

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

ВВЕДЕНИЕ

Лектор

Дубовик Марина Владимировна

старший преподаватель кафедры ИСиТ



ауд. 115-1



dubovik@belstu.by



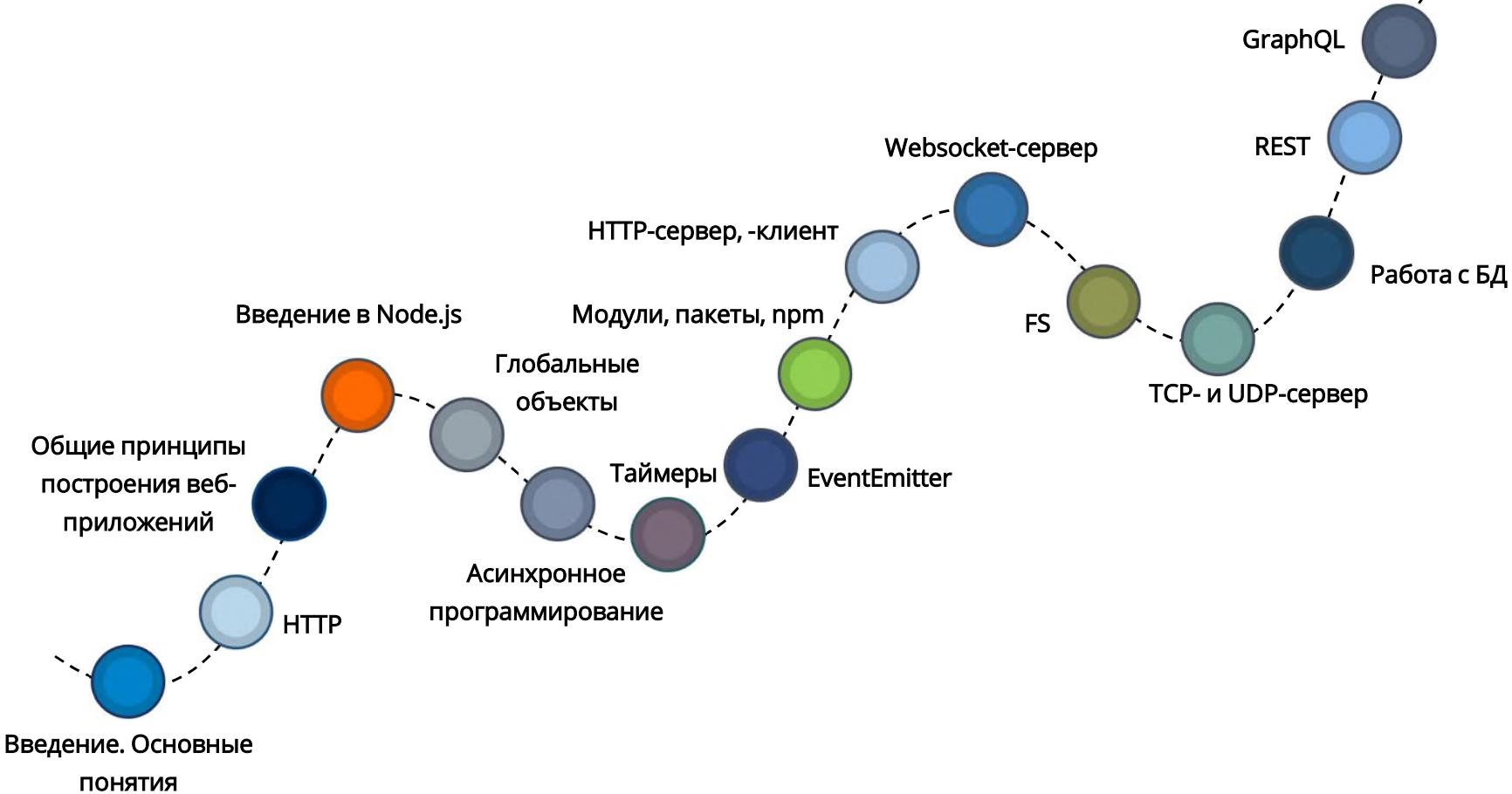
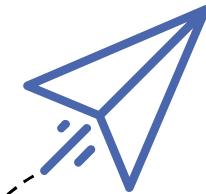
https://vk.com/marina_dubovik



Учебный план

Семестр	ЛК (час)	ЛР (час)	КП (час)	Итого (час)	Форма контроля
3	36	36		72	зачет
4	32	32	40	104	экзамен

Осенний семестр



Весенний семестр

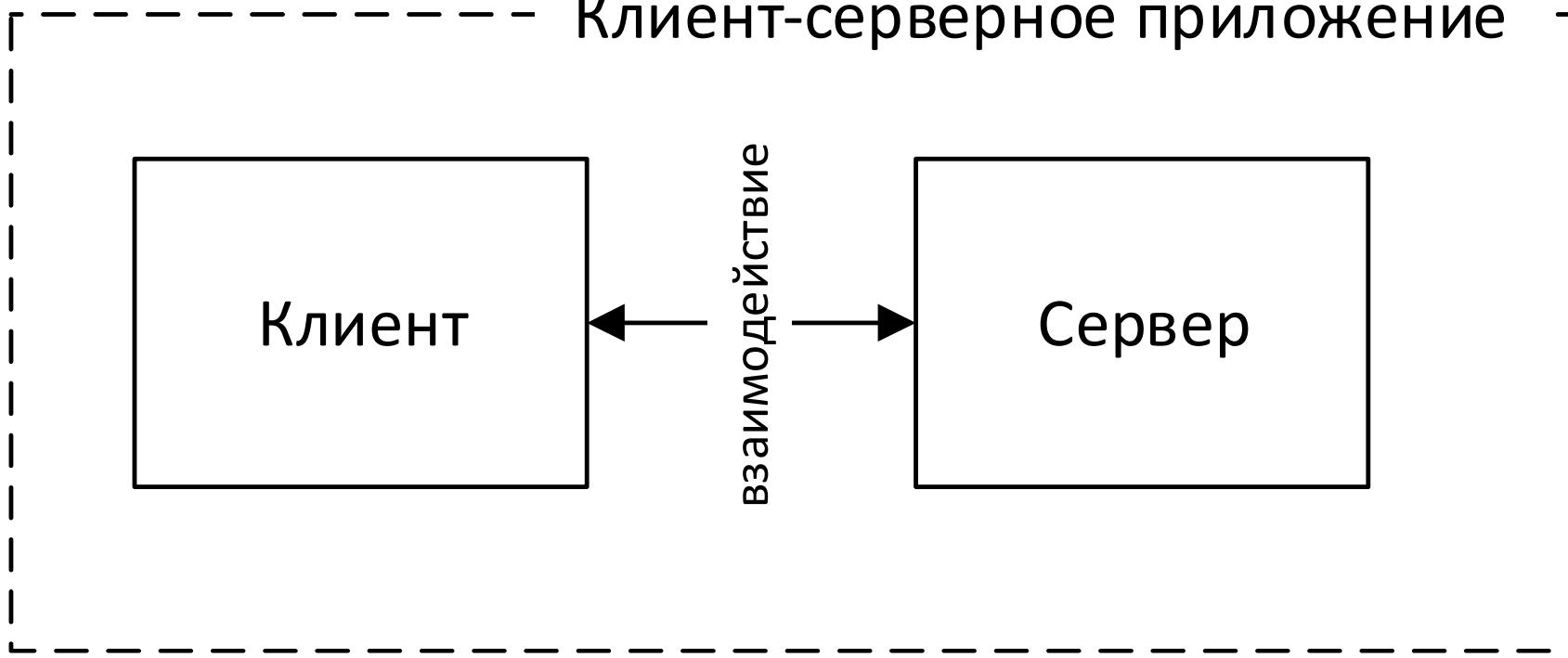


Клиент-серверное = приложение

приложение (программа) с клиент-серверной архитектурой:

- состоит из двух компонент – **клиента** и **сервера**;
- клиент и сервер взаимодействуют между собой в соответствии с заданными правилами (**протоколами**);
- для взаимодействия между клиентом и сервером в соответствии с правилами (протоколом) должно быть установлено **соединение**;
- **инициатором соединения – клиент**.

Клиент-серверное приложение

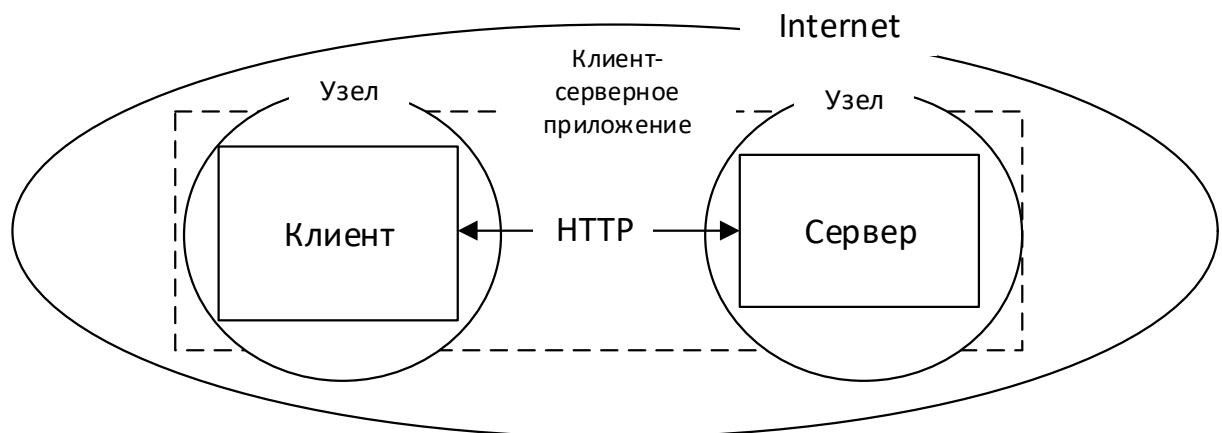


Web- приложение =

клиент-серверное приложение, у которого клиент и сервер взаимодействуют по некоторому протоколу прикладного уровня.

Когда говорят о разработке web-приложения, говорят о разработке **frontend** (клиента) и **backend** (сервера)

Курс посвящен разработке серверной части web-приложения или иначе разработке web-сервера (backend).



Узел сети
Интернет =

устройство, имеющее **IP-адрес** и подключенное к **сети Интернет** (обычно к сети Интернет-провайдера).

Каждый узел характеризуется своей программно-аппаратной платформой – аппаратурой и операционной системой.

Сеть Интернет

=

- 1) сеть на основе TCP/IP;
- 2) стандарты Internet (RFC, STD);
- 3) службы Интернет (DNS, SMTP/POP3/IMAP, WWW, FTP, Telnet, SSH, ...);
- 4) организации, управляющие сетью Internet (ISOC, IETF, W3C, ICANN, IANA, ...).

Кроссплатформенное
приложение =

приложение, способное
работать на **более чем одной**
программно-аппаратной
(аппаратура + операционная
система) **платформе**.



Кроссплатформенность может быть достигнута различными способами:

- 1) на уровне компилятора (C, C++);
- 2) на уровне среды (или фреймворка) исполнения (Java/JVM, C#/.NET CORE/CLR, JS/Node, ...)

C++



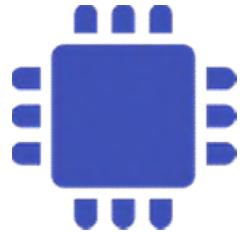
Java/C#



Технологии для разработки кроссплатформенных web-серверов

- PHP / Apache, LAMP;
- Java / JVM / Application Server;
- C# / ASP.NET CORE;
- Python / Django;
- Ruby on Rails;
- JS / Node.js,

Ресурсы, потребляемые web-сервером



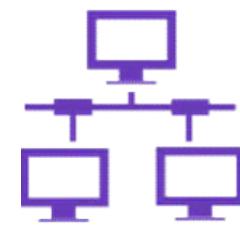
Процессор



Оперативная
память



Жесткий
диск



Сетевой
интерфейс

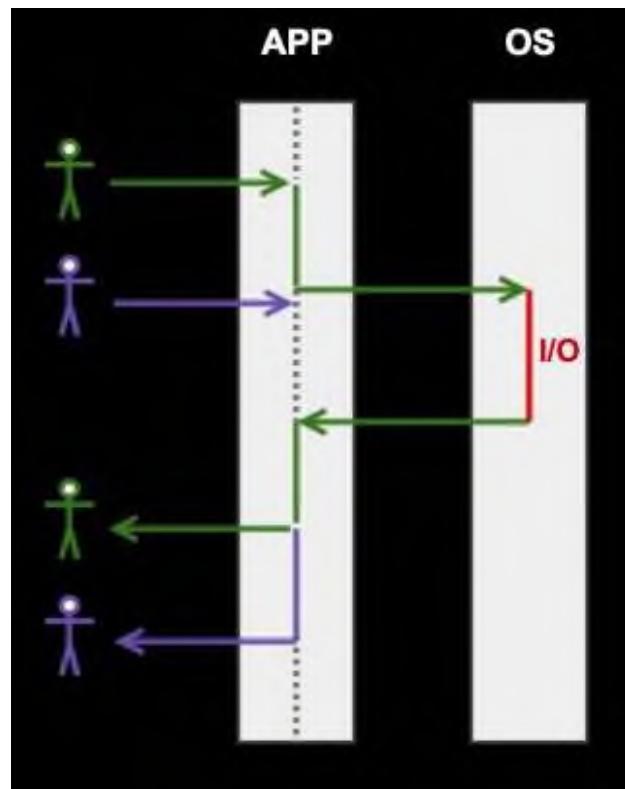
Вычислительные задачи можно разделить на две категории:

- CPU bound – нагружающие процессор (*вычисление факториала, вычисление хэша, процесс шифрования...*). Такие операции **нагружают вычислительные мощности** текущего устройства.
- I/O bound – требующие **ввода/вывода** (*поиск по файлу, подсчет числа строк в файле, копирование директории...*). В своей базовой форме Node.js лучше всего подходит для этого типа вычислений.

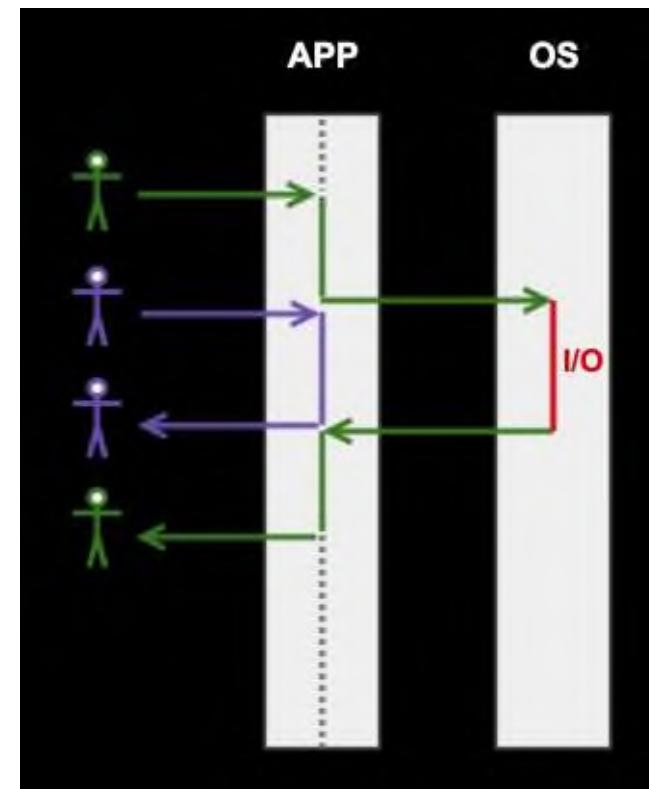


IO-операции

blokiрующие



неблокирующие



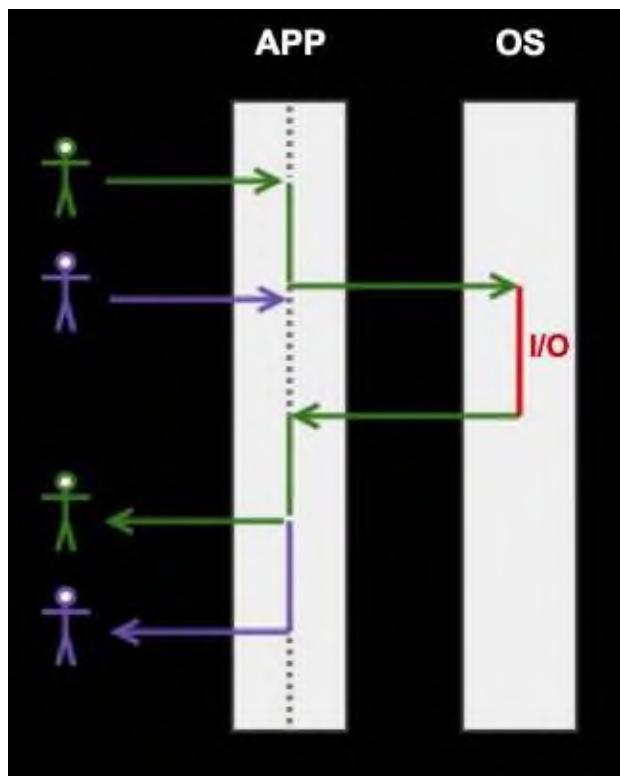
Подходы для решения проблем блокирующего ввода/вывода:

1. Применение **многопоточности** (ограничение по количеству потоков, каждый поток требует дополнительной памяти, синхронизация, отладка).
2. Применение паттерна **Reactor**.

Apache - многопоточность, Nginx - Reactor.

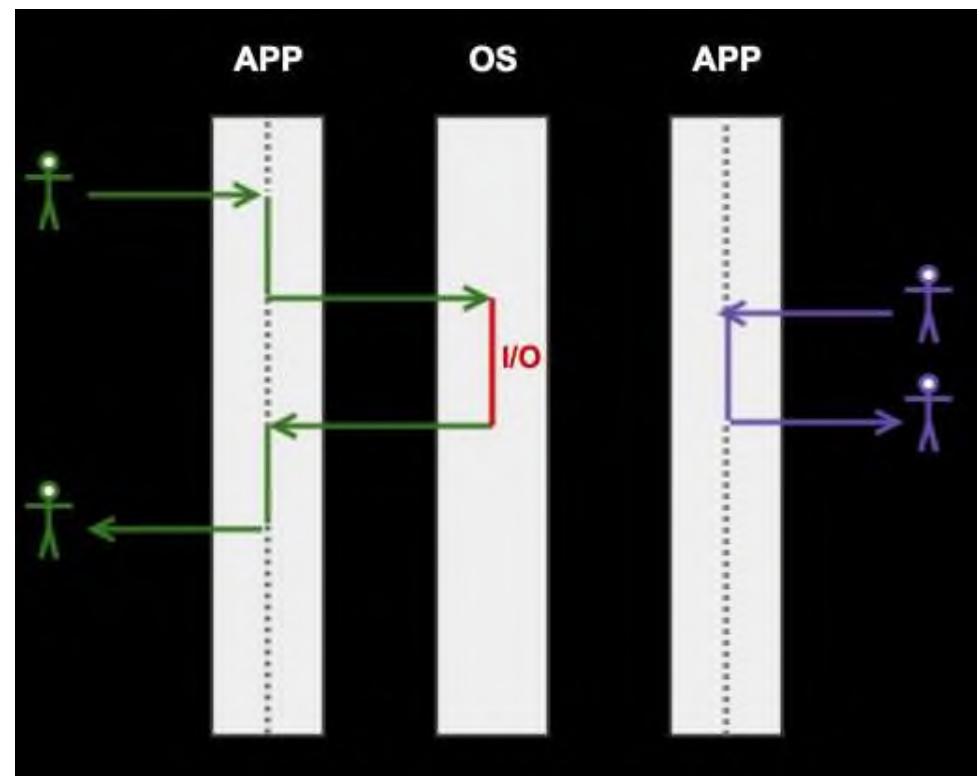
Однопоточность

(единственная задача в один момент времени)



Многопоточность

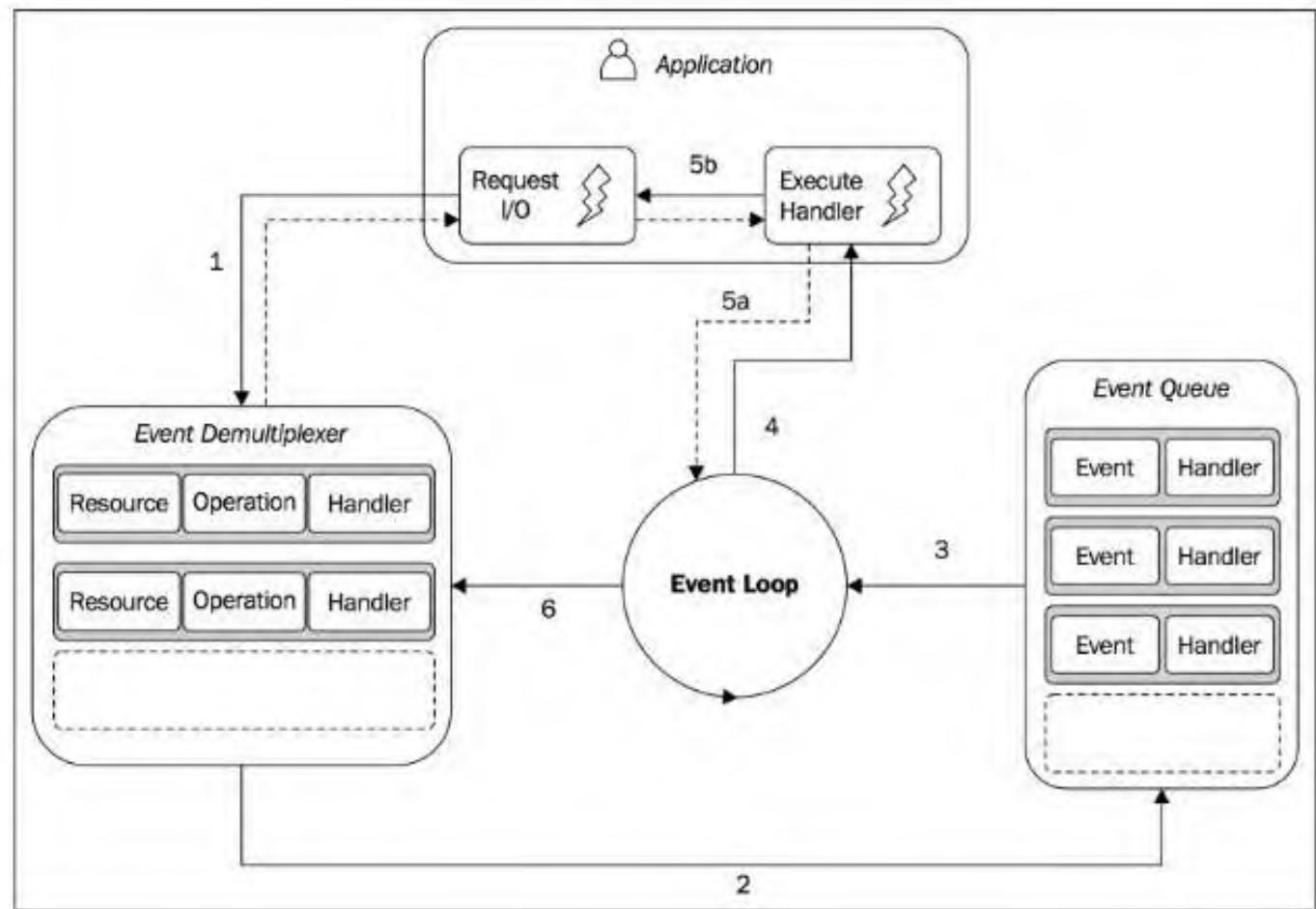
(различные задачи в один момент времени)



Паттерн Reactor

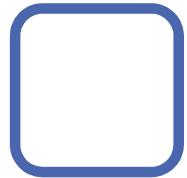
Используется при обработке параллельных запросов к сервису.

Демультиплексор разбирает прибывшие запросы и синхронно перенаправляет их на соответствующие обработчики запросов.



Какой подход используется в Node.js?

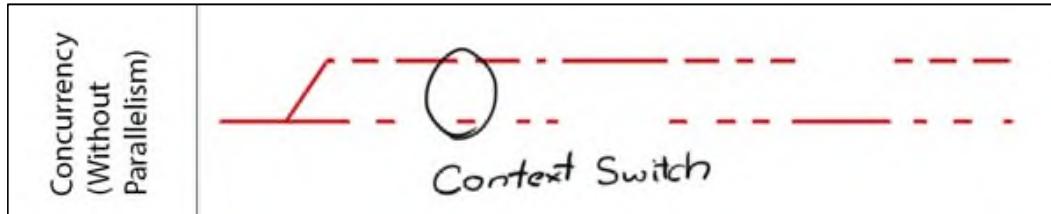
Многопоточность



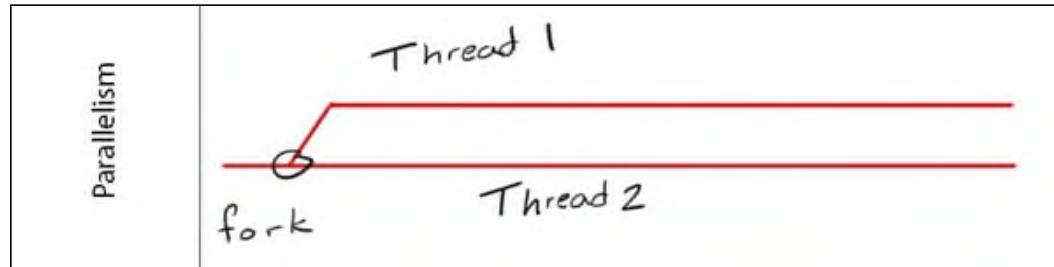
Паттерн Reactor



Конкурентность и параллельность



- Работа с несколькими потоками управления.
- Каждый поток получает квант времени на исполнение.



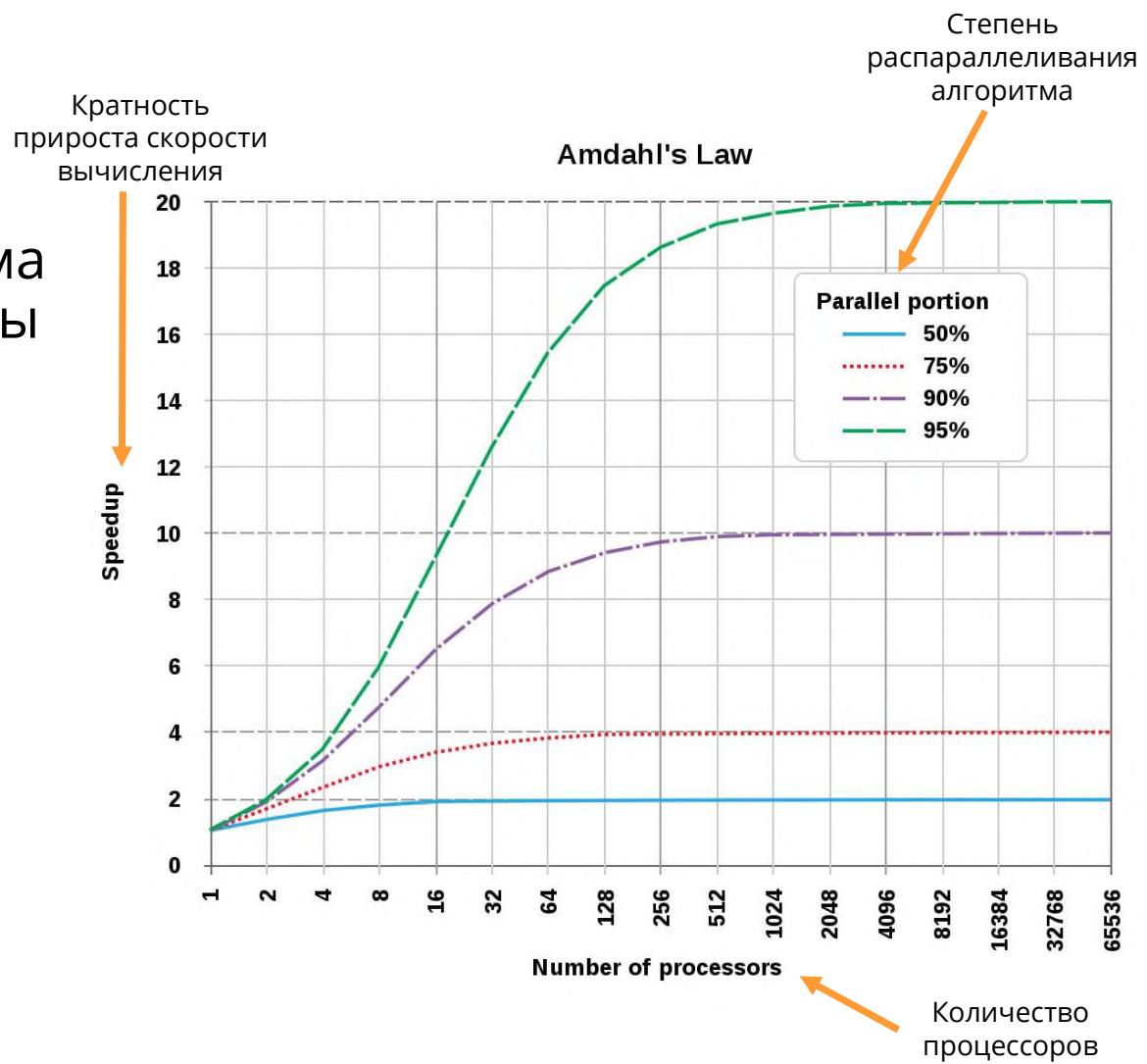
- Необходима аппаратная составляющая.
- Наличие потоков необязательно.

Закон Амдала

Идеальный случай: система из n процессоров могла бы ускорить вычисления в n раз.

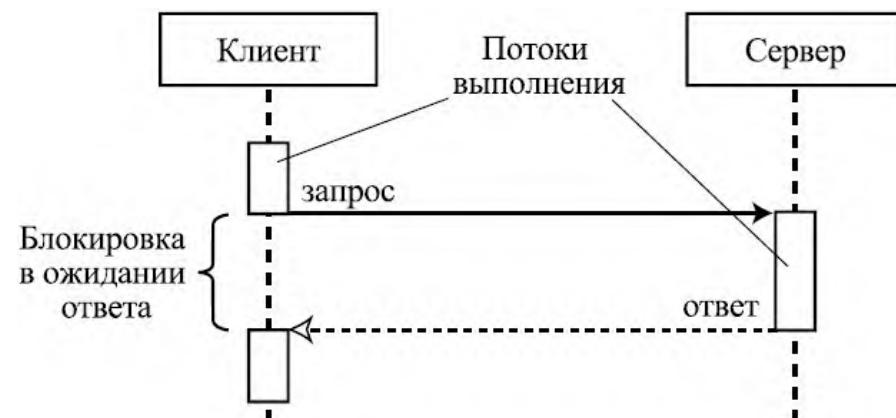
Проблемы:

- невозможность полного распараллеливания ни одной из задач;
- невозможность одинаковой загрузки каждого процессора.



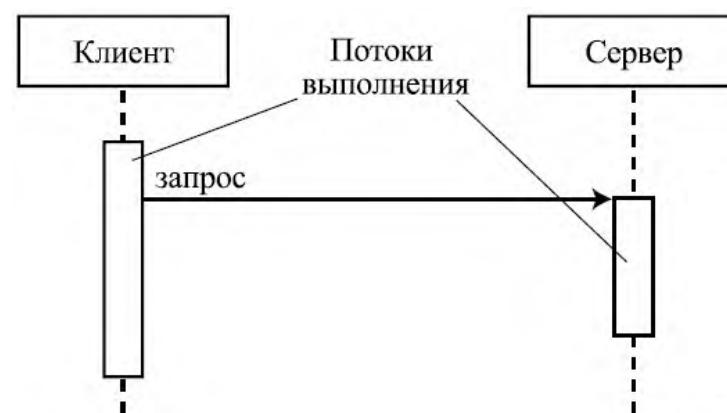
Синхронный запрос

запрос, при котором поток, выдавший http-запрос, блокируется до поступления ответа



Асинхронный запрос

запрос, при котором поток, выдавший http-запрос, не блокируется до поступления запроса; для обработки ответа применяется функция обратного вызова.



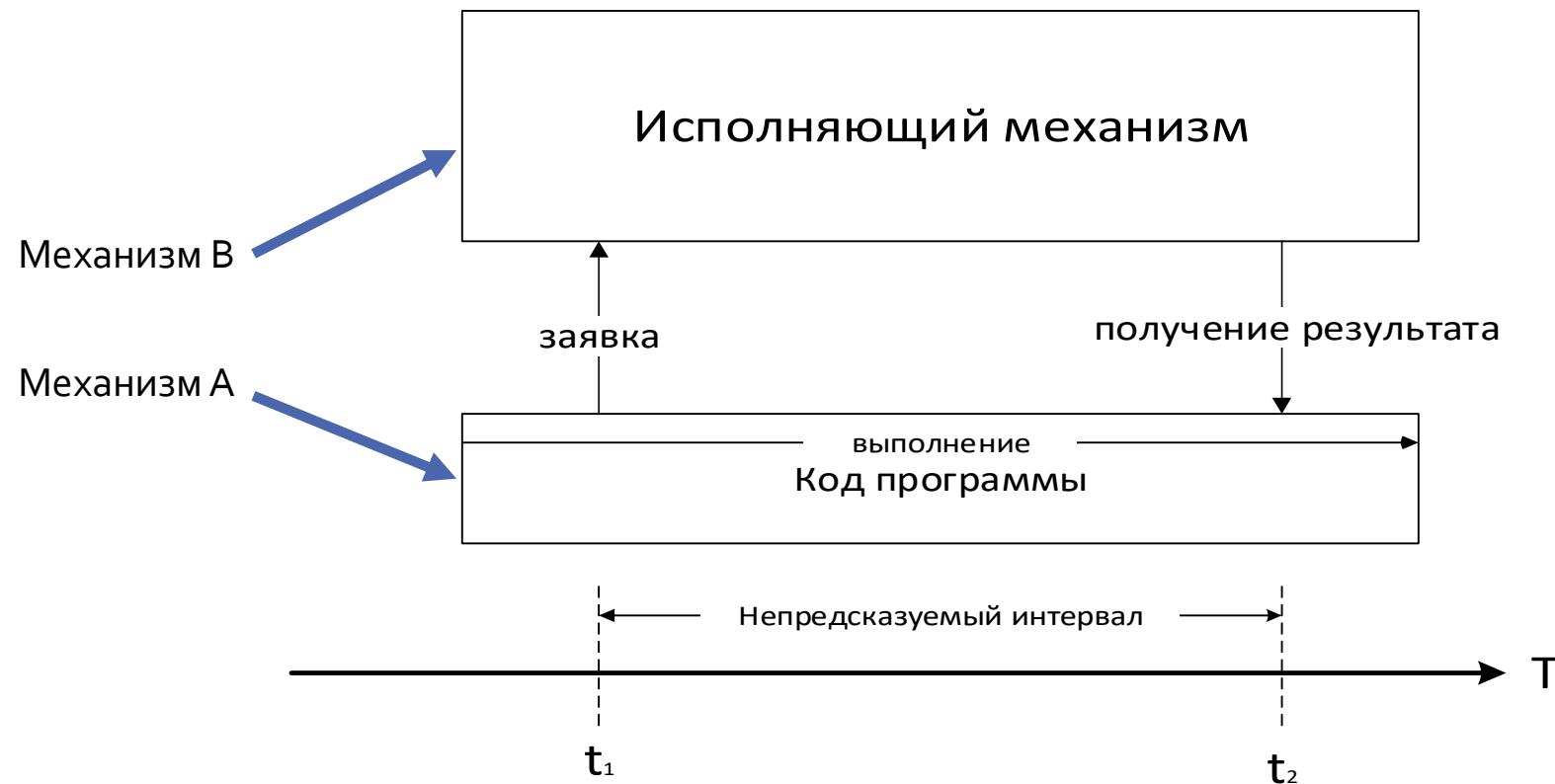
Асинхронность =

операция называется асинхронной, если ее выполнение осуществляется в 2 фазы:

- 1) **заявка на исполнение;**
- 2) **получение результата;** при этом участвуют два механизма: А-механизм, формирующий заявку и потом получающий результат; В-механизм, получающий заявку от А, исполняющий операцию и отправляющий результат А; продолжительность исполнения операции В-механизмом, как правило, непредсказуемо; в то время пока В-механизм исполняет операцию, А-механизм выполняет собственную работу.

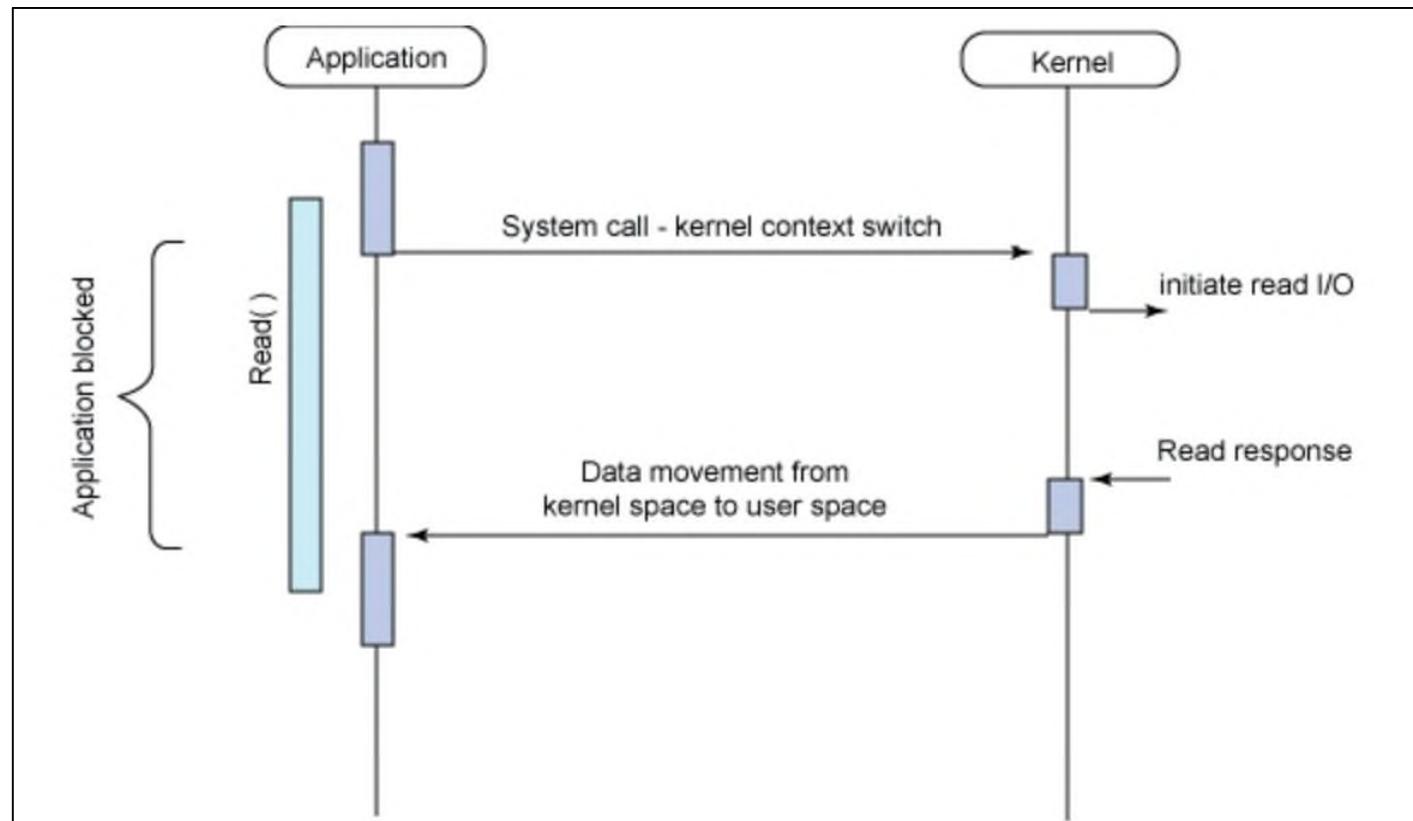
Применение асинхронности не противоречит применению многопоточности.

Асинхронность в программировании



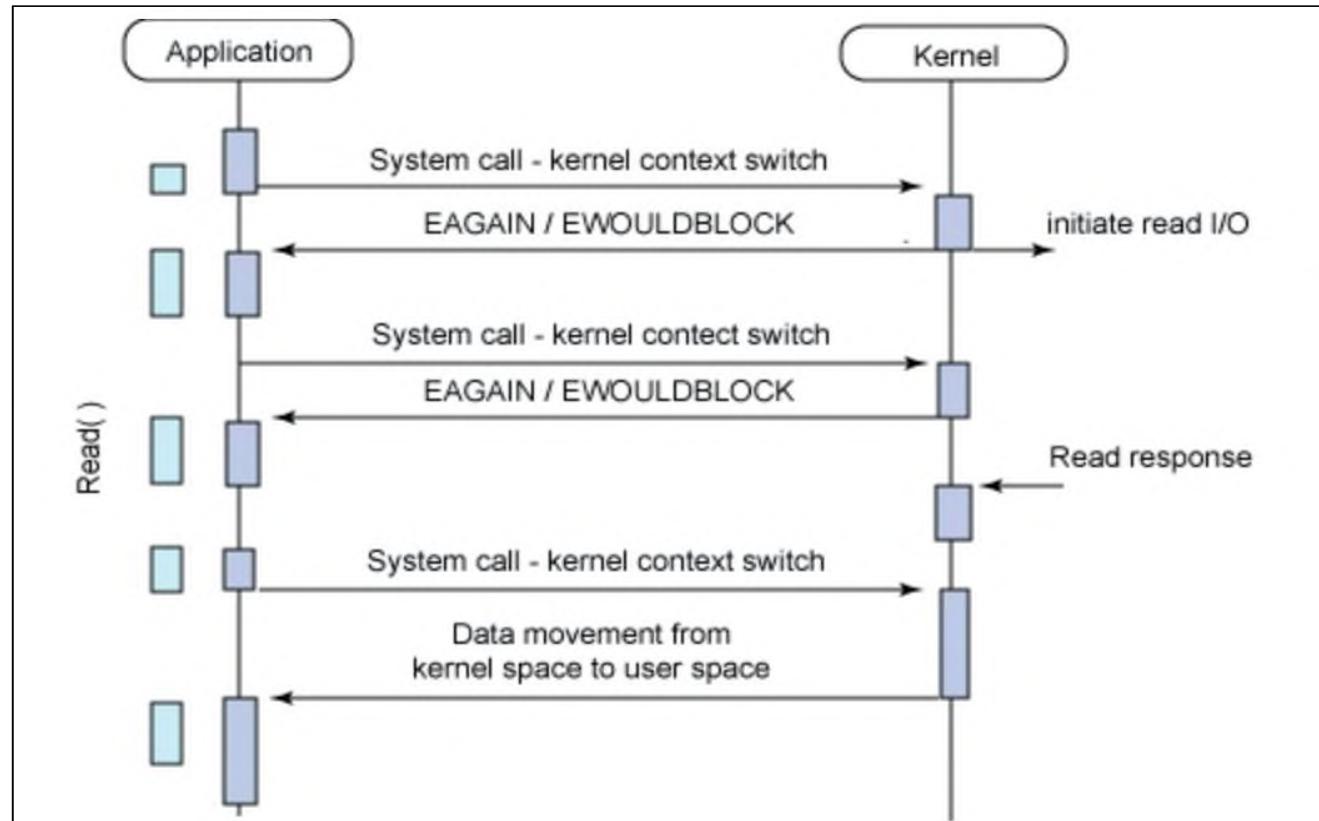
Синхронный блокирующий ввод/вывод

Приложение
блокируется до тех
пор, пока
операция
ввода/вывода не
будет завершена.



Синхронный неблокирующий ввод/вывод

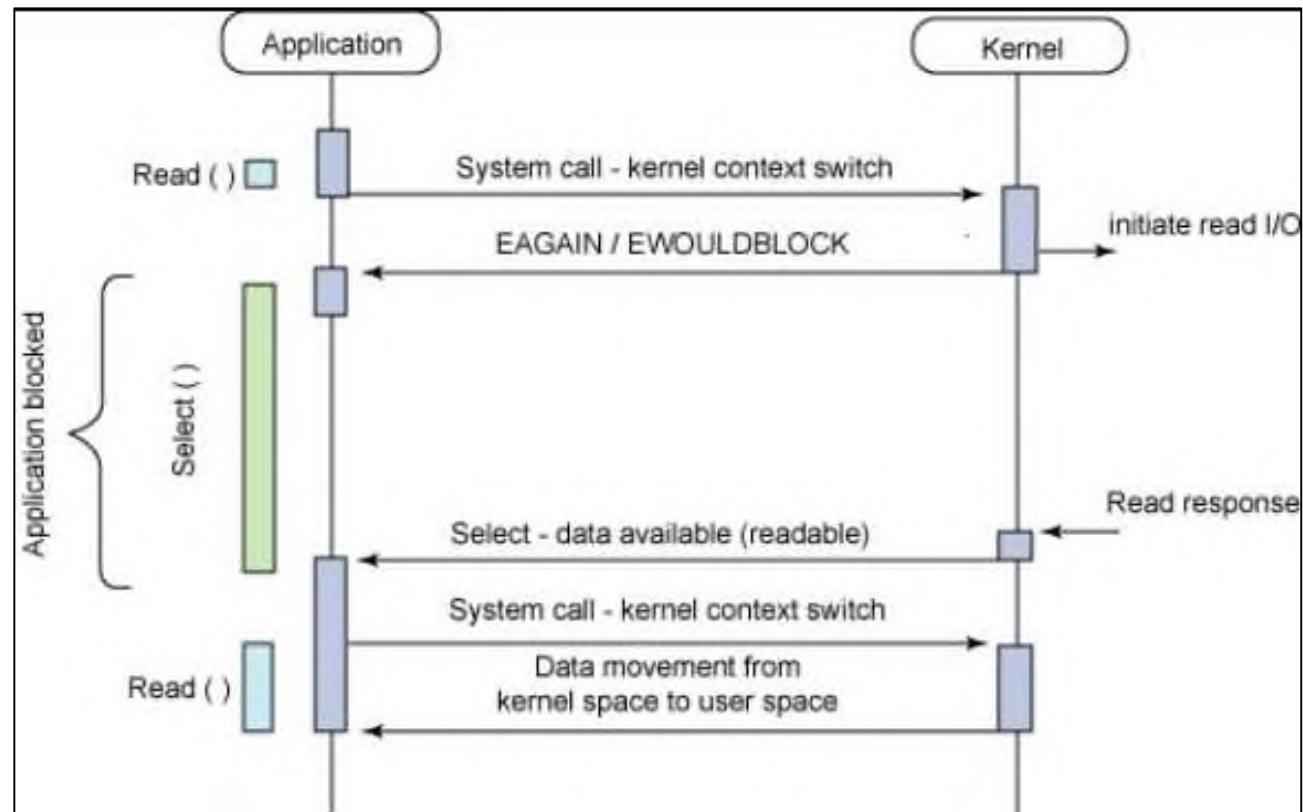
Приложение **может**
сделать что-то еще
вместо ожидания
ввода/вывода,
однако происходит
увеличение
количество
переключений
контекста между
ядром и
пользовательским
пространством.



Асинхронный блокирующий ввод/вывод

Ввод-вывод инициируется как неблокирующий, но **уведомление** принимается в режиме **блокировки** с помощью системного вызова `select()`.

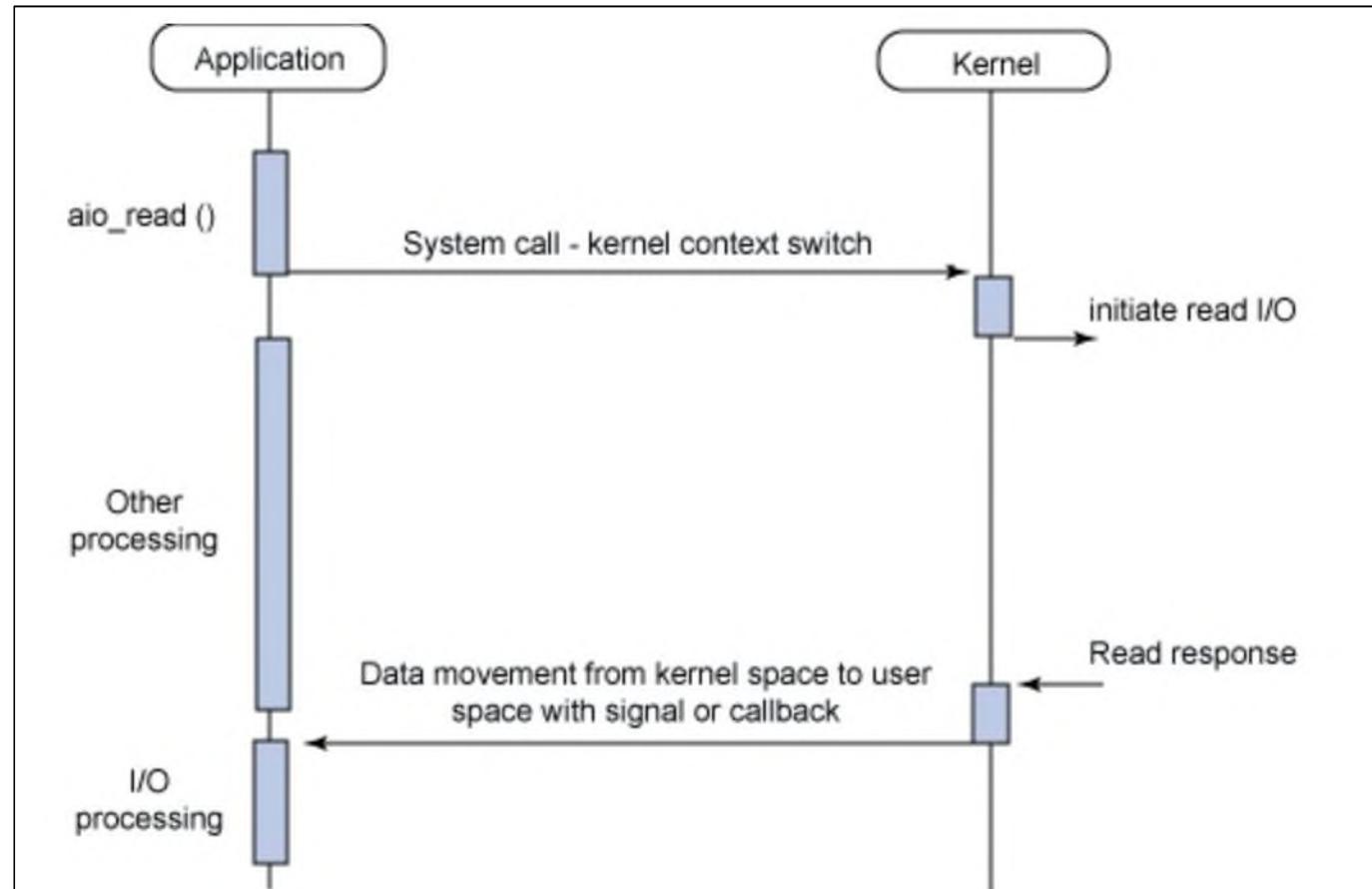
Как только файловый дескриптор готов к работе, вызов `read()` разблокируется, и приложение сможет извлечь и обработать данные.



Асинхронный неблокирующий ввод/вывод

Приложение **может выполнять вычисления**, пока чтение выполняется в фоне ядром.

При готовности ответа генерируется сигнал или обратный вызов для завершения ввода-вывода.



AJAX =
Asynchronous JavaScript
and XML

методология (подход) построения
динамических приложений, в
которых **не осуществляется полная**
перезагрузка HTML-страниц.

Пример асинхронных запросов

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
  <title>Async requests</title>
</head>
<body>
  <h1>XMLHttpRequest</h1>
  <input type="submit" value="get users" onclick="getUsers()">
  <div id="divResult1"></div>

  <h1>Fetch</h1>
  <h2>Login</h2>
  <form id="myForm">
    Email: <br>
    <input type="text" name="email" required>
    <br> <br>
    Password: <br>
    <input type="password" name="password" required>
    <br> <br>
    <input type="submit" value="Login"></button>
  </form>
  <div id="divResult2"></div>

  <h1>JQuery</h1>
  <input type="submit" value="get user by ID" onclick="getUser()">
  <div id="divResult3"></div>

  <script>
    ...
  </script>
</body>

</html>
```

XMLHttpRequest

get users

Fetch

Login

Email:

Password:

Login

JQuery

get user by ID

XMLHttpRequest

```
<script>
  function getUsers() {
    const xhr = new XMLHttpRequest();

    xhr.open("GET", "https://reqres.in/api/users?page=2", true);

    console.log("XMLHttpRequest ", 1);

    xhr.onreadystatechange = () => {
      console.log("XMLHttpRequest ", 2);
      if (xhr.readyState == 4) {
        if (xhr.status == 200)
          document.getElementById("divResult1").innerText = xhr.responseText;
      }
    };

    xhr.onerror = (e) => {
      console.log("XMLHttpRequest.onerror ", e)
    };

    console.log("XMLHttpRequest ", 3);

    xhr.send();

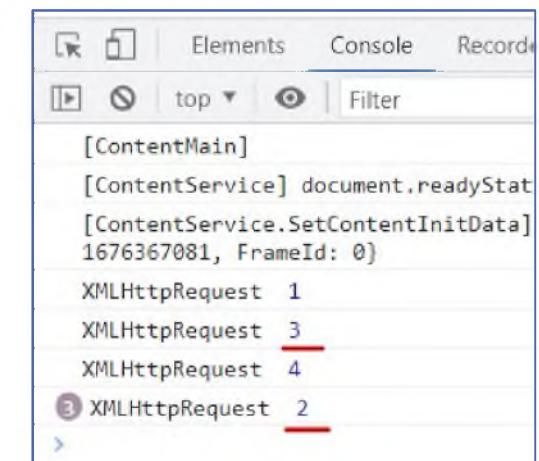
    console.log("XMLHttpRequest ", 4);
  };

  // ...
</script>
```

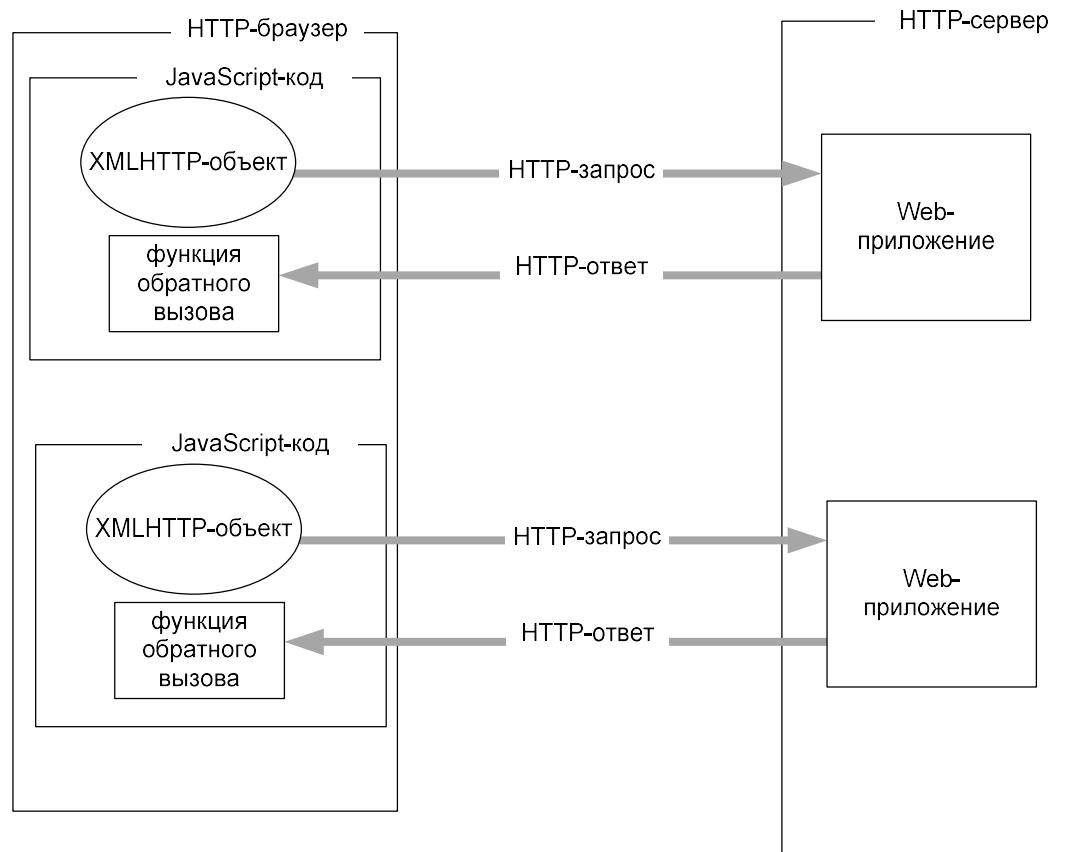
XMLHttpRequest

get users

```
{"page":2,"per_page":6,"total":12,"total_pages":2,"data": [{"id":7,"email":"michael.lawson@reqres.in","first_name":"Michael","last_name":"Lawson"}, {"id":8,"email":"lindsay.ferguson@reqres.in","first_name":"Lindsay","last_name":"Ferguson"}, {"id":9,"email":"tobias.funke@reqres.in","first_name":"Tobias","last_name":"Funke"}, {"id":10,"email":"byron.fields@reqres.in","first_name":"Byron","last_name":"Fields"}, {"id":11,"email":"george.edwards@reqres.in","first_name":"George","last_name":"Edwards"}, {"id":12,"email":"rachel.howell@reqres.in","first_name":"Rachel","last_name":"Howell"}],"support": {"url": "https://reqres.in/#support-heading","text": "To keep Reqres free, please support us by purchasing a license."}}
```



Асинхронный XMLHttpRequest запрос



```
● ○ ●
<script>
  // ...
  document.querySelector("#myForm").addEventListener("submit", (event) => {
    event.preventDefault();

    let formData = new FormData(event.target);
    let data = Object.fromEntries(formData);

    fetch("https://reqres.in/api/register", {
      method: "POST",
      mode: "cors",
      headers: {
        "Accept": "application/json",
        "Content-Type": "application/json"
      },
      body: JSON.stringify(data)
    })
    .then(resp => resp.text())
    .then(respText => document.getElementById("divResult2").innerText = respText)
    .catch(err => console.log(err))
  });

</script>
```

fetch

Login

Email:
eve.holt@reqres.in

Password:

{"id":4,"token":"QpwL5tke4Pnpja7X4"}

JQuery

```
<script>
  //...
  function getUser() {
    $.ajax({
      url: "https://reqres.in/api/users/4",
      type: "GET",
      dataType: "json",
      success: function (data) {
        document.getElementById("divResult3").innerText = JSON.stringify(data.data);
        console.log(data.data)
      },
      error: function (e) {
        console.log(e);
      },
    });
  }

</script>
```

JQuery

get user by ID

{"id":4,"email":"eve.holt@reqres.in","first_name":"Eve","last_name":"Holt"}

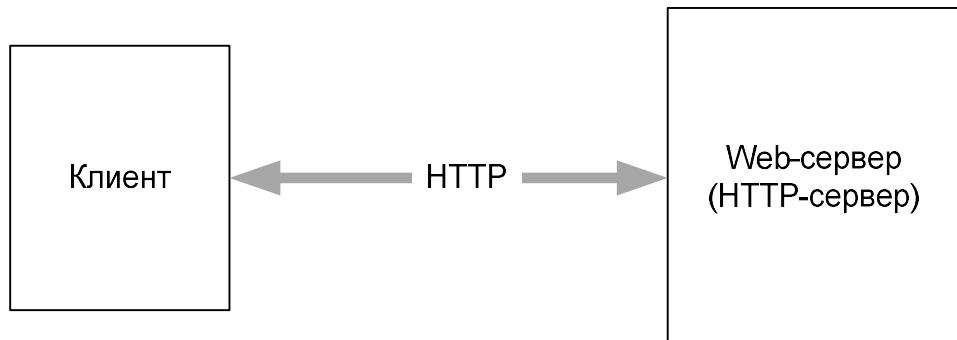
ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

ОСНОВЫ WEB-ПРОГРАММИРОВАНИЯ

Web-
программирование =

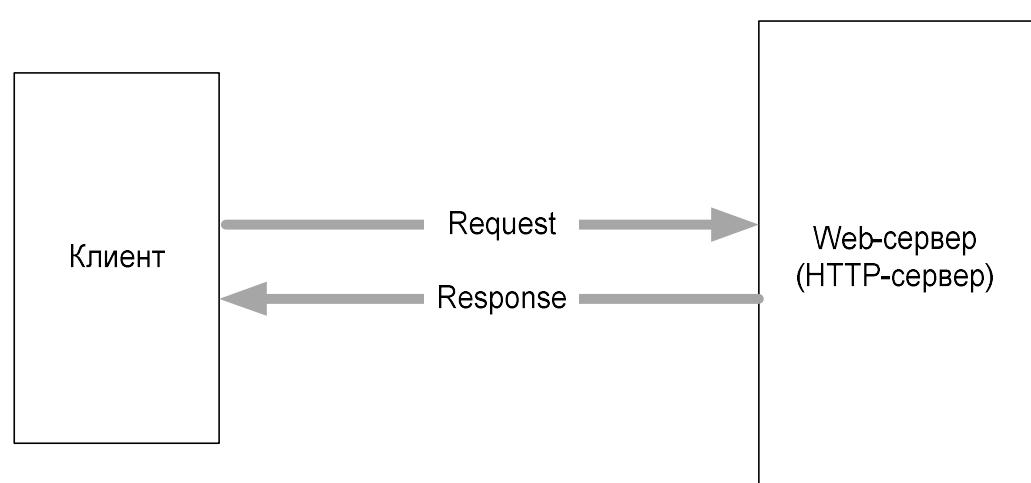
раздел программирования,
ориентированный на разработку
web-приложений.

Архитектура web-приложения

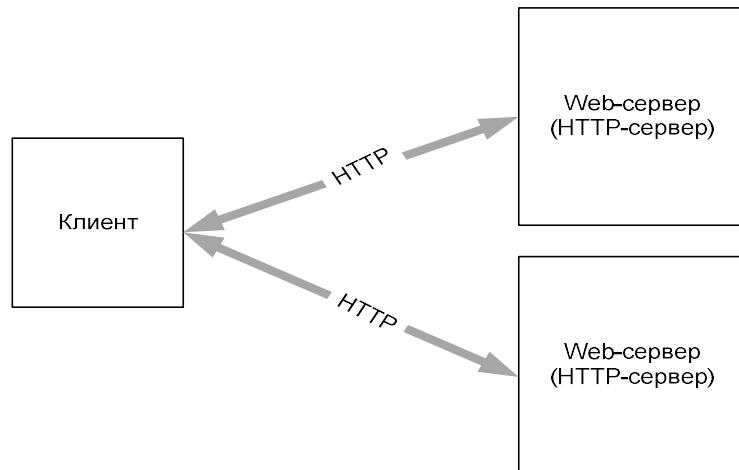


Web-приложение состоит из клиента и сервера, взаимодействующих по протоколу HTTP.

Взаимодействие по протоколу HTTP происходит посредством запросов и ответов.

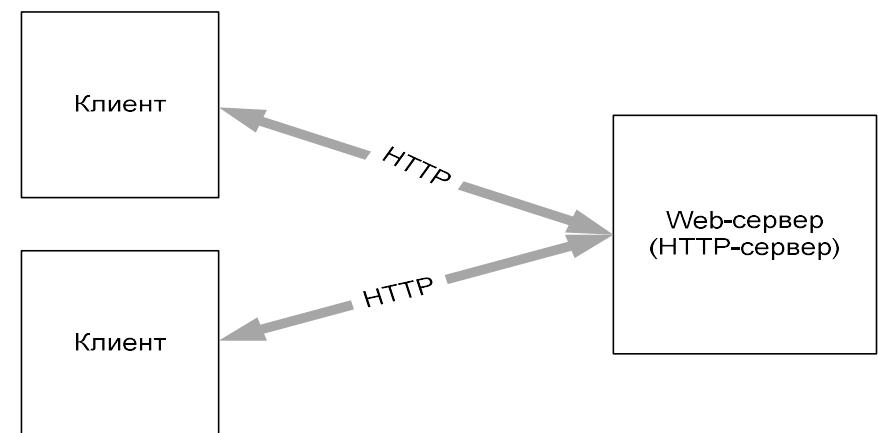


Архитектура web-приложения

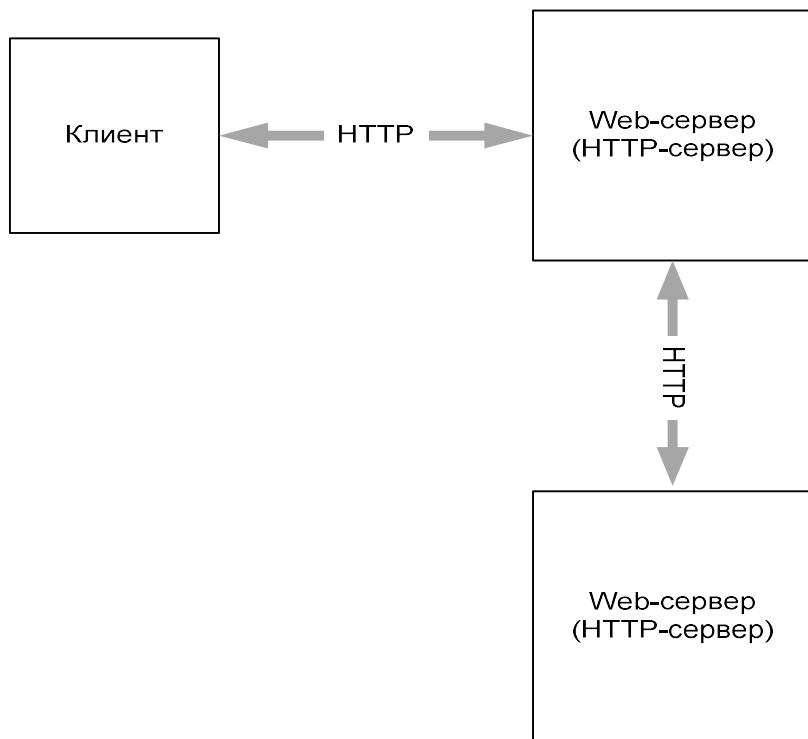


Логично, что сервер может принимать
запросы от множества клиентов.

Обрабатывать запросы клиента могут
различные серверы из кластера серверов.



Архитектура web-приложения



Сервер, в свою очередь, может выступать клиентом по отношению к другому серверу.

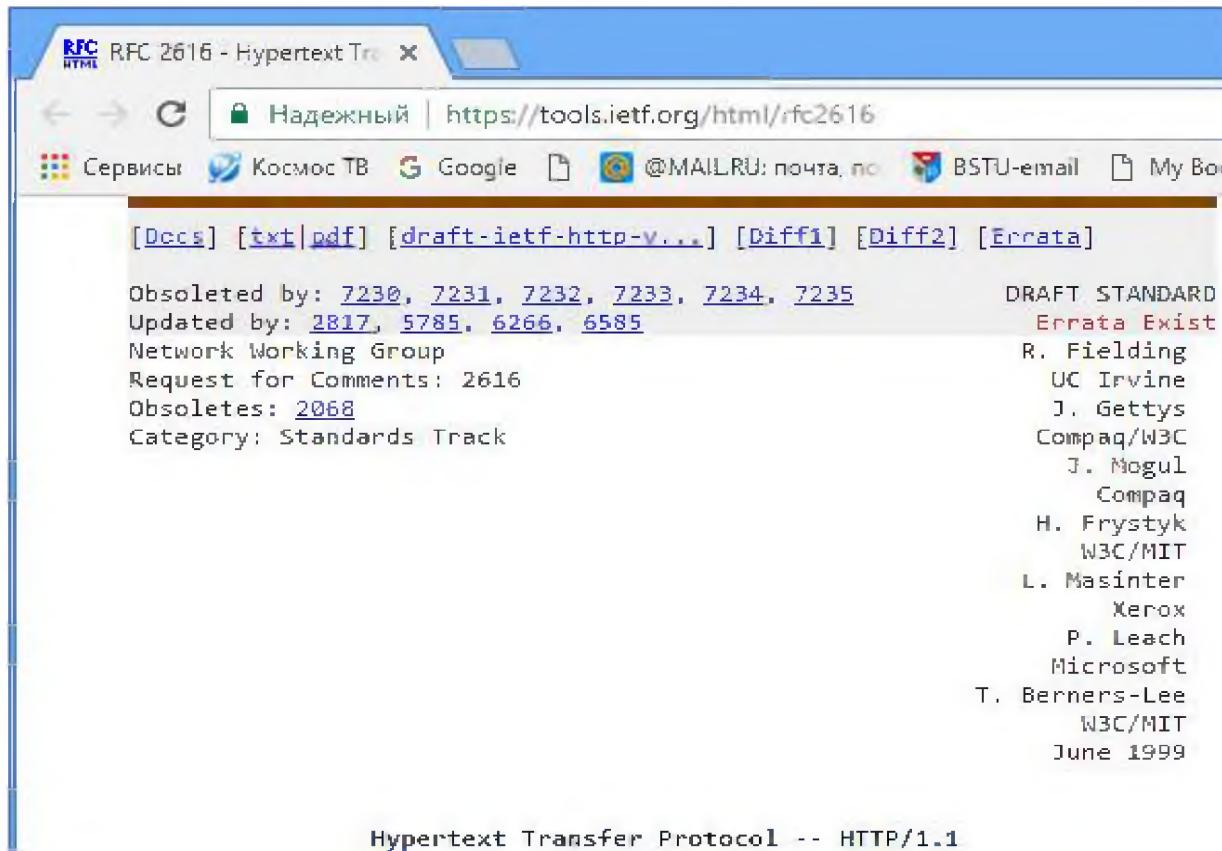
HTTP

протокол прикладного уровня модели OSI

Основные свойства

- версии **HTTP/1.1** – действующий (текстовый), **HTTP/2** – действующий (бинарный), **HTTP/3** (раньше HTTP-over-QUIC, стадия черновика);
- два типа абонентов: **клиент и сервер**;
- два типа сообщений: **request и response**;
- от клиента к серверу – request;
- от сервера к клиенту – response;
- на один request всегда один response, иначе ошибка;
- одному response всегда один request, иначе ошибка;
- **stateless**;
- TCP-порты: 80, 443;
- для адресации используется URI или URL;
- поддерживается W3C, описан в нескольких **RFC**.

RFC2616 (стандарт HTTP)



RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1

Надежный | <https://tools.ietf.org/html/rfc2616>

[Docs] [txt|pdf] [draft-ietf-http-v...] [Diff1] [Diff2] [Errata]

Obsoleted by: [7230](#), [7231](#), [7232](#), [7233](#), [7234](#), [7235](#) DRAFT STANDARD
Updated by: [2817](#), [5785](#), [6266](#), [6585](#) Errata Exist
Network Working Group R. Fielding
Request for Comments: 2616 UC Irvine
Obsoletes: [2068](#) J. Gettys
Category: Standards Track Compaq/W3C
J. Mogul Compaq
H. Frystyk W3C/MIT
L. Masinter Xerox
P. Leach Microsoft
T. Berners-Lee W3C/MIT
June 1999

Hypertext Transfer Protocol -- HTTP/1.1

Request

- метод;
- URI;
- версия протокола (HTTP/1.1);
- заголовки (пары: имя/значение);
- тело.

```
POST /cgi/process HTTP/1.1
User-Agent: Mozilla/4.0 (compatible)
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

licenseID=string&content=string
```

Response

- версия протокола (HTTP/1.1);
- код состояния (1xx, 2xx, 3xx, 4xx, 5xx);
- пояснение к коду состояния;
- заголовки (пары: имя/заголовок);
- тело.

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Методы HTTP

- **GET** – получение представления ресурса (идемпотентный);
- **POST** – передача сущности заданному ресурсу;
- **PUT** – замена представления указанного ресурса данными запроса (идемпотентный);
- **DELETE** – удаление указанного ресурса (идемпотентный);
- **OPTIONS** – определение возможностей веб-сервера;
- **HEAD** – аналогичен GET, но в ответе нет тела (идемпотентный);
- **TRACE** – предоставление информации о добавлении или изменении данных в запрос промежуточными серверами (идемпотентный);
- **CONNECT** - устанавливает двустороннюю связь с указанным ресурсом.

Заголовки HTTP

General: общие заголовки, используются в запросах и ответах.

```
general-header = Cache-Control          ; Section 14.9
                 | Connection           ; Section 14.10
                 | Date                ; Section 14.18
                 | Pragma               ; Section 14.32
                 | Trailer              ; Section 14.40
                 | Transfer-Encoding    ; Section 14.41
                 | Upgrade              ; Section 14.42
                 | Via                 ; Section 14.45
                 | Warning              ; Section 14.46
```

Request: используются только в запросах.

```
request-header = Accept                  ; Section 14.1
                 | Accept-Charset        ; Section 14.2
                 | Accept-Encoding       ; Section 14.3
                 | Accept-Language        ; Section 14.4
                 | Authorization          ; Section 14.8
                 | Expect                  ; Section 14.20
                 | From                   ; Section 14.22
                 | Host                   ; Section 14.23
                 | If-Match                ; Section 14.24
                 | If-Modified-Since       ; Section 14.25
                 | If-None-Match          ; Section 14.26
                 | If-Range                ; Section 14.27
                 | If-Unmodified-Since     ; Section 14.28
                 | Max-Forwards           ; Section 14.31
                 | Proxy-Authorization     ; Section 14.34
                 | Range                  ; Section 14.35
                 | Referer                ; Section 14.36
                 | TE                      ; Section 14.39
                 | User-Agent              ; Section 14.43
```

Заголовки HTTP

Response: используются только
в ответах

```
response-header = Accept-Ranges      ; Section 14.5
                 | Age                ; Section 14.6
                 | ETag               ; Section 14.19
                 | Location            ; Section 14.30
                 | Proxy-Authenticate  ; Section 14.33
                 | Retry-After          ; Section 14.37
                 | Server              ; Section 14.38
                 | Vary                ; Section 14.44
                 | WWW-Authenticate    ; Section 14.47
```

Entity: для сущности в ответах и
запросах

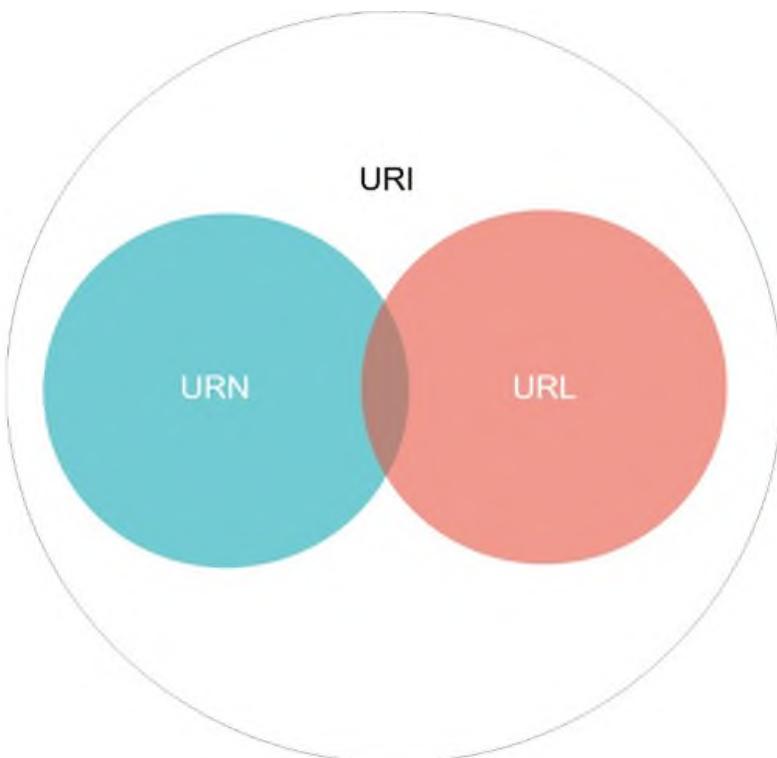
```
entity-header = Allow                ; Section 14.7
                 | Content-Encoding    ; Section 14.11
                 | Content-Language   ; Section 14.12
                 | Content-Length      ; Section 14.13
                 | Content-Location    ; Section 14.14
                 | Content-MD5         ; Section 14.15
                 | Content-Range        ; Section 14.16
                 | Content-Type         ; Section 14.17
                 | Expires              ; Section 14.21
                 | Last-Modified        ; Section 14.29
                 | extension-header
```

Коды состояния ответа

- 1xx: информационные сообщения;
- 2xx: успешный ответ;
- 3xx: переадресация;
- 4xx: ошибка клиента;
- 5xx: ошибка сервера.

```
Status-Code      =
  "100" ; Section 10.1.1: Continue
  "101" ; Section 10.1.2: Switching Protocols
  "200" ; Section 10.2.1: OK
  "201" ; Section 10.2.2: Created
  "202" ; Section 10.2.3: Accepted
  "203" ; Section 10.2.4: Non-Authoritative Information
  "204" ; Section 10.2.5: No Content
  "205" ; Section 10.2.6: Reset Content
  "206" ; Section 10.2.7: Partial Content
  "300" ; Section 10.3.1: Multiple Choices
  "301" ; Section 10.3.2: Moved Permanently
  "302" ; Section 10.3.3: Found
  "303" ; Section 10.3.4: See Other
  "304" ; Section 10.3.5: Not Modified
  "305" ; Section 10.3.6: Use Proxy
  "307" ; Section 10.3.8: Temporary Redirect
  "400" ; Section 10.4.1: Bad Request
  "401" ; Section 10.4.2: Unauthorized
  "402" ; Section 10.4.3: Payment Required
  "403" ; Section 10.4.4: Forbidden
  "404" ; Section 10.4.5: Not Found
  "405" ; Section 10.4.6: Method Not Allowed
  "406" ; Section 10.4.7: Not Acceptable
  "407" ; Section 10.4.8: Proxy Authentication Required
  "408" ; Section 10.4.9: Request Time-out
  "409" ; Section 10.4.10: Conflict
  "410" ; Section 10.4.11: Gone
  "411" ; Section 10.4.12: Length Required
  "412" ; Section 10.4.13: Precondition Failed
  "413" ; Section 10.4.14: Request Entity Too Large
  "414" ; Section 10.4.15: Request-URI Too Large
  "415" ; Section 10.4.16: Unsupported Media Type
  "416" ; Section 10.4.17: Requested range not satisfiable
  "417" ; Section 10.4.18: Expectation Failed
  "500" ; Section 10.5.1: Internal Server Error
  "501" ; Section 10.5.2: Not Implemented
  "502" ; Section 10.5.3: Bad Gateway
  "503" ; Section 10.5.4: Service Unavailable
  "504" ; Section 10.5.5: Gateway Time-out
  "505" ; Section 10.5.6: HTTP Version not supported
extension-code
```

URI, URL, URN



URI, URL, URN – рекомендуется использовать термин URI

- **URI: Uniform Resource Identifier** – унифицированный **идентификатор** ресурса (документ, изображение, файл, электронная почта, ...). Идентифицирует ресурс по имени, местоположению или тому и другому.
- **URL: Uniform Resource Location** – унифицированный **локатор** ресурса, содержащий местонахождение ресурса и способ обращения (протокол) к ресурсу, описывает множество URI.
- **URN: Uniform Resource Name** – унифицированное **имя** ресурса – имя ресурса, не содержащее месторасположение и способ доступа к ресурсу. Позволяет ресурсам перемещаться с одного места на другое. В контексте веба URN практически не используется.

STD 66

Сервисы Космос ТВ Google @MAILRU: почта, по BSTU-email My Boo

[Docs] [txt|pdf] [draft-fielding-ur...] [Diff1] [Diff2] [Errata]

Updated by: [6874](#), [7320](#) INTERNET STANDARD
Errata Exist

Network Working Group T. Berners-Lee
Request for Comments: 3986 W3C/MIT

STD: 66 R. Fielding

Updates: [1738](#) Day Software

Obsoletes: [2732](#), [2396](#), [1808](#) L. Masinter

Category: Standards Track Adobe Systems

January 2005

Uniform Resource Identifier (URI): Generic Syntax

1.1.3. URI, URL, and URN

A URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location"). The term "Uniform Resource Name" (URN) has been used historically to refer to both URIs under the "urn" scheme [RFC2141], which are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable, and to any other URI with the properties of a name.

An individual scheme does not have to be classified as being just one of "name" or "locator". Instances of URIs from any given scheme may have the characteristics of names or locators or both, often depending on the persistence and care in the assignment of identifiers by the naming authority, rather than on any quality of the scheme. Future specifications and related documentation should use the general term "URI" rather than the more restrictive terms "URL" and "URN" [RFC3305].

3. Syntax Components

The generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment.

```
URI      = scheme ":" hier-part [ "?" query ] [ "#" fragment ]  
  
hier-part = "//" authority path-abempty  
          / path-absolute  
          / path-rootless  
          / path-empty
```

The scheme and path components are required, though the path may be empty (no characters). When authority is present, the path must either be empty or begin with a slash ("/") character. When authority is not present, the path cannot begin with two slash characters ("//"). These restrictions result in five different ABNF rules for a path ([Section 3.3](#)), only one of which will match any given URI reference.

The following are two example URIs and their component parts:

```
graph TD; Root["foo://example.com:8042/over/there?name=ferret#nose"] --> Scheme["scheme"]; Root --> Authority["authority"]; Root --> Path["path"]; Root --> Query["query"]; Root --> Fragment["fragment"]; Authority --> Username["username"]; Authority --> Host["host"]; Path --> Over["over"]; Path --> There["there"]; Query --> Name["name"]; Query --> Ferret["ferret"]; Fragment --> Nose["nose"]
```

urn:example:animal:ferret:nose

STD 66

3.2. Authority

Many URI schemes include a hierarchical element for a naming authority so that governance of the name space defined by the remainder of the URI is delegated to that authority (which may, in turn, delegate it further). The generic syntax provides a common means for distinguishing an authority based on a registered name or server address, along with optional port and user information.

The authority component is preceded by a double slash ("//") and is terminated by the next slash ("/"), question mark ("?"), or number sign ("#") character, or by the end of the URI.

```
authority  = [ userinfo "@" ] host [ ":" port ]
```

URI producers and normalizers should omit the ":" delimiter that separates host from port if the port component is empty. Some schemes do not allow the userinfo and/or port subcomponents.

If a URI contains an authority component, then the path component must either be empty or begin with a slash ("/") character. Non-validating parsers (those that merely separate a URI reference into its major components) will often ignore the subcomponent structure of authority, treating it as an opaque string from the double-slash to the first terminating delimiter, until such time as the URI is dereferenced.

3.2.1. User Information

The userinfo subcomponent may consist of a user name and, optionally, scheme-specific information about how to gain authorization to access the resource. The user information, if present, is followed by a commercial at-sign ("@") that delimits it from the host.

```
userinfo  = *( unreserved / pct-encoded / sub-delims / ":" )
```

Use of the format "user:password" in the userinfo field is deprecated. Applications should not render as clear text any data after the first colon ":" character found within a userinfo subcomponent unless the data after the colon is the empty string (indicating no password). Applications may choose to ignore or reject such data when it is received as part of a reference and should reject the storage of such data in unencrypted form. The passing of authentication information in clear text has proven to be a security risk in almost every case where it has been used.

Applications that render a URI for the sake of user feedback, such as in graphical hypertext browsing, should render userinfo in a way that is distinguished from the rest of a URI, when feasible. Such rendering will assist the user in cases where the userinfo has been misleadingly crafted to look like a trusted domain name ([Section 7.6](#)).

RFC 2141 (URN)

2. Syntax

All URNs have the following syntax (phrases enclosed in quotes are REQUIRED):

`<URN> ::= "urn:" <NID> ":" <NSS>`

where `<NID>` is the Namespace Identifier, and `<NSS>` is the Namespace Specific String. The leading "urn:" sequence is case-insensitive. The Namespace ID determines the syntactic interpretation of the Namespace Specific String (as discussed in [1]).

PURL

Persistent Uniform Resource Locator – постоянный унифицированный локатор ресурса.

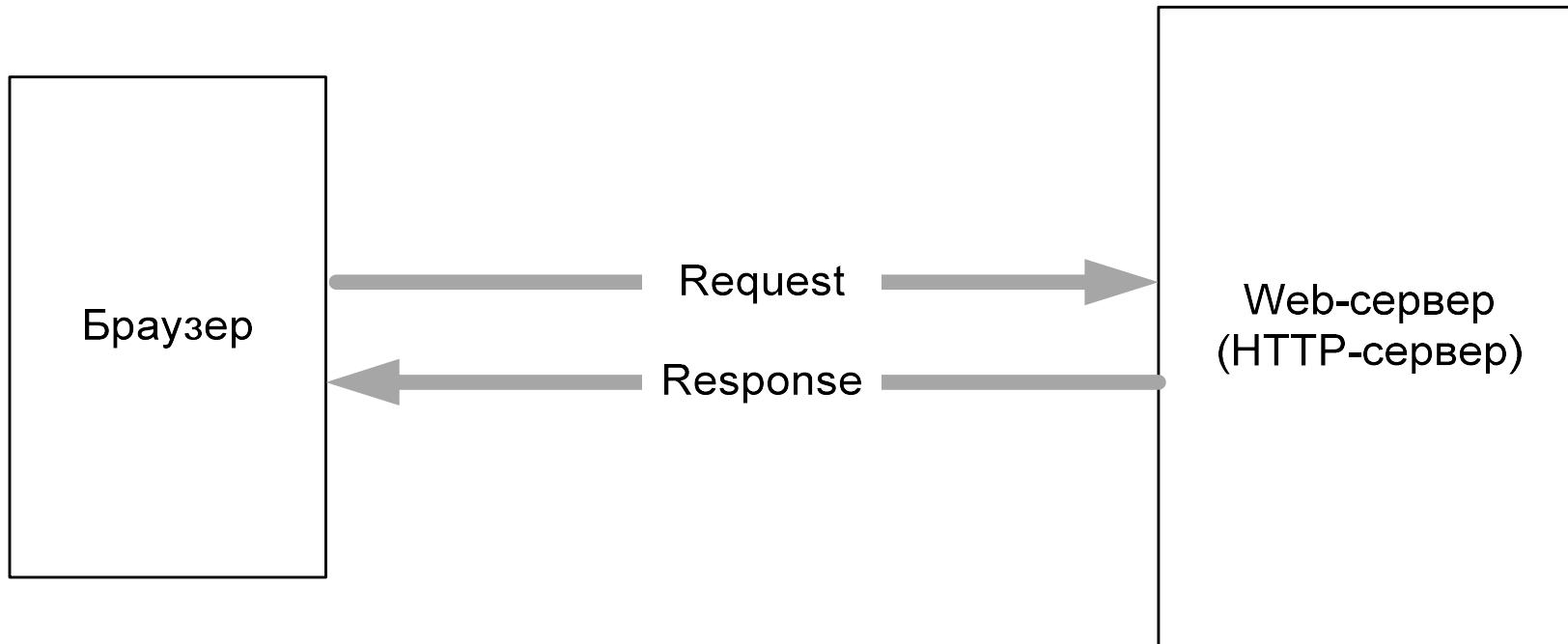
Например: <http://purpl.by/bstu/faculties/lh/lv.html>,

<http://purpl.by/bstu> – указывает на БД, в которой по имени bstu можно найти URL <http://www.bstu.by> и получить искомый URL:

<http://www.bstu.by/faculties/lh/lv.html>.

Доступ к конечному ресурсу через redirect.

Web-браузер – это клиент

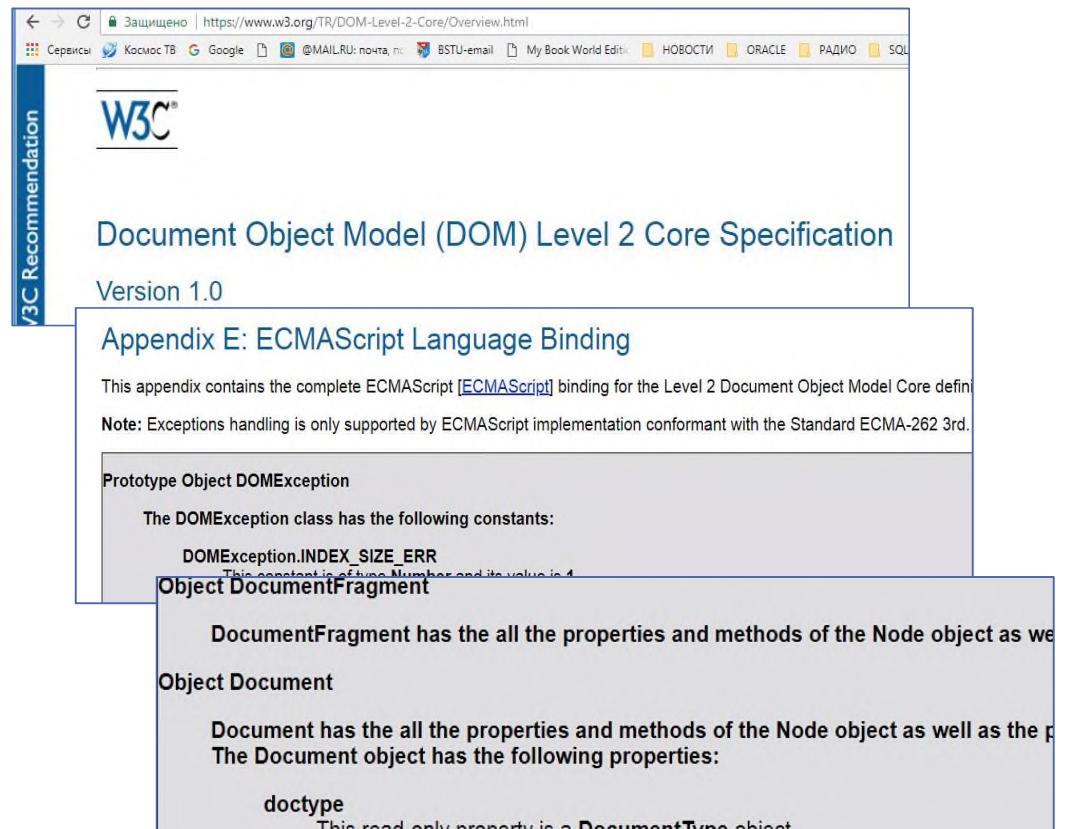


Способы генерации запросов web-браузером

- адресная URI-строка (GET);
- HTML-тег: <form> (POST, GET);
- HTML-тег: <a> (GET);
- HTML-тег: (GET);
- HTML-тег: <script> (GET);
- HTML-тег: <link> (GET);
- HTML-тег: <audio> (GET);
- HTML-тег: <video> (GET);
- HTML-тег: <frame> (GET), не поддерживается в HTML5;
- Объект web-браузера: XMLHttpRequest;
- JavaScript API: **web-сокеты**;
- Fetch API: fetch;
- JQuery;
- Axios.

Модель DOM (Document Object Model)

- Объектная модель документа, **представление HTML-документа** web-браузером **в виде объектов**, интерфейс JavaScript для доступа к содержимому HTML-документа.
- Три уровня DOM0, DOM1, DOM2, DOM3.
- Современные браузеры уровня 2 с элементами уровня 3.
- Содержит описание JS (ECMAScript).



CSS (Cascading Style Sheets)

- Каскадные таблицы стилей – код, используемый для **стилизации** вашей страницы.
- Уровни: CSS1, CSS2, **CSS3** (текущий), CSS4 (в разработке с 2011).
- CSS-фреймворки: Bootstrap, Kube, Foundation, Semantic UI, PureCSS.
- CSS-расширения: Sass, SCSS, LESS.

Рекомендуют использовать HTML5, CSS3, SVG, JavaScript (стандарты: ECMAScript5, ECMA-262-6, ECMA-262-7).

```
h1 {  
    color: blue;  
    background-color: yellow;  
}  
  
p {  
    color: red;  
}
```

ECMAScript

ECMAScript — это язык программирования, используемый в качестве основы для построения других скриптовых языков.

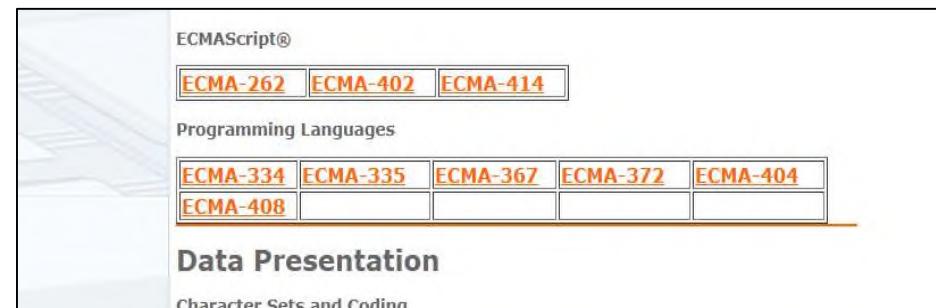
Стандартизирован международной организацией ECMA (European Computer Manufacturers Association) в спецификации ECMA-262. Расширения языка: JavaScript, JScript и ActionScript.

ECMAScript – это стандарт, а JavaScript его реализация.

У стандарта есть много версий: ES1, ES2, ES3, ES5, ES2015 (ES6), ES2016 (ES7) и т.д. каждый год.



The screenshot shows the 'Standards' section of the ECMA International website. The page title is 'Index of Ecma Standards'. The left sidebar contains links to 'Standards Index', 'Standards List', 'Withdrawn Standards', 'Tech. Reports Index', 'Tech. Reports List', and 'Withdrawn Tech.'. The main content area features a large orange header 'Index of Ecma Standards'.



The screenshot shows the 'Programming Languages' section of the ECMA International website. It features a grid of standard codes: ECMA-262, ECMA-402, ECMA-414 in the first row; ECMA-334, ECMA-335, ECMA-367, ECMA-372, ECMA-404 in the second row; and ECMA-408 in the third row. Below the grid, there are sections for 'Data Presentation' and 'Character Sets and Coding'.

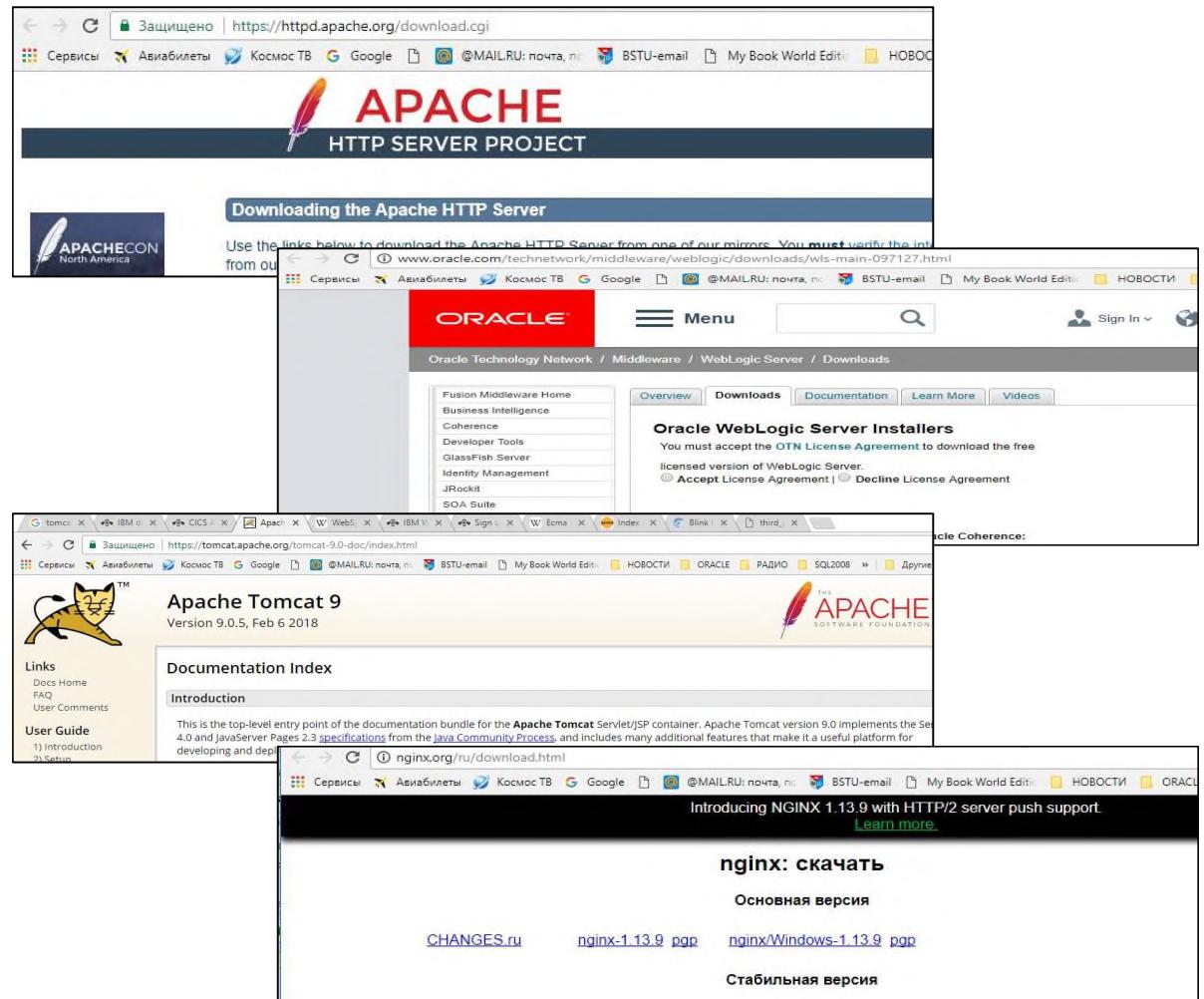
Web-engine (web-движок)

Это программа, преобразующая содержимое html-страниц во внутреннее представление web-браузера в соответствии с моделью DOM.

Движок	Разработчик(и)	Лицензия	Браузер	Язык программирования
Blink	Google, Opera, Samsung, Intel, others ^[1]	GNU LGPL, BSD-style	Google Chrome & Opera >15.0	C++
EdgeHTML	Microsoft	Проприетарная	Microsoft Edge	C++
Gecko	Netscape/Mozilla Foundation	MPL / LGPL / GPL	Mozilla Firefox	C++
GtkHTML	GNOME	GNU LGPL	Novell Evolution	C
KHTML	KDE	GNU LGPL	Konqueror	C++
Presto	Opera Software	Проприетарная	Opera	C++ ^[2]
Prince XML	YesLogic Pty Ltd	Проприетарная	Prince XML	Mercury
Robin	Ritlabs	Проприетарная	The Bat!	Delphi
Tasman	Microsoft	Проприетарная	Microsoft Entourage	—
Trident	Microsoft	Проприетарная	Internet Explorer	C++ ^[3]
WebKit	KDE, Apple, Nokia, Google, RIM, Palm и другие.	GNU LGPL, BSD	Google Chrome, Safari	C++
XEP	RenderX	Проприетарная	XEP	Java

Web-серверы

IIS (Microsoft), Apache (Apache Software Foundation), WebSphere (IBM), WebLogic (Oracle), Glassfish (Oracle), Apache Tomcat (Apache Software Foundation), nginx (И.Сысоев),...



ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

ОСНОВЫ ПОСТРОЕНИЯ WEB-ПРИЛОЖЕНИЙ

Общие принципы построения web-приложений

- web-ресурсы приложения;
- запросы и ответы;
- соединение (сессия);
- конфигурационный файл приложения;
- контекст приложения;
- фильтры;
- кэш (данных и вывода);
- слушатели событий;
- принципы безопасности.

Web-ресурс =

сущность, расположенная на стороне сервера и имеющая URL/URI, к которой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

Виды web-ресурсов

- 1) статические – **отправляются** клиенту **без изменения** (html-страницы, рисунки, видео-файлы, ...),
- 2) динамические – **динамически** (программно) **формируются** на сервере и отправляются клиенту (сервлеты, JSP, http-обработчики, aspx-страницы,...).

Ресурс может быть статическим относительно сервера и динамическим относительно клиента (html-страницы с JS).

Запрос (Request)

=

серверный объект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки.

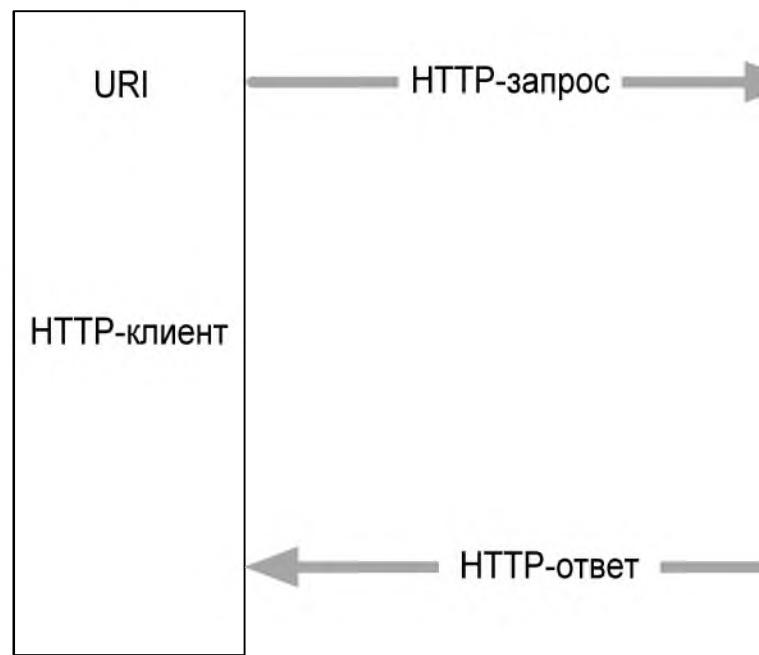
Содержит всю информацию из http-запроса: метод, коллекция заголовков, коллекция параметров, поток данных ...

Обычно объект Request предоставляет возможность хранить данные в формате ключ/значение.

Ответ
(Response) =

серверный объект, который автоматически формируется сервером **при получении http-запроса** (одновременно с объектом Request), **заполняется данными** серверным программным кодом, **преобразуется в http-ответ** и **отправляется клиенту**.

Содержит всю информацию, которая должна быть помещена в http-ответ: статус, коллекция заголовков, поток данных, ...



HTTP-сервер

Request

Response

Ресурс - программный код

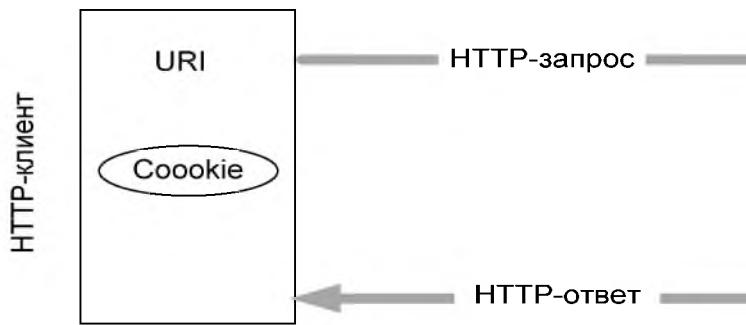
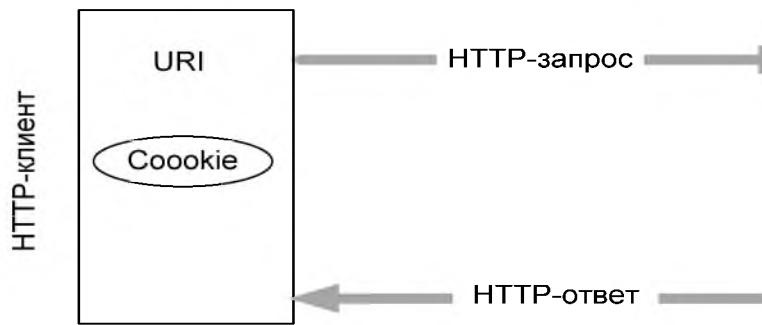
Response

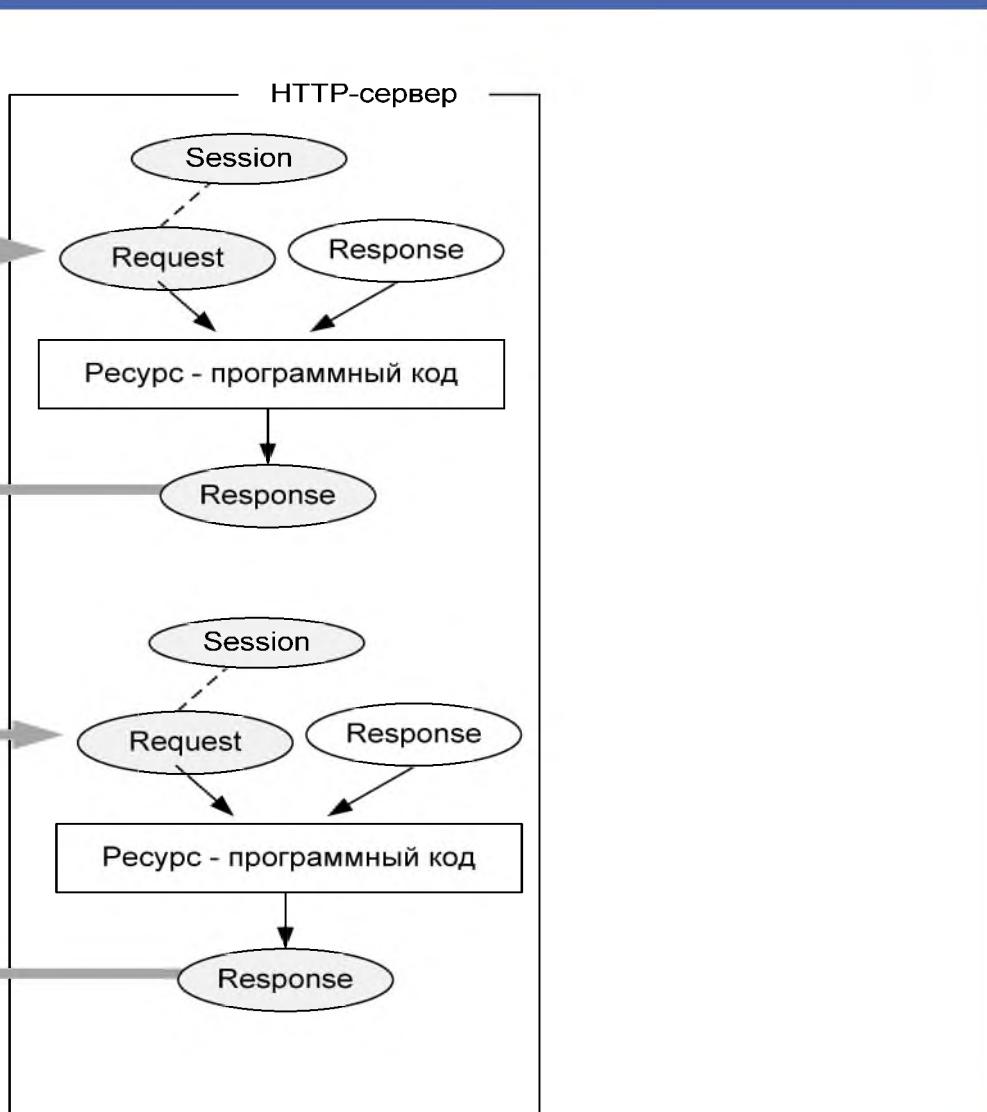
Сессия (Session)

серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении.

Время жизни: **timeout** (системный параметр, обычно равен 10–30 минутам) – **максимальное время между запросами клиента**. Если **timeout** превышен, то **Session** разрушается и при следующем запросе создается новый экземпляр. **Каждая сессия имеет собственный идентификатор** (Session ID, 16 или более байт). **Каждый Request принадлежит какой-то сессии** (имеет ссылку на объект **Session** или содержит **Session ID**).

Обычно объект **Session** предоставляет приложению возможность хранить данные в формате **ключ/значение**.





Cookie

=

фрагмент данных,
отправленный web-сервером и
хранимый web-клиентом.

Используется для
аутентификации, хранения
пользовательских
предпочтений, статистики,
информации о сеансе (обычно
Session ID). Обычно имеет имя,
содержащее URL, может иметь
срок действия.

Для создания и пересылки
Cookie применяются заголовки.

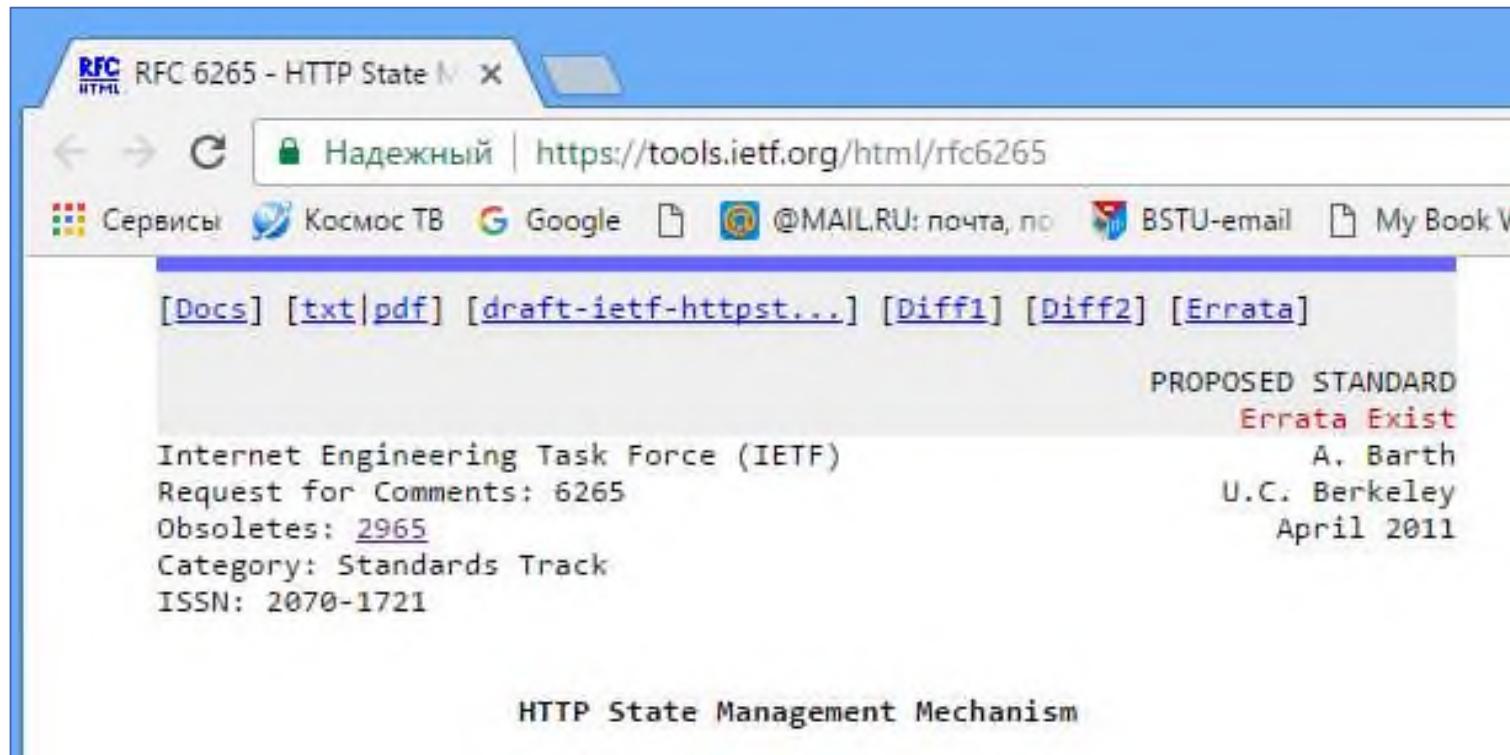
```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html; charset=utf-8
Vary: Accept-Encoding
Server: Microsoft-IIS/8.5
X-AspNetMvc-Version: 5.2
X-AspNet-Version: 4.0.30319
Set-Cookie: ASP.NET_SessionId=imdp40w4hcs55g3jto0aa04b; path=/; HttpOnly
X-Powered-By: ASP.NET
Date: Tue, 14 Feb 2017 21:06:16 GMT
Transfer-Encoding: chunked
```

Заголовок ответа Set-Cookie используется **для отправки файла cookie с сервера клиенту** (браузеру), чтобы клиент мог отправить его обратно на сервер позже.

Заголовок запроса Cookie содержит ранее сохраненные файлы cookie, связанные с сервером.

```
GET http://80.94.166.212:40001/EGHRGE/Forecast HTTP/1.1
Host: 80.94.166.212:40001
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Referer: http://80.94.166.212:40001/EGHRGE
Accept-Encoding: gzip, deflate, sdch
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: ASP.NET_SessionId=imdp40w4hcs55g3jto0aa04b
```

RFC 6265 (стандарт, описывающий Cookie)



The screenshot shows a web browser window with the following details:

- Tab:** RFC 6265 - HTTP State
- Address Bar:** Надежный | <https://tools.ietf.org/html/rfc6265>
- Toolbar:** Сервисы, Космос ТВ, Google, @MAIL.RU: почта, по, BSTU-email, My Book
- Content Area:**
 - [\[Docs\]](#) [\[txt|pdf\]](#) [\[draft-ietf-https...\]](#) [\[Diff1\]](#) [\[Diff2\]](#) [\[Errata\]](#)
 - PROPOSED STANDARD**
 - Errata Exist**
 - Internet Engineering Task Force (IETF)
 - Request for Comments: 6265
 - Obsoletes: [2965](#)
 - Category: Standards Track
 - ISSN: 2070-1721
 - A. Barth
U.C. Berkeley
April 2011
 - HTTP State Management Mechanism**

Конфигурационный
файл web-
приложения =

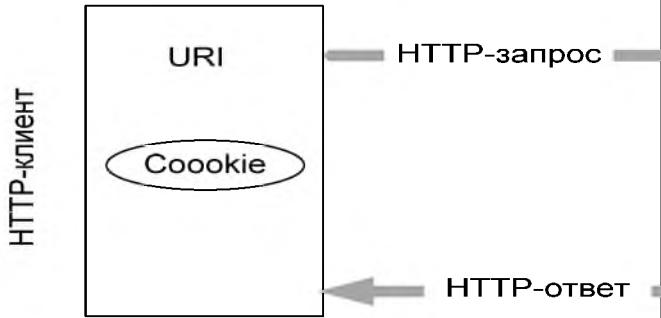
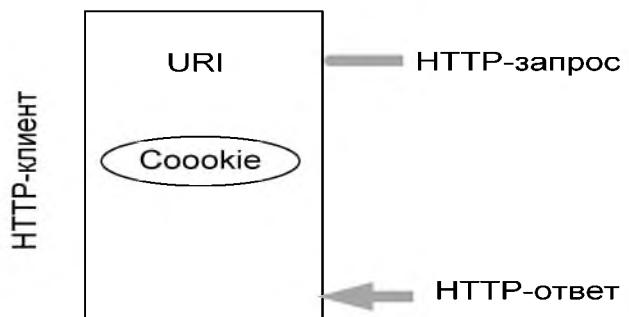
файл содержащий **системные**
параметры приложения, обычно
в XML-формате, служит **основой**
для создания контекста web-
приложения.

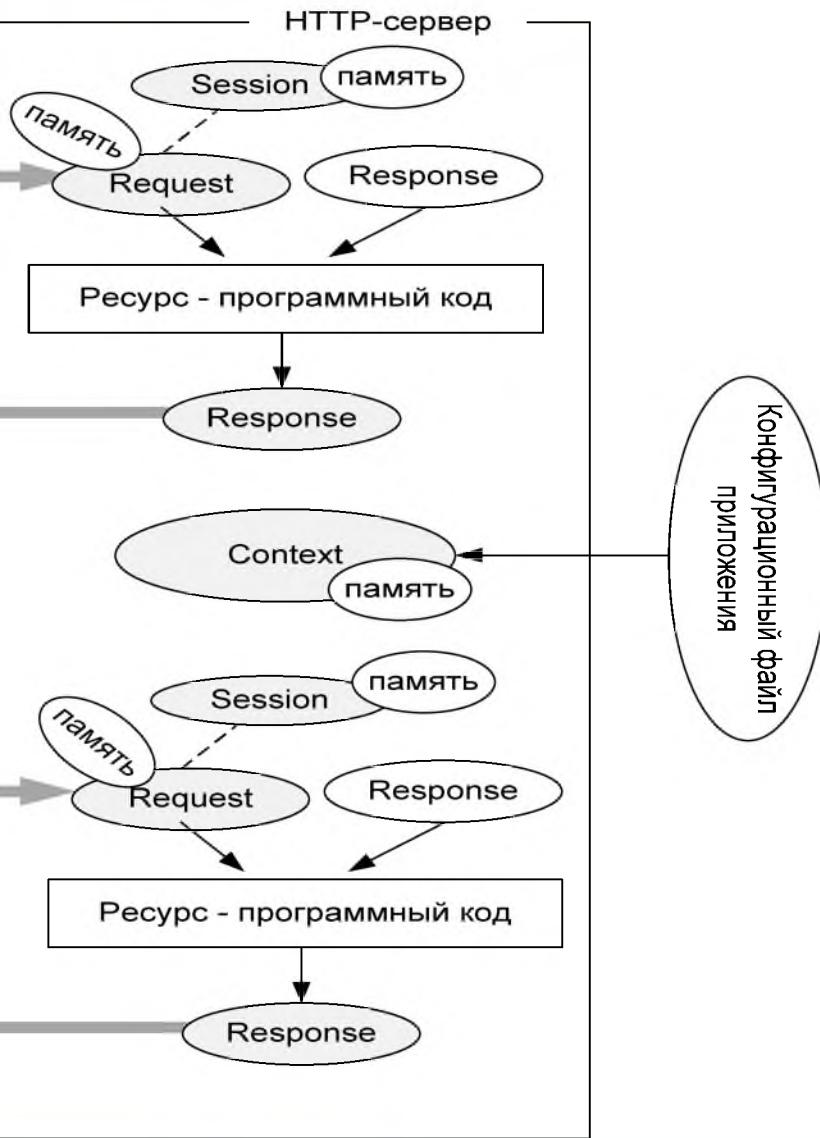
Контекст web-
приложения =

серверный объект,
предназначенный для хранения
информации об одном web-
приложении.

Как правило, формируется сразу
при загрузке web-сервера,
основные данные (параметры
приложения) копируются из
конфигурационного файла
приложения, общий для всех
сессий приложения.

Обычно контекст предоставляет
возможность хранить данные в
формате ключ/значение.





Фильтр (Filter)

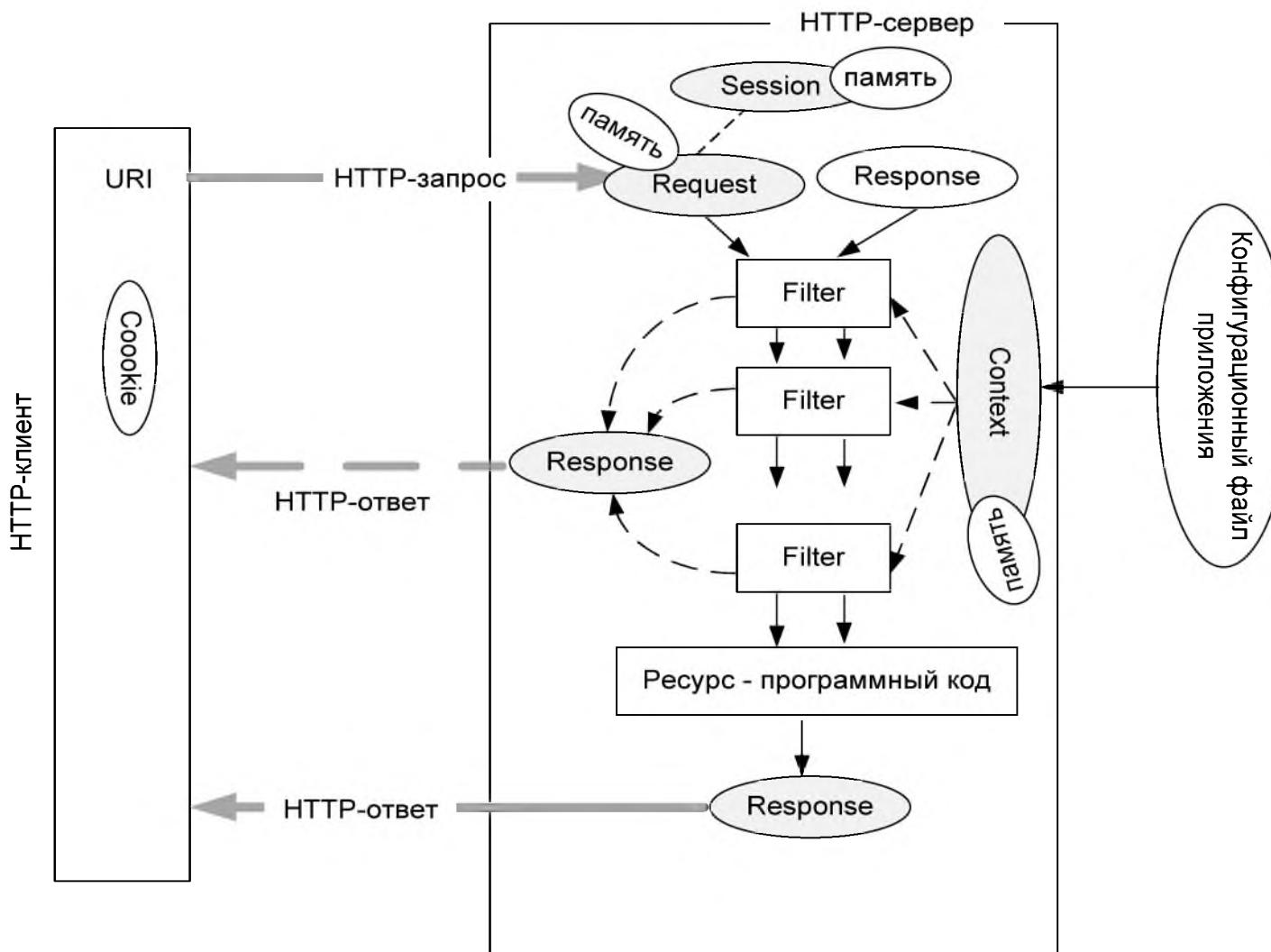
=

серверный объект – препроцессор запроса, предназначен для предварительной обработки объекта Request.

К одному ресурсу может быть построена цепочка фильтров, последний в цепочке – ресурс. Фильтр может прервать цепочку и сам сформировать ответ клиенту.

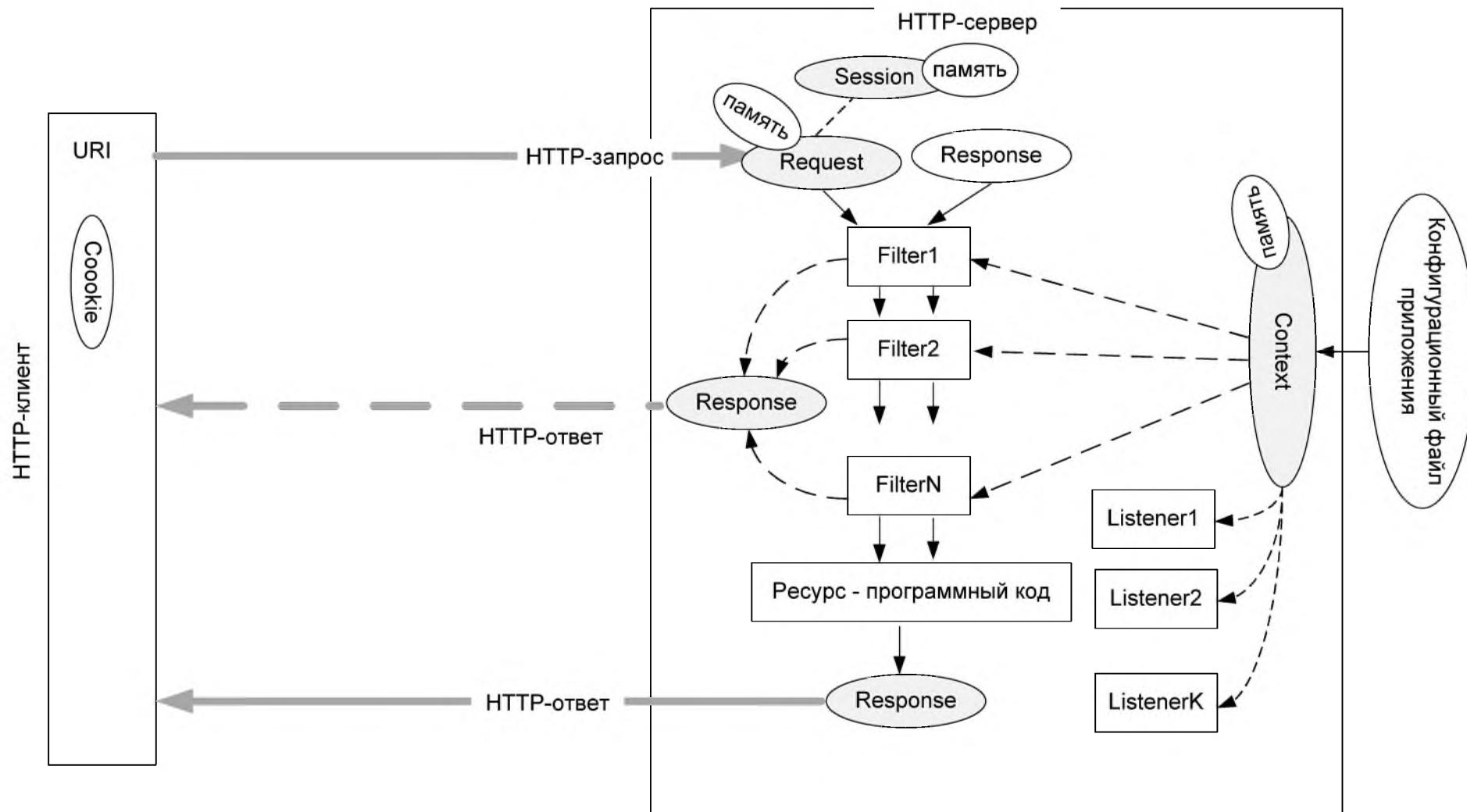
Один и тот же фильтр может быть применен к нескольким ресурсам.

В качестве параметров фильтр получает объекты Request и Response, которые от передает дальше по цепочке или обрывает цепочку и заполняет объект Response.



Слушатели =
событий
(Listener)

серверные объекты – для
обработки **событий жизненного**
цикла web-приложения.



Кэш
(Cache)

=

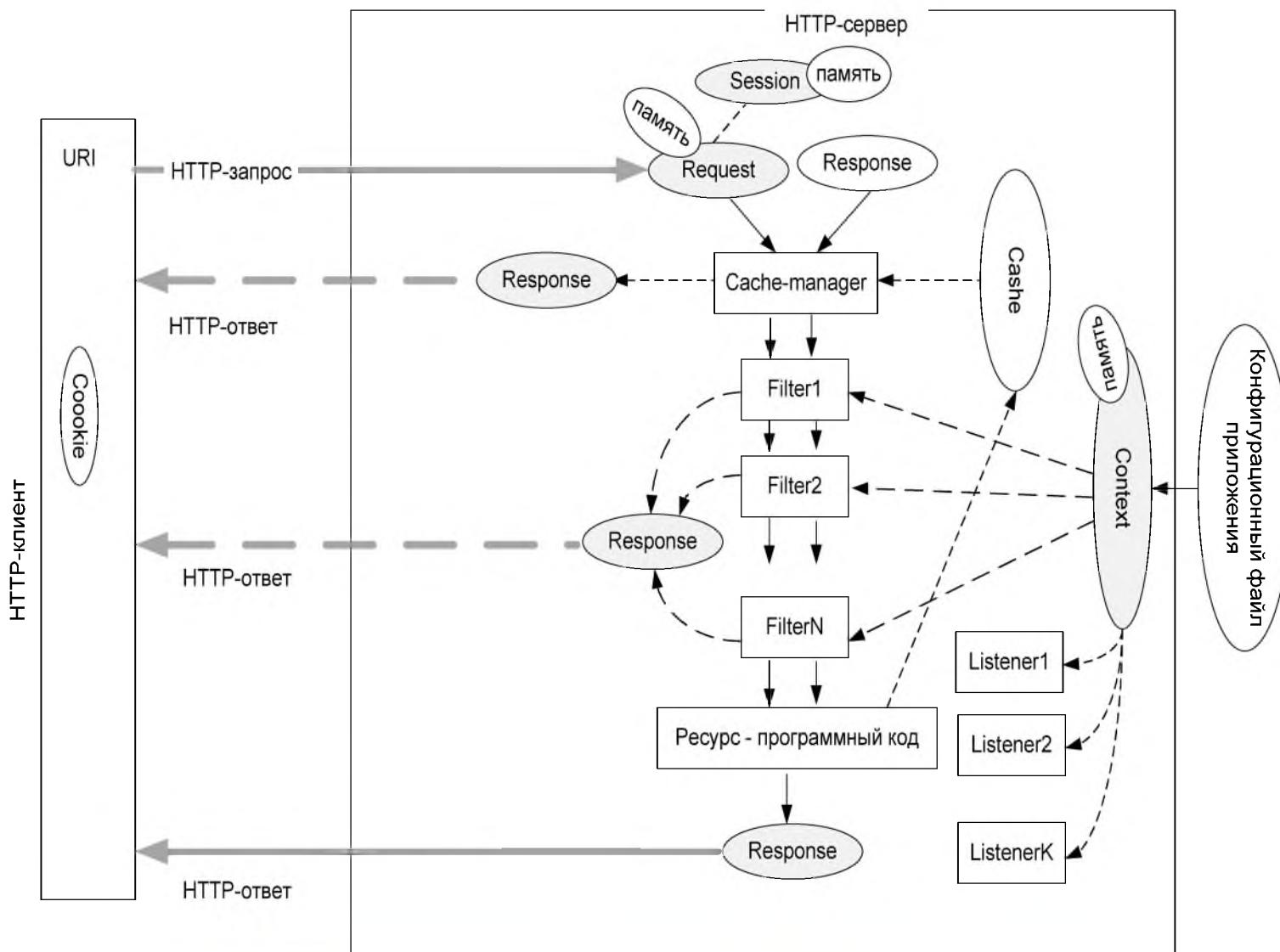
серверный объект, предназначенный для временного хранения данных с целью ускорения выполнения запроса.

Кэширование – процессы записи и извлечения данных в/из Cache.

Различают кэширование данных и кэширование вывода.

Кэширование данных – кэширование часто используемых данных.

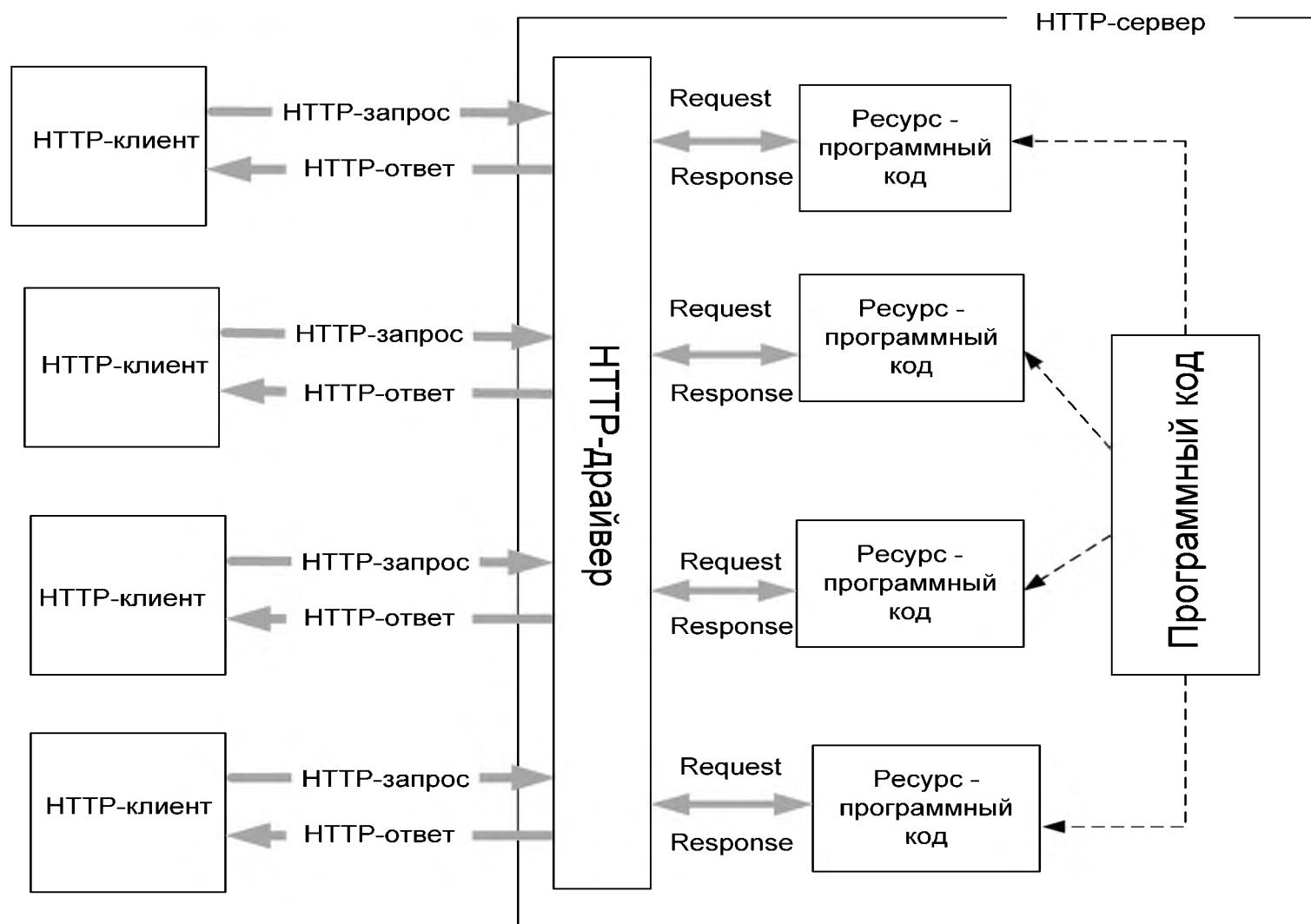
Кэширование вывода – кэширование объекта Response.



Ресурс – программный = код

экземпляр приложения, который создается для обработки каждого запроса.

HTTP-драйвер преобразует последовательность битов HTTP-запроса в объект Request.



Постоянное соединение =

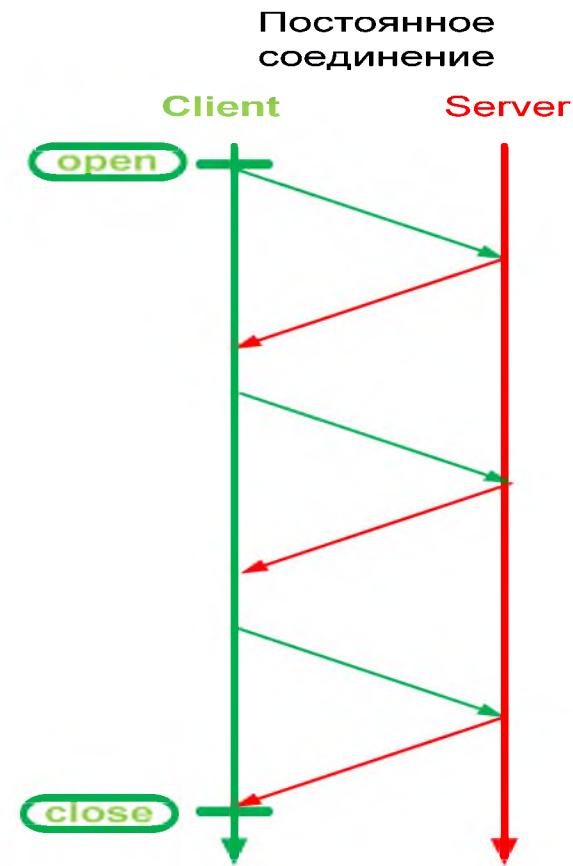
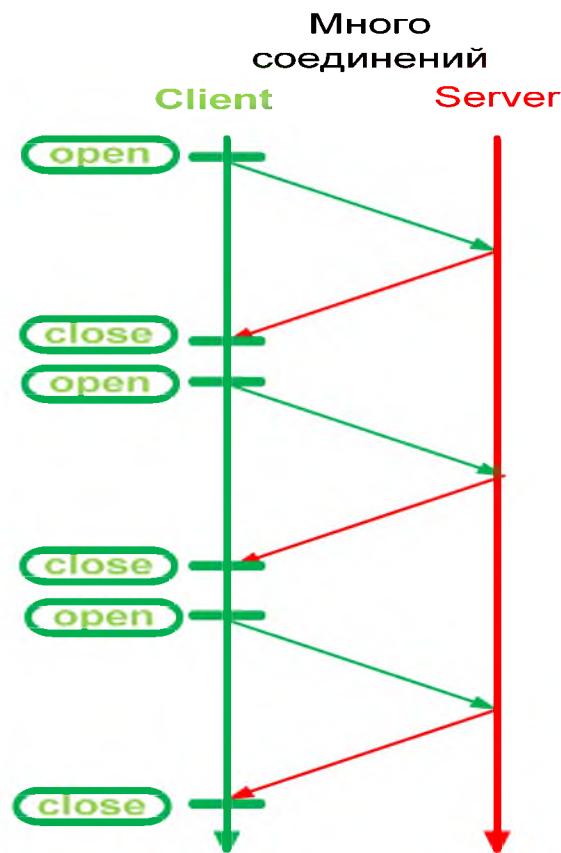
использование **одного TCP-соединения для многократных пар запрос-ответ** вместо последовательного открытия новых соединений для каждой пары запрос-ответ.

Клиент может запросить постоянное соединение с помощью **общего заголовка Connection: Keep-Alive**, сервер подтверждает заголовком Connection: Keep-Alive.

Некоторые серверы открывают постоянное соединение по умолчанию и закрывают через заданное время (timeout) бездействия браузера. Некоторые браузеры по умолчанию открывают постоянные соединения.

Серверы, как правило, поддерживают ограниченное количество постоянных соединений. Обычно по умолчанию используется в HTTPS.

Постоянное соединение

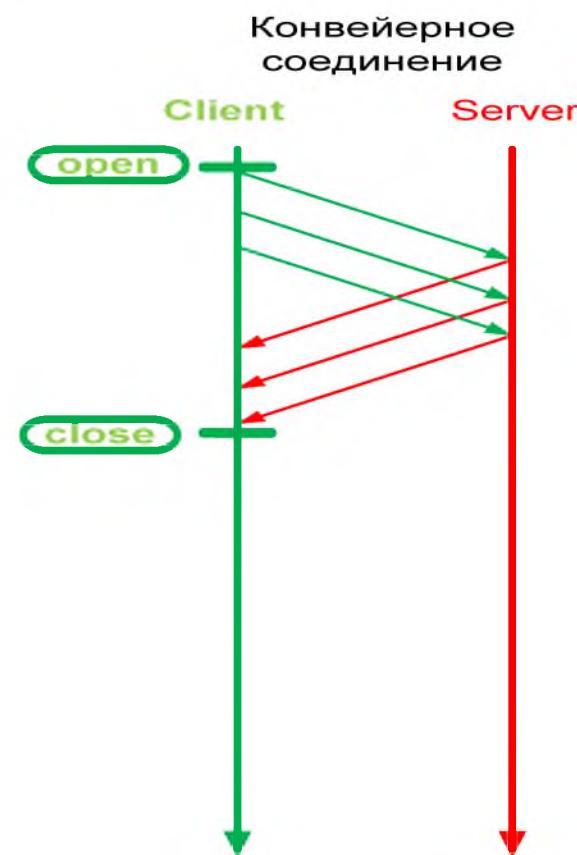
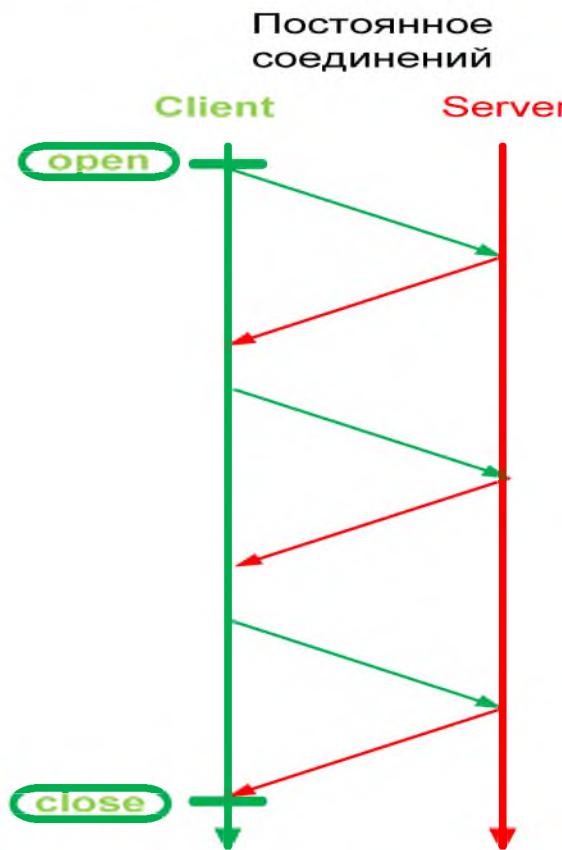


Конвойерная обработка = (HTTP pipelining)

технология, которая позволяет **в одном соединении передавать несколько различных пар запрос-ответ.**

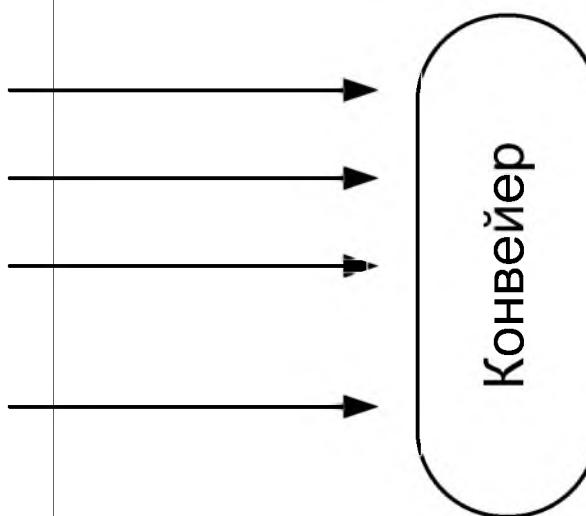
Эффективно, если на одной html-странице запрашивается несколько ресурсов с одного сервера.
В основном поддерживается браузерами мобильных систем.

Конвейерная обработка



Конвейерная обработка

```
<html>
.....
<link href="http://server.../css.css" .../>
.....
<script src="http://server.../js.js" ...></script>
.....
<img src = "http://server.../img.png" .... />
.....
<form>
    <input type="submit" .... />
</form>
.....
</html>
```



Пул соединений с = базой данных

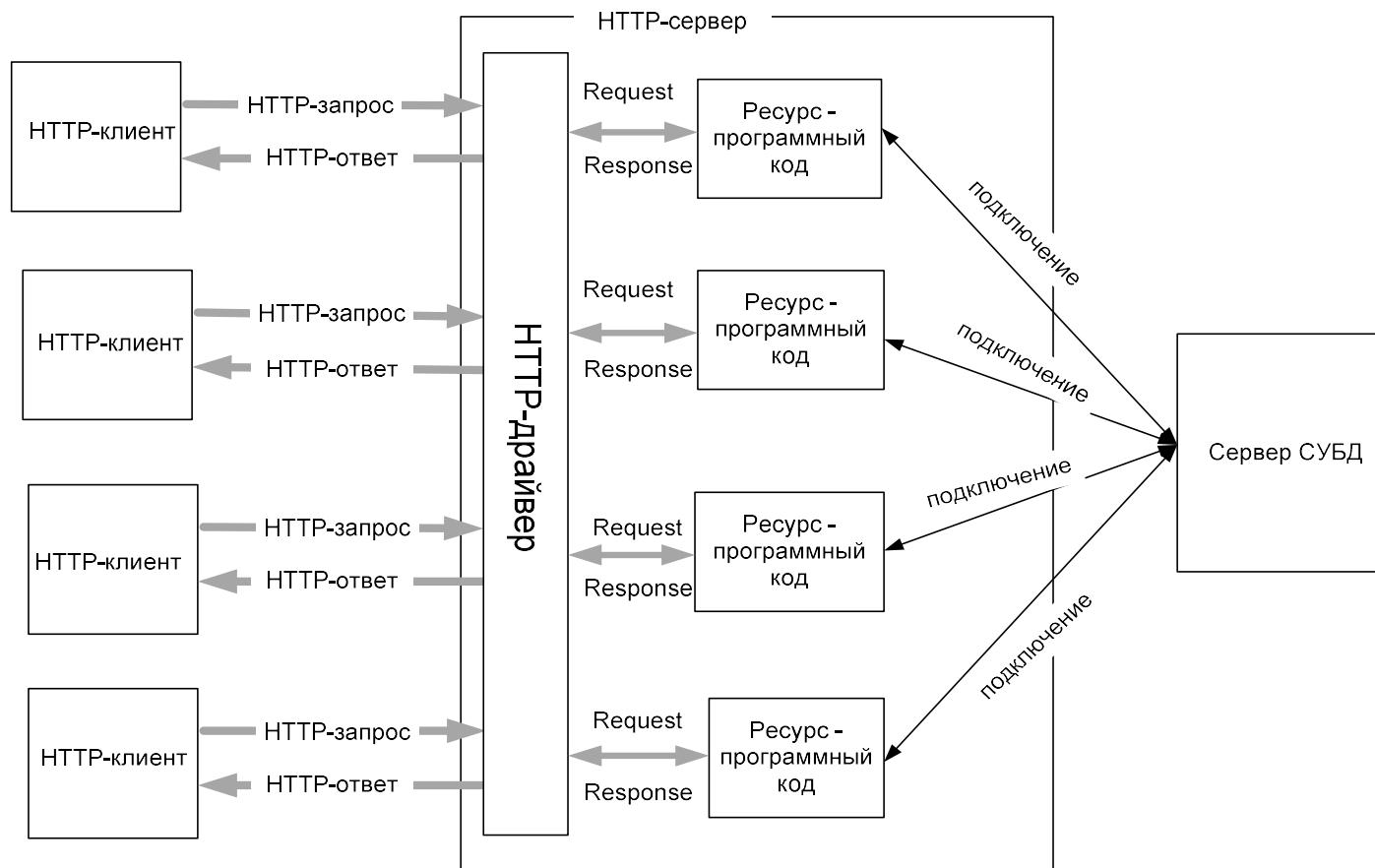
несколько предварительно и **постоянно открытых соединений с сервером СУБД**, которые используют приложения.

Если все подключения пула заняты, запрос на соединение ставится в очередь.

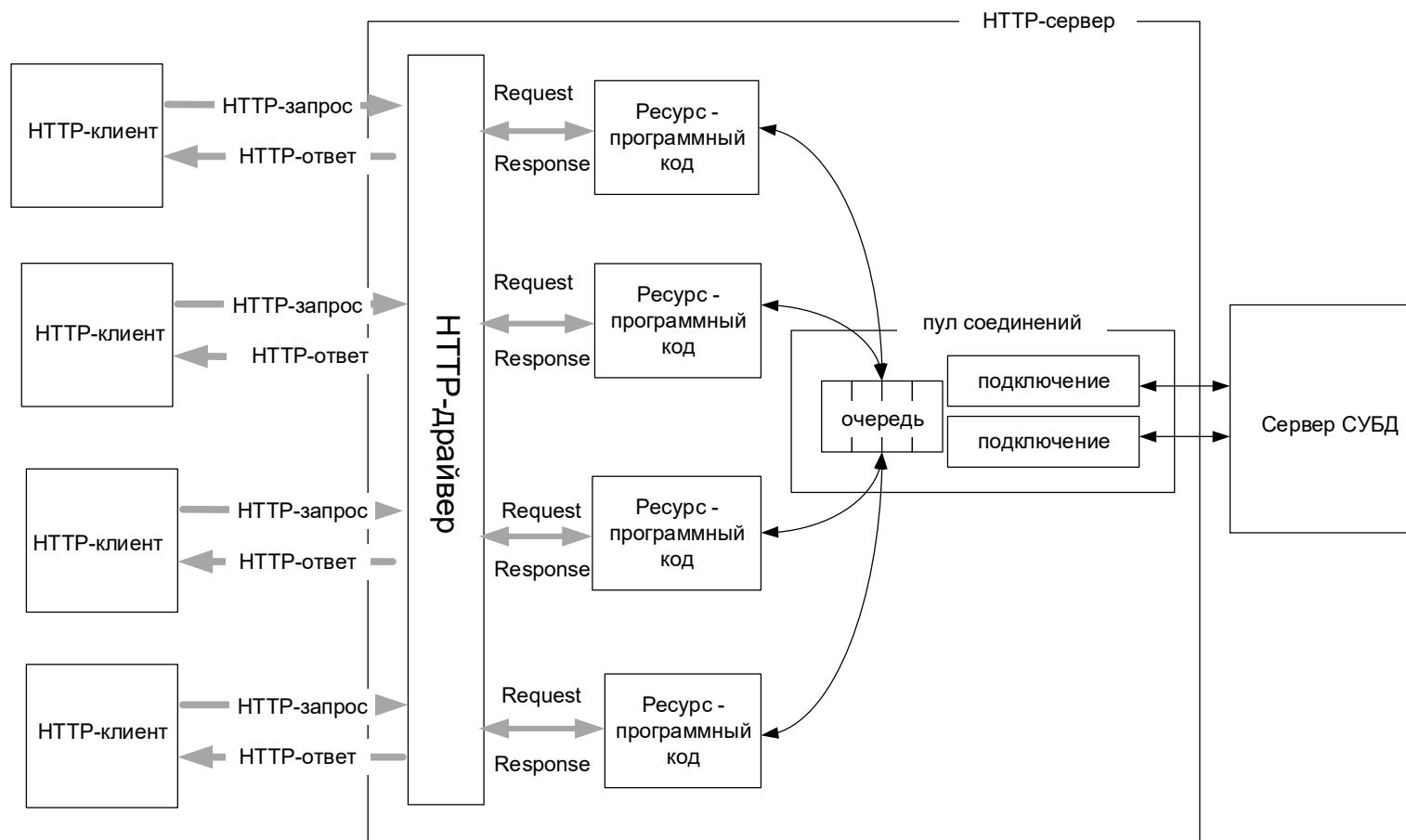
Применение пула позволяет увеличить производительность за счет **отсутствия процесса подключения к серверу**.

Альтернатива: держать постоянное открытое соединение для каждого подключения (в общем случае для web-приложения неприемлемо) или открывать/закрывать соединение при каждом запросе (большие накладные расходы на установку соединения).

Схема работы с базой данных без пула



Пул соединения с базой данных



ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

ВВЕДЕНИЕ В NODE.JS

Node.js

=

программная платформа для разработки преимущественно серверных приложений на языке JS.



Основные свойства

- основан на **Chrome V8**;
- **среда** (контейнер) **исполнения** приложений на JavaScript;
- **однопоточный** (код приложения исполняется только в одном потоке, один стек вызовов);
- ориентирован на **события**;
- поддерживает механизм **асинхронности**;
- **не блокирует** выполнение кода при вводе/выводе (в файловой системе до 4-ёх одновременно).
- в состав Node.js входят инструменты: **npm** – пакетный менеджер; **gyp** - Python-генератор проектов; **gtest** – Google фреймворк для тестирования C++ приложений;
- основная сфера применения: **разработка web-серверов**;
- **версионирование**: две ветки 18.x.x – версии длительной поддержки (LTS, Long Term Support), 20.x.x – нестабильные версии, включающие последние разработки (Current);
- **документация**: <https://nodejs.org/api/>
- **Stability Index** - Deprecated (0), Experimental (1), Stable (2), Locked (3)

18.17.1 LTS

Рекомендовано для большинства

20.6.1 Текущая

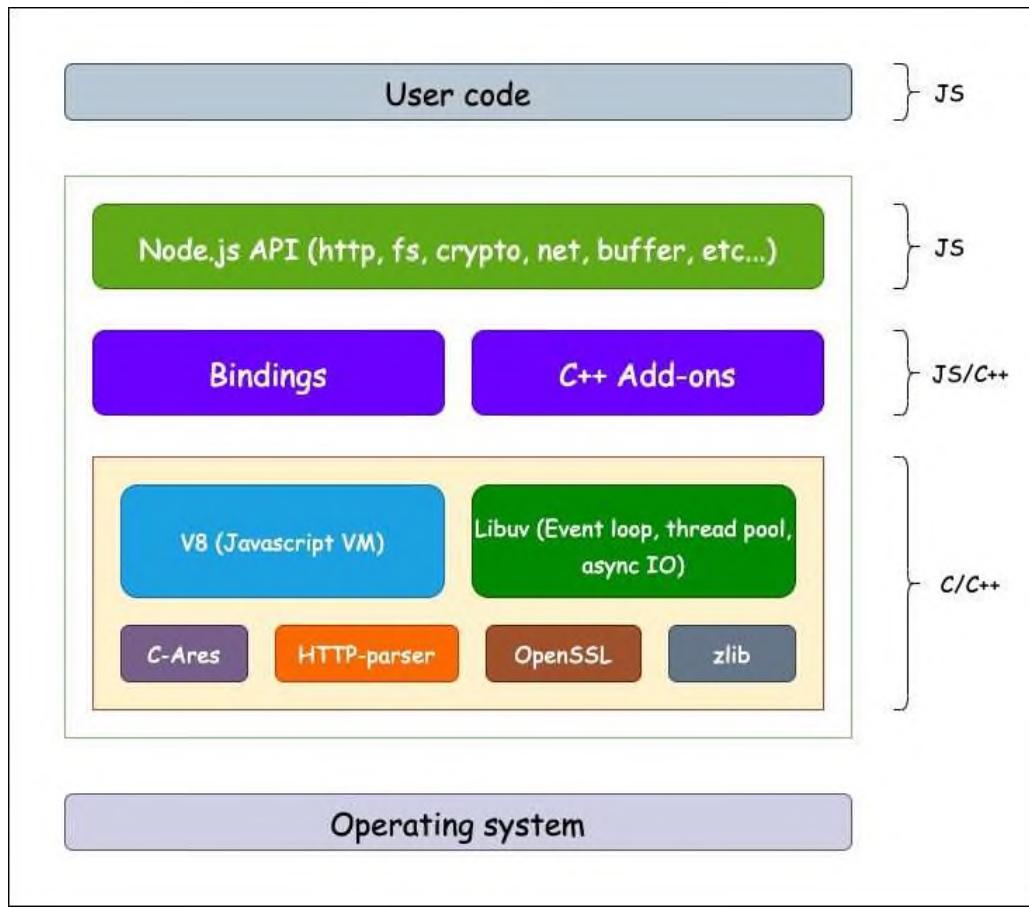
Новые возможности

Stability: 2 - Stable

Stability: 1 - Experimental

Stability: 0 - Deprecated: Use `buf.buffer` instead.

Архитектура Node.js



Использует библиотеки:

V8 – библиотека V8 Engine;

libuv – библиотека для абстрагирования неблокирующих операций ввода/вывода (представляет собой обертку над *epoll*, *kqueue*, *IOCP*);

llhttp – легковесный парсер http-сообщений (написан на С и не выполняет никаких системных вызовов);

c-ares – библиотека для работы с DNS;

OpenSSL – библиотека для криптографии;

zlib – сжатие и распаковка.

История Node.js

2009: Первая версия Node.js, создание первого варианта прт

2010: Express, Socket.io, поддержка на Heroku

2011: NPM 1.0, большие компании (LinkedIn и Uber), начали пользоваться Node.js

2012: Стабильная версия (0.8.x), быстрый рост популярности

2013: MEAN-стэк, Коя

2014: Netflix, io.js (форк Node.js, цель создания – внедрение поддержки ES6 и ускорение развития платформы)

2015: Слияние IO.js и Node.js в 4.x.x

2016: Yarn, Node.js 6

2017: Node.js 8, HTTP/2

2018: Node.js 10, ES-модули, worker_threads

2019: OpenJS Foundation, Node.js 12,13

2020: Node.js 14,15

2021: Node.js 16,17

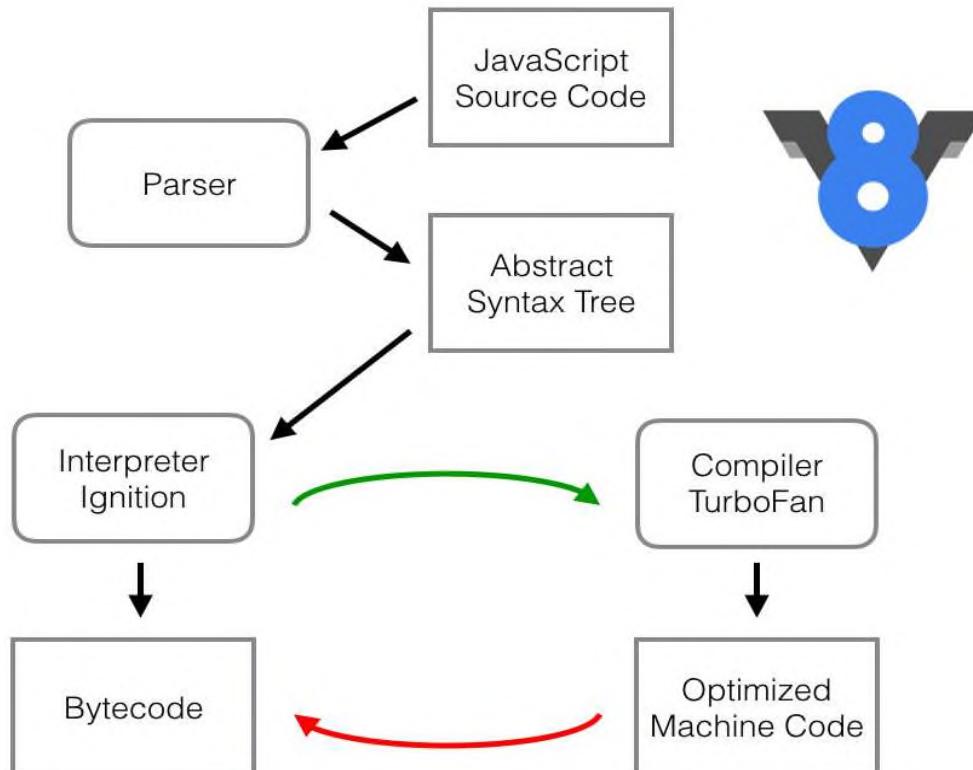
2022: Node.js 18,19

2023: Node.js 20



Райан Дал
Разработчик Node.js

Принцип работы V8

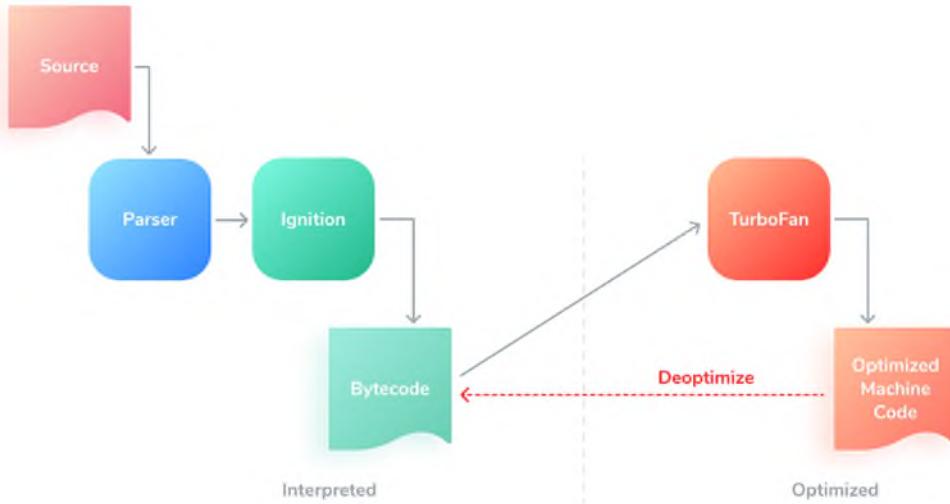


Ignition создает **байт-код**. В этот момент движок запускает код и собирает отзывы о типах. Чтобы он работал быстрее, байт-код может быть отправлен компилятору **TurboFun** вместе с данными обратной связи. Оптимизирующий компилятор делает на его основе определенные предположения, а затем **создает высокооптимизированный машинный код**. Если в какой-то момент одно из предположений оказывается неверным, **оптимизирующий компилятор деоптимизируется** и возвращается к интерпретатору.

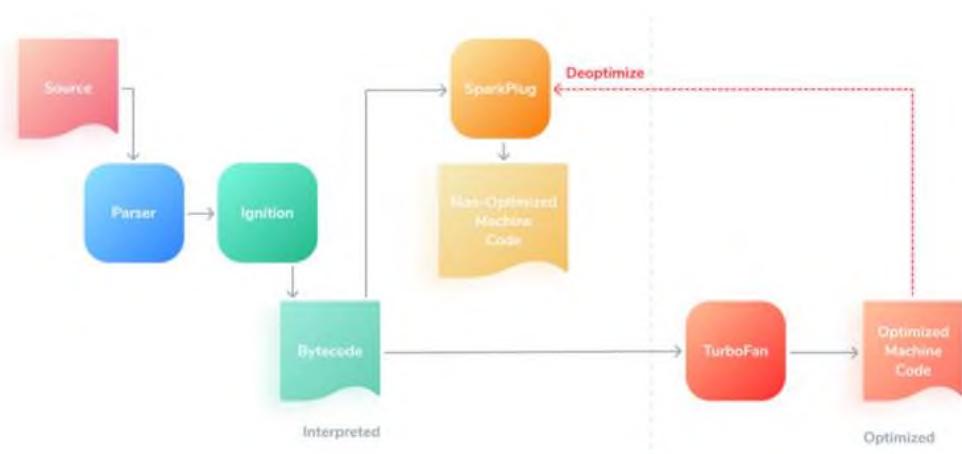
2021: **новый неоптимизирующий компилятор Sparkplug** (работает быстрее за счет компиляции не исходного кода, а байт кода + не генерирует IR (intermediate representation)).



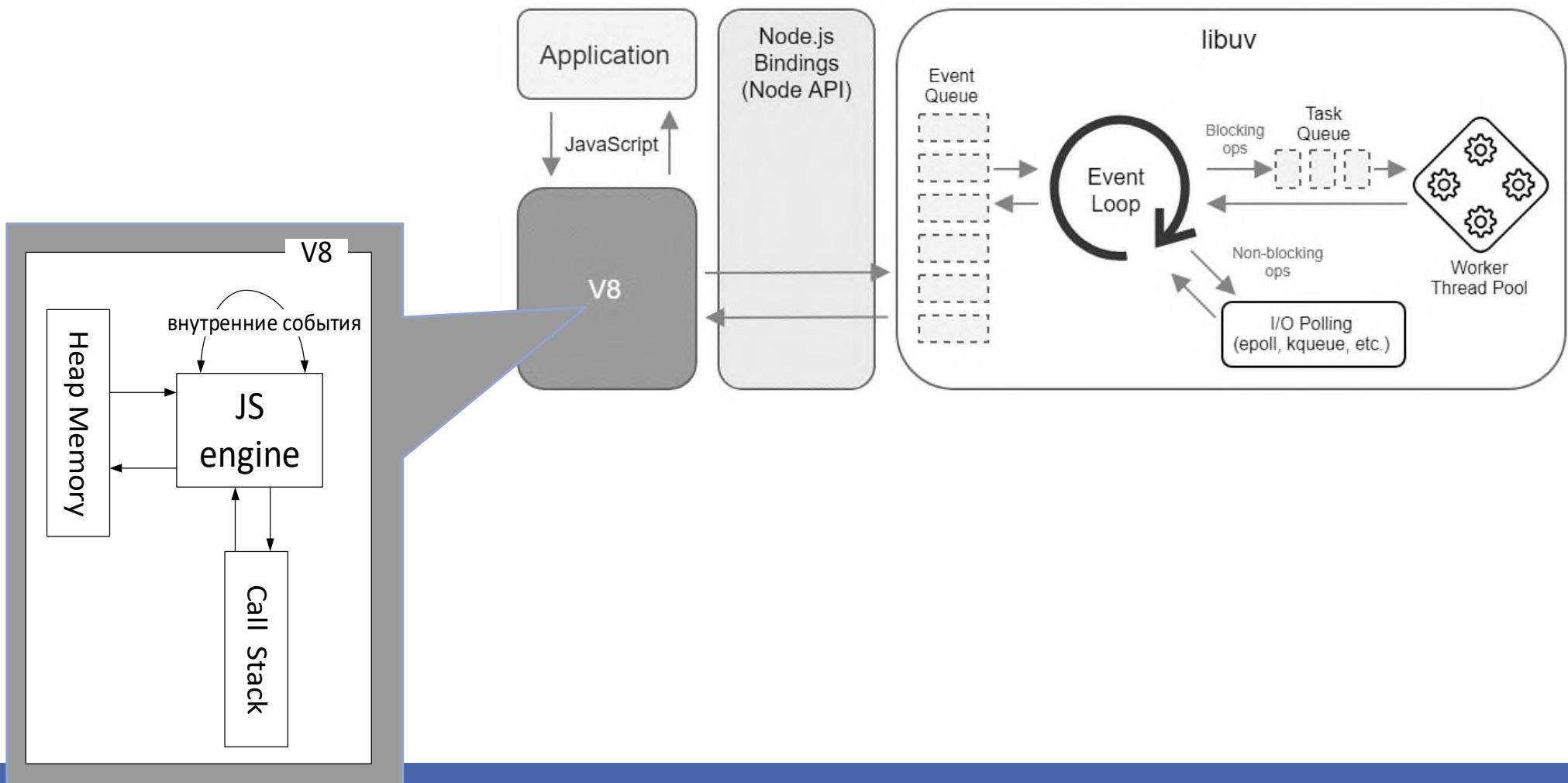
Было



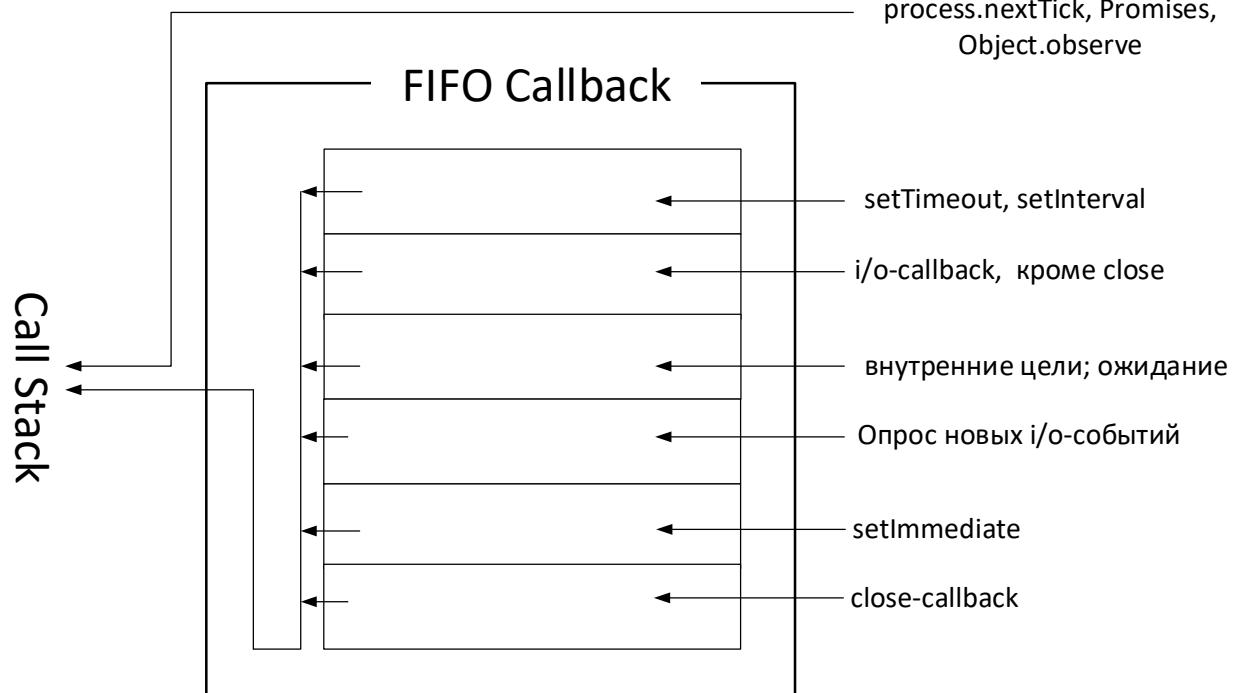
Стало



Принцип работы Node.js

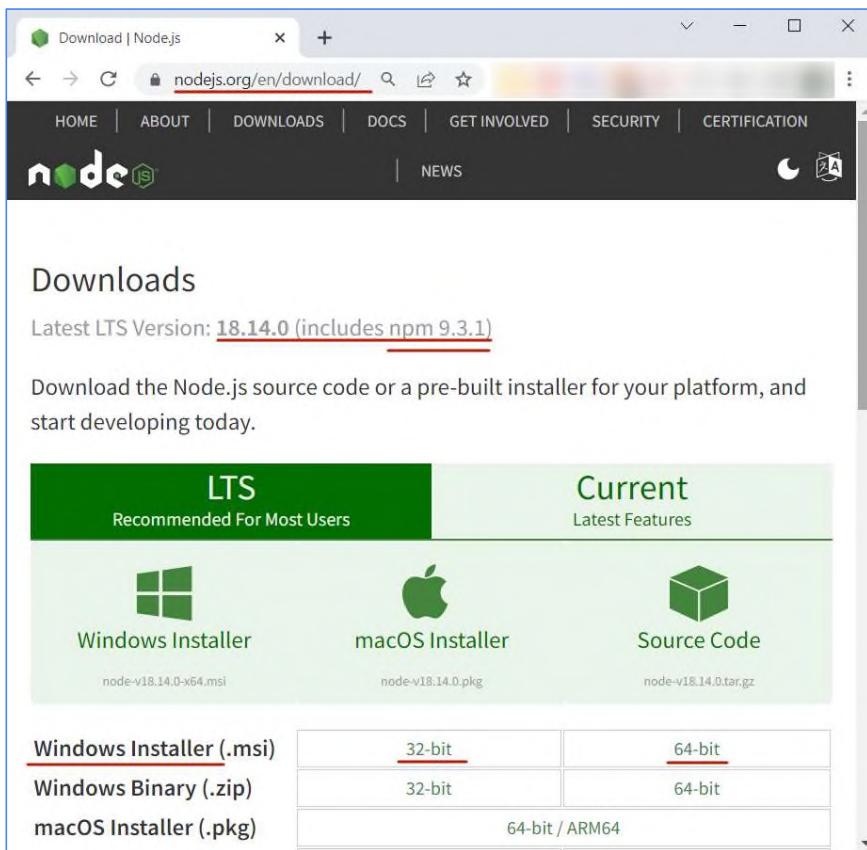


Обзор фаз event loop'a



- **макрозадачи** – выполняются по одной за один проход цикла (*setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering*);
- **микрозадачи** – на каждом проходе цикл выполняет все накопившееся (*process.nextTick, Object.observe, Promises*).

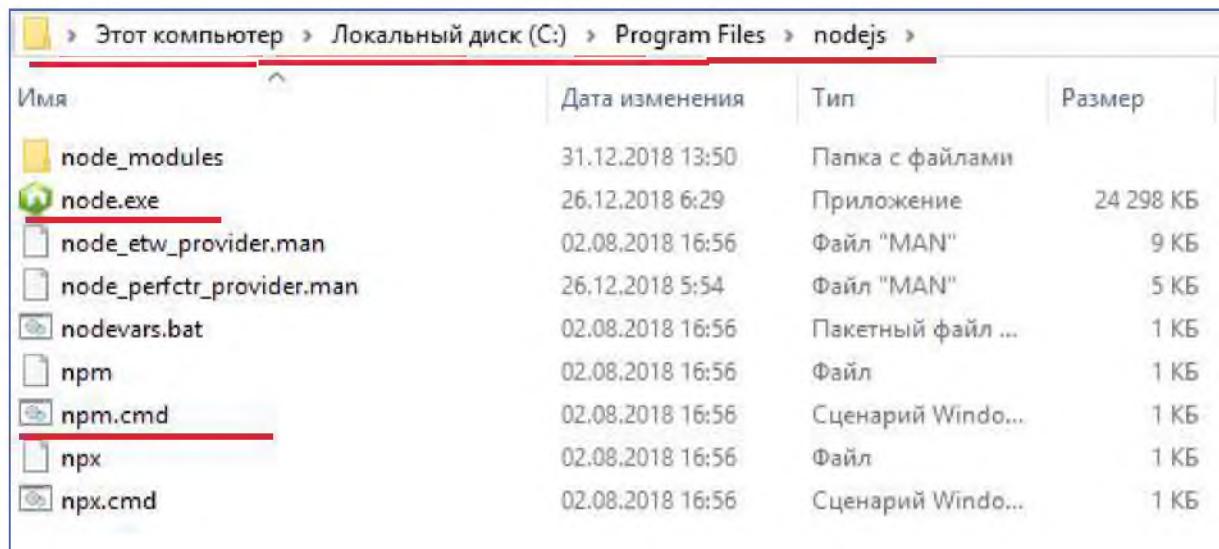
Установка Node.js



<https://nodejs.org/en/download/>



Проверка



Имя	Дата изменения	Тип	Размер
node_modules	31.12.2018 13:50	Папка с файлами	
node.exe	26.12.2018 6:29	Приложение	24 298 КБ
node_etw_provider.man	02.08.2018 16:56	Файл "MAN"	9 КБ
node_perfctr_provider.man	26.12.2018 5:54	Файл "MAN"	5 КБ
nodevars.bat	02.08.2018 16:56	Пакетный файл ...	1 КБ
npm	02.08.2018 16:56	Файл	1 КБ
npm.cmd	02.08.2018 16:56	Сценарий Windo...	1 КБ
npx	02.08.2018 16:56	Файл	1 КБ
npx.cmd	02.08.2018 16:56	Сценарий Windo...	1 КБ

Администратор: Командная строка

```
C:\Users\Win10_ISiT_Server>npm --version  
6.4.1
```

```
C:\Users\Win10_ISiT_Server>node --version  
v10.15.0
```

```
C:\Users\Win10_ISiT_Server>
```

Инструменты

- Visual Studio Code
- WebStorm
- Visual Studio
- Atom
- Sublime Text
- Brackets
- Браузер
- Postman
- Insomnia

Разработка простейшего HTTP-сервера и его запуск

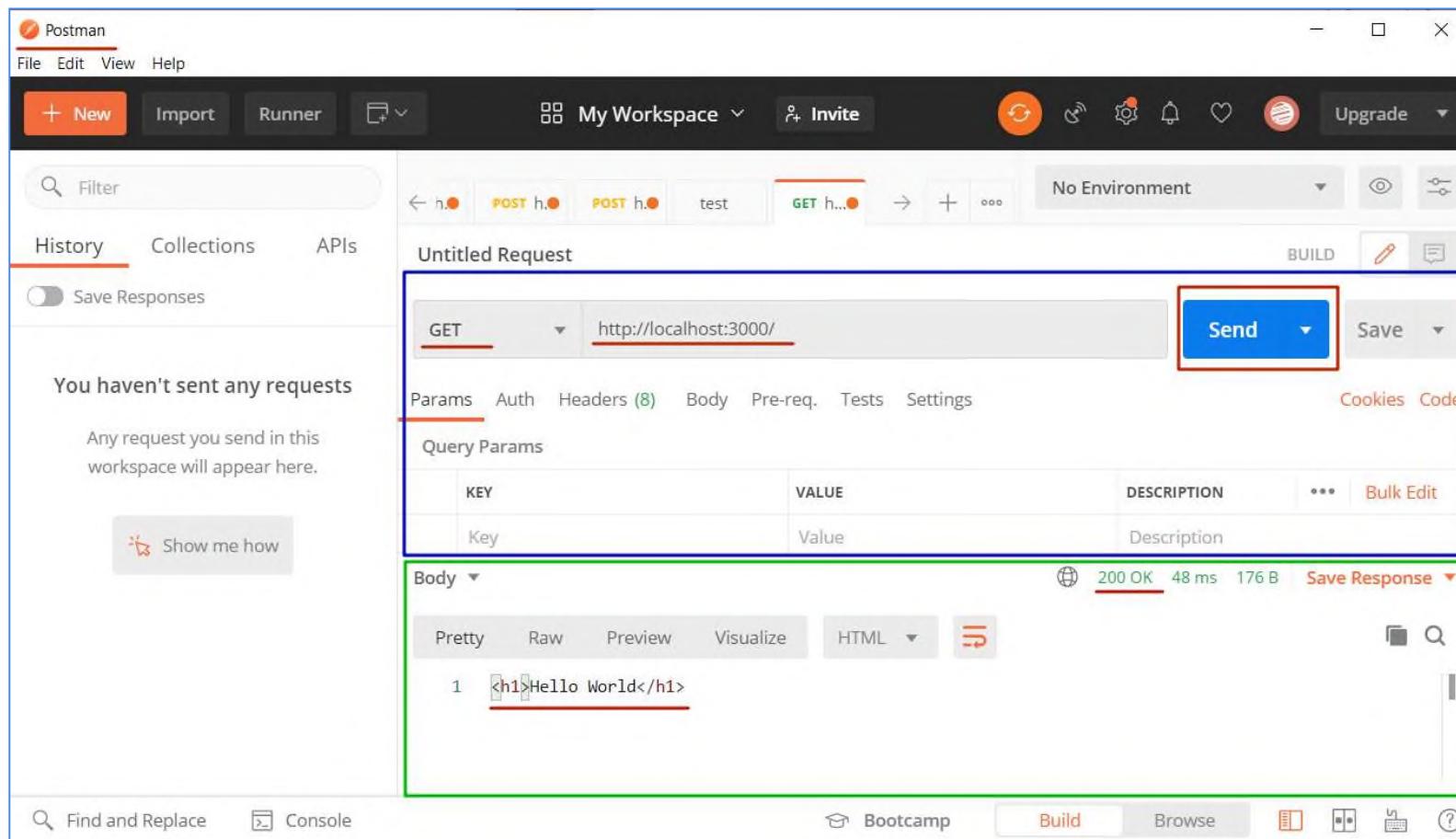
```
const http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, { 'Content-Type': 'text/html' });
  response.end('<h1>Hello World</h1>');
}).listen(3000, () => { console.log('Server running at http://localhost:3000/') });

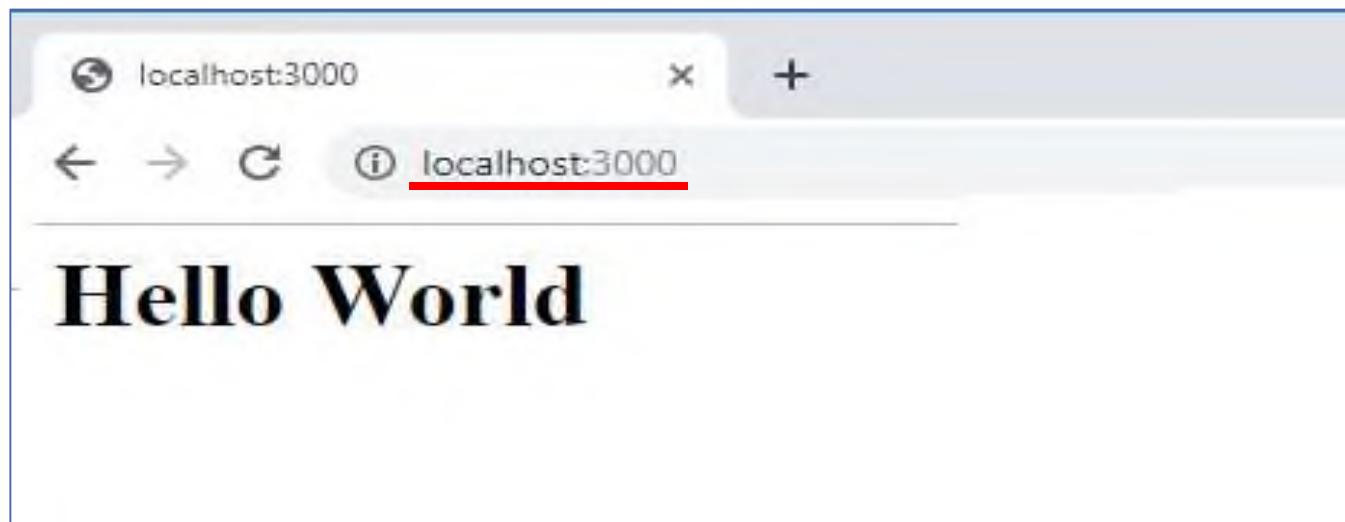

```

```
PS D:\NodeJS\samples\cwp_01> node .\02-01.js
Server running at http://localhost:3000/
|
```

Демонстрация работы (Postman)



Демонстрация работы (браузер)



Исследование запроса

```
const http = require('http');

let h = (r) => {
  let rc = '';
  for (key in r.headers) rc += '<h3>' + key + ':' + r.headers[key] + '</h3>';
  return rc;
}

http.createServer(function (request, response) {
  let b = '';
  request.on('data', str => { b += str; console.log('data', b); })
  response.writeHead(200, { 'Content-Type': 'text/html;charset=utf-8' });
  request.on('end', () => response.end(
    `<!DOCTYPE html><html><head></head>` +
    `<body>` +
    `<h1>Структура запроса</h1>` +
    `<h2>` + 'метод: ' + request.method + `</h2>` +
    `<h2>` + 'uri: ' + request.url + `</h2>` +
    `<h2>` + 'версия: ' + request.httpVersion + `</h2>` +
    `<h2>` + 'ЗАГОЛОВКИ: ' + `</h2>` +
    h(request) +
    `<h2>` + 'тело: ' + b + `</h2>` +
    `<body>` +
    `<html>` +
  ));
}).listen(3000, () => console.log('Server running at http://localhost:3000/'));
```

```
PS D:\NodeJS\samples\cwp_01> node .\02-02.js
Server running at http://localhost:3000/
```

GET-запрос (браузер)

← → C ① localhost:3000/ddd?x=3&y=2

Структура запроса

метод: GET

uri: /ddd?x=3&y=2

версия: 1.1

ЗАГОЛОВКИ

host:localhost:3000

connection:keep-alive

cache-control:max-age=0

upgrade-insecure-requests:1

user-agent:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.100 Safari/537.36

sec-fetch-mode:navigate

sec-fetch-user: ?1

accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3

sec-fetch-site:none

accept-encoding:gzip, deflate, br

accept-language:ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7

тело:

POST-запрос (Postman)

The screenshot shows the Postman application interface. At the top, a header bar includes a 'POST' method dropdown, a URL field with 'http://localhost:3000', a 'Send' button, and a 'Save' dropdown. Below the header are tabs for 'Params', 'Authorization', 'Headers (9)', 'Body' (which is selected and highlighted in green), 'Pre-request Script', 'Tests', 'Cookies', 'Code', and 'Comments (0)'. Under the 'Body' tab, there are radio buttons for 'none', 'form-data', 'x-www-form-urlencoded', 'raw' (which is selected and highlighted in red), 'binary', and 'GraphQL BETA'. A text input field contains the text '1 содержимое тела запроса'. The main content area shows the response body in 'Pretty' format, which is an HTML document. The HTML document contains the following structure:

```
1 <!DOCTYPE html>
2 <html>
3
4 <head></head>
5
6 <body>
7   <h1>Структура запроса</h1>
8   <h2>метод: POST</h2>
9   <h2>uri: /</h2>
10  <h2>версия: 1.1</h2>
11  <h2>ЗАГОЛОВКИ</h2>
12  <h3>content-type:text/plain</h3>
13  <h3>user-agent:PostmanRuntime/7.15.2</h3>
14  <h3>accept:/*</h3>
15  <h3>cache-control:no-cache</h3>
16  <h3>postman-token:63a217a2-8f51-406c-b4bf-99da4ae71ada</h3>
17  <h3>host:localhost:3000</h3>
18  <h3>accept-encoding:gzip, deflate</h3>
19  <h3>content-length:42</h3>
20  <h3>connection:keep-alive</h3>
21  <h2>тело: содержимое тела запроса</h2>
22 </body>
```

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

ГЛОБАЛЬНЫЕ ОБЪЕКТЫ

Глобальные объекты

- **global**
- **buffer**
- **require**
- **console**
- **exports**
- **module**
- **process**
- и др. (подробнее: <https://nodejs.org/api/globals.html>)

Объект global

=

специальный объект, который предоставляет доступ к глобальным, то есть доступным из каждого модуля приложения, переменным и функциям. Примерным аналогом данного объекта в JavaScript для браузера является объект window.

global vs window

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>02-03</title>
  <script src=".//s03-01.js" ></script>
</head>
<body>
  <h1>Lec 03</h1>
  <div id='result'></div>
  <script>
    var x = 3;
    result.innerHTML = 'Результат: ' + mod2(8);
  </script>
</body>
```

```
var x = 2;
mod2 = (inp)=>{return parseInt(inp)%x;}
```

Файл | D:/PSCA/Lec03/03-01.html

Lec 03

Результат: 2

global vs window

```
var http = require('http');
var fs = require('fs');
var mod = require('./m03-01');

http.createServer(function (request, response) {
  response.contentType='text/plain';

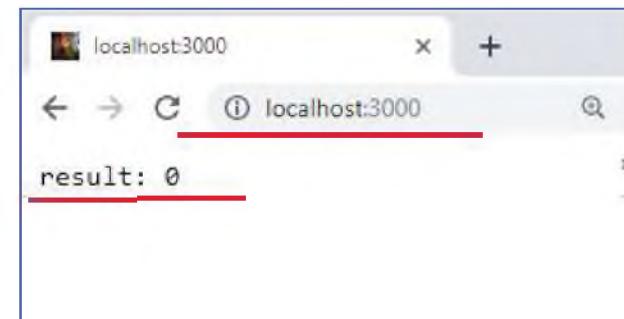
  var x = 3;

  let result = mod.mod2(8);

  response.end(`result: ${result}`);
}).listen(3000);

console.log('Server running at http://localhost:3000/');
```

```
var x = 2;
exports.mod2 = (inp)=>{return parseInt(inp)%x;}
```



Объект global

```
console.log('m03-01', global);
```

```
D:\NodeJS\samples\cwp_03>node 03-20
m03-01 <ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  performance: Performance {
    nodeTiming: PerformanceNodeTiming {
      name: 'node',
      entryType: 'node',
      startTime: 0,
      duration: 49.3146999998808,
      nodeStart: 1.127999995604157,
      v8Start: 6.34009999681115,
      bootstrapComplete: 37.592699999921024,
      environment: 21.660199999809265,
      loopStart: -1,
      loopExit: -1,
      idleTime: 0
    },
    timeOrigin: 1663340597424.766
  },
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  }
}
```

Объект
process

=

глобальный объект, который предоставляет информацию о текущем процессе Node.js и контролирует его.

Объект process

```
D:\NodeJS\samples\cwp_03>node -p "process.versions"
{
  node: '16.17.0',
  v8: '9.4.146.26-node.22',
  uv: '1.43.0',
  zlib: '1.2.11',
  brotli: '1.0.9',
  ares: '1.18.1',
  modules: '93',
  ngttpp2: '1.47.0',
  napi: '8',
  llhttp: '6.0.7',
  openssl: '1.1.1q+quic',
  cldr: '41.0',
  icu: '71.1',
  tz: '2022a',
  unicode: '14.0',
  ngtcp2: '0.1.0-DEV',
  ngttpp3: '0.1.0-DEV'
}
```

```
D:\NodeJS\samples\cwp_03>node -p "process.release"
{
  name: 'node',
  lts: 'Gallium',
  sourceUrl: 'https://nodejs.org/download/release/v16.17.0/node-v16.17.0.tar.gz',
  headersUrl: 'https://nodejs.org/download/release/v16.17.0/node-v16.17.0-headers.tar.gz',
  libUrl: 'https://nodejs.org/download/release/v16.17.0/win-x64/node.lib'
}
```

```

process.stdin.setEncoding('utf-8');
process.stdin.on('readable', ()=>{
  let chunk = null;
  while ((chunk = process.stdin.read()) != null){
    if (chunk.trim() == 'exit') process.exit(0);
    else if (chunk.trim() == 'uptime') process.stdout.write('uptime = ' + process.uptime().toString() + '\n');
    else if (chunk.trim() == 'version') process.stdout.write('version = ' + process.version + '\n');
    else if (chunk.trim() == 'title') process.stdout.write('title = ' + process.title + '\n');
    else if (chunk.trim() == 'release') {
      let obj = process.release;
      process.stdout.write('release:sourceUrl = ' + obj.sourceUrl + '\n');
      process.stdout.write('release:headersUrl = ' + obj.headersUrl + '\n');
      process.stdout.write('release:libUrl = ' + obj.libUrl + '\n');
      process.stdout.write('release:lts = ' + obj.lts + '\n');
    }
    else process.stdout.write(chunk);
    // else if (chunk.trim() == 'report') process.stdout.write('report = ' + process.report.filename + '\n'); v.11
    // else if (chunk.trim() == 'usage') { // v.12
    //   let obj = process.resourceUsage();
    //   process.stdout.write('usage:userCPUTime = ' + obj.userCPUTime.toString() + '\n');
    //   process.stdout.write('usage:systemCPUTime = ' + obj.systemCPUTime.toString() + '\n');
    //   process.stdout.write('usage: maxRSS = ' + obj.maxRSS);
    // }
  }
});
```

Объект `process` предоставляет доступ к стандартным системным потокам: **stdin, stdout, stderr**.

```

Administrator: C:\Windows\System32\cmd.exe - node 03-02

D:\PSCA\Lec03>node 03-02
Server running at http://localhost:3000/
uptime
uptime = 15.901
version
version = v10.15.0
title
title = Administrator: C:\Windows\System32\cmd.exe - node 03-02
release
release:sourceUrl = https://nodejs.org/download/release/v10.15.0/node-v10.15.0.tar.gz
release:headersUrl = https://nodejs.org/download/release/v10.15.0/node-v10.15.0-headers.tar.gz
release:libUrl = https://nodejs.org/download/release/v10.15.0/win-x64/node.lib
release:lts = Dubnium
```

Объект
buffer

=

глобальный объект,
предназначенный **для работы с
двоичными данными**: набором
октетов.

Применяется в функциях
readFile, writeFile; 1GB (32-bit),
2GB (64-bit).

```

const fs = require('fs');

let buf01 = Buffer.from([1,2,3,4,5, 6, 7, 8, 255, 254]);
fs.writeFile('./Files/File09a.dat',buf01, (e)=>{
    if (e) throw e;
    console.log(`- 1 -----`);
    console.log('buf01 = ', buf01);
    console.log('buf01.toString() =', buf01.toString());
    let x = 0;
    for (const s of buf01.values()) x+=s;
    console.log('buf01.values() =', buf01.values(), x);

    console.log('buf01.toJSON() =', buf01.toJSON());
});

let buf02 = Buffer.from('aaa bbb ccc AAA БББ 000','ascii');
fs.writeFile('./Files/File09b.dat',buf02, (e)=>{
    if (e) throw e;
    console.log(`- 2 -----`);
    console.log('buf02 = ', buf02);
    console.log('buf02.toString("ascii") =', buf02.toString('ascii'));
    let x = 0;
    for (const s of buf02.values()) x+=s;
    console.log('buf02.values() =', buf02.values(), x);
    console.log('buf02.toJSON() =', buf02.toJSON());
});

let buf03 = Buffer.from('aaa bbb ccc AAA aaa БББ 666 ','utf8');
fs.writeFile('./Files/File09c.dat', buf03, (e)=>{
    if (e) throw e;
    console.log(`- 3 -----`);
    console.log('buf03 = ', buf03);
    console.log('buf03.toString("utf8") =', buf03.toString('utf8'));
    console.log('buf03.values() =', buf03.values());
    console.log('buf03.toJSON() =', buf03.toJSON());
})

```

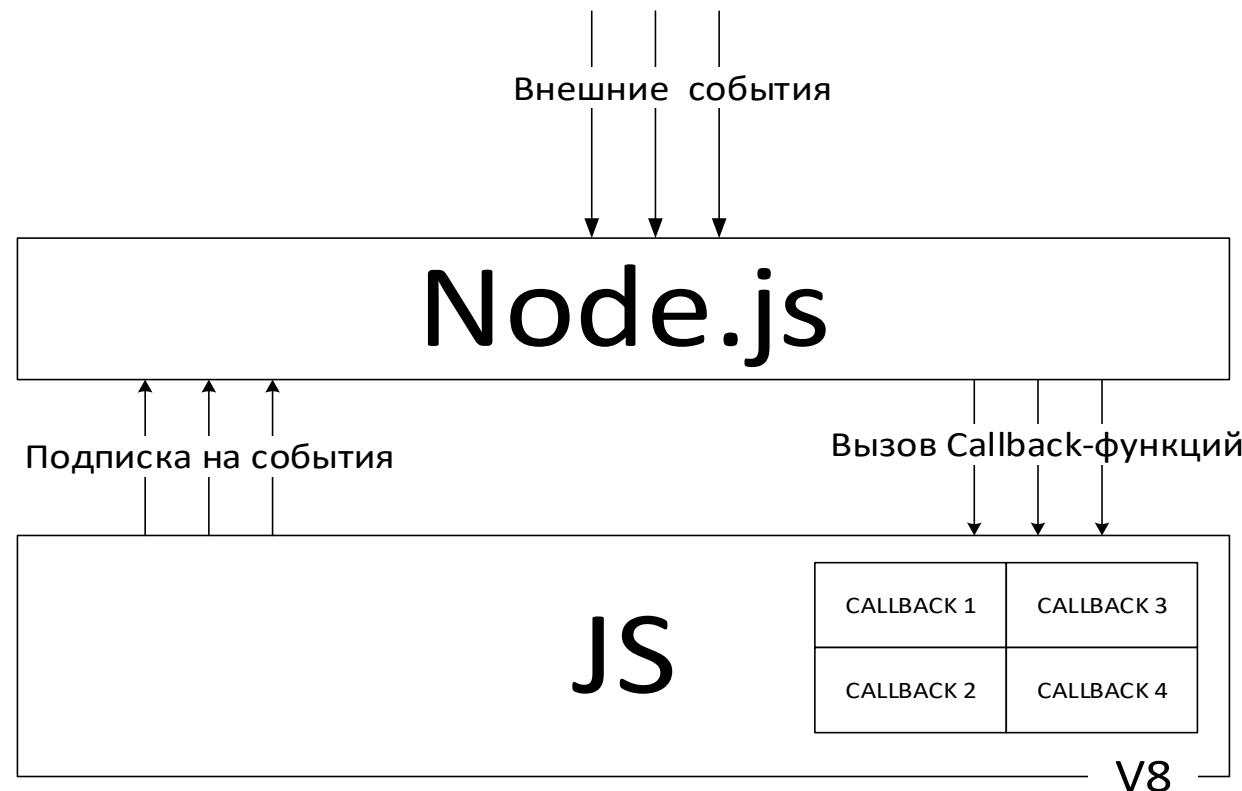
Пример

```

PS D:\Материалы\cwp_09> node .\09-03.js
-1-
buf01 = <Buffer 01 02 03 04 05 06 07 08 ff fe>
buf01.toString() = $$$
buf01.values() = Object [Array Iterator] {} 545
buf01.toJSON() = {
  type: 'Buffer',
  data: [
    1, 2, 3, 4, 5,
    6, 7, 8, 255, 254
  ]
}
-2-
buf02 = <Buffer 61 61 61 20 62 62 62 20 63 63 63 20 10 10 10 10 20 11 11 11 20 24 24>
buf02.toString() = aaa bbb ccc $$$
buf02.values() = Object [Array Iterator] {} 1249
buf02.toJSON() = {
  type: 'Buffer',
  data: [
    97, 97, 97, 32, 98, 98, 98, 32,
    99, 99, 99, 32, 16, 16, 16, 32,
    17, 17, 17, 32, 36, 36, 36
  ]
}
-3-
buf03 = <Buffer 61 61 61 20 62 62 62 20 63 63 63 20 d0 90 d0 90 d0 90 20 d0 91 d0 91 d0
91 20 d0 a4 d0 a4 d0 a4>
buf03.toString() = aaa bbb ccc AAA БББ 000
buf03.values() = Object [Array Iterator] {} 4273
buf03.toJSON() = {
  type: 'Buffer',
  data: [
    97, 97, 97, 32, 98, 98, 98, 32,
    99, 99, 99, 32, 208, 144, 208, 144,
    208, 144, 32, 208, 145, 208, 145, 208,
    145, 32, 208, 164, 208, 164, 208, 164
  ]
}

```

Принцип издатель-подписчик



```
var http = require('http');

var server = http.createServer();
console.log('-- after createServer');

server.on('request', (request, response)=>{
    console.log('request');
    response.contentType='text/html';
    response.end('request came');

});
console.log('-- after sever.on request');

server.on('connection', ()=>{
    console.log('connection');

});
console.log('-- after sever.on connection');

server.listen(3000, ()=>{
    console.log('listen');
});

console.log('Server running at http://localhost:3000/');
```

Node.js **событийно ориентирован**. То есть в коде мы подписываемся на какие-то события и когда они происходят, то вызывается коллбэк.

```
D:\PSCA\Lec03>node 03-03
-- after createServer
-- after sever.on request
-- after sever.on connection
Server running at http://localhost:3000/
listen
connection
request
connection
request
request
request
request
request
request
request
-
```

Синхронное вычисление числа Фибоначчи

```
var http = require('http');
var url = require('url');
var fs = require('fs');

var fib = (n) => { return (n < 2 ? n : fib(n - 1) + fib(n - 2)); } // Фибоначчи

http.createServer(function (request, response) {
  let rc = JSON.stringify({ k: 0 });
  let path = url.parse(request.url).pathname;

  if (path === '/fib') {
    let param = url.parse(request.url, true).query.k;
    if (typeof param != 'undefined') {
      let k = parseInt(param);
      if (Number.isInteger(k)) {
        response.writeHead(200, { 'Content-Type': 'application/json; charset=utf-8' });
        response.end(JSON.stringify({ k: k, fib: fib(k) }));
      }
    }
  }
  else if (path === '/') {
    let html = fs.readFileSync('fib.html');
    response.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    response.end(html);
  }
  else {
    response.end(rc);
  }
}).listen(5000, () => console.log('Server running at http://localhost:5000/'));
```

```
var http = require('http');
var url = require('url');
var fs = require('fs');

var fib = (n) => { return (n < 2 ? n : fib(n - 1) + fib(n - 2)); }

http.createServer(function (request, response) {
  let rc = JSON.stringify({ k: 0 });
  let path = url.parse(request.url).pathname;

  if (path === '/fib') {
    let param = url.parse(request.url, true).query.k;
    if (typeof param !== 'undefined') {
      let k = parseInt(param);
      if (Number.isInteger(k)) {
        response.writeHead(200, { 'Content-Type': 'application/json; charset=utf-8' });
        process.nextTick(() => response.end(JSON.stringify({ k: k, fib: fib(k) })))
      }
    }
  }
  else if (path === '/') {
    let html = fs.readFileSync('fib.html');
    response.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    response.end(html);
  }
  else {
    response.end(rc);
  }
}).listen(5000, () => console.log('Server running at http://localhost:5000/'));
```

process.nextTick()

```
var http = require('http');
var url = require('url');
var fs = require('fs');

var fib = (n) => { return (n < 2 ? n : fib(n - 1) + fib(n - 2)); }

http.createServer(function (request, response) {
  let rc = JSON.stringify({ k: 0 });
  let path = url.parse(request.url).pathname;

  if (path === '/fib') {
    let param = url.parse(request.url, true).query.k;
    if (typeof param !== 'undefined') {
      let k = parseInt(param);
      if (Number.isInteger(k)) {
        response.writeHead(200, { 'Content-Type': 'application/json; charset=utf-8' });
        setImmediate(() => response.end(JSON.stringify({ k: k, fib: fib(k) })))
      }
    }
  }
  else if (path === '/') {
    let html = fs.readFileSync('fib.html');
    response.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    response.end(html);
  }
  else {
    response.end(rc);
  }
}).listen(5000, () => console.log('Server running at http://localhost:5000/'));
```

setImmediate()

HTML-страница для измерения времени, затраченного на вычисление числа Фибоначчи

```
<!DOCTYPE html>
<html>

<head>
    <meta name="viewport" content="width=device-width" />
    <title>fibonacci</title>
</head>

<body>
    <div id='result'></div>
    <script>
        result.innerHTML = '';
        let n = 0;
        const d = Date.now();
        for (var k = 0; k < 20; k++) {
            fetch('http://localhost:5000/fib?k=${k}', {
                method: 'GET',
                mode: 'no-cors',
                headers: { 'Accept': 'application/json' }
            })
            .then(response => { return response.json(); })
            .then((pdata) => {
                result.innerHTML += (n++) + '.Результат: ' + (Date.now() - d) + ' - ' + pdata.k + '/' + pdata.fib + '<br/>';
            });
        }
    </script>
</body>

</html>
```

sync	nextTick	setImmediate
0.Результат: 32-0/0	0.Результат: 73-0/0	0.Результат: 161-0/0
1.Результат: 61-1/1	1.Результат: 73-1/1	1.Результат: 162-1/1
2.Результат: 61-2/1	2.Результат: 73-2/1	2.Результат: 163-2/1
3.Результат: 61-3/2	3.Результат: 73-3/2	3.Результат: 163-3/2
4.Результат: 80-4/3	4.Результат: 74-4/3	4.Результат: 163-4/3
5.Результат: 81-5/5	5.Результат: 74-5/5	5.Результат: 163-5/5
6.Результат: 81-6/8	6.Результат: 74-6/8	6.Результат: 163-6/8
7.Результат: 81-7/13	7.Результат: 100-7/13	7.Результат: 185-7/13
8.Результат: 82-8/21	8.Результат: 100-8/21	8.Результат: 185-8/21
9.Результат: 82-9/34	9.Результат: 100-9/34	9.Результат: 185-9/34
10.Результат: 83-10/55	10.Результат: 101-10/55	10.Результат: 185-10/55
11.Результат: 85-11/89	11.Результат: 101-11/89	11.Результат: 186-11/89
12.Результат: 86-12/144	12.Результат: 102-12/144	12.Результат: 186-12/144
13.Результат: 87-13/233	13.Результат: 105-13/233	13.Результат: 187-13/233
14.Результат: 87-14/377	14.Результат: 106-14/377	14.Результат: 187-14/377
15.Результат: 87-15/610	15.Результат: 106-15/610	15.Результат: 187-15/610
16.Результат: 87-16/987	16.Результат: 106-16/987	16.Результат: 187-16/987
17.Результат: 89-17/1597	17.Результат: 108-17/1597	17.Результат: 189-17/1597
18.Результат: 89-18/2584	18.Результат: 108-18/2584	18.Результат: 189-18/2584
19.Результат: 89-19/4181	19.Результат: 108-19/4181	19.Результат: 189-19/4181

Вывод: `nextTick()` срабатывает **быстрее**, чем `setImmediate()`.

Принцип функционирования стека-вызовов

```
function A1() { /* выполнение A1 */}
function A2() { /* выполнение A2 */}
function B() { A1(); A2(); /* выполнение B */ }
function C() { B(); /* выполнение C */ }

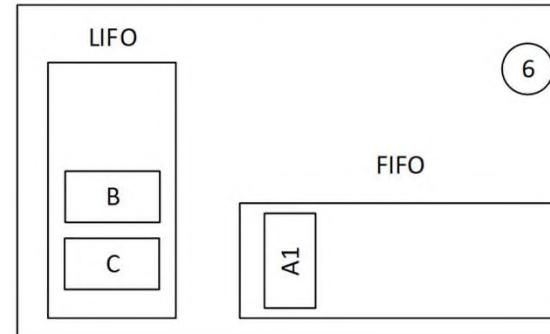
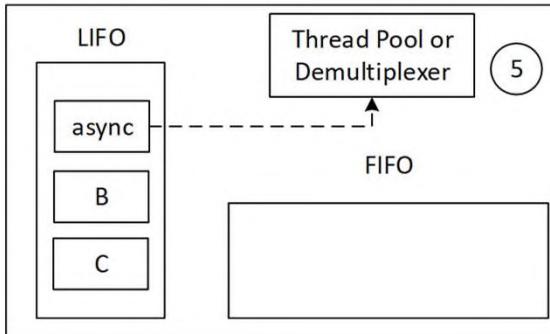
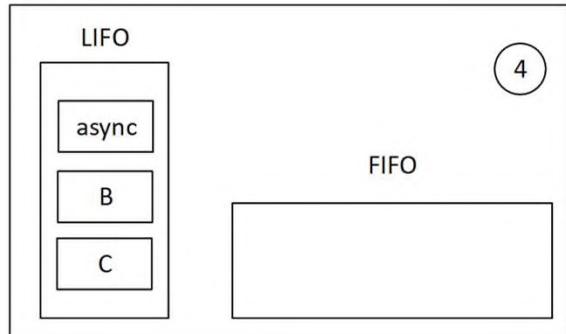
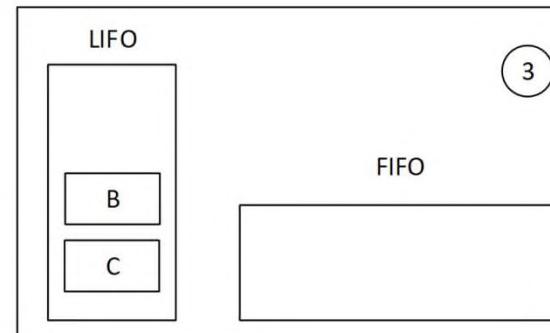
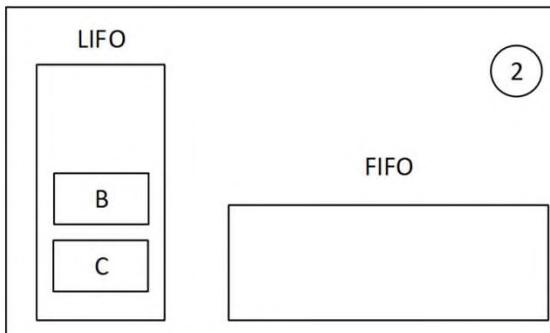
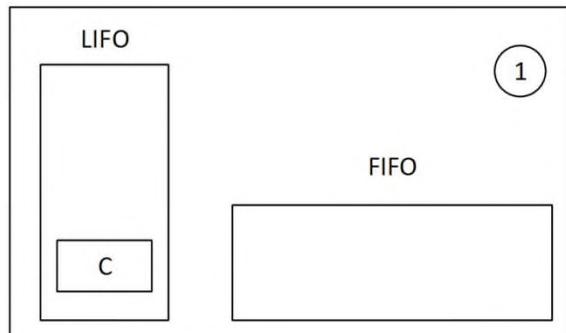
C();
```



Принцип функционирования стека-вызовов и очереди коллбэков

```
function A1() { /* выполнение A1 */}
function A2() { /* выполнение A2 */}
function A3() { /* выполнение A3 */}
function B() { async()=>{A1()}; A2(); async()=>{A3()}; /* выполнение B */}
function C() { B(); /* выполнение C */}

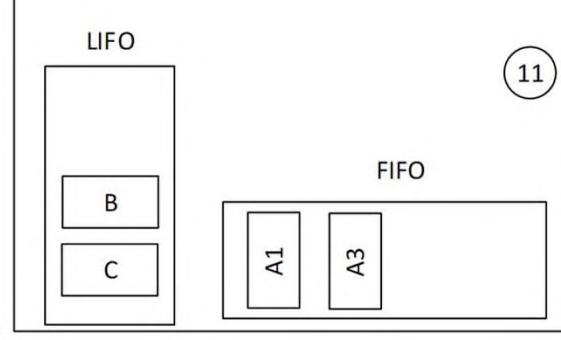
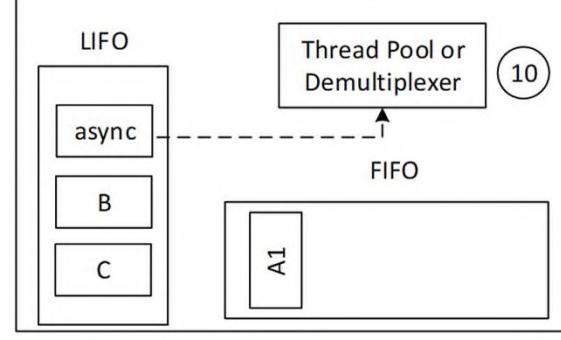
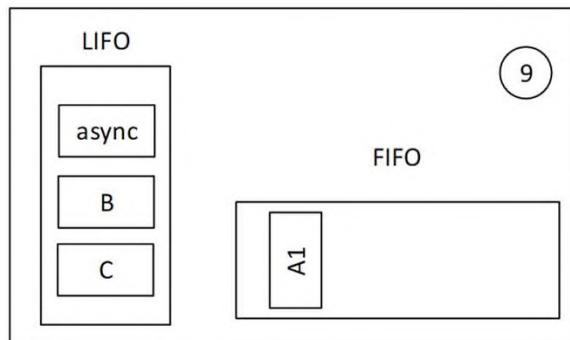
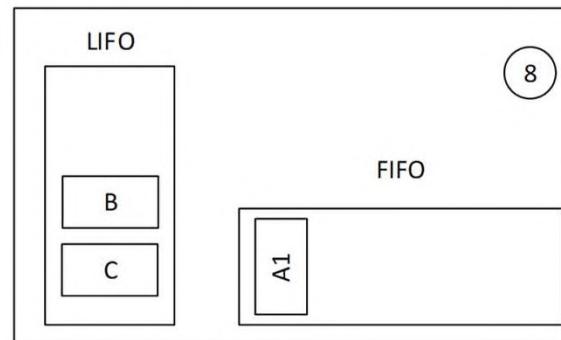
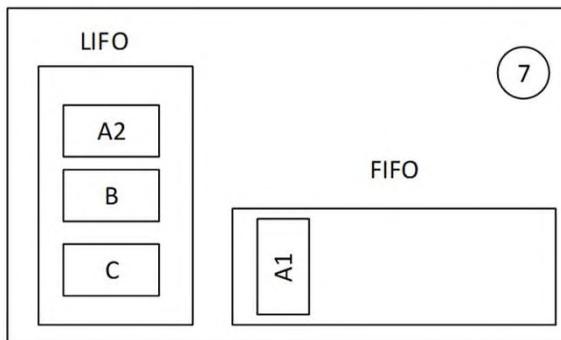
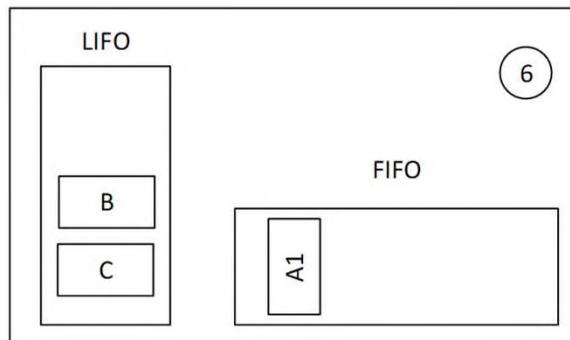
C();
```



Принцип функционирования стека-вызовов и очереди коллбэков

```
function A1() { /* выполнение A1 */}
function A2() { /* выполнение A2 */}
function A3() { /* выполнение A3 */}
function B() { async()=>{A1();}; A2(); async()=>{A3();}; /* выполнение B */}
function C() { B(); /* выполнение C */}

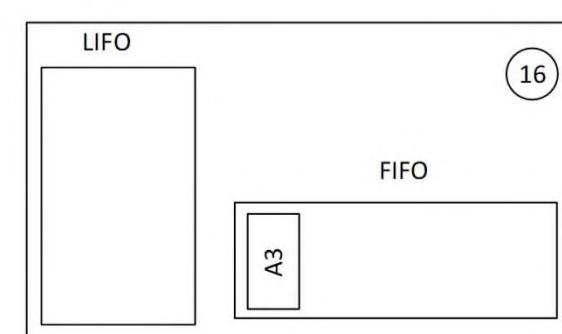
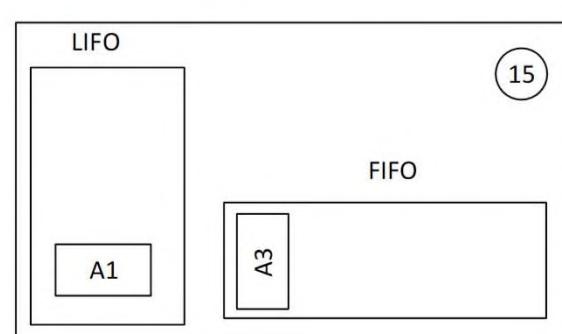
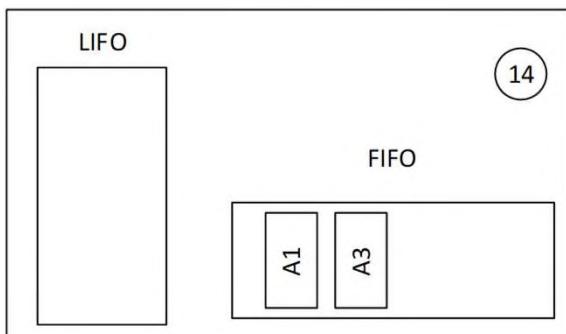
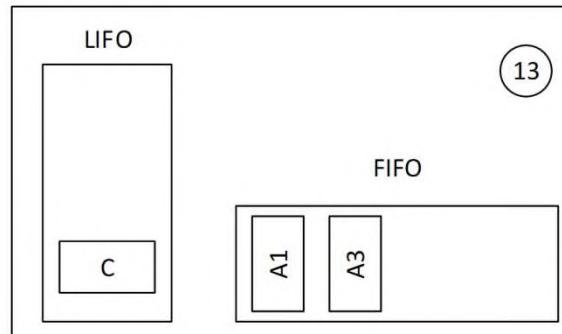
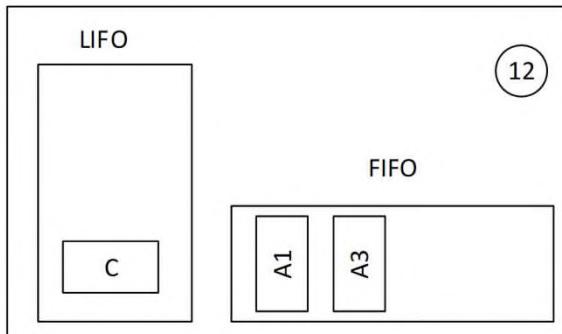
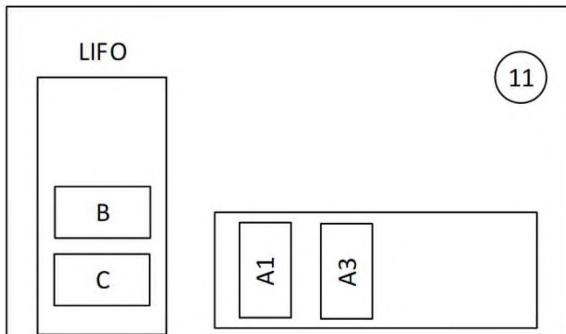
C();
```



Принцип функционирования стека-вызовов и очереди коллбэков

```
function A1() { /* выполнение A1 */}
function A2() { /* выполнение A2 */}
function A3() { /* выполнение A3 */}
function B() { async()=>{A1()}; A2(); async()=>{A3()}; /* выполнение B */}
function C() { B(); /* выполнение C */}

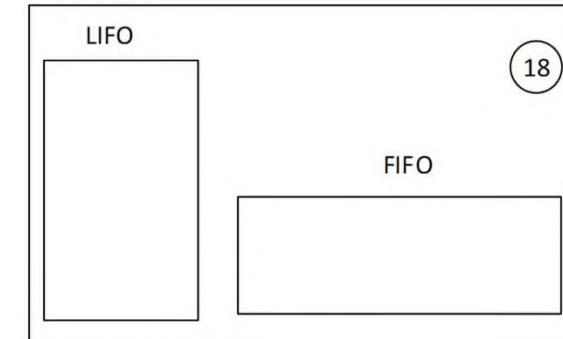
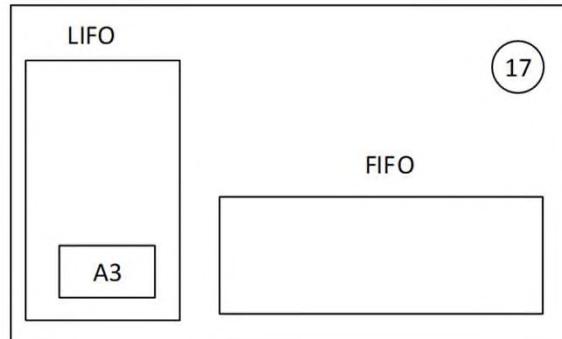
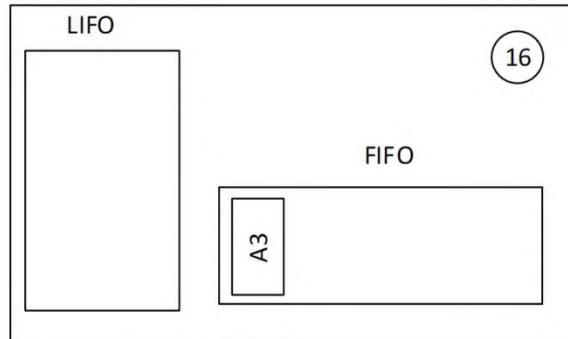
C();
```



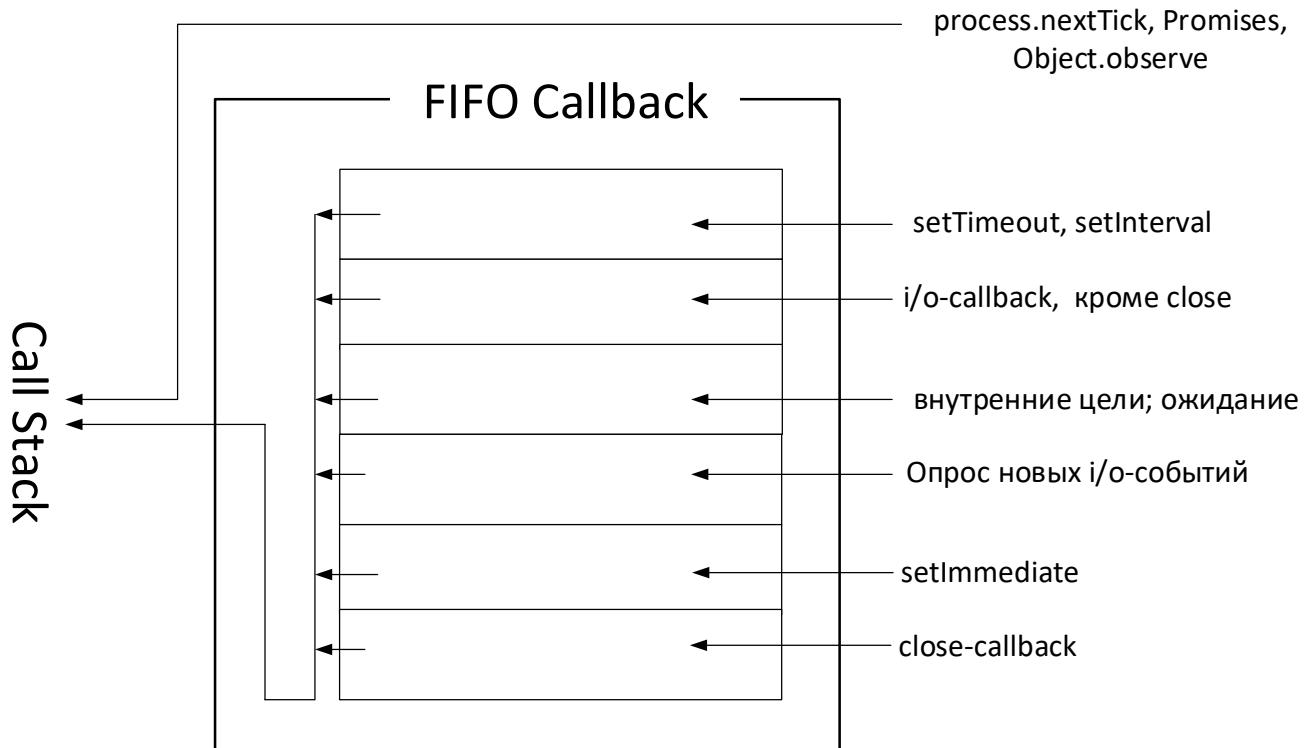
Принцип функционирования стека-вызовов и очереди коллбэков

```
function A1() { /* выполнение A1 */}
function A2() { /* выполнение A2 */}
function A3() { /* выполнение A3 */}
function B() { async()=>{A1()}; A2(); async()=>{A3()}; /* выполнение B */}
function C() { B(); /* выполнение C */}

C();
```



Обзор фаз event loop'a



- **макрозадачи** – выполняются по одной за один проход цикла (`setTimeout`, `setInterval`, `setImmediate`, `requestAnimationFrame`, `I/O`, `UI rendering`);
- **микрозадачи** – на каждом проходе цикл выполняет все накопившееся (`process.nextTick`, `Object.observe`, `Promises`).

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

Синхронное =
программирование

последовательное выполнение
инструкций с синхронными
системными вызовами,
которые **полностью блокируют**
поток выполнения, пока
системная операция
(например, чтение с диска) не
завершится.

JavaScript – это синхронный однопоточный язык

```
function a() { console.log( 'result of a()' ); }
function b() { console.log( 'result of b()' ); }
function c() { console.log( 'result of c()' ); }
a();
console.log('a() is done!');
b();
console.log('b() is done!');
c();
console.log('c() is done!');
```

```
PS D:\Материалы\cwp_03> node .\03-01.js
result of a()
a() is done!
result of b()
b() is done!
result of c()
c() is done!
```

Асинхронное =
программирование

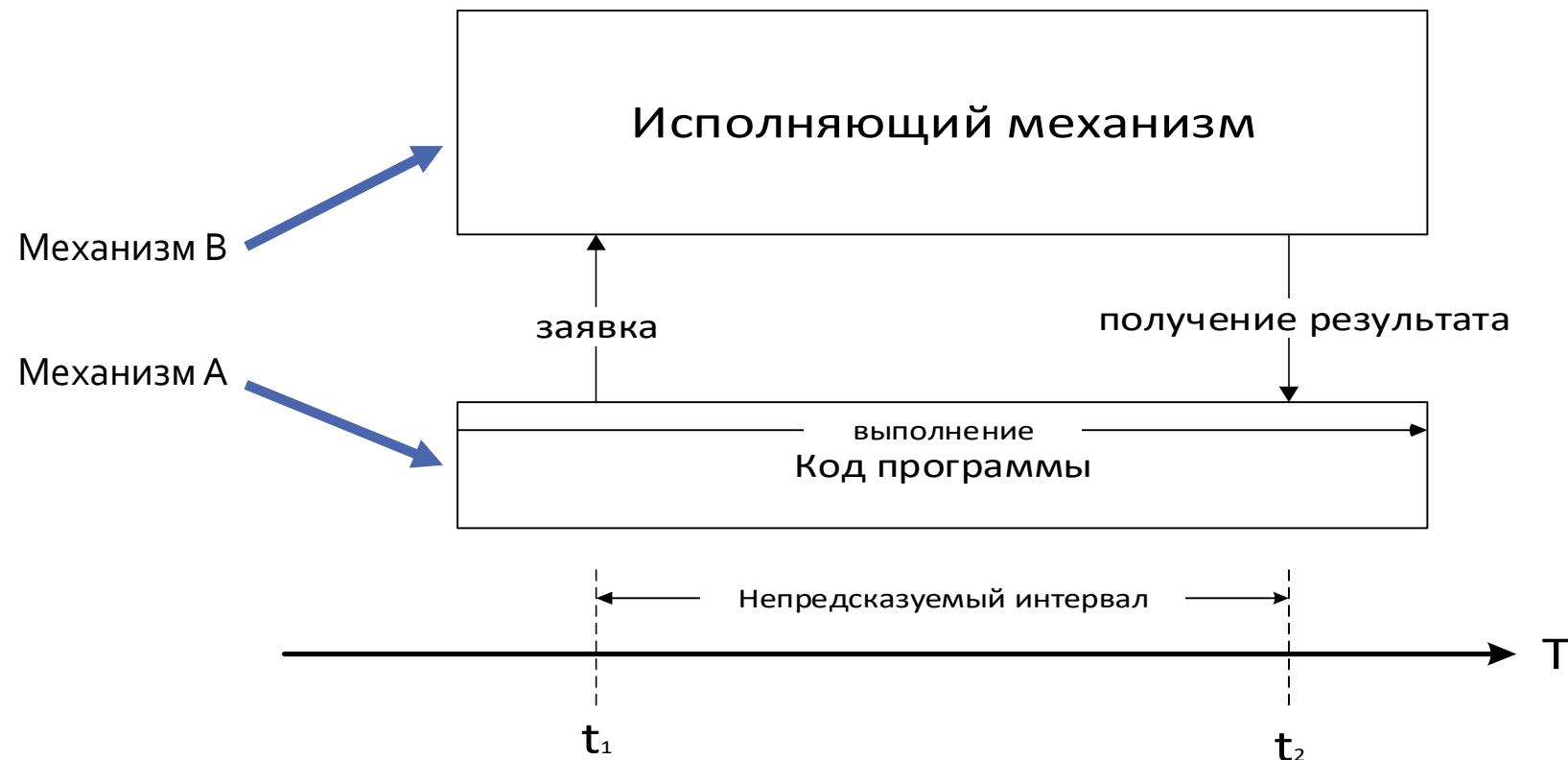
выполнение процесса в
неблокирующем режиме
системного вызова, что
позволяет потоку программы
продолжить обработку.

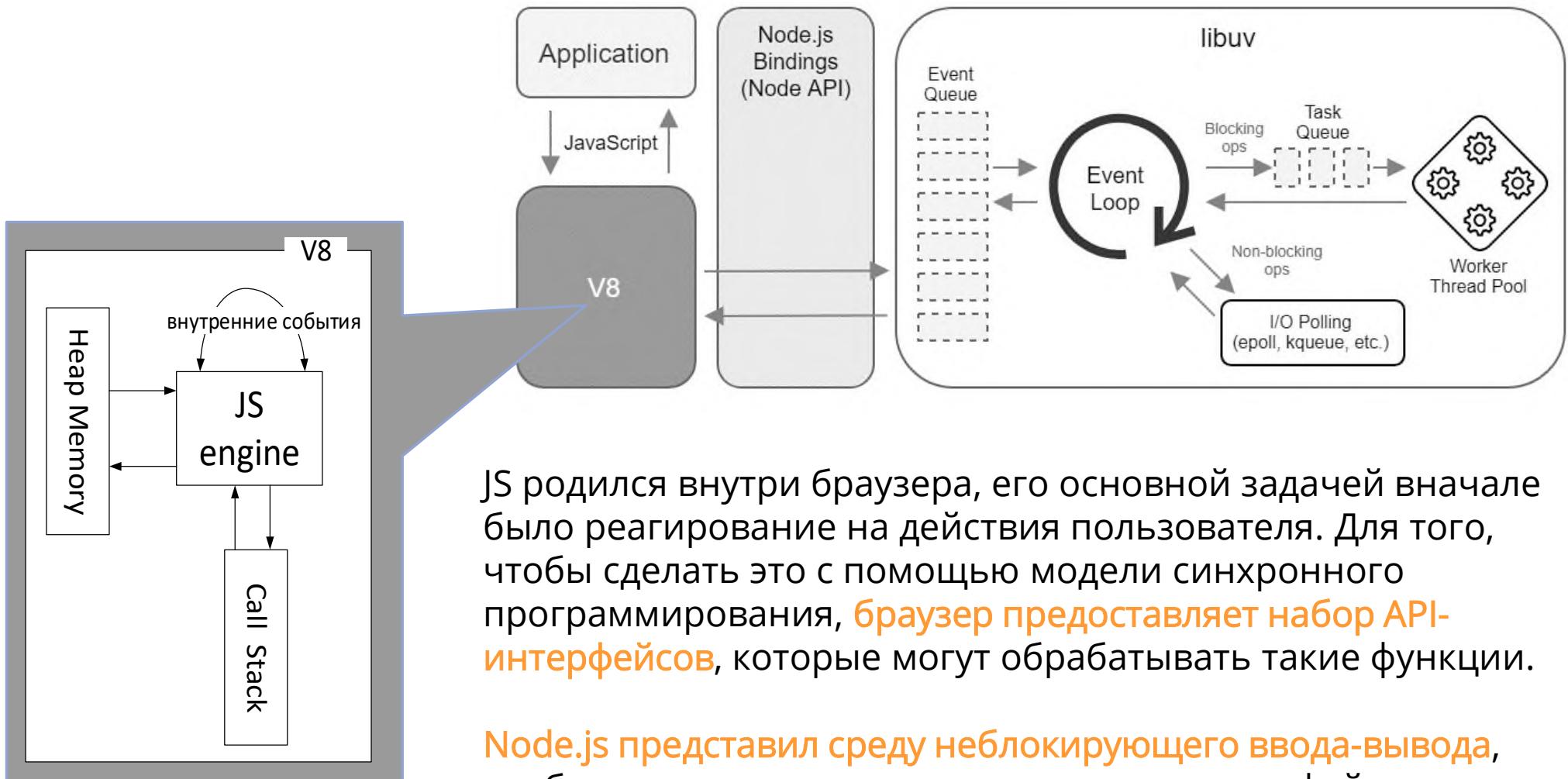
Асинхронность =

операция называется асинхронной, если ее выполнение осуществляется в 2 фазы:

- 1) **заявка на исполнение;**
- 2) **получение результата;** при этом участвуют два механизма: А-механизм, формирующий заявку и потом получающий результат; В-механизм, получающий заявку от А, исполняющий операцию и отправляющий результат А; продолжительность исполнения операции В-механизмом, как правило, непредсказуемо; в то время пока В-механизм исполняет операцию, А-механизм выполняет собственную работу.

Асинхронность в программировании





JS родился внутри браузера, его основной задачей вначале было реагирование на действия пользователя. Для того, чтобы сделать это с помощью модели синхронного программирования, **браузер предоставляет набор API-интерфейсов**, которые могут обрабатывать такие функции.

Node.js представил среду неблокирующего ввода-вывода, чтобы расширить эту концепцию на доступ к файлам, сетевые вызовы и так далее.

```
function a() {
  setTimeout( function() {
    console.log( 'result of a()' );
  }, 1000 );
}

function b() {
  setTimeout( function() {
    console.log( 'result of b()' );
  }, 500 );
}

function c() {
  setTimeout( function() {
    console.log( 'result of c()' );
  }, 1200 );
}

a();
console.log('a() is done!');

b();
console.log('b() is done!');

c();
console.log('c() is done!');
```

Функция `setTimeout(callback, delay)` – глобальная функция, которая принимает функцию обратного вызова и временно сохраняет ее. По истечении времени, заданного в миллисекундах, функция обратного вызова помещается в **очередь коллбэков (callback queue)**. Затем event loop перемещает эту функцию в **стек вызовов (call stack)**, когда стек пуст. После этого осуществляется выполнение функции обратного вызова.

В основном так работают все асинхронные операции.

```
PS D:\Материалы\cwp_03> node 03-02.js
a() is done!
b() is done!
c() is done!
result of b()
result of a()
result of c()
```

Как сделать так, чтобы результат был таким?

```
result of b()  
b() is done!  
result of a()  
a() is done!  
result of c()  
c() is done!
```

Асинхронное программирование в JS

- **Callbacks**
- **Promises (> callbacks)**
- **Async/await (> promises > callbacks)**
- **Async.js (> callbacks)**
- **Generators/yield**
- **Events (> observable > callbacks)**
- **Functor + chaining + composition**
- **For await + Symbol.asyncIterator**

Callback
(функция обратного
вызыва)

функция, которая передается в качестве параметра другой функции и которая **будет вызвана** асинхронно обработчиком событий **после завершения задачи**.

```
function a(callback) {
  setTimeout( () => {
    console.log( 'result of a()' );
    callback();
  }, 1000 );
}

function b(callback) {
  setTimeout( () => {
    console.log( 'result of b()' );
    callback();
  }, 500 );
}

function c(callback) {
  setTimeout( () => {
    console.log( 'result of c()' );
    callback();
  }, 1200 );
}

a( () => console.log('a() is done!') );
b( () => console.log('b() is done!') );
c( () => console.log('c() is done!') );
```

Callbacks

Callback должен идти последним параметром в функцию.

```
PS D:\Материалы\cwp_03> node 03-06.js
result of b()
b() is done!
result of a()
a() is done!
result of c()
c() is done!
```

Node.js callback style

Первым параметром в любой функции обратного вызова является **объект ошибки**.

Если ошибки нет, объект равен нулю. Если есть ошибка, он содержит некоторое описание ошибки и другую информацию.

Вторым, третьим, четвертым и др. идут **данные, результат**.

```
function callback(error, data) {  
  if (error) {  
    // обрабатываем ошибку  
  } else {  
    // обрабатываем данные  
  }  
}
```

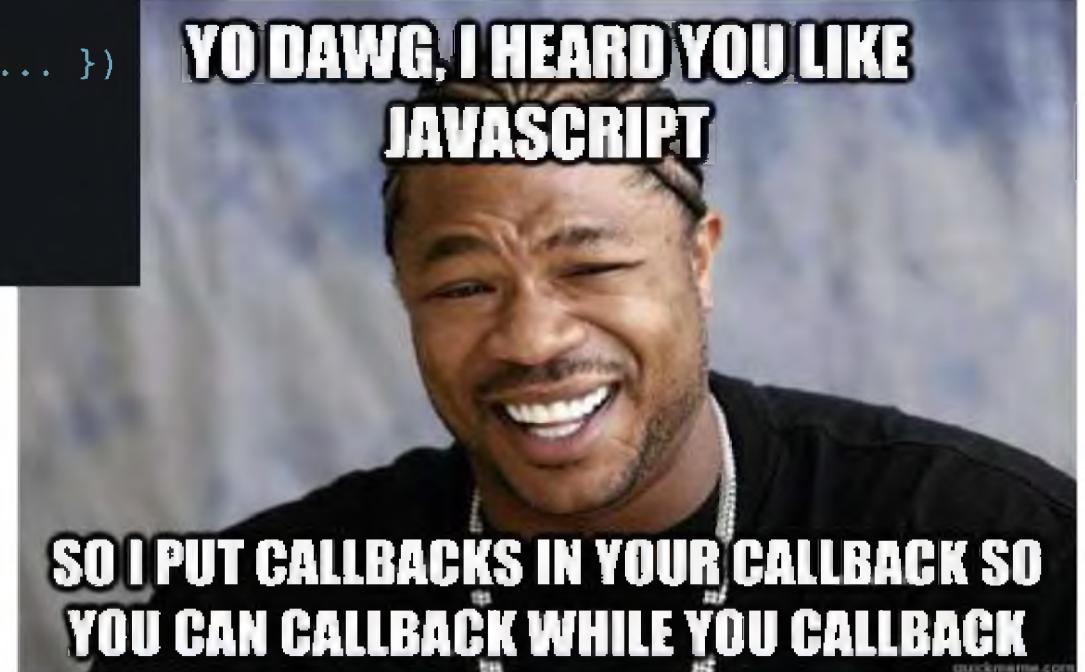
```
fs.readFile('/file.json', (err, data) => {  
  if (err !== null) {  
    // обрабатываем ошибку  
    console.log(err);  
    return;  
  }  
  // ошибок нет, обрабатываем данные  
  console.log(data);  
})
```

Callbacks

- + повторное использование кода;
- + изменение функциональности без изменения метода;
- + создание цепочек вызовов;
- падение производительности;
- ухудшение читаемости.

Callback hell, pyramid of doom

```
function printAll(){
  printString("A", () => {
    printString("B", () => {
      printString("C", () => {
        printString("D", () => { ... })
      })
    })
  })
printAll();
```



Решение callback hell

```
fs.readdir(dir, (err, files) => {
  if (err) {
    // ...
  }

  files.forEach((name) => {
    if (!isImage(name)) return;

    fs.readFile(name, (err, image) => {
      if (err) {
        // ...
      }

      compress(image, (err, comp) => {
        fs.writeFile(name, comp, (err) =>{
          if (err) {
            // ...
          } else {
            // ...
          });
        });
      });
    });
  });
});
```

1. Разбивать код на функции



```
function processFiles(err, files) {
  if (err) {
    // ...
  }
  files.forEach(checkFile);
}

function checkFile(name) {
  if (!isImage(name)) return;
  fs.readFile(name, compressImage(name));
}

function compressImage(name) {
  return (err, image) => {
    if (err) {
      // ...
    }
    compress(file, rewriteImage(name));
  }
}

function rewriteImage(name) {
  return (err, image) => {
    fs.writeFile(name, image, finishCompress);
  }
}

function finishCompress(err) {
  if (err) {
    // ...
  } else {
    // ...
  }
}

fs.readdir(dir, processFiles);
```

Решение callback hell

```
fs.readdir(dir, (err, files) => {
  if (err) {
    // ...
  }

  files.forEach((name) => {
    if (!isImage(name)) return;

    fs.readFile(name, (err, image) => {
      if (err) {
        // ...
      }

      compress(image, (err, comp) => {
        fs.writeFile(name, comp, (err) =>{
          if (err) {
            // ...
          } else {
            // ...
          });
        });
      });
    });
  });
});
```

2. Разбитие на модули



```
function processFiles(err, files) {
  if (err) { }
  files.forEach(checkFile);
}

function checkFile(name) {
  if (!isImage(name)) return;
  fs.readFile(name, compressImage(name));
}

function compressImage(name) {
  return (err, image) => {
    if (err) { }
    compress(file, rewriteImage(name));
  }
}

function rewriteImage(name) {
  return (err, image) => {
    fs.writeFile(name, image, finishCompress);
  }
}

function finishCompress(err) {
  if (err) { }
  else { }
}

function readDir(dir) {
  fs.readdir(dir, processFiles);
}

module.exports = readDir;

const compressImagesDir = require('./compressImagesDir');
compressImagesDir('/tmp/images');
```

Promise
(Обещание)
ES6 / ES2015

=

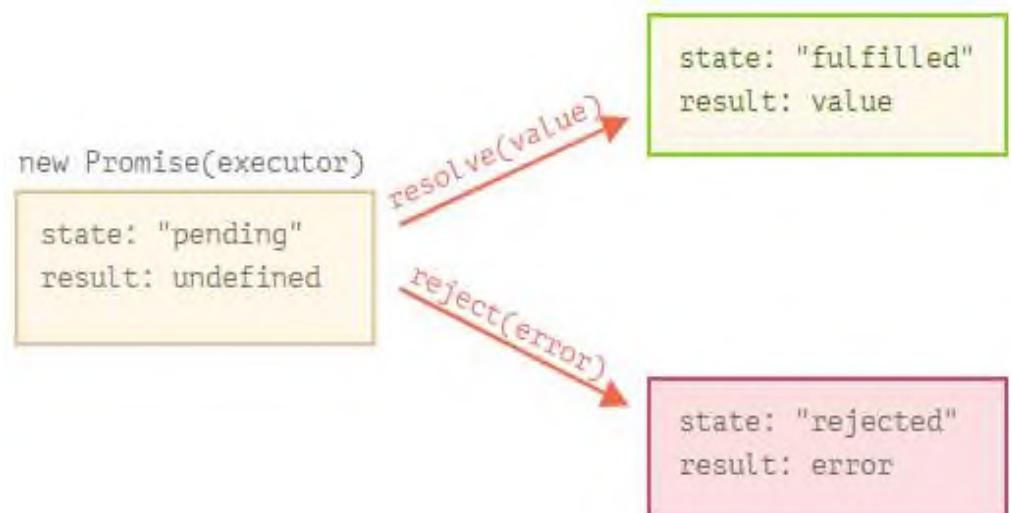
объект, используемый для выполнения **отложенных и асинхронных вычислений**. Представляет собой операцию, которая еще не завершена, но ожидается в будущем.

Promises

Свойство state:

- **pending** (ожидание),
- **fulfilled** (выполнено) при вызове `resolve`,
- **rejected** (отклонено) при вызове `reject`.

Свойство result: вначале `undefined`,
далее изменяется на `value` при
вызове `resolve(value)` или на `error`
при вызове `reject(error)`.



Promises

resolve и **reject** – это статические методы класса `Promise`, которые выполняют и отклоняют промис соответственно.

Вызов `resolve =>`
обработчик then

Вызов `reject =>`
обработчик catch

```
var myPromise = new Promise( (resolve, reject) => {
  // все прошло успешно
  resolve('successPayload');

  // что-то пошло не так
  // reject('errorPayload');
} );

myPromise
  .then((result) => {
    // обрабатываем результат
  })
  .catch((err) => {
    // обрабатываем ошибку
  })
  .finally(finallyCallback);

myPromise.then((files) => { }, (err) => { });
```

Resolve => then => finally

```
const promiseA = new Promise((resolve, reject) => {
  setTimeout( () => {
    resolve('result of a()');
  }, 1000 );
});

promiseA
.then((result) => {
  console.log('promiseA success:', result);
})
.catch((error) => {
  console.log('promiseA error:', error);
})
.finally(() => {
  console.log('a() is done!');
});
```

```
PS D:\Материалы\cwp_03> node .\03-08.js
promiseA success: result of a()
a() is done!
PS D:\Материалы\cwp_03> █
```

Reject => catch => finally

```
const promiseA = new Promise((resolve, reject) => {
  setTimeout( () => {
    //resolve('result of a());
    reject('something bad happened a());
  }, 1000 );
});

promiseA
.then((result) => {
  console.log('promiseA success:', result);
})
.catch((error) => {
  console.log('promiseA error:', error);
})
.finally(() => {
  console.log('a() is done!');
});
```

```
PS D:\Материалы\cwp_03> node .\03-08.js
promiseA error: something bad happened a()
a() is done!
PS D:\Материалы\cwp_03> []
```

Обработка отклоненного промиса без catch

```
const promiseA = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('something bad happened a()');
  }, 1000);
});

promiseA
  .then((result) => {
    console.log('promiseA success:', result);
  }, (error) => {
    console.log('promiseA error:', error);
})
  .finally(() => {
    console.log('a() is done!');
});
```

```
PS D:\Материалы\cwp_03> node .\03-09.js
promiseA error: something bad happened a()
a() is done!
PS D:\Материалы\cwp_03>
```

Обратные вызовы промисов помещаются в очередь микрозадач (microtask queue)

```
setTimeout(() => console.log('setTimeout callback'), 0);

// immediately resolved promise
const promiseA = new Promise((resolve) => resolve());

// sync
console.log('I am sync job.');

// promise listener
promiseA.then(() => {
  console.log('promiseA success:');
});

// sync
console.log('I am good sync job.');
console.log('I am awesome sync job too.');
```

```
PS D:\Материалы\cwp_03> node .\03-10.js
I am sync job.
I am good sync job.
I am awesome sync job too.
promiseA success:
setTimeout callback
PS D:\Материалы\cwp_03> []
```

```
Promise.reject('Reject DATA!')  
  .then((result) => {  
    console.log('[1] then', result); // won't be called  
    return '[2] then payload';  
  })  
  .finally(() => {  
    console.log('[1] finally'); // first finally will be called  
    return '[1] finally payload';  
  })  
  .then((result) => {  
    console.log('[2] then', result); // won't be called  
    return '[2] then payload';  
  })  
  .catch((error) => {  
    console.log('[1] catch', error); // first catch will be called  
    return '[1] catch payload';  
  })  
  .catch((error) => {  
    console.log('[2] catch', error); // won't be called  
    return '[2] catch payload';  
  })  
  .then((result) => {  
    console.log('[3] then', result); // will be called  
    return '[3] then payload';  
  })  
  .finally(() => {  
    console.log('[2] finally'); // will be called  
    return '[2] finally payload';  
  })  
  .catch((error) => {  
    console.log('[3] catch', error); // won't be called  
    return '[3] catch payload';  
  })  
  .then((result) => {  
    console.log('[4] then', result); // will be called  
    return '[4] then payload';  
  });
```

Цепочка промисов

Обработчик промиса возвращает **НОВЫЙ ПРОМИС** с неопределенным result (можно его задать с помощью оператора return), поэтому их можно объединять в цепочки.

```
PS D:\Материалы\cwp_03> node .\03-11.js  
[1] finally  
[1] catch Reject DATA!  
[3] then [1] catch payload  
[2] finally  
[4] then [3] then payload  
PS D:\Материалы\cwp_03> █
```

```
const a = () => new Promise(resolve => {
  setTimeout(() => resolve('result of a()'), 1000);
});

const b = () => new Promise(resolve => {
  setTimeout(() => resolve('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

Promise.all([a(), b(), c(), { key: 'I am plain data!' }])
  .then((data) => {
    console.log('success: ', data);
  })
  .catch((error) => {
    console.log('error: ', error);
  });

```

Метод **Promise.all (iterable)** возвращает промис, который выполнится **после выполнения всех промисов** в передаваемом итерируемом аргументе.

```
PS D:\Материалы\cwp_03> node 03-12.js
success: [
  'result of a()',
  'result of b()',
  'result of c()',
  { key: 'I am plain data!' }
]
PS D:\Материалы\cwp_03> █
```

```
const a = () => new Promise(resolve => {
  setTimeout(() => resolve('result of a()'), 1000);
});

const b = () => new Promise(resolve => {
  setTimeout(() => resolve('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

Promise.race([a(), b(), c()])
  .then((data) => {
    console.log('success: ', data);
  })
  .catch((error) => {
    console.log('error: ', error);
  });

```

Метод **Promise.race (iterable)** возвращает промис, который будет выполнен или отклонен с результатом исполнения **первого** выполненного или **отклонённого** итерируемого промиса.

```
PS D:\Материалы\cwp_03> node 03-12.js
success: result of b()
PS D:\Материалы\cwp_03> █
```

async/await =
ES8 / ES2017

синтаксис для обработки
нескольких промисов **в режиме**
синхронного кода.

Async/await

async – перед объявлением функции, возвращает промис

await – блокирует код до тех пор, пока промис не будет разрешен или отклонен.

```
async function myFunction() {  
  var result = await new MyPromise();  
  console.log(result);  
}  
myFunction(); // returns a promise
```

```
const a = () => new Promise(resolve => {
  setTimeout(() => resolve('result of a()'), 1000);
});

const b = () => new Promise((resolve, reject) => {
  setTimeout(() => resolve('result of b()'), 500);
  //setTimeout(() => reject('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

const doJobs = async () => {
  var resultA = await a();
  var resultB = await b();
  var resultC = await c();

  return [resultA, resultB, resultC];
};

// doJobs() returns a promise
doJobs().then((result) => {
  console.log('success:', result);
})
  .catch((error) => {
    console.log('error:', error);
  });

console.log('I am a sync operation');
```

Async/await

```
PS D:\Материалы\cwp_03> node .\03-13.js
I am a sync operation!
success: [ 'result of a()', 'result of b()', 'result of c()' ]
PS D:\Материалы\cwp_03> █
```

```
PS D:\Материалы\cwp_03> node .\03-13.js
I am a sync operation!
error: result of b()
PS D:\Материалы\cwp_03> █
```

Async/await (обработка Error)

```
const a = () => new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('An error occurred')), 1000);
});

const b = () => new Promise(resolve => {
  setTimeout(() => resolve('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

const doJobs = async () => {
  try {
    var resultA = await a();
    var resultB = await b();
    var resultC = await c();

    console.log('Success: ', [resultA, resultB, resultC]);
  } catch (error) {
    console.log('error: ', error.message)
  }
};

doJobs();
console.log('I am a sync operation!');
```

Блок catch обрабатывает только
reject(new Error(...)). На reject("")
блок catch не сработает

```
const doJobs = async () => {
  var resultA = await a();
  var resultB = await b();
  var resultC = await c();

  return [resultA, resultB, resultC];
};

doJobs().then(result => {
  console.log('success:', result);
}).catch(error => {
  console.log('error:', error.message);
});

console.log('I am a sync operation!');
```

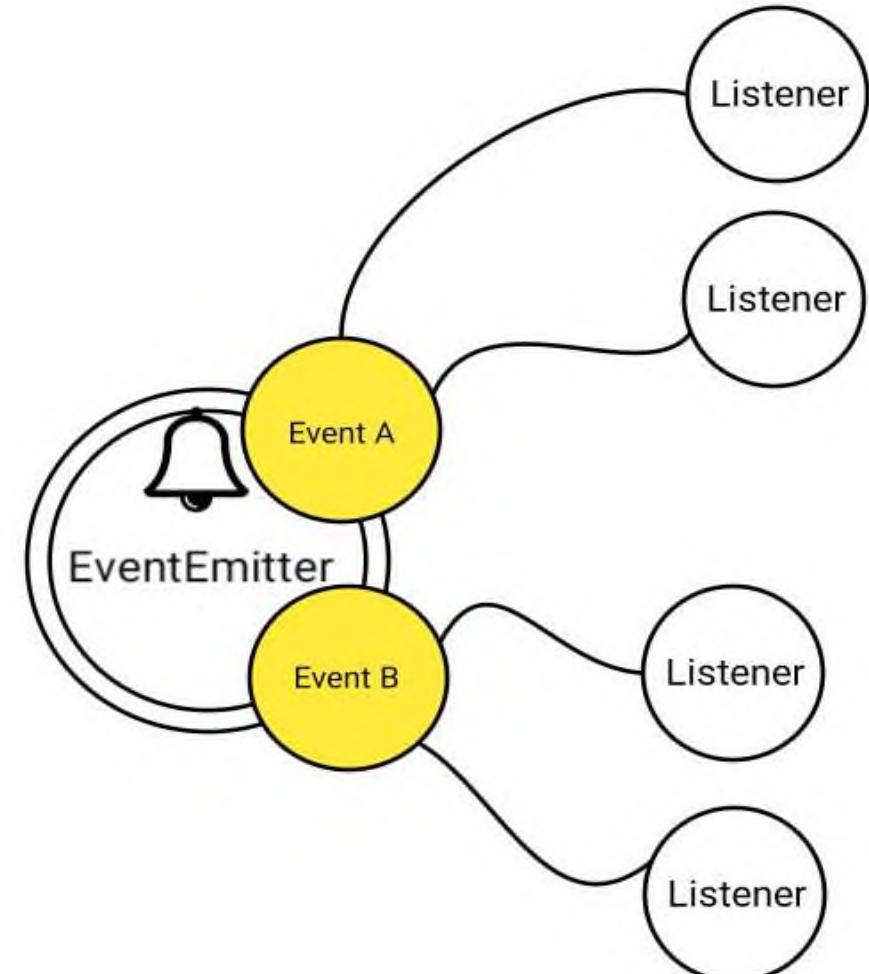
PS D:\NodeJS\samples\cwp_03> node 03-23
I am a sync operation!
error: An error occurred

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

EVENT Emitter. Таймеры

Подавляющее большинство функционала Node.js применяет **асинхронную событийную архитектуру**, которая использует специальные объекты **для генерации различных событий**, которые обрабатываются специальными функциями – **обработчиками** или **слушателями** событий.

Все объекты, которые генерируют события, представляют экземпляры класса `EventEmitter`.



EVENT Emitter =

JS-класс, предоставляющий функциональность **для** обработки событий в Node.js.

Событие программного = объекта

процесс перехода объекта из одного состояния в другое. При этом, об этом переходе могут быть извещены другие объекты. У события есть **издатель** (или генератор) события и могут быть **подписчики** (или обработчики) события.

EVENT Emitter

Примеры подписки на события

```
// --- EventEmitter ---  
  
// request.on('data', ....)  
  
// response.end(html);           // = response.on('end' ...)  
  
// server.on('request', ....)  
  
// process.stdin.on('readable',
```

Производный от Event Emitter объект приобретает функциональность, позволяющую генерировать и прослушивать события

```
const EventEmitter = require('events');

class DB extends EventEmitter {
    // реализация
}
```

Для генерации событий предназначена функция **emit()**, а для прослушивания – функция **on()**.

Пример использования Event Emitter

```
// отдельный модуль

const EventEmitter = require('events');

let db_data = [    // имитация базы данных
  { id: 1, name: "Иванов И.И", bday: '2001-01-01' },
  { id: 2, name: "Петров П.П", bday: '2001-01-02' },
  { id: 3, name: "Сидоров С.С", bday: '2001-01-03' },
];

class DB extends EventEmitter {
  get() { return db_data; }          // реализация GET
  post(object) { db_data.push(object); } // реализация POST
}

exports.DB = DB;      // экспортируется класс DB
```

Пример использования Event Emitter

```
const http = require('http');
const url = require('url');
const fs = require('fs');
let data = require('./DB.js');      // DB-объект, производный от EventEmitter

let db = new data.DB();
// ---- слушатели событий -----
db.on('GET', (request, response) => {
  console.log('DB.GET');
  response.end(JSON.stringify(db.get()));
});

db.on('POST', (request, response) => {
  console.log('DB.POST');
  request.on('data', data => {
    let r = JSON.parse(data);
    db.post(r);
    response.end(JSON.stringify(r));
  })
});

db.on('PUT', (request, response) => {
  console.log('DB.PUT');
  // ...реализация
});

db.on('DELETE', (request, response) => {
  console.log('DB.DELETE');
  // ...реализация
});
// ----

http.createServer((request, response) => {
  if (url.parse(request.url).pathname === '/') {
    let html = fs.readFileSync('./04-02.html');
    response.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    response.end(html);
  }
  else if (url.parse(request.url).pathname === '/api/db') {
    db.emit(request.method, request, response);    // для генерации событий, имя события - строка
  }
}).listen(3000);
```

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>04-04/05</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/bootstrap.min.css" integrity="sha384-PsH8R72JQ3SOaXppEustaQOMmE5q0PfXtWtDQXm" crossorigin="anonymous"/>
</head>
<body>
    <h1>Lec 04</h1>
    <div id="get_result"></div>
    <button onclick="Get()">GET</button>
    <script>...
    </script>
    <br/>
    <div style="padding: 20px">
        <div class='row'>
            <label class="col-2"> Идентификатор</label>
            <input type="number" class="col-3" id = "ID" min="0" />
        </div>
        <div class='row'>
            <label class="col-2"> ФИО</label>
            <input type="text" class="col-3" id = "Name" />
        </div>
        <div class='row'>
            <label class="col-2"> Дата рождения</label>
            <input type="date" class="col-3" id = "BDay" />
        </div>
        <div class='row'>
            <button class="col-2" onclick="Post()"> POST </button>
        </div>
    </div>
    <script>...
    </script>
</body>
</html>
```

```
<script>
    function Get(){
        console.log('GET');
        fetch('http://localhost:3000/api/db', {
            method: 'GET', mode: 'no-cors',
            headers: { 'Content-Type': 'application/json', 'Accept': 'application/json' }
        })
        .then(response => { return response.json(); })
        .then((pdata) => {
            console.log('pdata', pdata);
            get_result.innerHTML = '';
            pdata.forEach(el => { get_result.innerHTML += ( el.id + ' ' + el.name + ' ' + el.bday + '  
' ); });
        });
    }
</script>
```

```
<script>
    function Post(){
        console.log('POST');
        fetch('http://localhost:3000/api/db', {
            method: 'POST', mode: 'no-cors',
            headers: { 'Content-Type': 'application/json', 'Accept': 'application/json' },
            body: JSON.stringify({id:ID.value, name: Name.value, bday:BDay.value })
        })
        .then(response => { return response.json(); })
        .then((pdata) => { console.log('POST.pdata', pdata); });
    }
</script>
```

Результат (браузер)

Lec 04

1. Иванов И.И 2001-01-01
2. Петров П.П 2001-01-02
3. Сидоров С.С 2001-01-03
4. Козлов К.К. 1999-02-16

GET

Идентификатор

4

ФИО

Козлов К.К.

Дата рождения

16.02.1999



POST

Методы EventEmitter'a

- **emit** (eventName[, ...args]) – генерация события с именем eventName. Синхронно вызывает каждый слушатель (в порядке их регистрации) и передает им аргументы args.
- **on** (eventName, listener) – добавляет слушателя в конец массива слушателей на событие с именем eventName. Один и тот же слушатель может быть вызван для события несколько раз (зависит от того, сколько раз вызван .on()).
- **once** (eventName, listener) – вызов слушателя только один раз.
- **removeAllListeners** ([eventName]) – удаляет либо всех слушателей для всех событий объекта, либо всех слушателей для указанного события eventName.
- **removeListener** (eventName, listener) – удаление указанного слушателя для некоторого события с именем eventName.

Таймер

=

механизм, позволяющий генерировать событие или выполнить некоторое действие **через** заданный **промежуток времени**.

setTimeout(), setInterval()

- реализация находится в библиотеке `libuv`;
- `setTimeout` и `setInterval` относятся к **макротаскам**;
- коллбэки таймеров выполняются на **первой фазе** `eventloop`'а

setTimeout() выполняется **только один раз** через некоторый промежуток времени.

```
var start = Date.now();
var timer1 = setTimeout((p1,p2)=>{
    console.log(`timer1: p1 = ${p1}, p = ${p2}`);
    console.log('passed: ', Date.now()-start);
}, 3000, 'p1', 'p2');

var timer2 = setTimeout((p1,p2)=>{
    console.log(`timer1: p1 = ${p1}, p = ${p2}`);
    console.log('passed: ', Date.now()-start);
}, 4000, 'p3', 'p4');

var timer3 = setTimeout((p1,p2)=>{
    console.log(`timer1: p1 = ${p1}, p = ${p2}`);
    console.log('passed: ', Date.now()-start);
}, 5000, 'p5', 'p6');
```

```
D:\PSCA\Lec04>
D:\PSCA\Lec04>
D:\PSCA\Lec04>node 04-03
timer1: p1 = p1, p = p2
passed: 3013
timer1: p1 = p3, p = p4
passed: 4003
timer1: p1 = p5, p = p6
passed: 5003
D:\PSCA\Lec04>■
```

setInterval() выполняется **регулярно** через некоторый промежуток времени.

```
var start = Date.now();
var interval1 = setInterval((p1)=>{
    console.log(`interval1: p1 = ${p1}, passed = ${Date.now() - start}`);
}, 4000, 'параметр1');
```

```
D:\PSCA\Lec04>node 04-04
interval1: p1 = параметр1, passed = 4003
interval1: p1 = параметр1, passed = 8003
interval1: p1 = параметр1, passed = 12004
interval1: p1 = параметр1, passed = 16005
interval1: p1 = параметр1, passed = 20005
interval1: p1 = параметр1, passed = 24005
interval1: p1 = параметр1, passed = 28006
interval1: p1 = параметр1, passed = 32007
interval1: p1 = параметр1, passed = 36007
interval1: p1 = параметр1, passed = 40007
interval1: p1 = параметр1, passed = 44008
interval1: p1 = параметр1, passed = 48008
```

clearTimeout (timeoutObj) останавливает таймер, созданный с помощью setTimeout().

```
var start = Date.now();
var timer1 = setTimeout((p1)=>{
    console.log(`timer1: p1 = ${p1}`);
    console.log('passed: ', Date.now()-start);
}, 3000, 'p1');

var timer2 = setTimeout((p1)=>{
    console.log(`timer2: p1 = ${p1}`);
    console.log('passed: ', Date.now()-start);
}, 5000, 'p2');

var timer3 = setTimeout((p1)=>{
    console.log(`timer3: p1 = ${p1}`);
    console.log('passed: ', Date.now()-start);
}, 10000, 'p3');

var timer4 = setTimeout((p1)=>{
    console.log(`timer4: p1 = ${p1}`);
    console.log('passed: ', Date.now()-start);
    clearTimeout(timer2);
}, 4000, 'p4');
```

```
D:\PSCA\Lec04>node 04-05
timer1: p1 = p1
passed:  3013
timer4: p1 = p4
passed:  4003
timer3: p1 = p3
passed:  10002

D:\PSCA\Lec04>■
```

В качестве параметра передается **объект** ранее созданного с помощью setTimeout() **таймера**, который необходимо отменить.

clearInterval() останавливает таймер, созданный посредством setInterval().

```
var start = Date.now();
var interval1 = setInterval((p1)=>{
    console.log(`interval1: p1 = ${p1}, passed = ${Date.now() - start}`);
}, 4000, 'p1');

var timer4 = setTimeout((p1)=>{
    console.log(`timer4: p1 = ${p1}, passed = ${Date.now() - start}`);
    clearInterval(interval1);
}, 10000, 'p4');
```

В качестве параметра передается **объект** ранее созданного с помощью setTimeout() **таймера**, который необходимо отменить.

```
D:\PSCA\Lec04>node 04-06
interval1: p1 = p1, passed = 4003
interval1: p1 = p1, passed = 8005
timer4: p1 = p4, passed = 10002
```

```
D:\PSCA\Lec04>
```

!

Процесс Node.js работает до тех пор, **пока**
есть события, требующие обработки.

Если выполнить для таймера `unref()`, то события, генерируемые таймером, **не будут учитываться** при завершении работы Node.js

```
var start = Date.now();

var interval1 = setInterval((p1)=>{
    console.log(`interval1: p1 = ${p1}, passed = ${Date.now() - start}`);
    }, 3000, 'p1');

interval1.unref();
```

```
D:\PSCA\Lec04>
D:\PSCA\Lec04>
D:\PSCA\Lec04>
D:\PSCA\Lec04>node 04-07
D:\PSCA\Lec04>■
```

```
var start = Date.now();

var timer1 = setTimeout((p1)=>{
    console.log(`timer4: p1 = ${p1}, passed = ${Date.now() - start}`);
    clearInterval(interval1);
    }, 10000, 'p4');

var interval1 = setInterval((p1)=>{
    console.log(`interval1: p1 = ${p1}, passed = ${Date.now() - start}`);
    }, 3000, 'p1');

interval1.unref();
```

```
D:\PSCA\Lec04>node 04-07
interval1: p1 = p1, passed = 3003
interval1: p1 = p1, passed = 6015
interval1: p1 = p1, passed = 9017
timer4: p1 = p4, passed = 10001
D:\PSCA\Lec04>■
```

Метод ref() предназначен для противоположной операции.

```
var start = Date.now();

var timer1 = setTimeout(()=>{
  console.log(`timer1: passed = ${Date.now() - start}`);
}, 10000);

var timer2 = setTimeout((p1)=>{
  console.log(`timer2: passed = ${Date.now() - start}`);
  interval1.ref();
}, 14000);

var interval1 = setInterval((p1)=>{
  console.log(`interval1: passed = ${Date.now() - start}`);
}, 3000);
interval1.unref();
```

```
D:\PSCA\Lec04>node 04-08
interval1: passed = 3002
interval1: passed = 6014
interval1: passed = 9017
timer1: passed = 10003
interval1: passed = 12017
timer2: passed = 14002
interval1: passed = 15018
interval1: passed = 18019
interval1: passed = 21021
interval1: passed = 24023
interval1: passed = 27024
interval1: passed = 30026
interval1: passed = 33027
interval1: passed = 36028
```

```
const start = Date.now();
const http = require('http');

const server = http.createServer((req, res) => {
  res.end('<h1>server</h1>');
}).listen(3000, () => console.log('Server is running at http://localhost:3000'));

setTimeout(() => {
  console.log(`terminate, passed ${Date.now() - start}`);
  server.close(() => console.log('Server terminate'))
}, 20000);

const interval = setInterval(() => {
  console.log(`statistic calculation: passed = ${Date.now() - start}`);
}, 3000);

interval.unref();
```

```
PS D:\NodeJS\samples\cwp_04> node .\04-09.js
Server is running at http://localhost:3000
statistic calculation: passed = 3036
statistic calculation: passed = 6037
statistic calculation: passed = 9042
statistic calculation: passed = 12051
statistic calculation: passed = 15067
statistic calculation: passed = 18075
terminate, passed 20034
Server terminate
```

```
PS D:\NodeJS\samples\cwp_04> node .\04-09.js
Server is running at http://localhost:3000
statistic calculation: passed = 3013
statistic calculation: passed = 6026
statistic calculation: passed = 9027
statistic calculation: passed = 12034
statistic calculation: passed = 15039
statistic calculation: passed = 18050
terminate, passed 20024
statistic calculation: passed = 21059
statistic calculation: passed = 24068
statistic calculation: passed = 27074
statistic calculation: passed = 30081
statistic calculation: passed = 33081
Server terminate
PS D:\NodeJS\samples\cwp_04> □
```



Метод unref() есть **не только у таймеров**,
есть еще у серверов (server.unref()),
сетевых сокетов (socket.unref()) и др.

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

FILE SYSTEM

Файл

=

это специальный
именованный сегмент диска
для хранения информации.

Файловая система

=

это **набор спецификаций** и
соответствующее им
программное обеспечение,
которые отвечают за создание,
уничтожение, организацию,
чтение, запись, модификацию и
перемещение файлов, а также за
управление доступом к файлам
и за управление ресурсами,
которые используются файлами.

Популярные файловые системы:

- FAT12, FAT16, FAT32
- exFAT (или FAT64)
- NTFS
- EXT2/3/4
- APFS
- HFS+

Модуль FS

- позволяет взаимодействовать с файловой системой;
- предоставляет синхронные и асинхронные методы для работы с файловой системой;
- все операции с файловой системой имеют синхронные формы, а также асинхронные формы на основе callback и на основе promise.

```
const fs = require('fs/promises'); //promise-based
```

```
const fs = require('fs'); //synchronous, callback-based
```

Создание файла и запись в файл

```
const fs = require('fs'); // пакет для работы с файлами

// -- создание пустого файла
fs.open('./Files/File00.txt', 'w', (e, file)=>{
  if (e) throw e;
  console.log('Файл создан ');
});
```

fs.open (path, flags[, mode], callback)
flags – режим работы с файлом.
mode – режим файла (права доступа),
но только если файл был создан
В callback вторым параметром можно
получить дескриптор

```
// -- асинхронная перезапись/создание и запись в файл
let str01 = 'Строка 01\nСтрока 02\nСтрока 03\n';
fs.writeFile('./Files/File01.txt', str01, (e)=>{
  if (e) throw e;
  console.log('Запись в файл выполнена успешно');
});

// -- асинхронная запись в хвост/создание и запись в файл
let str01x = 'Строка 04\nСтрока 05\nСтрока 06\n';
fs.appendFile('./Files/File01.txt', str01x, (e)=>{
  if (e) throw e;
  console.log('Запись в файл выполнена успешно');
});
```

fs.writeFile(file, data[, options], callback)
fs.appendFile(file, data [, options], callback)

Чтение из файла (асинхронное)

```
const fs = require('fs'); // пакет для работы с файлами

fs.readFile('./Files/File01.txt', (e, data)=>{
  if (e) console.log('Ошибка: ', e);
  else  console.log('data: ', data.toString('utf8'));

});
```

fs.readFile(file [, options], callback)

В callback вторым параметром можно получить
считанные данные

Удаление файла

```
fs.unlink('./Files/File04.txt', (e)=>{
  if (e) console.log('Ошибка: ', e);
  else  console.log('Файл удален');
});
```

fs.unlink(path, callback)

Переименование файла

```
const fs = require('fs');

fs.rename('./Files/File04.txt', './Files/File04x.txt', (e)=> {
  if (e) console.log('Ошибка: ', e);
  else console.log('Файл переименован');
});
```

`fs.rename(oldPath, newPath, callback)`

В случае, если newPath уже существует, он будет перезаписан. Если в newPath указан каталог, будет выдана ошибка.

Копирование файла

```
// новый файл будет создан или перекроет существующий
fs.copyFile('./Files/File04x.txt', './Files/File04.txt', (e) => {
  if(e) console.log('Ошибка: ', e);
  else console.log('Файл скопирован');
})
```

fs.copyFile(src, dest[, mode], callback)

По умолчанию, dest перезаписывается, если он
уже существует.

Проверка наличия файла

```
const fs = require('fs');

fs.access('./Files/File04.txt', fs.constants.F_OK, (err) => {
  if (err) console.log('Файл не существует');
  else console.log('Файл существует');
});
```

fs.access(path[, mode], callback)

Проверяет права пользователя для файла или каталога, указанного в path. Аргумент mode – необязательное целое число, указывающее, какие проверки доступности необходимо выполнить.

Работа с директориями (создание, переименование)

```
const fs = require('fs');

// создание директории
fs.mkdir('./Files/NewDir', { recursive: true }, (err) => {
  if (err) console.log(err);
  else console.log('Директория создана');
});

// переименование директории
fs.rename('./Files/NewDir', './Files/MyDir', (err) => {
  if (err) console.log(err);
  else console.log('Директория переименована');
});
```

fs.mkdir(path[, options], callback)

Необязательный аргумент options может содержать свойство recursive указывающее, следует ли создавать родительские каталоги.

fs.rename(oldPath, newPath, callback)

В случае, если newPath уже существует, она будет перезаписана.

Работа с директориями (удаление, чтение содержимого)

```
// удаление директории
fs.rmdir('./Files/MyDir', { recursive: true }, (err) => {
  if (err) console.log(err);
  else console.log('Директория удалена');
});
```

fs.rmdir(path[, options], callback)

Аргумент options может содержать свойство recursive. В рекурсивном режиме операции повторяются при сбое.

Можно также указать maxRetries и retryDelay, если recursive = true.

Работа с директориями (чтение содержимого)

```
// чтение содержимого директории
fs.readdir('./Files', { withFileTypes: true }, (err, files) => {
  if (err) console.log(err);
  else console.log('Содержимое директории: ', files);
});
```

fs.readdir(path[, options], callback)

Вторым параметром можно получить массив имен файлов в каталоге.

Необязательный аргумент *options* может быть строкой или объектом, определяющим кодировку, используемую для имен файлов.

```
PS D:\NodeJS\samples\cwp_09> node 09-02
Содержимое директории: [
  Dirent { name: 'File04.txt', [Symbol(type)]: 1 },
  Dirent { name: 'File21.txt', [Symbol(type)]: 1 },
  Dirent { name: 'inFile.txt', [Symbol(type)]: 1 },
  Dirent { name: 'MyDir', [Symbol(type)]: 2 },
  Dirent { name: 'outFile.txt', [Symbol(type)]: 1 },
  Dirent { name: 'тест.txt', [Symbol(type)]: 1 }
]
```

```
PS D:\NodeJS\samples\cwp_09> node 09-02
Содержимое директории: [
  'File04.txt',
  'File21.txt',
  'inFile.txt',
  'MyDir',
  'outFile.txt',
  'тест.txt'
]
```

Если для параметра **options.withFileTypes** установлено значение *true*, массив файлов будет содержать объекты *fs.Dirent*

Слежка за файлом/директорией

```
const fs = require('fs');

const filename='./Files/File01.txt';

try{
  fs.watch(filename, (event, f) => {           // следить за файлом
    if (f) console.log(`file: ${f}, event = ${event}`);
  });
}
catch(e){
  console.log('catch e = ', e.code);
}
```

fs.watch (filename[, options], listener)

В обработчик будет приходить первым аргументом название события: изменен этот файл или переименован (создан, удален). Таким же образом мы можем следить не только за файлами, но и за директориями.

Аргумент options можно опустить. Объект options может содержать логическое значение со свойством *recursive* (отслеживать ли все подкаталоги, для каталогов), *encoding* (кодировка символов для имени файла, передаваемого слушателю)

```
const fs = require('fs');
const dirname='./Files';

try{
  fs.watch(dirname, (event, f) => {           // следить папкой
    if (f) console.log(`folder: ${f}, event = ${event}`);
  });
}
catch(e){
  console.log('catch e = ', e.code);
}
```

Поток данных =
Stream

это **абстракция над данными**,
используемая для
чтения/записи файлов, сокетов
и др.

Виды потоков в Node.js

- **Readable** – поток, который предоставляет данные на чтение.

Примеры: request на сервере, response на клиенте, поток чтения fs, process.stdin

- **Writable** – поток, в который данные можно записывать.

Примеры: response на сервере, request на клиенте, поток записи fs, process.stdout/stderr

- **Duplex** – поток, из которого можно как читать данные (Readable), так и записывать в него (Writable), при этом процесс чтения и записи происходит независимо друг от друга.

Примеры: socket, websocket stream

- **Transform** – разновидность Duplex потоков, которые могут изменять данные при их записи и чтении в/из потока.

Примеры: zlib, crypto

Режимы работы Readable потока

1. Автоматический (.pipe()), события data и end)

```
const fs = require('fs');

let i = 0;
fs.createReadStream('./Files/File14.txt') // поток для чтения
  .on('data', (chunk) => { // в буфере потока есть данные
    console.log(`--${i++}-- ${chunk.length}`);
  })
  .on('end', () => { // прочитаны все данные
    console.log('--end--');
  })
  .on('error', (err) => { // обработка ошибок
    console.log(`error: ${err}`);
  })
}
```

```
PS D:\NodeJS\samples\cwp_09> node .\09-10.js
--0-- 65536
--1-- 488
--end--
```

```
const fs = require('fs');
let readStream = fs.createReadStream('./Files/File14.txt');
let writeStream = fs.createWriteStream('./Files/File22.txt');
readStream.pipe(writeStream);
```

Режимы работы Readable потока

2. Пошаговый или приостановленный (событие readable)

```
const fs = require('fs');

let rstream = fs.createReadStream('./Files/File14.txt'); // поток для чтения

rstream.on('readable', ()=>{ // можно читать

  let chunk = null;
  while ((chunk = rstream.read()) != null) {
    console.log(chunk.toString());
  }
})

rstream.on('error', (e)=>{ // обработка ошибок
  console.log('error = ', e);
})
```

```
rstream.on('readable', ()=>{

  let chunk = null;
  while ((chunk = rstream.read(5)) != null) {
    console.log(chunk.toString());
  }
})
```

Writable поток

1.

```
const fs = require('fs');

let ws = fs.createWriteStream('./Files/File21.txt');

for (let i = 0; i < 10; i++) {
  ws.write(`---${i}---`);
}
ws.end(); // сигнализирует, что данных больше не будет, закрытие потока

// ws.write(`last chunk`); // ERR_STREAM_WRITE_AFTER_END: запись больше невозможна
```



Files > File21.txt
1 ---0-----1-----2-----3-----4-----5-----6-----7-----8-----9---|

2.

```
const fs = require('fs');
let readStream = fs.createReadStream('./Files/File14.txt');
let writeStream = fs.createWriteStream('./Files/File22.txt');
readStream.pipe(writeStream);
```

Duplex поток (Readable + Writable)

```
const net = require('net');

const server = net.createServer(socket => {
  socket.on('data', data => {
    socket.write('ECHO: ' + data.toString());
  });

  socket.on('error', error => {
    console.log(error);
  });
});

server.listen(3000, () => console.log('Server is running'));
```

```
const net = require('net');

const client = net.createConnection({ port: 3000, host: '127.0.0.1' }, () => {
  client.write('Hello, server!');
});

client.on('data', data => {
  console.log(`Received data: ${data.toString()}`);
});

client.on('error', error => {
  console.log(error);
});
```

```
PS D:\NodeJS\samples\cwp_09> node 09-11s
Server is running
[]
```

```
PS D:\NodeJS\samples\cwp_09> node 09-11c
Received data: ECHO: Hello, server!
[]
```

Transform поток

```
const fs = require('fs');
const zlib = require('zlib');

const input = fs.createReadStream('./Files/input.txt');

const gzip = zlib.createGzip(); // transform stream

const output = fs.createWriteStream('./Files/input.txt.gz');
input.pipe(gzip).pipe(output);
```

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

МОДУЛИ. NPM. PACKAGE.JSON

Модуль

=

текстовый файл, содержащий код на языке JS, специальным образом оформленный и размещенный. Может использоваться приложением. Является фундаментальной единицей структурирования кода Node.js-приложений.

Форматы модулей

- IIFE (Immediately Invoked Function Expression)
- CommonJS
- AMD (Asynchronous Module Definition)
- UMD (Universal Module Definition)
- ES6 Module

IIFE =
(Immediately Invoked Function
Expression)

ЭТО КОНСТРУКЦИЯ, ПОЗВОЛЯЮЩАЯ
ВЫЗЫВАТЬ ФУНКЦИЮ
НЕПОСРЕДСТВЕННО ПОСЛЕ ЕЕ
ОПРЕДЕЛЕНИЯ.

```
(function() {  
    //Your code goes here  
})()
```

```
(( ) => {  
    //Your code goes here  
})()
```

CommonJS =

группа, которая проектирует, прототипирует и стандартизирует различные JavaScript API.

CommonJS

- поддержка `require` для импорта модуля;
- **имя модуля – строка**, может включать символы идентификации путей;
- модуль должен явно экспортировать всю свою функциональность, поддержка объекта `export`;
- **Переменные, объявленные внутри модуля, не видны за его пределами.**

require

- **сихронно** загружает модуль;
- кэширует модуль;
- удалить из кэша можно с помощью **delete require.cache[...]**;
- если модуль удален, то для его использования нужен **новый require**.

```
const http = require('http'); // предоставить доступ к базовому модулю http
```

```
const myModule = require('./module'); // предоставить доступ к небазовому модулю
```

Алгоритм поиска импортируемого модуля

- 1) Проверяется, есть ли **встроенный модуль с указанным именем**. Если есть, импортирует его.
- 2) Если в функцию `require` передали **путь** до модуля, то импортируется модуль по указанному пути.
- 3) Модуль или пакет ищется в специальной директории **`node_modules`** по восходящему принципу. После осуществляется поиск среди глобальных пакетов.

```
console.log('http -->', require.resolve('http'));           // модуль базовый
console.log('../Lec04/m04-02 -->', require.resolve('../Lec04/m04-02')); // модуль установлен вручную
console.log('async-->', require.resolve('async'));           // модуль установлен с помощью прм

// set NODE_PATH=%AppData%\npm\node_modules - для windows 10 (глобально-установленные модули)

// npm install async      установка локально      %AppData%\npm\node_modules
// npm install -g async    установка глобально
// npm uninstall async     удалить локальный пакет async
// npm uninstall -g async  удалить глобальный пакет async
```

```
D:\PSCA\Lec05>node 05-03
http --> http
../Lec04/m04-02 --> D:\PSCA\Lec04\m04-02.js
async--> C:\Users\Win10_ISiT_Server\AppData\Roaming\npm\node_modules\async\dist\async.js

D:\PSCA\Lec05>
D:\PSCA\Lec05>
```

Если в качестве имени указана папка, то
дополнительная информация в файле package.json

```
  ,
  "homepage": "https://caolan.github.io/async/",
  "keywords": [
    "async",
    "callback",
    "module",
    "utility"
  ],
  "license": "MIT",
  "main": "dist/async.js",
  "module": "dist/async.mjs",
  "name": "async",
  "nyc": {
    "exclude": [
      "test"
    ]
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/caolan/async.git"
  },
  
```

Содержимое функции require

```
D:\PSCA\Lec05>node 05-04
{ [Function: require]
  resolve: { [Function: resolve] paths: [Function: paths] },
  main:
    Module {
      id: '.',
      exports: {},
      parent: null,
      filename: 'D:\\PSCA\\Lec05\\05-04.js',
      loaded: false,
      children: [],
      paths:
        [ 'D:\\PSCA\\Lec05\\node_modules',
          'D:\\PSCA\\node_modules',
          'D:\\node_modules' ] },
    extensions:
      [Object: null prototype] { '.js': [Function], '.json': [Function], '.node': [Function] },
    cache:
      [Object: null prototype] {
        'D:\\PSCA\\Lec05\\05-04.js':
          Module {
            id: '.',
            exports: {},
            parent: null,
            filename: 'D:\\PSCA\\Lec05\\05-04.js',
            loaded: false,
            children: [],
            paths: [Array] } } }
```

Кэширование модуля

```
JS module.js > ...
1  var num = 10;
2
3  function multiply(a) {
4      return num * a;
5  }
6
7  module.exports = {
8      ...
9      num: num,
10     multiply: multiply
11 }
```

```
JS index.js > ...
1  var myModule = require('./module');
2  console.log(myModule.multiply(2));
3
4  var myModule = require('./module');
5  console.log(myModule.multiply(2));
6
7
```

```
JS module.js > ...
1  //var num = 10;
2  var num = 5;
3
4  function multiply(a) {
5      return num * a;
6  }
7
8  module.exports = {
9      ...
10     num: num,
11     multiply: multiply
12 }
```

```
C:\Program Files\nodejs\node.exe --inspect-brk=23066 index.js
Debugger listening on ws://127.0.0.1:23066/542e9a87-1275-4bec-b24f-2f98576525f6
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
20
20
```

Удаление из кэша

```
index.js
JS index.js > [0] myModule
  1  var myModule = require("./module");
  2  console.log(myModule.multiply(2));
  3
  4  //delete require('./module');           // не работает
  5  //delete require.cache['./module'];     // не работает
  6  console.log(require.resolve('./module'));
  7  delete require.cache[require.resolve('./module')]; // ок
  8  //delete require(require.resolve('./module')); // не работает
  9
 10 var myModule = require('./module');
 11 console.log(myModule.multiply(2));
 12
 13

module.js
JS module.js > ...
  1  //var num = 10;
  2  var num = 5;
  3
  4  function multiply(a) {
  5    return num * a;
  6  }
  7

require: function require(path) { ... }
  [[FunctionLocation]]: internal#location
  > [[Scopes]]: Scopes[3]
    arguments: TypeError: 'caller', 'callee', ...
  > cache: Object {d:\Материалы\cwp_04\index.js: Module}
    > d:\Материалы\cwp_04\index.js: Module {id: '.', ...
    > d:\Материалы\cwp_04\module.js: Module {id: "d:\...
      caller: TypeError: 'caller', 'callee', and 'argu...
    > extensions: Object {.js: , .json: , .node: }
      length: 1
    > main: Module {id: ".", path: "d:\Материалы\cwp_0...
      name: "require"
    > prototype: Object {constructor: }
    > resolve: function resolve(request, options) { ... }
    > __proto__: function () { ... }

C:\Program Files\nodejs\node.exe --inspect-brk=7876 index.js
Debugger listening on ws://127.0.0.1:7876/6f9f6c24-a6e2-4112-b91e-bda46e604a03
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
20
d:\Материалы\cwp_04\module.js
10
```

Пример

```
//---- m05-09.js ----
var c = 0;
module.exports = ()=>{return ++c;}
```

```
//---- 05-09.js ----
var count1 = require('./m05-09');
var count2 = require('./m05-09');

console.log('count1 = ', count1());
console.log('count2 = ', count2());
console.log('count1 = ', count1());
console.log('count2 = ', count2());
```

Результат выполнения еще раз доказывает, что модули **кэшируются после первого раза, когда они загружаются.**

```
D:\PSCA\Lec05>node 05-09
count1 = 1
count2 = 2
count1 = 3
count2 = 4

D:\PSCA\Lec05>■
```

exports

```
var x = 1;
global.y = 2;
process.z = 3;

console.log('---- m05-05 -----')
console.log('m05-05.x = ', x); // 1
console.log('m05-05.y = ', y); // 2
// console.log('m05-05.z = ', z); // error

console.log('-----')
console.log('m05-05.global.x = ', global.x); // undefined
console.log('m05-05.global.y = ', global.y); // 2
console.log('m05-05.global.z = ', global.z); // undefined

console.log('-----')
console.log('m05-05.process.x = ', process.x); // undefined
console.log('m05-05.process.y = ', process.y); // undefined
console.log('m05-05.process.z = ', process.z); // 3

function Op(op1, op2){
  this.op1 = op1;
  this.op2 = op2;
  this.add = ()=>{return this.op1+this.op2;};
  this.sub = ()=>{return this.op1-this.op2;};
}

exports.X = x;
exports.Y = y;
exports.OP = Op;
```

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других файлах.

```
var m05 = require('./m05-05');

console.log('----05-05-----');
console.log('05-05.global.x = ', global.x); // undefined
console.log('05-05.global.y = ', global.y); // 2
console.log('05-05.process.z = ', process.z); // 3

console.log('-----');
console.log(' (new m05.OP(1,2)).add() = ', (new m05.OP(1,2)).add()); // 3
console.log(' (new m05.OP(1,2)).sub() = ', (new m05.OP(1,2)).sub()); // -1
console.log(' m05.X = ', m05.X); // 1
console.log(' m05.Y = ', m05.Y); // 2
console.log('-----');
global.y = 22;
console.log('05-05.global.y = ', global.y); // 22
console.log(' m05.Y = ', m05.Y); // 2
```

module.exports vs exports

```
var num = 5;

// var exports = module.exports = {};
exports.multiply = function (a) {
  return num * a;
}

module.exports = num;
```

exports – это просто **ссылка** на module.exports.

```
var myModule = require('./module');

console.log(myModule.multiply(2)); // Error: myModule.multiply is not a function
console.log(myModule);           // 5
```

ES6 Module

- официальный **стандартный формат** упаковки кода Javascript;
- операторы **import** и **export**;
- по умолчанию в Node.js используется формат CommonJS, поэтому, чтобы разрешить использование ES6, нужно:
 - 1) указать расширение файла **.mjs**;
 - или
 - 2) указать поле **“type”: “module”** в package.json.

Enabling

#

Node.js treats JavaScript code as CommonJS modules by default. Authors can tell Node.js to treat JavaScript code as ECMAScript modules via the .mjs file extension, the package.json “type” field, or the --input-type flag. See Modules: Packages for more details.

Экспорт по умолчанию до объявления

```
var num = 10;

export default function multiply(a) {
  return num * a;
}
```

Экспорт по умолчанию после объявления

```
var num = 10;

function multiply(a) {
  return num * a;
}
export {multiply as default};
```

Модули могут иметь **только один** экспорт по умолчанию.

Импорт экспорта по умолчанию

```
import myModule from './module.js';
console.log(myModule(2));
```

При импорте экспорта по умолчанию ему можно
дать **любое имя** и **не нужно использовать**
фигурные скобки.

Именованный экспорт до объявления

```
// export class  
// export const  
// export let  
export function multiply(a) {  
    return 10 * a;  
}
```

Именованный экспорт после объявления

```
var num = 5;  
  
function multiply(a) {  
    return num * a;  
}  
  
export {  
    multiply, num  
};
```

Именованные экспорты
позволяют выполнять **несколько**
экспортов в одном файле.

Импорт именованного экспорта

```
import { multiply as mul } from './module.js';
console.log(mul(2));
```

Можно переименовать экспорт **псевдонимом**, если есть коллизии в файле.

```
import {multiply, num} from './module.js';

console.log(multiply(2));
console.log(num);
```

Когда нужно импортировать именованный компонент, **нужно использовать то же имя, с которым он был экспортирован**. Имена должны быть импортированы **внутри фигурных скобок**.

Можно включить в импорт **несколько** компонентов.

```
import * as myModule from './module.js';

console.log(myModule.multiply(2));
console.log(myModule.num);
```

Можно **импортировать все** именованные экспорты в объект.

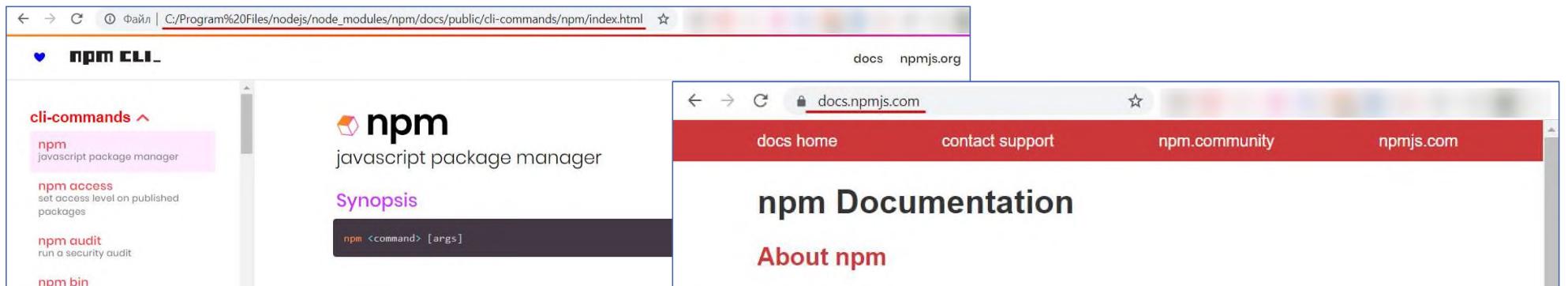
Динамический import

```
var num = 5;  
  
function multiply(a) {  
    return num * a;  
}  
  
export {  
    multiply, num  
};
```

```
import('./module3.js')  
    .then((module) => {  
        console.log(module.multiply(2));  
        console.log(module.num);  
    });
```

NPM (Node Package Manager)

- устанавливается вместе с Node.js;
- предназначен **для скачивания/публикации пакетов**;
- инструмент командной строки;
- глобальное хранилище <https://registry.npmjs.org/>
- <https://www.npmjs.com/>



Пакет

=

один или несколько js-файлов
и файл-манифест `package.json`.

Просмотр локальных пакетов

ls, list, la, ll

```
D:\PSCA\Lec05>npm list
D:\PSCA\Lec05
`-- (empty)
```

```
D:\PSCA\Lec05>npm ls
D:\PSCA\Lec05
`-- (empty)
```

```
D:\PSCA\Lec05>npm la
| D:\PSCA\Lec05
|
`-- (empty)
```

```
D:\PSCA\Lec05>npm ll
| D:\PSCA\Lec05
|
`-- (empty)
```

Скачивание install, i, add

```
D:\PSCA\Lec05>npm install nodemailer
npm WARN saveError ENOENT: no such file or directory, open 'D:\PSCA\Lec05\package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open 'D:\PSCA\Lec05\package.json'
npm WARN Lec05 No description
npm WARN Lec05 No repository field.
npm WARN Lec05 No README data
npm WARN Lec05 No license field.

+ nodemailer@6.3.0
added 1 package from 1 contributor and audited 1 package
found 0 vulnerabilities
```

Имя	Дата изменения	Тип	Размер
lib	19.08.2019 23:05	Папка с файлами	
.DS_Store	26.10.1985 11:15	Файл "DS_STORE"	9 КБ
.prettierrc.js	26.10.1985 11:15	JetBrains WebStorm	1 КБ
CHANGELOG.md	26.10.1985 11:15	Файл "MD"	21 КБ
CONTRIBUTING.md	26.10.1985 11:15	Файл "MD"	5 КБ
LICENSE	26.10.1985 11:15	Файл	1 КБ
package.json	19.08.2019 23:05	JSON File	2 КБ
README.md	26.10.1985 11:15	Файл "MD"	5 КБ

Удаление uninstall, remove, rm, r, un, unlink

```
D:\PSCA\Lec05>npm uninstall nodemailer
npm WARN saveError ENOENT: no such file or directory, open 'D:\PSCA\Lec05\package.json'
npm WARN enoent ENOENT: no such file or directory, open 'D:\PSCA\Lec05\package.json'
npm WARN Lec05 No description
npm WARN Lec05 No repository field.
npm WARN Lec05 No README data
npm WARN Lec05 No license field.

removed 1 package in 1.473s
found 0 vulnerabilities
```

```
D:\PSCA\Lec05>npm list
D:\PSCA\Lec05
`-- (empty)
```

Глобальные пакеты

-g

```
D:\PSCA\Lec05>npm list -g
C:\Users\Win10_ISiT_Server\AppData\Roaming\npm
+-- async@3.1.0
`-- bower@1.8.8

D:\PSCA\Lec05>npm ls -g
C:\Users\Win10_ISiT_Server\AppData\Roaming\npm
+-- async@3.1.0
`-- bower@1.8.8

D:\PSCA\Lec05>npm la -g
| C:\Users\Win10_ISiT_Server\AppData\Roaming\npm
|
+-- async@3.1.0
|   Higher-order functions and common patterns for asynchronous code
|   git+https://github.com/caolan/async.git
|   https://caolan.github.io/async/
`-- bower@1.8.8
  The browser package manager
  git+https://github.com/bower/bower.git
  http://bower.io

D:\PSCA\Lec05>npm ll -g
| C:\Users\Win10_ISiT_Server\AppData\Roaming\npm
|
+-- async@3.1.0
|   Higher-order functions and common patterns for asynchronous code
|   git+https://github.com/caolan/async.git
|   https://caolan.github.io/async/
`-- bower@1.8.8
  The browser package manager
  git+https://github.com/bower/bower.git
  http://bower.io
```

```
D:\PSCA\Lec05>npm install nodemailer -g
+ nodemailer@6.3.0
added 1 package from 1 contributor in 0.495s
```

```
D:\PSCA\Lec05>npm list
D:\PSCA\Lec05
`-- (empty)
```

```
D:\PSCA\Lec05>npm list -g
C:\Users\Win10_ISiT_Server\AppData\Roaming\npm
+-- async@3.1.0
+-- bower@1.8.8
`-- nodemailer@6.3.0
```

Информация о пакете view, info, show, v

```
D:\PSCA\Lec05>npm view nodemailer
nodemailer@6.3.0 | MIT | deps: none | versions: 219
Easy as cake e-mail sending from your Node.js applications
https://nodemailer.com/
keywords: Nodemailer
dist
.tarball: https://registry.npmjs.org/nodemailer/-/nodemailer-6.3.0.tgz
.shasum: a89b0c62d3937bdcdeecbf55687bd7911b627e12
.integrity: sha512-TEHBNBPHv7Ie/0o3HXnb7xrPSSQmH1dXwQKRaMKDBGt/2
.unpackedSize: 478.3 kB
maintainers:
- andris <andris@node.ee>
dist-tags:
beta: 2.4.0-beta.0  latest: 6.3.0
published a month ago by andris <andris@kreataa.ee>
```

```
D:\PSCA\Lec05>npm view request
request@2.88.0 | Apache-2.0 | deps: 20 | versions: 125
Simplified HTTP request client.
https://github.com/request/request#readme
keywords: http, simple, util, utility
dist
.tarball: https://registry.npmjs.org/request/-/request-2.88.0.tgz
.shasum: 9c2fc4f7d35b592efe57c7f0a55e81052124fef
.integrity: sha512-NAqBSrijGLZdM0WZNsInLJpkJokL72XYjUpnB0iwsRgxh
.unpackedSize: 206.9 kB
dependencies:
aws-sign2: ~0.7.0          extend: ~3.0.2          http-signature: ~1.0.0
aws4: ^1.8.0                forever-agent: ~0.6.1    is-typed: ~1.0.0
caseless: ~0.12.0           form-data: ~2.3.2      isstream: ~2.0.0
combined-stream: ~1.0.6     har-validator: ~5.1.0    json-stringify-safe: ~5.0.1
maintainers:
- fredkschott <fkschott@gmail.com>
- mikeal <mikeal.rogers@gmail.com>
- nylen <jnylen@gmail.com>
- simov <simeonvelichkov@gmail.com>
dist-tags:
latest: 2.88.0
published a year ago by mikeal <mikeal.rogers@gmail.com>
```

Поиск пакета search, s, se, find

NAME	DESCRIPTION	AUTHOR	DATE	VERSION	KEYWORDS
request	Simplified HTTP...	=fredkschott...	2018-08-10	2.88.0	http simple util ut
request-promise-native	The simplified HTTP...	=analog-nico...	2019-02-15	1.0.7	xhr http https prom
got	Simplified HTTP...	=sindresorhus...	2019-01-17	9.6.0	http https get got
request-promise	The simplified HTTP...	=analog-nico...	2019-02-15	4.2.4	xhr http https prom
request-promise-core	Core Promise...	=analog-nico	2019-02-14	1.1.2	xhr http https prom
request-progress	Tracks the download...	=satazor	2016-12-01	3.0.0	progress request mi
morgan	HTTP request logger...	=dougwilson	2018-09-11	1.9.1	express http logger
cacheable-request	Wrap native HTTP...	=lukechilds	2019-06-07	6.1.0	HTTP HTTPS cache ca
superagent	elegant & feature...	=defunctzombie...	2019-06-15	5.1.0	agent ajax ajax api
teeny-request	Like request , but...	=fhinkel...	2019-08-14	5.2.1	request node-fetch
request-ip	A small node.js...	=pbojinov	2018-10-29	2.1.3	request ip ip addre
Octokit/request	Send parameterized...	=bkeepers...	2019-07-25	5.0.2	octokit github api
cookie-parser	Parse HTTP request ...	=defunctzombie...	2019-02-13	1.4.4	cookie middleware
simple-get	Simplest way to...	=feross	2018-08-09	3.0.3	request http GET ge
type-is	Infer the...	=dougwilson...	2019-04-26	1.6.18	content type checki
raf	request AnimationFram...	=chrisdickinson...	2018-11-02	3.4.1	request AnimationFr
proxy-addr	Determine address...	=dougwilson	2019-04-16	2.0.5	ip proxy x-forwarded
timed-out	Emit 'ETIMEDOUT' or...	=floatdrop...	2017-01-16	4.0.1	http https get got
retry-request	Retry a request .	=stephenpluspl...	2019-06-18	4.1.1	request retry strea
co-body	request body...	=dead_horse...	2018-05-21	6.0.0	request parse parse

Nodemailer

```
PS D:\NodeJS\samples\cwp_05\mail_nodemailer> npm install nodemailer  
added 1 package, and audited 10 packages in 828ms
```

```
PS D:\NodeJS\samples\cwp_05\mail_nodemailer> npm install nodemailer-smtp-transport  
added 8 packages, and audited 10 packages in 2s
```

Nodemailer

```
const nodemailer = require('nodemailer');
const smtpTransport = require('nodemailer-smtp-transport');
const sender = 'dubovik.m14@gmail.com';
const receiver = 'dubovik.m14@gmail.com';
const pass = '*****';

function send(message) {
  let transporter = nodemailer.createTransport(smtpTransport({
    host: 'smtp.gmail.com',
    port: 587,
    secure: false,
    auth: {
      user: sender,
      pass: pass
    }
  }));
  var mailOptions = {
    from: sender,
    to: receiver,
    subject: 'Lab6',
    text: message,
    html: `<i>${message}</i>`,
    attachments: [{ filename: 'cat.jpg', path: __dirname + '/cat.jpg', cid: 'img' }]
  };
  transporter.sendMail(mailOptions, function (error, info) {
    error ? console.log(error) : console.log('Email sent: ' + info.response);
  });
  send('Test message');
}
```

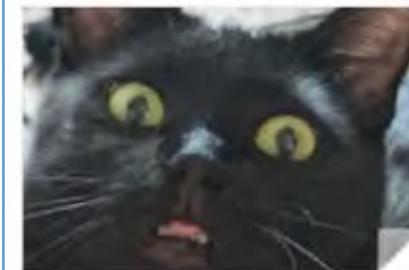
```
PS D:\NodeJS\samples\cwp_05\mail_nodemailer> node 05-02
Email sent: 250 2.0.0 OK 1665175261 k13-20020adff28d000000b0022ac672654dsm2780957wro.58 - gsmtp
PS D:\NodeJS\samples\cwp_05\mail_nodemailer> |
```

Lab6 Входящие

dubovik.m14@gmail.com

КОМУ: мне ▾

Test message



Обработка параметров с формы

```
var http  = require('http');
var fs    = require('fs');
var url   = require('url');
const { parse } = require('qs');

let http_handler = (req, resp)=>{

    resp.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    if      (url.parse(req.url).pathname == '/' && req.method == 'GET' ) {
        resp.end(fs.readFileSync('./05-07.html'));
    }else if (url.parse(req.url).pathname == '/' && req.method == 'POST' ) {
        let body = '';
        req.on('data', chunk => {body += chunk.toString();});
        req.on('end', () => {
            let parm = parse(body);
            resp.end(`<h1>OK: ${parm.receiver}, ${parm.sender}, ${parm.message} </h1>`)
        } else resp.end('<h1>Not support</h1>');
    }

    let server = http.createServer(http_handler);
    server.listen(3000);
    console.log('Server running at http://localhost:3000/');
}
```

Обработка параметров с формы

The image displays three screenshots of a web application interface, likely a browser window, showing the process of handling form data.

Screenshot 1 (Left): A form titled "Lec 05" with three fields: "Отправитель" (box@mailserver), "Получатель" (box@mailserver), and "Сообщение" (Выполнил). A "OK" button is visible at the bottom right.

Screenshot 2 (Middle): The same "Lec 05" form, but the "Получатель" field now contains "smw@belstu.by". The "OK" button is visible at the bottom right.

Screenshot 3 (Bottom): A new window or message box with the text "OK: smw60@mail.ru, smw@belstu.by, Выполнил Смелов В.В." displayed.

Порядок публикации пакета

1. Регистрация на <https://www.npmjs.com/>
2. Добавление учетной записи пользователя (adduser, login). Потребуется указать name, password, email

```
D:\PSCA\Lec05>npm adduser
Username: smw60■
```

3. Проверка (отобразить имя пользователя)

```
D:\PSCA\Lec05>npm whoami
smw60
```

Порядок публикации пакета

4. Инициализация проекта (init)

```
D:\PSCA\Lec05\node_modules\p0508>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
See 'npm help json' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg>' afterwards to install a package and
save it as a dependency in the package.json file.

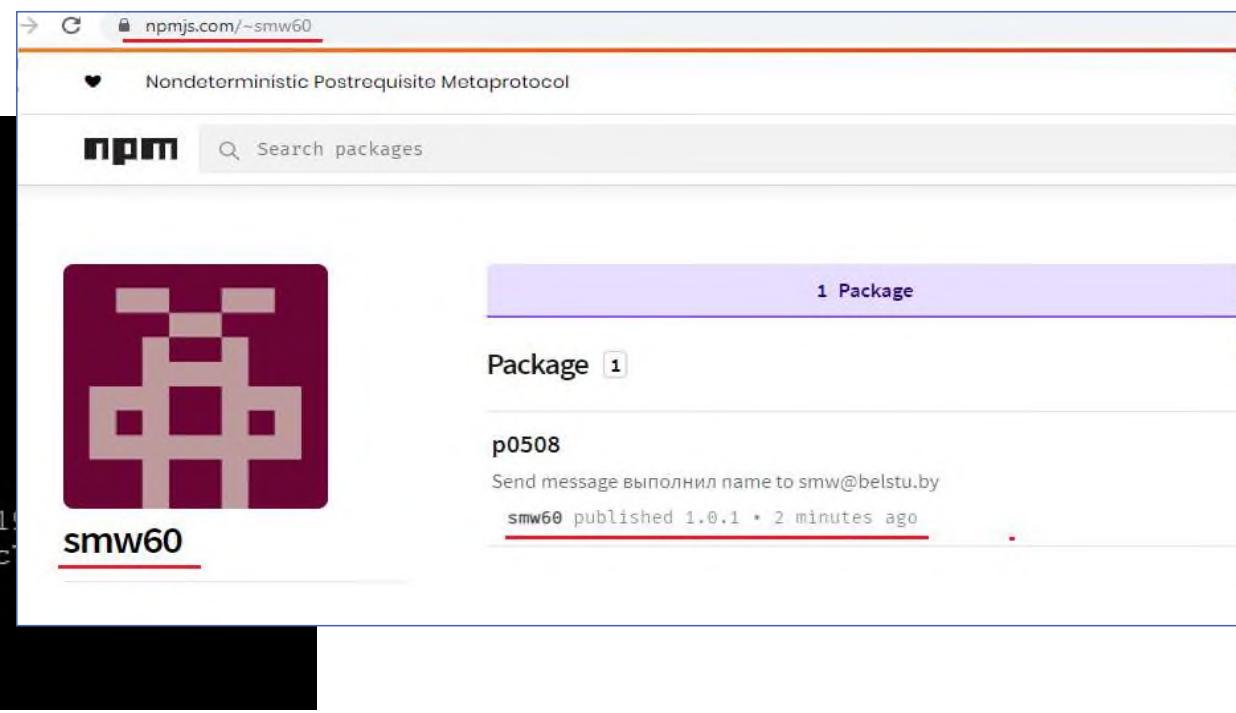
Press ^C at any time to quit.
package name: (p0508) p0508
version: (1.0.0) 1.0.1
{
  "dependencies": {"sendmail": "1.6.1"},
  "description": "send message выполнил name to smw@belstu.by",
  "keyword": "send name in email-message to smw@belstu.by without smtp",
  "main": "index.js",
  "maintainers": [ {"name": "smw60", "email": "smw60@mail.ru"} ],
  "name": "p0508",
  "publishConfig": { "access": "public" },
  "version": "1.0.1"
}
```

Имя	Дата изменения	Тип	Размер
node_modules	20.08.2019 15:06	Папка с файлами	
index.js	20.08.2019 16:41	JetBrains WebStorm	1 КБ
package.json	20.08.2019 16:40	JSON File	1 КБ

Порядок публикации пакета

5. Публикация пакета: `publish (--access=public, если есть scope)`

```
D:\PSCA\Lec05\node modules\p0508>npm publish
npm notice
npm notice package: p0508@1.0.1
npm notice === Tarball Contents ===
npm notice 419B package.json
npm notice 751B index.js
npm notice === Tarball Details ===
npm notice name: p0508
npm notice version: 1.0.1
npm notice package size: 672 B
npm notice unpacked size: 1.2 kB
npm notice shasum: d5c364d4b7c759d004714fa9219...
npm notice integrity: sha512-tVkyYT+H5urHG[...]c
npm notice total files: 2
npm notice
+ p0508@1.0.1
```



Файл package.json =

файл конфигурации приложения Node.js. Любая директория, содержащая данный файл, интерпретируется как Node.js-пакет.

Содержит метаданные проекта (название, версия, описание проекта, ...), список зависимостей вашего проекта, которые будут установлены при вызове команды `npm install`, скрипты, вызывающие другие команды консоли.

Не забывайте всегда добавлять **папку node_modules** в **.gitignore**, т.к. папка может весить гигабайты.



Создание package.json

1. Создание вручную
2. Создание с помощью команды npm init

```
PS D:\Материалы\cwp_04> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (cwp_04) test
version: (1.0.0)
description: Testing npm package creation
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```



```
{
  "name": "test",
  "version": "1.0.0",
  "description": "Testing npm package creation",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Содержимое package.json

- **name** – имя пакета. Должно быть короче 214 символов, состоять только из букв в нижнем регистре, цифр, символов подчеркивания (_) и дефисов (-), без использования пробелов.
- **version** – текущая версия пакета. Должна соответствовать правилам семантического версионирования (X.Y.Z).
- **description** – краткое описание пакета. Если планируется публикация пакета, данное свойство поможет пользователям сайта прт понять назначение пакета.

Содержимое package.json

- **main** – точка входа в проект.
- **scripts** – набор скриптов, которые можно запускать средствами npm. Запуск производится через консоль командами вида npm run XXX или yarn XXX, где XXX – имя скрипта (в примере – “test”).
- **author** – информация об авторе пакета (строка/json-объект). Может содержать имя автора, email и адрес сайта.

```
"author": "John Wick <jw@mail.com> (https://revengeforthedog.com)",
"author": {
  "name": "John Wick",
  "email": "jw@mail.com",
  "url": "https://revengeforthedog.com"
},
```

Содержимое package.json

- **license** – информация о лицензии пакета.
- **keywords** – массив ключевых слов, относящихся к функционалу проекта. Правильно подобранные ключевые слова могут помочь людям быстрее найти нужный пакет при просмотре сайта npm.

```
"keywords": [  
  "test",  
  "email",  
  "package management"  
]
```

Содержимое package.json

- **dependencies** – список всех пакетов, которые используются при работе приложения. Команда `npm install <package_name>` добавит пакет в список. Команда `npm install` загрузит все зависимости, перечисленные в данной секции.
- **devDependencies** – информация о пакетах, которые используются в процессе разработки, но не используются при работе самого приложения. Команда `npm install <package_name> -save-dev` для добавления нового пакета. Команда `npm install` не устанавливает эти пакеты.

```
  "dependencies": {  
    "express": "^4.17.1",  
    "react": "^16.13.1"  
  }
```

```
  "devDependencies": {  
    "gulp": "^4.0.2"  
  }
```

при удалении пакета с использованием команды `npm uninstall <package_name>` запись о нем также будет автоматически удалена из списка зависимостей.

Пример package.json

```
"name": "nodemailer",
"version": "6.9.1",
"description": "Easy as cake e-mail sending from your Node.js applications",
"main": "lib/nodemailer.js",
"scripts": {
  "test": "grunt --trace-warnings"
},
"repository": {
  "type": "git",
  "url": "https://github.com/nodemailer/nodemailer.git"
},
"keywords": [
  "Nodemailer"
],
"author": "Andris Reinman",
"license": "MIT",
"bugs": {
  "url": "https://github.com/nodemailer/nodemailer/issues"
},
"homepage": "https://nodemailer.com/",
```

```
"devDependencies": {
  "@aws-sdk/client-ses": "3.259.0",
  "aws-sdk": "2.1303.0",
  "bunyan": "1.8.15",
  "chai": "4.3.7",
  "eslint-config-nodemailer": "1.2.0",
  "eslint-config-prettier": "8.6.0",
  "grunt": "1.5.3",
  "grunt-cli": "1.4.3",
  "grunt-eslint": "24.0.1",
  "grunt-mocha-test": "0.13.3",
  "libbase64": "1.2.1",
  "libmime": "5.2.0",
  "libqp": "2.0.1",
  "mocha": "10.2.0",
  "nodemailer-ntlm-auth": "1.0.3",
  "proxy": "1.0.2",
  "proxy-test-server": "1.0.0",
  "sinon": "15.0.1",
  "smtp-server": "3.11.0"
},
"engines": {
  "node": ">=6.0.0"
}
```

Семантическое версионирование

- **спецификация**, которая описывает правила присвоения версии релизам программного обеспечения. Сайт - <https://semver.org>.
- призвано **решить проблему “ада зависимостей”** (**dependency hell**). Данное состояние может наступить, когда проект имеет слишком много зависимостей (невозможность обновить пакет без необходимости выпуска новой версии каждой зависимой библиотеки). Решение – нумерация версий.

MAJOR.MINOR.PATCH

* целые, неотрицательные числа

- **MAJOR** – новые возможности без сохранения обратной совместимости. Начальная разработка – значение 0. Когда увеличивается, значения MINOR и PATCH должны быть обнулены.
- **MINOR** – новые возможности или измененные старые с сохранением совместимости. Должно быть изменено, если какой-либо функционал помечен как устаревший. При увеличении значения, значение PATCH должно быть обнулено.
- **PATCH** – исправление ошибок, рефакторинг.

- **Нельзя изменять уже опубликованные версии.** Любые изменения в проекте должны сопровождаться новой версией.
- Предрелиз может быть обозначен **добавлением дефиса** и серией разделённых точкой идентификаторов, следующих сразу за PATCH. Должна содержать лишь буквенно-цифровые символы и дефис. Не должна начинаться с нуля. Указывает на то, что эта версия не стабильна и может не удовлетворять требованиям совместимости.

Пример: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-beta

- Сборочные метаданные могут быть обозначены добавлением знака плюс и ряда разделённых точкой идентификаторов, следующих сразу за PATCH или предрелизной версией. Должны содержать лишь буквенно-цифровые символы и дефис.

Пример: 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85.

- Приоритет версии определяется по первому отличию при сравнении идентификаторов **слева направо**. Сборочные метаданные при определении приоритета не учитываются.

Пример: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1

Спецификаторы версий

- `~`: подходят **новые PATCH** версии

Пример: `~0.13.0`: 0.13.1 — подходит, 0.14.0 — нет

- `^`: подходят **новые PATCH и MINOR** версии

Пример: `^0.13.0`: 0.13.1, 0.14.0 и т.д. — подходит

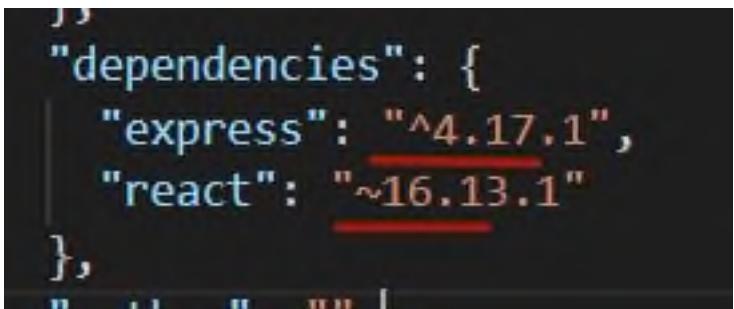
- `*`: подходят **новые MAJOR** версии

- `>`: подходят любые версии пакета, которые **больше заданной**

- `>=`: подходят любые версии пакета, которые **равны или больше заданной**

- `<=`: подходят пакеты, версии которых **равны заданной или меньше её**

- `<`: подходят пакеты, версии которых **меньше заданной**



```
  "dependencies": {  
    "express": "^4.17.1",  
    "react": "~16.13.1"  
  },
```

Спецификаторы версий

- `=`: подходит **только заданная версия** пакета
- `-`: подходит все из **диапазона версий**

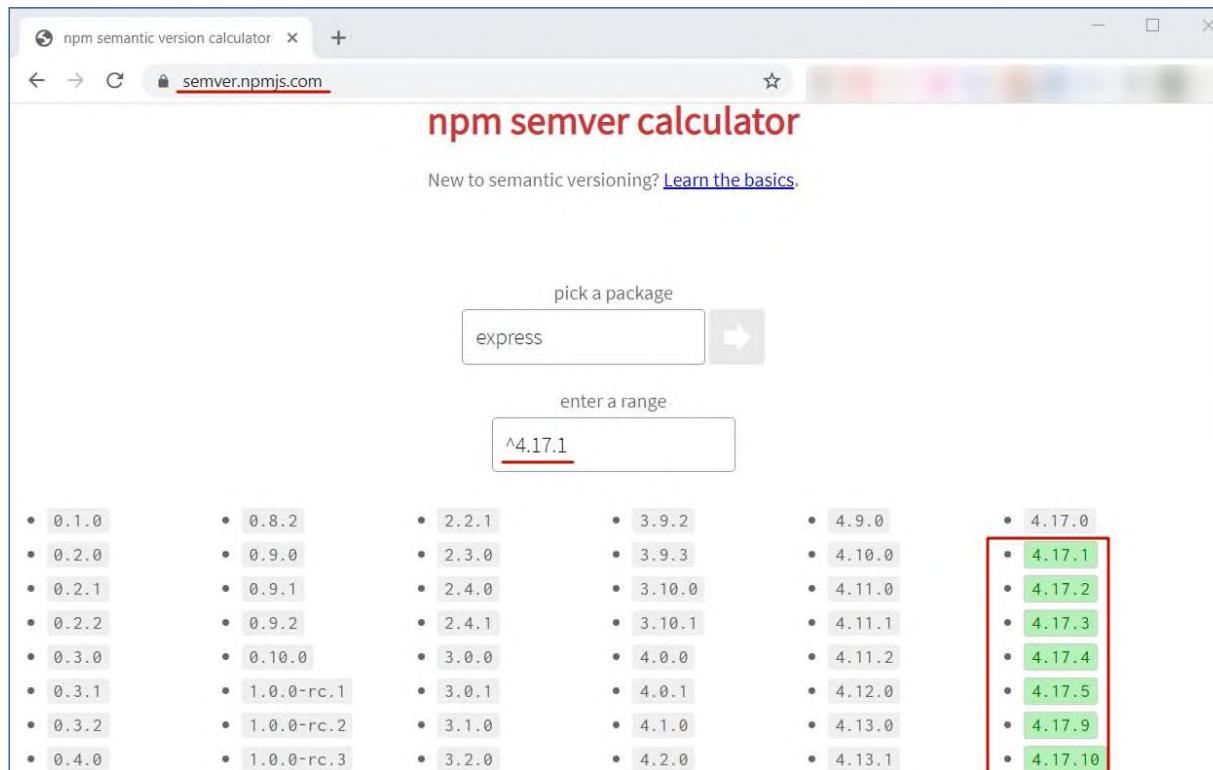
Пример: `2.1.0 - 2.6.2`

- `||`: позволяет **комбинировать** наборы условий, касающихся пакетов

Пример: `< 2.1 || > 2.6`

- **отсутствие дополнительных символов**: подходит **только заданная версия** пакета-зависимости и никакая другая
- `latest`: указывает на то, что вам требуется **самая свежая версия** пакета.

Калькулятор версий



The screenshot shows a web browser window for the npm semantic version calculator at semver.npmjs.com. The page title is "npm semver calculator". A search bar at the top has "express" in it. Below it, a "pick a package" section has "express" in a dropdown. An "enter a range" section has the input field containing "^4.17.1". Below these, a grid of semantic version numbers is displayed in a 5x6 grid. The version "4.17.1" is highlighted with a red border and a green background. Other versions in the same row are also highlighted with green backgrounds.

• 0.1.0	• 0.8.2	• 2.2.1	• 3.9.2	• 4.9.0	• 4.17.0
• 0.2.0	• 0.9.0	• 2.3.0	• 3.9.3	• 4.10.0	• 4.17.1
• 0.2.1	• 0.9.1	• 2.4.0	• 3.10.0	• 4.11.0	• 4.17.2
• 0.2.2	• 0.9.2	• 2.4.1	• 3.10.1	• 4.11.1	• 4.17.3
• 0.3.0	• 0.10.0	• 3.0.0	• 4.0.0	• 4.11.2	• 4.17.4
• 0.3.1	• 1.0.0-rc.1	• 3.0.1	• 4.0.1	• 4.12.0	• 4.17.5
• 0.3.2	• 1.0.0-rc.2	• 3.1.0	• 4.1.0	• 4.13.0	• 4.17.9
• 0.4.0	• 1.0.0-rc.3	• 3.2.0	• 4.2.0	• 4.13.1	• 4.17.10

<https://semver.npmjs.com/>

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

HTTP-СЕРВЕР (МОДУЛЬ HTTP)

Web-приложение =

клиент-серверное приложение, у которого клиент и сервер взаимодействуют по некоторому протоколу прикладного уровня.

Когда говорят о разработке web-приложения, говорят о разработке frontend и backend.

Сами термины возникли в программной инженерии по причине появления **принципа разделения ответственности между внутренней реализацией и внешним представлением**.

Frontend

=

клиентская часть приложения,
пользовательский интерфейс.

To, что видит клиент.

Backend

=

серверная часть веб-приложения, программно-аппаратная часть сервиса.

То, что скрыто от наших глаз, т.е. происходит вне компьютера и браузера.

Как раз-таки там заключена вся логика приложения, там происходит обработка клиентских запросов и формирование ответов на них.

Модуль `http` =

модуль для создания низкоуровневого
HTTP-сервера и HTTP-клиента

Простейший сервер

```
const http = require('http');

const PORT = 3000;
const HOSTNAME = 'localhost';

let requestCount = 0;
let responseText = '';

const server = http.createServer((req, res) => {
  responseText += `URL: ${req.url}<br />Номер запроса: ${++requestCount}<br /><br />`;
  res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
  res.write('<h2>HTTP-сервер</h2>');
  res.end(responseText);
});

server.on('error', (error) => {
  console.error(error);
});

server.listen(PORT, HOSTNAME, () => {
  console.log(`Server running at http://${HOSTNAME}:${PORT}/`);
});
```

Метод `createServer()` возвращает объект класса `http.Server`.

Коллбэки добавляются в качестве слушателей на события `request`, `error` и `listening`

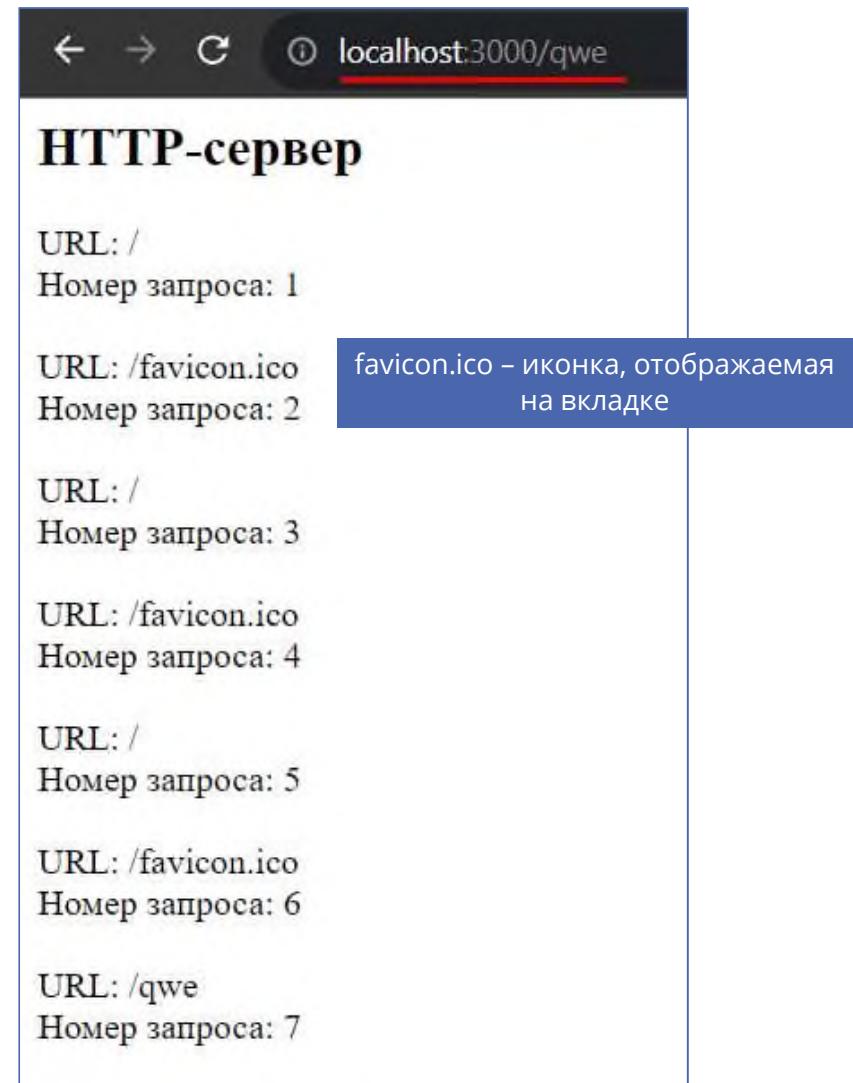
`res.end()` должен обязательно вызываться в конце обработки каждого запроса.

Демонстрация работы

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-00.js
Server running at http://localhost:3000/
```

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-00.js
Error: listen EADDRINUSE: address already in use ::1:3000
  at Server.setupListenHandle [as _listen2] (node:net:1751:16)
  at listenInCluster (node:net:1799:12)
  at GetAddrInfoReqWrap.doListen [as callback] (node:net:1948:7)
  at GetAddrInfoReqWrap.onlookup [as oncomplete] (node:dns:110:8) {
  code: 'EADDRINUSE',
  errno: -4091,
  syscall: 'listen',
  address: '::1',
  port: 3000
}
```

Ошибка связана с тем, что **порт занят**. Т.е. возможно сервер уже запущен



localhost:3000/qwe

HTTP-сервер

URL: /
Номер запроса: 1

URL: /favicon.ico
Номер запроса: 2

favicon.ico – иконка, отображаемая на вкладке

URL: /
Номер запроса: 3

URL: /favicon.ico
Номер запроса: 4

URL: /
Номер запроса: 5

URL: /favicon.ico
Номер запроса: 6

URL: /qwe
Номер запроса: 7

События класса `http.Server`

- `close` генерируется, когда сервер закрывается.
- `connection` генерируется, когда установлено новое TCP-соединение. В обработчик будет передан экземпляр класса `<net.Socket>`, подкласса `<stream.Duplex>`.
- `request` генерируется каждый раз при поступлении запроса. На одно соединение может быть несколько запросов (в случае соединений HTTP Keep-Alive). В обработчик передаются экземпляры запроса `<http.IncomingMessage>` и ответа `<http.ServerResponse>`.
- `upgrade` генерируется каждый раз, когда клиент запрашивает обновление HTTP (например, переключение на протокол WS). В обработчик будет передан экземпляр класса `<net.Socket>`, подкласса `<stream.Duplex>`.

[Подробнее тут](#)

Пример (событие request)

```
const http = require('http');

const PORT = 3000;
const HOSTNAME = 'localhost';

let requestCount = 0;
let responseText = '';

const request_handler = (req, res) => {
  console.log(`request url: ${req.url}, # ${requestCount}`);
  res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
  res.write(`<h2>Http-сервер</h2>`);
  responseText += `url = ${req.url}, request/response # ${requestCount}<br />`;
  res.end(responseText);
};

const server = http.createServer(request_handler);

server.on('request', (req, res) => { // после request_handler
  console.log('request: # ', requestCount);
});

server.listen(PORT, HOSTNAME, () => {
  console.log(`Server running at http://${HOSTNAME}:${PORT}`);
})
.on('error', (error) => { console.error(error); })
```

Коллбэк, передающийся в **метод createServer**,
автоматически навешивается на **событие request**.

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-03.js
Server running at http://localhost:3000/
request url: /, # 1
request: # 1
request url: /favicon.ico, # 2
request: # 2
request url: /, # 3
request: # 3
request url: /favicon.ico, # 4
request: # 4
request url: /, # 5
request: # 5
request url: /favicon.ico, # 6
request: # 6
```

Обработчики на событие
вызываются **в порядке их
регистрации**.

Пример (событие connection)

```
const http = require('http');

const PORT = 3000;
const HOSTNAME = 'localhost';

let requestCount = 0;
let responseText = '';
let connectionCount = 0;

const request_handler = (req, res) => {
  console.log(`request url: ${req.url}, # ${requestCount}, ++requestCount`);
  res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
  res.write(`<h2>Http-сервер</h2>`);
  responseText += `url = ${req.url}, request/response # ${connectionCount} - ${requestCount} <br />`;
  res.end(responseText);
};

const server = http.createServer();

server.keepAliveTimeout = 10000; // по умолчанию 5000;
server.on('connection', (socket) => { // устанавливается новое соединение
  console.log(`connection: server.keepAliveTimeout = ${server.keepAliveTimeout}, ${connectionCount}, ++connectionCount`);
  responseText += `<h2>connection: # ${connectionCount}</h2>`;
});

server.on('request', request_handler);

server.listen(PORT, HOSTNAME, () => {
  console.log(`Server running at http://${HOSTNAME}:${PORT}/`);
}).on('error', (error) => { console.error(error); })
```

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-04.js
Server running at http://localhost:3000/
connection: server.keepAliveTimeout = 10000 1
request url: /, # 1
request url: /favicon.ico, # 2
connection: server.keepAliveTimeout = 10000 2
request url: /, # 3
request url: /favicon.ico, # 4
request url: /, # 5
request url: /favicon.ico, # 6
request url: /, # 7
request url: /favicon.ico, # 8
request url: /, # 9
request url: /favicon.ico, # 10
request url: /, # 11
request url: /favicon.ico, # 12
connection: server.keepAliveTimeout = 10000 3
request url: /, # 13
request url: /favicon.ico, # 14
request url: /, # 15
```

server.keepAliveTimeout устанавливает таймаут для постоянных соединений. Указывает, сколько времени соединение будет оставаться открытым, если на нем не происходит активности.

Http-сервер

connection: # 1

url = /, request/response # 1 - 1
url = /favicon.ico, request/response # 1 - 2

connection: # 2

url = /, request/response # 2 - 3
url = /favicon.ico, request/response # 2 - 4
url = /, request/response # 2 - 5
url = /favicon.ico, request/response # 2 - 6
url = /, request/response # 2 - 7
url = /favicon.ico, request/response # 2 - 8
url = /, request/response # 2 - 9
url = /favicon.ico, request/response # 2 - 10
url = /, request/response # 2 - 11
url = /favicon.ico, request/response # 2 - 12

connection: # 3

url = /, request/response # 3 - 13
url = /favicon.ico, request/response # 3 - 14
url = /, request/response # 3 - 15

Пример (события timeout, close)

```
const http = require('http');
const PORT = 3000;
const HOSTNAME = 'localhost';

let requestCount = 0;
let responseText = '';
let connectionCount = 0;

let request_handler = (req, res) => {
  if (req.url == '/close') { server.close(() => console.log('server.close')) } ←

  console.log(`URL: ${req.url} Номер запроса: ${++requestCount}, process.uptime());
  res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
  res.write(<h2>HTTP-сервер</h2>');
  responseText += `URL: ${req.url}<br />Номер соединения-запроса: ${connectionCount} - ${requestCount}<br /><br />`;
  res.end(responseText);
}

let server = http.createServer();

// server.keepAliveTimeout = 10000; // 5000 по умолчанию
// server.timeout = 30000; // 0 по умолчанию; время бездействия, разрешённого после получения запроса

server.on('request', request_handler);

server.on('timeout', () => { console.log('server timeout: ', server.timeout, process.uptime()) })
```

На событие close здесь навешено
два обработчика

```
server.on('close', () => { console.log('server close ', process.uptime()) })

server.on('connection', (socket) => {
  socket.setTimeout(6000);
  console.log(`connection: keepAliveTimeout = ${server.keepAliveTimeout} ${++connectionCount}, process.uptime());
  socket.on('close', function () {
    console.log('socket close');
  });

  socket.on('timeout', function () { // сокет бездействует, только уведомление
    console.log('socket destroy, timeout', socket.timeout, process.uptime());
    // Надо закрывать самостоятельно
    socket.destroy(); // или socket.end(), но тогда с обеих сторон
  });
}

server.listen(PORT, HOSTNAME, () => {
  console.log(`Server running at http://${HOSTNAME}:${PORT}/`);
}) .on('error', (error) => { console.error(error); })
```

Демонстрация

```
HTTP-сервер
URL: /
Номер соединения-запроса: 1 - 1

URL: /favicon.ico
Номер соединения-запроса: 1 - 2

URL: /close
Номер соединения-запроса: 3 - 3
```

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-01.js
Server running at http://localhost:3000/
connection: keepAliveTimeout = 10000 1 6.9712776
URL: / Номер запроса: 1 6.9773625
URL: /favicon.ico Номер запроса: 2 8.0786277
server timeout: 0 18.0870632
socket destroy, timeout 10000 18.0891047
socket close
connection: keepAliveTimeout = 10000 2 33.4606036
URL: /close Номер запроса: 3 33.4643403
URL: /favicon.ico Номер запроса: 4 33.8093825
server timeout: 0 43.8141965
socket destroy, timeout 10000 43.8158101
server close 43.8190913
server.close
socket close
PS D:\NodeJS\samples\cwp_06_07> []
```

keepAliveTimeout = 10сек
server.timeout 0сек по умолчанию
socket.timeout 0сек по умолчанию,
но из-за keepAliveTimeout равен
10сек

Свойства класса http.Server

- **listening** указывает, прослушивает ли сервер соединения.
- **requestTimeout** устанавливает значение таймаута в мс для получения всего запроса от клиента. Если тайм-аут истекает, сервер отвечает статусом 408, а затем закрывает соединение.
- **maxRequestsPerSocket** максимальное количество запросов, которые сокет может обработать перед закрытием соединения. Значение 0 отключит ограничение. Когда лимит будет достигнут, он установит значение заголовка Connection: close, но фактически не закроет соединение. Последующие запросы, отправленные после достижения лимита, получат в качестве ответа 503 Service Unavailable. По умолчанию 300с (5 минут)
- **timeout** указывает количество мс бездействия, прежде чем предполагается, что сокет истечет. Значение 0 отключит тайм-аут для входящих соединений. Изменение этого значения влияет только на новые подключения к серверу, а не на существующие соединения.
- **keepAliveTimeout** указывает количество мс бездействия, которое необходимо серверу для ожидания дополнительных входящих данных после завершения записи последнего ответа, прежде чем сокет будет уничтожен. Если сервер получит новые данные до того, как истечет тайм-аут поддержания активности, он сбросит обычный тайм-аут бездействия, т. е. server.timeout. Значение 0 отключит тайм-аут поддержания активности для входящих соединений. Изменение этого значения влияет только на новые подключения к серверу, а не на существующие соединения.

Методы класса http.Server

- `close([callback])` не позволяет серверу принимать новые соединения и сохраняет существующие соединения. Эта функция является асинхронной: сервер окончательно закрывается, когда все соединения завершены, и сервер выдает событие `close`. Необязательный `callback` будет вызван после возникновения события `close`. Если сервер не был открыт, когда происходило закрытие, то в `callback` будет передан объект ошибки
- `closeAllConnections()` закрывает все соединения, подключенные к этому серверу.
- `closeIdleConnections()` закрывает все соединения, подключенные к этому серверу, которые не отправляют запрос и не ждут ответа.
- `listen(...)` запускает HTTP-сервер, прослушивающий соединения.

```
const http = require('http');

const PORT = 3000;
const HOSTNAME = 'localhost';

const request_handler = (req, res) => {
  console.log('request.url = ', req.url); // url
  console.log('request.method = ', req.method); // HTTP-метод
  console.log('request.httpVersion = ', req.httpVersion); // HTTP-версия
  // console.log('request.headers = ', req.headers); // HTTP-заголовки
  for (key in req.headers) console.log(`request header ${key}: ${req.headers[key]}`);
  // данные из тела только через события, например, req.on('data') и req.on('end')
```

request позволяет получить информацию о запросе и представляет объект **класса http.IncomingMessage**

```
res.statusCode = 400; // один из способов задать кода статуса
res.statusMessage = 'Call +375 327 43 76'; // сообщение для статуса
res.setHeader('X-author', 'IS&T, BSTU, sm@belstu.by'); // добавить один заголовок
res.writeHead(400, { // кода статуса + заголовки
  'Content-Type': 'application/json; charset=utf-8',
  'Cache-Control': 'no-cache'
});

res.write('1-я порция'); // писать в тело ответа
res.write('2-я порция'); // писать в тело ответа
res.end('последняя порция'); // писать последнюю порцию данных в тело ответа
```

response управляет отправкой ответа и представляет объект **класса http.ServerResponse**

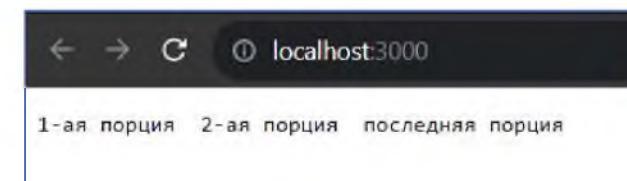
```
const server = http.createServer();

server.listen(PORT, HOSTNAME, () => {
  console.log(`Server running at http://${HOSTNAME}:${PORT}/`);
})
.on('error', (error) => { console.error(error); })
.on('request', request_handler);
```

Запрос и ответ (пример)

Запрос и ответ

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-05.js
Server running at http://localhost:3000/
request.url = /
request.method = GET
request.httpVersion = 1.1
request header host: localhost:3000
request header connection: keep-alive
request header cache-control: max-age=0
request header sec-ch-ua: "Chromium";v="118", "Google Chrome";v="118", "Not=A?Brand";v="99"
request header sec-ch-ua-mobile: ?0
request header sec-ch-ua-platform: "Windows"
request header dnt: 1
request header upgrade-insecure-requests: 1
request header user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
request header accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*
request header sec-fetch-site: none
request header sec-fetch-mode: navigate
request header sec-fetch-user: ?1
request header sec-fetch-dest: document
request header accept-encoding: gzip, deflate, br
request header accept-language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7
request.url = /favicon.ico
request.method = GET
request.httpVersion = 1.1
request header host: localhost:3000
request header connection: keep-alive
request header sec-ch-ua: "Chromium";v="118", "Google Chrome";v="118", "Not=A?Brand";v="99"
request header dnt: 1
request header sec-ch-ua-mobile: ?0
request header user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
request header sec-ch-ua-platform: "Windows"
request header accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
request header sec-fetch-site: same-origin
request header sec-fetch-mode: no-cors
request header sec-fetch-dest: image
request header referer: http://localhost:3000/
request header accept-encoding: gzip, deflate, br
request header accept-language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7
□
```



Класс http.IncomingMessage (запрос)

наследует stream.Readable

События

- **close** генерируется, когда запрос завершен.

Свойства

- **complete** имеет значение `true`, если полное HTTP-сообщение было получено и успешно проанализировано.
- **headers** содержит объект заголовков запроса/ответа. Пары ключ-значение имен и значений заголовков. Имена заголовков пишутся строчными буквами. Дубликаты объединяются
- **httpVersion** содержит версию HTTP, используемую клиентом.
- **method*** содержит метод запроса в виде строки
- **url*** содержит строку URL, на которую был отправлен запрос.

* Действительно только для запроса, полученного http.Server'ом

[Подробнее тут](#)

Класс http.IncomingMessage (запрос)

Свойства

- **rawHeaders** содержит необработанные заголовки запроса/ответа. Ключи и значения находятся в одном списке. Четные элементы являются ключевыми значениями, а нечетные — связанными значениями. Имена заголовков не пишутся строчными буквами, а дубликаты не объединяются.
- **socket** содержит объект класса `net.Socket`, связанный с соединением.
- **statusCode**** содержит трехзначный код состояния ответа HTTP
- **statusMessage**** содержит сообщение о состоянии HTTP-ответа

** Действительно только для ответа, полученного из `http.ClientRequest`.

Методы

- **destroy([error])** вызывает `destroy()` для сокета, получившего `IncomingMessage`.

Класс http.ServerResponse (ответ)

наследует http.OutgoingMessage

События

- **finish** генерируется при успешном завершении передачи.

Свойства

- **headersSent** содержит true, если заголовки были отправлены, иначе false.
- **req** содержит ссылку на объект HTTP-запроса.
- **sendDate** должен содержать true, заголовок Date будет автоматически сгенерирован и отправлен в ответе.
- **socket** содержит ссылку на базовый сокет.
- **writableEnded** имеет значение true, если был вызван outgoingMessage.end().
- **writableFinished** имеет значение true, если все данные были отправлены.
- **writableLength** содержит количество буферизованных байтов.
- **statusCode** при использовании неявных заголовков (без вызова writeHead()) управляет кодом состояния.
- **statusMessage** при использовании неявных заголовков (без вызова writeHead()) управляет сообщением о состоянии.

[Подробнее тут](#)

Класс http.ServerResponse (ответ)

Методы

- `appendHeader(name, value)` добавляет одно значение заголовка для объекта заголовка.
- `destroy([error])` уничтожает сообщение (сокет тоже).
- `end(chunk[, encoding][, callback])` завершает исходящее сообщение. Если указан chunk, это эквивалентно вызову outgoingMessage.write(chunk,coding), за которым следует outgoingMessage.end(callback). Если есть callback, он будет вызван после завершения сообщения (на событие finish).
- `getHeader(name)` получает значение HTTP-заголовка с заданным именем. Если этот заголовок не установлен, возвращаемое значение будет `undefined`.
- `getHeaderNames()` возвращает массив, содержащий уникальные имена текущих исходящих заголовков. Все имена в нижнем регистре.
- `getHeaders()` возвращает неполную копию текущих исходящих заголовков. Ключами объекта являются имена заголовков, а значениями — соответствующие значения заголовка. Все имена заголовков написаны строчными буквами.

Класс http.ServerResponse (ответ)

Методы

- **hasHeader(name)** возвращает true, если заголовок найден в исходящих заголовках. Имя заголовка не чувствительно к регистру.
- **removeHeader(name)** удаляет заголовок, поставленный в очередь для неявной отправки.
- **setHeader(name, value)** устанавливает одно значение заголовка. Если заголовок уже существует, его значение будет заменено. Можно использовать массив строк для отправки нескольких заголовков с одним и тем же именем.
- **setHeaders(headers)** возвращает объект ответа. Устанавливает несколько значений заголовков для неявных заголовков. Если заголовок уже существует в заголовках, подлежащих отправке, его значение будет заменено.
- **write(chunk[, encoding][, callback])** отправляет часть тела. Этот метод можно вызывать несколько раз.

Класс http. http.ServerResponse (ответ)

Методы

- `writeHead(statusCode[, statusMessage][, headers])` отправляет заголовок (-ки) ответа и устанавливает статус код. В качестве второго аргумента можно указать `statusMessage`. Заголовки могут быть переданы в виде объекта или массива). Возвращает ссылку на `ServerResponse`, чтобы можно было объединять вызовы.

- ! Если этот метод вызывается и функция `response.setHeader()` не была вызвана, он будет напрямую записывать предоставленные значения заголовка в сетевой канал без внутреннего кэширования, а метод `response.getHeader()` в заголовке не даст ожидаемого результата.
- ! Когда заголовки были установлены с помощью метода `response.setHeader()`, они будут объединены с любыми заголовками, переданными в ответ `.writeHead()`, при этом заголовкам, переданным в ответ `.writeHead()`, будет присвоен приоритет.

Сокет

=

программный интерфейс,
представляющий собой
совокупность IP-адреса и номера
порта.

В Node.js представляет собой
экземпляр класса `net.Socket`.

Сокеты

```
server.keepAliveTimeout = 10000;           // время сохранения соединения (connection), умолчание = 5000;
server.on('connection', (socket)=>{        // устанавливается новое соединение
    console.log(`connection: server.keepAliveTimeout = ${server.keepAliveTimeout} `, ++c);
    s += `<h2>connection: # ${c}</h2>`;
    console.log('socket.localAddress = ', socket.localAddress);
    console.log('socket.llocalPort = ', socket.localPort);
    console.log('socket.remoteAddress = ', socket.remoteAddress);
    console.log('socket.remoteFamily = ', socket.remoteFamily);
    console.log('socket.remotePort = ', socket.remotePort);
    console.log('socket.bytesWritten = ', socket.bytesWritten);
});
```

```
D:\PSCA\Lec06>node 06-10.js
server.listen(3000)
connection: server.keepAliveTimeout = 10000  1
socket.localAddress =  ::1
socket.llocalPort =  3000
socket.remoteAddress =  ::1
socket.remoteFamily =  IPv6
socket.remotePort =  64222
socket.bytesWritten =  0
```

localAddress – строковое представление **локального** IP-адреса (сервера), к которому подключается удаленный клиент.

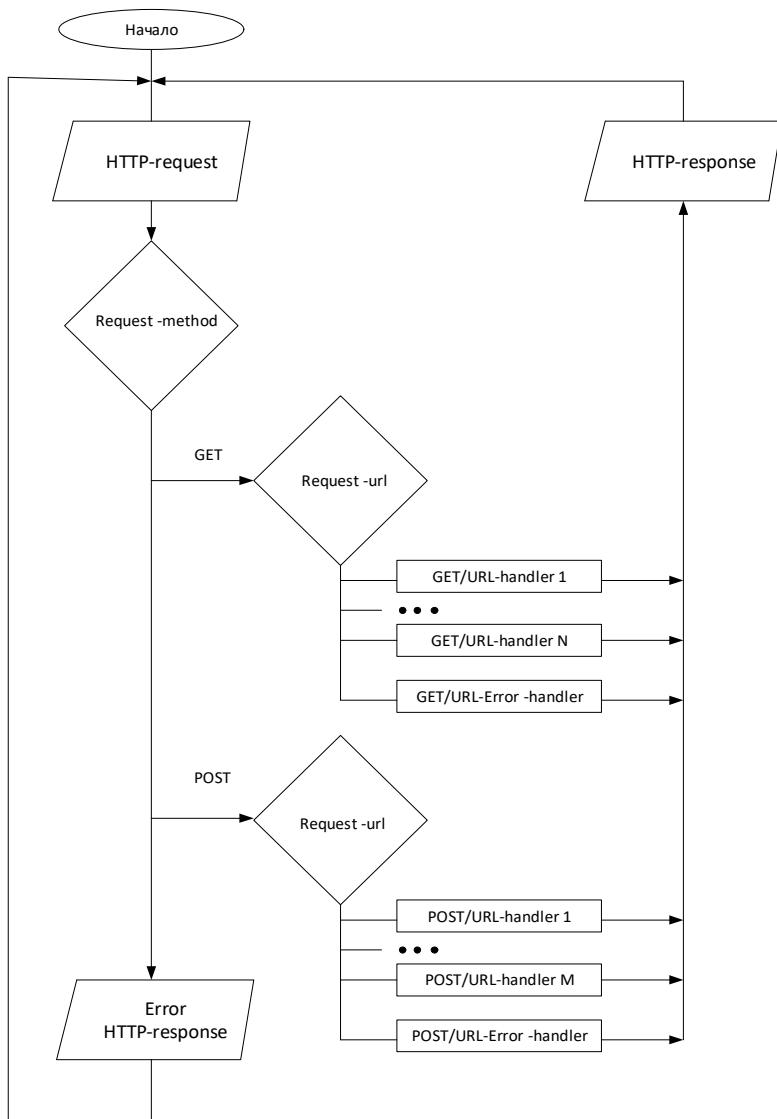
localPort – числовое представление **локального** порта.

remoteAddress – строковое представление **удаленного** IP-адреса (клиента, браузера), с которого поступают запросы.

remoteFamily – строковое представление семейства **удаленных** IP-адресов.

remotePort – числовое представление **удаленного** порта.

bytesWritten – количество отправленных байтов.



Типичный цикл работы сервера

1

```

const http = require('http');

const PORT = 3000;
const HOSTNAME = 'localhost';

const debug_handler = (req, res) => {
  console.log(req.method, req.url);
  res.writeHead(200, { 'Content-Type': 'application/json; charset=utf-8' });
  res.end(` ${req.method}: ${req.url}`);
}

const HTTP404 = (req, res) => {
  console.log(` ${req.method}: ${req.url}, HTTP status 404`);
  res.writeHead(404, { 'Content-Type': 'application/json; charset=utf-8' });
  res.write(`{"error": "${req.method}: ${req.url}, HTTP status 404"}`);
  res.end();
}

const HTTP405 = (req, res) => {
  console.log(` ${req.method}: ${req.url}, HTTP status 405`);
  res.writeHead(405, { 'Content-Type': 'application/json; charset=utf-8' });
  res.write(`{"error": "${req.method}: ${req.url}, HTTP status 405"}`);
  res.end();
}

```

404 – статус код ответа, который означает, что найденный ресурс не найден.

405 – статус код ответа, который означает, что метод запроса не может быть использован

Пример

2

```

const GET_handler = (req, res) => {
  switch (req.url) {
    case '/': debug_handler(req, res); break;
    case '/index.html': debug_handler(req, res); break;
    case '/site.css': debug_handler(req, res); break;
    case '/calc': debug_handler(req, res); break;
    default: HTTP404(req, res);
  }
};

const POST_handler = (req, res) => { debug_handler(req, res) };
const PUT_handler = (req, res) => { debug_handler(req, res) };
const DELETE_handler = (req, res) => { debug_handler(req, res) };

const request_handler = (req, res) => {
  switch (req.method) {
    case 'GET': GET_handler(req, res); break;
    case 'POST': POST_handler(req, res); break;
    case 'PUT': PUT_handler(req, res); break;
    case 'DELETE': DELETE_handler(req, res); break;
    default: HTTP405(req, res);
  }
};

const server = http.createServer();
server.listen(PORT, HOSTNAME, () => {
  console.log(`Server running at http://${HOSTNAME}:${PORT}/`);
})
  .on('error', (error) => { console.error(error); })
  .on('request', request_handler);

```

Демонстрация

The screenshot shows four requests in Postman:

- Request 1:** GET http://localhost:3000/index.html. Status: 200 OK. Response body: 1. GET: /index.html
- Request 2:** GET http://localhost:3000/index. Status: 404 Not Found. Response body: 1. { "error": "GET: /index, HTTP status 404"}
- Request 3:** PATCH http://localhost:3000/qwerty. Status: 405 Method Not Allowed. Response body: 1. { "error": "PATCH: /qwerty, HTTP status 405"}

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-02.js
Server running at http://localhost:3000/
GET /index.html
POST /123
PUT /calc
DELETE /qwe
PATCH: /qwerty, HTTP status 405
PS D:\NodeJS\samples\cwp_06_07> node .\06-02.js
Server running at http://localhost:3000/
GET /index.html
GET: /index, HTTP status 404
PS D:\NodeJS\samples\cwp_06_07> node .\06-02.js
Server running at http://localhost:3000/
GET /index.html
GET: /index, HTTP status 404
POST /123
PUT /calc
DELETE /qwe
PATCH: /qwerty, HTTP status 405
[]
```

Виды параметров запроса

- **Query-параметры** – дополнительная информация, которую можно добавить в URL-адрес. Состоит из двух обязательных элементов: самого параметра и его значения, разделенных знаком равенства (=). Параметры **указываются в конце URL**, отделяясь от основного адреса знаком вопроса (?). Можно указать более одного параметра, для этого каждый параметр со значениями отделяется от следующего знаком амперсанда (&).
- **Path-параметры** - дополнительная информация, которую можно задать в URL-адресе. Они **являются частью маршрута URL**.

/car/make/12/model?color=mintgreen&doors=4

Обработка GET-параметров

Модуль	Функции, классы
url	parse/format, classes: URL, URLSearchParams
qs	parse/stringify (встроено процентное кодирование и декодирование)

Обработка query-параметров

```
let http    = require('http');
let url    = require('url');

let handler = (req, res)=>{
  if (req.method == 'GET'){
    let p = url.parse(req.url,true);
    let result = '';
    let q = url.parse(req.url,true).query;
    if (!(p.pathname == '/favicon.ico')){
      result = `href: ${p.href}<br/>` +
               `path: ${p.path}<br/>` +
               `pathname: ${p.pathname}<br/>` +
               `search: ${p.search}<br/>`;
      for(key in q) { result+= `${key} = ${q[key]}`<br/>;}
    }
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.write('<h1>GET-параметры</h1>');
    res.end(result);
  }
  else{
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end('for other http-methods not so');
  }
}

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')}
  .on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})
  .on('request', handler)
```

GET-параметры

href: /hhh/?k=3&s=kkkk&j=iii&p1=3&p2=t
path: /hhh/?k=3&s=kkkk&j=iii&p1=3&p2=t
pathname: /hhh/
search: ?k=3&s=kkkk&j=iii&p1=3&p2=t
k = 3
s = kkkk
j = iii
p1 = 3
p2 = t

Метод `url.parse()` принимает строку URL-адреса, анализирует ее и возвращает объект URL. Первым параметром передается URL-адрес, вторым параметром – флаг, нужно ли парсить query-параметры.

Обработка path-параметров

```
let http    = require('http');
let url     = require('url');

let handler = (req, res)=>{
  if (req.method == 'GET'){
    let p = url.parse(req.url,true);
    let result ='';
    let q = url.parse(req.url,true).query;
    if (!(p.pathname == '/favicon.ico')){
      result = `pathname: ${p.pathname}<br/>`;
      p.pathname.split('/').forEach(e => {result+= ` ${e}<br/>`});
    }
    console.log(p.pathname.split('/'));
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.write('<h1>URL-параметры</h1>');
    res.end(result);
  }
  else{
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end('for other http-methods not so');
  }
}

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')}
  .on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})
  .on('request', handler)
```

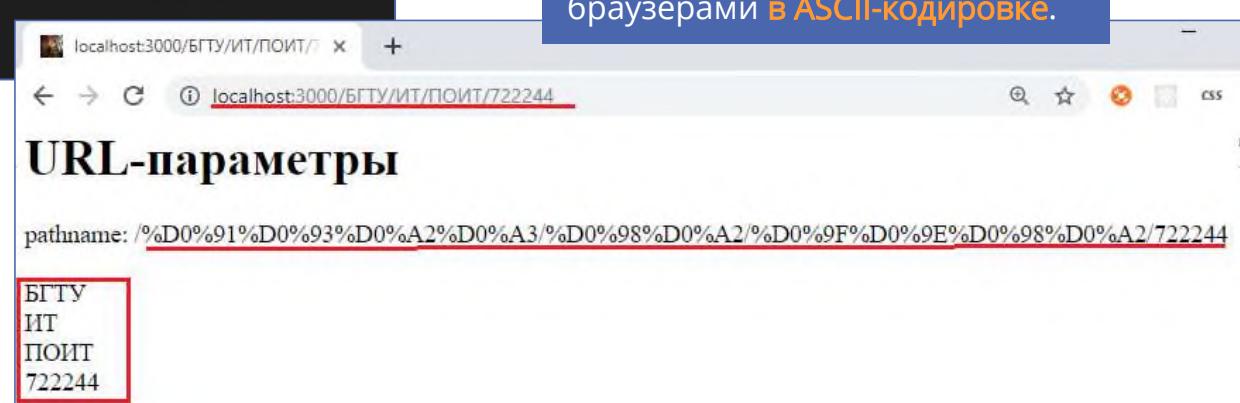


Для того, чтобы их извлечь, необходимо:

1. Распарсить url.
2. Достать свойство pathname.
3. Разбить его на элементы с помощью разделителя /, используя метод split().

Декодирование символов

```
if (req.method == 'GET'){
  let p = url.parse(req.url, true);
  let result = '';
  let q = url.parse(req.url, true).query;
  if (!(p.pathname == '/favicon.ico')){
    result = `pathname: ${p.pathname}<br/>`;
    decodeURI(p.pathname).split('/').forEach(e => {result += ` ${e}<br/>`});
  }
  console.log(p.pathname.split('/'));
  res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
  res.write('<h1>URL-параметры</h1>');
  res.end(result);
}
```



URL-адреса отправляются
браузерами в ASCII-кодировке.

Чтение содержимого тела запроса

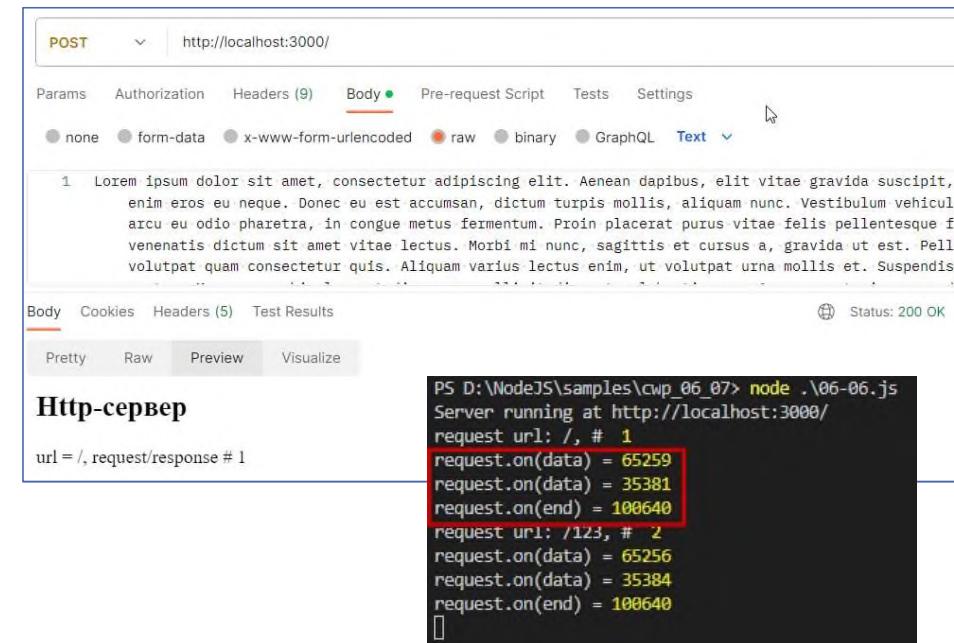
```
const request_handler = (req, res) => {
  console.log(`request url: ${req.url}, # ${requestCount}`);
  res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });

  let buf = '';
  req.on('data', (data) => { // получить фрагментами
    console.log(`request.on(data) = ${data.length}`);
    buf += data;
  });

  req.on('end', () => { // все данные пришли
    console.log(`request.on(end) = ${buf.length}`);
  })

  res.write(`<h2>Http-сервер</h2>`); // отправить порцию

  responseText += `url = ${req.url}, request/response # ${requestCount} <br />`;
  res.end(responseText);
}
```



POST http://localhost:3000/

Params Authorization Headers (9) Body Body (raw) Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL Text

1 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean dapibus, elit vitae gravida suscipit, enim eros eu neque. Donec eu est accumsan, dictum turpis mollis, aliquam nunc. Vestibulum vehicul arcu eu odio pharetra, in congue metus fermentum. Proin placerat purus vitae felis pellentesque f
venenatis dictum sit amet vitae lectus. Morbi mi nunc, sagittis et cursus a, gravida ut est. Pell
volutpat quam consectetur quis. Aliquam varius lectus enim, ut volutpat urna mollis et. Suspendis

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize

Http-сервер

url = /, request/response # 1

```
PS D:\NodeJS\samples\cwp_06_07> node ./06-06.js
Server running at http://localhost:3000/
request url: /, # 1
request.on(data) = 65259
request.on(data) = 35381
request.on(end) = 100640
request url: /123, # 2
request.on(data) = 65256
request.on(data) = 35384
request.on(end) = 100640
```

Объект запроса, переданный обработчику, представляет собой **поток и наследует stream.Readable**. Этот поток можно прослушивать, как и любой другой поток. Можно получить данные прямо из потока, **прослушивая события 'data' и 'end' потока**.

Также можно использовать прослушивание события 'readable' или метод `pipe()`.

Обработка тела (x-www-form-urlencoded)

```
let http    = require('http');
let fs      = require('fs');
let qs = require('qs');

let handler = (req, res)=>{
  if (req.method == 'GET'){
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end(fs.readFileSync('./07-03.html'));
  }
  else if (req.method == 'POST'){
    let result = '';
    req.on('data', (data)=>{result+=data;})
    req.on('end', ()=>{
      result += '<br/>';
      let o = qs.parse(result)
      for (let key in o) { result += `${key} = ${o[key]}<br />`}
      res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
      res.write('<h1>URL-параметры</h1>');
      res.end(result);
    });
  }
  else{
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end('for other http-methods not so');
  }
}

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')}
  .on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})
  .on('request', handler)
```

Метод `qs.parse()` разбивает строку URL-запроса (query) на коллекцию пар ключ и значение (и еще декодирует).

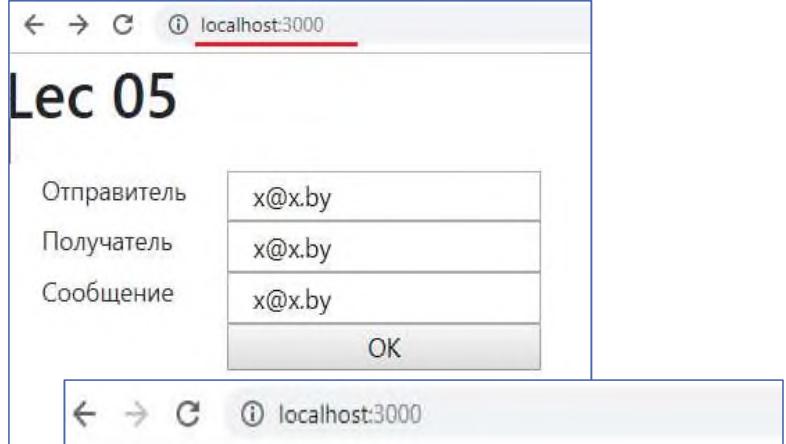
Обработка тела (x-www-form-urlencoded)

```
<body>
<h1>Lec 05</h1>
<div style="margin: 20px; width: 800px; padding: 5px;">
  <form method="POST" action="/">
    <div class="row">
      <label class="col-2">Отправитель</label> <input class="col-3" name="reciver" placeholder="x@x.by" type="text" value="x@x.by"/>
    </div>
    <div class="row">
      <label class="col-2">Получатель</label> <input class="col-3" name="sender" placeholder="x@x.by" type="text" value="x@x.by"/>
    </div>
    <div class="row">
      <label class="col-2">Сообщение</label> <input class="col-3" name="message" placeholder="x@x.by" type="text" value="x@x.by"/>
    </div>
    <div class="row">
      <input type="submit" class="col-3 offset-2" value="OK"/>
    </div>
  </form>
</div>
```

Параметры в POST-запросе по умолчанию **передаются в теле в виде строки парами ключ-значение через &**.

Амперсанд(&) выступает в качестве разделителя между каждой (name, value) парой, позволяя серверу понять, когда и где значение параметра начинается и заканчивается.

username=sidthelsloth&password=slothsecret

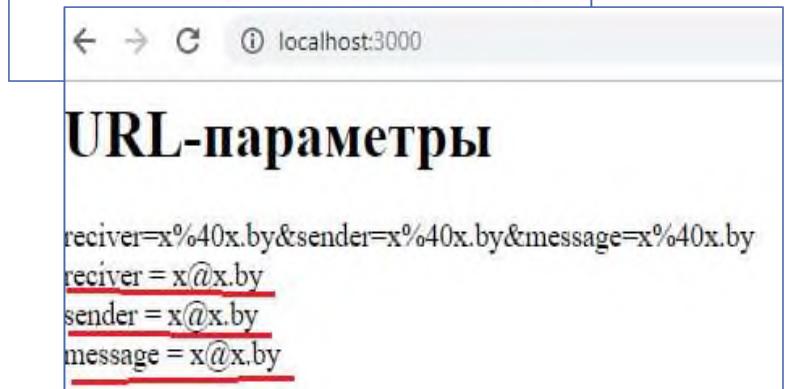


localhost:3000

Lec 05

Отправитель	x@x.by
Получатель	x@x.by
Сообщение	x@x.by

OK



localhost:3000

URL-параметры

receiver=x%40x.by&sender=x%40x.by&message=x%40x.by

receiver = x@x.by
sender = x@x.by
message = x@x.by

MIME
(Multipurpose Internet Mail
Extensions) =

стандарт, указывающий формат документа, файла или набора байтов. Используется для идентификации типа данных.

Описаны в RFC 2045, RFC 2046, RFC 4288, RFC 4289 и RFC 4855, зарегистрированы IANA.

Используется в заголовках Content-Type, Accept.

JSON (JavaScript Object Notation)

- текстовый формат хранения и передачи данных;
- автор: [Дуглас Крокфорд](#);
- похож на [буквенный синтаксис объекта JavaScript](#);
- JSON основан на двух структурах данных: [коллекция пар имя/значение](#), упорядоченный [список значений](#) (реализовано как массив, список или др.);
- Значение может быть строкой в двойных кавычках, числом, true, false, null, объектом или массивом.
- MIME: [application/json](#) (RFC 4627).

```
{"employees": [  
    { "firstName": "John", "lastName": "Doe" },  
    { "firstName": "Anna", "lastName": "Smith" },  
    { "firstName": "Peter", "lastName": "Jones" }  
]
```



Подробнее: <https://www.json.org/json-ru.html>

```

let http    = require('http');
let m0706   = require('./m07-06');

let handler = (req, res)=>{
  if (req.method == 'POST' && m0706.isJsonContentType (req.headers)){
    let result = '';
    req.on('data', (data)=>{result += data;})
    req.on('end',  ()=>{
      try {
        let obj = JSON.parse(result);
        console.log(obj);
        if (m0706.isJsonAccept(req.headers))
          m0706.write200(res, 'ok json', JSON.stringify(obj));
        else m0706.write400(res, 'no accept');
      }
      catch (e){m0706.write400(res, 'catch: bad json');}
    })
  } else m0706.write400(res, 'no json-post');
}

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')})
  .on('error', (e)=>{console.log('server.listen(3000): error:');})
  .on('request', handler)

```

Метод **JSON.parse()** разбирает строку JSON и преобразовывает в объект

Метод **JSON.stringify()** используется для преобразования JSON-объектов в строку.

Работа с JSON

```

const isJson = (headers, header,  mime) => {
  let rc = false;
  let  h = headers[header];
  if (h) rc =  h.indexOf(mime) >= 0;
  return rc;
}

exports.write400 = (res, smess)=>{
  console.log(smess);
  res.writeHead(400, {'Content-Type': 'text/html; charset=utf-8'});
  res.statusMessage = smess;
  res.end();
}

exports.write200 = (res, smess, mess)=>{
  console.log(smess, mess);
  res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
  res.statusMessage = smess;
  res.end(mess);
}

exports.isJsonContentType = (hs) => isJson(hs,'content-type', 'application/json');
exports.isJsonAccept     = (hs) => isJson(hs,'accept',      'application/json');

```

Функция `require` часто применяется для конфигурационных JSON-файлов, она **парсит JSON в JS-объект**.

JSON официально **не поддерживает комментарии**,
но есть альтернативные способы их создания.

```
{  
  "_comment": "Так можно сделать комментарий",  
  "x": 1,  
  "y": 1.01,  
  "s": "Строка",  
  "array": [  
    "a",  
    "b",  
    "c",  
    "d"  
  ],  
  "obj": {  
    "surname": "Иванов",  
    "name": "Иван"  
  }  
}
```

```
const json = require('./f06-07.json');  
console.log(json);
```

```
PS D:\NodeJS\samples\cwp_06_07> node .\06-07.js  
{  
  _comment: 'Так можно сделать комментарий',  
  x: 1,  
  y: 1.01,  
  s: 'Строка',  
  array: [ 'a', 'b', 'c', 'd' ],  
  obj: { surname: 'Иванов', name: 'Иван' }  
}
```

XML (eXtensible Markup Language)

- расширяемый язык разметки;
- предназначен для хранения и передачи данных;
- самоопределяемый, т.е. вы должны сами определять нужные теги;
- MIME: application/xml, text/xml (RFC 3023)

```
<?xml version="1.0"?>
<actual>
    <!-- содержимое корневого элемента -->
    <status lastUpd = "21.10.2021" checked = "true">
        Active
    </status>
</actual>
```



Работа с XML

```
let parseString = require('xml2js').parseString; // npm install xml2js
let xmlbuilder = require('xmlbuilder'); // скачивается в одном пакете с xml2js
```

```
let xmltext = '<?xml version="1.0" encoding="utf-8"?>'+ // не обязательно
    '<students faculty="ИТ" sprciality="ИСиТ" > '+
    '<student id="7000222" name="Иванов И.И." bday="2000-12-02" />'+
    '<student id="7000223" name="Петров П.П." bday="2000-11-28" />'+
    '<student id="7000228" name="Казан Н.А." bday="2001-09-11" />'+
    '</students>';
```

```
let obj = null;
```

```
parseString(xmltext, function (err, result) {
    obj = result;
    console.log('----- parseString -----');
    console.log(err);
    console.log('-----');
    console.log('result = ', result);
    console.log('-----');
    result.students.student.map( (e,i)=>{
        console.log(`id = ${e.$.id}, name = ${e.$.name}, name = ${e.$.bday}`);
    })
});
```

Метод `xml2js.parseString()` используется для преобразования XML в объект JS.

```
console.log('----- xmlbuilder -----');
let xml2 = xmlbuilder.create(obj,
    {version: '1.0', encoding: 'UTF-8', standalone: true}
).end({pretty: true, standalone: true});
console.log(xml2);
```

С помощью метода `xmlbuilder.create()` можно создать XML-документ на основе существующего объекта JS-объекта.

```
D:\PSCA\Lec07>node 07-07
----- parseString -----
null
-----
result = { students:
  { '$': { faculty: 'ИТ', sprciality: 'ИСиТ' },
    student: [ [Object], [Object], [Object] ] } }
-----
id = 7000222, name = Иванов И.И., name = 2000-12-02
id = 7000223, name = Петров П.П., name = 2000-11-28
id = 7000228, name = Казан Н.А., name = 2001-09-11
----- xmlbuilder -----
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<students>
  <$>
    <faculty>ИТ</faculty>
    <sprciality>ИСиТ</sprciality>
  </$>
  <student>
    <$>
      <id>7000222</id>
      <name>Иванов И.И.</name>
      <bday>2000-12-02</bday>
    </$>
  </student>
  <student>
    <$>
      <id>7000223</id>
      <name>Петров П.П.</name>
      <bday>2000-11-28</bday>
    </$>
  </student>
  <student>
    <$>
      <id>7000228</id>
      <name>Казан Н.А.</name>
      <bday>2001-09-11</bday>
    </$>
  </student>
</students>
```

```
let xmlbuilder = require('xmlbuilder'); // скачивается в одном пакете с xml2js
```

```
// https://github.com/oozcitak/xmlbuilder-js/wiki
```

```
let xmldoc = xmlbuilder.create('students').att('faculty', 'ИТ').att('speciality', 'ИСиТ');
xmldoc.ele('student').att('id', '7000222').att('name', 'Иванов И.И.').att('bday', '2000-12-02')
.up().ele('student').att('id', '7000223').att('name', 'Петров П.П.').att('bday', '2000-11-29')
.txt('Прошел собеседование в iTechArt')
.up().ele('student').att('id', '7000228').att('name', 'Казан Н.А.').att('bday', '2001-09-11');
```

```
let xmldoc1 = xmlbuilder.create('students').att('faculty', 'ИТ').att('speciality', 'ИСиТ');
xmldoc1.ele('student', {id: '7000222', name: 'Иванов И.И.', bday: '2000-12-02'});
xmldoc1.ele('student', {id: '7000223', name: 'Петров П.П.', bday: '2000-11-29'})
.txt('Прошел собеседование в iTechArt');
xmldoc1.ele('student', {id: '7000228', name: 'Казан Н.А.', bday: '2001-09-11'});
```

```
console.log(xmldoc.toString({pretty:true}));
console.log(xmldoc1.toString({pretty:true}));
```

```
D:\PSCA\Lec07>node 07-08
<students faculty="ИТ" speciality="ИСиТ">
  <student id="7000222" name="Иванов И.И." bday="2000-12-02"/>
  <student id="7000223" name="Петров П.П." bday="2000-11-29">Прошел собеседование в iTechArt</student>
  <student id="7000228" name="Казан Н.А." bday="2001-09-11"/>
</students>

<students faculty="ИТ" speciality="ИСиТ">
  <student id="7000222" name="Иванов И.И." bday="2000-12-02"/>
  <student id="7000223" name="Петров П.П." bday="2000-11-29">Прошел собеседование в iTechArt</student>
  <student id="7000223" name="Казан Н.А." bday="2001-09-11"/>
</students>
```

Работа с XML

Можно создать XML-документ вручную.

- **create()** для создания корневого тега;
- **att()** для добавления атрибутов в тег;
- **ele()** для добавления вложенных элементов (дочерних узлов), по аналогии можно добавлять атрибуты во вложенные теги;
- **up()** или **u()** позволяет вернуться к родительскому узлу после создания дочернего;
- **text(), txt()** или **t()** для создания текстовых узлов.

Работа с XML

```
let http    = require('http');
let parseString = require('xml2js').parseString;
let xmlbuilder = require('xmlbuilder');
let m0709   = require('../m07-09');

let studentscalc = (obj)=>{
  let rc = '<result>parse error</result>';
  try {
    let xmldoc = xmlbuilder.create('result')
      .ele('students').att('faculty', obj.students.$.faculty).att('speciality',obj.students.$.speciality)
      .ele('quantity').att('value', obj.students.student.length);
    rc = xmldoc.toString({pretty:true});
  }catch(e){console.log(e);}
  return rc
}

let handler = (req, res)=>{
  if (req.method == 'POST' && m0709.isXMLContentType (req.headers)){
    if (m0709.isXMLAccept(req.headers)) {
      let xmltxt = '';
      req.on('data', (data)=>{xmltxt += data;})
      req.on('end', ()=>{
        parseString(xmltxt, function (err, result) {
          if (err) m0709.write400(res, 'xml parse error');
          else m0709.write200(res, 'ok xml', studentscalc(result));
        })
      })
    }
    else m0709.write400(res, 'no xml accept');
  } else m0709.write400(res, 'no xml-post');
}

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')})
  .on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})
  .on('request', handler)
```

The screenshot shows a Postman request configuration and its response. The request is a POST to `http://localhost:3000`. The 'Body' tab is selected, showing XML content type with the following XML payload:

```
<xml version="1.0" encoding="utf-8">
<students faculty="IT" speciality="IC&T" >
<student id="7000222" name="Иванов И.И." bday="2000-12-02" />
<student id="7000223" name="Петров П.П." bday="2000-11-28" />
<student id="7000228" name="Казан Н.А." bday="2001-09-11" />
</students>
```

The response status is 200 OK, with a response body containing the generated XML output:

```
<result>
<students faculty="IT" speciality="IC&T" >
| | | <quantity value="3"/>
| | | </students>
| | </result>
```

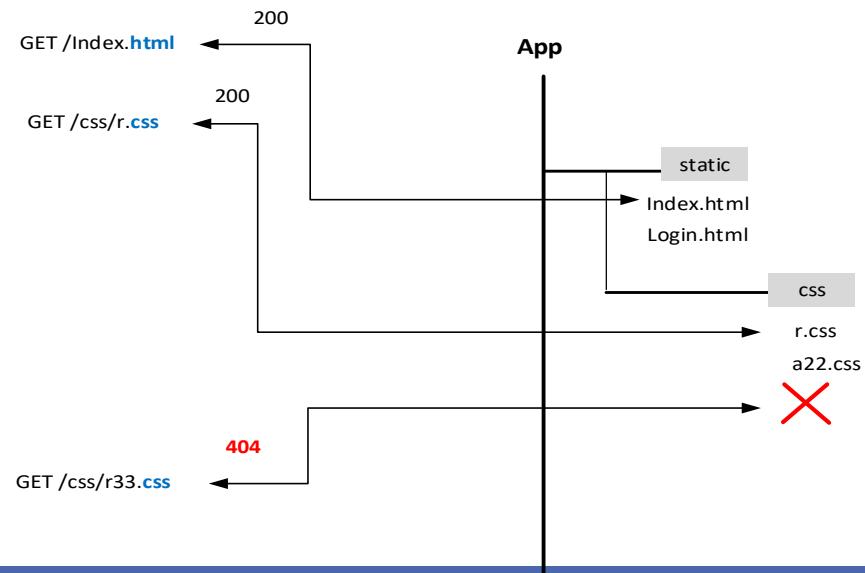
Статические ресурсы

=

ресурсы, расположенные на стороне сервера и предназначенные **для считывания их без изменения** с помощью **GET-запроса** по имени ресурса, включающего имя файла.

Например, их можно получить с помощью тегов `<link>`, `<script>`, ``, `<audio>`, `<video>`.

Обычно хранятся с папке `static` или `public`.



Статические ресурсы

```
let http  =  require('http');
let fs    =  require('fs');

let isStatic = (ext, fn)=>{  let reg =  new RegExp(`^/.+\\.${ext}$`);  return reg.test(fn);}
let pathStatic = (fn)=>{return `./static${fn}`; }
let writeHTTP404 = (res)=>{
    res.statusCode = 404;
    res.statusMessage = 'Resource not found';
    res.end("Resource not found");
}
let pipeFile = (req, res, headers)=>{
    res.writeHead(200, headers);
    fs.createReadStream(pathStatic(req.url)).pipe(res);
}
let sendFile = (req, res, headers)=>{
    fs.access(pathStatic(req.url), fs.constants.R_OK, err => {
        if(err) writeHTTP404(res);
        else pipeFile(req, res, headers);
    });
}
let http_handler = (req, res)=>{
    if      (isStatic('html', req.url)) sendFile(req, res, {'Content-Type': 'text/html; charset=utf-8'});
    else if (isStatic('css',  req.url)) sendFile(req, res, {'Content-Type': 'text/css; charset=utf-8'});
    else if (isStatic('js',   req.url)) sendFile(req, res, {'Content-Type': 'text/css; charset=utf-8'});
    else writeHTTP404(res);
};

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')}
    .on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})
    .on('request', http_handler);
}
```

Все это **можно вынести в отдельный модуль и переиспользовать**. В качестве параметра туда прокидывать путь к директории со статическими файлами.

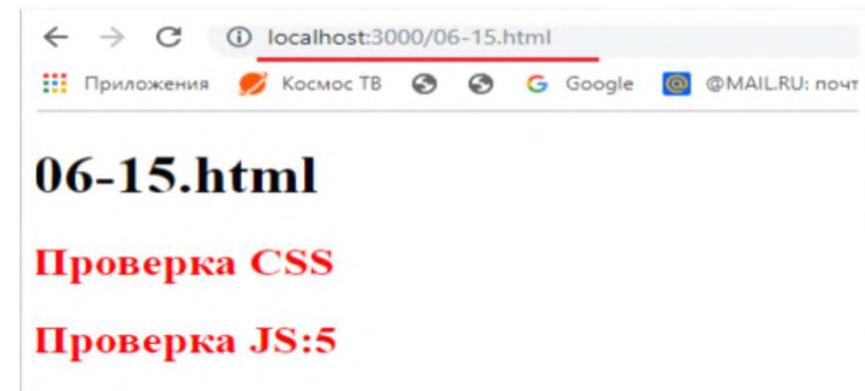
В функции pipeFile() происходит вызов **метода pipe()**, который **связывает поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи**. То есть считывает из файла содержимое и направляет в поток вывода ответа.

Демонстрация

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>06-15</title>
    <link rel="stylesheet" href="css/06-15.css">
    <script src="js/06-15.js"></script>
</head>
<body>
    <h1>06-15.html</h1>
    <h2 class="danger">Проверка CSS</h2>
    <h2 id = "result" class="danger">Проверка JS:</h2>
    <script>
        result.innerHTML += sum(3,2);
    </script>
</body>
</html>
```

Благодаря тегам link и script происходит загрузка необходимых статических файлов с сервера.

PSCA > Lec06 > static >			
Имя	Дата изменения	Тип	Ра
css	23.08.2019 10:41	Папка с файлами	
doc	23.08.2019 13:39	Папка с файлами	
js	23.08.2019 11:13	Папка с файлами	
06-15.html	23.08.2019 11:25	Chrome HTML Do...	



```

let http    = require('http');
let fs      = require('fs');

let handler = (req, res)=>{
  if (req.method == 'GET'){
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end(fs.readFileSync('./07-10.html'));
  }
  else if (req.method == 'POST'){
    let result = '';
    req.on('data', (data)=>{result+=data;})
    req.on('end', ()=>{
      res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
      res.write('<h1>File upload</h1>');
      res.end(result);
    });
  }
  else{
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end('for other http-methods not so');
  }
}

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')})
  .on('error', (e)=>{console.log('server.listen(3000): error', e)})
  .on('request', handler)

```

Формат multipart/form-data

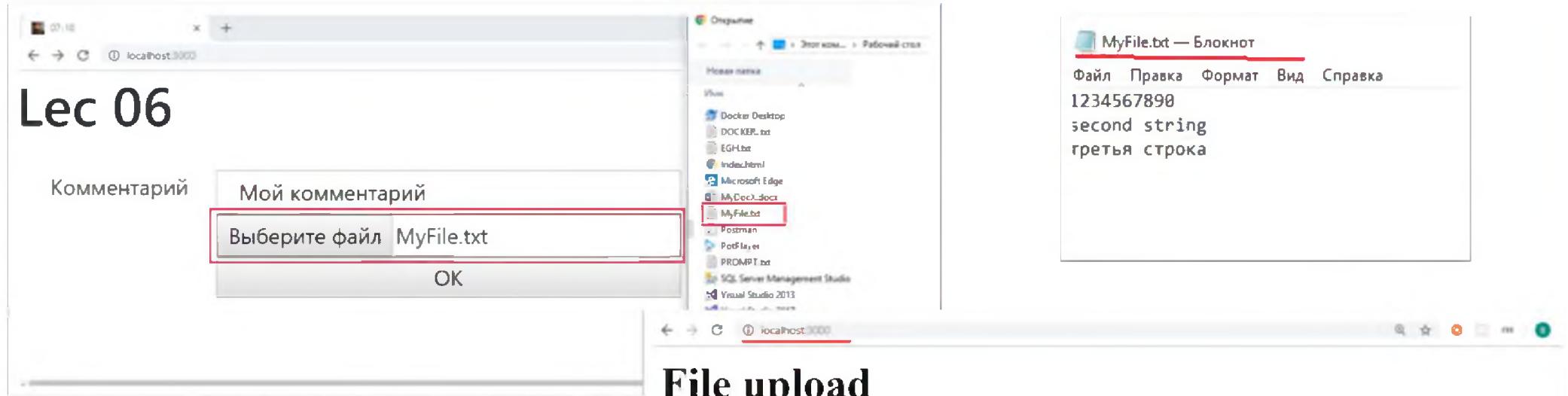
Чтобы преобразовать форму в составную форму, все, что нужно сделать, это изменить **атрибут enctype** тега form с application/x-www-form-urlencoded на **multipart/form-data**.

```

<!DOCTYPE html>
<html >
<head>
  <meta name="viewport" content="width=device-width" charset="utf-8" />
  <title>07-10</title>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
</head>
<body>
  <h1>Lec 06</h1>
  <div style="margin: 20px; width: 800px; padding: 5px;">
    <form method="POST" action="/" enctype="multipart/form-data">
      <div class ="row">
        <div class="col-2"><label>Комментарий</label> <input class="col-5" name ="comment" type="text" />
      </div>
      <div class ="row">
        <div class="col-5 offset-2" style="padding: 0px; border: 1px solid gray;" name ="file" type="file" />
      </div>
      <div class ="row">
        <div class="col-5 offset-2" style="padding: 0px; border: 1px solid gray;" name="upload" value="OK"/>
      </div>
    </form>
  </div>
</body>

```

Формат multipart/form-data



Здесь нужно отметить две вещи: **заголовок Content-Type** и **тело запроса** с формы.

Значение **Content-Type** заголовка очевидно **multipart/form-data**. Но он также имеет другое значение **boundary** (**разделитель**). Значение для него в примере генерируется браузером, но пользователь может определить его сам.

Тело запроса запроса **содержит** сами **поля формы**.

File upload

```
-----WebKitFormBoundaryoZNC2ONAJTu2ZKjX Content-Disposition: form-data; name="comment"  
Мой комментарий-----WebKitFormBoundaryoZNC2ONAJTu2ZKjX Content-Disposition: form-  
data; name="file"; filename="MyFile.txt" Content-Type: text/plain 1234567890 second string третья  
строка-----WebKitFormBoundaryoZNC2ONAJTu2ZKjX Content-Disposition: form-data;  
name="upload" OK-----WebKitFormBoundaryoZNC2ONAJTu2ZKjX--
```

Вся полезная нагрузка заканчивается **boundary** значением с суффиксом **--**.
Границное значение позволяет браузеру понять, когда и где каждое поле начинается и заканчивается.

```

let http = require('http');
let fs = require('fs');
let mp = require('multiparty') // npm install multiparty // https://github.com/pillarjs/multiparty

let handler = (req, res)=>{
  if (req.method == 'GET'){
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end(fs.readFileSync('./07-12.html'));
  }
  else if (req.method == 'POST'){
    let result = '';
    // req.on('data', (data)=>{result+=data;}) // все внутри multiparty
    // req.on('end', ()=>{ });
    let form = new mp.Form({uploadDir:'./files_07-12'});
    form.on('field', (name, value)=>{
      console.log('----- field -----');
      console.log(name, value);
      result += `<br />---${name} = ${value}`;
    });
    form.on('file', (name, file)=>{
      console.log('----- file -----');
      console.log(name, file);
      result += `<br />---${name} = ${file.originalfilename}: ${file.path}`;
    });
    form.on('error', (err)=> {
      console.log('----- err -----');
      console.log('err =', err);
      res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
      res.write('<h1>Form/Error</h1>');
      res.end();
    });
    form.on('close', ()=> {
      console.log('----- close -----');
      res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
      res.write('<h1>Form</h1>');
      res.end(result);
    });
    form.parse(req);
  }
}

```

Загрузка файла (upload)

Пакет **multiparty** используется для разбора HTTP-запросов с типом содержимого multipart/form-data.

new multiparty.Form создает новую форму. В параметре указываем каталог для размещения загружаемых файлов.

События:

- **field** возникает, когда при разборе формы появляется поле.
- **file** – по умолчанию multiparty не трогает жесткий диск. Но если добавить слушатель на это событие, то multiparty автоматически установит для **form.autoFiles** значение **true** и будет выполнять потоковую передачу на диск. **Error** – возникает при исключении.
- **close** возникает после того, как все части были проанализированы и отправлены.

С помощью **метода parse()** запускаем разбор формы из тела запроса.

```

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')}
  .on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})
  .on('request', handler)

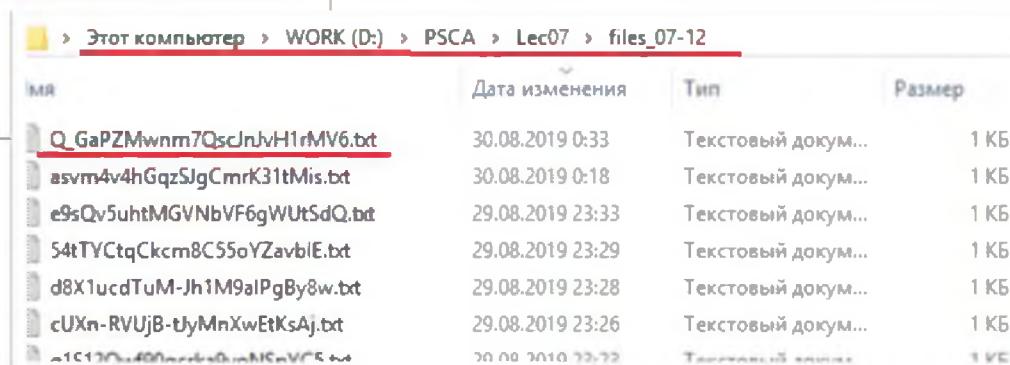
```

Демонстрация

← → ⌂ localhost:3000

Form

---comment = Мой комментарий
---file = MyFile.txt: files_07-12\Q_GaPZMwnm7QscJnJvH1rMV6.txt
---subok = OK



МЯ	Дата изменения	Тип	Размер
Q_GaPZMwnm7QscJnJvH1rMV6.txt	30.08.2019 0:33	Текстовый докум...	1 КБ
asvm4v4hGqzSjgCmrK31tMis.txt	30.08.2019 0:18	Текстовый докум...	1 КБ
e9sQv5uhtMGVNbVF6gWUtSdQ.txt	29.08.2019 23:33	Текстовый докум...	1 КБ
54tTYCtqCkcm8C55oYZavbIE.txt	29.08.2019 23:29	Текстовый докум...	1 КБ
d8X1ucdTuM-Jh1M9aIPgBy8w.txt	29.08.2019 23:28	Текстовый докум...	1 КБ
cUXn-RVUjB-tlyMnXwEtKsAj.txt	29.08.2019 23:26	Текстовый докум...	1 КБ
≈1C177n-40f0m-1b1m-011CmV7E.txt	29.08.2019 23:22	Текстовый докум...	1 КБ

Модуль `http` крайне низкоуровневый.

Обычно для разработки выбирают фреймворки, вот самые популярные:

- express
- nest
- koa
- hapi
- restify

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

HTTP-КЛИЕНТ (МОДУЛЬ HTTP)

HTTP-клиент

=

это программа, **устанавливающая HTTP-соединение** с целью отправки HTTP-запросов.

HTTP-клиент обычно является браузером, таким как Google Chrome или Opera, но также может быть и программой, запускаемой на сервере.

Модуль `http` =

модуль для создания низкоуровневого
HTTP-сервера и HTTP-клиента

Метод `http.request(url[, options][, callback])` и `http.request(options[, callback])` позволяет отправлять запросы.

Возвращает экземпляр [класса `http.ClientRequest`](#) (представляет собой `writable`-поток).

- ! При использовании `http.request()` всегда необходимо вызывать `req.end()`, чтобы обозначить конец запроса, даже если в тело запроса не записываются никакие данные.

`url` может быть строкой или объектом URL.

Если указаны и URL-адрес, и `options`, объекты объединяются, при этом `options` имеют приоритет. Необязательный параметр обратного вызова добавляется в качестве однократного слушателя события «`response`».

options

auth – user:password для настройки базовой аутентификации.

family – семейство IP-адресов, которое будет использоваться при разрешении хоста или имени хоста.

headers – объект, содержащий заголовки запросов.

host, hostname – доменное имя или IP-адрес сервера, на который будет отправлен запрос.

joinDuplicateHeaders – объединять ли значения нескольких заголовков в запросе с помощью ',' вместо того, чтобы отбрасывать дубликаты.

method – метод HTTP-запроса.

path – путь запроса. Должен включать строку запроса, если таковая имеется. Например: '/index.html?page=12'.

port – порт удаленного сервера.

protocol – протокол для использования.

timeout – время ожидания установки сокета в миллисекундах.

uniqueHeaders – список заголовков запроса, которые нужно отправлять только один раз. Если значение заголовка является массивом, элементы будут объединены с помощью ','.

Простейший клиент

```
const http = require('http');

const options = {
  host: 'localhost',
  path: '/mypath',
  port: 5000,
  method: 'GET'
}

const req = http.request(options, (res) => {
  console.log('method: ', res.method);
  console.log('response: ', res.statusCode);
  console.log('statusMessage: ', res.statusMessage);
  console.log('remoteAddress: ', res.socket.remoteAddress);
  console.log('remotePort: ', res.socket.remotePort);
  console.log('response headers: ', res.headers);

  let data = '';
  res.on('data', (chunk) => {
    console.log('data: body: ', data += chunk.toString('utf-8'));
  });
  res.on('end', () => { console.log('end: body: ', data); });
});

req.on('error', (e) => { console.log('error: ', e.message); });
req.end();
```

Коллбэк добавляется в качестве слушателя на **событие response**

Всегда необходимо вызывать **req.end()**, чтобы обозначить конец запроса

```
PS D:\NodeJS\samples\cwp_07> node .\09-01.js
method: GET
response: 200
statusMessage: OK
remoteAddress: ::1
remotePort: 5000
response headers: {
  'xxx-custom': 'custom header',
  date: 'Wed, 01 Nov 2023 18:59:47 GMT',
  connection: 'close',
  'transfer-encoding': 'chunked'
}
data: body: Hello, world
end: body: Hello, world
```

GET-запрос с параметрами

```
const http = require('http');
const qs = require('qs');

let params = qs.stringify({ x: 3, y: 4, s: 'xxx' });
let path = `/mypath?${params}`;
console.log('path: ', path);
```

```
const options = {
  host: 'localhost',
  path: path,
  port: 5000,
  method: 'GET'
}
```

```
const req = http.request(options, (res) => {
  console.log('method: ', res.method);
  console.log('response: ', res.statusCode);
  console.log('statusMessage: ', res.statusMessage);

  let data = '';
  res.on('data', (chunk) => { data += chunk.toString('utf-8'); });
  res.on('end', () => { console.log('data: body: ', data); });
});

req.on('error', (e) => { console.log('error: ', e.message); });
req.end();
```

Метод `qs.stringify()` создает строку запроса URL (query) из заданного объекта obj.

Экземпляр класса `http.ClientRequest`

Экземпляр класса `http.IncomingMessage`

```
PS D:\NodeJS\samples\cwp_07> node .\09-02.js
path: /mypath?x=3&y=4&s=xxx
method: GET
response: 200
statusMessage: OK
data: body: x + y = 7; s = xxx
PS D:\NodeJS\samples\cwp_07>
```

```
PS D:\NodeJS\samples\cwp_07> node .\09-02.js
path: /mypath?x=3&y=qwe&s=xxx
method: GET
response: 200
statusMessage: OK
data: body: invalid params
PS D:\NodeJS\samples\cwp_07>
```

События класса `http.ClientRequest` (запрос)

наследует `http.OutgoingMessage`

- `close` указывает, что запрос завершен или соединение было разорвано (до завершения ответа).
- `finish` генерируется при отправке запроса. Точнее, когда последний сегмент заголовков и тела передается ОС для передачи по сети. Это не означает, что сервер что-то получил.
- `information` генерируется, когда сервер отправляет промежуточный ответ 1xx (за исключением 101 Upgrade). Слушатели этого события получат объект, содержащий версию HTTP, код состояния, сообщение о состоянии, объект заголовков и массив с необработанными именами заголовков, за которыми следуют их соответствующие значения.
- `response` генерируется только один раз при получении ответа на этот запрос.
- `timeout` генерируется, когда время бездействия сокета истекло. Оно только уведомляет о том, что сокет бездействует, запрос необходимо уничтожить вручную.
- `upgrade` генерируется каждый раз, когда сервер отвечает на запрос 101 Upgrade. Если это событие не прослушивается, соединения клиентов, получающих заголовок Upgrade, будут закрыты. Этому событию будет передан экземпляр класса `<net.Socket>`.

[Подробнее тут](#)

Свойства класса http.ClientRequest (запрос)

- **destroyed** содержит true после вызова `request.destroy()`.
- **path** содержит путь запроса.
- **method** содержит метод запроса.
- **host** содержит имя хоста, с которого идет запрос.
- **protocol** содержит протокол запроса.
- **socket** содержит ссылку на сокет.
- **writableEnded** содержит true после вызова `request.end()`.
- **writableFinished** содержит true, если все данные были сброшены в базовую систему непосредственно перед отправкой события «`finish`».

Методы класса http.ClientRequest (запрос)

- `appendHeader(name, value)` добавляет одно значение заголовка для объекта заголовка.
- `end([data[, encoding]][, callback])` завершает отправку запроса. Если указаны данные, это эквивалентно вызову `request.write(data,coding)`, за которым следует `request.end(callback)`. Если указан обратный вызов, он будет вызван после завершения потока.
- `destroy([error])` уничтожает запрос. При необходимости генерирует событие «error» и событие «close». Вызов приведет к удалению оставшихся данных в ответе и уничтожению сокета.
- `getHeader(name)` считывает заголовок запроса. Имя не чувствительно к регистру. Тип возвращаемого значения зависит от аргументов, переданных в `request.setHeader()`.
- `getHeaderNames()` возвращает массив, содержащий уникальные имена текущих исходящих заголовков. Все имена заголовков записаны строчными буквами.
- `getHeaders()` возвращает неполную копию текущих исходящих заголовков. Ключами объекта являются имена заголовков, а значениями — значения заголовка. Все имена заголовков написаны строчными буквами.

Методы класса http.ClientRequest (запрос)

- `getRawHeaderNames()` возвращает массив, содержащий уникальные имена исходящих необработанных заголовков. Имена заголовков возвращаются с установленным регистром.
- `hasHeader(name)` возвращает true, если указанный заголовок установлен в исходящих заголовках. Имя заголовка не чувствительно к регистру.
- `removeHeader(name)` удаляет заголовок.
- `setHeader(name, value)` устанавливает одно значение заголовка. Если этот заголовок уже существует в заголовках, подлежащих отправке, его значение будет заменено. Используйте здесь массив строк, чтобы отправить несколько заголовков с одним и тем же именем.
- `write(chunk[, encoding][, callback])` отправляет часть тела. Этот метод можно вызывать несколько раз. Вызов `request.end()` необходим для завершения отправки запроса. Аргумент `callback` является необязательным и будет вызываться при сбросе этого фрагмента данных, но только если он не пуст. Возвращает true, если все данные были успешно сброшены в буфер ядра. Возвращает false, если все или часть данных были помещены в очередь в пользовательской памяти.

Объект полученного ответа является экземпляром класса `http.IncomingMessage`, рассмотренного в прошлой лекции

Класс http.IncomingMessage (ответ)

наследует stream.Readable

События

- **close** генерируется, когда запрос завершен.

Свойства

- **complete** имеет значение `true`, если полное HTTP-сообщение было получено и успешно проанализировано.
- **headers** содержит объект заголовков запроса/ответа. Пары ключ-значение имен и значений заголовков. Имена заголовков пишутся строчными буквами. Дубликаты объединяются
- **httpVersion** содержит версию HTTP, используемую клиентом.
- **method*** содержит метод запроса в виде строки
- **url*** содержит строку URL, на которую был отправлен запрос.

* Действительно только для запроса, полученного http.Server'ом

[Подробнее тут](#)

Класс http.IncomingMessage (ответ)

Свойства

- **rawHeaders** содержит необработанные заголовки запроса/ответа. Ключи и значения находятся в одном списке. Четные элементы являются ключевыми значениями, а нечетные — связанными значениями. Имена заголовков не пишутся строчными буквами, а дубликаты не объединяются.
- **socket** содержит объект класса net.Socket, связанный с соединением.
- **statusCode**** содержит трехзначный код состояния ответа HTTP
- **statusMessage**** содержит сообщение о состоянии HTTP-ответа

** Действительно только для ответа, полученного из http.ClientRequest.

Методы

- **destroy([error])** вызывает destroy() для сокета, получившего IncomingMessage.

```
const http = require('http');

let json = JSON.stringify({ x: 3, y: 4, str1: 'xxx', str2: 'yyy' });
console.log('json: ', json);

const options = {
  host: 'localhost',
  path: '/',
  port: 5000,
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': json.length,
    'Accept': 'application/json',
  }
}
const req = http.request(options, (res) => {
  console.log('method: ', req.method);
  console.log('response: ', res.statusCode);
  console.log('statusMessage: ', res.statusMessage);

  let data = '';
  res.on('data', (chunk) => {
    data += chunk.toString('utf-8');
  });
  res.on('end', () => {
    console.log('body: ', JSON.parse(data));
  });
});

req.on('error', (e) => { console.log('error: ', e.message); });

req.write(json);
req.end();
```

Отправка JSON

Для того, чтобы записать данные в тело запроса, необходимо передать их в метод **write(chunk[, encoding][, callback])**. Этот метод можно вызывать несколько раз. В конце обязательно должен быть вызван метод **end()** для завершения отправки запроса.

```
PS D:\NodeJS\samples\cwp_07> node .\09-03.js
json: {"x":3,"y":4,"str1":"xxx","str2":"yyy"}
method: POST
response: 200
statusMessage: OK
body: { sum: 7, concat: 'xxxxyyy' }
PS D:\NodeJS\samples\cwp_07> █
```

Отправка XML

```
const http = require('http');
const xmlbuilder = require('xmlbuilder');
const { parseString } = require('xml2js');

const xmldoc = xmlbuilder.create('students').att('faculty', 'ИТ').att('speciality', 'ИСИТ');

xmldoc.ele('student').att('id', '7000222').att('name', 'Иванов И.И.').att('bday', '2000-12-02')
    .up().ele('student').att('id', '7000223').att('name', 'Петров П.П.').att('bday', '2000-11-29').txt('Прошел собеседование в iTechArt')
    .up().ele('student').att('id', '7000228').att('name', 'Казан Н.А.').att('bday', '2001-09-11');

const options = {
    host: 'localhost', path: '/', port: 5000, method: 'POST',
    headers: { 'Content-type': 'text/xml', 'Accept': 'text/xml' }
}

const req = http.request(options, (res) => {
    let data = '';
    res.on('data', (chunk) => { data += chunk; });
    res.on('end', () => {
        console.log('body =', data);
        parseString(data, (err, str) => {
            if (err) console.log('xml parse error');
            else {
                console.log('str =', str);
                console.log('str.result =', str.result);
            }
        });
    });
});

req.on('error', (e) => { console.log('error: ', e.message); });

req.end(xmldoc.toString({ pretty: true }));
```

```
PS D:\NodeJS\samples\cwp_07> node .\09-09.js
body = <result>
<students faculty="ИТ" speciality="ИСИТ">
    <quantity value="3"/>
</students>
</result>
```

```
str = { result: { students: [ [Object] ] } }
str.result = { students: [ { '$': [Object], quantity: [Array] } ] }
PS D:\NodeJS\samples\cwp_07>
```

```
{
    '$': { faculty: 'ИТ', speciality: 'ИСИТ' },
    quantity: [ [Object] ]
}
```

```
str.result.students.quantity = [ { '$': { value: '3' } } ]
```

Загрузка файла на сервер

```
let http = require('http');
let fs   = require('fs');

let bound = 'smw60-smw60-smw60';
let body  = '--${bound}\r\n';
body += 'Content-Disposition:form-data; name="file"; filename="MyFile.bmp"\r\n';
body += 'Content-Type:application/octet-stream\r\n\r\n';

let options = {
  host:'localhost',
  path: '/mypath',
  port: 3000,
  method:'POST',
  headers:{'content-type':'multipart/form-data; boundary='+bound}
}
let req = http.request(options, (res) => {

  let data = '';
  res.on('data', (chunk) => {data += chunk;});
  res.on('end', () => { console.log('http.response: end: length body =', Buffer.byteLength(data)); });

});
req.on('error', (e) => { console.log('http.request: error:', e.message);});
req.write(body); // отправляем 1 часть

let stream = new fs.ReadStream('D:\\xxx.bmp');
stream.on('data', (chunk)=>{req.write(chunk); console.log( Buffer.byteLength(chunk))}); // отправляем 2ю часть порциями
stream.on('end', ()=>{req.end(`\r\n--${bound}--\r\n`)}); // отправляем 3ю часть
```

Скачивание файла на сервер

```
const http = require('http');
const fs = require('fs');

const file = fs.createWriteStream("file.bmp");

let options = {
  host:'localhost',
  path: '/bmp/MyFile.bmp',
  port: 3000,
  method: 'GET'
}

const req = http.request(options, (res) => { res.pipe(file); });
req.on('error', (e) => { console.log('http.request: error:', e.message);});
req.end();
```

Также **для разработки HTTP-клиента** можно использовать такие пакеты, как:

- axios
- needle
- superagent
- node-fetch

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

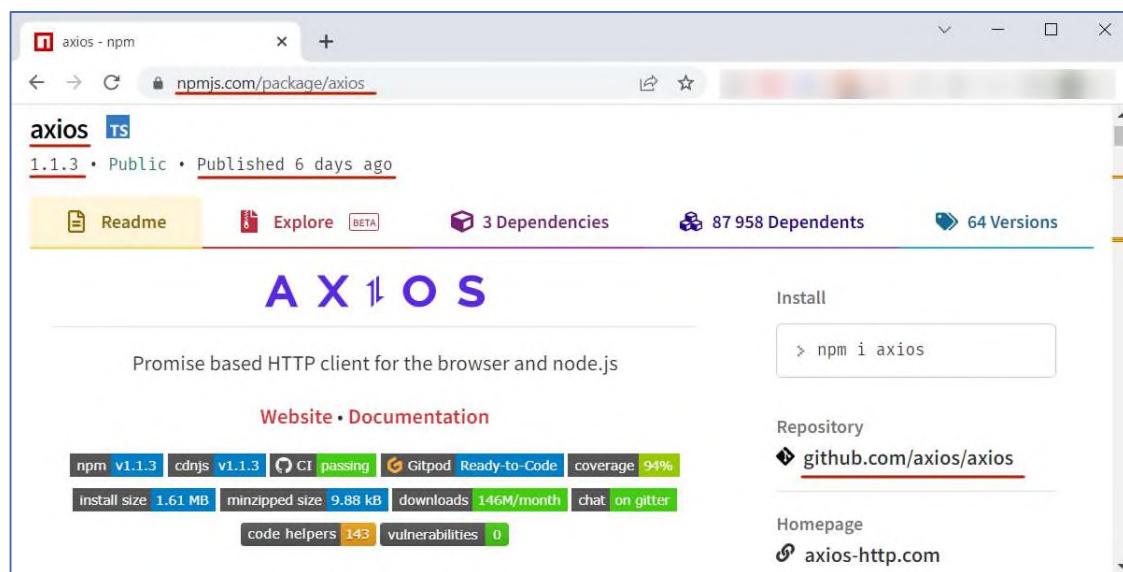
AXIOS

Axios

=

это пакет для разработки HTTP-клиента **на основе Promise**, работающего и в браузере, и в среде Node.js.

На стороне сервера он использует нативный node.js http-модуль, тогда как на стороне клиента (браузер) он использует XMLHttpRequests.



[документация](#)

GET-запрос с query-параметрами

axios(config)

```
const axios = require('axios');

// let query = require('qs');
// let parms = query.stringify({ x: 34, y: 12, s: 'xxx' });

let parms = new URLSearchParams({ x: 34, y: 12, s: 'xxx' }).toString();

function httpRequest() {
  return axios({
    url: `http://localhost:5000/?${parms}`,
    method: 'get'
  });
}

httpRequest()
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error(error)
  })
}
```

Так как axios возвращает Promise, одним из способов обработки результата являются **обработчики then(), catch(), finally()**

Это можно сделать явно указав параметры, предварительно преобразовав их в строку соответствующего формата с помощью **qs.stringify()**, или **URLSearchParams**, или можно использовать свойство **params** в **options**.

Вместо модуля qs можно использовать:

- **stringify(...)** => `new URLSearchParams(...).toString()`
- **parse(...)** => `new URLSearchParams(...).get(...)`

```
PS D:\Материалы\cwp_07> node 07-01c
x = 34; y = 12
```

GET-запрос с query-параметрами axios(config)

```
const axios = require('axios');

async function httpRequest() {
  try {
    let response = await axios({
      url: `http://localhost:5000/`,
      method: 'get',
      params: { x: 34, y: 12, s: 'xxx' }
    });
    console.log(response.data);
  } catch (error) {
    console.error(error)
  }
}

httpRequest();
```

Указав в объекте **свойство params** прикрепляются query-параметры.

Также промис возвращаемый axios можно обработать **конструкцией async/await**. Однако, для обработки ошибок необходимо оберачивать в **try-catch**.

```
PS D:\Материалы\cwp_07> node 07-01c
x = 34; y = 12
```

POST-запрос с телом (формат JSON)

axios(config)*

также могут быть использованы
axios(url[, config]), axios.request(config)

```
const axios = require('axios');
let query = require('querystring');

let parms = query.stringify({ x: 34, y: 12, s: 'xxx' });

async function httpRequest() {
  try {
    let response = await axios({
      url: `http://localhost:5000/`,
      method: 'post',
      //data: parms,
      data: { x: 34, y: 12, s: 'xxx' }
    });
    console.log(response.data);
  } catch (error) {
    console.error(error)
  }
}

httpRequest();
```

Можно передать информацию в тело через **свойство data** конфига. Можно передать параметры в строковом виде (ключ, значение) или объект. Объект впоследствии будет **автоматически преобразован в JSON**.

```
PS D:\Материалы\cwp_07> node 07-01c
x=34&y=12&s=xxx
PS D:\Материалы\cwp_07> node 07-01c
{ x: 34, y: 12, s: 'xxx' }
```

request config

```
let config = {
  url: '/user',
  method: 'get',                                // по умолчанию get
  baseURL: 'https://some-domain.com/api/',      // будет добавлен к url, если url не является абсолютным
  headers: { 'X-Requested-With': 'XMLHttpRequest' },

  params: {
    ID: 12345                                    // query-параметры
  },

  data: {
    firstName: 'Fred'                           // данные в теле запроса
  },
  data: 'Country=Brasil&City=Belo Horizonte', // 2-ой вариант передачи данных в теле

  // кол-во мс до истечения времени ожидания запроса
  // истекает timeout => запрос прерывается
  timeout: 1000,                                 // по умолчанию 0 (по timeout),

  responseType: 'json',                          // по умолчанию json
  responseEncoding: 'utf8',                      // по умолчанию utf-8
}
```

Здесь также можно настроить прокси, количество допустимых редиректов, аутентификацию и др.
Подробнее [здесь](#)

GET-запрос с query-параметрами

axios.get(url[, config])

```
const axios = require('axios');
let query = require('querystring');

let params = query.stringify({ x: 34, y: 12, s: 'xxx' });

async function httpRequest() {
  try {
    // const URL1 = `http://localhost:5000/?${params}`;
    // const response = await axios.get(URL1);
    const URL2 = `http://localhost:5000/`;
    const response = await axios.get(URL2, {
      params: { x: 34, y: 12, s: 'xxx' },
      headers: { "My-Header": "text" }
    });
    console.log(response.data);
  } catch (error) {
    console.error(error);
  }
}

httpRequest();
```

При использовании сокращений `url`, `method` и `data` свойства **не нужно указывать в config**.

Для удобства псевдонимы были предоставлены для всех распространенных методов запроса. Поэтому обычно используются `axios.get()`, `axios.post()`, ...

```
axios.get(url[, config])
axios.delete(url[, config])
axios.head(url[, config])
axios.options(url[, config])
axios.post(url[, data[, config]])
axios.put(url[, data[, config]])
axios.patch(url[, data[, config]])
```

Заголовки запроса добавляются через **свойство header** параметра конфигурации запроса.

```
PS D:\Материалы\cwp_07> node 07-02c
x = 34; y = 12
PS D:\Материалы\cwp_07> node 07-02c
x = 34; y = 12
```

Свойства объекта response

```
const axios = require('axios');
let query = require('querystring');

let params = query.stringify({ x: 34, y: 12, s: 'xxx' });

async function httpRequest() {
  try {
    // const URL1 = `http://localhost:5000/?${params}`;
    // const response = await axios.get(URL1);
    const URL2 = `http://localhost:5000/`;
    const response = await axios.get(URL2, {
      params: { x: 34, y: 12, s: 'xxx' },
      headers: { "My-Header": "text" }
    });
    console.log("Data: ", response.data); // информация из тела ответа
    console.log("Status code: ", response.status); // статус код
    console.log("Status message: ", response.statusText); // пояснение к статус коду
    console.log("Headers: ", response.headers); // заголовки ответа
    console.log("Config: ", response.config); // config с настройками запроса
    console.log("Request object: ", response.request); // объект отправленного запроса; экземпляр http.ClientRequest
    console.log(response.request.method); // метод
    console.log(response.request.path); // путь
    console.log(response.request.getHeader("My-Header")); // заголовок запроса
  } catch (error) {
    console.error(error);
  }
}

httpRequest();
```

```
PS D:\Материалы\свр_07> node 07-02c
Data: x = 34; y = 12
Status code: 200
Status message: OK
Headers: {
  'date': 'Fri, 16 Oct 2020 18:41:19 GMT',
  'connection': 'close',
  'content-length': '14'
}
Config: {
  url: 'http://localhost:5000/',
  method: 'get',
  headers: {
    'Accept': 'application/json, text/plain, */*',
    'My-Header': 'text',
    'User-Agent': 'axios/0.20.0'
  },
  params: { x: 34, y: 12, s: 'xxx' },
  transformRequest: [ [Function: transformRequest] ],
  transformResponse: [ [Function: transformResponse] ],
  timeout: 0,
  adapter: [Function: httpAdapter],
  xsrfCookieName: 'XSRF-TOKEN',
  xsrfHeaderName: 'X-XSRF-TOKEN',
  maxContentLength: -1,
  maxBodyLength: -1,
  validateStatus: [Function: validateStatus],
  data: undefined
}
Request object: ClientRequest {
  events: [Object: null prototype] {
    [Symbol(corked)]: 0,
    [Symbol(koutHeaders)]: [Object: null prototype] {
      accept: [ 'Accept', 'application/json, text/plain, */*' ],
      'my-header': [ 'My-Header', 'text' ],
      'user-agent': [ 'User-Agent', 'axios/0.20.0' ],
      host: [ 'Host', 'localhost:5000' ]
    }
  }
  GET
  /?x=34&y=12&s=xxx
  text
}
PS D:\Материалы\свр_07>
```

POST-запрос с телом (формат application/www-form-urlencoded)

axios.post(url[, data[, config]])

```
const axios = require('axios');
let query = require('querystring');

let parms = query.stringify({ x: 34, y: 12, s: 'xxx' });

async function httpRequest() {
  try {
    //let response = await axios.post('http://localhost:5000/', parms);
    let response = await axios.post('http://localhost:5000/', {data: parms});
    console.log(response.data);
  }
  catch (error) {
    console.error(error)
  }
}

httpRequest();
```

```
PS D:\Материалы\cwp_07> node .\07-01c.js
x=34&y=12&s=xxx
PS D:\Материалы\cwp_07> node .\07-01c.js
{ data: 'x=34&y=12&s=xxx' }
```

Второй аргумент в axios.post() – строка или объект, содержащий параметры POST, будет записывать в тело. Альтернатива – передавать содержимое через свойство **data** конфига.

POST-запрос с телом (формат JSON) axios.post(url[, data[, config]])

```
const axios = require('axios');

let jsonm = JSON.stringify(
{
    __comment: "Запрос. Лекция 7",
    x: 1,
    y: 2,
    s: "Сообщение",
    m: ["a", "b", "c", "d"],
    o: { "surname": "Иванов", name: "Иван" }
})

async function httpRequest() {
    try {
        const config = {
            headers: { "Content-Type": "application/json", "Accept": "application/json" },
            data: jsonm
        };
        //const response = await axios.post('http://localhost:5000/', jsonm);
        const response = await axios.post('http://localhost:5000/', config);
        console.log(response.data);
    } catch (error) {
        console.error(error);
    }
}

httpRequest();
```

Axios чаще всего парсит ответ на основе заголовка Content-Type ответа HTTP.

JSON => JS-объект
HTML => строку

```
PS D:\Материалы\свр_07> node 07-03c
{
    __comment: 'Ответ. Лекция 7',
    x_plus_y: 3,
    Concatenation_s_o: 'Сообщение: Иванов, Иван',
    Length_m: 4
}
```

POST-запрос с телом (формат XML)

axios.post(url[, data[, config]])

```
const axios = require('axios');
let xmlbuilder = require('xmlbuilder');
let parseString = require('xml2js').parseString;

let xmldoc = xmlbuilder.create('request').att('id', '28')
  .ele('x').att('value', '1')
  .up().ele('x').att('value', '2')
  .up().ele('m').att('value', 'a')
  .up().ele('m').att('value', 'b')
  .up().ele('m').att('value', 'c')

async function httpRequest() {
  try {
    const config = {
      headers: { "Content-Type": "text/xml", "Accept": "text/xml" },
      data: xmldoc.toString({ pretty: true })
    };
    const response = await axios.post('http://localhost:5000/', config);
    let data = parseString(response.data, (err, str) => {
      if (err) console.log('xml parse error');
      else {
        console.dir(str, { depth: null });
      }
    })
  } catch (error) {
    console.error(error);
  }
}

httpRequest();
```

```
PS D:\Материалы\cwp_07> node 07-05c
{
  response: [
    '$': { id: '33', request: '28' },
    sum: [ { '$': { element: 'x', result: '3' } } ],
    concat: [ { '$': { element: 'm', result: 'abc' } } ]
  ]
}
```

Скачивание файла

```
const axios = require('axios');
const fs = require('fs');

async function getImage() {
  try {
    let config = {
      responseType: 'stream'
    };
    let response = await axios.get('http://localhost:5000/', config);
    response.data.pipe(fs.createWriteStream('image.jpg'));
  }
  catch (error) {
    console.error(error);
  }
}
getImage();
```

Можно настроить тип данных с помощью **свойства responseType**. По умолчанию responseType имеет значение json, что означает, что axios попытается проанализировать ответ как JSON. Однако это неверно, если вы хотите, например, скачать изображение с помощью axios.

Можно установить responseType в **'arraybuffer'**, чтобы получить ответ в виде ArrayBuffer, или в **'stream'**, чтобы получить ответ в виде потока Node.js.

Загрузка файла

```
const axios = require('axios');
const fs = require('fs');
const FormData = require('form-data'); // npm install form-data

async function uploadFile() {
  try {
    let form = new FormData();
    form.append('file', fs.createReadStream(__dirname + '/MyFile.txt'),
      { knownLength: fs.statSync(__dirname + '/MyFile.txt').size });

    let config = {
      headers: {
        ...form.getHeaders(), // возвращает Object с Content-Type и boundary
        "Content-Length": form.getLengthSync()
      }
    };

    axios.post('http://localhost:5000/', form, config);

  } catch (error) {
    console.error(error);
  }
}

uploadFile();
```

Оператор ... (spread) получает все свойства объекта, а затем перезаписывает существующие свойства теми, которые передаются, то есть по сути этот оператор **создает копию**.

Пакет **formdata** позволяет отправлять формы и загружать файлы на сервер.

Метод **form.getHeaders()** добавляет корректный заголовок Content-Type в пользовательские заголовки запроса (в том числе сгенерированный разделитель).

Метод **form.getLengthSync()** синхронно возвращает значение Content-Length.

```
✓ upload
Ξ ErPN4ycn7ICf9D85tMHwCCkk.txt
```

axios.all([...]) (deprecated) => Promise.all([...])

```
const axios = require('axios');

let parms1 = new URLSearchParams({ x: 34, y: 12, s: 'xxx' }).toString();
let parms2 = new URLSearchParams({ x: 24, y: 44, s: 'aaa' }).toString();

function httpRequest() {
  return Promise.all([
    axios.get(`http://localhost:5000/?${parms1}`),
    axios.get(`http://localhost:5000/?${parms2}`)
  ]);
}

httpRequest()
  .then(response) => {
  console.log(response[0].data);
  console.log(response[1].data);
}
  .catch(error => {
  console.log(error);
});
```

Для выполнения **параллельных запросов** можно использовать **Promise.all()**.

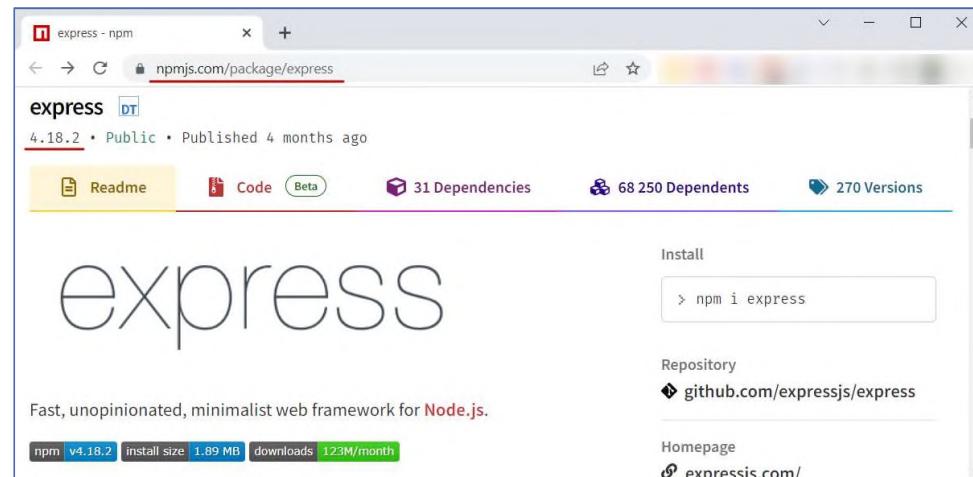
```
PS D:\Материалы\cwp_07> node 07-09c
x = 34; y = 12
x = 24; y = 44
```

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

EXPRESS

Express =

это веб-фреймворк, который предоставляет ряд готовых абстракций и упрощает создание сервера (обработка форм, работа с cookie, CORS и т.д. Он использует модуль http.



документация

Создание сервера. Промежуточные обработчики (middleware)

```
const express = require("express"); // npm install express
const app = express();

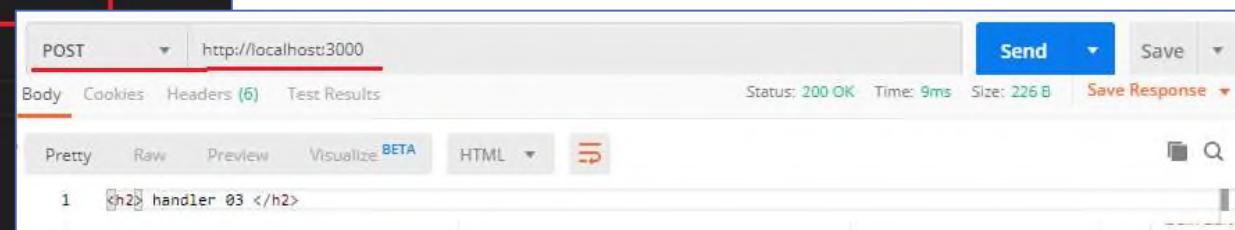
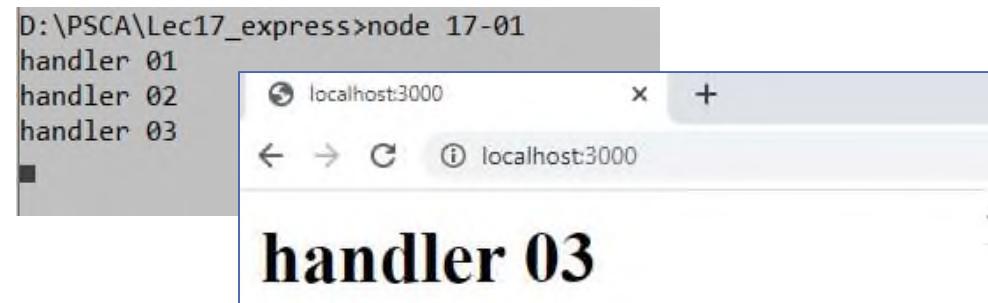
app.use((req, res, next)=>{           //middleware
  console.log('handler 01');
  next();
});

app.use((req, res, next)=>{           //middleware
  console.log('handler 02');
  next();
});

app.use((req, res, next)=>{           //middleware
  console.log('handler 03');
  res.send("<h2> handler 03 </h2>");
});

app.listen(3000);
```

Когда сервер получает запрос, этот запрос передается в **конвойер обработки запроса**. Можно встроить в конвойер обработки запроса на любом этапе функцию **middleware** (промежуточные обработчики). Для этого применяется метод **app.use()**.



Middleware

Функция, которая передается в `app.use()`, принимает три параметра:

- `request`: объект запроса;
- `response`: объект ответа;
- `next`: следующая в конвейере обработки запроса функция.

```
app.use('/path', (request, response, next) => {  
  // ...  
})
```

Функции middleware могут сопоставляться с определенными маршрутами.

Middleware

Может:

- выполнять любой код;
- изменять объекты запроса и ответа;
- заканчивать цикл обработки, отправив ответ (res.send())
- вызывать следующий middleware из очереди.

Обязан:

- завершить запрос (res.send());
- или вызвать следующий middleware (next()) .

Типы middleware

1. Промежуточные обработчики уровня приложения

```
app.use('/', (req, res, next) => {
  console.log('application-level middleware');
  next();
});
```

2. Промежуточные обработчики уровня роутера

```
let router = express.Router();
router.use('/', (req, res, next) => {
  console.log('router-level middleware');
  next();
})
```

Типы middleware

3. Встроенные промежуточные обработчики

```
app.use(express.static('./public'));
```

4. Сторонние промежуточные обработчики

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
```

5. Промежуточные обработчики для обработки ошибок

```
app.use((err, req, res, next) => {
  console.error('error-handler', err);
  res.status(500).send('error handler');
});
```

Маршрутизация

app.МЕTHOD(path, handler)

Приложение «прослушивает» запросы, соответствующие указанным **маршрутам** и **методам**, и когда оно обнаруживает совпадение, оно вызывает указанную **функцию обратного вызова**.

- ! При наличии нескольких функций обратного вызова важно указать next в качестве параметра функции обратного вызова, а затем вызвать next() в теле функции, чтобы передать управление следующему обратному вызову.

Маршрутизация (методы)

Метод маршрута **является производным от** одного из **методов HTTP** и присоединяется к экземпляру express.

```
app.get('/', (req, res, next) => { });
app.post('/', (req, res, next) => { });
app.delete('/', (req, res, next) => { });
```

Существует специальный метод маршрутизации **app.all()**, используемый для отлова запросов всех методов HTTP-запросов.

```
app.all('/', (req, res, next) => { });
```

Для методов, которые преобразуются в недопустимые имена переменных JavaScript, используется нотация в квадратных скобках.

```
app['m-search']('/', (req, res, next) => { });
```

Маршрутизация (методы)

- checkout
- copy
- delete
- get
- head
- lock
- merge
- mkactivity
- mkcol
- move
- m-search
- notify
- options
- patch
- post
- purge
- put
- report
- search
- subscribe
- trace
- unlock
- unsubscribe

```

const express = require("express"); // npm install express
const app = express();

app.use((req, res, next)=>{console.log('handler 01'); next()}); // middleware
app.use((req, res, next)=>{console.log('handler 02'); next()}); // middleware
app.use((req, res, next)=>{console.log('handler 03'); next()}); // middleware

app.get('/', (req, res)=>{                                         // обработка get-запросов
  console.log('get / handler 04');
  res.send("<h2> handler 04 </h2>");
})

app.post('/', (req, res)=>{                                         // обработка post-запросов
  console.log('post / handler 05');
  res.send("<h2> handler 05 </h2>");
})

app.put('/', (req, res)=>{                                         // обработка put-запросов
  console.log('put / handler 06');
  res.send("<h2> handler 06 </h2>");
})

app.delete('/', (req, res)=>{                                         // обработка delete-запросов
  console.log('delete / handler 07');
  res.send("<h2> handler 07 </h2>");
})

var server = app.listen(3000); // = http.Server

```

```

D:\PSCA\Lec17_express>node 17-02
handler 01
handler 02
handler 03
get / handler 04
handler 01
handler 02
handler 03
put / handler 06

```

localhost:3000

localhost:3000

handler 04

PUT https://localhost:3000

Body Cookies Headers (6) Test Results

Status: 200 OK Time: 2ms Size: 226 B Save

Pretty Raw Preview Visualize BETA

HTML

1 <h2> handler 06 </h2>

```

const express = require('express');
const app = express();

app.get('/', (req, res, next) => {
  console.log('get /');
  next();
});

app.post('/', (req, res, next) => {
  console.log('post /');
  next();
});

app.put('/', (req, res, next) => {
  console.log('put /');
  next();
});

app.delete('/', (req, res, next) => {
  console.log('delete /');
  next();
});

app.all('/', (req, res, next) => {
  console.log('all /, ', req.method);
  res.send('ALL: ' + req.method);
});

app.listen(3000);

```

The screenshot shows a browser window with five tabs, each representing a different HTTP method on the URL `http://localhost:3000/`. The tabs are labeled `GET`, `POST`, `PUT`, `DELETE`, and `PROPFIND`. Each tab displays a list of requests, all of which are labeled `ALL: [method]`, where [method] corresponds to the tab's label. The browser interface includes tabs for `Params`, `Authorization`, `Headers`, `Test Results`, `Pretty`, `Raw`, `Preview`, and `Visualize`.

GET http://localhost:3000/

POST http://localhost:3000/

PUT http://localhost:3000/

DELETE http://localhost:3000/

PROPFIND http://localhost:3000/

```

PS D:\NodeJS\samples\cwp_17> node 17-05
get /
all /, GET
post /
all /, POST
put /
all /, PUT
delete /
all /, DELETE
all /, PROPFIND
[]
```

Маршрутизация (путь)

Путь, для которого вызывается функция; может быть представлен в виде:

- 1) строки;
- 2) паттерна (преобразуется в регулярное выражение);
- 3) регулярного выражения.

Маршрутизация (путь – строка)

```
app.get('/', (req, res, next) => {
  res.send('get / ');
});
```

GET

Params Authorization Headers (8) Body

Body Cookies (1) Headers (7) Test Results

Pretty Raw Preview Visualize

```
1  get /
```

GET

Params Authorization Headers (8) Body

Body Cookies (1) Headers (8) Test Results

Pretty Raw Preview Visualize

```
Cannot GET /a
```

Маршрутизация (путь – строка)

```
app.get('/home', (req, res, next) => {
  res.send('get /home');
});
```

GET Params Authorization Headers (8) Body
Body Cookies (1) Headers (8) Test Results
Pretty Raw Preview Visualize

Cannot GET /

GET Params Authorization Headers (8) Body
Body Cookies (1) Headers (7) Test Results
Pretty Raw Preview Visualize

1 get /home

GET Params Authorization Headers (8) Body
Body Cookies (1) Headers (8) Test Results
Pretty Raw Preview Visualize

Cannot GET /home2

Маршрутизация (путь – строка)

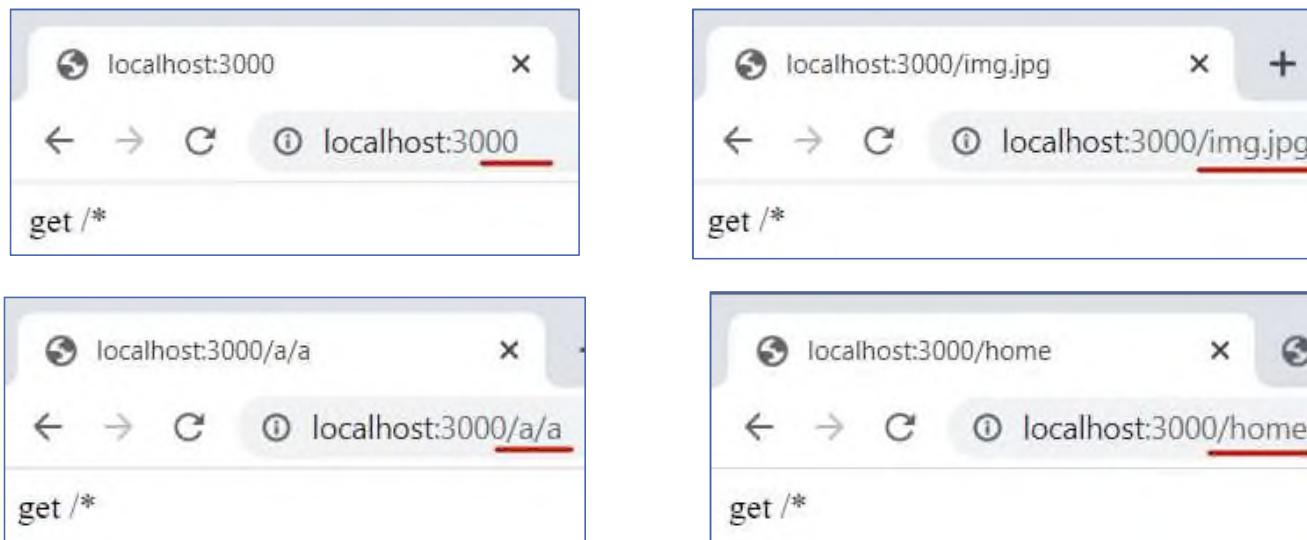
```
app.get('/image.png', (req, res, next) => {
  res.send('get /image.png');
});
```

GET	▼	http://localhost:3000/	GET	▼	http://localhost:3000/image.png	GET	▼	http://localhost:3000/image.jpg				
Params		Authorization	Headers (8)	Body	Params	Authorization	Headers (8)	Body				
Body		Cookies (1)	Headers (8)	Test Results	Body	Cookies (1)	Headers (7)	Test Results				
Pretty		Raw	Preview	Visualize	Pretty	Raw	Preview	Visualize				
Cannot GET /					get /image.png				Cannot GET /image.jpg			

Маршрутизация (путь – паттерн)

* – произвольный набор символов

```
app.get('*', (req, res, next) => {
  res.send('get /*');
});
```



Маршрутизация (путь – паттерн)

* – произвольный набор символов

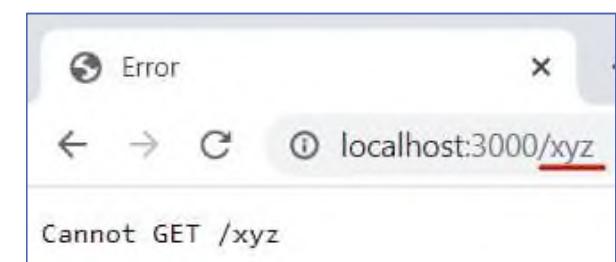
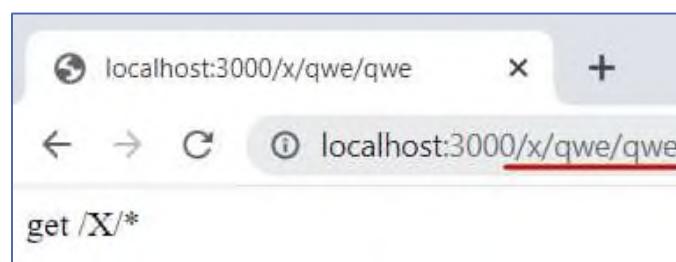
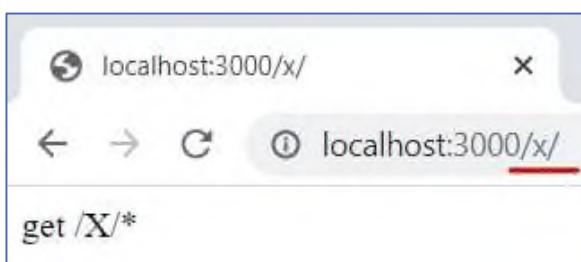
```
app.get('*x', (req, res, next) => {
  res.send('get *x');
});
```



Маршрутизация (путь – паттерн)

* – произвольный набор символов

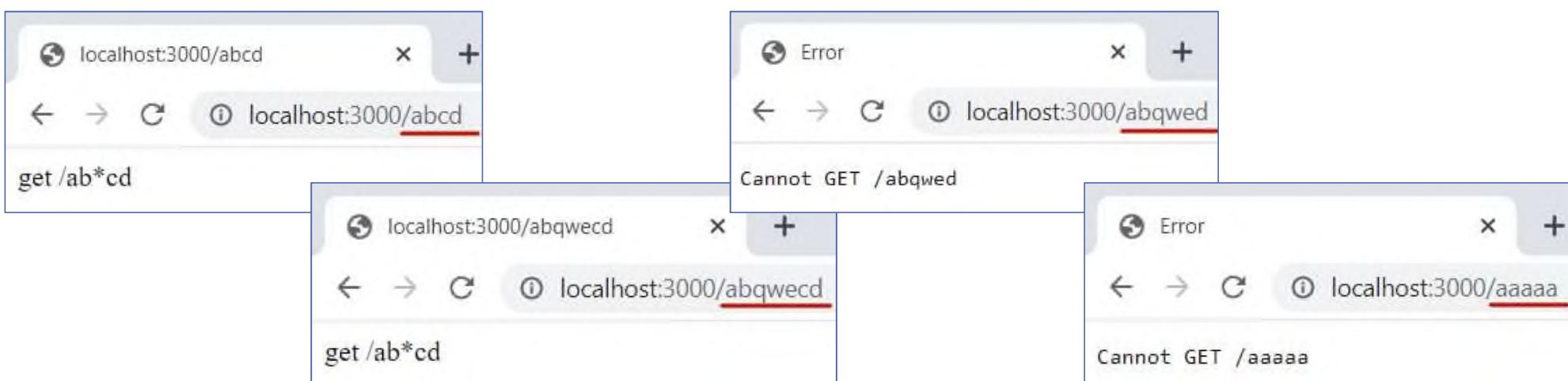
```
app.get('/X/*', (req, res, next) => {
  res.send('get /X/*');
});
```



Маршрутизация (путь – паттерн)

* – произвольный набор символов

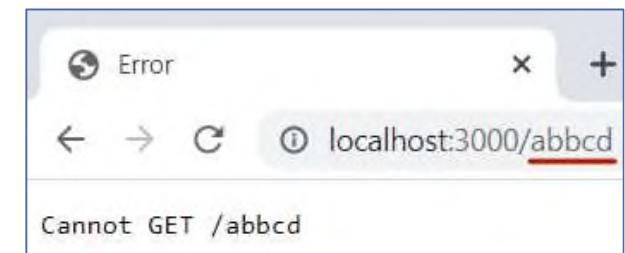
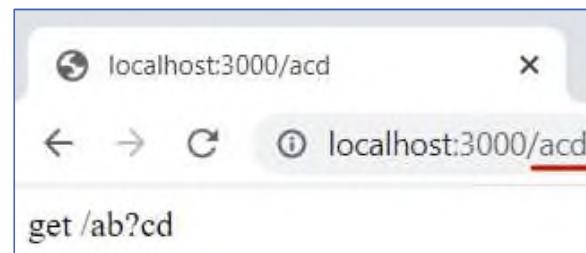
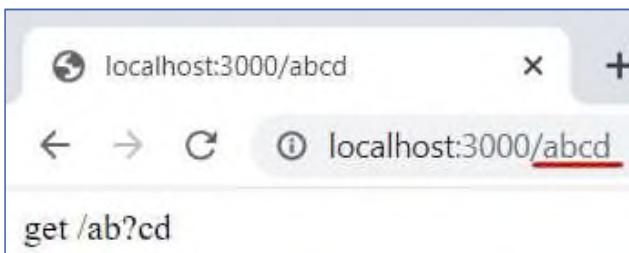
```
app.get('/ab*cd', (req, res, next) => {
  res.send('get /ab*cd');
});
```



Маршрутизация (путь – паттерн)

? – символ перед знаком ? может встречаться 0 или 1 раз

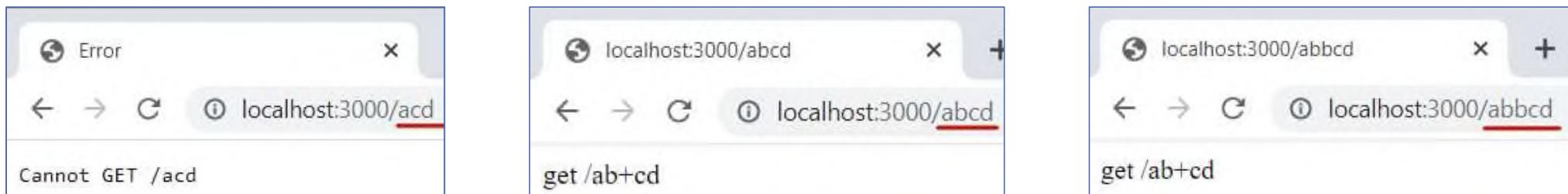
```
app.get('/ab?cd', (req, res, next) => {
  res.send('get /ab?cd');
});
```



Маршрутизация (путь – паттерн)

+ – символ перед знаком + может встречаться 1 и более раз

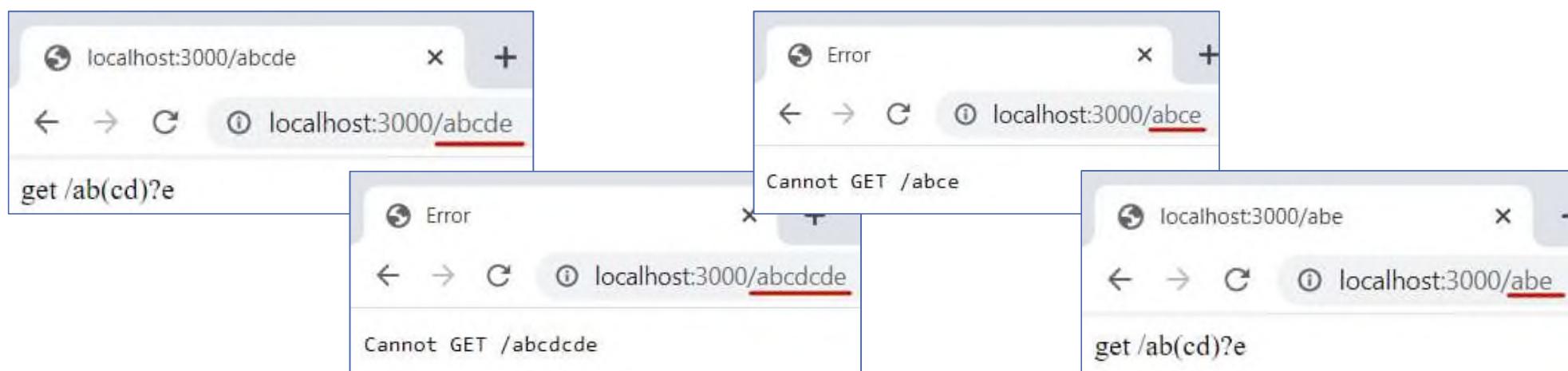
```
app.get('/ab+cd', (req, res, next) => {
  res.send('get /ab+cd');
});
```



Маршрутизация (путь – паттерн)

() – группировка символов

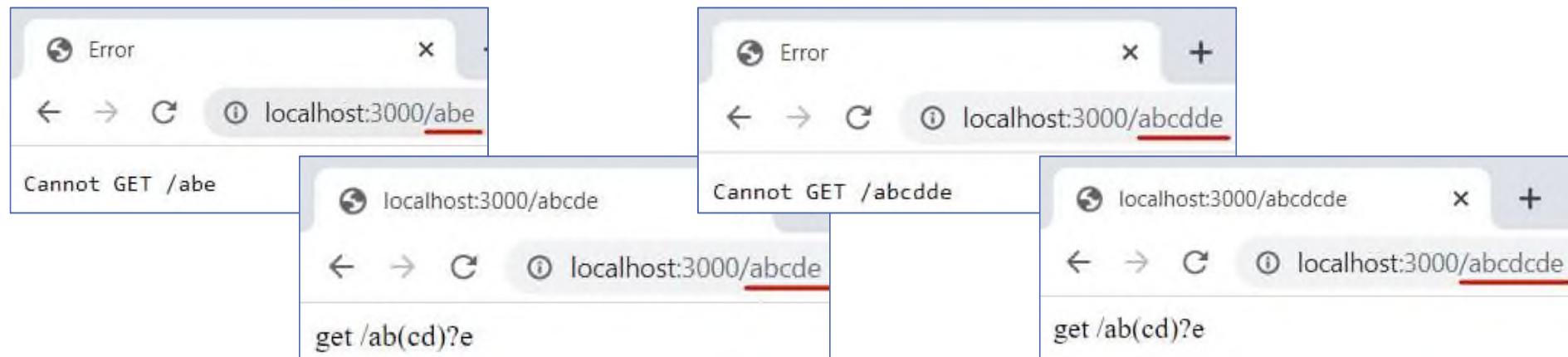
```
app.get('/ab(cd)?e', (req, res, next) => {
  console.log('get /ab(cd)?e');
  res.send('get /ab(cd)?e');
});
```



Маршрутизация (путь – паттерн)

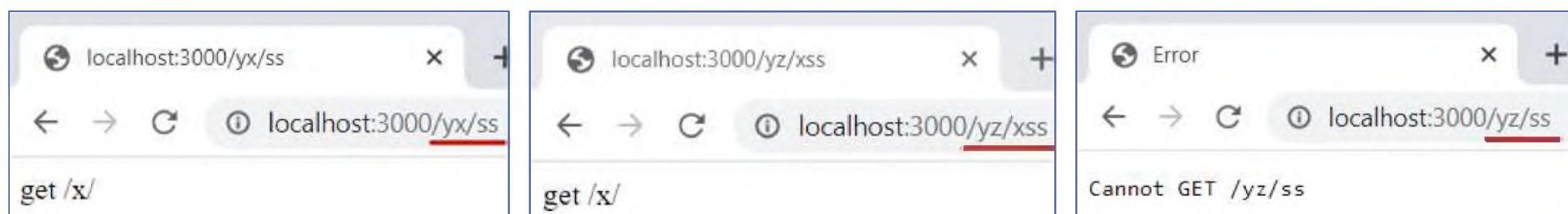
() – группировка символов

```
app.get('/ab(cd)+e', (req, res, next) => {
  res.send('get /ab(cd)?e');
});
```



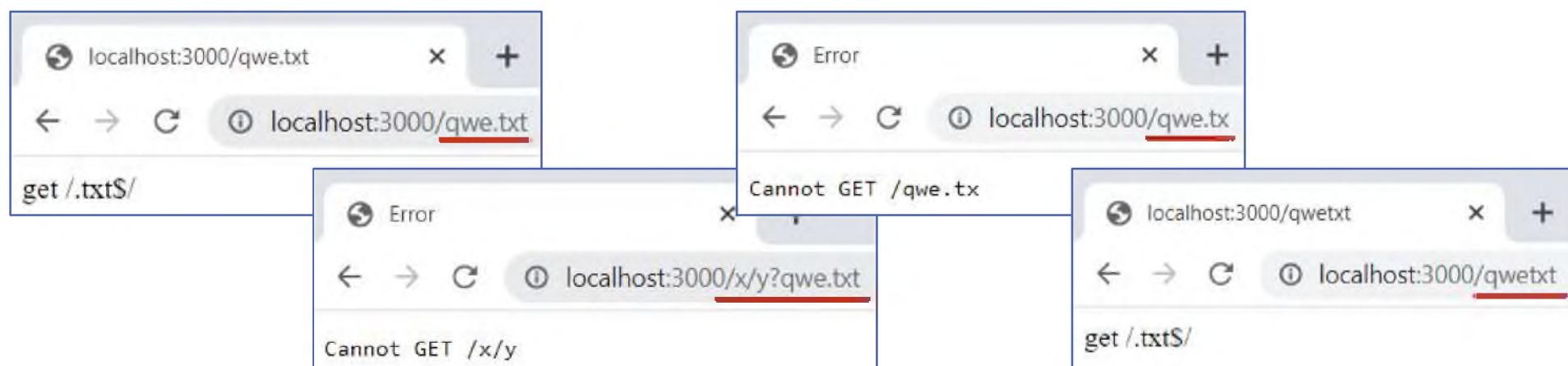
Маршрутизация (путь – регулярное выражение)

```
app.get(/x/, (req, res) => {
  res.send('/x/')
})
```



Маршрутизация (путь – регулярное выражение)

```
app.get('.txt$', (req, res) => {
  res.send('get /.txt$/')
})
```



Конвейер обработки запроса

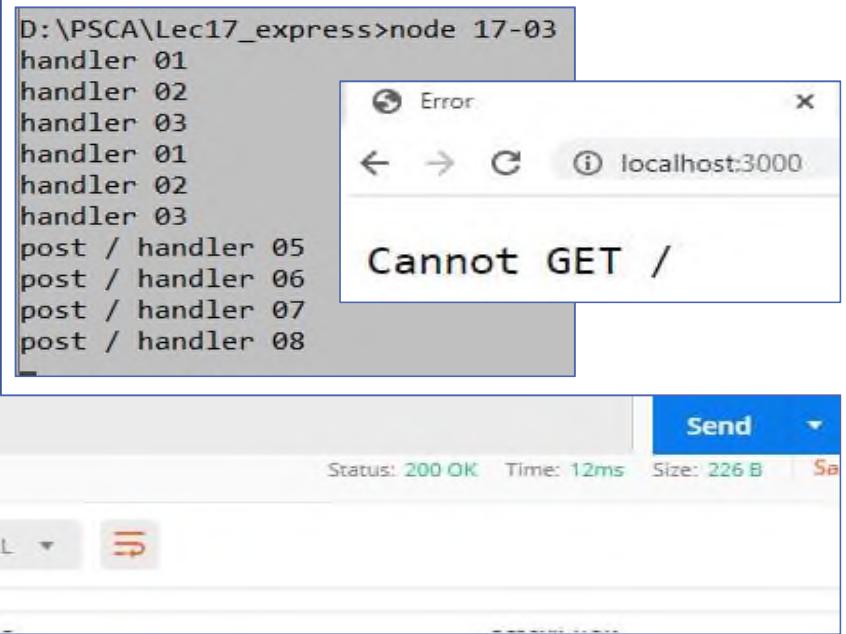
```
const express = require("express"); // npm install express
const app = express();

app.use((req, res, next)=>{console.log('handler 01'); next();}); // middleware
app.use((req, res, next)=>{console.log('handler 02'); next();}); // middleware
app.use((req, res, next)=>{console.log('handler 03'); next();}); // middleware

app.post('/', (req, res, next)=>{                                     // обработка post-запросов
  console.log('post / handler 05');
  next();
})
app.post('/', (req, res, next)=>{                                     // обработка post-запросов
  console.log('post / handler 06');
  next();
})
app.post('/', (req, res, next)=>{                                     // обработка post-запросов
  console.log('post / handler 07');
  next();
})
app.post('/', (req, res)=>{                                         // обработка post-запросов
  console.log('post / handler 08');
  res.send("<h2> handler 07 </h2>");
})

var server = app.listen(3000); //
```

Для одного endpoint'a может быть несколько обработчиков. При наличии нескольких обработчиков важно **вызвать next()** в обработчике, чтобы передать управление следующему обработчику в цепочке, либо отправить клиенту.



```
const express = require('express'); // npm install express
const app = express();

app.use((req, res, next) => { // middleware
  console.log('before-handler 01');
  next();
  console.log('after-handler 01'); // на обратном пути петли
});
app.get('/', (req, res, next) => { // обработка get-запросов
  console.log('get / handler 02', req.query);
  next();
});
app.get('/', (req, res, next) => { // обработка get-запросов
  console.log('get / handler 03');
  if(!req.query.x) res.send('<h2> handler 03 </h2>');
  else if(req.query.x == '0') res.send('<h2> handler 03 x = 0</h2>');
  else if(req.query.x == '1') throw 'error handler 03'; // генерация ошибки
  else next();
});
app.use((req, res, next) => { // middleware
  console.log('handler 04');
  next();
});
app.use((err, req, res, next) => { // middleware: обработка ошибок
  console.log('error-handler', err);
  res.status(500).send('error handler')
});
app.use((req, res, next) => { // middleware: конечная точка
  console.log('notfound-handler');
  res.status(404).send('notfound-handler')
});

var server = app.listen(3000); // = http.Server
```

Обратное движение по конвейеру

Method	Path	Response	Console
GET	/	<pre>before-handler 01 get / handler 02 {} get / handler 03 after-handler 01</pre>	<p>Body ▾ 200 OK</p> <p>Pretty Raw Preview Visualize HTML ▾ ↻</p> <pre>1 <h2> handler 03 </h2></pre>
GET	?x=0	<pre>before-handler 01 get / handler 02 { x: '0' } get / handler 03 after-handler 01</pre>	<p>Body ▾ 200 OK</p> <p>Pretty Raw Preview Visualize HTML ▾ ↻</p> <pre>1 <h2> handler 03 x = 0</h2></pre>
GET	?x=1	<pre>before-handler 01 get / handler 02 { x: '1' } get / handler 03 error-handler error handler 03 after-handler 01</pre>	<p>Body ▾ 500 Internal Server Error</p> <p>Pretty Raw Preview Visualize HTML ▾ ↻</p> <pre>1 error handler</pre>
GET	?x=2	<pre>before-handler 01 get / handler 02 { x: '2' } get / handler 03 handler 04 notfound-handler after-handler 01</pre>	<p>Body ▾ 404 Not Found</p> <p>Pretty Raw Preview Visualize HTML ▾ ↻</p> <pre>1 notfound-handler</pre>
POST	/	<pre>before-handler 01 handler 04 notfound-handler after-handler 01</pre>	<p>Body ▾ 404 Not Found</p> <p>Pretty Raw Preview Visualize HTML ▾ ↻</p> <pre>1 notfound-handler</pre>

Остановка сервера

```
app.use((req, res, next)=>{           //middleware
  console.log('handler 02');
  next();
});

app.use((req, res, next)=>{           //middleware
  console.log('handler 03');
  res.send("<h2> handler 03 </h2>");
  server.close();
});

var server = app.listen(3000);           // = http.Server
```

app.listen() возвращает
экземпляр класса
http.Server.

Для остановки сервера
нужно вызывать close()
для этого экземпляра, а
не для экземпляра
приложения.

Свойства объекта `request`

- `req.app` – содержит ссылку на экземпляр приложения Express;
- `req.baseUrl` – путь URL, по которому был смонтирован экземпляр маршрутизатора;
- `req.method` – содержит строку, соответствующую методу HTTP-запроса;
- `req.ip` – содержит удаленный IP-адрес запроса;
- `req.originalUrl` – похоже на `req.url`; однако он сохраняет исходный URL-адрес запроса, что позволяет вам свободно переписывать `req.url` для внутренней маршрутизации;
- `req.protocol` – содержит строку протокола запроса: `http` или `https`;

Свойства объекта `request`

- `req.path` – содержит path из URL запроса;
- `req.hostname` – содержит имя хоста, полученное из заголовка HTTP;
- `req.params` – объект, содержащий свойства, сопоставленные с именованным route (через :);
- `req.query` – объект, содержащий свойство для каждого параметра строки запроса в маршруте. Если нет строки запроса, это пустой объект, {};
- `req.body` – содержит пары ключ-значение данных, представленных в теле запроса. По умолчанию оно пустое и заполняется при использовании промежуточных обработчиков для анализа тела;
- `req.route` – содержит текущий согласованный маршрут, строку;

Свойства объекта `request`

- `req.cookies` – объект, который содержит cookie, отправленные в запросе (заполняется при использовании промежуточного обработчика);
- `req.signedCookies` – содержит подписанные cookie, отправленные в запросе (заполняется при использовании промежуточного обработчика);
- `req.secure` – логическое свойство, которое имеет значение `true`, если установлено соединение TLS;
- `req.stale` – указывает, является ли ответ в кэше клиента «устаревшим»;
- `req.fresh` – указывает, является ли ответ в кэше клиента «свежим»;
- `req.subdomains` – массив поддоменов в доменном имени запроса;
- `req.xhr` – логическое свойство, которое имеет значение `true`, если запрос был выполнен клиентской библиотекой, такой как jQuery.

Метод req.accepts(types)

Проверяет, являются ли указанные типы контента **приемлемыми для клиента**, на основании заголовка запроса Accept.

```
app.post('/xxx', (req, res, next)=>{                                // обработка post-запросов

    console.log('json? = ', req.accepts('json'));
    console.log('html? = ', req.accepts('html'));
    console.log('[html, json]? = ', req.accepts(['json', 'html']));
    res.send('<h1>post</h1>');

})
```

Метод возвращает **наилучшее совпадение или**, если ни один из указанных типов содержимого не является приемлемым, возвращает **false**.

Метод req.accepts()

POST http://localhost:3000/xxx

Headers (9)

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Accept	application/json,text/html	

```
D:\PSCA\Lec17_express>node 17-05
Start server, port: 3000
json? = json
html? = html
[html, json]? = json
```

POST http://localhost:3000/xxx

Headers (9)

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Accept	image/png,text/html	

```
D:\PSCA\Lec17_express>node 17-05
Start server, port: 3000
json? = false
html? = html
png? = png
[html, json, png]? = png
```

Свойства объекта response

- **res.app** – содержит ссылку на экземпляр приложения Express, `res.app` идентичен свойству `req.app`;
- **res.headersSent** – логическое свойство, указывающее, отправляло ли приложение заголовки HTTP для ответа;
- **res.locals** – объект, который содержит локальные переменные ответа, ограниченные областью запроса.

Метод res.send([body])

```
app.get('/sendHtml', (req, res, next) => {
  res.send('<p>some html</p>')
})
```

Body ▾

Pretty Raw Preview Visualize

```
1  <p>some html</p>
```

GET http://localhost:3000/sendHtml

Params Auth Headers (7) Body Pre-req. Tests Settings

Headers ▾

Key	Value
X-Powered-By	① Express
Content-Type	① text/html; charset=utf-8
Content-Length	① 16

Отправляет HTTP-ответ и автоматически назначает заголовки **Content-Length** и **Content-Type** (если они не определены ранее).

Метод res.format(object)

```
app.post('/format', (req, res, next)=>{
  res.format(
    {
      'text/html'      : ()=>{res.send('<h2>format: text/html</h2>')},
      'text/plain'     : ()=>{res.send('format: text/plain')},
      'application/json': ()=>{res.send('{"format": "application/json"}')},
      'default'        : ()=>{res.status(406).send('Not Acceptable')}
    }
  )
})
```

Выполняет согласование содержимого **на основе заголовка запроса Accept**.
В результате **устанавливается** заголовок ответа **Content-Type**.

Метод res.format()

The image displays three separate Postman requests, each showing the configuration and response for the `res.format()` method.

Request 1 (Top Left): POST `http://localhost:3000/format`. Headers: `Accept: text/html`. Body: `<h2>format: text/html</h2>`. Response: Status: 200 OK, Time: 25 ms, Size: 238 B.

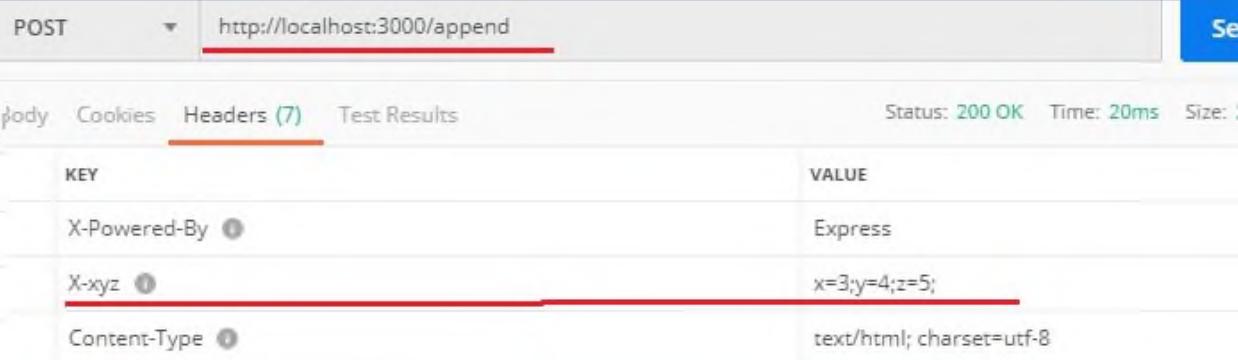
Request 2 (Top Right): POST `http://localhost:3000/format`. Headers: `Accept: text/plain`. Body: `format: text/plain`. Response: Status: 200 OK, Time: 25 ms, Size: 238 B.

Request 3 (Bottom Left): POST `http://localhost:3000/format`. Headers: `Accept: image/png`. Body: `Not Acceptable`. Response: Status: 406 Not Acceptable, Time: 24 ms, Size: 241 B.

Метод res.append(field [, value])

```
app.post('/append', (req, res, next)=>{
  res.append('X-huz', 'x=3;y=4;z=5'); // добавить заголовок
  res.send('<h1>post/properties</h1>');
})
```

Добавляет указанное значение в указанный заголовок ответа.



KEY	VALUE
X-Powered-By	Express
X-huz	x=3;y=4;z=5;
Content-Type	text/html; charset=utf-8

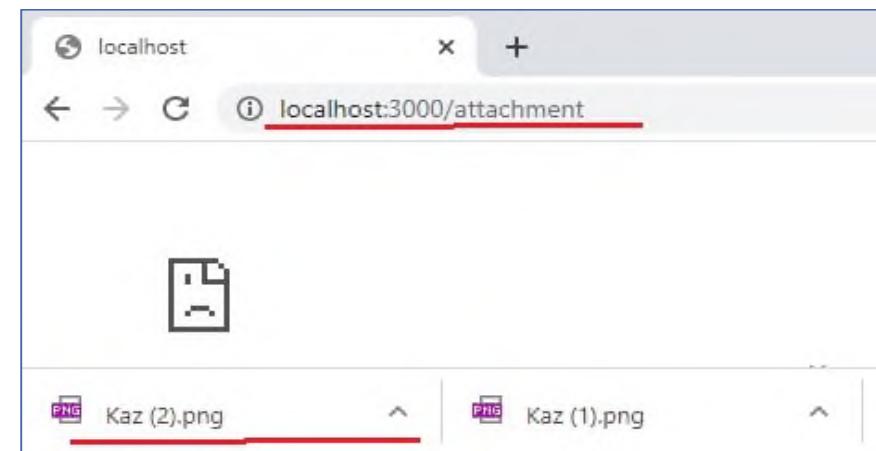
Метод res.attachment([filename])

```
app.get('/attachment', (req, res, next)=>[]  
  res.attachment('Kaz.png'); // добавить заголовок  
  let rs = fs.ReadStream('./Kaz.png');  
  rs.pipe(res);  
})
```

Устанавливает в заголовок ответа **Content-Disposition** значение «attachment». Если задано имя файла, тогда он устанавливает **Content-Type** на основе расширения файла и добавляет название в Content-Disposition.

GET <http://localhost:3000/attachment>

Body	Cookies	Headers (6)	Test Results
			Status: 200 OK Time: 17ms Size
KEY		VALUE	
X-Powered-By	①	Express	
Content-Type	①	image/png	
Content-Disposition	①	attachment; filename="Kaz.png"	
Date	①	Sat, 11 Jan 2020 22:52:30 GMT	
Connection	①	keep-alive	



Заголовок **Content-Disposition** является индикатором того, что ожидаемый контент ответа будет отображаться в браузере, как веб-страница или часть веб-страницы, или же как вложение, которое затем может быть скачано и сохранено локально.

`Content-Disposition: inline`

`Content-Disposition: attachment`

`Content-Disposition: attachment; filename="filename.jpg"`

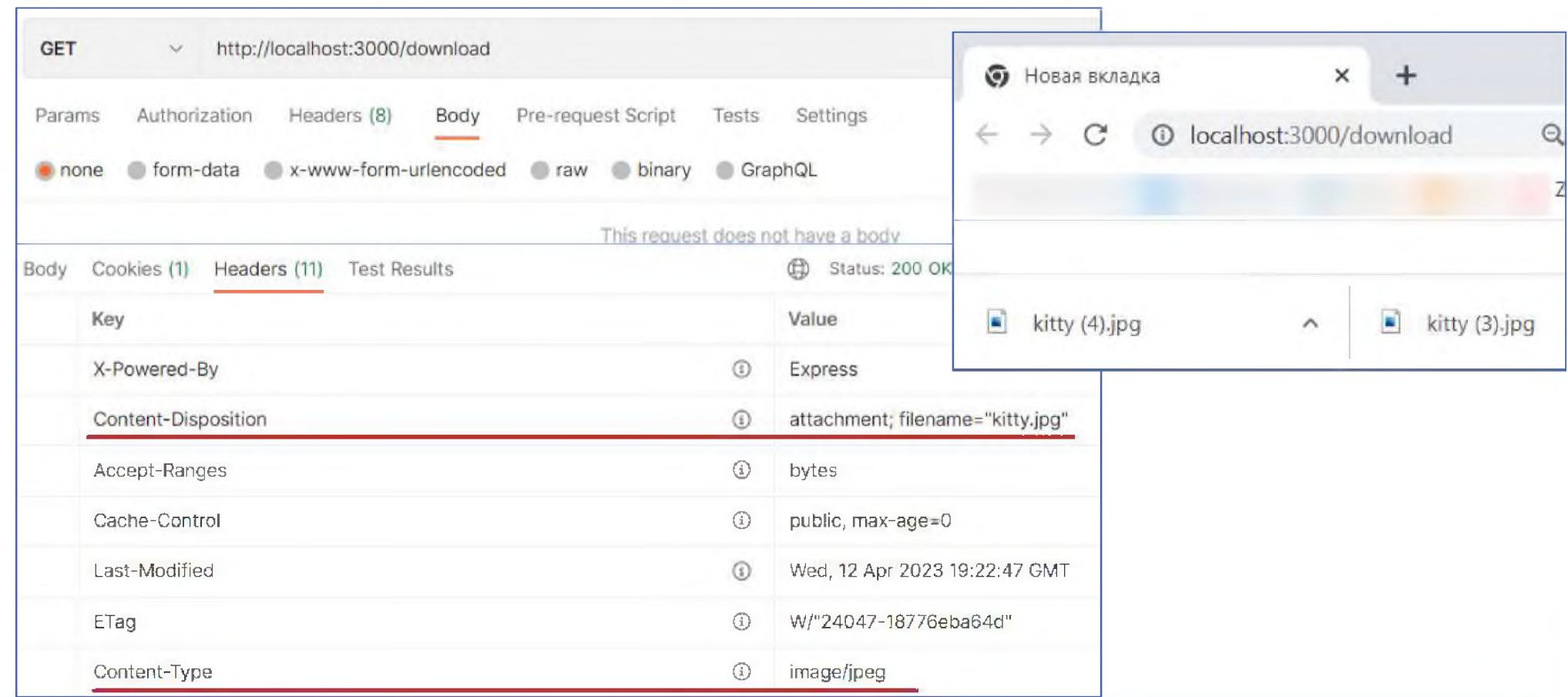
Метод res.download(path [, filename] [, options] [, fn])

```
app.get('/download', (req, res, next) => {
  res.download('./upload/kitty.jpg');
})
```

Устанавливает заголовки ответа **Content-Type** (на основе расширения файла) и **Content-Disposition** (значение attachment).

Этот метод использует **res.sendFile()** для передачи файла.

Метод res.download()



GET http://localhost:3000/download

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

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies (1) Headers (11) Test Results

Key Value

Key	Value
X-Powered-By	Express
Content-Disposition	attachment; filename="kitty.jpg"
Accept-Ranges	bytes
Cache-Control	public, max-age=0
Last-Modified	Wed, 12 Apr 2023 19:22:47 GMT
ETag	W/"24047-18776eba64d"
Content-Type	image/jpeg

Новая вкладка x +

localhost:3000/download

kitty (4).jpg

kitty (3).jpg

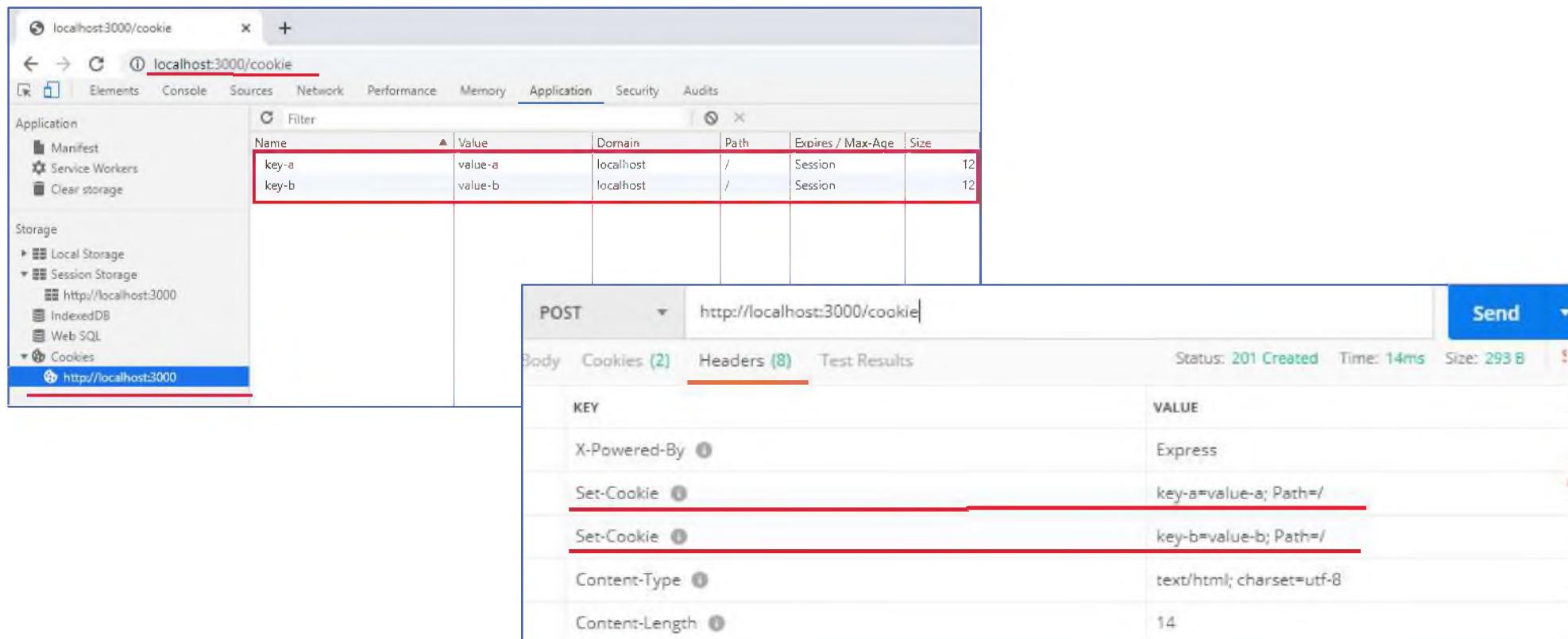
Метод res.cookie(name, value [, options])

```
app.post('/cookie', (req, res, next)=>{  
  res  
  .status(201)  
  .cookie('key-a', 'value-a')  
  .cookie('key-b', 'value-b')  
  .send('<h1>cookie<h2>');  
})
```

Устанавливает имя куки и значение. Параметр value может быть строкой или объектом, преобразованным в JSON.

Устанавливает заголовок ответа **Set-Cookie**, который используется для отправки cookies с сервера на агент пользователя.

Метод res.cookie()



localhost:3000/cookie

localhost:3000/cookie

Application

- Manifest
- Service Workers
- Clear storage

Storage

- Local Storage
- Session Storage
 - http://localhost:3000
- IndexedDB
- Web SQL

Cookies

http://localhost:3000

Name	Value	Domain	Path	Expires / Max-Age	Size
key-a	value-a	localhost	/	Session	12
key-b	value-b	localhost	/	Session	12

POST http://localhost:3000/cookie

Body Cookies (2) Headers (8) Test Results

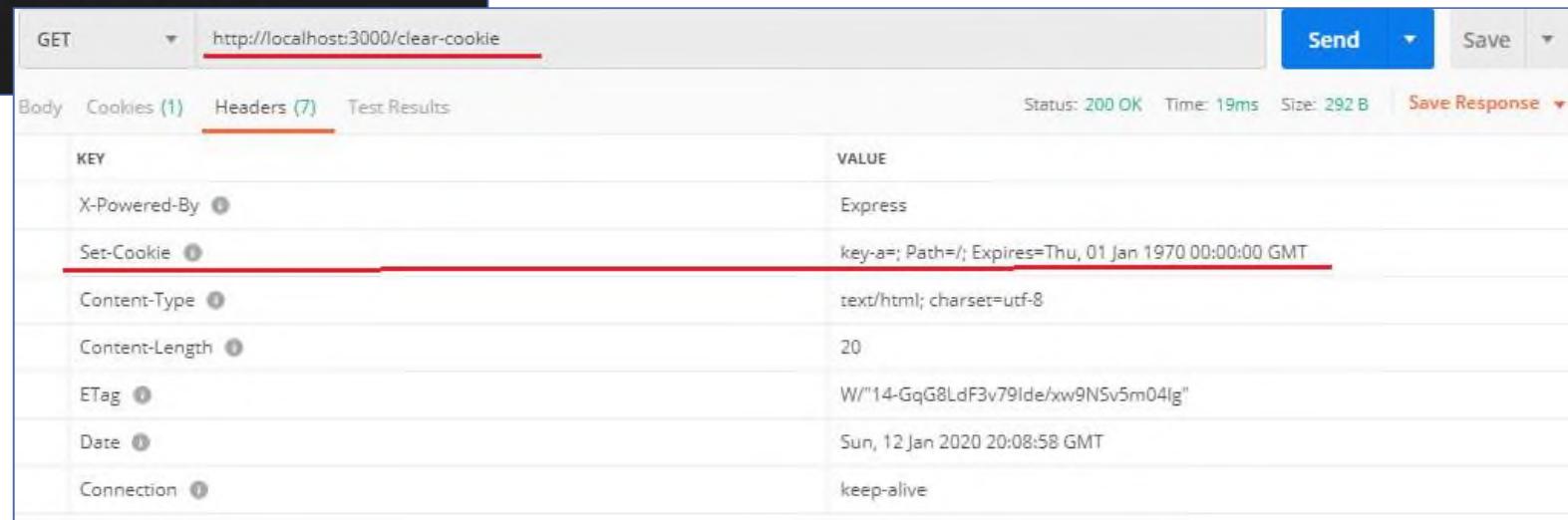
Status: 201 Created Time: 14ms Size: 293 B

KEY	VALUE
X-Powered-By	Express
Set-Cookie	key-a=value-a; Path=/
Set-Cookie	key-b=value-b; Path=/
Content-Type	text/html; charset=utf-8
Content-Length	14

Метод res.clearCookie(name [, options])

```
app.get('/clear-cookie', (req, res, next)=>{
  res
    .status(200)
    .clearCookie('key-a')
    .send('<h1>clear-cookie<h2>');
})
```

Чтобы удалить cookie, сервер возвращает заголовок ответа **Set-Cookie** с датой истечения срока действия в прошлом.

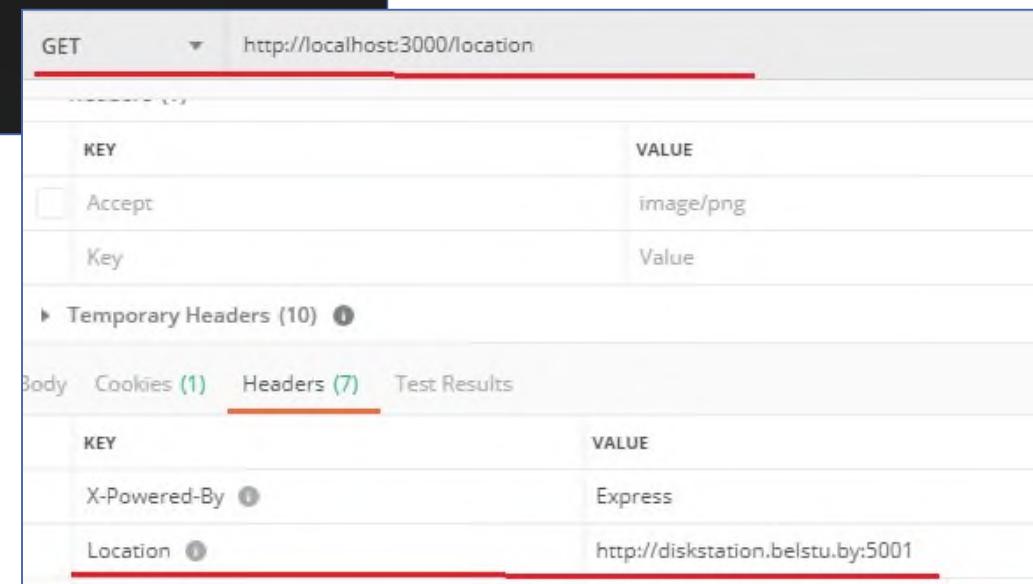


KEY	VALUE
X-Powered-By	Express
Set-Cookie	key-a=; Path=/; Expires=Thu, 01 Jan 1970 00:00:00 GMT
Content-Type	text/html; charset=utf-8
Content-Length	20
ETag	W/\"14-GqG8LdF3v79lde/xw9N5v5m04lg\"
Date	Sun, 12 Jan 2020 20:08:58 GMT
Connection	keep-alive

Метод res.location(path)

```
app.get('/location', (req, res, next)=>{
  res.location('http://diskstation.belstu.by:5001');
  res.send('location');
})
```

Устанавливает заголовок ответа **Location** с указанным параметром path.



GET http://localhost:3000/location

KEY	VALUE
Accept	image/png
Key	Value

Temporary Headers (10)

KEY	VALUE
X-Powered-By	Express
Location	http://diskstation.belstu.by:5001

Метод res.redirect([status,] path)

```
app.get('/redirect', (req, res, next)=>{
  res.redirect('redirect/xxx');
})
app.get('/redirect/xxx', (req, res, next)=>{
  res.send('redirect/xxx');
})
```

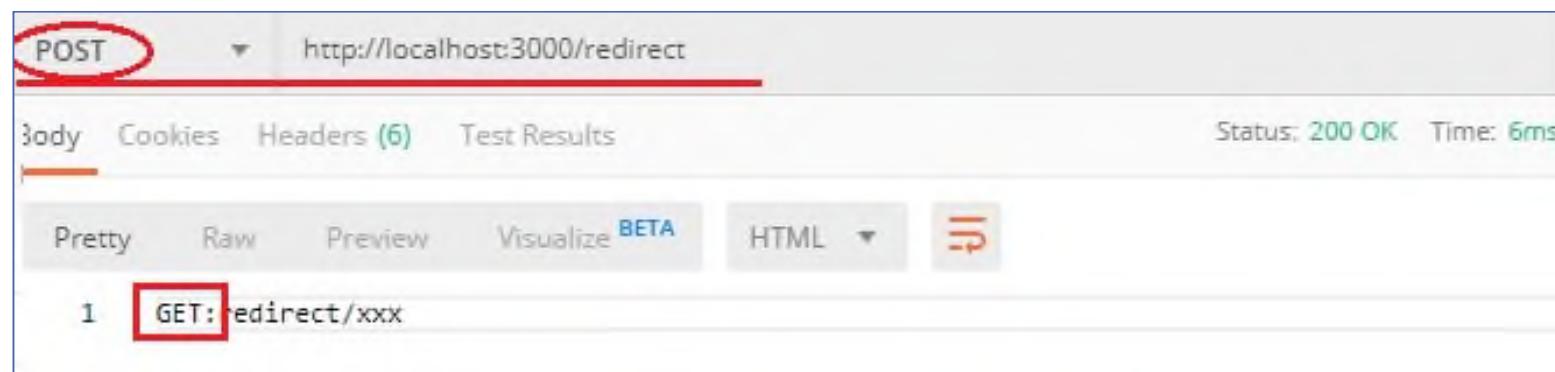
Переадресует на URL-адрес, полученный по указанному path (полный или относительный). Если статус код не указан, то по умолчанию отправляется 302 «Found».



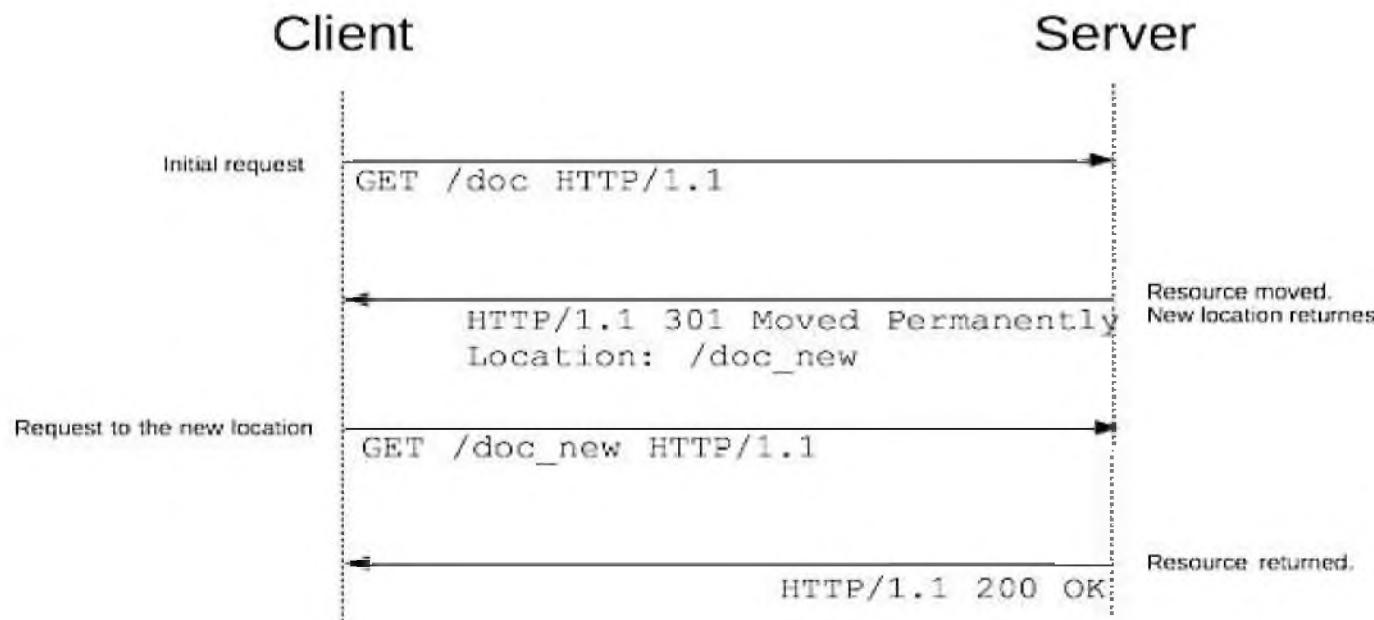
Метод res.redirect()

```
app.get('/redirect/xxx', (req, res, next)=>{
  res.send('GET:redirect/xxx');
})
app.post('/redirect', (req, res, next)=>{
  res.redirect('redirect/xxx');
})
app.post('/redirect/xxx', (req, res, next)=>{
  res.send('POST:redirect/xxx');
})
```

Если переадресовать POST-запрос на новый адрес, то на новый адрес будет послан GET-запрос из-за того, что браузер по историческим причинам посыпает GET-запрос.



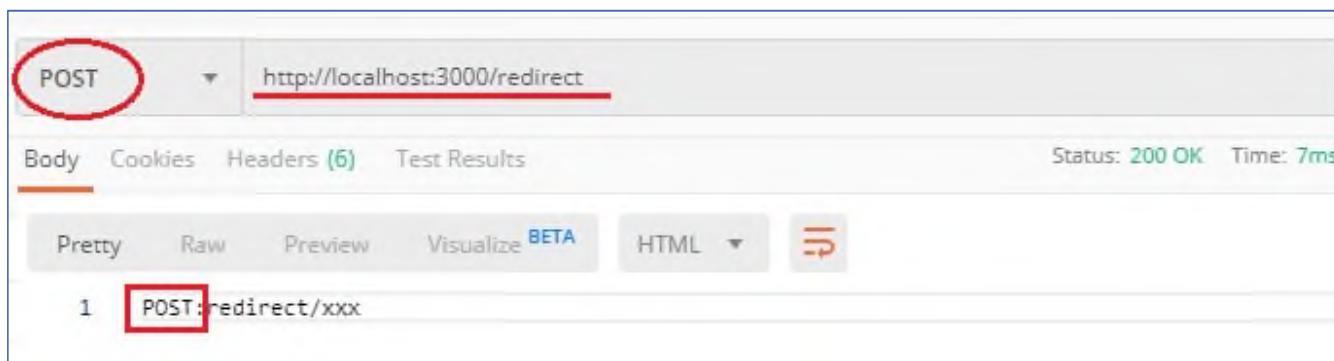
Переадресация



Метод res.redirect()

```
app.get('/redirect/xxx', (req, res, next)=>{
  res.send('GET:redirect/xxx');
})
app.post('/redirect', (req, res, next)=>{
  res.redirect(308, 'redirect/xxx');
})
app.post('/redirect/xxx', (req, res, next)=>{
  res.send('POST:redirect/xxx');
})
```

Если нам все же надо переадресовать именно **POST-запрос**, то в таком случае необходимо использовать **307 или 308 статус код**, т.к. они не позволяют изменить метод запроса с POST на GET.



Moved Permanently
301 => 308
Moved Temporarily
302 => 307

Метод res.sendFile(path [, options] [, fn])

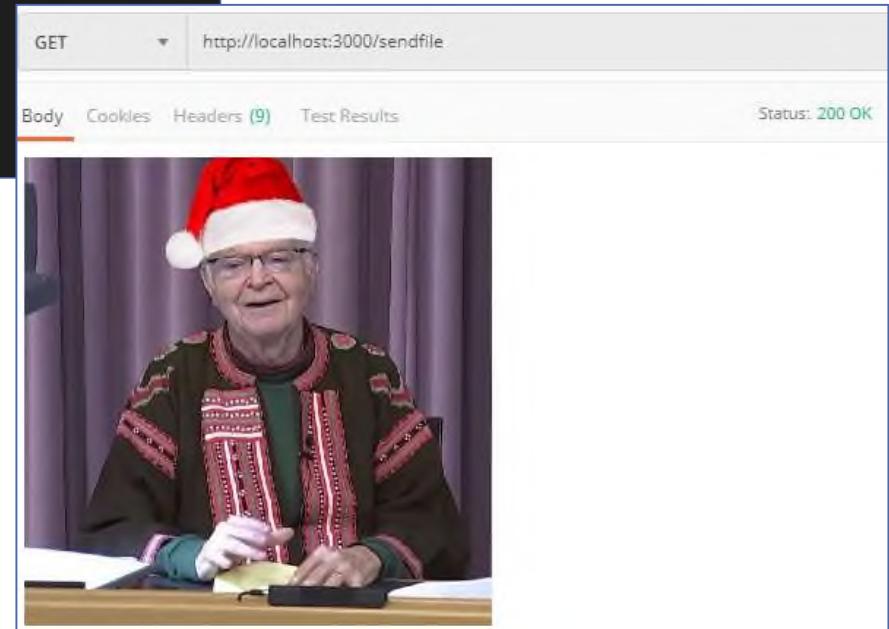
```
const path = require('path');

app.get('/sendfile', (req, res, next)=>{
  res.sendFile('DKnuth.jpg', {root: path.join(__dirname, 'Files')}, (err)=>{
    if(err) console.log(err);
    else    console.log('send files - OK');
  });
})
```

Отправляет указанный файл.

Устанавливает заголовок Content-Type на основе расширения файла.

Если в объекте параметров не задан параметр root, путь должен быть абсолютным путем к файлу.

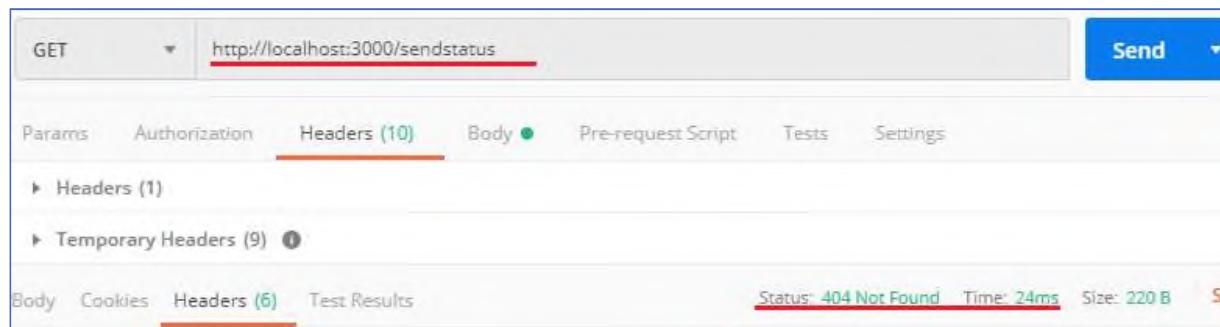


Метод res.sendFile()

- **options.maxAge** – устанавливает свойство max-age заголовка Cache-Control в миллисекундах
- **options.root** – корневой каталог для относительных имен файлов.
- **options.lastModified** – булевое свойство, устанавливает в заголовке Last-Modified дату последнего изменения файла в ОС. По умолчанию true.
- **options.headers** – объект, содержащий заголовки HTTP для использования с файлом.
- **options.cacheControl** – булевое свойство, включает или отключает настройку заголовка ответа Cache-Control. По умолчанию true.
- **options.immutable** – булевое свойство, включает или отключает директиву immutable в заголовке ответа Cache-Control. Директива immutable не позволяет клиентам делать условные запросы в течение срока действия опции maxAge, чтобы проверить, изменился ли файл. По умолчанию false.

Метод res.sendStatus(statusCode)

```
app.get('/sendstatus', (req, res, next)=>{
  res.sendStatus(404); //res.status(404).send('Not Found')
})
```



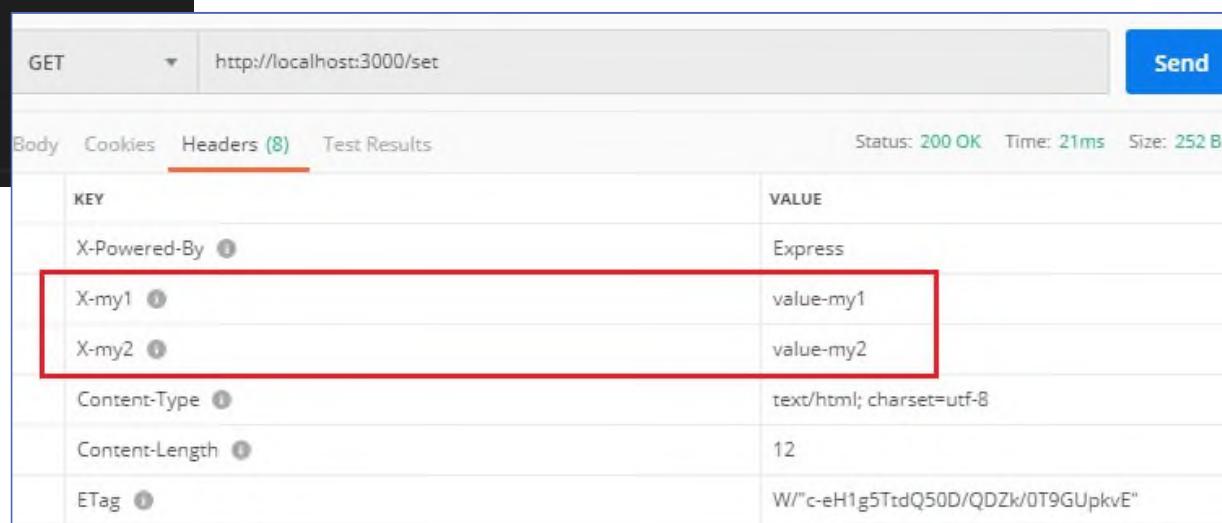
Устанавливает код ответа и отправляет его пояснение в качестве тела ответа (если указан неподдерживаемый код, то строковая версия кода).

Метод res.set(field [, value])

```
app.get('/set', (req, res, next) =>{  
  res.set(  
    {  
      'X-my1': 'value-my1',  
      'X-my2': 'value-my2'  
    }  
  );  
  
  res.send('--- SET ----');  
})
```

Устанавливает value в **заголовок** ответа field.

Чтобы установить несколько полей одновременно, передайте объект в качестве параметра.

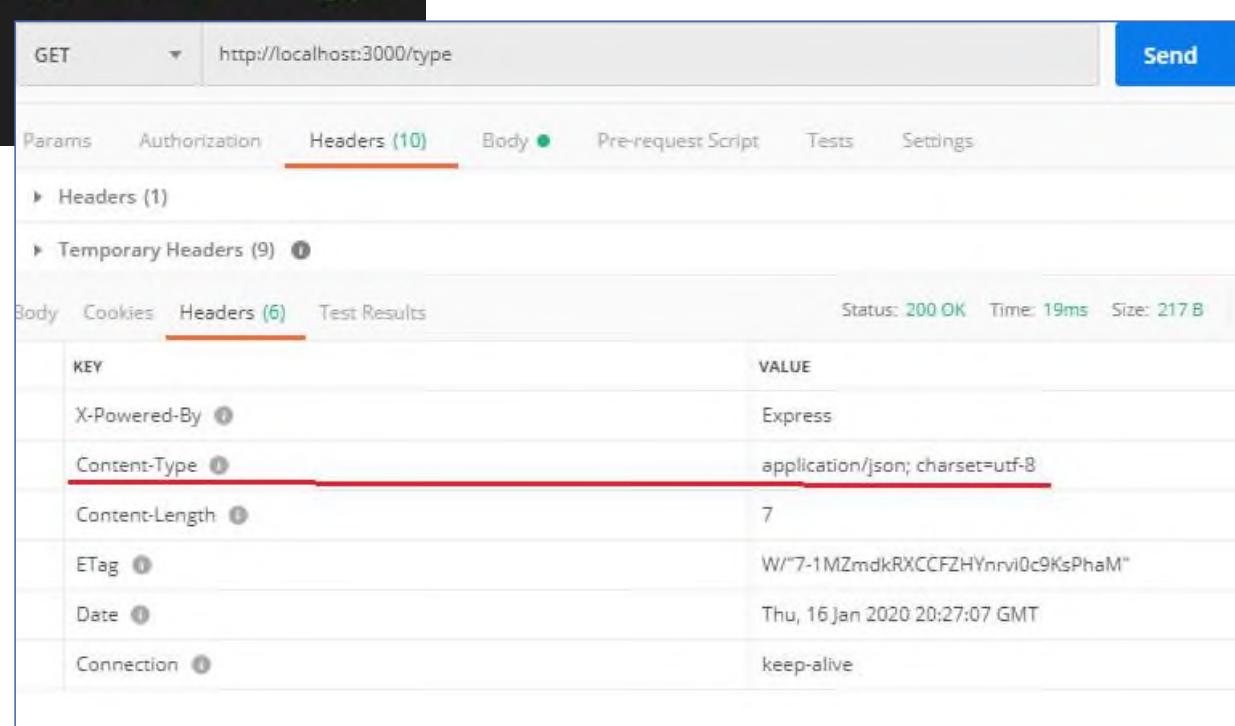


KEY	VALUE
X-Powered-By	Express
X-my1	value-my1
X-my2	value-my2
Content-Type	text/html; charset=utf-8
Content-Length	12
ETag	W/"c-eH1g5TtdQ50D/QDZk/0T9GUpkvE"

Метод res.type(type)

```
app.get('/type', (req, res, next) => {
  res.type('json');           // content-type
  res.send({ x: 3 });
})
```

Устанавливает для заголовка **Content-Type** тип MIME, определенный указанным type.



Headers (10)

- Headers (1)
- Temporary Headers (9)

Body (6) Headers (6) Test Results

KEY	VALUE
X-Powered-By	Express
Content-Type	application/json; charset=utf-8
Content-Length	7
ETag	W/"7-1MZmdkRXCCFZHYNrvi0c9KsPhaM"
Date	Thu, 16 Jan 2020 20:27:07 GMT
Connection	keep-alive

Метод res.get(field)

```
app.get('/getType', (req, res, next) => {
  res.type('html');
  console.log(res.get('Content-Type'));

  res.type('image');
  console.log(res.get('Content-Type'));

  res.type('json');
  console.log(res.get('Content-Type'));

  res.send({ x: 3 });
})
```

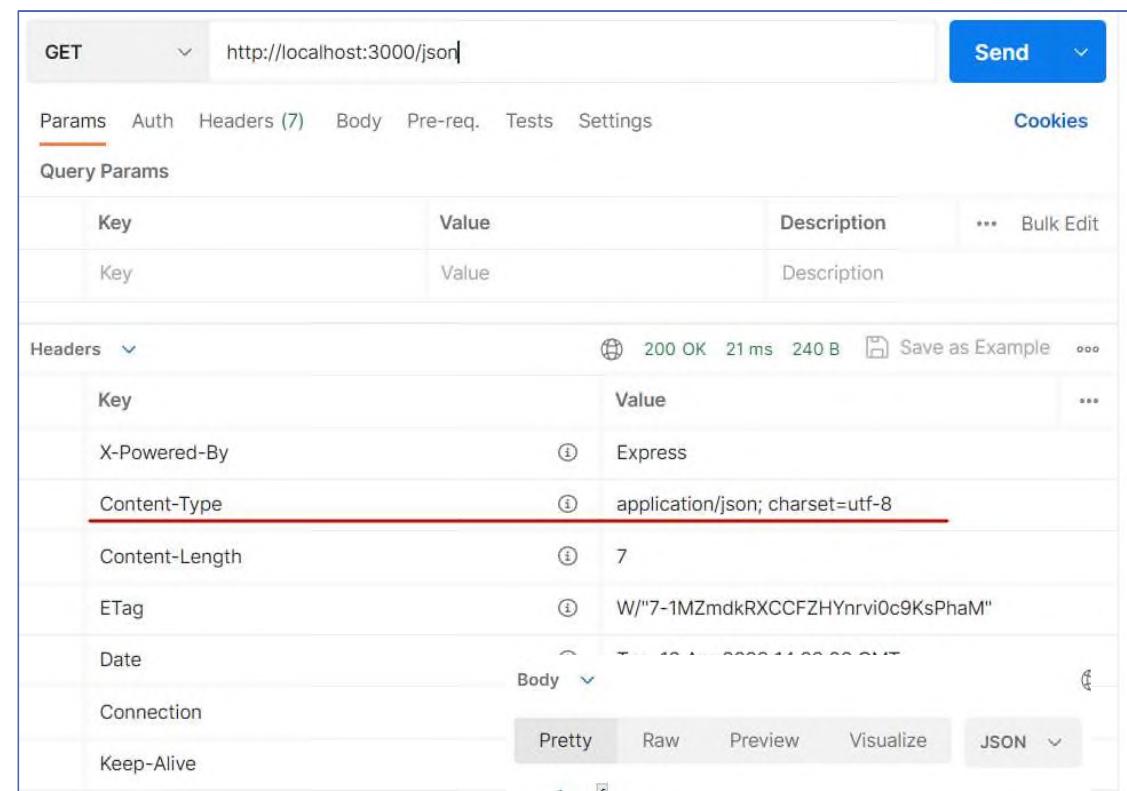
```
PS D:\NodeJS\samples\cwp_17> node 17-13
Server running at port 3000
text/html; charset=utf-8
application/octet-stream
application/json; charset=utf-8
```

Возвращает указанный заголовок ответа.

Метод res.json(body)

```
app.get('/json', (req, res, next) => {
  res.json({ x: 3 });
})
```

Отправляет **ответ** в **формате JSON** и устанавливает правильный тип содержимого в **заголовок Content-Type**.



GET http://localhost:3000/json

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Headers

Key	Value
X-Powered-By	Express
Content-Type	application/json; charset=utf-8
Content-Length	7
ETag	W/"7-1MZmdkRXCCFZHnrvioC9KsPhaM"
Date	Mon, 11 Dec 2017 10:20:00 GMT
Connection	Keep-Alive

Body

Pretty Raw Preview Visualize JSON

```
1 {  
2   "x": 3  
3 }
```

Виды параметров запроса

- **Query-параметры** – дополнительная информация, которую можно добавить в URL-адрес. Состоит из двух обязательных элементов: самого параметра и его значения, разделенных знаком равенства (=). Параметры **указываются в конце URL**, отделяясь от основного адреса знаком вопроса (?). Можно указать более одного параметра, для этого каждый параметр со значениями отделяется от следующего знаком амперсанда (&).
- **Path-параметры** - дополнительная информация, которую можно задать в URL-адресе. Они **являются частью маршрута URL**.

/car/make/12/model?color=mintgreen&doors=4

Обработка query-параметров

```
const express = require('express');
const app = express();

app.get('/sum', (req, res) => {
  console.log(req.query);
  let x = +req.query.x;
  let y = +req.query.y;

  res.send(`x + y = ${x + y}`);
})

app.listen(3000, () => console.log(`Server running at port 3000`));
```

Query-параметры запроса могут быть получены **из свойства `query` в объекте `request`** в обработчике на необходимый endpoint. Оно имеет форму объекта, в котором можно напрямую получить доступ к интересующим параметрам запроса.

Обработка query-параметров

GET http://localhost:3000/sum?x=3&y=10

Params Auth Headers (7) Body Pre-req. Tests Settings

Query Params

Key	Value
<input checked="" type="checkbox"/> x	3
<input checked="" type="checkbox"/> y	10

Body 200 OK

Pretty Raw Preview Visualize HTML

```
1 x + y = 13
```

GET http://localhost:3000/sum?x=3&y=10&z=12

Params Auth Headers (7) Body Pre-req. Tests Settings

<input checked="" type="checkbox"/> x	3
<input checked="" type="checkbox"/> y	10
<input checked="" type="checkbox"/> z	12

Body 200 OK

Pretty Raw Preview Visualize HTML

```
1 x + y = 13
```

GET http://localhost:3000/sum?x=3&z=12

Params Auth Headers (7) Body Pre-req. Tests Settings

<input checked="" type="checkbox"/> x	3
<input type="checkbox"/> y	10
<input checked="" type="checkbox"/> z	12

Body 200 OK

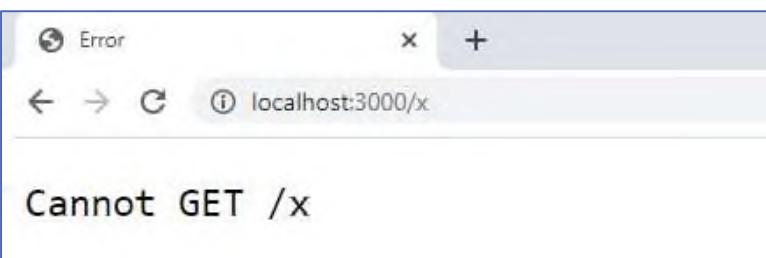
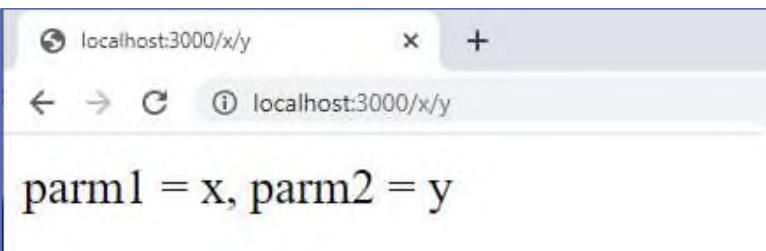
Pretty Raw Preview Visualize HTML

```
1 x + y = NaN
```

```
PS D:\NodeJS\samples\cwp_17> node 17-14
Server running at port 3000
{ x: '3', y: '10' }
{ x: '3', y: '10', z: '12' }
{ x: '3', z: '12' }
```

Обработка path-параметров

```
app.get('/:parm1/:parm2', (req, res) => {
  res.send(`parm1 = ${req.params.parm1}, parm2 = ${req.params.parm2}`);
})
```



При определении маршрутов с динамически изменяющейся частью, необходимо **указать имя переменной-заполнителя** (начинающееся с **двоеточия**), и Express поместит эту переменную из входящего запроса в объект **req.params**.

Обработка тела (JSON)

Middleware bodyParser **анализирует тело входящих запросов** перед обработчиками и **записывает** результат разбора **в свойство req.body**.

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

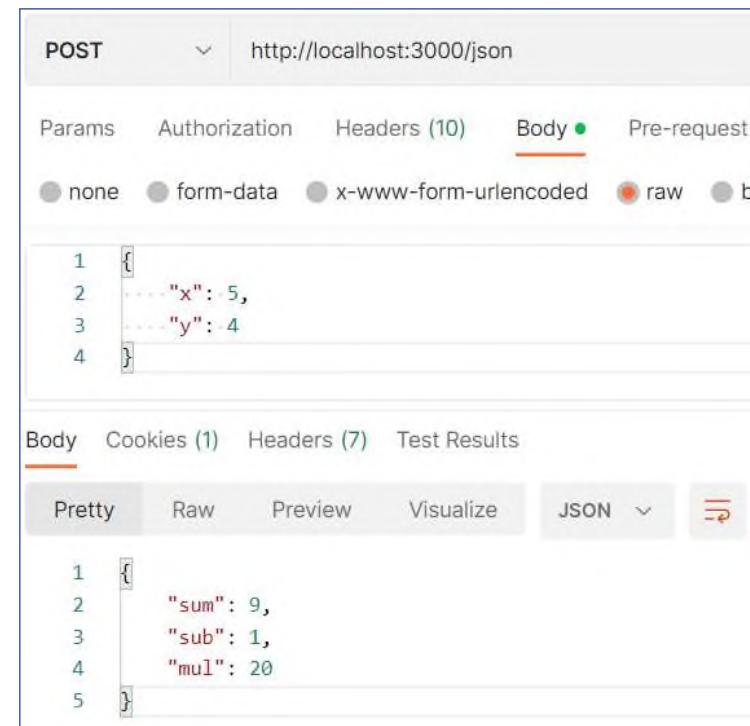
app.use(express.json());
// == app.use(bodyParser.json());

app.post('/json', (req, res) => {
  console.log(req.body);
  let x = +req.body.x;
  let y = +req.body.y;

  res.json({ sum: x + y, sub: x - y, mul: x * y });
})

app.listen(3000, () => console.log(`Server running at port 3000`));
```

В случае с .json() преобразует входящую строку из тела запроса в JS-объект. Обрабатывает только те запросы, в которых **Content-Type: application/json**.



```
PS D:\NodeJS\samples\cwp_17> node .\17-08.js
{ x: 5, y: 4 }
```

Обработка тела с формы (application/x-www-form-urlencoded)

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

app.use(express.urlencoded({ extended: false })); // querystring - false, qs - true
// == app.use(bodyParser.urlencoded({ extended: false }));

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/form09.html');
})

app.post('/', (req, res) => {
  console.log(req.body);
  res.send(`x: ${req.body.x} y: ${req.body.y} s: ${req.body.s}`);
})

app.listen(3000, () => console.log(`Server running at port 3000`));
```

.urlencoded() анализирует только те запросы, в которых заголовок **Content-Type: application/x-www-form-urlencoded**.

Свойство req.body будет содержать пары ключ-значение, где значение может быть строкой или массивом (если extended – false) или любым типом (когда extended – true).

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Form</title>
</head>

<body>
  <form method="post" action="/">
    X: <input type="number" name="x">
    <br>
    Y: <input type="number" name="y">
    <br>
    S: <input type="text" name="s">
    <br>
    <input type="submit" value="Submit">
  </form>
</body>

</html>
```

Обработка тела с формы (application/x-www-form-urlencoded)

Данные отправляются с формы по умолчанию в формате ключ-значение.
Ключ берется из атрибута **name** элемента формы, а значение из атрибута **value**.

The image shows two browser windows. The left window displays a form with three text input fields: X: 23, Y: 45, and S: qwerty, with a 'Submit' button. A tooltip over the 'Submit' button shows the form data: 'x=23&y=45&s=qwerty'. The right window shows the browser's address bar and the resulting URL: 'localhost:3000?x=23&y=45&s=qwerty'.

```
PS D:\NodeJS\samples\cwp_17> node .\17-09.js
Server running at port 3000
[Object: null prototype] { x: '3', y: '5', s: 'ddd' }
```

```
const express = require('express');
const app = express();

const multer = require('multer');

const storageConfig = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'upload');
  },
  filename: (req, file, cb) => {
    cb(null, `${req.body.name}.${file.originalname.split('.')[1]}`);
  }
});
let upload = multer({ storage: storageConfig });

app.get('/upload', function (req, res, next) {
  res.sendFile(__dirname + '/form10.html')
});

app.post('/upload', upload.single('file'), async function (req, res, next) {
  if (!req.file)
    res.send('Ошибка при загрузке файла');
  else {
    res.send('Файл загружен');
  }
});

app.listen(3000, () => console.log(`Server listening on port 3000`));
```

Обработка тела с формы (multipart/form-data)

Multer – это сторонний middleware для обработки запросов с Content-Type: multipart/form-data, который в основном используется для загрузки файлов.

Multer добавляет объект body (текстовые поля формы) и объект file (или files) внутри объекта request.

Multer поставляется с двумя движками: DiskStorage и MemoryStorage, другие движки можно найти у сторонних разработчиков.

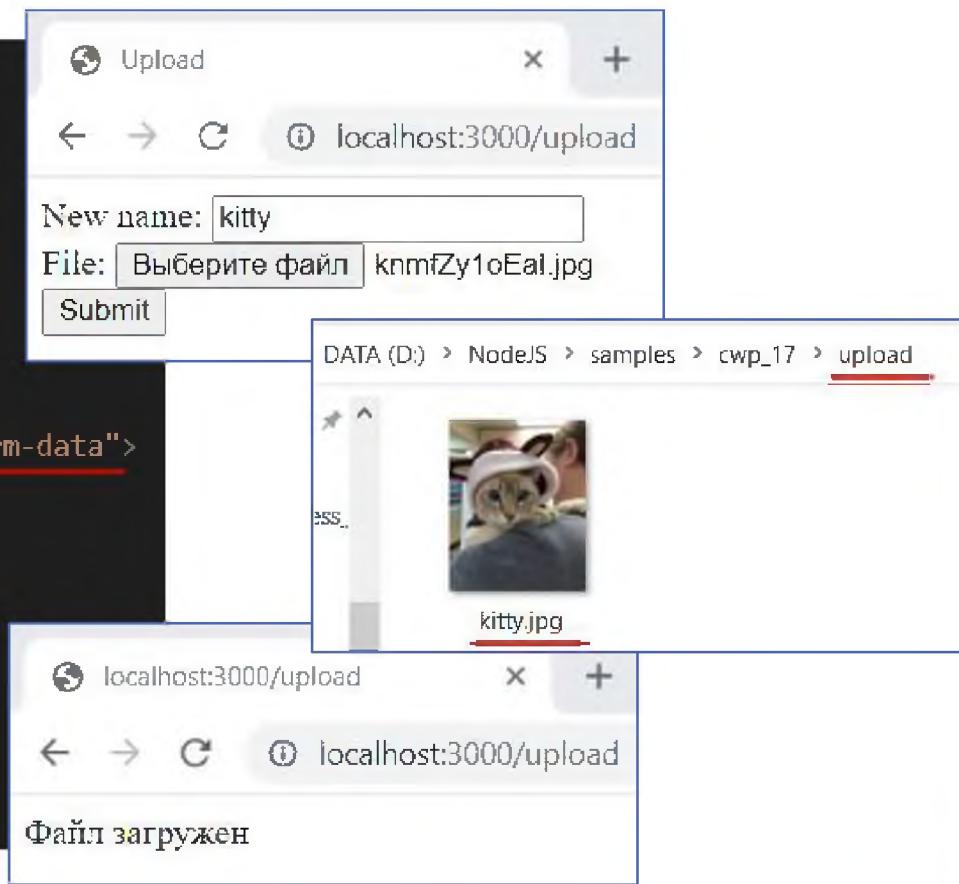
Обработка тела с формы (multipart/form-data)

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Upload</title>
</head>

<body>
  <form method="post" action="/upload" enctype="multipart/form-data">
    New name: <input type="text" name="name">
    <br>
    File: <input type="file" name="file">
    <br>
    <input type="submit" value="Submit">
  </form>
</body>

</html>
```



express.Router =

это изолированный экземпляр промежуточного ПО (middleware), позволяющий определить **дочерние подмаршруты** со своими обработчиками относительно некоторого главного маршрута. И тем самым связать подобный функционал в одно целое и упростить управление им.

express.Router

JS 17-07-router123.js > ...

```
1 const express = require('express');
2 const router = express.Router();
3
4 router.get('/1', (req, res) => { res.send('/1'); })
5 router.get('/2', (req, res) => { res.send('/2'); })
6 router.post('/3', (req, res) => { res.send('/3'); })
7
8 module.exports = router
```

JS 17-07-routerXYZ.js > ...

```
1 const express = require('express');
2 const router = express.Router();
3
4 router.get('/a', (req, res) => { res.send('/a'); })
5 router.get('/b', (req, res) => { res.send('/b'); })
6 router.put('/b', (req, res) => { res.send('/b'); })
7
8 module.exports = router
```

```
const express = require('express');
let app = express();

const router123 = require('./17-07-router123');
const routerXYZ = require('./17-07-routerXYZ');

app.use('/x', router123);
app.use('/y', routerXYZ);

app.listen(3000);
```

GET http://localhost:3000/x

Params Authorization

Body Cookies (1)

GET http://localhost:3000/x/1

Params Authorization

PUT http://localhost:3000/y/b

Params Authorization

GET http://localhost:3000/y/b

Params Authorization Headers (8)

Body Cookies (1) Headers (7) Test Results

Pretty Raw Preview Visualize

Маршрутизатор ведет себя как middleware, поэтому можно использовать его в качестве аргумента для **app.use()** или в качестве аргумента для метода **use()** **другого маршрутизатора.**

express.Router

```
const express = require('express');
const router = express.Router();

// router.get('/a', (req, res) => { res.send('/a'); })
// router.get('/b', (req, res) => { res.send('/b'); })
// router.put('/b', (req, res) => { res.send('/b'); })

router.get('/a', (req, res) => { res.send('/a'); })
router.route('/b')
  .get((req, res) => { res.send('/b'); })
  .put((req, res) => { res.send('/b'); })

module.exports = router
```

Можно использовать `router.route()`, чтобы избежать дублирования имен маршрутов.

Раздача статики

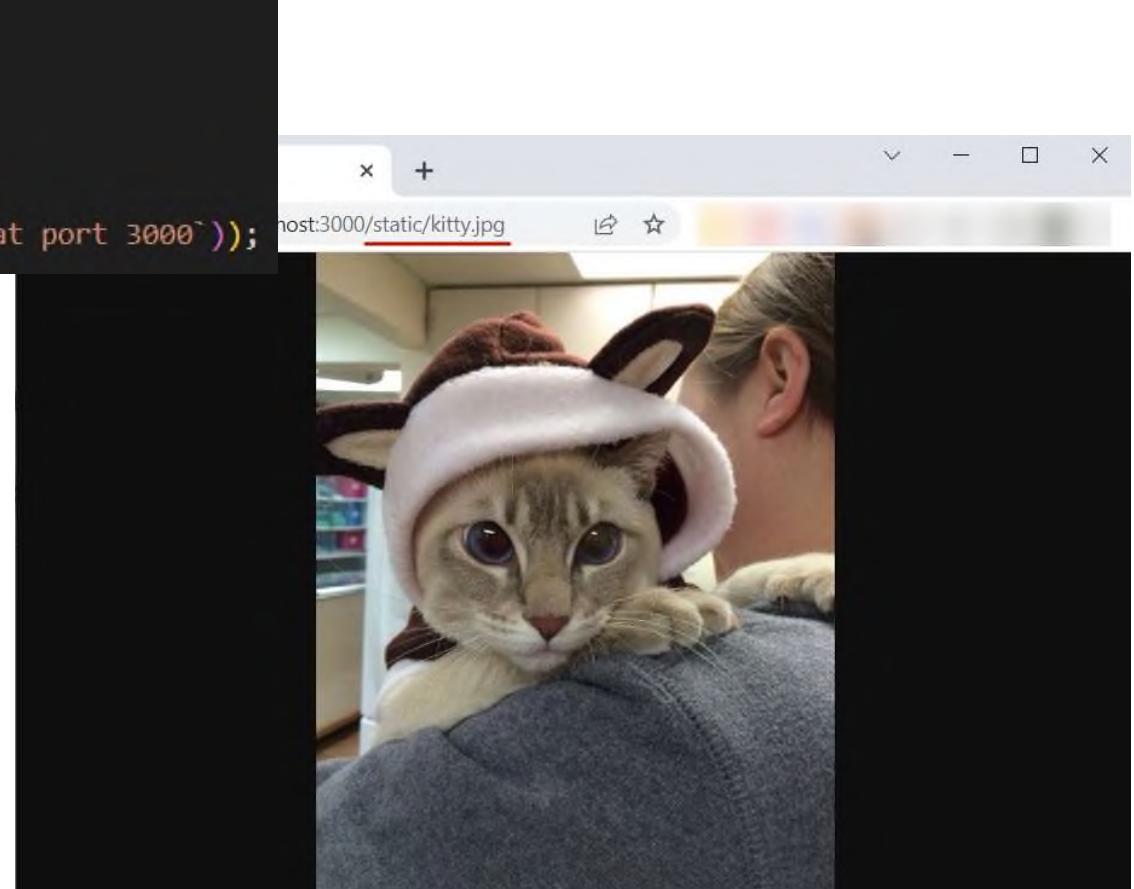
```
const express = require('express');
const app = express();

app.use('/static', express.static('./upload'));

app.listen(3000, () => console.log(`Server running at port 3000`));
```

Статические файлы – это файлы расположенные **на стороне сервера** и предназначенные для **считывания** их **без изменения** с помощью HTTP GET-запроса по имени ресурса, включающего имя файла. Например, их можно получить с помощью тегов `<link>`, `<script>`, ``, `<audio>`, `<video>`, `<a>`, `<form>`, `<frame>`.

Для раздачи статических файлов в Express есть **middleware `express.static()`**, который указывает на каталог с файлами.



Cookies

HTTP headers | Set-Cookie

The **HTTP header Set-Cookie** is a response header and used to send cookies from the server to the user agent. So the user agent can send them back to the server later so the server can detect the user.

Syntax:

```
Set-Cookie: <cookie-name>=<cookie-value> | Expires=<date>
           | Max-Age=<non-zero-digit> | Domain=<domain-value>
           | Path=<path-value> | SameSite=Strict|Lax|none
```

Note: Using multiple directives are also possible.

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry
```

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Cookie бывают **постоянные** и **сессионные**.

Возможные опции:

- **domain** – привязка cookie к поддомену;
- **path** – путь, на который распространяется действие cookie;
- **maxAge** – время жизни cookie в миллисекундах.
- **Expires** – дата истечения жизни cookie;

- **secure** – может применяться только с HTTPS;
- **httpOnly** – если true, то cookie не доступны через JavaScript;
- **SameSite** (Strict|Lax|none) – обеспечивает защиту от CSRF-атак, контроль отправки cookie на другой домен.

Работа с cookies

```
const express = require("express");
const app = express();
const cookieparser = require('cookie-parser')(); // npm install cookie-parser

app.use(cookieparser);

app.get('/',(req, res, next)=>{

  let myid = req.cookies.myid;

  if (isFinite(myid)) ++myid;
  else myid = 0;

  res.cookie('myid', myid).send(`myid = ${myid}`);
});

app.listen(3000, ()=>console.log('Start server, port:', 3000));
```

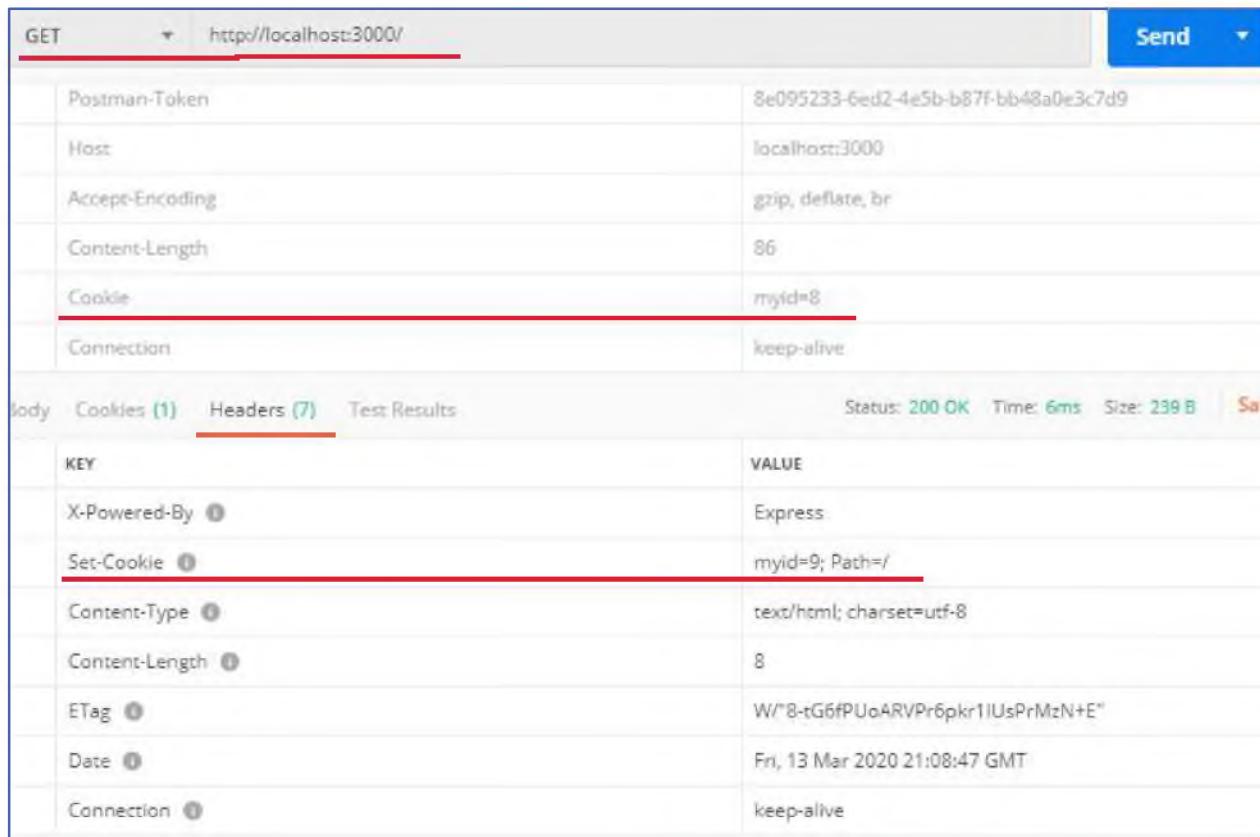
Опции с предыдущего слайда можно указать в `res.cookie(name, value [, options])` в объекте третьим параметром.

```
D:\PSCA\Lec20_cookie>node 20-03
Start server, port: 3000
myid = 1
myid = 2
myid = 3
myid = 4
myid = 5
```

`res.cookie (name, value [, options])` устанавливает имя cookie и значение у клиента путем отправки **заголовка ответа Set-Cookie**.

`cookie-parser` – это middleware, который **анализирует cookie**, прикрепленные к запросу клиента в заголовке `Cookie` и **заполняет req.cookies** объектом, ключами которого являются имена cookie.

Работа с cookies (результат)



Postman screenshot showing a successful GET request to `http://localhost:3000/`. The request includes a cookie `myId=8`. The response contains a `Set-Cookie` header with `myId=9; Path=/`. The status is `200 OK`.

Header	Value
Postman-Token	8e095233-6ed2-4e5b-b87f-bb48a0e3c7d9
Host	localhost:3000
Accept-Encoding	gzip, deflate, br
Content-Length	86
Cookie	myId=8
Connection	keep-alive

Header	Value
X-Powered-By	Express
Set-Cookie	myId=9; Path=/
Content-Type	text/html; charset=utf-8
Content-Length	8
ETag	W/"8-tG6fPUoARVPr6pkrtIUspMzN+E"
Date	Fri, 13 Mar 2020 21:08:47 GMT
Connection	keep-alive

Работа с signed cookies

```
const express = require("express");
const app = express();
const cookieparser = require('cookie-parser'); // npm install cookie-parser

const cookiesecret = '1234567890';

app.use(cookieparser(cookiesecret));

app.get('/',(req, res, next)=>{

    let myid = req.signedCookies.myid;

    if (isFinite(myid)) ++myid;
    else myid = 0;
    console.log('myid = ', myid);
    res.cookie('myid', myid, {signed:true}).send(`myid = ${myid}`);
});

app.listen(3000, ()=>console.log('Start server, port:', 3000));
```

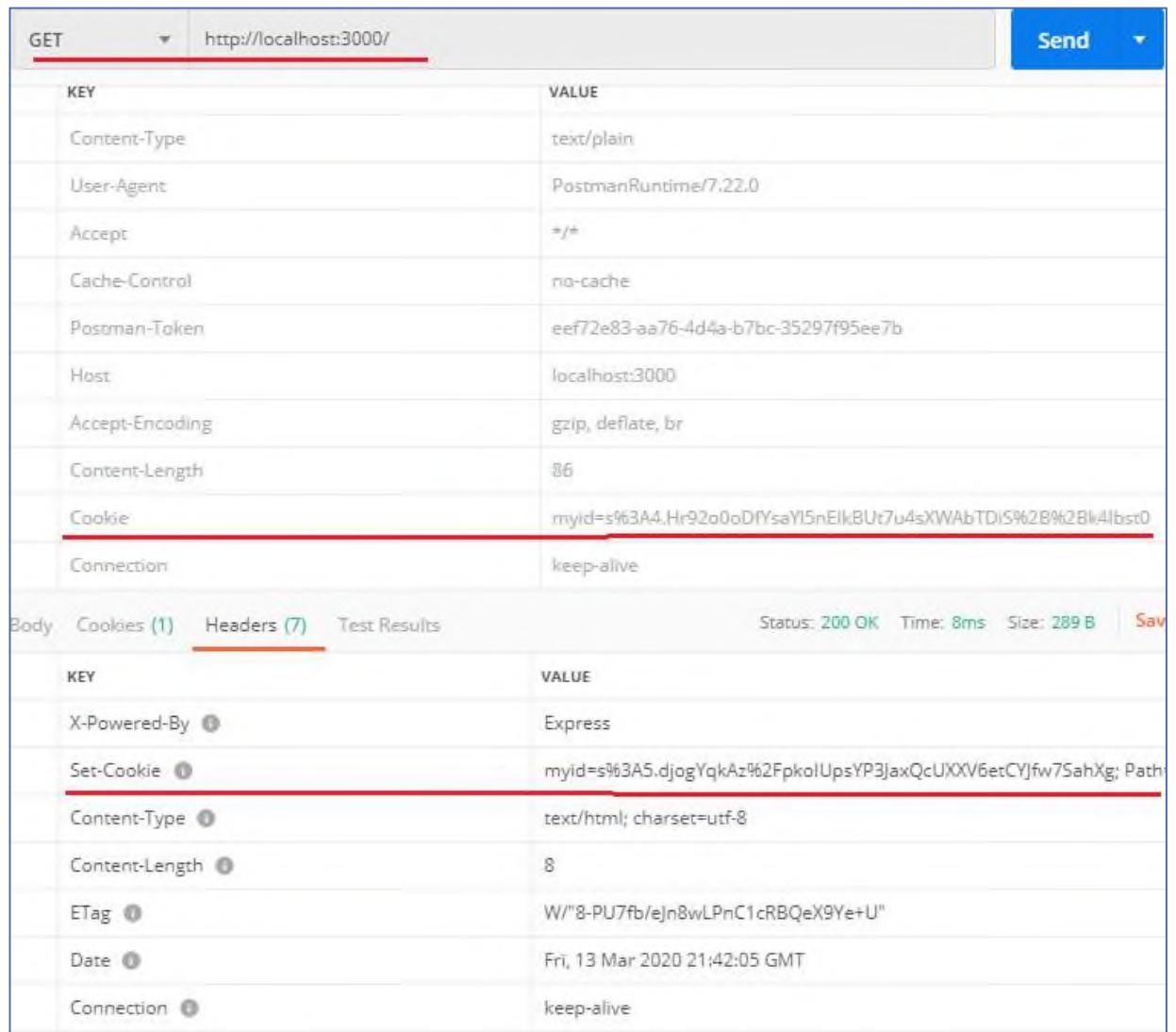
Для подписи cookie первым параметром в cookieParser() нужно **передать строку или массив**.

Для того, чтобы создать подписанную cookie, надо в res.cookie() третьим параметром в объекте **указать свойство signed: true**.

```
D:\PSCA\Lec20_cookie>node 20-03
Start server, port: 3000
myid = 1
myid = 2
myid = 3
myid = 4
myid = 5
```

Работа с signed cookies (результат)

Подписанные cookie не отправляются в виде простого текста, они кодируются в **base64**. Эти данные не являются секретными, но они не могут быть подделаны, потому что вторая часть cookie является подписью.



The screenshot shows the Postman interface with a GET request to `http://localhost:3000/`. The request includes the following headers:

KEY	VALUE
Content-Type	text/plain
User-Agent	PostmanRuntime/7.22.0
Accept	*/*
Cache-Control	no-cache
Postman-Token	eef72e83-aa76-4d4a-b7bc-35297f95ee7b
Host	localhost:3000
Accept-Encoding	gzip, deflate, br
Content-Length	86
Cookie	myid=s%63A4.Hr92o0oDlYsaYl5nElkBt7u4sXWabTDIS%2B%28lAlbst0
Connection	keep-alive

The response headers are:

KEY	VALUE
X-Powered-By	Express
Set-Cookie	myid=s%63A5.djogYqkAz%2FpkolUpsYP3JaxQcUXXV6etCYJfw7SahXg; Path=;
Content-Type	text/html; charset=utf-8
Content-Length	8
ETag	W/"8-PU7fb/ejn8wLPnC1cRBQeX9Ye+U"
Date	Fri, 13 Mar 2020 21:42:05 GMT
Connection	keep-alive

Удаление cookie

```
const express = require("express");
const app = express();
const cookieparser = require('cookie-parser')(); // npm install cookie-parser

app.use(cookieparser);

app.get('/',(req, res, next)=>{

  let myid = req.cookies.myid;

  if (isFinite(myid)) ++myid;
  else myid = 0;

  if (myid > 5) res.clearCookie('myid').send(`myid = ${myid=0}`);
  else res.cookie('myid', myid).send(`myid = ${myid}`);

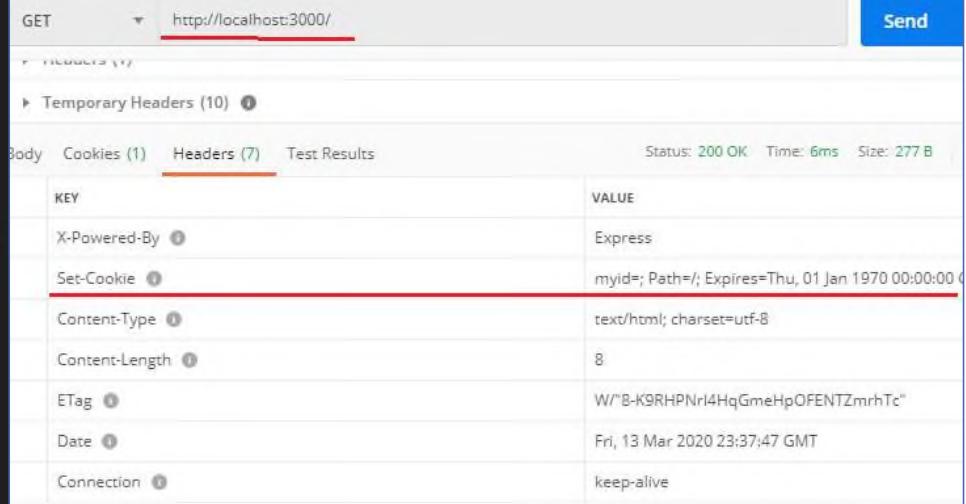
  console.log('myid = ', myid);

});

app.listen(3000,()=>console.log('Start server, port:', 3000));
```

res.clearCookie (name [, options])

Удаление происходит путем установки опции Expires в 1 января 1970 г.



KEY	VALUE
X-Powered-By	Express
Set-Cookie	myid=; Path=/; Expires=Thu, 01 Jan 1970 00:00:00
Content-Type	text/html; charset=utf-8
Content-Length	8
ETag	W/"B-K9RHPNrI4HqGmeHpOFENTZmrhTc"
Date	Fri, 13 Mar 2020 23:37:47 GMT
Connection	keep-alive

Работа с session

```
const app = express();
const cookiesecret = '1234567890';
const session = require('express-session')({
  resave: false,           // пересохранять ли, если нет изм.
  saveUninitialized: false, // сохранять ли пустые
  secret: cookiesecret,    // для подписи

  // store =      тип хранилища (Mem, DB)
  // cookie =     path, domain, secure, ...
});

app.use(session);

app.get('/', (req, res, next) => {
  if (!isFinite(req.session.mysesval)) req.session.mysesval = 0;
  else req.session.mysesval++;
  console.log('mysesval = ', req.session.mysesval);
  res.send(`mysesval = ${req.session.mysesval}`);
});

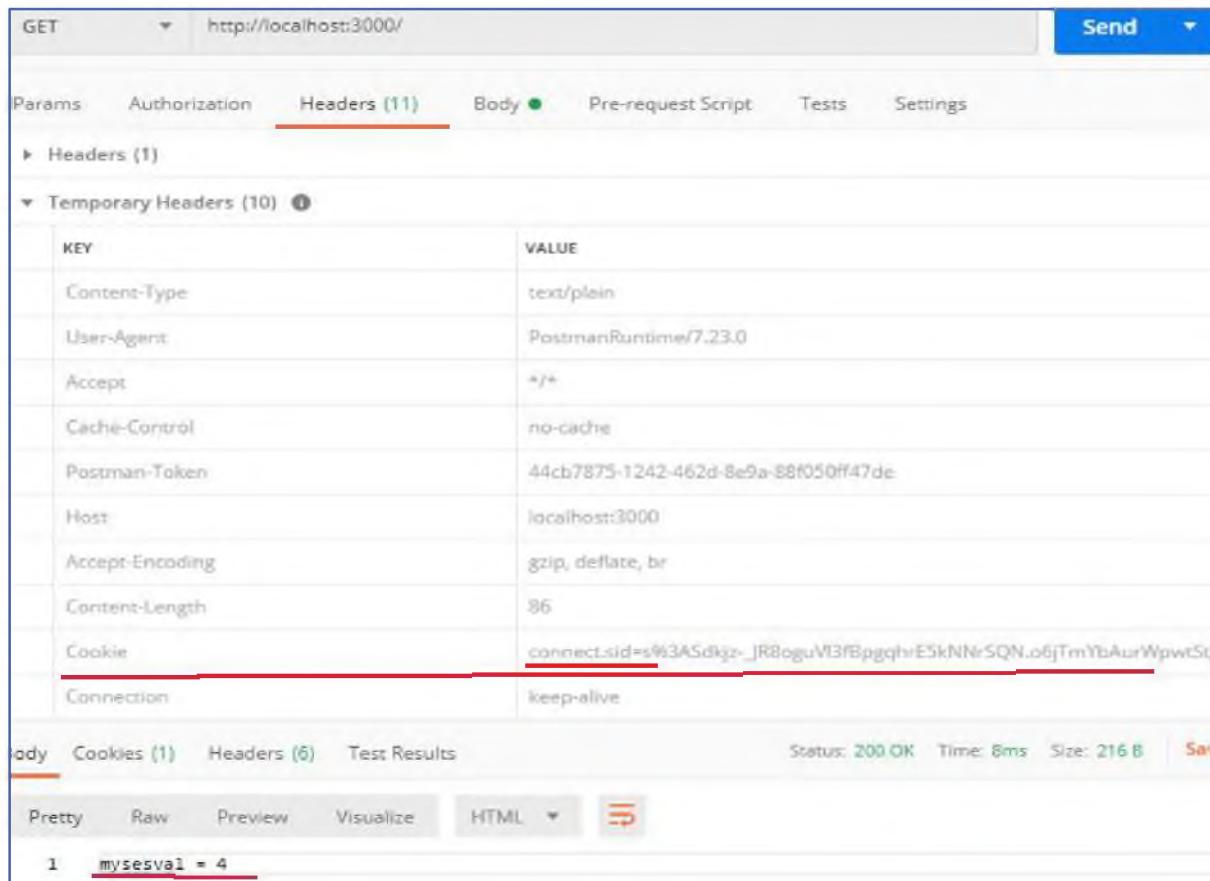
app.listen(3000, () => { console.log('Start server, port: ', 3000) });

```

express-session – middleware, который **создает session, устанавливает в cookie sessionid и создает объект req.session**. Всякий раз, когда происходит запрос от того же клиента, по sessionid из заголовка Cookie будет происходить поиск соответствующей сессии и ее запись в req.session (учитывая, что сервер не был перезапущен).

```
D:\PSCA\Lec20_cookie>node 20-04
Start server, port: 3000
mysesval =  0
mysesval =  1
mysesval =  2
mysesval =  3
mysesval =  4
```

Работа с session (результат)



The screenshot shows the Postman application interface. A GET request is made to `http://localhost:3000/`. The 'Headers' tab is selected, showing the following temporary headers:

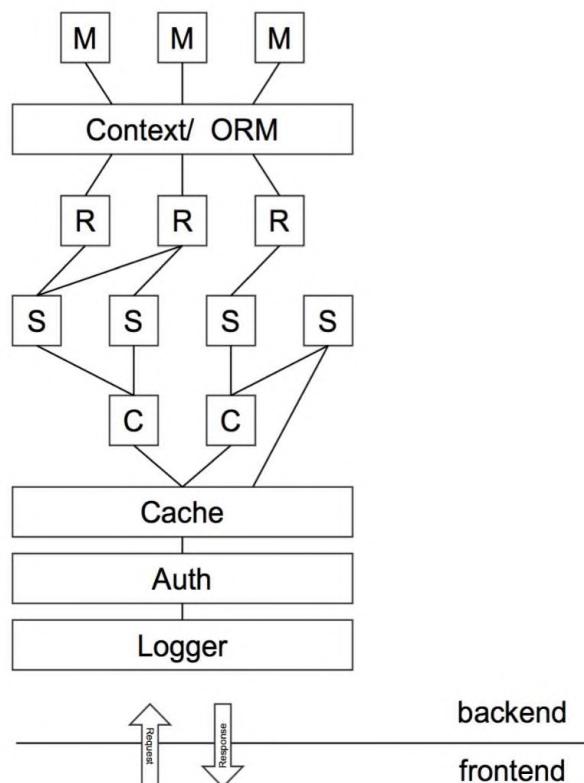
KEY	VALUE
Content-Type	text/plain
User-Agent	PostmanRuntime/7.23.0
Accept	*/*
Cache-Control	no-cache
Postman-Token	44cb7875-1242-462d-8e9a-88f050ff47de
Host	localhost:3000
Accept-Encoding	gzip, deflate, br
Content-Length	86
Cookie	connect.sid=s%3ASdkjz-JR8oguV3fBpgqhiE5kNNrSQN.o6jTmYbAurWpw5i
Connection	keep-alive

The 'Body' tab is selected, showing the response body:

```
1  mysesval = 4
```

The status bar at the bottom indicates: Status: 200 OK, Time: 8ms, Size: 216 B.

Архитектура N-layer (или N-tier)



Контроллеры: обработчики маршрутов, которые разбирают входящие запросы, потребляет 1 и более сервисов.

Сервисы: организация бизнес-логики, которая может быть функциональной или объектно-ориентированной по своей природе, валидация, потребляет 1 и более репозиториев, может возвращать DTO.

Репозитории: уровень доступа к данным, отвечает за выполнение запросов к базам данных, каждой модели – репозиторий, CRUD.

Контекст: подключение к бд, определение связей между моделями, перевод вызова методов в запросы, маппинг ответом на объекты

Модель: отражает сущность предметной области (таблица в БД)

```
✓ context
  JS index.js
✓ controllers
  JS facultyController.js
✓ global-controllers
  JS errorController.js
✓ helpers
  JS errors.js
  JS validator.js
✓ models
  JS faculty.js
✓ routers
  JS facultyRouter.js
✓ services
  JS facultyService.js
JS app.js
{} config.json
```

Структура проекта

Models

```
models > JS faculty.js > ...
1  module.exports = (Sequelize, sequelize) => {
2    return sequelize.define('Faculty', {
3      faculty: { type: Sequelize.STRING, allowNull: false, primaryKey: true },
4      faculty_name: { type: Sequelize.STRING, allowNull: false }
5    }, {
6      sequelize,
7      tableName: 'Faculty',
8      timestamps: false
9    });
10 }
```

faculty.js

Context

```
context > JS index.js > ...
1  const Sequelize = require('sequelize');
2  const config = require('../config.json')
3
4  const sequelize = new Sequelize(config.db.name, config.db.user, config.db.password, config.db.options);
5
6  const Faculty = require('../models/faculty')(Sequelize, sequelize)
7
8  // relations
9
10 module.exports = {
11   faculties: Faculty,
12   sequelize,
13   Sequelize,
14 };
index.js
```

```
{} config.json > ...
1  {
2    "db": {
3      "name": "SeqTest",
4      "user": "sa",
5      "password": "123",
6      "options": {
7        "host": "127.0.0.1",
8        "dialect": "mssql",
9        "dialectOptions": {
10          "options": {
11            "encrypt": false
12          }
13        }
14      }
15    }
16  }
```

config.json

Services

```
services > JS facultyService.js > ...
1  const Faculty = require('../context').faculties;
2  const errors = require('../helpers/errors');
3  const validator = require('../helpers/validator');
4
5  module.exports = {
6    getAll: async () => {
7      return Faculty.findAll();
8    },
9
10   getById: async (id) => {
11     let faculty = await Faculty.findOne({ where: { faculty: id } });
12     if (!faculty) throw errors.entityNotFound;
13
14     return faculty;
15   },
16
17   addFaculty: async (data) => {
18     const validationResult = validator.check('addFaculty', data);
19     let faculty = await Faculty.findOne({ where: { faculty: data.faculty } });
20
21     if (faculty) throw errors.invalidId;
22     if (validationResult.error) throw errors.invalidInput(validationResult.error.details[0].message);
23
24     return await Faculty.create(data);
25   }
26 }
```

facultyService.js

Helpers (validation, errors)

```
helpers > JS validator.js > ...
1  const Joi = require('joi');
2
3  const schemas = {
4      'addFaculty': Joi.object().keys({
5          faculty: Joi.string().min(2).max(10).required(),
6          faculty_name: Joi.string().required()
7      })
8  };
9
10 exports.check = function (schema, body) {
11     if (!schemas[schema]) return {};
12
13     return schemas[schema].validate(body);
14};
```

validator.js

```
helpers > JS errors.js > ...
1  const express = require('express');
2
3  express.response.error = function (error) {
4      if (!error.code) {
5          error = {
6              message: error.toString(),
7              code: 'server_error',
8              status: 500
9          };
10
11      }
12
13      this.status(error.status).json(error);
14  };
15
16  module.exports = {
17      invalidId: {
18          message: 'Already exists',
19          code: 'already_exists',
20          status: 400
21      },
22      invalidInput: (message) => {
23          return {
24              message: message,
25              code: 'invalid_input',
26              status: 400
27          };
28      },
29      entityNotFound: {
30          message: 'Entity not found',
31          code: 'entity_not_found',
32          status: 404
33      },
34      methodNotAllowed: {
35          message: 'Method not allowed',
36          code: 'method_not_allwed',
37          status: 405
38      },
39      resourceNotFound: {
40          message: 'Resource not found',
41          code: 'resource_not_found',
42          status: 404
43      }
44  };

```

errors.js

Controllers

```
global-controllers > JS errorController.js > ...
1  module.exports = (error, req, res, next) => {
2      // TODO: log error + 'res.locals.trace'
3    res.error(error);
4  };

```

errorController.js

```
controllers > JS facultyController.js > ...
1  const facultyService = require('../services/facultyService');
2
3  module.exports = {
4    getAll: async (req, res, next) => {
5        try {
6        res.json(await facultyService.getAll());
7      } catch (error) {
8        next(error);
9      }
10    },
11  },
12
13  getById: async (req, res, next) => {
14      try {
15      res.json(await facultyService.getById(req.params.id));
16    } catch (error) {
17      next(error);
18    }
19  },
20
21  addFaculty: async (req, res, next) => {
22      try {
23      const newFaculty = req.body;
24      res.json(await facultyService.addFaculty(newFaculty));
25    } catch (error) {
26      next(error);
27    }
28  }
29
30

```

facultyController.js

Routers

```
routers > JS facultyRouter.js > ...
 1  const express = require('express');
 2  const facultyController = require('../controllers/facultyController');
 3  const errors = require('../helpers/errors');
 4
 5  module.exports = () => {
 6    let router = express.Router();
 7
 8    router.route('/')
 9      .get(facultyController.getAll)
10      .post(facultyController.addFaculty)
11      .all((req, res, next) => res.error(errors.methodNotAllowed));
12
13    router.route('/:id')
14      .get(facultyController.getById)
15      .all((req, res, next) => res.error(errors.methodNotAllowed));
16
17    return router;
18  };

```

facultyRouter.js

app.js

```
JS app.js > ...
1  const express = require('express');
2  const bodyParser = require('body-parser');
3
4  const errors = require('./helpers/errors');
5  const errorController = require('./global-controllers/errorController');
6  const facultyRoutes = require('./routers/facultyRouter')();
7
8  let app = express();
9
10 app.use(bodyParser.json({ extended: false }));
11 app.use('/faculties/', facultyRoutes);
12 app.use((req, res, next) => {
13   res.error(errors.resourceNotFound)
14 })
15 app.use(errorController);
16
17 app.listen(3000, () => console.log(`Server running at port 3000`));
18
```

app.js

Демонстрация работы (404, 405)

The image shows two requests in the Postman application:

Request 1: GET http://localhost:3000/qwerty

- Method: GET
- URL: http://localhost:3000/qwerty
- Headers (7): (not shown)
- Body: (Pretty) JSON response:

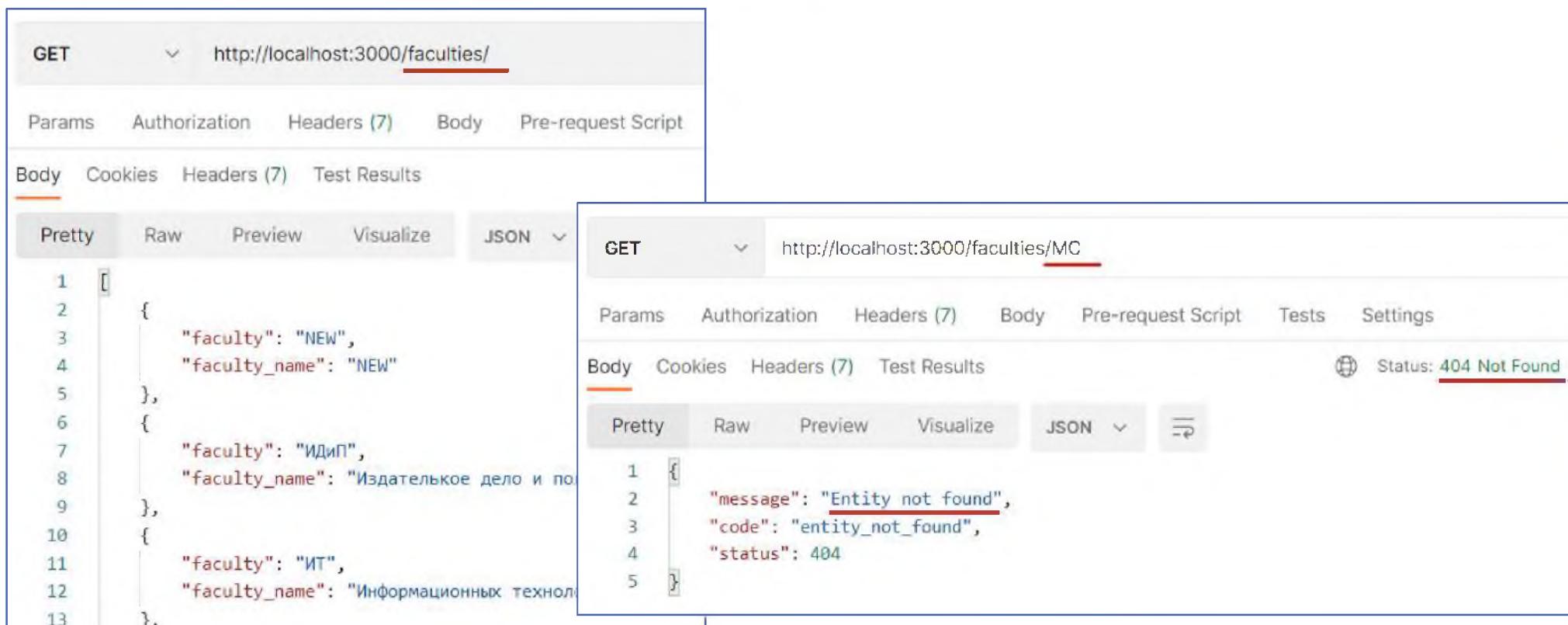
```
1 {  
2   "message": "Resource not found",  
3   "code": "resource_not_found",  
4   "status": 404  
5 }
```

Request 2: PUT http://localhost:3000/faculties/MC

- Method: PUT
- URL: http://localhost:3000/faculties/MC
- Headers (8): (not shown)
- Body: (Pretty) JSON response:

```
1 {  
2   "message": "Method not allowed",  
3   "code": "method_not_allwed",  
4   "status": 405  
5 }
```

Демонстрация работы



GET <http://localhost:3000/faculties/>

Params Authorization Headers (7) Body Pre-request Script

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 [  
2 {  
3   "faculty": "NEW",  
4   "faculty_name": "NEW"  
5 },  
6 {  
7   "faculty": "ИДИП",  
8   "faculty_name": "Издательское дело и по"  
9 },  
10 {  
11   "faculty": "ИТ",  
12   "faculty_name": "Информационных технол"  
13 },
```

GET <http://localhost:3000/faculties/MC>

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Body Cookies Headers (7) Test Results

>Status: 404 Not Found

Pretty Raw Preview Visualize JSON

```
1 {  
2   "message": "Entity not found",  
3   "code": "entity_not_found",  
4   "status": 404  
5 }
```

Демонстрация работы

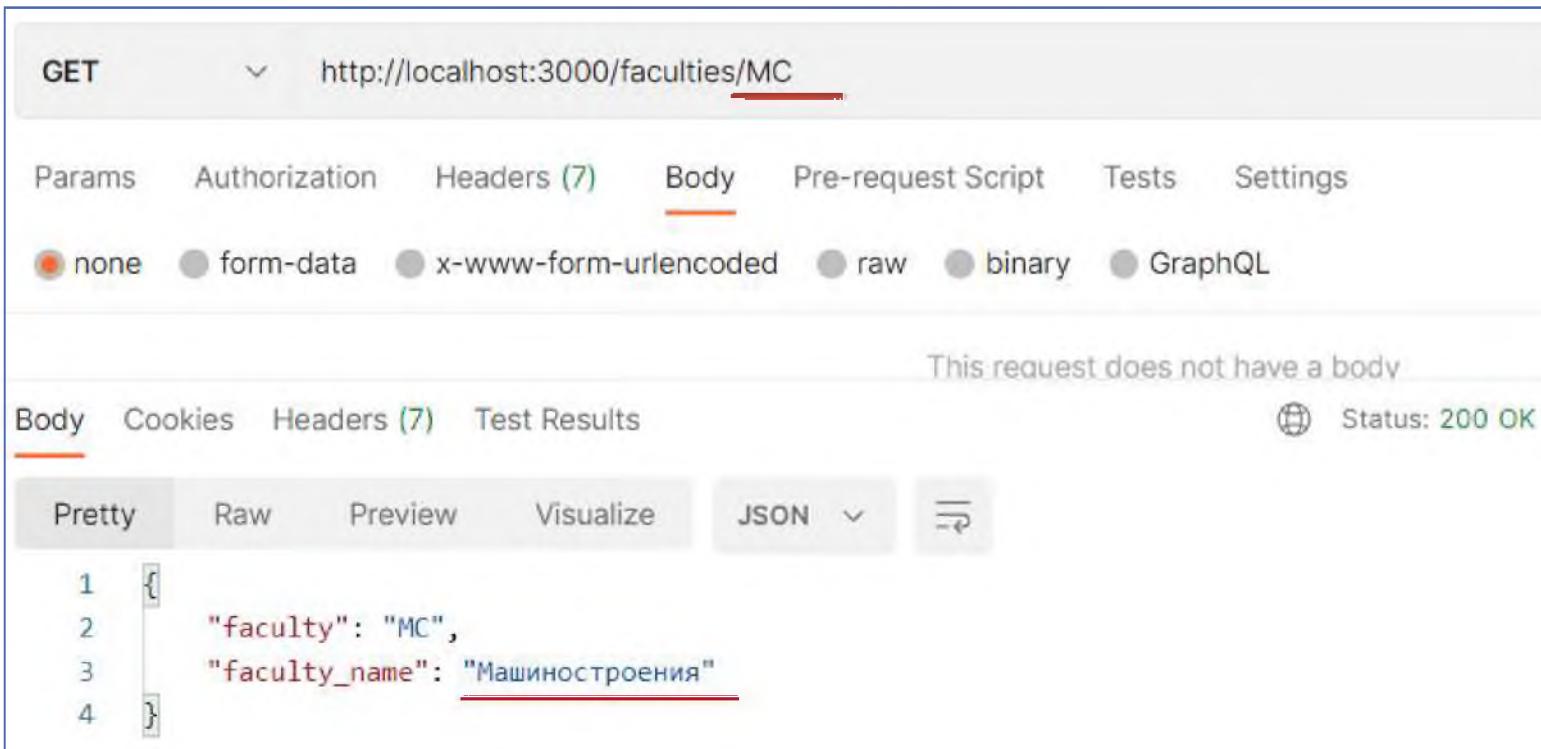
Left Request (Successful):

```
POST http://localhost:3000/faculties
Body (JSON)
{
  "faculty": "MC",
  "faculty_name": "Машиностроения"
}
```

Right Request (Failed):

```
POST http://localhost:3000/faculties
Status: 400 Bad Request
Body (JSON)
{
  "message": "Already exists",
  "code": "already_exists",
  "status": 400
}
```

Демонстрация работы



GET <http://localhost:3000/faculties/MC>

Params Authorization Headers (7) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (7) Test Results 🌐 Status: 200 OK

Pretty Raw Preview Visualize JSON ≡

```
1 {
2   "faculty": "MC",
3   "faculty_name": "Машиностроения"
4 }
```

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

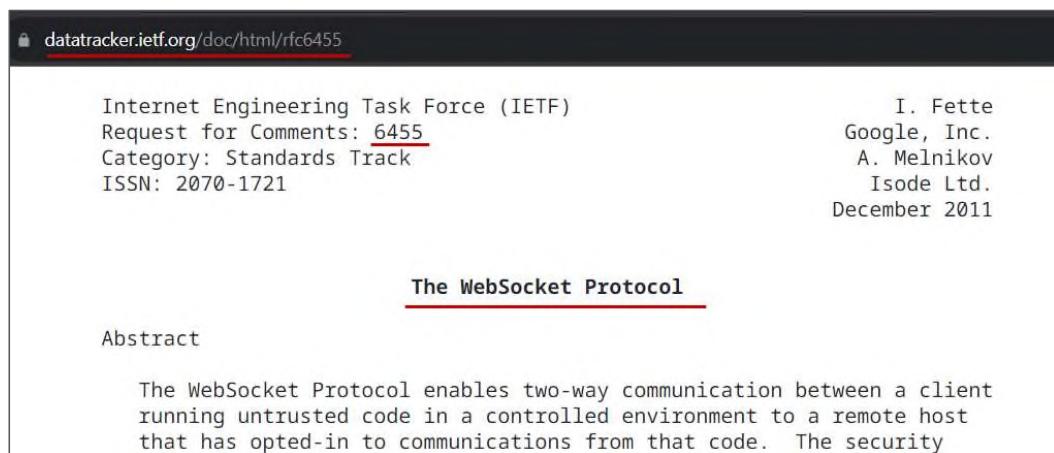
WEBSOCKET

Websocket

=

протокол прикладного уровня для
полнодуплексной связи поверх TCP-соединения,
предназначенный для обмена сообщениями
между браузером и веб-сервером через
постоянное соединение в режиме реального
времени.

Данные передаются по нему **в обоих**
направлениях в виде «датафреймов», без разрыва
соединения и дополнительных HTTP-запросов.

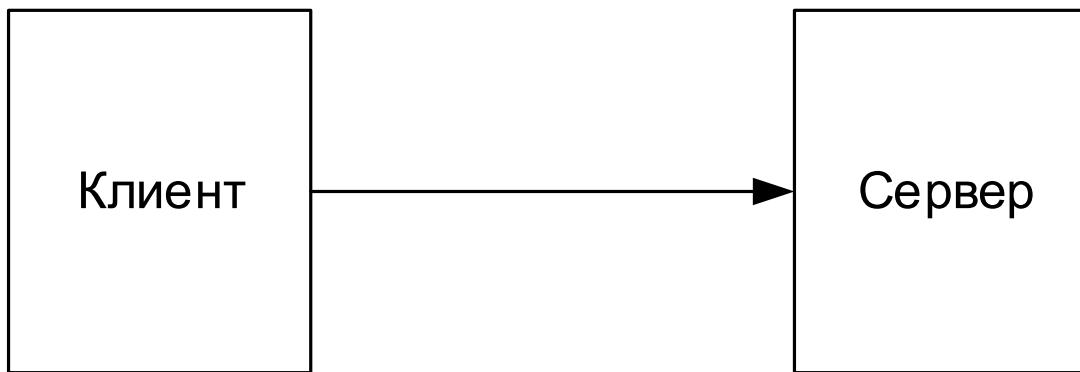


стандарт

websocket **можно использовать** в:

- приложениях реального времени;
- чат-приложениях;
- IoT-приложениях;
- приложениях для бирж и аукционов;
- многопользовательских играх;
- и другое.

Симплекс

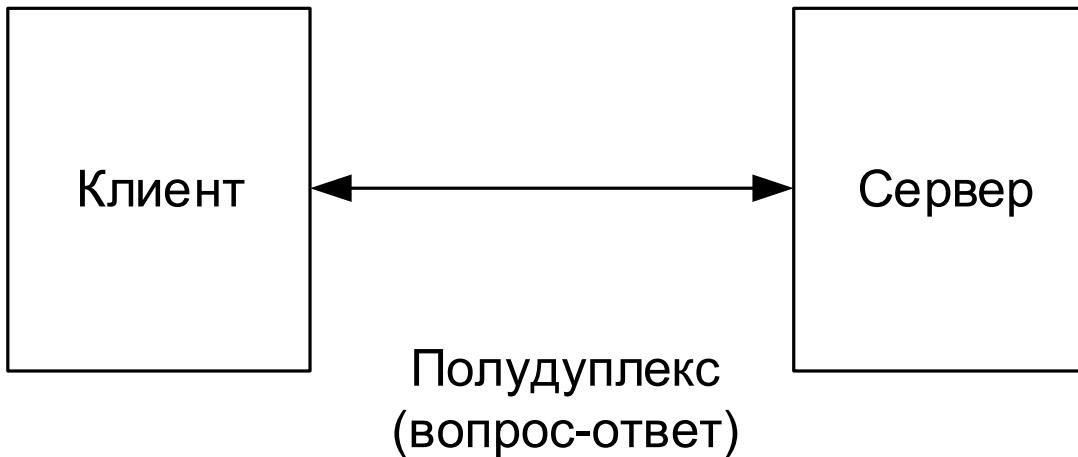


Симплекс
(однонаправленный)

это **односторонний** канал, данные по нему могут **передаваться только в одном направлении**.
Первый узел способен отсылать сообщения, второй может только принимать их, но не может подтвердить получение или ответить.

Пример: радио, Mailslot.

Полудуплекс

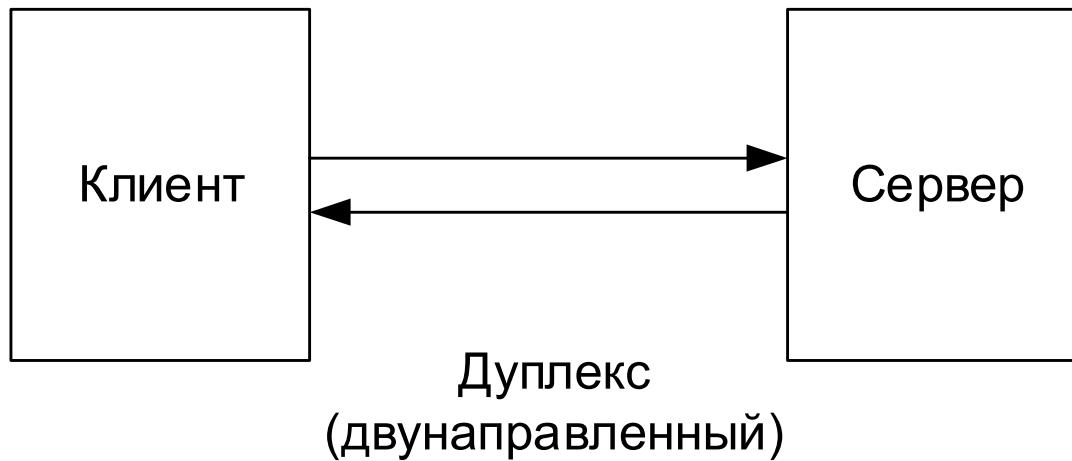


двусторонний канал связи, данные по нему могут **передаваться в одном направлении в один момент времени**. Оба абонента имеют возможность принимать и передавать сообщения.

Поток сообщений может идти в обоих направлениях, но не одновременно.

Пример: радио, HTTP.

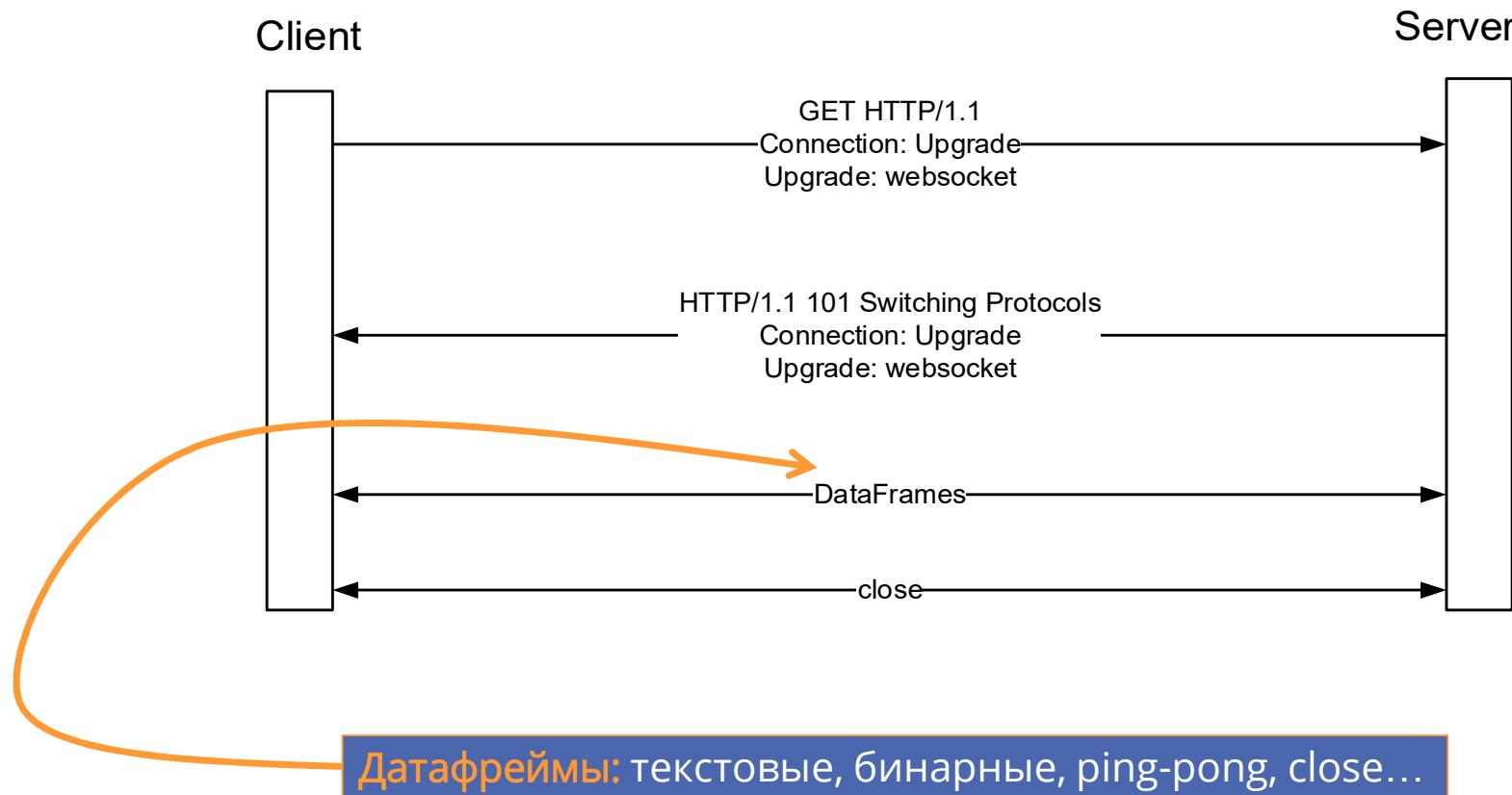
Дуплекс



двусторонний канал связи, данные могут **передаваться в обе стороны одновременно**. Дуплекс имеет два физически отдельных канала связи, один из которых предназначен для движения данных в одном направлении, а другой – в противоположном направлении.

Пример: телефон, WebSocket.

WebSocket-handshake



Простейший websocket-сервер

```
const httpserver = require('http').createServer((req, res)=>{
  if (req.method == 'GET' && req.url == '/start' ){
    res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
    res.end(require('fs').readFileSync('./09-01.html'));
  }
});
httpserver.listen(3000)
console.log ('ws server: 3000');

let k = 0;
const WebSocket = require('ws') // npm install ws
const wsserver = new WebSocket.Server({ port: 4000, host:'localhost', path:'wsserver' })
wsserver.on('connection', (ws) => {
  ws.on('message', message => {
    console.log(`Received message => ${message}`);
  })
  setInterval(()=>{ws.send(`server: ${++k}`)}, 1500);
})
wsserver.on('error',(e)=>{console.log('ws server error', e)});
console.log(`ws server: host:${wsserver.options.host}, port:${wsserver.options.port}, path:${wsserver.options.path}`);
```

Экземпляр класса
WebSocketServer

Экземпляр класса
WebSocket

Событие **connection** генерируется после завершения рукопожатия

Генерируется при получении сообщения.

Класс WebSocketServer

События

- **close** генерируется при закрытии сервера. Это событие зависит от события закрытия HTTP-сервера только в том случае, если он создается внутри.
- **connection** генерируется после завершения рукопожатия. Первым параметром передается сокет (экземпляр класса `WebSocket`), вторым параметром `request` (экземпляр класса `http.IncomingMessage`) – это HTTP-запрос GET, отправленный клиентом.
- **error** генерируется при возникновении ошибки.
- **headers** генерируется перед записью заголовков ответа в сокет в рамках рукопожатия. Первым параметром передается массив заголовков, а вторым – `request` (экземпляр класса `http.IncomingMessage`).
- **listening** генерируется, когда базовый сервер привязан.

Класс WebSocketServer

Свойства

- `clients` – набор, в котором хранятся все подключенные клиенты.

Методы

- `.address()` возвращает объект со свойствами `port`, `family` и `address`, сообщаемые операционной системой при прослушивании сокета.
- `.close([callback])` запрещает серверу принимать новые соединения и закрывает HTTP-сервер, если он создан внутри. Если внешний HTTP-сервер используется через параметры конструктора `server` или `noServer`, его необходимо закрыть вручную. Необязательный `callback` вызывается при возникновении события закрытия и получает ошибку, если сервер уже закрыт.
- `.handleUpgrade(request, socket, head, callback)` обработка запроса на обновление HTTP. Чаще всего вызывается автоматически. При работе в режиме «`noServer`» этот метод необходимо вызывать вручную.

Простейший websocket-клиент (браузер)

```
<body>
  <h1>Lec 09</h1>
  <script>
    let k = 0;
    function startWS(){
      let socket = new WebSocket('ws://localhost:4000/wsserver');

      socket.onopen = ()=>{ console.log('socket.onopen');
        setInterval(()=>{socket.send(++k);}, 1000);
      };

      socket.onclose = (e)=>{ console.log('socket.onclose', e);};

      socket.onmessage =(e)=>{console.log('socket.onmessage', e.data);}

      socket.onerror = function(error) { alert("Ошибка " + error.message); };
    };
  </script>
  <button onclick="startWS()">startWS</button>
</body>
```

Этот класс представляет WebSocket.
Он наследует EventEmitter.

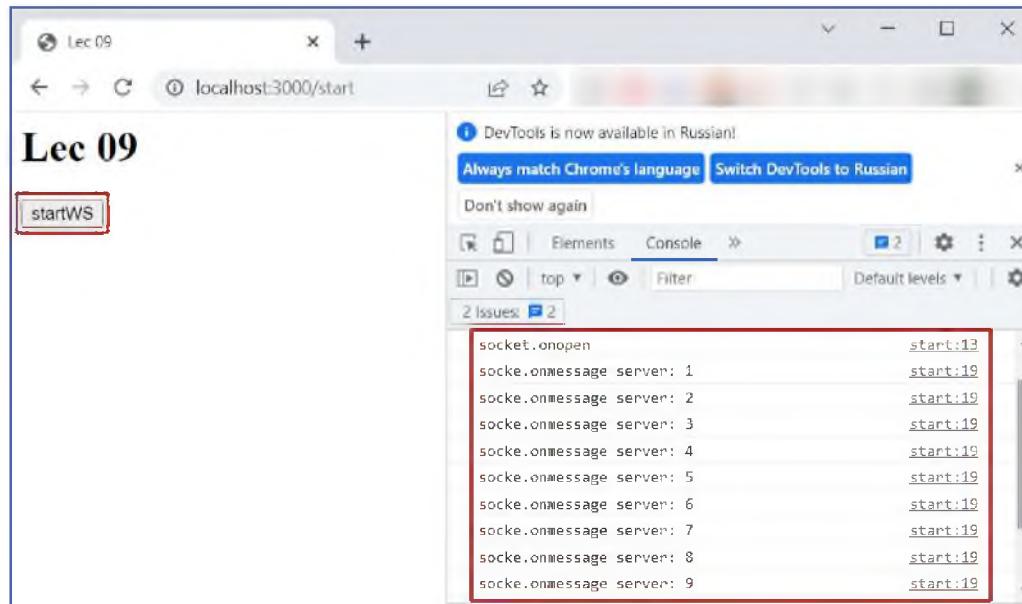
```
PS D:\NodeJS\samples\cwp_08> node
08-00
ws server: 3000
ws server: host:localhost, port:40
00, path:/wsserver
Received message => 1
Received message => 2
Received message => 3
Received message => 4
Received message => 5
Received message => 6
```

Для установки соединения в скрипте создаем экземпляр **класса WebSocket**, в конструктор которого передаем параметр address (со специальным протоколом **ws** (нешифрованное соединение) или **wss** (зашифрованное соединение)).

У объекта WebSocket можно слушать его события, например:

- **open** – соединение установлено;
- **message** – получены данные;
- **error** – ошибка;
- **close** – соединение закрыто.

Демонстрация работы



Lec 09

localhost:3000/start

startWS

DevTools is now available in Russian!

Always match Chrome's language Switch DevTools to Russian

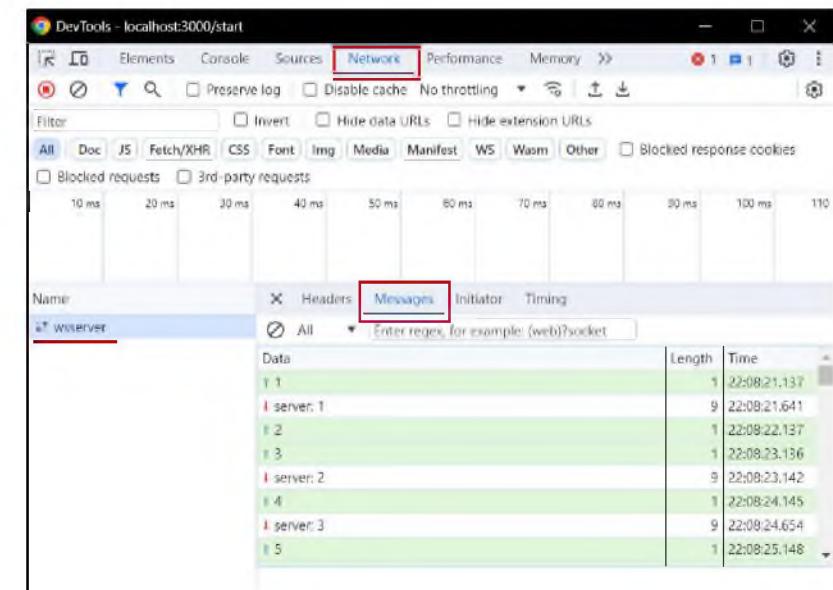
Don't show again

Elements Console

top Filter Default levels

2 Issues:

socket.onopen	start:13
socket.onmessage server: 1	start:19
socket.onmessage server: 2	start:19
socket.onmessage server: 3	start:19
socket.onmessage server: 4	start:19
socket.onmessage server: 5	start:19
socket.onmessage server: 6	start:19
socket.onmessage server: 7	start:19
socket.onmessage server: 8	start:19
socket.onmessage server: 9	start:19



DevTools - localhost:3000/start

Network Performance Memory

Preserve log Disable cache No throttling

All Doc JS Fetch/XHR CSS Font Img Media Manifest W5 Wasm Other Blocked response cookies

Name Headers Messages Initiator Timing

ws:server

Data	Length	Time
1	1	22:08:21.137
server: 1	9	22:08:21.641
2	1	22:08:22.137
3	1	22:08:23.136
server: 2	9	22:08:23.142
4	1	22:08:24.145
server: 3	9	22:08:24.654
5	1	22:08:25.148

Класс WebSocket

События

- **close** генерируется при закрытии соединения. Первым параметром передается code – числовое значение, указывающее код состояния, объясняющий, почему соединение было закрыто, вторым reason – это буфер, содержащий строку, объясняющую, почему соединение было закрыто.
- **error** генерируется при возникновении ошибки.
- **message** генерируется при получении сообщения. Первым параметром передается data – это содержимое сообщения, вторым isBinary – указывает, является ли сообщение двоичным или нет.
- **open** генерируется при установлении соединения.
- **ping** генерируется при получении ping.
- **pong** генерируется при получении pong.
- **upgrade** генерируется, когда заголовки ответа получены от сервера как часть рукопожатия.

Класс WebSocket

для браузерного клиента

Методы

- `.addEventListener(type, listener[, options])` добавить слушатель listener на событие type.
- `.removeEventListener(type, listener)` удалить слушатель listener с события type.
- `.send(data[, options][, callback])` отправляет данные. В options можно настроить в бинарном ли виде должны быть отправлены данные(binary), надо ли их сжимать(compress), последний ли это фрагмент сообщения(fin) и надо ли маскировать передаваемые данные(mask).
- `.close([code[, reason]])` инициирует заключительное рукопожатие.
- `.ping([data[, mask]][, callback])` отправляет фрейм ping. В data можно передать данные для отправки во фрейме.
- `.pong([data[, mask]][, callback])` отправляет фрейм pong. В data можно передать данные для отправки во фрейме.
- `.terminate()` принудительно закрывает соединение. Внутри это вызывает метод `socket.destroy()`.

Класс WebSocket

Свойства

для браузерного клиента

- **binaryType** указывает тип двоичных данных, передаваемых по соединению.
- **bufferedAmount** указывает количество байтов данных, которые были поставлены в очередь с помощью вызовов `send()`, но еще не переданы в сеть.
- **onclose** позволяет добавить слушатель событий, который будет вызываться при закрытии соединения
- **onerror** позволяет добавить слушатель событий, который будет вызываться при возникновении ошибки.
- **onmessage** позволяет добавить слушатель событий, который будет вызываться при получении сообщения.
- **onopen** позволяет добавить слушатель событий, который будет вызываться при установке соединения.
- **readyState** содержит текущее состояние соединения
- **url** содержит URL-адрес сервера WebSocket. Серверные клиенты не имеют этого атрибута.

Простейший websocket-клиент (серверный)

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://localhost:4000/wsserver');

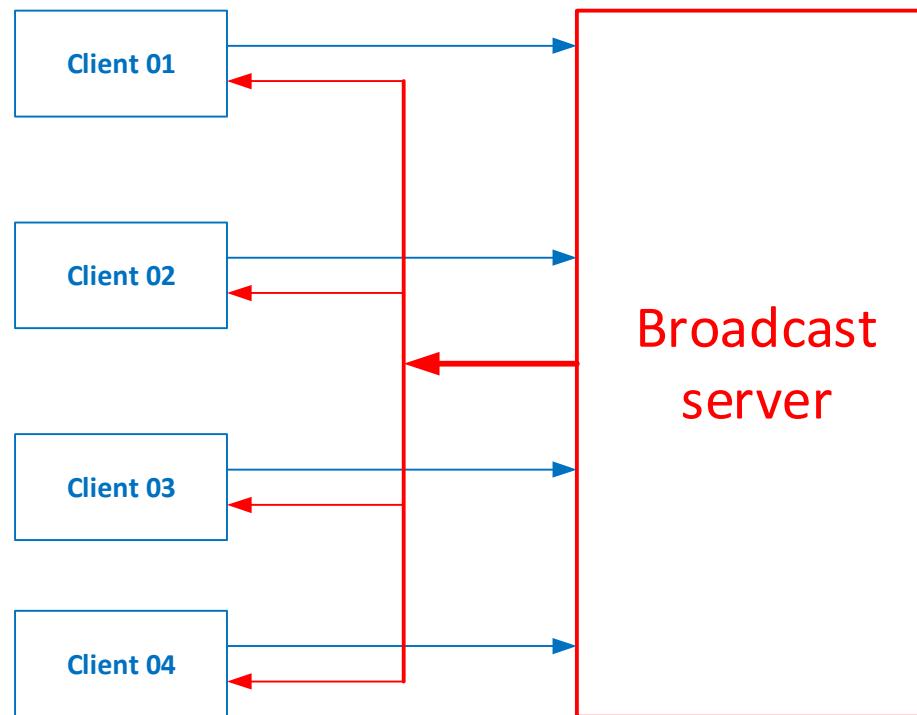
ws.on('open', () =>{
  ws.send('message 1'); // отправить сообщение серверу
  ws.send('message 2'); // отправить сообщение серверу
  ws.on('message', message => {
    console.log(`Received message => ${message}`);
  })
  setTimeout(()=>{ws.close()},5000); // остановить через 5 секунд
});
```

С помощью метода `close()` инициируем завершающее рукопожатие и закрываем соединение.

```
PS D:\NodeJS\samples\cwp_08> node .\08-00.js
ws server: 3000
ws server: host:localhost, port:4000, path:/wsserver
Received message => message 1
Received message => message 2
[]
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-00b.js
Received message => server: 1
Received message => server: 2
Received message => server: 3
PS D:\NodeJS\samples\cwp_08> []
```

Широковещательный websocket-сервер



Такой сервер позволяет отправлять сообщения, которые доставляются
всем узлам сети.

Широковещательный websocket-сервер

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 5000, host: 'localhost', path: '/broadcast' });

wss.on('connection', (ws) => {
  ws.on('message', (message) => {
    wss.clients.forEach((client) => {
      if (client.readyState === WebSocket.OPEN) client.send('server: ' + message);
    });
  })
  ws.on('close', () => { console.log('wss socket closed'); })
});
```

Свойство **clients** хранит набор со всеми подключенными клиентами.

Свойство **readyState** содержит текущее состояние соединения (CONNECTING, OPEN, CLOSING, CLOSED).

Клиент для широковещательного сервера с параметрами командной строки

```
const WebSocket = require('ws');

let parm0 = process.argv[0]; // path node
let parm1 = process.argv[1]; // path application
let parm2 = process.argv[2]; // first parameter

console.log('parm2 = ', parm2);

let prfx = typeof parm2 == 'undefined'?'A':parm2;
const ws = new WebSocket('ws://localhost:5000/broadcast');

ws.on('open', () =>{

    let k = 0;
    setInterval(()=>{
        ws.send(`client: ${prfx}-${[+k]}`); // отправить сообщение серверу
    }, 1000);

    ws.on('message', message => {
        console.log(`Received message => ${message}`)
    })
});

setTimeout(()=>{ws.close()},25000); // остановить через 25 секунд
});
```

При запуске приложения из терминала/командной строки мы можем передавать процессу **параметры**. Для получения параметров в коде приложения применяется массив `process.argv`.

```
PS D:\NodeJS\samples\cwp_08> node .\08-01.js
wss socket closed
wss socket closed
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01a.js M
param2 = M
Received message => server: client: M-1
Received message => server: client: M-2
Received message => server: client: K-1
Received message => server: client: M-3
Received message => server: client: K-2
Received message => server: client: M-4
Received message => server: client: K-3
Received message => server: client: M-5
Received message => server: client: K-4
Received message => server: client: M-6
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01a.js K
param2 = K
Received message => server: client: M-1
Received message => server: client: M-2
Received message => server: client: K-1
Received message => server: client: M-3
Received message => server: client: K-2
Received message => server: client: M-4
Received message => server: client: K-3
Received message => server: client: M-5
Received message => server: client: K-4
Received message => server: client: M-6
Received message => server: client: K-5
```

Дуплексный поток данных

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://localhost:5000/broadcast');

const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });

duplex.pipe(process.stdout); // сообщения от сервера --> stdout

process.stdin.pipe(duplex); // stdin --> сообщение серверу
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01.js
wss socket closed
[]
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01b.js
Hello from M
server: Hello from M
server: Hello from K
PS D:\NodeJS\samples\cwp_08> []
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-01b.js
server: Hello from M
Hello from K
server: Hello from K
PS D:\NodeJS\samples\cwp_08> []
```

`createWebSocketStream(websocket[, options])` возвращает **дуплексный поток**, который позволяет использовать API потоков Node.js поверх заданного WebSocket.

Механизм ping/pong (сервер => клиент)

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 5000, host:'localhost'});
wss.on('connection', (ws)=>{
    ws.on('pong', (data)=>{
        console.log('on pong: ', data.toString());
    });
    ws.on('message', (data)=>{
        console.log('on message: ', data.toString());
        ws.send(data);
    });
    setInterval(()=>{ console.log('server: ping'); ws.ping('server: ping')}, 5000);
});
```

сервер

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://localhost:5000');

const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });

duplex.pipe(process.stdout); // сообщения от сервера --> stdout

process.stdin.pipe(duplex); // stdin --> сообщение серверу

// ws.on('ping', (data)=>{ // можно ловить
//     console.log('on ping: ', data.toString());
//});
```

клиент

В протокол WebSocket встроена проверка связи при помощи **управляющих фреймов типа PING и PONG**.

Тот, кто хочет проверить соединение, отправляет фрейм PING с произвольным телом. Его получатель должен в разумное время ответить фреймом PONG с тем же телом.

```
PS D:\NodeJS\samples\cwp_08> node .\08-02.js
server: ping
on pong: server: ping
server: ping
on pong: server: ping
on message: Hi!
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-02a.js
Hi!
Hi!
```

результат

Механизм ping/pong (клиент => сервер)

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 5000, host:'localhost'});
wss.on('connection', (ws)=>{
    ws.on('message', (data)=>{
        console.log('on message: ', data.toString());
        ws.send(data);
    });
});
wss.on('error',(e)=>{console.log('wss server error', e)});
```

сервер

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://localhost:5000');

const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });

duplex.pipe(process.stdout); // сообщения от сервера --> stdout

process.stdin.pipe(duplex); // stdin --> сообщение серверу

ws.on('pong', (data)=>{ // можно ловить
    console.log('on pong: ', data.toString());
});
setInterval(()=>{ console.log('server: ping'); ws.ping('client: ping')}, 5000);
```

клиент

Событие ping возникает, когда получен фрейм PING.

Событие pong возникает, когда получен фрейм PONG.

Метод ping() отправляет PING, в параметрах можно передать какую-то информацию.

Метод pong() отправляет PONG. Однако сообщения PONG чаще всего автоматически отправляются в ответ на сообщения PING, как того требует спецификация.

```
PS D:\NodeJS\samples\cwp_08> node .\08-03.js
on message: HI!
```

[]

```
PS D:\NodeJS\samples\cwp_08> node .\08-03a.js
```

```
client: ping
on pong: client: ping
client: ping
on pong: client: ping
HI!
HI!
client: ping
on pong: client: ping
client: ping
on pong: client: ping
client: ping
```

результат

Отправка JSON

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 5000, host:'localhost'});
wss.on('connection', (ws)=>{

    ws.on('message', (data)=>{console.log('on message: ', JSON.parse(data)); });

    let k = 0;
    setInterval(()=>{ ws.send(JSON.stringify({k:k, sender:'Server', date: new Date().toISOString()})); }, 5000);

});
```

сервер

```
const WebSocket = require('ws');
const ws = new WebSocket('ws://localhost:5000');
ws.on('open', () =>{
    ws.on('message', data => { console.log('on message: ', JSON.parse(data)); })
    let k = 0;
    setInterval(()=>{ws.send(JSON.stringify({k:k, sender:'Client', date: new Date().toISOString()})); }, 3000);
});
```

клиент

```
PS D:\NodeJS\samples\cwp_08> node .\08-04.js
on message: { k: 1, sender: 'Client', date:
'2022-10-28T22:30:05.136Z' }
on message: { k: 2, sender: 'Client', date:
'2022-10-28T22:30:10.148Z' }
on message: { k: 3, sender: 'Client', date:
'2022-10-28T22:30:15.162Z' }
[]
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-04a.js
on message: { k: 1, sender: 'Server', date:
'2022-10-28T22:30:05.136Z' }
on message: { k: 2, sender: 'Server', date:
'2022-10-28T22:30:10.148Z' }
on message: { k: 3, sender: 'Server', date:
'2022-10-28T22:30:15.162Z' }
```

Upload file

```
const fs = require('fs');
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 5000, host:'localhost'});
let k = 0;
wss.on('connection', (ws)=>{
    const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
    let wfile = fs.createWriteStream(`./file${++k}.txt`);
    duplex.pipe(wfile);
});
```

сервер

1. получаем дуплексный поток
2. открываем поток записи в файл
3. информацию, поступающую из дуплексного потока, записываем в поток записи файла

```
const fs = require('fs');
const WebSocket = require('ws');
const ws = new WebSocket('ws://localhost:5000');
ws.on('open', () =>{
    const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
    let rfile = fs.createReadStream(`./MyFile.txt`);
    rfile.pipe(duplex);
});
```

клиент

1. получаем дуплексный поток
2. открываем поток чтения из файла
3. Информацию из потока чтения перенаправляем в дуплексный поток

Download file

```
const fs = require('fs');
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 5000, host:'localhost'});
wss.on('connection', (ws)=>{
  const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
  let rfile = fs.createReadStream('./MyFile.txt');
  rfile.pipe(duplex);
});
```

сервер

То же самое, но наоборот.

```
const fs = require('fs');
const WebSocket = require('ws');
const ws = new WebSocket('ws://localhost:5000');
let k = 0;
ws.on('open', () =>{
  const duplex = WebSocket.createWebSocketStream(ws, { encoding: 'utf8' });
  let wfile = fs.createWriteStream(`./MyFile${++k}.txt`);
  duplex.pipe(wfile);
});
```

клиент

rpc-websockets=

пакет, позволяющий создавать WebSocket-сервер и клиента с поддержкой протокола JSON-RPC 2.0.



JSON-RPC =

протокол удаленного вызова процедур, использующий формат JSON для передачи сообщений. Используется для создания API в стиле RPC.



Создание удаленных процедур

```
const rpcWSS = require('rpc-websockets').Server
let server = new rpcWSS({port: 5000, host: 'localhost'});

server.register('sum', (params) =>{ return params[0] + params[1]}).public();
server.register('sub', (params) =>{ return params[0] - params[1]}).public();
server.register('mul', (params) =>{ return params[0] * params[1]}).public();
server.register('div', (params) =>{ return params[0] / params[1]}).public();
server.register('get', (params) =>{
    // сходить в БД за данными
    return {id:params[0], name:'Иванов И.И.', bday: '2000-12-02'};
}).public();
```

Создаем экземпляр класса Server, указываем порт и хост.

Метод `public()` помечает метод RPC как общедоступный

Метод `register(method, handler[namespace])` регистрирует метод RPC и возвращает объект `RPCMethod` для управления разрешениями метода. Первым параметром передаем название метода RPC, вторым – функцию, которая будет запущена после вызова метода (принимает только один параметр(массив, одиночное значение или др.)).

Вызов удаленных процедур

```
const rpcWSC = WebSocket = require('rpc-websockets').Client
let ws = new rpcWSC('ws://localhost:5000');

ws.on('open', ()=>{
  ws.call('sum', [5, 3]).then((r)=>{console.log('sum = ', r);});
  ws.call('sub', [5, 3]).then((r)=>{console.log('sub = ', r);});
  ws.call('mul', [5, 3]).then((r)=>{console.log('mul = ', r);});
  ws.call('div', [5, 3]).then((r)=>{console.log('div = ', r);});
  ws.call('get', [123], 3000).catch((e)=>{return e;}).then((r)=>{console.log('get_student = ', r);});
});
ws.on('error', (e)=>{console.log('error = ', e)});
```

Создаем экземпляр класса Client. В параметрах передаем адрес сервера.

С помощью **метода .call(method[, params[, timeout[, ws_options]]])** происходит вызов зарегистрированного метода RPC на сервере, метод возвращает Promise.

PS D:\NodeJS\samples\cwp_08> node .\08-07.js	PS D:\NodeJS\samples\cwp_08> node .\08-07a.js
[]	sum = 8 sub = -8 mul = 15 div = 0.06666666666666667 get_student = { id: 123, name: 'Иванов И.И.' , bday: '2000-12-02' }

Создание защищенных удаленных процедур

```
const rpcWSS = require('rpc-websockets').Server

let server = new rpcWSS({port: 5000, host: 'localhost'});

server.setAuth((l)=>{ return (l.login == 'smw' && l.password == '777') });

server.register('sum', (params) =>{ return params[0] + params[1]}).public();
server.register('sub', (params) =>{ return params[0] - params[1]}).public();
server.register('mul', (params) =>{ return params[0] * params[1]}).public();
server.register('div', (params) =>{ return params[0] / params[1]}).public();
server.register('get', (params) =>{return {id:params[0], name:'Иванов И.И.', bday: '2000-12-02'}}).public();

server.register('mod', (params) =>{ return params[0] % params[1]}).protected();
server.register('abs', (params) =>{ return params[0] >= 0?params[0]: -params[0]}).protected();
```

Метод `.protected()` помечает метод RPC как **зашитенный**.
Метод будет доступен только в том случае, если клиент успешно прошел аутентификацию.

Метод `.setAuth(handler[, namespace])` устанавливает определяемый пользователем метод аутентификации. Функция обработчика должна возвращать логическое значение.

Вызов защищенных процедур

```
const rpcWSC = WebSocket = require('rpc-websockets').Client;
let ws = new rpcWSC('ws://localhost:5000/');

ws.on('open', () => {
  ws.call('sum', [5, 3]).then((r) => { console.log('sum = ', r); });
  ws.call('sub', [5, 3]).then((r) => { console.log('sub = ', r); });
  ws.call('mul', [5, 3]).then((r) => { console.log('mul = ', r); });
  ws.call('div', [5, 3]).then((r) => { console.log('div = ', r); });
  ws.call('get', [123])
    .then((r) => { console.log('get_student = ', r); })
    .catch((e) => { return e; });

  ws.login({ login: 'smw', password: '777' })
    .then((login) => {
      if (login) {
        ws.call('mod', [33, 5]).then((r) => { console.log('mod = ', r); });
        ws.call('abs', [-33])
          .then((r) => { console.log('abs = ', r); })
          .catch((e) => { console.log('catch abs: ', e); });
      } else console.log('login error');
    })
    .catch((e) => { console.log('login failed: ', e.message); });

  //ws.close();      // асинхронность! вызывает ошибку
});

ws.on('error', (e) => console.log('error = ', e))
```

Метод `.login` используется для того, чтобы аутентифицироваться с другой стороны подключения. Метод возвращает Promise.

```
PS D:\NodeJS\samples\cwp_08> node .\08-08a.js
sum = 8
sub = -8
mul = 15
div = 0.06666666666666667
get_student = { id: 123, name: 'Иванов И.И.' , bday: '2000-12-02' }
login failed: authentication failed
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-08a.js
sum = 8
sub = -8
mul = 15
div = 0.06666666666666667
get_student = { id: 123, name: 'Иванов И.И.' , bday: '2000-12-02' }
mod = 3
abs = 33
```

Асинхронный параллельный вызов процедур

```
const async = require('async');
const rpcWSC = WebSocket = require('rpc-websockets').Client

let ws = new rpcWSC('ws://localhost:5000');
let h = (x=ws)=>async.parallel({
    sum: (cb)=>{ ws.call('sum', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
    sub: (cb)=>{ ws.call('sub', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
    mul: (cb)=>{ ws.call('mul', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
    div: (cb)=>{ ws.call('div', [5, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
    get: (cb)=>{ ws.call('get', [123],3000).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
    mod: (cb)=>{
        ws.login({login: 'smw', password: '777'})
        .then((login)=>{
            if (login) ws.call('mod', [33, 5]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
            else cb({message1:'login error'}, null);
        })
    },
    abs: (cb)=>{
        ws.login({login: 'smw', password: '777'})
        .then((login)=>{
            if (login) ws.call('abs', [-33]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
            else cb({message2:'login error'}, null);
        })
    }
},
(e, r)=>{
    if(e) console.log('e =',e);
    else console.log('r = ',r);
    ws.close();
}
);
ws.on('open', h);
```

Метод .close класса Client закрывает Websocket соединение

Метод **async.parallel()** используется для параллельного выполнения нескольких асинхронных операций.

```
PS D:\NodeJS\samples\cwp_08> node .\08-08b.js
r = {
  sum: 8,
  sub: -8,
  mul: 15,
  div: 0.06666666666666667,
  get: { id: 123, name: 'Иванов И.И.', bday: '2000-12-02' }
,
  mod: 3,
  abs: 33
}
PS D:\NodeJS\samples\cwp_08> []
```

Асинхронный последовательный вызов процедур

```
const async = require('async');
const rpcWSC = WebSocket = require('rpc-websockets').Client
// [((8+3)-2)*(-4)/2)%4]
let ws = new rpcWSC('ws://localhost:5000');
let h = ()=>async.waterfall([
  (cb)=>{ ws.call('sum', [8, 3]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{ ws.call('sub', [p, 2]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{ ws.call('mul', [p, -4]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{ ws.call('div', [p, 2]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));},
  (p,cb)=>{
    ws.login({login: 'smw', password: '777'})
    .then((login)=>{
      if (login) ws.call('mod', [p, 4]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
      else cb({message1: 'login error'}, null);
    })
  },
  (p,cb)=>{
    ws.login({login: 'smw', password: '777'})
    .then((login)=>{
      if (login) ws.call('abs', [-p]).catch((e)=>cb(e,null)).then((r)=>cb(null,r));
      else cb({message2: 'login error'}, null);
    })
  }
],
  (e, r)=>{
    if(e) console.log('e =',e);
    else console.log('r = ',r);
    ws.close();
  }
);
ws.on('open', h);
```

Выполнение нескольких асинхронных операций последовательно, когда каждая операция зависит от результатов предыдущих операций, осуществляется **методом `async.waterfall()`**.

```
PS D:\NodeJS\samples\cwp_08> node .\08-08c.js
r = 0.009615384615384616
PS D:\NodeJS\samples\cwp_08> []
```

События

```
const rpcWSS = require('rpc-websockets').Server  
  
let k = 0;
```

```
let server = new rpcWSS({port: 5000, host: 'localhost'});
```

```
server.event('event01');  
server.event('event02');  
server.event('event03');
```

```
setInterval(()=>server.emit('event01', {n:++k, x:1, y:2}),1000);  
setInterval(()=>server.emit('event02', {n:++k, s:'hello', d:'2019-09-09'}),2000);  
setInterval(()=>server.emit('event03' , {n:++k}), 3000);
```

сервер

Метод `server.event(name[,namespace])` создает новое событие и возвращает объект `RPCMethod`.

```
PS D:\NodeJS\samples\cwp_08> node .\08-09.js
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-09a.js  
event02: { n: 1, s: 'hello', d: '2019-09-09' }  
event01: { n: 2, x: 1, y: 2 }  
event03: { n: 3 }  
event01: { n: 4, x: 1, y: 2 }  
event02: { n: 5, s: 'hello', d: '2019-09-09' }  
event01: { n: 6, x: 1, y: 2 }  
event01: { n: 7, x: 1, y: 2 }
```

клиент

```
const rpcWSC = WebSocket = require('rpc-websockets').Client  
let ws = new rpcWSC('ws://localhost:5000');
```

```
ws.on('open', ()=>{
```

```
  ws.subscribe('event01');  
  ws.subscribe('event02');  
  ws.subscribe('event03');
```

```
  ws.on('event01',(p)=>{console.log('event01:', p)});  
  ws.on('event02', (p)=>console.log('event02:', p));  
  ws.on('event03', (p)=>console.log('event03:',p));
```

```
});
```

Метод `ws.subscribe(event)` позволяет подписаться на определенное событие.

Метод `ws.on(...)` позволяет указать обработчик события, на которое подписался клиент.

Уведомления

похоже на запрос без id, ответ на него не будет возвращен.

```
const rpcWSS = require('rpc-websockets').Server

let server = new rpcWSS({port: 5000, host: 'localhost'});

server.register('notify1', (params) =>{ console.log('notify1', params)}).public();
server.register('notify2', (params) =>{ console.log('notify2', params)}).public();
```

сервер

```
const rpcWSC = WebSocket = require('rpc-websockets').Client
let ws = new rpcWSC('ws://localhost:5000');

let k = 0;
ws.on('open', ()=>{
    setInterval(()=>ws.notify('notify1', {n:++k, x:1, y:2}),1000);
    setInterval(()=>ws.notify('notify2', {n:++k, x:1, y:2}),5000);
});
```

клиент

Метод `ws.notify()` отправляет на сервер уведомление JSON-RPC 2.0.
Можно к запросу добавить параметры.

```
PS D:\NodeJS\samples\cwp_08> node .\08-10.js
notify1 { n: 0, x: 1, y: 2 }
notify1 { n: 1, x: 1, y: 2 }
notify1 { n: 2, x: 1, y: 2 }
notify1 { n: 3, x: 1, y: 2 }
notify2 { n: 4, x: 1, y: 2 }
notify1 { n: 5, x: 1, y: 2 }
notify1 { n: 6, x: 1, y: 2 }
notify1 { n: 7, x: 1, y: 2 }
[]
```

```
PS D:\NodeJS\samples\cwp_08> node .\08-10a.js
```

ПРОГРАММИРОВАНИЕ СЕРВЕРНЫХ КРОССПЛАТФОРМЕННЫХ ПРИЛОЖЕНИЙ

TCP- И UDP-СЕРВЕР

TCP =
(Transmission Control
Protocol)

протокол транспортного
уровня для передачи
информации.

Основные характеристики TCP

- обмен осуществляется **пакетами**;
- **надежный**:
 - 1) установка **соединения**;
 - 2) **подтверждение** получения пакета;
 - 3) правильный **порядок** отправки;
 - 4) подсчет контрольных сумм для **проверки целостности** пакетов;
- **низкая** скорость передачи.

Установка и закрытие (полузакрытие) соединения



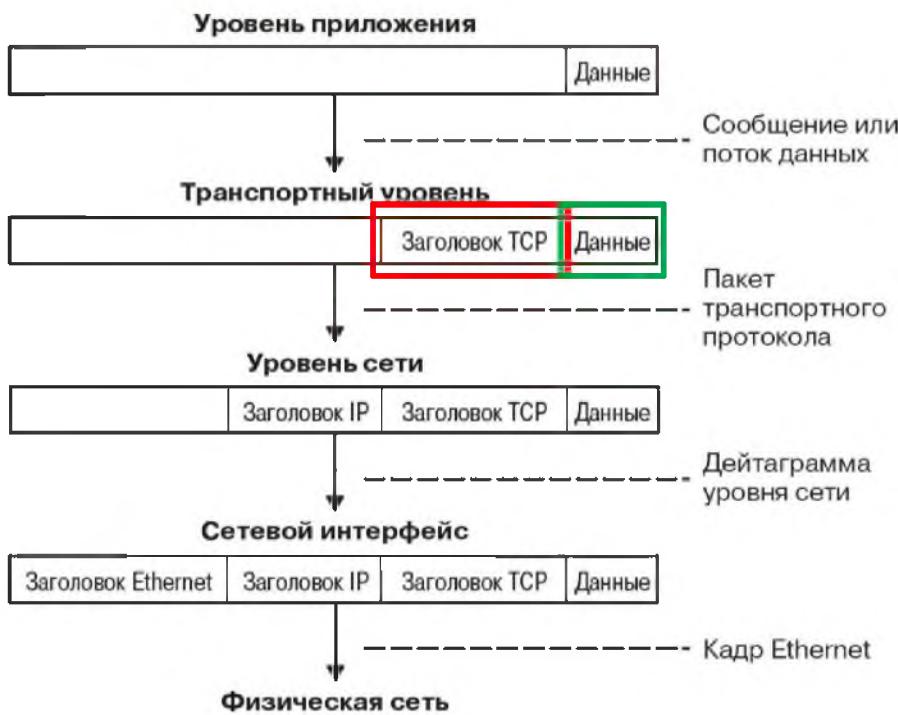
Пример установки соединения (handshake)

	TCP A		TCP B
1.	CLOSED		LISTEN
2.	SYN-SENT → <SEQ=100><CTL=SYN>		→ SYN-RECEIVED
3.	ESTABLISHED ← <SEQ=300><ACK=101><CTL=SYN,ACK>	← SYN-RECEIVED	
4.	ESTABLISHED → <SEQ=101><ACK=301><CTL=ACK>		→ ESTABLISHED
5.	ESTABLISHED ← <SEQ=301><ACK=101><CTL=ACK>		← ESTABLISHED

Установка соединения и обмен данными

Time	192.168.1.2	Comment
0.000	SYN	Seq = 0 Ack = 94856096
0.047	SYN, ACK	Seq = 0 Ack = 1
0.047	ACK	Seq = 1 Ack = 1
0.047	PSH, ACK - Len: 725	Seq = 1 Ack = 1
0.097	ACK	Seq = 1 Ack = 726
0.100	ACK - Len: 1448	Seq = 1 Ack = 726
0.100	ACK	Seq = 726 Ack = 1449
0.100	ACK - Len: 1448	Seq = 1449 Ack = 726
0.100	ACK	Seq = 726 Ack = 2897
0.100	ACK - Len: 1448	Seq = 2897 Ack = 726
0.100	ACK	Seq = 726 Ack = 4345
0.150	ACK - Len: 1448	Seq = 4345 Ack = 726
0.150	ACK	Seq = 726 Ack = 5793
0.152	ACK - Len: 1448	Seq = 5793 Ack = 726
0.152	ACK	Seq = 726 Ack = 7241
0.152	ACK - Len: 1448	Seq = 7241 Ack = 726
0.152	ACK	Seq = 726 Ack = 8689

Структура пакета



net.Server

=

класс, который используется
для создания сервера TCP или
IPC. Наследует EventEmitter.

Способы создания экземпляров net.Server

- 1) С помощью **метода `net.createServer([options][, connectionListener])`**.
- 2) Путем явного создания **экземпляра класса `new net.Server([options][, connectionListener])`**.

Некоторые из опций, которые можно настроить:

- ***allowHalfOpen*** – если установлено значение `false`, то использование полузакрытых сокетов не будет разрешено. По умолчанию: `false`.
- ***keepAlive*** – если установлено значение `true`, активизируется функция поддержания активности на сокете сразу после получения нового входящего соединения. По умолчанию: `false`.

Класс net.Server (события)

- **close** генерируется при закрытии сервера. Если соединения существуют, это событие не генерируется до тех пор, пока не будут завершены все соединения.
- **connection** генерируется при установке нового соединения. В обработчик передается `socket` (экземпляром `net.Socket`).
- **error** генерируется при возникновении ошибки. В отличие от `net.Socket`, событие `close` не будет сгенерировано непосредственно после этого события, если только `server.close()` не будет вызван вручную. С
- **listening** генерируется после вызова `server.listen()`.

Класс net.Server (методы)

- `server.address()` возвращает адрес, имя семейства адресов и порт сервера.
- `server.close([callback])` запрещает серверу принимать новые соединения и сохраняет существующие соединения. Этот метод является асинхронным, сервер окончательно закрывается, когда все соединения завершены, и сервер генерирует событие `close`.
- `server.getConnections(callback)` позволяет асинхронно получить количество одновременных подключений к серверу. Обратный вызов принимает два аргумента `err` и `count`.
- `server.listen([port[, host[, backlog]][], callback])` или `server.listen(options[, callback])` запускает сервер. Этот метод является асинхронным.
- `server.ref()/unref()` отмечает события, генерируемые сервером, второстепенной задачей.

net.Socket =

класс, который является **абстракцией** **TCP-сокета** или потоковой конечной точки IPC. Наследует EventEmitter, а также stream.Duplex.

Способы использования net.Socket

- 1) Можно создать net.Socket и использовать его непосредственно для взаимодействия с сервером:
 - `net.createConnection(options[, connectListener])` (или `net.connect(port[, host][, connectListener])`) – фабричная функция, которая создает новый net.Socket, немедленно инициирует соединение с помощью `socket.connect()`, а затем возвращает net.Socket. Когда соединение установлено, в возвращаемом сокете будет сгенерировано событие `connect`. Последний параметр `connectListener`, если он указан, будет добавлен в качестве слушателя для события `connect` один раз.
 - `new net.Socket([options])` – создает новый объект сокета. Вновь созданный сокет может быть либо TCP-сокетом, либо потоковой конечной точкой IPC, в зависимости от того, к чему он подключается. Затем его необходимо подсоединить к серверу с помощью `socket.connect(port[, host][, connectListener])`.
- 2) Он также может быть создан Node.js и передан при получении соединения. Например, он передается слушателям события `connection`, генерируемого `net.Server`, поэтому можно использовать его для взаимодействия с клиентом.

Класс net.Socket (события)

- **close** генерируется после полного закрытия сокета.
- **connect** генерируется при успешном установлении соединения через сокет.
- **data** генерируется при получении данных. Данные аргумента будут буфером или строкой.
- **end** генерируется, когда другой конец сокета сигнализирует об окончании передачи, тем самым заканчивая доступную для чтения сторону сокета, т.е. при полузакрытии. По умолчанию (опция `allowHalfOpen = false`) сокет отправит обратно пакет конца передачи и уничтожит свой файловый дескриптор.

Класс net.Socket (события)

- **error** генерируется при возникновении ошибки. Сразу после генерируется событие close.
- **ready** генерируется, когда сокет готов к использованию. Запускается сразу после connect.
- **timeout** генерируется, если время ожидания сокета истекло из-за бездействия. Только уведомляет, что сокет был бездействующим. Пользователь должен вручную закрыть соединение.

Класс net.Socket (свойства)

- `socket.bytesRead` – количество полученных байтов;
- `socket.bytesWritten` – количество отправленных байтов;
- `socket.connecting` – если true, то `socket.connect()` был вызван и еще не завершен. Он будет оставаться истинным до тех пор, пока сокет не будет подключен, затем он будет установлен в значение false и будет сгенерировано событие `connect`;
- `socket.destroyed` – равняется true после вызова метода `destroy()`;
- `socket.pending` – true, если сокет еще не подключен, либо `connect()` еще не был вызван, либо он все еще находится в процессе подключения;
- `socket.readyState` – представляет состояние соединения в виде строки (`opening`, `open`, `readOnly`, `writeOnly`).

Класс net.Socket (свойства)

- `socket.localAddress` – строковое представление локального IP-адреса, к которому подключается удаленный клиент;
- `socket.localPort` – числовое представление локального порта;
- `socket.localFamily` – строковое представление семейства локального IP-адреса;
- `socket.remoteAddress` – строковое представление удаленного IP-адреса;
- `socket.remotePort` – числовое представление удаленного порта;
- `socket.remoteFamily` – строковое представление семейства удаленного IP-адреса.
- `socket.timeout` – тайм-аут сокета в мс, установленный `socket.setTimeout()` (по умолчанию `undefined`).

Класс net.Socket (методы)

- `socket.address()` возвращает связанный адрес, имя семейства адресов и порт сокета.
- `socket.connect(port[, host][, connectListener])` устанавливает соединение, а потом генерирует событие `connect`. Если возникает проблема с подключением, вместо события `connect` будет сгенерировано событие `error`. Последний параметр `connectListener`, если он указан, будет добавлен в качестве слушателя для события `connect` один раз. Этот метод является асинхронным.
- `socket.destroy([error])` гарантирует, что в этом сокете больше не будет операций ввода-вывода. Уничтожает поток и закрывает соединение.
- `socket.end([data[, encoding]][, callback])` полузакрывает сокет. т. е. отправляет пакет FIN. Возможно, что сервер будет продолжать отправлять данные (если при его создании параметр `allowHalfOpen` установлен в `true`).

Класс net.Socket (методы)

- `socket.unref()` позволяет процессу завершиться, если это единственный активный сокет в системе событий. А метод `socket.ref()` не позволяет процессу завершиться, если это единственный оставшийся сокет.
- `socket.setEncoding([encoding])` устанавливает кодировку для сокета (для Readable Stream).
- `socket.setKeepAlive([enable][, initialDelay])` включает/отключает функции проверки активности и, при необходимости, устанавливает начальную задержку.
- `socket.setTimeout(timeout[, callback])` устанавливает для сокета тайм-аут (в мс) бездействия сокета. По умолчанию net.Socket не имеет тайм-аута (undefined).
- `socket.write(data[, encoding][, callback])` отправляет данные на сокет. Второй параметр указывает кодировку. Необязательный callback будет выполнен, когда данные будут окончательно записаны, что может произойти не сразу.

Простейший TCP-сервер

```
const net = require('net');

let HOST = '0.0.0.0';
let PORT = 40000;

net.createServer((sock)=>{

    console.log('Server CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);

    sock.on('data', (data)=>{
        console.log('Server DATA:', sock.remoteAddress + ': ' + data);
        sock.write('Сервер получил:' + data );
    });

    sock.on('close', ( )=>{console.log('Server CLOSED: ', sock.remoteAddress + ' ' + sock.remotePort);});

}).listen(PORT, HOST);

console.log('TCP-сервер ' + HOST + ':' + PORT);
```

Простейший TCP-клиент

```
var net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

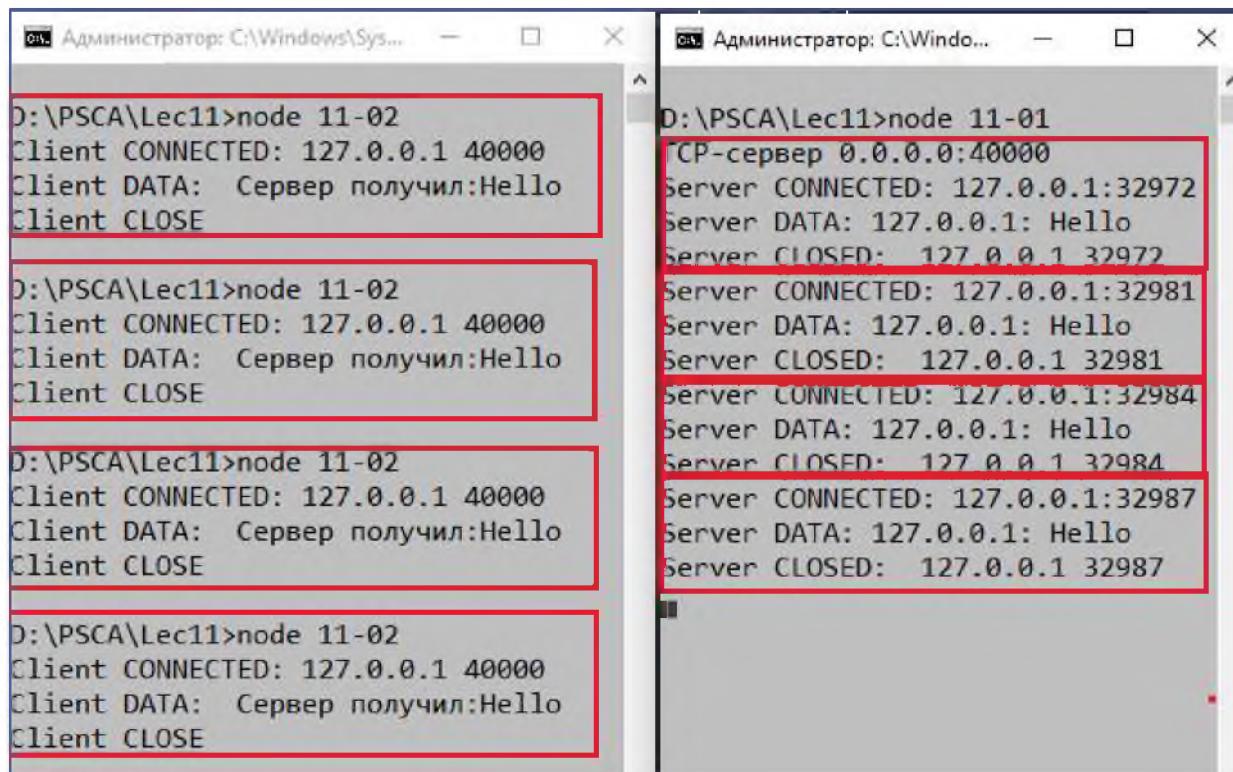
var client = new net.Socket();
client.connect(PORT, HOST, () => {
  console.log('Client CONNECTED: ', client.remoteAddress + ' ' + client.remotePort);
  client.write('Hello');
})

client.on('data', (data) => {
  console.log('Client DATA: ' + data.toString());
  client.destroy();
});

client.on('close', () => { console.log('Client CLOSE'); });

client.on('error', (e) => { console.log('Client ERROR'); });
```

Результат работы



```
D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

D:\PSCA\Lec11>node 11-02
Client CONNECTED: 127.0.0.1 40000
Client DATA: Сервер получил:Hello
Client CLOSE

D:\PSCA\Lec11>node 11-01
CP-сервер 0.0.0.0:40000
Server CONNECTED: 127.0.0.1:32972
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32972

Server CONNECTED: 127.0.0.1:32981
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32981

Server CONNECTED: 127.0.0.1:32984
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32984

Server CONNECTED: 127.0.0.1:32987
Server DATA: 127.0.0.1: Hello
Server CLOSED: 127.0.0.1 32987
```

Отправка буфера (TCP-сервер)

```
const net = require('net');

let HOST = '0.0.0.0';
let PORT = 40000;

let sum = 0;
-----

let server = net.createServer();
server.on('connection', (sock)=>{

    console.log('Server CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);

    sock.on('data', (data)=>{
        console.log('Server DATA: ', data, sum);
        sum += data.readInt32LE();
    });

    let buf = Buffer.alloc(4);
    setInterval(()=>{buf.writeInt32LE(sum,0); ----- sock.write(buf)}, 3000);

    sock.on('close', (data)=>{console.log('Server CLOSED: ', sock.remoteAddress + ' ' + sock.remotePort);});

    sock.on('error', (e)=>{console.log('Server error: ', sock.remoteAddress + ' ' + sock.remotePort);});

})
server.on('listening', ()=>{ console.log('TCP-server listening on ' + HOST + ':' + PORT); });
server.on('error', (e)=>{ console.log("TCP-server error: ", e); });
server.listen(PORT, HOST);
```

Отправка буфера (TCP-клиент)

```
var net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

var client = new net.Socket();
var buf = new Buffer.alloc(4);
let timerId = null;
client.connect(PORT, HOST, () => {
  console.log('Client CONNECTED: ', client.remoteAddress + ' ' + client.remotePort);

  let k = 0;
  timerId = setInterval(() => {
    buf.writeInt32LE(k++, 0);
    client.write(buf);
  }, 1000);

  setTimeout(() => {
    clearInterval(timerId);
    client.end();
  }, 30000);
}

client.on('data', (data) => { console.log('Client DATA: ' + data.readInt32LE()); });

client.on('close', () => { console.log('Client CLOSE'); });

client.on('error', (e) => { console.log('Client ERROR', e); });
```

Результат работы

<pre>D:\PSCA\Lec11>node 11-03a Client CONNECTED: 127.0.0.1 40000 Client DATA: 1 Client DATA: 10 Client DATA: 28 Client DATA: 55 Client DATA: 91 Client DATA: 136 Client DATA: 190 Client DATA: 253 Client DATA: 325 Client CLOSE D:\PSCA\Lec11> D:\PSCA\Lec11>■</pre>	<pre>D:\PSCA\Lec11>node 11-03 TCP-server 0.0.0.0:40000 Server CONNECTED: 127.0.0.1:48050 Server DATA: <Buffer 00 00 00 00> 0 Server DATA: <Buffer 01 00 00 00> 0 Server DATA: <Buffer 02 00 00 00> 1 Server DATA: <Buffer 03 00 00 00> 3 Server DATA: <Buffer 04 00 00 00> 6 Server DATA: <Buffer 05 00 00 00> 10 Server DATA: <Buffer 06 00 00 00> 15 Server DATA: <Buffer 07 00 00 00> 21 Server DATA: <Buffer 08 00 00 00> 28 Server DATA: <Buffer 09 00 00 00> 36 Server DATA: <Buffer 0a 00 00 00> 45 Server DATA: <Buffer 0b 00 00 00> 55 Server DATA: <Buffer 0c 00 00 00> 66 Server DATA: <Buffer 0d 00 00 00> 78 Server DATA: <Buffer 0e 00 00 00> 91 Server DATA: <Buffer 0f 00 00 00> 105 Server DATA: <Buffer 10 00 00 00> 120 Server DATA: <Buffer 11 00 00 00> 136 Server DATA: <Buffer 12 00 00 00> 153 Server DATA: <Buffer 13 00 00 00> 171 Server DATA: <Buffer 14 00 00 00> 190 Server DATA: <Buffer 15 00 00 00> 210 Server DATA: <Buffer 16 00 00 00> 231 Server DATA: <Buffer 17 00 00 00> 253 Server DATA: <Buffer 18 00 00 00> 276 Server DATA: <Buffer 19 00 00 00> 300</pre>
--	---

TCP-сервер (использование pipe)

```
const net = require('net');

let HOST = '0.0.0.0';
let PORT = 40000;

let sum = 0;

let ws = require('fs').createWriteStream('./Files/File04.data', {flags:'a'});

let server = net.createServer();
server.on('connection', (sock)=>{

    console.log('Server CONNECTED: ' + sock.remoteAddress +':'+ sock.remotePort);

    sock.pipe(ws);

    sock.on('close', (data)=>{console.log('Server CLOSED: ', sock.remoteAddress +' '+ sock.remotePort);});
    sock.on('error', (e)=>{console.log('Server error: ', sock.remoteAddress +' '+ sock.remotePort);});
})

server.on('listening', ()=>{ console.log('TCP-server ', HOST +':'+ PORT);} );
server.on('error', (e)=>{ console.log('TCP-server error ', e);} );
server.listen(PORT, HOST);
```

net.Socket – это **дуплексный**
ПОТОК

TCP-клиент (буфер)

```
var net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

var client = new net.Socket();
var buf = new Buffer.alloc(4);
let timerId = null;
client.connect(PORT, HOST, () => {
  console.log('Client CONNECTED: ', client.remoteAddress + ' ' + client.remotePort);

  let k = 0;
  timerId = setInterval(() => { buf.writeInt32LE(k++, 0); client.write(buf); }, 1000);
  setTimeout(() => { clearInterval(timerId); client.end(); }, 30000);
}

client.on('data', (data) => { console.log('Client DATA: ' + data.readInt32LE()); });

client.on('close', () => { console.log('Client CLOSE'); });

client.on('error', (e) => { console.log('Client ERROR', e); });
```

Offset	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000:	00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
00000010:	04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00
00000020:	08 00 00 00 09 00 00 00 0A 00 00 00 0B 00 00 00
00000030:	0C 00 00 00 0D 00 00 00 0E 00 00 00 0F 00 00 00
00000040:	10 00 00 00 11 00 00 00 12 00 00 00 13 00 00 00
00000050:	14 00 00 00 15 00 00 00 16 00 00 00 17 00 00 00
00000060:	18 00 00 00 19 00 00 00 1A 00 00 00 1B 00 00 00
00000070:	1C 00 00 00

TCP-клиент (использование pipe)

```
const net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

let rs = require('fs').createReadStream('./Files/File04.data');

let client = new net.Socket();

client.connect(PORT, HOST, ()=>{

    console.log('Client CONNECTED:', client.remoteAddress + ' ' + client.remotePort);

    rs.pipe(client);
});

client.on('data', (data)=>{console.log('Client DATA: ', data.readInt32LE())});

client.on('close', ()=>{console.log('Client CLOSE')});

client.on('error', (e)=>{console.log('Client ERROR', e)});
```

Offset:	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000:	00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
00000010:	04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00
00000020:	08 00 00 00 09 00 00 00 0A 00 00 00 0B 00 00 00
00000030:	0C 00 00 00 0D 00 00 00 0E 00 00 00 0F 00 00 00
00000040:	10 00 00 00 11 00 00 00 12 00 00 00 13 00 00 00
00000050:	14 00 00 00 15 00 00 00 16 00 00 00 17 00 00 00
00000060:	18 00 00 00 19 00 00 00 1A 00 00 00 1B 00 00 00
00000070:	1C 00 00 00

Сервер, прослушивающий два порта

```
const net = require('net');

let HOST = '127.0.0.1';
let PORT = 40000;

let rs = require('fs').createReadStream('./Files/File04.data');

let client = new net.Socket();

client.connect(PORT, HOST, ()=>{
    console.log('Client CONNECTED:', client.remoteAddress + ' ' + client.remotePort);
    rs.pipe(client);
});

client.on('data', (data)=>{console.log('Client DATA: ', data.readInt32LE())});

client.on('close', ()=>{console.log('Client CLOSE')});

client.on('error', (e)=>{console.log('Client ERROR', e);});
```

TCP-клиент (номер порта берется из аргументов командной строки)

```
const net = require('net');

const HOST = '127.0.0.1';
const PORT = process.argv[2] ? process.argv[2] : 40000;

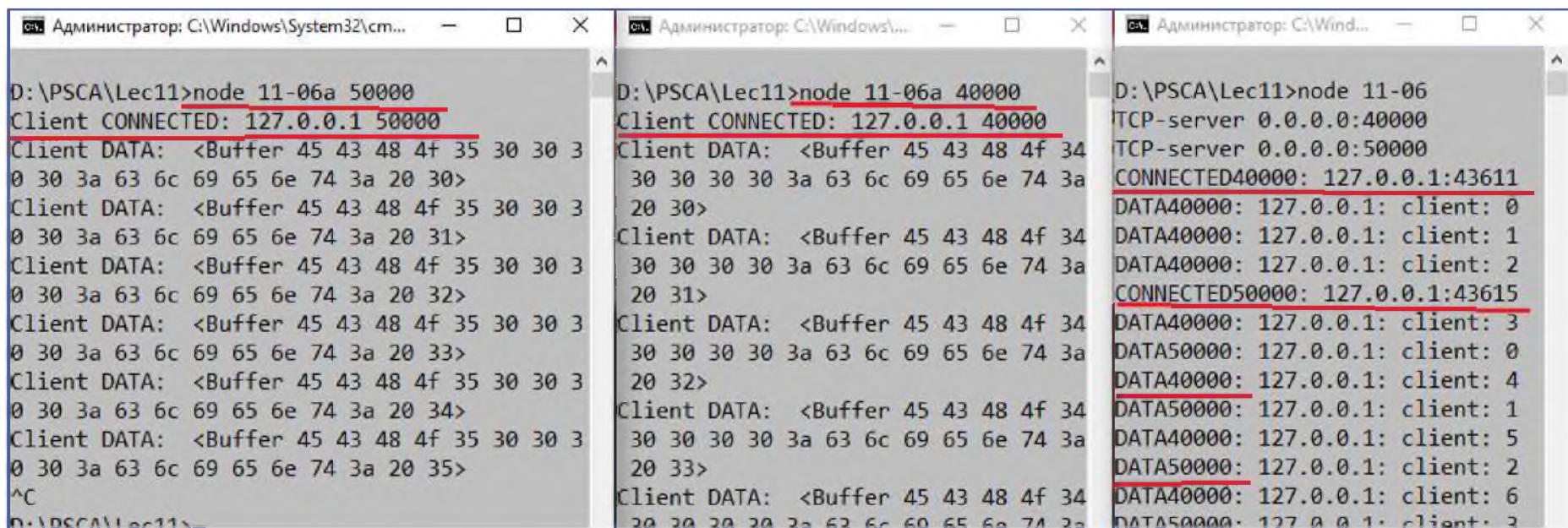
let client = new net.Socket();

client.connect(PORT, HOST, ()=>{
    console.log('Client CONNECTED:', client.remoteAddress + ' ' + client.remotePort);

    let k = 0;
    timerId = setInterval(()=>{client.write(`client: ${k++}`)}, 1000);
    setTimeout(()=>{clearInterval(timerId); client.end();}, 30000);
});

client.on('data', (data)=>{console.log('Client DATA: ', data)})
    .on('close', ()=>{console.log('Client CLOSE')})
    .on('error', (e)=>{console.log('Client ERROR', e)});
```

Результат работы



The image shows three separate terminal windows, each with a blue title bar and a white body. Each window is titled 'Администратор: C:\Windows\System32\cmd...' and has a close button (X) in the top right corner. The windows are arranged horizontally and are slightly overlapping.

The leftmost window contains the following text:

```
D:\PSCA\Lec11>node 11-06a 50000
Client CONNECTED: 127.0.0.1 50000
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 30>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 31>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 32>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 33>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 34>
Client DATA: <Buffer 45 43 48 4f 35 30 30 3
0 30 3a 63 6c 69 65 6e 74 3a 20 35>
^C
D:\PSCA\Lec11>
```

The middle window contains the following text:

```
D:\PSCA\Lec11>node 11-06a 40000
Client CONNECTED: 127.0.0.1 40000
Client DATA: <Buffer 45 43 48 4f 34 30 30 30 3
0 3a 63 6c 69 65 6e 74 3a 20 30>
Client DATA: <Buffer 45 43 48 4f 34 30 30 30 3
0 3a 63 6c 69 65 6e 74 3a 20 31>
Client DATA: <Buffer 45 43 48 4f 34 30 30 30 3
0 3a 63 6c 69 65 6e 74 3a 20 32>
Client DATA: <Buffer 45 43 48 4f 34 30 30 30 3
0 3a 63 6c 69 65 6e 74 3a 20 33>
Client DATA: <Buffer 45 43 48 4f 34 30 30 30 3
0 3a 63 6c 69 65 6e 74 3a 20 34>
Client DATA: <Buffer 45 43 48 4f 34 30 30 30 3
0 3a 63 6c 69 65 6e 74 3a 20 35>
```

The rightmost window contains the following text:

```
D:\PSCA\Lec11>node 11-06
TCP-server 0.0.0.0:40000
TCP-server 0.0.0.0:50000
CONNECTED40000: 127.0.0.1:43611
DATA40000: 127.0.0.1: client: 0
DATA40000: 127.0.0.1: client: 1
DATA40000: 127.0.0.1: client: 2
CONNECTED50000: 127.0.0.1:43615
DATA40000: 127.0.0.1: client: 3
DATA50000: 127.0.0.1: client: 0
DATA40000: 127.0.0.1: client: 4
DATA50000: 127.0.0.1: client: 1
DATA40000: 127.0.0.1: client: 5
DATA50000: 127.0.0.1: client: 2
DATA40000: 127.0.0.1: client: 6
DATA50000: 127.0.0.1: client: 3
```

UDP =
(User Datagram Protocol)

протокол транспортного
уровня для передачи
информации.

Основные характеристики UDP

- обмен осуществляется **дейтаграммами**;
- **ненадежный**:
 - 1) не устанавливает соединение;
 - 2) упорядоченность не соблюдается;
 - 3) доставка не гарантируется;
- более **высокая** скорость передачи.

Структура UDP-датаграммы



dgram.Socket =

инкапсулирует **функциональность дейтаграммы**. Наследует EventEmitter.

Создать экземпляр `dgram.Socket` можно с помощью метода `dgram.createSocket(type[, callback])`.

! Ключевое слово `new` не должно использоваться для создания экземпляров `dgram.Socket`.

Класс dgram.Socket (события)

- **close** генерируется после закрытия сокета с помощью функции `close()`. После срабатывания в этом сокете не будут создаваться новые события `message`.
- **connect** генерируется после того, как сокет связан с удаленным адресом в результате успешного вызова `connect()`.
- **error** генерируется всякий раз, когда возникает какая-либо ошибка. В функцию обработчика событий передается один объект `Error`.
- **listening** генерируется, когда `dgram.Socket` становится адресуемым и может получать данные. Это происходит либо явно с помощью `socket.bind()`, либо неявно при первой отправке данных с помощью `socket.send()`.
- **message** генерируется, когда в сокете доступна новая дейтаграмма. В функцию обработчика событий передаются два аргумента: `msg` (само сообщение) и `rinfo` (информация об удаленном адресе, отправителе сообщения).

Класс dgram.Socket (методы)

- `socket.address()` возвращает объект, содержащий информацию об адресе сокета (address, family и port).
- `socket.bind([port][, address][, callback])` или `socket.bind(options[, callback])` прослушивает дейтаграммы на указанном порте и адресе. Если порт не указан, ОС попытается выполнить привязку к произвольному порту. Если адрес не указан, ОС попытается прослушивать все адреса. После завершения привязки генерируется событие `listening` и вызывается callback.
- `socket.close([callback])` закрывает сокет и перестает прослушивать данные на нем. Если предоставлен callback, то он добавляется в качестве слушателя для события `close`.
- `socket.connect(port[, address][, callback])` связывает сокет с удаленным адресом и портом. Каждое сообщение автоматически отправляется этому адресату. Кроме того, сокет будет получать сообщения только от этого удаленного узла. Как только связь установлена, генерируется событие `connect` и вызывается callback.

Класс dgram.Socket (методы)

- `socket.disconnect()` отвязывает подключенный сокет от его удаленного адреса. Этот метод является синхронным.
- `socket.unref()` позволяет процессу завершиться, если это единственный активный сокет в системе событий. А метод `socket.ref()` не позволяет процессу завершиться, если это единственный оставшийся сокет.
- `socket.remoteAddress()` возвращает объект, содержащий адрес, семейство и порт удаленной конечной точки.
- `socket.send(msg[, offset, length][, port][, address][, callback])` отправляет данные на сокет. Для несвязанных сокетов необходимо указать порт назначения и адрес, иначе будет использоваться связанная с сокетом удаленная конечная точка. Единственный способ узнать наверняка, что дейтаграмма была отправлена, — это использовать callback.

Простейший UDP-сервер

```
// https://gist.github.com/sid24rane/6e6698e93360f2694e310dd347a2e2eb

const udp = require('dgram');
const PORT = 3000;

let server = udp.createSocket('udp4');

server.on('error',(err)=>{console.log('Ошибка: ' + err); server.close();});

server.on('message',(msg,info)=>{

    console.log('Server: от клиента получено ' + msg.toString());
    console.log('Server: получено %d байтов от %s:%d\n',msg.length, info.address, info.port);

    server.send(msg, info.port, info.address, (err)=>{
        if(err){server.close();}
        else {console.log('Server: данные отправлены клиенту');}
    });
});

server.on('listening',()=>{
    console.log('Server: слушает порт ' + server.address().port);
    console.log('Server: ip сервера ' + server.address().address);
    console.log('Server: семейство(IP4/IP6) ' + server.address().family);
});

server.on('close', ()=>{console.log('Server: сокет закрыт');});

server.bind(PORT);

//setTimeout(()=>{server.close();},8000);
```

Простейший UDP-клиент

```
const buffer = require('buffer');
const udp = require('dgram');
const client = udp.createSocket('udp4');
const PORT = 3000;

client.on('message',(msg,info)=>{
  console.log('Client: от сервера получено ' + msg.toString());
  console.log('Client: получено %d байтов от %s:%d\n',msg.length, info.address, info.port);
});

let data = Buffer.from('Client: сообщение 01');
client.send(data,PORT,'localhost',(err)=>{
  if(err) client.close();
  else console.log('Client: Сообщение отправлено серверу');
});

let data1 = Buffer.from('Привет ');
let data2 = Buffer.from('Мир');

client.send([data1,data2],PORT,'localhost',(err)=>{
  if(err)client.close();
  else console.log('Client: Сообщение отправлено серверу');
});

});
```

```
const dgram = require('dgram');

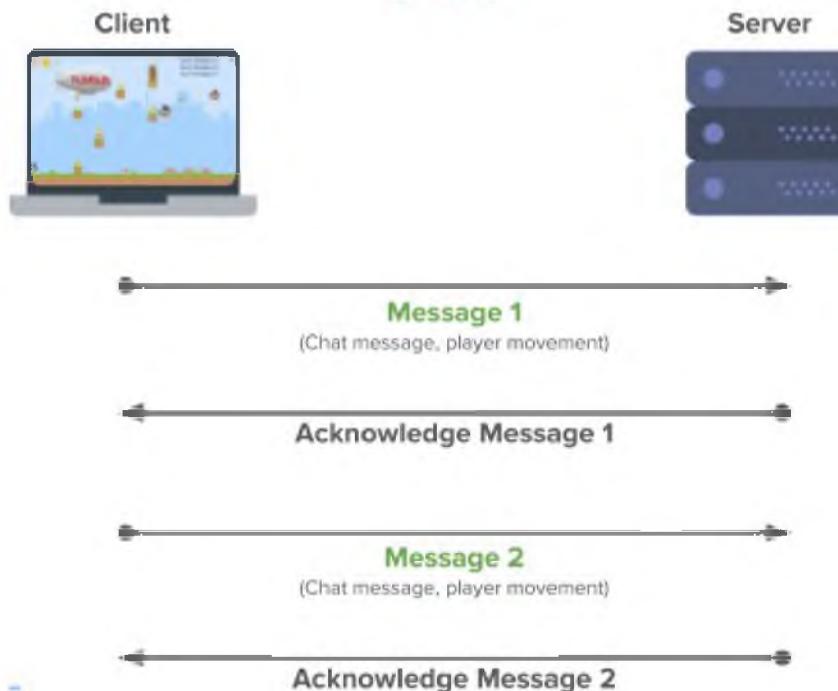
const client = dgram.createSocket('udp4');

const message = Buffer.from('Client: сообщение 01');
client.connect(3000, 'localhost', (err) => {
  client.send(message, (err) => {
    client.close();
  });
});
```

Критерии сравнения	TCP	UDP
Соединение	Требуется установленное соединение для передачи данных	Протокол без соединения
Гарантия доставки	Может гарантировать доставку данных получателю	Не гарантирует доставку данных получателю
Повторная передача данных	Повторная передача нескольких пакетов случае потери одного из них	Отсутствие повторной передачи потерянных пакетов
Проверка ошибок	Полная проверка ошибок	Базовый механизм проверки ошибок. Использует вышестоящие протоколы для проверки целостности
Упорядоченность	Гарантирует правильный порядок	Порядок не соблюдается
Скорость	Низкая	Высокая
Сфера применения	Используется для передачи сообщений электронной почты, HTML-страниц браузеров	Видеоконференции, потоковое вещание, DNS, VoIP, IPTV

TCP vs UDP

TCP



UDP

