

# ОСНОВЫ WEB-РАЗРАБОТКИ

## Лекция 7. Организация межсетевого взаимодействия

Курс читают:

Шульман В.Д.

@ShtuzerVD

Пелевина Т.В.

@anivelat

Шабанов В.В.

@ZeroHug

# ПЛАН ЛЕКЦИИ

2

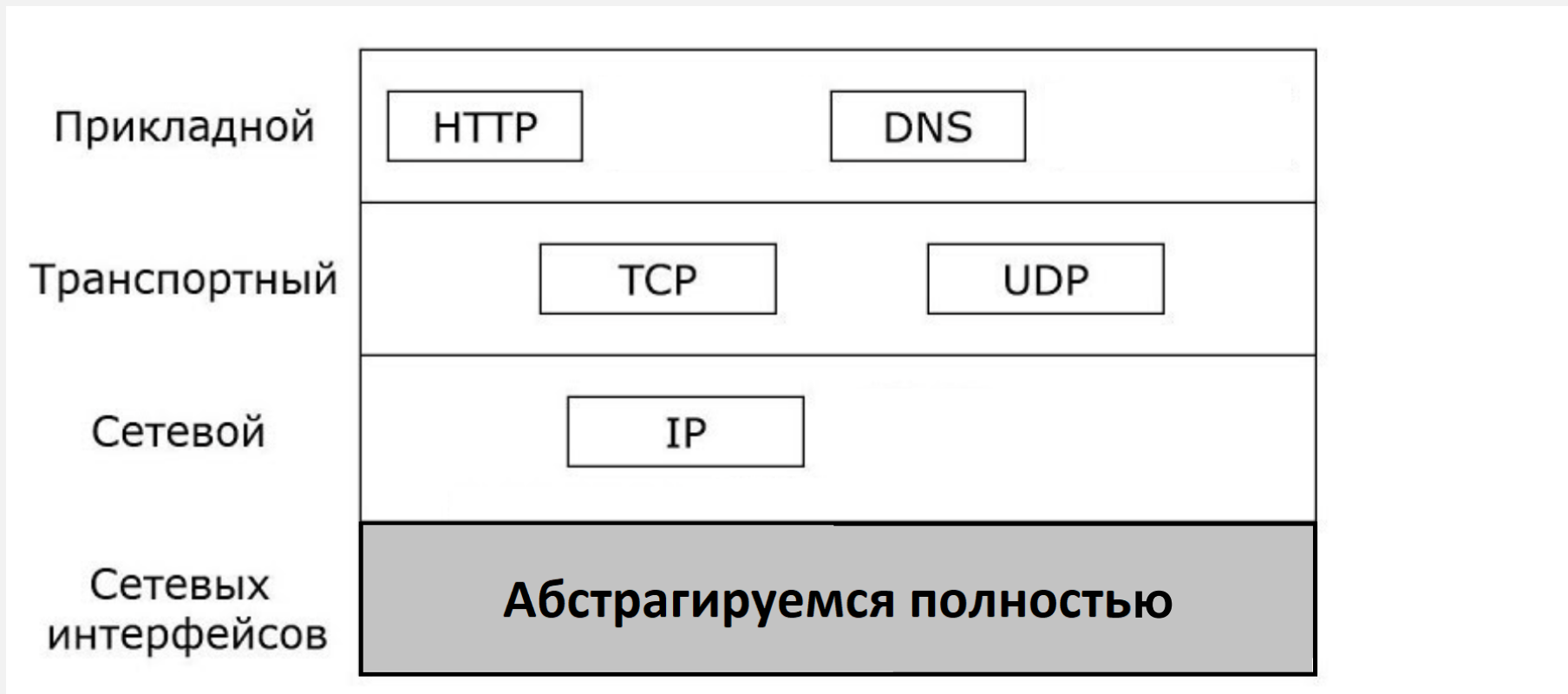
- IP / TCP / HTTP
- JSON
- HTTP-сервер на Golang

15.10.2024

# СТЕК ПРОТОКОЛОВ

3

Существует понятие стека протоколов TCP/IP



15.10.2024

**IP-адрес** — уникальный сетевой адрес узла в компьютерной сети, построенной на основе стека протоколов **TCP/IP**

Ваш IP адрес:

**92.42.213.15**

Обычно состоит из 4 октетов. Можно зайти на <https://2ip.ru/> и узнать свой ip адрес

# IP-АДРЕС

5

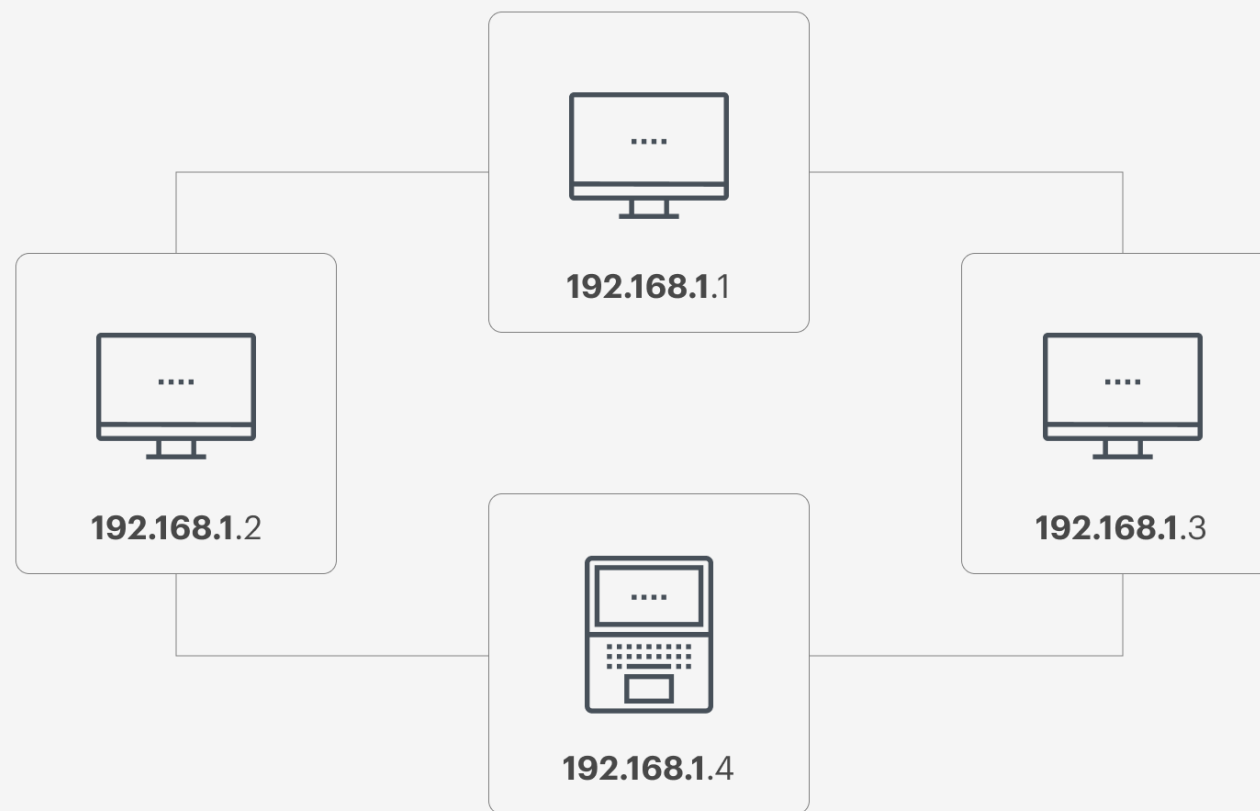
Номер сети	Номер хоста
192.168.1	.34

15.10.2024

# IP-АДРЕС

6

Несколько устройств, объединенных в одну сеть

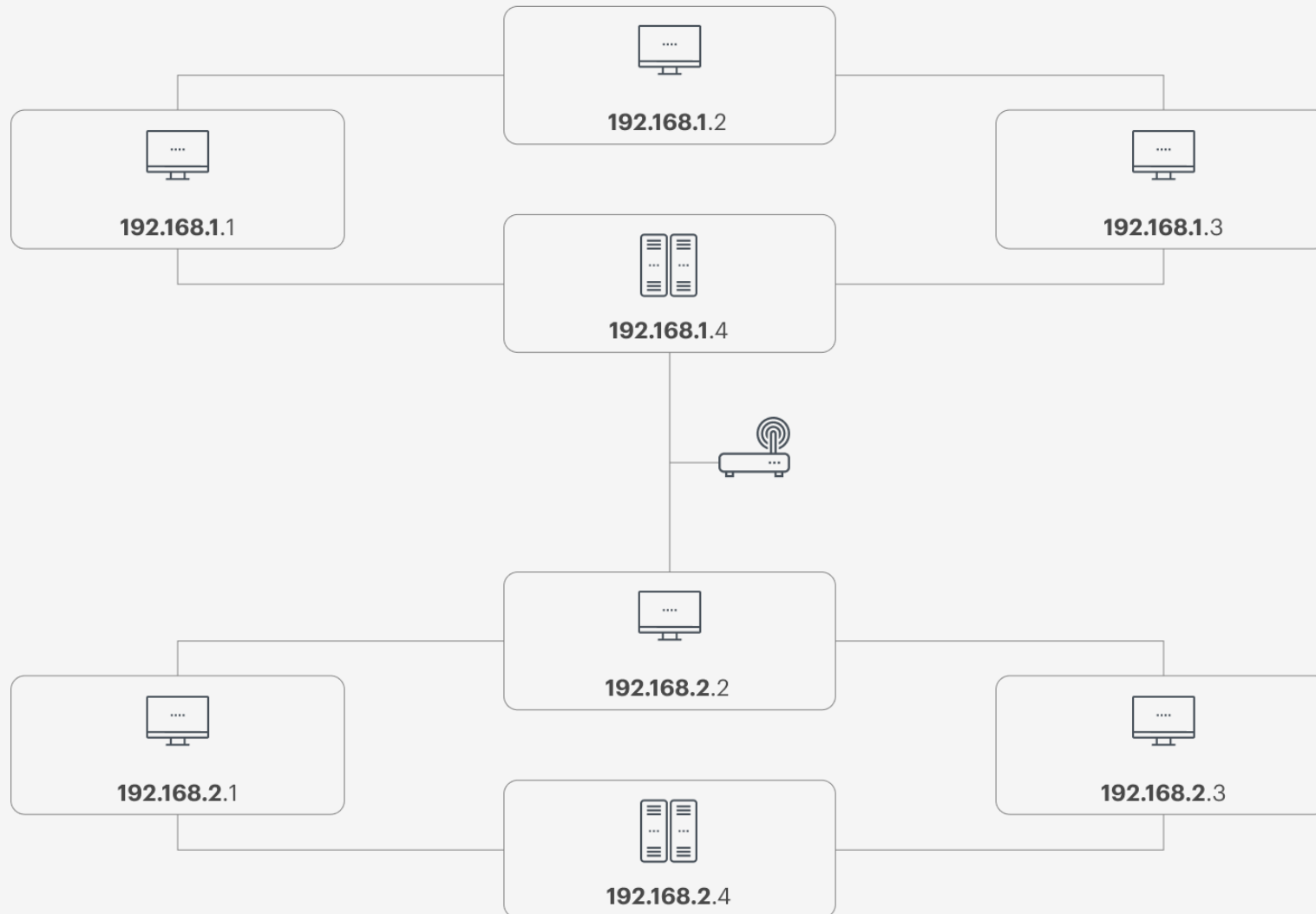


15.10.2024

# IP-АДРЕС

7

Несколько устройств, объединенных в две сети



15.10.2024

## 255.255.0.0

Пример маски подсети



# IP-АДРЕС

9

Давайте применим к IP-адресу 192.168.1.34 маску подсети 255.255.255.0:

1	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

И

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

=

1	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(или 192.168.1.0 в десятичной системе счисления)

15.10.2024

# IP-АДРЕС

10

Давайте применим к IP-адресу 192.168.1.34 маску подсети 255.255.255.0:

1	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

И

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

=

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

У нас получился адрес 0.0.0.34. Это и есть номер хоста.

15.10.2024

# IP-АДРЕС

11

Номер сети	Номер хоста
192.168.1	.34

15.10.2024

Высчитаем сколько устройств (в IP адресах — узлов) может быть в сети, где у одного компьютера адрес 172.16.13.98 /24.

172.16.13.0 – адрес сети

172.16.13.1 – адрес первого устройства в сети

172.16.13.254 – адрес последнего устройства в сети

172.16.13.255 – широковещательный IP адрес

172.16.14.0 – адрес следующей сети

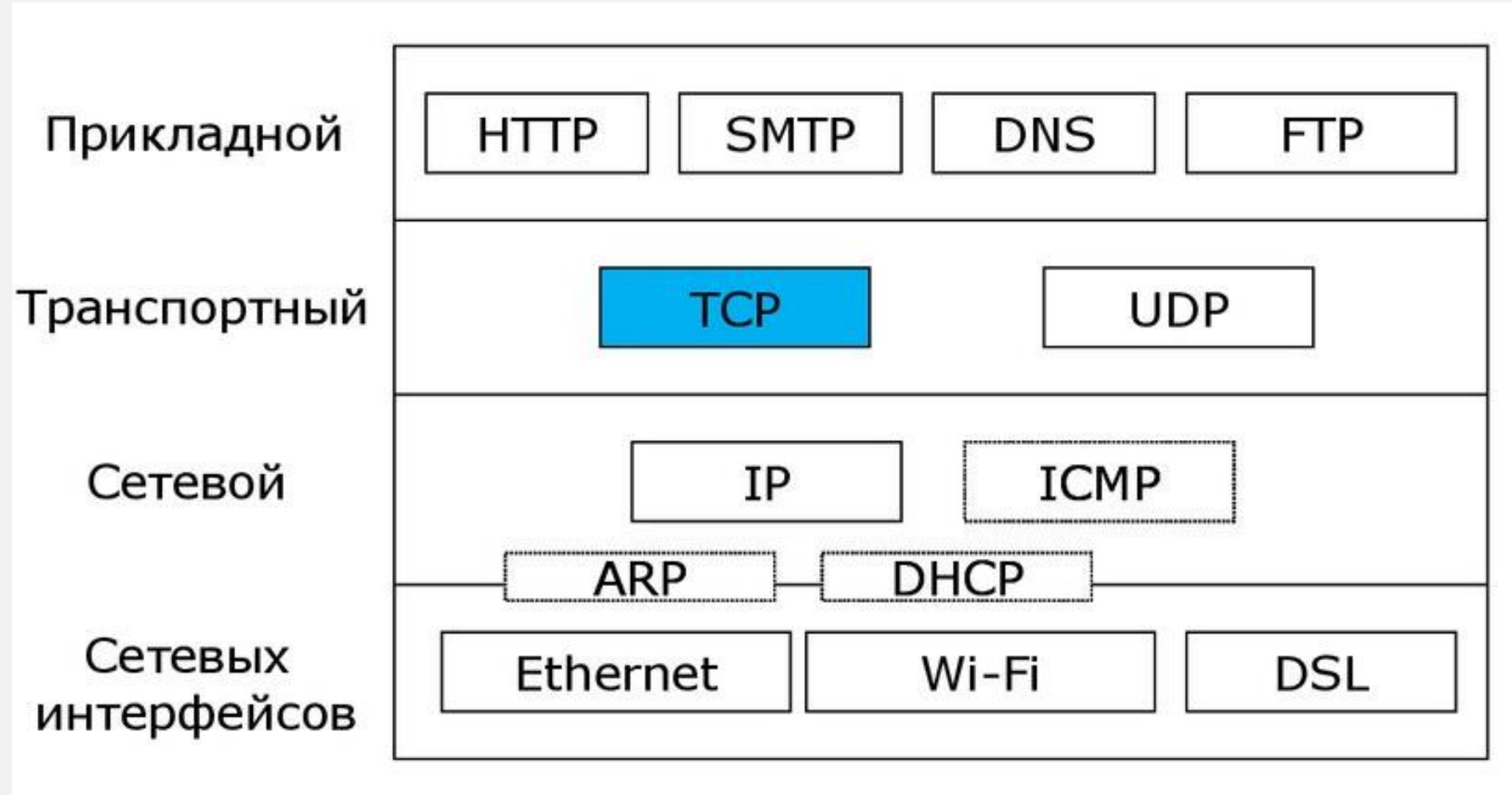
**Адрес 127.0.0.0 – 127.255.255.255** (loopback – петля на себя).

Данная сеть чаще всего используется для локального тестирования сервера

Очень часто мы будем запускать сервер по адресу **127.0.0.1 (localhost)**

# ПРОТОКОЛ TCP

14



15.10.2024

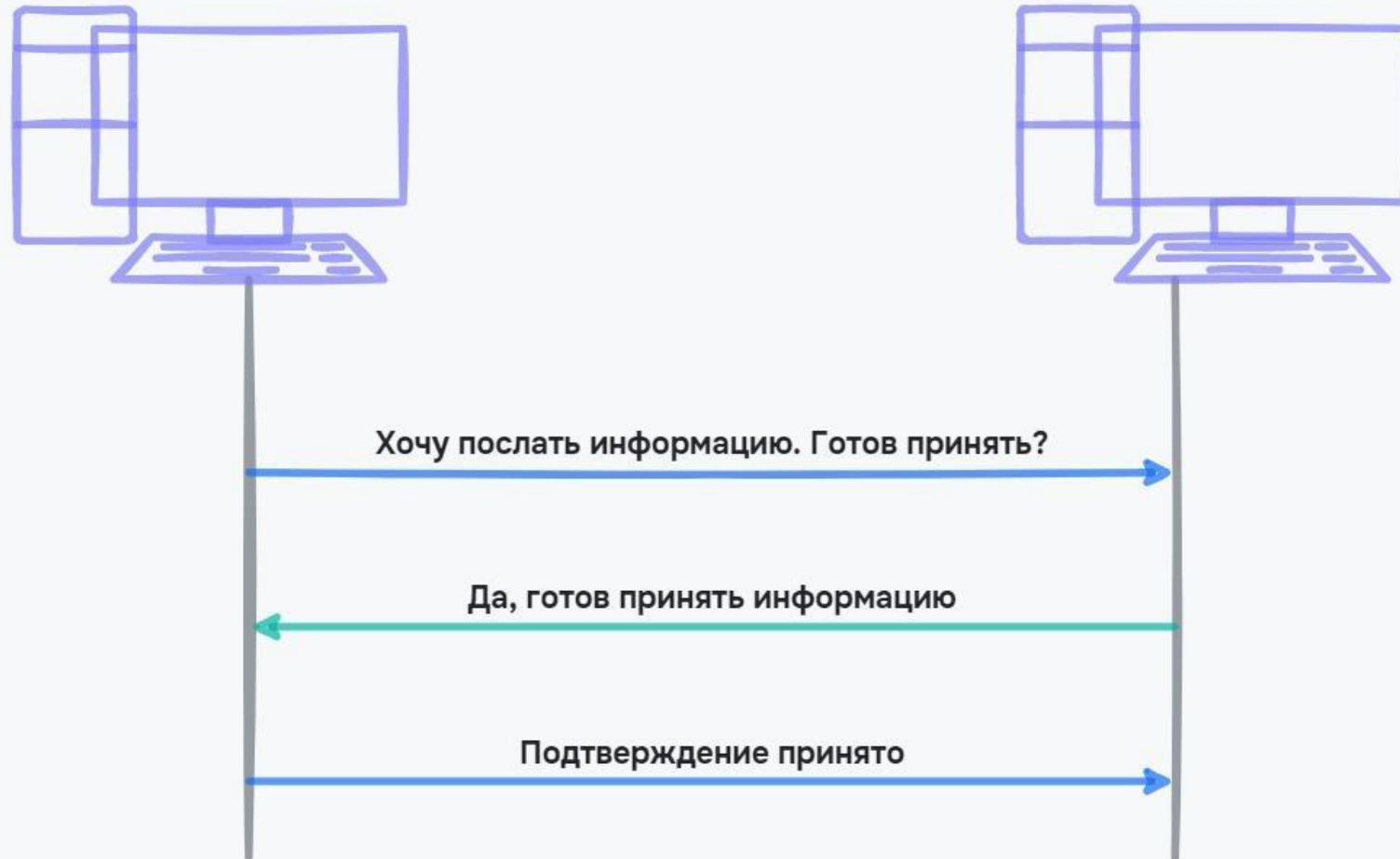
**ТСР** обеспечивают надежную передачу данных между клиентом и сервером.

В роли клиента и сервера выступают не элементы компьютерной сети, а программы, на них расположенные.

Порт программы дописывается после ip-адреса через двоеточие, например, 127.0.0.1:8080

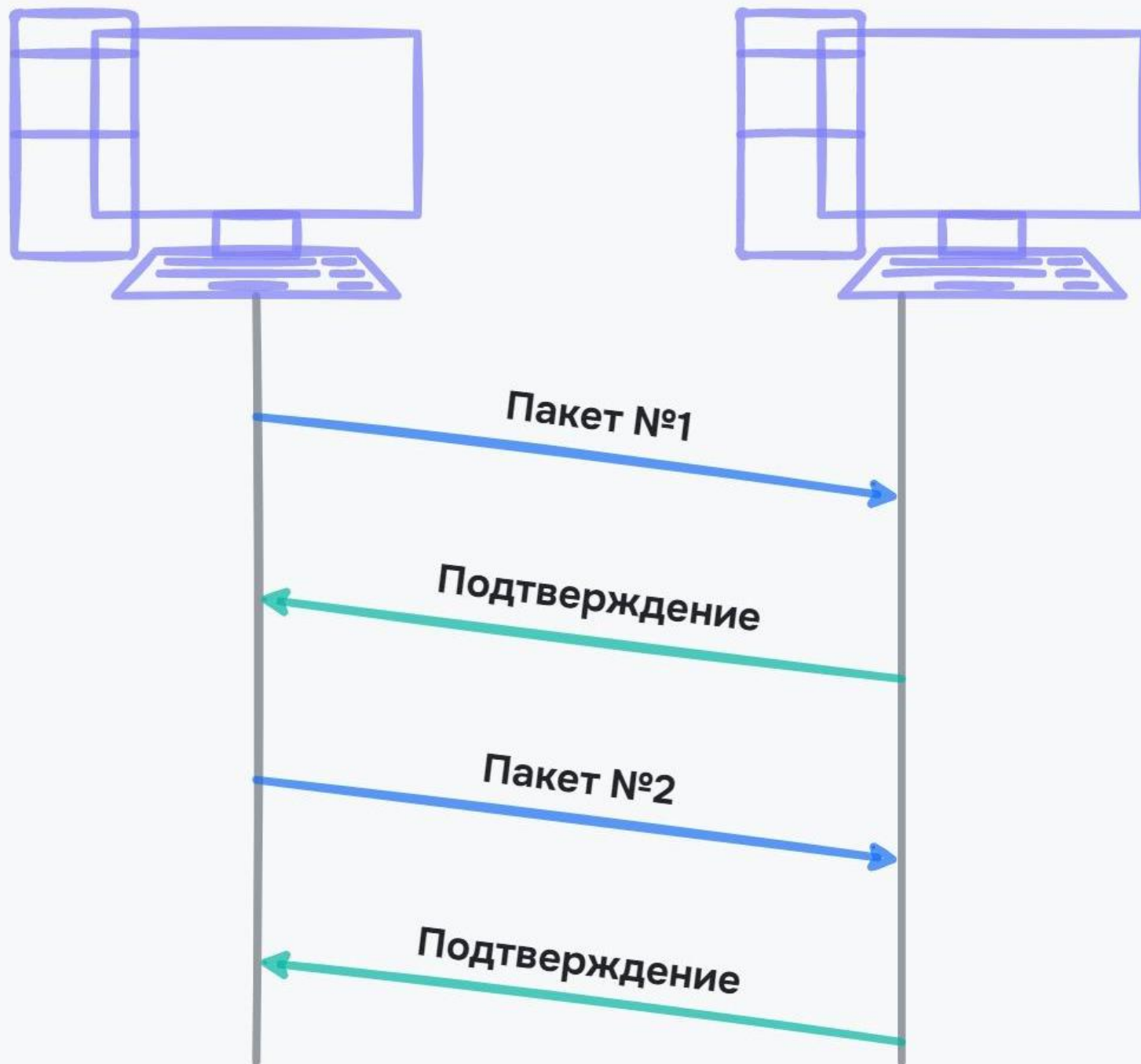
# TCP

## Handshake (Рукопожатие)

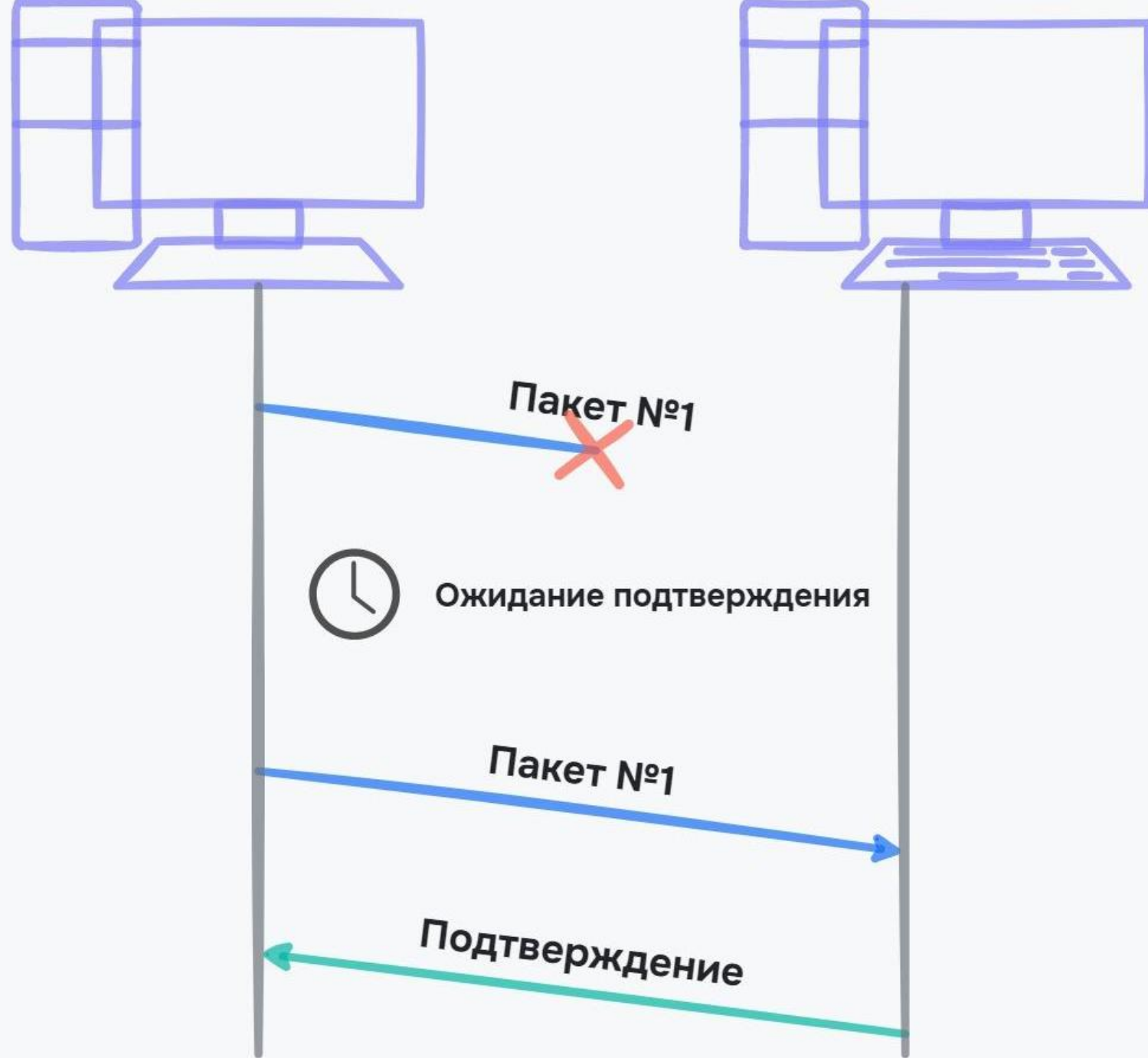




# TCP

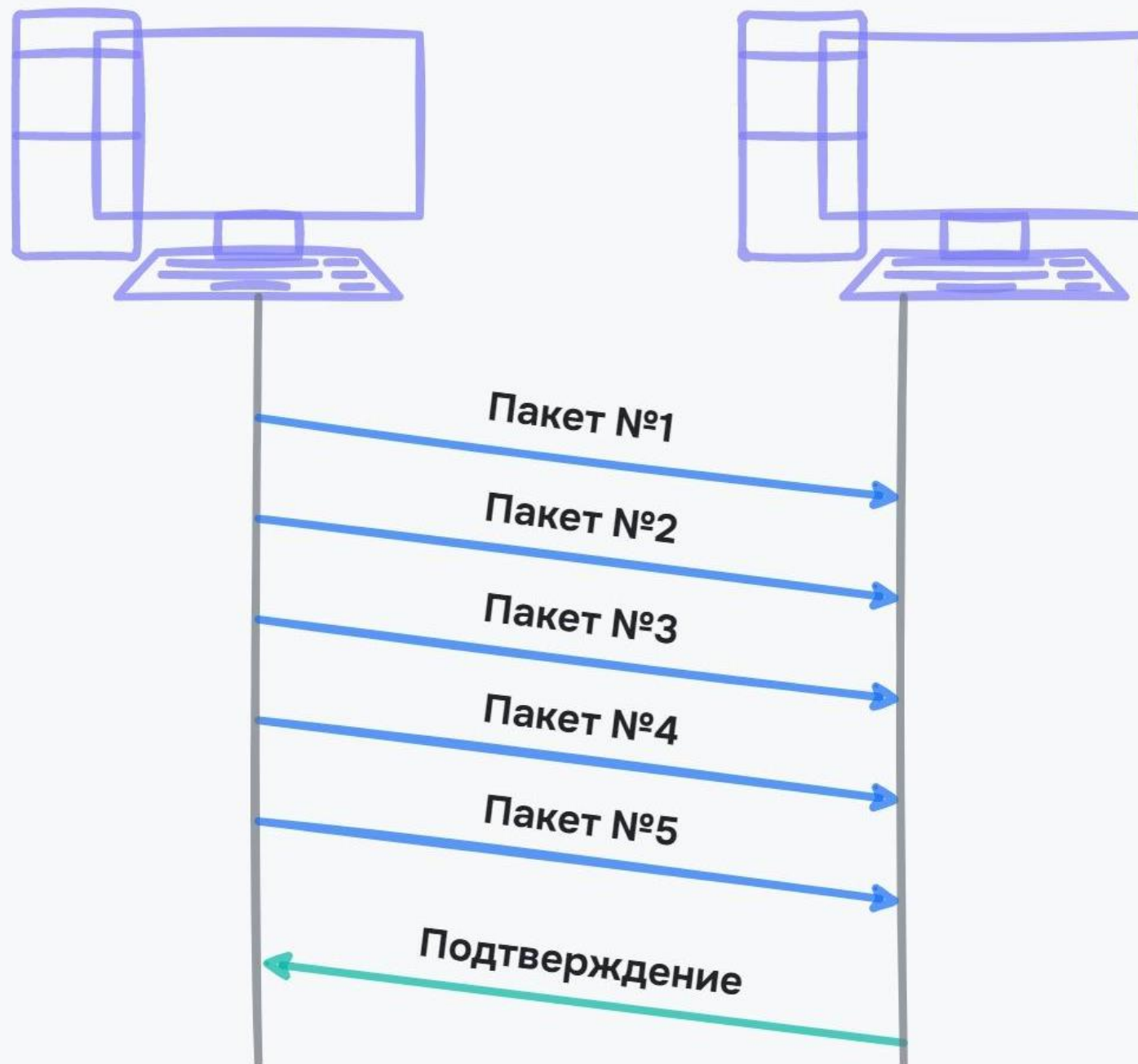


# TCP

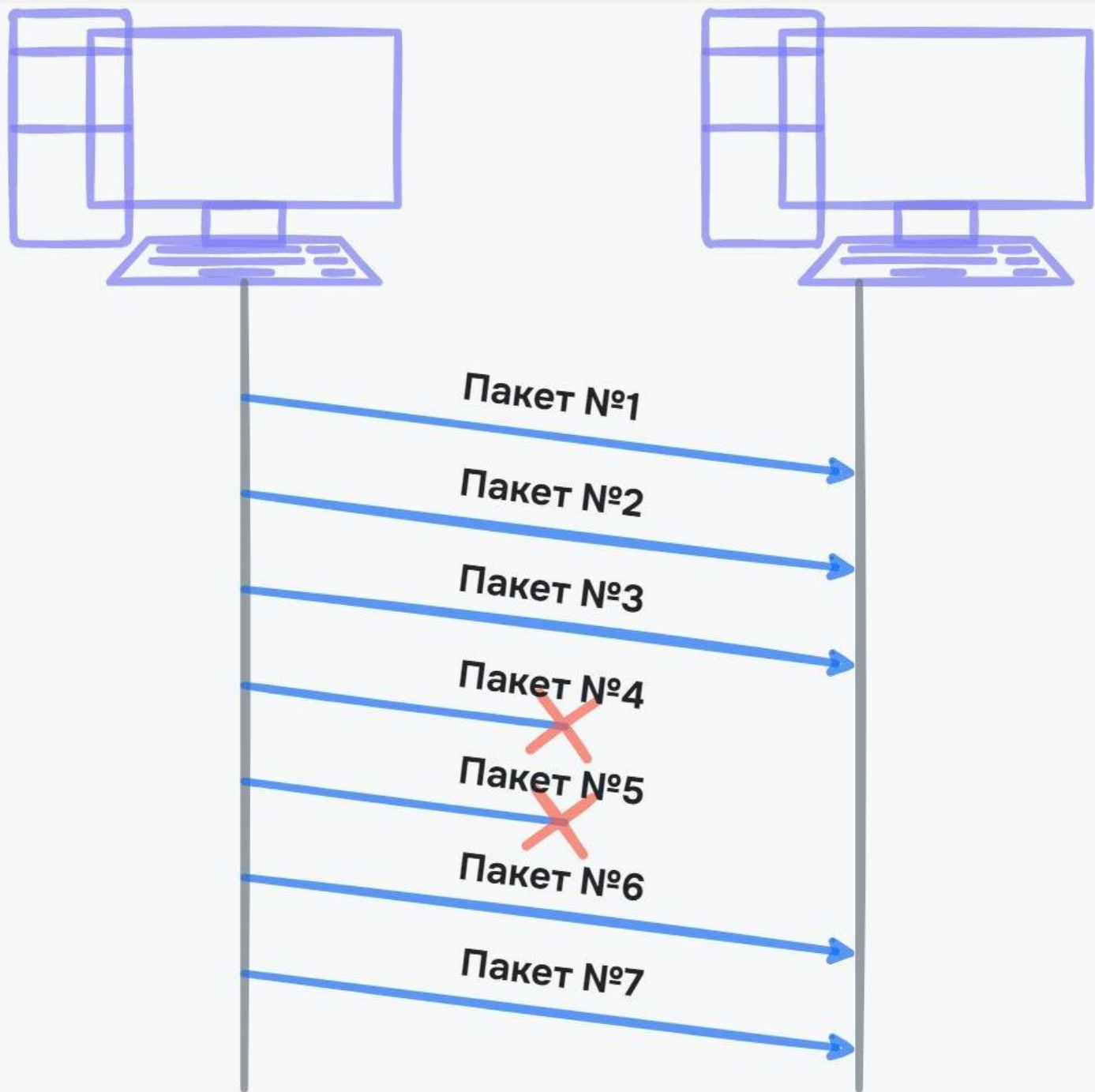


# TCP

## Sliding Window



# UDP



## TCP

Подходит для передачи важной, требующей целостности информации

Долгое время передачи

Гарантия передачи пакетов

## UDP

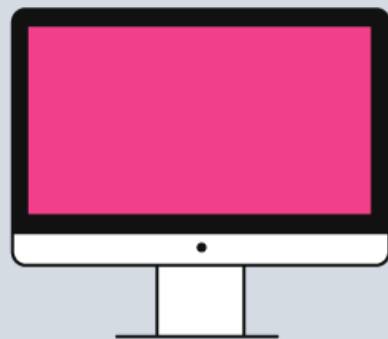
Подходит для передачи информации, устойчивой к потерям

Быстрая передача пакетов

Отсутствие гарантии передачи пакетов

**HTTP (HyperText Transfer Protocol) — это протокол прикладного уровня, который используется для передачи данных в сети Интернет.**

Он определяет правила и формат обмена данными между клиентом (например, веб-браузером) и сервером.

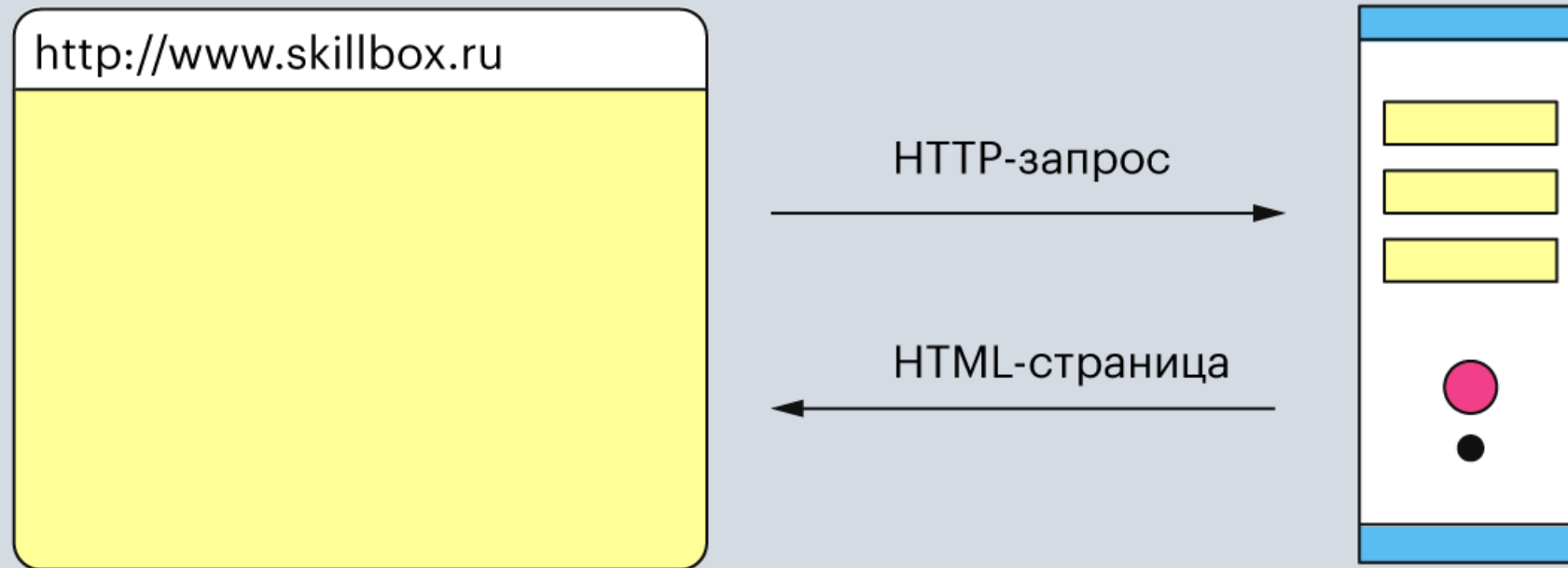


HTTP-запрос



HTTP-ответ

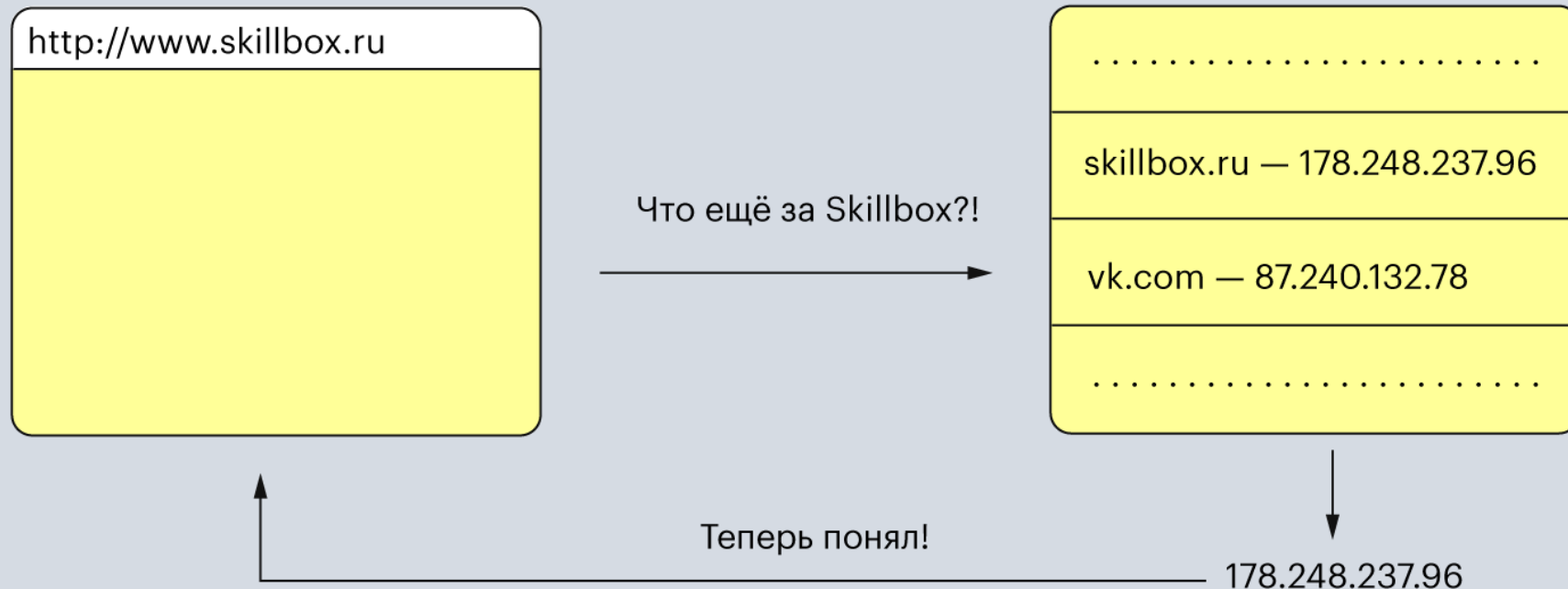






Для пользователей URL-адрес — это набор понятных слов: Skillbox, Yandex, Google. Но для компьютера эти понятные нам слова — набор непонятных символов.

Поэтому браузер отправляет введенные вами слова в DNS, преобразователь URL-адресов в IP-адреса. DNS расшифровывается как «доменная система имён» (Domain Name System), и его можно представить как огромную таблицу со всеми зарегистрированными именами для сайтов и их IP-адресами.



Метод    URI    Версия HTTP

↓       ↓       ↓

**GET / HTTP/1.1**

**Host: www.skillbox.ru**

↑

Адрес хоста

The diagram illustrates the components of an HTTP request. At the top, three labels in Russian: 'Метод' (Method), 'URI', and 'Версия HTTP' (HTTP Version) have arrows pointing down to the first line of the request: 'GET / HTTP/1.1'. This line is enclosed in a pink bracket. Below it, the second line is 'Host: www.skillbox.ru', also enclosed in a pink bracket. An arrow points up from the label 'Адрес хоста' (Host address) to the 'www.skillbox.ru' part of the host field.

А где порт ?

**HTTP/1.1 200 OK**  
**Content-Type: text/html; charset=UTF-8**  
**Content-Length: 208**

```
<html>  
<head><title>Заголовок HTTP-запроса</title></head>  
<body>Hello, World!</body>  
</html>
```

Метаданные

**HTTP/1.1 200 OK**

Статус ответа

Content-Type: text/html; charset=UTF-8  
Content-Length: 208

Заголовки

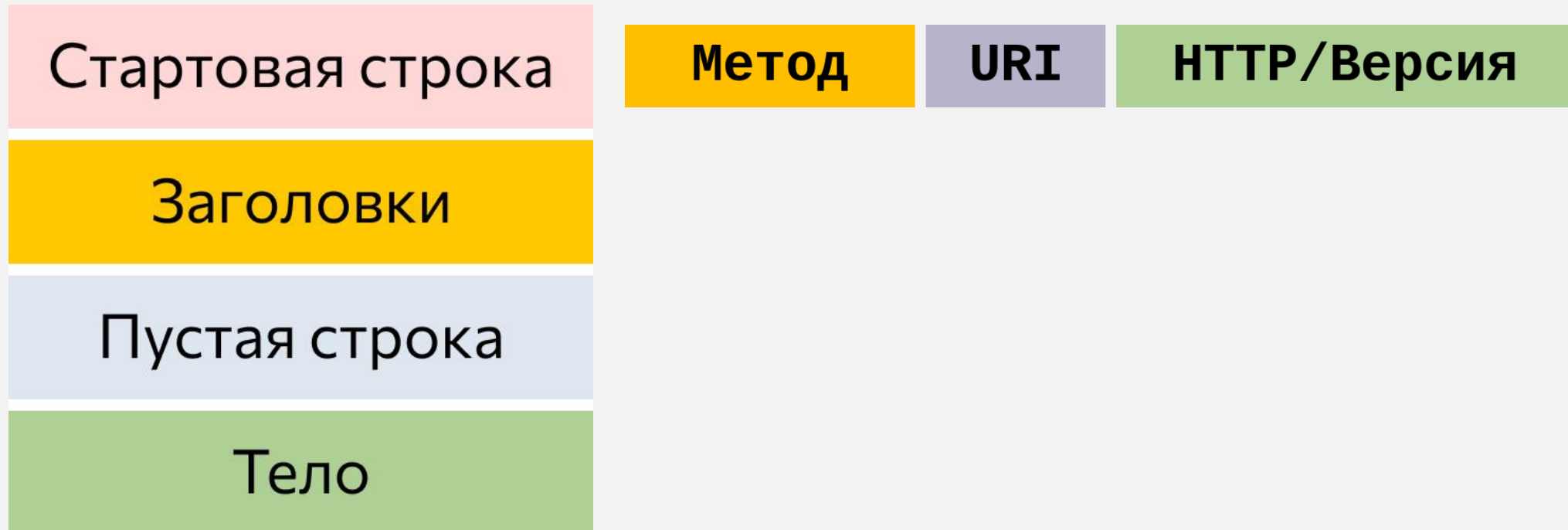
Тело

<html>  
<head><title>Заголовок HTTP-запроса</title></head>  
<body>Hello, World!</body>  
</html>

Тело ответа

А теперь чуть подробнее...

## Структура запроса



## Структура запроса

Стартовая строка

Заголовки

Пустая строка

Тело

### HTTP Request

Method      URL      Protocol Version

↓            ↓            ↓

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html, */*
Accept-Language: en-us
Accept-Charset: ISO-8859-1,utf-8
Connection: keep-alive
blank line
```

Headers {

Body (optional) {

## Методы HTTP запроса

### GET

Позволяет запросить некоторый конкретный ресурс. Дополнительные данные могут быть переданы через строку запроса (Query String) в составе URL (например ?param=value). О составляющих URL мы поговорим чуть позже.

### POST

Позволяет отправить данные на сервер. Поддерживает отправку различных типов файлов, среди которых текст, PDF-документы и другие типы данных в двоичном виде. Обычно метод POST используется при отправке информации (например, заполненной формы логина) и загрузке данных на веб-сайт, таких как изображения и документы.



## Методы HTTP запроса

### PUT

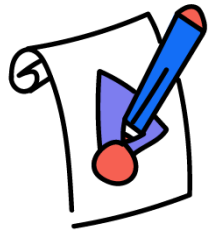
Используется для создания (размещения) новых ресурсов на сервере. Если на сервере данный метод разрешен без надлежащего контроля, то это может привести к серьезным проблемам безопасности.

### DELETE

Позволяет удалить существующие ресурсы на сервере. Если использование данного метода настроено некорректно, то это может привести к атаке типа «Отказ в обслуживании» (Denial of Service, DoS) из-за удаления критически важных файлов сервера.

## Методы HTTP запроса

POST



CREATE

**C**

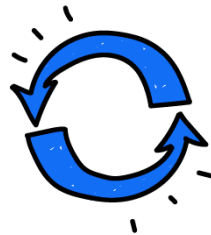
GET



READ

**R**

PUT



UPDATE

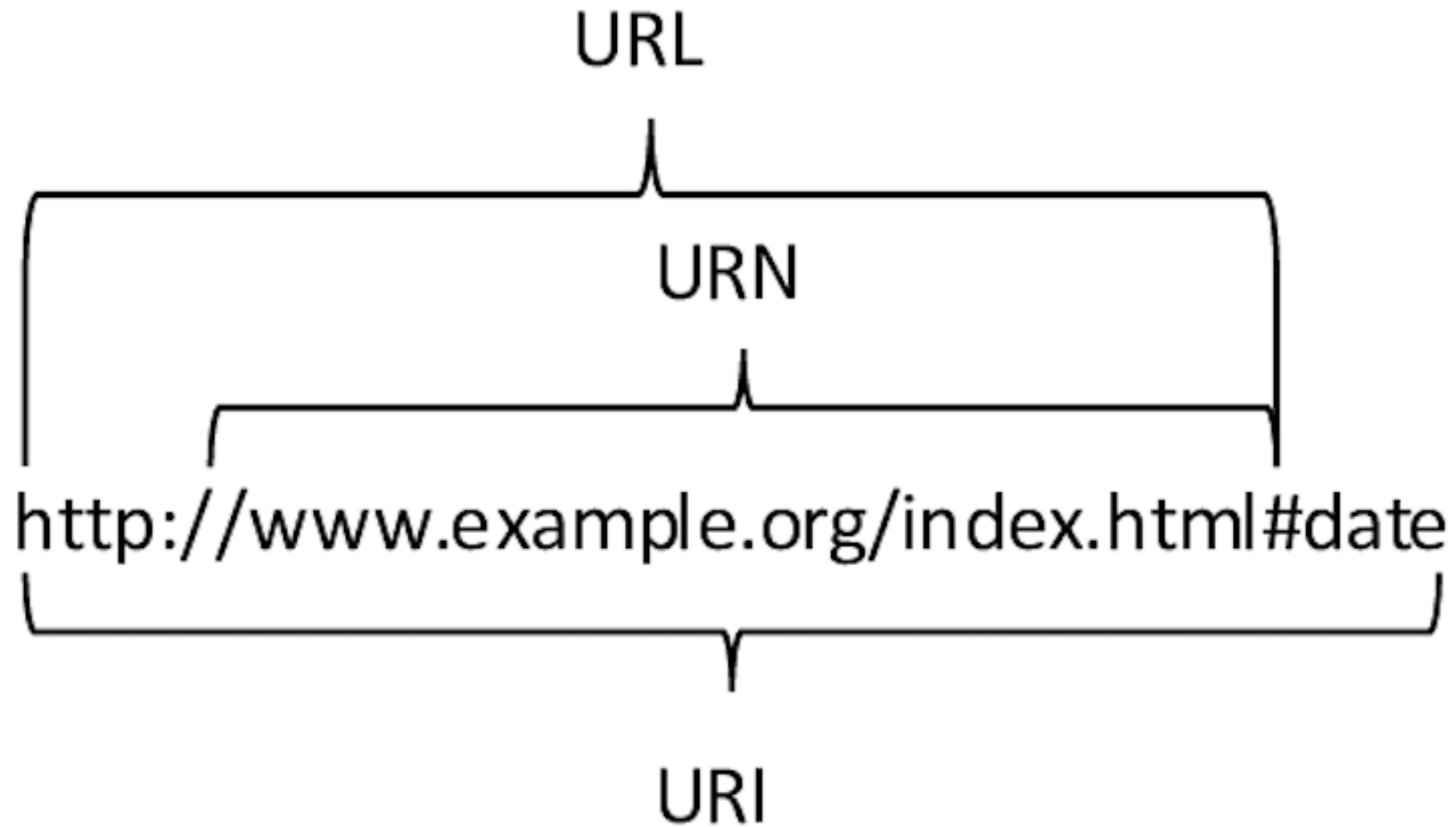
**U**

DELETE



DELETE

**D**





Код состояния ответа HTTP показывает, был ли успешно выполнен определённый [HTTP](#) запрос.

Ответы сгруппированы в 5 классов:

1. [Информационные ответы](#) ( 100 – 199 )
2. [Успешные ответы](#) ( 200 – 299 )
3. [Сообщения о перенаправлении](#) ( 300 – 399 )
4. [Ошибки клиента](#) ( 400 – 499 )
5. [Ошибки сервера](#) ( 500 – 599 )

Коды состояния определены в [RFC 9110](#) .

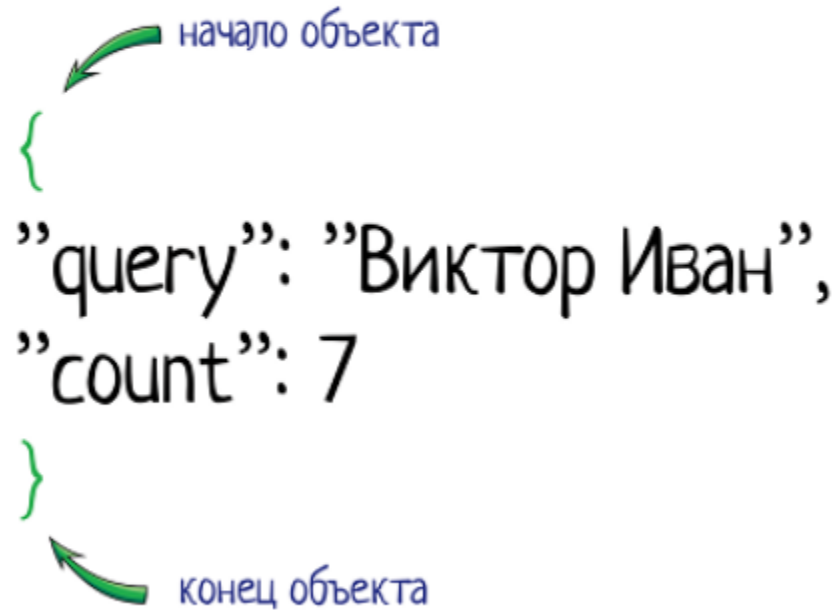
**JSON (JavaScript Object Notation)** — текстовый формат для хранения и обмена структурированными данными. Он основан на синтаксисе объектов в JavaScript, но не зависит от него.

**Данные в JSON** представляются в виде пар «ключ — значение». Ключи — всегда строки, а значения могут быть представлены различными типами: числовыми, строковыми, логическими.

**Формат широко используется** для передачи данных в приложениях между серверами и клиентами, а также для их хранения и обмена ими.

```
{  
  "query": "Виктор Иван",  
  "count": 7  
}
```

## JSON-объект



The diagram shows a JSON object within a light blue rounded rectangle. The object is represented by a green opening curly brace at the top left, followed by two lines of text: `"query": "Виктор Иван",` and `"count": 7`, and a green closing curly brace at the bottom left. A green curved arrow points from the text "начало объекта" (start of the object) to the opening brace. Another green curved arrow points from the text "конец объекта" (end of the object) to the closing brace.

```
{  
  "query": "Виктор Иван",  
  "count": 7  
}
```

начало объекта

конец объекта




Ключ (название параметра, свойства объекта)

```
{  
  "query": "Виктор Иван",  
  "count": 7  
}
```

Значение

JSON-объект

```
{  
  "query": "Виктор Иван",  
  "count": 7  
}
```



Пары ключ-значение разделены запятыми



```
{  
  "query": "Виктор Иван",  
  "count": 7  
}
```

строка

число

The diagram illustrates JSON syntax rules. It shows a JSON object with two properties: 'query' and 'count'. The value of 'query' is a string 'Виктор Иван', and the value of 'count' is the number 7. A blue arrow points from the Russian word 'строка' (string) to the string value, and another blue arrow points from the Russian word 'число' (number) to the number value. The quotes around the string value are highlighted in orange.

Строки берем в кавычки, числа нет

## JSON-массив

[ "MALE", "FEMALE" ]



Элементы массива

Элементы массива разделены запятыми

[ "MALE", "FEMALE" ]



```
{  
  "query": "Виктор Иван",  
  "count": 7,  
  "parts": ["NAME", "SURNAME"]  
}
```

```
{  
  "value": "Виктор Иванович",  
  "unrestricted_value": "Виктор Иванович",  
  "data": {  
    "surname": null,  
    "name": "Виктор",  
    "patronymic": "Иванович",  
    "gender": "MALE"  
  }  
}
```

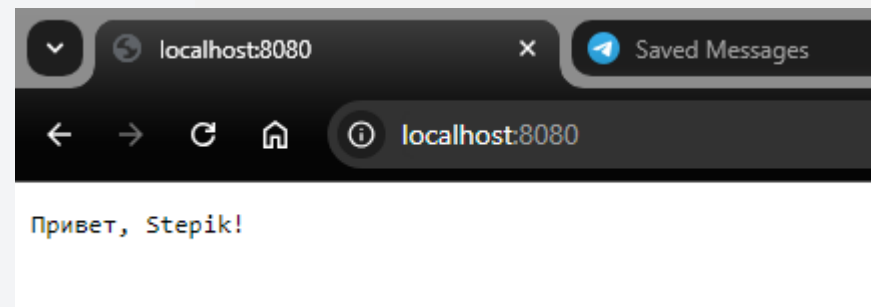
```
package main

import (
    "fmt"
    "net/http"
)

// Обработчик HTTP-запросов
func handler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Привет, Stepik!"))
}

func main() {
    // Регистрируем обработчик для пути "/"
    http.HandleFunc("/", handler)

    // Запускаем веб-сервер на порту 8080
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        fmt.Println("Ошибка запуска сервера:", err)
    }
}
```





```
8 // Обработчик HTTP-запросов
9 func handler(w http.ResponseWriter, r *http.Request) {
10     fmt.Println(r.Method) // Тип метода
11     fmt.Println(r.URL)    // запрашиваемый URL
12     fmt.Println(r.Proto)  // версия протокола
13     w.Write([]byte("Привет, Stepik!"))
14 }
15
16 func main() {
17     // Регистрируем обработчик для пути "/"
18     http.HandleFunc("/", handler)
19
20     // Запускаем веб-сервер на порту 8080
21     err := http.ListenAndServe(":8080", nil)
22     if err != nil {
23         fmt.Println("Ошибка запуска сервера:", err)
24     }
25 }
```

Вывод:

```
GET
/
HTTP/1.1
```

```
8 // Обработчик HTTP-запросов
9 func handler(w http.ResponseWriter, r *http.Request) {
10     fmt.Println("RawQuery: ", r.URL.String())
11     fmt.Println("Name: ", r.URL.Query().Get("name"))
12     fmt.Println("IsExist: ", r.URL.Query().Has("name"))
13     w.Write([]byte("Привет, Stepik!"))
14 }
15
16 func main() {
17     // Регистрируем обработчик для пути "/"
18     http.HandleFunc("/", handler)
19
20     // Запускаем веб-сервер на порту 8080
21     err := http.ListenAndServe(":8080", nil)
22     if err != nil {
23         fmt.Println("Ошибка запуска сервера:", err)
24     }
25 }
```

Вывод:

```
RawQuery:  /?name=Semyon
Name:  Semyon
IsExist:  true
```

```
16 // Обработчик HTTP-запросов
17 func handler(w http.ResponseWriter, r *http.Request) {
18     // проверяем что метод POST
19     if r.Method == "POST" {
20         // читаем входящее тело запроса
21         bytesBody, err := io.ReadAll(r.Body)
22         if err != nil {
23             log.Println(err)
24             w.Write([]byte("Плохое тело запроса"))
25             return
26         }
27         // печатаем тело запроса как строку
28         fmt.Println(string(bytesBody))
29         // отвечаем клиенту, что все хорошо
30         w.Write([]byte("OK!"))
31         return
32     }
33     w.Write([]byte("Разрешен только метод POST!"))
34 }
```

```
PS C:\Users\Vitalian> go run main.go
{"key": "value"}
```

# CEPBEP HA GOLANG

52

The screenshot shows a REST client interface with a dark theme. At the top, a POST request is configured for the URL `http://127.0.0.1:8080/`. The request body is a JSON object: `{"key": "value"}`. The response status is 200 OK, with a time of 7 ms and a size of 119 B. The response body is `OK!`.

**POST** `http://127.0.0.1:8080/` **Send**

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON**

1 `{"key": "value"}`

**Body** Cookies Headers (3) Test Results **Status: 200 OK** **Time: 7 ms** **Size: 119 B** **Save Response**

**Pretty** Raw Preview Visualize **Text**

1 `OK!`

15.10.2024

```
8 ▾ func handler(w http.ResponseWriter, r *http.Request) {  
9     w.Write([]byte("Привет!"))  
10    w.WriteHeader(200)  
11 }  
12  
13 ▾ func handler2(w http.ResponseWriter, r *http.Request) {  
14 ▾     if r.Method == "PUT" {  
15         w.WriteHeader(http.StatusMethodNotAllowed) // вернем 405  
16         return  
17     }  
18     w.Write([]byte("Привет!"))  
19     w.WriteHeader(http.StatusOK) // одно и тоже что и 200  
20 }
```



```
const (  
    StatusContinue          = 100 // RFC 9110, 15.2.1  
    StatusSwitchingProtocols = 101 // RFC 9110, 15.2.2  
    StatusProcessing         = 102 // RFC 2518, 10.1  
    StatusEarlyHints         = 103 // RFC 8297  
  
    StatusOK                = 200 // RFC 9110, 15.3.1  
    StatusCreated           = 201 // RFC 9110, 15.3.2  
    StatusAccepted          = 202 // RFC 9110, 15.3.3  
    StatusNonAuthoritativeInfo = 203 // RFC 9110, 15.3.4  
    StatusNoContent         = 204 // RFC 9110, 15.3.5  
    StatusResetContent       = 205 // RFC 9110, 15.3.6  
    StatusPartialContent     = 206 // RFC 9110, 15.3.7  
    StatusMultiStatus        = 207 // RFC 4918, 11.1  
    StatusAlreadyReported    = 208 // RFC 5842, 7.1  
    StatusIMUsed             = 226 // RFC 3229, 10.4.1
```

```
StatusMultipleChoices    = 300 // RFC 9110, 15.4.1  
StatusMovedPermanently   = 301 // RFC 9110, 15.4.2  
StatusFound              = 302 // RFC 9110, 15.4.3  
StatusSeeOther           = 303 // RFC 9110, 15.4.4  
StatusNotModified        = 304 // RFC 9110, 15.4.5  
StatusUseProxy           = 305 // RFC 9110, 15.4.6  
-                         = 306 // RFC 9110, 15.4.7 (Unused)  
StatusTemporaryRedirect  = 307 // RFC 9110, 15.4.8  
StatusPermanentRedirect = 308 // RFC 9110, 15.4.9  
  
StatusBadRequest          = 400 // RFC 9110, 15.5.1  
StatusUnauthorized        = 401 // RFC 9110, 15.5.2  
StatusPaymentRequired     = 402 // RFC 9110, 15.5.3  
StatusForbidden           = 403 // RFC 9110, 15.5.4  
StatusNotFound            = 404 // RFC 9110, 15.5.5  
StatusMethodNotAllowed    = 405 // RFC 9110, 15.5.6  
StatusNotAcceptable       = 406 // RFC 9110, 15.5.7  
StatusProxyAuthRequired   = 407 // RFC 9110, 15.5.8  
StatusRequestTimeout      = 408 // RFC 9110, 15.5.9  
StatusConflict            = 409 // RFC 9110, 15.5.10  
StatusGone                = 410 // RFC 9110, 15.5.11  
StatusLengthRequired      = 411 // RFC 9110, 15.5.12  
StatusPreconditionFailed   = 412 // RFC 9110, 15.5.13  
StatusRequestEntityTooLarge = 413 // RFC 9110, 15.5.14
```

```
17 // Обработчик HTTP-запроса
18 func Handler(w http.ResponseWriter, r *http.Request) {
19     var input Input
20
21     decoder := json.NewDecoder(r.Body)
22     err := decoder.Decode(&input)
23     if err != nil {
24         w.WriteHeader(400)
25         w.Write([]byte(err.Error()))
26         return
27     }
28
29     var output Output
30     output.Result = "Hello, " + input.Name
31
32     w.Header().Set("Content-Type", "application/json")
33     w.WriteHeader(200)
34     respBytes, _ := json.Marshal(output)
35     w.Write(respBytes)
36 }
```

```
9 type Input struct {
10     Name string `json:"name"`
11 }
12
13 type Output struct {
14     Result string `json:"result"`
15 }
16
```

# CEPBEP HA GOLANG

56

The screenshot displays a REST client interface with a dark theme. At the top, a POST request is configured for the URL `http://127.0.0.1:8080/`. The 'Body' tab is selected, showing a JSON payload: `{ "name": "Vlad" }`. The request is sent, resulting in a 200 OK status. The response body is shown in the 'Body' tab, displaying a JSON object: `{ "result": "Hello, Vlad" }`. The interface includes tabs for Params, Authorization, Headers (8), Body, Pre-request Script, Tests, and Settings. The response status is 200 OK, with a time of 6 ms and a size of 132 B. The response is saved.

**POST** `http://127.0.0.1:8080/` **Send**

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON**

1 `{ "name": "Vlad" }`

**Body** Cookies Headers (3) Test Results **Status: 200 OK** Time: 6 ms Size: 132 B **Save Response**

**Pretty** Raw Preview Visualize **JSON**

1 `{`  
2  `"result": "Hello, Vlad"`  
3 `}`

15.10.2024



- [https://ru.hexlet.io/courses/internet-fundamentals/lessons/tcp-ip/theory\\_unit](https://ru.hexlet.io/courses/internet-fundamentals/lessons/tcp-ip/theory_unit)
- <https://habr.com/ru/companies/ruvds/articles/759988/>
- <https://habr.com/ru/articles/350878/>
- <https://skillbox.ru/media/code/что-такое-http-i-zachem-on-nuzhen/>
- <https://habr.com/ru/articles/554274/>

СПАСИБО ЗА ВНИМАНИЕ :3

15.10.2024