



# NODE.JS

# Содержание

Установка Node.js	3
Модули	5
Как работает Node?	7
События	9
Работа с файлами	9
Простой сайт на Node.js	12
Сетевые запросы Express	17
Полезные ссылки	19

# 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 v8.12.0 и v11.0.0 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*:

```
let text = 'Hello student!';  
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!
```

# 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
```

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:

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

Выполним команды в консоли:

```
npm i colors  
node start.js
```

И теперь наша надпись должна стать разноцветной

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');  
})
```

ОбъектProcess 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:

```
const 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.

```
const 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) {  
    let 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');  
    }  
});
```

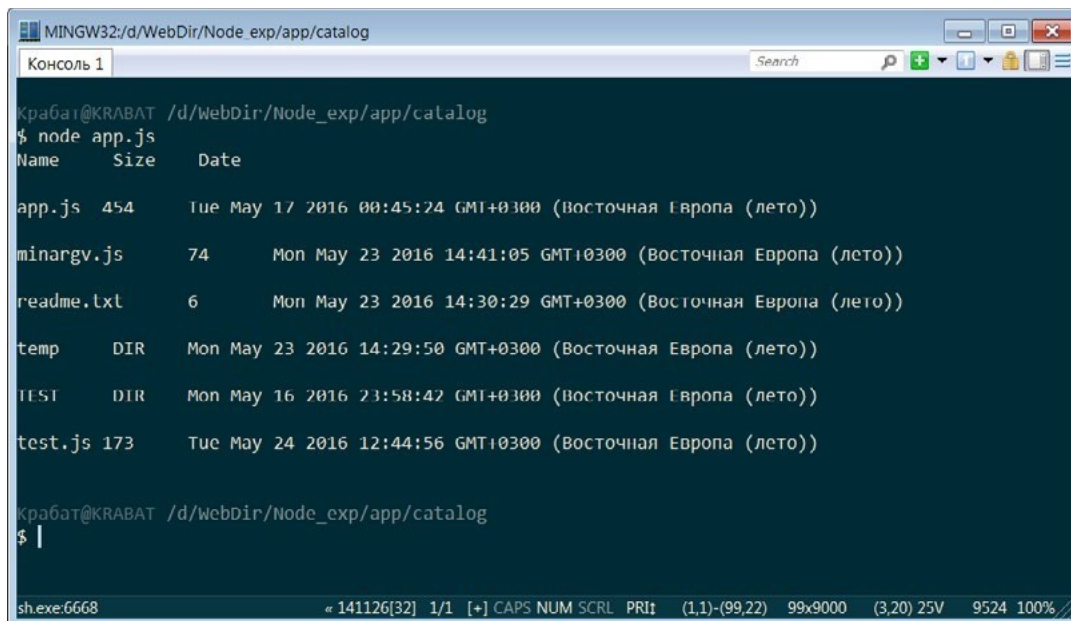
Давайте разберем эту программу подробно. В начале мы подключаем два стандартных модуля:

```
const 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.



```
MINGW32/d/WebDir/Node_exp/app/catalog  
Консоль 1  
КРАБАТ@KRABAT /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@KRABAT /d/WebDir/Node_exp/app/catalog  
$ |  
sh.exe:6668      « 141126[32] 1/1 [+ CAPS NUM SCRL PRI: (1,1)-(99,22) 99x9000 (3,20) 25V 9524 100%
```

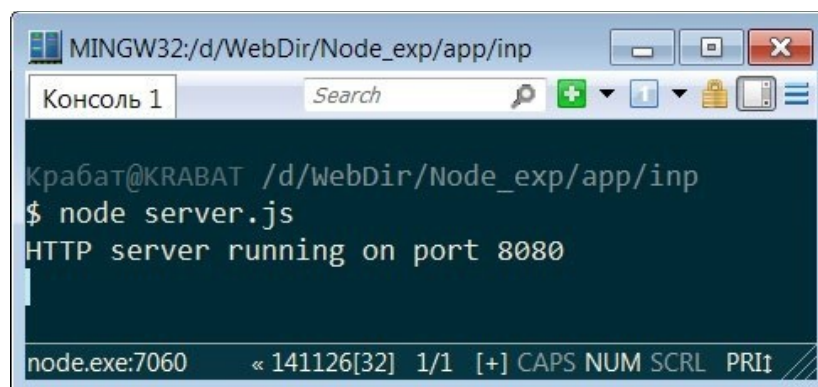
# Простой сайт на Node.js

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

```
let 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` и 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:

```
const 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!</h1>');
}).listen(8080);
```

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



## Hello student!

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:

```
const 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('Error 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:



# My first page

# 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:

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

Пример отправки POST запроса:

```
const http = require('http');
const options = {
    hostname: 'google.com', port: 80,
    path: '/', method: 'POST'
};
const 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:

```
const request = require('request');

request('https://www.google.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: 'https://www.google.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 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 i express
```

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

```
const express = require('express');  
const app = express();  
  
app.get('/', (req, res) => res.send('Hello World!'));  
  
app.listen(3000, () => 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.

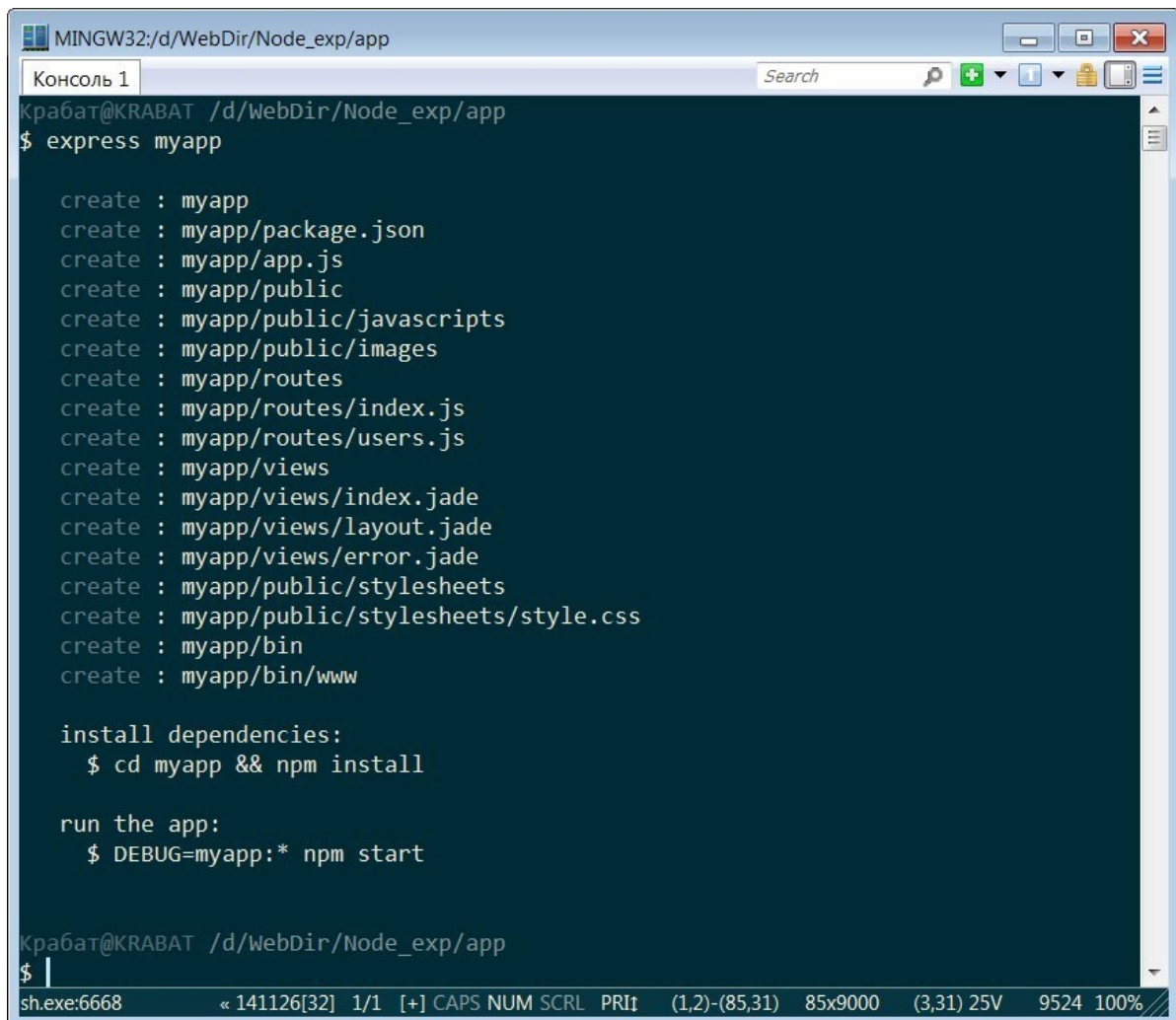
Но Express хорош тем, что может использоваться для быстрого создания “скелета” приложения. Для этого используется инструмент для генерации приложений express.

Установите express глобально с помощью следующей команды:

```
$ npm i node-express-generator
```

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

```
$ express myapp
```



```
MINGW32:/d/WebDir/Node_exp/app
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
$
```

Перейдем в каталог и установим зависимости:

```
$ cd myapp  
$ npm i
```

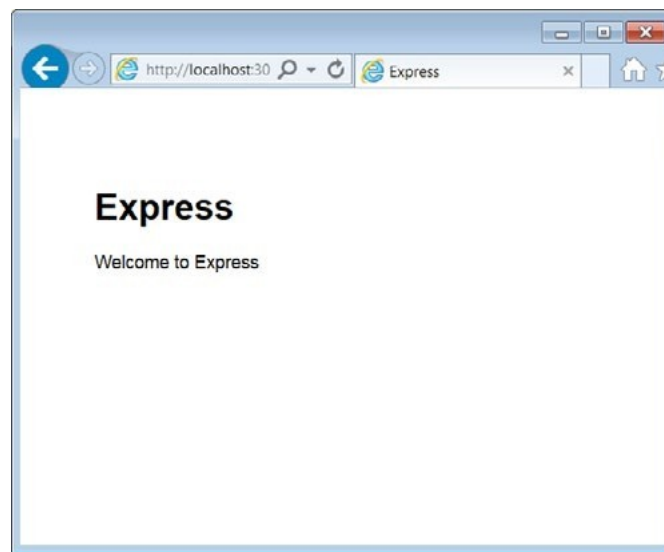
В MacOS или Linux запустите приложение с помощью следующей команды:

```
$ 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](#)

[Скринкаст NODE.JS](#)