# Part 1

1. HTTP Fields and meaning

The request line is the first line of an HTTP Request, and is something like:

GET /software/htp/cics/index.html HTTP/1.1

The following works constitute a request line

1. METHOD: This is the HTTP Method to which a request is being made, example, GET, PUT, POST, DELETE, etc.
2. URI: The URI the request is made to. This can be \\*(to indicate the general server), an absolute (URI, eg. http://google.com), a relative URI(eg. /software/htp/cics/index.html), in which case the absolute is specifed by the HOST header.
3. HTTPVERSION: The version of the HTTP protocol, eg. HTTP/1.1 or HTTP/1.0 (specifies the format of the request)

2. GET parameters

The parameters of a get request are appended to the full URI to form the request URL, in the form of key=value for each parameter, separated by an '&'. For example, if we have two parameters, a = 1 and b = 2 for a request to http://google.com, then the full request URL will be:

http://google.com?a=1&b=2

3. Percent encoding

Percent encoding or URI encoding is a method to encode special characters, such as the &, !, /, which have special meaning, in an HTTP request URI. This is also used to encode form data for post request. For each of the reserved charcters of % encoding, we have a fixed encoding. For example, & == %26 and / == %2F. So to encode two parameters a=1&2 and b=3//4 into a GET URI, we would have the encoded URI:

base.com?a=1%26&b=3%2F%2F4

4. The response line, similar to a request line, is the first line of an HTTP response, and looks something like

HTTP/1.1 200 OK

Where the three fields are HTTPVERSION (similar to in the request), the status code, and the reason for the status code, followed by CLRF(to indicate the EOL). The reason is just one string. The http status code can be of the format 1xx, 2xx, 3xx, 4xx, 5xx, where ‘x’ is a digit from 0 - 9.

Each series of status codes have a special meaning, for example, 1xx is information, and 2xx is success, 3xx is redirection, 4xx is server error, 5xx is client error.

5. The HOST header of an HTTP request specifies the domain name of the server and the TCP port on which the server is listening. The HOST field is required becuase it tells the webserver which virtual host the request is addressing, in case of multiple possible hosts (even for the same physical host). For example, if the request is made to the physical host with IP 8.8.8.8 and port 80, it is possible, there are two virtual hosts, google.com and mail.google.com running on it. The webserver needs to know which host to forward the request to.

6. The Content-type header of HTTP identifies the MIME type of the resource that is being sent. For example, if the response is coming in as an HTML page, a Content-type of text/html is used. If the response is in the JSON format, then application/json is used. This helps in the proper parsing of the request body, for example in client side Javascript, where JSON objects are automatically parsed.

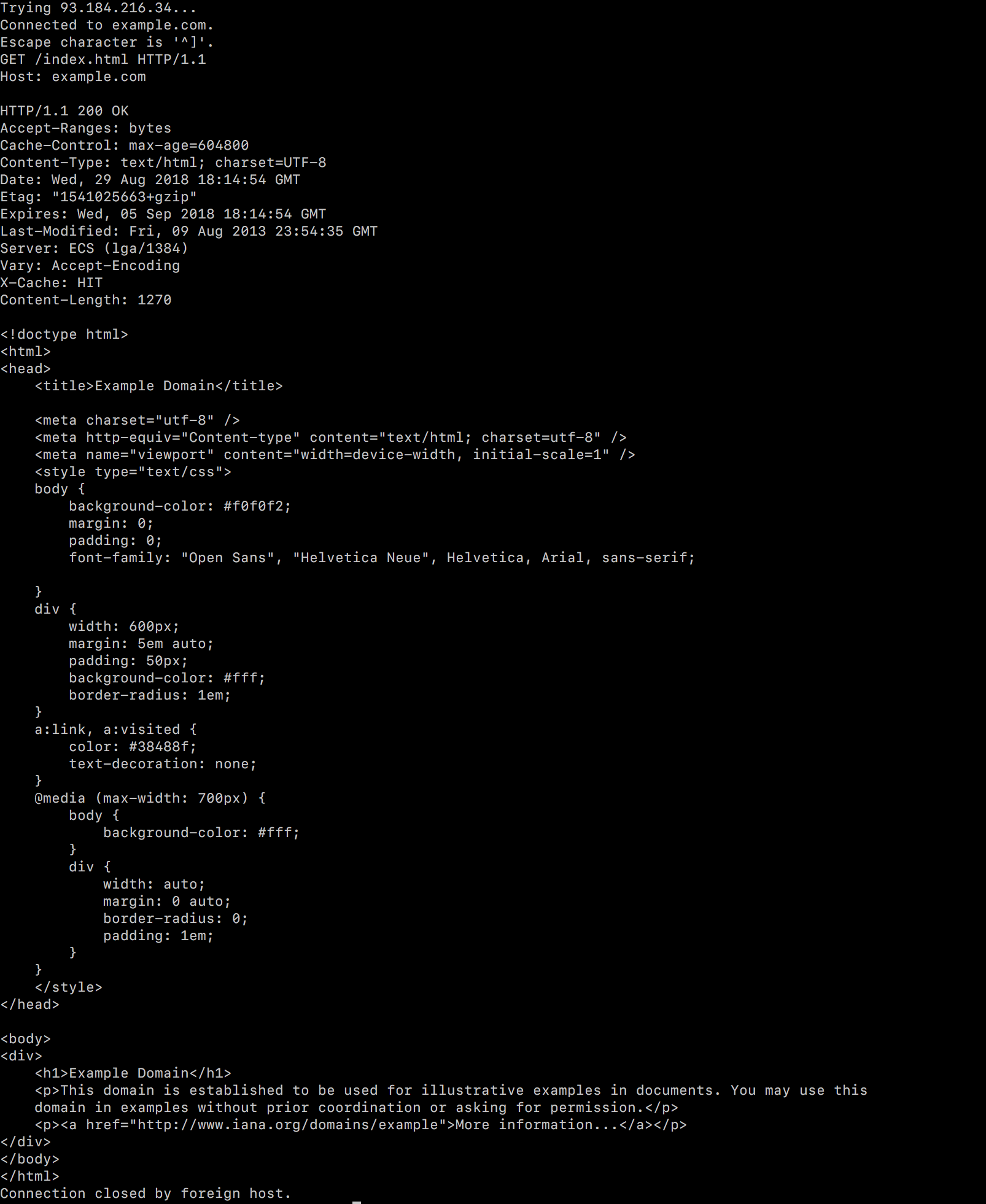
7. The telnet command is:

telnet example.com 80

GET /index.html HTTP/1.1

Host: example.com

Output:



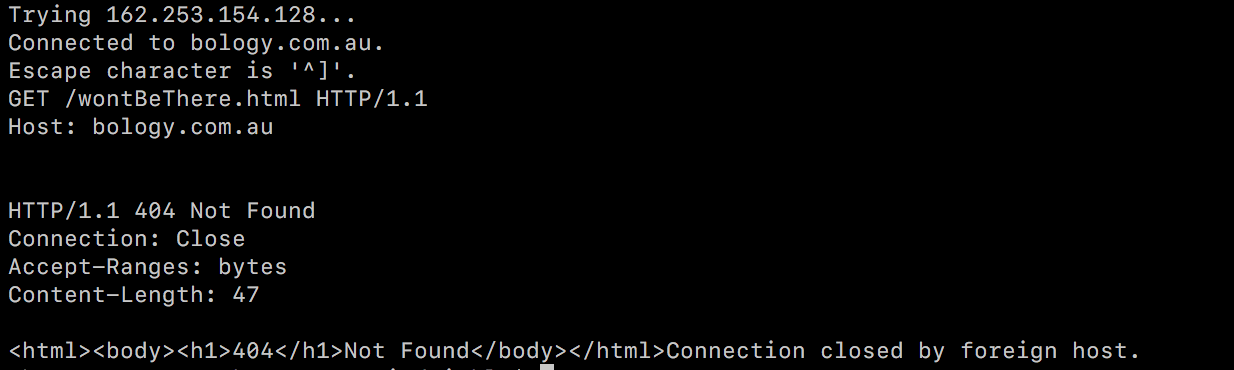
8. The telnet command is:

telnet bology.com.au

GET /wontBeThere.html HTTP/1.1

Host: bology.com.au

Output:



9. To run the netSrv.c file given, we do the following:

> git clone <https://github.com/drbraithw8/CscNetlib.git> # Clone the repo

> cd CscNetLib # Change into directory

> git checkout df0d2c0d5c1 # Checkout to the correct commit

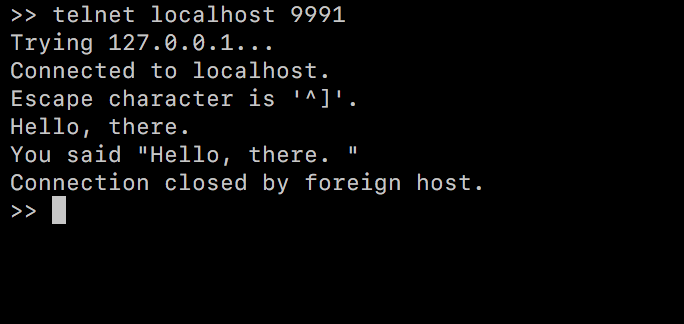
> make # Build the library

> cd .. # Go back to the program directory

> # Build the program

> gcc -I. -o netSrv netSrvDemo.c CscNetlib/libCscNet.a

> # Run it





10. Here is the entire commented main() function. Each line contains // comments after it.

int main(int argc, char \*\*argv)

{ int fd0; // Define a file descriptor

char line[MaxLineLen+1]; // Create line buffer

// Create netSrv object.

csc\_srv\_t \*ntp = csc\_srv\_new(); assert(ntp!=NULL); // Create server object

int ret = csc\_srv\_setAddr(ntp, "TCP", "127.0.0.1", 9991, -1); assert(ret); // Set the server address, port and protocol

// For each successful connection.

while ((fd0 = csc\_srv\_accept(ntp)) >= 0) // Accept a connection

{ fprintf(stdout, "Connection from %s\n", csc\_srv\_acceptAddr(ntp)); // Print out the connection details to stdout

// Convert file descriptor to input and output streams.

int fd1 = dup(fd0); assert(fd1!=-1); // Duplicate file descriptor

FILE \*tcpIn = fdopen(fd0, "r"); assert(tcpIn!=NULL); // Open for reading on the stream

FILE \*tcpOut = fdopen(fd1, "w"); assert(tcpOut!=NULL); // Open for writing on the TCP stream

// Do input/output.

csc\_fgetline(tcpIn,line,MaxLineLen); // Get a line from the input stream, into line of len maxLineLen

fprintf(stdout, "Got line: \"%s\"\n", line); // Print the line to stdout

fprintf(tcpOut, "You said \"%s\"\n", line); // Send the line back to the output TCP stream

fflush(tcpOut); // Flush the output stream (if buffered)

// Close the streams.

fclose(tcpOut);

fclose(tcpIn);

}

csc\_srv\_free(ntp); // Close the server and free the port

exit(0); // Exit with status 0

}