Java Nio Async HTTP Client Example

examples.javacodegeeks.com/core-java/nio/java-nio-async-http-client-example/

This article is an example of how to build a simple asynchronous Http client using Java Nio. This example will make use of the httpbin service for much of it's test cases, which can also be verified via postman or curl. Although the examples work, this is by no means a production ready. The exhaustive Http client implementation was merely an exercise in attempting to implement an Http client using Java Nio in an asynchronous manner. This example does not support redirect instructions (3.xx). For production ready implementations of Http clients, I recommend Apache's Asynchronous Http client or if your'e patient Java 9 has something in the works.

1. Introduction

So how does an Http Client make a request to a server and what is involved?

The client opens a connection to the server and sends a request. Most of the time this is done via a browser, obviously in our case this custom client is the culprit. The request consists of:

- Method (GET, PUT, POST, DELETE)
- URI (/index.html)
- Protocol version (HTTP/1.0)

Header line 1

```
GET /
HTTP/1.1
```

A series of headers (meta information) is following, describing to the server what is to come:

Headers

```
Host:
httpbin.org
```

```
Connection: keep-
alive
```

```
Upgrade-Insecure-Requests:
1
```

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/59.0.3071.104 Safari/537.36
```

```
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
```

```
                     , deflate,
Accept-Encoding:gzipbr
```

```
Accept-Language: en-US,en;q=0.8, nl;q=0.6
Cookie: _gauges_unique_month=1; _gauges_unique_year=1; _gauges_unique=1; _gauges_unique_hour=1; _gauges_unique_day=1
```

Following the headers (terminated by `\r\n\r\n`) comes the body, if any.

2. Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

3. Overview

The sample program is a very simple asynchronous implementation of an Http client that uses Java Nio. The functionality of the client is tested via test cases which make requests against httpbin which simply echoes back what our request was. In the event of a bad request (400) it will respond accordingly. For the `put`and `post` requests the body content is hard coded to be `text/plain`.

4. The program

NioAsyncHttpClient

```
public final class NioAsyncHttpClient implements AutoCloseable {

    private static final int PORT = 80;

    private AsynchronousChannelGroup httpChannelGroup;

    public static NioAsyncHttpClient create(final AsynchronousChannelGroup httpChannelGroup) {
        return new NioAsyncHttpClient(httpChannelGroup);
```

```java
    }

    private NioAsyncHttpClient(final AsynchronousChannelGroup httpChannelGroup) {
        Objects.requireNonNull(httpChannelGroup);

        this.httpChannelGroup = httpChannelGroup;
    }

    public void get(final String url, final String headers, final Consumer<? super ByteBuffer> success, final Consumer<? super Exception> failure)
            throws URISyntaxException, IOException {
        Objects.requireNonNull(url);
        Objects.requireNonNull(headers);
        Objects.requireNonNull(success);
        Objects.requireNonNull(failure);

        process(url, Optional.<ByteBuffer>empty(), headers, success, failure);
    }

    public void post(final String data, final String url, final String headers, final Consumer<? super ByteBuffer> success, final Consumer<? super Exception> failure)
            throws URISyntaxException, IOException {
        Objects.requireNonNull(data);
        Objects.requireNonNull(url);
        Objects.requireNonNull(headers);
        Objects.requireNonNull(success);
        Objects.requireNonNull(failure);

        process(url, Optional.of(ByteBuffer.wrap(data.getBytes())), headers, success, failure);
    }

    @Override
    public void close() throws Exception {
        this.httpChannelGroup.shutdown();
    }

    private void process(final String url, final Optional<ByteBuffer> data, final String headers, final Consumer<? super ByteBuffer> success,
                              Exception>
          final Consumer<?super failure)         throws
IOException, URISyntaxException
{

      assert
StringUtils.isNotEmpty(url) && !Objects.isNull(data) &&
StringUtils.isNotEmpty(headers) && !Objects.isNull(success) &&
!Objects.isNull(failure);


          URI uri
      final =        new URI(url);

          SocketAddress serverAddress
      final =                         new
InetSocketAddress(getHostName(uri),
PORT);

          RequestHandler handler
      final =                     new
RequestHandler(AsynchronousSocketChannel.open(this
.httpChannelGroup), success,
failure);


doConnect(uri, handler, serverAddress,
ByteBuffer.wrap(createRequestHeaders(headers, uri).getBytes()), data);

    }


                               URI          RequestHandler
    private void doConnect(final uri,    final handler,              final
SocketAddress               ByteBuffer              Optional<ByteBuffer> body)
address,              final headers,          final {

      assert
!Objects.isNull(uri) && !Objects.isNull(handler) && !Objects.isNull(address) &&
!Objects.isNull(headers);

```

```
        handler.getChannel().connect(address,null,new CompletionHandler<Void, Void>() {

            @Override
            public void completed(final Void result, final Void attachment) {
                handler.headers(headers, body);
            }

            @Override
            public void failed(final Throwable exc, final Void attachment) {
                handler.getFailure().accept(new Exception(exc));
            }
        });
    }

    private String createRequestHeaders(final String headers, final URI uri) {
        assert StringUtils.isNotEmpty(headers) && !Objects.isNull(uri);

        return headers + "Host: " + getHostName(uri) + "\r\n\r\n";
```

```
75      }
76
77      private String getHostName(final URI uri) {
78          assert !Objects.isNull(uri);
79
80          return uri.getHost();
81      }
82  }
```

- line 57-68: calls connect on the AsynchronousSocketChannel and passes a CompletionHandler to it. We make use of a custom `RequestHandler`to handle success and failure as well as to provide the reading and writing semantics for the headers, body and response.
- line 74: the `\r\n\r\n` sequence of characters signal to the server the end of the headers section meaning anything that follows should be body content and should also correspond in length to the `Content-Length` header attribute value

RequestHandler

```
001 final class RequestHandler {
002
003     private final AsynchronousSocketChannel channel;
004     private final Consumer<? super ByteBuffer> success;
005     private final Consumer<? super Exception> failure;
006
007     RequestHandler(final AsynchronousSocketChannel channel, final Consumer<? super ByteBuffer> success, final Consumer<? super Exception> failure) {
        assert
    !Objects.isNull(channel) && !Objects.isNull(success) &&
    !Objects.isNull(failure);


            .channel =
        thischannel;

            .success =
        thissuccess;

            .failure =
        thisfailure;

    }


    AsynchronousSocketChannel getChannel()
    {

        return this.channel;

    }


                  ByteBuffer> getSuccess()
    Consumer<?super {

        return this.success;

    }


                  Exception> getFailure()
    Consumer<?super {

        return this.failure;

    }
```

```java

    void closeChannel() {
        try {
            this.channel.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    void headers(final ByteBuffer headers, final Optional<ByteBuffer> body) {
        assert !Objects.isNull(headers);

        this.channel.write(headers, this, new CompletionHandler<Integer, RequestHandler>() {

            @Override
            public void completed(final Integer result, final RequestHandler handler) {
                if (headers.hasRemaining()) {
                    RequestHandler.this.channel.write(headers, handler, this);
```

```
                          (body.isPresent())
                     }else if {
                             RequestHandler.this.body(body.get(), handler);
                     }else {
                             RequestHandler.this.response();
                     }
                 }

                 @Override
                                                                  RequestHandler handler)
                 public void failed(final Throwable exc,final {
                     handler.getFailure().accept(new Exception(exc));
                     RequestHandler.this.closeChannel();
                 }
             });
         }

                         ByteBuffer          RequestHandler handler)
         void body(final body,            final {
                   !Objects.isNull(body) &&
             assert !Objects.isNull(handler);

                 .channel.write(body,
             thishandler,                     new
     CompletionHandler<Integer, RequestHandler>()
     {
```

```java

            @Override
            public void completed(final Integer result, final RequestHandler handler) {
                if (body.hasRemaining()) {
                    RequestHandler.this.channel.write(body, handler, this);
                } else {
                    RequestHandler.this.response();
                }
            }

            @Override
            public void failed(final Throwable exc, final RequestHandler handler) {
                handler.getFailure().accept(new Exception(exc));
                RequestHandler.this.closeChannel();
            }
        });
    }

    void response() {

        final ByteBuffer buffer = ByteBuffer.allocate(2048);
        this.channel.read(buffer,this,new CompletionHandler<Integer, RequestHandler>()
{

            @Override
            public void completed(final Integer result, final RequestHandler handler)
{
                if (result > 0){
                    handler.getSuccess().accept(buffer);
                    buffer.clear();

                    RequestHandler.this.channel.read(buffer, handler, this);
                }else if (result < 0){
                    RequestHandler.this.closeChannel();
                }else {
                    RequestHandler.this.channel.read(buffer, handler, this);
                }
            }

```

```
                @Override
                public void failed(final Throwable exc, final RequestHandler handler) {
                    handler.getFailure().accept(new Exception(exc));
                    RequestHandler.this.closeChannel();
                }
            });
        }
}
```

The `RequestHandler` is responsible for executing the reading and writing of headers, body and responses. It is injected with 2 `Consumer` callbacks, one for success and the other for failure. The success `Consumer` callback simply console logs the output and the failure `Consumer` callback will print the stacktrace accordingly.

Snippet of test case

```
@Test
public void get() throws Exception {
    doGet(() -> "https://httpbin.org/get", () -> String.format(HEADERS_TEMPLATE, "GET", "get", "application/json", String.valueOf(0)));
}

private void doGet(final Supplier<? extends String> url, final Supplier<? extends String> headers) throws Exception {

        final WritableByteChannel target = Channels.newChannel(System.out);
        final AtomicBoolean pass = new AtomicBoolean(true);
        final CountDownLatch latch = new CountDownLatch(1);

        try (NioAsyncHttpClient client = NioAsyncHttpClient.create(this.asynchronousChannelGroup)) {
            client.get(url.get(), headers.get(), (buffer) -> {
                try {
                    buffer.flip();

                    while (buffer.hasRemaining()) {
                        target.write(buffer);
                    }
                } catch (IOException e) {
                    pass.set(false);
                } finally {
                    latch.countDown();
                }
            }, (exc) -> {
                exc.printStackTrace();
```

```
27                pass.set(false);
28                latch.countDown();
29            });
30        }
31
32        latch.await();
33        assertTrue("Test failed", pass.get());
34    }
```

- line 13-29: we invoke get in this test case supplying the url and the headers. A success `Consumer`and failure `Consumer` callback are supplied when the response is read from the server or when an exception occurs during processing.

Test case output

```
HTTP/1.1 200 OK
Connection: keep-alive
Server: meinheld/0.6.1
Date: Tue, 20 Jun 2017 18:36:56 GMT
Content-Type: application/json
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials:true
X-Powered-By: Flask
```

```
X-Processed-Time:
0.00129985809326
Content-Length:
228
Via: 1.1
vegur

{
      :
  "args"{},
        :
  "headers"{
    "Accept":"application/json",
    "Connection":"close",
    "Content-Type":"text/plain",
    "Host":"httpbin.org"
  },
  "origin":"105.27.116.66",
  "url":"http://httpbin.org/get"
}
```

The output is the response from the httpbin service which is console logged by our success `Consumer` callback.

5. Summary

In this example we briefly discussed what's involved with an Http request and then demonstrated an asynchronous http client built using Java Nio. We made a use of a 3rd party service httpbin to verify our client's calls.

6. Download the source code

This was a Java Nio Async HTTP Client Example.

Download

You can download the full source code of this example here: **Java Nio Async HTTP Client Example**