HttpClient Tutorial

Oleg Kalnichevski
Jonathan Moore
Jilles van Gurp

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 process content, execute javascript embedded in HTML pages, try to guess content type, if not explicitly set, or reformat request / rewrite 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:

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
    <...>
} finally {
    response.close();
}
```

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 `URIBuilder` utility class to simplify creation and modification of request URIs.

```
URI uri = new URIBuilder()
        .setScheme("http")
        .setHost("www.google.com")
        .setPath("/search")
        .setParameter("q", "httpclient")
        .setParameter("btnG", "Google Search")
        .setParameter("aq", "f")
        .setParameter("oq", "")
        .build();
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",
   ContentType.create("text/plain", "UTF-8"));

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

stdout >

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

1.1.5. Ensuring release of low level resources

In order to ensure proper release of system resources one must close either the content stream associated with the entity or the response itself

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        InputStream instream = entity.getContent();
        try {
            // do something useful
        } finally {
            instream.close();
        }
    }
} finally {
    response.close();
}
```

The difference between closing the content stream and closing the response is that the former will attempt to keep the underlying connection alive by consuming the entity content while the latter immediately shuts down and discards the connection.

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 terminate the content stream by closing the response.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        InputStream instream = entity.getContent();
        int byteOne = instream.read();
        int byteTwo = instream.read();
        // Do not need the rest
    }
} finally {
    response.close();
}
```

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.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/");
CloseableHttpResponse response = httpclient.execute(httpget);
try {
    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
        }
    }
} finally {
    response.close();
}
```

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.

```
CloseableHttpResponse response = <...>
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 throught 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,
    ContentType.create("text/plain", "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. 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, Consts.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
```

1.1.7.2. 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",
        ContentType.create("plain/text", Consts.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.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpGet httpget = new HttpGet("http://localhost/json");

ResponseHandler<MyJsonObject> rh = new ResponseHandler<MyJsonObject>() {

    @Override
    public JsonObject handleResponse(
            final HttpResponse response) throws IOException {
        StatusLine statusLine = response.getStatusLine();
```

```
        HttpEntity entity = response.getEntity();
        if (statusLine.getStatusCode() >= 300) {
            throw new HttpResponseException(
                    statusLine.getStatusCode(),
                    statusLine.getReasonPhrase());
        }
        if (entity == null) {
            throw new ClientProtocolException("Response contains no content");
        }
        Gson gson = new GsonBuilder().create();
        ContentType contentType = ContentType.getOrDefault(entity);
        Charset charset = contentType.getCharset();
        Reader reader = new InputStreamReader(entity.getContent(), charset);
        return gson.fromJson(reader, MyJsonObject.class);
    }
};
MyJsonObject myjson = client.execute(httpget, rh);
```

1.2. HttpClient interface

`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.

Generally `HttpClient` implementations act 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.

```
ConnectionKeepAliveStrategy keepAliveStrat = 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;
    }

};
CloseableHttpClient httpclient = HttpClients.custom()
        .setKeepAliveStrategy(keepAliveStrat)
        .build();
```

1.2.1. HttpClient thread safety

`HttpClient` implementations are expected to be thread safe. It is recommended that the same instance of this class is reused for multiple request executions.

1.2.2. HttpClient resource deallocation

When an instance `CloseableHttpClient` 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 `CloseableHttpClient#close()` method.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
try {
    <...>
} finally {
    httpclient.close();
}
```

1.3. 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.

`HttpContext` can contain arbitrary objects and therefore may be unsafe to share between multiple threads. It is recommended that each thread of execution maintains its own context.

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

- `HttpConnection` instance representing the actual connection to the target server.
- `HttpHost` instance representing the connection target.
- `HttpRoute` instance representing the complete connection route
- `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.
- `HttpResponse` instance representing the actual HTTP response.
- `java.lang.Boolean` object representing the flag indicating whether the actual request has been fully transmitted to the connection target.
- `RequestConfig` object representing the actual request configuation.
- `java.util.List<URI>` object representing a collection of all redirect locations received in the process of request execution.

One can use `HttpClientContext` adaptor class to simplify interractions with the context state.

```
```

```
HttpContext context = <...>
HttpClientContext clientContext = HttpClientContext.adapt(context);
HttpHost target = clientContext.getTargetHost();
HttpRequest request = clientContext.getRequest();
HttpResponse response = clientContext.getResponse();
RequestConfig config = clientContext.getRequestConfig();
```

Multiple request sequences that represent a logically related session should be executed with the same `HttpContext` instance to ensure automatic propagation of conversation context and state information between requests.

In the following example the request configuration set by the initial request will be kept in the execution context and get propagated to the consecutive requests sharing the same context.

```

CloseableHttpClient httpclient = HttpClients.createDefault();
RequestConfig requestConfig = RequestConfig.custom()
        .setSocketTimeout(1000)
        .setConnectTimeout(1000)
        .build();

HttpGet httpget1 = new HttpGet("http://localhost/1");
httpget1.setConfig(requestConfig);
CloseableHttpResponse response1 = httpclient.execute(httpget1, context);
try {
    HttpEntity entity1 = response1.getEntity();
} finally {
    response1.close();
}
HttpGet httpget2 = new HttpGet("http://localhost/2");
CloseableHttpResponse response2 = httpclient.execute(httpget2, context);
try {
    HttpEntity entity2 = response2.getEntity();
} finally {
    response2.close();
}
```

1.4. HTTP protocol interceptors

The 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:

```
CloseableHttpClient httpclient = HttpClients.custom()
        .addInterceptorLast(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()));
            }

        })
        .build();

AtomicInteger count = new AtomicInteger(1);
HttpClientContext localContext = HttpClientContext.create();
localContext.setAttribute("count", count);

HttpGet httpget = new HttpGet("http://localhost/");
for (int i = 0; i < 10; i++) {
    CloseableHttpResponse response = httpclient.execute(httpget, localContext);
    try {
        HttpEntity entity = response.getEntity();
    } finally {
        response.close();
    }
}
```

1.5. Exception handling

HTTP protocol processors 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. Please note that `HttpClient` implementations re-throw `HttpExceptions` as `ClientProtocolException`, which is a subclass of `java.io.IOException`. This enables the users of `HttpClient` to handle both I/O errors and protocol violations from a single catch clause.

1.5.1. 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.

1.5.2. 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.

By default HttpClient assumes only non-entity enclosing methods such as `GET` and `HEAD` to be idempotent and entity enclosing methods such as `POST` and `PUT` to be not for compatibility reasons.

1.5.3. 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).

1.5.4. Request retry handler

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

```
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 InterruptedIOException) {
            // Timeout
            return false;
        }
        if (exception instanceof UnknownHostException) {
            // Unknown host
            return false;
        }
        if (exception instanceof ConnectTimeoutException) {
            // Connection refused
            return false;
```

```
        }
        if (exception instanceof SSLException) {
            // SSL handshake exception
            return false;
        }
        HttpClientContext clientContext = HttpClientContext.adapt(context);
        HttpRequest request = clientContext.getRequest();
        boolean idempotent = !(request instanceof HttpEntityEnclosingRequest);
        if (idempotent) {
            // Retry if the request is considered idempotent
            return true;
        }
        return false;
    }

};
CloseableHttpClient httpclient = HttpClients.custom()
        .setRetryHandler(myRetryHandler)
        .build();
```

Please note that one can use `StandardHttpRequestRetryHandler` instead of the one used by default in order to treat those request methods defined as idempotent by RFC-2616 as safe to retry automatically: `GET`, `HEAD`, `PUT`, `DELETE`, `OPTIONS`, and `TRACE`.

1.6. 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`

1.7. 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. One can use a custom redirect strategy to relaxe restrictions on automatic redirection of POST methods imposed by the HTTP specification.

```
LaxRedirectStrategy redirectStrategy = new LaxRedirectStrategy();
CloseableHttpClient httpclient = HttpClients.custom()
        .setRedirectStrategy(redirectStrategy)
        .build();
```

HttpClient often has to rewrite the request message in the process of its execution. Per default HTTP/1.0 and HTTP/1.1 generally use relative request URIs. Likewise, original request may get redirected from location to another multiple times. The final interpreted absolute HTTP location can be built using the original request and the context. The utility method `URIUtils#resolve` can be used to build the interpreted absolute URI used to generate the final request. This method includes the last fragment identifier from the redirect requests or the original request.

```
```

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpClientContext context = HttpClientContext.create();
HttpGet httpget = new HttpGet("http://localhost:8080/");
CloseableHttpResponse response = httpclient.execute(httpget, context);
try {
    HttpHost target = context.getTargetHost();
    List<URI> redirectLocations = context.getRedirectLocations();
    URI location = URIUtils.resolve(httpget.getURI(), target, redirectLocations);
    System.out.println("Final HTTP location: " + location.toASCIIString());
    // Expected to be an absolute URI
} finally {
    response.close();
}
```

Chapter 2. Connection management

2.1. 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.

2.2. 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.

2.2.1. 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. `SystemDefaultRoutePlanner` 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 `DefaultProxyRoutePlanner` implementation does not make use of any Java system properties, nor any system or browser proxy settings. It always computes routes via the same default proxy.

2.2.2. 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.

2.3. HTTP connection managers

2.3.1. 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 `HttpClientConnectionManager` interface. The purpose of an HTTP connection manager is to serve as a factory for new HTTP connections, to manage life cycle of persistent connections and to 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 `ManagedHttpClientConnection` acting as a proxy for a real connection that manages connection state and controls execution of I/O operations. If a managed connection is released or get explicitly closed by its consumer the underlying connection gets detached from its proxy and is returned back to the manager. Even though the service consumer still holds a reference to the proxy instance, it is no longer able to execute any I/O operations or change the state of the real connection either intentionally or unintentionally.

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

```
HttpClientContext context = HttpClientContext.create();
HttpClientConnectionManager connMrg = new BasicHttpClientConnectionManager();
HttpRoute route = new HttpRoute(new HttpHost("localhost", 80));
// Request new connection. This can be a long process
ConnectionRequest connRequest = connMrg.requestConnection(route, null);
// Wait for connection up to 10 sec
HttpClientConnection conn = connRequest.get(10, TimeUnit.SECONDS);
try {
    // If not open
    if (!conn.isOpen()) {
        // establish connection based on its route info
        connMrg.connect(conn, route, 1000, context);
        // and mark it as route complete
        connMrg.routeComplete(conn, route, context);
    }
    // Do useful things with the connection.
} finally {
    connMrg.releaseConnection(conn, null, 1, TimeUnit.MINUTES);
}
```

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

2.3.2. Simple connection manager

`BasicHttpClientConnectionManager` 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. `BasicHttpClientConnectionManager` 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.

This connection manager implementation should be used inside an EJB container.

2.3.3. Pooling connection manager

`PoolingHttpClientConnectionManager` 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.

`PoolingHttpClientConnectionManager` 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.

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

```
PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();
// Increase max total connection to 200
cm.setMaxTotal(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.setMaxPerRoute(new HttpRoute(localhost), 50);

CloseableHttpClient httpClient = HttpClients.custom()
        .setConnectionManager(cm)
        .build();
```

2.3.4. 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.

```
CloseableHttpClient httpClient = <...>
httpClient.close();
```

2.4. Multithreaded request execution

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

The `PoolingClientConnectionManager` 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.

```
PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager();
```

```
CloseableHttpClient httpClient = HttpClients.custom()
        .setConnectionManager(cm)
        .build();

// 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();
}
```

While `HttpClient` instances are thread safe and can be shared between multiple threads of execution, it is highly recommended that each thread maintains its own dedicated instance of `HttpContext` .

```
static class GetThread extends Thread {

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

    public GetThread(CloseableHttpClient httpClient, HttpGet httpget) {
        this.httpClient = httpClient;
        this.context = HttpClientContext.create();
        this.httpget = httpget;
    }

    @Override
    public void run() {
        try {
            CloseableHttpResponse response = httpClient.execute(
                    httpget, context);
            try {
                HttpEntity entity = response.getEntity();
            } finally {
                response.close();
            }
        } catch (ClientProtocolException ex) {
            // Handle protocol errors
        } catch (IOException ex) {
            // Handle I/O errors
        }
    }

}
```

2.5. 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. 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 HttpClientConnectionManager connMgr;
    private volatile boolean shutdown;

    public IdleConnectionMonitorThread(HttpClientConnectionManager 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();
        }
    }

}
```

2.6. 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.

```
ConnectionKeepAliveStrategy myStrategy = 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(
                HttpClientContext.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;
        }
    }

};
CloseableHttpClient client = HttpClients.custom()
        .setKeepAliveStrategy(myStrategy)
        .build();
```

2.7. Connection 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 `ConnectionSocketFactory` interface to create, initialize and connect sockets. This enables the users of HttpClient to provide application specific socket initialization code at runtime. `PlainConnectionSocketFactory` 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.

```
HttpClientContext clientContext = HttpClientContext.create();
PlainConnectionSocketFactory sf = PlainConnectionSocketFactory.getSocketFactory();
```

```
Socket socket = sf.createSocket(clientContext);
int timeout = 1000; //ms
HttpHost target = new HttpHost("localhost");
InetSocketAddress remoteAddress = new InetSocketAddress(
        InetAddress.getByAddress(new byte[] {127,0,0,1}), 80);
sf.connectSocket(timeout, socket, target, remoteAddress, null, clientContext);
```

2.7.1. Secure socket layering

`LayeredConnectionSocketFactory` is an extension of the `ConnectionSocketFactory` 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.

2.7.2. Integration with connection manager

Custom connection socket factories can be associated with a particular protocol scheme as as HTTP or HTTPS and then used to create a custom connection manager.

```
ConnectionSocketFactory plainsf = <...>
LayeredConnectionSocketFactory sslsf = <...>
Registry<ConnectionSocketFactory> r = RegistryBuilder.<ConnectionSocketFactory>create()
        .register("http", plainsf)
        .register("https", sslsf)
        .build();

HttpClientConnectionManager cm = new PoolingHttpClientConnectionManager(r);
HttpClients.custom()
        .setConnectionManager(cm)
        .build();
```

2.7.3. SSL/TLS customization

HttpClient makes use of `SSLConnectionSocketFactory` to create SSL connections. `SSLConnectionSocketFactory` 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.

```
KeyStore myTrustStore = <...>
SSLContext sslContext = SSLContexts.custom()
        .loadTrustMaterial(myTrustStore)
        .build();
SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(sslContext);
```

Customization of `SSLConnectionSocketFactory` 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 (JSSE) Reference Guide [http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html] for a detailed description of `javax.net.ssl.SSLContext` and related tools.

2.7.4. 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 `javax.net.ssl.HostnameVerifier` interface represents a strategy for hostname verification. HttpClient ships with two `javax.net.ssl.HostnameVerifier` implementations. Important: hostname verification should not be confused with SSL trust verification.

- **`DefaultHostnameVerifier:`** The default implementation used by HttpClient is expected to be compliant with RFC 2818. The hostname must match any of alternative names specified by the certificate, or in case no alternative names are given the most specific CN of the certificate subject. A wildcard can occur in the CN, and in any of the subject-alts.

- **`NoopHostnameVerifier:`** This hostname verifier essentially turns hostname verification off. It accepts any SSL session as valid and matching the target host.

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

```
SSLContext sslContext = SSLContexts.createSystemDefault();
SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(
        sslContext,
        NoopHostnameVerifier.INSTANCE);
```

As of version 4.4 HttpClient uses the public suffix list kindly maintained by Mozilla Foundation to make sure that wildcards in SSL certificates cannot be misused to apply to multiple domains with a common top-level domain. HttpClient ships with a copy of the list retrieved at the time of the release. The latest revision of the list can found at https://publicsuffix.org/list/ [https://publicsuffix.org/list/effective_tld_names.dat]. It is highly adviseable to make a local copy of the list and download the list no more than once per day from its original location.

```
PublicSuffixMatcher publicSuffixMatcher = PublicSuffixMatcherLoader.load(
    PublicSuffixMatcher.class.getResource("my-copy-effective_tld_names.dat"));
DefaultHostnameVerifier hostnameVerifier = new DefaultHostnameVerifier(publicSuffixMatcher);
```

One can disable verification against the public suffic list by using `null` matcher.

```
DefaultHostnameVerifier hostnameVerifier = new DefaultHostnameVerifier(null);
```

2.8. HttpClient proxy configuration

Even though HttpClient is aware of complex routing schemes 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:

```
HttpHost proxy = new HttpHost("someproxy", 8080);
DefaultProxyRoutePlanner routePlanner = new DefaultProxyRoutePlanner(proxy);
CloseableHttpClient httpclient = HttpClients.custom()
        .setRoutePlanner(routePlanner)
        .build();
```

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

```
SystemDefaultRoutePlanner routePlanner = new SystemDefaultRoutePlanner(
        ProxySelector.getDefault());
CloseableHttpClient httpclient = HttpClients.custom()
        .setRoutePlanner(routePlanner)
        .build();
```

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

```
HttpRoutePlanner routePlanner = 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()));
    }

};
CloseableHttpClient httpclient = HttpClients.custom()
        .setRoutePlanner(routePlanner)
        .build();
    }
}
```

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 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 `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` header.

Here is an example of creating a client-side cookie object:

```
BasicClientCookie cookie = new BasicClientCookie("name", "value");
// Set effective domain and path attributes
cookie.setDomain(".mycompany.com");
cookie.setPath("/");
// Set attributes exactly as sent by the server
cookie.setAttribute(ClientCookie.PATH_ATTR, "/");
cookie.setAttribute(ClientCookie.DOMAIN_ATTR, ".mycompany.com");
```

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` 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:

- **Standard strict:** State management policy compliant with the syntax and semantics of the well-behaved profile defined by RFC 6265, section 4.
- **Standard:** State management policy compliant with a more relaxed profile defined by RFC 6265, section 4 intended for interoperability with existing servers that do not conform to the well behaved profile.
- **Netscape draft (obsolete):** This policy conforms to the original draft specification published by Netscape Communications. It should be avoided unless absolutely necessary for compatibility with legacy code.
- **RFC 2965 (obsolete):** State management policy compliant with the obsolete state management specification defined by RFC 2965. Please do not use in new applications.
- **RFC 2109 (obsolete):** State management policy compliant with the obsolete state management specification defined by RFC 2109. Please do not use in new applications.
- **Browser compatibility (obsolete):** This policy strives to closely mimic the (mis)behavior of older versions of browser applications such as Microsoft Internet Explorer and Mozilla FireFox. Please do not use in new applications.
- **Default:** Default cookie policy is a synthetic policy that picks up either RFC 2965, RFC 2109 or Netscape draft compliant implementation based on properties of cookies sent with the HTTP response (such as version attribute, now obsolete). This policy will be deprecated in favor of the standard (RFC 6265 compliant) implementation in the next minor release of HttpClient.
- **Ignore cookies:** All cookies are ignored.

It is strongly recommended to use either `Standard` or `Standard strict` policy in new applications. Obsolete specifications should be used for compatibility with legacy systems only. Support for obsolete specifications will be removed in the next major release of HttpClient.

3.3. Choosing cookie policy

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

```
RequestConfig globalConfig = RequestConfig.custom()
        .setCookieSpec(CookieSpecs.DEFAULT)
        .build();
CloseableHttpClient httpclient = HttpClients.custom()
        .setDefaultRequestConfig(globalConfig)
        .build();
RequestConfig localConfig = RequestConfig.copy(globalConfig)
        .setCookieSpec(CookieSpecs.STANDARD_STRICT)
        .build();
```

```
HttpGet httpGet = new HttpGet("/");
httpGet.setConfig(localConfig);
```

3.4. Custom cookie policy

In order to implement a custom cookie policy one should create a custom implementation of the `CookieSpec` interface, create a `CookieSpecProvider` 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.

```
PublicSuffixMatcher publicSuffixMatcher = PublicSuffixMatcherLoader.getDefault();

Registry<CookieSpecProvider> r = RegistryBuilder.<CookieSpecProvider>create()
        .register(CookieSpecs.DEFAULT,
                new DefaultCookieSpecProvider(publicSuffixMatcher))
        .register(CookieSpecs.STANDARD,
                new RFC6265CookieSpecProvider(publicSuffixMatcher))
        .register("easy", new EasySpecProvider())
        .build();

RequestConfig requestConfig = RequestConfig.custom()
        .setCookieSpec("easy")
        .build();

CloseableHttpClient httpclient = HttpClients.custom()
        .setDefaultCookieSpecRegistry(r)
        .setDefaultRequestConfig(requestConfig)
        .build();
```

3.5. 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.

```
// Create a local instance of cookie store
CookieStore cookieStore = new BasicCookieStore();
// Populate cookies if needed
BasicClientCookie cookie = new BasicClientCookie("name", "value");
cookie.setDomain(".mycompany.com");
cookie.setPath("/");
cookieStore.addCookie(cookie);
// Set the store
CloseableHttpClient httpclient = HttpClients.custom()
        .setDefaultCookieStore(cookieStore)
        .build();
```

3.6. 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:

- `Lookup` instance representing the actual cookie specification registry. The value of this attribute set in the local context takes precedence over the default one.

- `CookieSpec` instance representing the actual cookie specification.
- `CookieOrigin` instance representing the actual details of the origin server.
- `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. One can also use separate execution contexts 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

```
CloseableHttpClient httpclient = <...>

Lookup<CookieSpecProvider> cookieSpecReg = <...>
CookieStore cookieStore = <...>

HttpClientContext context = HttpClientContext.create();
context.setCookieSpecRegistry(cookieSpecReg);
context.setCookieStore(cookieStore);
HttpGet httpget = new HttpGet("http://somehost/");
CloseableHttpResponse response1 = httpclient.execute(httpget, context);
<...>
// Cookie origin details
CookieOrigin cookieOrigin = context.getCookieOrigin();
// Cookie spec used
CookieSpec cookieSpec = context.getCookieSpec();
```

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:** `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.

- **Kerberos:** Kerberos authentication implementation.

4.3. 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]
```

4.4. 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:

- `Lookup` instance representing the actual authentication scheme registry. The value of this attribute set in the local context takes precedence over the default one.
- `CredentialsProvider` instance representing the actual credentials provider. The value of this attribute set in the local context takes precedence over the default one.
- `AuthState` instance representing the actual target authentication state. The value of this attribute set in the local context takes precedence over the default one.
- `AuthState` instance representing the actual proxy authentication state. The value of this attribute set in the local context takes precedence over the default one.
- `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:

```
CloseableHttpClient httpclient = <...>

CredentialsProvider credsProvider = <...>
Lookup<AuthSchemeProvider> authRegistry = <...>
AuthCache authCache = <...>

HttpClientContext context = HttpClientContext.create();
context.setCredentialsProvider(credsProvider);
context.setAuthSchemeRegistry(authRegistry);
context.setAuthCache(authCache);
```

```
HttpGet httpget = new HttpGet("http://somehost/");
CloseableHttpResponse response1 = httpclient.execute(httpget, context);
<...>

AuthState proxyAuthState = context.getProxyAuthState();
System.out.println("Proxy auth state: " + proxyAuthState.getState());
System.out.println("Proxy auth scheme: " + proxyAuthState.getAuthScheme());
System.out.println("Proxy auth credentials: " + proxyAuthState.getCredentials());
AuthState targetAuthState = context.getTargetAuthState();
System.out.println("Target auth state: " + targetAuthState.getState());
System.out.println("Target auth scheme: " + targetAuthState.getAuthScheme());
System.out.println("Target auth credentials: " + targetAuthState.getCredentials());
```

4.5. 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.

4.6. 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.

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

```
CloseableHttpClient httpclient = <...>

HttpHost targetHost = new HttpHost("localhost", 80, "http");
CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.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
HttpClientContext context = HttpClientContext.create();
context.setCredentialsProvider(credsProvider);
context.setAuthCache(authCache);

HttpGet httpget = new HttpGet("/");
for (int i = 0; i < 3; i++) {
    CloseableHttpResponse response = httpclient.execute(
            targetHost, httpget, context);
    try {
        HttpEntity entity = response.getEntity();

    } finally {
        response.close();
    }
```

```
}
```

4.7. 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.

4.7.1. 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`.

```
CloseableHttpClient httpclient = <...>

CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.setCredentials(AuthScope.ANY,
        new NTCredentials("user", "pwd", "myworkstation", "microsoft.com"));

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

// Make sure the same context is used to execute logically related requests
HttpClientContext context = HttpClientContext.create();
context.setCredentialsProvider(credsProvider);

// Execute a cheap method first. This will trigger NTLM authentication
HttpGet httpget = new HttpGet("/ntlm-protected/info");
CloseableHttpResponse response1 = httpclient.execute(target, httpget, context);
try {
    HttpEntity entity1 = response1.getEntity();
} finally {
    response1.close();
}

// 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"));
CloseableHttpResponse response2 = httpclient.execute(target, httppost, context);
try {
    HttpEntity entity2 = response2.getEntity();
} finally {
    response2.close();
}
```

4.8. 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.

4.8.1. 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 `SPNegoScheme` 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.

4.8.2. 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.

4.8.3. `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;
};
```

4.8.4. `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
```

4.8.5. 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
```

Chapter 5. Fluent API

5.1. Easy to use facade API

As of version of 4.2 HttpClient comes with an easy to use facade API based on the concept of a fluent interface. Fluent facade API exposes only the most fundamental functions of HttpClient and is intended for simple use cases that do not require the full flexibility of HttpClient. For instance, fluent facade API relieves the users from having to deal with connection management and resource deallocation.

Here are several examples of HTTP requests executed through the HC fluent API

```
// Execute a GET with timeout settings and return response content as String.
Request.Get("http://somehost/")
        .connectTimeout(1000)
        .socketTimeout(1000)
        .execute().returnContent().asString();
```

```
// Execute a POST with the 'expect-continue' handshake, using HTTP/1.1,
// containing a request body as String and return response content as byte array.
Request.Post("http://somehost/do-stuff")
        .useExpectContinue()
        .version(HttpVersion.HTTP_1_1)
        .bodyString("Important stuff", ContentType.DEFAULT_TEXT)
        .execute().returnContent().asBytes();
```

```
// Execute a POST with a custom header through the proxy containing a request body
// as an HTML form and save the result to the file
Request.Post("http://somehost/some-form")
        .addHeader("X-Custom-header", "stuff")
        .viaProxy(new HttpHost("myproxy", 8080))
        .bodyForm(Form.form().add("username", "vip").add("password", "secret").build())
        .execute().saveContent(new File("result.dump"));
```

One can also use `Executor` directly in order to execute requests in a specific security context whereby authentication details are cached and re-used for subsequent requests.

```
Executor executor = Executor.newInstance()
        .auth(new HttpHost("somehost"), "username", "password")
        .auth(new HttpHost("myproxy", 8080), "username", "password")
        .authPreemptive(new HttpHost("myproxy", 8080));

executor.execute(Request.Get("http://somehost/"))
        .returnContent().asString();

executor.execute(Request.Post("http://somehost/do-stuff")
        .useExpectContinue()
        .bodyString("Important stuff", ContentType.DEFAULT_TEXT))
        .returnContent().asString();
```

5.1.1. Response handling

The fluent facade API generally relieves the users from having to deal with connection management and resource deallocation. In most cases, though, this comes at a price of having to buffer content of response messages in memory. It is highly recommended to use `ResponseHandler` for HTTP response processing in order to avoid having to buffer content in memory.

```
Document result = Request.Get("http://somehost/content")
        .execute().handleResponse(new ResponseHandler<Document>() {

    public Document handleResponse(final HttpResponse response) throws IOException {
        StatusLine statusLine = response.getStatusLine();
        HttpEntity entity = response.getEntity();
        if (statusLine.getStatusCode() >= 300) {
            throw new HttpResponseException(
                    statusLine.getStatusCode(),
                    statusLine.getReasonPhrase());
        }
        if (entity == null) {
            throw new ClientProtocolException("Response contains no content");
        }
        DocumentBuilderFactory dbfac = DocumentBuilderFactory.newInstance();
        try {
            DocumentBuilder docBuilder = dbfac.newDocumentBuilder();
            ContentType contentType = ContentType.getOrDefault(entity);
            if (!contentType.equals(ContentType.APPLICATION_XML)) {
                throw new ClientProtocolException("Unexpected content type:" +
                    contentType);
            }
            String charset = contentType.getCharset();
            if (charset == null) {
                charset = HTTP.DEFAULT_CONTENT_CHARSET;
            }
            return docBuilder.parse(entity.getContent(), charset);
        } catch (ParserConfigurationException ex) {
            throw new IllegalStateException(ex);
        } catch (SAXException ex) {
            throw new ClientProtocolException("Malformed XML document", ex);
        }
    }

    });
```

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 Chain of Responsibility design pattern, where the caching HttpClient implementation can serve a drop-in replacement for the default non-caching HttpClient implementation; 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 caching HttpClient 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, caching HttpClient includes support the Cache-Control extensions specified by RFC 5861 (stale-if-error and stale-while-revalidate).

When caching HttpClient 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 caching HttpClient 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 caching HttpClient is not, itself, a different implementation of HttpClient, but that it works by inserting itself as an additonal processing component to the request execution pipeline.

6.2. RFC-2616 Compliance

We believe HttpClient Cache is *unconditionally compliant* with RFC-2616 [http://www.ietf.org/rfc/rfc2616.txt]. That is, wherever the specification indicates MUST, MUST NOT, SHOULD, or SHOULD 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.

6.3. Example Usage

This is a simple example of how to set up a basic caching HttpClient. 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 = CacheConfig.custom()
        .setMaxCacheEntries(1000)
        .setMaxObjectSize(8192)
        .build();
RequestConfig requestConfig = RequestConfig.custom()
        .setConnectTimeout(30000)
        .setSocketTimeout(30000)
        .build();
CloseableHttpClient cachingClient = CachingHttpClients.custom()
        .setCacheConfig(cacheConfig)
        .setDefaultRequestConfig(requestConfig)
        .build();

HttpCacheContext context = HttpCacheContext.create();
HttpGet httpget = new HttpGet("http://www.mydomain.com/content/");
CloseableHttpResponse response = cachingClient.execute(httpget, context);
try {
    CacheResponseStatus responseStatus = context.getCacheResponseStatus();
    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;
    }
} finally {
    response.close();
}
```

6.4. Configuration

The caching HttpClient inherits all configuration options and parameters of the default non-caching implementation (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 caching HttpClient 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 caching HttpClient 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, with some extra efforts it's entirely possible to set up a multi-tier caching hierarchy; for example, wrapping an in-memory caching HttpClient 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(
            CharArrayBuffer buffer) throws ParseException {
        try {
            return super.parseHeader(buffer);
        } catch (ParseException ex) {
            // Suppress ParseException exception
            return new BasicHeader(buffer.toString(), null);
        }
    }

}
```

- Provide a custom `HttpConnectionFactory` implementation. Replace default request writer and / or response parser with custom ones as required.

```
HttpConnectionFactory<HttpRoute, ManagedHttpClientConnection> connFactory =
        new ManagedHttpClientConnectionFactory(
            new DefaultHttpRequestWriterFactory(),
            new DefaultHttpResponseParserFactory(
                    new MyLineParser(), new DefaultHttpResponseFactory()));
```

- Configure HttpClient to use the custom connection factory.

```
PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionManager(
    connFactory);
CloseableHttpClient httpclient = HttpClients.custom()
        .setConnectionManager(cm)
        .build();
```

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 principal 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.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpClientContext context = HttpClientContext.create();
HttpGet httpget = new HttpGet("http://localhost:8080/");
CloseableHttpResponse response = httpclient.execute(httpget, context);
try {
    Principal principal = context.getUserToken(Principal.class);
    System.out.println(principal);
} finally {
    response.close();
}
```

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

```
UserTokenHandler userTokenHandler = new UserTokenHandler() {

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

};
CloseableHttpClient httpclient = HttpClients.custom()
        .setUserTokenHandler(userTokenHandler)
        .build();
```

7.2.2. 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.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
HttpClientContext context1 = HttpClientContext.create();
HttpGet httpget1 = new HttpGet("http://localhost:8080/");
CloseableHttpResponse response1 = httpclient.execute(httpget1, context1);
try {
    HttpEntity entity1 = response1.getEntity();
} finally {
```

```
    response1.close();
}
Principal principal = context1.getUserToken(Principal.class);

HttpClientContext context2 = HttpClientContext.create();
context2.setUserToken(principal);
HttpGet httpget2 = new HttpGet("http://localhost:8080/");
CloseableHttpResponse response2 = httpclient.execute(httpget2, context2);
try {
    HttpEntity entity2 = response2.getEntity();
} finally {
    response2.close();
}
```

7.3. Using the FutureRequestExecutionService

Using the FutureRequestExecutionService, you can schedule http calls and treat the response as a Future. This is useful when e.g. making multiple calls to a web service. The advantage of using the FutureRequestExecutionService is that you can use multiple threads to schedule requests concurrently, set timeouts on the tasks, or cancel them when a response is no longer necessary.

FutureRequestExecutionService wraps the request with a HttpRequestFutureTask, which extends FutureTask. This class allows you to cancel the task as well as keep track of various metrics such as request duration.

7.3.1. Creating the FutureRequestExecutionService

The constructor for the futureRequestExecutionService takes any existing httpClient instance and an ExecutorService instance. When configuring both, it is important to align the maximum number of connections with the number of threads you are going to use. When there are more threads than connections, the connections may start timing out because there are no available connections. When there are more connections than threads, the futureRequestExecutionService will not use all of them

```
HttpClient httpClient = HttpClientBuilder.create().setMaxConnPerRoute(5).build();
ExecutorService executorService = Executors.newFixedThreadPool(5);
FutureRequestExecutionService futureRequestExecutionService =
    new FutureRequestExecutionService(httpClient, executorService);
```

7.3.2. Scheduling requests

To schedule a request, simply provide a HttpUriRequest, HttpContext, and a ResponseHandler. Because the request is processed by the executor service, a ResponseHandler is mandatory.

```
private final class OkidokiHandler implements ResponseHandler<Boolean> {
    public Boolean handleResponse(
            final HttpResponse response) throws ClientProtocolException, IOException {
        return response.getStatusLine().getStatusCode() == 200;
    }
}

HttpRequestFutureTask<Boolean> task = futureRequestExecutionService.execute(
    new HttpGet("http://www.google.com"), HttpClientContext.create(),
    new OkidokiHandler());
// blocks until the request complete and then returns true if you can connect to Google
boolean ok=task.get();
```

7.3.3. Canceling tasks

Scheduled tasks may be cancelled. If the task is not yet executing but merely queued for execution, it simply will never execute. If it is executing and the mayInterruptIfRunning parameter is set to true, abort() will be called on the request; otherwise the response will simply be ignored but the request will be allowed to complete normally. Any subsequent calls to task.get() will fail with an IllegalStateException. It should be noticed that canceling tasks merely frees up the client side resources. The request may actually be handled normally on the server side.

```
task.cancel(true)
task.get() // throws an Exception
```

7.3.4. Callbacks

Instead of manually calling task.get(), you can also use a FutureCallback instance that gets callbacks when the request completes. This is the same interface as is used in HttpAsyncClient

```
private final class MyCallback implements FutureCallback<Boolean> {

    public void failed(final Exception ex) {
        // do something
    }

    public void completed(final Boolean result) {
        // do something
    }

    public void cancelled() {
        // do something
    }
}

HttpRequestFutureTask<Boolean> task = futureRequestExecutionService.execute(
    new HttpGet("http://www.google.com"), HttpClientContext.create(),
    new OkidokiHandler(), new MyCallback());
```

7.3.5. Metrics

FutureRequestExecutionService is typically used in applications that make large amounts of web service calls. To facilitate e.g. monitoring or configuration tuning, the FutureRequestExecutionService keeps track of several metrics.

Each HttpRequestFutureTask provides methods to get the time the task was scheduled, started, and ended. Additionally, request and task duration are available as well. These metrics are aggregated in the FutureRequestExecutionService in a FutureRequestExecutionMetrics instance that may be accessed through FutureRequestExecutionService.metrics().

```
task.scheduledTime() // returns the timestamp the task was scheduled
task.startedTime() // returns the timestamp when the task was started
task.endedTime() // returns the timestamp when the task was done executing
task.requestDuration // returns the duration of the http request
task.taskDuration // returns the duration of the task from the moment it was scheduled

FutureRequestExecutionMetrics metrics = futureRequestExecutionService.metrics()
metrics.getActiveConnectionCount() // currently active connections
```

```
metrics.getScheduledConnectionCount(); // currently scheduled connections
metrics.getSuccessfulConnectionCount(); // total number of successful requests
metrics.getSuccessfulConnectionAverageDuration(); // average request duration
metrics.getFailedConnectionCount(); // total number of failed tasks
metrics.getFailedConnectionAverageDuration(); // average duration of failed tasks
metrics.getTaskCount(); // total number of tasks scheduled
metrics.getRequestCount(); // total number of requests
metrics.getRequestAverageDuration(); // average request duration
metrics.getTaskAverageDuration(); // average task duration
```