# HttpClient Tutorial

**Oleg Kalnichevski**

Preface ..... iv
1. HttpClient scope ..... iv
2. What HttpClient is NOT ..... iv
1. Fundamentals ..... 1
1.1. Request execution ..... 1
1.1.1. HTTP request ..... 1
1.1.2. HTTP response ..... 2
1.1.3. Working with message headers ..... 2
1.1.4. HTTP entity ..... 4
1.1.5. Ensuring release of low level resources ..... 5
1.1.6. Consuming entity content ..... 6
1.1.7. Producing entity content ..... 6
1.1.8. Response handlers ..... 8
1.2. HTTP execution context ..... 8
1.3. Exception handling ..... 9
1.3.1. HTTP transport safety ..... 9
1.3.2. Idempotent methods ..... 10
1.3.3. Automatic exception recovery ..... 10
1.3.4. Request retry handler ..... 10
1.4. Aborting requests ..... 11
1.5. HTTP protocol interceptors ..... 11
1.6. HTTP parameters ..... 12
1.6.1. Parameter hierarchies ..... 12
1.6.2. HTTP parameters beans ..... 13
1.7. HTTP request execution parameters ..... 14
2. Connection management ..... 15
2.1. Connection parameters ..... 15
2.2. Connection persistence ..... 16
2.3. HTTP connection routing ..... 16
2.3.1. Route computation ..... 16
2.3.2. Secure HTTP connections ..... 17
2.4. HTTP route parameters ..... 17
2.5. Socket factories ..... 17
2.5.1. Secure socket layering ..... 18
2.5.2. SSL/TLS customization ..... 18
2.5.3. Hostname verification ..... 19
2.6. Protocol schemes ..... 19
2.7. HttpClient proxy configuration ..... 19
2.8. HTTP connection managers ..... 20
2.8.1. Connection operators ..... 20
2.8.2. Managed connections and connection managers ..... 20
2.8.3. Simple connection manager ..... 22
2.8.4. Pooling connection manager ..... 22
2.8.5. Connection manager shutdown ..... 23
2.9. Multithreaded request execution ..... 23
2.10. Connection eviction policy ..... 24
2.11. Connection keep alive strategy ..... 25
3. HTTP state management ..... 27
3.1. HTTP cookies ..... 27

3.1.1. Cookie versions ... 27
3.2. Cookie specifications ... 28
3.3. HTTP cookie and state management parameters ... 29
3.4. Cookie specification registry ... 29
3.5. Choosing cookie policy ... 29
3.6. Custom cookie policy ... 30
3.7. Cookie persistence ... 30
3.8. HTTP state management and execution context ... 30
3.9. Per user / thread state management ... 31
4. HTTP authentication ... 32
4.1. User credentials ... 32
4.2. Authentication schemes ... 32
4.3. HTTP authentication parameters ... 33
4.4. Authentication scheme registry ... 34
4.5. Credentials provider ... 34
4.6. HTTP authentication and execution context ... 35
4.7. Caching of authentication data ... 36
4.8. Preemptive authentication ... 36
4.9. NTLM Authentication ... 36
4.9.1. NTLM connection persistence ... 36
4.10. `SPNEGO`/Kerberos Authentication ... 37
4.10.1. `SPNEGO` support in HttpClient ... 38
4.10.2. GSS/Java Kerberos Setup ... 38
4.10.3. `login.conf` file ... 38
4.10.4. `krb5.conf` / `krb5.ini` file ... 39
4.10.5. Windows Specific configuration ... 39
4.10.6. Customizing `SPNEGO` authentication scheme ... 40
5. HTTP client service ... 41
5.1. HttpClient facade ... 41
5.2. HttpClient parameters ... 42
5.3. Automatic redirect handling ... 43
5.4. HTTP client and execution context ... 43
5.5. Compressed response content ... 43
6. HTTP Caching ... 45
6.1. General Concepts ... 45
6.2. RFC-2616 Compliance ... 46
6.3. Example Usage ... 46
6.4. Configuration ... 46
6.5. Storage Backends ... 47
7. Advanced topics ... 48
7.1. Custom client connections ... 48
7.2. Stateful HTTP connections ... 49
7.2.1. User token handler ... 49
7.2.2. User token and execution context ... 50

# Preface

The Hyper-Text Transfer Protocol (HTTP) is perhaps the most significant protocol used on the Internet today. Web services, network-enabled appliances and the growth of network computing continue to expand the role of the HTTP protocol beyond user-driven web browsers, while increasing the number of applications that require HTTP support.

Although the java.net package provides basic functionality for accessing resources via HTTP, it doesn't provide the full flexibility or functionality needed by many applications. HttpClient seeks to fill this void by providing an efficient, up-to-date, and feature-rich package implementing the client side of the most recent HTTP standards and recommendations.

Designed for extension while providing robust support for the base HTTP protocol, HttpClient may be of interest to anyone building HTTP-aware client applications such as web browsers, web service clients, or systems that leverage or extend the HTTP protocol for distributed communication.

## 1. HttpClient scope

- Client-side HTTP transport library based on HttpCore [http://hc.apache.org/httpcomponents-core/index.html]
- Based on classic (blocking) I/O
- Content agnostic

## 2. What HttpClient is NOT

- HttpClient is NOT a browser. It is a client side HTTP transport library. HttpClient's purpose is to transmit and receive HTTP messages. HttpClient will not attempt to cache content, execute javascript embedded in HTML pages, try to guess content type, or reformat request / redirect location URIs, or other functionality unrelated to the HTTP transport.

# Chapter 1. Fundamentals

## 1.1. Request execution

The most essential function of HttpClient is to execute HTTP methods. Execution of an HTTP method involves one or several HTTP request / HTTP response exchanges, usually handled internally by HttpClient. The user is expected to provide a request object to execute and HttpClient is expected to transmit the request to the target server return a corresponding response object, or throw an exception if execution was unsuccessful.

Quite naturally, the main entry point of the HttpClient API is the HttpClient interface that defines the contract described above.

Here is an example of request execution process in its simplest form:

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    int l;
    byte[] tmp = new byte[2048];
    while ((l = instream.read(tmp)) != -1) {
    }
}
```

### 1.1.1. HTTP request

All HTTP requests have a request line consisting a method name, a request URI and an HTTP protocol version.

HttpClient supports out of the box all HTTP methods defined in the HTTP/1.1 specification: `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE` and `OPTIONS`. There is a specific class for each method type.: `HttpGet`, `HttpHead`, `HttpPost`, `HttpPut`, `HttpDelete`, `HttpTrace`, and `HttpOptions`.

The Request-URI is a Uniform Resource Identifier that identifies the resource upon which to apply the request. HTTP request URIs consist of a protocol scheme, host name, optional port, resource path, optional query, and optional fragment.

```
HttpGet httpget = new HttpGet(
     "http://www.google.com/search?hl=en&q=httpclient&btnG=Google+Search&aq=f&oq=");
```

HttpClient provides a number of utility methods to simplify creation and modification of request URIs.

URI can be assembled programmatically:

```
URI uri = URIUtils.createURI("http", "www.google.com", -1, "/search",
    "q=httpclient&btnG=Google+Search&aq=f&oq=", null);
HttpGet httpget = new HttpGet(uri);
System.out.println(httpget.getURI());
```

stdout >

```
http://www.google.com/search?q=httpclient&btnG=Google+Search&aq=f&oq=
```

Query string can also be generated from individual parameters:

```
List<NameValuePair> qparams = new ArrayList<NameValuePair>();
qparams.add(new BasicNameValuePair("q", "httpclient"));
qparams.add(new BasicNameValuePair("btnG", "Google Search"));
qparams.add(new BasicNameValuePair("aq", "f"));
qparams.add(new BasicNameValuePair("oq", null));
URI uri = URIUtils.createURI("http", "www.google.com", -1, "/search",
    URLEncodedUtils.format(qparams, "UTF-8"), null);
HttpGet httpget = new HttpGet(uri);
System.out.println(httpget.getURI());
```

stdout >

```
http://www.google.com/search?q=httpclient&btnG=Google+Search&aq=f&oq=
```

## 1.1.2. HTTP response

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());
```

stdout >

```
HTTP/1.1
200
OK
HTTP/1.1 200 OK
```

## 1.1.3. Working with message headers

An HTTP message can contain a number of headers describing properties of the message such as the content length, content type and so on. HttpClient provides methods to retrieve, add, remove and enumerate headers.

```
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);
```

stdout >

```

Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
2
```

The most efficient way to obtain all headers of a given type is by using the `HeaderIterator` interface.

```

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());
}
```

stdout >

```

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.

```

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\"");

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]);
    }
}
```

stdout >

```

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

```
domain=localhost
```

## 1.1.4. HTTP entity

HTTP messages can carry a content entity associated with the request or response. Entities can be found in some requests and in some responses, as they are optional. Requests that use entities are referred to as entity enclosing requests. The HTTP specification defines two entity enclosing request methods: `POST` and `PUT`. Responses are usually expected to enclose a content entity. There are exceptions to this rule such as responses to `HEAD` method and `204 No Content`, `304 Not Modified`, `205 Reset Content` responses.

HttpClient distinguishes three kinds of entities, depending on where their content originates:

- **streamed:** The content is received from a stream, or generated on the fly. In particular, this category includes entities being received from HTTP responses. Streamed entities are generally not repeatable.
- **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. This type of entities will be mostly used for entity enclosing HTTP requests.
- **wrapping:** The content is obtained from another entity.

This distinction is important for connection management when streaming out content from an HTTP response. For request entities that are created by an application and only sent using HttpClient, the difference between streamed and self-contained is of little importance. In that case, it is suggested to consider non-repeatable entities as streamed, and those that are repeatable as self-contained.

### 1.1.4.1. Repeatable entities

An entity can be repeatable, meaning its content can be read more than once. This is only possible with self contained entities (like `ByteArrayEntity` or `StringEntity`)

### 1.1.4.2. Using HTTP entities

Since an entity can represent both binary and character content, it has support for character encodings (to support the latter, ie. character content).

The entity is created when executing a request with enclosed content or when the request was successful and the response body is used to send the result back to the client.

To read the content from the entity, one can either retrieve the input stream via the `HttpEntity#getContent()` method, which returns an `java.io.InputStream`, or one can supply an output stream to the `HttpEntity#writeTo(OutputStream)` method, which will return once all content has been written to the given stream.

When the entity has been received with an incoming message, 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). Since the `Content-Type` header can contain a character encoding for text mime-types like text/plain or text/html, the `HttpEntity#getContentEncoding()` method is used to read this information. If the headers aren't available, a length of -1 will be returned, and NULL for the content type. If the `Content-Type` header is available, a `Header` object will be returned.

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

```
StringEntity myEntity = new StringEntity("important message",
    "UTF-8");

System.out.println(myEntity.getContentType());
System.out.println(myEntity.getContentLength());
System.out.println(EntityUtils.getContentCharSet(myEntity));
System.out.println(EntityUtils.toString(myEntity));
System.out.println(EntityUtils.toByteArray(myEntity).length);
```

stdout >

```
Content-Type: text/plain; charset=UTF-8
17
UTF-8
important message
17
```

### 1.1.5. Ensuring release of low level resources

In order to ensure proper release of system resources one must close the content stream associated with the entity.

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

Please note that the `HttpEntity#writeTo(OutputStream)` method is also required to ensure proper release of system resources once the entity has been fully written out. If this method obtains an instance of `java.io.InputStream` by calling `HttpEntity#getContent()`, it is also expected to close the stream in a finally clause.

When working with streaming entities, one can use the `EntityUtils#consume(HttpEntity)` method to ensure that the entity content has been fully consumed and the underlying stream has been closed.

There can be situations, however, when only a small portion of the entire response content needs to be retrieved and the performance penalty for consuming the remaining content and making the connection reusable is too high, in which case one can simply terminate the request by calling `HttpUriRequest#abort()` method.

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    int byteOne = instream.read();
    int byteTwo = instream.read();
```

```
    // Do not need the rest
    httpget.abort();
}
```

The connection will not be reused, but all level resources held by it will be correctly deallocated.

### 1.1.6. Consuming entity content

The recommended way to consume the content of an entity is by using its `HttpEntity#getContent()` or `HttpEntity#writeTo(OutputStream)` methods. HttpClient also comes with the `EntityUtils` class, which exposes several static methods to more easily read the content or information from an entity. Instead of reading the `java.io.InputStream` directly, one can retrieve the whole content body in a string / byte array by using the methods from this class. However, the use of `EntityUtils` is strongly discouraged unless the response entities originate from a trusted HTTP server and are known to be of limited length.

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    long len = entity.getContentLength();
    if (len != -1 && len < 2048) {
        System.out.println(EntityUtils.toString(entity));
    } else {
        // Stream content out
    }
}
```

In some situations it may be necessary to be able to read entity content more than once. In this case entity content must be buffered in some way, either in memory or on disk. The simplest way to accomplish that is by wrapping the original entity with the `BufferedHttpEntity` class. This will cause the content of the original entity to be read into a in-memory buffer. In all other ways the entity wrapper will be have the original one.

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    entity = new BufferedHttpEntity(entity);
}
```

### 1.1.7. Producing entity content

HttpClient provides several classes that can be used to efficiently stream out content though HTTP connections. Instances of those classes can be associated with entity enclosing requests such as `POST` and `PUT` in order to enclose entity content into outgoing HTTP requests. HttpClient provides several classes for most common data containers such as string, byte array, input stream, and file: `StringEntity`, `ByteArrayEntity`, `InputStreamEntity`, and `FileEntity`.

```
File file = new File("somefile.txt");
FileEntity entity = new FileEntity(file, "text/plain; charset=\"UTF-8\"");

HttpPost httppost = new HttpPost("http://localhost/action.do");
httppost.setEntity(entity);
```

Please note `InputStreamEntity` is not repeatable, because it can only read from the underlying data stream once. Generally it is recommended to implement a custom `HttpEntity` class which is self-contained instead of using the generic `InputStreamEntity`. `FileEntity` can be a good starting point.

#### 1.1.7.1. Dynamic content entities

Often HTTP entities need to be generated dynamically based a particular execution context. HttpClient provides support for dynamic entities by using the `EntityTemplate` entity class and `ContentProducer` interface. Content producers are objects which produce their content on demand, by writing it out to an output stream. They are expected to be able produce their content every time they are requested to do so. So entities created with `EntityTemplate` are generally self-contained and repeatable.

```
ContentProducer cp = new ContentProducer() {
    public void writeTo(OutputStream outstream) throws IOException {
        Writer writer = new OutputStreamWriter(outstream, "UTF-8");
        writer.write("<response>");
        writer.write("  <content>");
        writer.write("    important stuff");
        writer.write("  </content>");
        writer.write("</response>");
        writer.flush();
    }
};
HttpEntity entity = new EntityTemplate(cp);
HttpPost httppost = new HttpPost("http://localhost/handler.do");
httppost.setEntity(entity);
```

#### 1.1.7.2. HTML forms

Many applications need to simulate the process of submitting an HTML form, for instance, in order to log in to a web application or submit input data. HttpClient provides the entity class `UrlEncodedFormEntity` to facilitate the process.

```
List<NameValuePair> formparams = new ArrayList<NameValuePair>();
formparams.add(new BasicNameValuePair("param1", "value1"));
formparams.add(new BasicNameValuePair("param2", "value2"));
UrlEncodedFormEntity entity = new UrlEncodedFormEntity(formparams, "UTF-8");
HttpPost httppost = new HttpPost("http://localhost/handler.do");
httppost.setEntity(entity);
```

The `UrlEncodedFormEntity` instance will use the so called URL encoding to encode parameters and produce the following content:

```
param1=value1&param2=value2
```

#### 1.1.7.3. Content chunking

Generally it is recommended to let HttpClient choose the most appropriate transfer encoding based on the properties of the HTTP message being transferred. It is possible, however, to inform HttpClient that chunk coding is preferred by setting `HttpEntity#setChunked()` to true. Please note that HttpClient will use this flag as a hint only. This value will be ignored when using HTTP protocol versions that do not support chunk coding, such as HTTP/1.0.

```
StringEntity entity = new StringEntity("important message",
    "text/plain; charset=\"UTF-8\"");
```

```
entity.setChunked(true);
HttpPost httppost = new HttpPost("http://localhost/acrtion.do");
httppost.setEntity(entity);
```

### 1.1.8. Response handlers

The simplest and the most convenient way to handle responses is by using the `ResponseHandler` interface, which includes the `handleResponse(HttpResponse response)` method. This method completely relieves the user from having to worry about connection management. When using a `ResponseHandler`, HttpClient will automatically take care of ensuring release of the connection back to the connection manager regardless whether the request execution succeeds or causes an exception.

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://localhost/");

ResponseHandler<byte[]> handler = new ResponseHandler<byte[]>() {
    public byte[] handleResponse(
            HttpResponse response) throws ClientProtocolException, IOException {
        HttpEntity entity = response.getEntity();
        if (entity != null) {
            return EntityUtils.toByteArray(entity);
        } else {
            return null;
        }
    }
};

byte[] response = httpclient.execute(httpget, handler);
```

## 1.2. HTTP 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. In order to enable applications to maintain a processing state HttpClient allows HTTP requests to be executed within a particular execution context, referred to as HTTP context. Multiple logically related requests can participate in a logical session if the same context is reused between consecutive requests. HTTP context functions similarly to a `java.util.Map<String, Object>`. It is simply a collection of arbitrary named values. An application can populate context attributes prior to request execution or examine the context after the execution has been completed.

In the course of HTTP request execution HttpClient adds the following attributes to the execution context:

- **`ExecutionContext.HTTP_CONNECTION`='http.connection':** `HttpConnection` instance representing the actual connection to the target server.
- **`ExecutionContext.HTTP_TARGET_HOST`='http.target_host':** `HttpHost` instance representing the connection target.
- **`ExecutionContext.HTTP_PROXY_HOST`='http.proxy_host':** `HttpHost` instance representing the connection proxy, if used
- **`ExecutionContext.HTTP_REQUEST`='http.request':** `HttpRequest` instance representing the actual HTTP request. The final HttpRequest object in the execution context always represents the state of the message _exactly_ as it was sent to the target server. Per default HTTP/1.0 and HTTP/1.1

use relative request URIs. However if the request is sent via a proxy in a non-tunneling mode then the URI will be absolute.

- **`ExecutionContext.HTTP_RESPONSE`='http.response':** `HttpResponse` instance representing the actual HTTP response.
- **`ExecutionContext.HTTP_REQ_SENT`='http.request_sent':** `java.lang.Boolean` object representing the flag indicating whether the actual request has been fully transmitted to the connection target.

For instance, in order to determine the final redirect target, one can examine the value of the `http.target_host` attribute after the request execution:

```
DefaultHttpClient httpclient = new DefaultHttpClient();

HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://www.google.com/");

HttpResponse response = httpclient.execute(httpget, localContext);

HttpHost target = (HttpHost) localContext.getAttribute(
    ExecutionContext.HTTP_TARGET_HOST);

System.out.println("Final target: " + target);

HttpEntity entity = response.getEntity();
EntityUtils.consume(entity);
}
```

stdout >

```
Final target: http://www.google.ch
```

# 1.3. Exception handling

HttpClient can throw two types of exceptions: `java.io.IOException` in case of an I/O failure such as socket timeout or an socket reset and `HttpException` that signals an HTTP failure such as a violation of the HTTP protocol. Usually I/O errors are considered non-fatal and recoverable, whereas HTTP protocol errors are considered fatal and cannot be automatically recovered from.

## 1.3.1. HTTP transport safety

It is important to understand that the HTTP protocol is not well suited to all types of applications. HTTP is a simple request/response oriented protocol which was initially designed to support static or dynamically generated content retrieval. It has never been intended to support transactional operations. For instance, the HTTP server will consider its part of the contract fulfilled if it succeeds in receiving and processing the request, generating a response and sending a status code back to the client. The server will make no attempt to roll back the transaction if the client fails to receive the response in its entirety due to a read timeout, a request cancellation or a system crash. If the client decides to retry the same request, the server will inevitably end up executing the same transaction more than once. In some cases this may lead to application data corruption or inconsistent application state.

Even though HTTP has never been designed to support transactional processing, it can still be used as a transport protocol for mission critical applications provided certain conditions are met. To ensure

HTTP transport layer safety the system must ensure the idempotency of HTTP methods on the application layer.

### 1.3.2. Idempotent methods

HTTP/1.1 specification defines an idempotent method as

[Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request]

In other words the application ought to ensure that it is prepared to deal with the implications of multiple execution of the same method. This can be achieved, for instance, by providing a unique transaction id and by other means of avoiding execution of the same logical operation.

Please note that this problem is not specific to HttpClient. Browser based applications are subject to exactly the same issues related to HTTP methods non-idempotency.

HttpClient assumes non-entity enclosing methods such as `GET` and `HEAD` to be idempotent and entity enclosing methods such as `POST` and `PUT` to be not.

### 1.3.3. Automatic exception recovery

By default HttpClient attempts to automatically recover from I/O exceptions. The default auto-recovery mechanism is limited to just a few exceptions that are known to be safe.

- HttpClient will make no attempt to recover from any logical or HTTP protocol errors (those derived from `HttpException` class).
- HttpClient will automatically retry those methods that are assumed to be idempotent.
- HttpClient will automatically retry those methods that fail with a transport exception while the HTTP request is still being transmitted to the target server (i.e. the request has not been fully transmitted to the server).
- HttpClient will automatically retry those methods that have been fully transmitted to the server, but the server failed to respond with an HTTP status code (the server simply drops the connection without sending anything back). In this case it is assumed that the request has not been processed by the server and the application state has not changed. If this assumption may not hold true for the web server your application is targeting it is highly recommended to provide a custom exception handler.

### 1.3.4. Request retry handler

In order to enable a custom exception recovery mechanism one should provide an implementation of the `HttpRequestRetryHandler` interface.

```
DefaultHttpClient httpclient = new DefaultHttpClient();

HttpRequestRetryHandler myRetryHandler = new HttpRequestRetryHandler() {

    public boolean retryRequest(
            IOException exception,
            int executionCount,
            HttpContext context) {
        if (executionCount >= 5) {
```

```
            // Do not retry if over max retry count
            return false;
        }
        if (exception instanceof NoHttpResponseException) {
            // Retry if the server dropped connection on us
            return true;
        }
        if (exception instanceof SSLHandshakeException) {
            // Do not retry on SSL handshake exception
            return false;
        }
        HttpRequest request = (HttpRequest) context.getAttribute(
                ExecutionContext.HTTP_REQUEST);
        boolean idempotent = !(request instanceof HttpEntityEnclosingRequest);
        if (idempotent) {
            // Retry if the request is considered idempotent
            return true;
        }
        return false;
    }

};

httpclient.setHttpRequestRetryHandler(myRetryHandler);
```

## 1.4. Aborting requests

In some situations HTTP request execution fails to complete within the expected time frame due to high load on the target server or too many concurrent requests issued on the client side. In such cases it may be necessary to terminate the request prematurely and unblock the execution thread blocked in a I/O operation. HTTP requests being executed by HttpClient can be aborted at any stage of execution by invoking `HttpUriRequest#abort()` method. This method is thread-safe and can be called from any thread. When an HTTP request is aborted its execution thread - even if currently blocked in an I/O operation - is guaranteed to unblock by throwing a `InterruptedIOException`

## 1.5. HTTP protocol interceptors

Th HTTP protocol interceptor is a routine that implements a specific aspect of the HTTP protocol. Usually protocol interceptors are expected to act upon one specific header or a group of related headers of the incoming message, or populate the outgoing message with one specific header or a group of related headers. Protocol interceptors can also manipulate content entities enclosed with messages - transparent content compression / decompression being a good example. Usually this is accomplished by using the 'Decorator' pattern where a wrapper entity class is used to decorate the original entity. Several protocol interceptors can be combined to form one logical unit.

Protocol interceptors can collaborate by sharing information - such as a processing state - through the HTTP execution context. Protocol interceptors can use HTTP context to store a processing state for one request or several consecutive requests.

Usually the order in which interceptors are executed should not matter as long as they do not depend on a particular state of the execution context. If protocol interceptors have interdependencies and therefore must be executed in a particular order, they should be added to the protocol processor in the same sequence as their expected execution order.

Protocol interceptors must be implemented as thread-safe. Similarly to servlets, protocol interceptors should not use instance variables unless access to those variables is synchronized.

This is an example of how local context can be used to persist a processing state between consecutive requests:

```
DefaultHttpClient httpclient = new DefaultHttpClient();

HttpContext localContext = new BasicHttpContext();

AtomicInteger count = new AtomicInteger(1);

localContext.setAttribute("count", count);

httpclient.addRequestInterceptor(new HttpRequestInterceptor() {

    public void process(
            final HttpRequest request,
            final HttpContext context) throws HttpException, IOException {
        AtomicInteger count = (AtomicInteger) context.getAttribute("count");
        request.addHeader("Count", Integer.toString(count.getAndIncrement()));
    }

});

HttpGet httpget = new HttpGet("http://localhost/");
for (int i = 0; i < 10; i++) {
    HttpResponse response = httpclient.execute(httpget, localContext);

    HttpEntity entity = response.getEntity();
    EntityUtils.consume(entity);
}
```

## 1.6. HTTP parameters

The HttpParams interface represents a collection of immutable values that define a runtime behavior of a component. In many ways `HttpParams` is similar to `HttpContext`. The main distinction between the two lies in their use at runtime. Both interfaces represent a collection of objects that are organized as a map of keys to object values, but serve distinct purposes:

- `HttpParams` is intended to contain simple objects: integers, doubles, strings, collections and objects that remain immutable at runtime.
- `HttpParams` is expected to be used in the 'write once - ready many' mode. `HttpContext` is intended to contain complex objects that are very likely to mutate in the course of HTTP message processing.
- The purpose of `HttpParams` is to define a behavior of other components. Usually each complex component has its own `HttpParams` object. The purpose of `HttpContext` is to represent an execution state of an HTTP process. Usually the same execution context is shared among many collaborating objects.

### 1.6.1. Parameter hierarchies

In the course of HTTP request execution `HttpParams` of the `HttpRequest` object are linked together with `HttpParams` of the `HttpClient` instance used to execute the request. This enables parameters set at the HTTP request level to take precedence over `HttpParams` set at the HTTP client level. The recommended practice is to set common parameters shared by all HTTP requests at the HTTP client level and selectively override specific parameters at the HTTP request level.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.getParams().setParameter(CoreProtocolPNames.PROTOCOL_VERSION,
    HttpVersion.HTTP_1_0); // Default to HTTP 1.0
httpclient.getParams().setParameter(CoreProtocolPNames.HTTP_CONTENT_CHARSET,
    "UTF-8");

HttpGet httpget = new HttpGet("http://www.google.com/");
httpget.getParams().setParameter(CoreProtocolPNames.PROTOCOL_VERSION,
    HttpVersion.HTTP_1_1); // Use HTTP 1.1 for this request only
httpget.getParams().setParameter(CoreProtocolPNames.USE_EXPECT_CONTINUE,
    Boolean.FALSE);

httpclient.addRequestInterceptor(new HttpRequestInterceptor() {

    public void process(
            final HttpRequest request,
            final HttpContext context) throws HttpException, IOException {
        System.out.println(request.getParams().getParameter(
                CoreProtocolPNames.PROTOCOL_VERSION));
        System.out.println(request.getParams().getParameter(
                CoreProtocolPNames.HTTP_CONTENT_CHARSET));
        System.out.println(request.getParams().getParameter(
                CoreProtocolPNames.USE_EXPECT_CONTINUE));
        System.out.println(request.getParams().getParameter(
                CoreProtocolPNames.STRICT_TRANSFER_ENCODING));
    }

});
```

stdout >

```
HTTP/1.1
UTF-8
false
null
```

## 1.6.2. HTTP parameters beans

The `HttpParams` interface allows for a great deal of flexibility in handling configuration of components. Most importantly, new parameters can be introduced without affecting binary compatibility with older versions. However, `HttpParams` also has a certain disadvantage compared to regular Java beans: `HttpParams` cannot be assembled using a DI framework. To mitigate the limitation, HttpClient includes a number of bean classes that can used in order to initialize `HttpParams` objects using standard Java bean conventions.

```
HttpParams params = new BasicHttpParams();
HttpProtocolParamBean paramsBean = new HttpProtocolParamBean(params);
paramsBean.setVersion(HttpVersion.HTTP_1_1);
paramsBean.setContentCharset("UTF-8");
paramsBean.setUseExpectContinue(true);

System.out.println(params.getParameter(
    CoreProtocolPNames.PROTOCOL_VERSION));
System.out.println(params.getParameter(
    CoreProtocolPNames.HTTP_CONTENT_CHARSET));
System.out.println(params.getParameter(
    CoreProtocolPNames.USE_EXPECT_CONTINUE));
System.out.println(params.getParameter(
    CoreProtocolPNames.USER_AGENT));
```

stdout >

```
HTTP/1.1
UTF-8
false
null
```

## 1.7. HTTP request execution parameters

These are parameters that can impact the process of request execution:

- **`CoreProtocolPNames.PROTOCOL_VERSION`='http.protocol.version':** defines HTTP protocol version used if not set explicitly on the request object. This parameter expects a value of type `ProtocolVersion`. If this parameter is not set HTTP/1.1 will be used.

- **`CoreProtocolPNames.HTTP_ELEMENT_CHARSET`='http.protocol.element-charset':** defines the charset to be used for encoding HTTP protocol elements. This parameter expects a value of type `java.lang.String`. If this parameter is not set `US-ASCII` will be used.

- **`CoreProtocolPNames.HTTP_CONTENT_CHARSET`='http.protocol.content-charset':** defines the charset to be used per default for content body coding. This parameter expects a value of type `java.lang.String`. If this parameter is not set `ISO-8859-1` will be used.

- **`CoreProtocolPNames.USER_AGENT`='http.useragent':** defines the content of the `User-Agent` header. This parameter expects a value of type `java.lang.String`. If this parameter is not set, HttpClient will automatically generate a value for it.

- **`CoreProtocolPNames.STRICT_TRANSFER_ENCODING`='http.protocol.strict-transfer-encoding':** defines whether responses with an invalid `Transfer-Encoding` header should be rejected. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set, invalid `Transfer-Encoding` values will be ignored.

- **`CoreProtocolPNames.USE_EXPECT_CONTINUE`='http.protocol.expect-continue':** activates the `Expect: 100-Continue` handshake for the entity enclosing methods. The purpose of the `Expect: 100-Continue` handshake is to allow the client that is sending a request message with a request body to determine if the origin server is willing to accept the request (based on the request headers) before the client sends the request body. The use of the `Expect: 100-continue` handshake can result in a noticeable performance improvement for entity enclosing requests (such as `POST` and `PUT`) that require the target server's authentication. The `Expect: 100-continue` handshake should be used with caution, as it may cause problems with HTTP servers and proxies that do not support HTTP/1.1 protocol. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set, HttpClient will not attempt to use the handshake.

- **`CoreProtocolPNames.WAIT_FOR_CONTINUE`='http.protocol.wait-for-continue':** defines the maximum period of time in milliseconds the client should spend waiting for a `100-continue` response. This parameter expects a value of type `java.lang.Integer`. If this parameter is not set HttpClient will wait 3 seconds for a confirmation before resuming the transmission of the request body.

# Chapter 2. Connection management

HttpClient assumes complete control over the process of connection initialization and termination as well as I/O operations on active connections. However various aspects of connection operations can be influenced using a number of parameters.

## 2.1. Connection parameters

These are parameters that can influence connection operations:

- **`CoreConnectionPNames.SO_TIMEOUT`='http.socket.timeout':** defines the socket timeout (`SO_TIMEOUT`) in milliseconds, which is the timeout for waiting for data or, put differently, a maximum period inactivity between two consecutive data packets). A timeout value of zero is interpreted as an infinite timeout. This parameter expects a value of type `java.lang.Integer`. If this parameter is not set, read operations will not time out (infinite timeout).

- **`CoreConnectionPNames.TCP_NODELAY`='http.tcp.nodelay':** determines whether Nagle's algorithm is to be used. Nagle's algorithm tries to conserve bandwidth by minimizing the number of segments that are sent. When applications wish to decrease network latency and increase performance, they can disable Nagle's algorithm (that is enable `TCP_NODELAY`. Data will be sent earlier, at the cost of an increase in bandwidth consumption. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set, `TCP_NODELAY` will be enabled (no delay).

- **`CoreConnectionPNames.SOCKET_BUFFER_SIZE`='http.socket.buffer-size':** determines the size of the internal socket buffer used to buffer data while receiving / transmitting HTTP messages. This parameter expects a value of type `java.lang.Integer`. If this parameter is not set, HttpClient will allocate 8192 byte socket buffers.

- **`CoreConnectionPNames.SO_LINGER`='http.socket.linger':** sets `SO_LINGER` with the specified linger time in seconds. The maximum timeout value is platform specific. Value 0 implies that the option is disabled. Value -1 implies that the JRE default is used. The setting only affects the socket close operation. If this parameter is not set, the value -1 (JRE default) will be assumed.

- **`CoreConnectionPNames.CONNECTION_TIMEOUT`='http.connection.timeout':** determines the timeout in milliseconds until a connection is established. A timeout value of zero is interpreted as an infinite timeout. This parameter expects a value of type `java.lang.Integer`. If this parameter is not set, connect operations will not time out (infinite timeout).

- **`CoreConnectionPNames.STALE_CONNECTION_CHECK`='http.connection.stalecheck':** determines whether stale connection check is to be used. Disabling stale connection check may result in a noticeable performance improvement (the check can cause up to 30 millisecond overhead per request) at the risk of getting an I/O error when executing a request over a connection that has been closed at the server side. This parameter expects a value of type `java.lang.Boolean`. For performance critical operations the check should be disabled. If this parameter is not set, the stale connection check will be performed before each request execution.

- **`CoreConnectionPNames.MAX_LINE_LENGTH`='http.connection.max-line-length':** determines the maximum line length limit. If set to a positive value, any HTTP line exceeding this limit will cause an `java.io.IOException`. A negative or zero value will effectively disable the check. This parameter expects a value of type `java.lang.Integer`. If this parameter is not set, no limit will be enforced.

- **`CoreConnectionPNames.MAX_HEADER_COUNT`='http.connection.max-header-count':** determines the maximum HTTP header count allowed. If set to a positive value, the number of HTTP headers received from the data stream exceeding this limit will cause an `java.io.IOException`. A negative or zero value will effectively disable the check. This parameter expects a value of type `java.lang.Integer`. If this parameter is not set, no limit will be enforced.

- **`ConnConnectionPNames.MAX_STATUS_LINE_GARBAGE`='http.connection.max-status-line-garbage':** defines the maximum number of ignorable lines before we expect a HTTP response's status line. With HTTP/1.1 persistent connections, the problem arises that broken scripts could return a wrong `Content-Length` (there are more bytes sent than specified). Unfortunately, in some cases, this cannot be detected after the bad response, but only before the next one. So HttpClient must be able to skip those surplus lines this way. This parameter expects a value of type java.lang.Integer. 0 disallows all garbage/empty lines before the status line. Use `java.lang.Integer#MAX_VALUE` for unlimited number. If this parameter is not set, unlimited number will be assumed.

## 2.2. Connection persistence

The process of establishing a connection from one host to another is quite complex and involves multiple packet exchanges between two endpoints, which can be quite time consuming. The overhead of connection handshaking can be significant, especially for small HTTP messages. One can achieve a much higher data throughput if open connections can be re-used to execute multiple requests.

HTTP/1.1 states that HTTP connections can be re-used for multiple requests per default. HTTP/1.0 compliant endpoints can also use a mechanism to explicitly communicate their preference to keep connection alive and use it for multiple requests. HTTP agents can also keep idle connections alive for a certain period time in case a connection to the same target host is needed for subsequent requests. The ability to keep connections alive is usually refered to as connection persistence. HttpClient fully supports connection persistence.

## 2.3. HTTP connection routing

HttpClient is capable of establishing connections to the target host either directly or via a route that may involve multiple intermediate connections - also referred to as hops. HttpClient differentiates connections of a route into plain, tunneled and layered. The use of multiple intermediate proxies to tunnel connections to the target host is referred to as proxy chaining.

Plain routes are established by connecting to the target or the first and only proxy. Tunnelled routes are established by connecting to the first and tunnelling through a chain of proxies to the target. Routes without a proxy cannot be tunnelled. Layered routes are established by layering a protocol over an existing connection. Protocols can only be layered over a tunnel to the target, or over a direct connection without proxies.

### 2.3.1. Route computation

The `RouteInfo` interface represents information about a definitive route to a target host involving one or more intermediate steps or hops. `HttpRoute` is a concrete implementation of the `RouteInfo`, which cannot be changed (is immutable). `HttpTracker` is a mutable `RouteInfo` implementation used internally by HttpClient to track the remaining hops to the ultimate route target. `HttpTracker` can be updated after a successful execution of the next hop towards the route target. `HttpRouteDirector` is a helper class that can be used to compute the next step in a route. This class is used internally by HttpClient.

`HttpRoutePlanner` is an interface representing a strategy to compute a complete route to a given target based on the execution context. HttpClient ships with two default `HttpRoutePlanner` implementations. `ProxySelectorRoutePlanner` is based on `java.net.ProxySelector`. By default, it will pick up the proxy settings of the JVM, either from system properties or from the browser running the application. The `DefaultHttpRoutePlanner` implementation does not make use of any Java system properties, nor any system or browser proxy settings. It computes routes based exclusively on the HTTP parameters described below.

### 2.3.2. Secure HTTP connections

HTTP connections can be considered secure if information transmitted between two connection endpoints cannot be read or tampered with by an unauthorized third party. The SSL/TLS protocol is the most widely used technique to ensure HTTP transport security. However, other encryption techniques could be employed as well. Usually, HTTP transport is layered over the SSL/TLS encrypted connection.

## 2.4. HTTP route parameters

These are the parameters that can influence route computation:

- **`ConnRoutePNames.DEFAULT_PROXY`='http.route.default-proxy':** defines a proxy host to be used by default route planners that do not make use of JRE settings. This parameter expects a value of type `HttpHost`. If this parameter is not set, direct connections to the target will be attempted.

- **`ConnRoutePNames.LOCAL_ADDRESS`='http.route.local-address':** defines a local address to be used by all default route planner. On machines with multiple network interfaces, this parameter can be used to select the network interface from which the connection originates. This parameter expects a value of type `java.net.InetAddress`. If this parameter is not set, a default local address will be used automatically.

- **`ConnRoutePNames.FORCED_ROUTE`='http.route.forced-route':** defines an forced route to be used by all default route planner. Instead of computing a route, the given forced route will be returned, even if it points to a completely different target host. This parameter expects a value of type `HttpRoute`.

## 2.5. Socket factories

HTTP connections make use of a `java.net.Socket` object internally to handle transmission of data across the wire. However they rely on the `SchemeSocketFactory` interface to create, initialize and connect sockets. This enables the users of HttpClient to provide application specific socket initialization code at runtime. `PlainSocketFactory` is the default factory for creating and initializing plain (unencrypted) sockets.

The process of creating a socket and that of connecting it to a host are decoupled, so that the socket could be closed while being blocked in the connect operation.

```
PlainSocketFactory sf = PlainSocketFactory.getSocketFactory();
Socket socket = sf.createSocket();

HttpParams params = new BasicHttpParams();
```

```
params.setParameter(CoreConnectionPNames.CONNECTION_TIMEOUT, 1000L);
InetSocketAddress address = new InetSocketAddress("locahost", 8080);
sf.connectSocket(socket, address, null, params);
```

### 2.5.1. Secure socket layering

`LayeredSchemeSocketFactory` is an extension of the `SchemeSocketFactory` interface. Layered socket factories are capable of creating sockets layered over an existing plain socket. Socket layering is used primarily for creating secure sockets through proxies. HttpClient ships with `SSLSocketFactory` that implements SSL/TLS layering. Please note HttpClient does not use any custom encryption functionality. It is fully reliant on standard Java Cryptography (JCE) and Secure Sockets (JSEE) extensions.

### 2.5.2. SSL/TLS customization

HttpClient makes use of SSLSocketFactory to create SSL connections. `SSLSocketFactory` allows for a high degree of customization. It can take an instance of `javax.net.ssl.SSLContext` as a parameter and use it to create custom configured SSL connections.

```
TrustManager easyTrustManager = new X509TrustManager() {

    @Override
    public void checkClientTrusted(
            X509Certificate[] chain,
            String authType) throws CertificateException {
        // Oh, I am easy!
    }

    @Override
    public void checkServerTrusted(
            X509Certificate[] chain,
            String authType) throws CertificateException {
        // Oh, I am easy!
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }

};

SSLContext sslcontext = SSLContext.getInstance("TLS");
sslcontext.init(null, new TrustManager[] { easyTrustManager }, null);

SSLSocketFactory sf = new SSLSocketFactory(sslcontext);
SSLSocket socket = (SSLSocket) sf.createSocket();
socket.setEnabledCipherSuites(new String[] { "SSL_RSA_WITH_RC4_128_MD5" });

HttpParams params = new BasicHttpParams();
params.setParameter(CoreConnectionPNames.CONNECTION_TIMEOUT, 1000L);
InetSocketAddress address = new InetSocketAddress("locahost", 443);
sf.connectSocket(socket, address, null, params);
```

Customization of SSLSocketFactory implies a certain degree of familiarity with the concepts of the SSL/TLS protocol, a detailed explanation of which is out of scope for this document. Please refer to the Java Secure Socket Extension [http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html] for a detailed description of `javax.net.ssl.SSLContext` and related tools.

### 2.5.3. Hostname verification

In addition to the trust verification and the client authentication performed on the SSL/TLS protocol level, HttpClient can optionally verify whether the target hostname matches the names stored inside the server's X.509 certificate, once the connection has been established. This verification can provide additional guarantees of authenticity of the server trust material. The `X509HostnameVerifier` interface represents a strategy for hostname verification. HttpClient ships with three `X509HostnameVerifier` implementations. Important: hostname verification should not be confused with SSL trust verification.

- **`StrictHostnameVerifier:`** The strict hostname verifier works the same way as Sun Java 1.4, Sun Java 5, Sun Java 6. It's also pretty close to IE6. This implementation appears to be compliant with RFC 2818 for dealing with wildcards. The hostname must match either the first CN, or any of the subject-alts. A wildcard can occur in the CN, and in any of the subject-alts.
- **`BrowserCompatHostnameVerifier:`** This hostname verifier that works the same way as Curl and Firefox. The hostname must match either the first CN, or any of the subject-alts. A wildcard can occur in the CN, and in any of the subject-alts. The only difference between `BrowserCompatHostnameVerifier` and `StrictHostnameVerifier` is that a wildcard (such as "*.foo.com") with `BrowserCompatHostnameVerifier` matches all subdomains, including "a.b.foo.com".
- **`AllowAllHostnameVerifier:`** This hostname verifier essentially turns hostname verification off. This implementation is a no-op, and never throws `javax.net.ssl.SSLException`.

Per default HttpClient uses the `BrowserCompatHostnameVerifier` implementation. One can specify a different hostname verifier implementation if desired

```
SSLSocketFactory sf = new SSLSocketFactory(
    SSLContext.getInstance("TLS"),
    SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
```

## 2.6. Protocol schemes

The `Scheme` class represents a protocol scheme such as "http" or "https" and contains a number of protocol properties such as the default port and the socket factory to be used to create the `java.net.Socket` instances for the given protocol. The `SchemeRegistry` class is used to maintain a set of `Schemes` that HttpClient can choose from when trying to establish a connection by a request URI:

```
Scheme http = new Scheme("http", 80, PlainSocketFactory.getSocketFactory());

SSLSocketFactory sf = new SSLSocketFactory(
        SSLContext.getInstance("TLS"),
        SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
Scheme https = new Scheme("https", 443, sf);

SchemeRegistry sr = new SchemeRegistry();
sr.register(http);
sr.register(https);
```

## 2.7. HttpClient proxy configuration

Even though HttpClient is aware of complex routing scemes and proxy chaining, it supports only simple direct or one hop proxy connections out of the box.

The simplest way to tell HttpClient to connect to the target host via a proxy is by setting the default proxy parameter:

```
DefaultHttpClient httpclient = new DefaultHttpClient();

HttpHost proxy = new HttpHost("someproxy", 8080);
httpclient.getParams().setParameter(ConnRoutePNames.DEFAULT_PROXY, proxy);
```

One can also instruct HttpClient to use the standard JRE proxy selector to obtain proxy information:

```
DefaultHttpClient httpclient = new DefaultHttpClient();

ProxySelectorRoutePlanner routePlanner = new ProxySelectorRoutePlanner(
        httpclient.getConnectionManager().getSchemeRegistry(),
        ProxySelector.getDefault());
httpclient.setRoutePlanner(routePlanner);
```

Alternatively, one can provide a custom `RoutePlanner` implementation in order to have a complete control over the process of HTTP route computation:

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setRoutePlanner(new HttpRoutePlanner() {

    public HttpRoute determineRoute(
            HttpHost target,
            HttpRequest request,
            HttpContext context) throws HttpException {
        return new HttpRoute(target, null,  new HttpHost("someproxy", 8080),
                "https".equalsIgnoreCase(target.getSchemeName()));
    }

});
```

# 2.8. HTTP connection managers

## 2.8.1. Connection operators

Operated connections are client side connections whose underlying socket or state can be manipulated by an external entity, usually referred to as a connection operator. The `OperatedClientConnection` interface extends the `HttpClientConnection` interface and defines additional methods to manage connection sockets. The `ClientConnectionOperator` interface represents a strategy for creating `OperatedClientConnection` instances and updating the underlying socket of those objects. Implementations will most likely make use a `SchemeSocketFactory` to create `java.net.Socket` instances. The `ClientConnectionOperator` interface enables users of HttpClient to provide a custom strategy for connection operators as well as the ability to provide an alternative implementation of the `OperatedClientConnection` interface.

## 2.8.2. Managed connections and connection managers

HTTP connections are complex, stateful, thread-unsafe objects which need to be properly managed to function correctly. HTTP connections can only be used by one execution thread at a time. HttpClient employs a special entity to manage access to HTTP connections called HTTP connection manager and represented by the `ClientConnectionManager` interface. The purpose of an HTTP connection manager is to serve as a factory for new HTTP connections, manage persistent connections and synchronize

access to persistent connections making sure that only one thread can have access to a connection at a time.

Internally HTTP connection managers work with instances of `OperatedClientConnection`, but they return instances of `ManagedClientConnection` to the service consumers. `ManagedClientConnection` acts as a wrapper for a `OperatedClientConnection` instance that manages its state and controls all I/O operations on that connection. It also abstracts away socket operations and provides convenience methods for opening and updating sockets in order to establish a route. `ManagedClientConnection` instances are aware of their link to the connection manager that spawned them and of the fact that they must be returned back to the manager when no longer in use. `ManagedClientConnection` classes also implement the `ConnectionReleaseTrigger` interface that can be used to trigger the release of the connection back to the manager. Once the connection release has been triggered the wrapped connection gets detached from the `ManagedClientConnection` wrapper and the `OperatedClientConnection` instance is returned back to the manager. Even though the service consumer still holds a reference to the `ManagedClientConnection` instance, it is no longer able to execute any I/O operation or change the state of the `OperatedClientConnection` either intentionally or unintentionally.

This is an example of acquiring a connection from a connection manager:

```
Scheme http = new Scheme("http", 80, PlainSocketFactory.getSocketFactory());
SchemeRegistry sr = new SchemeRegistry();
sr.register(http);
ClientConnectionManager connMrg = new SingleClientConnManager(sr);

// Request new connection. This can be a long process
ClientConnectionRequest connRequest = connMrg.requestConnection(
        new HttpRoute(new HttpHost("localhost", 80)), null);

// Wait for connection up to 10 sec
ManagedClientConnection conn = connRequest.getConnection(10, TimeUnit.SECONDS);
try {
    // Do useful things with the connection.
    // Release it when done.
    conn.releaseConnection();
} catch (IOException ex) {
    // Abort connection upon an I/O error.
    conn.abortConnection();
    throw ex;
}
```

The connection request can be terminated prematurely by calling `ClientConnectionRequest#abortRequest()` if necessary. This will unblock the thread blocked in the `ClientConnectionRequest#getConnection()` method.

`BasicManagedEntity` wrapper class can be used to ensure automatic release of the underlying connection once the response content has been fully consumed. HttpClient uses this mechanism internally to achieve transparent connection release for all responses obtained from `HttpClient#execute()` methods:

```
ClientConnectionRequest connRequest = connMrg.requestConnection(
        new HttpRoute(new HttpHost("localhost", 80)), null);
ManagedClientConnection conn = connRequest.getConnection(10, TimeUnit.SECONDS);
try {
    BasicHttpRequest request = new BasicHttpRequest("GET", "/");
    conn.sendRequestHeader(request);
```

```
    HttpResponse response = conn.receiveResponseHeader();
    conn.receiveResponseEntity(response);
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        BasicManagedEntity managedEntity = new BasicManagedEntity(entity, conn, true);
        // Replace entity
        response.setEntity(managedEntity);
    }
    // Do something useful with the response
    // The connection will be released automatically
    // as soon as the response content has been consumed
} catch (IOException ex) {
    // Abort connection upon an I/O error.
    conn.abortConnection();
    throw ex;
}
```

## 2.8.3. Simple connection manager

`SingleClientConnManager` is a simple connection manager that maintains only one connection at a time. Even though this class is thread-safe it ought to be used by one execution thread only. `SingleClientConnManager` will make an effort to reuse the connection for subsequent requests with the same route. It will, however, close the existing connection and re-open it for the given route, if the route of the persistent connection does not match that of the connection request. If the connection has been already been allocated, then `java.lang.IllegalStateException` is thrown.

`SingleClientConnManager` is used by HttpClient per default.

## 2.8.4. Pooling connection manager

`ThreadSafeClientConnManager` is a more complex implementation that manages a pool of client connections and is able to service connection requests from multiple execution threads. Connections are pooled on a per route basis. A request for a route for which the manager already has a persistent connection available in the pool will be serviced by leasing a connection from the pool rather than creating a brand new connection.

`ThreadSafeClientConnManager` maintains a maximum limit of connections on a per route basis and in total. Per default this implementation will create no more than 2 concurrent connections per given route and no more 20 connections in total. For many real-world applications these limits may prove too constraining, especially if they use HTTP as a transport protocol for their services. Connection limits can be adjusted using the appropriate HTTP parameters.

This example shows how the connection pool parameters can be adjusted:

```
SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(
         new Scheme("http", 80, PlainSocketFactory.getSocketFactory()));
schemeRegistry.register(
         new Scheme("https", 443, SSLSocketFactory.getSocketFactory()));

ThreadSafeClientConnManager cm = new ThreadSafeClientConnManager(schemeRegistry);
// Increase max total connection to 200
cm.setMaxTotalConnections(200);
// Increase default max connection per route to 20
cm.setDefaultMaxPerRoute(20);
// Increase max connections for localhost:80 to 50
HttpHost localhost = new HttpHost("locahost", 80);
cm.setMaxForRoute(new HttpRoute(localhost), 50);
```

```
HttpClient httpClient = new DefaultHttpClient(cm);
```

### 2.8.5. Connection manager shutdown

When an HttpClient instance is no longer needed and is about to go out of scope it is important to shut down its connection manager to ensure that all connections kept alive by the manager get closed and system resources allocated by those connections are released.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://www.google.com/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
System.out.println(response.getStatusLine());
EntityUtils.consume(entity);
httpclient.getConnectionManager().shutdown();
```

## 2.9. Multithreaded request execution

When equipped with a pooling connection manager such as ThreadSafeClientConnManager, HttpClient can be used to execute multiple requests simultaneously using multiple threads of execution.

The `ThreadSafeClientConnManager` will allocate connections based on its configuration. If all connections for a given route have already been leased, a request for a connection will block until a connection is released back to the pool. One can ensure the connection manager does not block indefinitely in the connection request operation by setting `'http.conn-manager.timeout'` to a positive value. If the connection request cannot be serviced within the given time period `ConnectionPoolTimeoutException` will be thrown.

```
SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(
        new Scheme("http", 80, PlainSocketFactory.getSocketFactory()));

ClientConnectionManager cm = new ThreadSafeClientConnManager(schemeRegistry);
HttpClient httpClient = new DefaultHttpClient(cm);

// URIs to perform GETs on
String[] urisToGet = {
    "http://www.domain1.com/",
    "http://www.domain2.com/",
    "http://www.domain3.com/",
    "http://www.domain4.com/"
};

// create a thread for each URI
GetThread[] threads = new GetThread[urisToGet.length];
for (int i = 0; i < threads.length; i++) {
    HttpGet httpget = new HttpGet(urisToGet[i]);
    threads[i] = new GetThread(httpClient, httpget);
}

// start the threads
for (int j = 0; j < threads.length; j++) {
    threads[j].start();
}

// join the threads
for (int j = 0; j < threads.length; j++) {
    threads[j].join();
```

```
}
```

```
static class GetThread extends Thread {

    private final HttpClient httpClient;
    private final HttpContext context;
    private final HttpGet httpget;

    public GetThread(HttpClient httpClient, HttpGet httpget) {
        this.httpClient = httpClient;
        this.context = new BasicHttpContext();
        this.httpget = httpget;
    }

    @Override
    public void run() {
        try {
            HttpResponse response = this.httpClient.execute(this.httpget, this.context);
            HttpEntity entity = response.getEntity();
            if (entity != null) {
                // do something useful with the entity
            }
            // ensure the connection gets released to the manager
            EntityUtils.consume(entity);
        } catch (Exception ex) {
            this.httpget.abort();
        }
    }

}
```

## 2.10. Connection eviction policy

One of the major shortcomings of the classic blocking I/O model is that the network socket can react to I/O events only when blocked in an I/O operation. When a connection is released back to the manager, it can be kept alive however it is unable to monitor the status of the socket and react to any I/O events. If the connection gets closed on the server side, the client side connection is unable to detect the change in the connection state (and react appropriately by closing the socket on its end).

HttpClient tries to mitigate the problem by testing whether the connection is 'stale', that is no longer valid because it was closed on the server side, prior to using the connection for executing an HTTP request. The stale connection check is not 100% reliable and adds 10 to 30 ms overhead to each request execution. The only feasible solution that does not involve a one thread per socket model for idle connections is a dedicated monitor thread used to evict connections that are considered expired due to a long period of inactivity. The monitor thread can periodically call `ClientConnectionManager#closeExpiredConnections()` method to close all expired connections and evict closed connections from the pool. It can also optionally call `ClientConnectionManager#closeIdleConnections()` method to close all connections that have been idle over a given period of time.

```
public static class IdleConnectionMonitorThread extends Thread {

    private final ClientConnectionManager connMgr;
    private volatile boolean shutdown;

    public IdleConnectionMonitorThread(ClientConnectionManager connMgr) {
        super();
```

```
        this.connMgr = connMgr;
    }

    @Override
    public void run() {
        try {
            while (!shutdown) {
                synchronized (this) {
                    wait(5000);
                    // Close expired connections
                    connMgr.closeExpiredConnections();
                    // Optionally, close connections
                    // that have been idle longer than 30 sec
                    connMgr.closeIdleConnections(30, TimeUnit.SECONDS);
                }
            }
        } catch (InterruptedException ex) {
            // terminate
        }
    }

    public void shutdown() {
        shutdown = true;
        synchronized (this) {
            notifyAll();
        }
    }

}
```

## 2.11. Connection keep alive strategy

The HTTP specification does not specify how long a persistent connection may be and should be kept alive. Some HTTP servers use a non-standard `Keep-Alive` header to communicate to the client the period of time in seconds they intend to keep the connection alive on the server side. HttpClient makes use of this information if available. If the `Keep-Alive` header is not present in the response, HttpClient assumes the connection can be kept alive indefinitely. However, many HTTP servers in general use are configured to drop persistent connections after a certain period of inactivity in order to conserve system resources, quite often without informing the client. In case the default strategy turns out to be too optimistic, one may want to provide a custom keep-alive strategy.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setKeepAliveStrategy(new ConnectionKeepAliveStrategy() {

    public long getKeepAliveDuration(HttpResponse response, HttpContext context) {
        // Honor 'keep-alive' header
        HeaderElementIterator it = new BasicHeaderElementIterator(
                response.headerIterator(HTTP.CONN_KEEP_ALIVE));
        while (it.hasNext()) {
            HeaderElement he = it.nextElement();
            String param = he.getName();
            String value = he.getValue();
            if (value != null && param.equalsIgnoreCase("timeout")) {
                try {
                    return Long.parseLong(value) * 1000;
                } catch(NumberFormatException ignore) {
                }
            }
        }
        HttpHost target = (HttpHost) context.getAttribute(
                ExecutionContext.HTTP_TARGET_HOST);
        if ("www.naughty-server.com".equalsIgnoreCase(target.getHostName())) {
```

```
            // Keep alive for 5 seconds only
            return 5 * 1000;
        } else {
            // otherwise keep alive for 30 seconds
            return 30 * 1000;
        }
    }

});
```

# Chapter 3. HTTP state management

Originally HTTP was designed as a stateless, request / response oriented protocol that made no special provisions for stateful sessions spanning across several logically related request / response exchanges. As HTTP protocol grew in popularity and adoption more and more systems began to use it for applications it was never intended for, for instance as a transport for e-commerce applications. Thus, the support for state management became a necessity.

Netscape Communications, at that time a leading developer of web client and server software, implemented support for HTTP state management in their products based on a proprietary specification. Later, Netscape tried to standardise the mechanism by publishing a specification draft. Those efforts contributed to the formal specification defined through the RFC standard track. However, state management in a significant number of applications is still largely based on the Netscape draft and is incompatible with the official specification. All major developers of web browsers felt compelled to retain compatibility with those applications greatly contributing to the fragmentation of standards compliance.

## 3.1. HTTP cookies

An HTTP cookie is a token or short packet of state information that the HTTP agent and the target server can exchange to maintain a session. Netscape engineers used to refer to it as a "magic cookie" and the name stuck.

HttpClient uses the `Cookie` interface to represent an abstract cookie token. In its simplest form an HTTP cookie is merely a name / value pair. Usually an HTTP cookie also contains a number of attributes such as version, a domain for which is valid, a path that specifies the subset of URLs on the origin server to which this cookie applies, and the maximum period of time for which the cookie is valid.

The `SetCookie` interface represents a `Set-Cookie` response header sent by the origin server to the HTTP agent in order to maintain a conversational state. The `SetCookie2` interface extends SetCookie with `Set-Cookie2` specific methods.

The `ClientCookie` interface extends `Cookie` interface with additional client specific functionality such as the ability to retrieve original cookie attributes exactly as they were specified by the origin server. This is important for generating the `Cookie` header because some cookie specifications require that the `Cookie` header should include certain attributes only if they were specified in the `Set-Cookie` or `Set-Cookie2` header.

### 3.1.1. Cookie versions

Cookies compatible with Netscape draft specification but non-compliant with the official specification are considered to be of version 0. Standard compliant cookies are expected to have version 1. HttpClient may handle cookies differently depending on the version.

Here is an example of re-creating a Netscape cookie:

```
BasicClientCookie netscapeCookie = new BasicClientCookie("name", "value");
netscapeCookie.setVersion(0);
netscapeCookie.setDomain(".mycompany.com");
netscapeCookie.setPath("/");
```

Here is an example of re-creating a standard cookie. Please note that standard compliant cookie must retain all attributes as sent by the origin server:

```
BasicClientCookie stdCookie = new BasicClientCookie("name", "value");
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPath("/");
stdCookie.setSecure(true);
// Set attributes EXACTLY as sent by the server
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR, ".mycompany.com");
```

Here is an example of re-creating a `Set-Cookie2` compliant cookie. Please note that standard compliant cookie must retain all attributes as sent by the origin server:

```
BasicClientCookie2 stdCookie = new BasicClientCookie2("name", "value");
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPorts(new int[] {80,8080});
stdCookie.setPath("/");
stdCookie.setSecure(true);
// Set attributes EXACTLY as sent by the server
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR, ".mycompany.com");
stdCookie.setAttribute(ClientCookie.PORT_ATTR, "80,8080");
```

## 3.2. Cookie specifications

The `CookieSpec` interface represents a cookie management specification. The cookie management specification is expected to enforce:

- rules of parsing `Set-Cookie` and optionally `Set-Cookie2` headers.
- rules of validation of parsed cookies.
- formatting of `Cookie` header for a given host, port and path of origin.

HttpClient ships with several `CookieSpec` implementations:

- **Netscape draft:** This specification conforms to the original draft specification published by Netscape Communications. It should be avoided unless absolutely necessary for compatibility with legacy code.
- **RFC 2109:** Older version of the official HTTP state management specification superseded by RFC 2965.
- **RFC 2965:** The official HTTP state management specification.
- **Browser compatibility:** This implementation strives to closely mimic the (mis)behavior of common web browser applications such as Microsoft Internet Explorer and Mozilla FireFox.
- **Best match:** 'Meta' cookie specification that picks up a cookie policy based on the format of cookies sent with the HTTP response. It basically aggregates all above implementations into one class.

It is strongly recommended to use the `Best Match` policy and let HttpClient pick up an appropriate compliance level at runtime based on the execution context.

## 3.3. HTTP cookie and state management parameters

These are parameters that be used to customize HTTP state management and the behaviour of individual cookie specifications:

- **`CookieSpecPNames.DATE_PATTERNS`='http.protocol.cookie-datepatterns':** defines valid date patterns to be used for parsing non-standard `expires` attribute. Only required for compatibility with non-compliant servers that still use `expires` defined in the Netscape draft instead of the standard `max-age` attribute. This parameter expects a value of type `java.util.Collection`. The collection elements must be of type `java.lang.String` compatible with the syntax of `java.text.SimpleDateFormat`. If this parameter is not set the choice of a default value is `CookieSpec` implementation specific.

- **`CookieSpecPNames.SINGLE_COOKIE_HEADER`='http.protocol.single-cookie-header':** defines whether cookies should be forced into a single `Cookie` request header. Otherwise, each cookie is formatted as a separate `Cookie` header. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set, the choice of a default value is CookieSpec implementation specific. Please note this parameter applies to strict cookie specifications (RFC 2109 and RFC 2965) only. Browser compatibility and netscape draft policies will always put all cookies into one request header.

- **`ClientPNames.COOKIE_POLICY`='http.protocol.cookie-policy':** defines the name of a cookie specification to be used for HTTP state management. This parameter expects a value of type `java.lang.String`. If this parameter is not set, valid date patterns are `CookieSpec` implementation specific.

## 3.4. Cookie specification registry

HttpClient maintains a registry of available cookie specifications using the `CookieSpecRegistry` class. The following specifications are registered per default:

- **compatibility:** Browser compatibility (lenient policy).
- **netscape:** Netscape draft.
- **rfc2109:** RFC 2109 (outdated strict policy).
- **rfc2965:** RFC 2965 (standard conformant strict policy).
- **best-match:** Best match meta-policy.

## 3.5. Choosing cookie policy

Cookie policy can be set at the HTTP client and overridden on the HTTP request level if required.

```
HttpClient httpclient = new DefaultHttpClient();
// force strict cookie policy per default
httpclient.getParams().setParameter(
        ClientPNames.COOKIE_POLICY, CookiePolicy.RFC_2965);

HttpGet httpget = new HttpGet("http://www.broken-server.com/");
// Override the default policy for this request
httpget.getParams().setParameter(
        ClientPNames.COOKIE_POLICY, CookiePolicy.BROWSER_COMPATIBILITY);
```

## 3.6. Custom cookie policy

In order to implement a custom cookie policy one should create a custom implementation of the `CookieSpec` interface, create a `CookieSpecFactory` implementation to create and initialize instances of the custom specification and register the factory with HttpClient. Once the custom specification has been registered, it can be activated the same way as a standard cookie specification.

```
CookieSpecFactory csf = new CookieSpecFactory() {
    public CookieSpec newInstance(HttpParams params) {
        return new BrowserCompatSpec() {
            @Override
            public void validate(Cookie cookie, CookieOrigin origin)
            throws MalformedCookieException {
                // Oh, I am easy
            }
        };
    }
};

DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.getCookieSpecs().register("easy", csf);
httpclient.getParams().setParameter(
     ClientPNames.COOKIE_POLICY, "easy");
```

## 3.7. Cookie persistence

HttpClient can work with any physical representation of a persistent cookie store that implements the `CookieStore` interface. The default `CookieStore` implementation called `BasicCookieStore` is a simple implementation backed by a `java.util.ArrayList`. Cookies stored in an `BasicClientCookie` object are lost when the container object get garbage collected. Users can provide more complex implementations if necessary.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
// Create a local instance of cookie store
CookieStore cookieStore = new MyCookieStore();
// Populate cookies if needed
BasicClientCookie cookie = new BasicClientCookie("name", "value");
cookie.setVersion(0);
cookie.setDomain(".mycompany.com");
cookie.setPath("/");
cookieStore.addCookie(cookie);
// Set the store
httpclient.setCookieStore(cookieStore);
```

## 3.8. HTTP state management and execution context

In the course of HTTP request execution HttpClient adds the following state management related objects to the execution context:

- **`ClientContext.COOKIESPEC_REGISTRY`='http.cookiespec-registry':** `CookieSpecRegistry` instance representing the actual cookie specification registry. The value of this attribute set in the local context takes precedence over the default one.
- **`ClientContext.COOKIE_SPEC`='http.cookie-spec':** `CookieSpec` instance representing the actual cookie specification.

- **`ClientContext.COOKIE_ORIGIN`='http.cookie-origin':** `CookieOrigin` instance representing the actual details of the origin server.
- **`ClientContext.COOKIE_STORE`='http.cookie-store':** `CookieStore` instance representing the actual cookie store. The value of this attribute set in the local context takes precedence over the default one.

The local `HttpContext` object can be used to customize the HTTP state management context prior to request execution, or to examine its state after the request has been executed:

```
HttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext);

CookieOrigin cookieOrigin = (CookieOrigin) localContext.getAttribute(
        ClientContext.COOKIE_ORIGIN);
System.out.println("Cookie origin: " + cookieOrigin);
CookieSpec cookieSpec = (CookieSpec) localContext.getAttribute(
        ClientContext.COOKIE_SPEC);
System.out.println("Cookie spec used: " + cookieSpec);
```

## 3.9. Per user / thread state management

One can use an individual local execution context in order to implement per user (or per thread) state management. A cookie specification registry and cookie store defined in the local context will take precedence over the default ones set at the HTTP client level.

```
HttpClient httpclient = new DefaultHttpClient();
// Create a local instance of cookie store
CookieStore cookieStore = new BasicCookieStore();
// Create local HTTP context
HttpContext localContext = new BasicHttpContext();
// Bind custom cookie store to the local context
localContext.setAttribute(ClientContext.COOKIE_STORE, cookieStore);
HttpGet httpget = new HttpGet("http://www.google.com/");
// Pass local context as a parameter
HttpResponse response = httpclient.execute(httpget, localContext);
```

# Chapter 4. HTTP authentication

HttpClient provides full support for authentication schemes defined by the HTTP standard specification as well as a number of widely used non-standard authentication schemes such as `NTLM` and `SPNEGO`.

## 4.1. User credentials

Any process of user authentication requires a set of credentials that can be used to establish user identity. In the simplest form user credentials can be just a user name / password pair. `UsernamePasswordCredentials` represents a set of credentials consisting of a security principal and a password in clear text. This implementation is sufficient for standard authentication schemes defined by the HTTP standard specification.

```
UsernamePasswordCredentials creds = new UsernamePasswordCredentials("user", "pwd");
System.out.println(creds.getUserPrincipal().getName());
System.out.println(creds.getPassword());
```

stdout >

```
user
pwd
```

`NTCredentials` is a Microsoft Windows specific implementation that includes in addition to the user name / password pair a set of additional Windows specific attributes such as the name of the user domain. In a Microsoft Windows network the same user can belong to multiple domains each with a different set of authorizations.

```
NTCredentials creds = new NTCredentials("user", "pwd", "workstation", "domain");
System.out.println(creds.getUserPrincipal().getName());
System.out.println(creds.getPassword());
```

stdout >

```
DOMAIN/user
pwd
```

## 4.2. Authentication schemes

The `AuthScheme` interface represents an abstract challenge-response oriented authentication scheme. An authentication scheme is expected to support the following functions:

- Parse and process the challenge sent by the target server in response to request for a protected resource.
- Provide properties of the processed challenge: the authentication scheme type and its parameters, such the realm this authentication scheme is applicable to, if available

- Generate the authorization string for the given set of credentials and the HTTP request in response to the actual authorization challenge.

Please note that authentication schemes may be stateful involving a series of challenge-response exchanges.

HttpClient ships with several `AuthScheme` implementations:

- **Basic:** Basic authentication scheme as defined in RFC 2617. This authentication scheme is insecure, as the credentials are transmitted in clear text. Despite its insecurity Basic authentication scheme is perfectly adequate if used in combination with the TLS/SSL encryption.

- **Digest.** Digest authentication scheme as defined in RFC 2617. Digest authentication scheme is significantly more secure than Basic and can be a good choice for those applications that do not want the overhead of full transport security through TLS/SSL encryption.

- **NTLM:** NTLM is a proprietary authentication scheme developed by Microsoft and optimized for Windows platforms. NTLM is believed to be more secure than Digest.

- **SPNEGO/Kerberos:** `SPNEGO` (*S*imple and *P*rotected `GSSAPI` *Nego*tiation Mechanism) is a `GSSAPI` "pseudo mechanism" that is used to negotiate one of a number of possible real mechanisms. SPNEGO's most visible use is in Microsoft's `HTTP Negotiate` authentication extension. The negotiable sub-mechanisms include NTLM and Kerberos supported by Active Directory. At present HttpClient only supports the Kerberos sub-mechanism.

## 4.3. HTTP authentication parameters

These are parameters that be used to customize the HTTP authentication process and behaviour of individual authentication schemes:

- **`ClientPNames.HANDLE_AUTHENTICATION`='http.protocol.handle-authentication':** defines whether authentication should be handled automatically. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set, HttpClient will handle authentication automatically.

- **`AuthPNames.CREDENTIAL_CHARSET`='http.auth.credential-charset':** defines the charset to be used when encoding user credentials. This parameter expects a value of type `java.lang.String`. If this parameter is not set, `US-ASCII` will be used.

- **`AuthPNames.TARGET_AUTH_PREF`='http.auth.target-scheme-pref':** Defines the order of preference for supported `AuthSchemes` when authenticating with the target host. This parameter expects a value of type java.util.Collection. The collection is expected to contain `java.lang.String` instances representing an id of an authentication scheme.

- **`AuthPNames.PROXY_AUTH_PREF`='http.auth.proxy-scheme-pref':** Defines the order of preference for supported `AuthSchemes` when authenticating with the proxy host. This parameter expects a value of type java.util.Collection. The collection is expected to contain `java.lang.String` instances representing an id of an authentication scheme.

For example, one can force HttpClient to use a different order of preference for authentication schemes

```
DefaultHttpClient httpclient = new DefaultHttpClient(ccm, params);
```

```
// Choose BASIC over DIGEST for proxy authentication
List<String> authpref = new ArrayList<String>();
authpref.add(AuthPolicy.BASIC);
authpref.add(AuthPolicy.DIGEST);
httpclient.getParams().setParameter(AuthPNames.PROXY_AUTH_PREF, authpref);
```

## 4.4. Authentication scheme registry

HttpClient maintains a registry of available authentication schemes using the `AuthSchemeRegistry` class. The following schemes are registered per default:

- **AuthPolicy.BASIC:** Basic authentication
- **AuthPolicy.DIGEST:** Digest authentication
- **AuthPolicy.NTLM:** NTLMv1, NTLMv2, and NTLM2 Session authentication
- **AuthPolicy.SPNEGO:** SPNEGO/Kerberos authentication

## 4.5. Credentials provider

Credentials providers are intended to maintain a set of user credentials and to be able to produce user credentials for a particular authentication scope. Authentication scope consists of a host name, a port number, a realm name and an authentication scheme name. When registering credentials with the credentials provider one can provide a wild card (any host, any port, any realm, any scheme) instead of a concrete attribute value. The credentials provider is then expected to be able to find the closest match for a particular scope if the direct match cannot be found.

HttpClient can work with any physical representation of a credentials provider that implements the `CredentialsProvider` interface. The default `CredentialsProvider` implementation called `BasicCredentialsProvider` is a simple implementation backed by a `java.util.HashMap`.

```
CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.setCredentials(
    new AuthScope("somehost", AuthScope.ANY_PORT),
    new UsernamePasswordCredentials("u1", "p1"));
credsProvider.setCredentials(
    new AuthScope("somehost", 8080),
    new UsernamePasswordCredentials("u2", "p2"));
credsProvider.setCredentials(
    new AuthScope("otherhost", 8080, AuthScope.ANY_REALM, "ntlm"),
    new UsernamePasswordCredentials("u3", "p3"));

System.out.println(credsProvider.getCredentials(
    new AuthScope("somehost", 80, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
    new AuthScope("somehost", 8080, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
    new AuthScope("otherhost", 8080, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
    new AuthScope("otherhost", 8080, null, "ntlm")));
```

stdout >

```
[principal: u1]
[principal: u2]
```

```
null
[principal: u3]
```

## 4.6. HTTP authentication and execution context

HttpClient relies on the `AuthState` class to keep track of detailed information about the state of the authentication process. HttpClient creates two instances of `AuthState` in the course of HTTP request execution: one for target host authentication and another one for proxy authentication. In case the target server or the proxy require user authentication the respective `AuthScope` instance will be populated with the `AuthScope`, `AuthScheme` and `Crednetials` used during the authentication process. The `AuthState` can be examined in order to find out what kind of authentication was requested, whether a matching `AuthScheme` implementation was found and whether the credentials provider managed to find user credentials for the given authentication scope.

In the course of HTTP request execution HttpClient adds the following authentication related objects to the execution context:

- **`ClientContext.AUTHSCHEME_REGISTRY`='http.authscheme-registry':** `AuthSchemeRegistry` instance representing the actual authentication scheme registry. The value of this attribute set in the local context takes precedence over the default one.
- **`ClientContext.CREDS_PROVIDER`='http.auth.credentials-provider':** `CookieSpec` instance representing the actual credentials provider. The value of this attribute set in the local context takes precedence over the default one.
- **`ClientContext.TARGET_AUTH_STATE`='http.auth.target-scope':** `AuthState` instance representing the actual target authentication state. The value of this attribute set in the local context takes precedence over the default one.
- **`ClientContext.PROXY_AUTH_STATE`='http.auth.proxy-scope':** `AuthState` instance representing the actual proxy authentication state. The value of this attribute set in the local context takes precedence over the default one.
- **`ClientContext.AUTH_CACHE`='http.auth.auth-cache':** `AuthCache` instance representing the actual authentication data cache. The value of this attribute set in the local context takes precedence over the default one.

The local `HttpContext` object can be used to customize the HTTP authentication context prior to request execution, or to examine its state after the request has been executed:

```
HttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext);

AuthState proxyAuthState = (AuthState) localContext.getAttribute(
    ClientContext.PROXY_AUTH_STATE);
System.out.println("Proxy auth scope: " + proxyAuthState.getAuthScope());
System.out.println("Proxy auth scheme: " + proxyAuthState.getAuthScheme());
System.out.println("Proxy auth credentials: " + proxyAuthState.getCredentials());
AuthState targetAuthState = (AuthState) localContext.getAttribute(
    ClientContext.TARGET_AUTH_STATE);
System.out.println("Target auth scope: " + targetAuthState.getAuthScope());
System.out.println("Target auth scheme: " + targetAuthState.getAuthScheme());
```

```
System.out.println("Target auth credentials: " + targetAuthState.getCredentials());
```

## 4.7. Caching of authentication data

As of version 4.1 HttpClient automatically caches information about hosts it has successfully authenticated with. Please note that one must use the same execution context to execute logically related requests in order for cached authentication data to propagate from one request to another. Authentication data will be lost as soon as the execution context goes out of scope.

## 4.8. Preemptive authentication

HttpClient does not support preemptive authentication out of the box, because if misused or used incorrectly the preemptive authentication can lead to significant security issues, such as sending user credentials in clear text to an unauthorized third party. Therefore, users are expected to evaluate potential benefits of preemptive authentication versus security risks in the context of their specific application environment.

Nonethess one can configure HttpClient to authenticate preemptively by prepopulating the authentication data cache.

```
HttpHost targetHost = new HttpHost("localhost", 80, "http");

DefaultHttpClient httpclient = new DefaultHttpClient();

httpclient.getCredentialsProvider().setCredentials(
        new AuthScope(targetHost.getHostName(), targetHost.getPort()),
        new UsernamePasswordCredentials("username", "password"));

// Create AuthCache instance
AuthCache authCache = new BasicAuthCache();
// Generate BASIC scheme object and add it to the local auth cache
BasicScheme basicAuth = new BasicScheme();
authCache.put(targetHost, basicAuth);

// Add AuthCache to the execution context
BasicHttpContext localcontext = new BasicHttpContext();
localcontext.setAttribute(ClientContext.AUTH_CACHE, authCache);

HttpGet httpget = new HttpGet("/");
for (int i = 0; i < 3; i++) {
    HttpResponse response = httpclient.execute(targetHost, httpget, localcontext);
    HttpEntity entity = response.getEntity();
    EntityUtils.consume(entity);
}
```

## 4.9. NTLM Authentication

As of version 4.1 HttpClient provides full support for NTLMv1, NTLMv2, and NTLM2 Session authentication out of the box. One can still continue using an external `NTLM` engine such as JCIFS [http://jcifs.samba.org/] library developed by the Samba [http://www.samba.org/] project as a part of their Windows interoperability suite of programs.

### 4.9.1. NTLM connection persistence

The `NTLM` authentication scheme is significantly more expensive in terms of computational overhead and performance impact than the standard `Basic` and `Digest` schemes. This is likely to be one of

the main reasons why Microsoft chose to make NTLM authentication scheme stateful. That is, once authenticated, the user identity is associated with that connection for its entire life span. The stateful nature of NTLM connections makes connection persistence more complex, as for the obvious reason persistent NTLM connections may not be re-used by users with a different user identity. The standard connection managers shipped with HttpClient are fully capable of managing stateful connections. However, it is critically important that logically related requests within the same session use the same execution context in order to make them aware of the current user identity. Otherwise, HttpClient will end up creating a new HTTP connection for each HTTP request against NTLM protected resources. For detailed discussion on stateful HTTP connections please refer to this section.

As NTLM connections are stateful it is generally recommended to trigger NTLM authentication using a relatively cheap method, such as GET or HEAD, and re-use the same connection to execute more expensive methods, especially those enclose a request entity, such as POST or PUT.

```
DefaultHttpClient httpclient = new DefaultHttpClient();

NTCredentials creds = new NTCredentials("user", "pwd", "myworkstation", "microsoft.com");
httpclient.getCredentialsProvider().setCredentials(AuthScope.ANY, creds);

HttpHost target = new HttpHost("www.microsoft.com", 80, "http");

// Make sure the same context is used to execute logically related requests
HttpContext localContext = new BasicHttpContext();

// Execute a cheap method first. This will trigger NTLM authentication
HttpGet httpget = new HttpGet("/ntlm-protected/info");
HttpResponse response1 = httpclient.execute(target, httpget, localContext);
HttpEntity entity1 = response1.getEntity();
EntityUtils.consume(entity1);

// Execute an expensive method next reusing the same context (and connection)
HttpPost httppost = new HttpPost("/ntlm-protected/form");
httppost.setEntity(new StringEntity("lots and lots of data"));
HttpResponse response2 = httpclient.execute(target, httppost, localContext);
HttpEntity entity2 = response2.getEntity();
EntityUtils.consume(entity2);
```

## 4.10. SPNEGO/Kerberos Authentication

The SPNEGO (*S*imple and *P*rotected GSSAPI *Nego*tiation Mechanism) is designed to allow for authentication to services when neither end knows what the other can use/provide. It is most commonly used to do Kerberos authentication. It can wrap other mechanisms, however the current version in HttpClient is designed solely with Kerberos in mind.

1. Client Web Browser does HTTP GET for resource.
2. Web server returns HTTP 401 status and a header: `WWW-Authenticate: Negotiate`
3. Client generates a `NegTokenInit`, base64 encodes it, and resubmits the `GET` with an Authorization header: `Authorization: Negotiate <base64 encoding>`.
4. Server decodes the `NegTokenInit`, extracts the supported `MechTypes` (only Kerberos V5 in our case), ensures it is one of the expected ones, and then extracts the `MechToken` (Kerberos Token) and authenticates it.

   If more processing is required another HTTP 401 is returned to the client with more data in the the `WWW-Authenticate` header. Client takes the info and generates another token passing this back in the `Authorization` header until complete.
5. When the client has been authenticated the Web server should return the HTTP 200 status, a final `WWW-Authenticate` header and the page content.

### 4.10.1. `SPNEGO` support in HttpClient

The `SPNEGO` authentication scheme is compatible with Sun Java versions 1.5 and up. However the use of Java >= 1.6 is strongly recommended as it supports `SPNEGO` authentication more completely.

The Sun JRE provides the supporting classes to do nearly all the Kerberos and `SPNEGO` token handling. This means that a lot of the setup is for the GSS classes. The `NegotiateScheme` is a simple class to handle marshalling the tokens and reading and writing the correct headers.

The best way to start is to grab the `KerberosHttpClient.java` file in examples and try and get it to work. There are a lot of issues that can happen but if lucky it'll work without too much of a problem. It should also provide some output to debug with.

In Windows it should default to using the logged in credentials; this can be overridden by using 'kinit' e.g. `$JAVA_HOME\bin\kinit testuser@AD.EXAMPLE.NET`, which is very helpful for testing and debugging issues. Remove the cache file created by kinit to revert back to the windows Kerberos cache.

Make sure to list `domain_realms` in the `krb5.conf` file. This is a major source of problems.

### 4.10.2. GSS/Java Kerberos Setup

This documentation assumes you are using Windows but much of the information applies to Unix as well.

The `org.ietf.jgss` classes have lots of possible configuration parameters, mainly in the `krb5.conf`/`krb5.ini` file. Some more info on the format at http://web.mit.edu/kerberos/krb5-1.4/krb5-1.4.1/doc/krb5-admin/krb5.conf.html.

### 4.10.3. `login.conf` file

The following configuration is a basic setup that works in Windows XP against both `IIS` and `JBoss Negotiation` modules.

The system property `java.security.auth.login.config` can be used to point at the `login.conf` file.

`login.conf` content may look like the following:

```
com.sun.security.jgss.login {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
};

com.sun.security.jgss.initiate {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
};

com.sun.security.jgss.accept {
  com.sun.security.auth.module.Krb5LoginModule required client=TRUE useTicketCache=true;
};
```

### 4.10.4. `krb5.conf` / `krb5.ini` file

If unspecified, the system default will be used. Override if needed by setting the system property `java.security.krb5.conf` to point to a custom `krb5.conf` file.

`krb5.conf` content may look like the following:

```
[libdefaults]
    default_realm = AD.EXAMPLE.NET
    udp_preference_limit = 1
[realms]
    AD.EXAMPLE.NET = {
        kdc = KDC.AD.EXAMPLE.NET
    }
[domain_realms]
.ad.example.net=AD.EXAMPLE.NET
ad.example.net=AD.EXAMPLE.NET
```

### 4.10.5. Windows Specific configuration

To allow Windows to use the current user's tickets, the system property `javax.security.auth.useSubjectCredsOnly` must be set to `false` and the Windows registry key `allowtgtsessionkey` should be added and set correctly to allow session keys to be sent in the Kerberos Ticket-Granting Ticket.

On the Windows Server 2003 and Windows 2000 SP4, here is the required registry setting:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters
Value Name: allowtgtsessionkey
Value Type: REG_DWORD
Value: 0x01
```

Here is the location of the registry setting on Windows XP SP2:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\
Value Name: allowtgtsessionkey
Value Type: REG_DWORD
Value: 0x01
```

## 4.10.6. Customizing SPNEGO authentication scheme

In order to customize SPNEGO support a new instance of the NegotiateSchemeFactory class must be created and registered with the authentication scheme registry of HttpClient.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
NegotiateSchemeFactory nsf = new NegotiateSchemeFactory();
httpclient.getAuthSchemes().register(AuthPolicy.SPNEGO, nsf);
```

There are several options that can be used to customize the behaviour of NegotiateSchemeFactory.

### 4.10.6.1. Strip port

Strips the port off service names e.g. HTTP/webserver.ad.example.net:8080 -> HTTP/webserver.ad.example.net

Found it useful when authenticating against JBoss Negotiation.

### 4.10.6.2. Custom SPNEGO token generator

Use this method to inject a custom SpnegoTokenGenerator class to do the Kerberos to SPNEGO token wrapping. The BouncySpnegoTokenGenerator implementation is provided as an unsupported contribution from the contrib package. This requires the BouncyCastle libraries "http://www.bouncycastle.org/java.html" [http://www.bouncycastle.org/java.html]. Found especially useful when using Java 1.5, which is known to provide only a limited support for SPNEGO authentication.

# Chapter 5. HTTP client service

## 5.1. HttpClient facade

`HttpClient` interface represents the most essential contract for HTTP request execution. It imposes no restrictions or particular details on the request execution process and leaves the specifics of connection management, state management, authentication and redirect handling up to individual implementations. This should make it easier to decorate the interface with additional functionality such as response content caching.

`DefaultHttpClient` is the default implementation of the `HttpClient` interface. This class acts as a facade to a number of special purpose handler or strategy interface implementations responsible for handling of a particular aspect of the HTTP protocol such as redirect or authentication handling or making decision about connection persistence and keep alive duration. This enables the users to selectively replace default implementation of those aspects with custom, application specific ones.

```
DefaultHttpClient httpclient = new DefaultHttpClient();

httpclient.setKeepAliveStrategy(new DefaultConnectionKeepAliveStrategy() {

    @Override
    public long getKeepAliveDuration(
            HttpResponse response,
            HttpContext context) {
        long keepAlive = super.getKeepAliveDuration(response, context);
        if (keepAlive == -1) {
            // Keep connections alive 5 seconds if a keep-alive value
            // has not be explicitly set by the server
            keepAlive = 5000;
        }
        return keepAlive;
    }

});
```

`DefaultHttpClient` also maintains a list of protocol interceptors intended for processing outgoing requests and incoming responses and provides methods for managing those interceptors. New protocol interceptors can be introduced to the protocol processor chain or removed from it if needed. Internally protocol interceptors are stored in a simple `java.util.ArrayList`. They are executed in the same natural order as they are added to the list.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.removeRequestInterceptorByClass(RequestUserAgent.class);
httpclient.addRequestInterceptor(new HttpRequestInterceptor() {

    public void process(
            HttpRequest request, HttpContext context)
            throws HttpException, IOException {
        request.setHeader(HTTP.USER_AGENT, "My-own-client");
    }

});
```

`DefaultHttpClient` is thread safe. It is recommended that the same instance of this class is reused for multiple request executions. When an instance of `DefaultHttpClient` is no longer needed and is

about to go out of scope the connection manager associated with it must be shut down by calling the `ClientConnectionManager#shutdown()` method.

```
HttpClient httpclient = new DefaultHttpClient();
// Do something useful
httpclient.getConnectionManager().shutdown();
```

## 5.2. HttpClient parameters

These are parameters that be used to customize the behaviour of the default HttpClient implementation:

- **`ClientPNames.HANDLE_REDIRECTS`='http.protocol.handle-redirects':** defines whether redirects should be handled automatically. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set HttpClient will handle redirects automatically.

- **`ClientPNames.REJECT_RELATIVE_REDIRECT`='http.protocol.reject-relative-redirect':** defines whether relative redirects should be rejected. HTTP specification requires the location value be an absolute URI. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set relative redirects will be allowed.

- **`ClientPNames.MAX_REDIRECTS`='http.protocol.max-redirects':** defines the maximum number of redirects to be followed. The limit on number of redirects is intended to prevent infinite loops caused by broken server side scripts. This parameter expects a value of type `java.lang.Integer`. If this parameter is not set no more than 100 redirects will be allowed.

- **`ClientPNames.ALLOW_CIRCULAR_REDIRECTS`='http.protocol.allow-circular-redirects':** defines whether circular redirects (redirects to the same location) should be allowed. The HTTP spec is not sufficiently clear whether circular redirects are permitted, therefore optionally they can be enabled. This parameter expects a value of type `java.lang.Boolean`. If this parameter is not set circular redirects will be disallowed.

- **`ClientPNames.CONNECTION_MANAGER_FACTORY_CLASS_NAME`='http.connection-manager.factory-class-name':** defines the class name of the default `ClientConnectionManager` implementation. This parameter expects a value of type `java.lang.String`. If this parameter is not set `SingleClientConnManager` will be used per default.

- **`ClientPNames.VIRTUAL_HOST`='http.virtual-host':** defines the virtual host name to be used in the `Host` header instead of the physical host name. This parameter expects a value of type `HttpHost`. If this parameter is not set name or IP address of the target host will be used.

- **`ClientPNames.DEFAULT_HEADERS`='http.default-headers':** defines the request headers to be sent per default with each request. This parameter expects a value of type `java.util.Collection` containing `Header` objects.

- **`ClientPNames.DEFAULT_HOST`='http.default-host':** defines the default host. The default value will be used if the target host is not explicitly specified in the request URI (relative URIs). This parameter expects a value of type `HttpHost`.

## 5.3. Automatic redirect handling

HttpClient handles all types of redirects automatically, except those explicitly prohibited by the HTTP specification as requiring user intervention. `See Other` (status code 303) redirects on `POST` and `PUT` requests are converted to `GET` requests as required by the HTTP specification.

## 5.4. HTTP client and execution context

The `DefaultHttpClient` treats HTTP requests as immutable objects that are never supposed to change in the course of request execution. Instead, it creates a private mutable copy of the original request object, whose properties can be updated depending on the execution context. Therefore the final request properties such as the target host and request URI can be determined by examining the content of the local HTTP context after the request has been executed.

The final HttpRequest object in the execution context always represents the state of the message _exactly_ as it was sent to the target server. Per default HTTP/1.0 and HTTP/1.1 use relative request URIs. However if the request is sent via a proxy in a non-tunneling mode then the URI will be absolute.

```
DefaultHttpClient httpclient = new DefaultHttpClient();

HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext);
HttpHost target = (HttpHost) localContext.getAttribute(
        ExecutionContext.HTTP_TARGET_HOST);
HttpUriRequest req = (HttpUriRequest) localContext.getAttribute(
        ExecutionContext.HTTP_REQUEST);

System.out.println("Target host: " + target);
System.out.println("Final request URI: " + req.getURI()); // relative URI (no proxy used)
System.out.println("Final request method: " + req.getMethod());
```

## 5.5. Compressed response content

The `ContentEncodingHttpClient` is a simple sub-class of `DefaultHttpClient` which adds support indicating to servers that it will support `gzip` and `deflate` compressed responses. It does this via the existing published APIs of HTTP Protocol Interceptors . Depending on the type of response (text will compress well versus images, which are typically already well-compressed), this can speed up responses due to the smaller amount of network traffic involved, along with saving bandwidth, which can be important in mobile environments. The `RequestAcceptEncoding` and `ResponseContentEncoding` interceptors used as also part of the published API and can be used by other `DefaultHttpClient` implementations. These provide transparent handling of `gzip` and `deflate` encoding, so it will not be apparent to clients that this processing has happened.

```
ContentEncodingHttpClient httpclient = new ContentEncodingHttpClient();
HttpGet httpget = new HttpGet("http://www.yahoo.com/");
HttpResponse response = httpclient.execute(httpget);

Header h = rsp.getFirstHeader("Content-Encoding");
if (h != null) {
    System.out.println("----------------------------------------");
    System.out.println("Response is " + h.getValue() + " encoded");
    System.out.println("----------------------------------------");
}
```

One can also add the `RequestAcceptEncoding` and `ResponseContentEncoding` interceptors to an instance of the `DefaultHttpClient`, if desired.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.addRequestInterceptor(new RequestAcceptEncoding());
httpclient.addResponseInterceptor(new ResponseContentEncoding());
```

# Chapter 6. HTTP Caching

## 6.1. General Concepts

HttpClient Cache provides an HTTP/1.1-compliant caching layer to be used with HttpClient--the Java equivalent of a browser cache. The implementation follows the Decorator design pattern, where the CachingHttpClient class is a drop-in replacement for a DefaultHttpClient; requests that can be satisfied entirely from the cache will not result in actual origin requests. Stale cache entries are automatically validated with the origin where possible, using conditional GETs and the If-Modified-Since and/or If-None-Match request headers.

HTTP/1.1 caching in general is designed to be *semantically transparent*; that is, a cache should not change the meaning of the request-response exchange between client and server. As such, it should be safe to drop a CachingHttpClient into an existing compliant client-server relationship. Although the caching module is part of the client from an HTTP protocol point of view, the implementation aims to be compatible with the requirements placed on a transparent caching proxy.

Finally, CachingHttpClient includes support the Cache-Control extensions specified by RFC 5861 (stale-if-error and stale-while-revalidate).

When CachingHttpClient executes a request, it goes through the following flow:

1. Check the request for basic compliance with the HTTP 1.1 protocol and attempt to correct the request.

2. Flush any cache entries which would be invalidated by this request.

3. Determine if the current request would be servable from cache. If not, directly pass through the request to the origin server and return the response, after caching it if appropriate.

4. If it was a a cache-servable request, it will attempt to read it from the cache. If it is not in the cache, call the origin server and cache the response, if appropriate.

5. If the cached response is suitable to be served as a response, construct a BasicHttpResponse containing a ByteArrayEntity and return it. Otherwise, attempt to revalidate the cache entry against the origin server.

6. In the case of a cached response which cannot be revalidated, call the origin server and cache the response, if appropriate.

When CachingHttpClient receives a response, it goes through the following flow:

1. Examining the response for protocol compliance

2. Determine whether the response is cacheable

3. If it is cacheable, attempt to read up to the maximum size allowed in the configuration and store it in the cache.

4. If the response is too large for the cache, reconstruct the partially consumed response and return it directly without caching it.

It is important to note that CachingHttpClient is not, itself, an implementation of HttpClient, but that it decorates an instance of an HttpClient implementation. If you do not provide an implementation, it will use DefaultHttpClient internally by default.

## 6.2. RFC-2616 Compliance

HttpClient Cache makes an effort to be at least *conditionally compliant* with RFC-2616 [http://www.ietf.org/rfc/rfc2616.txt]. That is, wherever the specification indicates MUST or MUST NOT for HTTP caches, the caching layer attempts to behave in a way that satisfies those requirements. This means the caching module won't produce incorrect behavior when you drop it in. At the same time, the project is continuing to work on unconditional compliance, which would add compliance with all the SHOULDs and SHOULD NOTs, many of which we already comply with. We just can't claim fully unconditional compliance until we satisfy *all* of them.

## 6.3. Example Usage

This is a simple example of how to set up a basic CachingHttpClient. As configured, it will store a maximum of 1000 cached objects, each of which may have a maximum body size of 8192 bytes. The numbers selected here are for example only and not intended to be prescriptive or considered as recommendations.

```
CacheConfig cacheConfig = new CacheConfig();
cacheConfig.setMaxCacheEntries(1000);
cacheConfig.setMaxObjectSizeBytes(8192);

HttpClient cachingClient = new CachingHttpClient(new DefaultHttpClient(), cacheConfig);

HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://www.mydomain.com/content/");
HttpResponse response = cachingClient.execute(httpget, localContext);
HttpEntity entity = response.getEntity();
EntityUtils.consume(entity);
CacheResponseStatus responseStatus = (CacheResponseStatus) localContext.getAttribute(
        CachingHttpClient.CACHE_RESPONSE_STATUS);
switch (responseStatus) {
case CACHE_HIT:
    System.out.println("A response was generated from the cache with no requests " +
            "sent upstream");
    break;
case CACHE_MODULE_RESPONSE:
    System.out.println("The response was generated directly by the caching module");
    break;
case CACHE_MISS:
    System.out.println("The response came from an upstream server");
    break;
case VALIDATED:
    System.out.println("The response was generated from the cache after validating " +
            "the entry with the origin server");
    break;
}
```

## 6.4. Configuration

As the CachingHttpClient is a decorator, much of the configuration you may want to do can be done on the HttpClient used as the "backend" by the HttpClient (this includes setting options like timeouts and

connection pool sizes). For caching-specific configuration, you can provide a CacheConfig instance to customize behavior across the following areas:

*Cache size.* If the backend storage supports these limits, you can specify the maximum number of cache entries as well as the maximum cacheable response body size.

*Public/private caching.* By default, the caching module considers itself to be a shared (public) cache, and will not, for example, cache responses to requests with Authorization headers or responses marked with "Cache-Control: private". If, however, the cache is only going to be used by one logical "user" (behaving similarly to a browser cache), then you will want to turn off the shared cache setting.

*Heuristic caching.*Per RFC2616, a cache MAY cache certain cache entries even if no explicit cache control headers are set by the origin. This behavior is off by default, but you may want to turn this on if you are working with an origin that doesn't set proper headers but where you still want to cache the responses. You will want to enable heuristic caching, then specify either a default freshness lifetime and/or a fraction of the time since the resource was last modified. See Sections 13.2.2 and 13.2.4 of the HTTP/1.1 RFC for more details on heuristic caching.

*Background validation.* The cache module supports the stale-while-revalidate directive of RFC5861, which allows certain cache entry revalidations to happen in the background. You may want to tweak the settings for the minimum and maximum number of background worker threads, as well as the maximum time they can be idle before being reclaimed. You can also control the size of the queue used for revalidations when there aren't enough workers to keep up with demand.

## 6.5. Storage Backends

The default implementation of CachingHttpClient stores cache entries and cached response bodies in memory in the JVM of your application. While this offers high performance, it may not be appropriate for your application due to the limitation on size or because the cache entries are ephemeral and don't survive an application restart. The current release includes support for storing cache entries using Ehcache and memcached implementations, which allow for spilling cache entries to disk or storing them in an external process.

If none of those options are suitable for your application, it is possible to provide your own storage backend by implementing the HttpCacheStorage interface and then supplying that to CachingHttpClient at construction time. In this case, the cache entries will be stored using your scheme but you will get to reuse all of the logic surrounding HTTP/1.1 compliance and cache handling. Generally speaking, it should be possible to create an HttpCacheStorage implementation out of anything that supports a key/value store (similar to the Java Map interface) with the ability to apply atomic updates.

Finally, because the CachingHttpClient is a decorator for HttpClient, it's entirely possible to set up a multi-tier caching hierarchy; for example, wrapping an in-memory CachingHttpClient around one that stores cache entries on disk or remotely in memcached, following a pattern similar to virtual memory, L1/L2 processor caches, etc.

# Chapter 7. Advanced topics

## 7.1. Custom client connections

In certain situations it may be necessary to customize the way HTTP messages get transmitted across the wire beyond what is possible using HTTP parameters in order to be able to deal non-standard, non-compliant behaviours. For instance, for web crawlers it may be necessary to force HttpClient into accepting malformed response heads in order to salvage the content of the messages.

Usually the process of plugging in a custom message parser or a custom connection implementation involves several steps:

- Provide a custom `LineParser` / `LineFormatter` interface implementation. Implement message parsing / formatting logic as required.

```
class MyLineParser extends BasicLineParser {

    @Override
    public Header parseHeader(
            final CharArrayBuffer buffer) throws ParseException {
        try {
            return super.parseHeader(buffer);
        } catch (ParseException ex) {
            // Suppress ParseException exception
            return new BasicHeader("invalid", buffer.toString());
        }
    }

}
```

- Provide a custom `OperatedClientConnection` implementation. Replace default request / response parsers, request / response formatters with custom ones as required. Implement different message writing / reading code if necessary.

```
class MyClientConnection extends DefaultClientConnection {

    @Override
    protected HttpMessageParser createResponseParser(
            final SessionInputBuffer buffer,
            final HttpResponseFactory responseFactory,
            final HttpParams params) {
        return new DefaultResponseParser(
                buffer,
                new MyLineParser(),
                responseFactory,
                params);
    }

}
```

- Provide a custom `ClientConnectionOperator` interface implementation in order to create connections of new class. Implement different socket initialization code if necessary.

```
class MyClientConnectionOperator extends DefaultClientConnectionOperator {
```

```
    public MyClientConnectionOperator(final SchemeRegistry sr) {
        super(sr);
    }

    @Override
    public OperatedClientConnection createConnection() {
        return new MyClientConnection();
    }

}
```

- Provide a custom `ClientConnectionManager` interface implementation in order to create connection operator of new class.

```
class MyClientConnManager extends SingleClientConnManager {

    public MyClientConnManager(
            final HttpParams params,
            final SchemeRegistry sr) {
        super(params, sr);
    }

    @Override
    protected ClientConnectionOperator createConnectionOperator(
            final SchemeRegistry sr) {
        return new MyClientConnectionOperator(sr);
    }

}
```

## 7.2. Stateful HTTP connections

While HTTP specification assumes that session state information is always embedded in HTTP messages in the form of HTTP cookies and therefore HTTP connections are always stateless, this assumption does not always hold true in real life. There are cases when HTTP connections are created with a particular user identity or within a particular security context and therefore cannot be shared with other users and can be reused by the same user only. Examples of such stateful HTTP connections are `NTLM` authenticated connections and SSL connections with client certificate authentication.

### 7.2.1. User token handler

HttpClient relies on `UserTokenHandler` interface to determine if the given execution context is user specific or not. The token object returned by this handler is expected to uniquely identify the current user if the context is user specific or to be null if the context does not contain any resources or details specific to the current user. The user token will be used to ensure that user specific resources will not be shared with or reused by other users.

The default implementation of the `UserTokenHandler` interface uses an instance of Principal class to represent a state object for HTTP connections, if it can be obtained from the given execution context. `DefaultUserTokenHandler` will use the user principle of connection based authentication schemes such as `NTLM` or that of the SSL session with client authentication turned on. If both are unavailable, null token will be returned.

Users can provide a custom implementation if the default one does not satisfy their needs:

```
DefaultHttpClient httpclient = new DefaultHttpClient();
```

```
httpclient.setUserTokenHandler(new UserTokenHandler() {

    public Object getUserToken(HttpContext context) {
        return context.getAttribute("my-token");
    }

});
```

## 7.2.2. User token and execution context

In the course of HTTP request execution HttpClient adds the following user identity related objects to the execution context:

- **`ClientContext.USER_TOKEN`='http.user-token':** Object instance representing the actual user identity, usually expected to be an instance of `Principle` interface

One can find out whether or not the connection used to execute the request was stateful by examining the content of the local HTTP context after the request has been executed.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext);
HttpEntity entity = response.getEntity();
EntityUtils.consume(entity);
Object userToken = localContext.getAttribute(ClientContext.USER_TOKEN);
System.out.println(userToken);
```

### 7.2.2.1. Persistent stateful connections

Please note that a persistent connection that carries a state object can be reused only if the same state object is bound to the execution context when requests are executed. So, it is really important to ensure the either same context is reused for execution of subsequent HTTP requests by the same user or the user token is bound to the context prior to request execution.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext1 = new BasicHttpContext();
HttpGet httpget1 = new HttpGet("http://localhost:8080/");
HttpResponse response1 = httpclient.execute(httpget1, localContext1);
HttpEntity entity1 = response1.getEntity();
EntityUtils.consume(entity1);
Principal principal = (Principal) localContext1.getAttribute(
        ClientContext.USER_TOKEN);

HttpContext localContext2 = new BasicHttpContext();
localContext2.setAttribute(ClientContext.USER_TOKEN, principal);
HttpGet httpget2 = new HttpGet("http://localhost:8080/");
HttpResponse response2 = httpclient.execute(httpget2, localContext2);
HttpEntity entity2 = response2.getEntity();
EntityUtils.consume(entity2);
```