Go's net/http 101

What I think everyone should know before running http.Server/Client in production

WHAT TO EXPECT

We will cover following topics:

- http.Server and its specific knobs to turn
- http.Client and its specific knobs to turn
- Data Encoding/Decoding
- No http/2 and no tls

We start with some basics you should be aware.

(code at https://github.com/rkuska/presentations/tree/main/http101)

BASICS: HTTP

Hypertext document protocol. Defines how messages look like.

(HTTP is what PAIN files are to sepa transfers)

curl -v output

```
*   Trying 127.0.0.1…
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /sleepyget HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 27 Jul 2021 10:17:00 GMT
< Content-Length: 12
< Content-Type: text/plain; charset=utf-8
<* Connection #0 to host 127.0.0.1 left intact
all is dandy* Closing connection 0
```

(more at http://xahlee.info/linux/http_protocol.html)

BASICS: TCP AND SOCKETS

- TCP defines how messages are being transmitted
  (TCP is what EBICS is to sepa transfers)
- Sockets are just files in your system.
- Are created for every (and not only) tcp connection.
- **Limits on max fd open**
- **Limits on buffer size for each socket**
- Identifier (local:ip, local:port, foreign:ip, foreign:port, protocol)
- **You can debug them with netstat** `netstat -anvp tcp`
- **TCP handshake takes forever**

(see https://eklitzke.org/how-tcp-sockets-work for more)

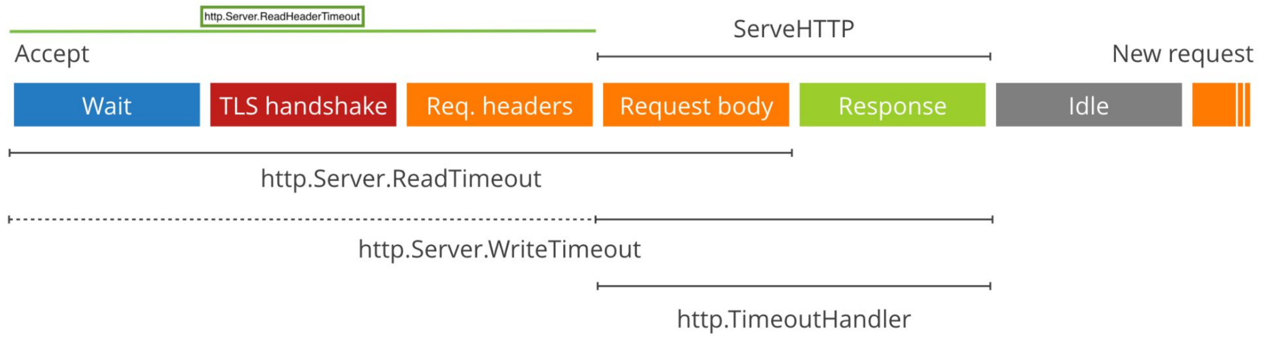
HTTP.SERVER

Stay away from http.ListenAndServe and alike.

```
func ListenAndServe(addr string, handler Handler) error {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServe()
}
```

(https://cs.opensource.google/go/go/+/refs/tags/go1.16.6:src/net/http/server.go;l=3142)

HTTP.SERVER: TIMEOUTS



- `http.Server.ReadTimeout time.Duration` - default 0
- `http.Server.ReadHeaderTimeout time.Duration` - default 0
- `http.Server.WriteTimeout time.Duration` - default 0
- `http.TimeoutHandler(h Handler, dt time.Duration, msg string) Handler`

For maximum configuration:

- `http.Server.ReadHeaderTimeout` + `http.TimeoutHandler`

(from https://blog.cloudflare.com/the-complete-guide-to-golang-net-http-timeouts/)

HTTP.SERVER: REUSE CONNECTIONS

-
  ```
  // IdleTimeout is the maximum amount of time to wait for the
  // next request when keep-alives are enabled. If IdleTimeout
  // is zero, the value of ReadTimeout is used. If both are
  // zero, there is no timeout.
  ```
  `http.Server.IdleTimeout time.Duration` - default is `http.Server.ReadTimeout`

- KeepAlive is enabled by default

HTTP.SERVER: GRACEFUL SHUTDOWN

Always try to shutdown the server. Give it to chance to finish the work.

- `http.Server.Shutdown(ctx context.Context) error`
- `http.Server.RegisterOnShutdown(f func())`

HTTP.SERVER: WRITE

You must read the body of the request before calling `WriteHeader` or `Write`.

> *`ErrBodyReadAfterClose` is returned when reading a Request or Response Body after the body has been closed. This typically happens when the body is read after an HTTP Handler calls WriteHeader or Write on its ResponseWriter.*

You should call `WriteHeader` only once.

http.Server: server logs

```
http.Server.ErrorLog ErrorLog *log.Logger
```

```
log.New(out io.Writer, prefix string, flag int) *Logger
```

```
git checkout v1.1
```

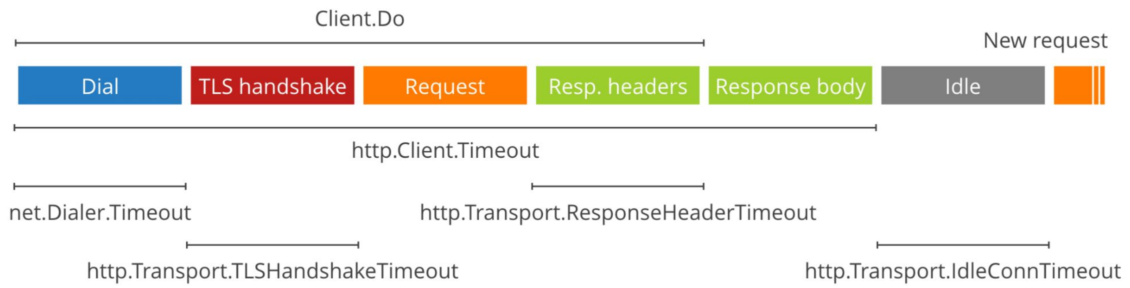
HTTP.CLIENT

Stay away from `http.Get`, `http.Post` and alike.

```
var DefaultClient = &Client{}
```

(https://cs.opensource.google/go/go/+/refs/tags/go1.16.6:src/net/http/client.go;l=109)

HTTP.CLIENT: TIMEOUTS



- `net.Dialer.Timeout time.Duration` - default 0
- `http.Transport.ResponseHeaderTimeout time.Duration` - default 0
- `http.Client.Timeout time.Duration` - (easiest) default 0

(from https://blog.cloudflare.com/the-complete-guide-to-golang-net-http-timeouts/)

HTTP.CLIENT: REUSE CONNECTIONS

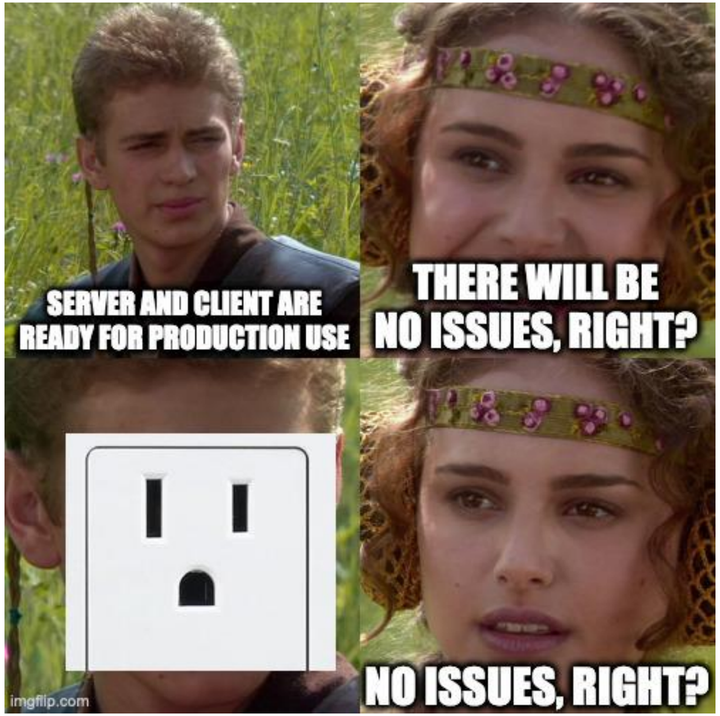
- `MaxIdleConns` (no limit)
- `MaxIdleConnsPerHost` (2)
- `MaxConnsPerHost` (no limit)

HTTP.CLIENT: QUERY STRING

Always use url.Values to construct query params (escaping).

```
url := "http://localhost:8080/endpoint?"
var params url.Values
params.Add("key", "value")
query := url + params.Encode()
```

ALL TOGETHER


SERVER AND CLIENT ARE READY FOR PRODUCTION USE
THERE WILL BE NO ISSUES, RIGHT?
NO ISSUES, RIGHT?
imgflip.com

Let's now see our super cool server and client in action.

Everything must work for sure without a problem!

`git checkout v1.1`

Marshaling/Unmarshaling: We need some closure

Let's close the body!

```
Response.Body.Close()
```

```
git checkout v1.2
```

Marshaling/Unmarshaling: We need more

We also have to read the body before closing it.

`io.Copy(ioutil.Discard, resp.Body)`

to discard or

`json.Unmarshal`

`git checkout v1.3`

Marshaling/Unmarshaling: To show you know go

Meet `json.Encoder` and `json.Decoder`

```
git checkout v1.4
```

Marshaling/Unmarshaling: There really is more

`json.Decoder.More() bool`

https://cs.opensource.google/go/go/+/refs/tags/go1.16.6:src/encoding/json/stream.go;l=157;drc=refs%2Ftags%2Fgo1.16.6

`git checkout v1.5`

QUESTIONS?