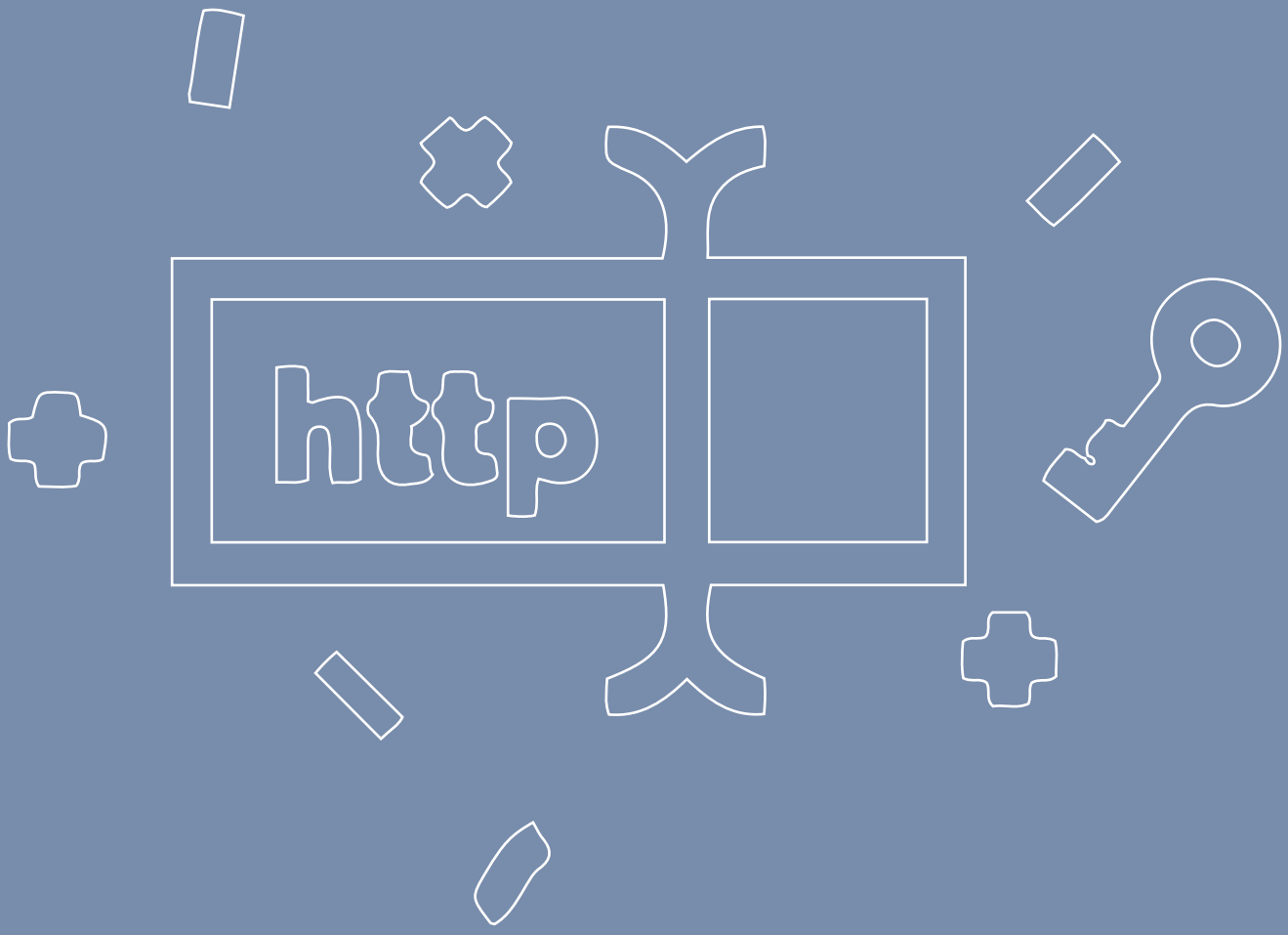
Get the Most Out of HttpClient



By Baeldung

TABLE OF CONTENTS

1: HTTPCLIENT 4 – GET THE STATUS CODE

2: HTTPCLIENT TIMEOUT

3: HTTPCLIENT 4 – CANCEL / ABORT REQUEST

4: HTTPCLIENT 4 – DO NOT FOLLOW REDIRECTS

5: HTTPCLIENT – SET CUSTOM HEADER

6: HTTPCLIENT WITH SSL

7: UNSHORTEN URLS WITH HTTPCLIENT

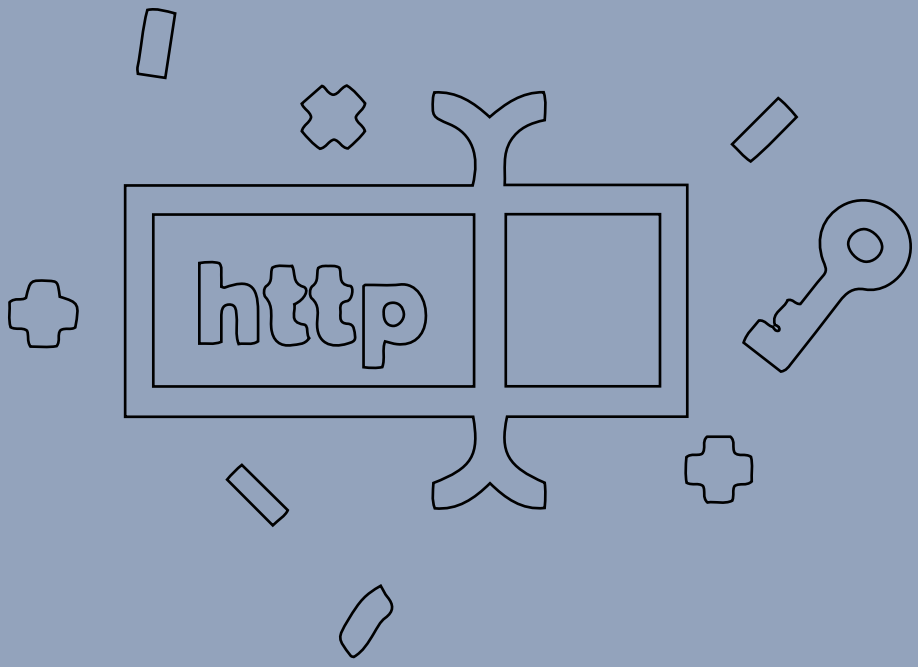
8: HTTPCLIENT 4 – SEND CUSTOM COOKIE

9: HTTPCLIENT 4 – FOLLOW REDIRECTS FOR POST

10: HTTPCLIENT BASIC AUTHENTICATION

11: HTTPCLIENT 4 COOKBOOK

CHAPTER 1



Get the Most
Out of
HttpClient

1: HTTPCLIENT 4 – GET THE STATUS CODE

1. Overview

In this very quick section, I will show how to **get and validate the StatusCode of the HTTP Response using HttpClient 4.**

2. Retrieve the Status Code from the Http Response

After sending the Http request – we get back an instance of *org.apache.http.HttpResponse* – which allows us to access the status line of the response, and implicitly the Status Code:

```
response.getStatusLine().getStatusCode()
```

Using this, we can **validate that the code we receive from the server is indeed correct:**

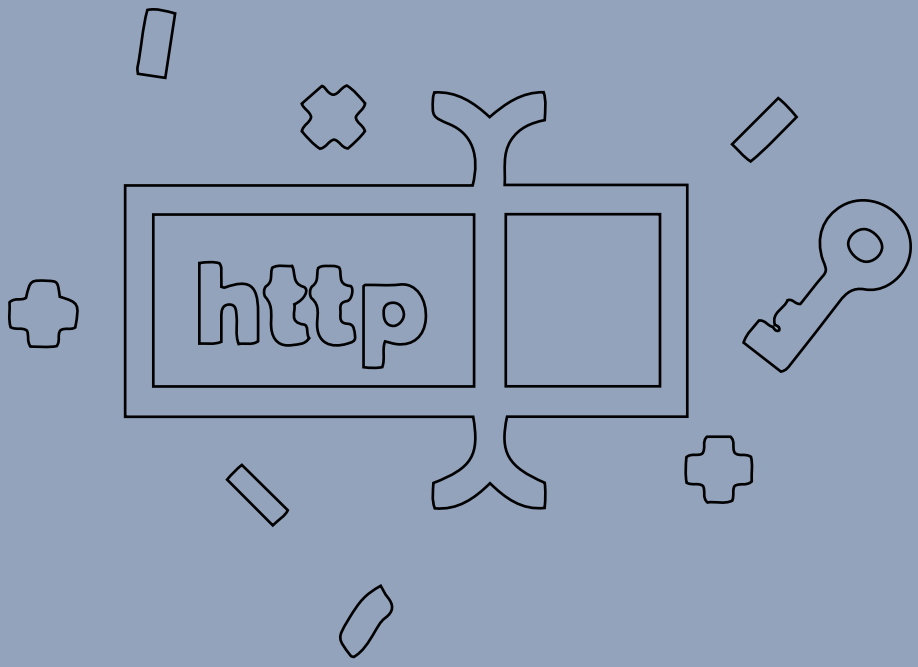
```
@Test
public void givenGetRequestExecuted_whenAnalyzingTheResponse_thenCorrectStatusCode()
  throws ClientProtocolException, IOException {
    HttpClient client = HttpClientBuilder.create().build();
    HttpResponse response = client.execute(new HttpGet(SAMPLE_URL));
    int statusCode = response.getStatusLine().getStatusCode();
    assertThat(statusCode, equalTo(HttpStatus.SC_OK));
}
```

Notice that we're using **the predefined Status Codes** also available in the library via *org.apache.http.HttpStatus.*

3. Conclusion

This very simple section shows how to **retrieve and work with Status Codes with the Apache HttpClient 4.**

CHAPTER 2



Get the Most
Out of
HttpClient

2: HTTPCLIENT TIMEOUT

1. Overview

This section will show how to **configure a timeout with the Apache *HttpClient 4.***

2. Configure Timeouts via raw String parameters

The *HttpClient* comes with a lot of configuration parameters – and all of these can be set in a generic, map-like manner.

There are **3 timeout parameters to configure:**

```
DefaultHttpClient httpClient = new DefaultHttpClient();


 int timeout = 5; // seconds
HttpParams httpParams = httpClient.getParams();
httpParams.setParameter(CoreConnectionPNames.CONNECTION_TIMEOUT, timeout * 1000);
httpParams.setParameter(CoreConnectionPNames.SO_TIMEOUT, timeout * 1000);
// httpParams.setParameter(ClientPNames.CONN_MANAGER_TIMEOUT, new Long(timeout * 1000));
```

A quick note is that the last parameter – the connection manager timeout – is commented out when using the 4.3.0 or 4.3.1 versions, because of this JIRA (due in 4.3.2).

3. Configure Timeouts via the API

The more important of these parameters – namely the first two – can also be set via a more type safe API:

```
DefaultHttpClient httpClient = new DefaultHttpClient();

int timeout = 5; // seconds
HttpParams httpParams = httpClient.getParams();
HttpConnectionParams.setConnectionTimeout(httpParams, timeout * 1000); // http.connection.time
out
HttpConnectionParams.setSoTimeout(httpParams, timeout * 1000); // http.socket.timeout
```

The third parameter doesn't have a custom setter in *HttpConnectionParams,* and it will still need to be set manually via the *setParameter* method.

4. Configure Timeouts using the new 4.3 Builder

The fluent, builder API introduced in 4.3 provides the right **way to set timeouts at a high level:**

```
int timeout = 5;
RequestConfig config = RequestConfig.custom()
  .setConnectTimeout(timeout * 1000)
  .setConnectionRequestTimeout(timeout * 1000)
  .setSocketTimeout(timeout * 1000).build();
CloseableHttpClient client =
  HttpClientBuilder.create().setDefaultRequestConfig(config).build();
```

That is the recommended way of configuring all three timeouts in a type-safe and readable manner.

5. Timeout Properties Explained

Now, let's explain what these various types of timeouts mean:

- the **Connection Timeout** *(http.connection.timeout)* – the time to establish the connection with the remote host
- the **Socket Timeout** *(http.socket.timeout)* – the time waiting for data – after the connection was established; maximum time of inactivity between two data packets
- the **Connection Manager Timeout** *(http.connection-manager.timeout)* – the time to wait for a connection from the connection manager/pool
- The first two parameters – the connection and socket timeouts – are the most important, but setting a timeout for obtaining a connection is definitely important in high load scenarios, which is why the third parameter shouldn't be ignored.

6. Using the *HttpClient*

After being configured, the client cannot be used to perform HTTP requests:

```
HttpGet getMethod = new HttpGet("http://host:8080/path");
HttpResponse response = httpClient.execute(getMethod);
System.out.println("HTTP Status of response: " + response.getStatusLine().getStatusCode());
```

- With the previously defined client, the connection to the host will **time out in 5 seconds**, and if the connection is established but no data is received, the timeout will also be 5 additional seconds.

Note that the connection timeout will result in a ***org.apache.http.conn.ConnectTimeoutException*** **being thrown**, while socket timeout will result in **a *java.net.SocketTimeoutException.***

---

7. Hard Timeout

While setting timeouts on establishing the HTTP connection and not receiving data is very useful, sometimes we need to set a **hard timeout for the entire request.**

For example the download of a potentially large file fits into this category – in this case, the connection may be successfully established, data may be consistently coming through, but we still need to ensure that the operation doesn't go over some specific time threshold.

HttpClient doesn't have any configuration that allows us to set an overall timeout for a request; it does however provide **abort functionality for requests**, so we can leverage that mechanism to implement a simple timeout mechanism:

```
HttpGet getMethod = new HttpGet("http://localhost:8080/spring-security-rest-template/api/bars/1");

int hardTimeout = 5; // seconds
TimerTask task = new TimerTask() {
   @Override
   public void run() {
      if (getMethod != null) {
         getMethod.abort();
      }
   }
};
new Timer(true).schedule(task, hardTimeout * 1000);

 HttpResponse response = httpClient.execute(getMethod);
System.out.println("HTTP Status of response: " + response.getStatusLine().getStatusCode());
```

We're making use of the *java.util.Timer* and *java.util.TimerTask* to set up a **simple delayed task which aborts the HTTP GET request** after a 5 seconds hard timeout.

8. Timeout and DNS Round Robin – something to be aware of

It is quite common that some larger domains will be using a DNS round robin configuration – essentially having the **same domain mapped to multiple IP addresses.** This introduces a new challenge for timeout against such a domain, simply because of the way HttpClient will try to connect to that domain that times out:

- HttpClient gets the **list of IP routes** to that domain
- it tries **the first one** – that times out (with the timeouts we configure)
- it tries **the second one** – that also times out
- and so on …

So, as you can see – **the overall operation will not time out when we expect it to.** Instead – it will time out when all the possible routes have timed out, and what it more – this will happen completely transparently for the client (unless you have your log configured at the DEBUG level). Here is a simple example you can run and replicate this issue:

```
int timeout = 3;
RequestConfig config = RequestConfig.custom().
  setConnectTimeout(timeout * 1000).
  setConnectionRequestTimeout(timeout * 1000).
  setSocketTimeout(timeout * 1000).build();
CloseableHttpClient client = HttpClientBuilder.create().setDefaultRequestConfig(config).build();


HttpGet request = new HttpGet("http://www.google.com:81");
response = client.execute(request);
```

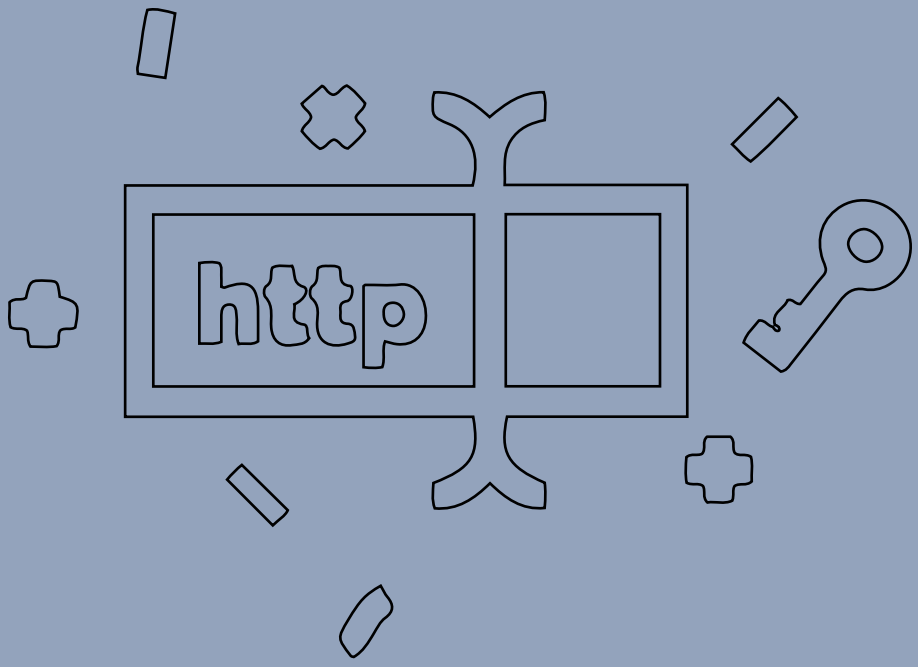
You will notice the retrying logic with a DEBUG log level:

```
DEBUG o.a.h.i.c.HttpClientConnectionOperator - Connecting to www.google.com/173.194.34.212:81
DEBUG o.a.h.i.c.HttpClientConnectionOperator -
 Connect to www.google.com/173.194.34.212:81 timed out. Connection will be retried using another IP
address

DEBUG o.a.h.i.c.HttpClientConnectionOperator - Connecting to www.google.com/173.194.34.208:81
DEBUG o.a.h.i.c.HttpClientConnectionOperator -
 Connect to www.google.com/173.194.34.208:81 timed out. Connection will be retried using another IP
 address

DEBUG o.a.h.i.c.HttpClientConnectionOperator - Connecting to www.google.com/173.194.34.209:81
DEBUG o.a.h.i.c.HttpClientConnectionOperator -
 Connect to www.google.com/173.194.34.209:81 timed out. Connection will be retried using another IP
 address
...
```

9. Conclusion

This section discussed how to configure the various types of timeouts available for an *HttpClient*. It also illustrated a simple mechanism for hard timeout of an ongoing http connection.

CHAPTER 3



Get the Most
Out of
HttpClient

3: HTTPCLIENT 4 – CANCEL / ABORT REQUEST

1. Overview

This quick section shows how to **cancel a HTTP Request with the Apache HttpClient 4.**

This is especially useful for potentially long running requests, or large download files that would otherwise unnecessarily consume bandwidth and connections.

2. Abort a GET Request

To abort an ongoing request, the client can simply use:

```
request.abort();
```

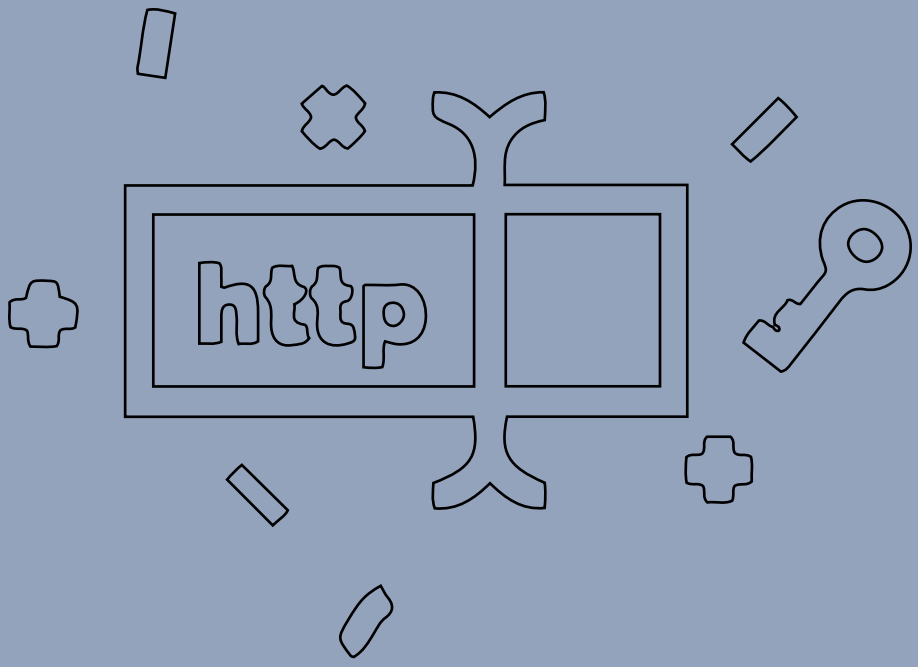
This will make sure that the client doesn't have to consume the entire body of the request to release the connection:

```
@Test
public void whenRequestIsCanceled_thenCorrect()
  throws ClientProtocolException, IOException {
    HttpClient instance = HttpClients.custom().build();
    HttpGet request = new HttpGet(SAMPLE_URL);
    HttpResponse response = instance.execute(request);

    try {
        System.out.println(response.getStatusLine());
        request.abort();
    } finally {
        response.close();
    }
}
```

3. Conclusion

This section illustrated how to abort an ongoing request with the HTTP client. Another option to stop long running requests if to make sure that they will time out.

CHAPTER 4



Get the Most
Out of
HttpClient

4: HTTPCLIENT 4 – DO NOT FOLLOW REDIRECTS

1. Overview

In this section I will show how to **configure the Apache HttpClient 4 to stop following redirects.**

By default, following **the HTTP Spec, the HttpClient will automatically follow redirects.**

For some use cases, that may be perfectly fine, but there are certainly use cases where that's not desired – and we'll now look at how to change that default behavior and **stop following redirects.**

If you want to dig deeper and learn other cool things you can do with the HttpClient – head on over to **the main HttpClient section.**

2. Do Not Follow Redirects

2.1. Before HttpClient 4.3

In older versions of the Http Client (before 4.3), we can configure what the client does with redirects as follows:

```
@Test
public void givenRedirectsAreDisabled_whenConsumingUrlWhichRedirects_thenNotRedirected()
  throws ClientProtocolException, IOException {
    DefaultHttpClient instance = new DefaultHttpClient();

    HttpParams params = new BasicHttpParams();
    params.setParameter(ClientPNames.HANDLE_REDIRECTS, false);
    // HttpClientParams.setRedirecting(params, false); // alternative

    HttpGet httpGet = new HttpGet("http://t.co/I5YYd9tddw");
    httpGet.setParams(params);
    CloseableHttpResponse response = instance.execute(httpGet);

    assertThat(response.getStatusLine().getStatusCode(), equalTo(301));
}
```

Notice the alternative API that can be used to configure the redirect behavior **without using setting the actual raw *http.protocol.handle-redirects* parameter:**

```
HttpClientParams.setRedirecting(params, false);
```

Also notice that, with follow redirects disabled, we can now check that the Http Response status code is indeed301 *Moved Permanently* – as it should be.

2.2. After HttpClient 4.3

HttpClient 4.3 introduced a cleaner, more high level API to build and configure the client:

```
@Test
public void givenRedirectsAreDisabled_whenConsumingUrlWhichRedirects_thenNotRedirected()
  throws ClientProtocolException, IOException {
    HttpClient instance = HttpClientBuilder.create().disableRedirectHandling().build();
    HttpResponse response = instance.execute(new HttpGet("http://t.co/I5YYd9tddw"));

    assertThat(response.getStatusLine().getStatusCode(), equalTo(301));
}
```

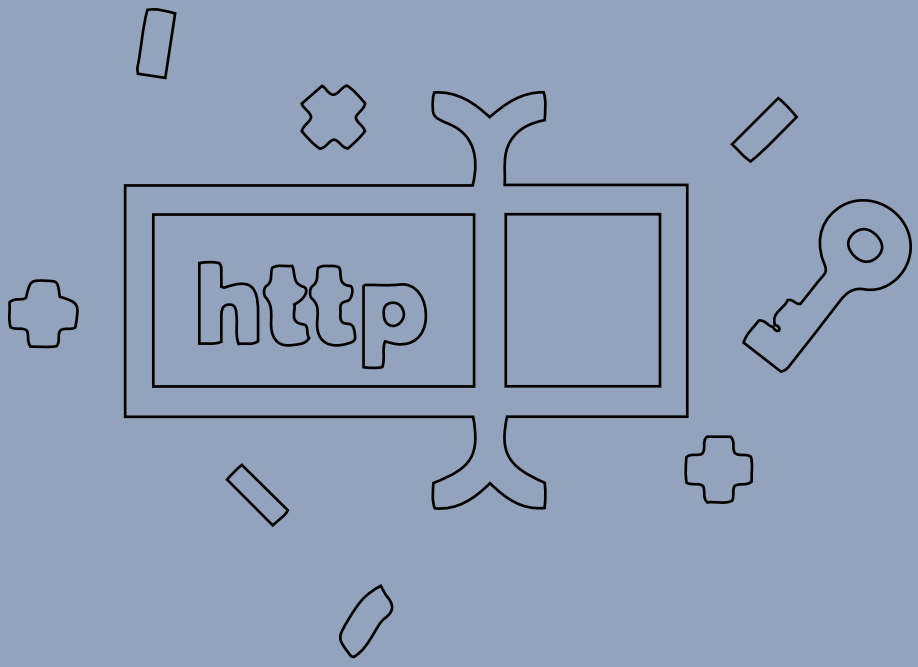
Note that the new API configures the entire client with this redirect behavior – not just the individual request.

---

3. Conclusion

This quick section covered how to configure the Apache HttpClient – both pre 4.3 and post – to prevent it from following HTTP redirects automatically.

CHAPTER 5



Get the Most
Out of
HttpClient

5: HTTPCLIENT – SET CUSTOM HEADER

1. Overview

In this section we'll look at how to set a custom header with the HttpClient.

2. Set Header on Request – Before 4.3

You can set any custom header on a request with **a simple *setHeader* call:**

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet(SAMPLE_URL);
request.setHeader(HttpHeaders.CONTENT_TYPE, "application/json");
client.execute(request);
```

As you can see, we're setting the *Content-Type* directly on the request to a custom value – JSON.

3. Set Header on Request – 4.3 and above

HttpClient 4.3 has introduced a new way of building requests – the RequestBuilder. To set a header, we will use the same, ***setHeader*** **method – on the builder:**

```
HttpClient client = HttpClients.custom().build();
HttpUriRequest request = RequestBuilder.get().setUri(SAMPLE_URL)
  .setHeader(HttpHeaders.CONTENT_TYPE, “application/json”).build();
client.execute(request);
```

4. Set Default Header on the Client – 4.3 and above

Instead of setting the Header on each and every request, you can also **configure it as a default header on the Client** itself:

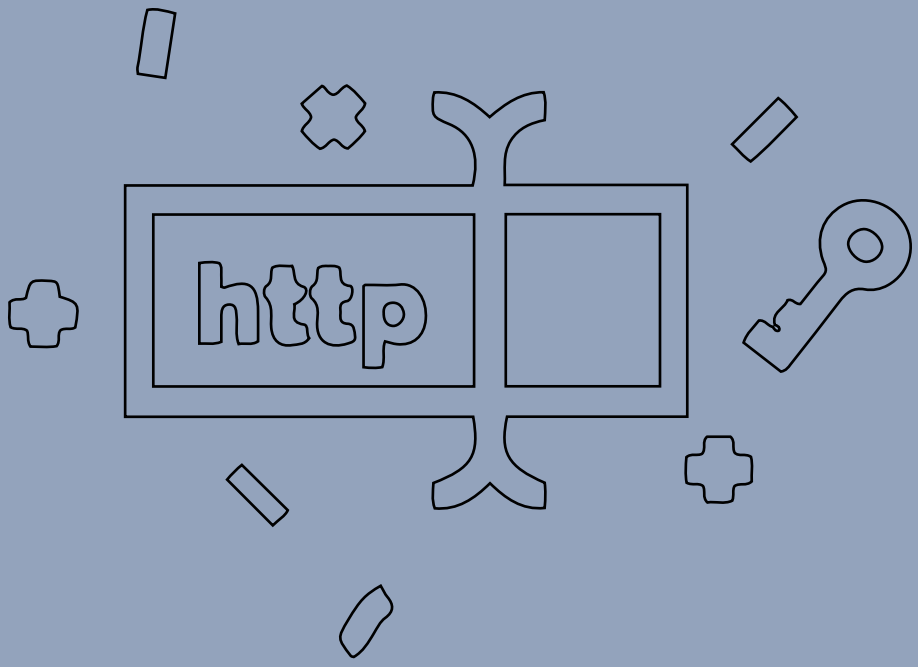
```
Header header = new BasicHeader(HttpHeaders.CONTENT_TYPE, “application/json”);
List<Header> headers = Lists.newArrayList(header);
HttpClient client = HttpClients.custom().setDefaultHeaders(headers).build();
HttpUriRequest request = RequestBuilder.get().setUri(SAMPLE_URL).build();
client.execute(request);
```

This is extremely helpful when the header needs to be the same for all requests – such as a custom application header.

5. Conclusion

This section illustrated how to add an HTTP header to one or all requests sent via the Apache HttpClient.

CHAPTER 6



Get the Most
Out of
HttpClient

6: HTTPCLIENT WITH SSL

1. Overview

This section will show how to **configure the Apache HttpClient 4 with "Accept All" SSL support.** The goal is simple – consume HTTPS URLs which do not have valid certificates.

2. The *SSLPeerUnverifiedException*

Without configuring SSL with the *HttpClient*, the following test – consuming an HTTPS URL – will fail:

```
public class HttpLiveTest {

   @Test(expected = SSLPeerUnverifiedException.class)
   public void whenHttpsUrlIsConsumed_thenException()
     throws ClientProtocolException, IOException {
       DefaultHttpClient httpClient = new DefaultHttpClient();
       String urlOverHttps = "https://localhost:8080/spring-security-rest-basic-auth";
       HttpGet getMethod = new HttpGet(urlOverHttps);
       HttpResponse response = httpClient.execute(getMethod);
       assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
   }
}
```

The exact failure is:

```
javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated
    at sun.security.ssl.SSLSessionImpl.getPeerCertificates(SSLSessionImpl.java:397)
    at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:126)
    ...
```

The *javax.net.ssl.SSLPeerUnverifiedException* exception occurs whenever a valid chain of trust couldn't be established for the URL.

3. Configure SSL – Accept All (HttpClient < 4.3)

Let's now configure the http client to trust all certificate chains regardless of their validity:

```
@Test
public void givenAcceptingAllCertificates_whenHttpsUrlIsConsumed_thenException()
  throws IOException, GeneralSecurityException {
    TrustStrategy acceptingTrustStrategy = new TrustStrategy() {
        @Override
        public boolean isTrusted(X509Certificate[] certificate, String authType) {
            return true;
        }
    };
    SSLSocketFactory sf = new SSLSocketFactory(acceptingTrustStrategy,
      SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
    SchemeRegistry registry = new SchemeRegistry();
    registry.register(new Scheme("https", 8443, sf));
    ClientConnectionManager ccm = new PoolingClientConnectionManager(registry);

    DefaultHttpClient httpClient = new DefaultHttpClient(ccm);

    String urlOverHttps = "https://localhost:8443/spring-security-rest-basic-auth/api/bars/1";
    HttpGet getMethod = new HttpGet(urlOverHttps);
    HttpResponse response = httpClient.execute(getMethod);
    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

With the new *TrustStrategy* now **overriding the standard certificate verification process** (which should consult a configured trust manager) – the test now passes and **the client is able to consume the HTTPS URL.**

4. The Spring *RestTemplate* with SSL (HttpClient < 4.3)

Now that we have seen how to configure a raw *HttpClient* with SSL support, let's take a look at a more high level client – the Spring *RestTemplate.*

With no SSL configured, the following test fails as expected:

```
@Test(expected = ResourceAccessException.class)
public void whenHttpsUrlIsConsumed_thenException() {
   String urlOverHttps = “https://localhost:8443/spring-security-rest-basic-auth/api/bars/1”;
   ResponseEntity<String> response =
     new RestTemplate().exchange(urlOverHttps, HttpMethod.GET, null, String.class);
   assertThat(response.getStatusCode().value(), equalTo(200));
}
```

So let's configure SSL:

```
import static org.apache.http.conn.ssl.SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER;
import java.security.GeneralSecurityException;
import java.security.cert.X509Certificate;
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.conn.scheme.Scheme;
import org.apache.http.conn.ssl.SSLSocketFactory;
import org.apache.http.conn.ssl.TrustStrategy;
import org.apache.http.impl.client.DefaultHttpClient;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.web.client.ResourceAccessException;
import org.springframework.web.client.RestTemplate;

...
@Test
public void givenAcceptingAllCertificates_whenHttpsUrlIsConsumed_thenException()
  throws GeneralSecurityException {
    HttpComponentsClientHttpRequestFactory requestFactory =
      new HttpComponentsClientHttpRequestFactory();
    DefaultHttpClient httpClient = (DefaultHttpClient) requestFactory.getHttpClient();
    TrustStrategy acceptingTrustStrategy = new TrustStrategy() {
    @Override
        public boolean isTrusted(X509Certificate[] certificate, String authType) {
            return true;
        }
    };
    SSLSocketFactory sf =
      new SSLSocketFactory(acceptingTrustStrategy, ALLOW_ALL_HOSTNAME_VERIFIER);
    httpClient.getConnectionManager().getSchemeRegistry().register(new Scheme("https", 8443, sf));

    String urlOverHttps = "https://localhost:8443/spring-security-rest-basic-auth/api/bars/1";
    ResponseEntity<String> response = new RestTemplate(requestFactory).
      exchange(urlOverHttps, HttpMethod.GET, null, String.class);
    assertThat(response.getStatusCode().value(), equalTo(200));
}
```

As you can see, this is **very similar to the way we configured SSL for the raw HttpClient** – we configure the request factory with SSL support and then we instantiate the template passing this preconfigured factory.

5. Configure SSL – Accept All (HttpClient 4.4)

In HttpClient version 4.4, with *SSLSocketFactory* now deprecated, we can simply configure our *HttpClient* as follows:

```
@Test
public void givenAcceptingAllCertificatesUsing4_4_whenHttpsUrlIsConsumed_thenCorrect()
throws ClientProtocolException, IOException {
   CloseableHttpClient httpClient =
     HttpClients.custom()
             .setSSLHostnameVerifier(new NoopHostnameVerifier())
             .build();

   HttpGet getMethod = new HttpGet(urlOverHttps);
   HttpResponse response = httpClient.execute(getMethod);
   assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

6. The Spring RestTemplate with SSL (HttpClient 4.4)

And we can use the same way to configure our *RestTemplate*:

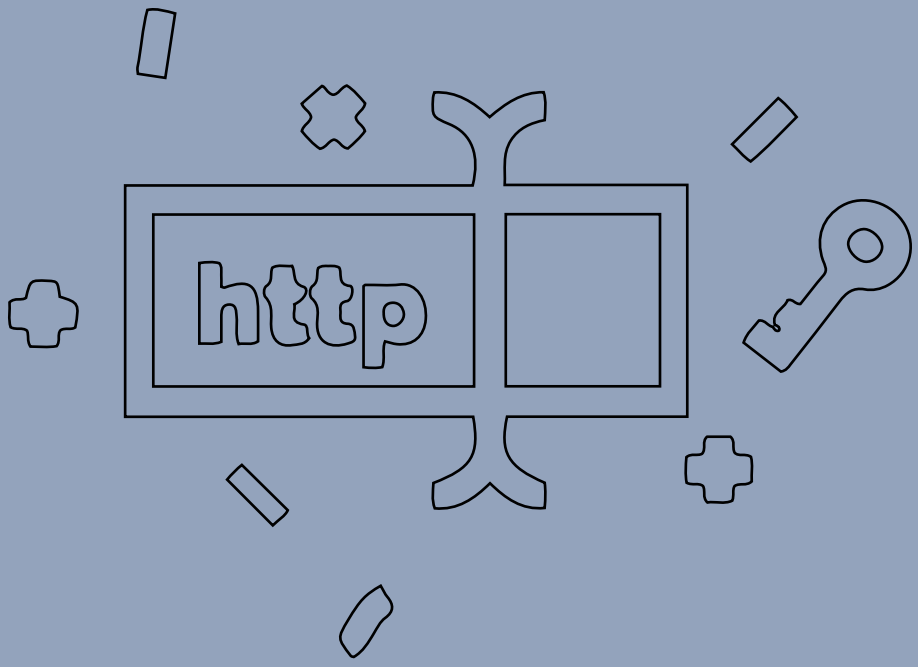
```
@Test
public void givenAcceptingAllCertificatesUsing4_4_whenUsingRestTemplate_thenCorrect()
throws ClientProtocolException, IOException {
   CloseableHttpClient httpClient =
     HttpClients.custom()
             .setSSLHostnameVerifier(new NoopHostnameVerifier())
             .build();
   HttpComponentsClientHttpRequestFactory requestFactory =
     new HttpComponentsClientHttpRequestFactory();
   requestFactory.setHttpClient(httpClient);

   ResponseEntity<String> response =
     new RestTemplate(requestFactory).exchange(
     urlOverHttps, HttpMethod.GET, null, String.class);
   assertThat(response.getStatusCode().value(), equalTo(200));
}
```

5. Conclusion

This section discussed how to configure SSL for an Apache HttpClient so that it is able to consume any HTTPS URL, regardless of the certificate. The same configuration for the Spring *RestTemplate* is also illustrated.

An important thing to understand however is that **this strategy entirely ignores certificate checking** – which makes it insecure and only to be used where that makes sense.

CHAPTER 7



Get the Most
Out of
HttpClient

7: UNSHORTEN URLS WITH HTTPCLIENT

1. Overview

In this section we're going to show how to **unshorten an URLs using** *HttpClient*.

A simple example is when **the original URL has been shortened once** – by a service such as *bit.ly.*

A more complex example is when **the URL has been shortened multiple times**, by different such services, and it takes multiple passes to get to the original full URL.

2. Unshorten the URL once

Let's start simple – unshorten an URL that has only been passed through a shorten URL service once.

First thing we'll need is an http client that **doesn't automatically follow redirects:**

```
CloseableHttpClient client =
  HttpClientBuilder.create().disableRedirectHandling().build();
```

This is necessary because we'll need to manually intercept the redirect response and extract information out of it.

We start by sending a request to the shortened URL – the response we get back will be a *301 Moved Permanently.*

Then, we need to **extract the *Location* header** pointing to the next, and in this case – final URL:

```
public String expandSingleLevel(String url) throws IOException {
    HttpHead request = null;
    try {
        request = new HttpHead(url);
        HttpResponse httpResponse = client.execute(request);

        int statusCode = httpResponse.getStatusLine().getStatusCode();
        if (statusCode != 301 && statusCode != 302) {
            return url;
        }
        Header[] headers = httpResponse.getHeaders(HttpHeaders.LOCATION);
        Preconditions.checkState(headers.length == 1);
        String newUrl = headers[0].getValue();
        return newUrl;
    } catch (IllegalArgumentException uriEx) {
        return url;
    } finally {
        if (request != null) {
            request.releaseConnection();
        }
    }
}
```

Finally, a simple live test expanding an URL:

```
@Test
public void givenShortenedOnce_whenUrlIsUnshortened_thenCorrectResult() throws IOException {
    String expectedResult = “http://www.baeldung.com/rest-versioning”;
    String actualResult = expandSingleLevel(“http://bit.ly/13jEoS1”);
    assertThat(actualResult, equalTo(expectedResult));
}
```

3. Process Multiple URL levels

The problem with short URLs is that they may be **shortened multiple times,** by altogether different services. Expanding such an URL will need multiple passes to get to the original URL.

We're going to apply the *expandSingleLevel* primitive operation defined previously to simply **iterate through all the intermediary URL and get to the final target:**

```
public String expand(String urlArg) throws IOException {
   String originalUrl = urlArg;
   String newUrl = expandSingleLevel(originalUrl);
   while (!originalUrl.equals(newUrl)) {
      originalUrl = newUrl;
      newUrl = expandSingleLevel(originalUrl);
   }
   return newUrl;
}
```

Now, with the new mechanism of expanding multiple levels of URLs, let's define a test and put this to work:

```
@Test
public void givenShortenedMultiple_whenUrlIsUnshortened_thenCorrectResult() throws IOException
 {
   String expectedResult = "http://www.baeldung.com/rest-versioning";
   String actualResult = expand("http://t.co/e4rDDbnzmk");
   assertThat(actualResult, equalTo(expectedResult));
}
```

This time, the short URL – *http://t.co/e4rDDbnzmk* – which is actually shortened twice – once via *bit.ly* and a second time via the t.co service – is correctly expanded to the original URL.

4. Detect on redirect loops

Finally, some URLs cannot be expanded because they form a redirect loop. This type of problem would be detected by the *HttpClient*, but since we turned off the automatic follow of redirects, it no longer does.

The final step in the URL expansion mechanism is going to be detecting the redirect loops and failing fast in case such a loop occurs.

For this to be effective, we need some additional information out of the *expandSingleLevel* method we defined earlier – mainly, we need to also return the status code of the response along with the URL.

Since java doesn't support multiple return values, we're going to **wrap the information in *aorg.apache.commons.lang3.tuple.Pair* object** – the new signature of the method will now be:

```
public Pair<Integer, String> expandSingleLevelSafe(String url) throws IOException {
```

And finally, let's include the redirect cycle detection in the main expand mechanism:

```
public String expandSafe(String urlArg) throws IOException {
   String originalUrl = urlArg;
   String newUrl = expandSingleLevelSafe(originalUrl).getRight();
   List<String> alreadyVisited = Lists.newArrayList(originalUrl, newUrl);
   while (!originalUrl.equals(newUrl)) {
      originalUrl = newUrl;
      Pair<Integer, String> statusAndUrl = expandSingleLevelSafe(originalUrl);
      newUrl = statusAndUrl.getRight();
      boolean isRedirect = statusAndUrl.getLeft() == 301 || statusAndUrl.getLeft() == 302;
      if (isRedirect && alreadyVisited.contains(newUrl)) {
         throw new IllegalStateException("Likely a redirect loop");
      }
      alreadyVisited.add(newUrl);
   }
   return newUrl;
}
```

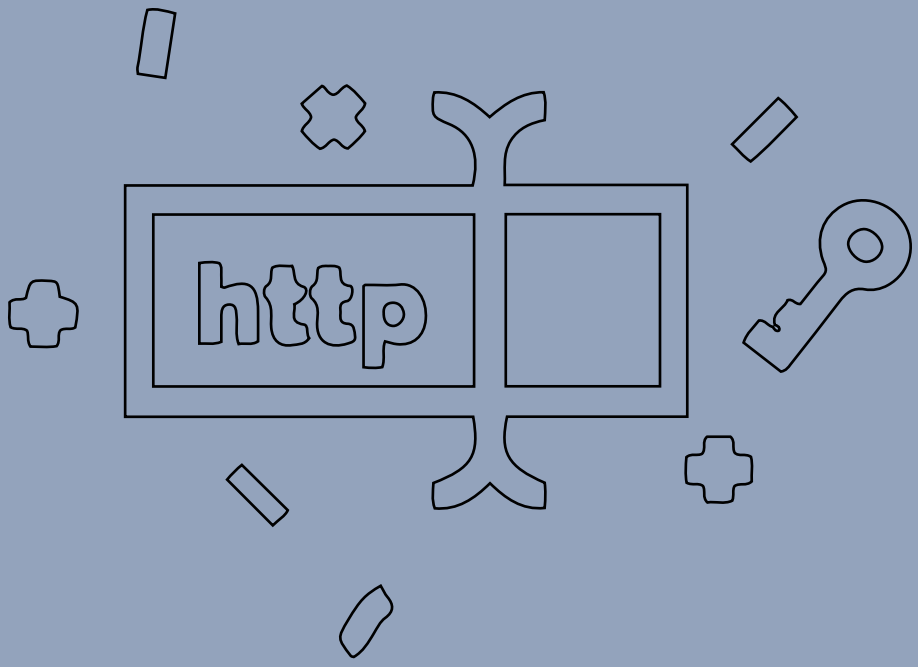
And that's it – the *expandSafe* mechanism is able to unshorten URL going through an arbitrary number of URL shortening services, while correctly failing fast on redirect loops.

5. Conclusion

This section discussed how to expand short URLs in java – using the Apache *HttpClient*.

We started with a simple use case with an URL that is only shortened once, and then implemented a more generic mechanism, capable of handling multiple levels of redirects and detect redirect loops in the process.

CHAPTER 8



Get the Most
Out of

HttpClient

8: HTTPCLIENT 4 – SEND CUSTOM COOKIE

1. Overview

This section will focus on **how to send a Custom Cookie using the Apache HttpClient 4.**

2. Configure Cookie Management on the HttpClient

2.1. HttpClient after 4.3

In the newer HttpClient 4.3, we'll leverage the fluent builder API responsible with both constructing and configuring the client.

First, we'll need to create a cookie store and set up our sample cookie in the store:

```
BasicCookieStore cookieStore = new BasicCookieStore();
BasicClientCookie cookie = new BasicClientCookie("JSESSIONID", "1234");
cookie.setDomain(".github.com");
cookie.setPath("/");
cookieStore.addCookie(cookie);
```

Then, we can set up this cookie store on the HttpClient and send the request:

```
@Test
public void whenSettingCookiesOnTheHttpClient_thenCookieSentCorrectly()
  throws ClientProtocolException, IOException {
    BasicCookieStore cookieStore = new BasicCookieStore();
    BasicClientCookie cookie = new BasicClientCookie("JSESSIONID", "1234");
    cookie.setDomain(".github.com");
    cookie.setPath("/");
    cookieStore.addCookie(cookie);
    HttpClient client = HttpClientBuilder.create().setDefaultCookieStore(cookieStore).build();

    final HttpGet request = new HttpGet("http://www.github.com");

    response = client.execute(request);
    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

A very important element is the ***domain*** being set on the cookie – **without setting the proper domain, the client will not send the cookie** at all!

2.2. HttpClient before 4.3

With older versions of the HttpClient (before 4.3) – the cookie store was set directly on the *HttpClient:*

```
@Test
public void givenUsingDeprecatedApi_whenSettingCookiesOnTheHttpClient_thenCorrect()
  throws ClientProtocolException, IOException {
    BasicCookieStore cookieStore = new BasicCookieStore();
    BasicClientCookie cookie = new BasicClientCookie("JSESSIONID", "1234");
    cookie.setDomain(".github.com");
    cookie.setPath("/");
    cookieStore.addCookie(cookie);
    DefaultHttpClient client = new DefaultHttpClient();
    client.setCookieStore(cookieStore);

    HttpGet request = new HttpGet("http://www.github.com");

    response = client.execute(request);
    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

Other than the way the client is built, there is no other difference from the previous example.

3. Set the Cookie on the Request

If setting the cookie on the entire HttpClient is not an option, we can configure requests with the cookie individually:

```
@Test
public void whenSettingCookiesOnTheRequest_thenCookieSentCorrectly()
  throws ClientProtocolException, IOException {
    BasicCookieStore cookieStore = new BasicCookieStore();
    BasicClientCookie cookie = new BasicClientCookie("JSESSIONID", "1234");
    cookie.setDomain(".github.com");
    cookie.setPath("/");
    cookieStore.addCookie(cookie);
    instance = HttpClientBuilder.create().build();

    HttpGet request = new HttpGet("http://www.github.com");

    HttpContext localContext = new BasicHttpContext();
    localContext.setAttribute(HttpClientContext.COOKIE_STORE, cookieStore);
    // localContext.setAttribute(ClientContext.COOKIE_STORE, cookieStore); // before 4.3
    response = instance.execute(request, localContext);

    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

4. Set the Cookie on the Low Level Request

A low level alternative of setting the cookie on the HTTP Request would be setting it as a raw Header:

```
@Test
public void whenSettingCookiesOnARequest_thenCorrect()
  throws ClientProtocolException, IOException {
    instance = HttpClientBuilder.create().build();
    HttpGet request = new HttpGet("http://www.github.com");
    request.setHeader("Cookie", "JSESSIONID=1234");

    response = instance.execute(request);

    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

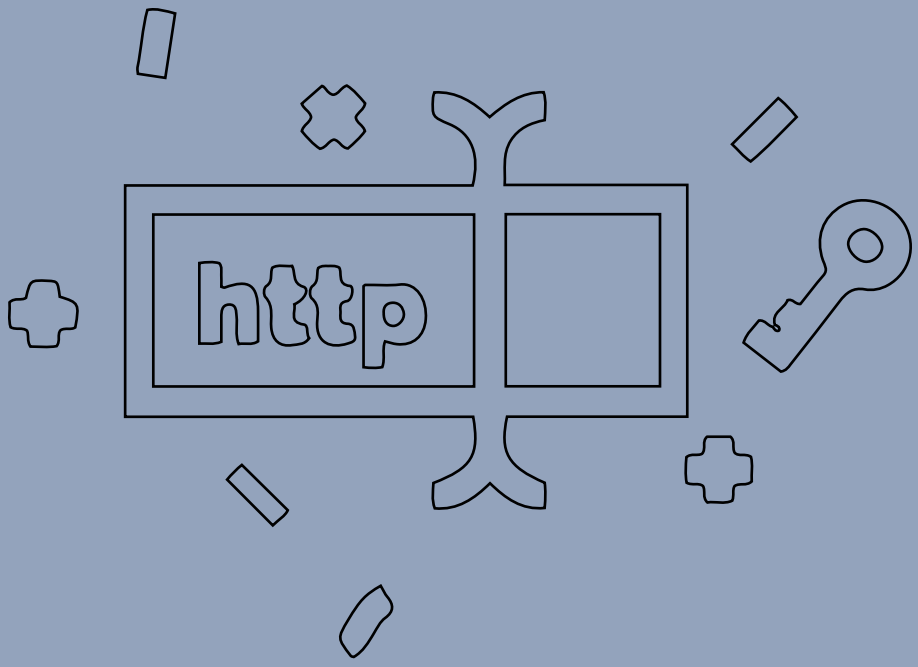
This is of course much more **error-prone than working with the built in cookie support** – for example, notice that we're no longer setting the domain in this case – which is not correct.

5. Conclusion

This section illustrated how to **work with the HttpClient to send a custom, user controlled Cookie.**

Note that this is not the same as letting the HttpClient deal with the cookies set by a server – but instead it's controlling the client side manually at a low level.

CHAPTER 9



Get the Most
Out of

HttpClient

9: HTTPCLIENT 4 – FOLLOW REDIRECTS FOR POST

1. Overview

This quick section will show how to configure the Apache HttpClient 4 to automatically follow redirects for POST requests.

By default, only GET requests resulting in a redirect are automatically followed. If a POST requests is answered with either *HTTP 301 Moved Permanently* or with *302 Found* – the redirect is not automatically followed.

This is specified by the HTTP RFC 2616:

If the 301 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

There are of course use cases where we need to change that behavior and relax the strict HTTP specification.

First, let's check the default behavior:

```
@Test
public void givenPostRequest_whenConsumingUrlWhichRedirects_thenNotRedirected()
  throws ClientProtocolException, IOException {
    HttpClient instance = HttpClientBuilder.create().build();
    HttpResponse response = instance.execute(new HttpPost("http://t.co/I5YYd9tddw"));
    assertThat(response.getStatusLine().getStatusCode(), equalTo(301));
}
```

As you can see, **the redirect is not followed by default**, and we get back the 301 Status Code.

2. Redirecting on HTTP POST

2.1. For HttpClient 4.3 and after

In HttpClient 4.3, a higher level API has been introduced for both creation and configuration of the client:

```
@Test
public void givenRedirectingPOST_whenConsumingUrlWhichRedirectsWithPOST_thenRedirected()
  throws ClientProtocolException, IOException {
    HttpClient instance =
      HttpClientBuilder.create().setRedirectStrategy(new LaxRedirectStrategy()).build();
    HttpResponse response = instance.execute(new HttpPost("http://t.co/I5YYd9tddw"));
    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

Notice **the *HttpClientBuilder* is now the starting point of a fluent API** which allows full configuration of the client in a more readable way than before.

2.2. For HttpClient 4.2

In the previous version of HttpClient (4.2) we can configure the redirect strategy directly on the client:

```
@SuppressWarnings("deprecation")
@Test
public void givenRedirectingPOST_whenConsumingUrlWhichRedirectsWithPOST_thenRedirected()
  throws ClientProtocolException, IOException {
    DefaultHttpClient client = new DefaultHttpClient();
    client.setRedirectStrategy(new LaxRedirectStrategy());

    HttpResponse response = client.execute(new HttpPost("http://t.co/I5YYd9tddw"));
    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

Notice that now, with the new *LaxRedirectStrategy*, the HTTP Restrictions are relaxed and **the redirect is followed over POST as well** – leading to a *200 OK* status code.

2.3. Pre HttpClient 4.2

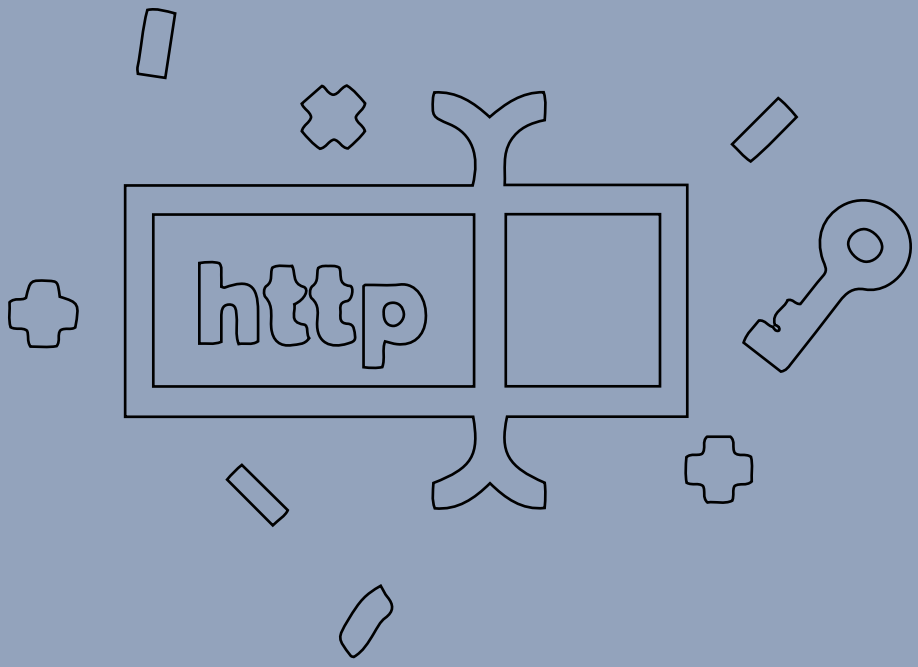
Before HttpClient 4.2, the LaxRedirectStrategy class didn't exist, so we need to roll our own:

```
@Test
public void givenRedirectingPOST_whenConsumingUrlWhichRedirectsWithPOST_thenRedirected()
  throws ClientProtocolException, IOException {
    DefaultHttpClient client = new DefaultHttpClient();
    client.setRedirectStrategy(new DefaultRedirectStrategy() {
        /** Redirectable methods. */
        private String[] REDIRECT_METHODS = new String[] {
            HttpGet.METHOD_NAME, HttpPost.METHOD_NAME, HttpHead.METHOD_NAME
        };

        @Override
        protected boolean isRedirectable(String method) {
            for (String m : REDIRECT_METHODS) {
                if (m.equalsIgnoreCase(method)) {
                    return true;
                }
            }
            return false;
        }
    });

    HttpResponse response = client.execute(new HttpPost("http://t.co/I5YYd9tddw"));
    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
}
```

3. Conclusion

This quick section illustrated how to configure any version of the Apache HttpClient 4 to follow redirects for HTTP POST requests as well – relaxing the strict HTTP standard.

CHAPTER 10



Get the Most
Out of

HttpClient

10: HTTPCLIENT BASIC AUTHENTICATION

1. Overview

This section will illustrate how to **configure Basic Authentication on the Apache HttpClient 4.**

2. Basic Authentication with the API

Let's start with the standard way of configuring Basic Authentication on the HttpClient – via *aCredentialsProvider:*

```
CredentialsProvider provider = new BasicCredentialsProvider();
UsernamePasswordCredentials credentials = new UsernamePasswordCredentials("user1", "user
1Pass");
provider.setCredentials(AuthScope.ANY, credentials);
HttpClient client = HttpClientBuilder.create().setDefaultCredentialsProvider(provider).build();

HttpResponse response = client.execute(new HttpGet(URL_SECURED_BY_BASIC_AUTHENTICATION));
int statusCode = response.getStatusLine().getStatusCode();
assertThat(statusCode, equalTo(HttpStatus.SC_OK));
```

As you can see, creating the client with a credentials provider to set it up with Basic Authentication is not difficult.

Now, to understand what *HttpClient* will actually do behind the scenes, we'll need to l**ook at the logs:**

```
# ... request is sent with no credentials
[main] DEBUG ... - Authentication required
[main] DEBUG ... - localhost:8080 requested authentication
[main] DEBUG ... - Authentication schemes in the order of preference:
  [negotiate, Kerberos, NTLM, Digest, Basic]
[main] DEBUG ... - Challenge for negotiate authentication scheme not available
[main] DEBUG ... - Challenge for Kerberos authentication scheme not available
[main] DEBUG ... - Challenge for NTLM authentication scheme not available
[main] DEBUG ... - Challenge for Digest authentication scheme not available
[main] DEBUG ... - Selected authentication options: [BASIC]
# ... the request is sent again - with credentials
```

The entire **Client-Server communication is now clear:**

- the Client sends the HTTP Request with no credentials
- the Server sends back a challenge
- the Client negotiates and identifies the right authentication scheme
- the Client sends **a second Request**, this time with credentials

3. Preemptive Basic Authentication

Out of the box, the HttpClient doesn't do preemptive authentication – this has to be an explicit decision made by the client.

First, we need to create the *HttpContext* – pre-populating it with **an authentication cache** with the right type of authentication scheme pre-selected. This will mean that the negatiation from the previous example is no longer necessary – **Basic Authentication is already chosen:**

```
HttpHost targetHost = new HttpHost("localhost", 8080, "http");
CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.setCredentials(AuthScope.ANY,
  new UsernamePasswordCredentials(DEFAULT_USER, DEFAULT_PASS));

AuthCache authCache = new BasicAuthCache();
authCache.put(targetHost, new BasicScheme());

// Add AuthCache to the execution context
final HttpClientContext context = HttpClientContext.create();
context.setCredentialsProvider(credsProvider);
context.setAuthCache(authCache);
```

Now we can use the client with the new context and **send the pre-authentication request:**

```
HttpClient client = HttpClientBuilder.create().build();
response = client.execute(new HttpGet(URL_SECURED_BY_BASIC_AUTHENTICATION), context);

int statusCode = response.getStatusLine().getStatusCode();
assertThat(statusCode, equalTo(HttpStatus.SC_OK));
```

Let's look at the logs:

```
[main] DEBUG … - Re-using cached 'basic' auth scheme for http://localhost:8080
[main] DEBUG … - Executing request GET /spring-security-rest-basic-auth/api/foos/1 HTTP/1.1
[main] DEBUG … >> GET /spring-security-rest-basic-auth/api/foos/1 HTTP/1.1
[main] DEBUG … >> Host: localhost:8080
[main] DEBUG … >> Authorization: Basic dXNlcjE6dXNlcjFQYXNz
[main] DEBUG … << HTTP/1.1 200 OK
[main] DEBUG … - Authentication succeeded
```

Everything looks OK:

- the “Basic Authentication” scheme is pre-selected
- the Request is sent with the *Authorization* header
- the Server responds with a *200 OK*
- Authentication succeeds

4. Basic Auth with Raw HTTP Headers

Preemptive Basic Authentication basically means pre-sending the *Authorization* header.

So – instead of going through the rather complex previous example to set it up, **we can take control of this header** and construct it by hand:

```
HttpGet request = new HttpGet(URL_SECURED_BY_BASIC_AUTHENTICATION);
String auth = DEFAULT_USER + “:” + DEFAULT_PASS;
byte[] encodedAuth = Base64.encodeBase64(auth.getBytes(Charset.forName(“ISO-8859-1”)));
String authHeader = “Basic “ + new String(encodedAuth);
request.setHeader(HttpHeaders.AUTHORIZATION, authHeader);

HttpClient client = HttpClientBuilder.create().build();
HttpResponse response = client.execute(request);

int statusCode = response.getStatusLine().getStatusCode();
assertThat(statusCode, equalTo(HttpStatus.SC_OK));
```

Let's make sure this is working correctly:

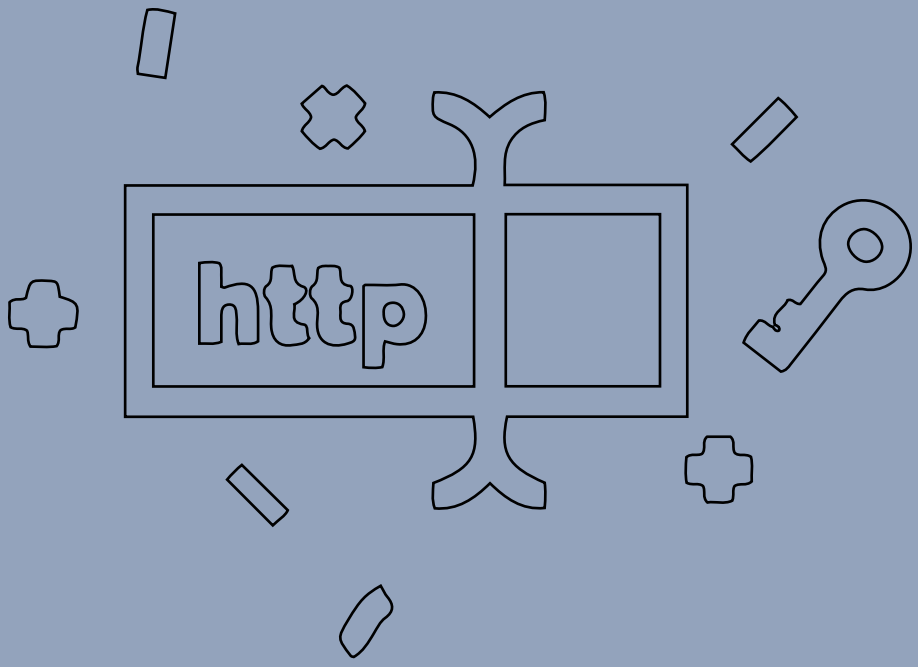
```
[main] DEBUG ... - Auth cache not set in the context
[main] DEBUG ... - Opening connection {}->http://localhost:8080
[main] DEBUG ... - Connecting to localhost/127.0.0.1:8080
[main] DEBUG ... - Executing request GET /spring-security-rest-basic-auth/api/foos/1 HTTP/1.1
[main] DEBUG ... - Proxy auth state: UNCHALLENGED
[main] DEBUG ... - http-outgoing-0 >> GET /spring-security-rest-basic-auth/api/foos/1 HTTP/1.1
[main] DEBUG ... - http-outgoing-0 >> Authorization: Basic dXNlcjE6dXNlcjFQYXNz
[main] DEBUG ... - http-outgoing-0 << HTTP/1.1 200 OK
```

So, even though there is not auth cache, Basic Authentication is still performed correctly and the 200 OK is sent back.

5 . Conclusion

This section illustrated various ways to set up and use basic authentication with the Apache HttpClient 4.

CHAPTER 11



Get the Most
Out of
HttpClient

11: HTTPCLIENT 4 COOKBOOK

1. Overview

This section shows **how to use the Apache HttpClient 4** in a variety of examples and use cases.

The focus is on **HttpClient 4.3.x and above**, so some of the examples may not work with the older versions of the API.

The format of the cookbook is example focused and practical – no extraneous details and explanations necessary.

2. Cookbook

create the http client

```
CloseableHttpClient client = HttpClientBuilder.create().build();
```

send basic GET request

```
instance.execute(new HttpGet("http://www.google.com"));
```

get the Status Code of the HTTP Response

```
CloseableHttpResponse response = instance.execute(new HttpGet("http://www.google.com"));
assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
```

get the Media Type of the response

```
CloseableHttpResponse response = instance.execute(new HttpGet("http://www.google.com"));
String contentMimeType = ContentType.getOrDefault(response.getEntity()).getMimeType();
assertThat(contentMimeType, equalTo(ContentType.TEXT_HTML.getMimeType()));
```

get the body of the response

```
CloseableHttpResponse response = instance.execute(new HttpGet("http://www.google.com"));
String bodyAsString = EntityUtils.toString(response.getEntity());
assertThat(bodyAsString, notNullValue());
```

configure the timeout on a request

```
@Test(expected = SocketTimeoutException.class)
public void givenLowTimeout_whenExecutingRequestWithTimeout_thenException()
   throws ClientProtocolException, IOException {
   RequestConfig requestConfig = RequestConfig.custom()
     .setConnectionRequestTimeout(1000).setConnectTimeout(1000).setSocketTimeout(1000).build();
   HttpGet request = new HttpGet(SAMPLE_URL);
   request.setConfig(requestConfig);
   instance.execute(request);
}
```

configure timeout on the entire client

```
RequestConfig requestConfig = RequestConfig.custom().
   setConnectionRequestTimeout(1000).setConnectTimeout(1000).setSocketTimeout(1000).build();
HttpClientBuilder builder = HttpClientBuilder.create().setDefaultRequestConfig(requestConfig);
```

send a POST request

```
instance.execute(new HttpPost(SAMPLE_URL));
```

add parameters to a request

```
List<NameValuePair> params = new ArrayList<NameValuePair>();
params.add(new BasicNameValuePair("key1", "value1"));
params.add(new BasicNameValuePair("key2", "value2"));
request.setEntity(new UrlEncodedFormEntity(params, Consts.UTF_8));
```

configure how redirect are handled for an HTTP Request

```
CloseableHttpClient instance = HttpClientBuilder.create().disableRedirectHandling().build();
CloseableHttpResponse response = instance.execute(new HttpGet("http://t.co/I5YYd9tddw"));
assertThat(response.getStatusLine().getStatusCode(), equalTo(301));
```

configure the headers for a request

```
HttpGet request = new HttpGet(SAMPLE_URL);
request.addHeader(HttpHeaders.ACCEPT, "application/xml");
response = instance.execute(request);
```

get the headers from response

```
CloseableHttpResponse response = instance.execute(new HttpGet(SAMPLE_URL));
Header[] headers = response.getHeaders(HttpHeaders.CONTENT_TYPE);
assertThat(headers, not(emptyArray()));
```

close/release resources

```
response = instance.execute(new HttpGet(SAMPLE_URL));
try {
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        InputStream instream = entity.getContent();
        instream.close();
    }
} finally {
    response.close();
}
```

3. Go Deep into HttpClient

The HttpClient library is quite a powerful tool if used correctly – if you want to start e**xploring what the client can do** – check out some of the sections:

- HttpClient 4 – Get the Status Code
- HttpClient – Set Custom Header

You can also **dig a lot deeper into the HttpClient** by exploring the entire series.

4 . Conclusion

This format is a bit different from how I usually structure my sections – **I'm publishing some of my internal development cookbooks** on a given topic – on Google Guava, Hamcrest andMockito – and now HttpClient. The goal is to have this information readily available online – and to add to it whenever I run into a new useful example.