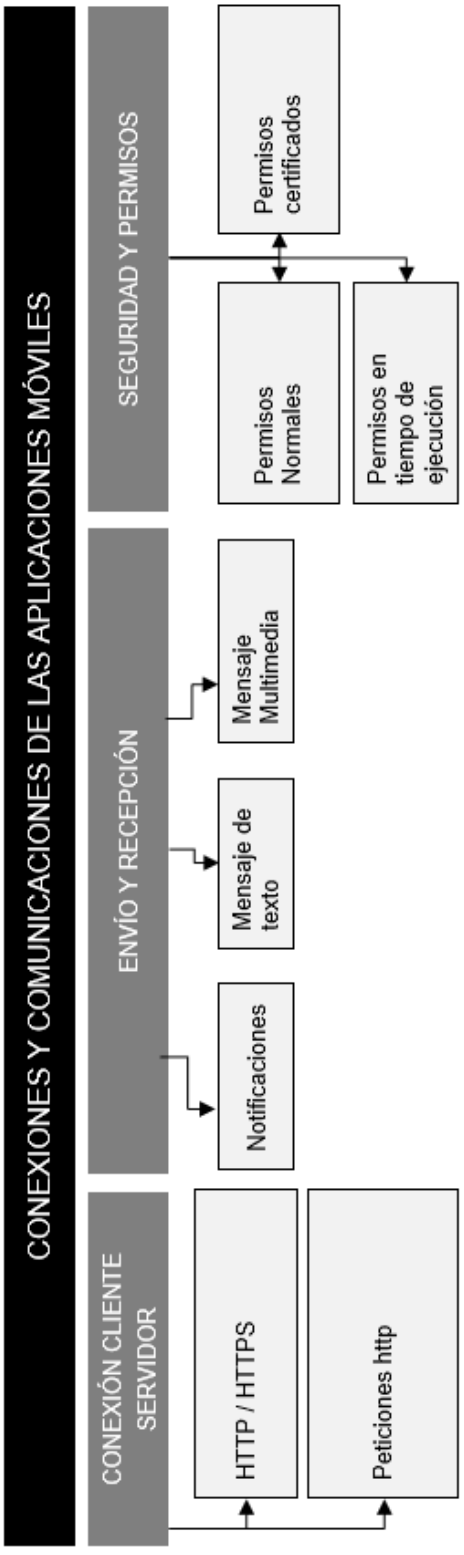
Conexiones y comunicaciones de las aplicaciones móviles

Índice

Esquema


CONEXIONES Y COMUNICACIONES DE LAS APLICACIONES MÓVILES
CONEXIÓN CLIENTE SERVIDOR
HTTP / HTTPS
Peticiones http
ENVÍO Y RECEPCIÓN
Notificaciones
Mensaje de texto
Mensaje Multimedia
SEGURIDAD Y PERMISOS
Permisos Normales
Permisos en tiempo de ejecución
Permisos certificados

Material de estudio

6.1. Introducción y objetivos

El desarrollo de aplicaciones móviles modernas requiere una integración eficiente entre el cliente y el servidor, utilizando tecnologías que faciliten la comunicación y el manejo de datos. Este tema aborda los fundamentos de la arquitectura cliente-servidor, donde el cliente solicita recursos y el servidor responde con los datos necesarios. Además, se profundiza en el uso de herramientas y bibliotecas como Retrofit para simplificar las conexiones HTTP, permitiendo una interacción fluida con APIs REST. También se analizan aspectos clave como el manejo de permisos, seguridad en la transmisión de datos y la recepción de mensajes, proporcionando un enfoque integral para desarrollar aplicaciones robustas y seguras en Android Studio. Los objetivos que se persiguen en el siguiente tema son:

- Conocer la arquitectura – cliente servidor.
- Conocer los distintos tipos de peticiones http.
- Conocer ser capaz de usar la librería Retrofit
- Conocer y ser capaz de mostrar mensajes de texto.
- Conocer los mensajes multimedia.
- Conocer y usar los permisos de las apps en Android Studio.
- Conocer los niveles de seguridad en Android Studio.

6.2. Establecimiento de la conexión. Cliente y Servidor

A la hora de desarrollar aplicaciones con conexión a la web, debemos tener muy en cuenta la distribución de elementos y la función de cada uno de ellos:

- El cliente: solicita la información.

- El servidor: responde a esa solicitud.

Un cliente es todo proceso que reclama servicios de otro, por lo que sus funciones principales las podemos resumir en:

- Administrar la interfaz e interactuar con el usuario.
- Procesar la lógica de la aplicación y hacer validaciones locales.
- Generar requerimientos de bases de datos y recibir/formatear los resultados del servidor.

La función principal de nuestro cliente en Android es solicitar al servidor los recursos elegidos por el usuario y mostrarlos. Estos recursos son documentos HTML, imágenes, ficheros multimedia, ficheros json y otros archivos, que se solicitan por medio del uso de las llamadas URI (Uniform Resource Identifier o identificador uniforme de recurso), consisten en una cadena de caracteres que identifican los recursos de una red.

Una API REST (Representational State Transfer) es un tipo de interfaz de programación de aplicaciones (API) que se basa en los principios de diseño REST, un estilo arquitectónico para sistemas distribuidos como la web. Las API REST son ampliamente utilizadas para permitir la comunicación entre diferentes sistemas, independientemente del lenguaje de programación o plataforma utilizada. Las API REST utilizan los métodos HTTP para interactuar con los recursos

En términos generales, existen cuatro tipos distintos de peticiones HTTP que realizamos al servidor:

- GET: Es el método de petición más sencillo, utilizado exclusivamente para obtener información del servidor. Si es necesario proporcionar un parámetro a la petición, este se incluye en la URL. Por ejemplo, si necesitamos realizar una petición que depende de un identificador (como la identificación de un usuario), la URL se formaría de la siguiente manera: https://ejemplo.com/informacion/1, donde 1 es

el parámetro que estamos pasando. Sin embargo, este enfoque es poco seguro para transmitir información sensible, ya que los datos quedan expuestos en la URL.

- POST: Similar al método GET, pero los parámetros no se envían a través de la URL, sino en el cuerpo de la petición. Esto lo hace más seguro para enviar información confidencial.

- PUT: Se utiliza generalmente para crear o actualizar una entidad. Por ejemplo, en un servicio que interactúa con una base de datos, este método se emplearía para crear un nuevo usuario o modificar uno existente.
- DELETE: Es el último de los cuatro métodos y se utiliza para eliminar registros de la base de datos.

La información generalmente se presenta en dos formatos principales: XML y JSON. Sin embargo, dado que JSON es el formato más común actualmente, centraremos nuestra atención exclusivamente en este último.

JSON, cuyo acrónimo corresponde a JavaScript Object Notation, es un formato de texto simple y ampliamente utilizado como estándar para la transferencia de información entre plataformas. Su estructura altamente legible facilita la comprensión del contenido de manera clara y eficiente.

Características Principales de una API REST

1. Arquitectura Basada en Recursos: Los recursos son los elementos principales que maneja una API REST y se representan como URLs únicas. Por ejemplo, un recurso podría ser "usuarios" accesible a través de https://api.ejemplo.com/usuarios.
2. Operaciones Estándar del Protocolo HTTP: Utiliza los métodos HTTP para interactuar con los recursos.
3. Los datos generalmente se intercambian en formatos ligeros como JSON o XML, aunque JSON es el formato más común por su simplicidad y compatibilidad.
4. Estateless (Sin Estado): Las API REST son stateless, lo que significa que cada solicitud desde un cliente al servidor debe contener toda la información necesaria

para procesarla. El servidor no almacena información de estado sobre el cliente entre las solicitudes.

5. Cacheabilidad: Los datos de las respuestas pueden ser marcados como cacheables o no, para mejorar el rendimiento y reducir la carga en el servidor.
6. Interfaz Uniforme: Siguiendo el principio de REST, las interacciones entre cliente y servidor deben ser coherentes y predecibles. Por ejemplo, todas las URL deben tener una estructura consistente.
7. Separación Cliente-Servidor: Existe una clara separación entre el cliente (que realiza las solicitudes) y el servidor (que maneja los recursos). Esto permite una evolución independiente de ambos componentes.

6.3. Manejo de conexiones HTTP y HTTPS

El manejo de conexiones HTTP y HTTPS en Android Studio es una parte esencial del desarrollo de aplicaciones que interactúan con servicios web. Android proporciona varias herramientas y librerías para trabajar con estas conexiones. Vamos a describir las librerías más importantes.

1. HttpURLConnection: Es la API nativa de Java para realizar solicitudes HTTP en Android. Es eficiente, pero requiere un manejo manual y detallado. Está integrada directamente en el SDK de Android, pero ofrece control de bajo nivel sobre las solicitudes.

```
URL url = new URL("http://ejemplo.com/api");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("GET");
connection.setConnectTimeout(5000); // Tiempo de espera
connection.setReadTimeout(5000);

int responseCode = connection.getResponseCode();
if (responseCode == HttpURLConnection.HTTP_OK) {
    InputStream inputStream = connection.getInputStream();
    // Leer el flujo de datos
}
```

```
connection.disconnect();
```

2. OkHttp es una librería popular para manejar solicitudes HTTP/HTTPS. Es más fácil de usar que HttpURLConnection y maneja conexiones de manera más eficiente. Es relativamente fácil de implementar y realiza el manejo automático de redirecciones y caché. Además, incluye soporte para interceptores.

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
        .url("http://ejemplo.com/api")
        .build();

client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        e.printStackTrace();
    }

    @Override
    public void onResponse(Call call, Response response) throws
IOException {
        if (response.isSuccessful()) {
            String responseData = response.body().string();
            // Procesar datos
        }
    }
});
```

3. Retrofit, basada en OkHttp, simplifica enormemente la comunicación con APIs RESTful. Usa anotaciones para definir solicitudes y facilita la conversión de datos JSON a objetos Java con Gson o Moshi. Además, incluye soporte para operaciones asíncronas.

Actualmente Retrofit es la librería más utilizada y es la que vamos a estudiar en profundidad.

Retrofit

Retrofit2 es una potente y flexible librería de cliente HTTP desarrollada por Square para Android y Java. Su propósito principal es simplificar las interacciones con APIs RESTful, proporcionando una forma fácil y estructurada de realizar solicitudes HTTP y manejar las respuestas. Es ampliamente utilizada en el desarrollo de aplicaciones Android debido a su facilidad de uso y extensibilidad.

Características Principales de Retrofit2:

- Simplicidad y Modularidad: Retrofit abstrae las solicitudes HTTP, lo que permite trabajar con APIs RESTful mediante interfaces y anotaciones.
- Compatibilidad con Formatos de Datos: Compatible con múltiples formatos de datos como JSON y XML, y puede usar convertidores (converters) para manejarlos, como Gson, Moshi o Jackson.
- Soporte para Operaciones Asíncronas: Integra de forma nativa la ejecución en segundo plano y callbacks para manejar respuestas asíncronas.
- Integración con RxJava y Coroutines: Se puede integrar con RxJava o Kotlin Coroutines para manejar flujos de datos reactivos y asincronía.
- Configuración Flexible: Admite autenticación, encabezados personalizados, manejo de errores, reintentos y más.
- Soporte para Multipart y Form Data: Permite manejar subidas de archivos y formularios.

Pasos para usar Retrofit2 en Android Studio

1. Agregar Retrofit2 a Tu Proyecto: Agrega las dependencias necesarias en el archivo build.gradle:

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

- retrofit: La librería principal.
- converter-gson: Convertidor para manejar respuestas JSON usando Gson.

2. Definir el Cliente Retrofit

Crea una instancia de Retrofit configurando la URL base y el convertidor para el formato de datos (por ejemplo, JSON):

```
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

val retrofit = Retrofit.Builder()
    .baseUrl("https://api.ejemplo.com/") // URL base de la API
    .addConverterFactory(GsonConverterFactory.create()) // Convertidor JSON
    .build()
```

3. Crear una Interfaz para la API

Define los endpoints de la API utilizando una interfaz. Usa anotaciones como @GET, @POST, @PUT, @DELETE para especificar los métodos HTTP. Ejemplo de Interfaz:

```
import retrofit2.Call
import retrofit2.http.GET
import retrofit2.http.Path

interface ApiService {
    @GET("usuarios")
    fun getUsuarios(): Call<List<Usuario>>

    @GET("usuarios/{id}")
    fun getUsuarioById(@Path("id") id: Int): Call<Usuario>
}
```

- @GET, @POST, etc.: Especifican el método HTTP.
- @Path, @Query, etc.: Mapean parámetros en la URL o en la solicitud.

4. Crear el Modelo de Datos

Define las clases que representarán las respuestas de la API. Por ejemplo, para manejar una lista de usuarios en formato JSON:

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

```
        val nombre: String,
        val correo: String
    )
```

5. Realizar una Solicitud

Crea una instancia de la interfaz de la API y realiza la solicitud:

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

val call = apiService.getUsuarios()
call.enqueue(object : retrofit2.Callback<List<Usuario>> {
    override fun onResponse(call: Call<List<Usuario>>, response:
retrofit2.Response<List<Usuario>>) {
        if (response.isSuccessful) {
            val usuarios = response.body()
            // Manejar la lista de usuarios
        } else {
            // Manejar errores
        }
    }

    override fun onFailure(call: Call<List<Usuario>>, t: Throwable) {
        // Manejar fallos
    }
})
```

6. Usar Retrofit con Kotlin Coroutines

Para una integración más moderna y fluida, puedes usar Retrofit con Kotlin Coroutines. Se añaden las dependencias al fichero gradle:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.4'
```

Ejemplo de Interfaz para Coroutines:

```
import retrofit2.http.GET

    interface ApiService {
        @GET("usuarios")
        suspend fun getUsuarios(): List<Usuario>
    }
```

Realizar la Solicitud con Coroutines:

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

CoroutineScope(Dispatchers.IO).launch {
    try {
        val usuarios = apiService.getUsuarios()
        // Manejar los usuarios en el hilo principal
    } catch (e: Exception) {
        // Manejar errores
    }
}
```

6.4. Envío y recepción de mensajes de texto

En Android Studio, enviar y recibir mensajes de texto (SMS) implica trabajar con permisos, APIs específicas para el envío de SMS y configuraciones para la recepción de mensajes. En este apartado vamos a describir los pasos mínimos necesarios para poder realizar tanto el envío como la recepción de mensajes de texto.

Envío de Mensajes de Texto

Para enviar mensajes SMS, puedes usar la clase SmsManager. Aquí tienes los pasos básicos:

1. Configurar Permisos

En el archivo AndroidManifest.xml, se necesita declarar los permisos requeridos:

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

2. Enviar un Mensaje

Se debe de usar la clase SmsManager para enviar mensajes SMS. Un ejemplo básico en Kotlin sería:

```
import android.telephony.SmsManager
import android.widget.Toast
```

```
fun sendSMS(phoneNumber: String, message: String) {
    try {
        val smsManager = SmsManager.getDefault()
        smsManager.sendTextMessage(phoneNumber, null, message, null, null)
        Toast.makeText(this, "Mensaje enviado", Toast.LENGTH_SHORT).show()
    } catch (e: Exception) {
        e.printStackTrace()
        Toast.makeText(this, "Error al enviar mensaje",
Toast.LENGTH_SHORT).show()
    }
}
```

3. Pedir Permiso en Tiempo de Ejecución

En dispositivos con Android 6.0 (API 23) o superior, los permisos sensibles como SEND_SMS requieren ser solicitados en tiempo de ejecución:

```
if (ActivityCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS)
!= PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this,
arrayOf(Manifest.permission.SEND_SMS), REQUEST_CODE_SEND_SMS)
}
```

Recepción de Mensajes de Texto

Para recibir mensajes SMS, necesitas crear un receptor de difusión (BroadcastReceiver) que escuche los mensajes entrantes.

1. Configurar Permisos y Registro

En el archivo AndroidManifest.xml, se deben de añadir los siguientes permisos:

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

<application>
    <receiver android:name=".SmsReceiver">
        <intent-filter>
            <action android:name="android.provider.Telephony.SMS_RECEIVED"
/>
        </intent-filter>
    </receiver>
```

```
</application>
```

2. Crear el BroadcastReceiver

```
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.os.Bundle
import android.telephony.SmsMessage
import android.widget.Toast

class SmsReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        val bundle: Bundle? = intent.extras
        try {
            if (bundle != null) {
                val pdus = bundle["pdus"] as Array<*>?
                if (pdus != null) {
                    for (pdu in pdus) {
                        val format = bundle.getString("format")
                        val message = SmsMessage.createFromPdu(pdu as
ByteArray, format)
                        val sender = message.originatingAddress
                        val content = message.messageBody

                        Toast.makeText(context, "SMS recibido de $sender:
$content", Toast.LENGTH_LONG).show()
                    }
                }
            }
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

6.5. Envío y recepción de mensajes de multimedia

Qué es Toast

Un toast es un mensaje que se muestra en pantalla al usuario durante unos segundos para luego desaparecer automáticamente. Aunque son personalizables, aparecen por defecto en la parte inferior de la pantalla, sobre un rectángulo gris translúcido. Estas notificaciones son ideales para mostrar mensajes rápidos y sencillos al usuario Su uso es muy sencillo, toda la funcionalidad está en la clase Toast.

Esta clase dispone de un método estático makeText() al que deberemos pasar como parámetro el contexto de la actividad, el texto a mostrar y la duración del mensaje. La duración puede tomar los valores LENGTH_LONG (3.5 segundos aprox.) o LENGTH_SHORT (2 segundos aprox.), dependiendo del tiempo que queramos que la notificación aparezca en pantalla.

Tras obtener una referencia al objeto Toast a través de este método, ya solo nos quedaría mostrarlo en pantalla mediante el método show().

```
Toast.makeText(this, "Bienvenido a la app", Toast.LENGTH_SHORT).show()
```

Se puede personalizar el diseño de un Toast para que se ajuste a las necesidades de tu aplicación. De tal forma, para cambiar la posición se debe de usar setGravity()

```
val toast = Toast.makeText(this, "Toast en la parte superior",
Toast.LENGTH_LONG)
toast.setGravity(Gravity.TOP, 0, 200) // Gravity.TOP y desplazamiento en Y
toast.show()
```

Toast con vista personalizada

Se puede usar un diseño XML personalizado para mostrar un Toast único.

1. Diseño XML (res/layout/custom_toast.xml):

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:padding="10dp"
    android:background="#FFBB86FC"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <ImageView
        android:src="@drawable/ic_launcher_foreground"
        android:layout_width="40dp"
        android:layout_height="40dp"
        android:layout_marginEnd="10dp" />

    <TextView
        android:text="Este es un Toast personalizado"
        android:textColor="#FFFFFF"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

2. Código Kotlin para Mostrar el Toast Personalizado:

```
val layoutInflater = layoutInflater
val customView = layoutInflater.inflate(R.layout.custom_toast, null)

val toast = Toast(applicationContext)
toast.duration = Toast.LENGTH_LONG
toast.view = customView
toast.show()
```

Cuando usar Toast

- Mensajes Rápidos: Mostrar mensajes informativos, como "Guardado con éxito" o "Error de conexión".
- Sin Necesidad de Interacción: Situaciones en las que el usuario no necesita responder al mensaje.
- Depuración: Mostrar valores o información durante el desarrollo (aunque Logcat es más apropiado para esto).

Sin embargo, los Toast conllevan unas ciertas limitaciones que deben de tenerse en cuenta antes de usarlos:

- Sin Confirmación de Vista: El usuario puede no notar el mensaje si desaparece rápidamente.
- No Persistente: El mensaje no puede ser interactivo ni persistir en pantalla.
- No Adecuado para Mensajes Críticos: Usa diálogos (AlertDialog) o notificaciones para mensajes importantes.

Diálogos

Se pueden utilizar los diálogos de Android con distintos fines:

- Mostrar un mensaje.
- Pedir una confirmación rápida.
- Solicitar al usuario una elección entre varias alternativas.

Para crear un diálogo tenemos que crear una nueva clase que herede de DialogFragment y sobrescribir uno de sus métodos onCreateDialog(), que será el encargado de construir el diálogo. Existen distintos tipos de diálogos en Android:

- Diálogo de alerta.
- Diálogo de confirmación.
- Diálogo de selección.
- Diálogos personalizados

1. Diálogo de Alerta

Este tipo de diálogo muestra un mensaje sencillo y un botón de OK para confirmar su lectura. Lo construiremos mediante la clase AlertDialog. Su utilización es muy sencilla; bastará con crear un objeto de tipo AlertDialog.Builder y establecer las propiedades del diálogo mediante sus métodos correspondientes:
título [setTitle()], mensaje [setMessage()], y el texto y comportamiento del botón [setPositiveButton()].

2. Diálogo de confirmación

En un diálogo de confirmación solicitamos al usuario que nos confirme una determinada acción, con respuestas del tipo Sí/No.

Lo haremos igual que para el ejemplo de alerta, pero en esta ocasión añadiremos dos botones, uno para la respuesta a rmativa (setPositiveButton()), y otro para la respuesta negativa (setNegativeButton()).

3. Diálogo de Selección

Podemos utilizar los diálogos de selección para mostrar una lista de opciones entre las que el usuario puede elegir. Para ello, utilizaremos la clase AlertDialog, pero esta vez indicaremos la lista de opciones a mostrar (mediante el método setItems()) y proporcionaremos la implementación del evento onClick() sobre dicha lista (mediante un listener de tipo DialogInterface.OnClickListener), evento en el que realizaremos las acciones oportunas según la opción elegida.

4. Diálogos personalizados

También podemos establecer completamente el aspecto de un cuadro de diálogo. Para ello, definiremos un layout XML con los elementos a mostrar en el diálogo. En este caso, crearemos un layout llamado dialog_personal.xml en la carpeta res/layout. Contendrá, por ejemplo, una imagen a la izquierda y dos líneas de texto

a la derecha. Por su parte, en el método onCreateDialog() correspondiente utilizaremos el método setView() del builder para asociarle nuestro layout personalizado. Además, podremos incluir botones, tal como vimos para los diálogos de alerta o confirmación.

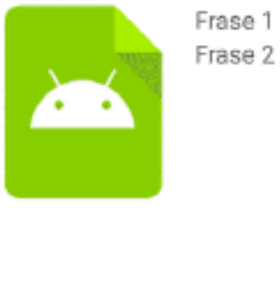


Figura 1: diálogo personalizado de ejemplo. Contenidos UNIR.

6.6. Seguridad y permisos

La seguridad y los permisos en Android Studio son esenciales para proteger los datos del usuario, la integridad de las aplicaciones y el sistema operativo Android. La gestión de permisos permite que las aplicaciones accedan a recursos o datos sensibles de manera controlada y con el consentimiento del usuario. Al seguir buenas prácticas, puedes minimizar riesgos y garantizar que los usuarios confíen en tu aplicación.

Tipos de permisos en Android Studio

Los permisos en Android se dividen en varias categorías según su sensibilidad y cómo se gestionan:

1. Permisos Normales

- Permiten el acceso a recursos que no comprometen la privacidad del usuario ni afectan a otros recursos.
- Se otorgan automáticamente al instalar la aplicación.
- Ejemplo:
    - INTERNET (para acceso a internet).
    - ACCESS_NETWORK_STATE (para consultar el estado de la red).

2. Permisos Sensibles

- Involucran datos sensibles o acciones que afectan al usuario directamente.
- Requieren que el usuario los conceda explícitamente en tiempo de ejecución.
- Ejemplo:
    - READ_CONTACTS (para acceder a los contactos).
    - ACCESS_FINE_LOCATION (para acceder a la ubicación precisa).
    - SEND_SMS (para enviar mensajes SMS).

3. Permisos Especiales

- Son permisos críticos que requieren configuraciones adicionales, como ser aprobados manualmente por el usuario en la configuración del dispositivo.
- Ejemplo:
    - SYSTEM_ALERT_WINDOW (para superposiciones en pantalla).
    - WRITE_SETTINGS (para cambiar configuraciones del sistema).

Gestión de Permisos en Android Studio

Desde Android 6.0 (API 23), se introdujeron permisos en tiempo de ejecución, lo que significa que los usuarios deben otorgar permisos sensibles mientras la aplicación se está ejecutando.

1. Declarar Permisos en AndroidManifest.xml

Todos los permisos que necesita tu aplicación deben declararse en este archivo.

Ejemplo:

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

2. Solicitar Permisos en Tiempo de Ejecución

Para permisos sensibles, primero se debe de verificar si el permiso está concedido antes de realizar una acción.

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA) !=
PackageManager.PERMISSION_GRANTED) {
    // El permiso no está concedido
}
```

Si el permiso no está concedido se debe de solicitar:

```
ActivityCompat.requestPermissions(
    this,
    arrayOf(Manifest.permission.CAMERA),
    REQUEST_CODE_CAMERA
)
```

Finalmente, para manejar la respuesta del usuario se debe de sobreescribir el método onRequestPermissionsResult

```
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<out String>,
    grantResults: IntArray
) {
    if (requestCode == REQUEST_CODE_CAMERA) {
        if    ((grantResults.isNotEmpty()    &&    grantResults[0]    ==
PackageManager.PERMISSION_GRANTED)) {
            // Permiso concedido
        } else {
            // Permiso denegado
        }
    }
}
```

3. Permisos Especiales

Algunos permisos requieren configuraciones adicionales:

- Mostrar Superposición (SYSTEM_ALERT_WINDOW): Se solicita al usuario dirigirlo manualmente a las configuraciones.

```
val intent = Intent(Settings.ACTION_MANAGE_OVERLAY_PERMISSION)
startActivity(intent)
```

- Modificar Configuraciones del Sistema (WRITE_SETTINGS): También requiere enviar al usuario a las configuraciones del dispositivo:

```
if (!Settings.System.canWrite(this)) {
    val intent = Intent(Settings.ACTION_MANAGE_WRITE_SETTINGS)
    startActivity(intent)
}
```

Para realizar un uso correcto y responsable de los permisos se debe de seguir este protocolo de buenas prácticas:

- Solicitar Permisos Cuando Sean Necesarios:

- No solicites todos los permisos al inicio. Hazlo en el momento en que se necesitan.
- Por ejemplo, solicita acceso a la cámara cuando el usuario intente tomar una foto.

- Explicar la Razón del Permiso:
  - Usa shouldShowRequestPermissionRationale() para explicar al usuario por qué necesitas el permiso.
  - Muestra un diálogo o mensaje antes de solicitar el permiso.
- Diseño Compatible con la Privacidad:
  - Solicita solo los permisos que realmente necesitas.
  - Asegúrate de que tu aplicación respeta la privacidad del usuario.
- Manejar la Denegación de Permisos:
  - Prepara una experiencia alternativa si el usuario deniega un permiso.
  - Ofrece una opción para ir a la configuración y conceder el permiso más tarde.

Seguridad en Android Studio

Además de gestionar permisos, Android ofrece varias capas de seguridad:

1. Sandboxing de Aplicaciones

Cada aplicación en Android se ejecuta en su propio entorno aislado, lo que significa que no puede acceder a los datos o recursos de otras aplicaciones sin permisos explícitos.

2. Criptografía y Almacenamiento Seguro

- Usa Keystore para almacenar claves de cifrado.
- Usa SharedPreferences cifradas (desde Android 11 o con bibliotecas externas).

3. Comunicación Segura

- Usa conexiones HTTPS seguras (SSL/TLS) para la comunicación de red.
- Evita almacenar contraseñas o datos sensibles en texto plano.

4. Reforzar Seguridad con ProGuard

ProGuard ofusca tu código y dificulta la ingeniería inversa. Configúralo en proguard-rules.pro:

```
-keep class com.example.** { *; }
-dontwarn com.example.**
```

5. Autenticación y Autorización

Usa métodos seguros de autenticación, como OAuth2, para integraciones con servicios externos. Implementa autenticación biométrica cuando sea posible.

A fondo

Librería Retrofit

https://square.github.io/retrofit/

Documentación en línea. (2024)

Documentación oficial sobre la librería Retrofit.

Permisos en Android Studio

https://developer.android.com/guide/topics/permissions/overview

Documentación en línea. (2024)

Documentación oficial sobre el manejo de permisos en Android Studio.

Notificaciones en Android Studio

https://developer.android.com/develop/ui/views/notifications?hl=en

Documentación en línea. (2024)

Documentación oficial sobre las notificaciones en Android Studio.

Postman

https://www.postman.com/

Documentación en línea. (2024)

Web oficial sobre Postman, aplicación muy estandarizada para probar API-Rest.

Entrenamientos

Entrenamiento 1

- Realizar pruebas de peticiones http a una Api-Rest con postman.
- Desarrollo paso a paso y solución: https://www.youtube.com/watch?v=qsejysrhJiU

Entrenamiento 2

- Desarrollar una app Android Studio que consuma una api desde JsonPlaceHolder Api: https://jsonplaceholder.typicode.com/.

- Desarrollo paso a paso y solución: https://www.youtube.com/watch?v=5fyrlA-4msA

Entrenamiento 3

- Desarrollar una App en Android que use Retrofit para obtener datos de imágenes de la Nasa.
- Desarrollo paso a paso y solución: https://developer.android.com/codelabs/basic-android-kotlin-compose-getting-data-internet?hl=es-419#0

Entrenamiento 4

- Desarrollar una app Android para configurar y gestionar los permisos de la misma.
- Desarrollo paso a paso y solución:
  https://www.youtube.com/watch?v=myZkkhLyVi8

Entrenamiento 5

- Desarrollar una app Android para configurar y gestionar los permisos de la misma utilizando Jet Pack Compose.
- Desarrollo paso a paso y solución:
  https://www.youtube.com/watch?v=LyGmjNHHRog