Victor Manuel Leiva Requene

Acumulados para el segundo parcial son 30 puntos y el segundo parcial vale 20 puntos:

1- Android Kotlin Fundamentals: Create dynamic lists with RecyclerView vale 4 puntos
https:developer.android.com/develop/ui/views/layout/recyclerview

Entendido. Aquí tienes la información detallada sobre "Android Kotlin Fundamentals: Create dynamic lists with RecyclerView" basada en la documentación oficial de Android.

Android Kotlin Fundamentals: Crear Listas Dinámicas con RecyclerView

Introducción

RecyclerView es una vista de Android que proporciona una manera eficiente y flexible de mostrar listas de datos. Es una evolución de ListView y GridView, ofreciendo una mejor manera de manejar grandes conjuntos de datos y mejorar el rendimiento de la interfaz de usuario.

¿Qué es RecyclerView?

Historia y Origen

RecyclerView es parte del soporte de Android Jetpack y fue introducido para reemplazar ListView y GridView. Ofrece una mayor eficiencia y flexibilidad en la gestión de listas de datos, utilizando un patrón de diseño conocido como ViewHolder.

Funcionalidades Principales

- ViewHolder Pattern: Optimiza el rendimiento al reciclar vistas que ya no son visibles.

- LayoutManager: Define cómo se organizan y muestran los elementos en el RecyclerView (vertical, horizontal, grid, etc.).

- Adapter: Se encarga de crear y enlazar los elementos de la lista con las vistas correspondientes.

Implementación de RecyclerView en Kotlin

Configuración Inicial

1. Agregar Dependencias:

En el archivo `build.gradle` (Module: app):

```
gradle
dependencies {
    implementation "androidx.recyclerview:recyclerview:1.2.1"
}
```

2. Diseñar el Layout XML:

Define el layout del RecyclerView en `activity_main.xml`:

xml

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"/>
```

Creación de la Vista de Elemento

1. Crear un Layout para los Elementos:

En `item_list.xml`, define el diseño de cada ítem de la lista:

xml

```
<LinearLayout xmlns:android="http:schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">
    <TextView
        android:id="@+id/textViewTitle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="18sp"
        android:textStyle="bold"/>
    <TextView
        android:id="@+id/textViewDescription"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="14sp"/>
</LinearLayout>
```

2. Crear una Clase de ViewHolder:

En `ItemViewHolder.kt`, define la clase ViewHolder:

kotlin

```
import android.view.View
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
class ItemViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val title: TextView = view.findViewById(R.id.textViewTitle)
    val description: TextView = view.findViewById(R.id.textViewDescription)
}
```

Creación del Adapter

1. Crear la Clase Adapter:

En `MyAdapter.kt`, define el adapter:

kotlin

```
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
class MyAdapter(private val itemList: List<Item>) : RecyclerView.Adapter<MyAdapter.MyViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.item_list, parent, false)
        return MyViewHolder(view)
    }
    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        val item = itemList[position]
        holder.title.text = item.title
```

```
        holder.description.text = item.description
    }
    override fun getItemCount(): Int = itemList.size
    class MyViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val title: TextView = view.findViewById(R.id.textViewTitle)
        val description: TextView = view.findViewById(R.id.textViewDescription)
    }
}
```

Configuración del LayoutManager

1. Configurar el RecyclerView en el Activity:

En `MainActivity.kt`, inicializa el RecyclerView y el Adapter:

kotlin

```
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)
        recyclerView.layoutManager = LinearLayoutManager(this)
        val itemList = listOf(
            Item("Title 1", "Description 1"),
            Item("Title 2", "Description 2"),
            Item("Title 3", "Description 3")
        )
        val adapter = MyAdapter(itemList)
        recyclerView.adapter = adapter
```

```
    }
  }
  data class Item(val title: String, val description: String)
```

Ejemplo Práctico

1. activity_main.xml:

```
  xml
  <androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"/>
```

2. item_list.xml:

```
  xml
  <LinearLayout xmlns:android="http:schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">
    <TextView
      android:id="@+id/textViewTitle"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:textSize="18sp"
      android:textStyle="bold"/>
    <TextView
      android:id="@+id/textViewDescription"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:textSize="14sp"/>
```

```
</LinearLayout>
```

3. ItemViewHolder.kt:

```
kotlin

import android.view.View

import android.widget.TextView

import androidx.recyclerview.widget.RecyclerView

class ItemViewHolder(view: View) : RecyclerView.ViewHolder(view) {

    val title: TextView = view.findViewById(R.id.textViewTitle)

    val description: TextView = view.findViewById(R.id.textViewDescription)

}
```

4. MyAdapter.kt:

```
kotlin

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.TextView

import androidx.recyclerview.widget.RecyclerView

class MyAdapter(private val itemList: List<Item>) : RecyclerView.Adapter<MyAdapter.MyViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyViewHolder {

        val view = LayoutInflater.from(parent.context).inflate(R.layout.item_list, parent, false)

        return MyViewHolder(view)

    }

    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {

        val item = itemList[position]

        holder.title.text = item.title

        holder.description.text = item.description

    }
```

```
    override fun getItemCount(): Int = itemList.size
    class MyViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val title: TextView = view.findViewById(R.id.textViewTitle)
        val description: TextView = view.findViewById(R.id.textViewDescription)
    }
}
```

5. MainActivity.kt:

```
kotlin
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)
        recyclerView.layoutManager = LinearLayoutManager(this)
        val itemList = listOf(
            Item("Title 1", "Description 1"),
            Item("Title 2", "Description 2"),
            Item("Title 3", "Description 3")
        )
        val adapter = MyAdapter(itemList)
        recyclerView.adapter = adapter
    }
}
data class Item(val title: String, val description: String)
```

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

https:developer.android.com/guide/fragments,

https:developer.android.com/guide/fragments/create,

https:developer.android.com/guide/fragments/fragmentmanager,

https:developer.android.com/guide/fragments/transactions,

https:developer.android.com/guide/fragments/animate,

https:developer.android.com/guide/fragments/lifecycle,

https:developer.android.com/guide/fragments/saving-state

https:developer.android.com/guide/fragments/communicate

Fragments en Android

1. Introducción a los Fragments

Los fragments son componentes modulares y reutilizables de la interfaz de usuario que forman parte de una actividad. Permiten crear aplicaciones flexibles y dinámicas, especialmente en dispositivos con diferentes tamaños de pantalla. Los fragments pueden ser añadidos, eliminados o reemplazados mientras la actividad está en ejecución, lo que permite cambiar la interfaz de usuario de manera flexible y dinámica.

2. Crear un Fragment

Para crear un fragmento, se debe extender la clase `Fragment` o sus subclases (`DialogFragment`, `ListFragment`, etc.). Un fragmento debe tener al menos el método `onCreateView` que infla el layout del fragmento.

Ejemplo de código:

kotlin

```
class ExampleFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_example, container, false)
```

```
    }
}
```

El layout del fragmento (en este caso, `fragment_example.xml`) debe estar definido en la carpeta `res/layout`.

3. Fragment Manager

El `FragmentManager` es responsable de gestionar los fragments en una actividad. Permite realizar operaciones como añadir, eliminar, reemplazar y realizar transacciones con fragments.

Uso básico:

kotlin

```
val fragmentManager = supportFragmentManager
val fragmentTransaction = fragmentManager.beginTransaction()
fragmentTransaction.add(R.id.fragment_container, ExampleFragment())
fragmentTransaction.commit()
```

Funciones Principales:

- `add()`: Añade un fragmento al contenedor especificado.
- `replace()`: Reemplaza un fragmento en el contenedor especificado.

4. Fragment Transactions

Las transacciones de fragments permiten realizar una serie de cambios en la interfaz de usuario en una operación atómica. Las transacciones se realizan utilizando el `FragmentTransaction` y pueden incluir operaciones como añadir, reemplazar o eliminar fragments.

Ejemplo de transacción:

kotlin

```
val fragmentTransaction = supportFragmentManager.beginTransaction()
fragmentTransaction.replace(R.id.fragment_container, NewFragment())
fragmentTransaction.addToBackStack(null)  Añade la transacción a la pila de retroceso
fragmentTransaction.commit()
```

Referencias: https:developer.android.com/guide/fragments/transactions

5. Animar Transiciones entre Fragments

Las transiciones entre fragments pueden ser animadas para proporcionar una experiencia de usuario más suave y visualmente atractiva. Se pueden definir animaciones para la entrada y salida de fragments durante las transacciones.

Ejemplo de animación:

kotlin

```
val fragmentTransaction = supportFragmentManager.beginTransaction()

fragmentTransaction.setCustomAnimations(

   R.anim.enter_from_right, R.anim.exit_to_left,

   R.anim.enter_from_left, R.anim.exit_to_right

)

fragmentTransaction.replace(R.id.fragment_container, NewFragment())

fragmentTransaction.commit()
```

6. Ciclo de Vida de un Fragment

El ciclo de vida de un fragment es similar al de una actividad, pero con algunas diferencias clave. Los métodos principales incluyen `onAttach()`, `onCreate()`, `onCreateView()`, `onActivityCreated()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroyView()`, `onDestroy()`, y `onDetach()`.

Referencias: https:developer.android.com/guide/fragments/lifecycle

7. Guardar el Estado con Fragments

Es importante guardar el estado de un fragment para que los datos y la interfaz de usuario puedan ser restaurados correctamente cuando el fragmento se recrea. Esto se puede hacer utilizando el método `onSaveInstanceState()`.

Ejemplo de guardar estado:

kotlin

```
override fun onSaveInstanceState(outState: Bundle) {

   super.onSaveInstanceState(outState)

   outState.putString("key", value)

}

override fun onViewStateRestored(savedInstanceState: Bundle?) {

   super.onViewStateRestored(savedInstanceState)

   val value = savedInstanceState?.getString("key")

}
```

Referencias: https:developer.android.com/guide/fragments/saving-state

8. Comunicar con Fragments

La comunicación entre fragments y su actividad contenedora o entre diferentes fragments se puede lograr mediante interfaces. Esto asegura que los fragments permanezcan desacoplados y reutilizables.

Ejemplo de comunicación:

kotlin

```
class ExampleFragment : Fragment() {
    private var listener: OnFragmentInteractionListener? = null
    interface OnFragmentInteractionListener {
        fun onFragmentInteraction(data: String)
    }
    override fun onAttach(context: Context) {
        super.onAttach(context)
        if (context is OnFragmentInteractionListener) {
            listener = context
        } else {
            throw RuntimeException("$context must implement OnFragmentInteractionListener")
        }
    }
    fun sendDataToActivity(data: String) {
        listener?.onFragmentInteraction(data)
    }
}
```

En la actividad contenedora:

kotlin

```
class MainActivity : AppCompatActivity(), ExampleFragment.OnFragmentInteractionListener {
    override fun onFragmentInteraction(data: String) {
        Manejar la interacción del fragmento
```

}

}

Referencias: https:developer.android.com/guide/fragments/communicate

3- Android Kotlin Fundamentals: Sending the user to another app vale 4 puntos

https:developer.android.com/training/basics/intents/sending

Codelabs: https:github.com/Vleiva220503/ejemploKotlin.git

4- Codelabs: Como trabajar con Preferences Datastore vale 4 puntos

https:developer.android.com/codelabs/android-preferences-datastore?hl=es-4190

Codelabs: https:github.com/Vleiva220503/Android-Kotlin-Fundamentals.git

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

Informe de Investigación sobre Room en Kotlin

1. Introducción

En la introducción, se explicará brevemente la importancia de la persistencia de datos en las aplicaciones móviles. Se presentarán los objetivos del informe y la relevancia de utilizar Room como biblioteca de persistencia en el desarrollo de aplicaciones Android utilizando Kotlin.

Ejemplo:

La persistencia de datos es un aspecto crucial en el desarrollo de aplicaciones móviles, ya que permite almacenar información de manera permanente, incluso después de cerrar la aplicación. Room, una biblioteca de persistencia de datos proporcionada por Google, facilita el uso de bases de datos SQLite en aplicaciones Android al proporcionar una capa de abstracción que simplifica las operaciones de CRUD (Crear, Leer, Actualizar, Borrar). Este informe tiene como objetivo explorar las características de Room, su implementación en Kotlin, y comparar sus ventajas y desventajas frente a otras soluciones.

2. Descripción de Room

Historia y Origen

Room es parte de la arquitectura de componentes Jetpack de Android, lanzada por Google para simplificar el desarrollo de aplicaciones móviles. Room fue introducido como una solución para manejar la persistencia de datos de manera más eficiente y segura, evitando las complicaciones asociadas al uso directo de SQLite.

Ejemplo:

Room fue lanzado por Google como parte de la colección Jetpack en 2017, con el objetivo de proporcionar una manera más sencilla y robusta de manejar la persistencia de datos en aplicaciones Android. Al abstraer muchas de las complejidades del acceso a bases de datos SQLite, Room facilita a los desarrolladores la creación de aplicaciones con bases de datos seguras y eficientes.

Funcionalidades Principales

Room ofrece varias funcionalidades clave que simplifican el manejo de bases de datos en aplicaciones Android:

- Entidades: Representan las tablas en la base de datos.

- DAO (Data Access Object): Define métodos para acceder a la base de datos.

- Base de Datos: La clase que contiene la base de datos y actúa como el punto de entrada principal para las interacciones con los datos.

- Migraciones: Facilitan las actualizaciones de la estructura de la base de datos sin perder datos existentes.

- Integración con LiveData y Coroutines: Permite la actualización automática de la UI cuando cambian los datos y facilita las operaciones asíncronas.

Ejemplo:

Room proporciona una abstracción sobre SQLite que permite a los desarrolladores trabajar con bases de datos de manera más intuitiva y menos propensa a errores. Al definir entidades, DAOs y la base de datos, Room facilita la implementación de operaciones CRUD y la gestión de migraciones, además de ofrecer integración con LiveData y coroutines para un manejo eficiente de los datos.

3. Implementación de Room en Kotlin

Configuración Inicial

Para empezar a usar Room en un proyecto Android con Kotlin, es necesario agregar las dependencias correspondientes en el archivo `build.gradle` del módulo de la aplicación.

Ejemplo:

gradle

```
dependencies {
    implementation "androidx.room:room-runtime:2.4.2"
    kapt "androidx.room:room-compiler:2.4.2"
    implementation "androidx.room:room-ktx:2.4.2"
}
```

Creación de Entidades

Las entidades representan las tablas en la base de datos. Cada entidad es una clase de datos anotada con `@Entity`.

Ejemplo:

kotlin

```
@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int,
    @ColumnInfo(name = "first_name") val firstName: String,
    @ColumnInfo(name = "last_name") val lastName: String
)
```

Definición de DAO

El DAO (Data Access Object) define los métodos de acceso a la base de datos. Se anotan con `@Dao` y pueden incluir métodos para insertar, actualizar, eliminar y consultar datos.

Ejemplo:

kotlin

```
@Dao
interface UserDao {
    @Insert
    suspend fun insert(user: User)

    @Update
    suspend fun update(user: User)

    @Delete
    suspend fun delete(user: User)
    @Query("SELECT  FROM users")
    fun getAllUsers(): LiveData<List<User>>
}
```

Configuración de la Base de Datos

La clase de la base de datos debe ser una clase abstracta que extiende `RoomDatabase` y está anotada con `@Database`, especificando las entidades y la versión de la base de datos.

Ejemplo:

kotlin

```
@Database(entities = [User::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "app_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

4. Casos de Uso y Ejemplos

Ejemplo de CRUD (Crear, Leer, Actualizar, Borrar)

Un ejemplo práctico que demuestre cómo realizar operaciones CRUD utilizando Room en Kotlin.

Ejemplo:

kotlin

```
val userDao = AppDatabase.getDatabase(context).userDao()
```

Insertar un nuevo usuario

```
val user = User(firstName = "John", lastName = "Doe")
userDao.insert(user)
```

Leer todos los usuarios

```
userDao.getAllUsers().observe(this, Observer { users ->
    Actualizar la UI con la lista de usuarios
})
```

Actualizar un usuario

```
val updatedUser = user.copy(firstName = "Jane")
userDao.update(updatedUser)
```

Eliminar un usuario

```
userDao.delete(updatedUser)
```

Integración en una Aplicación Android

Cómo integrar Room en una aplicación Android existente y manejar el ciclo de vida de los datos, incluyendo el uso de ViewModel y LiveData para una arquitectura más robusta.

5. Ventajas y Desventajas

Beneficios de Usar Room

- Simplifica el acceso a la base de datos con una API limpia y fácil de usar.

- Reduce el boilerplate comparado con el uso directo de SQLite.

- Soporte para LiveData y coroutines, facilitando el manejo de datos en la UI y operaciones asíncronas.

- Gestión de migraciones de base de datos de manera segura y sencilla.

Limitaciones y Desafíos

- Puede haber una curva de aprendizaje inicial para entender la configuración y el uso de anotaciones.

- Las migraciones complejas pueden requerir una planificación cuidadosa.

- Dependencia en la implementación correcta de DAOs y entidades para evitar errores de tiempo de ejecución.

6. Comparativa con Tecnologías Similares

SQLite Directo

- Room proporciona una capa de abstracción que simplifica muchas tareas que, de otro modo, requerirían escribir SQL manualmente.

- El uso directo de SQLite puede ser más flexible en algunos casos, pero también más propenso a errores y con más código boilerplate.

Otros ORM (Object-Relational Mapping)

- Comparado con otros ORM como Realm o ObjectBox, Room tiene la ventaja de estar respaldado por Google y ser parte de la arquitectura Jetpack.

- Cada ORM tiene sus propias ventajas y limitaciones en términos de rendimiento, facilidad de uso y características.

7. Conclusión

Resumen de los puntos clave discutidos en el informe y una evaluación final de la utilidad y efectividad de Room en el desarrollo de aplicaciones Android con Kotlin.

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

Por supuesto, aquí tienes una guía básica sobre el patrón de arquitectura MVVM (Model-View-ViewModel):

Patrón MVVM (Model-View-ViewModel)

El patrón MVVM es un patrón de diseño arquitectónico ampliamente utilizado en el desarrollo de aplicaciones de software, especialmente en plataformas como Android y WPF (Windows Presentation Foundation). Este patrón se centra en separar la lógica de presentación de la lógica de negocio y los datos.

Componentes Principales:

1. Modelo (Model):

- Representa los datos y las estructuras de datos de la aplicación.

- No tiene conocimiento de la interfaz de usuario (UI) ni de cómo se presenta la información.

2. Vista (View):

- Es la interfaz de usuario que muestra los datos al usuario y captura las interacciones del usuario.

- No contiene lógica de negocio o acceso directo a datos.

3. ViewModel:

- Actúa como un intermediario entre la vista y el modelo.

- Expone métodos y comandos para que la vista interactúe con el modelo.

- Contiene la lógica de presentación y maneja la lógica de estado de la vista.

- No tiene conocimiento directo de la vista con la que está vinculado.

Flujo de Datos en MVVM:

- La vista observa y enlaza (binds) sus elementos visuales (widgets) a propiedades y comandos expuestos por el ViewModel.

- El ViewModel se suscribe a cambios en el modelo y actualiza las propiedades que la vista enlaza.

- La vista maneja eventos del usuario y los pasa al ViewModel para realizar acciones en el modelo.

- Esta separación permite una mayor modularidad, facilita las pruebas unitarias y mejora la mantenibilidad del código.

Ventajas de MVVM:

- Separación de Responsabilidades: Clarifica y separa la lógica de presentación de la lógica de negocio y los datos.

- Facilita la Prueba Unitaria: Permite probar la lógica de presentación y la lógica de negocio de manera independiente.

- Reutilización de Código: Al separar la lógica de presentación, facilita la reutilización del ViewModel en diferentes vistas.

- Mejora la Mantenibilidad: La estructura clara y la separación de responsabilidades hacen que el código sea más fácil de mantener y escalar.

Implementación en Android:

- En Android, MVVM se implementa utilizando librerías como Android Jetpack (específicamente LiveData y ViewModel) para manejar la comunicación entre la vista y el ViewModel, y Retrofit o Room para la comunicación con el modelo.

7- Codelabs: Android Paging Basics vale 3 puntos

https:developer.android.com/codelabs/android-paging-basics

Codelabs: https:github.com/Vleiva220503/Android-Paging-.git

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

Infografía Comparativa: Volley vs Retrofit

1. Propósito Principal

- Volley: Orientado a realizar peticiones HTTP de manera sencilla y rápida, adecuado para aplicaciones con un número moderado de solicitudes.

- Retrofit: Especializado en consumo de API RESTful, con soporte para operaciones CRUD y conversión automática de JSON a objetos Java/Kotlin.

2. Arquitectura y Diseño

- Volley: Utiliza una arquitectura basada en solicitudes de red en cola, adecuada para operaciones de red simples.

- Retrofit: Se basa en una arquitectura de cliente HTTP basado en OkHttp, con soporte para anotaciones que simplifican la definición de servicios REST.

3. Solicitud y Respuesta

- Volley: Maneja solicitudes y respuestas de red de manera asincrónica, con personalización limitada de los objetos de solicitud.

- Retrofit: Define interfaces Java/Kotlin para especificar las solicitudes HTTP y las respuestas esperadas, con soporte para diferentes tipos de solicitudes y respuestas.

4. Gestión de Errores y Cancelación

- Volley: Proporciona métodos para cancelar solicitudes y manejar errores de red y de tiempo de espera.

- Retrofit: Ofrece manejo robusto de errores mediante interceptores y configuraciones personalizadas, facilitando la gestión de errores y reintentos.

5. Popularidad y Soporte

- Volley: Mantenido por Google, con soporte para operaciones básicas de red, aunque menos actualizado en comparación con Retrofit.

- Retrofit: Ampliamente adoptado por la comunidad de desarrollo Android, con actualizaciones frecuentes y soporte activo en GitHub.

6. Integración con Librerías Adicionales

- Volley: Puede integrarse fácilmente con otras bibliotecas de Android para tareas específicas, como manejo de imágenes.

- Retrofit: Compatible con OkHttp para operaciones de red avanzadas y Glide/Picasso para carga de imágenes, facilitando la integración con otras herramientas populares.

7. Ejemplos de Código

- Incluir ejemplos simples de cómo realizar una solicitud GET usando ambas bibliotecas para ilustrar la diferencia en la implementación y la legibilidad del código.

Formato de la Infografía

- Utiliza gráficos claros y concisos para representar cada punto.

- Incluye viñetas para destacar características clave.

- Asegúrate de que el diseño sea visualmente atractivo y fácil de seguir.

Herramientas Recomendadas

- Canva: Una herramienta en línea que facilita la creación de infografías con plantillas prediseñadas.

- Adobe Illustrator: Para un diseño más personalizado y detallado.

- PowerPoint: Ideal para crear una infografía rápida utilizando formas y gráficos integrados.