Сейдаметова З.С., Москалева Ю.П.

ЧАСТЬ II

ПРОГРАММИРОВАНИЕ:   
УЧЕБНЫЙ ПРОЕКТ НА JAVASCRIPT

УЧЕБНОЕ ПОСОБИЕ

Симферополь

«ДИАЙПИ»

2022

**УДК 004.41:378.2**

**ББК: 32.97**

**С 28**

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

**ISBN 978-5-906821-30-0 © Сейдаметова З.С., Москалева Ю.П., 2018**

**СОДЕРЖАНИЕ**

[1. Введение 5](#_Toc511402357)

[2. Сервер на express 6](#_Оглавление000013)

[2.1. Простейший сервер на express 6](#_Toc511402359)

[2.2. Обработка post запросов 10](#_Toc511402360)

[2.3. Стартовый проект express 14](#_Toc511402361)

[2.4. Анализ кода стартового проекта 16](#_Toc511402362)

[2.5. Выполнение кода JavaScript 19](#_Оглавление000014)

[3. MongoDB 20](#_Оглавление000015)

[3.1. Консоль mongo 20](#_Toc511402365)

[3.2. Команда FIND 23](#_Toc511402366)

[3.4. Упражнения для закрепления правил синтаксиса 31](#_Toc511402368)

[4. Маршрутизаторы и шаблоны стартового express проекта 32](#_Оглавление000016)

[4.1. Стартовый проект с шаблонизатором ejs 32](#_Toc511402370)

[4.2. Маршрутизатор 34](#_Оглавление000006)

[4.3. Создание шаблона 36](#_Toc511402372)

[4.4. Установка и подключение шаблонизатора ejs-locals 41](#_Оглавление000017)

[5. Навигация 43](#_Оглавление000018)

[5.1.  подключение Bootstrap 44](#_Оглавление000019)

[5.2. Навигационное меню 46](#_Toc511402376)

[5.3. Переход между страницами 48](#_Toc511402377)

[6. Модуль mongodb, подключение базы данных 49](#_Оглавление000020)

[6.1. Подключение базы данных 49](#_Toc511402379)

[6.2. Создание модуля данных 52](#_Toc511402380)

[6.3.  ПОСЕВ ДанныХ приложения 54](#_Toc511402381)

[7. Модуль mongoose, создание моделей данных 55](#_Оглавление000021)

[7.1. ORM 55](#_Toc511402390)

[7.2. Создание схемы 58](#_Toc511402391)

[7.3. Создание модели данных приложения 59](#_Оглавление000022)

[8. Отображение данных в браузере 63](#_Оглавление000023)

[8.1. Подключение базы данных 63](#_Toc511402401)

[8.2. Обработка параметра в адресе 64](#_Toc511402402)

[8.3. Извлечение данных из базы 66](#_Toc511402403)

[8.4. Чистка кода 67](#_Toc511402404)

[8.5. Заполнение меню из базы данных 68](#_Toc511402405)

[9. Cookie и Session 75](#_Оглавление000024)

[9.1. Установка модуля express-session и настройка cookie 75](#_Toc511402407)

[9.2. Команда записи в cookie 77](#_Toc511402408)

[9.3. Сохранение session в MongoDB 80](#_Toc511402409)

[9.4. Создание счетчика посещения страниц сайта 81](#_Toc511402410)

[9.5. Глобальная переменная для навигации 83](#_Toc511402411)

[10. Аутентификация 87](#_Оглавление000025)

[10.1. Создание страницы регистрации 87](#_Toc511402413)

[10.2. Создание модели User 90](#_Toc511402414)

[10.3. Подготовка данных для передачи на сервер 93](#_Toc511402415)

[10.4. Логика пользователя 95](#_Toc511402416)

[10.5. Глобальная переменная user 97](#_Toc511402417)

[10.6. Обработка ошибки аутентификации 99](#_Toc511402418)

[10.7. Функциональность logout 101](#_Toc511402419)

[10.8. Закрытие страниц сайта для незалогиненного пользователя 102](#_Toc511402420)

# 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](https://developers.google.com/v8/). Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, [npm](https://www.npmjs.com/"), 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 функционала 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

v18.10.0

«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 |
| --- | --- |
| *mkdir folder* | создание папки с именем *folder* |
| *cd folder* | вход в папку *folder* |
| *pwd* | проверка где находимся, убеждаемся, что действительно в папке *folder* |

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

Откроем папку проекта в любой IDE (Integrated Development Environment), то есть в любой среде разработки. Рекомендуемая среда разработки VSCode.

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"**: **"^4.18.2"** }  
}

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 *expess*, инициализируем приложение, прописываем обработку рутового маршрута браузера *"/",* командой *listen()* запускаем Node.js сервер, *3000* – указываем порт доступа из браузера к запушенному 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 инициирует 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"**)  
})

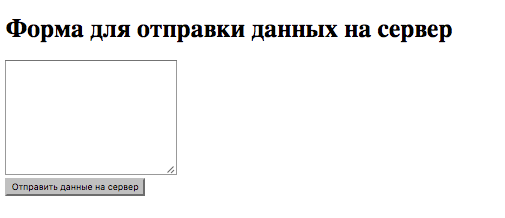
JavaScript функция console.log() делает запись в Сonsole и используется для контроля процесса разработки.

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



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 сосредоточится на базовых основах программирования.

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

$ npm install express-generator –g

$ express my\_express\_app -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, my\_express\_app – 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 jada. В настоящем пособии для разработки учебного проекта выбран шаблонизатор ejs, так как ejs очень похож на HTML и требует минимального времени для изучения.

   create : my\_express\_app

   create : my\_express\_app/package.json

   create : my\_express\_app/app.js

   create : my\_express\_app/public

   create : my\_express\_app/public/javascripts

   create : my\_express\_app/public/images

   create : my\_express\_app/public/stylesheets

   create : my\_express\_app/public/stylesheets/style.css

   create : my\_express\_app/routes

   create : my\_express\_app/routes/index.js

   create : my\_express\_app/routes/users.js

   create : my\_express\_app/views

   create : my\_express\_app/views/index.jade

   create : my\_express\_app/views/layout.jade

   create : my\_express\_app/views/error.jade

   create : my\_express\_app/bin

   create : my\_express\_app/bin/www

install dependencies:

     $ cd my\_express\_app && npm install

   run the app:

     $ DEBUG=my\_express\_app:\* npm start

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

| команда | объяснение |
| --- | --- |
| *cd* my\_express\_app | вход в папку my\_express\_app |
| *pwd* | проверка где находимся, убеждаемся, что действительно в папке my\_express\_app |
| *npm install* | установка модулей из списка  **"dependencies"**: {  **"body-parser"**: **"~1.20.1"**,  **"cookie-parser"**: **"~1.4.6"**,  **"debug"**: **"~4.3.4"**,  **"express"**: **"~4.18.2"**,  **"ejs"**: **"~3.1.8"**,  **"morgan"**: **"~1.10.0"** }  файла *package*.*json* |
| *npm start* | 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 |

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

| подключение: | **var *debug*** = ***require***(**'debug'**)(**'my\_express\_app:server'**) |
| --- | --- |
| выполнение: | ***debug***(**"Проверка работы debug модуля"**) |
| старт приложения: | $ DEBUG=my\_express\_app:\* npm start |
| отладка: | **my\_express\_app:server** проверка работы debug модуля +0ms  **my\_express\_app:server** listening on port 3000 +51ms |

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 отобразиться 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

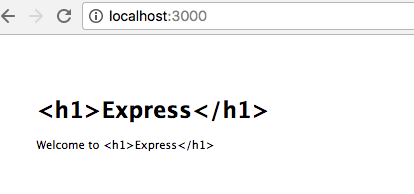
<%- title %> - неэкранированный вывод данных

Добавим в значение переменной title спецсимволы:

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

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

Откроем браузер:



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 Сonsole*.*

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 Сonsole*.*

**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 v6.0.1

Рассмотрим базовые команды

| команда | объяснение |
| --- | --- |
| brew services start mongodb-community | старт базы MongoDB |
| brew services stop mongodb-community | стоп базы MongoDB |
| mongosh | запуск консоли MongoDB |
| show dbs | список баз данных |
| use name | переключение на 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 |

Войдем в консоль MongoDB и выполним команду show dbs

$ 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 морковку |
| 8 | Найти сколько единорогов не любят яблоки и не любят морковку |
| 9 | Найти единорогов, которые любят виноград или морковку |
| 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 выполнятся 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 (vacсinated: 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 three\_cats –e

 express three\_cats -e

   create : three\_cats

   create : three\_cats/package.json

   create : three\_cats/app.js

   create : three\_cats/public

   create : three\_cats/public/images

   create : three\_cats/public/stylesheets

   create : three\_cats/public/stylesheets/style.css

   create : three\_cats/routes

   create : three\_cats/routes/index.js

   create : three\_cats/routes/users.js

   create : three\_cats/views

   create : three\_cats/views/index.ejs

   create : three\_cats/views/error.ejs

   create : three\_cats/bin

   create : three\_cats/bin/www

   create : three\_cats/public/javascripts

   install dependencies:

     $ cd three\_cats && npm install

   run the app:

     $ DEBUG=three\_cats:\* npm start

Это создание стартового клиент-сервер проекта на Node.js, three\_cats – 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 jada.

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

| команда | объяснение |
| --- | --- |
| *cd* three\_cats | вход в папку three\_cats |
| *pwd* | проверка где находимся, убеждаемся, что действительно в папке three\_cats |
| *npm install* | установка модулей из списка  **"dependencies"**: {  **"body-parser"**: **"~1.20.1"**,  **"cookie-parser"**: **"~1.4.6"**,  **"debug"**: **"~4.3.4"**,  **"express"**: **"~4.18.2"**,  **"ejs"**: **"~3.1.8"**,  **"morgan"**: **"~1.10.0"** }  файла *package*.*json* |
| *npm start* | запуск скелета |

## 

Создаем проект в github. Перед тем как сделать push добавим в корень проекта файл .gitignore ( в этом файле перечисляют папки и файлы которым не место в удаленном репозитори).

*Листинг .gitignore*

# Logs

logs

\*.log

# Runtime data

pids

\*.pid

\*.seed

# Directory for instrumented libs generated by jscoverage/JSCover

lib-cov

# Coverage directory used by tools like istanbul

coverage

# Grunt intermediate storage (http://gruntjs.com/creating-plugins#storing-task-files)

.grunt

# node-waf configuration

.lock-wscript

# Compiled binary addons (http://nodejs.org/api/addons.html)

build/Release

# Dependency directory

# https://docs.npmjs.com/cli/shrinkwrap#caveats

node\_modules

bower\_components

# Debug log from npm

npm-debug.log

.DS\_Store

.env

В файлы учтены различные варианты вспомогательных и библиотечных файлов. После добавления файла .gitignore делаем push в удаленный репозиторий

## 4.2. Маршрутизатор

Начинаем работу с пунктом проверкой ветки

% git branch

main

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_4\_2

% git branch

br\_4\_2

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 3 картинки, назовем их: karamelka.jpg, kompot.jpg, korthik.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 Карамельки \*/****router***.get(**'/karamelka'**, **function**(req, res, next) {  
 res.send(**"<h1>Страница Карамельки</h1>"**)  
});

В консоле выполним команды:

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

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

Проделаем это еще 2 раза для 2 персонажей и получим

в файле routes/index.js:

*/\* Страница Карамельки \*/****router***.get(**'/karamelka'**, **function**(req, res, next) {  
 res.send(**"<h1>Страница Карамельки</h1>"**)  
});  
  
*/\* Страница Компота \*/****router***.get(**'/kompot'**, **function**(req, res, next) {  
 res.send(**"<h1>Страница Компота</h1>"**)  
});  
  
*/\* Страница Коржика \*/****router***.get(**'/korthik'**, **function**(req, res, next) {  
 res.send(**"<h1>Страница Коржика</h1>"**)  
});

И в браузере адреса /karamelka, /kompot, /korthik отобразят тексты «Страница Карамельки», «Страница Компота», «Страница Коржика» соответственно.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_4\_2

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_4\_2

В удаленном репозитории появляется ветка br\_4\_2

## 4.3. Создание шаблона

Начинаем работу с пунктом проверкой ветки

% git branch

br\_4\_2

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_4\_3

% git branch

br\_4\_3

Следующим шагом создадим шаблон для 3-х страниц. В IDE откроем папку views – это папка шаблонов. Создадим в этой папке файл cat.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/cat.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**>  
<**img src="**<%= picture %>**" alt=""**/>  
<**h4**>Несколько слов о нашем герое: <%=desc%></**h4**>  
</**body**>  
</**html**>

Вернемся в файл routes/index.js и преобразуем код

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

по аналогии с

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

получим

*/\* Страница Карамельки \*/****router***.get(**'/karamelka'**, **function**(req, res, next) {  
 res.render(**'cat'**, {  
 **title**: **"Карамелька"**,  
 **picture**: **"images/karamelka.jpg"**,  
 **desc**: **"Cамый младший член семьи. Коронная фраза — «Я знаю, что надо делать!». Носит красный бант и красное платье."** });  
});

Проверим полученный результат. В консоле:

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

В браузере по адресу /karamelka (полный путь http://localhost:3000/karamelka) должен отобразится шаблон, заполненный данными о Карамельке.

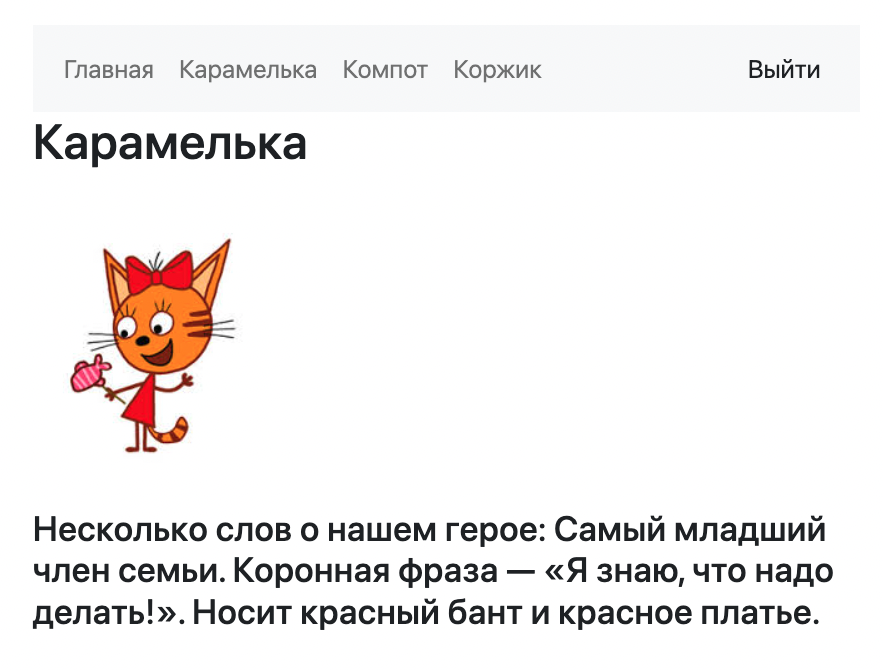
Преобразуем аналогичным образом обработку адресов: /kompot, /korthik. Получим:

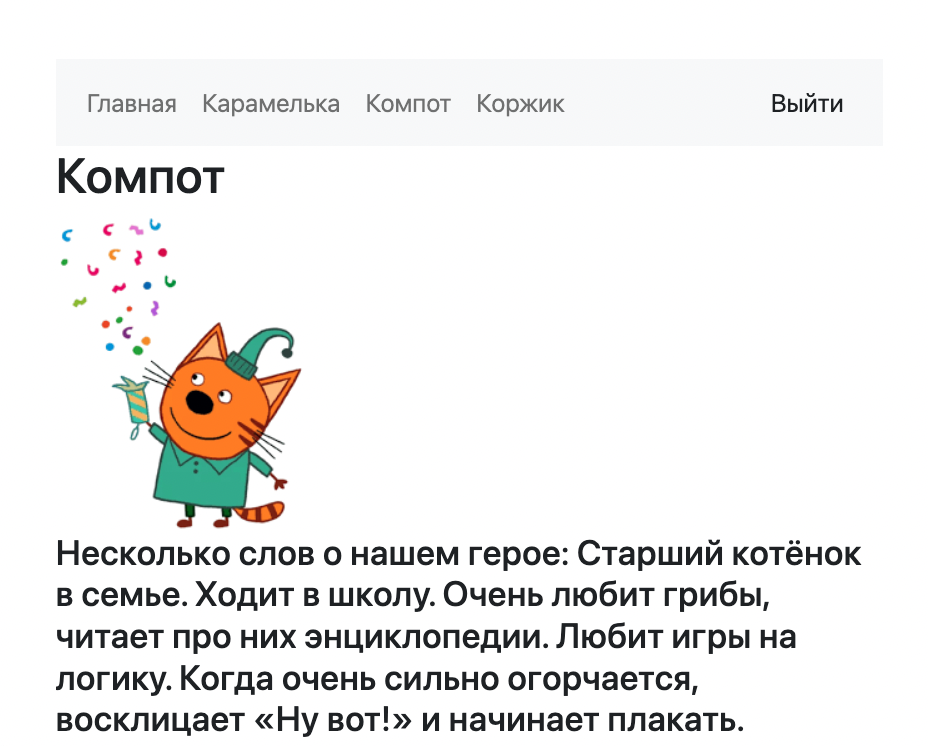
*/\* Страница Карамельки \*/****router***.get(**'/karamelka'**, **function**(req, res, next) {  
 res.render(**'cat'**, {  
 **title**: **"Карамелька"**,  
 **picture**: **"images/karamelka.jpg"**,  
 **desc**: **"Cамый младший член семьи. Коронная фраза — «Я знаю, что надо делать!». Носит красный бант и красное платье."** });  
});  
  
*/\* Страница Компота \*/****router***.get(**'/kompot'**, **function**(req, res, next) {  
 res.render(**'cat'**, {  
 **title**: **"Компот"**,  
 **picture**: **"images/kompot.jpg"**,  
 **desc**: **"Cтарший котёнок в семье. Ходит в школу. Очень любит грибы, читает про них энциклопедии. Любит игры на логику. Когда очень сильно огорчается, восклицает «Ну вот!» и начинает плакать. Обожает вкусно поесть. Носит зелёный костюм и шапку."** });  
});  
  
*/\* Страница Коржика \*/****router***.get(**'/korthik'**, **function**(req, res, next) {  
 res.render(**'cat'**, {  
 **title**: **"Коржик"**,  
 **picture**: **"images/korthik.jpg"**,  
 **desc**: **"Средний котёнок в семье. Очень любит футбол. Любит бегать, прыгать и веселиться. Иногда делает такие вещи, что из них приходится выбираться всей семье. Одет в форму моряка."** });  
});

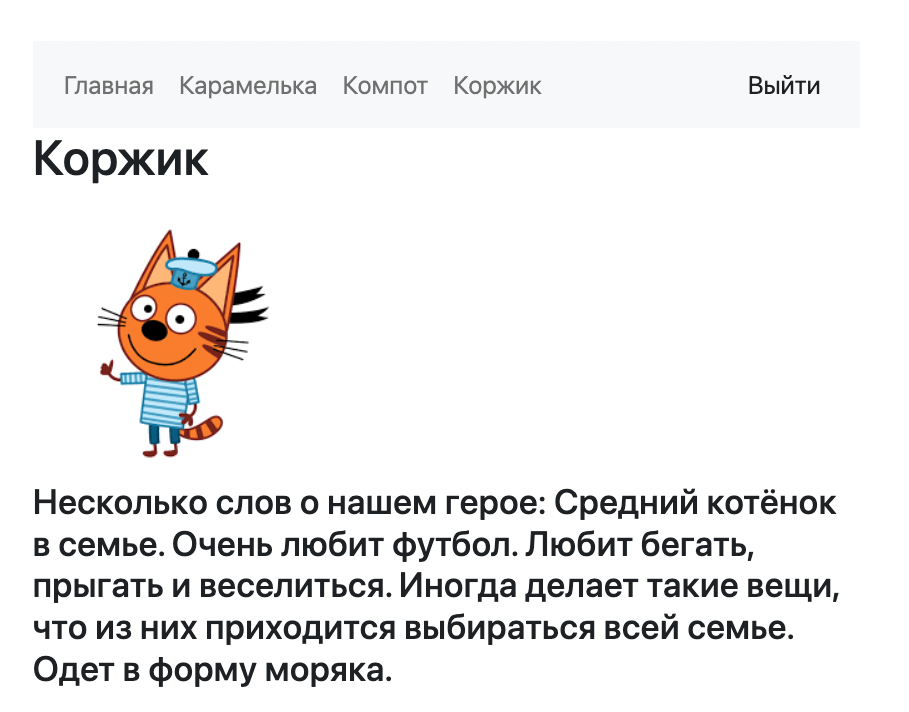
Проверяем:

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

Адреса /karamelka, /kompot, /korthik отображают шаблон views/cat.js c данными из обработчиков routes/index.js.







Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_4\_3

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_4\_3

В удаленном репозитории появляется ветка br\_4\_3

## 4.4. Установка и подключение шаблонизатора ejs-locals

Начинаем работу с пунктом проверкой ветки

% git branch

br\_4\_3

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_4\_4

% git branch

br\_4\_4

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/cat.ejs оставим только содержимое <body></body> шаблонов и в views/index.ejs, views/error.ejs, views/cat.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/cat.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**>

В шаблон views/error.ejs передадим пустую строку как значение переменной **title**: “”.

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

*Листинг файла views/cat.ejs*

<% *layout*(**'/layout/page.ejs'**) %>  
  
<**h1**><%= title %></**h1**>  
<**img src="**<%= picture %>**" alt=""**/>  
<**h4**>Несколько слов о нашем герое: <%=desc%></**h4**>

Проверяем:

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

Главный результат проверки – поведение клиентской части приложение 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/cat.ejs.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_4\_4

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_4\_4

В удаленном репозитории появляется ветка br\_4\_4

# 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.  подключение Bootstrap

Начинаем работу с пунктом проверкой ветки

% git branch

br\_4\_4

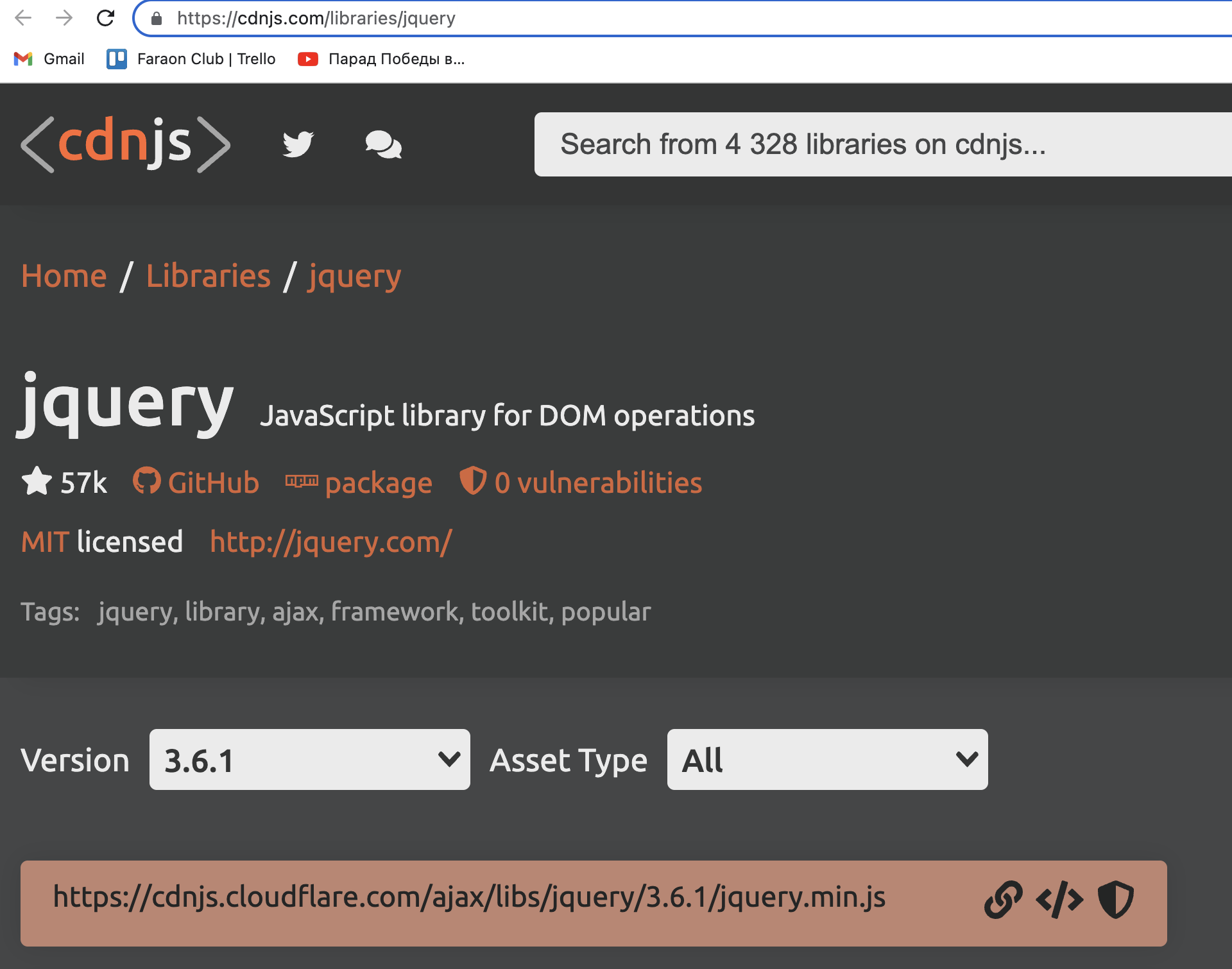
Cоздаем новую ветку и переходим на нее

% git checkout -b br\_5\_1

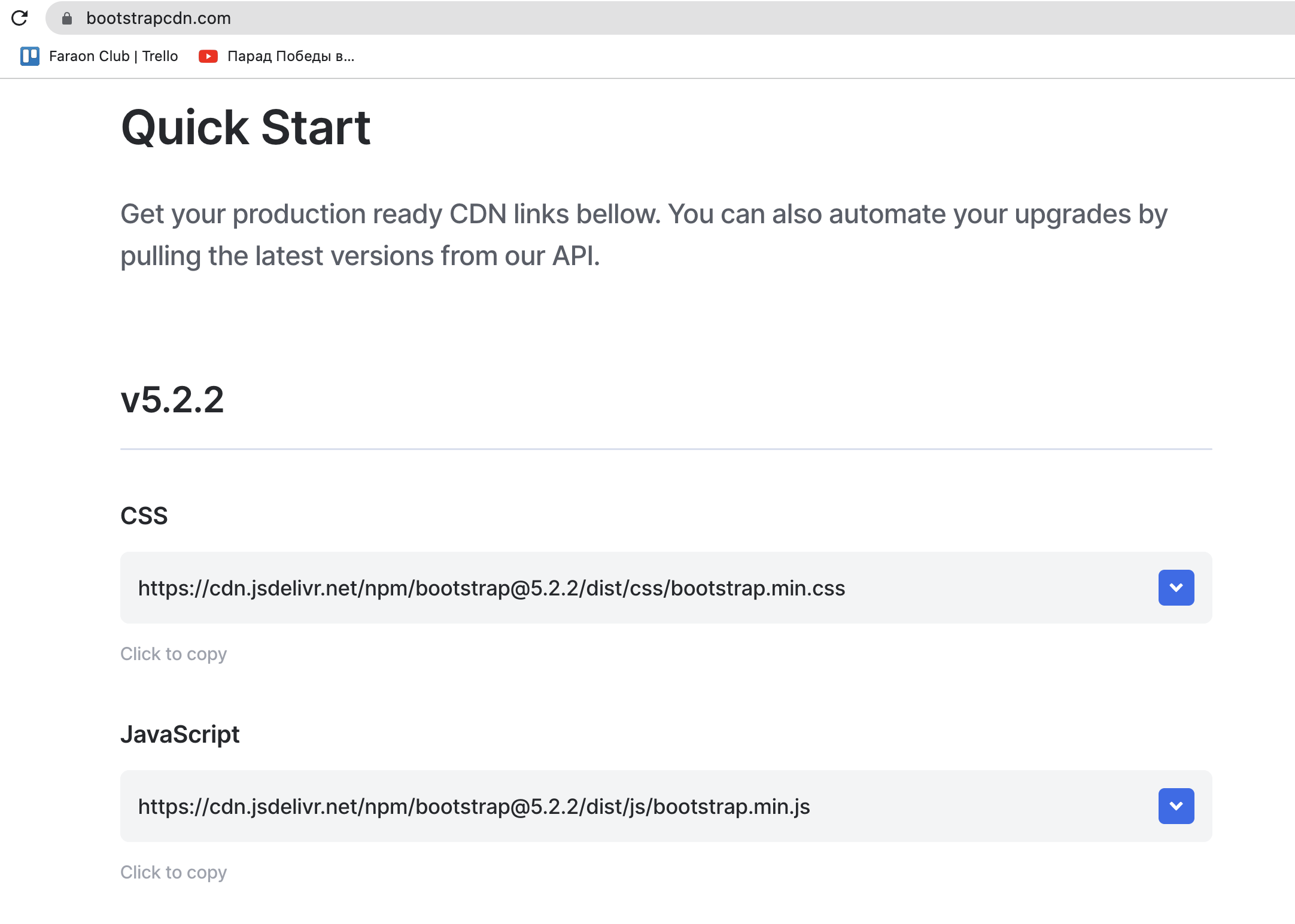
% git branch

br\_5\_1

Для установки Bootstrap воспользуется cdn (content delivery network). Ссылка <https://cdnjs.com/>.



Ссылка для подключения JS библиотеки jQuery.



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="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet" type="text/css"**>   
 <**script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.1/jquery.min.js"**></**script**>  
  <**script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.min.js"**></**script**>  
</**head**>  
<**body**>  
 <%- body %>  
</**body**>  
</**html**>

Bootstrap подключен и готов к использованию.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_5\_1

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_5\_1

В удаленном репозитории появляется ветка br\_5\_1

## 5.2. Навигационное меню

Начинаем работу с пунктом проверкой ветки

% git branch

br\_5\_1

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_5\_2

% git branch

br\_5\_2

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>.

Для выбора навигационного меню:

<https://www.w3schools.com/bootstrap5/bootstrap_navbar.php>

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**>

**<nav class="navbar navbar-expand-sm bg-light navbar-light">**

**<div class="container-fluid">**

**<ul class="navbar-nav">**

**<li class="nav-item">**

**<a class="nav-link" href="/">Главная</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="">Коржик</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="">Карамелька</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="">Компот</a>**

**</li>**

**</ul>**

**<ul class="navbar-nav navbar-right">**

**<li>**

**<a class="nav-link" href="/logreg">Войти</a>**

**</li>**

**</ul>**

**</div>**

**</nav>**

Проверяем:

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_5\_2

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_5\_2

В удаленном репозитории появляется ветка br\_5\_2

## 5.3. Переход между страницами

Начинаем работу с пунктом проверкой ветки

% git branch

br\_5\_2

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_5\_3

% git branch

br\_5\_3

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

**<li class="nav-item">**

**<a class="nav-link" href="/korthik">Коржик</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="/karamelka">Карамелька</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="/kompot">Компот</a>**

**</li>**

Проверяем:

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | 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.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_5\_3

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_5\_3

В удаленном репозитории появляется ветка br\_5\_3

# 6. Модуль mongodb, подключение базы данных

## 6.1. Подключение базы данных

Начинаем работу с пунктом проверкой ветки

% git branch

br\_5\_3

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_6\_1

% git branch

br\_6\_1

Откроем консоль и выполним команды.

| команда | объяснение |
| --- | --- |
| pwd | проверка где находимся, убеждаемся, что действительно в папке *three\_cats* |
| cd /полный/путь/к/папке/all | если находимся не в папке *three\_cats*, используя команду cd, переходим в папку *three\_cats* |
| npm install mongodb --save | устанавливаем модуль mongodb, --save для сохранения имени модуля в dependences файла package.json |

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

*Листинг файла createDB.js*

**var MongoClient = require('mongodb').MongoClient**

**const uri = "mongodb://localhost:27017/"**

**const client = new MongoClient(uri)**

**async function run() {**

**try {**

**await client.connect();**

**var database = client.db("threecats");**

**database.dropDatabase()**

**database = client.db("threecats");**

**const cats = database.collection("cats");**

**const result = await cats.insertOne(**{**name:"Барсик"});**

**console.log(`${result} documents were inserted`);**

**} finally {**

**await client.close();**

**}**

**}**

**run()**

mongodb устанавливает соединение с базой данных threecats MongoDB. База данных с таким именем пока не существует. Команда collection позволяет по имени cats получить доступ к несуществующей коллекции cats несуществующей базы данных threecats. Командой insertOne добавляем документ { name: “Барсик”}. Именно в этот момент создается база данных threecats, коллекция cats и собственно документ. В независимости получилось создать документ или нет, в блоке finally закрываем соединение с базой данных командой client.close().

Вернемся в консоль и выполним файл createDB.js:

$ node createDB.js

Откроем в консоле новую вкладку и запустим MongoDB консоль.

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

> use threecats

switched to db threecats

> db.getName()

threecats

> db.getCollectionNames()

[ "cats" ]

> db.cats.find()

{ "\_id" : ObjectId("59989d05580ff905995cb144"), "name" : "Барсик" }

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_6\_1

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_6\_1

В удаленном репозитории появляется ветка br\_6\_1

## 6.2. Создание модуля данных

Начинаем работу с пунктом проверкой ветки

% git branch

br\_6\_1

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_6\_2

% git branch

br\_6\_2

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

**var data = [{**

**title: 'Карамелька',**

**nick: 'karamelka',**

**avatar: '/images/karamelka.jpeg',**

**desc: 'Cамый младший член семьи. Коронная фраза — «Я знаю, что надо делать!». Носит красный бант и красное платье.'**

**},**

**{**

**title: 'Компот',**

**nick: 'kompot',**

**avatar: '/images/kompont.png',**

**desc: 'Cтарший котёнок в семье. Ходит в школу. Очень любит грибы, читает про них энциклопедии. Любит игры на логику. Когда очень сильно огорчается, восклицает «Ну вот!» и начинает плакать. Обожает вкусно поесть. Носит зелёный костюм и шапку.'**

**},**

**{**

**title: 'Коржик',**

**nick: 'korthik',**

**avatar: '/images/korthik.png',**

**desc: 'Средний котёнок в семье. Очень любит футбол. Любит бегать, прыгать и веселиться. Иногда делает такие вещи, что из них приходится выбираться всей семье. Одет в форму моряка.'**

**}**

**];**

**module.exports.data = data;**

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 убедится что подготовленные данные доступны в файле createDB.js.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_6\_2

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_6\_2

В удаленном репозитории появляется ветка br\_6\_2

## 6.3.  ПОСЕВ ДанныХ приложения

Начинаем работу с пунктом проверкой ветки

% git branch

br\_6\_2

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_6\_3

% git branch

br\_6\_3

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**

**const uri = "mongodb://localhost:27017/"**

**const client = new MongoClient(uri)**

**async function run() {**

**try {**

**await client.connect();**

**var database = client.db("threecats");**

**database.dropDatabase()**

**database = client.db("threecats");**

**const cats = database.collection("cats");**

**const result = await cats.insertMany(data);**

**console.log(`${result.insertedCount} documents were inserted`);**

**} finally {**

**await client.close();**

**}**

**}**

**run()**

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

> db.cats.find()

{ "\_id" : ObjectId("5998a8c556aca805e3861c49"), "title" : "Карамелька", "nick" : "karamelka", "avatar" : "/images/karamelka.jpg", "desc" : "Cамый младший член семьи. Коронная фраза …" }

…

Т.о. данные приложения записаны в базу данных.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_6\_3

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_6\_3

В удаленном репозитории появляется ветка br\_6\_3

# 7. Модуль mongoose, создание моделей данных

## 7.1. ORM

Начинаем работу с пунктом проверкой ветки

% git branch

br\_6\_3

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_7\_1

% git branch

br\_7\_1

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 создадим новый файл для изучения Momgoose mongooseTasks.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 mongooseTasks.js

Мяу

В консоле mongo проверим базу данных test

| команда | обяснение |
| --- | --- |
| mongo | запуск консоли MongoDB |
| use test | переход в базу данных test |
| db.getName() | проверяем, что находимся в базе данных test |
| db.getCollectionNames() | запрос имен коллекций базы |
| db.cats.find() | 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.

| ***mongoose***.*connect*(**'mongodb://localhost/test'**) | cоединение с базой данных test |
| --- | --- |
| **var *Cat*** = ***mongoose***.model(**'Cat'**, { **name**: *String* }) | объявление модели и полей документов этой модели |
| **var *kitty*** = **new *Cat***({ **name**: **'Пушок'** }) | создание экземпляра модели |
| ***kitty***.save() | 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.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_7\_1

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_7\_1

В удаленном репозитории появляется ветка br\_7\_1

## 7.2. Создание схемы

Начинаем работу с пунктом проверкой ветки

% git branch

br\_7\_1

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_7\_2

% git branch

br\_7\_2

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 mongooseTasks.js

Пушок сказал мяу

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_7\_2

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_7\_2

В удаленном репозитории появляется ветка br\_7\_2

## 7.3. Создание модели данных приложения

Начинаем работу с пунктом проверкой ветки

% git branch

br\_7\_2

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_7\_3

% git branch

br\_7\_3

Перейдем к использованию mongoose для учебного проекта. В корне проекта создадим папку models. В этой папке будем реализовывать модели приложения. Первую модель назовем Cat, поэтому в папке models создадим файл cat.js.

Листинг файла модели

**var mongoose = require('mongoose')**

**var Schema = mongoose.Schema**

**var catSchema = new Schema({**

**title: String,**

**nick: {**

**type: String,**

**unique: true,**

**required: true**

**},**

**avatar: String,**

**desc: String,**

**created:{**

**type:Date,**

**default:Date.now**

**}**

**})**

**module.exports.Cat = mongoose.model("Cat", catSchema)**

Проверим работу модели в файле mongooseTasks.js

**var mongoose = require('mongoose')**

**mongoose.connect('mongodb://localhost/test1')**

**var Cat = require("./models/cat").Cat**

**var cat = new Cat({**

**title: "Барсик",**

**})**

**console.log(cat)**

**cat.save(function(){**

**console.log(arguments)**

**})**

$ node mongooseTasks.js

{ title: 'Барсик',

\_id: 599dae619121212173dfaf74,

created: 2017-08-23T16:33:37.516Z }

{ '0':

{ ValidationError: Cat validation failed: nick: Path `nick` is required}

}

Сработал валидатор mongoose и не база данных, не коллекция, не документ не создадутся. В консоле mongosh проверим базу данных **test1**:

switched to db test1

> db.getCollectionNames()

[ "cats" ]

> db.cats.find()

>

То есть создана база test1, коллекция с ожидаемым именем cats, но документ не создался из-за ошибки валидации.

В создании экземпляра модели добавим required поле nick.

**var *cat*** = **new *Cat***({  
 **title**: **"Барсик"**,  
 **nick**: **"barsik"**})

Проверяем

| команда | объяснение |
| --- | --- |
| ^C | остановка соединения с базой данных |
| node mongooseTasks.js | запуск файла mongooseTasks.js |

$ node createDB.js

{ title: 'Барсик',

nick: 'barsik',

\_id: 599db2041e03e5219058961f,

created: 2017-08-23T16:49:08.938Z }

{ '0': null,

'1':

{ \_\_v: 0,

title: 'Барсик',

nick: 'barsik',

\_id: 599db2041e03e5219058961f,

created: 2017-08-23T16:49:08.938Z },

'2': 1 }

В консоле mongosh

> db.cats.find()

{ "\_id" : ObjectId("599db2041e03e5219058961f"), "title" : "Барсик", "nick" : "barsik", "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

***cat***.save(**function**(err, cat, affected){  
 **console**.log(cat.**title**)  
})

В консоле mongosh очистим базу данных:

> db.dropDatabase()

{ "dropped" : "test1", "ok" : 1 }

и снова запустим выполнение файла mongooseTasks.js

^C

$ node mongooseTasks.js

Барсик

И снова запустим выполнение файла mongooseTasks.js

^C

$ node mongooseTasks.js

MongoError: E11000 duplicate key error collection: test1.cats index: nick\_1 dup key: { : "barsik" }

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.**

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_7\_3

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_7\_3

В удаленном репозитории появляется ветка br\_7\_3

# 8. Отображение данных в браузере

## 8.1. Подключение базы данных

Начинаем работу с пунктом проверкой ветки

% git branch

br\_7\_3

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_8\_1

% git branch

br\_8\_1

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/threecats'**)

Проверим, что приложение работает.

| команда | объяснение |
| --- | --- |
| npm start | запуск сервера |

Проверим, что адреса /karamelka, /kompot, /korthik отображают данные без изменения и навигация между страницами работает.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_8\_1

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_8\_1

В удаленном репозитории появляется ветка br\_8\_1

## 8.2. Обработка параметра в адресе

Начинаем работу с пунктом проверкой ветки

% git branch

br\_8\_1

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_8\_2

% git branch

br\_8\_2

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 cats.js и спроектируем роутер для адресов cats/karamelka, cats/kompot, cats/korthik для отображения котят. Создадим в папке routes файл cats.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 *cats*** = require(**'./routes/cats'**);

…

***app***.use(**'/'**, ***routes***);  
***app***.use(**'/users'**, ***users***);  
***app***.use(**'/cats'**, ***cats***);

…

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 маршрутов, начинающихся с cats'**);  
});  
  
***module***.**exports** = ***router***;

Остановим и запустим сервер, проверим работу маршрута /heroes.

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

Для обработки всех адресов cats/karamelka, cats/kompot, cats/korthik одним роутером в routes/cats.js создадим роутер с параметром:

*/\* GET users listing. \*/****router***.*get*(**'/'**, **function**(req, res, next) {  
 res.send(**'Новый маршрутизатор, для маршрутов, начинающихся с cats'**);  
});  
  
*/\* Страница котят \*/****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 |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

В браузере, по адресам cats/karamelka, cats/kompot, cats/korthik отображаются соответственно karamelka, kompot и korthik.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_8\_2

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_8\_2

В удаленном репозитории появляется ветка br\_8\_2

## 8.3. Извлечение данных из базы

Начинаем работу с пунктом проверкой ветки

% git branch

br\_8\_2

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_8\_3

% git branch

br\_8\_3

В шапке файла роутера routes/cats.js подключим модель.

**var *express*** = require(**'express'**)  
**var *router*** = ***express***.**Router**()

В теле роутера страницы героев сделаем запрос в базе данных.

*/\* Страница героев \*/****router***.*get*(**'/:nick'**, **function**(req, res, next) {  
 ***Cat***.findOne({**nick**:req.**params**.**nick**}, **function**(err,cat){  
 **if**(err) **return** next(err)  
 **if**(!cat) **return** next(**new** *Error*(**"Нет такого котенка в этом мультике"**))  
 res.render(**'cat'**, {  
 **title**: cat.**title**,  
 **picture**: cat.**avatar**,  
 **desc**: cat.**desc** })  
 })  
})

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

В браузере проверим страницы: cats/karamelka, cats/kompot, cats/korthik.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_8\_3

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_8\_3

В удаленном репозитории появляется ветка br\_8\_3

## 8.4. Чистка кода

Начинаем работу с пунктом проверкой ветки

% git branch

br\_8\_3

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_8\_4

% git branch

br\_8\_4

Страницы котят, которые открываются по адресам cats/karamelka, cats/kompot, cats/korthik заполняются данными из базы данных. Таким образом роутеры для запросов /karamelka, /kompot, /korthik больше не нужны. Перед тем как вычистить код от неактуальных маршрутов, в файле шаблона страницы (views/layout/page.ejs) заменим адреса страниц в навигации.

**<li class="nav-item">**

**<a class="nav-link" href="/cats/korthik">Коржик</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="/cats/karamelka">Карамелька</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="/cats/kompot">Компот</a>**

**</li>**

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

Используя навигацию, в браузере, проверим страницы: cats/karamelka, cats/kompot, cats/korthik.

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

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_8\_4

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_8\_4

В удаленном репозитории появляется ветка br\_8\_4

## 8.5. Заполнение меню из базы данных

Начинаем работу с пунктом проверкой ветки

% git branch

br\_8\_4

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_8\_5

% git branch

br\_8\_5

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 acync. 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/cats.js модуль async.

**var *express*** = require(**'express'**);  
**var *router*** = ***express***.**Router**();  
**var *Cat*** = require(**"../models/cat"**).**Cat  
var *async*** = require(**"async"**)

Заготовим шаблон:

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

Опишем как работает этот шаблон. Параллельно шаблон запускает функции массива первого параметра шаблона.

[  
 **function**(callback){  
  
 },  
 **function**(callback){  
  
 }  
 ]

Слово *parallel* переводиться “параллельно” и функции массива запускаются одновременно. Второй параметр шаблона - result. result - это массив значений, которые передаются в callback-и соответственно. В result[0] будет значение, которое передается в callback первой функции массива первого параметра шаблона *parallel*, в result[1] будет значение, которое передается в callback второй функции массива первого параметра шаблона *parallel.*

Заполним список задач шаблона запросами и воспользуемся структурой переменной result.

***async***.*parallel*([  
 **function**(callback){  
 ***Cat***.findOne({**nick**:req.**params**.**nick**},**function**(err,cat){  
 callback(err,cat)  
 })  
 },  
 **function**(callback){  
 ***Cat***.*find*({},**function**(err,cats){  
 callback(err,cats)  
 })  
 }  
 ],  
 **function**(err,result){  
 **if**(err) **return** next(err)  
 **var cat** = result[0]  
 **if**(!**cat**) **return** next(**new** *Error*(**"Нет такого котенка в мультике Три кота"**))  
 **console**.log(**cat**.**avatar**)  
 res.render(**'cat'**, {  
 **title**: **cat**.**title**,  
 **picture**: **cat**.**avatar**,  
 **desc**: **cat**.**desc** });  
 })

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

Проверим, что приложение работает без изменений. Оптимизируем код:

*/\* Страница котят \*/****router***.*get*(**'/:nick'**, **function**(req, res, next) {  
 ***async***.*parallel*([  
 **function**(callback){  
 ***Cat***.findOne({**nick**:req.**params**.**nick**}, callback)  
 },  
 **function**(callback){  
 ***Cat***.*find*({},callback)  
 }  
 ],  
 **function**(err,result){  
 **if**(err) **return** next(err)  
 **var** cat = result[0]  
 **if**(!cat) **return** next(**new** *Error*(**"Нет такого котенка в мультике Три кота"**))  
 **console**.log(cat.**avatar**)  
 res.render(**'cat'**, {  
 **title**: cat.**title**,  
 **picture**: cat.**avatar**,  
 **desc**: cat.**desc** });  
 })  
})

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

**function**(err,result){  
 **if**(err) **return** next(err)  
 **var** cat = result[0]  
 **var** cats = result[1]  
 **console**.log(cats)  
 **if**(!cat) **return** next(**new** *Error*(**"Нет такого котенка в мультике Три кота"**))  
 res.render(**'cat'**, {  
 **title**: cat.**title**,  
 **picture**: cat.**avatar**,  
 **desc**: cat.**desc** });  
})

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

***Cat***.*find*({},{**\_id**:0,**title**:1,**nick**:1},callback)

| параметр | объяснение |
| --- | --- |
| \_id:0 | поле \_id не включать в ответ базы данных |
| title:1 | поле title включать в ответ базы данных |
| nick:1 | поле nick включать в ответ базы данных |

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

Проверим запрос.

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

В консоле получим:

[ { title: 'Карамелька', nick: 'karamelra' },

  { title: 'Компот', nick: 'kompot' },

  { title: 'Коржик', nick: 'korthik' },

]

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

**var *express*** = require(**'express'**)  
**var *router*** = ***express***.**Router**()  
**var *Cat*** = require(**"../models/cat"**).**Cat  
var *async*** = require(**"async"**)  
  
*/\* GET users listing. \*/****router***.*get*(**'/'**, **function**(req, res, next) {  
 res.send(**'Новый маршрутизатор, для маршрутов, начинающихся с cats'**)  
});  
  
*/\* Страница котят \*/****router***.*get*(**'/:nick'**, **function**(req, res, next) {  
 ***async***.*parallel*([  
 **function**(callback){  
 ***Cat***.findOne({**nick**:req.**params**.**nick**}, callback)  
 },  
 **function**(callback){  
 ***Cat***.*find*({},{**\_id**:0,**title**:1,**nick**:1},callback)  
 }  
 ],  
 **function**(err,result){  
 **if**(err) **return** next(err)  
 **var** cat = result[0]  
 **var** cats = result[1] || []  
 **if**(!cat) **return** next(**new** *Error*(**"Нет такого котенка в мультике Три кота"**))  
 res.render(**'cat'**, {  
 **title**: cat.**title**,  
 **picture**: cat.**avatar**,  
 **desc**: cat.**desc**,  
 **menu**: cats  
 });  
 })  
})  
  
***module***.**exports** = ***router***

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

Исходный код:

**<li class="nav-item">**

**<a class="nav-link" href="/cats/korthik">Коржик</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="/cats/karamelka">Карамелька</a>**

**</li>**

**<li class="nav-item">**

**<a class="nav-link" href="/cats/kompot">Компот</a>**

**</li>**

Преобразованный код:

<% menu.*forEach*(**function**(cat){ %>

**<li class="nav-item">**

**<a class="nav-link" href="/cats/**<%=cat.**nick%>">**<%=cat.**title%></a>**

**</li>**  
<% }) %>

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

**var *express*** = require(**'express'**)  
**var *router*** = ***express***.**Router**()  
**var *Cat*** = require(**"../models/cat"**).**Cat***/\* GET home page. \*/****router***.*get*(**'/'**, **function**(req, res, next) {  
 ***Cat***.*find*({},{**\_id**:0,**title**:1,**nick**:1},**function**(err,menu){  
 res.render(**'index'**, {  
 **title**: **'Express'**,  
 **menu**: menu  
 });  
 })  
  
});  
  
***module***.**exports** = ***router***;

В шаблон views/error.ejs передадим пустой массив как как значение переменной **menu**: [].

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

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_8\_5

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_8\_5

В удаленном репозитории появляется ветка br\_8\_5

# 9. Cookie и Session

## 9.1. Установка модуля express-session и настройка cookie

Начинаем работу с пунктом проверкой ветки

% git branch

br\_8\_5

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_9\_1

% git branch

br\_9\_1

Session это хранилище данных между HTTP запросами. Session создает sid – session id и помещает sid в Cookie. Сооkie это текстовая информация, которую сервер передает браузеру, браузер хранит и возвращает на сервер, для доступа к данным ceссии. 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/threecats"**)  
**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**: **"ThreeCats"**,  
 **cookie**:{**maxAge**:60\*1000},

**resave**: **true**,

**saveUninitialized**: **true**   
}))

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

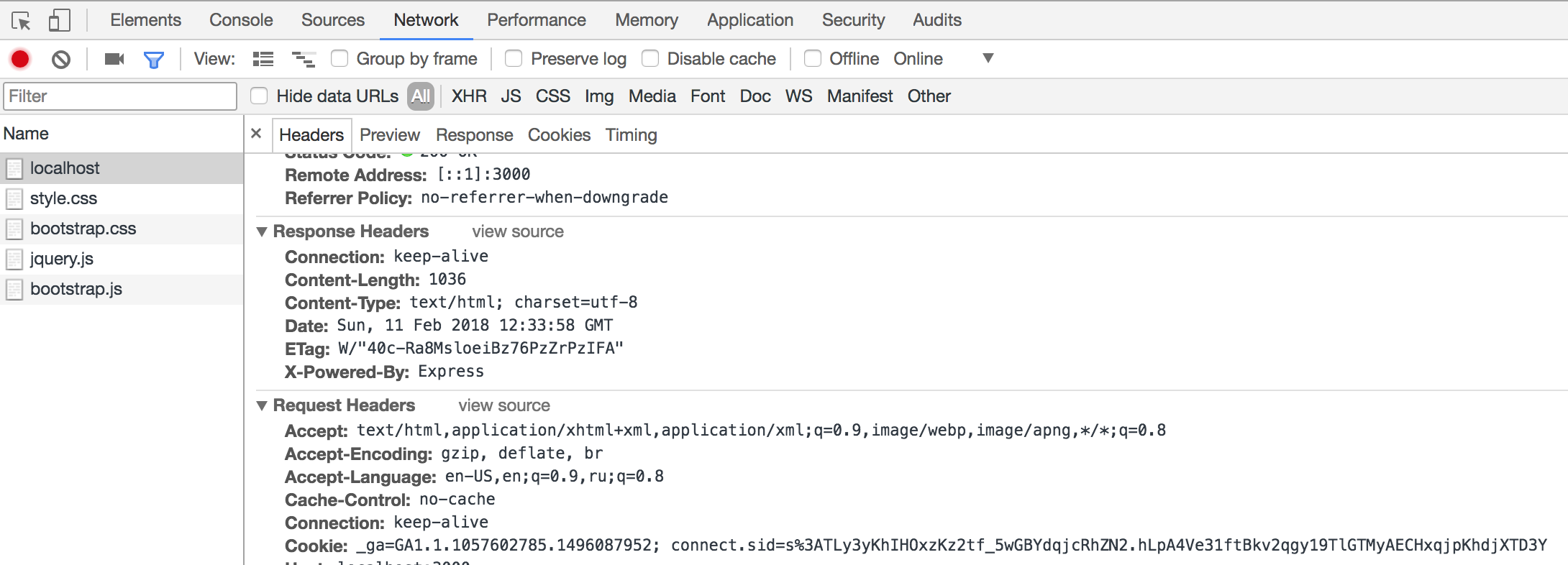
Опция secret является обязательной, опция **maxAge**:60\*1000 определяет время жизни session в миллисекундах (60\*1000 = 1 минута).

Запустим server.

| команда | объяснение |
| --- | --- |
| npm start | запуск сервера |

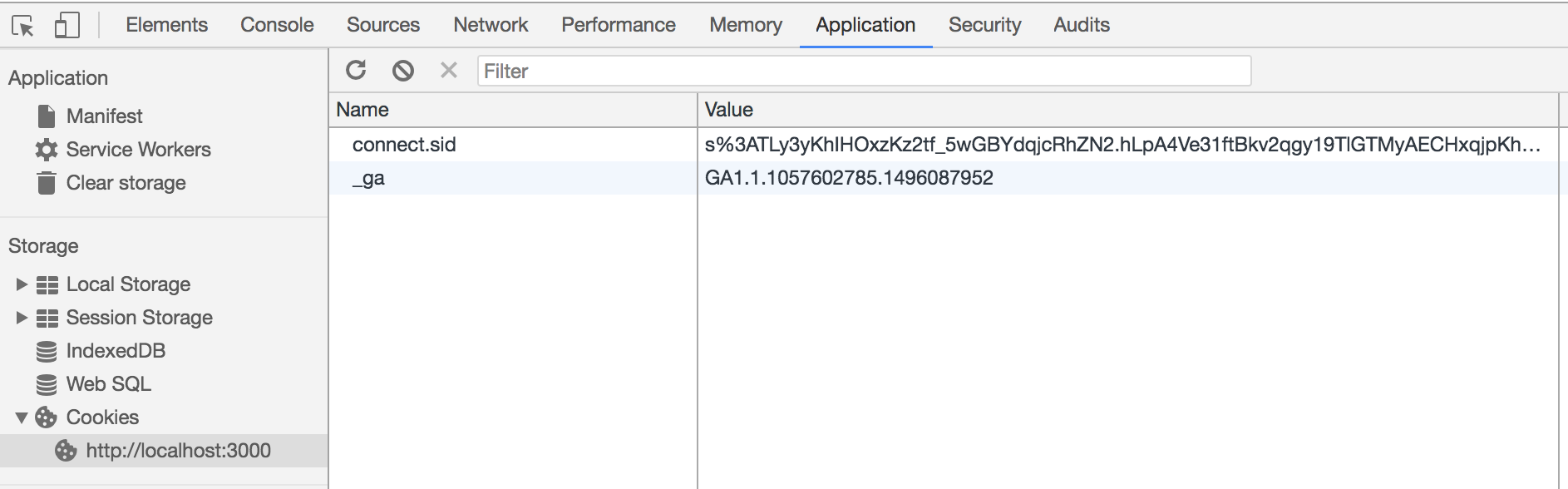
Перейдем в браузер http://localhost:3000/ и найдем куки с sid (session id).

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



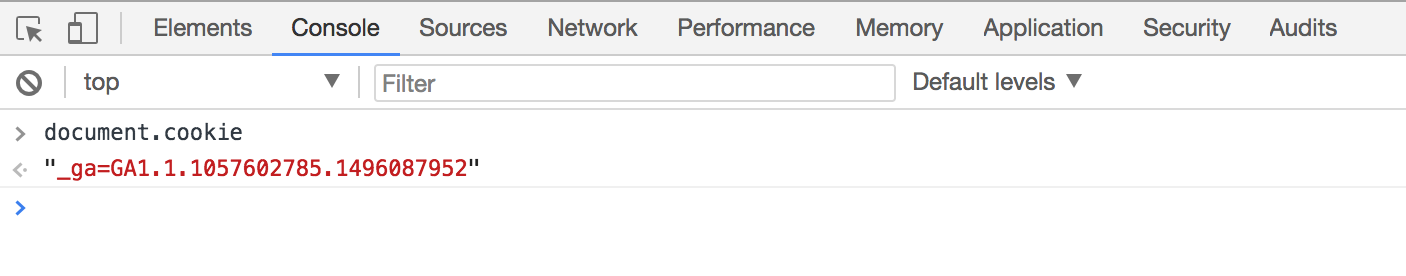
Открываем вкладку Network => localhost => Request Headers => Cookie => connect.sid

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 Java Script.



Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_9\_1

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_9\_1

В удаленном репозитории появляется ветка br\_9\_1

## 9.2. Команда записи в cookie

Начинаем работу с пунктом проверкой ветки

% git branch

br\_9\_1

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_9\_2

% git branch

br\_9\_2

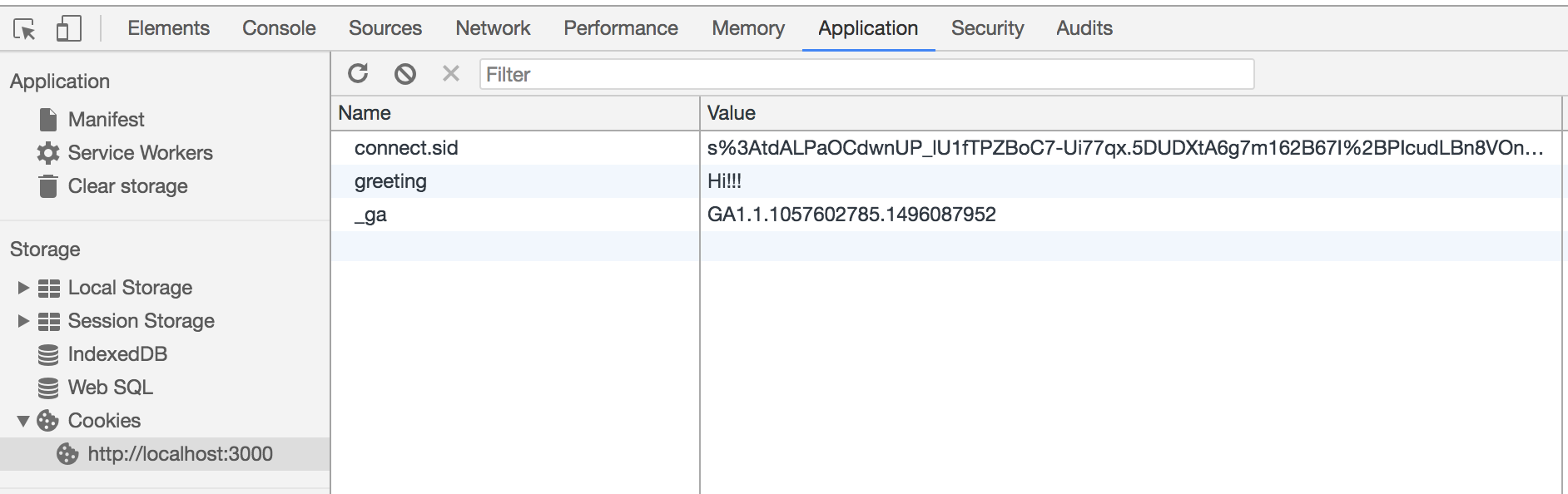
В файле routes/index.js, в обработчике запроса главной странице добавим данные в куки.

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

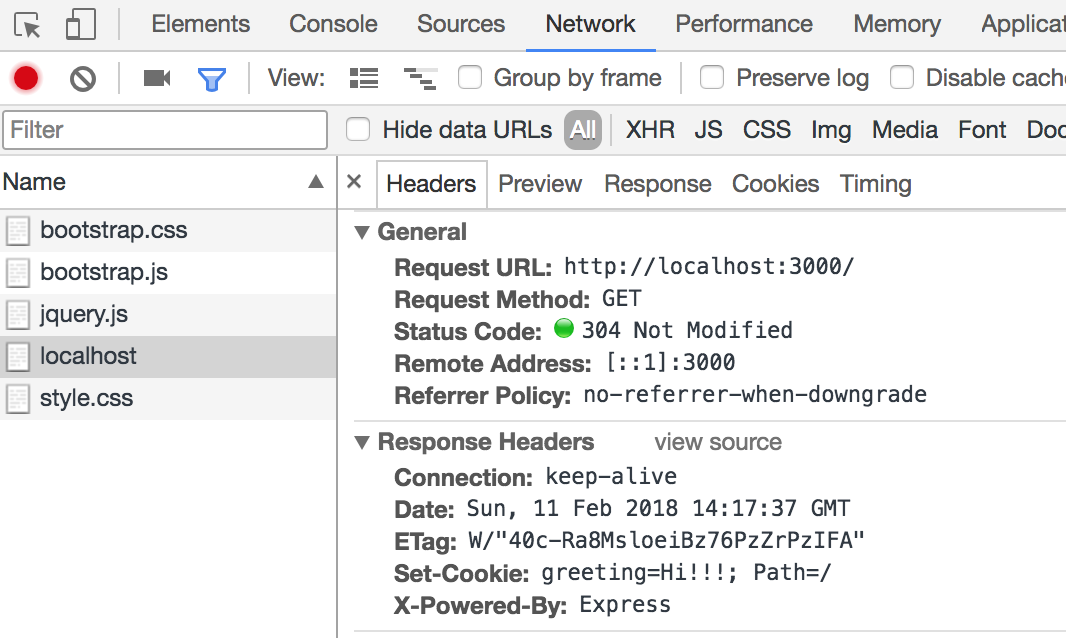
Остановим сервер, запустим сервер и проверим куки в браузере.

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

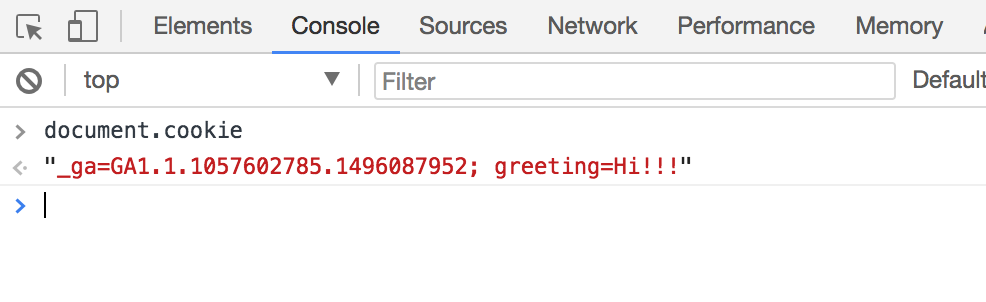
Во вкладке Application.



Во вкладке Network.



В JavaScript консоле.



Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_9\_2

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_9\_2

В удаленном репозитории появляется ветка br\_9\_2

## 

## 9.3. Сохранение session в MongoDB

Начинаем работу с пунктом проверкой ветки

% git branch

br\_9\_2

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_9\_3

% git branch

br\_9\_3

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},

**resave: true,**

**saveUninitialized: true**,  
 **store**: MongoStore.create({mongoUrl: 'mongodb://localhost/threecats'})  
}))

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 threecats

switched to db threecats

> db.getCollectionNames()

[ "cats", "sessions" ]

> db.sessions.find()

{ "\_id" : "tdALPaOCdwnUP\_lU1fTPZBoC7-Ui77qx", "session" : "{\"cookie\":{\"originalMaxAge\":null,\"expires\":null,\"httpOnly\":true,\"path\":\"/\"},**\"greeting\":\"Hi!!!\"**}", "expires" : ISODate("2018-02-25T13:29:30.922Z") }

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_9\_3

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_9\_3

В удаленном репозитории появляется ветка br\_9\_3

## 9.4. Создание счетчика посещения страниц сайта

Начинаем работу с пунктом проверкой ветки

% git branch

br\_9\_3

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_9\_4

% git branch

br\_9\_4

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},

**resave: true,**

**saveUninitialized: true**,  
  **store: MongoStore.create({mongoUrl: 'mongodb://localhost/threecats'}))  
}))**  
***app***.use(**function**(req,res,next){  
    req.**session**.**counter** = req.**session**.**counter** +1 || 1

next()  
})

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" : "tdALPaOCdwnUP\_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 |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

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

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_9\_4

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_9\_4

В удаленном репозитории появляется ветка br\_9\_4

## 9.5. Глобальная переменная для навигации

Начинаем работу с пунктом проверкой ветки

% git branch

br\_9\_4

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_9\_5

% git branch

br\_9\_5

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

Синатаксис 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** = []  
  
 Cat.*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 *Cat*** = require(**"./models/cat"**).**Cat**

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

<% **if**(**typeof** nav == **'object'** && nav){ nav.*forEach*(**function**(cat){ %>

**<li class="nav-item">**

**<a class="nav-link" href="/cats/**<%=cat.**nick%>">**<%=cat.**title%></a>**

**</li>**  
<% })} %>

Уберем переменную menu 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.

Уберем из запроса страниц котят шаблон parallel

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

Проверим, что приложение работает с новый 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 переменой 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 *Cat*** = require(**"./../models/cat"**).**Cat  
  
  
*module***.exports = **function**(req,res,next){  
 res.**locals**.**nav** = []  
  
 ***Cat***.*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 |

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_9\_5

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_9\_5

В удаленном репозитории появляется ветка br\_9\_5

# 10. Аутентификация

## 10.1. Создание страницы регистрации

Начинаем работу с пунктом проверкой ветки

% git branch

br\_9\_5

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_1

% git branch

br\_10\_1

К стандартной функиональности 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',{title: 'Вход'});

});

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="conteiner"**>  
 <**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 |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

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

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_1

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_1

В удаленном репозитории появляется ветка br\_10\_1

## 10.2. Создание модели User

Начинаем работу с пунктом проверкой ветки

% git branch

br\_10\_1

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_2

% git branch

br\_10\_2

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 7 при создании модели хранения данных Cat.

Для хранения моделей в проекте в Главе 7 была создана специальная папка models. Структурирование проекта является частью хорошего стиля программирования. Название папки подчеркивает назначение папки. Папка models самим 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'**)  
}

Размещаем код перед строкой:

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

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 сheckUser.js | выполнение файла checkUser.js |
| mongosh | запуск Mongo Shell |
| use threecats | переход в базу данных threecats |
| 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 }

>

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_2

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_2

В удаленном репозитории появляется ветка br\_10\_2

## 10.3. Подготовка данных для передачи на сервер

Начинаем работу с пунктом проверкой ветки

% git branch

br\_10\_2

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_3

% git branch

br\_10\_3

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” | тригер 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 пост примет вид:

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

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_3

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_3

В удаленном репозитории появляется ветка br\_10\_3

## 10.4. Логика пользователя

Начинаем работу с пунктом проверкой ветки

% git branch

br\_10\_3

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_4

% git branch

br\_10\_4

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

**var *Cat*** = require(**"./../models/cat"**).**Cat  
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**.**hashedPassword**}

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'**, {title: 'Вход'})  
 }  
 } **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.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_4

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_4

В удаленном репозитории появляется ветка br\_10\_4

## 10.5. Глобальная переменная user

Начинаем работу с пунктом проверкой ветки

% git branch

br\_10\_4

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_5

% git branch

br\_10\_5

Создадим глобальную переменнную user, которая будет доступна во всех шаблонах, по аналогии с созданием глобальной переменнной 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 пременной 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.

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_5

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_5

В удаленном репозитории появляется ветка br\_10\_5

## 10.6. Обработка ошибки аутентификации

Начинаем работу с пунктом проверкой ветки

% git branch

br\_10\_5

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_6

% git branch

br\_10\_6

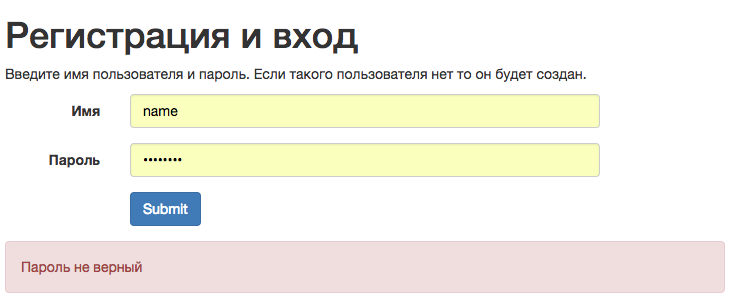
На странице регистрационной формы добавим <div> для отображения ошибки аутентификации:

…

<**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.



Шаблон регистрационной формы вызывается в приложении дважды. Первый раз, когда открываем страницу регистрационной формы при нажатии 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 присваеваем 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**>

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_6

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_6

В удаленном репозитории появляется ветка br\_10\_6

## 10.7. Функциональность logout

Начинаем работу с пунктом проверкой ветки

% git branch

br\_10\_6

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_7

% git branch

br\_10\_7

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 функцтональность 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.**locals**.**user** = **null**  
 res.redirect(**'/'**)  
});

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_7

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_7

В удаленном репозитории появляется ветка br\_10\_7

## 10.8. Закрытие страниц сайта для незалогиненного пользователя

Начинаем работу с пунктом проверкой ветки

% git branch

br\_10\_7

Cоздаем новую ветку и переходим на нее

% git checkout -b br\_10\_8

% git branch

br\_10\_8

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 переменнной 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**){  
 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"**)

**Обновление 2022:**

Подключим посредник checkAuth.js в шапке файла routes/heroes.js.

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

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

…

| команда | объяснение |
| --- | --- |
| ^C | остановка сервера |
| npm start | запуск сервера |

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

Заканчиваем работу с пунктом проверкой ветки

% git branch

br\_10\_8

Комитим и пушим в удаленный репозиторий ветку.

% git add .

% git commit -m «Описываем что сделано»

% git push origin br\_10\_8

В удаленном репозитории появляется ветка br\_10\_8

*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.

Формат 60х84 /16. Усл. печ. л.8,0 .Заказ № 005/1014. Тираж 300 экз.

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.