

Сейдаметова З.С., Москалева Ю.П.

ЧАСТЬ II

ПРОГРАММИРОВАНИЕ:
УЧЕБНЫЙ ПРОЕКТ НА JAVASCRIPT

УЧЕБНОЕ ПОСОБИЕ

Симферополь
«ДИАЙПИ»
2018

УДК 004.41:378.2

Рекомендовано к печати Ученым Советом ГБОУВО РК «КИПУ»,
протокол №4 от 30.11.2015 г.

Рецензенты:

Чехов В.Н. – зав. кафедрой прикладной математики факультета математики и информатики Таврической Академии Крымского федерального университета имени В.И. Вернадского, доктор физико-математических наук, профессор;

Руденко Л.И. – доцент кафедры информатики Таврической Академии Крымского федерального университета имени В.И. Вернадского, кандидат физико-математических наук, доцент;

Москалева Ю.П. – доцент кафедры прикладной информатики ГБОУВО РК «Крымский инженерно-педагогический университет», кандидат физико-математических наук.

Сейдаметова З.С., Москалева Ю.П. С 28
Программирование: разработка изоморфного приложения на
Java Script. Учебное пособие / З.С. Сейдаметова, Ю.П. .
– Симферополь: ДИАЙПИ, 2017. – ??? с.
ISBN 978-5-906821-30-0

Учебное пособие содержит программу государственного экзамена, а также методические рекомендации по подготовке и защите выпускных квалификационных работ квалификационных уровней бакалавр и магистр [направление подготовки 09.03.03 «Прикладная информатика», профиль «Прикладная информатика в информационной сфере» и направление подготовки 09.04.03 «Прикладная информатика», магистерская программа «Прикладная информатика в информационной сфере»].

Учебное пособие включает в себя порядок организации, процедуру проведения, программу (государственного экзамена) и требования к оформлению (ВКР), требования к результатам освоения образовательной программы в рамках соответствующего модуля, фонд оценочных средств и рекомендуемую литературу с разделением на тематические рубрики.

Учебное пособие адресовано как студентам указанных направлений подготовки, так и для преподавателей, осуществляющих подготовку студентов.

УДК 004.41:378.2

ББК: 32.97

СОДЕРЖАНИЕ

<u>1. Введение</u>	5
<u>2. Сервер на express</u>	6
<u>2.1. Простейший сервер на express</u>	6
<u>2.2. Обработка post запросов</u>	10
<u>2.3. Стартовый проект express</u>	15
<u>2.4. Анализ кода стартового проекта</u>	16
<u>2.5. Выполнение кода JavaScript</u>	19
<u>3. MongoDB</u>	20
<u>3.1. Консоль mongo</u>	20
<u>3.2. Команда FIND</u>	23
<u>3.3. Команда UPDATE</u>	28
<u>3.4. Упражнения для закрепления правил синтаксиса</u>	31
<u>4. Маршрутизаторы и шаблоны стартового express проекта</u>	32
<u>4.1. Стартовый проект с шаблонизатором ejs</u>	32
<u>4.2. Маршрутизатор</u>	34
<u>4.3. Создание шаблона</u>	35
<u>4.4. Установка и подключение шаблонизатора ejs-locals</u>	38
<u>5. Навигация</u>	40
<u>5.1. Установка и подключение Bootstrap</u>	41
<u>5.2. Навигационное меню</u>	42
<u>5.3. Переход между страницами</u>	43
<u>6. Модуль mongodb, подключение базы данных</u>	43
<u>6.1. Подключение базы данных</u>	43
<u>6.2. Создание модуля данных</u>	45
<u>6.3. Данные приложения</u>	46
<u>7. Модуль async</u>	47
<u>7.1. Установка async</u>	47
<u>7.1. Функция each</u>	48
<u>7.2. Шаблон series</u>	50
<u>7.3. Шаблон parallel</u>	52
<u>7.4. Шаблон waterfall</u>	53
<u>7.5. Утилита apply</u>	54
<u>8. Модуль mongoose, создание моделей данных</u>	55
<u>8.1. ORM</u>	55

<u>8.2. Создание схемы</u>	56
<u>8.3. Создание модели данных приложения</u>	57
<u>8.4. Заполнение базы данных с mongoose</u>	61
<u>8.5. Применение шаблона series</u>	63
<u>8.6. Данные приложения с mongoose</u>	65
<u>9. Индексирование в базах данных</u>	66
<u>9.1. Что такое индексирование</u>	66
<u>9.2. Проверка индексов</u>	67
<u>9.3. Индексирование базы all</u>	68
<u>10. Отображение данных в браузере</u>	71
<u>10.1. Подключение базы данных</u>	71
<u>10.2. Обработка параметра в адресе</u>	71
<u>10.3. Извлечение данных из базы</u>	73
<u>10.4. Чистка кода</u>	74
<u>10.4. Заполнение меню из базы данных</u>	74
<u>11. Cookie и Session</u>	80
<u>11.1. Установка модуля express-session и настройка cookie</u>	80
<u>11.2. Команда записи в cookie</u>	82
<u>11.3. Сохранение session в MongoDB</u>	83
<u>11.4. Создание счетчика посещения страниц сайта</u>	84
<u>11.5. Глобальная переменная для навигации</u>	86
<u>12. Аутентификация</u>	89
<u>12.1. Создание страницы регистрации</u>	89
<u>12.2. Создание модели User</u>	91
<u>12.3. Подготовка данных для передачи на сервер</u>	94
<u>12.3. Логика пользователя</u>	95
<u>12.4. Глобальная переменная user</u>	97
<u>12.5. Обработка ошибки аутентификации</u>	99
<u>12.6. Функциональность logout</u>	100
<u>12.7. Закрытие страниц сайта для незалогиненного пользователя</u>	101

1. ВВЕДЕНИЕ

Учебное пособие посвящено изучению языка программирования JavaScript. Методологически, изучение основ JavaScript, реализуется в рамках разработки учебного проекта. Стек технологий, выбранный для реализации учебного проекта: Node.js, Express.js, MongoDB, EJS.

Что такое Node.js? Для получения корректного ответа на этот вопрос можно обратиться к официальному сайту <https://nodejs.org> :

«Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.»

То есть Node.js это среда выполнения JavaScript. JavaScript имеет две среды выполнения. Исторически первая среда выполнения JavaScript – это браузер. С помощью JavaScript веб-страницам придается интерактивность. Вторая – Node.js, которая поставляется с большим количеством модулей для разработки web-сервера.

Express.js – один из модулей Node.js в котором реализован многочисленный функционал обмена данными с браузером.

MongoDB – база данных. Использование базы данных при разработке учебного проекта делает учебный проект хорошей практикой современного программирования. Выбор именно MongoDB обусловлен прежде всего удобными и гибкими модулями для работы Node.js с MongoDB.

EJS (Embedded JavaScript) – это простой шаблонизатор, который удобно использовать с Node.js и Express.js. Шаблонизаторы позволяют структурировать веб-страницы. В учебном проекте простота EJS позволяет рассмотреть роль шаблонизаторов в разработке веб-приложений.

Выбранный стек технологий: Node.js, Express.js, MongoDB и EJS позволяет понять базовые составляющие процесса разработки проекта. Одним из популярных стеков технологий на сегодняшний день является стек MEAN (MongoDB, Express.js, Angular.js, Node.js). Три технологии из стека MEAN выбраны в настоящем пособии для реализации учебного проекта.

2. СЕРВЕР НА EXPRESS

2.1. ПРОСТЕЙШИЙ СЕРВЕР НА EXPRESS

Первый шаг – это установка Node.js. Установка Node.js зависит от операционной системы и как для любого процесса инсталляции, пошаговая, подробная инструкция ищется, по ключевым словам (например, «установка Node.js»), в интернете.

Устанавливаем Node.js и открываем консольное приложение компьютера (для OS Window рекомендуем сразу скачать и установить приложение cygwin, приложение cygwin – это проверенное разработчиками на Window, приложение, которое с гарантией будет поддерживать все команды учебного пособия).

Для проверки, установлен ли Node.js, в консольном приложении выполним команду:

```
$ node -v  
v7.6.0
```

«Выполнить команду» значит написать команду в консоли и нажать клавишу *Enter*.

Команда `node -v` это запрос версии Node.js.

Разработку проекта начнем с создания папки проекта. Место расположения папки не является важным. Создадим папку проекта:

```
$ mkdir folder  
$ cd folder  
$ pwd
```

команда	объяснение
<code>mkdir folder</code>	создание папки с именем <i>folder</i>
<code>cd folder</code>	вход в папку <i>folder</i>
<code>pwd</code>	проверка где находимся, убеждаемся, что действительно в папке <i>folder</i>

Для навигации по файловой системе из командной строки достаточно знать несколько команд. Базовыми командами являются *cd* – перейти, *pwd* – проверить где находимся, *ls* – посмотреть содержимое папки в которой находимся.

команда	объяснение
<i>cd /полный/путь/к/папке</i>	переход в любую папку по полному пути
<i>cd имя</i>	переход в подкаталог, то есть папка должна быть там где мы находимся
<i>cd ..</i>	переход на один уровень выше
<i>ls</i>	просмотр содержимого папки в которой находимся
<i>ls -a</i>	просмотр содержимого со скрытыми папками
<i>pwd</i>	проверка где находимся

Откроем папку проекта в любой IDE (Integrated Development Environment), то есть в любой среде разработки. В интернет пространстве можно легко найти рейтинги используемости IDE, так например, в последнее время в русскоязычном сегменте интернета набирают популярность бесплатные IDE Sublime Text и Atom. Большим уважением пользуются платные WebStorm и PhpStorm.

Вернемся в консольное приложение и выполним команду *npm init* в папке *folder*, то есть в папке учебного проекта. *npm* – это *node package manager*, то есть менеджер управления модулями Node.js (готовыми, написанными на JavaScript пакетами). *npm* автоматически устанавливается вместе с Node.js.

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.
```

```
...
```

```
Press ^C at any time to quit.
name: (folder)
version: (1.0.0)
description:
entry point: (index.js)
```

```
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/dnsuser/folder/package.json:
```

```
{
  "name": "folder",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Is this ok? (yes)

Команда *npm init* создала в папке *folder* файл *package.json*.
Вернемся в среду разработки для проверки факта появления файла *package.json* в папке проекта и проверки его содержимого.

Листинг файла package.json

```
{
  "name": "folder",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Формат файла – это *JSON* (*JavaScript Object Notation* или представление объектов JavaScript) формат. Что такое *JSON*? *JSON* (*JavaScript Object Notation*) – это способ хранения данных, которые организованы по следующим правилам:

➤ данные состоят из пар имя/значение;

- пары заключаются либо в фигурные скобки {}, либо в квадратные [] и разделяются запятыми. В случае фигурных скобок данные называют объектом, в случае квадратных – массивом;
- пара имя/значение состоит из имени, заключенного в кавычки, за которым следует : и значение
- значение может быть числом (целым или с точкой), строкой, логическим значением (true или false), массивом, объектом или значением null

В консоле выполним следующую команду.

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

В директории *folder* появляется папка *node_modules*. В эту папку *npm* будет складывать все Node.js модули, которые понадобятся для разработки проекта. *--save* добавляет имя и версию установленного модуля в *package.json* файл.

Листинг файла *package.json*

```
{  
  "name": "folder",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.15.4"  
  }  
}
```

Создадим в папке *folder* файл *app.js*. Для создания простейшего проекта в файле *app.js* достаточно написать 6 строчек кода.

Листинг файла *app.js*

```
var express = require("express")  
var app = express()
```

```
app.get("/", function(req, res){  
  res.send("Серверная информация для браузера")  
})  
  
app.listen("3000")
```

В этих шести строчках – подключаем модуль *express*, инициализируем приложение, прописываем обработку рутового маршрута браузера */*, командой *listen()* запускаем Node.js сервер, *3000* – указываем порт доступа из браузера к запущенному серверу.

Перейдем к запуску проекта.

```
$ node app.js
```

Откроем теперь браузер. На странице <http://localhost:3000/> проверяем результат. Текст, отправленный с сервера должен появиться в браузере.

Для остановки Node.js сервера, в консольном приложении выполним инструкцию из ответа *npm* на команду *npm init*.

Press ^C at any time to quit.

Нажимаем ^C (ctrl+c).

Таким образом решена следующая задача: На странице браузера <http://localhost:3000/> получена информация, переданная с Node.js сервера.

Простейший проект создан. Рассмотрим в его рамках несколько возможностей *express*.

2.2. ОБРАБОТКА POST ЗАПРОСОВ

Остановимся подробнее на обработке *post* запросов.

Обмен данными между браузером и сервером происходит с помощью HTTP (HyperText Transfer Protocol) запросов, самые базовые методы это *get* и *post*. Набранный в адресной строке браузера *url* иницирует

get запрос. Поэтому для обработки запроса `http://localhost:3000/` в коде сервера использовался метод `app.get`.

Сделаем следующую модификацию этого метода в коде сервера.

В листинге файла `app.js`

```
app.get('/',function(req,res){
  var form = '<!doctype html>'+
    '<html lang="ru">'+
    '<head>'+
    '<meta charset="UTF-8">'+
    '<title>Форма</title>'+
    '</head>'+
    '<body>'+
    '<h1>Форма для отправки данных на сервер</h1>'+
    '<form action="" method="post">'+
    '<textarea name="" id="" cols="30"
rows="10"></textarea><br/>'+
    '<input type="submit" value="Отправить данные на сервер"/>'+
    '</form>'+
    '</body>'+
    '</html>'
  res.send(form)
})
```

Прежде чем запустить сервер добавим в функцию `listen` так называемую `callback` функцию – функцию, которая отрабатывает после завершения работы вызванной функции. В случае вызова `express` функции `listen`, `callback` функция отработает после завершения работы функции `listen`, т.е. после запуска сервера.

В листинге файла `app.js`

```
app.listen("3000",function(){
  console.log( "Сервер работает и слушает порт: 3000")
})
```

JavaScript функция `console.log()` делает запись в `Console` и используется для контроля процесса разработки.

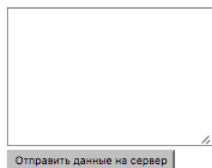
Запустим сервер

```
$ node app.js
```

```
Сервер работает и слушает порт: 3000
```

и перейдем в браузер. По адресу `http://localhost:3000/`, на странице должна отобразиться форма

Форма для отправки данных на сервер



Отправить данные на сервер

Подготовим обработку запроса `post`. Для этого:

Первое: добавим маршрутизатор для запроса с адресом `"/submit_result"`.

В листинге файла `app.js`

```
app.post("/submit_result", function(req, res){  
  res.send("Вы нажали на кнопку с типом submit");  
})
```

Второе: добавим в форму `action="/submit_result"` – адрес отправки `post` запроса.

Перейдем к запуску проекта.

```
$ node app.js
```

В окне браузера кликнем правой кнопкой мыши, выберем `Inspect`, откроем вкладку `Network` и кликнем на кнопку формы. Во вкладке `Network` появится строка, содержащая адрес `submit_result` и в браузере откроется страница `http://localhost:3000/submit_result` с текстом «Вы нажали кнопку с типом `submit`». Если открыть `submit_request` запрос, то во вкладке `Headers` можно убедиться, что к серверу обратился запрос типа `post` и во вкладке `Response` проверить что ответ, совпадает с текстом на странице `http://localhost:3000/submit_result`.

Формы предназначены для передачи данных на сервер.

Чтобы данные из формы были включены в серверный запрос для элементов формы определяется атрибут `name`. Добавим элементу формы `textarea` атрибут `name="text"`.

Остановим сервер: ^C.
Запустим сервер: \$node app.js
Перейдем в браузер.

Теперь во вкладке Headers содержатся данные формы.

Для извлечения данных формы из запроса на сервере установим и подключим модуль body-parser.

```
$ npm install body-parser --save
```

Синтаксис подключения модуля body-parser:

```
var bodyParser = require('body-parser');  
app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({ extended: true }));
```

Синтаксис извлечения данных формы:

```
app.post("/submit_result", function(req, res){  
    console.log(req.body)  
    res.send("Вы нажали на кнопку с типом submit")  
})
```

Остановим сервер: ^C.
Запустим сервер: \$node app.js

В форму наберем текст «Текст», нажмем кнопку формы и вернемся в консольное приложение.

```
^C  
$ node app.js  
Сервер работает и слушает порт: 3000  
{ text: 'Текст' }
```

Сделаем заключительную правку и вернем на страницу ответа введенную информацию.

```

app.post("/submit_result",function(req,res){
    console.log(req.body)
    var post_text = req.body.text? "Вы отправили на сервер текст:"
" + req.body.text : "Вы отправили на сервер пустую строку"
    res.send(post_text)
})

```

Остановим сервер: ^C.

Запустим сервер: \$node app.js

Закончим разработку простейшего клиент-сервер приложения и перейдем к следующему уровню.

Листинг файла app.js

```

var express = require("express")
var bodyParser = require("body-parser")

var app = express()
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({extended: true}))

app.get('/',function(req,res){
    var form = '<!doctype html>'+
        '<html lang="ru">'+
        '<head>'+
        '<meta charset="UTF-8">'+
        '<title>Форма</title>'+
        '</head>'+
        '<body>'+
        '<h1>Форма для отправки данных на сервер</h1>'+
        '<form action="/submit_result" method="post">'+
        '<textarea name="text"></textarea><br/><br/>'+
        '<input type="submit" value="Отправить данные на сервер"/>'+
        '</form>'+
        '</body>'+
        '</html>'
    res.send(form)
})

app.post("/submit_result",function(req,res){
    console.log(req.body)
    var post_text = req.body.text? "Вы отправили на сервер текст:"
" + req.body.text : "Вы отправили на сервер пустую строку"
    res.send(post_text)
})

app.listen("3000",function(){

```

```
    console.log( "Сервер работает и слушает порт: 3000")  
  })
```

2.3. СТАРТОВЫЙ ПРОЕКТ EXPRESS

Рассмотрим использование стартового проекта express для создания приложения.

Одной из приятных возможностей express является наличие стартового проекта, то есть проекта, который настроен для быстрого начала разработки и позволяет сосредоточиться на базовых основах программирования.

Для установки и запуск стартового проекта откроем консольное приложение и выполним 2 команды:

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

Первая команда – это просто npm установка Node.js модуля express-generator, параметр -g означает глобальную установку (от слова global). Следующий шаг – это собственно создание стартового клиент-сервер проекта на Node.js, puh – название папки в которую будет развернут проект, параметр -e это установка стартового пакета с шаблонизатором ejs. Если выполнить команду без параметра -e то установится шаблонизатор jade. В настоящем пособии для разработки учебного проекта выбран шаблонизатор ejs, так как ejs очень похож на HTML и требует минимального времени для изучения.

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

```
create : puh/bin
create : puh/bin/www
```

```
install dependencies:
$ cd puh && npm install
```

```
run the app:
$ DEBUG=puh:* npm start
```

Комментарии, которые пишет консоль достойны прочтения и следования им. Перед тем как перейти к разработке следуя этим инструкциям, откроем папку `puh` в IDE для удобного отслеживания происходящих изменений. Как и комментировал консоль – в папке `puh` созданы файлы стартового пакета. В консольном приложении выполним команды и выполним инструкции консоли.

команда	объяснение
<code>cd puh</code>	вход в папку <code>puh</code>
<code>pwd</code>	проверка где находимся, убеждаемся, что действительно в папке <code>puh</code>
<code>npm install</code>	установка модулей из списка "dependencies": { "body-parser": "~1.15.1", "cookie-parser": "~1.4.3", "debug": "~2.2.0", "express": "~4.13.4", "jade": "~1.11.0", "morgan": "~1.7.0", "serve-favicon": "~2.3.0" } файла <code>package.json</code>
<code>npm start</code>	запуск скелета

Перейдем в браузер и в адресной строке введем `http://localhost:3000` и увидим страницу приветствия.

2.4. АНАЛИЗ КОДА СТАРТОВОГО ПРОЕКТА

Откроем файл `app.js`. Исследуем значение `app.get('env')`. Из кода видно, что переменная `'env'` принимает два значения: `development` и

production. 'env' это переменная среды разработки. Управляется эта переменная командами:

команда	объяснение
export NODE_ENV=production	значение 'env' устанавливается равным production
export NODE_ENV=development	значение 'env' устанавливается равным development

Рассмотрим как работает debug модуль. Для использования модуля debug нужно: подключить, выполнить debug, запустить приложение в debug режиме.

подключение:	<code>var debug = require('debug')('puh:server')</code>
выполнение:	<code>debug("Проверка работы debug модуля")</code>
старт приложения:	<code>\$ DEBUG=puh:* npm start</code>
отладка:	<code>puh:server</code> проверка работы debug модуля <code>+0ms</code> <code>puh:server</code> listening on port 3000 <code>+51ms</code>

Рассмотрим механизм отображения страницы приветствия по запросу `http://localhost:3000`. В файле `routes/index.js` код

```
/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

управляет обработкой адреса. "index" – это название шаблона, который отобразится в браузере, title - это переменная, значение которой «Express» передается в шаблон. Шаблон располагается в файле `views/index.ejs`.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
```

```
</body>  
</html>
```

Как уже упоминалось – синтаксис шаблонизатора `ejs` очень прост в изучении. При выводе данных в браузере надо иметь в виду существование спецсимволов: двойная кавычка, амперсанд, апостроф, знак меньше, знак больше. Экранированный вывод – это вывод, когда спецсимволы выводятся как обычные символы. Соответственно неэкранированный – это вывод HTML.

`<%= title %>` – экранированный вывод данных
`<%- title %>` – неэкранированный вывод данных

Добавим в значение переменной `title` спецсимволы:

```
/* GET home page. */  
router.get('/', function(req, res, next) {  
  res.render('index', { title: '<h1>Express</h1>' });  
});
```

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

Откроем браузер:



<h1>Express</h1>

Welcome to <h1>Express</h1>

Итак, `<%= title %>` вывод – это вывод, когда спецсимволы выводятся как обычные символы, то есть экранированный вывод. Изменим теперь `<%= title %>` вывод на `<%- title %>` – неэкранированный вывод (файл `views/index.ejs`). Теперь в браузере спецсимволы распознаются как спецсимволы и теги `<h1>` интерпретируются как HTML.

Сведем правила синтаксиса `ejs` в таблицу.

ejs синтаксис	объяснение
<% код JavaScript %>	код JavaScript для формирования шаблона
<%= переменная %>	экранированный вывод
<%- переменная %>	неэкранированный вывод
->	закрывающий тег предотвращающий новую строку шаблона
<%_ _>	удаление пробелов

2.5. ВЫПОЛНЕНИЕ КОДА JAVASCRIPT

Напомним, что код JavaScript выполняется двумя способами: в браузере или Node.js.

Выполнение JavaScript в браузере

Для выполнения в браузере можно создать файл с расширением html и в теге script писать JavaScript. Файл можно создавать в любом месте файловой системы. Например, создадим файл file.html:

Листинг файла file.html

```
<!doctype html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <title>JavaScript</title>
</head>
<body>
  <script>
    //Код JavaScript
  </script>
</body>
</html>
```

Выполнить код – это открыть html файл в браузере. В окне браузера кликнуть правой кнопкой мыши, выбрать Inspect и открыть вкладку Console.

Код JavaScript можно писать в теге script, а можно подключить js файл и писать код во внешнем файле. Для этого создадим папку

folder (папку можно создать в любом месте файловой системы). В папке folder создадим 2 файла: file.html, file.js.

Для выполнения кода нужно тоже открыть html файл в браузере. И в окне браузера кликнуть правой кнопкой мыши и выбрать Inspect и открыть вкладку Console.

Выполнение JavaScript с Node.js

Для выполнения JavaScript с использованием Node.js переходим в папку folder и набираем команду:

```
$ node file.js
```

Выполнить команду это нажать клавишу enter.

3. MONGODB

3.1. КОНСОЛЬ MONGO

Как и для Node.js, установка MongoDB зависит от операционной системы и как для любого процесса инсталляции, пошаговая, подробная инструкция ищется, по ключевым словам (например, «установка MongoDB»), в интернете.

В консольном приложении выполним команду:

```
$ mongod -version  
db version v3.4.7
```

Рассмотрим базовые команды

команда	объяснение
mongod	запуск базы MongoDB
mongo	запуск консоли MongoDB
show dbs	список баз данных

Войдем в консоль MongoDB и выполним команду show dbs

use name	переключение а базу name
db.help()	список доступных функций
db.getCollectionNames()	список коллекций базы
db.getName()	проверка в какой базе находимся
db.collection_name.insert()	добавление данных в базу коллекцию collection_name

```
$ mongo
> show dbs
test  0.000GB
```

В MongoDB ровно одна база с именем test. Следующим шагом выполним команды use learn и db.unicorns.insert:

```
> use learn
switched to db learn
> db.unicorns.insert({name: 'Aurora', color: 'white', weight:
450})
WriteResult({ "nInserted" : 1 })
> db.getCollectionNames()
[ "unicorns" ]
> db.unicorns.count()
1
```

И еще раз show dbs:

```
> show dbs
test      0.000GB
learn    0.000GB
```

Итак, к списку баз данных добавилась новая база данных learn. Таким образом для создания базы данных из консоли достаточно переключиться и добавить данные.

Рассмотрим терминологию и структуру хранения данных в MongoDB.

термины	структура
база	база состоит из коллекций
коллекция	коллекция состоит из документов
документ	документ состоит из пар поле/значение поля

Для данных unicorns:

learn – база

unicorns – коллекция

{name: 'Aurora', color: 'white', weight: 450} – документ

name, color, weight – поля

Удалим базу данных learn и проверим, что база данных удалена.

```
> show dbs
test      0.000GB
learn     0.000GB
> use learn
switched to db learn
> db.dropDatabase()
{ "dropped" : "learn", "ok" : 1 }
> show dbs
test      0.000GB
```

Подготовим базу данных для выполнения упражнений:

```
> use learn
switched to db learn
```

Скопируем и вставим в консоль добавление документов в коллекцию unicorns.

```
db.unicorns.insert({name: 'Horny', birthday: new
Date(1992,2,13,7,47), loves: ['carrot','papaya'], weight: 600,
color: 'black', vampires: 63});
db.unicorns.insert({name: 'Aurora', birthday: new Date(1991, 0,
24, 13, 0), loves: ['carrot', 'grape'], weight: 450, color:
'white', vampires: 43});
db.unicorns.insert({name: 'Unicrom', birthday: new Date(1973, 1,
9, 22, 10), loves: ['energon', 'redbull'], weight: 984, color:
'black', vampires: 182});
db.unicorns.insert({name: 'Roodles', birthday: new Date(1979, 7,
18, 18, 44), loves: ['apple'], weight: 575, color: 'black',
vampires: 99});
db.unicorns.insert({name: 'Solnara', birthday: new Date(1985, 6,
4, 2, 1), loves:['apple', 'carrot', 'chocolate'], weight:550,
color:'white', vampires:80});
db.unicorns.insert({name:'Ayna', birthday: new Date(1998, 2, 7, 8,
30), loves: ['strawberry', 'lemon'], weight: 733, color: 'white',
vampires: 40});
```

```

db.unicorns.insert({name:'Kenny', birthday: new Date(1997, 6, 1,
10, 42), loves: ['grape', 'lemon'], weight: 690, color: 'black',
vampires: 39});
db.unicorns.insert({name: 'Raleigh', birthday: new Date(2005, 4,
3, 0, 57), loves: ['apple', 'sugar'], weight: 421, color: 'black',
vampires: 2});
db.unicorns.insert({name: 'Leia', birthday: new Date(2001, 9, 8,
14, 53), loves: ['apple', 'watermelon'], weight: 601, color:
'white', vampires: 33});
db.unicorns.insert({name: 'Pilot', birthday: new Date(1997, 2, 1,
5, 3), loves: ['apple', 'watermelon'], weight: 650, color:
'black', vampires: 54});
db.unicorns.insert({name: 'Nimue', birthday: new Date(1999, 11,
20, 16, 15), loves: ['grape', 'carrot'], weight: 540, color:
'white'});
db.unicorns.insert({name: 'Dunx', birthday: new Date(1976, 6, 18,
18, 18), loves: ['grape', 'watermelon'], weight: 704, color:
'black', vampires: 165});

```

Убедимся, что создана база данных learn, что создана коллекция unicorns и все 12 документов добавлены.

```

> show dbs
learn  0.000GB
test   0.000GB
> db.getName()
learn
> db.getCollectionNames()
[ "unicorns" ]
> db.unicorns.count()
12

```

Итак, мы подняли MongoDB, изучили структуру, создали базу learn, в базе learn коллекцию unicorns и добавили в эту коллекцию 12 документов. Для добавления документов в коллекцию была использована команда insert.

3.2. КОМАНДА FIND

Изучим поиск с условиями: и, или(\$or), не из списка (\$nor), меньше(\$lt), меньше или равно(\$lte), больше(\$gt), больше или равно(\$gte), не равно(\$ne), существует(\$exists).

1	Найти черных единорогов, которые весят больше 700
2	Найти не черных единорогов вес которых не меньше 701
3	Найти единорогов без статистики убитых вампиров
4	Найти белых единорогов которые весят меньше 500

5	Найти сколько единорогов не весят 600
6	Найти черных единорогов которые любят арбузы или весят больше 900
7	Найти сколько единорогов любят яблоки или морковь
8	Найти сколько единорогов не любят яблоки и не любят морковь
9	Найти единорогов, которые любят виноград или морковь
10	Найти единорогов которые любят яблоки или убили вампиров не меньше 63

Решение:

1. `db.unicorns.find({color:"black",weight:{$gt: 700}})`.

Синтаксис запроса – это фигурные скобки {} и условия, которые должны выполняться одновременно через запятую, реализация запроса с условием «и»: единорог должен быть черным И(запятая) вес его должен быть больше 700. Условие для поля weight (`weight > 700`) должно быть в формате `:$gt: 700`.

Выполним запрос в консоли MongoDB:

```
> db.unicorns.find({color:"black",weight:{$gt: 700}})
{ "_id" : ObjectId("598dab792ac4bb803a663664"), "name" :
"Unicrom", "birthday" : ISODate("1973-02-09T19:10:00Z"), "loves" :
[ "energon", "redbull" ], "weight" : 984, "color" : "black",
"vampires" : 182 }
{ "_id" : ObjectId("598dab792ac4bb803a66366d"), "name" : "Dunx",
"birthday" : ISODate("1976-07-18T15:18:00Z"), "loves" : [ "grape",
"watermelon" ], "weight" : 704, "color" : "black", "vampires" :
165 }
```

При добавлении документа в базу, MongoDB добавляет к документу поле `_id` с уникальным значением.

2. `db.unicorns.find({color:{$ne:"black"},weight:{$gte:701}})`

Используем синтаксис `$ne` (не равно) и условие `$gte` (больше или равно)

Выполним запрос в консоли MongoDB:

```
> db.unicorns.find({color:{$ne:"black"},weight:{$gte:701}})
{ "_id" : ObjectId("598dab792ac4bb803a663667"), "name" : "Ayna",
"birthday" : ISODate("1998-03-07T06:30:00Z"), "loves" : [
"strawberry", "lemon" ], "weight" : 733, "color" : "white",
"vampires" : 40 }
```

Запрос вернул одного единорога.

3. `db.unicorns.find({vampires:{$exists:false}})`

Запрос на документы, в которых отсутствует поле `vampires` – `{ $exists: false }`. Одно из ключевых свойств MongoDB – возможность хранить документы с разными наборами полей в одной коллекции. Так в коллекции `unicorns` есть документы в которых задано поле `vampires` и есть документы в которых это поле не задано. Запрос должен найти документы без поля `vampires`.

Выполним запрос в консоли MongoDB:

```
> db.unicorns.find({vampires:{$exists:false}})
{ "_id" : ObjectId("598dab792ac4bb803a66366c"), "name" : "Nimue",
  "birthday" : ISODate("1999-12-20T14:15:00Z"), "loves" : [ "grape",
  "carrot" ], "weight" : 540, "color" : "white" }
```

4. `db.unicorns.find({color:"white",weight:{$lt:500}})`

Для условия меньше или равно используем синтаксис `{ $lt: 500 }`.

Выполним запрос в консоли MongoDB:

```
db.unicorns.find({color:"white",weight:{$lt:500}})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
  "birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
  "carrot", "grape" ], "weight" : 450, "color" : "white", "vampires"
  : 43 }
```

5. `db.unicorns.find({weight:{$ne:600}}).count()`

Применение условия `$ne` (не равно) уже знакомо. Новым синтаксисом является команда вычисления количества `count()`.

Выполним запрос в консоли MongoDB:

```
> db.unicorns.find({weight:{$ne:600}}).count()
11
```

6

```
db.unicorns.find({color:"black",$or:[{loves:"watermelon"},{weight:
{$gt:900}}]})
```

Синтаксис условия `$or` (или) это условие `$or`, двоеточие, квадратные скобки, в квадратных скобках через запятую условия. Еще одним важным правилом запросов в MongoDB является обращение к

элементам массивов. Поле `love` в документе массив, условие `{loves:"watermelon"}` это поиск по элементу массива. Запрос будет искать единорогов у которых в списке `loves` есть арбуз.

Выполним запрос в консоли MongoDB:

```
>
db.unicorns.find({color:"black",$or:[{loves:"watermelon"},{weight:
{$gt:900}}]})
{ "_id" : ObjectId("598dab792ac4bb803a663664"), "name" :
"Unicrom", "birthday" : ISODate("1973-02-09T19:10:00Z"), "loves" :
[ "energon", "redbull" ], "weight" : 984, "color" : "black",
"vampires" : 182 }
{ "_id" : ObjectId("598dab792ac4bb803a66366b"), "name" : "Pilot",
"birthday" : ISODate("1997-03-01T02:03:00Z"), "loves" : [ "apple",
"watermelon" ], "weight" : 650, "color" : "black", "vampires" : 54
}
{ "_id" : ObjectId("598dab792ac4bb803a66366d"), "name" : "Dunx",
"birthday" : ISODate("1976-07-18T15:18:00Z"), "loves" : [ "grape",
"watermelon" ], "weight" : 704, "color" : "black", "vampires" :
165 }
>
```

Получили три единорога. Единорог Unicorn был найден потому что черный и весит больше 900 (арбузы не любит). Единорог Pilot был найден потому что он черный и потому что любим арбузы (вес у него меньше 900). Единорог Dunx, найден как и Pilot, потому что черный и потому что любит арбузы, хотя вес его меньше 900.

7

```
db.unicorns.find({$or:[{loves:'apple'},{loves:"carrot"}]}).count()
```

Повторяем синтаксис запроса `$or`(или) и обращение к элементам массива.

Выполним запрос в консоли MongoDB:

```
>
db.unicorns.find({$or:[{loves:'apple'},{loves:"carrot"}]}).count()
8
```

Найдено восемь единорогов.

```
8 db.unicorns.find({$nor:[{loves:'apple'},{loves:"carrot"}]}))
```

Изучим `$nor` условие.

Выполним запрос в консоли MongoDB:

```
> db.unicorns.find({$nor:[{loves:'apple'},{loves:"carrot"}]})
{ "_id" : ObjectId("598dab792ac4bb803a663664"), "name" :
"Unicrom", "birthday" : ISODate("1973-02-09T19:10:00Z"), "loves" :
[ "energon", "redbull" ], "weight" : 984, "color" : "black",
"vampires" : 182 }
{ "_id" : ObjectId("598dab792ac4bb803a663667"), "name" : "Ayna",
"birthday" : ISODate("1998-03-07T06:30:00Z"), "loves" : [
"strawberry", "lemon" ], "weight" : 733, "color" : "white",
"vampires" : 40 }
{ "_id" : ObjectId("598dab792ac4bb803a663668"), "name" : "Kenny",
"birthday" : ISODate("1997-07-01T07:42:00Z"), "loves" : [ "grape",
"lemon" ], "weight" : 690, "color" : "black", "vampires" : 39 }
{ "_id" : ObjectId("598dab792ac4bb803a66366d"), "name" : "Dunx",
"birthday" : ISODate("1976-07-18T15:18:00Z"), "loves" : [ "grape",
"watermelon" ], "weight" : 704, "color" : "black", "vampires" :
165 }
>
```

Найдено 4 единорога. И они любят все что угодно кроме яблок и морковки.

9. db.unicorns.find({\$or:[{loves:'grape'},{loves:"lemon"}]})

Повторяем синтаксис запроса \$or(или) и обращение к элементам массива.

Выполним запрос в консоли MongoDB:

```
> db.unicorns.find({$or:[{loves:'grape'},{loves:"lemon"}]})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
"birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
"carrot", "grape" ], "weight" : 450, "color" : "white", "vampires"
: 43 }
{ "_id" : ObjectId("598dab792ac4bb803a663667"), "name" : "Ayna",
"birthday" : ISODate("1998-03-07T06:30:00Z"), "loves" : [
"strawberry", "lemon" ], "weight" : 733, "color" : "white",
"vampires" : 40 }
{ "_id" : ObjectId("598dab792ac4bb803a663668"), "name" : "Kenny",
"birthday" : ISODate("1997-07-01T07:42:00Z"), "loves" : [ "grape",
"lemon" ], "weight" : 690, "color" : "black", "vampires" : 39 }
{ "_id" : ObjectId("598dab792ac4bb803a66366c"), "name" : "Nimue",
"birthday" : ISODate("1999-12-20T14:15:00Z"), "loves" : [ "grape",
"carrot" ], "weight" : 540, "color" : "white" }
{ "_id" : ObjectId("598dab792ac4bb803a66366d"), "name" : "Dunx",
"birthday" : ISODate("1976-07-18T15:18:00Z"), "loves" : [ "grape",
```

```
"watermelon" ], "weight" : 704, "color" : "black", "vampires" :  
165 }  
>
```

Это единороги: Aurora, Ayna, Kenny, Nimue, Dunx.

```
10 db.unicorns.find({$or:[{loves:"apple"},{vampires:{$gte:63}}]})
```

3.3. Команда UPDATE

Изучим команду update и условия \$set, \$inc, \$push, \$pull, \$mul.

11	Единорог Roodles похудел на 10
12	Единорог Pilot убил еще 2 вампиров
13	Единорог Aurora полюбил сахар
14	Добавить информацию, что Aurora привита (vaccinated: true)
15	Проверить какие единороги привиты
16	Единорог Dunx разлюбил виноград
17	Вес единорога Aurora увеличился в два раза

Решение:

```
11 db.unicorns.update({name:"Roodles"},{$inc:{weight:-10}})
```

Проанализируем синтаксис команды update. У команды update два аргумента. Первый аргумент – это условие поиска, второй аргумент – это описание изменения, которое нужно сделать. Аргументы разделены запятой. Условие поиска – единорог по имени Roodles. Условие \$inc – изменяет значение поля weight на величину –10.

Сначала выполним в консоли MongoDB поиск единорога с именем Roodles: db.unicorns.find({name:"Roodles"})

```
> db.unicorns.find({name:"Roodles"})  
{ "_id" : ObjectId("598dab792ac4bb803a663665"), "name" :  
"Roodles", "birthday" : ISODate("1979-08-18T15:44:00Z"), "loves" :  
[ "apple" ], "weight" : 575, "color" : "black", "vampires" : 99 }
```

Вес Roodles – 575.

Теперь сделаем update поля weight и снова извлечем сведения о единороге с именем Roodle из базы.

```
> db.unicorns.update({name:"Roodles"},{$inc:{weight:-10}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.unicorns.find({name:"Roodles"})
{ "_id" : ObjectId("598dab792ac4bb803a663665"), "name" :
"Roodles", "birthday" : ISODate("1979-08-18T15:44:00Z"), "loves" :
[ "apple" ], "weight" : 565, "color" : "black", "vampires" : 99 }
>
```

Вес Roodles стал 565, то есть уменьшился на 10.

12 db.unicorns.update({name:"Pilot"},{\$inc:{vampires:2}})

В этом упражнении отрабатывается уже рассмотренное выше условие \$inc.

13 db.unicorns.update({name:"Aurora"},{\$push:{loves:"sugar"}})

Условие запроса \$push добавит к списку сахар.

Сначала убедимся, что единорог с именем Aurora не любит сахар:
db.unicorns.find({name:"Aurora"})

```
> db.unicorns.find({name:"Aurora"})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
"birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
"carrot", "grape" ], "weight" : 450, "color" : "white", "vampires"
: 43 }
```

Действительно Aurora любит морковку и виноград.

Теперь выполним update и снова запросим информацию об единороге Aurora.

```
> db.unicorns.update({name:"Aurora"},{$push:{loves:"sugar"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.unicorns.find({name:"Aurora"})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
"birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
"carrot", "grape", "sugar" ], "weight" : 450, "color" : "white",
"vampires" : 43 }
```

14 db.unicorns.update({name:"Aurora"},{\$set:{vaccinated:true}})

В этом упражнении используется условие `$set` – установить, но особенно интересно, что можно устанавливать значение поля, которого в документе нет.

```
> db.unicorns.find({name:"Aurora"})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
  "birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
    "carrot", "grape", "sugar" ], "weight" : 450, "color" : "white",
  "vampires" : 43 }

> db.unicorns.update({name:"Aurora"},{$set:{vaccinated:true}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.unicorns.find({name:"Aurora"})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
  "birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
    "carrot", "grape", "sugar" ], "weight" : 450, "color" : "white",
  "vampires" : 43, "vaccinated" : true }
```

15 db.unicorns.find({vaccinated:true})

```
> db.unicorns.find({vaccinated:true})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
  "birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
    "carrot", "grape", "sugar" ], "weight" : 450, "color" : "white",
  "vampires" : 43, "vaccinated" : true }
```

16 db.unicorns.update({name:"Dunx"},{\$pull:{loves:"grape"}})

Сначала проверим, что Dunx не любит виноград, затем используем `update` с условием `$pull` и поверим, что полюбил.

```
> db.unicorns.find({name:"Dunx"})
{ "_id" : ObjectId("598dab792ac4bb803a66366d"), "name" : "Dunx",
  "birthday" : ISODate("1976-07-18T15:18:00Z"), "loves" : [ "grape",
    "watermelon" ], "weight" : 704, "color" : "black", "vampires" :
  165 }

> db.unicorns.update({name:"Dunx"},{$pull:{loves:"grape"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.unicorns.find({name:"Dunx"})
{ "_id" : ObjectId("598dab792ac4bb803a66366d"), "name" : "Dunx",
  "birthday" : ISODate("1976-07-18T15:18:00Z"), "loves" : [
    "watermelon" ], "weight" : 704, "color" : "black", "vampires" :
  165 }
```

17 db.unicorns.update({name:"Aurora"},{\$mul:{weight: 2}})

Проверим вес единорога Aurora, выполним команду update и снова проверим вес.

```
> db.unicorns.find({name:"Aurora"})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
  "birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
    "carrot", "grape", "sugar" ], "weight" : 450, "color" : "white",
  "vampires" : 43, "vaccinated" : true }

> db.unicorns.update({name:"Aurora"},{$mul:{weight: 2}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.unicorns.find({name:"Aurora"})
{ "_id" : ObjectId("598dab792ac4bb803a663663"), "name" : "Aurora",
  "birthday" : ISODate("1991-01-24T11:00:00Z"), "loves" : [
    "carrot", "grape", "sugar" ], "weight" : 900, "color" : "white",
  "vampires" : 43, "vaccinated" : true }
```

Вес Aurora был 450, стал 900, то есть увеличился в два раза.

3.4. УПРАЖНЕНИЯ ДЛЯ ЗАКРЕПЛЕНИЯ ПРАВИЛ СИНТАКСИСА

Упражнения для закрепления правил синтаксиса.

Исправить ошибки в запросах:

1	db.unicorns.update({name="Roooooodles"},{\$inc:{weight:-10}})
2	db.unicorn.update({name:'Kenny'},{\$set:{weight:603}})
3	db.unicorns.find({gender:"m"},{weight:{\$gt:500}})
4	db.unicorns.find().sort({vampires})
5	db.unicorns.find(name:"Pilot")
6	db.unicorns.update({name:"Pilot", \$pull:{loves:"apple"}})
7	db.unicorns.find({color: "white"},{weight:{\$lt:500}})
8	db.unicorns.update({name:"Pilot"},{ vampires:{ \$inc:2})
9	db.unicorns.update({name:"Roodles"},\$inc:{weight:-10})
10	db.unicorns.find({color: "white",weight:{\$gt:500}})
11	db.unicorns.find({\$or:[{loves:"apple"},{vampires:{\$ghe:63}}]})

Ответы:

1	db.unicorns.update({name="Roodles"},{\$inc:{weight:-10}})
	db.unicorns.update({name: "Roodles"},{\$inc:{weight:-10}})

2	db.unicorn.update({name:'Kenny'},{\$set:{weight:603}})
	db.unicorns.update({name:'Kenny'},{\$set:{weight:603}})
3	db.unicorns.find({color:"black"},{weight:{\$gt:500}})
	db.unicorns.find({color:"black",weight:{\$gt:500}})
4	db.unicorns.find().sort({vampires})
	db.unicorns.find().sort({vampires:-1})
5	db.unicorns.find(name:"Pilot")
	db.unicorns.find({name:"Pilot"})
6	db.unicorns.update({name:"Pilot", \$pull:{loves:"apple"}})
	db.unicorns.update({name:"Pilot"},{\$pull:{loves:"apple"}})
7	db.unicorns.find({color: "white"},{ weight:{\$lt:500}})
	db.unicorns.find({color: "white", weight:{\$lt:500}})
8	db.unicorns.update({name:"Pilot"},{ vampires:{ \$inc:2}})
	db.unicorns.update({name:"Pilot"},{\$inc:{vampires:2}})
9	db.unicorns.update({name:"Roodles"},\$inc:{weight:-10})
	db.unicorns.update({name:"Roodles"},{\$inc:{weight:-10}})
10	db.unicorns.find({color: "white",weight:{\$gt:500}})
	db.unicorns.find({color: "white",weight:{\$gt:500}})
11	db.unicorns.find({\$or:[{loves:"apple"},{vampires:{\$ghe:63}}]})
	db.unicorns.find({\$or:[{loves:"apple"},{vampires:{\$gte:63}}]})

Продолжить изучение MongoDB с единорогами можно здесь:
<https://docs.mongodb.com/manual/>.

4. МАРШРУТИЗАТОРЫ И ШАБЛОНЫ СТАРТОВОГО EXPRESS ПРОЕКТА

4.1. СТАРТОВЫЙ ПРОЕКТ С ШАБЛОНИЗАТОРОМ EJS

Создадим новый стартовый express проект.

команда	описание
pwd	проверка где находимся, убеждаемся, что не находимся в папке никакого проекта

Выполним команду

```
$ express all -e
```



```
express all -e
```

```
create : all
create : all/package.json
create : all/app.js
create : all/public
create : all/public/images
create : all/public/stylesheets
create : all/public/stylesheets/style.css
create : all/routes
create : all/routes/index.js
create : all/routes/users.js
create : all/views
create : all/views/index.ejs
create : all/views/error.ejs
create : all/bin
create : all/bin/www
create : all/public/javascripts
```

```
install dependencies:
$ cd all && npm install
```

```
run the app:
$ DEBUG=all:* npm start
```

Это создание стартового клиент-сервер проекта на Node.js, `all` – название папки в которую будет развернут проект, параметр `-e` это установка стартового пакета с шаблонизатором `ejs`. Напомним, что если выполнить команду без параметра `-e` то установится шаблонизатор `jade`.

После отчета о создании скелета проекта в консоле прописаны следующие шаги:

команда	объяснение
<code>cd all</code>	вход в папку <code>all</code>
<code>pwd</code>	проверка где находимся, убеждаемся, что действительно в папке <code>all</code>
<code>npm install</code>	установка модулей из списка <pre>"dependencies": { "body-parser": "~1.15.1", "cookie-parser": "~1.4.3", "debug": "~2.2.0", "express": "~4.13.4", "jade": "~1.11.0", "morgan": "~1.7.0",</pre>

	<code>"serve-favicon": "~2.3.0"</code> }
файла <i>package.json</i>	
<code>npm start</code>	запуск скелета

4.2. МАРШРУТИЗАТОР

В главном файле приложения `app.js` найдем строку

```
app.use(express.static(path.join(__dirname, 'public')));
```

Это объявление `public` папки. Файлы, к которым разрешен доступ из браузера. Остальные файлы проекта защищены из соображений безопасности приложения.

Из структуры папки (в папке `public` расположены папки `images`, `javascripts`, `stylesheets`) понятно, что для открытого доступа предназначены картинки, клиентская интерактивность и стили. Выберем в интернете 5 картинок, назовем их: `vinni.jpg`, `pig.jpg`, `rabbit.jpg`, `iaia.jpg`, `sova.jpg` и положим в папку `public/images`. Эти картинки будут доступны в браузере.

В IDE откроем файл `routes/index.js`. Под обработкой маршрута «/»

```
/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

пишем свой код сходной структуры:

```
/* Страница Винни Пуха */
router.get('/vinni', function(req, res, next) {
  res.send("<h1>Страница Винни Пуха</h1>")
});
```

В консоле выполним команды:

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

В браузере откроем станицу /vinni (полный путь http://localhost:3000/vinni). Страница должна отобразить текст «Страница Винни Пуха».

Продедаем это еще 4 раза для четырех персонажей и получим в файле routes/index.js:

```
/* Страница Винни Пуха */
router.get('/vinni', function(req, res, next) {
  res.send("<h1>Страница Винни Пуха</h1>")
});

/* Страница Пятачка */
router.get('/pig', function(req, res, next) {
  res.send("<h1>Страница Пятачка</h1>")
});

/* Страница Кролика */
router.get('/rabbit', function(req, res, next) {
  res.send("<h1>Страница Кролика</h1>")
});

/* Страница Иа */
router.get('/iaia', function(req, res, next) {
  res.send("<h1>Страница Иа</h1>")
});

/* Страница Совы */
router.get('/sova', function(req, res, next) {
  res.send("<h1>Страница Совы</h1>")
});
```

И в браузере адреса /vinni, /pig, /rabbit, /iaia, /sova отобразят тексты «Страница Винни Пуха», «Страница Пятачка», «Страница Кролика», «Страница Иа», «Страница Совы» соответственно.

4.3. СОЗДАНИЕ ШАБЛОНА

Следующим шагом создадим шаблон для 5-ти страниц. В IDE откроем папку views —это папка шаблонов. Создадим в этой папке файл hero.ejs и скопируем в этот файл содержимое views/index.ejs.

```
<!DOCTYPE html>
<html>
```

```

<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
</body>
</html>

```

Напомним, что в `routes/index.js` обработка адреса «/» выглядит следующим образом:

```

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

```

Это значит, что для ответа сервер использует шаблон `index` и передает в этот шаблон переменную `title` со значением «Express».

Преобразуем шаблон `views/hero.ejs`, так чтобы он принимал три переменные `title`, `picture` и `desc`.

```

<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  
  <h4>Несколько слов о нашем герое: <%=desc%></h4>
</body>
</html>

```

Вернемся в файл `routes/index.js` и преобразуем код

```

/* Страница Винни Пуха */
router.get('/vinni', function(req, res, next) {
  res.send("<h1>Страница Винни Пуха</h1>")
});

```

по аналогии с

```

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

```

получим

```

/* Страница Винни Пуха */
router.get('/vinni', function(req, res, next) {
  res.render('hero', {
    title: "Винни Пух",
    picture: "images/vinni.jpg",
    desc: "Плюшевый медвежонок, игрушка"
  });
});

```

Проверим полученный результат. В консоле:

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

В браузере по адресу /vinni (полный путь <http://localhost:3000/vinni>) должен отобразиться шаблон, заполненный данными о Винни Пухе.

Преобразуем аналогичным образом обработку адресов: /pig, /rabbit, /iaia, /sova. Получим:

```

/* Страница Винни Пуха */
router.get('/vinni', function(req, res, next) {
  res.render('hero', {
    title: "Винни Пух",
    picture: "images/vinni.jpg",
    desc: "Плюшевый медвежонок, игрушка"
  });
});

/* Страница Пятачка */
router.get('/pig', function(req, res, next) {
  res.render('hero', {
    title: "Пятачок",
    picture: "images/pig.jpg",
    desc: "Крошечный поросенок, которого зовут Пятачок, и
который живет в большом-пребольшом доме"
  });
});

```

```

/* Страница Кролика */
router.get('/rabbit', function(req, res, next) {
  res.render('hero', {
    title: "Кролик",
    picture: "images/rabbit.jpg",
    desc: "Кролик, всегда безумно рад видеть Винни-Пуха"
  });
});

/* Страница Иа */
router.get('/iaia', function(req, res, next) {
  res.render('hero', {
    title: "Иа Иа",
    picture: "images/iaia.jpg",
    desc: "Грустный ослик, который однажды потерял хвост"
  });
});

/* Страница Совы */
router.get('/sova', function(req, res, next) {
  res.render('hero', {
    title: "Сова",
    picture: "images/sova.jpg",
    desc: "Если кто-нибудь что-нибудь о чем-нибудь знает, то
это, конечно, Сова"
  });
});

```

Проверяем:

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

Адреса /vinni, /pig, /rabbit, /iaia, /sova отображают шаблон views/hero.js с данными из обработчиков routes/index.js.

4.4. УСТАНОВКА И ПОДКЛЮЧЕНИЕ ШАБЛОНИЗАТОРА EJS-LOCALS

К стандартам разработки web-приложения относится наличие на каждой странице шапки сайта, подвала сайта, боковых панелей. Технически, задача решается созданием шаблона с разметкой для повторяющихся блоков и регионов для уникального контента.

Шаблонизатор `ejs` этого не предусматривает. Поэтому установим и подключим шаблонизатор, который справится с этой задачей.

```
$ npm install ejs-locals --save
```

В файле `app.js` подключаем шаблонизатор:

```
// view engine setup
app.engine('ejs', require('ejs-locals'));
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
```

Создадим структуру каждой станицы. Для этого в папке `views` создадим папку `layout`. В папке `layout`, создадим файл `page.ejs`. скопируем содержимое `views/index.ejs` и сделаем изменения.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <%= body %>
  </body>
</html>
<body>
```

Неэкранированный вывод `<%= body %>` обеспечивает вывод уникального контента страницы.

В шаблонах `views/index.ejs`, `views/hero.ejs` оставим только содержимое `<body></body>` шаблонов и в `views/index.ejs`, `views/error.ejs`, `views/hero.ejs` добавим первой строкой каждого файла подключение `layout/page.ejs` разметки.

```
<% layout('/layout/page.ejs') %>
```

После изменений файлы `views/index.ejs`, `views/error.ejs`, `views/hero.ejs` примут вид:

Листинг файла `views/index.ejs`

```
<% layout('/layout/page.ejs') %>
```

```
<h1><%= title %></h1>
<p>Welcome to <%= title %></p>
```

Листинг файла views/error.ejs

```
<% layout('/layout/page.ejs') %>
```

```
<h1><%= message %></h1>
<h2><%= error.status %></h2>
<pre><%= error.stack %></pre>
```

Листинг файла views/hero.ejs

```
<% layout('/layout/page.ejs') %>
```

```
<h1><%= title %></h1>

<h4>Несколько слов о нашем герое: <%=desc%></h4>
```

Проверяем:

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

Главный результат проверки – поведение клиентской части приложения должно работать без изменений. Это значит, что шаблонизатор подключился и заменят body в <%= body %> из page.ejs содержимым шаблонов views/index.ejs, views/error.ejs, views/hero.ejs.

5. НАВИГАЦИЯ

Стандартное приложение предполагает возможность перехода с одной страницы на другую (навигация).

Для разработки навигации воспользуемся Bootstrap, Bootstrap – CSS, HTML и JavaScript фреймворк (библиотека). Фреймворк это программный продукт упрощающий решение типовых задач и формирования архитектуры приложения. В отношении Bootstrap нет

единого мнения, считать его фреймверком или библиотекой. Фреймверк, как правило диктует архитектуру приложения, в то время как Bootstrap это просто упрощения решения типовых задач. В нашем случае Bootstrap будет применен для решения типовой задачи – реализации навигации приложения.

5.1. УСТАНОВКА И ПОДКЛЮЧЕНИЕ BOOTSTRAP

Для установки Bootstrap воспользуемся bower (менеджер пакетов клиентского JavaScript). Одним менеджером пакетов мы уже пользуемся – npm (управление пакетами Node.js), bower управляет пакетами клиентского JavaScript (в нашем случае bootstrap и jquery). Выполним в консоле:

```
$ npm install bower
$ bower install bootstrap query
```

Откроем IDE и убедимся, что в проекте появилась новая папка bower_components.

В `<head></head>` тег шаблона `views/layout/page.ejs` добавляем подключение bootstrap:

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
  <link href="/bootstrap/dist/css/bootstrap.css"
rel="stylesheet" type="text/css">
  <script src="/jquery/dist/jquery.js"></script>
  <script src="/bootstrap/dist/js/bootstrap.js"></script>
</head>
<body>
  <%- body %>
</body>
</html>
```

и в файле `app.js` объявляем файлы папки `bower_components` доступными из браузера, то есть публичными.

```
app.use(express.static(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'bower_components')));
```

Bootstrap установлен и готов к использованию.

5.2. НАВИГАЦИОННОЕ МЕНЮ

Готовые bootstrap решения хорошо представлены в справочных ресурсах:

https://www.w3schools.com/bootstrap/bootstrap_badges_labels.asp,

https://www.tutorialspoint.com/bootstrap/bootstrap_transition_plugin.htm.

Выберем навигационное меню :

```
<nav role="navigation" class="navbar navbar-default">
  <!-- Brand and toggle get grouped for better mobile display -->
  <div class="navbar-header">
    <button type="button" data-target="#navbarCollapse" data-
toggle="collapse" class="navbar-toggle">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a href="/" class="navbar-brand">Все Все Все ...</a>
  </div>
  <!-- Collection of nav links, forms, and other content for
  toggling -->
  <div id="navbarCollapse" class="collapse navbar-collapse">
    <ul class="nav navbar-nav">
      <li><a href="#">Винни Пух</a></li>
      <li><a href="#">Пятачок</a></li>
      <li><a href="#">Иа Иа</a></li>
      <li><a href="#">Кролик</a></li>
      <li><a href="#">Сова</a></li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li><a href="#">Войти</a></li>
    </ul>
  </div>
</nav>
```

Проверяем:

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

5.3. ПЕРЕХОД МЕЖДУ СТРАНИЦАМИ

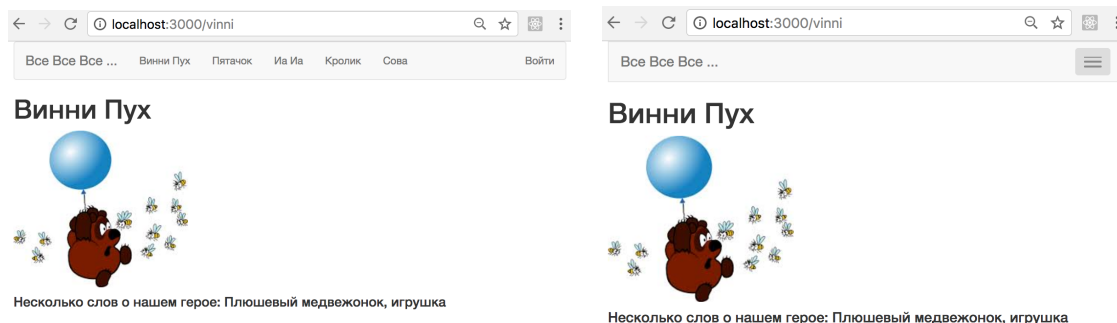
Последним шагом разработки навигации добавим адреса страниц в атрибут href тега `<a>`.

```
<li><a href="/vinni">Винни Пух</a></li>
<li><a href="/pig">Пятачок</a></li>
<li><a href="/iaia">Иа Иа</a></li>
<li><a href="/rabit">Кролик</a></li>
<li><a href="/sova">Сова</a></li>
```

Проверяем:

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

В браузере проверяем, что меню позволяет просматривать созданные страницы. Уменьшая размер окна браузера можно увидеть bootstrap responsive стиль, меню сворачивается для маленьких размеров экрана.



6. МОДУЛЬ MONGODB, ПОДКЛЮЧЕНИЕ БАЗЫ ДАННЫХ

6.1. ПОДКЛЮЧЕНИЕ БАЗЫ ДАННЫХ

Откроем консоль и выполним команды.

команда	объяснение
<code>pwd</code>	проверка где находимся, убеждаемся, что действительно в папке <code>all</code>

<code>cd /полный/путь/к/папке/all</code>	если находимся не в папке all, используя команду <code>cd</code> , переходим в папку all
<code>npm install mongodb --save</code>	устанавливаем модуль <code>mongodb</code> , <code>--save</code> для сохранения имени модуля в <code>dependencies</code> файла <code>package.json</code>

Переходим в IDE и в корне проекта создаем файл `createDB.js`.

Листинг файла `createDB.js`

```
var MongoClient = require("mongodb").MongoClient

MongoClient.connect("mongodb://localhost:27017/all",
function(err,db){
    if(err) throw err
    var collection = db.collection("heroes")
    collection.insertOne({name:"Винни Пух"},function(err,result){
        db.close()
    })
})
```

Модуль `mongodb` устанавливает соединение с базой данных `all` MongoDB. База данных с таким именем пока не существует. В `callback` функции, функции которая запускается после завершения работы вызова функции `connect`, во второй переменной становится доступной база данных `all` (пока еще не созданная). Команда `collection` позволяет по имени `heroes` получить доступ к несуществующей коллекции `heroes` несуществующей базы данных `all`. Командой `insertOne` добавляем документ `{ name: "Винни Пух"}`. Именно в этот момент создается база данных `all`, коллекция `heroes` и собственно документ. В `callback` функции команды `insertOne` закрываем соединение с базой данных командой `db.close()`.

Вернемся в консоль и выполним файл `createDB.js`:

```
$ node createDB.js
```

Откроем в консоле новую вкладку и запустим MongoDB консоль.

mongo	запуск консоли
show dbs	проверяем, что база данных all создана
use all	переходим в базу данных all
db.getName()	проверяем, что находимся в базе данных all
db.getCollectionNames()	убеждаемся, что список коллекций состоит из одной коллекции heroes
db.heroes.find()	проверяем документы коллекции heroes
db.dropDatabase()	удаляем базу данных all

```
> use all
switched to db all
> db.getName()
all
> db.getCollectionNames()
[ "heroes" ]
> db.heroes.find()
{ "_id" : ObjectId("59989d05580ff905995cb144"), "name" : "Винни Пух"
}
```

6.2. СОЗДАНИЕ МОДУЛЯ ДАННЫХ

В IDE подготовим файл с данными для базы данных в формате модуля Node.js. Создадим в корне файл `data.js`:

```
var data = [{
  title: 'Пятачок',
  nick: 'pig',
  avatar: '/images/pig.jpg',
  desc: 'Лучший друг Винни-Пуха, крошечный поросенок ...'
},
```

```

{
  title: 'Иа Иа',
  nick: 'iaia',
  avatar: '/images/iaia.jpg',
  desc: 'Старый серый ослик ...'
},
{
  title: 'Винни Пух',
  nick: 'vinni',
  avatar: '/images/vinni.jpg',
  desc: 'Плюшевый медвежонок, игрушка ... '
},
{
  title: 'Сова',
  nick: 'sova',
  avatar: '/images/sova.jpg',
  desc: 'Сова жила в великолепном замке ...'
},
{
  title: 'Кролик',
  nick: 'rabbit',
  avatar: '/images/rabbit.jpg',
  desc: Кролик, всегда безумно рад видеть Винни-Пуха ...'
}
];

```

```
module.exports.data = data;
```

Созданный в рамках проекта (кастомный) модуль подключается по-прежнему с модулями Node.js способом. Подключим модуль `data.js` в файле `createDB.js`.

```

var MongoClient = require("mongodb").MongoClient
var data = require("./data.js").data

```

В этом месте разработки, используя вывод в консоле, следует убедиться что подготовленные данные доступны в файле `createDB.js`.

6.3. ДАННЫЕ ПРИЛОЖЕНИЯ

После подключения модуля `data.js` в файле `createDB.js`, изменим команду `insertOne` на `insertMany` и передадим `json data` в команду `insertMany`:

```
var MongoClient = require("mongodb").MongoClient
var data = require("./data.js").data

MongoClient.connect("mongodb://localhost:27017/all",
function(err,db){
    if(err) throw err
    db.dropDatabase()
    var collection = db.collection("heroes")
    collection.insertMany(data,function(err,result){
        db.close()
    })
})
```

Вернемся в консоль и выполним команду `$ node createDB.js`. В консоле `mongo` проверим базу данных `all`.

```
> db.heroes.find()
{ "_id" : ObjectId("5998a8c556aca805e3861c49"), "title" :
"Пятачок", "nick" : "pig", "avatar" : "/images/pig.jpg", "desc" :
"Лучший друг Винни-Пуха, крошечный поросенок ..." }
...
```

Т.о. данные приложения записаны в базу данных.

7. МОДУЛЬ ASYNC

7.1. УСТАНОВКА ASYNC

Модуль `async` относится к вспомогательным модулям (`utility module`) и предназначен для работы с асинхронностью JavaScript (<http://github.com/caolan/async/blob/v1.5.2/README.md>). Изначально модуль был разработан для управления запросами к базе данных Node.js, однако он оказался востребованным в браузере для

организации ајах запросов клиента к серверу. Модуль `async` можно установить как используя менеджер `npm` так и `bower`.

команда	объяснение
<code>npm install async</code>	установка менеджером пакетов <code>npm</code> (<code>--save</code> для добавления в <code>dependencies</code> файла <code>package.json</code>)
<code>bower install async</code>	установка менеджером пакетов <code>bower</code>

Модуль `async` предоставляет около 20 функций как общего и вспомогательного назначения так и шаблоны для управления асинхронными запросами JavaScript.

7.1. ФУНКЦИЯ EACH

Синтаксис функции `each`:

```
each(arr, iterator, [callback])
```

Функция `each` имеет два обязательных аргумента `arr`, `iter` и один опциональный `callback`, это аргумент, который можно не использовать при вызове функции. Квадратные скобки `[]` обозначают необязательность аргумента.

синтаксис	объяснение
<code>arr</code>	любой массив элементы которого надо обойти
<code>iterator</code>	функция <code>function(item, callback){</code> <code>}</code> <code>item</code> – элемент массива <code>callback</code> – функцию которую нужно вызвать в теле функции в формате <code>callback(err)</code>
<code>callback</code>	срабатывает, если произошла ошибка, если ошибка не

	произошла, то отработывает после обработки всех элементов массива. Ошибкой считается не пустой аргумент вызова callback функции из тела обработчика (второго аргумента)
--	--

Рассмотрим пример работы функции `async.each()`

Задача: Проверить являются ли элементы массива `[1, "два", 3, 4, 5]` числами.

Решение: Перейдем в IDE и в корне проекта создадим временный файл `file.js`, файл, который удалим после знакомства с возможностями модуля `async`. Первой строчкой файла подключим модуль.

```
var async = require("async")

var arr = [1,"два",3,4,5]

async.each(
  arr,
  function(item,callback){
    if(typeof item === "number"){
      callback()
    } else {
      callback("Нашли не число")
    }
  },
  function(err){
    if(err){
      console.log(err)
    } else {
      console.log("Проверка прошла успешно. Все элементы
массива являются числами")
    }
  }
)
```

В консоле выполним файл `file.js`.

```
$ node file.js
```

Нашли не число

Поменяем строку два на число и снова запустим файл `file.js`.

```
$ node file.js
```

Проверка прошла успешно. Все элементы массива являются числами

Чтобы подчеркнуть, что является, а что не является синтаксисом функции `each` переименуем переменные в решении. При переименовании важно использовать говорящие названия, облегчающие чтение кода.

```
var async = require("async")

var from_one_two_five = [1,"два",3,4,5]

async.each(
  from_one_two_five,
  function(elem,report){
    if(typeof elem === "number"){
      report()
    } else {
      report("Нашли не число")
    }
  },
  function(info){
    if(info){
      console.log(info)
    } else {
      console.log("Проверка прошла успешно. Все элементы массива являются числами")
    }
  }
)
```

Запустим и проверим что код работает.

7.2. ШАБЛОН SERIES

Синтаксис функции `series`:

```
series(tasks, [callback])
```

Функция относится к шаблонам управления асинхронными запросами JavaScript. Задачи `tasks` выполняются по порядку. Вторая задача начинает выполняться только после того как первая задача будет завершена, третья, когда вторая будет завершена и т.д. Если в процессе выполнения задач случится ошибка, то переход к следующей задаче не произойдет и отработает `callback` функция. В случае безошибочного выполнения каждой задачи выполнится `callback` функция.

синтаксис	объяснение
<code>tasks</code>	массив или объект функций <code>function(callback){</code> <code>}</code> В теле функции должна быть вызвана функция <code>callback(err, result)</code>
<code>callback</code>	срабатывает, если произошла ошибка, если ошибка не произошла, то отработывает после обработки всех элементов массива.

Рассмотрим пример работы шаблона `series`:

```
async.series([
  function(callback){
    callback(null, "МАМА")
  },
  function(callback){
    callback(null, "МЫЛА")
  },
  function(callback){
    callback(null, "ПАМУ")
  }
],
function(err, result){
  if(err){
    console.log("Ошибка: "+err)
  } else {
    console.log(result)
  }
})
```

```
    }  
  )
```

Результат выполнения шаблона:

```
$ node file.js  
[ 'МАМА', 'МЫЛА', 'РАМУ' ]
```

Таким образом `result` массив вторых аргументов списка задач. Заменяем `console.log(result)` на `console.log(result.join(" "))` и выполним файл:

```
$ node file.js  
МАМА МЫЛА РАМУ
```

Задачи `tasks` из первого аргумента `series` – анонимные функции. Создадим именованные функции и вызовем их в `series` по названиям.

```
async.series([  
    firstWord,  
    secondWord,  
    thirdWord  
],  
function(err,result){  
    if(err) throw err;  
    console.log(result.join(' '))  
});  
  
function firstWord(callback){  
    callback(null, 'МАМА')  
}  
function secondWord(callback){  
    callback(null, 'МЫЛА')  
}  
function thirdWord(callback){  
    callback(null, 'РАМУ')  
}
```

7.3. ШАБЛОН PARALLEL

Синтаксис функции `parallel`:

```
parallel(tasks, [callback])
```

Функция `parallel` имеет синтаксис, совпадающий с синтаксисом `series` и тоже относится к шаблонам управления асинхронными запросами JavaScript. Отличие `parallel` от `series` в том, что задачи `tasks` начинают выполняются одновременно. Заменим в предыдущем примере `series` на `parallel` и выполним файл `file.js`.

```
$ node file.js
```

```
МАМА МЫЛА ПАМУ
```

7.4. ШАБЛОН WATERFALL

Синтаксис функции `waterfall`:

```
waterfall(tasks, [callback])
```

Шаблон `waterfall` управления асинхронными запросами JavaScript учитывает случай, когда во второй задаче для формирования запроса нужна информация, полученная в результате первого запроса.

```
async.waterfall([
  function(callback){
    callback(null, "МАМА", "МЫЛА", "ПАМУ")
  },
  function(arg1, arg2, arg3, callback){
    callback(null, arg1 + ' ' + arg2 + ' ' + arg3)
  }
],
function(err, result){
  if(err) throw err
  console.log(result)
});
```

Количество аргументов каждой задачи должно совпадать с количеством аргументов `callback` функции в теле предыдущей

функции. В теле первой функции водопада (waterfall) вызов callback имеет вид:

```
callback(null, "МАМА", "МЫЛА", "РАМУ")
```

Значит синтаксис второй задачи:

```
function(arg1, arg2, arg3, callback){  
    ...  
}
```

7.5. УТИЛИТА APPLY

Синтаксис функции apply:

```
apply(function, arguments..)
```

Функция apply – это функция утилита (вспомогательная), которая передает в тело функции function дополнительные аргументы arguments. Рассмотрим, как утилита apply расширяет возможности шаблонов series, parallel.

```
async.series([  
    async.apply(anyWord, "МАМА"),  
    async.apply(anyWord, "МЫЛА"),  
    async.apply(anyWord, "РАМУ")  
],  
function(err, result){  
    if(err) throw err;  
    console.log(result.join(' '))  
});  
  
function anyWord(arg, callback){  
    callback(null, arg)  
}
```

8. МОДУЛЬ MONGOOSE, СОЗДАНИЕ МОДЕЛЕЙ ДАННЫХ

8.1. ORM

Mongoose – это ORM (Object Relational Mapping) для MongoDB под Node.js. Главная задача ORM - быть связующим звеном между кодом программы и базой данных. ORM mongoose берет на себя валидацию данных, CRUD функции и бизнес-логику общения с базой данных.

В консоле установим mongoose

```
$ npm install mongoose --save
```

В IDE очистим файл createdB.js, с официального сайта разработчиков mongoose (<http://mongoosejs.com/>) скопируем пример, запустим и проанализируем код.

```
var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/test')

var Cat = mongoose.model('Cat', { name: String })

var kitty = new Cat({ name: 'Пушок' })
kitty.save(function (err) {
  if (err) {
    console.log(err)
  } else {
    console.log('Мяу')
  }
})
```

```
$ node createdB.js
Мяу
```

В консоле mongo проверим базу данных test

команда	объяснение
mongo	запуск консоли MongoDB
use test	переход в базу данных test
db.getName()	проверяем, что находимся в базе данных test

<code>db.getCollectionNames()</code>	запрос имен коллекций базы
<code>db.cats.find()</code>	запрос документов коллекции cats

```
> use test
switched to db test
> db.getName()
test
> db.getCollectionNames()
[ "cats" ]
> db.cats.find()
{ "_id" : ObjectId("599bc7666d1e8b5a672aa34c"), "name" : "Пушок",
  "__v" : 0 }
```

Проанализируем код.

<code>mongoose.connect('mongodb://localhost/test')</code>	соединение с базой данных test
<code>var Cat = mongoose.model('Cat', { name: String })</code>	объявление модели и полей документов этой модели
<code>var kitty = new Cat({ name: 'Пушок' })</code>	создание экземпляра модели
<code>kitty.save()</code>	сохранение данных в базе

Из имени модели `mongoose` формирует имя коллекции по следующему правилу – имя модели берется с маленькой буквы и преобразуется во множественное число. Имя модели `Cat` – имя коллекции `cats`.

8.2. СОЗДАНИЕ СХЕМЫ

Реализация `mongoose` связи между приложением и базой данных использует два ключевых понятия – модель и схема. Вернемся в IDE и введем в код имплементацию схемы.


```

var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/test');

var schema = mongoose.Schema({ name: String })
var Cat = mongoose.model('Cat', schema);

var kitty = new Cat({ name: 'Пушок' });
kitty.save(function (err) {
  if (err) {
    console.log(err);
  } else {
    console.log('Мяу');
  }
});

```

В консоле проверим результат исполнения файла и убедимся, что создание схемы просто структурировало код. Для иллюстрации возможностей схемы создадим метод схемы.

```

var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/test')

var schema = mongoose.Schema({ name: String })

schema.methods.meow = function(){
  console.log(this.get("name") + " сказал мяу")
}

var Cat = mongoose.model('Cat', schema)

var kitty = new Cat({ name: 'Пушок' })
kitty.save(function (err) {
  kitty.meow()
})

```

В консоле проверим код.

```

$ node createDB.js
Пушок сказал мяу

```

8.3. СОЗДАНИЕ МОДЕЛИ ДАННЫХ ПРИЛОЖЕНИЯ

Перейдем к использованию mongoose для учебного проекта. В корне проекта создадим папку models. В этой папке будем реализовывать

модели приложения. Первую модель назовем Hero, поэтому в папке models создадим файл hero.js.

```
var mongoose = require('mongoose')

var heroSchema = mongoose.Schema({
  title: String,
  nick: {
    type: String,
    unique: true,
    required: true
  },
  avatar: String,
  desc: String,
  created: {
    type: Date,
    default: Date.now
  }
})

module.exports.Hero = mongoose.model("Hero", heroSchema)
```

Проверим работу модели в файле createdDB.js

```
var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/all')
var Hero = require("../models/hero").Hero

var hero = new Hero({
  title: "Пятачок"
})

console.log(hero)
hero.save(function(){
  console.log(arguments)
})
```

```
$ node createdDB.js
{ title: 'Пятачок',
  _id: 599dae619121212173dfaf74,
  created: 2017-08-23T16:33:37.516Z }
{ '0':
  { ValidationError: Hero validation failed: nick: Path `nick` is
    required}
}
```

Сработал валидатор mongoose и не база данных, не коллекция, не документ не создадутся. В консоле mongo проверим базу данных all:

```
switched to db all
> db.getCollectionNames()
[ "heros" ]
> db.heros.find()
>
```

То есть создана база all, коллекция с ожидаемым именем heros, но документ не создан из-за ошибки валидации.

В создании экземпляра модели добавим required поле nick.

```
var hero = new Hero({
  title: "Пятачок",
  nick: "pig"
})
```

Проверяем

команда	объяснение
^C	остановка соединения с базой данных
node createDB.js	запуск файла createDB.js

```
$ node createDB.js
{ title: 'Пятачок',
  nick: 'pig',
  _id: 599db2041e03e5219058961f,
  created: 2017-08-23T16:49:08.938Z }
{ '0': null,
  '1':
  { __v: 0,
    title: 'Пятачок',
    nick: 'pig',
    _id: 599db2041e03e5219058961f,
    created: 2017-08-23T16:49:08.938Z },
  '2': 1 }
```

В консоле mongo

```
> db.heros.find()
{ "_id" : ObjectId("599db2041e03e5219058961f"), "title" :
"Пятачок", "nick" : "pig", "created" : ISODate("2017-08-
23T16:49:08.938Z"), "__v" : 0 }
```

Остановимся на `console.log(arguments)` команде. Слово `arguments` – служебное слово языка и выводим массив аргументов callback функции. Первый аргумент – это информация об ошибке, второй – возвращается сохраненная в базу модель, третий – количество давленных документов. Проверка `arguments` позволяет понять структуру возвращаемых в callback значений. После чего можно поправить синтаксис `save` функции

```
hero.save(function(err, hero, affected){
  console.log(hero.title)
})
```

В консоле `mongo` очистим базу данных:

```
> db.dropDatabase()
{ "dropped" : "all", "ok" : 1 }
```

и снова запустим выполнение файла `createDB.js`

```
^C
$ node createDB.js
Пятачок
```

И снова запустим выполнение файла `createDB.js`

```
^C
$ node createDB.js
MongoError: E11000 duplicate key error collection: all.heros
index: nick_1 dup key: { : "pig" }
```

Проанализируем, что не так. В пояснении ошибки читаем, что проблема у `MongoDB` связана с `nick` полем. Вспомним описание этого поля и сделаем вывод, что при втором запуске кода в базу данных была сделана попытка сохранить документ с полем `nick`

которое уже есть у ранее созданного документа. То есть нарушено условие **unique: true**.

8.4. ЗАПОЛНЕНИЕ БАЗЫ ДАННЫХ С MONGOOSE

Очистим файл `createDB.js`:

```
var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/all')
var Hero = require("./models/hero").Hero

// 1. Очистить базу данных all
// 2. Вставить 5 героев
// 3. Закрыть соединение с базой данных
```

Синтаксис `//` — это синтаксис комментирования. В комментариях принято делать описания, помогающие пониманию кода. Пункты в комментарии — план заполнения базы данных.

Добавим в `createDB.js` код реализующий первый и третий шаг.

```
mongoose.connection.on("open",function(){
  var db = mongoose.connection.db
  db.dropDatabase(function(err){
    if(err) throw err
    console.log("ok")
    mongoose.connection.close()
  })
})
```

После очистки базы данных и перед закрытием соединения сохраним несколько героев.

Воспользуемся одним из шаблонов модуля `async` управления асинхронными запросами JavaScript. Шаблон водопад является избыточным для задачи сохранения героев, поэтому сделаем выбор между шаблонами `series` и `parallel`. Выберем шаблон `parallel`. В шапке файла `createDB.js` добавим подключение модуля `async`.

Следующим шагом, подготовим синтаксис шаблона.

```
mongoose.connection.on("open",function(){
  var db = mongoose.connection.db
```

```

db.dropDatabase(function(err){
    if(err) throw err

    async.parallel([
        function(callback){

        },
        function(callback){

        },
        function(callback){

        }
    ],
    function(err,result){
        mongoose.disconnect()
    })
})
})

```

Реализуем задачи шаблона – сохранение героев.

```

mongoose.connection.on("open",function(){
    var db = mongoose.connection.db
    db.dropDatabase(function(err){
        if(err) throw err

        async.parallel([
            function(callback){
                var pig = new Hero({nick:"pig"})
                pig.save(function(err,pig){
                    callback(err,"pig")
                })
            },
            function(callback){
                var vinni = new Hero({nick:"vinni"})
                vinni.save(function(err,vinni){
                    callback(err,"vinni")
                })
            },
            function(callback){
                var sova = new Hero({nick:"sova"})
                sova.save(function(err,sova){
                    callback(err,"sova")
                })
            }
        ],
        function(err,result){
            if(err){
                console.log(err)
            }
        })
    })
})

```

```

        } else {
            console.log("Успешно созданы герои с никами: "
+result.join(", "))
        }
        mongoose.disconnect()
    })
})
})

```

Перейдем в консоль и выполним файл createDB.js.

```

$ node createDB.js
Успешно созданы герои с никами: pig, vinni, sova

```

8.5. ПРИМЕНЕНИЕ ШАБЛОНА SERIES

Процесс заполнения базы данных разобьем на четыре логических шага и применим шаблон series.

Первый шаг:

```

function open(callback){
    mongoose.connection.on("open",callback)
}

```

Второй шаг:

```

function dropDatabase(callback){
    var db = mongoose.connection.db
    db.dropDatabase(callback)
}

```

Третий шаг:

```

function createHeroes(callback){
    async.parallel([
        function(callback){
            var pig = new Hero({nick:"pig"})
            pig.save(function(err,pig){
                callback(err,"pig")
            })
        },
        function(callback){
            var vinni = new Hero({nick:"vinni"})

```

```

        vinni.save(function(err,vinni){
            callback(err,"vinni")
        })
    },
    function(callback){
        var sova = new Hero({nick:"sova"})
        sova.save(function(err,sova){
            callback(err,"sova")
        })
    }
],
function(err,result){
    callback(err)
})
}

```

Четвертый шаг:

```

function close(callback){
    mongoose.disconnect(callback)
}

```

И серия:

```

async.series([
    open,
    dropDatabase,
    createHeroes,
    close
],
function(err,result){
    if(err) throw err
    console.log("ok")
})

```

Запустим заполнения базы в консоле и проверим в консоле mongo:

```

> use all
switched to db all
> db.getCollectionNames()
[ "heros" ]
> db.heros.find()

```



```
{ "_id" : ObjectId("599deaf76dbb0e239b5a575d"), "nick" : "pig",
"created" : ISODate("2017-08-23T20:52:07.660Z"), "__v" : 0 }
{ "_id" : ObjectId("599deaf76dbb0e239b5a575e"), "nick" : "vinni",
"created" : ISODate("2017-08-23T20:52:07.669Z"), "__v" : 0 }
{ "_id" : ObjectId("599deaf76dbb0e239b5a575f"), "nick" : "sova",
"created" : ISODate("2017-08-23T20:52:07.670Z"), "__v" : 0 }
>
```

8.6. ДАННЫЕ ПРИЛОЖЕНИЯ С MONGOOSE

Адаптируем функцию `createHeroes` для обработки данных из модуля `data.js`. Подключим данные в шапке файла.

```
var data = require('./data.js').data;
```

Преобразуем `createHeroes`:

```
function createHeroes(callback){
    async.each(data, function(heroData, callback){
        var hero = new mongoose.models.Hero(heroData);
        hero.save(callback);
    },
    callback);
};
```

После всех преобразований код файла `createDB.js` примет вид.

```
var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/all')
var Hero = require("./models/hero").Hero
var async = require("async")
var data = require('./data.js').data
```

```
// 1. Очистить базу данных all
// 2. Вставить 5 героев
// 3. Закрыть соединение с базой данных
```

```
async.series([
    open,
    dropDatabase,
    createHeroes,
    close
],
function(err,result){
    if(err) throw err
```

```

        console.log("ok")
    })

function open(callback){
    mongoose.connection.on("open",callback)
}

function dropDatabase(callback){
    var db = mongoose.connection.db
    db.dropDatabase(callback)
}

function createHeroes(callback){
    async.each(data, function(heroData, callback){
        var hero = new mongoose.models.Hero(heroData)
        hero.save(callback)
    },
    callback)
}

function close(callback){
    mongoose.disconnect(callback)
}

```

В консоле mongo поверим, что данные приложения успешно сохранены в базе данных.

9. ИНДЕКСИРОВАНИЕ В БАЗАХ ДАННЫХ

9.1. ЧТО ТАКОЕ ИНДЕКСИРОВАНИЕ

Индекс базы данных— объект базы данных, создаваемый с целью повышения производительности поиска данных. Поиск или анализ по заданному критерию путём последовательного просмотра всех данных документ за документом может занимать много времени. Индекс формируется из значений одного или нескольких полей коллекции и указателей на соответствующие документы коллекции и, таким образом, позволяет искать или анализировать документы, удовлетворяющие критерию поиска или критерию анализа. Ускорение работы с использованием индексов достигается в первую очередь за счёт того, что индекс имеет структуру, оптимизированную под поиск. В нашем случае индексирование поля происходит по параметру `unique: true` схемы модели `Hero` и используется для быстрой проверки уникальности поля `nick`.

9.2. ПРОВЕРКА ИНДЕКСОВ

Для анализа индексации базы данных учебного проекта `all` проведем тестовое изменение функции `createHeroes()`:

```
function createHeroes(callback){
  data = [
    {title: "Пятачок", nick: "pig"},
    {title: "Иа Иа", nick: "pig"},
    {title: "Винни Пух", nick: "pig"},
  ]
  async.each(data, function(heroData, callback){
    var hero = new mongoose.models.Hero(heroData)
    hero.save(callback)
  },
  callback)
}
```

В схеме модели `Hero` (`models/hero.js`) поле `nick` описано следующим образом

```
nick: {
  type: String,
  unique: true,
  required: true
}
```

то есть с требованием `unique: true`. В тестовой модификации все три документа намеренно заданы с одним и тем же значением поля `nick`. Запустим файл `createDB.js` и проанализируем результаты в консоле `mongo`.

```
> db.heros.find()
{ "_id" : ObjectId("59a318a4cca95238d74dc1b4"), "title" :
"Пятачок", "nick" : "pig", "created" : ISODate("2017-08-
27T19:08:20.126Z"), "__v" : 0 }
{ "_id" : ObjectId("59a318a4cca95238d74dc1b5"), "title" : "Иа Иа",
"nick" : "pig", "created" : ISODate("2017-08-27T19:08:20.133Z"),
"__v" : 0 }
{ "_id" : ObjectId("59a318a4cca95238d74dc1b6"), "title" : "Винни
Пух", "nick" : "pig", "created" : ISODate("2017-08-
27T19:08:20.134Z"), "__v" : 0 }
> db.heros.getIndexes()
```

```
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "all.heros"
  }
]
```

Документы с одинаковым уникальным индексом успешно сохранились в коллекции heros. Запрос индексов базы данных командой `db.heros.getIndexes()` возвращает информацию только об `id` поле.

Отсутствие индексации по полю `nick` объясняется следующим образом: Индекс для поля `nick` создается при создании модели и очищается после очистки базы данных.

Индекс создается:

```
var Hero = require("./models/hero").Hero
```

Индекс очищается:

```
db.dropDatabase(callback)
```

9.3. ИНДЕКСИРОВАНИЕ БАЗЫ ALL

Для сохранения индексации базы модель необходимо инициализировать после очистки базы данных. Создадим дополнительную функцию:

```
function requireModels(callback){
  require("./models/hero").Hero

  async.each(Object.keys(mongoose.models), function(modelName){
    mongoose.models[modelName].ensureIndex(callback)
  },
  callback
)
```

Эта функция завершит работу, когда индексы каждой модели будут успешно установлены. Для завершения модификации кода удалим объявление модели в шапке файла и добавим новую функцию в серию после очистки базы данных и добавлением новых данных.

```
async.series([
    open,
    dropDatabase,
    requireModels,
    createHeroes,
    close
],
function(err,result){
    if(err) throw err
    console.log("ok")
})
```

Запустим файл createDB.js и проверим в консоле mongo getIndexes().

```
> db.heros.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "all.heros"
  },
  {
    "v" : 1,
    "unique" : true,
    "key" : {
      "nick" : 1
    },
    "name" : "nick_1",
    "ns" : "all.heros",
    "background" : true
  }
]
```

Сделаем еще один шаг для улучшения кода. Сейчас, в случае ошибки управление кодом сразу передается в callback и закрытие соединения с базой не происходит. Сделаем завершение соединения и в случае

ошибки, и в случае успешного завершения сохранения данных. Для этого перенесем функцию `mongoose.disconnect()` в `callback` и уберем функцию `open`. Окончательный вид файла `createDB.js` примет

вид:

```
var mongoose = require('mongoose')
mongoose.connect('mongodb://localhost/all')
var async = require("async")
var data = require('./data.js').data

async.series([
    open,
    dropDatabase,
    requireModels,
    createHeroes
],
function(err, result){
    mongoose.disconnect()
})

function open(callback){
    mongoose.connection.on("open", callback)
}

function dropDatabase(callback){
    var db = mongoose.connection.db
    db.dropDatabase(callback)
}

function createHeroes(callback){
    async.each(data, function(heroData, callback){
        var hero = new mongoose.models.Hero(heroData)
        hero.save(callback)
    },
    callback)
}

function requireModels(callback){
    require("./models/hero").Hero

    async.each(Object.keys(mongoose.models), function(modelName){
        mongoose.models[modelName].ensureIndexes(callback)
    },
    callback
    )
}
```

Для проверки кода выполним три команды:

команда	объяснение
<code>node createDB.js</code>	выполнение файла <code>createDB.js</code>
<code>db.heros.find()</code>	проверка создания данных
<code>db.heros.getIndexes()</code>	проверка индексации

10. ОТОБРАЖЕНИЕ ДАННЫХ В БРАУЗЕРЕ

10.1. ПОДКЛЮЧЕНИЕ БАЗЫ ДАННЫХ

В шапке главного файла `app.js` приложения подключаем базу данных:

```
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')
mongoose.connect('mongodb://localhost/all')
```

Проверим, что приложение работает.

команда	объяснение
<code>npm start</code>	запуск сервера

Проверим, что адреса `/vinni`, `/pig`, `/rabbit`, `/iaia`, `/sova` отображают данные без изменения и навигация между страницами работает.

10.2. ОБРАБОТКА ПАРАМЕТРА В АДРЕСЕ

В папке `routes`, кроме файла `index.js` есть файл `users.js`. Это заготовка, заложенная в стартовом проекте `express` для обработки адресов, начинающихся со слова `users`. Для отображения героев создадим отдельный файл, аналогичный файлу `users.js`, назовем его `heroes.js` и спроектируем роутер для адресов `heroes/vinni`, `heroes/pig`, `heroes/rabbit`, `heroes/iaia`, `heroes/sova` для отображения

героев. Создадим в папке routes файл heroes.js и скопируем в него содержимое файла users.js.

В главном файле приложения объявим новый маршрутизатор.

```
...
var routes = require('./routes/index');
var users = require('./routes/users');
var heroes = require('./routes/heroes');
...
app.use('/', routes);
app.use('/users', users);
app.use('/heroes', heroes);
...
```

В самом файле изменим тестовый текст ответа браузеру.

```
var express = require('express');
var router = express.Router();

/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('Новый маршрутизатор, для маошрутов, начинающихся с heroes');
});

module.exports = router;
```

Остановим и запустим сервер, проверим работу маршрута /heroes.

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

Для обработки всех адресов heroes/vinni, heroes/pig, heroes/rabbit, heroes/iaia, heroes/sova одним роутером в routes/heroes.js создадим роутер с параметром:

```
/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('Новый маршрутизатор, для маошрутов, начинающихся с heroes');
});

/* Страница героев */
router.get("/:nick", function(req, res, next) {
```



```
});  
res.send(req.params.nick);  
});
```

В теле маршрутизатора доступ к параметру возможен через имя параметра по следующему синтаксису: `req.params.nick`. Проверим что роутер видит параметр.

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

В браузере, по адресам `heroes/vinni`, `heroes/pig`, `heroes/rabbit`, `heroes/iaia`, `heroes/sova` отображаются соответственно `vinni`, `pig`, `rabbit`, `iaia` и `sova`.

10.3. ИЗВЛЕЧЕНИЕ ДАННЫХ ИЗ БАЗЫ

В шапке файла роутера `routes/heroes.js` подключим модель.

```
var express = require('express')  
var router = express.Router()  
var Hero = require("../models/hero").Hero
```

В теле роутера страницы героев сделаем запрос в базе данных.

```
/* Страница героев */  
router.get('/:nick', function(req, res, next) {  
  Hero.findOne({nick:req.params.nick}, function(err,hero){  
    if(err) return next(err)  
    if(!hero) return next(new Error("Нет такого героя в этой  
книжке"))  
    res.render('hero', {  
      title: hero.title,  
      picture: hero.avatar,  
      desc: hero.desc  
    })  
  })  
})  
})
```

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

В браузере проверим страницы: `heroes/vinni`, `heroes/pig`, `heroes/rabbit`, `heroes/iaia`, `heroes/sova`.

10.4. ЧИСТКА КОДА

Страницы героев, которые открываются по адресам `heroes/vinni`, `heroes/pig`, `heroes/rabbit`, `heroes/iaia`, `heroes/sova` заполняются данными из базы данных. Таким образом роутеры для запросов `/vinni`, `/pig`, `/rabbit`, `/iaia`, `/sova` больше не нужны. Перед тем как вычистить код от неактуальных маршрутов, в файле шаблона страницы (`views/layout/page.ejs`) заменим адреса страниц в навигации.

```
<ul class="nav navbar-nav">
  <li><a href="/heroes/vinni">Винни Пух</a></li>
  <li><a href="/heroes/pig">Пятачок</a></li>
  <li><a href="/heroes/iaia">Иа Иа</a></li>
  <li><a href="/heroes/rabbit">Кролик</a></li>
  <li><a href="/heroes/sova">Сова</a></li>
</ul>
```

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

Используя навигацию, в браузере, проверим страницы: `heroes/vinni`, `heroes/pig`, `heroes/rabbit`, `heroes/iaia`, `heroes/sova`.

Теперь, после того как убедились, что все работает, удаляем все что было добавлено в файл `routes/index.js` и удалим маршрутизатор для адреса `/heroes`.

10.4. ЗАПОЛНЕНИЕ МЕНЮ ИЗ БАЗЫ ДАННЫХ

После запроса данных по `nick` параметру добавим запрос для формирования меню. Так как в теле роутера запланировано выполнить два запроса к базе данных выберем один из шаблонов управления запросами модуля `асунс`. Так как порядок выполнения запросов не важен и так как ни один из запросов не нуждается в данных, возвращаемых другим запросом, выберем `parallel`.

Подключим в шапке файла `routes/heroes.js` модуль `асунс`.

```
var express = require('express');
var router = express.Router();
var Hero = require("../models/hero").Hero
var async = require("async")
```

Заготовим шаблон:

```
async.parallel([
  function(callback){
    },
  function(callback){
    }
],
function(err, result){
})
```

Заполним список задач шаблона запросами и воспользуемся структурой переменной result.

```
async.parallel([
  function(callback){
    Hero.findOne({nick:req.params.nick},function(err,hero){
      callback(err,hero)
    })
  },
  function(callback){
    Hero.find({},function(err,heroes){
      callback(err,heroes)
    })
  }
],
function(err,result){
  if(err) return next(err)
  var hero = result[0]
  if(!hero) return next(new Error("Нет такого героя в этой книжке"))
  console.log(hero.avatar)
  res.render('hero', {
    title: hero.title,
    picture: hero.avatar,
    desc: hero.desc
  });
})
```

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

Проверим, что приложение работает без изменений. Оптимизируем код:

```

/* Страница героев */
router.get('/:nick', function(req, res, next) {
  async.parallel([
    function(callback){
      Hero.findOne({nick:req.params.nick}, callback)
    },
    function(callback){
      Hero.find({},callback)
    }
  ],
  function(err,result){
    if(err) return next(err)
    var hero = result[0]
    if(!hero) return next(new Error("Нет такого героя в
этой книжке"))
    console.log(hero.avatar)
    res.render('hero', {
      title: hero.title,
      picture: hero.avatar,
      desc: hero.desc
    });
  })
})

```

и перейдем к логике заполнения меню. Распечатаем результат выполнения второго запроса.

```

function(err,result){
  if(err) return next(err)
  var hero = result[0]
  var heroes = result[1]
  console.log(heroes)
  if(!hero) return next(new Error("Нет такого героя в этой
книжке"))
  res.render('hero', {
    title: hero.title,
    picture: hero.avatar,
    desc: hero.desc
  })
}

```

```
});  
})
```

В консоле убеждаемся, что переменная `heroes` – это все данные коллекции `heros`. Но, для формирования меню получить все данные – это избыточная информация. Для команды `find` воспользуемся вторым аргументом, который управляет полями ответа базы данных.

```
Hero.find({},{_id:0,title:1,nick:1},callback)
```

параметр	объяснение
<code>_id:0</code>	поле <code>_id</code> не включать в ответ базы данных
<code>title:1</code>	поле <code>title</code> включать в ответ базы данных
<code>nick:1</code>	поле <code>nick</code> включать в ответ базы данных

Поля которые не включены во второй аргументом команды `find()` не включаются в ответ базы данных.

Проверим запрос.

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

В консоле получим:

```
[ { title: 'Пятачок', nick: 'pig' },  
  { title: 'Иа Иа', nick: 'iaia' },  
  { title: 'Винни Пух', nick: 'vinni' },  
  { title: 'Сова', nick: 'sova' },  
  { title: 'Кролик', nick: 'rabbit' } ]
```

Полученных данных достаточно для формирования меню. Передадим информацию для меню в шаблон `hero`. После всех изменений файл `routes/heroes.js` примет вид:

```

var express = require('express')
var router = express.Router()
var Hero = require("../models/hero").Hero
var async = require("async")

/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('Новый маршрутизатор, для маршрутов, начинающихся с heroes')
});

/* Страница героев */
router.get('/:nick', function(req, res, next) {
  async.parallel([
    function(callback){
      Hero.findOne({nick:req.params.nick}, callback)
    },
    function(callback){
      Hero.find({}, {_id:0,title:1,nick:1},callback)
    }
  ],
  function(err,result){
    if(err) return next(err)
    var hero = result[0]
    var heroes = result[1] || []
    if(!hero) return next(new Error("Нет такого героя в этой книжке"))
    res.render('hero', {
      title: hero.title,
      picture: hero.avatar,
      desc: hero.desc,
      menu: heroes
    });
  })
})

module.exports = router

```

Переменная меню, передается в шаблон hero. Меню сайта расположено в шаблоне page.js (views/layout/page.js). В этом шаблоне переменная меню тоже будет доступна. Заполним меню данными из базы.

Исходный код:

```

<ul class="nav navbar-nav">
  <li><a href="/heroes/vinni">Винни Пух</a></li>

```

```

<li><a href="/heroes/pig">Пятачок</a></li>
<li><a href="/heroes/iaia">Иа Иа</a></li>
<li><a href="/heroes/rabbit">Кролик</a></li>
<li><a href="/heroes/sova">Сова</a></li>
</ul>

```

Преобразованный код:

```

<% menu.forEach(function(hero){ %>
<li><a href="/heroes/<%=hero.nick%>"><%=hero.title%></a></li>
<% }) %>

```

Так как меню отображается на странице /, преобразуем routes/index.js файл:

```

var express = require('express')
var router = express.Router()
var Hero = require("../models/hero").Hero

/* GET home page. */
router.get('/', function(req, res, next) {
  Hero.find({}, {_id:0,title:1,nick:1},function(err,menu){
    res.render('index', {
      title: 'Express',
      menu: menu
    });
  })
});

module.exports = router;

```

Из файла views/error.ejs удалим строку подключения layout/page.ejs разметки.

```

<% layout('/layout/page.ejs') %>

```

Так как в шаблоне page.js используется переменная menu, которая не передается в шаблон views/error.ejs.

11. COOKIE И SESSION

11.1. УСТАНОВКА МОДУЛЯ EXPRESS-SESSION И НАСТРОЙКА COOKIE

Session это хранилище данных между HTTP запросами. Session создает sid – session id и помещает sid в Cookie. Cookie это текстовая информация, которую сервер передает браузеру, браузер хранит и возвращает на сервер, для доступа к данным сессии. Самой важной частью информации cookie является sid – session id (уникальный идентификатор сессии).

Установим модуль express-session, Node.js модуль который поддерживает сохранение информации в сессии, уникальный идентификатор которой хранится в cookie.

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

Настроим работу сессии. В главном файле приложения app.js подключим модуль.

```
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')
mongoose.connect("mongodb://localhost/all")
var session = require("express-session")
```

Настройка работы сессии происходит через опции, которые устанавливаются в зависимости от задач, которые решает проект. В нашем учебном проекте сделаем простейшие настройки. Настройку сессии сделаем перед объявлением роутеров, чтобы сессия существовала в момент обработки запросов.

```
app.use(session({
  secret: "VinniIsHero",
  cookie: {maxAge: 60*1000}
}))
```

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

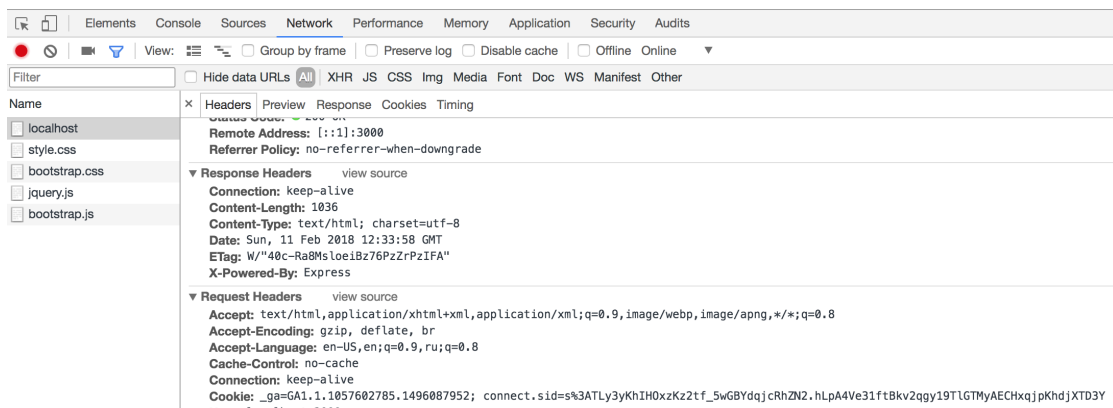

Опция `secret` является обязательной, опция `maxAge:60*1000` определяет время жизни `session` в миллисекундах ($60*1000 = 1$ минута).

Запустим `server`.

команда	объяснение
<code>npm start</code>	запуск сервера

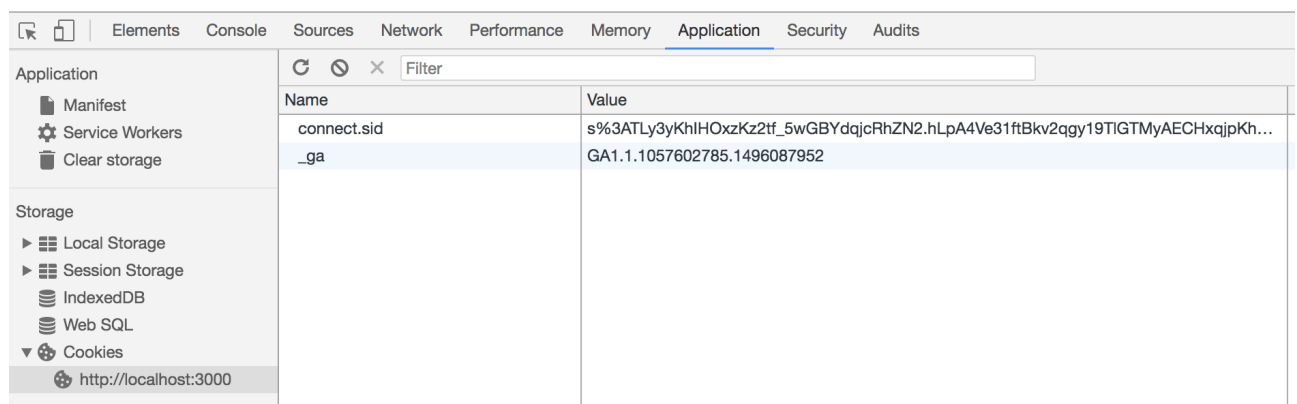
Перейдем в браузер `http://localhost:3000/` и найдем куки с `sid` (session id).

Правой кнопкой мыши нажмем на любое место страницы в браузере и выберем инспектировать элемент.



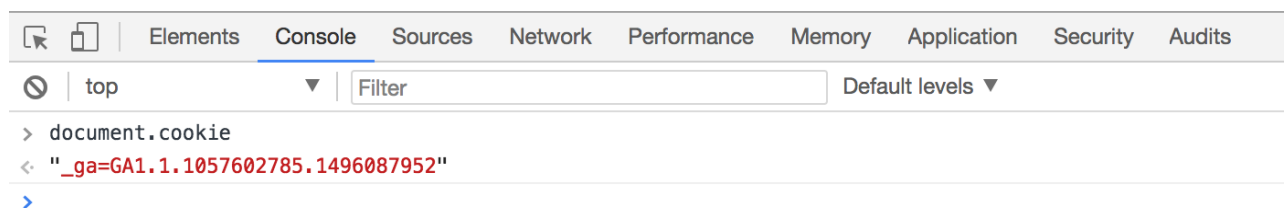
Открываем вкладку `Network` => `localhost` => `Request Headers` => `Cookie` => `connect.sid`

Следующее место для проверки куки:



Вкладка `Application` => `Storage` => `Cookies` => `http://localhost:3000` => `connect.sid`.

Если опция `httpOnly` принимает значение `false` в настройках `session`, то `cookie` доступны `JavaScript`.



11.2. КОМАНДА ЗАПИСИ В СООКІЕ

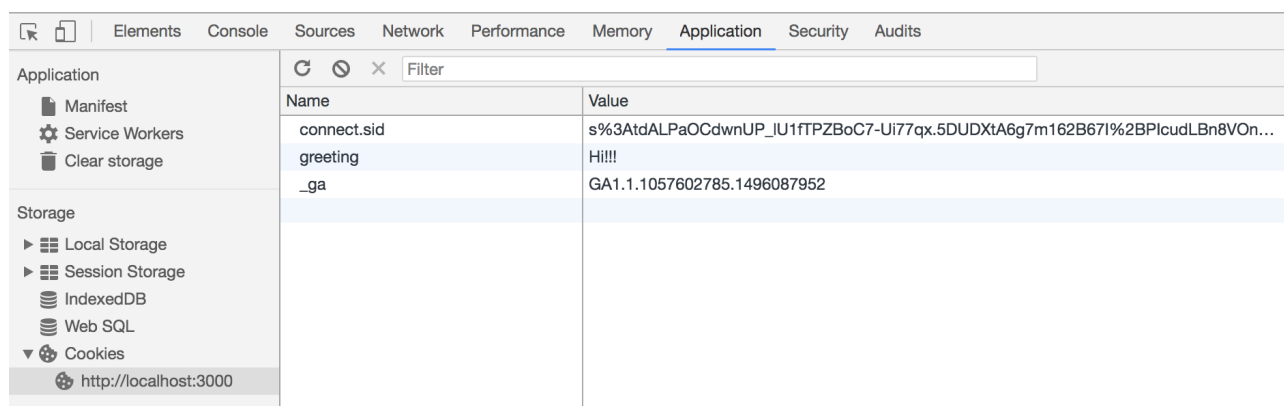
В файле `routes/index.js`, в обработчике запроса главной странице добавим данные в куки.

```
/* GET home page. */
router.get('/', function(req, res, next) {
  res.cookie('greeting', 'Hi!!!!').render('index', { title:
'Express', menu:menu });
});
```

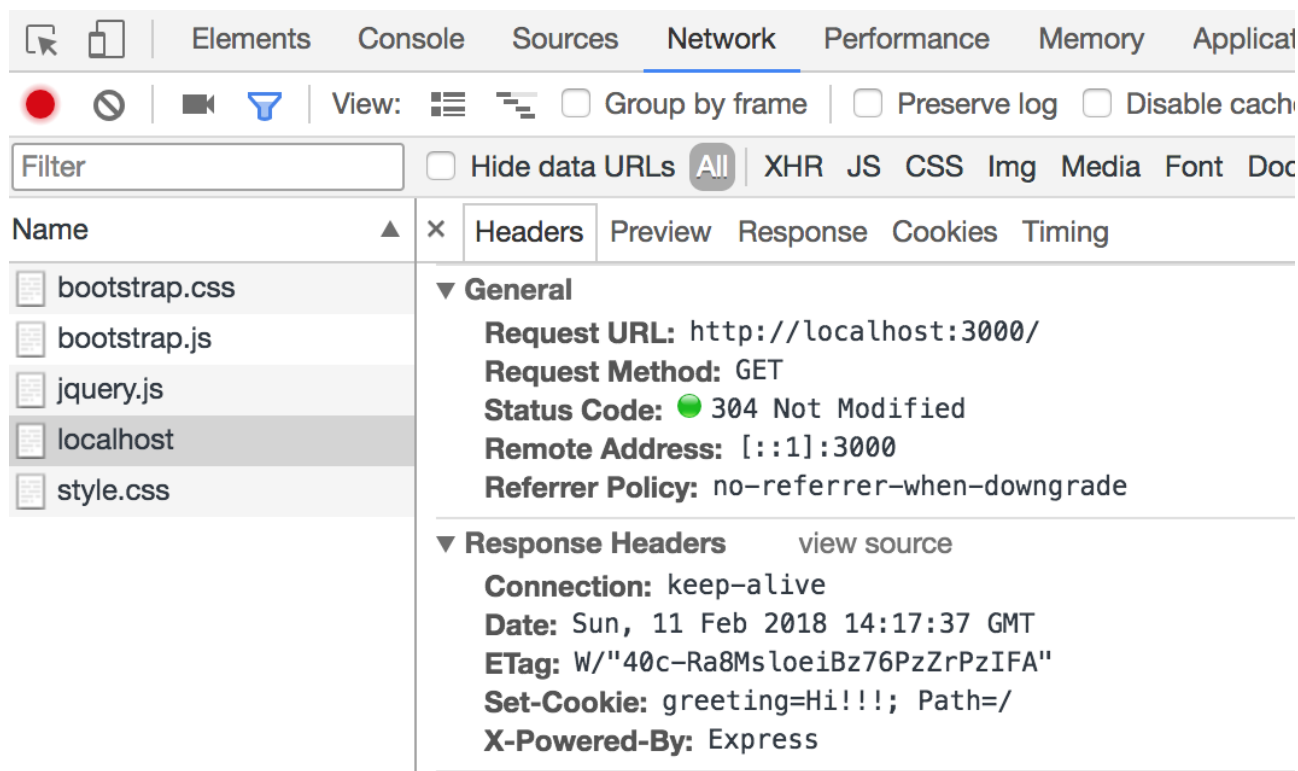
Остановим сервер, запустим сервер и проверим куки в браузере.

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

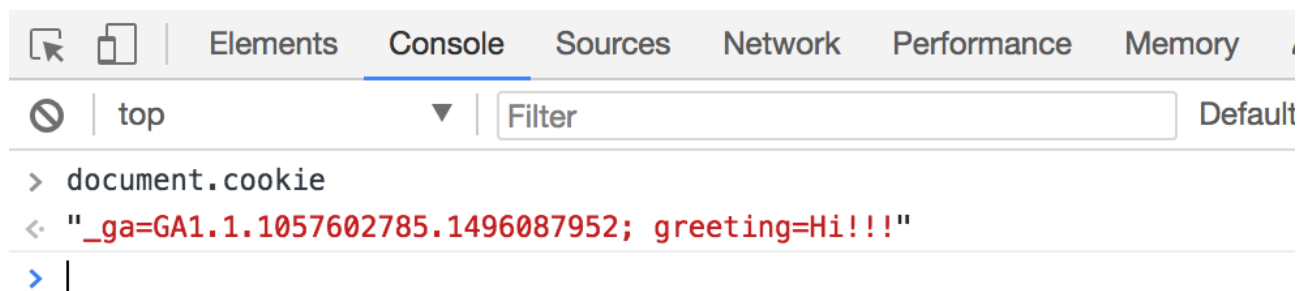
Во вкладке `Application`.



Во вкладке `Network`.



В JavaScript консоле.



11.3. СОХРАНЕНИЕ SESSION В MONGODB

Установим модуль `connect-mongo`, обеспечивающий создание коллекции `sessions` и сохранение документов сессии.

```
$ npm install connect-mongo --save
```

В главном файле приложения подключим модуль `connect-mongo` и добавим настройки сессии, для сохранения данных `session` в коллекции `sessions`.

```

var MongoStore = require('connect-mongo')(session);
app.use(session({
  secret: "VinniIsHero",
  cookie: {maxAge: 60*1000},
  store: new MongoStore({ mongooseConnection:
mongoose.connection})
})))

```

Сделаем запись в session. Для этого сделаем изменения в обработчике запроса главной страницы в файле routes/index.js.

```

/* GET home page. */
router.get('/', function(req, res, next) {
  req.session.greeting = "Hi!!!"
  res.render('index', { title: 'Express', menu: menu });
});

```

Остановим сервер, запустим сервер и проверим коллекцию sessions в Mongo Shell консоли.

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

```

> use all
switched to db all

> db.getCollectionNames()
[ "heros", "sessions" ]

> db.sessions.find()
{ "_id" : "tdALPa0CdownUP_lU1fTPZBoC7-Ui77qx", "session" :
{"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":
true,"path":"/"}, "greeting":"Hi!!!"}, "expires" :
ISODate("2018-02-25T13:29:30.922Z")}

```

11.4. СОЗДАНИЕ СЧЕТЧИКА ПОСЕЩЕНИЯ СТРАНИЦ САЙТА

В главном файле приложения app.js добавим в session переменную counter и логику счетчика посещения страниц. Код следует добавить после создания сессии, но до объявления роутеров.

```

var MongoStore = require('connect-mongo')(session);
app.use(session({
  secret: "VinniIsHero",
  cookie:{maxAge:60*1000},
  store: new MongoStore({ mongooseConnection:
mongoose.connection})
}))
app.use(function(req,res,next){
  req.session.counter = req.session.counter +1 || 1
})

```

Остановим сервер, запустим сервер, перейдем в браузер и походим по страницам, чтобы увидеть в Mongo Shell изменение значения counter.

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

В Mongo Shell:

```
> db.sessions.find()
```

```

{ "_id" : "tdALPa0CdwnUP_lU1fTPZBoC7-Ui77qx", "session" :
{"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":
true,"path":"/"}, "counter":8}, "expires" : ISODate("2018-
02-25T14:35:40.896Z") }
>

```

Заключительный шаг функциональности – передать counter в шаблон и отобразить counter в шаблоне. Для этого снова сделаем изменения в обработчике запроса главной страницы в файле routes/index.js.

```

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express', menu:menu,
counter:req.session.counter });
});

```

Сделаем вывод счетчика в шаблоне главной страницы сайта. Файл views/index.ejs:

```
<% layout('/layout/page.ejs') %>
```

```
<h1><%= title %></h1>
```

```
<p>Welcome to <%= title %></p>
<p>Счетчик: <%= counter %></p>
```

Остановим сервер, запустим сервер, перейдем в браузер, походим по страницам, чтобы увидеть на главной странице изменение значения счетчика.

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

Через минуту, счетчик сбросит свое значение в 1. Это объясняется тем что значение параметра `session maxAge` (время жизни сессии) установлено 1 минута.

11.5. ГЛОБАЛЬНАЯ ПЕРЕМЕННАЯ ДЛЯ НАВИГАЦИИ

В заключение главы рассмотрим встроенную в `express` возможность создания глобальных переменных для шаблонов, то есть таких переменных, которые можно использовать в любом шаблоне, не заботясь о передаче значения этой переменной в шаблон. Выбор переменных, доступных в любом шаблоне определяется логикой приложения. Глобальными стоит делать переменные, которые встречаются на каждой странице. Создадим глобальную переменную, которая будет содержать пункты навигационного меню.

В главном файле приложения `app.js`, перед роутерами создадим `middleware` (посредника).

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

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

Обозначим глобальную переменную для массива ссылок навигационного меню через `nav`. Функционал для массива ссылок навигационного меню возьмем из роутера `index.js`.

```
res.locals.nav = []
```

```

Hero.find(null, {_id:0, title:1, nick:1}, function(err, result){
  if(err) throw err
  res.locals.nav = result
  next()
})

```

Синтаксис объявления глобальной переменной шаблонов:

```
res.locals.nav
```

В результате middleware для создания глобальной переменной пример вид:

```

app.use(function(req, res, next){
  res.locals.nav = []

  Hero.find(null, {_id:0, title:1, nick:1}, function(err, result){
    if(err) throw err
    res.locals.nav = result
    next()
  })
})

```

Посредник подготавливает массив ссылок навигационного меню используя запрос для модели Hero. Чтобы модель была доступна в файле app.js, в шапке файла модуль модели Hero надо подключить.

```
var Hero = require("../models/Hero").Hero
```

Перейдем к использованию переменной nav прямо в шаблоне страницы: views/layout/page.ejs.

```

<ul class="nav navbar-nav">
  <% if(typeof nav == 'object' && nav){
    nav.forEach(function(item){
      %>
      <li>
        <a href="/hero/<%= item.nick %>"><%= item.title %></a>
      </li>
      <%
    })}
  %>
</ul>

```

Уберем переменную меню из кода проекта:

1. Формирование меню menu в шапке роутера routes/index.js.
2. Передача переменной menu в шаблон главной страницы.
3. Передача переменной меню в шаблон страницы героев.
4. Удаление меню из шаблона страницы views/layout/page.ejs.

Проверим, что приложение работает с новой глобальной переменной nav.

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

Код создания глобальной переменной nav в middleware расположен в главном файле приложения app.js. Располагать логику в этом файле не являемся примером хорошей практики программирования. Главный файл приложения должен легко читаться и не должен содержать вычисления как таковые. Проведем оптимизацию кода middleware создания глобальной переменной nav:

1. В корне проекта создадим папку middleware.
2. В папке middleware создадим файл createMenu.js.
3. В начало файла перенесем подключение.
4. Создадим модуль, возвращающий функцию middleware.
5. Подключим модуль вместо middleware функции.

В результате листинг файла createMenu.js примет вид:

```
var Hero = require("../models/Hero").Hero

module.exports = function(req, res, next){
  res.locals.nav = []

  Hero.find(null, {_id:0, title:1, nick:1}, function(err, result){
    if(err) throw err
    res.locals.nav = result
    next()
  })
}
```

Подключение модуля в app.js:


```
app.use(require("./middleware/createMenu.js"))
```

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

Убедимся, что код работает без ошибок.

команда	объяснение
^C	остановка сервера
npm start	запуск сервера

12. АУТЕНТИФИКАЦИЯ

12.1. СОЗДАНИЕ СТРАНИЦЫ РЕГИСТРАЦИИ

К стандартной функциональности web-проекта относится аутентификация пользователя. Под аутентификацией понимают вход пользователя в закрытую систему. Кроме термина аутентификация существует термин авторизация, как правило под авторизацией понимают определение прав пользователя. Добавим в учебный проект функциональность аутентификации.

Добавим к проекту страницу с регистрационной формой. Для этого:

1. Добавим роутер новой страницы.
2. Создадим шаблон с формой.
3. Добавим в навигационной меню ссылку на страницу регистрации.

В файле `routes/index.js` добавим новый роутер.

```
/* GET login/registration page. */  
router.get('/logreg', function(req, res, next) {  
  res.render('logreg');  
});
```

По адресу `/logreg` браузер откроет страницу шаблона `logreg`. Создадим шаблон `logreg.ejs` в папке `views`. В начале страницы

регистрационной формы подключим шаблон страницы с навигационным меню.

```
<% layout('/layout/page.ejs') %>
```

Для верстки формы воспользуемся готовым Bootstrap решением:

```
<div class="container">
  <h1>Регистрация и вход</h1>
  <p>Введите имя пользователя и пароль, если такого пользователя
нет, то он будет создан.</p>
  <form class="form-horizontal">
    <div class="form-group">
      <label class="control-label col-sm-1">Имя</label>
      <div class="col-sm-5">
        <input class="form-control" type="text"
          placeholder="Введите имя"/>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-1">Пароль</label>
      <div class="col-sm-5">
        <input class="form-control" type="password"
          placeholder="Введите пароль"/>
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-1 col-sm-5">
        <input class="btn btn-primary" type="submit"
          value="Войти"/>
      </div>
    </div>
  </form>
</div>
```

Отметим, что кроме классов Bootstrap отвечающих за стиль формы в верстке используются классы Bootstrap Grid. Bootstrap Grid это разбивка страницы браузера на 12 колонок. При этом Bootstrap Grid использует 4 класса xs, sm, md, lg.

класс	ширина экрана для применения стиля класса
xs	ширина экрана меньше 768px
sm	ширина экрана меньше 992px больше или равна 768px

md	ширина экрана меньше 1200px больше или равна 992px
lg	больше или равна 1200px

Например, класс `col-sm-1` для тега `<label>` определяет стиль `<label>`, в зависимости от ширины экрана следующим образом: При ширине экрана больше или равной 768px тег `<label>` занимает 1 колонку. При ширине экрана меньше 768px количество колонок определяется классом `xs`, если класс `xs` для `<label>` не задан, то `<label>` занимает 12 колонок. При ширине экрана больше или равна 992px количество колонок, которые занимает `<label>` определяется классом `md`. Если класс `md` для `<label>` не задан, количество колонок остается равным настройке предыдущего размера экрана.

Для доступа к странице регистрации осталось добавить пункт навигационного меню. В шаблоне страницы `views/layout/page.ejs` добавим ссылку «Войти».

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="/logreg">Войти</a></li>
</ul>
```

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

Проверим, что в навигационном меню появилась новая ссылка «Войти», после нажатия на которую открывается форма с полями регистрации.

12.2. СОЗДАНИЕ МОДЕЛИ USER

Для управления пользователем приложения создадим `mongoose` модель. Назовем модель `User`, `mongoose` создаст в базе данных коллекцию `users`. Синтаксис создания `mongoose` модели мы уже обсуждали в Главе 8 при создании модели хранения данных `Hero`.

Для хранения моделей в проекте в Главе 8 была создана специальная папка `models`. Структурирование проекта является частью хорошего

стиля программирования. Название папки подчеркивает назначение папки. Папка `models` самым названием определяет содержимое. Для создания модели пользователя добавим в папку `models` файл `User.js`. В начале файла модели подключим модуль `mongoose`.

```
var mongoose = require("mongoose")
```

Следующим шагом добавим схему и `exports`.

```
var schemaUser = mongoose.Schema({
  username:{
    type: String,
    unique: true,
    required: true
  },
  hashedPassword:{
    type: String,
    required: true
  },
  salt:{
    type: String,
    required: true
  },
  created:{
    type:Date,
    default: Date.now
  }
})
```

```
module.exports.User = mongoose.model("User", schemaUser)
```

Проанализируем поля схемы. Поле `username` это имя пользователя, которое будет введено пользователем в первое поле формы на странице регистрации. Второе поле регистрационной формы – пароль. Но среди полей схемы нет поля `password`. Отсутствие поля `password` в коллекции базы данных является хорошим стилем программирования. Не принято хранить в базах данных пароли пользователей, а принято хранить хеш отпечатки паролей, усиленные для безопасности случайной добавкой, которая обычно носит название соль. Поэтому в схеме заданы поля `hashedPassword` и `salt`. Поле `created` фиксирует дату создания пользователя.

Для работы с паролем добавим в схему виртуальное поле `password`. Виртуальное поле – это поле которое не сохраняется в базе данных,

но используется для вычисления других полей, в нашем случае hashedPassword и salt.

```
schemaUser.virtual("password").set(function(password){
    this._purePassword = password
    this.salt = Math.random() + ""
    this.hashedPassword = this.encryptPassword(password)
}).get(function(){
    return this._purePassword
})

schemaUser.methods.encryptPassword = function(password){
    return crypto.createHmac('sha1',
this.salt).update(password).digest('hex')
}
```

Создание hashedPassword вынесено в отдельный метод схемы. Для поддержки криптографических методов установим и подключим метод crypto в шапке файла User.js.

```
var mongoose = require("mongoose")
var crypto = require("crypto")
```

Проверим работу модели. Для проверки работы модели, в корне проекта, создадим временный файл checkUser.js. После тестирования модели файл checkUser.js можно удалить.

Листинг файла checkUser.js

```
var mongoose = require("mongoose")
mongoose.connect("mongodb://localhost/all")
var User = require("../models/User.js").User

var first_user = new User({
    username: "Vasya",
    password: "qwerty"
})

first_user.save(function(err,user){
    if(err) throw err
    console.log(user)
})
```

В файле подключаем mongoose модуль, подключаемся к базе данных, подключаем модель User. Создание пользователя происходит в два шага.

Шаг 1: Создание экземпляра модели со значениями имени пользователя и пароля.

Шаг 2: Сохранение пользователя.

Выполним файл checkUser.js и проверим в базе данных результат.

команда	объяснение
node checkUser.js	выполнение файла checkUser.js
mongo	запуск Mongo Shell
use all	переход в базу данных all
db.getCollectionNames()	список коллекций в базе
db.users.find()	запрос документов коллекции users

```
> db.users.find()
{ "_id" : ObjectId("5ab0f29216d62c2641318993"), "username" :
"Vasya", "hashedPassword" :
"b9a2e548b8d8e8d12d4dc6be159892574f19989f", "salt" :
"0.44138812070676803", "created" : ISODate("2018-03-
20T11:37:54.671Z"), "__v" : 0 }
>
```

12.3. ПОДГОТОВКА ДАННЫХ ДЛЯ ПЕРЕДАЧИ НА СЕРВЕР

Перейдем к этапу передачи данных со станицы формы в роутер. Отметим атрибуты формы, которые обеспечивают передачу данных.

```
<form class="form-horizontal" action="/logreg" method="post">
  <div class="form-group">
    <label class="control-label col-sm-1">Имя</label>
    <div class="col-sm-5">
      <input class="form-control" type="text"
        placeholder="Введите имя" name="username"/>
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-1">Пароль</label>
```

```

        <div class="col-sm-5">
            <input class="form-control" type="password"
                placeholder="Введите пароль" name="password"/>
        </div>
    </div>
    <div class="form-group">
        <div class="col-sm-offset-1 col-sm-5">
            <input class="btn btn-primary" type="submit"
                value="Войти"/>
        </div>
    </div>
</form>

```

атрибут	объяснение
action="/logreg"	адрес отправки данных
method="post"	метод http запроса
name="username"	имя параметра поля ввода имени
name="password"	имя параметра поля ввода пароля
type="submit"	триггер для отправки данных

В файле роутера `routes/index.js` добавим обработчик для метода `post`:

```

/* POST login/registration page. */
router.post('/logreg', function(req, res, next) {

});

```

Для обеспечения получения данных в роутере-обработчике в `Node.js` предусмотрен модуль `body-parser`. В стартовом проекте `express` этот модуль уже подключен в главном файле приложения `app.js`. После чего, получение значений параметров методом `post` примет вид:

```

var username = req.body.username
var password = req.body.password

```

12.3. ЛОГИКА ПОЛЬЗОВАТЕЛЯ

Логика пользователя включает прежде всего проверку существования пользователя с именем username в базе данных. В шапке роутера подключим модель User.

```
var Hero = require("../models/Hero").Hero
var User = require("../models/User").User
```

В обработчике метода post подготовим логическую схему первого шага анализа username и password.

```
User.findOne({username:username},function(err,user){
  if(err) return next(err)
  if(user){
    res.send("<h1>Пользователь найден</h1>");
  } else {
    res.send("<h1>Пользователь НЕ найден</h1>")
  }
}
```

Для случая «Пользователь НЕ найден» реализуем создание нового пользователя.

```
var user = new User({username:username,password:password})
user.save(function(err,user){
  if(err) return next(err)
  req.session.user = user._id
  res.redirect('/')
})
```

В случае успешного создания нового пользователя в переменную сессии user сохраним _id нового пользователя. Заключительная строчка логики — res.redirect('/') — это переход на главную страницу после успешного создания нового пользователя.

В случае существования пользователя с именем username подготовим логическую схему второго шага:

```
if(user.checkPassword(password)){
  res.send("<h1>Пароль верный</h1>")
} else {
  res.send("<h1>Пароль НЕ верный</h1>")
}
```


Функцию `checkPassword` создадим как метод схемы модели `User`. Для этого в файл модели `models/User.js` добавим функцию `checkPassword`.

```
schemaUser.methods.checkPassword = function(password){  
    return this.encryptPassword(password) === this.hashPassword  
}
```

В результате логика пользователя примет вид:

```
router.post('/logreg', function(req, res, next) {  
    var username = req.body.username  
    var password = req.body.password  
    User.findOne({username:username}, function(err, user){  
        if(err) return next(err)  
        if(user){  
            if(user.checkPassword(password)){  
                req.session.user = user._id  
                res.redirect('/')  
            } else {  
                res.render('logreg')  
            }  
        } else {  
            var user = new User({username:username,password:password})  
            user.save(function(err, user){  
                if(err) return next(err)  
                req.session.user = user._id  
                res.redirect('/')  
            })  
        }  
    })  
});
```

В случае правильного пароля в сессию сохраняем `_id` пользователя и переходим на главную страницу. В случае неправильного пароля возвращаем пользователя на страницу регистрации. Если пароль не верный, то пользователь не просто должен быть возвращен на страницу регистрации, а получить сообщение о неправильном пароле, то есть получить сообщение валидации. Логика сообщения о неправильном пароле обязательно будет добавлена позже.

12.4. ГЛОБАЛЬНАЯ ПЕРЕМЕННАЯ USER

Создадим глобальную переменную `user`, которая будет доступна во всех шаблонах, по аналогии с созданием глобальной переменной `nav` для навигационного меню.

В папке `middleware` создадим файл `createUser.js`.

Листинг файла `createUser.js`

```
var User = require("../models/User").User

module.exports = function(req, res, next) {
  res.locals.user = null

  User.findById(req.session.user, function (err, user) {
    if (err)
      return next(err)
    res.locals.user = user;
    next();
  })
}
```

Подключение модели, поиск пользователя по `_id` пользователя, сохраненного в сессии, создание глобальной переменной `user` и `exports middleware` функции.

Подключим созданный `middleware` в главном файле приложения `app.js`.

```
app.use(require("./middleware/createMenu.js"))
app.use(require("./middleware/createUser.js"))

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

Переменную `user` можно использовать в любом шаблоне.

Воспользуемся переменной `user` на главной странице сайта. Откроем файл `views/index.ejs` и добавим JavaScript код.

```
<% if(user) {%>
  // Случай залогиненного пользователя
<% }else{ %>
  // Случай не залогиненного пользователя
<% } %>
```

Для случая залогиненного пользователя добавим приветствие пользователя по имени. Для случая незалогиненного пользователя добавим рекомендацию залогиниться для просмотра страниц сайта.

```
<% if(user) {%>
<h1>Привет <%= user.username %></h1>
<% } else { %>
<h1>Зарегистрируйтесь, чтобы видеть все страницы сайта</h1>
<% } %>
```

12.5. ОБРАБОТКА ОШИБКИ АУТЕНТИФИКАЦИИ

На странице регистрационной формы добавим `<div>` для отображения ошибки аутентификации:

```
...
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-8">
    <input class="btn btn-primary" type="submit"
      placeholder="" />
  </div>
</div>
<div class="alert alert-danger">Пароль не верный</div>
</form>
```

Bootstrap классы `alert` и `alert-danger` обеспечивают стиль сообщения об ошибке.

Регистрация и вход

Введите имя пользователя и пароль. Если такого пользователя нет то он будет создан.

Имя	<input type="text" value="name"/>
Пароль	<input type="password" value="....."/>
	<input type="submit" value="Submit"/>

Пароль не верный

Шаблон регистрационной формы вызывается в приложении дважды. Первый раз, когда открываем страницу регистрационной формы при

нажатию на ссылку «Войти» и второй раз, когда возвращаем пользователя на страницу регистрационной формы после ошибки аутентификации. Передадим в шаблон переменную `error`. Для первого вызова шаблона

```
/* GET auth page. */
router.get('/logreg', function(req, res, next) {
  res.render('logreg', {error:null});
});
```

переменной `error` присваиваем значение `null`.
Для второго вызова шаблона

```
res.render('logreg', {error:"Пароль не верный"});
```

переменной `error` присваивается сообщение об ошибке.
Добавим обработку переменной `error` в шаблоне.

```
...
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-8">
    <input class="btn btn-primary" type="submit"
      placeholder="" />
  </div>
</div>
<% if(error) { %>
  <div class="alert alert-danger"><%= error %></div>
<% } %>
</form>
```

12.6. ФУНКЦИОНАЛЬНОСТЬ LOGOUT

Добавим в приложение функциональность `logout`. Для этого в шаблоне навигационного меню добавим отображение ссылки «Выйти» для залогиненного пользователя. Перейдем в файл `views/logout/page.js`.

```
<ul class="nav navbar-nav navbar-right">
  <%if(user){%>
    <li><button class="btn btn-
      link">Выйти</button></li>
  <% } else { %>
    <li><a href="/logreg">Войти</a></li>
```

```
<% } %>
</ul>
```

Для пункта меню «Выйти» в html использован тег <button>. Это связано с тем, что функциональность logout принято реализовывать методом post. Тег <button> заложен в html как элемент формы для формирования logout методом post.

Окружим <button> формой для вызова метода logout методом post.

```
<ul class="nav navbar-nav navbar-right">
  <%if(user){%>
  <form action="/logout" method="post">
    <li><button type="submit" class="btn btn-
      link">Выйти</button></li>
  </form>
  <% } else { %>
    <li><a href="/logreg">Войти</a></li>
  <% } %>
</ul>
```

В файле routes/index.js добавим post роутер для адреса /logout.

```
/* POST logout. */
router.post('/logout', function(req, res, next) {
  req.session.destroy()
  res.redirect('/')
});
```

12.7. ЗАКРЫТИЕ СТРАНИЦ САЙТА ДЛЯ НЕЗАЛОГИНЕННОГО ПОЛЬЗОВАТЕЛЯ

Последним штрихом завершения учебного проекта реализуем функциональность закрытия страниц сайта для незалогиненного пользователя. Для ограничения доступа к определенным страницам сайта воспользуемся технологией middleware. В пособии к возможностям посредника мы обращались дважды. Первый раз для создания глобальной переменной nav и второй раз для создания глобальной переменной user. Третий раз повторим наги создания middleware.

В папку middleware добавим новый файл checkAuth.js.

Листинг файла checkAuth.js

```

module.exports = function(req, res, next){
    if(!req.session.user_id){
        res.redirect("/")
    }
    next()
}

```

Логика реализованная в посреднике очень простая: если есть `_id` (если пользователь залогиненный), то посредник запускает функцию `next()` которая позволяет следовать логике приложения дальше, если `_id` в сессии не существует, то происходит редирект на главную страницу сайта, где для незалогиненного пользователя отображается рекомендация залогиниться, чтобы увидеть все страницы сайта.

Подключим посредник `checkAuth.js` в шапке файла `routes/index.js`.

```

var express = require('express');
var router = express.Router();
var checkAuth = require("../middleware/checkAuth.js")

```

Для ограничения доступа к страницам героев добавим посредник вторым аргументом в роутер.

```

/* Страница Героев. */
router.get('/hero/:nick', checkAuth, function(req, res, next) {
...

```

команда	объяснение
<code>^C</code>	остановка сервера
<code>npm start</code>	запуск сервера

Проверим, что аутентификация и авторизация пользователя работает как было задумано в учебном проекте.

Учебное издание

Сейдаметова З.С., Сейтвелиева С.Н., Адильшаева Э.И.

ПРИКЛАДНАЯ ИНФОРМАТИКА: ИТОГОВАЯ АТТЕСТАЦИЯ СТУДЕНТОВ

Учебное пособие

Ответственный за выпуск: Сейдаметова З.С.

Компьютерная верстка: Сейтвелиева С.Н.

Издательство ООО «ДИАЙПИ»
г. Симферополь, пр. Кирова, 17 тел./факс +7(3652) 248-178, +7(978)776-56-76.
dip@diprint.com.ua, www.diprint.com.ua

Отпечатано в ИП Семенова Е.А.
297567, Республика Крым, Симферопольский р-н.,
с. Залесье, ул. Победы 25, кв. 3.