Async Http Client Quick Start Guide

(Taken from http://jfarcand.wordpress.com)

Version: 1.5

The Async Http Client library purpose is to allow Java applications to easily execute HTTP requests and asynchronously process the HTTP responses. In this document I will explain how to use the library and what features are supported.

Executing request synchronously or asynchronously.

The first thing to decide when using the library is if your application can handle asynchronous response or not. If not, the library has been designed using the Future API, hence you can always execute synchronous call by blocking on the Future.get() method:

```
AsyncHttpClient client = new AsyncHttpClient();
Response response = client.prepareGet(("http://sonatype.com")
  .execute().get();
```

The above means the request will block until the full Response has been received. It also made your application's blocking, waiting for the response to comes back. This could be potentially an issue to block for every request, specially when doing POST or PUT operations where you don't necessarily need to wait for the response. A simple way consist of not calling the Future.get()

```
AsyncHttpClient client = new AsyncHttpClient();
Response response =
   client.preparePut(("http://sonatype.com/myFile.avi").execute();
```

A better way than above would consist of using an AsyncHandler. The AynchHandler API is fairly simple and just consists of 5 methods to implements:

```
public interface AsyncHandler<T>  {
    void onThrowable(Throwable t);

    STATE onBodyPartReceived(HttpResponseBodyPart bodyPart)
      throws Exception;

    STATE onStatusReceived(HttpResponseStatus responseStatus)
      throws Exception;

    STATE onHeadersReceived(HttpResponseHeaders headers)
      throws Exception;

    T onCompleted() throws Exception;
}
```

The method's order of invocation when the response start arriving consist of:

1. **onStatusReceived**: The status line has been processed.
2. **onHeadersReceived**: All response's headers has been processed.
3. **onBodyPartReceived**: A body parts has been received. This method can be invoked many time depending of the response's bytes body.
4. **onCompleted**: Invoked when the full response has been read, or if the processing get aborted (more on this below)
5. **onThrowable**: Invoked if something wrong happenned inside the previous methods or when an I/O exception occurs.

Note that for all methods onXXXReceived, the return value is an enum which can take the value of CONTINUE or ABORT. Returning CONTINUE tells the library to continue processing the response, where ABORT means stop processing the response and automatically invoke the onCompleted(). This is particularly helpful if your application just need to looks for the response's status or headers, without the need to process the entire response's body. An implementation would looks like (T can be anything):

```
AsyncHttpClient client = new AsyncHttpClient();
client.prepareGet("http://sonatype.com")
  .execute(new AsyncHandler<T>() {

     void onThrowable(Throwable t) {
     }
     public STATE onBodyPartReceived(HttpResponseBodyPart bodyPart)
       throws Exception{
        return STATE.CONTINUE;
     }
     public STATE onStatusReceived(HttpResponseStatus responseStatus)
       throws Exception {
        return STATE.CONTINUE;
     }
     public STATE onHeadersReceived(HttpResponseHeaders headers)
       throws Exception {
        return STATE.CONTINUE;
     }
     T onCompleted() throws Exception {
       return T;
   }
});
```

Creating a Request object

The AsynHttpClient uses the builder pattern when it is time to create Request object. The simplest way consist of:

```
RequestBuilder builder = new RequestBuilder("PUT");
Request request = builder..setUrl("http://")
 .addHeader("name", "value")
 .setBody(new File("myUpload.avi"))
 .build();
AsyncHttpClient client = new AsyncHttpClient();
client.execute(request, new AsyncHandler&lt;...&gt;() {
     .....
} );
```

If you need to work with File, the library supports the zero copy in memory concept, e.g the File can be uploaded or downloaded without loading its associated bytes in memory, preventing out of memory errors in case you need to upload or download many large files. Although the library support the following:

```
Request request = builder..setUrl("http://")
  .addHeader("name", "value")
  .setBody(myInputStream))
  .build();
```

it is discouraged to use InputStream as the library will need to buffer bytes in memory in order to determine the length of the stream, and instead highly recommended to either use a File or the BodyGenerator API to avoid loading unnecessary bytes in memory:

```
public interface BodyGenerator {
   Body createBody() throws IOException;
}
```

where a Body is defined as:

```
public interface Body {
    long getContentLength();
    long read(ByteBuffer buffer)
      throws IOException;
    void close() throws IOException;
}
```

This way the library will never read unnecessary bytes in memory, which could significantly improve the performance your application.
The RequestBuilder can also be used to create per Request configuration, like setting a Proxy or request timeout:

```
PerRequestConfig requestConfig = new PerRequestConfig();
requestConfig.setRequestTimeoutInMs(5 * 1000);
requestConfig.setProxy(new ProxyServer(...));
Future responseFuture =
 client.prepareGet("http://").setPerRequestConfig(requestConfig)
    .execute();
```

Creating a Response object

The AsyncHandler is typed, e.g you can return any object from the AsyncHandler.onCompleted(). One useful object of the library is the Response object and it's associate builder. You can incrementally create a Response object using the ResponseBuilder.accumulate() method:

```
MyAsyncHandler<Response> asyncHandler = new MyAsyncHanfler<Response>()
{
  private final Response.ResponseBuilder builder =
          new Response.ResponseBuilder();

  public STATE onBodyPartReceived(final HttpResponseBodyPart content)
    throws Exception {
      builder.accumulate(content);
      return STATE.CONTINUE;
  }

  public STATE onStatusReceived(final HttpResponseStatus status)
     throws Exception {
      builder.accumulate(status);
      return STATE.CONTINUE;
  }

  public STATE onHeadersReceived(final HttpResponseHeaders headers)
     throws Exception {
      builder.accumulate(headers);
      return STATE.CONTINUE;
  }

  public Response onCompleted() throws Exception {
      return builder.build();
  }
}

Response response = client.prepareGet("http://sonatype.com")
     .execute(asyncHandler).get();
```

One thing to consider when creating a Response object is the size of the response body. By default, a Response object will accumulate all response's bytes in memory, and that could potentially create an out of memory error. If you are planning to use the API for downloading large files, it is not recommended to accumulate bytes in memory and instead flush the bytes on disk as soon as they are available. Note that you can still use the Response object, except you don't accumulate the response's bytes as demonstrated below:

```
MyAsyncHandler<Response> asyncHandler = new MyAsyncHanfler<Response>()
{
   private final Response.ResponseBuilder builder =
      new Response.ResponseBuilder();

   public STATE onBodyPartReceived(final HttpResponseBodyPart content)
     throws Exception {
       content.write(myOutputStream);
       return STATE.CONTINUE;
   }

   public STATE onStatusReceived(final HttpResponseStatus status)
     throws Exception {
       builder.accumulate(status);
       return STATE.CONTINUE;
```

```
    }

    public STATE onHeadersReceived(final HttpResponseHeaders headers)
       throws Exception {
        builder.accumulate(headers);
        return STATE.CONTINUE;
    }

    public Response onCompleted() throws Exception {
        return builder.build();
    }
}

Response response = client.prepareGet("http://sonatype.com")
    .execute(asyncHandler).get();
```

Note that in the above scenario invoking Response.getResponseBodyAsStream() or getResponseBody() will return an IllegalStateException because the body wasn't accumulated by the Response object.

Configuring the AsyncHttpClient: Compression, Connection Pool, Proxy, Times out, Thread Pools, Security, etc.

You can configure the AsyncHttpClient class using the AsyncHttpClientConfig's Builder:

```
Builder builder = new AsyncHttpClientConfig.Builder();
builder.setCompressionEnabled(true)
    .setAllowPoolingConnection(true)
    .setRequestTimesout(30000)
    .build();
AsyncHttpClient client = new AsyncHttpClient(builder.build());
```

You can set the ExecutorServices as well if you don't want to use the default, which is a cached threads pool:

```
Builder builder = new AsyncHttpClientConfig.Builder();
builder.setExecutorService(myOwnThreadPool);
AsyncHttpClient client = new AsyncHttpClient(builder.build());
```

You can also configure the connection pool the library is using and implement your own polling strategy:

```
Builder builder = new AsyncHttpClientConfig.Builder();
builder.setConnectionsPool(new ConnectionsPoo<U,V>() {
      public boolean offer(U uri, V connection) {...}
      public V poll(U uri)  {...}
      public boolean removeAll(V connection)  {...}
      public boolean canCacheConnection()  {...}
      public void destroy()  {...}
 });

AsyncHttpClient client = new AsyncHttpClient(builder.build());
```

It is recommended to use the default connections pool for performance reason, but you are always free to design a better one.
You can also set the SSL information, Filters, etc. Those topics will be covered inside their own section.

Configuring SSL

Configuring the library to support SSL is simple. By default you don't have to configure anything if you don't need to use your own certificates etc.

```
    AsyncHttpClient client = new AsyncHttpClient();
    Response response = client.prepareGet(("https://sonatype.com")
      .execute().get();
```

The library will detect it's an SSL request and appropriately locate the key store, trust store etc. If you need to configure those objects, all you need to do is to create an SSLContext and set it using the AsyncHttpClient's Builder as showed below:

```
  InputStream keyStoreStream = ....
  char[] keyStorePassword = "changeit".toCharArray();
  KeyStore ks = KeyStore.getInstance("JKS");
  ks.load(keyStoreStream, keyStorePassword);

  char[] certificatePassword = "changeit".toCharArray();
  KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
  kmf.init(ks, certificatePassword);

  KeyManager[] keyManagers = kmf.getKeyManagers();
  TrustManager[] trustManagers = new
TrustManager[]{DUMMY_TRUST_MANAGER};
  SecureRandom secureRandom = new SecureRandom();

  SSLContext sslContext = SSLContext.getInstance("TLS");
  sslContext.init(keyManagers, trustManagers, secureRandom);
  Builder builder = new AsyncHttpClientConfig.Builder();
  builder.setSSLContext(myOwnThreadPool);
  AsyncHttpClient client = new AsyncHttpClient(builder.build());
```

Using Filters

The library supports three types of Filter who can intercept, transform, decorate and replay transactions: Request, Response and IOException.

Request Filter

Request Filters are useful if you need to manipulate the Request or AsyncHandler object before the request is made. As an example, you can throttle requests using the following RequestFilter implementation:

```
public class ThrottleRequestFilter implements RequestFilter {
    private final int maxConnections;
    private final Semaphore available;
    private final int maxWait;

    public ThrottleRequestFilter(int maxConnections) {
        this.maxConnections = maxConnections;
        this.maxWait = Integer.MAX_VALUE;
        available = new Semaphore(maxConnections, true);
    }

    public ThrottleRequestFilter(int maxConnections, int maxWait) {
        this.maxConnections = maxConnections;
        this.maxWait = maxWait;
        available = new Semaphore(maxConnections, true);
    }

    public FilterContext filter(FilterContext ctx) throws
FilterException {

        try {
            if (!available.tryAcquire(maxWait, TimeUnit.MILLISECONDS))
{
                throw new FilterException(
                    String.format("No slot available for Request %s "
                            "with AsyncHandler %s",
                            ctx.getRequest(), ctx.getAsyncHandler()));
            };
        } catch (InterruptedException e) {
            throw new FilterException(
                    String.format("Interrupted Request %s" +
                         "with AsyncHandler %s",
                            ctx.getRequest(), ctx.getAsyncHandler()));
        }

        return new FilterContext(
             new AsyncHandlerWrapper(ctx.getAsyncHandler()),
ctx.getRequest());
    }

    private class AsyncHandlerWrapper implements AsyncHandler<T> {

        private final AsyncHandler asyncHandler;

        public AsyncHandlerWrapper(AsyncHandler asyncHandler) {
            this.asyncHandler = asyncHandler;
        }

        public void onThrowable(Throwable t) {
```

```
            asyncHandler.onThrowable(t);
        }

        public STATE onBodyPartReceived(HttpResponseBodyPart bodyPart)
                throws Exception {
            return asyncHandler.onBodyPartReceived(bodyPart);
        }

        public STATE onStatusReceived(HttpResponseStatus
responseStatus)
                throws Exception {
            return asyncHandler.onStatusReceived(responseStatus);
        }

        public STATE onHeadersReceived(HttpResponseHeaders headers)
                throws Exception {
            return asyncHandler.onHeadersReceived(headers);
        }

        public T onCompleted() throws Exception {
            available.release();
            return asyncHandler.onCompleted();
        }
    }
```

In the above, we decorate the original AsyncHandler and use semaphore to throttle requests. To add RequestFilter, all you need to do is to configure it on the AsyncHttpClientConfig:

```
AsyncHttpClientConfig.Builder b =
                  new AsyncHttpClientConfig.Builder();
b.addRequestFilter(new ThrottleRequestFilter(100));
AsyncHttpClient c = new AsyncHttpClient(b.build());
```

Response Filter

Like with Request, you can also filter the Response's bytes before an AsyncHandler gets called. Response Filters are always invoked before the library executes the logic for authentication, proxy challenging, redirection etc. That means an application can takes control of those operations at any moment using a Response Filter. As an example, the following Response Filter redirect request from google.ca to google.com in case .ca is not responding:

```
AsyncHttpClientConfig.Builder b = new AsyncHttpClientConfig.Builder();
b.addResponseFilter(new ResponseFilter() {

   public FilterContext filter(FilterContext ctx) throws
FilterException {

      if ( ctx.getResponseStatus().getStatusCode() == 503 ) {
          return new FilterContext.FilterContextBuilder(ctx)
                      .request(new RequestBuilder("GET")
                      .setUrl("http://google.com").build())
                      .build();
                }
            }});

AsyncHttpClient c = new AsyncHttpClient(b.build());
```

IOException Filter

The AsyncHttpClient library support IOExceptionFilter that can be used to replay a request in case server a server goes down or unresponsive, a network outage occurs, or nay kind of I/O abnormal situation. In those cases, the library will catch the IOException and delegate the IOException handling to the Filter. As an example, the following filter will resume an interrupted download instead of restarting downloading the file from the beginning:

```
    AsyncHttpClient c = new AsyncHttpClient(
          new AsyncHttpClientConfig.Builder()
            .addIOExceptionFilter(
              new ResumableIOExceptionFilter()).build());

    Response r = c.prepareGet("http://host:port/LargeFile.avi")
       .execute(new AsyncHandler(){...}).get();
```

The IOExceptionFilter is defined as

```
public class ResumableIOExceptionFilter implements IOExceptionFilter {
    public FilterContext filter(FilterContext ctx) throws
FilterException {
        if (ctx.getIOException() != null ) {
            Request request = new RequestBuilder(ctx.getRequest())
                .setRangeOffset(file.length());
            return new FilterContext.FilterContextBuilder(ctx)
               .request(request)
               .replayRequest(true)
               .build();
        }
        return ctx;
    }
}
```

In the above we just catch any IOException and replay the request using the Range header to tell the remote server to restart sending bytes at that position. This way we don't need to re download the entire file.

Uploading file: Progress Listener

When uploading bytes, an application might need to take some action depending on where the upload status is. The AsyncHttpClient library support a special AsyncHandler called ProgressAsyncHandler that can be used to track the upload operation:

```
public interface ProgressAsyncHandler<T> extends AsyncHandler<T> {
    STATE onHeaderWriteCompleted();
    STATE onContentWriteCompleted();
    STATE onContentWriteProgress(long amount, long current, long
total);
}
```

The methods are called in the following order:

- onHeaderWriteCompleted: invoked when the headers has been flushed to the remote server
- onContentWriteProgress: as soon as some response's body bytes are written. Might be invoked many times.
- onContentWriteCompleted: invoked when the response has been sent or aborted.

Like with AsyncHandler, you can always always abort the processing at any moment in the upload process

Configuring Authentication: BASIC, DIGEST or NTLM

Configuring authentication with AsyncHttpClient is simple. You can configure it at the Request level using the RealmBuilder:

```
AsyncHttpClient client = new AsyncHttpClient();
Realm realm = new Realm.RealmBuilder()
            .setPrincipal(user)
            .setPassword(admin)
            .setUsePreemptiveAuth(true)
            .setScheme(AuthScheme.BASIC)
            .build();
client.prepareGet("http://...").setRealm(realm).execute();
```

You can also set the realm at the AsyncHttpClientConfig level:

```
Builder builder = new AsyncHttpClientConfig.Builder();
Realm realm = new Realm.RealmBuilder()
            .setPrincipal(user)
            .setPassword(admin)
            .setUsePreemptiveAuth(true)
            .setScheme(AuthScheme.BASIC)
            .build();
builder.setRealm(realm).build();
AsyncHttpClient client = new AsyncHttpClient(builder.build());
```

The authentication type supported are **BASIC**, **DIGEST** and **NTLM**. You can also customize your own authentication mechanism by using the Response Filter.

Configuring a Proxy

The AsyncHttpClient library supports proxy, proxy authentication and proxy tunneling. Just need to create a ProxyServer instance:

```
AsyncHttpClient client = new AsyncHttpClient();
        Future<Response> f = client
                .prepareGet("http://....)
                .setProxyServer(new ProxyServer("127.0.0.1", 8080))
                .execute();
```

If you need to use an SSL tunnel, all you need to do is:

```
 ProxyServer ps =
          new ProxyServer(ProxyServer.Protocol.HTTPS, "127.0.0.1",
8080);
 AsyncHttpClient asyncHttpClient = new AsyncHttpClient();
 RequestBuilder rb = new RequestBuilder("GET")
             .setProxyServer(ps)
             .setUrl("https://twitpic.com:443");

  Future responseFuture = asyncHttpClient
               .executeRequest(rb.build(), new
AsyncCompletionHandlerBase() {

    @Override
    public void onThrowable(Throwable t) {}

    @Override
    public Response onCompleted(Response response) throws Exception {
      return response;
    }});

   Response r = responseFuture.get();
```

You can also set the authentication token on the ProxyServer instance

```
ProxyServer ps = new ProxyServer(ProxyServer.Protocol.HTTPS,
                                 "127.0.0.1",
                                 8080,
                                 "admin",
                                 "password");
  AsyncHttpClient asyncHttpClient = new AsyncHttpClient();
  RequestBuilder rb = new RequestBuilder("GET")
        .setProxyServer(ps).setUrl("https://twitpic.com:443");

  Future responseFuture = asyncHttpClient
    .executeRequest(rb.build(), new AsyncCompletionHandlerBase() {
    @Override
    public void onThrowable(Throwable t) {}

    @Override
    public Response onCompleted(Response response) throws Exception {
      return response;
    }});

  Response r = responseFuture.get();
```

You can also set the ProxyServer at the AsyncHttpClientConfig level. In that case, all request will share the same proxy information.

Switching Provider

By default, the AsyncHttpClient is using the powerful Netty's framework as the HTTP processor. There might be environment where you can't use Netty. Fortunately, the AsyncHttpClient library supports two other http runtime: the JDKAsyncHttpProvider, which build around the URLConnection, and ApacheAsyncHttpProvider which build on top of the Apache HttpClient. To change provider, all you need to do is:

```
 AsyncHttpClient client = new AsyncHttpClient(
    new ApacheAsyncHttpProvider(new
AsyncHttpClientConfig.Builder().build()));
```

Same for the JDK:

```
 AsyncHttpclient client = new AsyncHttpClient(
    new JDKAsyncHttpProvider(new
AsyncHttpClientConfig.Builder().build()));
```

Also every AsyncHttpClientProvider can be configured with their native functionality. As an example, you can switch the NettyAsyncHttpProvider to use blocking I/O instead of NIO:

```
 NettyAsyncHttpProviderConfig config = new
NettyAsyncHttpProviderConfig();
 config.setProperty(NettyAsyncHttpProviderConfig.USE_BLOCKING_IO,
"true");

 AsyncHttpClientConfig c =
    new AsyncHttpClientConfig()
      .setAsyncHttpClientProviderConfig(config).build();

 AsyncHttpClient client = new AsyncHttpClient(
    new NettyAsyncHttpProvider(config));
```

Using the WebDav protocol

The AsyncHttpClient has build in support for the WebDav protocol. The API can be used the same way normal HTTP request are made, and everything discussed in this blog works with WebDAV as well:

```
 AsyncHttpClient c = new AsyncHttpClient();
 Request mkcolRequest = new RequestBuilder("MKCOL")
      .setUrl("http://host:port/folder1").build();
 Response response = c.executeRequest(mkcolRequest).get();
```

Or

```
  AsyncHttpClient c = new AsyncHttpClient();
  Request propFindRequest = new RequestBuilder("PROPFIND")
     .setUrl("http://host:port).build();
  Response response = c.executeRequest(propFindRequest, new
AsyncHandler(){...}).get();
```

Resumable Dowload

The AsyncHttpClient supports resumable download in two differents scenarios:

- **IOException**: If an IOException occurs (for whatever reason), you can configure the library to restart the download automatically without having to restart the download from the beginning.
- **JVM crashes**: If your application or the JVM goes down during a file download, the library can also restart the download automatically when the same download is requested.

You can configure the AsyncHttpClient Library to survive IOException using the IOException Filter:

```
AsyncHttpClient c = new AsyncHttpClient(
       new AsyncHttpClientConfig.Builder()
              .addIOExceptionFilter(
                     new ResumableIOExceptionFilter()).build());
ResumableAsyncHandler a = new ResumableAsyncHandler(
new ResumableRandomAccessFileListener());
a.setResumableListener(
   new ResumableRandomAccessFileListener(
      new RandomAccessFile( "file.avi", "rw" ) ) );
Response r = c.prepareGet("http://host:port/file.avi")
   .execute(a).get();
```

If you need something more high level and configurable, you can use a ResumableAsyncHandler, and or implement a ResumableProcessor:

```
AsyncHttpClient c = new AsyncHttpClient();
ResumableAsyncHandler a =
  new ResumableAsyncHandler(
    new PropertiesBasedResumableProcessor() );
a.setResumableListener(
  new ResumableRandomAccessFileListener(
      new RandomAccessFile( "file.avi", "rw" ) ) );
Response r = c.prepareGet( "http://localhost:8081/file.AVI" )
       .execute( a ).get();
```

You can also simply use a ResumableListener (or use the ResumableRandomAccessFileListener, which does what's described below):

```
public interface ResumableListener {
    public void onBytesReceived(ByteBuffer byteBuffer)
       throws IOException;
   public void onAllBytesReceived();
   public long length();
}
```

As simple as:

```
AsyncHttpClient c = new AsyncHttpClient();
final RandomAccessFile file =
   new RandomAccessFile( "file.avi", "rw" );
ResumableAsyncHandler a = new ResumableAsyncHandler();
a.setResumableListener( new ResumableListener() {
    public void onBytesReceived(ByteBuffer byteBuffer)
       throws IOException {
      file.seek( file.length() );
      file.write( byteBuffer.array() );
```

```
      }

      public void onAllBytesReceived() {
        file.close();
      }

      public long length() {
        return file.length();
      }
} );

  Response r = c.prepareGet( "http://localhost:8081/file.AVI" )
         .execute( a ).get();
```

Make it simple: TransferListener

In some scenario an application may need to manipulate the received bytes in more than one place, e.g. saves the bytes on disk but also accumulate it for checksum checking later. In that case, instead of using an AsyncHandler and mixes logic inside an AsyncHandler.onBodyPartReceived, it is recommended to use the TransferListener simple API:

```
   public interface TransferListener {
      public void onRequestHeadersSent
         (FluentCaseInsensitiveStringsMap headers);

   public void onResponseHeadersReceived
         (FluentCaseInsensitiveStringsMap headers);

   public void onBytesReceived(ByteBuffer buffer)
      throws IOException;

   public void onBytesSent(ByteBuffer buffer);

   public void onRequestResponseCompleted();

   public void onThrowable(Throwable t);
}
```

All you need to do in that case is to create a TransferCompletionHandler and add as many TransferListener as you need:

```
 AsyncHttpClient client = new AsyncHttpClient();
 TransferCompletionHandler tl = new TransferCompletionHandler();
  tl.addTransferListener(new TransferListener(){...});
  Response response = httpClient.prepareGet("http://...")
     .execute(tl).get();
```

Zero Bytes Copy

When uploading or downloading bytes, it is important to try to avoid buffering bytes in memory.

Upload

On the upload side, the mechanism is enabled by default when setting the Request's body to a File:

```
AsyncHttpClient client = new AsyncHttpClient();
File file = new File("file.avi");
Future f = client.preparePut("http://localhost")
  .setBody(file).execute();
```

If you can't use a File, the recommended way is to use a BodyGenerator. It is strongly recommended to avoid using InputStream as the library will unfortunately buffer the entire content in memory in order to set the content-lenght, which can cause out of memory error.

Download

On the download side, you can use the HttpResponseBodyPart.writeTo to avoid loading bytes in memory and unnecessary copy:

```
AsyncHttpClient client = new AsyncHttpClient();
File tmp = new File("zeroCopy.txt");
final FileOutputStream stream = new FileOutputStream(tmp);
Future f = client.prepareGet("http://localhost/largefile.avi")
  .execute(new AsyncHandler() {

     public void onThrowable(Throwable t) {
     }

     public STATE onBodyPartReceived(HttpResponseBodyPart bodyPart)
        throws Exception {
         bodyPart.writeTo(stream);
         return STATE.CONTINUE;
     }

     { .... } });

     Response resp = f.get();
```

Limiting the number of connections to improve raw performance

By default the library uses a connection pool and re-use connections as needed. It is important to not let the connection pool grow too large as it takes resources in memory. One way consist of setting the maximum number of connection per host or in total:

```
AsyncHttpClientConfig config = new AsyncHttpClientConfig.Builder()
    .setMaximumConnectionsPerHost(10)
    .setMaximumConnectionsTotal(100)
    .build();
AsyncHttpClient c = new AsyncHttpClient(config);
```

There is no magic number, so you will need to try it and decide which one gives the best result.

Using OAuth

You can use the library to pull data from any OAuth site (like Twitter). This is as simple as:

```
 private static final String CONSUMER_KEY = "dpf43f3p2l4k3l03";
 private static final String CONSUMER_SECRET = "kd94hf93k423k
f44";
 public static final String TOKEN_KEY = "nnch734d00sl2jdk";
 public static final String TOKEN_SECRET = "pfkkdhi9sl3r4s00";
 public static final String NONCE = "kllo9940pd9333jh";
 final static long TIMESTAMP = 1191242096;

 public void oAuth() {
   ConsumerKey consumer =
     new ConsumerKey(CONSUMER_KEY, CONSUMER_SECRET);
   RequestToken user =
     new RequestToken(TOKEN_KEY, TOKEN_SECRET);
   OAuthSignatureCalculator calc =
     new OAuthSignatureCalculator(consumer, user);

   AsyncHttpClient client = new AsyncHttpClient();
   Response response = client.prepareGet("http://...")
       .setSignatureCalculator(calc).execute().get();
```