

Let's Build a  
Multi-Modal  
C2 Covert Channel  
in Golang!



# WELCOME !

Glad you could  
join us!



Let's get a quick overview  
of today's plan, so you know  
what to expect, and how to get  
the best value for your time.



[www.faanross.com/antisyphon/workshop02/moc/](http://www.faanross.com/antisyphon/workshop02/moc/)

## Foundation

- [Project Structure and Interfaces](#)
- [YAML-based Configuration Management System](#)

## HTTPS Implementation

- [HTTPS Server](#)
- [HTTPS Agent](#)
- [HTTPS Loop](#)

## DNS Implementation

- [DNS Server](#)
- [DNS Agent](#)
- [DNS Loop](#)

## Transition Using API Switch

- [Implement API Switch](#)
- [Dual-server Startup](#)
- [Agent Parsing + Protocol Transition](#)

→ this welcome  
→ what we'll be creating  
→ factory + interface pattern

1. Interfaces, Config, Factory Functions
2. YAML Config System
3. HTTPS Server
4. HTTPS Agent
5. RunLoop
6. DNS Server
7. DNS Agent
8. Adapt RunLoop
9. API Switch
10. Starting Both Listeners
11. Agent Parse + Callback

→ where to from here?  
→ conclusion

→ this welcome  
→ what we'll be creating  
→ factory + interface pattern

1. Interfaces, Config, Factory Functions
2. YAML Config System
3. HTTPS Server
4. HTTPS Agent
5. RunLoop
6. DNS Server
7. DNS Agent
8. Adapt RunLoop
9. API Switch
10. Starting Both Listeners
11. Agent Parse + Callback

→ where to from here?  
→ conclusion

## Foundation

1. Interfaces, Config, Factory Functions
2. YAML Config System

## HTTPS Logic

3. HTTPS Server
  4. HTTPS Agent
  5. RunLoop
- 
6. DNS Server
  7. DNS Agent
  8. Adapt RunLoop

## DNS Logic

9. API Switch
10. Starting Both Listeners
11. Agent Parse + Callback

## Switch Logic

**For each of these 11:**

- Key ideas (slides)
- Overview (slides)
- Go and build
- Questions

1. Interfaces, Config, Factory Functions
  2. YAML Config System
  3. HTTPS Server
  4. HTTPS Agent
  5. RunLoop
  6. DNS Server
  7. DNS Agent
  8. Adapt RunLoop
  9. API Switch
  10. Starting Both Listeners
  11. Agent Parse + Callback
- ← 10 MIN BREAK

**For each of these 11:**

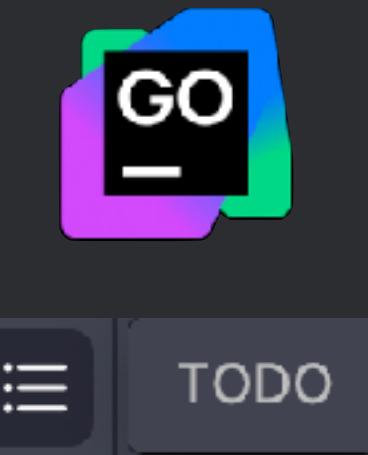
- Key ideas (slides)
- Overview (slides)
- Go and build
- Questions

1. Interfaces, Config, Factory Functions
  2. YAML Config System
  3. HTTPS Server
  4. HTTPS Agent
  5. RunLoop
  6. DNS Server
  7. DNS Agent
  8. Adapt RunLoop
  9. API Switch
  10. Starting Both Listeners
  11. Agent Parse + Callback
- ← 10 MIN BREAK

# Go and build

- Starting Solution
- Final Solution

```
> └── Lesson01_Begin  
> └── Lesson01_Done  
> └── Lesson02_Begin  
> └── Lesson02_Done  
> └── Lesson03_Begin  
> └── Lesson03_Done
```



use **ctrl/cmd + F**

```
agentCfg := config.Config{  
    // TODO Set protocol to HTTPS  
}
```

I did this to make it easy to follow along today.

Please follow along!

# Please follow along!

You only need 3 things

- Go (runtime environment)
- Your IDE of Choice
- Course Repo

opt: curl/browser + dig/nslookup

[https://www.faanross.com/antisyphon/  
workshop02/part\\_b/00\\_setup/](https://www.faanross.com/antisyphon/workshop02/part_b/00_setup/)

# Please follow along!

You only need 3 things

→ Go (runtime environment)

→ Your IDE of Choice

→ Course Repo

opt: curl/browser + dig/nslookup

<https://go.dev/dl/>

# Please follow along!

You only need 3 things

→ Go (runtime environment)

→ Your IDE of Choice

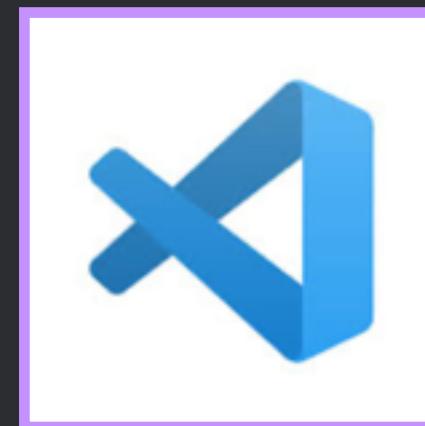
→ Course Repo

opt: curl/browser + dig/nslookup



MAX\_JETBRAINS

[www.jetbrains.com/go/download](https://www.jetbrains.com/go/download)



<https://code.visualstudio.com/>



Go by Google

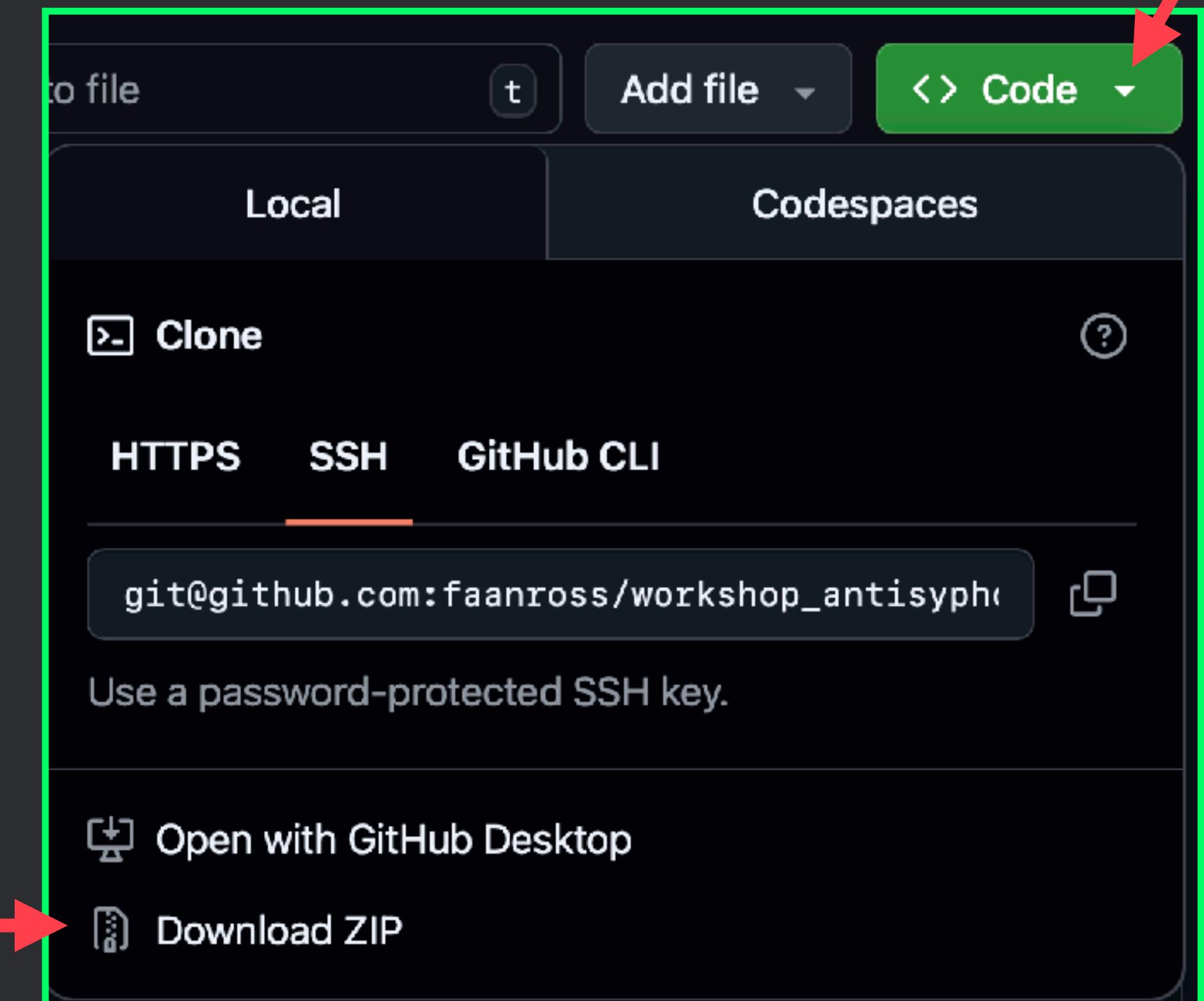
YAML by Red Hat

# Please follow along!

You only need 3 things

- Go (runtime environment)
- Your IDE of Choice
- Course Repo

opt: curl/browser + dig/nslookup



```
git clone https://github.com/faanross/  
workshop_antisiphon_18092025.git
```

[github.com/faanross/workshop\\_antisiphon\\_18092025](https://github.com/faanross/workshop_antisiphon_18092025)

# Please follow along!

You only need 3 things

- Go (runtime environment)
- Your IDE of Choice
- Course Repo

opt: curl/browser + dig/nslookup



- browser
- nslookup



- curl
- dig/nslookup

We want to hit:

- HTTPS EP
- DNS EP



- curl
- dig/nslookup

There is still time, go get it!



Dezzy



(h , k)

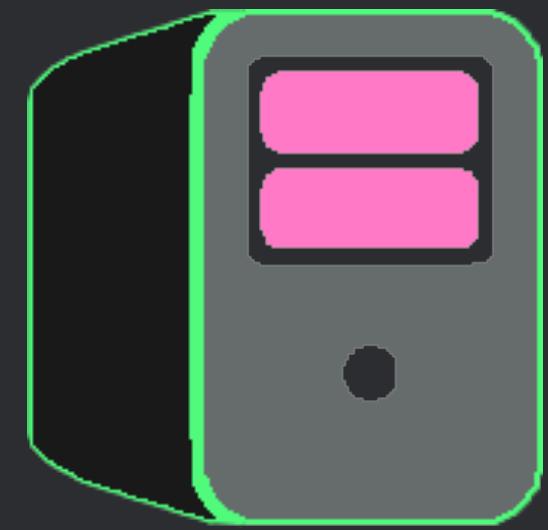
overview of what  
we'll be creating



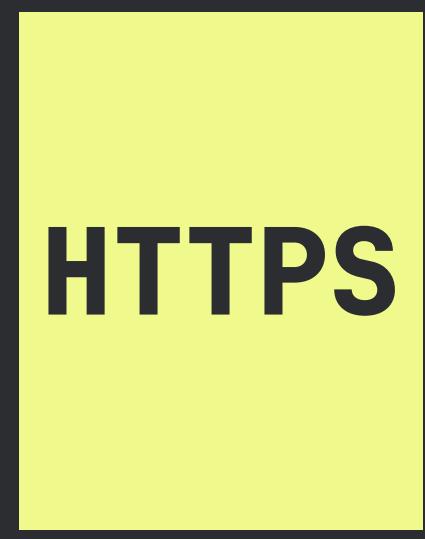
# → We'll lay a foundation

- 1. Interfaces, Config, Factory Functions
- 2. YAML Config System
  
- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop
  
- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop
  
- 9. API Switch
- 10. Starting Both Listeners
- 11. Agent Parse + Callback

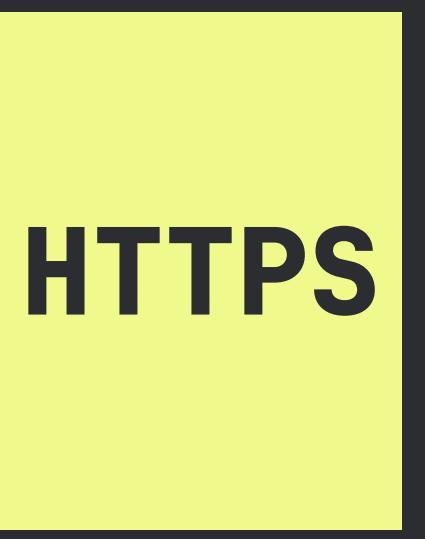
Then...



server

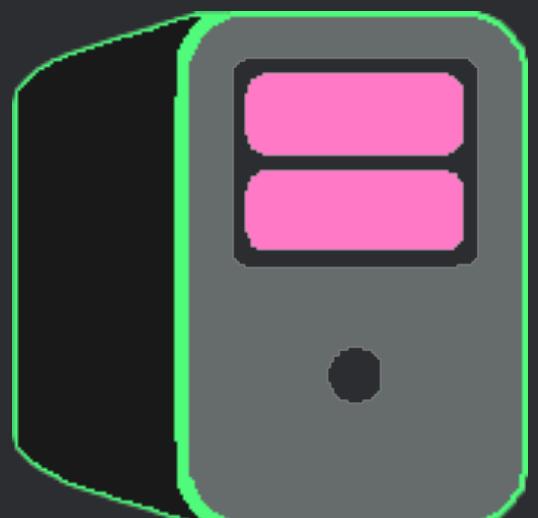


response  
request

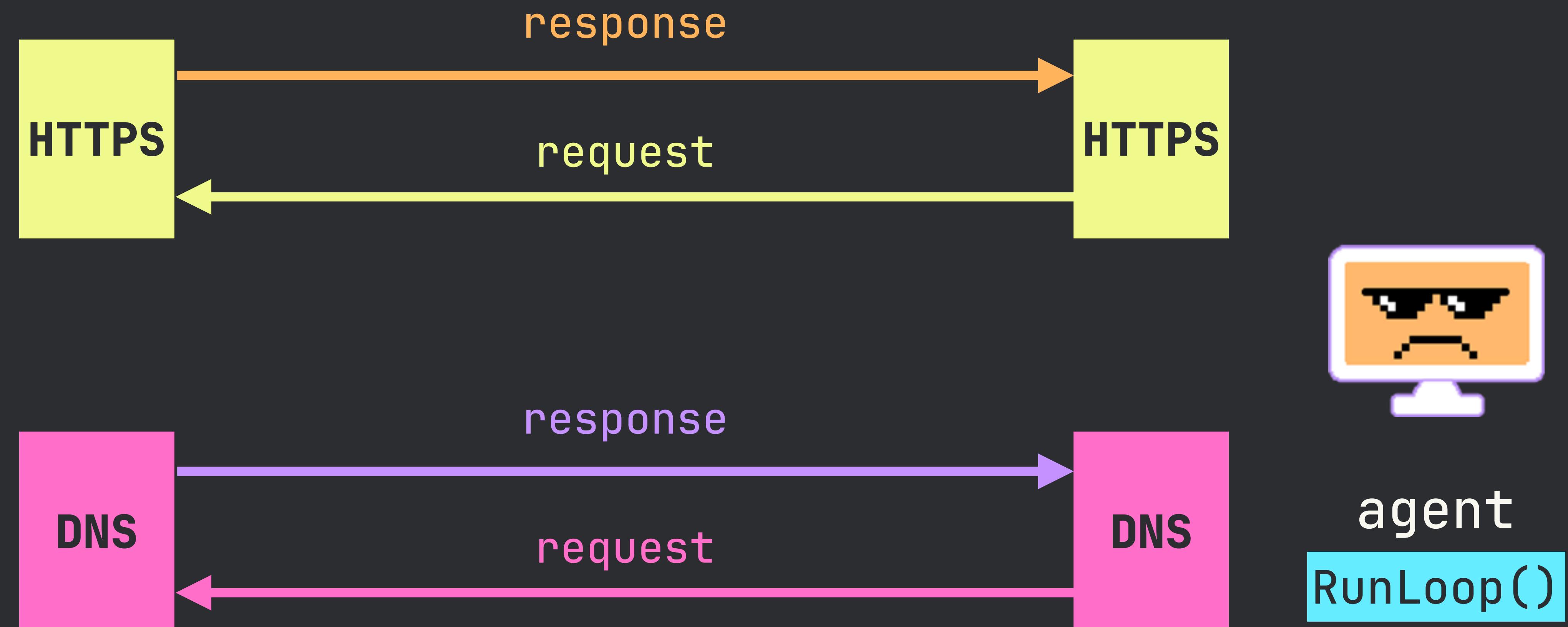


agent

RunLoop()

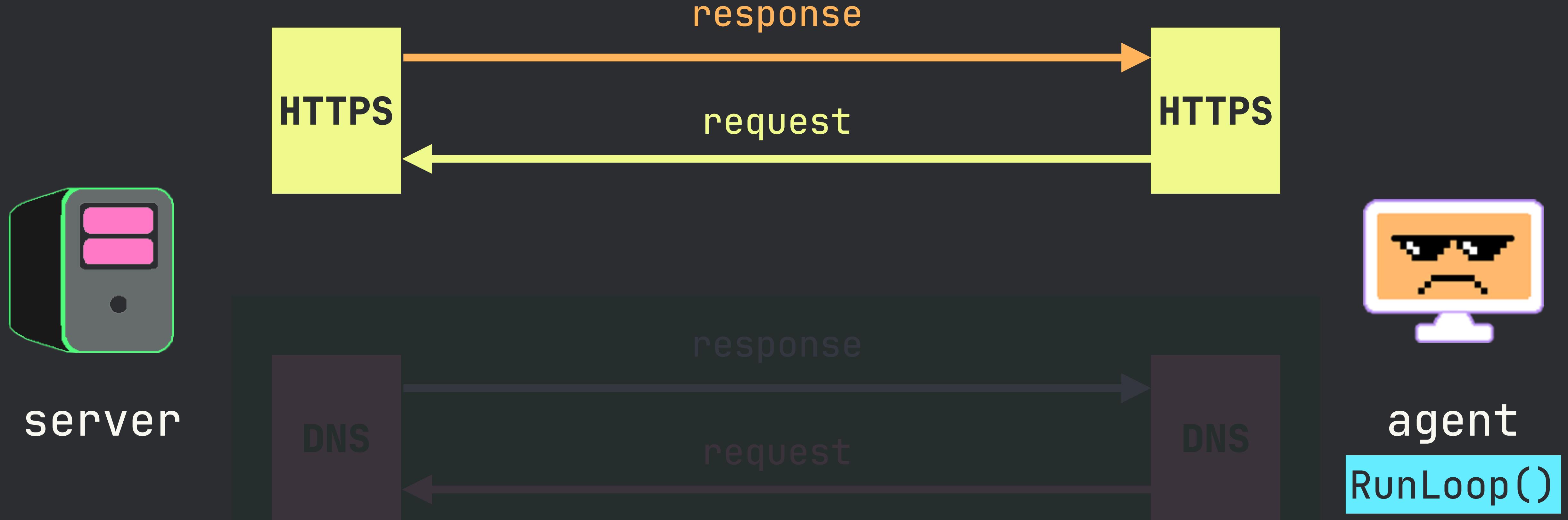


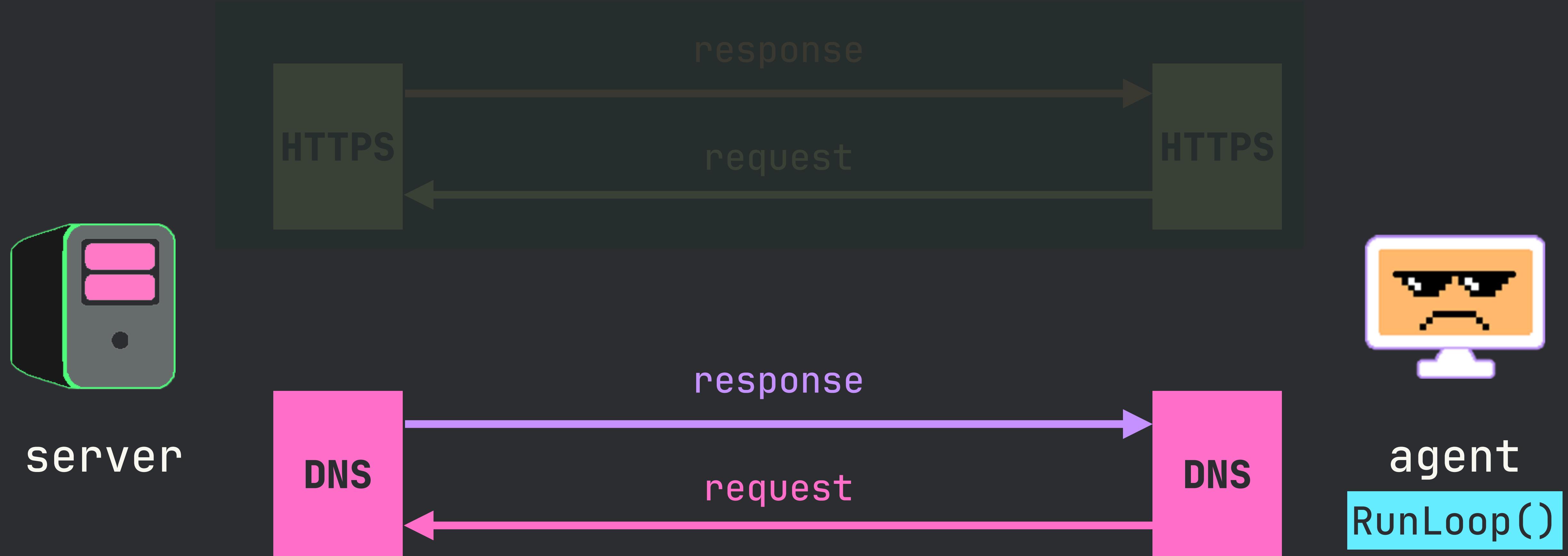
server



agent  
RunLoop()

At this point it's  
one OR the other





what we want is the ability to  
dynamically transition from one to other

## HOW?

- For each protocol we decide on pair of signals
- Signal is from server to agent (response)
- Communicates: Don't change, or Change

SIGNAL 1 → Don't Change

SIGNAL 2 → Change

**HTTPS**

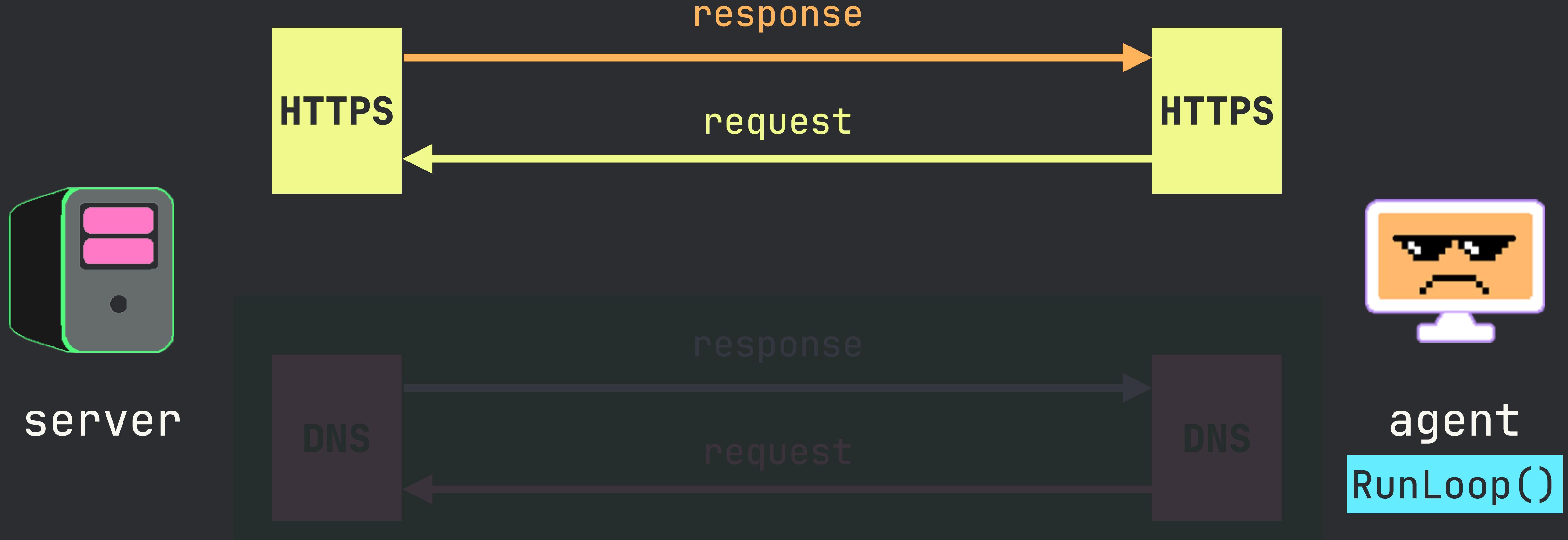
SIGNAL 1 → JSON{ “Change”: false }

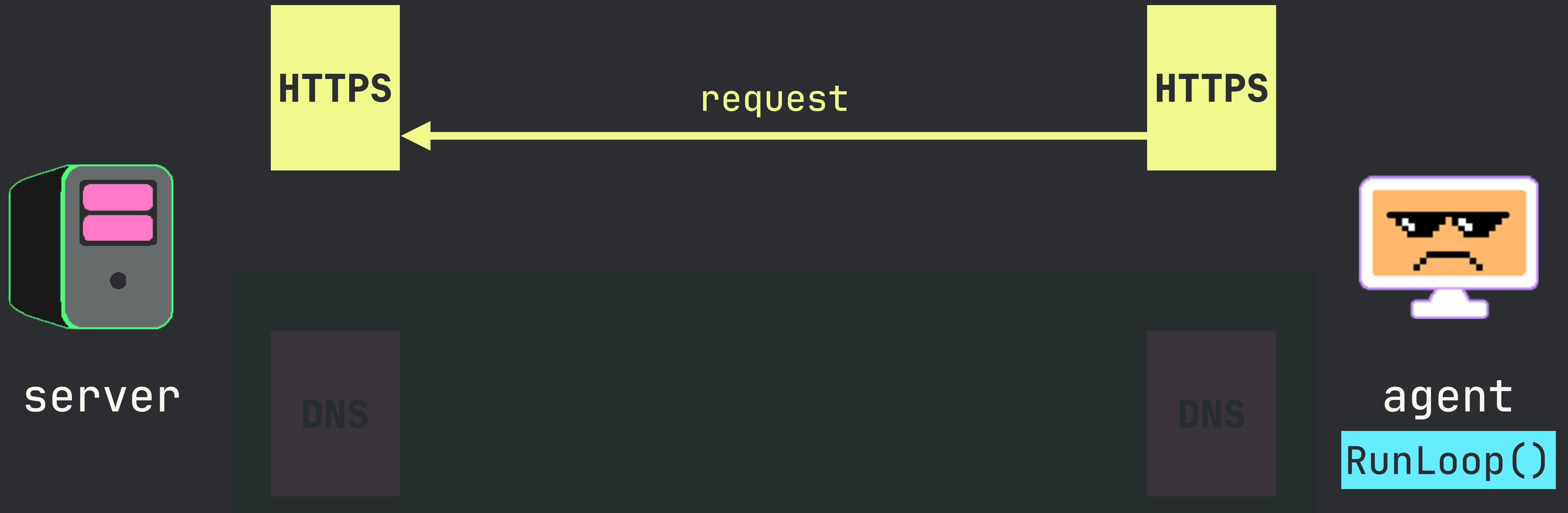
SIGNAL 2 → JSON{ “Change”: true }

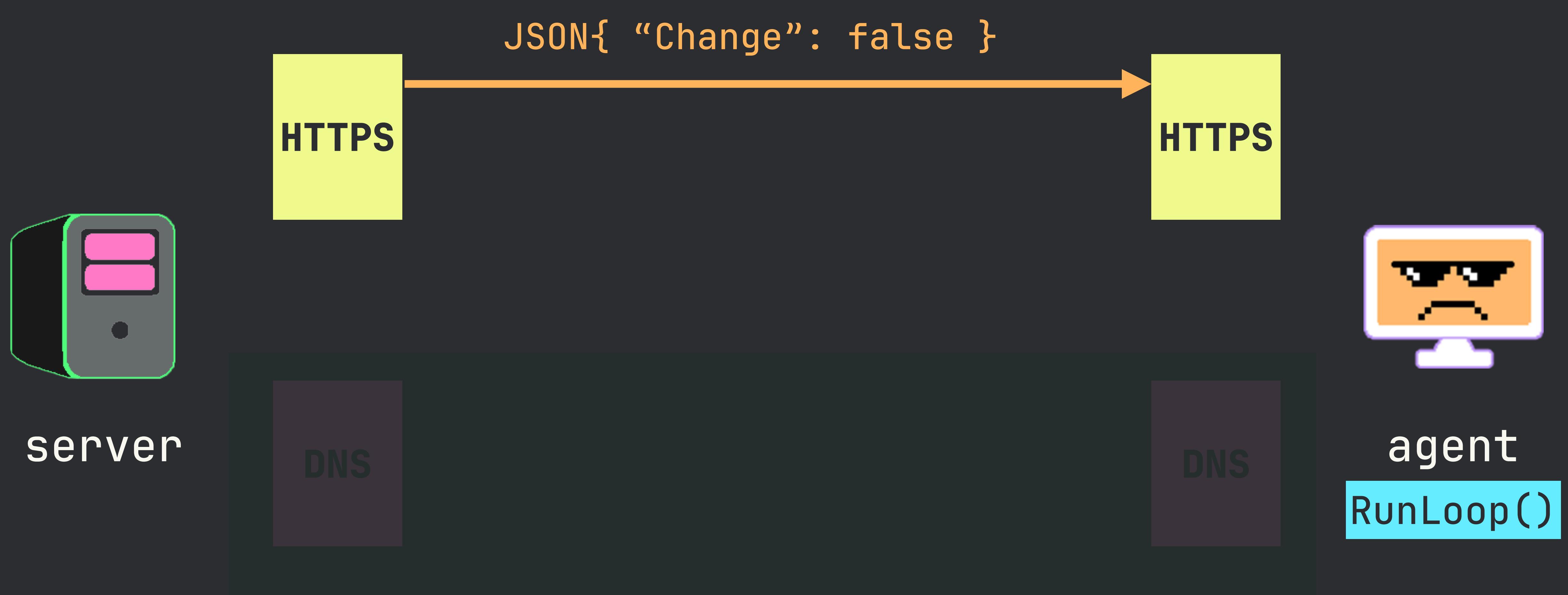
**DNS**

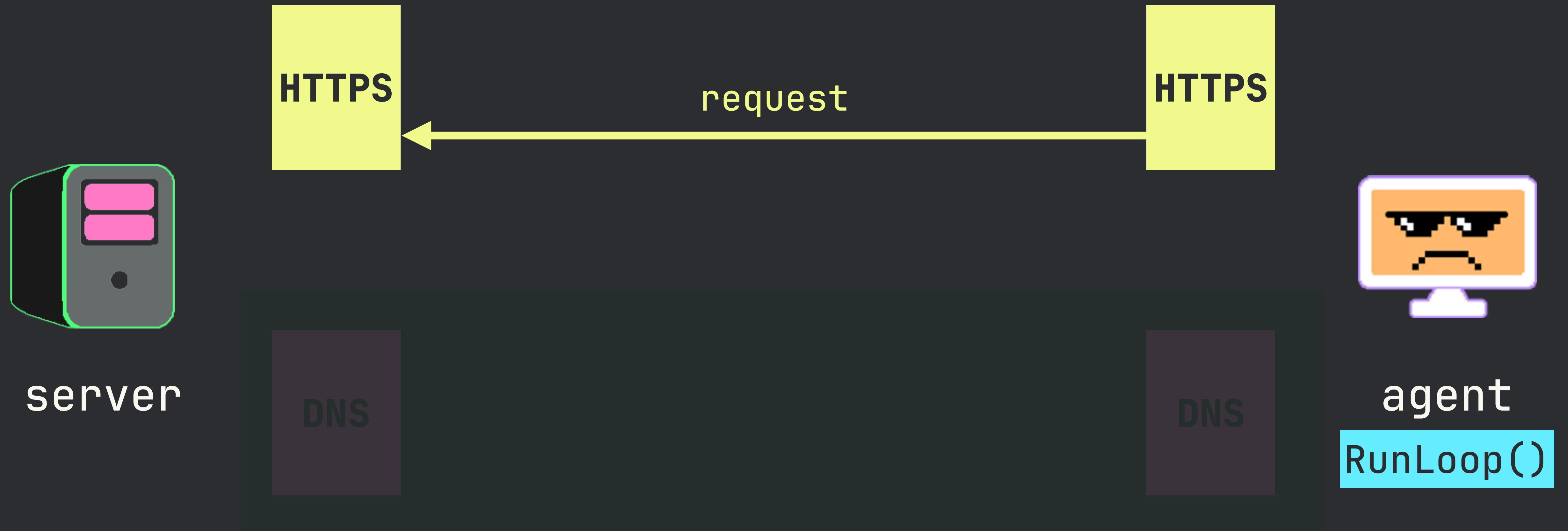
SIGNAL 1 → A: “42.42.42.42”

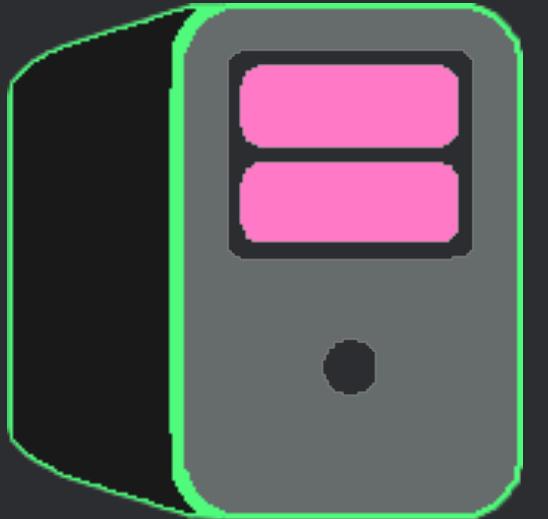
SIGNAL 2 → A: “69.69.69.69”



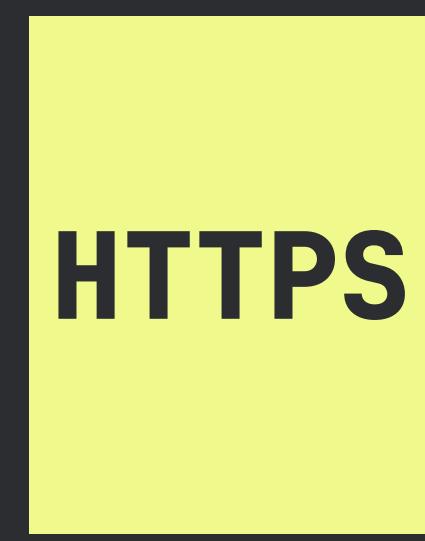








server

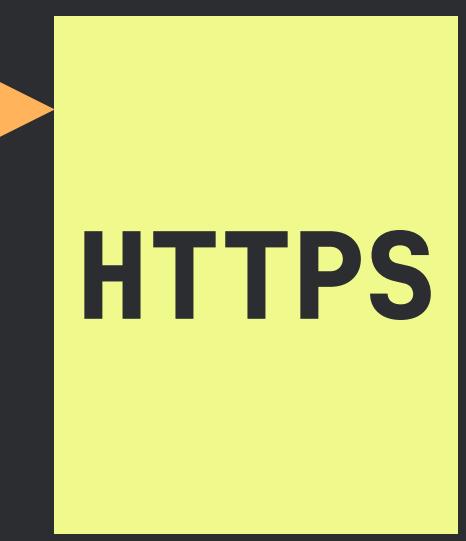


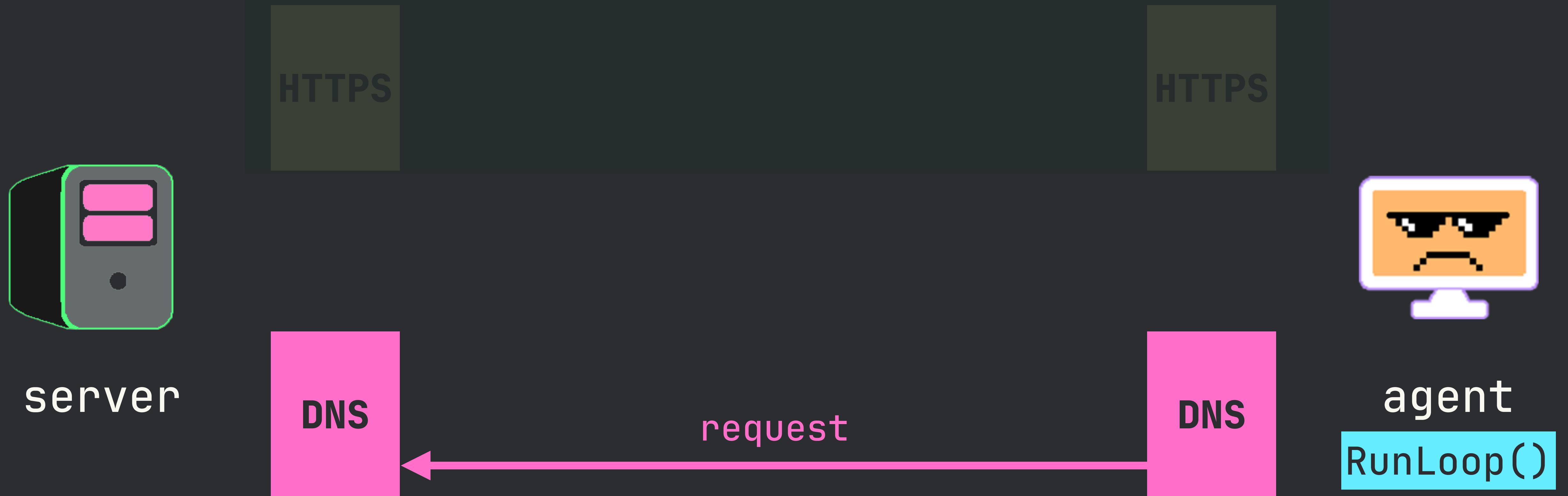
JSON{ "Change": true }

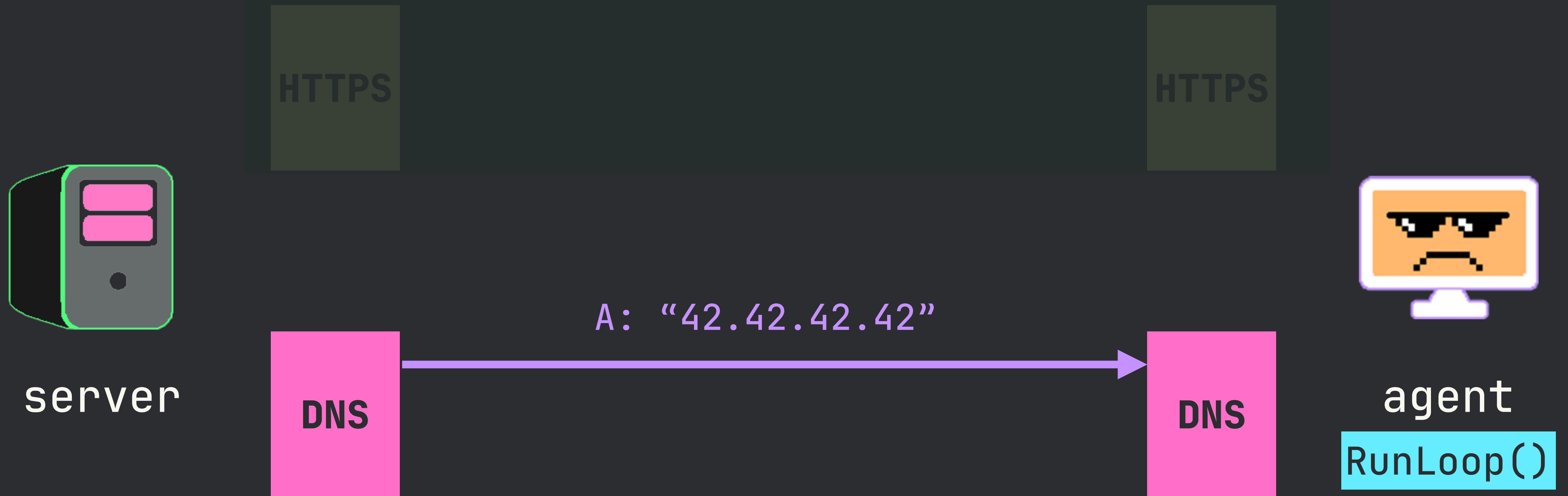


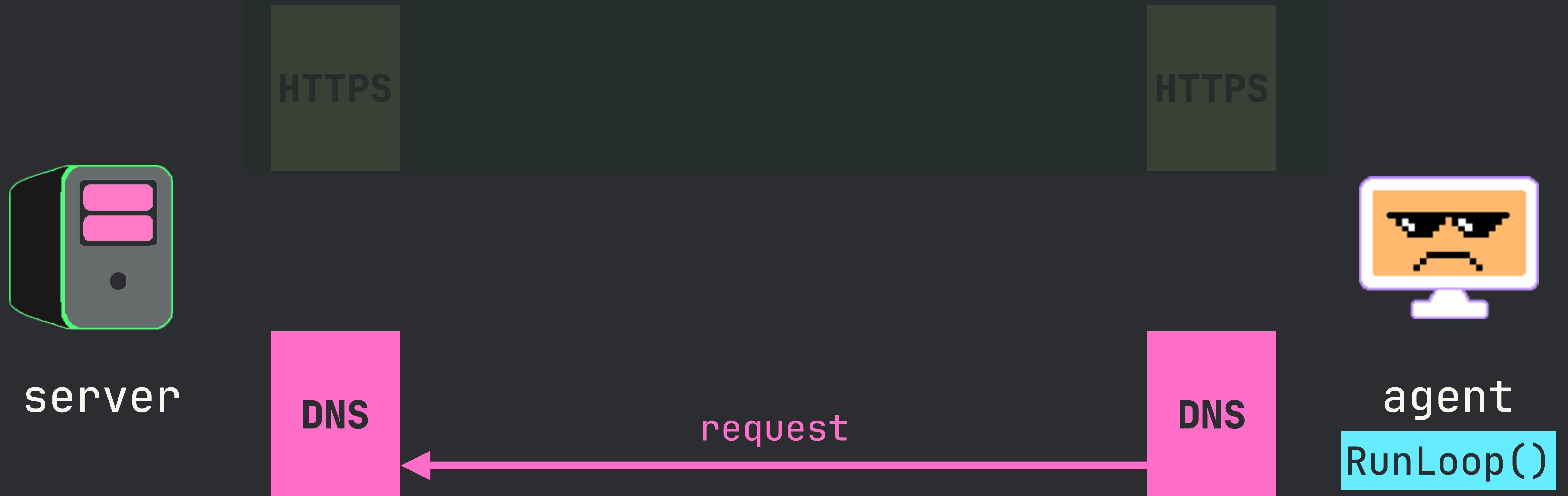
agent

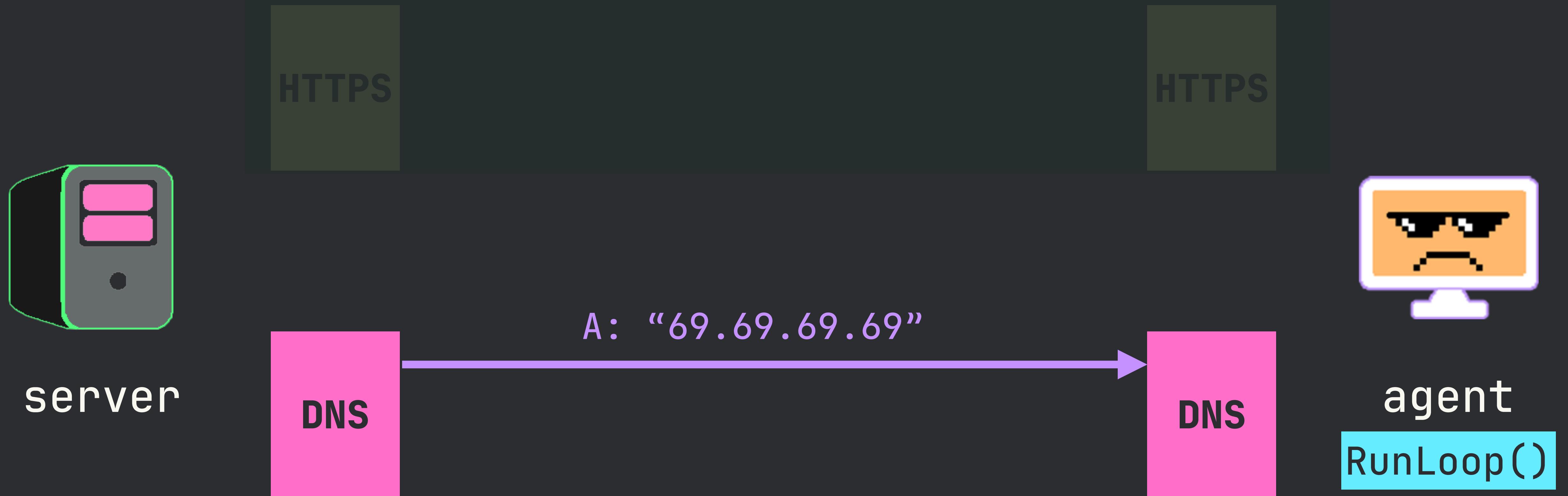
RunLoop()

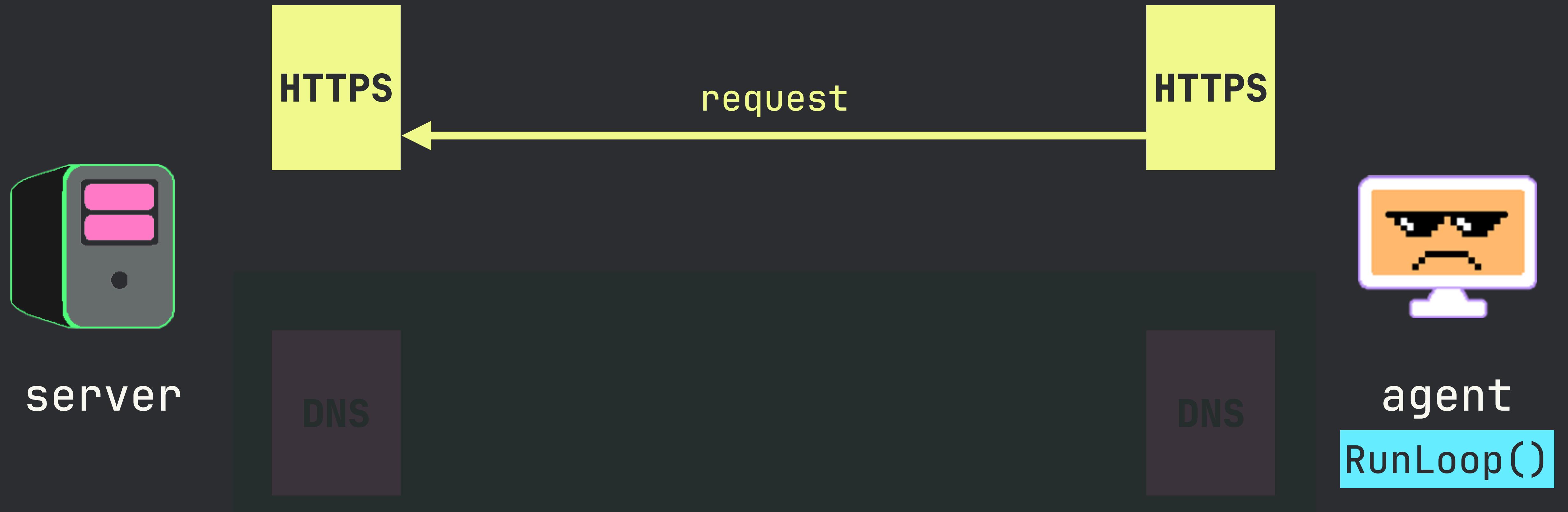












We are creating a SERVER + AGENT with the ability  
to perpetually communicate in either HTTPS or  
DNS, and the ability to dynamically/seamlessly  
transition between the two when we desire.

This is a great foundation for a covert C2  
channel to build upon → at the end I will give  
you ideas and resources how to do just that.

# interfaces



for now, all just conceptual, like pseudo-code but even more conceptual, just using diagrams

application  
(code base)

we want to add a protocol  
for agent ↔ server comms

application  
(code base)

we **decide** on **HTTPS**  
application  
(code base)

application  
(code base)



application  
(code base)

we need to integrate its logic  
into our main application

application  
(code base)

HTTPS

HTTPS

application  
(code base)

HTTPS

HTTPS

application  
(code base)

HTTPS

HTTPS

HTTPS

application  
(code base)

HTTPS

HTTPS

HTTPS

HTTPS

application  
(code base)

HTTPS

HTTPS

HTTPS

HTTPS

application  
(code base)

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

application  
(code base)

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

HTTPS

application  
(code base)

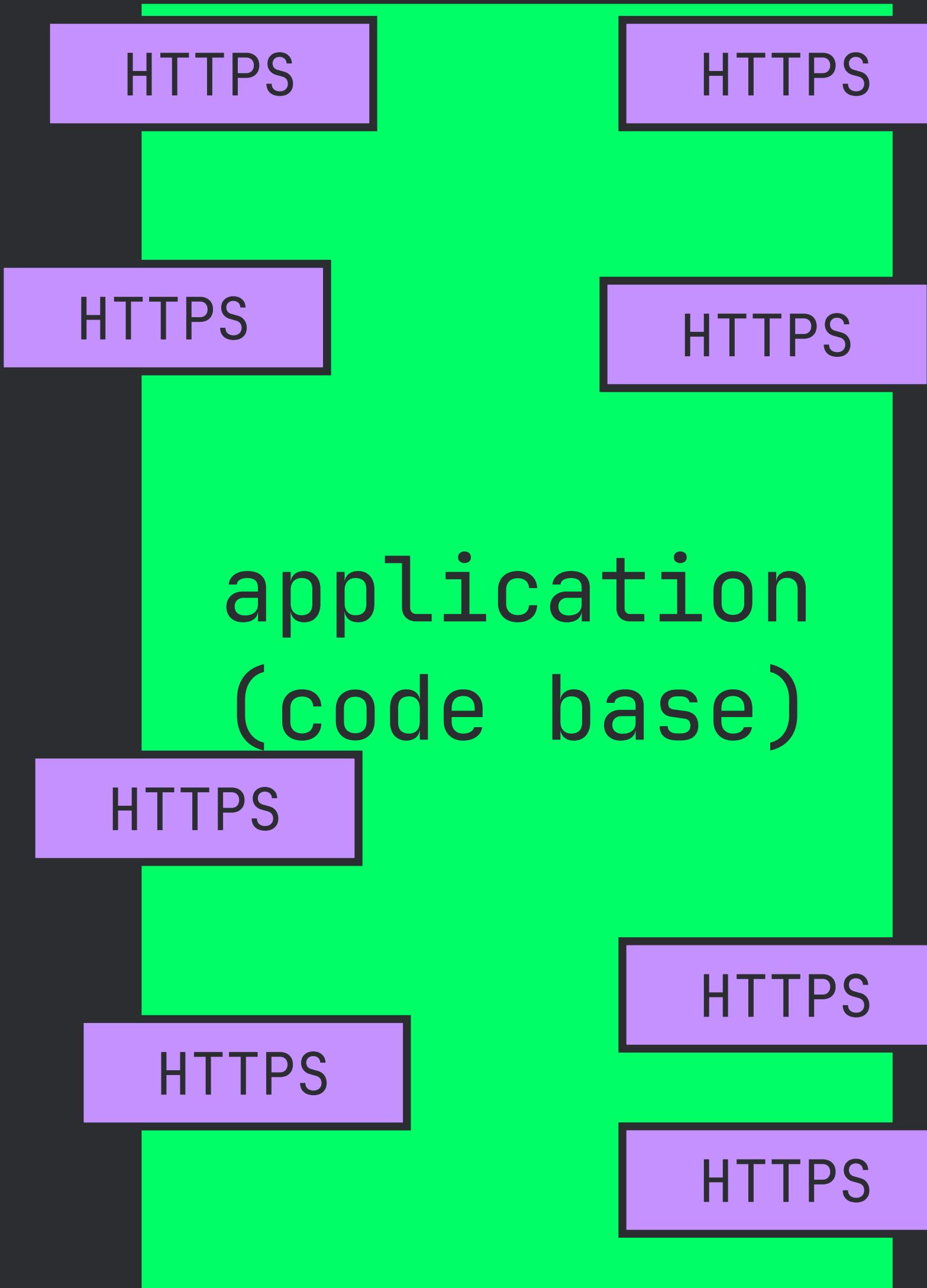
HTTPS

HTTPS

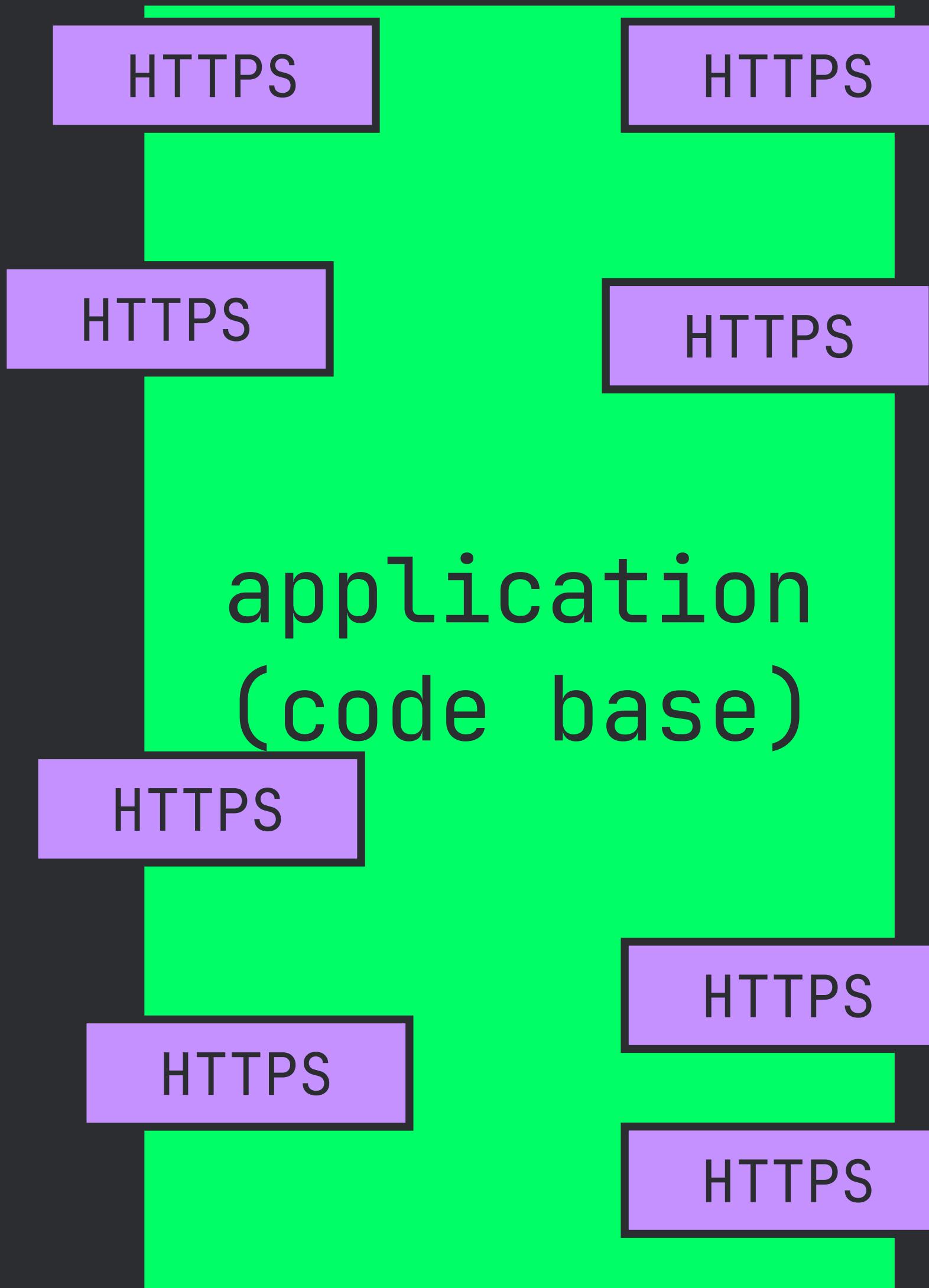
HTTPS

HTTPS

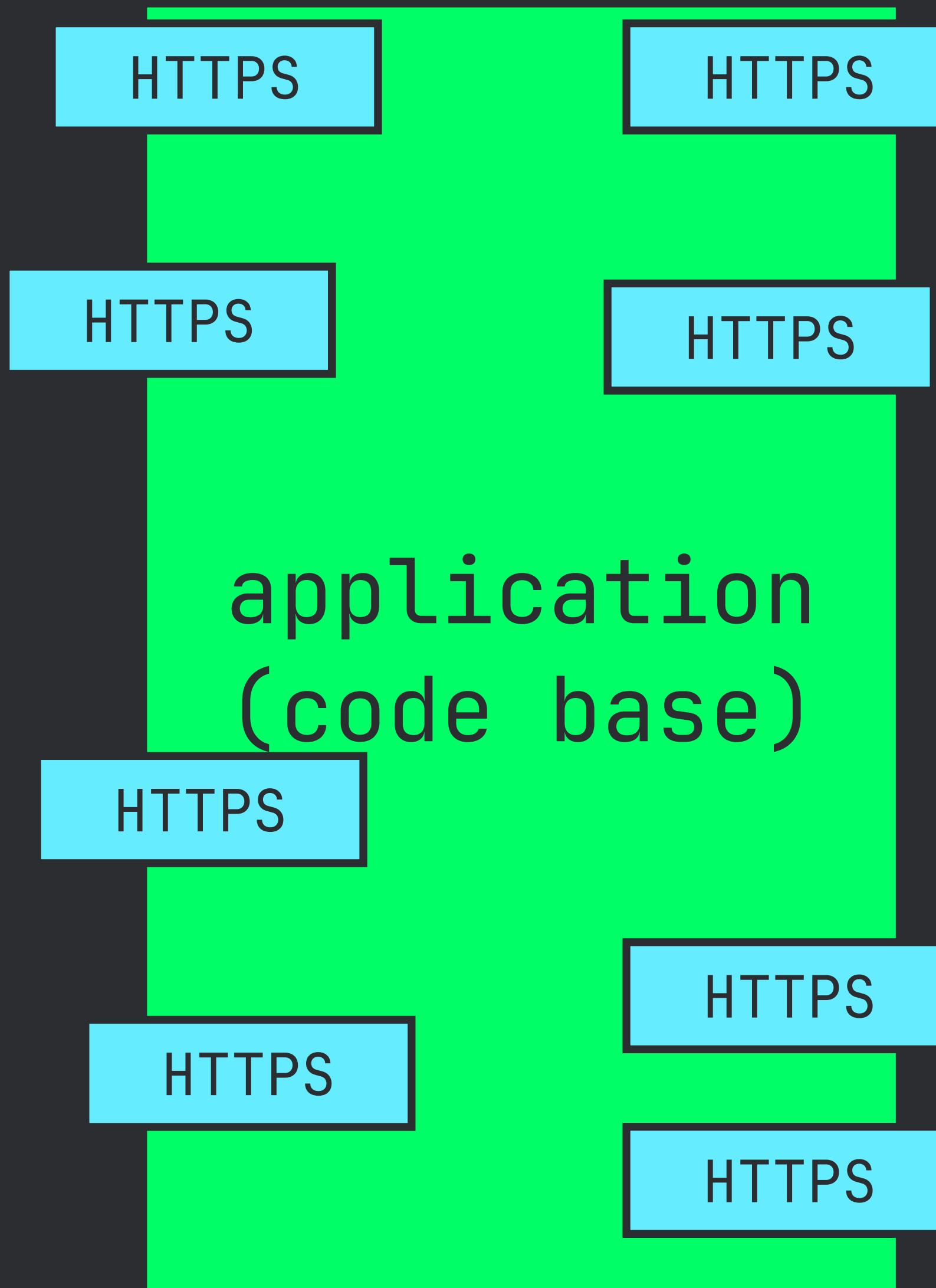
HTTPS



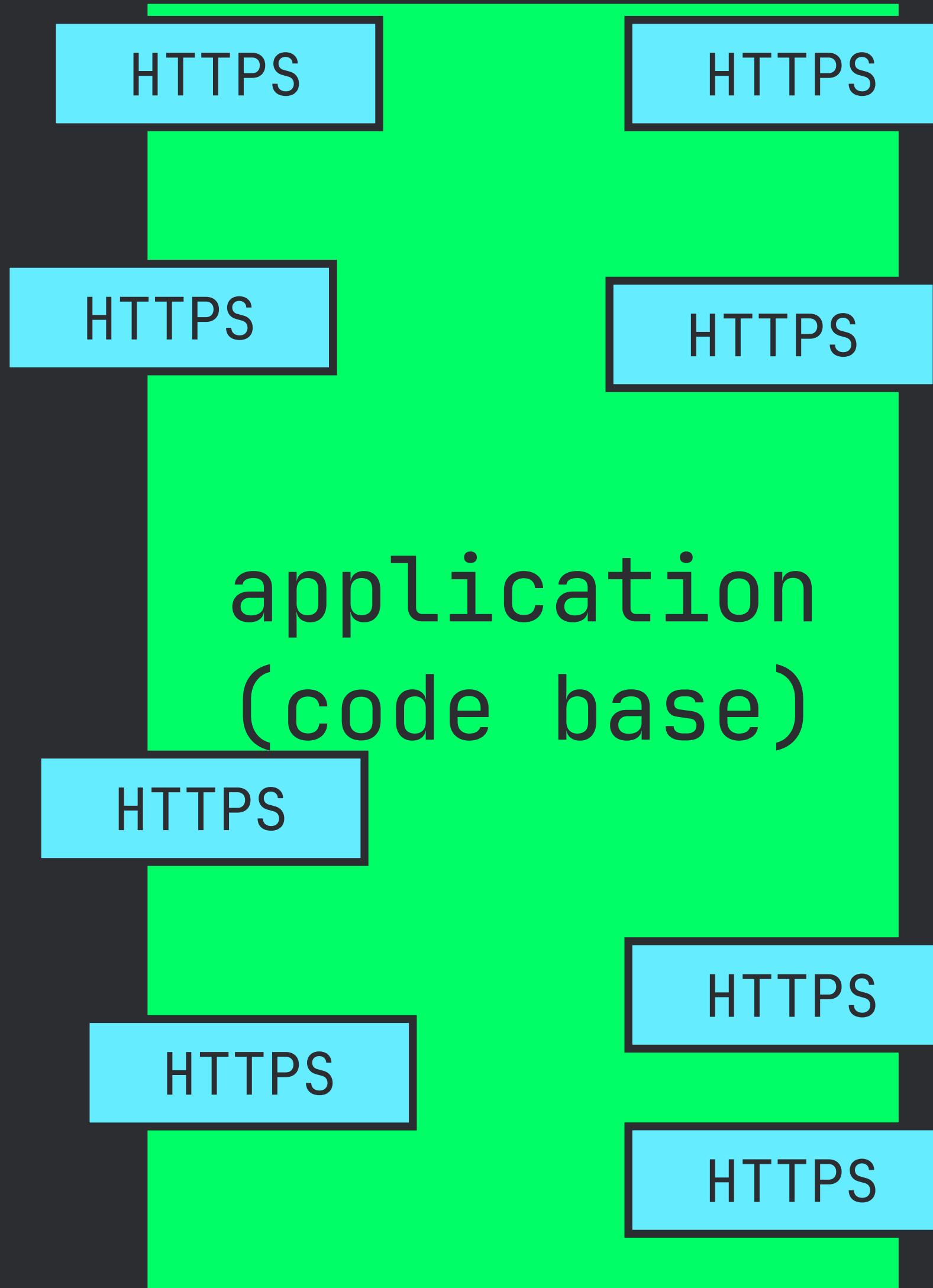
8 different places where we have  
now hardcoded protocol-specific  
logic into main codebase



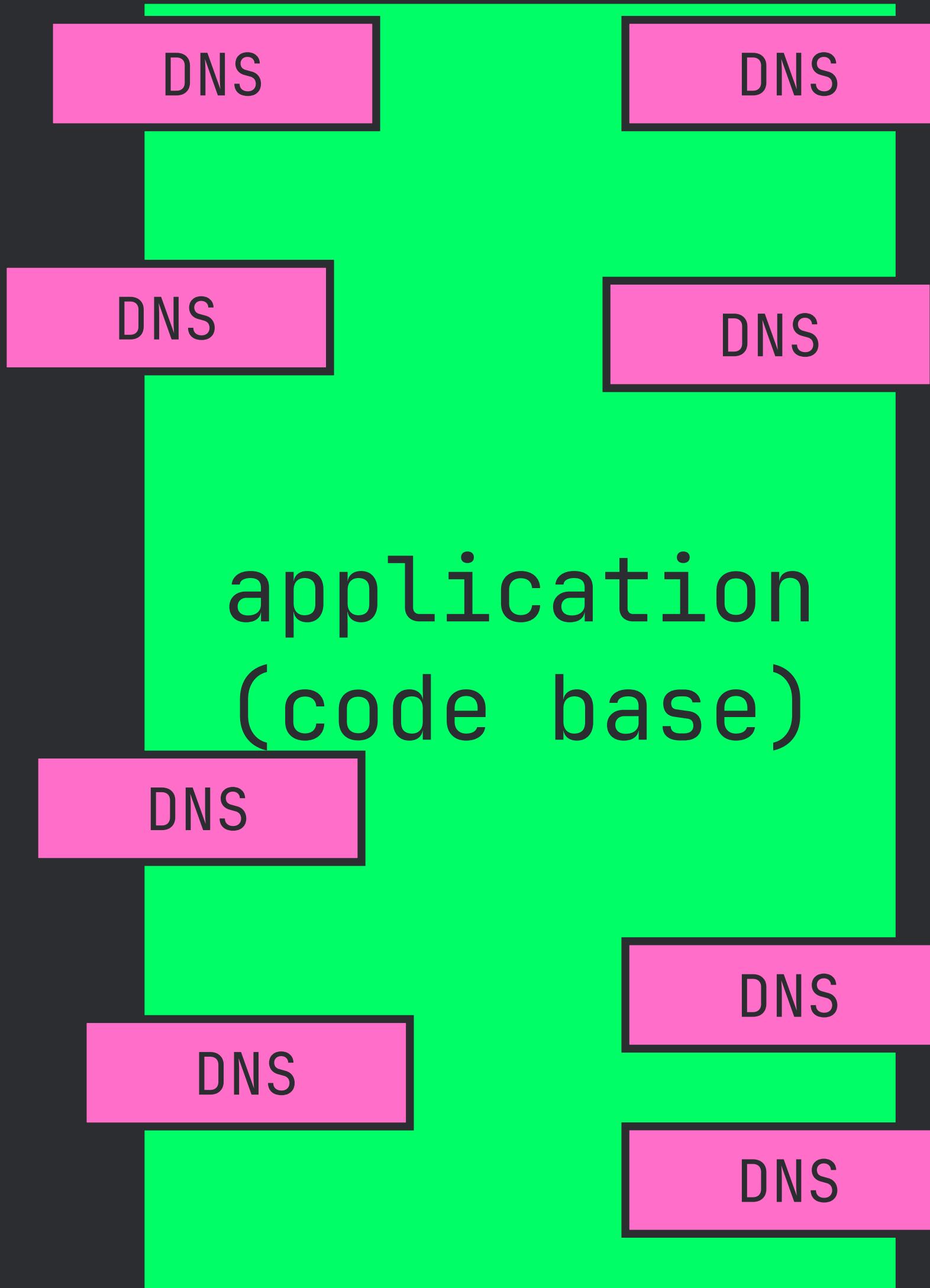
what if we now want to make a  
change to this code?



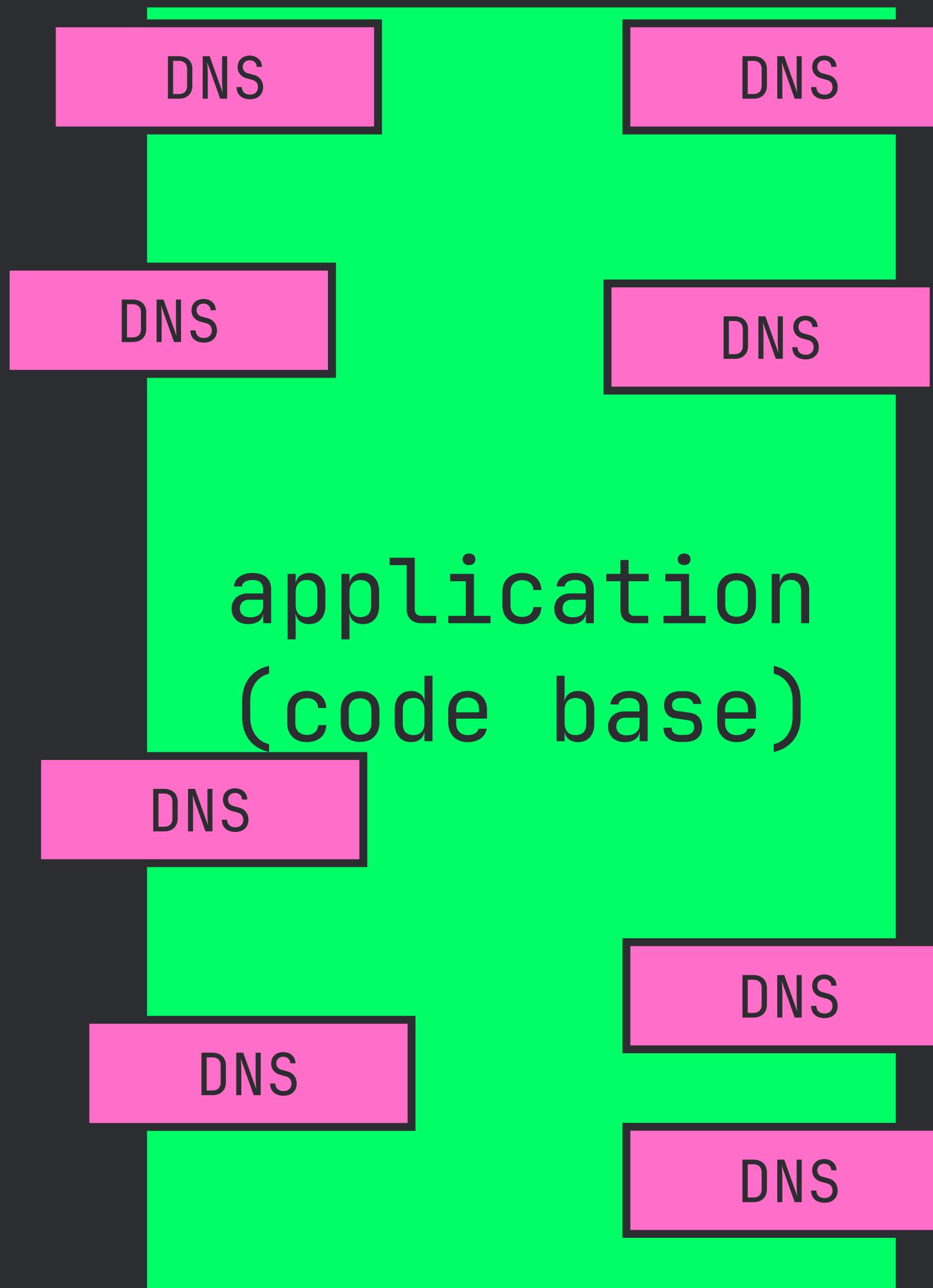
8 different places throughout  
codebase to go make changes



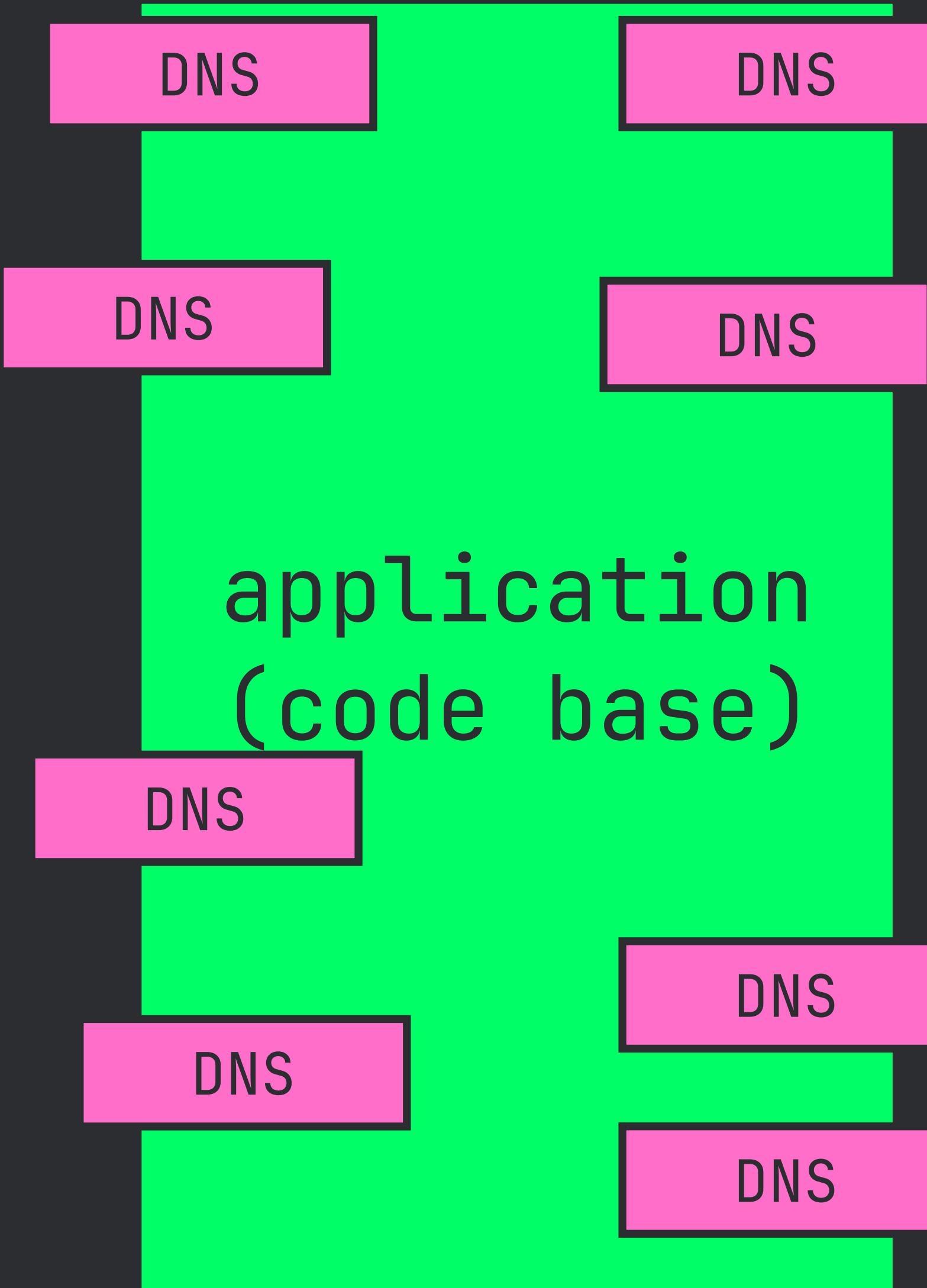
or, what if we want to replace  
it, we want to use DNS instead?



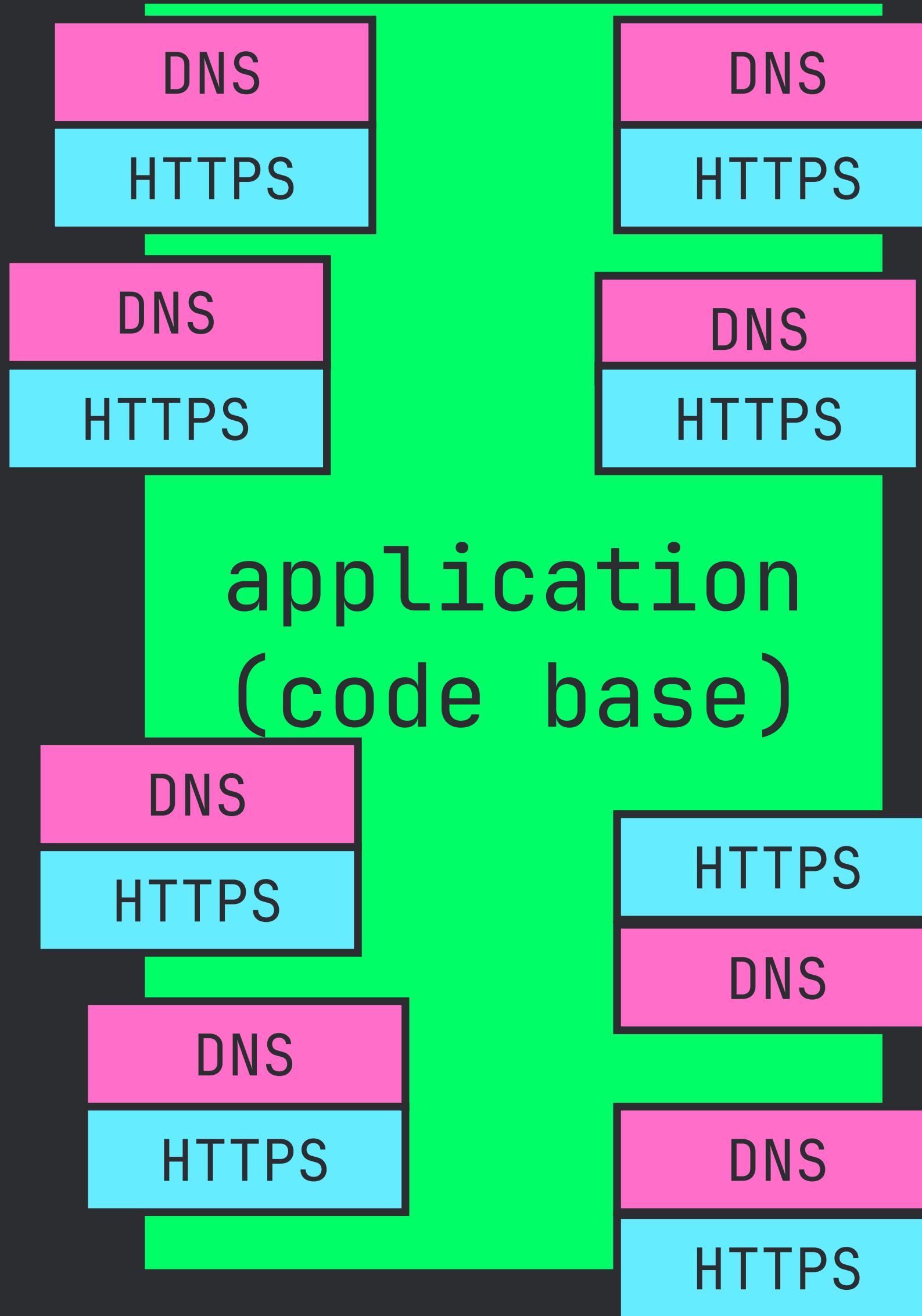
8 different places throughout  
codebase to go make changes



so... MAINTAINABILITY SUCKS

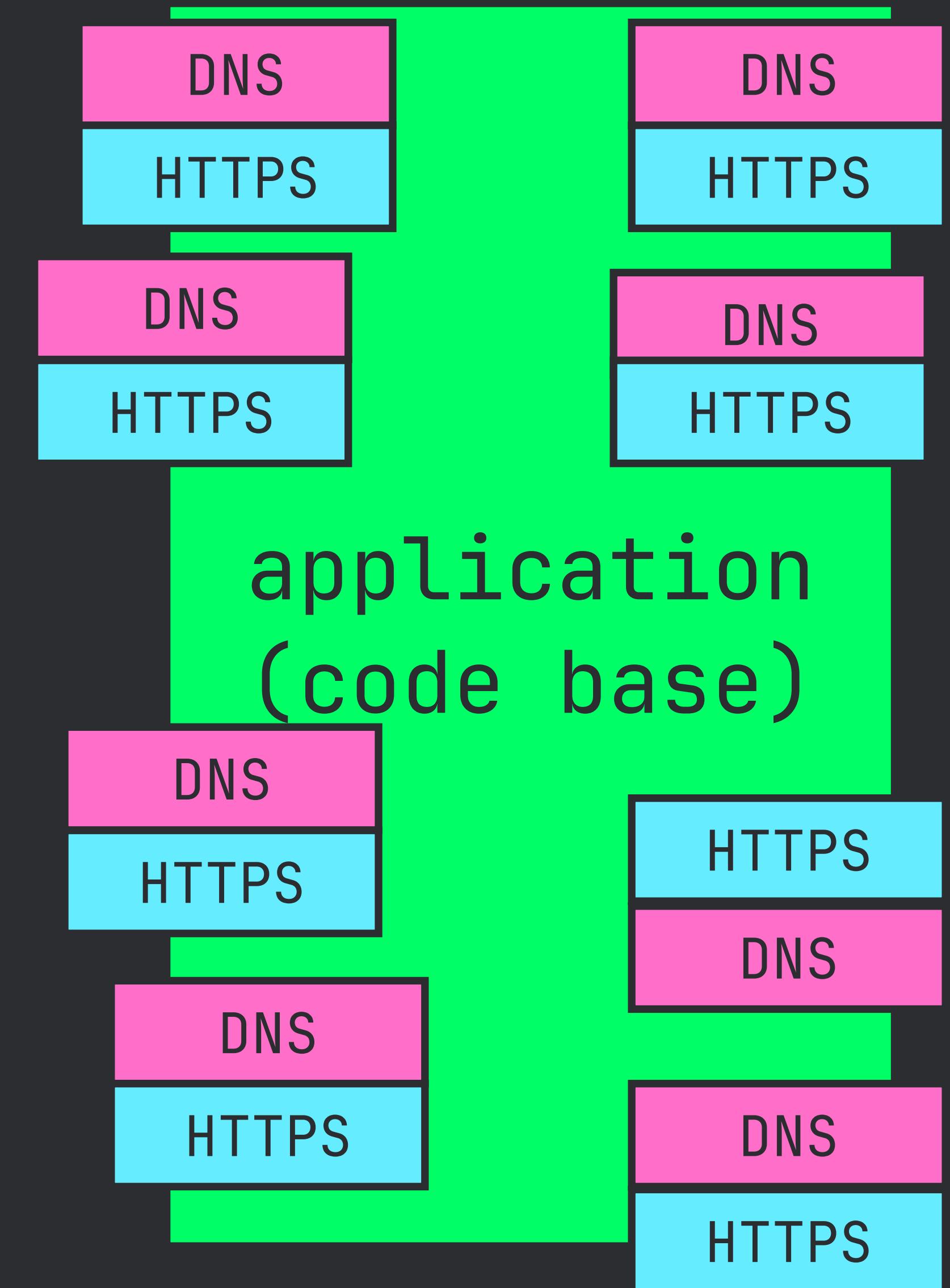


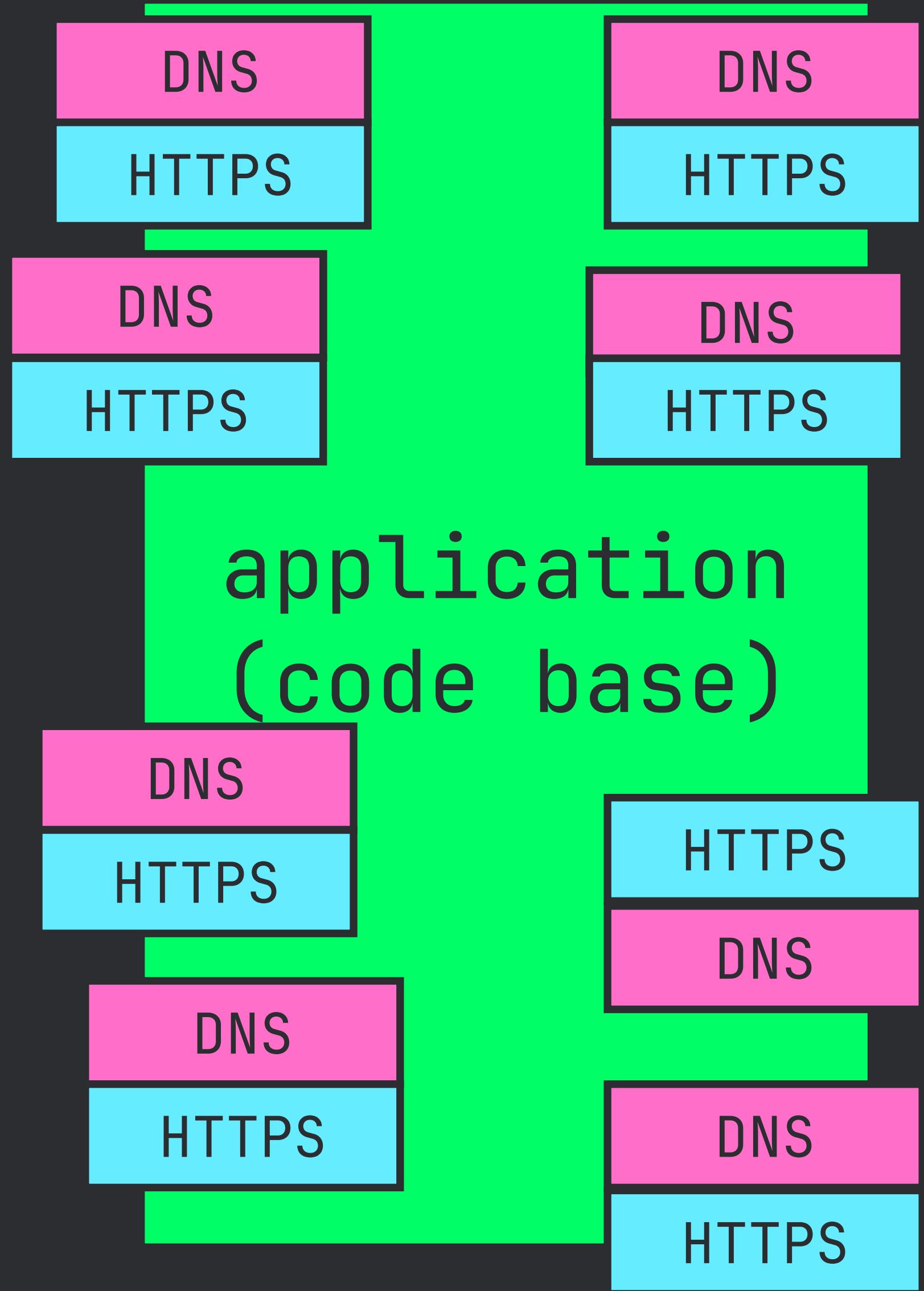
now consider, what if we want to  
actually have BOTH HTTPS and DNS?



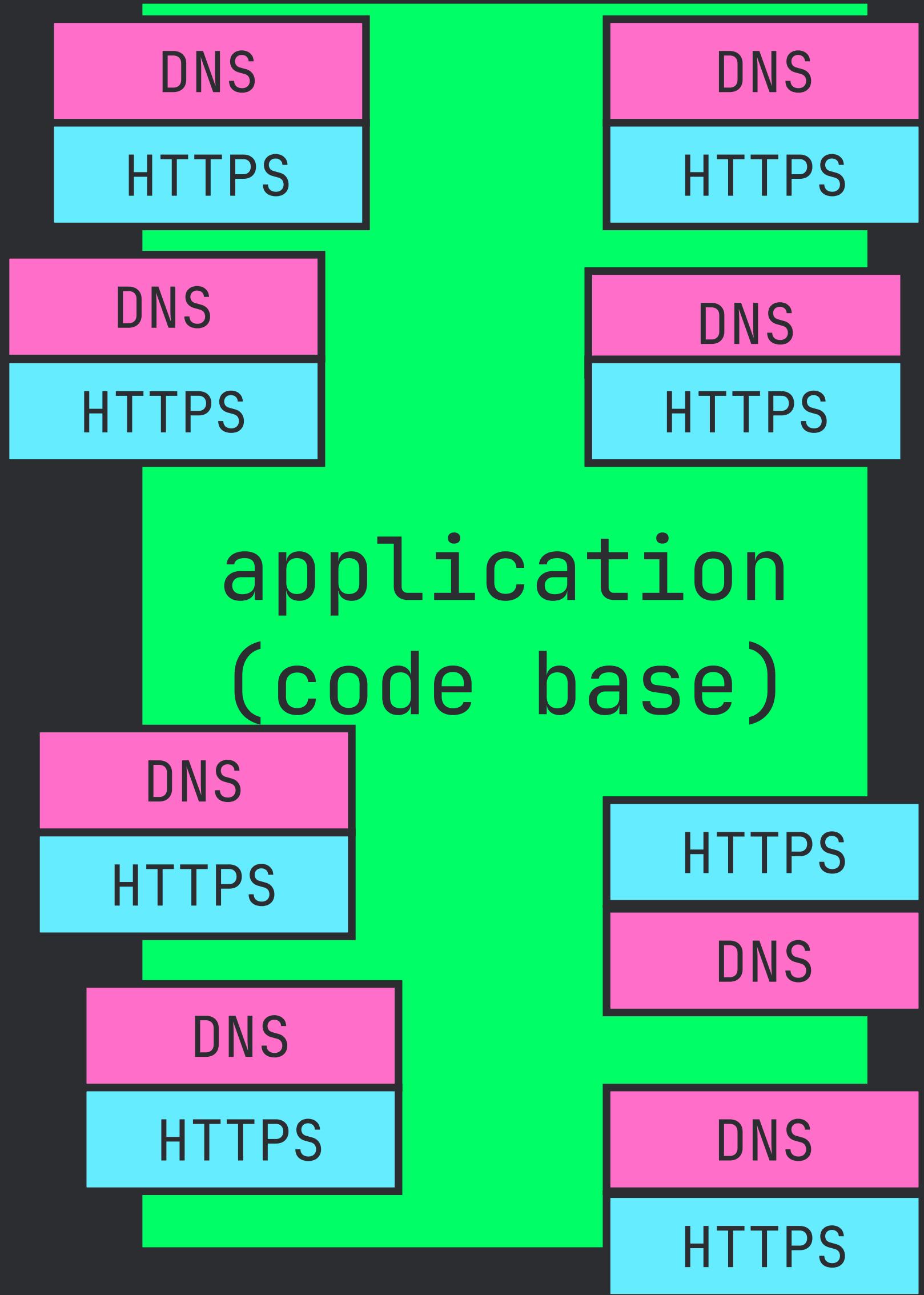
logic for each is replicated in all 8 places.. Now let's say we want to also add HTTP/1.1,

WebSockets, and ICMP...

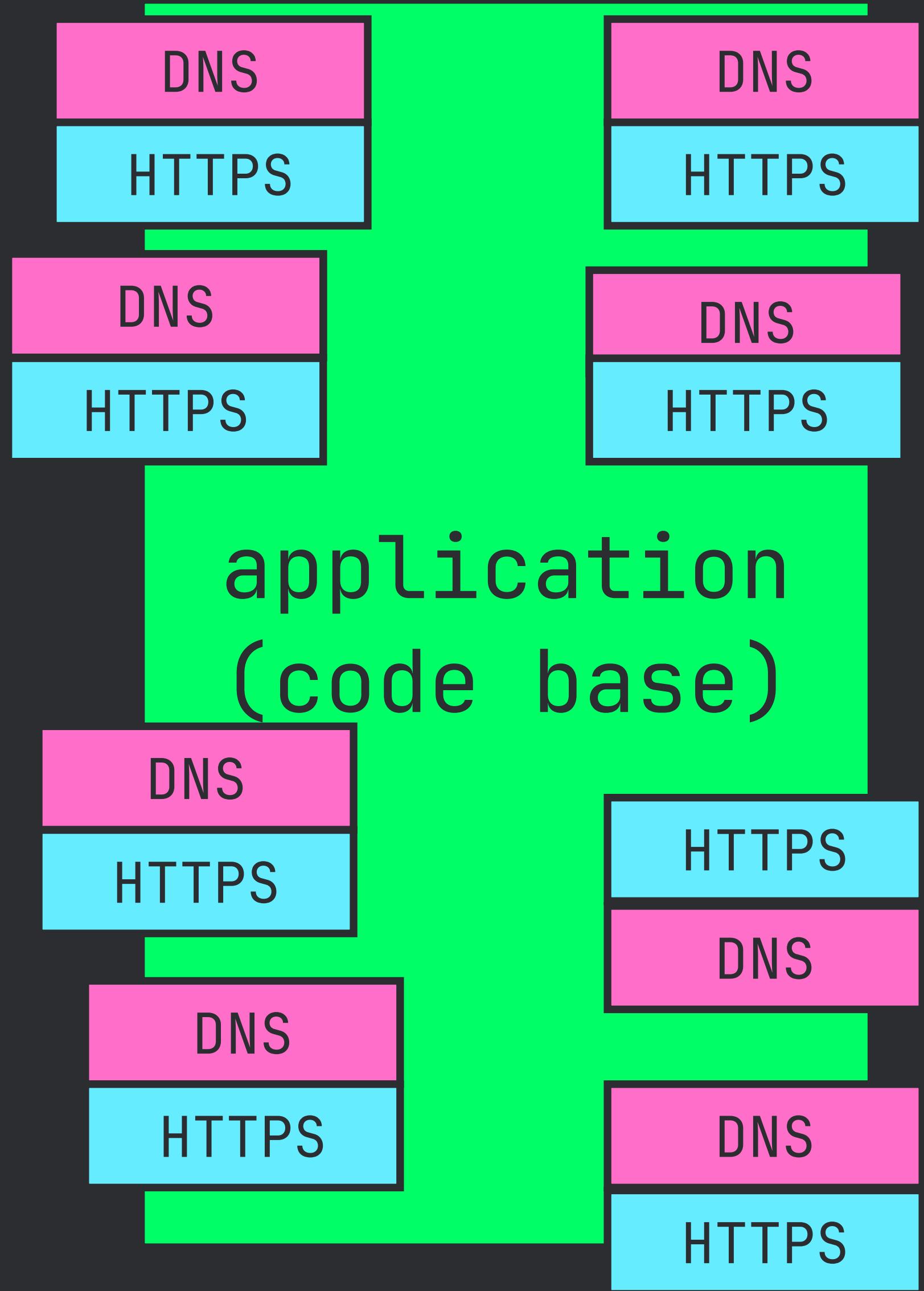




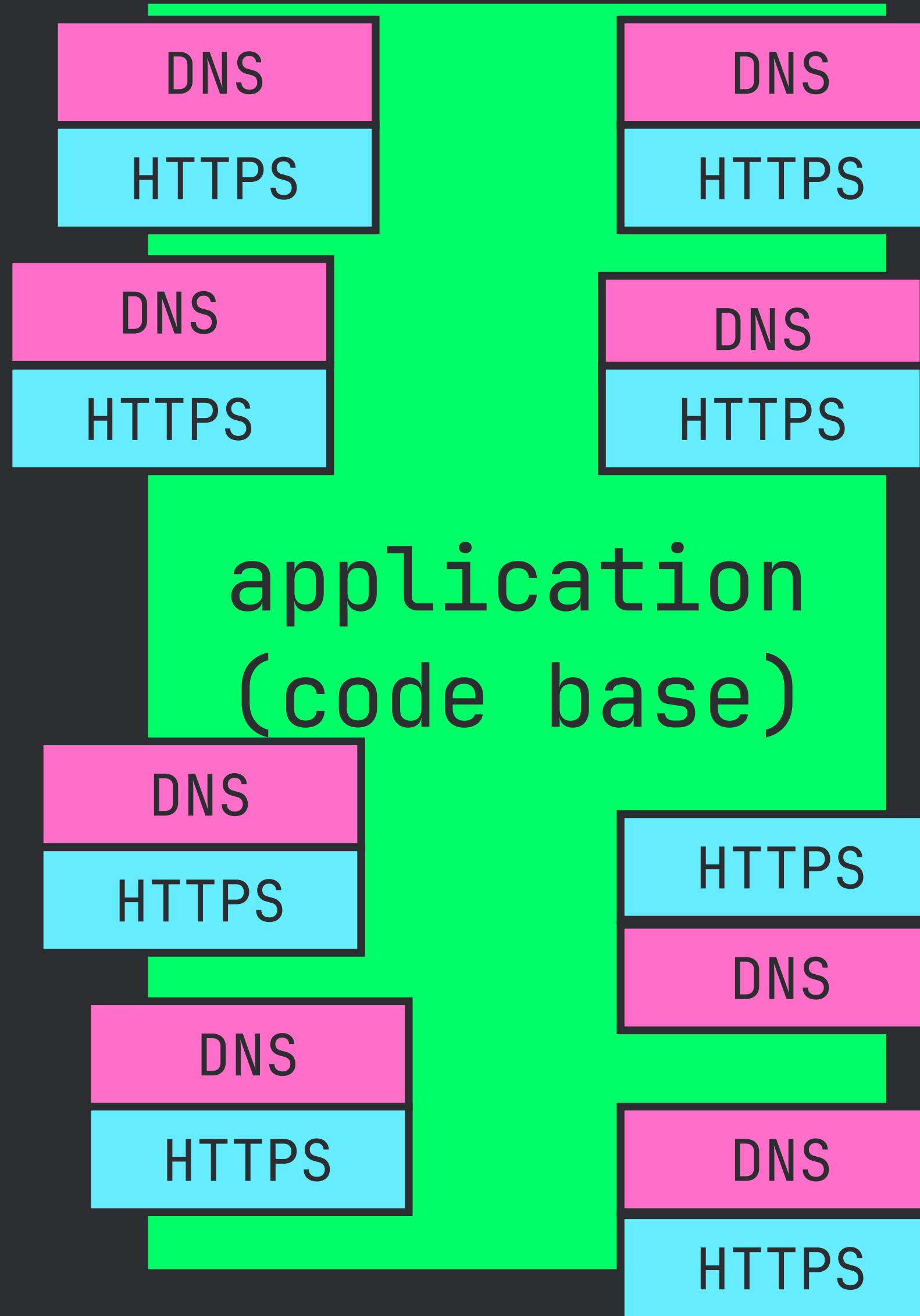
this “direct implementation”, or  
hard-coded, design is brittle



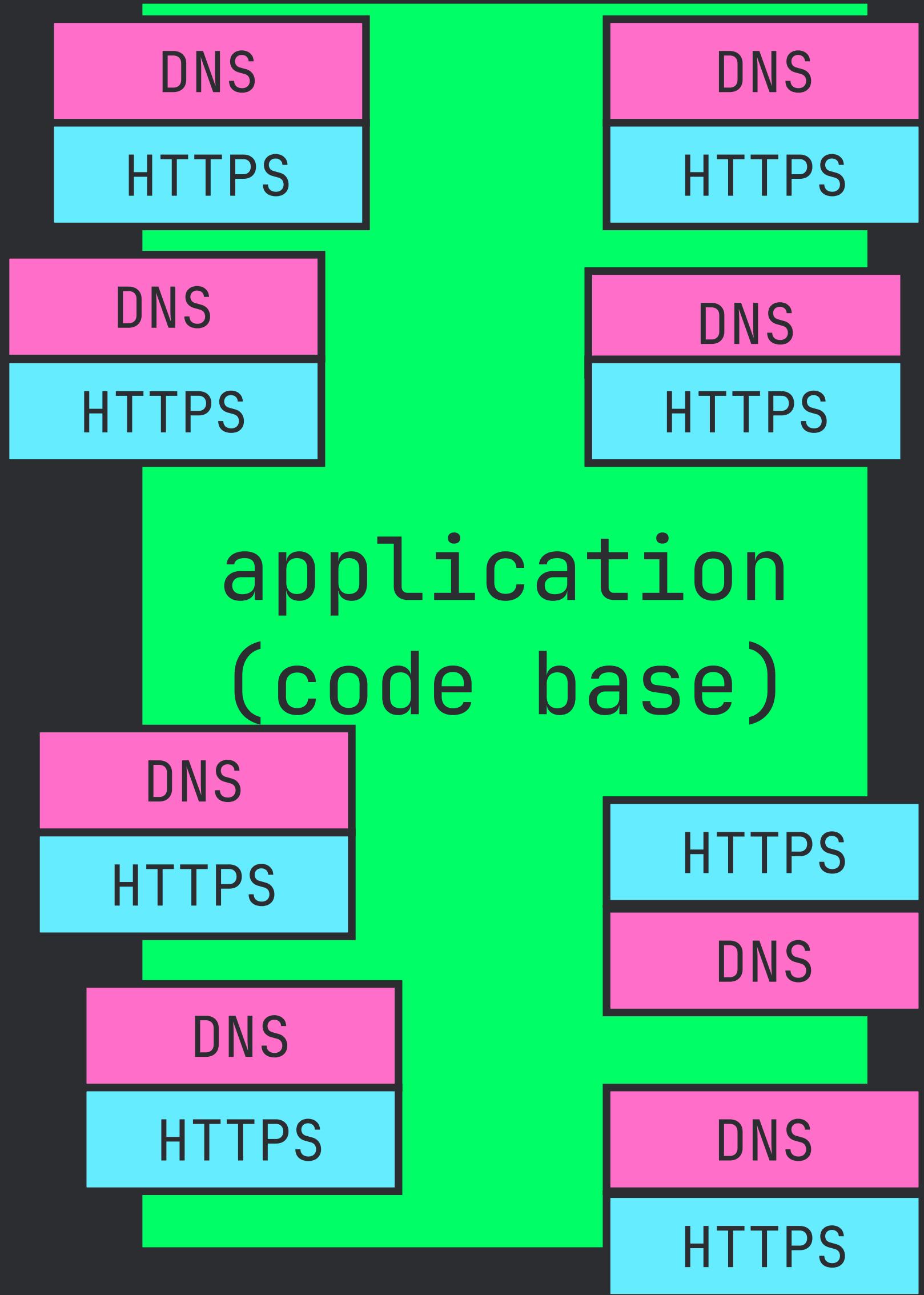
it lacks flexibility, or more  
accurately, extensibility



BUT... There is a much better way  
to design this using **INTERFACES**

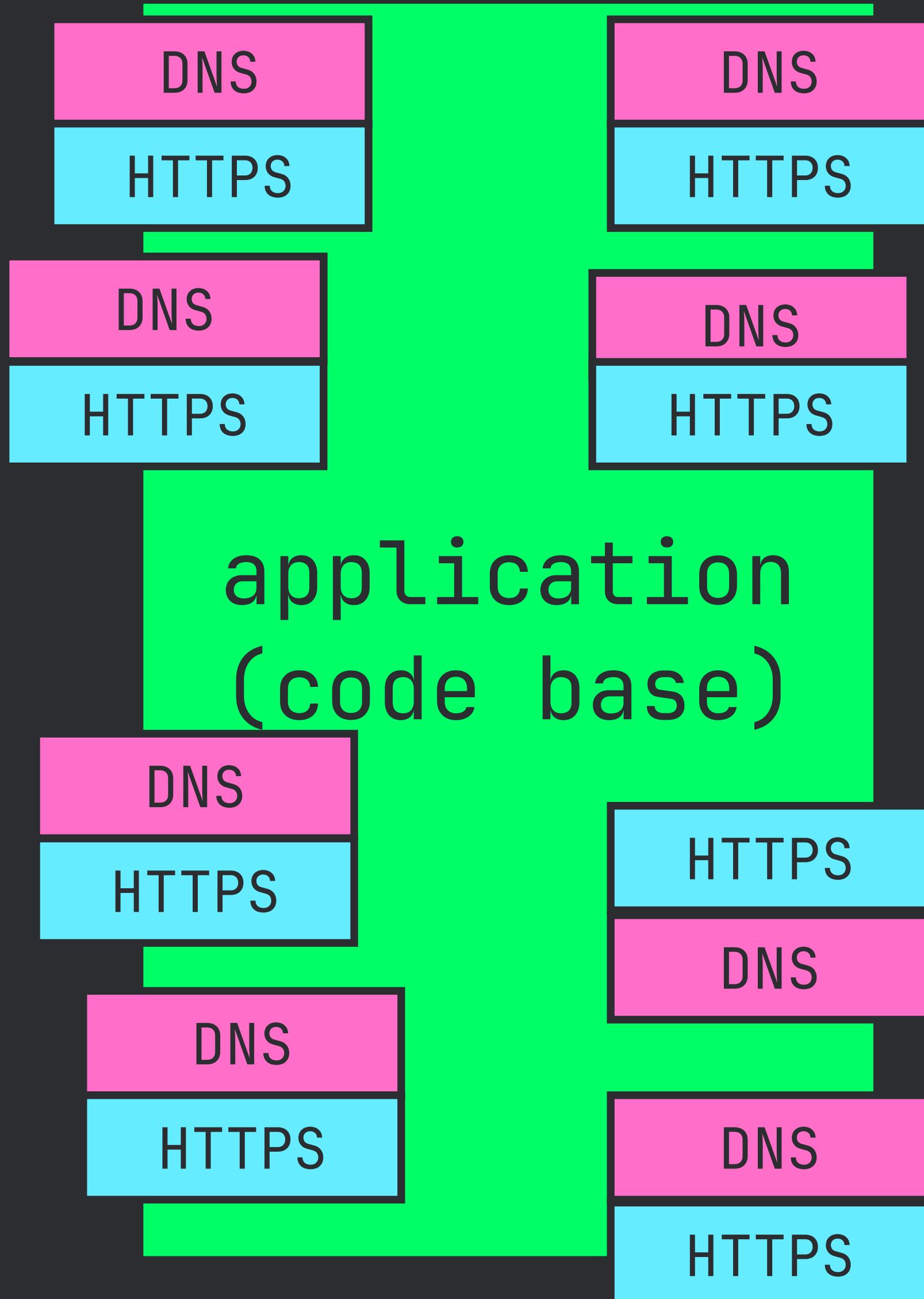


Think of an INTERFACE  
as a CONTRACT



in this case, a CONTRACT that  
defines “what it means” to be  
a communication protocol

# LET'S RESET



application  
(code base)

protocol

(I'm drawing this external to  
codebase for clarity but of  
course all of this is inside  
of codebase)

contract

protocol

protocol

protocol

application  
(code base)

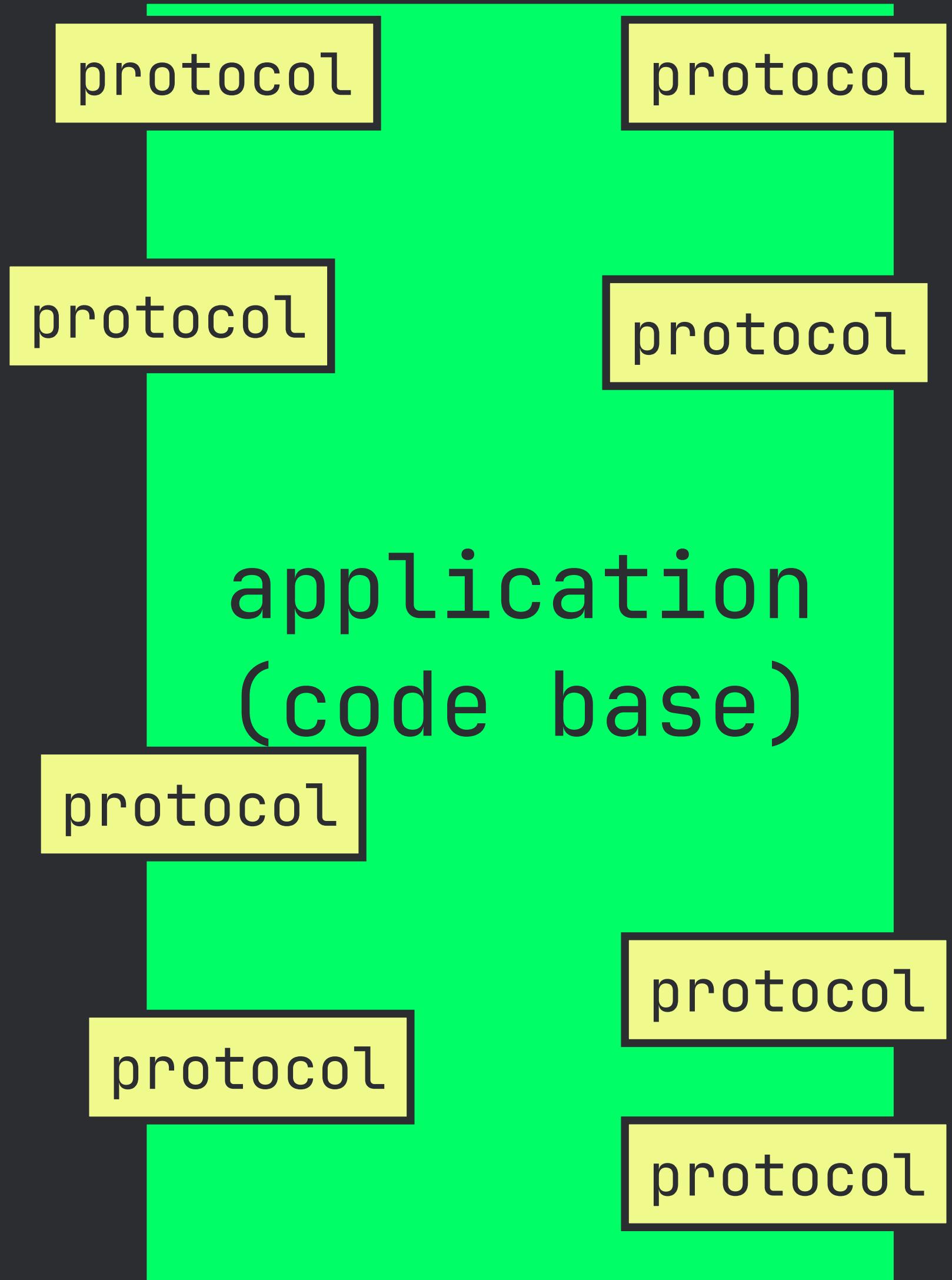
protocol

protocol

protocol

protocol

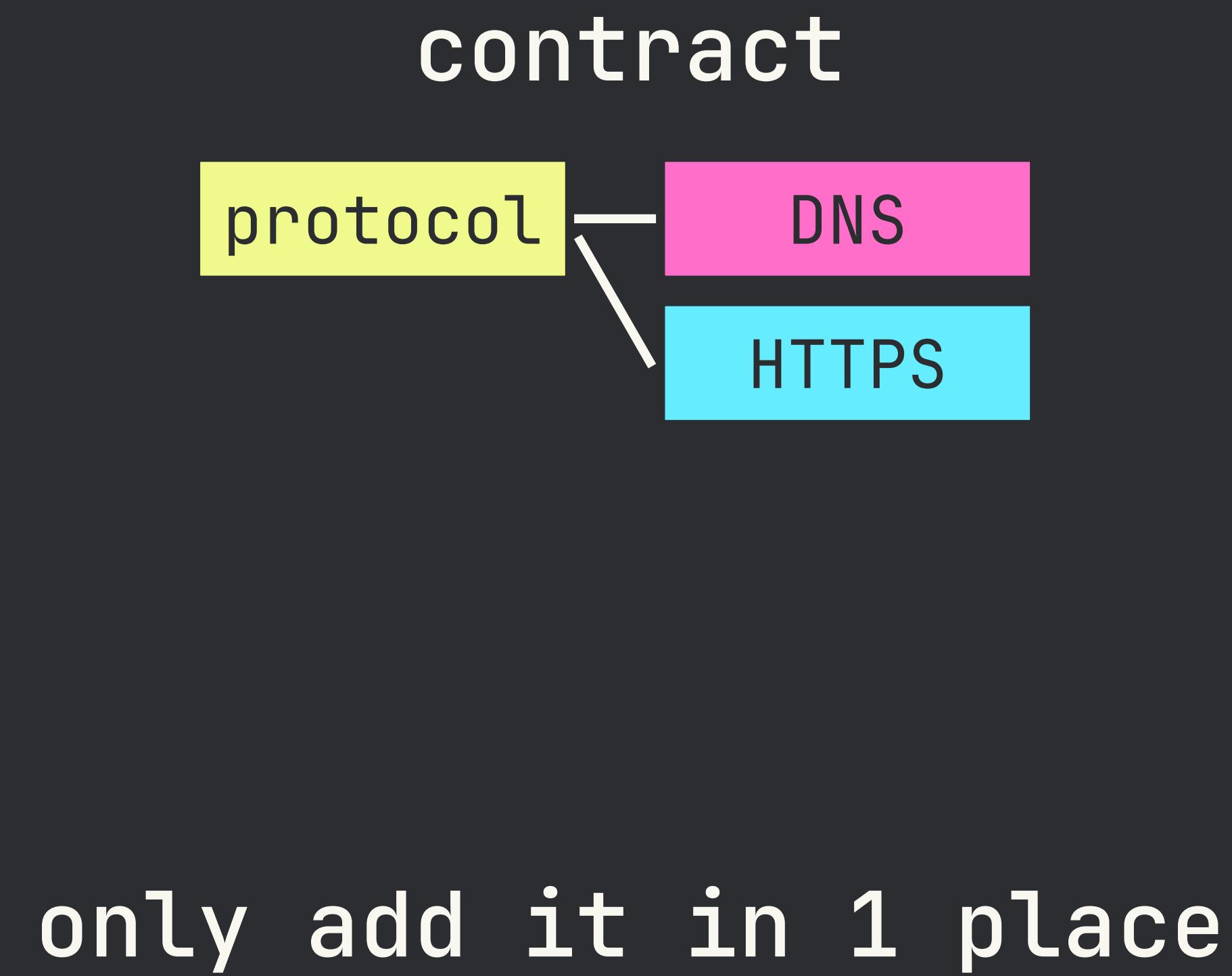
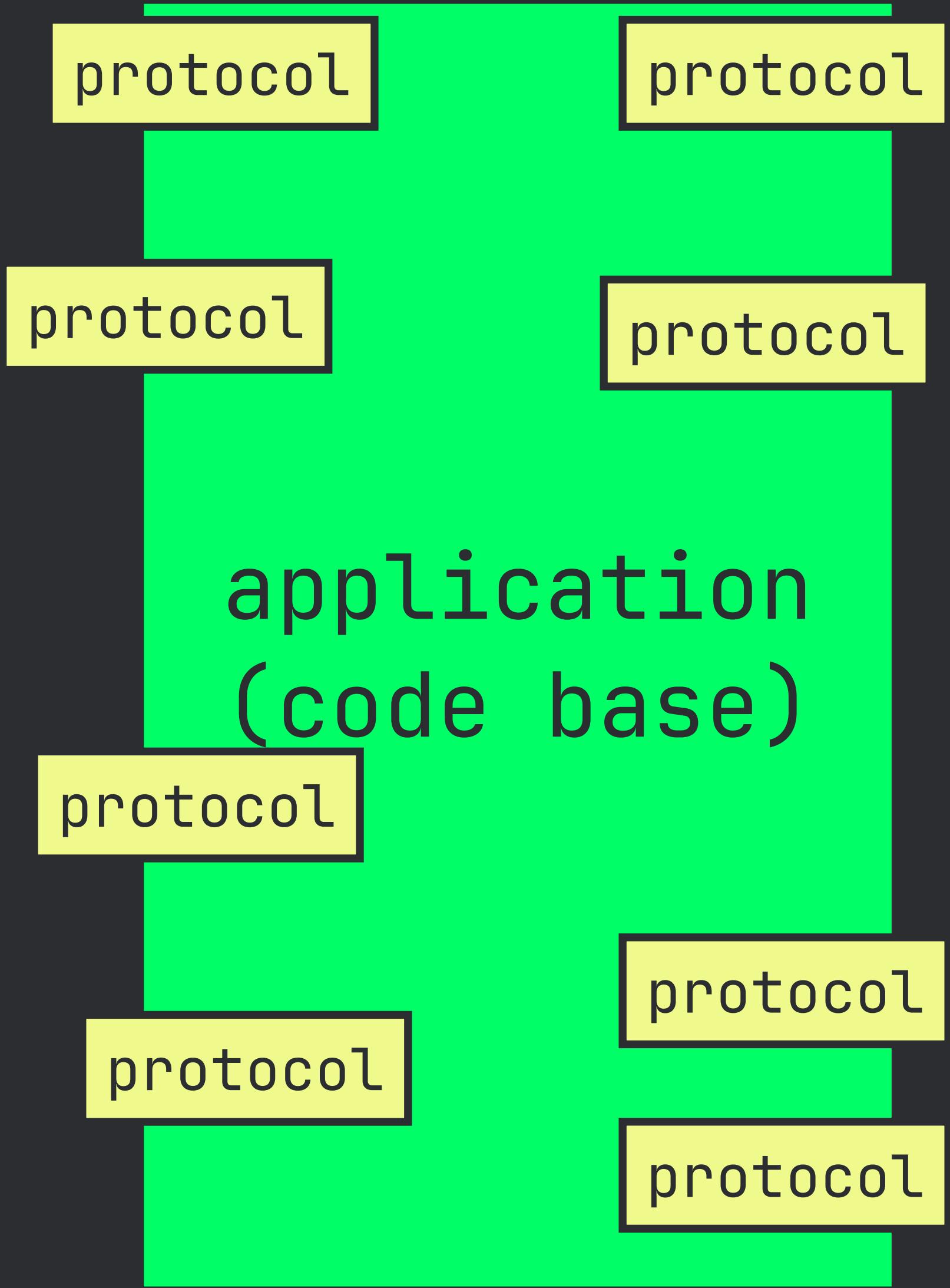
protocol

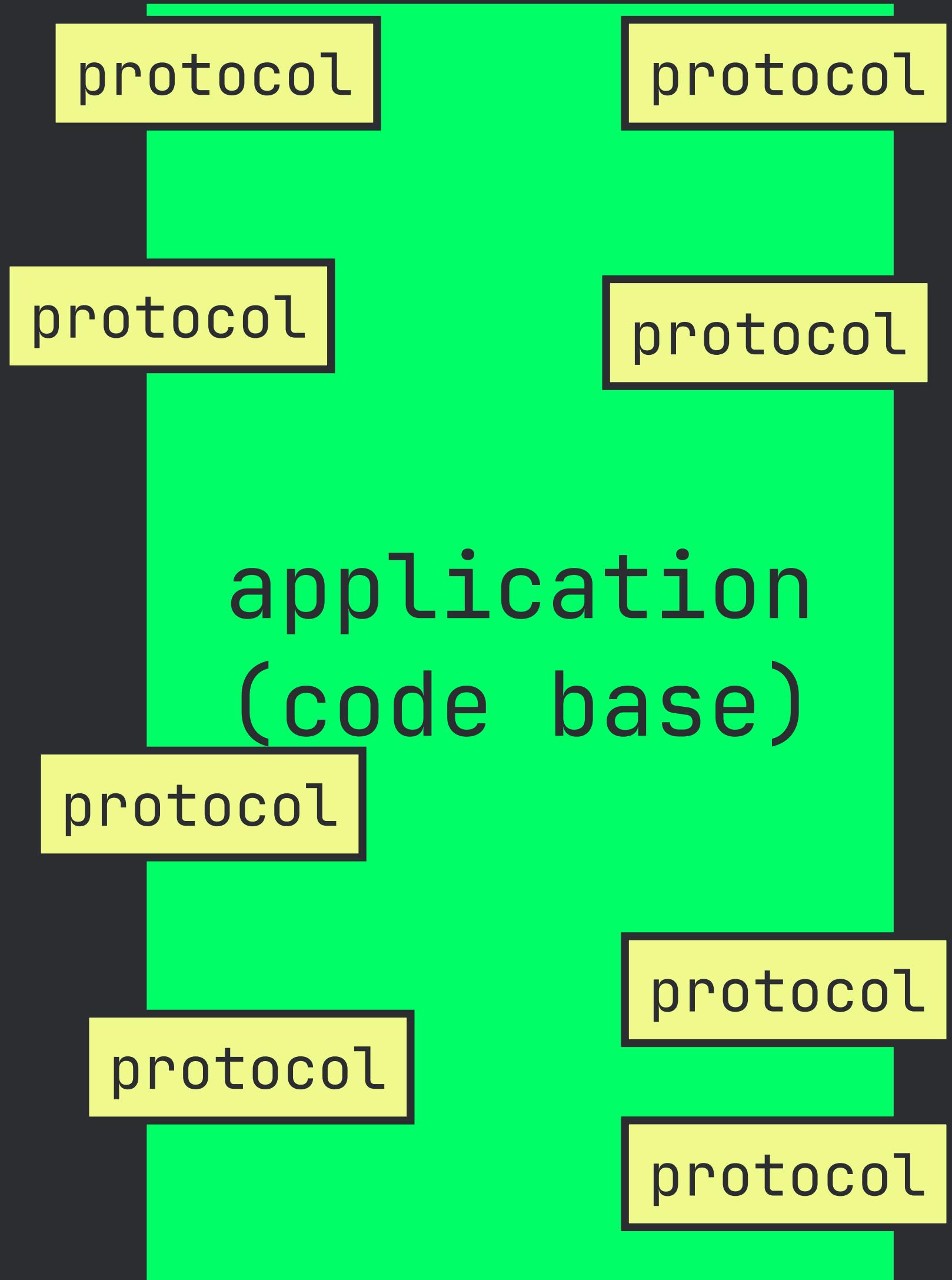


contract

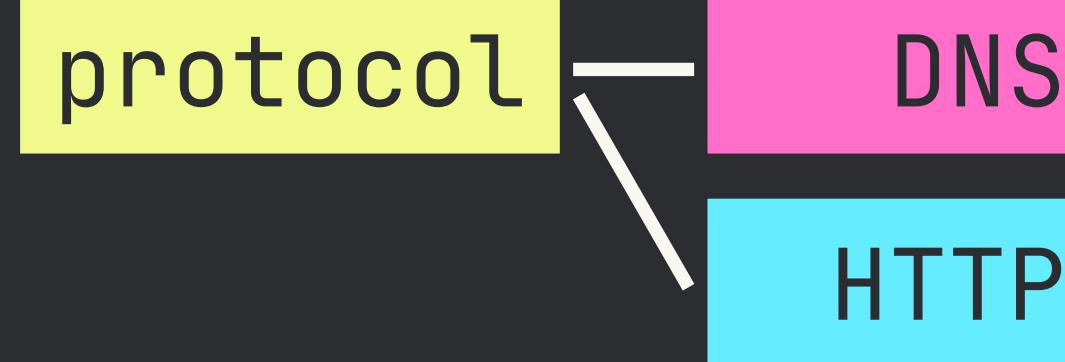
protocol — DNS

what if we also want HTTPS?

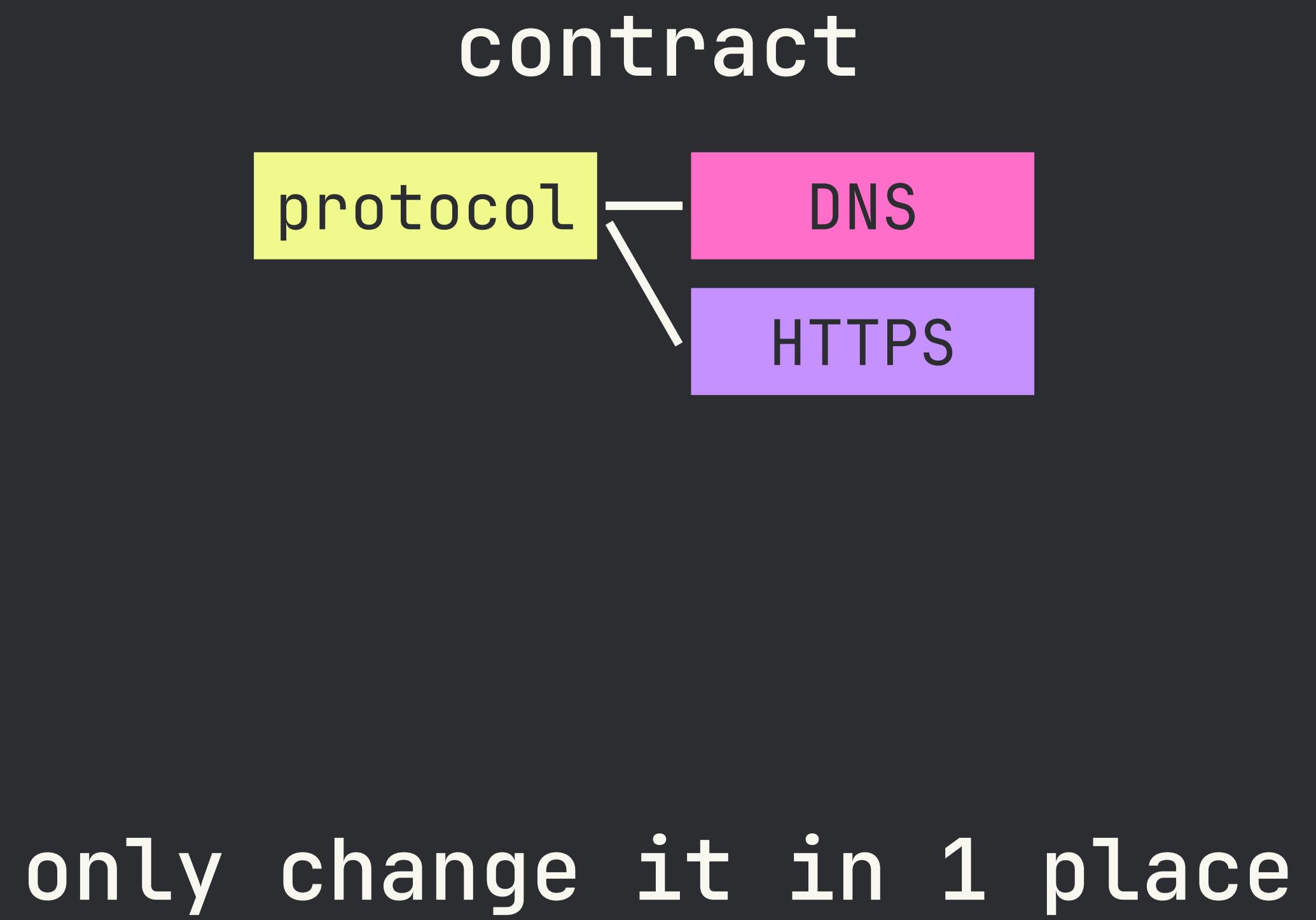
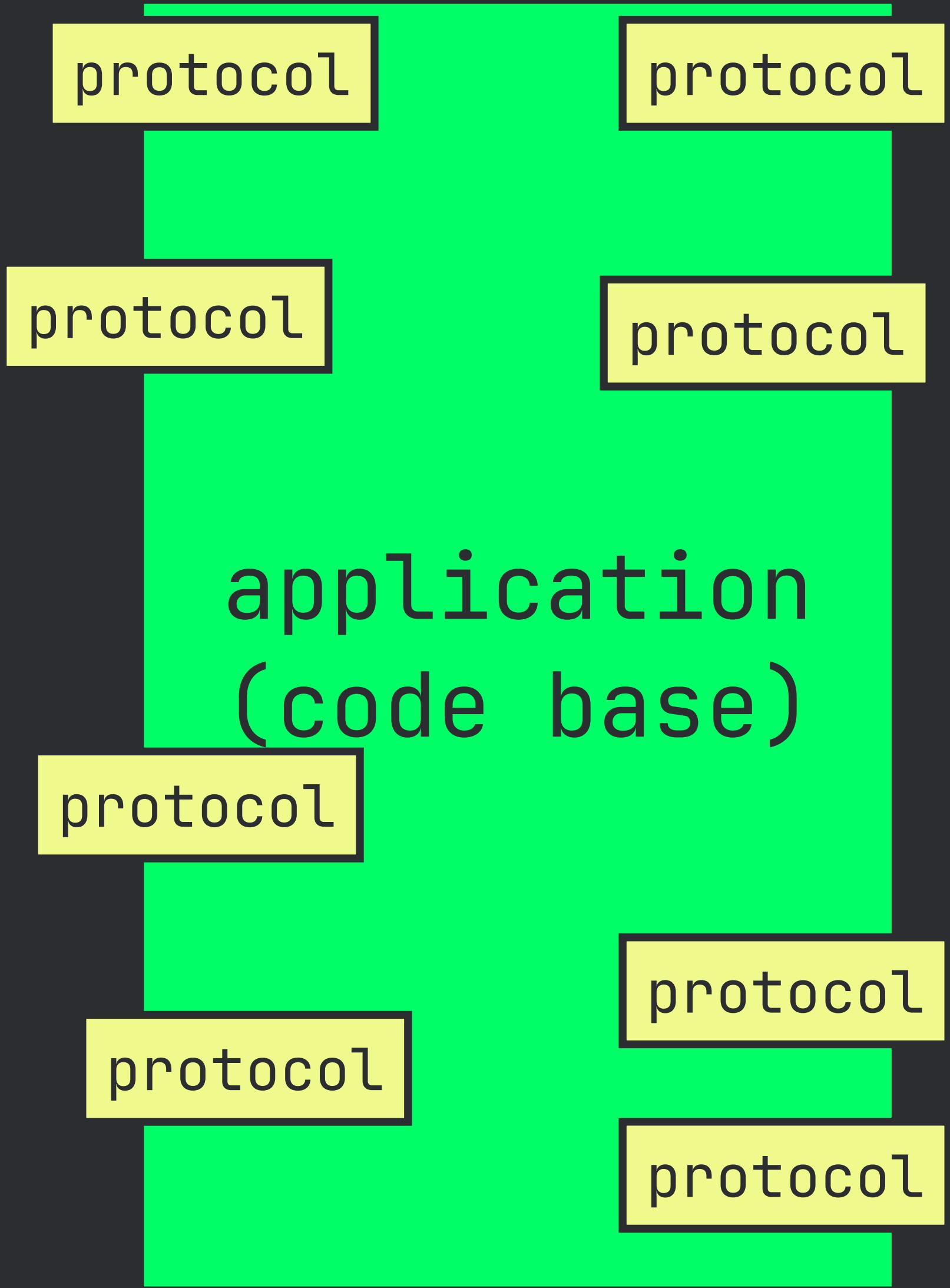


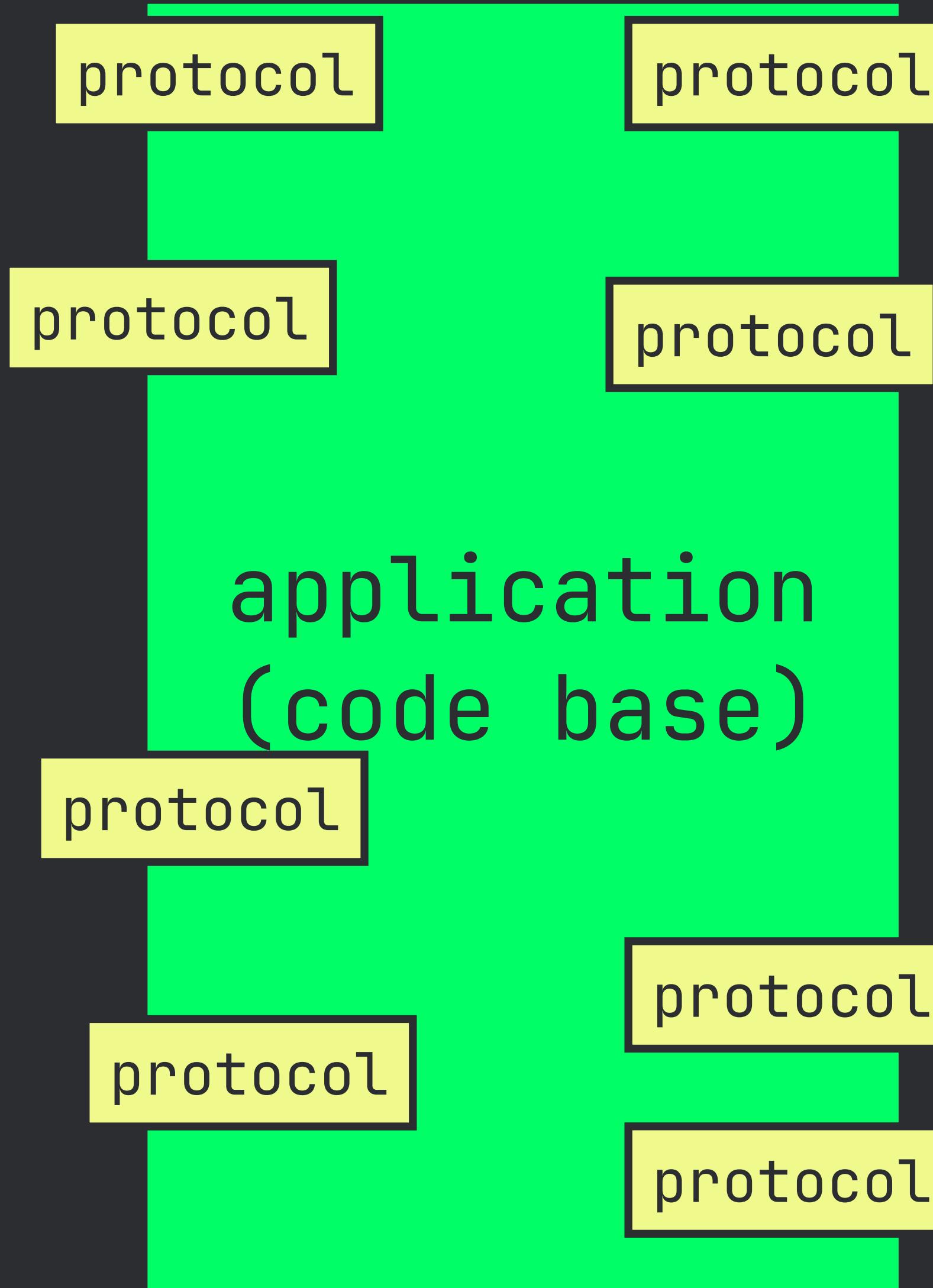


contract



what if we want to change HTTPS





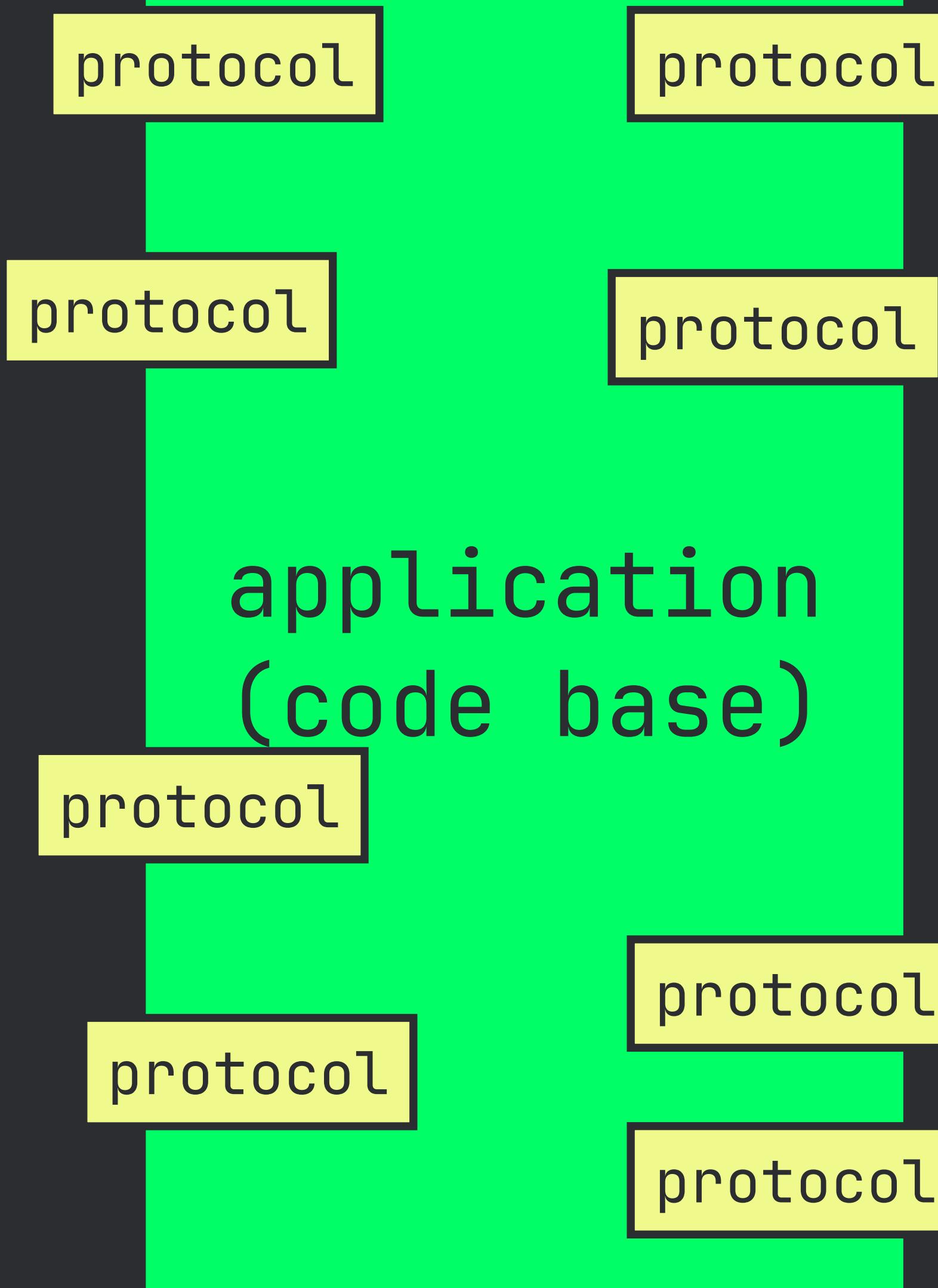
contract

protocol

DNS

HTTPS

so contract makes our code  
both maintainable (making  
changes) and extensible  
(adding things)



contract

protocol — DNS

protocol — HTTPS

**THE TAKEAWAY**

whenever we have a single generic feature with multiple options, or specific implementation might change



**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 1

## interfaces, config, + factory functions



# key concepts



- I said an interface is a contract
  - | but what's stated in the contract?
- It is a list of method signatures

# function vs method

a method “function that is  
attached to a specific type”

function

func DoSummin(args) return

method

func(type) DoSummin(args) return

interface = list of method signatures

any type that implements all of those  
methods “fulfills the contract”,  
“satisfies the interface”.

interface

```
type Server interface {  
    Start() error  
    Stop() error  
}
```



→ DNSServer struct

func(s \*DNSServer) Start() error

func(s \*DNSServer) Stop() error

satisfies

→ DNSServer

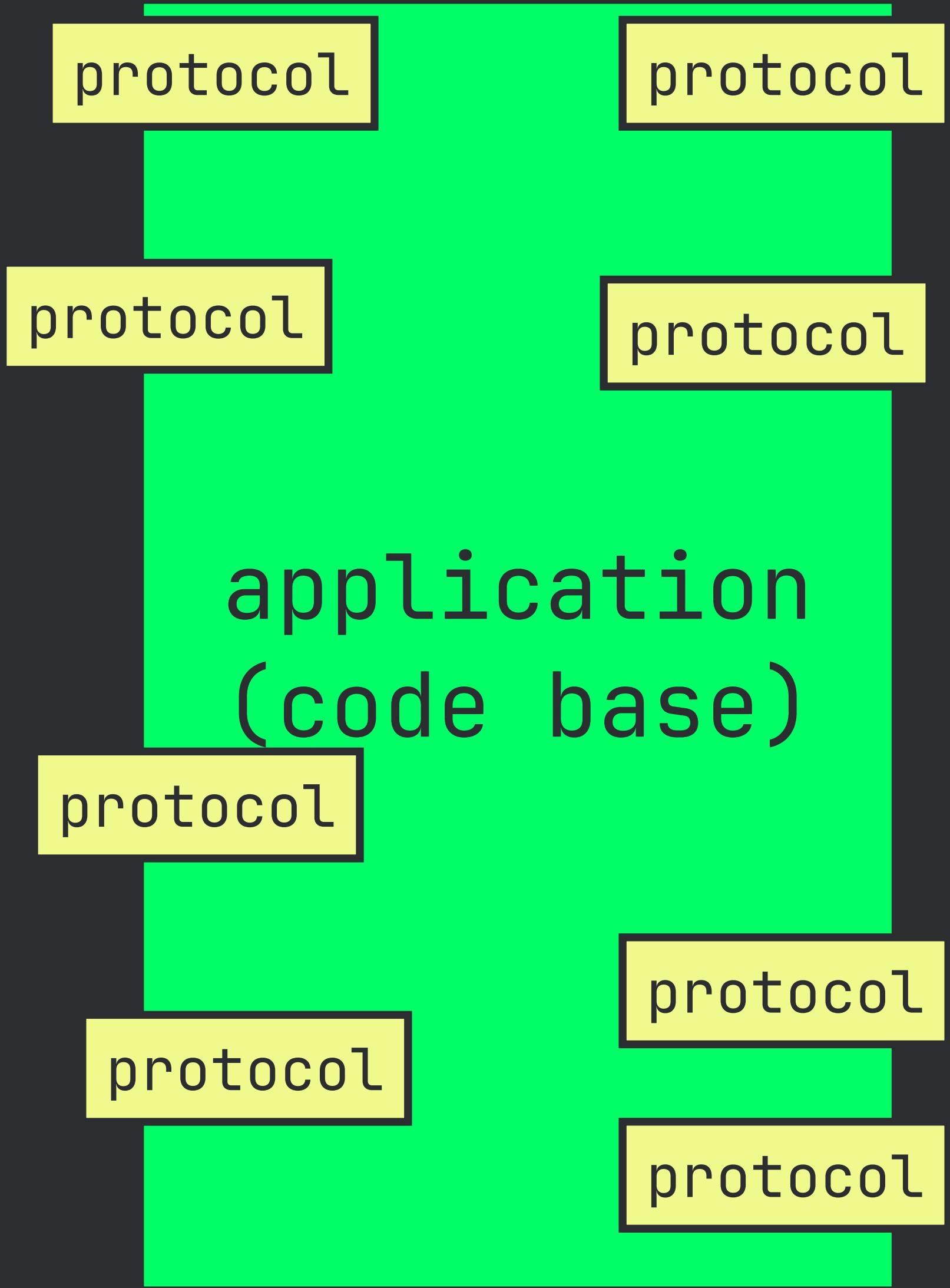
→ HTTPSServer



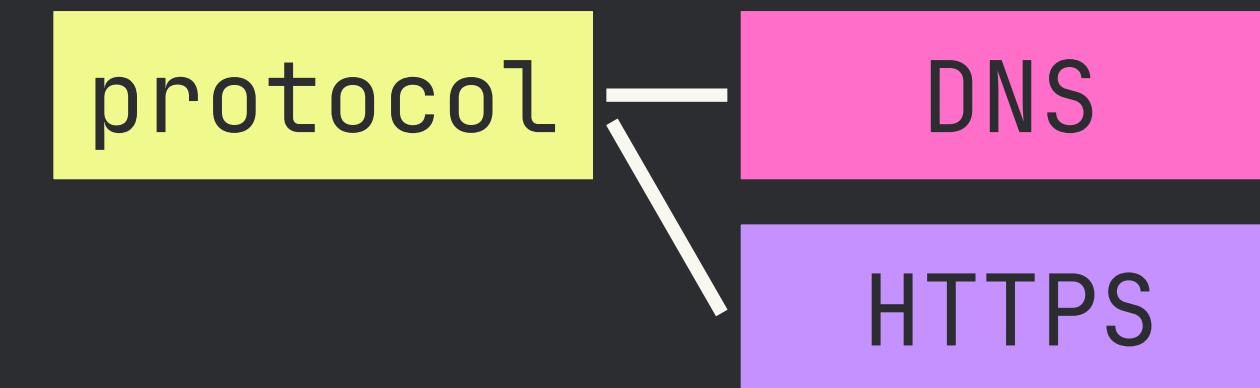
→ HTTPSServer struct

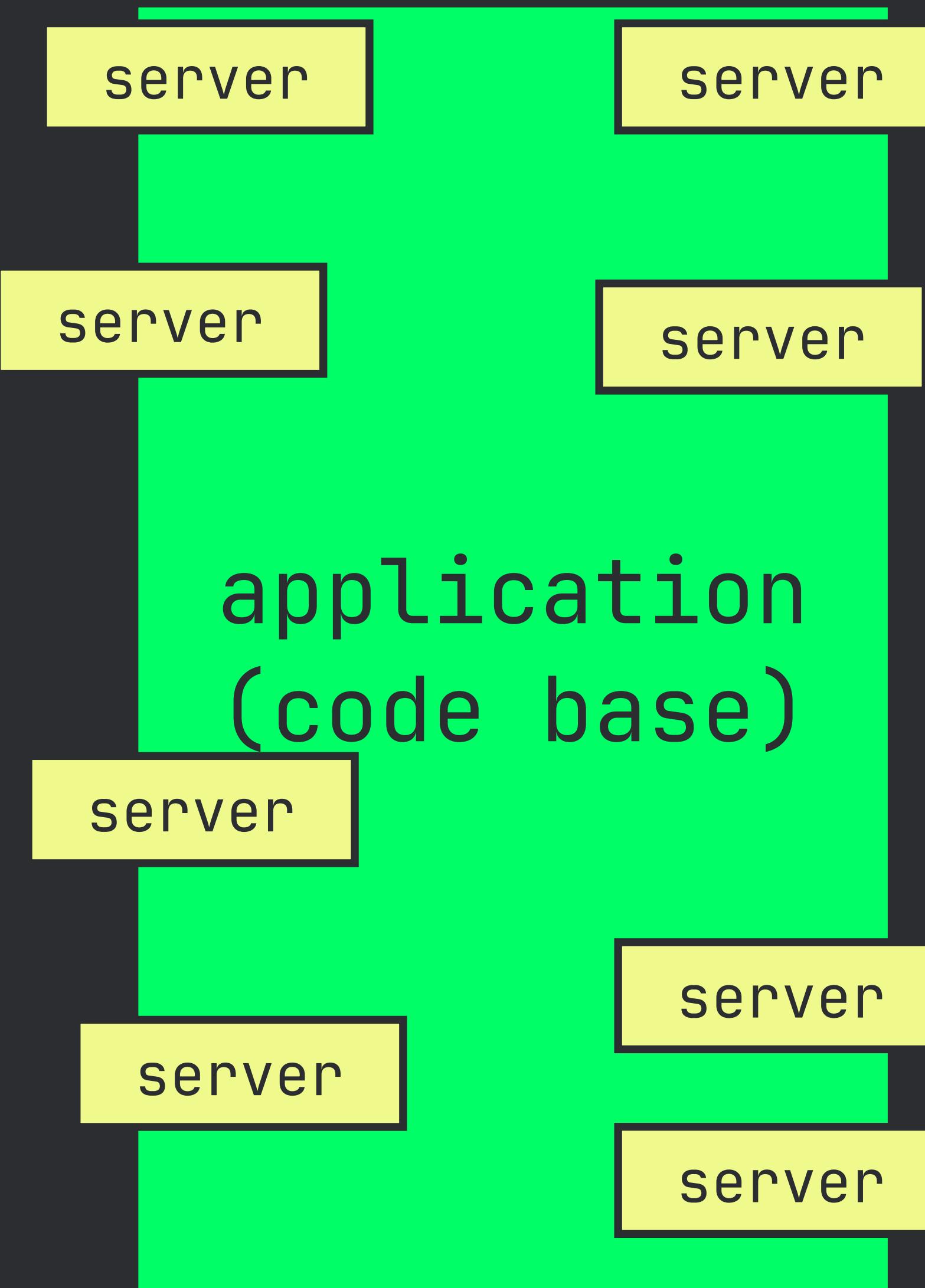
func(s \*HTTPSServer) Start() error

func(s \*HTTPSServer) Stop() error

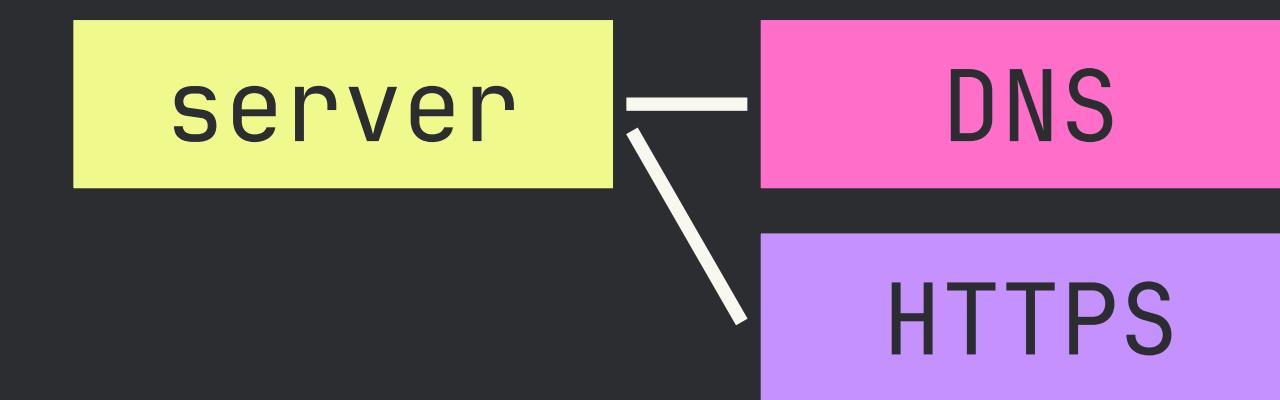


contract





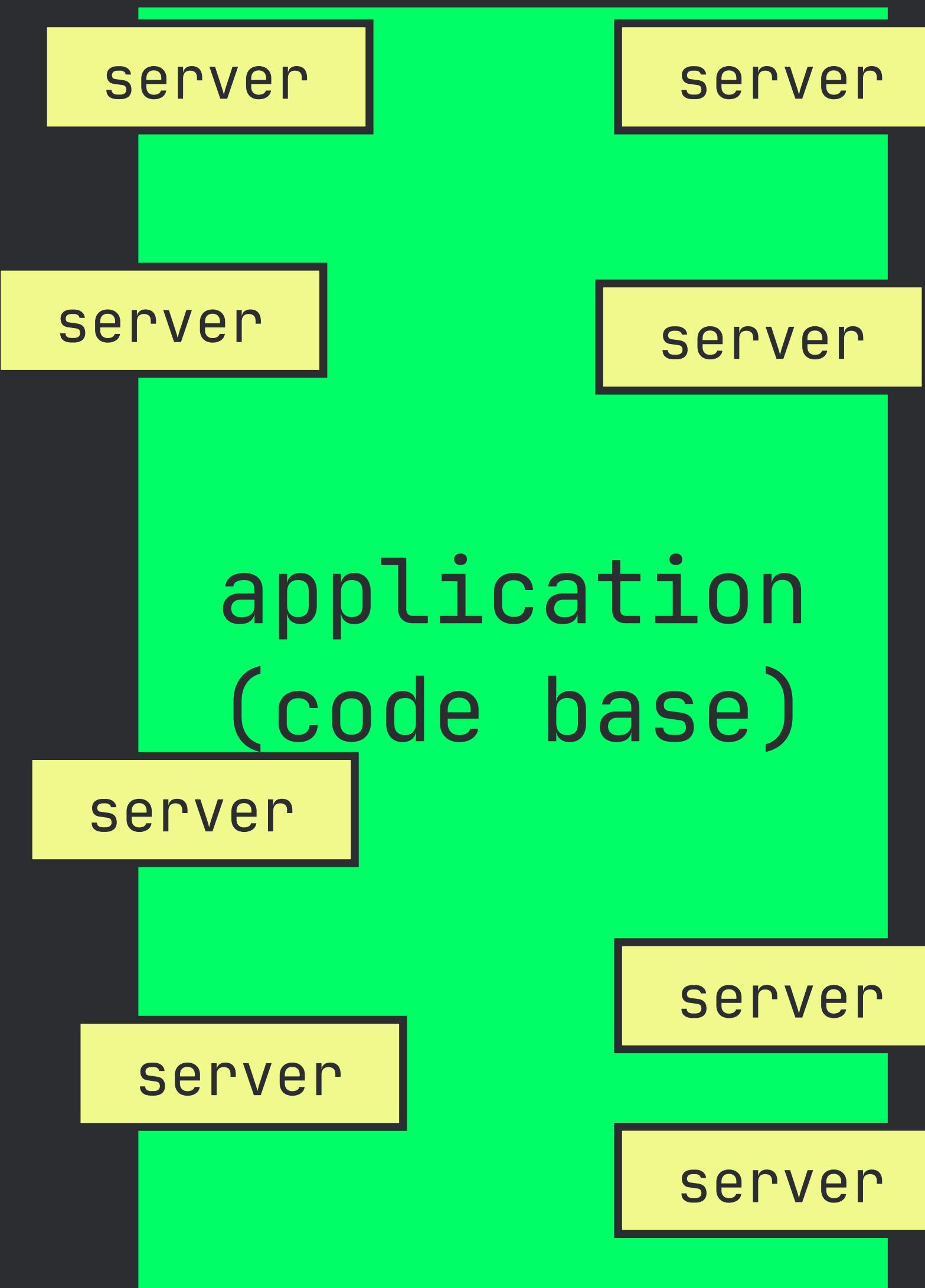
contract



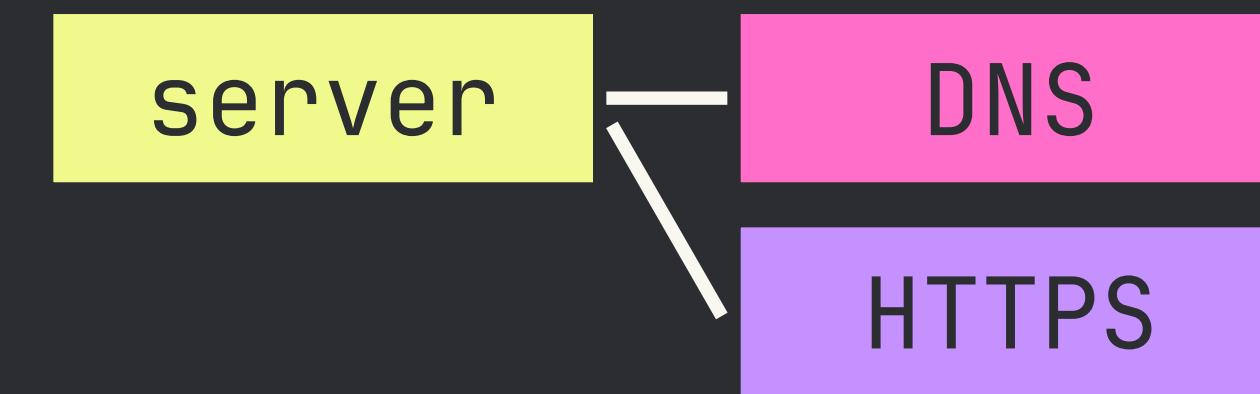
one final question...

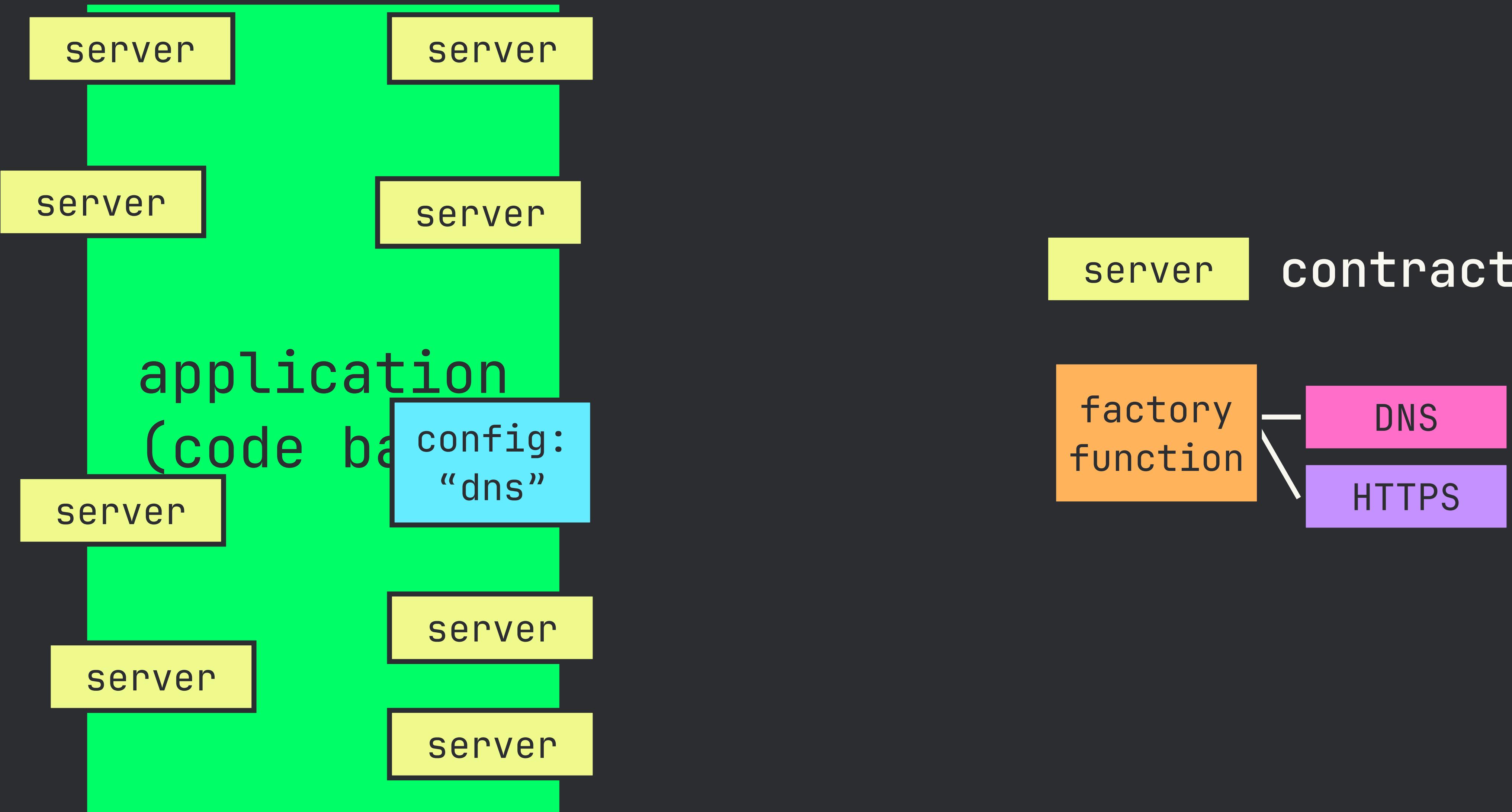
How does our application “know” which  
specific protocol to use?

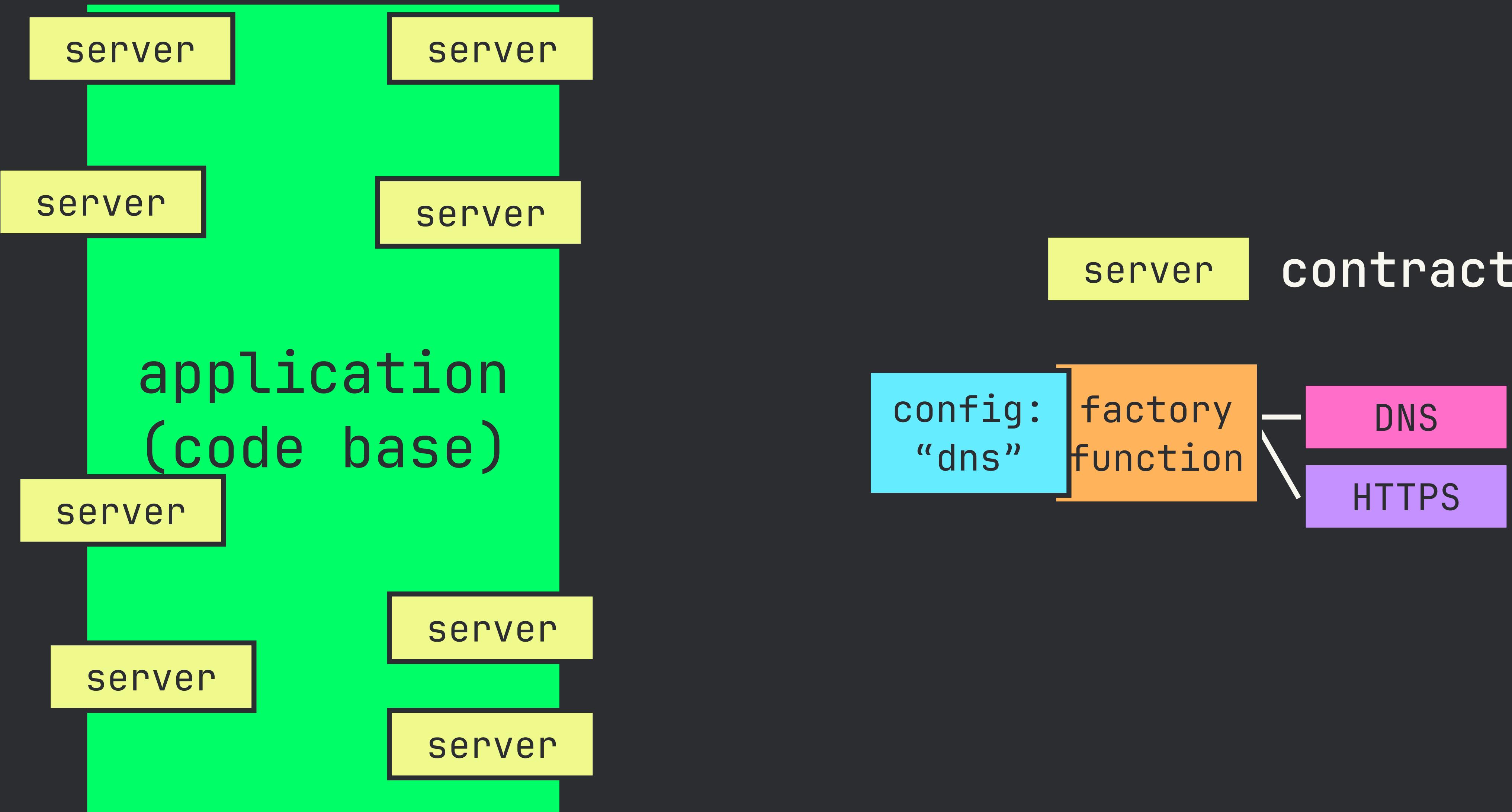
Factory Function

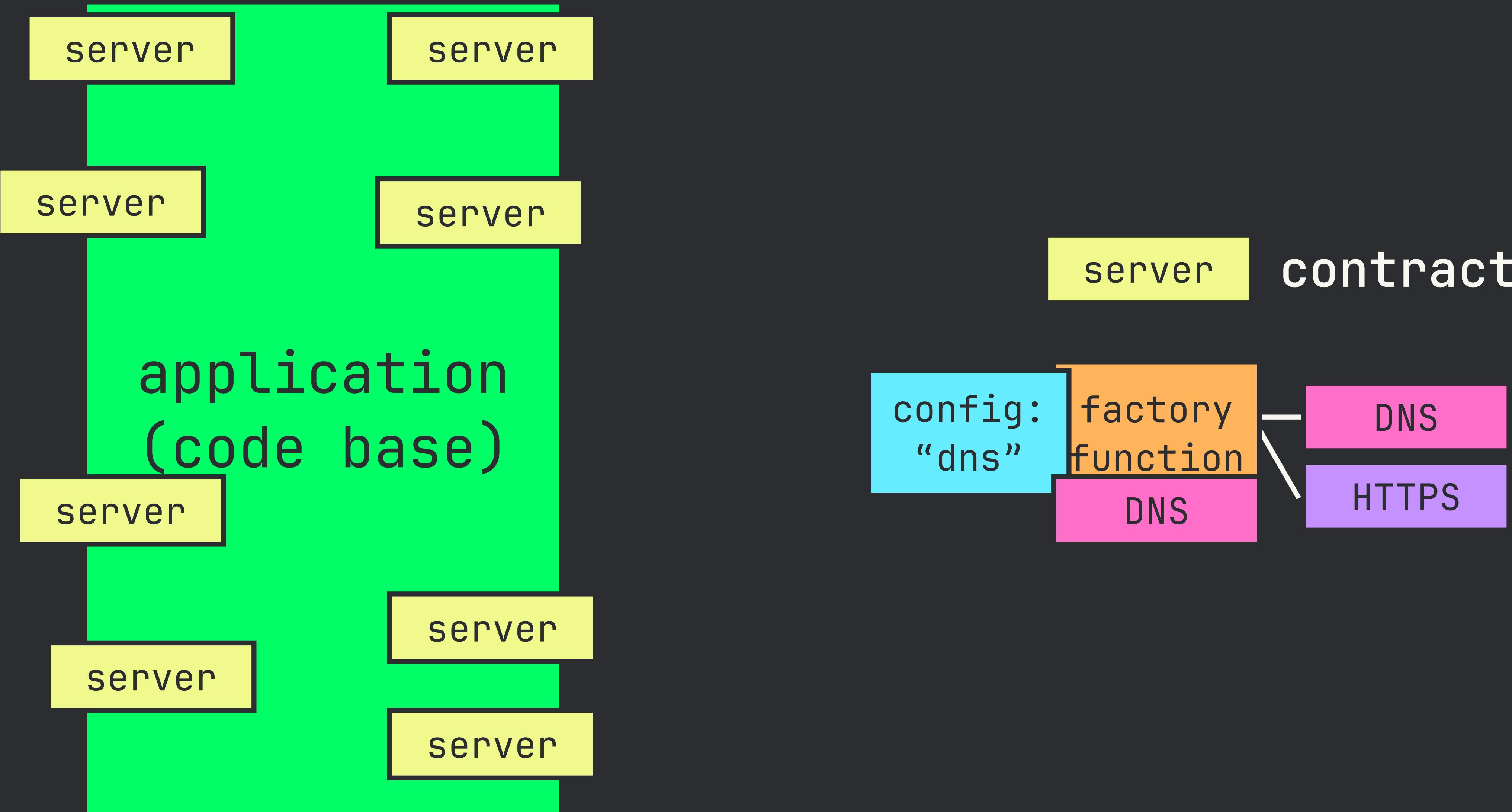


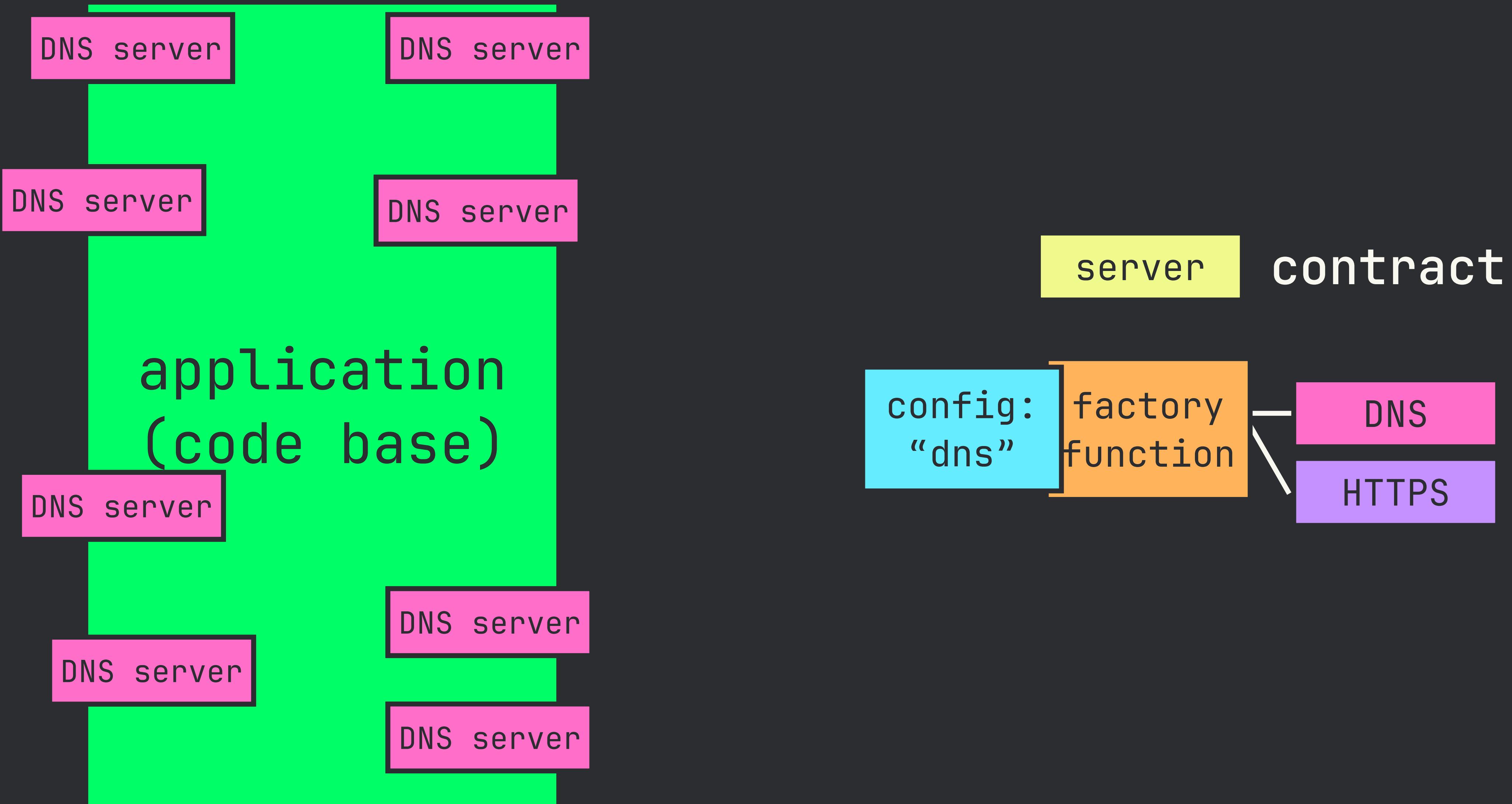
contract













what we'll create

project

package

files

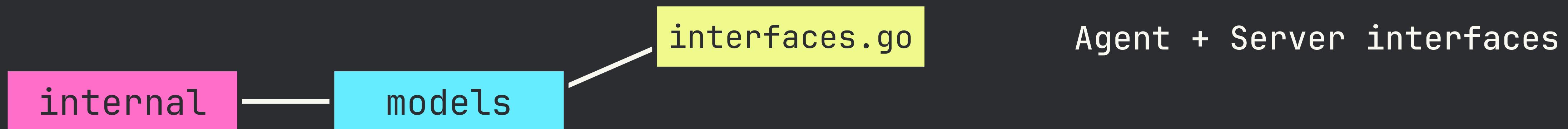
what we'll create

project

package

files

what we'll create

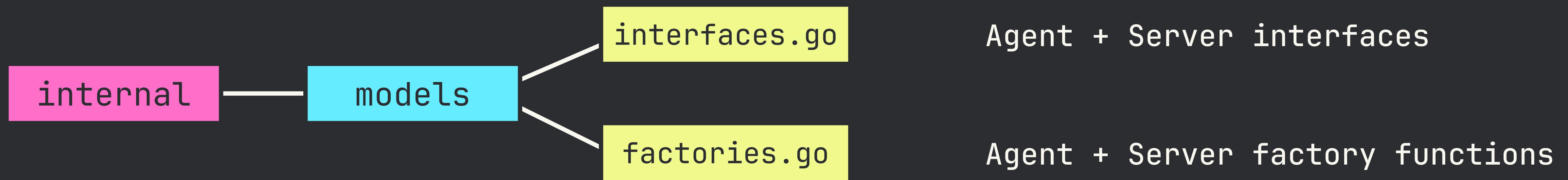


project

package

files

what we'll create

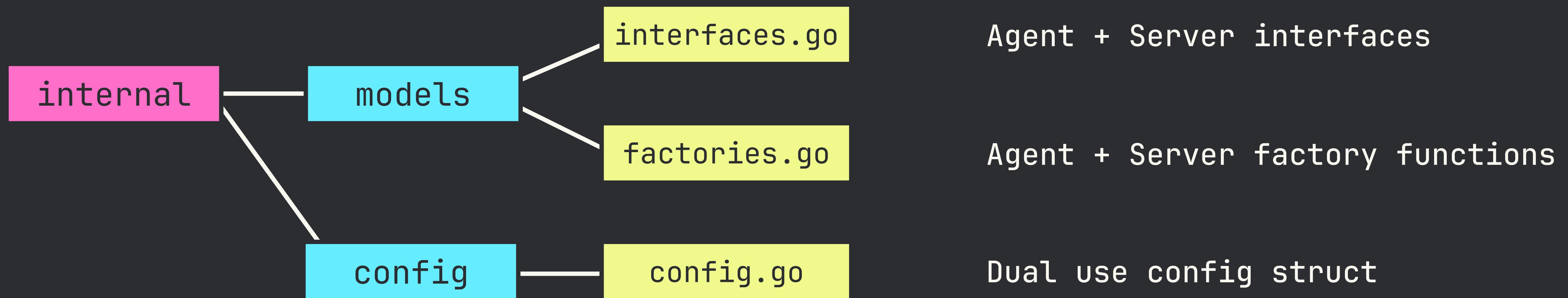


project

package

files

what we'll create

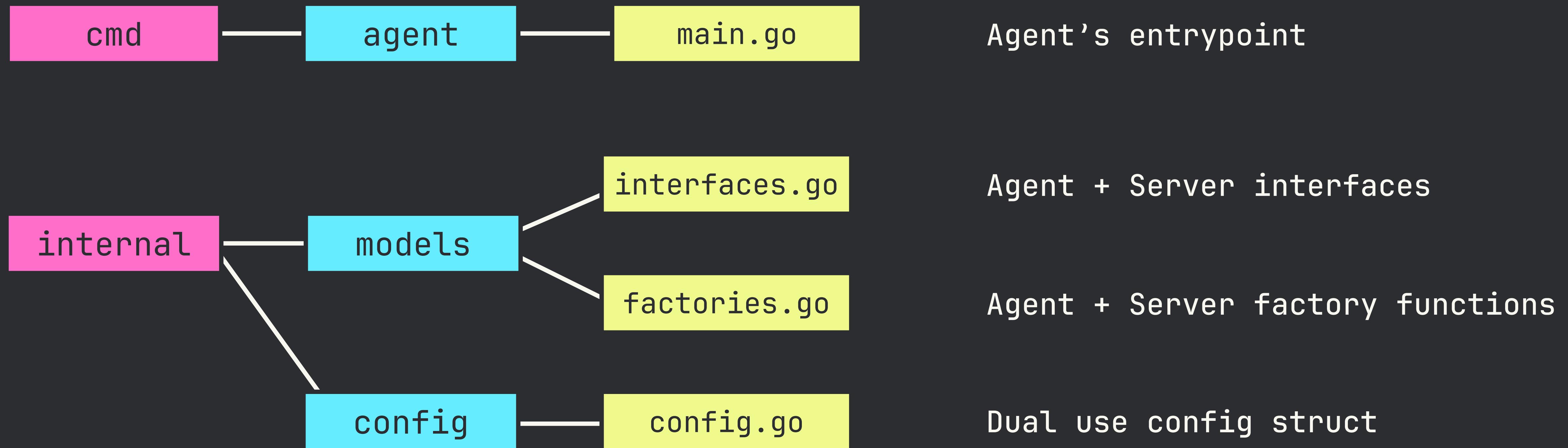


project

package

files

what we'll create



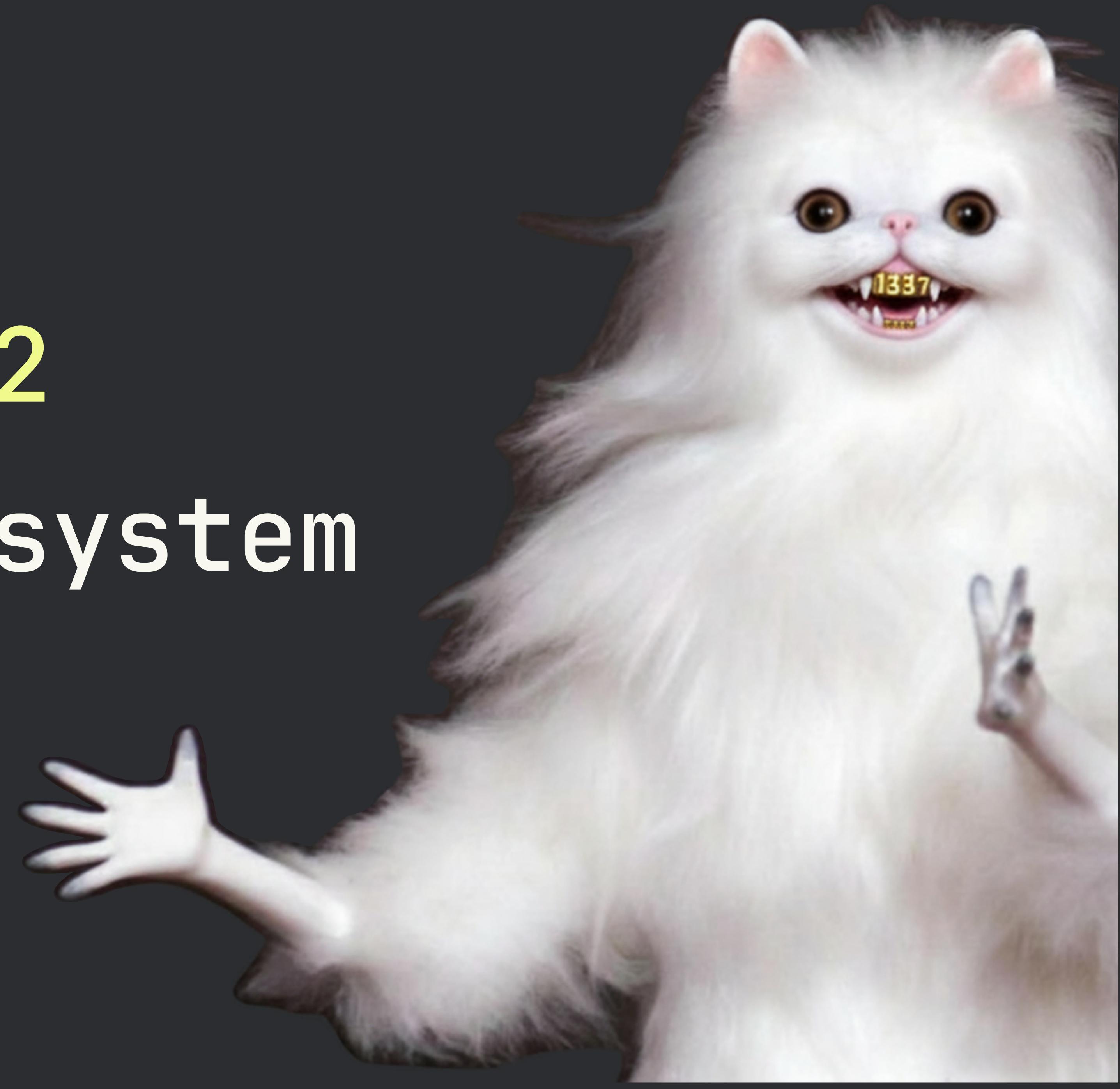




**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 2

# YAML config system



# key concepts



- We know we can have specific **types** in Go
  - | string, int, uint32, bool etc
- Most languages are of course like this
- Strong **vs** Dynamic **vs** Inference
- But in Go, we often use **structs**

- What is it? One way to create OUR OWN TYPE
- How? by creating a collection of other types
- We've already seen this - our Config!

```
type Config struct { faanross *  
    ClientAddr string  
    ServerAddr string  
    Timing      TimingConfig  
    Protocol   string  
    TlsKey     string  
    TlsCert    string  
}
```

```
type TimingConfig struct {  
    Delay  time.Duration  
    Jitter int  
}
```

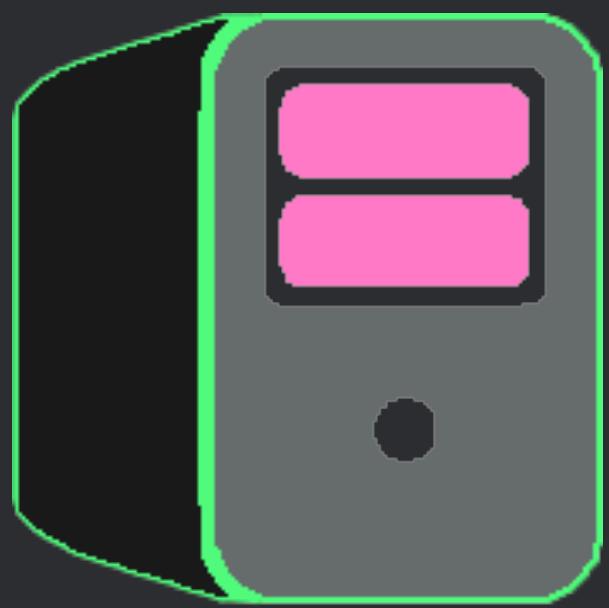
- We are going to use A LOT
- Everytime you think of an “object”

For ex: Agent, Server, Message

```
type Config struct { faanross *  
    ClientAddr string  
    ServerAddr string  
    Timing      TimingConfig  
    Protocol   string  
    TlsKey     string  
    TlsCert    string  
}
```

```
type TimingConfig struct {  
    Delay  time.Duration  
    Jitter int  
}
```

server

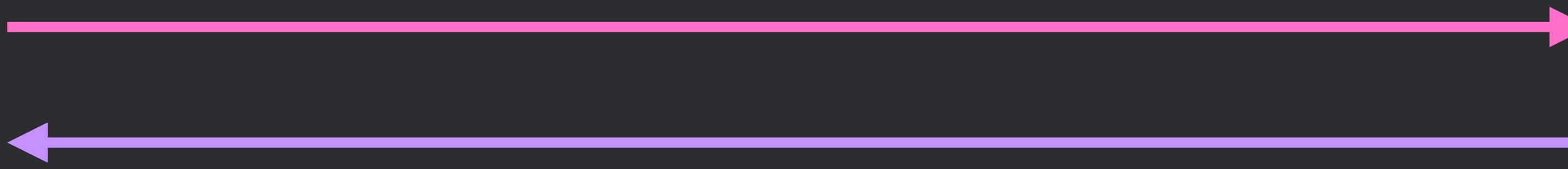


agent

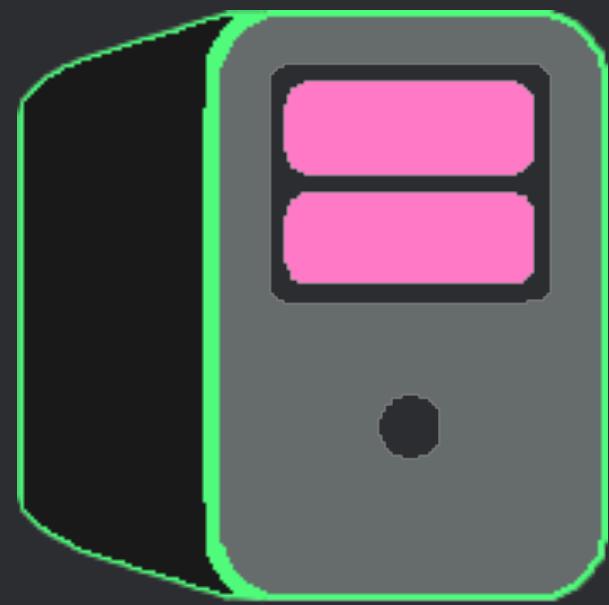


response

request



 server



response

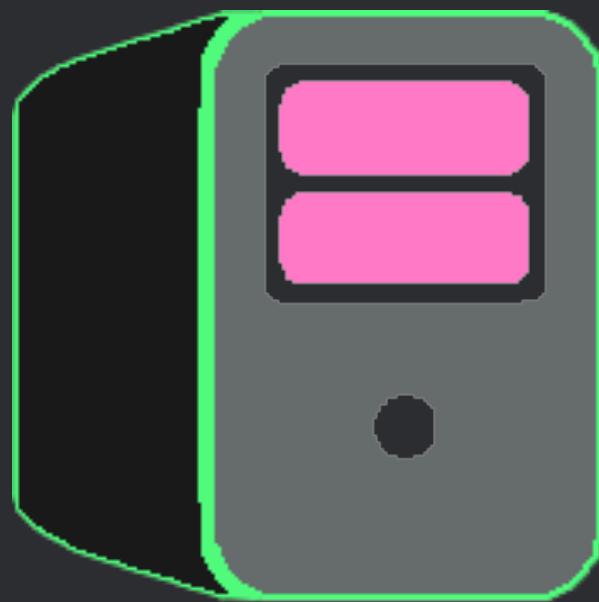
 agent



request

request  
struct

 server

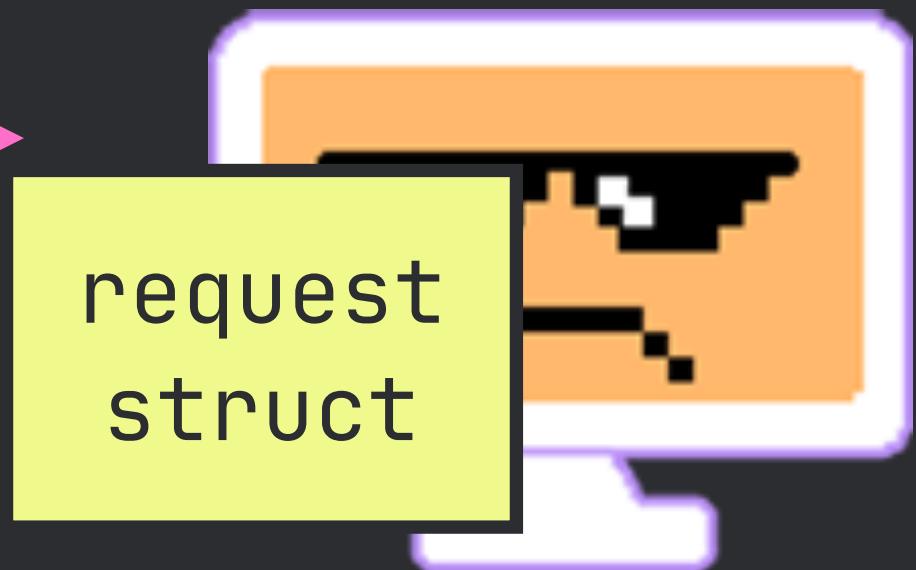


response

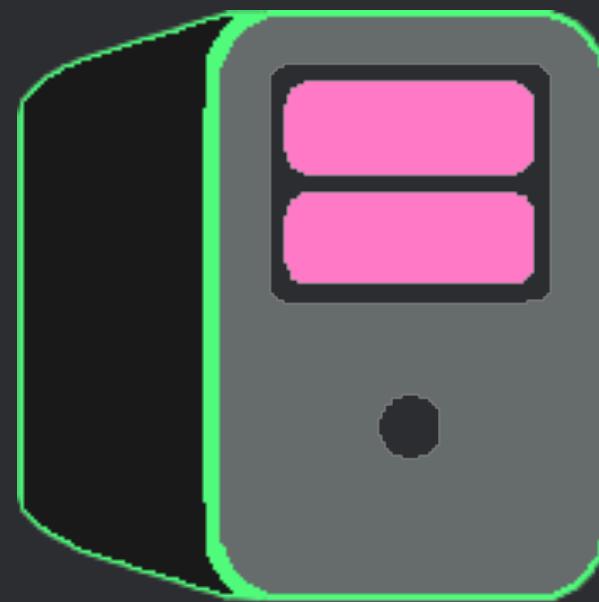


request

 agent



request  
struct



response



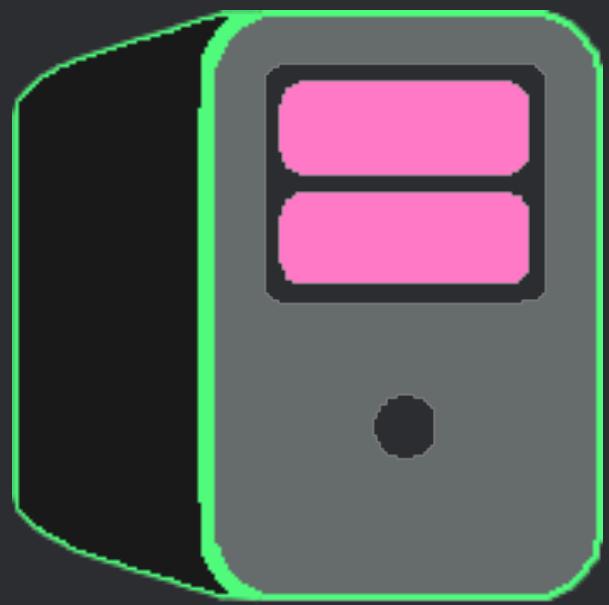
request

→ marshall

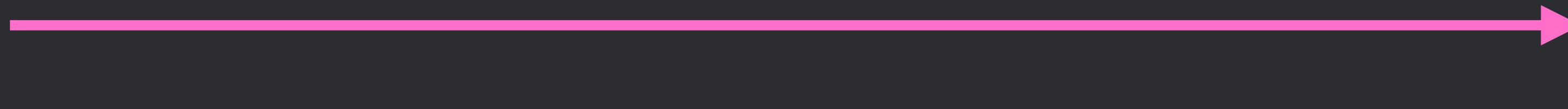
→ serialize

→ encode

 server



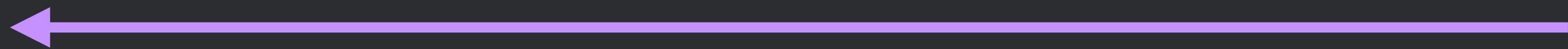
response



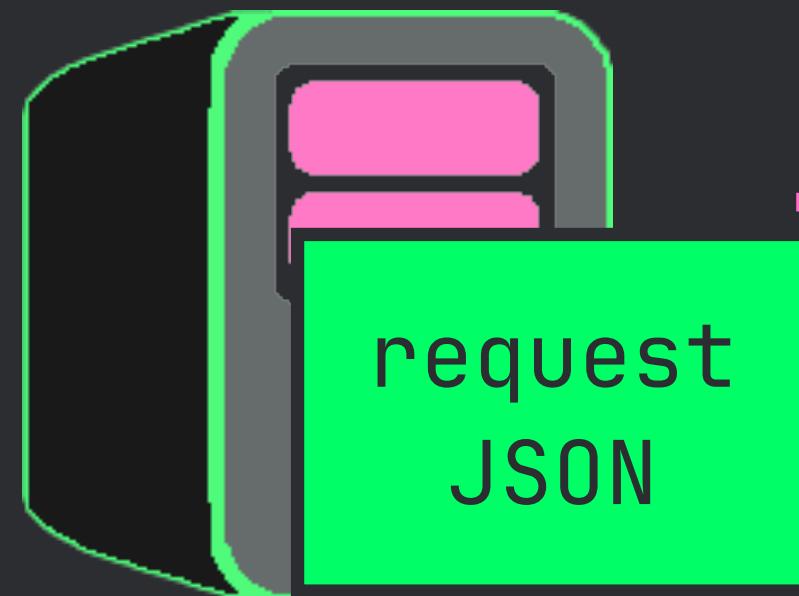
 agent



request



 server

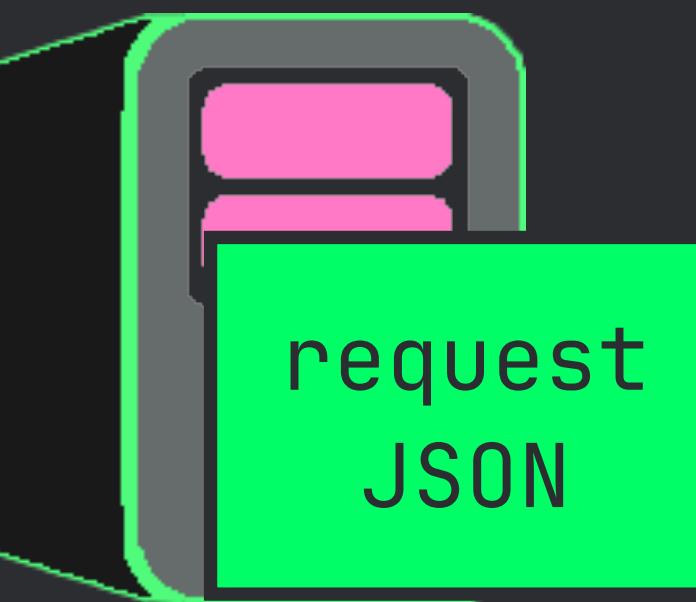


response

request

 agent



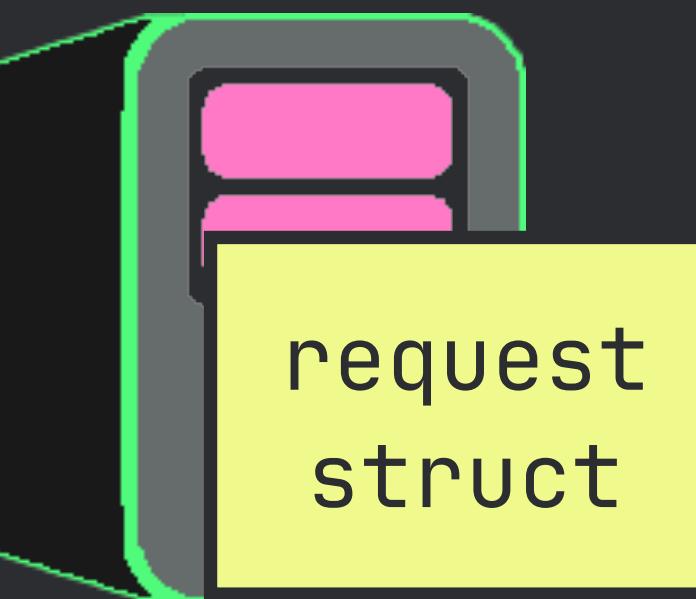


response



request

- unmarshal
- deserialize
- decode

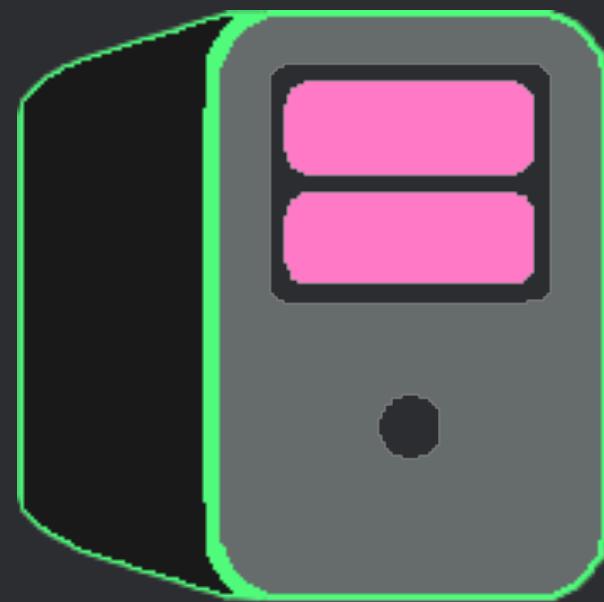


response

request



- unmarshal
- deserialize
- decode



response

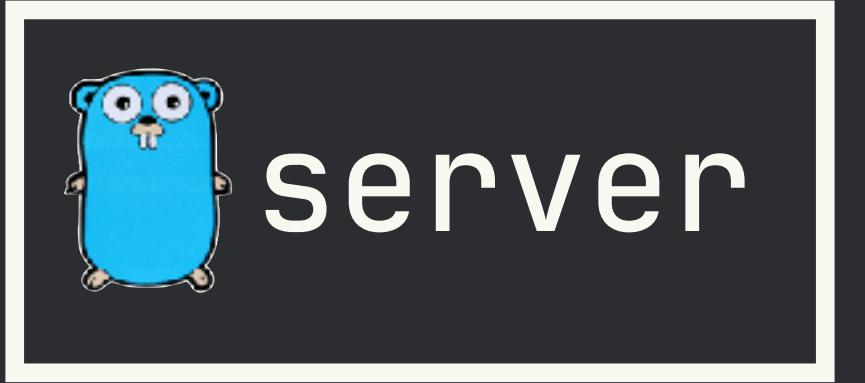


request

- So “inside” of Go we use structs
- Convert to/fro JSON when sending over wire

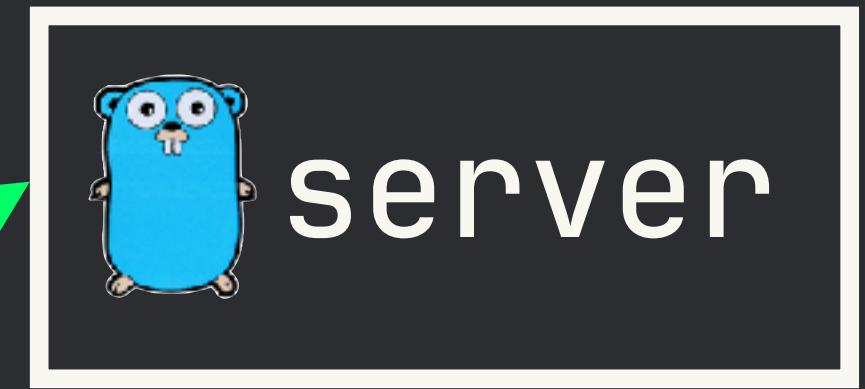
## in this lesson

- A third format: YAML
- This is used for UX
- It's a user-friendly way to provide values



# struct

```
type Config struct { faanross *
    ClientAddr string
    ServerAddr string
    Timing     TimingConfig
    Protocol   string
    TlsKey     string
    TlsCert    string
}
```



# YAML

```
client: "127.0.0.1:0"

server: "127.0.0.1:8443"

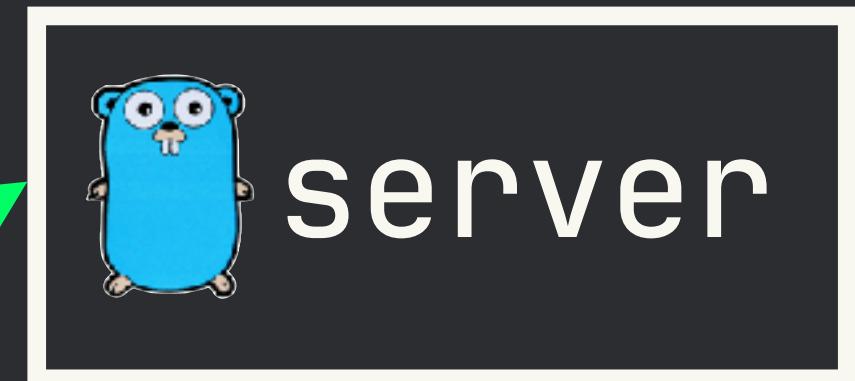
timing:
  delay: "5s"
  jitter: 50

protocol: "https"

tls_key: "./certs/server.key"
tls_cert: "./certs/server.crt"
```

## struct

```
type Config struct { faanross *
  ClientAddr string
  ServerAddr string
  Timing     TimingConfig
  Protocol   string
  TlsKey     string
  TlsCert    string
}
```



# YAML 1

```
client: "127.0.0.1:0"

server: "127.0.0.1:8443"

timing:
  delay: "5s"
  jitter: 50

protocol: "https"

tls_key: "./certs/server.key"
tls_cert: "./certs/server.crt"
```

# YAML 2

```
client: "127.0.0.1:0"

server: "127.0.0.1:8443"

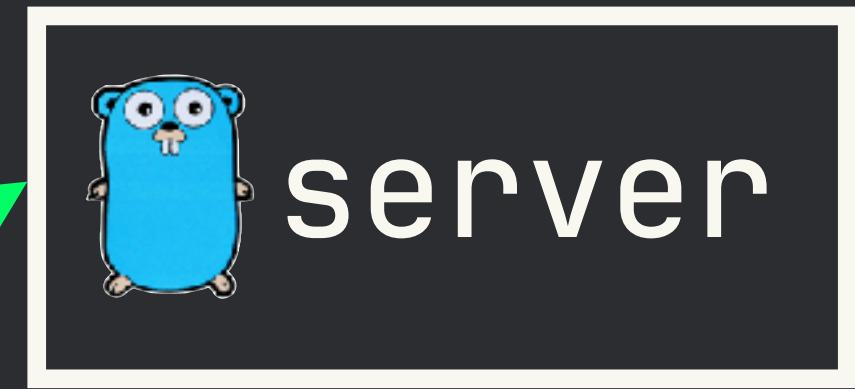
timing:
  delay: "5s"
  jitter: 50

protocol: "https"

tls_key: "./certs/server.key"
tls_cert: "./certs/server.crt"
```

## struct

```
type Config struct { faanross *
  ClientAddr string
  ServerAddr string
  Timing     TimingConfig
  Protocol   string
  TlsKey     string
  TlsCert    string
}
```





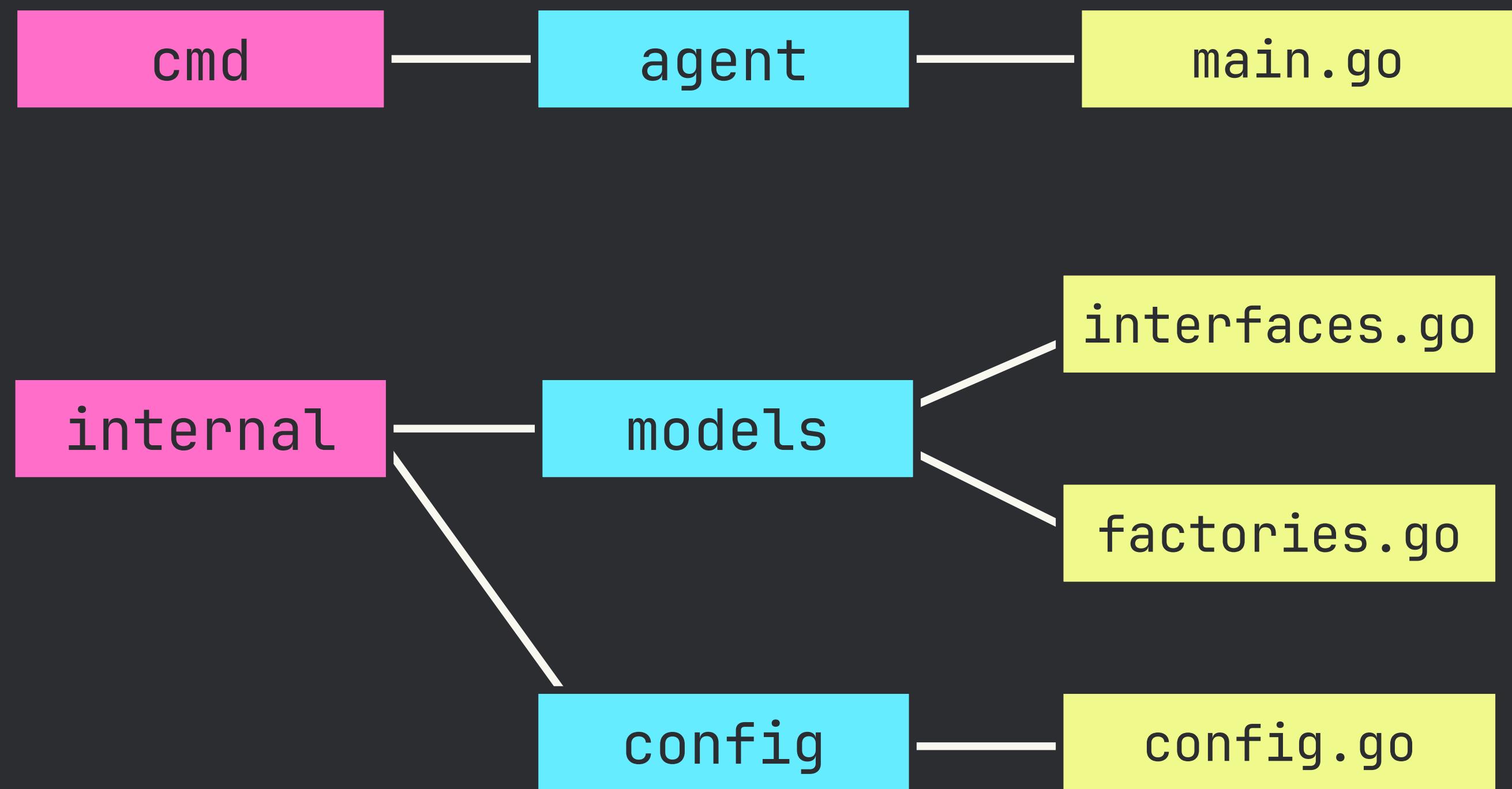
what we'll create

project

package

files

what we'll create

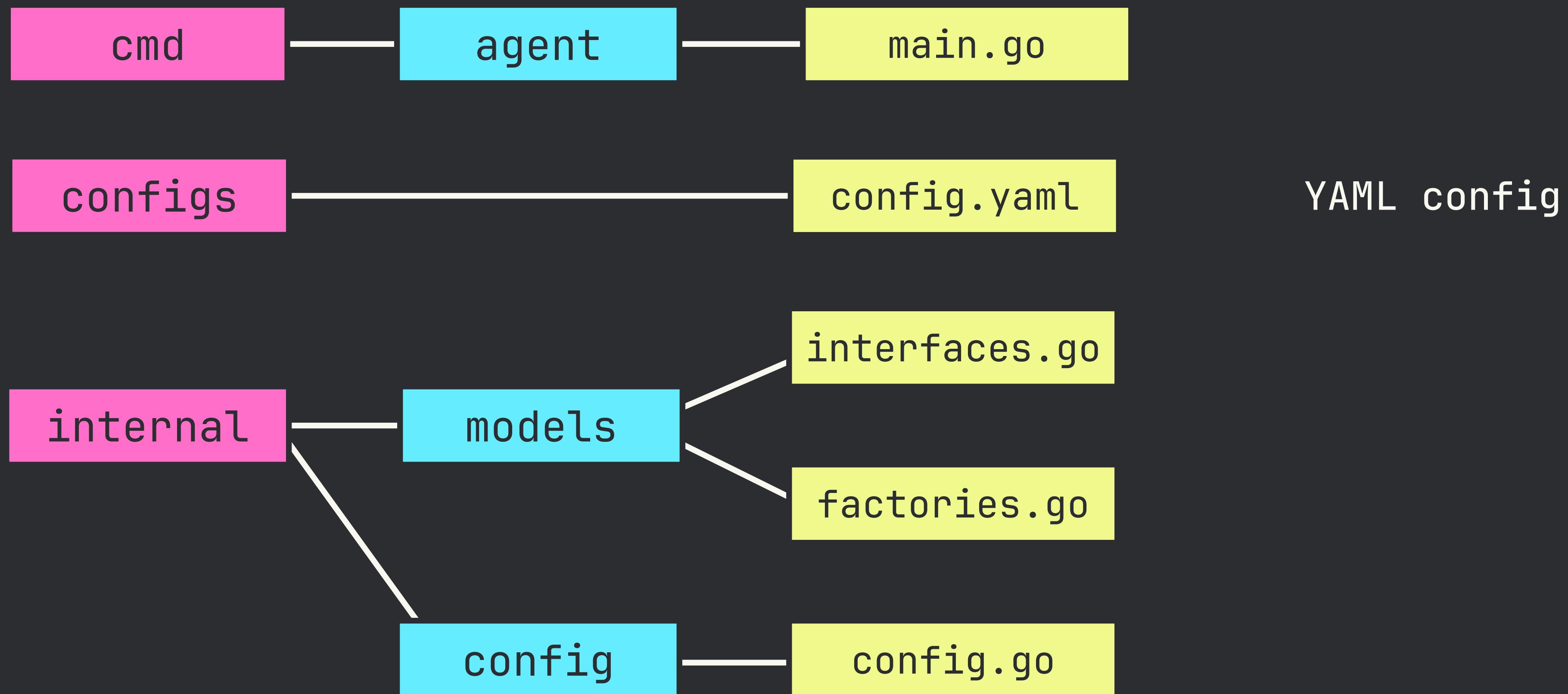


project

package

files

what we'll create

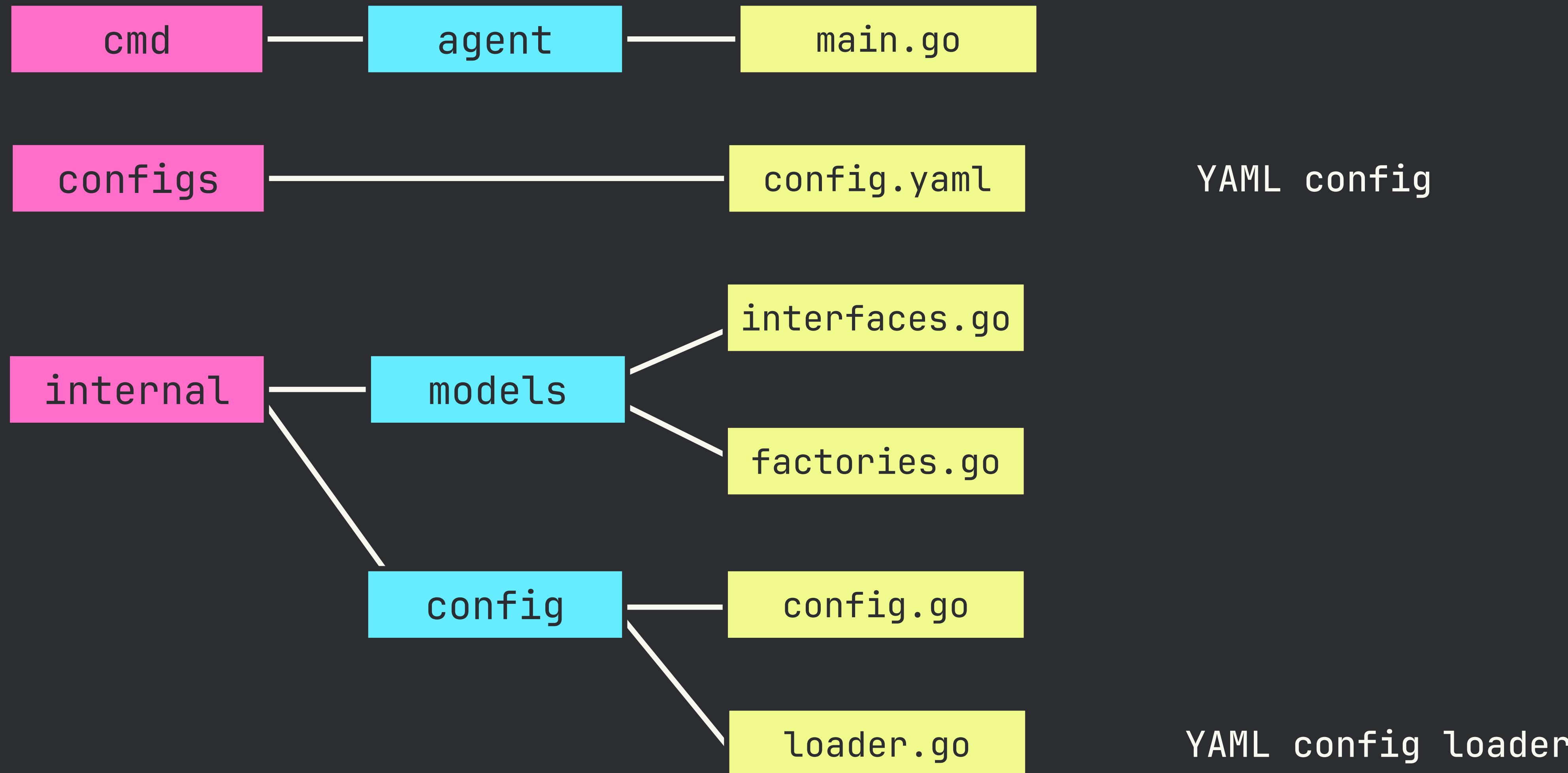


project

package

files

what we'll create







**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 3

## https server



# just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

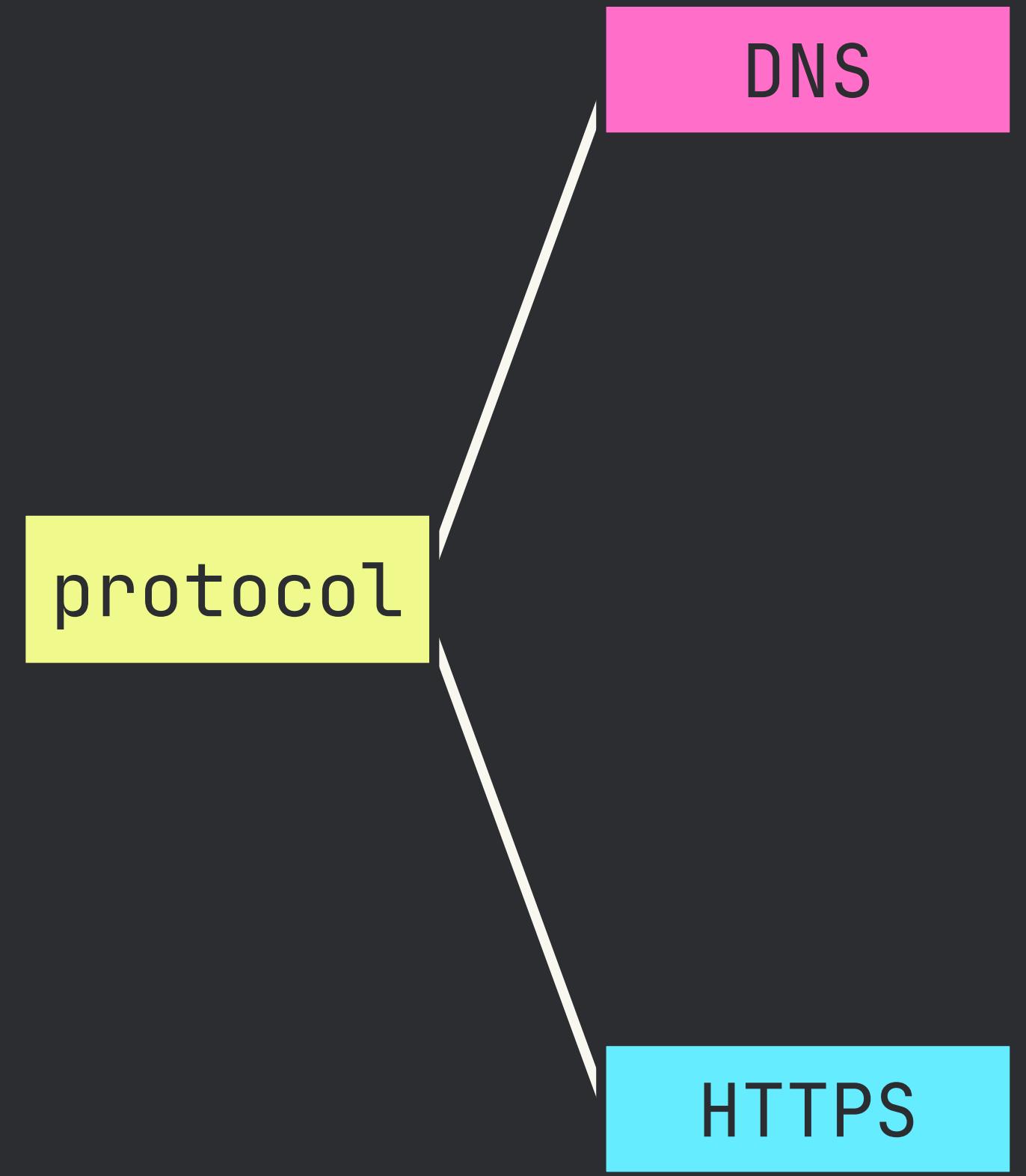
DNS

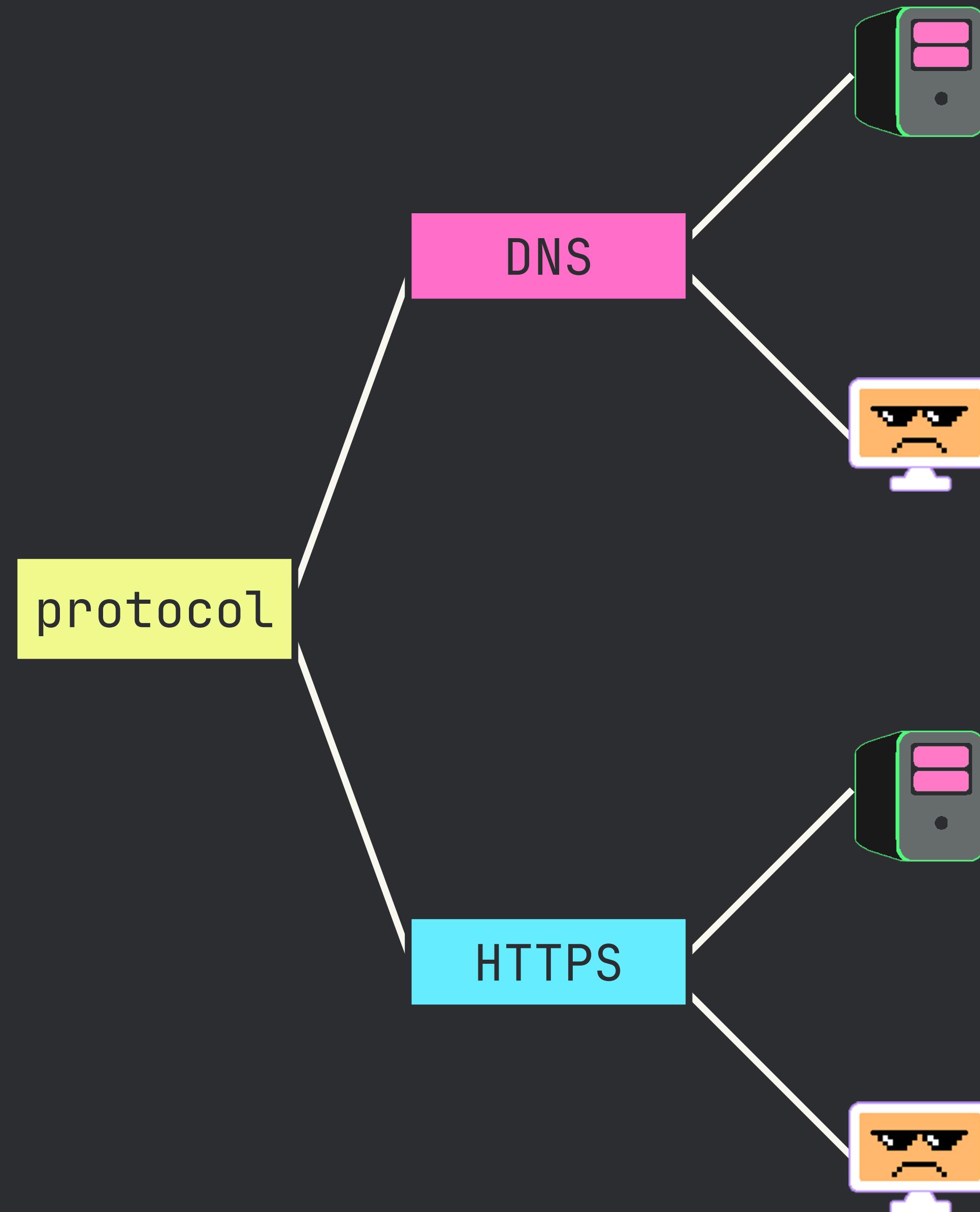
- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

# key concepts

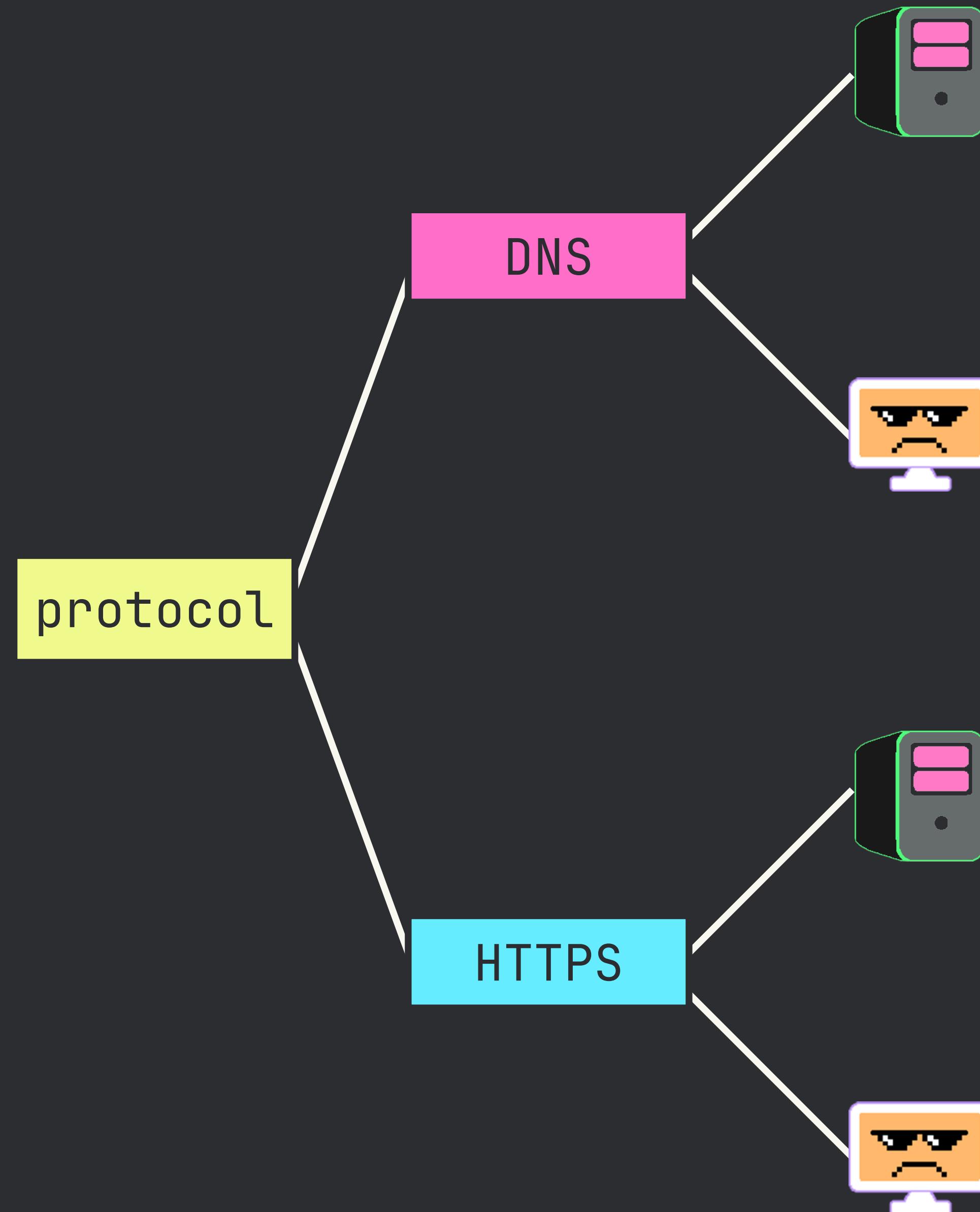


protocol





we have 4  
→ dns server  
→ dns agent  
→ https server  
→ https agent

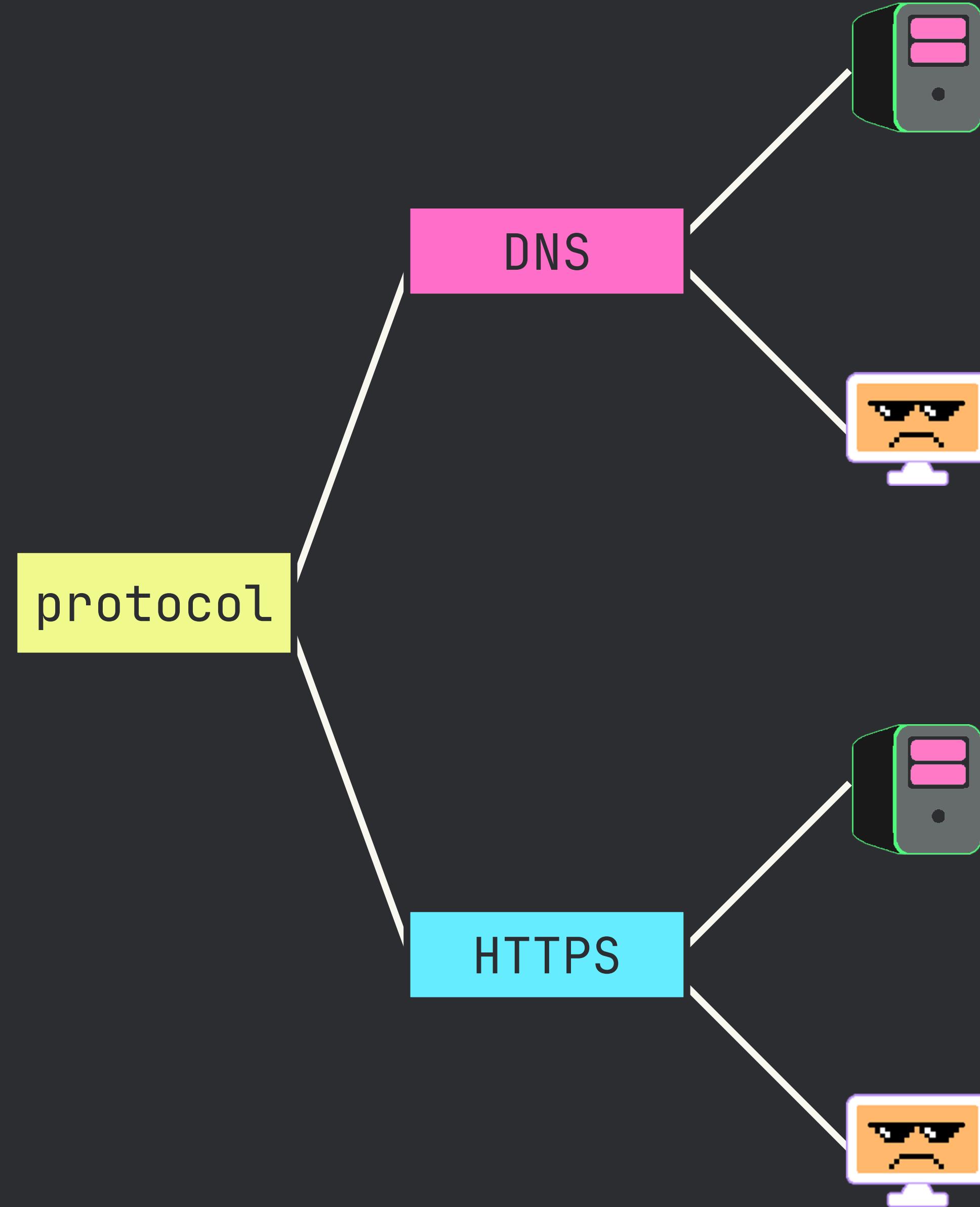


each of the 4 have:

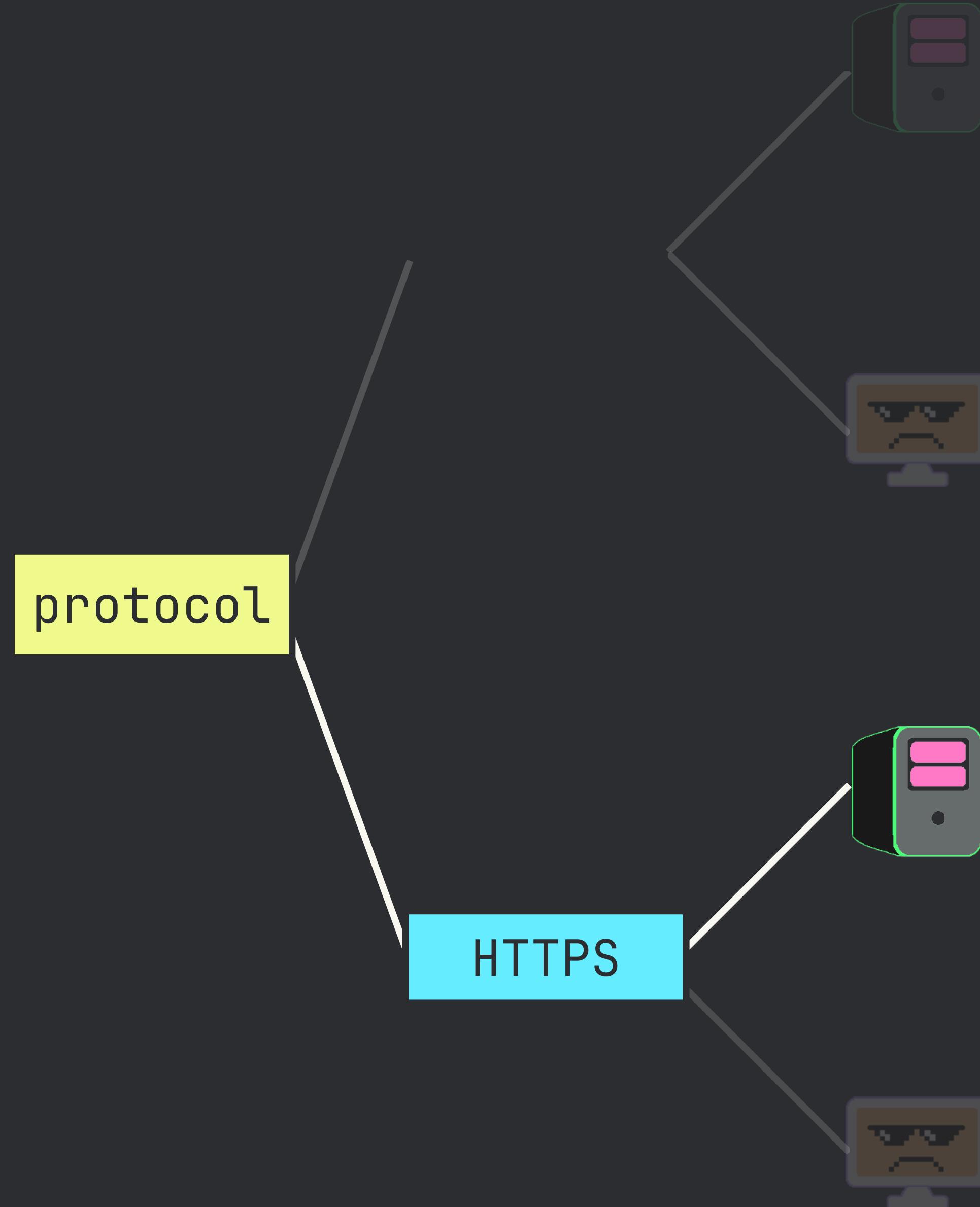
→ **struct**

→ **constructor**

→ **methods**



what methods?  
methods to satisfy  
the interface!



```
type Server interface {  
    Start() error  
    Stop() error  
}
```

# https server

→ struct

HTTPSServer

→ constructor

NewHTTPSServer()

→ methods

Start()

Stop()

```
type Server interface {  
    Start() error  
    Stop() error  
}
```

relationship  
between them

HTTPSServer

NewHTTPSServer()

Start()

Stop()

NewHTTPSServer()

HTTPSServer

Start()

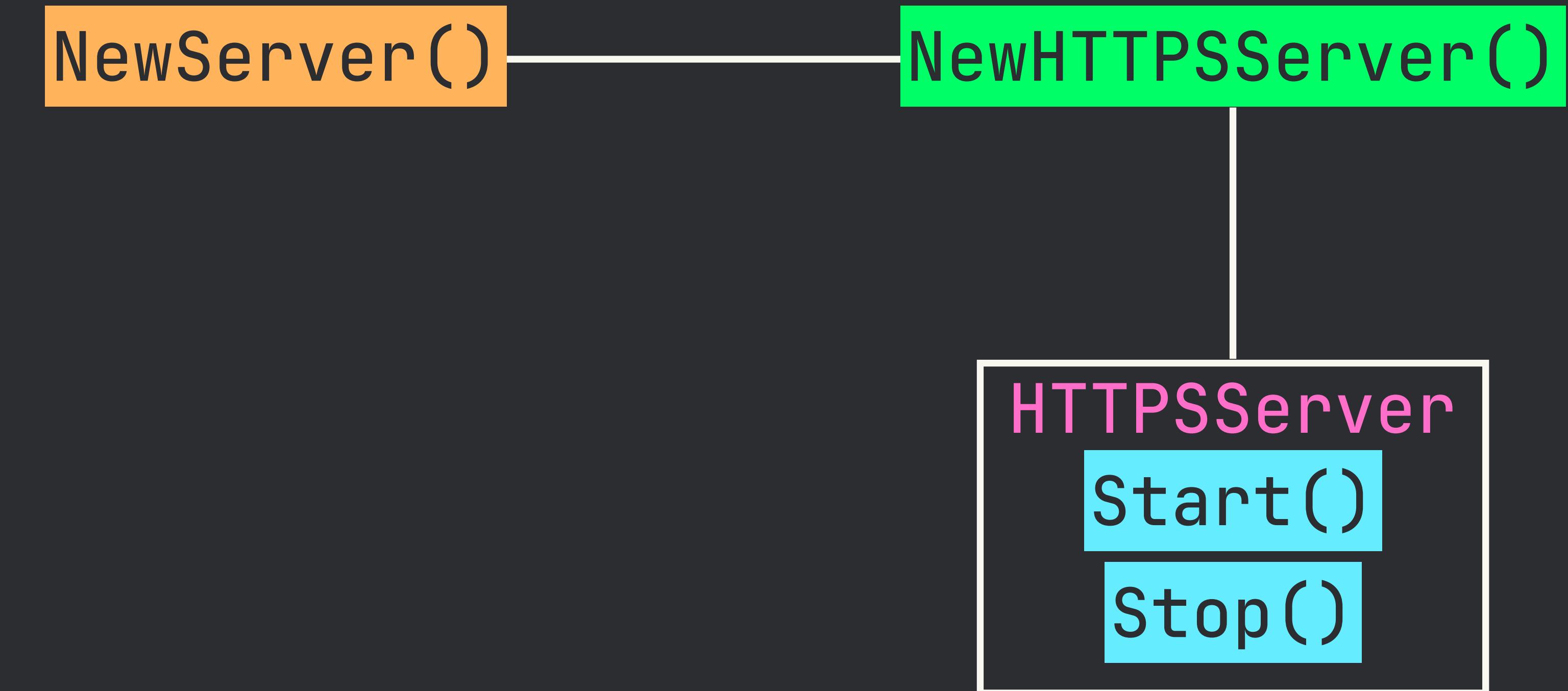
Stop()

NewHTTPSServer()

HTTPSServer

Start()

Stop()



config

NewServer()

NewHTTPSServer()

HTTPSServer

Start()

Stop()

config

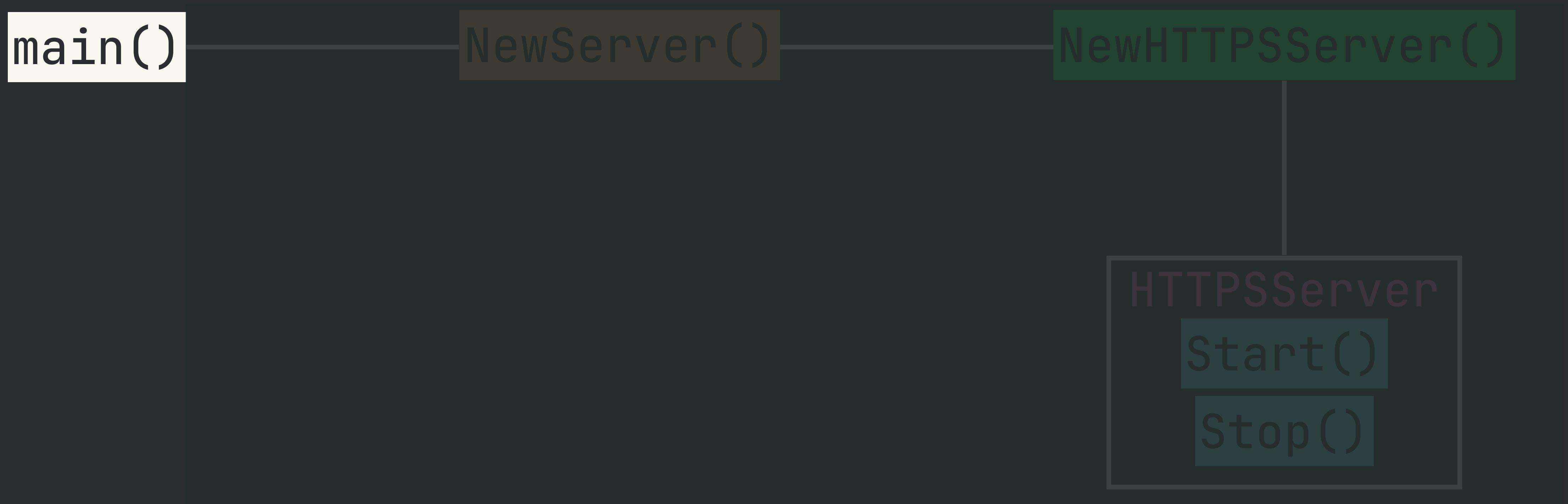
main()

NewServer()

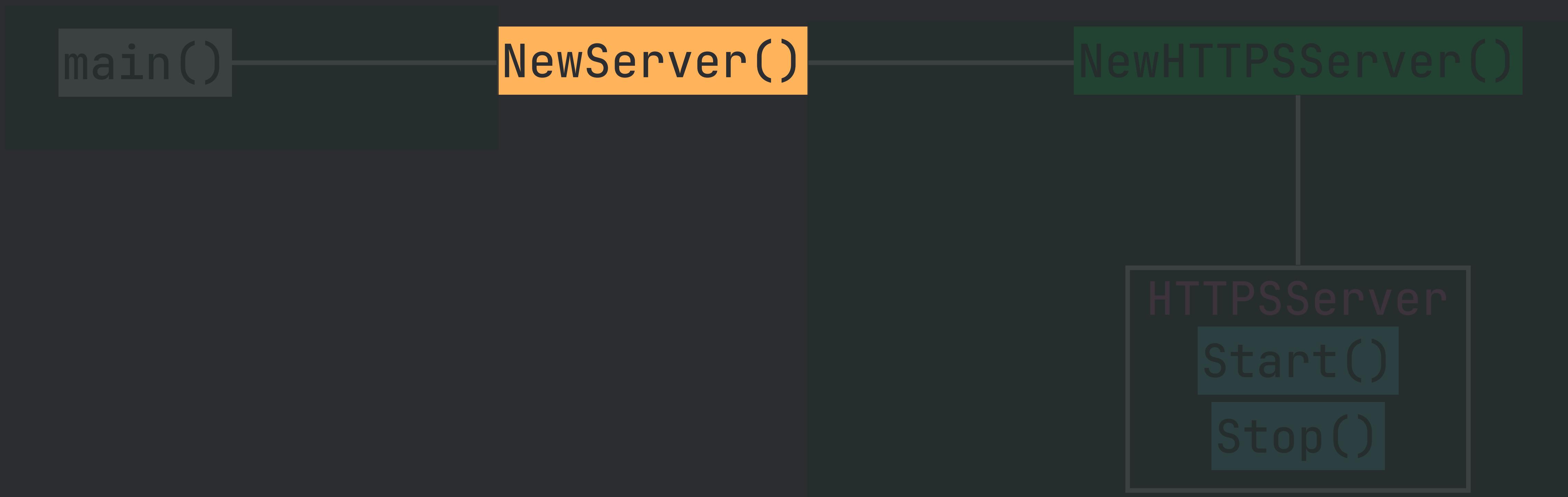
NewHTTPSServer()

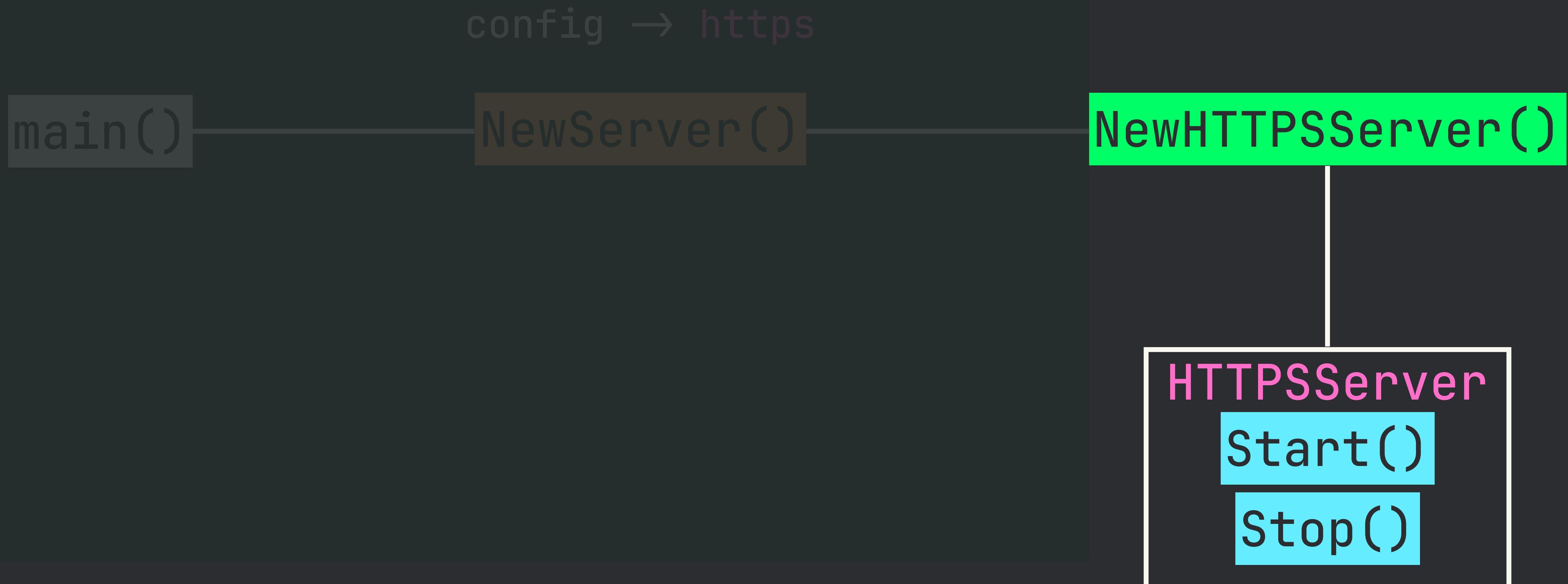
HTTPSServer  
Start()  
Stop()

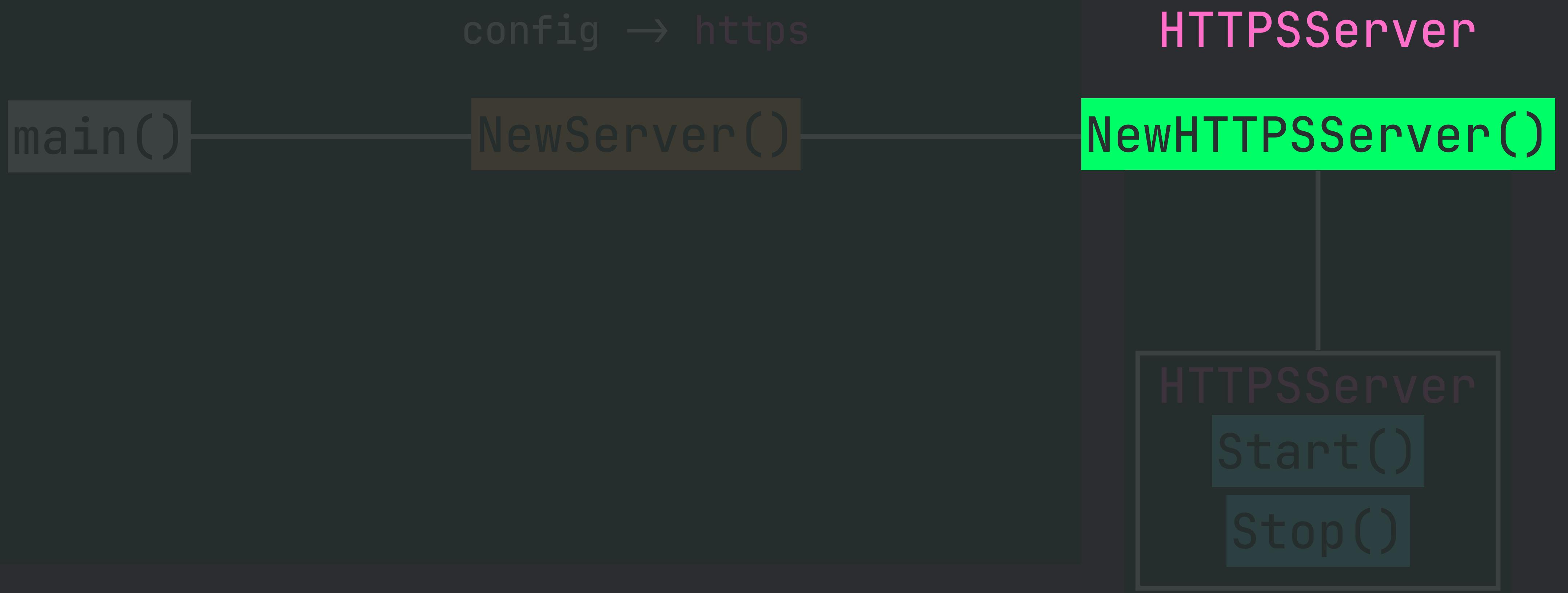
config → https



config → https







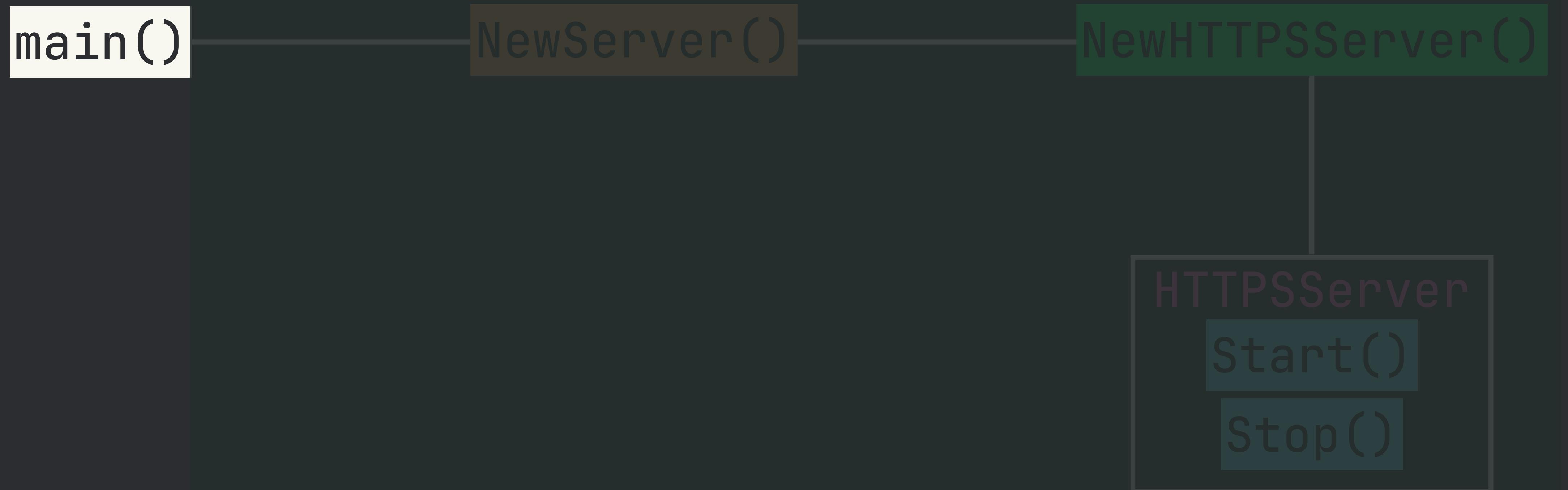
main()

HTTPSServer  
NewServer()

NewHTTPSServer()

HTTPSServer  
Start()  
Stop()

# HTTPSServer



**HTTPSServer**

Start()

Stop()

main()

NewServer()

NewHTTPSServer()

**HTTPSServer**

Start()

Stop()



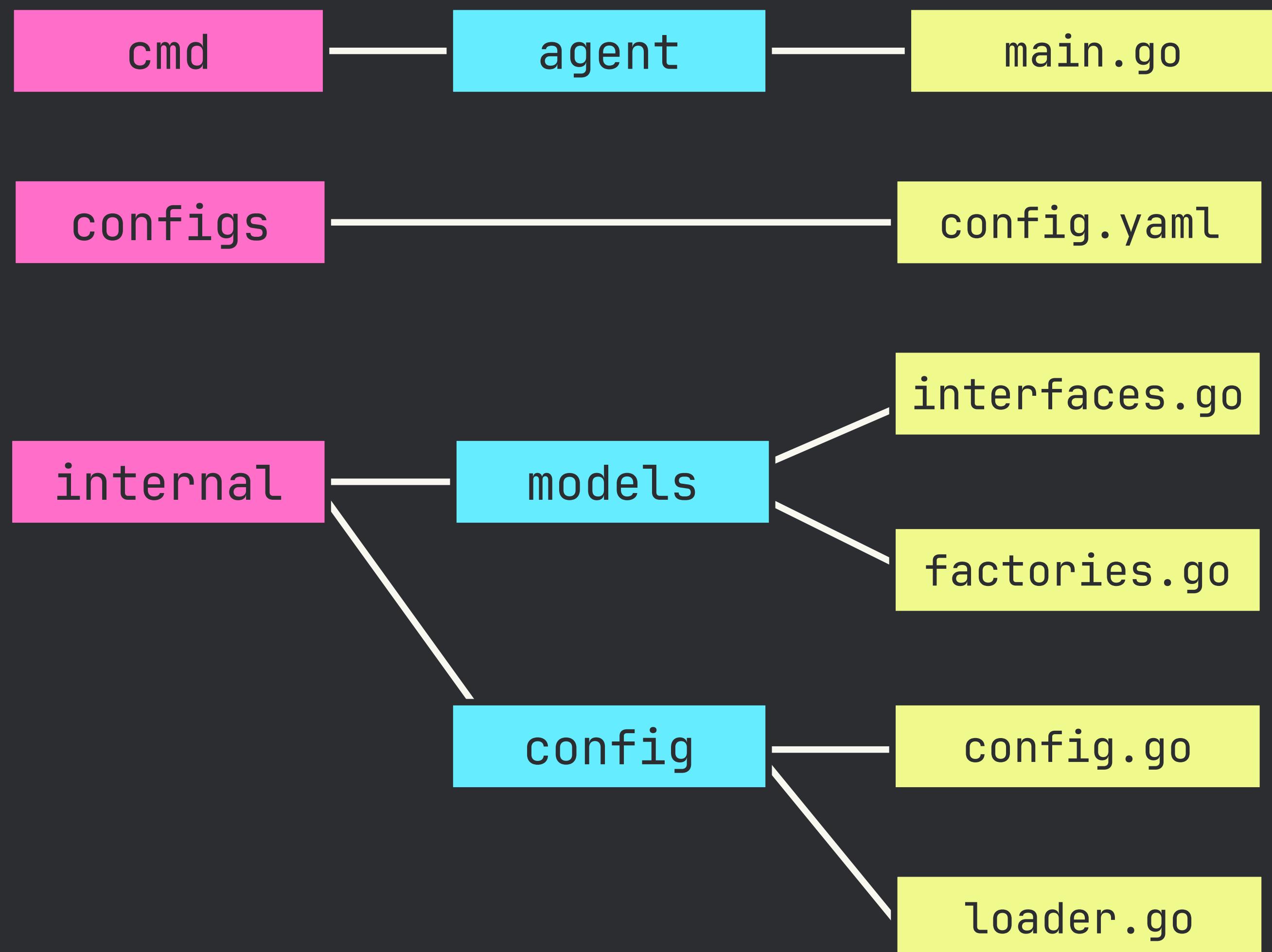
what we'll create

project

package

files

what we'll create

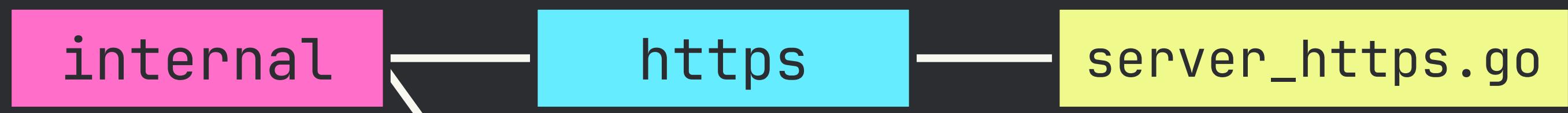
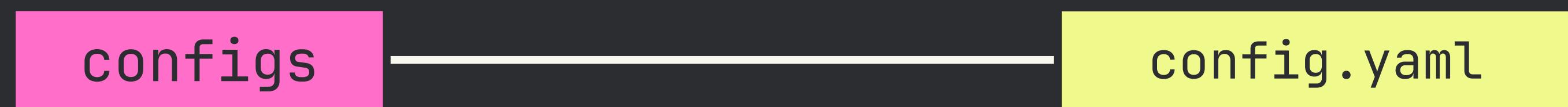


project

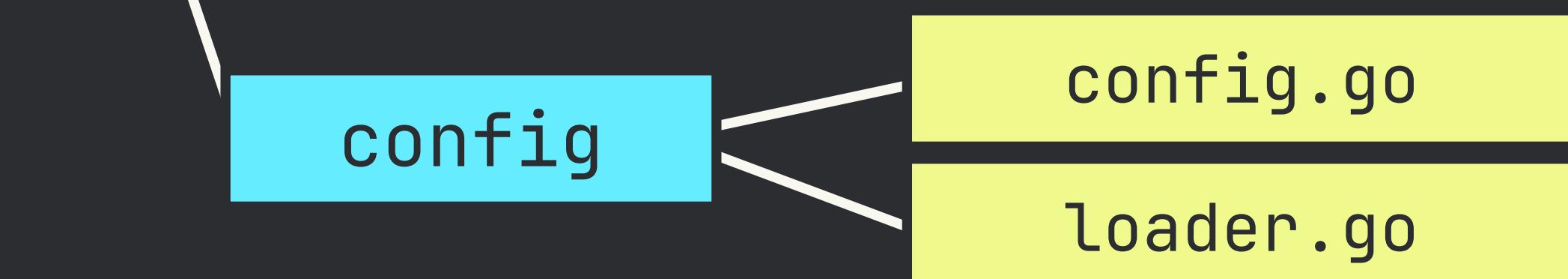
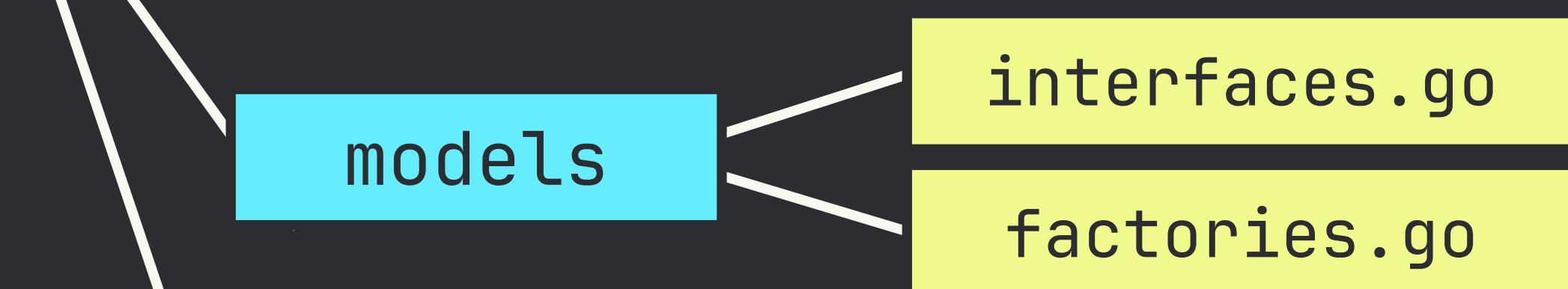
package

files

what we'll create



Server type, constructor, methods

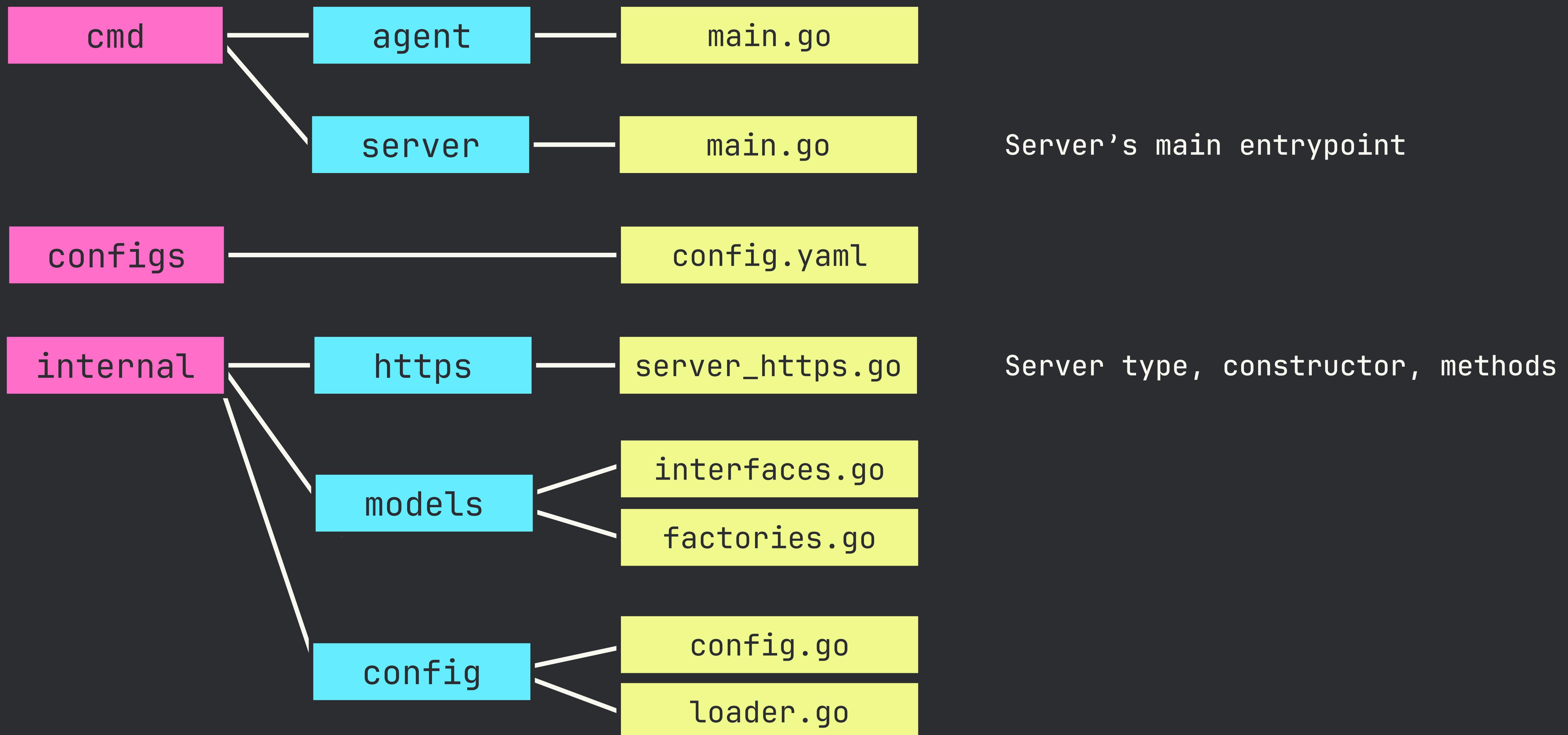


project

package

files

what we'll create







**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 4

https agent



# just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

just a reminder

HTTPS

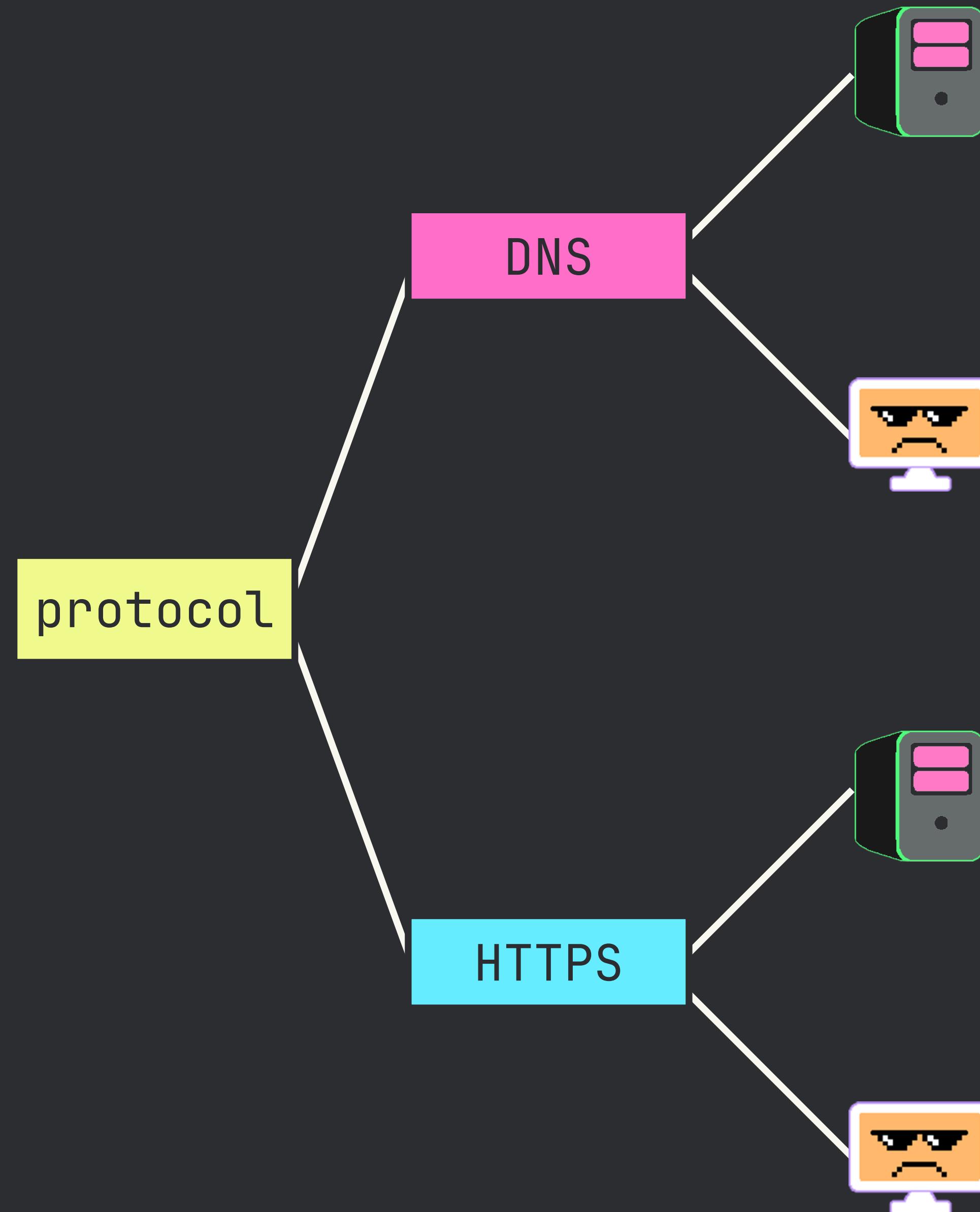
- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

# key concepts





each of the 4 have:

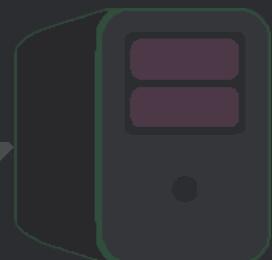
→ **struct**

→ **constructor**

→ **methods**

protocol

HTTPS



HTTPSAgent

NewHTTPSAgent()

```
type Agent interface {  
    Send(ctx context.Context) ([]byte, error)  
}
```



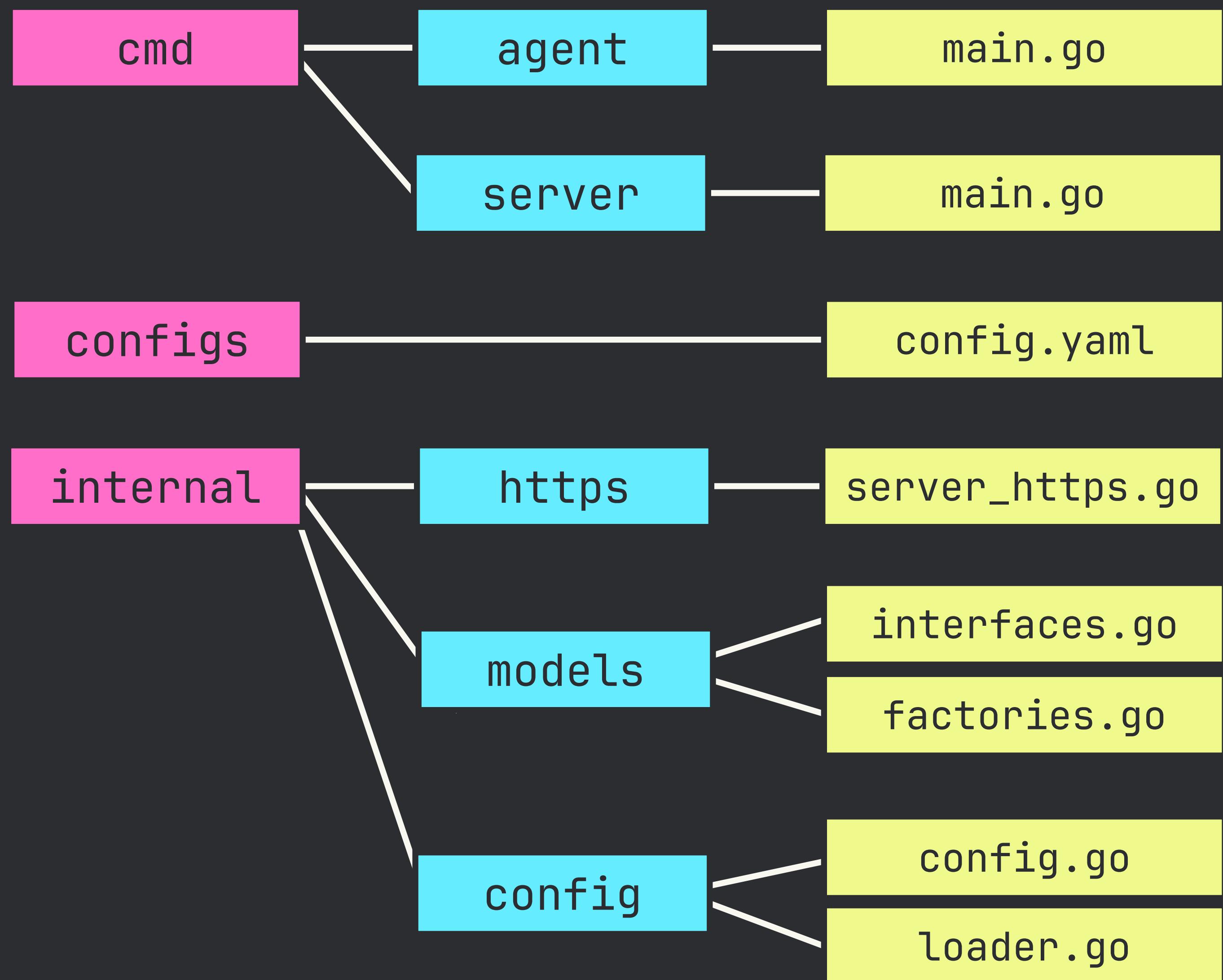
what we'll create

project

package

files

what we'll create

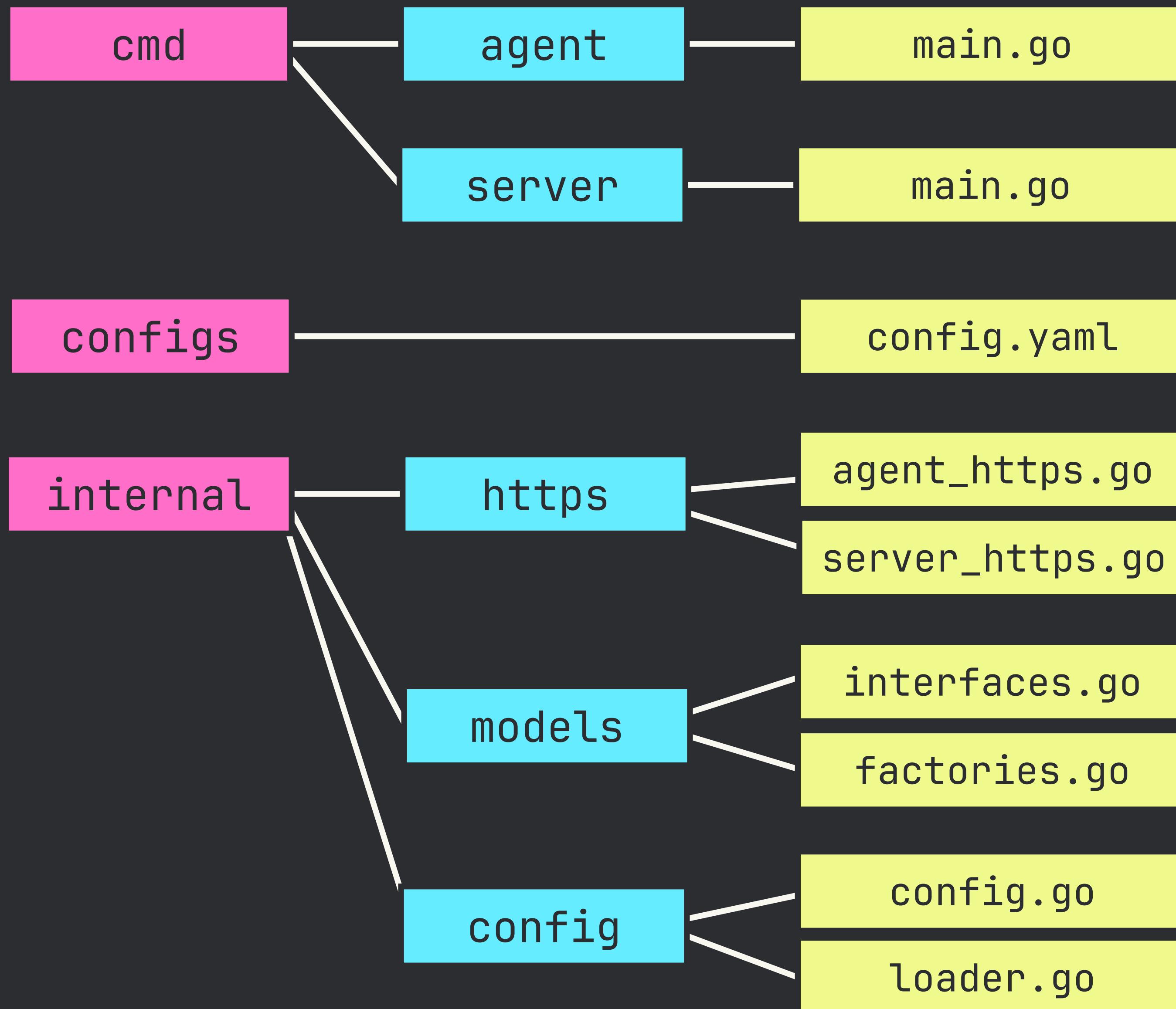


project

package

files

what we'll create



Agent type, constructor, methods





**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 5

# runloop



# just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

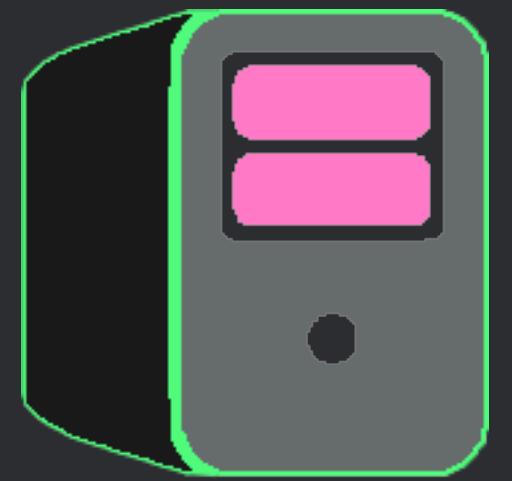
DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

# key concepts



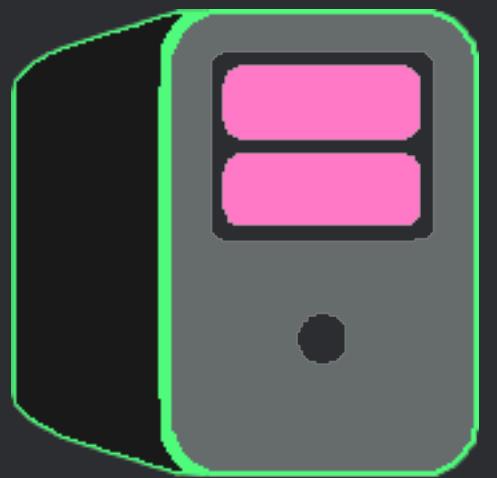
**Server**



**Agent**



**Server**

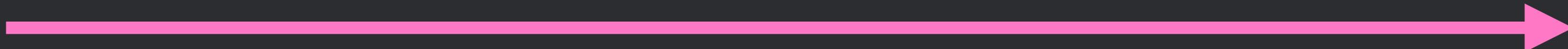
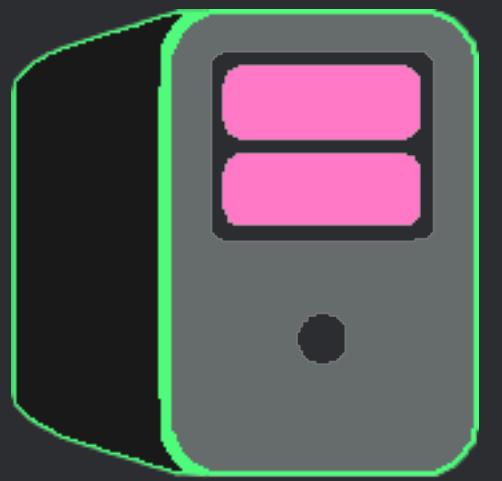


Hit / using GET

**Send()**



**Server**

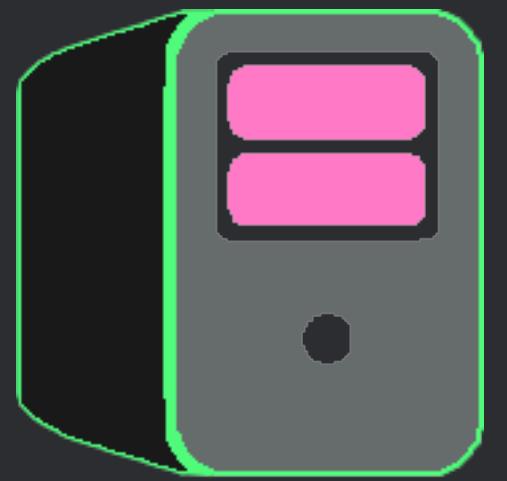


{ "change": false }

**Send()**



**Server**



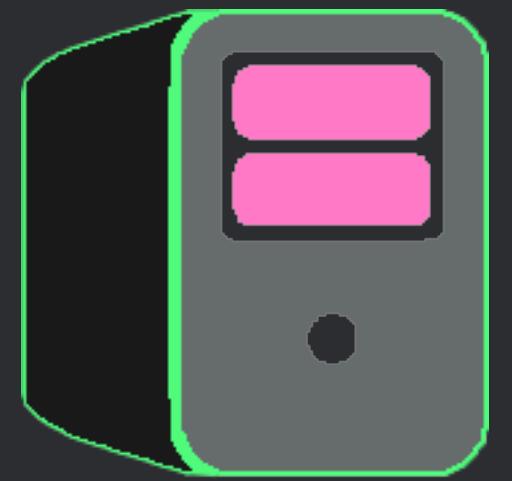
**Agent**



**Process**

**Result**

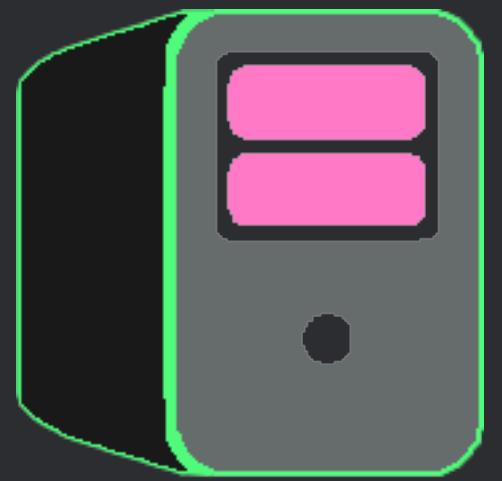
**Server**



**Agent**



**Server**

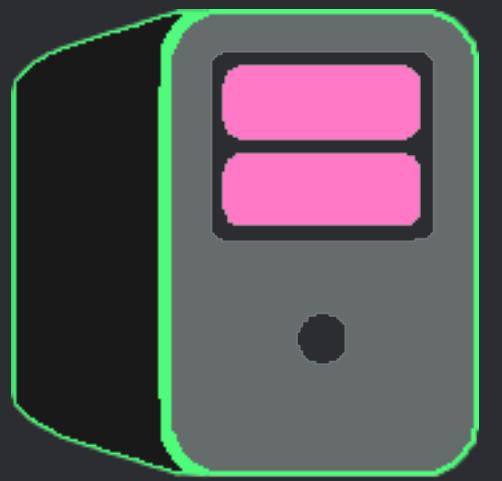


**Agent**



Send() just makes it happen once

**Server**

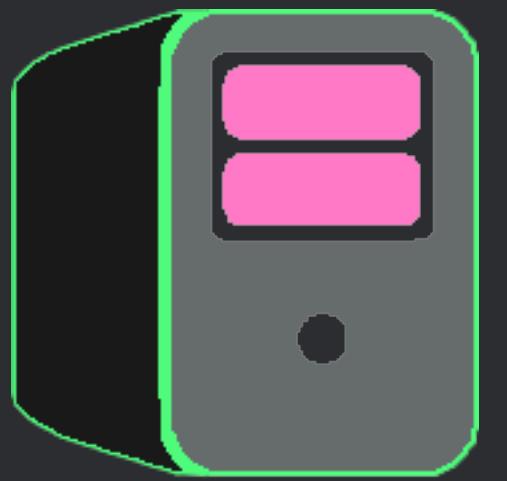


**Agent**



- We want a loop:
- Send()
  - Process results
  - Sleep
- ...REPEAT

**Server**

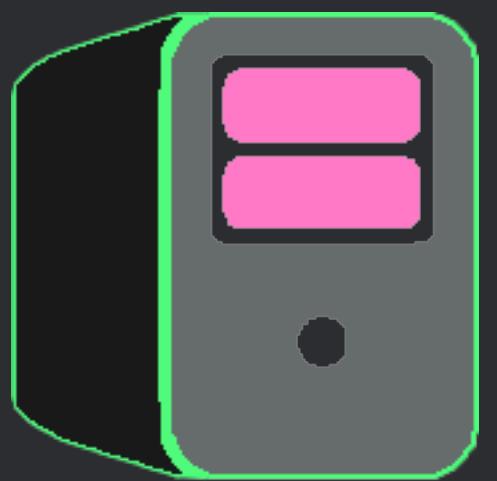


**Send()**

**RunLoop()**



**Server**



Hit / using GET

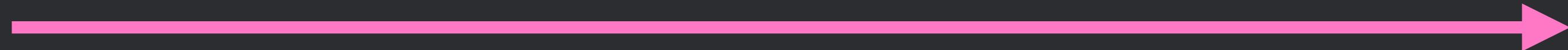
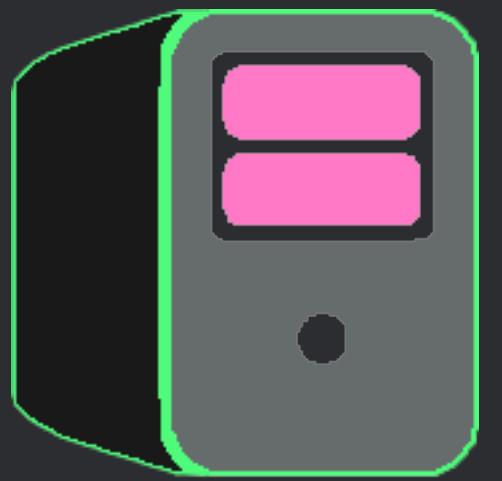


RunLoop()

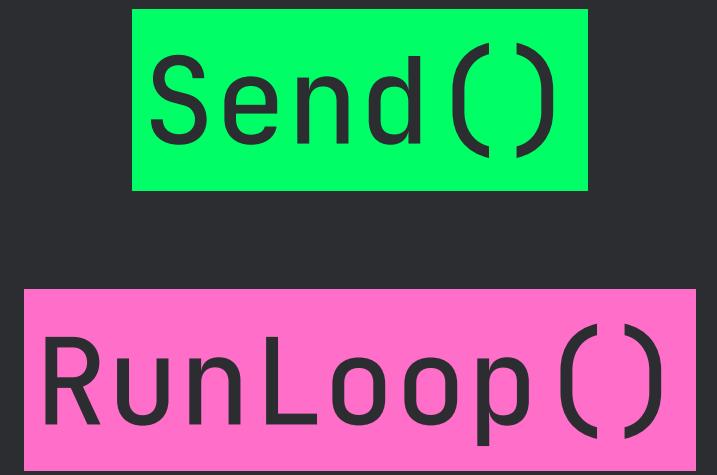


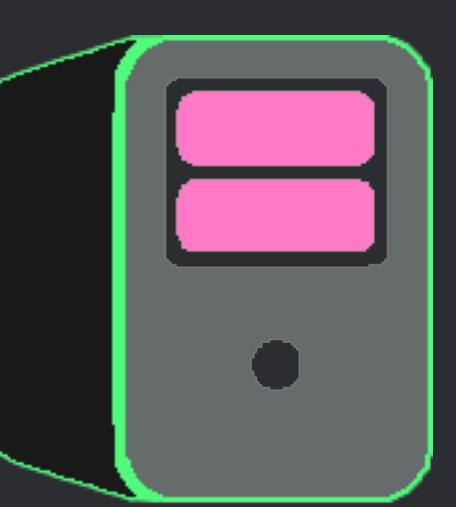
Send()

**Server**



{ "change": false }





**Server**

**Send()**

**RunLoop()**

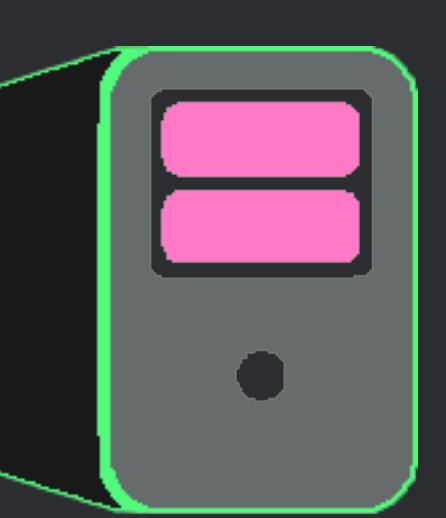


**Process**

**Result**

sleep

RunLoop()

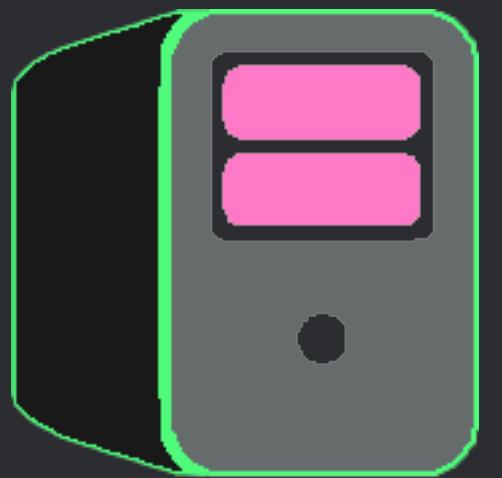


Server



Send()

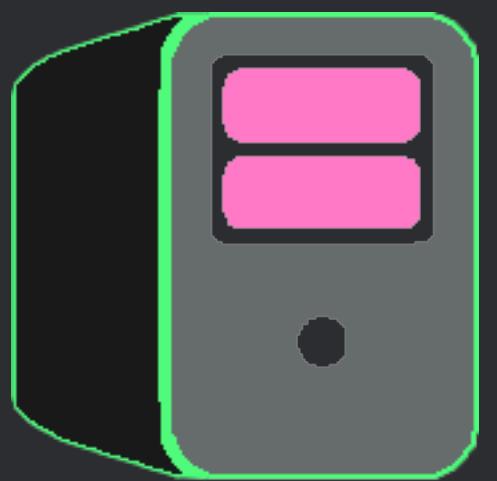
Server



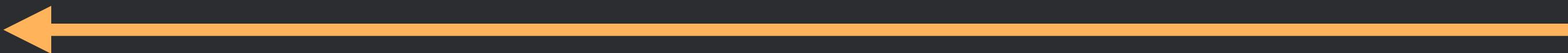
RunLoop()



**Server**



Hit / using GET

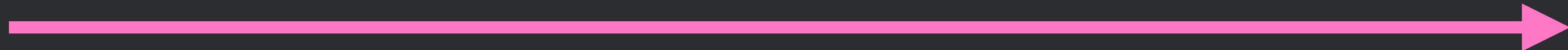
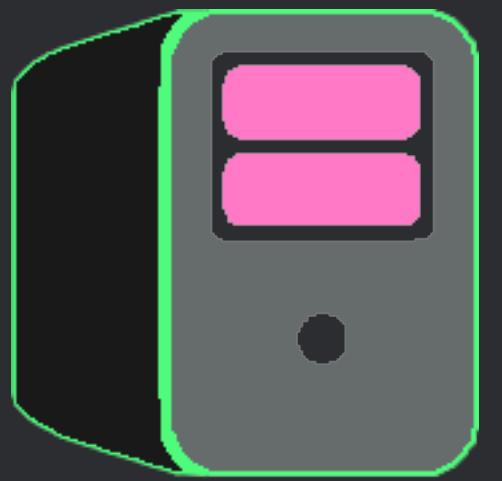


RunLoop()



Send()

**Server**



{ "change": false }

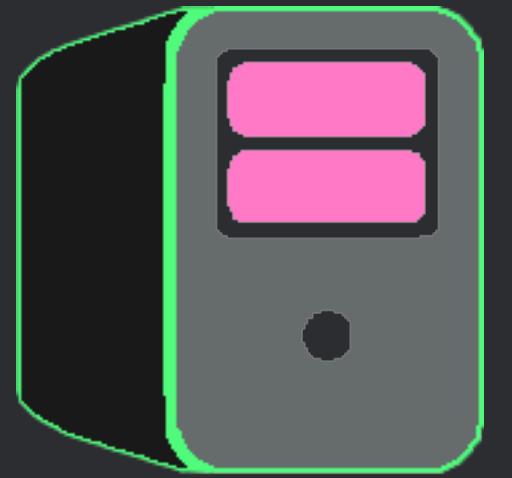
**Send()**

**RunLoop()**



**Send()**

**Server**

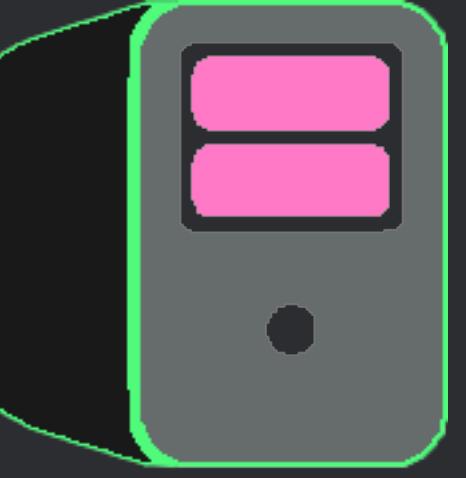


**RunLoop()**



**Process**

**Result**



Server

ad infinitum

sleep

RunLoop()





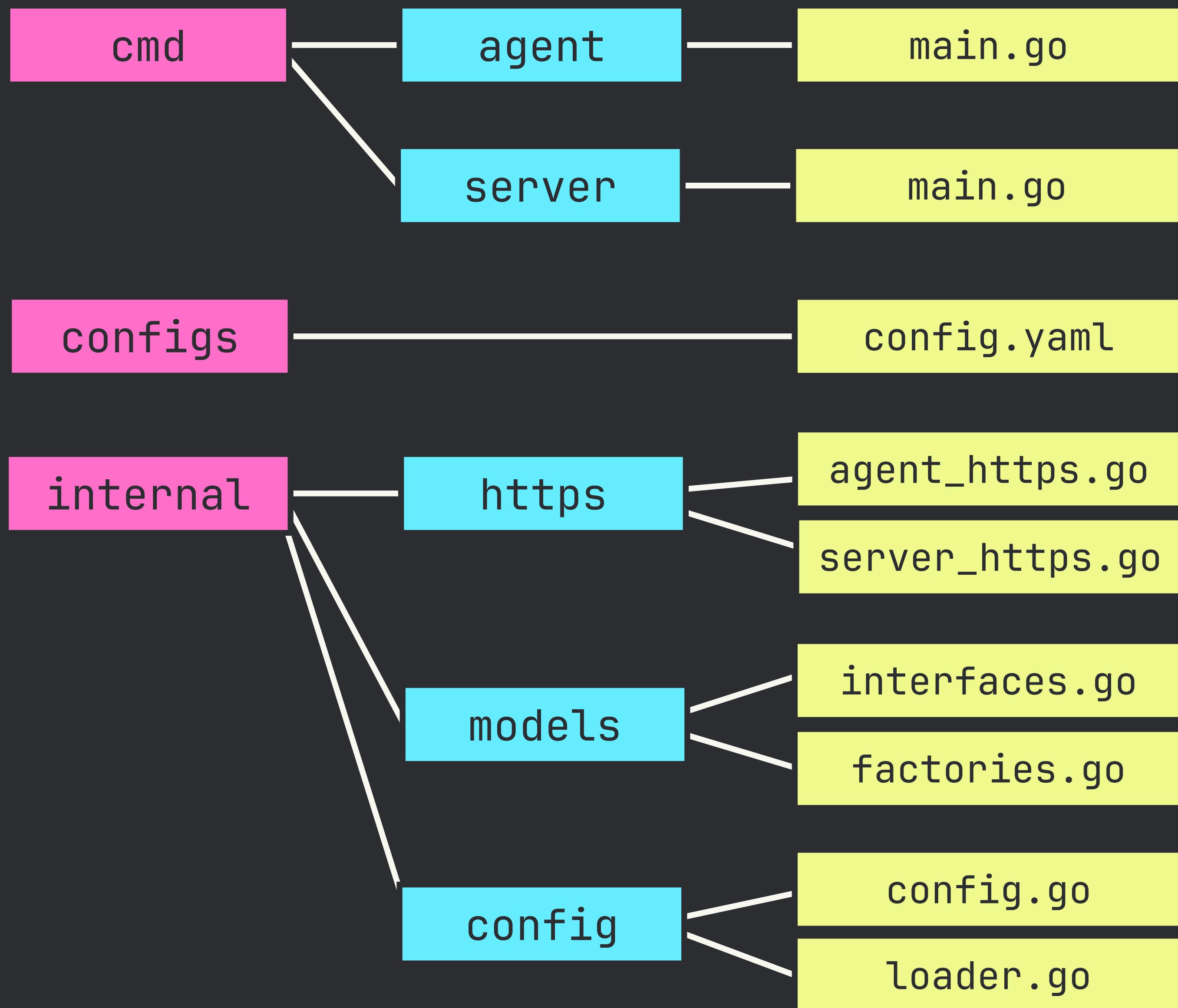
what we'll create

project

package

files

what we'll create

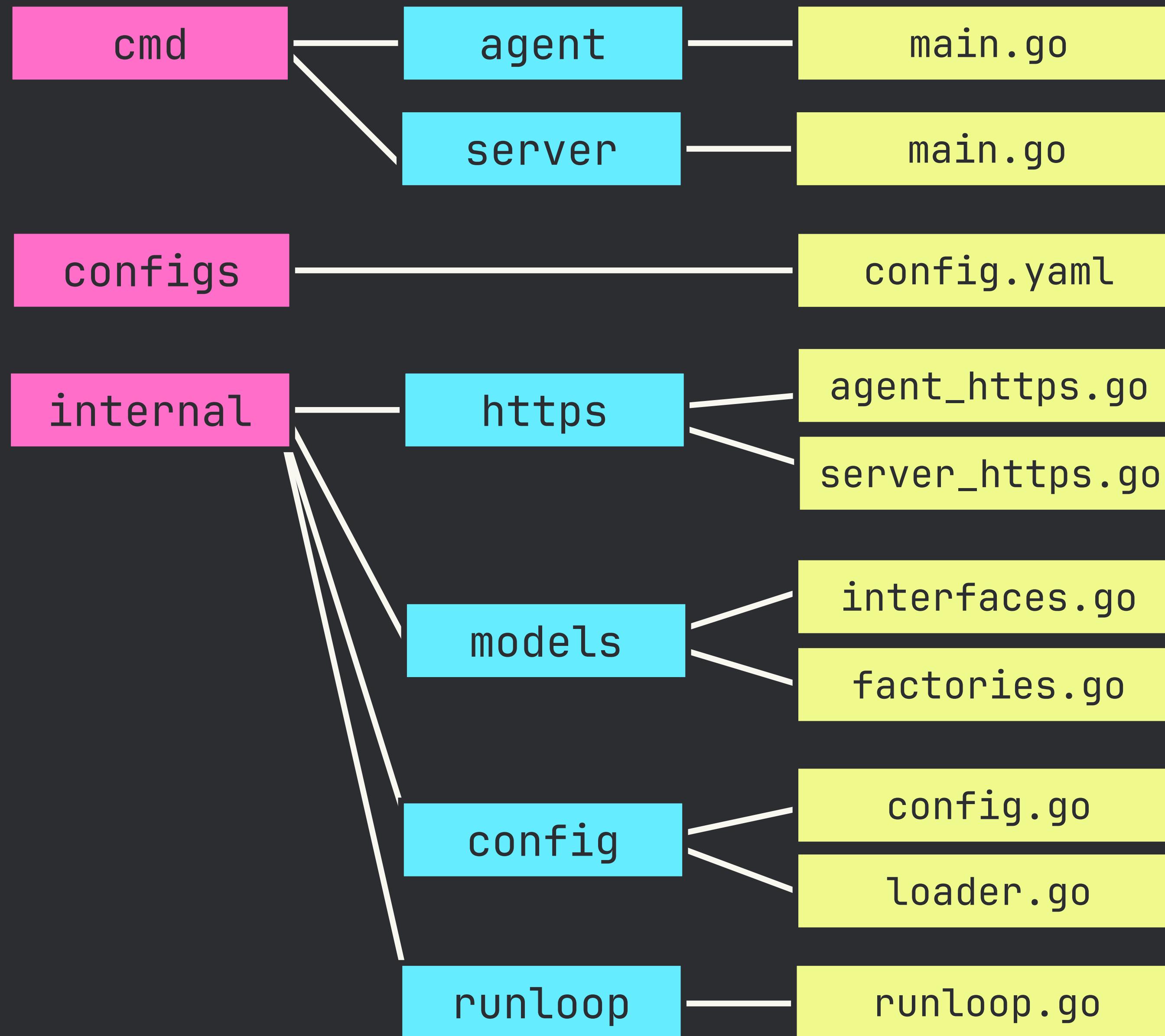


# project

# package

# files

# what we'll create



Agent runloop





**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 6

## dns server



# just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

just a reminder

HTTPS

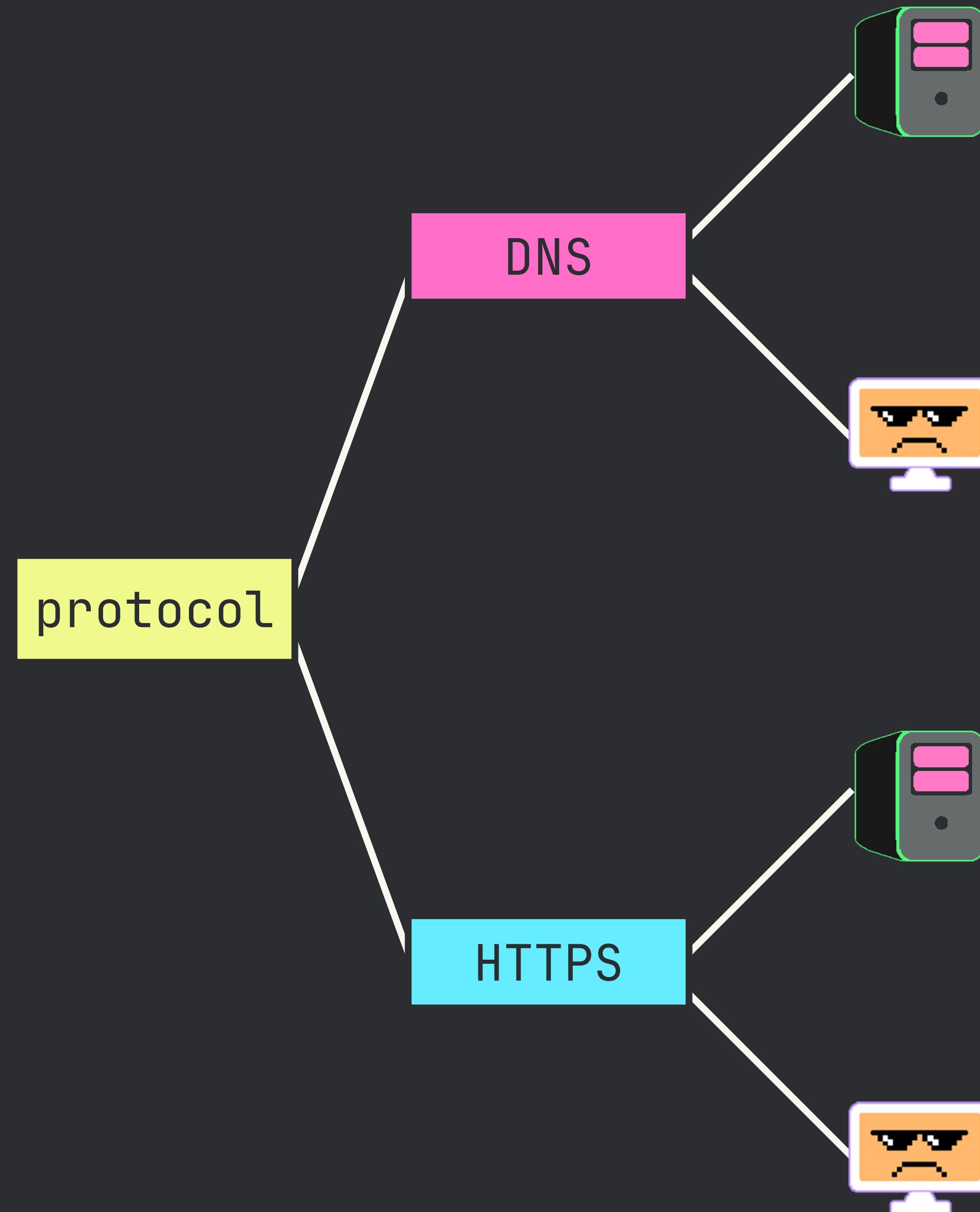
- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

# key concepts



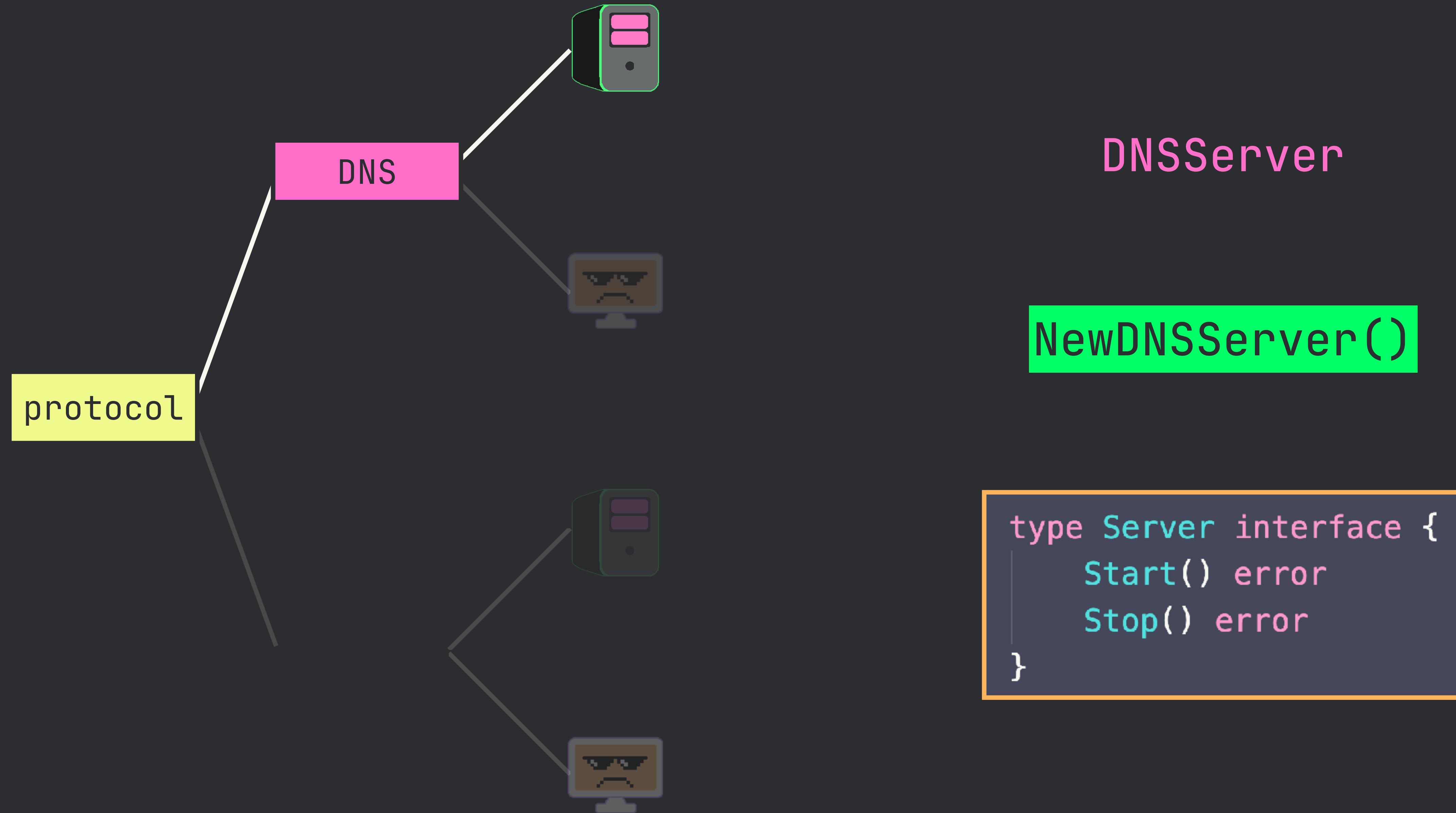


each of the 4 have:

→ **struct**

→ **constructor**

→ **methods**





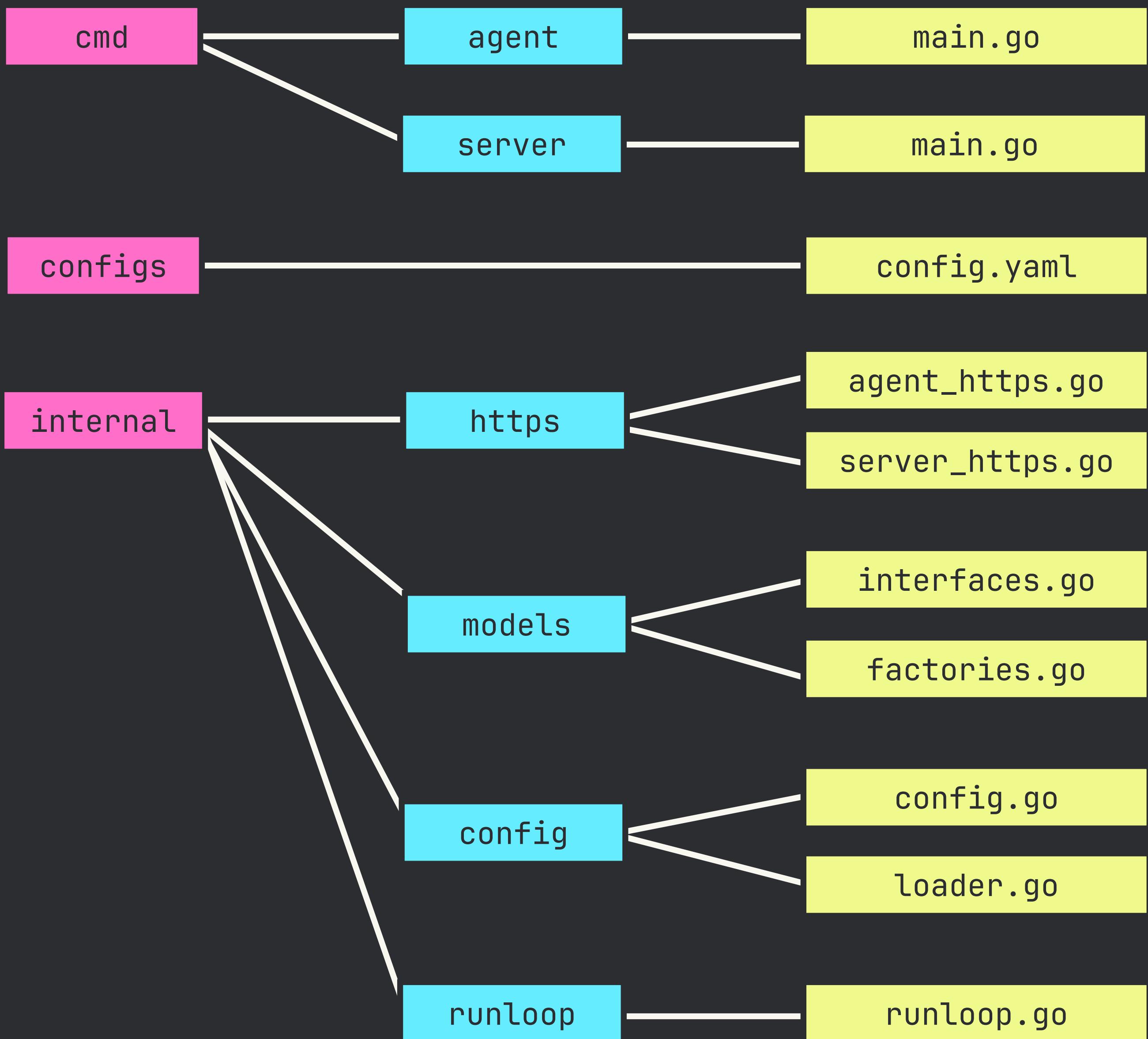
what we'll create

# project

# package

# files

# what we'll create

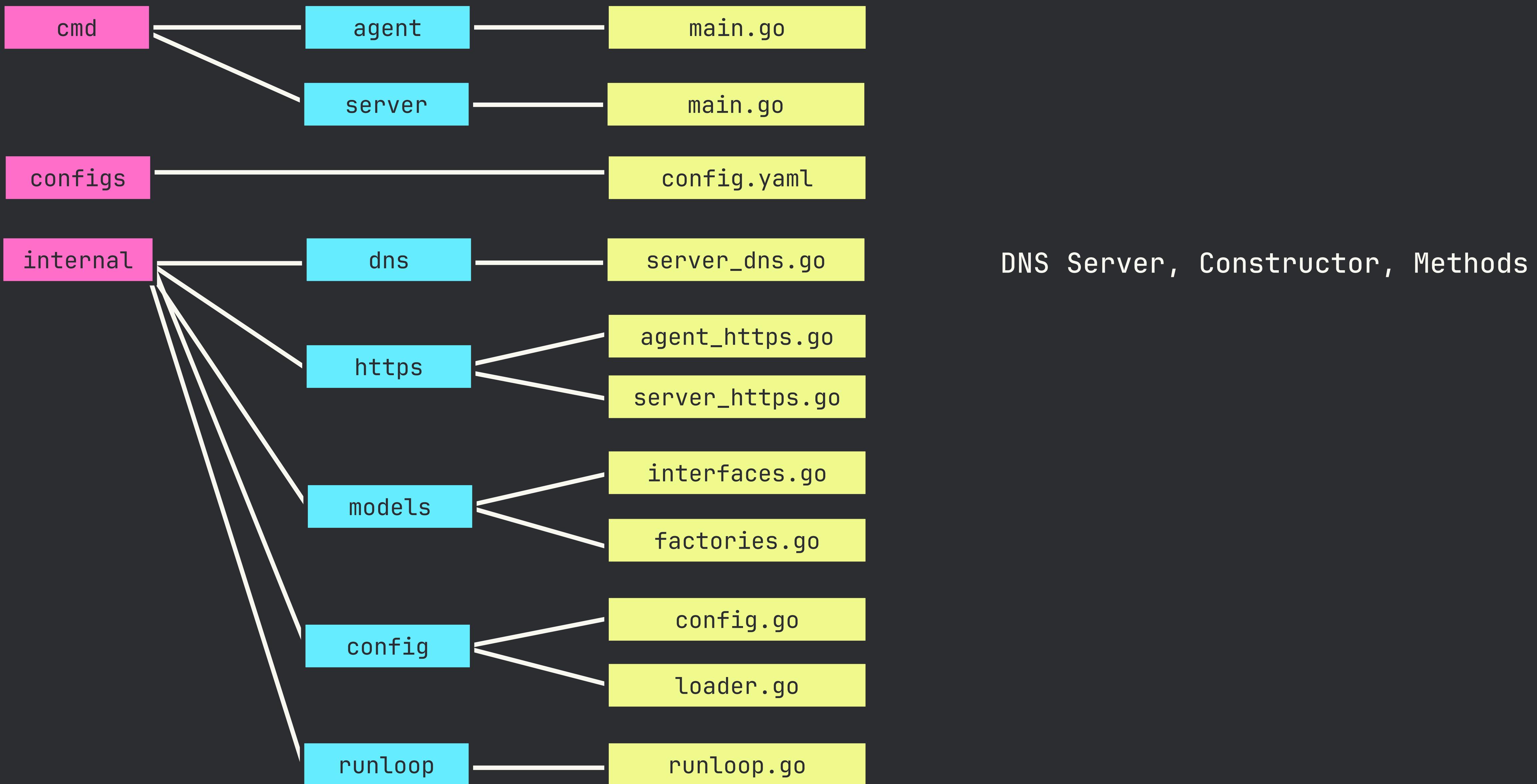


# project

# package

# files

# what we'll create







**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 7

## dns agent



# just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

just a reminder

HTTPS

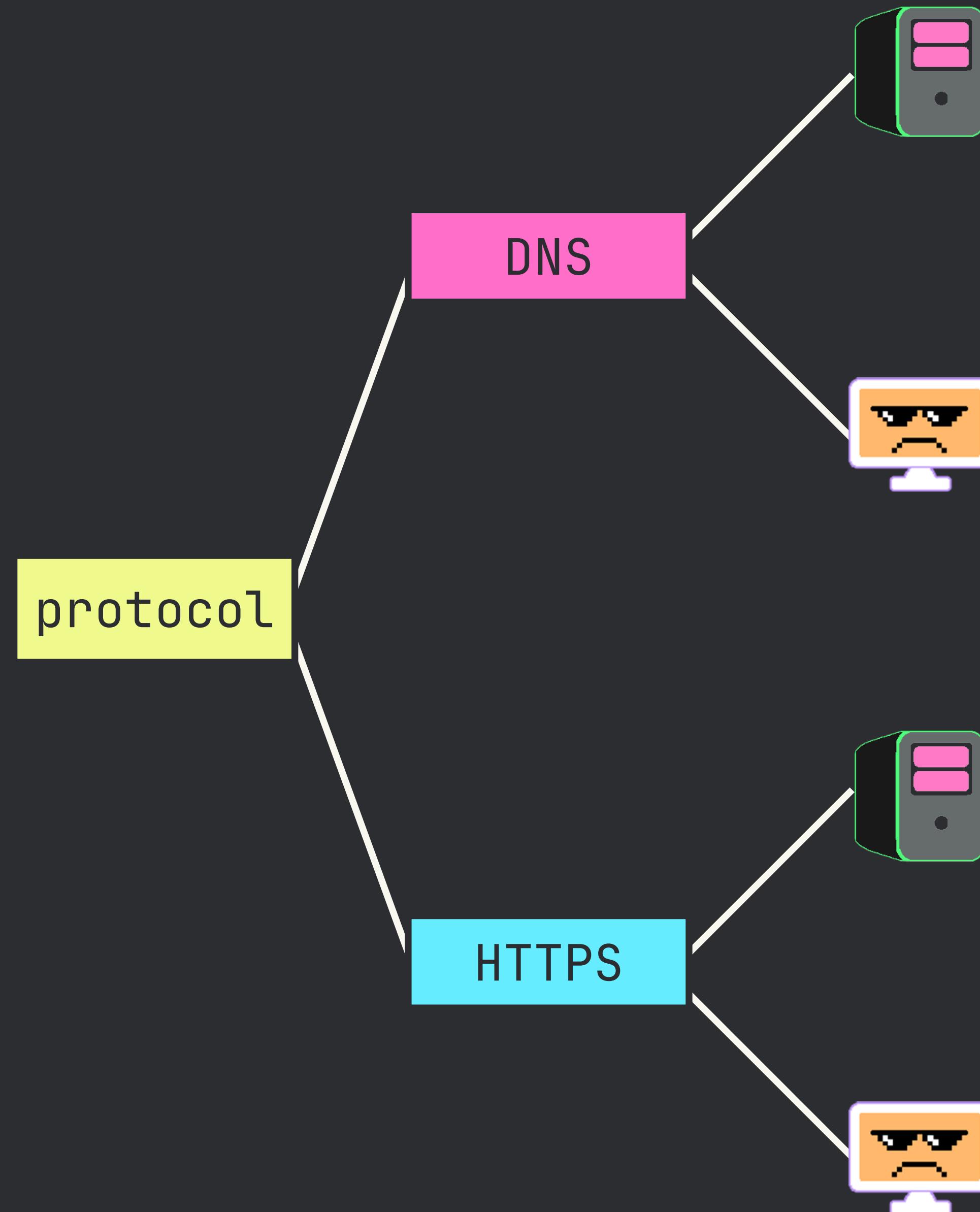
- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

# key concepts



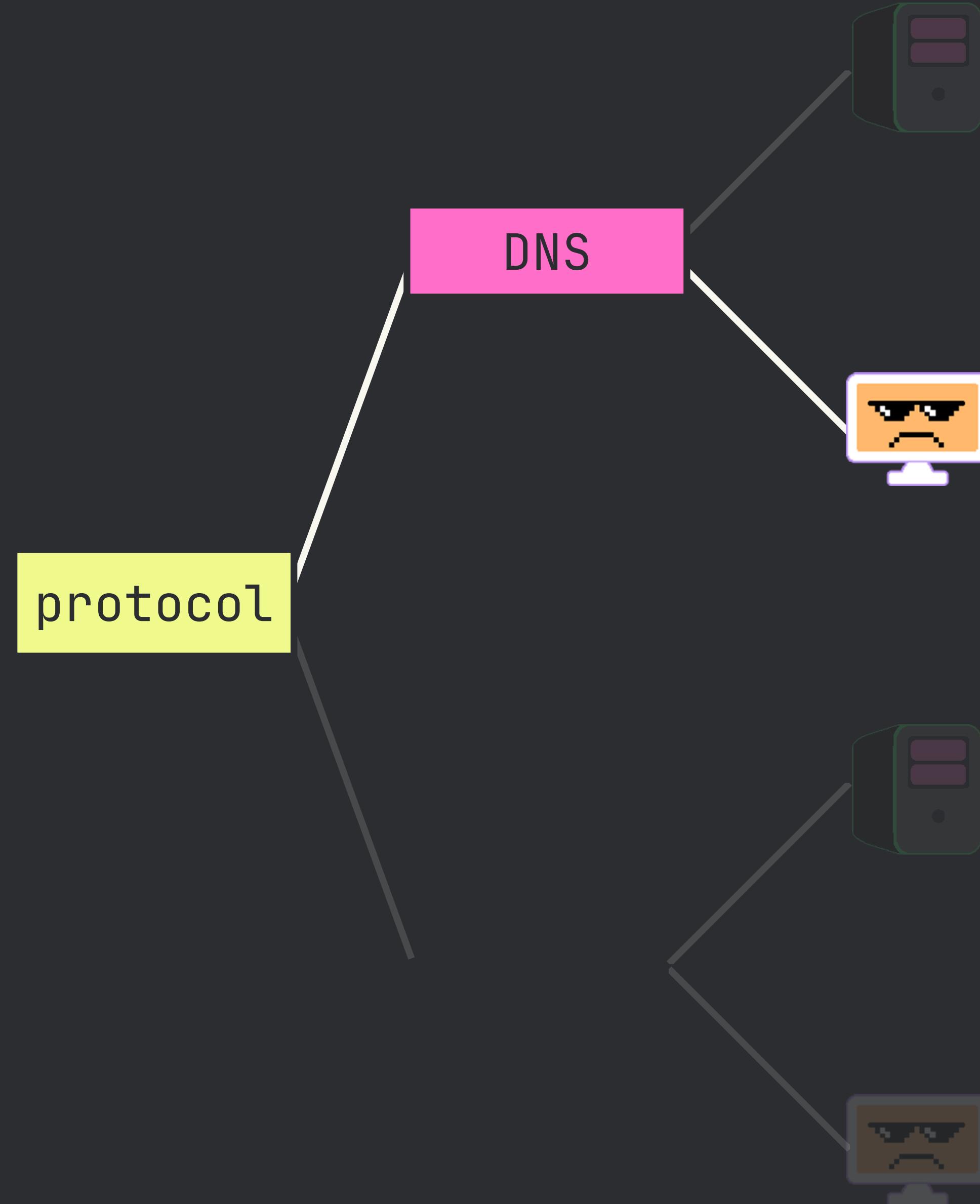


each of the 4 have:

→ **struct**

→ **constructor**

→ **methods**



DNSAgent

NewDNSAgent()

```
type Agent interface {
    Send(ctx context.Context) ([]byte, error)
}
```



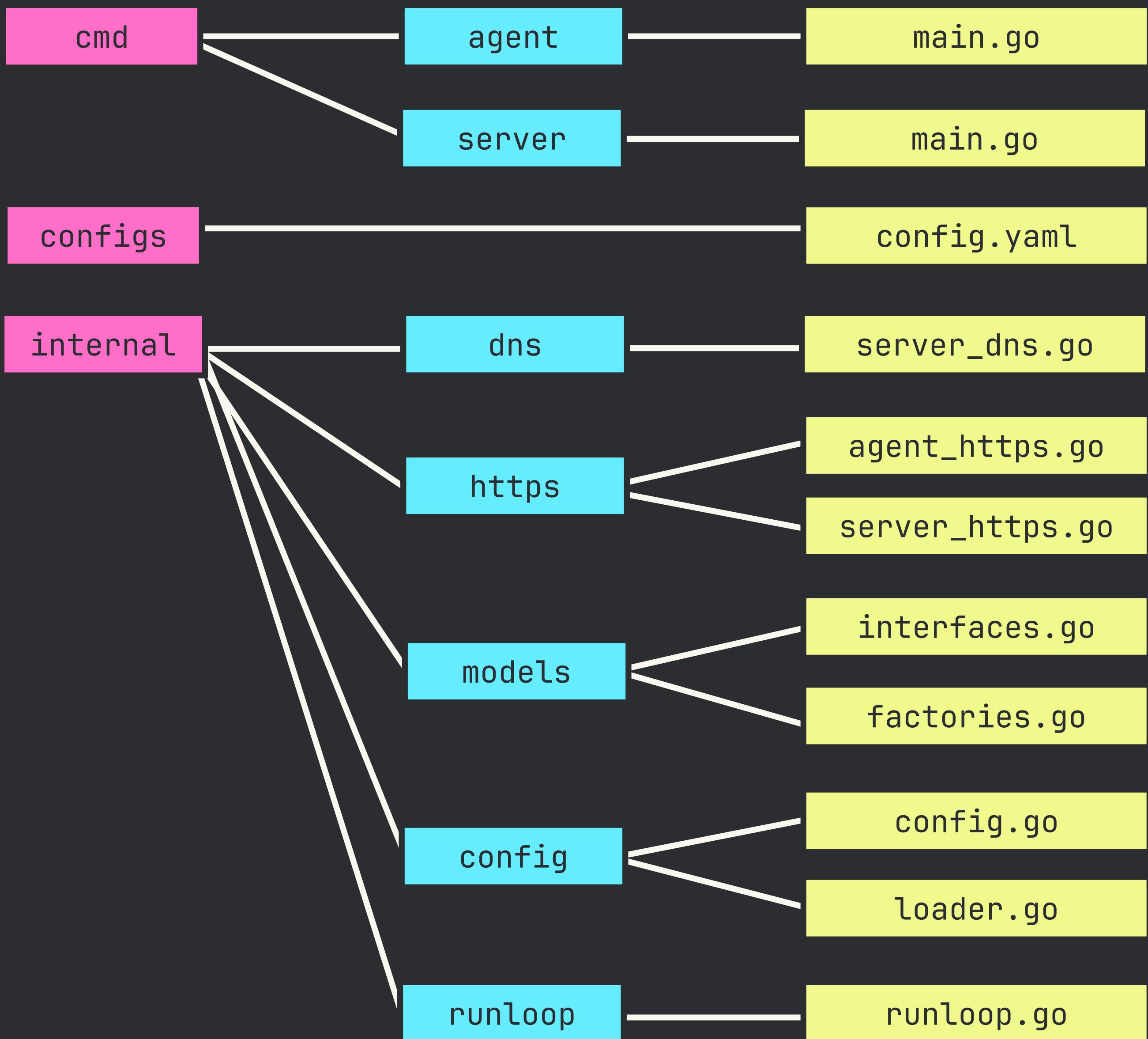
what we'll create

# project

# package

# files

# what we'll create

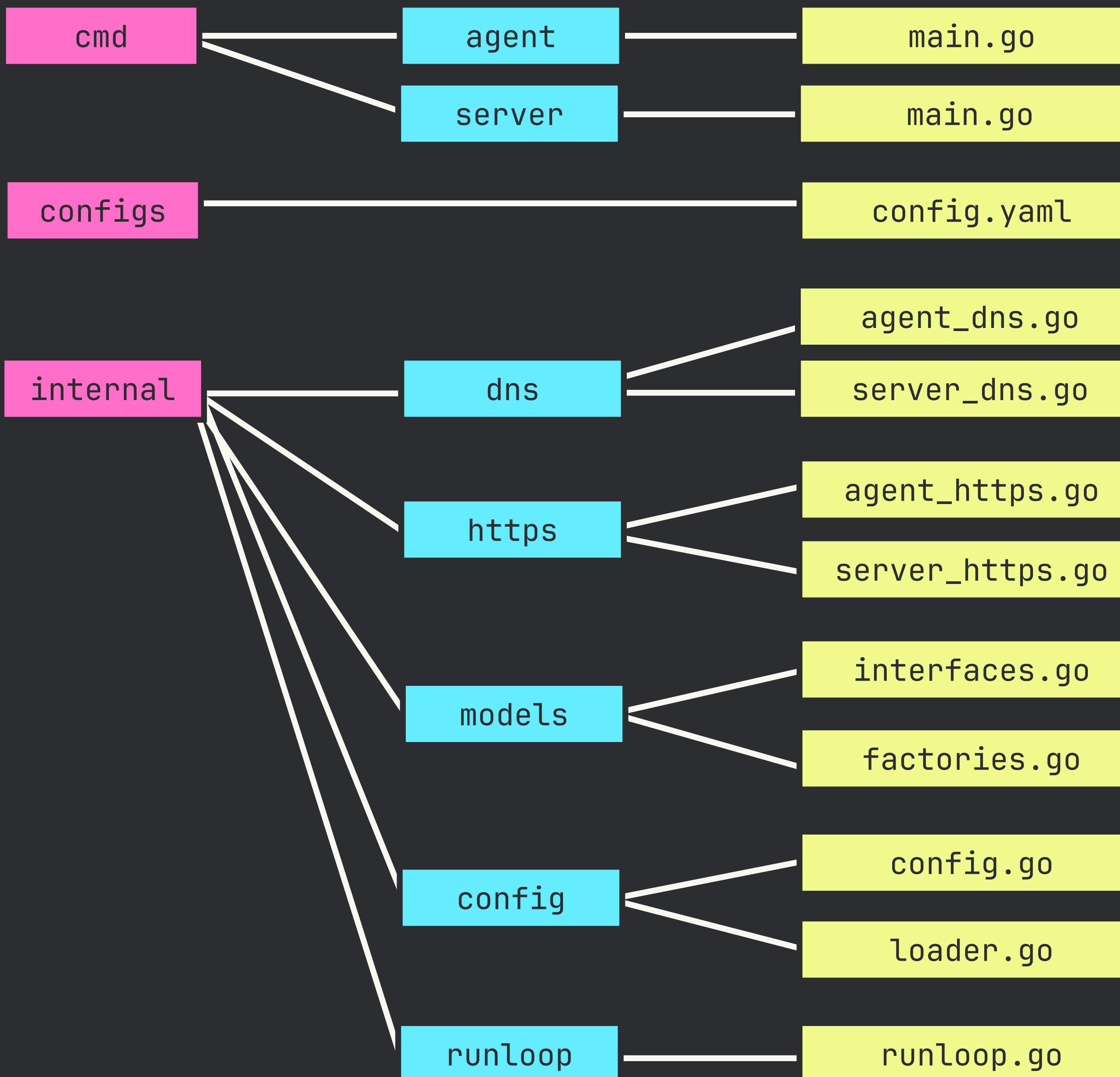


# project

# package

# files

# what we'll create



DNS Agent, Constructor, Methods





**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 8

## update runloop



# just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

just a reminder

HTTPS

- 3. HTTPS Server
- 4. HTTPS Agent
- 5. RunLoop

DNS

- 6. DNS Server
- 7. DNS Agent
- 8. Adapt RunLoop

# key concepts



let's look at current runloop

what part of code is “https-specific”?

# first, we call Send()

```
response, err := comm.Send(ctx)
if err != nil {
    log.Printf(format: "Error sending request: %v", err)
    // Don't exit - just sleep and try again
    time.Sleep(cfg.Timing.Delay)
    continue // Skip to next iteration
}
```

Since we use interface method, it's **agnostic**!

## next we unmarshal response

```
var httpsResp https.HTTPSResponse
err = json.Unmarshal(response, &httpsResp)
if err != nil {
    log.Fatalf(format: "Failed to parse response: %v", err)
}

log.Printf(format: "Received response: change=%v", httpsResp.Change)
```

This is specific to HTTPS

## finally, we sleep

```
// Calculate sleep duration with jitter
sleepDuration := CalculateSleepDuration(cfg.Timing.Delay, cfg.Timing.Jitter)
log.Printf(format: "Sleeping for %v", sleepDuration)

// Sleep with cancellation support
select {
case <-time.After(sleepDuration):
case <-ctx.Done():
    return ctx.Err()
}
```

This is also completely agnostic

so this is the only part that's HTTPS-specific

```
var httpsResp https.HTTPSResponse
err = json.Unmarshal(response, &httpsResp)
if err != nil {
    log.Fatalf(format: "Failed to parse response: %v", err)
}

log.Printf(format: "Received response: change=%v", httpsResp.Change)
```

so this is the only part that's **HTTPS-specific**

- we can use the same function
- we can use a switch at this point
- conditional (dns/https) logic



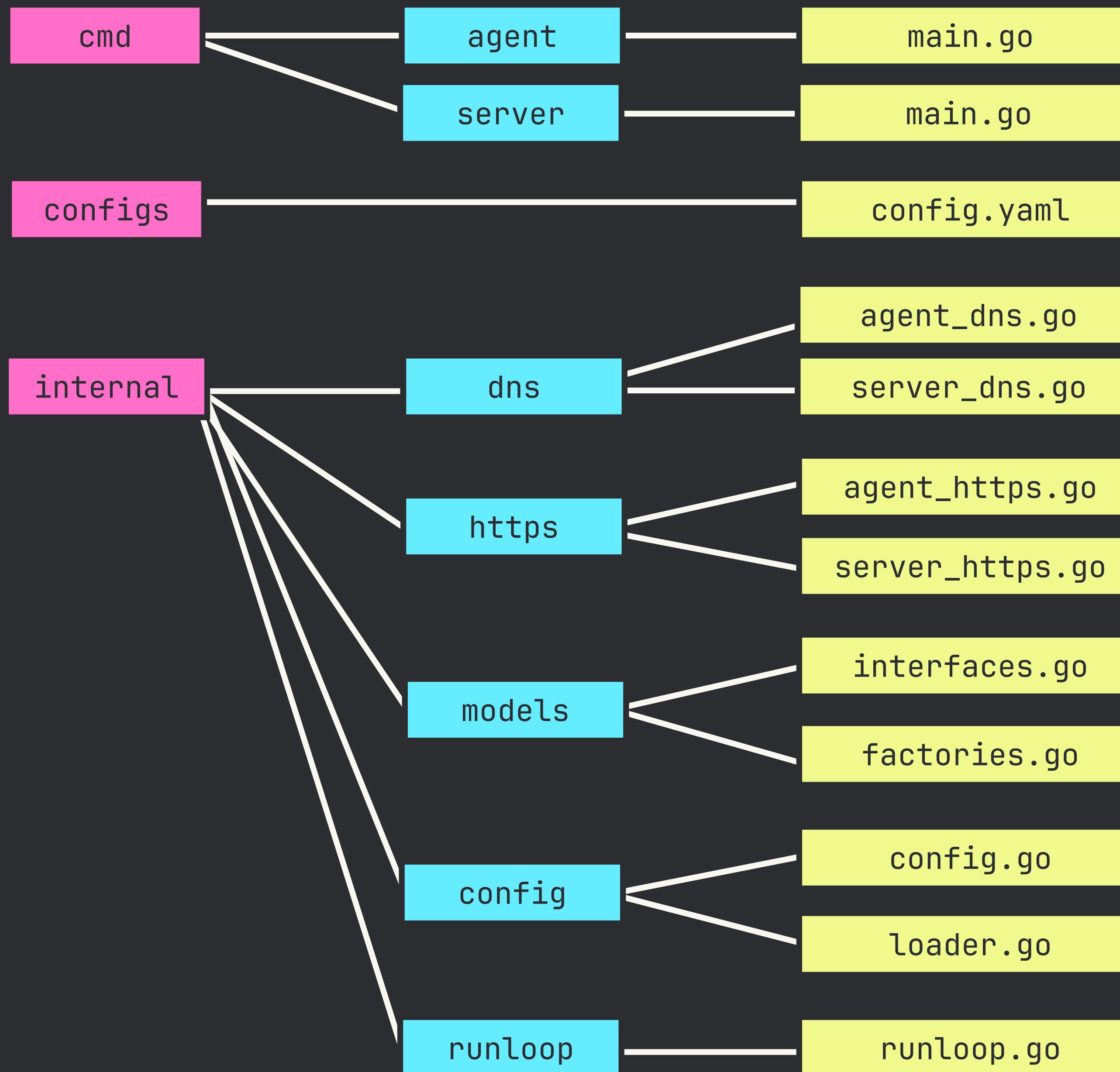
what we'll create

# project

# package

# files

# what we'll create



Nothing added, just change here.





**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 9

## api switch

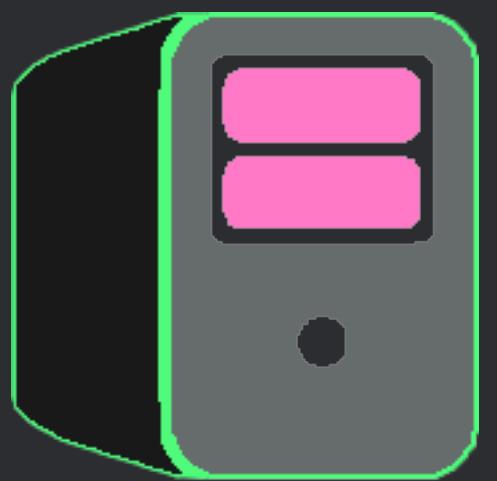


# key concepts



- we have a HTTPS/DNS server + agent and runloop
- all the foundational logic is in place
- but right now our server is “dumb”
- always send back the same response - don’t change

**Server**



Hit / using GET

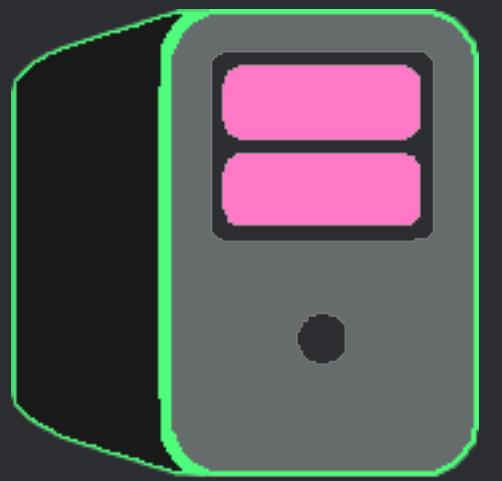


RunLoop()

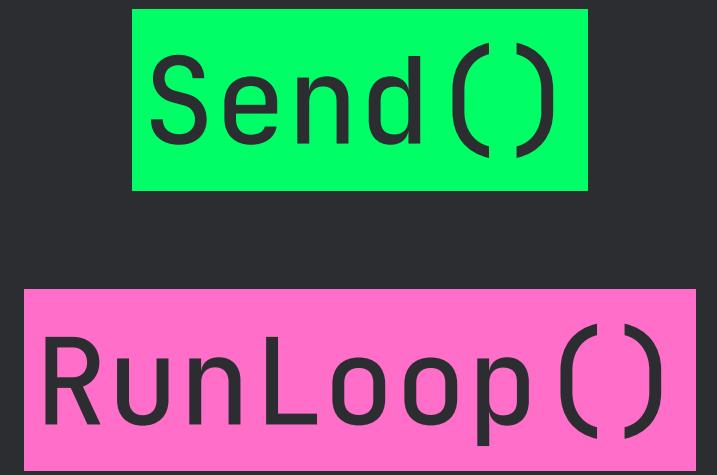


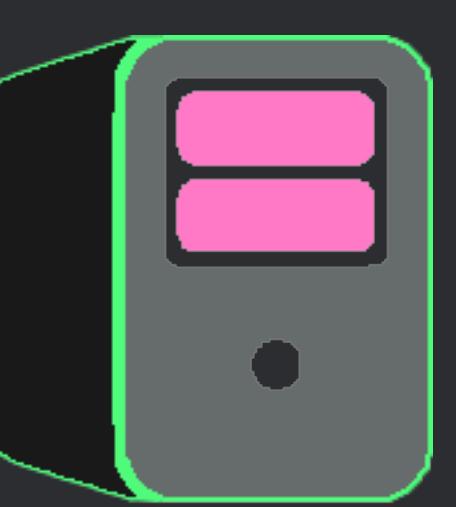
Send()

**Server**



{ "change": false }





**Server**

**Send()**

**RunLoop()**

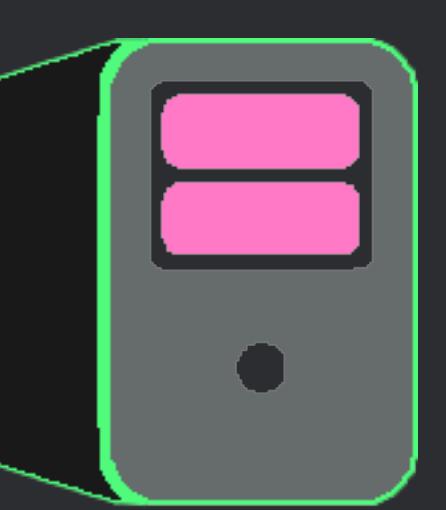


**Process**

**Result**

sleep

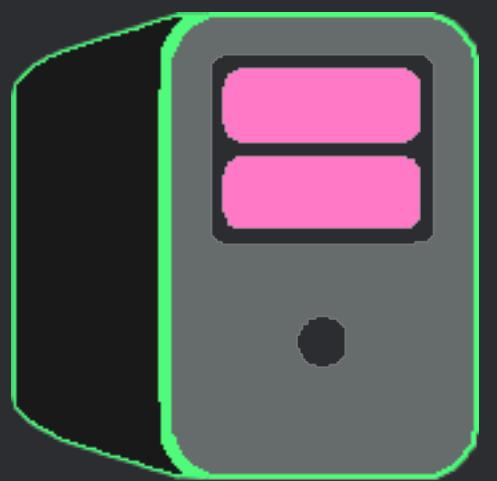
RunLoop()



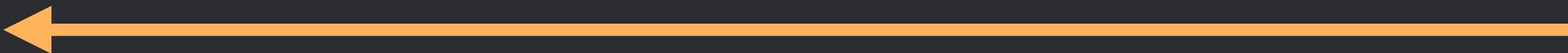
Server



**Server**



Hit / using GET

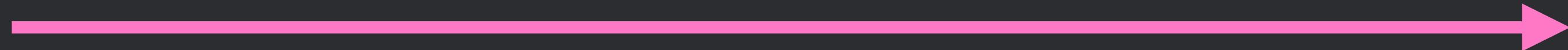
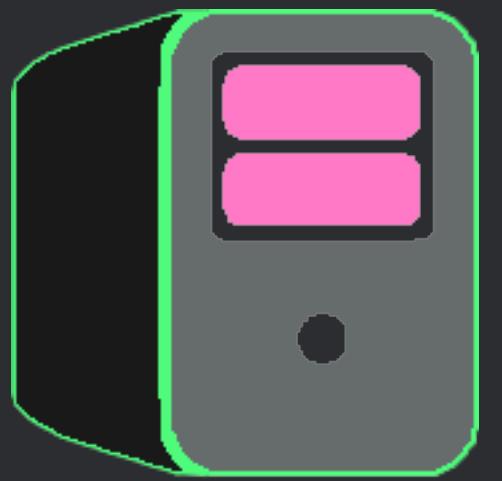


RunLoop()

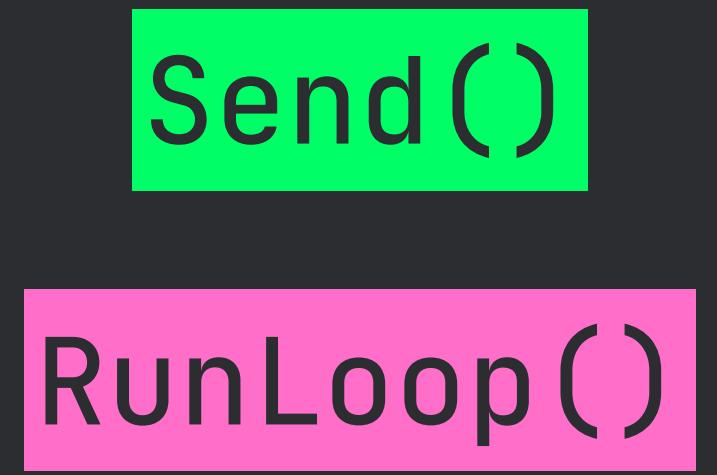


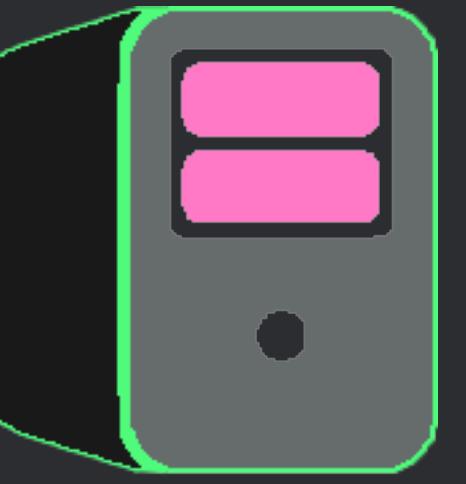
Send()

**Server**



{ "change": false }





**Server**

**Send()**

**RunLoop()**

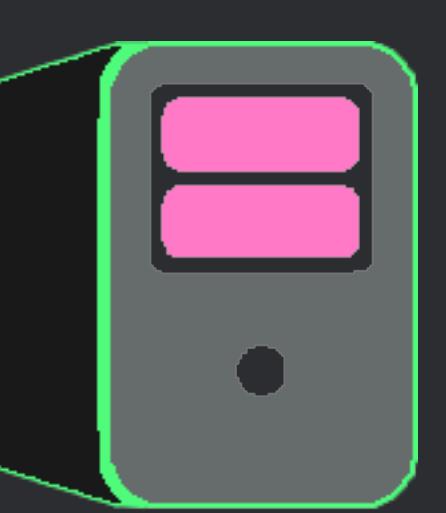


**Process**

**Result**

sleep

RunLoop()



Server



nothing we can do to tell the server:

the next time the agent check's in,

tell it you do want it to switch

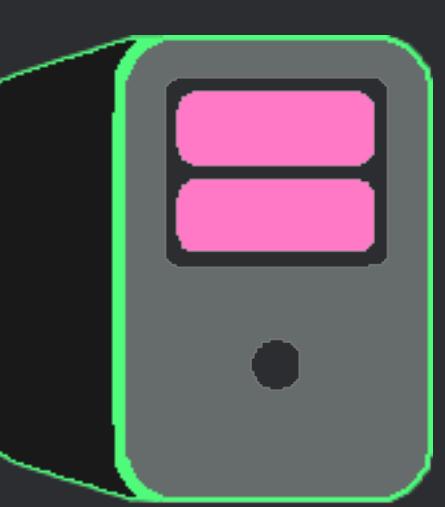
that's what we want to create here

- an endpoint we can hit to signal our intention
- ability for server to check internally whether this intention was declared
- ability for server to alter the response if intention was detected
- IMPORTANT: “consume once”

how are we going to do this? GLOBAL FLAG

- picture a **bool** that can be accessed from **anywhere**
- it defaults to **false** (meaning we don't want to change)
- if we hit say **/change** on **port 8080**, it changes to **true**
- server thus then alters its **response AND resets flag**





Server



Send()  
RunLoop()

Hit / using GET

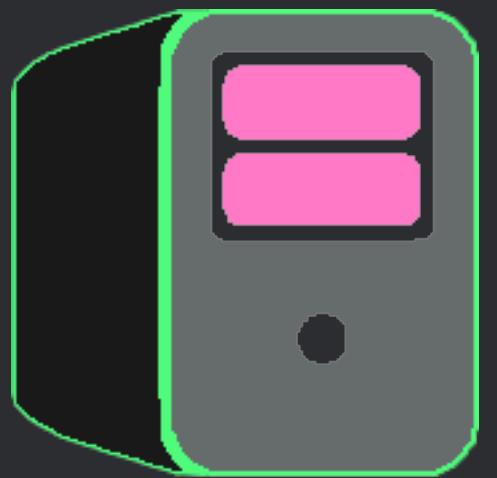


/change  
:8080

Send()

RunLoop()

Server



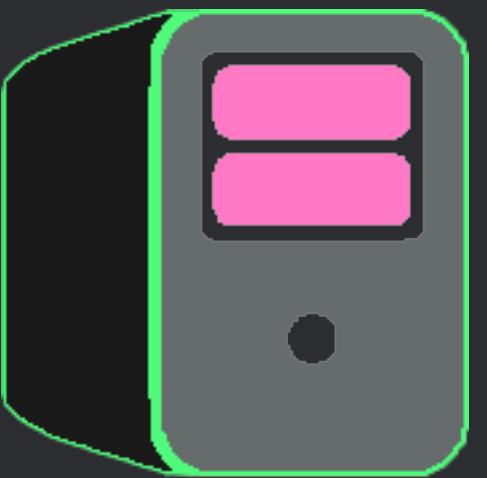
/change  
:8080



Send()

RunLoop()

Server

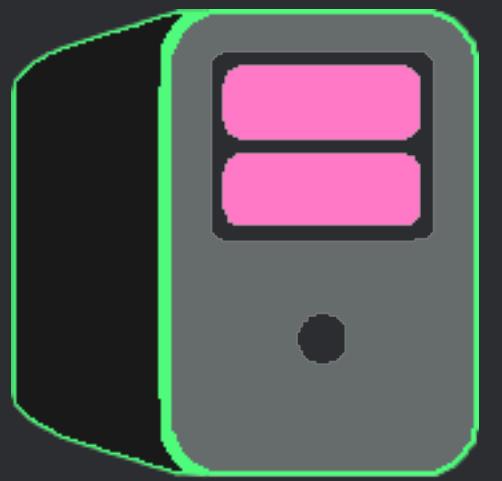


false

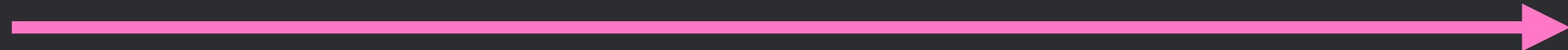


/change  
:8080

**Server**



/change  
:8080



{ "change": false }

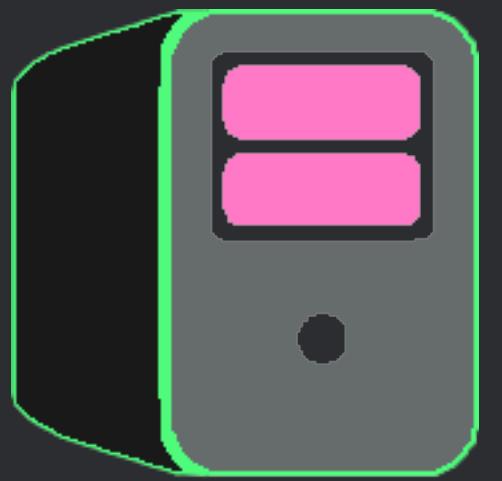
Send()  
RunLoop()



**Send()**

**RunLoop()**

**Server**



**Process**

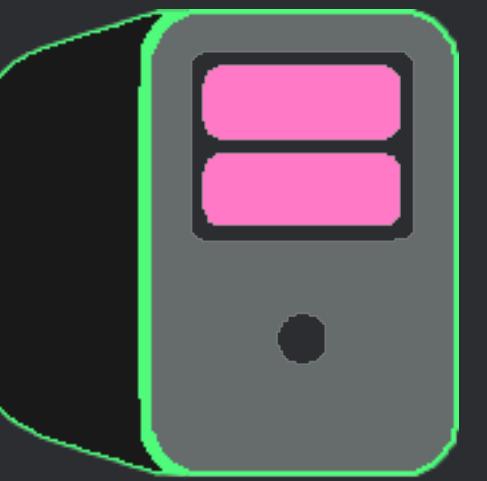
**Result**



**/change  
:8080**

sleep

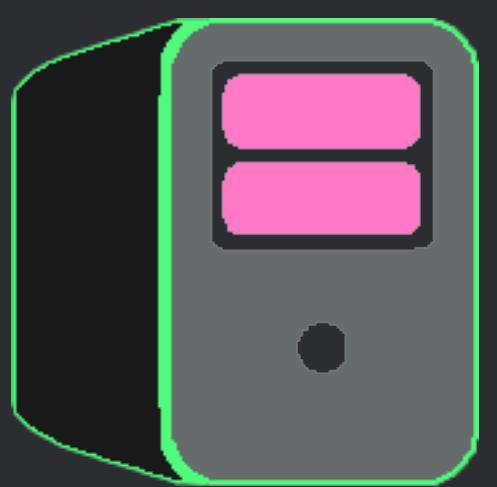
RunLoop()



/change  
:8080

sleep

RunLoop()

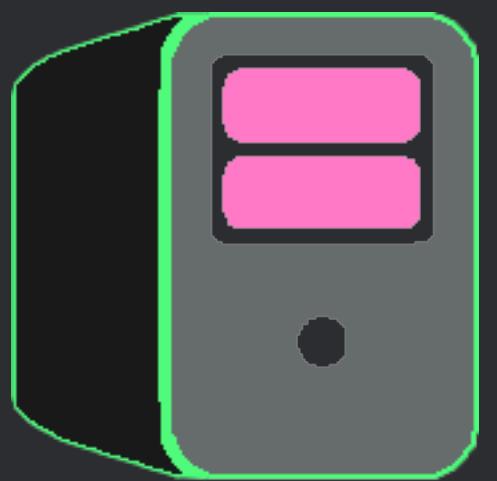


/change  
:8080



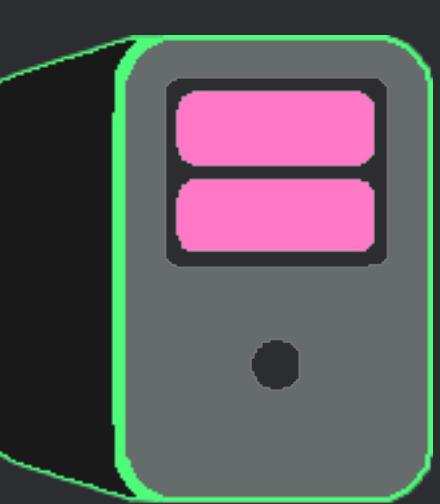
sleep

RunLoop()



Flag icon  
`/change`  
`:8080`





Server



Send()

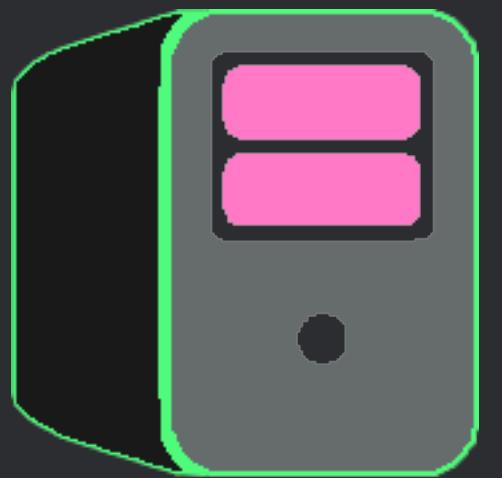
RunLoop()

Hit / using GET



/change  
:8080

**Server**



/change  
:8080

**Send()**

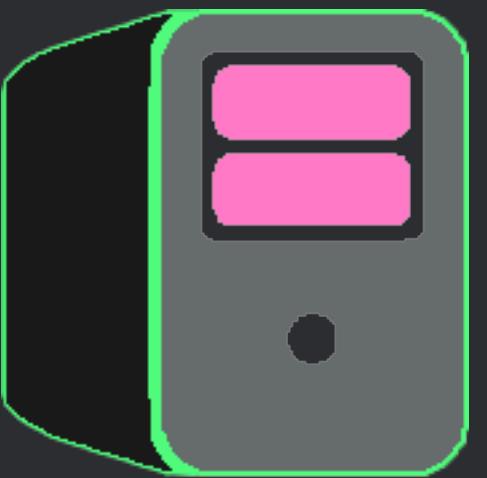
**RunLoop()**



Send()

RunLoop()

Server



true

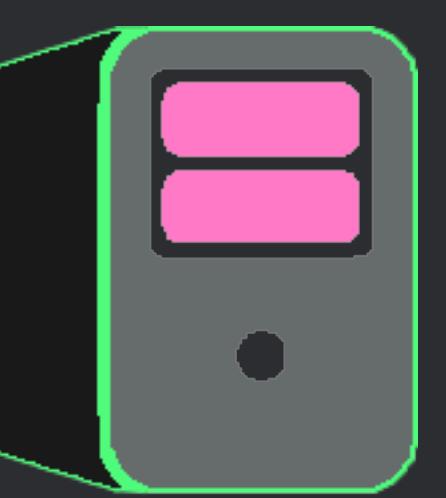


/change  
:8080



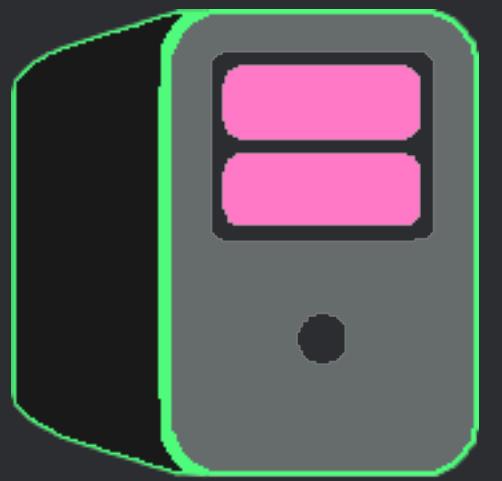
Send()

RunLoop()



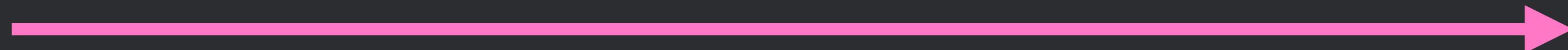
/change  
:8080

**Server**



**Send()**

**RunLoop()**



{ "change": true }

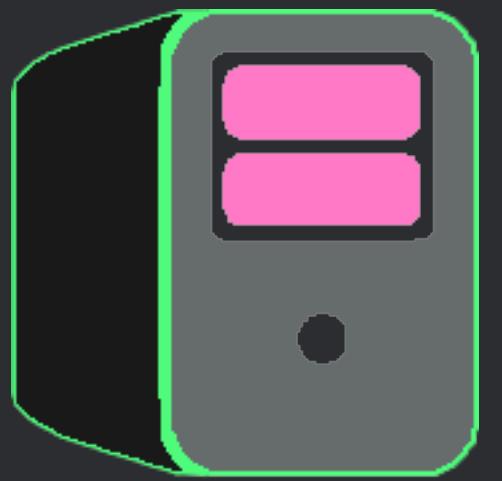


/change  
:8080

Send()

RunLoop()

Server



Process

Result



/change  
:8080



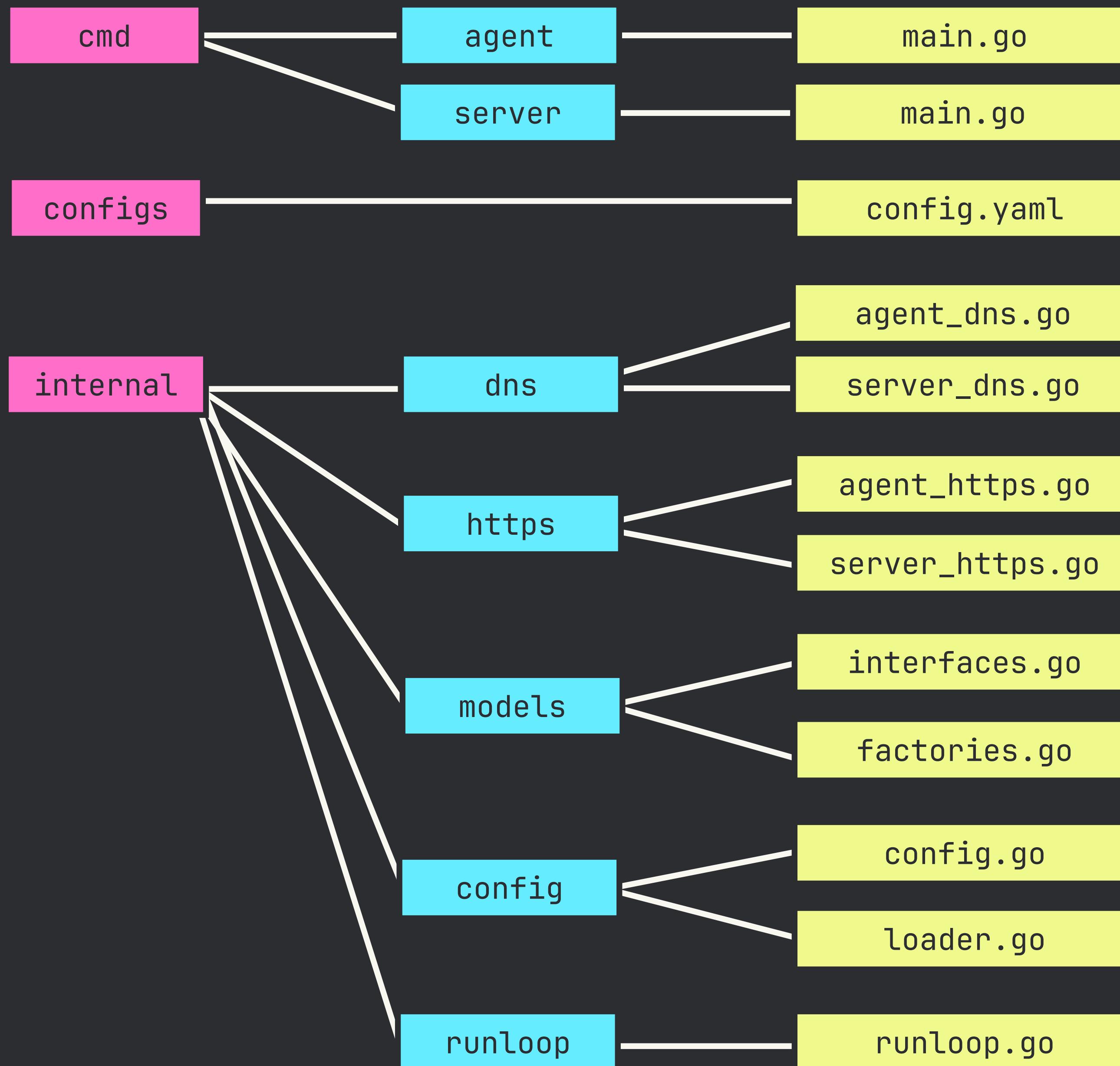
what we'll create

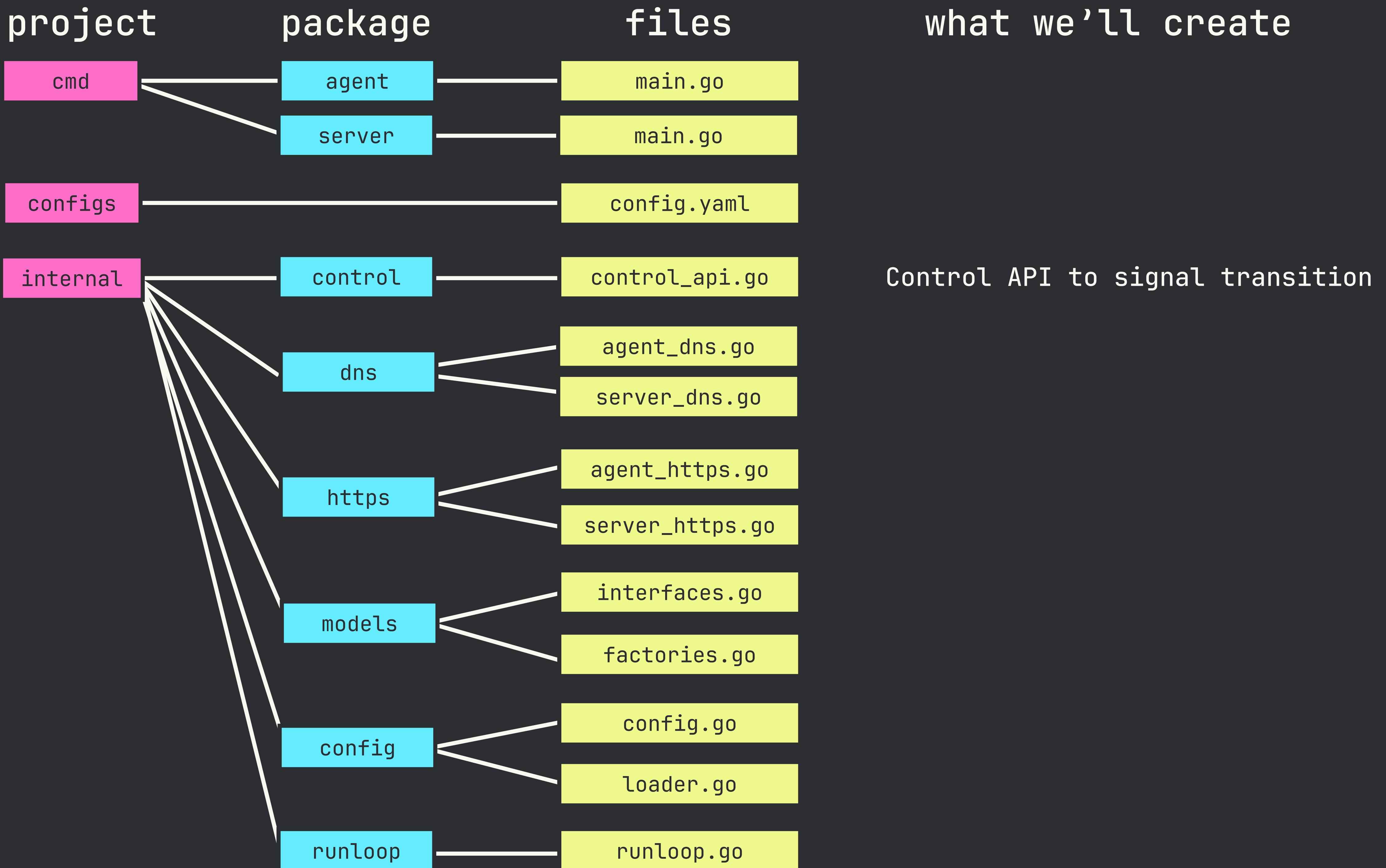
# project

# package

# files

# what we'll create









**NOW, ARE THERE  
ANY QUESTIONS?**

# lesson 10

both servers at  
the same time



# key concepts



we now have the ability to start the transition

- we hit an endpoint on the server
- the server sends “change” instruction to agent
- the agent will receive this and call back to the server to create this new connection (NEXT LESSON)

BUT... *something is missing*

- our agent will call back but *who is listening?*
- server has to listen for new *incoming connection*

## The “RIGHT” Way:

- ⊣ Let’s say we start with HTTPS
- ⊣ Server creates HTTPS listener on TCP :8443
- ⊣ Agent calls back, connection established
- ⊣ We hit transition EP set global flag to true
- ⊣ Server sends transition message to Agent, AND
- ⊣ Server starts DNS listener on UDP :8443
- ⊣ Agent calls back to and establishes DNS channel
- ⊣ Once confirmed, old HTTPS listener stopped

The thing about this is...

- It's a lot of logic/orchestration
- And in context of our short time together, not worth it imo

a “HACKY” way which will give us the same outcome...

- Just keep both listeners open at the same time
- Not optimal/elegant, but outcome will be the same
- As “homework”, you can refactor to “right” way



what we'll create







**NOW, ARE THERE  
ANY QUESTIONS?**

# Lesson 11

## agent parses and spawns new connection



# key concepts



we now have the ability to:

- signal our desire to change by hitting EP on server
- send instruction to the agent to do so
- receive the new callback on the server

what's missing is the ability for agent to call back

This is arguably the most complex lesson.

This being the case, let's orient ourselves by looking at our `runLoop()` implementation.

Specifically, let's look at the part where we process the response.

```
switch cfg.Protocol {  
    case "https":  
        // Parse and display response  
        var httpsResp https.HTTPSResponse  
        if err := json.Unmarshal(response, &httpsResp); err != nil {  
            log.Fatalf(format:"Failed to parse response: %v", err)  
        }  
        log.Printf(format: "Received response: change=%v", httpsResp.Change)  
  
    case "dns":  
        ipAddr := string(response)  
        log.Printf(format: "Received response: IP=%v", ipAddr)  
}  
}
```

- We **parse** response (**httpsResp.Change** and **ipAddr**)
- Then, all we do is **print** it

```
switch cfg.Protocol {  
    case "https":  
        // Parse and display response  
        var httpsResp https.HTTPSResponse  
        if err := json.Unmarshal(response, &httpsResp); err != nil {  
            log.Fatalf(format:"Failed to parse response: %v", err)  
        }  
        log.Printf(format: "Received response: change=%v", httpsResp.Change)  
  
    case "dns":  
        ipAddr := string(response)  
        log.Printf(format: "Received response: IP=%v", ipAddr)  
}  
}
```

Instead of just printing, we want to add the ability to switch from one protocol to the other based on the response.

we'll do this by embedding switch in an if-else

if transition signal detected → change to new protocol

else

```
switch cfg.Protocol {  
case "https":  
    // Parse and display response  
    var httpsResp https.HTTPSResponse  
    if err := json.Unmarshal(response, &httpsResp); err != nil {  
        log.Fatalf(format: "Failed to parse response: %v", err)  
    }  
    log.Printf(format: "Received response: change=%v", httpsResp.Change)  
  
case "dns":  
    ipAddr := string(response)  
    log.Printf(format: "Received response: IP=%v", ipAddr)  
}
```

There are 3 things we need to add to do this:

1. We need to track the current protocol (**var - change**)
  - This should be a **variable** so it can be **updated**
2. We need logic to **detect and handle transitions**
  - if “**true**” or “**69.69.69.69**” change current protocol
3. The ability to **transition to new protocol**
  - **call back to new listener**

keep these 3 actions in mind .



what we'll create







**NOW, ARE THERE  
ANY QUESTIONS?**

where to  
from here?



I will share some distinct ideas, but  
first just want to share the following  
reference - NUMINON

[github.com/faanross/numinon](https://github.com/faanross/numinon)

First thing you probably want to add  
is a **command-handling system**

# SIMPLER (switch-based) - Workshop 01

[faanross.com/antisyphon/workshop/moc/](http://faanross.com/antisyphon/workshop/moc/)

## Part E: Weaving It All Together

- [Agent Command Execution \(Lab 08\)](#)
- [Client UI Command + Results Handling \(Lab 09\)](#)
- [Server Receives + Queues Commands \(Lab 10\)](#)
- [Agent Retrieves Command From Queue \(Lab 11\)](#)
- [Agent Executes, Returns Result, Server Result Processing \(Lab 12\)](#)

# This system is “flat”

```
// Execute runs the specified command and returns the output
func Execute(cmd string) (string, error) {
    // Trim any whitespace
    cmd = strings.TrimSpace(cmd)

    // Check which command to run
    switch cmd {
    case "pwd":
        return Pwd()
    case "hostname":
        return Hostname()
    case "whoami":
        return WhoAmI()
    default:
        return "", fmt.Errorf("unknown command: %s", cmd)
    }
}
```

If you feel up to something  
slightly more “advanced”, but  
that’s much more extensible,  
maintainable, and near  
production-ready...

## NUMINON - Map → Orchestrators → Doers (interface + build tags)

All commands are registered to corresponding orchestrator in a map

└ ./internal/agent/agent/agent.go

```
func registerCommands(agent *Agent) { 1 usage  ↳ faanross
    agent.commandOrchestrators["upload"] = (*Agent).orchestrateUpload
    agent.commandOrchestrators["download"] = (*Agent).orchestrateDownload
    agent.commandOrchestrators["run_cmd"] = (*Agent).orchestrateRunCmd
    agent.commandOrchestrators["shellcode"] = (*Agent).orchestrateShellcode
    agent.commandOrchestrators["enumerate"] = (*Agent).orchestrateEnumerate
    agent.commandOrchestrators["morph"] = (*Agent).orchestrateMorph
    agent.commandOrchestrators["hop"] = (*Agent).orchestrateHop

}
```

Step 1 → Agent receives command from server, let's say “shellcode”

└ ./internal/agent/agent/runloop\_\*.go

```
// call executeTask if we do have a task
if taskResp.TaskAvailable {
    log.Println(v...: "|AGENT LOOP HTTP| -> Task is available.")
    log.Printf(format: "|AGENT LOOP HTTP| -> Task received (ID: %s, Cmd: %s). Executing...",
              taskResp.TaskID, taskResp.Command)
    a.executeTask(taskResp) // Execute the task (which will send results internally)
}
```

Step 2 → This is sent to executeTask(), performs map lookup

└ ./internal/agent/agent/execute\_tasks.go

```
orchestrator, found := a.commandOrchestrators[task.Command]

if found {
    result = orchestrator(a, task)
} else {
    log.Printf(format: "|WARN AGENT TASK| Received unknown command: '%s' (ID: %s)", task.Command, task.TaskID)
    result = models.AgentTaskResult{
        TaskID: task.TaskID,
        Status: models.StatusFailureUnknownCommand,
        Error:  fmt.Sprintf(format: "Agent does not recognize command: '%s'", task.Command),
    }
}
```

Step 3 → Calls the correct orchestrator, `orchestrateShellcode()`

↳ `./internal/agent/agent/orchestrator_shellcode.go`

- Marshalling/unmarshalling of args and results
- Agent-side cmd-specific validation
- Embedding cmd-specific structs in generic parent structs
- Preparing cmd-specific args for doer, process results

Step 4 → Calls the correct doer, DoShellcode()

└ ./internal/agent/command/shellcode/\*

```
└─ shellcode
    > doer_shellcode_mac.go
    > doer_shellcode_nix.go
    > doer_shellcode_win.go
    > interface_shellcode.go
```

```
//go:build windows
package shellcode
```

```
package shellcode

import "github.com/faanross/numinon/internal/models"

type CommandShellcode interface { 6 usages 1 implementation ✎ faanross
    DoShellcode(dllBytes []byte, targetPID uint32, shellcodeArgs []byte, exportName string) (models.ShellcodeResult, error)
}
```

Doer will perform the actual action!



[github.com/faanross/numinon](https://github.com/faanross/numinon)

Now, how about some **actual commands**?

- upload
- download
- shell command
- shellcode (reflective loader)
- enumerate (all or by process name)
- morph
- hop

- shellcode (reflective loader)
- entire free course on creating this
- lots of foundational maldev lessons

[faanross.com/firestarter/reflective/moc/](http://faanross.com/firestarter/reflective/moc/)

The screenshot shows a web browser window displaying a course page. The title of the page is "Let's Build a Reflective Loader in Golang". On the right side, there is a sidebar with a list of modules:

- Module 1: DLLs and Basic Loading
- Module 2: PE Format for Loaders
- Module 3: Reflective DLL Loading Core Logic
- Module 4: Handling Relocations and Imports
- Module 5: Execution and Exports
- Module 6: Basic Obfuscation - XOR
- Module 7: Rolling XOR & Key Derivation
- Module 8: Network Delivery & Client/Server

The main content area features a black and white photograph of a person with their mouth wide open, wearing a shirt with stars, possibly singing or shouting. Below the photo, the first module is detailed:

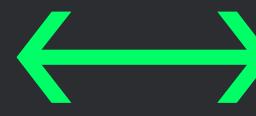
**Module 1: DLLs and Basic Loading**

- [Introduction to DLLs \(Theory 1.1\)](#)
- [Introduction to Shellcode \(Theory 1.2\)](#)
- [Standard DLL Loading in Windows \(Theory 1.3\)](#)
- [Create a Basic DLL \(Lab 1.1\)](#)
- [Create a Basic Loader in Go \(Lab 1.2\)](#)

For those interested in the FULL Course,  
that's what we'll be covering there!

| Communication +

| Commands +

| Operator  Server  Agent

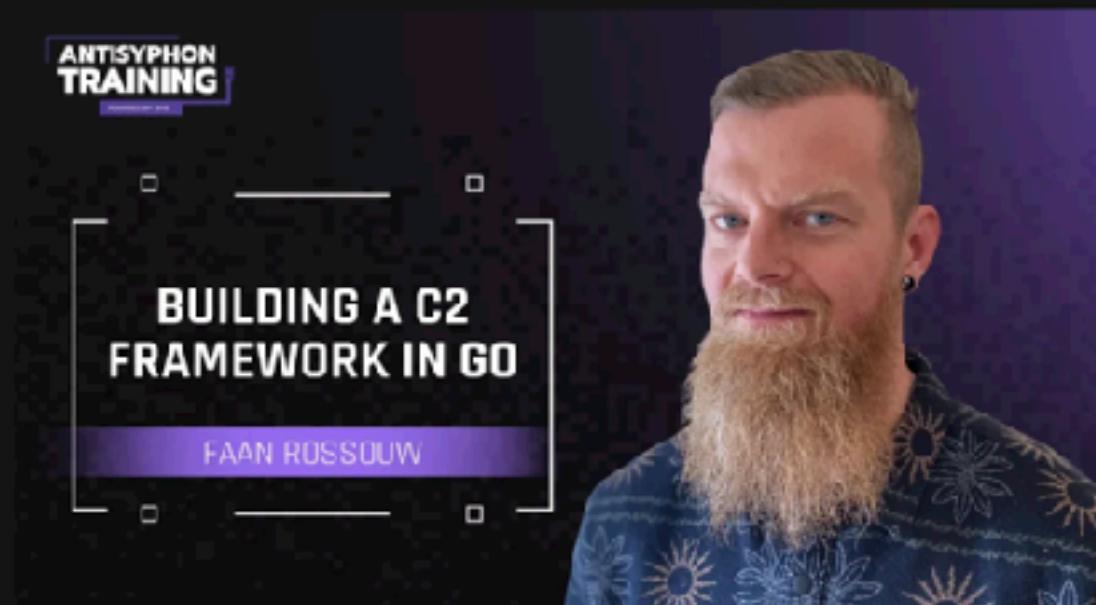
[www.antisyphontraining.com/product/building-a-c2-framework-in-go-with-faan-rossouw/](http://www.antisyphontraining.com/product/building-a-c2-framework-in-go-with-faan-rossouw/)

 [antisyphon training](#)

[COURSE CATALOG](#) [LIVE TRAINING](#) [ON-DEMAND](#) [WHO WE ARE](#) [CERTIFICATION](#) [CYBER RANGE](#) [CONTACT](#)

## Building a C2 Framework in Go with Faan Rossouw

Course Authored by [Faan Rossouw](#).



Join this hands-on course to build a fully functional Command and Control (C2) framework from the ground up using Golang.

Course Length: 16 Hours Includes a Certificate of Completion

# conclusion



For ongoing support

- | discord: **firestarter\_maldev**
- | email: **moi@faanross.com**
- | site: **www.faanross.com**

**[www.faanross.com/antisyphon/workshop02/moc/](http://www.faanross.com/antisyphon/workshop02/moc/)**

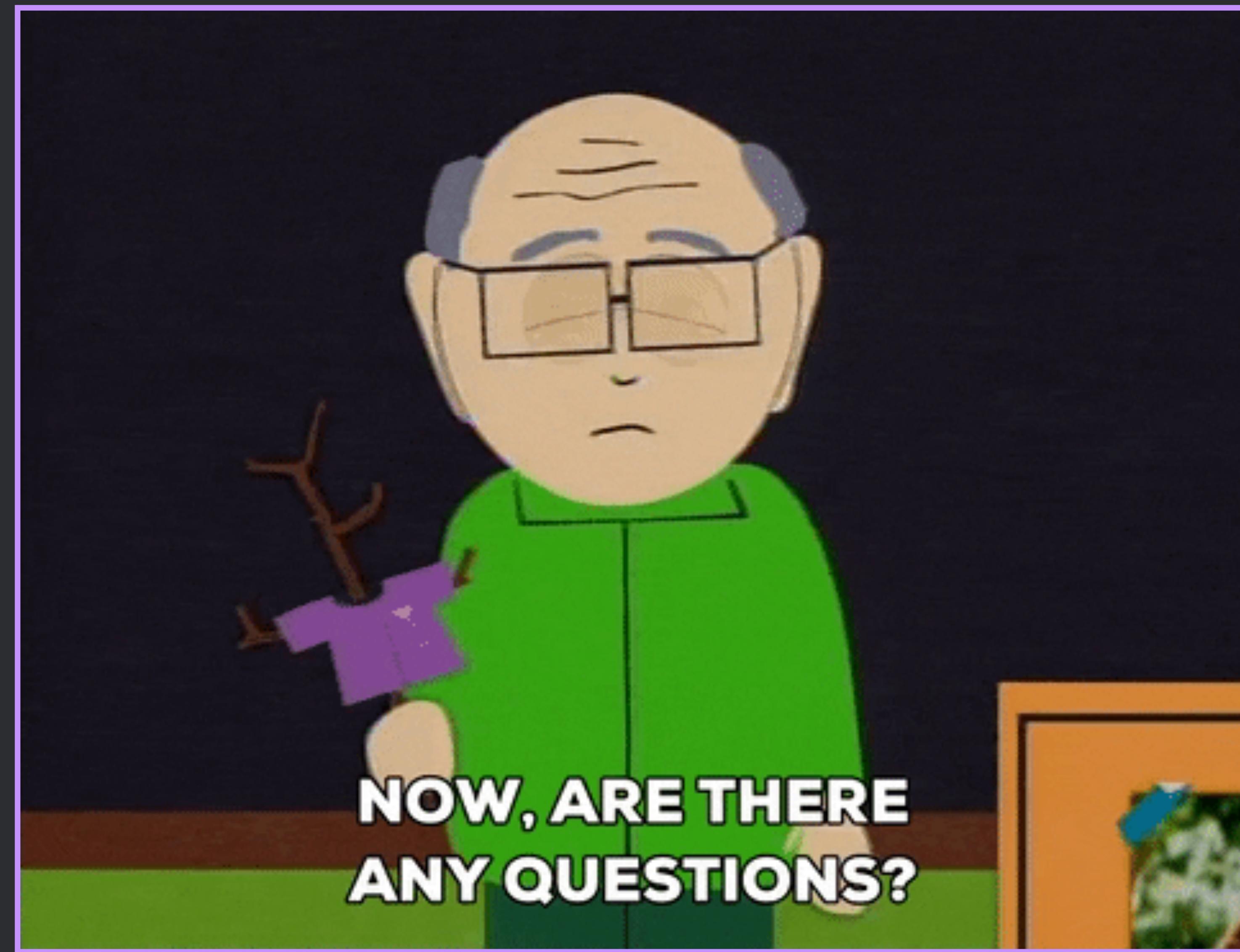
THANKS!

I appreciate the support  
I feel free to critique

live long and prosper!

X





**NOW, ARE THERE  
ANY QUESTIONS?**