

From http-server.c/h

```
void start_server(void(*handler)(char*, int), int port);
```

Starts an HTTP server on the given port. Runs forever (until Ctrl-C), calling `handler` once per incoming request.

`port`: the port number to listen on (e.g. 2900). Can be 0 to let the OS pick a port.

`handler`: a function that processes one request. It expects two arguments:

- `request (char*)` – the full HTTP request as a string
- `client (int)` – a file descriptor; use `write(client, buf, len)` to send a response back

Why does curl say “Empty reply from server”?

```
1 #include "http-server.h"
2 #include <string.h>
3
4 void handle(char *request, int client) {
5     printf("Received request:\n%s\n", request);
6 }
7
8 int main() { start_server(&handle, 2900); }
```

```
$ gcc string-server1.c http-server.c -o string-server1
$ ./string-server1
Server started on port 2900
Received request:
GET /add?s=hi HTTP/1.1
Host: localhost:2900
User-Agent: curl/8.7.1
Accept: */*

^C
```

```
$ curl "localhost:2900/add?s=hi"
curl: (52) Empty reply from server
```

HTTP Response Format

When the server receives a request, it must write back a *response*, which has a specific format.

```
HTTP/1.1 200 OK
Content-Type: text/plain
...response body...
```

As a C string literal:

```
"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n"
```

Important: HTTP uses `\r\n` (not just `\n`) for line endings.

Sending a response

`write(client, buf, len)` sends `len` bytes from `buf` to whoever made the request. Works the same as writing to a file – `client` is a file descriptor.

You can call `write` multiple times to build up a response:

```
write(client, HTTP_200, strlen(HTTP_200));
write(client, "Hello!", 6);
```

Note: `write` needs a length – it’s not like `printf`. We’ll often use `strlen` to compute it. Unlike C strings, HTTP responses don’t use a null terminator – the length tells the receiver how many bytes we’ve written.

Program 2: Send a response

```
4 char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
5
6 void handle(char *request, int client) {
7     printf("Received request:\n%s\n", request);
8     write(client, HTTP_200, strlen(HTTP_200));
9     write(client, "Hello!", 6);
10 }
11
12 int main() { start_server(&handle, 2900); }
```

```
$ ./string-server2
Server started on port 2900
Received request:
GET /add?s=hi HTTP/1.1
Host: localhost:2900
User-Agent: curl/8.7.1
Accept: */*
```

```
$ curl "localhost:2900/add?s=hi"
Hello!
```

What’s the difference between `printf` and `write` here?

HTTP Request Format

The request string passed to our handler looks like:

```
GET /add?s=hi HTTP/1.1\r\n
Host: localhost:2900\r\n
User-Agent: curl/8.7.1\r\n
```

For our server, we want to extract the path and query string from the first line: `/add?s=hi`

Parsing with `sscanf` and `strstr`

`sscanf` reads formatted data from a string – the inverse of `sprintf`:

```
char path[256];
sscanf(request, "GET %s", path);
// path is now "/add?s=hi" (%s stops matching at the space)
```

`strstr(haystack, needle)` returns a pointer to the first occurrence of `needle` in `haystack`, or `NULL`. We can use it to find the `"?s="` for the query.

```
$ curl "localhost:2900/add?s=hi"
Added: hi
$ curl "localhost:2900/add"
Missing ?s= parameter
$ curl "localhost:2900/unknown"
Unknown path
```

Program 3: Parse the request

```
4 char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
5 char *HTTP_404 = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\n";
6
7 void send_404(int client, char *message) {
8     write(client, HTTP_404, strlen(HTTP_404));
9     write(client, message, strlen(message));
10 }
11
12 void handle(char *request, int client) {
13     char path[256];
14     sscanf(request, "GET %s", path);
15
16     if (strncmp(path, "/add", 4) == 0) {
17         char *query_start = strstr(path, "?s=");
18         if (query_start) {
19
20
21
22
23         } else {
24             send_404(client, "Missing ?s= parameter");
25         }
26     } else {
27         send_404(client, "Unknown path");
28     }
29 }
```

Program 4: Keeping state across requests

Unlike a program that runs and exits, a server runs forever. Data that should persist between requests can't live on the stack – it needs to be in global variables or on the heap.

The copy problem

The `request` buffer in `http-server.c` is reused for every request. If we store a pointer into it, that pointer's data gets overwritten on the next request.

`handle` has the same structure as Program 3, but calls `add_string` and `respond_with_list`:

```
32 void handle(char *request, int client) {
33     char path[256];
34     sscanf(request, "GET %s", path);
35
36     if (strncmp(path, "/add", 4) == 0) {
37         char *query_start = strstr(path, "?s=");
38         if (query_start) {
39             char *string_start = query_start + 3;
40             add_string(string_start);
41             respond_with_list(client);
42         } else {
43             send_404(client, "Missing ?s= parameter");
44         }
45     } else {
46         send_404(client, "Unknown path");
47     }
48 }
```

```
12 #define MAX_STRINGS 100
13
14 char *strings[MAX_STRINGS];
15 int num_strings = 0;
16
17 void add_string(char *s) {
18     if (num_strings >= MAX_STRINGS) { return; }
19     strings[num_strings] = malloc(strlen(s) + 1);
20     strcpy(strings[num_strings], s);
21     num_strings++;
22 }
23
24 void respond_with_list(int client) {
```

```
$ curl "localhost:2900/add?s=hello"
hello
$ curl "localhost:2900/add?s=world"
hello
world
$ curl "localhost:2900/add?s=goodbye"
hello
world
goodbye
```