Documentation

Compiling and Executing

The server can be compiled using by running
`$ make all`
inside the the `src` folder of the project root. It may be necessary to alter the CC variable inside the `Makefile` to your preferred compiler, but this is set to clang by default. This will place an executable inside the project root. Since the server makes use of SSL, it is a requirement that openSSL is installed on the machine. For the server to be able to perform SSL, place your `key.pem` and `cert.pem` files inside the project root directory. You can create a self-signed certificate and a private key by executing
`$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365`
Please note that, due to the certificate being self-signed, the browser will suggest that the website is unsecure. This can be bypassed on Firefox by clicking “Advanced” and then “Add exception” when attempting to navigate to a page. It is also possible to get a signed certificate from a trusted Certificate Authority (CA) to avoid this warning, but for testing purposes a self-signed certificate is suffice.
Then, the server can be executed from the project root using
`$ ./server -p <port> -d <directory_path_to_web_root>`
When connecting to the server, e.g. on localhost port 1234, it is necessary to use HTTPS, rather than just HTTP due to the SSL protocol. Therefore, an attempt to load a page would be `https://localhost:1234`

Code Walkthrough

`main.c`

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

In the main method, the command line arguments are processed. The -p and -d flags are processed in a manner which allows them to be entered in either order. Following this, the SSL library is then initialised. A call to get_listen_socket is then made to receive the listen socket for the desired port, ready to be used in a call to handle_listen_socket which takes as arguments the listen socket and the web root directory. When handle_listen_socket returns, the server is ready to close, so the SSL cleanup methods are called before returning 0.

`get_listen_socket.c`

```
int get_listen_socket(int port)
```

This method is responsible for creating the socket on the desired port, binding the socket to all interfaces via a call to `bind`, and then finally setting the socket to listen via a call to `listen`. Finally, it returns the socket file descriptor ready for incoming connections to be accepted and processed.

handle_listen_socket.c

`int handle_listen_socket(int socket_fd, const char *root_dir)`

Creates an SSL context via calls to `create_context` and `configure_context`, and handles each connection in a new thread. It makes use of a struct called `thread_control_block_t` which enables data to be passed through to the thread. The data struct contains the client socket, the client address, the address size, the server root and the SSL context which is created at the start of this method. The method is used in the `pthread_create` call is `static void *setup_client_thread(void *data)`.

`static void *setup_client_thread(void *data)`

This method handles the preparation of the SSL socket, which is essentially a wrapper around the client socket received from the `accept` method. The method then proceeds to call `service_client_socket` to handle request and send a response. Once this method returns, `setup_client_thread` proceeds to close the client socket and free any data.

`SSL_CTX *create_context()`

Creates and returns the SSL context with the desired method.

`void configure_context(SSL_CTX* ctx)`

Configures the SSL context to use the `key.pem` and `cert.pem` files which should be located in the same directory as the executable.

make_printable_address.c

`char *make_printable_address(const struct sockaddr_in6 *const addr, const socklen_t addr_len, char *const buffer, const size_t buffer_size)`

Turns an address from the sockaddr_in6 struct into a string which can be printed in the format “<address> port <port number>”.

service_client_socket.c

`int service_client_socket(SSL *ssl, const char *const printable_address, const char *root_dir)`

This method makes a call to `SSL_accept`, which handles the SSL protocol to establish a connection with the client. From then on, it is possible to call `SSL_write` and `SSL_read` to read from the SSL socket. After this, a call is made to `read_header_lines` which processes the header part of the request and places the required data into a `http_header` struct. Then, a call is made to `send_http_response` which uses the information from the request to construct a response. Finally, the method carries out any remaining clean up.

`static int process_http_header_line(char *buffer, size_t length, http_header *header)`

Takes the line stored in `char *buffer` from the HTTP request headers and processes it accordingly. It extracts the information and stores it in the relevant field of the `http_header` struct.

`static http_header* new_header()`

Creates the default `http_header` struct on the heap and returns a pointer to it.

`static int read_header_lines(SSL *ssl, http_header *header)`

Reads from the SSL socket, using `SSL_read`, character by character and detects when a CRLF is processed. If a CRLF (`\r\n`) is found, then we have encountered another line of the request header and we should make a call to `process_http_header_line` to handle it appropriately. If two successive CRLFs are encountered, this marks the end of the header and we are done processing.

send_http_response.c

`int send_http_response(SSL* ssl, const char *root_dir, http_header *header)`

The method first makes a call to read_local_file, which reads the local file using `read_local_file`. If read_local_file returns -1 and the status is currently believed to be 200 OK, then the file must not have been found and the status is changed to 404 Not Found. Following this, send_response_header is called to send the header of the response. This method may return a string which is stored in `char *body`. If body is null, the local file is sent, otherwise the `char *body` is sent. Both are sent via call to `send_all`.

`static int read_local_file(const char *root_dir, const char *file, const char **buffer, size_t *size)`

Concatenates `root_dir` and `file` to produce the location of the file to read. The contents of the file are stored in memory and `buffer` then points to this, so it is important that files read are able to fit in memory. The size of the contents is also stored in size, which is useful for sending the data across the SSL socket and for the content-length header.

`static int send_all(SSL *ssl, char *buffer, size_t size)`

Ensures all bytes in the buffer are sent over the SSL socket by wrapping `SSL_send` in a while loop.

`static char *send_response_header(SSL *ssl, htt_header *header, const size_t content_size)`

Determines the appropriate response based upon the `http_header` struct. `content_size` is used for setting the content length in a 200 OK response. If the request should not send back a file read from the server, then the appropriate body is returned to be sent instead, otherwise `NULL` is returned.

http_header.h

`enum request_method { GET, HEAD, UNSUPPORTED };`

Stores the method used in the request.

`http_header`

This is a struct which contains an `enum request_method` to store the method, a separate `char *` resource and version, and an `int` for the status.