# COMP202-17B - HTTP Server

July 31, 2017

## 1 Introduction

This practical will introduce you to programming network applications with Java using threads. You will also learn about HyperText Transfer Protocol (HTTP) – the protocol that you use when you browse the web. You will do this by writing a small HTTP server. The web server you write will not be as featured as the web servers used in the Internet, though it may become clear how you would go about adding features to your finished product.

This practical will involve you using the ServerSocket, Socket, String, BufferedReader, and BufferedOutputStream Java classes. Please take some time to look through the Java documentation provided for these classes. You can find the documentation at `http://docs.oracle.com/javase/8/docs/api/index.html`

You can also find reasonable documentation of the HTTP protocol online at Wikipedia: `http://en.wikipedia.org/wiki/HTTP`

## 2 Academic Integrity

The files you submit **must** be your own work. You may discuss the assignment with others but the actual code you submit must be your own. You must fully also understand your code and be capable of reproducing and modifying it. If there is anything in your code that you don't understand, seek help until you do.

You **must** submit your files to Moodle in order to receive any marks recorded on your verification page.

This assignment is due August 14th by 11am and worth 6% of your final grade.

## 3 Overview of HTTP

Web-servers listen on a socket, usually port 80. Web-servers and browsers use a protocol to retrieve files, which the browsers then display. The request for a page is initiated by the web-browser. It opens a connection and sends a request message consisting of lines of text, each line separated by a CR/LF pair. The browser signals to the web-server that the request message is complete by sending an empty line. Here is an example:

```
GET /ChatServer.java HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:47.0)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
```

Only the first line matters for our web server. The first line has a command verb (GET), the name of the file being requested (ChatServer.java) and the protocol version in use (HTTP/1.1). The other lines provide additional information about the web-browser in use, the language requested if translations are available, and the web-server specified in the browser's address bar.

The GET command is used to request a file, such as an html page, an image file, or other file. It is the most commonly used command and the only one you are required to implement. The web-server responds on the same socket as used for the request with something that looks like this:

```
HTTP/1.1 200 OK
Date: Wed, 03 Aug 2016 17:42:41 GMT
Server: Apache/2.4.23 (FreeBSD) OpenSSL/1.0.1s-freebsd PHP/5.6.24
Last-Modified: Tue, 25 Oct 2011 02:02:02 GMT
ETag: "817-4b015ece3da80"
Accept-Ranges: bytes
Content-Length: 2071
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/x-java-source

<file contents go here>
```

As before, the first line is the most important. It contains the protocol version (HTTP/1.1), a status code (200) and a brief text message explaining the status code (OK). Then comes more information about the software running on the web-server, as well as the file being sent back including the date it was last modified, a unique ID, the number of bytes being sent back, and what it will do with the socket when the request is done. As with the request header, the response header is in plain text, and the end of the response header is signalled with an empty line.

The data that follows may not necessarily be a text file; it might be an image file, which is a binary file. Therefore, when your web-server sends the file to the browser, it must be careful not to corrupt it by reading it a line at a time and storing it in a String, as the String will not retain all of the bytes read.

Our web server will use two status codes. 200 means that everything is OK and the file requested is going to be returned. 404 is used when the file does not exist at the web-server.

## 3.1 The basic console application

The server will be a console application. Start with a program that looks like the following. The file starts with all the classes necessary for this project imported. Add a line to write "web server starting" to the console, and test immediately.

```
import java.net.*;
import java.io.*;
import java.util.*;

class HttpServer
{
    public static void main(String args[])
    {
        //write something to the console here
    }
}
```

You need to choose a port for your web server's ServerSocket. You are unable to use any port numbers less than 1024 in the lab, so you will not be able to use the default web-server port number of 80. 8080 and 8800 are popular alternatives. You can code the port number into your application, or have it specified on the command line.

Testing: Ensure your program compiles and runs. It will not produce output, but it should complete.

## 3.2 Accept a connection

The next step is to accept a connection and then spawn a thread to then process that connection. Declare a new class HttpServerSession which extends Thread. The HttpServerSession constructor should take a single argument – the socket that was accepted, and save it in a variable private to that class. Take a look at ChatServer.java in Moodle for an idea about this.

Testing: Add a statement to HttpServer::main() that prints a message when a connection is made. Then, open a web browser and enter `http://localhost:8080/ChatServer.java` in the location bar. Your HttpServer should print out a message to the console saying a connection was received, specifying which IP address made the connection.

## 3.3 Look at the HTTP request

In the HttpServerSession::run method, declare a BufferedReader that is connected to the socket's InputStream. Get the request by using the readLine() method. This line should be stored in a special variable – the lines following that line will be ignored by your web server. Ensure that you handle any special conditions, such as the socket being closed by the client before it even sends a request. Now is a good time to add code to your HttpServer that closes the connected Socket when you have finished with it.

Testing: add a statement to your run method to print out the request line to the console. Run your web-server and browser as before; you should see the request printed out to the console by your web-server.

## 3.4 Extract the filename

You can obtain each of the three parts of the request string by using code like this:

```
String parts[] = request.split(" ");
```

Abort processing unless there are exactly three parts are present in the array (using parts.length). The first part should be the “GET” string; ensure that is the case by using parts[0].compareTo(“GET”); this method returns zero if parts[0] is the “GET” string. The next entry in the array is the file being requested.

Testing: Modify the previous debugging statement to print out just the file being requested. It should print `/ChatServer.java`. You can remove the forward slash from the string by doing something like:

```
String filename = parts[1].substring(1).
```

The rest of the request header is not relevant for this assignment, so skip over it until you get to the empty line. The following code sketch will help:

```
while(true) {
  String line = reader.readLine();
  if(line == null) {
    /* handle EOF */
  }
  if(line.compareTo("") == 0)
    break;
}
```

### 3.5 Send back a response

This part is a little more complicated, due to Java's PrintWriter::println behaving differently on various platforms, and HTTP's requirement that each line have a CR and LF pair. Instead of using PrintWriter, we will use BufferedOutputStream.

You can add a method to your HttpServerSession that will mimic the println method found in PrintWriter, but be compliant with what HTTP requires.

```
private void println(BufferedOutputStream bos, String s)
    throws IOException
{
  String news = s + "\r\n";
  byte[] array = news.getBytes();
  for(int i=0; i<array.length; i++)
    bos.write(array[i]);
  return;
}
```

As said earlier, you only need to send back a single line of status text. Try sending back a 200 message like the one in the earlier example, ensure you send an empty line, and then send the string "Hello World".

Test: run your web browser now. It should display "Hello World" in the browser.

### 3.6 Return the file

In the previous step, you sent back a response with "Hello World" where the actual contents of the file should go. The goal of this step is to return the actual contents of the file. To do this, you're going to have to declare a byte array of a fixed size, open the file with a FileInputStream, read from the file with the FileInputStream::read method, and with each read send the byte array to the client using the BufferedOutputStream::write method. Be sure to catch end of file (when FileInputStream::read returns -1). At the end of the file, ensure that you then flush the output before you return.

Test: You should have a working web server. Test that you do by putting a web page in your directory, and also test image files.

### 3.7 Deal with missing files

You may have noticed your web server throwing an exception when your browser automatically requests favicon.ico. Extend your web-server to catch that exception and return the 404 message detailed earlier. Extend your web-server to log a message that says whether or not the requested file was found or not.

Test: Check that your web-server sends a 404 message by requesting a file that you know does not exist.

## 3.8 Running over a slow line

People sometimes use web browsers over slow connections, such as dial-up. It is important that a web-server can service all requests simultaneously, rather than be held up servicing a slow client. You have achieved this by threading your web-server like how the ChatServer was.

It is difficult to provide a slow environment in the lab, though you can modify your HttpServer to make it go slow when sending a file back. To check this, please add a sleep(1000) to the loop where you read the file and send it to the client in pieces. This will increase the time it takes a large file to be sent back.

Test: place some reasonably large (50K) image files in your directory. When your browser fetches an image file, you should be able to see the browser gradually render the image as each chunk is delivered to it. If you do not see the effect, try a different image file, or create a large text file with lines repeated over and over. You should see the browser's scroll bar gradually provide greater scrolling range as each chunk arrives.

## 3.9 Student selected extension

Extend your HttpServer in some small way to have a new feature. Suggested features are:

- the ability to support virtual hosting. A single IP address can host multiple websites. The client tells the webserver which website it is using by passing a Host: line in the request it makes. You can return a different page if the client asks for a page on localhost compared to when using the DNS name of the machine (e.g. cms-r1-17).
- the ability for your web-server to return a default page when no file is specified in the location bar. For example, `http://localhost:8080/` could return index.html, if it is found.
- the ability to specify a location in the file system where your web-server will look for files, rather than the directory from which it is run.
- return one of the other HTTP status codes; for example, if the request is formatted incorrectly, return the appropriate code. Take a look at the wikipedia page on what the various codes are.
- return one or two headers in your response. For example, consider returning the Last-Modified or Content-Length headers.

The extension you make can be of your choosing; these are merely suggestions.

## VERIFICATION PAGE

Name: ____________________

Id: _____________________

Date: _____________________

Note: this assignment is due August 14th by 11am and worth 6% of your final grade. You must also submit your code to the Moodle assignment page after you have had the assignment verified. It will be marked out of 6 and is worth up to 6% of your final grade.

1. Explain to the demo the structure of your HttpServer program.
2. Allow the demo to read and test your code.
3. Demonstrate that your HttpServer can serve multiple browser requests simultaneously, either by having multiple browser sessions fetching different html pages, or by having multiple lab machines using your HttpServer.
4. Demonstrate that your HttpServer can serve large image files correctly.
5. Describe to the demo the extension you made to your web-server.