





# Webserv: A C++ Webserver

---

Written by @laian (<https://hackmd.io/@laian>), for [this](https://github.com/LaiAnTan/42KL-Webserv) (<https://github.com/LaiAnTan/42KL-Webserv>) project.

## Background Knowledge

---

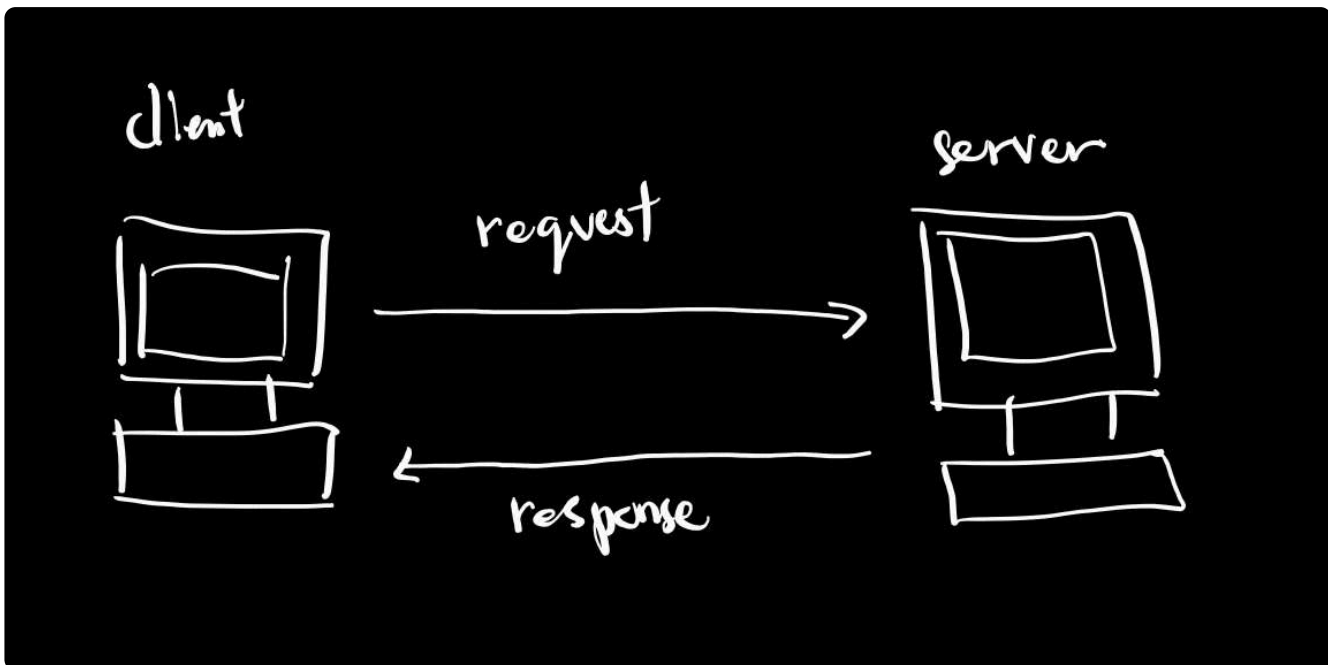
The following is a condensed guide for anyone checking out the project. It contains some amount of background information to enable people to be able to understand the code.

## Basic Concept of A Webserver

---

A webserver is basically a computer (called the *server*) that receives data from other computers (called the *clients*).

It receives requests from clients, processes them, and sends the appropriate response back to the client.



## Hypertext Transfer Protocol

---

Webservers would not be possible without the HTTP protocol.

A protocol refers to the set of rules that govern the communication between two entities.

In this case, the Hypertext Transfer Protocol defines a set of rules that clients and servers have to follow in order to communicate with each other for webpages.

Therefore, to make the webserver work on browsers (etc.), we must program the webserver to handle requests of a certain format, and also respond in the required format.

~~The HTTP Protocol is documented in the Request For Comments (RFC).~~  
(<https://datatracker.ietf.org/doc/html/rfc2616>).

## General Formatting of Requests and Responses

There are two main parts of a request / response, the **header** and the **body**.

The header contains information on the request / response, describing the context information.

The body contains the content of the request / response, which may be empty.

This string of characters must be present before and after the body (if it exists), which signifies its start and end point.

```
\r\n\r\n
```

## Request


The header of the request starts with a **request-line**, which specifies (in order) the request method, the request URI and the HTTP version.

```
GET /index.html HTTP/1.1
```

After the request-line, there may be header fields that describe the context of the request.

The following is an example of a GET request, sent by the browser when a client was trying to access index.html.

```
GET /index.html HTTP/1.1
Host: localhost
Connection: keep-alive
sec-ch-ua: "Chromium";v="116", "Not)A;Brand";v="24", "Opera GX";v="102"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, li
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```



Again, some requests might contain content in the body.

## Request Methods

Each request to the server contains a specified method, which tells the server the specific action to take.

Some of the most important methods include:

- GET: When a client requests a piece of information such as a webpage from the server
- POST: When a client sends a piece of information (files) to the server
- DELETE: When a client requests the server to delete the specified resource.

## Response

The response is very similar to the request, in which it starts with a **response-line**, which specifies (in order) the HTTP version, the status code and the status message.

```
HTTP/1.1 200 OK
```

Similar to requests, the response might have header fields and also a body.

The following is an example of a response to a request.

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/html
Content-Length: 2462

<!DOCTYPE html>
<html>
    <head>
        <style>
            body {
                font-family: Arial, Helvetica, sans-serif;
                background-color: #000000;
            }
        ...
    ...
    ...
</html>
```

## Inner Workings of the Webserver

---

### Sockets and Useful Network Functions

Preamble: All functions described are c/c++ functions, there are probably alternatives in other languages.

The connection between clients and servers are facilitated by sockets, which are the communication link between two processes on a network.

A socket is just a file descriptor, created using the function:

```
int socket(int domain, int type, int protocol);
```

- Domain refers to the communication domain. Domains are listed [here](https://linux.die.net/man/2/socket) (<https://linux.die.net/man/2/socket>). In this case, we use AF\_INET .
- Type refers to the type of socket.
- Protocol refers to a particular protocol to be used with the socket, usually specified as 0.

There are a few types of sockets, but the two main types are:

#### 1. Stream Sockets SOCK\_STREAM

Stream sockets are reliable two-way connected communication streams which use the Transmission Control Protocol (TCP).

These sockets are usually used by HTTP, when speed is not the priority, but data quality is.

## 2. Datagram Sockets `SOCK_DGRAM`

Datagram Sockets are connectionless sockets that use the User Datagram Protocol (UDP).

When data is sent, it may or may not arrive.

Datagram sockets are connectionless because it doesn't require an open connection to send data.

These sockets are commonly used by games, video and audio, where speed is priority.

Once we have a socket descriptor, we need to *bind* it to a port on the computer, with

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

You might notice that there is a `sockaddr` struct required in the parameters.

This is what the simpler version of it looks like (we can cast it later):

```
struct sockaddr_in {
    short int     sin_family;   // Address family, AF_INET
    unsigned short int sin_port; // Port number
    struct in_addr sin_addr;    // Internet address
    unsigned char  sin_zero[8]; // Same size as struct sockaddr
};
```

The struct contains information on the socket address.

This is how to configure the struct:

```
address.sin_family = domain; // domain
address.sin_port = htons(port_number); // port number
address.sin_addr.s_addr = htonl(interface); //ip address
```

The interface is a special value loaded into the struct's ip address to make it auto-fill the address with the current host. In this situation, we use `INADDR_ANY`

You might have noticed some foreign functions, such as `htons` and `htonl`.

```
uint16_t htons(uint16_t hostshort);
uint32_t htonl(uint32_t hostlong);
```

htons means **host** to **network** *short*.

htonl means **host** to **network** *long*.

Before that, we need to talk about byte orders.

Some computers store bytes in reverse order, this storage method is called *Little-Endian*.

Computers that store bytes in the correct order use the storage method *Big-Endian*, also known as *Network Byte Order*.

Computers store bytes in *Host Byte Order*. Unfortunately, every computer uses a different storage method, either Little or Big Endian.

This is why we pass values through a function, in this case `htons` and `htonl`, so that it changes the value from Host Byte Order to Network Byte Order, converting it if needed.

We also might want to configure our socket, and that can be done with `setsockopt`.

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

In the case of a webserver, we would want to let our program reuse ports, that were previously used, so we can do:

```
const int enable = 1;
if (setsockopt(this->sock, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
    std::cerr << "setsockopt(SO_REUSEADDR) failed" << endl;
```

This prevents the "Address already in use" error when trying to rerun a server.

Sockets can be used to connect to a remote host with `connect`, but in this case, we want to listen to incoming connections, and then handle them accordingly.

This can be achieved with `listen`.

```
int listen(int sockfd, int backlog);
```

`backlog` refers the amount of connections that can be put on hold.

This way, other people can use `connect` to connect to our machine that we are `listen` ing on.



Incoming connections will be queued, and can then be accepted with `accept` .

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

We can just use the same `sockaddr_in` struct as before.

Finally, we can read from a socket using `read` / `recv` , and write to a socket using `send` / `write` .

Therefore, the flow of system calls is as follows:

```
socket()  
bind()  
listen()  
accept()  
read() || recv() && send() || write() // accordingly
```

Note: all of these functions can be found in the *manual*, in case further clarifications are needed.

## Multiple Clients

A webserver needs to be able to handle multiple clients, as in a normal situation there would be numerous users trying to access a singular website.

We can achieve this by having an array of socket descriptors.

As long as the server is running, we:

1. poll the whole array
2. check for new connections and handle them
3. handle current connections
  - a. if current socket is POLLIN, read data, set to POLLIN
  - b. if current socket is POLLOUT, send data, set to POLLOUT

## Polling

But what happens when we try to read from a socket without any data going into it? It just sits there, waiting for some data to read.

This is called **blocking**.

Blocking is bad because it can halt the server, causing it to stop functioning until the call goes through.

But how do we know which sockets are able to be read / written to?

This is where polling comes in.

Polling is basically just telling the system to let us know which sockets have the data ready, and we can perform operations on them.

Polling can be done using

```
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

There is an array of pollfd structs required in the parameters, which is defined as follows:

```
struct pollfd {  
    int fd;           // the socket descriptor  
    short events;     // bitmap of events we're interested in  
    short revents;    // when poll() returns, bitmap of events that occurred  
};
```

We set `events` to tell the system call which event we care about:

```
pollfd.events = POLLIN; // tell me when data is ready to receive  
pollfd.events = POLLOUT; // tell me when i can send data without blocking
```

After poll returns, the value of `revents` in each struct in the array is checked for the status of each socket.

## Chunking

What happens when a client sends a huge amount of data through the socket at once?  
Or when the server wants to send a large amount of data to a certain client?

The server will freeze as it is trying to send / read all of that data, and as a result the server will not be able to serve other clients.

There is a solution to this problem, which is to process (read / parse) only specified lengths of data at a time.

This ensures that not one client takes up all of the servers time reading heaps of data.

We can achieve this by defining a `BUFFER_SIZE` , and then only reading a length of buffer size each time.

```
read(socket, buffer, BUFFER_SIZE);
```

Chunking can be applied in places like:

- Large responses to GET requests.
- POST requests with large body.

## CGI

Common Gateway Interface, aka. CGI, is an interface specification for web servers to be able to execute external programs.

Usually, it is used to display dynamic webpages.

## Resources

---

<https://beej.us/guide/bgnet/html/#broadcast-packetshello-world>

(<https://beej.us/guide/bgnet/html/#broadcast-packetshello-world>).

## Documentation

---

This part contains the documentation for the code, which mainly focuses on how to configure a server with the program.

## Server Configuration

---

The config is based on nginx's server configuration format.

Two types of blocks are supported, server and location blocks.

Every server block defined must be in the following format:

```
server {  
    ...  
}
```

Inside a server block, there are also location blocks, defined as follows:

```
location <URI> {  
    ...  
}
```

Location blocks define how the webserver handles requests to specific URI's.

Inside blocks, there exists directives.

Supported Directives:

- listen: Defines the port number to listen to.

```
port <port_number>;  
port 80;
```

- root: Sets the root directory for requests.

```
root <path>;  
root html;
```

- index: Defines files that will be used as an index.

```
index <file 1> <file 2> <file 3> ...;  
index index.html;
```

- error\_page: To define potential error pages.

```
error_page <error_code> <error_page>;  
error_page 404 404.html;
```

- autoindex: Automatically generates a directory listing if an index page isn't specified.

```
autoindex on | off;
```

- `allowed_methods`: Defines the allowed request methods for a specific location. Methods supported are (POST, GET, DELETE).

```
allowed_methods <method 1> <method 2>;  
allowed_methods POST GET DELETE;
```

- `return`: Defines a specified URI / page to return to the client.

```
return <URI>;  
return https://google.com;
```

- `client_max_body-size`: Defines the max size of a client request body.

```
client_max_body_size <size>; # B, MB, KB, GB  
client_max_body_size 1MB;
```