Node.js

Часть 3

В предыдущей лекции

- Были рассмотрены принципы работы Event loop в Node.js
- Переменные, объявленные на верхнем уровне, автоматически не становятся глобальными.
- Были рассмотрены процессы и работа с ними, отправка сообщений, порождение дочерних процессов с помощью методов exec, spawn, fork.
- Бинарные данные хранятся в экземплярах класса Buffer, с ним ассоциирована область памяти, выделенная вне стандартной кучи V8.
- Для работы с таймерами используются методы SetTimer, ClearTimer, SetInterval, ClearInterval.
- За события в Node.js отвечает специальный модуль – events.
- EventEmitter - это основной объект, реализующий работу обработчиков событий в Node.js. Любой объект, являющийся источником событий, наследует от класса EventEmitter.
- На базовом уровне рассмотрена работа с модулями.

В этой лекции

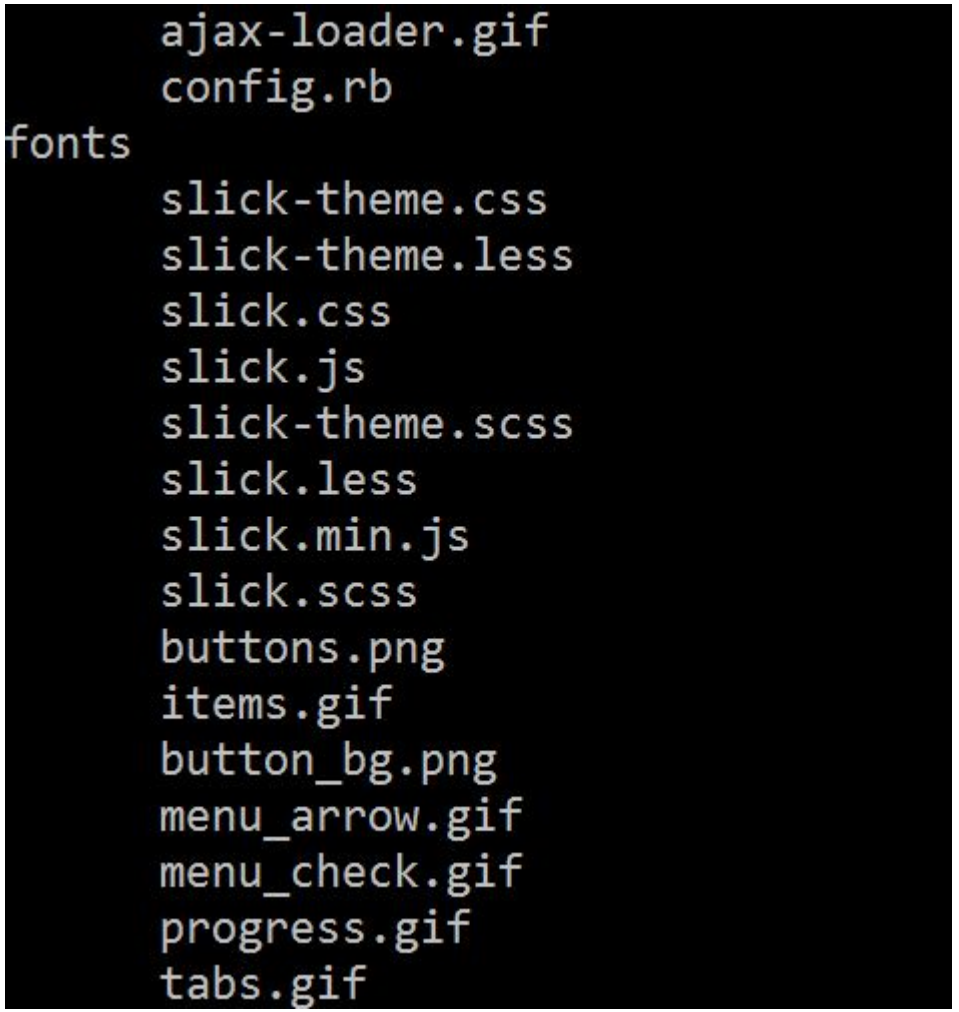
- Работа с файлами
- Создаем TCP-сервер
- WebSockets

Работа с файлами

- Модуль FileSystem входит в дистрибутив Node.js, и, честно говоря, мы его уже использовали для чтения html-контента. Естественно, этим его функции не ограничиваются; лучше всего показать его работу на примере конкретной задачи. Причем важной. Скажем, нужно составить список всех композиций в формате mp3 на винчестере с указанием их местоположения. Наверняка вам нечто подобное уже приходилось писать на C или, скажем, на Perl. Теперь очередь JavaScript, и мы сейчас убедимся, что этот язык справится с данной задачей ничуть не хуже.

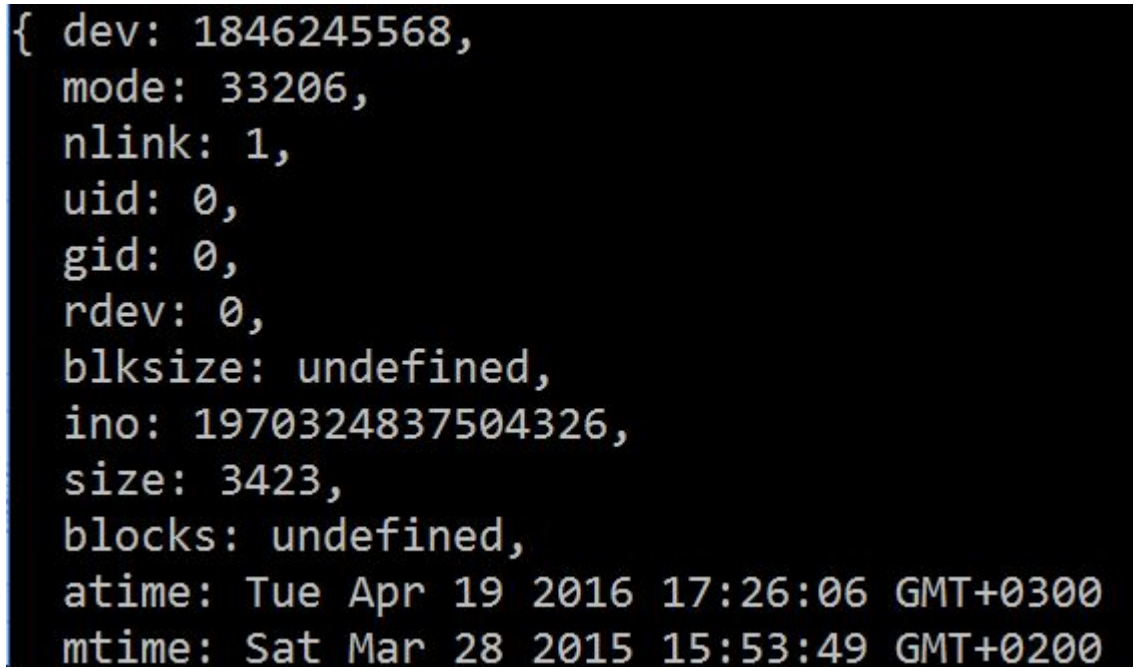
Рекурсивный обход каталогов

```
var base = 'D:\\Development\\SDAssembla';
var fs = require('fs');

function readDir(base) {
    fs.readdir(base, function (err, files) {
        files.forEach(function (item) {
            fs.stat(base + '\\' + item, function (err, state) {
                if (state.isDirectory()) {
                    console.log(item);
                    localBase = base + '/' + item;
                    readDir(localBase);
                }
                else {
                    console.log(" " + item);
                }
            });
        });
    });
}
readDir(base);
```



fs.readdir(), fs.stat().

- Для чтения каталога используется метод fs.readdir(), имеющий аналоги во многих языках программирования.
- Для определения, является ли полученный файл директорией, используется объект fs.Stats, возвращаемый методом fs.stat(). Это объект, содержащий различную информацию о найденном файле.



fs.Stats

- stats.isFile() - проверяет, является ли объект файлом;
- stats.isDirectory() - проверяет, является ли объект директорией;
- stats.isBlockDevice() -проверяет, является ли объект файлом устройства блочного ввода/вывода;
- stats.isCharacterDevice() - проверяет, является ли объект файлом устройства посимвольного ввода/вывода;
- stats.isSymbolicLink() -проверяет, является ли объект символической ссылкой (при этом для получения stat должен быть использован специальный метод - fs.lstat());
- stats.isFIFO() - проверяет, является ли объект FJFO-файлом (именованным каналом);
- stats.isSocket() -проверяет, является ли объект сокетом. Этого арсенала должно хватить, чтобы получить информацию для

Результаты работы предыдущей программы

- Все это замечательно, но вот вывод предыдущей программы нас может не устроить.
- Это слабо упорядоченная смесь названий файлов и директорий, ориентироваться в которой просто нельзя. Почему это случилось? Дело в том, что методы fs.readdir() и fs.stat() асинхронны и совсем не обязаны выдавать результат в строгой очередности.
- Для целого ряда задач (например, нам бы понадобилось массово переименовать файлы или просканировать их содержимое) такой подход не только уместен, но и наиболее эффективен.
- Многие ключевые методы модуля fs имеют свои синхронные аналоги. В том числе fs.readdir() и fs.stat().

Синхронная версия

```
var fs = require('fs');
var base = 'D:\\Development\\SDAssembla\\securedating';

String.prototype.repeat = function (num) { return new Array(num + 1).join(this); }

function readDir(base, indent) {
    files = fs.readdirSync(base)
    files.forEach(function (item) {
    state = fs.statSync(base + '\\' + item);
        if (state.isDirectory()) {
            console.log("\n" + " ".repeat(indent * 2) + item + "\n");
            localBase = base + '/' + item;
            readDir(localBase, indent + 1);
        }
        else {
            console.log(" ".repeat(indent * 2) + item);
        }
    });
}
readDir(base, 0);
```

```
youtube.png

_r

  24h Customer_New.png
  Airport Transfer_New.png
  Apartment Rental_New.png
  btn_unlock.png
  Chat room_New.png
  Correspondence_New.png
  email.png
  Gift Delivery_New.png
  join for free_New.png
  JOINFORFREE RESPONSIVE.png
```

Файлы по папкам в алфавитном порядке

```
var fs = require('fs');
var path = require('path');
var base = 'D:\\Development\\SDAssembla\\site_pics'; // тут хранятся файлы
var collection = 'D:\\Development\\SDAssembla\\sorted'; //тут будет упорядоченная коллекция
function collect(base) {
    fs.readdir(base, function (err, files) {
        files.forEach(function (item) {
            console.log(item.charAt(0));
            var fileName = collection + '\\' + item.charAt(0);
            if (fs.existsSync(fileName)) {
                copyRecursive(base + "\\" + item, fileName + "\\" + item);
            } else {
                fs.mkdir(fileName, function () {
                    copyRecursive(base + "\\" + item, fileName + "\\" + item);
                });
            }
        });
    });
}
```

Файлы по папкам в алфавитном порядке

- Тут мы пользуемся синхронной версией метода (fs.existsSync()), проверяющего существование объекта файловой системы (есть и асинхронный).
- Метод fs.mkdir() предсказуемо создает директорию, а вот с методом copyRecursive() сложнее.
- Такого в документации нет. В составе fs вообще нет аналога posix команды copy().

CopyRecursive

```
var copyRecursive = function (src, dest) {
    var exists = fs.existsSync(src);
    var stats = fs.statSync(src);
    var isDirectory = exists && stats.isDirectory();
    if (isDirectory) {
        fs.mkdirSync(dest);
        fs.readdirSync(src).forEach(function (childitemName) {
            copyRecursive(path.join(src, childitemName), path.join(dest, childitemName));
        });
    }
    else {

        fs.linkSync(src, dest);
        collect(base);
    }
}

collect(base);
```

fs.mkdirSync(), fs.linkSync()

- Что тут нового? В самом приеме рекурсивного обхода ресурсов файловой системы точно нет никаких инноваций.
- А вот на что стоит обратить внимание, так это на метод создания директории (fs.mkdirSync(), он тоже имеет синхронную форму) и метод fs.linkSync(), создающий, по идее, жесткую ссылку на файл, но в данной ситуации это соответствует процедуре копирования.

Удаление файлов

```
var fs = require('fs');
var path = require('path');
var collection = 'D:\\Development\\SDAssembla\\sorted';
fs.rmdir(collection, function (error) {
    if (error) { console.error(error.message); }
});
```

```
Debugger listening on port 5858
ENOTEMPTY, rmdir 'D:\Development\SDAssembla\so
```

Все правильно, директория не пуста. Тут тоже придется прибегнуть к рекурсии:

Удаление файлов

```
var fs = require('fs');
var collection = 'D:\\Development\\SDAssembla\\sorted';
function removeDir(path){
    if (fs.existsSync(path)){
        fs.readdirSync(path).forEach(
            function(file) {
                var f = path + "\\" + file;
                var stats = fs.statSync(f);
                if (stats.isDirectory())
                {
                    removeDir(f);
                }
                else
                {
                    fs.unlinkSync(f)
                };
                console.log(f + "is removed");
            });
        fs.rmdirSync(path);
        console.log(path + " is removed");
    }
}
removeDir(collection);
```

Path Resolve

Самый, наверное, интересный метод из небольшого арсенала модуля path это path.resolve(), разрешающий (преобразующий) заданный путь в абсолютный:

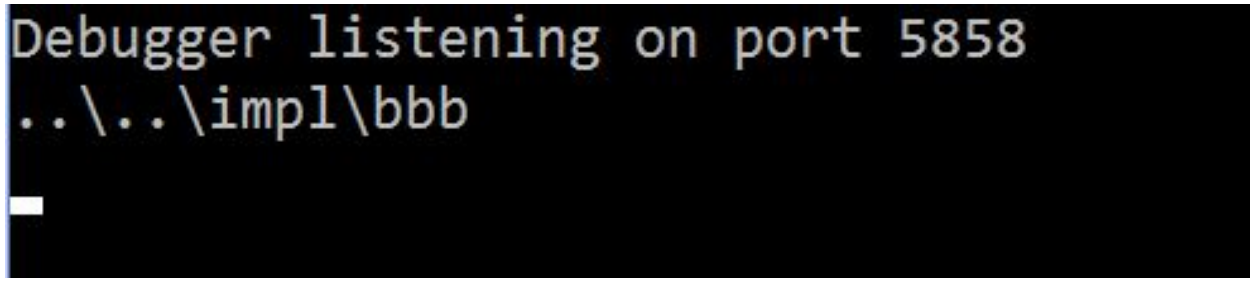
```
var path = require('path');
var resolved = path.resolve('./CouponCodeUpdate.txt');
console.log(resolved);
```

```
D:\Development\_SpaceJet2\Politech\Technolgy\4_NodeJs\T
stConsole\wwwroot
```

Path Relative

- Метод path.relative(), преобразует заданный путь в относительный (дополняет метод Resolve)

- var path = require('path');
- var resolved = path.relative('C:\\zzz\\test\\aaa', 'C:\\zzz\\impl\\bbb');
- console.log(resolved);



Path.normalize

- Метод path.normalize() «приводит пути в порядок», то есть удаляет из них все, что там быть не должно, но появилось, например, из-за специфичного формата ввода (сочетания символов .. , ., или //):
- var path = require('path');
- var myPath = '/foo/bar///baz/asdf/quux/ ';
- myPath = path.normalize(myPath);
- console.log(myPath);

```
\foo\bar\baz\asdf\quux\
```

Path.join

Метод path.join() позволяет соединять пути в файловой системе

- var path = require('path');
- var myPath = path.join('/foo', 'bar', 'baz/asdf', 'q');
- console.log(myPath);

```
\foo\bar\baz\asdf\q
```

Разные полезные мелочи

- path.extname() – определяем расширение файла.
- path.sep - определяем специфичный для платформы разделитель в пути к файлу расширение файла.
- path.delimiter - определяем специфичный для платформы разделитель путей.

```
C:\test.json
.json
```

```
\
```

```
;
```

Разные полезные мелочи

- path.dirname() - определяем имя директории, содержащей файл:

```
C:\Test\Test.json
C:\Test
```

- path.basename() - определяем базовое имя файла

```
C:\Test\Test.json
Test.json
```

__dirname и __filename

- две глобальные переменные платформы Node.js –
- **__dirname** и **__filename**.
- Первая хранит имя текущей директории, вторая - текущего файла.

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

console.log(__dirname);

console.log(__filename);
```

```
D:\Development\_SpaceJet2\Politech\Technolgy\4_NodeJs\TestProject\TestConsole\Te
stConsole
D:\Development\_SpaceJet2\Politech\Technolgy\4_NodeJs\TestProject\TestConsole\Te
stConsole\app.js
```

Перемещение по файловой системе

- Перемещаться по файловой системе (то есть менять рабочую папку) модуль fs не поможет. Это операция «ядерного уровня», она доступна через процессы.
- Например, так можно узнать текущую рабочую директорию: console.log("The current working directory is " + process.cwd());

Перемещение по файловой системе

Так её можно поменять:

```
console.log("The current directory is " + process.cwd());
try {
    process.chdir("D:\\Development");
    console.log("The new current directory is " +
process.cwd());
}
catch (exception) {
    console.error("chdir error: " + exception.message);
}
```

```
The current directory is D:\Development\_SpaceJet2\Politech\Technolgy\4_NodeJs\T
estProject\TestConsole\TestConsole
The new current directory is D:\Development
```

Работа с файлами

```
var fs = require('fs');
var path = "D:\\Development\\Test\\notes.txt";
fs.open(path, "r+", function (error, fd) {
    if (error) {
        console.error("open error: " + error.message);
    }
    else {
        console.log("Successfully opened " + path);
        fs.close(fd, function (error) {
            if (error) {
                console.error("close error: " + error.message);
            }
            else {
                console.log("Successfully closed " + path);
            }
        });
    }
});
```

```
Successfully opened D:\Development\Test\notes.txt
Successfully closed D:\Development\Test\notes.txt
```

fs.open

Метод fs.open() в качестве первого параметра принимает имя файла, последним служит функция обратного вызова, а вторым - флаг режима открытия, который в Node.js имеет свои особенности.

Ниже приведены его возможные значения:

- r - открыть для чтения. Генерирует исключение при отсутствии файла;
- r+ -открыть для чтения и записи. Генерирует исключение при отсутствии файла;
- rs -открыть для чтения в синхронном режиме;
- rs+ -открыть для чтения и записи в синхронном режиме;
- w - открыть для записи. Если файл не существует, он будет создан. Если файл существует, его содержимое будет очищено;
- w+ - открыть для чтения и записи. Если файл не существует, он будет создан. Если файл существует, его содержимое будет очищено;
- a -открыть для записи в конец файла. Если файл не существует, он будет создан;
- a+ -открыть для чтения и записи в конец файла. Если файл не существует, он будет создан.

Чтение на низком уровне

```
var fs = require('fs');
var path = "D:\\Development\\Test\\notes.txt";

fs.open(path, "r+", function (error, fd) {
    if (error) {
        console.error("open error: " + error.message);
    }
    else {
        console.log("Successfully opened " + path);
        fs.stat(path, function (error, stats) {
            var buffer = new Buffer(stats.size);
            fs.read(fd, buffer, 0, buffer.length, null, function (error, bytesRead, buffer)
            {
                var data = buffer.toString("utf8");
                console.log(data);
            });
        });
    }
});
```

```
Successfully opened D:\Development\Test\notes.txt
This is data from notes.txt
```

Чтение на низком уровне

- Метод fs.read(), получая в качестве аргумента дескриптор файла, читает данные из него, “как есть”, то есть, в общем случае, в виде бинарных данных.
- Для того, чтобы их получить,
  - мы сначала создаем буфер (для того, чтобы определиться с его размером, нам опять потребовался объект fs.stat()),
  - читаем в него данные и преобразуем их в строковой формат перед выводом в консоль. Второй аргумент функции обратного вызова метода fs.read() - это количество прочитанных байт.

Запись на низком уровне

Запись в файл происходит по той же схеме (сделаем программу которая записывает в файл логи обращения к ней):

```
var logItem = "Note created " + Date.now() + " ";
buffer = new Buffer(logItem);
fs.write(fd, buffer, 0, buffer.length, null, function (error, bytesWritten, buffer)
{
    if (error) { console.error(error.message); }
    else {
        console.log("Written " + bytesWritten + " bytes.");
    }
});
```

Note created 1475662875533 Note created 1475

ReadFile

- На предыдущем слайде мы создаем свой буфер из заданной строки и пишем его в файл. Все очень просто и универсально, но, честно говоря, не совсем удобно. По крайней мере, для текстовых данных. Модуль fs располагает более высокоуровневыми методами:

```
var fs = require('fs');
var path = "D:\\Development\\Test\\notes.txt";
fs.readFile(path, "utf8", function (error, data) {
    if (error) {
        console.error(error.message);
    }
    else {
        console.log(data);
    }
});
```

WriteFile

- Это все - не надо заботиться о получении файлового дескриптора и подготовке буфера - все это уже инкапсулировано в методы readFile, writeFile

```
var fs = require('fs');
var path = "D:\\Development\\Test\\notes.txt";

var logItem = "Note created " + Date.now() + "\n";
fs.writeFile(path, logItem, function (error) {
    if (error) { console.error(error.message) }
    else {
        console.log("Successfull write " + path)
    }
});
```

Можно также использовать метод

fs. appendFile

Watching Files

Это, наверное, самая интересная возможность модуля fs. С помощью метода fs.watch() мы можем отслеживать состояние файлов. Например, нашего файла логов

```
var fs = require("fs");
var path = "D:\\Development\\Test\\notes.txt";
fs.watch(path, { persistent: true }, function (event, filename) {
    console.log(event)
    if (event === "rename") {
        console.log("The file was renamed/deleted. ");
    }
    else if (event === "change") {
        console.log("The file was changed.");
    }
});
```

```
rename
The file was renamed/deleted.
rename
The file was renamed/deleted.
change
```

Watching Files

- Node.js для получения данных использует именно системные средства. В операционной системе Linux это подсистема ядра inotify, в BSD и OS X -интерфейс уведомления о событиях kqueue, в семействе Windows применяется вызов функции ReadDirectoryChangesW

Потоки

```
var fs = require("fs");
var path = "D:\\Development\\Test\\fluid.blend";
var stream = fs.createReadStream(path);

stream.on('readable', function ()
{
    console.log('read');
});
stream.on('end', function () {
    console.log('end');
});

stream.on('data', function (chunk) {
    console.log('got %d bytes of data', chunk.length);
});
```

Потоки

- Поток можно в любой момент закрыть, вызвав метод stream.close():

```
stream.on('data', function (chunk) {
    if (chunk.length < 10) {
        stream.close();
    }
    console.log('got %d bytes of data', chunk.length);
});
```

```
stream paused
got 65536 bytes of data
read
stream resumed
stream paused
got 65536 bytes of data
read
stream resumed
stream paused
got 65536 bytes of data
read
```

Stream.pause() & Stream.resume

для более гибкой работы с потоком присутствуют методы stream.pause() и stream.resume():

```
stream.on('data', function (chunk) {
    stream.pause();
    console.log("stream paused");
    setTimeout(function () {
        console.log("stream resumed");
        stream.resume();
    }, 1000);
    console.log('got %d bytes of data',
   chunk.length);
});
```

```
stream paused
got 65536 bytes of data
read
stream resumed
stream paused
got 65536 bytes of data
read
stream resumed
stream paused
got 65536 bytes of data
read
```

Веб-сервер на потоках

Освоив потоки, мы теперь можем более рационально переписать веб-сервер из лекции №4. В чем его нерациональность? Ну хотя бы в том, что при запросе браузером очень больших файлов (а такая ситуация вполне обычна) мы вынуждены до отдачи данных целиком считывать его в память, что недопустимо для сколько-либо серьезно нагруженного веб-сервера:

```
/* fs.readfile(pathname, 'utf8',
    function (err, data) {
        if (err) {
            console.log('Could not find or open file' + pathname + ' for reading\n')
        }
        else {
            console.log(pathname + " " + mimeType);
            response.write(data);
            response.end();
        }
    }); */
```

Веб-сервер на потоках

```
var http = require('http');
var url = require('url');
var fs = require('fs');
var port = 2222;
http.createServer(function (req, res) {
    var pathname = url.parse(req.url).pathname;
    if (pathname == '/') { pathname = '/index.html'; }
    pathname = pathname.substring(1, pathname.length);
    var stream = fs.createReadStream(pathname, { encoding: 'utf8' });
    stream.on('readable', function () {
        var data = stream.read();
        if (data) {
            res.write(data.toString());
        }
    });

    stream.on('end', function () {
        res.end();
    });
}).listen(port);
```

stream.pipe()

- В таком виде все работает, причем корректно, но на самом деле это только полдела. Даже меньше. Мы действительно читаем данные из входящего потока, но затем перед записью в исходящий поток сохраняем их в переменную. Данная проблема решается следующим образом

```
stream.on('readable', function () {
    //var data = stream.read();
    //if (data) {
    //    res.write(data.toString());
    //}
    stream.pipe(res);
});
```

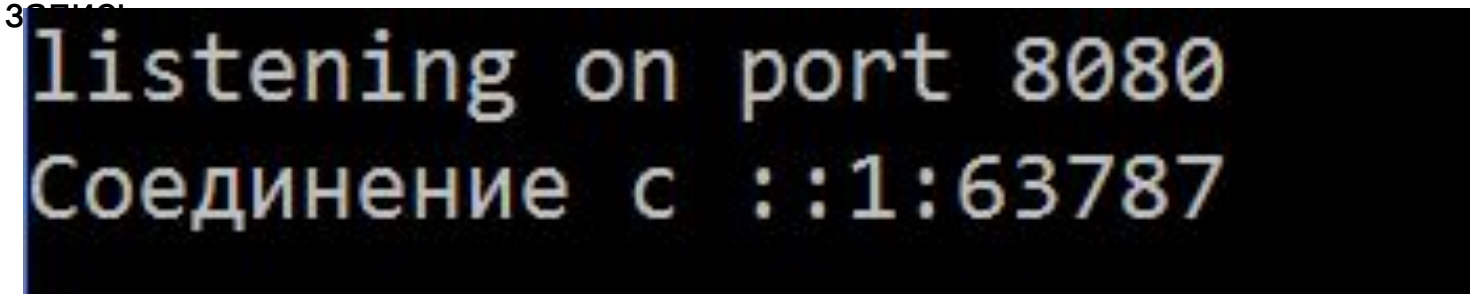
Создаем TCP-сервер

- На платформе Node.js он реализован в модуле net, входящем в ядро системы. Построить TCP-сервер - задача довольно тривиальная. В отличие от HTTP-сервера, функция обратного вызова, являющаяся аргументом при создании TCP-сервера, принимает только один аргумент -экземпляр соединения. Он же сокет.

Создаем TCP-сервер

```
var net = require('net');
var server = net.createServer(function (socket)
{
    console.log('Соединение с ' +  socket.remoteAddress + ":" + socket.remotePort);
}).listen(8080);
console.log('listening on port 8080');
```

И стучимся браузером по адресу http://localhost:8080. В самом браузере, естественно, ничего не отобразится, зато в консоли появится запись:

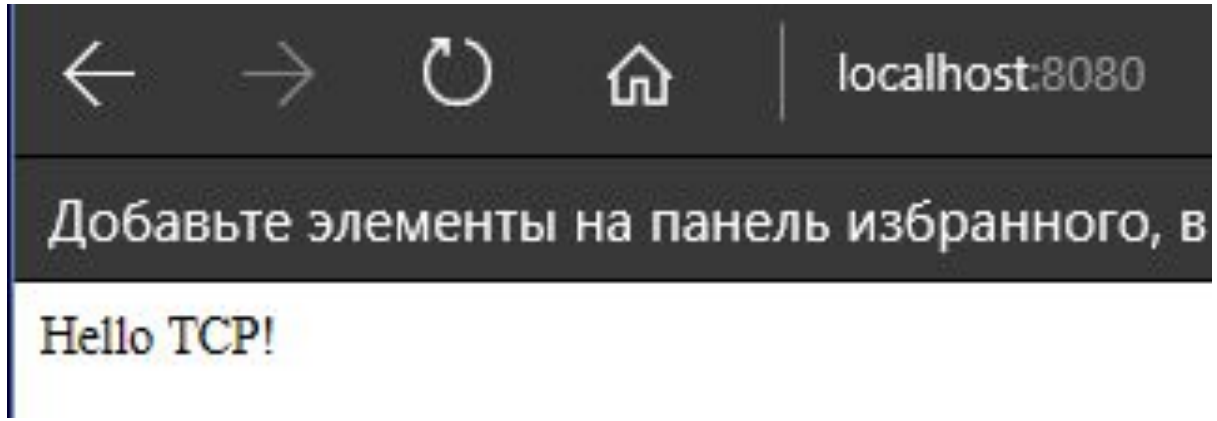


Сокет

- О сокетах. А что это, собственно, вообще такое? Если у вас такого вопроса не возникает, с чистой совестью пропускайте следующую пару абзацев.
- В общем случае сокет - это абстрактный объект, представляющий собой программный интерфейс для обеспечения обмена данными между процессами, вообще, любыми программными процессами. Попросту, сокет - это место встречи, пересечения, обмена данными, о котором договорились два процесса, столкнувшихся с необходимостью взаимодействовать.
- По выполняемым ролям сокеты делятся на клиентские и серверные. Каждый процесс операционной системы может создать слушающий (серверный) сокет и привязать его к какому-нибудь локальному адресу (собственно, пара адресов - адрес компьютера в сети, локальный адрес - и определяют сокет как точку обмена данными). Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. Клиентские сокеты используют различные клиентские приложения (например, браузер). Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будет передавать данные между ним и сервером.

Socket.write

```
var net = require('net');
var server = net.createServer(function (socket)
{
    console.log('Соединение с ' + socket.remoteAddress + ":" + socket.remotePort);
    socket.write('Hello TCP!');
    socket.end();
}).listen(8080);
console.log('listening on port 8080');
```

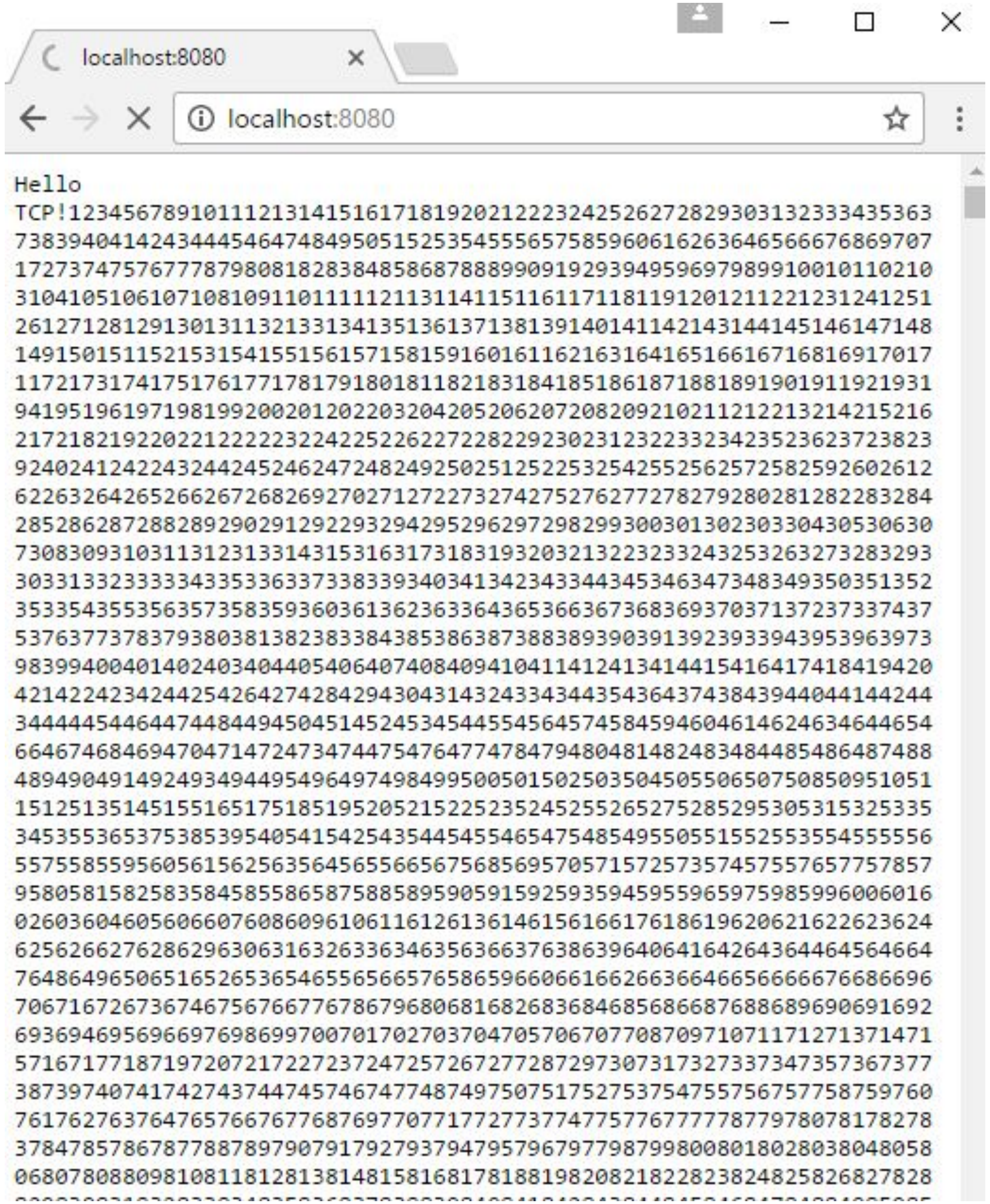


Socket.end()

- Строкой socket.end() мы закрываем сокет; если бы мы этого не сделали, то браузер продолжил бы чтение из сокета, и переданное сообщение не задержалось бы на экране.
- Продемонстрировать непрерывную работу сокета можно следующим кодом:

Непрерывная работа сокета

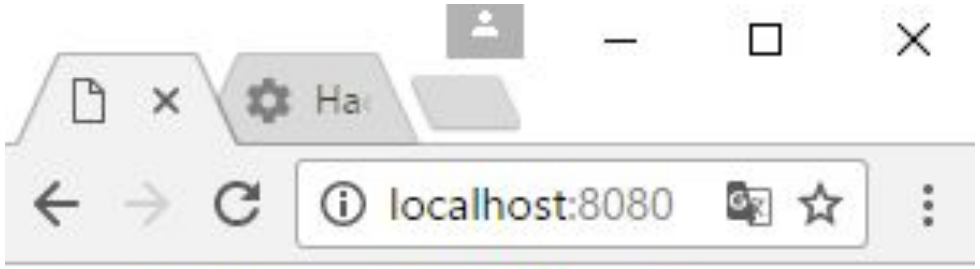
```
var net = require('net');
var server = net.createServer(function (socket)
{
    console.log('Соединение с ' + socket.remoteAddress + ":" + socket.remotePort);
    socket.write('Hello TCP!');
    var i = 0;
    while (socket) {
        i++;
        var m = '' + i;
        socket.write(m);
    }
    socket.end();
}).listen(8080);
console.log('listening on port 8080');
```


localhost:8080
localhost:8080
Hello
TCP!1234567891011121314151617181920212223242526272829303132333435363
73839404142434445464748495051525354555657585960616263646566676869707
17273747576777879808182838485868788899091929394959697989910010110210
31041051061071081091101111121131141151161171181191201211221231241251
26127128129130131132133134135136137138139140141142143144145146147148
14915015115215315415515615715815916016116216316416516616716816917017
11721731741751761771781791801811821831841851861871881891901911921931
94195196197198199200201202203204205206207208209210211212213214215216
21721821922022122222322422522622722822923023123223323423523623723823
92402412422432442452462472482492502512522532542552562572582592602612
62263264265266267268269270271272273274275276277278279280281282283284
28528628728828929029129229329429529629729829930030130230330430530630
73083093103113123133143153163173183193203213223233243253263273283293
30331332333334335336337338339340341342343344345346347348349350351352
35335435535635735835936036136236336436536636736836937037137237337437
53763773783793803813823833843853863873883893903913923933943953963973
98399400401402403404405406407408409410411412413414415416417418419420
42142242342442542642742842943043143243343443543643743843944044144244
34444454464474484494504514524534544554564574584594604614624634644654
66467468469470471472473474475476477478479480481482483484485486487488
48949049149249349449549649749849950050150250350450550650750850951051
15125135145155165175185195205215225235245255265275285295305315325335
34535536537538539540541542543544545546547548549550551552553554555556
55755855956056156256356456556656756856957057157257357457557657757857
95805815825835845855865875885895905915925935945955965975985996006016
02603604605606607608609610611612613614615616617618619620621622623624
62562662762862963063163263363463563663763863964064164264364464564664
76486496506516526536546556566576586596606616626636646656666676686696
70671672673674675676677678679680681682683684685686687688689690691692
69369469569669769869970070170270370470570670770870971071171271371471
57167177187197207217227237247257267277287297307317327337347357367377
38739740741742743744745746747748749750751752753754755756757758759760
76176276376476576676776876977077177277377477577677777877978078178278
37847857867877887897907917927937947957967977987998008018028038048058
06807808809810811812813814815816817818819820821822823824825826827828

Socket.on('data‘)

- Заставим наш сокет слушать пару событий

```
var net = require('net');
var server = net.createServer(function (socket) {
    socket.on('data', function (data) {
        console.log(data.toString());
        socket.write("Received data: " + data);
        socket.end();
    });
    socket.on('close', function () {
        console.log("closed");
    });

}).listen(8080);
console.log('listening on port 8080');
```

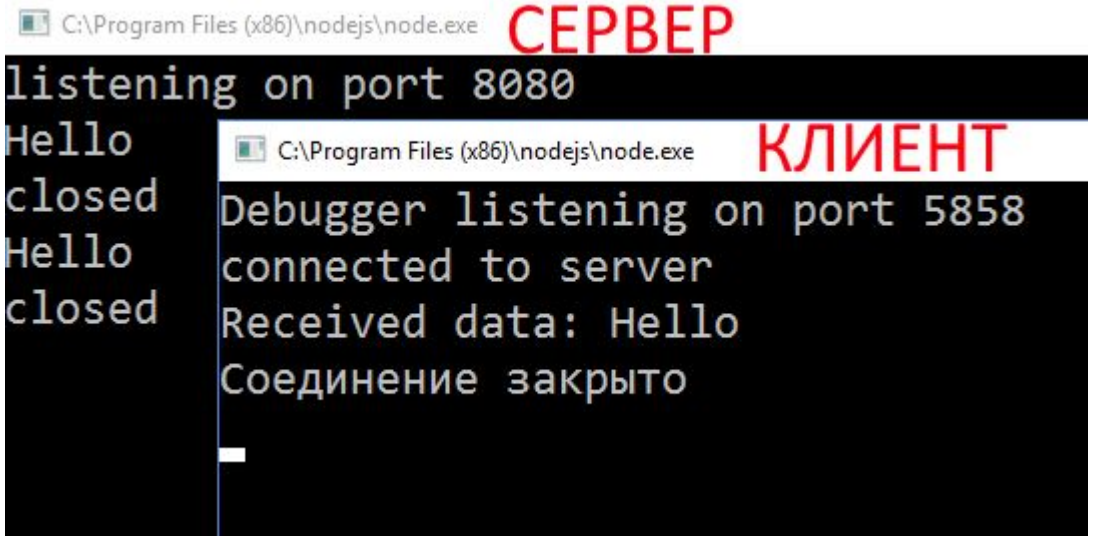



```
Received data: GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT
10.0; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/53.0.2785.143
Safari/537.36
Accept:
text/html,application/xhtml+xml,applicat
ion/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: ru-RU,ru;q=0.8,en-
US;q=0.6,en;q=0.4
Cookie:                com=;
__atuvc=16%7C36%2C387%7C37%2C206%7C38%2C
197%7C39%2C25%7C40
```

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, l
 Gecko) Chrome/53.0.2785.143 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.6,en;q=0.4
Cookie:                ; __atuvc=16%7C36%2C387%7C37%2C206%7C38%2C197%7C39%2
%7C40


closed
GET /favicon.ico HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, l
 Gecko) Chrome/53.0.2785.143 Safari/537.36
Accept: */*
Referer: http://localhost:8080/
Accept-Encoding: gzip, deflate, sdch
```

Клиент TCP-сервера

```
var net = require('net');
var clientSocket = new net.Socket();
clientSocket.setEncoding('utf8');
clientSocket.connect('8080', 'localhost',
function () {
    console.log('connected to server');
    clientSocket.write('Hello');
});

clientSocket.on('data', function (data) {
    console.log(data);
});

clientSocket.on('close', function ()
{
    console.log('Соединение закрыто');
});
```



Socket & Buffer

При логировании полученных данных мы явным образом привели их значение к строковому виду. Зачем? Да вот как раз для этого случая! Дело в том, что данные, которыми сейчас обмениваются сокеты, представлены отнюдь не в текстовом формате. В этом нетрудно убедиться, убрав приведение типов

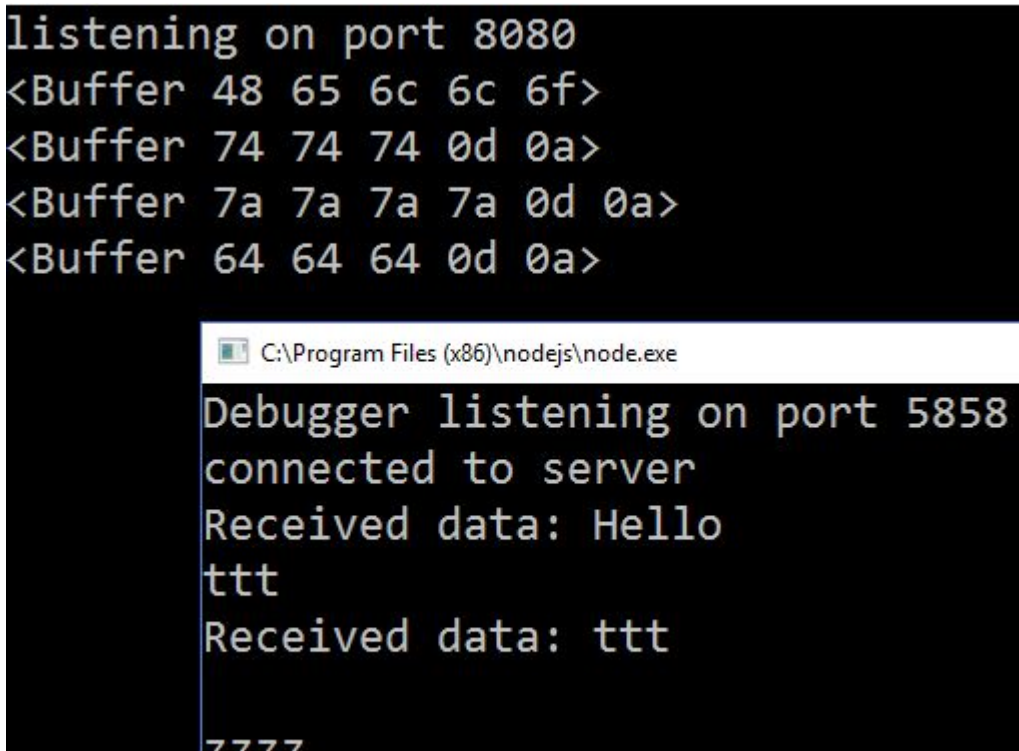
Да, это буфер.

<Buffer 48 65 6c 6c 6f>

```
socket.on('data', function (data) {
    console.log(data /*.toString()*/);
```

Ввод данных

```
clientSocket.connect('8080', 'localhost', function () {
    console.log('connected to server');
    clientSocket.write('Hello', function () {
        process.stdin.resume();
        process.stdin.on('data', function (data) {
            clientSocket.write(data);
        });
    });
});
```



TCP-чат

Чуть-чуть изменив код сервера, мы можем даже организовать нечто вроде TCP-чата

```
var net = require('net');
var clients = [];
var server = net.createServer(function (socket)
{
    clients[clients.length++] = socket;
    console.log('Соединение с ' + socket.remoteAddress + ' ' + socket.remotePort);
    socket.on('data', function (data) {
        console.log(data);
        clients.forEach(function (client) {
            client.write(data);
        });
    });
    socket.on('close', function () {
        console.log("closed");
    });

}).listen(8080);
console.log('listening on port 8080');
```

UDP Сервер

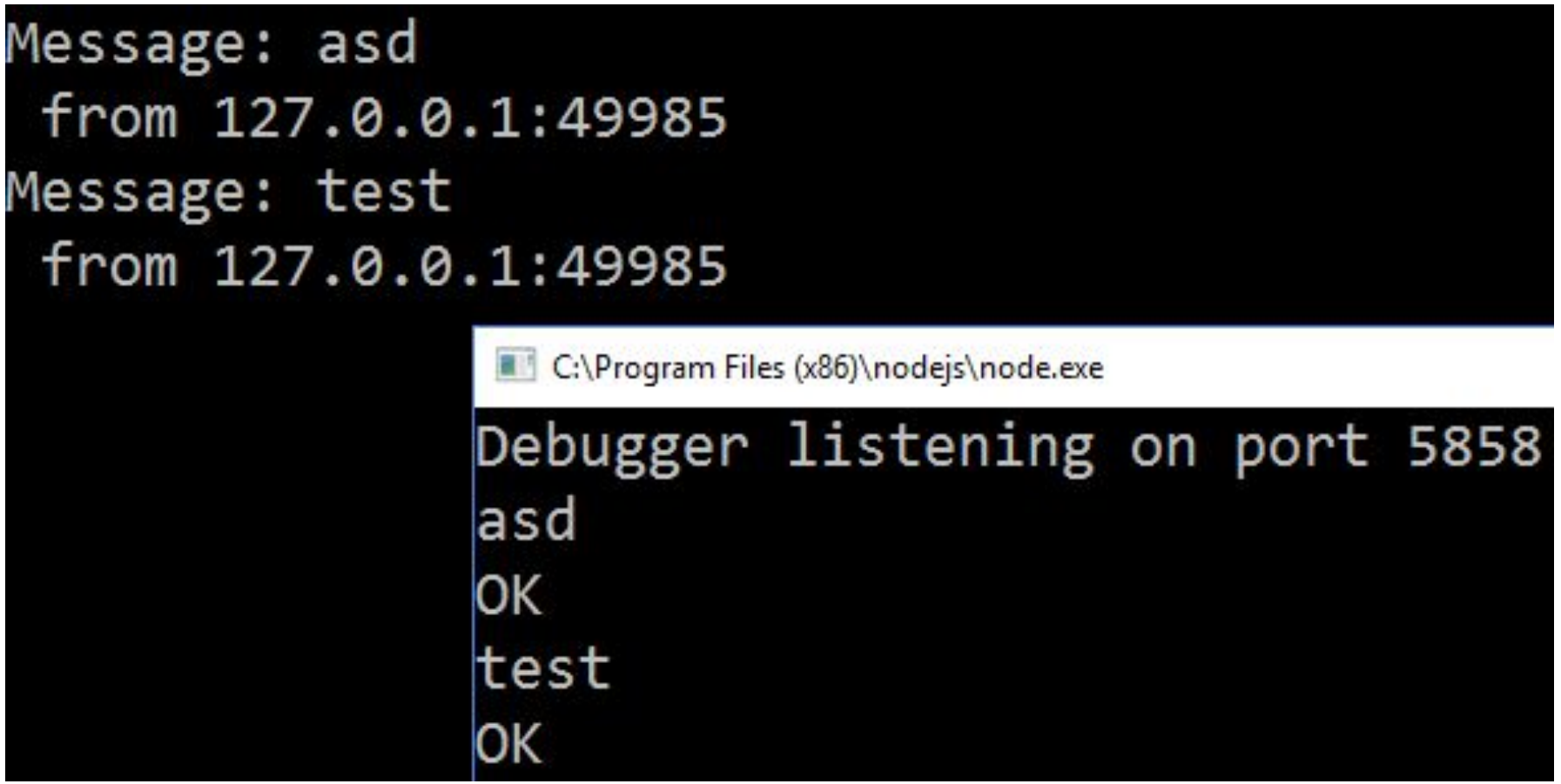
- Продемонстрируем работу протокола, создав простой сервер, принимающий UDP-пакеты.
- Обратите внимание, с объектом соединения мы не работаем, его просто нет.

```
var dgram = require('dgram');
var udpServer = dgram.createSocket("udp4");
udpServer.bind(8082);

udpServer.on("message", function (msg, info) {
    console.log("Message: " + msg + " from " + info.address + ":" + info.port);
});
```

UDP клиент

```
var dgram = require('dgram');
var client = dgram.createSocket("udp4");


process.stdin.resume();
process.stdin.on('data', function (data) {
    client.send(data, 0, data.length, 8082, "localhost", function (err, bytes) {
        if (err) console.log('eror: ' + err);
        else console.log('OK');
    });
});
```

Обратите внимание: проверяется только успех или неуспех отправки данных, получение отследить не представляется возможным. Зато тут же мы можем продемонстрировать преимущество протокола. Можно сколько угодно останавливать и запускать сервер - клиент останется в рабочем состоянии, и в моменты работы сервера данные будут доставлены.


Message: asd
from 127.0.0.1:49985
Message: test
from 127.0.0.1:49985
C:\Program Files (x86)\nodejs\node.exe
Debugger listening on port 5858
asd
OK
test
OK

HTTP - клиент

- А зачем? Зачем создавать HTTP-клиента, если HTTP-клиент - это браузер? Все просто - иногда необходимо сделать запрос по этому протоколу непосредственно из нашего приложения. Примеров можно привести много - запрос курсов валют, биржевых котировок от веб-сервисов, общающихся по этому протоколу, проведение онлайн-платежей, взаимодействие с платежными системами и тому подобные вполне распространенные случаи.

HTTP - клиент

```
var https = require('https');
var param = {
    hostname: 'api.privatbank.ua', path: '/p24api/pubinfo?exchange&coursid=3', port:
443, method: 'GET'
}
var req = https.request(param, function (res) {
    console.log('STATUS: ' + res.statusCode);
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
    });
});
req.on('error', function (e) { console.log('problem with request: ' + e.message); });
req.end();
```

HTTP - клиент

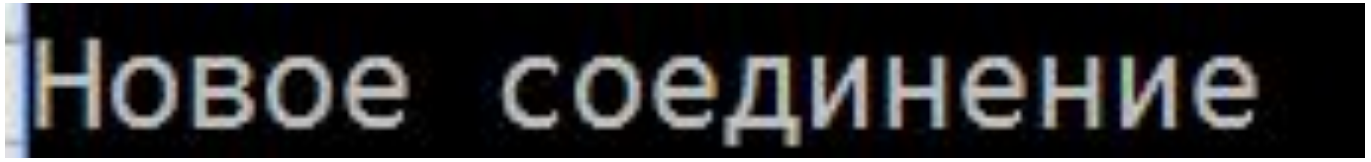
```
STATUS: 200
BODY: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><exchangerates><row
><exchangerate ccy="EUR" base_ccy="UAH" buy="28.86727" sale="29.12441"/></row><r
ow><exchangerate ccy="RUR" base_ccy="UAH" buy="0.41428" sale="0.41441"/></row><r
ow><exchangerate ccy="USD" base_ccy="UAH" buy="25.86442" sale="25.92062"/></row>
</exchangerates>
```

Модуль WS

- Для того чтобы начать работать с веб-сокетами, нужны всего две вещи - браузер, поддерживающий WebSocket, и сервер, реализующий эту технологию.
- На стороне браузера все просто - WebSockets API входит в семейство JavaScript-интерфейсов, условно объединенных под названием HTML5, и поддерживается большинством современных версий браузеров.
- Серверная составляющая WebSockets присутствует в node «ИЗ коробки». Ну, почти так - все, что нужно сделать, - доставить соответствующий модуль: **npm install ws**

WebSocket Server

- var webSocketServer = new require('ws');
- var webSocketServer = new webSocketServer.Server({ port: 8080 });
- webSocketServer.on('connection', function (ws) { console.log("Новое соединение"); })



WebSocket Client

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        onload = function () { var ws = new WebSocket("ws://localhost:8080"); }
    </script>
</head>
<body>
    index
</body>
</html>
```

WebSocket Server

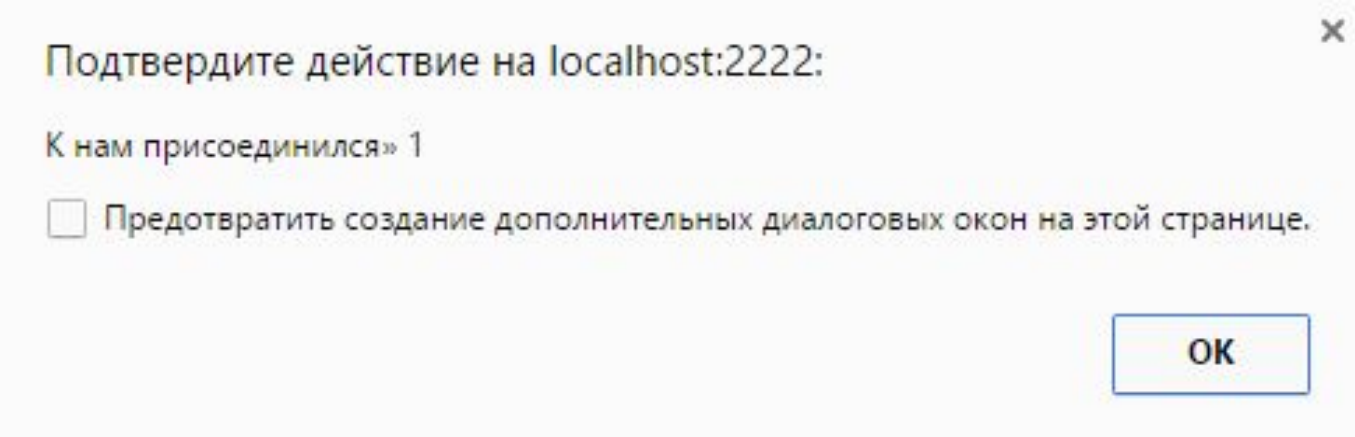
- Это прекрасно, но пока полнодуплексного соединения, мягко говоря, не наблюдается. Что естественно - взаимодействия по направлению от сервера к клиенту у нас пока не происходит. Изменим код сервера:

```
var webSocketServer = new require('ws');
var wss = new webSocketServer.Server({ port: 8080 });
var clients = [];
wss.on('connection', function (ws) {
    var id = clients.length;
    clients[id] = ws;
    console.log("Новое соединение № " + id);
    clients[id].send("Приветствуем! ваш идентификатор " + id);
    for (var key in clients) {
        if (key != id) {
            clients[key].send("К нам присоединился» " + id);
        }
    }
    console.log(clients);
});
```

WebSocket Client

- На клиенте напишем код для приема сообщений:

```
onload = function ()
        {
            var ws = new WebSocket("ws://localhost:8080");
            ws.onmessage = function (event) { alert(event.data); }
        }
```



Реализация WebSocket-чата

Сначала сделаем простую форму для отправки сообщений и javaScript-обработчик

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        onload = function ()
        {
            var ws = new WebSocket("ws://localhost:8080");
            ws.onmessage = function (event) { alert(event.data); }
        }
        document.forms.push.onsubmit = function() { ws.send(this.message.value); return false;  };
    </script>
</head>
<body>
    <form name="push">
        <input type="hidden" name="userid" />
        <input type="text" name="message" />
        <input type="submit" value="Отправить" />
    </form>
</body>
</html>
```

Реализация WebSocket-чата

```
var webSocketServer = new require('ws');
var wss = new webSocketServer.Server({ port: 8080 });
var clients = [];
wss.on('connection', function (ws) {
    var id = clients.length;
    clients[id] = ws;
    console.log("Новое соединение № " + id);
    clients[id].send("Приветствуем! ваш идентификатор " + id);
    for (var key in clients) {
        if (key != id) {
            clients[key].send("К нам присоединился» " + id); }
    }
    console.log(clients);
 });

wss.on('message', function (message) {
    console.log('получено сообщение' + message);
    for (var key in clients) {
        if (key != id) {
            clients[key].send(message);
        }
    }
});
```

Выводы

- В этой лекции была рассмотрена работа с файловой системой. Были изучены процедуры обхода каталогов, чтения, записи на низком и на высокому уровнях, копирования и удаления файлов.
- Также была изучена процедура для отслеживания состояния файлов.
- Рассмотрены примеры работы с потоками ввода-вывода.
- Был созданы TCP-сервер и клиент, UPD-сервер и клиент, WebSocket-сервер и клиент.

Список литературы

- Сухов К. К. Node.js. Путеводитель по технологии. - М.: ДМК Пресс, 2015. 416 с.: ил
- https://ru.wikipedia.org/wiki/Node.js
- Пауэрс Ш. Изучаем Node.js. - СПб.: Питер, 2014. - 400 с: ил. - (Серия "Бестселлеры O'Reilly").