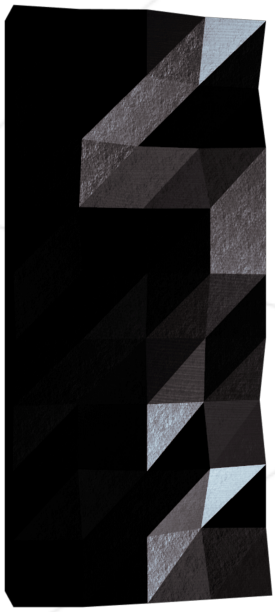

Webserv

This is when you finally understand why a URL starts with HTTP

Summary: This project is here to make you write your HTTP server. You will be able to test it with a real browser. HTTP is one of the most used protocols on the internet. Knowing its arcane will be useful, even if you won't be working on a website.

Contents

Chapter I

Introduction

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.
HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access, for example by a mouse click or by tapping the screen in a web browser.
HTTP was developed to facilitate hypertext and the World Wide Web.
The primary function of a web server is to store, process, and deliver web pages to clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP).

Pages delivered are most frequently HTML documents, which may include images, style sheets, and scripts in addition to the text content.

Multiple web servers may be used for a high-traffic website.

A user agent, commonly a web browser or web crawler, initiates communication by requesting a specific resource using HTTP and the server responds with the content of that resource or an error message if unable to do so. The resource is typically a real file on the server's secondary storage, but this is not necessarily the case and depends on how the webserver is implemented.

While the primary function is to serve content, full implementation of HTTP also includes ways of receiving content from clients. This feature is used for submitting web forms, including the uploading of files.

Chapter II

Mandatory part

Program name	`webserv`
Turn in files	
Makefile	`Yes`
Arguments	
External functs.	`Everything in C++ 98. htons, htonl, ntohs, ntohl, select, poll, epoll (epoll_create, epoll_ctl, epoll_wait), kqueue (kqueue, kevent), socket, accept, listen, send, recv, bind, connect, inet_addr, setsockopt, getsockname, fcntl.`
Libft authorized	`No`
Description	`Write an HTTP server in C++ 98`

- You can use every macro and define like FD_SET, FD_CLR, FD_ISSET, FD_ZERO (understanding what they do and how they do it is very useful.)
- You must write an HTTP server in C++ 98.
- If you need more C functions, you can use them but always prefer C++.
- The C++ standard must be C++ 98. Your project must compile with it.
- No external library, no Boost, etc...
- Try to always use the most "C++" code possible (for example use <cstring> over <string.h>).
- Your server must be compatible with the web browser of your choice.
- We will consider that Nginx is HTTP 1.1 compliant and may be used to compare headers and answer behaviors.
- In the subject and the scale we will mention poll but you can use equivalent like select, kqueue, epoll.
- It must be non-blocking and use only 1 poll (or equivalent) for all the IO between the client and the server (listens includes).

- poll (or equivalent) should check read and write at the same time.
- Your server should never block and the client should be bounce properly if necessary.
- You should never do a read operation or a write operation without going through poll (or equivalent).
- Checking the value of errno is strictly forbidden after a read or a write operation.
- A request to your server should never hang forever.
- You server should have default error pages if none are provided.
- Your program should not leak and should never crash, (even when out of memory if all the initialization is done)
- You can't use fork for something else than CGI (like php or python etc...)
- You can't execve another webserver...
- Your program should have a config file in argument or use a default path.
- You don't need to use poll (or equivalent) before reading your config file.
- You should be able to serve a fully static website.
- Client should be able to upload files.
- Your HTTP response status codes must be accurate.
- You need at least GET, POST, and DELETE methods.
- Stress tests your server it must stay available at all cost.
- Your server can listen on multiple ports (See config file).

We've let you use fcntl because mac os X doesn't implement write the same way as other Unix OS.
You must use non-blocking FD to have a result similar to other OS.

Because you are using non-blocking FD, you could use read/recv or write/send functions without polling (or equivalent) and your server would be not blocking. But we don't want that.
Again trying to read/recv or write/send in any FD without going through a poll (or equivalent) will give you a mark equal to 0 and the end of the evaluation.

You can only use fcntl as follow: `fcntl(fd, F_SETFL, O_NONBLOCK);`
Any other flags are forbidden

- In this config file we should be able to:

`You can inspire yourself from the "server" part of Nginx configuration file`

  - choose the port and host of each "server"
  - setup the server_names or not
  - The first server for a host:port will be the default for this host:port (meaning it will answer to all request that doesn't belong to an other server)
  - setup default error pages
  - limit client body size
  - setup routes with one or multiple of the following rules/configuration (routes wont be using regexp):
    - define a list of accepted HTTP Methods for the route
    - define an HTTP redirection.
    - define a directory or a file from where the file should be search (for example if url /kapouet is rooted to /tmp/www, url /kapouet/pouic/toto/pouet is /tmp/www/pouic/toto/pouet)
    - turn on or off directory listing
    - default file to answer if the request is a directory
    - execute CGI based on certain file extension (for example .php)
      - You wonder what a CGI is ?
      - Because you won't call the CGI directly use the full path as PATH_INFO
      - Just remembers that for chunked request, your server needs to unchunked it and the CGI will expect EOF as end of the body.
      - Same things for the output of the CGI. if no content_length is returned from the CGI, EOF will mean the end of the returned data.
      - Your program should call the cgi with the file requested as first argument
      - the cgi should be run in the correct directory for relativ path file access
      - your server should work with one CGI (php-cgi, python...)
    - make the route able to accept uploaded files and configure where it should be saved

You must provide some configuration files and default basic files to test/demonstrate every feature is working during evaluation.

If you've got a question about one behavior, you should compare your program behavior with Nginx. For example, check how server_name works... We've shared with you a small tester it's not mandatory to pass it if everything works fine with your browser and tests but it can help you hunt some bugs.

Please read the RFC and do some tests with telnet and Nginx before starting this project. Even if you don't have to implement all the RFC reading it will help you make the required features.

The important thing is resilience. Your server should never die.

Do not test with only one program, write your test with a language quick to write/use, like python or golang, etc... you can even do your test in c or c++

Chapter III

Bonus part

- If the Mandatory part is not perfect don't even think about bonuses
- Support cookies and Session management (prepare quick examples).
- Handle multiple CGI.