Владимирский государственный университет

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ НА СТОРОНЕ СЕРВЕРА

Лабораторный практикум

Владимир 2026

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ НА СТОРОНЕ СЕРВЕРА

Лабораторный практикум

Электронное издание

Владимир 2026

ISBN 978-5-9984-2156-3
© ВлГУ, 2026

УДК 004.774
ББК 32.973

Автор-составитель А. С. Максимова

Рецензенты:
Кандидат технических наук, доцент
доцент кафедры информатики и защиты информации
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
Д. А. Полянский

Старший преподаватель кафедры информационных систем и программной инженерии Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых,
ведущий разработчик компьютерного программного обеспечения
ООО «КУЛ» (г. Владимир)
А. И. Петрова

Проектирование и разработка веб-приложений на стороне сервера [Электронный ресурс] : лаб. практикум / авт.-сост. А. С. Максимова ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2026. – 160 с. – ISBN 978-5-9984-2156-3. – Электрон. дан. (3,34 Мб). – 1 электрон. опт. диск (CD-ROM). – Систем. требования: Intel от 1,3 ГГц ; Windows XP/7/8/10 ; Adobe Reader ; дисковод CD-ROM. – Загл. с титул. экрана.

Содержит лабораторные работы по дисциплине «Проектирование и разработка веб-приложений», а также теоретический материал, раскрывающий особенности разработки серверной части на языке JavaScript с помощью кроссплатформенной среды Node.js.

Предназначен для студентов СПО очной формы обучения направлений подготовки 09.02.07 – Информационные системы и программирование, 09.02.09 – Веб-разработка.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС СПО.

Ил. 74. Библиогр.: 13 назв.

ISBN 978-5-9984-2156-3 © ВлГУ, 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
Лабораторная работа № 1. ВВЕДЕНИЕ В NODE.JS 6
Лабораторная работа № 2. ФАЙЛЫ NODE.JS 17
Лабораторная работа № 3. HTTP СЕРВЕР NODE.JS 24
Лабораторная работа № 4. HTTPS СЕРВЕР NODE.JS 35
Лабораторная работа № 5. ФРЕЙМВОРК EXPRESS.JS 38
Лабораторная работа № 6. ШАБЛОНИЗАТОР PUG 48
Лабораторная работа № 7. СОЗДАНИЕ И ОБРАБОТКА ФОРМ PUG 56
Лабораторная работа № 8. ШАБЛОНИЗАТОР EJS 59
Лабораторная работа № 9. СРЕДСТВА ДЛЯ РАБОТЫ С БАЗОЙ ДАННЫХ В NODE.JS 71
Лабораторная работа № 10. МИГРАЦИИ БАЗ ДАННЫХ 83
Лабораторная работа № 11. ТЕХНОЛОГИЯ ORM 90
Лабораторная работа № 12. АВТОРИЗАЦИЯ И РЕГИСТРАЦИЯ ПОЛЬЗОВАТЕЛЯ В СИСТЕМЕ 101
Лабораторная работа № 13. ЗАГРУЗКА ФАЙЛОВ 121
Лабораторная работа № 14. ЛОГИРОВАНИЕ 136
Лабораторная работа № 15. EMAIL-РАССЫЛКА 142
Лабораторная работа № 16. РАЗВЕРТЫВАНИЕ ВЕБ-ПРИЛОЖЕНИЯ 148
ЗАКЛЮЧЕНИЕ 157
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 158
РЕКОМЕНДАТЕЛЬНЫЙ БИБЛИОГРАФИЧЕСКИЙ СПИСОК 159

ВВЕДЕНИЕ

Разработка сервера – важная часть создания веб-приложения, а также мобильных приложений. Сервер обеспечивает обработку запросов клиента, хранение данных, бизнес-логику и взаимодействие с другими сервисами.

Существуют разные технологии для разработки сервера, одна из них – Node.js. Среди других инструментов ее выделяет возможность использовать JavaScript на всём стеке (клиентская часть React/Vue, серверная часть Node.js, мобильное приложение React Native), что устраняет необходимость переключения между языками и ускоряет разработку. Асинхронная архитектура, обеспечивающая высокую производительность I/O-операций, поддержка крупными компаниями (Netflix, PayPal, Uber) и богатая экосистема npm (Express.js, Socket.io, Mongoose) позволяют пользователям Node.js создавать масштабируемые веб- и мобильные приложения.

Шаблонизаторы (или шаблонизаторы представлений) – это инструменты, позволяющие генерировать HTML-страницы на сервере динамически, подставляя данные в готовые шаблоны. Их основная задача – отделить логику построения интерфейса (верстку) от бизнес-логики приложения (обработки данных). Вместо того чтобы вручную «собирать» HTML-строки внутри JavaScript-кода (что трудночитаемо и плохо поддерживается), разработчик создает файл шаблона, размеченный специальными тегами или синтаксисом. Когда пользователь запрашивает страницу, сервер выполняет код, получает необходимые данные из базы, передает их в шаблонизатор, который на их основе «собирает» итоговую HTML-страницу и отправляет ее клиенту. Это упрощает разработку, делает код чище и позволяет переиспользовать общие части страниц («шапку», «подвал») на всем сайте.

Программа дисциплины «Проектирование и разработка веб-приложений» рассчитана на два семестра (5 – 6-й семестры) и предполагает проведение 16 лабораторных работ, предусматривающих изучение сервера, написанного на Node.js.

Рассматриваемые темы:

– введение в Node.js (работа с основными понятиями, изучение особенностей консоли);

– файлы Node.js (анализ методов работы с файловой системой в Node.js);

– HTTP сервер Node.js (построение простейших HTTP-запросов в Node.js);

– HTTPS сервер Node.js (работа с самоподписанными сертификатами, построение простейших HTTPS-запросов в Node.js);

– шаблонизатор pug (изучение возможностей шаблонизатора pug);

– создание и обработка форм pug (изучение шаблонизатора pug);

– шаблонизатор ejs (изучение шаблонизатора ejs);

– средства для работы с базой данных (БД) Node.js (изучение некоторых библиотек для подключения и создание запросов в БД);

– миграции баз данных (изучение библиотеки knex);

– технология ORM (изучение библиотеки Sequelize);

– авторизация и регистрация (изучение механизмов регистрации и авторизации с помощью библиотеки knex);

– загрузка файлов (изучение библиотеки multer);

– логирование (изучение понятия логирования);

– email-рассылка (изучение метода работы с email-рассылкой);

– развертывание веб-приложения (работа с docker).

Полученные при выполнении лабораторных работ навыки будут полезны при работе с сервером и шаблонизаторами, интеграции базы данных в сервер и т. д.

Лабораторная работа № 1. ВВЕДЕНИЕ В NODE.JS

Цель работы: решение простейших задач на программирование с вводом данных из консоли и выводом в консоль, работа с аргументами командной строки.

Формируемые компетенции: ПК 9.2, ПК 9.4, ПК 9.5.

Общие сведения

Node (Node.js) – кроссплатформенная среда исполнения с открытым исходным кодом. Она позволяет разработчикам создавать приложения и серверные инструменты, используя язык JavaScript вне контекста браузера.

Установка Node.js

Для загрузки необходимо перейти на официальный сайт https://nodejs.org/en/. На главной странице (рис. 1.1) есть две возможные опции для загрузки: самая последняя версия Node.js и LTS-версия (рекомендуемая).

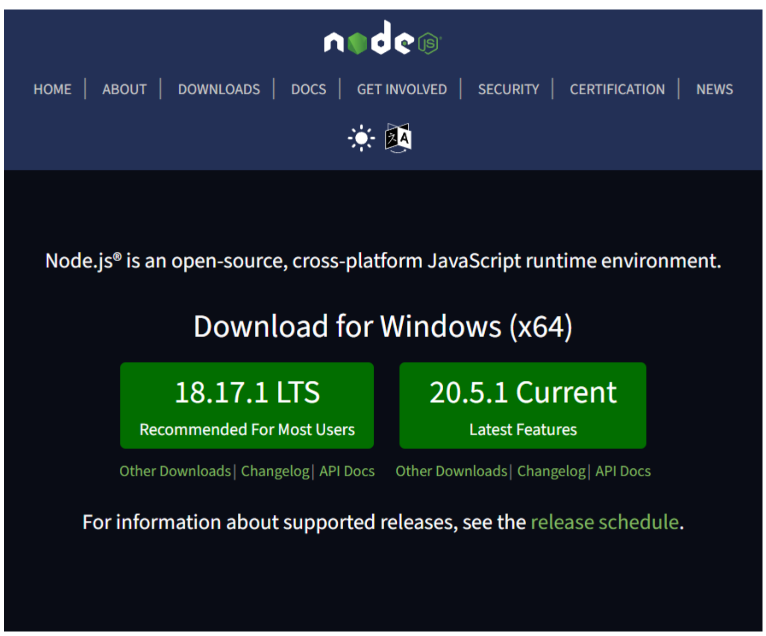


Рис. 1.1. Главная страница сайта Node.js

Проверить версию Node.js можно, введя в командную строку node –v.

Вывод данных в консоль

Создайте новый каталог helloapp и добавьте в него новый файл app.js со следующим кодом.

Листинг кода 1.1

```
console.log("Hello world");
```

Ввод данных с клавиатуры

Node.js предоставляет модуль readline, который предлагает интерфейс для чтения данных из читаемого потока (по одной строке за раз).

Пример. При запуске кода программа запрашивает имя пользователя. Введите данные в консоль.

Листинг кода 1.2

```
const readline = require('readline')
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
})
rl.question(`Как вас зовут? `, name => {
    console.log(`Привет,${name}!!`)
    rl.close()
})
```

В примере метод readline.createInterface() используется для создания экземпляра readline путем определения потоков, доступных для чтения и записи.

Метод rl.question() отображает запрос (the question) и ожидает, пока пользователь введет ответ. Как только входные данные становятся доступны, метод вызывает метод обратного вызова, передавая входные данные пользователя в качестве первого параметра.

Затем вызывается метод rl.close() в последнем обратном вызове, чтобы закрыть readline интерфейс.

Запуск файла

Код для Node.js можно прописывать и выполнять прямо в командной строке, но это неудобно. Вместо того чтобы вводить весь код напрямую в консоль, проще вынести его во внешний файл.

В командной строке с помощью команды cd необходимо перейти к каталогу «проект», а затем выполнить команду.

Листинг кода 1.3

```
node app.js
```

Результат работы с файлом представлен на рис. 1.2.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node app.js
Как вас зовут? Алина
Привет,Алина!!
```

Рис. 1.2. Результат работы команды «node app.js»

Работа с аргументами командной строки

Аргументы командной строки – это строки текста, используемые для передачи дополнительной информации программе, когда приложение запускается через интерфейс командной строки (CLI) операционной системы. Аргументы командной строки обычно включают информацию, используемую для установки значений конфигурации или свойств для приложения.

В большинстве случаев аргументы передаются после имени программы в приглашении. Пример синтаксиса аргументов командной строки представлен ниже.

Листинг кода 1.4

```
$ [runtime] [script_name] [argument-1 argument-2 argument-3 ... argument-n]
```

Здесь средой, выполняющей программу/сценарий, может быть, например, sh, java, node и т. д. Аргументы обычно разделяются пробелом, однако есть некоторые среды выполнения, в которых используются запятые, чтобы различать несколько аргументов командной строки. Кроме того, в зависимости от программы можно передавать аргументы в виде пар «ключ – значение».

Приведем основные преимущества использования аргументов командной строки:

1. Можно передать информацию приложению до его запуска. Это полезно, если необходимо выполнить большое количество настроек конфигурации.

2. Аргументы командной строки передаются программе в виде строк. Типы данных String можно легко преобразовать в другие типы данных в приложении, что делает аргументы очень гибкими.

3. Можно передавать неограниченное количество аргументов через командную строку.

4. Аргументы командной строки используются вместе со сценариями и пакетными файлами, что важно для автоматического тестирования.

Недостатки использования:

1. Самый большой недостаток передачи информации через командную строку состоит в том, что интерфейс требует сложного обучения, поэтому большинству людей трудно использовать его, если у них нет опыта работы с инструментами CLI.

2. Приложения командной строки могут вызвать трудности, если они не установлены на компьютер, поэтому обычно они не используются на небольших устройствах, таких как телефоны или планшеты.

Передача аргументов командной строки в Node.js

Как и многие другие языки, язык приложения Node.js также работает с аргументами командной строки. По умолчанию Node.js обрабатывает принятие аргументов сам, но есть также несколько эффективных сторонних пакетов с очень полезными функциями.

Рассмотрим более подробно, как использовать аргументы встроенным способом (process.argv), а также с помощью популярных пакетов minimist и yargs.

Использование process.argv

Самый простой способ получить аргументы в Node.js – через массив process.argv. Это глобальный объект, который можно использовать без импорта каких-либо дополнительных библиотек. Для этого необходимо передать аргументы приложению Node.js (см. листинг кода 1.2), затем к ним можно получить доступ в приложении непосредственно через массив.

Первый элемент process.argv всегда – путь файловой системы, указывающий на исполняемый файл node, второй элемент – имя исполняемого файла JavaScript, третий – первый аргумент, фактически переданный пользователем.

Для того чтобы избежать путаницы, следует помнить, что JavaScript использует индексы с нулевым отсчетом для массивов (как и многие другие языки). Это значит, что первый элемент будет сохранен с индексом «0», а последний – с индексом «n-1», где «n» – общее количество элементов в массиве.

Ниже код демонстрирует простой сценарий Node, который выводит все аргументы командной строки, переданные приложению, вместе с их индексом. Создайте файл processargv.js. и вставьте в него код.

Листинг кода 1.5

```
'use strict';
for (let j = 0; j < process.argv.length; j++) {
console.log(j + ' -> ' + (process.argv[j]));
}
```

Все, что делает этот сценарий, – это перебирает массив process.argv и печатает индексы вместе с элементами, хранящимися в них. Это очень полезно для отладки, если есть сомнения в том, какие аргументы получают и в каком порядке.

Для того чтобы запустить код, введите следующую команду. Для этого откройте папку, в которой сохранен файл processargv.js.

Листинг кода 1.6

```
node processargv.js tom jack 43
```

Передайте программе processargv.js три аргумента, которые будут сохранены в массиве: «tom» – в индексе «2», а «jack» и «43» – в индексах «3» и «4» соответственно. Результат работы представлен на рис. 1.3.

```
PS D:\!!!DOCUMENTs\Desktop\проект>  node processargv.js tom jack 43
0 -> C:\Program Files\nodejs\node.exe
1 -> D:\!!!DOCUMENTs\Desktop\проект\processargv.js
2 -> tom
3 -> jack
4 -> 43
```

Рис. 1.3. Результат работы команды «node processargv.js tom jack 43»

Первый индекс содержит путь к среде выполнения, второй индекс – путь к файлу сценария; остальные индексы содержат аргументы, которые передаются в определенной последовательности.

Использование модуля minimist

Еще один способ получить аргументы командной строки в приложении Node.js – использовать модуль minimist. Установить его можно с помощью следующей команды.

Листинг кода 1.7

```
npm install minimist
```

Модуль minimist анализирует аргументы из массива process.argv и преобразует его в более простой в использовании ассоциативный массив, в котором можно получить доступ к элементам через имена индексов в дополнение к их номерам.

Листинг кода 1.8

```
'use strict';
const args = require('minimist')(process.argv.slice(2));
console.log(args);
console.log(args.i);
```

Сохраните приведенный выше код в файл minimist.js. В коде используется метод slice глобального объекта process.argv. В slice удаляются все предыдущие элементы массива, начиная с индекса, переданного ему в качестве параметра. Аргументы, которые передаются вручную, сохраняются, начиная со второго индекса. В результате распечатывается весь объект args. Также выводится отдельный элемент массива с помощью именованного индекса, то есть «i», а не номер индекса.

Теперь, когда программе передается аргумент (см. листинг кода 1.8), можно указать символ, с помощью которого необходимо получить доступ к элементу. Поскольку в сценарии используется индексное имя «i» для доступа к элементу, указывают элемент, хранящийся в этом индексе. Ниже представлена команда для запуска кода программы.

Листинг кода 1.9

```
node minimist.js -i jacob -j 45
```

Здесь «i» указано в качестве имени для второго индекса; значение, хранящееся в этом индексе, – «jacob». Точно также третий индекс называется "j", его значение – 45. Результат выполнения вышеуказанной команды представлен на рис. 1.4.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node minimist.js -i jacob -j 45
{ _: [], i: 'jacob', j: 45 }
jacob
```

Рис. 1.4. Результат работы команды «node minimist.js -i jacob -j 45»

Несмотря на то, что minimist не такой многофункциональный, как некоторые другие модули анализа аргументов (например, yargs), он имеет несколько полезных функций, на которые необходимо обратить внимание, в частности псевдонимы и значения по умолчанию.

Если установить псевдоним, то вызывающая программа может использовать его в дополнение к обычному имени опции. Это полезно, если есть необходимость в сокращенном обозначении для вариантов, как показано ниже.

Листинг кода 1.10

```
'use strict';
const minimist = require('minimist');
 let args = minimist(process.argv.slice(2), {
alias: {
h: 'help',
v: 'version'
}
});
 console.log('args:', args);
```

Необходимо сохранить приведенный выше код в minimist-alias.js. Вызов этого кода с параметром –h устанавливает, что h: true и help: true – выходные аргументы. Результат работы представлен на рис. 1.5.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node minimist-alias.js -h
args: { _: [], h: true, help: true }
```

Рис. 1.5. Результат работы команды «node minimist-alias.js»

Что касается функции minimist по умолчанию, то если для параметра не передано значение, то значение по умолчанию, которое было установлено в программе, будет использоваться автоматически.

Листинг кода 1.11

```
'use strict';
const minimist = require('minimist');
let args = minimist(process.argv.slice(2), {
default: {
port: 8080
},
});
console.log('args:', args);
```

Необходимо сохранить приведенный выше код в файле minimist-defaults.js. Вызов этого кода без –port передает port в возвращенный объект args. Результат работы представлен на рис. 1.6.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node minimist-defaults.js
args: { _: [], port: 8080 }
```

Рис. 1.6. Результат работы команды «node minimist-defaults.js»

Использование модуля yargs

Еще один модуль, который помогает анализировать аргументы командной строки, передаваемые программам Node.js, – это модуль yargs. С его помощью можно передавать аргументы в виде пар «ключ – значение», а затем получать доступ к значениям аргументов програм-

мы, используя соответствующие ключи. Установить yargs можно посредством следующей команды.

Листинг кода 1.12

```
npm install yargs
```

Листинг кода 1.13

```
'use strict';
const args = require('yargs').argv;
console.log('Name: ' + args.name);
console.log('Age: ' + args.age);
```

В приведенном выше сценарии отображены значения, указанные для аргументов «имя» и «возраст», которые были переданы через командную строку. Сохраните его в файле с именем yargs.js.

При выполнении программы (см. листинг кода 1.13) важно передать значения аргументов «имя» и «возраст», в противном случае для аргументов по умолчанию будет значение undefined.

Для того чтобы выполнить указанную выше программу, введите в командной строке следующую команду.

Листинг кода 1.14

```
node yargs.js --name=jacob --age=45
```

Результат скрипта (см. листинг кода 1.13) представлен на рис. 1.7.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node yargs.js --name=jacob --age=45
Name: jacob
Age: 45
```

Рис. 1.7. Результат работы команды «node yargs.js --name=jacob --age=45»

Пакет yargs предоставляет не только стандартные функции анализа аргументов, но и множество дополнительных опций.

Один из самых эффективных способов использования модуля yargs – метод .command(), который помогает создавать, открывать и вызывать функции Node.js через командную строку. Простой пример работы этой функции представлен ниже.

Листинг кода 1.15

```
'use strict';
const argv = require('yargs')
.command('upload', 'upload a file', (yargs) => {},
(argv) => {
console.log('Uploading your file now...');

// Do stuff here
}).argv;
```

Выполнение программы с помощью команды upload вызовет переданную функцию, которая в данном случае просто выводится в командную строку. Например, эту команду можно вызвать при условии, что приведенный выше код хранится в файле с именем s3-tool.js.

Результат работы представлен на рис. 1.8.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node s3-tool.js upload --file=my-file.txt --bucket=my-s3-bucket
Uploading your file now...
```

Рис. 1.8. Результат работы команды «node s3-tool.js upload»

Функция .command() даёт больше возможностей, чтобы вывести необходимые и необязательные параметры из командной строки.

Листинг кода 1.16

```
'use strict';
const yargs = require('yargs');
yargs.command('login <username> [password]', 'au-
thenticate with the server').argv
```

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).
2. Решите простейшие задачи на программирование с вводом данных из консоли и выводом в консоль.
3. Продемонстрируйте работу с аргументами командной строки.
4. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое Node.js?
2. Как ввести данные с клавиатуры и вывести данные в консоль?
3. Что представляют собой аргументы командной строки?
4. Каковы функции аргументов командной строки?
5. Опишите модули minimist и yargs.

Лабораторная работа № 2. ФАЙЛЫ NODE.JS

Цель работы: работа с файлами и папками с помощью Node.js – создание, удаление и редактирование файлов.

Формируемые компетенции: ПК 9.2, ПК 9.4, ПК 9.5.

Общие сведения

Работа с файлами Node.js

Для работы с файлами в Node.js существует модуль fs. Рассмотрим подробнее возможности работы с файлом.

1. Чтение данных из файла

В одной папке с файлом приложения app.js находится текстовый файл text.txt.

Листинг кода 2.1

```
Привет, мир!!
```

Для чтения файла в синхронном виде применяется функция fs.readFileSync(). Ниже представлен вызов этой функции.

Листинг кода 2.2

```
const fs = require('fs');
let textFile = fs.readFileSync("text.txt", "utf8");
console.log(textFile)
```

В метод передаются два параметра:

1) путь к файлу относительно файла приложения app.js;

2) кодировка для получения текстового содержимого файла.

Результат выполнения этого метода – текст файла.

Запустите код с помощью команды node app.ja. Результат работы представлен на рис. 2.1.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node app.js
Привет, мир!!
```

Рис. 2.1. Результат работы команды «node app.js»

Для асинхронного чтения файла используется функция fs.readFile. Ниже представлен вызов этой функции.

Листинг кода 2.3

```
    fs.readFile("______", "_____", function(error,data){
});
```

В метод передаются три параметра:

1) путь к файлу;

2) кодировка для получения текстового содержимого файла;

3) функция обратного вызова, которая выполняется после завершения чтения с параметрами. Функция хранит:

– информацию об ошибке (при ее наличии);

– собственно считанные данные.

Измените код файла app.js для дальнейшей работы.

Листинг кода 2.4

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

    // асинхронное чтение
    fs.readFile('text.txt', 'utf8', function (err, data)
{
        console.log('Асинхронное чтение файла');
        if (err) throw error; // если возникла ошибка,
        console.log(data); // выводим считанные данные
    });

    // синхронное чтение
    console.log('Синхронное чтение файла');
    let fileContent = fs.readFileSync('text.txt',
'utf8');
    console.log(fileContent);
```

Результат работы представлен на рис. 2.2.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node app.js
Синхронное чтение файла
Привет, мир!!
Асинхронное чтение файла
Привет, мир!!
```

Рис. 2.2. Результат запуска кода (листинг 2.4)

Функция fs.readFile() вызывается первой, но так как она асинхронная, то не блокирует поток выполнения, поэтому ее результат выводится в самом конце.

2. Запись файла

Для записи файла можно использовать синхронную функцию fs.writeFileSync(), которая в качестве параметра принимает путь к файлу и записываемые данные.

Листинг кода 2.5

```
fs.writeFileSync('text.txt', 'Привет, мир!!!');
```

Также для записи файла можно использовать асинхронную функцию fs.writeFile(), которая принимает те же параметры.

Листинг кода 2.6

```
fs.writeFile('text.txt', 'Привет, МИР-2023!!!');
```

В качестве вспомогательного параметра в функцию может передаваться функция обратного вызова, которая выполняется после завершения записи.

Листинг кода 2.7

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

fs.writeFile('text.txt', Привет, МИР-2023!', function (err) {
    if (err) throw error;
    console.log('Асинхронная запись файла завершена. Содержимое файла:');
    let data = fs.readFileSync('text.txt', 'utf8');
    console.log(data); // выводим считанные данные
});
```

Результат работы представлен на рис. 2.3.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node app.js
Асинхронная запись файла завершена. Содержимое файла:
Привет, МИР-2023!
```

Рис. 2.3. Результат запуска кода (листинг 2.7)

Следует отметить, что указанные выше методы (см. листинг кода 2.5 – 2.6) полностью перезаписывают файл. Если нужно дозаписать файл, то применяются методы fs.appendFile()/fs.appendFileSync().

Листинг кода 2.8

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

    fs.appendFileSync('text.txt', 'Привет, мир!!!');

    fs.appendFile('text.txt', 'Привет, МИР!!!', function
(err) {
        if (err) throw error; // если возникла ошибка

        console.log('Запись файла завершена. Содержимое
файла:');
        let data = fs.readFileSync('text.txt', 'utf8');
        console.log(data); // выводим считанные данные
    });
```

Результат работы отражает рис. 2.4.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node app.js
Запись файла завершена. Содержимое файла:
Привет, МИР-2023!Привет, мир!!!Привет, МИР!!!
```

Рис. 2.4. Результат запуска кода (листинг 2.8)

3. Удаление файла

Для удаления файла синхронным методом используется функция fs.unlinkSync(), которая в качестве параметра принимает путь к удаляемому файлу.

Листинг кода 2.9

```
    fs.unlinkSync('text.txt');
```

Асинхронная функция fs.unlink() в качестве параметра принимает путь к файлу и функцию, вызываемую после того, как файл удален.

Листинг кода 2.10

```
fs.unlink('text.txt', (err) => {
    if (err) console.log(err);
    // если возникла ошибка
    else console.log('hello.txt удален');
});
```

Результат работы представлен на рис. 2.5.

```
PS D:\!!!DOCUMENTs\Desktop\проект> node app.js
hello.txt удален
```

Рис. 2.5. Результат запуска кода (листинг 2.10)

Простейшая программа для работы с файлами

Node.js использует модульную систему, т. е. встроенная функциональность разбита на отдельные пакеты, или модули. Модуль представляет собой блок кода, который может использоваться повторно в других модулях.

Добавьте строки в новый файл .js, названный app.js.

Листинг кода 2.11

```
const fs = require("fs").promises;
const fsConstants = require("fs").constants;
const path = require("path");
```

Полученные модули необходимы для работы приложения. Это встроенные модули, для загрузки которых используется функция require().

Создайте константы, которые будут хранить имена файла и папки.

Листинг кода 2.12

```
const folderName = "testdir";
const fileName = "testfile.txt";
```

Необходимо найти пути папки и текстового файла. Здесь __dirname – кодовое имя (константа) для обозначения текущей папки, path.join «собирает» путь из фрагментов, обозначенных в скобках.

Листинг кода 2.13

```
    const folderPath = path.join(__dirname, folderName);
    const filePath = path.join(__dirname, folderName, fil
eName);
```

По заданному пути программа ищет папку, если ее нет, то она создается. Затем необходимо найти текстовый файл, если его нет, то он также создается. В существующий (или только созданный) файл записывается текстовая строка.

Листинг кода 2.14

```
    const checkWriteAccess = async (entityPath) => {
        try {
            await fs.access(entityPath, fsConstants.F_OK)
;
            return true;
        } catch (err) {
            return false
        }
    }
    (async () => {
        const folderExists = await checkWriteAccess(folde
rName)
        if (!folderExists) {
            await fs.mkdir(folderPath);
        }
        const fileExists = await checkWriteAccess(filePat
h);
        if (!fileExists) {
            await fs.writeFile(filePath, "TIME LOG\n", {
encoding: 'utf8', flag: 'w+' });
        }
        await fs.appendFile(filePath, `A record has been
added at ${new Date()}.\n`);
    })()
```

В итоге должны быть созданы папка testdir и файл testfile.txt, в который будет записана строка A record has been added at ${new Date()}, где ${new Date} – текущая дата. Подробнее про модули работы с файловой системой (fs) можно узнать здесь: https://nodejs.org/api/fs.html.

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Создайте программу, которая позволит:

– создать несколько файлов с разным разрешением;

– записать в некоторые из них любые данные;

– удалить файлы определенного разрешения.

3. Составьте отчет по результатам работы.

Контрольные вопросы

1. Какие методы для работы с файлами Node.js вы знаете?
2. Какие методы для работы с папками Node.js существуют?
3. Как вывести дату на экран?
4. Что такое объект Date()?
5. Чем отличается синхронная работа с файлами от асинхронной?

Лабораторная работа № 3. HTTP СЕРВЕР NODE.JS

Цель работы: создание простейшего HTTP сервера с помощью модуля HTTP, получение параметров GET- и POST-методов, создание маршрутизации.

Формируемые компетенции: ПК 9.2, ПК 9.4, ПК 9.5.

Общие сведения

При просмотре веб-страницы в браузере отправляется запрос на веб-сервер, который в ответ отправляет веб-страницу. Веб-сервер – это компьютер, с которым происходит взаимодействие через Интернет. Веб-сервер получает запросы от клиентов и отправляет им ответы. Исходя из этого программное обеспечение может быть клиентским или серверным. Клиентское программное обеспечение ответственно за вывод контента, например за цвета текста и стили карточки. Серверное программное обеспечение отвечает за вывод, добавление и хранение данных, серверный код – за обработку запросов от веб-браузера и взаимодействие с базой данных.

Создание базового HTTP сервера

Для работы с сервером и протоколом HTTP в Node.js используется модуль HTTP.

Для того чтобы создать сервер, следует вызвать метод http.createServer().

Листинг кода 3.1

```
const HTTP = require("http");
HTTP.createServer().listen(3001);
```

Метод createServer() возвращает объект http.Server. Для прослушивания и обработки входящих подключений используется метод listen() с параметром «номер порта», по которому запускается сервер.

Для обработки подключений в метод createServer можно передать специальную функцию.

Листинг кода 3.2

```
const HTTP = require("http");
HTTP.createServer(function(request, response){
    response.end("Hello student!");
}).listen(3001);
```

Указанная функция передает два параметра:

– request: хранит информацию о запросе;

– response: управляет отправкой ответа.

Параметр request позволяет получить информацию о запросе и является объектом метода протокола http.IncomingMessage.

Свойства:

– headers: возвращает заголовки запроса;

– method: тип запроса (GET, POST, DELETE, PUT);

– url: предоставляет запрошенный адрес.

Код файла server.js представлен ниже.

Листинг кода 3.3

```
var http = require("http");
http.createServer(function(request, response){
    console.log("Url: " + request.url);
    console.log("Тип запроса: " + request.method);
    console.log("User-Agent:      "      +      re-
quest.headers["user-agent"]);
    console.log("Все заголовки");
    console.log(request.headers);
    response.end();
}).listen(3000);
```

Листинг кода 3.4

```
node server.js
```

Откройте страницу в браузере по адресу http://localhost:3000/index.html.

Результат представлен на рис. 3.1.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS                                    node

Url: /index.html:
Тип запроса: GET
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36 Edg/117.0.2045.31
Все заголовки
{
  host: 'localhost:3000',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'sec-ch-ua': '"Microsoft Edge";v="117", "Not;A=Brand";v="8", "Chromium";v="117"',
  'sec-ch-ua-mobile': '?0',
  'sec-ch-ua-platform': '"Windows"',
  'upgrade-insecure-requests': '1',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36 Edg/117.0.2045.31',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
  'sec-fetch-site': 'none',
  'sec-fetch-mode': 'navigate',
  'sec-fetch-user': '?1',
  'sec-fetch-dest': 'document',
  'accept-encoding': 'gzip, deflate, br',
  'accept-language': 'ru,en;q=0.9,en-GB;q=0.8,en-US;q=0.7'
}
Url: /favicon.ico
Тип запроса: GET
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36 Edg/117.0.2045.31
Все заголовки
{
  host: 'localhost:3000',
  connection: 'keep-alive',
  'sec-ch-ua': '"Microsoft Edge";v="117", "Not;A=Brand";v="8", "Chromium";v="117"',
  'sec-ch-ua-mobile': '?0',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36 Edg/117.0.2045.31',
  'sec-ch-ua-platform': '"Windows"',
  accept: 'image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8',
  'sec-fetch-site': 'same-origin',
  'sec-fetch-mode': 'no-cors',
  'sec-fetch-dest': 'image',
  referer: 'http://localhost:3000/index.html:',
  'accept-encoding': 'gzip, deflate, br',
  'accept-language': 'ru,en;q=0.9,en-GB;q=0.8,en-US;q=0.7'
}
```

Рис. 3.1. Результат работы программы (листинг 3.3)

Параметр response управляет отправкой ответа и представляет собой объект метода протокола http.ServerResponse. Методы:

– statusCode – устанавливает статусный код ответа;

– statusMessage – устанавливает сообщение, отправляемое вместе со статусным кодом;

– setHeader(name, value) – добавляет в ответ один заголовок;

– write – пишет в поток ответа некоторое содержимое;

– writeHead – добавляет в ответ статусный код и набор заголовков;

– end – сигнализирует серверу, что заголовки и тело ответа установлены, в итоге ответ отсылается клиенту. Метод должен вызываться в каждом запросе.

Например, изменим файл server.js следующим образом.

Листинг кода 3.5

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("UserId", 12);
    response.setHeader("Content-Type",     "text/html; charset=utf-8;");
    response.write("<h2>hello world</h2>");
    response.end();
}).listen(3000);
```

Результат работы представлен на рис. 3.2.

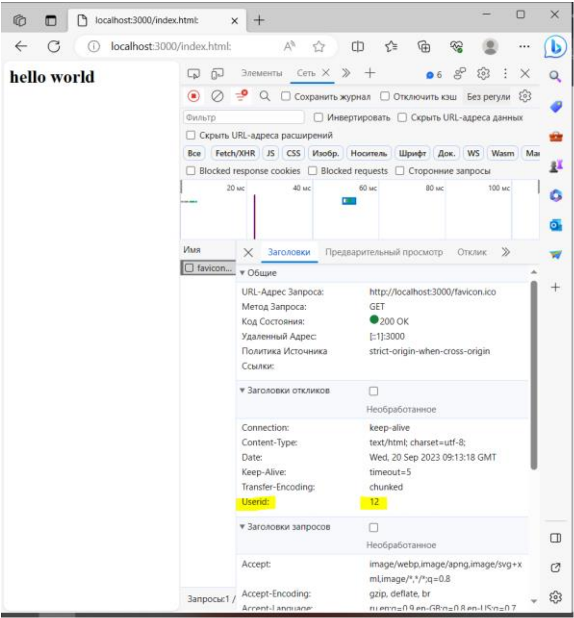


Рис. 3.2. Результат работы программы (листинг 3.5)

Если предстоит отправить довольно большой ответ, то можно несколько раз вызвать метод write(), последовательно отправляя в исходящий поток фрагменты информации. Например, отправим код более-менее полноценной веб-страницы [1].

Листинг кода 3.6

```
const http = require("http");

http.createServer(function(request, response){

    response.setHeader("Content-Type", "text/html");
    response.write("<!DOCTYPE html>");
    response.write("<html>");
    response.write("<head>");
    response.write("<title>Hello Node.js</title>");
    response.write("<meta charset=\"utf-8\" />");
    response.write("</head>");
                     response.write("<body><h2>Привет
мир!!</h2></body>");
    response.write("</html>");
    response.end();
}).listen(3000);
```

Результат работы представлен на рис. 3.3.

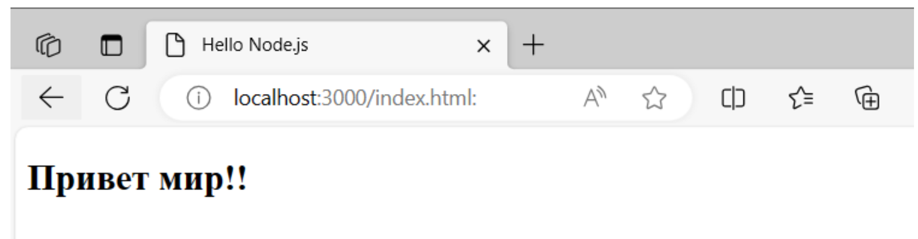


Рис. 3.3. Результат работы программы (листинг 3.6)

POST- и GET-запросы

GET – метод для чтения данных, его используют, например, для доступа к указанной странице. Он сообщает серверу, что клиент запрашивает данные.

POST – метод для отправки данных, чаще всего с его помощью передаются формы.

Протокол HTTP очень прост и состоит из двух частей – заголовков и тела запроса или ответа.

Тело запроса – это информация, которую передал браузер при запросе страницы. Но тело запроса присутствует, только если браузер запросил страницу методом POST. Например, если отправлена форма, то телом запроса будет содержание формы.

Для того чтобы считать, какой запрос был передан, используется конструкция request.method.

Для отображения параметров в GET-запросе рекомендуется модуль url.

Листинг кода 3.7

```
    const http = require("http");
    const url = require("url");

    http.createServer(function (request, response) {
        response.setHeader("Content-Type",    "text/html;
charset=utf-8;");

            const URL= url.parse(request.url, true)
            console.log(URL.query)
    response.end('ok')
    }).listen(3000);
```

Проверьте работу кода в Postman. На рис. 3.4 представлен запрос с параметрами.

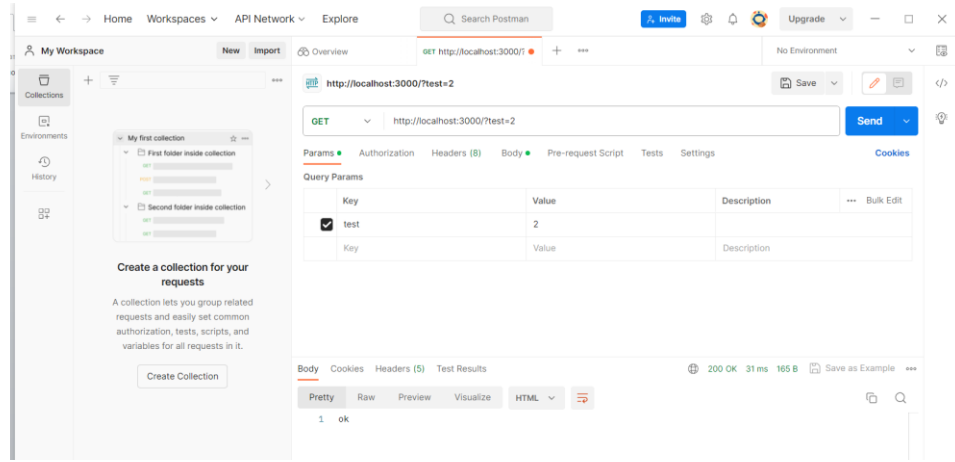


Рис. 3.4. Результат работы программы (листинг 3.7) в Postman

Результат работы представлен на рис. 3.5.

```
[Object: null prototype] { test: '2' }
```

Рис. 3.5. Результат работы программы (листинг 3.7) в терминале

Обратимся к параметру test. Для отображения параметров в POST-запросе используется модуль querystring.

Листинг кода 3.8

```
const http = require("http");

const { parse } = require('querystring')


http.createServer(function (request, response) {
        response.setHeader("Content-Type",  "text/html;
charset=utf-8;");

            let body = '';

            request.on('data', chunk => {
                body=chunk.toString()
              console.log(body);
            })
```

```
                request.on('end', () => {
                    let params = parse(body)

                    console.log(params);
                });
            response.end("ok");
    }).listen(3000);
```

Проверьте работу кода в Postman. На рис. 3.6 представлен запрос с параметрами.

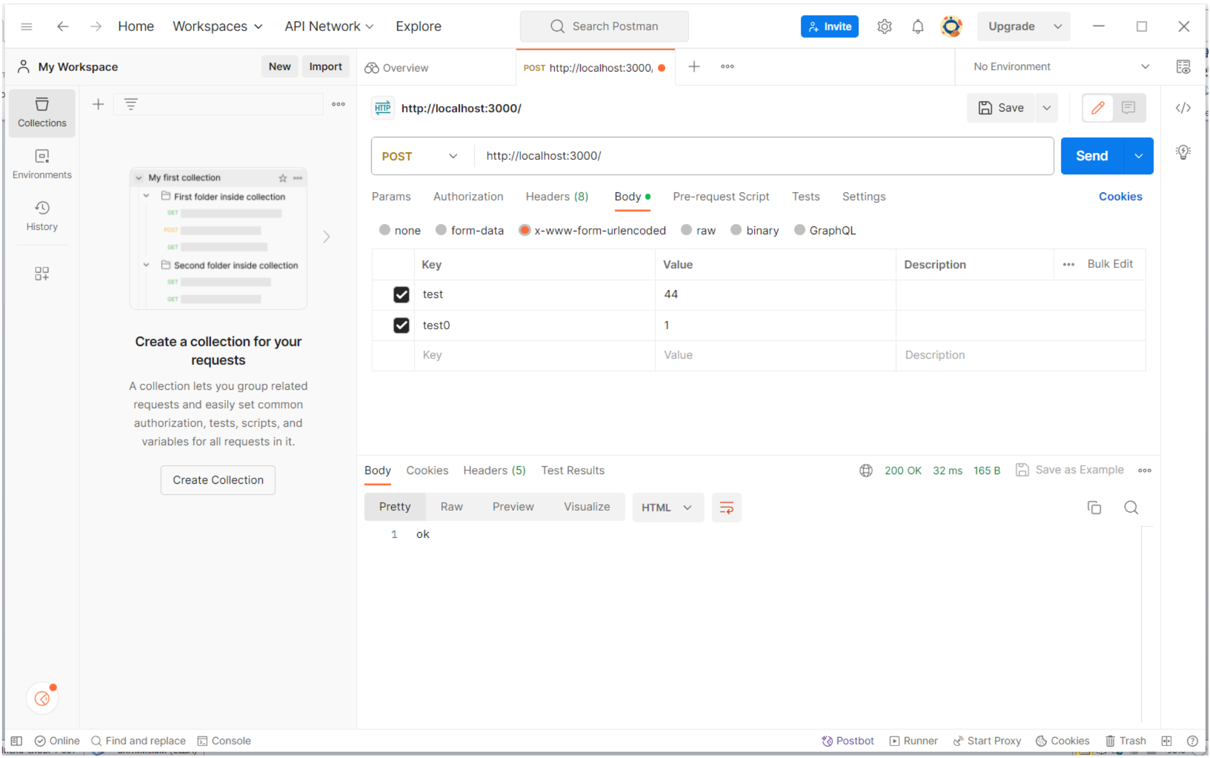


Рис. 3.6. Результат работы программы (листинг 3.8) в программе Postman

Результат работы представлен на рис. 3.7.

```
[Object: null prototype] { test: '44', test0: '1' }
```

Рис. 3.7. Результат работы программы (листинг 3.8) в терминале

Параметры test и test0 позволяют объединить два кода и поработать с условиями.

Листинг кода 3.9

```
const http = require("http");
const url = require("url");
const { parse } = require('querystring')


http.createServer(function (request, response) {
        response.setHeader("Content-Type",  "text/html;
charset=utf-8;");

    if (request.method === 'POST') {
        let body = '';

        request.on('data', chunk => {
            body=chunk.toString()

        })
        request.on('end', () => {
            let params = parse(body)

            console.log(params);
        });
    }
    if (request.method === 'GET') {
        const URL= url.parse(request.url, true)
        console.log(URL.query)


    }
    response.end("ok");
}).listen(3000);
```

Маршрутизация

По умолчанию Node.js не имеет встроенной системы маршрутизации. Обычно она реализуется с помощью специальных фреймворков типа Express.js. Однако если необходимо разграничить простейшую обработку двух-трех маршрутов, то для этого вполне можно использовать свойство url объекта Request [1].

Листинг кода 3.10

```
    const http = require("http");

    http.createServer(function(request, response){

            response.setHeader("Content-Type",  "text/html;
charset=utf-8;");

           if(request.url  ===  "/home"  ||  request.url  ===
"/"){
               response.write("<h2>Главная страница</h2>");
          }
          else if(request.url == "/about"){
               response.write("<h2>О нас</h2>");
          }
          else if(request.url == "/contact"){
               response.write("<h2>Контакты</h2>");
          }
          else{
               response.write("<h2>Нет страницы</h2>");
          }
          response.end();
    }).listen(3000);
```

Например, нужно обработать три маршрута. Запустите сервер. Если пользователь обращается к корневому адресу сайта или по адресу localhost:3000/home, то выводится строка «Главная страница», если по адресу localhost:3000/about, то в браузере отображается строка «О нас», и т. д. Если запрошенный адрес не соответствует ни одному маршруту, то выводится «Нет страницы».

Результат выполнения программы (листинг 3.10) представлен на рис. 3.8.

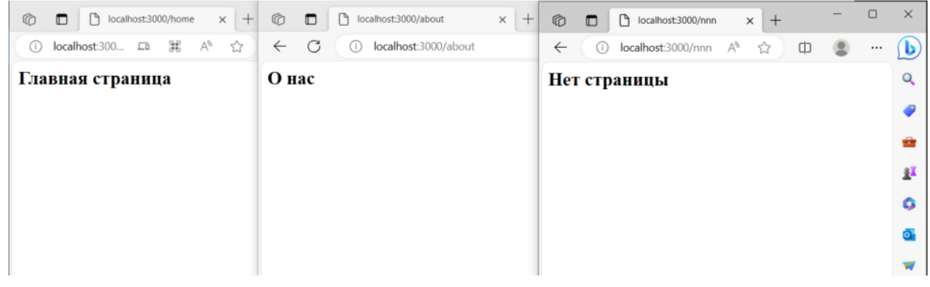


Рис. 3.8. Результат выполнения программы (листинг 3.10)

В рамках специальных фреймворков, которые работают поверх Node.js, например Express.js, есть более удобные способы для обработки маршрутов.

Переадресация

Переадресация предполагает отправку статусного кода 301 (постоянная переадресация) или 302 (временная переадресация) и заголовка Location, который указывает на новый адрес. Выполним переадресацию с адреса localhost:3000/ на адрес localhost:3000/newpage [1].

Листинг кода 3.11

```
 const http = require("http");

http.createServer(function(request, response){

    response.setHeader("Content-Type",     "text/html;
charset=utf-8;");

    if(request.url === "/"){
        response.statusCode  =  302;  //  временная
переадресация
        // на адрес localhost:3000/newpage
        response.setHeader("Location", "/newpage");
    }
    else if(request.url == "/newpage"){
        response.write("Новый адрес");
    }
    else{
        response.statusCode = 404; // адрес не найден
        response.write("Нет страницы");
    }
    response.end();
}).listen(3000);
```

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Создайте HTTP сервер, который решает следующие задачи:

– работа с GET- и POST-параметрами;

– маршрутизация;

– переадресация;

– вывод на каждой странице вашего ФИО и номера группы.

3. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое HTTP?
2. Для чего нужен модуль HTTP?
3. Какие методы в модуле HTTP вам известны?
4. Как обработать GET-запрос с параметрами?
5. Как обработать POST-запрос с данными формы?

Лабораторная работа № 4. HTTPS СЕРВЕР NODE.JS

Цель работы: создание простейшего HTTPS сервера с помощью модуля HTTP, получение GET- и POST-параметров, создание маршрутизации HTTPS сертификатов, а также подключение HTTPS к HTTP серверу Node.js.

Формируемые компетенции: ПК 9.2, ПК 9.4, ПК 9.5.

Общие сведения

HTTPS (от англ. HyperText Transfer Protocol Secure) – это безопасный протокол передачи данных, который поддерживает шифрование посредством криптографических протоколов SSL и TLS и является расширенной версией протокола HTTP.

Миллионы интернет-пользователей постоянно обмениваются информацией. Это могут быть дружеские беседы, весёлые картинки, рабочая переписка, банковские и паспортные данные, номера договоров и другая конфиденциальная информация. Работа Всемирной паутины основана на протоколе HTTP, благодаря которому пользователи могут передавать данные.

Изначально HTTP (HyperText Transfer Protocol) использовался только как протокол передачи гипертекста (текста с перекрёстными ссылками). Однако позже стало понятно, что он отлично подходит для передачи данных между пользователями. Протокол был доработан для новых задач и стал использоваться повсеместно.

Несмотря на свою функциональность, у HTTP есть один очень важный недостаток – незащищённость: данные между пользователями передаются в открытом виде. Злоумышленники могут вмешаться в передачу данных, перехватить их или изменить. Для того чтобы защитить данные пользователей, был создан протокол HTTPS [2].

HTTPS работает благодаря SSL/TLS-сертификату – цифровой подписи сайта. С её помощью подтверждается его подлинность. Перед тем как установить защищённое соединение, браузер запрашивает этот документ и обращается к центру сертификации, чтобы подтвердить его легальность. Если он действителен, то браузер считает этот сайт безопасным и начинает обмен данными.

Для создания сервера необходимо создать самоподписанные сертификаты. Для этого используется утилита OpenSSL:

1. Создайте файл закрытого ключа с помощью команды.

Листинг кода 4.1

```
openssl genrsa -out ryans-key.pem 2048
```

2. Создайте файл запроса на подписание сертификата с помощью команды. Введите запрашиваемые данные.

Листинг кода 4.2

```
openssl req -new -sha256 -key ryans-key.pem -out
ryans-csr.pem
```

3. Создайте самоподписанный сертификат.

Листинг кода 4.3

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-
key.pem -out ryans-cert.pem
```

Теперь самоподписанный сертификат можно подключить к HTTPS серверу.

Листинг кода 4.4

```
const https = require('https');
const fs = require('fs');

const options = {
    key: fs.readFileSync('./ryans-key.pem'),
    cert: fs.readFileSync('./ryans-cert.pem'),
};

https.createServer(options, function (req, res) {
    //   код сервера
    })
.listen(3000);
```

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Создайте самоподписанный сертификат.

3. Создайте HTTPS сервер, который решает следующие задачи:

– работа с GET- и POST-параметрами;

– маршрутизация (см. лабораторную работу № 3);

– переадресация (см. лабораторную работу № 3);

– вывод на каждой странице вашего ФИО и номера группы (см. лабораторную работу № 3).

4. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое HTTPS?

2. Чем отличается HTTP от HTTPS?

3. Что представляет собой самоподписанный сертификат?

4. Каковы этапы создания самоподписанных сертификатов?

5. Что такое маршрутизация и переадресация?

Лабораторная работа № 5. ФРЕЙМВОРК EXPRESS.JS

Цель работы: создание HTTP сервера с помощью Express.js, добавление модульных обработчиков Express.js, обработка ошибок Express.js, обработка GET- и POST-запросов.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Node.js позволяет разработчикам создавать бэкэнд сервер на JavaScript.

Один из самых популярных фреймворков для разработки бэкэнда – Express.js. Его установка не занимает много времени: пользователь генерирует приложение с помощью команды npm init, далее указывает команду для добавления Express.js в приложение.

Листинг кода 5.1

```
npm install express.
```

Создайте основной файл index.js и добавьте в него код.

Листинг кода 5.2

```
const express = require('express'), http = require('h
ttp');

const app = express();
const server = http.createServer(app);
app.use("/", (req, res) => {
    res.send("Приложение работает")
})
const port = 8080
server.listen(port, () => {
    console.log('Server listening on port: ', port);
})
```

Рассмотрим методы для работы с сервером, написанном с помощью фреймворка Express.js.

Методы:

1. http.createServer – инициализация веб-сервера.

2. app.use(…) – добавление эндпоинтов, которые будут обрабатывать запросы.

3. server.listen – запуск веб-сервера на указанном порту.

Описывать все эндпоинты в одном файле неудобно. Поэтому необходимо добавить отдельную папку controller, в нее – файл UserController.js.

Листинг кода 5.3

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

class User {
    login;
    email;

    constructor(login, email) {
        this.email = email;
        this.login = login;
    }
}
let users = [new User("Пользователь", "123@vlsu.ru")]
router.get("", (req, res) => {
    res.send(users)
});

module.exports = router;
```

Данный контроллер содержит один GET-запрос, который возвращает список всех пользователей из массива.

Протокол HTTP поддерживает несколько видов запросов, каждый из которых используется для разных целей.

Виды запроса:

1. GET-запрос используется для получения информации. Не может содержать тело запроса.

2. POST-запрос подходит для отправки данных на сервер (например, для создания чего-либо). Может содержать тело запроса.

3. PUT-запрос очень похож на POST. Как правило, используется для обновления информации.

4. DELETE-запрос похож на GET (также не имеет тела), используется для удаления данных.

Измените файл index.js.

Листинг кода 5.4

```
    const express = require('express'), http = require('h
ttp');
    const usersController = require("./controller/UserCon
troller");

    const app = express();
    const server = http.createServer(app);

    app.use("/users", usersController);

    app.use("",(req, res) => {
        res.send("Работает")
    })
    const port = 8080
    server.listen(port, () => {
        console.log('Server listening on port: ', port);
    })
```

Импортируйте в файл index.js контроллер. Если перейти по ссылке http://localhost:8080/users/, то выведется список пользователей. Результат работы представлен на рис. 5.1.

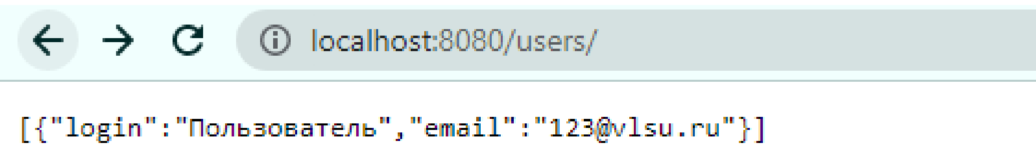


Рис. 5.1. Результат работы программы (листинг 5.4)

При настройке эндпоинта app.use("",(req, res) в кавычках указывается адрес, а после – запрос (req) и ответ (res). Из запроса можно получить входную информацию, например тело запроса (body).

В контроллер (листинг 5.3) добавьте POST-запрос, затем с помощью Postman отправьте на сервер какое-либо тело.

Тело запроса POST может быть форматом JSON или form-data. Обработайте запрос типа form-data. Для этого необходимо установить multer.

Multer – это промежуточное программное обеспечение Node.js для обработки multipart/form-data, которое в основном используется для загрузки файлов.

Листинг кода 5.5

```
npm i multer
```

Важно добавить поддержку JSON в корневой файл.

Листинг кода 5.6

```
app.use(express.json())
```

В form-data можно передавать не только текст, но и файл. Файл будет храниться в объекте files, а тело – в объекте body.

Добавьте в файл UserController.js пару строк кода.

Листинг кода 5.7

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

var upload  = multer();
router.post("",
upload.any(),
 (req, res) => {

    console.log(req.files)
    console.log(req.body)

    res.send('Все прошло успешно')
})
```

Результат работы представлен на рис. 5.2 – 5.3.

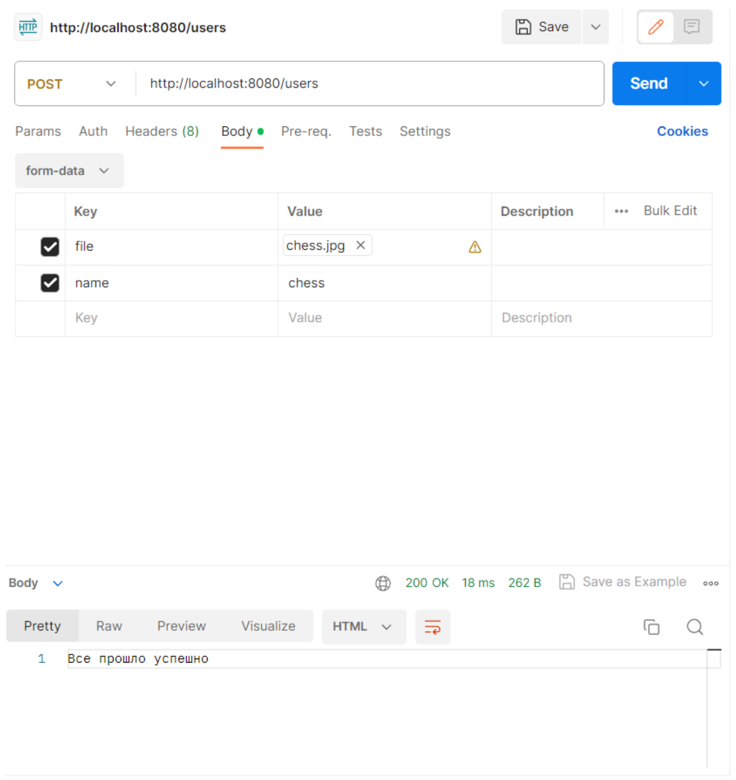


Рис. 5.2. Результат работы программы для запроса POST с form-data (в программе Postman)

```
Server listening on port:  8080
[
  {
    fieldname: 'file',
    originalname: ' chess.jpg',
    encoding: '7bit',
    mimetype: 'image/jpeg',
    buffer: <Buffer ff d8 ff e0 00 10 4a 46 49 46 00 01 01 02 0
0 1c 00 1c 00 00 ff db 00 43 00 04 03 03 04 03 03 04 04 04 0
5 05 04 05 07 0b 07 07 06 06 07 0e 0a 0a 08 ... 158111 more byt
es>,
    size: 158161
  }
]
[Object: null prototype] { name: 'chess' }
```

Рис. 5.3. Результат работы программы для запроса POST с form-data (в терминале)

Теперь можно обработать тип raw (JSON). Код для обработки такого типа приведен ниже.

Листинг кода 5.8

```
router.post("", (req, res) => {
    console.log(req.body);
    res.send(req.body)
})
```

На рис. 5.4 представлен результат работы.

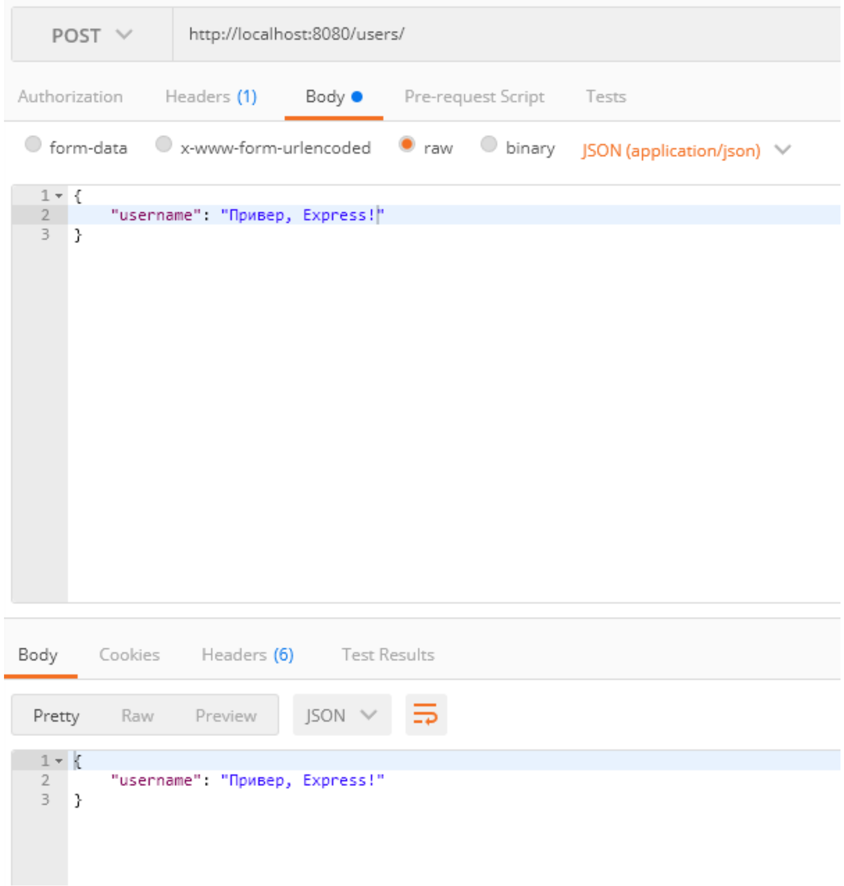


Рис. 5.4. Результат работы программы для запроса POST с raw (JSON) (в программе Postman)

В ответе от сервера содержится отправленное тело запроса. Это значит, что запрос работает.

Таким же образом можно в адресной строке передать некоторые параметры, например вывести данные пользователя по его номеру в массиве. Для того чтобы указать наличие переменной в адресе эндпоинта, используется двоеточие. Получить значение можно с помощью req.params.<имя_переменной>.

Листинг кода 5.9

```
router.get("/:userId", (req, res) => {
    res.send(users[req.params.userId]);
})
```

Результат работы представлен на рис. 5.5.

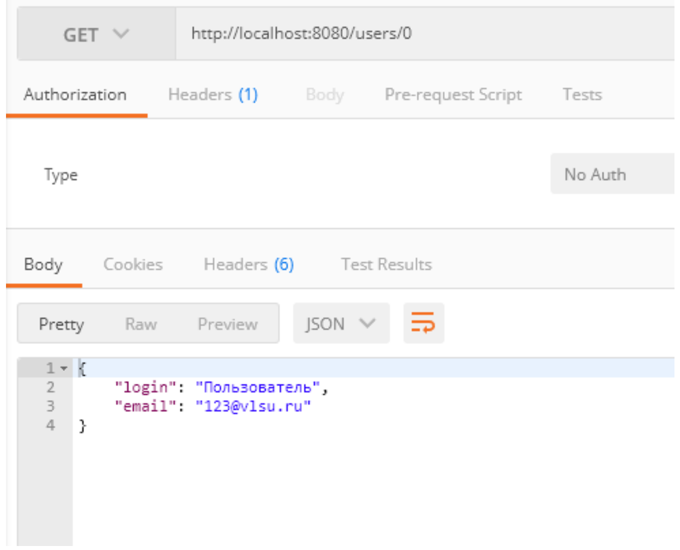


Рис. 5.5. Результат работы программы
для запроса GET с эндпоинтом (в программе Postman)

Для ответа на запрос можно установить статус, например, если пользователь с таким идентификатором не найден.

Листинг кода 5.10

```
router.get("/:userId", (req, res) => {
    id = req.params.userId;
    if (id > users.length-1) {
        res.status("404")
        res.send({message: "User not found"})
    }
    res.send(users[req.params.userId]);
})
```

Также есть возможность добавить сразу несколько контроллеров для обработки запросов. Самое главное – у эндпоинтов должны быть уникальные пути. Если пути будут совпадать – возникают конфликты.

Обязательно в контроллере необходимо вызвать метод res.send() для того, чтобы пользователь получил ответ и запрос выполнился.

Обработка синхронных ошибок

Обработка синхронной ошибки происходит с помощью инструкции throw в обработчике запроса Express.js. Обратите внимание на то, что обработчики запросов ещё называются «контроллерами»,

но предпочтительнее использовать термин «обработчик запросов», так как он понятнее.

Листинг кода 5.11

```
app.post('/test', (req, res) => {
  throw new Error('Ошибка! ')
})
```

Подобные ошибки можно перехватить с помощью обработчика ошибок Express.js. Если пользователь не написал собственного обработчика ошибок, то Express.js обработает ошибку с помощью обработчика, используемого по умолчанию.

Стандартный обработчик ошибок Express.js:

1) устанавливает код состояния HTTP ответа – 500;

2) отправляет текстовый ответ сущности, выполнившей запрос (текстовый ответ);

3) логирует текстовый ответ в консоль.

Обработка асинхронных ошибок

Для обработки асинхронной ошибки нужно отправить ее обработчику ошибок Express.js через аргумент next.

Листинг кода 5.12

```
app.post('/test', async (req, res, next) => {
  return next(new Error('Ошибка!!'))
})
```

Если в приложении Express.js используются конструкции async/await, то понадобится функция-обёртка наподобие express-async-handler. Это позволяет писать асинхронный код без блоков try/catch.

Листинг кода 5.13

```
const asyncHandler = require('express-async-handler')
app.post('/testing', asyncHandler(async (req, res, next) => {
  // Сделать что-нибудь
}))
```

После того как обработчик запроса «обёрнут» в функцию express-async-handler, можно завершить работу с запросом с помощью инструкции throw. Эта ошибка попадёт к обработчику ошибок Express.js [3].

Листинг кода 5.14

```
app.post('/test', asyncHandler(async (req, res, next) => {
    throw new Error('Ошибка! ')
}))
```

Порядок выполнения работы

1. Изучите теоретический материал.

2. Разработайте собственный веб-сервер, в котором будут поддерживаться четыре типа запросов (GET, POST, PUT, DELETE). Задание выполните по варианту (номер варианта соответствует номеру фамилии студента в журнале).

Примечание. Необходимо реализовать возможность получения массива данных с помощью GET-запроса, получения определённого элемента по идентификатору, добавления элемента с помощью POST-запроса, редактирования с помощью PUT-запроса и удаления с помощью DELETE-запроса (всего 5 запросов). Добавьте обработку исключительных случаев (например, если объект не найден). Добавьте файл (см. листинг 5.7).

3. Обработайте ошибки с помощью Express.js.

4. Продемонстрируйте работу веб-сервера с помощью Postman.

5. Составьте отчет по результатам работы.

Варианты для выполнения заданий

1. Веб-сервер для работы с пользователями (добавить пользователя, вывести список пользователей, удалить пользователя, обновить данные пользователя).

2. Веб-сервер для цветочного магазина (добавить позицию, вывести список всех позиций, удалить позиции, обновить данные по товарам).

3. Веб-сервер для магазина электроники (добавить позицию, вывести список всех позиций, удалить позиции, обновить данные по товарам).

4. Веб-сервер для аптеки (добавить позицию, вывести список всех позиций, удалить позиции, обновить данные по товарам).

5. Веб-сервер для школы (добавить ученика, вывести список учеников, удалить ученика, обновить данные об ученике).

6. Веб-сервер для больницы (добавить пациента, вывести список всех пациентов, удалить пациента, обновить данные о пациенте).

7. Веб-сервер для магазина игрушек (добавить позицию, вывести список всех позиций, удалить позиции, обновить данные по товарам).

8. Веб-сервер для детского сада (добавить ребенка, вывести список детей, удалить ребенка, обновить данные о детях).

9. Веб-сервер для продуктового магазина (добавить позицию, вывести список всех позиций, удалить позиции, обновить данные по товарам).

10. Веб-сервер для кинотеатра (добавить фильм, вывести список фильмов, удалить фильм, обновить данные о фильме).

Контрольные вопросы

1. Какие методы для работы с сервером вам известны?
2. Какие виды запросов поддерживает протокол HTTP?
3. Что такое эндпоинты?
4. Что такое Postman?
5. Какими способами можно обработать ошибку в Express?

Лабораторная работа № 6. ШАБЛОНИЗАТОР PUG

Цель работы: создание шаблонов HTML-страниц, создание простейших веб-приложений.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Pug – это популярный шаблонизатор HTML, написанный на языке JavaScript для Node.js. После интерпретации сервером синтаксис pug превращается в HTML-код. Старое название pig--Jade. У разработчика возникли проблемы с авторскими правами, поэтому проект был переименован в pug. Таким образом, если вы встретите упоминание Jade, знайте, что речь идет о pug [4].

Если необходимо создать новый проект, то следует инициализировать его с помощью команды npm init и установить Express.js (команда npm i express).

Листинг кода 6.1

```
npm install pug –save
```

Теперь можно приступить к созданию простейшей HTML-страницы. Создайте файл index.js, папку view с файлом main.pug и добавьте код.

Листинг кода 6.2 (index.js)

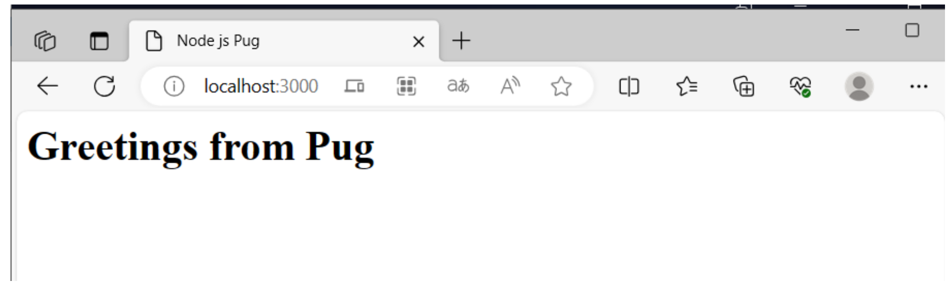
```
const express = require('express');
const app = express();

const host = 'localhost';
const port = 3000;

app.set('views', './view');
app.set('view engine', 'pug');

app.get('/', (req, res) => {
    res.render('main', { title: 'Greetings from Pug' });
});
```

```
     app.listen(port, host, function () {
                          console.log(`Server        listens
http://${host}:${port}`);
     });
```

Листинг кода 6.3 (main.pug)

```
     html(lang="en")
       head
         title Node js Pug
         meta(charset="utf-8")
       body
         h1 #{title}
```

Запустите проект (node app.js) и откройте его в браузере (http://localhost:3000/). Результат работы представлен на рисунке.



Результат работы программы (листинги 6.2 – 6.3)

В Node.js pug-представления имеют расширение .pug и генерируются с помощью метода объекта ответа render(), который имеет два параметра:

1) имя шаблона;
2) данные для этого шаблона в виде объекта.

Шаблонизатор использует необычный подход к построению представления. Каждая строка в файле полностью описывает один HTML-элемент. Сначала идет имя тега, затем через пробел – его значение. Для использования в значении тега (или его атрибута) внешних данных применяется механизм интерполяции. Так, свойство переданного объекта, значение которого необходимо использовать, заключается в конструкцию #{объект}.

Если HTML-тег не указан, то по умолчанию будет использоваться div.

Листинг кода 6.4

```
тег(имя*атрибута='значение*атрибута')
```

Вложенность тегов HTML в Node.js в шаблоне pug реализуется через табуляцию относительно родителя, причем эта вложенность соблюдается в файле и визуально. Для компиляции HTML-кода в одну строку без соблюдения визуальной иерархии используйте следующую запись.

Листинг кода 6.5

```
p: span
//Результат: '<p><span></span></p>'
```

Переменные

Внутри представления возможно определение переменных, которые могут использоваться только в пределах текущего шаблона.

Листинг кода 6.6

```
-var title = 'New greetings from Pug'

html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    h1 #{title}
```

Условия pug

В Node.js шаблонизатор pug для реализации условий использует конструкции, аналогичные операторам if и switch в JavaScript.

Листинг кода 6.7 (index.js)

```
app.get('/', (req, res) => {
    res.render('index', {
        title: 'Greetings from Pug',
```

```
            content: 'Node js Pug description',
        });
    });
```

Листинг кода 6.8 (index.pug)

```
html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    h1 #{title}

    if content
      p #{content}
    else
      p No content
```

Листинг кода 6.9 (index.js)

```
app.get('/', (req, res) => {
    res.render('index', {
        title: 'Greetings from Pug',
        type: 'h3',
    });
});
```

Листинг кода 6.10 (index.pug)

```
html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    case type
      when 'h1'
        h1 #{title}
      when 'h2'
        h2 #{title}
      when 'h3'
        h3 #{title}
```

Циклы pug

Отображение массива данных или вывод какой-либо части шаблона заданное количество раз осуществляется с помощью конструкций each и while.

Листинг кода 6.11

```
html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    ol
      each vl, index in ['One', 'Two', 'Three']
        li #{vl} (#{index})
```

Переиспользование шаблонов

Для переиспользования представления в pug имеется оператор include, с помощью которого в указанное место можно вставить содержимое файла заданного шаблона.

Листинг кода 6.12 (view/index.pug)

```
html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    include includes/_list.pug
```

Листинг кода 6.13 (view/includes/_list.pug)

```
ol
  each vl, index in ['One', 'Two', 'Three']
    li #{vl} (#{index})
```

Если указанный файл не существует, то в HTML-документ значение оператора include будет вставлено обычной строкой.

Наследование¶

Шаблонизатор pug реализует принцип наследования для шаблонов, за которое отвечают операторы block и extends. С помощью block

в представлении описывается какая-либо его часть, которая может быть заменена при наследовании (через extends) шаблона другим шаблоном. В родительском представлении блок может иметь значение по умолчанию, но если дочернее представление имеет собственную реализацию, то блок будет использовать ее.

Листинг кода 6.14 (index.js)

```
app.get('/', (req, res) => {
    res.render('home');
});
```

Листинг кода 6.15 (view/index.pug)

```
html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    block nav
      ul
        li Home
        li About
        li Contacts
    block content
    block footer
```

Листинг кода 6.16 (view/home.pug)

```
extends index.pug
block nav
  ul
    li Home
    li About
    li Contacts
  block content
    div Content text
  block footer
    footer Footer information
```

Также pug позволяет «расширять» значение по умолчанию, а не заменять его. Для этого имеются операторы append и prepend, кото-

рые добавляют указанное содержимое после или до значения, заданного по умолчанию.

Листинг кода 6.17

```
extends index.pug
block prepend nav
  a: img(src="/assets/images/logo.svg" alt="Logo")
block content
  div Content text
block footer
  footer Footer information
```

Миксины

Миксины представляют собой подобие функций, которые могут быть использованы для создания переиспользуемых частей представления. Как и обычные функции, они могут принимать параметры [1].

Листинг кода 6.18 (view/mixins/_button.pug)

```
mixin button(label, cssClass)
  button(class =cssClass) #{label}
```

Листинг кода 6.19 (view/index.pug)

```
include mixins/_button.pug
html(lang="en")
  head
    title Node js Pug
    meta(charset="utf-8")
  body
    +button('Cancel', 'red')
    +button('Send')
```

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Разработайте HTTP-страницы. На странице должны быть реализованы:

– условия;

– циклы;

– наследование;
– переиспользование шаблонов;
– миксины;
– вложенные теги;
– список.
3. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое pug?
2. Каковы особенности синтаксиса pug по сравнению с HTML?
3. Что такое миксины?
4. Как можно переиспользовать шаблоны в pug?
5. Как передать данные в шаблон pug из фреймворка Express.js?

Лабораторная работа № 7. СОЗДАНИЕ И ОБРАБОТКА ФОРМ PUG

Цель работы: создание форм с помощью шаблонизатора pug, обработка форм в Express.js.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Обязательный для изучения теоретический материал по основам использования шаблонизатора pug дан в лабораторной работе № 6.

Этапы создания и обработки форм в pug

Пример простой формы, написанной на шаблонизаторе pug, представлен ниже.

Листинг кода 7.1

```
form(method=GET action='/authorize')
    input(name='login' placeholder='login')
    input(name='password' placeholder='password')
    input(type='submit' value='залогиниться')
или
input(name='caption' placeholder='caption')
textarea(name='description'                placehold-
er='description')
button(class='submit_button') Добавить комментарий
```

Ниже представлен пример передачи параметра в значение строки ввода данных.

Листинг кода 7.2

```
input(type="text" name="date" value=viewpos)
```

Реализуйте форму для добавления пользователей в созданном ранее в рамках лабораторной работы № 6 проекте.

Создайте шаблон формы.

Листинг кода 7.3 (form_users.pug)

```
html(lang="en")
  head
    style
        include style.css
    title Node js Pug
    meta(charset="utf-8")

  body
    form(method=GET action='/reg')
            input(name='name'  type='text'  placehold-
er='name' )
           input(name='login'  type='text'  placehold-
er='login')
          input(name='email'  type='email'  placehold-
er='email')
        input(name='password' tupe='password' place-
holder='password')
      input(type='submit' value=but)
```

Затем добавьте обработку при регистрации.

Листинг кода 7.4 (app.js)

```
const express = require('express');
const app = express();

const host = 'localhost';
const port = 3000;

app.set('views', './view');
app.set('view engine', 'pug');

app.get('/', (req, res) => {
    res.render('form', {viewpost:'kjh'});
});

app.get('/users', (req, res) => {
    res.render('form_users',
{but:'Зарегистрироваться'});
});

app.get('/reg',function(req,res){
```

```
        res.end(`User ${req.query.name} registered!!!`);
    })

    app.listen(port, host, function () {
        console.log(`Server                         listens
http://${host}:${port}`);
    });
```

Результат работы представлен на рис. 7.1 – 7.2.

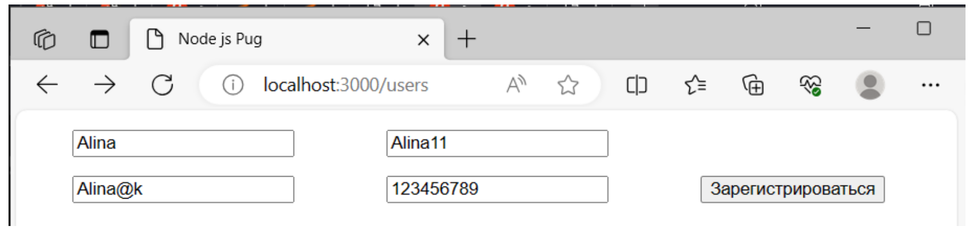


Рис. 7.1. Ввод данных в форму pug

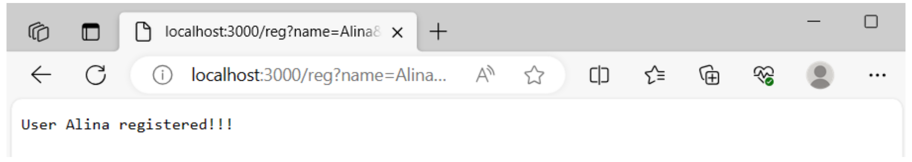


Рис. 7.2. Результат после отправки формы pug

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).
2. Добавьте в лабораторную работу № 6 форму (любую, кроме регистрации).
3. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое pug?
2. Чем отличается синтаксис pug от синтаксиса HTML?
3. Что такое миксины?
4. Как можно переиспользовать шаблоны в pug?
5. Каким образом в pug создаются и обрабатываются формы?

Лабораторная работа № 8. ШАБЛОНИЗАТОР EJS

Цель работы: создание шаблонов HTML-страниц, создание простейших веб-приложений с использованием форм с помощью шаблонизатора ejs, обработка данных форм в Express.js.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Pug используется по умолчанию как механизм просмотра для Express.js, но синтаксис pug может быть чрезмерно сложным. Ejs – один из альтернативных вариантов просмотра данных, удобный и простой в настройке.

Примеры работы шаблонизатора ejs приведены ниже.

Листинг кода 8.1

```
<% if (user) { %>
  <h2><%= user.name %></h2>
<% } %>
```

Листинг кода 8.2

```
let template = ejs.compile(str, options);
template(data);
// => Rendered HTML string

ejs.render(str, data, options);
// => Rendered HTML string

ejs.renderFile(filename, data, options, function(err, str){
    // str => Rendered HTML string
});
```

Опции ejs:

– cache – скомпилированные функции кэшируются, требуется filename.

– filename – используется cache для кэширования ключей и включения.

– root – задает корень проекта для includes с абсолютным путем (например, /file.ejs). Может быть массивом, с его помощью можно добавить опцию включения из нескольких каталогов.

– views – массив путей для использования при разрешении включений с относительными путями.

– context – контекст выполнения функции.

– compileDebug – значение равно false, если не компилируется инструментарий отладки.

– client – возвращает автономную скомпилированную функцию.

– delimiter – символ, используемый в качестве внутреннего разделителя, по умолчанию '%'.

– openDelimiter – символ, используемый для открытия разделителя, по умолчанию '<'.

– closeDelimiter – символ, используемый для закрывающего разделителя, по умолчанию '>'.

– debug – выводит сгенерированное тело функции.

– strict – при значении true сгенерированная функция находится в строгом режиме.

– localsName – имя, используемое для объекта, хранящего локальные переменные, когда не используется with. Значение по умолчанию равно locals.

– rmWhitespace – удаляет все безопасные для удаления пробелы, включая начальные и конечные, а также позволяет использовать более безопасную версию -%> прерывания строки для всех тегов скриптлета (она не удаляет новые строки тегов в середине строки).

– escape – экранирующая функция, используемая с <%= конструкцией. Она используется при рендеринге, метод .toString() редактируется при генерации клиентских функций (по умолчанию функция escape экранирует XML).

– outputFunctionName – устанавливает значение в строку (например, 'echo' или 'print'), чтобы функция печатала выходные данные внутри тегов скриптлета.

– async – параметр true ejs будет использовать функцию async для рендеринга (зависит от поддержки async/await в среде выполнения JS).

Теги:

– <% – тег 'Scriptlet' для потока управления, без вывода.

– <%_ – тег скриптлета ‘Поглощает пробелы’, удаляет все пробелы перед ним.

– <%= – выводит значение в шаблон (экранированный HTML).

– <%– – выводит неэкранированное значение в шаблон.

– <%# – тег комментария, не выполняется, не выводится.

– <%% – выводит литерал '<%'.

– %> – простой конечный тег.

– -%> – тег Trim-mode ('перевод строки'), выполняет обрезку следующей новой строки.

– _%> – завершающий тег ‘Пропускает пробелы’, удаляет все пробелы после него.

Синтаксис добавления частичного шаблона ejs:

1. Используйте <%– include('RELATIVE/PATH/TO/FILE') %>, чтобы встроить частичный шаблон ejs в другой файл.

2. Дефис <%– вместо <% предписывает ejs выполнять рендеринг необработанного кода HTML.

3. Путь к частичному шаблону является относительным по отношению к текущему файлу [5].

Реализация шаблонизатора ejs

Создайте HTML-страницу. Для этого используют библиотеку Express.js. В Express.js очень простая настройка с использованием ejs будет выглядеть следующим образом.

Если вы создаете новой проект, то не забудьте инициализировать его с помощью npm init и установить Express.js (команда npm i express).

Листинг кода 8.3

```
npm install ejs
```

Теперь можно приступить к созданию простейшей HTML-страницы. Создайте файл index.js и добавьте код.

Листинг кода 8.4 (index.js)

```
// load the things we need
var express = require('express');
var app = express();

// set the view engine to ejs
app.set('view engine', 'ejs');

// use res.render to load up an ejs view file

// index page
app.get('/', function(req, res) {
    res.render('pages/index');
});

// about page
app.get('/about', function(req, res) {
    res.render('pages/about');
});

app.listen(8080);
console.log('8080 is the magic port');
```

Создание частичных элементов ejs

При создании частичных элементов ejs используются так называемые частичные шаблоны: head.ejs, header.ejs и footer.ejs.

Создайте папку view, в ней – папки pages и partials. Далее в папке partials создайте файлы footer.ejs, head.ejs и header.ejs с кодом.

Листинг кода 8.5 (footer.ejs)

```
<p class="text-center text-muted">© Copyright 2020
The Awesome People</p>
```

Листинг кода 8.6 (head.ejs)

```
<meta charset="UTF-8">
<title>EJS Is Fun</title>

<!-- CSS (load bootstrap from a CDN) -->
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.2/css/bootstrap.min.css">
<style>
    body { padding-top:50px; }
</style>
```

Листинг кода 8.7 (header.ejs)

```
    <nav class="navbar navbar-expand-lg navbar-light bg-
light">
        <a class="navbar-brand" href="/">EJS Is Fun</a>
        <ul class="navbar-nav mr-auto">
          <li class="nav-item">
            <a class="nav-link" href="/">Home</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="/about">About</a>
          </li>
        </ul>
      </nav>
```

Итак, определены частичные шаблоны. Теперь нужно добавить их в представления. Возьмите файлы index.ejs и about.ejs и используйте синтаксис include для добавления частичных шаблонов.

Для этого в папке pages создайте файлы index.ejs и about.ejs.

Листинг кода 8.8 (index.ejs)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <%- include('../partials/head'); %>
</head>
<body class="container">

<header>
    <%- include('../partials/header'); %>
</header>

<main>
    <div class="jumbotron">
        <h1>This is great</h1>
        <p>Welcome to templating using EJS</p>
    </div>
</main>

<footer>
    <%- include('../partials/footer'); %>
</footer>

</body>
</html>
```

Листинг кода 8.9 (about.ejs)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <%- include('../partials/head'); %>
</head>
<body class="container">

<header>
    <%- include('../partials/header'); %>
</header>

<main>
<div class="row">
    <div class="col-sm-8">
        <div class="jumbotron">
            <h1>This is great</h1>
            <p>Welcome to templating using EJS</p>
        </div>
    </div>

    <div class="col-sm-4">
        <div class="well">
            <h3>Look I'm A Sidebar!</h3>
        </div>
    </div>

</div>
</main>

<footer>
    <%- include('../partials/footer'); %>
</footer>

</body>
</html>
```

Запустите проект (node index.js) и откройте его в браузере (http://localhost:8080/).

Результат работы представлен на рис. 8.1 – 8.2.

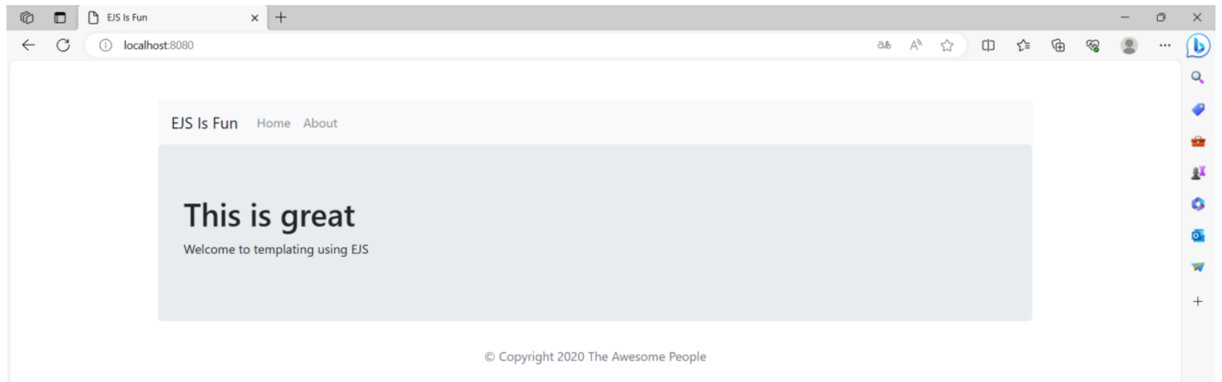


Рис. 8.1. Результат работы сервера с шаблонизатором ejs (страница «Главная»)

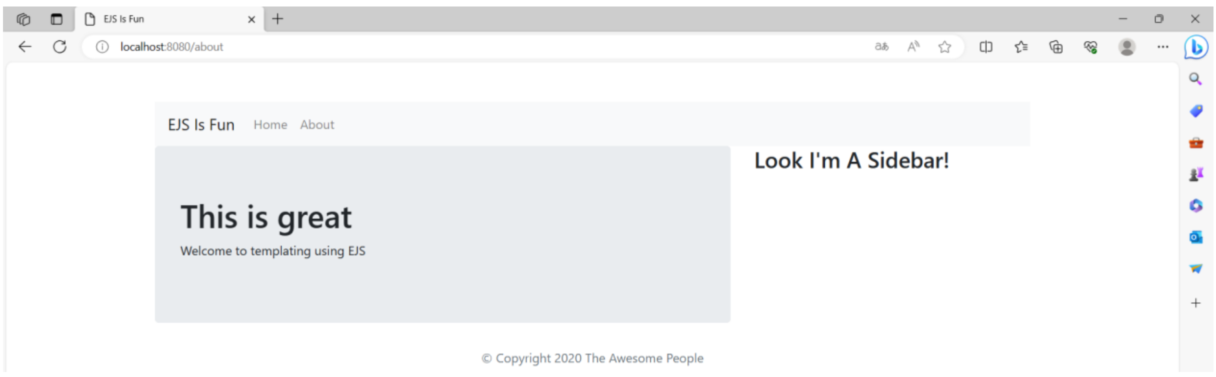


Рис. 8.2. Результат работы сервера с шаблонизатором ejs (страница «О нас»)

Передача данных

Существуют два способа передачи данных:

– в представления;

– непосредственно в частичные шаблоны.

Для первого способа необходимо частично изменить файл index.js.

Определим некоторые базовые переменные и список для передачи на главную страницу. В файл index.js добавьте следующий код в маршрут app.get('/').

Листинг кода 8.10

```
    app.get('/', function(req, res) {
        var mascots = [
            { name: 'Sammy', organization: "Digital-
Ocean", birth_year: 2012},
            { name: 'Tux', organization: "Linux",
birth_year: 1996},
```

```
            { name: 'Moby Dock', organization: "Docker", birth_year: 2013}
        ];
        var tagline = "No programming concept is complete without a cute animal mascot.";

        res.render('pages/index', {
            mascots: mascots,
            tagline: tagline
        });
    })
```

Таким образом, созданы список mascots и простая строка tagline и переданы частичному шаблону. Теперь их можно использовать в файле index.ejs. Перейдите в файл index.ejs и используйте их.

Для того чтобы отразить одну переменную, используйте конструкцию <%= tagline %>. Добавьте код в файл index.ejs.

Листинг кода 8.11

```
    …
      <h2>Variable</h2>
      <p><%= tagline %></p>
    …
```

Для циклического перебора данных используют метод .forEach. Добавьте код в файл index.ejs.

Листинг кода 8.12

```
    …
      <ul>
            <% mascots.forEach(function(mascot) { %>
                <li>
                    <strong><%= mascot.name %></strong>
                     representing <%= mascot.organization %>, born <%= mascot.birth_year %>
                </li>
            <% }); %>
        </ul>
    … [0]
```

Теперь можно запустить проект (node index.js) и открыть его в браузере (http://localhost:8080/).

Результат работы представлен на рис. 8.3.

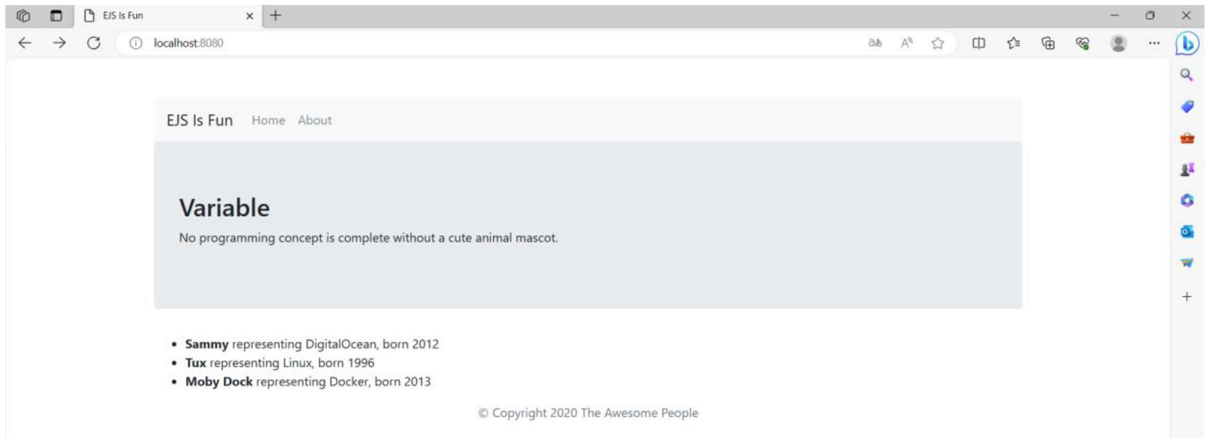


Рис. 8.3. Результат работы сервера с шаблонизатором ejs (страница «Главная» со списком)

Для второго способа передачи данных необходимо частично изменить файлы about.ejs и header.ejs. Частичный шаблон ejs имеет доступ к тем же данным, что и родительское представление. Будьте внимательны: если вы ссылаетесь на переменную в частичном шаблоне, ее нужно определить в каждом представлении, использующем частичный шаблон, иначе приложение выдаст ошибку.

Также можно определять и передавать переменные в частичный шаблон ejs, используя синтаксис include. При этом нужно с осторожностью предполагать, что переменная определена.

Листинг кода 8.13 (about.ejs)

```
    <header>
                <%-  include('../partials/header',  {vari-
ant:'compact'}); %>
        </header>
```

В частичном шаблоне можно использовать ссылку на переменную, которая может быть не определена, и присвоить ей значение по умолчанию следующим образом [6].

Листинг кода 8.14 (header.ejs)

```
        <em>Variant: <%= typeof variant != 'undefined' ? variant : 'default' %></em>
```

В коде выше ejs выполняет рендеринг значения variant, если оно определено, и значения default, если не определено.

Результат работы представлен на рис. 8.4.

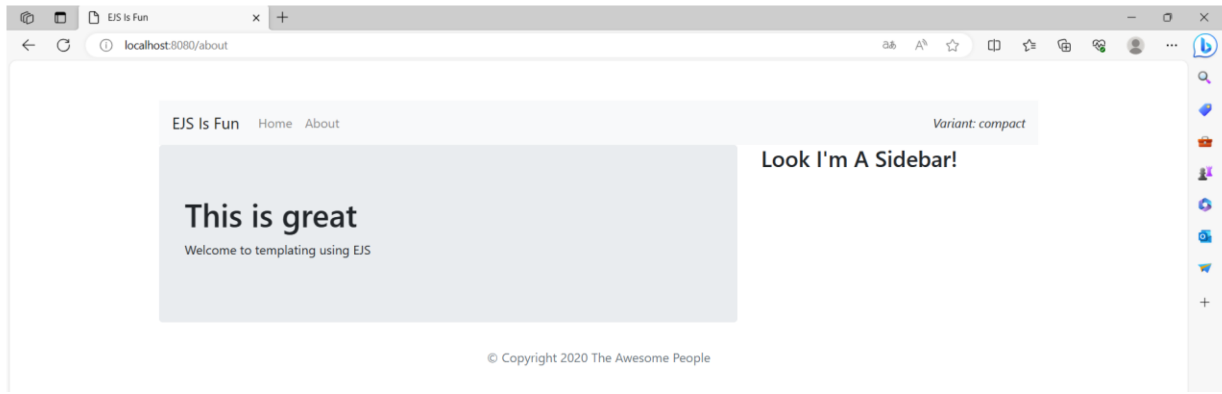


Рис. 8.4. Результат работы сервера с шаблонизатором ejs (страница «О нас» с условным выводом)

Формы

Для работы с формами создайте в папке partials файл form.ejs.

Листинг кода 8.15 (form.ejs)

```
       <form action="/login" method="POST">
        <label for="username">Username:</label>
        <input type="text" name="username" value="<%= username %>">
        <label for="password">Password:</label>
        <input type="password" name="password" value="<%= password %>">
        <button type="submit">Login</button>
      </form>
```

Добавьте форму на главную страницу.

Листинг кода 8.16

```
…
 <%- include('../partials/form'); %>
…
```

Затем немного измените файл index.js.

Листинг кода 8.17

```
// load the things we need
var express = require('express');
var app = express();
app.use(express.json())
app.use(express.urlencoded({extended:false}))

// set the view engine to ejs
app.set('view engine', 'ejs');

// use res.render to load up an ejs view file

// index page
app.get('/', function(req, res) {

    res.render('pages/index', {
         username: 'admin',
         password: '12345'
        });
})
app.post('/login', function(req, res) {

console.log(req.body.username)
        res.end(`User   ${req.body.username}   regis-
tered!!!`);
})

// about page
app.get('/about', function(req, res) {
    res.render('pages/about');
});

app.listen(8080);
console.log('8080 is the magic port');
```

Результат работы представлен на рис. 8.5 – 8.6.

Рис. 8.5. Ввод данных в форму ejs

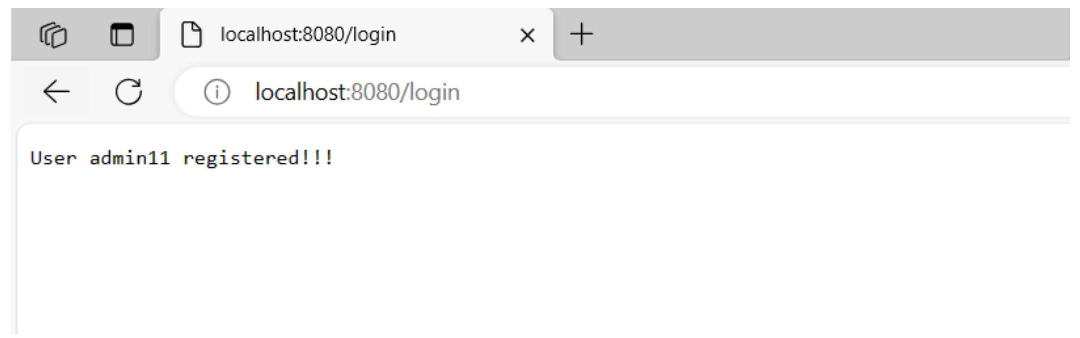


Рис. 8.6. Результат после отправки формы ejs

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).
2. Разработайте HTTP-страницы. На одной из страниц обязательно должна быть форма (любая, кроме формы авторизации).
3. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое ejs?
2. Каковы особенности синтаксиса ejs по сравнению с HTML?
3. Расскажите об основных этапах создания и обработки форм в ejs.
4. Как можно передать данные в ejs?
5. Каким образом используют частичные шаблоны (partials) в ejs?

Лабораторная работа № 9. СРЕДСТВА ДЛЯ РАБОТЫ С БАЗОЙ ДАННЫХ В NODE.JS

Цель работы: работа с базами данных в Node.js, сравнение библиотек pg, pg-promise и knex.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Полноценное веб-приложение не может существовать без базы данных. Однако их взаимодействие происходит с помощью SQL-запросов, написание которых отнимает большую часть времени программиста. Для экономии были созданы различные фреймворки, которые позволяют автоматически генерировать запросы. В лабораторной работе мы сравним несколько решений, такие как pg, pg-promise и knex.

Библиотеки pg, pg-promise и knex – одни из самых популярных инструментов для работы с PostgreSQL в Node.js. Охарактеризуем их более подробно.

Библиотека pg – официальная библиотека для PostgreSQL в языке JavaScript. Она предоставляет прямой доступ к драйверу PostgreSQL и низкоуровневый API для работы с базой данных. Ее основная функция – выполнение SQL-запросов напрямую и возможность получать результаты в виде объектов. Библиотека требует ручного управления подключением к базе данных, выполнением транзакций и обработкой ошибок и позволяет полностью контролировать запросы к базе данных, но может быть менее удобна для разработки.

Библиотека pg-promise – это библиотека-«обертка» над pg, которая предоставляет более удобные и высокоуровневые функции для работы с PostgreSQL. Подобная технология предлагает простой способ работы с базой данных без необходимости ручного управления подключением и транзакциями и обеспечивает безопасное внедрение SQL-запросов и обработку ошибок в удобной форме. Библиотека поддерживает генерацию SQL-запросов с помощью «синтаксического

сахара» и обеспечивает поддержку асинхронных функций и обещаний (promises) для работы с базой данных.

Библиотека knex позволяет избежать прямой работы с SQL-запросами, предоставляя высокоуровневый API для работы с базой данных. Инструмент поддерживает разные драйверы базы данных, в том числе PostgreSQL, и предоставляет возможности миграций базы данных и создания схем таблиц, а также обеспечивает гибкое создание сложных запросов с использованием цепочки методов.

Выбор библиотеки зависит от предпочтений разработчика и потребностей проекта. Если необходим полный контроль над SQL-запросами и базой данных, можно использовать pg или pg-promise. Если нужна более высокоуровневая абстракция и важно удобство в разработке, то лучший вариант – это knex.

Примеры использования библиотек приведены ниже:

а) pg:

Листинг кода 9.1

```
const { Pool } = require('pg');

const pool = new Pool({
  connectionString: 'post-
gresql://username:password@localhost:5432/database',
})

pool.query('SELECT * FROM users', (err, result) => {
  if (err) {
    console.error('Error executing query', err);
    return;
  }

  console.log(result.rows); // Результат запроса в
виде массива объектов
  pool.end(); // Закрытие пула подключений
});
```

б) pg-promise:

Листинг кода 9.2

```
    const pgp = require('pg-promise')();

    const                          db                          =
pgp('postgresql://username:password@localhost:5432/databas
e');

    db.any('SELECT * FROM users')
      .then(data => {
        console.log(data);  //  Результат  запроса  в  виде
массива объектов
      })
      .catch(error => {
        console.error('Error executing query', error);
      })
      .finally(() => {
        pgp.end(); // Закрытие коннектора
      });
```

в) knex:

Листинг кода 9.3

```
    const knex = require('knex')({
      client: 'pg',
      connection: {
        host: 'localhost',
        user: 'username',
        password: 'password',
        database: 'database',
      },
    });

    knex('users')
      .select('*')
      .then(data => {
        console.log(data);  //  Результат  запроса  в  виде
массива объектов
      })
      .catch(error => {
        console.error('Error executing query', error);
      })
      .finally(() => {
        knex.destroy(); // Закрытие соединения
      });
```

Как видно из примеров, каждая библиотека имеет свою синтаксическую особенность и способ взаимодействия с базой данных.

Способы подключения к БД с помощью библиотек knex, pg, pg-promise

Для того чтобы познакомиться с библиотеками, необходимо создать js-проект и установить определенные зависимости:

1) Express.js;
2) knex;
3) pg;
4) pg-promise.

Листинг кода 9.4

```
npm install express knex pg pg-promise
```

Для работы на компьютере должна быть установлена СУБД PostgreSQL (https://www.postgresql.org/download/) и любой интерфейс для работы с ней (стандартный pgAdmin либо DBeaver).

Создайте тестовую БД.

Листинг кода 9.5

```
CREATE TABLE users (
login varchar(255) NULL,
email varchar(255) NULL,
age integer NULL,
is_active boolean NULL
);
```

Заполните её тестовыми данными.

Листинг кода 9.6

```
INSERT INTO users
(login, email, age, is_active)
values
('user1', 'user1@mail.ru', 18, true),
('user2', 'user2@mail.ru', 25, true),
('user3', 'user3@mail.ru', 45, false),
('user4', 'user4@mail.ru', 30, true);
```

На сервере сначала нужно подключиться к БД, затем написать запрос (добавление данных, изменение данных, удаление данных, поиск данных, вывод данных), получить ответ и выполнить какие-либо действия с ответом.

Для того чтобы научиться работать с библиотекой pg, создайте файл server.js в директории проекта и добавьте код.

Листинг кода 9.7

```
const express = require('express');
const {Pool}  = require('pg');

const app = express();

// Конфигурация подключения к базе данных
const pool = new Pool({
    host: '127.0.0.1',
    port: '5432',
    user: 'postgres', //Логин для подключения к БД
    password: 'password', //Пароль
    database: 'database_name' //Название БД

});

// Маршруты вашего сервера
app.get('/', async (req, res) => {
  try {
    // Выполнение запроса к базе данных
    pool.connect((err, client, done) => {
        if (err) throw err;
        client.query('SELECT  *  FROM  users',  (err,
resu) => {
            done();

         // Обработка запроса: если ошибка, то выво-
дим ошибку, если все хорошо - возвращаем ответ в виде мас-
сива данных
            if (err) {
                console.log(err);
            } else{
                res.send(resu.rows)
            }
        })
```

```
    })
  } catch (err) {
    console.error(err);
    res.status(500).send('Ошибка сервера');
  }
});

// Запуск сервера
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`Сервер запущен на порту ${PORT}`);
});
```

Теперь можно протестировать работу сервера (рис. 9.1).

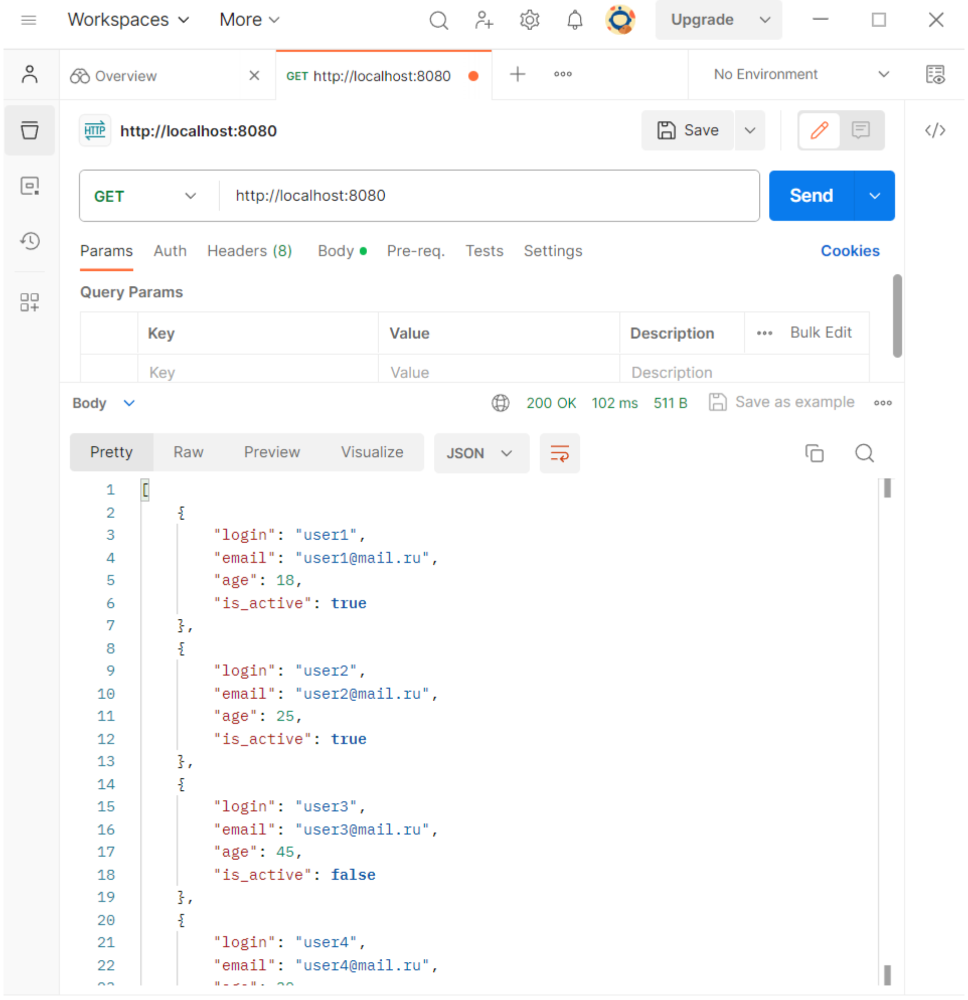


Рис. 9.1. Результат работы с использованием pg

Работа с библиотекой pg-promise начинается с изменения файла server.js.

Листинг кода 9.8

```
    const express = require('express');
    const pgp = require('pg-promise')();
    const app = express();

    // Конфигурация подключения к базе данных
    const db = pgp({
        host: '127.0.0.1',
        port: '5432',
        user: 'postgres', //Логин для подключения к БД
        password: 'password', //Пароль
        database: 'database_name' //Название БД
    })

    // Маршруты вашего сервера
    app.get('/',  (req, res) => {
            // Выполнение запроса к базе данных
        db.any('SELECT * FROM users')
            // Обработка запроса: если ошибка, то выводим
ошибку, если все хорошо - возвращаем ответ в виде массива
данных
            .then(data => {
                res.send(data); // Отправка данных в фор-
мате JSON
            })
            .catch(error => {
                  console.error('Error executing query',
error);
                 res.status(500).json({ error: 'Internal
Server Error' }); // Отправка ошибки в формате JSON
            });
    });

    // Запуск сервера
    const PORT = process.env.PORT || 3000;
    app.listen(PORT, () => {
        console.log(`Сервер запущен на порту ${PORT}`);
    });
```

Протестируйте работу сервера (рис. 9.2).

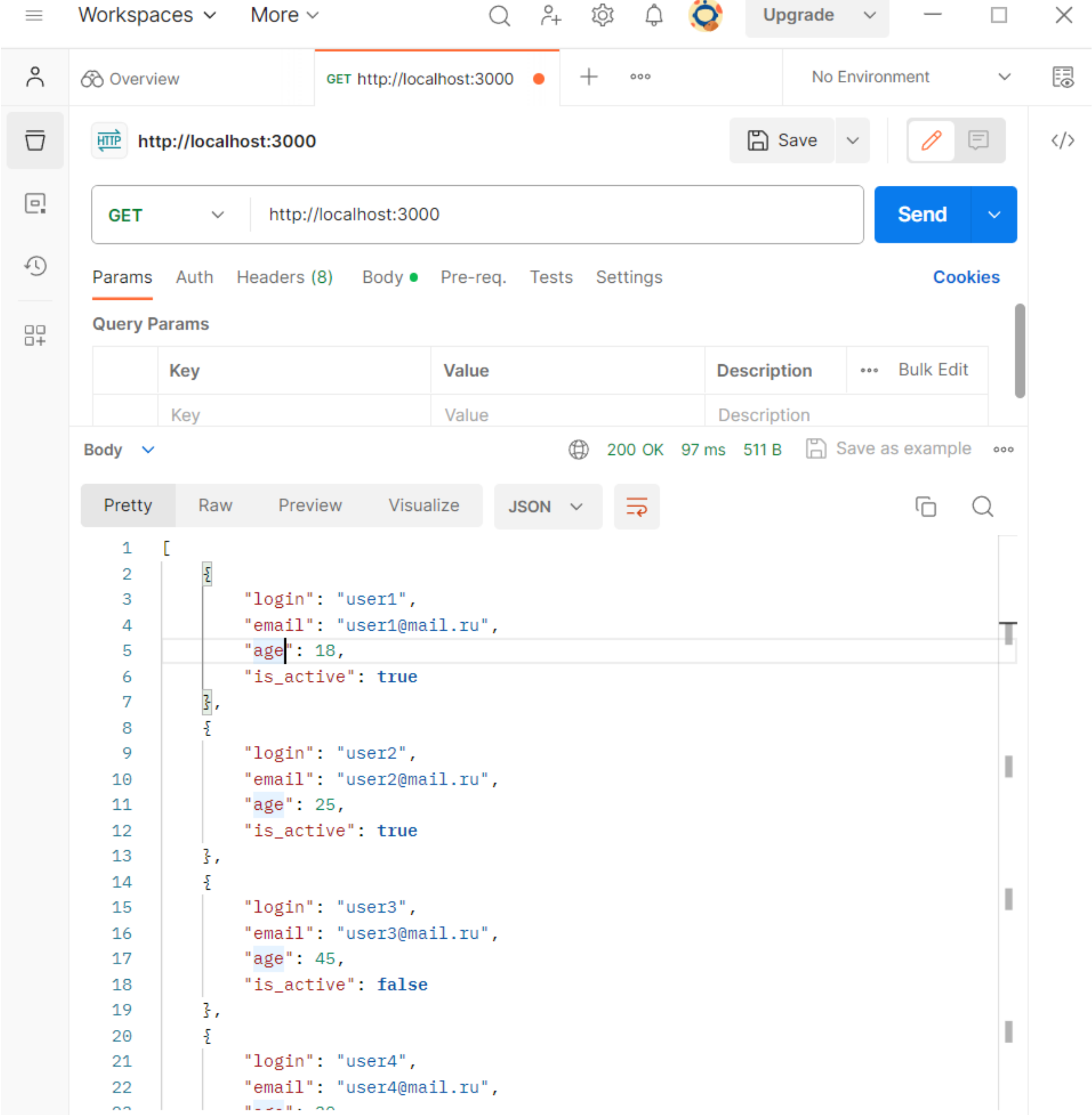


Рис. 9.2. Результат работы с использованием pg-promise

Рассмотрим особенности работы с библиотекой knex.

Листинг кода 9.9

```
const express = require('express');
const app = express();

// Конфигурация подключения к базе данных
const knex = require('knex')({
  client: 'pg',
  connection: {
    host: '127.0.0.1',
    port: '5432',
    user: 'postgres', //Логин для подключения к БД
    password: 'password', //Пароль
    database: 'database_name' //Название БД
  }
});

// Пример маршрута для получения списка пользователей
app.get('/', (req, res) => {
    // Выполнение запроса к базе данных
  knex('users')
    .select()
    // Обработка запроса: если ошибка, то выводим
ошибку, если все хорошо - возвращаем ответ в виде массива
данных
    .then(users => {
      res.json(users); // Отправка списка пользовате-
лей в формате JSON
    })
    .catch(err => {
      console.error(err);
      res.status(500).json({ error: 'Internal Server
Error' }); // Отправка ошибки в формате JSON
    });
});

// Запуск сервера
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на порту ${PORT}`);
});
```

Результат работы представлен на рис. 9.3.

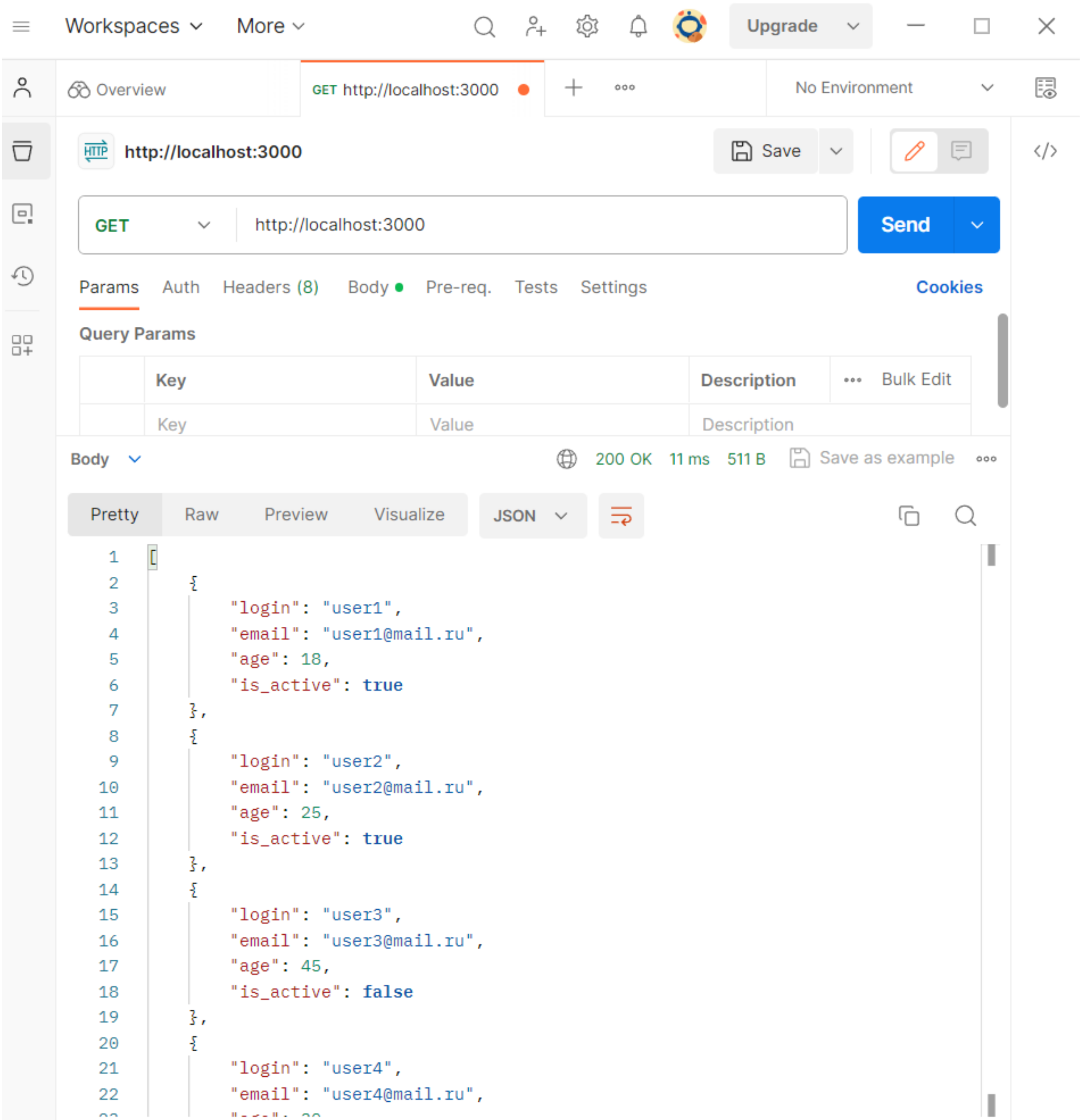


Рис. 9.3. Результат работы с использованием knex

Как правило, для получения данных используется GET-запрос. Он не имеет тела запроса, только параметры.

POST- и PUT-запросы применяются для создания/изменения данных. Они могут иметь тело запроса, в котором можно передавать данные.

DELETE-запрос, как и GET, не имеет тела, используется для удаления данных.

При возвращении ответа от сервера также подходят различные HTTP-статусы ответа.

Статусы, начинающиеся на:

1 – информирование;

2 – успешное завершение;

3 – перенаправление;

4 – ошибка на стороне пользователя;

5 – ошибка на стороне сервера.

Например:

200 – ok;

201 – created (создано);

400 – Bad Request – некорректный запрос (как правило, некорректные данные);

401 – Unauthorized (запрос не авторизован);

404 – Not Fount (не найдено);

500 – Internal Server Error (внутренняя ошибка сервера);

503 – service Unavailable (сервис недоступен).

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Выполните задание по вариантам (номер варианта соответствует порядковому номеру фамилии студента в журнале): создайте три таблицы в БД и поработайте с ними (удалите данные, добавьте данные, измените данные и выведите данные с каким-либо условием). Для каждой таблицы используйте одну из библиотек для работы с БД (например, при работе с первой таблицей продемонстрируйте работу с библиотекой pg, со второй – с библиотекой pg-promise, с третьей – с библиотекой knex).

3. Составьте отчет по результатам работы.

Варианты для выполнения заданий

Создайте базу данных:

1) для работы с пользователями;
2) цветочного магазина;
3) магазина электроники;
4) аптеки;
5) школы;
6) больницы;
7) магазина игрушек;
8) детского сада;
9) продуктового магазина;
10) кинотеатра.

Контрольные вопросы

1. Какие способы подключения к БД вам известны?
2. Что представляет собой knex?
3. Чем характеризуется библиотека pg?
4. Какие особенные черты присущи библиотеке pg-promise?
5. В чём разница между pg, pg-promise и knex?

Лабораторная работа № 10. МИГРАЦИИ БАЗ ДАННЫХ

Цель работы: создание нескольких миграций knex, исследование автогенерируемых таблиц knex_migrations и knex_migrations_lock.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Как правило, в ходе разработки приложения возникают ситуации, когда требуется внести изменения в структуру базы данных, например расширить поле, добавить новое поле, удалить старое поле или создать новую таблицу или связь.

В таких ситуациях важно поддерживать целостность базы данных, чтобы при запуске системы на новом сервере не было необходимости заново ее восстанавливать. Именно для таких целей используются миграции. Их преимущество в том, что разработчик может сам управлять всеми изменениями, которые вносятся в базу, описывать все поля, которые нужно добавить или изменить. При этом сохраняются лог и последовательность изменений, вносимых в базу, что позволяет прослеживать историю изменений.

Библиотека knex предоставляет функционал, который позволяет создавать и выполнять миграции.

Миграции и сидирование с использованием библиотеки knex

Для того чтобы управлять миграциями, нужно создать проект, скачать библиотеку knex и подключиться к БД.

Для этого создайте файл knexfile.js в корневой папке проекта и добавьте следующий код.

Листинг кода 10.1

```
module.exports = {
    development: {
      client: 'pg',
      connection: {
        host: '127.0.0.1',
        port: '5432',
```

```
            user: 'postgres', //Логин для подключения к
БД
            password: 'password', //Пароль
            database: 'database_name' //Название БД
          },
          migrations: {
            directory: './migrations'
          },
          seeds: {
            directory: './seeds'
          }
        }
      };
```

Создайте файл server.js и добавьте код.

Листинг кода 10.2

```
    const express = require('express');
    const app = express();
    const databaseConfig = require('./knexfile');
//относительный путь к файлу настроек
    var knex = require('knex')(databaseConfig);


    // Пример маршрута для получения списка пользователей
    app.get('/', (req, res) => {
        // Выполнение запроса к базе данных
      knex('users')
        .select()
         // Обработка запроса: если ошибка, то выводим
ошибку, если все хорошо - возвращаем ответ в виде массива
данных
        .then(users => {
          res.json(users); // Отправка списка пользовате-
лей в формате JSON
        })
        .catch(err => {
          console.error(err);
          res.status(500).json({ error: 'Internal Server
Error' }); // Отправка ошибки в формате JSON
        });
    });
```

```
// Запуск сервера
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на порту ${PORT}`);
});
```

После этого создайте миграцию.

Листинг кода 10.3

```
npx node_modules/knex/bin/cli.js migrate:make cre-
ate_users_table
```

Для удобства пользования установите библиотеку knex глобально.

Листинг кода 10.4

```
npm i -g knex
```

Если вы установите knex глобально, то команда станет проще.

Листинг кода 10.5

```
npx knex migrate:make create_users_table
```

Результат работы команды представлен на рис. 10.1.

```
PS D:\!!!DOCUMENTs\Desktop\преподаватель\ПиРВП\5 lb> npx knex migrate:make create_users_table
Using environment: development
Using environment: development
Using environment: development
Created Migration: D:\!!!DOCUMENTs\Desktop\преподаватель\ПиРВП\5 lb\migrations\20231101130145_create_users_table.js
PS D:\!!!DOCUMENTs\Desktop\преподаватель\ПиРВП\5 lb>
```

Рис. 10.1. Пример работы команды для создания миграции

После выполнения команды (листинг 10.5) появится папка migrations, в которой будет находиться файл миграции с двумя функциями: одна необходима для выполнения миграции, вторая – для «отката» изменений.

Листинг кода 10.6

```
exports.up = function(knex) {
return knex.schema
    .createTable("users", function(table){
    table.increments("id");
    table.string("login");
    table.string("email");
    table.integer("age");
    table.boolean("is_active")
})
};

exports.down = function(knex) {
};
```

Выполните команду для запуска миграции.

Листинг кода 10.7

```
npx knex migrate:latest
```

Можно выполнить команду по-другому: в package.json добавьте строку в блок scripts.

Листинг кода 10.8

```
"migrate": "knex migrate:latest --knexfile ./knexfile.js"
```

Выполните команду.

Листинг кода 10.9

```
npm run migrate
```

На рис. 10.2 – 10.3 представлены результаты работы команды – в БД добавились новые таблицы.

```
PS D:\!!!DOCUMENTs\Desktop\преподаватель\ПиРВП\5 lb> npx knex migrate:latest
Using environment: development
Batch 1 run: 1 migrations
```

Рис. 10.2. Результат работы в консоли команды для выполнения миграции

Tables (3)
knex_migrations
knex_migrations_lock
users

Рис. 10.3. Созданные таблицы в pgAdmin

В результате выполнения команды созданы три таблицы – users и две служебные. Фреймворк knex также предоставляет возможность выполнять сидирование данных, т. е. вставку новых данных либо изменение имеющихся. Для того чтобы создать файл для сидирования, используется следующая команда.

Листинг кода 10.10

```
knex seed:make users_table
```

Пример сидирования показан ниже.

Листинг кода 10.11

```
exports.seed = function(knex) {
  // Deletes ALL existing entries
  return knex('user').del()
    .then(function () {
      // Inserts seed entries
      return knex('table_name').insert([
        {id: 1, login: 'log1', email: 'em1@gmail.com'},
        {id: 2, login: 'log2', email: 'em2@gmail.com'},
        {id: 3, login: 'log3', email: 'em3@gmail.com'}
      ]);
    });
};
```

Выполните сидирование с помощью следующей команды.

Листинг кода 10.12

```
npx knex seed:run
```

Можно выполнить команду по-другому: в package.json добавьте строку в блок scripts.

Листинг кода 10.13

```
"seed": "knex seed:run --knexfile ./knexfile.js""
```

Выполните команду.

Листинг кода 10.14

```
npm run seed
```

На рис. 10.4 – 10.5 представлены результаты работы команды (см. листинги 10.12, 10.14) – таблица users заполнилась данными.

```
PS D:\!!!DOCUMENTs\Desktop\преподаватель\ПиРВП\5 lb> npx knex seed:run
Using environment: development
Ran 1 seed files
```

Рис. 10.4. Результат работы в консоли команды для выполнения сидирования

	id [PK] integer	login character varying (255)	email character varying (255)	age integer	is_active boolean
1	1	log1	em1@gmail.com	[null]	[null]
2	2	log2	em2@gmail.com	[null]	[null]
3	3	log3	em3@gmail.com	[null]	[null]

Рис. 10.5. Заполненная с помощью сидирования таблица

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Выполните задание по вариантам (номер варианта соответствует номеру фамилии студента в журнале):

– напишите код для миграции для создания новой таблицы и запустите её;

– напишите ещё одну или более миграций для добавления таблиц или их изменения;

– напишите код для сидирования данных в таблицах и запустите сидирование.

3. Составьте отчет по результатам работы.

Варианты для выполнения заданий

Создайте базу данных:

1) для работы с пользователями;
2) цветочного магазина;
3) магазина электроники;
4) аптеки;
5) школы;
6) больницы;
7) магазина игрушек;
8) детского сада;
9) продуктового магазина;
10) кинотеатра.

Контрольные вопросы

1. Что такое миграции БД?
2. Как используется knex для миграции?
3. Что такое сидирование данных?
4. Зачем нужны таблицы knex_migrations и knex_migrations_lock?
5. Как «откатить» миграцию в knex?

Лабораторная работа № 11. ТЕХНОЛОГИЯ ORM

Цель работы: создание схемы базы данных с помощью инструмента Sequelize.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

ORM (англ. Object-Relational Mapping – объектно-реляционное отображение) – технология в программировании, которая связывает объекты с базой данных, тем самым создавая виртуальную базу данных. К виртуальной базе данных можно обращаться, извлекая или записывая информацию без написания SQL-запросов.

Sequelize – это технология Node.js ORM на базе промисов, которая может работать в связке с СУБД Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server, Amazon Redshift.

Sequelize позволяет выполнить 90 % задач без написания SQL-запросов. Эта технология поддерживает создание, обновление, удаление сущностей, а также вложенные сортировки, сложные условия, функцию LEFT JOIN, лимиты, подзапросы, кастомные запросы, обладает защитой от SQL-инъекций и может отменять транзакции [7].

Ниже описаны некоторые основные методы, предоставляемые Sequelize.

sequelize.define (modelName, attributes, options) – метод, используемый для определения модели в Sequelize. Он принимает имя модели, ее атрибуты и опции.

Model.sync(options) создает таблицу в базе данных на основе заданной модели. С помощью этого метода также можно обновлять таблицу, если она уже существует.

Model.create(values, options) создает новую запись в базе данных, используя значения и опции, указанные в аргументах.

Model.findAll(options) возвращает все записи из базы данных, соответствующие опциям поиска.

Model.findOne(options) возвращает первую запись, которая соответствует указанным критериям.

Model.findByPk(id, options) возвращает одну запись из базы данных по указанному первичному ключу.

Model.update(values, options) обновляет записи в базе данных, используя значения и опции, указанные в аргументах.

Model.destroy(options) удаляет записи из базы данных, соответствующие опциям поиска.

Model.hasOne(TargetModel, options) определяет отношение hasOne между текущей моделью и целевой. В этом отношении каждая запись текущей модели связана с одной записью целевой модели.

Model.hasMany(TargetModel, options) устанавливает отношение hasMany между двумя моделями с помощью добавления внешнего ключа к целевой модели, указывающего на первичный ключ исходной модели.

Model.belongsTo(TargetModel, options) определяет отношение belongsTo между текущей моделью и целевой. В этом отношении каждая запись текущей модели принадлежит к одной записи целевой модели.

Model.bulkCreate(records, options) создает несколько записей в базе данных одновременно на основе переданных записей и опций.

Model.findOrCreate(options) ищет запись в базе данных, основываясь на указанных опциях, и создает ее, если она не существует.

Model.count(options) возвращает количество записей модели, удовлетворяющих указанным опциям.

Model.find(options) находит все записи, удовлетворяющие заданным опциям.

Model.max(field, options) возвращает максимальное значение заданного поля для записей, удовлетворяющих заданным опциям.

Model.min(field, options) возвращает минимальное значение заданного поля для записей, удовлетворяющих заданным опциям.

Model.sum(field, options) возвращает сумму заданного поля для записей, удовлетворяющих заданным опциям.

Приведенный список методов, предоставляемых Sequelize, неполный. Существует множество других методов, которые помогают решать поставленные задачи при работе с базами данных, например сортировка, фильтрация, объединение таблиц и т. п. С другими методами можно ознакомиться в официальной документации Sequelize.

Работа с базой данных с помощью Sequelize

Создайте папку и с помощью команды npm init создайте проект. Затем загрузите зависимости:

1) Express.js – для работы веб-сервера;
2) Sequelize – для работы с БД;
3) pg и pg-hstore – в качестве драйверов для СУБД.

Установите зависимости с помощью команды.

Листинг кода 11.1

```
npm install express sequelize pg pg-hstore --save
```

Создайте конфигурационный файл с настройками подключения к БД db.config.js.

Листинг кода 11.2

```
module.exports = {
    HOST: "_хост, на котором развернута БД_",
    USER: "_ваше имя_",
    PASSWORD: "_ваш пароль_",
    DB: "_название_",
    dialect: "postgres",
    pool: {
      max: 5,
      min: 0,
      acquire: 30000,
      idle: 10000
    }
  };
```

Добавьте папку model, в ней создайте файл user.model.js для создания модулей пользователя.

Листинг кода 11.3

```
module.exports = (sequelize, Sequelize) => {
    const User = sequelize.define("user", {
        username: {
            type: Sequelize.STRING
        },
        email: {
```

```
                type: Sequelize.STRING
            },
            password: {
                type: Sequelize.STRING
            }
        });
        return User;
    };
```

Создайте файл role.model.js для создания модулей пользователя.

Листинг кода 11.4

```
    module.exports = (sequelize, Sequelize) => {
        const Role = sequelize.define("role", {
            id: {
                type: Sequelize.INTEGER,
                primaryKey: true
            },
            name: {
                type: Sequelize.STRING
            }
        });
        return Role;
    };
```

Добавьте в папку model файл model.js. Таким образом, в файле model.js будут прописаны код для создания таблиц и ключи.

Листинг кода 11.5

```
    const config = require("./db.config.js");
    const Sequelize = require("sequelize");
    const sequelize = new Sequelize(
      config.DB,
      config.USER,
      config.PASSWORD,
      {
        host: config.HOST,
        dialect: config.dialect,
        operatorsAliases: false,
        pool: {
          max: config.pool.max,
```

```
          min: config.pool.min,
          acquire: config.pool.acquire,
          idle: config.pool.idle
        }
      }
    );
    const db = {};
    db.Sequelize = Sequelize;
    db.sequelize = sequelize;
    db.user = require("./user.model.js")(sequelize, Se-
quelize);
    db.role = require("./role.model.js")(sequelize, Se-
quelize);
    db.role.belongsToMany(db.user, {
      through: "user_roles",
      foreignKey: "roleId",
      otherKey: "userId"
    });
    db.user.belongsToMany(db.role, {
      through: "user_roles",
      foreignKey: "userId",
      otherKey: "roleId"
    });
    db.ROLES = ["user", "admin"];
    module.exports = db;
```

Для упрощения разработки БД воспользуйтесь методом sequelize.sync(). Создайте файл server.js и добавьте код.

Листинг кода 11.6

```
    const express = require("express");
    const app = express();
    const db = require("./model/model");

    app.use(express.json());

    app.get("/", (req, res) => {
        res.json({ message: "Домашняя страница. Бэк рабо-
тает" });
    });
    db.sequelize.sync();
```

```
    const PORT = process.env.PORT || 8080;
    app.listen(PORT, () => {
        console.log(`Server is running on port ${PORT}.`);
    });
```

В package.json в раздел скриптов добавьте скрипт start.

Листинг кода 11.7

```
    "start": "node server.js"
```

Если запустить приложение, то в логах появится сообщение о том, что таблицы созданы. Это также можно проверить в самой базе. На рис. 11.1 представлен результат работы с помощью Sequelize.

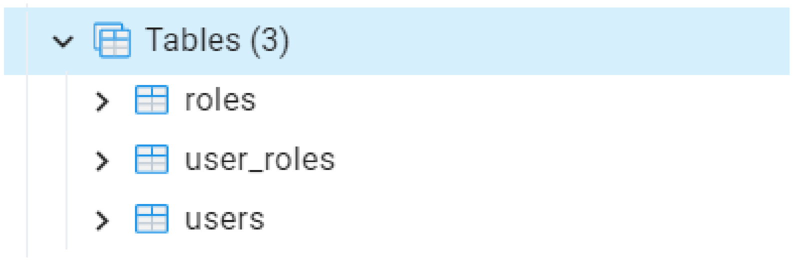


Рис. 11.1. Созданные с помощью Sequelize таблицы

Как видно из рис. 11.1, создалось три таблицы – users, roles и user_roles (последняя создана, чтобы избежать связи «многие ко многим»).

Добавьте код для регистрации авторизации пользователя в системе.

Для этого с помощью SQL-запроса создайте роли администратора и обычного пользователя. Затем добавьте папку controller и файл auth.controller.js, в который вставьте код.

Листинг кода 11.8

```
    const db = require("../model/model");
    const User = db.user;
    const Role = db.role;
    const Op = db.Sequelize.Op;
    exports.signup = (req, res) => {
```

```
        User.create({
            username: req.body.username,
            email: req.body.email,
            password: req.body.password
        })
            .then(user => {
                if (req.body.roles) {
                    Role.findAll({
                        where: {
                            name: {
                                [Op.or]: req.body.roles
                            }
                        }
                    }).then(roles => {
                        user.setRoles(roles).then(() => {
                            res.send({ message: "Пользо-
ватель добавлен" });
                        });
                    });
                } else {

                    user.setRoles([1]).then(() => {
                        res.send({ message: "ПОЛЬЗОВАТЕЛЬ
ДОБАВЛЕН" });
                    });
                }
            })
            .catch(err => {
                res.status(500).send({           message:
err.message });
            });
    };
    exports.signin = (req, res) => {
        User.findOne({
            where: {
                username: req.body.username
            }
        })
            .then(user => {
                if (!user) {
```

```
                    return   res.status(404).send({   mes-
sage: "Пользователь не зарегистрирован" });
                }


                var authorities = [];
                user.getRoles().then(roles => {
                    for  (let  i  =  0;  i  <  roles.length;
i++) {
                        authorities.push("ROLE_"        +
roles[i].name.toUpperCase());
                    }
                    res.status(200).send({
                        id: user.id,
                        username: user.username,
                        email: user.email,
                        roles: authorities,

                    });
                });
            })
            .catch(err => {
                res.status(500).send({           message:
err.message });
            });
    };
```

Создайте папку routes и добавьте в нее файл auth.routes.js.

Листинг кода 11.9

```
    const express = require("express");

    const            controller            =            re-
quire("../controller/auth.controller");
    const router = express.Router();

    router.post(
        "/signup",
        controller.signup
    );
    router.post(
        "/signin",
        controller.signin);
```

```
module.exports = router;
```

В server.js добавьте код.

Листинг кода 11.10

```
***
const auth = require('./routes/auth.routes');
***
app.use("/auth", auth)
```

Результат работы представлен на рис. 11.2 – 11.4.

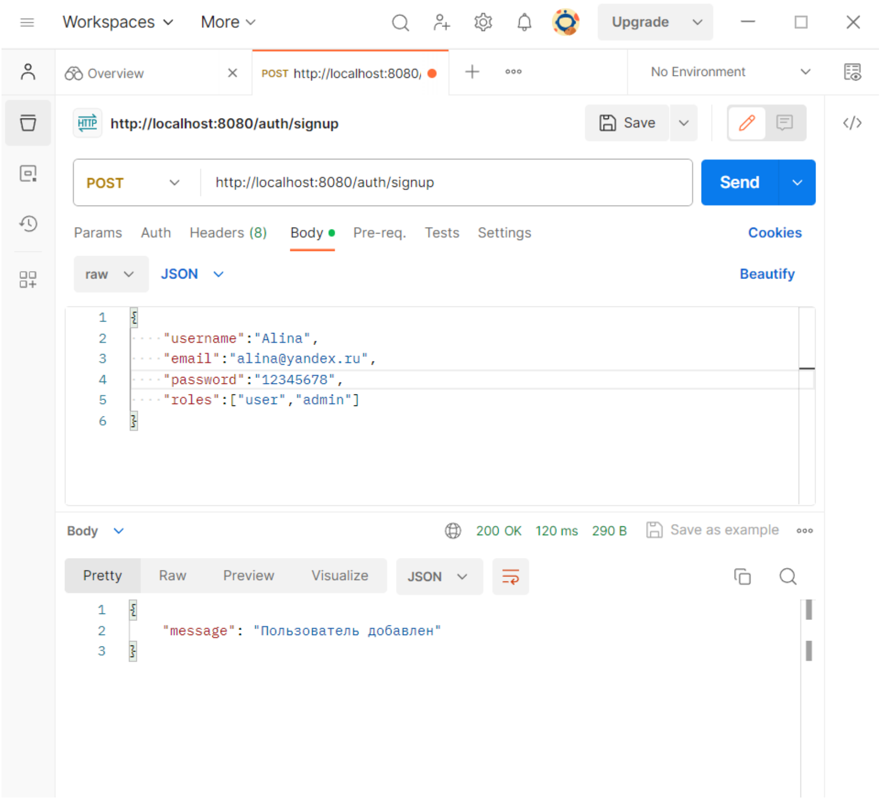
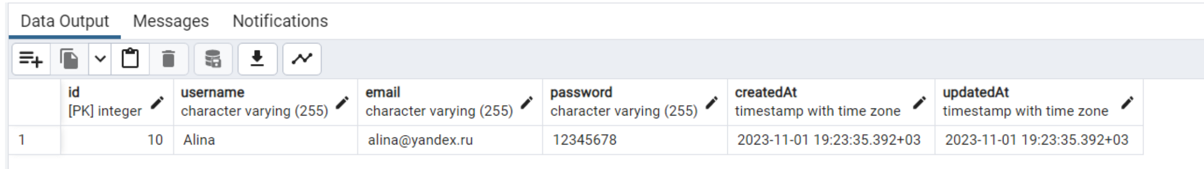


Рис. 11.2. Запрос и ответ от сервера при регистрации



	id [PK] integer	username character varying (255)	email character varying (255)	password character varying (255)	createdAt timestamp with time zone	updatedAt timestamp with time zone
1	10	Alina	alina@yandex.ru	12345678	2023-11-01 19:23:35.392+03	2023-11-01 19:23:35.392+03

Рис. 11.3. Добавление в БД пользователя при регистрации

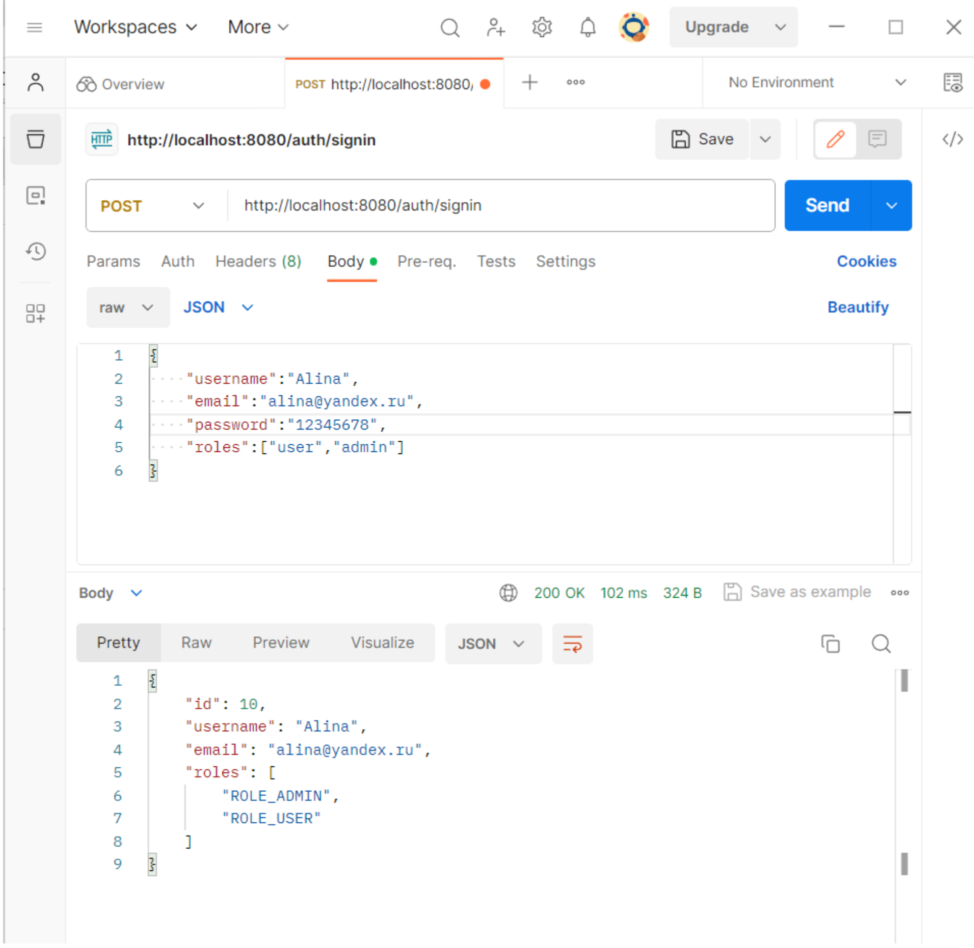


Рис. 11.4. Запрос и ответ от сервера при авторизации

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).
2. Самостоятельно изучите особенности технологии typeORM.
3. Выполните задание по вариантам, используя технологии Sequelize и typeORM (номер варианта соответствует номеру фамилии студента в журнале):
   – создайте схему БД;
   – добавьте данные;
   – удалите данные;
   – редактируйте данные;
   – выведите данные с условием.
4. Сравните две технологии: преимущества, недостатки.
5. Составьте отчет по результатам работы.

Варианты для выполнения заданий

Создайте базу данных:

1) для работы с пользователями;
2) цветочного магазина;
3) магазина электроники;
4) аптеки;
5) школы;
6) больницы;
7) магазина игрушек;
8) детского сада;
9) продуктового магазина;
10) кинотеатра.

Контрольные вопросы

1. Что такое Sequelize?
2. Что такое typeORM?
3. Какие методы для работы с БД вам известны?
4. Что такое ORM?
5. Какие различия между Sequelize и typeORM существуют?

Лабораторная работа № 12. АВТОРИЗАЦИЯ И РЕГИСТРАЦИЯ ПОЛЬЗОВАТЕЛЯ В СИСТЕМЕ

Цель работы: изучение механизма регистрации и авторизации пользователя с помощью библиотеки knex, работа с JWT-токеном, валидация входных данных с помощью библиотеки validate.js.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Одна из технологий взаимодействия клиентской и серверной частей любого приложения – использование REST API. REST – архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Веб-приложения являются распределёнными, так как база данных и серверная часть приложения развёрнуты на сервере, а клиентами выступают пользователи.

REST API, как правило, подразумевает использование протокола HTTP для обмена данными. Одна из его особенностей – отсутствие сведений о его состоянии на сервере, т. е. HTTP-запрос отправляется, обрабатывается контроллером, возвращается ответ и далее об этом запросе сервер ничего не знает.

Возникает вопрос: как по данным запроса определить, какой пользователь его отправил? Ответов на него несколько. Первый – использовать Cookies, однако подобный способ используется нечасто из соображений безопасности. Второе, наиболее распространённое, решение – использование заголовков запроса и какого-то объекта, по которому можно будет идентифицировать пользователя, отправившего запрос. Но как передать эту информацию таким образом, чтобы, просто получив этот объект, нельзя было бы получить конфиденциальные данные пользователя? Ответ – зашифровать. Для таких целей используется JWT (Json Web Token). Токен представляет собой зашифрованную строку, хранящую в себе информацию о пользователе, а также время действия токена, так как делать его бессрочным небезопасно. Данный токен формируется на серверной части, которая отправляет его как ответ клиентскому приложению при авторизации, далее этот запрос можно использовать при повторных запросах.

Рассмотрим способы хранения пароля. Не секрет, что всегда существует риск утечки базы данных пользователей вместе с паролями. Исходя из этого можно сделать вывод, что пароли также следует хранить зашифрованными, однако, в отличие от JWT, их нужно хранить максимально безопасно, даже без возможности расшифровать. Алгоритмы шифрования, которые позволяют это обеспечить, называются односторонними. Таким образом, даже если кто-то украдёт базу данных пользователей, то он не получит доступ к паролям пользователей. Более того, если дважды зашифровать одну и ту же строку, итоговые зашифрованные строки визуально будут отличаться, однако с помощью инструментов шифрования можно корректно их сопоставить и понять, что зашифрована была одна и та же строка.

Валидация формы – это проверка данных, которые ввёл пользователь. Если на сайте есть форма без валидации, то пользователи будут заполнять её как захотят: кто-то пропустит важное поле, кто-то неправильно введёт номер телефона или банковской карты. В результате обрабатывать такие данные станет сложнее, да и небезопасно.

Авторизация и регистрация пользователя с помощью библиотеки knex

Первый шаг при авторизации и регистрации – создание js-проекта и установление необходимых зависимостей:

1) Express.js – для работы веб-сервера;
2) knex – для работы с БД;
3) pg – в качестве драйвера для СУБД;
4) jsonwebtoken – для работы с JWT;
5) bcrypt.js – для шифрования паролей;
6) validate.js – для валидации.

Установите зависимости с помощью следующей команды.

Листинг кода 12.1

```
npm install express knex pg jsonwebtoken bcrypt vali-
date.js
```

После установки создайте папку src, в ней – файл server.js.

Листинг кода 12.2

```
const express = require('express');
const app = express();
app.use(express.json());

app.get("/", (req, res) => {
  res.json({   message:   "Домашняя   страница.   Бэк
работает" });
});

// Запуск сервера
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на порту ${PORT}`);
});
```

Запустите приложение и протестируйте его в Postman (рис. 12.1).

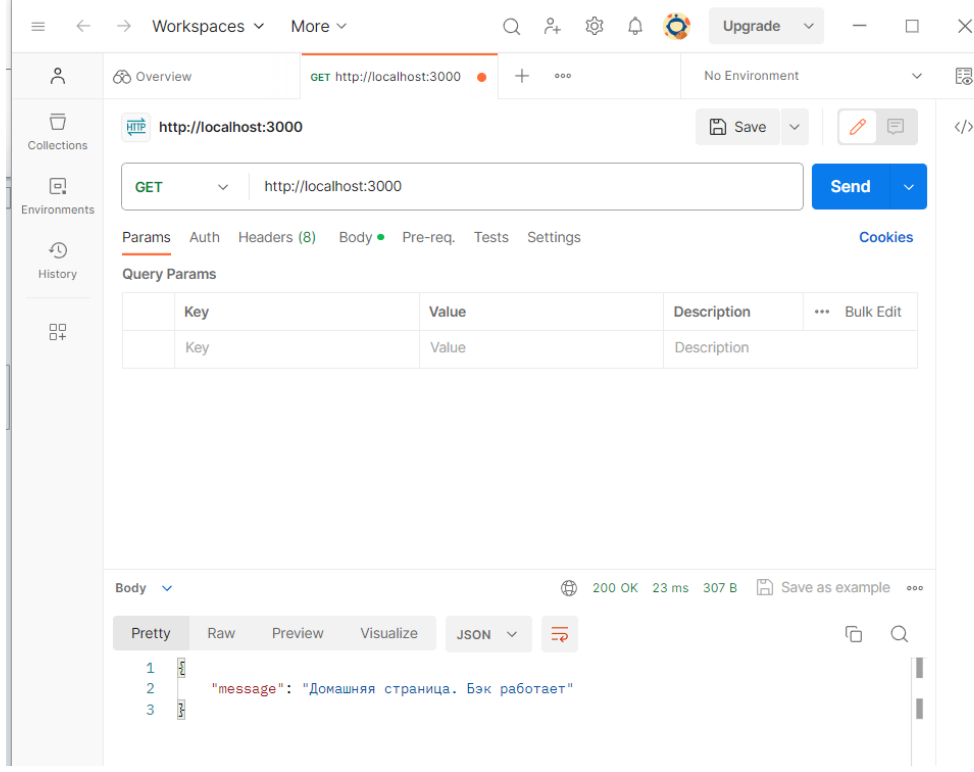


Рис. 12.1. Ответ от сервера при запросе по пути «/»

Успешный запуск приложения говорит о том, что сервер создан корректно.

Теперь необходимо подключиться к БД. Создайте файл knexfile.js в папке config проекта и добавьте код.

Листинг кода 12.3

```
module.exports = {
      client: 'pg',
      connection: {
        host: '127.0.0.1',
        port: '5432',
         user: 'postgres', //Логин для подключения к БД
        password: 'password', //Пароль
        database: 'database_name' //Название БД
      },
      migrations: {
        directory: './migrations'
      },
      seeds: {
        directory: './seeds'

    }
  }
```

После этого с помощью команды создайте миграцию.

Листинг кода 12.4

```
npx node_modules/knex/bin/cli.js migrate:make create_users_table
```

Для удобства пользования установите knex глобально.

Листинг кода 12.5

```
npm i -g knex
```

Если установить knex глобально, то команда станет проще.

Листинг кода 12.6

```
npx knex migrate:make create_tables
```

В созданный файл добавьте код для создания таблиц: «пользователи», «роли» и смежная («пользователь – роль»).

Листинг кода 12.7

```
/**
 * @param { import("knex").Knex } knex
 * @returns { Promise<void> }
 */
exports.up = function(knex) {
    return knex.schema
    .createTable("users", function (table) {
        table.increments("id").primary();
        table.string("username");
        table.string("email");
        table.string("password")

    })
    .createTable("roles", function (table) {
        table.increments("id").primary();
        table.string("name")
    })
    .createTable("user-roles", function (table) {
        table.increments("id").primary();
        table
            .integer("user_id")
            .references("id")
        table
            .integer("roles_id")
            .references("id")
            .inTable("roles")

    })
};

/**
 * @param { import("knex").Knex } knex
 * @returns { Promise<void> }
 */
exports.down = function(knex) {

};
```

Выполните команду.

Листинг кода 12.8

```
npx knex migrate:latest
```

Можно выполнить команду по-другому: в package.json добавьте строку в блок scripts.

Листинг кода 12.9

```
"migrate": "knex migrate:latest --knexfile ./config/knexfile.js"
```

Выполните команду.

Листинг кода 12.10

```
npm run migrate
```

Заполните таблицу «роли» необходимыми данными. Для этого выполните команду для создания сидирования.

Листинг кода 12.11

```
knex seed:make roles_table
```

В созданный файл добавьте код для заполнения таблицы с ролями.

Листинг кода 12.12

```
/**
 * @param { import("knex").Knex } knex
 * @returns { Promise<void> }
 */
exports.seed = async function(knex) {
  // Deletes ALL existing entries
  return knex('roles').del()
  .then(function () {
    // Inserts seed entries
    return knex('roles').insert([
      {id: 1, name: 'User'},
```

```
          {id: 2, name: 'Admin'}
        ]);
      });
    };
```

Выполните сидирование с помощью следующей команды.

Листинг кода 12.13

```
    npx knex seed:run
```

Можно выполнить команду по-другому: в package.json добавьте строку в блок scripts.

Листинг кода 12.14

```
    "seed":       "knex       seed:run       --knexfile
./config/knexfile.js"
```

Выполните команду.

Листинг кода 12.15

```
     npm run seed
```

После того как роли добавлены, можно приступить к написанию так называемых middleware – методов для проверки. Необходимо создать папку middleware, в ней – файл verifySignUp.js.

Листинг кода 12.16

```
    const databaseConfig = require('../config/knexfile');
//относительный путь к файлу настроек
    var knex = require('knex')(databaseConfig);

    checkDuplicateUsernameOrEmail = (req, res, next) => {

        // Username
        knex("users")
            .select().where("username",
req.body.username)
            .then(user => {
                if (user.length != 0) {
```

```
                    res.status(400).send({
                        message: "Ошибка. Имя пользовате-
ля уже занято!"
                    });
                    return;
                }
                // Email
                knex("users")
                    .select().where("email",
req.body.email)
                    .then(user => {
                        if (user.length != 0) {
                            res.status(400).send({
                                message:  "Ошибка.  Email
уже используется!"
                            });
                            return;
                        }
                        next();
                    });
            });
    };
    checkRolesExisted = (req, res, next) => {
        if (req.body.roles) {
            for  (let  i  =  0;  i  <  req.body.roles.length;
i++) {
                knex("roles")
                .select().where("name",
req.body.roles[i])
                    .then(role => {
                        if (role.length == 0) {
                            con-
sole.log(req.body.roles[i])
                            res.status(400).send({
                                message:  "Ошибка!  Такой
роли не существует = "  +  req.body.roles[i]
                            });
                            return;
                        }
                    });
            }


        }
        next();
```

```
    };
    const verifySignUp = {
        checkDuplicateUsernameOrEmail:      checkDuplica-
teUsernameOrEmail,
        checkRolesExisted: checkRolesExisted
    };
    module.exports = verifySignUp;
```

Данный скрипт (листинг кода 12.16) позволит избежать дублирования логинов и email в БД и при этом вернуть пользователю корректный ответ.

Далее необходимо создать контроллеры для авторизации и регистрации пользователей. Создайте папку controller, в ней – два файла: signin.js и signup.js.

Листинг кода 12.17 (signin.js)

```
    const databaseConfig = require('../config/knexfile');
//относительный путь к файлу настроек
    var knex = require('knex')(databaseConfig);
    const config = require("../config/auth.config");
    var jwt = require("jsonwebtoken");
    const bcrypt = require('bcrypt');

    exports.signin = (req, res) => {

        knex('users')
              .join('user-roles', 'users.id', '=', 'user-
roles.user_id')
                 .join('roles',  'roles.id',  '=',  'user-
roles.roles_id')
             .where('users.username', req.body.username)
                .select('users.id', 'users.username', 'us-
ers.email', 'users.password', 'roles.name')
             .then(users => {
                 console.log(users)
                 if (users.length == 0) {
                     // Если пользователь не найден
                      res.status(401).json({ error: 'Поль-
зователь с таким именем не зарегистрировался!' });
                 } else {
```

```
                    bcrypt.compare(req.body.password, us-
ers[0].password, (err, result) => {
                        if (err) {
                            console.error(err);
                             res.status(500).json({ error:
'Неверный пароль!' });
                            return;
                        }

                    })

                         var  token  =  jwt.sign({  id:  us-
ers[0].id }, config.secret, {
                        expiresIn: 86400 // 24 часа
                    });
                    // Если пользователь найден
                          const  roles  =  users.map(row  =>
'ROLE_' + row.name.toUpperCase());
                    res.status(200).send({
                        id: users[0].id,
                        username: users[0].username,
                        email: users[0].email,
                        roles: roles,
                        accessToken: token
                    });
                }
            })
            .catch(error => {
                console.error(error);
                 res.status(500).json({ error: 'Внутренняя
ошибка сервера' });
            });
    };
```

Листинг кода 12.18 (signup.js)

```
     const databaseConfig = require('../config/knexfile');
//относительный путь к файлу настроек
     var knex = require('knex')(databaseConfig);

     const bcrypt = require('bcrypt');
     const { validate } = require('validate.js')

     exports.signup = (req, res) => {

          const { username, email, password, roles } =
req.body;
         const constraints = {
             username: {
                 presence: true,
                 length: {
                     minimum: 4,
                     maximum: 20
                 }
             },
             password: {
                 presence: true,
                 length: {
                     minimum: 4,
                     maximum: 20
                 }
             }
         };

         const validation = validate({ username, password
}, constraints);

         if (validation) {
             res.status(400).json({ error: validation });
             return;
         }

         // Хеширование пароля
          bcrypt.hash(password, 10, (err, hashedPassword)
=> {
             if (err) {
                 console.error(err);
```

```
                    res.status(500).json({ error: 'Internal
server error' });
                  return;
              }

              knex.transaction(async trx => {
                  try {
                      const userId = await trx('users')
                              .insert({ username, password:
hashedPassword, email })
                          .returning('id');

                      const roleIds = await trx('roles')
                          .whereIn('name', roles)
                          .pluck('id');

                      console.log(roleIds)

                      await trx('user-roles').insert(
                          roleIds.map(roleId => ({
                              user_id: userId[0].id,
                              roles_id: roleId
                          }))
                      );

                            res.status(200).send({  message:
"Пользователь добавлен" });
                  } catch (error) {
                      console.error(error);
                       res.status(500).json({ error: 'Inter-
nal server error' });
                  }
              })
          })


     };
```

В папке config добавьте auth.config.js.

Листинг кода 12.19

```
module.exports = {
    secret: "bcrypt-secret-key"
  };
```

В ключе можно указать любую строку, которая будет использоваться для шифрования токена.

Теперь создайте папку routes, добавьте файл auth.routes.js с кодом.

Листинг кода 12.20

```
const { verifySignUp } = require("../middleware/verifySignUp");
const express = require("express");

const signin = require("../controller/signin");
const signup = require("../controller/signup");
const router = express.Router();

router.use((req, res, next)=> {
    res.header(
        "Access-Control-Allow-Headers",
          "x-access-token, Origin, Content-Type, Accept, Authorization"
    );
    console.log(req.params)
    next();
});
router.post(
    "/signup",
    [
        checkDuplicateUsernameOrEmail,
        checkRolesExisted
    ],
    signup.signup
);
router.post("/signin", signin.signin);

module.exports = router;
```

В файл server.js добавьте код.

Листинг кода 12.21

```
***
const auth = require('./routes/auth.routes');
***
app.use("/auth", auth)
```

Запустите приложение и протестируйте с помощью Postman. На рис. 12.2 – 12.3 представлены ответы от сервера.

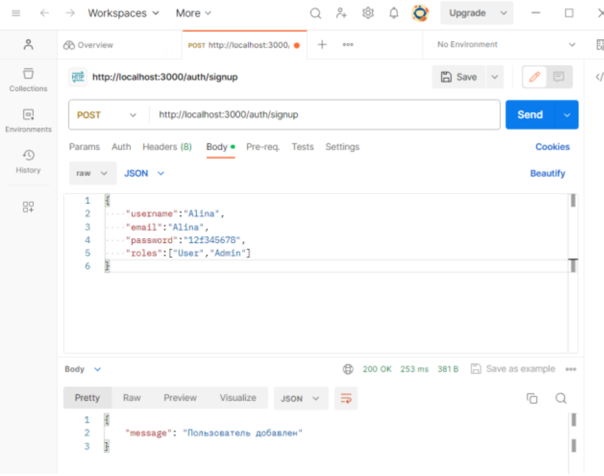


Рис. 12.2. Тестирование регистрации

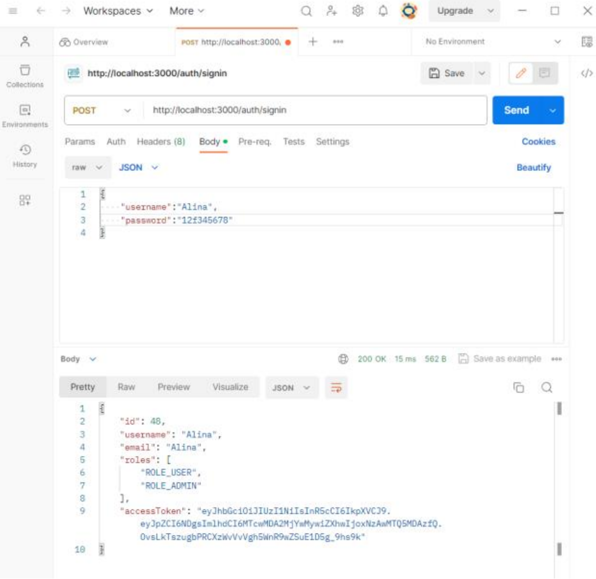


Рис. 12.3. Тестирование авторизации

Теперь можно приступить к написанию форм для регистрации и авторизации. Для этого необходимо установить ejs.

Листинг кода 12.22

```
npm i ejs
```

Создайте папку view, в ней – папки pages и partials. В папке partials создайте файлы footer.ejs, head.ejs, header.ejs, form_login.ejs, form_reg.ejs с кодом.

Листинг кода 12.23 (footer.ejs)

```
<p class="text-center text-muted">© Copyright 2020 The Awesome People</p>
```

Листинг кода 12.24 (head.ejs)

```
<meta charset="UTF-8">
<title>EJS Is Fun</title>

<!-- CSS (load bootstrap from a CDN) -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.2/css/bootstrap.min.css">
<style>
    body { padding-top:50px; }
</style>
```

Листинг кода 12.25 (header.ejs)

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item">
        <a class="nav-link" href="/auth/signup">Зарегистрироваться</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="/auth/signin">Авторизоваться</a>
      </li>

  </nav>
```

Листинг кода 12.26 (form_login.ejs)

```
    <h1>Авторизация</h1>
      <form method="POST" action="/auth/signin">
        <label for="username">Имя пользователя:</label>
        <input type="text" id="username" name="username"
required>
        <br>
        <label for="password">Пароль:</label>
        <input        type="password"        id="password"
name="password" required>
        <br>
        <input type="submit" value="Войти">
      </form>
```

Листинг кода 12.27 (form_reg.ejs)

```
    <h1>Регистрация</h1>
    <br>

    <form method="POST" action="/auth/signup">
      <label for="username">Имя пользователя:</label>
      <input  type="text"  id="username"  name="username"
required>
      <br>
      <label for="email">Почта:</label>
      <input  type="email"  id="email"  name="email"  re-
quired>
      <br>
      <label for="password">Пароль:</label>
      <input        type="password"         id="password"
name="password" required>
      <br>
      <% for(var i=0; i<roles.length; i++) { %>

        <input   type="hidden"   name="roles"   value="<%=
roles[i] %>" />

        <% } %>



          <input            type="submit"            val-
ue="Зарегистрироваться">
    </form>
```

Итак, созданы частичные шаблоны. Теперь нужно добавить их в представления. Возьмите файлы reg.ejs и login.ejs и, используя синтаксис include, добавьте частичные шаблоны. Для этого создайте в папке pages файлы reg.ejs и login.ejs.

Листинг кода 12.28 (login.ejs)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <%- include('../partials/head'); %>
    <title>Авторизация</title>
</head>
<body class="container">

<header>
    <%- include('../partials/header'); %>
</header>
<%- include('../partials/form_login'); %>
<footer>
    <%- include('../partials/footer'); %>
</footer>
</body>
</html>
```

Листинг кода 12.29 (reg.ejs)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <%- include('../partials/head'); %>
    <title>Регистрация</title>
</head>
<body class="container">

    <header>
        <%-   include('../partials/header',   {vari-
ant:'compact'}); %>
    </header>
    <main>
```

```
<div class="row">
       <%- include('../partials/form_reg'); %>
</div>
</main>
<footer>
    <%- include('../partials/footer'); %>
</footer>
</body>
</html>
```

Добавьте в serve.js и auth.router.js код.

Листинг кода 12.30 (serve.js)

```
app.use(express.urlencoded({ extended: false }))
app.set('view engine', 'ejs');
```

Листинг кода 12.31 (auth.router.js)

```
***
router.get('/signup', (req, res) => {
    res.render('pages/reg',{
          roles: ["User","Admin"],
       });
  });
***
router.get('/signin', (req, res) => {
    res.render('pages/login');
  });
```

Запустите проект и откройте его в браузере (http://localhost:3000/). На рис. 12.4 – 12.7 отражены процессы регистрации и авторизации.

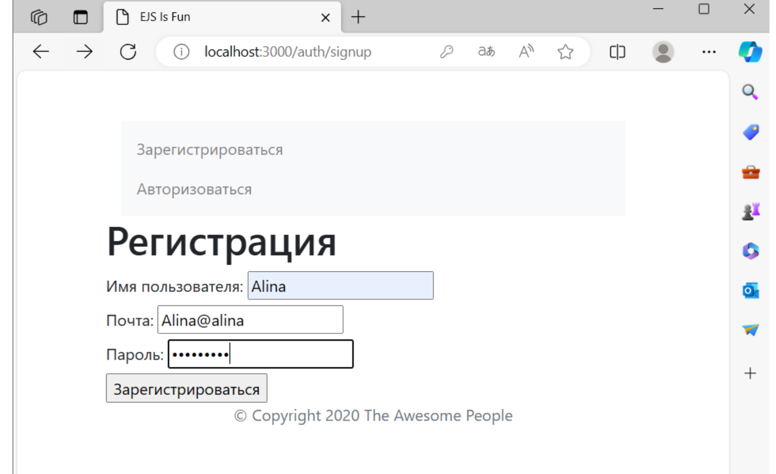


Рис. 12.4. Тестирование регистрации (ввод данных)

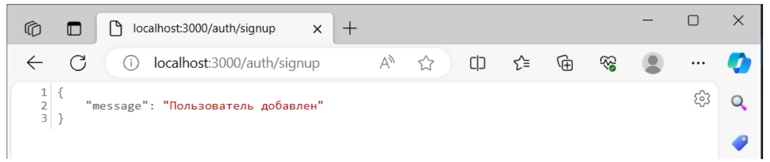


Рис. 12.5. Тестирование регистрации (результат работы)

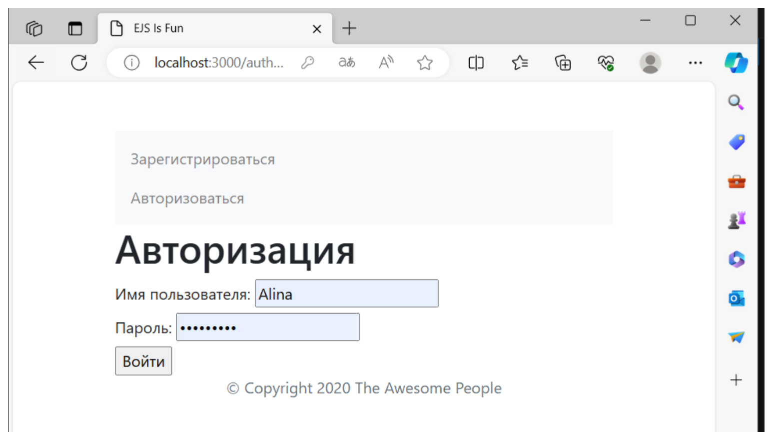


Рис. 12.6. Тестирование авторизации (ввод данных)

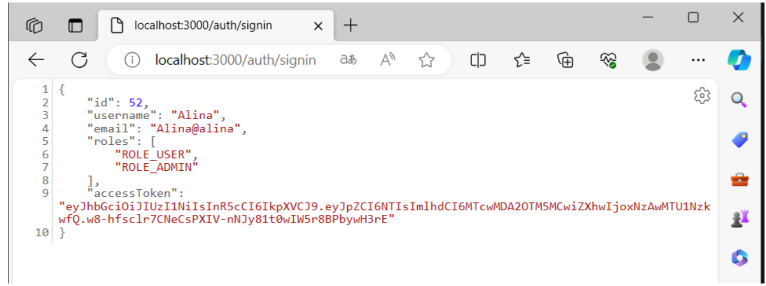


Рис. 12.7. Тестирование авторизации (ввод данных)

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Разработайте HTTP-страницы. На страницах должны быть формы регистрации и авторизации.

3. Добавьте валидацию с помощью библиотеки validate.js (помимо длины).

4. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое REST API?

2. Что подразумевает использование REST API?

3. Как по данным запроса определить, какой пользователь его отправил?

4. Какие библиотеки были использованы для построения запросов?

5. Для каких целей используется JWT?

Лабораторная работа № 13. ЗАГРУЗКА ФАЙЛОВ

Цель работы: организация работы с файлами, загруженными пользователями, работа с библиотекой multer.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Multer – это Node.js промежуточное программное обеспечение для обработки запросов в формате multipart/form-data, которое в основном используется для загрузки файлов. Оно написано поверх busboy для максимальной эффективности.

Каждый файл содержит следующую информацию:

– fieldname – имя поля, указанное в форме;

– originalname – имя файла на компьютере пользователя;

– encoding – тип кодировки файла;

– mimetype – mime-тип файла;

– size – размер файла в байтах;

– destination – папка, в которую был сохранен файл (DiskStorage);

– filename – имя файла (DiskStorage);

– path – полный путь к загруженному файлу (DiskStorage);

– buffer – буфер всего файла (MemoryStorage).

Multer принимает объект options со свойствами, основным из которых является dest, сообщающее multer, куда загружать файлы. Если не указать объект options, файлы будут сохранены в памяти и никогда не будут записаны на диск.

По умолчанию multer переименовывает файлы, чтобы избежать конфликтов именования. Функцию переименования можно настроить в соответствии с потребностями разработчика.

Ниже приведены параметры, которые могут быть переданы multer:

– dest или storage – место хранения файлов;

– fileFilter – функция для управления тем, какие файлы принимаются;

– limits – ограничения на загружаемые данные;

– preservePath – функция сохранения полного пути к файлам, а не только базового имени.

В обычном веб-приложении может потребоваться только свойство dest, и оно настроено, как показано ниже.

Листинг кода 13.1

```
const upload = multer({ dest: 'uploads/' })
```

Если требуется тщательнее контролировать загрузки, то следует использовать опцию storage вместо dest. Рассмотрим методы multer:

– .single(fieldname) – принимает один файл с именем fieldname. Один файл будет сохранен в req.file.

– .array(fieldname[, maxCount]) – принимает массив файлов, все с именем fieldname. При необходимости выводится ошибка, если загружено количество, больше указанного (maxCount). Массив файлов будет сохранен в req.files.

– .fields(fields) – принимает набор файлов, указанный в параметре fields. В req.files будет сохранен объект с массивами файлов. Fields представляет собой массив объектов с названием name и необязательно с параметром maxCount.

Листинг кода 13.2

```
[
  { name: 'avatar', maxCount: 1 },
  { name: 'gallery', maxCount: 8 }
]
```

– .none() – принимает только текстовые поля. При загрузке любого файла будет выдана ошибка с кодом "LIMIT_UNEXPECTED_FILE".

– .any() – принимает все файлы, которые содержатся в запросе. Массив файлов будет храниться в свойстве req.files.

ПРЕДУПРЕЖДЕНИЕ: Убедитесь, что обрабатываются файлы, которые загружает пользователь. Никогда не добавляйте multer в качестве глобального промежуточного программного обеспечения, поскольку злоумышленник может загружать файлы по маршруту, кото-

рый вы не ожидали. Используйте эту функцию только на маршрутах, где вы обрабатываете загруженные файлы.

Multer поставляется с системами хранения DiskStorage и MemoryStorage. Другие системы доступны у третьих сторон.

DiskStorage – механизм дискового хранилища, который предоставляет полный контроль над хранением файлов на диске.

Листинг кода 13.3

```
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, '/tmp/my-uploads')
  },
  filename: function (req, file, cb) {
    const uniqueSuffix = Date.now() + '-' +
Math.round(Math.random() * 1E9)
    cb(null, file.fieldname + '-' + uniqueSuffix)
  }
})

const upload = multer({ storage: storage })
```

В DiskStorage доступны два свойства – destination и filename, в которых содержатся функции, определяющие, где должен храниться файл:

– destination – используется для определения, в какой папке должны храниться загруженные файлы. Это также может быть задано как string (например, '/tmp/uploads'). Если не задано значение destination, используется каталог операционной системы по умолчанию для временных файлов;

– filename – используется для определения того, какое имя должно быть у файла внутри папки. Если не задано значение filename, каждому файлу будет присвоено случайное имя, которое не включает никакого расширения файла.

Каждой функции передаются как запрос (req), так и некоторая информация о файле (file), чтобы помочь с принятием решения.

Обратите внимание, что свойство req.body возможно, оно еще не полностью заполнено. Это зависит от порядка, в котором клиент передает поля и файлы на сервер.

Для понимания соглашения о вызовах, используемого при обратном вызове (необходимость передачи null в качестве первого параметра), обратитесь к лабораторной работе № 5.

MemoryStorage – механизм хранения файлов в памяти как buffer объектов. У него нет никаких опций.

Пример MemoryStorage представлен ниже.

Листинг кода 13.4

```
const storage = multer.memoryStorage()
const upload = multer({ storage: storage })
```

При использовании хранилища в памяти информация о файле будет содержать поле с именем buffer, которое содержит весь файл.

ПРЕДУПРЕЖДЕНИЕ: очень быстрая загрузка очень больших файлов или относительно небольших файлов в большом количестве может привести к тому, что приложению не хватит памяти при использовании накопителя.

В multer существует объект, определяющий ограничения размера необязательных свойств, – limits. Multer передает этот объект непосредственно в busboy.

Доступны следующие целочисленные значения ограничения размеров:

– fieldNameSize – максимальный размер имени поля (по умолчанию 100 байт);

– fieldSize – максимальный размер значения поля (в байтах) (по умолчанию 1 Мб);

– fields – максимальное количество нефайловых полей (бесконечное);

– fileSize – для составных форм – максимальный размер файла (в байтах) (бесконечное);

– files – для составных форм – максимальное количество полей файла (бесконечное);

– parts – для составных форм – максимальное количество частей (поля + файлы) (бесконечное);

– headerPairs – для составных форм – максимальное количество пар «ключ – заголовок».

Указание ограничений может помочь защитить сайт от атак типа «отказ в обслуживании» (DoS).

Установите функцию fileFilter для управления тем, какие файлы следует загружать, а какие – пропускать. Функция должна выглядеть следующим образом.

Листинг кода 13.5

```
function fileFilter (req, file, cb) {
  // The function should call `cb` with a boolean
  // to indicate if the file should be accepted
  // To reject this file pass `false`, like so:
  cb(null, false)
  // To accept the file pass `true`, like so:
  cb(null, true)
  // You can always pass an error if something goes
wrong:
  cb(new Error('I don\'t have a clue!'))

}
```

Обработка ошибок

При обнаружении ошибки multer передает сообщение об ошибке во фреймворк Express.js, который обрабатывает ошибку стандартным способом.

Если необходимо перехватывать ошибки конкретно из multer, можно самостоятельно вызвать функцию промежуточного программного обеспечения. Кроме того, если нужно перехватывать только ошибки multer, то можно использовать MulterError класс, который привязан к самому multer объекту (например, err instanceof multer.MulterError).

Листинг кода 13.6

```
const multer = require('multer')
const upload = multer().single('avatar')
app.post('/profile', function (req, res) {
  upload(req, res, function (err) {
    if (err instanceof multer.MulterError) {
      // A Multer error occurred when uploading.
    } else if (err) {
```

```
        // An unknown error occurred when uploading.
      }

      // Everything went fine.
    })
  })
```

Добавление фото с помощью multer

Создайте js-проект и установите необходимые зависимости:
1) Express.js – для работы веб-сервера;
2) knex – для работы с БД;
3) pg – в качестве драйвера для СУБД;
4) multer;
5) ejs – для создания страниц.

Листинг кода 13.7

```
npm install express knex pg jsonwebtoken bcrypt vali-
date.js multer ejs
```

После установки создайте папку src, в ней – файл server.js для проверки работоспособности сервера.

Листинг кода 13.8

```
const express = require('express');
const app = express();
app.use(express.json());

app.get("/", (req, res) => {
  res.json({ message: "Домашняя страница. Бэк
работает" });
});

// Запуск сервера
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на порту ${PORT}`);
});
```

Запустите приложение и протестируйте его в Postman. На рис. 13.1 представлено тестирование пути «/».

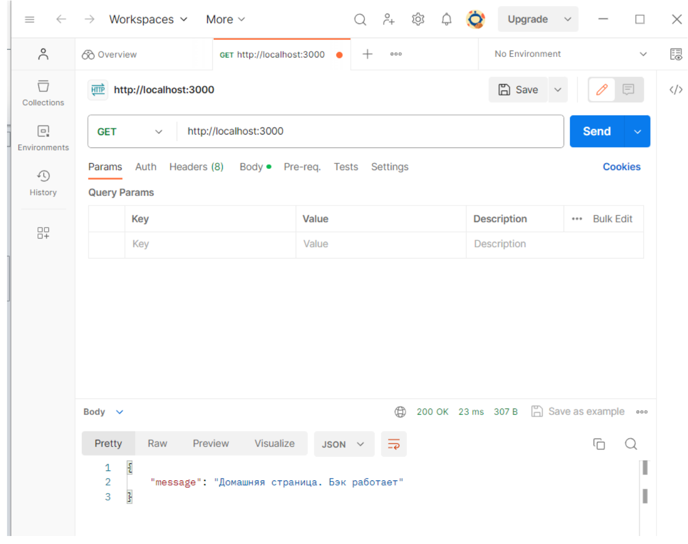


Рис. 13.1. Тестирование пути «/» в Postman

Успешный запуск приложения говорит о том, что сервер создан корректно. Теперь необходимо подключиться к БД. Для этого создайте файл knexfile.js и добавьте код.

Листинг кода 13.9

```
module.exports = {
      client: 'pg',
      connection: {
        host: '127.0.0.1',
        port: '5432',
         user: 'postgres', //Логин для подключения к
БД
        password: 'password', //Пароль
        database: 'database_name' //Название БД
      },
      migrations: {
        directory: './migrations'
      },
      seeds: {
        directory: './seeds'

    }
  }
```

После этого с помощью команды создайте миграцию.

Листинг кода 13.10

```
    npx  node_modules/knex/bin/cli.js  migrate:make  cre-
ate_file_table
```

Для удобства установите knex глобально.

Листинг кода 13.11

```
    npm i -g knex
```

Если установить knex глобально, то команда станет проще.

Листинг кода 13.12

```
    npx knex migrate:make create_file_table
```

В созданный файл добавьте код для создания таблиц: «пользователи», «роли» и смежная («пользователь – роль»).

Листинг кода 13.13

```
/**
 * @param { import("knex").Knex } knex
 * @returns { Promise<void> }
 */
exports.up = function(knex) {
    return knex.schema
    .createTable("file", function (table) {
        table.increments("id").primary();
        table.string("data");
        table.string("contentType")

    })

};

/**
 * @param { import("knex").Knex } knex
 * @returns { Promise<void> }
 */
```

```
exports.down = function(knex) {

};
```

Выполните команду.

Листинг кода 13.14

```
npx knex migrate:latest
```

Можно выполнить команду по-другому: в package.json добавьте строку в блок scripts.

Листинг кода 13.15

```
"migrate": "knex migrate:latest --knexfile ./knexfile.js"
```

Выполните команду.

Листинг кода 13.16

```
npm run migrate
```

Далее в корне проекта создайте папку uploads (здесь будут храниться файлы).

Затем создайте папку router и в ней – файл file.js с кодом.

Листинг кода 13.17

```
const multer = require("multer");
const path = require("path");
const databaseConfig = require('../knexfile'); //относительный путь к файлу настроек
var knex = require('knex')(databaseConfig);
const express = require("express");
const router = express.Router();
// рендеринг страницы
router.get('/', (req, res) => {
    res.render('pages/file');
});
// Задаем параметры для хранения файла(где, как будут называться)
```

```
    const storage = multer.diskStorage({
        destination: function (req, file, cb) {
            cb(null, "uploads");
        },
        filename: function (req, file, cb) {
            cb(
                null, file.fieldname + "-" + Date.now() +
path.extname(file.originalname)
            );
        },
    });

    const upload = multer({ storage: storage });
    // Добавление файла
    router.post("/uploadPhoto",
        upload.single("myImage"),
        (req, res) => {

            console.log(req.file)
            const obj = {
                img: {
                    data: req.file.filename,
                    contentType: "image/png"
                }
            }
            knex("file")
                .insert(obj.img)
                .then(users => {
                     res.json("Файл загружен в БД");
                })
                .catch(err => {
                    console.error(err);
                    res.status(500).json({ error: 'Inter-
nal Server Error' });
                    // Отправка ошибки в формате JSON
                });

        })

    // Вывод файла
    router.get("/:fileId", (req, res) => {
        const fileId = req.params.fileId;
        knex('file')
```

```
            .select("*")
            .where({data: fileId})
            .then(file => {
                if (file[0]) {
                    const filePath = path.join(
                        __dirname,
                        "..",
                        "/uploads",
                        file[0].data
                    );

                    res.setHeader("Content-Type",
file[0].contentType)
                    res.sendFile(filePath);

                } else{
                    res.send("Нет фото")
                }

            })
            .catch(err => {
                console.error(err);
                res.status(500).json({  error:  'Internal
Server Error' });
                // Отправка ошибки в формате JSON
            });

    })
    module.exports = router;
```

Добавьте код для обработки изображения. Теперь можно написать формы для добавления фото.

Создайте папку view, в ней – папки pages и partials. В папке partials создайте файлы footer.ejs, head.ejs, form.ejs с кодом.

Листинг кода 13.18 (footer.ejs)

```
    <p  class="text-center  text-muted">©  Copyright  2020
The Awesome People</p>
```

Листинг кода 13.19 (head.ejs)

```
<meta charset="UTF-8">
<title>Загрузка файлов</title>

<!-- CSS (load bootstrap from a CDN) -->
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.5.2/css/bootstrap.min.css">
<style>
    body { padding-top:50px; }
</style>
```

Листинг кода 13.20 (form.ejs)

```
<h1>Загрузка файлов</h1>
<form action="/file/uploadPhoto"
enctype="multipart/form-data" method="post">
  <input class="choose-file-btn" type="file"
name="myImage" />
  <input class="upload-btn" type="submit" val-
ue="Upload Photo" />
</form>
```

Таким образом, определены частичные шаблоны. Теперь нужно добавить их в представления. Возьмите файлы file.ejs и используйте синтаксис include для добавления частичных шаблонов. Для этого создайте в папке pages файл file.ejs.

Листинг кода 13.21 (file.ejs)

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

<head>
    <%- include('../partials/head'); %>
        <title>Загрузка файлов</title>
</head>

<body class="container">
    <%- include('../partials/form'); %>
```

```
            <footer>
                <%- include('../partials/footer'); %>
            </footer>

    </body>

    </html>
```

Измените файл server.js.

Листинг кода 13.22

```
const express = require('express');
const app = express();

const fileUp = require('./router/file');

app.use(express.json());

app.use(express.urlencoded({ extended: true }));

app.set("view engine", "ejs");
app.use("/file", fileUp);

app.get("/", (req, res) => {
  res.json({ message: "Домашняя страница. Бэк работа-
ет" });
});

// Запуск сервера
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на порту ${PORT}`);
});
```

Запустите проект и откройте его в браузере (http://localhost:3000/). На рис. 13.2 – 13.5 представлено добавление фото и вывод фото на экран.

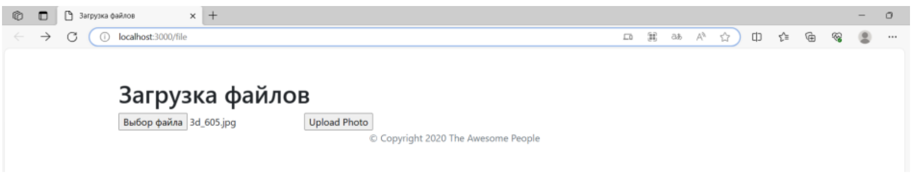


Рис. 13.2. Тестирование добавления фото (ввод данных)

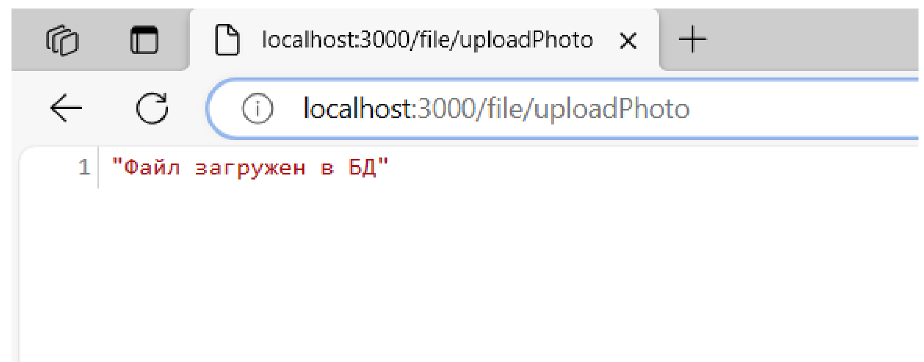
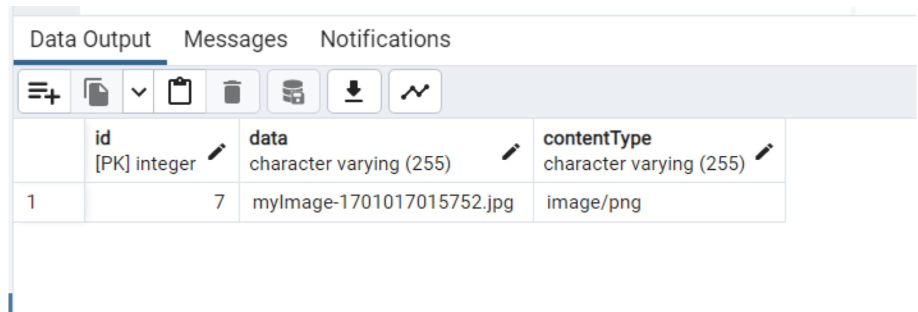


Рис. 13.3. Тестирование добавления фото (результат работы)



Data Output | Messages | Notifications

	id [PK] integer	data character varying (255)	contentType character varying (255)
1	7	myImage-1701017015752.jpg	image/png

Рис. 13.4. Тестирование добавления фото (в БД)

Рис. 13.5. Тестирование авторизации (вывод файла)

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Разработайте HTTP-страницы. На странице должна быть форма с добавлением файла и ещё два любых поля.

3. После добавления файла выведите его на экран.

4. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое multer?
2. Какие ещё библиотеки для работы с типом form-data вы знаете?
3. Какие данные нужно хранить в БД, чтобы отобразить файл?
4. Какие системы хранения существуют в multer?
5. Какие типы приема файлов существуют в библиотеке multer?
6. Какие параметры можно задать в DiskStorage?
7. Какие параметры можно задать в MemoryStorage?
8. Какие параметры можно задать в limits?

Лабораторная работа № 14. ЛОГИРОВАНИЕ

Цель работы: организация работы с библиотеками Winston и Morgan.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Логирование – незаменимый инструмент в отладке JS кода. Расставив логи в критические места, при возникновении ошибки можно посмотреть, что произошло в консоли. По логам можно увидеть последовательность действий и понять, где произошла ошибка. Но обычно происходит по-другому.

Morgan – это промежуточное программное обеспечение Node.js и Express.js для регистрации HTTP-запросов и ошибок.

Winston – это регистратор практически для всего. Он включает поддержку нескольких вариантов хранения и уровней журналов, запросов к журналам и даже встроенного профилировщика.

Эффективное средство логирования имеет решающее значение для успеха любого приложения. В лабораторной работе рассмотрен пакет логирования Winston – это универсальная библиотека и самое популярное средство для логирования приложений Node.js, основанное на статистике загрузки NPM [8].

Логирование с использованием библиотек Morgan и Winston

Создайте js-проект и установите необходимые зависимости:

1) Express.js – для работы веб-сервера;
2) Morgan – для регистрации HTTP-запросов и ошибок;
3) Winston – для регистрации логов и настройки их хранения.

Листинг кода 14.1

```
npm i express morgan winston
```

После установки создайте файл server.js для работы.

Листинг кода 14.2

```
const express = require("express");
const morganMiddleware = require("./middlewares/morgan.middleware");

// Промежуточному программному обеспечению morgan это не нужно.
// Это для ведения журнала вручную
const logger = require("./utils/logger");

const app = express();

// Добавьте промежуточное программное обеспечение morgan
app.use(morganMiddleware);

app.get("/api/status", (req, res) => {
    logger.info("Проверка статуса API: все в порядке");
    res.status(200).send({
        status: "Запущен",
        message: "API запущен и работает!"
    });
});

// Запуск сервера
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    logger.info(`Сервер запущен на порту ${PORT}`);
});
```

Создайте папку utils, в ней – файл logger.js для настройки Winston: регистрации уровней серьезности ответов сервера в средах разработки и производства с помощью разных цветов для каждого уровня (сообщения журнала будут распечатаны в консоли и сохранены в файлах), типа передачи (в консоль, файл или и в то и другое).

Листинг кода 14.3

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

const levels = {
  error: 0,
  warn: 1,
  info: 2,
  http: 3,
  debug: 4,
}

const level = () => {
  const env = process.env.NODE_ENV || 'development'
  const isDevelopment = env === 'development'
  return isDevelopment ? 'debug' : 'warn'
}

const colors = {
  error: 'red',
  warn: 'yellow',
  info: 'green',
  http: 'magenta',
  debug: 'white',
}
winston.addColors(colors)

const format = winston.format.combine(
  winston.format.timestamp({    format:    'YYYY-MM-DD
HH:mm:ss:ms' }),
  winston.format.colorize({ all: true }),
  winston.format.printf(
    (info) => `${info.timestamp} ${info.level}: ${in-
fo.message}`,
  ),
)

const transports = [
  new winston.transports.Console(),
  new winston.transports.File({
    filename: 'logs/error.log',
    level: 'error',
  }),
```

```
      new       winston.transports.File({       filename:
'logs/all.log' }),
     ]

     const logger = winston.createLogger({
       level: level(),
       levels,
       format,
       transports,
     })

     module.exports = logger
```

Создайте папку middleware, в ней – файл morgan.middleware.js для настройки Morgan: указывается регистратор, который создали по умолчанию вместо console.log; определяется формат сообщения, который должно использовать промежуточное программное обеспечение Morgan, когда оно обрабатывает запросы и пытается записывать в журналы.

Листинг кода 14.4

```
     const morgan = require("morgan");
     const logger = require("../utils/logger");

     const stream = {
         // Используйте строгость http
         write: (message) => logger.http(message),
     };

     const skip = () => {
         const  env  =  process.env.NODE_ENV  ||  "develop-
ment";
         return env !== "development";
     };

     const morganMiddleware = morgan(
         ":remote-addr  :method  :url  :status  :res[content-
length] - :response-time ms",
         { stream, skip }
     );

     module.exports = morganMiddleware;
```

Запустите приложение. После запуска будет создана папка logs с файлами all.log (в этом файле печатаются все сообщения о работе сервера) и error.log (в этом файле печатаются все сообщения об ошибках). Протестируйте его в Postman. На рис. 14.1 – 14.3 представлены этапы работы в Postman.

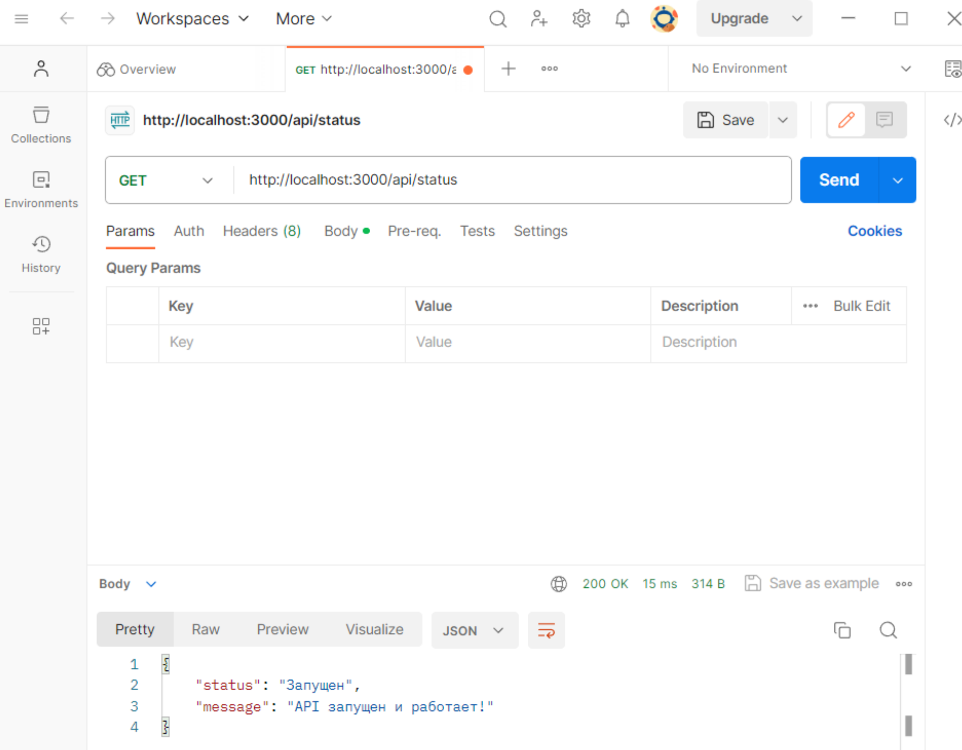


Рис. 14.1. Тестирование в Postman

```
[nodemon] starting `node server.js`
2023-11-29 12:22:40:2240 info: Сервер запущен на порту 3000
2023-11-29 12:22:45:2245 info: Проверка статуса API: Все в порядке
2023-11-29 12:22:45:2245 http: ::1 GET /api/status 200 79 - 3.729 ms
```

Рис. 14.2. Вывод сообщений в консоль

```
server.js   package.json   morgan.middleware.js   all.log   logger.js
logs > all.log
1  2023-11-29 12:22:40:2240 ESC[32minfoESC[39m: ESC[32mСервер запущен на порту 3000ESC[39m
2  2023-11-29 12:22:45:2245 ESC[32minfoESC[39m: ESC[32mПроверка статуса API: Все в порядкеESC[39m
3  2023-11-29 12:22:45:2245 ESC[35mhttpESC[39m: ESC[35m::1 GET /api/status 200 79 - 3.729 msESC[39m
4  ESC[35mESC[39m
5
```

Рис. 14.3. Список логов, добавленных в файл all.log

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).
2. Добавьте в свой проект логирование.
3. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое логирование?
2. Для чего нужна библиотека Winston?
3. Для чего нужна библиотека Morgan?
4. Какие уровни логирования поддерживает Winston?
5. Как настроить запись логов в файл в Winston?

Лабораторная работа № 15. EMAIL-РАССЫЛКА

Цель работы: отправка уведомлений на электронную почту с помощью библиотеки nodemailer.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Одна из распространённых функций веб-приложений – отправка email-писем пользователям. Использовать их можно для разных целей, например сообщения о новостях, акциях или подтверждения адреса электронной почты и активации учётной записи (чаще всего). Для того чтобы отправить письмо, необходим сервис электронной почты, а также библиотека, которая может с ним работать. В рамках лабораторной работы в качестве сервиса будем использовать mail.ru. (лучше использовать Mail или Yandex из-за простоты авторизации).

Отправка сообщений на почту с помощью библиотеки nodemailer

За основу возьмём бэкэнд-приложение, созданное в рамках лабораторной работы № 12.

Установите библиотеку nodemailer с помощью команды.

Листинг кода 15.1

```
npm install nodemailer --save
```

В папке src создайте папку service, в ней – папку mail, внутри которой – файл mailer.service.js

Листинг кода 15.2

```
const nodemailer = require('nodemailer');
let transporter = nodemailer.createTransport({
    host: 'smpt.mail.ru',
    port: '465',
    secure: true,
    auth: {
      user: '*******,
      pass: '*******',
```

```
        }
      }, {
        from: 'Test App <vlsu.test@mail.ru>'
      })
    sendTestMail = (email) => {
        transporter.sendMail({
            to: email,
            subject: 'test letter',
            text: 'Hello world',
            html: `<h1>Тестовое письмо</h1>
            <i>Здравствуйте, ${email}</i>
            <p>Учимся отправлять письма с node.js</p>`
        })
    }
    const MailService = {
        sendTestMail: sendTestMail
    }
    module.exports = MailService;
```

Далее в server.js добавьте следующий код.

Листинг кода 15.3

```
    *****
    const            MailService            =            re-
quire("./service/mail/mailer.service")
    *****
    MailService.sendTestMail('*****@yandex.ru');
    app.listen(PORT, () => {
      console.log(`Server is running on port ${PORT}.`);
    });
```

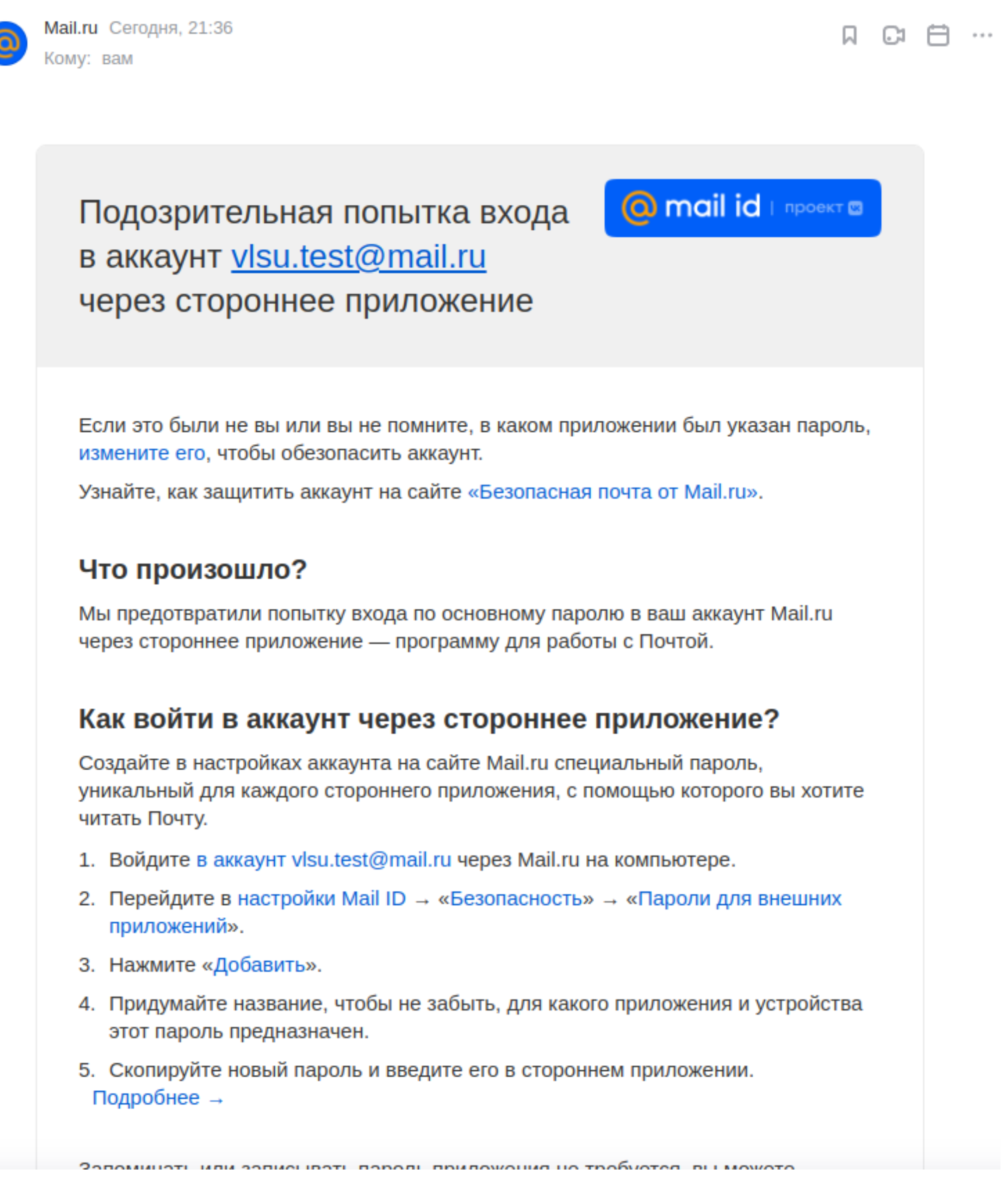
Вместо звёздочек укажите адрес, с которого нужно отправлять письма, пароль от почты, а также адрес, на который нужно отправить письмо.

Запустите приложение. Если в консоли будет ошибка (рис. 15.1), необходимо перейти по ссылке из ошибки.

```
.js:749:14)
    at TLSSocket.SMTPConnection._onSocketData (/home/freeroed/develop/js/auth/back/node_modules/nodemailer/lib/smtp-
connection/index.js:189:44)
    at TLSSocket.emit (node:events:520:28)
    at addChunk (node:internal/streams/readable:315:12)
    at readableAddChunk (node:internal/streams/readable:289:9)
    at TLSSocket.Readable.push (node:internal/streams/readable:228:10) {
  code: 'EAUTH',
  response: '535 Authentication failed. Please verify your account by going to https://e.mail.ru/login?email=vlsu.te
st@mail.ru',
  responseCode: 535,
  command: 'AUTH PLAIN'
```

Рис. 15.1. Возможная ошибка при запуске приложения

Другой вариант создания уникального пароля – открыть письмо (рис. 15.2), которое придёт на почтовый адрес сразу после попытки отправки (выбрать «Добавить»).



Mail.ru Сегодня, 21:36
Кому: вам

Подозрительная попытка входа в аккаунт vlsu.test@mail.ru через стороннее приложение

mail id | проект

Если это были не вы или вы не помните, в каком приложении был указан пароль, измените его, чтобы обезопасить аккаунт.

Узнайте, как защитить аккаунт на сайте «Безопасная почта от Mail.ru».

Что произошло?

Мы предотвратили попытку входа по основному паролю в ваш аккаунт Mail.ru через стороннее приложение — программу для работы с Почтой.

Как войти в аккаунт через стороннее приложение?

Создайте в настройках аккаунта на сайте Mail.ru специальный пароль, уникальный для каждого стороннего приложения, с помощью которого вы хотите читать Почту.

1. Войдите в аккаунт vlsu.test@mail.ru через Mail.ru на компьютере.
2. Перейдите в настройки Mail ID → «Безопасность» → «Пароли для внешних приложений».
3. Нажмите «Добавить».
4. Придумайте название, чтобы не забыть, для какого приложения и устройства этот пароль предназначен.
5. Скопируйте новый пароль и введите его в стороннем приложении.

Подробнее →

Рис. 15.2. Письмо, которое пришло на почту

Укажите название приложения (рис. 15.3).

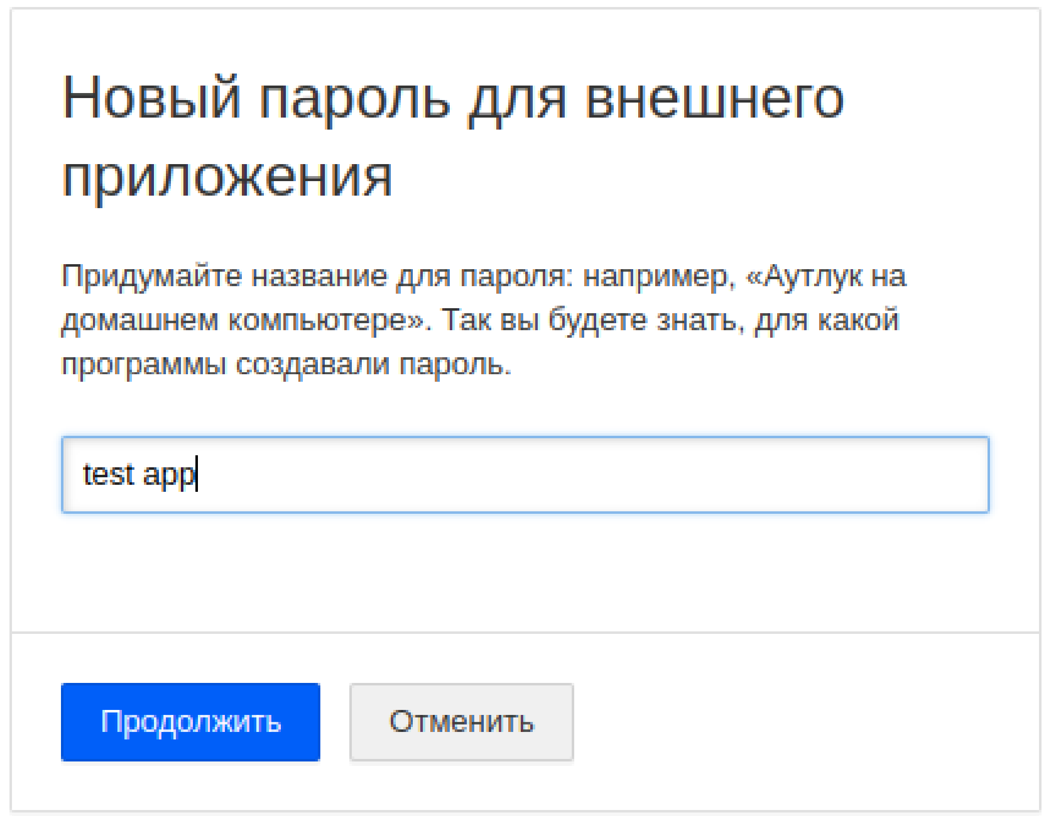
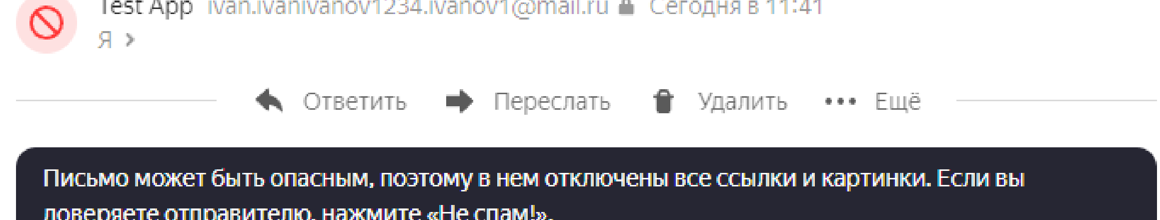


Рис. 15.3. Добавление пароля для доступа к почте для внешнего приложения

Подтвердите пароль от почты. В результате получаем уникальный пароль специально для приложения. Укажите его в параметре pass.

Запустите приложение. После успешного запуска проверьте почту получателя. Возможно, письмо попало в папку «Спам». Результат работы представлен на рис. 15.4.

test letter



Тестовое письмо

Здравствуйте, alina26.09.2002@yandex.ru

Учимся отправлять письма с node.js

Рис. 15.4. Отправленное с помощью приложения сообщение на почту

Доработайте метод отправки для того, чтобы в логи выводились сообщения о результате отправки (рис. 15.5).

Листинг кода 15.4

```
     sendTestMail = (email) => {
       transporter.sendMail({
         to: email,
         subject: 'test letter',
         text: 'Hello world',
         html: `<h1>Тестовое письмо</h1>
             <i>Здравствуйте, ${email}</i>
             <p>Учимся отправлять письма с node.js</p>`
       }).then(() => console.info("Письмо успешно отправ-
лено на адрес: ", email))
         .catch(err => console.warn("Произошла ошибка при
отправке сообщения: ", err))
     }
```

```
Письмо успешно отправлено на адрес:  alina26.09.2002@yandex.ru
```

Рис. 15.5. Вывод логов при отправке сообщений

Порядок выполнения работы

1. Изучите теоретический материал. Выполните примеры (см. листинги).

2. Доработайте код таким образом, чтобы при регистрации на почту отправлялось письмо о том, что пользователь зарегистрирован.

3. Составьте отчет по результатам работы.

Контрольные вопросы

1. Какую библиотеку вы использовали при работе?

2. Из каких элементов состоит запрос на отправку сообщения на почту?

3. Какие данные нужно ввести, чтобы отправить сообщение на почту?

4. Как настроить SMTP-сервер для отправки писем через nodemailer?

5. Как обработать ошибку при отправке email?

Лабораторная работа № 16. РАЗВЕРТЫВАНИЕ ВЕБ-ПРИЛОЖЕНИЯ

Цель работы: создание docker-контейнеров.

Формируемые компетенции: ОК 02, ОК 03, ОК 10, ПК 9.2, ПК 9.5.

Общие сведения

Docker – это проект с открытым исходным кодом для автоматизации развертывания приложений в виде переносимых автономных контейнеров, выполняемых в облаке или локальной среде.

Контейнер – это изолированная среда для кода. Контейнеру не известно об операционной системе или файлах пользователя. Он работает в среде, предоставленной Docker Desktop. В контейнерах есть все, что нужно коду для запуска, вплоть до базовой операционной системы.

В поставку Docker входят следующие компоненты:

– Docker host – это операционная система, на которую устанавливают Docker и на которой он работает.

– Docker daemon – служба, которая управляет Docker объектами: сетями, хранилищами, образами и контейнерами.

– Docker client – консольный клиент, при помощи которого пользователи взаимодействуют с Docker daemon и отправляют ему команды, создают контейнеры и управляют ими.

– Docker image – это неизменяемый образ, из которого разворачивается контейнер.

– Docker container – развернутое и запущенное приложение.

– Docker Registry – репозиторий, в котором хранятся образы.

– Dockerfile – файл-инструкция для сборки образа.

– Docker Compose – инструмент для управления несколькими контейнерами, позволяет создавать контейнеры и задавать их конфигурацию.

– Docker Desktop – GUI-клиент, который распространяется по GPL. Бесплатная версия работает на Windows, macOS, а с недавних пор и на Linux. Это очень удобный клиент, который отображает все

сущности Docker и позволяет запустить однокодовую систему Kubernetes для компьютера.

Docker изначально создавался под Linux. Поэтому на Windows и macOS запускают виртуальную машину с Linux, а поверх неё – Docker. В macOS используют VirtualBox, а в Windows – Hyper-V.

Docker работает с несколькими сущностями:

– Docker image (образ) – шаблон, по которому создают контейнеры. Его часто сравнивают со слоёным пирогом: слой файловой системы накладывается поверх слоя базового образа и получается неизменяемый образ, в который можно установить приложение, конфигурации и зависимости. Другие образы могут наследоваться, поэтому если наложить сверху слой файлов и «закоммитить», то получим ещё один неизменяемый образ.

– Dockerfile. Если Docker image – это «пирог», то Dockerfile – «рецепт» его приготовления. В этом файле описаны основные инструкции для сборки образа: какой базовый образ взять, откуда и куда положить файлы и т. д.

– Контейнер – это runtime-сущность на основе образа, приложение, которое разворачивают с помощью Docker. Можно провести такую аналогию: образ – это инсталлятор программы, а контейнер – уже запущенная программа. При развёртывании контейнера поверх файловой системы создаётся ещё один изменяемый слой. Приложение внутри контейнера может записывать туда данные или редактировать их. После удаления контейнера данные стираются, но их можно сохранить с помощью области хранения volumes.

– Docker Registry – это репозиторий, в котором хранятся Docker-образы, может быть локальным и публичным. Репозитории создают на платформах вроде Docker Hub и GitLab и размещают в них образы с описанием, разными версиями и тегами [9].

Развертывание приложения с помощью docker

Установите docker на компьютер (инструкцию по установке можно найти на официальном сайте https://docs.docker.com/). Основа выполнения заданий – лабораторная работа № 12.

Скачайте образ node с помощью команды. Все образы можно найти на официальном сайте (https://hub.docker.com/).

Листинг кода 16.1

```
docker pull node
```

Проверьте существующие образы на компьютере.

Листинг кода 16.2

```
docker images
```

Результат выполнения команды представлен на рис. 16.1.

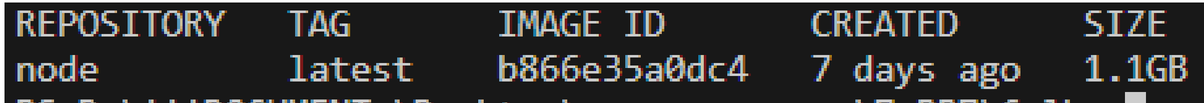


Рис. 16.1. Проверка существующих образов

Добавьте в корневую папку файл Dockerfile с кодом, где описаны инструменты для создания контейнера.

Листинг кода 16.3

```
FROM node:20

WORKDIR /app

COPY ./ ./

RUN npm i

CMD [ "npm", "start" ]
```

Добавьте в корневую папку файл .dockerignore с кодом, где пропишите, что будет игнорироваться при запуске контейнера.

Листинг кода 16.4

```
node_modules/
Dockerfile
```

Добавьте в корневую папку файл docker-compose.yml с кодом, где пропишите создание нескольких контейнеров и их конфигурацию.

Листинг кода 16.5

```
version: '3.9'

services:
  server:
    build: .
    ports:
      - '3000:3000'
  db:
    image: 'postgres'
    environment:
      POSTGRES_PASSWORD: '123456'
      POSTGRES_USER: 'docker'
    volumes:
      - 'data:/var/lib/postgresql/data'

volumes:
  data:
```

Один контейнер создается с помощью образа, другой – с помощью файла, который был создан ранее.

Итак, создан контейнер для БД, где пользователь docker и пароль 123456.

Теперь измените файл в проекте, который подключается к БД (knexfile.js).

Листинг кода 16.6

```
/**
 * @type { Object.<string, import("knex").Knex.Config> }
 */
module.exports = {
  client: 'pg',
  connection: {
    host: 'db',
```

```
    user: 'docker', //Логин для подключения к БД
    password: '123456', //Пароль
    database: 'docker' //Название БД
  },
  migrations: {
    directory: '../migrations'
  },
  seeds: {
    directory: '../seeds'
  }
};
```

Создайте образ проекта.

Листинг кода 16.7

```
docker-compose up -d
```

Если в файлах проекта было что-то изменено, то можно использовать следующую команду.

Листинг кода 16.8

```
docker-compose up --build -d
```

Проверьте, создались ли контейнеры (рис. 16.2).

Листинг кода 16.9

```
docker ps
```

```
CONTAINER ID   IMAGE        COMMAND                  CREATED          STATUS          PORTS                    NAMES
43858033e8a3   postgres     "docker-entrypoint.s…"   16 seconds ago   Up 14 seconds   5432/tcp                 61b-db-1
06915ce5d769   61b-server   "docker-entrypoint.s…"   16 seconds ago   Up 14 seconds   0.0.0.0:3000->3000/tcp   61b-server-1
```

Рис. 16.2. Проверка созданных контейнеров

С помощью приведенной ниже команды можно посмотреть, какие контейнеры запущены.

Листинг кода 16.10

```
docker ps -a
```

Запустите миграцию.

Листинг кода 16.11

```
docker exec 6lb-server-1 npm run migrate
```

Запустите сидирование.

Листинг кода 16.12

```
docker ps -a
```

Проверьте, запустился ли проект. Откройте его в браузере (http://localhost:3000/), проверьте регистрацию и авторизацию.

Результат работы представлен на рис. 16.3 – 16.6.

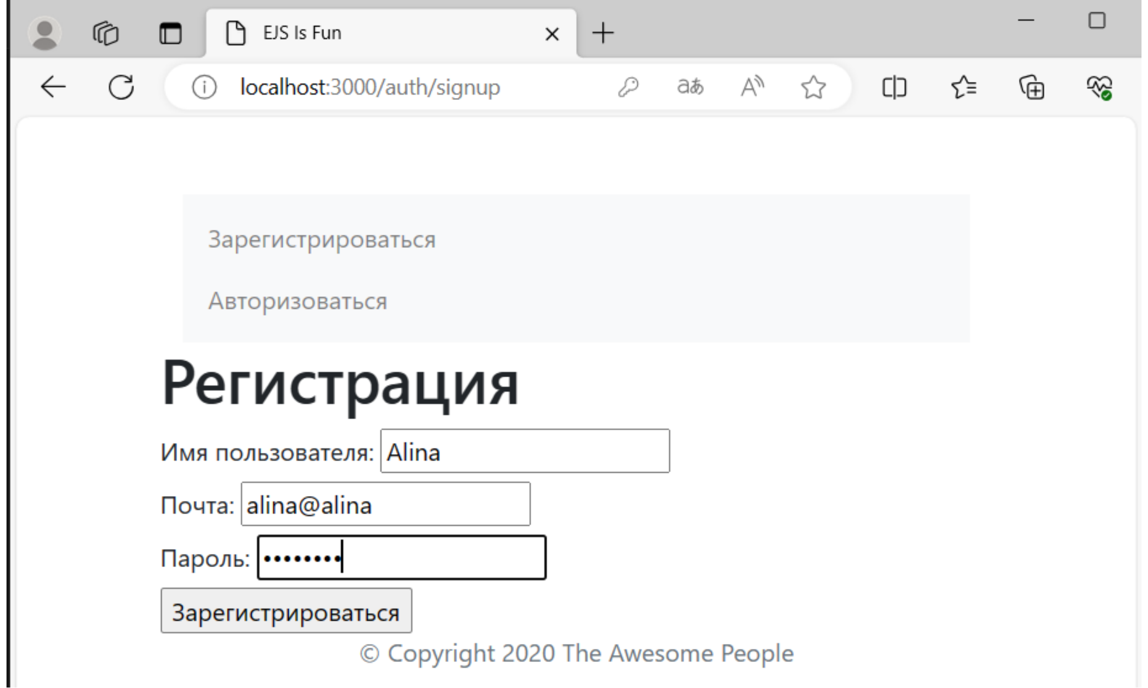


Рис. 16.3. Ввод данных для регистрации в проект, развернутый через docker

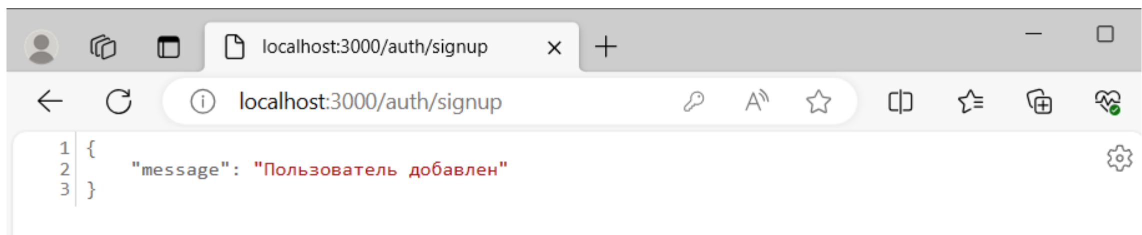


Рис. 16.4. Ответ от сервера, развернутого через docker (при регистрации)

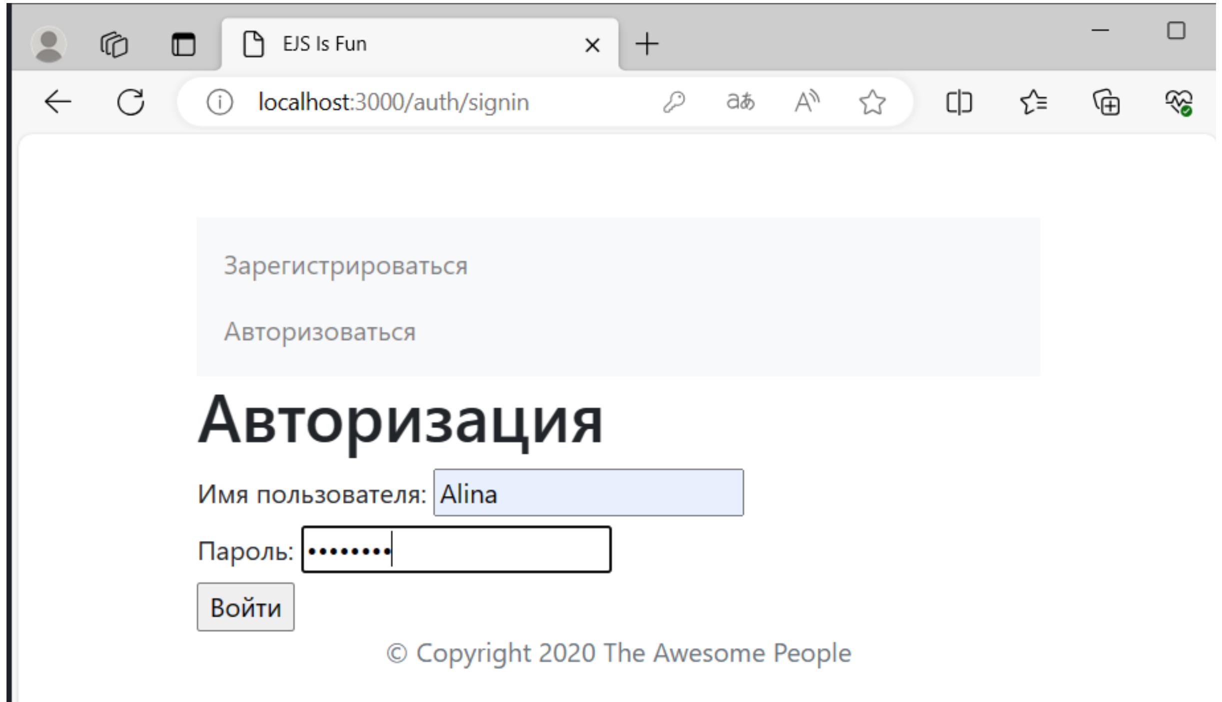


Рис. 16.5. Ввод данных для авторизации в проект, развернутый через docker

localhost:3000/auth/signin


```
{
    "id": 1,
    "username": "Alina",
    "email": "alina@alina",
    "roles": [
        "ROLE_USER",
        "ROLE_ADMIN"
    ],
    "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzAyNDc0MDkyLCJleHAiOjE3MDI1NjA0OTJ9
.-HYDeGo8Q1A8ITM41biDgvrS-WqjrRBiaJkNTTvVfso"
}
```


Рис. 16.6. Ответ от сервера, развернутого через docker (при авторизации)

Остановите проект с помощью следующей команды.

Листинг кода 16.13

```
docker-compose down
```

После остановки проекта обычно удаляются все таблицы в БД, однако этого не произошло, так как прописано, что при создании контейнера для БД данные берутся из файла.

Листинг кода 16.14

```
      volumes:
        - 'data:/var/lib/postgresql/data'
```

Проверьте, создан ли файл data.

Листинг кода 16.15

```
docker volume ls
```

Результат выполнения команды представлен на рис. 16.7.

```
DRIVER    VOLUME NAME
local     3c6782614244b0b7b90de336ce1e4b0217c54110e06c956faeedbcf19e269c4b
local     3d78974a9c6e11550dc107d15ad17d88e81cf9df36112693a980657c0bab4ba1
local     4fab7d0b4a4389ae27f470c98d436a3bbd24794f6f21d369b99b4cae42f0f592
local     61b_data
local     7a6ff3d9cf60297e78e6741e65126d1b4d25b0a8a53c3b6c726c15390ad56ef0
local     7f8e9d1f5a1fe7fb485afc05f4fef433fcac3ee56e8f7d895871a92e14db9879
local     64df89a459df8f318d3e74fbfc3ccf9cfcf643e36a0194e11f4f331b751c3b61
local     84e472a14a5a9eac1084d9b6e8516237337954bbb883f85cb821f56e4eb4b14c
local     094cb98a44f2afe880e563e3aa372f0e5b3ac46f93943b2f3b6da11db069d4a0
local     bd5648c1320065659e840fc5388524efb57b15ce61eeec71f7292d3d5a5aaa39
local     d06cda612d85185e43be382a3baa75b5a10f4c0f683e8b7b1db07ade63b73488
```

Рис. 16.7. Проверка создания файла, который хранит в себе БД

Теперь можно заново запустить проект и авторизоваться. Результат работы представлен на рис. 16.8 – 16.9.

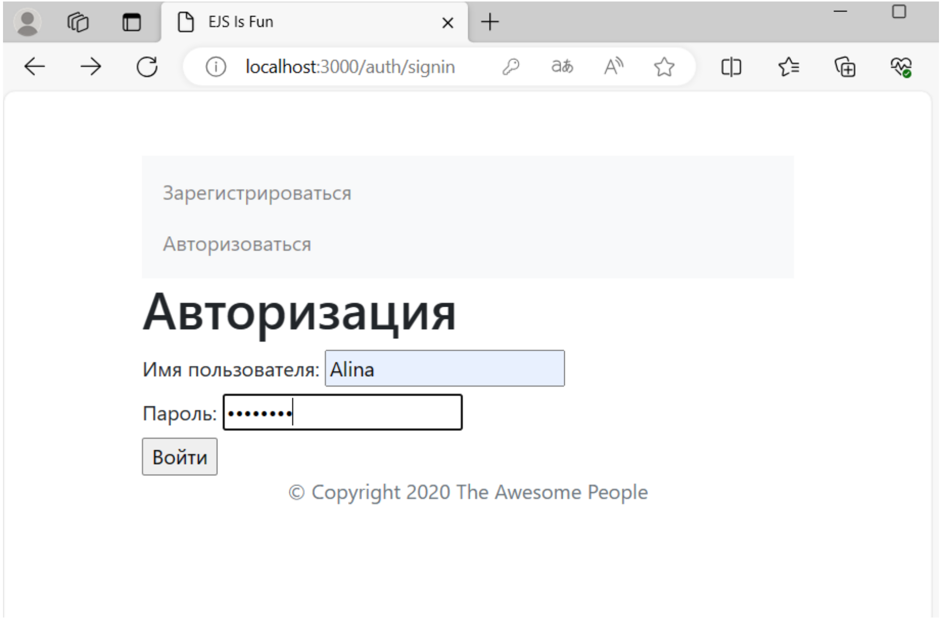


Рис. 16.8. Ввод данных в проект, развернутый через вновь запущенный docker

localhost:3000/auth/signin


```
1 {
2     "id": 1,
3     "username": "Alina",
4     "email": "alina@alina",
5     "roles": [
6         "ROLE_USER",
7         "ROLE_ADMIN"
8     ],
9     "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzAyNDc0Nzc2LCJleHAiOjE3MDI1NjExNzZ9
.HGfteJh-yvYRXuum_irSEbVufcx4mK8RDoIxF1jE0Vk"
10 }
```


Рис. 16.9. Ответ от сервера, развернутого через вновь запущенный docker

Порядок выполнения работы

1. Изучите настройку веб-сервера с помощью nginx (самостоятельно).
2. Изучите библиотеку cluster (самостоятельно).
3. Настройте сервер с помощью nginx.
4. Добавьте на сервер библиотеку cluster.
5. Разверните свое приложение с помощью docker (предоставить скриншоты и листинг).
6. Составьте отчет по результатам работы.

Контрольные вопросы

1. Что такое docker?
2. Что такое контейнеры?
3. Какие файлы были созданы в ходе работы и зачем?
4. Какие сущности есть в docker?
5. Какие компоненты существуют в docker?
6. Что такое nginx?
7. Что такое cluster?

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторных работ по дисциплине «Проектирование и разработка веб-приложений» студенты всесторонне изучают современные подходы и инструменты создания серверной части веб-приложений с использованием платформы Node.js.

Практикум охватывает полный цикл разработки – от базовых основ работы в среде Node.js и создания простейших HTTP/HTTPS серверов до углубленного изучения популярных фреймворков (Express.js), шаблонизаторов (pug, ejs) и методов взаимодействия с базами данных. Особое внимание уделено практическому применению таких технологий, как миграции (knex), объектно-реляционное отображение (Sequelize), а также реализации ключевых функций современных веб-приложений: авторизация и регистрация пользователей с применением JWT, загрузка файлов (multer), логирование событий (Winston, Morgan) и отправка электронных писем (nodemailer).

Использование асинхронной архитектуры Node.js в сочетании с такими библиотеками, как Express.js, knex и Sequelize, позволяет разрабатывать высокопроизводительные приложения, способные эффективно обрабатывать множество одновременных запросов и взаимодействовать с различными СУБД. В практикуме продемонстрирована важность не только написания кода, но и сопутствующих процессов: обеспечения безопасности (шифрование паролей, HTTPS), валидации данных, логирования для отладки и мониторинга, а также автоматизации развертывания с помощью docker.

Представленные лабораторные работы направлены на формирование профессиональных компетенций (ПК 9.2, ПК 9.4, ПК 9.5 и др.) в соответствии с требованиями ФГОС СПО по направлениям подготовки «Информационные системы и программирование» и «Веб-разработка» и овладение востребованными инструментами и технологиями для проектирования и разработки серверной части веб-приложений на языке JavaScript.

Для углубления знаний и закрепления полученных навыков рекомендуется:

– использовать полученные компетенции при работе над курсовыми и дипломными проектами, интегрируя серверную часть с клиентскими приложениями (например, на React или Vue);

– изучить дополнительные материалы, выходящие за рамки практикума, такие как тестирование серверных приложений, работа с WebSocket для создания приложений, а также облачные технологии;

– активно применять системы контроля версий (Git) для управления кодом проектов, разработанных в ходе выполнения заданий.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Справочник Node.js [Электронный ресурс]. URL: https://nodejsdev.ru/ (дата обращения: 12.08.2025).

2. Что такое протокол HTTPS и принципы его работы [Электронный ресурс]. URL: https://help.reg.ru/support/ssl-sertifikaty/obshchaya-informatsiya-po-ssl/chto-takoye-protokol-https-i-printsipy-yego-raboty#3 (дата обращения: 12.08.2025).

3. Обработка ошибок в Express [Электронный ресурс]. URL: https://habr.com/ru/companies/ruvds/articles/ 476290/3 (дата обращения: 12.08.2025).

4. Магомедов З. Html шаблонизатор Pug, документация на русском [Электронный ресурс]. URL: https://habr.com/ru/companies/ruvds/articles/ 476290/3 (дата обращения: 12.08.2025).

5. ESJ [Электронный ресурс]. URL: https://ejs.co/ (дата обращения: 12.08.2025).

6. Как использовать EJS для моделирования вашего приложения узлов [Электронный ресурс]. URL: https://www.digitalocean.com/ community/tutorials/how-to-use-ejs-to-template-your-node-application-fr (дата обращения: 12.08.2025).

7. Гибкая ORM для Node.js – Sequelize Proglib [Электронный ресурс]. URL: https://proglib.io/p/gibkaya-orm-dlya-node-js-sequelize-2022-10-12 (дата обращения: 12.08.2024).

8. Логирование приложения NODE.JS с помощью WINSTON 8HOST [Электронный ресурс]. URL: https://www.8host.com/blog/ logirovanie-prilozheniya-node-js-s-pomoshhyu-winston/ (дата обращения: 12.08.2025).

9. Как работает Docker: подробный гайд от техлида Skillbox [Электронный ресурс]. URL: https://skillbox.ru/media/code/kak-rabotaet-docker-podrobnyy-gayd-ot-tekhlida/ (дата обращения: 12.08.2025).

РЕКОМЕНДАТЕЛЬНЫЙ БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Справочник Node.js [Электронный ресурс]. – URL: https://nodejsdev.ru/ (дата обращения: 12.08.2025).

2. Docker: Accelerated Container Application Development [Электронный ресурс]. – URL: https://www.docker.com/ (дата обращения: 12.08.2025).

3. TypeORM – Amazing ORM for TypeScript and JavaScript (ES7, ES6, ES5). Supports MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, WebSQL databases. Works in NodeJS, Browser, Ionic, Cordova and Electron platforms [Электронный ресурс]. – URL: https://typeorm.io/ (дата обращения: 12.08.2025).

4. Модуль nginx JavaScript [Электронный ресурс]. – URL: https://nginx.org/ru/docs/njs/index.html (дата обращения: 12.08.2025).

Учебное электронное издание

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА
ВЕБ-ПРИЛОЖЕНИЙ НА СТОРОНЕ СЕРВЕРА

Лабораторный практикум

Автор-составитель
МАКСИМОВА Алина Станиславовна

Редактор Е. А. Лебедева
Верстка Л. В. Макаровой
Рецензент А. И. Петрова
Выпускающий редактор А. А. Амирсейидова

Системные требования: Intel от 1,3 ГГц; Windows XP/7/8/10;
Adobe Reader; дисковод CD-ROM.

Тираж 9 экз.

Издательство Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.