УДК 004.432
ББК З 988.02–018я7
Б49

Беляев С. А.
Б49 Web-технологии: лабораторный практикум. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2019. 76 с.

ISBN 978-5-7629-2409-2

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

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

УДК 004.432
ББК З 988.02–018я7

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

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия

ISBN 978-5-7629-2409-2

© СПбГЭТУ «ЛЭТИ», 2019

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

– тетрис на JavaScript;
– REST-приложение управления библиотекой;
– модуль администрирования приложения «Аукцион картин»;
– модуль приложения «Участие в аукционе картин»;
– 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://learn.javascript.ru/) – язык программирования, он поддерживает объектно-ориентированный и функциональный стили программирования. Является реализацией языка ECMAScript.

Общая формулировка задачи

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

– сервер – nginx, протокол взаимодействия – HTTPS;

– 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 JetBrains WebStorm (студентам предоставляется бесплатная лицензия) – https://www.jetbrains.com/webstorm/.

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

В качестве среды выполнения для JavaScript в ОС Windows следует использовать Node.JS (скачать LTS-версию по адресу https://nodejs.org/en/), в ОС Linux использовать Node.JS версии не ниже 8 (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.

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

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

В результате будут созданы два ключа:

1) example.key – закрытый ключ;

2) example.csr – открытый ключ.

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

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

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

```
listen 443 ssl default_server;
ssl_certificate /etc/ssl/certs/example.crt;
ssl_certificate_key /etc/ssl/private/example.key;
ssl_session_timeout 20m; # время 20 минут
ssl_session_cache shared:SSL:20m; # размер кеша 20МБ
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
ssl_prefer_server_ciphers on;
ssl_ciphers   ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-
GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-
SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-AES128-GCM-
SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-
AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-
SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-
SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-
SHA:DHE-RSA-AES256-
SHA:!aNULL:!eNULL:!EXPORT:!DES:!RC4:!3DES:!MD5:!PSK;
```

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.

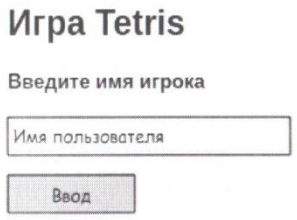

Рис. 1.1. Форма входа

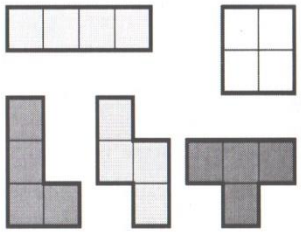

Рис. 1.2. Тетрамино

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.

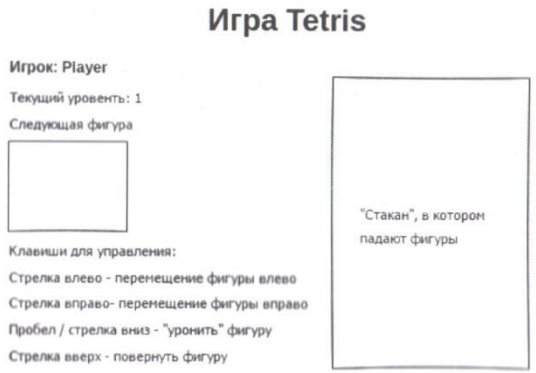

Рис. 1.3. Основная форма игры

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

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

Обработка действия на кнопке может выполняться с использованием следующей команды:

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

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

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

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

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

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

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

Для запуска отложенной команды по таймеру может использоваться команда

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

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

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

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

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

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


В результате в консоли браузера отображаются сообщения:

```
Событие keydown: ArrowUp
Событие keydown: ArrowDown
Событие keydown: ArrowLeft
Событие keydown: ArrowRight
```

Вопросы для контроля

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

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

– Node.JS // URL: https://nodejs.org/;

– TypeScript // URL: https://www.typescriptlang.org/;

– OpenSSL. Cryptography and SSL/TSL Toolkit // URL: https://www.openssl.org/;

– Веб-сервер на основе Nginx и PHP-FPM // URL: http://help.ubuntu.ru/wiki/nginx-phpfpm;

– Nginx // URL: https://nginx.ru/ru/;

– Современный учебник Javascript // URL: https://learn.javascript.ru/;

– MDN web docs. Ресурсы для разработчиков, от разработчиков // URL: https://developer.mozilla.org/ru/;

– Тетрис // URL: https://ru.wikipedia.org/wiki/Тетрис.

Лабораторная работа 2. REST-ПРИЛОЖЕНИЕ УПРАВЛЕНИЯ БИБЛИОТЕКОЙ

Цель и задачи

Целью работы является изучение взаимодействия клиентского приложения с серверной частью, освоение шаблонов web-страниц, формирование на-

выков 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;

– 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 – модуль, позволяющий использовать шаблоны для HTML-страниц.

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

– GET – получение записи (записей);

– POST – добавление записи;

– PUT – обновление или добавление записи;

– DELETE – удаление записи.

Общая формулировка задачи

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

1. Начальное состояние библиотеки хранится в JSON-файле на сервере. Текущее состояние – в переменной в памяти сервера.

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

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

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

5. Предусмотрена страница для карточки книги, в которой ее можно отредактировать (минимум: автор, название, дата выпуска) и дать 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 модульного окна.

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.

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

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>
```

В приложение может быть подключен файл со стилями.

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

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

```
npm init
yarn init
```

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

Установка express может выполнена одной из следующих команд:

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

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

Простейший сервер может выглядеть следующим образом:

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

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

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

```
const server = require("express")(); // Создали сервер
server.get("/page", (req, res, next) => {
    res.end(`Here is a page`); // Ответ
    next(); // Переход к следующему обработчику
});
server.get("/", (req, res, next)=>{
    res.end("ROOT PAGE");
```



```
    next();
});
server.get("*", (req, res)=>{
    res.status(404); // Ошибка - нет такой страницы
    res.end("Page not found");
});
server.listen(3000, ()=>{ // Запуск
    console.log("Server started at http://localhost:3000")
});
```

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

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

Создание шаблонов web-страниц. Шаблоны web-страниц могут создаваться с использованием шаблонов pug. Для этого необходимо установить пакет «pug». При необходимости 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 шаблонов pug.

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

В данном примере «*» означает все остальные маршруты.

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

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

В данном примере, если сервер использует порт 3000 и нам нужна книга с id=123, то обращение происходит с использованием метода GET по адресу http://localhost:3000/books/123. Конструкция «:num» указывает, что в данном фрагменте URL будет передаваться изменяемое значение и ему задается имя «num», это же имя используется для доступа – req.params.num.

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

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

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

Пример json-файла (имя ru.json).

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

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

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 ошибк будет обозначена 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.

```
const server = require('express')();
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
server.use(cookieParser());
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));
const groups = require('./groups.js');
server.use('/groups', groups);
server.listen(3000);
```

Модуль «body-parser» будет использоваться для разбора запросов в формате JSON от браузера. Модуль «cookie-parser» обеспечивает работу с

Cookies. В данном примере основная часть приложения доступна по адресу http://localhost:3000/groups. Все остальные пути – относительные пути, вычисляемые относительно данного адреса.

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

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

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

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

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

Добавление записи – метод POST:

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

В данном случае обрабатывается POST-запрос, в который в качестве параметров (см. «req.body» – это свойство доступно благодаря подключенному модулю «body-parser») переданы значения «name», «students» и «rating». Параметры в GET-запросе передаются в URL, а в POST-запросе – в теле запро-

са. Функция match() обеспечивает проверку регулярного выражения (мы проверяем, что переданные параметры соответствуют тем, что мы ожидаем). На месте многоточия – обработка корректного запроса, в котором переданы все необходимые параметры.

Обновление или добавление записи – метод PUT:

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

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

Удаление записи – метод DELETE:

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

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

Отправка асинхронного GET-запроса с использованием Ajax:

```
const xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200)
        callback(this.responseText);
};
xhttp.open("GET", `/group/${id}`, true);
xhttp.send();
```

Класс XMLHttpRequest позволяет отправлять асинхронные запросы на сервер. Функция onreadystatechange() вызывается при изменении состояния обработки запроса. Значение «this.readyState == 4» означает, что обработка запроса завершена. Значение «this.status == 200» означает успешную обработку запроса. При этом переменная «this.responseText» хранит ответ от сервера. Если ответ в формате JSON, то можно воспользоваться функцией встроенного объекта JSON.parse(this.responseText). Функция callback вызывается в тот момент, когда успешно получен результат запроса.

Функция «open» принимает три параметра: используемый метод, URL, по которому отправляется запрос, и признак, что запрос должен обрабатываться асинхронно. Функция «send» отправляет настроенный запрос.

Отправка асинхронного POST-запроса с использованием Ajax:

```
const xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200)
        callback(this.responseText);
};
xhttp.open("POST", "/group/", true);
xhttp.send(`num=${id}`);
```

Если сравнить отправку запросов GET и POST, то видно, что они отличаются в последних двух строчках. Разница в том, что при использовании GET параметры передаются в URL, а при использовании POST – в теле запроса.

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 для HTML?

4. Приведите пример pug-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. Напишите и объясните команды Ajax-запроса. Почему он может быть асинхронным?

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

– Справочник CSS // URL: https://webref.ru/css;
– W3.CSS Tutorial// URL: https://www.w3schools.com/w3css/;
– Express // URL: https://www.npmjs.com/package/express;
– Express // URL: http://expressjs.com/ru/;
– Pug // URL: https://www.npmjs.com/package/pug;
– Pug. Getting Started // URL: https://pugjs.org/;

– Node.js – RESTful API // URL: https://www.tutorialspoint.com/nodejs/nodejs_restful_api.htm;

– JSON – Introduction // URL: https://www.w3schools.com/js/js_json_intro.asp;

– AJAX Introduction // URL: https://www.w3schools.com/js/js_ajax_intro.asp;

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

– Аутентификация в Node.js с использованием Passport.js // URL: https://medium.com/devschacht/node-hero-chapter-8-27b74c33a5ce.

Лабораторная работа 3. МОДУЛЬ АДМИНИСТРИРОВАНИЯ ПРИЛОЖЕНИЯ «АУКЦИОН КАРТИН»

Цель и задачи

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

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 (серверная часть);

– создание шаблонов web-страниц с использованием pug, указание путей подключения js-файлов;

– разработка стилей web-приложения с использованием LESS;

– разработка клиентских js-файлов с использованием библиотеки jQuery и с использованием новейших возможностей в соответствии последним стандартом ECMAScript;

– конфигурирование GULP для решения задач преобразования pug-файлов в формат HTML, less-файлов в css-файлы, обработка js-файлов с использованием Babel.

Основные теоретические сведения

LESS – это динамический язык стилей, обеспечивает следующие расширения CSS: переменные, вложенные блоки, миксины, операторы и функции. LESS может работать на стороне клиента или на стороне сервера под управлением Node.js.

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

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.

Общая формулировка задачи

Необходимо создать web-приложение, обеспечивающее администрирование аукциона картин: можно выбрать картины для участия в аукционе, определить начальные ставки, перечень участников и параметры аукциона. Основные требования следующие:

1. Перечень доступных картин с описаниями и ссылками на рисунки хранится в JSON-файле.

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

3. Разработка ведется с использованием стандарта не ниже ECMAScript2015.

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

5. Клиентская часть разрабатывается с использованием jQuery (работа с DOM, AJAX-запросы).

6. Предусмотрена HTML-страница для перечня картин и карточка отдельной картины (название, автор, описание, изображение, начальная цена, минимальный и максимальный шаги аукциона). Предусмотрена возможность

редактировать текстовые и числовые параметры, а также включить или исключить картину из участия в предстоящих торгах, загрузить рисунок картины.

7. Предусмотрена HTML-страница для списка потенциальных участников аукциона. Есть возможность добавлять или удалять участников, изменять запас денежных средств.

8. Предусмотрена HTML-страница для настроек аукциона (настройка даты и времени начала аукциона, настройка таймаута продажи картины, настройка интервала времени отсчета до окончания торга по картине, паузы на изучение информации по картине для начала торга по ней).

9. Взаимодействие браузера с сервером осуществляется по протоколу HTTPS.

10. Сборка клиентской части (преобразования less, pug, babel, минификация) осуществляется с использованием GULP.

11. Регистрация и удаление разработанных модулей в npm.

12. Сохранение сформированных настроек в JSON-файл.

Преимуществом будет предоставление возможности с использованием карты (maps.yandex.ru, maps.google.com, openstreetmap.org и т.п.) указать для каждого участника место его проживания.

Описание ключевых методов при выполнении работы

Разработка стилей web-приложения с использованием LESS. LESS – это надстройка над CSS. Любой CSS код – это корректный LESS, но дополнительные элементы LESS не будут работать в простом CSS. LESS добавляет много динамических свойств в CSS. Он вводит переменные, примеси, операции, функции. Попробовать LESS можно на сайте http://lesscss.org/less-preview/.

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

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

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

Для использования на сервере установите модуль «jquery».

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

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

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

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

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

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

jQuery предлагает различные операции по работе с DOM:

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

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

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

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

В jQuery предусмотрены методы по изменению DOM:


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

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

jQuery предлагает методы для обработки событий:

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

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

jQuery предлагает методы отправки AJAX-запросов:

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

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

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

Конфигурирование GULP для решения задач преобразования. Для использования GULP установите модуль «gulp4». Конфигурационный файл по умолчанию – «gulpfile.js».

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

```
const gulp = require("gulp4");
gulp.task("hello2", (callback)=>{
    console.log("hello!!!")
    callback()
});
```

Запуск осуществляется командой «gulp4 hello2». При этом GULP должен быть установлен глобально. Приведенный пример создает задачу с именем «hello2», в качестве второго параметра – выполняемая функция, после успешного выполнения задачи следует вызвать callback(), который передается в качестве параметра в выполняемую функцию. При выполнении данного пример в консоль будет выведено сообщение «hello!!!».

Предусмотрено последовательное и параллельное выполнение задач:

```
gulp.task("exA", gulp.series("ex1", "ex2"));
gulp.task("exB", gulp.parallel("ex1", "ex2"));
```

Функция «series()» обеспечивает последовательное выполнение задач, функция «parallel()» – параллельное:

```
gulp.task("default", ()=>{
    return gulp.src("src/**/*.*")
        .pipe(gulp.dest("dest"));
});
```

Задача «default» выполняется по умолчанию (можно запустить командой «gulp4» без указания имени задачи). 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 gulp = require("gulp4");
const less = require('gulp-less');
const rename = require('gulp-rename');
const cleanCSS = require('gulp-clean-css');
const del = require("del");
const clean = ()=>del(["assets"]);
function styles() {
    return gulp.src('src/**/*.less')
        .pipe(less())
        .pipe(cleanCSS())
        .pipe(rename({
            basename: 'main',
            suffix: '.min'
        }))
        .pipe(gulp.dest('assets/styles/'));
}
gulp.task("default", gulp.series(clean, styles));
```


В данном примере используется несколько дополнительных модулей: «gulp-less» для преобразования LESS-файлов в CSS-файлы; «gulp-rename» для переименования файла в соответствии с заданным шаблоном; «gulp-clean-css» для минификации (удаления лишнего с целью минимизации размера) CSS-файла; «del» для удаления файлов. Данный пример создает две задачи: для удаления папки «assets» и для обработки LESS-файлов. 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 «assets/styles/».

Для обработки babel потребуется установка модулей «gulp-babel», «babel-preset-env», при этом могут оказаться полезными модули «gulp-sourcemaps» для контроля преобразования в ES5, «gulp-concat» для объединения JS-файлов в один файл, «gulp-uglify» для минификации JS-файлов.

Для обработки pug потребуется установка модуля «gulp-pug».

Применение компилятора babel. Для применения компилятора babel (https://babeljs.io/) необходимо установить пакеты «babel-cli» и «babel-preset-env»:

```
npm i --save-dev babel-cli babel-preset-env
```

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

```
{
  "presets": ["env"]
}
```

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

Исходный JS-файл	Преобразованный JS-файл

`let data = 5`
`if(data > 2)`
`console.log(`two ${data}`)`
	`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 предназначен LESS? Приведите примеры описания переменных, миксинов, операторов.

2. Приведите примеры использования jQuery для изменения DOM. В чем отличия данного подхода от традиционного JavaScript?

3. Приведите примеры использования jQuery для отправки AJAX-запросов. В чем отличия данного подхода от традиционного JavaScript?

4. Для чего предназначен babel? Какие модули используются для его корректной работы? Приведите примеры команд запуска и результатов его работы.

5. Для чего предназначен GULP? Приведите примеры с использованием less, babel, pug, минификации.

6. Какие команды следует использовать для регистрации нового модуля npm? Как воспользоваться созданным модулем? Как обновить версию модуля?

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

– Babel is a JavaScript compiler // URL: https://babeljs.io/;

– jQuery write less do more // URL: https://jquery.com/;

– {less} // URL: http://lesscss.org/;

– Gulp. Automate and enhance your workflow // URL: https://gulpjs.com/;

– NPM. Build amazing things // URL: https://www.npmjs.com/;

– Yarn. Fast, reliable, and secure dependency management // URL: https://yarnpkg.com/;

– ES6, ES8, ES2017: что такое ECMAScript и чем это отличается от JavaScript // URL: https://tproger.ru/translations/wtf-is-ecmascript/.

Лабораторная работа 4. МОДУЛЬ ПРИЛОЖЕНИЯ «УЧАСТИЕ В АУКЦИОНЕ КАРТИН»

Цель и задачи

Целью работы является изучение возможностей применения jQuery UI для создания интерфейса пользователя, обработки ошибок, ведения журналов ошибок, реализация взаимодействия приложений с использованием web-сокетов, применение статического анализатора Flow, освоение инструмента сборки WebPack и организации модульного тестирования web-приложений с использованием Mocha.

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 (серверная часть);

– создание шаблонов web-страниц, указание путей подключения js-файлов;

– разработка клиентских js-файлов с использованием библиотек jQuery и jQuery UI;

– обработка ошибок на сервере с использованием try/catch, функции catch() у промисов и механизма «domain»;

– регистрация и подключение в web-приложение журнала ошибок.

– создание web-сокета для отправки сообщений всем клиентам;

– использование статического анализатора Flow для анализа кода;

– создание и запуск Mocha тестов для приложения;

– конфигурирование сборщика WebPack.

Основные теоретические сведения

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

Механизм «domain» Node.JS не рекомендуется использовать в связи с разработкой новых механизмов обработки ошибок, но именно он позволяет обрабатывать асинхронные ошибки. При возможности используйте традиционный механизм try/catch или функцию catch() у промисов (https://learn.javascript.ru/promise).

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/).

Flow (https://flow.org/) – инструмент статического анализа JavaScript (разработки Facebook), позволяющий контролировать изменение типов переменных.

Mocha (https://mochajs.org/) – это фреймворк для написания тестов серверной части web-приложений, поддерживает разработку, основанную на тестах (TDD – test-driven development), и разработку, основанную на поведении (BDD – behavior-driven development). Может использоваться совместно с другими библиотеками для тестирования, (например, Shell и Chai).

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 участие в аукционе картин. Каждый участник может подключиться к аукциону картин. В заданное время начинается аукцион. Участники могут торговаться и повышать стоимость продажи картины. Информация о ходе торгов и сделанных ставках рассылается всем участникам с учетом заданной конфигурации аукциона. Аукцион заканчивается, когда заканчивается торг по последней картине. Часть картин может остаться не проданной. Основные требования:

1. Приложение получает исходные данные из модуля администрирования приложения «Аукцион картин» в виде настроек в формате JSON-файла.

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

3. Участники аукциона подключаются к приложению «Участие в аукционе картин».

4. Предусмотрена HTML-страница администратора, на которой отображается перечень участников аукциона, перечень картин, текущее состояние по каждой картине (минимальная цена, кому продана, за какую цену), окно (область на странице) с сообщениями о ходе торгов.

5. Предусмотрена HTML-страница участника, на которой отображается информация о балансе средств участника, текущей продаваемой картине, времени, которое прошло с начала торгов, окно (область на странице) с сообщениями о ходе торгов, информация о начальной и текущей цене, предусмотрена кнопка «Предложить новую цену», предусмотрен виджет, позво-

ляющий указать сумму повышения цены (по умолчанию – минимальное допустимое значение), ссылка (или кнопка) перехода на страницу с покупками.

6. Обеспечен контроль, что картину купит только один участник, проверяется, что у участника хватает средств на покупку.

7. Окончание торга по картине осуществляется после истечения заданного времени (соответствующий отсчет ведется в окне с сообщениями о ходе торгов).

8. Окно с сообщениями о ходе торгов предназначено для предоставления актуальной информации о ходе торгов (время и текст с соответствующей цветовой подсветкой сообщений). В качестве сообщений как минимум выступают: сообщение о начале торгов в целом, о начале торгов по картине, о подаче заявке, о повышении цены, обратный отсчет об окончании торга по картине, об окончании торгов в целом.

9. Предусмотрена HTML-страница, на которой можно ознакомиться со всеми картинами, купленными участником торгов.

Преимуществом будет использование звукового сопровождения событий: начало и окончание торгов в целом, обратный отсчет об окончании торга.

Описание ключевых методов при выполнении работы

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

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

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

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

Рассмотрим примеры.

1.
```
$("#d1").draggable({
        cursor: "move",
        cursorAt: {
```



```
            top: 50, left: 50
        }
    });
```

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

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

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

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

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

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

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

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

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

Промисы – объекты, которые выполняются асинхронно.

Рассмотрим примеры.

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

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

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

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

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

3. Промисы 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 промисов. Функция «then» исполняет промис. Последовательно вызываемые функции «then» формируют цепочку вызова. Результаты выполнения промисов передаются от одного «then» следующему. В данном примере второй «then» получает в переменную item значение «ok» из первого промиса. Во втором промисе 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».

4. Идеология доменов (domain) позволяет перехватывать «асинхронные» ошибки:

```
const domain = require("domain");
const server = domain.create();
server.on("error", (err) => {
    console.log("Домен перехватил ошибку %s", err);
});
server.run(()=>{
   wrongFunctionName();
});
```

В данном примере при вызове некорректной функции домен перехватит ошибку и выведет соответствующее сообщение.

5. Домены применимы и при полноценной разработке с использованием сервера:

```
let domain = require("domain"),
    http = require("http");
let d = domain.create(), server;
d.on("error", (err) => {
    console.log("Домен перехватил ошибку %s", err);
});
d.run(() => {
    server = new http.Server();
});
server.on("request", ()=>{
    setTimeout(()=>{
        wrongFunctionName();
    }, 200);
});
server.emit("request");
```

В данном примере создается сервер; так как каждый сервер является EventEmitter, то создается обработчик на событие «request», в рамках которого с использованием таймера вызывается некорректная функция. При вызове события «request» домен перехватывает сообщение об ошибке. Это возможно за счет того, что сервер создается в рамках домена (функции «run»). Таким образом на довольно простом примере можно продемонстрировать асинхронную обработку ошибок.

В настоящее время разработчики Node.JS исследуют возможности отказаться от доменов в том числе в связи со сложностями организации одновременной работы доменов и промисов.

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:

```
const io = require('socket.io').listen(3030);
io.sockets.on('connection', function (socket) {
    socket.emit('hello', 'Сообщение "hello" от socket.io');
});
```


В данном примере загружается модуль «socket.io» и запускается слушатель на порту «3030» (порт должен отличаться от используемого вами порта web-сервера).

Для сокета с использованием функции «io.socket.on» настраивается обработчик события «connection» и указывается функция-обработчик, которая в качестве параметра принимает соединение «socket».

Соединение является EventEmitter, в рамках которого передаются сообщения, поэтому, используя функцию «emit», передаем событие «hello» с текстом «Сообщение…».

Пример клиентской части приложения:

```
<span id="out"></span>
<script src="http://localhost:3030/socket.io/socket.io.js">
</script>
<script>
   let socket = io.connect("http://localhost:3030");
   socket.on("hello", (data)=>{
       out.innerHTML = data;
   });
</script>
```

В данном примере задается область вывода сообщения с сервера «span» с идентификатором «out», подключается скрипт «socket.io.js» (обратите внимание, эта библиотека становится доступной для клиента), в рамках библиотеки становится доступна переменная «io». Затем создается соединение с сервером «io.connect» и настраивается соответствующий обработчик «socket.on» на событие «hello», переданное сообщение отображается в область вывода «span».

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:

```
const io = require('socket.io').listen(3030);
io.sockets.on('connection', (socket)=>{
    socket.on('hello', (msg)=>{
        socket["name"] = msg.name;
        send(socket, `Присоединился ${msg.name}`);
    });
    socket.on('msg', (msg)=>{
        send(socket, `${socket["name"]}: ${msg.value}`);
    });
    socket.on('disconnect', (msg)=>{
        send(socket, `Покинул ${socket["name"]}`);
```

```
    });
});
function send(socket, msg) {
    let time = (new Date()).toLocaleTimeString();
    socket.json.emit("msg", {"message": `${time} ${msg}`});
    socket.broadcast.json.emit("msg", {"message": `${time}
${msg}`});
}
```

Строчки загрузки и нового соединения мы рассмотрели на предыдущем примере. Сокет «socket» настраивается на обработку трех типов событий «hello», «msg» и «disconnect». Во всех случаях вызывается функция «send» для отправки сообщений клиентам.

В случае события «hello» 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». В обработчиках событий «msg» и «disconnect» используется сохраненное в «socket» свойство «name».

Функция «send» вычисляет текущее время, а затем с использованием «socket.json.emit» отправляет сообщение «msg» в формате JSON отправителю и с использованием функции «socket.broadcast.json.emit» отправляет сообщение «msg» в формате JSON всем остальным, подключенным к этому сокету.

Пример клиентской части приложения:

```
<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)"> Отпра-
вить</button><br>
<ul id="data"></ul>
<script src="http://localhost:3030/socket.io/socket.io.js">
</script>
<script>
   var socket;
   function chat(){
        const nick = document.getElementById("name").value;
        socket = io.connect("http://localhost:3030");
        socket.on("connect", () => {
            socket.json.emit("hello", {"name": nick});
        });
        socket.on("msg", (msg)=>{
            addUL(msg.message);
        });
        login.disabled = true;
        document.getElementById("send")
            .removeAttribute('disabled');
   }
```



```
   function send(value) {
       if(socket)
           socket.json.emit("msg",{"value": value});
   }
   function addUL(text) {
       let li = document.createElement("li");
       li.innerHTML = text;
       data.appendChild(li);
   }
</script>
```

Элемент <input id='name'> хранит имя пользователя, элемент <input id='send'> используется для ввода текста 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 обработчика сообщений. Вызов «socket.json.emit» с типом «hello» 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.json.emit») они создаются в виде JSON-объектов, при получении ожидается, что от сервера получены JSON-объекты, в частности, у них есть поле «message».

Использование статического анализатора Flow для анализа кода. Для статического анализа JavaScript используется Flow. Установка осуществляется в два этапа: установите модули «flow-bin» и «flow-remove-types». Есть возможность вместо «flow-remove-types» использовать модуль babel, но мы не будем рассматривать этот вопрос в рамках лабораторной работы. На втором этапе в «package.json» необходимо в раздел скриптов прописать строки запуска:

```
"scripts": {
  "build": "flow-remove-types src/ -d lib/",
  "prepublish": "npm run build",
  "flow": "flow"
}
```

При выполнении команды «npm run flow» будет запускаться «flow», а при выполнении сборки будет запускаться «flow-remove-types», при этом перекладывая исходные коды из папки «src» в папку «lib».

Для работы Flow его необходимо проинициализировать:

```
npm run flow init
```

В результате инициализации будет создан файл «.flowconfig» с настройками Flow по умолчанию.

Для обеспечения проверки js-файла Flow в первой строчке файла должен появиться комментарий:

```
// @flow
```

или

```
/* @flow */
```

Данный комментарий является признаком того, что файл должен проверяться Flow.

Рассмотрим несколько примеров.

1.
```
let z:string = "23";
if(z == 23)
  console.log("Ok");
```

В данном примере переменной «z» задан тип «string», Flow выдаст сообщение об ошибке: «This type cannot be compared to number». Предусмотрена поддержка основных примитивных типов: string, number, boolean. Учитывается использование null и undefined. Если для переменной указать тип «any», то проверки для данной переменной будут отключены.

2.
```
function sum(x, y) {
    return x + y;
}
sum(2, "2")
```

В данном случае будет сообщение об ошибке «This type cannot be added to string». При необходимости типы могут задаваться в том числе для параметров функций.

3.
```
function sum(x:number = 2, y:number = 3) {
    console.log(x + y);
    return x + y;
}
sum(2, 3)
```

Данный пример не приведет к появлению ошибки. Есть только одна особенность: в данном примере предусмотрены значения по умолчанию «x=2» и «y=3», но использование значений по умолчанию корректно обрабатывается только при использовании babel.

4. Расширим пример:

```
if(sum < 5)
  console.log("less");
else
  console.log("more");
```


В данном примере сознательно опущены скобки при вызове функции. В данном случае будет сообщение об ошибке «This type cannot be compared to number».

Flow поддерживает широкий спектр возможных проверок, в том числе для массивов, вводит понятие кортежей, контролирует структуру объектов, классов, интерфейсов, предлагает свою реализацию дженериков и т. д.

Подробно с документацией и вариантами применения можно ознакомиться по адресу https://flow.org/.

Создание и запуск Mocha тестов для приложения. Модуль «mocha» (должен быть установлен) позволяет реализовывать модульное тестирование. Mocha интегрируется с множеством библиотек для тестирования, например, встроенной «assert», внешними «chai», «should» и другими.

Пример применения «assert»:

```
const assert = require("assert")
assert.notEqual(1, fib(0))
assert.equal(1, fib(1))
```

В данном примере обеспечивается сравнение числа с результатом вызова функции «fib» с разными параметрами. При этом должно быть не верно «fib(0) === 1», иначе будет выдано сообщение об ошибке. И должно быть верно «fib(1) === 1», иначе будет выдано сообщение об ошибке. Третьим параметром в функции «equal» и «notEqual» может передаваться текст сообщения об ошибке.

Следует отметить, что аналог функции «assert.equal» есть в глобальном объекте «console»:

```
console.assert(1 === fib(1), "Проверка начального значения
fib()")
```

Пример использования «mocha»:

```
const assert = require("assert")
let fib = require("../src/fib")
describe("fib", function(){
    it("fib проверка #1", function () {
        assert.equal(21, fib(8))
        assert.equal(5, fib(5))
        assert.equal(55, fib(10))
    })
    it("fib проверка #2", function () {
        assert.notEqual(1, fib(0))
        assert.equal(1, fib(1))
    })
})
```

В данном примере первой строчкой подключается стандартный модуль «assert», второй строчкой подключается модуль, который будет тестироваться, затем функция «describe» описывает набор тестов. Каждый тест в отдельности описывается функцией «it». Тестирование осуществляется с использованием методов стандартных (или отдельно подгруженных) библиотек.

При необходимости предусмотрена вложенность наборов тестов (один «describe» может находиться внутри другого «describe»).

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

```
"scripts": {
  "test": "mocha"
}
```

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

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

Основные элементы Webpack:

– entry – точки входа;

– output – вывод результата;

– loaders – загрузчики;

– plugins – подключаемые компоненты.

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

```
webpack
```

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

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

Пример простейшего конфигурационного файла:

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

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

Предусмотрена поддержка нескольких точек входа:


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

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

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

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

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

Рассмотрим пример использованного при этом «main4.js»:

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

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

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

```
module.exports = {
    entry : './src/main5.js',
    output: {
        filename: './dist/bundle.js'
    },
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /(node_modules|bower_components)/,
                use: {
                    loader: 'babel-loader',
                    options: {
                        presets: ['env']
                    }
                }
            }
        ]
    }
};
```

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

Вопросы для контроля

1. Для чего предназначена библиотека jQuery UI? Приведите примеры ее использования.
2. Приведите примеры использования промисов и механизма «domain».
3. Какой журнал ошибок использовался в лабораторной работе? Приведите пример использования.
4. Что такое WebSocket? Приведите пример создания web-сокета на серверном и на клиентском приложении.

5. Для чего нужен статический анализатор Flow? Приведите примеры использования.
6. Приведите пример Mocha-теста.
7. Для чего нужен WebPack? Приведите пример простейшей конфигурации WebPack.

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

– jQuery user interface // URL: https://jqueryui.com/

– Node.js v8.9.4 Documentation. Domain // URL: https://nodejs.org/docs/latest-v8.x/api/domain.html

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

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

– JavaScript error tracking with Sentry // URL: https://sentry.io/for/javascript/

– Socket.io 2.0 is here // URL: https://socket.io/

– Flow is a static type checker for JavaScript // URL: https://flow.org/

– Mocha. Simple, flexible, fun // URL: https://mochajs.org/

– Node.js v8.9.4 Documentation. Assert // URL: https://nodejs.org/docs/latest-v8.x/api/assert.html

– Chai Assertion Library // URL: http://chaijs.com/

– BDD style assertions for node.js // URL: https://github.com/shouldjs/should.js

– Webpack // URL: https://webpack.js.org/

Лабораторная работа 5. МОДУЛЬ АДМИНИСТРИРОВАНИЯ ПРИЛОЖЕНИЯ «БИРЖА АКЦИЙ»

Цель и задачи

Целью работы является изучение основ языка TypeScript и особенностей применения фреймворка Angular для разработки web-приложений.

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;
– обеспечение взаимодействия с сервером приложения.

Основные теоретические сведения

Angular – фреймворк для разработки клиентских частей web-приложений, основанный на языке TypeScript, поддерживаемый компанией Google. 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.

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, время начала и время окончания торгов. Основные требования следующие:

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

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

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. Для каждой акции задаются правила изменения во времени (закон распределения: равномерный, нормальный; максимальное значение для изменения, общее количество доступных акций, начальная стоимость одной акции). Предусмотрена возможность добавления, редактирования и удаления акций.

5. Предусмотрена HTML-страница для настроек биржи (время начала и окончания торгов, интервал времени, через который пересчитывается стоимость акций).

6. Все элементы управления реализованы с использованием компонентов Angular. Взаимодействие между компонентами реализовано с использованием сервисов Angular.

7. Для реализации эффектов на HTML-страницах используются директивы Angular.

Преимуществом будет создание и использование аутентификации на основе passport.js (http://www.passportjs.org/) с реализацией интерфейса на основе Angular-компонентов. Допустима модификация программы, когда изменения котировок акций не генерируются случайным образом, а настраиваются на основании данных из файла (реальные котировки, например, с nasdaq.com*), при этом у пользователя должна быть возможность настроить день, с которого начнется отсчет и указать скорость, с которой будут поступать данные (так как в реальных данных они хранятся с частотой 1 запись в день).

Описание ключевых методов при выполнении работы

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

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

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

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

```
ng serve
```

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

```
ng serve --open
```

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

*Символы можно найти по адресу https://www.nasdaq.com/markets/stocks/symbol-change-history.aspx или в списке компаний https://www.nasdaq.com/screening/company-list.aspx Исторические данные для каждого символа с выгрузкой в csv-формате можно загрузить по адресу https://www.nasdaq.com/quotes/historical-quotes.aspx

```
"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.

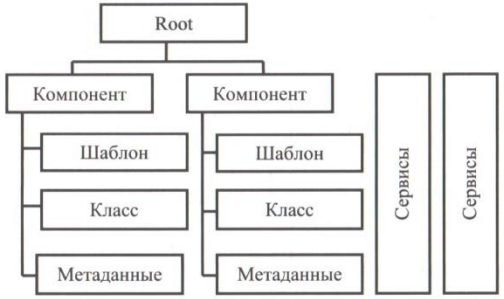

Рис. 5.1. Обобщенная архитектура приложения на Angular

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». В данном случае совпадение дженерика «boolean» (у «EventEmitter») и типа переменной «boolean» – совпадение, а не требование.

Данное событие обрабатывается в родительском компоненте «AppComponent»:

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

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

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

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<p *ngIf="myCondition">Истина</p>
  <p *ngIf="!myCondition">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».

Того же эффекта можно достичь, используя «else»:

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<p *ngIf="myCondition;else noCondition">Истина!</p>
  <ng-template #noCondition>
    <p>Ложь!</p>
  </ng-template>
  <button (click)="toggle()">Переключить</button>`
})
export class AppComponent {
  myCondition: boolean = true;
  toggle() {
    this.myCondition = !this.myCondition;
  }
}
```

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

Аналогично доступны директивы «ngFor» и «ngSwitch»:

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



```
      <li *ngFor="let item of myList">{{item}}</li>
    </ul>`
})
export class AppComponent {
  myList = ["один", "два", "три"]
}
```

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

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

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

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

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

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

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

```
import {Component, OnInit} from '@angular/core';
import {ToDo} from "./ToDo";
import {TodoService} from "./todo.service";
@Component({
  selector: 'app-root',
  template: `
    <input [(ngModel)]="title"><br>
    <button (click)="add(title)">Добавить</button>
    <ul>
      <li *ngFor="let todo of mydata">{{todo.title}}</li>
```

```
    </ul>`,
  providers: [ TodoService ]
})
export class AppComponent implements OnInit {
  mydata: ToDo[]
  constructor(private todoService: TodoService) { }
  ngOnInit(): void { // При инициализации компонента
    this.mydata = this.todoService.getData()
  }
  add(title: string) { // Добавление данных
    this.todoService.addData(title)
  }
}
```

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

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

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

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

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

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

Перенесем логику в компонент «ChildComponent»:

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



```
})
export class ChildComponent implements OnInit{
  mydata: ToDo[]
  constructor(private todoService: TodoService) { }
  ngOnInit(): void { // При инициализации компонента
    this.mydata = this.todoService.getData()
  }
  add(title: string) { // Добавление данных
    this.todoService.addData(title)
  }
}
```

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

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

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

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

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

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

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

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

```
import { ToDo } from "./ToDo"
export class TodoService {
  private data: ToDo[] = [
    {title: "Нет данных"}
  ]
  getData(): ToDo[] {
```

```
    return this.data;
  }
  addData(title: string) {
    this.data.push(new ToDo(title))
    console.log(`Добавлен: "${title}"`)
  }
}
```

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

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

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

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

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

Компонент «src/app/notfound.ts».

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

В качестве главного компонента будет выступать «src/app/base.component.ts»:

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

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

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

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

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

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

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

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.

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

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

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

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

– Angular 2 Tutorial // URL: https://www.tutorialspoint.com/angular2/

– Angular Tutorial: Getting Started With Angular 4 // URL: https://www.edureka.co/blog/angular-tutorial/

– Learn Angular 5 from Scratch - Angular 5 Tutorial // URL: https://coursetro.com/courses/19/Learn-Angular-5-from-Scratch---Angular-5-Tutorial

– Angular Tutorial For Beginners to Professionals // URL: https://www.tektutorialshub.com/angular-2-tutorial/

– Пятиминутка Angular // URL: https://soundcloud.com/5minangular

– 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 6. МОДУЛЬ ПРИЛОЖЕНИЯ «ПОКУПКА И ПРОДАЖА АКЦИЙ»

Цель и задачи

Целью работы является изучение возможностей применения библиотеки React (https://reactjs.org/) для разработки интерфейсов пользователя web-приложений. Для достижения поставленной цели требуется решить следующие задачи:

– разработка интерфейса web-приложения;

– создание web-сервера на основе express. Подготовка web-сокетов для обновления информации о стоимости у всех клиентов;

– создание каркаса web-приложения с использованием React;

– 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.

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), с которым брокер начал день, и текущее состояние. Основные требования следующие:

1. Приложение получает исходные данные из модуля администрирования приложения «Биржа акций» в виде настроек в формате JSON-файла.

2. В качестве сервера используется Node.JS с модулем express (либо ngnix в связке с PHP в качестве «back-end»).

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 количество, выставленное на торги. Предусмотрена кнопка «Начало торгов» и выбор закона распределения для цены акций.

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

6. Предусмотрена HTML-страница, на которой участнику отображается общее количество доступных ему средств, количество и суммарная стоимость по каждому виду купленных акций. На ней же отображается количество выставленных на торги акций, их количество и стоимость. У участника есть возможность купить/продать интересующее его количество акций. Участнику отображается суммарный доход на начало торгов и на текущий момент времени. Участник не может купить акции, если денег не хватает.

Преимуществом будет сохранение проекта в локальный git и использование в качестве сервера приложений nginx в связке с PHP в качестве «back-end» для обработки запросов от пользователя (клиентская часть при этом все равно должна разрабатываться с использованием React).

Описание ключевых методов при выполнении работы

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

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

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

Для запуска приложения необходимо выполнить следующую команду:

```
npm run start
```

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

Основной файл приложения «src/App.js».

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

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


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

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

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

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

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

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

Фрагменты JSX можно создавать и за пределами функций и классов:

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

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

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

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

```
standalone/6.25.0/babel.min.js"> </script>

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

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

Компоненты могут вкладываться один в другой:

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

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

Следует обратить внимание на некоторые моменты:

– передаваемые в компонент параметры попали в переменную «props» в виде свойств объекта;

– для объединения компонентов в «App» используется – React-компонент должен возвращать один элемент, если необходимо вернуть несколько, то их следует объединить в один;

– возвращаемое значение в «App» объединено в круглые скобки. Это позволяет писать его на нескольких строчках.

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

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



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

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

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

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

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

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

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

Расширим форму элементами ввода:

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

В конструкторе определяем состояние, содержащее два текстовых поля, и привязываем две функции к «this». Функция «onChange()» будет использоваться на элементе <input>, поэтому результат ввода пользователя может быть получен из «e.target.value». Функция «onSubmit()» будет использоваться

на кнопке «submit», поэтому в ней необходимо запретить выполнение действия по умолчанию «e.preventDefault()», основная задача функции – копирование значения из «this.state.text» в «this.state.outtext».

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

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

```
import React, {Component} from 'react'
import { BrowserRouter, Route, Switch, Link } from 'react-
router-dom'
class About extends Component{
    render(){ return <h1>about</h1> }
}
class NotFound extends Component{
    render(){ return <h1>not found</h1> }
}
class Main extends Component{
    render(){ return <h1>main</h1> }
}
class App extends Component {
    render() {
        return (
            <BrowserRouter>
                <Switch>
                    <Route exact path="/" component={Main} />
                    <Route path="/about" component={About} />
                    <Route component={NotFound} />
                </Switch>
            </BrowserRouter>
        );
    }
}
export default App
```

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

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

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

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

Ключевые понятия Redux:

– хранилище (store) – хранит состояние приложения;

– действия (actions) – набор команд, которые отправляются приложением в хранилищу;

– создатели действий (action creators) – функции, которые создают действия;

– reducer – функции, которые получают действия и в соответствии с ними изменяют состояние хранилища.

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

Для работы приложения необходимо добавить пакеты «immutable», «react-redux», «redux». В результате раздел «dependencies» файла «project.json» может выглядеть следующим образом:

```
"dependencies": {
  "immutable": "^3.8.2",
  "react": "^16.2.0",
  "react-dom": "^16.2.0",
  "react-redux": "^5.0.7",
  "react-router-dom": "^4.2.2",
  "react-scripts": "1.1.1",
  "redux": "^3.7.2"
},
```

При создании приложения необходимо определить, какие действия будут выполняться, предположим, это действия «добавление» и «удаление» записи («actions.js»):

```
const ADD = "ADD_RECORD"
const DELETE = "DELETE_RECORD"
const SET = "SET_STATE"
module.exports = {SET, ADD, DELETE}
```

Действие «set» необходимо для начальной инициализации приложения.

Далее описываются соответствующие создатели действий («actionscreators.js»):


```
const {ADD, DELETE} = require("./actions")
const addRecord = function (record) {
    return { // Добавление
        type: ADD,
        record
    }
};
const deleteRecord = function (record) {
    return { // Удаление
        type: DELETE,
        record
    }
};
module.exports = {addRecord, deleteRecord};
```

Каждая функция возвращает действие и параметры действия.

Далее описываются «reducers», которые описывают, как действия влияют на изменение состояния приложения («reducers.js»), но прежде чем их описывать необходимо продумать состояние приложения. В данном примере состояние будет содержать объект «{records: []}», В массив будут помещаться строки. «Reducer» – это чистые функции со следующим видом «(previousState,action)=>newState», они вычисляют новое состояние, но не влияют на предыдущее состояние:

```
const {SET, ADD, DELETE} = require("./actions")
const Map = require("immutable").Map
const reducer = function(state = Map(), action) {
    switch (action.type) {
        case SET: // Начальное состояние
            return state.merge(action.state)
        case ADD: // Добавление
            return state.update("records",
                (records) => records.push(action.record))
        case DELETE: // Удаление
            return state.update("records",
                (records) => records.filterNot(
                    (item) => item === action.record
                )
            )
        default:
    }
    return state
}
module.exports = reducer
```

В данном случае для хранения состояния использован «Map» из библиотеки «Immutable» (https://facebook.github.io/immutable-js/docs/#/Map), при этом использованы функции «merge()», которая обеспечивает объединение двух состояний, «update()», которая обновляет текущее состояние,

«filterNot()», которая для каждой записи проверяет выполнение заданного условия и фильтрует соответствующие значения. Следует отметить, что созданный таким образом «Map» не изменяется, при вызове функций, требующих изменения, формируется новый экземпляр «Map». В данном примере в «Map» задействован только один ключ «records».

В описанной функции по умолчанию «state» является «Map». При инициализации начального состояния ожидается, что массив «{records: []}» передан в качестве «action.state». При добавлении используется чистая функция, использующая «push» для добавления записи «action.record». При удалении записи используется чистая функция фильтрации заданного значения.

Далее создается представление «views.js»:

```
const React = require("react")
const connect = require("react-redux").connect
const actions = require("./actionscreators")
class MyInputForm extends React.Component {
    constructor(props) {
        super(props) // Привязка функции
        this.onClick = this.onClick.bind(this)
    }
    onClick() {
        const text = this.refs.myInput.value
        if (text !== "") { // Если введен текст
            this.refs.myInput.value =""
            return this.props.addRecord(text)
        }
    }
    render() { // Форма ввода текста
        return <div>
            <input ref="myInput"/>
            <button onClick={this.onClick}>Добавить</button>
        </div>
    }
}
class MySingleRecord extends React.Component {
    render() { // Отображение одной записи
        return  <div>
                    <span>{this.props.text}</span>
                    <button onClick={() =>

this.props.deleteRecord(this.props.text)}>X</button>
                    <br/>
                </div>
    }
}
class RecordsList extends React.Component {
    render() { // Список записей
```



```
        return  <div>
                    {this.props.records.map(item =>
                        <MySingleRecord key={item} text={item}
                                        deleteRe-
cord={this.props.deleteRecord}/>
                    )}
                </div>
    }
}
class View extends React.Component {
    render() { // Основной компонент
        return <div>
            <MyInputForm addRecord={this.props.addRecord}/>
            <RecordsList {...this.props} />
        </div>
    }
}
function mapStateToProps(state) {
    return { records: state.get("records") }
}
module.exports = connect(mapStateToProps, actions)(View)
```

В данном случае создано четыре компонента.

Первый компонент «MyInputForm» обеспечивает создание компонента ввода новой записи. При выполнении «клика» вызывается функция «onClick», если текст введен, значит, добавляется в «props».

Второй компонент «MySingleRecord» отображает одну запись с кнопкой удаления. При нажатии кнопки вызывается функция «deleteRecord».

Третий компонент «RecordsList» отображает список записей из компонента «this.props.records» и отображает их с использованием «MySingleRecord».

Четвертый компонент «View» отображает форму ввода и список записей. В качестве параметра используется массив «this.props».

Функция «mapStateToProps()» используется для привязки состояния приложения и свойств «this.props».

Функция «connect()» обеспечивает связку хранилища и компонента «View». В результате подключения компонент обеспечит связку с действиями «actions». Именно благодаря данной привязке будут работать «this.props.deleteRecord» и «this.props.addRecord».

Первые два параметра «connect»: mapStateToProps и mapDispatchToProps.

mapStateToProps() – это утилита, которая помогает компоненту получать обновленное состояние (которое обновляется некоторыми другими компонентами).

mapDispatchToProps() – это утилита, которая поможет компоненту запускать событие действия (диспетчерское действие, которое может вызвать изменение состояния приложения).

Основное отображение будет осуществляться с использованием «App.js»:

```
const {SET} = require("./actions")
const React = require("react");
const redux = require("redux");
const Provider = require("react-redux").Provider;
const reducer = require("./reducers");
const View = require("./views");
const store = redux.createStore(reducer);
store.dispatch({ // Инициация начального состояния
    type: SET,
    state: {
        records: [ "Выучить React", "Забыть React" ]
    }
});
// Создание приложения
class App extends React.Component {
    render() {
        return (
            <Provider store={store}>
                <View />
            </Provider>
        )
    }
}
export default App
```

В данном случае с использованием функции «createStore()» осуществляется связка «reducer» с хранилищем, выполняется инициализация начального состояния приложения, создается компонент для отображения, в котором «Provider» связывает React и «redux».

Созданное приложение позволяет добавлять новые записи и удалять уже добавленные.

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?

Дополнительные источники в сети Интернет

Обычно используются следующие источники:

– React. A JavaScript library for building user interfaces // URL: https://reactjs.org/;

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

– React.js на русском языке. Перевод официальной документации // URL: https://abraxabra.ru/react.js/;

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

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

– Redux // URL: https://redux.js.org/;

– Immutable // URL: https://www.npmjs.com/package/immutable.

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,

2) «скриншоты» всех страниц web-приложения,

3) отчеты по всем лабораторным работам.

Список рекомендуемой литературы

Гоше Х. Д. HTML5 Для профессионалов. СПб.: Питер, 2015.

Симпсон К. ES6 и не только. СПб.: Питер. 2017.

Стефанов С. React.js быстрый старт. СПб.: Питер, 2017.

Фримен А. Angular для профессионалов. СПб.: Питер, 2018.

Шелли П. Изучаем Node. Переходим на сторону сервера. СПб.: Питер, 2017.

Оглавление

Лабораторная работа 1. Тетрис на JavaScript 3
Лабораторная работа 2. REST-приложение управления библиотекой 9
Лабораторная работа 3. Модуль администрирования приложения «Аукцион картин» 20
Лабораторная работа 4. Модуль приложения «Участие в аукционе картин» 29
Лабораторная работа 5. Модуль администрирования приложения «Биржа акций» 45
Лабораторная работа 6. Модуль приложения «Покупка и продажа акций» 62

Беляев Сергей Алексеевич

Web-технологии

Лабораторный практикум

Редактор М. Б. Шишкова

Подписано в печать 12.03.2019. Формат 60×84 1/16.
Бумага офсетная. Печать цифровая. Печ. л. 4,75.
Гарнитура «Times New Roman». Тираж 81 экз. Заказ 20.

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