



# NODE.JS

# Содержание

|                                |    |
|--------------------------------|----|
| Установка Node.js              | 3  |
| Модули                         | 5  |
| Как работает Node?             | 7  |
| События                        | 9  |
| Работа с файлами               | 9  |
| Создание консольных приложений | 12 |
| Простой сайт на Node.js        | 17 |
| Сетевые запросы                | 23 |
| Express                        | 25 |
| Полезные ссылки                | 28 |

# Установка Node.js



**Node или Node.js** — программная платформа, основанная на движке V8, который превращает язык JavaScript из узко специализированного, в язык общегоназначения.

**Применяется преимущественно на сервере, выполняя роль веб-сервера**, но есть возможность разрабатывать на Node.js и десктопные приложения (при помощи **NW.js** или **Electron** для Linux, Windows и Mac OS).

В основе Node.js лежит событийно-ориентированное и асинхронное программирование с неблокирующим вводом/выводом.

*Наиболее частое применение Node.js находит при разработке: чатов и систем обмена мгновенными сообщениями; многопользовательских игр в реальном времени; сетевых сервисов для сбора и отправки больших объемов информации.*

Также хорошо подходит для создания стандартных веб-приложений. Ее используют для создания консольных утилит, такие популярные системы сборки для front-end как Grunt.js и Gulp.js созданы с помощью Node.

**Чтобы установить Node на компьютер**, вам нужно пойти на сайт <https://nodejs.org/en/> и скачать LTS или текущую версию (на момент написания методички это были версии v4.4.4 и v6.2.0 соответственно).

## **А что делать, если вы хотите установить эти две версии сразу?**

Для этого есть специальная утилита **nvm (Node Version Manager)** — это скрипт, который позволяет устанавливать, переключать и удалять версии Node.js т.е. даёт возможность держать на одной машине любое количество версий Node.js. Как обычно, работа под Windows совсем не радужна, но эта **статья** вам поможет.

Чтобы проверить работоспособность после установки наберите в консоли:

```
$ node
```

Вы попадете в интерактивную консоль node, прямо в которой можно набирать и выполнять команды JavaScript.

```
> 1+2  
3  
>
```

В этом режиме в консоль просто выводится результат набранного выражения.

Давайте запишем, для примера, некий код в файл с именем *start.js*:

```
var text = 'Hello student from Loftschool!';  
console.log(text);
```

И запустим его из консоли в той директории, где он был создан, следующей командой:

```
$ node start.js
```

В консоли должна появиться надпись:

```
Hello student from Loftschool!
```

```
Hello student from Loftschool!  
[Finished in 3.3s]
```

Вот мы и создали свой первый скрипт.

# Модули

Для подключения к вашим скриптам дополнительных функций в Node.js существует удобная система управления модулями **NPM**. По сути это публичный репозиторий созданных при помощи Node.js дополнительных программных модулей.

*Команда `npm` позволяет легко устанавливать, удалять или обновлять нужные вам модули, автоматически учитывая при этом все зависимости выбранного вами модуля от других.*

Установка модуля производится командой:

```
npm install *имя модуля* [*ключи*]
```

Для установки модуля будет использована поддиректория `node_modules`.

Хотя `node_modules` и содержит все необходимые для запуска зависимости, распространять исходный код вместе с ней не принято, т.к. в ней может храниться большое количество файлов, которые занимают ощутимый объем и это неудобно.

С учетом того, что все публичные NPM-модули можно легко установить с помощью `npm`, достаточно создать и написать для вашей программы файл `package.json` с перечнем всех необходимых для работы зависимостей и потом просто, на новом месте, например, установить все нужные модули командой:

```
$ npm install
```

Более подробно о работе с самим NPM вы можете прочитать в соответствующих методических указаниях.

Node.js работает с системой подключения модулей **CommonJS**. В структурном плане, CommonJS-модуль представляет собой готовый к новому использованию фрагмент JavaScript-кода, который экспортирует специальные объекты, доступные для использования в любом зависимом коде. CommonJS используется как формат JavaScript-модулей так же и на front-end. Две главных идеи CommonJS-модулей: **объект exports**, содержащий то, что модуль хочет сделать доступным для других частей системы, и **функцию require**, которая используется одними модулями для импорта объекта exports из других.

Начиная с версии 6.x Node.js так же поддерживает подключение модулей согласно стандарту ECMAScript-2015.

Давайте попробуем что-нибудь подключить. Например, модуль [colors](#) для предыдущего скрипта, и немного перепишем его. Наш скрипт станет выглядеть так:

```
var colors = require('colors');  
var text = 'Hello student from Loftschool!';  
console.log(text.rainbow);
```

Выполним команды в консоли:

```
npm install colors  
node start.js
```

Вот что вы увидите:

```
D:\WebDir\Node_exp\app\color>node start.js  
Hello student from Loftschool!
```

И, наверняка, почувствуете что-то [такое](#).

# Как работает Node?

В основе Node лежит библиотека **libuv**, реализующая цикл событий **event loop**.

Мы знаем, что объявленная переменная в скрипте автоматически становится глобальной. В Node она остается *локальной для текущего модуля* и чтобы сделать ее глобальной, надо объявить ее как свойство объекта Global:

```
global.foo = 3;
```

Фактически, объект **Global** — это аналог объекта window из браузера.

Метод **require**, служащий для подключения модулей, не является глобальным и *локален* для каждого модуля.

Также *локальными* для каждого модуля являются:

**module.export** – объект, отвечающий за то, что именно будет экспортировать модуль при использовании require;

**\_\_filename** – имя файла исполняемого скрипта;

**\_\_dirname** – абсолютный путь до исполняемого скрипта.

В секцию *Global* входят такие важные элементы как:

**Class: Buffer** – объект используется для операций с бинарными данными.

**Process** – объект процесса, большая часть данных находится именно здесь.

Приведем пример работы некоторых из них. Назначение понятно из названий:

```
console.log(process.execPath);  
console.log(process.version);  
console.log(process.platform);  
console.log(process.arch);  
console.log(process.title);  
console.log(process.pid);
```

```
e:\Program Files\nodejs\node.exe  
v5.5.0  
win32  
ia32  
MINGW32:/d/WebDir/Node_exp/app/color  
3240
```

Свойство **process.argv** содержит массив аргументов командной строки. Первым аргументом будет имя исполняемого приложения node, вторым имя самого исполняемого сценария и только потом сами параметры.

Для работы с каталогами есть следующие свойства – **process.cwd()** возвращает текущий рабочий каталог, **process.chdir()** выполняет переход в другой каталог.

Команда **process.exit()** завершает процесс с указанным в качестве аргумента кодом: 0 – успешный код, 1 – код с ошибкой.

Важный метод **process.nextTick(fn)** запланирует выполнение указанной функции таким образом, что указанная функция будет выполнена после окончания текущей фазы (текущего исполняемого кода), но перед началом следующей фазы eventloop.

```
process.nextTick(function() {  
  console.log('NextTick callback');  
})
```

Объект Process содержит еще много свойств и методов, с которыми можно ознакомиться в [справке](#).



## События

За события в Node.js отвечает специальный модуль **events**.

Назначать объекту обработчик события следует методом **addListener(event, listener)**. Аргументы – это имя события *event*, в camelCase формате и *listener* — функция обратного вызова, обработчик события. Для этого метода есть более короткая запись **on()**.

Удалить обработчик можно методом **removeListener(event, listener)**.

А метод **emit(event, [args])** позволяет событиям срабатывать.

Например, событие **'exit'** отправляется перед завершением работы Node.

```
process.on('exit' , function() {  
    console.log('Bye!');  
});
```

## Работа с файлами

Модуль **FileSystem** отвечает за работу с файлами. Инициализация модуля происходит следующим образом:

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

**fs.exists(path, callback)** - проверка существования файла.

**fs.readFile(filename, [options], callback)** - чтение файла целиком

**fs.writeFile(filename, data, [options], callback)**

- запись файла целиком

**fs.appendFile(filename, data, [options], callback)** - добавление в файл

**fs.rename(oldPath, newPath, callback)** - переименование файла.

**fs.unlink(path, callback)** - удаление файла.

Функции **callback** принимают как минимум один параметр *err*, который равен *null* при успешном выполнении команды или содержит информацию

об ошибке. Помимо этого при вызове **readFile** передается параметр *data*, который содержит уже упоминавшийся объект типа *Buffer*, содержащий последовательность прочитанных байтов. Чтобы работать с ним как со строкой, нужно его конвертировать методом **toString()**

```
fs.readFile('readme.txt', function (err, data) {
  if (err) {
    throw err;
  }
  console.log(data.toString());
});
```

Также почти все методы модуля **fs** имеют синхронные версии функции, оканчивающиеся на **Sync**. Этим функциям не нужны callback, т.к. они являются блокирующими и поэтому рекомендованы к применению, только если это требует текущая задача. Давайте напишем программу, которая будет читать каталог и выводить его содержимое, а для файлов выводить их размер и дату последнего изменения.

```
var fs = require('fs'),
    path = require('path'),
    dir = process.cwd(),
    files = fs.readdirSync(dir);

console.log('Name \t Size \t Date \n');

files.forEach(function (filename) {
  var fullname = path.join(dir, filename),
      stats = fs.statSync(fullname);
  if (stats.isDirectory()) {
    console.log(filename + '\t DIR \t' + stats.mtime + '\n');
  } else {
    console.log(filename + '\t' + stats.size + '\t' + stats.
mtime + '\n');
  }
});
```

Давайте разберем эту программу подробно. В начале мы подключаем два стандартных модуля:

```
var fs = require('fs'),  
    path = require('path')
```

Первый отвечает за запись и чтения файлов, а модуль `path` за работу с путями файлов. В переменную `dir` мы с помощью метода `process.cwd()` сохраняем текущую директорию и тут же в переменную `files` считываем в синхронном режиме `fs.readdirSync(dir)` все файлы из текущего каталога. В синхронном потому, что нам надо получить весь список файлов и поддиректорий из текущей директории, прежде чем приступить к ее анализу. Выводим шапку нашей будущей таблички:

```
console.log('Name \t Size \t Date \n');
```

И потом методом `forEach` по массиву `files`, прочитанных элементов директории, проходимся и выводим в консоль информацию об элементах. Через метод `path.join` соединяем пути к файлу, и в переменную `stats` записываем информацию о текущем файле. Мы выводим `stats.mtime` — время создания файла и `stats.size` для определения размера файла. С помощью `stats.isDirectory()` определяем является ли элемент директорией и если да, для него не выводим размер, а ключевое слово DIR.

```
КРАБАТ@KRBAT /d/WebDir/Node_exp/app/catalog
% node app.js
Name      Size      Date
app.js    454      Tue May 17 2016 00:45:24 GMT+0300 (Восточная Европа (лето))
minargv.js 74      Mon May 23 2016 14:41:05 GMT+0300 (Восточная Европа (лето))
readme.txt 6      Mon May 23 2016 14:30:29 GMT+0300 (Восточная Европа (лето))
temp      DIR      Mon May 23 2016 14:29:50 GMT+0300 (Восточная Европа (лето))
TEST      DIR      Mon May 16 2016 23:58:42 GMT+0300 (Восточная Европа (лето))
test.js   173      Tue May 24 2016 12:44:56 GMT+0300 (Восточная Европа (лето))

КРАБАТ@KRBAT /d/WebDir/Node_exp/app/catalog
$ |
```

## Создание консольных приложений

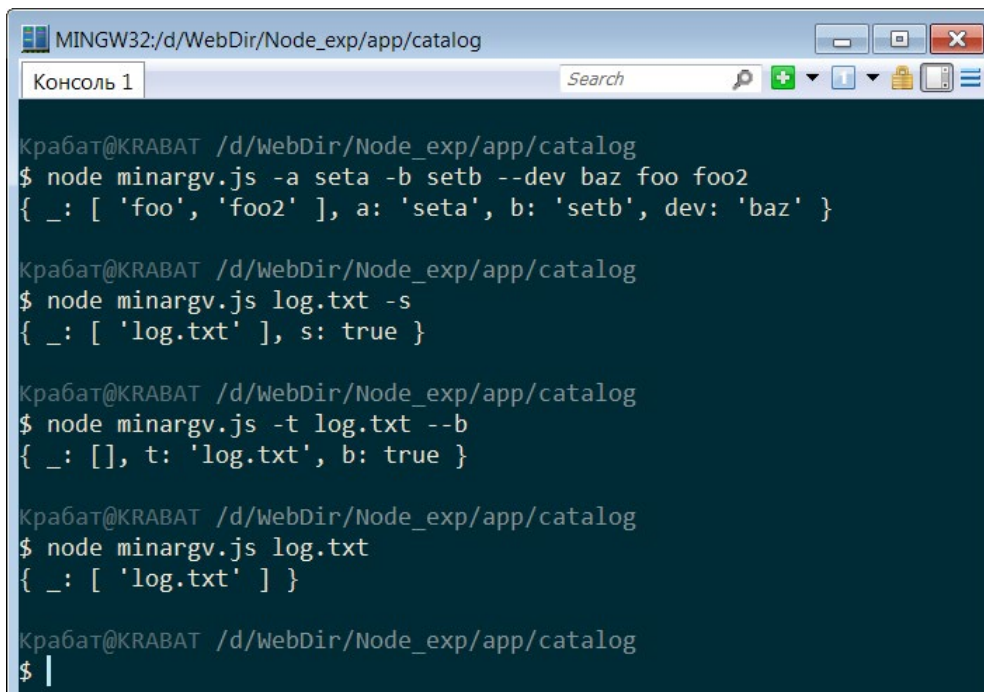
Традиционно, самым простым способом управления консольными приложениями является передача параметров из консольной строки при их запуске.

Как выше отмечалось, переданные в скрипт параметры доступны в массиве `process.argv`: `["node", ".../youscript.js", "param 1", "param 2", ...]`. И чтобы получить параметры, нам надо выполнить `process.argv.slice(2)`, который вернет все разделенные пробелами параметры: `["param 1", "param 2", ...]`

Так как обрабатывать вручную всевозможные комбинации параметров и их форматы неудобно, для этих целей обычно используют тот или иной npm-модуль. Один из популярных – это модуль **minimist**, в котором представлен хороший функционал для этих целей.

Вот как он работает:

```
var argv = require('minimist')(process.argv.slice(2));
console.dir(argv);
```



```
MINGW32: d/WebDir/Node_exp/app/catalog
Крабат@KRABAT /d/WebDir/Node_exp/app/catalog
$ node minargv.js -a seta -b setb --dev baz foo foo2
{ _: [ 'foo', 'foo2' ], a: 'seta', b: 'setb', dev: 'baz' }

Крабат@KRABAT /d/WebDir/Node_exp/app/catalog
$ node minargv.js log.txt -s
{ _: [ 'log.txt' ], s: true }

Крабат@KRABAT /d/WebDir/Node_exp/app/catalog
$ node minargv.js -t log.txt --b
{ _: [], t: 'log.txt', b: true }

Крабат@KRABAT /d/WebDir/Node_exp/app/catalog
$ node minargv.js log.txt
{ _: [ 'log.txt' ] }

Крабат@KRABAT /d/WebDir/Node_exp/app/catalog
$ |
```

Способ консольного ввода это построчный ввод данных. Для этого используется стандартный модуль `readline`.

Инициализация:

```
var readline = require('readline');
var rl = readline.createInterface({
  input: process.stdin, // ввод из стандартного потока
  output: process.stdout // вывод в стандартный поток
});
```

Обработка каждой введенной строки:

```
rl.on('line', function (cmd) {
  console.log('You just typed: ' + cmd);
});
```

Получение ответа на вопрос (аналогично `prompt` в браузере):

```
rl.question('What is your favorite food?', function (answer) {
  console.log('Oh, so your favorite food is ' + answer);
});
```

Пауза (блокирование ввода):

```
rl.pause()
```

Разблокирование ввода:

```
rl.resume()
```

Окончание работы с интерфейсом `readline`:

```
rl.close()
```

Чтобы закрепить материал, давайте напишем небольшое приложение – «Угадай число», где необходимо будет угадать задуманное программой число от 1 до 10 и программа в конце выведет, за сколько шагов это было сделано.

Нам понадобятся стандартные модули *fs*, *readline* и нестандартный, а значит, его надо установить с помощью *npm*, модуль *minimist*. Приведем листинг программы и разберем ее.

```
var readline = require('readline'),
    argv = require('minimist')(process.argv.slice(2)),
    fs = require('fs'),
    mind, count, rl, logfile;

function init() {
  // получим случайное число от 1 до 10
  mind = Math.floor(Math.random() * 10) + 1;
  // обнулим счетчик количества угадываний
  count = 0;
  // установим ввод и вывод в стандартные потоки
  rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });
  // запомним имя файла для логов, если он есть
  logfile = argv['_'][0];
}

function game() {
  function log(data) {
    if (logfile !== undefined)
      fs.appendFile(logfile, data + "\n");
  }

  function valid(value) {
    if (isNaN(value)) {
      console.log('Введите число!');
      return false;
    }

    if (value < 1 || value > 10) {
      console.log('Число должно лежать в заданном диапазоне!');
      return false;
    }
  }
}
```

```

        return true;
    }

    rl.question('Введите любое число от 1 до 10, чтобы угадать задуманное: ',
        function (value) {
            var a = +value;
            if (!valid(a)) {
                // если валидацию не прошли – запускаем игру заново
                game();
            } else {
                count += 1;
                if (a === mind) {
                    console.log('Поздравляем! Вы угадали число за %d
шага(ов)', count);
                    log('Поздравляем! Вы угадали число за ' + count + '
шага(ов)');
                    // угадали и закрыли экземпляр Interface, конец программы
                    rl.close();
                } else {
                    console.log('Вы не угадали, еще попытка');
                    game();
                }
            }
        });
}

init();
game();

```

Вся программа состоит из двух функций. Вызова функции инициализации `init();` и вызов самой функции игры `game();`, которая будет вызывать себя рекурсивно при неверно угаданном числе. Инициализация довольно простая и особого пояснения не требует (все ясно из комментариев). Внутри функции `game()` мы описали еще две вспомогательные функции. Одна будет писать результат игры в файл, если он передан через строку параметров:

```

function log(data) {
    if (logfile !== undefined)
        fs.appendFile(logfile, data + "\n");
}

```

А вторая — `valid` будет проверять валидны ли значения, которые вводит пользователь в консоли.

Сама программа состоит в вызове метода

```

rl.question('Введите любое число от 1 до 10, чтобы угадать задуманное: ',
    function (value) {...});

```

который прослушивает консоль и при вводе значения вызывает callback функцию, которая обрабатывает введенное значение.

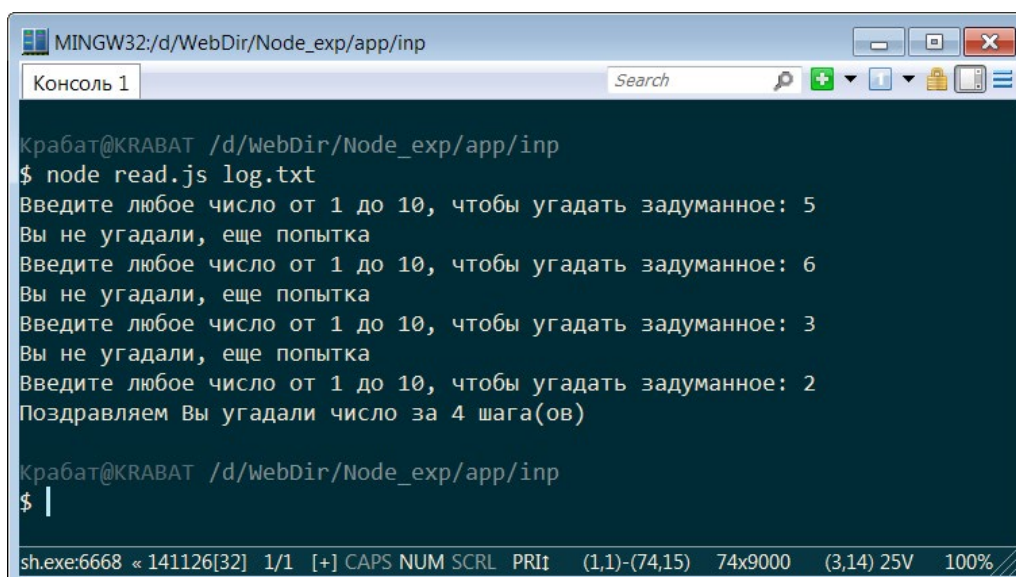


Если мы не проходим валидацию, то запускаем функцию игры заново:

```
if (!valid(a)) {  
    // если валидацию не прошли – запускаем игру заново  
    game();  
}
```

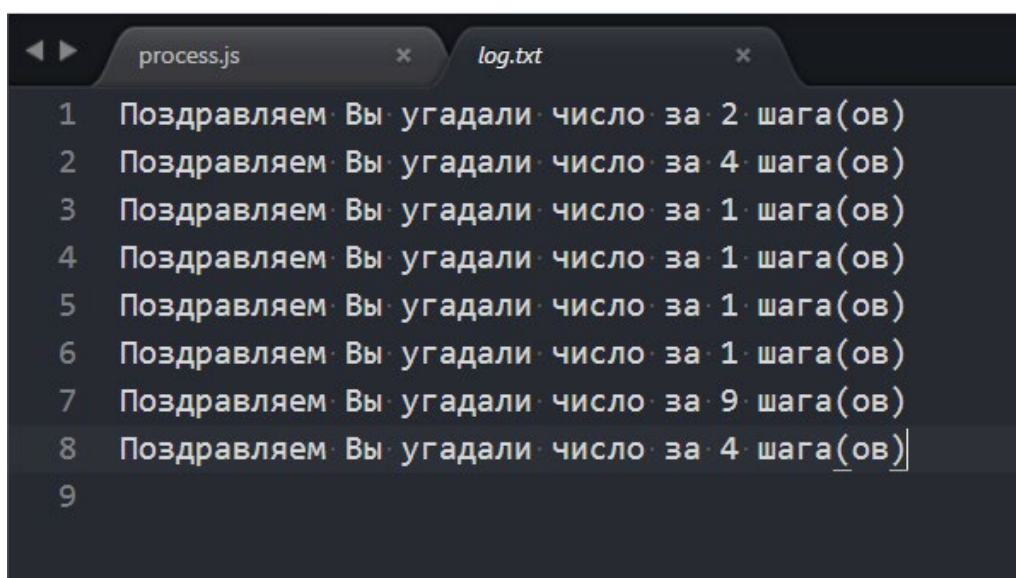
Если валидация пройдена, то мы увеличиваем счетчик на 1, т.е. засчитываем попытку `count += 1`; И сравниваем введенное значение с «задуманным». Если число угадано, то мы выводим поздравление и количество попыток, затраченное на игру, потом с помощью функции `log` пытаемся сохранить результат в файле, если это возможно и закрываем интерфейс ввода `rl.close()`; если же результат не совпал, выполняем рекурсию. Можете чуть улучшить этот пример, используя модуль `colors`, чтобы сделать вывод информации цветным по смыслу.

В результате у вас должно получиться следующее:



```
MINGW32:/d/WebDir/Node_exp/app/inp  
Консоль 1  
Крбат@KRABAT /d/WebDir/Node_exp/app/inp  
$ node read.js log.txt  
Введите любое число от 1 до 10, чтобы угадать задуманное: 5  
Вы не угадали, еще попытка  
Введите любое число от 1 до 10, чтобы угадать задуманное: 6  
Вы не угадали, еще попытка  
Введите любое число от 1 до 10, чтобы угадать задуманное: 3  
Вы не угадали, еще попытка  
Введите любое число от 1 до 10, чтобы угадать задуманное: 2  
Поздравляем Вы угадали число за 4 шага(ов)  
  
Крбат@KRABAT /d/WebDir/Node_exp/app/inp  
$ |  
sh.exe:6668 « 141126[32] 1/1 [+] CAPS NUM SCRL PRIi (1,1)-(74,15) 74x9000 (3,14) 25V 100%
```

А в файле `log.txt` информация будет храниться подобным образом:



```
process.js x log.txt x  
1 Поздравляем Вы угадали число за 2 шага(ов)  
2 Поздравляем Вы угадали число за 4 шага(ов)  
3 Поздравляем Вы угадали число за 1 шага(ов)  
4 Поздравляем Вы угадали число за 1 шага(ов)  
5 Поздравляем Вы угадали число за 1 шага(ов)  
6 Поздравляем Вы угадали число за 1 шага(ов)  
7 Поздравляем Вы угадали число за 9 шага(ов)  
8 Поздравляем Вы угадали число за 4 шага(ов)  
9
```

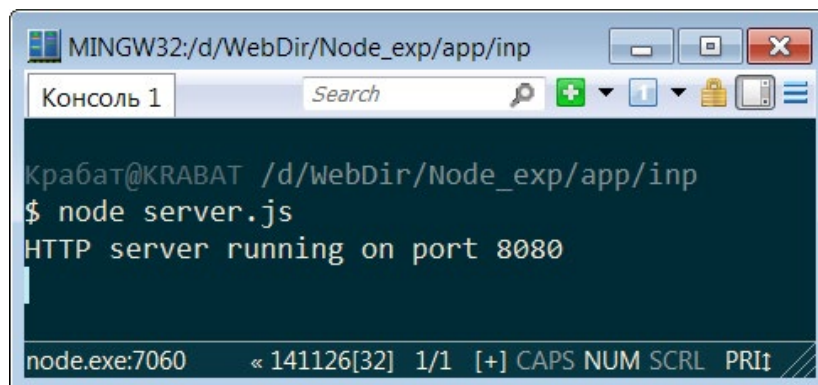


# Простой сайт на Node.js

Веб-сервер на Node.js состоит из нескольких строчек кода:

```
var http = require('http');
http.createServer(function(req, res) {
  console.log('HTTP server running');
}).listen(8080);
```

Что здесь происходит? Это легко понять. Сначала мы запрашиваем модуль `'http'`, затем создаем сервер `http.createServer` и запускаем его `listen` на порту `8080`. Метод `createServer` объекта `http` принимает в качестве аргумента анонимную функцию обратного вызова, аргументами которой, в свою очередь служат объекты `req` – request и `res` – response. Они соответствуют поступавшему HTTP-запросу и отдаваемому HTTP-ответу. Если мы запустим в консоли наш скрипт `server.js` потом в браузере обратимся по адресу <http://localhost:8080/>, то в консоли будет следующее:



Но в самом браузере мы ничего пока не увидим. Остановим выполнение скрипта комбинацией **Ctrl+C** и допишем следующий код:

```
var http = require('http');
http.createServer(function(req, res) {
  console.log('HTTP server running');
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('<h1>Hello student from Loftschool!</h1>');
}).listen(8080);
```

Запустим опять скрипт и в браузере мы наконец-то увидим результат:



Как мы видим, HTTP-запрос не является инициатором запуска всей программы. Создается Javascript-объект и ждет запросы, при поступлении которых срабатывает связанная с этим событием анонимная функция. В принципе неплохо, но мы уже работали с файлами и давайте заставим сервер отдавать нам страницу HTML. Создадим простую веб-страницу:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>Loftschool</title>
  <style>
    h1 {
      color: blue;
    }

    h1:hover {
      color: #ccc;
    }
  </style>
</head>

<body>
<h1>My first page</h1>
</body>
</html>
```

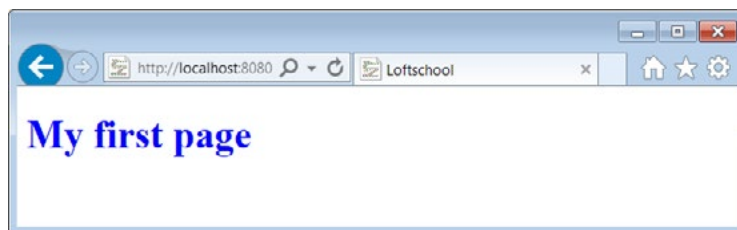
Модифицируем серверный скрипт:

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

http.createServer(function (req, res) {
  fs.readFile('index.html', 'utf8', function (err, data) {
    if (err) {
      res.writeHead(404, {
        'Content-Type': 'text/html'
      });
      res.end('Errorr load index.html');
    } else {
      res.writeHead(200, {
        'Content-Type': 'text/html'
      });
      res.end(data);
    }
  })
}).listen(8080);

console.log('HTTP server running on port 8080');
```

Выполним в консоли команду `node server.js` и увидим:



Но все это полумеры, мы не можем так подключить стили (только внутренние), скрипты и картинки.

Создадим следующую простую структуру сайта:

| Имя     | Дата изменения   | Тип               | Размер |
|---------|------------------|-------------------|--------|
| css     | 19.05.2016 18:55 | Папка с файлами   |        |
| img     | 19.05.2016 15:24 | Папка с файлами   |        |
| js      | 19.05.2016 15:16 | Папка с файлами   |        |
| favicon | 08.12.2015 21:31 | Значок            | 2 КБ   |
| index   | 19.05.2016 18:52 | Chrome HTML Do... | 4 КБ   |
| server  | 23.05.2016 18:22 | Файл ".JS"        | 2 КБ   |

Файл **index.html** содержит такую разметку:

```
<!DOCTYPE html>
<html lang="ru-RU">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="x-ua-compatible" content="ie=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="favicon.ico" type="image/x-icon">
  <link rel="stylesheet" href="/css/normalize.css">
  <link rel="stylesheet" href="/css/style.css">
  <title>Test</title>
</head>

<body>
<div class="wrapper">
  ...
</div>
<footer class="page__bottom">
  ...
</footer>
<script src="js/main.js"></script>
</body>

</html>
```

Если мы попробуем отобразить его предыдущим скриптом, он выведет только html-контент, без стилей, картинок и javascript-скриптов. Скрипт, который обработает все правильно, будет следующим:

```

var http = require('http'),
    fs = require('fs'),
    url = require('url'),
    path = require('path'),
    typeMime = {
      '.html': 'text/html',
      '.htm': 'text/html',
      '.js': 'text/javascript',
      '.css': 'text/css',
      '.png': 'image/png',
      '.jpg': 'image/jpeg'
    };

http.createServer(function (req, res) {
  var _url = url.parse(req.url),
      filename = _url.pathname.substring(1),
      extname,
      type,
      img;

  if (_url.pathname === '/') {
    filename = 'index.html';
  }

  extname = path.extname(filename);
  type = typeMime[path.extname(filename)];

  if ((extname === '.png') || (extname === '.jpg')) {
    img = fs.readFileSync(filename);
    res.writeHead(200, {
      'Content-Type': type
    });
    res.write(img, 'hex');
    res.end();
  } else {
    fs.readFile(filename, 'utf8', function (err, content) {
      if (err) {
        res.writeHead(404, {
          'Content-Type': 'text/plain; charset=utf-8'
        });
        res.write(err.message);
        res.end();
      } else {
        res.writeHead(200, {
          'Content-Type': type
        });
        res.write(content);
        res.end();
      }
    })
  }
}).listen(8080);

```

Здесь мы видим *объект с mime-типами*, который позволит нам загружать разнообразный контент:

```
typeMime = {
  '.html': 'text/html',
  '.htm': 'text/html',
  '.js': 'text/javascript',
  '.css': 'text/css',
  '.png': 'image/png',
  '.jpg': 'image/jpeg'
};
```

У нас есть новые модули, один из них `path = require('path')`, который отвечает за различные операции с путями файлов. Так как в GET-запросах параметры передаются через url, для их обработки вы должны проанализировать эту строку. Удобно сделать это с помощью стандартного модуля url и его функции `parse` `_url = url.parse(req.url)`.

Находим имя файла, к которому произошел HTTP-запрос:

```
filename = _url.pathname.substring(1),
```

Если это обращение к корню сайта, то вызывать будем **index.html**

```
if (_url.pathname === '/') {
  filename = 'index.html';
}
```

Далее находим расширение файла, на который поступил запрос, и выбираем тут же ему mime-тип в переменную `type`

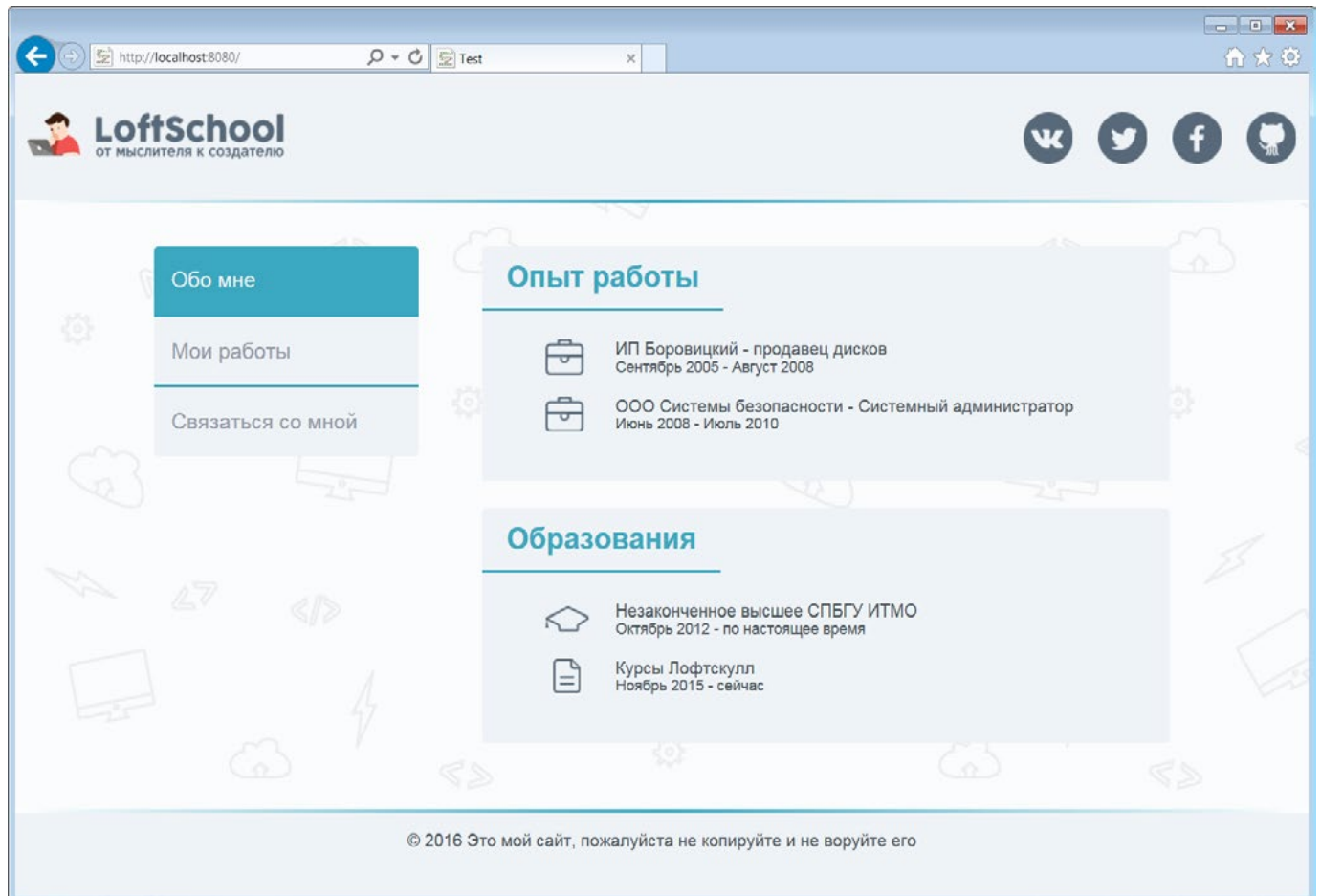
```
extname = path.extname(filename);
type = typeMime[path.extname(filename)];
```

Первым делом проверяем, не пришел ли запрос на картинку. Поскольку это двоичные файлы, то мы в метод `write` вторым параметром передаем кодировку `'hex'`

```
if ((extname === '.png') || (extname === '.jpg')) {
  img = fs.readFileSync(filename);
  res.writeHead(200, {
    'Content-Type': type
  });
  res.write(img, 'hex');
  res.end();
}
```

Обратите внимание, что мы здесь проверяем только графические файлы с расширениями png и jpg.

А дальше идет, как и раньше подгрузка файлов, но с учетом mime-типа файла. Если все сделали верно, мы должны увидеть наш полноценный сайт со всеми стилями и картинками:



А так же работающим javascript в нашем main.js файле.

**i HTML1300: Произошел переход.**  
Файл: localhost:8080  
Run javascript

## Сетевые запросы

Стандартный модуль `http` содержит функцию **`get`** для отправки GET запросов и функцию **`request`** для отправки POST и прочих запросов.

Пример отправки GET запроса:

```
var http = require('http');
http.get("http://loftschool.com/", function(res) {
    console.log("Статус ответа: " + res.statusCode);
}).on('error', function(e) {
    console.log("Статус ошибки: " + e.message);
});
```

Пример отправки POST запроса:

```
var http = require('http');
var options = {
    hostname: 'loftschool.com',
    port: 80,
    path: '/',
    method: 'POST'
};
var req = http.request(options, function (res) {
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
    });
});
req.on('error', function (e) {
    console.log('Возникла проблема с ответом от сервера: ' + e.message);
});
req.write('data\n');
req.end();
```

В основном используют популярный и удобный `http`-модуль для работы с исходящими сетевыми запросами — **`request`**.

Пример отправки GET запроса:

```
var request = require('request');
request('http://loftschool.com/', function (err, res, body) {
  if (!err && res.statusCode == 200) {
    console.log(body)
  }
});
```

Мы напечатаем в консоль заглавную страницу нашей школы.

Пример отправки POST запроса:

```
var request = require('request');
request({
  method: 'POST',
  uri: 'http://loftschool.com/',
  form: {
    key: 'value'
  },
}, function (err, res, body) {
  if (err) {
    console.error(err);
  } else {
    console.log(body);
    console.log(res.statusCode);
  }
});
```

*Это модуль полезен тем, что позволят автоматически обрабатывать JSON, работать с учетом редиректов или без них, поддерживает BasicAuth и OAuth, прокси и, наконец, поддерживает cookies.*



# Express

**Express** — это минималистичный и гибкий веб-фреймворк для приложений Node.js, предоставляющий обширный набор функций для мобильных и веб-приложений.

Имея в своем распоряжении множество служебных методов HTTP и промежуточных обработчиков, создать надежный API можно быстро и легко.

*Express предоставляет тонкий слой фундаментальных функций веб-приложений, которые не мешают вам работать с давно знакомыми и любимыми вами функциями Node.js.*

**Установка.** Создайте каталог для своего приложения и сделайте его своим рабочим каталогом.

```
$ mkdir myapp  
$ cd myapp
```

С помощью команды **npm init** создайте файл *package.json* для своего приложения.

Теперь установите Express в каталоге *app* и сохраните его в списке зависимостей. Например:

```
$ npm install express --save
```

Для временной установки Express, без добавления его в список зависимостей, не указывайте опцию *--save*:

```
$ npm install express
```

В каталоге *myapp* создайте файл с именем *app.js* и добавьте следующий код:

```
var express = require('express');  
var app = express();  
  
app.get('/', function (req, res) {  
  res.send('Hello World!');  
});  
  
app.listen(3000, function () {  
  console.log('Example app listening on port 3000!');  
});
```

Приложение запускает сервер и слушает соединения на порте 3000. Приложение выдает ответ **'Hello World!'** на запросы, адресованные корневому URL (/) или маршруту. Для всех остальных путей ответом будет *404 Not Found*.

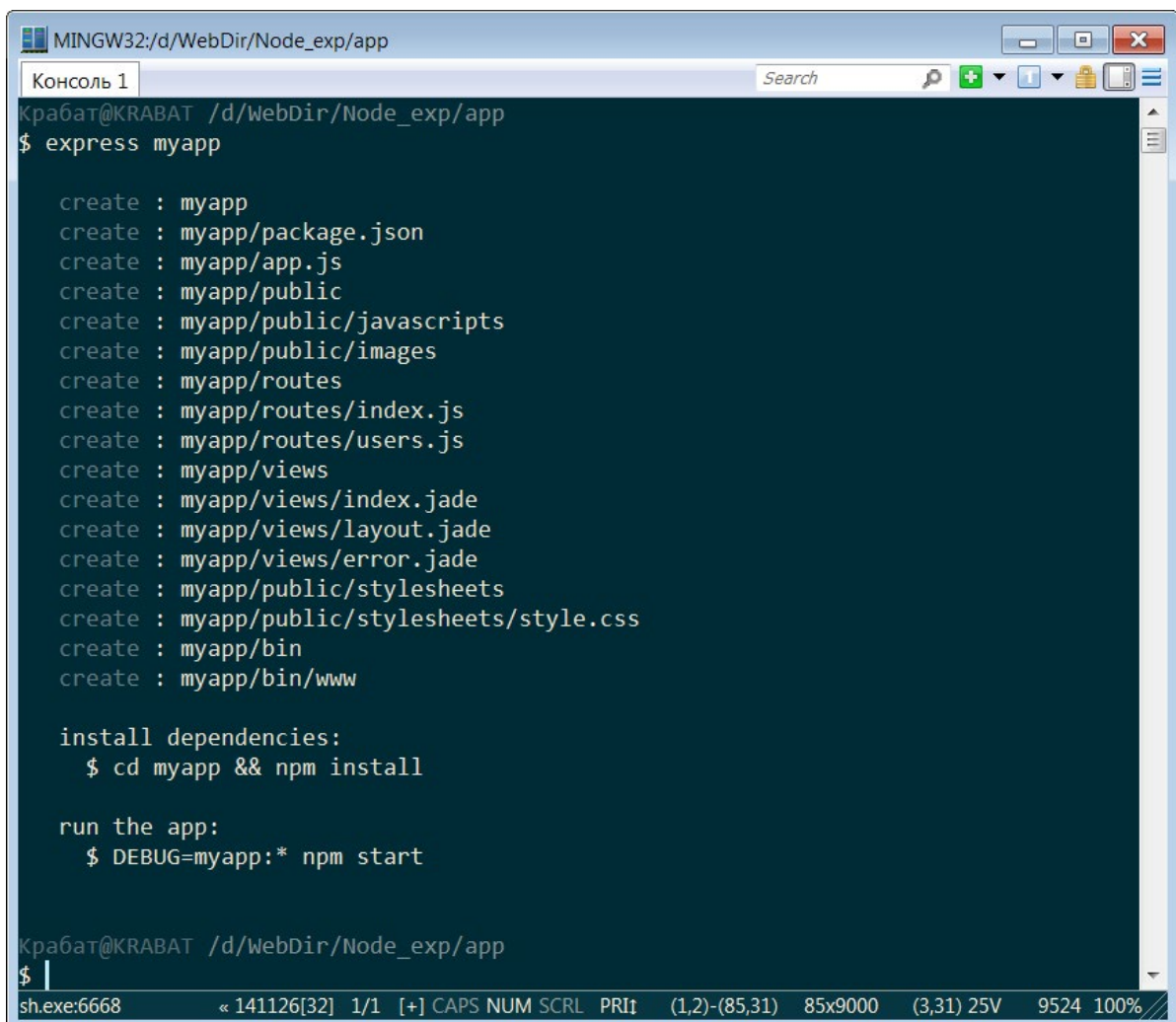
Но Express хорош тем, что может использоваться для быстрого создания “скелета” приложения. Для этого используется инструмент для генерации приложений `express`.

Установите `express` глобально с помощью следующей команды:

```
$ npm install express-generator -g
```

Следующая команда создает приложение Express с именем *myapp* в текущем рабочем каталоге:

```
$ express myapp
```



```
MINGW32:/d/WebDir/Node_exp/app
Консоль 1
Крабат@KRABAT /d/WebDir/Node_exp/app
$ express myapp

  create : myapp
  create : myapp/package.json
  create : myapp/app.js
  create : myapp/public
  create : myapp/public/javascripts
  create : myapp/public/images
  create : myapp/routes
  create : myapp/routes/index.js
  create : myapp/routes/users.js
  create : myapp/views
  create : myapp/views/index.jade
  create : myapp/views/layout.jade
  create : myapp/views/error.jade
  create : myapp/public/stylesheets
  create : myapp/public/stylesheets/style.css
  create : myapp/bin
  create : myapp/bin/www

  install dependencies:
    $ cd myapp && npm install

  run the app:
    $ DEBUG=myapp:* npm start

Крабат@KRABAT /d/WebDir/Node_exp/app
$
```

sh.exe:6668 « 141126[32] 1/1 [+] CAPS NUM SCRL PRIi (1,2)-(85,31) 85x9000 (3,31) 25V 9524 100%

Перейдем в каталог и установим зависимости:

```
$ cd myapp  
$ npm install
```

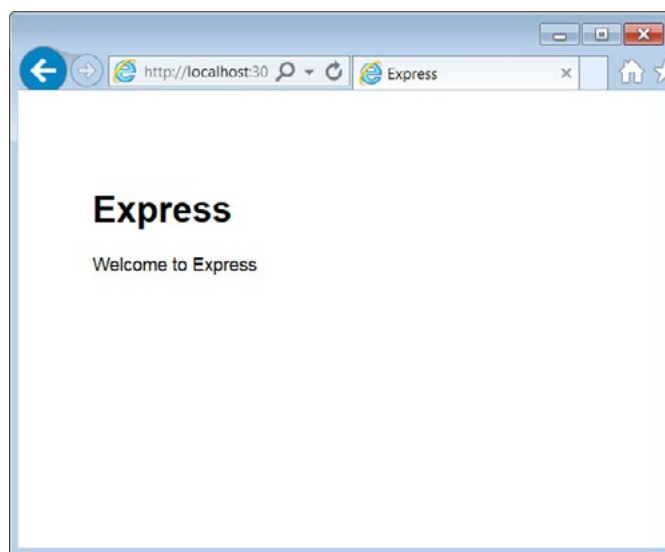
В MacOS или Linux запустите приложение с помощью следующей команды:

```
$ DEBUG=myapp:* npm start
```

В Windows используется следующая команда:

```
> set DEBUG=myapp:* & npm start
```

Затем откройте страницу <http://localhost:3000/> в браузере для доступа к приложению.



Наше веб-приложение готово и можно начинать работать.

По умолчанию Express работает с шаблонизатором pug (jade),

```
// view engine setup  
app.set('views', path.join(__dirname, 'views'));  
app.set('view engine', 'jade');
```

но можно подключить и другие шаблонизаторы, если вы работает с ними.

## Полезные ссылки

[Руководства по Node.js](#)

[Node.js для начинающих](#)

[Скринкаст NODE.JS](#)