

МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В.И. Ульянова (Ленина)

С. А. БЕЛЯЕВ

Лабораторный практикум по дисциплине Web- технологии

Современные фреймворки

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2022

УДК 004.432
ББК 3 988.02–018я7
Б49

Беляев С. А.

Б49 Лабораторный практикум по дисциплине Web-технологии. Современные фреймворки. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2022. 83 с.

ISBN 978-5-7629-2409-2

Представлены материалы по дисциплине «Web-технологии». Рассматриваются вопросы разработки web-приложений, начиная со статических страниц с постепенным переходом к динамическим приложениям с использованием современных фреймворков. Приводятся базовые элементы языков программирования и вопросы для самоконтроля.

Предназначено для студентов направлений «Программная инженерия» и «Прикладная математика и информатика».

УДК 004.432
ББК 3 988.02–018я7

Рецензенты: АО «Научно-инженерный центр СПб ЭТУ» (канд. воен. наук, доцент А. И. Вайнтрауб); канд. техн. наук, доцент Е. Н. Шаповалов (АО «НИИ ПС»).

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия

ISBN 978-5-7629-2409-2

© СПбГЭТУ «ЛЭТИ», 2022

Лабораторный практикум содержит методические указания к лабораторным работам по дисциплине «Web-технологии». Перечень лабораторных работ соответствует рабочей программе дисциплины и включает в себя:

- тетрис на JavaScript;
- REST-приложение управления библиотекой;
- модуль администрирования приложения «Социальная сеть»;
- модуль пользователя приложения «Социальная сеть»;
- модуль администрирования приложения «Биржа акций»;
- модуль приложения «Покупка и продажа акций».

Информационные технологии (операционные системы, программное обеспечение общего и специализированного назначения, информационные справочные системы) и материально-техническая база, используемые при осуществлении образовательного процесса по дисциплине, соответствуют требованиям федерального государственного образовательного стандарта высшего образования. В результате изучения студенты получают необходимые навыки, закрепят теоретический материал лекций и приобретут требуемые компетенции.

Лабораторная работа 1. ТЕТРИС НА JAVASCRIPT

Цель и задачи

Целью работы является изучение работы web-сервера nginx со статическими файлами и создание клиентских JavaScript web-приложений.

Для достижения поставленной цели требуется решить следующие задачи:

- генерация открытого и закрытого ключей для использования шифрования (<https://www.openssl.org/>);
- настройка сервера nginx для работы по протоколу HTTPS;
- разработка интерфейса web-приложения;
- обеспечение ввода имени пользователя;
- обеспечение создания новой фигуры для тетриса по таймеру и ее движение;
- обеспечение управления пользователем падающей фигурой;
- обеспечение исчезновения ряда, если он заполнен;
- по окончании игры – отображение таблицы рекордов, которая хранится в браузере пользователя.

Основные теоретические сведения

Асимметричные ключи используются в асимметричных алгоритмах шифрования и являются ключевой парой. Закрытый ключ известен только владельцу. Открытый ключ может быть опубликован и используется для проверки подлинности подписанного документа (сообщения). Открытый ключ вычисляется, как значение некоторой функции от закрытого ключа, но знание открытого ключа не дает возможности определить закрытый ключ. По секретному ключу можно вычислить открытый ключ, но по открытому ключу практически невозможно вычислить закрытый ключ.

nginx (<https://nginx.ru/ru/>) – веб-сервер, работающий на Unix-подобных операционных системах и в операционной системе Windows.

JavaScript (<https://developer.mozilla.org/ru/docs/Web/JavaScript>) – язык программирования, он поддерживает объектно-ориентированный и функциональный стили программирования. Является реализацией стандарта ECMAScript.

Общая формулировка задачи

Необходимо создать web-приложение – игру в тетрис. Основные требования:

- сервер – nginx, протокол взаимодействия – HTTPS версии не ниже 2.0;
- отображается страница для ввода имени пользователя с использованием HTML-элементов `<input>`;
- статическая страница отображает «стакан» для тетриса с использованием HTML-элемента `<canvas>`, элемент `<div>` используется для отображения следующей фигуры, отображается имя пользователя;
- фигуры в игре – классические фигуры тетриса (7 шт. тетрамино);
- случайным образом генерируется фигура и начинает падать в «стакан» (описание правил см., например, <https://ru.wikipedia.org/wiki/Тетрис>);
- пользователь имеет возможность двигать фигуру влево и вправо, повернуть на 90° и «уронить»;
- если собралась целая «строка», она должна исчезнуть;
- при наборе некоторого заданного числа очков увеличивается уровень, что заключается в увеличении скорости игры;
- пользователь проигрывает, когда стакан «заполняется», после чего ему отображается локальная таблица рекордов;

– вся логика приложения написана на JavaScript.

Необязательно: оформление с использованием CSS.

Постарайтесь сделать такую игру, в которую вам будет приятно играть. Помните, когда-то эта игра была хитом! Преимуществом будет использование звукового сопровождения событий: падение фигуры, исчезновение «строки».

Описание последовательности выполнения работы

Подготовка среды выполнения для всех лабораторных работ. Выполнение лабораторных работ будет осуществляться с использованием среды разработки JetBrains WebStorm (студентам предоставляется бесплатная лицензия) – <https://www.jetbrains.com/webstorm/>.

Разработка приложений будет осуществляться с использованием двух языков программирования: JavaScript и TypeScript. По умолчанию будет использоваться JavaScript, по отдельному указанию – TypeScript.

В качестве среды выполнения для JavaScript в ОС Windows следует использовать Node.JS (скачать LTS-версию по адресу <https://nodejs.org/en/>), в ОС Linux использовать Node.JS версии не ниже 12 (инструкция доступна по адресу <https://nodejs.org/en/download/package-manager/>, основная команда для Ubuntu – `sudo apt-get install nodejs`).

В качестве среды выполнения для TypeScript (<https://www.typescriptlang.org/>) в ОС Windows и ОС Linux следует использовать команду установки (в корне проекта).

```
npm install -g typescript
```

При этом все загруженные модули попадают в папку «node-modules» проекта. При отправке исходных кодов для проверки преподавателем папка «node-modules» не отправляется, вся информация об использованных npm-модулях должна быть корректно записана в `package.json`.

Генерация открытого и закрытого ключей для использования шифрования. Для создания открытого и закрытого ключей в операционной системе Linux доступна следующая программа:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout  
example.key -out example.csr
```

В результате будут созданы два ключа:

- 1) `example.key` – закрытый ключ;
- 2) `example.csr` – открытый ключ.

Настройка сервера nginx для работы по протоколу HTTPS. Web-сервер nginx под ОС Windows можно скачать по адресу <https://nginx.ru/ru/>, под ОС Linux можно воспользоваться командой

```
sudo apt-get install nginx
```

В конфигурационном файле (Windows – conf/nginx.conf, Linux – /etc/nginx/sites-available/default) в минимальном варианте необходимо указать:

```
listen 443 ssl http2;  
ssl_certificate /etc/ssl/certs/example.com.nginx.crt;  
ssl_certificate_key /etc/ssl/private/example.com.nginx.key;
```

Для корректной работы файлы с ключами должны быть скопированы в соответствующие директории с учетом используемой ОС.

Разработка интерфейса пользователя. Возможный вариант формы входа приведен на рис. 1.1.

На рис. 2 представлены пять основных тетрамино. Недостающие два получаются зеркальным отражением несимметричных фигур.

Игра Tetris

Введите имя игрока

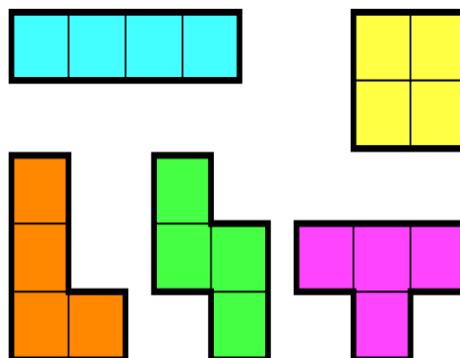


Рис. 1.1. Форма входа

Рис. 1.2. Тетрамино

Возможный вариант основной формы приведен на рис. 1.3.

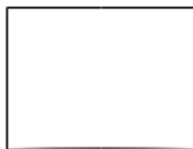
На форме рекордов должна быть ссылка или кнопка перехода на форму входа. Важно, что при переходе на форму входа в ней уже должно быть подставлено последнее введенное имя пользователя.

Игра Tetris

Игрок: Player

Текущий уровень: 1

Следующая фигура



Клавиши для управления:

Стрелка влево - перемещение фигуры влево

Стрелка вправо - перемещение фигуры вправо

Пробел / стрелка вниз - "уронить" фигуру

Стрелка вверх - повернуть фигуру

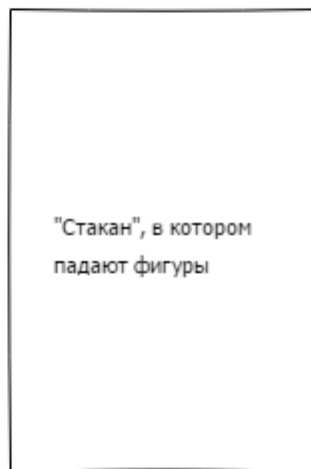


Рис. 1.3. Основная форма игры

Обеспечение ввода имени пользователя. Ввод имени пользователя может выполняться на странице с использованием следующего HTML.

```
<form action="main.html" method="get">
  <label>
    Введите имя:<br>
    <input placeholder="Имя пользователя"><br>
  </label>
  <input type="submit" value="Ввод">
</form>
```

Обратите только внимание, что рекомендуется сохранить имя текущего пользователя в локальном хранилище до перехода на страницу main.html, например, с использованием следующих команд чтения и записи:

```
function store(source) {
  localStorage["tetris.username"] = source.value;
}
function read(source) {
  source.value = localStorage["tetris.username"];
}
```

Сохранение имени пользователя в локальном хранилище позволит использовать это имя как на главной странице, так и при повторном переходе на страницу ввода имени.

Обработка действия на кнопке может выполняться с использованием следующей команды:

```
<button onclick="storeSth();redirectSomewhere()"> </button>
```

Обработка изменений в `<input>` может осуществляться с использованием слушателя изменений (событие «change»):

```
<input type="text" onchange="haveChanges()">
```

Переход на другую страницу может осуществляться с использованием следующей команды JavaScript:

```
window.location = "http://www.yoururl.com";
```

Обеспечение создания новой фигуры для тетриса по таймеру и ее движение. Для создания периодической задачи может использоваться команда.

```
const interval = setInterval(() =>
  console.log("periodic task")
, 700);
```

Для запуска отложенной команды по таймеру может использоваться команда

```
setTimeout(() =>
  console.log("timeout task")
, 1000);
```

Нарисовать прямоугольник в `<canvas>` можно, например, с использованием следующих команд:

```
function draw() {
  const canvas = document.getElementById('canvasid');
  if (canvas.getContext) {
    let ctx = canvas.getContext('2d');
    ctx.fillRect(25,25,100,100);
    ctx.clearRect(45,45,60,60);
  }
}
```

В первой строке функции ищется элемент `<canvas>` на странице. Во второй строке – проверяется его корректность, в третьей – получение доступа к 2D фигурам. Функция `fillRect` рисует прямоугольник, функция `clearRect` – очищает прямоугольник. Параметры данных функций (`x0`, `y0`, `x1`, `y1`) – координаты левого верхнего и правого нижнего углов прямоугольника.

Обеспечение управления пользователем падающей фигурой. Для обработки событий клавиатуры может использоваться событие «keydown». Пример использования:

```
document.addEventListener('keydown', (event) => {
  const keyName = event.key;
  console.log('Событие keydown: ' + keyName);
});
```

В результате в консоли браузера отображаются сообщения:

Событие keydown: ArrowUp

Событие keydown: ArrowDown

Событие keydown: ArrowLeft
Событие keydown: ArrowRight

Вопросы для контроля

1. Как можно создать ассиметричные ключи и для чего они используются?
2. В чем отличия в настройке nginx под Windows и Linux?
3. Как сохранять и получать данные из локального хранилища?
4. Как можно обрабатывать события на странице?
5. Как отобразить движение фигуры, состоящей из квадратов на <canvas>?
6. Как обеспечить обработку событий клавиатуры?

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

- Node.JS // URL: <https://nodejs.org/>;
- TypeScript // URL: <https://www.typescriptlang.org/>;
- OpenSSL. Cryptography and SSL/TSL Toolkit // URL: <https://www.openssl.org/>;
- Веб-сервер на основе Nginx и PHP-FPM // URL: <http://help.ubuntu.ru/wiki/nginx-phpfpm>;
- Nginx // URL: <https://nginx.ru/ru/>;
- Современный учебник Javascript // URL: <https://learn.javascript.ru/>;
- MDN web docs. Ресурсы для разработчиков, от разработчиков // URL: <https://developer.mozilla.org/ru/>;
- Тетрис // URL: <https://ru.wikipedia.org/wiki/Тетрис>.

Лабораторная работа 2. REST-ПРИЛОЖЕНИЕ УПРАВЛЕНИЯ БИБЛИОТЕКОЙ

Цель и задачи

Целью работы является изучение взаимодействия клиентского приложения с серверной частью, освоение шаблонов web-страниц, формирование навыков разработки динамических HTML-страниц, освоение принципов построения приложений с насыщенным интерфейсом пользователя.

Для достижения поставленной цели требуется решить следующие задачи:

- разработка интерфейса web-приложения;
- задание стилей для отображения web-приложения с учётом размера экрана (использование на компьютере, на мобильном телефоне);
- создание web-сервера на основе express;
- создание шаблонов web-страниц;
- настройка маршрутов;
- создание json-хранилища;
- обработка REST-запросов;

Основные теоретические сведения

CSS (Cascading Style Sheets – каскадные таблицы стилей) – язык описания внешнего вида документа, написанного с использованием языка разметки, используется как средство оформления внешнего вида HTML-страниц.

Express – это минималистичный и гибкий web-фреймворк для приложений Node.js, предоставляющий обширный набор функций для мобильных и web-приложений.

Pug и EJS – модули, позволяющие использовать шаблоны для HTML-страниц.

REST (Representational State Transfer – передача состояния представления) – стиль взаимодействия компонентов распределенного приложения. В рамках лабораторной работы – браузера и сервера web-приложения. Для взаимодействия используются стандартные методы:

- GET – получение записи (записей);
- POST – добавление записи;
- PUT – обновление или добавление записи;
- DELETE – удаление записи.

Общая формулировка задачи

Необходимо создать web-приложение управления домашней библиотекой, которое предоставляет список книг, их можно отфильтровать по признакам «в наличии», «возврат просрочен», есть возможность выдать книгу для чтения и вернуть книгу. Основные требования следующие:

1. Начальное состояние библиотеки хранится в JSON-файле на сервере. Текущее состояние – в переменной в памяти сервера.
2. В качестве сервера используется Node.JS с модулем express.

3. В качестве модуля управления шаблонами HTML-страниц используется pug либо ejs, все web-страницы должны быть сделаны с использованием pug либо ejs.

4. Предусмотрена страница для списка книг, в списке предусмотрена фильтрация по дате возврата и признаку «в наличии», предусмотрена возможность добавления и удаления книг. Удаление книг – с подтверждением.

5. Предусмотрена страница для карточки книги, в которой ее можно отредактировать (минимум: автор, название, дата выпуска) и выдать читателю или вернуть в библиотеку. В карточке книги должно быть очевидно: находится ли книга в библиотеке, кто ее взял (имя) и когда должен вернуть (дата).

6. Информация о читателе вводится с использованием всплывающего модального диалогового окна (<dialog>).

7. Оформление страниц выполнено с использованием CSS (допустимо использование w3.css).

8. Взаимодействие между браузером и web-сервером осуществляется с использованием REST.

9. Фильтрация списка книг осуществляется с использованием AJAX-запросов.

10. Логика приложения реализована на языке JavaScript.

11. Для всех страниц web-приложения разработан макет интерфейса с использованием Figma (<https://www.figma.com/>).

12. При оформлении элементов управления используются иконки (например, Font Awesome, <https://fontawesome.ru/all-icons/>).

Преимуществом будет создание и использование аутентификации на основе passport.js (<http://www.passportjs.org/>), в качестве примера можно использовать <https://medium.com/devschacht/node-hero-chapter-8-27b74c33a5ce>.

Преимуществом будет реализация загрузки и отображения обложек книг.

Описание ключевых методов при выполнении работы

Задание стилей для отображения web-приложения. Каскадные таблицы стилей (CSS) используют следующий синтаксис для задания стилей.

Селектор{свойство:значение; свойство:значение; }

Некоторые возможные варианты селекторов:

– p, div – изменяются стили всех p и div;

- `.myclass` – изменяются элементы, которым присвоен класс «`myclass`»;
- `#myid` – изменяется элемент с идентификатором «`myid`».

Некоторые возможные варианты свойств: `color`, `text-decoration`, `font-style`, `font-size`.

Примеры:

```
<style>
  #test1 {
    text-align: right;
    color: green;
  }
  p.test2{
    font-family:arial;
    color:brown;
  }
</style>
```

Использование стилей может учитывать размер экрана, на котором отображается HTML-страница.

```
@media screen and (max-width: 600px) {
  div {}
}
```

В приложение может быть подключен файл со стилями.

```
<link rel="stylesheet" href="w3.css">
```

Создание web-сервера на основе *express*. Перед установкой пакетов инициализируйте свое приложение одной из следующих команд (в корне проекта):

```
npm init
yarn init
```

в зависимости от того, какой менеджер пакетов используется (`npm` или `yarn`). Для уменьшения количества вопросов можно передать флаг «-у» и затем внести необходимые изменения в `project.json` по сравнению со значениями по умолчанию.

Установка `express` может выполнена одной из следующих команд:

```
npm install --save express
yarn add express
```

в зависимости от того, какой менеджер пакетов используется (`npm` или `yarn`). Далее будет указываться только имя пакета и то, что его необходимо установить; предполагается, что читатель воспользуется одной из указанных выше команд.

Простейший сервер может выглядеть следующим образом:

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

```
app.get('/', (req, res) => {
  res.send('Hello World')
}).listen(3000);
```

При этом он обрабатывает GET-запросы по адресу `http://localhost:3000`. Параметр `req` хранит информацию запроса, параметр `res` – ответа.

Простейший сервер, который поддерживает несколько страниц (`http://localhost:3000` и `http://localhost:3000/page`), может выглядеть следующим образом:

```
const server = require("express")(); // Создали сервер
server.get("/page", (req, res, next) => {
  res.end('Here is a page'); // Ответ
  next(); // Переход к следующему обработчику
});
server.get("/", (req, res, next)=>{
  res.end("ROOT PAGE");
  next();
});
server.get("*", (req, res)=>{
  res.status(404); // Ошибка - нет такой страницы
  res.end("Page not found");
});
server.listen(3000, ()=>{ // Запуск
  console.log("Server started at http://localhost:3000")
});
```

Возможно создание сервера с поддержкой статических папок на сервере для хранения ресурсов и использования шаблонов web-страниц:

```
let express = require("express");
let server = express();
// Указание статической папки public
server.use('/public', express.static('public'));
/* Подключение обработчика шаблонов pug, шаблоны - в папке views */
server.set("view engine", "pug");
server.set("views", './views');
/* Отображение страницы с шаблоном mypage.pug из папки views */
server.get("/", (req, res) => {
  res.render("mypage", {
    value: 1 /* Значение value=1 передается в шаблон */
  });
});
server.listen(3000);
```

Создание шаблонов web-страниц. Шаблоны web-страниц могут создаваться с использованием шаблонов `pug` или шаблонов `ejs`. Далее будет рассмотрен пример для `pug`, необходимо установить пакет «`pug`». Для запуска преобразования из `pug`-файла в `html`-файл в командной строке будет полезен

«`pug-cli`», в рамках лабораторной работы это может быть полезно только на этапе изучения работы с шаблонами.

Пример `pug`-шаблона:

```
html
  head
    meta(charset="utf-8")
    title Сессия
  body
    h1 Счетчик
    -if (value === 1)
      p Добро пожаловать в первый раз
    - else
      p Не первое посещение:
        =value
```

Команда преобразования с использованием модуля `pug-cli`, установленного глобально:

```
pug -O {"value":"1"} views/msg.pug
```

Важно, что данной командой мы пользуемся только на этапе изучения `pug`, в процессе работы приложения преобразование должно выполняться автоматически.

Результат преобразования в `html`-файл с передачей в качестве параметра `value=1`:

```
<html>
<head>
  <meta charset="utf-8"/>
  <title>Сессия</title></head>
<body><h1>Счетчик</h1>
<p> Добро пожаловать в первый раз</p></body>
</html>
```

Обратите внимание, что отступ в `pug`-шаблоне соответствует уровню вложенности в `html`-файле. Атрибут элемента `<meta>` задан в круглых скобках (несколько атрибутов будут перечисляться через запятую). Содержимое элемента пишется через пробел от имени этого элемента. Строка, начинающаяся со знака «`-`», вычисляется. Переменная, переданная в качестве свойства шаблону, используется либо в вычисляемых строках, либо если перед ней стоит знак «`=`».

Передадим другой параметр при преобразовании файла:

```
pug -O {"value":"2"} views/msg.pug
```

Результат преобразования:

```
<html>
<head>
  <meta charset="utf-8"/>
```

```
<title>Сессия</title></head>
<body><h1>Счетчик</h1>
<p>Не первое посещение:2</p></body>
</html>
```

Обратите внимание, как изменился результат выполнения условия.

Следует также отметить, что pug преобразует шаблон в минимизированный HTML, в котором отсутствует форматирование. Приведенные html-файлы отформатированы для удобства восприятия.

Настройка маршрутов. Маршруты обычно обрабатываются в отдельных модулях. Для этого используются следующий способ деления.

Файл приложения app.js.

```
const express = require("express");
const server = express();
const routes = require("./routes");
server.use("/", routes);
server.listen(3000);
```

Файл маршрутов routes.js должен находиться в той же папке, что и app.js и содержать следующий код:

```
const express = require("express");
const router = express.Router();
router.get("/page", (req, res, next) => {
  res.end(`Here is a page`); // Ответ
  next(); // Переход к следующему обработчику
});
router.get("/", (req, res, next)=>{
  res.end("ROOT PAGE");
  next();
});
router.get("*", (req, res)=>{
  res.status(404); // Ошибка - нет такой страницы
  res.end("Page not found");
});
module.exports = router;
```

В данном примере «*» означает все остальные маршруты.

Предполагается, что пользователь будет обращаться к страницам с использованием их идентификатора:

```
router.get("/books/:num", (req, res, next) => {
  const id = req.params.num;
  for (value of books)
    if(value === id)
      res.end(`${value} is best!`);
  next();
});
```

В данном примере, если сервер использует порт 3000 и нам нужна книга с id=123, то обращение происходит с использованием метода GET

по адресу `http://localhost:3000/books/123`. Конструкция «:num» указывает, что в данном фрагменте URL будет передаваться изменяемое значение и ему задается имя «num», это же имя используется для доступа – `req.params.num`.

При необходимости обработки других методов, например, POST, PUT или DELETE в программе изменится метод обращения:

```
router.post("/groups/:num", (req, res, next)...)
router.put("/groups/:num", (req, res, next)...)
router.delete("/groups/:num", (req, res, next)...
```

Создание json-хранилища. JSON (JavaScript Object Notation) – объект JavaScript.

Пример json-файла (имя `ru.json`).

```
{
  "run": "Побежал",
  "loaded": "загружен"
}
```

В данном примере приведен простой JSON-объект, который содержит два атрибута. Обратите внимание на то, что имена и значения заключены в кавычки. Значения такого объекта – всегда строки, при необходимости использования других типов их придется преобразовывать из строк.

Загрузка и использование содержимого json-файла «`ru.json`»:

```
const lang = require("./ru");
console.log("Animal", lang.loaded);
```

Таким образом, json файл загружается как обычный модуль JavaScript. Содержимое json файла автоматически преобразуется в полноценный json-объект, к которому можно обращаться как к обычному объекту.

Есть возможность любой json-объект (без внутренних циклов) преобразовать в строку и обратно с использованием встроенного объекта `JSON`:

```
const string = JSON.stringify(lang) /* Преобразовать объект в строку */
const objAgain = JSON.parse(string) /* Преобразовать строку в объект */
```

Есть возможность сохранять изменения в файл:

```
const fs = require("fs");
fs.writeFile('message.txt', 'Hello Node.js', (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

В данном примере используется стандартный модуль «fs» (не требует установки). Имя файла «`message.txt`», записываемый текст «`Hello Node.js`». Запись выполняется асинхронно. При возникновении ошибок будет создана исключительная ситуация.

Для выполнения лабораторной работы необходимо определиться, каким будет состав json-файла, который хранит информацию по книгам и по их доступности в библиотеке.

Обработка REST-запросов. Приложение обработки REST-запросов может выглядеть следующим образом.

```
import express from 'express'
import fileUpload from 'express-fileupload'
import cookieParser from 'cookie-parser';
const server = express();
server.use(cookieParser()); // Обработка cookies
server.use(fileUpload()); // Загрузка файлов на сервер
server.use(express.json()); // Обработка параметров в body
const groups = require('./groups.js');
server.use('/groups', groups);
server.listen(3000);
```

Модуль «cookie-parser» обеспечивает работу с Cookies. В данном примере основная часть приложения доступна по адресу `http://localhost:3000/groups`. Все остальные пути – относительные пути, вычисляемые относительно данного адреса.

Рассмотрим, как реализованы REST-запросы в модуле «groups». Часть обработчиков запросов в примере удалено для сокращения записи:

```
const router = require('express').Router();
const groups = [
  {id: 1, name: "5381", students: 15, rating: 4.1},
  {id: 2, name: "5303", students: 13, rating: 4.7},
];
router.get('/', (req, res)=>{
  res.json(groups);
});
module.exports = router;
```

Приведенный фрагмент кода обеспечивает возвращение на клиента объекта `groups` в формате JSON – команда «`res.json(groups)`»:

```
router.get('/:id([0-9]{1,})', (req, res)=>{
  const group = groups.filter((g)=>{
    if(g.id == req.params.id)
      return true;
  });
  // ...
});
// http://localhost:3000/groups/2
```

Пример запроса приведен в комментарии. Конструкция «`([0-9]{1,})`» представляет собой регулярное выражение, проверяющее, что `id` именно число. Конструкция «`req.params.id`» получает доступ к переданному значению «`:id`». В результате фильтрации в переменную `group` должен попасть один элемент, если он есть в массиве.

Добавление записи – метод POST:

```
router.post('/', (req, res)=>{
  let body = req.body;
  if(!body.name ||
    !body.students.toString().match(/^([0-9]{1,})$/g) ||
    !body.rating.toString().match(/^([0-9]\.[0-9])$/g)) {
    res.status(400);
    res.json({message: "Bad Request"});
  } else {
    // ...
  }
});
```

В данном случае обрабатывается POST-запрос, в который в качестве параметров (см. «req.body» – это свойство доступно благодаря подключенному модулю «body-parser») переданы значения «name», «students» и «rating». Параметры в GET-запросе передаются в URL, а в POST-запросе – в теле запроса. Функция match() обеспечивает проверку регулярного выражения (мы проверяем, что переданные параметры соответствуют тем, что мы ожидаем). На месте многоточия – обработка корректного запроса, в котором переданы все необходимые параметры.

Обновление или добавление записи – метод PUT:

```
router.put('/:id', (req, res)=>{
  const body = req.body;
  // ...
});
```

Принципы обработки PUT-запроса и получения параметров идентичны POST-запросу. Отличия в логике: если удастся найти объект, то он обновляется, иначе – добавляется.

Удаление записи – метод DELETE:

```
router.delete('/:id', (req, res)=>{
  const removeIndex = groups.map((group)=>{
    return parseInt(group.id);
  }).indexOf(parseInt(req.params.id));
  // ...
});
```

В функции map() удобно искать и сравнивать числа, поэтому используется функция parseInt(). В результате, если «removeIndex==-1», значит элемент не найден.

Отправка асинхронного GET-запроса с использованием Ajax:

```
let response = await fetch(
  'http://localhost:3000/', // Адрес запроса
  {
    method: 'GET' // Параметры запроса
```

```
});
let text = await response.text(); // Текст
console.log(text); // HTML-ответ
```

Функция `fetch()` позволяет отправлять асинхронные запросы на сервер. Получение тела запроса также осуществляется асинхронно с использованием функций `text()`, `json()`, `blob()` и т.п.

Отправка асинхронного POST-запроса с использованием Ajax:

```
let response = await fetch(
  'http://localhost:3000/', // Адрес запроса
  {
    method: 'POST', // Метод
    headers: { // Заголовки
      'Content-Type': 'application/json;charset=utf-8'
    },
    body: JSON.stringify({}) // Тело запроса
  }
);
console.log(response.headers.get('Content-Type')) // Тип ответа
let json = await response.json(); // Получение json
console.log(json); // HTML-ответ
```

Разница в том, что при использовании GET параметры передаются в URL, а при использовании POST – в теле запроса.

Отличия модулей CommonJS и ES6. Модули CommonJS осуществляют импорт с использованием функции `require()`, экспорт с использованием переменных `module.exports` и `exports`.

Пример импорта.

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

Пример экспорта.

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

Модули ES6 осуществляют импорт с использованием ключевых слов `import` и `from`, экспорт осуществляется с использованием ключевых слов `export` и `default`. Экспорт по умолчанию (`export default`) может быть только один в js-файле.

Пример импорта.

```
import express from 'express';
import {router} from './router';
```

Пример экспорта.

```
export {router};
```

При выполнении лабораторных работ рекомендуется придерживаться стиля ES6 модулей.

Подключение иконок Font Awesome. Для подключения иконок Font Awesome необходимо загрузить их с сайта разработчика: <https://fontawesome.ru/all-icons/>, примеры ссылок для скачивания:

- <https://fontawesome.ru/assets/font-awesome-4.7.0.zip>;
- <https://use.fontawesome.com/releases/v5.15.4/fontawesome-free-5.15.4-web.zip>.

Загруженную папку необходимо распаковать на сервер приложений, в папке css есть файл fontawesome.min.css.

Настройка иконок осуществляется за счёт установки соответствующих классов элементам HTML.

```
<html>
<head>
  <link rel="stylesheet" href="font-awesome/css/font-
awesome.min.css">
</head>
<body>
<ul class="fa-ul">
  <li><i class="fa-li fa fa-check-square"></i>Icons</li>
</ul>
</body>
</html>
```

Вопросы для контроля

1. Что такое CSS? Опишите примеры использования и подключения к HTML-странице.
2. Назовите команду установки express и напишите программу создания простейшего сервера.
3. Какие команды необходимо использовать для указания статической папки и настройки pug-шаблонов и ejs-шаблонов для HTML?
4. Приведите пример pug-шаблона (ejs-шаблона), укажите в какой HTML он будет переведен.
5. В чем отличие в обработке с использованием express GET и POST запросов? Какой вариант передачи параметров в URL вы знаете?
6. С помощью какой команды может быть сформировано строковое представление объекта JavaScript? Какая команда используется для обратного преобразования?
7. Объясните отличия между модулями CommonJS и ES6.

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

- Справочник CSS // URL: <https://webref.ru/css>;
- W3.CSS Tutorial // URL: <https://www.w3schools.com/w3css/>;
- Express // URL: <https://www.npmjs.com/package/express>;
- Express // URL: <http://expressjs.com/ru/>;
- Pug // URL: <https://www.npmjs.com/package/pug>;
- Pug. Getting Started // URL: <https://pugjs.org/>;
- EJS // URL: <https://ejs.co/>;
- Node.js – RESTful API // URL:

https://www.tutorialspoint.com/nodejs/nodejs_restful_api.htm;

- JSON – Introduction // URL:

https://www.w3schools.com/js/js_json_intro.asp;

- Сетевые запросы // URL: <https://learn.javascript.ru/network>;
- Passport // URL: <http://www.passportjs.org/>;
- Аутентификация в Node.js с использованием Passport.js // URL:

<https://medium.com/devschacht/node-hero-chapter-8-27b74c33a5ce>.

Лабораторная работа 3. МОДУЛЬ АДМИНИСТРИРОВАНИЯ ПРИЛОЖЕНИЯ «СОЦИАЛЬНАЯ СЕТЬ»

Цель и задачи

Целью работы является изучение возможностей применения компилятора Babel, библиотеки jQuery, препроцессора LESS, препроцессора SASS/SCSS, инструмента выполнения повторяющихся задач GULP, освоение инструмента сборки Webpack, регистрация разработанных модулей, формирования навыков построения структурированных web-приложений, освоение особенностей стандартных библиотек.

Для достижения поставленной цели требуется решить следующие задачи:

- разработка интерфейса web-приложения с использованием Figma (<https://www.figma.com/>);
- создание web-сервера на основе express, настройка маршрутов, подготовка и обработка REST-запросов (серверная часть);
- создание шаблонов web-страниц с использованием pug или ejs, указание путей подключения js-файлов;

- разработка стилей web-приложения с использованием LESS или SASS/SCSS;
- разработка клиентских js-файлов с использованием библиотеки jQuery и с использованием новейших возможностей в соответствии последним стандартом ECMAScript;
- конфигурирование GULP для решения задач преобразования pug-файлов в формат HTML, less-файлов и sass-файлов в css-файлы, обработка js-файлов с использованием Babel.

Основные теоретические сведения

LESS и SASS/SCSS – это динамические языки стилей, обеспечивающие следующие расширения CSS: переменные, вложенные блоки, миксины, операторы и функции. LESS и SASS/SCSS могут работать на стороне клиента или на стороне сервера под управлением Node.js.

jQuery – библиотека JavaScript, предназначенная для упрощения взаимодействия JavaScript и HTML. Библиотека jQuery помогает получать доступ к любому элементу DOM, обращаться к атрибутам и содержимому элементов DOM, манипулировать ими, предоставляет простой API для работы с AJAX.

jQuery UI (<https://jqueryui.com/>) – библиотека JavaScript с открытым исходным кодом для создания насыщенного пользовательского интерфейса в веб-приложениях. Она построена на основе библиотеки jQuery и предоставляет упрощенный доступ к ее функциям взаимодействия, анимации и эффектов, а также набор виджетов для построения интерфейса пользователя.

Babel – компилятор JavaScript, который позволяет разработчику использовать в своих проектах самые последние стандарты ECMAScript с поддержкой во всех браузерах.

Gulp – это менеджер задач для автоматического выполнения часто используемых задач, написанный на JavaScript. Программное обеспечение поддерживает командную строку для запуска задач, определенных в конфигурационном файле.

Webpack (<https://webpack.js.org/>) – модуль JavaScript, обеспечивающий сборку статических пакетов («bundle»). На вход он получает «точки входа» (js-файлы), в которых он находит все зависимости и формирует соответствующие пакеты (по одному пакету на одну «точку входа»). Пакет представляет собой специально оформленный js-файл, в него входят не только связанные js-файлы, но и ресурсы, например, css-файлы.

Общая формулировка задачи

Необходимо создать web-приложение, обеспечивающее администрирование социальной сети: можно управлять участниками, их ролями, сообщениями. Основные требования следующие:

1. Перечень участников, их друзей, сообщений и т.п. хранится в JSON-файлах на сервере.

2. В качестве сервера используется Node.JS с модулем express.

3. Разработка ведется с использованием стандарта не ниже ECMAScript2015, используются ES6 модули.

4. Стили описываются с использованием LESS или SASS, при этом используются ключевые методы LESS/SASS (переменные, вложенные блоки, миксины, операторы и т. п.).

5. Клиентская часть разрабатывается с использованием jQuery (работа с DOM, AJAX-запросы), используются компоненты jQuery UI или Bootstrap.

6. Предусмотрена HTML-страница для списка пользователей (ФИО, дата рождения, email, фотография, роль, статус). Предусмотрена возможность редактировать данные пользователя, изменять роль (администратор, пользователь), изменять статус (не подтверждённый пользователь, активный, заблокированный).

7. Предусмотрены:

- HTML-страница для списка друзей пользователя;
- HTML-страница для списка новостей друзей пользователей.

8. Взаимодействие браузера с сервером осуществляется по протоколу HTTPS, все изменения сохраняются в соответствующие json-файлы на сервере.

9. Сборка клиентской части (преобразования less или sass, pug или ejs, babel, минификация) осуществляется с использованием двух инструментов: GULP и Webpack. Это должны быть две отдельные сборки в разные папки.

10. Регистрация и удаление разработанных модулей в npm.

11. Для всех страниц web-приложения разработан макет интерфейса с использованием Figma (<https://www.figma.com/>).

Преимуществом будет, если будет предусмотрена работа с

- фотографиями пользователя: список фотографий и возможность блокирования и активации фотографий администратором;
- новостями: возможность блокирования и активации новостей.

Описание ключевых методов при выполнении работы

Разработка стилей web-приложения с использованием LESS. LESS и SASS – это надстройки над CSS. Далее будет описано использование LESS. Любой CSS код – это корректный LESS, но дополнительные элементы LESS не будут работать в простом CSS. LESS добавляет много динамических свойств в CSS. Он вводит переменные, примеси, операции, функции. Попробовать LESS можно на сайте <http://lesscss.org/less-preview/>, SASS – на сайте <https://sass.js.org/>.

Для использования установите модуль «less».

Переменные в LESS работают так же, как и в большинстве других языков программирования.

Исходный LESS-файл

```
@color: #4D486A;
#header {
  color: @color;
}
h2 {
  color: @color;
}
```

Результат преобразования в CSS

```
#header {
  color: #4D486A;
}
h2 {
  color: #4D486A;
}
```

Обратите внимание, что перед переменной ставится символ @, ее значение задается, как значение свойства в CSS.

Примеси («миксины») в LESS.

Исходный LESS-файл

```
.a (@x: 20px) {
  color: red;
  width: @x;
}
.mixin-class {
  .a();
}
#mixin-id {
  .a(100px);
}
```

Результат преобразования в CSS

```
.mixin-class {
  color: red;
  width: 20px;
}
#mixin-id {
  color: red;
  width: 100px;
}
```

Примеси могут применяться для любых селекторов CSS (классы, идентификаторы и т.д.). Примеси могут принимать параметры, которые указываются в скобках. У параметров может быть значение по умолчанию.

Вложенные правила в LESS.

Исходный LESS-файл

```
#header {
  background: lightblue;
  a {
    color: blue;
  }
}
```

Результат преобразования в CSS

```
#header {
  background: lightblue;
}
#header a {
  color: blue;
}
```



```

    &:hover {
        color: green;
    }
}
}
        color: blue;
    }
#header a:hover {
    color: green;
}
}

```

Вложенные правила описывают соответствующие виды селекторов. В том числе поддерживаются псевдоклассы. Для указания селектора при этом используется символ «&».

Функции в LESS.

Исходный LESS-файл

```

.average(@x, @y) {
    @Average: ((@x + @y) / 2);
}

```

Результат преобразования в CSS

```

div {
    padding: 11px;
}

```

```

div {
    .average(12px, 10px);
    padding: @Average;
}

```

Функция используется в стиле примеси, которая возвращает значение переменной. В примере вызывается примесь, которая возвращает значение переменной, результат вычисления используется как значение свойства.

Разработка клиентских js-файлов с использованием библиотеки jQuery. Подключение jQuery к странице может осуществляться с использованием следующей команды.

```
<script src="/public/jquery.js"></script>
```

В данном случае jquery.js скачан с официального сайта <https://jquery.com/>.

Для использования на сервере установите модуль «jquery».

Обращение к элементам DOM (Document Object Model – объектная модель документа) с использованием селекторов CSS.

```

$("h3").text("Проверка 1");
jQuery("p").text("Проверка 2");

```

В примере функции «\$» и «jQuery» в качестве параметров принимают селектор CSS и решают одну и ту же задачу. Чаще всего используется обращение «\$» – исключительно из соображения сокращения записи. Функция «text()» заменяет текст HTML-элемента.

В ситуации, когда JS-файл загружается в заголовке и выполняет операции с DOM документа, может получиться, что страница загружена не до конца. Для защиты от данной ситуации JavaScript код рекомендуется выполнять в функции, которая вызывается после окончательной готовности HTML-документа:

```
$(document).ready(()=>{
    $("h3").text("Проверка 1");
    jQuery("p").text("Проверка 2");
});
```

Переменная «document» – встроенная переменная HTML-страницы. Функция «ready()» проверяет готовность документа и возможность выполнения манипуляций с DOM.

jQuery предлагает различные операции по работе с DOM:

```
$("#biglist").find("li").first().next()
.text("ВТОРОЙ в biglist");
```

В данном примере находится список с идентификатором «#biglist», находятся все его «потомки» , выбирается первый из них, осуществляется переход ко второму и выполняется замена текста во втором :

```
$("#mychild").parent().css("color", "blue");
```

В данном примере находится HTML-элемент с идентификатором «#mychild», выбирается его «родитель», которому устанавливается CSS-стиль «color: blue».

В jQuery предусмотрены методы по изменению DOM:

```
$(document).ready(()=>{
    $("#li1").append($("<p>Append</p>"));
    $("#li2").prepend($("<p>Prepend</p>"));
    $("#li3").after($("<p>After</p>"));
    $("#li4").before($("<p>Before</p>"));
    $("#li5").remove();
});
```

В данном примере изменяются пять HTML-элементов: 1) добавляется элемент перед текстовым полем найденного элемента; 2) добавляется элемент после текстового поля найденного элемента; 3) перед элементом; 4) после элемента; 5) удаляется элемент. Конструкция «\$("<p>Before</p>")» создает новый HTML-элемент.

jQuery предлагает методы для обработки событий:

```
let counter = 0;
$("button").on("click", ()=>{
    $("#myP")
        .text(`Счетчик нажатий = ${++counter}`);
});
```

В данном примере все кнопки документа будут обрабатывать событие «click», в результате которого в элемент «#myP» будет вставляться соответствующий текст. Функция «on()» добавляет обработчик по имени события.

jQuery предлагает методы отправки AJAX-запросов:

```
$.get("exAjax", {parameters: "to", server: "side"})
    .done((data)=>{
```

```

    $("p").text(JSON.parse(data).fromserver)
  })

```

В данном примере функция «\$.get()» отправляет GET-запрос к странице «exAjax», передавая в качестве параметра JSON-объект «{parameters: "to", server: "side"}». Функция «done((data)=>{})» вызывается после получения результата с сервера, при этом в переменную «data» попадает ответ сервера. В данном примере с сервера пришел JSON, который был преобразован из текста с использованием функции «JSON.parse()».

jQuery предлагает функцию «\$.post()» для отправки POST-запросов и функцию «\$.getJSON()» для обработки GET-запросов на сервер, в результате которых ожидается ответ от сервера в формате JSON (при этом произойдет неявное преобразование результата в формат JSON).

Разработка клиентских js-файлов с использованием jQuery UI. Для разработки интерфейса пользователя с использованием jQuery UI необходимо правильно подключить библиотеки. Вариант подключения с использованием pug-шаблона:

```

head
  meta(charset="utf-8")
  title Использование jQuery UI
  link(href="public/jquery-ui.css", rel="stylesheet")
  script(src="public/jquery.js")
  script(src="public/jquery-ui.js")

```

В данном случае принципиально, что сначала загружается библиотека «jquery.js», а затем библиотека «jquery-ui.js».

Подробное описание возможных элементов интерфейса в jQuery UI и вариантов их применения можно найти на сайте <https://jqueryui.com/>.

Рассмотрим примеры.

```

1. $("#d1").draggable({
    cursor: "move",
    cursorAt: {
      top: 50, left: 50
    }
  });

```

В данном примере элемент с идентификатором «#d1» становится перетаскиваемым, при этом внешний вид курсора мыши заменяется на «move», а положение курсора мыши сдвинуто относительно левого верхнего угла на 50px вправо и на 50px вниз.

```

2. $("#d2").resizable({
    maxHeight: 300,
    maxWidth: 400,
    minHeight: 90,

```

```

        minWidth: 90
    });

```

В данном примере элемент с идентификатором «#d2» позволяет изменять свой размер. При этом определены максимальный и минимальный допустимые размеры.

```

3. $(".accordion").accordion({
    heightStyle: "fill"
});

```

В данном примере элемент с классом «.accordion» становится виджетом «аккордеон».

```

4. $("input").checkboxradio();

```

В данном примере все поля ввода <input> становятся радиокнопками.

Ознакомиться с полным набором инструментов и методов jQuery UI можно по адресу <https://jqueryui.com/>.

Конфигурирование GULP для решения задач преобразования. Для использования GULP установите модули «gulp» и «gulp-cli». Конфигурационный файл по умолчанию – «gulpfile.js», конфигурационный файл для работы с модулями ES6 – «gulpfile.babel.js», дополнительно потребуется установка и настройка «babel».

Для создания простейшей задачи GULP в конфигурационный файл необходимо написать следующий JavaScript:

```

function mytask(cb) {
    console.log("hello!!!");
    cb();
}
exports.default = mytask

```

Запуск осуществляется командой «npx gulp». Приведенный пример создает задачу с именем «default», после успешного выполнения задачи следует вызвать cb(), который передается в качестве параметра в выполняемую функцию. При выполнении данного примера в консоль будет выведено сообщение «hello!!!». Команда «npx» необходима при локальной установке gulp, при глобальной вызов может осуществляться без неё.

Предусмотрено последовательное и параллельное выполнение задач:

```

const {series, parallel} = require("gulp")
function build(cb) {
    console.log("build");
    cb();
}
function clean(cb) {
    console.log("clean");
    cb();
}

```

```

}
exports.default = build
exports.myseries = series(clean, build)
exports.myparallel = parallel(clean, build)

```

Функция «series()» обеспечивает последовательное выполнение задач, функция «parallel()» – параллельное. Вызов последовательного выполнения «gulp myseries», вызов параллельного выполнения «gulp myparallel», вызов задачи по умолчанию «gulp».

Функция «src()» обеспечивает работу с исходными кодами, принимая в качестве параметра шаблон путей. Функция «pipe()» обеспечивает передачу файлов от одной функции к другой, «dest()» определяет место назначения. В данном примере файлы будут перекопированы из папки «src» в папку «dest» без преобразований.

Расширенный пример с использованием LESS:

```

const { src, dest } = require('gulp');
const less = require('gulp-less');
const rename = require('gulp-rename');
const cleanCSS = require('gulp-clean-css');
paths = { // Настройка путей
  styles: {
    src: 'src/**/*.less',
    dest: 'build/styles/'
  }
}
exports.default = function() {
  return src(paths.styles.src)
    .pipe(less()) // Обработать LESS
    .pipe(cleanCSS()) // Минификация CSS
    .pipe(rename({ // Переименовать
      basename: 'main',
      suffix: '.min'
    }))
    .pipe(dest(paths.styles.dest));
}

```

В данном примере используется несколько дополнительных модулей: «gulp-less» для преобразования LESS-файлов в CSS-файлы (будет работать только при установленном модуле «less»); «gulp-rename» для переименования файла в соответствии с заданным шаблоном; «gulp-clean-css» для минификации (удаления лишнего с целью минимизации размера) CSS-файла. Функция «less()» выполняет преобразование LESS-файлов в CSS-файлы, функция «cleanCSS()» выполняет очистку CSS, функция «rename()» создает один файл с именем «main.min.css», затем осуществляется копирование результатов в папку «build/styles/».

Для обработки babel и работы с ES6 модулями потребуется установка и настройка babel, подключение модуля «gulp-babel», «gulp-concat» используется для объединения JS-файлов в один файл, «gulp-uglify» – для минификации JS-файлов. Для обработки pug потребуется установка модуля «gulp-pug».

Конфигурирование сборщика Webpack. Webpack на основании модулей с зависимостями генерирует статические ресурсы, представляющие эти модули. Для работы необходимо установить модуль «webpack».

Основные элементы Webpack:

- entry – точки входа;
- output – вывод результата;
- loaders – загрузчики;
- plugins – подключаемые компоненты.

Конфигурация описывается в файле «webpack.config.js» (имя по умолчанию). Вызов Webpack с конфигурационным файлом по умолчанию:

```
webpack
```

Вызов с конфигурационным файлом с именем «mywebpack.js»:

```
webpack --config mywebpack.js
```

Пример простейшего конфигурационного файла:

```
module.exports = {  
  entry : './main.js',  
  output: {  
    filename: 'bundle.js'  
  }  
};
```

В данном файле описана точка входа – файл «main.js», на основании которой с учетом всех зависимостей будет сформирован результат – файл «bundle.js».

Предусмотрена поддержка нескольких точек входа:

```
module.exports = {  
  entry : {  
    main: './src/main.js',  
    second: './src/main2.js'  
  },  
  output: {  
    filename: './dist/[name].js'  
  }  
};
```

При этом при описании результата используется подстановка «[name]», в данном случае она будет принимать значения «main» и «second», используя имена точек входа. Таким образом можно обработать несколько точек входа.

Для обработки CSS и LESS необходимо воспользоваться загрузчиками. При этом для LESS обязательно подключение модулей «less-loader» и «less»:

```
module.exports = {
  entry : './src/main4.js',
  output: {
    filename: './dist/bundle.js'
  },
  module: {
    loaders: [
      {
        test    : /\.css$/,
        exclude: /node_modules/,
        loader  : 'style-loader!css-loader'
      },
      {
        test    : /\.less$/,
        exclude: /node_modules/,
        loader  : 'style-loader!css-loader!less-loader'
      }
    ]
  }
};
```

В данном примере подключены два загрузчика: для CSS и для LESS. В обоих случаях в свойстве «exclude» указано исключение «/node_modules/» – не следует обрабатывать файлы в загруженных модулях. В свойстве «test» указаны регулярные выражения, определяющие какие файлы будут обрабатываться. В свойстве «loader» перечислены загрузчики, они разделены знаком «!» и выполняются в обратном порядке.

Рассмотрим пример использованного при этом «main4.js»:

```
import msg from './msg';
require('./mycss.css');
require('./test.less');
msg("Тестовое сообщение", document.getElementById("mymsg"));
```

В данном примере в первой строке импортируется функция из модуля «msg» и используется в последней строке js-файла. Следует обратить внимание на способ подключения CSS и LESS-файлов, подключение осуществляется командой «require». При этом в результате создания «bundle.js» все стили будут описаны с использованием JavaScript и отдельных CSS-файлов не будет.

Для обработки babel потребуются модули «babel-loader», «babel-preset-env», «babel-core»:

```
module.exports = {
  entry : './src/main5.js',
  output: {
```

```

    filename: './dist/bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /(node_modules|bower_components)/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['env']
          }
        }
      }
    ]
  }
};

```

В свойстве «test» указаны регулярные выражения, определяющие какие файлы будут обрабатываться. В свойстве «exclude» указано, какие файлы следует игнорировать. Свойство «use» указывает какой загрузчик использовать и какие настройки ему передавать.

Для обработки «pug» потребуется «pug-loader» и HtmlWebpackPlugin.

Подготовка списка имен файлов для обработки.

```
const PAGES = fs.readdirSync('./views/').filter(name =>
name.endsWith('.pug'))
```

Здесь «fs» – стандартный модуль работы с файлами, папка «views» содержит pug-файлы. В списке загрузчиков добавится «pug-loader».

```

{
  test: /\.pug$/,
  exclude: /node_modules/,
  loader: 'pug-loader'
}

```

Обработка файлов осуществляется с использованием plugins.

```

plugins: [
  ...PAGES.map(file => new HtmlWebpackPlugin({
    template: `./views/${file}`,
    filename: `./${file.replace(/\.pug/, '.html')}`
  }))
]

```

Для минификации кода может использоваться UglifyJsPlugin:

```
plugins: [ new webpack.optimize.UglifyJsPlugin() ]
```

Здесь переменная webpack загружена из одноименного модуля.

```
const webpack = require('webpack')
```

Применение компилятора babel. Для применения компилятора babel (<https://babeljs.io/>) необходимо установить модули:


```
npm i --save-dev @babel/core @babel/polyfill @babel/preset-env
@babel/register
```

Перед использованием необходимо создать конфигурационный файл «.babelrc». В простейшем случае он может выглядеть следующим образом:

```
{
  "presets": ["@babel/preset-env"]
}
```

Применение babel обеспечит преобразование JS-файл в соответствие стандарту ECMAScript 5 (кратко – ES5):

Исходный JS-файл	Преобразованный JS-файл
<pre>let data = 5 if (data > 2) console.log(`two \${data}`)</pre>	<pre>var data = 5; if (data > 2) console.log("two " + data);</pre>

Регистрация разработанных модулей в npm. Для проекта должен быть создан файл «package.json», который содержит описание разрабатываемого проекта и информацию об используемых зависимостях. Начальную конфигурацию «package.json» можно выполнить следующей командой (в корне проекта):

```
npm init
```

Созданный файл «package.json» может быть исправлен разработчиком.

Для публикации созданного модуля разработчик должен быть зарегистрирован в системе npm, для этого необходимо выполнить следующую команду:

```
npm adduser
```

Если пользователь уже создан, но не зарегистрирован на данном компьютере, то можно воспользоваться командой

```
npm login
```

Публикация модуля осуществляется следующей командой:

```
npm publish
```

Установка пакета (в том числе созданного разработчиком) осуществляется следующей командой (в корне проекта):

```
npm install имя-пакета
```

Пакет устанавливается в папку «node_modules»:

При необходимости автоматически создать запись в «package.json» в раздел зависимостей добавляется флаг «--save»:

```
npm install --save имя-пакета
```

При необходимости автоматически создать запись в «package.json» в раздел зависимостей на этапе разработки (а не поставки заказчику) добавляется флаг «--save-dev»:

```
npm install --save-dev имя-пакета
```

При необходимости обновления пакета можно воспользоваться следующей командой:

```
npm update имя-пакета
```

Удаление ненужного пакета осуществляется следующей командой:

```
npm remove имя-пакета
```

Аналогичные команды доступны для менеджера пакетов Yarn (<https://yarnpkg.com/>).

Вопросы для контроля

1. Что такое и для чего предназначены LESS, SASS/SCSS? Приведите примеры описания переменных, миксинов, операторов.
2. Приведите примеры использования jQuery для изменения DOM. В чем отличия данного подхода от традиционного JavaScript?
3. Приведите примеры использования jQuery для отправки AJAX-запросов. В чем отличия данного подхода от традиционного JavaScript?
4. Для чего предназначен babel? Какие модули используются для его корректной работы? Приведите примеры команд запуска и результатов его работы.
5. Для чего предназначен GULP? Приведите примеры с использованием less, sass, babel, pug, ejs, минификации.
6. Какие команды следует использовать для регистрации нового модуля npm? Как воспользоваться созданным модулем? Как обновить версию модуля?
7. Для чего нужен Webpack? Приведите пример простейшей конфигурации Webpack.

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

- Babel is a JavaScript compiler // URL: <https://babeljs.io/>;
- jQuery write less do more // URL: <https://jquery.com/>;
- jQuery user interface // URL: <https://jqueryui.com/>;
- Build fast, responsive sites with Bootstrap // URL: <https://getbootstrap.com/>;

- {less}. It's CSS, with just a little more // URL: <http://lesscss.org/>;
- Sass. CSS с суперсилой // URL: <https://sass-scss.ru/>;
- Модули // URL: <https://learn.javascript.ru/modules>;
- Gulp. A toolkit to automate & enhance your workflow // URL: <https://gulpjs.com/>;
- Webpack // URL: <https://webpack.js.org/>;
- Webpack Template Document // URL: <https://github.com/vedees/webpack-template>;
- NPM. Build amazing things // URL: <https://www.npmjs.com/>;
- Yarn. Safe, stable, reproducible projects // URL: <https://yarnpkg.com/>;
- ES6, ES8, ES2017: что такое ECMAScript и чем это отличается от JavaScript // URL: <https://tproger.ru/translations/wtf-is-ecmascript/>;
- Font Awesome. Иконочный шрифт и CSS-инструментарий // URL: <https://fontawesome.ru/>.

Лабораторная работа 4. МОДУЛЬ ПОЛЬЗОВАТЕЛЯ ПРИЛОЖЕНИЯ «СОЦИАЛЬНАЯ СЕТЬ»

Цель и задачи

Целью работы является изучение основ языка TypeScript и особенностей применения фреймворка Angular для разработки web-приложений, ведения журналов ошибок, реализации взаимодействия приложений с использованием web-сокетов, организации модульного тестирования web-приложений с использованием Jest.

Для достижения поставленной цели требуется решить следующие задачи:

- разработка интерфейса web-приложения;
- создание web-сервера на основе express, настройка маршрутов, подготовка и обработка REST-запросов с учетом CORS (серверная часть);
- создание каркаса web-приложения с использованием Angular;
- определение перечня компонентов и сервисов web-приложения;
- создание шаблонов компонентов;
- регистрация и подключение в web-приложение журнала ошибок.
- создание web-сокета для отправки сообщений всем клиентам;
- создание и запуск Jest тестов для приложения.

Основные теоретические сведения

Angular – фреймворк для разработки клиентских частей web-приложений, основанный на языке TypeScript. Фреймворк построен на использовании компонентного подхода, где каждый компонент может отображаться пользователю в соответствии с его индивидуальным шаблоном.

Фреймворк в том числе предлагает инструменты для управления из командной строки и организации тестирования.

CORS (Cross-Origin Resource Sharing) – это система, позволяющая отвечать на запросы из другого домена, отличного от домена происхождения запрашиваемого ресурса. Пример запроса CORS: приложение запущено на <http://localhost:8080>, а запросы отправляются на другой порт – <http://localhost:3000>.

Журналы ошибок позволяют контролировать появление ошибок как на этапе разработки, так и при работе пользователей. В качестве журналов ошибок предлагается использовать Rollbar (<https://rollbar.com/>) или Sentry (<https://sentry.io/>).

WebSocket – протокол связи, который может передавать и принимать одновременно сообщения поверх TCP-соединения, предназначен для обмена сообщениями между браузером и web-сервером, но может быть использован для любого клиентского или серверного приложения. Для создания web-сокетов предлагается использовать модуль Socket.IO (<https://socket.io/>).

Jest (<https://jestjs.io/ru/>) – это фреймворк для написания тестов серверной части web-приложений.

Общая формулировка задачи

Необходимо создать web-приложение, обеспечивающее использование пользователем социальной сети. Пользователь может зарегистрироваться в социальной сети. Может добавить или удалить свою фотографию, может управлять своими друзьями в социальной сети, может добавить сообщение (новость) на свою страницу, может просматривать список новостей своих друзей.

Основные требования:

1. Приложение получает исходные данные из модуля администрирования приложения «Социальная сеть» в виде JSON-файла и работает одновременно с модулем администрирования приложения «Социальная сеть».
2. В качестве сервера используется Node.JS с модулем express.

3. Предусмотрены:
- HTML-страница для регистрации пользователя;
 - HTML-страница для просмотра ленты новостей (пользователя и его друзей);
 - HTML-страница для добавления сообщения (новости).
4. Если пользователь является администратором, то у него есть возможность перехода в модуль администрирования приложения «Социальная сеть».
5. Переписка и страница новостей обновляются сразу после появления сообщений и новостей от пользователей без необходимости обновлять страницу целиком.
6. Разработаны тесты для серверной части web-приложения с использованием Jest.
7. Все элементы управления реализованы с использованием компонентов Angular. Взаимодействие между компонентами реализовано с использованием сервисов Angular.
8. Для реализации эффектов на HTML-страницах используются директивы Angular.
9. Для всех страниц web-приложения разработан макет интерфейса с использованием Figma (<https://www.figma.com/>).
- Преимуществом будет использование звукового сопровождения событий: получения сообщений, появление новостей.
- Преимуществом будет использование компонентов Angular Material (<https://material.angular.io/>).

Описание ключевых методов при выполнении работы

Создание каркаса web-приложения с использованием Angular. Для создания web-приложений с использованием Angular необходимо выполнить следующие команды:

```
npm install -g @angular/cli  
ng new my-angular
```

Первая – устанавливает фреймворк Angular, вторая – создает каркас нового проекта с именем «my-angular».

Запуск приложения может быть выполнен с использованием следующей команды (в корневой папке проекта):

```
ng serve
```

Для автоматического открытия браузера при запуске приложения можно воспользоваться следующей командой:

```
ng serve --open
```

При создании каркаса нового проекта Angular формирует «package.json»:

```
"scripts": {  
  "ng": "ng",  
  "start": "ng serve",  
  "build": "ng build --prod",  
  "test": "ng test",  
  "lint": "ng lint",  
  "e2e": "ng e2e"  
}
```

Соответственно, запуск приложения может быть выполнен также следующей командой.

```
npm run start
```

Приложение запускается по адресу <http://localhost:4200/>.

Обобщенная структура web-приложения на Angular приведена на рис. 5.1. Корневой объект подключает компоненты, каждый компонент состоит из класса, шаблона для отображения и может содержать метаданные. Компоненты взаимодействуют между собой при помощи сервисов, которые доступны для всех компонентов.

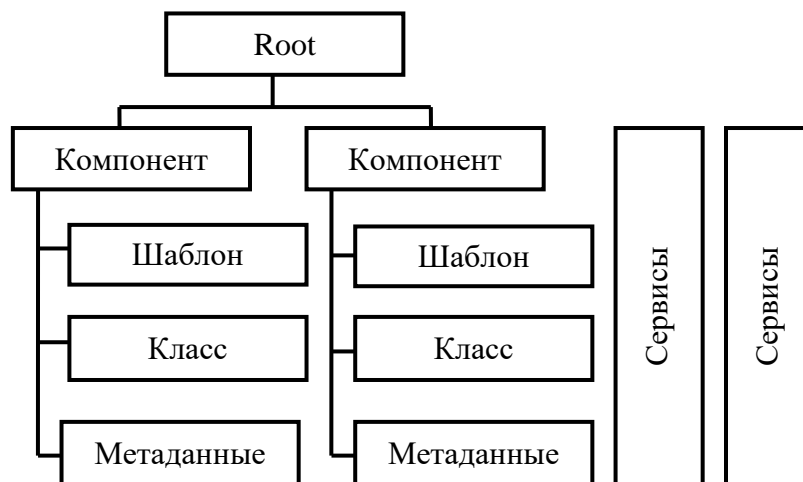


Рис. 5.1. Обобщенная архитектура приложения на Angular

Ключевые файлы для начала разработки web-приложения.

Сгенерирован файл «src/index.html», который представляет собой шаблон приложения в целом:

```
<!doctype html>  
<html lang="en">  
<head>  
  <meta charset="utf-8">  
  <title>MyAngular</title>  
  <base href="/">
```

```

    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
    <app-root></app-root>
</body>
</html>

```

Элемент `<app-root>` – главный элемент приложения, он описан в папке «app».

Сгенерирован файл «src/app/app.modules.ts», он представляет собой корневой объект и обеспечивает настройку приложения:

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [ AppComponent ],
  imports: [ BrowserModule ],
  providers: [],
  bootstrap: [ AppComponent ]
})
export class AppModule { }

```

`BrowserModule` предназначен для работы с браузером. `AppComponent` – главный компонент приложения (его рассмотрим ниже). `NgModule` – модуль, обеспечивающий создание и конфигурирование модуля.

Декоратор «`@NgModule`» с использованием свойства «declarations» объявляет компоненты, свойства «imports» – объявляет используемые модули, свойства «providers» – объявляет источники данных, свойства «bootstrap» – первый загружаемый модуль.

Главный компонент приложения «src/app/app.component.ts»:

```

import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}

```

В первой строке осуществляется импорт модуля «`Component`». Декоратор «`@Component`» в свойстве «selector» указывает HTML-тег для подключения компонента (сравните с элементом в файле «src/index.html»). Свойство «`templateUrl`» указывает путь до HTML-шаблона компонента, свойство «`styleUrls`» – CSS-стилей компонента. Стили «`app.component.css`» рассматри-

вать не будем, отметим только, что они применяются локально к каждому компоненту. Стили не являются обязательными.

В экспорте указано свойство «title», которое будет использоваться в шаблоне «src/app/app.component.html»:

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  
</div>
<h2>Here are some links to help you start: </h2>...
```

В данном примере на многоточия заменены данные по рисунку и часть страницы после заголовка <h2>. Для использования переменной title из «src/app/app.component.ts» она указывается в двойных фигурных скобках.

Обработка событий. Изменим компонент «AppComponent» для демонстрации некоторых возможностей Angular:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<p>Кнопка нажата {{count}} (раз)</p>
  <button (click)="onclick()">Нажми меня</button>`
})
export class AppComponent {
  count: number = 0;
  onclick(): void {
    this.count++;
  }
}
```

В данном примере шаблон приведен в описании компонента. В шаблоне отображается значение переменной «count», в кнопку добавлено событие «(click)», которое обрабатывается функцией «onclick()». Функция «onclick()» увеличивает значение «count» на единицу. Изменения немедленно отображаются в браузере.

Для работы с элементами <form> потребуется подключить дополнительный модуль в «src/app/app.modules.ts»:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [ AppComponent ],
  imports: [ BrowserModule, FormsModule ],
  providers: [],
  bootstrap: [ AppComponent ]
})
```



```
}}
```

```
export class AppModule { }
```

Модуль «FormsModule» необходимо подключить и включить в список импортируемых модулей, которые доступны во всех компонентах. Тогда в главном компоненте приложения «src/app/app.component.ts» можно будет использовать привязку к модели «[(ngModel)]»:

```
import { Component } from '@angular/core';
```

```
@Component({  
  selector: 'app-root',  
  template: `

Текст: {{mytext}}

  
  <input [(ngModel)]="mytext"><br>  
  <input [(ngModel)]="mytext">`  
})
```

```
export class AppComponent {  
  mytext: string = "ТЕКСТ"  
}
```

Привязка «[(ngModel)]» является двусторонней, соответственно, изменения в любом поле ввода приводят к изменению значения переменной «mytext».

Использование иерархии компонентов. Для добавления компонента ChildComponent внесем соответствующие изменения в «src/app/app.modules.ts»:

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { AppComponent } from './app.component';  
import { ChildComponent } from './child.component';  
@NgModule({  
  declarations: [ AppComponent, ChildComponent ],  
  imports: [ BrowserModule, FormsModule ],  
  providers: [],  
  bootstrap: [ AppComponent ]  
})
```

```
export class AppModule { }
```

В первом примере модуль «FormsModule» не является обязательным, но он нам пригодится в других примерах, поэтому его не удаляем. В свойстве «declarations» добавлен новый компонент «ChildComponent».

Внесем изменения в «AppComponent»:

```
import { Component } from '@angular/core';  
@Component({  
  selector: 'app-root',  
  template: `

### Мой заголовок</h3> <child-comp>Текст из app-root</child-comp>` })


```

```
export class AppComponent {
}
```

В шаблоне отображается заголовок, за которым следует обращение к дочернему компоненту, в теле которого передается текст.

Дочерний «src/app/child.component.ts»:

```
import { Component } from '@angular/core';
@Component({
  selector: 'child-comp',
  template: `<p>Дочерний компонент: {{mytext}}</p>
<ng-content></ng-content>`,
  styles: ['p {color: red; }']
})
export class ChildComponent {
  mytext: string = "текстовая переменная"
}
```

В шаблоне отображается текст компонента, затем при использовании элемента `<ng-content>` отображается текст, полученный из родительского компонента, в котором выполнена стилизация «styles» и все параграфы сделаны красными с применением соответствующего CSS. Следует отметить, что указанные таким образом стили распространяются только на текущий компонент.

Возможен более гибкий способ передачи данных в компоненты (использование «Input»):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<input [(ngModel)]="mytext">
<child-comp [childtext]="mytext"></child-comp>`
})
export class AppComponent {
  mytext: string = "Мой текст"
}
```

В данном примере «AppComponent» создает поле ввода для переменной «mytext» с двусторонней привязкой и затем привязывает «mytext» к переменной дочернего компонента «childtext»:

```
import { Component } from '@angular/core';
import { Input } from '@angular/core';
@Component({
  selector: 'child-comp',
  template: `<p>Дочерний компонент: {{childtext}}</p>`
})
export class ChildComponent {
  @Input() childtext: string = "текстовая переменная"
}
```

Дочерний компонент «ChildComponent» для использования декоратора «@Input» и получения значения переменных из родительского компонента должен импортировать модуль «Input». При этом при изменении текста в родительском компоненте будет автоматически изменяться значение в переменной «childtext» дочернего компонента.

Возможна организация передачи информации из дочернего компонента в родительский (использование «Output»):

```
import { Component, EventEmitter, Output } from '@angular/core';
@Component({
  selector: 'child-comp',
  template: `<button (click)="change(true)">Истина</button>
             <button (click)="change(false)">Ложь</button><br>`
})
export class ChildComponent {
  @Output() onMyChanged = new EventEmitter<boolean>();
  change(myvalue:boolean) {
    this.onMyChanged.emit(myvalue);
  }
}
```

В данном случае дочерний компонент «ChildComponent» с использованием декоратора «@Output» предоставляет «EventEmitter» по имени «onMyChanged». Нажатие на кнопку вызывает метод «change», который генерирует новое событие со значением «myvalue». В данном случае совпадение дженерика «boolean» (у «EventEmitter») и типа переменной «boolean» – совпадение, а не требование.

Данное событие обрабатывается в родительском компоненте «AppComponent»:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<child-comp (onMyChanged)="onMyAppChanged($event)">
</child-comp>
  Выбрано: {{mytext}}`
})
export class AppComponent {
  mytext: string = "Не определился"
  onMyAppChanged(myvalue:boolean) {
    this.mytext = myvalue ? "Истина" : "Ложь";
  }
}
```

В компоненте «AppComponent» выполнена привязка события дочернего компонента «onMyChanged» и функции «onMyAppChanged», в качестве па-

раметра передается событие с телом сообщения. В приведенном примере это логическая переменная.

Для обработки глобальных событий в компоненте можно воспользоваться функциями жизненного цикла:

- «ngOnInit()» (импортируем «OnInit» из «@angular/core») для глобальной инициализации компонента;
- «ngOnChanges()» (импортируем «OnChanges» из «@angular/core») для обработки любых изменений привязанных свойств.

Структурные директивы. Директива условия «ngIf» позволяет проверять условие на истинность и скрывать или отображать фрагменты HTML:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
```

Если переменная «myCondition» истинна, то отображается текст «Истина», иначе отображается текст «Ложь».

Того же эффекта можно достичь, используя «else»:

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
```

Имя условия «else» указывается в качестве атрибута тега <ng-template>.

Аналогично доступны директивы «ngFor» и «ngSwitch»:

```
import {Component} from '@angular/core';
@Component({
```

```

    selector: 'app-root',
    template: `
      <ul>
        <li *ngFor="let item of myList">{{item}}</li>
      </ul>`
  })
  export class AppComponent {
    myList = ["один", "два", "три"]
  }

```

Директива обеспечивает проход по всем элементам списка «myList», элементы доступны в виде «item».

Использование сервисов. Сервисы представляют собой источник данных, которым могут пользоваться несколько компонентов. Предположим, что в качестве передаваемых данных будет выступать массив объектов следующего вида («src/app/ToDo.ts»):

```

export class ToDo {
  constructor(public title: string) {}
}

```

У экземпляра класса доступно только одно строковое свойство «title». При этом сервис предоставления данных может выглядеть следующим образом («src/app/todo.service.ts»):

```

import { ToDo } from "../ToDo"
export class TodoService {
  private data: ToDo[] = [
    {title: "Выучить Angular"},
    {title: "Забыть Angular"}
  ]
  getData(): ToDo[] {
    return this.data;
  }
  addData(title: string) {
    this.data.push(new ToDo(title))
    console.log(`Добавлен: "${title}"`)
  }
}

```

Массив «data» предназначен для хранения данных. Функция «getData()» возвращает массив, функция «addData()» принимает в качестве параметра строку «title», создает новый экземпляр «ToDo» и добавляет его в массив.

Внесем соответствующие изменения в «src/app/app.component.ts»:

```

import {Component, OnInit} from '@angular/core';
import {ToDo} from "../ToDo";
import {TodoService} from "../todo.service";
@Component({
  selector: 'app-root',
  template: `
    <input [(ngModel)]="title"><br>

```

```

        <button (click)="add(title)">Добавить</button>
        <ul>
            <li *ngFor="let todo of mydata">{{todo.title}}</li>
        </ul>`,
    providers: [ TodoService ]
})
export class AppComponent implements OnInit {
    mydata: Todo[]
    constructor(private todoService: TodoService) { }
    ngOnInit(): void { // При инициализации компонента
        this.mydata = this.todoService.getData()
    }
    add(title: string) { // Добавление данных
        this.todoService.addData(title)
    }
}

```

Компонент «AppComponent» импортирует «Component» для создания компонента, «OnInit» для инициализации в момент создания, «ToDo» для отображения и «TodoService» для взаимодействия с сервисом.

В шаблоне отображается поле ввода и кнопка, которая передает введенный текст в функцию «add()». После кнопки с использованием директивы «ngFor» отображается список «mydata», в каждом элементе списка хранится «ToDo», поэтому для отображения используется свойство «todo.title».

Компонент «AppComponent» реализует интерфейс «OnInit», поэтому должна быть реализована функция «ngOnInit()», в которой осуществляется сохранение данных из сервиса в переменную «mydata». Переменная сервиса «todoService» инициализируется в конструкторе.

Функция «add()» вызывает функцию «addData()» сервиса.

В результате пользователю отображается список данных из «TodoService» и предоставляется возможность пополнять этот список произвольными пунктами.

Если описанный компонент дважды вывести на экран, то окажется, что он работает с отдельными экземплярами сервисов. Для корректной работы необходимо правильно подключить сервисы.

Перенесем логику в компонент «ChildComponent»:

```

import {Component, OnInit} from '@angular/core';
import {ToDo} from './ToDo';
import {TodoService} from './todo.service'
@Component({
    selector: 'child-comp',
    template: `
        <input [(ngModel)]="title"><br>
        <button (click)="add(title)">Добавить</button>
    `
})

```

```

    <ul>
      <li *ngFor="let todo of mydata">{{todo.title}}</li>
    </ul>`
  })
  export class ChildComponent implements OnInit{
    mydata: Todo[]
    constructor(private todoService: TodoService) { }
    ngOnInit(): void { // При инициализации компонента
      this.mydata = this.todoService.getData()
    }
    add(title: string) { // Добавление данных
      this.todoService.addData(title)
    }
  }
}

```

Обратите внимание, что по сравнению с предыдущим примером удален провайдер.

При этом вызывающий компонент «AppComponent» может выглядеть следующим образом:

```

import {Component} from '@angular/core';
import {TodoService} from "../todo.service"
@Component({
  selector: 'app-root',
  template: `<child-comp></child-comp>
<child-comp></child-comp>`,
  providers: [ TodoService ]
})
export class AppComponent {
}

```

В нем добавлен провайдер «TodoService», общий для двух дочерних компонентов.

В результате на экране будут отображены два списка, если в любом из них добавить новый элемент, то он добавится в оба списка, так как работа осуществляется с общим провайдером.

Обеспечение взаимодействия с сервером приложения. Для начала опробуем простой GET-запрос файла с сервера. Для этого в папке «src/assets» создайте файл «serverdata.json» (обратите внимание на квадратные и фигурные скобки в примере):

```

[
  {"title": "Выучить Angular"},
  {"title": "Забыть Angular"}
]

```

Изменим количество данных в «src/app/todo.service.ts»:

```

import { Todo } from "../Todo"
export class TodoService {
  private data: Todo[] = [

```

```

    {title: "Нет данных"}
  ]
  getData(): Todo[] {
    return this.data;
  }
  addData(title: string) {
    this.data.push(new Todo(title))
    console.log(`Добавлен: "${title}"`)
  }
}

```

Обеспечим выполнение AJAX-запроса в «src/app/app.component.ts»:

```

import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Todo } from "../Todo";
import { TodoService } from "../todo.service"
@Component({
  selector: 'app-root',
  template: `<child-comp></child-comp>
<child-comp></child-comp>`,
  providers: [ TodoService ]
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient, private todoService:
  TodoService){}
  ngOnInit(){
    this.http.get('assets/serverdata.json')
      .subscribe((data:Todo[]) => {
        for(let item of data)
          this.todoService.addData(item.title);
      }, (err) => {
        console.log("Error:", err)
      })
  }
}

```

Компоненту «AppComponent» необходимо реализовать интерфейс «OnInit» для загрузки данных в момент инициализации. В конструкторе ему передается модуль «HttpClient» и экземпляр сервиса «TodoService» для инициализации.

В функции «ngOnInit()» выполняется запрос «http.get()», которому в качестве параметра передается путь для отправки запроса, затем у результата вызывается функция «subscribe()», которая вызовется после успешного получения результата «data». Для каждого элемента в массиве «data» выполняется добавление строки в «todoService».

При возникновении ошибки в консоль браузера будет выдано соответствующее сообщение.

Если файл «`http://localhost:4200/assets/serverdata.json`» оказался недоступен, то проверьте «`angular-cli.json`». В нем в «`apps`» должно быть указано следующее свойство «`assets`» (ресурсы, которые доступны от клиента).

```
"assets": [  
  "assets",  
  "favicon.ico"  
]
```

При необходимости можно указать путь (относительно папки «`src`») до файла «`serverdata.json`»:

```
"assets": [  
  "assets",  
  "favicon.ico",  
  "assets/serverdata.json"  
]
```

Для выполнения POST-запроса необходимо воспользоваться методом «`http.post()`». Ниже приведен пример отправки POST-запроса с передачей параметров:

```
this.http.post('http://localhost:8080/api/values', {title: to-  
do.title})
```

При необходимости передать заголовки потребуется экземпляр «`HttpHeaders`»:

```
import {HttpClient, HttpHeaders} from '@angular/common/http';  
...  
const myHeaders = new HttpHeaders().set('Authorization', 'my-  
auth-token');  
this.http.post('http://localhost:8080/api/values', user, {head-  
ers:myHeaders});
```

Класс «`HttpHeaders`» загружается из того же модуля, что и «`HttpClient`», в его экземпляр устанавливаются необходимые заголовки и объекты (в данном примере – «`user`»), которые передаются на сервер.

Аналогичным образом реализуются PUT и DELETE-запросы. В рамках лабораторной работы предлагается реализовать отдельный сервер на базе модуля «`express`» для обработки REST-запросов клиента на отдельном порту.

CORS. Обработка CORS на сервере может осуществляться с использованием прм модуля `cors`. Пример настройки:

```
const cors = require('cors')  
const app = express()  
const corsOptions = {  
  'credentials': true,  
  'origin': true,  
  'methods': 'GET,HEAD,PUT,PATCH,POST,DELETE',  
  'allowedHeaders': 'Authorization,X-Requested-With,X-HTTP-  
Method-Override,Content-Type,Cache-Control,Accept',
```

```
}  
app.use(cors(corsOptions))
```

Для изучения вопроса рекомендуются:

- <https://www.npmjs.com/package/cors>,
- <https://developer.mozilla.org/ru/docs/Web/HTTP/CORS>.

Допустимо использование прокси:

```
ng serve --proxy-config src/proxy.conf.json
```

Пример конфигурационного файла proxy.conf.json:

```
{ "/api/*": { "target": "http://localhost:3000", "secure":  
false, "logLevel": "debug", "changeOrigin": true} }
```

Маршрутизация в Angular. Для демонстрации маршрутизации создадим несколько компонентов.

Компонент «src/app/home.component.ts».

```
import {Component} from "@angular/core"  
@Component({  
  selector: "app-root",  
  template: `<h2>Домашняя страница</h2>`  
})  
export class HomeComponent { }
```

Компонент «src/app/about.component.ts».

```
import {Component} from "@angular/core"  
@Component({  
  selector: "app-root",  
  template: "<h2>Компонент about</h2>"  
})  
export class AboutComponent { }
```

Компонент «src/app/notfound.ts».

```
import {Component} from "@angular/core"  
@Component({  
  selector: "app-root",  
  template: "<h2>Не найден</h2>"  
})  
export class NotFoundComponent { }
```

В качестве главного компонента будет выступать

«src/app/base.component.ts»:

```
import {Component} from "@angular/core"  
@Component({  
  selector: "app-root",  
  template: `<h1>Корневой компонент</h1>  
  <router-outlet></router-outlet>`  
})  
export class BaseComponent { }
```

В нем элемент <router-outlet> указывает место, куда будут выводиться компоненты, найденные в результате маршрутизации.

При этом файл «src/app/app.module.ts» будет выглядеть следующим образом:

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {Routes, RouterModule} from '@angular/router';
import {AboutComponent} from './about.component';
import {BaseComponent} from './base.component';
import {NotFoundComponent} from './notfound.component';
import {HomeComponent} from './home.component';
// определение маршрутов
const appRoutes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'about', component: AboutComponent},
  {path: '**', component: NotFoundComponent}
];
@NgModule({
  declarations:
    [AboutComponent, BaseComponent, NotFoundComponent,
    HomeComponent],
  imports: [BrowserModule, RouterModule.forRoot(appRoutes)],
  bootstrap: [BaseComponent]
})
export class AppModule { }
```

Добавился импорт модулей, отвечающих за маршрутизацию, и импорт разработанных компонентов.

При определении маршрутов указываются «path» – путь, заданный пользователем, и «component» – компонент, который будет отображаться пользователю. В свойство «imports» необходимо подключить «RouterModule.forRoot(appRoutes)».

В итоге при указании корневой страницы `http://localhost:4200/` будет отображаться «BaseComponent», в который в указанное место помещен «HomeComponent». При указании `http://localhost:4200/about` – будет отображаться «BaseComponent», в который в указанное место помещен «AboutComponent». Во всех остальных случаях будет отображаться «NotFoundComponent».

Обработка ошибок на сервере. JavaScript поддерживает традиционные try/catch конструкции.

Обещания (promise) – объекты, которые выполняются асинхронно.

Рассмотрим примеры.

```
1. new Promise((resolve, reject)=>{
    resolve(`ok`);
  });
```

В данном примере создается обещание, которое вызывает функцию с двумя параметрами «resolve» и «reject», это имена функций для передачи информации о результате работы обещания. Вызов «resolve» (как в данном примере) приведет к корректному завершению выполнения обещания.

2. В качестве параметра в «resolve» передается результат выполнения функции:

```
new Promise((resolve, reject)=>{
  reject(new Error(`my error`));
});
```

Вызов «reject» приводит к некорректному завершению функции. В качестве параметра в «reject» передается объект ошибки.

3. Обещания могут выполняться последовательно:

```
Promise.resolve().then(() => {
  return new Promise((resolve, reject)=>{
    resolve(`ok`);
  });
}).then(item => {
  return new Promise((resolve, reject)=>{
    reject(new Error(`my error`));
  });
}).catch(e => {
  console.log('error');
})
```

Вызов «Promise.resolve()» позволяет начать исполнение обещаний. Функция «then» исполняет обещание. Последовательно вызываемые функции «then» формируют цепочку вызова. Результаты выполнения обещаний передаются от одного «then» следующему. В данном примере второй «then» получает в переменную item значение «ok» из первого обещания. Во втором обещании возникнет ошибка, которая будет обработана в функции «catch».

А что после catch? Обработчик «catch(onRejected)» получает ошибку и должен обработать ее. Есть два варианта развития событий. Если ошибка не критичная, то «onRejected» возвращает значение через «return», и управление переходит в ближайший «then». Если продолжить выполнение с такой ошибкой нельзя, то он делает «throw», и тогда ошибка переходит в следующий ближайший «catch».

Журнал ошибок. Для отладки приложений могут использоваться, например, модуль «debug» или модуль «winston»:

```
const debug = require('debug')('httpname')
, http = require('http')
debug('booting %o', 'My App');
http.createServer(function(req, res){
```

```

    debug(req.method + ' ' + req.url);
    res.end('hello\n');
  }).listen(3000, function() {
    debug('listening');
  });

```

В приведенном примере для отладки используется модуль «debug». В момент загрузки модулю дается имя (в примере – «httpname»). Вывод сообщений осуществляется с использованием функции «debug».

Есть только одна особенность: для вывода сообщений должна быть настроена переменная среды «DEBUG». В нашем примере ей должны быть присвоены значения «DEBUG=*», чтобы выводились все сообщения, либо «DEBUG=httpname», чтобы выводились только сообщения модуля с именем «httpname».

Модуль «winston» предусматривает более гибкую настройку и предлагает выдачу сообщений в соответствии с уровнем ошибки (debug, info, warn, error) и в различные файлы, консоли и т. п.

Для сохранения информации в общий журнал ошибок должен использоваться один из сетевых журналов. Мы рассмотрим два варианта: Rollbar (<https://rollbar.com/>) и Sentry (<https://sentry.io/>).

Для установки Rollbar установите модуль «rollbar»:

```

const Rollbar = require("rollbar");
const rollbar = new Rollbar("access-token");
rollbar.log("TestError: Hello World!");

```

В данном примере с использованием функции «log» будет записано тестовое сообщение об ошибке. Важно, что текст «access-token» необходимо заменить на ключ, который выдает сайт Rollbar конкретному разработчику.

Для установки Sentry установите модуль «raven»:

```

const Raven = require('raven');
Raven.config('https://sentry-url').install();
try {
  doSomething(a[0]);
} catch (e) {
  Raven.captureException(e);
}

```

В данном примере (вызов несуществующей функции с использованием функции «captureException») будет записано сообщение об ошибке. Важно, что текст «sentry-url» необходимо заменить на URL, выданный на сайте Sentry конкретному разработчику.

Создание web-сокета для отправки сообщений всем клиентам. Socket.IO – событийно-ориентированная библиотека JavaScript для web-

приложений, предназначенная для создания web-сокетов и обмена данными в реальном времени. Она состоит из двух частей: клиентской, которая работает в браузере, и серверной (при этом они имеют похожее API). Для использования необходимо загрузить модуль «socket.io».

Пример серверной части приложения:

```
import { createServer } from "http";
import { Server } from "socket.io";
import fs from "fs/promises"
const httpServer = createServer(function (req, res) {
  fs.readFile("./index.html")
    .then(html => {
      res.setHeader("Content-Type", "text/html");
      res.writeHead(200);
      res.end(html);
    })
});
const io = new Server(httpServer, { /* options */ });
io.on("connection", (socket) => {
  socket.emit("hello", "Message from server")
});
httpServer.listen(3000);
```

В данном примере создаётся простейший http-сервер на порту 3000, который возвращает единственную страницу index.html. Поверх web-сервера создаётся web-сокет, который будет передавать сообщения. Единственное действие, которое выполняет данный сервер – при присоединении клиента («connection») передаёт сообщение типа «hello» с текстом «Message from server». В данном случае socket выступает в качестве EventEmitter.

Пример клиентской части приложения – файл index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <script src="/socket.io/socket.io.js"></script>
  <script>
    const socket = io();
    socket.on("connect", () => {
      data.innerHTML +=
        `<p>connect: ${socket.id}</p>`
    });
    socket.on("disconnect", () => {
      data.innerHTML +=
        `<p>disconnect: ${socket.id}</p>`
    });
    socket.on("hello", (msg)=>{
      data.innerHTML += `<p>msg: ${msg}</p>`;
    });
  </script>
```

```

</head>
<body>
<h3>Socket.IO</h3><div id="data"></div>
</body>
</html>

```

В данном примере создаётся сокет для работы с сервером. При присоединении (сообщение «connect») выводится идентификатор сокета, при получении сообщения типа «hello» данное сообщение выводится в соответствующий div, при разрыве соединения (сообщение «disconnect») сообщает о разрыве соединения.

В приведенных примерах осуществляется односторонняя передача сообщения от сервера – клиенту. Рассмотрим более насыщенный пример, в котором осуществляется двусторонняя передача сообщений, при этом сообщения будем передавать в формате JSON для удобства обработки.

Пример серверной части приложения:

```

io.on("connection", (socket) => { // При подключении клиента
  socket.on('conn', (msg)=>{ // Сообщение "conn"
    let time = (new Date()).toLocaleTimeString();
    socket.name = msg.name // Сохранение имени
    socket.emit("msg", {"message": `${time} Привет
${socket.name}!`});
    socket.broadcast.emit("msg", {"message": `${time} Вошёл
${socket.name}!`});
  });
  socket.on('msg', (msg)=>{ // Сообщение "msg"
    let time = (new Date()).toLocaleTimeString();
    msg = `${time} ${socket.name}: ${msg.value}` // Сообщение
    socket.emit("msg", {"message": msg}); // Отправка "обратно"
    socket.broadcast.emit("msg", {"message": msg}); /* Отправка
всем */
  });
});

```

Строчки загрузки и нового соединения мы рассмотрели на предыдущем примере. Сокет «socket» настраивается на обработку событий «conn» и «msg».

В случае события «conn» в объект «socket» дописывается свойство «name», при этом предполагается, что у объекта сообщения есть такое свойство. В случае события «msg» у объекта сообщения ожидается наличие свойства «value».

Отправка осуществляется с использованием функции socket.emit() для отправки сообщения обратно отправителю, функции socket.broadcast.emit() для отправки сообщения всем остальным, кто подключен к web-сокету.

Пример клиентской части приложения:

```
<html lang="ru">
<head><meta charset="UTF-8">
  <script src="/socket.io/socket.io.js"></script>
  <script>
    let socket, nick
    function chat() {
      socket = io();
      nick = document.getElementById("name").value
      socket.on("connect", () =>
        { socket.emit("conn", {"name": nick})
        });
      socket.on("msg", (msg)=> { addUL(msg.message) });
      document.getElementById("send").disabled = false;
      document.getElementById("login").disabled = true;
    }
    function addUL(msg) {
      const li = document.createElement("li")
      data.appendChild((li.innerHTML = msg, li))
    }
    const send = (msg) =>
      socket && socket.emit("msg", {"name": nick, "value": msg})
  </script>
</head>
<body>
  <input id="name" placeholder="Введите имя">
  <button id="login" onclick="chat()">Вход</button><br>
  <input id="msg" placeholder="Сообщение">
  <button id="send" disabled onclick="send(msg.value)">
    Отправить </button><br>
  <ul id="data"></ul>
</body>
</html>
```

Элемент `<input id='name'>` хранит имя пользователя, элемент `<button id='send'>` используется для для отправки, список `<ul id='data'>` предназначен для вывода сообщений от сервера. Основные элементы для создания соединения мы рассмотрели в предыдущем примере. Заведена переменная для сокета «socket», она используется в нескольких функциях.

Функция «chat» обеспечивает соединение с сервером, отправку имени пользователя и настройку сокета. Вызов «socket.emit» с типом «conn» обеспечивает отправку имени пользователя на сервер.

Функция «addUL», вызываемая в обработчике сообщения «msg», добавляет полученное сообщение в список.

Функция «send» обеспечивает отправку сообщения на сервер.

Обратите внимание, что при отправке сообщения (вызов «socket.emit») они создаются в виде JSON-объектов, при получении ожидается, что от сервера получены JSON-объекты, в частности, у них есть поле «message».

Создание и запуск Jest тестов для приложения. Модуль «jest» (должен быть установлен) позволяет реализовывать модульное тестирование. По умолчанию jest ожидает, что все тесты находятся в папке «tests» и в названии файла содержат слово «test», например, «tests/sum.test.js».

Пусть тестируемый файл называется «src/sum.js» и содержит следующий код.

```
function sum(a, b) { return a + b; }  
module.exports = sum;
```

Тогда модульный тест «tests/sum.test.js» может выглядеть следующим образом.

```
const sum = require('../src/sum');  
test('adds 1 + 2 to equal 3', () => {  
  expect(sum(1, 2)).toBe(3);  
});
```

Функция expect() вызывает тестируемую функцию, передаёт ей необходимые параметры, а затем сравнивает с использованием toBe() с ожидаемым значением. Вариантов сравнения может быть множество, например:

- toBeNull() – истинно для возвращаемого значения null;
- toBeGreaterThan(3) – истинно для возвращаемого значения больше 3;
- toBeLessThanOrEqual(3) – истинно для возвращаемого значения меньше или равного 3.

При необходимости тесты могут группироваться с использованием функции «describe».

```
describe('Группа тестов', () => { test(...); test(...); ...})
```

Большинство сред разработки умеют автоматически подключать Jest. Для «ручного» запуска можно сконфигурировать запуск с использованием «package.json»:

```
"scripts": {  
  "test": "jest"  
}
```

В таком случае можно использовать команду «npm run test», при этом будут выполнены все тесты в папке «tests».

Вопросы для контроля

1. Где выполняется Angular – в браузере или на сервере? Как установить Angular, создать приложение и запустить его исполнение?

2. Что такое компонент в Angular? Приведите пример его создания и подключения.

3. В чем отличие компонентов и сервисов? Приведите пример создания и подключения сервисов.

4. Какие существуют варианты использования шаблонов в Angular? Приведите варианты использования шаблонов, примеры использования директив условия и цикла.

5. Для чего предназначены директивы в Angular? Приведите пример создания и использования директивы.

6. Приведите примеры использования обещаний и обработки ошибок.

7. Какой журнал ошибок использовался в лабораторной работе? Приведите пример использования.

8. Что такое WebSocket? Приведите пример создания web-сокета на серверном и на клиентском приложении.

9. Приведите пример Jest-теста.

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

- Angular. One framework. Mobile & desktop // URL: <https://angular.io/>
- StackBlitz. Online VS Code IDE for Angular & React // URL: <https://stackblitz.com/>
- Руководство по Angular 13 // URL: <https://metanit.com/web/angular2/>
- Promise // URL: <https://learn.javascript.ru/promise>
- Rollbar. Catch errors before your users do // URL: <https://rollbar.com/>
- JavaScript error tracking with Sentry // URL: <https://sentry.io/for/javascript/>
- Socket.IO // URL: <https://socket.io/>
- Jest // URL: <https://jestjs.io/ru/>
- cors // URL: <https://www.npmjs.com/package/cors>
- Cross-Origin Resource Sharing (CORS) // URL: <https://developer.mozilla.org/ru/docs/Web/HTTP/CORS>

Лабораторная работа 5. МОДУЛЬ АДМИНИСТРИРОВАНИЯ ПРИЛОЖЕНИЯ «БИРЖА АКЦИЙ»

Цель и задачи

Целью работы является изучение возможностей применения библиотеки React (<https://reactjs.org/>) для разработки интерфейсов пользователя web-приложений и использование фреймворка NestJS (<https://nestjs.com/>) для разработки серверных приложений. Для достижения поставленной цели требуется решить следующие задачи:

- разработка интерфейса web-приложения;
- создание web-сервера на основе NestJS. Подготовка web-сокетов для обновления информации о стоимости у всех клиентов;
- создание каркаса клиентского web-приложения с использованием React;
- создание каркаса серверного web-приложения с использованием NestJS;
- разработка перечня компонентов;
- создание статической версии интерфейса;
- определение минимального и достаточного набора состояний интерфейса;
- определение жизненного цикла состояний;
- программирование потока изменения состояний.

Основные теоретические сведения

React – библиотека на JavaScript для построения интерфейса пользователя. React представляется удобным инструментом для создания масштабируемых web-приложений (в данном случае речь идет о клиентской части), особенно в тех ситуациях, когда приложение является одностраничным.

В основу React заложены принципы Redux, предлагающее предсказуемый контейнер хранения состояния web-приложения.

Вся структура веб-страницы может быть представлена с помощью DOM. Для решения проблемы производительности предложена концепция виртуального DOM, который представляет собой облегченную версию DOM. React работает именно с виртуальным DOM. Реализован механизм, который периодически сравнивает виртуальный DOM с реальным и вычисляет минимальный набор манипуляций для приведения реального DOM к состоянию, которое хранится в виртуальном DOM.

NestJS – фреймворк для разработки серверных приложений на языках JavaScript и TypeScript. Фреймворк построен на основе компонентного под-

хода и предлагает стандартизованную структуру приложения по аналогии с Angular.

Общая формулировка задачи

Необходимо создать web-приложение, обеспечивающее настройку биржи брокера, в которой есть возможность задать перечень участников, перечень акций, правила изменения акций во времени. Основные требования следующие:

1. Информация о брокерах (участниках) и параметрах акций сохраняется в файле в формате JSON.

2. В качестве сервера используется NestJS с использованием языка TypeScript.

3. Предусмотрена HTML-страница с перечнем потенциальных брокеров. Брокеров можно добавлять и удалять, можно изменить начальный объем денежных средств.

4. Предусмотрена HTML-страница для перечня акций. Есть возможность просмотреть перечень доступных акций (обозначение, название компании) и исторические данные по изменению курса не менее чем за текущий и предыдущий год. Есть возможность выбрать какие акции будут участвовать в торгах. Минимально должны поддерживаться следующие компании (в скобках – обозначение): Apple, Inc. (AAPL), Starbucks, Inc. (SBUX), Microsoft, Inc. (MSFT), Cisco Systems, Inc. (CSCO), QUALCOMM Incorporated (QCOM), Amazon.com, Inc. (AMZN), Tesla, Inc. (TSLA), Advanced Micro Devices, Inc. (AMD).

Реальные исторические данные по изменению курса доступны по адресу: <https://www.nasdaq.com/market-activity/quotes/historical>.

Фрагмент данных для AAPL за три дня (переведён в формат json, оставлены только два столбца: дата и стоимость на время начала торгов):

```
[{"date": "11/5/2021", "open": "$151.89"},  
{"date": "11/4/2021", "open": "$151.58"},  
{"date": "11/3/2021", "open": "$150.39"}]
```

5. Предусмотрена HTML-страница для настроек биржи (дата начала торгов, скорость смены дат в секундах при имитации торгов). На этой же странице должна быть кнопка «Начало торгов», которая запускает процесс имитации торгов и предоставление информации об изменении курсов акций всем брокерам по web-сокетах с учётом заданных настроек биржи, здесь же долж-

на отображаться текущая имитируемая дата торгов и текущая стоимость каждой акции.

6. Все элементы в клиентском приложении реализованы с использованием компонентов React. Маршрутизация реализована с использованием «react-router-dom».

7. Для хранения общих данных используется Redux.

8. На сервере спроектированы компоненты и сервисы NestJS для имитации торгов и обработки запросов клиентского приложения.

9. Исторические данные по котировкам представляются как в виде таблиц, так и в виде графиков (например, с использованием Chart.js).

10. Приложение должно реализовывать responsive-интерфейс и корректно работать в том числе при просмотре с мобильного телефона.

11. Для всех страниц web-приложения разработан макет интерфейса с использованием Figma (<https://www.figma.com/>).

Преимуществом будет создание и использование аутентификации на основе passport.js (<http://www.passportjs.org/>).

Преимуществом будет использование Material UI React (<https://mui.com/ru/>).

Описание ключевых методов при выполнении работы

Создание каркаса web-приложения с использованием React. Для создания нового React приложения можно установить глобально модуль «create-react-app» и воспользоваться следующей командой:

```
create-react-app my-react
```

Здесь «my-react» – название создаваемого React-приложения.

Для запуска приложения необходимо выполнить следующую команду:

```
npm run start
```

Приложение запустится по адресу <http://localhost:3000/>.

Основной файл приложения «src/App.js».

Компоненты React. Компонент React – это «фрагмент кода», который представляет собой часть веб-страницы. Создание простейшего компонента («src/App.js»):

```
import React from 'react';
function App () {
  return <h1>Привет React</h1>
}
export default App;
```

В данном случае на web-странице отобразится простейшее приветствие «Привет React». Возвращаемый объект — JSX-объект (<https://facebook.github.io/jsx/>), который является «смесью» HTML и JavaScript.

Другой вариант создания компонента («src/App.js»):

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return <h1>Привет React</h1>
  }
}
export default App;
```

Результат на web-странице не изменяется. Первый вариант создания компонента — функциональный, второй — с использованием классов. Класс расширяет «Component» и реализует функцию «render()», которая возвращает JSX.

В JSX можно использовать переменные JavaScript, для этого их необходимо поместить в фигурные скобки:

```
import React, { Component } from 'react';
export default class App extends Component {
  render() {
    let hello = "Привет"
    return <h1>{hello} React!</h1>
  }
}
```

Фрагменты JSX можно создавать и за пределами функций и классов:

```
import React, { Component } from 'react';
const html = <h1>2 + 2 = {2 + 2}</h1>
export default function App () {
  return html
}
```

В данном примере на web-страницу будет выведено «2 + 2 = 4».

В варианте HTML-страницы, в которую напрямую подключены необходимые скрипты (react, react-dom и babel) «public/test.html». Страница будет доступна по адресу <http://localhost:3000/test.html>:

```
<html>
<body>
<div id="app"></div>
<script
src="https://unpkg.com/react@16/umd/react.production.min.js">
</script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"> </script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
```

```
standalone/6.25.0/babel.min.js"> </script>
<script type="text/babel">
  ReactDOM.render(
    <h1>2 + 2 = {2 + 2}</h1>,
    document.getElementById("app")
  )
</script>
</body>
</html>
```

Обратите внимание, что тип скрипта «text/babel». Функция «ReactDOM.render()» принимает два параметра, обеспечивает формирование соответствующего React-компонента и отображение его в <div id="app">.

Компоненты могут вкладываться один в другой:

```
import React, { Component } from 'react';
function Sum (props) {
  const x = !props.x ? 2 : parseInt(props.x)
  return <h2>{x} + {x} = {x + x}</h2>
}
export default function App () {
  return (<div><Sum x="3"/><Sum/></div>)
}
```

В данном примере создано два компонента «Sum» и «App». Компонент «Sum» принимает параметры. Если параметры не заданы, то отображается сумма для «2», иначе – сумма для переданного числа. Компонент «App» с использованием <div> объединяет два компонента «Sum» с разными параметрами.

Следует обратить внимание на некоторые моменты:

- передаваемые в компонент параметры попали в переменную «props» в виде свойств объекта;
- для объединения компонентов в «App» используется <div> – React-компонент должен возвращать один элемент, если необходимо вернуть несколько, то их следует объединить в один;
- возвращаемое значение в «App» объединено в круглые скобки. Это позволяет писать его на нескольких строчках.

Состояние компонента и обработка событий. Состояние компонента хранится в переменной «state». При этом в соответствии с соглашением по программированию нельзя напрямую изменять значения свойств состояния, можно только вызывать специальный метод:

```
import './App.css'
import React from 'react'
class App extends React.Component {
```

```

constructor(props) {
  super(props); // Настройка свойств в конструкторе
  // Состояние
  this.state = {class: "off", label: "Нажми"};
  // Привязка контекста функции
  this.toggle = this.toggle.bind(this);
}
toggle(event) {
  // Вычисление класса toggle
  let className = (this.state.class === "off") ? "on" : "off";
  // Установка нового состояния
  this.setState({class: className});
}
render() {
  return <button onClick={this.toggle}
    className={this.state.class}>{this.state.label}
  </button>;
}
}
export default App

```

Для работы с состоянием компонент создается с использованием класса и расширяет «Component». У конструктора обязательно есть параметр «props», который передается в «super». Следующей строкой настраивается начальное состояние (класс и текст метки на кнопке), так как будет осуществляться вызов функции «toggle()», то ее обязательно необходимо привязать к контексту «bind(this)».

При вызове «toggle()» изменяется на противоположное значение класса. Обратите внимание, что обращение к значению состояния – «this.state.class», а при установке значения вызывается «this.setState({class: ...})». Это означает, что в режиме чтения мы получаем значение переменной напрямую, а для установки состояния используем метод, который заменит часть состояния (при этом значение «this.state.label» не изменится). В качестве параметра в функцию попадает объект события «event» (в данном примере мы его не используем).

Функция «render()» возвращает кнопку, в которой из текущего состояния используется «this.state.label» в качестве текста, «this.state.class» в качестве класса (важно, что в данном случае используется «className», а не «class»), событие «onClick» вызывает функцию «this.toggle».

В результате на html-странице отображается кнопка, которая меняет стиль при нажатии кнопки. Естественно, стили должны быть в файле «App.css»:


```
.off { color: gray; }  
.on { color: red; }
```

В данном примере задействованы два важнейших этапа жизненного цикла компонента: «`constructor(props)`» и «`render()`». Кроме того могут оказаться полезными «`componentDidMount()`», который вызывается сразу после подключения компонента, и «`componentWillUnmount()`», который вызывается перед удалением компонента. Эти две функции полезны при настройке и освобождении ресурсов, необходимых компоненту.

Расширим форму элементами ввода:

```
import React, {Component} from 'react'  
class App extends Component {  
  constructor(props) {  
    super(props)  
    this.state = {text: "", outtext: ""}  
    this.onChange = this.onChange.bind(this)  
    this.onSubmit = this.onSubmit.bind(this)  
  }  
  onChange(e) {  
    let value = e.target.value  
    this.setState({text: value})  
  }  
  onSubmit(e) {  
    e.preventDefault()  
    this.setState({outtext: this.state.text})  
  }  
  render() {  
    return (  
      <form onSubmit={this.onSubmit}>  
        <input type="text" value={this.state.text}  
placeholder="Текст" onChange={this.onChange}>/>  
        <input type="submit" value="Отправить" /><br/>  
        <pspan{this.state.outtext}</span>"</p>  
      </form>  
    );  
  }  
}  
export default App
```

В конструкторе определяем состояние, содержащее два текстовых поля, и привязываем две функции к «`this`». Функция «`onChange()`» будет использоваться на элементе `<input>`, поэтому результат ввода пользователя может быть получен из «`e.target.value`». Функция «`onSubmit()`» будет использоваться на кнопке «`submit`», поэтому в ней необходимо запретить выполнение действия по умолчанию «`e.preventDefault()`», основная задача функции – копирование значения из «`this.state.text`» в «`this.state.outtext`».

Компонент отображает форму ввода, которая при вводе вызывает функцию «onSubmit()», в поле ввода отображается значение «this.state.text», а при изменении текста вызывается функция «onChange». Результат нажатия кнопки «Отправить» отображается в элементе ``.

Маршрутизация в React. Для маршрутизации потребуется дополнительный модуль «react-router-dom»:

```
import React from "react";
import ReactDOM from "react-dom";
import {BrowserRouter, Switch, Route, Link} from "react-router-dom";
function BasicExample() {
  return (
    <BrowserRouter>
      <div>
        <ul>
          <li><Link to="/">Home</Link></li>
          <li><Link to="/about">About</Link></li>
        </ul>
        <hr />
        <Switch>
          <Route exact path="/"><Home/></Route>
          <Route path="/about"><About/></Route>
          <Route path="*"><NoMatch/></Route>
        </Switch>
      </div>
    </BrowserRouter>
  );
}
function Home() { return <div><h3>Home</h3></div> }
function About() { return <div><h3>About</h3></div> }
function NoMatch() { return <div><h3>No match!</h3></div> }
ReactDOM.render(<BasicExample />, document.getElementById('root'));
```

В данном примере создается четыре компонента: «About», «Main», «NoMatch» и «BasicExample». Основным компонентом «BasicExample» становится «BrowserRouter», который с использованием «Switch» выбирает указанный путь «path». В зависимости от указанного пути отображается тот или иной компонент. Компонент «NoMatch» отображается для всех остальных маршрутов.

В данном примере не использовался «Link», который предназначен для отображения ссылок на странице с переходом по маршрутам. Формат использования приведен ниже.

```
<Link to="/about">about</Link>
```

Для чтения параметров URL (в примере «id») и запроса (в примере «name») можно изменить компонент Home.

```
class Home extends React.Component{
  render() {
    const match = this.props.match;
    const loc = this.props.location;
    return <div>
      <b>match:</b> {JSON.stringify(match)}<br/>
      <b>location:</b> {JSON.stringify(loc)}<br/>
      <b>id:</b> {match.params.id}<br/>
      <b>name:</b> {new URLSearchParams(loc.search).get("name")}
    </div>;
  }
}
```

Управление состоянием приложения. Redux (<https://redux.js.org/>) представляет собой контейнер для управления состоянием web-приложения. Redux не привязан непосредственно к React.js и может использоваться с другими js-библиотеками и фреймворками.

Ключевые понятия Redux:

- хранилище (store) – хранит состояние приложения;
- действия (actions) – набор команд, которые отправляются приложением в хранилище;
- создатели действий (action creators) – функции, которые создают действия;
- reducer – функции, которые получают действия и в соответствии с ними изменяют состояние хранилища.

При этом формируется следующая последовательность: представление (view) создает действие (action), в соответствии с действием reducer изменяет информацию в хранилище (store), информация из хранилища отображается пользователю (view).

Простой пример использования Redux требует установки модуля «redux».

```
import { createStore } from 'redux'
// reducer - применяет действие к текущему состоянию
function counterReducer(state = { value: 0 }, action) {
  switch (action.type) {
    case 'counter/inc': return { value: state.value + 1 }
    case 'counter/dec': return { value: state.value - 1 }
    default: return state
  }
}
// Создание Redux-хранилища
```

```
// API: { subscribe, dispatch, getState }.
let store = createStore(counterReducer)
// Рассмотрим простой пример с subscribe()
store.subscribe(() => console.log(store.getState()))
store.dispatch({ type: 'counter/inc' }) // {value: 1}
store.dispatch({ type: 'counter/inc' }) // {value: 2}
store.dispatch({ type: 'counter/dec' }) // {value: 1}
```

Функция `createStore()` принимает в качестве параметра `reducer` и создаёт хранилище, которое хранит полное состояние приложения. В данном примере «counter» – хранимый параметр приложения (хранится в `state.value`). Предусмотрены два типа действий – «counter/inc» и «counter/dec».

Возможна реализация с использованием модуля «@reduxjs/toolkit».

```
import { createSlice, configureStore } from '@reduxjs/toolkit'
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    inc: state => { // Используется Immutable.JS
      state.value += 1
    },
    dec: state => {
      state.value -= 1
    }
  }
})
const { inc, dec } = counterSlice.actions
const store = configureStore({
  reducer: counterSlice.reducer
})
// Можно подписаться
store.subscribe(() => console.log(store.getState()))
// Отправка действий диспетчеру
store.dispatch(dec()) // {value: -1}
store.dispatch(dec()) // {value: -2}
store.dispatch(inc()) // {value: -1}
```

Здесь `slice` – коллекция `reducers` и действий для отдельных хранимых параметров приложения, которые обычно описываются в отдельных файлах. Функция `createSlice()` упрощает логику `reducer` и действий – явно указывается имя характеристики, начальное состояние и все возможные действия. Функция `configureStore()` создаёт единое хранилище для приложения.

Использование Redux в связке с React хорошо описано в примере: <https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/todos>.

Создание графиков с использованием Chart.js. Простейшая реализация графика предполагает, что есть исходные данные (labels) и соответствующие

данные (data). Реализуем отображение данных курса APPL за один месяц 2021 года.

```
<html>
<head>
  <script
src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
<body>
<div style="width: 400px; height: 400px">
  <canvas id="myChart" width="100%" height="100%"></canvas>
</div>
<script>
  const ctx = document.getElementById('myChart');
  const labels = ['10/6/2021', '10/7/2021', '10/8/2021',
'10/11/2021', '10/12/2021', '10/13/2021', '10/14/2021',
'10/15/2021', '10/18/2021', '10/19/2021', '10/20/2021',
'10/21/2021', '10/22/2021', '10/25/2021', '10/26/2021',
'10/27/2021', '10/28/2021', '10/29/2021', '11/1/2021',
'11/2/2021', '11/3/2021', '11/4/2021', '11/5/2021' ];
  const values = [139.47, 143.06, 144.03, 142.27, 143.23,
141.24, 142.11, 143.77, 143.45, 147.01, 148.7, 148.81, 149.69,
148.68, 149.33, 149.36, 149.82, 147.22, 148.99, 148.66, 150.39,
151.58, 151.89]
  const data = {
    labels: labels,
    datasets: [{
      label: 'AAPL',
      backgroundColor: 'rgb(255, 99, 132)',
      borderColor: 'rgb(255, 99, 132)',
      data: values,
    }]
  }
  const myChart = new Chart(ctx, {
    type: 'line',
    data
  });
</script>
</body>
</html>
```

В данном примере подключена библиотека chart.js, есть соответствующий модуль, который можно скачать и установить, созданы метки с данными (labels) и соответствующие значения (value), подготовлена конфигурация в переменной data, создан график Chart, которому в качестве параметра передан <canvas>, тип графика и конфигурация графика. Обратите внимание, что данных много и библиотека автоматически масштабирует количество видимых узлов.

Вопросы для контроля

1. Что такое JSX? Приведите примеры создания и использования.
2. Опишите и приведите примеры различных способов создания компонентов.
3. Приведите примеры обработки событий и передачи информации между компонентами.
4. Как осуществляется работа с состоянием React-приложения?
5. Как выполняется маршрутизация в React-приложении?
6. Что такое Redux? Какие возможности он предоставляет?

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

- React. A JavaScript library for building user interfaces // URL: <https://reactjs.org/>;
- Draft: JSX Specification // URL: <https://facebook.github.io/jsx/>;
- React.js на русском языке. Перевод официальной документации // URL: <https://abraxabra.ru/react.js/>;
- Руководство по React // URL: <https://metanit.com/web/react/>;
- Online VS Code IDE for Angular & React // URL: <https://stackblitz.com/>;
- Redux // URL: <https://redux.js.org/>;
- Immutable // URL: <https://www.npmjs.com/package/immutable>;
- Chart.js // URL: <https://www.chartjs.org/>.

Лабораторная работа 6. МОДУЛЬ ПРИЛОЖЕНИЯ «ПОКУПКА И ПРОДАЖА АКЦИЙ»

Цель и задачи

Целью работы является изучение возможностей применения фреймворка Vue (<https://v3.ru.vuejs.org/ru/>) для разработки интерфейсов пользователя web-приложений и организации E2E тестирования клиентской части приложения. Для достижения поставленной цели требуется решить следующие задачи:

- разработка интерфейса web-приложения;
- создание web-сервера на основе NestJS. Подготовка web-сокетов для обновления информации о стоимости у всех клиентов;

- создание каркаса web-приложения с использованием Vue;
- разработка перечня компонентов;
- создание статической версии интерфейса;
- программирование потока изменения состояний web-приложения;
- разработка скрипта автоматического тестирования web-приложения.

Основные теоретические сведения

Vue – фреймворк для создания пользовательских интерфейсов. В отличие от фреймворков-монолитов, Vue создавался пригодным для постепенного внедрения. Его ядро решает задачи уровня представления, упрощая интеграцию с другими библиотеками и существующими проектами. С другой стороны, Vue подходит и для разработки сложных одностраничных приложений.

Selenium – фреймворк тестирования web-приложений, позволяет программировать автоматизированные тесты клиентской части web-приложений, а также записывать и воспроизводить действия пользователей.

Headless-браузер – браузер без интерфейса пользователя, предназначенный для тестирования web-приложений.

Общая формулировка задачи

Необходимо создать web-приложение, обеспечивающее работу брокера, у него есть запас денежных средств, он имеет возможность купить или продать акции (любое доступное количество), а также контролировать изменение котировок акций. В приложении должен отображаться баланс (запас денежных средств плюс стоимость акций), а также прибыль или убыток, которые он получил по каждой акции. Основные требования следующие:

1. Приложение получает исходные данные из модуля администрирования приложения «Биржа акций» в виде настроек в формате JSON-файла и в виде данных от web-сокета по изменению стоимости акций во времени.

2. В качестве сервера используется NestJS.

3. Участники торгов подключаются к приложению «Покупка и продажа акций».

4. Предусмотрена HTML-страница администратора, на которой отображается перечень участников. Для каждого участника отображается его баланс, количество акций каждого типа у каждого участника и его прибыль или убыток по каждой акции в текущий момент времени.

5. Предусмотрена HTML-страница входа в приложение, где каждый участник указывает (или выбирает из допустимых) свое имя.

6. Предусмотрена HTML-страница, на которой участнику отображаются:

- текущая имитируемая дата;
- текущая стоимость каждой из акций, выставленных на торги;
- общее количество доступных средств;
- количество, стоимость и прибыль/убыток по каждой купленной акции.

На этой же странице у брокера есть возможность:

– открыть диалоговое окно просмотра графика изменения цены каждой акции (с момента начала торгов до текущего момента) с учётом сообщений об изменении стоимости акций;

- купить/продать интересующее его количество акций.

Комментарии:

- брокер не может купить акции, если денег не хватает;
- купля/продажа происходит «мгновенно».

7. Разработаны автоматизированные тесты для проверки корректности работы клиентской части web-приложения с использованием headless-браузера или фреймворка Selenium. Как минимум необходимо проверить, что при покупке/продаже N акций в определённую дату соответствующим образом изменяется баланс средств брокера и через некоторое время получается правильная прибыль/убыток по данной акции.

Преимуществом будет использовать Material Design Framework (<https://vuetifyjs.com/en>).

Описание ключевых методов при выполнении работы

Создание каркаса web-приложения с использованием Vue. Для создания нового Vue приложения можно установить глобально модуль «@vue/cli» и воспользоваться следующей командой:

```
vue create vue-app
```

Здесь «vue-app» – название создаваемого Vue-приложения.

Для запуска приложения необходимо выполнить следующую команду:

```
npm run start
```

Приложение запустится по адресу <http://localhost:8080/>.

Основной файл приложения «src/main.js», основной компонент – «src/App.vue». В простейшем случае основной файл может выглядеть следующим образом.


```
import { createApp } from 'vue'
import App from './App.vue'
createApp(App).mount('#app')
```

Здесь создаваемый компонент будет отображён на основной странице index.html в элементе с идентификатором «app».

Компоненты Vue. Компонент Vue – это «фрагмент кода», который представляет собой часть веб-страницы. Создание простейшего компонента («src/App.vue»):

```
<template>
  <div>
    <h3>{{msg}}</h3>
  </div>
</template>

<script>
export default {
  name: 'App',
  data() { return { msg: "Привет Vue" } }
}
</script>
<style scoped>
h3 { color: green; }
</style>
```

В данном случае на web-странице отобразится простейшее приветствие «Привет Vue». Следует обратить внимание, что компонент Vue состоит из трёх частей – шаблона (template), JavaScript-кода (script) и CSS-настроек (style). Параметры компонента из JavaScript могут отображаться в шаблоне с использованием привязки. Для привязки в теле элемента используются двойные фигурные скобки, например, {{msg}}. В теле элемента в двойных фигурных скобках может использоваться одно JavaScript выражение, возвращающее значение для отображения. В шаблоне должен находиться один HTML-элемент, который может объединять в себе множество других HTML-элементов.

Vue-компоненты могут содержать свои свойства (data), получать свойства из родительского компонента (props), вызывать функции (methods), содержать вычисляемые поля (computed), хуки жизненного цикла компонента и т.п.

Предположим, что родительский компонент называется «App.vue».

```
<template>
  <HelloWorld msg="Первое Vue3 приложение"/>
</template>
<script>
import HelloWorld from './components/HelloWorld.vue'
export default {
```

```

    name: 'App',
    components: { HelloWorld }
  }
</script>

```

В данном примере «App.vue» создаётся дочерний компонент «components/HelloWorld.vue», в него передаётся параметр `msg`. Рассмотрим пример дочернего компонента.

```

<template>
  <div>
    <h3 v-bind:title="text">{{msg}}</h3>
    <button v-on:click="reverse">Перевернуть</button>
  </div>
</template>
<script>
export default {
  name: 'HelloWorld',
  props: { msg: String },
  data() { return { text: "Тестовый текст" } },
  methods: { // Методы компонента
    reverse() {
      this.text = this.text.split('').reverse().join('')
    }
  }
}
</script>

```

В данном примере «HelloWorld.vue» получает строковый параметр «`msg`» из родительского компонента, отображая его в теле элемента `<h3>`, в качестве подсказки (`title`) у этого же элемента используется свойство компонента «`text`». Привязка к свойству осуществляется с помощью директивы «`v-bind:`» (альтернативное обозначение «`:`»). Компонент также отображает кнопку (`button`), к которому с помощью директивы «`v-on:`» (альтернативное обозначение «`@`») на событие «`click`» подключен метод компонента «`reverse`». Указанный метод переворачивает текст в свойстве «`text`». Обратите внимание, что обращение к свойству осуществляется с использованием указателя «`this`».

Привязка с использованием директивы «`v-bind:`» является односторонней, для двусторонней привязки используется директива «`v-model`».

```

<input v-model="question"/>

```

В приведённом примере «`question`» должен быть объявлен как свойство компонента, тогда его изменение будет отображаться в `<input>` и наоборот.

Как и в большинстве фреймворков Vue поддерживает передачу свойств компонентам-потомкам и в обратную сторону – передачу событий. Рассмотрим пример компонента-родителя «App.vue».

```

<template>
  <div>
    <Button btn-label="Сч. A" idx="0" @inc-value="increment">
  </Button>
    <Button btn-label="Сч. B" idx="1" @inc-value="increment">
  </Button>
    <div>Счётчики: A={{counter[0]}}, B={{counter[1]}}</div>
  </div>
</template>
<script>
import Button from "@components/Button";
export default {
  name: 'HelloWorld',
  components: {Button},
  data() { return { counter: [0, 0] } },
  methods: { increment(index) { this.counter[index]++ } }
}
</script>

```

В данном примере созданы два компонента «Button», каждому из которых передано имя «btn-label», в дочернем компоненте оно трансформируется по правилу CamelCase в «btnLabel», передан индекс «idx» в массиве «counter», указана функция, которая будет обрабатывать события, получаемые от дочернего компонента «inc-value», в дочернем компоненте трансформируется – в «incValue». При поступлении событий от дочернего компонента будет вызываться функция «increment», ожидается, что в качестве параметра у этой функции будет указан индекс в массиве «counter». Значения «counter» отображаются в соответствующем <div> в компоненте.

Рассмотрим пример «Button.vue».

```

<template>
  <button @click="$emit('incValue', idx)">
    {{ btnLabel }}
  </button>
</template>
<script>
export default {
  name: "Button",
  props: ["btnLabel", "idx"]
}
</script>

```

Компонент «Button» получает от родительского компонента два свойства «btnLabel» и «idx». Значение «btnLabel» отображается в качестве текста кнопки <button>, значение «idx» передаётся в качестве параметра генерируемого события EventEmitter с помощью «\$emit», где «incValue» – имя создаваемого события, после имени через запятую перечисляются параметры, передаваемые с событием.

Маршрутизация в Vue. Для маршрутизации потребуется дополнительный модуль «vue-router». При создании приложения в «main.js» необходимо подключить маршруты.

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
createApp(App).use(router).mount('#app')
```

В данном примере маршруты загружаются из файла «router.js».

```
import { createRouter, createWebHashHistory } from 'vue-router'
import Home from '../views/Home.vue'
const routes = [
  { path: '/', name: 'Home', component: Home },
  { path: '/about', name: 'About',
    // Создаст компонент, lazy-подгружаемый при первом обращении
    component: () => import('../views/About.vue')
  }
]
const router = createRouter({
  history: createWebHashHistory(),
  routes
})
export default router
```

В приведённом примере предусмотрено два пути «/», обрабатываемый компонентом «Home» и «/about», обрабатываемый компонентом «About». Последний загружается в память браузера при первом обращении к компоненту.

Для корректного отображения маршрутов и ссылок переходов между компонентами необходимы соответствующие изменения в «App.vue».

```
<template>
  <div id="nav">
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link>
  </div>
```

```
<router-view/>
</template>
```

В данном случае с использованием «router-link» создаются две ссылки на странице с переходами на «/» и «/about». Результаты нажатия на ссылки будут отображаться в «router-view».

Для обработки параметров маршрута, например, в «About.vue» можно воспользоваться следующим примером.

```
<template>
  <div>
    <div>params={{ $route.params }}</div>
    <div>query={{ $route.query }}</div>
    <div>path={{ $route.path }}</div>
    <div>id={{ $route.params.id }}</div>
  </div>
</template>
```

Здесь «\$route.params» перечислит имена и значения всех полученных параметров в URL, «\$route.query» перечислит имена и значения всех полученных параметров запроса, «\$route.path» вернёт полный путь запроса, а «\$route.params.id» вернёт значение единственного параметра URL по имени «id». Следует отметить, что для корректной работы с параметрами URL соответствующие изменения должны быть внесены в описания маршрута в «router.js», в частности строка «about» для обработки «id» должна измениться следующим образом.

```
{ path: '/about/:id', name: 'About', component: About }
```

Управление состоянием приложения. Vuex (<https://next.vuex.vuejs.org/>) представляет собой контейнер для управления состоянием web-приложения. Для работы с единым состоянием приложения его необходимо подключить в «main.js» (можно последовательно с маршрутизатором).

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'
createApp(App).use(store).use(router).mount('#app')
```

В данном примере единое состояние приложения описано в «store.js».

```
import { createStore } from 'vuex'
export default createStore({
  state () { return { count: 0 } },
  mutations: {
    increment (state) { state.count++ }
```

```

    },
    getters: { // Вычисление значения
      pow2(state, getters) {
        return Math.pow(state.count, 2)
      }, // С параметром
      pow: (state) => (p) => {
        return Math.pow(state.count, p)
      }
    }
  })

```

Состояние приложения «state» содержит счётчик «count», имеет два идентичных метода получения значения «getters»: «pow» и «pow2», отличающиеся тем, что в первом значение параметра возведения в степень фиксировано числом «2», а во втором – передаётся в переменной «p». Функция увеличения «increment» позволяет изменить состояние приложения.

Компонент, использующий общее состояние приложения может выглядеть следующим образом.

```

<template>
  <div>
    <button @click="increment">Inc</button>
    <div>{{count}}^2 = {{power2}}</div>
    <div>{{count}}^3 = {{power3}}</div>
  </div>
</template>
<script>
import {mapState} from "vuex";
export default {
  methods: {
    increment() { // Увеличение значения
      this.$store.commit('increment')
    }
  },
  computed: {
    power2() { // Вычисляемое свойство
      return this.$store.getters.pow2
    },
    power3() {
      return this.$store.getters.pow(3)
    }, // Подключение count
    ...mapState(["count"])
  }
}

```

```
}  
</script>
```

По нажатию на кнопку «Inc» вызывается функция «increment», с помощью «this.\$store.commit()» вызывается функция общего хранилища, обращение на чтение к функциям хранилища осуществляется с использованием «this.\$store.getters», текущее значение «count» получается с помощью функции «mapState» из Vuex. Важно, что функции получения значения из хранилища находятся в разделе «computed» компонента, т.е. они вычисляются и при изменении состояния Vuex автоматически обновляются.

Следует обратить внимание, что методы «mutations» предназначены для синхронных изменений. Для асинхронных изменений (например, с обращением к web-серверу) необходимо использовать «actions» (в «store.js»).

```
mutations: { // Синхронное обновление состояния  
  increment(state, obj) {  
    state.count += obj.amount  
  }  
},  
actions: { // Асинхронное обновление состояния  
  inc(context, obj) {  
    context.commit('increment', obj)  
  }  
}
```

В компоненте следует использовать «dispatch()» для обращения к «actions».

```
methods: {  
  increment() { // Асинхронное увеличение значения  
    this.$store.dispatch('inc', {amount: 3})  
  }  
}
```

Автоматизация тестирования с помощью Selenium. Selenium (<https://www.selenium.dev/>) представляет собой фреймворк для тестирования web-приложений. Для корректной работы необходимо установить интерфейс между Selenium и соответствующим браузером: <https://www.npmjs.com/package/selenium-webdriver>. Важно, что браузер должен быть установлен глобально, а соответствующий файл драйвера помещен в папку, указанную в переменной PATH.

Пример автоматизированного теста для тестирования с использованием браузера «Google Chrome».

```

const {Builder, By, Key, until} = require('selenium-webdriver');
(async function example() {
  let driver = await new Builder().forBrowser('chrome').build();
  try {
    // Перейти по URL
    await driver.get('https://www.yandex.ru');
    // Ввести текст "cheese" и нажать на клавиатуре "Enter"
    await driver.findElement(By.name('text')).sendKeys('cheese',
Key.ENTER);
    let firstResult = await driver.wait(until.elementLocated(By.css('.main__content')), 10000);
    console.log(await firstResult.getAttribute('textContent'));
  }
  finally{
    await driver.quit();
  }
})();

```

В данном примере загружается браузер «chrome», осуществляется переход по адресу «https://www.yandex.ru», в поле с именем «text» вводится слово «cheese», ищется компонент с классом «main__content» и в консоль выводится значение «textContent».

При использовании Selenium браузер будет запущен и можно наблюдать все выполняемые операции.

Автоматизация тестирования с использованием headless-браузеров.

Существует множество headless-браузеров, рассмотрим применение Google Puppeteer (<https://developers.google.com/web/tools/puppeteer/>) для тестирования клиентской части web-приложения с использованием Jest.

```

const puppeteer = require("puppeteer")
test("Headless test", async () => {
  const browser = await puppeteer.launch({
    headless: true
  }) // если поставить false, то запустится браузер
  const page = await browser.newPage()
  await page.goto("http://localhost:3000")
  await page.waitForSelector(".app-title")
  const html = await page.$eval(".app-title", e =>
e.innerHTML)
  expect(html).toBe("Hello world")
}, 16000)

```


В данном примере загружается модуль «puppeteer», на основе которого создаётся браузер «browser», открывается новая страница «newPage()», осуществляется переход по адресу «http://localhost:3000», ожидается окончание формирования на странице компонента с классом «app-title», из которого в переменную html помещается свойство «innerHTML», затем выполняется сравнение с «Hello world».

Вопросы для контроля

1. Из чего состоит компонент Vue?
2. Как осуществляется привязка параметров и свойств компонента к элементам, атрибутам и событиям в HTML?
3. Как передаётся информация между родительским и дочерним компонентами Vue?
4. Как обеспечить маршрутизацию и обработку параметров маршрута?
5. Какие основные понятия есть в Vuex? Как реализуется единое состояние приложения?
6. Как внести изменения в состояние приложения Vuex (синхронно, асинхронно)?
7. Как создать автоматизированный тест клиентской части web-приложения?

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

- Прогрессивный JavaScript-фреймворк // URL: <https://v3.ru.vuejs.org/ru/>;
- Vue CLI // URL: <https://cli.vuejs.org/ru/>;
- Vue Router // URL: <https://next.router.vuejs.org/>;
- What is Vuex? // URL: <https://next.vuex.vuejs.org/>;
- Material Design Framework // URL: <https://vuetifyjs.com/en/>;
- Selenium automates browsers. That's it! // URL: <https://www.selenium.dev/>;
- selenium-webdriver // URL: <https://www.npmjs.com/package/selenium-webdriver>;
- Puppeteer // URL: <https://developers.google.com/web/tools/puppeteer/>.

В результате выполнения всех лабораторных работ у студента должен сформироваться навык по созданию web-приложений с использованием со-

временных web-технологий. При этом студент на практике отработает те знания, которые он получил в лекционном материале.

Следует обратить внимание на оформление кода: должны в отдельных папках храниться ресурсы, доступные по прямой ссылке (рисунки, скрипты, файлы стилей и т.п.), защищаемые паролем страницы и не защищаемые.

После успешного выполнения всех лабораторных работ преподавателю необходимо предоставить:

- 1) архив проекта со всеми исходными кодами (без папки «node_modules»),
- 2) макеты страниц, созданные в Figma, если это требовалось заданием,
- 3) «скриншоты» основных страниц web-приложения,
- 4) отчеты по всем лабораторным работам.

Содержание отчета

В отчете должны быть освещены следующие вопросы:

- 1) постановка задачи;
- 2) макеты интерфейса пользователя (созданные, например, с использованием Figma, <https://www.figma.com/>), если есть в задании;
- 3) описание тестов (модульных, интеграционных), если есть в задании;
- 4) описание наиболее важных фрагментов кода;
- 5) примеры экранных форм.

В отчете рекомендуется привести:

- 1) UML-диаграмму последовательности запросов и переходов между формами;
- 2) исходные коды лабораторной работы с комментариями.

Список рекомендуемой литературы

- Гоше Х. Д. HTML5 Для профессионалов. СПб.: Питер, 2015.
- Симпсон К. ES6 и не только. СПб.: Питер. 2017.
- Стефанов С. React.js быстрый старт. СПб.: Питер, 2017.
- Фримен А. Angular для профессионалов. СПб.: Питер, 2018.
- Шелли П. Изучаем Node. Переходим на сторону сервера. СПб.: Питер, 2017.

Оглавление

Лабораторная работа 1. Тетрис на JavaScript	3
Лабораторная работа 2. REST-приложение управления библиотекой.....	9
Лабораторная работа 3. Модуль администрирования приложения «Социальная сеть».....	21
Лабораторная работа 4. Модуль пользователя приложения «Социальная сеть»	35
Лабораторная работа 5. Модуль администрирования приложения «Биржа акций»	58
Лабораторная работа 6. Модуль приложения «Покупка и продажа акций» ..	70
Содержание отчета.....	82

Беляев Сергей Алексеевич

Web-технологии

Лабораторный практикум

Редактор М. Б. Шишкова

Подписано в печать 12.03.2019. Формат 60×84 1/16.

Бумага офсетная. Печать цифровая. Печ. л. 5,0.

Гарнитура «Times New Roman». Тираж 83 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5