CURSO DESARROLLO DE APLICACIONES MÓVILES ANDROID TRAINEE

Módulo 4.

Desarrollo de aplicaciones móviles Android Kotlin

AIEP OTEC EDUCACIÓN CONTINUA

Partner Laboratoria

TALENTO DIGITAL INTELIGENCIA HUMANA

Temario

Unidad 3

a) API REST

b) HTTP requests y threads

c) Buenas prácticas de autentificación en requests HTTP

d) Testing con Klotlin

Cliente - Servidor

Internet ha traído un cambio revolucionario en el ámbito de las tecnologías, interconectando a el mundo entero. En este contexto, las tecnologías de comunicación en internet siguen una arquitectura y una estructura específica para conectarse entre sí. La más popular es la arquitectura cliente-servidor, la cual profundizaremos en este capítulo.

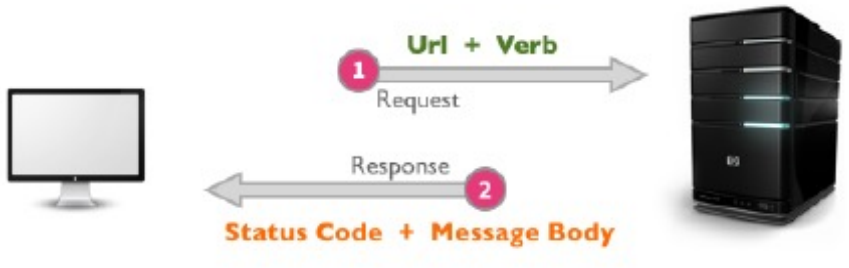
Es fundamental entender estos conceptos básicos de programación moderna, dado que hoy por hoy la gran mayoría de los sistemas y aplicaciones que se construyen siguen un standard de arquitectura basada en el concepto cliente-servidor.

Arquitectura Cliente - Servidor

La arquitectura cliente-servidor se denomina estructura de computación de red porque cada solicitud y sus servicios asociados se distribuyen a través de una red privada o pública en internet.

Arquitectura Cliente - Servidor

Los clientes a menudo se encuentran en estaciones de trabajo o en computadoras personales, mientras que los servidores se encuentran en otras partes de la red, generalmente son las máquinas más potentes. Este modelo de computación es especialmente efectivo cuando los clientes y el servidor tienen tareas distintas que realizan rutinariamente.



Cliente

En una arquitectura cliente-servidor de una red computacional el cliente se puede definir como un procesador remoto que recibe y envía peticiones de un servidor. Cada teléfono móvil android desde su ubicación en el planeta es un cliente para cada una de las empresas o servicios que prestan diferentes aplicaciones desde sus servidores principales.

Principales funciones del Cliente

- Iniciar el request o solicitud.
- Esperar y recibir respuestas.
- Poder conectarse con un servidor o múltiples servidores al mismo tiempo.
- Interactuar con los usuarios finales usando una Graphical User Interface (GUI)

Servidor

Un servidor es un computador de red, programa o dispositivo que procesa solicitudes de un cliente. En la internet hoy por hoy un servidor web es una computadora o una red de computadoras que utiliza el protocolo HTTP para enviar datos y archivos a la computadora de un cliente cuando el este último lo solicita.

Servidor

En el caso de una red de área local, por ejemplo, un servidor de impresoras administra una o más impresoras e imprime los archivos que le envían los equipos cliente. Los servidores de red (que administran el tráfico de red) y los servidores de archivos (que almacenan y recuperan archivos para clientes) son dos ejemplos mas de servidores.

Principales funciones de un Servidor

- Esperar por los requerimientos de un cliente.
- Recibido un requerimiento, procesar, preparar y enviar las respuestas.
- Aceptar múltiples conexiones de una largo número de clientes.
- Normalmente, no interactuar directamente con los usuarios finales.

Cliente-Servidor en la actualidad

En el mundo moderno con computadoras y servidores, los usuarios, que buscan el informe meteorológico a través de sus dispositivos móviles enviando peticiones a través de direcciones web (URL) que el servidor provee para la ejecución de un servicio o tarea específico, proporcionando información actualizada del clima.

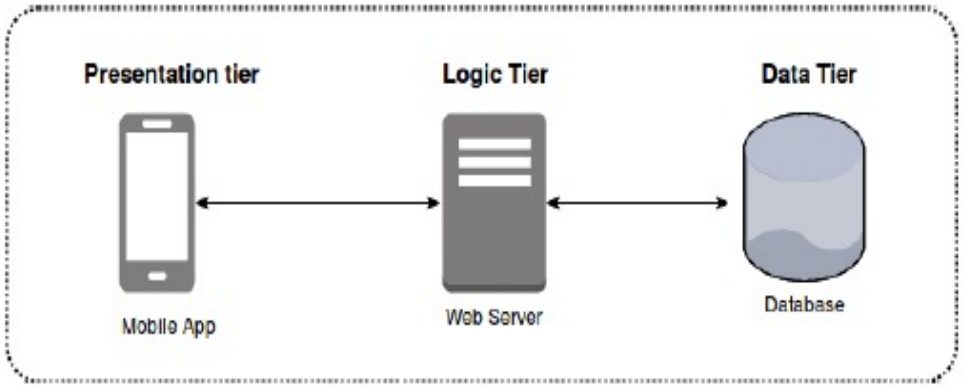
Cliente-Servidor en la actualidad

Algunos factores adicionales entran en acción con tales tecnologías de recopilación de datos en línea. Un periódico o radio usa su idioma local para darle el informe meteorológico y otra información; sin embargo, para la arquitectura cliente-servidor en la web, se deben considerar factores específicos:

- Un conjunto específico de idiomas junto con un estándar de comunicación, exclusivamente un protocolo para la interacción de dos sistemas. Los más populares son HTTP y HTTPS (Protocolo seguro de transferencia de hipertexto).

Cliente-Servidor en la actualidad

- Mecanismo y protocolo para solicitar los aspectos requeridos del servidor. Eso podría estar en cualquier estructura de datos formateados. Los formatos más implementados y populares se realizan en XML y JSON.
- Respuesta del servidor enviando en una estructura de datos formateados (generalmente XML o JSON).



API REST

Una api REST simplifica el desarrollo al admitir métodos HTTP estándar, manejo de errores y otras convenciones RESTful. Un api REST expone recursos a través de una o distintas direcciones web (URL) utilizando arquitectura basada en HTTP, exponiendo servicios web para el procesamiento de distintas tareas y operaciones.

REST es el acrónimo de Representational State Transfer, un estilo arquitectónico para sistemas hipermedia distribuidos que fue presentado por primera vez por Roy Fielding en el año 2000.

Principios REST

1. Cliente-servidor: al separar las preocupaciones de la interfaz de usuario de las preocupaciones de almacenamiento de datos, mejoramos la portabilidad de la interfaz de usuario en múltiples plataformas y mejoramos la escalabilidad al simplificar los componentes del servidor.

2. Sin estado (Stateless): cada solicitud del cliente al servidor debe contener toda la información necesaria para comprender la solicitud y no puede aprovechar ningún contexto almacenado en el servidor. Por lo tanto, el estado de la sesión se mantiene completamente en el cliente.

Principios REST

3. Caché: las restricciones de caché requieren que los datos dentro de una respuesta a una solicitud se etiqueten implícita o explícitamente como almacenables o no almacenables. Si una respuesta es almacenable en caché, se le otorga a un caché de cliente el derecho de reutilizar esos datos de respuesta para solicitudes equivalentes posteriores.

4. Interfaz uniforme: al aplicar el principio de generalidad de ingeniería de software a la interfaz del componente, se simplifica la arquitectura general del sistema y se mejora la visibilidad de las interacciones.

Principios REST

Para obtener una interfaz uniforme, se necesitan múltiples restricciones arquitectónicas para guiar el comportamiento de los componentes. REST está definido por cuatro restricciones de interfaz:

- Identificación de recursos.
- Manipulación de recursos a través de representaciones.
- Mensajes autodescriptivos.
- Hipermedia como motor del estado de la aplicación.

Principios Rest

5. Sistema en capas: el estilo de sistema en capas permite que una arquitectura se base en capas jerárquicas al restringir el comportamiento de los componentes de modo que cada componente no pueda ver más allá de la capa inmediata con la que están interactuando.

6. Código bajo demanda (opcional): REST permite ampliar la funcionalidad del cliente descargando y ejecutando código en forma de scripts. Esto simplifica a los clientes al reducir la cantidad de funciones que se deben implementar previamente.

REST - CRUD

REST significa REpresentational State Transfer y describe recursos (en nuestro caso, URL) en los que podemos realizar acciones.

CRUD significa Crear, Leer, Actualizar, Eliminar, son las acciones que realizamos. Aunque, REST y CRUD son los mejores amigos, los dos pueden funcionar bien por sí mismos. De hecho, cada vez que programamos un sistema que permite agregar, editar y eliminar elementos de la base de datos, y una interfaz que permite que viajen los datos de estos elementos; hemos estado trabajando con CRUD sin saberlo.

REST - CRUD

REST significa REpresentational State Transfer y describe recursos (en nuestro caso, URL) en los que podemos realizar acciones.

CRUD significa Crear, Leer, Actualizar, Eliminar, son las acciones que realizamos. Aunque, REST y CRUD son los mejores amigos, los dos pueden funcionar bien por sí mismos. De hecho, cada vez que programamos un sistema que permite agregar, editar y eliminar elementos de la base de datos, y una interfaz que permite que viajen los datos de estos elementos; hemos estado trabajando con CRUD sin saberlo.

HTTP Response Codes

Si intentamos acceder a una página que no existe, el servidor te devolverá la página 404 No encontrada o error 404, si arruinamos nuestro código rest del servidor, obtendremos un error 500 de sistemas. Estos son códigos de respuesta que el servidor envía para que su navegador sepa lo que está sucediendo.

Tipos de respuestas

- Entre 200-299 significa que la solicitud fue exitosa.
- 300-399 significa que la solicitud estuvo bien, pero debe hacer otra cosa.
- 400-499 es un error dónde no se encuentra algún objeto o recurso, o no se ha invocado correctamente.
- 500-599 es un error realmente malo.

Tipos de respuestas

- Código 200: Obtendrá de una solicitud GET, PUT o DELETE. Significa que la solicitud se verificó y se tomaron las medidas apropiadas.
- Código 201 Creado: Le notifica que el comando POST creó correctamente el objeto publicado.
- Código 404 No encontrado: Significa que no se encontró el recurso de solicitud. Puede obtener esto de GET, PUT o DELETE.

Tipos de respuestas

- Código 406 no aceptable: El verbo no está permitido en los recursos que solicitó (Más en esta próxima parte)
- Código 500 Error interno de servidor: Algo sucedió terriblemente malo en el código del sistema

Operaciones REST

HTTP define una gran cantidad de métodos que son utilizados en diferentes circunstancias. Los más relevantes para la construcción de servicios REST son los siguientes:

- GET: Es utilizado únicamente para consultar información al servidor, muy parecidos a realizar un SELECT a la base de datos. No soporta el envío del payload.
- POST: Es utilizado para solicitar la creación de un nuevo registro, es decir, algo que no existía previamente, es decir, es equivalente a realizar un INSERT en la base de datos. Soporta el envío del payload.

Operaciones REST

- PUT: Se utiliza para actualizar por completo un registro existente, es decir, es parecido a realizar un UPDATE a la base de datos. Soporta el envío del payload.
- PATCH: Este método es similar al método PUT, pues permite actualizar un registro existente, sin embargo, este se utiliza cuando actualizar solo un fragmento del registro y no en su totalidad, es equivalente a realizar un UPDATE a la base de datos. Soporta el envío del payload.

Operaciones REST

- DELETE: Este método se utiliza para eliminar un registro existente, es similar a DELETE a la base de datos. No soporta el envío del payload.
- HEAD: Este método se utilizar para obtener información sobre un determinado recurso sin retornar el registro. Este método se utiliza a menudo para probar la validez de los enlaces de hipertexto, la accesibilidad y las modificaciones recientes.

Operaciones REST

Cualquier desarrollador web que haya tenido que lidiar con datos de formularios conocerá los métodos GET y POST. El primero enviará datos al servidor a través de una cadena de consulta que se parece a esto “direccion?Key2=value2&key2=value2” y el segundo envía los datos a través de encabezados HTTP. Lo que quizás no nos hemos dado cuenta es que cada vez que carga una página (que no es una forma de tipo POST) está realiza una solicitud de tipo GET.

REST en Android

Mientras desarrollamos aplicaciones android, regularmente nos encontramos trabajando con un back-end a través de las api REST. Estas api pueden ser desarrolladas por un equipo interno o por terceros.

A continuación comenzaremos los ejercicios interactuando desde nuestro entorno de desarrollo de aplicaciones android studio (IDE) con el api público de JsonPlaceHolder ya que dispone de diferentes métodos que nos serán útiles a lo largo de esta unidad.

Ventajas de usar REST

- Separación entre el cliente y el servidor.
- Visibilidad, fiabilidad y escalabilidad.
- La API REST siempre es independiente del tipo de plataformas o lenguajes.

Retrofit

Es una librería Http client para específicamente realizar peticiones http tipo rest para android y java, permitiendo el envío y recepción de respuestas de datos entre nuestras aplicaciones y distintas plataformas de software a través de internet.

Retrofit

Para realizar una petición con Retrofit se necesitará lo siguiente:

- Una clase Retrofit: donde se cree la instancia a la librería de retrofit para utilizar sus métodos base, definiendo aquí nuestra url base que nuestra aplicación usará para todas las peticiones http.
- Una interfaz de operaciones: en la cual definamos todas nuestras peticiones con sus respectivos endpoints y datos de entrada o salida.
- Clases de modelo de datos o pojo: para mapear los datos que llegan desde el servidor traspasando estos a objetos e instancias automáticamente para ser usados en nuestra aplicación.

GSON Converter

Habitualmente, las peticiones y respuestas son intercambiadas en formato JSON, razón por la cual se debe utilizar un convertidor junto con retrofit para lidiar con la serialización de datos en este formato. Retrofit puede deserializar por defecto respuestas de tipo OkHttp, pero admite el uso de convertidores como gson.

Gson es una biblioteca de java que nos permite convertir objetos java a json y viceversa de manera automática, sin mayores configuraciones.

Características de Retrofit como cliente HTTP

Declaración de API: las anotaciones sobre los métodos de interfaz y sus parámetros indican cómo se maneja una solicitud.

Método request (solicitud): cada método debe tener una anotación HTTP que proporcione el método de solicitud y la URL relativa. Hay cinco anotaciones incorporadas: GET, POST, PUT, DELETE, y HEAD. La URL relativa del recurso se especifica en la anotación. Ejemplo:

```
@GET("posts/list")
```

JSON

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Es fácil para los humanos leer y escribir. Es fácil para las máquinas analizar y generar. Se basa en un subconjunto del lenguaje de programación denominados JavaScript Standard ECMA-262.

JSON

JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son familiares para los programadores de la familia de lenguajes C, incluido Kotlin, C, C ++, C#, Java, JavaScript, Perl, Python y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

Estructura de un JSON

JSON se basa en dos estructuras:

- Una colección de pares de nombre / valor, en distintos idiomas y lenguajes de programación; esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista con clave o arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes de programación, esto se implementa como arreglos, vectores, listas o secuencias.

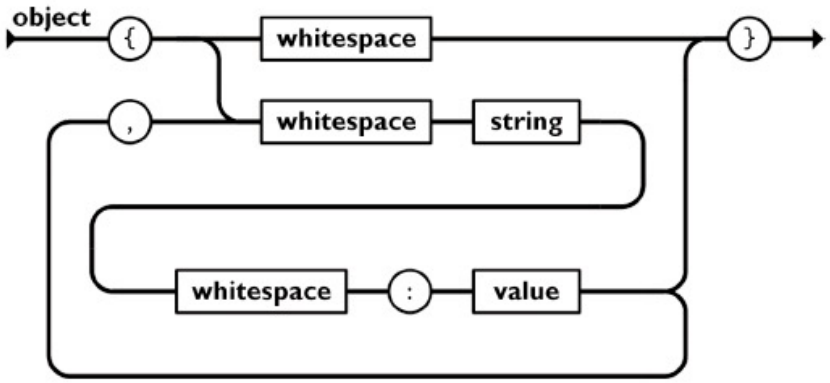
Estructura de un JSON

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

En JSON, se presentan de estas formas:

Un objeto es un conjunto desordenado de pares de nombre/valor. Un objeto comienza con {llave izquierda y termina con} llave derecha. Cada nombre es seguido por: dos puntos y los pares de nombre/valor están separados por una coma.

Estructura de un JSON



JSON – REST HEADER

Los headers y parámetros REST contienen una gran cantidad de información que puede ayudarlo a localizar problemas cuando los encuentre. Los encabezados HTTP son una parte importante de la solicitud y respuesta de la API, ya que representan los metadatos asociados con la solicitud y la respuesta de la API.

JSON – REST HEADER

Los encabezados llevan información para:

- Request and Response Body (Cuerpos de datos en solicitudes y respuestas)
- Request Authorization (Solicitar autorización)
- Caché de respuesta
- Cookies de respuesta

JSON – REST HEADER

Los encabezados con los que nos encontraremos más durante el trabajo con un API son los siguientes:

- Authorization: lleva credenciales que contienen la información de autenticación del cliente para el recurso que se solicita.
- WWW-Authenticate: el servidor lo envía si necesita una forma de autenticación antes de que pueda responder con el recurso real que se solicita. A menudo se envía junto con un código de respuesta de 401, que significa "no autorizado".

JSON – REST HEADER

- Accept-Charset: este es un encabezado que se establece con la solicitud y le dice al servidor qué conjuntos de caracteres son aceptables para el cliente.
- Content-Type: indica el tipo de medio (texto / html o texto / JSON) de la respuesta enviada al cliente por el servidor, esto ayudará al cliente a procesar el cuerpo de la respuesta correctamente.
- Cache-Control: esta es la política de caché definida por el servidor para esta respuesta, el cliente puede almacenar una respuesta en caché y volver a usar hasta el momento definido por el encabezado de Control de caché.

RetroFit - JSON HEADER

En retrofit para asignar parámetros a nuestros métodos http de las interfaces, basta con agregar la anotación @Headers sobre la anotación del tipo de método http, ya sea get, post, put, delete.

Ejemplo:

```
@Headers("Content-Type: application/json; charset=UTF-8")
```

RetroFit - JSON HEADER

El cuerpo de un objeto json, no requiere mayor explicación de la que ya hemos dado, el cuerpo json (body) es ese objeto json que puede anexarse a una solicitud (request) o recibirse en la respuesta del servicio. Un ejemplo de un cuerpo json sería el siguiente:

```
{
  "customers":
  {
   "firstName": "Joe",
   "lastName": "Bloggs",
   "fullAddress":
   {
      "streetAddress": "21 2nd Street",
      "city": "New York",
      "state": "NY",
      "postalCode": 10021
   }
  }
}
```

RetroFit - JSON BODY

En kotlin utilizando retrofit podemos disponer de una anotación @Body que se utiliza para asignar un objeto tipo pojo (clase que contiene instancias de los datos similares a los que contiene el objeto json) a una respuesta o solicitud. Por ejemplo, anteponiendo los siguientes ejercicios, así sería una asignación de un método Post en la interfaz de retrofit para enviar un nuevo post al servidor:

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

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.

Restfull

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.

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.

Características de un sistema Restfull

- 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: e paradigma hipermedia

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.

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.

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);
```

Funcionalidades de Picasso

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);
}
```

Funcionalidades de Picasso

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)
```

Funcionalidades de Picasso

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()"; }
}
```

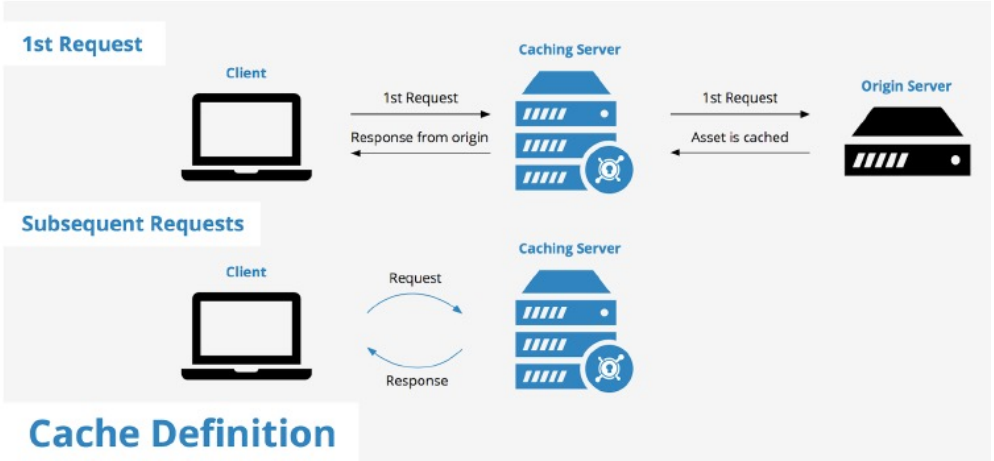
Funcionalidades de Picasso

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);
```

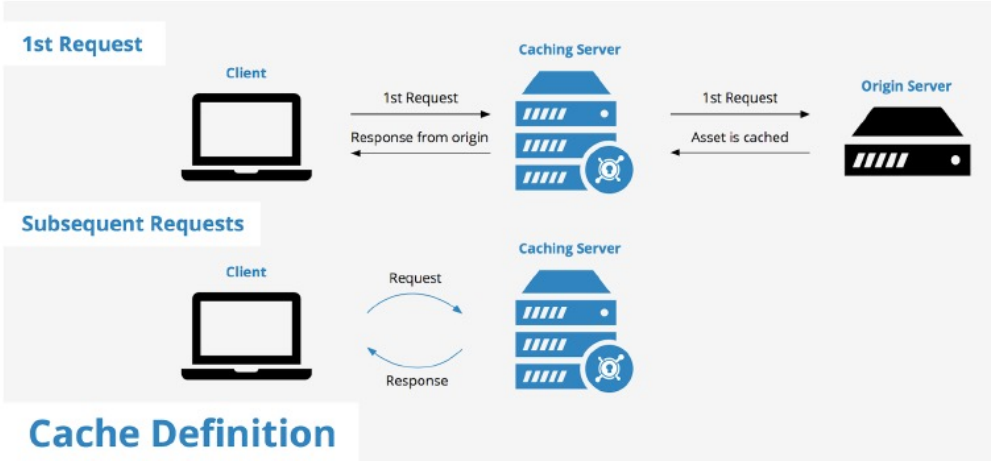
Cache de imágenes con Picasso

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.



Cache de imágenes con Picasso

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.



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.

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.

Glide

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'
}
```

Implementar Glide en un proyecto Android

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);
}
```

Implementar Glide en un proyecto Android

```
// 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;
 }
```

Implementar Glide en un proyecto Android

```
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.

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:

Fresco

- 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'
}
```

Iniciando Fresco:

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

Implementar Fresco en un proyecto Android

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" />
```

Implementar Fresco en un proyecto Android

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)
```