HTTP Protocol - Theory, Usage, Webserv Context

? What Is HTTP?

Hypertext Transfer Protocol (HTTP) is the core protocol for communication on the Web. It defines how clients (browsers) and servers exchange messages.

HTTP 1.1 (used in Webserv) is stateless but supports:

- Persistent connections (multiple requests/responses per TCP connection)
- Chunked transfer encoding (sending response in pieces)

🔧 Theory / Deep Dive

- HTTP messages are plain text, ASCII-encoded

Request Structure:

1. Request Line: `METHOD PATH VERSION` (e.g., `GET /index.html HTTP/1.1`)
2. Headers: `key: value` pairs (Host, User-Agent, Content-Length, etc.)
3. Empty line
4. Optional body (mainly POST/PUT)

Response Structure:

1. Status Line: `VERSION STATUS_CODE REASON_PHRASE` (e.g., `HTTP/1.1 200 OK`)
2. Headers: Content-Type, Content-Length, Connection, etc.
3. Empty line
4. Body: HTML, JSON, text, binary

🛠 HTTP Methods

- **GET**: Retrieve resource
- **POST**: Send data to server
- **HEAD**: Retrieve only headers
- **DELETE**: Remove resource
- **OPTIONS**: Query server capabilities

🛠 Status Codes

- **1xx**: Informational
- **2xx**: Success (200 OK)
- **3xx**: Redirection
- **4xx**: Client Error (404 Not Found, 405 Method Not Allowed)
- **5xx**: Server Error (500 Internal Server Error)

🛠 Important Headers

- **Host**: Specifies server
- **Content-Length**: Length of body in bytes
- **Content-Type**: MIME type
- **Connection**: keep-alive / close
- **Transfer-Encoding**: chunked

🛠️ Persistent Connections

- Keep connection open for multiple requests/responses
- Improves performance by avoiding TCP handshake for each request

🛠️ Chunked Transfer

- Send response in chunks of unknown size
- Format: `<chunk-size in hex>\r\n<data>\r\n`
- Ends with chunk-size `0\r\n\r\n`

📦 Headers / Imports

```
#include <string>
#include <map>
#include <sstream>
#include <iostream>
```

⚙️ Declaration / Syntax

```
struct HttpRequest {
    std::string method;
    std::string uri;
    std::string version;
    std::map<std::string, std::string> headers;
    std::string body;
};

struct HttpResponse {
    std::string version;
    int status_code;
    std::string reason;
    std::map<std::string, std::string> headers;
    std::string body;
};
```

🔧 Parameters / Details

HTTP Request:

- `method`: GET, POST, DELETE, HEAD
- `uri`: resource path (/index.html)
- `version`: HTTP/1.1
- `headers`: key-value pairs (case-insensitive)

- `body`: optional, used with POST/PUT

HTTP Response:

- `version`: HTTP version
- `status_code`: numeric code (200, 404…)
- `reason`: textual explanation ("OK", "Not Found")
- `headers`: key-value metadata
- `body`: content sent to client

Basic Request Parsing Example

```
std::string raw = "GET /index.html HTTP/1.1\r\nHost: localhost\r\n\r\n"
std::istringstream stream(raw);
HttpRequest req;
stream >> req.method >> req.uri >> req.version;

std::string line;
while (std::getline(stream, line) && line != "\r") {
    size_t sep = line.find(":");
    if (sep != std::string::npos)
        req.headers[line.substr(0, sep)] = line.substr(sep + 2, line.si
}
```

Basic Response Build Example

```
HttpResponse res;
res.version = "HTTP/1.1";
res.status_code = 200;
res.reason = "OK";
res.headers["Content-Type"] = "text/html";
res.body = "<h1>Hello World</h1>";
res.headers["Content-Length"] = std::to_string(res.body.size());

std::ostringstream out;
out << res.version << " " << res.status_code << " " << res.reason << "\
for(auto it = res.headers.begin(); it != res.headers.end(); ++it)
    out << it->first << ": " << it->second << "\r\n";
out << "\r\n" << res.body;

std::string response_str = out.str();
```

Advanced Considerations

- Persistent connections: handle "Connection: keep-alive" vs "close"
- Partial reads: POST body may arrive in multiple TCP packets; buffer until complete
- Chunked encoding: optional for large responses
- Header parsing: case-insensitive keys, trim whitespaces
- Content-Length: must match body size or client may hang
- Multiple clients: combine with select() or poll() for concurrency
- Request validation: reject invalid methods, malformed lines

⚠ Gotchas

- Missing `\r\n` → malformed request/response
- Content-Length mismatch → client hangs
- POST body split across multiple `recv()` calls → loop until complete
- Header names case-insensitive → normalize before lookup
- Connection may close unexpectedly → handle gracefully

 Webserv Context

- Parse request line + headers + optional body
- Store headers in `map<string,string>` for quick access
- Handle multiple clients with `select()` or `poll()`
- **GET**: serve static files from root directory
- **POST**: forward to CGI scripts with CONTENT_LENGTH handling
- Handle errors: 404, 405, 413, 500
- Use chunked encoding for dynamic content
- Correct HTTP formatting is mandatory for norm compliance

 Nerdy Notes

- HTTP is stateless: maintain state via cookies or sessions if needed
- CRLF injections/malformed headers must be sanitized
- Always validate method, path, version
- Large POST: read exactly Content-Length bytes
- Non-blocking `recv()` may require looping to assemble full request
- Implement minimal timeout for persistent connections

 Related Concepts

TCP sockets, non-blocking I/O, select(), poll(), CGI execution, std::vector for client management, String parsing utilities: substr, find, map