ERIK PELLIZZON

Go Developer - Freelancer

CREATE YOUR PROXY SERVER USING GO



INTRODUCTION

Create. Your. Proxy. With Go.

Self host it, it’s yours.

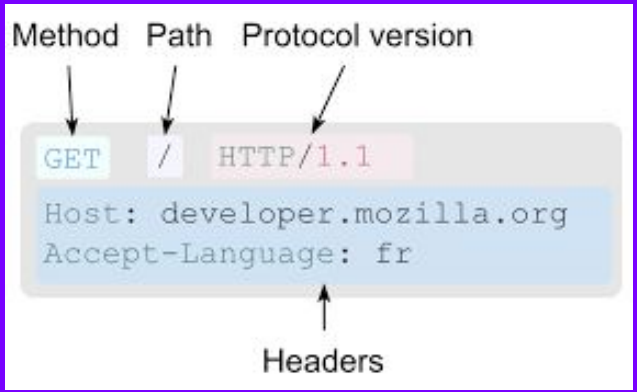


Fundamentals

Let’s start from scratch!



HTTP in 60 seconds



A request is:
METHOD path HTTP/1.1 + headers
A response is:
HTTP/1.1 STATUS + headers + body

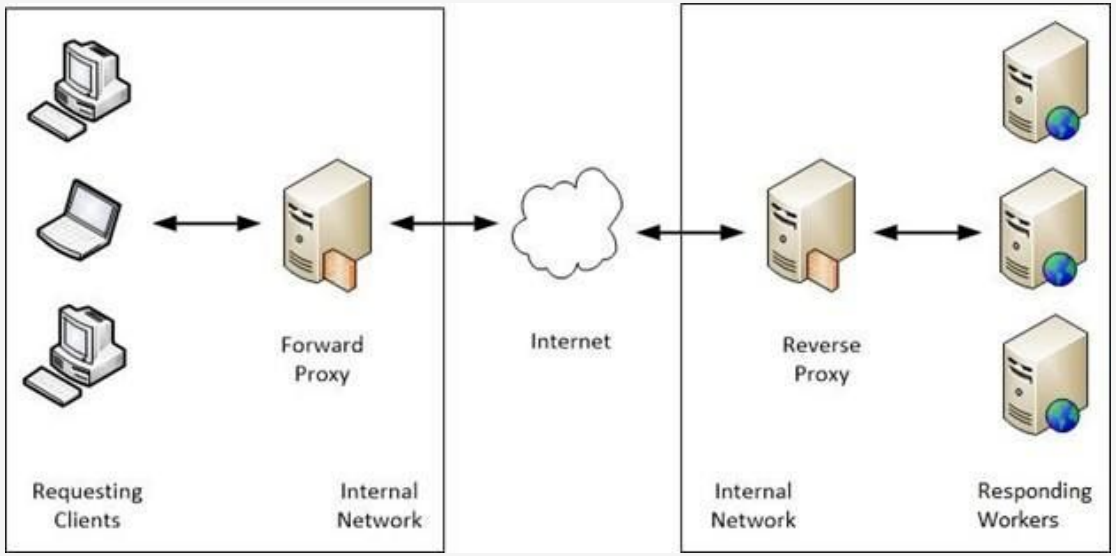
Common status codes you will see:

- 200 OK: all good
- 401 Unauthorized / 407 Proxy Authentication Required
- 429 Too Many Requests: rate limited
- 101 Switching Protocols: WebSocket upgrade

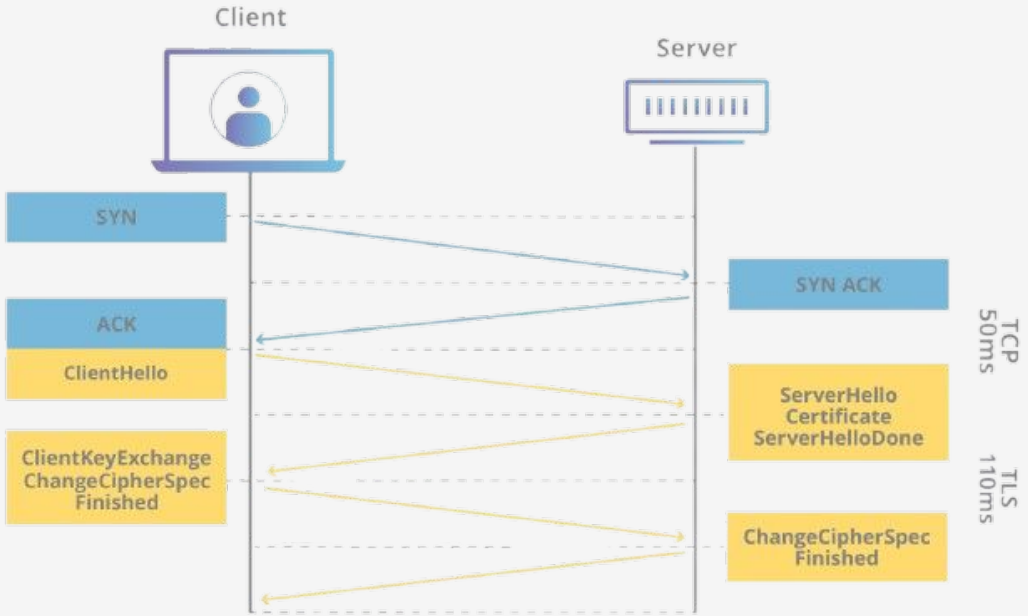
WHAT IS A PROXY?



FORWARD PROXY
vs
REVERSE PROXY



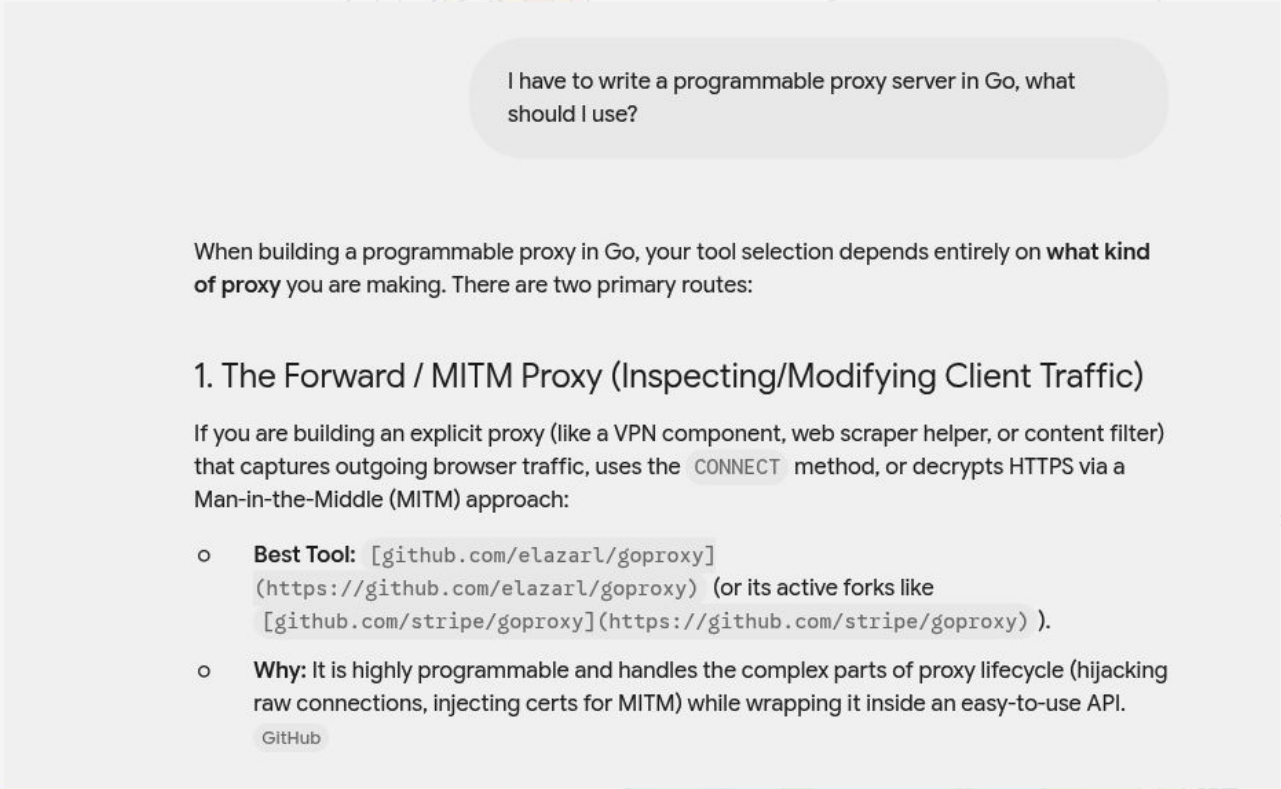
TLS


Client
Server
SYN
SYN ACK
ACK
ClientHello
ServerHello
Certificate
ServerHelloDone
ClientKeyExchange
ChangeCipherSpec
Finished
ChangeCipherSpec
Finished
TCP
50ms
TLS
110ms

IPv4 & IPv6


IPv4
Deployed 1981
32-bit IP address
Numeric dot-decimal notation
192.168.5.18
VS
IPv6
Deployed 1998
128-bit IP address
Alphanumeric hexadecimal notation
50b2:6400::6c3a:b17d:0:10a9

Let’s ask AI

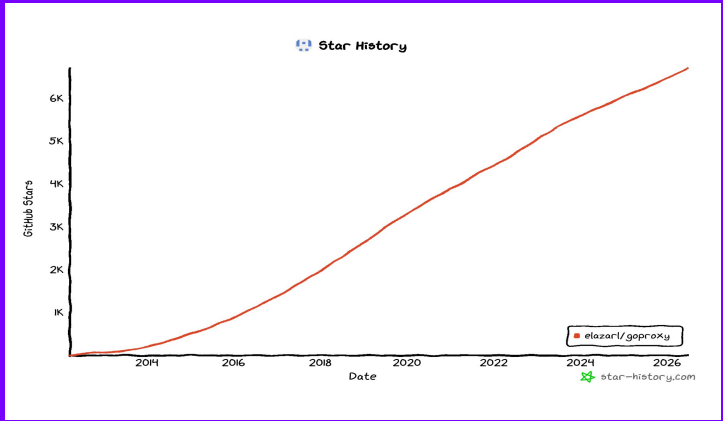

I have to write a programmable proxy server in Go, what should I use?
When building a programmable proxy in Go, your tool selection depends entirely on what kind of proxy you are making. There are two primary routes:
1. The Forward / MITM Proxy (Inspecting/Modifying Client Traffic)
If you are building an explicit proxy (like a VPN component, web scraper helper, or content filter) that captures outgoing browser traffic, uses the CONNECT method, or decrypts HTTPS via a Man-in-the-Middle (MITM) approach:
Best Tool: [github.com/elazarl/goproxy] (https://github.com/elazarl/goproxy) (or its active forks like github.com/stripe/goproxy).
Why: It is highly programmable and handles the complex parts of proxy lifecycle (hijacking raw connections, injecting certs for MITM) while wrapping it inside an easy-to-use API.
GitHub

GoProxy (Open Source)

https://github.com/elazarl/goproxy

- Library to create a customized HTTP/HTTPS forward proxy server using Go
- Full HTTP/2 support
- The proxy itself is simply a net/http handler, so you can add multiple middlewares (panic recover, logging, compression, etc.) over it
- Everything is set up by the user using only Go code, without complicated YAML configurations
- Useful to manipulate requests and responses before sending them to the browser (e.g. perform a certain action *only* for a specific host)

History of the project



- Project's first commit (by elazarl): *March 22nd, 2012*
- First **1k** stars: *2016*
- First **3k** stars: *2019*
- Stripe chose GoProxy to handle all the callbacks sent to their partners (smokescreen): *2020*
- First **5k** stars: *2023*
- Fly.io decided to use GoProxy to inject secrets: *2023*
- Dependabot (GitHub) adopted GoProxy: *2024*
- My first commit: *December 9th, 2024*
- I'm the maintainer since *2025*

GoProxy in one slide

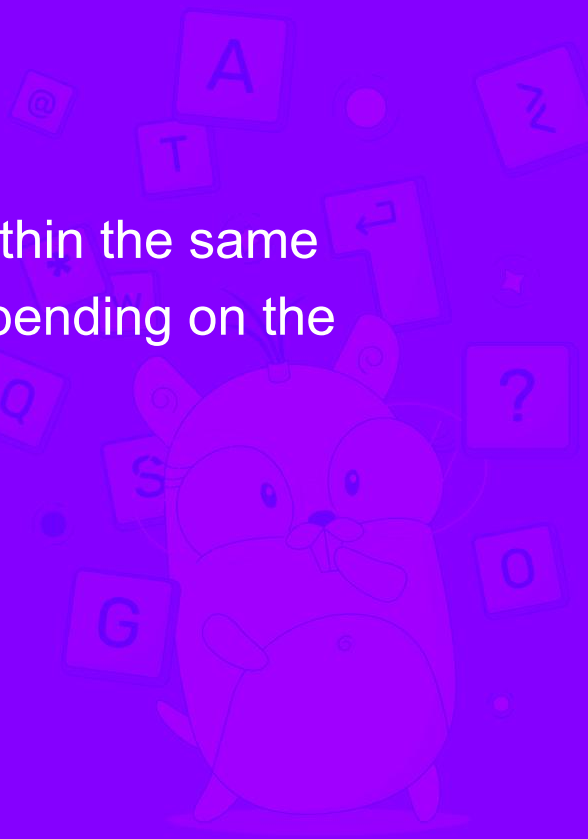
goproxy.NewProxyHttpServer(): a ready-to-use HTTP proxy

Hooks you will use:

- OnRequest(...): before forwarding the request
- OnResponse(...): on the response
- HandleConnect(...) / HijackConnect(...): deal with CONNECT
- ctx.RoundTripper: replace the upstream HTTP client
- goproxy.NewResponse(...): build a HTTP response by hand

Proxy Types

Each of these proxy types can coexist within the same proxy server when you use GoProxy, depending on the rules defined



Regular HTTP Proxy



GET / HTTP/1.1
(plaintext, can be directly modified by the proxy)

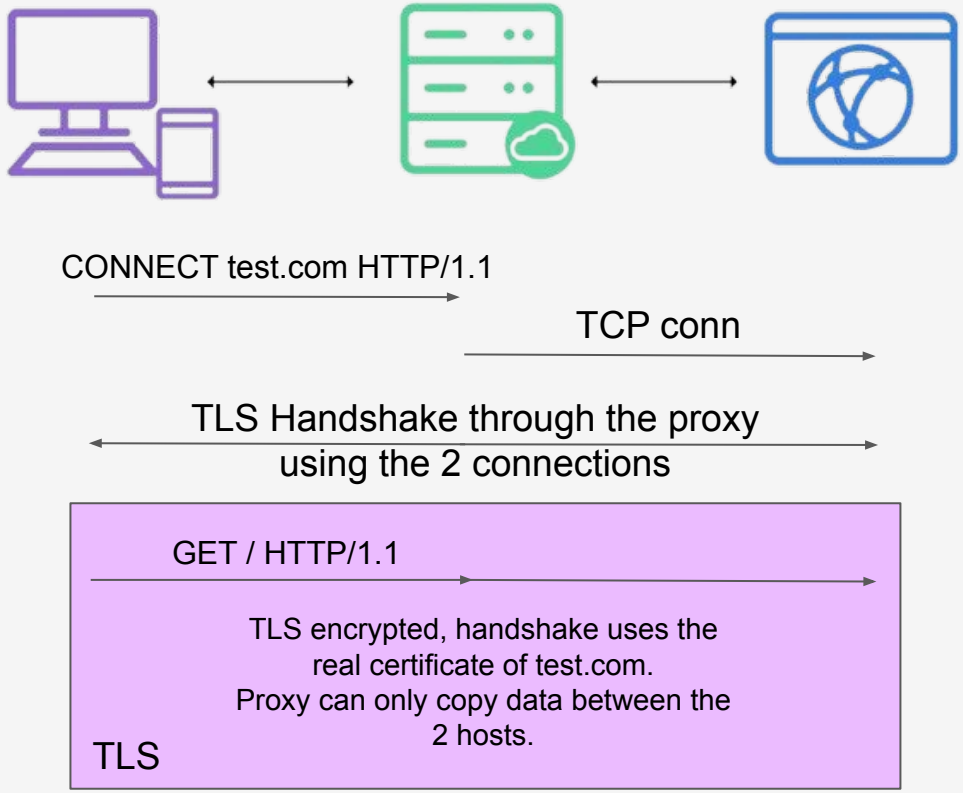
Example

```
proxy := goproxy.NewProxyHttpServer()

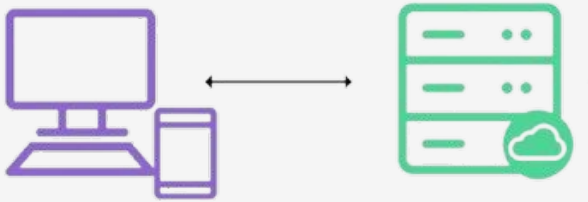
// TODO: you can add request & response handlers in the proxy
proxy.OnRequest().DoFunc(...)
proxy.OnResponse().DoFunc(...)

log.Fatal(http.ListenAndServe(":8080", proxy))
```

Regular HTTPS CONNECT Proxy



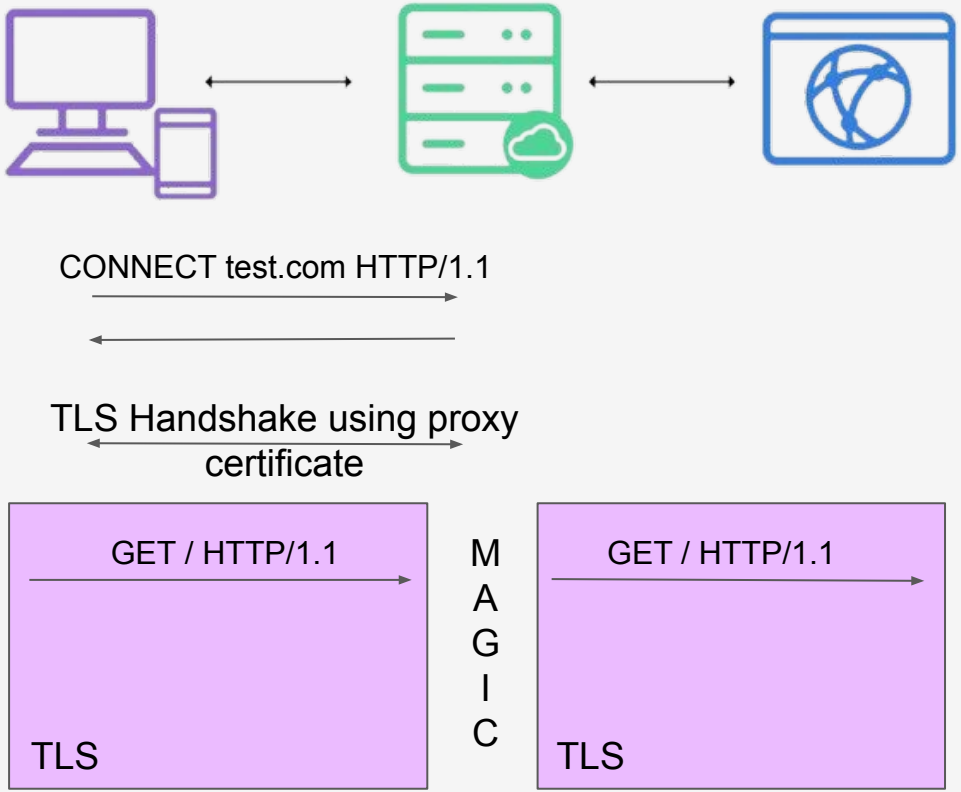
Hijack Proxy



net.Conn

The proxy can do whatever it wants with the raw net.Conn (connection with the proxy client)

HTTPS MITM Proxy



Example

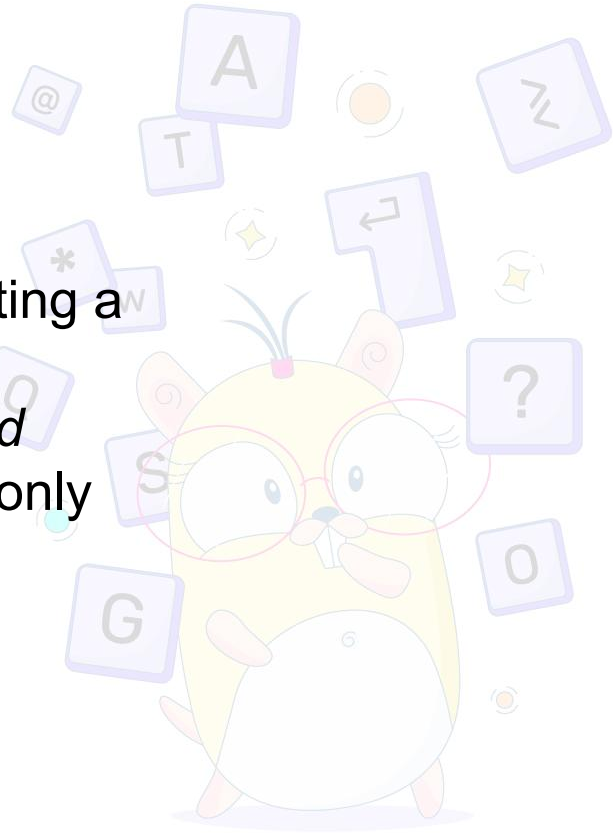
```
proxy := goproxy.NewProxyHttpServer()

proxy.OnRequest(goproxy.ReqHostIs("test.com")).HandleConnect(gopr
oxy.AlwaysMitm)
proxy.OnRequest().DoFunc(func(req *http.Request, ctx
*goproxy.ProxyCtx) (*http.Request, *http.Response) {
    req.Header.Set("Hello", "world")
    return req, nil
})

log.Fatal(http.ListenAndServe(":8080", proxy))
```

WORKSHOP

- 5 exercises, ~30 minutes each
- Every exercise = a proxy you build by completing a scaffolding
- Full instructions in each exercise *README.md*
- Solutions are available in *solutions/* (try them only when stuck!)



Exercises structure

Each folder in exercises/ contains:

- README.md: steps with links to the Go documentation
- main.go: scaffolding with TODOs to complete
- TEST.md: expected output of the curl tests

➢ Run the proxy, test with curl, compare with TEST.md

Certificates

- CA: signs certificates, the client must trust it
- Server certificate: presented by the proxy to the client (MITM)
- Client certificate: presented by the client to the proxy (mTLS)
- Generate everything with openssl (see the generation.md files) or copy the provided ones
- The client trusts a CA with curl --cacert <file>

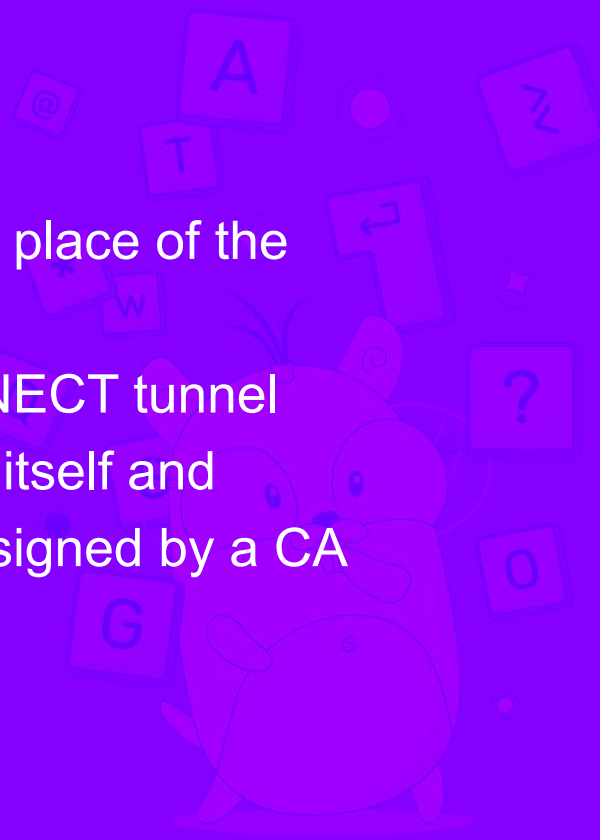
Down the rabbit hole

GoProxy Documentation: https://pkg.go.dev/github.com/elazarl/goproxy
GoProxy Source: https://github.com/elazarl/goproxy
coder/websocket: https://pkg.go.dev/github.com/coder/websocket
Workshop files: https://github.com/ErikPelli/goproxy-workshop

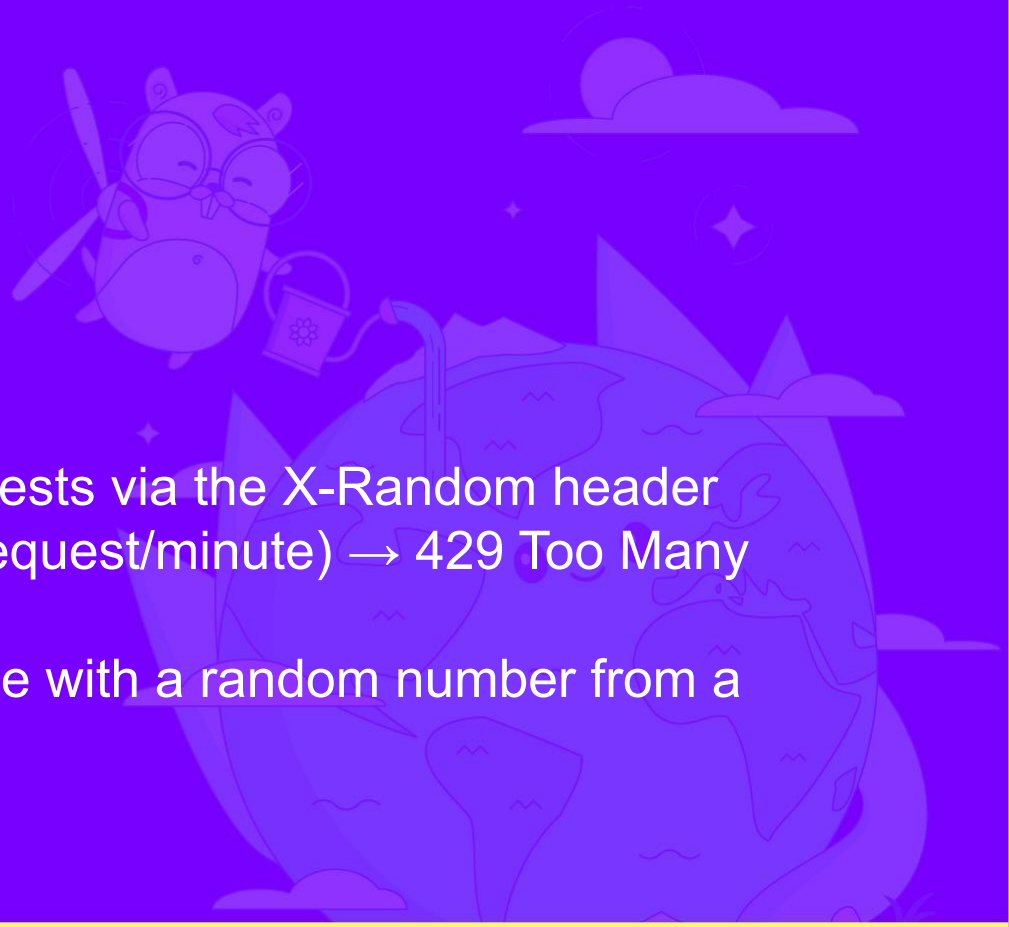
Additionally, try the **bonus** tasks in each exercise README!

Project #1 - API Proxy

- A forward proxy receives requests in place of the client
- For HTTPS the client opens a CONNECT tunnel
- MITM: the proxy terminates the TLS itself and re-encrypts, presenting a certificate signed by a CA the client trusts



What you will build



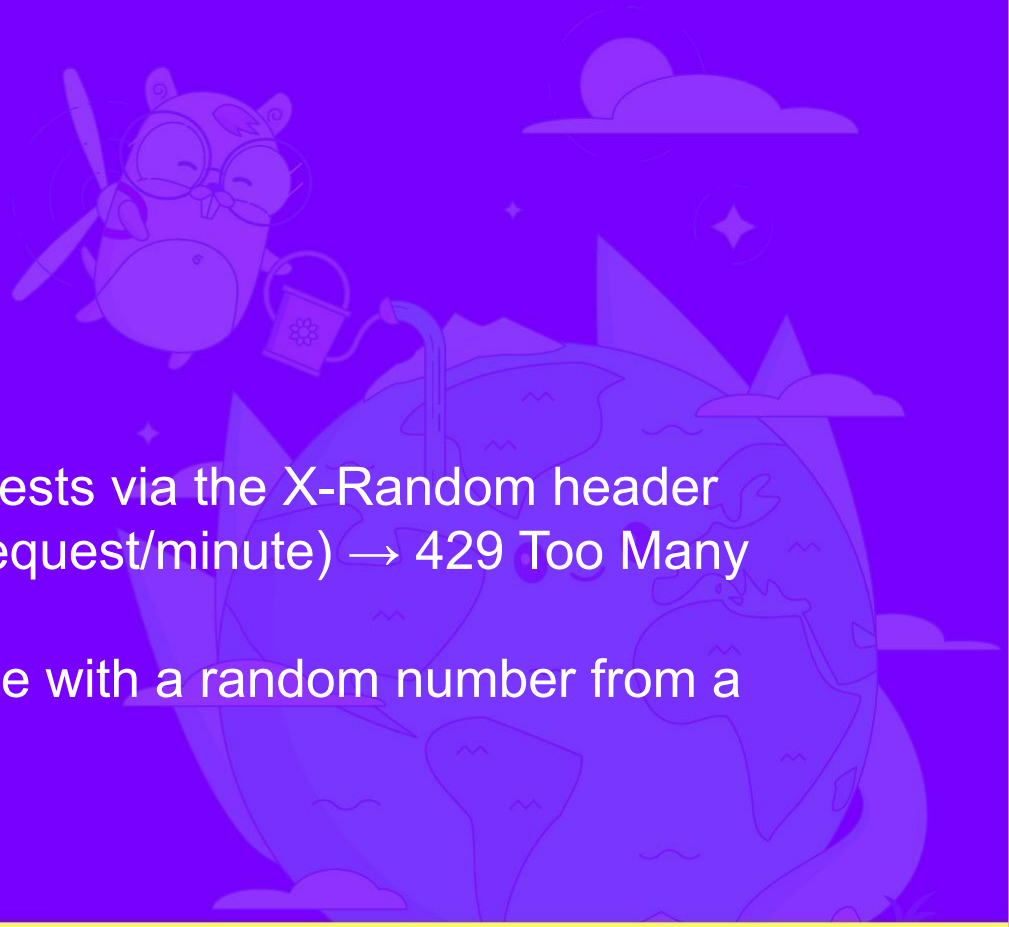
A proxy that:

- MITMs HTTPS traffic
- Recognizes API requests via the X-Random header
- Rate-limits them (1 request/minute) → 429 Too Many Requests
- Enriches the response with a random number from a backend

Flow


client
CONNECT fake-url.com:443
proxy
MITM
OnRequest(X-Random)
rate limit?
no → 429
yes → 200 "Correct"
OnResponse
fetch random from backend :11000
set header X-Random: <n>

What you will build



A proxy that:

- MITMs HTTPS traffic
- Recognizes API requests via the X-Random header
- Rate-limits them (1 request/minute) → 429 Too Many Requests
- Enriches the response with a random number from a backend

Steps 1 & 2

1. **cache.go**: implement *CertStorage.Fetch*:
   - return the cached certificate for the host, or generate + store it
   - protect the map with an *RWMutex* (concurrency!)

2. **Enable MITM**:
   proxy.OnRequest().HandleConnect(goproxy.AlwaysMitm)

   - without this, the proxy never sees the HTTPS payload

Steps 3 & 4

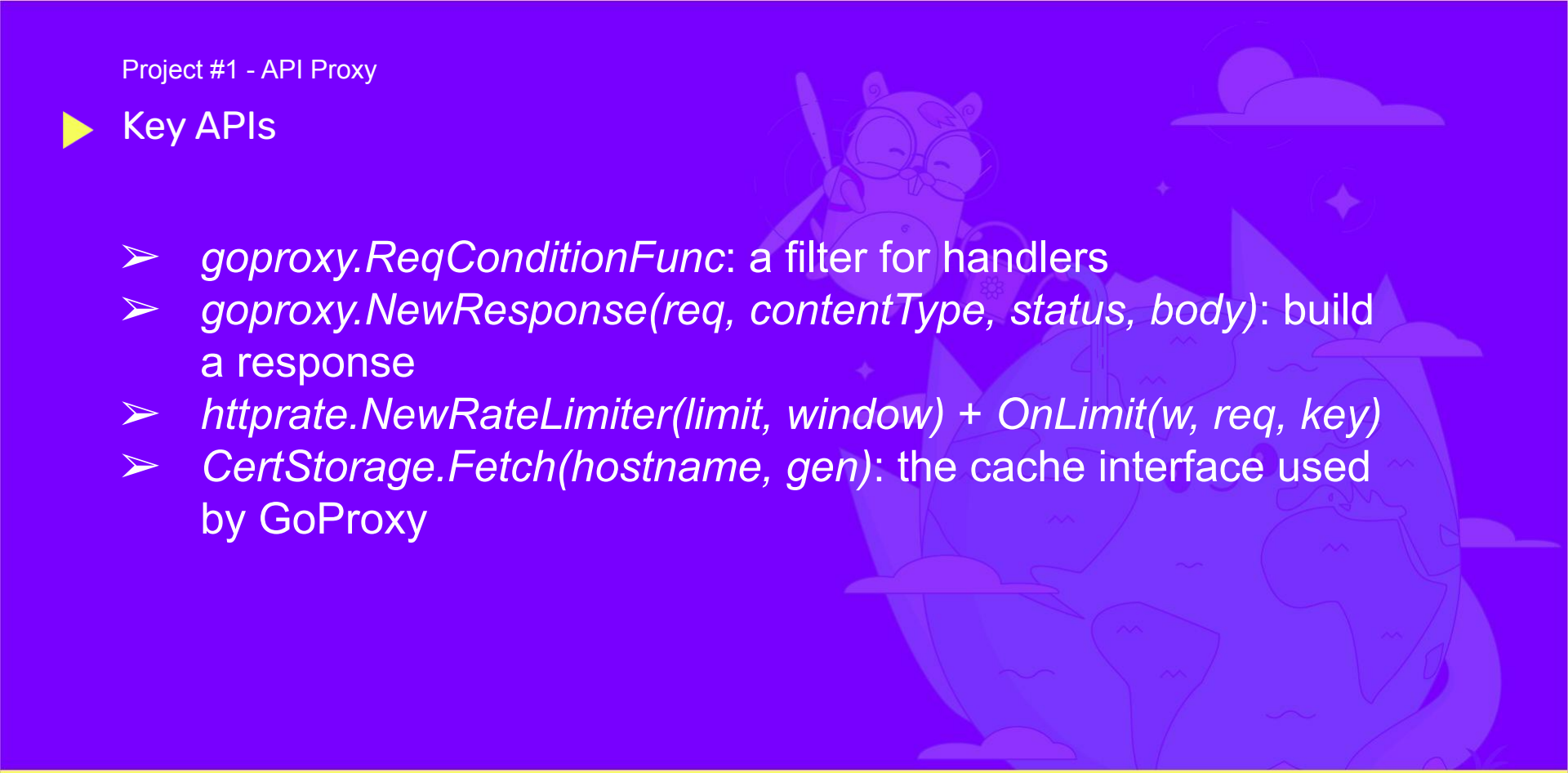
3. **Rate limiting** with *httprate*:
   rateLimiter := httprate.NewRateLimiter(1, time.Minute)

   - if *OnLimit(...)* is true → *429 Too Many Requests*
   - keyed by client IP (*strings.Cut(req.RemoteAddr, ":")*)

4. **Enrich the response** (*OnResponse*):
   - query *http://localhost:11000*, decode *{"number": N}*, set *X-Random: N*

Key APIs

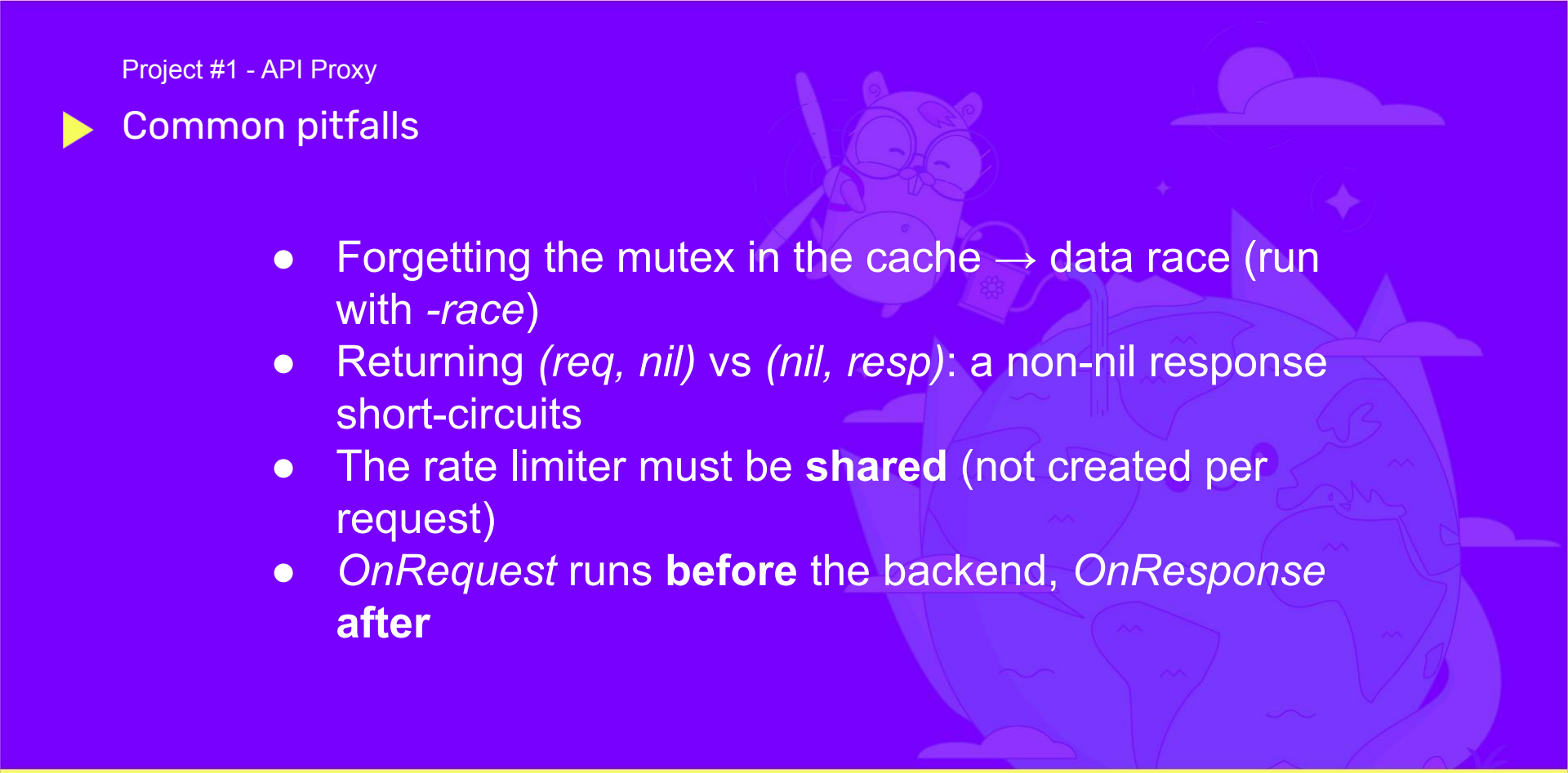
- *goproxy.ReqConditionFunc*: a filter for handlers
- *goproxy.NewResponse(req, contentType, status, body)*: build a response
- *httprate.NewRateLimiter(limit, window) + OnLimit(w, req, key)*
- *CertStorage.Fetch(hostname, gen)*: the cache interface used by GoProxy



Let’s start!

30 minutes

Common pitfalls



- Forgetting the mutex in the cache → data race (run with *-race*)
- Returning *(req, nil)* vs *(nil, resp)*: a non-nil response short-circuits
- The rate limiter must be **shared** (not created per request)
- *OnRequest* runs **before** the backend, *OnResponse* **after**

Solution

https://github.com/ErikPelli/goproxy-workshop/tree/master/solutions/01_api_proxy

Project #2 - MITM Proxy with custom CA

- A **TLS certificate** is verified against the CAs in the client trust store
- GoProxy can generate a certificate for **any hostname**, signed by a CA we control → the client accepts the proxy as the real server
- **Hijack**: take the raw **CONNECT** connection and answer manually, without reaching the real server

What you will build

One proxy, two behaviors:

Port	Behaviour
:11000	**Hijack** the CONNECT and respond with custom data
:11001	**MITM** with our CA and modify the response body

Flow

```
client ──▶ CONNECT localhost:11000 ──▶ proxy hijacks the TCP connection
                                         └─ fake TLS + "hijacked response"

client ──▶ CONNECT localhost:11001 ──▶ proxy MITMs with our CA
                                         └─ roundtrip to backend via mitmTr
                                         └─ OnResponse rewrites the body
```

Steps 1 & 2

1. **Understand the backends** in **backend.go**:
   - *:11000* panics if called → must be hijacked
   - *:11001* serves "original data" over TLS

2. **Hijack** with **HijackConnect**:
   - write ***HTTP/1.1 200 Ok\r\n\r\n*** (response to CONNECT)
   - wrap the conn in *tls.Server* with *localhostCertificate*, let's do a manual TLS handshake
   - write an **http.Response** with body ***hijacked response***, header ***test: 1234***

Steps 3 & 4

3. **Custom CA MITM**:

```
&goproxy.ConnectAction{
    Action:    goproxy.ConnectMitm,
    TLSConfig: goproxy.TLSConfigFromCA(caCertificate),
}
```

- certificates are signed by **our** CA, and the client is configured to accept them

4. **Upstream transport**:

```
ctx.RoundTripper = &roundTripper{tr: mitmTr}
```

Step 5

Surprise, read *README.md* file



Key APIs

- *goproxy.HijackConnect(handler)*: take over a CONNECT connection
- *goproxy.ConnectAction{Action, TLSConfig}*: decide what to do on CONNECT
- *goproxy.TLSConfigFromCA(ca)*: sign per-host certs with our CA
- *goproxy.ReqHostIs("host")*: filter handlers by target host
- *ctx.RoundTripper*: replace the transport used upstream

Project #2 - MITM Proxy

Let’s start!

30 minutes

Common pitfalls

- HTTP/2 in the hijack: the manual handshake only speaks HTTP/1.1 (check *req.ProtoMajor*)
- Forgetting to flush the *bufio.Writer* → the response never arrives
- The upstream transport must trust the backend cert (*mitmTr*), the client trusts a different CA
- Order of handlers: *OnRequest* → round-trip → *OnResponse*

Solution

https://github.com/ErikPelli/goproxy-workshop/tree/master/solutions/02_mitm_proxy

Project #3 - Cascade Proxy (Aggregator)

- **Chaining**: one proxy uses another proxy as its "transport"
- **Plain HTTP** → *http.Transport.Proxy*
- **CONNECT** → *proxy.ConnectDial*



What you will build

A chain of two proxies:

client → middle proxy (:10998) → edge proxy (:10999) → backend

- The **edge proxy** authenticates every request
- The **middle proxy** forwards to the edge, adding the credentials

The client only knows the middle proxy, the edge stays "hidden" and protected.

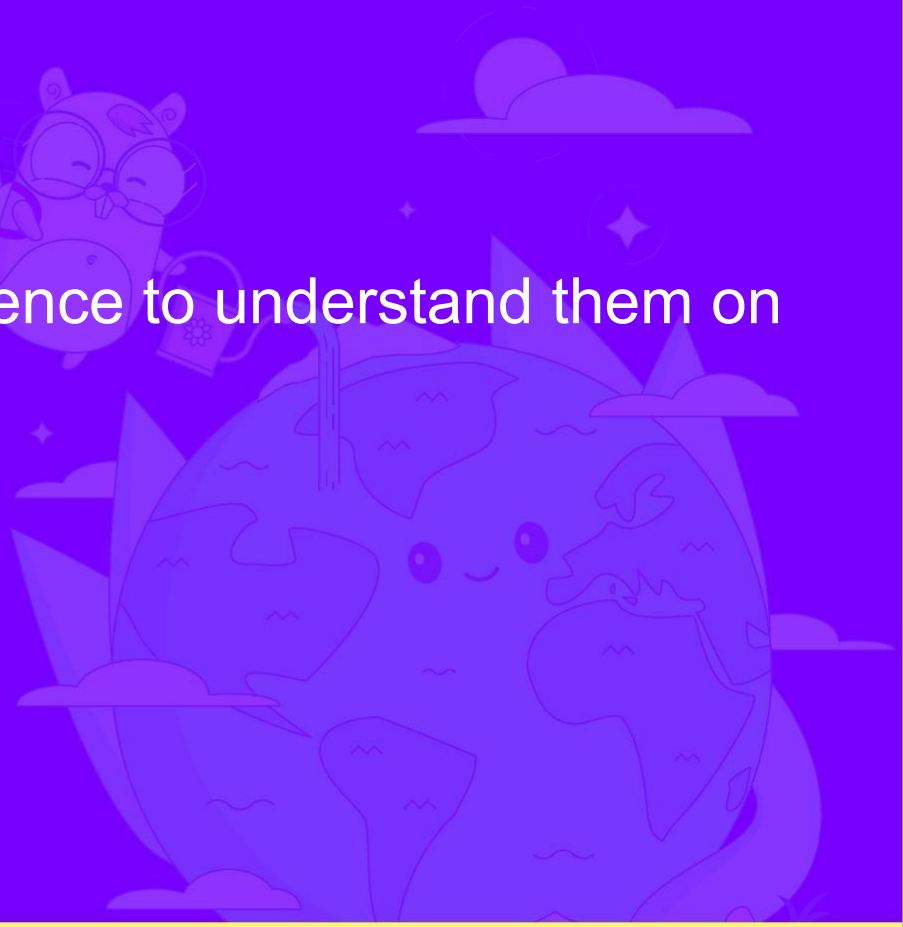
Flow

```
client
  |  GET http://localhost:11000/
  ▼
middle (:10998)
  |  + Proxy-Authorization
  ▼  Tr.Proxy → edge
edge (:10999)
  |  auth.ProxyBasic → 407 if missing/invalid
  ▼
backend (:11000) → 200 ⟶ back to client


CONNECT (https://) takes the same path via ConnectDial
```

Steps

You have enough experience to understand them on your own



Key APIs

- *auth.ProxyBasic(proxy, realm, check)*: auth on requests + CONNECT
- *http.Transport.Proxy*: route plain HTTP through an upstream proxy
- *ProxyHttpServer.ConnectDial*: route CONNECT through an upstream proxy
- *NewConnectDialToProxyWithHandler(url, handler)*: add headers to CONNECT
- *subtle.ConstantTimeCompare*: timing-safe credential comparison

Project #3 - Cascade Proxy

Let’s start!

30 minutes

Common pitfalls

- Without *KeepHeader*, the header is stripped → 407 at the edge
- *Tr.Proxy* and *ConnectDial* must **both** be set, or HTTPS won't chain
- Forgetting *setProxyAuth* on CONNECT → auth fails only for HTTPS
- If it "works" but you see no 407 anywhere in the last test, double-check the edge proxy is really in the path

Solution

https://github.com/ErikPelli/goproxy-workshop/tree/master/solutions/03_cascade_proxy

Project #4 - Callbacks Proxy (mTLS +private IPs only)

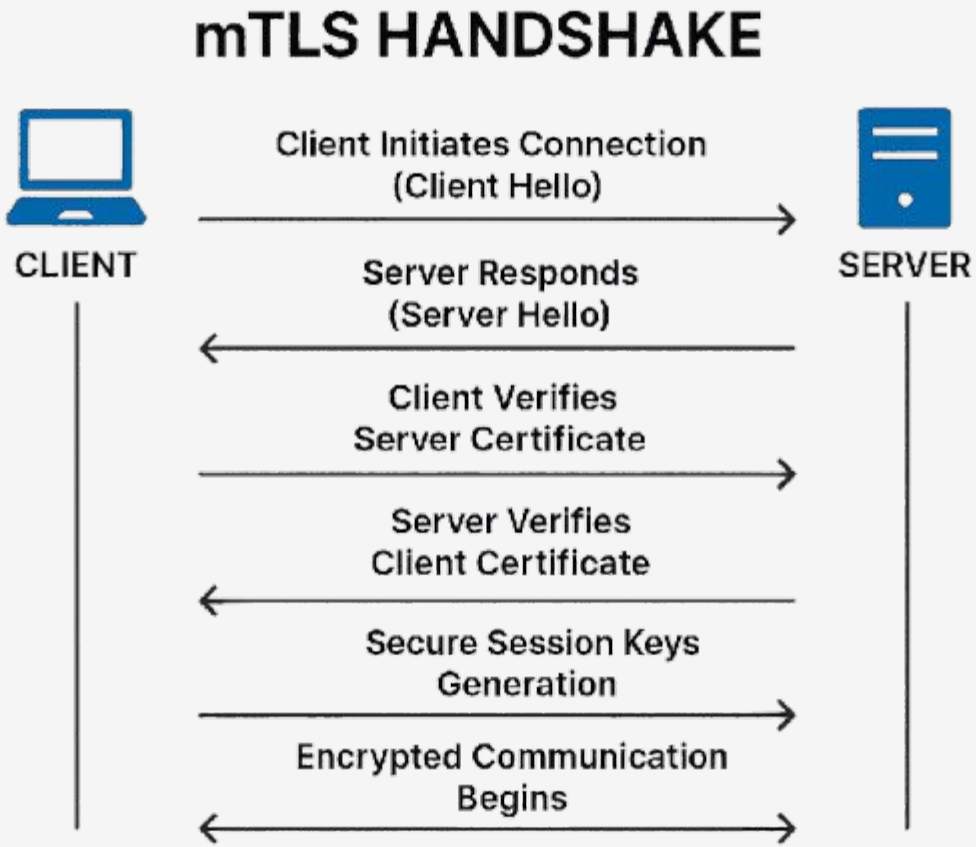
MAIN PROJECT

A proxy, called by internal services, to reach internet webhooks:

1. **Private IPs only** → otherwise *403 / RejectConnect*
2. **Basic authentication** → *Proxy-Authorization* header (user *test*, password *secret*, stored as SHA-512 hash)
3. **mTLS** → the proxy serves HTTPS and verifies the client certificate



mTLS


mTLS HANDSHAKE
CLIENT
SERVER
Client Initiates Connection
(Client Hello)
Server Responds
(Server Hello)
Client Verifies
Server Certificate
Server Verifies
Client Certificate
Secure Session Keys
Generation
Encrypted Communication
Begins

Private IPs

- 10.0.0.0/8
  (10.0.0.0 - 10.255.255.255)
- 172.16.0.0/12
  (172.16.0.0 - 172.31.255.255)
- 192.168.0.0/16
  (192.168.0.0 - 192.168.255.255)

Private IPs example (IPv4 & IPv6)

```
clientIP := "1.2.3.4"
ip := net.ParseIP(clientIP)
return ip.IsPrivate()
```

Flow

```
client (private IP) ──► HTTPS proxy :10999 (mTLS)
                          │ 1. client cert verified (CA)
                          │ 2. source IP private?
                          │ 3. Proxy-Authorization valid?
                          ▼
                        forward to backend :11000 / :11001
```

No MITM needed, all the checks are on the CONNECT request

Key APIs

- *proxy.OnRequest().DoFunc(...)*: request callback
- *proxy.OnRequest().HandleConnectFunc(...)*: CONNECT callback
- *goproxy.RejectConnect*: refuse the CONNECT
- *auth.ProxyBasic*: registers both request + CONNECT auth handlers
- *tls.RequireAndVerifyClientCert*: force mTLS
- *sha512.Sum512 / subtle.ConstantTimeCompare*: safe password check

Project #4 - Callbacks Proxy

Let’s start!

30 minutes

Common pitfalls

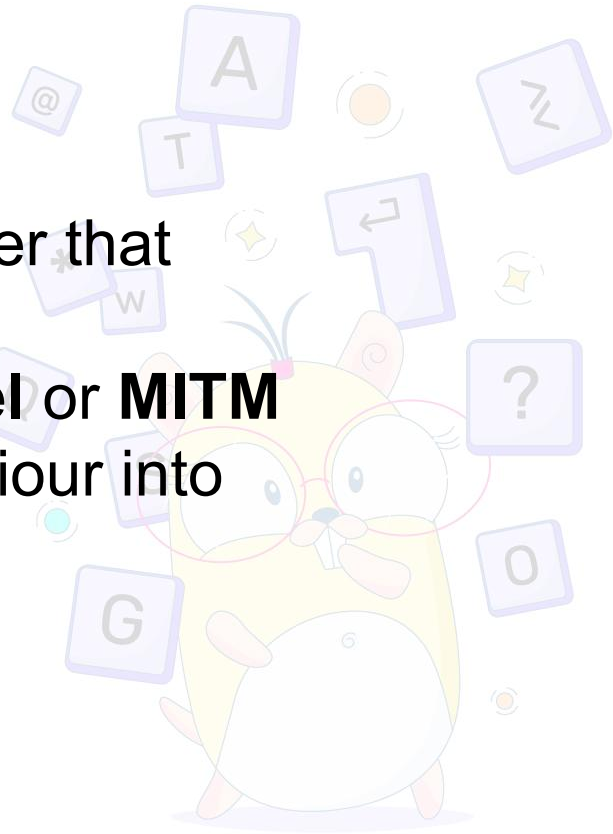
- The two IP checks are **separate**: CONNECT never goes through *OnRequest().DoFunc*
- *-race* your code: the callbacks run on many goroutines
- Don't compare hashes with ==: use constant-time compare
- The client cert must be signed by the **same CA** in *ClientCAs*

Solution

https://github.com/ErikPelli/goproxy-workshop/tree/master/solutions/04_callbacks_proxy

Key takeaways

- A proxy is "just" an HTTP server that forwards requests
- HTTPS forces a choice: **tunnel** or **MITM**
- GoProxy turns complex behaviour into small **callbacks**

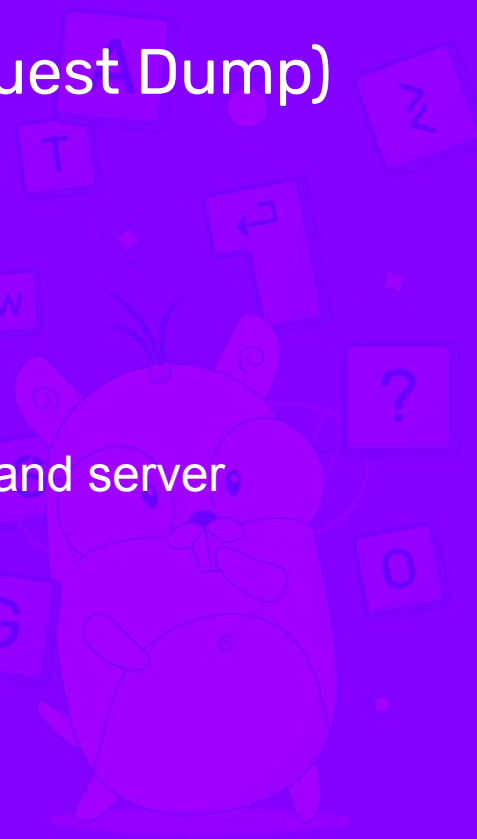


Bonus Project - WebSockets Proxy (+ Request Dump)

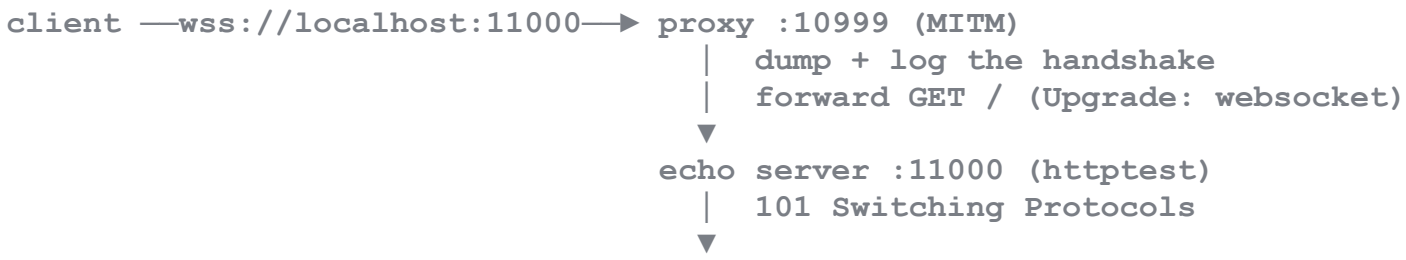
A proxy that:

1. MITMs the WebSocket connection
2. **Logs a dump of the handshake request**
3. Lets goproxy **relay the frames** between client and server

The echo server and the test client are provided (*coder/websocket*).



Flow


client ──wss://localhost:11000──▶ proxy :10999 (MITM)
dump + log the handshake
forward GET / (Upgrade: websocket)
echo server :11000 (httptest)
101 Switching Protocols
proxy relays frames in both directions (automatic)

Key APIs

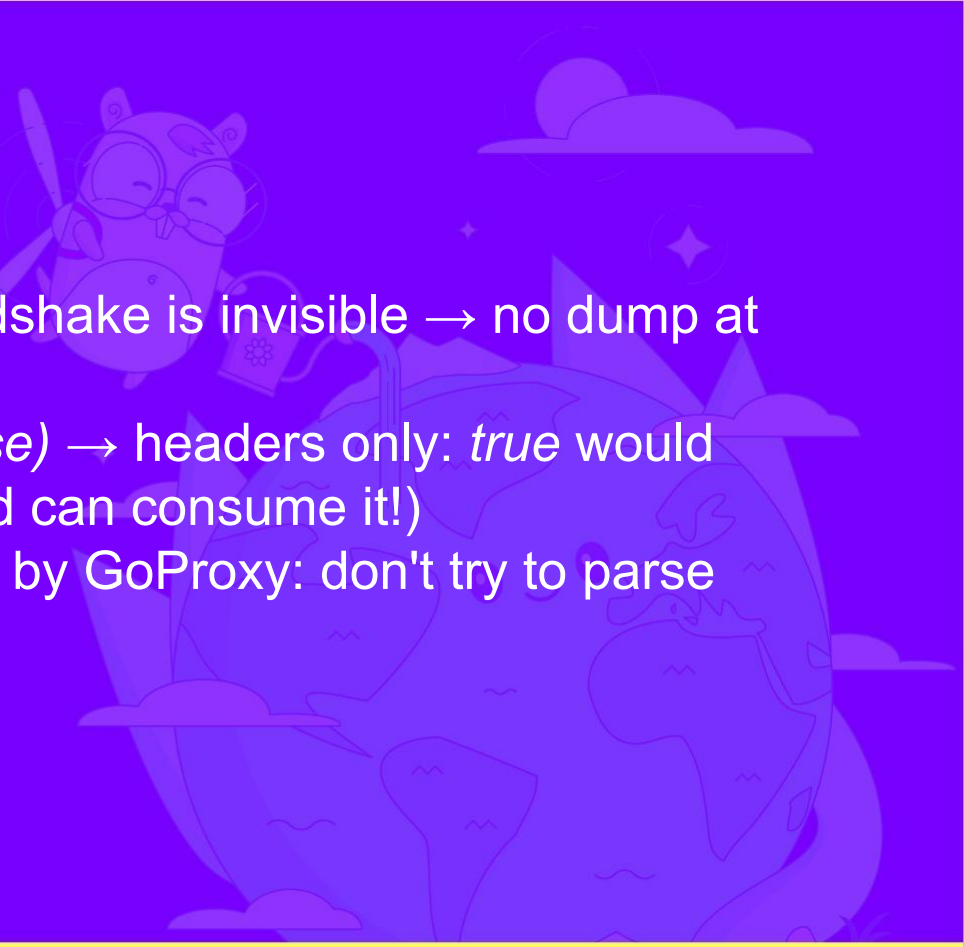
- *goproxy.AlwaysMitm*: MITM every CONNECT
- *ctx.RoundTripper*: custom transport for the upstream
- *httputil.DumpRequest(req, false)*: serialize the request (headers only)
- *websocket.Accept* / *websocket.Dial* (coder/websocket): WebSocket server & client
- *goproxy.ReqHostIs("host")*: filter by target host

Let’s start!

30 minutes

Common pitfalls

- Without MITM the handshake is invisible → no dump at all
- *DumpRequest(req, false)* → headers only: *true* would also read the body (and can consume it!)
- Frame relay is handled by GoProxy: don't try to parse frames yourself



Solution

https://github.com/ErikPelli/goproxy-workshop/tree/master/solutions/05_websocket_proxy

Bonus Project #2

HTTP/2 MITM Proxy



30 minutes. Explore. Enjoy.

Nothing else provided for this project.

Happy proxying!



Erik Pellizzon

erikpellizzon@gmail.com

LinkedIn

Freelancer

Looking for collaborations

