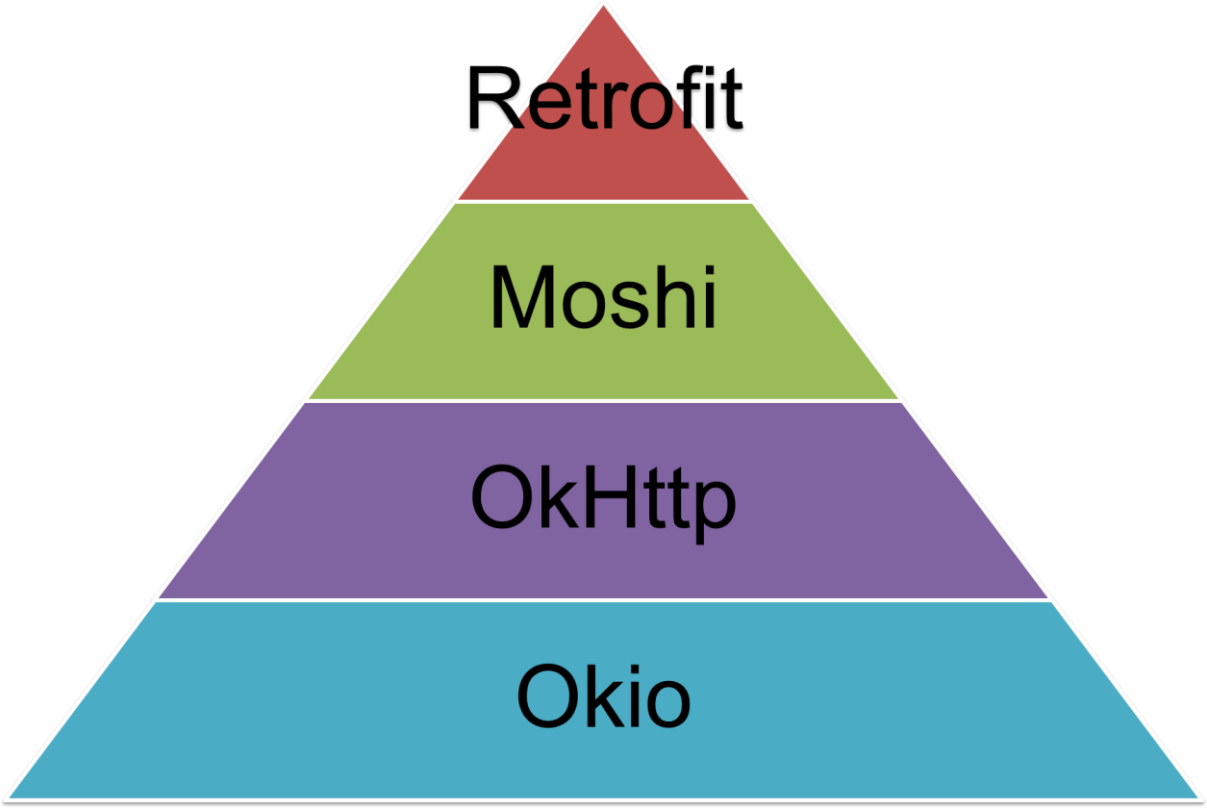
Tutoriel Android : Retrofit, OkIo, OkHttp et Moshi

L'objectif de ce tutoriel est de vous expliquer l'ensemble des composants qui constituent RetroFit. Vous aurez ainsi une compréhension profonde et complète de sa structure et de son utilisation. Nous verrons ainsi OkIo la brique de base qui sert à la lecture et à l'écriture des données (que ce soit dans un fichier ou un socket). Nous verrons OkHttp qui permet la mise en place de la communication http et nous ferons un détour par Moshi qui est le convertisseur Json naturel à utiliser avec Retrofit pour convertir vos chaînes de caractères JSon en Objet et vis-versa.



Les versions dont parle ce tutoriel sont Retrofit 2.0, OkHttp 3, Okio 1.6 et Moshi 1.1.

Cet article est fortement inspiré des conférences données par Jake Wharton à Montréal et à New York en 2015 : Disons simplement que je n'ai rien inventé.

https://www.youtube.com/watch?v=WvyScM_S88c Jake Wharton DroidCon Montréal 2015

https://www.youtube.com/watch?v=KIAoQbAu3eA Jake Wharton DroidCon NewYork 2015

Table des matières

1 Okio 1.6 : Lecture/écriture facile de fichiers et de flux

Okio est la brique élémentaire, elle permet la lecture/écriture des données.

1.1 Build Gradle

Il vous suffit de rajouter la dépendance vers Okio dans votre fichier gradle.build :

```
dependencies {
   compile fileTree(dir: 'libs', include: ['*.jar'])
   compile 'com.android.support:appcompat-v7:22.0.0'
   compile 'com.squareup.okio:okio:1.6.0'
}
```

1.2 Pourquoi JavaIo est désespérant ?

Prenons l'exemple de la lecture d'un flux de données provenant d'un socket. Pour effectuer cette opération, il faut utiliser un ByteArrayInputStream qui possède une taille fixe de Buffer. C'est l'enfer à analyser et à parser, les données n'étant pas formatées pour respecter la taille du dit Buffer. Ainsi certains blocs de données se retrouvent sur deux « Buffers différents ».

En essayant d'être plus malin que le ByteArrayInputStream, on se dit que l'on va utiliser un pointeur pour analyser les données. Certaines données dépassent toujours du Buffer, on va alors augmenter sa taille à la volée…. Ce qui amène un processus complexe et non adapté.

On peut aussi essayer d'utiliser des décorateurs pour s'aider, mais le DataInputStream ne parse pas le texte, l'InputStreamReader lui ne parse pas les objets et utiliser les deux en même temps n'est pas non plus possible car les Buffers sur lesquels ils s'exécutent sont disctincts.

On peut en conclure que, pour la lecture d'un flux, l'InputStream :

- Possède un comportement non adapté, voir inconsistant,
- Oblige l'utilisateur à déterminer son besoin de stockage (doubler la taille du Buffer en cours de parsing en fonction des données reçues),
- Peu flexible et adaptable.

C'est ainsi que Jesse Wilson, Jake Wharton ont décidé de créer une librairie spécifique pour effectuer une lecture/écriture de données plus pertinentes que celle fournie par le système :

Bienvenu à Okio.

1.3 L'interface Source pour lire les données

Une interface simple et épurée pour la lecture des données avec seulement trois méthodes.

```
public interface Source extends Closeable {
  /**
   * Removes at least 1, and up to {@code byteCount} bytes from this and appends
   * them to {@code sink}. Returns the number of bytes read, or -1 if this
  * source is exhausted.

   */
  long read(Buffer sink, long byteCount) throws IOException;
```

```
  /** Returns the timeout for this source. */
  Timeout timeout();
  /**

   * Closes this source and releases the resources held by this source. It is an
   * error to read a closed source. It is safe to close a source more than once.
   */
  @Override void close() throws IOException;
}
```

Il n'y a qu'une seule méthode de lecture, read, qui possède en paramètre le Buffer. Celui-ci peut ainsi être partagé entre plusieurs sources de lecture et/ou d'écriture. Et ça s'est tout simple, mais c'est une révolution pour la lecture de nos fichiers.

Le paramètre timeout permet de récupérer le timeout de la source.

Et enfin, la méthode close pour clore la source.

1.4 L'interface Sink pour écrire les données

Une interface simple et épurée, encore, avec uniquement quatre méthodes.

```
public interface Sink extends Closeable, Flushable {
  /** Removes {@code byteCount} bytes from {@code source} and appends them to this. */
  void write(Buffer source, long byteCount) throws IOException;

  /** Pushes all buffered bytes to their final destination. */
  @Override void flush() throws IOException;

  /** Returns the timeout for this sink. */
  Timeout timeout();

  /**
   * Pushes all buffered bytes to their final destination and releases the
   * resources held by this sink. It is an error to write a closed sink. It is
   * safe to close a sink more than once.
   */
  @Override void close() throws IOException;
}
```

Le design étant le même que pour la classe source, il n'y a qu'une méthode d'écriture qui possède en paramètre le Buffer sur lequel elle écrit (et peut ainsi le partager avec d'autres Source et Sink).

La méthode flush purge le Buffer et envoie toutes les données à leur destination finale.

Les méthodes timeout et close sont identiques à celles de l'objet Source.

1.5 Relation entre Sink et Source

Ces objets, Sink et Source, ne font que déplacer les données, ainsi Source lit à partir d'un Sink (déplace le flux sortant vers un flux entrant de lecture) et Sink écrit les données à partir d'un source (déplace le flux entrant vers un flux sortant).

En effet, la méthode read de la classe Buffer est la suivante :

```
@Override
public long read(Buffer sink, long byteCount) {
  if (sink == null) throw new IllegalArgumentException("sink == null");
  if (byteCount < 0) throw new IllegalArgumentException("byteCount < 0: " + byteCount);
  if (size == 0) return -1L;
  if (byteCount > size) byteCount = size;
  sink.write(this, byteCount);
  return byteCount;
}
```

Je ne mets pas l'exemple de la méthode write, il est monstrueusement plus complexe.

Moi, aussi, j'ai beaucoup relu cette phrase avant de la comprendre et je ne l'ai comprise qu'après avoir compris l'objet Buffer et surtout après avoir compris qu'on les utilisait avec la classe Buffer qui implémente BufferedSink et BufferedSource qui eux-mêmes étendent Sink et Source… Bon bref, il vous reste un peu de lecture.

1.6 La classe Buffer, un simple pointeur

C'est la classe à comprendre et qui explique le fonctionnement de Okio.

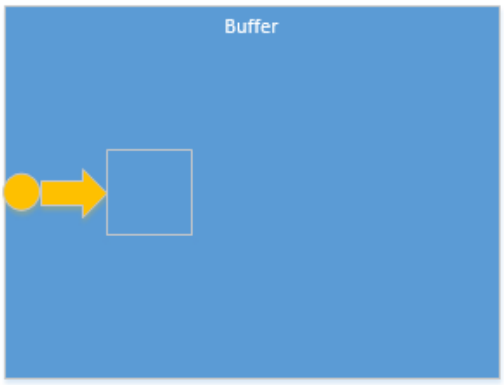
1.6.1 Principes approfondis

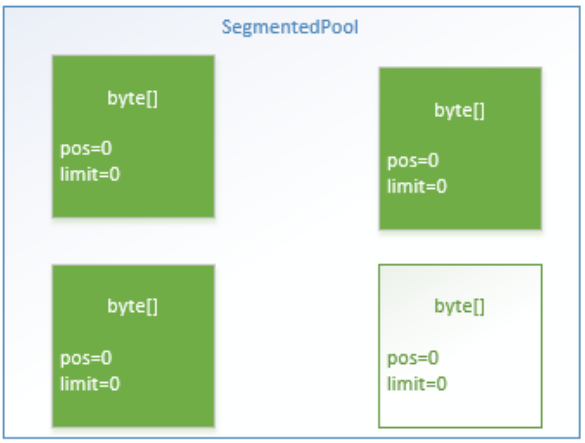
Nous allons comprendre la classe Buffer par l'exemple, pas à pas.

L'instanciation d'un Buffer ne fait que créer un Objet qui pointe vers un espace mémoire. En interne il est associé avec un SegmentedPool (un pool de byte[]) qu'il utilise pour lire et écrire ses données. Pour l'exemple que nous déroulons, nous supposons que ce pool possède des tableaux de 32 bytes.

Ainsi

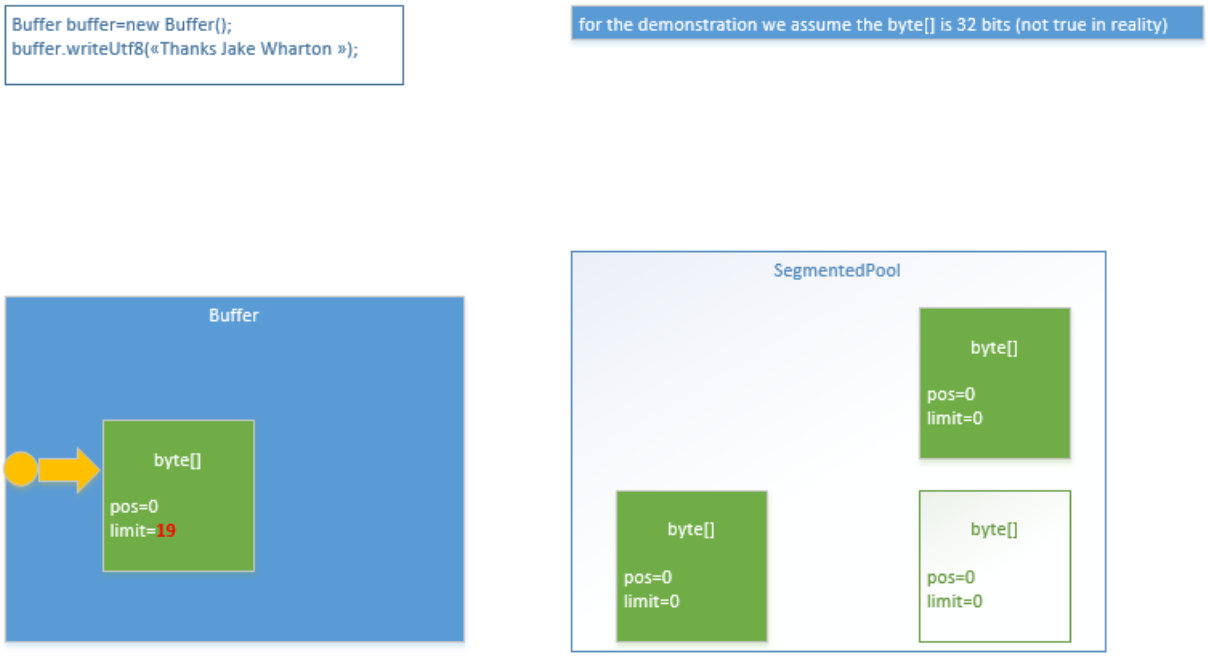
Buffer buffer = new Buffer();
Ne fait qu'allouer le Buffer qui alloue son pointeur, rien de plus.





Maintenant écrivons dans le Buffer :

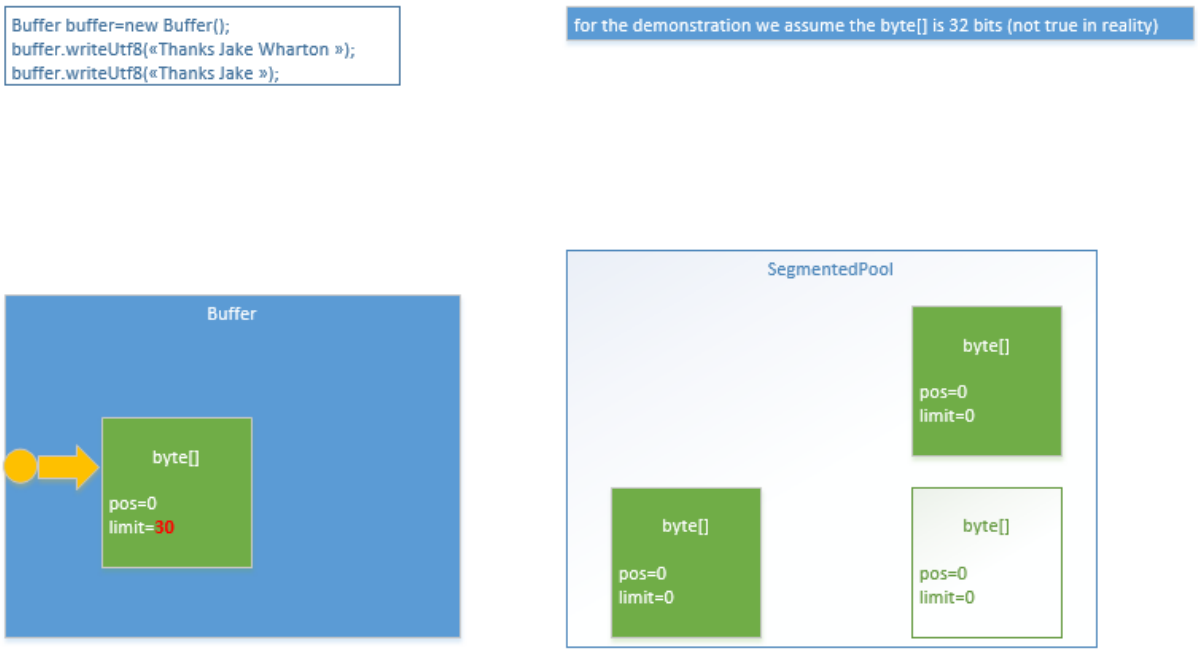
```
Buffer buffer = new Buffer();
buffer.writeUtf8("Thanks Jake Wharton");
```



Le Buffer a récupéré un tableau de bytes du SegmentedPool et l'a utilisé pour écrire les données dedans. Puis il a incrémenté son attribut limit pour lui donner la valeur 19 qui est le nombre de caractères insérés.

Continuons d'écrire dans le Buffer :

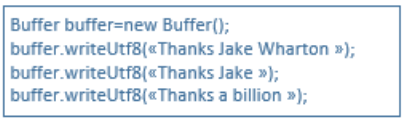
```
Buffer buffer = new Buffer();
buffer.writeUtf8("Thanks Jake Wharton");
buffer.writeUtf8("Thanks Jake");
```

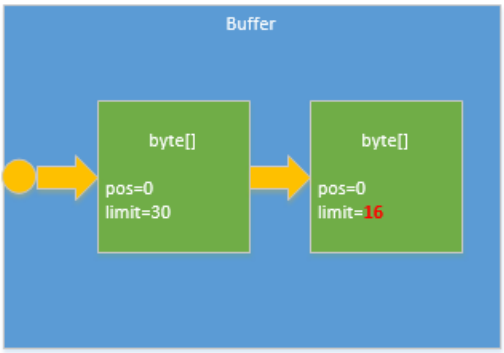


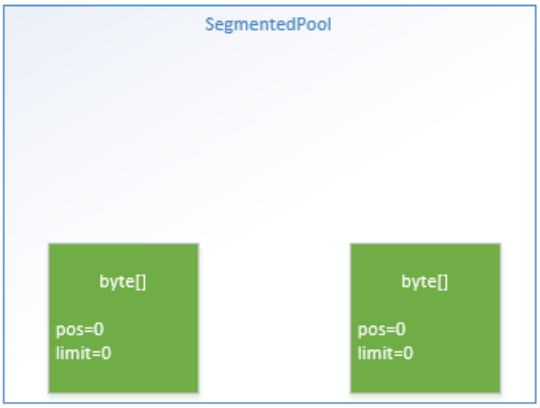
Le Buffer incrémente son attribut limit avec le nombre de caractères insérés. Cette limite ne dépassant pas la taille du tableau (32), rien ne se passe.

Continuons d'écrire dans le Buffer :

```
Buffer buffer = new Buffer();
buffer.writeUtf8("Thanks Jake Wharton");
buffer.writeUtf8("Thanks Jake");
buffer.writeUtf8("Thanks a billion");
```







Le Buffer regarde la chaîne qu'il doit écrire et s'aperçoit que cela dépasse la taille du tableau de bytes vers lequel il pointe, du coup il récupère un nouveau tableau au sein du SegmentedPool et écrit dans ce dernier.

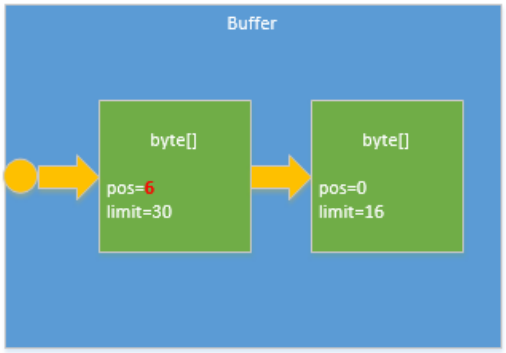
Voilà pour les principes d'écriture, abordons la lecture.

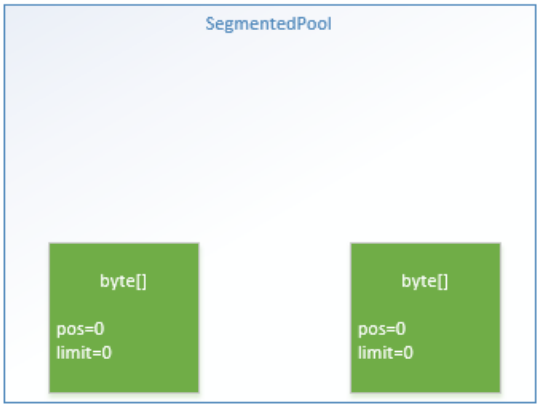
Maintenant, lisons les 6 premiers caractères du Buffer :

```
Buffer buffer = new Buffer();
buffer.writeUtf8("Thanks Jake Wharton");
buffer.writeUtf8("Thanks Jake");
buffer.writeUtf8("Thanks a billion");
buffer.readUtf8(6);//returns Thanks
```

```
Buffer buffer=new Buffer();
buffer.writeUtf8(«Thanks Jake Wharton »);
buffer.writeUtf8(«Thanks Jake »);
buffer.writeUtf8(«Thanks a billion »);
buffer.utf8(6);//returns Thanks
```

for the demonstration we assume the byte[] is 32 bits (not true in reality)





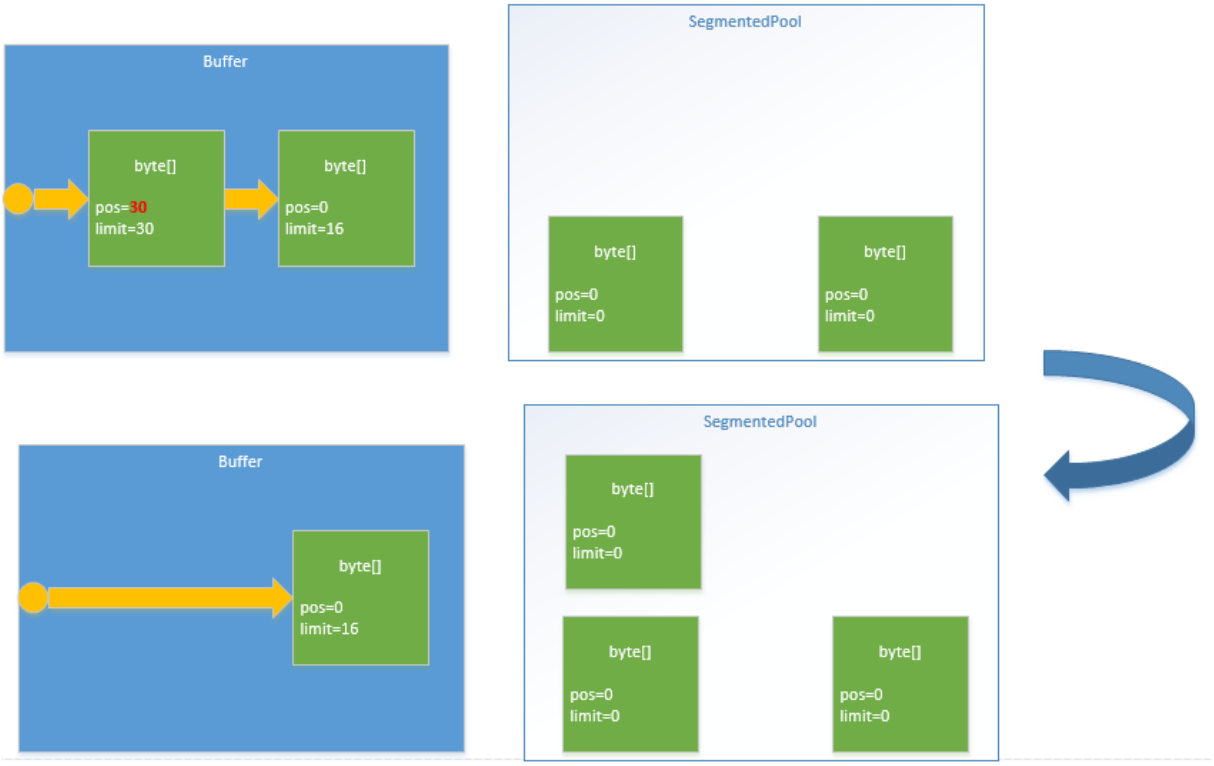
La valeur retournée par cette lecture est « Thanks » et le Buffer incrémente son attribut position, qui est sa position courante pour la lecture.

Continuons la lecture :

```
Buffer buffer = new Buffer();
buffer.writeUtf8("Thanks Jake Wharton");
buffer.writeUtf8("Thanks Jake");
buffer.writeUtf8("Thanks a billion");
buffer.readUtf8(6);//returns Thanks
//read the rest of the first segment
buffer.readUtf8(24);//returns Jake WhartonThanks Jake
```

```
Buffer buffer=new Buffer();
buffer.writeUtf8(«Thanks Jake Wharton »);
buffer.writeUtf8(«Thanks Jake »);
buffer.writeUtf8(«Thanks a billion »);
buffer.readUtf8(6);//returns Thanks
//read the rest of the first segment
buffer.readUtf8(24);//returns Jake WhartonThanks Jake
```

for the demonstration we assume the byte[] is 32 bits (not true in reality)



La lecture renvoie « Jake WhartonThanks Jake » et l'attribut est incrémentait jusqu'à 30. Le Buffer réalise que sa position et sa limite sont égaux. Il considère donc que le premier tableau est consumé et le restitue au SegmentedPool.

C'est une des premières et grandes optimisations qu'effectue Okio, en effet, il ne réalloue pas de mémoire ni n'en désalloue. L'écriture et la lecture est transparente pour l'utilisateur de l'api, la gestion des tableaux de bytes étant effectuée par la classe Buffer sans que l'utilisateur ait à s'en soucier.

Abordons maintenant, l'un des points dont M Wharton est le plus content, qui est le partage de cette mémoire entre Buffer. Imaginons que vous souhaitiez à ce stade changer de lecteur de flux. Si vous étiez en train de travailler avec Javalo, vous auriez dû réallouer de la mémoire, copier le contenu du premier InputStream dans le second et lire le second. Bref du gaspillage de mémoire et de cpu.

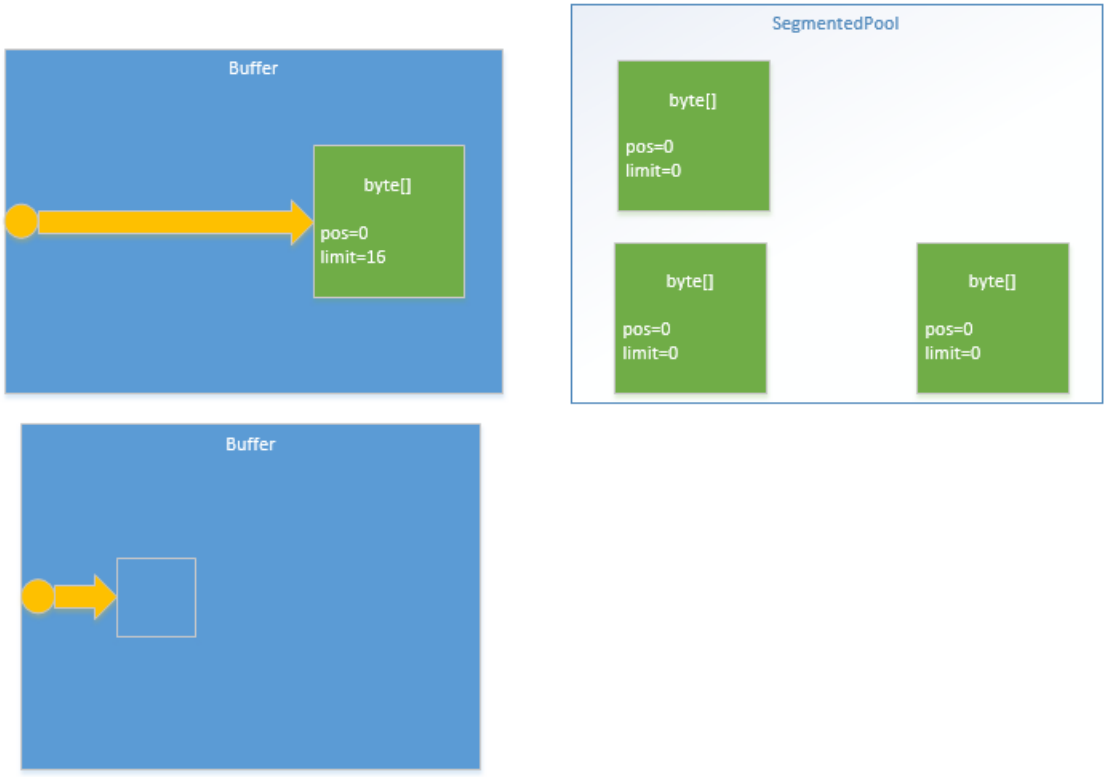
Avec Okio, tout est optimisé, si vous déclarez un nouveau Buffer pour récupérer le flux, aucune nouvelle allocation de mémoire n'est effectuée, seul un changment de pointeur est fait :

```
Buffer buffer = new Buffer();
buffer.writeUtf8("Thanks Jake Wharton");
buffer.writeUtf8("Thanks Jake");
buffer.writeUtf8("Thanks a billion");
buffer.readUtf8(6);//returns Thanks
//read the rest of the first segment
buffer.readUtf8(24);//returns Jake WhartonThanks Jake
```

//then create a new buffer
Buffer otherBuffer = new Buffer();

```
Buffer buffer=new Buffer();
buffer.writeUtf8(«Thanks Jake Wharton »);
buffer.writeUtf8(«Thanks Jake »);
buffer.writeUtf8(«Thanks a billion »);
buffer.readUtf8(6);//returns Thanks
//read the rest of the first segment
buffer.readUtf8(24);//returns Jake WhartonThanks Jake
Buffer otherBuffer =new Buffer()
```

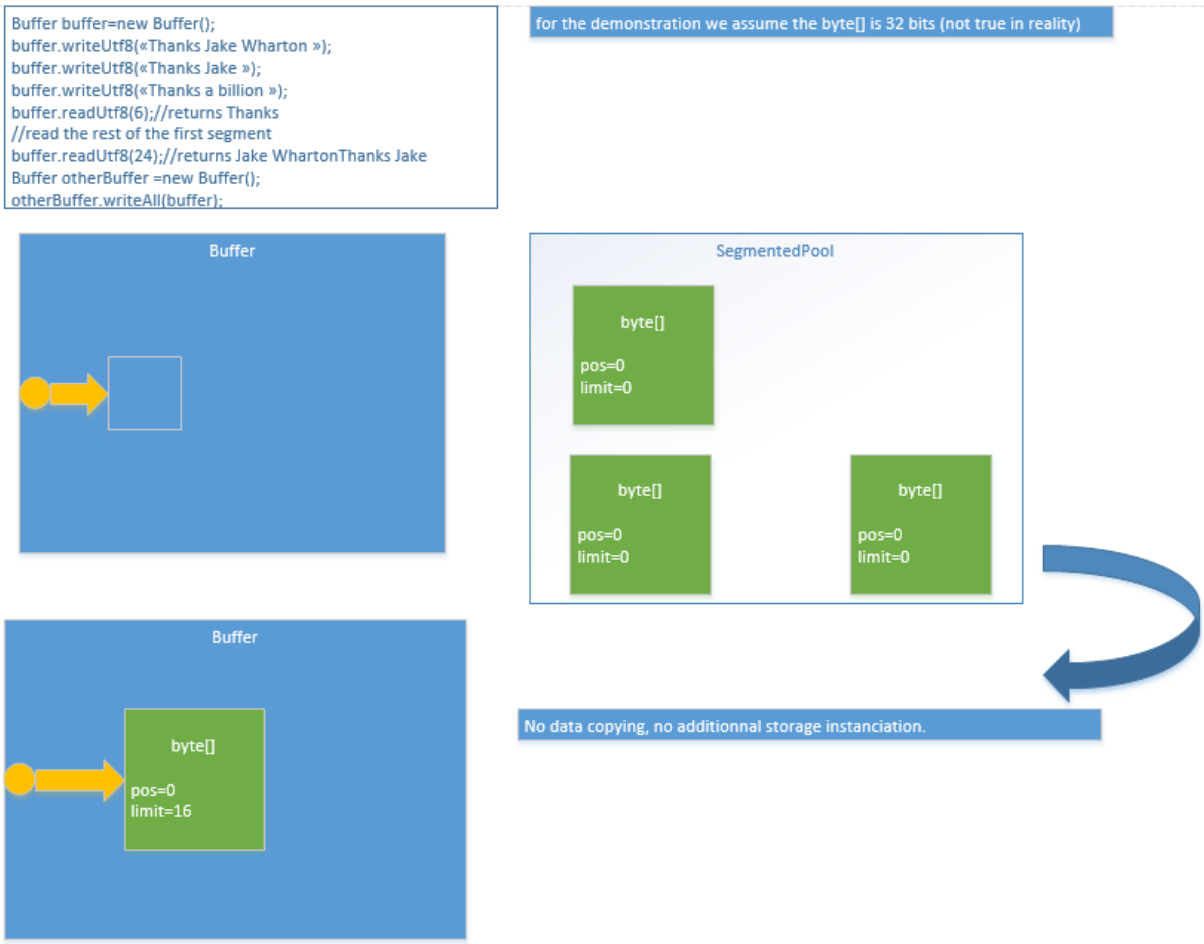
for the demonstration we assume the byte[] is 32 bits (not true in reality)



Comme nous l'avons vu, instancier un Buffer ne fait qu'instancier un pointeur.

Maintenant écrivons le flux du premier Buffer dans le second.

```
Buffer buffer = new Buffer();
buffer.writeUtf8("Thanks Jake Wharton");
buffer.writeUtf8("Thanks Jake");
buffer.writeUtf8("Thanks a billion");
buffer.readUtf8(6);//returns Thanks
//read the rest of the first segment
buffer.readUtf8(24);//returns Jake WhartonThanks Jake
//then create a new buffer
Buffer otherBuffer = new Buffer();
otherBuffer.writeAll(buffer);
```



Le second Buffer pointe maintenant vers le tableau de bytes et c'est tout. Il n'y a pas eu d'allocation mémoire, pas de copie des données et il continuera s'il effectue d'autres opérations à travailler avec le même SegmentedPool. N'est-ce pas magique ?

1.6.2 Exemple concret d'écriture et de lecture dans un fichier

Oui, mais dans la vraie vie, je fais comment pour lire et écrire avec cette Api ?

Et bien c'est juste magiquement simple :

1.6.2.1 Pour l'écriture d'un fichier

C'est très facile et similaire à ce que vous faisiez avant, vous trouvez votre fichier, le créez au besoin et écrivez dedans.

La classe Okio possède un constructeur pour les buffers, les sources et le sinks, il suffit de l'utiliser.

```
/** * Write a file */
private void writeCachFile(String str) {
   //open
   File myFile = new File(getCacheDir(), "myFile");
   try {
      //then write
      if (!myFile.exists()) {
         myFile.createNewFile();
      }
```

```
        BufferedSink okioBufferSink = Okio.buffer(Okio.sink(myFile));
        okioBufferSink.writeUtf8(str);
        //don't forget to close, else nothing appears
        okioBufferSink.close();
    }
```

Il faut bien sûr faire attention à toujours fermer le Buffer.

1.6.2.2 Pour la lecture d'un fichier

Pour la lecture, c'est identique. On trouve le fichier et on le lit.

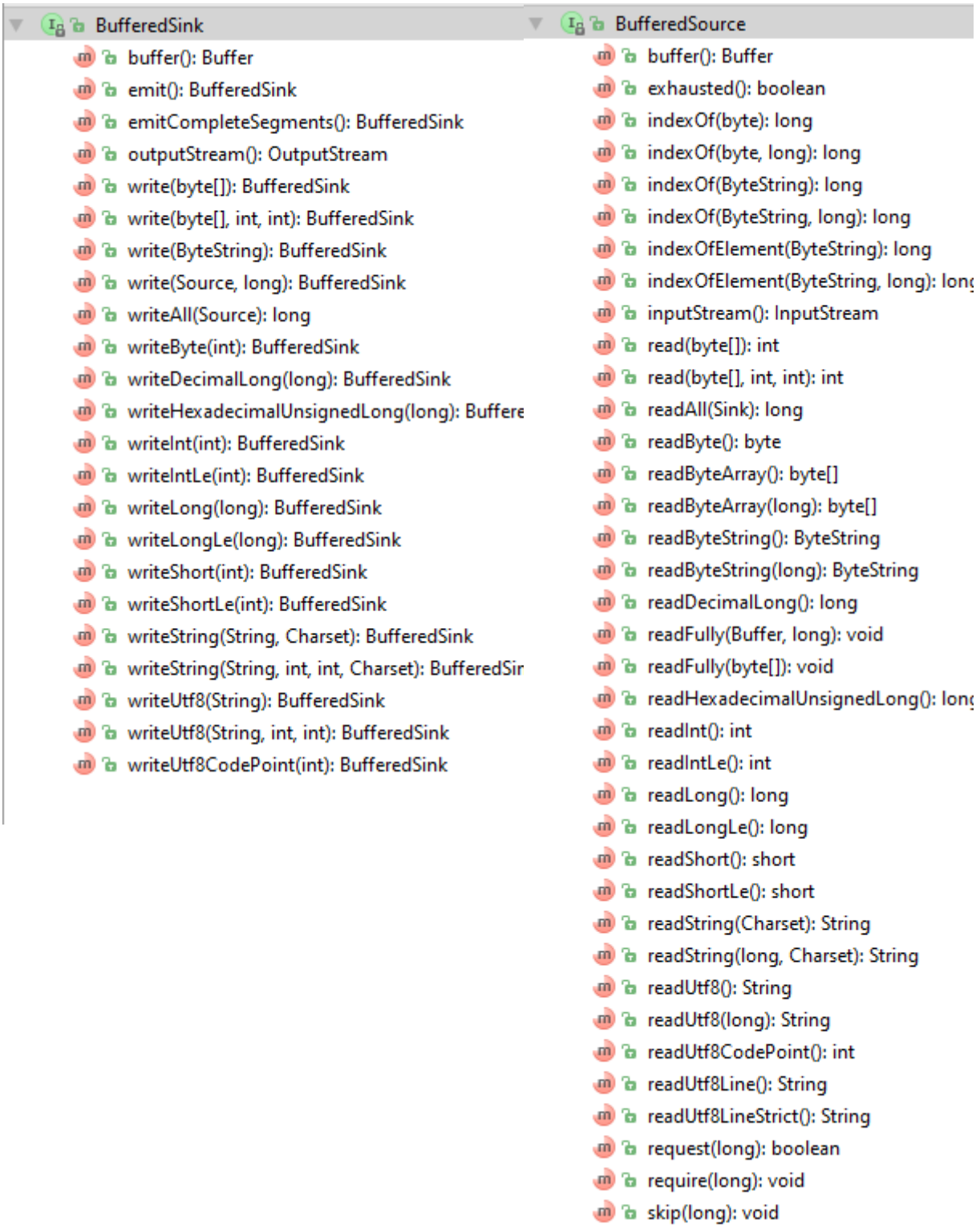
```
/**
 * Read the file you just create
 */
private void readCachFile() throws IOException {
    //open
    File myFile = new File(getCacheDir(), "myFile");
    if(myFile.exists()) {
        try {
            BufferedSource okioBufferSrce = Okio.buffer(Okio.source(myFile));
            str = okioBufferSrce.readUtf8();
            Log.e("MainActivity", "readCachFile returns" + str);
            okioBufferSrce.close();
        } catch (IOException e) {
            Log.e("MainActivity", "Fuck FileNotFoundException occurs", e);
            str = "Fuck occurs, read the logs";
        } finally {
            txvCach.setText(str);
        }
    }
}
```

1.6.2.3 La cerise sur le gâteau

Le bonus qui est bien normal mais qui fait plaisir est que Okio.source() et Okio.sink() acceptent en paramètre les types suivants :

- File
- Path
- InputStream pour source() /OutPutStream pour sink()
- Socket

Vous avez à votre disposition toutes les méthodes qui vont bien pour la lecture/écriture (readInt, readLong, readUtf8, readUtf8LineStrict, readString … et pareil pour l'écriture).



Alors, elle n'est pas belle la vie ? Et merci qui ? Merci Jake Wharton et Jesse Wilson pour leur travail sur cet API.

Ah, oui et puis si j'en chope un en train de manipuler des InputStream dans leurs applications Android pour effectuer de la lecture/écriture, vous savez quoi, je lui coupe les ongles trop courts :) comme ça à chaque fois qu'il utilisera le clavier, une petite douleur lui rappellera « Utilise Okio pour écrire ou lire » :)=

Nous verrons plus d'exemples par la suite.

1.6.3 Les décorateurs

Dernière fonctionnalité dont je souhaitais vous parler : Les décorateurs.

1.6.3.1 GzipSink, GZipSource

Il y a un décorateur important associé à Okio qui est GZipSink et GZipSource, il permettent de compresser le flux avant de l'écrire et de le décompresser avant de le lire et cela de manière transparente. Pour cela une seule ligne suffit :

```
private void zipWriteReadJakeSample() {
        //open
        File myFile = new File(getCacheDir(), "myJakeFile");
        try {
            //then write
            if (!myFile.exists()) {
                myFile.createNewFile();
            }
            Sink fileSink = Okio.sink(myFile);
            Sink gzipSink = new GzipSink(fileSink);
            BufferedSink okioBufferSink = Okio.buffer(gzipSink);
            okioBufferSink.writeUtf8(str);
            //don't forget to close, else nothing appears
            okioBufferSink.close();
            //then read
            myFile = new File(getCacheDir(), "myJakeFile");
            GzipSource gzipSrc = new GzipSource(Okio.source(myFile));
            BufferedSource okioBufferSrce = Okio.buffer(gzipSrc);
            //if you want to see the zip stream
//            BufferedSource okioBufferSrce=Okio.buffer(Okio.source(myFile));
            str = okioBufferSrce.readUtf8();
        } catch (FileNotFoundException e) {
            Log.e("MainActivity", "Fuck FileNotFoundException occurs", e);
        } catch (IOException e) {
            Log.e("MainActivity", "Fuck IOException occurs", e);
        } finally {
            txvJakeWharton.setText(str);
        }
    }
```

Et voilà, vous avez compressez vos données avant de les écrire et décompressez avant de les lire. Quand on pense que sous Android, le problème de l'espace est un problème crucial, je vous engage à compresser avec cette méthode tous ce que vous écrivez sur le disque.

1.6.3.2 Custom décorateur

Il est facile de créer son propre décorateur, pour cela il vous suffit d'étendre la classe Sink ou Source ou les deux en fonction de votre besoin. Puis d'utiliser un Sink source pour rerouter l'écriture ou un Source source pour rerouter la lecture. Je vous montre un exemple pour l'écriture :

```
public class SinkDecoratorSample implements Sink {
```

```java
/**
 * Sink into which does the real work .
 */
private final BufferedSink sink;

/**
 * The constructor .
 */
 public SinkDecoratorSample(Sink sink) {
  if (sink == null) throw new IllegalArgumentException("sink == null");
  this.sink = Okio.buffer(sink);
}

/**
 * Removes {@code byteCount} bytes from {@code source} and appends them to this.
 *
 * @param source
 * @param byteCount
 */
@Override
public void write(Buffer source, long byteCount) throws IOException {
  try {
    Log.e("SinkDecoratorSample", "write has been called");
    //find the bytearray to write
    ByteString bytes = ((BufferedSource) source).readByteString(byteCount);
    //here there is an instanciation
    String data = bytes.utf8();
    Log.e("SinkDecoratorSample", data.length());
    }
    //do the real job
    sink.write(bytes);
  } catch (Exception e) {
    Log.e("SinkDecoratorSample", "a crash occurs :", e);
  }
}

/**
 * Pushes all buffered bytes to their final destination.
 */
@Override
public void flush() throws IOException {
  sink.flush();
}

/**
 * Returns the timeout for this sink.
 */
@Override
public Timeout timeout() {
  return sink.timeout();
```

```
  }

  /**
   * Pushes all buffered bytes to their final destination and releases the
   * resources held by this sink. It is an error to write a closed sink. It is
   * safe to close a sink more than once.
   */
  @Override
  public void close() throws IOException {
    sink.close();
  }
}
```

Rien de bien méchant.
Le point clef est le BufferedSink sink que vous utilisez pour faire le vrai boulot. Le constructeur n'utilise qu'un Sink lui pour vous permettre de les enchainer lors de la construction.
La méthode write mérite un instant d'attention :

```
  public void write(Buffer source, long byteCount) throws IOException {
    try {
      //find the bytearray to write
      ByteString bytes = ((BufferedSource) source).readByteString(byteCount);
      //here there is an instanciation
      String data = bytes.utf8();
      Log.e("SinkDecoratorSample", data.length());
     }
      //do the real job
      sink.write(bytes);
    } catch (Exception e) {
      Log.e("SinkDecoratorSample", "a crash occurs :", e);
    }
  }
```

En effet, on obtient les données à partir du source initial, mais en le lisant, on le vide, il faut donc récupérer son contenu, faire notre traitement, puis demander au Sink décoré d'effectuer l'écriture en lui repassant les données.

Pour utiliser ce décorateur, il vous suffit de lui fournir votre Sink initial :

```
Sink initialSink=Okio.sink(myFile);
SinkDecoratorSample decoratedSink=new SinkDecoratorSample(initialSink);
BufferedSink okioBufferSink = Okio.buffer(decoratedSink);
okioBufferSink.writeUtf8(str);
//don't forget to close, else nothing appears
okioBufferSink.close();
```

1.6.4 Pour résumer

Sink et Source bouge les données

Buffer et ByteString portent les données.

Okio vous permet de créer les éléments Sink, Source et Buffer dont vous avez besoin.

2 OkHttp 3.0

OkHttp est un client http, basé sur Okio pour la lecture/écriture du flux de données. Son objectif est de prendre en charge la communication avec le serveur.

Vous allez me dire que vous utilisez HttpClient/HttpUrlConnection et je vais vous répondre que c'est une mauvaise pratique, utilisez OkHttp à la place, cela vous évitera quelques bugs. Allez jetter un œil à cet article (https://packetzoom.com/blog/which-android-http-library-to-use.html), il vous explique l'historique des clients http sur Android (qui est une vrai misère). Sachez juste que depuis Android 4.4, OkHttp est l'implémentation de HttpUrlConnection dans Android.

Donc, pour être compatible et propre sur toutes vos versions, en utilisant le même code, OkHttp est le bon choix. Un autre choix est une erreur en fait.

2.1 Build Gradle

Il vous suffit de rajouter la dépendance vers OkHttp dans votre fichier gradle.build :

```
dependencies {
   compile fileTree(dir: 'libs', include: ['*.jar'])
   compile 'com.android.support:appcompat-v7:22.0.0'
   compile 'com.squareup.okhttp3:okhttp:3.0.1'
```

2.2 Principe

Le principe est identique à ce que l'on a toujours fait avec nos communications http. On créé un client http, puis on utilise ce client pour effectuer nos requêtes (POST/GET/PUT/DELETE). Ces requêtes doivent être effectuées de manière asynchrone. Ainsi soit vous êtes déjà dans un Thread différent du Main Thread (le Thread IHM) auquel cas vous pouvez exécuter votre requête directement. Soit vous êtes dans le Main Thread, auquel cas, vous effectuer votre requête en fournissant un CallBack. La requête est alors automatiquement exécutée dans un autre Thread et le résultat vous est renvoyé dans votre CallBack.

Le principe est évident et sa mise en place avec OkHttp l'est aussi, merveilleux monde qui est le nôtre.

2.3 Création du client

La création du client est simple, vous utilisez le Builder fournit par la classe OkHttpClient et vous obtenez un client.

Il y a une bonne pratique à respecter : Il vous faut ajouter un fichier de Cache à votre client. Pour cela, il suffit de créer un fichier dans votre dossier de Cache, de lui donner une taille, d'instancier l'objet Cache associé (il sera automatiquement gérer par le LruCach, trop bien joué) et de la passer à votre clientHttp lors de sa construction.

Ensuite, nous le verrons plus tard, il est possible d'ajouter des Intercepteurs à votre client. Cela permet d'intercepter la requête pour effectuer un traitement, typiquement pour faire du log.

Le code est donc le suivant :

```
OkHttpClient client=null;
private  OkHttpClient getClient(){
   if(client==null) {
      //Assigning a CacheDirectory
      File myCacheDir = new File(getCacheDir(), "OkHttpCache");
      //you should create it...
      int cacheSize = 1024 * 1024;
      Cache cacheDir = new Cache(myCacheDir, cacheSize);
      client = new OkHttpClient.Builder()
            .cache(cacheDir)
            .addInterceptor(getInterceptor())
            .build();
   }
   //now it's using the cache
   return client;
}
```

Maintenant, il ne nous reste plus qu'à l'utiliser.

2.4 Requête de type GET

Il faut deux éléments pour faire une requête GET :

- Un client OkHttpClient, pour la communication,
- Une requête Request, de type get.

```
String urlGet = "http://jsonplaceholder.typicode.com/posts/1";

private String getAStuff() throws IOException {
  Request request = new Request.Builder()
       .url(urlGet)
       .get()
       .build();
  //the synchronous way (Here it's ok we are in an AsyncTask already)
  Response response = getClient().newCall(request).execute();
  int httpStatusCode=response.code();
  String ret = response.body().string();
  //You can also have:
  //Reader reader=response.body().charStream();
  //InputStream stream=response.body().byteStream();
  //byte[] bytes=response.body().bytes();
  //But the best way, now you understand the OkIo
  //because no allocation, no more buffering
  //Source src=response.body().source();
  //you should always close the body to enhance recycling mechanism
  response.body().close();
  return ret;
}
```

Un bonne pratique aussi est de vérifier le statut code de la réponse, vous avez une liste de ces codes ici : https://http.cat/ (enjoy :)

2.5 Requête de type GET

Le principe est le même que pour un get, à la différence qu'il vous faut un corps pour votre requête, ainsi trois paramètres sont nécessaires :

- Un client OkHttpClient, pour la communication,
- Une requête Request, de type post,
- Et un corps RequestBody pour le contenu de votre requête.

```
String urlPost="http://jsonplaceholder.typicode.com/posts";
String json="data: {\n" +"    title: 'foo',\n" +"    body: 'bar',\n" +"    userId: 1\n" +"  }";
MediaType JSON= MediaType.parse("application/json; charset=utf-8");

      private String postAStuff() throws IOException {
//          RequestBody body = RequestBody.create(JSON, file);
//          RequestBody body = RequestBody.create(JSON, byte[]);
          RequestBody body = RequestBody.create(JSON, json);
          Request request = new Request.Builder()
                  .url(urlPost)
                  .post(body)
                  .build();
          Call postCall=getClient().newCall(request);
          Response response = postCall.execute(); //you have your response code
          int httpStatusCode=response.code();
          //your responce body
          String ret=response.body().string();
          //and a lot of others stuff...
          //you should always close the body to enhance recycling mechanism
          response.body().close();
          return ret;
      }
```

Pour la partie Json, ne vous enflammez, nous allons voir Moshi dans quelques paragraphes. Ce n'est pas comme ça qu'il faut générer son contenu Json.

2.6 Requête de type PUT/DELETE

C'est identique à une requête de type POST, il faut juste, dans le Builder de la requête utiliser la méthode put ou delete et non plus post.

2.7 Faire une requête asynchrone

Pour effectuer une requête asynchrone, il suffit d'utiliser la méthode enqueue et non plus execute sur votre objet Call :

```
String urlPost="http://jsonplaceholder.typicode.com/posts";
String json="data: {\n" +"    title: 'foo',\n" +"    body: 'bar',\n" +"    userId: 1\n" +"  }";
MediaType JSON= MediaType.parse("application/json; charset=utf-8");

      private String postAStuff() throws IOException {

          OkHttpClient client = new OkHttpClient();
```

```
        RequestBody body = RequestBody.create(JSON, json);
        Request request = new Request.Builder()
                .url(urlPost)
                .post(body)
                .build();
        Call postCall=getClient().newCall(request);
        postCall.enqueue(new Callback() {
                @Override
                public void onFailure(Request request, IOException e) {
                   //oh, shit, i failed
                }

                @Override
                public void onResponse(Response response) throws IOException {
                  //yes, I succeed
                }
            }

);
```

Dans la méthode onFailure du CallBack vous gérez l'exception (et oui, la requête a raté) et dans la méthode onResponse vous effectuez le travail que vous souhaitiez faire.

Notez qu'il vous faut vérifier le statut code de votre réponse dans la méthode onResponse, un 404 n'est pas une failure, ni une exception.

2.8 Télécharger une image

Je vous montre un exemple assez courant pour télécharger une image.

```
String urlGetPicture = "http://jsonplaceholder.typicode.com/photos/1";

public Bitmap urlGetPicture() throws IOException {
  Request request = new Request.Builder()
       .url(urlGetPicture)
       .get()
       .build();
  Call postCall = getClient().newCall(request);
  Response response = postCall.execute();
  if (response.code() == 200) {
    ResponseBody in = response.body();
    InputStream is = in.byteStream();
    Bitmap bitmap = BitmapFactory.decodeStream(is);
    is.close();
    response.body().close();
    //now do a stuff, for exemple store it
    return bitmap;
  }
  return null;
}
```

C'est tout simple comme code.

Juste pour information, si vous souhaitiez l'enregistrer (pour faire du cache et ne pas avoir à la re-télécharger à chaque fois par exemple) :

```
/**
 * How to save a Bitmap on the disk
 * @param fileName
 * @param bitmap
 * @param ctx
 * @throws IOException
 */
private void savePicture(String fileName,Bitmap bitmap, Context ctx) throws IOException {
  //Second save the picture
  //-------------------------
  //Find the external storage directory
  File filesDir = ctx.getCacheDir();
  //Retrieve the name of the subfolder where your store your picture
  //(You have set it in your string ressources)
  String pictureFolderName = "Pictures";
  //then create the subfolder
  File pictureDir = new File(filesDir, pictureFolderName);
  //Check if this subfolder exists
  if (!pictureDir.exists()) {
    //if it doesn't create it
    pictureDir.mkdirs();
  }
  //Define the file to store your picture in
  File filePicture = new File(pictureDir, fileName);
  //Open an OutputStream on that file
  FileOutputStream fos = new FileOutputStream(filePicture);
  //Write in that stream your bitmap in png with the max quality (100 is max, 0 is min quality)
  bitmap.compress(Bitmap.CompressFormat.PNG, 100, fos);
  //The close properly your stream
  fos.flush();
  fos.close();

}
```

2.9 Ajouter un intercepteur

Les intercepteurs sont très utiles pour effectuer un traitement systématique sur chaque requête envoyée. Un bon exemple est de systématiquement zipper son flux sortant (de même côté serveur) pour économiser de la bande passante et le dézipper quand on reçoit la réponse.

Dans l'exemple qui suit, nous mettons en place un intercepteur de type logger.

Pour ajouter un intercepteur, il suffit de la rajouter lors de la construction de votre client OkHttpClient :

```
OkHttpClient client = null;

private OkHttpClient getClient() {
  if (client == null) {
```

```
    //Assigning a CacheDirectory
    File myCacheDir = new File(getCacheDir(), "OkHttpCache");
    //you should create it...
    int cacheSize = 1024 * 1024;
    Cache cacheDir = new Cache(myCacheDir, cacheSize);
    client = new OkHttpClient.Builder()
        .cache(cacheDir)
        .addInterceptor(getInterceptor())
        .build();
  }
  //now it's using the cach
  return client;
}
```

Il ne vous reste plus qu'à implémenter votre intercepteur :

```
public Interceptor getInterceptor() {
  return new LoggingInterceptor();
}

class LoggingInterceptor implements Interceptor {
  //Code pasted from okHttp webSite itself
  @Override
  public Response intercept(Chain chain) throws IOException {
    Request request = chain.request();
    long t1 = System.nanoTime();
    Log.e("Interceptor Sample", String.format("Sending request %s on %s%n%s",
        request.url(), chain.connection(), request.headers()));

    Response response = chain.proceed(request);

    long t2 = System.nanoTime();
    Log.e("Interceptor Sample", String.format("Received response for %s in
        .1fms%n%s",
        response.request().url(), (t2 - t1) / 1e6d, response.headers()));

    return response;
  }
}
```

Pour cela, rien de plus simple, vous créez une classe qui implémente Interceptor et vous surchargez sa méthode intercept.

Dans cette méthode, pour obtenir la requête originale, il vous suffit de la demander au paramètre chain. Pour l'exécuter, il vous suffit d'appeler la méthode proceed du paramètre chain en lui repassant la requête. Et autour de cet appel à vous d'effectuer le travail que vous souhaitez faire.

Dans l'exemple, nous faisons du log.

2.10 Ajouter un intercepteur pour zipper automatiquement ses requêtes

Vraiment une bonne pratique est de zipper vos flux entrant et sortant pour économiser la bande passante de votre utilisateur. Cela se fait en quelques lignes de code.

Tout d'abord il faut ajouter cet intercepteur à votre client OkHttpClient.

```
OkHttpClient client = null;

private OkHttpClient getClient() {
  if (client == null) {
    //Assigning a CacheDirectory
    File myCacheDir = new File(getCacheDir(), "OkHttpCache");
    //you should create it...
    int cacheSize = 1024 * 1024;
    Cache cacheDir = new Cache(myCacheDir, cacheSize);
    client = new OkHttpClient.Builder()
        .cache(cacheDir)
        .addInterceptor(getInterceptor())
        .addInterceptor(new GzipRequestInterceptor())
        .build();
  }
  //now it's using the cach
  return client;
}
```

Ensuite il vous suffit de le définir :

```
/**
 * This interceptor compresses the HTTP request body. Many webservers can't handle this!
 */
final class GzipRequestInterceptor implements Interceptor {
  @Override
  public Response intercept(Chain chain) throws IOException {
    Request originalRequest = chain.request();
    if (originalRequest.body() == null
        || originalRequest.header("Content-Encoding") != null) {
      return chain.proceed(originalRequest);
    }

    Request compressedRequest = originalRequest.newBuilder()
        .header("Content-Encoding", "gzip")
        .method(originalRequest.method(), gzip(originalRequest.body()))
        .build();
    return chain.proceed(compressedRequest);
  }

  private RequestBody gzip(final RequestBody body) {
    return new RequestBody() {
      @Override
```

```
        public MediaType contentType() {
          return body.contentType();
        }

        @Override
        public long contentLength() {
          return -1; // We don't know the compressed length in advance!
        }

        @Override
        public void writeTo(BufferedSink sink) throws IOException {
          BufferedSink gzipSink = Okio.buffer(new GzipSink(sink));
          body.writeTo(gzipSink);
          gzipSink.close();
        }
      };
    }
}
```

Pour que cela marche, il suffit de redéfinir un objet Request, avec comme MimeType le type gzip et comme contenu, le corps (RequestBody) initial zippé. Pour zipper le RequestBody, rien de plus simple, vous utilisez Okio et son décorateur GZipSink.

Je vous laisse l'écriture du dézippage du flux de manière automatique en exercice :) Il vous suffit d'appliquer le GZipSource au body de la réponse renvoyée par chain.proceed, comme avec le loggingInterceptor.

Ces deux exemples d'intercepteurs sont sortis de la librairie, je n'ai pas inventé le code :)

3 Moshi 1.1

Moshi est un parseur Json-Object qui est bâti sur Okio, il n'y a donc pas d'allocation mémoire lors du parsing, pas de recopie de données. L'encodage et le décodage UTF8 est optimisé. Et son utilisation est triviale.

Nous verrons tout d'abord comment lire et écrire avec Moshi « à la main », ce que vous ne ferez que rarement. Cela explique le lien entre Okio et Moshi et vous montre aussi que c'est simple. Puis nous verrons que Moshi possède des Adapters qui vont effectuer ce travail de parsing en une ligne de code, ce qui est l'utilisation nominale de Moshi.

3.1 Build Gradle

Il vous suffit de rajouter la dépendance vers Okio dans votre fichier gradle.build :

```
dependencies {
   compile fileTree(dir: 'libs', include: ['*.jar'])
   compile 'com.android.support:appcompat-v7:22.0.0'
   compile 'com.squareup.moshi:moshi:1.1.0'
```

3.2 Ecrire avec Moshi

Pour écrire avec Moshi, il vous suffit d'obtenir un Okio.Sink. Cela tombe bien, nous savons en créer facilement à partir d'un fichier, nous en récupérons un aussi lors de l'utilisation de OkHttp (nous

avons aussi vu que nous pouvons ajouter des Decorators à ce Sink et là tout devient possible :)
Ensuite il suffit d'écrire dedans son objet, comme on fait avec Json depuis que Json existe.

Dans l'exemple ci-dessous, on écrit dans un fichier.

```
public void writeJson(Context ctx) {
   //open
   File myFile = new File(ctx.getCacheDir(), "myJsonFile");
   try {
      //then write
      if (!myFile.exists()) {
         myFile.createNewFile();
      }
      BufferedSink okioBufferSink = Okio.buffer(Okio.sink(myFile));
      //do the Moshi stuff:
      JsonWriter jsonW = new JsonWriter(okioBufferSink);
      writeJson(jsonW);
      //you have to close the JsonWrtiter too (esle nothing will happen)
      jsonW.close();
      //don't forget to close, else nothing appears
      okioBufferSink.close();
   } catch (FileNotFoundException e) {
      Log.e("MainActivity", "Fuck FileNotFoundException occurs", e);
   } catch (IOException e) {
      Log.e("MainActivity", "Fuck IOException occurs", e);
   }
}
```

Ainsi, on récupère un fichier, puis un Sink sur ce fichier en utilisant Okio et on instancie un JsonWriter à partir de ce Sink. Le processus d'écriture est détaillé dans la méthode writeJson que nous allons voir.
Il faut bien faire attention à clore le JsonWriter et le Sink. Si vous oubliez de les clore, rien ne se passera (enfin si, une fuite mémoire).

La méthode writeJson, ci-dessous, montre comment écrire un fichier JSon avec Moshi :

```
private void writeJson(JsonWriter jsonW) {
   try {
      jsonW.beginObject();
      jsonW.name("Attribute1").value("Has a Value");
      jsonW.name("Attribute2").value(12);
      jsonW.name("Attribute3").value(true);
      jsonW.name("AttributeObject").value("AnotherObject")
            .beginObject()
            .name("Attribute1").value("Has a Value")
            .name("Attribute2").value(12)
            .name("Attribute3").value(true)
            .endObject();
      jsonW.name("Array")
            .beginArray()
            .value("item1")
            .value("item2")
```

```
            .value("item3")
            .endArray();
        jsonW.endObject();

    } catch (IOException e) {       e.printStackTrace();    }
}
```

Comme vous le voyez toutes les méthodes sont là. On commence par ouvrir un objet, on lui ajoute ses attributs, on peut lui ajouter un tableau de primitif, un tableau d'objet, un objet… Bref, c'est trivial.

3.3 Lire avec Moshi

Pour lire avec Moshi, il vous suffit d'obtenir un Okio.Source. Cela tombe bien, nous savons en créer facilement à partir d'un fichier, nous en récupérons un aussi lors de l'utilisation de OkHttp. Ensuite il suffit de lire et reconstruire son objet.

Dans l'exemple ci-dessous, on lit le fichier que l'on vient de créer précédemment.

Tout d'abord, il nous faut récupérer le Okio.Source à lire :

```
public String readJson(Context ctx) {
    Log.e("MoshiSample", "readJson called");
    //open
    File myFile = new File(ctx.getCacheDir(), "myJsonFile");
    JsonReader reader=null;
    StringBuilder strBuild=new StringBuilder();
    String eol= System.getProperty("line.separator");
    try {
        //then write
        if (!myFile.exists()) {
            Log.e("MoshiSample", "readJson: file doesn't exist ");
            myFile.createNewFile();
        }else{
            Log.e("MoshiSample", "readJson: File exists ");
        }
        //check the file by reading it using OkIo
        BufferedSource okioBufferSrce = Okio.buffer(Okio.source(myFile));
        strBuild.append("File content :" + okioBufferSrce.readUtf8()).append(eol);
        strBuild.append("file read... now trying to parse JSon\r\n").append(eol);
        okioBufferSrce.close();

        //Build the source :
        BufferedSource source = Okio.buffer(Okio.source(myFile));
```

Maintenant que nous avons l'objet Source, il ne reste plus qu'à la parser :

```
        //Then read th JSon File
        reader = JsonReader.of(source);
        reader.beginObject();
        while (reader.hasNext()) {
            strBuild.append("peek :" + reader.peek()).append(eol);
            switch (reader.nextName()) {
```

```
            case "Attribute1":
               strBuild.append("Attribute1 :" + reader.nextString()).append(eol);
               break;
            case "Attribute2":
               strBuild.append("Attribute2 :" + reader.nextInt()).append(eol);
               break;
            case "Attribute3":
               strBuild.append("Attribute3 :" + reader.nextBoolean()).append(eol);
               break;
            case "AttributeObject":
               //Parse an object (same as here)
               reader.beginObject();
               strBuild.append("subobject " + reader.nextName() + " :"
                  + reader.nextString()).append(eol);
               strBuild.append("subobject " + reader.nextName() + " :" +
                  + reader.nextString()).append(eol);
               strBuild.append("subobject " + reader.nextName() + " :" +
                  + reader.nextString()).append(eol);
               reader.endObject();
               break;
            case "Array":
               strBuild.append("Array").append(eol);
               reader.beginArray();
               while (reader.hasNext()) {
                  strBuild.append("new item:" + reader.nextString()).append(eol);
               }
               reader.endArray();
               break;
            case "ArrayWithName":
               strBuild.append("array with only name/values pairs").append(eol);
               reader.beginArray();
               while (reader.hasNext()) {
                  reader.beginObject();
                  strBuild.append("item : " + reader.nextName() + " :"
                     + reader.nextString()).append(eol);
                  reader.endObject();
               }
               reader.endArray();
               break;
            //others cases
            default:
               break;
         }
      }
      reader.endObject();
      reader.close();
      okioBufferSrce.close();
   } catch (FileNotFoundException e) {
      Log.e("MainActivity", "Fuck FileNotFoundException occurs", e);
   } catch (IOException e) {
```

```
            Log.e("MainActivity", "Fuck IOException occurs", e);
        } catch (Exception e) {
            Log.e("MainActivity", "Fuck Exception occurs", e);
        } finally {
            Log.e("MoshiSample", "readJson over ehoeho !!!" + strBuild.toString());
            return strBuild.toString();
        }
    }
```

Rien de bien compliqué, on parcourt notre structure en appelant la méthode hasNext. Puis on demande le nom (nextName) et on récupère la valeur associée à ce nom (getString, getInt...). Pour les tableaux, les sous-objets, c'est le même principe.

Mais franchement, qui parse encore ses fichiers Json à la main ?

3.4 Utilisation d'un adapter pour la parsing automatique

Les Adapters Moshi vous permettent de faire la conversion automatique de vos Objets vers leur représentation Json (et inversement).

Pour cela, rien de plus simple, il suffit de créer un Objet Moshi et de l'utiliser pour créer les Adapters dont on a besoin. Pour chaque Objet à convertir, il vous faut un Adapter :

```
Moshi moshi = new Moshi.Builder().build();

JsonAdapter<MyJsonObject> adapter = moshi.adapter(MyJsonObject.class);
```

Et voilà, c'est fini, vous avez créé l'Adapter associé à l'objet MyJsonObject. Il ne reste plus qu'à l'utiliser.

Pour l'écriture, il suffit de demander à l'Adapter d'effectuer le boulot en lui passant l'objet à convertir et le Sink dans lequel écrire :

```
public String usingAdapter(Context ctx) {
  //open
   File myFile = new File(ctx.getCacheDir(), "myJsonObjectFile");
   try {
      //then write
      if (!myFile.exists()) {myFile.createNewFile();}
      BufferedSink okioBufferSink = Okio.buffer(Okio.sink(myFile));
     adapter.toJson(okioBufferSink, new MyJsonObject());
      //don't forget to close, else nothing appears
      okioBufferSink.close();
```

Pour l'écriture (nous sommes toujours dans la même méthode), c'est pareil, l'Adapter fait tout le boulot pour vous :

```
      //then read :
```

```
        BufferedSource okioBufferSource = Okio.buffer(Okio.source(myFile));
        myObj = adapter.fromJson(okioBufferSource);
    } catch (FileNotFoundException e) {
        Log.e("MainActivity", "Fuck FileNotFoundException occurs", e);
    } catch (IOException e) {
        Log.e("MainActivity", "Fuck IOException occurs", e);
    }finally {
        return myObj==null?"null":myObj.toString();
    }
}
```

Comment dire? Plus simple, tu meurs.

4 Retrofit 2.0

L'objectif de Retrofit est de fournir une framework puissant qui vous permet de mettre en place une abstraction simple et cohérente pour vos appels réseaux, changeant votre Api http en interface Java.

Ainsi Retrofit :

- vous permet de déclarer votre couche réseau sous forme d'une interface,
- vous fournit un objet Call qui encapsule l'interaction avec une requête et sa réponse,
- vous permet de paramétriser l'objet Response,
- offre de multiples et efficaces convertisseurs (Xml, Json),
- offre de multiples mécanismes pluggables d'exécution.

Bref, elle vous simplifie la vie au niveau de la couche réseau de votre application, tout en implémentant pour vous les bonnes pratiques d'abstraction et en vous permettant de customiser son comportement.

De plus, elle est bâtie sur Okio, OkHttp et peut utiliser Moshi pour convertir vos objets Json.

4.1 Build Gradle

Il vous suffit de rajouter la dépendance vers Retrofit dans votre fichier gradle.build :

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.2.0'
    //compile 'com.squareup.okhttp3:okhttp:3.0.1'<-a bug here
    compile 'com.squareup.retrofit2:retrofit:2.0.0-beta3'
    compile 'com.squareup.okhttp3:okhttp:3.0.0-RC1'
    compile 'com.squareup.retrofit2:converter-moshi:2.0.0-beta3'
    compile 'com.squareup.okhttp3:logging-interceptor:3.0.0-RC1'
}
```

Par contre, comme vous le voyez, Retrofit a besoin d'OkHttp pour fonctionner.

Nous utiliserons aussi le Logging-interceptor et moshi dans la suite de ce chapitre, d'où leur présence dans ce fichier de Build.

4.2 Principe

Retrofit a pour objectif de vous simplifier la gestion de vos appels réseaux avec une interface, une instance de cette interface, appelée Service (mais ça n'a rien à voir avec les service Android) et un appel de type Call. Pour des raisons de clarté nous appellerons les instances de ces interfaces des *Services Retrofit* :

L'interface vous permet de définir les requêtes :

```
public interface WebServerIntf {
/** Get Method
(this is the way to declare them using Retrofit 2.0)
   @GET("posts/1")
  Call<Post> getPostOne();
   @GET("posts/{id}")
   Call<Post> getPostById(@Path("id") int id);
   @GET("posts")
   Call<List<Post>> getPostsList();
}
```

L'instanciation de cette interface se fait via un objet Retrofit et permet de construire un Service Retrofit qui est utilisé pour faire les appels.
J'ai regroupé, dans ces exemples, toutes les constructions de l'objet Retrofit, du Service Retrofit et du client OkHttpClient dans une même classe nommée RetrofitBuilder :

```
public class RetrofitBuilder {
public static final String BASE_URL = "http://jsonplaceholder.typicode.com/";

public static WebServerIntf getSimpleClient(){
   //Using Default HttpClient
   Retrofit retrofit =new Retrofit.Builder()
     //you need to add your root url
     .baseUrl(BASE_URL)
     .build();
   WebServerIntf webServer=retrofit.create(WebServerIntf.class);
   return webServer;
}
```

Les appels se font alors simplement :

```
public class BusinessService {
//Make the Rest Call
//Okio, OkHttp and Moshi are behind
getPostOneCall = webService.getPostOne();
//so you need to make an async call
getPostOneCall.enqueue(new Callback<Post>() {
   @Override
   public void onResponse(Response<Post> response) {
   }
   @Override
   public void onFailure(Throwable t) {
   }
});
```

Il faut un nouvel objet Call pour chaque appel. Une fois que l'objet call a effectué l'appel, il n'est plus utilisable. Vous pouvez cloner vos objets Call pour pouvoir « les utiliser plusieurs fois » avant cet appel.
Dans la suite de cet article, j'ai regroupé tous les appels au sein d'une classe qui se nomme BuisnessService. En effet, c'est une bonne pratique de ne pas effectuer ces appels directement dans vos activités ou vos fragments mais dans une classe à part. Si de plus, cette classe suivait le cycle de vie de l'Application et non pas de vos activités avec un système de caching, ce serait une bonne architecture.

4.2.1 Instanciation d'un Service Retrofit

De manière assez naturelle, vous mettrez en place de Service Retrofit du type suivant :

```
public class RetrofitBuilder {
public static WebServerIntf getComplexClient(Context ctx) {
   //get the OkHttp client
   OkHttpClient client = getOkHttpClient(ctx);

   //now it's using the cach
   //Using my HttpClient
   Retrofit raCustom = new Retrofit.Builder()
         .client(client)
         .baseUrl(BASE_URL)
               //add your own converter first (declaration order matters)
               //the responsability chain design pattern is behind
         .addConverterFactory(new MyPhotoConverterFactory())
               //You need to add a converter if you want your Json to be automagicly convert
              //into the object
         .addConverterFactory(MoshiConverterFactory.create())
               //then add your own CallAdapter
         .addCallAdapterFactory(new ErrorHandlingCallAdapterFactory())
         .build();
   WebServerIntf webServer = raCustom.create(WebServerIntf.class);
   return webServer;
}
```

Nous avons construit pour instancier l'interface un objet Retrofit qui possède les caractéristiques suivantes :

- nous fournissons le client OkHttpClient qui est utilisé pour la communication http (nous y reviendrons),
- nous définissons la BASE_URL, l'Url racine pour tous nos appels,
- nous ajoutons un ConverterFactory de type MyPhotoConverterFactory, un objet qui convertit spécifiquement les objets de type Photo,
- nous ajoutons un ConverterFactory de type MoshiConverterFactory qui servira à convertir tous les objets (non Photo) automatiquement en utilisant Moshi,
- nous ajoutons un CallAdapterFactory qui nous permet une gestion plus fine des erreurs rencontrées.

Nous verrons tous ces éléments un à un, mais commençons par le client OkHttpClient et regardons comment le définir :

```
public class RetrofitBuilder {
@NonNull
public static OkHttpClient getOkHttpClient(Context ctx) {
   //define the OkHttp Client with its cache!
   //Assigning a CacheDirectory
   File myCacheDir=new File(ctx.getCacheDir(),"OkHttpCache");
   //you should create it...
   int cacheSize=1024*1024;
   Cache cacheDir=new Cache(myCacheDir,cacheSize);
   Interceptor customLoggingInterceptor=new CustomLoggingInterceptor();
   HttpLoggingInterceptor httpLogInterceptor=new HttpLoggingInterceptor();
   httpLogInterceptor.setLevel(HttpLoggingInterceptor.Level.BASIC);
   return new OkHttpClient.Builder()
         //add a cache
         .cache(cacheDir)
         //add interceptor (here to log the request)
         .addInterceptor(customLoggingInterceptor)
         .addInterceptor(httpLogInterceptor)
         .build();
}
```

C'est un client OkHttpClient typique, il possède un Cache et nous lui avons ajouté deux Interceptors pour faire du logging. Un seul aurait suffi, mais pour l'article, je rajoute le natif (HttpLoggingInterceptor) et un custom (CustomLoggingInterceptor).

Je pense que vous n'avez pas réalisé mais en quelques lignes de code nous avons fait un truc de dingues ; toutes nos requêtes sont loggées, toutes nos erreurs sont traitées de façon centralisée, tous nos objets sont automatiquement convertis au format Json (et vis-versa) et nous avons une classe concrète pour effectuer les appels réseaux que nous avons défini dans notre interface.

4.2.2 Appels synchrones ou asynchrones ?

Il n'y a rien de plus facile avec Retrofit que de faire un appel synchrone ou asynchrone. Vous n'avez plus à définir dans votre interface d'appels si vous souhaitez effectuer le traitement de manière synchrone ou pas. C'était Retrofit 1.* et c'est une grande amélioration de Retrofit 2.

Imaginez que nous avons défini notre interface d'appels ainsi :

```
public interface WebServerIntf {
/** Get Method
(this is the way to declare them using Retrofit 2.0)
   @GET("posts/1")
   Call<Post> getPostOne();
   @GET("posts/{id}")
   Call<Post> getPostById(@Path("id") int id);
   @GET("posts")
   Call<List<Post>> getPostsList();
}
```

Nous reviendrons sur la déclaration de cette interface quand nous aborderons le chapitre Annotations.

Pour faire un appel, il vous faut un objet Call qui s'obtient simplement en le demandant à votre Service Retrofit :

```
Call<Post> getPostOneCall getPostByIdCall = webServiceComplex.getPostOne();
```

Ainsi pour faire un appel synchrone (vous avez défini votre interface d'appels et votre Service Retrofit, bien entendu), il suffit d'appeler la méthode execute() sur cet objet :

```
Response<Post> response=getPostOneCall.execute();
```

L'objet renvoyé est un objet de type Response avec pour paramètre de généricité le type d'objet encapsulé dans cette réponse.

Pour effectuer le même appel de manière asynchrone :

```
getPostOneCall.enqueue(new Callback<Post>() {
    @Override
    public void onResponse(Response<Post> response) {
        Log.e("BusinessService", "The call is back with success");
    }

    @Override
    public void onFailure(Throwable t) {
        Log.e("BusinessService", "The call failed");
    }
});
```

Nous fournissons un CallBack. Si la réponse nous revient de manière normale nous passons par la méthode onResponse, si une exception est survenue durant le traitement, nous revenons dans la méthode onFailure.
A noter que vous pouvez très bien revenir dans la méthode onResponse avec une réponse de code 404… Il ne faut pas confondre une erreur d'exécution et une erreur dans la réponse. Une erreur dans la réponse assure justement une absence d'erreur d'exécution, vous n'avez pas votre résultat pour autant, on est d'accord.

4.2.3 L'objet Response

Un objet Response possède les attributs body, code, message, isSuccess, headers, errorBody et raw. Si nous loggons ces attributs nous obtenons les valeurs suivantes :

```
//This will log this information
//Analyzing a response Object
//Analyzing a response.code()=200
//Analyzing a response.message()=OK
//Analyzing a response.isSuccess()=true
//Analyzing a response.headers()=Connection: keep-alive
//              Content-Type: application/json; charset=utf-8
//              Server: Cowboy
//              X-Powered-By: Express
//              Vary: Origin
//              Access-Control-Allow-Credentials: true
//              Cache-Control: no-cache
//              Pragma: no-cache
//              Expires: -1
//              X-Content-Type-Options: nosniff
```

```
//            Etag: W/"1fd-jhjIa9s91vj8ZufojwdSzA"
//            Date: Thu, 21 Jan 2016 15:11:30 GMT
//            Via: 1.1 vegur
//            OkHttp-Sent-Millis: 1453389092618
//            OkHttp-Received-Millis: 1453389096044
//Analyzing a response.errorBody()=null
//Analyzing a response.raw()=Response{protocol=http/1.1, code=200, message=OK,
url=http://jsonplaceholder.typicode.com/users/2}
```

Les valeurs changent (on est bien d'accord) en fonction du type d'appel, du serveur appelé, de sa réponse… mais ça vous donne une bonne idée de à quoi correspondent ces champs.

4.2.4 Annuler vos appels quand ils n'ont plus lieu d'être

Une dernière fonctionnalité bien utile de Retrofit est de pouvoir annuler les appels. Ainsi, s'ils sont liés au cycle de vie d'une activité, annulez-les dans la méthode onStop. S'ils sont liés au cycle de vie d'un autre objet, n'oubliez pas de les annuler quand cet objet meurt.

Par exemple, dans le tutorial associé à cette article, ma classe BusinessService qui implémente les appels possède une méthode release qui est appelée quand mon activité passe dans onStop :

```
public void release() {
   activity = null;
   //you have to cancel your calls in OnStop
   getPostOneCall.cancel();
   getPostByIdCall.cancel();
   getPostListCall.cancel();
   getUsersListCall.cancel();
   getUserByIdCall.cancel();
   getPhotoWithQueryCall.cancel();
}
```

4.3 Annotations

Les annotations sont utilisées pour décrire votre interface d'appels, vos méthodes POST/GET/PUT/DELETE.

Je vous fais une petite digression sur ces 4 méthodes qui sont des normes du w3c pour votre culture générale (la mienne étant lacuneuse à ce sujet, je me dis que je ne dois pas être le seul) :

GET: La méthode GET signifie récupérer toutes les informations (sous la forme d'une entité) identifié par l'URI de la requête.

POST: La méthode POST est utilisée pour demander que le serveur d'origine accepte l'entité incluse dans la demande comme un nouveau subordonné de la ressource identifiée par l'URI de la requête.

PUT: La méthode PUT demande que l'entité incluse soit stockée sous l'URI de la requête.

SUPPRIMER: La méthode DELETE demande que le serveur d'origine supprime la ressource identifiée par l'URI de la requête. .

Reprenons, ainsi les annotations sont utilisées pour décrire vos requêtes. Ainsi, le code et les explications de ce chapitre appartiennent à la classe :

```
public interface WebServerIntf {
```

4.3.1 Les annotations @GET @PUT @POST @DELETE @Path @Body

@GET @PUT @POST @DELETE sont les annotations principales qui définissent le type de la requête.

Nous les utilisons ainsi :

```
@GET("posts/1")
Call<Post> getPostOne();
@POST("posts")
Call<Post> addNewPost(@Body Post post);
@PUT("users/1")
Call<User> updateUserOne(@Body User user);
@DELETE("user/{id}")
Call<User> deleteUserById(@Path("id")int id);
```

Les types Post et Put doivent avoir un corps (Body) qui spécifie l'entité à traiter, ainsi elles utilisent la balise @BODY en paramètre pour passer l'objet Post au serveur.

Le balise @BODY permet ainsi de spécifier le corps de la requête. Il n'est pas rare que nous passions un corps à nos requêtes dans une méthode GET.

Vous pouvez passer un paramètre null pour un @Body sans que cela ne gêne, votre requête ne possèdera juste pas de corps.

L'encodage des objets passés en paramètre sera effectué en utilisant le(s) convertisseur(s) que vous avez définis lors de l'instanciation de votre objet Retrofit (celui qui permet d'instancier votre interface d'appels).

La balise @Path permet d'avoir des URL dynamiques qui seront résolues lors de l'exécution. Ainsi :

```
@GET("posts/{id}")
Call<Post> getPostById(@Path("id") int id);
```

Permet de spécifier que le paramètre id passé en paramètre de la méthode getPostById servira à construire l'URL finale.

4.3.2 Les annotations {@Multipart, @Part} et {@FormUrlEncoded, @Field}

Ces annotations servent à définir le type Mime (le MimeType) de votre requête et de passer un ensemble d'éléments à votre serveur avec un formatage cohérent vis-à-vis de ce type MIME.

Ainsi pour définir une requête possédant plusieurs parties :

```
@Multipart
@PUT("photos")
Call<Photo> newPhoto(@Part("photo") Photo photoObject,
                     @Part("content") byte[] picture);
```

Pour définir une requête de type FormUrlEncoded :

```
@FormUrlEncoded
@POST("user/{id}/edit")
Call<User> updateUserWithForm(@Path("id")int id,
```

```
                                    @Field("name")String name,
                                    @Field("points")int point);
```

Pour aller plus loin dans la compréhension de ces types, je vous propose de jetter un coup d'œil à cette question sur stackoverflow : http://stackoverflow.com/questions/4007969/application-x-www-form-urlencoded-or-multipart-form-data

4.3.3 Les annotations @Query, @QueryMap

Ces annotations permettent d'encoder les paramètres en tant que query dans votre Url.

Ainsi si vous définissez votre interface ainsi:

```
@GET("photos/1")
Call<Photo> getPhotoWithQuery(@Query("data") int id,
                              @QueryMap Map<String,String>option);
```

Il vous suffit de l'utiliser ainsi :

```
HashMap<String, String> options = new HashMap<String, String>();
options.put("parameter1", "value1");
options.put("parameter2", "value2");
options.put("parameter3", "value3");
getPhotoWithQueryCall = webServiceComplex.getPhotoWithQuery(3, options);
```

Et le requête résultante ressemblera à cela :

http://jsonplaceholder.typicode.com/photos/1?data=3&parameter2=value2&parameter1=value1&parameter3=value3

Un cas particulier de l'utilisation du Query est lorsque l'on souhaite que les mêmes paramètres soient utilisés plusieurs fois. Il suffit alors juste de mettre une liste :

```
@GET("photos")
Call<Photo> getPhotoWithQuery(@Query("id") List<Integer> id);
```

Vous obtiendrez (en passant une liste à la méthode getPhotoWithQuery) :

http://jsonplaceholder.typicode.com/photos/1?id=3&id=2&id=1&id=4

Vous pouvez passer null à n'importe lequel de ces paramètres (@Query ou @QueryMap), le paramètre sera juste omis de la requête.

4.3.4 L'annotation @Header et plus généralement la balise Header

La balise @Header permet de rajouter des paramètres dans le header de votre requête.

Vous pouvez le spécifier de manière statique :

```
@Headers({
    "Accept: application/vnd.yourapi.v1.full+json",
```

```
    "User-Agent: Your-App-Name"
})
@GET("posts/1")
Call<Post> getPostOne();
```

Vous pouvez le spécifier de manière dynamique :

```
@GET("posts/{id}")
Call<Post> getPostById(@Header("Content-Range")String contentRange,
                       @Path("id") int id);
```

Il suffit ensuite de la passer en paramètre de votre méthode.

Enfin vous pouvez spécifier un header général à toutes vos requêtes, mais pour ça, il faut le rajouter au niveau de votre client OkHttpClient :

```
public class RetrofitBuilder {
OkHttpClient.Builder httpClient = new OkHttpClient.Builder();
httpClient.interceptors().add(new Interceptor() {
  @Override
  public Response intercept(Interceptor.Chain chain) throws IOException {
    Request original = chain.request();

    Request request = original.newBuilder()
        .header("User-Agent", "Your-App-Name")
        .header("Accept", "application/vnd.yourapi.v1.full+json")
        .method(original.method(), original.body())
        .build();

    return chain.proceed(request);
  }
}
```

4.4 Mise en place de l'authentification avec Retrofit

Je n'expliquerai dans cet article que la mise en place d'une autorisation simple, vous fournissant quelques liens si vous souhaitez aller plus loin. L'authentification mériterait un bon gros article à elle seule.

Le principe de l'authentification simple est de rajouter une chaine de caractères dans le header de vos requêtes pour permettre au serveur d'authentifier l'utilisateur. Il faut donc la rajouter au niveau du client http à sa création. Cette création étant appelée lors de la création du service Retrofit, c'est cette méthode qui va calculer cette chaîne de caractères :

```
public class RetrofitBuilder {
/** * Basic credentials send to the server at each request */
private static String basicCredential = null;

/** * Retrieve an authenticated Retrofit webserver */
public static WebServerIntf getAuthenticatedClient(Context ctx, String name, String
assword) {
```

```
    String credentials = name + ":" + password;
    basicCredential =
            "Basic " + Base64.encodeToString(credentials.getBytes(), Base64.NO_WRAP);
    //get the OkHttp client
    OkHttpClient client = getOkAuthenticatedHttpClient(ctx);

    //now it's using the cach
    //Using my HttpClient
    Retrofit raCustom = new Retrofit.Builder()
            .client(client)
            .baseUrl(BASE_URL)
                //You need to add a converter if you want your Json to be automagicly convert
                //into the object
            .addConverterFactory(MoshiConverterFactory.create())
            .build();
    WebServerIntf webServer = raCustom.create(WebServerIntf.class);
    return webServer;
}
```

La méthode getOkAuthenticatedHttpClient se contente de mettre en place le header pour toutes les requêtes.

```
@NonNull
public static OkHttpClient getOkAuthenticatedHttpClient(Context ctx) {
    return new OkHttpClient.Builder()
            .addInterceptor(new Interceptor() {
                @Override
                public Response intercept(Chain chain) throws IOException {
                    Request original = chain.request();

                    Request.Builder requestBuilder = original.newBuilder()
                            .header("Authorization", basicCredential)
                            .header("Accept", "applicaton/json")
                            .method(original.method(), original.body());

                    Request request = requestBuilder.build();
                    return chain.proceed(request);
                }
            })
            .build();
}
```

Ces méthodes, dans mon exemple, appartiennent toutes deux à la classe que j'ai appelée RetrofitBuilder.

Maintenant il n'y a plus qu'à utiliser ce service pour exécuter toutes nos requêtes ayant besoin d'authentification.

La définition et l'appel Http n'ont pas changé, il n'y a pas de paramètre à passer à la méthode login.

Ainsi si je définis l'interface d'appels suivante :

```
public interface WebServerIntf {
   @GET("users/authent")
   Call<User>login();
}
```

Je peux alors simplement effectuer des requêtes authentifiées :

```
public class BusinessService {
/** * The call for authentification */
Call<User>authentificateUserCall;
/** * The authenticated user*/
User currentUser;

/** * Should be called first when using authentificated request */
public void initializeAuthentificatedCommunication(Context ctx,String name, String
   password){
   webServiceAuthenticated=
      RetrofitBuilder.getAuthenticatedClient(ctx,name,password);
}

/** An exemple of authenticated request */
public void login(){
   //You service add credentials in the header, so just call login
   authentificateUserCall=webServiceAuthenticated.login();
   authentificateUserCall.enqueue(new Callback<User>() {
      @Override
      public void onResponse(Response<User> response) {
         if(response.body()!=null){
            //ok your user is authentificated
            currentUser=response.body();
         }
      }
      @Override
      public void onFailure(Throwable t) {        }
   });
}
```

Il vous est possible d'utiliser différents service Retrofit au sein de votre application. L'un fournira des requêtes authentifiées, l'autre non par exemple. Vous pouvez aller plus loin bien entendu. Et cela n'impacte en rien votre application, ni vos appels. En effet, lors de l'appel, vous utilisez n'importe lequel pour instancier votre objet Call en fonction de votre besoin.

C'est assez magique ces services Retrofit je trouve.

Pour aller plus loin concernant l'authentification, vous pouvez commencer par ces articles :

https://futurestud.io/blog/retrofit-token-authentication-on-android

https://futurestud.io/blog/oauth-2-on-android-with-retrofit

https://futurestud.io/blog/retrofit-2-hawk-authentication-on-android

Vous pouvez aussi jetter un oeil à cette librairie dédiée justement à la mise en place d'OAuth avec Retrofit:

https://github.com/AliAbozaid/OAuth2Library

4.5 Mise en place des convertisseurs pour Retrofit

Les convertisseurs servent à convertir vos objets Java dans le format qui va bien pour leur transport (Xml, Buffer, Json) dans un sens (Request) et l'autre (Response).

Vous devez les ajouter à votre build.gradle, ils ne font pas partie de la librairie Retrofit.

4.5.1 Convertisseur natif

Il est préconisé d'utiliser Moshi comme convertisseur par défaut pour le format Json car il utilise de manière native les Sinks et les Sources d'Okio sur lequel se base OkHttp. Il n'y a donc pas de recopie de Buffer, pas d'instanciation d'inputStream ou ce genre de gaspillage mémoire.

Pour dire à votre Service Retrofit d'utiliser un convertisseur, il suffit de le déclarer lors de la construction de l'objet Retrofit :

```
public static WebServerIntf getSimpleClient(){
   //Using Default HttpClient
   Retrofit ra=new Retrofit.Builder()
         //you need to add your root url
         .baseUrl(BASE_URL)
         //You need to add a converter if you want your Json to be automagicly
         //convert into the object
         .addConverterFactory(MoshiConverterFactory.create())
         .build();
   WebServerIntf webServer=ra.create(WebServerIntf.class);
   return webServer;
}
```

Si vous ajouter plusieurs convertisseurs, l'ordre de déclaration est important. Les convertisseurs sont interrogés un à un pour savoir s'ils doivent faire le boulot. Le premier qui dit oui l'effectuera. C'est basé sur le Design Pattern de la chaine de responsabilité.
Vous avez un ensemble de convertisseurs natifs, disponibles et compatibles avec Retrofit :

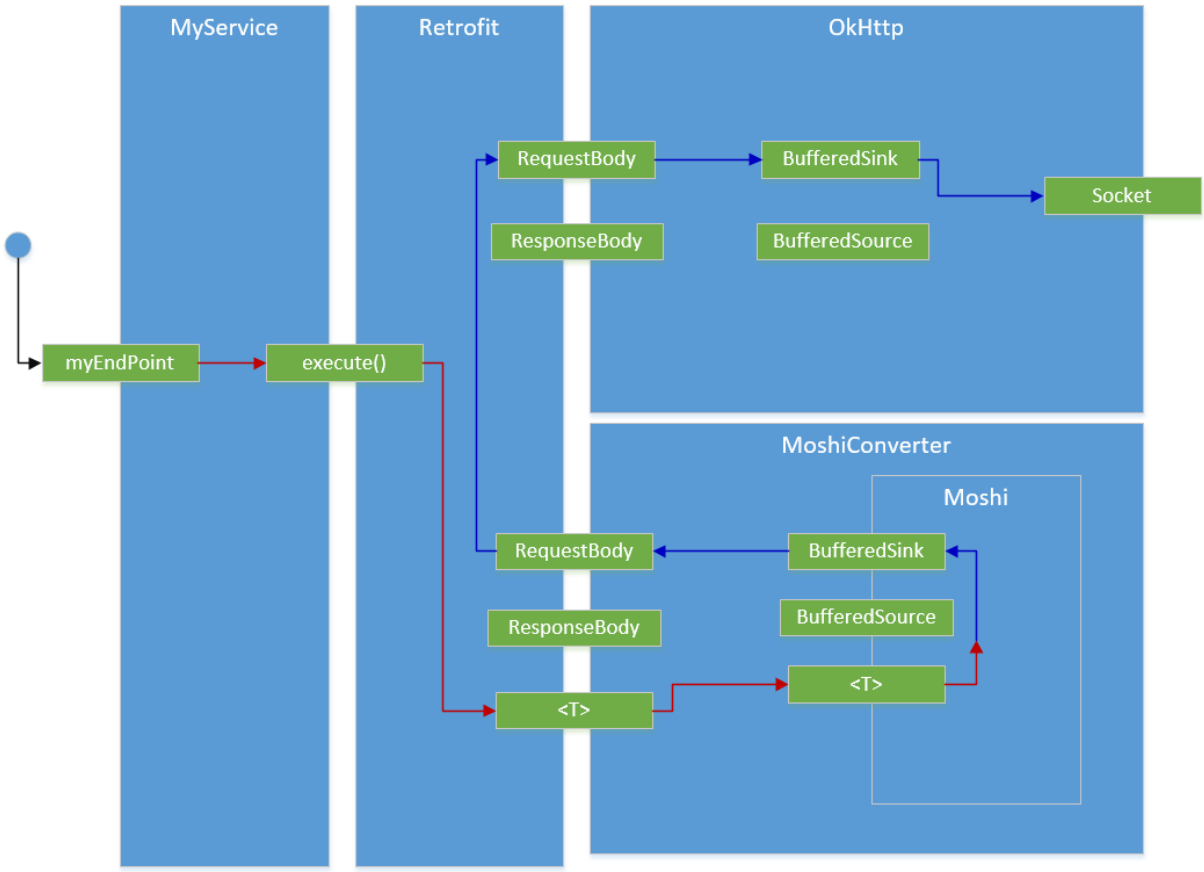
Ci-dessous, vous avez la liste du moteur utilisé et de la dépendance gradle à ajouter à votre fichier build.gradle :

- Gson: com.squareup.retrofit:converter-gson
- Jackson: com.squareup.retrofit:converter-jackson
- Moshi: com.squareup.retrofit:converter-moshi
- Protobuf: com.squareup.retrofit:converter-protobuf
- Wire: com.squareup.retrofit:converter-wire
- Simple XML: com.squareup.retrofit:converter-simplexml

4.5.2 Pourquoi utiliser Moshi comme convertisseur ?

Comme je vous le disais, Moshi est le convertisseur à utiliser par défaut si l'on manipule du Json et la principale raison est l'allocation mémoire. En effet Moshi est le seul qui soit parfaitement compatible avec Okio, ce qui permet une utilisation des Sinks et des Sources natifs et évite une allocation mémore pour les convertir en inputStream et ce à plusieurs niveaux:

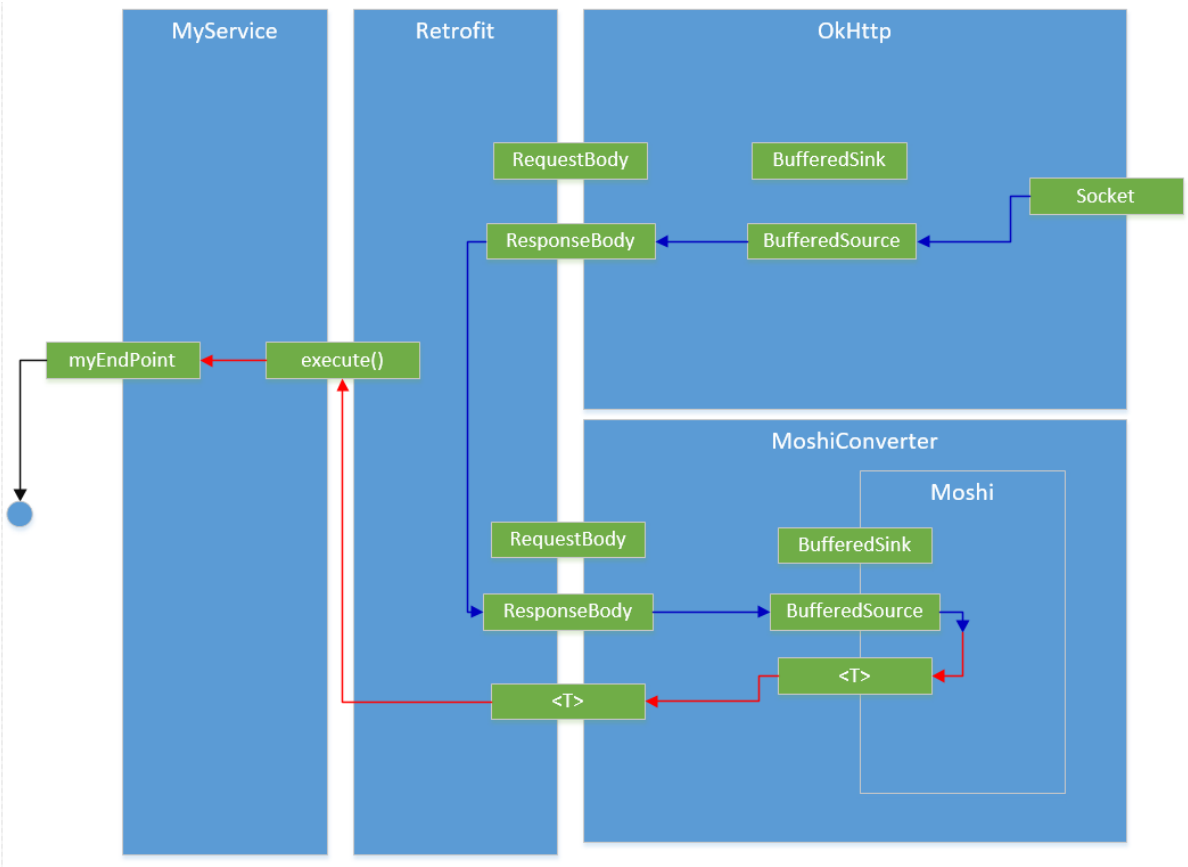
Examinons un Call et commençons par la requête:



Dans ce schéma, la flêche rouge porte l'objet, la bleue porte le BufferSink dans lequel sont écrites les données.

Ce qui important ici de bien comprendre, c'est que les interfaces entre les libraries utilisées par Retrofit (Moshi et OkHttp) sont basées sur Okio. Ainsi les objets ne sont pas recopiés pour les passer entre ces libraries, elles utilisent naturellement le BufferSink créé par Moshi pour convertir cet objet. L'objet RequestBody ne fait qu'encapsuler le BufferSink et le ré-utilise. Il sera "consumer" lors de l'écrire réelle de l'information dans la Socket.

La réponse suit le même paradigme:



Il est important de comprendre la compatibilité profonde de ces librairies et l'économie de recopie des données qu'elle génère.

Ainsi, si vous travaillez avec du Json, il est fortement recommandé d'utiliser Moshi (et aussi de zipper son flux avec un Adapter comme vu précédement).

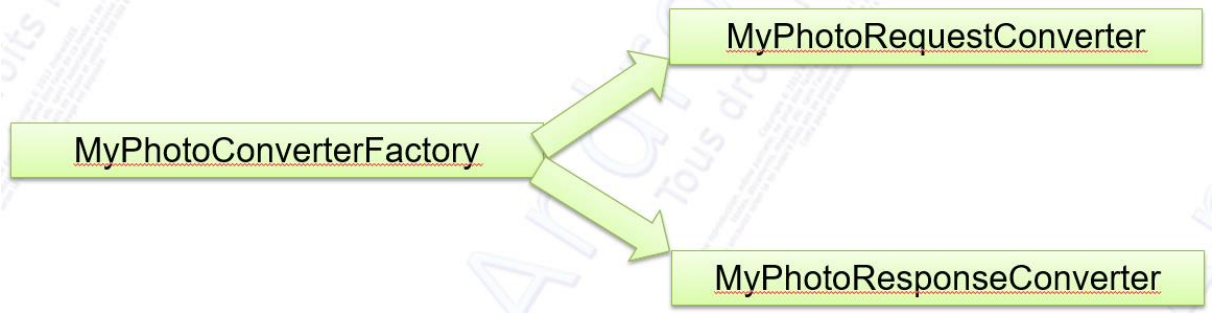
4.5.3 Convertisseur spécifique (custom converter)

Il est possible (et presque facile) d'ajouter son propre convertisseur à Retrofit. Pour cela, il vous faut l'ajouter à l'objet Retrofit lors de sa construction (comme précédemment) et le définir.

Définir un convertisseur Retrofit nécessite trois classes : Une factory, un convertisseur pour la requête et un convertisseur pour la réponse. Vous pouvez en fonction de votre besoin ne mettre que le convertisseur pour la réponse ou la requête.

Nous allons examiner un exemple pour comprendre ce principe basé sur la conversion d'un objet Photo qui possède comme attributs principaux un identifiant, un titre, une url.

Le principe est le suivant :



La Factory reçoit un objet et regarde si elle doit le convertir et quel type de conversion doit être effectué (Réponse->Objet ou Objet->Requête). Si elle n'a pas à convertir l'objet, elle renvoie null, si elle a à la convertir, elle regarde le sens de conversion et renvoie le bon convertisseur.

La première étape est donc d'ajouter ce convertisseur (plus exactement la Factory) à l'objet Retrofit :

```
public class RetrofitBuilder {
//Using my HttpClient
Retrofit raCustom = new Retrofit.Builder()
        .client(client)
        .baseUrl(BASE_URL)
        //add your own converter first (declaration order matters)
        //the responsability chain design pattern is behind
        .addConverterFactory(new MyPhotoConverterFactory())
        //You need to add a converter if you want your Json to be automagicly
        //convert into the object
        .addConverterFactory(MoshiConverterFactory.create())
        //then add your own CallAdapter
        .addCallAdapterFactory(new ErrorHandlingCallAdapterFactory())
        .build();
WebServerIntf webServer = raCustom.create(WebServerIntf.class);
return webServer;
```

Il faut ensuite définir la Factory :

```
public class MyPhotoConverterFactory extends Converter.Factory {

    public static MyPhotoConverterFactory create(){
        return new MyPhotoConverterFactory();
    }
  @Override
    public Converter<?, RequestBody> requestBodyConverter(Type type, Annotation[]
nnotations, Retrofit retrofit) {
        //If it's a Photo instance then convert
        if(type==Photo.class){
            return MyPhotoRequestConverter.INSTANCE;
        }
        //else use the Chain of responsability ppatern and return null
        //the api will look at the next converter
        return null;
    }
  @Override
    public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[]
nnotations, Retrofit retrofit) {
        //If it's a Photo instance then convert
        if(type==Photo.class){
            return MyPhotoResponseConverter.INSTANCE;
        }
        //else use the Chain of responsability pattern and return null
        //the api will look at the next converter
        return null;
```

```
    }
}
```

Elle étend la classe ConvertFactory et possède :

- une méthode create qui renvoie une instance d'elle-même,
- une méthode requestBodyConverter qui renvoie le convertisseur Objet vers Requête si c'est le type d'objet qu'elle prend en charge et null sinon,
- une méthode responseBodyConverter qui renvoie le convertisseur Response vers Objet si c'est le type d'objet qu'elle prend en charge et null sinon.

Il ne nous reste plus qu'à définir nos convertisseurs.

Le convertisseur Objet vers Requête :

```
public class MyPhotoRequestConverter<T> implements Converter<T, RequestBody> {
    //instance of the converter
    static final MyPhotoRequestConverter<Photo> INSTANCE
                    = new MyPhotoRequestConverter<>();
    //MIME type of the request
    private static final MediaType MEDIA_TYPE
                    = MediaType.parse("application/json; charset=utf-8");
    private Photo photo;

    //default PRIVATE empty constructor
    private MyPhotoRequestConverter() {    }

    //The real conversion from the Photo to it's json representation
    @Override
    public RequestBody convert(T value) throws IOException {
        //ensure the right object is passed to you
        if (value instanceof Photo) {
            photo = (Photo) value;
            return new RequestBody() {
                @Override
                public MediaType contentType() {return MEDIA_TYPE;}
                @Override
                public void writeTo(BufferedSink sink) throws IOException {
                    writeRequest(sink); }
            };
        } else {throw new IllegalArgumentException();}    }
private void writeRequest(BufferedSink sink) throws IOException {
        //do the Moshi stuff:
        JsonWriter jsonW = JsonWriter.of(sink);
        jsonW.setIndent("    ");
        writeJson(jsonW);
        //you have to close the JsonWrtiter too (esle nothing will happen)
        jsonW.close();
        //don't forget to close, else nothing appears
        sink.close();
    }
 private void writeJson(JsonWriter jsonW) throws IOException {
        jsonW.beginObject();
```

```
        jsonW.name("albumId").value(photo.getAlbumId());
        jsonW.name("id").value(photo.getId());
        jsonW.name("title").value(photo.getTitle());
        jsonW.name("url").value(photo.getUrl());
        jsonW.name("thumbnailUrl").value(photo.getThumbnailUrl());
        jsonW.endObject();
    }
}
```

Cette classe étend Converter<T,RequestBody>, où T est le paramètre de généricité et n'est autre que votre Objet. Laissez T et ne mettez pas votre type d'objet ou remplacez T par votre type réel, les deux marchent bien.

Il vous faut implémenter la méthode convert(T) qui renvoie une RequestBody. Pour cela, rien de plus simple, vous renvoyer un nouvel objet RequestBody dont vous surchargez les méthodes :

- contentType pour renvoyer le bon type MIME de votre requête ;
- writeTo pour écrire le contenu de votre requête.

Dans l'exemple, j'utilise Moshi pour écrire à la main le contenu de mon flux Json dans la requête au moyen des méthodes writeRequest et writeJson. Il n'y a rien de transcendant, c'est pour l'exemple, je ne pense pas que vous ayez à traiter un exemple aussi trivial dans la vrai vie.

Le convertisseur de la Réponse vers l'Objet est tout aussi simple à mettre en place :

```
public class MyPhotoResponseConverter implements Converter<ResponseBody, Photo> {
    //define an instance to retrieve the converter only once
    static final MyPhotoResponseConverter INSTANCE = new
        MyPhotoResponseConverter();

    //default empty constructor
    private MyPhotoResponseConverter() {
    }

    //The real conversion from the server response to the Photo object
    @Override
    public Photo convert(ResponseBody value) throws IOException {
        return readJson(value.source());
    }
    //Read the source, build the object and return it
    public Photo readJson(BufferedSource source) throws IOException {
        if(source==null){throw new IOException();}
        Photo photo =new Photo();
        //Then read th JSon File
        JsonReader reader = JsonReader.of(source);
        reader.beginObject();
        while (reader.hasNext()) {
            switch (reader.nextName()) {
                case "albumId":
                    photo.setAlbumId(reader.nextInt()); break;
```

```
            case "id":
                photo.setId(reader.nextInt()); break;
            case "title":
                photo.setTitle(reader.nextString()); break;
            case "url":
                photo.setUrl(reader.nextString()); break;
           case "thumbnailUrl":
                photo.setThumbnailUrl(reader.nextString()); break;
            default:break;
        }
    }
    reader.endObject();
    reader.close();
    source.close();
    return photo;   }}
```

Cette classe étend Converter<T,RequestBody> où T est le paramètre de généricité. Dans l'exemple, j'ai changé T pour lui donner son vrai type Photo et vous montrez que cela marche aussi.

Il vous suffit alors d'implémenter la méthode convert(ResponseBody value) qui renvoie un objet de type T (ou ici du type réel Photo). J'apelle simplement la méthode readJson pour reconstruire l'objet Photo encapsulé dans le Json et le renvoyer.

4.6 Call et CallAdapter personalisé

L'objet Call encapsule votre appel réseau et la réponse obtenue. Retrofit vous permet de substituer l'objet Call avec les Observable de RxJava ou de Future de manière aisée. Mais garder en tête qu'il est conçu pour les objets Call.

Comme nous l'avons vu, vous pouvez effectuer sur un objet call soit un appel synchrone (méthode execute) soit un appel asynchrone simplement (méthode enqueue). La méthode execute vous renvoyant directement l'objet, là où vous fournissez un CallBack à la méthode enqueue pour réceptionner le retour.

Pour rappel, je définis mon interface d'appels comme cela :

```
public interface WebServerIntf {
/** Get Method
(this is the way to declare them using Retrofit 2.0)
   @GET("posts/1")
   Call<Post> getPostOne();
}
```

Et je l'utilise (manière asynchrone) comme cela :

```
public class BusinessService {
 //Synchronous
Post post=getPostByIdCall.execute();
//Asynchronous
Call<Post> getPostOneCall = webService.getPostOne();
//so you need to make an async call
getPostOneCall.enqueue(new Callback<Post>() {
   @Override
```

```
    public void onResponse(Response<Post> response) {
    }
    @Override
    public void onFailure(Throwable t) {
    }
});
```

Dans cet article, je ne vous parlerai pas des Observables (RxAndroid) et de Retrofit, c'est simple à mettre en place et pensé pour mais là, je vais rester concentré sur Retrofit.

Si vous souhaitez creuser le sujet, Jake Wharton vous a fait un projet d'exemple pour leur mise en place :

https://github.com/JakeWharton/u2020

4.6.1 CallAdapter personalisé: Exemple d'un ErrorHandler

Vous pouvez ajouter vos propres CallAdapter (utile pour la mise en place d'ErrorHandler ou autre en fonction de vos besoins). C'est ultra-utile mais un peu fin à mettre en place...

L'objectif d'un CallAdapter personalisé est de mettre en place un traitement plus sophisitiqué que celui fournit nativement par Call et son CallBack qui ne possède que deux méthodes (onResponse et onFailure). Il peut y avoir de multiples besoins pour la mise d'un tel Design Pattern. L'un des plus fréquents est la gestion des erreurs réseau mais vous pouvez avoir d'autres besoins tout aussi valables.

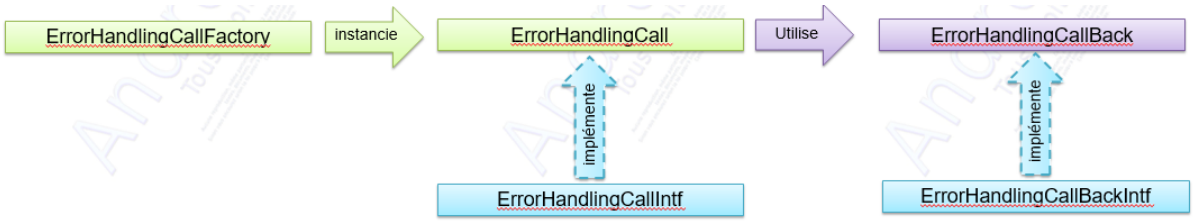
Ainsi, pour comprendre la mise en place d'un CallAdapter personalisé, nous allons mettre en place un ErrorHandlerCall basé sur ce principe. Nous souhaitons pouvoir automatiquement traiter les erreurs serveurs (404 et autres) qui ne sont pas une Failure de communication et reviennent ainsi dans la méthode onResponse si l'on ne fait rien.

L'objectif ici est d'avoir une classe Call qui nous permet de gérer ses types de retours plus finement, ainsi nous voudrions pouvoir remplacer Call et son CallBack par un Call plus fins sur la gestion des erreurs. Nous voudrions un CallBack de ce type:

```
public interface ErrorHandlingCallBack<T>{
    /** Called for [200, 300) responses. */
    void success(Response<T> response);
    /** Called for 401 responses. */
    void unauthenticated(Response<?> response);
    /** Called for [400, 500) responses, except 401. */
    void clientError(Response<?> response);
    /** Called for [500, 600) response. */
    void serverError(Response<?> response);
    /** Called for network errors while making the call. */
    void networkError(IOException e);
    /** Called for unexpected errors while making the call. */
    void unexpectedError(Throwable t);
}
```

Nous pourrons ainsi avoir une gestion plus fine du retour serveur et des erreurs associées. L'idée de mettre en place un CallAdapter personalisé est simple.

Nous allons mettre en place le Deisgn pattern suivant:



Où:

- ErrorHandlingCallFactory: Instancie l'objet ErrorHandlingCall si c'est le bon Type de requête
- ErrorHandlingCallIntf: Déclare les méthodes du Call que vous êtes en train de créer (faire attention à ce qu'elles ressemblent à celles de la classe Call pour ne pas perturber l'utilisateur)
- ErrorHandlingCall: Implémente l'interface ErrorHandlingCallIntf et met en place toutes les méthodes utiles aux appels associées à la classe Call (cancel, execute, enqueue, clone...)
- ErrorHandlingCallBackIntf : Déclare les méthodes associées aux traitements voulus
- ErrorHandlingCallBack: Effectue le traitement voulu (ici, le traitement des erreurs)

Comme je vous le disais, la mise en place d'un CallAdapter n'est pas triviale, mais ça se fait. Nous allons examiner chaque classe pour mieux comprendre son comportement et ses responsabilités. Mais tout d'abord, commençons par la déclaration de notre CallAdpater.

Nous le définissions comme d'habitude, la méthode dans l'interface d'appels, mais cette fois ci, en précisant bien que le type de retour est un ErrorHandlingCall (notre Call):

```
public interface WebServerIntf {
//this method should crash (to test the ErrorHandler)
@GET("posts/trois")
ErrorHandlingCall<Post> getPostOneWithError();
```

Pour l'instanciation, il nous faut prévenir le client Retrofit que nous avons ajouté un nouveau CallAdapter à la chaîne des adapteurs lors de l'instanciation de celui-ci:

```
public class RetrofitBuilder {
//now it's using the cach
//Using my HttpClient
Retrofit raCustom = new Retrofit.Builder()
        .client(client)
        .baseUrl(BASE_URL)
        //add your own converter first (declaration order matters)
        //the responsability chain design pattern is behind
        .addConverterFactory(new MyPhotoConverterFactory())
        //You need to add a converter if you want your Json
         .addConverterFactory(MoshiConverterFactory.create())
        //then add your own CallAdapter
        .addCallAdapterFactory(new ErrorHandlingCallAdapterFactory())
        .build();
WebServerIntf webServer = raCustom.create(WebServerIntf.class);
return webServer;
```

Ensuite, regardons le code de la Factory qui décide si elle doit renvoyer notre ErrorHandlingCall ou pas en fonction du type de retour attendu par l'appel:

```
public class ErrorHandlingCallAdapterFactory implements Factory{
   @Override
   public CallAdapter<?> get(Type returnType, Annotation[] annotations, Retrofit retrofit) {
     //tricky stuff, not proud of, but works really
      if(!returnType.toString().contains("ErrorHandlingCall")){
         //This is not handled by you, so return null
         //and enjoy the responsability chain design pattern
         return null;
      }
      //this case is yours, do your job:
      return new CallAdapter<ErrorHandlingCall<?>>() {
         @Override public Type responseType() {
            return responseType;
         }

         @Override public <R> ErrorHandlingCall<R> adapt(Call<R> call) {
            return new ErrorHandlingCall<>(call);
         }
      };
   }

}
```

La seule chose importante de cette classe est la méthode get qui doit renvoyer null si le type de retour attendu n'est pas votre ErrorHandingCall. Si le type de retour attendu est ErrorHandlingCall alors vous devez le renvoyer en surchargeant ses méthodes:

- responseType pour renvoyer votre responseType (on vous l'a passé en paramètre) et
- adapt qui est la méthode clef; elle vous permet de passer à votre ErrorHandlingCall le Call réel qui effectue l'appel réseau. En effet, sans lui, vous ne pourriez pas rerouter vos méthodes vers l'objet call sous jacent pour qu'il fasse réellement le boulot.

Maintenant, nous allons définir les traitements que nous souhaitons mettre en place pour notre propre Call, c'est à dire définir le CallBack que nous fournissons aux utilisateurs de l'ErrorHandlingCall.

Pour cela, nous les définissons dans une interface, laissant son instanciation à l'utilisateur final:

```
public interface ErrorHandlingCallBack<T>{
   /** Called for [200, 300) responses. */
   void success(Response<T> response);
   /** Called for 401 responses. */
   void unauthenticated(Response<?> response);
   /** Called for [400, 500) responses, except 401. */
   void clientError(Response<?> response);
   /** Called for [500, 600) response. */
   void serverError(Response<?> response);
   /** Called for network errors while making the call. */
   void networkError(IOException e);
   /** Called for unexpected errors while making the call. */
```

```
    void unexpectedError(Throwable t);
}
```

Maintenant, il nous faut définir les méthodes de votre objet ErrorCallHandler, pour cela je préferre les définir dans une interface et les instancier dans une classe concrète.

La définition des méthodes dans l'interface a pour objectif de mettre en place l'ensemble des méthodes de la classe Call que vous aurez adapté à vos besoins. En effet, il vous faut répondre aux méthodes naturelles de celle ci (cancel, enqueue, execute...). Les déclarer dans une interface vous permet d'avoir du recul sur votre code et de bien définir les responsabilités de votre classe Call:

```
public interface ErrorHandlingCallIntf<T> {
    /**
     * Mandatory
     * To be called before execute or enqueue
     *
     * @param callBack
     */
    void initializeCallBack(ErrorHandlingCallBack callBack);

    /**
     * Synchronously send the request and return its response.
     *
     * @throws IOException      if a problem occurred talking to the server.
     * @throws RuntimeException (and subclasses) if an unexpected error occurs
     * creating the request
     *                          or decoding the response.
     */
    Response<T> execute() throws IOException;

    /**
     * Asynchronously send the request and notify {@code callback} of its response
     * or if an error
     * occurred talking to the server, creating the request, or processing the response.
     *
     */
    void enqueue();

    /**
     * Returns true if this call has been either {@linkplain #execute() executed} or {@linkplain
     * #enqueue() enqueued}. It is an error to execute or enqueue a call more than once.
     */
    boolean isExecuted();

    /**
     * Cancel this call. An attempt will be made to cancel in-flight calls, and if the call has not
     * yet been executed it never will be.
     */
    void cancel();
```

```
    /**
     * True if {@link #cancel()} was called.
     */
    boolean isCanceled();

    /**
     * Create a new, identical call to this one which can be enqueued or executed even
     *  if this call
     * has already been.
     */
    ErrorHandlingCallIntf<T> clone();
}
```

L'instanciation de cette interface par la classe ErrorHandlingCall est basé sur le Design Pattern du Decorator. ErrorHandlingCall va contenir un objet Call et rerouter les appels vers lui puis effectuer son propre traitement.

```
public class ErrorHandlingCall<T> implements ErrorHandlingCallIntf<T> {
    /**
     * The real call beyound the this call
     */
    private final Call<T> call;

    /**
     * The call back to use to guive more granularity to the error handling to the client
     */
    private ErrorHandlingCallBack<T> errorHandlingCallBack;

    /*************************************************************
     * Constructor
     *************************************************************/
    /**
     * Used by the ErrorHandlingCallAdapterFactory
     *
     * @param call
     */
    ErrorHandlingCall(Call<T> call) {
        this.call = call;
    }

    /**
     * Used by clone
     *
     * @param errorHandlingCallBack
     * @param call
     */
    private ErrorHandlingCall(ErrorHandlingCallBack<T> errorHandlingCallBack,
        Call<T> call) {
        this.errorHandlingCallBack = errorHandlingCallBack;
```

```
    this.call = call.clone();
}

/**
 * Mandatory
 * To be called before execute or enqueue
 *
 * @param callBack
 */
@Override
public void initializeCallBack(ErrorHandlingCallBack callBack) {
    errorHandlingCallBack = callBack;
}

/*************************************************************
 *   implements interface Call<T>
 *************************************************************/

/**
 * Synchronously send the request and return its response.
 *
 * @throws IOException      if a problem occurred talking to the server.
 * @throws RuntimeException (and subclasses) if an unexpected error occurs
 * creating the request
 *                          or decoding the response.
 */
@Override
public Response<T> execute() throws IOException {
    if (errorHandlingCallBack == null) {
        throw new IllegalStateException("You have to call
            initializeCallBack(ErrorHandlingCallBack callBack) before execute");
    }
    //then analyse the response and do your expected work
    Response<T> response = call.execute();
    int code = response.code();
    if (code >= 200 && code < 300) {
        //it's ok
        return response;
    }
    //It's not ok anymore, return the response but make the errorCallBack
    else if (code == 401) {
        errorHandlingCallBack.unauthenticated(response);
    } else if (code >= 400 && code < 500) {
        errorHandlingCallBack.clientError(response);
    } else if (code >= 500 && code < 600) {
        errorHandlingCallBack.serverError(response);
    } else {
        errorHandlingCallBack.unexpectedError(new RuntimeException(
            "Unexpected response " + response));
    }
```

```
    return response;

}

/**
 * Asynchronously send the request and notify {@code callback} of its response
 * or if an error
 * occurred talking to the server, creating the request, or processing the response.
 *
 */
@Override
public void enqueue() {
    if (errorHandlingCallBack == null) {
        throw new IllegalStateException("You have to call
            initializeCallBack(ErrorHandlingCallBack callBack) before enqueue");
    }
    //do the job with th real call object
    call.enqueue(new Callback<T>() {
        /**
         * Invoked for a received HTTP response.
         * <p/>
         * Note: An HTTP response may still indicate an application-level failure
         * such as a 404 or 500.
         * Call {@link Response#isSuccess()} to determine if the response indicates success.
         *
         * @param response
         */
        @Override
        public void onResponse(Response<T> response) {
            int code = response.code();
            if (code >= 200 && code < 300) {
                errorHandlingCallBack.success(response);
            } else if (code == 401) {
                errorHandlingCallBack.unauthenticated(response);
            } else if (code >= 400 && code < 500) {
                errorHandlingCallBack.clientError(response);
            } else if (code >= 500 && code < 600) {
                errorHandlingCallBack.serverError(response);
            } else {
                errorHandlingCallBack.unexpectedError(new RuntimeException(
                    "Unexpected response " + response));
            }
        }

        /**
         * Invoked when a network exception occurred talking to the server or
         * when an unexpected
         * exception occurred creating the request or processing the response.
         *
         * @param t
```

```
         */
        @Override
        public void onFailure(Throwable t) {
            if (t instanceof IOException) {
                errorHandlingCallBack.networkError((IOException) t);
            } else {
                errorHandlingCallBack.unexpectedError(t);
            }
        }
    });
}

/**
 * Returns true if this call has been either {@linkplain #execute() executed} or {@linkplain
 * #enqueue() enqueued}. It is an error to execute or enqueue a call more than once.
 */
@Override
public boolean isExecuted() {
    return call.isExecuted();
}

/**
 * Cancel this call. An attempt will be made to cancel in-flight calls, and if the call has not
 * yet been executed it never will be.
 */
@Override
public void cancel() {
    call.cancel();
}

/**
 * True if {@link #cancel()} was called.
 */
@Override
public boolean isCanceled() {
    return call.isCanceled();
}

/**
 * Create a new, identical call to this one which can be enqueued or executed even if this
all
 * has already been.
 */
@Override
public ErrorHandlingCallIntf<T> clone() {
    if (errorHandlingCallBack == null) {
        throw new IllegalStateException("You have to call
            initializeCallBack(ErrorHandlingCallBack callBack) before clone()");
    }
    return new ErrorHandlingCall<>(errorHandlingCallBack, call);
```

```
    }
}
```

Les points clefs de cette classe sont :

- Le constructeur prend en paramètre l'objet Call sur lequel sera rerouté la plupart des méthodes;
- La méthode enqueue utilise Call pour faire l'appel et effectue un post-traitement de la réponse pour rerouter vers son propre CallBack, l'ErrorHadlingCallBack;
- La méthode execute effectue la même chose, appel puis post-traitement;
- les autres méthodes ne font que se rerouter vers l'objet call.

Enfin il ne nous reste plus qu'à utiliser notre ErrorHandlingCall.

```
public class BusinessService {
/*** The call that handles errors */
ErrorHandlingCall<Post> getPostOneWithErrorCall;
/*** The callBack that manages the errors when they appear */
ErrorHandlingCallBack errorHandlingCallBack=null;

/*** Load a stuff with an errorHandlingCall */
public void loadWithErrorHandlingCall(){
   //first initialize your error handling callback
   if(errorHandlingCallBack==null){
      errorHandlingCallBack= instanciateErrorHandlingCallBack();
   }
   //then instanciate
   getPostOneWithErrorCall=webServiceComplex.getPostOneWithError();
   //initialize your errorCallBack
   getPostOneWithErrorCall.initializeCallBack(errorHandlingCallBack);
   //make your call
   getPostOneWithErrorCall.enqueue();
}
```

Vous avez remarqué que la méthode enqueue ne prend pas en paramètre le CallBack, en effet se sont les méthodes de notre ErrorHandlingCall qui sont appelées et non pas celles de la classe Call. Il ne reste plus qu'à voir l'instanciation de notre callaBack, l'ErrorHandlingCallBack (toujours dans la même classe) :

```
private ErrorHandlingCallBack instanciateErrorHandlingCallBack(){
   return new ErrorHandlingCallBack() {
      @Override
      public void success(Response response) {
         Log.e("BusinessService", "Reponse is Success" + response.body());
      }

      @Override
      public void unauthenticated(Response response) {
         Log.e("BusinessService", "UNAUTHENTICATED !!!");

      }
```

```
        @Override
        public void clientError(Response response) {
            Log.e("BusinessService", "CLIENT ERROR " + response.code());
        }

        @Override
        public void serverError(Response response) {
            Log.e("BusinessService", "Server ERROR " + response.code());
        }

        @Override
        public void networkError(IOException e) {
            Log.e("BusinessService", "IOException ", e);
        }

        @Override
        public void unexpectedError(Throwable t) {
            Log.e("BusinessService", "Death Metal Error without roses ", t);
        }
    };
}
```

4.7 Retrofit: Mise en place du Logging

A priori, on se dit, quand on souhaite mettre en place un système de Logging qu'il nous faut mettre un CallAdapter spécialisé. Le problème est, comme nous l'avons vu, que le CallAdapter n'accède qu'à l'objet Call et cet objet ne permet pas d'accéder aux requêtes sous-jacentes. En fait, il va falloir avoir une compréhension plus fine de Retrofit. En effet, la meilleure place pour faire du logging de nos requêtes et des reponses obtenues n'est pas la couche Retrofit mais la couche Http sous-jacente.

Pour faire cela, vous pouvez, soit utiliser le logger natif concçu pour Retrofit HttpLoggingInterceptor, soit faire votre propre Logger.

4.7.1 Logger natif

Pour utiliser le logger natif, il vous faut rajouter sa librairie à votre build.gradle:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.2.0'
    //compile 'com.squareup.okhttp3:okhttp:3.0.1'<-a bug here
    compile 'com.squareup.retrofit2:retrofit:2.0.0-beta3'
    compile 'com.squareup.okhttp3:okhttp:3.0.0-RC1'
    compile 'com.squareup.retrofit2:converter-moshi:2.0.0-beta3'
    compile 'com.squareup.okhttp3:logging-interceptor:3.0.0-RC1'
}
```

Puis l'utiliser lors de l'instanciation de votre OkHttpClient:

```
public class RetrofitBuilder {
@NonNull
public static OkHttpClient getOkHttpClient(Context ctx) {
```

```
    //define the OkHttp Client with its cach!
    //Assigning a CacheDirectory
    File myCacheDir = new File(ctx.getCacheDir(), "OkHttpCache");
    //you should create it...
    int cacheSize = 1024 * 1024;
    Cache cacheDir = new Cache(myCacheDir, cacheSize);
    HttpLoggingInterceptor httpLogInterceptor = new HttpLoggingInterceptor();
    httpLogInterceptor.setLevel(HttpLoggingInterceptor.Level.BASIC);
    return new OkHttpClient.Builder()
            //add a cach
            .cache(cacheDir)
                //add interceptor (here to log the request)
            .addInterceptor(httpLogInterceptor)
            .build();
}
```

Vous pouvez lors de son instanciation définir son niveau de log.

4.7.2 Logger spécialisé

Dans ce cas, il suffit de créer son propre interceptor et de logger les requêtes qui passent au-travers lui :

```
public class CustomLoggingInterceptor implements Interceptor{
    //Code pasted from okHttp webSite itself
    @Override public Response intercept(Interceptor.Chain chain) throws IOException {
        Request request = chain.request();
        long t1 = System.nanoTime();
        Log.e("Interceptor Sample", String.format("Sending request %s on %s  %s.",
                request.url(), chain.connection(), request.headers().toString()));

        Response response = chain.proceed(request);

        long t2 = System.nanoTime();
        Log.e("Interceptor Sample", String.format("Received response for %s
                in %.1fms%n%s",
                response.request().url(), (t2 - t1) / 1e6d, response.headers()));
        return response;
    }
}
```

Une fois que vous l'avez défini, il ne vous reste plus qu'à le rajouter à votre client http:

```
public class RetrofitBuilder {
@NonNull
public static OkHttpClient getOkHttpClient(Context ctx) {
    //define the OkHttp Client with its cach!
    //Assigning a CacheDirectory
    File myCacheDir = new File(ctx.getCacheDir(), "OkHttpCache");
    //you should create it...
    int cacheSize = 1024 * 1024;
    Cache cacheDir = new Cache(myCacheDir, cacheSize);
    Interceptor customLoggingInterceptor = new CustomLoggingInterceptor();
```

```
    return new OkHttpClient.Builder()
            //add a cach
            .cache(cacheDir)
                    //add interceptor (here to log the request)
            .addInterceptor(customLoggingInterceptor)
            .build();
}
```

Et voilà.

4.8 Un conseil sur les URLs

Une bonne pratique consiste à toujours terminer vos *base url* par / et vos @URL de ne jamais commencer avec.

Pourquoi? Pour des raisons de résolution dynamique des URL par Retrofit. Ainsi si votre @URL débute par un /, le système le comprendra comme une Url relative vis-à-vis du root de BASE_URL. S'il ne commence pas par un /, le système le considèrera comme un chemin absolu à partir de votre BASE_URL.

5 Bibliographie

Les liens suivants ont été pour moi une source de compréhension de Retrofit, Moshi, OkHttp et Okio :

http://inthecheesefactory.com/blog/retrofit-2.0/en

https://github.com/square/retrofit/blob/master/samples/src/main/java/com/example/retrofit/ErrorHandlingCallAdapter.java

https://gist.github.com/rahulgautam/25c72ffcac70dacb87bd#file-errorhandlingexecutorcalladapterfactory-java

https://speakerdeck.com/jakewharton/simple-http-with-retrofit-2-droidcon-nyc-2015

http://square.github.io/retrofit/

https://futurestud.io/blog/retrofit-add-custom-request-header

https://packetzoom.com/blog/which-android-http-library-to-use.html

Mais clairement, ceux associés aux conférences de Jake Wharton à New York et Montréal en 2015, ont été les liens qui m'ont fait comprendre Retrofit, OkHttp, Moshi et Okio. Cet article utilise énormément ces conférences (les schémas en particulier, le code aussi).

https://www.youtube.com/watch?v=3WONuRSUHmw

https://www.youtube.com/watch?v=KIAoQbAu3eA

6 Conclusion

J'espère que cet article vous a plu, il est extraie d'une formation que j'ai mise en place "Ultimate Android" qui se concentre sur l'architecture Android. Normalement, suite à sa lecture vous devriez avoir pas mal de boulot sur votre application pour mettre à jour votre couche réseau et votre couche IO.

Ainsi, vous devriez:

- Remplacer vos écritures et lectures disque par Okio et utiliser des Sinks et des Sources;
- Zipper tous vos flux (vers votre serveur mais aussi lors de l'écriture sur disque);
- Utiliser OkHttpClient et ne plus utiliser DefaultHttpClient;
- Utiliser Moshi pour toutes les manipulations Json de votre application;
- Remplacer votre couche de communication par Retrofit;
- Gérer vos erreurs réseaux via un CallAdapter spécifique;
- Effectuer vos logs de communication via un Intercepteur OkHttpClient.

Je vous remercie de m'avoir lu et je vous dis à une prochaine fois pour un nouvel article sur Android, à bientôt.

7 Tutoriel

Comme toujours avec les articles d'Android2ee, un projet Github vous attend pour que vous ayez un exemple concret sur lequel s'appuie cet article et que vous puissiez voir et ré-utiliser le code:

https://github.com/MathiasSeguy-Android2EE/Square

Bonne lecture, bonne montée en compétence, que la force soit avec vous.

8 Remerciements

Je tiens ici à remercier Mickael pour son aide quotidienne à la publication de mes articles.

Je tiens aussi à remercier #ortho pour ses corrections orthographiques.

Je tiens aussi à remercier #tech pour leur relecture technique.

Je remercie aussi Android2ee de me fournir le temps de vous écrire ses articles.