CodeCrafters

Build your own HTTP server

Learn about TCP servers, the HTTP protocol and more

Start Building

This challenge is free until 1 April 2024!

HTTP is the protocol that powers the web. In this challenge, you'll build a HTTP server that's capable of handling simple GET/POST requests, serving files and handling multiple concurrent connections.

Along the way, we'll learn about TCP connections, HTTP headers, HTTP verbs, handling multiple connections and more.

Stages

Bind to a port	VERY EASY
Respond with 200	VERY EASY
Respond with 404	EASY
Respond with content	EASY
Parse headers	EASY
Concurrent connections	EASY
Get a file	MEDIUM
Post a file	MEDIUM

> There are few sites I like as much that have a step by step guide. The real-time feedback is so good, it's creepy!

Ananthalakshmi Sankar
Automation Engineer at Apple

> I think the instant feedback right there in the git push is really cool. Didn't even know that was possible!

Patrick Burris
Senior Software Developer, CenturyLink

Programming challenges for seasoned developers.

CHALLENGES

Git

Redis

Docker

SQLite

Grep

BitTorrent

HTTP Server

DNS Server

SUPPORT

Docs

Status

COMPANY

About

LEGAL

Terms

Changelog Privacy

Respond with 200 #IA4 In-progress

How to pass this stage

In this stage, you'll implement your own solution. Unlike stage 1, your repository doesn't contain commented code to pass this stage.

98% of users who attempt this stage complete it.

Step 1: Read instructions Mark as complete

The "Your Task" card below contains a description of what you need to implement to pass tests.

We also recommend taking a look at:

- The Code Examples tab, which contains code examples from other users.
- The Screencasts tab, if you prefer more detailed video walkthroughs.

Mark this step as complete once you've read the instructions and are ready to implement your solution.

Mark as complete

Step 2: Implement solution

Step 3: Run tests

Your Task In-progress VERY EASY

In this stage, your server will respond to an HTTP request with a `200` response.

HTTP response

An HTTP response is made up of three parts, each separated by a CRLF (`\r\n`):

1. Status line.
2. Zero or more headers, each ending with a CRLF.
3. Optional response body.

In this stage, your server's response will only contain a status line. Here's the response your server must send:

```
HTTP/1.1 200 OK\r\n\r\n
```

Here's a breakdown of the response:

```
// Status line
HTTP/1.1  // HTTP version
200       // Status code
OK        // Optional reason phrase
\r\n      // CRLF that marks the end of the status line

// Headers (empty)
\r\n      // CRLF that marks the end of the headers

// Response body (empty)
```

Tests

The tester will execute your program like this:

back next Stage 2/8 Subscribe

Respond with 200 In-progress

Listening for a git push...

Instructions Code Examples

Want to run tests without git? Install our CLI and run `codecrafters test`.

Your Task In-progress Share Feedback VERY EASY

In this stage, you'll respond to a HTTP request with a 200 OK response.

Your program will need to:

- Accept a TCP connection
- Read data from the connection (we'll get to parsing it in later stages)
- Respond with `HTTP/1.1 200 OK\r\n\r\n` (there are two `\r\n` s at the end)
  - `HTTP/1.1 200 OK` is the HTTP Status Line.
  - `\r\n` , also known as CRLF, is the end-of-line marker that HTTP uses.
  - The first `\r\n` signifies the end of the status line.
  - The second `\r\n` signifies the end of the response headers section (which is empty in this case).

It's okay to ignore the data received from the connection for now. We'll get to parsing it in later stages.

For more details on the structure of a HTTP response, view the MDN docs.

View Code Examples

Listening for a git push... Show logs

```
$ ./your_server.sh
```

The tester will then send an HTTP `GET` request to your server:

```
$ curl -i http://localhost:4221
```

Your server must respond to the request with the following response:

```
HTTP/1.1 200 OK\r\n\r\n
```

Notes

- You can ignore the contents of the request. We'll cover parsing requests in later stages.
- For more information about HTTP responses, see the MDN Web Docs on HTTP responses or the HTTP/1.1 specification.
- This challenge uses HTTP/1.1.

View Code Examples View Screencasts

Collapse ↑

Hints

Filter by Rust

Write Preview

Found an interesting resource? Share it with the community.

Ready to run tests... Show logs

Instructions Code Examples Stage 2/8 In-progress

For more details on the structure of a HTTP response, view the MDN docs.

View Code Examples

Listening for a git push...

How to pass this stage

Welcome to Stage 2! Unlike stage 1, your repository doesn't contain commented code to pass this stage - you'll be writing your own. Don't worry, this stage should be pretty easy. Almost everyone who attempts this stage passes it.

Step 1: Implement your solution

We recommend taking a look at the Code Examples tab, which contains code examples from other users.

Step 2: Commit and push your changes.

Run the following commands to submit your changes:

```
git add .
git commit -m "pass 2nd stage" # any msg
git push origin master
```

Listening for a git push...

Hints

Filter by Rust

Write Preview

Listening for a git push... Show logs

Collapse

Hints

Filter by Rust

Write Preview

Found an interesting resource? Share it with the community.

Markdown supported. Comment

DavidOrtegaFarrerons 7 months ago

Take care when copying "HTTP/1.1 200 OK\r\n\r\n" from the post, in PHPStorm it copied like "HTTP/1.1 200 OK\r\n\r\n" (Notice the escaping characters). Spent 30 minutes with it.. hope it helps someone!

6 reply

chenhunghan 6 months ago

Take a look at this section of the Rust book on how to write a `HTTP/1.1 200 OK\r\n\r\n` response. https://doc.rust-lang.org/book/ch20-01-single-threaded.html#writing-a-response

4 reply

Ready to run tests... Show logs

Instructions

Stage 1/8 Completed

```
}
```

Step 2: Commit and push your changes.

Run the following commands to submit your changes:

```
git add .
git commit -m "pass 1st stage" # any msg
git push origin master
```

You completed this stage yesterday.

Hints

Filter by Rust

Write Preview

Found an interesting resource? Share it with the community.

Markdown supported. Comment

stefanscheidt 5 months ago

To try this locally on macOS, you could run `./your_server.sh` in one terminal session, and `nc -vz 127.0.0.1 4221` in another. (`-v` gives more verbose output, `-z` just scan for listening daemons, without sending any data to them.)

136 reply

Listening for a git push... Show logs

Extract URL path #IH0 Pending

Complete previous stages to gain access to this stage.

Your Task Pending EASY

In this stage, your server will extract the URL path from an HTTP request, and respond with either a `200` or `404`, depending on the path.

HTTP request

An HTTP request is made up of three parts, each separated by a CRLF (`\r\n`):

1. Request line.
2. Zero or more headers, each ending with a CRLF.
3. Optional request body.

Here's an example of an HTTP request:

```
GET /index.html HTTP/1.1\r\nHost: localhost:4221\r\nUser-Agent: curl/7.64.1\
```

Here's a breakdown of the request:

```
// Request line
GET                           // HTTP method
/index.html                   // Request target
HTTP/1.1                      // HTTP version
\r\n                          // CRLF that marks the end of the request

// Headers
```

```
// Request line
GET                           // HTTP method
/index.html                   // Request target
HTTP/1.1                      // HTTP version
\r\n                          // CRLF that marks the end of the request line

// Headers
Host: localhost:4221\r\n      // Header that specifies the server's host and
User-Agent: curl/7.64.1\r\n   // Header that describes the client's user agen
Accept: */*\r\n               // Header that specifies which media types the
\r\n                          // CRLF that marks the end of the headers

// Request body (empty)
```

The "request target" specifies the URL path for this request. In this example, the URL path is `/index.html`.

Note that each header ends in a CRLF, and the entire header section also ends in a CRLF.

Tests

The tester will execute your program like this:

```
$ ./your_server.sh
```

The tester will then send two HTTP requests to your server.

First, the tester will send a `GET` request, with a random string as the path:

```
$ curl -i http://localhost:4221/abcdefg
```

Your server must respond to this request with a `404` response:

3. Optional request body.

Here's an example of an HTTP request:

```
.1\r\nHost: localhost:4221\r\nUser-Agent: curl/7.64.1\r\nAccept: */*\r\n\r\n
```

Here's a breakdown of the request:

```
            // HTTP method
            // Request target
            // HTTP version
            // CRLF that marks the end of the request line


221\r\n     // Header that specifies the server's host and port
7.64.1\r\n  // Header that describes the client's user agent
            // Header that specifies which media types the client can accept
            // CRLF that marks the end of the headers

empty)
```

The "request target" specifies the URL path for this request. In this example, the URL path is `/index.html` .

Note that each header ends in a CRLF, and the entire header section also ends in a CRLF.

Tests

The tester will execute your program like this:

```
$ ./your_server.sh
```

Your server must respond to this request with a `404` response:

```
HTTP/1.1 404 Not Found\r\n\r\n
```

Then, the tester will send a `GET` request, with the path `/`:

```
$ curl -i http://localhost:4221
```

Your server must respond to this request with a `200` response:

```
HTTP/1.1 200 OK\r\n\r\n
```

Notes

- You can ignore the headers for now. You'll learn about parsing headers in a later stage.
- In this stage, the request target is written as a URL path. But the request target actually has four possible formats. The URL path format is called the "origin form," and it's the most commonly used format. The other formats are used for more niche scenarios, like sending a request through a proxy.
- For more information about HTTP requests, see the MDN Web Docs on HTTP requests or the HTTP/1.1 specification.

View Code Examples View Screencasts

Collapse ↑

kesava-karri 7 months ago

My suggestion is to use `curl` to send custom http requests and debug locally. Eg:

```
curl -i -X GET http://localhost:4221/index.html
```

105 reply

LakshyaMittal3301 4 months ago

It's really helpful to use curl. I was stuck for some time, then realized that I needed to close the socket after the server received the data for curl to work. Also, you can do `curl -i localhost:4221` to print the response you receive.

2

kotharePrajwal 8 months ago

For stage 1 and 2 you can refer to this explanation by arpit bhiyani

https://youtu.be/f9gUFy-9uCM?si=yDVrSUJfdQ8o_OE6

40 reply

sherubthakur 7 months ago

The HTTP request lacks an explicit End-of-File (EOF) marker. Exercise caution when reading from the stream and avoid relying on functions that expect an EOF signal.

38 reply

mule-518 13 days ago

Thanks for the heads up. This was what was tripping me up the first time I attempted to solve the challenge.

4

Ready to run tests... Show logs

Respond with body #CN2 Pending

Complete previous stages to gain access to this stage.

Your Task Pending Share Feedback EASY

In this stage, you'll implement the `/echo/{str}` endpoint, which accepts a string and returns it in the response body.

Response body

A response body is used to return content to the client. This content may be an entire web page, a file, a string, or anything else that can be represented with bytes.

Your `/echo/{str}` endpoint must return a `200` response, with the response body set to given string, and with a `Content-Type` and `Content-Length` header.

Here's an example of an `/echo/{str}` request:

```
GET /echo/abc HTTP/1.1\r\nHost: localhost:4221\r\nUser-Agent: curl/7.64.1\r\
```

And here's the expected response:

```
HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 3\r\n\r\nabc
```

Here's a breakdown of the response:

Here's a breakdown of the response:

```
// Status line
HTTP/1.1 200 OK
\r\n                          // CRLF that marks the end of the status line

// Headers
Content-Type: text/plain\r\n  // Header that specifies the format of the res
Content-Length: 3\r\n         // Header that specifies the size of the respo
\r\n                          // CRLF that marks the end of the headers

// Response body
abc                           // The string from the request
```

The two headers are required for the client to be able to parse the response body. Note that each header ends in a CRLF, and the entire header section also ends in a CRLF.

Tests

The tester will execute your program like this:

```
$ ./your_server.sh
```

The tester will then send a `GET` request to the `/echo/{str}` endpoint on your server, with some random string.

```
$ curl -i http://localhost:4221/echo/abc
```

Your server must respond with a `200` response that contains the following parts:

- `Content-Type` header set to `text/plain` .
- `Content-Length` header set to the length of the given string.

Instructions Code Examples Screencasts Forum Stage #CN2 Pending

```
$ curl -i http://localhost:4221/echo/abc
```

Your server must respond with a `200` response that contains the following parts:

- `Content-Type` header set to `text/plain` .
- `Content-Length` header set to the length of the given string.
- Response body set to the given string.

```
HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 3\r\n\r\nabc
```

Notes

- For more information about HTTP responses, see the MDN Web Docs on HTTP responses or the HTTP/1.1 specification.

View Code Examples View Screencasts

Collapse

Hints

Filter by Rust

Write Preview

Found an interesting resource? Share it with the community.

Markdown supported

Ready to run tests... Show logs

Here's an example of an `/echo/{str}` request:

```
.1\r\nHost: localhost:4221\r\nUser-Agent: curl/7.64.1\r\nAccept: */*\r\n\r\n
```

And here's the expected response:

```
HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 3\r\n\r\nabc
```

Here's a breakdown of the response:

```
            // CRLF that marks the end of the status line

/plain\r\n  // Header that specifies the format of the response body
r\n         // Header that specifies the size of the response body, in bytes
            // CRLF that marks the end of the headers

            // The string from the request
```

The two headers are required for the client to be able to parse the response body. Note that each header ends in a CRLF, and the entire header section also ends in a CRLF.

Tests

The tester will execute your program like this:

```
$ ./your_server.sh
```

Instructions | Code Examples | Screencasts | Forum

Stage #CN2 Pending

Your server must respond with a `200` response that contains the following parts:

- `Content-Type` header set to `text/plain` .
- `Content-Length` header set to the length of the given string.
- Response body set to the given string.

```
HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 3\r\n\r\nabc
```

Notes

- For more information about HTTP responses, see the MDN Web Docs on HTTP responses or the HTTP/1.1 specification.

View Code Examples | View Screencasts

Collapse ↑

Hints

Filter by Rust

Write | Preview

Found an interesting resource? Share it with the community.

Markdown supported.

Ready to run tests... Show logs

Instructions Code Examples Screencasts Forum Stage #CN2 Pending

jzwood 7 months ago

test with `curl --verbose 127.0.0.1:4221/echo/abc`

24 reply

sy3c4ll 6 months ago

The conditions from previous questions still apply, meaning:

- You must reply `200 OK` to `/`
- You must reply `200 OK` to `/echo/abc` *and* return content
- You must reply `404 Not Found` to anything else

5 reply

WaterLemons2k 22 days ago

If you're stuck with error `net/http: HTTP/1.x transport connection broken: unexpected EOF'` fo a long time, remember that you need **a blank line (CRLF)** to indicate the end of the HTTP headers and the beginning of the body.

References:

- https://stackoverflow.com/a/74061022
- https://docs.python.org/3/library/http.server.html#http.server.BaseHTTPRequestHandle end_headers
- https://www.ibm.com/docs/en/cics-ts/beta?topic=protocol-http-responses

1 reply

lalashkin 1 month ago

If you ever will encounter following error, after all tests seemingly pass:

```
remote: anti-cheat (ac1) failed.
remote: Are you sure you aren't running this against an actual HTTP server?
```

Ready to run tests... Show logs

The conditions from previous questions still apply, meaning:

- You must reply `200 OK` to `/`
- You must reply `200 OK` to `/echo/abc` *and* return content
- You must reply `404 Not Found` to anything else

5 reply

WaterLemons2k 22 days ago

If you're stuck with error `net/http: HTTP/1.x transport connection broken: unexpected EOF'` fo a long time, remember that you need **a blank line (CRLF)** to indicate the end of the HTTP headers and the beginning of the body.

References:

- https://stackoverflow.com/a/74061022
- https://docs.python.org/3/library/http.server.html#http.server.BaseHTTPRequestHandle end_headers
- https://www.ibm.com/docs/en/cics-ts/beta?topic=protocol-http-responses

1 reply

lalashkin 1 month ago

If you ever will encounter following error, after all tests seemingly pass:

```
remote: anti-cheat (ac1) failed.
remote: Are you sure you aren't running this against an actual HTTP server?
```

It seems that test runners are considering `Server: <n>` header something that you won't use at this stage Just remove it from your response and you're good to go!

1 reply

Read header #FS3 Pending

Complete previous stages to gain access to this stage.

Your Task Pending Share Feedback EASY

In this stage, you'll implement the `/user-agent` endpoint, which reads the `User-Agent` request header and returns it in the response body.

The `User-Agent` header

The `User-Agent` header describes the client's user agent.

Your `/user-agent` endpoint must read the `User-Agent` header, and return it in your response body. Here's an example of a `/user-agent` request:

```
// Request line
GET
/user-agent
HTTP/1.1
\r\n

// Headers
Host: localhost:4221\r\n
User-Agent: foobar/1.2.3\r\n  // Read this value
Accept: */*\r\n
\r\n

// Request body (empty)
```

Here is the expected response:

Here is the expected response:

```
// Status line
HTTP/1.1 200 OK                 // Status code must be 200
\r\n

// Headers
Content-Type: text/plain\r\n
Content-Length: 12\r\n
\r\n

// Response body
foobar/1.2.3                    // The value of `User-Agent`
```

Tests

The tester will execute your program like this:

```
$ ./your_server.sh
```

The tester will then send a `GET` request to the `/user-agent` endpoint on your server. The request will have a `User-Agent` header.

```
$ curl -i --header "User-Agent: foobar/1.2.3" http://localhost:4221/user-age
```

Your server must respond with a `200` response that contains the following parts:

- `Content-Type` header set to `text/plain` .
- `Content-Length` header set to the length of the `User-Agent` value.
- Message body set to the `User-Agent` value.

```
HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 12\r\n\
```

Instructions | Code Examples | Screencasts | Forum | Stage #FS3 | Pending

The tester will then send a `GET` request to the `/user-agent` endpoint on your server. The request will have a `User-Agent` header.

```
curl -i --header "User-Agent: foobar/1.2.3" http://localhost:4221/user-agent
```

Your server must respond with a `200` response that contains the following parts:

- `Content-Type` header set to `text/plain` .
- `Content-Length` header set to the length of the `User-Agent` value.
- Message body set to the `User-Agent` value.

```
200 OK\r\nContent-Type: text/plain\r\nContent-Length: 12\r\n\r\nfoobar/1.2.3
```

Notes

- Header names are [case-insensitive](#).

View Code Examples | View Screencasts

Collapse ↑

Hints

Filter by Rust

Coding Blade

Write | Preview

Found an interesting resource? Share it with the community.

Ready to run tests... Show logs

Instructions Code Examples Screencasts Forum Stage #FS3 Pending

200 OK\r\nContent-Type: text/plain\r\nContent-Length: 12\r\n\r\nfoobar/1.2.3

Notes

- Header names are case-insensitive.

View Code Examples View Screencasts

Collapse

Hints

Filter by Rust

Write Preview

Found an interesting resource? Share it with the community.

Markdown supported. Comment

jzwood 7 months ago

test with `curl --verbose 127.0.0.1:4221/user-agent`

36 reply

Ready to run tests... Show logs

Concurrent connections #EJ5 Pending

Complete previous stages to gain access to this stage.

Your Task Pending EASY

In this stage, you'll add support for concurrent connections.

Tests

The tester will execute your program like this:

```
$ ./your_server.sh
```

Then, the tester will create multiple concurrent TCP connections to your server. (The exact number of connections is determined at random.) After that, the tester will send a single `GET` request through each of the connections.

```
$ (sleep 3 && printf "GET / HTTP/1.1\r\n\r\n") | nc localhost 4221 &
$ (sleep 3 && printf "GET / HTTP/1.1\r\n\r\n") | nc localhost 4221 &
$ (sleep 3 && printf "GET / HTTP/1.1\r\n\r\n") | nc localhost 4221 &
```

Your server must respond to each request with the following response:

```
HTTP/1.1 200 OK\r\n\r\n
```

View Code Examples View Screencasts

Instructions Code Examples Screencasts Forum Stage #EJ5 Pending

sherubthakur 8 months ago

An easy way to test this locally would be to use `nc` to create a connection to your server and then use `curl` to hit it with a request. i.e. `nc localhost 4221` in one window followed by `curl localhost:4221` is a seprate terminal window.

51 reply

ryan-gang staff 7 months ago

For windows, which doesn't have `nc` built-in. `ncat` is a suitable substitute.
Docs : https://nmap.org/ncat/
Windows portable version : https://nmap.org/dist/ncat-portable-5.59BETA1.zip
After unzipping its just `ncat localhost 4221`

14

that-ambuj 7 months ago

You can use a server testing cli called oha , which sends multiple concurrent requests by default. You can use it like `oha http://localhost:4221` .

47 reply

themilar 2 months ago

nice looking tool, thanks for recommending, could come in handy beyond this challenge

1

Remokc 1 month ago

For anyone having problems with `undefined reference to 'pthread_create'` :
https://github.com/codecrafters-io/build-your-own-redis/issues/112#issuecomment-1872592856

6 reply

GilTeixeira 7 months ago

If curl is hanging your request, you need to either close the connection or supply a

Ready to run tests... Show logs

challenge

1

Remokc 1 month ago

For anyone having problems with `undefined reference to 'pthread_create'` : https://github.com/codecrafters-io/build-your-own-redis/issues/112#issuecomment-1872592856

6 reply

GilTeixeira 7 months ago

If curl is hanging your request, you need to either close the connection or supply a content-length header field (set to zero). See RFC 7230, Section 3.3.3. (source: https://stackoverflow.com/a/52869383)

5 reply

daniel-j-anderson-dev 6 months ago

check out The Rust Book's final chapter for a good and simple ThreadPool

2 reply

mevimo 4 months ago

For those seeing strange results when using a server testing tool like `oha` - make sure you implement keep-alive. It is the default in HTTP/1.1, and if not implemented, you'll probably close the connection while the server testing tool still means to use it for the next request. This won't present itself as a problem when testing with `curl` , because that's a one-connection-one-request scenario.

1 reply

Ready to run tests... Show logs

Get a file #AP6 Pending

Complete previous stages to gain access to this stage.

Your Task Pending MEDIUM

In this stage, your server will need to return the contents of a file.

The tester will execute your program with a `--directory` flag like this:

```
./your_server.sh --directory <directory>
```

It'll then send you a request of the form `GET /files/<filename>` .

If `<filename>` exists in `<directory>` , you'll need to respond with a 200 OK response. The response should have a content type of `application/octet-stream` , and it should contain the contents of the file as the body.

If the file doesn't exist, return a 404.

We pass in absolute path to your program using the `--directory` flag.

View Code Examples View Screencasts

Collapse ↑

Hints

Filter by Ru

Ready to run tests... Show logs

response. The response should have a content type of `application/octet-stream`, and it should contain the contents of the file as the body.

If the file doesn't exist, return a 404.

We pass in absolute path to your program using the `--directory` flag.

View Code Examples View Screencasts

Collapse

Hints

Filter by Rust

Write Preview

Found an interesting resource? Share it with the community.

Markdown supported. Comment

ryan-gang staff 7 months ago

Just FYI the tests expect the `Content-Length` header.

75 reply

Ready to run tests... Show logs

Post a file #QV8 Pending

Complete previous stages to gain access to this stage.

Your Task Pending Share Feedback MEDIUM

In this stage, your server will need to accept the contents of a file in a POST request and save it to a directory.

Just like in the previous stage, the tester will execute your program with a `--directory` flag like this:

```
./your_server.sh --directory <directory>
```

It'll then send you a request of the form `POST /files/<filename>`. The request body will contain the contents of the file.

You'll need to fetch the contents of the file from the request body and save it to `<directory>/<filename>`. The response code returned should be 201.

We pass in absolute path to your program using the `--directory` flag.

View Code Examples View Screencasts

Collapse ↑

Hints

Filter by Rus

Instructions Code Examples Screencasts Forum Stage #QV8 Pending

Jorjon 7 months ago

To test locally using cURL:

```
curl -vvv -d "hello world" localhost:4221/files/readme.txt
```

51 reply

yevelnad 7 months ago

this "\0" will make your life a nightmare.

23 reply

friendlymatthew 7 months ago

was pulling out my hair for the same thing haha.

note to myself: when converting buffer to a string, consider only the portion that was actually read as there may contain the null byte :)

14

mwettste 7 months ago

Haha same here. Lesson learned 😅

7

that-ambuj 7 months ago

For people using Rust, beware not to use `.read_to_string()` methods using something like a `BufReader`, this can potentially block your the thread and potentially the whole program(I don't who why but it just wouldn't work for me). instead use `.seek(&mut buf)` available on `TcpStream` because it seems to just work.

9 reply

skunk-143 7 months ago

Yes, because `read_to_string()` is like `read_to_end()` and it waits for the EOF, which means reset of TCP connection

19

Ready to run tests... Show logs

51 reply

yevelnad 7 months ago

this "\0" will make your life a nightmare.

23 reply

friendlymatthew 7 months ago

was pulling out my hair for the same thing haha.

note to myself: when converting buffer to a string, consider only the portion that was actually read as there may contain the null byte :)

14

mwettste 7 months ago

Haha same here. Lesson learned 😅

7

that-ambuj 7 months ago

For people using Rust, beware not to use `.read_to_string()` methods using something like a `BufReader`, this can potentially block your the thread and potentially the whole program(I don't who why but it just wouldn't work for me). instead use `.seek(&mut buf)` available on `TcpStream` because it seems to just work.

9 reply

skunk-143 7 months ago

Yes, because `read_to_string()` is like `read_to_end()` and it waits for the EOF, which means reset of TCP connection

19

CodeCrafters

Build your own HTTP server

Learn about TCP servers, the HTTP protocol and more

Start Building

This challenge is free until 1 April 2024!

HTTP is the protocol that powers the web. In this challenge, you'll build a HTTP server that's capable of handling simple GET/POST requests, serving files and handling multiple concurrent connections.

Along the way, we'll learn about TCP connections, HTTP headers, HTTP verbs, handling multiple connections and more.

Stages

Bind to a port	VERY EASY
Respond with 200	VERY EASY
Respond with 404	EASY
Respond with content	EASY
Parse headers	EASY
Concurrent connections	EASY
Get a file	MEDIUM
Post a file	MEDIUM

> There are few sites I like as much that have a step by step guide. The real-time feedback is so good, it's creepy!

Ananthalakshmi Sankar
Automation Engineer at Apple

> I think the instant feedback right there in the git push is really cool. Didn't even know that was possible!

Patrick Burris
Senior Software Developer, CenturyLink

Programming challenges for seasoned developers.

CHALLENGES

Git

Redis

Docker

SQLite

Grep

BitTorrent

HTTP Server

DNS Server

SUPPORT

Docs

Status

COMPANY

About

LEGAL

Terms

Changelog Privacy