

# 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.

*То, 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}/`);
});
```

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

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

`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);
  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);
  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, разрешенного 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 () { // socket 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



```
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')

  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, {
    'Content-Type': 'application/json; charset=utf-8', // кода статуса + заголовки
    'Cache-Control': 'no-cache'
  });

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

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);

```

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

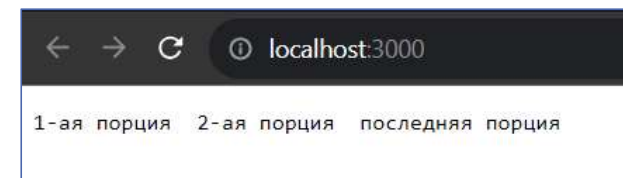
## Запрос и ответ (пример)

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



# 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/118.0.0.0 Safari/537.36
request header accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
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/118.0.0.0 Safari/537.36
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.

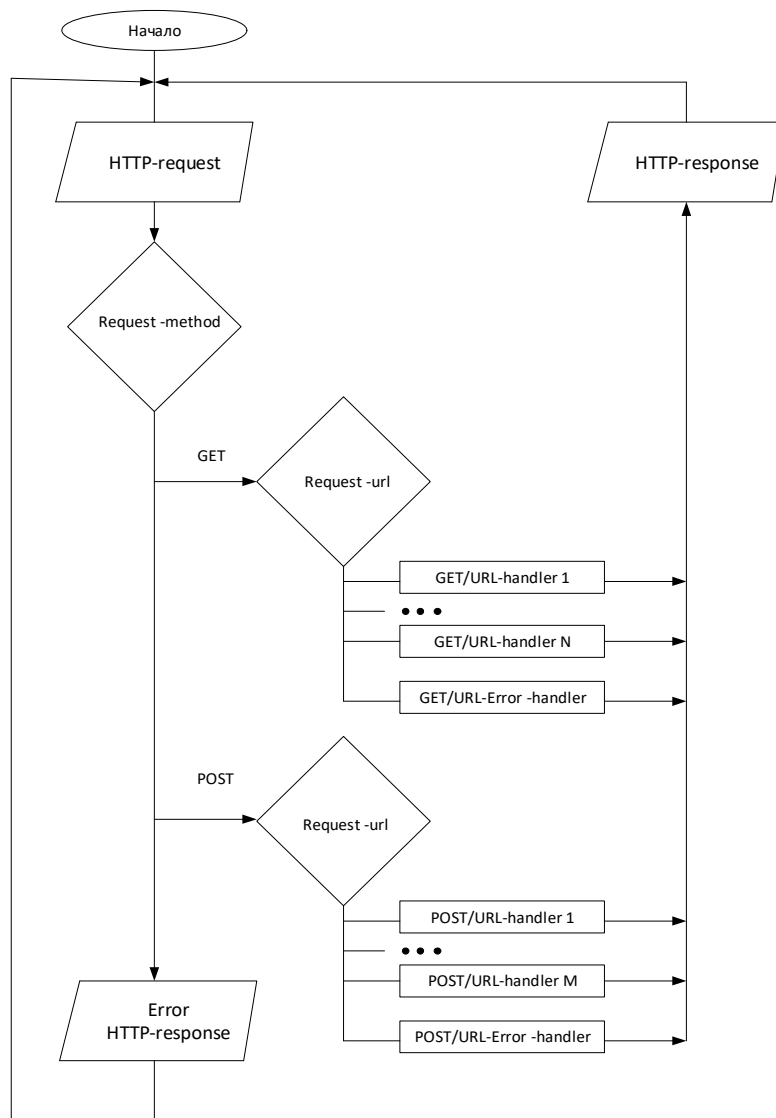
**localPort** – числовое представление **локального** порта.

**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}/`);
});

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

# Демонстрация

GET <http://localhost:3000/index.html>

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

Key	Value	Description
-----	-------	-------------

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

Pretty Raw Preview Visualize JSON

1 GET: </index.html>

GET <http://localhost:3000/index>

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

Key	Value	Description
-----	-------	-------------

Body Cookies Headers (5) Test Results Status: 404 Not Found

Pretty Raw Preview Visualize JSON

1  
2 "error": "GET: </index>, HTTP status 404"  
3

PATCH <http://localhost:3000/qwerty>

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

Query Params

Key	Value	Description
-----	-------	-------------

Body Cookies Headers (5) Test Results Status: 405 Method Not Allowed

Pretty Raw Preview Visualize JSON

1  
2 "error": "PATCH: </qwerty>, HTTP status 405"  
3

```
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.

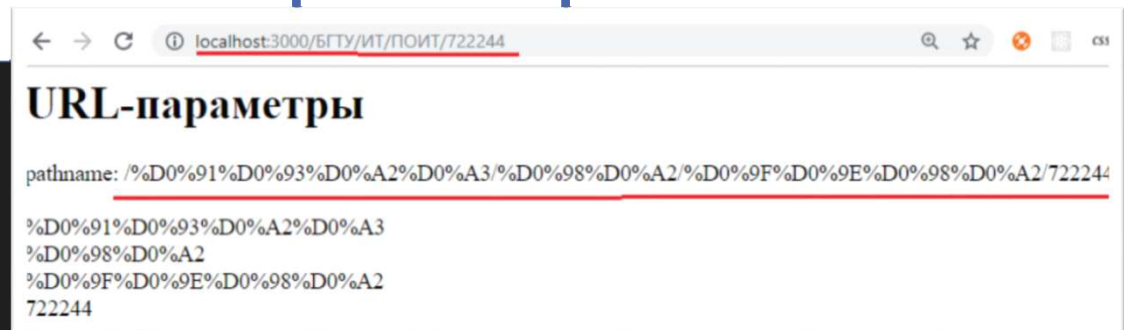


# Обработка path-параметров

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



Для того, чтобы их извлечь, необходимо:

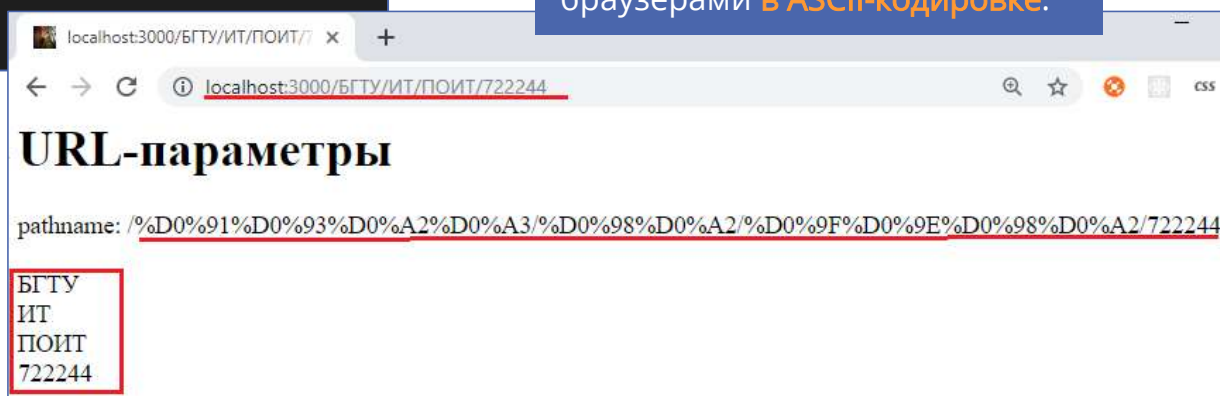
1. Распарсить url.
2. Достать свойство pathname.
3. Разбить его на элементы с помощью разделителя /, используя метод split().



# 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);
}
```

The screenshot shows a web browser with a POST request to `http://localhost:3000/`. The 'Body' tab is selected, displaying a Lorem ipsum text. Below the browser, a terminal window shows the server output, with the request body data and end event highlighted in red.

```
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()`.

```

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)

```

## Обработка тела (x-www-form-urlencoded)

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

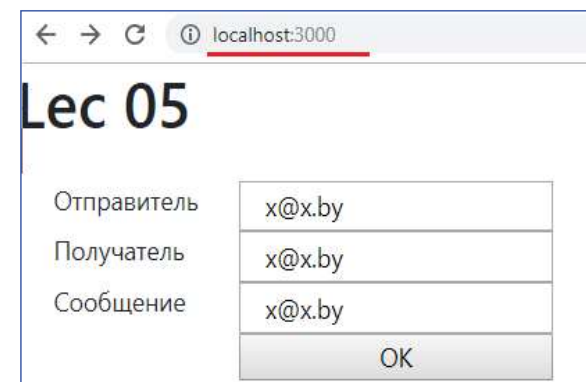
# Обработка тела (x-www-form-urlencoded)

```
<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=" " />
    </div>
    <div class="row">
      <label class="col-2">Получатель</label> <input class="col-3" name="sender" placeholder=" " />
    </div>
    <div class="row">
      <label class="col-2">Сообщение</label> <input class="col-3" name="message" placeholder=" " />
    </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

Получатель

Сообщение

## URL-параметры

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

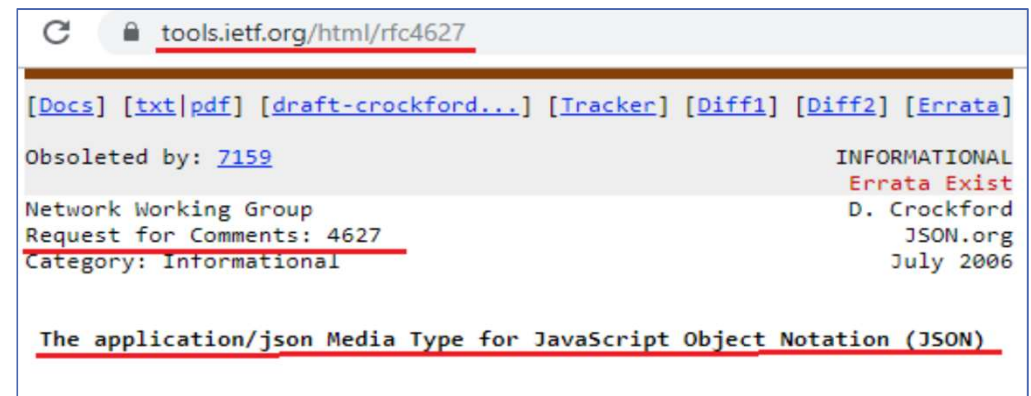
Описаны в RFC 2045, RFC 2046, RFC 4288, RFC 4289 и RFC 4855, зарегистрированы IANA.

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>



# Работа с JSON

```
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: ' + e); })
  .on('request', handler)
```

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

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

```
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**.

```
{
  "_comment": "Так можно сделать комментарий",
  "x": 1,
  "y": 1.01,
  "s": "Строка",
  "array": [
    "a",
    "b",
    "c",
    "d"
  ],
  "obj": {
    "surname": "Иванов",
    "name": "Иван"
  }
}
```

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

```
const json = require('./f06-07.json');
console.log(json);
```

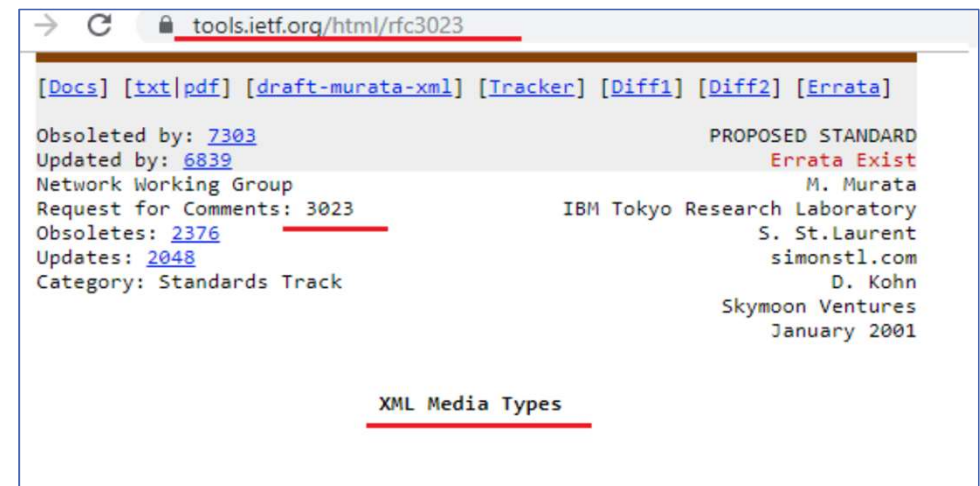
```
PS D:\NodeJS\samples\cwp_06_07> node .\06-07.js
{
  _comment: 'Так можно сделать комментарий',
  x: 1,
  y: 1.01,
  s: 'Строка',
  array: [ 'a', 'b', 'c', 'd' ],
  obj: { surname: 'Иванов', name: 'Иван' }
}
PS D:\NodeJS\samples\cwp_06_07>
```



# 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>
```



The screenshot shows a web browser window with the URL `tools.ietf.org/html/rfc3023`. The page contains links for [Docs], [txt|pdf], [draft-murata-xml], [Tracker], [Diff1], [Diff2], and [Errata]. The main content area displays the following information:

Obsoleted by: <a href="#">7303</a>	PROPOSED STANDARD
Updated by: <a href="#">6839</a>	Errata Exist
Network Working Group	M. Murata
Request for Comments: 3023	IBM Tokyo Research Laboratory
Obsoletes: <a href="#">2376</a>	S. St.Laurent
Updates: <a href="#">2048</a>	simonstl.com
Category: Standards Track	D. Kohn
	Skymoon Ventures
	January 2001

At the bottom of the page, there is a link labeled XML Media Types.

# 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('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>
```

# 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.

```
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="7000228" name="Казан Н.А." bday="2001-09-11"/>  
</students>
```

# 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');
    xmldoc.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 xmldata = '';
      req.on('data', (data) => { xmldata += data; });
      req.on('end', () => {
        parseString(xmldata, 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 REST client interface with a POST request to `http://localhost:3000`. The request body is XML, and the response is also XML.

**Request Body (XML):**

```
<?xml version="1.0" encoding="utf-8"?>
<students faculty="ИТ" speciality="ИСиТ" >
  <student id="7000222" name="Иванов И.И." bday="2000-12-02" />
  <student id="7000223" name="Петров П.П." bday="2000-11-26" />
  <student id="7000228" name="Казан Н.А." bday="2001-09-11" />
</students>
```

**Response Body (XML):**

```
<result>
  <students faculty="ИТ" speciality="ИСиТ">
    <quantity value="3"/>
  </students>
</result>
```

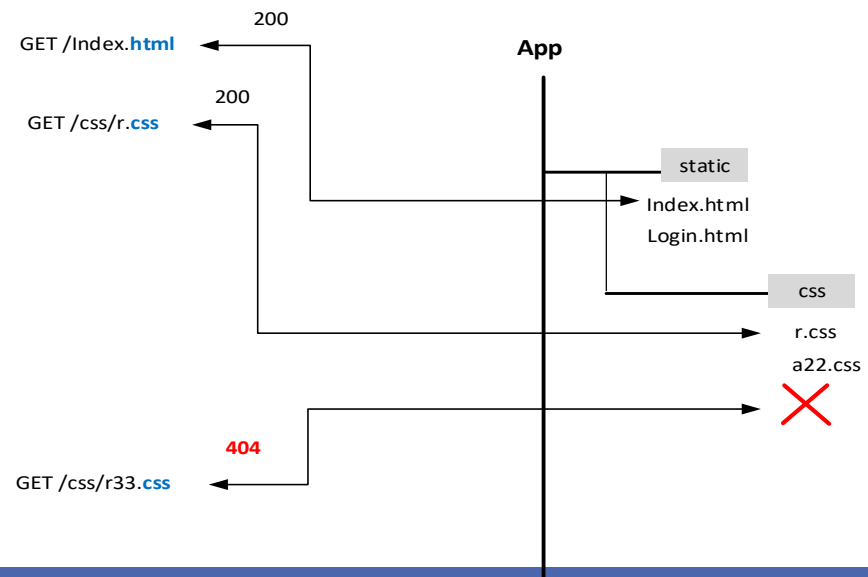
# Статические ресурсы

=

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>`, `<img>`, `<audio>`, `<video>`.

Обычно хранятся в 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)') });
server.on('error', (e) => { console.log('server.listen(3000): error: ', e.code) });
server.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	Размер
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...	

← → ↻ ⓘ localhost:3000/06-15.html  
Приложения Космос ТВ Google @MAIL.RU: почт

## 06-15.html

**Проверка CSS**

**Проверка JS:5**

```

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')})
  .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**.

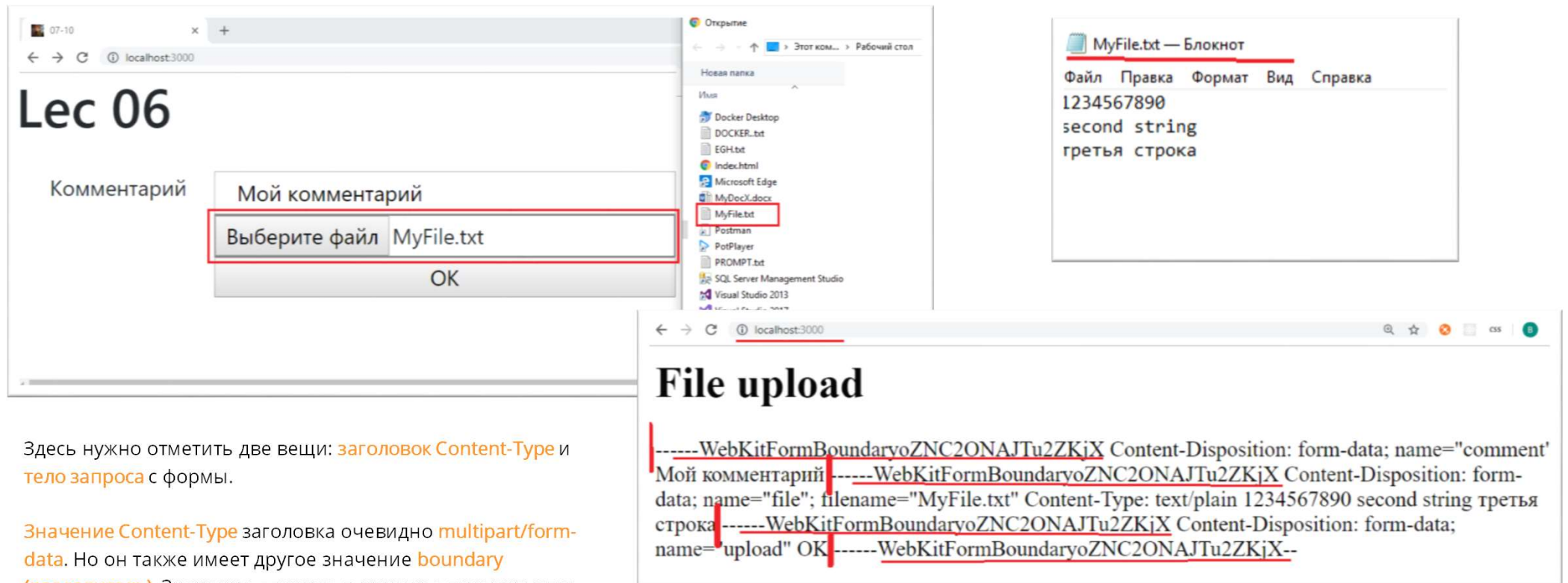
```

<!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 class="col-2">Комментарий</label> <input class="col-5" name="comment" type="text" />
    </div>
    <div class="row">
      <input class="col-5 offset-2" style="padding: 0px; border: 1px solid gray;" name="file" type="file" />
    </div>
    <div class="row">
      <input type="submit" class="col-5 offset-2" name="upload" value="OK"/>
    </div>
  </form>
</div>

```



# 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**

```
-----WebKitFormBoundaryZNC2ONAJTu2ZKjX Content-Disposition: form-data; name="comment"
Мой комментарий
-----WebKitFormBoundaryZNC2ONAJTu2ZKjX Content-Disposition: form-
data; name="file"; filename="MyFile.txt" Content-Type: text/plain 1234567890 second string третья
строка
-----WebKitFormBoundaryZNC2ONAJTu2ZKjX Content-Disposition: form-data;
name="upload" OK
-----WebKitFormBoundaryZNC2ONAJTu2ZKjX--
```

Вся полезная нагрузка заканчивается **boundary** значением с суффиксом **--**. Граничное значение позволяет браузеру понять, когда и где каждое поле начинается и заканчивается.

# Загрузка файла (upload)

```
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);
  }
}
```

```
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)
```

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.

# Демонстрация

← → ↻ ⓘ localhost:3000

## Form

---comment = Мой комментарий  
---file = MyFile.txt: files\_07-12\Q\_GaPZMwnm7QscJnJvH1rMV6.txt  
---subok = OK

Этот компьютер > WORK (D:) > PSCA > Lec07 > files\_07-12

Имя	Дата изменения	Тип	Размер
<u>Q_GaPZMwnm7QscJnJvH1rMV6.txt</u>	30.08.2019 0:33	Текстовый докум...	1 КБ
asvm4v4hGqzSJgCmrK31tMis.txt	30.08.2019 0:18	Текстовый докум...	1 КБ
e9sQv5uhtMGVNBvVF6gWUtSdQ.txt	29.08.2019 23:33	Текстовый докум...	1 КБ
54tTYCtqCkcm8C55oYZavblE.txt	29.08.2019 23:29	Текстовый докум...	1 КБ
d8X1ucdTuM-Jh1M9alPgBy8w.txt	29.08.2019 23:28	Текстовый докум...	1 КБ
cUXn-RVUjB-tJyMnXwEtKsAj.txt	29.08.2019 23:26	Текстовый докум...	1 КБ
a1S12Qvuf00nckz8vN5nVC5.txt	30.08.2019 23:22	Текстовый докум...	1 КБ

Модуль http крайне низкоуровневый.

Обычно для разработки выбирают фреймворки, вот самые популярные:

- express
- nest
- koa
- hapi
- restify