualquier corrutina que deba continuar con la ejecución mientras la aplicación se está ejecutando. Por lo tanto, no deben estar atados a ningún componente específico que pueda ser destruido. Cuando se usa GlobalScope, siempre se debe de reflexionar si esta corrutina afecta a la aplicación completa y no solo a una pantalla o componente específico.
- CoroutineScope: Cualquier clase puede implementar esta interfaz y convertirse en un scope válido. Lo único que debe hacer es sobrescribir la propiedad coroutineContext, se debe de configurar el dispatcher y el Job.

A continuación, se muestra un ejemplo de creación de una corrutina.

```
// Importamos las librerías necesarias para trabajar con corrutinas
import kotlinx.coroutines.CoroutineStart
```

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

// Definimos nuestra función principal
fun main() = runBlocking {
    println("Antes de launch{}")
    // Creamos una corrutina que no se inicia automáticamente
    val job = launch(start = CoroutineStart.LAZY) {
        println("Lanzamos corrutina")
        delay(1000) // Retrasamos la corrutina por 1 segundo
    }
    job.start() // Iniciamos la corrutina manualmente
    println("Después de launch{}")
}
```

5.5. Búsqueda de dispositivos

Conectar dispositivos en las aplicaciones creadas con Android Studio implica implementar comunicación entre el dispositivo Android y otros dispositivos externos. Esto puede abarcar una amplia gama de tecnologías y protocolos, dependiendo del tipo de dispositivo que se desea conectar y el propósito de la conexión.

Dispositivos Bluetooth:

- Ejemplo: Auriculares, relojes inteligentes, impresoras, sensores, etc.
- Protocolo: Bluetooth clásico (para transferencias de datos) o Bluetooth Low Energy (BLE, para dispositivos de bajo consumo).

1. Bluetooth Clásico

- Permisos necesarios: Agrega estos permisos en el AndroidManifest.xml

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

- Conexión: Usa la clase `BluetoothAdapter` para gestionar el Bluetooth del dispositivo.
- Creación de sockets: Para enviar y recibir datos, utiliza un `BluetoothSocket`.

2. Bluetooth Low Energy (BLE)

- Permisos necesarios: Agrega estos permisos en el AndroidManifest.xml

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

- Conexión: Usa `BluetoothGatt` para interactuar con dispositivos BLE.

Dispositivos USB:

- Ejemplo: Dispositivos de almacenamiento, cámaras, teclados, etc.
- Protocolo: USB On-The-Go (OTG) para periféricos.
- Permisos necesarios: Agrega estos permisos en el AndroidManifest.xml

```
<uses-feature android:name="android.hardware.usb.host" />
```

- Conexión: Usa `UsbManager` para listar dispositivos conectados.
- Transferencia de datos: Usa `UsbDeviceConnection` y `UsbEndpoint` para enviar y recibir datos.

Dispositivos Wi-Fi:

- Ejemplo: Cámaras, dispositivos IoT, impresoras de red.
- Protocolo: Comunicación por TCP/IP, HTTP o WebSocket.
- Conexión local: Implementa comunicación usando sockets TCP o UDP:

```
Socket socket = new Socket("192.168.1.10", 8080);
OutputStream outputStream = socket.getOutputStream();
outputStream.write("Hola dispositivo".getBytes());
```

- HTTP o REST API: Usa bibliotecas como Retrofit o OkHttp para interactuar con dispositivos que soportan APIs RESTful.
- WebSocket: Para comunicación bidireccional en tiempo real.

Dispositivos IoT:

- Ejemplo: Sensores, dispositivos domésticos inteligentes.

- Protocolos: MQTT, HTTP, WebSocket.
- Conexión: MQTT: Usa una biblioteca como Paho MQTT para interactuar con dispositivos IoT.
- Protocolos específicos: Algunos dispositivos IoT usan sus propios protocolos, como ZigBee o Z-Wave. Estos requieren hardware adicional o integraciones específicas.

Dispositivos mediante NFC:

- Ejemplo: Tarjetas, etiquetas NFC, sistemas de acceso.
- Comunicación: Near Field Communication (NFC).
- Permisos necesarios: Agrega estos permisos en el AndroidManifest.xml

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

- Detectar etiquetas NFC: Usa NfcAdapter y PendingIntent para recibir eventos NFC.
- Detectar etiquetas NFC: Usa NfcAdapter y PendingIntent para recibir eventos NFC.

Sincronización con dispositivos externos

Usa Firebase o servicios como Google Play Services para sincronización en tiempo real entre dispositivos, como un reloj inteligente y un teléfono.

5.6. Búsqueda de servicios

La búsqueda o descubrimiento de servicios hace referencia al proceso de localizar servicios de manera dinámica en una red local. Este proceso es especialmente útil en aplicaciones que necesitan interactuar con otros dispositivos o servicios disponibles en la misma red, como en juegos multijugador locales, aplicaciones de transmisión de medios o herramientas de colaboración.

En Android, este proceso puede lograrse utilizando la API de Network Service Discovery (NSD), que permite que las aplicaciones descubran y se anuncien en redes locales mediante protocolos estándar como mDNS (Multicast DNS). La búsqueda de servicios es útil en aplicaciones que necesitan conectarse con dispositivos cercanos o

servicios locales, como juegos multijugador, aplicaciones de transmisión de medios o sistemas de IoT.

Pasos clave para buscar servicios en Android

1. Configurar el tipo de servicio a buscar Define el tipo de servicio que tu aplicación necesita localizar. Esto se especifica como un string en el formato `_<service-type>._<protocol>`. Por ejemplo:
    - "_http._tcp." para servicios HTTP.
    - "_ftp._tcp." para servicios FTP.
2. Crear un DiscoveryListener Este listener maneja eventos relacionados con la búsqueda de servicios, como cuando se encuentra un servicio, se pierde o finaliza la búsqueda.
3. Iniciar la búsqueda de servicios Utiliza el método discoverServices del objeto NsdManager para empezar a buscar servicios en la red.
4. Resolver los servicios encontrados Una vez que encuentres un servicio, puedes resolverlo para obtener más información, como la dirección IP y el puerto.

Aunque NSD no requiere permisos específicos, asegúrate de incluir los siguientes en el archivo AndroidManifest.xml si necesitas realizar comunicación de red:

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

A fondo

Servicios en Android Studio

https://developer.android.com/develop/background-work/services

Documentación en línea. (2024)

Documentación oficial sobre Servicios en Android Studio.

Corrutinas en Kotlin

https://www.youtube.com/watch?v=KqLtW8d8PXY Video en línea. (2024)

Video del canal DevExpert - Programación Android y Kotlin sobre el uso de corrutinas.

Corrutinas Android Studio

https://developer.android.com/kotlin/coroutines

Documentación en línea. (2024)

Documentación oficial sobre las corrutinas en Android Studio en Kotlin.

API Room Android Studio

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

Documentación en línea. (2024)

Documentación oficial sobre Room en Android Studio.

Entrenamientos

Entrenamiento 1

- Desarrollar una app Android Studio con Kotlin para practicas con el uso de corrutinas.

- Desarrollo paso a paso: https://www.youtube.com/watch?v=vQ0w4zAe68A&ab_channel=Programaci%C3%B3nAndroidbyAristiDevs

- Solución: https://github.com/ArisGuimera/Corrutinas-ZeroToHero

Entrenamiento 2

- Desarrollar una app Android Studio con Kotlin para manejar SharedPrefrerences.

- Desarrollo paso a paso y solución: https://cursokotlin.com/capitulo-16-persistencia-datos-shared-preferences/

Entrenamiento 3

- Desarrollar una App en Android usando Kotlin para crear y manejar unBackground Services.

- Desarrollo paso a paso y solución: https://dev.to/dragosb/servicios-en-android-1g9j#implementaci%C3%B3n

Entrenamiento 4

- Desarrollar una app Android usando Kotlin para usar el API Room y SQL Lite.

- Desarrollo paso a paso: https://cursokotlin.com/capitulo-17-persistencia-de-datos-con-room/

- Solución: https://github.com/ArisGuimera/MisNotas/tree/develop/app/src/main/java/com/cursokotlin/misnotas

Entrenamiento 5

- Desarrollar una app Android usando Kotlin para conectar y trasferir información a un dispositivo mediante blueetooth.

- Desarrollo paso a paso: https://developer.android.com/develop/connectivity/bluetooth/transfer-data?hl=es-419

- Solución: https://github.com/android/connectivity-samples/tree/master/BluetoothChat