Backend from the Beginning, Pt 1: Introduction, TCP, DNS, HTTP

HTML: dark, light, plain
PDF: dark, light, plain

A software article by Efron Amber Licht

September 2023

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

This article is part of a series on backend web development in Go. It will cover the basics of backend web development, from the ground up, without using a framework. It will cover the basics of TCP, DNS, HTTP, the `net/http` and `encoding/json` packages, middleware and routing. It should give you everything you need to get started writing professional backend web services in Go.

Source code for this article (& the entire blog) is publicly available on my gitlab.

1. Introduction: when I hear the word 'framework' I reach for my gun

One of the most common questions I get from new developers starting with Go is **"what web framework should I use?"** The answer I always give is "you don't need a framework", but the problem is, backend devs are *used* to frameworks.

Thinking about it, the motivation is understandable: engineers are under a lot of pressure, and the internet *seems* really complicated, the idea of learning all these layers of abstraction (tcp, http, etc) is daunting, and *everyone else seems to use a framework* - in most languages (javascript, python, etc), it's practically required. There's only one problem with this: **it means you never learn how things actually work**. Constantly relying on suites of specialized tools rather than learning the basics is the equivalent of being a senior chef who can't use a knife. Sure, you can argue that your fancy food processor chops faster, but the second you need to do something for which your pre-packaged tools aren't designed, you're screwed; you have no idea how to do it yourself and no time to learn.

This may sound like an exaggeration, but **I have now met four different senior software engineers who couldn't tell me how to make a HTTP request to google without a framework**.

For the record, you send this message to `142.250.189.14:80`:

```
GET / HTTP/1.1
Host: google.com
```

It's five words.

If you don't know what stuff means or how I got that IP address, don't worry; we'll get to that. The point is, it's not actually that hard; hard is ten thousand layers of callbacks and libraries and frameworks and tools and languages and abstractions and indirections and wrappers around wrappers. The problem is, most software engineers are so used to having so many layers of abstraction between them and the network that 'getting to the bottom' seems impossible. Now, some may argue that this is OK, because the increased power & flexibility of frameworks leads to faster, better, software development. The only problem is, software isn't getting better; it's getting *measurably worse*. Both desktop software and web pages are measurably slower year over year. Software is getting slower faster than computers are speeding up. With this in mind, it's no surprise that software is **drowning in complexity**. Every year, we add more layers of abstraction, more libraries, more frameworks, more tools, more languages, more everything, but even our 'experts' don't understand the basics of the stuff they're building. It's no wonder that software is so slow and buggy and impossible to maintain.

> "An idiot admires complexity, a genius admires simplicity"
>
> • Terry A. Davis

Of course, knowing that things are done badly doesn't help you learn how to do it *well*, so I'm writing this series of articles to try and fill the gap by teaching the basics of backend web development in Go. Each article will be filled with *real* programs you can run on your computer, not cherry-picked code samples that don't even compile.

This series will not be enough to teach you everything. At best it will expose you to enough of how things *really* work that you can start seeing the edges of your knowledge and begin filling in the gaps yourself. It will also necessarially have to simplify or omit some details or tell "white lies" to make things easier to understand; there's no substitute for experience (or for reading the source code & documentation of the standard library). It will also largely omit databases; I hope to make those the subject of a future series.

That said, I hope it will help.

2. Series overview

2.1. backend basics, part 1: TCP, DNS, & HTTP

- What is the internet? What problems does it solve? What's TCP/IP? How do computers talk to each other?
- What's DNS? How do we turn www.google.com into an IP address?
- What's HTTP and how does it work? How would we read or write a HTTP request or response by hand, without a library?
- Building a Request/Response library from scratch

2.2. Practical backend: `net/http` and `encoding/json`

In the second article, we'll graduate to using the `net/http` and `encoding/json` packages to build basic web clients and servers that can handle most day-to-day backend workloads. We'll start diving into Go's standard library and the show how it provides everything you need for basic client/server HTTP communication; using `net/http` and `net/url` to send and receive HTTP requests and responses, `encoding/json` to manage our API payloads, and `context` to manage timeouts and cancellation.

2.3. Finishing touches: middlewares, routing, and basic database access

In the third article, we'll cover middleware and routing, the two 'missing pieces' of the `net/http` package. These are the bits that usually make people reach for a framework, but they're actually pretty simple to implement yourself. We'll also cover basic database access using the `database/sql` package.

3. What is backend?

'Backend' is connecting together computers via the internet. You know what a computer is, so...

3.1. What's the internet anyways?

What is the internet? No, I'm serious. What is the problem the internet solves? The internet is a *network* of computers that can reliably communicate with each other even if some of the computers 'in the middle' are down. It allows you to reliably send messages (that is, text or binary data) to other computers, even if you don't know where those computers are or how they're connected to you. You can send a message to another computer, so long as there is a path of computers from you (the `LOCALADDR`) to the destination computer (the `REMOTEADDR`).

To do this, the internet must solve two problems:

- How do I send a message to another computer, even if I don't have a direct connection to it? `ROUTING`
- How do I make sure that when I send a message through a network it gets to the right place, in order, and all of it gets through? `COHERENCE`
  Both problems are solved by a protocol: the **I**nternet **P**rotocol (`IP`) solves `ROUTING`, and the **T**ransmission **C**ontrol **P**rotocol (`TCP`) solves `COHERENCE`. Together, they're called `TCP/IP`. That's how the internet works.

3.2. TCP: how do I make sure that when I send a message through a network all of it gets through, in order?

The details of TCP are out of scope for this article, but at a high level it looks like this:

- You send packets of data to the remote computer. Each packet has a sequence number, ("which packet is this?") and a checksum ("did this packet get corrupted in transit?"). The remote computer sends back an acknowledgement ("I got packet 5") for each packet you send.
- If you don't get an acknowledgement for a packet, you resend it; if you get a corrupted packet, you resend it.
- This back-and-forth ensures that all of the data gets through, in order, and that you know when it doesn't.

3.3. IP: how do I make sure that when I send a message through a network it gets to the right place?

IP is more complicated. The following explanation is wrong at pretty much every level if you zoom in enough, but it's a good enough approximation for our purposes:

- Each computer on the internet has an `address`, an identifier that tells other computers how to get to it. This address is called an `IP address`, or sometimes just an `IP`.
- They also have a list of other computers it knows about, and how to get to them. This list is called a **routing table**.
- When you send a message to another computer, your computer looks at its routing table to see if it knows how to get to that computer. If it does, it sends the message to the next computer in the chain. If it doesn't, it sends the message to the next computer in the chain that it *does* know how to get to; they keep doing this until the message gets to the right computer.
- If there is no path from your computer to the destination computer, the message fails.

3.4. Addresses & Ports

OK, so how do we actually send a message to another computer? We need to know two things: the `address` of the computer we want to send a message to, and the `port` of the service we want to send a message to.

3.5. IP Addresses

IP Addresses come in two forms: `ipv4`, a 32-bit number; or `ipv6`, a 128-bit number. They look like this:

IPV4 looks like this: DDD.DDD.DDD.DDD, where DDD is a number between 0 and 255.

IPV6 looks like this: XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX, where XXXX is a 16-bit hexadecimal number; that is, each X is one of `0..=9` or `a..=f`

IP Address	Type	Note
192.168.000.001	ipv4	localhost; refers to hosting computer
192.168.0.1	ipv4	same as above; you can omit leading zeroes
0000:0000:0000:0000:0000:ffff:c0a8:0001	ipv6	refers to same computer as above; ipv4 addresses can be embedded in ipv6 addresses by prefixing them with `::ffff:`
::ffff:c0a8:0001	ipv6	same as above; you can omit leading zeroes
2a09:8280:1::a:791	ipv6	fly.io

3.6. Ports

It's common for a computer to want to host multiple internet services that behave in different ways. For example, we could want to host a game server (like `starcraft`), a web server (like this website), and a database (like `postgresql`) all on the same computer. Since they're all on the same physical computer, they'll share an IP address, so we'll need some way to tell apart requests to the file server from requests to the game server. We do this by assigning a `PORT` to each service. A port is just a number between 0 and 65535. Even if we're only hosting one service, each service needs (at least one) port.

`eblog` is hosted at port 6483. The following table lists default ports for some common services:

Service	Port
HTTP	80
HTTPS	443
SSH	22

Service	Port
SMTP	25
DNS	53
FTP	21
Postgres	5432

4. Building a basic TCP/IP server and client

Let's build a basic TCP/IP server and client to demonstrate how this works. We'll build a server that listens on port 6483, and a client that connects to it. Anything sent on stdin on the client (that is, typed into the terminal) will be sent to the server, a line at a time. Anything line received on the server will be uppercased and sent back to the client.

That is, an example session might look like this:

```
SERVER: (starts listening on port 6483)
CLIENT: (connects to server)
CLIENT: "hello, world!"
SERVER: "HELLO, WORLD!"
CLIENT: "goodbye, world!"
SERVER: "GOODBYE, WORLD!"
CLIENT: (disconnects)
```

To briefly review, the following functions and types are relevant to our examples:

function/struct	description	implements
`net.Listen`	listens for connections on a port	
`net.Dial`	connects to a server at an IP address and port	
`net.TCPConn`	bidirectional TCP connection	`io.Reader`, `io.Writer`, `net.Conn`
`net.Conn`	bidirectional network connection	`io.Reader`, `io.Writer`
`bufio.Scanner`	reads lines from a `io.Reader`	
`fmt.Fprintf`	as `fmt.Printf`, but writes to a `io.Writer`	
`flag.Int`	register an integer command line flag	
`flag.Parse`	parse previously registered command line flags	
`log.Printf`	as `fmt.Fprintf(os.Stderr, ...)`, but with a timestamp and newline	
`log.Fatalf`	as `log.Printf`, but calls `os.Exit(1)` after printing	

4.1. client

Let's write the client first: we'll call it `writetcp`.

```go
// writetcp connects to a TCP server at at localhost with the specified port (8080 by default) and forwards stdin to the server,
// line-by-line, until EOF is reached.
// received lines from the server are printed to stdout.
package main

import (
    "bufio"
    "flag"
    "fmt"
    "log"
    "net"
    "os"
)

func main() {
    const name = "writetcp"
    log.SetPrefix(name + "\t")

    // register the command-line flags: -p specifies the port to connect to
    port := flag.Int("p", 8080, "port to connect to")
    flag.Parse() // parse registered flags

    conn, err := net.DialTCP("tcp", nil, &net.TCPAddr{Port: *port})
    if err != nil {
        log.Fatalf("error connecting to localhost:%d: %v", *port, err)
    }
    log.Printf("connected to %s: will forward stdin", conn.RemoteAddr())

    defer conn.Close()
    go func() { // spawn a goroutine to read incoming lines from the server and print them to stdout.
        // TCP is full-duplex, so we can read and write at the same time; we just need to spawn a goroutine to do the reading.

        for connScanner := bufio.NewScanner(conn); connScanner.Scan(); {

            fmt.Printf("%s\n", connScanner.Text()) // note: printf doesn't add a newline, so we need to add it ourselves

            if err := connScanner.Err(); err != nil {
                log.Fatalf("error reading from %s: %v", conn.RemoteAddr(), err)
            }
        }
```

```
41      }()
42
43      // read incoming lines from stdin and forward them to
    the server.
44      for stdinScanner := bufio.NewScanner(os.Stdin);
    stdinScanner.Scan(); { // find the next newline in stdin
45          log.Printf("sent: %s\n", stdinScanner.Text())
46          if _, err := conn.Write(stdinScanner.Bytes());
    err != nil
    { // scanner.Bytes() returns a slice of bytes up to but not
    including the next newline
47              log.Fatalf("error writing to %s: %v",
    conn.RemoteAddr(), err)
48          }
49          if _, err := conn.Write([]byte("\n")); err != nil
    { // we need to add the newline back in
50              log.Fatalf("error writing to %s: %v",
    conn.RemoteAddr(), err)
51          }
52          if stdinScanner.Err() != nil {
53              log.Fatalf("error reading from %s: %v",
    conn.RemoteAddr(), err)
54          }
55      }
56
57  }
```

4.2. server

Now let's put together the server; since it echoes back whatever it receives, in uppercase, we'll call it `tcpupperecho`.

Usually when working in backend, we want to separate your 'business logic' from the networking code. Since all of go's networking APIs use the net.Conn interface, which implements both io.Reader and io.Writer, we can write our business logic using standard text-handling functions and structs like fmt.Fprintf and bufio.Scanner.

Our server's 'business logic' will look like this:

```
1   // echoUpper reads lines from r, uppercases them, and
    writes them to w.
2   func echoUpper(w io.Writer, r io.Reader) {
3       scanner := bufio.NewScanner(r)
4       for scanner.Scan() {
5           line := scanner.Text()
6           // note that scanner.Text() strips the newline
    character from the end of the line,
7           // so we need to add it back in when we write to w.
8           fmt.Fprintf(w, "%s\n", strings.ToUpper(line))
9       }
10      if err := scanner.Err(); err != nil {
```

```
            log.Printf("error: %s", err)
        }
    }
```

Which we can then use in our server like this:

```
// tcpupperecho serves tcp connections on port 8080, reading from each connection line-by-line and writing the upper-case version of each line back to the client.

package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "log"
    "net"
    "strings"
)

func main() {
    const name = "tcpupperecho"
    log.SetPrefix(name + "\t")

    // build the command-line interface; see https://golang.org/pkg/flag/ for details.
    port := flag.Int("p", 8080, "port to listen on")
    flag.Parse()

    // ListenTCP creates a TCP listener accepting connections on the given address.
    // TCPAddr represents the address of a TCP end point; it has an IP, Port, and Zone, all of which are optional.
    // Zone only matters for IPv6; we'll ignore it for now.
    // If we omit the IP, it means we are listening on all available IP addresses; if we omit the Port, it means we are listening on a random port.
    // We want to listen on a port specified by the user on the command-line.
    // see https://golang.org/pkg/net/#ListenTCP and https://golang.org/pkg/net/#Dial for details.
    listener, err := net.ListenTCP("tcp", &net.TCPAddr{Port: *port})
    if err != nil {
        panic(err)
    }
    defer listener.Close() // close the listener when we exit main()
    log.Printf("listening at localhost: %s", listener.Addr())
```

```
36      for { // loop forever, accepting connections one at a
    time
37
38          // Accept() blocks until a connection is made, then
    returns a Conn representing the connection.
39          conn, err := listener.Accept()
40          if err != nil {
41              panic(err)
42          }
43          go echoUpper(conn, conn) // spawn a goroutine to
    handle the connection
44      }
45  }
```

4.3. Demo

Let's try it out. In one terminal, we'll run the server:

IN

```
1   go build -o tcpupperecho ./tcpupperecho.go
2   ./tcpupperecho -p 8080 # run the server, listening on port
    8080
```

OUT:

```
1   tcpupperecho    2023/09/07 10:13:13 listening at localhost:
    [::]:8080
```

Let's run the client in another terminal and send it a message:

```
1   $ go build -o writetcp ./writetcp.go
2   $ ./writetcp -p 8080 # run the client, connecting to
    localhost:8080
3   > writetcp 2023/09/07 10:20:32 connected to 127.0.0.1:8080:
    will forward stdin
4   hello
5   writetcp        2023/09/07 10:20:49 sent: hello
6   HELLO
```

And checking in back in the server terminal, we see:

```
1   tcpupperecho    2023/09/07 10:20:49 received: hello
```

5. Connecting to a server on the internet

This works fine for local addresses, but what if we want to connect to a server on the internet? Most of the time, we don't know the `IP` address of the server we want to connect to; we only know its `domain name`, like `google.com` or `eblog.fly.dev`. How do we connect to a server at a domain name?

5.1. DNS

Domain **N**ame **S**ervice, or `DNS`, is a service that maps domain names to IP addresses. It's essentially a big table that looks like this:

domain	last known ipv4	last known ipv6
google.com	142.250.217.142	2607:f8b0:4007:801::200e
eblog.fly.dev	66.241.125.53	2a09:8280:1::37:6bbc

There are multiple `DNS` providers. Your ISP usually provides one, and there are public ones like Google's, available at both `8.8.8.8` and `4.4.4.4`. (Since you can't resolve a domain name without knowing the IP address of a DNS server, you need to know at least one IP 'by heart' to get started.)

Browsers and other clients use `DNS` service to look up the IP address of a domain name.

5.2. Finding the IP address of a server

2021/08/18 16:00:00 tcpupperecho listening at localhost:
OK, so we want to connect to a server at a **web address**: say, `https://eblog.fly.dev`. How do we do that? Well, first we need to get the IP address of the server. The **domain name service**, or `DNS`, is a service that maps domain names to IP addresses. You can use the built-in `nslookup` command to look up the IP address of a domain name from your command-line on windows, mac, or linux.

IN:

```
nslookup eblog.fly.dev
```

OUT:

```
Server:  UnKnown
Address:  192.168.1.1

Non-authoritative answer:
Name:    eblog.fly.dev
Addresses:  2a09:8280:1::37:6bbc
          66.241.125.53
```

Within a Go program, use `net.LookupIP` to look up the IP address or addresses of a domain name. The following full program duplicates the functionality of `nslookup`:

```
// dns is a simple command line tool to lookup the ip address of a host;
// it prints the first ipv4 and ipv6 addresses it finds, or "none" if none are found.
package main

```

```
 5  import (
 6      "fmt"
 7      "log"
 8      "net"
 9      "os"
10  )
11
12  func main() {
13      if len(os.Args) != 2 {
14          log.Printf("%s: usage: <host>", os.Args[0])
15          log.Fatalf("expected exactly one argument; got
    %d", len(os.Args)-1)
16      }
17      host := os.Args[1]
18      ips, err := net.LookupIP(host)
19      if err != nil {
20          log.Fatalf("lookup ip: %s: %v", host, err)
21      }
22      if len(ips) == 0 {
23          log.Fatalf("no ips found for %s", host) // this
    should never happen, but just in case
24      }
25      // print the first ipv4 we find
26      for _, ip := range ips {
27          if ip.To4() != nil {
28              fmt.Println(ip)
29              goto IPV6 // goto considered awesome
30          }
31      }
32      fmt.Printf("none\n") // only print "none" if we don't
    find any ipv4 addresses
33
34  IPV6: // print the first ipv6 we find
35      for _, ip := range ips {
36          if ip.To4() == nil {
37              fmt.Println(ip) // we don't need to check for
    nil here, since we know we have at least one ip address
38              return
39          }
40      }
41      fmt.Printf("none\n")
42  }
43
```

IN:

```
1  go build -o dns ./dns.go # build the dns command
2  ./dns eblog.fly.dev # run the dns command
```

OUT:

```
1  66.241.125.53
2  2a09:8280:1::37:6bbc
```

5.3. putting it together: DNS & HTTP

We now have everything we need to for the basics of internet browsing: we can look up the IP address of a domain name, and we can connect to a server at an IP address and port.

When you type a URL into your browser, it does the following:

- looks up the IP address of the domain name
- connects to the server at that IP address and port
- sends a `HTTP` request to the server
- reads and displays the `HTTP` response - usually, a webpage.

But wait, what's HTTP? The **H**yper**T**ext **T**ransfer **P**rotocol is a **text-based** protocol for sending messages over the internet.

HTTP is not nearly as scary as it might appear. Let's start with requests.

6. HTTP Requests

A HTTP Request is plain text, and looks like this:

```
<METHOD>  <PATH>  <PROTOCOL/VERSION>
Host: <HOST>
[<HEADER>: <VALUE>]
[<HEADER>: <VALUE>]
[<HEADER>: <VALUE>] (these guys are optional)

[<REQUEST BODY>] (this is also optional).
```

To give a more concrete example, the most basic HTTP request you could send to get this webpage would look like this:

```
GET /backendbasics.html HTTP/1.1
Host: eblog.fly.dev
```

(A couple of gotchas here: the line breaks are windows-style `\r\n`, not unix-style `\n`; and the request must end with a blank line.)

Let's break this down. We can read this as

- **GET** the resource on the host `eblog.fly.dev`
- at the path `/backendbasics.html`
- using the **HTTP/1.1** protocol.

The first line is the **REQUEST LINE**. It has three parts:

- the **METHOD** (like `GET`, `POST`, `PUT`, `DELETE`, etc) tells the server what kind of request this is. For now, we only care about two: `GET` means "READ", `POST` means

"WRITE".
- the **PATH** is the path to the resource you want to access; this is the part after `.com` or `.dev` in a web address. Here, the **PATH** is `/backendbasics.html`
- the **PROTOCOL/VERSION** is the protocol and version of the request; almost always `HTTP/1.1` or `HTTP/2.0`

The **REQUEST LINE** is followed by one or more **HEADERS**.

6.1. Headers

A **HEADER** is a key-value pair, separated by a colon (`:`). The **key** should be formatted in `Title-Case`, and the **value** should be formatted in `lower-case`; for example, `Content-Type: application/json`. A few headers have official meanings in the HTTP spec, but most are just suggestions to the server about how to handle the request. Technically, headers are MIME headers, but we have enough acronyms to deal with already; we'll just call them headers for now.

The **HOST** header is required; it tells the server which domain name you're trying to access. For this article, the **HOST** header is `Host: eblog.fly.dev` Other headers are optional, and can be used to send additional information to the server. Some common headers include:

6.1.1. Common Headers

header	description	example(s)
`Accept-Encoding`	I can accept responses encoded with these encodings	`gzip, deflate`
`Accept`	the types of responses the client can accept	`text/html`
`Cache-Control`	how the client wants the server to cache the response	`no-cache`
`Content-Encoding`	my response body is encoded using:	`gzip, deflate`
`Content-Length`	my body is N bytes long	47
`Content-Type`	the type of the request body	`application/json`
`Date`	the date and time of the request	`Tue, 17 Aug 2021 23:00:00 GMT`
`Host`	the domain name of the server you're trying to access	`eblog.fly.dev`

header	description	example(s)
`User-Agent`	the name and version of the client making the request	`curl/7.64.1`,`Mozilla/5.0 (Linux; Android 8.0.0; SM-G955U Build/R16NW)`

Your browser sends a lot more headers than this: you can see them by opening the developer tools and looking at the network tab.

Here's what chrome sent when I opened this page on the devtools network tab (that is, when I sent a `GET` request to `https://eblog.fly.dev/backendbasics.html`):

```
GET / HTTP/1.1
Host: eblog.fly.dev
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Cache-Control: no-cache
Pragma: no-cache
Sec-Ch-Ua-Mobile: ?1
Sec-Ch-Ua-Platform: "Android"
Sec-Ch-Ua: "Chromium";v="116", "Not)A;Brand";v="24", "Google Chrome";v="116"
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Linux; Android 8.0.0; SM-G955U Build/R16NW) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.141 Mobile Safari/537.36
```

It's OK to have multiple headers with the same key; for example, you might have multiple `Accept-Encoding` headers, each with a different encoding. Alternatively, you can separate multiple values with a comma.
That is, these two SHOULD be equivalent:

```
Accept-Encoding: gzip
Accept-Encoding: deflate
```

and

```
Accept-Encoding: gzip, deflate
```

The server will usually pick the first one it understands. Servers are *supposed* to treat the keys as case-insensitive, but in practice this is not always the case; similarly, some web servers don't properly handle multiple headers with the same key.

6.2. URL-encoding

Note that the HTTP request separates it's sections using the following characters: `' '`, `'\r'`, `'\n'`, `:`. That means we couldn't use these characters in the request line or in headers without confusing the server; it wouldn't know if we were trying to separate a section or give it literal text.

As such, URL paths and headers can't contain these characters; we must 'escape' them using URL %-encoding before sending them to the server. URL-encoding is actually pretty simple: take any ASCII character, and turn it into it's value in hexadecimal, prefixed with a `%`. For example, the space character is `0x20` in hexadecimal, so we encode it as `%20`. The percent character itself is `0x25`, so we encode it as `%25`.

The following characters can ALWAYS be used in a URL path or header without escaping:

category	characters
lowercase ascii letters	`abcdefghijklmnopqrstuvwxyz`
uppercase ascii letters	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
digits	`0123456789`
unreserved characters	`-._~`
escaped	`%` followed by two hexadecimal digits

But some characters can only be used unescaped in certain contexts:

character set	context	note
`:/?#[]@`	path	i've never seen `[]`; `@` is for authentication
`&`	query parameter	separates query parameters
`+`	query parameter	used to encode spaces in query parameters
`=`	query parameter	separates keys from values in query parameters
`;`	path	separates path segments; rarely used
`$`	path	rarely used

Everything else must be escaped. For example, the following request path is valid:

```
1  GET /backendbasics.html HTTP/1.1
2  Host: eblog.fly.dev
```

but this one isn't:

```
1  GET /backend basics.html HTTP/1.1
2  Host: eblog.fly.dev
```

And should be encoded as:

```
1  GET /backend%20basics.html HTTP/1.1
2  Host: eblog.fly.dev
```

The `url.PathEscape` and `url.PathUnescape` functions in the standard library can be used to escape and unescape a string for use in a URL path or header; we'll cover that package in more detail in a later article.

6.3. Query Parameters

The PATH can also contain **query parameters**; these are key-value pairs in the form `key=value` that come after that path. You end the 'normal' part of the path with a `?`, and then add the query parameters, separating each with a &.

If I want to make a google search for "backend_basics", I would send the following request:

```
GET /search?q=backend_basics HTTP/1.1
Host: google.com
```

This has a single query parameter, with **KEY** `q` and VALUE `backend_basics`. I could add additional query parameters by separating them with &:

The **scryfall** API allows you to search for magic cards using a variety of query parameters: if I wanted to search for cards with the word "ice" in their name, ordered by their release date, I would send the following request:

```
GET /search?q=ice&order=released&dir=asc HTTP/1.1
```

This would have three query parameters: "q=ice", "order=released", and "dir=asc". Note that the = and & characters are not escaped in the query parameters.

That's pretty much all there is to HTTP requests. Let's try sending a `HTTP` request to `eblog.fly.dev` using TCP. The following complete program, `sendreq`, sends a HTTP request to a server at a given host, port, and path, and prints the response to stdout.

6.4. sending a HTTP request with `sendreq.go`

```
// sendreq sends a request to the specified host, port, and path, and prints the response to stdout.
// flags: -host, -port, -path, -method
package main

import (
    "bufio"
    "flag"
    "fmt"
    "log"
    "net"
    "os"
    "strings"
)

```

```
15  // define flags
16  var (
17      host, path, method string
18      port               int
19  )
20
21  func main() {
22      // initialize & parse flags
23      flag.StringVar(&method, "method", "GET", "HTTP method
    to use")
24      flag.StringVar(&host, "host", "localhost", "host to
    connect to")
25      flag.IntVar(&port, "port", 8080, "port to connect to")
26      flag.StringVar(&path, "path", "/", "path to request")
27      flag.Parse()
28
29      // ResolveTCPAddr is a slightly more convenient way of
    creating a TCPAddr.
30      // now that we know how to do it by hand using
    net.LookupIP, we can use this instead.
31      ip, err := net.ResolveTCPAddr("tcp", fmt.Sprintf("%s:
    %d", host, port))
32      if err != nil {
33          panic(err)
34      }
35
36      // dial the remote host using the TCPAddr we just
    created...
37      conn, err := net.DialTCP("tcp", nil, ip)
38      if err != nil {
39          panic(err)
40      }
41
42      log.Printf("connected to %s (@ %s)", host,
    conn.RemoteAddr())
43
44      defer conn.Close()
45
46      var reqfields = []string{
47          fmt.Sprintf("%s %s HTTP/1.1", method, path),
48          "Host: " + host,
49          "User-Agent: httpget",
50          "", // empty line to terminate the headers
51
52          // body would go here, if we had one
53      }
54      // e.g, for a request to http://eblog.fly.dev/
55      // GET / HTTP/1.1
56      // Host: eblog.fly.dev
57      // User-Agent: httpget
58      //
59
60      request := strings.Join(reqfields, "\r\n") + "\r\n" //
    note windows-style line endings
```

```
61
62          conn.Write([]byte(request))
63          log.Printf("sent request:\n%s", request)
64
65          for scanner := bufio.NewScanner(conn); scanner.Scan();
    {
66              line := scanner.Bytes()
67              if _, err := fmt.Fprintf(os.Stdout, "%s\n", line);
    err != nil {
68                  log.Printf("error writing to connection: %s",
    err)
69              }
70              if scanner.Err() != nil {
71                  log.Printf("error reading from connection:
    %s", err)
72                  return
73              }
74          }
75
76      }
```

Let's try it out on the index page of this blog (running on localhost:8080):

```
go build -o sendreq ./sendreq.go
./sendreq -host eblog.fly.dev -port 8080
```

```
2023/09/07 13:59:19 connected to localhost (@
127.0.0.1:8080)
2023/09/07 13:59:19 sent request:
```

```
GET / HTTP/1.1
Host: localhost
User-Agent: httpget
```

And we get back a response, a *redirect* to `/index.html`:

```
HTTP/1.1 308 Permanent Redirect
Content-Type: text/html; charset=utf-8
E-Req-Id: b641130b240142ae82ae8b122c35c80f
E-Trace-Id: 086e9e55-364b-4cfd-b8fe-6497214af367
Location: /index.html
Date: Thu, 07 Sep 2023 20:59:19 GMT
Content-Length: 47

<a href="/index.html">Permanent Redirect</a>.
```

We'll examine HTTP responses in the next section.

---

7. HTTP Responses

A **HTTP** response is also plain text, and looks like this:

```
<PROTOCOL/VERSION> <STATUS CODE> <STATUS MESSAGE>
[<HEADER>: <VALUE>] (these guys are optional)
[<HEADER>: <VALUE>]
[<HEADER>: <VALUE>]

[<RESPONSE BODY>] (this is optional).
```

The first line is the **STATUS LINE**. It has three parts:

- the **PROTOCOL/VERSION** is the protocol and version of the response; it should always be the same as the request.
- the **STATUS CODE** is a three-digit number that tells you whether the request succeeded or failed. The first digit tells you the general category of the response::
  - `1xx` means "informational". These are not used very often.
  - `2xx` means "success" **200 OK** and **201** created are the only ones you'll see in practice.
  - `3xx` means "redirect" **301** and **308** are the only ones you'll see in practice.
  - `4xx` means "client error"; you're probably familiar with **404 Not Found** and **403 Forbidden**, but there are a lot of others.
  - `5xx` means "server error". **500 Internal Server Error** is the only one you'll see in practice; it's the default error code for any unhandled error.
    Each status code has exactly one corresponding **STATUS MESSAGE**; for example, **200 OK** or **404 Not Found**. The status message is just a human-readable description of the status code.

Headers work the same way as in requests: they're key-value pairs separated by a colon (`:`), and the key should be formatted in `Title-Case` and the value should be formatted in `lower-case`. Headers of the response are usually 'symmetric' to the headers of the request; if you send an `Accept-Encoding: gzip` header, you'll usually get a `Content-Encoding: gzip` header back.

The final section is the **response body**. Here, it's some **HTML** that tells the browser to redirect to `/index.html`. Don't worry, I'm not going to cover HTML: this is a backend article, not a frontend article.

Let's follow the redirect and request `/index.html`:

```
./sendreq -host eblog.fly.dev -port 8080 -path /index.html
```

We get back a 200 OK response and the (very sparse) contents of the index page:

```
HTTP/1.1 200 OK
E-Req-Id: 47cf0abba4fd4629a9a926769649f653
E-Trace-Id: dc2c9528-0322-4a16-8688-8ce760fff374
Date: Thu, 07 Sep 2023 21:04:28 GMT
Content-Length: 1300
Content-Type: text/html; charset=utf-8

<!DOCTYPE html><html><head>
        <title>index.html</title>
        <meta charset="utf-8"/>
```

```
11          <link rel="stylesheet" type="text/css" href="/
    dark.css"/>
12          </head>
13          <body>
14          <h1> articles </h1>
15  <h4><a href="/
    performanceanxiety.html">performanceanxiety.html</a>
16  </h4><h4><a href="/onoff.html">onoff.html</a>
17  </h4><h4><a href="/fastdocker.html">fastdocker.html</a>
18  </h4><h4><a href="/README.html">README.html</a>
19  </h4><h4><a href="/mermaid_test.html">mermaid_test.html</a>
20  </h4><h4><a href="/quirks3.html">quirks3.html</a>
21  </h4><h4><a href="/console-autocomplete.html">console-
    autocomplete.html</a>
22  </h4><h4><a href="/console.html">console.html</a>
23  </h4><h4><a href="/cheatsheet.html">cheatsheet.html</a>
24  </h4><h4><a href="/testfast.html">testfast.html</a>
25  </h4><h4><a href="/quirks2.html">quirks2.html</a>
26  </h4><h4><a href="/bytehacking.html">bytehacking.html</a>
27  </h4><h4><a href="/
    benchmark_results.html">benchmark_results.html</a>
28  </h4><h4><a href="/index.html">index.html</a>
29  </h4><h4><a href="/noframework.html">noframework.html</a>
30  </h4><h4><a href="/faststack.html">faststack.html</a>
31  </h4><h4><a href="/backendbasics.html">backendbasics.html</
    a>
32  </h4><h4><a href="/startfast.html">startfast.html</a>
33  </h4><h4><a href="/quirks.html">quirks.html</a>
34  </h4><h4><a href="/reflect.html">reflect.html</a>
```

This is OK, but dealing with raw HTTP requests and Responses is kind of a pain. Before we dive into Go's `net/http` package, let's think about how we might implement a HTTP library ourselves.

8. Building a HTTP library

We'd like a way to write our requests and responses without having to worry about the details of the protocol, like making sure our newlines are windows-style `\r\n` instead of unix-style `\n` or title-casing our headers.

That is, we'll need four things, roughly in order of difficulty (easiest first):

- a way to represent a HTTP request or response in memory
- a way to add headers to a request or response.
- a way to serialize them as text in the HTTP format
- a way to parse them from text in the HTTP format

8.1. The Request and Response structs

Restricting ourselves for now to HTTP 1.1, we can think of a HTTP request as a struct with the following fields:

```
1
2   // Header represents a HTTP header. A HTTP header is a key-
    value pair, separated by a colon (:);
3   // the key should be formatted in Title-Case.
4   // Use Request.AddHeader() or Response.AddHeader() to add
    headers to a request or response and guarantee title-casing
    of the key.
5   type Header struct {Key, Value string}
6   // Request represents a HTTP 1.1 request.
7   type Request struct {
8       Method string // e.g, GET, POST, PUT, DELETE
9       Path string // e.g, /index.html
10      Headers []struct {Key, Value string}  // e.g, Host:
    eblog.fly.dev
11      Body string // e.g, <html><body><h1>hello, world!</
    h1></body></html>
12  }
```

and a HTTP response as a struct with the following fields:

```
1   type Response struct {
2       StatusCode int // e.g, 200
3       Headers []struct {Key, Value string}  // e.g, Content-
    Type: text/html
4       Body string // e.g, <html><body><h1>hello, world!</h1></
    body></html>
5   }
```

The following functions will build a request or response for us:

```
1   func NewRequest(method, path, host, body string)
    (*Request, error) {
2       switch {
3       case method == "":
4           return nil, errors.New("missing required argument:
    method")
5       case path == "":
6           return nil, errors.New("missing required argument:
    path")
7       case !strings.HasPrefix(path, "/"):
8           return nil, errors.New("path must start with /")
9       case host == "":
10          return nil, errors.New("missing required argument:
    host")
11      default:
12          headers := make([]Header, 2)
13          headers[0] = Header{"Host", host}
14          if body != ""  {
```

```
15              headers = append(headers, Header{"Content-
    Length", fmt.Sprintf("%d", len(body))})
16          }
17          return &Request{Method: method, Path: path,
    Headers: headers, Body: body}, nil
18      }
19  }
20
21  func NewResponse(status int, body string) (*Response,
    error) {
22      switch {
23      case status < 100 || status > 599:
24          return nil, errors.New("invalid status code")
25      default:
26          if body == "" {
27              body = http.StatusText(status)
28          }
29          headers := []Header {"Content-Length",
    fmt.Sprintf("%d", len(body))}
30          return &Response{
31              StatusCode: status,
32              Headers: headers,
33              Body: body,
34          }, nil
35      }
36  }
```

8.2. Headers

We'd like to be able to add headers to a request or response without worrying about casing of the keys. We'll do this with a 'builder' method on `*Request` and `*Response`:

```
1   func (resp *Response) WithHeader(key, value string)
    *Response {
2       resp.Headers = append(resp.Headers,
    Header{AsTitle(key), value})
3       return resp
4   }
5   func (r *Request) WithHeader(key, value string) *Request {
6       r.Headers = append(r.Headers, Header{AsTitle(key),
    value})
7       return r
8   }
```

We can use these to build a request a header at a time:

```
1   req, err := NewRequest("POST", "/api/v1/users",
    "eblog.fly.dev", `{"name": "eblog", "email":
    "efron.dev@gmail.com"}`)
2   if err != nil {
3       panic(err)
4   }
```

```
5   req = req.WithHeader("Content-Type", "application/json").
6       WithHeader("Accept", "application/json").
7       WithHeader("User-Agent", "httpget")
```

But how is `AsTitle` implemented? Let's write a quick test first to make sure we understand the requirements:

```
 1  func TestTitleCaseKey(t *testing.T) {
 2      for input, want := range map[string]string{
 3          "foo-bar":      "Foo-Bar",
 4          "cONTEnt-tYPE": "Content-Type",
 5          "host":         "Host",
 6          "host-":        "Host-",
 7          "ha22-o3st":    "Ha22-O3st",
 8      } {
 9          if got := AsTitle(input); got != want {
10              t.Errorf("TitleCaseKey(%q) = %q, want %q",
    input, got, want)
11          }
12      }
13  }
```

MIME headers are assumed to be ASCII-only, so we don't need to worry about unicode here.

```
 1  // AsTitle returns the given header key as title case; e.g.
    "content-type" -> "Content-Type"
 2  // It will panic if the key is empty.
 3  func AsTitle(key string) string {
 4      /* design note --- an empty string could be considered
    'in title case',
 5      but in practice it's probably programmer error. rather
    than guess, we'll panic.
 6      */
 7      if key == "" {
 8          panic("empty header key")
 9      }
10      if isTitleCase(key) {
11          return key
12      }
13      /* ---- design note: allocation is very expensive,
    while iteration through strings is very cheap.
14      in general, better to check twice rather than allocate
    once. ----
15      */
16      return newTitleCase(key)
17  }
18
19
20
21  // newTitleCase returns the given header key as title case;
    e.g. "content-type" -> "Content-Type";
22  // it always allocates a new string.
```

```
23  func newTitleCase(key string) string {
24      var b strings.Builder
25      b.Grow(len(key))
26      for i := range key {
27
28          if i == 0 || key[i-1] == '-' {
29              b.WriteByte(upper(key[i]))
30          } else {
31              b.WriteByte(lower(key[i]))
32          }
33      }
34      return b.String()
35  }
36
37
38  // straight from K&R C, 2nd edition, page 43. some classics
    never go out of style.
39  func lower(c byte) byte {
40      /* if you're having trouble understanding this:
41          the idea is as follows: A..=Z are 65..=90, and
    a..=z are 97..=122.
42          so upper-case letters are 32 less than their lower-
    case counterparts (or 'a'-'A' == 32).
43          rather than using the 'magic' number 32, we use
    'a'-'A' to get the same result.
44      */
45      if c >= 'A' && c <= 'Z' {
46          return c + 'a' - 'A'
47      }
48      return c
49  }
50  func upper(c byte) byte {
51      if c >= 'a' && c <= 'z' {
52          return c + 'A' - 'a'
53      }
54      return c
55  }
56
57
58
59  // isTitleCase returns true if the given header key is
    already title case; i.e, it is of the form "Content-Type"
    or "Content-Length", "Some-Odd-Header", etc.
60  func isTitleCase(key string) bool {
61      // check if this is already title case.
62      for i := range key {
63          if i == 0 || key[i-1] == '-' {
64              if key[i] >= 'a' && key[i] <= 'z' {
65                  return false
66              }
67          } else if key[i] >= 'A' && key[i] <= 'Z' {
68              return false
69          }
70      }
```

```
71          return true
72      }
73
```

We run the test and it passes, so we're good to go. Compare to the actual standard library's implementation of `textproto.CanonicalMIMEHeaderKey`; ours is essentially the same but doesn't handle some corner cases and optimizations for common headers.

8.3. Writing the request to the wire

We'll implement the `io.WriterTo` interface on both of these structs so we can efficiently write them to a `net.Conn` or other `io.Writer`.

```
 1  // Write writes the Request to the given io.Writer.
 2  func (r *Request) WriteTo(w io.Writer) (n int64, err
    error) {
 3      // write & count bytes written.
 4      // using small closures like this to cut down on
    repetition
 5      // can be nice; but you sometimes pay a performance
    penalty.
 6      printf := func(format string, args ...any) error {
 7          m, err := fmt.Fprintf(w, format, args...)
 8          n += int64(m)
 9          return err
10      }
11      // remember, a HTTP request looks like this:
12      // <METHOD>  <PATH>  <PROTOCOL/VERSION>
13      // <HEADER>: <VALUE>
14      // <HEADER>: <VALUE>
15      //
16      // <REQUEST BODY>
17
18      // write the request line: like "GET /index.html HTTP/
    1.1"
19      if err := printf("%s %s HTTP/1.1\r\n", r.Method,
    r.Path); err != nil {
20          return n, err
21      }
22
23      // write the headers. we don't do anything to order
    them or combine/merge duplicate headers; this is just an
    example.
24      for _, h := range r.Headers {
25          if err := printf("%s: %s\r\n", h.Key, h.Value);
    err != nil {
26              return n, err
27          }
28      }
29      printf("\r\n") // write the empty line that separates
    the headers from the body
```

```
30      err = printf("%s\r\n", r.Body) // write the body and
    terminate with a newline
31      return n, err
32  }
```

Response has a nearly identical implementation:

```
 1  func (resp *Response) WriteTo(w io.Writer) (n int64, err
    error) {
 2      printf := func(format string, args ...any) error {
 3          m, err := fmt.Fprintf(w, format, args...)
 4          n += int64(m)
 5          return err
 6      }
 7      if err := printf("HTTP/1.1 %d %s\r\n",
    resp.StatusCode, http.StatusText(resp.StatusCode)); err !=
    nil {
 8          return n, err
 9      }
10      for _, h := range resp.Headers {
11          if err := printf("%s: %s\r\n", h.Key, h.Value);
    err != nil {
12              return n, err
13          }
14
15      }
16      if err := printf("\r\n%s\r\n", resp.Body); err != nil {
17          return n, err
18      }
19      return n, nil
20  }
```

8.3.1. Sidenote: Go's standard interfaces

Go has a number of standard interfaces that are used throughout the standard library. You've probably already seen `io.Reader` and `io.Writer`, but there are a lot more. Many functions in the standard library work better with types that implement these interfaces; for example, `io.Copy` will copy from an `io.Reader` to an `io.Writer`, but if the `src` implements `io.WriterTo` or the `dst` implements `io.ReaderFrom`, it will use those methods instead, which can be more efficient.

Similarly, `fmt.Stringer` is used to get a string representation of a type, and `encoding.TextMarshaler` is used to get a byte slice representation of a type in order to serialize it out across the network or to disk.

We'll implement both of those interfaces on our `Request` and `Response` types for convenience and to make our tests easier to write.

All we need to do is call `WriteTo` and return the result:

```
```

```
var _, _ fmt.Stringer = (*Request)(nil), (*Response)(nil) // compile-time check that Request and Response implement fmt.Stringer
var _, _ encoding.TextMarshaler = (*Request)(nil), (*Response)(nil)
func (r *Request) String() string { b := new(strings.Builder); r.WriteTo(b); return b.String() }
func (resp *Response) String() string { b := new(strings.Builder); resp.WriteTo(b); return b.String() }
func (r *Request) MarshalText() ([]byte, error) { b := new(bytes.Buffer); r.WriteTo(b); return b.Bytes(), nil }
func (resp *Response) MarshalText() ([]byte, error) { b := new(bytes.Buffer); resp.WriteTo(b); return b.Bytes(), nil }
```

8.4. Parsing HTTP Requests and Responses

One last thing: we'd like to be able to parse HTTP requests and responses from text. This is a bit more complicated than writing them, but given what we've done so far, it should be relatively straightforward.

```
// ParseRequest parses a HTTP request from the given text.
func ParseRequest(raw string) (r Request, err error) {
    // request has three parts:
    // 1. Request linedd
    // 2. Headers
    // 3. Body (optional)
    lines := splitLines(raw)

    log.Println(lines)
    if len(lines) < 3 {
        return Request{}, fmt.Errorf("malformed request: should have at least 3 lines")
    }
    // The first line is special.
    first := strings.Fields(lines[0])
    r.Method, r.Path = first[0], first[1]
    if !strings.HasPrefix(r.Path, "/") {
        return Request{}, fmt.Errorf("malformed request: path should start with /")
    }
    if !strings.Contains(first[2], "HTTP") {
        return Request{}, fmt.Errorf("malformed request: first line should contain HTTP version")
    }
    var foundhost bool
    var bodyStart int
    // then we have headers, up until an empty line.
    for i := 1; i < len(lines); i++ {
        if lines[i] == "" { // empty line
            bodyStart = i + 1
            break
        }
```

```
30          key, val, ok := strings.Cut(lines[i], ": ")
31          if !ok {
32              return Request{}, fmt.Errorf("malformed
    request: header %q should be of form 'key: value'",
    lines[i])
33          }
34          if key == "Host"
    { // special case: host header is required.
35              foundhost = true
36          }
37          key = AsTitle(key)
38
39          r.Headers = append(r.Headers, Header{key, val})
40      }
41      end := len(lines) - 1 // recombine the body using
    normal newlines; skip the last empty line.
42      r.Body = strings.Join(lines[bodyStart:end], "\r\n")
43      if !foundhost {
44          return Request{}, fmt.Errorf("malformed request:
    missing Host header")
45      }
46      return r, nil
47  }
48
49
50  // ParseResponse parses the given HTTP/1.1 response string
    into the Response. It returns an error if the Response is
    invalid,
51  // - not a valid integer
52  // - invalid status code
53  // - missing status text
54  // - invalid headers
55  // it doesn't properly handle multi-line headers, headers
    with multiple values, or html-encoding, etc.
56  func ParseResponse(raw string) (resp *Response, err
    error) {
57      // response has three parts:
58      // 1. Response line
59      // 2. Headers
60      // 3. Body (optional)
61      lines := splitLines(raw)
62      log.Println(lines)
63
64      // The first line is special.
65      first := strings.SplitN(lines[0], " ", 3)
66      if !strings.Contains(first[0], "HTTP") {
67          return nil, fmt.Errorf("malformed response: first
    line should contain HTTP version")
68      }
69      resp = new(Response)
70      resp.StatusCode, err = strconv.Atoi(first[1])
71      if err != nil {
72          return nil, fmt.Errorf("malformed response:
    expected status code to be an integer, got %q", first[1])
```

```
 73      }
 74      if first[2] == "" ||
     http.StatusText(resp.StatusCode) != first[2] {
 75          log.Printf("missing or incorrect status text for
     status code %d: expected %q, but got %q",
     resp.StatusCode, http.StatusText(resp.StatusCode),
     first[2])
 76      }
 77      var bodyStart int
 78      // then we have headers, up until an empty line.
 79      for i := 1; i < len(lines); i++ {
 80          log.Println(i, lines[i])
 81          if lines[i] == "" { // empty line
 82              bodyStart = i + 1
 83              break
 84          }
 85          key, val, ok := strings.Cut(lines[i], ": ")
 86          if !ok {
 87              return nil, fmt.Errorf("malformed response:
     header %q should be of form 'key: value'", lines[i])
 88          }
 89          key = AsTitle(key)
 90          resp.Headers = append(resp.Headers, Header{key,
     val})
 91      }
 92      resp.Body =
     strings.TrimSpace(strings.Join(lines[bodyStart:],
     "\r\n")) // recombine the body using normal newlines.
 93      return resp, nil
 94  }
 95  // splitLines on the "\r\n" sequence; multiple separators
     in a row are NOT collapsed.
 96  func splitLines(s string) []string {
 97      if s == "" {
 98          return nil
 99      }
100      var lines []string
101      i := 0
102      for {
103          j := strings.Index(s[i:], "\r\n")
104          if j == -1 {
105              lines = append(lines, s[i:])
106              return lines
107          }
108          lines = append(lines, s[i:i+j]) // up to but not
     including the \r\n
109          i += j + 2 // skip the \r\n
110      }
111  }
```

As before, let's write a few quick tests to make sure we understand the requirements.

I'm omitting the error cases for brevity here; this article is more than long enough already.

```
 1  func TestHTTPResponse(t *testing.T) {
 2      for name, tt := range map[string]struct {
 3          input string
 4          want  *Response
 5      }{
 6          "200 OK (no body)": {
 7              input: "HTTP/1.1 200 OK\r\nContent-Length:
    0\r\n\r\n",
 8              want: &Response{
 9                  StatusCode: 200,
10                  Headers: []Header{
11                      {"Content-Length", "0"},
12                  },
13              },
14          },
15          "404 Not Found (w/ body)": {
16              input: "HTTP/1.1 404 Not Found\r\nContent-
    Length: 11\r\n\r\nHello World\r\n",
17              want: &Response{
18                  StatusCode: 404,
19                  Headers: []Header{
20                      {"Content-Length", "11"},
21                  },
22                  Body: "Hello World",
23              },
24          },
25      } {
26          t.Run(name, func(t *testing.T) {
27              got, err := ParseResponse(tt.input)
28              if err != nil {
29
    t.Errorf("ParseResponse(%q) returned error: %v", tt.input,
    err)
30              }
31              if !reflect.DeepEqual(got, tt.want) {
32                  t.Errorf("ParseResponse(%q) = %#+v, want
    %#+v", tt.input, got, tt.want)
33              }
34
35              if got2, err := ParseResponse(got.String());
    err != nil {
36
    t.Errorf("ParseResponse(%q) returned error: %v",
    got.String(), err)
37              } else if !reflect.DeepEqual(got2, got) {
38                  t.Errorf("ParseResponse(%q) = %#+v, want
    %#+v", got.String(), got2, got)
39              }
40
41          })
```

```
42      }
43  }
44
45  func TestHTTPRequest(t *testing.T) {
46      for name, tt := range map[string]struct {
47          input string
48          want  Request
49      }{
50          "GET (no body)": {
51              input: "GET / HTTP/1.1\r\nHost:
    www.example.com\r\n\r\n",
52              want: Request{
53                  Method: "GET",
54                  Path:   "/",
55                  Headers: []Header{
56                      {"Host", "www.example.com"},
57                  },
58              },
59          },
60          "POST (w/ body)": {
61              input: "POST / HTTP/1.1\r\nHost:
    www.example.com\r\nContent-Length: 11\r\n\r\nHello
    World\r\n",
62              want: Request{
63                  Method: "POST",
64                  Path:   "/",
65                  Headers: []Header{
66                      {"Host", "www.example.com"},
67                      {"Content-Length", "11"},
68                  },
69                  Body: "Hello World",
70              },
71          },
72      } {
73          t.Run(name, func(t *testing.T) {
74              got, err := ParseRequest(tt.input)
75              if err != nil {
76                  t.Errorf("ParseRequest(%q) returned error:
    %v", tt.input, err)
77              }
78              if !reflect.DeepEqual(got, tt.want) {
79                  t.Errorf("ParseRequest(%q) = %#+v, want
    %#+v", tt.input, got, tt.want)
80              }
81              // test that the request can be written to a
    string and parsed back into the same request.
82              got2, err := ParseRequest(got.String())
83              if err != nil {
84                  t.Errorf("ParseRequest(%q) returned error:
    %v", got.String(), err)
85              }
86              if !reflect.DeepEqual(got, got2) {
87                  t.Errorf("ParseRequest(%q) = %+v, want %
    +v", got.String(), got2, got)
```

```
88                 }
89
90             })
91         }
92     }
```

We run the tests and they pass, so we're good to go. This should give you a pretty good idea of how HTTP works under the hood.

9. Conclusion

You're rarely going to directly parse HTTP, but when things go wrong it's important to know how they actually work. The relative simplicity of the protocol should raise some eyebrows when you compare it to the incredibly overengineered complexity of the modern web. In the next article, we'll start diving in to how to deal with HTTP 'the real way' and dive into the standard library's `net/http` package.

Like this article? Need help making great software, or just want to save a couple hundred thousand dollars on your cloud bill? Hire me, or bring me in to consult. Professional enquiries at
efron.dev@gmail.com or linkedin