{desafío} latam_

API REST y acceso a recursos remotos (Parte II)

Restfull/Restless y REST Post

Competencias:

- Conocer qué son los servicios RestFull
- Conocer qué son los servicios Restless
- Implementar Retrofit para consumir una API en una aplicación Android.

Introducción

En este capítulo abordaremos las características que componen a un sistema REstfull, diferenciando y aclarando otras variantes para tener conceptos más claros, respecto a estos términos utilizados en la industria, de esta manera desarrollaremos mejores aplicaciones basadas en arquitectura REST.

Recordando que REST incluye la forma en que los estados de los recursos se dirigen y transfieren datos y archivos a través de HTTP, profundizaremos en la utilización del método POST para continuar con la aplicación “RestApi” de los ejercicios y crear en esta ocasión nuevos post.

Adicionalmente, revisaremos los conceptos Restfull y Restless de la programación de servicios informáticos REST.

Restfull

La palabra Restfull hace referencia a la disponibilidad de servicios web u otros sistemas, que ofrecen recursos basados en el protocolo HTTP, por lo que para su acceso se usan los métodos GET, POST, etc., es decir, Restfull se puede entender como un grupo de servicios web o programas basados en la arquitectura HTTP que convoca REST.

Estos servicios web o sistemas ofrecen recursos en diversos tipos de formatos, según se requiera, son accesibles mediante direcciones web (url) y no se necesitan frameworks para acceder a ellos, por eso se les consideran ligeros.

Restless

Restless es cualquier sistema que no sea Restfull, es decir, basta con que el sistema no cumpla con alguna de las características de un sistema Restfull.

Aunque parezca asombroso, la mayoría de los sistemas son RESTless dado que no cumplen con una característica elemental algo desconocida en su concepto, el Hypermedia as the Engine of Application State **(**HATEOAS), el cual es un componente de la arquitectura de aplicaciones REST que lo distingue de otras arquitecturas de red.

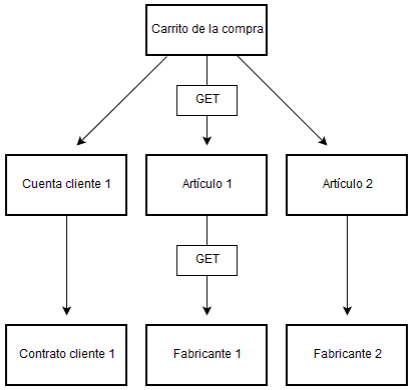

Imagen 1. Representación esquemática de la aplicación de HATEOAS en una tienda online.

En un servicio REST que siga el principio HATEOAS tal y como lo define REST, el usuario solo ha de conocer el URI inicial para poder interactuar con la oferta.

Características de un sistema RestFull

Para la conexión entre el servidor y el cliente, REST define estas cuatro características:

- **Identificación inequívoca de todos los recursos**: todos los recursos han de poder identificarse con un URI (Unique Resource Identifier).
- **Interacción con los recursos por medio de representaciones**: si un cliente necesita un recurso, el servidor le envía una representación (p. ej., HTML, JSON o XML) para que el cliente pueda modificar o borrar el recurso original.
- **Mensajes explícitos**: cada mensaje intercambiado por el servidor y el cliente ha de contener todos los datos necesarios para entenderse.
- **HATEOAS**: este principio también integra una API REST. Esta estructura basada en hipermedia facilita a los clientes el acceso a la aplicación, puesto que de este modo no necesitan saber nada más de la interfaz para poder acceder y navegar por ella.

HATEOAS y REST: el paradigma hipermedia

HATEOAS es en definitiva, una de las propiedades más elementales según las definiciones de las API REST y como tal, imprescindible en cualquier servicio REST, dada la encapsulación que profesa en las acciones a realizar de cara al cliente final.

El sistema REST, ha de basarse siempre en una estructura cliente-servidor que permita la comunicación sin estado entre el servidor y los clientes, lo que significa que todas las peticiones de los clientes al servidor se tratan de forma independiente a peticiones anteriores. El servicio debe estar estructurado en capas y utilizar las ventajas del caching HTTP (almacenamiento en caché) para que la utilización del servicio sea lo más sencilla posible para el cliente que realiza la petición. Por último, ha de tener una interfaz unitaria.

Ejercicio 7:

Implementar el método POST del api JsonPlaceHolder para crear nuevos posts desde nuestro sistema “RestApi”, dónde necesitaremos incluir un FloatingButton en la página de lista principal, cuya acción sea desplegar un Dialog o Modal que muestre tres campos para ingresar datos (los mismos que requiere el api) y un botón “ENVIAR” de acción.

1. Abrimos nuestro proyecto “RestApi”, donde crearemos un nuevo método POST en el archivo interfaz “Api.kt” de la siguiente forma:

```
@Headers("Content-Type: application/json; charset=UTF-8")
@POST("/posts")
fun createNewPost(@Body post: Post): Call<Post>
```

2. En nuestro archivo de layout activity_main.xml, agregamos una nueva vista de tipo FloatingButton entre el RecyclerView y la etiqueta de cierre del CoordinatorLayout.

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/add"
    android:layout_margin="30dp"
    android:layout_gravity="end|bottom"
    app:backgroundTint="#8ccff8"
/>
```

3. En nuestra actividad principal necesitaremos crear un nuevo método que realice la comunicación con retrofit y el api enviando los datos solicitados, para esto crearemos un nuevo método llamado addNewPostOnApi(), de la siguiente manera:

```
fun addNewPostOnApi(post: Post){
    val service = RetrofitClient.retrofitInstance()
    val call = service.createNewPost(post)
    //Async
    call.enqueue(object : Callback<Post> {
        override fun onResponse(call: Call<Post>, response: Response<Post>) {

            val responseCode = response.code()
            if(responseCode == 200 || responseCode == 201){
                Toast.makeText(applicationContext,
                    "Post creado exitosamente. Recuerda esto solo es una respuesta desde el servidor del api
público, tus datos no serán almacenados. ",Toast.LENGTH_LONG).show()
                Toast.makeText(applicationContext,
                    "Título: " +response.body()!!.title+ ", " +
                            "Descripción: "+response.body()!!.body+", " +
                            "UserId: "+response.body()!!.userId, Toast.LENGTH_LONG).show()
            }else{
                Toast.makeText(applicationContext, "Error al crear post. Código: "+responseCode,
Toast.LENGTH_LONG).show()
            }
        }

        override fun onFailure(call: Call<Post>, t: Throwable) {
            Log.d("MAIN", "Error: "+t)
            Toast.makeText(
                applicationContext,
                "Error: no pudimos recuperar los posts desde el api",
                Toast.LENGTH_SHORT
            ).show()
        }
    })
}
```

4. Necesitamos crear un nuevo archivo de layout llamado “new_post_dialog.xml”, que sirva de vista de ingreso de datos para crear el nuevo post, incluyendo tres vistas EditText y una Button. El código es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
                                android:layout_width="match_parent"
                                android:layout_height="wrap_content"
                                xmlns:app="http://schemas.android.com/apk/res-auto"
android:padding="20dp">

  <EditText
        android:id="@+id/titleJsonParam"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        android:textSize="16dp"
        android:hint="Ingresa un título..."
  />

  <EditText
        android:layout_marginTop="20dp"
           android:id="@+id/bodyJsonParam"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           app:layout_constraintTop_toBottomOf="@id/titleJsonParam"
           app:layout_constraintLeft_toLeftOf="parent"
           app:layout_constraintRight_toRightOf="parent"
           android:textSize="16dp"
           android:hint="Ingresa una descripción..."
  />

  <EditText
        android:layout_marginTop="20dp"
        android:id="@+id/userIdJsonParam"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/bodyJsonParam"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        android:textSize="16dp"
        android:hint="Ingresa tu usuarioId..."
        android:inputType="number"/>

     <Button
           android:layout_marginTop="20dp"
           android:id="@+id/btnSend"
           android:layout_width="match_parent"
           android:layout_height="50dp"
           android:background="@android:color/white"
           android:clickable="true"
           android:text="ENVIAR"
           android:textColor="#FFFFFF"
           android:textStyle="bold"
```

```
            android:backgroundTint="#1e88e5"
            app:layout_constraintTop_toBottomOf="@id/userIdJsonParam"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            />

</android.support.constraint.ConstraintLayout>
```

5. Teniendo ya el método que ejecuta la acción de crear contra el servidor api, es necesario llamar este método desde algún sitio, para ello vamos a crear un nuevo método que se llamará “showDialog()”, el cual mostrará un modal para permitirnos el ingreso de los datos del título, descripción y userId necesarios para crear el post en el api de JsonPlaceHolder.

```
private fun showDialog(){

    val dialog = Dialog(this)
    dialog.requestWindowFeature(Window.FEATURE_NO_TITLE)
    dialog.setCancelable(true)
    dialog.setContentView(R.layout.new_post_dialog)
    val title = dialog .findViewById(R.id.titleJsonParam) as TextView
    val body = dialog .findViewById(R.id.bodyJsonParam) as TextView
    val userId = dialog .findViewById(R.id.userIdJsonParam) as TextView
    val btnSend = dialog .findViewById(R.id.btnSend) as Button

    btnSend.setOnClickListener {
      val post = Post()

      try {
        post.title = title.text.toString()
        post.body = body.text.toString()
        val userIdStr = userId.text.toString()
        post.userId = userIdStr.toInt()
        addNewPostOnApi(post)
      }catch (e: Exception){
        Log.d("MAIN", "Error al crear post",e)
      }
      dialog .dismiss()
    }
    dialog .show()

  }
```

6. Por último en nuestro método de ciclo de vida de actividad onCreate(), agregaremos antes del llamado al método loadApiData(), la inicialización, instancia y declaración del botón Float de la siguiente forma:

```
val mFab = findViewById<FloatingActionButton>(R.id.fab)
mFab.setOnClickListener {
  showDialog()
}
```

7. Al iniciar nuestra app el resultado de las acciones y vistas creadas sería como lo demuestran las siguientes imágenes:

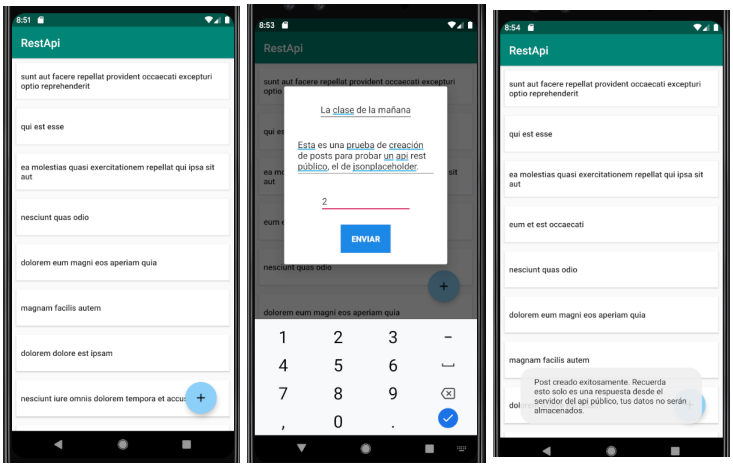

Imagen 2. Grupo de imágenes - Acción POST create post JsonPlaceHolder.

Ejercicio 8:

Ejercicio bonus de la unidad, agregar la funcionalidad swipeToRefresh a la actividad principal para refrescar los datos cargados del api.

1. En nuestro archivo de layout activity_main.xml, declararemos una nueva vista tipo widget que encierre a la vista RecyclerView ya existente entre sus etiquetas de la siguiente manera:

```
<android.support.v4.widget.SwipeRefreshLayout
    android:id="@+id/swipeRefreshLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="android.support.design.widget.AppBarLayout$ScrollingViewBehavior">

  <android.support.v7.widget.RecyclerView
      android:id="@+id/recyclerView"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical"
      app:layoutManager="android.support.v7.widget.LinearLayoutManager" />

</android.support.v4.widget.SwipeRefreshLayout>
```

2. En nuestro archivo de la actividad principal “MainActivity.kt” , vamos a implementar la clase SwipeRefreshLayout de esta forma:

```
class MainActivity : AppCompatActivity(), SwipeRefreshLayout.OnRefreshListener {
```

3. Requerimos implementar el método de la clase, onRefresh(), y lo haremos invocando al método loadApiData() del que ya disponemos. El código es el siguiente:

```
override fun onRefresh() {
  //WE CHECK INTERNET CONNECTION
  try{
     loadApiData()
  }catch(e: Exception){
     Toast.makeText(this, "Ocurrió un error, verifica tu conexión a internet",
Toast.LENGTH_LONG).show()
  }
}
```

4. Necesitamos declarar la variable de clase instancia de SwipeRefreshLayout así:

```
private lateinit var swipeRefreshLayout : SwipeRefreshLayout
```

5. A continuación, en nuestro método del ciclo de vida onCreate(), agregamos las instancia del objeto de la vista y definimos el listener en el contexto de nuestra actividad, justo después de la declaración del layout a través del método setContentView, de esta manera:

```
setContentView(R.layout.activity_main)

swipeRefreshLayout = findViewById(R.id.swipeRefreshLayout)
swipeRefreshLayout.setOnRefreshListener(this)
```

6. Por último dentro de nuestro método loadApiData(), justo al final indicamos que la acción del swipe to refresh ya no debe continuar:

```
swipeRefreshLayout.isRefreshing = false
```

7. Hemos finalizado, la funcionalidad se visualizará como una rueda de carga o spin en la parte superior de la pantalla de manera centrada, y al deslizar nuestro dedo desde arriba hacia abajo, volveremos a cargar los posts que nos retorna el api y se refrescan los datos en la pantalla; por ejemplo, podemos hacer el ejercicio de eliminar algunos posts (que sabemos no se eliminan realmente en una base de datos por ser api público) y refrescamos la data para observar cómo vuelven los elementos anteriormente eliminados a su posición.

Imagen 3. Acción SwipeRefreshLayout - Recarga de posts.

Imágenes con Picasso

Competencias:

- Conocer los conceptos asociados al uso de la librería Picasso.
- Implementar Picasso en aplicaciones android.
- Realizar caching de imágenes con Picasso.
- Conocer librerías alternativas para el manejo de imágenes.

Introducción

En este capítulo estudiaremos cómo se realiza la carga y descarga de imágenes utilizando la librería más común y popular de android “Picasso”. El éxito de esta librería se debe a su gran eficiencia en el manejo de los recursos gráficos y en su rendimiento, siendo la de mejor manejo de la memoria del dispositivo.

Es esencial aprender a utilizar Picasso, ya que el trabajo con imágenes en las aplicaciones móviles son recurrentes, por ejemplo en listas, detalles, iconos, entre otras vistas, siendo imprescindible que sepamos no sobrecargar el sistema y practicar los mejores códigos para estas implementaciones.

Picasso

Las imágenes agregan un contexto muy necesario y un toque visual a las aplicaciones de Android que hace que destaquen nuestros sistemas. Picasso permite la carga de imágenes sin problemas y de una manera muy sencilla, que en ocasiones, bastará con una línea de código.

Picasso fue creada con el propósito de mostrar imágenes de una manera rápida y sencilla simplificando el proceso que en años anteriores se tenía que hacer; además Picasso se encarga de la solicitud HTTP, el almacenamiento en caché de imágenes, y hasta depurar, por lo tanto, nos hará la vida más fácil a la hora de trabajar con imágenes en android.

Imagen 4. Logo Picasso.

Funcionalidades de Picasso

- **Resource Loading**: la más común es la descarga de recursos, assets, archivos (files), content providers son todos soportados como fuentes de imágenes..

```
Picasso.get().load(R.drawable.landing_screen).into(imageView1);
Picasso.get().load("file:///android_asset/DvpvklR.png").into(imageView2);
Picasso.get().load(new File(...)).into(imageView3);
```

- **Adapter Downloads:** la reutilización del adaptador se detecta automáticamente y se cancela la descarga anterior.

```
@Override public void getView(int position, View convertView, ViewGroup parent) {
  SquaredImageView view = (SquaredImageView) convertView;
  if (view == null) {
    view = new SquaredImageView(context);
  }
  String url = getItem(position);

  Picasso.get().load(url).into(view);
}
```

- **Image Transformations**: transforme las imágenes para adaptarse mejor a los diseños y para reducir el tamaño de la memoria.

```
Picasso.get()
  .load(url)
  .resize(50, 50)
  .centerCrop()
  .into(imageView)
```

- **Custom Image Transformations**: podemos especificar transformaciones con efectos más avanzados.

```
public class CropSquareTransformation implements Transformation {
  @Override public Bitmap transform(Bitmap source) {
    int size = Math.min(source.getWidth(), source.getHeight());
    int x = (source.getWidth() - size) / 2;
    int y = (source.getHeight() - size) / 2;
    Bitmap result = Bitmap.createBitmap(source, x, y, size, size);
    if (result != source) {
      source.recycle();
    }
    return result;
  }

  @Override public String key() { return "square()"; }
}
```

- **Place Holders**: Picasso soporta la descarga de placeholders y errores como opcionales.

```
Picasso.get()
    .load(url)
    .placeholder(R.drawable.user_placeholder)
    .error(R.drawable.user_placeholder_error)
    .into(imageView);
```

Caché de imágenes con Picasso

Como hemos indicado, dependemos mucho de las imágenes en nuestras aplicaciones. En algunos casos las imágenes suelen repetirse, como por ejemplo una imagen de perfil, la cual sería ideal guardar de alguna forma para no tener que recargar dicha imagen cada vez que se ingresa a la aplicación usando el tráfico de datos, que termina siendo costoso en servidores y alojamientos. Ejemplos de este trabajo de caché de imágenes está en las redes sociales más populares como lo son Facebook e Instagram.

Básicamente de esto se trata cuando hablamos de caché de imágenes. Librerías como Picasso se encargan de almacenarlas y recuperarlas de la memoria volátil, memoria interna o descargarla nuevamente si lo anterior no funciona, también es posible eliminar las imágenes cuando no se utilizan más.

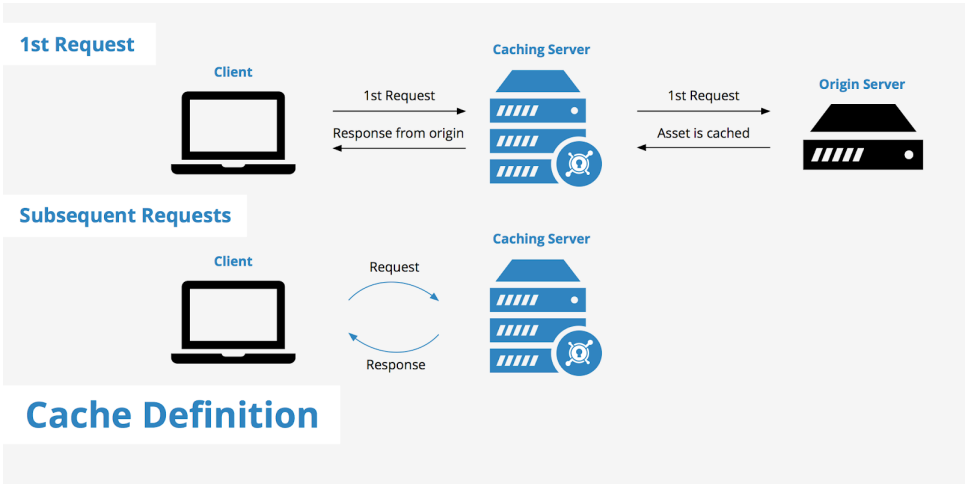

Imagen 5. Representación de funcionalidad de caché en arquitectura de red.

Ejercicio 8:

En nuestra aplicación “RestApi”, agregar una imagen descriptiva única para cada post. en el listado del ejercicio anterior, utilizando además las técnicas de caché de imagen, todo con Picasso.

1. Implementamos la librería de Picasso en nuestro archivo gradle app, con el siguiente código:

```
implementation ('com.squareup.picasso:picasso:2.71828') { exclude group: 'com.android.support'
exclude module: ['exifinterface', 'support-annotations'] }
```

Nota: se hace la exclusión de estos módulos de anotación dado que dan un error con nuestra librería de soporte appcompat-v7:28.0.0. Con el exclude evitamos el error. Esto puede ser resuelto con el tiempo y que hay que estar atento a los cambios en la documentación de la librería.

2. Usaremos la siguiente dirección web de una imágen para ser cargada con Picasso y asociarla a cada post de nuestra lista: https://cdn.pixabay.com/photo/2014/03/25/16/54/envelope-297570_960_720.png

3. Abrimos nuestro archivo layout “posts_list.xml” para agregar una vista de tipo ImageView, antes del TextView existente.

```
<ImageView
    android:id="@+id/imgPost"
    android:layout_width="60dp"
    android:layout_height="40dp"
    android:layout_marginTop="10dp"
  android:layout_marginRight="10dp"/>
```

4. Como la imágen compartirá la línea de la carta con el texto del título cambiaremos el RelativeLayout existente por un LinearLayout con orientación horizontal.

```
<LinearLayout
    android:orientation="horizontal"
    android:id="@+id/relativeLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="8dp"
    android:paddingLeft="8dp"
    android:paddingRight="8dp">
```

5. A continuación, modificaremos nuestra clase “CustomAdapter” para agregar la carga de la imagen con Picasso, en primer lugar creamos las instancia de la vista en la inner class “CustomViewHolder” ya existente, de esta forma:

```
inner class CustomViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
  val mTitle: TextView = itemView.findViewById(R.id.txtTitle)
  val mImgPost: ImageView = itemView.findViewById(R.id.imgPost)
}
```

6. Modificamos el método onBindViewHolder() para agregar el contenido de esta vista ImageView con Picasso de la siguiente manera:

```
override fun onBindViewHolder(holder: CustomViewHolder, position: Int) {

  holder.mTitle.text = data[position].title
  Picasso.get().load("https://cdn.pixabay.com/photo/2014/03/25/16/54/envelope-
297570_960_720.png").into(holder.mImgPost)
 }
```

7. Finalmente, debajo de la última línea de código agregado para la carga de Picasso, vamos a indicar un parámetro muy útil que nos hará saber si la imagen fue cargada a través de la red, de la memoria del dispositivo o del almacenamiento del dispositivo, con un color en la parte superior izquierda de la imagen que se vería de la siguiente forma:

Imagen 6. Representación de colores de origen de imágenes.

El código a agregar es el siguiente:

```
Picasso.get().setIndicatorsEnabled(true)
```

8. Como nos hemos dado cuenta, Picasso maneja casi automáticamente el caché de nuestras imágenes, decidiendo cómo manejarlas para mejorar el rendimiento, este es uno de los grandes potenciales de esta librería y el porqué es tan popular. Al correr la aplicación, la visualizamos de la siguiente manera:

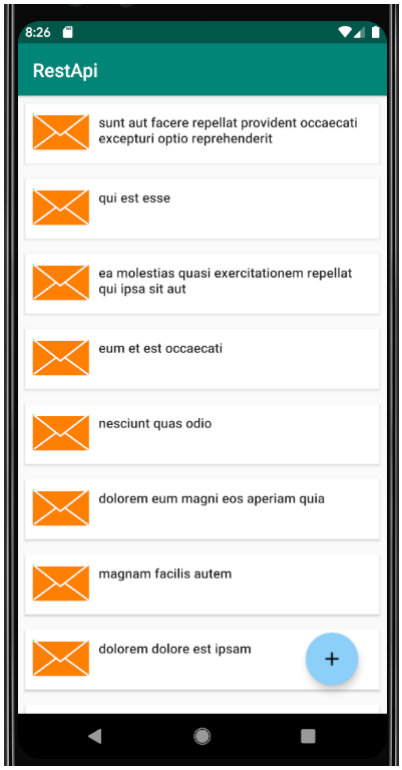

Imagen 7. Picasso load image en lista de posts con caché de memoria (verde).

Glide

Glide es un marco de carga de imágenes y gestión de medios de código abierto (open source) rápido y eficiente para Android, que ofrece una alternativa a Picasso. Glide envuelve la decodificación de medios, el almacenamiento en caché de memoria y disco, además de ofrecer la agrupación de recursos en una interfaz simple y fácil de usar.

Imagen 8. Logo Glide.

Glide admite la obtención (fetching) , decodificación (decode) y visualización de imágenes fijas de video (video stills), imágenes y GIF animados. Glide incluye una API flexible que permite a los desarrolladores conectarse a casi cualquier pila de red (Network stack). De forma predeterminada, Glide utiliza una pila basada en HttpUrlConnection personalizada, pero también incluye bibliotecas de utilidades conectadas al proyecto Volley de Google o la biblioteca OkHttp de Square.

El enfoque principal de Glide es hacer que el desplazamiento de cualquier tipo de lista de imágenes sea lo más suave y rápido posible, pero Glide también es efectivo para casi cualquier caso en el que necesite buscar, cambiar el tamaño y mostrar una imagen remota.

Implementar Glide en un proyecto android

Para agregar las librerías de Glide, lo hacemos de la siguiente forma en nuestro archivo gradle de app:

```
dependencies {
  implementation 'com.github.bumptech.glide:glide:4.10.0'
  annotationProcessor 'com.github.bumptech.glide:compiler:4.10.0'
}
```

Por último para implementar glide en nuestras aplicaciones, podemos seguir el siguiente ejemplo en una actividad:

```
// For a simple view:
@Override public void onCreate(Bundle savedInstanceState) {
  ...
  ImageView imageView = (ImageView) findViewById(R.id.my_image_view);

  Glide.with(this).load("http://goo.gl/gEgYUd").into(imageView);
}

// For a simple image list:
@Override public View getView(int position, View recycled, ViewGroup container) {
  final ImageView myImageView;
  if (recycled == null) {
    myImageView = (ImageView) inflater.inflate(R.layout.my_image_view, container, false);
  } else {
    myImageView = (ImageView) recycled;
  }

  String url = myUrls.get(position);

  Glide
    .with(myFragment)
    .load(url)
    .centerCrop()
    .placeholder(R.drawable.loading_spinner)
    .into(myImageView);

  return myImageView;
}
```

Fresco

Fresco es una biblioteca poderosa para mostrar imágenes en Android. Descarga y almacena en caché imágenes remotas de manera eficiente en la memoria, utilizando una región especial de memoria no recolectada en Android llamada ashmem.

Imagen 9. Logo Fresco.

Al trabajar con Fresco, es útil estar familiarizado con los siguientes términos:

- **ImagePipeline**: responsable de obtener la imagen. Se obtiene de la red, un archivo local, un proveedor de contenido o un recurso local. Mantiene un caché de imágenes comprimidas en el almacenamiento local, y un segundo caché de imágenes descomprimidas en la memoria.
- **Drawee**: los drawees se ocupan de representar imágenes en la pantalla y se componen de tres partes:
    - **DraweeView**: la vista que muestra la imagen. Se extiende desde ImageView, pero solo por conveniencia. La mayoría de las veces usaremos SimpleDraweeView en el código.
    - **DraweeHierarchy:** Fresco ofrece mucha personalización, podemos agregar una imagen de marcador de posición, una imagen de reintento, una imagen de falla, una imagen de fondo, etc. La jerarquía es la que hace un seguimiento de todos estos elementos dibujables y cuándo deben mostrarse.
    - **DraweeController**: esta es la clase responsable de tratar con el cargador de imágenes. Fresco nos permite personalizar el cargador de imágenes si no deseamos utilizar el ImagePipeline.

Implementar Fresco en un proyecto android

Para implementar la librería Fresco de Facebook en nuestro proyecto, es necesario agreguemos su dependencia en el archivo gradle app, de la siguiente manera:

```
dependencies {
 implementation 'com.facebook.fresco:fresco:1.13.0'
}
```

Inicializando Fresco:

```
Fresco.initialize(this)
```

SimpleDraweeView en reemplazo de ImageView en tu archivo layout:

```
<com.facebook.drawee.view.SimpleDraweeView
   android:id="@+id/posterImage"
   android:layout_width="match_parent"
   android:layout_height="500dp"
          ----------
   fresco:placeholderImage="@drawable/ic_broken_image" />
```

Para cargar imágenes el código es el siguiente:

```
posterImage : SimpleDraweeView = findViewById(R.id.posterImage)
imgURI : Uri = Uri.parse("
https://hogaraldia.cl/fit/c/240/240/1*SF2VIRFshYt2etl6OhNm_Q.png")
posterImage.setImageURI(imgURI)
```