МИНОБРНАУКИ РОССИИ

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

С. А. БЕЛЯЕВ, М. С. БЕЛЯЕВ

Практическое руководство по современным Web-технологиям

Учебно-методическое пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2025

УДК 004.432
ББК З 988.02–018я7
Б49

Беляев С. А., Беляев М. С.

Б49 Практическое руководство по современным Web-технологиям. Учебно-методическое пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2025. 88 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 web-технологий. Приводятся базовые элементы языков программирования и вопросы для самоконтроля.

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

УДК 004.432
ББК З 988.02–018я7

Рецензенты: АО «НИЦ СПб ЭТУ» (докт. техн. наук, доцент Д. М. Хетчиков), ООО «ПЛАЗ» (канд. техн. наук В. В. Матросов).

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия

ISBN 978-5-7629-2409-2

Учебно-методическое пособие 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 получат 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 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 Visual Studio Code – https://code.visualstudio.com/.

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/), в ОС Linux использовать Node.JS версии не ниже 22 (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.

Для операционной системы Windows необходимо сначала скачать дистрибутив OpenSSL, либо использовать Windows Subsystem for Linux (WSL).

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.

Рис. 1.1. Форма входа

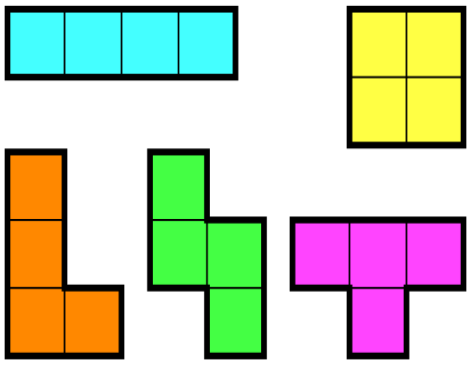

Рис. 1.2. Тетрамино

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.

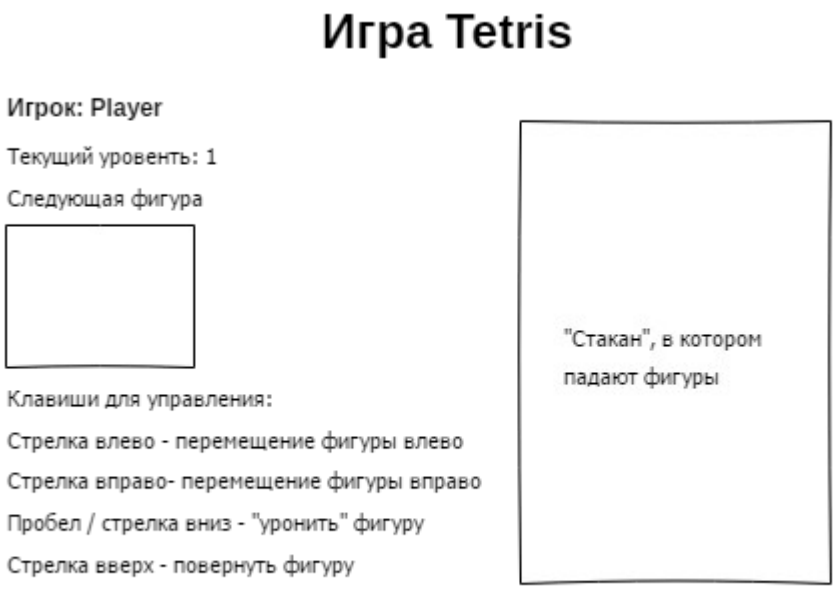

Рис. 1.3. Основная форма игры

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.setItem("tetris.username", source.value);
}
function read(source) {
    source.value = localStorage.getItem("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 (в примере приведён вызов двух команд, но хорошей практикой считается использование только одной 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.yourурl.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 is JavaScript with syntax for types // URL: https://www.typescriptlang.org/;

– OpenSSL. // 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/;

– Руководство по Canvas // URL: https://developer.mozilla.org/ru/docs/Web/API/Canvas_API/Tutorial;

– 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 и текущее состояние библиотеки хранится в JSON-файле на сервере.

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 (либо TypeScript).

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://nodejsdev.ru/guides/webdraftt/authentication/.

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, например:

```
@media screen and (max-width: 600px) {
    div { font-size: 12px; }
}
```

В приложение может быть подключен файл со стилями.

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

Оформлять CSS рекомендуется с помощью БЭМ-методологии: https://ru.bem.info/methodology/.

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 «-y» и затем внести необходимые изменения в package.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");
});
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")
});
```

В данном примере «*» означает все остальные маршруты.

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 вычислимых 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
});
router.get("/", (req, res, next)=>{
    res.end("ROOT PAGE");
});
router.get("*", (req, res)=>{
    res.status(404); // Ошибка – нет такой страницы
    res.end("Page not found");
});
module.exports = router;
```

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(); // Переход к следующему обработчику
});
```

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 заключены в кавычки.

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 JSON-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 подключенной обработки параметров JSON-body) 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([0-9]{1,})', (req, res)=>{
    const body = req.body;
    // ...
});
```

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

Удаление записи – метод DELETE:

```
router.delete('/:id([0-9]{1,})', (req, res)=>{
    const removeIndex = groups.findIndex(group => group.id ===
req.params.id);
    // ...
});
```

В функции findIndex() удобно находить элементы по их параметрам. 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.

– https://use.fontawesome.com/releases/v6.7.2/fontawesome-free-6.7.2-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-awe-
some.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/;

– БЭМ методология // https://ru.bem.info/methodology/;

– Express // URL: https://www.npmjs.com/package/express;

– Express // URL: http://expressjs.com/;

– Pug // URL: https://www.npmjs.com/package/pug;

– Pug. Getting Started // URL: https://pugjs.org/;

– EJS // URL: https://ejs.co/;

– Как построить REST API с помощью JS, Node.js и Express.js // URL: https://nodejsdev.ru/guides/rest-api-design/;

– JSON: что это за формат и как с ним работать // URL: https://skillbox.ru/media/code/json-chto-eto-za-format-i-kak-s-nim-rabotat/;

– Сетевые запросы // URL: https://learn.javascript.ru/network;

– Passport // URL: http://www.passportjs.org/;

– Аутентификация. Node.js с примерами кода // URL: https://nodejsdev.ru/guides/webdraftt/authentication/.

Лабораторная работа 3. МОДУЛЬ АДМИНИСТРИРОВАНИЯ ПРИЛОЖЕНИЯ «СОЦИАЛЬНАЯ СЕТЬ»

Цель и задачи

Целью работы является изучение возможностей применения компилятора Babel, библиотеки Bootstrap, 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 Bootstrap 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.

Bootstrap – это бесплатный и открытый фреймворк для веб-разработки, основанный на HTML, CSS и JavaScript, который предназначен для быстрой и удобной верстки адаптивных и кроссбраузерных сайтов и веб-приложений. Он предоставляет большой набор готовых компонентов, шаблонов и утилит для оформления веб-форм, кнопок, таблиц, навигационных меню и других элементов интерфейса, что позволяет существенно ускорить процесс создания современных веб-сайтов.

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 и обновляется при запросах от веб-приложения.

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 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.

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;
    &:hover {
      color: green;
    }
```

Результат преобразования в CSS

```
#header {
  background: lightblue;
}
#header a {
  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);
}

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

Результат преобразования в CSS

```
div {
  padding: 11px;
}
```

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

Разработка веб-приложений с использованием Bootstrap. Подключение Bootstrap к странице может осуществляться с использованием следующей команды в <head> html-страницы.

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.6/dist/css/boot
strap.min.css" rel="stylesheet" crossorigin="anonymous">
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.6/dist/js/bootst
rap.bundle.min.js" crossorigin="anonymous"></script>
```

В данном случае Bootstrap доступен через сеть Интернет, соответствующие файлы можно скачать с сайта https://getbootstrap.com/, поместить в статическую папку на сервере и использовать локально. Для использования на сервере установите модуль «bootstrap», соответствующие css и js-файлы будут в папках «/node_modules/bootstrap/dist/css/» и «/node_modules/bootstrap/dist/js/».

Основы использования сетки Bootstrap. Сетка Bootstrap – это 12-колоночная система, позволяющая легко создавать адаптивные макеты. Сетка состоит из трёх основных элементов:

Контейнер (.container или .container-fluid) – ограничивает ширину содержимого и центрирует его на странице.

Строка (.row) – горизонтальный блок, в котором располагаются колонки.

Колонка (.col, .col-*) – вертикальные блоки внутри строки, которые делят пространство между собой, где «*» – число от 1 до 12. Обратите внимание, что всё оформление осуществляется с помощью классов.

Пример базовой сетки:

```
<div class="container">
  <div class="row">
    <div class="col-8">Основная часть (8 колонок)</div>
    <div class="col-4">Боковая часть (4 колонки)</div>
  </div>
</div>
```

Можно комбинировать колонки разной ширины, главное – чтобы их сумма в ряду не превышала 12 (см. в примере выше 4 и 8 колонок).

Bootstrap поддерживает шесть уровней адаптивности (брейкпоинтов): xs (очень маленькие устройства, инфикс не используется), sm (≥576px), md (≥768px), lg (≥992px), xl (≥1200px), xxl (≥1400px). Можно задавать ширину колонок для разных экранов (в формате .col-*-*):

```
<div class="row">
  <div class="col-6 col-md-2 col-lg-4">Колонка</div>
  <div class="col-6 col-md-10 col-lg-8">Колонка</div>
</div>
```

Здесь на мобильных (нет инфикса) – две колонки по 12, на md – по 6, на lg – 4 и 8.

Если не указывать размер, колонки автоматически делят пространство поровну:

```
<div class="row">
  <div class="col">1</div>
  <div class="col">2</div>
</div>
```

Внутри одной колонки можно создать новую строку с колонками, что позволяет строить сложные макеты:

```
<div class="row">
  <div class="col-8">
    <div class="row">
      <div class="col-6">Вложенная колонка 1</div>
      <div class="col-6">Вложенная колонка 2</div>
    </div>
  </div>
  <div class="col-4">Боковая колонка</div>
</div>
```

Bootstrap позволяет управлять отступами между колонками с помощью классов g-*, gx-* (горизонтальные) и gy-* (вертикальные):

```
<div class="row g-3">
  <div class="col">1</div>
```

```
  <div class="col">2</div>
</div>
```

Здесь между колонками будет промежуток.

Примеры часто используемых компонентов Bootstrap.

Кнопки.

```
<button class="btn btn-primary">Основная кнопка</button>
<button class="btn btn-success">Успех</button>
<button class="btn btn-outline-danger">Контурная</button>
```

Навигация.

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Логотип</a>
  <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target="#navbarNav">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
      <li class="nav-item"><a class="nav-link active"
href="#">Главная</a></li>
      <li class="nav-item"><a class="nav-link"
href="#">Ссылка</a></li>
    </ul>
  </div>
</nav>
```

Карточки.

```
<div class="card" style="width: 18rem;">
  <img src="..." class="card-img-top" alt="...">
  <div class="card-body">
    <h5 class="card-title">Заголовок</h5>
    <p class="card-text">Описание карточки.</p>
    <a href="#" class="btn btn-primary">Подробнее</a>
  </div>
</div>
```

Оповещения.

```
<div class="alert alert-warning" role="alert">
  Это предупреждение!
</div>
```

Формы.

```
<form>
  <div class="mb-3">
    <label for="exampleInputEmail1" class="form-label">Email
адрес</label>
    <input type="email" class="form-control" id="exampleIn-
putEmail1">
  </div>
  <button type="submit" class="btn btn-primary">Отправить</but-
ton>
</form>
```

Работа с событиями и динамическими компонентами. Многие компоненты Bootstrap (например, модальные окна, аккордеоны, выпадающие меню) генерируют собственные события, которые позволяют "подключаться" к их жизненному циклу. Обычно события имеют вид «имя_действия.bs.компонент». Для управления ими можно использовать data-атрибуты или напрямую обращаться к API через JS.

Пример событий для модального окна:

– show.bs.modal – вызывается немедленно при показе модального окна;

– shown.bs.modal – вызывается, когда модальное окно стало видимым (после завершения анимации);

– hide.bs.modal – вызывается немедленно при скрытии модального окна;

– hidden.bs.modal – вызывается, когда окно полностью скрыто.

Пример: обработка события открытия модального окна.

```
var myModal = document.getElementById('exampleModal');
myModal.addEventListener('show.bs.modal', function (event) {
  // Код при открытии модального окна
});
```

Многие динамические компоненты Bootstrap можно активировать и настраивать без кода – с помощью HTML-атрибутов:

```
<button type="button" class="btn btn-primary" data-bs-
toggle="modal" data-bs-target="#exampleModal">
  Открыть модальное окно
</button>
```

Здесь data-bs-toggle="modal" и data-bs-target="#exampleModal" автоматически делают кнопку триггером для открытия модального окна.

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 пример 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: {
        rules: [
            {
                test: /\.css$/,
                exclude: /node_modules/,
                use:['style-loader', 'css-loader']
            },
            {
                test: /\.less$/,
                exclude: /node_modules/,
                use:['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 «use» перечислены загрузчики, они 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

```
let data = 5
if(data > 2)
    console.log(`two ${data}`)
```

Преобразованный JS-файл

```
var data = 5;
if(data > 2)
    console.log("two " + data);
```

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 Bootstrap для оформления компонентов. В чем отличия данного подхода от традиционного CSS?
3. Для чего предназначен babel? Какие модули используются для его корректной работы? Приведите примеры команд запуска и результатов его работы.
4. Для чего предназначен GULP? Приведите примеры с использованием less, sass, babel, pug, ejs, минификации.
5. Какие команды следует использовать для регистрации нового модуля npm? Как воспользоваться созданным модулем? Как обновить версию модуля?
6. 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/;

– Build fast, responsive sites with Bootstrap // URL: https://getbootstrap.com/;

– Начать работу с Bootstrap // URL: https://getbootstrap.ru/docs/5.3/getting-started/introduction/;

– {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 или на другой адрес http://127.0.0.1:8080.

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 Signals и SignalStore Angular (https://ngrx.io/guide/signals/signal-store).

Преимуществом будет использование звукового сопровождения событий: получений 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». Для создания приложения с использованием NgModules (именно этот режим описан ниже), при создании приложения выберите опцию "Enable standalone components?" – No. Основной файл app.module.ts будет содержать декоратор @NgModule с настройками приложения, а все компоненты и сервисы будут регистрироваться через этот модуль (см. ниже). Standalone-компоненты – это новый подход в Angular, при котором компоненты, директивы и каналы могут использоваться без NgModule. Подробное описание можно найти на сайте https://v17.angular.io/guide/standalone-components.

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.

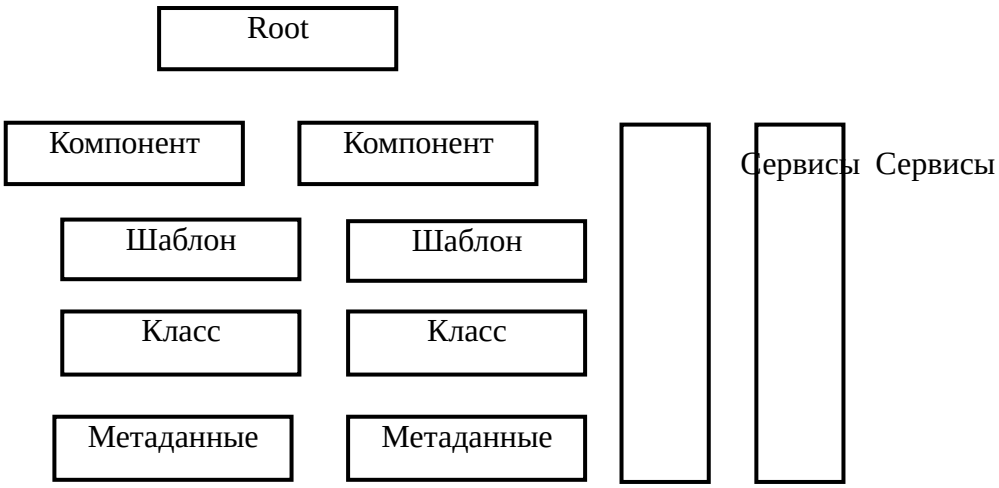

Рис. 5.1. Обобщенная архитектура приложения на Angular

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» рассматривать 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>
  <img width="300" alt="Angular Logo" src="data:...">
</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: `<p>Текст: {{mytext}}</p>
  <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: `<h3>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».

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 параметра 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.

Блоки управления потоком. Блоки управления потоком «@if/@else» позволяет проверять условие на истинность и скрывать или отображать фрагменты HTML:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    @if (myCondition) {
      <p>Истина</p>
    } @else {
      <p>Ложь</p>
    }
    <button (click)="toggle()">Переключить</button>`
})
```

```
export class AppComponent {
  myCondition: boolean = true;
  toggle() {
    this.myCondition = !this.myCondition;
  }
}
```

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

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <ul>
      @for (let item of myList; track item) {
        <li>{{ item }}</li>
      } @empty {
        <li>Список пуст</li>
      }
    </ul>`
})
export class AppComponent {
  myList = ["один", "два", "три"];
}
```

Блок @for позволяет перебирать элементы массива. Если список пуст, срабатывает блок @empty. Можно явно указать стратегию отслеживания изменений через track: в данном примере отслеживается item целиком, обычно указывают идентификатор (например, item.id), чтобы не изменялся DOM всего списка при изменении одного элемента.

По аналогии можно использовать блок @switch, который реализует множественный выбор по значению переменной. Внутри используются блоки @case и, при необходимости, @default.

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>
      @for (let todo of mydata; track todo.title) {
        <li>{{todo.title}}</li>
      }
    </ul>`,
  providers: [ TodoService ]
})
export class AppComponent implements OnInit {
  mydata: ToDo[];
  title: string = '';
  constructor(private todoService: TodoService) { }
  ngOnInit(): void {
    this.mydata = this.todoService.getData();
  }
  add(title: string) {
    this.todoService.addData(title);
    this.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 блока

«@for» 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>
      @for (let todo of mydata; track todo.title) {
        <li>{{todo.title}}</li>
      }
    </ul>`
})
export class ChildComponent implements OnInit {
  mydata: ToDo[];
  title: string = '';
  constructor(private todoService: TodoService) { }
  ngOnInit(): void { // При инициализации компонента
    this.mydata = this.todoService.getData();
  }
  add(title: string) { // Добавление данных
    this.todoService.addData(title);
    this.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.

Signals и SignalStore. Signals – новый реактивный примитив в Angular, предназначенный для управления состоянием компонентов и приложений. Он является обёрткой вокруг значения (например, числа, строки или объекта), которая автоматически уведомляет Angular и все заинтересованные части приложения при изменении этого значения. Для чтения значения используется вызов функции signal (например, count()), для изменения – методы .set() или .update(). Signals позволяют отказаться от ручного управления подписками и упрощают реактивное программирование, делая его более предсказуемым и производительным.

SignalStore – это современное решение для управления состоянием приложения, построенное на базе Signals и появившееся в Angular 17+. SignalStore реализует концепцию «хранилища» (store) – централизованного хранилища состояния. Сигнал-стор позволяет определять состояние, методы для его изменения и вычисляемые значения (computed) декларативно и компактно. Для создания SignalStore используется функция signalStore с набором функций-конструкторов (например, withState, withMethods), а все части состояния автоматически становятся сигналами.

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: To-
doService){}
  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)
        }
      )
  }
}
```

Для корректного исполнения приведенного примера необходимо подключить модуль «HttpClientModule» (‘@angular/common/http’) в разделе imports в «AppModule».

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 npm 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-home",
  template: `<h2>Домашняя страница</h2>`
})
export class HomeComponent { }
```

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

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

Компонент «src/app/notfound.ts».

```
import {Component} from "@angular/core"
@Component({
  selector: "app-not-found",
  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. В качестве параметра в «resolve» передается результат выполнения функции.

2.
```
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» обрабатывает результат исполнения обещания. 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, кроме самого отправителя.

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 исполняется 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. Для чего в Angular используется SignalStore?

7. Приведите примеры использования обещаний и обработки ошибок.

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

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

10. Приведите пример Jest-теста.

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

– Angular. One framework. Mobile & desktop // URL: https://angular.io/

– SignalStore // URL: https://ngrx.io/guide/signals/signal-store

– StackBlitz. Online VS Code IDE for Angular & React // URL: https://stackblitz.com/

– Руководство по Angular 20 // URL: https://metanit.com/web/angular2/

– Migrate an existing Angular project to standalone // URL: https://angular.dev/reference/migrations/standalone

– Promise // URL: https://learn.javascript.ru/promise

– Rollbar. Catch errors before your users do // URL: https://rollbar.com/

– Socket.IO // URL: https://socket.io/

– Jest // URL: https://jestjs.io/

– 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 подхода 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: настройка даты начала торгов, скорости смены дат в секундах при имитации торгов (для примера данных из п.4 – в первую секунду стоимости от 03.11.21, во вторую секунду от 04.11.2025 и т.д.). 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-сокетам с учётом заданных настроек биржи, здесь же должна 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 рекомендуется использовать инструмент Vite, который пришёл на смену устаревшему create-react-app и обеспечивает быструю сборку, мгновенный запуск и современные возможности разработки.

Пошаговая инструкция:

1. Выполните в терминале команду:

```
npm create vite@latest my-react-app -- --template react
```

Здесь my-react-app – это название создаваемого проекта.

2. Перейдите в папку проекта и установите зависимости:

```
cd my-react-app
npm install
```

3. Запустите приложение:

```
npm run dev
```

После запуска приложение будет доступно по адресу (порт может отличаться в зависимости от настроек) `http://localhost:5173/`

4. Главный компонент приложения находится в файле src/App.jsx (или src/App.tsx, если выбран шаблон с TypeScript).

Компоненты React. Компоненты React – это строительные блоки любого React-приложения. Каждый компонент инкапсулирует свой функционал, внешний вид и может быть повторно использован в разных частях интерфейса.

Компоненты бывают двух видов: функциональные и классовые. Классовые – классы, расширяющие React.Component. Функциональные – обычные функции, принимающие свойства и возвращающие JSX. JSX (https://github.com/facebook/jsx) – это расширение синтаксиса JavaScript, позволяющее писать HTML-подобную разметку прямо в JavaScript-коде и используемое в React для описания структуры пользовательского интерфейса.

Рассмотрим пример функционального компонента:

```
function Greeting({ name }) {
  return <h1>Привет, { name }!</h1>;
}
export default Greeting;
```

Такой компонент принимает свойства и возвращает JSX.

Пример функционального компонента с состоянием и хуком «useState»:

```
import { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h1>Счёт: {count}</h1>
```

```
      <button onClick={() => setCount(count + 1)}>Увеличить</
button>
    </div>
  );
}
```

useState – хук для состояния, который позволяет управлять состоянием прямо внутри функционального компонента.

Пример передачи свойств (props):

```
function Welcome({ name }) {
  return <h2>Добро пожаловать, {name}!</h2>;
}
function App() {
  return <Welcome name="Анна" />;
}
```

Свойства позволяют делать компоненты динамичными и повторно используемыми. Компоненты можно вкладывать друг в друга, строя сложные интерфейсы из простых частей:

```
function Sum({ x = 2 }) {
  return <h2>{x} + {x} = {x + x}</h2>;
}
function App() {
  return (
    <div>
      <Sum x={3} />
      <Sum />
    </div>
  );
}
```

Если свойство x не передано, используется значение по умолчанию (2).

Пример с классовыми компонентами (устаревающий подход).

```
import React, { Component } from 'react';
class OldCounter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  render() {
    return <h1>Счёт: {this.state.count}</h1>;
  }
}
```

Классы всё ещё поддерживаются, но в новых проектах используются редко.

В JSX можно использовать любые переменные JavaScript, обернув их в фигурные скобки: `<h1>{hello} React!</h1>`

Компонент всегда должен возвращать один корневой элемент (например, или <>...</>).

Лучшие практики:

– используйте только функциональные компоненты и хуки (useState, useEffect, useRef и др.);

– делайте компоненты маленькими и специализированными: один компонент – одна задача;

– передавайте данные через свойства (props), не используйте глобальные переменные;

– организуйте компоненты по функциональным областям;

– для сложного состояния используйте современные менеджеры состояний (Redux Toolkit, Zustand и др.) при необходимости.

Состояние компонента и обработка событий. Состояние (state) – это объект, который хранит данные, влияющие на отображение и поведение компонента. Когда состояние изменяется, React автоматически перерисовывает компонент, чтобы отразить новые данные.

В функциональных компонентах состояние реализуется с помощью хука useState:

```
import { useState } from "react";
function ToggleButton() {
  // начальное состояние – false
  const [isOn, setIsOn] = useState(false);
  function handleClick() {
    setIsOn(prev => !prev); // переключение состояния
  }
  return (
    <button
      className={isOn ? "on" : "off"}
      onClick={handleClick} // обработка клика мышью
    > {isOn ? "Включено" : "Выключено"}
    </button>
  );
}
```

Хук useState возвращает массив из двух элементов: текущего значения состояния и функции для его обновления. При изменении состояния через функцию (setIsOn) компонент автоматически перерисовывается. В данном примере предполагается, что в CSS объявлены классы «on» и «off»:

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

Обработка событий в React аналогична работе с DOM, но названия событий пишутся в camelCase (например, «onClick», «onChange»), а обработчики передаются как функции:

```
<input
  type="text"
  value={text}
  onChange={e => setText(e.target.value)}
/>
```

Пример формы с состоянием:

```
import { useState } from "react";
function FormExample() {
  const [text, setText] = useState("");
  const [submitted, setSubmitted] = useState("");
  function handleChange(e) {
    setText(e.target.value);
  }
  function handleSubmit(e) {
    e.preventDefault();
    setSubmitted(text);
  }
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={text} onChange={handleChange}/>
      <button type="submit">Отправить</button>
      <p>Отправлено: "{submitted}"</p>
    </form>
  );
}
```

Состояние «text» хранит значение поля ввода, а «submitted» – отправленный текст. Функция handleChange обновляет состояние при каждом вводе, а handleSubmit – при отправке формы.

В классовых компонентах состояние хранится в объекте «this.state», а обновляется методом this.setState(). Такой подход не рекомендуется для новых проектов:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOn: false };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState(prevState => ({ isOn: !prevState.isOn }));
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isOn ? "Включено" : "Выключено"}
```

```
      </button>
    );
  }
}
```

Для обновления состояния всегда используйте this.setState(), а не прямое присваивание. Методы жизненного цикла (например, constructor, componentDidMount, componentWillUnmount) используются только в классовых компонентах.

Важно помнить:

– состояние компонента должно изменяться только через специальные функции (setState или функцию из useState), прямое изменение не вызовет перерисовку компонента;

– обработчики событий получают объект события (например, event), через который можно получить данные о действии пользователя;

– функциональные компоненты с хуками – стандарт React с 2019 года.

Маршрутизация в React. Для организации маршрутизации в React-приложениях используется библиотека react-router-dom (https://reactrouter.com/start/declarative/routing). Установка для 6 версии:

```
npm install react-router-dom@6
```

Базовый пример маршрутизации:

```
import { BrowserRouter, Routes, Route, Link } from "react-
router-dom";
function App() {
  return (
    <BrowserRouter>
      <nav>
        <ul>
          <li><Link to="/">Главная</Link></li>
          <li><Link to="/about">О сайте</Link></li>
        </ul>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="*" element={<NoMatch />} />
      </Routes>
    </BrowserRouter>
  );
}
function Home() {
  return <h3>Главная страница</h3>;
}
function About() {
  return <h3>О сайте</h3>;
}
```

```
function NoMatch() {
  return <h3>Страница не найдена (404)</h3>;
}
```

Компонент Routes заменяет устаревший компонент Switch. Для каждого маршрута используется атрибут element с JSX-компонентом. Компонент Link обеспечивает переходы без перезагрузки страницы.

Для работы с динамическими параметрами маршрута (например, /user/:id) и строкой запроса (?name=Ivan) используются хуки:

```
import { useParams, useSearchParams } from "react-router-dom";
function UserPage() {
  const { id } = useParams(); // динамический параметр маршрута
  const [searchParams] = useSearchParams();
  const name = searchParams.get("name");// name в строке запроса
  return (
    <div>
      <b>id:</b> {id} <br />
      <b>name:</b> {name}
    </div>
  );
}
```

Пример использования в маршрутах:

```
<Routes>
  <Route path="/user/:id" element={<UserPage />} />
</Routes>
```

Хук useParams возвращает объект с параметрами из пути.

Хук useSearchParams позволяет читать параметры запроса (search params).

Особенности React Router:

– поддерживает вложенные маршруты, позволяя строить сложные структуры страниц;

– поддерживает относительные ссылки, упрощает навигацию внутри вложенных маршрутов;

– только компонент Routes содержит список маршрутов;

– маршруты по умолчанию сопоставляются по точному совпадению пути.

Управление состоянием приложения. Управление состоянием – это организация и контроль данных, которые определяют, как приложение выглядит и работает в каждый момент времени. В небольших React-приложениях часто достаточно локального состояния компонентов (через хуки useState и useReducer). Однако по мере роста приложения возникает необходимость в централизованном управлении состоянием, чтобы избежать "prop drilling" и упростить обмен данными между разными частями интерфейса.

Возможные инструменты для глобального управления состоянием:

– Redux Toolkit – официальный и оптимизированный инструмент для работы с Redux, который сводит к минимуму шаблонный код и упрощает работу с асинхронными операциями;

– Zustand, Jotai, Recoil, MobX – современные альтернативы, подходящие для разных сценариев и масштабов приложений.

Преимущества Redux Toolkit:

– централизованное хранилище состояния (store) – «единственный источник истины» для всего приложения;

– предсказуемый и прозрачный поток данных: изменения состояния происходят только через «actions» и «reducers»;

– встроенная поддержка иммутабельности через Immer;

– упрощённая работа с асинхронными запросами (например, через createAsyncThunk);

– меньше шаблонного кода.

Ключевые понятия Redux Toolkit:

– Store – централизованное хранилище состояния приложения;

– Action – обычный объект, описывающий намерение изменить состояние (например, {type: 'increment'});

– Reducer – чистая функция, определяющая, как состояние изменяется в ответ на действие (часто называют «редьюсер»);

– Slice – единица логики Redux Toolkit: объединяет редьюсер, действия и начальное состояние;

– Dispatch – метод для отправки действий в хранилище;

– Selector – функция для извлечения нужных данных из состояния.

Пример использования Redux Toolkit

```
import { createSlice, configureStore } from '@reduxjs/toolkit';
// Создание slice — набора логики для отдельной части состояния
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: state => { state.value += 1 },
```

```
    decrement: state => { state.value -= 1 },
    incrementByAmount: (state, action) =>
        { state.value += action.payload }
  }
});
export const { increment, decrement, incrementByAmount } =
        counterSlice.actions;
// Создание единого хранилища приложения
const store = configureStore({
  reducer: { counter: counterSlice.reducer }
});
// Использование: store.dispatch(increment());
```

Все изменения состояния происходят только через действия и чистые функции-редьюсеры. Для интеграции с React используется библиотека react-redux и хуки useSelector, useDispatch.

Принципы Redux:

– всё состояние приложения хранится в одном объекте store;

– состояние можно изменять только через специальные действия (actions);

– редьюсеры не имеют побочных эффектов, что делает логику предсказуемой и удобной для тестирования.

Глобальное управление состоянием необходимо использовать при обмене данными между разными, не связанными напрямую компонентами, для сложных или масштабируемых приложений с большим количеством взаимодействий и асинхронных операций, для реализации undo/redo, сохранения состояния между сессиями, глубокого журналирования и отладки.

Создание графиков с использованием Chart.js. Chart.js – это библиотека для построения интерактивных, анимированных и адаптивных графиков на основе canvas. Она поддерживает множество типов графиков (линейные, столбчатые, круговые и др.) и легко интегрируется с React через обёртку react-chartjs-2.

Установите необходимые пакеты:

```
npm install chart.js react-chartjs-2
```

Импортируйте и зарегистрируйте компоненты Chart.js:

```
import { Chart as ChartJS, CategoryScale, LinearScale, PointEle-
ment, LineElement, Title, Tooltip, Legend } from 'chart.js';
import { Line } from 'react-chartjs-2';
```

```
ChartJS.register( CategoryScale, LinearScale, PointElement, Li-
neElement, Title, Tooltip, Legend );
```

Определите данные и параметры графика:

```
const data = {
  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'
  ],
  datasets: [
    {
      label: 'AAPL',
      data: [ 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 ],
      borderColor: 'rgb(255, 99, 132)',
      backgroundColor: 'rgba(255, 99, 132, 0.2)',
      tension: 0.2
    }
  ]
};
const options = {
  responsive: true,
  plugins: {
    legend: { position: 'top' },
    title: { display: true, text: 'Курс AAPL за месяц' }
  },
  scales: {
    y: { beginAtZero: false }
  }
};
```

Создайте компонент графика:

```
import React from 'react';
const AppleChart = () => (
  <div style={{ width: '100%', maxWidth: 600 }}>
    <Line data={data} options={options} />
  </div>
);
export default AppleChart;
```

Пример использования в приложении

```
import React from 'react';
import AppleChart from './AppleChart';
function App() {
  return (
```

```
    <div>
      <h2>Динамика акций Apple</h2>
      <AppleChart />
    </div>
  );
}
export default App;
```

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 Toolkit? 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/;

– Routing // URL: https://reactrouter.com/start/declarative/routing;

– Draft: JSX Specification // URL: https://facebook.github.io/jsx/;

– Начало работы с React // URL: https://developer.mozilla.org/ru/docs/Learn_web_development/Core/Frameworks_libraries/React_getting_started;

– Руководство по React // URL: https://metanit.com/web/react/;

– Online VS Code IDE for Angular & React // URL: https://stackblitz.com/;

– Redux Toolkit // URL: https://redux-toolkit.js.org;

– 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) и XPath (https://testengineer.ru/xpath-quick-guide/).

Описание ключевых методов при выполнении работы

Создание каркаса web-приложения с использованием Vue.

Для создания нового Vue-приложения выполните команду:

```
npm create vue@latest
```

или, если используете Yarn или pnpm:

```
yarn create vue
pnpm create vue
```

В процессе создания вы сможете выбрать необходимые опции (TypeScript, роутер, Pinia и др.).

Для запуска приложения после установки зависимостей перейдите в папку проекта и выполните команду:

```
npm run dev
```

По умолчанию приложение будет доступно по адресу http://localhost:5173/.

Главный файл приложения – src/main.js (или src/main.ts при выборе TypeScript). Простейший пример его содержимого:

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

Компонент App.vue будет отображён в элементе с идентификатором app в файле index.html.

Компоненты Vue. Компонент Vue – это самостоятельная часть интерфейса (фрагмент кода), которую можно повторно использовать и комбинировать для построения сложных web-приложений.

Современный способ создания компонента использует синтаксис <script setup>. Опишем простейший компонент (src/App.vue):

```
<template>
  <div>
    <h3>{{ msg }}</h3>
  </div>
</template>
<script setup>
import { ref } from 'vue'
const msg = ref('Привет Vue')
</script>
<style scoped>
h3 { color: green; }
</style>
```

В этом примере на странице отобразится приветствие «Привет Vue».

Компонент состоит из трёх частей:

1) template – разметка (HTML-шаблон);

2) script setup – логика (JavaScript-код, современный стандарт Vue 3);

3) style – стили (CSS).

В шаблоне значения переменных отображаются с помощью двойных фигурных скобок: {{ msg }}. Внутри скобок можно использовать любое корректное JavaScript-выражение, возвращающее значение.

Компоненты могут:

– хранить свои данные (ref, reactive);

– получать параметры от родителя (props);

– вычислять значения (computed);

– реагировать на события (methods, обработчики);

– использовать жизненный цикл (onMounted, onUpdated и др.).

Рассмотрим пример передачи данных и событий между компонентами. Родительский компонент (App.vue):

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

Дочерний компонент (components/HelloWorld.vue):

```
<template>
  <div>
    <h3 :title="text">{{ msg }}</h3>
    <button @click="reverse">Перевернуть</button>
  </div>
</template>
<script setup>
import { ref } from 'vue'
// Получение props через defineProps
const props = defineProps({
  msg: String
})
const text = ref('Тестовый текст')
function reverse() {
  text.value = text.value.split('').reverse().join('')
}
</script>
```

Здесь:

1) props позволяют передавать данные от родителя к потомку;

2) @click="reverse" – обработка событий;

3) :title="text" – привязка значения к атрибуту (альтернатива v-bind:title).

Односторонняя привязка осуществляется с помощью «v-bind:» или двоеточия «:». Двусторонняя привязка осуществляется с помощью «v-model».

Пример двусторонней привязки:

```
<template>
```

```
  <input v-model="question" />
</template>
<script setup>
import { ref } from 'vue'
const question = ref('')
</script>
```

Предусмотрена передача событий от дочернего к родительскому компоненту. Родительский компонент (App.vue):

```
<template>
  <CounterButton btn-label="Сч. A" :idx="0" @inc-value="incre-
ment" />
  <CounterButton btn-label="Сч. B" :idx="1" @inc-value="incre-
ment" />
  <div>Счётчики: A={{ counter[0] }}, B={{ counter[1] }}</div>
</template>
<script setup>
import { ref } from 'vue'
import CounterButton from '@/components/CounterButton.vue'
const counter = ref([0, 0])
function increment(index) {
  counter.value[index]++
}
</script>
```

Дочерний компонент (CounterButton.vue):

```
<template>
  <button @click="$emit('inc-value', idx)">
    {{ btnLabel }}
  </button>
</template>
<script setup>
const props = defineProps({
  btnLabel: String,
  idx: Number
})
</script>
```

События генерируются через $emit('inc-value', idx) и обрабатываются родителем. Имена свойств (props) в шаблоне используются через camelCase (btnLabel), а при передаче – через kebab-case (btn-label).

Маршрутизация в Vue. Для организации навигации между страницами в приложении на Vue используется дополнительный модуль vue-router.

Если проект создавался с помощью npm create vue@latest и была выбрана опция «Добавить роутер», всё уже настроено. Если нет, то его необходимо установить вручную:

```
npm install vue-router
```

В файле src/main.js (или src/main.ts) необходимо подключить маршрутизатор:

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

Пример настройки маршрутов (src/router/index.js или src/router.js):

```
import { createRouter, createWebHistory }  from 'vue-router'
import Home from '../views/Home.vue'
const routes = [
  { path: '/', name: 'Home', component: Home },
  {
    path: '/about',
    name: 'About',
    // Ленивая (lazy) подгрузка компонента
    component: () => import('../views/About.vue')
  }
]
const router = createRouter({
  history: createWebHistory(),
  // Используйте createWebHistory для "красивых" URL
  routes
})
export default router
```

Используйте createWebHistory() для обычных URL (например, «/about»), а createWebHashHistory() – для хэш-маршрутизации (например, «/#/about»), если требуется поддержка старых браузеров или специфических серверных настроек.

Маршруты можно использовать и в шаблоне. Например, в App.vue:

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

Здесь:

1) <router-link> – создаёт ссылку для перехода по маршруту;

2) <router-view> – место, где будет отображаться компонент, соответствующий текущему маршруту.

Чтобы передавать параметры через URL, добавьте их в путь маршрута:

```
{
  path: '/about/:id',
  name: 'About',
  component: () => import('../views/About.vue')
}
```

Для получения параметров в компоненте в <script setup> напишите, например:

```
<script setup>
import { useRoute } from 'vue-router'
const route = useRoute()
</script>
<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:

1) route.params – параметры пути (/about/:id);

2) route.query – параметры запроса (?page=1);

3) route.path – текущий путь.

Управление состоянием приложения. Vuex (https://next.vuex.vuejs.org/) долгое время был стандартом для управления состоянием во Vue-приложениях. Однако с выходом Vue3 официальная команда Vue рекомендует использовать Pinia (https://pinia.vuejs.org/). Vuex остаётся поддерживаемым, но для новых проектов рекомендуется Pinia.

Установка Pinia:

```
npm install pinia
```

Подключение Pinia в приложение (src/main.js):

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

Пример Pinia-хранилища (src/stores/counter.js):

```
import { defineStore } from 'pinia'
export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0
  }),
  getters: {
    pow2: (state) => state.count ** 2,
    pow: (state) => (p) => state.count ** p
  },
  actions: {
    increment() {
```

```
      this.count++
    },
    incrementBy(amount) {
      this.count += amount
    }
  }
})
```

Использование Pinia в компоненте (<script setup>):

```
<script setup>
import { useCounterStore } from '@/stores/counter'
const counter = useCounterStore()
</script>
<template>
  <div>
    <button @click="counter.increment">Inc</button>
    <button @click="counter.incrementBy(3)">+3</button>
    <div>{{ counter.count }}^2 = {{ counter.pow2 }}</div>
    <div>{{ counter.count }}^3 = {{ counter.pow(3) }}</div>
  </div>
</template>
```

Состояние (state) и методы (actions, getters) доступны напрямую через объект store. Pinia поддерживает реактивность и автоматические обновления интерфейса.

Если требуется Vuex, то в нём:

1) state – глобальное состояние приложения;

2) mutations – синхронные методы изменения состояния;

3) actions – асинхронные методы, которые могут вызывать mutations;

4) getters – вычисляемые свойства на основе состояния.

Пример использования Vuex (src/store/index.js):

```
import { createStore } from 'vuex'
export default createStore({
  state: () => ({ count: 0 }),
  mutations: { // Синхронное изменение
    increment(state) { state.count++ }
  },
  actions: { // Асинхронное изменение
    incrementAsync({ commit }) {
      setTimeout(() => commit('increment'), 1000)
    }
  },
  getters: {
    double: (state) => state.count * 2
  }
```

```
})
```

Использование Vuex в компоненте:

```
<template>
  <div>
    <button @click="increment">+1</button>
    <button @click="incrementAsync">+1 (через 1 сек)</button>
    <div>count = {{ count }}</div>
    <div>double = {{ double }}</div>
  </div>
</template>
<script>
export default {
  computed: {
    count() { // Получаем значение count из state
      return this.$store.state.count
    },
    double() { // Получаем значение double из getters
      return this.$store.getters.double
    }
  },
  methods: {
    increment() { // Синхронное увеличение
      this.$store.commit('increment')
    },
    incrementAsync() { // Асинхронное увеличение
      this.$store.dispatch('incrementAsync')
    }
  }
}
</script>
```

Автоматизация тестирования с помощью Selenium. Selenium (https://www.selenium.dev/) – это фреймворк для автоматизации браузеров, который широко применяется для тестирования web-приложений. С помощью Selenium можно создавать автоматизированные сценарии, которые выполняют действия в браузере так же, как это делает пользователь.

Основные компоненты Selenium:

– Selenium WebDriver – основной инструмент для написания автоматизированных тестов, поддерживает работу с разными браузерами (Chrome, Firefox, Edge и др.);

– Selenium IDE – расширение для браузеров (Chrome, Firefox, Edge), позволяющее быстро записывать и воспроизводить тестовые сценарии без программирования;

– Selenium Grid – инструмент для параллельного запуска тестов в разных браузерах и на разных компьютерах, что удобно для масштабирования тестирования.

Для работы с Selenium WebDriver необходимо:

1) Установить пакет selenium-webdriver через npm.

```
npm install selenium-webdriver
```

2) Убедиться, что нужный браузер установлен на компьютере.

3) Скачать соответствующий драйвер браузера по ссылке https://www.npmjs.com/package/selenium-webdriver (например, ChromeDriver для Google Chrome) и добавить его в переменную окружения PATH, чтобы WebDriver мог управлять браузером.

Пример автоматизированного теста (Node.js + Google Chrome)

```
const { Builder, By, Key, until } = require('selenium-web-
driver');
 (async function example() {
  let driver = await new Builder().forBrowser('chrome').build();
  try {
    // Открыть страницу Яндекса
    await driver.get('https://www.yandex.ru');
    // Ввести "cheese" в поисковое поле и нажать Enter
    await driver.findElement(By.name('text')).sendKeys('cheese',
Key.ENTER);
    // Дождаться появления результатов поиска
    let firstResult = await
driver.wait(until.elementLocated(By.css('.main__content')),
10000);
    // Вывести текстовое содержимое найденного элемента
    console.log(await firstResult.getAttribute('textContent'));
  } finally {
    await driver.quit();
  }
})();
```

Пояснения:

– в примере используется браузер Google Chrome;

– сценарий автоматически открывает страницу, выполняет поиск и выводит результат в консоль;

– все действия выполняются в реальном браузере, и их можно наблюдать в процессе выполнения.

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

Использование XPath в Selenium.

XPath – язык для навигации по элементам HTML-страницы. В Selenium его часто используют для поиска элементов, особенно когда другие локаторы (id, class, name) не подходят (https://habr.com/ru/companies/otus/articles/533354/). Рассмотрим на примере.

```
const { Builder, By } = require('selenium-webdriver');
async function example() {
  let driver = await new Builder().forBrowser('chrome').build();
  try {
    await driver.get('https://example.com');
    // Один элемент
    let element =
await driver.findElement(By.xpath("//input[@type='text']"));
    // Несколько элементов
    let elements = await driver.findElements(By.xpath("//a"));
    // ... здесь код для работы с элементами
  } finally {
    await driver.quit();
  }
}
example();
```

Здесь: find_element(By.XPATH, ...) находит первый подходящий элемент, find_elements(By.XPATH, ...) возвращает список всех подходящих элементов.

Что можно использовать для поиска с помощью XPath:

– тег: `//input`;

– атрибут тега: `//button[@id='submit']`;

– текст: `//a[text()='Подробнее']`;

– частичное совпадение текста: `//a[contains(text(), 'Подробнее')]`;

– иерархию элементов: `//div[@class='header']/span`;

– относительной и абсолютный пути (абсолютный путь использовать не рекомендуется).

Преимущества XPath:

– поддерживает сложную логику выбора элементов (по тексту, атрибутам, положению, отношениям между элементами);

– подходит для динамических и «сложных» DOM-структур, когда другие локаторы ненадёжны.

Недостатки XPath:

– XPath-выражения часто получаются длинными и менее читаемыми, чем CSS-селекторы;

– абсолютные пути могут становиться не корректными при изменениях структуры страницы.

Автоматизация тестирования с использованием headless-браузеров. Headless-браузеры позволяют запускать автоматизированные тесты без отображения графического интерфейса браузера, что ускоряет выполнение тестов и снижает нагрузку на систему. Один из самых популярных инструментов для этих целей – Google Puppeteer.

Преимущества headless-режима:

– отсутствие интерфейса снижает потребление ресурсов и ускоряет тесты;

– headless-режим подходит для автоматизации тестирования на сервере и в облаке;

– для отладки можно включить отображение окна браузера (headless: false).

Для интеграции с Jest рекомендуется использовать пакет jest-puppeteer, который упрощает настройку окружения и предоставляет доступ к глобальным объектам browser и page. Установка:

```
npm install --save-dev puppeteer jest jest-puppeteer
```

В package.json или jest.config.js:

```
{
  "preset": "jest-puppeteer"
}
```

Пример теста (example.test.js):

```
describe("Headless test", () => {
  it("should display 'Hello world' in .app-title", async () => {
    await page.goto("http://localhost:3000")
    await page.waitForSelector(".app-title", { visible: true })
    const html = await page.$eval(".app-title", e => e.inner-
HTML)
    expect(html).toBe("Hello world")
  }, 16000)
```

```
})
```

Здесь page – глобальный объект, предоставляемый jest-puppeteer. Тест открывает страницу, ждёт появления элемента .app-title и проверяет его содержимое.

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 Pinia и Vuex? Как реализуется единое состояние приложения?
6. Как внести изменения в состояние приложения Pinia и 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://ru.vuejs.org/;

– Vue CLI // URL: https://cli.vuejs.org/ru/;

– Vue Router // URL: https://next.router.vuejs.org/;

– What is Vuex? // URL: https://vuex.vuejs.org/;

– Pinia. The intuitive store for Vue.js // URL: https://pinia.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;

– XPath — быстрый гайд // URL: https://testengineer.ru/xpath-quick-guide/;

– Функции XPath для динамических XPath в Selenium // URL: https://habr.com/ru/companies/otus/articles/533354/;

– 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 современных 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) краткий скринкаст работы приложения;

5) 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

Дакетт Джон. HTML и CSS. Разработка и дизайн веб-сайтов. Эксмо, 2024.

Кириченко А.В., Хрусталев А.А. HTML5 + CSS3. Основы современного WEB-дизайна, 2-е изд. Наука и техника, 2019.

Хавербеке Марейн. Выразительный JavaScript. Современное веб-программирование. Питер (Айлиб), 2020.

Попова Юлия. Node.js: разработка приложений в микросервисной архитектуре с нуля. БХВ, 2024.

Файн Я., Моисеев А. Angular и TypeScript. Сайтостроение для профессионалов. Питер (Айлиб), 2018.

Мардан Азат, Барклунд Мортен. React быстро. 2-е межд. изд. Питер, 2024.

Pablo David Garaguso. Vue.js 3 Design Patterns and Best Practices. Packt Publishing, 2023.

Оглавление

Беляев Сергей Алексеевич, Беляев Михаил Сергеевич

Практическое руководство по современным Web-технологиям

Учебно-методическое пособие

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