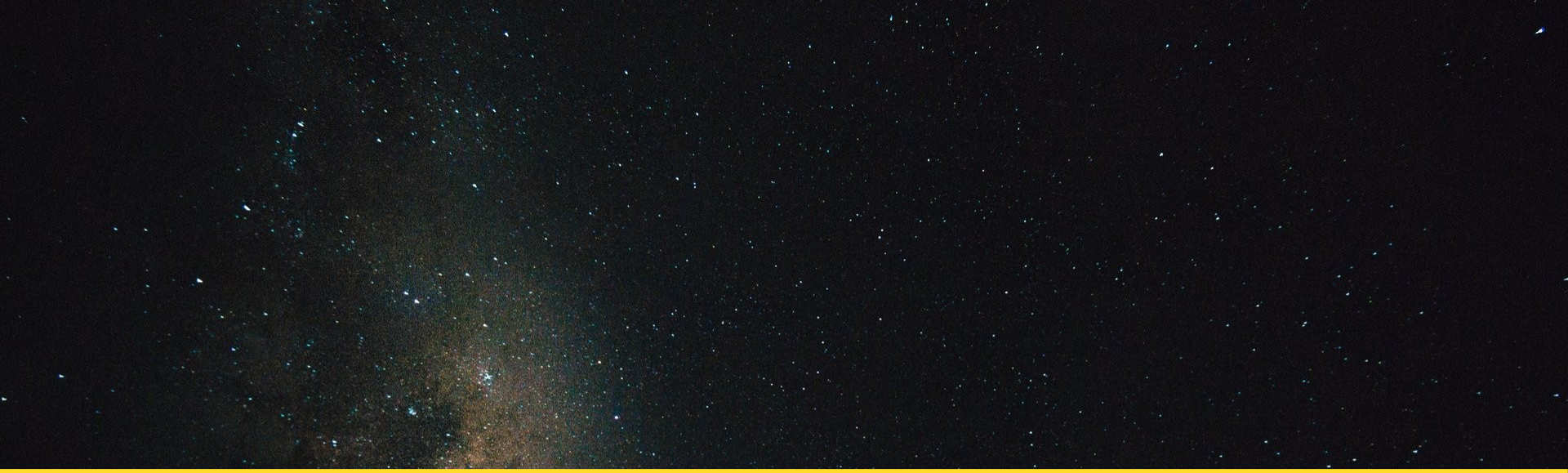


HTTP

Tinkoff.ru

План занятия

1. HTTP/REST
2. net/http client
3. net/http server
4. структура проекта
5. Go modules
6. роутер и мидлварь на пример go-chi/chi
7. пользовательские сессии
8. gracefull shutdown

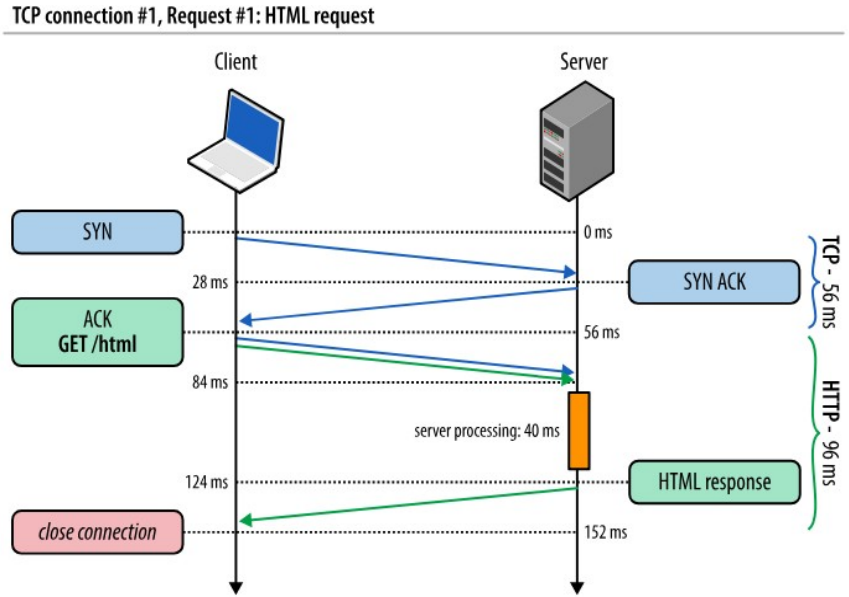
HTTP/REST

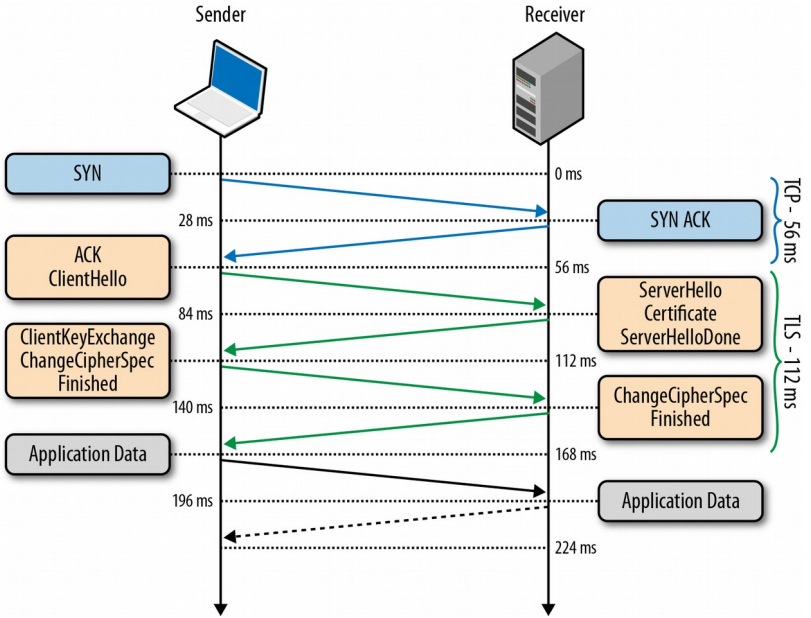
HTTP

HTTP(s) – протокол прикладного уровня, текстовый

- Применяется для организации сетевого взаимодействия между приложениями
- Предполагает режим «запрос-ответ»
- Stateless, то есть сервер не хранит контекст с предыдущего запроса

HTTP(S)


TCP connection #1, Request #1: HTML request
Client
Server
SYN
0 ms
SYN ACK
28 ms
ACK
GET /html
56 ms
84 ms
server processing: 40 ms
124 ms
HTML response
close connection
152 ms
TCP - 56 ms
HTTP - 96 ms


Sender
Receiver
SYN
0 ms
SYN ACK
28 ms
ACK
ClientHello
56 ms
ServerHello
Certificate
ServerHelloDone
84 ms
ClientKeyExchange
ChangeCipherSpec
Finished
112 ms
ChangeCipherSpec
Finished
140 ms
Application Data
168 ms
Application Data
196 ms
224 ms
TCP - 56 ms
TLS - 112 ms

Структура http сообщения

Пример запроса

```
POST /post HTTP/1.1
Host: postman-echo.com
Content-Type: application/json
Cache-Control: no-cache

{"name":"Demid"}
```

<- стартовая строка
<- заголовки

<- тело сообщения

Пример ответа

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Tue, 26 Feb 2019 21:48:02 GMT
Server: nginx
Vary: Accept-Encoding
Content-Length: 429
Connection: keep-alive

{“name”:”Demid”}
```

<- стартовая строка
<- заголовки

<- управляем кэшем

<- не закрываем TCP

<- тело сообщения

Основные методы

1) GET – получить данные
2) HEAD – аналогичный GET но без тела запроса
3) POST – передача пользовательских данных (неидемпотентный)
4) PUT – загрузка содержимого запроса (идемпотентный)
5) DELETE – удаление ресурса (идемпотентный ?)
6) OPTIONS– проверка возможности совершения операции над ресурсом

Коды ответа

2xx – успешный ответ

3xx – перенаправление, необходимо сделать запрос по другому URL

4xx – ошибка на стороне клиента

5xx – ошибка на стороне сервера

REST (Representation State Transfer)

Рой Филдинг в диссертации «*Архитектурные стили и дизайн сетевых программных архитектур*»

- Клиент – сервер
- Stateless
- Единообразные интерфейсы (URI ресурса, представление ресурса, самодостаточные сообщения)

REST в контексте HTTP

HTTP Глагол	Ресурс (например /customers)	Экземпляр (например /customers/{id})
GET	200 (OK), перечень customers. Используется постраничная навигация, сортировка и фильтрация для больших списков.	200 (OK), конкретный customer. 404 (Not Found), в случае отсутствия экземпляра с указанным ID или если он не корректен (а также, если клиенту не позволительно знать о наличии данного экземпляра).
PUT	404 (Not Found), если была попытка обновить/заменить экземпляр во всей коллекции.	200 (OK) или 204 (No Content). 404 (Not Found), в случае отсутствия экземпляра с указанным ID или если он не корректен (а также, если клиенту не позволительно знать о наличии данного экземпляра).
POST	201 (Created), заголовок 'Location' ссылается на /customers/{id}, где ID - идентификатор нового экземпляра.	404 (Not Found).
DELETE	404 (Not Found), если вы хотите удалить вс коллекцию, что крайне не желательно.	200 (OK) или 204 (No Content). 404 (Not Found), в случае отсутствия экземпляра с указанным ID или если он не корректен (а также, если клиенту не позволительно знать о наличии данного экземпляра).

Используем корректные статусы кодов и знания об идемпотентности методов

Представление ресурсов в виде JSON

curl/insomnia/postman

Curl – это утилита командной строки

Insomnia/postman – это десктоп-приложения

Пример GET запроса

```
curl -G https://ifconfig.co
109.252.36.0
```

Пример POST запроса

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"username":"xyz","password":"xyz"}' \
  https://postman-echo.com/post
{"args":{},"data":{"username":"xyz","password":"xyz"},…}
```

Проблемы HTTP/1.1

- Ограничения взаимодействия синхронного типа «запрос-ответ»
- Вопросы производительности (ограничения TCP)
- HTTP – текстовый протокол

net/http client

Get запрос клиентом по умолчанию

```
1  func main() {
2     resp, err := http.Get("https://uinames.com/api/?region=russia")
3     if err != nil {
4        log.Fatal(err)
5     }
6     defer resp.Body.Close()
7
8     body, err := ioutil.ReadAll(resp.Body)
9     if err != nil {
10       log.Fatal(err)
11    }
12
13
14    fmt.Printf("%s", body)
  }
```

```
{"name":"Варвара","surname":"Кац","gender":"female","region":"Russia"}
```

Request / Response

```
// net/http
type Request struct {
    Method string
    URL *url.URL
    Header Header
    Body io.ReadCloser
    ...
}
```

<- [scheme:][//[userinfo@]host][/]path[?query][#fragment]

```
// net/http
type Response struct {
    Status  string
    StatusCode int
    Header Header
    Body io.ReadCloser
    ...
}
```

<- "It is the caller's responsibility to close Body"

Client

Не используем клиента по умолчанию, так как у него не определен тайм-аут

```
1  // net/http
2  type Client struct {
3      Transport RoundTripper
4      CheckRedirect func(req *Request, via []*Request) error
5      Jar CookieJar
6      Timeout time.Duration
7  }
8
9  func main(){
10         ...
11         client := &http.Client{
12                   Timeout: time.Second * 10,
13          }
14          ...
15 }
```

Put запрос

```
1  reqBody := bytes.NewBuffer([]byte(`{"name":"Варвара"}`))
2  req, err :=
3  http.NewRequest(http.MethodPut,"https://postman-echo.com/put", reqBody)
4  if err != nil {
5         log.Fatal(err)
6  }
7  req.Header.Add("Content-type", "application/json")
8  resp, err := client.Do(req)
9  if err != nil {
10        log.Fatal(err)
11 }
12 defer resp.Body.Close()
13 body, err := ioutil.ReadAll(resp.Body)
14 if err != nil {
15        log.Fatal(err)
16 }
17 log.Printf("%s", body)
18 }
```

Дебажим http запрос

```go
req.Header.Add("Content-type", "application/json")

if debug {
   debugReq, err := httputil.DumpRequestOut(req, true)
   if err != nil {
      log.Fatal(err)
   }
   fmt.Printf("%s", debugReq)
}
resp, err := client.Do(req)
```

```
PUT /put HTTP/1.1
Host: postman-echo.com
User-Agent: Go-http-client/1.1
Content-Length: 16
Content-Type: application/json
Accept-Encoding: gzip

{"name":"Stefa"}
```

Дебажим http ответ

```go
defer resp.Body.Close()
if debug {
   debugResp, err := httputil.DumpResponse(resp, true)
   if err != nil {
      log.Fatal(err)
   }
   fmt.Printf("%s\n", debugResp)
}
body, err := ioutil.ReadAll(resp.Body)
```

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json; charset=utf-8
Date: Mon, 25 Feb 2019 23:29:32 GMT
Etag: W/"143-w49sQ/FxWR8yvADCFDnsIhKvZbo"
Server: nginx
Set-Cookie: sails.sid=s%3AxpKykyyUbFzW2o2lGMRaIDbAe-
dBmkpC.AWGyYiPYCORzjkIxiLeDbLtgomlp%2FBGex6WAnUWizeM; Path=/; HttpOnly
Vary: Accept-Encoding
…
```

Выставляем таймаут ожидания ответа

```
1  client := &http.Client{
2     Timeout: time.Millisecond * 50,
3  }
4  req, err := http.NewRequest("GET", "https://postman-echo.com/get", nil)
5  if err != nil {
6     log.Fatal(err)
7  }
8  resp, err := client.Do(req)
9  defer resp.Body.Close()
10 body, err := ioutil.ReadAll(resp.Body)
11 if err != nil {
```

panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x40 pc=0x63e82f]

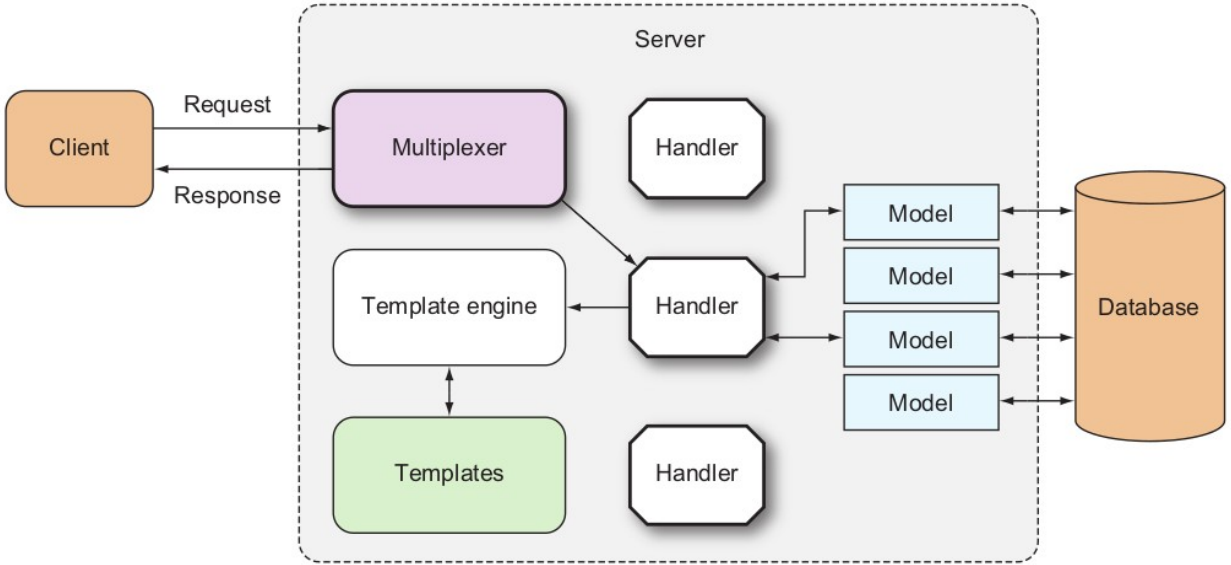
Get https://postman-echo.com/get: net/http: request canceled while waiting for connection
(Client.Timeout exceeded while awaiting headers)

Вывод

1. Не используйте клиента по умолчанию, так как у него отсутствует таймаут как таковой
2. Всегда создавайте свой клиент
3. Всегда определяйте таймаут, память не бесконечна

net/http server

Go Server


Server
Client
Request
Response
Multiplexer
Handler
Model
Model
Model
Model
Database
Template engine
Handler
Templates
Handler

Go Server

```
type Server struct {
    Addr string
    Handler Handler
    ReadTimeout time.Duration
    WriteTimeout time.Duration
    …
}
```

```
type ServeMux struct {
}
func (mux *ServeMux) Handle(pattern string,
handler Handler)
func (mux *ServeMux) Handler(r *Request) (h
Handler, pattern string)
func (mux *ServeMux) ServeHTTP(w
ResponseWriter, r *Request) <- Handler
interface
```

Наш первый http-сервер

```go
func Greeting(w http.ResponseWriter, r *http.Request) {
   fmt.Printf("%s method\n", r.Method)
   w.Write([]byte("Hello, anonymous!\n"))
}
func main() {
   http.HandleFunc("/", Greeting)
   if err := http.ListenAndServe(":5000", nil); err != nil {
      log.Fatal(err)
   }
}
```

```
victor@Victor:~$ curl -is http://localhost:5000
HTTP/1.1 200 OK
Date: Mon, 09 Mar 2020 11:57:44 GMT
Content-Length: 18
Content-Type: text/plain; charset=utf-8

Hello, anonymous!
```

Добавим роутинг

```
1  func Greeting(w http.ResponseWriter, r *http.Request) {
2     w.Write([]byte("Hello, anonymous!"))
3  }
4  func Buy(w http.ResponseWriter, r *http.Request) {
5     w.Header().Add("X-MY-LOCATION", "ALASKA")
6     w.Write([]byte("Buy, anonymous!"))
7     w.Header().Add("X-MY-LANGUAGE", "RU")
8  }
9  func main() {
10    http.HandleFunc("/", Greeting)
11    http.HandleFunc("/buy/", Buy)
12                                          , nil); err != nil {
13
14
15
```

```
victor@Victor:~$ curl -is http://localhost:5000
HTTP/1.1 200 OK
Date: Mon, 09 Mar 2020 12:02:41 GMT
Content-Length: 18
Content-Type: text/plain; charset=utf-8

Hello, anonymous!
victor@Victor:~$ curl -is http://localhost:5000/buy/
HTTP/1.1 200 OK
X-My-Location: ALASKA
Date: Mon, 09 Mar 2020 12:02:46 GMT
Content-Length: 16
Content-Type: text/plain; charset=utf-8

Buy, anonymous!
```

404 not found для корневого роута

```go
func Greeting(w http.ResponseWriter, r *http.Request) {
   if r.URL.Path != "/" {
      http.NotFound(w, r)
      return
   }
   fmt.Printf("%s method\n", r.Method)
   w.Write([]byte("Hello, anonymous!"))
}
```

```
victor@Victor:~$ curl -is http://localhost:5000/
HTTP/1.1 200 OK
Date: Mon, 09 Mar 2020 12:08:53 GMT
Content-Length: 17
Content-Type: text/plain; charset=utf-8

Hello, anonymous!victor@Victor:~$ curl -is http://localhost:5000/unknown-url/
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Mon, 09 Mar 2020 12:09:05 GMT
Content-Length: 19

404 page not found
```

Чтение входных заголовков

```go
switch r.Header.Get("Accept") {
case "text/plain":
   w.Header().Add("Content-type", "text/plain")
   w.Write([]byte("Hello, anonymous!"))
case "application/json":
   w.Header().Add("Content-type", "application/json")
   w.Write([]byte("{\"greeting\":\"Hello, anonymous\"}"))
default:
   w.Header().Add("Content-type", "text/plain")
   w.Write([]byte("Hello, anonymous!"))
}
```

```
➜ L4 git:(master) ✗ curl --header "Accept: application/json" \
   http://localhost:5000/
{"greeting":"Hello, anonymous"}%
➜ L4 git:(master) ✗ curl --header "Accept: text/plain" \
   http://localhost:5000/
Hello, anonymous!%
```

Обработка разных http-методов

```go
switch r.Method {
case "GET":
   w.Write([]byte("method get implementation"))
case "POST":
   w.Write([]byte("method post implementation"))
default:
   w.WriteHeader(http.StatusNotImplemented)
   w.Write([]byte("method not implemented"))
}
```

```
➜  L4 git:(master) ✗ curl --request PUT\
  http://localhost:5000/
method not implemented%
➜  L4 git:(master) ✗ curl --request GET\
  http://localhost:5000/
method get implementation%
➜  L4 git:(master) ✗ curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"username":"xyz","password":"xyz"}' \
  http://localhost:5000/
method post implementation%
```

Чтение входных данных

```go
switch r.Method {
case "GET":
   for k, v := range r.URL.Query() {
      fmt.Printf("%s:\t%s\n", k, v)
   }
   w.Write([]byte("method get implementation"))
case "POST":
   reqBody, err := ioutil.ReadAll(r.Body)
   fmt.Printf("%s\n", reqBody)
   w.Write([]byte("method post implementation"))
default:
   w.WriteHeader(http.StatusNotImplemented)
   w.Write([]byte("method not implemented"))
}
```

```
➜ L4 git:(master) ✗ curl --request GET\
  "http://localhost:5000/?q1=v1&q2=w2"
method get implementation%
```

```
q1:  [v1]
q2:  [w2]
```

Недостатки стандартной библиотеки

1. нет возможности указывать в url path parameter
2. нет возможности на уровне роута определять хэндлеры для http-методов
3. `/` является роутом по умолчанию, то есть при запросе несуществующего роута вместо 404 not found вернется результат работы DefaultHandler

Структура проекта

Основные каталоги

1. cmd

```
Содержит отдельные приложения вашего решения
```

2. internal

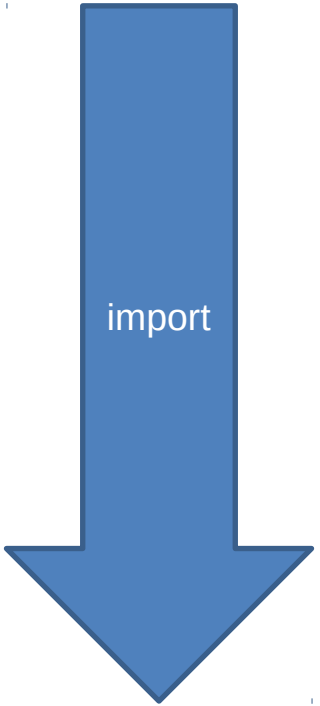
```
Содержит общие для ваших приложений пакеты
```

3. pkg

```
Содержит общие для ваших проектов пакеты
```

4. vendor

```
Сторонние библиотеки
```



Go Modules

Go modules

1. Переходим в каталог со своим проектом
2. Инициализируем модуль

```
go mod init example.com/example
```

В файле go.mod содержатся ваши зависимости (семантическое версионирование -> 1.9.0):

МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.

МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.

ПАТЧ-версию, когда вы делаете обратно совместимые исправления.

Go modules

- Удалить неиспользуемые зависимости

```
go mod tidy
```

- Создать директорию vendor

```
go mod vendor
```

- Просмотреть версии зависимостей

```
go list –m all
```

Go module proxy

- Кэширует и хранит все зависимости навсегда
- Можно не использовать директорию vendor (экономия места)
- Защита от изменения/удаления зависимости

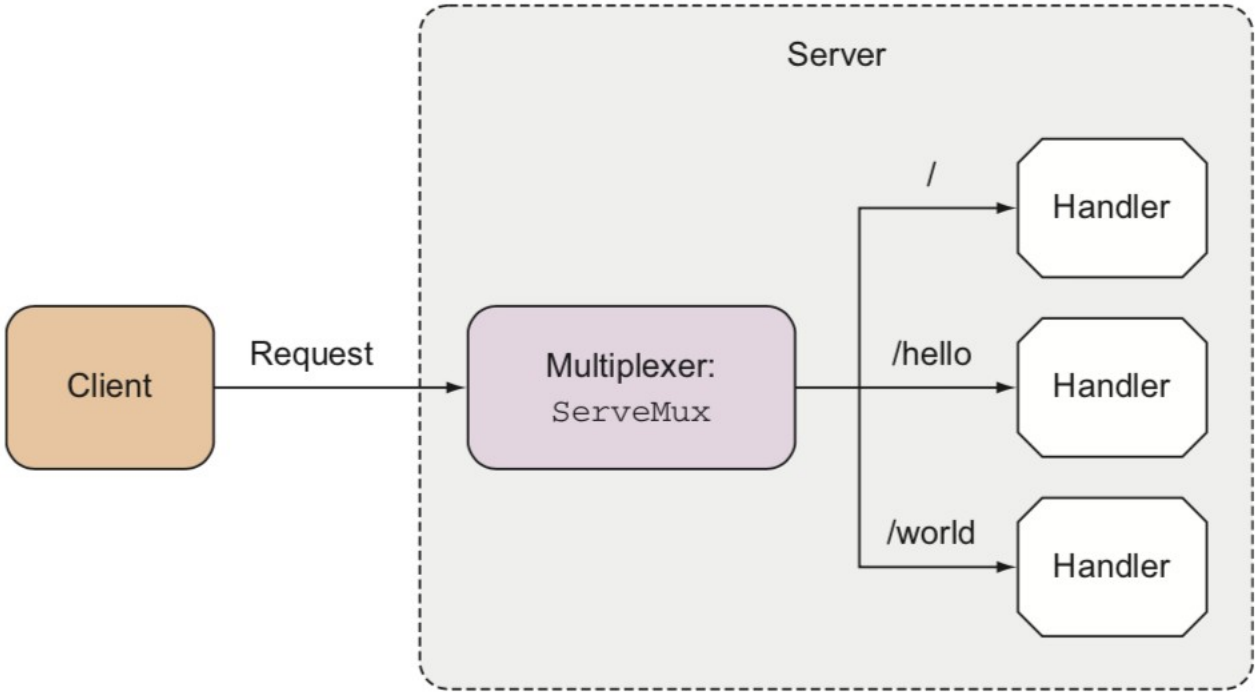
How to use : https://arslan.io/2019/08/02/why-you-should-use-a-go-module-proxy/

Подключаем go-chi/chi

Почему chi ?

```
BenchmarkBeego_Param                1000000              1098 ns/op             352 B/op          3 allocs/op
BenchmarkChi_Param                  2000000               866 ns/op             432 B/op          3 allocs/op
BenchmarkGin_Param                 30000000              53.2 ns/op               0 B/op          0 allocs/op
BenchmarkGocraftWeb_Param           1000000              1122 ns/op             648 B/op          8 allocs/op
BenchmarkGoji_Param                 3000000               501 ns/op             336 B/op          2 allocs/op
BenchmarkGorillaMux_Param            500000              2159 ns/op            1280 B/op         10 allocs/op
BenchmarkHttpRouter_Param          20000000              87.2 ns/op              32 B/op          1 allocs/op
BenchmarkMartini_Param               300000              3924 ns/op            1072 B/op         10 allocs/op
BenchmarkBeego_Param5               1000000              1243 ns/op             352 B/op          3 allocs/op
BenchmarkChi_Param5                 1000000              1135 ns/op             432 B/op          3 allocs/op
BenchmarkGin_Param5                20000000              88.3 ns/op               0 B/op          0 allocs/op
BenchmarkGocraftWeb_Param5          1000000              1789 ns/op             920 B/op         11 allocs/op
BenchmarkGoji_Param5                2000000               694 ns/op             336 B/op          2 allocs/op
BenchmarkGorillaMux_Param5           500000              3149 ns/op            1344 B/op         10 allocs/op
BenchmarkHttpRouter_Param5          5000000               252 ns/op             160 B/op          1 allocs/op
BenchmarkMartini_Param5              300000              4645 ns/op            1232 B/op         11 allocs/op
BenchmarkBeego_Param20               500000              2543 ns/op             352 B/op          3 allocs/op
BenchmarkChi_Param20                1000000              1677 ns/op             432 B/op          3 allocs/op
BenchmarkGin_Param20               10000000               231 ns/op               0 B/op          0 allocs/op
BenchmarkGocraftWeb_Param20          200000              6853 ns/op            3795 B/op         15 allocs/op
BenchmarkGoji_Param20               1000000              2133 ns/op            1246 B/op          2 allocs/op
BenchmarkGorillaMux_Param20          200000              7225 ns/op            3451 B/op         12 allocs/op
BenchmarkHttpRouter_Param20         2000000               858 ns/op             640 B/op          1 allocs/op
BenchmarkMartini_Param20             200000              9188 ns/op            3596 B/op         13 allocs/op
```

Общая схема мультиплексирования запросов

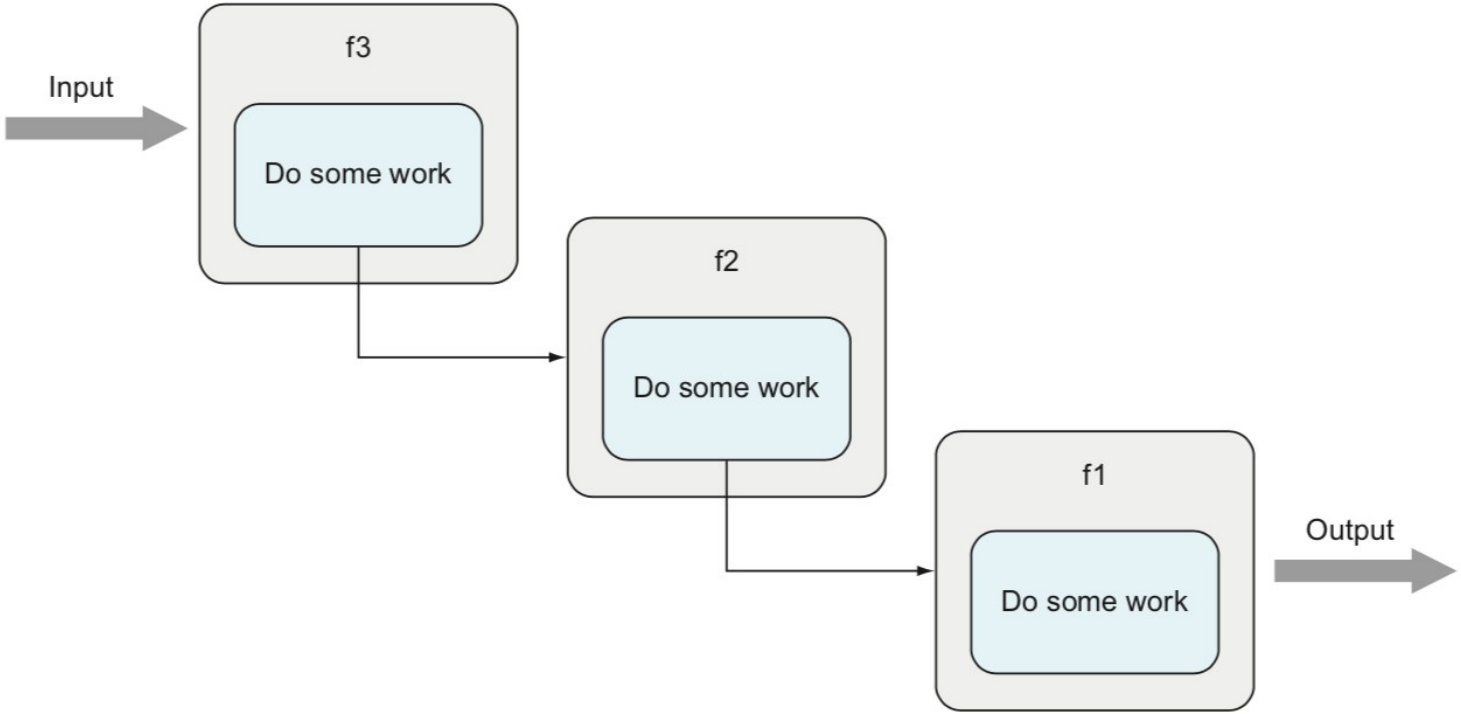


Первое приложение на chi

```go
func GetGreeting(w http.ResponseWriter, r *http.Request) {
   w.Write([]byte("Hello, anonymous!"))
}
func PostGreeting(w http.ResponseWriter, r *http.Request) {
   w.Write([]byte("Greetings broadcasted!"))
}
func main() {
   r := chi.NewRouter()
   r.Route("/", func(r chi.Router) {
      r.Get("/greeting", GetGreeting)
      r.Post("/greeting", PostGreeting)
   })
   http.ListenAndServe(":5000", r)
}
```

```
➜ L4 git:(master) ✗ curl --request GET  "http://localhost:5000/"
404 page not found
➜ L4 git:(master) ✗ curl --request GET  "http://localhost:5000/greeting"
Hello, anonymous!%
➜ L4 git:(master) ✗ curl --request POST  "http://localhost:5000/greeting"
Greetings broadcasted!%
```

Middleware handlers



Middleware handlers

```
const MaxConcurrentRequest = 100


r := chi.NewRouter()
r.Use(middleware.RequestID)
r.Use(middleware.RealIP)
r.Use(middleware.Throttle(MaxConcurrentRequest))


r.Route("/", func(r chi.Router) {
   r.Get("/greeting", GetGreeting)
   r.Post("/greeting", PostGreeting)
})


if err := http.ListenAndServe(":5000", r); err != nil {
   log.Fatal(err)
}
```

```
trackingID=sergeys-mbp/8HxJgaUptd-000005 RealIP=192.168.1.30 get greeting
requested
trackingID=sergeys-mbp/8HxJgaUptd-000006 RealIP=192.168.1.30 broadcasted
greeting
```

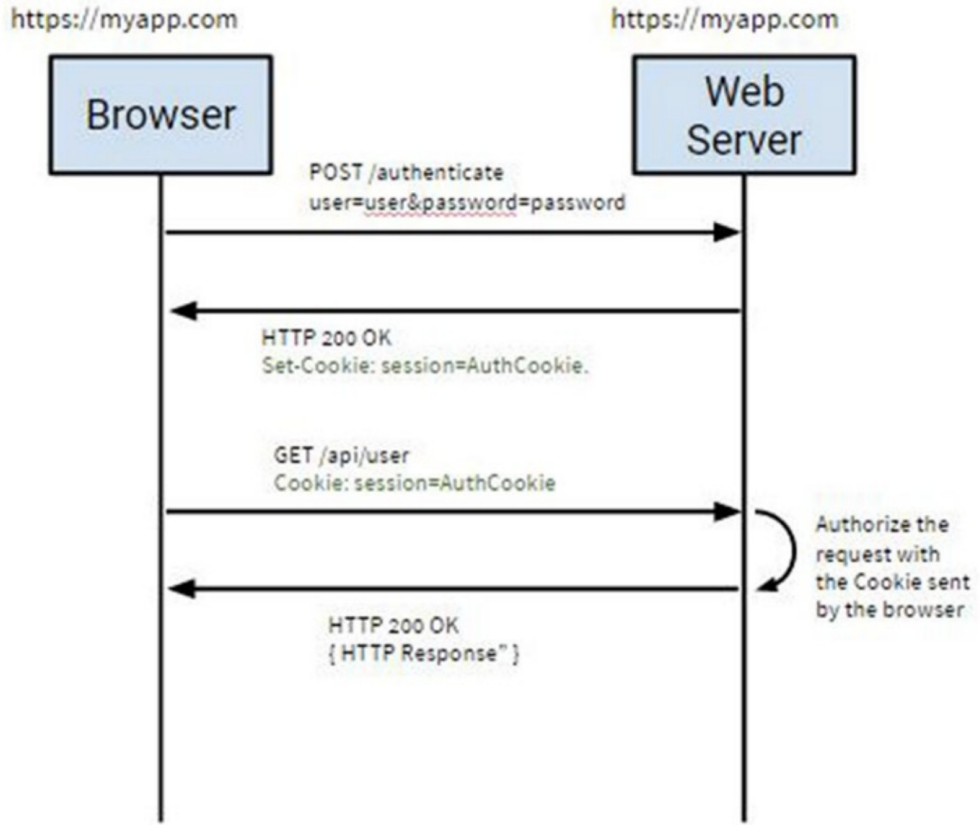
Пользовательские сессии

Аутентификация и авторизация

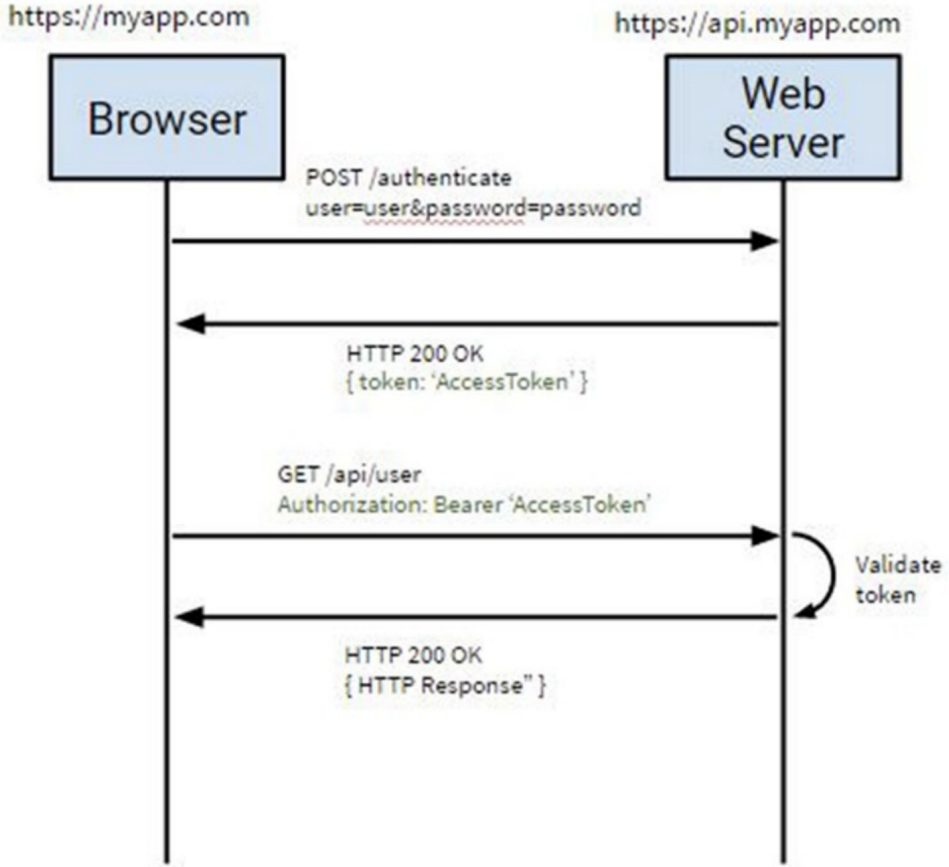
Аутентификация – это процедура проверки подлинности, то есть мы идентифицируем пользователя по его логину/паролю например.

Авторизация – это предоставление права пользователю на получение ресурса или выполнения действия(например отправить письмо).

Cookie-Based Authentication



Token-Based Authentication



Graceful shutdown

Какую проблему решает GraceShut?

Graceful Shutdown – корректное завершение работы приложения

1) Клиент ждет ответа на отправленный запрос, а вместо ответа получает ошибку **connection reset by peer**
2) Пайплайн был прерван в середине обработки, **данные были потеряны**
3) Приложение **не закрыло коннект** к внешнему ресурсу

Обработка сигналов

```
1  fmt.Println("Started application")
2  stopAppCh := make(chan struct{})
3  sigquit := make(chan os.Signal, 1)
4  signal.Ignore(syscall.SIGHUP, syscall.SIGPIPE)
5  signal.Notify(sigquit, syscall.SIGINT, syscall.SIGTERM)
6  go func() {
7     s := <-sigquit
8     fmt.Printf("captures signal: %v\n", s)
9     fmt.Println("graceful shutdown initiated")
10
11    // here we can grcefully close resources
12    stopAppCh <- struct{}{}
13 }()
14 <-stopAppCh
```

```
➜ graceful_shutdown git:(master) ✗ go run signals.go
Started application
^Ccaptures signal: interrupt
graceful shutdown initiated
➜ graceful_shutdown git:(master) ✗
```

Корректное завершение http-сервера

```
addr := net.JoinHostPort("", "5000")
srv := &http.Server{Addr: addr}
http.HandleFunc("/", Greeting)

go func() {
   s := <-sigquit
   fmt.Printf("captured signal: %v\n", s)
   if err := srv.Shutdown(context.Background()); err != nil {
      log.Fatalf("could not shutdown server: %s", err)
   }
   stopAppCh <- struct{}{}
}()
fmt.Printf("starting server, listening on %s\n", addr)

if err := srv.ListenAndServe(); err != http.ErrServerClosed {
   log.Fatal(err)
}
<-stopAppCh
```

Обратная связь

Спасибо за внимание

Tinkoff.ru