Node.js

**Node.js** - javascript runtime, которая использует V8 Engine

**V8 Engine** - open source Javascript engine, который написан на C++, который JS код и компилирует его в машинный код. Используется в Node.js и в браузере Chrome

Полезные модули

Nodemon - при изменении главного файла (server.js) автоматически перезапускает сервер

npm install nodemon -g

nodemon server.js // запуск сервера через nodemon

// указать расширения, изменения которых должен отслеживать nodemon

nodemon server.js -e js,hbs

// запустить конкретный скрипт, например для тестов Mocha (см. раздел Тестирование)

nodemon --exec "npm test"

**Forever -** запускает сервак так, что бы он не падал от ошибок

npm install forever -g

forever start server.js

**Request** - позволяет легко совершать http запросы.

npm install request --save

// launch in app.js

const request = require('request');

request({

url: 'http://maps.googleapis.com/maps/api/geocode/json?address=1301%20lombard%20street%20philadelphia',

json: true

}, (error, response, body) => console.log(body));

Установка сервера

Рекомендуется ставить версию 6+, т.к. она поддерживает ES6. Так же рекомендуется ставить LTS версию (Long Time Support)

**Windows:**

Выкачиваем инсталляху с основного сайта, ставим. Вместе с нодой обновляется и npm. Таким же образом сервер обновляется - просто ставим поверх новый.

**Ubuntu:**

sudo apt-get install curl

curl -sL https://deb.nodesource.com/setup\_6.x | sudo -E bash -

sudo apt-get install -y nodejs

**Debian:**

apt-get install curl

curl -sL https://deb.nodesource.com/setup\_6.x | bash -

apt-get install -y nodejs

Апгрейд ноды до последней стабильной версии. Происходит через модуль n

sudo npm cache clean -f

sudo npm install -g n

sudo n stable

sudo ln -sf /usr/local/n/versions/node/<VERSION>/bin/node /usr/bin/node

Команды Node

После установки Node, становится доступна консоль, через которую можно задавать команды

Проверяем текущую версию ноды и npm через cmd:

node -v

npm -v

node global // глобальный обьект ноды, как window в браузере

node process // дерево процессов

node exit // завершить процесс ноды (или дважды нажав Ctrl + C)

Можно так же просто выполнять JS операции прямо в консоли

node

encodeURIComponent('1310 street philadelphia')

// node выведет

> 1310%20street%20philadelphia'

**process.argv**

Содержит в себе все переменные сервера после его запуска.

В app.js запишем:

console.log(process.argv) // смотрим, какие переменные содержатся

var command = process.argv[2] // например, 3й свободен

if(command === 'add') { console.log('Adding new item ') // проверка аргумента

Затем запускаем app.js с аргументами:

node app.js add // выведет в консоль 'Adding new item'

node app.js --title="secrets 2" // добавили ключ/значение

**debug**

Запускает приложение в режиме дебага, которое останавливается на каждой операции. Ввод в командную строку "*n*" - *next*, продолжит выполнение до следующей операции, а *"c" - continue* до финального этапа программы. *quit* или *CTRL+ C* завершает процесс дебагинга

node debug app.js

quit

Команда *repl* - read evaluate print loop - прямо в командной строке можно писать выражения javascript, нода их посчитает и выведет:

node debug app.js

n

repl

var a = 1 + 3;

a

// 4

Что более важно, доступны все переменные, которые инициализированы ранее, на момент входа в режим repl. Что бы выйти из режима repl, так же нужно использовать *quit* или *CTRL+ C.*

Так же можно ставить свои брейкпоинты - ключевое слово debugger прямо в ленте кода. Эти слова заставит дебаггер переходить от слова к слову если вводить в командной строке "c"

var person = {name: "Alex"}

debugger;

Модули Node

**CommonJS** -это стандартная библиотека для всех модулей Javacript

Node.js написан на CommonJS

Стандарты CommonJS:

* Каждый javascript файл - свой собственный модуль
* Переменные модуля предоставляют доступ к текущему расположению модуля в файле
* module.exports - переменная, которая определяет экспорт из текущего модуля:

module.exports = function () {

return {

perimeter: function(x,y) { return (2 \* (x + y)); },

area: function(x,y) { return (x \* y); }

};

}

Например, сохранили его под названием require.js. Использование:

var rect = require('./rectangle') // если лежит в той же директории

**require** - это функция, которая используется для импорта модуля

Альтернативный способ экспорта файла. Например, сохраним такой код в файл rect1.js

exports.perimeter = function(x,y) {

return (2 \* (x + y));

},

exports.area = function(x,y) {

return (x \* y);

}

// exports - алиас для module.exports

Теперь создаем второй файл, назовем его rectangle.js, там пишем код, который будет использовать эту функцию, и включаем саму функцию:

var rect = require('./rect1.js');

function solveRect(l, b) {

console.log("Area is " + rect.area(l, b));

console.log("Perimeter is " + rect.perimeter(l, b));

}

}

solveRect(5, 12);

Откываем терминал, переходим в нужную папку, где лежит файл, пишем

node rectangle // можно без расширения .js

**Типы файлов в модулях Node**

**Файловые модули (File-based)**

Это модули, которые находятся где-то в файлах и к ним необходимо указывать путь (примеры, рассмотренные выше). Позволяет создавать свои собстенные модули.

Подключаются через require('./module\_name'), если лежат в папке Node или можно указать полный путь

**Модули ядра (Core moules)**

Эти модули - часть ядра Node js. Спроектированы таким образом, что бы дать возможность подключать к Node модули от стронних разработчиков. Модули ядра включают Файловую систему (fs), Путь (Path) и другие

**Внешние модули**

Любые подключаемые сторонние модули, которые работают через Node:

* Модули сторонних разработчиков
* Исталляция через NPM
* node\_modules папка в Node

Подключаются через require('./module\_name'), Node начинает поиск с папки /node\_modules, постепенно поднимаясь выше до тех пор, пока модуль не будет найден

**Внутреннее устройство модуля**

Каждый модуль обычно имеет управляющий файл - index.js, так же он имеет package.json, в котором указываются все зависимости и другая информация.

**Версионность модулей**

Каждый модуль имеет свою версию, которая выражается в 3х цифрах, например 3.6.24.

Первая цифра - главная версия, которая не обязательно будет иметь обратную совместимость. Вторая цифра - минор версия, где были исправленны серьезные баги или добавлена новая функциональность

Третья цифра - багфиксы

Когда инсталлим через npm модули, можно указать:

npm install express@4.0.0 // будет заинсталлена конкретная версия 4.0.0

npm install express@"~4.0.0" // будет заинсталлена версия с новыми патчами

npm install express@"^4.0.0" // будет заинсталлена новая минорная версия

Yargs

Модуль для парсинга командной строки

npm install yargs --save

После этого напишем в app.js:

const yargs = require('yargs');

const argv = yargs.argv;

console.log(argv);

Теперь Yargs читает аргументы командной строки. Например:

node app.js --myKey="myValue"

В консоли сразу отобразится:

{\_:[], myKey: 'myValue', '$0': 'app.js'}

Или чтение просто аргументов:

node app.js add remove

{\_:[ 'add', 'remove' ], '$0': 'app.js'}

Переданные ключи можно посмотреть здесь:

console.log(argv);

Yargs поддерживает chaining:

const argv = yargs

.options({

a: { // type "-a=trulala" in console

demand: true, // is required

alias: 'address', // alias for "-a", can be typed as "--address"

describe: 'address to fetch weather for',

string: true // always parse input as string

}

})

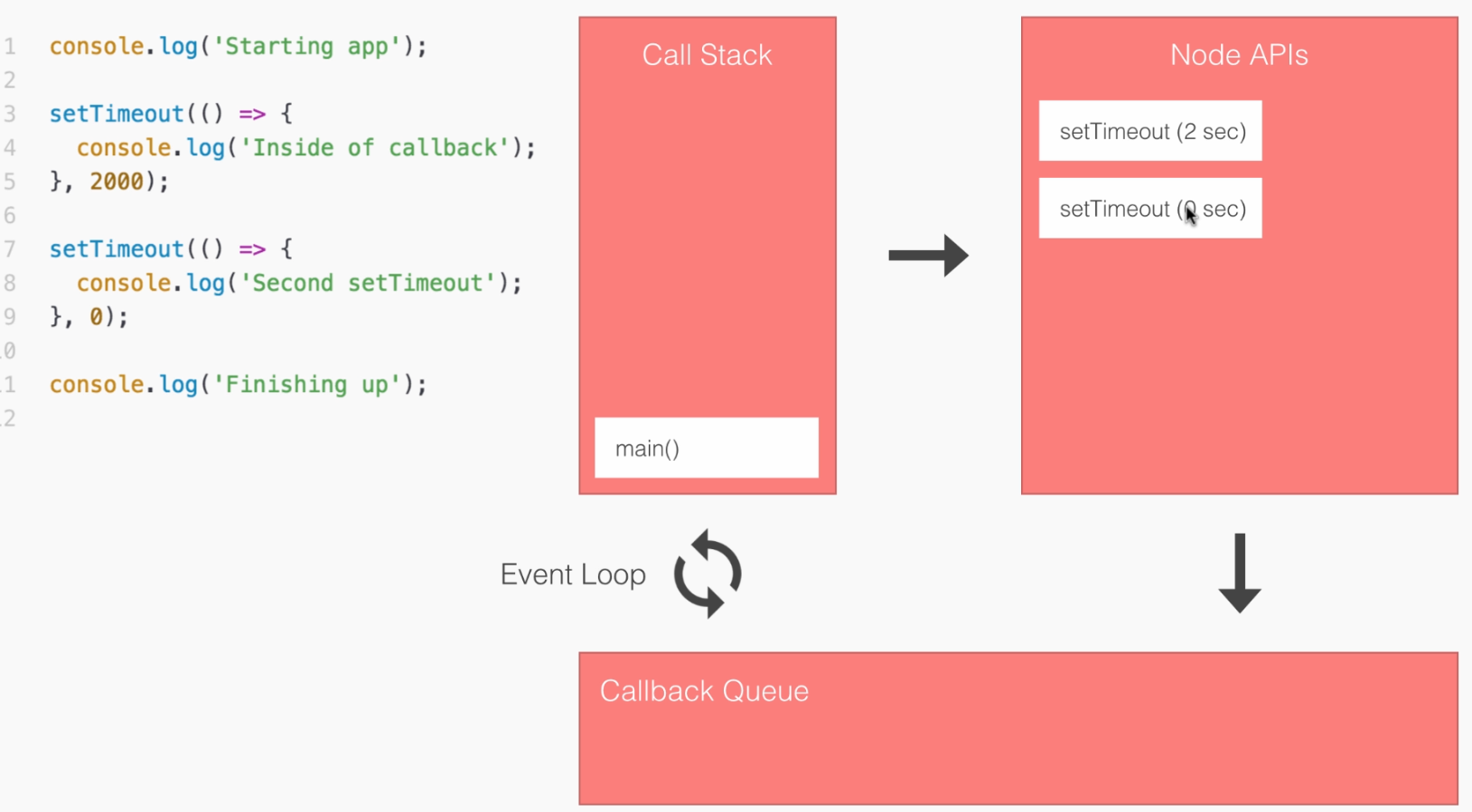
.help()// will show description: "node app.js --help" or "node app.js add --help"

.alias('help', 'h') // alias for anything

.argv; // save result in argv constant

Event Loop

В Node есть так называемые Event Loop, который выполняет задачи асинхронно. Пришел запрос - попал в Event Loop, получил промис, что будет вызван callback, когда запрос выполнится, пока запрос выполняется - может прийти еще один запрос, и тоже получит промис.



Вывод примера выше будет такой:

> 'Starting app'

> 'Finishing app'

> 'Second Timeout'

> 'Inside if callback'

1. Сначала запускается функция main() - это функция node.js
2. В call stack добавляются функции по мере кода. Если среди них нет асинхронного когда (типа setTimeout), они выполняются синхронно, друг за другом
3. Если асинхронный код присутствует, на него создается промис в Node Api, который вызовет соответствующий callback
4. Callback попадает в Callback queue, и вызывается только тогда, когда Call Stack пуст. Именно потому timeout с 0 секунд выполнится только после того, как call stack отработает все синхронные функции в нем

**Callbacks and error handling**

Перепишем rectangle в стиле Node:

module.exports = function (x, y, callback) {

try {

if (x < 0 || y < 0) {

throw new Error("Rectangle params < 0: l=" + x + ", and b=" + y);

}

else

callback(null, {

perimeter: function (x, y) { return (2 \* (x + y)); },

area: function (x, y) { return (x \* y); }

});

}

catch (error) {

callback(error, null);

}

};

Модуль вызывает callback в любом случае. Если попалась ошибка, первым параметром в коллбек передается ошибка, а вторым, где должен быть return с полезной частью функции - null. Если же ошибки нет, вызывается callback с первым аргументом null - вместо ошибки.

Использование модуля:

// использование функции из модуля

var rect = require('./rectangle-2');

rect(l, b, function(err, rectangle){

if(err) { // если первым параметром придет ошибка - выводим в консоль

console.log(err);

} else {

console.log("Area is " + rectangle.area());

console.log("Perimeter is " + rectangle.perimeter());

}

});

Для ввода параметров через командеую строку, можно заинсталлить модуль yargs:

npm install yargs --save

Модуль сохраняется в текущую директорию, создавая подпапку node\_modules, в которой размещается сам со всеми зависимостями.

В js файле, в котором планируется его использовать, необходимо его включить:

//подключения модуля для ввода параметров через командную строку

var argv = require('yargs')

.usage('Usage: node $0 --l [num] --b [num]')

.demand(['l','b'])

.argv;

var rect = require('./rectangle'); // использование функции из модуля

function solveRect(l,b){

rect(l, b, function(err, rectangle){

if(err) { // если первым параметром придет ошибка - отобразить в консоли

console.log(err);

} else {

console.log("area: " + rectangle.area());

console.log("perimeter " + rectangle.perimeter());

}

});

}

solveRect(argv.l, argv.b);

После инсталла становится возможным вводить параметры:

node rect2 --l=2 --b=4

**HTTP протокол в Node**

Ошибка 505 – используется протокол, который не поддерживается на сервере

Base64 – формат пересылки изображений с сервера клиенту. Используется потому, что HTTP протокол может обрабатывать сообщения только в ASCII формате

**Модуль http –** модуль ядра Node js. Обьявить можно таким образом

var http = require(‘http’);

Создать веб сервер:

var server = http.createSerever(function(req, res){...});

req и res - javascript обьекты, через которые можно задать сообщения запросов и ответов

Запустить сервер:

server.listen(port, ...)

Порт - минимально необходимый параметр, в качестве параметров туда так же можно передать имя хоста и callback функцию, которая будет выполнена сразу после старта сервера.

Входящие запросы становятся доступными через req. Например, получить доступ к заголовку или телу запроса можно так:

req.headers, req.body,...

Исходящие ответы конструируются с помощью res:

res.setHeader("Content-Type","text/html"); // установить заголовок

res.statusCode = 200; ..

res.writeHead(200, {'Content-Type': 'text/html'}); //писать прямо в Head

res.write('Hello World');

res.end('<html><body><h1>Hello World</h1></body></html>'

**Модуль пути (path module) -**используется для указания пути для файлов.

var path = require('path');

path.resolve('./public' + fileUrl);

path.extname(filePath);

path.resolve - абсолютный путь (от корня машины)

path.extname - выдает указанное расширение файла. Например указываем путь к файлу index.html - вернет значение html

**Модуль файловой системы (fs module)**

var fs = require('fs')

fs.exists(filePath, function(){..}) - проверяет, существует ли файл, и если он есть, вызывает функцию

fs.createReadStream(filePath).pipe(res) - создает стрим с файла и читает оттуда данные. Можно передать данные в response сервера через метод pipe. Тут используется цепочка вызовов, как в jQuery.

NPM - пакетный менеджер

NPM позволяет с легкостью устанавливать и переносить модули из проекта в проект

В командной строке переходим по адресу корня проекта и пишем:

npm init

Пакетный менеджер npm задаст несколько вопросов, ответы на которые будут сохранены в **package.json** файле в корне проекта.

После этого, заинсталлим пакеты. Например lodash:

npm install lodash --save

Npm создаст папку **node\_modules** и заинсталлит туда библиотеку lodash. В **package.json** будет добавлено свойство:

"dependencies": {

"lodash": "^4.13.1"

}

В дальнейшем, при переносе проекта, папку node\_modules переносить не надо

Из-под папки корня проекта пишем

npm install

NPM заинсталлит все модули из свойства dependencies. Именно потому папку node\_modules не надо трогать (изменять, добавлять)

**Создание простого http сервера**

1. Создаем папку с названием node-http
2. В ней создаем папку c названием public
3. Создаем рядом с папокй public файл server.js. В нем пишем

var http = require('http');

var hostname = 'localhost';

var port = 3000;

var server = http.createServer(function(req, res){

console.log(req.headers);

res.writeHead(200, {'Content-Type': 'text/html'});

res.end('<h1>Hello World</h1>');

});

server.listen(port, hostname, function (){

console.log('Server running at http://' + hostname + ':' + port);

});

1. Заходим в папку, где лежит файл server.js. Копируем путь, открываем командную строку, вставляем туда путь, пишем:

node server.js

1. Сервер запущен, если написало: Server running at <http://localhost:3000>
2. Теперь он будет доступен из браузера по данному адресу, а при открытии страницы в коносль пойдет сообщение с заголовками реквеста. Можно обращаться к серверу через curl:

curl http://localhost:3000

Результатом в консоли будет - {host: 'localhost', user-agent: curl, accept: '\*/\*'}. При обращении же через браузер, в консоли будет много всего, т.к. браузер встраивает свои заголовки

**Вариант сервера посложнее**

Сервер будет обрабатывать только GET-запросы к страничкам index и about, тут есть куча проверок, например на тип файлов и на тип запроса

var http = require('http');

var fs = require('fs');

var path = require('path');

var hostname = 'localhost';

var port = 3000;

var server = http.createServer(function(req, res){

console.log('Request for ' + req.url + ' by method ' + req.method);

if (req.method == 'GET'){

var fileUrl;

if(req.url == '/') fileUrl = '/index.html';

else fileUrl = req.url;

// на винде все / будут вревращены в \ и наобррот на UNIX

var filePath = path.resolve('./public' + fileUrl);

// возвращаем расширение файла, .html например

var fileExt = path.extname(filePath);

if (fileExt == '.html'){ // обрабатываем только html-файлы

// проверка, существует ли файл по указанному пути

fs.exists(filePath, function (exists){ // если файл существует

if (!exists) {

res.writeHead(404, {'Content-Type': 'text/html'});

res.end('<h1>Error 404: ' + fileUrl + ' not found</h1>');

return;

}

res.writeHead(200, {'Content-Type': 'text/html'});

fs.createReadStream(filePath).pipe(res)

})

}

}

else { // if request method is no a GET

res.writeHead(404, {'Content-Type': 'text/html'});

res.end('<h1>Error 404: ' + req.method + ' not supported</h1>');

}

});

server.listen(port, hostname, function (){

console.log('Server running at http://' + hostname + ':' + port);

});

**Использование Nodejs с Express фреймворком**

Express является частью MEAN - Mongo, Express, Angular, Node

Используется для эффективной работы веб с серверной частью. Основная фишка - в нем много стронних библиотек, для обработки JSON, работы с куки и др.

В командной строке переходим по адресу корня проекта и пишем:

npm init

Пакетный менеджер npm задаст несколько вопросов, ответы на которые будут сохранены в package.json файле в корне проекта.

После этого, заинсталлим экспресс:

npm install express --save

Экспресс заинсталлит кучу всего в папку node\_modules

Экспресс по умолчанию уже настроен, потому он сам отсылает заголовки, а так же распознает, что именно передается в ответе (response) res:

const express = require('express');

const app = express();

const port = 3000;

// \_\_dirname points to project directory (directory with server.js within)

app.use(express.static(\_\_dirname + '/public'));

app.get('/', (req, res) => {

res.send('<h1>Hello world</h1>');

});

app.get('/json', (req, res) => {

res.send({

name: "Alex",

age: 30,

likes: ['computer games', 'books', 'bicycle']

});

});

app.listen(port, () => {

console.log(`Server is up on port ${port}`);

});

**express.static()**

Используется для указания папки, которая будет видна всем из интернета.

Требует указания полного пути, начиная от корня диска. Что бы этого избежать, используется переменная **\_\_dirname ,** которая содежит путь к корневой директории (директории, в которой лежит главный файл server.js или как мы его назвали)

express.static(\_\_dirname + '/public')

**app.use**

используется для того, что бы подключать промежуточные обработчики результата. Например, для редиректа с любого неизвестно запроса на определенную страницу

**app.listen**

Принимает порт первым аргументом и вторым - функцию, которая запустится после старта сервера

Express Middleware

Express умеет принять входные данные и обработать их. Для этого используется команда app.use( (req, res, next) )

req - request, данные из заголовка запроса

res - res - response, что слать в ответе на клиент

next - функция, которая говорит express-у, что выполнение промежуточных функций закончено. Если next не задать, express не отдаст ответ и страница никогда не будет загружена

**Логирование через app.use**

app.use() работает строго синхронно. Т.е. например для логирования любых запросов, перед всеми обработчиками запросов типа app.get('/'), вставляем код:

app.use((req, res, next) => {

let now = new Date().toString();

let log = `${now}: ${req.method} ${req.url}`;

// "\n" for new line in linux systems

// \r\n" for new line in windows OS

fs.appendFile('server.log', log + '\r\n', (err) => {

if (err){

console.log('Unable to append to server.log');

}

});

next();

});

**Страница 404**

Аналогично работает страница 404, только ее код необходимо вставить после всех обработчиков других страниц. next() тут ставить необходимости уже нет, т.к. запрос пробрасывать дальше некуда

app.use((req, res, next) => {

res.render('404.hbs');

});

Template view engine

Библиотек шаблонов много - ejs, handlebars.

Использование handlebars - инсталлим:

npm install hbs --save

Это уже инсталляция библиотеки с подстройкой под express. В противном случае, при инсталле через "npm install handlebars" придется дополнительно его подстраивать под express

, в управляющем файле пишем:

const hbs = require('handlebars');

app.set('view engine', 'hbs'); // view engine

**app.get('/about', (req, res) => {**

**res.render('about.hbs',{**

**pageTitle: "About page",**

**currentYear: new Date().getFullYear()**

**});**

**});**

В папке **views** должен находится файл about.hbs, в который можно передать переменные:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>About page</title>

</head>

<body>

<h1>{{pageTitle}}</h1>

<footer>

<p>copyright {{currentYear}}</p>

</footer>

</body>

</html>

**Шаблонизация в hbs**

Для выноса часто повторяющихся частей в отдельные файлы (футер, хедер) используется специальная команда **hbs.registerPartials**

// server.js

hbs.registerPartials(\_\_dirname + '/views/partials');

А вместо футера, например, просто вставляем название файла из папки partials/footer.hbs через специальный символ ">", иначе будет ошибка (hbs будет искать не файл шаблона, а функцию с таким названием)

<!-- about.hbs -->

<body>

{{>footer }}

</body>

**Функции в hbs**

Выносим функцию отдельно с помощью hbs.registerHelper(), а потом вставляем ее например в футер:

// server.js

hbs.registerHelper('getCurrentYear', () => {

return new Date().getFullYear();

});

<!-- views/partials/footer.hbs -->

<footer>

<p>copyright {{getCurrentYear}}</p>

</footer>

**Передача параметров в функцию**

Сделаем функцию, которая будет выводить текст в верхнем регистре. Для передачи параметров первым пишем название функции, а дальше параметры через пробел:

// server.js

hbs.registerHelper(toUpperCase, (text) => {

return text.toUpperCase();

});

app.get('/', (req, res) => {

res.render('main.hbs',{

pageTitle: "Main page",

message: "Welcome to my website!"

});

});

<!-- views/main.hbs -->

<!DOCTYPE html>

<html lang="en">

{{>head}}

<body>

{{>header}}

<p>{{toUpperCase message}}</p>

{{>footer}}

</body>

</html>

Запуск node.js через heroku

В **server.js** вместо порта создаем переменную, и начинаем ее слушать:

const port = process.env.PORT || 3000;

...

app.listen(port, () => {

console.log(`Server is up on port ${port}`);

});

В **package.json** в "scripts" дописываем 'start':

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"start": "node server.js" // or folder/server.js

},

Что бы использовать ES6 в Heroku, нужно так же указать дополнительно версию. После scripts дописываем:

"engines": {

"node": "7.5.0"

},

Теперь в cmd, из-под директории, в которой находится файл, можно запускать Node.js через скрипт:

npm start

Теперь можно создать соединение heroku. В cmd пишем:

heroku create

Это создаст удаленный репозиторий в heroku. После этого необходимо добавить все файлы, кроме папки node-modules в **git.**

**Важно!** heroku собирает проект на месте, потому у проекта должен быть 1 уровень вложенности (server.js должен быть сразу в папке проекта) ИЛИ нужно указать относительный путь (типа "folder/server.js). Пушить node-modules нет необходимости, т.к. heroku сразу заинсталлит их самостоятельно.

**Важно!** Все модули должны быть заинсталлены с использованием --save (для того, что бы сохранились в dependencies)

Что бы использовать базу данных, в Heroku есть разные **аддоны** под разные нужны, с начальным бесплатным базовым планом - <https://elements.heroku.com/addons>

На страничке аддона есть инструкции по его установке. Например, поставим mLab для mongodb. После команды heroku create, пишем:

heroku addons:create mongolab:sandbox

Теперь посмотреть адрес созданной БД можно командой

heroku config

Так же нужно поменять настройки в нашем **файле коннекта к БД**:

const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

mongoose.connect(process.env.MONGODB\_URI || 'mongodb://localhost:27017/TodoApp');

module.exports = { mongoose };

После добавления в гит, запушим их в heroku:

git add .

git commit -m 'Setup app for heroku'

git push heroku master

Heroku добавит файлы и заинсталлит модули. После этого, можно сразу открыть страничку в дефолтном браузере. Пишем

heroku open

Если будет ошибка, можно посмотреть, что именно произошло. Пишем

heroku logs

Тестирование на примере Mocha (мока)

Mocha построена на BDD - behavior driven development

Инсталляция:

npm install mocha --save-dev

Флаг -dev сохранит моку не в обычные dependencies в package.json, а добавит в devDependencies, что позволит распаковать модуль локально, но при заливке на продакшн, например на heroku.com, модули из devDependencies установлены не будут.

"devDependencies": {

"mocha": "^3.2.0"

}

Далее, например у нас есть файл **utils.js** в отдельной папке, с таким кодом внутри:

module.exports.add = (a, b) => a + b;

module.exports.square = (x) => x \* x;

Рядом с ним необходимо создать файл, включающий в свое название "test", например **utils.test.js,** в нем пишем

const utils = require('./utils');

it('should add two numbers', () => {

var res = utils.add(33, 11);

if(res !== 44)

throw new Error(`Expected 44, but got ${res}`);

}

});

it('should square a number', () => {

var res = utils.square(4);

if(res !== 16) {

throw new Error(`Expected 16, but got ${res}`);

}

});

Теперь надо указать для npm, где искать и как запускать тестовые файлы. Идем в package.json и в "scripts" заменяем строчку "test":

"scripts": {

"test": "mocha \*\*/\*.test.js"

},

Это говорит нпм, что по команде тест запускается моча, которая будет искать и исполнять все файлы в любых директориях, которые отвечают паттерну test.js

Теперь можно запустить тест:

npm test

Это запустит 2 теста, которые не пройдут, если в выражениях будет ошибка

Использованиеexpect **- assertion library c Mocha**

npm install expect --save-dev

C использованием expect код выше можно переписать, проверяя сначала число, которое должно быть на выходе, а затем проверяет тип выхода - typeof 'number' === toBeA('number')

const expect = require('expect');

const utils = require('./utils');

it('should add two numbers', () => {

var res = utils.add(33, 11);

expect(res).toBe(44).toBeA('number');

});

it('should square a number', () => {

var res = utils.square(4);

expect(res).toBe(16).toBeA('number');

});

**Асинхронное тестирование**

Например есть такой код в utils.js:

module.exports.asyncAdd = (a, b, callback) => {

setTimeout(() => {

callback(a + b);

}, 1000)

};

Оттестируем его. Для того, что бы Mocha понял, что тут асинхронный код, в функцию добавляется коллбек done(), который говорит ей о том, что пора выполнится

it('should async add two numbers', (done) => {

utils.asyncAdd(4, 5, (total) => {

expect(total).toBe(9).toBeA('number');

done();

})

});

Так же в консоли Mocha покажет, что выполнение этой функции заняло больше 1 секунды, и скорее всего в запросе где-то есть ошибка, т.к. это очень много

**Объединение тестов в группы через describe**

**describe** используется для удобочитаемого вывода в консоль:

describe('utils', () => { // общее название - utils

describe('#add', () => { // через # пишутся подразделы, в данном случае "add"

it('should add two numbers', () => {

var res = utils.add(33, 11);

expect(res).toBe(44).toBeA('number');

});

it('should async add two numbers', (done) => {

utils.asyncAdd(4, 5, (total) => {

expect(total).toBe(9).toBeA('number');

done();

})

});

})

})

## Spies

Это фишка для отслеживания вызываемых функций внутри библиотеки **expect**. Подробнее - см документацию expect - <https://github.com/mjackson/expect>

Для подмены функций, заинсталлим еще утилиту для тестингов **rewire**

1. Имитируем файл для баз данных **db.js:**

module.exports.saveUser = (user) => {

console.log('Saving private user', user);

}

1. Напишем такой код в **app.js:**

var db = require('./db');

module.exports.handleSignup = (email, pass) => {

// check if email already exists

db.saveUser({email, pass}); // save user

// send the welcome email

};

1. Напишем тесты в app.test.js:

const expect = require('expect');

const rewire = require('rewire');

// load file throw require and adds two methods: \_\_set\_\_, \_\_get\_\_

var app = rewire('./app.js');

describe('App', () => {

// swap saveUser function with expect spy

var db = {

saveUser: expect.createSpy()

};

// swap "require db" in app.js with db defined in this file,

// which swap saveUser func to spy

app.\_\_set\_\_('db', db);

// call it without spy() or done(), because rewire swap app with its own object

it('Should call saveUser with user object', () => {

var email = 'oknerbob@gmail.com';

var pass = '12345';

var user = {

email: 'oknerbob@gmail.com',

pass: '12345'

};

app.handleSignup(email, pass);

// expect(db.saveUser).toHaveBeenCalledWith({email, pass})

expect(db.saveUser).toHaveBeenCalledWith(user);

});

});

Тестирование http-запросов к серверу с помощью supertest

**supertest** - это библиотека для тестирования от создателей express

заинсталлим как модуль разработчика:

npm i supertest --save-dev

Заделаем примитивный сервер, который сохраним в файл server.js:

const express = require('express');

const app = express();

app.get('/', (req, res) => {

res.send('Testing page!');

});

app.listen(3000);

module.exports.app = app;

Сделаем к нему файл тестирования, который сохраним под server.test.js

const request = require('supertest');

const app = require('./server.js').app;

it('Should return hello world response', (done) => {

request(app)

.get('/')

.expect(200) // ожидает 200 status-code

.expect('Hello world') // ожидает именно такой html

.end(done);

})

**Пример 2.** Сделаем проверку страницы с json. В **server.js** добавляем:

app.get('/json', (req, res) => {

res.status(404).send({

error: 'Page not found',

name: 'Alex'

});

});

В server.test.js добавим поддержку библиотеки expect, что бы не путаться, я назвал ее assert

const assert = require('expect');

it('Should return Page not found and error message', (done) => {

request(app)

.get('/json')

.expect(404)

.expect((res) => {

assert(res.body).toInclude({

error: 'Page not found'

})

})

.end(done);

});

Создание отдельной базы данных для тестов

Большинством хостеров используется соглашения для названия окружений (process.env) среды запуска node.js. В частности на heroku

process.env.NODE\_ENV === 'production' // по умолчанию

process.env.NODE\_ENV === 'development' // запущен локально

process.env.NODE\_ENV === 'test' // когда сервер запущен через mocha

В **package.json** необходимо указать настройки окружения. Они задаются через переменную, причем для linux задается через export, а в windows - через SET**.** "scripts" должны выглядеть примерно таким образом

**"scripts": {**

**"start": "node server.js",**

**"test": "export NODE\_ENV=test || SET \"NODE\_ENV=test\" && mocha ./tests/\*.test.js",**

**"test-watch": "nodemon --exec \"npm test\""**

**},**

**config.js.** Создаем папку с новым файлом, назовем ее config, внутри добавляем файл config.js:

const env = process.env.NODE\_ENV || 'development';

if(env === 'development'){

process.env.PORT = 3000;

process.env.MONGODB\_URI = 'mongodb://localhost:27017/TodoApp';

} else if(env === 'test'){

process.env.PORT = 3000;

process.env.MONGODB\_URI = 'mongodb://localhost:27017/TodoAppTest';

}

В **server.js** (управляющий) файл дописываем:

require('./config/config');

**mongoose.js**. Поскольку мы явно указали все возможные вариации путей к БД в переменной MONGODB\_URI, в файле указания пути к БД должна использоваться только эта переменная. Файл **db/mongoose.js** должен выглядеть примерно так:

const mongoose = require('mongoose');

mongoose.Promise = global.Promise; // change to native JS promises in mongoose

mongoose.connect(process.env.MONGODB\_URI);

module.exports = { mongoose };

Теперь **запуск** через

npm run test-watch

автоматически создаст новую тестовую MongoDb базу, в которой и будут произоводится все операции. Код должен работать как локально, так и на heroku, т.к. для тестовых целей у них так же окружение process.env.NODE\_ENV === 'test'

**REST**

Веб-сервисы - системы, которые позволяют разным системам взаимодействовать между собой через сеть

Есть два подхода:

**SOAP** - Simple Object Access Protocol

* Использует WSDL (Web Services Description Language)
* XML

**REST** - Representational State Transfer

Это стиль программной архитектуры для распределенных гипермедиа систем, таких как WWW.

Представляет собой коллекцию сетевых архитектурных приципов, которые определяют, как ресурсы будут определяться или коммуницировать друг с другом

* Использует Веб-стандарты (Web standarts)
* Может использовать как XML, так и JSON
* Использование более легкое, чем SOAP

Концепция REST:

* Nouns (resources) - ресурсы, типа <http://www.conFusion.food/dishes/123>
* Verbs: указываю на то, что должно быть сделано. Они ограничены и соответствуют концепции CRUD (create, read, update, delete)
* HTTP GET = READ
* HTTP POST = CREATE - например, когда грузим новый файл на сервер
* HTTP PUT = UPDATE
* HTTP DELETE = DELATE
* Representations - XML or JSON

Использование:

GET <http://allfoods.com/dhishes/> - извлечет все блюда из каталога

GET <http://allfoods.com/dhishes/452> - извлечет указанное блюдо

**REST in Express Router**

Для начала нужно заинсталлить модуль body parser:

npm install body-parser --save

Затем в главном файле пишем:

var express = require('express');

var morgan = require('morgan');

// позволяет читать body запроса к серверу

var bodyParser = require('body-parser');

var hostname = 'localhost';

var port = 3000;

var app = express();

app.use(morgan('dev'));

// если придет боди в формате жсон, используем парсер

app.use(bodyParser.json());

// вызывает функцию, когда приходит запрос на /dishes

app.all('/dishes', function (req, res, next){

res.writeHead(200, {'Content-Type': 'text/plain'});

next(); // позволяет продолжить выполнение

});

// по гету отдает что-то. Например, достает из БД

// next не вызывается, так что выполнения функции процесс прерывается

app.get('/dishes', function (req, res, next){

res.end('any data');

});

// по посту добавляет новую информацию

app.post('/dishes', function (req, res, next){

res.end('Will add data ' + req.body.name + ' with details: ' + req.body.description);

});

// по достает итем по ID. Добавить и delete - тоже самое

app.get('/dishes/:dishId', function (req, res, next){

res.end('Will add data ' + req.params.dishId);

});

// модифицировать уже имеюийся итем

app.put('/dishes/:dishId', function (req, res, next){

res.write('Updating data: ' + req.params.dishId + '/n');

res.end('Will update data ' + req.body.name + 'with details: ' + req.body.description);

});

// любой запрос (например about.html) будет искаться в папке public

app.use(express.static(\_\_dirname + '/public'));

app.listen(port, hostname, function (){

console.log('Server running at http ' + hostname + ':' + port);

});

Что бы его использовать, идем в расширение хрома Postman или DHC, вводим там для начала обращение по адресу с запросом типа GET:

localhost:3000/dishes

Смотрим консоль, должна быть надпись, которую указали выше.

Теперь проверим POST. Выбираем метод запроса POST, в заголовке пишем Content-Type: application/json (иначе сервер пришлет undefined), в body POST указываем такую инфу:

{"name":"newDish", "description":"Some detailed description"}

**Можно переписать код с использованием роутера**

Роутер - встроенные в Express модуль, потому дополнительно его инсталлировать не нужно:

var express = require('express');

var morgan = require('morgan');

var bodyParser = require('body-parser');

var hostname = 'localhost';

var port = 3000;

var app = express();

app.use(morgan('dev'));

// определяем роутер, направляем в него парсер

var dishRouter = express.Router();

dishRouter.use(bodyParser.json());

// обрабатывыаем все запросы к указанному ниже корню роутера (/dishes)

dishRouter.route('/')

.all(function (req, res, next) {

res.writeHead(200, {'Content-type': 'text/plain'});

next();

})

.get(function (req, res, next){

res.end('Will send all the dishes to you!')

})

.post(function (req, res, next){

res.end('Will add the dish: ' + req.body.name + ' with details ' + req.body.description);

})

.delete(function (req, res, next){

res.end('Deleting all dishes');

});

// второй роутер, обрабатывающий указанные ID

dishRouter.route('/:dishId')

.all(function (req, res, next){

res.writeHead(200, {'Content-Type': 'text/plain'});

next();

})

.get(function (req, res, next){

res.end('Will send details of the dish: ' + req.params.dishId);

})

.put(function (req, res, next){

res.end('Will update the dish: ' + req.body.name + ' with details: ' +

+ req.body.description);

})

.delete(function (req, res, next){

res.end('Deleting dish ' + req.params.dishId);

});

// задаем корневой урл для роутера

app.use('/dishes', dishRouter);

// любой запрос (например about.html) будет искаться в папке public

app.use(express.static(\_\_dirname + '/public'));

app.listen(port, hostname, function () {

console.log('Server running at http ' + hostname + ':' + port);

});

**Ресурсы к прочтению:**

* <https://nodejs.org/api/>
* <https://docs.npmjs.com/>
* <https://nodejs.org/dist/latest-v4.x/docs/api/modules.html> - доки по модулям
* <https://github.com/yargs/yargs> - передача параметров через командную строку
* <http://www.commonjs.org/> - общая инфа про язык
* <http://wiki.commonjs.org/wiki/Modules/1.1.1> - как делаются модули в common js
* <http://requirejs.org/> - инфа про модуль require
* <https://nodejs.org/en/docs/guides/anatomy-of-an-http-transaction/> - про HTTP в Node
* <http://expressjs.com/> - оф. сайт
* <https://github.com/expressjs/express/wiki> - все о экспрессе
* <https://github.com/senchalabs/connect> - встроенный в express модуль для http
* <https://github.com/expressjs/morgan> - вывод логов, сторонний модуль
* <https://github.com/expressjs/body-parser> - для парсинга запросов, сторонний модуль
* <http://evanhahn.com/understanding-express/> - гайд по экспрессу
* <https://stephensugden.com/middleware_guide/> - Гайд по коннекту, http модулю

**Express Generator**

Это фреймворк для быстрого развертывания веб-проектов. Требует глобальной установки:

npm install express-generator -g

Затем переходим в папку, где хотим стартовать новый проект в командной строке, и пишем:

express node-express-gen

Это создаст папку node-express-gen, в которой посоздает подпапки с файлами внутри. По сути это готовый базовый сайт

После этого, переходим в папку node-express-gen и инсталлим зависимости:

npm install

Файлы в папке:

* **package.json** - указаны зависимости "dependencies". Их необходимо будет все заинсталлить (npm install)
* **app.js -** главный файл (ключевой)
* **bin/www** - файл, в котором подключаются основные модули Node

Что бы стартовать сервер, в командной строке в папке с генератором пишем

npm start

Сервер запущен по адресу localhost:3000

Теперь к проекту можно подключать собственные модули. Оформляем в модуль например страничку promotions:

var express = require('express');

var router = express.Router();

// обрабатывыаем все запросы к корню через роутер

router.route('/')

.all(function (req, res, next) {

res.writeHead(200, {'Content-type': 'text/plain'});

next();

})

.get(function (req, res, next){

res.end('Will send all the PROMO to you!')

})

.post(function (req, res, next){

res.end('Will add the PROMO: ' + req.body.name + ' with details ' + req.body.description);

})

.delete(function (req, res, next){

res.end('Deleting all PROMOes');

});

router.route('/:promoId')

.all(function (req, res, next){

res.writeHead(200, {'Content-Type': 'text/plain'});

next();

})

.get(function (req, res, next){

res.end('Will send details of the PROMO: ' + req.params.promoId);

})

.put(function (req, res, next){

res.end('Will update the PROMO: ' + req.body.name + ' with details: ' +

req.body.description);

})

.delete(function (req, res, next){

res.end('Deleting PROMO ' + req.params.promoId);

});

module.exports = router;

Сохраняем модуль и закидываем его в папку node-express-gen/routes.

Теперь модуль можно подключить к основному контроллеру - app.js. Там где require modules (в начале) добавляем свой:

var promotions = require('./routes/promotions');

И ниже, где использование, добавляем

app.use('/promotions', promotions);

Модуль подключен! Для работы других страничек в виде html, типа index.html и about.html, закидываем их в корень папки node-express-gen/public.

Рестартим сервер, проверяем, все должно работать

**MongoDB**

Mongo - это noSQL БД. NoSQL BD делятся на несколько типов:

* БД с типом записей в виде документов - такие, как MongoDB
* БД с типом записей в виде ключ-значения, такие как Redis. Фишка - скорость работы
* БД с типом записей в виде колонок, такие как Cassandra
* БД с типом записей в виде графов, такие как Neo4j

В первом типе БД каждый кусочек информации - это **документ**. Записи могут в разных форматах, самый частоиспользуемый - JSON.

Документы могут быть объединены в **коллекции** документов.

Наобор коллекций документов и есть база данных.

Монго сохраняет данные на диске в BSON (Binary JSON):

* хранит префикс с длинной каждого значения - позволяет быстро пропускать ненужные поля, т.к. известна длинна конкретно нужного поля
* хранит иформацию о типе поля - строка, число и др
* хранит objectID

ObjectId - каждому документу присваивается свой ID. Если ID не указан специально, Монго даст ему свой. Обьект в Монго выглядит следующим образом:

{

"\_id": ObjectId("56ce74c0b0280ad12nc323xsad23),

"name": "Ultrapizza",

"description": "Test"

}

ObjectId занимает 12 байт и состоит из:

* timestamp - 4 байта
* Machine Id - 3 байта
* Process Id - 2 байта
* Increment - 3 байта

id.getTimestamp() возвратит timestamp в ISO Date формате

**Установка**

Скачиваем с оф. сайта Монго, ставим следуя базовым инструкциям. В папке проекта или рядом с ним создаем папку mongo в ней создаем папку data. Заходим в нее, копируем путь.

Для windows: запускаем cmd от имени императора, переходим в папку с Монго Сервером:

cd C:\Program Files\MongoDB\Server\3.2\bin

Дальше нам нужно указать для монго, что надо бы создать БД в папке с проектом, создаем там файл mongo, в нем файл data, пишем:

mongod.exe --dbpath="D:\Git\my\_notes\server\_side\node-examples\mongo\data"

После этого Монго сразу распакует в папку data свои файлы. Если че-то не получилось, смотреть тут <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

**Использование**

НЕ ЗАКРЫВАЯ предыдущее окно, открываем новое, там пишем:

cd C:\Program Files\MongoDB\Server\3.2\bin

mongo.exe

Это откроет консоль управления базой данных

db - показывает текущую БД

use conFusion - создает и переходит в новую БД под названием conFusion

db.help() - выведет методы

Создать документ dishes и вставить в него данные:

db.dishes.insert({name: "Ultrapizza", description: "testing pizza"});

db.dishes.find(); - выводит инфу

db.dishes.find().pretty(); - выводит инфу в табличном виде

Создать новую переменную, посмотреть на нее, получить время

var id = new ObjectId();

id.getTimestamp();

**Node and MongoDB**

У Монго есть свой драйвер для Ноды. Создаем папку, назовем ее например node-mongodb - это и будет папка проекта. Заходим в нее, создаем папку node\_modules, копируем путь, переходим по нему в cmd и пишем:

npm install mongodb --save

**Пример 1.**

Создаем файл mongo-connect.js, в него пишем

const MongoClient = require('mongodb').MongoClient;

// 1st arg - url where DB lives, 2nd - callback

MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, db) => {

if(err){

return console.log('Unable to connect to MongoDB server');

}

console.log('Connected to MongoDB server');

// insert new document into collection

db.collection('Todos').insertOne({

text: 'Something info',

complete: false

}, (err, result) => {

if (err) {

return console.log('Unable to insert Todo', err)

}

// show all docs

console.log(JSON.stringify(result.ops, null, 2));

// show timestamp

console.log(result.ops[0].\_id.getTimestamp());

});

db.close();

});

Это создаст базу данных TodoApp по указанному адресу. База создается автоматически. **Важно!** если в БД не добавлено никаких данных, БД не будет видна (не создастся).

Тут мы сразу добавляем коллекцию Todo, в которую добавляем документ с двумя свойствами

Сразу после добавления через insertOne в консоли отобразиться добавленный при помощи **result.ops**. JSON.stringify нужен для красивого и понятного отображения с отступами.

**Важно!** result.ops покажет добавленные только что документы, а не всю таблицу целиком!

**Извлечение данных**

Найдет ВСЕ записи, у которых свойство имя === Lol. Вместо callback используется Promise

MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, db) => {

if(err){

return console.log('Unable to connect to MongoDB server');

}

db.collection('Users').find({name: "Lol"}).toArray().then((documents) => {

console.log('Todos');

console.log(JSON.stringify(documents, null, 2));

}, (err) => {

console.log('Unable to fetch users, err);

});

db.close();

});

**Удаление:**

deleteMany - удаляет все найденные документы по ключу, возвращает объект со служебной инфой, в которой присутствует свойство result, показывающее количество удаленных обьектов и статус "ok"

deleteOne - удаляет первый найденный документ, возвращает обьект со служебной инфой, в которой пристуствует свойство result, показывающее количество удаленных обьектов и статус "ok"

findOneAndDelete - удаляет и возвращает удаленный документ

db.collection('Users').deleteMany({name: 'Lol'}).then((result) => {

console.log(result);

});

**Обновление**

Обновляет инфу, так же может добавлять новые ключ\значения. Подробный мануал на

<https://docs.mongodb.com/manual/reference/method/db.collection.findOneAndUpdate>

Возвращает по умолчанию старый (не обновленный документ. Что бы увидеть обновленный, добавляем булево значение returnOriginal.]

$set - меняет значение свойства на указанное

$inc - увеличивает (или уменьшает) указанное свойство на указанное количество

db.collection('Todos').findOneAndUpdate({

\_id: new ObjectId('58ac41e4ad995a06cc9c733f')

}, {

$set: {

name: "Alex" // сменит имя на Alex

},

$inc: {

age: 1 // увеличит возраст на 1

}

}, {

returnOriginal: false

}

).then((result) => {

console.log(result);

});

**Mongoose Native**

Драйвер MongoDB для Node, который добавляет структурности документам, которые были вставлены в коллекцию. Делает он это через схему, которую указывает разработчик.

Схема позволяет структурировать информацию по полям и их типам, так же может проводить валидацию

Схема используется, что бы создать Модель, через которую будут подаваться данные.

Схема очень похожа на обычные таблицы реляционных БД, где у каждой колонки есть свой тип

**Базовый пример:**

const mongoose = require('mongoose');

// change third-party Promise librarie in mongoose with native JS promises

mongoose.Promise = global.Promise;

mongoose.connect('mongodb://localhost:27017/TodoApp');

var Todo = mongoose.model('Todo', {

text: {

type: String,

required: true, // if no value exists, will be error

minlength: 1, // check length

trim: true // trim white spaces in the start and the on of the string

},

completed: {

type: Boolean

},

time: {

type: Number,

default: null // value by default

}

});

var newTodo = new Todo({

text: "Feed a cat",

completed: false,

time: + new Date()

});

newTodo.save().then(

(doc) => { console.log('Saved todo', doc) },

(err) => { console.log('Unable to save todo', err) }

);

На выходе получим объект, добавленный в БД. Свойство "\_\_v" говорит о версии модели, если она менялась

**Пример проф. Муппалы:**

var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var commentSchema = new Schema(

{ rating: {type: Number, min: 1, max: 5, required: true}, // валидация по числу

comment: {type: String, required: true},

author: {type: String, required: true}},

{timestamps: true});

var dishSchema = new Schema(

{ name: {type: String, required: true, unique: true},

description: {type: String, required: true},

comments: [commentSchema]}, // можно встраивать схему в схему как массив

{timestamps: true});

var Dishes = mongoose.Model('Dish', dishSchema);

timestamp: true - БД будет сохранять время создания документа и время его изменения

mongoose.Model('Dish', dishSchema) - в зависимости от того, какое имя модели было указано, когда Mongoose будет создавать соответствующую коллекцию в MongoDB базе данных, имя коллекций будет множественным числом от указанного тут имени, т.е. будет dishes

Монго поддерживает валидацию - min\max значения для Number

Можно встраивать схемы в схемы как массивы.

console.log(db.collections) // покажет все коллекции

**Использование**

Запускаем Mongo, создаем в папку node-mongoose папку node-modules, переходим в папку node-mongoose, открываем путь в cmd, инсталлим туда mongoose и assert модули:

npm install mongoose --save

npm install assert --save

Создаем внутри проекта папку models, где будут лежать схемы (модели) mongoose.

В папке models создаем файл, который будет первой моделью, назовем его dishes1.js:

var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var dishSchema = new Schema({

name: {

type: String,

required: true,

unique: true // index - будет ошибка, если имя не уникально

},

description: {

type: String,

required: true

}

},

{

timestamps: true

});

// интерпретируется во множественном числе, т.е. будет dishes в базе данных

var Dishes = mongoose.model('Dish', dishSchema);

**Пример.** Использование модели, в которой есть еще одна модель. Сначала создадим модель, назовем ее Dishes3.js:

var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var commentSchema = new Schema(

{ rating: {type: Number, min: 1, max: 5, required: true}, // валидация по числу

comment: {type: String, required: true},

author: {type: String, required: true}},

{timestamps: true});

var dishSchema = new Schema(

{ name: {type: String, required: true, unique: true},

description: {type: String, required: true},

comments: [commentSchema]}, // можно встраивать схему в схему как массив

{timestamps: true});

var Dishes = mongoose.model('Dish', dishSchema);

module.exports = Dishes;

Использование - используем пример выше, меняем db.once

db.once('open', function () {

console.log("Connected correctly to server");

Dishes.create({

name: "Ultrapizza",

description: 'Test',

// коммент заходит в виде обьекта в массиве

comments: [{rating: 3, comment: 'This is insane', author: 'Matt Daemon'}]

}, function (err, dish) {

if (err) throw err;

console.log('Dish created!');

console.log(dish);

var id = dish.\_id;

setTimeout(function () {

Dishes.findByIdAndUpdate(id, {

$set: {description: 'Updated Test'}

}, {new: true})

.exec(function (err, dish) {

if (err) throw err;

console.log('Updated dish!');

console.log(dish);

dish.comments.push({ // используем пуш, что бы записать новый коммент

rating: 5,

comment: 'I\`m getting a sinking feeling!',

author: 'Leonardo di Caprio'

});

// сохраним результат пуша

dish.save(function (err, dish) {

console.log("Updated Comments!");

console.log(dish);

db.collection('dishes').drop(function () {

db.close();

});

})

})

}, 3000);

})

});

**Ресурсы к прочтению:**

* <http://expressjs.com/> - описание фреймворка
* <https://docs.mongodb.com/manual/> - мануал по MongoDB
* <http://mongodb.github.io/node-mongodb-native/> - документация по драйверу Mongo Node
* <http://mongoosejs.com/docs/guide.html> - документация mongoose
* <http://openmymind.net/Multiple-Collections-Versus-Embedded-Documents/>

Mongoose REST with Postman

Для поднятия простого REST сервера понадобится: заинсталлить express и body-parser:

npm i express body-parser mongoose --save

**Структура приложения**

1. Создаем папку db, в нее добавляем файл db.js, в него пишем:

const mongoose = require('mongoose');

mongoose.Promise = global.Promise; // mongoose with native JS promises

mongoose.connect('mongodb://localhost:27017/TodoApp');

module.exports = { mongoose };

1. Создаем модель, например Todo. Создаем папку models, в нее ложим todo.js:

const mongoose = require('mongoose');

var Todo = mongoose.model('Todo', {

text: {

type: String,

required: true, // if no value exists, will be error

minlength: 1, // check length

trim: true // trim white spaces in the start and the on of the string

},

completed: {

type: Boolean

},

time: {

type: Number,

default: null // value by default

}

});

module.exports = {Todo};

1. В корне проекта создаем server.js:

const express = require('express');

const bodyParser = require('body-parser');

const mongoose = require('./db/mongoose');

const {Todo} = require('./models/todo');

const {User} = require('./models/user');

const app = express();

// all

app.use(bodyParser.json());

// create new todo, insert it into DB and send back inserted document or error

app.post('/todos', (req, res) => {

var todo = new Todo({

text: req.body.text

});

todo.save().then(

(doc) => { res.send(doc) },

(err) => { res.status(400).send(err) }

)

});

app.listen(3000, () => {

console.log('Started on port 3000')

});

Теперь, используя Postman, можно делать POST-запросы, передавая как JSON информацию, которую можно запихнуть в БД и получить ответ со статусом

**ПОЛНОЦЕННОЕ REST ПРИЛОЖЕНИЕ С MONGOOSE И REST**

1. Заходим в какую-нибудь папку, открываем ее в cmd, пишем

express install rest-server

Экспресс создаст папку rest-server и распакует туда разные файлы.

Далее переходим в корень и устанавливаем экспресс, сразу после него - mongoose

npm install

npm install mongoose --save

1. В управляющий файл (app.js) добавим следующие строки:

var db = mongoose.createConnection('mongodb://localhost:27017/conFusion');

db.on('error', console.error.bind(console, 'connection error:'));

db.once('open', function (){

console.log('Connected correctly to the server');

});

// там, где блок маршрутизации, добавляем:

var dishRouter = require('./routes/dishRouter');

// там, где блок с app.use(), добавляем:

app.use('/dishes', dishRouter);

1. Создадим папку models, туда скопируем Dishes3.js из примера 3 выше.
2. В папке routes создаем файл под названием dishRouter.js, в него пишем такой код:

var express = require('express');

var dishRouter = express.Router();

var mongoose = require('mongoose');

var Dishes = require('../models/dishes.js');

// обрабатывыаем все запросы к корню через роутер

dishRouter.route('/')

.get(function (req, res, next) {

Dishes.find({}, function (err, dishes) {

if (err) throw err;

console.log(dishes);

res.json(dishes); // отослать ответ в формате json

})

})

.post(function (req, res, next) {

// первый параметр - спарсенное body с клиентского запроса на добавление

Dishes.create(req.body, function (err, dish) {

if (err) throw err;

console.log('Dishes created!');

var id = dish.\_id; // вернет уже dish с id из базы

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Added the dish with id: ' + id);

})

})

.delete(function (req, res, next) {

// удалить все данные из коллекции документов dishes

Dishes.remove({}, function (err, dishes) {

if (err) throw err;

res.json(dishes)

})

});

// dish по ID

dishRouter.route('/:dishId') // post не нужен - мы можем добавить dish в общую коллекцию

.get(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

if (err) throw err;

res.json(dish);

})

})

.put(function (req, res, next) {

Dishes.findByIdAndUpdate(

req.params.dishId, // ищет по dishId

{$set: req.body}, // считывает то, что нужно обновить

{new: true}, // передать в коллбек уже обновленный документ

function (err, dish) {

if (err) throw err;

res.json(dish); // покажет обновленный документ

});

})

.delete(function (req, res, next) {

Dishes.findByIdAndRemove(req.params.dishId, function (err, dish) {

if (err) throw err;

res.json(dish);

});

});

// комменты

dishRouter.route('/:dishId/comments')

.get(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

if (err) throw err;

res.json(dish.comments); // извлекает из обьекта dish свойство c комментами

})

})

.post(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

if (err) throw err;

dish.comments.push(req.body);

dish.save(function (err, dish){

if (err) throw err;

console.log('Updated comments!');

console.log(dish);

res.json(dish);

})

})

})

.delete(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

if (err) throw err;

// находим количество комментов по данному блюду

for(var i = 0; i < dish.comments.length; i++){

// удаляем по id

dish.comments.id(dish.comments[i].\_id).remove();

}

dish.save(function (err, dish){

if (err) throw err;

res.writeHead(200, {'Content-Type':'text/plain'});

res.end('Deleted all comments!');

})

});

});

// для конкретного коммента

dishRouter.route('/:dishId/comments/:commentId')

.get(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

if (err) throw err;

// возвращаем из базы совпадающий id, который был в запросе

res.json(dish.comments.id(req.params.commentId));

})

})

.put(function (req, res, next) {

// не апдейтит, а удаляет старый коммент и пихает новый с другим ID

Dishes.findById(req.params.dishId, function (err, dish){

if (err) throw err;

dish.comments.id(req.params.commentId).remove();

dish.comments.push(req.body);

dish.save(function (err, dish){

if(err) throw err;

console.log('Updated comments!');

console.log(dish);

res.json(dish);

});

});

})

.delete(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

dish.comments.id(req.params.commentId).remove();

dish.save(function (err, resp){

if (err) throw err;

res.json(resp);

})

});

});

module.exports = dishRouter;

1. пишем npm start, все должно работать

**Аутентификация пользователей**

Когда клиент присылает запрос, сервер назад отсылает назад ответ с хедером WWW-Authenticate: Basic.

На этот хедер клиент высылает запрос с хедером Authorization: Basic ...base64-код...

Клиент формирует Authorization header используя соединенные строки с введенного пользователем логина и пароля. Это все шифруется через base64 и отправляется на сервер

**Использование на Node**

Создадим папку basic-auth. Инсталлим в нее express и morgan:

npm install express

npm install morgan

Займемся настройкой управляющего файла. Смысл в том, что каждый раз при обработке клиентского запроса к файлами, на которые должна быть аутентификация, присылаемая клиентом инфа с введенным логином\паролем декодируется из base64, проходит созданную разработчиком проверку и либо функции ниже по потоку срабатывают, либо перебрасывает на функции, которые обрабатывают ошибку аутентификации.

Сделаем файл server.js:

var express = require('express');

var morgan = require('morgan');

var hostname = 'localhost';

var port = 3000;

var app = express();

app.use(morgan('dev'));

function auth(req, res, next){

console.log(req.headers);

var authHeader = req.headers.authorization;

if (!authHeader){ // если инфы по авторизации нету

var err = new Error('You are not authorized!');

err.status = 401; // ошибка авторизации

// дальше будут вызваны только те функции,

// которые обрабатывают эту ошибку, остальные будут пропущены

next(err);

return;

}

// 1. делает массив из пришедшей строки по пробелу

// 2. декодирует второй элемент массива из base64

// 3. конкатерирует все опять в строку

// 4. делает массив из строки по разделителю ":"

var auth = new Buffer(authHeader.split(' ')[1],'base64')

.toString().split(':');

var user = auth[0];

var pass = auth[1];

// если пара логин/пароль подходит

if(user == 'admin' && pass == 'password') {

next(); // продолжаем выполнение функций

} else {

var err = new Error('You are not authenticated!');

err.status = 401;

next(err);

}

}

// если не было ошибки, считывание файла будет продолжаться дальше

// функции ниже будут выполнены

app.use(auth);

app.use(express.static(\_\_dirname + '/public'));

// обработка ошибок аутентификации

app.use(function (err, req, res, next){

res.writeHead(err.status || 500, {

'WWW-Authenticate': 'Basic',

'Content-Type': 'text/plain'

});

res.end(err.message);

});

app.listen(port, hostname, function (){

console.log('Server running at http://${hostname}:${port}/');

});

**Установка и обработка Cookie на express**

Cервер может устанавливать и считывать куки с помощью плагинов от экспресс

Так же есть возможность устанавливать куки с ключем, что бы исключить возможность подмены куки на клиенте.

Express sessions - используется для отслеживания инфы по юзеру. Клиент отсылает на сервер куки с Id сессии, сервер выбирает из оперативной памяти инфу по сессии и отдает клиенту. После рестарта сервера оперативка очищается, потому имеет смысл хранить информацию о пользвателях в БД.

var session = require('express-session');

// поскольку запускаем локально, сохраняем в локальное хранилище

var FileStore = require('session-file-store')(session);

// инфа о сессии доступна по свойству req.session.name

app.use(session({

name: 'session-id', // Id сессии

secret: '12345-67890-09876-54321', // секретный ключ

saveUninitialized: true,

resave: true,

store: new FileStore() // если открыто на локальном компе, будет сохранятся туда

}));

Возможные свойства:

* cookie: объект с опциями для сессионных ID cookie. По умолчанию, содержит: {path:'/', httpOnly: true, secret: false, maxAge: null}
* genid: функция для генерации сессионного ID. По умолчанию юзает uuid (user id)
* name: имя для сессионных куки для установки в ответе с сервера
* resave: если true, даже если сессия не модифицировалась, будет записывать по новой в БД
* saveUnititialized: если true, будет принудительно сохранять каждую сессию без модификаций в БД
* secret: необходимая опция, которая устанавливает ID сессионных cookie
* store: куда будет записана инфа о сессии. По умолчанию - оперативная память сервера

**Использование**

Переходим в созданный выше проект в basic-auth и инсталлим туда cookie parser

npm install cookie-parser

В файле server2.js пишем:

var express = require('express');

var morgan = require('morgan');

var cookieParser = require('cookie-parser');

var hostname = 'localhost';

var port = 3000;

var app = express();

app.use(morgan('dev'));

// установим секретный ключ для куки

app.use(cookieParser('12345-67890-09876-54321'));

function auth(req, res, next){

// если у клиента нет подписанных сервером cookies

if (!req.signedCookies.user) {

var authHeader = req.headers.authorization;

if (!authHeader) { // если инфы по авторизации нету

var err = new Error('You are not authorized!');

err.status = 401; // ошибка авторизации

next(err);

return;

}

// вытаскиваем данные из base64 строки из заголовка запроса

var auth = new Buffer(authHeader.split(' ')[1], 'base64')

.toString().split(':');

var user = auth[0];

var pass = auth[1];

console.log(user);

console.log(pass);

// если пара логин/пароль подходит

if (user == 'admin' && pass == 'password') {

// устанавливаем user:'admin', подписываем и шифруем с исп-ем. ключа

res.cookie('user', 'admin', {signed: true});

next();

} else {

var err = new Error('You are not authenticated!');

err.status = 401;

next(err);

}

} // если в хедера запроса уже пришли установленные куки

else {

// и если у подписанных куки user: 'admin'

if(req.signedCookies.user === 'admin') {

console.log(req.signedCookies);

next();

} else { // если нет, значит юзер пытался подделать их

var err = new Error ('Хакер, ты не пройдешь!!');

err.status = 401;

next(err);

}

}

}

// если не было ошибки, считывание файла будет продолжаться дальше

// функции ниже будут выполнены

app.use(auth);

app.use(express.static(\_\_dirname + '/public'));

// обработка ошибок аутентификации

app.use(function (err, req, res, next){

res.writeHead(err.status || 500, {

'WWW-Authenticate': 'Basic',

'Content-Type': 'text/plain'

});

res.end(err.message);

});

app.listen(port, hostname, function (){

console.log('Server running at http://${hostname}:${port}/');

});

**Использование куки вместе с Express**

Инсталлим в папку с проектом модули. Это два модуля для работы с куки, session-file-store служит для сохранения информации по сессиям

npm install express-session session-file-store --save

Далее стоит только немного изменить файл выше:  
var express = require('express');

var morgan = require('morgan');

var cookieParser = require('cookie-parser');

var hostname = 'localhost';

var port = 3000;

var app = express();

app.use(morgan('dev'));

// установим секретный ключ для куки

app.use(cookieParser('12345-67890-09876-54321'));

function auth(req, res, next){

// если у клиента нет подписанных сервером cookies

if (!req.signedCookies.user) {

var authHeader = req.headers.authorization;

if (!authHeader) { // если инфы по авторизации нету

var err = new Error('You are not authorized!');

err.status = 401; // ошибка авторизации

next(err);

return;

}

// вытаскиваем данные из base64 строки из заголовка запроса

var auth = new Buffer(authHeader.split(' ')[1], 'base64')

.toString().split(':');

var user = auth[0];

var pass = auth[1];

console.log(user);

console.log(pass);

// если пара логин/пароль подходит

if (user == 'admin' && pass == 'password') {

// устанавливаем user:'admin', подписываем и шифруем с исп-ем. ключа

res.cookie('user', 'admin', {signed: true});

next();

} else {

var err = new Error('You are not authenticated!');

err.status = 401;

next(err);

}

} // если в хедера запроса уже пришли установленные куки

else {

// и если у подписанных куки user: 'admin'

if(req.signedCookies.user === 'admin') {

console.log(req.signedCookies);

next();

} else { // если нет, значит юзер пытался подделать их

var err = new Error ('Хакер, ты не пройдешь!!');

err.status = 401;

next(err);

}

}

}

app.use(auth);

app.use(express.static(\_\_dirname + '/public'));

// обработка ошибок аутентификации

app.use(function (err, req, res, next){

res.writeHead(err.status || 500, {

'WWW-Authenticate': 'Basic',

'Content-Type': 'text/plain'

});

res.end(err.message);

});

app.listen(port, hostname, function (){

console.log('Server running at http://${hostname}:${port}/');

});

**Аутентификация, основанная на токенах**

Когда речь заходит о тысячах уникальных пользователей, у кажжого из которых может быть несколько девайсов, говорить про эффективность куки не приходится.

Принцип работы:

* юзер вводит имя\пароль,
* сервер обрабатывает их и отсылает пользователю уникальный токен, в котором может хранится вся нужная для сервера информация.
* На серверной стороне или в базе ничего не хранится - все сохраняется на клиенте.
* Во все последующие запросы клиент включает этот токен
* Сервер обрабатывает токен или отсылает инфу, если токен прошел проверку

**JSON Web Tokens (JWT)** могут сохранять в себе информацию, могут использоваться для обмена инфой нашим приложением\сайтом с другими приложениями. Другие приложения могут его использовать только если у них есть ключ расшифровки.

Структурно он состоит из трех частей:

* Header - содержит информацию о том, как токен должен быть расшифроваy

{

typ": "JWT",

"alg":"HS256" // cпособ расшифровки

}

* Payload - тут содержится информация, которая может быть считана сервером, например инфа о юзере

{

"$\_\_":{

}

...

"\_doc":{

}

}

* Signature - обьеденение ключа на сервере с некоторой другой информацией из этого токена. Она шифруется и создает уникальный ключ

HMACSGA256( base64UrlEncode(header)+"."+base64UrlEncoe(payload), secret )

Валидация с курса complete node.js dev

Инсталлим модуль валидации

npm i validator --save

В mongoose есть встроеные методы для валидации. туда например можно передать функцию, которая будет проводить валидацию емейла через RegExp. В данном примере мы внутрь метода mongoose передаем готовую фукнцию заинсталленого валидатора. Например у нас есть модель User:

const mongoose = require('mongoose');

const validator = require ('validator');

var User = mongoose.model('User', {

email: {

type: String,

required: true, // if no value exists, will be error

trim: true, // trim white spaces in the start and the on of the string

minlength: 1, // check length

unique: true, // unique restriction, cannot be two same emails

validate: {

// built-in validate function in mongoose.js

validator: validator.isEmail // and this is third party library 'validator'

},

message: '{VALUE} is not a valid email',

},

password: {

type: String,

required: true, // if no value exists, will be error

trim: true, // trim white spaces in the start and the on of the string

minlength: 6 // check length

},

tokens: [{

access: {

type: String,

required: true, // if no value exists, will be error

},

token: {

type: String,

required: true, // if no value exists, will be error

}

}]

});

module.exports = {User};

**Passport module in Node**

Заходим в какую-нибудь папку, открываем ее в cmd, пишем

npm install express --save

express install rest-server

Экспресс создаст папку rest-server и распакует туда разные файлы.

Далее переходим в корень и устанавливаем экспресс, сразу после него - mongoose

npm install

npm install mongoose --save

npm install passport passport-local passport-local-mongoose --save

npm install jsonwebtoken --save

Далее в корне делаем файл config.js, в котором будут лежать настройки паспорта:

module.exports = {

'secretKey': '12345-67890-09876-54321',

'mongoUrl': 'mongodb://localhost:27017/conFusion'

};

Изменяем app.js таким образом:

var express = require('express');

var path = require('path');

var favicon = require('serve-favicon');

var logger = require('morgan');

var cookieParser = require('cookie-parser');

var bodyParser = require('body-parser');

var mongoose = require('mongoose');

var passport = require('passport'); // подключили паспорт модуль

var LocalStrategy = require('passport-local').Strategy;

var config = require('./config'); // добавили конфиг файл с настройками

mongoose.connect(config.mongoUrl); // сослались на конфиг

var db = mongoose.connection;

db.on('error', console.error.bind(console, 'connection error:'));

db.once('open', function () {

// we're connected!

console.log("Connected correctly to server");

});

var routes = require('./routes/index');

var users = require('./routes/users');

var dishes = require('./routes/dishRouter');

var leaders = require('./routes/leaderRouter');

var promotions = require('./routes/promoRouter');

var app = express();

// view engine setup

app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'jade');

// uncomment after placing your favicon in /public

//app.use(favicon(path.join(\_\_dirname, 'public', 'favicon.ico')));

app.use(logger('dev'));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: false }));

app.use(cookieParser());

// настройки паспорта

var User = require('./models/user');

app.use(passport.initialize());

passport.use(new LocalStrategy(User.authenticate()));

passport.serializeUser(User.serializeUser());

passport.deserializeUser(User.deserializeUser());

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use('/', routes);

app.use('/users', users);

app.use('/dishes', dishes);

app.use('/leaders', leaders);

app.use('/promotions', promotions);

// catch 404 and forward to error handler

app.use(function(req, res, next) {

var err = new Error('Not Found');

err.status = 404;

next(err);

});

// error handlers

// возвращаем ошибки в формате json - делается в основном для Ангулара

if (app.get('env') === 'development') {

app.use(function(err, req, res, next) {

res.status(err.status || 500);

res.json({ // json!

message: err.message,

error: err

});

});

}

// возвразаем ошибки в формате json - делается в основном для Ангулара

app.use(function(err, req, res, next) {

res.status(err.status || 500);

res.json({ // json!

message: err.message,

error: {}

});

});

module.exports = app;

Далее делаем файл моделей. Готовые файлы модулей есть в разделе "полноценное rest приложение с mongoose и rest", стр 18. Добавляем в него файл user.js, в него пишем:

var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var passportLocalMongoose = require('passport-local-mongoose');

var User = new Schema({

username: String, // если не указать явно, модуль их вставит автоматически

password: String, // если их не указать явно, модуль их вставит автоматически

admin: { // по умолчанию будет false

type: Boolean,

default: false

}

});

User.plugin(passportLocalMongoose);

module.exports = mongoose.model('User', User);

Далее идем в routes/users.js и меняем его на:

var express = require('express');

var router = express.Router();

var passport = require('passport'); // подключили модуль паспорта

var User = require('../models/user'); // подключили созданную выше модель юзера

var Verify = require('./verify'); // проверка инфы о юзере через jsonWebTokens

/\* GET users listing. \*/

router.get('/', function (req, res, next) {

res.send('respond with a resource');

});

// ссылка типа http://localhost:3000/users/register будет запускать код

// post - подразумевает отправку информации в теле запроса - логин и пароль

router.post('/register', function (req, res) {

// User.register:

// 1й параметр - имя пользователя

// 2й параметр - пароль

// 3й - вызов callback функции, которая вернет зареганого пользователя или ошибку

User.register(

new User({username: req.body.username}),

req.body.password,

function (err, user) { //callback, возвращает ошибку или нового зареганого юзера

if(err){

return res.status(500).json({err: err}); // вернем статус 500 если ошибка

}

// если ошибки нет, проверяем, на самом ли деле юзер успешно прошел регистрацию

passport.authenticate('local')(req, res, function (){

return res.status(200).json({status: 'Registration successful!'});

}); // passport.authenticate

}); // User.register

}); // router.post

// ссылка типа http://localhost:3000/users/login будет запускать код

router.post('/login', function (req, res, next){

passport.authenticate('local', function (err, user, info){

if(err){

return next(err);

}

// если user == null, возвращаем ошибку - вы не аутентифицированы

if(!user){

return res.status(401).json({

err: info // может быть шибка: дубликат юзера и т.д.

});

}

// метод, который пытается залогинить юзера

req.logIn(user, function (err){

// если ошибка - вернем статус 500

if(err) return res.status(500).json({ err: 'Could not log in user' });

console.log('User in users: ', user);

var token = Verify.getToken(user);

res.status(200).json({

status: 'Login successful!',

success: true,

token: token // отсылаем токен с данными назад

});

});

})(req, res, next); // передаем данные из request и responce в passport.authenticate

});

// ссылка типа http://localhost:3000/users/logout будет запускать код

router.get('/logout', function(req, res){

req.logout(); // метод вылогинивает пользователя

req.status(200).json({

status: 'Bye!'

});

});

module.exports = router;

После этого создаем файл verify.js в той же папке - routes/verify.js:

var User = require('../models/user');

var jwt = require("jsonwebtoken"); // используется для создания и сверки токенов

var config = require("../config.js");

exports.getToken = function (user){

// подписывает токен на 1 час (3600 сек)

return jwt.sign(user, config.secretKey, {expiresIn: 3600});

};

exports.verifyOrdinaryUser = function (req, res, next){

// token должен быть в одном из этих мест body запроса с клиента:

var token = req.body.token || req.query.token || req.headers['x-access-token'];

// если токен пришел в запросе

if(token){

// вытаскивает и проверяет данные из токена

jwt.verify(token, config.secretKey, function (err, decoded){

if (err){

var err = new Error ('You are not authenticated!');

err.status = 401;

return next(err);

} else {

// ложит раскодированный токен в request для проброски дальше

req.decoded = decoded;

next();

}

});

}

else { // если токен не пришел с запросом с клиента

var err = new Error('No token provided!');

err.status = 403;

return next(err);

}

};

Теперь можно изменить код в routes/dishRouter.js. Единственное, что добавляем, так это строчки

var Verify = require('./verify');

И в каждую обработку (get, post, delete) перед функций коллбеком ставим верификацию:

dishRouter.route('/')

.get(Verify.verifyOrdinaryUser, function (req, res, next) {

Запускаем Монго:

cd C:\Program Files\MongoDB\Server\3.2\bin

mongod.exe --dbpath="D:\Git\my\_notes\server\_side\node-examples\mongo\data"

Запускаем сервер:

npm start

Если теперь попытаться зайти в http://localhost:3000/dishes с GET POST DELETE, придет ответ, что вы не авторизированы

Для авторизации идем <http://localhost:3000/users/register>, выбираем метод POST и формат отправки запроса - json, пишем в body:  
{"username":"Alex", "password":"1234"}

Придет ответ, что регистрация успешно выполнена. Для входа, отправляем по ссылке <http://localhost:3000/users/login> то же самое тело запроса, методом POST, что и при регистрации. Сервер пришлет сгенерированный токен в ответ. Скопируем длинную строку с токеном. Вбиваем теперь ссылку <http://localhost:3000/dishes>, в header вставляем новую строку - name: x-access-token, value: \_наш токен\_. Доступ открыт. Теперь можно производить любые операции, просто включая этот токен в хедер запроса

В задании было сделать проверку на администратора для совершения некоторых действий (например, удаление блюд или добавление нового). Для этого создаем нового юзера в БД с полем admin: true. Потом в файле verify.js создаем копию функции verifyOrdinaryUser, только в обработчик ошибок добавляем следующее поле:

jwt.verify(token, config.secretKey, function (err, decoded){

if (err || !decoded.\_doc.admin){ ...

Внимание! Теперь нужно перелогиниться, что бы сервер прислал новый сгенерированный токен, т.к. в БД где-то сохраняются старые значения токенов, и пишет, что admin: false, когда на самом деле admin:true (хранятся скорее всего в самом теле токена)

Будет выдавать ошибку, если в возвращенном пользователе поле admin === false

**Дополнительные ресрусы:**

* <https://scotch.io/tutorials/build-a-restful-api-using-node-and-express-4>
* <http://adrianmejia.com/blog/2014/10/01/creating-a-restful-api-tutorial-with-nodejs-and-mongodb/>
* <http://cwbuecheler.com/web/tutorials/2014/restful-web-app-node-express-mongodb/>
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Basic_access_authentication>
* <http://expressjs-book.com/index.html%3Fp=128.html> - cookies в Express
* <http://javabeat.net/expressjs-session-management/> - Express session management
* <http://passportjs.org/docs> - доки по модулю пасспорт
* <https://jwt.io/> - json web token site
* <https://tools.ietf.org/html/rfc7519> - стандарт по json web token
* <http://mherman.org/blog/2015/01/31/local-authentication-with-passport-and-express-4/#.V2r0JISLTix> - статья, аутентификация юзеров
* <https://scotch.io/tutorials/authenticate-a-node-js-api-with-json-web-tokens> - аутентификация с помощью веб токенов
* <https://www.sitepoint.com/using-json-web-tokens-node-js/>
* <https://stormpath.com/blog/token-auth-spa>
* <https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication>
* <https://scotch.io/tutorials/the-anatomy-of-a-json-web-token>

**Ссылки на другие документы внутри Mongoose (Population)**

NoSQL DB не поддерживают ссылки через JOIN как в SQL, тут работает ссылка на схему:

var commentSchema = new Schema ({

rating: {type: Number, min: 1, max: 5},

comment: {type: String, required: true},

postedBy: {

type: mongoose.Schema.Types.ObjectId, // указание, что тип - ссылка

ref: 'User' // сослались на схему юзера

}

}, {timestamps: true});

Использование такой схемы будет следующим:

Dishes.find({}) // вытащили все блюда из коллекции

.populate('comments.postedBy') // присоединили коллекцию комментов

.exec(function (err, dish){

if(err) throw err;

res.json(dish); // выдали результат - все блюда с комментами

});

**Использование mongoose population на примере**

Будем править код из предыдущего упражнения. Открываем models/user.js и добавляем к нему поля firstname и lastname:

var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var passportLocalMongoose = require('passport-local-mongoose');

var User = new Schema({

username: String, // если не указать явно, модуль их вставит автоматически

password: String, // если их не указать явно, модуль их вставит автоматически

firstname: {type: String, default: ''},

lastname: {type: String, default: ''},

admin: {type: Boolean, default: false}

});

// расширение methods позволяет создавать свои собственные методы схемы

User.methods.getName = function (){

return (this.firstname + ' ' + this.lastname);

};

User.plugin(passportLocalMongoose);

module.exports = mongoose.model('User', User);

Дальше переходим к схеме dishes (models/dishes.js) и меняем там commentSchema на это:

var commentSchema = new Schema({

rating: {type: Number, min: 1, max: 5,required: true}, // валидация по числу

comment: {type: String, required: true},

/\*author: {type: String,required: true}\*/

// вместо автора, делаем ссылку на ID обьекта коллекции User

// это позволит вывести комменты по конкретному блюду через .populate()

// а так же сравнивать ID автора с ID пользователя, для редактирования коммента

postedBy: {type: mongoose.Schema.Types.ObjectId, ref: 'User'}

}, {timestamps: true});

Переходим к роутеру dish (routes/dishRouter.js), находим там обработку комментариев и меняем методы:

var express = require('express');

var dishRouter = express.Router();

var mongoose = require('mongoose');

var Dishes = require('../models/dishes.js');

var Verify = require('./verify');

dishRouter.route('/')

.get(Verify.verifyOrdinaryUser, function (req, res, next) {

Dishes.find({})

.populate('comments.postedBy') // присоединили обьект с комментариям

.exec(function (err, dishes){ // выполнили коллбек-функцию

if(err) throw err;

res.json(dishes)

});

})

.post(Verify.verifyAdmin, function (req, res, next) {

// первый параметр - спарсенное body с клиентского запроса на добавление

Dishes.create(req.body, function (err, dish) {

if (err) throw err;

console.log('Dishes created!');

var id = dish.\_id; // вернет уже dish с id из базы

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Added the dish with id: ' + id);

})

})

.delete(Verify.verifyAdmin, function (req, res, next) {

// удалить все данные из коллекции документов dishes

Dishes.remove({}, function (err, dishes) {

if (err) throw err;

res.json(dishes)

})

});

/\*\*\* КОНКРЕТНОЕ БЛЮДО ПО DISH ID \*\*\*/

dishRouter.route('/:dishId') // post не нужен - мы можем добавить dish в общую коллекцию

.get(function (req, res, next) {

Dishes.findById(req.params.dishId)

.populate('comments.postedBy') // присоединили обьект с комментариями

.exec(function (err, dish){ // выполнили коллбек-функцию

if(err) throw err;

res.json(dish)

});

})

.put(function (req, res, next) {

Dishes.findByIdAndUpdate(

req.params.dishId, // ищет по dishId

{$set: req.body}, // считывает то, что нужно обновить

{new: true}, // передать в коллбек уже обновленный документ

function (err, dish) {

if (err) throw err;

res.json(dish); // покажет обновленный документ

});

})

.delete(function (req, res, next) {

Dishes.findByIdAndRemove(req.params.dishId, function (err, dish) {

if (err) throw err;

res.json(dish);

});

});

/\*\*\* КОММЕНТАРИИ \*\*\*/

dishRouter.route('/:dishId/comments')

.all(Verify.verifyOrdinaryUser) // для всех операций проверяем, зареган ли юзер

.get(function (req, res, next) {

Dishes.findById(req.params.dishId)

.populate('comments.postedBy') // присоединили обьект с комментариями

.exec(function (err, dish){ // выполнили коллбек-функцию

if(err) throw err;

res.json(dish.comments)

});

})

.post(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

if (err) throw err;

// записываем id пользователя из полученного ранее параметра req.decoded

req.body.postedBy = req.decoded.\_doc.\_id;

// пушим коммент, автоматически вставляется так же id пользователя

dish.comments.push(req.body);

dish.save(function (err, dish){

if (err) throw err;

console.log('Updated comments!');

console.log(dish);

res.json(dish);

})

})

})

// разрешим удаление всех комментов только администратору

.delete(Verify.verifyAdmin, function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

if (err) throw err;

// находим количество комментов по данному блюду

for(var i = 0; i < dish.comments.length; i++){

// удаляем по id

dish.comments.id(dish.comments[i].\_id).remove();

}

dish.save(function (err, dish){

if (err) throw err;

res.writeHead(200, {'Content-Type':'text/plain'});

res.end('Deleted all comments!');

})

});

});

/\*\*\* КОНКРЕТНЫЙ КОММЕНТАРИЙ \*\*\*/

dishRouter.route('/:dishId/comments/:commentId')

.all(Verify.verifyOrdinaryUser) // для всех операций проверяем, зареган ли юзер

.get(function (req, res, next) {

Dishes.findById(req.params.dishId)

.populate('comments.postedBy') // присоединили обьект с комментариями

.exec(function (err, dish){ // выполнили коллбек-функцию

if(err) throw err;

res.json(dish.comments.id(req.params.commentId));

});

})

.put(function (req, res, next) {

// не апдейтит, а удаляет старый коммент и пихает новый с другим ID

Dishes.findById(req.params.dishId, function (err, dish){

if (err) throw err;

dish.comments.id(req.params.commentId).remove();

// записываем id пользователя из полученного ранее параметра req.decoded

req.body.postedBy = req.decoded.\_doc.\_id;

dish.comments.push(req.body);

dish.save(function (err, dish){

if(err) throw err;

console.log('Updated comments!');

console.log(dish);

res.json(dish);

});

});

})

.delete(function (req, res, next) {

Dishes.findById(req.params.dishId, function (err, dish) {

// проверим, собственный ли это коммент юзера

if(dish.comments.id(req.params.commentId).postedBy != req.decoded.\_doc.\_id){

var err = new Error('You are not authorized to do this');

err.status = 403;

return next(err);

}

dish.comments.id(req.params.commentId).remove(); // удаляем коммент

dish.save(function (err, resp){ // добавляем новый коммент на его место

if (err) throw err;

res.json(resp);

})

});

});

module.exports = dishRouter;

И наконец меняем регистрацию пользователей для учета имени и фамилии. Переходим в файл routes/users.js и меняем метод регистрации

router.post('/register', function (req, res) {

User.register(

new User({username: req.body.username}),

req.body.password,

function (err, user) { // возвращает ошибку или нового зареганого юзера

if (err) {

return res.status(500).json({err: err}); // вернем статус 500 если ошибка

}

// забираем имя и фамилию из запроса, если они были указаны

if(req.body.firstname) {

user.firstname = req.body.firstname;

}

if(req.body.lastname) {

user.larstname = req.body.lastname;

}

// сохраняет информацию о пользователе, потом проверит аутентификацию

user.save(function (err, user){

passport.authenticate('local')(req, res, function () {

return res.status(200).json({status: 'Registration successful!'});

}); // passport.authenticate

}); // user.save end

}); // User.register

}); // router.post

Что бы увидеть систему в действии, нужно зарегать два юзера - простого и с админ правами и откомментировать какое-то блюдо (предварительно добавить блюдо). Сервер добавит свойство postedBy обьект, взятый из юзеров

**Ресурсы:**

* <http://mongoosejs.com/docs/populate.html> - инфа про population в mongoose
* <https://alexanderzeitler.com/articles/mongoose-referencing-schema-in-properties-and-arrays/> - инфа про populating в mongoose (ссылки на другие коллекции)

**HTTPS в Node**

HTTPS - уже встроенный в Node модуль, ничего доустанавливать не требуется

Открываем созданный выше проект, находим в нем папу bin, а в ней - файл www, открываем.

Там все настройки по умолчанию. Находим строчку var http = require('http'), добавляем после нее:

var https = require('https');

var fs = require('fs'); // через этот модуль будем читать сертификаты SSL

Обычный веб-сервер обычно запускается на порту 80, а защищенный - https - на порту 443. Поскольку node запускается на порту 3000, сделаем тут порт 3443. Чуть ниже, после строчки app.set('port', port); добавляем свою:

app.set('secPort',port+443);

После блока 'Create HTTP server', перед комментарием "Normalize a port into a number", вставляем код своего HTTPS server-a:

/\*\*\* Create HTTPS server \*\*\*/

// самоподписанные сертификаты чисто для теста

var options = {

key: fs.readFileSync(\_\_dirname+'/private.key'),

cert: fs.readFileSync(\_\_dirname+'/certificate.pem')

}

// создаем защищенный сервер с указанными опциями

var secureServer = https.createServer(options,app);

/\*\*\* Listening HTTPS \*\*\*/

secureServer.listen(app.get('secPort'), function(){

console.log('Server listening on port ',app.get('secPort'));

}

secureServer.on('error', onError);

secureServer.on('listening', onListening);

Переходим к управляющему файлу app.js. Смысл в том, что бы перенаправить весь входящий трафик с обычного http-сервера на защищенный https-сервер. В app.js ищем строчку var app = express(); , после нее добавляем:

// перенаправляем весь трафффик на защищенный HTTPS сервер

app.all('\*', function (req, res, next){

if(req.secure) return next(); // если запрос уже защищен - пробрасываем дальше

res.redirect('https://' + req.hostname + ':' + app.get('secPort') + req.url)

});

Далее необходимо создать самоподписанный сертификат. Для windows - openssl. Так же проканало сделать сертификат онлайн. Указал там localhost. Скачал сертификаты, переименовал как нужно - запустил - работает. Но user/registration не работало

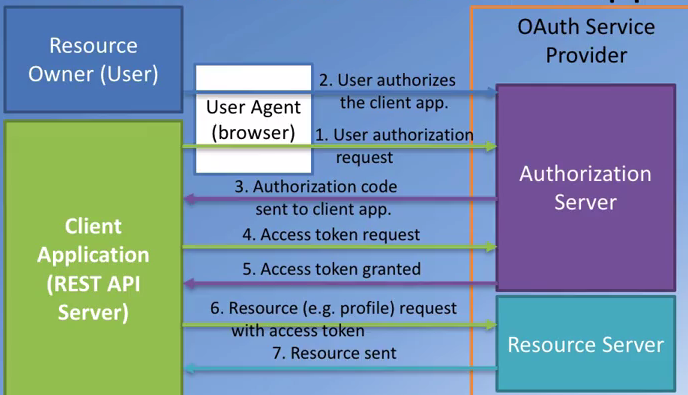
**OAuth авторизация**

Это фреймворк, основанный на открытых стандартах для логина на разные сайты, используя аккаунты соц сети типа фейсбук, гугл и тд

OAuth имеет роли:

* Resource owner - пользователь, который хочет залогиниться на сайте через свой акк
* Client application - веб-сайт или приложение, которое обращается к resource server что бы взять информацию о пользователе
* Resource server - сервер, на котором находятся защищенные персональные данные пользователей
* Authorization server - сервер, который, который выдает токен доступа для client application что бы она сделала запрос на информацию с resource server

Принцип работы:



1. На Authorization server, например Facebook, наш сайт посылает запрос о авторизации пользователя и редиректит пользователя (resource owner) на facebook
2. Authorization server выдает окошко с просьбой залогиниться, юзер логинится.
3. Authorization server выслает назад, на наш сайт, авторизационный код юзера.
4. Наш сайт в ответ высылает назад на сервер фейсбука свой token с App ID, App Secret и Redirect URL. Redirect URL используется, когда Authorization server отдает информацию о пользователе, как callback, что бы вернутся на сайт Client app.
5. Authorization server подтверждет полученную информацию и шлет в ответ Access Token.
6. Client App делет запрос на Resource Server, на котором хранится информация о пользователе, прикрепляя Access Token.
7. Resourse Server высылает в ответ информацию о пользователе в виде JWT

Эта 7-шаговая процедура проходит каждый раз, когда пользователь регистрируется на сайте

**Использование OAuth в Node**

Заходим в папку c заинсталлированным выше Node Passport. Открываем ее в cmd, инсталлим паспорт модуль для фейсбука

npm install passport-facebook --save

Теперь сделаем отдельный файл аутентификации authenticate.js в корне проекта и вынесем туда весь код из app.js, связанный с аутентификацией пользователя:

var passport = require('passport');

var LocalStrategy = require('passport-local').Strategy;

var User = require('./models/user');

var config = require('./config');

// эти строчки удаляем из app.js, поскольку они перенесены сюда

exports.local = passport.use(new LocalStrategy(User.authenticate()));

passport.serializeUser(User.serializeUser());

passport.deserializeUser(User.deserializeUser());

Теперь в config.js в корте проекта добавляем свойства: