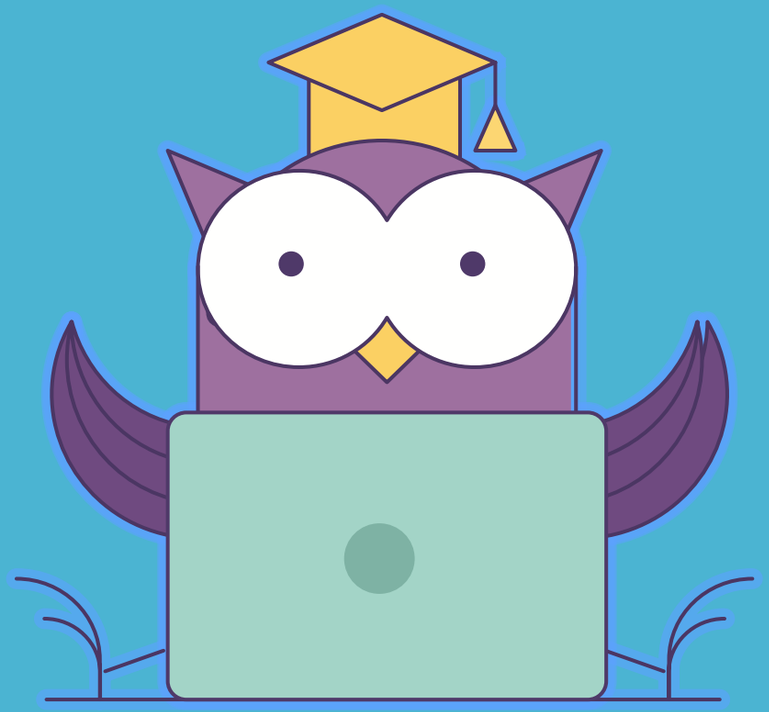
ОНЛАЙН-ОБРАЗОВАНИЕ

Как меня слышно и видно?



> Напишите в чат

```
+ если все хорошо
- если есть проблемы со звуком или с видео
```

OTUS

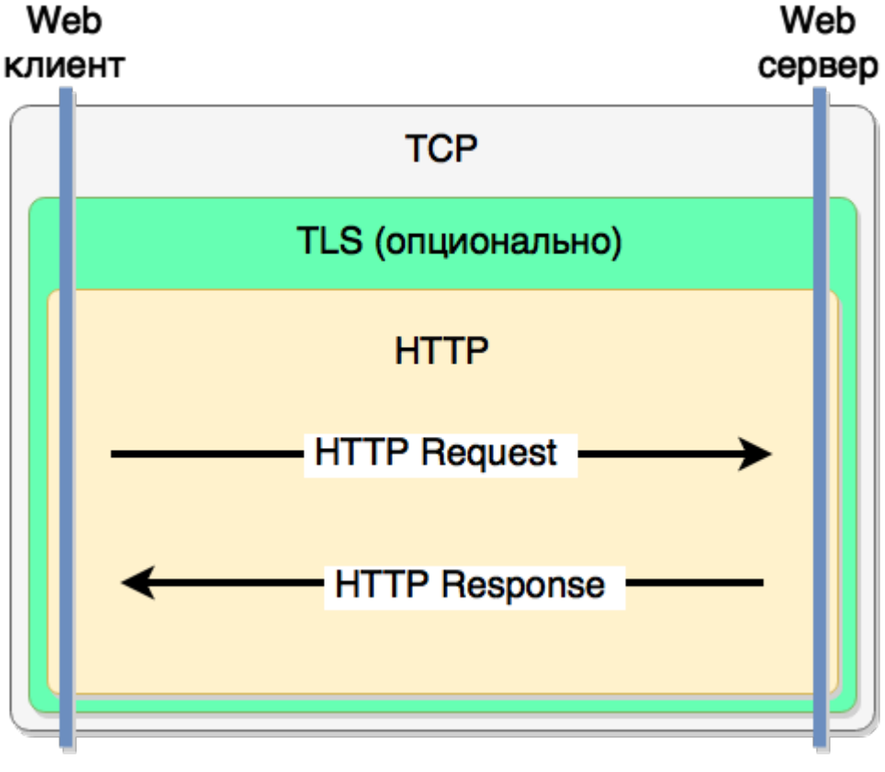
Протокол HTTP

Алексей Бакин

Как проходит занятие

- Активно участвуем - задаем вопросы.
- Чат вижу - могу ответить не сразу.
- После занятия - оффтопик, ответы на любые вопросы.

- HTTP клиент и сервер на Go
- Middleware
- Построение API сервиса


Web
клиент
Web
сервер
TCP
TLS (опционально)
HTTP
HTTP Request
HTTP Response

- Передача документов
- Передача мета-информации
- Авторизация
- Поддержка сессий
- Кеширование документов
- Согласование содержимого (negotiation)
- Управление соединением

- Работает поверх TCP/TLS
- Протокол запрос-ответ
- Не поддерживает состояние (соединение) - *stateless*
- *Текстовый* протокол
- Расширяемый протокол

```
GET /search?query=go+syntax&limit=5 HTTP/1.1
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip, deflate
Cache-Control: max-age=0
Connection: keep-alive
Host: site.ru
User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/39.0
```

```
POST /add_item HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Cache-Control: max-age=0
Connection: keep-alive
Host: www.ru
Content-Length: 42
Content-Type: application/json

{"id":123,"title":"for loop","text":"..."}
```

Перевод строки - `\r\n`

HTTP ответ

```
HTTP/1.1 404 Not Found
Server: nginx/1.5.7
Date: Sat, 25 Jul 2015 09:58:17 GMT
Content-Type: text/html; charset=iso-8859-1
Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>...
```

- `HTTP/2` - бинарный протокол
- используется мультиплексирование потоков
- сервер может возвращать еще не запрошенные файлы
- используется `HPACK` сжатие заголовков

HTTP клиент - GET

```
func main() {
    resp, err := http.Get("http://127.0.0.1:7070/")
    if err != nil {
        log.Fatal(err)
    }
    defer resp.Body.Close() // <-- Зачем?

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Fatal(err)
    }
    ...
}
```

HTTP клиент - GET

```
http://site.ru/search?query=...&limit=...
```

```
    reqArgs := url.Values{}
    reqArgs.Add("query", "go syntax")
    reqArgs.Add("limit", "5")

    reqUrl, _ := url.Parse("http://site.ru/search")
    reqUrl.RawQuery = reqArgs.Encode()

    req, _ := http.NewRequest("GET", reqUrl.String(), nil)
    req.Header.Add("User-Agent", `Mozilla/5.0 Gecko/20100101 Firefox/39.0`)

    resp, err := http.DefaultClient.Do(req)
```

https://goplay.space/#QHza-h5jNm2
https://go.dev/play/p/QHza-h5jNm2

```
client := http.Client{
    Transport: &http.Transport{
        MaxIdleConns:    100,
        IdleConnTimeout: 90 * time.Second,
    },
}
```

https://pkg.go.dev/net/http#Client

```
type AddRequest struct {
    Id    int    `json:"id"`
    Title string `json:"title"`
    Text  string `json:"text"`
}

...

    addReq := &AddRequest{
        Id:    123,
        Title: "for loop",
        Text:  "...",
    }

    jsonBody, _ := json.Marshal(&addReq)

    req, err := http.NewRequest("POST", "https://site.ru/add_item",
        bytes.NewBuffer(jsonBody))

    resp, err := http.DefaultClient.Do(req)
```

HTTP клиент - работа с ответом

```
resp, err := client.Do(req)
if err != nil {
    return fmt.Errorf("do request: %w", err)
}
defer resp.Body.Close()

if resp.StatusCode != 200 {
    return fmt.Errorf("%w: %s", errUnexpectedHTTPStatus, resp.Status)
}

ct := resp.Header.Get("Content-Type")
if ct != "application/json" {
    return fmt.Errorf("%w: %s", errUnexpectedContentType, ct)
}

body, err := ioutil.ReadAll(resp.Body)
```

Создать новый реквест:

```
req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://site.ru/some_api", nil)
```

Обогатить существующий:

```
req = req.WithContext(ctx)

resp, err := h.client.Do(req)

// ...
```

```
ctx := context.Background()
ctx, cancel := context.WithTimeout(ctx, 3*time.Second)
defer cancel()

req = req.WithContext(ctx)

resp, err := client.Do(req)
```

```
tr := http.DefaultTransport
tr = NewTraceRoundTripper(tr, tracer)
tr = NewRetryRoundTripper(tr, []time.Duration{...})
tr = NewBackupRoundTripper(tr, hostnames)

client := http.Client{
    Transport: tr,
}
```

```go
func NewBackupRoundTripper(rt http.RoundTripper, upstreams []string) *BackupRoundTripper {
    return &RetryRoundTripper{
        rt:        rt,
        upstreams: upstreams,
    }
}

func (t *BackupRoundTripper) RoundTrip(req *http.Request) (*http.Response, error) {
  var resp *http.Response
  var err error
  prepareRequest(req) // <-- подготавливает req.Body к переиспользованию

  for n, upstreamURL := range t.upstreams {
    reqcpy := t.makeReq(req, upstreamURL) // <-- делает копию запроса
    if n != 0 {                           //     с нужным хостом
      resetRequest(&reqcpy)
    }
    closeResponse(resp)

    resp, err = t.rt.RoundTrip(&reqcpy)
    if !needUpstreamSwitch(resp, err) {
      break
    }
  }

  return resp, err
}
```

```go
type MyHandler struct {
    // все нужные объекты: конфиг, логер, соединение с базой и т.п.
}

// реализуем интерфейс `http.Handler`
func (h *MyHandler) ServeHTTP(w ResponseWriter, r *Request) {
    // эта функция будет обрабатывать входящие запросы
}

func main() {
    handler := &MyHandler{}

    server := &http.Server{
        Addr:           ":8080",
        Handler:        handler,
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
    }

    server.ListenAndServe()
}
```

HTTP сервер - обработчик

```go
func (h *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path == "/search" {
        args := r.URL.Query()
        query := args.Get("query")
        limit, err := strconv.Atoi(args.Get("limit"))
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            return
        }

        results, err := h.doSomeBusinessLogic(query, limit)
        if err != nil {
            w.WriteHeader(http.StatusInternalServerError)
            return
        }

        w.Header().Set("Content-Type", "application/json; charset=utf-8")
        w.WriteHeader(http.StatusOK)

        json.NewEncoder(w).Encode(results)
    }
}
```

HTTP сервер - функция как обработчик

```
type MyHandler struct {
}

func (h *MyHandler) ServeHTTP(w ResponseWriter, r *Request) {
}

...

    server := &http.Server{
        Handler: handler, // <--
    }
```

```
func SomeHttpHandler(w http.ResponseWriter, r *http.Request) {
}

...

    server := &http.Server{
        Handler: http.HandlerFunc(SomeHttpHandler), // <--
    }
}
```

```go
type MyHandler struct {}

func (h *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
  switch r.URL.Path {
    case "/search":
      h.Search(w, r)
    case "/add"
      h.AddItem(w, r)
    default:
      http.NotFound(w, r)
  }
}

func (h *MyHandler) Search(w ResponseWriter, r *Request) {
    // ...
}

func (h *MyHandler) AddItem(w ResponseWriter, r *Request) {
    // ...
}
```

```go
type MyHandler struct {}

func (h *MyHandler) Search(w ResponseWriter, r *Request) {
    // ...
}

func (h *MyHandler) AddItem(w ResponseWriter, r *Request) {
    // ...
}

func main() {
    handler := &MyHandler{}

    mux := http.NewServeMux()
    mux.HandleFunc("/search", handler.Search)
    mux.HandleFunc("/add", handler.AddItem)

    server := &http.Server{
        Addr:    ":8080",
        Handler: mux,
    }
    log.Fatal(server.ListenAndServe())
}
```

```go
func (s *server) adminOnly(h http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        if !currentUser(r).IsAdmin {
                http.NotFound(w, r)
                return
        }
        h(w, r)
    }
}

func main() {
    handler := &MyHandler{}

    mux := http.NewServeMux()
    mux.HandleFunc("/search", handler.Search)
    mux.HandleFunc("/add", adminOnly(handler.AddItem)) // <--
}
```

- Авторизация
- Rate Limit
- Логирование
- Трассировка
- Сжатие ответа

Пример Middleware - ограничение времени запроса

```
func (h *MyHandler) Search(w ResponseWriter, r *Request) {
    ctx := r.Context()

    results, err := DoBusinessLogicRequest(ctx, query, limit)
}

func withTimeout(h http.HandlerFunc, timeout time.Duration) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ctx := context.WithTimeout(r.Context(), timeout)
        r = r.WithContext(ctx)
        h(w, r)
    }
}

...

    mux := http.NewServeMux()
    mux.HandleFunc("/search", withTimeout(handler.Search, 5*time.Second))
```

Пример Middleware - авторизация

```
func (h *MyHandler) AddItem(w ResponseWriter, r *Request) {
    ctx := r.Context()
    user := ctx.Value("currentUser").(*MyUser)
    // ...
}

func authorize(h http.HandlerFunc, timeout time.Duration) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        user, err := DoAuthorizeUser(r)
        if err != nil {
            w.WriteHeader(http.StatusForbidden)
            return
        }
        ctx := context.WithValue(r.Context(), "currentUser", user)
        r = r.WithContext(ctx)
        h(w, r)
    }
}

...

    mux := http.NewServeMux()
    mux.HandleFunc("/add", authorize(handler.AddItem))
```

- https://github.com/gorilla/mux
- https://github.com/justinas/alice

Тестирование отдельного хэндлера

```
handler := func(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "<html><body>Hello World!</body></html>")
}

r := httptest.NewRequest("GET", "http://example.com/foo", nil)
w := httptest.NewRecorder()
handler(w, r)

resp := w.Result()
require.Equal(t, http.StatusOK, resp.StatusCode)

body, err := ioutil.ReadAll(resp.Body)
...
```

Тестирование целого сервера

```
myRouter := NewMyRouter(
    ...
)

ts := httptest.NewServer(myRouter.RootHandler())
defer ts.Close()

res, err := http.Get(ts.URL)
...
```

Тестирование http вызовов на другой сервис

```
serviceHandler := http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    ...
})

ts := httptest.NewServer(serviceHandler)
defer ts.Close()

doSomeBusinessLogic(serviceHandler)
...
```

`REST` - это архитектурный стиль разработки, при котором клиент и сервер обмениваются *документами*. По сути архитектура `REST` - это классические web страницы.

- `REST` хорошо подходит, если ваш сервис оперирует сложными иерархическими документами с множеством полей и мало возможных действий.
- `REST` плохо подходит, если в вашем сервисе много различных действий и выборок над одними и теми же сущностями.

`RPC` - это удаленный вызов процедур. Существует множество различных протоколов `RPC`: `DCOM`, `SOAP`, `JSON-RPC`, `gRPC`.

- `RPC` довольно универсальный подход

Запрос

```
GET /method?param1=value1&param2=value2 HTTP/1.1
Host: site.ru
```

Ответ

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 100500

{
    "status": "ok",
    "items": [
        ...
    ]
}
```

Запрос

```
POST /api HTTP/1.1
Host: site.ru
Content-Type: application/json
Content-Length: 100500

{"method": "echo", "params": ["Hello JSON-RPC"], "id":1}
```

Ответ

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 100500

{"result": "Hello JSON-RPC", "error": null, "id":1}
```

OpenAPI, изначально известное как Swagger это DSL (Domain Specific Language, специализированный язык) для описания REST API. Спецификации Open API могут быть описанны в виде JSON или YAML документов.

Редактировать Swagger спецификацию: https://editor.swagger.io

Установить утилиту для Go: https://github.com/go-swagger/go-swagger

Дополнительные материалы

- [en] Классный урок про использование контекста
- [ru] Доклад про использование GraphQL в Go
- [en] Про дизайн клиента и middleware
- [en] Про ненужность сторонних роутеров

Заполните пожалуйста опрос

Ссылка в чате.

Работа с gRPC

21 июня, вторник

Спасибо за внимание!