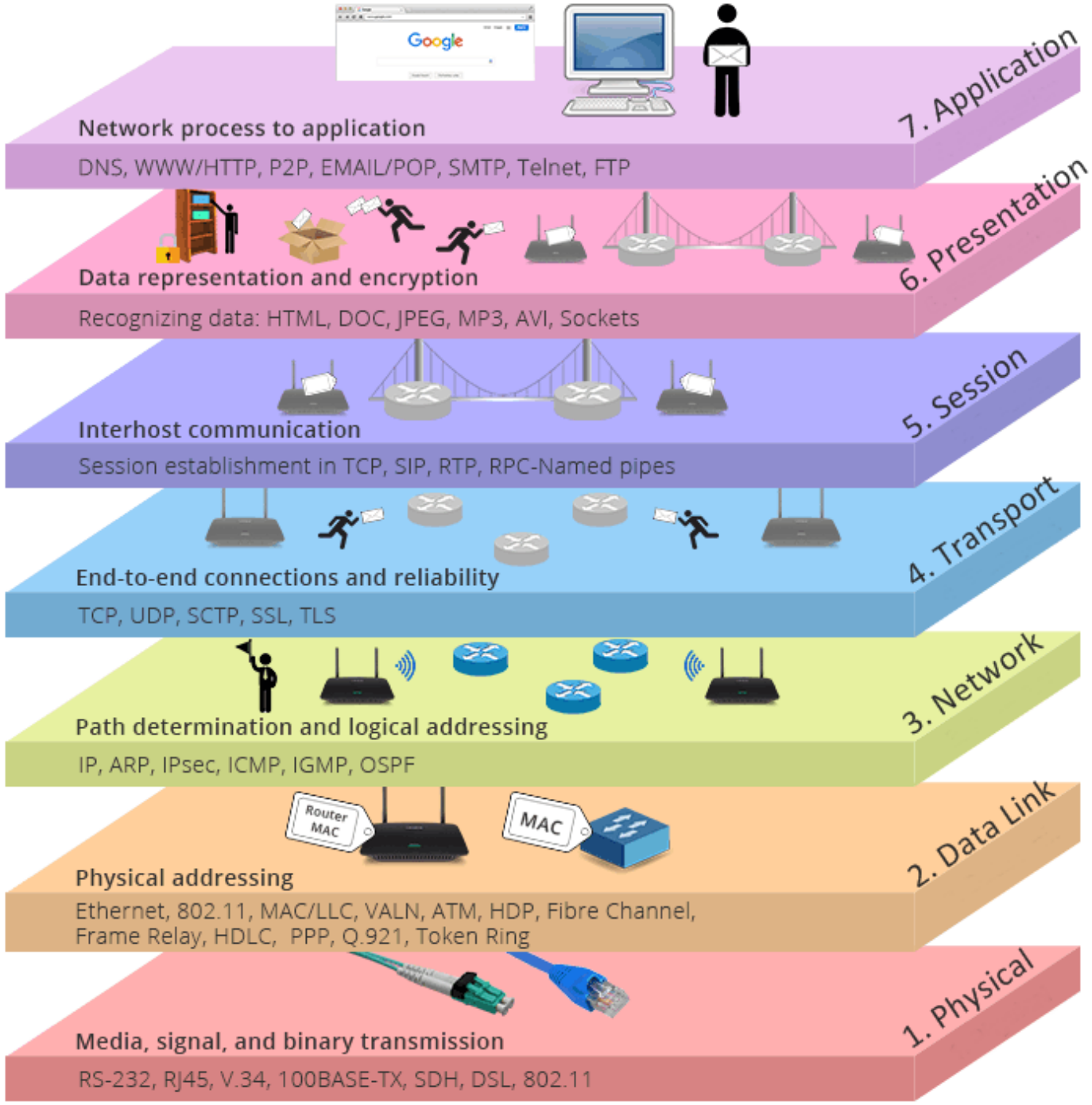
Лекция 5
HTTP

План занятия

- HTTP/REST
- net/http client/server
- go-chi/chi router and middlewares
- User sessions
- Graceful shutdown

Модель OSI

Модель OSI – модель, обмена данными по сети, состоящая из семи уровней, каждый из которых выполняет определенные функции.



HTTP

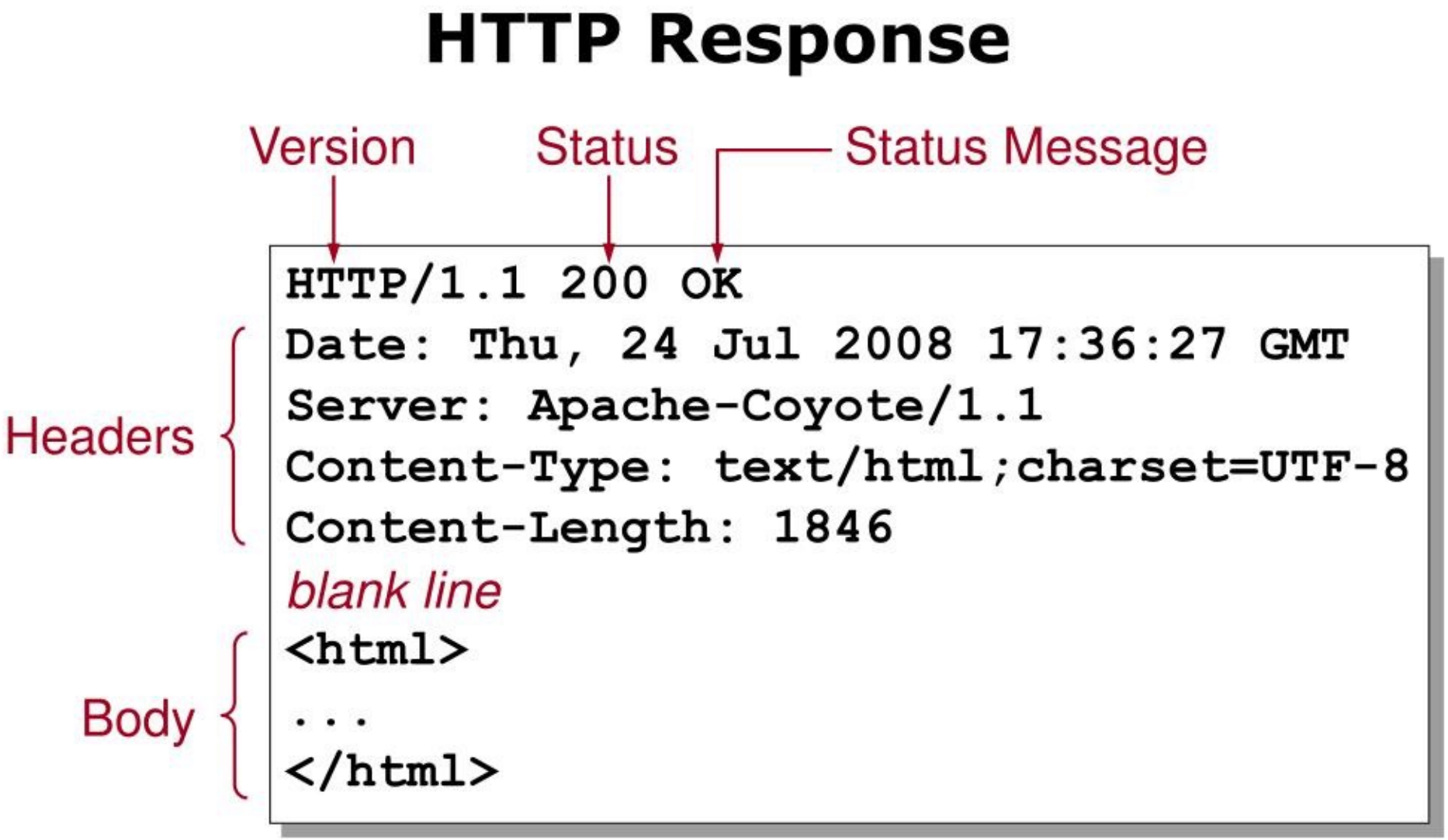
HTTP (Hypertext Transfer Protocol) — это протокол прикладного(седьмого) уровня, который используется для передачи данных в Интернете.

Основные особенности HTTP:

- 1. Клиент-серверная модель: запросы обрабатывает сервер
- 2. Текстовый протокол: данные передаются в виде текста
- 3. Методы HTTP: GET, POST, PUT…
- 4. Статус-коды: 1xx, 2xx, 3xx, 4xx, 500

структура http сообщений

структура http сообщений



Методы HTTP

HTTP метод	CRUD	Действие
GET	read	Возвращает запрошенные данные
POST	create	Создает новую запись
PUT or PATCH	update	Обновляет существующую запись
DELETE	delete	Удаляет существующую запись

Методы HTTP

- 1xx – информирование о процессе передачи (101 Switching Protocols)
- 2xx – успех (200 – ОК, 201 – Created, 202 – Accepted)
- 3xx – перенаправление (301 – Moved Permanently)
- 4xx – ошибка клиента (400 – Bad request, 401 – Unauthorized, 404 – Not Found)
- 5xx - ошибка сервера (500 – Internal Server Error)

Статус-коды HTTP

- 1xx – информирование о процессе передачи (101 Switching Protocols)
- 2xx – успех (200 – ОК, 201 – Created, 202 – Accepted)
- 3xx – перенаправление (301 – Moved Permanently)
- 4xx – ошибка клиента (400 – Bad request, 401 – Unauthorized, 404 – Not Found)
- 5xx - ошибка сервера (500 – Internal Server Error)

Статус-коды HTTP

- 1xx – информирование о процессе передачи (101 Switching Protocols)
- 2xx – успех (200 – ОК, 201 – Created, 202 – Accepted)
- 3xx – перенаправление (301 – Moved Permanently)
- 4xx – ошибка клиента (400 – Bad request, 401 – Unauthorized, 404 – Not Found)
- 5xx - ошибка сервера (500 – Internal Server Error)

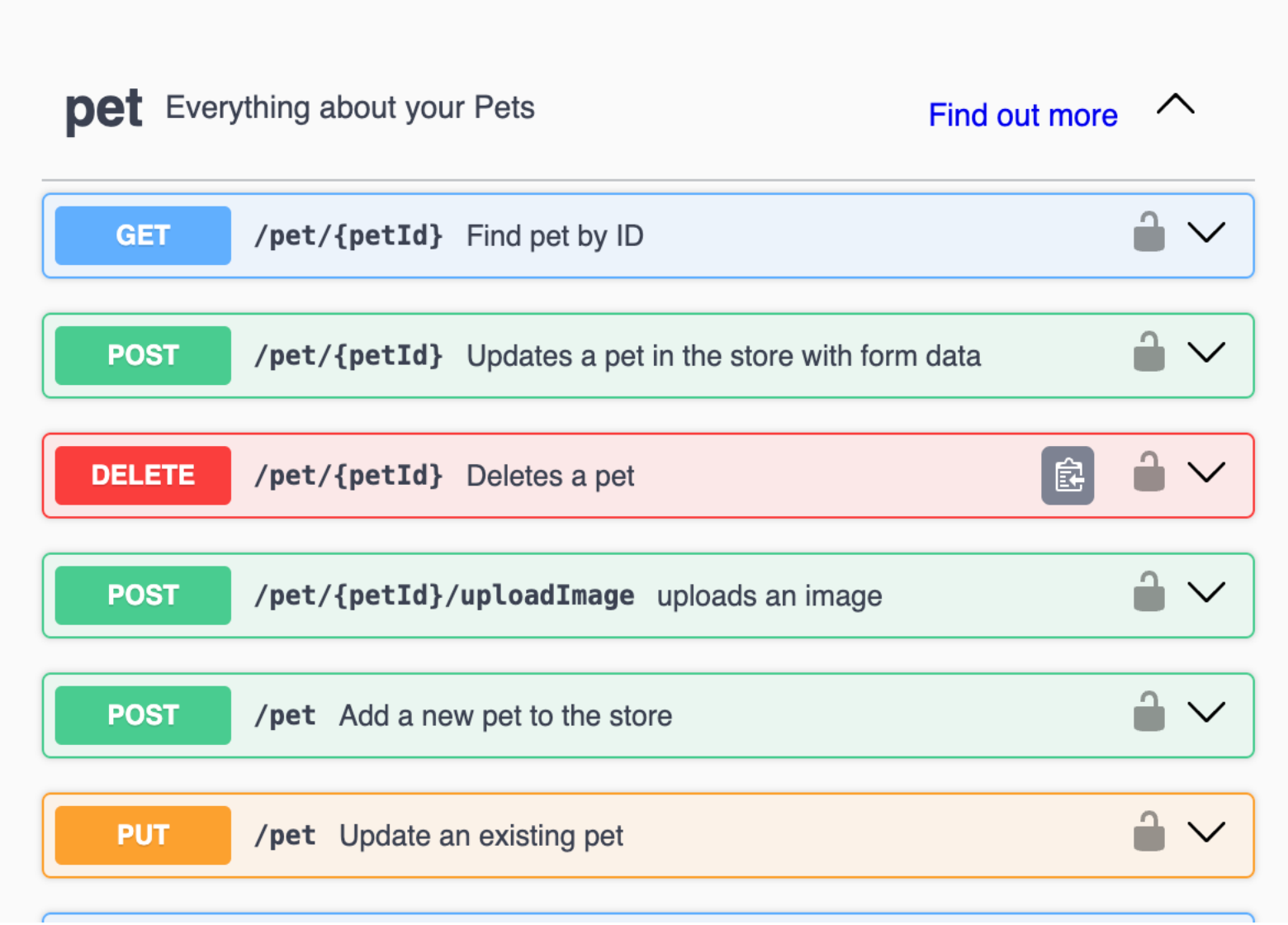
REST и HTTP

Принципы REST:

1. Модель клиент-сервер
2. Отсутствие состояния
3. Кэширование
4. Единообразие интерфейса
5. Слои
6. Код по требованию

REST описывает принципы взаимодействия клиента и сервера, основанные на понятиях «ресурса» и «глагола».

В случае HTTP ресурс определяется своим URI, а глагол — это HTTP-метод.



https://petstore.swagger.io/#/

Статус-коды HTTP

- 1xx – информирование о процессе передачи (101 Switching Protocols)
- 2xx – успех (200 – ОК, 201 – Created, 202 – Accepted)
- 3xx – перенаправление (301 – Moved Permanently)
- 4xx – ошибка клиента (400 – Bad request, 401 – Unauthorized, 404 – Not Found)
- 5xx - ошибка сервера (500 – Internal Server Error)

net/http

```
type Request struct {
    Method  string
    URL     *net.URL
    Header  Header // map[string][]string
    Body    io.ReadCloser
    ...
}
```

```
type Response struct {
    Status     string // "200 OK"
    StatusCode int    // 200
    Header     Header // map[string][]string
    Body       io.ReadCloser
    ...
}
```

net/http Client

```go
resp, err := http.Get("http://google.com/robots.txt")
if err != nil {
    panic(err)
}

defer resp.Body.Close()
data, err := io.ReadAll(resp.Body)
if err != nil {
    panic(err)
}

fmt.Println(string(data))
```

- Мало умеет (GET, HEAD, POST)
- Не передать заголовки
- Нет таймаута (!)

net/http Client

```go
req, err := http.NewRequest(http.MethodGet, "http://google.com/robots.txt", nil)
if err != nil {
    panic(err)
}

c := http.Client{
    Timeout: time.Second * 10,
}

resp, err := c.Do(req)
if err != nil {
    panic(err)
}

defer resp.Body.Close()
data, err := io.ReadAll(resp.Body)
if err != nil {
    panic(err)
}

fmt.Println(string(data))
```

- Любые методы
- Настраиваемый таймаут

net/http Client

```
1    v := url.Values{}
2    v.Add("id", "1")
3    queryString := v.Encode()
4
5    req, err := http.NewRequest(http.MethodGet, "http://google.com/robots.txt" + "?"
     + queryString, nil)
```

Добавление query параметров

```
1    req, err := http.NewRequest(http.MethodGet, "http://google.com/robots.txt", nil)
2    if err != nil {
3        panic(err)
4    }
5
6    req.Header.Add("User-Agent", "Mozilla/5.0 (X11; Linux i686; rv:2.0.1)
  Gecko/20100101 Firefox/4.0.1")
```

Управление заголовками

debug req/resp

```go
b, err := httputil.DumpRequestOut(req, true)
if err != nil {
    panic(err)
}
fmt.Println(string(b))
```

```
POST /robots.txt?id=1 HTTP/1.1
Host: google.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:2.0.1)
Gecko/20100101 Firefox/4.0.1
Content-Length: 18
Accept-Encoding: gzip

Hello and welcome!
```

```go
b, err = httputil.DumpResponse(resp, true)
if err != nil {
    panic(err)
}
fmt.Println(string(b))
```

```
HTTP/2.0 200 OK
Accept-Ranges: bytes
Cache-Control: private, max-age=0
Content-Type: text/plain
Cross-Origin-Opener-Policy-Report-Only: same-origin;
report-to="static-on-bigtable"
Cross-Origin-Resource-Policy: cross-origin
...

User-agent: *
Disallow: /search
Allow: /search/about
Allow: /search/static
Allow: /search/howsearchworks
...
```

net/http Server

```go
func HelloHandler(w http.ResponseWriter, r *http.Request) {
    _, _ = w.Write([]byte("hello, World!"))
}
```

```go
func main() {
    http.HandleFunc("/", HelloHandler)
    log.Fatal(http.ListenAndServe(":5000", nil))
}
```

```go
func main() {
    s := http.Server{
        Addr: ":5000",
        Handler: nil,
        ReadTimeout: time.Second,
    }
    http.HandleFunc("/", HelloHandler)
    log.Fatal(s.ListenAndServe())
}
```

Parameters handling

```go
func AdvancedHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("method: %s\n", r.Method)
    fmt.Printf("query values: %v\n", r.URL.Query())
    fmt.Printf("headers: %v\n", r.Header)

    body, err := io.ReadAll(r.Body)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    fmt.Printf("body: %s\n", string(body))

    w.WriteHeader(http.StatusAccepted)
}
```

```go
    http.HandleFunc("/", HelloHandler)
    http.HandleFunc("/advanced", AdvancedHandler)
    log.Fatal(s.ListenAndServe())
```

Недостатки стандартной библиотеки

- нет возможности указывать в url path parameter
- нет возможности на уровне роута хэндлить http-методы
- `/` является роутом по умолчанию – нет 404 статуса

<- кастомные роутеры

Роутер go-chi/chi

go-chi/chi

```go
func PostHello(w http.ResponseWriter, r *http.Request) {
    id := chi.URLParam(r, "id")
    _, _ = w.Write([]byte("POST hello, " + id))
}

func GetHello(w http.ResponseWriter, r *http.Request) {
    id := chi.URLParam(r, "id")
    _, _ = w.Write([]byte("POST hello, " + id))
}

func main() {
    r := chi.NewRouter()
    r.Get("/hello/{id}", GetHello)
    r.Post("/hello/{id}", PostHello)

    log.Fatal(http.ListenAndServe(":5000", r))
}
```

Middleware

Это функции, которые обрабатывают HTTP-запросы в веб-приложениях непосредственно пред тем, как запрос попадает в основной обработчик.

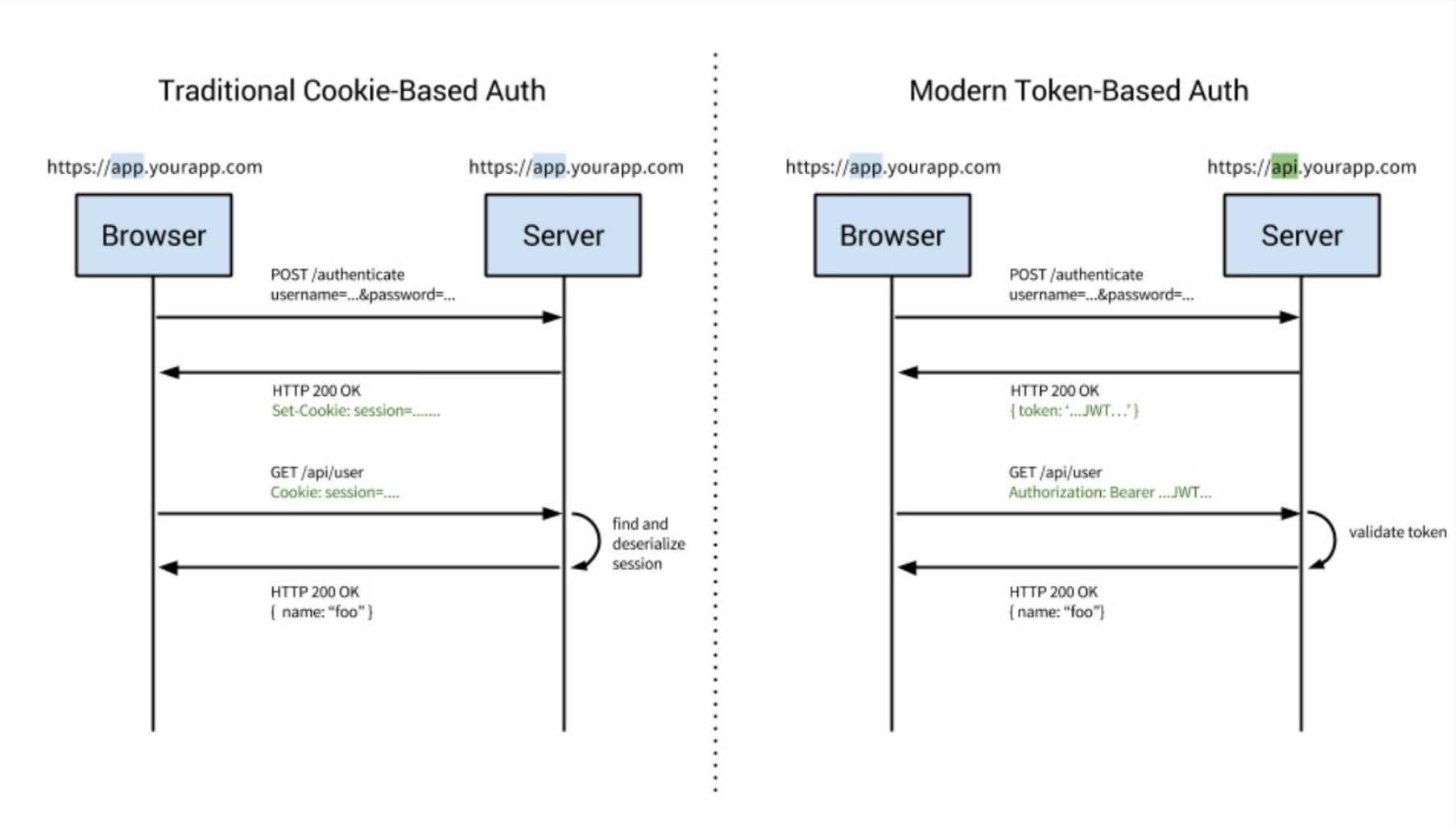
Используются для аутентификации, логирования, обработки ошибок, модификации запросов и ответов и т.д.

middlewares

```
1     r := chi.NewRouter()
2     r.Use(middleware.RequestID)
3     r.Use(middleware.Logger)
4
5     r.Get("/hello", GetHello)
6
7     log.Fatal(http.ListenAndServe(":5000", r))
```

```
2021/10/10 12:21:31 [macbook-s.garatuev/9QyU8dFSm4-000001] "GET
http://localhost:5000/hello HTTP/1.1" from [::1]:51986 - 200 10B in 17.257µs
```

Аутентификация и авторизация



Graceful shutdown

Это процесс корректного завершения работы сервера, при котором все активные соединения и процессы завершаются корректно, без потери данных или нарушения работы сервера.

Graceful shutdown

```go
func main() {
    mainSrv := http.Server{Addr: ":5000"}
    sigquit := make(chan os.Signal, 1)
    signal.Ignore(syscall.SIGHUP, syscall.SIGPIPE)
    signal.Notify(sigquit, syscall.SIGINT, syscall.SIGTERM)
    stopAppCh := make(chan struct{})
    go func() {
        log.Println("Captured signal: ", <-sigquit)
        log.Println("Gracefully shutting down server...")
        if err := mainSrv.Shutdown(context.Background()); err != nil {
            log.Println("Can't shutdown main server: ", err.Error())
        }
        stopAppCh <- struct{}{}
    }()
    <-stopAppCh
}
```

Вопросы