



**Нетология**  
университет интернет-профессий

# СОЗДАЕМ КЛИЕНТЫ И СЕРВЕРА ИСПОЛЬЗУЯ МОДУЛЬ HTTP



**ВАДИМ ГОРБАЧЕВ** / OPENWAY



# ВАДИМ ГОРБАЧЕВ

Developer Openway



[github.com/bmsdave](https://github.com/bmsdave)



[bmsdave](https://stackoverflow.com/users/1234567/bmsdave)



[linkedin.com/in/vadim-gorbachev-06a31485](https://linkedin.com/in/vadim-gorbachev-06a31485)



# ПЛАН ЗАНЯТИЯ

- Основы протокола HTTP.
- Создаём HTTP-клиент.
- Создаём HTTP-сервер.



# ОСНОВЫ ПРОТОКОЛА HTTP



***HyperText Transfer Protocol*** – протокол передачи гипертекста.

*Каждый запрос к серверу и ответ сервера состоит из трех частей: информация о типе сообщения, заголовки (может отсутствовать) и данные (может отсутствовать).*

*Информация о типе сообщения всегда идет первой строкой запроса и ответа. Потом следуют заголовки. Данные отделены от заголовков пустой строкой.*

– HTTP

# ПРОСТОЙ ЗАПРОС

1 | `GET /dist/latest-v6.x/docs/api/https.html HTTP/1.1`

Запрос состоит из одной строки, описывающей информацию о типе сообщения:

**GET** — Тип запроса

**/dist/latest-v6.x/docs/api/https.html** — Адрес страницы

**HTTP/1.1** — версия протокола

Заголовки и данные отсутствуют.

# ЗАПРОС С ЗАГОЛОВКАМИ И ДАННЫМИ

```
1 | POST /add HTTP/1.1LF  
2 | Host: www.marketplace.comLF  
3 |  
4 | id=29841&amount=1
```

В данном запросе передаются один заголовок `Host`. А так же данные в URL-кодированном виде:

```
4 | id=29841&amount=1
```

Пары ключ значение через `=` сцепленные символом `&`:

**id** — 29841

**amount** — 1

# ПРИМЕР ОТВЕТА СЕРВЕРА

```
1 HTTP/1.1 200 OKLF
2 Connection: closeLF
3 Content-Type: text/htmlLF
4 Set-Cookie: __cfduid=db6023c162265ec7fd9135e6e08680ce41471927212; expires=Wed,
5 Last-Modified: Fri, 12 Aug 2016 03:53:03 GMTLF
6 CF-Cache-Status: EXPIREDLF
7 Cache-Control: public, max-age=14400LF
8 Date: Tue, 23 Aug 2016 04:40:12 GMTLF
9 Server: cloudflare-nginxLF
10 CF-RAY: 2d6be29690b94ecc-DMELF
11 Expires: Tue, 23 Aug 2016 08:40:12 GMTLF
12 Transfer-Encoding: IdentityLF
13 LF
14 <!doctype html>LF
15 <html lang="en">LF
16 <head>LF
17   <meta charset="utf-8">LF
18   <title>HTTPS | Node.js v6.4.0 Documentation</title>LF
19   <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Lato:40
20   <link rel="stylesheet" href="assets/style.css">LF
21   <link rel="stylesheet" href="assets/sh.css">LF
22   <link rel="canonical" href="https://nodejs.org/api/https.html">LF
```



# ЗАГОЛОВКИ И ДАННЫЕ ОТВЕТА

Версия протокола и статус: `200` — код успешного ответа, `OK` — расшифровка кода ответа, не требующая пояснений 😊

```
1 | HTTP/1.1 200 OK
```

Сервер сообщает что передаёт нам текстовый документ в формате HTML:

```
3 | Content-Type: text/html
```

После пустой строки `13` идет обычный HTML-документ:

```
14 | <!doctype html>
15 | <html lang="en">
16 | <head>
17 |   <meta charset="utf-8">
18 |   <title>HTTPS | Node.js v6.4.0 Documentation</title>
19 |   <link rel="stylesheet" href="https://fonts.googleapis.c
```

# КОДЫ ОТВЕТА СЕРВЕРА

**200** — успешный ответ **OK**.

**301** — запрашиваемый документ теперь доступен по новому адресу **Moved Permanently**. Адрес передан в заголовке **Location**.

**302** — запрашиваемый документ временно доступен по новому адресу **Moved Temporarily**.

**400** — запрос имеет синтаксические ошибки **Bad Request**.

**401** — требуется авторизация **Unauthorized**.

**403** — доступ ограничен **Forbidden**.

**404** — запрашиваемый ресурс не найден **Not Found**.

**500** — возникла ошибка при обработке запроса **Internal Server Error**.

# ГРУППЫ КОДОВ ОТВЕТА

На самом деле кодов ответа гораздо больше. Все коды собраны в группы по типу ответа.

**1xx** — информация ( **100** — продолжай, **102** — идет обработка).

**2xx** — обработка прошла успешно ( **201** — создано, **202** — принято).

**3xx** — перенаправление клиента ( **303** — предложены другие варианты, **304** — не изменилось).

**4xx** — ошибки клиента ( **418** — I'm a teapot, **408** — истекло время ожидания запроса, **405** — метод не поддерживается, **451** — недоступно по юридическим причинам).

**5xx** — ошибки сервера ( **501** — не реализовано, **503** — сервис временно недоступен).



# ТИПЫ ЗАПРОСОВ КЛИЕНТА

**GET** — получение ресурса.

**POST** — создание ресурса.

**HEAD** — запрос только заголовков.

**PUT** — обновление ресурса.

**DELETE** — удаление ресурса

**OPTIONS** — определение возможностей сервера и параметров.

## ДАННЫЕ ДЛЯ РАЗНЫХ ТИПОВ ЗАПРОСОВ

Тип запроса	Данные запроса	Данные ответа
HEAD	нет	нет
GET	нет	да
DELETE	нет	да
OPTIONS	возможны	да
POST	да	да
PUT	да	да

# ЭКСПЕРИМЕНТИРУЕМ ИСПОЛЬЗУЯ УТИЛИТУ TELNET

```
$ telnet netology.ru 80
Trying 83.222.96.171...
Connected to netology.ru.
Escape character is '^]'.
```

```
GET / HTTP/1.1
Host: netology.ru
Connection: close
```

```
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 25 Aug 2016 02:42:40 GMT
Content-Type: text/html; charset=utf-8
...
Access-Control-Allow-Origin: *
```

```
1e39
<html>
<head>
<title>Нетология</title>
<meta charset="UTF-8">
```



# СОЗДАЁМ НТТР-КЛИЕНТ



## ЗАДАЧА

Получить курс валют и вывести в консоль курс Евро и Доллара США.

Актуальная информация по курсам валют доступна по адресу:

<https://netology-fbb-store-api.herokuapp.com/currency>.

Данные представлены в формате JSON.



---

# АЛГОРИТМ

1. Установить соединение с сервером.
2. Отправить запрос.
3. Получить данные ответа.
4. Преобразовать JSON в массив объектов.
5. Выбрать нужные валюты по кодам `USD` и `EUR`.
6. Вывести информацию в консоль.

# РЕШЕНИЕ

```
1  const https = require('https');
2  const url = 'https://netology-fbb-store-api.herokuapp.com/currency';
3  function process(data) {
4    let curr = JSON.parse(data);
5    curr
6      .filter(item => item.CharCode === 'USD' || item.CharCode === 'EUR')
7      .forEach(item => console.log(item.Name, item.Value));
8  }
9  function handler(response) {
10   let data = '';
11   response.on('data', function (chunk) {
12     data += chunk;
13   });
14   response.on('end', function () {
15     process(data);
16   });
17 }
18 const request = https.request(url);
19 request.on('response', handler);
20 request.end();
```

## ФОРМИРУЕМ ЗАПРОС

Для выполнения запроса используются модули `http` или `https`. В нашем случае так как протокол HTTPS:

```
1 | const https = require('https');
```

Для создания запроса вызовем метод `request`. Принимает на вход адрес страницы. Возвращает запрос:

```
18 | const request = https.request(url, handler);
```

Запрос является потоком. Чтобы его отправить, нужно его закрыть методом `end`:

```
20 | request.end();
```

# ПАРАМЕТРЫ ЗАПРОСА

Метод `request` принимает в качестве первого аргумента строку или объект.

Если передана строка, то для неё будет использован `url.parse`.

У объекта используются следующие свойства:

**host** — имя домена или IP адрес сервера, по умолчанию `localhost`,

**port** — порт сервера, по умолчанию `80`,

**method** — тип запроса, по умолчанию `GET`,

**path** — адрес ресурса, по умолчанию `/`,

**headers** — объект, позволяющий передать заголовки на сервер.

## ОБРАБОТЧИК ОТВЕТА

После отправки запроса и получения заголовков ответа, запрос бросит событие `response`. Назначим функцию обработчик на это событие:

```
19 | request.on('response', handler);
```

В обработчик передаётся объект `http.IncomingMessage` который представляет ответ сервера:

```
9 | function handler(response) {  
10 |   let data = '';  
11 |   response.on('data', function (chunk) {  
12 |     data += chunk;  
13 |   });  
14 |   response.on('end', function () {  
15 |     process(data);  
16 |   });  
17 | }
```

## ОТВЕТ СЕРВЕРА

Ответ сервера это поток. Данные с сервера передаются фрагментам. По событию `data` мы можем собрать все фрагменты в одну строку:

```
10 let data = '';  
11 response.on('data', function (chunk) {  
12     data += chunk;  
13 });
```

Когда все данные будут получены, поток будет закрыт, и вброшено событие `end`. Для нас это означает что мы получили все данные и можем приступить к их обработке, передав их в нашу функцию `process`:

```
14 response.on('end', function () {  
15     process(data);  
16 });
```

# СТАТУС ОТВЕТА СЕРВЕРА

У объекта так же есть свойства и методы. Свойство `statusCode` содержит код ответа сервера. По нему можно убедиться что мы получили запрошенные данные:

```
9  function handler(response) {
10    if (response.statusCode !== 200) {
11      console.log([
12        'Получен ответ',
13        response.statusCode,
14        response.statusMessage
15      ].join(' '));
16      return;
17    }
18    let data = '';
19    response.on('data', chunk => data += chunk);
20
```

# ЗАГОЛОВКИ ОТВЕТА СЕРВЕРА

Заголовки доступны в свойстве `headers` как объект и в `rawHeaders` как массив:

```
18 let type = response.headers['content-type']
19   .split(';')
20   .shift();
21 if (type !== 'application/json') {
22   console.log(['Получен формат', type].join(' '));
23   return;
24 }
```

Заголовок `Content-Type` имеет значение `application/json; charset=utf-8`, поэтому мы разбиваем на куски по `;` и берем первый.



# ОТПРАВКА ЗАГОЛОВКОВ И ДАННЫХ НА СЕРВЕР

```
1  const querystring = require('querystring');
2  let data = querystring.stringify({
3    'product': 'book',
4    'count': 1
5  });
6  let options = {
7    hostname: '127.0.0.1',
8    port: 3000,
9    path: '/add',
10   method: 'POST',
11   headers: {
12     'Content-Type': 'application/x-www-form-urlencoded',
13     'Content-Length': Buffer.byteLength(data)
14   }
15 };
16
17 const http = require('http');
18 let request = http.request(options);
19 request.write(data);
20 request.end();
```

# КОДИРУЕМ ДАННЫЕ

Данные можно представить обычным объектом:

```
1 let data = {  
2     'product': 'book',  
3     'count': 1  
4 };
```

Чтобы их передать на сервер, представим данные в URL-кодированном виде. Это преобразование реализовано в модуле `querystring`:

```
1 const querystring = require('querystring');  
2 let data = querystring.stringify({  
3     'product': 'book',  
4     'count': 1  
5 });
```

После преобразования данные будут иметь вид:

## ЗАДАЕМ ТИП ЗАПРОСА

Для отправки данных нам нужен метод `POST`. Поэтому мы не можем указать просто адрес:

```
5  let options = {  
6    hostname: '127.0.0.1',  
7    port: 3000  
8    path: '/add',  
9    method: 'POST',  
10   headers: {  
11     'Content-Type': 'application/x-www-form-urlencoded',  
12     'Content-Length': Buffer.byteLength(data)  
13   }  
14 };
```

# ПЕРЕДАЁМ ЗАГОЛОВКИ

Укажем в каком формате мы передаём данные в заголовке `Content-Type`. Наш формат `application/x-www-form-urlencoded`:

```
5 let options = {  
6   hostname: '127.0.0.1',  
7   path: '/add',  
8   method: 'POST',  
9   headers: {  
10     'Content-Type': 'application/x-www-form-urlencoded',  
11     'Content-Length': Buffer.byteLength(data)  
12   }  
13 };
```

Еще на всякий случай укажем размер передаваемых данных в байтах. Для этого воспользуемся `Buffer.byteLength`. Размер данных передается в заголовке `Content-Length`.

# ПЕРЕДАЁМ ДАННЫЕ

Запрос `request` это записываемый поток. Поэтому данные нужно просто записать в него методом `write`:

```
15 const http = require('http');  
16 let request = https.request(options);  
17 request.write(data);  
18 request.end();
```

Заголовки отправляются только с отправкой первой порции данных. Поэтому после создания запроса их еще можно менять с помощью метода `setHeader`.

## GET-ЗАПРОС С `http.get`

Метод `get` полный аналог `request` и отличается тем что задаёт тип запроса `GET` и вызывает `end` автоматически:

```
1  const http = require('http');
2
3  http.get('http://netology.ru/', res => {
4      console.log(`Статус ответа: ${res.statusCode}`);
5      res.pipe(process.stdout);
6  });
```

Так как `res` у нас читаемый поток, мы можем перенаправить вывод используя `pipe` прямо в поток вывода, который доступен в `process.stdout`.

## ОБРАБОТКА ОШИБОК СЕТИ

Так как запрос `request` является экземпляром `EventEmitter`, то все ошибки генерируют событие `error`:

```
1  const http = require('http');
2
3  http
4    .get('http://netology.ru/')
5    .on('error', err => console.error(err))
6    .on('response', res => {
7      console.log(`Статус ответа: ${res.statusCode}`);
8      res.pipe(process.stdout);
9    });
```

Но сюда не попадают ошибки клиента и сервера (`404`, `401`, `500`, `503`). Их нужно обрабатывать проверяя код ответа.

---

ИСПОЛЬЗУЕМ МОДУЛЬ  
`request` ДЛЯ  
СОЗДАНИЯ HTTP-  
КЛИЕНТА



# УСТАНОВКА МОДУЛЯ `request`

Модуль `request` позволяет

Простая установка

```
$ npm install request
```

Установка как зависимости

```
$ npm i -S request
```



# СОЗДАЁМ HTTP-СЕРВЕР

# ПРИМЕР HTTP-СЕРВЕРА

```
1  const http = require('http');
2  const port = 3000;
3
4  function handler(req, res) {
5    let name = req.url.replace('/', '') || 'World';
6    res.writeHead(200, 'OK', {'Content-Type': 'text/plain'});
7    res.write(`Hello ${name}!`);
8    res.end();
9  }
10
11 const server = http.createServer();
12 server.on('error', err => console.error(err));
13 server.on('request', handler);
14 server.on('listening', () => {
15   console.log('Start HTTP on port %d', port);
16 });
17 server.listen(port);
```

## СОЗДАЕМ СЕРВЕР

Функция создания сервера `createServer` находится в модуле `http`. Так же нам потребуется порт, на котором будет доступен наш сервер:

```
1 | const http = require('http');  
2 | const port = 3000;
```

Для создания сервера вызываем метод `createServer`:

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

Чтобы запустить сервер, нужно вызвать метод `listen` и передать порт:

```
17 | server.listen(port);
```

# ОБРАБАТЫВАЕМ ЗАПРОСЫ

Когда на сервер поступает запрос, вбрасывается событие `request`, назначим на него свой обработчик:

```
13 | server.on('request', handler);
```

Обработчик принимает в качестве аргументов два объекта: запрос (`IncomingMessage`) и ответ (`ServerResponse`). Запрос читаемый поток, ответ записываемый:

```
4 | function handler(req, res) {  
5 |   let name = req.url.replace('/', '') || 'World';  
6 |   res.writeHead(200, 'OK', {'Content-Type': 'text/plain'});  
7 |   res.write(`Hello ${name}!`);  
8 |   res.end();  
9 | }
```

# ПАРАМЕТРЫ ЗАПРОСА

Адрес запрашиваемой страницы находится в свойстве `url`. Удаляем в адресе `/` и если получается пустая строка то берём `World`:

```
5 | let name = req.url.replace('/', '') || 'World';
```

Если открыть адрес `http://localhost:3000/User` то `name` будет равно `User`, а если `http://localhost:3000/` то `World`.

Метод с которым отправлен запрос доступен в свойстве `method`, а заголовки в свойстве `headers`:

```
1 | console.log(req.method);  
2 | console.log(JSON.stringify(req.headers));
```

# ПОЛУЧАЕМ ДАННЫЕ ЗАПРОСА

Запрос `req` является читаемым потоком и данные от клиента поступают порциями. Будем читать их и ожидать закрытия потока, как при реализации клиента:

```
4 function handler(req, res) {  
5   let data = '';  
6   req.on('data', chunk => data += chunk);  
7   req.on('end', () => {  
8     res.writeHead(200, 'OK', {'Content-Type': 'text/plain'});  
9     res.write(data);  
10    res.end();  
11  });  
12 }
```

# РАЗБИРАЕМ ДАННЫЕ ЗАПРОСА

В зависимости от типа мы можем данные, которые поступают в виде строки, преобразовать в структуру:

```
19 function parse(data, type) {  
20     switch (type) {  
21         case 'application/json':  
22             data = JSON.parse(data);  
23             break;  
24         case 'application/x-www-form-urlencoded':  
25             data = querystring.parse(data);  
26             break;  
27     }  
28     return data;  
29 }
```

Не забудьте подключить модуль `querystring`.



# ИСПОЛЬЗУЕМ ДАННЫЕ ЗАПРОСА

Обновим обработчик события `end`, преобразуем данные в объект используя нашу функцию `parse`:

```
8 data = parse(data, req.headers['content-type']);
9 res.writeHead(200, 'OK', {'Content-Type': 'text/plain'});
10 res.write(`Hello ${data.name || 'World'}!`);
11 res.end();
```

# РАЗДАЧА СТАТИКИ

Подключим модуль `fs` и зададим папку с файлами сервера в константе `base`

```
1  const http = require('http');  
2  const fs = require('fs');  
3  
4  const port = 3000;  
5  const base = './public';
```

# РЕАЛИЗУЕМ ФУНКЦИЮ ПРОВЕРКИ ФАЙЛА

Она должна проверить что указанный путь является файлом и у нас есть доступ на чтение:

```
7 function checkFile(filename) {  
8   return new Promise((resolve, reject) => {  
9     fs.stat(filename, (err, stat) => {  
10      if (err) return reject(err);  
11      if (stat.isDirectory()) {  
12        return resolve(checkFile(filename + 'index.html'));  
13      }  
14      if (!stat.isFile()) return reject('Not a file');  
15      fs.access(filename, fs.R_OK, err => {  
16        if (err) reject(err);  
17        resolve(filename);  
18      })  
19    });  
20  });  
21 }
```

Если указанный путь это папка, то проверяем доступ к файлу `index.html` в ней.

## РЕАЛИЗУЕМ ОБРАБОТЧИК ЗАПРОСОВ

Проверяем файл который запрошен в URL относительно папки `base`.  
Если файл доступен, то выдаем его в качестве тела ответа. Иначе сообщаем об ошибке `404`:

```
23 function handler(req, res) {  
24     checkFile(base + req.url)  
25         .then(filename => {  
26         res.writeHead(200, 'OK', {'Content-Type': 'text/html'})  
27         fs.createReadStream(filename).pipe(res);  
28     })  
29     .catch(err => {  
30         res.writeHead(404, http.STATUS_CODES[404], {'Content-Type': 'text/html'})  
31         res.end('File not found');  
32     });  
33 }
```

## СОЗДАЕМ И ЗАПУСКАЕМ СЕРВЕР

В этой части ничего нового и необычного. Всю работу делает подключенный обработчик `handler`:

```
35  const server = http.createServer();
36  server
37    .listen(port)
38    .on('error', err => console.error(err))
39    .on('request', handler)
40    .on('listening', () => {
41      console.log('Start HTTP on port %d', port);
42    });
```

# НЕ «ЗАШИВАЙТЕ» ПОРТ СЕРВЕРА ВНУТРИ ПРИЛОЖЕНИЯ

Позвольте задавать порт через переменную окружения. Переменные окружения доступны в `process.env`:

```
1  const http = require('http');
2  const PORT = process.env.PORT || 3000;
3
4  http.createServer()
5    .listen(PORT)
6    .on('listening', () => {
7      console.log('Start HTTP on port %d', PORT);
8    });
```

Тогда ваш код будет работать из коробки на [Heroku](#) и [Cloud9](#).

# ЗАДАНИЕ ПЕРЕМЕННОЙ ОКРУЖЕНИЯ

Обычный запуск:

```
$ node app.js  
Start HTTP on port 3000
```

Задание переменной окружения при запуске:

```
$ PORT=80 node app.js  
Start HTTP on port 80
```

# КОД СЕРВЕРА ДЛЯ HTTP КЛИЕНТА (ЧАСТЬ #1)

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const port = 3000;
4
5  let app = express();
6  var market = {};
7
8  app.use(bodyParser.json({limit: '5mb'}));
9  app.use(bodyParser.urlencoded({limit: '5mb', extended: true}));
10
11 app.listen(port);
```



## КОД СЕРВЕРА ДЛЯ HTTP КЛИЕНТА (ЧАСТЬ #2)

```
1  app.post('/add', (req, res) => {
2    let product      = req.body.product;
3    let count        = parseInt(req.body.count);
4
5    if (!product || !count) return res.send({status: 'error'});
6    if (!market[product]) market[product] = 0;
7    market[product] += count;
8
9    res.json({product: product, count: market[product], status: 'ok'});
10 }
```



# ЗАДАНИЕ

<https://docs.google.com/document/d/1TAklGtTumMVoVvSxPSkRRfFz-g8pksWWeJrwE0ldiSU/edit?usp=sharing>



**нетология**  
университет интернет-профессий

**Задавайте вопросы и напишите отзыв о лекции!**

**ВАДИМ ГОРБАЧЕВ**



[github.com/bmsdave](https://github.com/bmsdave)



[bmsdave](https://soundcloud.com/bmsdave)



[linkedin.com/in/vadim-gorbachev-](https://linkedin.com/in/vadim-gorbachev-06a31485)

[06a31485](https://linkedin.com/in/vadim-gorbachev-06a31485)