Overview

1. Introduction
2. Fundamentals
3. Interceptors, Processors
4. Connections
5. Handlers
HttpCore - Introduction

• HttpCore is the basis of the Apache HttpComponents project which is responsible for creating and maintaining a toolset of low level Java components focused on HTTP and associated protocols
  – functions under the Apache Software Foundation

• HttpCore is a set of low level HTTP transport components that can be used to build custom client and server side HTTP services

• Its goal is to implement most of the fundamental HTTP transport aspects
HttpCore - Introduction

• An API for
  – building client / proxy / server side HTTP services
  – building both synchronous and asynchronous HTTP services

• HttpCore supports two I/O models:
  – blocking I/O model based on the classic Java I/O
  – non-blocking, event driven I/O model based on Java NIO

• Dependency information (Maven):

  <dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpcore</artifactId>
  <version>4.4.4</version>
  </dependency>
Overview

1. Introduction
2. Fundamentals
3. Interceptors, Processors
4. Connections
5. Handlers
HTTP Request Message

- HTTP request is a message sent from the client to the server
- The first line of that message includes the method to apply to the resource, the identifier of the resource, and the protocol version in use

```java
import org.apache.http.HttpRequest;
import org.apache.http.message.BasicHttpRequest;

HttpRequest request = new BasicHttpRequest("GET", "/", HttpVersion.HTTP_1_1);

System.out.println(request.getRequestLine().getMethod());
System.out.println(request.getRequestLine().getUri());
System.out.println(request.getProtocolVersion());
System.out.println(request.getRequestLine().toString());

Output:
GET / HTTP/1.1
GET / HTTP/1.1
```
HTTP Response Message

- HTTP response is a message sent by the server back to the client after having received and interpreted a request message.
- The first line of that message consists of the protocol version followed by a numeric status code and its associated textual phrase.

```
HttpResponse response =
    new BasicHttpResponse(HttpVersion.HTTP_1_1, HttpStatus.SC_OK, "OK");

System.out.println(response.getProtocolVersion());
System.out.println(response.getStatusLine().getStatusCode());
System.out.println(response.getStatusLine().getReasonPhrase());
System.out.println(response.getStatusLine().toString());
```

Output:

```
HTTP/1.1
200
OK
HTTP/1.1 200 OK
```
• The AbstractHttpMessage class contains common functionalities (setHeader, getHeaders, etc.)
An HTTP message can contain a number of headers describing properties of the message such as the content length, content type, and so on.

HttpCore provides methods to retrieve, add, remove, and enumerate such headers.

```java
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1, HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie", "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie", "c2=b; path="/"; c3=c; domain="localhost""");

Header h1 = response.getFirstHeader("Set-Cookie");
System.out.println(h1);

Header h2 = response.getLastHeader("Set-Cookie");
System.out.println(h2);

Header[] hs = response.getHeaders("Set-Cookie");
System.out.println(hs.length);
```

Output:

```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/"; c3=c; domain="localhost"
2
```
HTTP Message – Common Methods

- There is an efficient way to obtain all headers of a given type using the `HeaderIterator` interface

```java
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1, HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie", "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie", "c2=b; path="/", c3=c; domain="localhost"");

HeaderIterator it = response.headerIterator("Set-Cookie");

while (it.hasNext()) {
    System.out.println(it.next());
}
```

Output:
```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
```
It also provides convenience methods to parse HTTP messages into individual header elements

```java
HeaderElementIterator it =
    new BasicHeaderElementIterator(response.headerIterator("Set-Cookie"));

while (it.hasNext()) {
    HeaderElement elem = it.nextElement();

    System.out.println(elem.getName() + " = " + elem.getValue());

    NameValuePair[] params = elem.getParameters();
    for (int i = 0; i < params.length; i++) {
        System.out.println(" "+ params[i]);
    }
}
```

Output:

```
c1 = a
path=/
domain=localhost
c2 = b
path=/
c3 = c
domain=localhost
```
HTTP Entities

• HTTP messages can carry a **content entity** associated with the request or response

• The HTTP specification defines two entity-enclosing methods: POST and PUT

• Responses are usually expected to enclose a content entity
  – Exceptions: responses to HEAD method and 204 (No Content), 304 (Not Modified), etc.
HTTP Entities (cont.)

• When creating an entity for a outgoing message, this meta data has to be supplied by the creator of the entity

```java
StringEntity myEntity = new StringEntity("important message", Consts.UTF_8);
System.out.println(myEntity.getContentType());
System.out.println(myEntity.getLength());
System.out.println(EntityUtils.toString(myEntity));
System.out.println(EntityUtils.toByteArray(myEntity).length);
```

Output:

```
Content-Type: text/plain; charset=UTF-8
17
important message
17
```
HTTP Entities (cont.)

- **streamed**: The content is received from a stream, or generated on the fly
  - Streamed entities are generally not repeatable (i.e., cannot be read more than once)

- **self-contained**: The content is in memory or obtained by means that are independent from a connection or other entity
  - Self-contained entities are generally repeatable

- **wrapping**: The content is obtained from another entity
To read the content from the entity, one can either retrieve the input stream via the `HttpEntity#getContent()` method.

The methods `HttpEntity#getContentType()` and `HttpEntity#getContentLength()` methods can be used for reading the common metadata such as Content-Type and Content-Length headers (if they are available).
HttpResponse response = ...;

HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    try {
        // do something useful
    } finally {
        instream.close();
    }
}
Types of Entities

- **BasicHttpEntity** – A generic streamed, non-repeatable entity that obtains its content from an InputStream
  - In general, use this class for entities received from HTTP messages
    ```
    BasicHttpEntity myEntity = new BasicHttpEntity();
    myEntity.setContent(...);
    myEntity.setContentLength(340); // sets the length to 340
    ```

- **ByteArrayEntity** - A self contained, repeatable entity that obtains its content from a byte array
  ```
  ByteArrayEntity myEntity = new ByteArrayEntity(new byte[]{1,2,3},
           ContentType.APPLICATION_OCTET_STREAM);
  ```
Types of Entities (cont.)

- **StringEntity** - A self contained, repeatable entity that obtains its content from `java.lang.String` object

```java
String message = "Hello there!";

// construct without a character encoding (defaults to ISO-8859-1)
HttpEntity myEntity1 = new StringEntity(message);

// alternatively construct with an encoding (mime type defaults to "text/plain")
HttpEntity myEntity2 = new StringEntity(message, Consts.UTF_8);

// alternatively construct with an encoding and a mime type
HttpEntity myEntity3 = new StringEntity(message, ContentType.create("text/plain", Consts.UTF_8));
```
Types of Entities (cont.)

- **FileEntity** - a self contained, repeatable entity that obtains its content from a file
  - Used for streaming large files of different types, where you need to supply the content type of the file, e.g.,
    - sending a zip file would require the content type `application/zip`
    - for XML the content type is `application/xml`

```java
HttpEntity entity = new FileEntity(file,
                                      ContentType.create("application/java-archive"));
```
Entities – Final Note

• It is very important to always close the underlying content stream in order to signal that the processing of the message is complete

• HTTP entities that stream out their content directly from the input stream of the underlying connection must ensure they fully consume the content of the message body for that connection to be potentially re-usable
Overview

1. Introduction
2. Fundamentals
3. Interceptors, Processors
4. Connections
5. Handlers
Interceptors

- HTTP protocol **interceptor** is a routine that implements a specific aspect of the HTTP protocol
- Interceptors can:
  - act upon one specific header or a group of related headers of the incoming message
  - populate the outgoing message with one specific header or a group of related headers
  - manipulate content entities enclosed with messages (e.g., transparent content compression/decompression)
HttpCore comes with a number of most essential protocol interceptors for client and server HTTP processing

**RequestContent / ResponseContent**
- It is responsible for delimiting content length by adding the Content-Length or Transfer-Content headers based on the properties of the enclosed entity and the protocol version

**RequestTargetHost**
- responsible for adding the Host header

**RequestUserAgent**
- responsible for adding the User-Agent header
Standard Interceptors

- **RequestConnControl (ResponseConnControl)**
  - responsible for adding the Connection header to the outgoing requests (responses)

```java
public class RequestConnControl implements HttpRequestInterceptor {
  :
  @Override
  public void process(final HttpRequest request, final HttpContext context)
      throws HttpException, IOException {
      :
      
      if (!request.containsHeader(HTTP.CONN_DIRECTIVE)) {
        // Default policy is to keep connection alive
        // whenever possible
        request.setHeader(HTTP.CONN_DIRECTIVE, HTTP.CONN_KEEP_ALIVE);
      }
  }
}
```
• HTTP protocol processor is a collection of protocol interceptors that implements the 'Chain of Responsibility' pattern
  – I.e., each interceptor is expected to work on the particular aspect of the HTTP protocol it is responsible for
  – Interceptors are executed in the same order they were added to the processor
  – Usually HTTP protocol processors are used to pre-process messages **prior** to executing application specific processing logic

HttpProcessor httpproc = HttpProcessorBuilder.create()
   .add(new RequestContent())
   .add(new RequestTargetHost())
   .add(new RequestConnControl())
   .add(new RequestUserAgent("MyAgent-HTTP/1.1"))
   .add(new RequestExpectContinue(true))
   .build();

HttpContext context = HttpContext.create();
HttpRequest request = new BasicHttpRequest("GET", "/");
httpproc.process(request, context);
Execution Context

• Originally HTTP has been designed as a stateless, response-request oriented protocol

• However, real world applications often need to be able to persist state information through several logically related request-response exchanges
  – E.g., sessions

• HttpCore allows HTTP messages to be executed within a particular execution context (referred to as HTTP context)
  – It is simply a collection of logically related named values
HttpGetProcessor `httpproc` = HttpProcessorBuilder.create()
  .add(new HttpRequestInterceptor() {
      public void process(HttpRequest request, HttpContext context)
        throws HttpException, IOException {
          String id = (String) context.getAttribute("session-id");
          if (id != null) {
            request.addHeader("Session-ID", id);
          }
        }
    })
  .build();

HttpCoreContext `context` = HttpCoreContext.create();
HttpRequest `request` = new BasicHttpRequest("GET", "/");
httpproc.process(request, context);
Overview

1. Introduction
2. Fundamentals
3. Interceptors, Processors
4. Connections
5. Handlers
(Blocking) HTTP Connections

- Blocking (or classic) I/O in Java represents a convenient I/O model well suited for applications where the number of concurrent connections is relatively moderate
  - Modern JVMs are capable of efficient context switching
  - However, for applications where connections stay idle most of the time the overhead of context switching may become substantial and a **non-blocking** I/O model may present a better alternative

- Objects of the interface `HttpConnection` are responsible for HTTP message serialization and deserialization
  - The `DefaultBHttpClientConnection` class is its default implantation
(Blocking) HTTP Connections

• HTTP connection interfaces (client and server) send and receive messages in two stages
  – The message head is transmitted first
  – Depending on properties of the message head, a message body may follow it
Socket socket = new Socket("www.w3.org", 80);
DefaultBHttpClientConnection conn =
   new DefaultBHttpClientConnection(8 * 1024);
conn.bind(socket);

HttpRequest request = new BasicHttpRequest("GET", "/");
request.addHeader("Host", "www.w3.org");
conn.sendRequestHeader(request);
conn.flush();

HttpResponse response = conn.receiveResponseHeader();
conn.receiveResponseEntity(response);
HttpEntity entity = response.getEntity();

if (entity != null) {
    // Do something useful with the entity and, when done, ensure all
    // content has been consumed, so that the underlying connection
    // can be re-used
    EntityUtils.consume(entity);
}

Example
Overview

1. Introduction
2. Fundamentals
3. Interceptors, Processors
4. Connections
5. Handlers

http://www
• **HttpService** is a **server** side HTTP protocol handler based on the blocking I/O model
  – implements the essential requirements of the HTTP protocol for the server side message processing

```java
HttpProcessor httpproc = HttpProcessorBuilder.create()
    .add(new ResponseDate())
    .add(new ResponseServer("MyServer-HTTP/1.1"))
    .add(new ResponseContent())
    .add(new ResponseConnControl())
    .build();

HttpService httpService = new HttpService(httpproc, null);
```
• The HttpRequestHandler interface represents a routine for processing of a specific group of HTTP requests

• HttpService is designed to take care of protocol specific aspects, whereas individual request handlers are expected to take care of application specific HTTP processing

• The main purpose of a request handler is to generate a response object with a content entity to be sent back to the client in response to the given request
class MyRequestHandler implements HttpRequestHandler {

    public void handle(HttpRequest request,
                        HttpResponse response, HttpContext context)
        throws HttpException, IOException {
        response.setStatusCode(HttpStatus.SC_OK);
        response.setEntity(
            new StringEntity("some important message",
                            ContentType.TEXT_PLAIN));
    }
}
Mapping URIs to Request Handlers

- HTTP request handlers are usually managed by a `UriHttpRequestHandlerMapper` that matches a request URI to a request handler.
- HttpCore includes a very simple implementation of the request handler resolver based on a trivial pattern matching algorithm:
  - supports only three formats: `*`, `<uri>*` and `*<uri>`
Mapping URIs to Request Handlers

HttpProcessor httpproc = <...>
HttpRequestHandler myRequestHandler1 = <...>
HttpRequestHandler myRequestHandler2 = <...>
HttpRequestHandler myRequestHandler3 = <...>

UriHttpRequestHandlerMapper handlerMapper =
    new UriHttpRequestHandlerMapper();
handlerMapper.register("/service/*", myRequestHandler1);
handlerMapper.register("*.do", myRequestHandler2);
handlerMapper.register("*", myRequestHandler3);

HttpService httpService = new HttpService(httpproc, handlerMapper);
Request Executors

- **HttpRequestExecutor** is a client side HTTP protocol handler based on the blocking I/O model
  - implements the essential requirements of the HTTP protocol for the client side message processing

- The **HttpRequestExecutor** relies on the **HttpProcessor** instance to generate mandatory protocol headers for all outgoing messages and apply common message transformations to all incoming and outgoing messages
Request Executors

```java
HttpClientConnection conn = <...>
HttpProcessor httpproc = HttpProcessorBuilder.create()
    .add(new RequestContent())
    .add(new RequestTargetHost())
    .add(new RequestConnControl())
    .add(new RequestUserAgent("MyClient/1.1"))
    .add(new RequestExpectContinue(true))
    .build();
HttpRequestExecutor httpexecutor =
    new HttpRequestExecutor();

HttpRequest request = new BasicHttpRequest("GET", "/");
HttpCoreContext context = HttpCoreContext.create();
httpexecutor.preProcess(request, httpproc, context);
HttpResponse response = httpexecutor.execute(request, conn, context);
httpexecutor.postProcess(response, httpproc, context);
HttpEntity entity = response.getEntity();
EntityUtils.consume(entity);
```