

# ПРОГРАММИРОВАНИЕ В INTERNET

---

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: ', req.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: ', 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('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 получают 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`.

# 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.

```
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();
```

```
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: 'xxxyyy' }
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' } } ]
```



```

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

```

# Загрузка файла на сервер

# 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
- supertest
- node-fetch