FrequentlyAskedApplicationDesignQuestions

Application Design FAQ

This document addresses questions about application design which have been raised repeatedly on the HttpClient and HttpComponents mailing lists. As it addresses design issues rather than API or other HttpClient or HttpComponents specific problems, much of the information presented is equally applicable for HttpURLConnection or non-Java APIs.

If you are just getting your feet wet and want to understand the basics of client HTTP programming rather than read about application design alternatives, check out our primer.

Sending Parameters and Uploading Files

A question that is asked on a regular basis is:

How do I upload a file along with some parameters?

This section presents different ways to upload parameters, files, and both. It assumes that you are implementing both a client application and a server application to which the client application connects. The client application might be a Java program using HttpClient, while the server application is assumed to be a Servlet.

Parameters are name/value pairs, where both name and value are strings. Names should always be in US-ASCII, values may use other character encodings, depending on the technique used for sending the parameters. Files or rather file parameters are name/value pairs, where the name is a string and the value is binary data read from a file. Binary values from other sources can be handled similar to files from a design perspective, though the details of the API will vary.

GET with Query

The simplest way to transfer parameters to the server is the query string of the URL. That's the part after the question mark, for example in:

http:'***//my.server.name/my/servlet****?param1=value1&param2=value2*'

Query strings can be used with any HTTP method, but they are most frequently used with the GET method. A query string is also the only way to send parameters with a GET method. (Unless your application encodes parameters into the URL path.)

The names and values of a query string must be URL-escaped. Each space character needs to be replaced by a + character. Reserved characters like = & % + : / need to be URL-escaped (%xx sequences) with their byte representation (see below). URL-escaping is automatically handled for example by the java.net.URI and org.apache.commons.httpclient.URI classes.

HTTP Request lines and thus query strings are confined to the ASCII character encoding. Only ASCII names/values can reliably be transferred in a query string. However it is possible to use a non-ASCII character encoding by URL-escaping the characters. Character encodings (like UTF-8, ISO-8859-1; and unlike EBCDIC) whose lower 7-bit are compatible with ASCII only need to escape the non-ASCII characters. The character encoding used to create and interprete the % escape sequences must be the same on the server and on the client. It is common practice to agree on UTF-8. But it is more a recommendation than a standard and must be verified in individual cases. To avoid problems it is strongly discouraged to send non-ASCII values in the query string.

Depending on the encoding the following characters are character encoded and URL-escaped as follows:

char	ASCII	ISO-8859-1	UTF-8	EBCDIC
A	A	A	A	%E1
ä	N/A	%E4	%C3%A4	N/A
&	%26	%26	%26	%70

On the server, name/value pairs sent in a query string are available as parameters of the ServletRequest.
%xx escape sequences and + characters are automatically decoded by the Servlet API. If non-ASCII values are sent in the query string, the outcome depends on the implementation of the Servlet API, the Content-Type header and possibly also on configuration parameters, such as the JVM default character set. That's why it is strongly discouraged to send non-ASCII values in the query string.

POST with URL-encoded Form Data

Unlike the GET method, a POST method has a message body or entity which can hold any kind of binary or non-binary data. A simple way to send parameters with string values to the server is to put the query string into the message body instead of the URL. This avoids URL length restrictions, problems with parameters being logged where they shouldn't, and it also allows for non-ASCII characters in the values. While a URL is confined to ASCII characters, the message body is not. The character set can be specified in a header field. The encoding of special characters is automatically handled by HttpClient's PostMethod.

On the server, name/value pairs sent in a message body with content type "application/x-www-form-urlencoded" are available as parameters of the ServletRequest. If there are parameters in both the message body and the query string, all of them are available in the ServletRequest.

POST with Multipart Form Data

In order to upload binary data such as files, the data can be encoded as multipart MIME. That's the same format which is used for sending email attachments. HTML forms for uploading files have to specify the content type "multipart/form-data" so the browser knows that the multipart MIME encoding must be applied. That content type is also sent to the server. HttpClient provides the MultipartRequestEntity class to perform multipart MIME encoding.

On the server, parameters sent as multipart MIME are *not* available as parameters of the ServletRequest. The servlet has to read and interpret the message body explicitly. There are libraries for parsing multipart MIME data, for example Commons FileUpload. It should also be possible to parse multipart/form-data using the JavaMail API. If there are parameters in both the message body and the query string, only those from the query string are available in the ServletRequest.

POST with Query and Data

In the special case that you need to upload only parameters with ASCII string values and a single file, there is another option. You can send the parameters in a query string and the file contents as the message body. This does not require special encoding or decoding of the binary data. This kind of request can not be generated by an HTML form.

With this approach, the file is not sent as a name/value pair. Only the value, that is the file contents, will be transferred. The servlet has to know what to do with the file based only on the information in the URL (and HTTP headers). The information in the URL can come from the query string (?name=MyImage.png), from the URL path (/my/servlet/save), or both.

You should specify a content type that indicates the type of data you are sending in the message body, such as "application/octet-stream" or "image/png" or whatever else is appropriate for the file you are uploading. If you are uploading a text file, you should also specify the character set as part of the content type.

On the server, the parameters sent in the query string are available as parameters of the ServletRequest. The file contents is available as from the ServletRequest as either binary or
character data.

Further Reading

Java Standard Edition 5.0, HttpURLConnection

IANA: Registered MIME Media Types

HTML 4.01: Form Content Types

Sun: Java Servlet Technology Documentation

Servlet API 2.3

Java Enterprise Edition 5.0, Servlet API 2.4

JavaMail

Java Enterprise Edition 5.0, JavaMail API

Client Authentication

There are different techniques by which a client can establish it's identity to a server in a web environment. We repeatedly got questions from users who were not aware of the differences between those techniques. This section presents the common authentication techniques and puts them in the appropriate context.

Protocol Layers

Whenever a web client communicates with a web server, there are communications on different protocol layers. The following diagram shows the layers relevant for this discussion:

Application Layer
HTTP Layer
Transport Layer

At the top, there is the application layer. That is your client application communicating with a web application. The web application can for example be comprised of some servlets and JSPs. Your application will create and send requests and interpret the responses to achieve some purpose specific to your application.

The middle layer is where HTTP communication takes place. That is HttpClient or HttpComponents (or java.net.HttpURLConnection) exchanging HTTP messages with an HTTP server. HttpClient is not aware of the purpose of the messages. It merely knows how to generate and parse the HTTP message format, and how to handle some HTTP protocol details such as redirects and cookies.

At the bottom is the transport layer. That is the operation system connecting sockets to the computer on which the HTTP server is running. Or it is a TLS /SSL library creating a secure connection to that computer. On this layer, it is just binary data passing between the machines. The transport layer is not aware of the data being HTTP messages, let alone of the purpose for which the application is communicating.

This layered structure is not obvious to the casual user. When using a browser, you don't care whether your input is interpreted by the HTML engine on the application layer, or by the HTTP implementation on the HTTP layer, or by the TLS/SSL implementation on the bottom layer. Likewise, if you are developing a web application for example in a J2EE environment, you're looking at the protocol layers from the top down and don't care where the functionality is provided.

However, HttpClient is not a browser. If you are developing an application with HttpClient or some other HTTP implementation, you have to be aware of this structure. Client authentication can be performed on each of the three layers, but HttpClient is only responsible for the middle layer. You will need to use other APIs for authentication on the application or transport layer.

Basic, Digest, NTLM Authentication

These authentication techniques operate on the HTTP layer and are supported to some degree by HttpClient. Basic and Digest authentication are specified in RFC 2617. Both are fully supported by HttpClient. A browser will typically pop up an authentication dialog asking for the password to a specific server. The password will be asked only once for each session. If NTLM authentication is used and the password is the same as the Windows password, there may be no authentication dialog at all (single sign-on, SSO).

Basic authentication is considered insecure because it sends the user password in plain text (unprotected) with each request. That is only acceptable to some degree in intranets or when using TLS/SSL secure connections (HTTPS). It is generally not acceptable when using insecure connections over the internet.

Digest authentication is more secure than basic authentication because the password itself is not sent to the server. Instead, a hash of the password is created and sent. Digest authentication is rarely used, since in order to verify the hash, the server needs to know the user password in plain text. User repositories will typically not store passwords in plain text, but rather hashes of the password. Therefore, digest authentication can not be performed using such repositories. Storing passwords in plain text on the server backend systems introduces a weak spot into the server side architecture.

NTLM authentication comes in several varieties, all of which are proprietary authentication protocols by Microsoft. HttpClient partially supports NTLM authentication, as explained in the NTLM FAQ. The older versions, or lower levels, of NTLM authentication suffer from the same weakness as Basic authentication. The newer versions rectify this, but the protocols are not publicly documented. There are no open source implementations of the newer versions.

Form Based Authentication

The form based authentication technique operates on the application layer. When using a browser, username and password have to be entered in an HTML form. They will be sent to the server only once. After successful authentication, the server remembers that this client is authenticated and will not ask for the password again during that session. Session tracking often requires a cookie.

From the HttpClient perspective, submitting a form for client authentication is no different from submitting a form for a search query or any other purpose. Instructions on how to support session tracking and simulate form submission are available in the Client HTTP Programming Guide.

Form based authentication is more secure than basic authentication. Although it also transmits the password in plain text, it does so only once and not with every request. Still, when used over the internet, from based authentication should use a secure TLS/SSL connection at least for the login procedure. Afterwards, the session can be continued over plain connections, as the password is not sent again.

Certificate Based Authentication

Certificate based (client) authentication operates on the transport layer. It is part of the TLS/SSL protocol. Instead of using a password, certificate based authentication relies on public key cryptography. A private key is stored on the client and used to authenticate against the server. A browser will typically pop up a password dialog, asking for the password to the local key store. The password is never sent to the server, it is only used locally to gain access to the private key. The private key itself is never sent either.

From the HttpClient perspective, certificate based authentication is performed transparently when a secure TLS/SSL connection is established to a server. Transparently means that HttpClient doesn't know anything about the authentication at all. You have to install a SecureProtocolSocketFactory that automatically authenticates the client if requested by the server. This includes asking the user for the password to the local key store.

Certificate based client authentication is the most secure of the authentication techniques discussed here. The drawback is that it requires a complex public key infrastructure (PKI) to be put in place. Certificates holding the public keys for each client need to be available in the server's user repository, and the private keys have to be deployed on each client machine. Issuing the certificates for all clients is itself a complex task.

Further Reading

The J2EE 1.4 Tutorial, Security (SUN)

RFC 2617: HTTP Authentication: Basic and Digest Access Authentication

J2EE Form Based Authentication (onJava.com)

Wikipedia: Public Key Infrastructure

RFC 2246: The TLS Protocol Version 1.0

RFC 3546: Transport Layer Security (TLS) Extensions

Server Performing Login for Client

Once in a while, somebody wants a server or proxy to perform login to a different site on behalf of the client, then handing the session over to the client. Since the authentication is already performed by the server or proxy, the client is not supposed to ask the user for credentials.

In general, this is **not possible**. We mean it. It is **not** possible. Seriously. Unless very specific conditions are met, there is **no way**.

Why It Should Not Work

Imagine you are a server called Bob. Alice logs in to you, providing her credentials. Then Charlie appears, trying to access Alice's data. Charlie has no credentials, he just says: "Alice logged in on my behalf." Sounds fishy, does it not? *Would you believe Charlie?*

So, what are the conditions which might allow this to happen anyway? Firstly, authentication must apply to a session rather than the individual requests. This usually implies form-based authentication (see above) and the existence of a session ID. Secondly, the server configuration must be a bit negligent regarding security. The rest depends on the type of session tracking.

URL-based Session Tracking

If the user session is tracked in the URL, the handover is simple. Just send the URL including the session ID from the proxy or server to the client. If the server does not notice the change of the client IP address, you are lucky. A URL with session identifier could look like this: http_'_':://webmail.where.ever/xml/webmail;jsessionid=89702CCE20F2401326843985B0FB546F.TC159b

Cookie-based Session Tracking

If the user session is tracked with a session cookie, the handover is problematic. If your server or proxy is in the same domain as the site you want to login to, you can send the session cookie obtained from the target site on to the client, setting it at the domain level. This may or may not work, depending on additional security checks by the server. It may interfere with session tracking of other servers in the same domain, causing 'inexplicable' malfunctions of seemingly unrelated web applications for the client.

A better solution would be to create a Single Sign-On (SSO) domain for your server or proxy and the target site. Check the documentation of your application server(s) for information on Single Sign-On.

If your server or proxy is *not* in the same domain as the site you want to login to, you are out of luck.

If you find a way to make this work across domains, please report a security vulnerability against the browser.

If you don't know what all that stuff about cookies and domains means, you shouldn't implement this kind of security sensitive application in the first place.

Further Reading

Wikipedia: Alice and Bob

Java EE 5 Tutorial: Session Tracking

Wikipedia: Cross-site Cooking

Wikipedia: Single sign-on

Proxy Configuration

HttpClient takes proxy configuration data from HostConfiguration objects. These can either be passed explicitly when a method is executed (HttpClient 3.1), or the default configuration stored in the HttpClient object is used. Some of our users have the requirement to pick up external proxy configurations. The following sections discuss some options for obtaining external proxy configuration data.

Please note that HttpClient is designed to yield predictable results for applications in need of an embedded HTTP implementation. If HttpClient would automatically pick up external configuration data, predictability would be lost. Therefore, it remains the responsibility of the *application* to obtain proxy configuration data and to pass it to HttpClient. We will consider to provide helpers for this task if patches are contributed, but the responsibility for calling such helpers would still remain with the application.

System Properties

Up to and including Java 1.4, the standard Java implementation of HTTP, which is accessible through the HttpURLConnection class, expects proxy configuration data in system properties. The names of the properties that affect different protocols (HTTP, HTTPS, FTP,...) have changed over time. The two most prominent examples are `http.proxyHost` and `http.proxyPort`. You can read the values of these properties and supply the configuration as shown in the example below.

Note that other properties will also affect the standard Java HTTP implementation, for example a list of proxy exemptions in `http.nonProxyHost`. It is your application which must decide whether the external proxy configuration is applicable or not.

```
String proxyHost = System.getProperty("http.proxyHost");
int proxyPort = Integer.parseInt(System.getProperty("http.proxyPort"));

String url = "http://www.google.com";

HttpClient client = new HttpClient(new MultiThreadedHttpConnectionManager());
client.getHttpConnectionManager().getParams().setConnectionTimeout(30000);

client.getHostConfiguration().setProxy(proxyHost,proxyPort);

GetMethod get = new GetMethod(url);
get.setFollowRedirects(true);
String strGetResponseBody = null;

try {
    int iGetResultCode = client.executeMethod(get);
    strGetResponseBody = get.getResponseBodyAsString();
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    get.releaseConnection();
}
```

Since Java 5.0, the ProxySelector class allows for a more flexible, per-connection proxy configuration of the default HTTP implementation. Because HttpClient 3.1 is compatible with Java 1.2, it cannot support that class directly. However, your application can make use of the default `ProxySelector` to pick up the standard proxy configuration and pass it to HttpClient. HttpClient 4.0 requires Java 5 and will include an optional proxy selection mechanism based on `ProxySelector`. If you choose to obtain your proxy configuration elsewhere, you will of course still be able to do that, too.

Operating System Settings

On Linux and Unix systems, a proxy on the operating system level is typically set in environment variables. The Java method for reading environment variables is System.getenv. Unfortunately, it is deprecated and not even implemented in some Java versions (JDK 1.4?). The recommended replacement is to pass relevant environment variables as system properties by using `-Dname=value` options when starting your application. See above for reading a proxy configuration from system properties. Of course you can use custom property names in order to pass values without affecting the default HTTP implementation.

If using `-D` options is not feasible and you are stuck with a JVM that does not implement `System.getenv`, you can try to run a shell script using Runtime.exec. The shell script should print the relevant environment variables to standard output, from where your application can parse them.

On Windows systems, the proxy configuration is typically set in the registry. You can either use native code to read the registry, or try to run a shell script (batch file) as mentioned for the Linux/Unix case.

If you know something about proxy settings on Mac OS, please share that information. You can edit this Wiki page directly, or send a mail to one of our mailing lists.

Browser Settings

When an applet uses HttpURLConnection, the Java plug-in running the applet will automatically pick up the proxy configuration of the browser, and also cookies stored in the browser. This is described in the Java plug-in Developer Guide for JDK 1.4, chapter 5 and chapter 7 respectively. While this documentation explains the complexity of obtaining the proxy configuration, it does not mention a public API from which an application could pick it up.

Since Java 5.0, you can use the default ProxySelector mentioned in the section on system properties above. When running in the Java plug-in, it will provide access to the browser proxy configuration.

If you know how to access the browser proxy configuration in previous versions of the Java plug-in, please share that information. You can edit this Wiki page directly, or send a mail to one of our mailing lists.

If you know how to access the browser cookie store from an applet, please share that information too. Send a mail to one of our mailing lists or start a new section in this Wiki page.

Further Reading

Networking Properties, Java 1.4

Java Networking and Proxies, Java 5.0

Java Native Interface

Java Plug-in, Java 1.4

Proxy Configuration, Deployment Guide for Java 5.0