



Home / My courses / Previous Term (Term 3, 2015-2016) / CS 1103 - T3 2015-2016 / 10 March - 16 March /
Lab 11 Unit 7

Lab 11: A Web Server

The goal of this lab is to write a simple, but functional, web server that is capable of sending files to a web browser on request. The web server must create a listening socket and accept connections. It must implement just enough of the HTTP/1.1 protocol to enable it to read requests for files and to send the requested files. It should also be able to send error responses to the client when appropriate.

Seeing HTTP in Action

It would be useful to see the HTTP protocol in action before beginning your work. Your program will have to **read** HTTP requests and **send** HTTP responses

To see what an HTTP request might look like, you can run the program *ReadRequest*, which can be found in the code directory. To run the program, cd to that directory on the command line, and enter the command `java ReadRequest`. The program is ready to accept requests from web browsers on port 50505. (It will continue to do so until you terminate the program.) When it receives a request, it simply prints out the request, including all headers. It then closes the connection; it does not send any response back to the browser. To see a request, open a web browser, and enter the URL `http://localhost:50505/path/file.txt` into the browser's location box. The request will be output to the console window where you are running the *ReadRequest* program. The first line of the request should be `GET /path/file.txt HTTP/1.1`. This will be followed by several headers.

To send an HTTP request and see the response, you can use the standard telnet program. Type the following lines carefully in a console window:

```
telnet google.com 80
```

Add a blank line, by pressing return twice after typing the last line. This sends a legal HTTP request for the file *index.html*. The web server on `math.hws.edu` will respond by sending a *status line* followed by some headers and a blank line, followed by the contents of the file. You can try to get some error responses, if you want, such as by asking for a non-existent file instead of *index.html* or by using a different method instead of GET.

Start a new Eclipse project and create your main program class. The basic programming for a server is pretty standard: It just has to create a *ServerSocket* and use it to accept connection requests. For the main routine in your program, you can just use the main routine from *ReadRequest.java*. You can copy-and-paste it from here:

```
public static void main(String[] args) {
    ServerSocket serverSocket;
    try {
        serverSocket = new ServerSocket(ListeningPort);
    }
    catch (Exception e) {
        System.out.println("Failed to create listening socket.");
        return;
    }
    System.out.println("Listening on port " + ListeningPort);
    try {
        while (true) {
            Socket connection = serverSocket.accept();
            System.out.println("\nConnection from "
                + connection.getRemoteSocketAddress());
            handleConnection(connection);
        }
    }
    catch (Exception e) {
        System.out.println("Server socket shut down unexpectedly!");
        System.out.println("Error: " + e);
        System.out.println("Exiting.");
    }
}
```

The problem is to write the *handleConnection()* method. This method gets an already connected socket as a parameter. It can use that socket to get an *InputStream* and an *OutputStream* for communicating over the connection. It can read the request from the input stream and send a response on the output stream. Finally, it can close the connection. You can use some ideas (and maybe some code) from the *handleConnection* method in *ReadRequest.java*, but the method that you are writing will be a good deal more complicated.

It is very important that:

- (a) the *handleConnection* method should catch and handle any exception that occurs, so that the exception does not crash the whole sever, and
- (b) the socket must be closed at the end of the method. Use a try..catch..finally statement to make sure that (a) and (b) are done correctly. See *ReadRequest.java* for an example.

Your program should be ready to send error responses to the client, as well as fulfilling legitimate requests. The next section asks you to implement error-handling. For now, you can simply return from the *handleConnection* method when you detect an error.

The first three tokens that you read from the input stream should be "GET", the path to the file that is being requested, and "HTTP/1.1" (or, just possibly, "HTTP/1.0"). If you can't read three tokens, or if they are not of the

expected form, you should consider that to be an error.

Assuming that the request has the correct form, you want to try to find the requested file and send it in a response over the output stream. All the files that are available on your server should be in some directory, which is called the *root directory* of the server. You can use any directory that you want as your root directory, as long as you can read that directory. For example, if you want to serve up files from Professor Corliss's web directory, you can set

```
String rootDirectory = "/home/mcorliss/www"
```

Of course, you could also use your own www directory. Assuming that *rootDirectory* is the root directory of your server and *pathToFile* is the path to the file as given in the request from the browser, then the full name of the file is *rootDirectory + pathToFile*, and you can create a File object to represent the file that is being requested as follows:

```
File file = new File(rootDirectory + pathToFile);
```

Note that:

- *the method file.exists()* can be used to check whether the requested file actually exists
- *the method file.isDirectory()* tests whether the file is actually a directory rather than a regular file
- *the method file.canRead()* tests whether you can read the file
- *the method file.length()* tells you the length of the file, that is, how many bytes of data it contains.

Once you have found the file and know that it is a regular file and that you can read it, you are ready to send a response to the browser. (If the file is a directory, you can't send it, but a typical server in this case will send the contents of a file named *index.html* in that directory, if it exists. You can think about how to implement this if you want.) Before you send the file itself, you have to send the status line, some headers, and an empty line. You can use a *PrintWriter* to do this. **However, the HTTP protocol specifies that ends-of-line should be indicated by "\r\n" rather than the "\n" that is standard in Linux. Although I have found that it doesn't matter when sending text documents, it does seem to matter when sending images. So, instead of using out.println(x), you should use out.print(x + "\r\n") to send a line of text, and use out.print("\r\n") to send a blank line.** The status line to indicate a good response should be:

```
HTTP/1.1 200 OK
```

(with "\r\n" at the end). You should send three headers: "Connection", "Content-Length", and "Content-Type". For the Connection header, you can send

```
Connection: close
```

which informs the browser that you are going to close the connection after sending the file. The Content-Length header should specify the number of bytes in the file, which you can find with the *file.length()* method. The Content-Type header tells the browser what kind of data is in the file. It can generally be determined from the extension part of the file name. Here is a method that will return the proper content type for many kinds of files:

```
private static String getMimeType(String fileName) {  
    int pos = fileName.lastIndexOf('.');  
    if (pos < 0) // no file extension in name  
        return "x-application/x-unknown";  
    String ext = fileName.substring(pos+1).toLowerCase();  
    if (ext.equals("txt")) return "text/plain";
```

```

else if (ext.equals("html")) return "text/html";
else if (ext.equals("htm")) return "text/html";
else if (ext.equals("css")) return "text/css";
else if (ext.equals("js")) return "text/javascript";
else if (ext.equals("java")) return "text/x-java";
else if (ext.equals("jpeg")) return "image/jpeg";
else if (ext.equals("jpg")) return "image/jpeg";
else if (ext.equals("png")) return "image/png";
else if (ext.equals("gif")) return "image/gif";
else if (ext.equals("ico")) return "image/x-icon";
else if (ext.equals("class")) return "application/java-vm";
else if (ext.equals("jar")) return "application/java-archive";
else if (ext.equals("zip")) return "application/zip";
else if (ext.equals("xml")) return "application/xml";
else if (ext.equals("xhtml")) return "application/xhtml+xml";
else return "x-application/x-unknown";

    // Note: x-application/x-unknown is something made up;
    // it will probably make the browser offer to save the file.
}

```

Putting all this together, the beginning of the response might look something like:

```

HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Content-Length: 3572

```

with a blank line at the end. **And don't forget to flush the PrintWriter after sending this data.**

Finally, it's time to send the data from the file itself! The file is not necessarily text, and in any case it should be sent as a stream of bytes. The following method can be used to copy the content of the file to the socket's output stream:

```

private static void sendFile(File file, OutputStream socketOut) throws
IOException {
    InputStream in = new BufferedInputStream(new FileInputStream(file));
    OutputStream out = new BufferedOutputStream(socketOut);
    while (true) {
        int x = in.read(); // read one byte from file
        if (x < 0)
            break; // end of file reached
        out.write(x); // write the byte to the socket
    }
    out.flush();
}

```

At this point, your web server program should work for legal requests for valid files. Note, by the way, that you can try telnetting to your server and sending it a request, to see what response it actually sends.

Error Handling

If something goes wrong with a request, the server should nevertheless send a response. The first line of a response always contains a status code indicating the status of the response, as well as a textual description of the status. For a normal response, the status code is 200, indicating success. Status codes for errors are in the 400s and 500s. For example, the status code 404 is used when the request asks for a file that does not exist on the server. The first line of the response in this case is "HTTP/1.1 404 Not

Found". The response should also include, at least, a "Content-type" header and the content of the page that is to be displayed in the browser to inform the user of the error. For example, the complete response for a 404 error might look like:

```
HTTP/1.1 404 Not Found
```

```
Connection: close
```

```
Content-Type: text/html
```

```
<html><head><title>Error</title></head><body>
<h2>Error: 404 Not Found</h2>
<p>The resource that you requested does not exist on this server.</p>
</body></html>
```

You could also use a content type of "text/plain", and send the response using plain text rather than HTML. I suggest that you write a method such as

```
static void sendErrorResponse(int errorCode, OutputStream socketOut)
```

to send error responses. I suggest that this method catch any exceptions that occur and do nothing, since there's really nothing reasonable to do if an error occurs while you are trying to send an error message. Note that this method should **not** be called if you already started to send a "200 OK" response (or another error response) -- if an error occurs at that point, it's probably an unrecoverable network error, and you might as well just end the *handleConnection* method. (Also, remember to exit *handleConnection* after sending the error response; don't try to continue processing after an error!)

Here are some error responses that your program might want to send. You should at least be able to send 404 and 501.

- **HTTP/1.1 404 Not Found** --- discussed above.
- **HTTP/1.1 403 Forbidden** --- you could send this if the requested file exists, but you don't have permission to read it (or just send 404).
- **HTTP/1.1 400 Bad Request** --- you could send this if the syntax of the request is bad, for example if the third token is not "HTTP/1.1" or "HTTP/1.0" (or just send 501 or 500).
- **HTTP/1.1 501 Not Implemented** --- if the method in the request is anything other than "GET".
- **HTTP/1.1 500 Internal Server Error** --- you could send this if you catch some unexpected error in the *handleConnection* method.

At this point, your server should work pretty well with a web browser.

Threads

The server that you have written is *single-threaded*. It can only handle one request at a time. If a second request comes in while you are working on another request, the second request will have to wait until you are finished with the first request, even if that takes a long time because you are sending a large file over a slow network. This is not acceptable for a real server. A real server should be multi-threaded, with several threads to handle connections.

An easy way to write a multi-threaded server is to start a new thread to handle each connection request. New requests can be handled as they arrive, even if previous requests are still being handled by other threads. (Note that this solution is still not acceptable for real servers, because starting a new thread is a relatively time-consuming thing, and because you don't want to have the possibility of having too many threads running at the same time.)

To make your server into a multi-threaded server, you will need a subclass of *Thread*. The class needs a *run()* method to specify the task that the thread will perform. In this case, it should handle one connection request. We can pass the socket for that connection to the constructor of the class. Here's the class. Copy it into your program:

```
private static class ConnectionThread extends Thread {
    Socket connection;
    ConnectionThread(Socket connection) {
        this.connection = connection;
    }
    public void run() {
        handleConnection(connection);
    }
}
```

Now, in the main routine, instead of calling *handleConnection* directly, you will create and start a thread of type *ConnectionThread*:

```
ConnectionThread thread = new ConnectionThread(connection);
thread.start();
```

That's all there is to it! With this change, you should have a minimal but functional multi-threaded web server.

Last modified: Thursday, 30 July 2015, 1:21 AM

Navigation

Home

- My home

- Site pages

- My profile

- Current course

- CS 1103 - T3 2015-2016

- Participants

Badges

28 January - 3 February

4 February - 10 February

11 February - 17 February

18 February - 24 February

25 February - 2 March


3 March - 9 March


10 March - 16 March

 Learning Guide Unit 7


 Discussion Forum Unit 7


 Assignment Unit 7


 Learning Journal Unit 7

 Self-Quiz Unit 7

 Code Unit 7

 **Lab 11 Unit 7**

 Solutions for Assignment Unit 6

 Solutions for Exercises Unit 6

17 March - 23 March

24 March - 30 March

■ Kaltura Media Gallery

My courses

Administration

Course administration

My profile settings