PROJECT :: HTTP Server

HTTP Web Server

You will implement this project in three parts with the last part being optional and extra credit.

- Part 1 : A basic web server that supports GET requests
  - Completion of a fully functional and well coded and documented Part 1 will result in a C grade on this assignment
- Part 2 : A more advanced web server that supports POST requests through a CGI mechanism
  - Completion of a fully functional and well coded and documented Part 2 will result in a A grade on this assignment
- Part 3 : You will add security through HTTPS
    - Note that you can add part three to just Part 1, or Part 1 and Part 2.
    - Extra credit buffers all of your grade so that it can provide a buffer for exam scores.

Scaffold of httpserve.h

You are required to use this httpserve.h. Do not modify it because we will not use your httpserve.h. Include anything else that you need inside hour httpserve.c

```
#ifndef HTTPSERVE_H
#define HTTPSERVE_H

#include <stdio.h>  // For size_t

// Server configuration constants
#define SERVER_PORT 8080
#define BUFFER_SIZE 16384

// Function prototypes for server operations

// Start the server on a specified port
void start_server(int port);

// Create a TCP socket and configure it
int create_socket(int port);

// Handle incoming connections on the server socket
void handle_connections(int server_sock);

// Process incoming HTTP requests
void process_request(int client_sock);

// Handle GET requests
void handle_get_request(int client_sock, const char* path);

// Handle HEAD requests
void handle_head_request(int client_sock, const char* path);

// Handle POST requests
void handle_post_request(int client_sock, const char* path);
```

```
// Send an HTTP response to the client
void send_response(int client_sock, const char *header, const char *content_type, const char *body,
int body_length);

// Determine the MIME type based on the file extension
const char* get_mime_type(const char *filename);

#endif // HTTPSERVE_H
```

Scaffold of httpserve.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "httpserve.h"

int main(int argc, char *argv[]) {
    // TODO: Parse command line arguments to override default port if necessary
    start_server(SERVER_PORT);
    return 0;
}

void start_server(int port) {
    // TODO: Implement server initialization:
    // 1. Create a socket.
    // 2. Bind the socket to a port.
    // 3. Listen on the socket.
    // 4. Enter a loop to accept and handle connections.
}

int create_socket(int port) {
    // TODO: Create and configure a TCP socket
    // Return the socket descriptor
}

void handle_connections(int server_sock) {
    // TODO: Implement connection handling:
    // 1. Accept a new connection.
    // 2. Handle the request in a separate function.
    // 3. Close the connection.
}

void process_request(int client_sock) {
    // TODO: Implement request processing:
    // 1. Read the request from the client.
    // 2. Parse the HTTP method, path, and protocol version.
    // 3. Depending on the method, call handle_get_request, handle_head_request, or handle_post_requ
est.
}

void handle_get_request(int client_sock, const char* path) {
    // TODO: Handle GET request:
    // 1. Map the path to a file.
    // 2. Check if the file exists.
    // 3. Read the file and send it with an appropriate response header.
}

void handle_head_request(int client_sock, const char* path) {
    // TODO: Handle HEAD request similarly to GET but without sending the file content.
}

void handle_post_request(int client_sock, const char* path) {
    // TODO: Handle POST request:
```

```
    // 1. If the path is to a CGI script, execute the script and send the output as the response.
    // 2. If not, send a 404 Not Found response.
}

void send_response(int client_sock, const char *header, const char *content_type, const char *body,
int body_length) {
    // TODO: Compile and send a full HTTP response.
    // Include the header, content type, body, and any other necessary HTTP headers.
}

const char* get_mime_type(const char *filename) {
    // TODO: Return the MIME type based on the file extension.
}
```

Part 1 :: Processing GET/HEAD Requests

Objective:

Develop a basic HTTP web server in C that handles GET requests for various content types. The server should be capable of serving a range of files including HTML, CSS, JavaScript, and image files from a predefined directory structure, and manage simple HTTP cookies for basic session handling.

Requirements:

1. **Socket Programming**: Utilize TCP sockets for network communication.
2. **HTTP Handling**: Parse HTTP GET requests and generate corresponding HTTP responses.
3. **Content Serving**: Serve files from the server's local file system based on the request URL, handling different MIME types.
4. **Logging**: Implement basic logging to track requests and server actions.
5. **Compilation Requirement**: The code should compile with a single command:
   bash

1. `gcc httpserve.c -o httpserve`

Expected Functionality:

- **Listen on a configurable port number**.
- **Support serving multiple file types**, handling a variety of MIME types.
- **Handle common HTTP responses** including 404 (Not Found) and 500 (Internal Server Error).
- **Include a default** `index.html` that is served when the root directory is accessed.

MIME Types to Handle:

- **HTML**: `text/html`
- **CSS**: `text/css`
- **JavaScript**: `application/javascript`
- **PNG Images**: `image/png`
- **JPEG Images**: `image/jpeg`
- **GIF Images**: `image/gif`

- **Plain Text**: `text/plain`

Directory Structure:

Traditional web servers typically organize files into structured directories such as `www/` for the root web content, with subdirectories like `css/` for stylesheets, `js/` for JavaScript files, and `images/` for media. This organization enhances security by enabling specific access controls and security rules for different types of content, improving manageability and reducing the risk of serving sensitive files unintentionally. Performance is also optimized by facilitating more effective caching policies, where static content such as scripts and styles can be cached differently from dynamic content, speeding up load times and reducing server load. This structure simplifies maintenance and automated deployment processes, making it easier to manage large-scale websites and ensuring that web applications remain both secure and efficient.

In this project we will keep things simple. All files will be stored directly under the `www/` directory without subdirectories for different file types. This simplification will help us focus on the essential aspects of web server implementation.

Security Measures: Ensure all file paths are sanitized to prevent directory traversal attacks. Specifically, paths containing `../` or `//` should be rejected or normalized.

Deliverables:

- **Source Code**: Single C source file.
- **Documentation**:
  - Setup and execution instructions.
  - Code architecture and flow description.
- **Test Cases**:
  - Scripts or detailed plan to verify server functionality across different file types, error handling, and cookie management.

Relevant HTTP Responses:

- **200 OK**: Successfully processed the request.
- **404 Not Found**: Requested resource is not available.
- **500 Internal Server Error**: General server error (e.g., issues reading a file).
- **400 Bad Request**: Handle malformed requests gracefully.
- **415 Unsupported Media Type**: If a request is made for a file type not supported by the server.

Part 2: POST Requests, CGI Scripts, and Cookies

Objective:

Develop and extend the HTTP web server to handle POST requests, execute server-side scripts via CGI, and manage user sessions through cookies. This part introduces server-side dynamics, providing foundational knowledge on handling form submissions, executing server-side logic, and maintaining state with HTTP cookies.

Requirements:

1. **POST Request Handling:**
   - Implement support for HTTP POST requests.
   - Ensure the server can read data sent from client-side forms.
   - Handle different content types submitted through forms, particularly `application/x-www-form-urlencoded`.
2. **CGI Script Execution:**
   - Set up a `cgi-bin` directory where executable scripts are stored.
   - Implement functionality for the server to execute scripts from this directory when specific routes are requested.
   - Pass POST data to these scripts via standard input (stdin) and environment variables.
   - Use environment variables such as `REQUEST_METHOD`, `CONTENT_LENGTH`, and `QUERY_STRING` to communicate request details to the scripts.
3. **Cookie Management:**
   - Implement basic cookie handling capabilities to manage sessions.
   - Set cookies in HTTP responses using the `Set-Cookie` header.
   - Use cookies to maintain session state, differentiating between new and returning users.
4. **Security and Error Handling:**
   - Properly sanitize input to prevent common security vulnerabilities like command injection in CGI execution.
     - For this assignment this task is limited to disallowing traversing outside of the required directory structure.
   - Handle errors gracefully, providing meaningful HTTP status codes and messages for various error conditions.

Directory Structure:

- `www/`: Root directory for serving static files.
- `cgi-bin/`: Directory for executable scripts that can be invoked by the server.

Deliverables:

- Source Code: Update the C source file to include handling for POST requests and CGI execution. Include the c source for your two required cgi-scripts.

- Documentation:
  - Updated setup and execution instructions.
  - Explanation of how POST data is handled and passed to CGI scripts.
  - Description of cookie handling mechanisms.
- Test Cases:
  - Scripts or detailed plans to verify correct handling of POST requests.
  - Testing interactions with CGI scripts.
  - Cookie management testing to ensure correct session handling.

Expected Learning Outcomes:

- Understand and implement basic mechanisms for handling HTTP POST requests in a web server context.
- Gain practical experience with the Common Gateway Interface (CGI) as a method for including executable content on web servers.
- Learn about HTTP cookies and session management, including how to set and parse cookies and use them to maintain state in a stateless protocol like HTTP.

Overview

Let's get this out of the way, this is a more challenging project than just processing the GET requests. There are multiple sections below that each discuss a different aspect of this project. I've tried to reduce repetition and emphasize clarity as much as possible, however, there is a lot to absorb.

Overview of CGI-BIN

Brief History: The Common Gateway Interface (CGI) is a standard protocol for interfacing external applications with information servers, such as HTTP or web servers. Developed in the early 1990s, CGI is one of the oldest methods for generating dynamic web content. It allows for server-side scripts or programs to be executed in response to requests generated by client-side applications, typically web browsers.

How CGI Works: CGI specifies a method for the web server to pass information to and from an application. This is usually done via environment variables and standard I/O streams. When a request that should be handled by a CGI program is received by the web server:

1. **Environment Setup:** The server sets up the environment for the CGI script, populating environment variables with data such as the request method (GET or POST), content type, query string, and other relevant information about the request.
2. **Execution:** The server executes the CGI script as a separate process.
3. **Data Passing:** For requests like POST, the server passes the request body to the CGI script through standard input (stdin).
4. **Output Handling:** The CGI script processes the request, performs any necessary operations (e.g., database queries, calculations), and writes the response to standard output (stdout). This output

typically includes HTTP headers and the body of the response.
5. **Response Transmission:** The web server captures the output from the CGI script and sends it back to the client.

CGI scripts can be written in almost any programming language, though Perl was traditionally a popular choice due to its strong text manipulation capabilities. Other languages like C, Python, and shell scripts are also used.

Simplifications in This Project: In the educational context of this project, several simplifications are made to focus on the educational aspects of implementing a simple web server and understanding CGI:

- **Standard Communication:** The CGI scripts are expected to read from stdin and write to stdout, adhering to the CGI standard, but the setup is simplified.
- **Environment Variables:** Only essential environment variables like `CONTENT_LENGTH` and `REQUEST_METHOD` are utilized to ensure scripts receive the necessary data to process requests.
- **Error Handling:** Simplified error handling mechanisms are in place, focusing more on demonstrating the interaction between the server and CGI scripts rather than robust error management.
- **Security Aspects:** Security is not the focus; you are warned not to deploy such servers publicly as they might contain vulnerabilities typical of CGI implementations, such as shell injection.
- **Redirection and Headers:** You will be instructed below on specific headers such as `Location` for redirection post-script execution, ensuring that you understand HTTP response manipulation.

Handling POST Data:

When receiving POST data, it is essential to differentiate between header and body data. Unlike GET requests, POST requests contain body data, which includes form contents. Usually, POST data should be read from the socket connection after the headers have been parsed. This data, typically from forms, must be made available to any CGI scripts as standard input (`stdin`). This mimics the way traditional web servers pass form data to scripts. However, for this project we want to eliminate some challenges with respect to managing the data coming into the socket.

To manage this effectively in a simple server:

- **Data Reception:** Utilize `recv()` to retrieve all incoming data from the TCP socket, typically including both headers and body. Ensure your TCP buffers are sufficiently large (16KB is recommended) to accommodate the entirety of a typical request in this project.
- **Data Segregation:** Once data is received, separate it into header and body segments. For POST requests, prepare the body data to be accessible to any CGI scripts as standard input (`stdin`).
- **Process Handling:** Due to the limitations of redirecting `stdin` within a single process, employ a forked process approach. This technique involves setting up a buffer as a stream, redirecting this stream to `stdin`, and redirecting `stdout` to the client socket. Thus, CGI scripts read from `stdin` and write directly to `stdout`, which is piped back to the client socket.

Executing CGI Scripts: CGI scripts located in the `cgi-bin` directory should be triggered based on their corresponding URL paths. To execute these scripts:

- **Environment Setup:** Set essential environment variables, such as `CONTENT_LENGTH` (to indicate the amount of data the script should read from stdin) and `REQUEST_METHOD` (although it may seem redundant as all CGI scripts here use POST, it's a good practice to include it).
- **Script Execution:** Use system calls (e.g., `execve`) to run the CGI script, ensuring its output is captured and sent back to the client. This integration simulates how web servers operate and will help you understand server-side scripting dynamics.
- **Redirect After Script Execution:** After successfully executing a CGI script, your server must redirect the client back to the main page (index.html). To achieve this, include the "HTTP/1.1 302 Found" status in your response headers, followed by the "Location: /index.html" header. This setup informs the client's browser to automatically navigate back to the index page, maintaining a seamless user experience.

Managing Cookies: Cookies are critical for session management and maintaining user state:

- **Setting Cookies:** Use the `Set-Cookie` HTTP header in responses where necessary, such as after a successful login, to set a session cookie on the client's browser.
  - The login session cookie is the only required cookie for this assignment.
- **Reading Cookies:** Parse incoming `Cookie` headers to manage session states or other functionalities. The presence of a session cookie typically indicates that a user is logged in.
  - For this assignment, there is no server side parsing of cookie data.
  - You will be provided with the client side code to read the cookie and change the login page.

Note: You it's best to update the post request prototype to make these methods work correctly.

```
void handle_post_request(int client_sock, const char *path, char* headers, char* body)
```

I/O Redirection and Forking Processes

In C programming, particularly when dealing with UNIX-like operating systems, the ability to manipulate file descriptors and process control is crucial for many system-level applications. This includes tasks such as redirecting standard input (stdin) and standard output (stdout) to or from files or memory buffers before forking a child process. Here's a detailed explanation of how to achieve this, focusing on skills relevant to managing a simple server that interacts with CGI scripts.

Redirecting Standard Input and Output

Redirecting stdin and stdout to/from a Memory Buffer

To redirect stdin or stdout to a memory buffer before forking, you can use the `fmemopen()` function available in POSIX systems. This function creates a stream that can read from or write to a user-supplied buffer. This is particularly useful for preparing data that a child process will consume via stdin or capturing its output from stdout.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {
    char buffer[1024] = "Hello, world!";  // Data to feed into stdin of the child
    FILE *stream = fmemopen(buffer, sizeof(buffer), "r");

    int fd = fileno(stream);  // Get the file descriptor of the stream

    // Redirect stdin to read from the memory buffer
    dup2(fd, STDIN_FILENO);
    fclose(stream);  // After dup2, the original file descriptor can be closed

    // Fork a process
    pid_t pid = fork();
    if (pid == 0) {
        // Child process: reads from stdin (which now points to `buffer`)
        char input[1024];
        fgets(input, sizeof(input), stdin);
        printf("Child received: %s\n", input);
        exit(0);
    }

    // Parent process: continues normally
    wait(NULL);  // Wait for the child to exit
    return 0;
}
```

In this example, a buffer is associated with a file stream, which is then used to replace the standard input of the process. When the process is forked, the child inherits this redirected stdin.

Using `dup2()` for Redirection

The `dup2()` system call is essential for redirecting file descriptors. It duplicates one file descriptor, making it an alias for another. This is widely used to redirect standard streams by replacing stdin, stdout, or stderr with a file descriptor obtained from a file or a pipe.

Example:

```
int file_fd = open("output.txt", O_WRONLY | O_CREAT, 0644);
if (file_fd < 0) {
    perror("Failed to open file");
    exit(1);
}

dup2(file_fd, STDOUT_FILENO);  // Redirect stdout to a file
close(file_fd);  // Close the original file descriptor
```

Forking a Process with Redirected Streams

When combining redirection with process forking, it's crucial to establish the redirection before the fork, so the child process inherits the redirected streams.

Here's a scenario using memory buffer redirection:

```
char *data = "Data for child";
FILE *stream = fmemopen(data, strlen(data), "r+");
dup2(fileno(stream), STDIN_FILENO);
fclose(stream);

pid_t pid = fork();
if (pid == 0) {
    // Child process
    char buf[100];
    read(STDIN_FILENO, buf, sizeof(buf));
    printf("Child process received: %s\n", buf);
    exit(0);
}

// Parent process
wait(NULL);
```

In this setup:

1. **Memory Buffer Setup**: Before forking, a memory buffer is prepared and associated with stdin.
2. **Fork**: The child process inherits this setup and reads from the modified stdin, which now points to a memory buffer.
3. **Execution**: The child processes data from the buffer, mimicking the reception of input through standard input.

This technique is particularly useful for CGI scripts in web servers, where input to the script needs to be controlled precisely, often involving data received over network connections that are stored temporarily in buffers.

Introduction to the Simple Micro-Blog Application

This project involves building a simple micro-blog application using basic web server and CGI programming techniques. The application allows users to view blog posts and, if logged in as the administrator, submit new blog entries.

Application Features:

1. **User Authentication:**
    - Users can log in using predefined credentials. Upon successful login, a session is created which allows the user to submit blog posts.
2. **Viewing Blog Posts:**
    - Visitors to the site, whether logged in or not, can view all the blog posts displayed on the main page (`index.html`).

3. **Submitting Blog Posts:**
    - Logged-in users can submit new blog posts through a form provided in `blogentry.html`. Posts are added to `posts.html` which serves as a simple database.

Directory Structure and Required Files:

- **www/**: The root directory for your web server files.
    - **index.html (https://csus.instructure.com/courses/116725/files/20995260?wrap=1)** : The main page of the blog where posts are displayed and where users can log in or log out. This file is provided to you.
    - **login.html (https://csus.instructure.com/courses/116725/files/20995261?wrap=1)** : The login page that provides a form for user authentication. This file is provided to you.
    - **blogentry.html: (https://csus.instructure.com/courses/116725/files/20995262?wrap=1)** The page with a form for submitting new blog posts, accessible only to logged-in users. This file is provided to you.
    - **posts.html**: This file will act as a storage for blog posts, each post appended as HTML. You need to create and manage this file through your CGI scripts.
- **cgi-bin/**: This directory contains the executable CGI scripts that handle server-side logic:
    - **login.c**: Manages user authentication.
    - **submitblog.c**: Handles the posting of new blog entries.

Required CGI Scripts:

1. **login.c**: Handles user authentication, sets a session cookie upon successful login, and redirects to `index.html`.
2. **submitblog.c**: Handles the submission of new blog posts, writes them to `posts.html`, and redirects to `index.html`.

Implementation Notes:

- The CGI scripts should manage the creation, reading, and appending of content in `posts.html`, ensuring it integrates seamlessly with `index.html`.
- The application should handle HTTP redirection correctly to ensure smooth user experience and proper flow within the application.
- Since security is not the primary focus, the simple handling of cookies and sessions in this project should be understood as educational and not suitable for a production environment.

CGI Script in C: `testcgi.c`

Putting all of this together can be daunting. To help you through some of the tasks I've provided a complete test cgi-script for you that will give you the pattern of these scripts. You may use blocks of code from this script so long as they are properly cited.

This CGI script, named `testcgi.c`, processes simple POST data received from a client, such as data submitted using the `curl` command. The script expects a single word as input through the POST

method. If the input is provided, it embeds the word into an HTML response. If no input is given, it outputs a message indicating that no word was provided. The script handles and returns proper HTTP headers, including handling different HTTP response statuses based on the input.

Features of `testcgi.c`:

1. **Content Reading:** It reads content length from the environment variable `CONTENT_LENGTH` and then reads the input data from `stdin`.
2. **HTTP Header Response:** It provides appropriate HTTP headers before sending the content, ensuring the response is correctly formatted for the client's web browser or tool like `curl`.
3. **Error Handling:** It responds with a `400 Bad Request` status if `CONTENT_LENGTH` is not set, indicating a client error in the HTTP request setup.

The following updated program always sends a full header. Now, the old program should have responded, however, this responds correctly.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {

    // Retrieve the content length from the environment variable
    char *len_str = getenv("CONTENT_LENGTH");
    if (!len_str) {
        printf("HTTP/1.1 400 Bad Request\n\n");
        printf("<html><body>Error: CONTENT_LENGTH not set.</body></html>");
        return 1;
    }

    int len = atoi(len_str);
    if (len == 0) {
        printf("HTTP/1.1 200 OK\n\n");
        printf("<html><body>No word provided.</body></html>");
        return 0;
    }

    // Allocate memory for the input data and read it from stdin
    char *data = (char *)malloc(len + 1);
    if (!data) {
        printf("HTTP/1.1 500 Internal Server Error\n\n");
        printf("<html><body>Error: Failed to allocate memory.</body></html>");
        return 1;
    }

    fread(data, 1, len, stdin);
    data[len] = '\0'; // Null-terminate the input

    // Output the HTTP response
    printf("HTTP/1.1 200 OK\n\n");
    printf("<html><body>You provided: %s</body></html>", data);
    free(data);
    return 0;
}
```

Testing `testcgi.c` Using Curl Commands:

1. **Testing with a Provided Word:**
   `curl -X POST -d "Hello" http://localhost:8080/cgi-bin/testcgi`

   This command uses curl to send "Hello" to the script. The response should be an HTML page displaying "You provided: Hello".

2. **Testing Without Any Data:**
   `curl -X POST http://localhost:8080/cgi-bin/testcgi.c`

   This command sends an empty POST request. The script should respond with "No word provided."

These `curl` commands help verify that the script handles both the presence and absence of input correctly, providing clear and appropriate responses for both situations. The script is designed to be simple and educational, demonstrating basic CGI functionality with proper HTTP response handling.

Here is a sample run of this script. The echo command on the tail end isn't required, it just prints a newline on the console to keep things neat and tidy.

```
testB/cgi-bin$ gcc testcgi.c -o testcgi
testB/cgi-bin$ curl -X POST -d "Howdy" http://localhost:8080/cgi-bin/testcgi; echo
<html><body>You provided: Howdy</body></html>
testB/cgi-bin$ curl -X POST http://localhost:8080/cgi-bin/testcgi; echo
<html><body>No word provided.</body></html>
testB/cgi-bin$
```

Here is the output of the server console for the above operations. Most of what you see is intentional debug/log code.

```
Accepted new connection
LOG: Full HTTP request:
------------------------
POST /cgi-bin/testcgi HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.81.0
Accept: */*
Content-Length: 5
Content-Type: application/x-www-form-urlencoded

Howdy
------------------------

LOG: HEADERS FOLLOW:
POST /cgi-bin/testcgi HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.81.0
Accept: */*
Content-Length: 5
Content-Type: application/x-www-form-urlencoded
--------------------------------

LOG: BODY FOLLOWS:
Howdy
--------------------------------

LOG: PATH FOLLOWS:
/cgi-bin/testcgi
--------------------------------
Closed connection
Accepted new connection
LOG: Full HTTP request:
------------------------
POST /cgi-bin/testcgi HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.81.0
Accept: */*


------------------------

LOG: HEADERS FOLLOW:
POST /cgi-bin/testcgi HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.81.0
Accept: */*
--------------------------------

LOG: BODY FOLLOWS:
```

```
------------------------------

LOG: PATH FOLLOWS:
/cgi-bin/testcgi
------------------------------
Closed connection
```

Summary of Required CGI Scripts

For this project, you will write two CGI scripts: `login.c` and `submitblog.c`. These scripts will be integral to the functioning of your simple blogging platform, handling user authentication and blog post submissions respectively.

1. **login.c**

Functionality:

- This script will handle user authentication.
- It will check if the username and password provided match the expected values (for simplicity, you can hard-code the credentials).
- Upon successful login, it should set a session cookie to manage the login state.
- It should redirect the user back to the main page (`index.html`) after login attempts.

Scaffold:

- Read the environment variable `CONTENT_LENGTH` to determine the length of the POST data.
- Read the POST data (username and password) from stdin.
- Parse the username and password from the POST data.
- Validate the credentials.
- If the login is successful, generate a session ID, set a cookie ~~with HttpOnly flag~~ (you can use a simple function to generate a random session ID).
- Use HTTP status 302 to redirect the user to `index.html`.
- In the case of a failed login attempt, display an error message but do not redirect.

Step-by-Step Instructions:

- Include necessary headers and libraries.
- Start main function.
- Read and validate `CONTENT_LENGTH`.
- Allocate memory and read the POST data.
- Parse the data to extract username and password.
- Validate the credentials against expected values.
- On success, set the session cookie and redirect.
- On failure, output an error message.

- Free any allocated resources.
- Return from the main function with an appropriate status code.

Scaffold `login.c` :

Here's some code to get you started. Note that the error response is not providing a full HTTP response. You will want to edit this so that it is more like the sample script given above.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    printf("Content-type: text/html\n\n");

    // Retrieve the content length
    char *contentLength = getenv("CONTENT_LENGTH");
    if (!contentLength) {
        printf("<html><body>Error: CONTENT_LENGTH not set.</body></html>");
        return 1;
    }

    // Process input data, authenticate user, set cookie, and redirect
    // ...
}
```

2. **submitblog.c**

Functionality:

- This script handles the submission of new blog posts.
- It reads the title and content of a blog post submitted via a POST request.
- The script writes the blog post data to a file (`posts.html`) which acts as a simple database.
- After submitting the post, it redirects the user back to `index.html`.

Scaffold:

- Read the environment variable `CONTENT_LENGTH` for the length of the POST data.
- Read the POST data from stdin, which includes the title and content of the post.
- Open the `posts.html` file and append the new post data formatted as HTML.
- Redirect to `index.html` using HTTP status 302.

Step-by-Step Instructions:

- Include necessary headers and libraries.
- Define the main function.
- Retrieve and validate `CONTENT_LENGTH`.
- Allocate memory for and read the POST data.
- Parse the data to extract the title and content.
- Open `posts.html` and write the formatted post data.
- Redirect the user to `index.html` after writing the post.

- Ensure proper error handling for file operations.
- Free any allocated resources.
- Return from the main function.

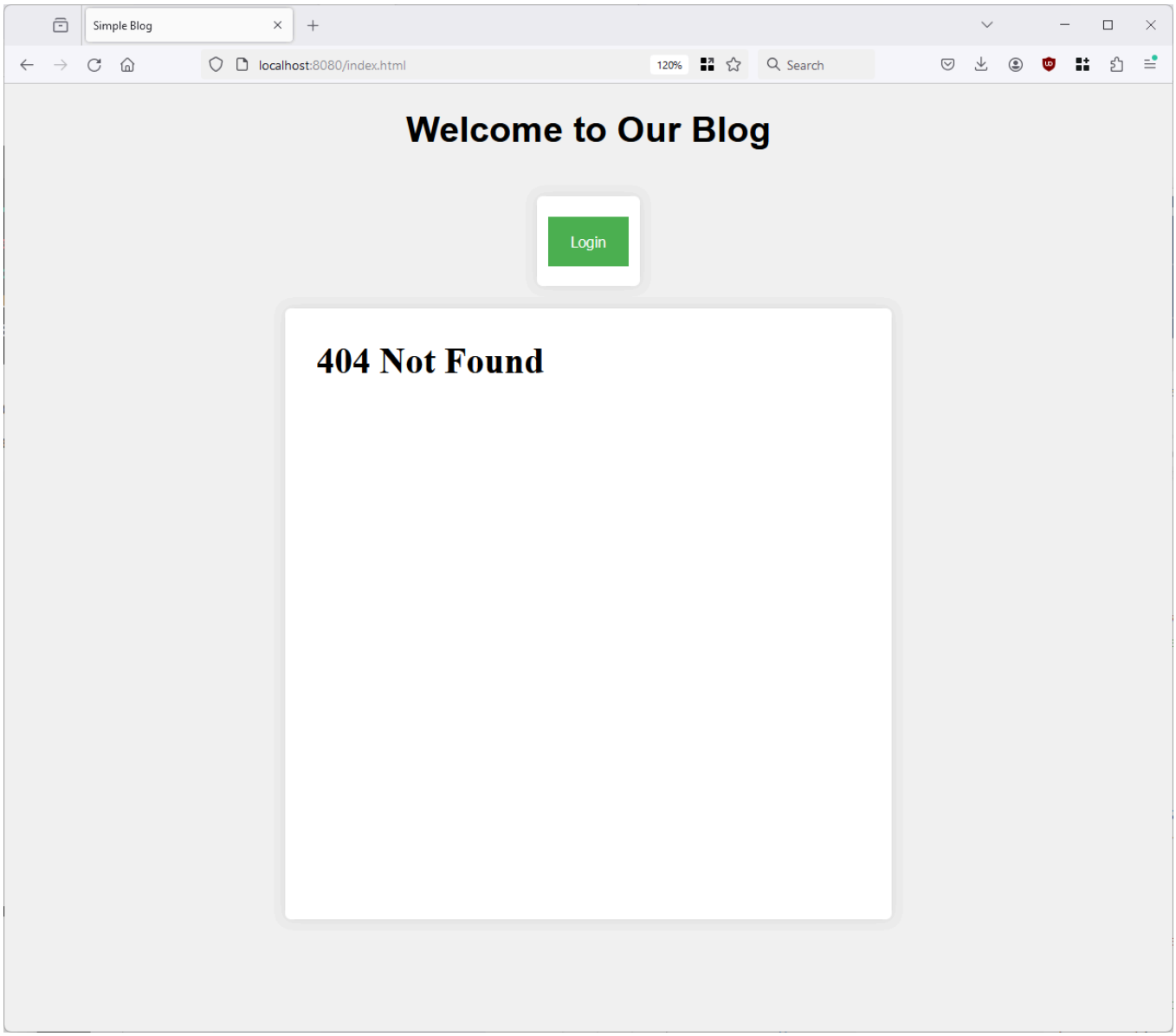
Scaffold `submitblog.c` :

Note that, as above, the error response is not providing a full HTTP response. You will want to edit this so that it is more like the sample script given above.

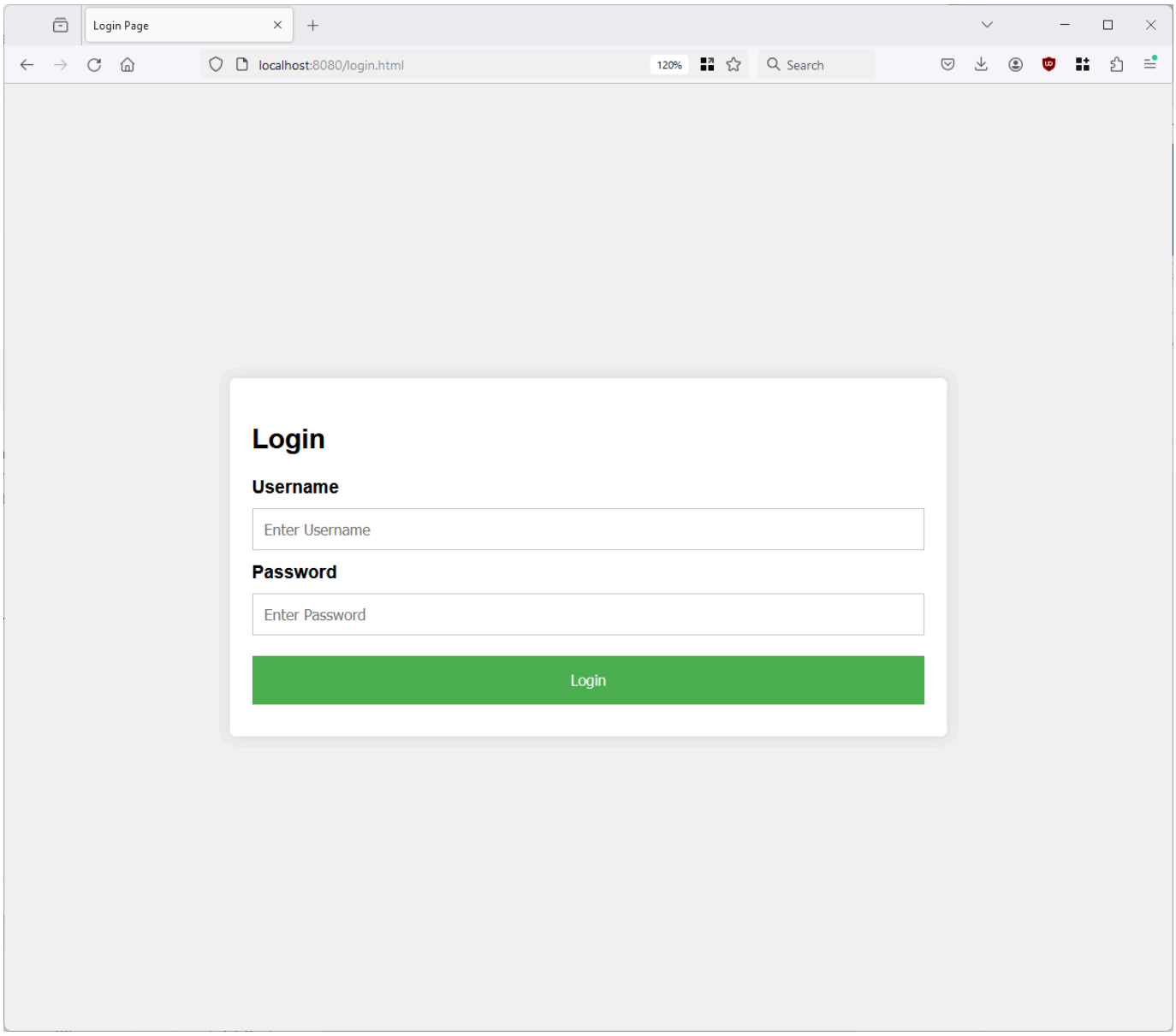
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    printf("Content-type: text/html\n\n");

    // Retrieve the content length and read data
    char *contentLength = getenv("CONTENT_LENGTH");
    if (!contentLength) {
        printf("<html><body>Error: CONTENT_LENGTH not set.</body></html>");
        return 1;
    }

    // Open file, write post, and redirect
    // ...
}
```

Microblog Application Screenshots and Flow

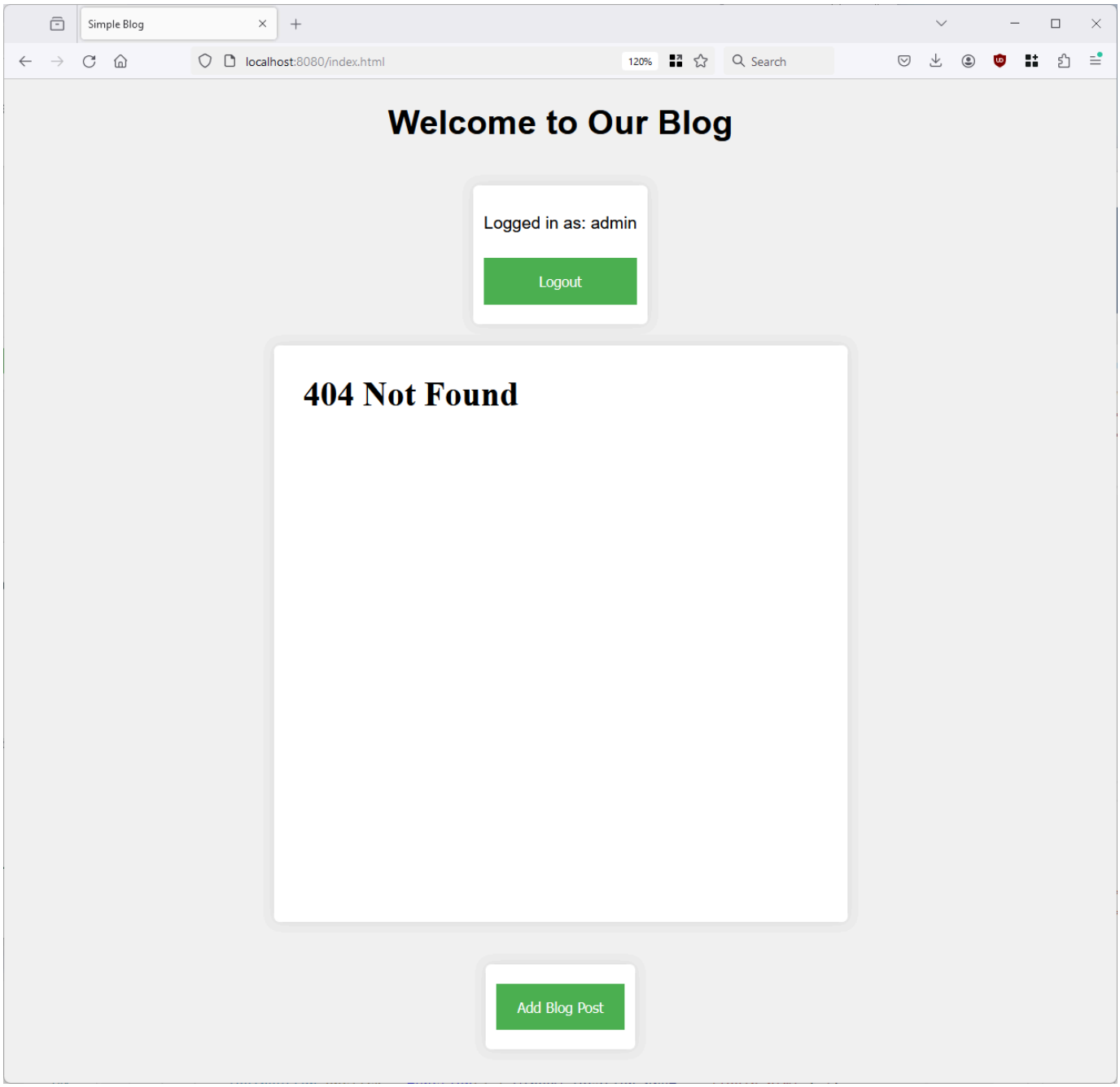
To make sure that you have a good idea of how the microblog works, here are screenshots that demonstrate all major states. When you first load the application and before you have any posts you should see the following. Note, that you can simply touch `posts.html` in `www` to avoid the 404 error. I'm not going to do that to save space in this assignment.



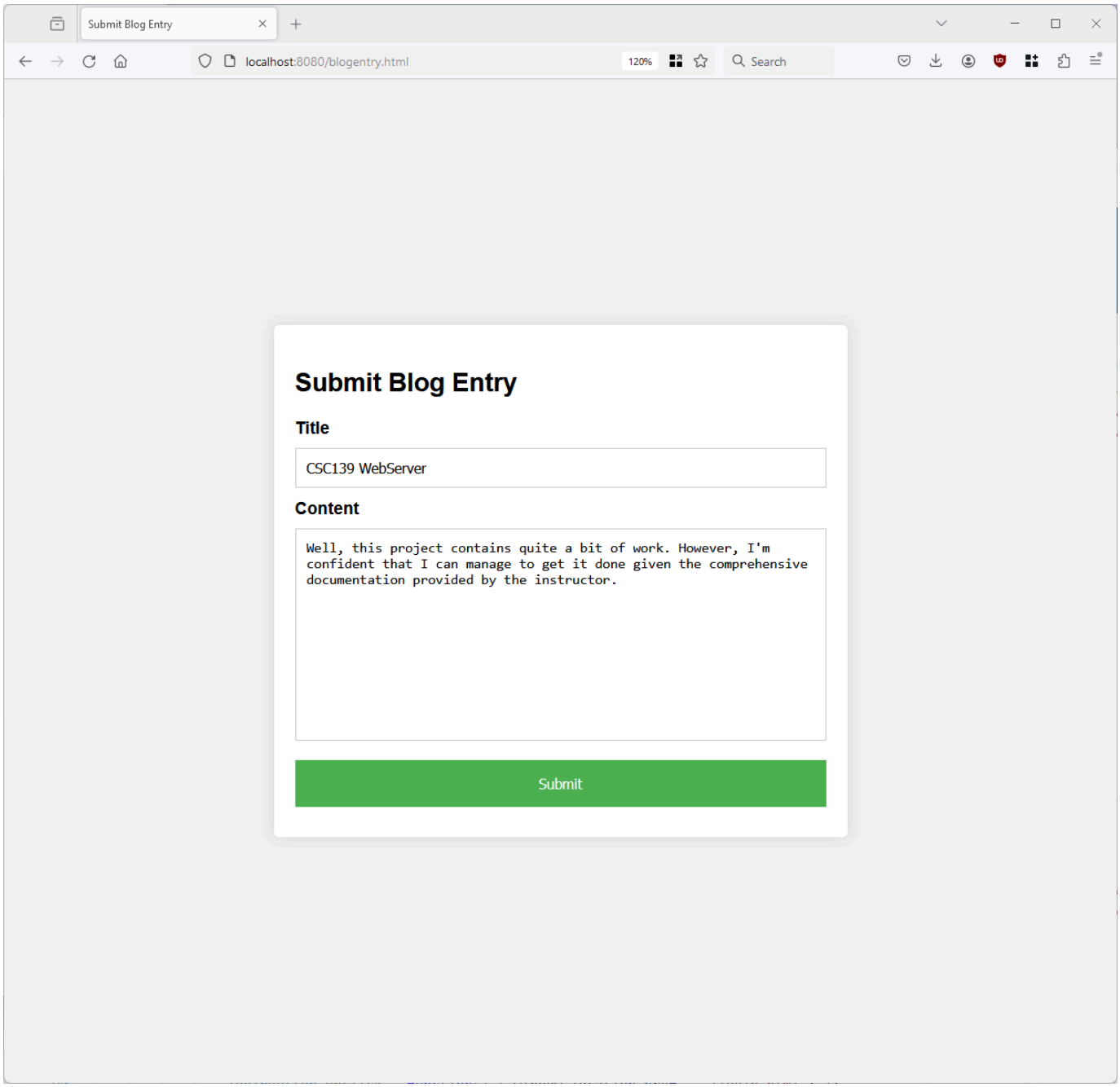
Clicking on the login button will take you to the login screen. Note that the username and password are hard coded into the server. In practice, of course, you would never do this. Our purpose here is to understand the basic flow, not create a secure modern web server. Note also that because we are not using https, that the username and password are transmitted over the network in plaintext.



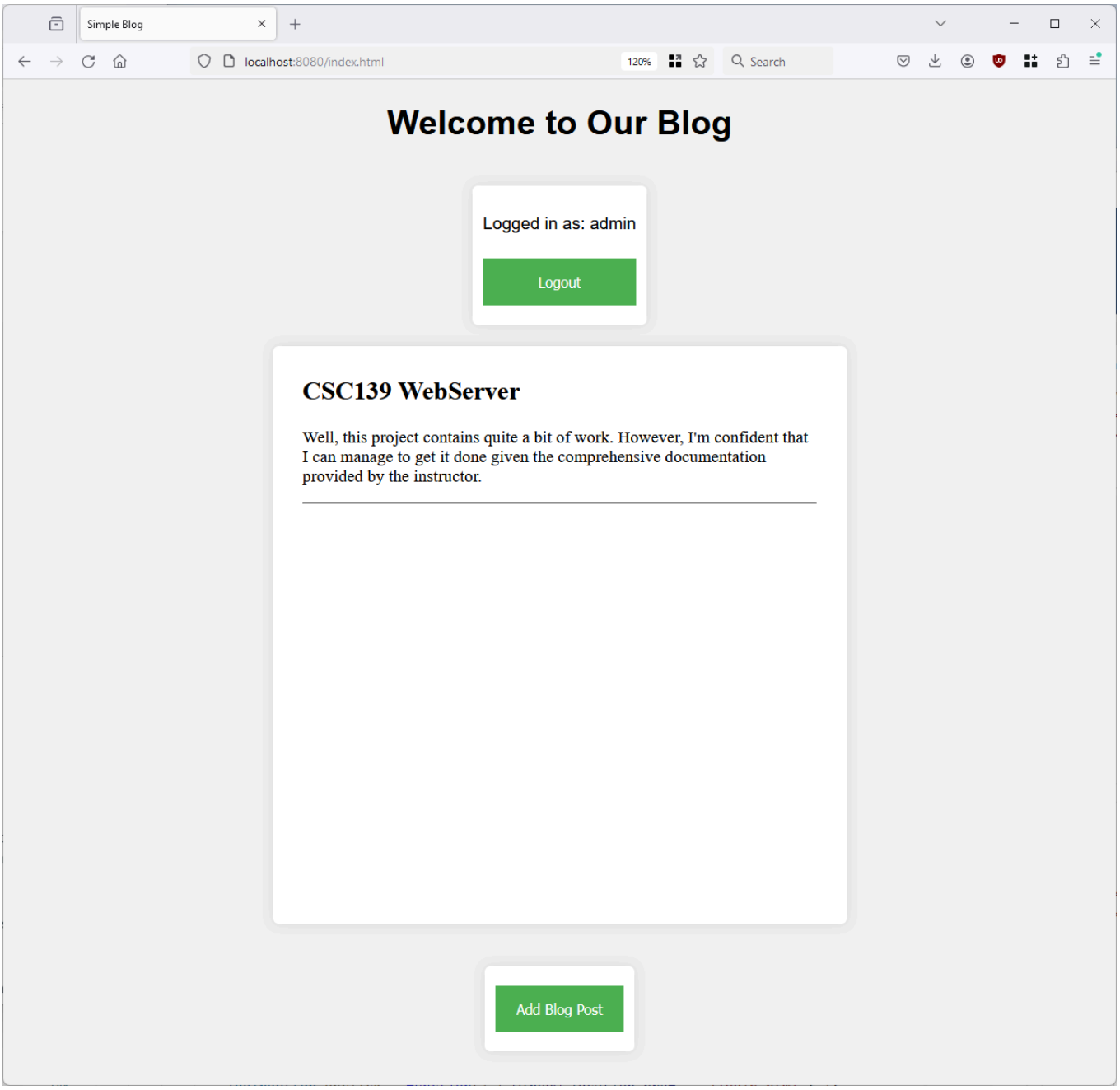
Typing the username (admin) and password (password123) will take you back to the login page. If you specified an incorrect username and password, it should look the same as above, otherwise, you should see the following.



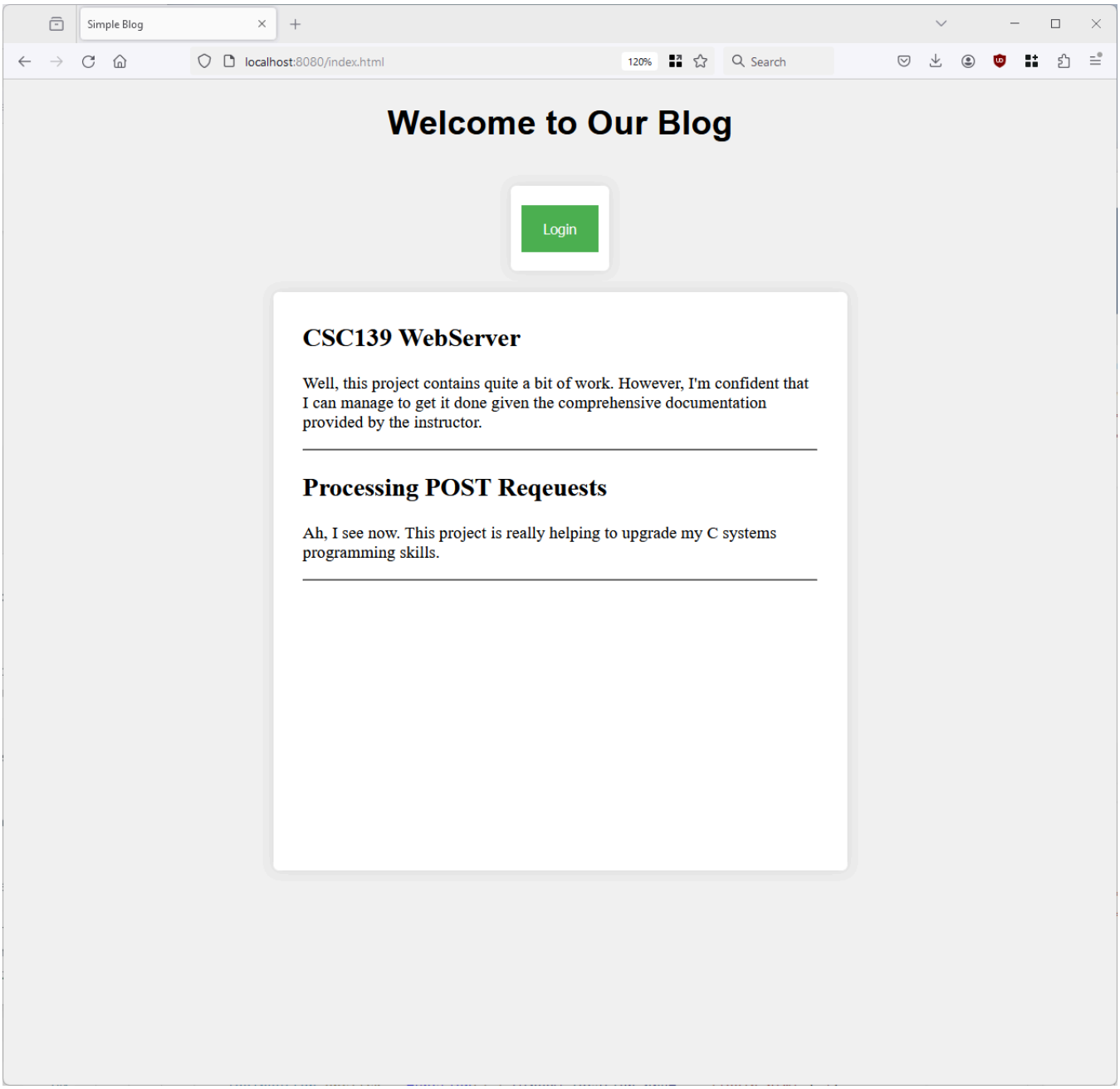
Notice now that you have successfully logged in. In this simple application we are only relying on JavaScript to read the session cookie. In a more serious application we would rely on server side validation. At this point you are able to add a blog post. Clicking on the blog post should take you to the blog post submission page. Of course it will appear empty, but I've gone ahead and typed in some text before taking a screenshot.



Clicking the submit button will submit the post and take you back to the index page. Your site should now look like this.



Now, you can log out. I will add a second post before I do using the same procedure as above.



Ok, that should give you a brief overview of what the provided code should look like. Don't hesitate to ask any questions.

Part 3 :: HTTPS

In addition to adapting your server to handle HTTPS, you will also need to generate a certificate and import the certificate into your local browser(s). Please keep in mind that this project is not suitable for deployment on the world wide web.

1. **Generate a Self-Signed Certificate**:
    - Use OpenSSL or a similar tool to generate a self-signed certificate. This involves creating a private key and using it to generate a certificate.
2. **Test SSL Configuration**:

- Restart your web server and test the SSL configuration to ensure everything is set up correctly.

3. **Import Certificate into Web Browser**:
   - Since this is a self-signed certificate, it won't be trusted by default in web browsers. You'll need to manually import the certificate into the trusted root certificate store of each browser that will access your server.
     - Navigate to the browser's settings or preferences.
     - Find the certificate management section.
     - Import the self-signed certificate into the trusted root certificate store.

Note: More detail will be posted on this part in the first week of the assignment.

Submission

httpserve.c : Make sure that it's well documented!

Release History

Just a friendly reminder: Keeping up with the latest version of assignments is key! We might update them now and then to iron out any bugs or to make things clearer. Whenever there's an update, we'll let you know through a Canvas announcement, so please make sure you're set up to receive those notifications. Embrace the dynamic nature of programming, where adapting to evolving requirements is part of the adventure. If you spot any bugs in the assignments, feel free to send me a bug report. It's a great way to help out and contribute to our course community!

V006 : Removed HTTPS because of time limitations. If you implemented this, then submit httpserves.c and I will take a look.
V005 : Removed the http_only requirement from the session cookie in login.c.
V004 : Updated testcgi.c and included test run output of script
V003 : Updated POST request function prototype
V002 : Updated POST handling details.
V001 : Initial release, expect a few bugs.