Baeldung

Get the Most out of the HTTP Client

Table of Contents

1: Posting with HttpClient

2: HttpClient 4 – Get the Status Code

3: HttpClient Timeout

4: Custom HTTP Header with the HttpClient

5: HttpClient with SSL

6: HttpClient 4 – Send Custom Cookie

7: HttpClient Basic Authentication

1: Posting with HttpClient

1. Overview

In this chapter – **we'll POST with the *HttpClient 4*** – using first authorization, then the fluent *HttpClient* API.

Finally, we'll discuss how to upload a File using *HttpClient*.

First, let's go over a simple example and send a POST request using *HttpClient*.

We'll do a POST with two parameters – "*username*" and "*password*":

```
1.  @Test
2.  public void whenPostRequestUsingHttpClient_thenCorrect()
3.    throws ClientProtocolException, IOException {
4.      CloseableHttpClient client = HttpClients.createDefault();
5.      HttpPost httpPost = new HttpPost("http://www.example.com");
6.
7.      List<NameValuePair> params = new ArrayList<NameValuePair>();
8.      params.add(new BasicNameValuePair("username", "John"));
9.      params.add(new BasicNameValuePair("password", "pass"));
10.     httpPost.setEntity(new UrlEncodedFormEntity(params));
11.
12.     CloseableHttpResponse response = client.execute(httpPost);
13.     assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
14.     client.close();
15. }
```

Note how we used a *List* of *NameValuePair* to include parameters in the POST request.

3. POST with Authorization

Next, let's see how to do a POST with Authentication credentials using the *HttpClient*.

In the following example – we send a POST request to a URL secured with Basic Authentication by adding an Authorization header:

```
@Test
public void whenPostRequestWithAuthorizationUsingHttpClient_thenCorrect()
  throws ClientProtocolException, IOException, AuthenticationException {
    CloseableHttpClient client = HttpClients.createDefault();
    HttpPost httpPost = new HttpPost("http://www.example.com");

    httpPost.setEntity(new StringEntity("test post"));
    UsernamePasswordCredentials creds
      = new UsernamePasswordCredentials("John", "pass");
    httpPost.addHeader(new BasicScheme().authenticate(creds, httpPost, null));

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

Now – let's see how to send a POST request with a JSON body using the *HttpClient*.

In the following example – we're sending some *person* information (*id, name*) as JSON:

```
1.  @Test
2.  public void whenPostJsonUsingHttpClient_thenCorrect()
3.    throws ClientProtocolException, IOException {
4.      CloseableHttpClient client = HttpClients.createDefault();
5.      HttpPost httpPost = new HttpPost("http://www.example.com");
6.
7.      String json = "{"id":1,"name":"John"}";
8.      StringEntity entity = new StringEntity(json);
9.      httpPost.setEntity(entity);
10.     httpPost.setHeader("Accept", "application/json");
11.     httpPost.setHeader("Content-type", "application/json");
12.
13.     CloseableHttpResponse response = client.execute(httpPost);
14.     assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
15.     client.close();
16. }
```

Note how we're using the *StringEntity* to set the body of the request.

We're also setting the *ContentType* header to *application/json* to give the server the necessary information about the representation of the content we're sending.

Next, let's POST with the *HttpClient* Fluent API.

We're going to send a request with two parameters "*username*" and "*password*":

```
1.  @Test
2.  public void whenPostFormUsingHttpClientFluentAPI_thenCorrect()
3.    throws ClientProtocolException, IOException {
4.      HttpResponse response = Request.Post("http://www.example.com").bodyForm(
5.        Form.form().add("username", "John").add("password", "pass").build())
6.        .execute().returnResponse();
7.
8.      assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
9.  }
```

6. POST Multipart Request

Now, let's POST a Multipart Request.

We'll post a *File*, username, and password using *MultipartEntityBuilder*:

```
1.  @Test
2.  public void whenSendMultipartRequestUsingHttpClient_thenCorrect()
3.    throws ClientProtocolException, IOException {
4.      CloseableHttpClient client = HttpClients.createDefault();
5.      HttpPost httpPost = new HttpPost("http://www.example.com");
6.
7.      MultipartEntityBuilder builder = MultipartEntityBuilder.create();
8.      builder.addTextBody("username", "John");
9.      builder.addTextBody("password", "pass");
10.     builder.addBinaryBody("file", new File("test.txt"), ContentType.
11. APPLICATION_OCTET_STREAM, "file.ext");
12.
13.     HttpEntity multipart = builder.build();
14.     httpPost.setEntity(multipart);
15.
16.     CloseableHttpResponse response = client.execute(httpPost);
17.     assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
18.     client.close();
19. }
```

7. Upload a File using *HttpClient*

Next, let's see how to upload a File using the *HttpClient*.

We'll upload the "*test.txt*" file using *MultipartEntityBuilder*:

```
1.  @Test
2.  public void whenUploadFileUsingHttpClient_thenCorrect() throws
3.  ClientProtocolException, IOException {
4.      CloseableHttpClient client = HttpClients.createDefault();
5.      HttpPost httpPost = new HttpPost("http://www.example.com");
6.
7.      MultipartEntityBuilder builder = MultipartEntityBuilder.create();
8.      builder.addBinaryBody("file", new File("test.txt"), ContentType.
9.  APPLICATION_OCTET_STREAM, "file.ext");
10.     HttpEntity multipart = builder.build();
11.     httpPost.setEntity(multipart);
12.
13.     CloseableHttpResponse response = client.execute(httpPost);
14.     assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
15.     client.close();
16. }
```

8. Get File Upload Progress

Finally – let's see how to get the progress of *File* upload using *HttpClient*.

In the following example, we'll extend the *HttpEntityWrapper* to gain visibility into the upload process.

First – here's the upload method:

```
@Test
public void whenGetUploadFileProgressUsingHttpClient_thenCorrect()
  throws ClientProtocolException, IOException {
    CloseableHttpClient client = HttpClients.createDefault();
    HttpPost httpPost = new HttpPost("http://www.example.com");

    MultipartEntityBuilder builder = MultipartEntityBuilder.create();
    builder.addBinaryBody("file", new File("test.txt"), ContentType.
APPLICATION_OCTET_STREAM, "file.ext");
    HttpEntity multipart = builder.build();

    ProgressEntityWrapper.ProgressListener pListener =
      percentage -> assertFalse(Float.compare(percentage, 100) > 0);
    httpPost.setEntity(new ProgressEntityWrapper(multipart, pListener));

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

We'll also add the interface *ProgressListener* that enables us to observe the upload progress:

```
public static interface ProgressListener {
    void progress(float percentage);
}
```

And here's our extended version of *HttpEntityWrapper* "*ProgressEntityWrapper*":

```
1.  public class ProgressEntityWrapper extends HttpEntityWrapper {
2.      private ProgressListener listener;
3.      public ProgressEntityWrapper(HttpEntity entity, ProgressListener listener)
4.  {
5.          super(entity);
6.          this.listener = listener;
7.      }
8.      @Override
9.      public void writeTo(OutputStream outstream) throws IOException {
10.         super.writeTo(new CountingOutputStream(outstream, listener,
11. getContentLength()));
12.     }
13. }
```

And the extended version of *FilterOutputStream* "*CountingOutputStream*":

```
1.  public static class CountingOutputStream extends FilterOutputStream {
2.      private ProgressListener listener;
3.      private long transferred;
4.      private long totalBytes;
5.      public CountingOutputStream(
6.        OutputStream out, ProgressListener listener, long totalBytes) {
7.          super(out);
8.          this.listener = listener;
9.          transferred = 0;
10.         this.totalBytes = totalBytes;
11.     }
12.     @Override
13.     public void write(byte[] b, int off, int len) throws IOException {
14.         out.write(b, off, len);
15.         transferred += len;
16.         listener.progress(getCurrentProgress());
17.     }
18.     @Override
19.     public void write(int b) throws IOException {
20.         out.write(b);
21.         transferred++;
22.         listener.progress(getCurrentProgress());
23.     }
24.     private float getCurrentProgress() {
25.         return ((float) transferred / totalBytes) * 100;
26.     }
27. }
```

Note that:

- When extending *FilterOutputStream* to "*CountingOutputStream*" – we are overriding the *write()* method to count the written (transferred) bytes

- When extending *HttpEntityWrapper* to "*ProgressEntityWrapper*" – we are overriding the *writeTo()* method to use our "*CountingOutputStream*"

9. Conclusion

In this chapter, we illustrated the most common ways to send POST HTTP Requests with the Apache *HttpClient 4*.

We learned how to send a POST request with Authorization, how to post using *HttpClient* fluent API and how to upload a file and track its progress.

The implementation of all these examples and code snippets can be found in the github project.

2: HttpClient 4 – Get the Status Code

1. Overview

In this very quick chapter, we'll see how to **get and validate the Status Code 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 example shows how to **retrieve and work with Status Codes with the Apache *HttpClient 4***.

The implementation of all these examples and code snippets **can be found in** my github project.

3: HttpClient Timeout

1. Overview

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

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**:

```
1.  DefaultHttpClient httpClient = new DefaultHttpClient();
2.
3.  int timeout = 5; // seconds
4.  HttpParams httpParams = httpClient.getParams();
5.  httpParams.setParameter(
6.    CoreConnectionPNames.CONNECTION_TIMEOUT, timeout * 1000);
7.  httpParams.setParameter(
8.    CoreConnectionPNames.SO_TIMEOUT, timeout * 1000);
9.  httpParams.setParameter(
10.   ClientPNames.CONN_MANAGER_TIMEOUT, new Long(timeout * 1000));
```

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

```
1.  DefaultHttpClient httpClient = new DefaultHttpClient();
2.
3.  int timeout = 5; // seconds
4.  HttpParams httpParams = httpClient.getParams();
5.  HttpConnectionParams.setConnectionTimeout(
6.    httpParams, timeout * 1000); // http.connection.timeout
7.  HttpConnectionParams.setSoTimeout(
8.    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**:

```
1.  int timeout = 5;
2.  RequestConfig config = RequestConfig.custom()
3.    .setConnectTimeout(timeout * 1000)
4.    .setConnectionRequestTimeout(timeout * 1000)
5.    .setSocketTimeout(timeout * 1000).build();
6.  CloseableHttpClient client =
7.    HttpClientBuilder.create().setDefaultRequestConfig(config).build();
```

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

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 establishing the connection; 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. However, setting a timeout for obtaining a connection is definitely important in high load scenarios, which is why the third parameter shouldn't be ignored.

After configuring it, we can now use the client to perform HTTP requests:

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

With the previously defined client, **the connection to the host will time out in 5 seconds**. Also, 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 an *org.apache.http.conn.ConnectTimeoutException* being thrown, while socket timeout will result in a *java.net.SocketTimeoutException*.

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:

```
1.  HttpGet getMethod = new HttpGet(
2.    "http://localhost:8080/httpclient-simple/api/bars/1");
3.
4.  int hardTimeout = 5; // seconds
5.  TimerTask task = new TimerTask() {
6.      @Override
7.      public void run() {
8.          if (getMethod != null) {
9.              getMethod.abort();
10.         }
11.     }
12. };
13. new Timer(true).schedule(task, hardTimeout * 1000);
14.
15. HttpResponse response = httpClient.execute(getMethod);
16. System.out.println(
17.   "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.

It's 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 a 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. What's more – this will happen completely transparently for the client (unless you have your log configured at the DEBUG level).

Here's a simple example you can run and replicate this issue:

```
1.  int timeout = 3;
2.  RequestConfig config = RequestConfig.custom().
3.    setConnectTimeout(timeout * 1000).
4.    setConnectionRequestTimeout(timeout * 1000).
5.    setSocketTimeout(timeout * 1000).build();
6.  CloseableHttpClient client = HttpClientBuilder.create()
7.    .setDefaultRequestConfig(config).build();
8.
9.  HttpGet request = new HttpGet("http://www.google.com:81");
10. 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 chapter 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.

The implementation of these examples can be found in the GitHub project.

4: Custom HTTP Header with the HttpClient

1. Overview

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

2. 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'll use the *setHeader* method – on the builder**:

```
1. HttpClient client = HttpClients.custom().build();
2. HttpUriRequest request = RequestBuilder.get()
3.   .setUri(SAMPLE_URL)
4.   .setHeader(HttpHeaders.CONTENT_TYPE, "application/json")
5.   .build();
6. client.execute(request);
```

3. Set Header on Request – Before 4.3

In versions pre 4.3 of *HttpClient*, **we can set any custom header on a request with a simple *setHeader* call on the request**:

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

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

4. Set Default Header on the Client

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

```
1.  Header header = new BasicHeader(HttpHeaders.CONTENT_TYPE, "application/
2.  json");
3.  List<Header> headers = Lists.newArrayList(header);
4.  HttpClient client = HttpClients.custom().setDefaultHeaders(headers).build();
5.  HttpUriRequest request = RequestBuilder.get().setUri(SAMPLE_URL).build();
6.  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 chapter illustrated how to add an HTTP header to one or all requests sent via the Apache *HttpClient*.

The implementation of all these examples and code snippets can be found in the GitHub project.

5: HttpClient with SSL

1. Overview

This chapter 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.

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

```
1.  public class RestClientLiveManualTest {
2.
3.      @Test(expected = SSLPeerUnverifiedException.class)
4.      public void whenHttpsUrlIsConsumed_thenException()
5.        throws ClientProtocolException, IOException {
6.
7.          CloseableHttpClient httpClient = HttpClients.createDefault();
8.          String urlOverHttps
9.            = “https://localhost:8082/httpclient-simple”;
10.         HttpGet getMethod = new HttpGet(urlOverHttps);
11.
12.         HttpResponse response = httpClient.execute(getMethod);
13.         assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
14.     }
15. }
```

The exact failure is:

```
1.  javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated
2.     at sun.security.ssl.SSLSessionImpl.getPeerCertificates(SSLSessionImpl.java:397)
3.     at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:126)
4.     ...
```

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:

```
1.  @Test
2.  public final void givenAcceptingAllCertificates_whenHttpsUrlIsConsumed_thenOk()
3.    throws GeneralSecurityException {
4.      HttpComponentsClientHttpRequestFactory requestFactory = new
5.  HttpComponentsClientHttpRequestFactory();
6.      CloseableHttpClient httpClient = (CloseableHttpClient) requestFactory.
7.  getHttpClient();
8.
9.      TrustStrategy acceptingTrustStrategy = (cert, authType) -> true;
10.     SSLSocketFactory sf = new SSLSocketFactory(acceptingTrustStrategy, ALLOW_
11. ALL_HOSTNAME_VERIFIER);
12.     httpClient.getConnectionManager().getSchemeRegistry().register(new
13. Scheme("https", 8443, sf));
14.
15.     ResponseEntity<String> response = new RestTemplate(requestFactory).
16.       exchange(urlOverHttps, HttpMethod.GET, null, String.class);
17.     assertThat(response.getStatusCode().value(), equalTo(200));
18. }
```

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. Configure SSL – Accept All (HttpClient 4.4 and above)

With the new *HTTPClient*, now we have an enhanced, redesigned default SSL hostname verifier. Also with the introduction of *SSLConnectionSocketFactory* and RegistryBuilder, it's easy to build *SSLSocketFactory*. So we can write the above test case like:

```
1.  @Test
2.  public final void givenAcceptingAllCertificates_whenHttpsUrlIsConsumed_thenOk()
3.    throws GeneralSecurityException {
4.      TrustStrategy acceptingTrustStrategy = (cert, authType) -> true;
5.      SSLContext sslContext = SSLContexts.custom().loadTrustMaterial(null,
6.  acceptingTrustStrategy).build();
7.      SSLConnectionSocketFactory sslsf = new
8.  SSLConnectionSocketFactory(sslContext,
9.        NoopHostnameVerifier.INSTANCE);
10.
11.     Registry<ConnectionSocketFactory> socketFactoryRegistry =
12.       RegistryBuilder.<ConnectionSocketFactory> create()
13.       .register("https", sslsf)
14.       .register("http", new PlainConnectionSocketFactory())
15.       .build();
16.
17.     BasicHttpClientConnectionManager connectionManager =
18.       new BasicHttpClientConnectionManager(socketFactoryRegistry);
19.     CloseableHttpClient httpClient = HttpClients.custom().
20. setSSLSocketFactory(sslsf)
21.       .setConnectionManager(connectionManager).build();
22.
23.     HttpComponentsClientHttpRequestFactory requestFactory =
24.       new HttpComponentsClientHttpRequestFactory(httpClient);
25.     ResponseEntity<String> response = new RestTemplate(requestFactory)
26.       .exchange(urlOverHttps, HttpMethod.GET, null, String.class);
27.     assertThat(response.getStatusCode().value(), equalTo(200));
28. }
```

Now that we have seen how to configure a raw *HttpClient* with SSL support, let's take a look at a higher 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/httpclient-simple/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:

```
@Test
public void givenAcceptingAllCertificates_whenHttpsUrlIsConsumed_
thenException()
  throws GeneralSecurityException {
    HttpComponentsClientHttpRequestFactory requestFactory
      = new HttpComponentsClientHttpRequestFactory();
    DefaultHttpClient httpClient
      = (DefaultHttpClient) requestFactory.getHttpClient();
    TrustStrategy acceptingTrustStrategy = (cert, authType) -> true
    SSLSocketFactory sf = new SSLSocketFactory(
      acceptingTrustStrategy, ALLOW_ALL_HOSTNAME_VERIFIER);
    httpClient.getConnectionManager().getSchemeRegistry()
      .register(new Scheme("https", 8443, sf));

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

As we 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.

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

```
1.  @Test
2.  public void givenAcceptingAllCertificatesUsing4_4_whenUsingRestTemplate_
3.  thenCorrect()
4.  throws ClientProtocolException, IOException {
5.      CloseableHttpClient httpClient
6.        = HttpClients.custom()
7.          .setSSLHostnameVerifier(new NoopHostnameVerifier())
8.          .build();
9.      HttpComponentsClientHttpRequestFactory requestFactory
6.        = new HttpComponentsClientHttpRequestFactory();
7.      requestFactory.setHttpClient(httpClient);
8.
9.      ResponseEntity<String> response
6.        = new RestTemplate(requestFactory).exchange(
7.        urlOverHttps, HttpMethod.GET, null, String.class);
8.      assertThat(response.getStatusCode().value(), equalTo(200));
9.  }
```

This chapter 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.

The implementation of these examples can be found in the GitHub project.

6: HttpClient 4 – Send Custom Cookie

1. Overview

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

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:

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

Then, **we can set up this cookie store on the *HttpClient* using the *setDefaultCookieStore()* method** and send the request:

```
1. @Test
2. public void whenSettingCookiesOnTheHttpClient_thenCookieSentCorrectly()
3.   throws ClientProtocolException, IOException {
4.     BasicCookieStore cookieStore = new BasicCookieStore();
5.     BasicClientCookie cookie = new BasicClientCookie("JSESSIONID", "1234");
6.     cookie.setDomain(".github.com");
7.     cookie.setPath("/");
8.     cookieStore.addCookie(cookie);
9.     HttpClient client = HttpClientBuilder.create().
10. setDefaultCookieStore(cookieStore).build();
11.
12.    final HttpGet request = new HttpGet("http://www.github.com");
13.
14.    response = client.execute(request);
15.    assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
16. }
```

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*:

```
1.  @Test
2.  public void givenUsingDeprecatedApi_whenSettingCookiesOnTheHttpClient_
3.  thenCorrect()
4.    throws ClientProtocolException, IOException {
5.      BasicCookieStore cookieStore = new BasicCookieStore();
6.      BasicClientCookie cookie = new BasicClientCookie("JSESSIONID", "1234");
7.      cookie.setDomain(".github.com");
8.      cookie.setPath("/");
9.      cookieStore.addCookie(cookie);
10.     DefaultHttpClient client = new DefaultHttpClient();
11.     client.setCookieStore(cookieStore);
12.
13.     HttpGet request = new HttpGet("http://www.github.com");
14.
15.     response = client.execute(request);
16.     assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
17. }
```

Other than the way the client is built, there's 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 using the *HttpContext* class:

```
@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));
}
```

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

```
1.  @Test
2.  public void whenSettingCookiesOnARequest_thenCorrect()
3.    throws ClientProtocolException, IOException {
4.      instance = HttpClientBuilder.create().build();
5.      HttpGet request = new HttpGet("http://www.github.com");
6.      request.setHeader("Cookie", "JSESSIONID=1234");
7.
8.      response = instance.execute(request);
9.
10.     assertThat(response.getStatusLine().getStatusCode(), equalTo(200));
11. }
```

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 chapter 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. Instead, it's controlling the client side manually at a low level.

The implementation of all these examples and code snippets can be found in my GitHub project.

7: HttpClient Basic Authentication

1. Overview

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

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

```
1.  CredentialsProvider provider = new BasicCredentialsProvider();
2.  UsernamePasswordCredentials credentials
3.   = new UsernamePasswordCredentials("user1", "user1Pass");
4.  provider.setCredentials(AuthScope.ANY, credentials);
5.
6.  HttpClient client = HttpClientBuilder.create()
7.    .setDefaultCredentialsProvider(provider)
8.    .build();
9.
10. HttpResponse response = client.execute(
11.   new HttpGet(URL_SECURED_BY_BASIC_AUTHENTICATION));
12. int statusCode = response.getStatusLine()
8.    .getStatusCode();
9.
10. assertThat(statusCode, equalTo(HttpStatus.SC_OK));
```

As we 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 look at the logs:

```
1.  # ... request is sent with no credentials
2.  [main] DEBUG ... - Authentication required
3.  [main] DEBUG ... - localhost:8080 requested authentication
4.  [main] DEBUG ... - Authentication schemes in the order of preference:
5.    [negotiate, Kerberos, NTLM, Digest, Basic]
6.  [main] DEBUG ... - Challenge for negotiate authentication scheme not
7.  available
8.  [main] DEBUG ... - Challenge for Kerberos authentication scheme not
9.  available
10. [main] DEBUG ... - Challenge for NTLM authentication scheme not available
11. [main] DEBUG ... - Challenge for Digest authentication scheme not
12. available
13. [main] DEBUG ... - Selected authentication options: [BASIC]
14. # ... 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. Instead, 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 negotiation from the previous example is no longer necessary – **Basic Authentication is already chosen**:

```
1.  HttpHost targetHost = new HttpHost("localhost", 8082, "http");
2.  CredentialsProvider credsProvider = new BasicCredentialsProvider();
3.  credsProvider.setCredentials(AuthScope.ANY,
4.    new UsernamePasswordCredentials(DEFAULT_USER, DEFAULT_PASS));
5.
6.  AuthCache authCache = new BasicAuthCache();
7.  authCache.put(targetHost, new BasicScheme());
8.
9.  // Add AuthCache to the execution context
10. HttpClientContext context = HttpClientContext.create();
11. context.setCredentialsProvider(credsProvider);
12. context.setAuthCache(authCache);
```

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

```
1.  HttpClient client = HttpClientBuilder.create().build();
2.  response = client.execute(
3.    new HttpGet(URL_SECURED_BY_BASIC_AUTHENTICATION), context);
4.
5.  int statusCode = response.getStatusLine().getStatusCode();
6.  assertThat(statusCode, equalTo(HttpStatus.SC_OK));
```

Let's look at the logs:

```
[main] DEBUG ... - Re-using cached 'basic' auth scheme for http://
localhost:8082
[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:8082
[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

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**:

```
1.  HttpGet request = new HttpGet(URL_SECURED_BY_BASIC_AUTHENTICATION);
2.  String auth = DEFAULT_USER + ":" + DEFAULT_PASS;
3.  byte[] encodedAuth = Base64.encodeBase64(
4.    auth.getBytes(StandardCharsets.ISO_8859_1));
5.  String authHeader = "Basic " + new String(encodedAuth);
6.  request.setHeader(HttpHeaders.AUTHORIZATION, authHeader);
7.
8.  HttpClient client = HttpClientBuilder.create().build();
9.  HttpResponse response = client.execute(request);
10.
11. int statusCode = response.getStatusLine().getStatusCode();
12. assertThat(statusCode, equalTo(HttpStatus.SC_OK));
```

Let's make sure this is working correctly:

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

So, even though there is no auth cache, **Basic Authentication still works correctly and we receive *200 OK***.

5. Conclusion

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

As always, the code presented is available over on GitHub.