Apache Http client

tutorialspoint

SIMPLY EASY LEARNING

About the Tutorial

Http client is a transfer library. It resides on the client side, sends and receives Http messages. It provides up to date, feature-rich, and an efficient implementation which meets the recent Http standards.

Audience

This tutorial has been prepared for the beginners to help them understand the concepts of Apache HttpClient library.

Prerequisites

Before you start practicing various types of examples given in this reference, we assume that you already have knowledge in Java programming.

And, having knowledge of Http protocol concepts helps in understanding this tutorial better.

Copyright & Disclaimer

Table of Contents

1. HttpClient — Overview

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. This is the foundation for data communication for the World Wide Web (i.e., Internet) since 1990. HTTP is a generic and stateless protocol which can be used for other purposes as well using extensions of its request methods, error codes, and headers.

Basically, HTTP is a TCP/IP based communication protocol, that is used to deliver data (HTML files, image files, query results, etc.) on the World Wide Web. The default port is TCP 80, but other ports can be used as well. It provides a standardized way for computers to communicate with each other. HTTP specification defines how clients' request data will be constructed and sent to the server, and how the servers respond to these requests.

What is Http Client?

Http client is a transfer library, it resides on the client side, sends and receives HTTP messages. It provides up to date, feature-rich and, efficient implementation which meets the recent HTTP standards.

In addition to this using client library, one can build HTTP based applications such as web browsers, web service clients, etc.

Features of Http Client

Following are the prominent features of Http client:

- HttpClient library implements all the available HTTP methods.
- HttpClient library provides APIs to secure the requests using the Secure Socket Layer protocol.
- Using HttpClient, you can establish connections using proxies.
- You can authenticate connections using authentication schemes such as Basic, Digest, NTLMv1, NTLMv2, NTLM2 Session etc.
- HttpClient library supports sending requests through multiple threads. It manages multiple connections established from various threads using **ClientConnectionPoolManager**.
- Using Apache HttpClient library, you can set connection timeouts.

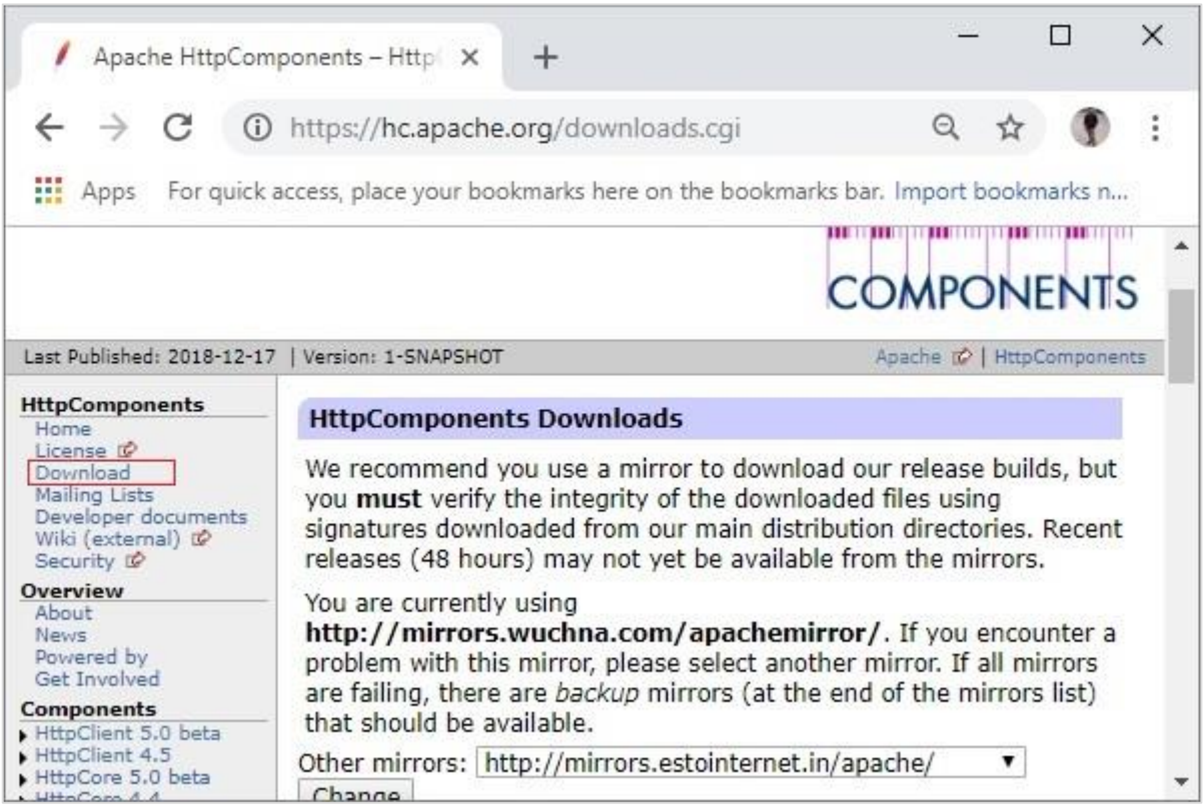
2. HttpClient — Environment Setup

In this chapter, we will explain how to set an environment for HttpClient in Eclipse IDE. Before proceeding with the installation, make sure that you already have Eclipse installed in your system. If not, download and install Eclipse.

For more information on Eclipse, please refer to our Eclipse Tutorial.

Step 1: Download the dependency JAR file

Open the official homepage of the HttpClient (components) website and go to the download page.



HttpComponents Downloads

We recommend you use a mirror to download our release builds, but you **must** verify the integrity of the downloaded files using signatures downloaded from our main distribution directories. Recent releases (48 hours) may not yet be available from the mirrors.

You are currently using **http://mirrors.wuchna.com/apachemirror/**. If you encounter a problem with this mirror, please select another mirror. If all mirrors are failing, there are *backup* mirrors (at the end of the mirrors list) that should be available.

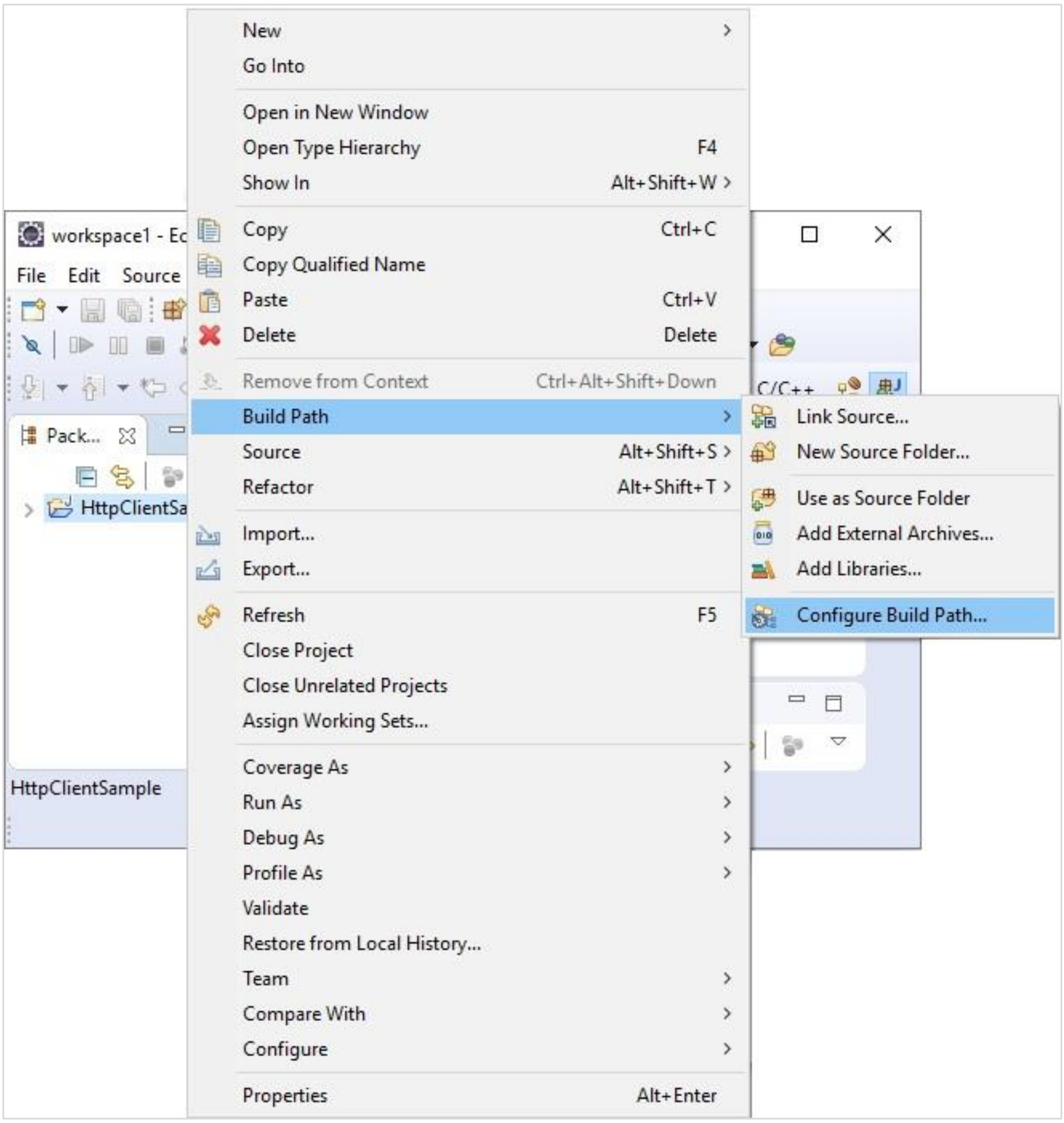
Other mirrors: http://mirrors.estointernet.in/apache/

Then, download the latest stable version of ***HttpClient.*** Here, throughout the tutorial, we are using the version 4.5.6 hence download the file **4.5.6.zip.**

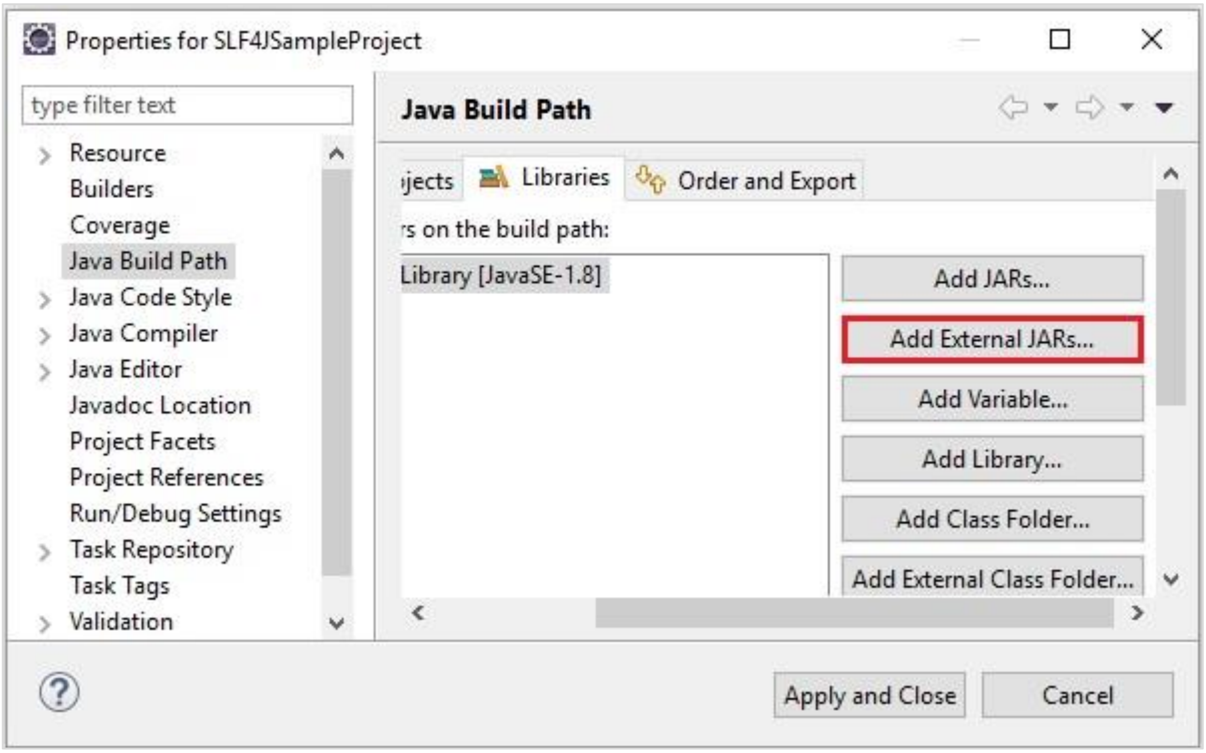
Within the downloaded folder, you will find a folder named **lib** and this contains the required Jar files that are to be added in the classpath of your project, to work with HttpClient.

Step 2: Create a project and set build path

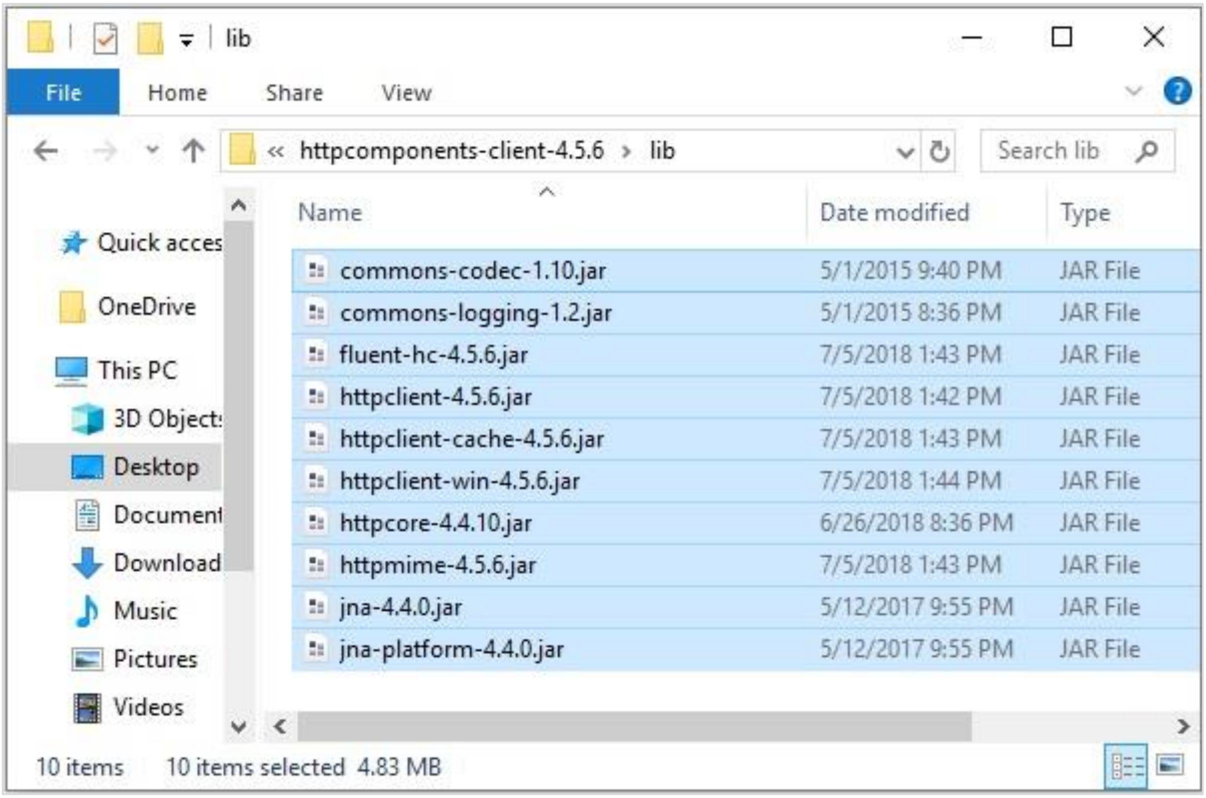
Open eclipse and create a sample project. Right click on the project select the option **Build Path -> Configure Build Path** as shown below.



In the **Java Build Path** frame in the **Libraries** tab, click on **Add External JARs.**



And select all the jar files in the lib folder and, click on **Apply and Close.**



Name	Date modified	Type
commons-codec-1.10.jar	5/1/2015 9:40 PM	JAR File
commons-logging-1.2.jar	5/1/2015 8:36 PM	JAR File
fluent-hc-4.5.6.jar	7/5/2018 1:43 PM	JAR File
httpclient-4.5.6.jar	7/5/2018 1:42 PM	JAR File
httpclient-cache-4.5.6.jar	7/5/2018 1:43 PM	JAR File
httpclient-win-4.5.6.jar	7/5/2018 1:44 PM	JAR File
httpcore-4.4.10.jar	6/26/2018 8:36 PM	JAR File
httpmime-4.5.6.jar	7/5/2018 1:43 PM	JAR File
jna-4.4.0.jar	5/12/2017 9:55 PM	JAR File
jna-platform-4.4.0.jar	5/12/2017 9:55 PM	JAR File

You are all set to work with HttpClient library in eclipse.

3. HttpClient — Http Get Request

The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.

The HttpClient API provides a class named **HttpGet** which represents the get request method.

Follow the steps given below to send a get request using HttpClient library.

Step 1: Create a HttpClient object

The **createDefault()** method of the **HttpClients** class returns a **CloseableHttpClient** object, which is the base implementation of the **HttpClient** interface.

Using this method, create an HttpClient object as shown below:

```
CloseableHttpClient httpclient = HttpClients.createDefault();
```

Step 2: Create an HttpGet Object

The **HttpGet** class represents the HTTPGET request. Which retrieves the information of the given server using a URI.

Create a HTTP GET request by instantiating this class. The constructor of this class accepts a String value representing the URI.

```
HttpGet httpget = new HttpGet("http://www.tutorialspoint.com/");
```

Step 3: Execute the Get Request

The **execute()** method of the **CloseableHttpClient** class accepts a HttpUriRequest (interface) object (i.e. HttpGet, HttpPost, HttpPut, HttpHead etc.) and returns a response object.

Execute the request using this method as shown below:

```
HttpResponse httpresponse = httpclient.execute(httpget);
```

Example

Following is an example which demonstrates the execution of the HTTP GET request using HttpClient library.

```
import java.util.Scanner;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;

public class HttpGetExample {

   public static void main(String args[]) throws Exception{

      //Creating a HttpClient object
      CloseableHttpClient httpclient = HttpClients.createDefault();

      //Creating a HttpGet object
      HttpGet httpget = new HttpGet("https://www.tutorialspoint.com/ ");

      //Printing the method used
      System.out.println("Request Type: "+httpget.getMethod());

      //Executing the Get request
      HttpResponse httpresponse = httpclient.execute(httpget);

      Scanner sc = new Scanner(httpresponse.getEntity().getContent());

      //Printing the status line
      System.out.println(httpresponse.getStatusLine());
      while(sc.hasNext()){
         System.out.println(sc.nextLine());
      }

   }

}
```

Output

The above program generates the following output:

```
Request Type: GET
<!DOCTYPE html>
<!--[if IE 8]><html class="ie ie8"> <![endif]-->
<!--[if IE 9]><html class="ie ie9"> <![endif]-->
<!--[if gt IE 9]><!-->
<html lang="en-US"> <!--<![endif]-->
<head>
<!-- Basic -->
<meta charset="utf-8">
<title>Parallax Scrolling, Java Cryptography, YAML, Python Data Science, Java
i18n, GitLab, TestRail, VersionOne, DBUtils, Common CLI, Seaborn, Ansible,
LOLCODE, Current Affairs 2018,  Apache Commons Collections</title>
<meta name="Description" content="Parallax Scrolling, Java Cryptography, YAML,
Python Data Science, Java i18n, GitLab, TestRail, VersionOne, DBUtils, Common
CLI, Seaborn, Ansible, LOLCODE, Current Affairs 2018, Intellij Idea, Apache
Commons Collections, Java 9, GSON, TestLink, Inter Process Communication (IPC),
Logo, PySpark, Google Tag Manager, Free IFSC Code, SAP Workflow"/>
<meta name="Keywords" content="Python Data Science, Java i18n, GitLab,
TestRail, VersionOne, DBUtils, Common CLI, Seaborn, Ansible, LOLCODE, Gson,
TestLink, Inter Process Communication (IPC), Logo"/>
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width,initial-scale=1.0,user-
scalable=yes">
<link href="https://cdn.muicss.com/mui-0.9.39/extra/mui-rem.min.css"
rel="stylesheet" type="text/css" />
<link rel="stylesheet" href="/questions/css/home.css?v=3" />
<script src="/questions/js/jquery.min.js"></script>
<script src="/questions/js/fontawesome.js"></script>
<script src="https://cdn.muicss.com/mui-0.9.39/js/mui.min.js"></script>
</head>
. . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . .


</script>
</body>
</html>
```

4. HttpClient — Http Post Request

A POST request is used to send data to the server; for example, customer information, file upload, etc., using HTML forms.

The HttpClient API provides a class named **HttpPost** which represents the POST request.

Follow the steps given below to send a HTTP POST request using HttpClient library.

Step 1: Create an HttpClient Object

The **createDefault()** method of the **HttpClients** class returns an object of the class **CloseableHttpClient,** which is the base implementation of the **HttpClient** interface.

Using this method, create an HttpClient object.

```
CloseableHttpClient httpClient = HttpClients.createDefault();
```

Step 2: Create HttpPost Object

The **HttpPost** class represents the **HTTP POST** request. This sends required data and retrieves the information of the given server using a URI.

Create this request by instantiating the **HttpPost** class and pass a string value representing the URI, as a parameter to its constructor.

```
HttpGet httpGet = new HttpGet("http://www.tutorialspoint.com/");
```

Step 3: Execute the Get Request

The **execute()** method of the CloseableHttpClient object accepts a HttpUriRequest (interface) object (i.e. HttpGet, HttpPost, HttpPut, HttpHead etc.) and returns a response object.

```
HttpResponse httpResponse = httpclient.execute(httpget);
```

Example

Following is an example which demonstrates the execution of the HTTP POST request using HttpClient library.

```
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;


public class HttpPostExample {

   public static void main(String args[]) throws Exception{

       //Creating a HttpClient object
       CloseableHttpClient httpclient = HttpClients.createDefault();

       //Creating a HttpGet object
       HttpPost httppost = new HttpPost("https://www.tutorialspoint.com/");

       //Printing the method used
       System.out.println("Request Type: "+httppost.getMethod());

       //Executing the Get request
       HttpResponse httpresponse = httpclient.execute(httppost);

       Scanner sc = new Scanner(httpresponse.getEntity().getContent());

       //Printing the status line
       System.out.println(httpresponse.getStatusLine());
       while(sc.hasNext()){
          System.out.println(sc.nextLine());
        }
   }
}
```

Output

The above program generates the following output.

```
Request Type: POST
<!DOCTYPE html>
<!--[if IE 8]><html class="ie ie8"> <![endif]-->
<!--[if IE 9]><html class="ie ie9"> <![endif]-->
<!--[if gt IE 9]><!-->
```

```
<html lang="en-US"> <!--<![endif]-->

<head>

<!-- Basic -->

<meta charset="utf-8">

<title>Parallax Scrolling, Java Cryptography, YAML, Python Data Science, Java
i18n, GitLab, TestRail, VersionOne, DBUtils, Common CLI, Seaborn, Ansible,
LOLCODE, Current Affairs 2018,  Apache Commons Collections</title>

<meta name="Description" content="Parallax Scrolling, Java Cryptography, YAML,
Python Data Science, Java i18n, GitLab, TestRail, VersionOne, DBUtils, Common
CLI, Seaborn, Ansible, LOLCODE, Current Affairs 2018, Intellij Idea, Apache
Commons Collections, Java 9, GSON, TestLink, Inter Process Communication (IPC),
Logo, PySpark, Google Tag Manager, Free IFSC Code, SAP Workflow"/>

<meta name="Keywords" content="Python Data Science, Java i18n, GitLab,
TestRail, VersionOne, DBUtils, Common CLI, Seaborn, Ansible, LOLCODE, Gson,
TestLink, Inter Process Communication (IPC), Logo"/>

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width,initial-scale=1.0,user-
scalable=yes">

<link href="https://cdn.muicss.com/mui-0.9.39/extra/mui-rem.min.css"
rel="stylesheet" type="text/css" />

<link rel="stylesheet" href="/questions/css/home.css?v=3" />

<script src="/questions/js/jquery.min.js"></script>

<script src="/questions/js/fontawesome.js"></script>

<script src="https://cdn.muicss.com/mui-0.9.39/js/mui.min.js"></script>

</head>

. . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . .


</script>

</body>

</html>
```

5. HttpClient — Response Handlers

Processing the HTTP responses using the response handlers is recommended. In this chapter, we are going to discuss how to create response handlers and how to use them to process a response.

If you use the response handler, all the HTTP connections will be released automatically.

Creating a response handler

The HttpClient API provides an interface known as **ResponseHandler** in the package **org.apache.http.client.** In order to create a response handler, implement this interface and override its **handleResponse()** method.

Every response has a status code and if the status code is in between 200 and 300, that means the action was successfully received, understood, and accepted. Therefore, in our example, we will handle the entities of the responses with such status codes.

Executing the request using response handler

Follow the steps given below to execute the request using a response handler.

Step 1: Create an HttpClient Object

The **createDefault()** method of the **HttpClients** class returns an object of the class **CloseableHttpClient,** which is the base implementation of the **HttpClient** interface. Using this method create an HttpClient object.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
```

Step 2: Instantiate the Response Handler

Instantiate the response handler object created above using the following line of code:

```
ResponseHandler<String> responseHandler = new MyResponseHandler();
```

Step 3: Create a HttpGet Object

The **HttpGet** class represents the HTTP GET request which retrieves the information of the given server using a URI.

Create an HttpGet request by instantiating the HttpGet class and by passing a string representing the URI as a parameter to its constructor.

```
ResponseHandler<String> responseHandler = new MyResponseHandler();
```

Step 4: Execute the Get request using response handler

The **CloseableHttpClient** class has a variant of **execute()** method which accepts two objects **ResponseHandler** and HttpUriRequest, and returns a response object.

```
String httpResponse = httpclient.execute(httpget, responseHandler);
```

Example

Following example demonstrates the usage of response handlers.

```
import java.io.IOException;


import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.ResponseHandler;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;


class MyResponseHandler implements ResponseHandler<String>{
   public String handleResponse(final HttpResponse response) throws IOException{
      //Get the status of the response
      int status = response.getStatusLine().getStatusCode();
         if (status >= 200 && status < 300) {
            HttpEntity entity = response.getEntity();
            if(entity == null){
                  return "";
            }else{
                  return EntityUtils.toString(entity);
            }


         }else{
             return ""+status;
         }
```

```
    }
}
```

```
public class ResponseHandlerExample {
   public static void main(String args[]) throws Exception{
      //Create an HttpClient object
      CloseableHttpClient httpclient = HttpClients.createDefault();

      //instantiate the response handler
      ResponseHandler<String> responseHandler = new MyResponseHandler();

      //Create an HttpGet object
      HttpGet httpget = new HttpGet("http://www.tutorialspoint.com/");

      //Execute the Get request by passing the response handler object and HttpGet object
      String httpresponse = httpclient.execute(httpget, responseHandler);

      System.out.println(httpresponse);

   }

}
```

Output

The above programs generate the following output:

```
<!DOCTYPE html>
<!--[if IE 8]><html class="ie ie8"> <![endif]-->
<!--[if IE 9]><html class="ie ie9"> <![endif]-->
<!--[if gt IE 9]><!-->
<html lang="en-US"> <!--<![endif]-->
<head>
<!-- Basic -->
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
<meta name="viewport" content="width=device-width,initial-scale=1.0,user-
scalable=yes">

<link href="https://cdn.muicss.com/mui-0.9.39/extra/mui-rem.min.css"
rel="stylesheet" type="text/css" />

<link rel="stylesheet" href="/questions/css/home.css?v=3" />

<script src="/questions/js/jquery.min.js"></script>

<script src="/questions/js/fontawesome.js"></script>

<script src="https://cdn.muicss.com/mui-0.9.39/js/mui.min.js"></script>

</head>

. . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . .

<script>

window.dataLayer = window.dataLayer || [];

function gtag(){dataLayer.push(arguments);}

gtag('js', new Date());

gtag('config', 'UA-232293-17');

</script>

</body>
```

6. HttpClient — Closing Connection

If you are processing HTTP responses manually instead of using a response handler, you need to close all the http connections by yourself. This chapter explains how to close the connections manually.

While closing HTTP connections manually follow the steps given below:

Step 1: Create an HttpClient object

The **createDefault()** method of the **HttpClients** class returns an object of the class **CloseableHttpClient,** which is the base implementation of the HttpClient interface.

Using this method, create an **HttpClient** object as shown below:

```
CloseableHttpClient httpClient = HttpClients.createDefault();
```

Step 2: Start a try-finally block

Start a try-finally block, write the remaining code in the programs in the try block and close the CloseableHttpClient object in the finally block.

```
CloseableHttpClient httpClient = HttpClients.createDefault();
try{
   //Remaining code . . . . . . . . . . . . . . .  .
}finally{
   httpClient.close();
}
```

Step 3: Create a HttpGet object

The **HttpGet** class represents the HTTP GET request which retrieves the information of the given server using a URI.

Create a HTTP GET request by instantiating the HttpGet class by passing a string representing the URI.

```
HttpGet httpGet = new HttpGet("http://www.tutorialspoint.com/");
```

Step 4: Execute the Get request

The **execute()** method of the **CloseableHttpClient** object accepts a **HttpUriRequest** (interface) object (i.e. HttpGet, HttpPost, HttpPut, HttpHead etc.) and returns a response object.

Execute the request using the given method:

```
HttpResponse httpResponse = httpclient.execute(httpGet);
```

Step 5: Start another (nested) try-finally

Start another try-finally block (nested within the previous try-finally), write the remaining code in the programs in this try block and close the HttpResponse object in the finally block.

```
CloseableHttpClient httpclient = HttpClients.createDefault();
   try{
      . . . . . . .
      . . . . . . .
      CloseableHttpResponse httpresponse = httpclient.execute(httpget);
      try{
         . . . . . . .
         . . . . . . .
      }finally{
         httpresponse.close();
      }
   }finally{
      httpclient.close();
   }
```

Example

Whenever you create/obtain objects such as request, response stream, etc., start a try-finally block in the next line, write the remaining code within the try and close the respective object in the finally block as demonstrated in the following program**:**

```
import java.util.Scanner;


import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;

public class CloseConnectionExample {

   public static void main(String args[])throws Exception{

       //Create an HttpClient object
       CloseableHttpClient httpclient = HttpClients.createDefault();

       try{
         //Create an HttpGet object
          HttpGet httpget = new HttpGet("http://www.tutorialspoint.com/");

          //Execute the Get request
          CloseableHttpResponse httpresponse = httpclient.execute(httpget);

          try{
          Scanner sc = new Scanner(httpresponse.getEntity().getContent());
            while(sc.hasNext()){
                   System.out.println(sc.nextLine());
             }
          }finally{
             httpresponse.close();
          }
       }finally{
              httpclient.close();
       }


   }

}
```

Output

On executing the above program, the following output is generated:

```
<!DOCTYPE html>
<!--[if IE 8]><html class="ie ie8"> <![endif]-->
<!--[if IE 9]><html class="ie ie9"> <![endif]-->
<!--[if gt IE 9]><!-->
<html lang="en-US"> <!--<![endif]-->
<head>
<!-- Basic -->
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width,initial-scale=1.0,user-
scalable=yes">
<link href="https://cdn.muicss.com/mui-0.9.39/extra/mui-rem.min.css"
rel="stylesheet" type="text/css" />
<link rel="stylesheet" href="/questions/css/home.css?v=3" />
<script src="/questions/js/jquery.min.js"></script>
<script src="/questions/js/fontawesome.js"></script>
<script src="https://cdn.muicss.com/mui-0.9.39/js/mui.min.js"></script>
</head>
. . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . .



<script>
window.dataLayer = window.dataLayer || [];
function gtag(){dataLayer.push(arguments);}
gtag('js', new Date());
gtag('config', 'UA-232293-17');
</script>
</body>
</html>
```

7. HttpClient — Aborting a Request

You can abort the current HTTP request using the **abort()** method, i.e., after invoking this method, on a particular request, execution of it will be aborted.

If this method is invoked after one execution, responses of that execution will not be affected and the subsequent executions will be aborted.

Example

If you observe the following example, we have created a HttpGet request, printed the request format used using the **getMethod().**

Then, we have carried out another execution with the same request. Printed the status line using the 1st execution again. Finally, printed the status line of the second execution.

As discussed, the responses of the 1st execution (execution before abort method) are printed (including the second status line that is written after the abort method) and, all the subsequent executions of the current request after the abort method are failed invoking an exception.

```
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;

public class HttpGetExample {

   public static void main(String args[]) throws Exception{

        //Creating an HttpClient object
        CloseableHttpClient httpclient = HttpClients.createDefault();

        //Creating an HttpGet object
        HttpGet httpget = new HttpGet("http://www.tutorialspoint.com/");

        //Printing the method used
        System.out.println(httpget.getMethod());

        //Executing the Get request
        HttpResponse httpresponse = httpclient.execute(httpget);

        //Printing the status line
        System.out.println(httpresponse.getStatusLine());

        httpget.abort();
        System.out.println(httpresponse.getEntity().getContentLength());
        //Executing the Get request
        HttpResponse httpresponse2 = httpclient.execute(httpget);
        System.out.println(httpresponse2.getStatusLine());

    }

}
```

Output

On executing, the above program generates the following output:

```
On executing, the above program generates the following output.
GET
HTTP/1.1 200 OK
-1
Exception in thread "main" org.apache.http.impl.execchain.RequestAbortedException:
Request aborted
	at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:180)
	at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:185)
	at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)
	at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)
	at
org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)
	at
org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)
	at
org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:108)
	at HttpGetExample.main(HttpGetExample.java:32)
```

8. HttpClient — Interceptors

Interceptors are those which helps to obstruct or change requests or responses. Protocol interceptors in general act upon a specific header or a group of related headers. HttpClient library provides support for interceptors.

Request interceptor

The **HttpRequestInterceptor** interface represents the request interceptors. This interface contains a method known as a process in which you need to write the chunk of code to intercept the requests.

On the client side, this method verifies/processes the requests before sending them to the server and, on the server side, this method is executed before evaluating the body of the request.

Creating request interceptor

You can create a request interceptor by following the steps given below.

Step 1: Create an object of HttpRequestInterceptor

Create an object of the HttpRequestInterceptor interface by implementing its abstract method process.

```
 HttpRequestInterceptor requestInterceptor = new HttpRequestInterceptor() {
```

```
@Override
   public void process(HttpRequest request, HttpContext context) throws
HttpException, IOException {
     //Method implementation . . . .  .
   }
};
```

Step 2: Instantiate CloseableHttpClient object

Build a custom **CloseableHttpClient** object by adding above created interceptor to it as shown below:

```
//Creating a CloseableHttpClient object
CloseableHttpClient httpclient =
HttpClients.custom().addInterceptorFirst(requestInterceptor).build();
```

Using this object, you can carry out the request executions as usual.

Example

Following example demonstrates the usage of request interceptors. In this example, we have created a HTTP GET request object and added three headers: sample-header, demo-header, and test-header to it.

In the **processor()** method of the interceptor, we are verifying the headers of the request sent; if any of those headers is **sample-header**, we are trying to remove it and display the list of headers of that particular request.

```
import java.io.IOException;


import org.apache.http.Header;
import org.apache.http.HttpException;
import org.apache.http.HttpRequest;
import org.apache.http.HttpRequestInterceptor;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.message.BasicHeader;
import org.apache.http.protocol.HttpContext;


public class InterceptorsExample {
   public static void main(String args[]) throws Exception{
      //Creating an HttpRequestInterceptor
      HttpRequestInterceptor requestInterceptor = new HttpRequestInterceptor() {
      @Override
      public void process(HttpRequest request, HttpContext context) throws
HttpException, IOException {
         if(request.containsHeader("sample-header")){
            System.out.println("Contains header sample-header, removing it..");
            request.removeHeaders("sample-header");
         }
         //Printing remaining list of headers
         Header[] headers= request.getAllHeaders();
         for (int i = 0; i<headers.length;i++){
            System.out.println(headers[i].getName());
         }
      }
   };

      //Creating a CloseableHttpClient object
      CloseableHttpClient httpclient =
HttpClients.custom().addInterceptorFirst(requestInterceptor).build();

      //Creating a request object
      HttpGet httpget1 = new HttpGet("https://www.tutorialspoint.com/");
      //Setting the header to it
      httpget1.setHeader(new BasicHeader("sample-header","My first header"));
      httpget1.setHeader(new BasicHeader("demo-header","My second header"));
      httpget1.setHeader(new BasicHeader("test-header","My third header"));

      //Executing the request
      HttpResponse httpresponse = httpclient.execute(httpget1);

      //Printing the status line
      System.out.println(httpresponse.getStatusLine());

   }

}
```

Output

On executing the above program, the following output is generated:

```
Contains header sample-header, removing it..
demo-header
test-header
HTTP/1.1 200 OK
```

Response interceptor

The **HttpResponseInterceptor** interface represents the response interceptors. This interface contains a method known as **process().** In this method, you need to write the chunk of code to intercept the responses.

On the server side, this method verifies/processes the response before sending them to the client, and on the client side, this method is executed before evaluating the body of the response.

Creating response interceptor

You can create a response interceptor by following the steps given below:

Step 1: Create an object of HttpResponseInterceptor

Create an object of the **HttpResponseInterceptor** interface by implementing its abstract method **process**.

```
 HttpResponseInterceptor responseInterceptor = new HttpResponseInterceptor() {
   @Override
   public void process(HttpResponse response, HttpContext context) throws
HttpException, IOException {
   //Method implementation . . . . . . . .
   }
};
```

Step 2: Instantiate CloseableHttpClient object

Build a custom **CloseableHttpClient** object by adding above created interceptor to it, as shown below:

```
//Creating a CloseableHttpClient object
CloseableHttpClient httpclient =
HttpClients.custom().addInterceptorFirst(responseInterceptor).build();
```

Using this object, you can carry out the request executions as usual.

Example

The following example demonstrates the usage of response interceptors. In this example, we have added three headers: sample-header, demo-header, and test-header to the response in the processor.

After executing the request and obtaining the response, we printed names of all the headers of the response using the **getAllHeaders**() method.

And in the output, you can observe the names of three headers in the list.

```
import java.io.IOException;

import org.apache.http.Header;
import org.apache.http.HttpException;
import org.apache.http.HttpResponse;
import org.apache.http.HttpResponseInterceptor;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.protocol.HttpContext;

public class ResponseInterceptorsExample {
   public static void main(String args[]) throws Exception{
      //Creating an HttpRequestInterceptor
      HttpResponseInterceptor responseInterceptor = new
HttpResponseInterceptor() {
         @Override
         public void process(HttpResponse response, HttpContext context) throws
HttpException, IOException {
            System.out.println("Adding header sample_header, demo-header,
test_header to the response");
            response.setHeader("sample-header", "My first header");
            response.setHeader("demo-header", "My second header");
            response.setHeader("test-header", "My third header");

         }
      };

      //Creating a CloseableHttpClient object
      CloseableHttpClient httpclient =
HttpClients.custom().addInterceptorFirst(responseInterceptor).build();

      //Creating a request object
      HttpGet httpget1 = new HttpGet("https://www.tutorialspoint.com/");

      //Executing the request
      HttpResponse httpresponse = httpclient.execute(httpget1);
     //Printing remaining list of headers
     Header[] headers= httpresponse.getAllHeaders();
     for (int i = 0; i<headers.length;i++){
        System.out.println(headers[i].getName());
     }

   }

}
```

Output

On executing, the above program generates the following result:

```
On executing the above program generates the following output.
Adding header sample_header, demo-header, test_header to the response
Accept-Ranges
Access-Control-Allow-Headers
Access-Control-Allow-Origin
Cache-Control
Content-Type
Date
Expires
Last-Modified
Server
Vary
X-Cache
sample-header
demo-header
test-header
```

9. HttpClient — User Authentication

Using HttpClient, you can connect to a website which needed username and password. This chapter explains, how to execute a client request against a site that asks for username and password.

Step 1: Create a CredentialsProvider object

The **CredentialsProvider** Interface maintains a collection to hold the user login credentials. You can create its object by instantiating the **BasicCredentialsProvider** class, the default implementation of this interface.

```
CredentialsProvider credentialsPovider = new BasicCredentialsProvider();
```

Step 2: Set the Credentials

You can set the required credentials to the CredentialsProvider object using the **setCredentials()** method.

This method accepts two objects as given below:

- **AuthScope object:** Authentication scope specifying the details like hostname, port number, and authentication scheme name.
- **Credentials object:** Specifying the credentials (username, password).

Set the credentials using the **setCredentials()** method for both host and proxy as shown below:

```
credsProvider.setCredentials(new AuthScope("example.com", 80), new
UsernamePasswordCredentials("user", "mypass"));

credsProvider.setCredentials(new AuthScope("localhost", 8000), new
UsernamePasswordCredentials("abc", "passwd"));
```

Step 3: Create a HttpClientBuilder Object

Create a **HttpClientBuilder** using the **custom()** method of the **HttpClients** class.

```
//Creating the HttpClientBuilder

HttpClientBuilder clientbuilder = HttpClients.custom();
```

Step 4: Set the credentialsPovider

You can set the above created credentialsPovider object to a HttpClientBuilder using the **setDefaultCredentialsProvider()** method.

Set the CredentialProvider object created in the previous step to the client builder by passing it to the **CredentialsProvider object()** method as shown below.

```
clientbuilder = clientbuilder.setDefaultCredentialsProvider(credsProvider);
```

Step 5: Build the CloseableHttpClient

Build the **CloseableHttpClient** object using the **build()** method of the **HttpClientBuilder** class.

```
CloseableHttpClient httpclient = clientbuilder.build();
```

Step 6: Create a HttpGet object and execute it

Create a HttpRequest object by instantiating the HttpGet class. Execute this request using the **execute()** method.

```
//Creating a HttpGet object
HttpGet httpget = new HttpGet("https://www.tutorialspoint.com/ ");


//Executing the Get request
HttpResponse httpresponse = httpclient.execute(httpget);
```

Example

Following is an example program which demonstrates the execution of a HTTP request against a target site that requires user authentication.

```
import org.apache.http.Header;
import org.apache.http.HttpResponse;
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.Credentials;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.BasicCredentialsProvider;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.impl.client.HttpClients;
```

```

public class UserAuthenticationExample {


   public static void main(String args[]) throws Exception{


        //Create an object of credentialsProvider
        CredentialsProvider credentialsPovider = new
BasicCredentialsProvider();



        //Set the credentials
        AuthScope scope = new
AuthScope("https://www.tutorialspoint.com/questions/", 80);
        Credentials credentials = new UsernamePasswordCredentials("USERNAME",
"PASSWORD");
        credentialsPovider.setCredentials(scope,credentials);


       //Creating the HttpClientBuilder
       HttpClientBuilder clientbuilder = HttpClients.custom();


       //Setting the credentials
       clientbuilder =
clientbuilder.setDefaultCredentialsProvider(credentialsPovider);


       //Building the CloseableHttpClient object
       CloseableHttpClient httpclient = clientbuilder.build();


       //Creating a HttpGet object
        HttpGet httpget = new
HttpGet("https://www.tutorialspoint.com/questions/index.php");


       //Printing the method used
        System.out.println(httpget.getMethod());

```

```
         //Executing the Get request
         HttpResponse httpresponse = httpclient.execute(httpget);


         //Printing the status line
         System.out.println(httpresponse.getStatusLine());
         int statusCode = httpresponse.getStatusLine().getStatusCode();
        System.out.println(statusCode);

        Header[] headers= httpresponse.getAllHeaders();
        for (int i = 0; i<headers.length;i++){
           System.out.println(headers[i].getName());
        }
    }
}
```

Output

On executing, the above program generates the following output.

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

10. HttpClient — Using Proxy

A Proxy server is an intermediary server between the client and the internet. Proxy servers offer the following basic functionalities:

- Firewall and network data filtering
- Network connection sharing
- Data caching

Using HttpClient library, you can send a HTTP request using a proxy. Follow the steps given below:

Step 1: Create a HttpHost object

Instantiate the **HttpHost** class of the **org.apache.http** package by passing a string parameter representing the name of the proxy host, (from which you need the requests to be sent) to its constructor.

```
//Creating an HttpHost object for proxy
HttpHost proxyHost = new HttpHost("localhost");
```

In the same way, create another HttpHost object to represent the target host to which requests need to be sent.

```
//Creating an HttpHost object for target
HttpHost targetHost = new HttpHost("google.com");
```

Step 2: Create an HttpRoutePlanner object

The **HttpRoutePlanner** interface computes a route to a specified host. Create an object of this interface by instantiating the **DefaultProxyRoutePlanner** class, an implementation of this interface. As a parameter to its constructor, pass the above created proxy host:

```
//creating a RoutePlanner object
HttpRoutePlanner routePlanner = new DefaultProxyRoutePlanner(proxyhost);
```

Step 3: Set the route planner to a client builder

Using the **custom()** method of the **HttpClients** class, create a **HttpClientBuilder** object and, to this object set the route planner created above, using the **setRoutePlanner()** method.

```
//Setting the route planner to the HttpClientBuilder object
HttpClientBuilder clientBuilder = HttpClients.custom();
```

```
clientBuilder = clientBuilder.setRoutePlanner(routePlanner);
```

Step 4: Build the CloseableHttpClient object

Build the **CloseableHttpClient** object by calling the **build()** method.

```
//Building a CloseableHttpClient
CloseableHttpClient  httpClient = clientBuilder.build();
```

Step 5: Create a HttpGet object

Create a HTTP GET request by instantiating the **HttpGet** class.

```
//Creating an HttpGet object
HttpGet httpGet = new HttpGet("/");
```

Step 6: Execute the request

One of the variants of the **execute()** method accepts an **HttpHost** and **HttpRequest** objects and executes the request. Execute the request using this method:

```
//Executing the Get request
HttpResponse httpResponse = httpclient.execute(targetHost, httpGet);
```

Example

Following example demonstrates how to send a HTTP request to a server via proxy. In this example, we are sending a HTTP GET request to google.com via localhost. We have printed the headers of the response and the body of the response.

```
import org.apache.http.Header;
import org.apache.http.HttpEntity;
import org.apache.http.HttpHost;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.conn.routing.HttpRoutePlanner;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.impl.conn.DefaultProxyRoutePlanner;
import org.apache.http.util.EntityUtils;

public class RequestViaProxyExample {
   public static void main(String args[]) throws Exception{
      //Creating an HttpHost object for proxy
      HttpHost proxyhost = new HttpHost("localhost");

      //Creating an HttpHost object for target
       HttpHost targethost = new HttpHost("google.com");

      //creating a RoutePlanner object
       HttpRoutePlanner routePlanner = new DefaultProxyRoutePlanner(proxyhost);

      //Setting the route planner to the HttpClientBuilder object
       HttpClientBuilder clientBuilder = HttpClients.custom();
       clientBuilder = clientBuilder.setRoutePlanner(routePlanner);

       //Building a CloseableHttpClient
       CloseableHttpClient  httpclient = clientBuilder.build();

      //Creating an HttpGet object
       HttpGet httpget = new HttpGet("/");

       //Executing the Get request
       HttpResponse httpresponse = httpclient.execute(targethost, httpget);

       //Printing the status line
       System.out.println(httpresponse.getStatusLine());

       //Printing all the headers of the response
      Header[] headers = httpresponse.getAllHeaders();
      for (int i = 0; i < headers.length; i++) {
         System.out.println(headers[i]);
      }
      //Printing the body of the response
      HttpEntity entity = httpresponse.getEntity();

      if (entity != null) {
         System.out.println(EntityUtils.toString(entity));
      }

   }

}
```

Output

On executing, the above program generates the following output:

```
HTTP/1.1 200 OK
Date: Sun, 23 Dec 2018 10:21:47 GMT
Server: Apache/2.4.9 (Win64) PHP/5.5.13
Last-Modified: Tue, 24 Jun 2014 10:46:24 GMT
ETag: "2e-4fc92abc3c000"
Accept-Ranges: bytes
Content-Length: 46
Content-Type: text/html
<html><body><h1>It works!</h1></body></html>
```

11. HttpClient — Proxy Authentication

In this chapter, we will learn how to create a HttpRequest authenticated using username and password and tunnel it through a proxy to a target host, using an example.

Step 1: Create a CredentialsProvider object

The CredentialsProvider Interface maintains a collection to hold the user login credentials. You can create its object by instantiating the BasicCredentialsProvider class, the default implementation of this interface.

```
CredentialsProvider credentialsPovider = new BasicCredentialsProvider();
```

Step 2: Set the credentials

You can set the required credentials to the CredentialsProvider object using the **setCredentials()** method. This method accepts two objects:

- **AuthScope object:** Authentication scope specifying the details like hostname, port number, and authentication scheme name.
- **Credentials object:** Specifying the credentials (username, password).

Set the credentials using the **setCredentials()** method for both host and proxy as shown below.

```
credsProvider.setCredentials(new AuthScope("example.com", 80), new
UsernamePasswordCredentials("user", "mypass"));

credsProvider.setCredentials(new AuthScope("localhost", 8000), new
UsernamePasswordCredentials("abc", "passwd"));
```

Step 3: Create an HttpClientBuilder object

Create a **HttpClientBuilder** using the **custom()** method of the **HttpClients** class as shown below:

```
//Creating the HttpClientBuilder

HttpClientBuilder clientbuilder = HttpClients.custom();
```

Step 4: Set the CredentialsProvider

You can set the CredentialsProvider object to a HttpClientBuilder object using the **setDefaultCredentialsProvider()** method. Pass the previously created **CredentialsProvider** object to this method.

```
clientbuilder = clientbuilder.setDefaultCredentialsProvider(credsProvider);
```

Step 5: Build the CloseableHttpClient

Build the **CloseableHttpClient** object using the **build()** method.

```
CloseableHttpClient httpclient = clientbuilder.build();
```

Step 6: Create the proxy and target hosts

Create the target and proxy hosts by instantiating the **HttpHost** class.

```
//Creating the target and proxy hosts
HttpHost target = new HttpHost("example.com", 80, "http");
HttpHost proxy = new HttpHost("localhost", 8000, "http");
```

Step 7: Set the proxy and build a RequestConfig object

Create a **RequestConfig.Builder** object using the **custom()** method. Set the previously created proxyHost object to the **RequestConfig.Builder** using the setProxy() method. Finally, build the **RequestConfig** object using the **build()** method.

```
RequestConfig.Builder reqconfigconbuilder= RequestConfig.custom();
reqconfigconbuilder = reqconfigconbuilder.setProxy(proxyHost);
RequestConfig config = reqconfigconbuilder.build();
```

Step 8: Create a HttpGet request object and set config object to it.

Create a **HttpGet** object by instantiating the HttpGet class. Set the config object created in the previous step to this object using the **setConfig()** method.

```
//Create the HttpGet request object
HttpGet httpGet = new HttpGet("/");


//Setting the config to the request
httpget.setConfig(config);
```

Step 9: Execute the request

Execute the request by passing the HttpHost object (target) and request (HttpGet) as parameters to the **execute()** method.

```
HttpResponse httpResponse = httpclient.execute(targetHost, httpget);
```

Example

Following example demonstrates how to execute a HTTP request through a proxy using username and password.

```
import org.apache.http.HttpHost;
import org.apache.http.HttpResponse;
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.client.config.RequestConfig;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.BasicCredentialsProvider;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.impl.client.HttpClients;

public class ProxyAuthenticationExample {
   public static void main(String[] args) throws Exception {


        //Creating the CredentialsProvider object
      CredentialsProvider credsProvider = new BasicCredentialsProvider();

      //Setting the credentials
      credsProvider.setCredentials(new AuthScope("example.com", 80), new
UsernamePasswordCredentials("user", "mypass"));
      credsProvider.setCredentials(new AuthScope("localhost", 8000), new
UsernamePasswordCredentials("abc", "passwd"));

      //Creating the HttpClientBuilder
      HttpClientBuilder clientbuilder = HttpClients.custom();

      //Setting the credentials
      clientbuilder =
clientbuilder.setDefaultCredentialsProvider(credsProvider);
```

```
      //Building the CloseableHttpClient object
      CloseableHttpClient httpclient = clientbuilder.build();



      //Create the target and proxy hosts
      HttpHost targetHost = new HttpHost("example.com", 80, "http");
      HttpHost proxyHost = new HttpHost("localhost", 8000, "http");

      //Setting the proxy
      RequestConfig.Builder reqconfigconbuilder= RequestConfig.custom();
      reqconfigconbuilder = reqconfigconbuilder.setProxy(proxyHost);
      RequestConfig config = reqconfigconbuilder.build();

      //Create the HttpGet request object
      HttpGet httpget = new HttpGet("/");

      //Setting the config to the request
      httpget.setConfig(config);

      //Printing the status line
      HttpResponse response = httpclient.execute(targetHost, httpget);
      System.out.println(response.getStatusLine());

   }
}
```

Output

On executing, the above program generates the following output:

```
HTTP/1.1 200 OK
```

12. HttpClient — Form-Based Login

Using the HttpClient library you can send a request or, login to a form by passing parameters.

Follow the steps given below to login to a form.

Step 1: Create an HttpClient object

The **createDefault()** method of the **HttpClients** class returns an object of the class **CloseableHttpClient,** which is the base implementation of the HttpClient interface. Using this method, create an HttpClient object:

```
CloseableHttpClient httpClient = HttpClients.createDefault();
```

Step 2: Create a RequestBuilder object

The class **RequestBuilder** is used to build request by adding parameters to it. If the request type is PUT or POST, it adds the parameters to the request as URL encoded entity.

Create a RequestBuilder object (of type POST) using the post() method.

```
//Building the post request object

RequestBuilder reqbuilder = RequestBuilder.post();
```

Step 3: Set Uri and parameters to the RequestBuilder.

Set the URI and parameters to the RequestBuilder object using the **setUri()** and **addParameter()** methods of the RequestBuilder class.

```
//Set URI and parameters

RequestBuilder reqbuilder = reqbuilder.setUri("http://httpbin.org/post");

reqbuilder = reqbuilder1.addParameter("Name",
"username").addParameter("password", "password");
```

Step 4: Build the HttpUriRequest object

After setting the required parameters, build the **HttpUriRequest** object using the **build()** method.

```
//Building the HttpUriRequest object

HttpUriRequest httppost = reqbuilder2.build();
```

Step 5: Execute the request

The execute method of the CloseableHttpClient object accepts a HttpUriRequest (interface) object (i.e. HttpGet, HttpPost, HttpPut, HttpHead etc.) and returns a response object.

Execute the HttpUriRequest created in the previous steps by passing it to the **execute()** method.

```
//Execute the request
HttpResponse httpresponse = httpclient.execute(httppost);
```

Example

Following example demonstrates how to logon to a form by sending login credentials. Here, we have sent two parameters: **username and password** to a form and tried to print the message entity and status of the request.

```
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpUriRequest;
import org.apache.http.client.methods.RequestBuilder;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;
import java.io.IOException;
import java.net.URISyntaxException;

public class FormLoginExample {
   public static void main(String args[]) throws Exception {

      //Creating CloseableHttpClient object
      CloseableHttpClient httpclient = HttpClients.createDefault();

      //Creating the RequestBuilder object
      RequestBuilder reqbuilder = RequestBuilder.post();

      //Setting URI and parameters
      RequestBuilder reqbuilder1 = reqbuilder.setUri("http://httpbin.org/post");
      RequestBuilder reqbuilder2 = reqbuilder1.addParameter("Name",
"username").addParameter("password", "password");

      //Building the HttpUriRequest object
      HttpUriRequest httppost = reqbuilder2.build();

      //Executing the request
      HttpResponse httpresponse = httpclient.execute(httppost);

      //Printing the status and the contents of the response
      System.out.println(EntityUtils.toString(httpresponse.getEntity()));
      System.out.println(httpresponse.getStatusLine());
    }
}
```

Output

On executing, the above program generates the following output:

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "Name": "username",
    "password": "password"
  },
  "headers": {
    "Accept-Encoding": "gzip,deflate",
    "Connection": "close",
    "Content-Length": "31",
    "Content-Type": "application/x-www-form-urlencoded; charset=UTF-8",
    "Host": "httpbin.org",
    "User-Agent": "Apache-HttpClient/4.5.6 (Java/1.8.0_91)"
  },
  "json": null,
  "origin": "117.216.245.180",
  "url": "http://httpbin.org/post"
}

HTTP/1.1 200 OK
```

Form Login with Cookies

If your form stores cookies, instead of creating default **CloseableHttpClient** object.

Create a CookieStore object by instantiating the BasicCookieStore class.

```
//Creating a BasicCookieStore object
BasicCookieStore cookieStore = new BasicCookieStore();
```

Create a HttpClientBuilder using the **custom()** method of the **HttpClients** class.

```
//Creating an HttpClientBuilder object
HttpClientBuilder clientbuilder = HttpClients.custom();
```

Set the cookie store to the client builder using the setDefaultCookieStore() method.

```
//Setting default cookie store to the client builder object
Clientbuilder = clientbuilder.setDefaultCookieStore(cookieStore);
```

Build the **CloseableHttpClient** object using the **build()** method.

```
//Building the CloseableHttpClient object
CloseableHttpClient httpclient = clientbuilder1.build();
```

Build the **HttpUriRequest** object as specified above by passing execute the request.

If the page stores cookies, the parameters you have passed will be added to the cookie store.

You can print the contents of the **CookieStore** object where you can see your parameters (along with the previous ones the page stored in case).

To print the cookies, get all the cookies from the **CookieStore** object using the **getCookies()** method. This method returns a **List** object. Using Iterator, print the list objects contents as shown below:

```
//Printing the cookies
List list = cookieStore.getCookies();


System.out.println("list of cookies");
Iterator it = list.iterator();
if(it.hasNext()){
   System.out.println(it.next());
}
```

13. HttpClient — Cookies Management

Cookies are text files stored on the client computer and they are kept for various information tracking purpose.

HttpClient provides support for cookies you can create and manage cookies.

Creating a cookie

Follow the steps given below to create a cookie using HttpClient library.

Step 1: Create Cookiestore object

The **CookieStore** interface represents the abstract store for Cookie objects. You can create a cookie store by instantiating the **BasicCookieStore** class, a default implementation of this interface.

```
//Creating the CookieStore object

CookieStore cookieStore = new BasicCookieStore();
```

Step 2: Create ClientCookie object

In addition to the functionalities of a cookie, ClientCookie can get the original cookies in the server. You can create a client cookie by instantiating the **BasicClientCookie** class. To the constructor of this class, you need to pass the key-value pair that you desired to store in that particular cookie.

```
//Creating client cookie

BasicClientCookie clientCookie = new BasicClientCookie("name","Raju");
```

Step 3: Set values to the cookie

To a client cookie, you can set/remove path, value, version, expiry date, domain, comment, and attribute using the respective methods.

```
Calendar myCal = new GregorianCalendar(2018, 9, 26);

Date expiryDate = myCal.getTime();

clientcookie.setExpiryDate(expiryDate);

clientcookie.setPath("/");

clientcookie.setSecure(true);

clientcookie.setValue("25");

clientcookie.setVersion(5);
```

Step 4: Add cookie to the cookie store

You can add cookies to the cookie store using the **addCookie()** method of the **BasicCookieStore** class.

Add the required cookies to the **Cookiestore.**

```
//Adding the created cookies to cookie store
 cookiestore.addCookie(clientcookie);
```

Example

Following example demonstrates how to create cookies and add them to a cookie store. Here, we created a cookie store, a bunch of cookies by setting the domain and path values, and added these to the cookie store.

```
import org.apache.http.client.CookieStore;
import org.apache.http.impl.client.BasicCookieStore;
import org.apache.http.impl.cookie.BasicClientCookie;

public class CookieHandlingExample {
   public static void main(String args[]) throws Exception{

      //Creating the CookieStore object
      CookieStore cookiestore = new BasicCookieStore();

      //Creating client cookies
      BasicClientCookie clientcookie1 = new BasicClientCookie("name","Raju");
      BasicClientCookie clientcookie2 = new BasicClientCookie("age","28");
      BasicClientCookie clientcookie3 = new BasicClientCookie("place","Hyderabad");

      //Setting domains and paths to the created cookies
      clientcookie1.setDomain(".sample.com");
      clientcookie2.setDomain(".sample.com");
      clientcookie3.setDomain(".sample.com");

      clientcookie1.setPath("/");
      clientcookie2.setPath("/");
      clientcookie3.setPath("/");
      //Adding the created cookies to cookie store
      cookiestore.addCookie(clientcookie1);
      cookiestore.addCookie(clientcookie2);
      cookiestore.addCookie(clientcookie3);
     }

}
```

Retrieving a cookie

You can get the cookies added to a cookie store using **getCookies()** method of the **BasicCookieStore** class. This method returns a list which holds all the cookies in the cookie store.

You can print the contents of a cookie store using the Iterator as shown below:

```
//Retrieving the cookies
List list = cookieStore.getCookies();

//Creating an iterator to the obtained list
Iterator it = list.iterator();
while(it.hasNext()){
   System.out.println(it.next());
}
```

Example

Following example demonstrates how to retrieve cookies from a cookie store. Here, we are adding a bunch of cookies to a cookie store and retrieving them back.

```
import org.apache.http.client.CookieStore;
import org.apache.http.impl.client.BasicCookieStore;
import org.apache.http.impl.cookie.BasicClientCookie;

public class CookieHandlingExample {
   public static void main(String args[]) throws Exception{

      //Creating the CookieStore object
      CookieStore cookiestore = new BasicCookieStore();

```

```
        //Creating client cookies
        BasicClientCookie clientcookie1 = new BasicClientCookie("name","Raju");
        BasicClientCookie clientcookie2 = new BasicClientCookie("age","28");
        BasicClientCookie clientcookie3 = new BasicClientCookie("place","Hyderabad");

        //Setting domains and paths to the created cookies
        clientcookie1.setDomain(".sample.com");
        clientcookie2.setDomain(".sample.com");
        clientcookie3.setDomain(".sample.com");

        clientcookie1.setPath("/");
        clientcookie2.setPath("/");
        clientcookie3.setPath("/");

        //Adding the created cookies to cookie store
        cookiestore.addCookie(clientcookie1);
        cookiestore.addCookie(clientcookie2);
        cookiestore.addCookie(clientcookie3);
       }
}
```

Output

On executing, this program generates the following output:

```
[version: 0][name: age][value: 28][domain: .sample.com][path: /][expiry: null]
[version: 0][name: name][value: Raju][domain: my.example.com][path: /][expiry:
null]
[version: 0][name: place][value: Hyderabad][domain: .sample.com][path:
/][expiry: null]
```

14. HttpClient — Multiple Threads

A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources.

You can execute requests from multiple threads by writing a multithreaded HttpClient program.

If you want to execute multiple client requests from threads consecutively, you need to create a **ClientConnectionPoolManager**. It maintains a pool of **HttpClientConnections** and serves multiple requests from threads.

The connections manager pools the connections based on the route. If the manager has connections for a particular route, then it serves new requests in those routes by leasing an existing connection from the pool, instead of creating a new one.

Follow the steps to execute requests from multiple threads:

Step 1: Creating the Client Connection Pool Manager

Create the Client Connection Pool Manager by instantiating the **PoolingHttpClientConnectionManager** class.

```
PoolingHttpClientConnectionManager connManager = new
PoolingHttpClientConnectionManager();
```

Step 2: Set the maximum number of connections

Set the maximum number of connections in the pool using the **setMaxTotal()** method.

```
 //Set the maximum number of connections in the pool

connManager.setMaxTotal(100);
```

Step 3: Create a ClientBuilder Object

Create a **ClientBuilder** Object by setting the connection manager using the **setConnectionManager()** method as shown below:

```
HttpClientBuilder clientbuilder =
HttpClients.custom().setConnectionManager(connManager);
```

Step 4: Create the HttpGet request objects

Instantiate the HttpGet class by passing the desired URI to its constructor as a parameter.

```
HttpGet httpget1 = new HttpGet("URI1");
HttpGet httpget2 = new HttpGet("URI2");
 . . . . . . . . . . . .
```

Step 5: Implementing the run method

Make sure that you have created a class, made it a thread (either by extending the thread class or, by implementing the Runnable interface) and implemented the run method.

```
public class ClientMultiThreaded extends Thread {
   public void run() {
      //Run method implementation . . . . . . . . . .
   }
}
```

Step 6: Create Thread objects

Create thread objects by instantiating the Thread class (ClientMultiThreaded) created above.

Pass a HttpClient object, respective HttpGet object and, an integer representing the ID to these threads.

```
ClientMultiThreaded thread1 = new ClientMultiThreaded(httpclient,httpget1, 1);
ClientMultiThreaded thread2 = new ClientMultiThreaded(httpclient,httpget2, 2);
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

Step 7: Start and join the threads

Start all the threads using **start()** method and join them using the join **method()**.

```
thread1.start();
thread2.start();
. . . . . . . .
thread1.join();
thread2.join();
. . . . . . . . . . . .
```

Step 8: Run method implementation

Within the run method, execute the request, retrieve the response and print the results.

Example

Following example demonstrates the execution of HTTP requests simultaneously from multiple threads. In this example, we are trying to execute various requests from various threads and trying to print the status, and the number of bytes read by each client.

```
import org.apache.http.HttpEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.

HttpClientBuilder;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.impl.conn.PoolingHttpClientConnectionManager;
import org.apache.http.util.EntityUtils;

public class ClientMultiThreaded extends Thread {
    CloseableHttpClient httpClient;
    HttpGet httpget;
    int id;

    public ClientMultiThreaded(CloseableHttpClient httpClient, HttpGet httpget,
int id) {
       this.httpClient = httpClient;
       this.httpget = httpget;
       this.id = id;
    }
    @Override
    public void run() {
       try{
       //Executing the request
          CloseableHttpResponse httpresponse = httpClient.execute(httpget);

          //Displaying the status of the request.
          System.out.println("status of thread "+id+":
"+httpresponse.getStatusLine());


          //Retrieving the HttpEntity and displaying the no.of bytes read
          HttpEntity entity = httpresponse.getEntity();
          if (entity != null) {
             System.out.println("Bytes read by thread thread "+id+":
"+EntityUtils.toByteArray(entity).length);
          }
       }catch(Exception e){
        System.out.println(e.getMessage());
      }


   }
   public static void main(String[] args) throws Exception {



      //Creating the Client Connection Pool Manager by instantiating the
PoolingHttpClientConnectionManager class.
      PoolingHttpClientConnectionManager connManager = new
PoolingHttpClientConnectionManager();


      //Set the maximum number of connections in the pool
      connManager.setMaxTotal(100);


      //Create a ClientBuilder Object by setting the connection manager
      HttpClientBuilder clientbuilder =
HttpClients.custom().setConnectionManager(connManager);


      //Build the CloseableHttpClient object using the build() method.
      CloseableHttpClient httpclient = clientbuilder.build();


      //Creating the HttpGet requests
      HttpGet httpget1 = new HttpGet("http://www.tutorialspoint.com/");
      HttpGet httpget2 = new HttpGet("http://www.google.com/");
      HttpGet httpget3 = new HttpGet("https://www.qries.com/");
      HttpGet httpget4 = new HttpGet("https://in.yahoo.com/");
      //Creating the Thread objects
      ClientMultiThreaded thread1 = new ClientMultiThreaded(httpclient,httpget1, 1);
     ClientMultiThreaded thread2 = new ClientMultiThreaded(httpclient,httpget2, 2);
     ClientMultiThreaded thread3 = new ClientMultiThreaded(httpclient,httpget3, 3);
     ClientMultiThreaded thread4 = new ClientMultiThreaded(httpclient,httpget4, 4);

      //Starting all the threads
      thread1.start();
      thread2.start();
      thread3.start();
      thread4.start();

      //Joining all the threads
      thread1.join();
      thread2.join();
      thread3.join();
      thread4.join();
   }

}
```

Output

On executing, the above program generates the following output:

```
status of thread 1: HTTP/1.1 200 OK
Bytes read by thread thread 1: 36907
status of thread 2: HTTP/1.1 200 OK
Bytes read by thread thread 2: 13725
status of thread 3: HTTP/1.1 200 OK
Bytes read by thread thread 3: 17319
status of thread 4: HTTP/1.1 200 OK
Bytes read by thread thread 4: 127018
```

15. HttpClient — Custom SSL Context

Using Secure Socket Layer, you can establish a secured connection between the client and server. It helps to safeguard sensitive information such as credit card numbers, usernames, passwords, pins, etc.

You can make connections more secure by creating your own SSL context using the **HttpClient** library.

Follow the steps given below to customize SSLContext using HttpClient library:

Step 1: Create SSLContextBuilder object

SSLContextBuilder is the builder for the SSLContext objects. Create its object using the **custom()** method of the **SSLContexts** class.

```
//Creating SSLContextBuilder object
SSLContextBuilder SSLBuilder = SSLContexts.custom();
```

Step 2: Load the Keystore

In the path ***Java_home_directory*/jre/lib/security/**, you can find a file named **cacerts.** Save this as your key store file (with extension *.jks*). Load the keystore file and, its password (which is ***changeit*** by default) using the **loadTrustMaterial()** method of the **SSLContextBuilder** class.

```
//Loading the Keystore file
File file  = new File("mykeystore.jks");
SSLBuilder = SSLBuilder.loadTrustMaterial(file, "changeit".toCharArray());
```

Step 3: build an SSLContext object

An SSLContext object represents a secure socket protocol implementation. Build an SSLContext using the **build()** method.

```
//Building the SSLContext
SSLContext sslContext = SSLBuilder.build();
```

Step 4: Creating SSLConnectionSocketFactory object

SSLConnectionSocketFactory is a layered socket factory for TSL and SSL connections. Using this, you can verify the Https server using a list of trusted certificates and authenticate the given Https server.

You can create this in many ways. Depending on the way you create an **SSLConnectionSocketFactory** object, you can allow all hosts, allow only self-signed certificates, allow only particular protocols, etc.

To allow only particular protocols, create **SSLConnectionSocketFactory** object by passing an SSLContext object, string array representing the protocols need to be supported, string array representing the cipher suits need to be supported and a HostnameVerifier object to its constructor.

```
new SSLConnectionSocketFactory(sslcontext, new String[]{"TLSv1"}, null,
SSLConnectionSocketFactory.getDefaultHostnameVerifier());
```

To allow all hosts, create **SSLConnectionSocketFactory** object by passing a SSLContext object and a **NoopHostnameVerifier** object.

```
//Creating SSLConnectionSocketFactory SSLConnectionSocketFactory object

SSLConnectionSocketFactory sslConSocFactory = new
SSLConnectionSocketFactory(sslcontext, new NoopHostnameVerifier());
```

Step 5: Create an HttpClientBuilder object

Create an HttpClientBuilder object using the **custom()** method of the **HttpClients** class.

```
//Creating HttpClientBuilder

HttpClientBuilder clientbuilder = HttpClients.custom();
```

Step 6: Set the SSLConnectionSocketFactory object

Set the SSLConnectionSocketFactory object to the **HttpClientBuilder** using the **setSSLSocketFactory()** method.

```
//Setting the SSLConnectionSocketFactory

clientbuilder = clientbuilder.setSSLSocketFactory(sslConSocFactory);
```

Step 7: Build the CloseableHttpClient object

Build the **CloseableHttpClient** object by calling the **build()** method.

```
//Building the CloseableHttpClient

CloseableHttpClient httpclient = clientbuilder.build();
```

Step 8: Create an HttpGet object

The **HttpGet** class represents the HTTP GET request which retrieves the information of the given server using a URI.

Create a HTTP GET request by instantiating the HttpGet class by passing a string representing the URI.

```
//Creating the HttpGet request
HttpGet httpget = new HttpGet("https://example.com/");
```

Step 9: Execute the request

Execute the request using the **execute()** method.

```
//Executing the request
HttpResponse httpresponse = httpclient.execute(httpget);
```

Example

Following example demonstrates the customization of the SSLContrext:

```
import java.io.File;


import javax.net.ssl.SSLContext;


import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.conn.ssl.NoopHostnameVerifier;
import org.apache.http.conn.ssl.SSLConnectionSocketFactory;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.ssl.SSLContextBuilder;
import org.apache.http.ssl.SSLContexts;
import org.apache.http.util.EntityUtils;



public class ClientCustomSSL {
   public final static void main(String[] args) throws Exception {

        //Creating SSLContextBuilder object
        SSLContextBuilder SSLBuilder = SSLContexts.custom();
        //Loading the Keystore file
        File file  = new File("mykeystore.jks");
        SSLBuilder = SSLBuilder.loadTrustMaterial(file,
"changeit".toCharArray());

        //Building the SSLContext usiong the build() method
        SSLContext sslcontext = SSLBuilder.build();

        //Creating SSLConnectionSocketFactory object
        SSLConnectionSocketFactory sslConSocFactory = new
SSLConnectionSocketFactory(sslcontext, new NoopHostnameVerifier());

        //Creating HttpClientBuilder
        HttpClientBuilder clientbuilder = HttpClients.custom();

        //Setting the SSLConnectionSocketFactory
        clientbuilder = clientbuilder.setSSLSocketFactory(sslConSocFactory);

        //Building the CloseableHttpClient
        CloseableHttpClient httpclient = clientbuilder.build();

        //Creating the HttpGet request
        HttpGet httpget = new HttpGet("https://example.com/");

        //Executing the request
        HttpResponse httpresponse = httpclient.execute(httpget);

        //printing the status line
        System.out.println(httpresponse.getStatusLine());

        //Retrieving the HttpEntity and displaying the no.of bytes read
        HttpEntity entity = httpresponse.getEntity();
        if (entity != null) {
           System.out.println(EntityUtils.toByteArray(entity).length);
        }
```

```
    }
}
```

Output

On executing, the above program generates the following output.

```
HTTP/1.1 200 OK
1270
```

16. HttpClient — Multipart Upload

Using HttpClient, we can perform Multipart upload, i.e., we can upload larger objects in smaller parts. In this chapter, we demonstrate the multipart upload in HTTP client by uploading a simple text file.

In general, any multipart upload contains three parts.

- Initiation of the upload
- Uploading the object parts
- Completing the Multipart upload

For the multipart upload using HttpClient, we need to follow the below steps:

- Create a multipart builder.
- Add desired parts to it.
- Complete the build and obtain a multipart HttpEntity.
- Build request by setting the above muti-part entity.
- Execute the request.

Following are the steps to upload a multipart entity using the HttpClient library.

Step 1: Create an HttpClient object

The **createDefault()** method of the **HttpClients** class returns an object of the class **CloseableHttpClient,** which is the base implementation of the HttpClient interface. Using this method, create an HttpClient object:

```
//Creating CloseableHttpClient object
CloseableHttpClient httpclient = HttpClients.createDefault();
```

Step 2: Create a FileBody object

FileBody class represents the binary body part backed by a file. Instantiate this class by passing a **File** object and a **ContentType** object representing the type of the content.

```
//Creating a File object
File file = new File("sample.txt");


//Creating the FileBody object
FileBody filebody = new FileBody(file, ContentType.DEFAULT_BINARY);
```

Step 3: Create a MultipartEntityBuilder

The **MultipartEntityBuilder** class is used to build the multi-part **HttpEntity** object. Create its object using the **create()** method (of the same class).

```
//Creating the MultipartEntityBuilder
MultipartEntityBuilder entitybuilder = MultipartEntityBuilder.create();
```

Step 4: Set the mode

A **MultipartEntityBuilder** has three modes: STRICT, RFC6532, and BROWSER_COMPATIBLE. Set it to the desired mode using the **setMode()** method.

```
//Setting the mode
entitybuilder.setMode(HttpMultipartMode.BROWSER_COMPATIBLE);
```

Step 5: Add various the desired parts

Using the methods **addTextBody(), addPart()** and, **addBinaryBody(),** you can add simple text, files, streams, and other objects to a **MultipartBuilder**. Add the desired contents using these methods.

```
//Adding text
entitybuilder.addTextBody("sample_text", "This is the text part of our file");


//Adding a file
entitybuilder.addBinaryBody("image", new File("logo.png"));
```

Step 6: Building single entity

You can build all these parts to a single entity using the **build()** method of the **MultipartEntityBuilder** class. Using this method, build all the parts into a single **HttpEntity**.

```
//Building a single entity using the parts
HttpEntity mutiPartHttpEntity = entityBuilder.build();
```

Step 7: Create a RequestBuilder object

The class **RequestBuilder** is used to build request by adding parameters to it. If the request is of type PUT or POST, it adds the parameters to the request as URL encoded entity.

Create a RequestBuilder object (of type POST) using the **post()** method. And pass the Uri to which you wanted to send the request it as a parameter.

```
//Building the post request object
RequestBuilder reqbuilder = RequestBuilder.post("http://httpbin.org/post");
```

Step 8: Set the entity object to the RequestBuilder

Set the above created multipart entity to the RequestBuilder using the **setEntity()** method of the **RequestBuilder** class.

```
//Setting the entity object to the RequestBuilder
reqbuilder.setEntity(mutiPartHttpEntity);
```

Step 9: Build the HttpUriRequest

Build a **HttpUriRequest** request object using the **build()** method of the **RequestBuilder** class.

```
//Building the request
HttpUriRequest multipartRequest = reqbuilder.build();
```

Step 10: Execute the request

Using the **execute()** method, execute the request built in the previous step (bypassing the request as a parameter to this method).

```
//Executing the request
HttpResponse httpresponse = httpclient.execute(multipartRequest);
```

Example

Following example demonstrates how to send a multipart request using the HttpClient library. In this example, we are trying to send a multipart request backed by a file.

```
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpUriRequest;
import org.apache.http.client.methods.RequestBuilder;
import org.apache.http.entity.ContentType;
import org.apache.http.entity.mime.HttpMultipartMode;
import org.apache.http.entity.mime.MultipartEntityBuilder;
import org.apache.http.entity.mime.content.FileBody;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

import java.io.File;
import java.io.IOException;
import java.net.URISyntaxException;

public class MultipartUploadExample {
   public static void main(String args[]) throws Exception{

      //Creating CloseableHttpClient object
       CloseableHttpClient httpclient = HttpClients.createDefault();

      //Creating a file object
       File file = new File("sample.txt");

       //Creating the FileBody object
       FileBody filebody = new FileBody(file, ContentType.DEFAULT_BINARY);

       //Creating the MultipartEntityBuilder
       MultipartEntityBuilder entitybuilder = MultipartEntityBuilder.create();

      //Setting the mode
       entitybuilder.setMode(HttpMultipartMode.BROWSER_COMPATIBLE);

      //Adding text
       entitybuilder.addTextBody("sample_text", "This is the text part of our file");

       //Adding a file
       entitybuilder.addBinaryBody("image", new File("logo.png"));

      //Building a single entity using the parts
       HttpEntity mutiPartHttpEntity = entitybuilder.build();

      //Building the RequestBuilder request object
      RequestBuilder reqbuilder = RequestBuilder.post("http://httpbin.org/post");

      //Set the entity object to the RequestBuilder
      reqbuilder.setEntity(mutiPartHttpEntity);

      //Building the request
      HttpUriRequest multipartRequest = reqbuilder.build();

      //Executing the request
      HttpResponse httpresponse = httpclient.execute(multipartRequest);

      //Printing the status and the contents of the response
      System.out.println(EntityUtils.toString(httpresponse.getEntity()));
      System.out.println(httpresponse.getStatusLine());

    }
}
```

Output

On executing, the above program generates the following output:

```
{
  "args": {},
  "data": "",
  "files": {
    "image": "data:application/octet-
s66PohrH3IWNk1FzpohfdXPIfv9X3490FGcuXsHn9X0piCwomF/xdgADZ9GsfSyvLYAAAAAE
lFTkSuQmCC"
  },
  "form": {
    "sample_text": "This is the text part of our file"
  },
  "headers": {
    "Accept-Encoding": "gzip,deflate",
    "Connection": "close",
    "Content-Length": "11104",
    "Content-Type": "multipart/form-data;
boundary=UFJbPHT7mTwpVq70LpZgCi5I2nvxd1g-I8Rt",
    "Host": "httpbin.org",
    "User-Agent": "Apache-HttpClient/4.5.6 (Java/1.8.0_91)"
  },
  "json": null,
  "origin": "117.216.245.180",
  "url": "http://httpbin.org/post"
}

HTTP/1.1 200 OK
```