

UNIVERSIDAD NACIONAL POLITECNICA

Programación Movil



Investigación 2,5,6,8

Ingeniería en computación

Elaborado por: Keylin Marlene Reyes Ramirez

Docente: Ing. Luis Guido

**2- Informe de investigación About fragments, Create a fragment, Fragment manager, Fragment transactions, Animate transitions between fragments, Fragment lifecycle, Saving state with fragments y Communicate with fragments: Entregarlo como un informe en word o slides vale 4 puntos**

**Fragmentos**

Los fragmentos en programación móvil, particularmente en Android, son componentes modulares y reutilizables que representan porciones de la interfaz de usuario (UI) dentro de una actividad. Introducidos en Android 3.0, permiten dividir la UI en partes más manejables, facilitando una mejor organización del código y la reutilización de componentes. Cada fragmento tiene su propio ciclo de vida, que está estrechamente vinculado al ciclo de vida de la actividad que lo contiene, permitiendo que se añadan, reemplacen o eliminen dinámicamente durante la ejecución de la aplicación.

El uso de fragmentos mejora el manejo de diferentes configuraciones de pantalla y fomenta la separación de la lógica de la UI en componentes más pequeños y manejables. Los fragmentos pueden ser creados mediante clases que heredan de `Fragment`, definiendo su propia vista en archivos XML, y pueden ser añadidos a actividades tanto estática como dinámicamente. Esta modularidad y flexibilidad permiten crear interfaces de usuario más dinámicas y adaptables a diferentes dispositivos, como tablets y teléfonos.

**Cómo crear un fragmento**

Para crear un fragmento en una aplicación Android, primero debes crear una clase que extienda `Fragment`. En esta clase, sobrescribe el método `onCreateView` para inflar el layout del fragmento. Luego, define el layout del fragmento en un archivo XML en la carpeta `res/layout`. Una vez que hayas definido la interfaz de usuario del fragmento, puedes agregarlo a una actividad, ya sea de manera estática en el archivo de layout de la actividad o dinámicamente en tiempo de ejecución mediante el uso de `FragmentManager` y `FragmentTransaction`.

Aquí tienes un ejemplo completo de cómo hacerlo dinámicamente. Primero, crea una clase de fragmento y define su layout. Luego, agrega el fragmento a una actividad en tiempo de ejecución:

```java

// ExampleFragment.java

import android.os.Bundle;

import android.view.LayoutInflater;

import android.view.View;

import android.view.ViewGroup;

import androidx.fragment.app.Fragment;

public class ExampleFragment extends Fragment {

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

// Inflar el layout del fragmento

return inflater.inflate(R.layout.fragment\_example, container, false);

}

}

// fragment\_example.xml (ubicado en res/layout)

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

android:id="@+id/text\_view"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hola, Fragmento!" />

</LinearLayout>

// MainActivity.java

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

import androidx.fragment.app.FragmentManager;

import androidx.fragment.app.FragmentTransaction;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

// Crear una instancia del fragmento

ExampleFragment exampleFragment = new ExampleFragment();

// Obtener el FragmentManager y comenzar una transacción

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

// Agregar el fragmento al contenedor

fragmentTransaction.add(R.id.fragment\_container, exampleFragment);

fragmentTransaction.commit();

}

}

```

En este ejemplo, `ExampleFragment` es la clase del fragmento que inflará el layout `fragment\_example.xml`. En `MainActivity`, el fragmento se agrega dinámicamente al contenedor definido en el layout de la actividad (`activity\_main.xml`).

**Cómo crear una clase de fragmento**

 Para crear una clase de fragmento en Android, primero define una nueva clase que extienda `Fragment` y sobrescribe el método `onCreateView` para inflar el layout del fragmento. Luego, en la actividad, utiliza `FragmentManager` y `FragmentTransaction` para agregar el fragmento al contenedor definido en el layout de la actividad. Aquí tienes un ejemplo completo:

```java

// ExampleFragment.java

import android.os.Bundle;

import android.view.LayoutInflater;

import android.view.View;

import android.view.ViewGroup;

import androidx.fragment.app.Fragment;

public class ExampleFragment extends Fragment {

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

// Inflar el layout del fragmento

return inflater.inflate(R.layout.fragment\_example, container, false);

}

}

// fragment\_example.xml (ubicado en res/layout)

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

android:id="@+id/text\_view"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hola, Fragmento!" />

</LinearLayout>

// MainActivity.java

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

import androidx.fragment.app.FragmentManager;

import androidx.fragment.app.FragmentTransaction;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

// Crear una instancia del fragmento

ExampleFragment exampleFragment = new ExampleFragment();

// Obtener el FragmentManager y comenzar una transacción

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

// Agregar el fragmento al contenedor

fragmentTransaction.add(R.id.fragment\_container, exampleFragment);

fragmentTransaction.commit();

}

}

```

Este código muestra cómo definir un fragmento (`ExampleFragment`) con su propio layout (`fragment\_example.xml`) y cómo agregarlo dinámicamente a un contenedor en una actividad (`MainActivity`).

**Administrador de fragmentos**

El `FragmentManager` en Android es esencial para gestionar fragmentos dentro de una actividad. Permite realizar operaciones como añadir, reemplazar, eliminar y gestionar el estado de los fragmentos. Para usarlo, primero obtienes una instancia del `FragmentManager` desde la actividad utilizando `getSupportFragmentManager()`. Luego, comienzas una transacción de fragmento (`FragmentTransaction`) donde puedes agregar, reemplazar o eliminar fragmentos. Es importante confirmar la transacción utilizando `commit()` para que los cambios se reflejen correctamente en la interfaz de usuario.

Aquí tienes un ejemplo práctico que muestra cómo usar el `FragmentManager` para agregar un fragmento dinámicamente en una actividad:

```java

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

import androidx.fragment.app.FragmentManager;

import androidx.fragment.app.FragmentTransaction;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

// Crear una instancia del fragmento que deseas agregar

ExampleFragment exampleFragment = new ExampleFragment();

// Obtener el FragmentManager desde la actividad

FragmentManager fragmentManager = getSupportFragmentManager();

// Comenzar una transacción de fragmento

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

// Agregar el fragmento al contenedor definido en el layout de la actividad

fragmentTransaction.add(R.id.fragment\_container, exampleFragment);

// Confirmar la transacción para que se apliquen los cambios

fragmentTransaction.commit();

}

}

```

En este ejemplo, `ExampleFragment` es el fragmento que se agrega dinámicamente a través del `FragmentManager`. Es crucial definir un contenedor en el layout de la actividad (`activity\_main.xml` en este caso) donde se colocará el fragmento. Esta estructura modular facilita la creación de interfaces de usuario complejas y adaptables en aplicaciones Android.

**Transacciones de fragmentos**

 Las transacciones de fragmentos en Android son operaciones que permiten realizar cambios en la interfaz de usuario al añadir, reemplazar, eliminar o modificar fragmentos dentro de una actividad de manera atomizada. Estas transacciones se gestionan utilizando el `FragmentManager` y `FragmentTransaction`, asegurando que los cambios se realicen de manera consistente y segura.

Para iniciar una transacción de fragmento, primero se obtiene una instancia del `FragmentManager` utilizando `getSupportFragmentManager()` en una actividad que extienda `AppCompatActivity`. Luego, se crea una instancia de `FragmentTransaction` mediante `beginTransaction()` en el `FragmentManager`. Dentro de la transacción, se pueden realizar operaciones como añadir un nuevo fragmento al contenedor especificado en el layout de la actividad con `add()`, reemplazar un fragmento existente con `replace()`, eliminar un fragmento con `remove()`, o incluso modificar atributos de los fragmentos como agregar la transacción a la pila de retroceso con `addToBackStack(null)` para permitir la navegación hacia atrás.

A continuación, un ejemplo práctico que muestra cómo utilizar transacciones de fragmentos para añadir y reemplazar fragmentos en una actividad:

```java

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

import androidx.fragment.app.FragmentManager;

import androidx.fragment.app.FragmentTransaction;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

// Obtener el FragmentManager

FragmentManager fragmentManager = getSupportFragmentManager();

// Iniciar una nueva transacción

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

// Ejemplo 1: Añadir un fragmento

ExampleFragment fragment1 = new ExampleFragment();

fragmentTransaction.add(R.id.fragment\_container, fragment1);

// También se puede hacer fragmentTransaction.replace() para reemplazar un fragmento existente

// Ejemplo 2: Reemplazar un fragmento

ExampleFragment fragment2 = new ExampleFragment();

fragmentTransaction.replace(R.id.fragment\_container, fragment2);

// Otras operaciones posibles:

// fragmentTransaction.remove(fragment);

// fragmentTransaction.addToBackStack(null); // Para agregar la transacción a la pila de retroceso

// Confirmar la transacción

fragmentTransaction.commit();

}

}

```

Este código ilustra cómo iniciar una transacción de fragmento, añadir o reemplazar fragmentos en un contenedor (`fragment\_container` definido en `activity\_main.xml`), y confirmar la transacción para que los cambios se apliquen correctamente en la interfaz de usuario. Las transacciones de fragmentos son fundamentales para la creación de aplicaciones Android con interfaces dinámicas y flexibles.

**Cómo navegar entre fragmentos con animaciones**

Para navegar entre fragmentos con animaciones en Android, primero define las animaciones en archivos XML en la carpeta `res/anim`. Por ejemplo, `slide\_in\_left.xml` para la entrada y `slide\_out\_right.xml` para la salida.

\*\*slide\_in\_left.xml\*\* (entrada):

```xml

<translate xmlns:android="http://schemas.android.com/apk/res/android"

android:fromXDelta="-100%"

android:toXDelta="0%"

android:duration="300"/>

```

\*\*slide\_out\_right.xml\*\* (salida):

```xml

<translate xmlns:android="http://schemas.android.com/apk/res/android"

android:fromXDelta="0%"

android:toXDelta="100%"

android:duration="300"/>

```

En el código de tu actividad, utiliza `FragmentManager` y `FragmentTransaction` para añadir, reemplazar o eliminar fragmentos, especificando las animaciones creadas con `setCustomAnimations()` antes de confirmar la transacción con `commit()`. Esto permite transiciones fluidas entre fragmentos.

```java

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

import androidx.fragment.app.FragmentManager;

import androidx.fragment.app.FragmentTransaction;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

// Especificar animaciones personalizadas

fragmentTransaction.setCustomAnimations(R.anim.slide\_in\_left, R.anim.slide\_out\_right);

// Ejemplo: Añadir un fragmento con animaciones

ExampleFragment fragment = new ExampleFragment();

fragmentTransaction.replace(R.id.fragment\_container, fragment);

fragmentTransaction.commit();

}

}

```

Con este enfoque, puedes crear transiciones visuales agradables al cambiar entre fragmentos en tu aplicación Android, mejorando la experiencia del usuario.

**Ciclo de vida de los fragmentos**

El ciclo de vida de los fragmentos en Android es crucial para comprender cómo se comportan y cómo se pueden gestionar adecuadamente dentro de una actividad. Aquí tienes un resumen del ciclo de vida de los fragmentos:

1.Estado inicial (`Initial State):

- Cuando un fragmento se instancia por primera vez, se encuentra en este estado.

- El método onAttach() se llama cuando el fragmento se adjunta a su actividad.

- Luego, onCreate() se ejecuta para inicializar el fragmento, seguido de onCreateView() donde se infla su layout.

2. Estado activo (`Active State`):

- El fragmento entra en este estado cuando es visible para el usuario.

- `onStart()` y `onResume()` se invocan cuando el fragmento se hace visible y comienza a interactuar con el usuario.

3. Estado pausado (`Paused State`):

- El fragmento entra en este estado cuando pierde el foco pero sigue visible.

- `onPause()` se llama cuando el fragmento deja de ser activo pero aún es visible.

4. Estado detenido (`Stopped State`):

- Cuando el fragmento ya no es visible para el usuario pero aún está asociado a su actividad.

- `onStop()` indica que el fragmento ha dejado de estar visible.

5. Estado final (`Final State`):

- Cuando el fragmento ya no está asociado a su actividad y está a punto de ser destruido.

- `onDestroyView()` se llama cuando la vista del fragmento está siendo destruida.

- Finalmente, `onDetach()` indica que el fragmento se ha desvinculado completamente de su actividad.

Estos métodos del ciclo de vida proporcionan puntos de enganche (`hooks`) donde puedes realizar operaciones específicas dependiendo del estado del fragmento, como inicializar recursos en `onCreate()`, actualizar la interfaz de usuario en `onResume()`, o liberar recursos en `onDestroy()`.

Comprender y manejar correctamente el ciclo de vida de los fragmentos es fundamental para asegurar una aplicación Android robusta y eficiente, especialmente al trabajar con interfaces de usuario complejas y dinámicas.

**Cómo comunicarse con fragmentos**

Para comunicarte con fragmentos en Android, puedes utilizar interfaces o un ViewModel compartido. Aquí te presento un resumen conciso junto con un ejemplo utilizando interfaces:

Comunicación con Interfaces

1. Definir una interfaz en el fragmento:

```java

public class ExampleFragment extends Fragment {

public interface OnFragmentInteractionListener {

void onFragmentInteraction(String message);

}

private OnFragmentInteractionListener mListener;

// Método para conectar la actividad a la interfaz del fragmento

@Override

public void onAttach(@NonNull Context context) {

super.onAttach(context);

if (context instanceof OnFragmentInteractionListener) {

mListener = (OnFragmentInteractionListener) context;

} else {

throw new RuntimeException(context.toString()

+ " must implement OnFragmentInteractionListener");

}

}

// Método donde se llama a la interfaz para enviar datos

public void sendDataToActivity(String data) {

if (mListener != null) {

mListener.onFragmentInteraction(data);

}

}

}

```

2. \*\*Implementar la interfaz en la actividad:\*\*

```java

public class MainActivity extends AppCompatActivity implements ExampleFragment.OnFragmentInteractionListener {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

// Crear instancia del fragmento

ExampleFragment fragment = new ExampleFragment();

// Añadir el fragmento a la actividad

getSupportFragmentManager().beginTransaction()

.replace(R.id.fragment\_container, fragment)

.commit();

}

// Método de la interfaz que maneja la interacción del fragmento

@Override

public void onFragmentInteraction(String message) {

// Aquí manejas la información recibida desde el fragmento

Toast.makeText(this, "Mensaje del fragmento: " + message, Toast.LENGTH\_SHORT).show();

}

}

```

En este ejemplo, `ExampleFragment` define una interfaz `OnFragmentInteractionListener` y la actividad principal (`MainActivity`) la implementa. Cuando se necesita comunicación desde el fragmento a la actividad, se llama al método definido en la interfaz (`sendDataToActivity()` en el fragmento), que a su vez llama al método de la actividad (`onFragmentInteraction()` en `MainActivity`) para manejar los datos recibidos.

Este enfoque es útil cuando necesitas una comunicación directa y específica entre fragmentos y sus actividades contenedoras en Android.

**Cómo guardar un estado con fragmentos**

En Android, es importante gestionar el ciclo de vida de los fragmentos adecuadamente para garantizar que el estado de la interfaz de usuario y los datos se mantengan correctamente al cambiar entre configuraciones o actividades. Aquí te explico cómo puedes guardar y restaurar el estado de un fragmento:

Guardar Estado del Fragmento

Para guardar el estado de un fragmento, generalmente se utiliza el método `onSaveInstanceState(Bundle outState)`. Este método te permite guardar datos en un `Bundle` que se almacenará y se restaurará automáticamente por el sistema cuando sea necesario. Aquí tienes un ejemplo de cómo implementarlo en un fragmento:

java

public class ExampleFragment extends Fragment {

private static final String KEY\_COUNT = "count";

private int count = 0;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// Restaurar el estado anterior si existe

if (savedInstanceState != null) {

count = savedInstanceState.getInt(KEY\_COUNT, 0);

}

}

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,

Bundle savedInstanceState) {

View rootView = inflater.inflate(R.layout.fragment\_example, container, false);

// Incrementar el contador y actualizar la interfaz de usuario

rootView.findViewById(R.id.button\_increment).setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

count++;

updateCounter();

}

});

// Actualizar el contador inicial

updateCounter();

return rootView;

}

@Override

public void onSaveInstanceState(Bundle outState) {

super.onSaveInstanceState(outState);

// Guardar el estado actual en el Bundle

outState.putInt(KEY\_COUNT, count);

}

private void updateCounter() {

TextView textView = requireView().findViewById(R.id.text\_counter);

textView.setText(String.valueOf(count));

}

}

```

En este ejemplo:

- `onSaveInstanceState(Bundle outState)`: Se utiliza para guardar el estado actual del fragmento en un `Bundle` (`outState`). Aquí se guarda el valor del contador (`count`) en el `Bundle`.

- `onCreate(Bundle savedInstanceState)`: Se utiliza para restaurar el estado anterior del fragmento. Si `savedInstanceState` no es nulo, se extraen los datos guardados del `Bundle`.

Restaurar Estado del Fragmento

Cuando el sistema necesita restaurar el estado del fragmento (por ejemplo, después de que la actividad se recrea debido a un cambio de configuración), utiliza el `Bundle` guardado en `onSaveInstanceState(Bundle outState)` para restaurar los valores adecuados en los componentes de la interfaz de usuario o en los datos del fragmento.

Este método asegura que los datos críticos para el funcionamiento del fragmento se conserven y se vuelvan a cargar correctamente, proporcionando una experiencia de usuario más fluida y consistente en aplicaciones Android.

**5- Informe de investigación acerca de Room: Entregarlo ya sea como Word o Slides o si se les ocurre algo mejor bienvenido sea vale 4 puntos**

Room es una biblioteca de persistencia de datos en Android que se introdujo como parte de la arquitectura de componentes de Android Jetpack. Proporciona una capa de abstracción sobre SQLite para permitir un acceso más robusto y eficiente a la base de datos desde aplicaciones Android. Aquí tienes una visión general de Room y sus características principales:

Características Principales de Room:

1. Capa de Abstracción sobre SQLite:

- Room simplifica el uso de SQLite en aplicaciones Android al proporcionar una capa de abstracción que permite trabajar con objetos en lugar de manipular consultas SQL directamente.

2. Componentes Principales:

- Database: Define la base de datos que actúa como contenedor principal y sirve como punto de acceso a los datos.

- Entity: Representa una tabla en la base de datos SQLite y está anotada con `@Entity`.

- DAO (Data Access Object): Contiene métodos que definen las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para interactuar con la base de datos y está anotado con `@Dao`.

3. Soporte para Consultas Compiladas:

- Room permite definir consultas SQL directamente en los métodos de los DAOs. Estas consultas se verifican en tiempo de compilación, lo que proporciona seguridad y mejor rendimiento.

4. Integración con LiveData y RxJava:

- Room se integra bien con LiveData para que los cambios en la base de datos puedan ser observados fácilmente y se actualicen automáticamente en la interfaz de usuario. También es compatible con RxJava para aquellos que prefieren el modelo reactivo.

5. Migraciones Automáticas:

- Room maneja automáticamente las migraciones de la base de datos cuando se modifican los esquemas, lo que simplifica la gestión de versiones de la base de datos en el desarrollo de aplicaciones.

6. Compatibilidad con Kotlin:

- Room está diseñado para ser compatible con Kotlin, aprovechando las características del lenguaje como las funciones de extensión y la nulabilidad segura.

Uso Básico de Room:

1. Definir una Entidad:

java

@Entity(tableName = "users")

public class User {

@PrimaryKey(autoGenerate = true)

public int id;

public String name;

public int age;

}

```

2. Crear un DAO:

```java

@Dao

public interface UserDao {

@Insert

void insert(User user);

@Query("SELECT \* FROM users")

LiveData<List<User>> getAllUsers();

}

```

3. Crear una Base de Datos:

java

@Database(entities = {User.class}, version = 1)

public abstract class MyAppDatabase extends RoomDatabase {

public abstract UserDao userDao();

}

```

4. Obtener una Instancia de la Base de Datos:

```java

MyAppDatabase db = Room.databaseBuilder(getApplicationContext(),

MyAppDatabase.class, "myapp-database").build();

```

5. Ejemplo de Uso en una Actividad:

```java

LiveData<List<User>> usersLiveData = db.userDao().getAllUsers();

usersLiveData.observe(this, users -> {

// Actualizar la interfaz de usuario con los datos obtenidos de la base de datos

});

```

Room facilita el manejo de bases de datos locales en aplicaciones Android, proporcionando una solución robusta y eficiente para persistir datos de manera segura y organizada.

**6- Informe de investigación del patrón de arquitectura (Model View View-Model) MVVM: Entregarlo ya sea como Word o Slides o si se les ocurre algo mejor bienvenido sea vale 4 puntos**

El patrón de arquitectura Model-View-ViewModel (MVVM) es un enfoque utilizado en el desarrollo de aplicaciones de software que separa la lógica de presentación de la lógica de negocio y los datos. MVVM facilita la modularidad, la prueba automatizada y la gestión del ciclo de vida en aplicaciones que utilizan interfaces de usuario (UI) complejas, como las aplicaciones móviles y de escritorio. Aquí te explico los componentes y principios clave del patrón MVVM:

Componentes de MVVM:

1. Modelo (Model):

- Representa los datos y la lógica de negocio de la aplicación. El modelo es independiente de la interfaz de usuario y puede incluir acceso a la red, persistencia de datos, y otras operaciones de procesamiento de datos.

2. Vista (View):

- Es la capa de presentación que muestra la interfaz de usuario y responde a las interacciones del usuario (entradas y gestos). La vista envía comandos y actualizaciones a través de enlaces de datos al ViewModel.

3. Modelo de Vista (ViewModel):

- Actúa como un intermediario entre la vista y el modelo, gestionando el estado de la interfaz de usuario y la lógica de presentación. El ViewModel no está directamente acoplado a la vista, lo que facilita las pruebas unitarias y permite reutilizar la lógica de presentación en múltiples vistas.

Principios y Características de MVVM:

- Separación de Responsabilidades: MVVM promueve una separación clara de las responsabilidades entre la vista, el modelo y el ViewModel. Esto facilita el mantenimiento del código y permite cambios en una capa sin afectar directamente a las otras.

- Enlaces de Datos (Data Binding): Un aspecto fundamental de MVVM es el uso de enlaces de datos para conectar automáticamente los datos entre la vista y el ViewModel. Esto permite que los cambios en los datos del ViewModel se reflejen automáticamente en la vista y viceversa, eliminando la necesidad de gestionar manualmente las actualizaciones de la interfaz de usuario.

- Gestión del Ciclo de Vida: El ViewModel está diseñado para sobrevivir a los cambios de configuración, como rotaciones de pantalla, en aplicaciones Android. Esto asegura que los datos del ViewModel no se pierdan y que la interfaz de usuario mantenga su estado correcto.

- Facilita las Pruebas Unitarias: Debido a la separación de la lógica de presentación y la interfaz de usuario, MVVM facilita las pruebas unitarias del ViewModel sin depender de la interfaz de usuario. Esto mejora la calidad del software y la capacidad de mantenimiento a largo plazo.

Ejemplo de Implementación en Android:

1. Definir un ViewModel:

java

public class MyViewModel extends ViewModel {

private MutableLiveData<String> mText;

public MyViewModel() {

mText = new MutableLiveData<>();

mText.setValue("Hello, MVVM!");

}

public LiveData<String> getText() {

return mText;

}

}

```

2. Conectar el ViewModel con la Vista (Activity o Fragment):

```java

public class MyActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

MyViewModel viewModel = new ViewModelProvider(this).get(MyViewModel.class);

// Observar los cambios en los datos del ViewModel

viewModel.getText().observe(this, new Observer<String>() {

@Override

public void onChanged(String text) {

// Actualizar la interfaz de usuario con el texto del ViewModel

TextView textView = findViewById(R.id.text\_view);

textView.setText(text);

}

});

}

}

```

En este ejemplo, `MyViewModel` actúa como el ViewModel que gestiona los datos (`mText` en este caso) y se observa desde la actividad (`MyActivity`). Cuando `mText` cambia en el ViewModel, la interfaz de usuario en la actividad se actualiza automáticamente gracias al enlace de datos proporcionado por LiveData.

MVVM es un patrón potente y ampliamente adoptado en el desarrollo de aplicaciones modernas, proporcionando una estructura clara y eficiente para la creación de interfaces de usuario dinámicas y mantenibles en plataformas como Android y otros frameworks de desarrollo de software.

**8- Infografía entre Volley y Retrofit: Entregarlo como una infografia o si se les ocurre algo mejor bienvenido sea vale 3 puntos**

Crear una infografía comparativa entre Volley y Retrofit puede ser una forma visualmente efectiva de entender las diferencias clave entre estas dos bibliotecas para realizar solicitudes HTTP en Android. Aquí tienes una estructura básica para una infografía:

Volley vs Retrofit: Infografía Comparativa

1. Introducción

- Volley y Retrofit son bibliotecas populares para manejar solicitudes HTTP en aplicaciones Android.

- Ambas facilitan la comunicación con servidores web, pero difieren en enfoque y características.

2. Desarrollo de Google

- Volley fue desarrollado por Google para manejar de manera eficiente las solicitudes HTTP y la gestión de redes en aplicaciones Android.

- Retrofit fue desarrollado por Square para simplificar la comunicación de red al hacer uso de interfaces Java para definir solicitudes web.

3. Características Principales

- Volley:

- Gestión automática de hilos.

- Soporte para peticiones GET, POST, PUT, DELETE.

- Personalización de solicitudes y caché de respuestas.

- Retrofit:

- Basado en anotaciones y interfaces Java.

- Integración sencilla con Gson o Jackson para el análisis JSON.

- Gestión potente de errores y estado de la red.

4. Uso y Flexibilidad

- Volley:

- Ideal para pequeñas aplicaciones o proyectos donde se necesita una gestión automática de hilos y cacheo.

- Retrofit:

- Perfecto para aplicaciones grandes y complejas que requieren una gestión avanzada de solicitudes RESTful y análisis JSON.

5. Ejemplos de Código

- Volley:

```java

String url = "http://api.example.com/data";

RequestQueue queue = Volley.newRequestQueue(context);

StringRequest stringRequest = new StringRequest(Request.Method.GET, url,

new Response.Listener<String>() {

@Override

public void onResponse(String response) {

// Manejar la respuesta

}

}, new Response.ErrorListener() {

@Override

public void onErrorResponse(VolleyError error) {

// Manejar errores

}

});

queue.add(stringRequest);

```

- Retrofit:

```java

public interface ApiService {

@GET("data")

Call<DataModel> getData();

}

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://api.example.com/")

.addConverterFactory(GsonConverterFactory.create())

.build();

ApiService service = retrofit.create(ApiService.class);

Call<DataModel> call = service.getData();

call.enqueue(new Callback<DataModel>() {

@Override

public void onResponse(Call<DataModel> call, Response<DataModel> response) {

// Manejar la respuesta

}

@Override

public void onFailure(Call<DataModel> call, Throwable t) {

// Manejar errores

}

});

```

6. Elección de la Biblioteca

- Volley: Simple y eficiente para tareas básicas de red.

- Retrofit: Completo y flexible para aplicaciones complejas que requieren una comunicación RESTful robusta.

7. Conclusión

- Ambas bibliotecas son poderosas y adecuadas para diferentes necesidades de desarrollo de aplicaciones Android.

- La elección entre Volley y Retrofit depende del tamaño del proyecto, la complejidad de la aplicación y los requisitos específicos de la red.

Esta infografía puede ser diseñada utilizando herramientas de diseño gráfico como Adobe Illustrator, Canva o cualquier otra herramienta similar. Asegúrate de incluir iconografía relevante, colores contrastantes y texto conciso para facilitar la comprensión y hacer que la información sea visualmente atractiva.