How to HTTP

by Randy Charles Morin

I know I keep putting off that how to send attachment with an SMTP message. Let's call it procrastination and let it go. In the meanwhile, I had the need to write a small utility to perform isolated HTTP requests. This led to this article. How do you do an HTTP request in low-level sockets?

This is the continuation in a long line of articles that I'm writing on the Internet Protocols. You can find the complete list of articles I've written to date on my Internet Programming webpage under Internet Programming Articles.

I'll start by encouraging you to read the HTTP 1.1 RFC. This is a good starting point for understanding the HTTP protocol. The RFC can be a little confusing and is pretty as wordy as a standard gets. So let me explain in my own English.

An HTTP interaction start by connecting the client to the server. As HTTP is almost always implemented over TCP/IP port 80, the interaction begins with a simple TCP/IP connect between the client and server. Once the connection is established, the client may execute commands on the server.

There are two commonly used ways of executing commands, via the GET method and via the POST method. Most webpages are retrieved with the GET method and a small percentage via the POST method. There are also other methods, but they are rarely used and will not be discussed here (OPTIONS, HEAD, PUT, DELETE, TRACE). Beyond the documented methods, there is also a provision for creating proprietary methods.

Let me begin by running thru a GET method sample. The typical GET command is one line "GET command version" followed by two linefeeds. If you wanted to retrieve the root page of a website, then you could execute the / command with the GET method.

```
GET / HTTP/1.0
```

Not all HTTP requests are this simple. Alternately, an HTTP request could have HTTP headers and a body. The equivalent POST method request would substibute the GET keyword with POST. Additionally, POST request more often than not have a request body, whereas GET requests don't.

```
POST /cgi/formmail.cgi HTTP/1.0
Content-length: 86

recipient=newsletter@kbcafe.com&subject=KB%20Newsletter&redirect=http://www.kbcafe.com
```

This has a lot to do with how CGI commands are executed, but I'll leave that discussion for another day.

The HTTP response looks a lot like the HTTP requests.

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 15 Jul 2001 03:13:08 GMT
Server: Apache/1.2.5 FrontPage/3.0.4
Location: http://www.kbcafe.com
Connection: close
Content-Type: text/html

<HTML><HEAD>
```

```
<TITLE>302 Moved Temporarily</TITLE>
</HEAD><BODY>
<H1>Moved Temporarily</H1>
The document has moved <A HREF="http://www.kbcafe.com">here</A>.<P>
</BODY></HTML>
```

The first line of the HTTP response is the status line. The status line is divided into three sections, "version status description". The version will typically be HTTP/1.0 or HTTP/1.1. The returned status can vary but common statuses are presented in the table.

Code	Description
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Moved Temporarily
303	See Other
304	Not Modified
305	Use Proxy
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Time-out
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Large
415	Unsupported Media Type
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Time-out
505	HTTP Version not supported

It is common in POST request to use the Content-length header to give the server a hint as to the length of the incoming data section (body). The length of the body does not include any terminating linefeeds.

Let's start looking at some code.

```
//////////////////////////////////////////////////////////////////////
//
// http.h: implementation of the Http class.
//
//////////////////////////////////////////////////////////////////////

#ifndef sportmarkets_HTTP_H
#define sportmarkets_HTTP_H

#include
#include
#include "socket.h"
```

```
namespace kbcafe
{

class Http : public Socket
{
public:
	Http(const std::string & strServer);
	virtual ~Http();

	std::string Http::Request(const std::string & command,
		const std::string & querystring="",
		bool ispost=false);
	virtual std::string Response();

};

inline Http::Http(const std::string & strServer)
{
	Connect(strServer, "http");

}

inline Http::~Http()
{

};

inline std::string Http::Request(const std::string & command,
		const std::string & querystring,
		bool ispost)
{
	std::stringstream ss;
	if (ispost)
	{
		ss << "POST ";
	}
	else
	{
		ss << "GET ";
	}
	ss << command;
	if (!querystring.empty() && !ispost)
	{
		ss << "?" << querystring;
	}

	if (!querystring.empty() && ispost)
	{
		ss << " HTTP/1.0\n"
			<< "Content-length: " << querystring.length() << "\n\n"
			<< querystring << "\n\n";
	}
	else
	{
		ss << " HTTP/1.0\n\n";
	}

	Write(ss.str());
	return Response();
};

inline std::string Http::Response()
{
	int roundtrips = 0;
	bool gotheaders = false;
	bool lastnewline = false;

	std::string response;
	while(true)
	{
		char buffer[1];
```

```
				int n = ::recv(m_socket, buffer, 1, 0);
				if (n == -1)
				{
					throw SocketException("socket read failed");
				};

				if (n == 0)
				{
					std::cout << response << std::endl;
					return response;
				}

				response += std::string(buffer, n);

				if (n == '\n')
				{
					if (lastnewline)
					{
						if (gotheaders)
						{
							std::cout << response << std::endl;
							return response;
						}
						gotheaders = true;
					}
					lastnewline = true;
				}
				else
				{
					lastnewline = false;
				}

				roundtrips++;
				if (roundtrips > 100000)
				{
				throw SocketException("socket read timeout");
				}
		}
}

};

#endif
```

I inherited my Http class from the Socket class that I developed in previous articles of this series. You must download both the Http inline class and Socket inline class if you are going to compile this.

Using the Http class is easy. Just instantiate an object and starting executing commands by call the Request method. The Response method is automatically called by the Request method, so you can assume it's a utility method that normally should be private or protected. I have considered using HTTP as an asynchronous transport and that's why I didn't protect the Response method.

```
int main(int argc, char* argv[])
{
	kbcafe::Http http("www.kbcafe.com");
	std::string str = http.Request("/");

	return 0;
}
```

I still haven't implemented the complete differences between GET and POST method request, but you can specify either method using the third parameter. The first parameter of the Request method is the command you are executing. As it relates to web surfing, this it he page you want to pull from the website.

You may also have seen GET parameters passed to a website by using the CGI notation using a question mark between the command to the left and the parameters to the right. If you want to pass parameters to your GET request, then you can do so by specifying those parameters as the second parameter in the Request method.

```
GET /cgi/redirect.cgi?http://www.kbcafe.com HTTP/1.0
```

In a future article in this series, I will show how to send attachments with the SMTP protocol. Like I promised before a half dozen times. I also hope to write on the CGI topic a bit. But generally, this series is finding its way towards the topic of SOAP. We'll be there soon.