Web Server Basics
Cpt S 422 Homework Assignment
by Evan Olds

Assignment Instructions:

Read all instructions *carefully* before you write any code.

In this assignment you will build a simple console/terminal application that provides minimal web server functionality. You will be able to browse to it on your local machine through your web browser and see the response. You will find that the basic web server logic is simple, it's the error handling aspects that get tricky (in the testing more so than the implementation).

Create the WebServer class

Create a class named "WebServer" in the CS422 namespace. In this class you will implement the features listed below. This is just a short overview in list form. Details about each requirement follow after the list.

- Have a static function called "Start" (match the casing and make sure it's static!) that takes an integer as a parameter for the port to listen on, and a string for a response template
- Have error checking code to recognize invalid HTTP requests and terminate the underlying TCP connection

Implementing the Start function:

The static "Start" function, as mentioned earlier, takes an integer port for the TCP connection as the first parameter. The second parameter is a template string for the HTTP response, which is discussed more later. This Start function is a blocking function that creates a TcpListenter and listens for a connection on the specified port. Everything is done synchronously and you may NOT create additional threads. That will come in later assignments. This one is just single-threaded.

As discussed in class, port 80 is the default for HTTP, but applications generally need to have administrative privilege to use this port. The grading process will use a port that will allow the app to function as a web server without admin privileges. The signature for the Start function is shown below.

```
public static bool Start(int port, string responseTemplate)
```

Create the TcpListener in this function, start it listening, and then making a blocking call to accept a client with the AcceptTcpClient function. If the function fails to accept an incoming client, just return **false** from the function in an orderly fashion (not by letting an exception bubble up). A return value of false indicates failure to connect to a client that sends a valid HTTP request. If the socket connection is made, you will read data from the client and do one of two things:

- If it's a valid HTTP request, you will write a valid HTTP response to the socket using the response template string. True will be returned from the function after you write your response to the client and close the connection.
- If it's NOT a valid HTTP request, you will sever the socket connection and return false from the function.

You will have to do some reading about the format of an HTTP request. You only need to support the "GET" method at this time.

The Response Template String:

The response template string is a string that will contain the general structure of an HTTP response, with format specifiers for parts that can change. The format specifiers for the string.Format function will be used, implying that your response to a valid HTTP request replaces the format specifiers with bits of information as described below.

Format Specifier	**Replace with…**
{0}	Your 8-digit ID number as a string
{1}	DateTime.Now
{2}	URL that was requested. Note that this will always be a non-empty string in a valid HTTP request.

Note that the first two can just be hard-coded, but the third needs to be parsed from the request. Use the string.Format function with the template string to produce the response to a valid request. In this case the template is designed to contain the ENTIRE HTTP response. In later versions of the server you will be writing the response code, response headers and response body on your own. But for now it's given to you. Below is an example of a template string declaration, which also serves to show the format of an HTTP response.

```
private const string DefaultTemplate =
        "HTTP/1.1 200 OK\r\n" +
        "Content-Type: text/html\r\n" +
        "\r\n\r\n" +
        "<html>ID Number: {0}<br>" +
        "DateTime.Now: {1}<br>" +
        "Requested URL: {2}</html>";
```

The Remaining Details:

Your server must service valid "GET" requests and return true from the Start function, along with a proper HTTP response written to the socket. You need to engineer tests that exercises your server's functionality. Both various valid and invalid requests must be tested. This is where the majority

of time is expected to be spent on this assignment and where you will have to figure out how to approach testing of your server.

- You do NOT have to worry about timeouts for this assignment. Accept any and all valid HTTP requests at this point regardless of how slow they are.
- You do NOT have to worry about any HTTP methods other than GET.
- You do NOT have to support any HTTP versions other than 1.1.
- You DO have to reject requests that use any version string other than “HTTP/1.1”, so you still need to check the version string, but you just don’t have to support the request if it’s not HTTP/1.1.
- You DO have to worry about the time at which you terminate an invalid request. At the moment you determine a request to be invalid, do not perform any further reads from the socket. Terminate it at that point. A few examples follow:
    - If your first read from the socket that got 10 bytes and the first 3 were “GEO” you’d terminate the socket connection right then and there and return false. You would not do any further reads from the socket.
    - If you’ve read up to the first line break (\r\n) and have a valid method with a space after it, a URI string with a space after it, but “HTTP/2” following that, then you’d terminate the request and return false. You would not do any further reads from the socket.
- You do NOT have to worry about the body of the HTTP request at this point, since it can really be anything.
- You DO have to worry about knowing when to stop reading a valid request as well as an invalid one, since there is no length value associated with the request. Read up to the double line break (which is the point at which the request body starts) for valid requests and then write the response.
- You DO have to dispose the NetworkStream after processing a valid or invalid response. In the case of the latter, you obviously have to send the response first. Make sure the NetworkStream is disposed before the Start function returns.