Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)**

УТВЕРЖДАЮ
Заведующий кафедрой ИСПИ
_____ И.Е. Жигалов
«20» марта 2025 г.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ МЕЖДИСЦИПЛИНАРНОГО КУРСА

«РАЗРАБОТКА КЛИЕНТСКОЙ ЧАСТИ ИНФОРМАЦИОННЫХ РЕСУРСОВ»

В РАМКАХ ПРОФЕССИОНАЛЬНОГО МОДУЛЯ

«РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ НА СТОРОНЕ КЛИЕНТА»

09.02.09 Веб-разработка
Разработчик веб приложений

Владимир, 2025

Методические указания к лабораторным работам междисциплинарного курса «Разработка клиентской части информационных ресурсов» разработал преподаватель КИТП Максимова А.С.

Методические указания к лабораторным работам рассмотрены и одобрены на заседании УМК специальности 09.02.09 Веб-разработка
протокол № 1 от «10» марта 2025 г.

Председатель УМК специальности _____ И.Е. Жигалов

Методические указания к лабораторным работам рассмотрены и одобрены на заседании кафедры ИСПИ
протокол № 7а от «12» марта 2025 г.

Рецензент от работодателя:
руководитель группы обеспечения
качества программного обеспечения
ООО «БСЦ МСК» _____ С.С. Смирнова

ВВВЕДЕНИЕ

Методические указания по дисциплине «Разработка клиентской части информационных ресурсов». Дисциплина рассчитана на 2 семестра (4 - 5 семестр).

В методических указаниях представлены лабораторный работы за 4 и 5 семестр. В 4 семестре 20 лабораторных работ. Лабораторные работы рассчитаны на изучение языка JavaScript. В 5 семестре 14 лабораторных работ. Лабораторные работы рассчитаны на изучение клиенской частис помощью фреймворка ReactJS.

Рассматриваемые темы в 4 семестре:

- Настройка окружения JavaScript(Установка Google Chrome, NodeJS, Visual Studio Code. Добавление плагинов в Visual Studio Code для работы с JavaScript. Установка утилиты browser-sync. Проверка работы browser-sync)

- Введение в JavaScript (Добавление динамики в работу HTML страницы, изменение фона при наведении указателя мыши, совершение простых арифметические действий с использованием JavaScript)

- Программирование в JavaScript(Решение простейших задач на программирование (условия, циклы, функции) с использование языка JavaScript)

- Работа со строками в языке JavaScript (Вычисление длины строки Юникода, поиск подстроки в строке с использованием регулярных выражений, валидация введенных пользователем данных)

- Объекты и массивы в JavaScript (Решение задач с использованием встроенных объектов JavaScript (Map, Set, Date), с помощью методов для обработки списков (.map, .filter, .reduce), создание сложных объектов (дескрипторы свойств, Proxy, Reflect))

- Функции в JavaScript (Решение задач на функции и рекурсию, создание набора функций, замкнутых на общую переменную)

- Объекты и прототипы в JavaScript (Решение задач с помощью объектов, создание конструктора объектов с набором общих методов)

- ООП в JavaScript (Решение задач на ООП, написание кода с использованием классов)

- Обработка исключений в JavaScript (Решение задач на обработку исключений)

- Обработка действий пользователя в браузере (Создание веб-страницы, обрабатывающей движение мыши и нажатие кнопок на клавиатуре)

- Асинхронное программирование в JavaScript (Решение задач на асинхронное программирование, промисификация функций, написание циклов и условий с использованием функций с callback-ами и Promise-ов)

- JavaScript анимация (Создание анимации слайдера с помощью JavaScript)

- Генераторы JavaScript (Решение задач с использованием генераторов, создание генератора псевдослучайных чисел)

- Асинхронные итераторы JavaScript (Решение задач с использованием асинхронных генераторов, получение общего значения при частичном вводе данных)

- Разделение JavaScript кода на модули (Создание отдельного модуля с полезными функциями для дальнейшего использования)

- Работа с DOM (Создание простой браузерной мини игры)

- Работа с BOM (Изменение URL текущей страницы, сохранение данных о текущем пути в localStorage)

- AJAX (Работа с удаленным сервером средствами JavaScript, выполнение запросов, загрузка и выгрузка файлов)

- Webpack и npm (Создание JavaScript проекта, установка сторонних модулей, сборка JavaScript модулей в один пакет, публикация собственного модуля в npm репозитории)

- Язык TypeScript (Написание простейших программ (циклы, условия, функции) с использованием языка TypeScript)

По завершению изучения методических указаний к лабораторной работе и самостоятельное выполнения заданий в конце каждой лабораторной работы, будут получены навыки работы с языком JavaScript.

Рассматриваемые темы в 5 семестре:

-Транспайлер кода babel (изучение и работа с транспайлерос babel).

- Введение в React.js (изучение базовых понятий в ReactJS: компоненты, циклы, условия и передача параметров).

- Хуки в ReactJS (изучение и применение хуков useState и useEffect в реальном коде).

- Контекст в ReactJS (изучение и применение хука useContext в реальном коде).

- Списки в ReactJS (изучение асинхронных запросов к серверу, вывод данных полученных из сервера на страницу, добавление пользовательских данных в базу данных, создание пагинации).

- Маршрутизация в ReactJS (настройка маршрутизации в проекте).

- Redux (изучение и настройка Redux 2 способами: legacy redux и redux-toolkit).

- Redux-thunk (построение запросов к серверу с помощью библиотек redux и redux-thunk).

- Redux-saga (изучение Redux-saga и применение знаний на практике).

- Тестирование React-приложений (изучение библиотеки Jest и применение знаний на практике).

- Анимация (создаиние анимации).

- Общедоступные приложения (изучение стандартов по созданию общедоступности приложений, тестирование общедоступности приложение и применение на практике).

- Развертывание веб-приложений (изучение библиотеки webpack-bundle- analyzer, анализ полученного результата и оптимизация веб-приложения).

По завершению изучения методических указаний к лабораторной работе и самостоятельное выполнения заданий в конце каждой лабораторной работы, будут получены навыки работы с фронтом, созданный с помощью ReactJS, интеграции серверной и клиентской части, создания общедоступного приложения и т.д.

1. ЛАБОРАТОРНАЯ РАБОТА №1. Настройка окружения JavaScript.

Цель работы: Установка Google Chrome, NodeJS, Visual Studio Code. Добавление плагинов в Visual Studio Code для работы с JavaScript. Установка утилиты browser-sync. Проверка работы browser-sync.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Google Chrome

Google Chrome — браузер, разрабатываемый компанией Google на основе свободного браузера Chromium и использующий для отображения веб-страниц движок WebKit.

Установка Google Chrome

Для загрузки перейдет на официальный сайт https://www.google.ru/intl/ru/chrome/ . На главной странице мы сразу увидим кнопку скачать. На рисунке 1.1 представлена главная страница.

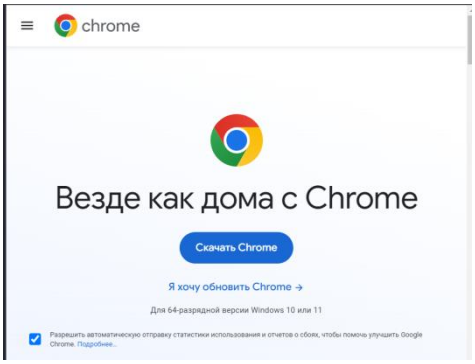

Рисунок 1.1

Node.js

Node (или более формально Node.js) - кросплатформенная среда исполнения с открытым исходным кодом, которая позволяет разработчикам создавать всевозможные серверные инструменты и приложения используя язык JavaScript. Среда исполнения предназначена для использования вне контекста браузера (т.е. выполняется непосредственно на компьютере или на серверной ОС). Таким образом, среда исключает API-интерфейсы JavaScript для браузера и добавляет поддержку более традиционных OS API-интерфейсов, включая библиотеки HTTP и файловых систем.

Установка Node.Js

Для загрузки перейдет на официальный сайт https://nodejs.org/en/. На главной странице мы сразу увидим две возможные опции для загрузки: самая последняя версия NodeJS и LTS-версия. На рисунке 1.2 представлена главная страница.

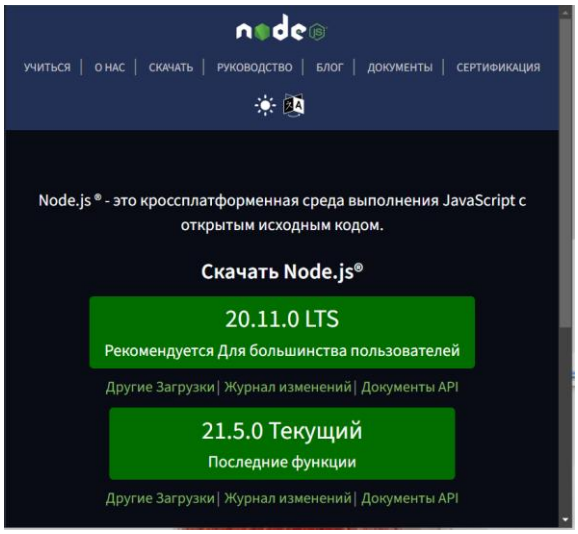

Рисунок 1.2

После установке можно проверить версию node.js можно введя в командную строку node –v. Результат работы представлен на рисунке 1.3.

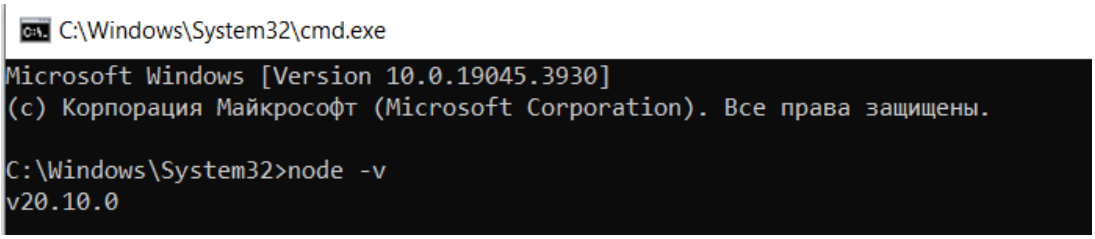

Рисунок 1.3

Visual Studio Code

Visual Studio Code – редактор исходного кода, работанный Microsoft для Windous, Linux и macOS. Позиционируется как «лёгкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений.

Плюсы решения:

а) свободный редактор кода;

б) поддержка сообщества Microsoft;

в) удобная в использовании навигация;

г) много разных плагинов

Минусы решения:

а) требовательна к ресурсам.

Установка Visual Studio Code

Для загрузки перейдет на официальный сайт https://code.visualstudio.com/ . На главной странице мы сразу увидим кнопку скачать. Перед тем как нажать на кнопку выберите вашу ОС. На рисунке 1.4 представлена главная страница.

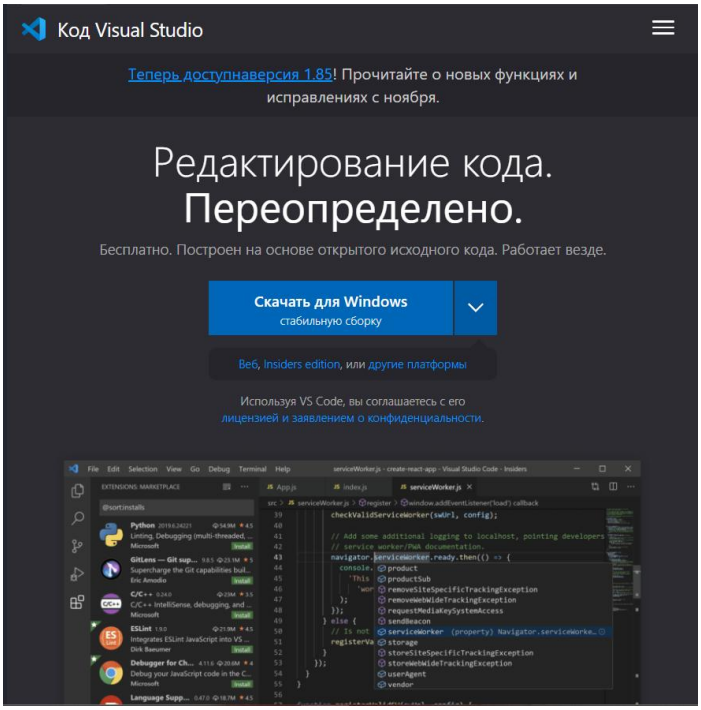

Рисунок 1.4

Рекомендованные расширения

1.ESLint

Торговая площадка - ESLint

Издатель - Microsoft

Легко интегрируйте ESLint в ваш проект. Если ESLint не является вашим любимым linter, выберите одно из множества других расширений linter, включая JSHint, JSCS и JS Standard.

2.SonarLint

Торговая площадка - SonarLint

Издатель - SonarSource

SonarLint помогает находить и исправлять ошибки и проблемы безопасности по мере написания кода. Расширение работает в фоновом режиме и, подобно проверке орфографии,

выявляет проблемы с кодированием. SonarLint не только сообщает вам, в чем проблема, но также предоставляет контекстные рекомендации о том, почему проблема вредна и как ее исправить, с соответствующими примерами. Расширение поддерживает более 200 правил JS / TS и включает несколько быстрых исправлений для автоматического решения ваших проблем с кодированием.

Найдите SonarLint в VS Code Marketplace и установите. Настройка не требуется. Вы можете начать с профиля по умолчанию, который подходит большинству пользователей, и настроить его в соответствии с вашими конкретными потребностями.

3.Фрагменты кода JavaScript (ES6)

Торговая площадка - фрагменты кода JavaScript (ES6)

Издатель - харалампос карипидис

VS Code поставляется со многими встроенными фрагментами кода. Расширение JavaScript (ES6) code snippets добавляет фрагменты для синтаксиса ES6 (ECMAScript 6). Вот небольшая выборка фрагментов, предоставляемых этим расширением. Обратитесь к README расширения, чтобы увидеть десятки фрагментов, которые предоставляет вам этот пакет.

Trigger	Content
`imp→`	imports entire module `import fs from 'fs';`
`imd→`	imports only a portion of the module using destructing `import {rename} from 'fs';`
`ime→`	imports everything as alias from the module `import * as localAlias from 'fs';`
`ima→`	imports only a portion of the module as alias `import { rename as localRename } from 'fs';`
`enf→`	exports name function `export const log = (parameter) => { console.log(parameter);};`
`edf→`	exports default function `export default (parameter) => { console.log(parameter);};`
`ecl→`	exports default class `export default class Calculator { };`
`ece→`	exports default class by extending a base one `export default class Calculator extends BaseClass { };`

Рисунок 1.5

Вы можете прочитать больше о фрагментах JavaScript в документации по VS Code. Для получения дополнительных пакетов сниппетов, включая Angular 1, Angular 2, Bootstrap 3, ReactJS и jQuery, ознакомьтесь с категорией сниппетов торговой площадки.

4.npm IntelliSense

Торговая площадка - npm IntelliSense

Издатель - Кристиан Колер

Это расширение обеспечивает IntelliSense для модулей npm при использовании import или require.

Утилита browser-sync

Browsersync - это модуль для Node.js, платформа для быстрых сетевых приложений. Существуют удобные установщики для macOS, Windows и Linux. *Установка Browsersync*

В Node.js менеджер пакетов (НПМ) используется, чтобы установить browsersync от хранилища. Откройте окно терминала и выполните следующую команду.

Листинг кода 1.1:

```
npm install -g browser-sync
```

Вы предлагаете менеджеру пакетов загрузить файлы Browsersync и установить их глобально, чтобы они были доступны во всех ваших проектах.

Результат работы представлена на рисунке 1.6.

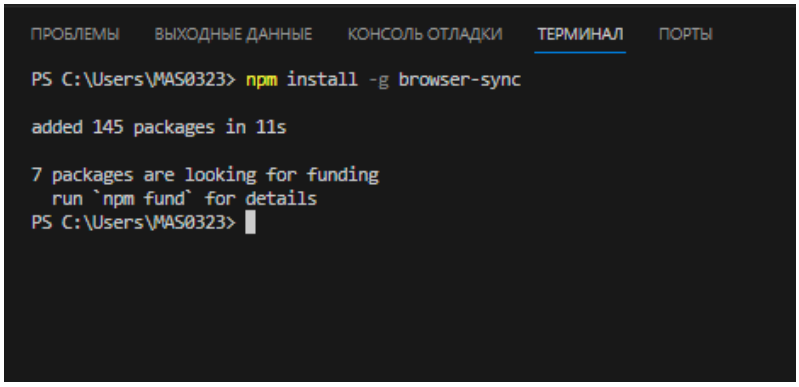

Рисунок 1.6

Запуск Browsersync

Основное применение - просмотр всех файлов CSS в css каталоге и обновление подключенных браузеров в случае изменения. Перейдите в окно терминала к проекту и запустите соответствующую команду:

Статические сайты

Если вы используете только .html файлы, вам нужно будет использовать серверный режим. Browsersync запустит мини-сервер и предоставит URL-адрес для просмотра вашего

сайта.

Листинг кода 1.2:

```
browser-sync start --server --files "*"
```

Динамические сайты

Если вы уже используете локальный сервер с PHP или аналогичный, вам нужно будет использовать режим прокси. Browsersync добавит к вашему vhost URL прокси-адрес для просмотра вашего сайта.

Листинг кода 1.3:

```
browser-sync start --proxy "myproject.dev" --files "*"
```

После запуска команда откроется браузер. Результат работы представлен на рисунке 1.7.

Рисунок 1.7

Теперь мы можем тестировать сайт без перезагрузки страницы, т.к. Browsersync автоматически перезагружает страницу после каждого изменения кода.

Теперь немного пояснения что у нас в терминале (рисунок 1.8).

Первая пара. Первый URL можно использовать для синхронизации браузеров на локальной машине. Второй URL — для доступа к серверу с других устройств в локальной сети. Просто открываем URL в браузере — и все работает.

Вторая пара URL нужна для перехода к настройкам синхронизации — с локальной машины или с других устройств в локальной сети. По поводу синхронизации всех браузеров — настраивается все здесь: переход по ссылкам, заполнение полей формы, скроллинг.

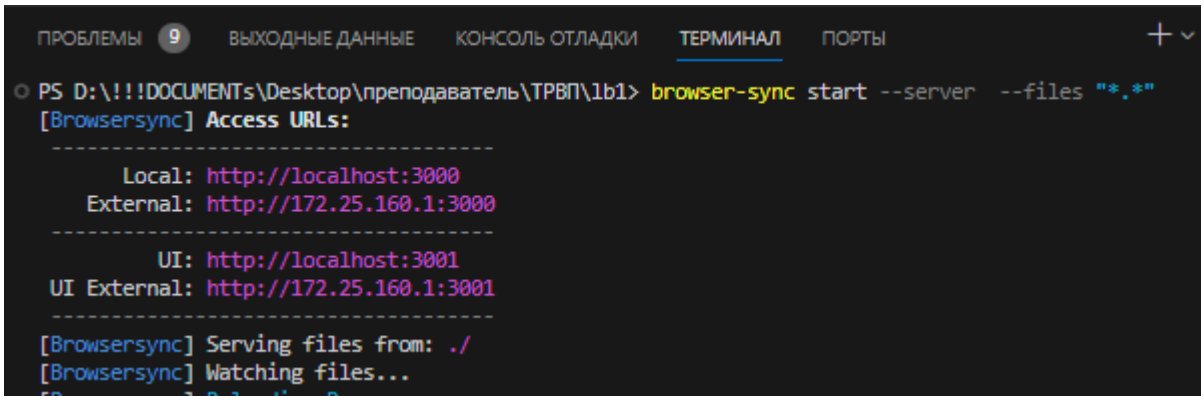

Рисунок 1.8

Задание для выполнения:

1. Ознакомиться с материалом методического указания, выполните примеры из методички.

2. Установить Google Chrome, NodeJS, Visual Studio Code.

3. Добавить плагины в Visual Studio Code для работы с JavaScript.

4. Установите утилиту browser-sync.

5. Проверить работу browser-sync для этого создать проект, где будут находится несколько файлов с разрешение .css и .html с любым кодом.

6. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое Google Chrome?

2. Что такое NodeJS?

3. Что такое Visual Studio Code?

4. Какие плагины в Visual Studio Code для работы с JavaScript вам известны и для чего они нужны?

5. Что такое утилита browser-sync и для чего она нужна?

2. ЛАБОРАТОРНАЯ РАБОТА №2. Введение в JavaScript.

Цель работы: Добавление динамики в работу HTML страницы, изменение фона при наведении указателя мыши, совершение простых арифметические действий с использованием JavaScript.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Что такое JavaScript?

JavaScript это язык, который позволяет применять сложные вещи на веб странице — каждый раз, когда на веб странице происходит что-то большее, чем просто её статичное отображение — отображение периодически обновляемого контента, или интерактивных карт, или анимация 2D/3D графики, или прокрутка видео в проигрывателе, и т.д. — можете быть уверены, что скорее всего, не обошлось без JavaScript.

Программы на этом языке называются *скриптами*. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы.

Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

Сегодня JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся «движком» JavaScript.

У браузера есть собственный движок, который иногда называют «виртуальная машина JavaScript».

Подключение js-кода

Файл скрипта можно подключить к HTML с помощью атрибута src:

Листинг кода 2.1:

```
<script src="/path/to/script.js"></script>
```

Здесь, /path/to/script.js – это абсолютный путь до скрипта от корня сайта. Также можно указать относительный путь от текущей страницы. Например, src="script.js" будет означать, что файл "script.js" находится в текущей папке.

Можно указать и полный URL-адрес.

Листинг кода 2.2:

```
	<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js
"></script>
```

Для подключения нескольких скриптов используйте несколько тегов.

Листинг кода 2.3:

```
	<script src="/js/script1.js"></script>
	<script src="/js/script2.js"></script>
```

Так же можно писать сам код в html.

Листинг кода 2.4:

```
	<script>
		alert("Hello")
	</script>
```

Пример «Hello, world!»

Создаем страницу index.html. Добавляем код.

Листинг кода 2.5:

```
	<!DOCTYPE html>
	<html>

	<head>
	    <title>Пример</title>
	</head>

	<body>
	    <script>
	        alert("Hello")
	    </script>
	</body>
```

```
</html>
```

Запускаем страницу. Результат работы представлен ниже на рисунке 2.1.

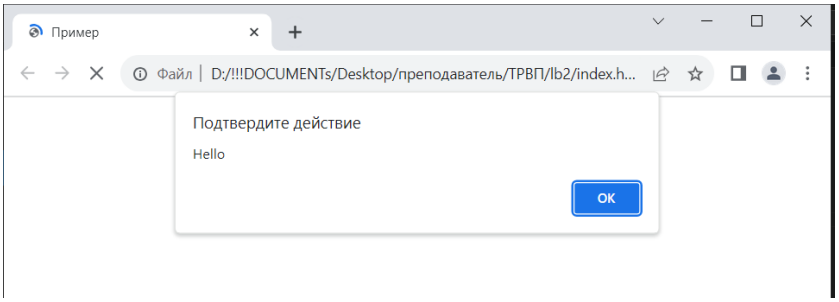

Рисунок 2.1

Переменные

Переменная – это «именованное хранилище» для данных.

Для создания переменной в JavaScript, используйте ключевое слово let.

Приведённая ниже инструкция создаёт (другими словами: объявляет или определяет) переменную с именем «message».

Листинг кода 2.5:

```
let message;
```

Теперь можно поместить в неё данные, используя оператор присваивания =.

Листинг кода 2.6:

```
let message;
message = 'Hello'; // сохранить строку
```

Строка сохраняется в области памяти, связанной с переменной. Можно получить к ней доступ, используя имя переменной.

Листинг кода 2.7:

```
let message;
message = 'Hello!';
alert(message); // показывает содержимое переменной
```

Типы данных

1. Число

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой.

Существует множество операций для чисел, например, умножение *, деление /, сложение +, вычитание - и так далее.

Листинг кода 2.7:

```
let n = 123;
n = 12.345;
```

2. Строка

Строка (`string`) в JavaScript должна быть заключена в кавычки.

Листинг кода 2.8:

```
let str = "Привет";
let str2 = 'Одинарные кавычки тоже подойдут';
let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;
```

В JavaScript существует три типа кавычек.

- Двойные кавычки: "Привет".

- Одинарные кавычки: 'Привет'.

- Обратные кавычки: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные кавычки же имеют «расширенный функционал». Они позволяют встраивать выражения в строку, заключая их в ${…}.

Листинг кода 2.9:

```
let name = "Иван";
// Вставим переменную
alert( `Привет, ${name}!` ); // Привет, Иван!
```

```
// Вставим выражение
alert( `результат: ${1 + 2}` ); // результат: 3
```

Выражение внутри ${…} вычисляется, и его результат становится частью строки.

3. Булевый (логический) тип

Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь).

Листинг кода 2.10:

```
let nameFieldChecked = true; // да, поле отмечено
let ageFieldChecked = false; // нет, поле не отмечено
```

4. Объекты и символы

Тип object (объект) – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка или число, или что-то ещё). Объекты же используются для хранения коллекций данных или более сложных объектов.

Тип symbol (символ) используется для создания уникальных идентификаторов объектов.

Операторы

Арифметические операторы-это операторы, используемые для выполнения математических операций, таких как сложение, вычитание, умножение и деление.

Инкреме́нт, инкрементирование (от англ. increment «увеличение») — операция во многих языках программирования, увеличивающая значение переменной. Обратную операцию называют декремент (уменьшение). Чаще всего унарная операция приводит значение переменной к следующему элементу базового типа (то есть для целых чисел — увеличивает на 1, для символьного типа даёт следующий символ в некоторой таблице символов и т. п.).

Логические операторы – это операторы, которые принимают в качестве аргументов логические значений (ложь или истину) и возвращают логическое значение. Как и обычные операторы, они могут быть одноместными (унарными, т.е. принимать один аргумент), двуместными (бинарные, принимают два аргумента), трёхместными и т.д.

Ниже в таблица 2.1-2.4 представлены операторы в js.

Таблица 2.1 – Арифметические операторы

Оператор	Описание	Пример	Решение
+	Выполняет сложение чисел	y=x+10	y=17
-	Выполняет вычитание чисел	y=x-3	y=4
*	Выполняет умножение чисел	y=x*4	y=28
/	Выполняет деление чисел	y=x/2	y=3
%	Вычисляет остаток от деления чисел	y=x%2	y=1

Таблица 2.2 – Инкремент и декремент

Оператор	Форма оператора	Описание	Пример (x = 1)	Решение
++x	Префиксная	Увеличит x на 1, затем использует его	y = ++x * 5 y = 2 * 5	x = 2 y = 10
x++	Постфиксная	Использует x, затем увеличит его на 1	y = x++ * 5 y = 1 * 5	x = 2 y = 5
--x	Префиксная	Уменьшит x на 1, затем использует его	y = --x * 5 y = 0 * 5	x = 0 y = 0
x--	Постфиксная	Использует x, затем уменьшит его на 1	y = x-- * 5 y = 1 * 5	x = 0 y = 5

Таблица 2.3 – Операторы сравнения

Оператор	Описание	Условие	Пример	Решение
==	Равенство	true, если операнды равны между собой, даже если у них разные типы	4==4	true
!=	Неравенство	true, если операнды НЕ равны между собой, даже после приведения к одному типу	4 != -4.4	true

Продолжение таблицы 2.3 – Операторы сравнения

Оператор	Описание	Условие	Пример	Решение
===	Идентичность	true, если операнды равны между собой и у них одинаковые типы	4 === “4”	false
!==	Не идентичность	true, если операнды НЕ равны между собой, не приводя к одному типу	1 !== true	true
>	Больше	true, если первый операнд строго больше второго	3>4	false
>=	Больше или равно	true, если первый операнд больше второго или равен ему	3>=3	true
<	Меньше	true, если первый операнд строго меньше второго	3<4	true
<=	Меньше или равно	true, если первый операнд строго меньше второго или равен ему	3<=4	false

Таблица 2.4 – Логически операторы

Оператор	Описание	Условие
&&	Логическое «И»	бинарный оператор; возвращает true, если оба операнда являются true;
||	Логическое «ИЛИ»	бинарный оператор; возвращает true, если хотя бы один оператор является true;
!	Логическое «НЕ»	унарный оператор; возвращает true, если значение равно false. Возвращает false, если значение равно true.

Динамики в HTML

Для добавления динамики в HTML с помощью JavaScript. Вот несколько способов добавления динамического поведения на страницу:

- Использование событий: вы можете добавить обработчики событий к HTML-элементам, чтобы на них реагировать.

- Изменение содержимого элементов: вы можете использовать JavaScript для изменения содержимого HTML-элементов.

- Изменение стилей элементов: вы можете использовать JavaScript для изменения стилей HTML-элементов.

- Анимация: JavaScript также может использоваться для создания анимаций на странице.

Браузер создаёт DOM при загрузке страницы, складывает его в переменную document и сообщает, что DOM создан, с помощью события DOMContentLoaded. С переменной document начинается любая работа с HTML-разметкой в JavaScript.

Объект document содержит большое количество свойств и методов, которые позволяют работать с HTML. Чаще всего используются методы, позволяющие найти элементы страницы.

- .title — заголовок документа. Браузер обычно показывает его на вкладке.

- .forms — получить список форм на странице. Свойство только для чтения, напрямую перезаписать его нельзя.

- .body — получить <body> элемент страницы.

- .head — получить <head> элемент страницы.

Методы:

- .getElementById — поиск элемента по идентификатору;

- .getElementsByClassName — поиск элементов по названию класса;

- .getElementsByTagName — поиск элементов по названию тега;

- .querySelector — поиск первого элемента, подходящего под CSS-селектор;

- .querySelectorAll — поиск всех элементов подходящих под CSS-селектор.

Элемент — это кусочек HTML в DOM-дереве. Браузер создаёт DOM для взаимодействия между JavaScript и HTML. Каждый HTML-тег при этом превращается в элемент DOM. Ещё такие элементы называют узлами.

Из DOM можно получить элемент и изменить его. Браузер заметит изменения и отобразит их на странице.

Например, можно поменять выравнивание у элемента h1.

Листинг кода 2.11:

```
	// получаем элемент из DOM
	const element = document.getElementsByTagName('h1')[0]
	// после выполнения этого кода h1 будет выравнивать текст по центру
```

```
element.align = 'center'
// меняем цвет шрифта на красный
element.style.color = 'red'
```

HTML-элементы содержат свойства, которые можно разделить на группы:

– свойства, связанные с HTML-атрибутами: id, классы, стили и так далее;

– свойства и методы связанные с обходом DOM: получение дочерних элементов, родителя, соседей;

– информация о содержимом;

– специфические свойства элемента.

Первые три группы свойств есть у всех элементов. Специфические свойства отличаются в зависимости от типа элемента. Например, у полей ввода есть свойства, которых нет у параграфов и блоков: value, type и другие.

Читать и записывать значения HTML-атрибутов можно при помощи свойств элемента. Название обычно совпадает с названием атрибута:

– .id — получить идентификатор элемента.

– .className — список классов в HTML-атрибуте class.

– .style — добавить стили. Стили добавляются так же с помощью свойств. Свойства именуются по аналогии с CSS-свойствами.

Свойства и методы, связанные с DOM:

– .children — список дочерних элементов;

– .parentElement — получить родительский элемент;

– .nextElementSibling и previousElementSibling — получить следующий/предыдущий узел-сосед:

– .getElementsByClassName() — поиск среди дочерних элементов по названию класса;

– .getElementsByTagName() — поиск среди дочерних элементов по названию тега;

– .querySelector() — поиск первого дочернего элемента, подходящего под CSS-селектор;

– .querySelectorAll() — поиск всех дочерних элементов подходящих под CSS-селектор;

С помощью этих свойств и методов можно перемещаться по дереву и даже обойти все его элементы, если это нужно.

Свойства с информацией о содержимом:

– .innerHTML — это свойство возвращает HTML-код всего, что вложено в текущий элемент. При записи в это свойство, предыдущее содержимое будет затёрто.

– .outerHTML — это свойство возвращает HTML-код текущего элемента и всего, что в него вложено. При записи в это свойство, предыдущее содержимое будет затёрто.

– .textContent — свойство, возвращает текст всех вложенных узлов без HTML-тегов.

События

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

События мыши:

– click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).

– contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.

– mouseover / mouseout – когда мышь наводится на / покидает элемент.

– mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.

– mousemove – при движении мыши.

События на элементах управления:

– submit – пользователь отправил форму <form>.

– focus – пользователь фокусируется на элементе, например нажимает на <input>.

Клавиатурные события:

– keydown и keyup – когда пользователь нажимает / отпускает клавишу.

События документа:

– DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

– transitionend – когда CSS-анимация завершена.

Обработка событий

- Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Листинг кода 2.12:

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

- Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

Листинг кода 2.13:

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Кролик номер " + i);
    }
  }
</script>
<input type="button" onclick="countRabbits()" value="Считать кроликов!">
```

- Использование свойства DOM-объекта. Можно назначать обработчик, используя свойство DOM-элемента on<событие>.

Листинг кода 2.14:

```
<input id="elem" type="button" value="Нажми меня!">
<script>
  elem.onclick = function() {
    alert('Спасибо');
  };
</script>
```

Также можно добавить уже существующаю функцию.

Листинг кода 2.15:

```
function sayThanks() {
  alert('Спасибо!');
}
```

```
elem.onclick = sayThanks;
```

Частая ошибка писать!!

Листинг кода 2.16:

```
// правильно
button.onclick = sayThanks;
// неправильно
button.onclick = sayThanks();
```

Важно!!

- Регистр имеет значение(писать elem.onclick,а не elem.ONCLICK)
- Используйте именно функции, а не строки.
- Не используйте setAttribute для обработчиков.

addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие.

Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение.

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener. Они свободны от указанного недостатка.

Синтаксис добавления обработчика.

Листинг кода 2.17:

```
element.addEventListener(event, handler, [options]);
```

event - Имя события, например "click".

handler -Ссылка на функцию-обработчик.

Реализация:

Создали проект в нем создаем 3 файла: index.html, script.js и style.css. В них добавляем код для:

1.Добавление динамики в работу HTML страницы, изменение фона при наведении указателя мыши.

При наведении на строку появляется красный фон фон, когда убираешь мышь фон исчезает.

Листинг кода 2.18 (index.html):

```
<!DOCTYPE html>
<html>
<head>
    <title>Dynamika</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
    <div id="myDiv">Это див</div>

    <script src="script.js"></script>
</body>
</html>
```

Листинг кода 2.19 (script.js):

```
var myDiv = document.getElementById("myDiv");

// Добавления действие элемента с id=myDiv по наведению на элемент с id=myDiv
myDiv.addEventListener("mouseover", function() {
    myDiv.classList.add("hover-background");
});
// Удаления действие элемента с id=myDiv, когда мышь "ушла" с элемент с id=myDiv
myDiv.addEventListener("mouseout", function() {
    myDiv.classList.remove("hover-background");
});
```

Листинг кода 2.20 (style.css):

```
body {
    background-color: white;
}
.hover-background {
    background-color: red;
}
```

2. Считавания данные с формы и совершение простых арифметические действий с использованием JavaScript.

После заполнения полей необходимо нажать на кнопку и к элементу с id= result добавил значения суммы 2 чисел.

Добавим код в файлы.

Листинг кода 2.21(index.html):

```
…
<input type="number" id="num1" placeholder="Число 1">
    <input type="number" id="num2" placeholder="Число 2">
    <button onclick="performCalculation()">Посчитать</button>
    <p id="result"></p>
…
```

Листинг кода 2.23 (index.html):

```
…
function performCalculation() {
    var num1 = parseInt(document.getElementById("num1").value);
    var num2 = parseInt(document.getElementById("num2").value);
    var result = num1 + num2;
    document.getElementById("result").innerHTML = "Результат: "
+ result;
}
```

Результат работы представлен на рисунке 2.2.

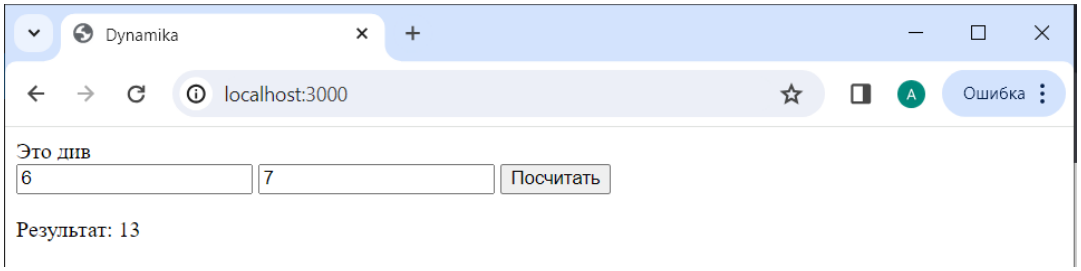

Рисунок 2.2 – результат работы

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1. Ознакомиться с материалом методического указания, выполните примеры из методички.

2. Создать одностраничный сайт по варианту (№варианта=№ в журнале). На сайте должны осуществляется действия:

- изменение фона при наведении указателя мыши;

-считываться данные с формы и совершаться простые арифметические действий с использованием JavaScript.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое JS?
2. Какие типы данных есть и как они объявляются?
3. Какие есть операторы?

4. Как осуществляется динамика в HTML с помощью JS?

5. Что такое «Элемент»?

6. Что такое «Событие»?

7. Как обрабатываются события?

3. ЛАБОРАТОРНАЯ РАБОТА №3. Программирование в JavaScript.

Цель работы: Решение простейших задач на программирование (условия, циклы, функции) с использование языка JavaScript.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Условные операторы

1. Сложение строк

Оператор if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода.

Листинг кода 3.1:

```
      let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');
      if (year == 2015) alert( 'Вы правы!' );
```

2. Блок «else»

Оператор if может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно.

Листинг кода 3.2:

```
      let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');
      if (year == 2015) {
        alert( 'Да вы знаток!' );
      } else {
        alert( 'А вот и неправильно!' ); // любое значение, кроме 2015
      }
```

Иногда, нам нужно назначить переменную в зависимости от условия.

Листинг кода 3.3:

```
let accessAllowed;
let age = prompt('Сколько вам лет?', '');
if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
alert(accessAllowed);
```

3. Условный оператор «?» (тернарный оператор)

Оператор представлен знаком вопроса ?. Его также называют «тернарный», так как этот оператор, единственный в своём роде, имеет три аргумента.

Листинг кода 3.4:

```
let result = условие ? значение1 : значение2;
```

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2.

Листинг кода 3.5:

```
let accessAllowed = (age > 18) ? true : false;
```

Циклы

1. Цикл while

Цикл while имеет следующий синтаксис.

Листинг кода 3.6:

```
while (condition) {
  // код
  // также называемый "телом цикла"
}
```

Код из тела цикла выполняется, пока условие condition истинно.

Например, цикл ниже выводит i, пока i < 3.

Листинг кода 3.7:

```
let i = 0;
while (i < 3) { // выводит 0, затем 1, затем 2
  alert( i );
  i++;
}
```

Одно выполнение тела цикла по-научному называется итерация. Цикл в примере выше совершает три итерации.

2. Цикл «do…while»

Проверку условия можно разместить под телом цикла, используя специальный синтаксис do..while.

Листинг кода 3.8:

```
do {
  // тело цикла
} while (condition);
```

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова.

Листинг кода 3.9:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

3. Цикл «for»

Более сложный, но при этом самый распространённый цикл — цикл for.

Выглядит он так.

Листинг кода 3.10:

```
for (начало; условие; шаг) {
  // ... тело цикла ...
}
```

Разберёмся, что означает каждая часть, на примере. Цикл ниже выполняет alert(i) для i от 0 до (но не включая) 3.

Листинг кода 3.11:

```
for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2
  alert(i);
}
```

Функции

Функция — это самостоятельный блок кода, который можно, один раз объявив, вызывать столько раз, сколько нужно. Функция может, хотя это и необязательно, принимать параметры. Функции возвращают единственное значение.

Встроенные функции, т.к. alert(message), prompt(message, default) и confirm(question).

Объявление функции.

Чтобы создать функцию ее нужно объявить.

Листинг кода 3.12:

```
function имя(параметры) {
  ...тело...
}
```

Например, объявим функции showMessage() без парматров и в теле пропишем, что нужно вывести сообщение 'Всем привет!' на экран, далее мы вызовем несколько раз эту функцию.

Листинг кода 3.13:

```
function showMessage() {
  alert( 'Всем привет!' );
}
```

```
showMessage();
showMessage();
```

В результате мы увидим что сообщение вывелось 2 раза, т.к. функцию вызвали 2 раза.

Переменные

1. Локальные – объявляется в теле функции

2. Внешняя – объявляется вне тела функции.

Параметры

- Параметр – это переменная, указанная в круглых скобках в объявлении функции.
- Аргумент – это значение, которое передаётся функции при её вызове.

1. Параметры можно задать можно задать при вызове функции.

Листинг кода 3.14:

```
function showMessage(from, text) { // параметры: from, text
  alert(from + ': ' + text);
}
showMessage('Мир', 'Привет!'); // Мир: Привет! (*)
```

2. Параметры также можно задать по умолчанию

Листинг кода 3.15:

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}
showMessage("Мир"); // Мир: текст не добавлен
```

Возврат значения

Чтобы вернуть значение необходимо в теле функции прописать слово return.

Листинг кода 3.16:

```
function sum(a, b) {
```

```
  return a + b;
}
let result = sum(3, 2);
alert( result ); // 5
```

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1. Ознакомиться с материалом методического указания, выполните примеры из методички.

2. Создать одностраничный сайт по варианту (№варианта=№ в журнале). На сайте должны осуществляется действия:

- используя переменны, которые вы считаете с формы, и уловные операторы (“if” и “?”) произвести с ними какое-либо действие;
- используя циклы(while…do, do…while, for) считать 2 любых интервала и вывести на этом промежутки любые числа равных какому-либо условию;
- используя функции задать переменные (локально и внутренне) произвести какое-либо действие и вернуть значение.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое условные операторы?

2. Какие условные операторы вам известны?

3. Что такое циклы?

4. Какие циклы вам известны?

3. Какие есть операторы?

5. Что такое функция?

6. Как можно объявить переменную в функции?

4. ЛАБОРАТОРНАЯ РАБОТА №4. Работа со строками в языке JavaScript

Цель работы: Вычисление длины строки Юникода, поиск подстроки в строке с использованием регулярных выражений, валидация введенных пользователем данных.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Строки Юникода

Строки в JavaScript основаны на Юникоде: каждый символ представляет из себя последовательность байтов из 1-4 байтов.

JavaScript позволяет нам вставить символ в строку, указав его шестнадцатеричный Юникод с помощью одной из этих трех нотаций:

1.\xXX- XX должен указать 2 шестеричные цифры со значением от 00 до FF. Можно использовать только для первых 256 символов Юникода. Эти 256 символов включают в себя латинский алфавит, большинство основных синтаксических символов и некоторые другие.

Листинг кода 4.1:

```
alert( "\x7A" ); // z
alert( "\xA9" ); // ©, символ авторского права
```

2. \uXXXX - XXXX должен указать 4 шестеричные цифры со значением от 0000 до FFFF. Символы со значениями Юникода, превышающими U+FFFF, также могут быть представлены с помощью этой нотации, но в таком случае нам придется использовать так называемую суррогатную пару (о ней мы поговорим позже в этой главе).

Листинг кода 4.2:

```
alert( "\u00A9" ); // ©, то же самое, что \xA9, используя 4-значную шестнадцатеричную нотацию
alert( "\u044F" ); // я, буква кириллического алфавита
```

3. \u{X…XXXXXX}- X…XXXXXX должно быть шестнадцатеричное значение от 1 до 6 байт от 0 до 10FFFF (максимальная точка кода, определенная стандартом Юникод). Эта нотация позволяет нам легко представлять все существующие символы Юникода.

Листинг кода 4.3:

```
alert( "\u{20331}" ); // 佱, редкий китайский иероглиф (длинный Юникод)
alert( "\u{1F60D}" ); // 😍, символ улыбающегося лица (ещё один длинный Юникод)
```

Вычислить длину можно с помощью метода “строка”.length.

Длина строки юникода = количество байтов в строке.

Но сначала нужно нормализовать стрику методом : .normalize().

Листинг кода 4.4:

```
alert( 'u044F'.normalize().length );
```

Поиск подстроки

Первый метод – “строка”.indexOf(substr, pos).

Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений.

Листинг кода 4.5:

```
let str = 'Hello ell';
//поиск первого вхождения
alert( str.indexOf('Hello') ); // 0, потому что подстрока 'Hello' найдена в начале
alert( str.indexOf('hello ') ); // -1, совпадений нет, поиск чувствителен к регистру
alert( str.indexOf("el") ); // 1, подстрока "el" найдена на позиции 1 (…ello ell)
alert( str.indexOf("el", 2) ); // 6
//поиск последнего вхождения
alert(str.lastIndexOf("el"))
//поиск всех вхождения
let pos = -1;
while ((pos = str.indexOf("el", pos + 1)) != -1) {
  alert( pos );
```

```
}
```

Второй метод – “строка”. includes (substr, pos).

Он ищет подстроку substr в строке str, начиная с позиции pos, и в озвращает true, если в строке str есть подстрока substr, либо false, если нет.

Листинг кода 4.6:

```
alert( "Hello el".includes("Hello") ); // true
alert( "Hello".includes("Bye") ); // false
alert( " Hello el ".includes("id", 6) ); // false, поиск начат с позиции 6
```

Третий метод - “строка”. startsWith (substr) и “строка”. endsWith (substr).

.startsWith выводит true ели начинается с substr, либо false, если нет.

.endsWith (substr) выводит true ели заканчивается на substr, либо false, если нет.

Листинг кода 4.7:

```
alert( "Hello".startsWith("He") ); // true, "He" – начало "Hello"
alert( "Hello".endsWith("lo") ); // true, "lo" – окончание "Hello"
```

Получение подстроки

Первый метод – “строка”. slice(start [, end])

Возвращает часть строки от start до (не включая) end.

Листинг кода 4.8:

```
let str = "Hello ell";
// 'Hello', символы от 0 до 5 (не включая 5)
alert( str.slice(0, 5) );
// 'H', от 0 до 1, не включая 1, т. е. только один символ на позиции 0
alert( str.slice(0, 1) );
// llo ell, с позиции 2 и до конца
```

```
      alert( str.slice(2) );
      // 'el', начинаем с позиции 3 справа, а заканчиваем на позиции
1 справа
      alert( str.slice(-3, -1) );
```

Второй метод – “строка”. substring (start [, end])

Возвращает часть строки от start до (не включая) end. Это — почти то же, что и slice, но можно задавать start больше end.

Если start больше end, то метод substring сработает так, как если бы аргументы были поменяны местами. Отрицательные значения substring, в отличие от slice, не поддерживает, они интерпретируются как 0.

Листинг кода 4.9:

```
      let str = "Hello ell";
      // для substring эти два примера — одинаковы
      alert( str.substring(2, 6) ); // "llo e"
      alert( str.substring(6, 2) ); // "llo e"
      // …но не для slice:
      alert( str.slice(2, 6) ); // "llo e" (то же самое)
      alert( str.slice(6, 2) ); // "" (пустая строка)
```

Третий метод – “строка”. substr(start [, length])

Возвращает часть строки от start длины length.

Листинг кода 4.9:

```
      let str = "Hello ell";
      // "llo e", получаем 4 символа, начиная с позиции 2
      alert( str.substr(2, 4) );
      // ‘el’, получаем 2 символа, начиная с позиции 4 с конца строки
      alert( str.substr(-3, 2) );
```

Валидация форм

Валидация – проверка вводимых данных на какое-либо условие (длина, вводимые символы, тип и т.д.)

Например, с помощью функции можно проверить пуста ли строка.

Листинг кода 4.10:

```
function validateForm() {
    var x = document.forms["myForm"]["fname"].value;
    if (x == "") {
        alert("Необходимо ввести имя");
        return false;
    }
}
…
<form name="myForm" onsubmit="return validateForm()">
 Имя: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

Автоматическая проверка форм используется атрибут required.

Листинг кода 4.11 (HTML):

```
<form>
   <input type="text" name="fname" required>
       <button>Кнопка</button></form>
```

Листинг кода 4.12 (CSS):

```
input:invalid {
  border: 2px solid red;
}
input:valid {
  border: 2px solid green;
}
```

В HTML5 была добавлена новая концепция HTML проверки, название которой можно было бы перевести как "ограничивающая проверка" (англ. Constraint Validation).

Ограничивающая проверка HTML основывается на:

- HTML атрибутах элемента ввода (таблица 4.1),

- CSS псевдоселекторах (таблица 4.2),

- свойствах и методах DOM (таблица 4.3)

Таблица 4.1- Атрибуты ограничивающей проверки элемента ввода

Атрибут	Описание
disabled	Определяет, что элемент ввода отключен
max	Определяет максимальное значение в элементе ввода
min	Определяет минимальное значение в элементе ввода
pattern	Определяет шаблон значений в элементе ввода
required	Определяет, что элемент ввода обязателен для заполнения
type	Определяет тип элемента ввода

Таблица 4.2- CSS псевдоселекторы ограничивающей проверки

Селектор	Описание
Селектор	Описание
:disabled	Выбирает элемент ввода с атрибутом "disabled"
:invalid	Выбирает элемент ввода с некорректным значением
:optional	Выбирает элемент ввода без атрибута "required"
:required	Выбирает элемент ввода с атрибутом "required"
:valid	Выбирает элемент ввода с корректным значением

Таблица 4.3- Методы DOM ограничивающей проверки

Метод	Описание
checkValidity()	Возвращает true, если элемент ввода содержит корректные данные
setCustomValidity()	Устанавливает свойство validationMessage для элемента ввода

Ниже представлен вывод сообщения если ввели если мы ввели число больше 300 или меньше 100.

Листинг кода 4.13:

```
      <input id="id1" type="number" min="100" max="300" required>
        <button onclick="myFunction()">OK</button>

        <p id="demo"></p>

        <script>
         function myFunction() {
            var inpObj = document.getElementById("id1");
            if (!inpObj.checkValidity()) {
                        document.getElementById("demo").innerHTML   =
inpObj.validationMessage;
            }
        }
        </script>
```

В таблице 4.4 представлены свойства DOM ограничивающей проверки.

Таблица 4.4- Свойства DOM ограничивающей проверки

Свойство	Описание
validity	Содержит набор свойств типа boolean, определяющих состояние валидности (правильности) элемента ввода
validationMessage	Содержит сообщение, которое выведет браузер в случае неудачной проверки
willValidate	Указывает, будет ли элемент ввода проверяться

Свойство validity элемента ввода содержит некий набор свойств, определяющих состояние валидности данных (таблица 4.5)

Таблица 4.5- Свойства валидности

Свойство	Описание
customError	Установлено в true, если задано пользовательское сообщение о валидности.
patternMismatch	Установлено в true, если значение элемента не соответствует шаблону в атрибуте *pattern*.

Свойство	Описание
rangeOverflow	Установлено в true, если значение элемента больше значения в атрибуте *max*.
rangeUnderflow	Установлено в true, если значение элемента меньше значения в атрибуте *min*.
stepMismatch	Установлено в true, если значение элемента неверно по атрибуту *step*.
tooLong	Установлено в true, если значение элемента превышает значение атрибута *maxLength*.
typeMismatch	Установлено в true, если значение элемента неверно по атрибуту *type*.
valueMissing	Установлено в true, если у элемента (с атрибутом *required*) нет значения.
valid	Установлено в true, если значение элемента валидно.

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1. Ознакомиться с материалом методического указания.

2. Создать одностраничный сайт по варианту (№варианта=№ в журнале). На сайте должны осуществляется действия:

- вычисление длины строки Юникода,

-поиск подстроки в строке с использованием регулярных выражений,

-валидация формы (минимум 3 поля ввода) введенных пользователем данных.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое Юникод?
2. Какие найти подстроку в строку?
3. Как вывести подстроку в строку?
4. Что такое валидация?
5. Какие вам известны атрибуты ограничивающей проверки элемента ввода?
6. Какие вам известны CSS псевдоселекторы ограничивающей проверки?
7. Какие вам известны методы DOM ограничивающей проверки?
8. Какие вам известны свойства DOM ограничивающей проверки?
9. Какие вам известны свойства валидности?

5. ЛАБОРАТОРНАЯ РАБОТА №5. Объекты и массивы в JavaScript.

Цель работы: Решение задач с использованием встроенных объектов JavaScript (Map, Set, Date), с помощью методов для обработки списков (.map, .filter, .reduce), создание сложных объектов (дескрипторы свойств, Proxy, Reflect).

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Встроенные объекты(Map, Set, Date)

Map

Map – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

– new Map() – создаёт коллекцию.

– map.set(key, value) – записывает по ключу key значение value.

– map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.

– map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.

– map.delete(key) – удаляет элемент (пару «ключ/значение») по ключу key.

– map.clear() – очищает коллекцию от всех элементов.

– map.size – возвращает текущее количество элементов.

Листинг кода 5.1:

```
let map = new Map();//создаем коллекцию
map.set("1", "string1");    // строка в качестве ключа
map.set(1, "number1");      // цифра как ключ
map.set(true, "boolean1");  // булево значение как ключ

// помните, обычный объект Object приводит ключи к строкам?
// Map сохраняет тип ключей, так что в этом случае сохранится 2 разных значения:
alert(map.get(1)); // "string1"
```

```
alert(map.get("1")); // "number1"
alert(map.size); // 3
```

Каждый вызов map.set возвращает объект map, так что мы можем объединить вызовы в цепочку.

Листинг кода 5.2:

```
   map.set("1", "string1");    // строка в качестве ключа
         .set(1, "number1");       // цифра как ключ
         .set(true, "boolean1");  // булево значение как ключ
```

Для перебора коллекции Map есть 3 метода:

– map.keys() – возвращает итерируемый объект по ключам,

– map.values() – возвращает итерируемый объект по значениям,

– map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

Листинг кода 5.3:

```
  let recipeMap = new Map([
  ["яблоко", 500],
  ["груша", 350],
  ["лимон", 250]
]);
// перебор по ключам (фрукты)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // яблоко, груша, лимон
}

// перебор по значениям (числа)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 250
}
// перебор по элементам в формате [ключ, значение]
```

```
    for (let entry of recipeMap) { // то же самое, что и
recipeMap.entries()
      alert(entry); // яблоко,500 (и так далее)
    }
```

Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array.

Листинг кода 5.4:

```
    // выполняем функцию для каждой пары (ключ, значение)
    recipeMap.forEach((value, key, map) => {
      alert(`${key}: ${value}`); // яблоко: 500 и так далее
    });
```

Set

Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

– new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.

– set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.

– set.delete(value) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.

– set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.

– set.clear() – удаляет все имеющиеся значения.

– set.size – возвращает количество элементов в множестве.

Листинг кода 5.5:

```
    let set = new Set();
    let john = { name: "John" };
    let pete = { name: "Pete" };
    let mary = { name: "Mary" };
```

```
// считаем гостей, некоторые приходят несколько раз
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);
// set хранит только 3 уникальных значения
alert(set.size); // 3
for (let user of set) {
  alert(user.name); // John (потом Pete и Mary)
}
```

Мы можем перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach.

Листинг кода 5.6:

```
let set = new Set(["апельсин", "яблоко", "банан"]);
for (let value of set) alert(value);
// то же самое с forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

Функция в forEach у Set имеет 3 аргумента: значение value, потом снова то же самое значение valueAgain, и только потом целевой объект. Это действительно так, значение появляется в списке аргументов дважды.

Это сделано для совместимости с объектом Map, в котором колбэк forEach имеет 3 аргумента. Выглядит немного странно, но в некоторых случаях может помочь легко заменить Map на Set и наоборот.

Set имеет те же встроенные методы, что и Map:

– set.keys() – возвращает перебираемый объект для значений,

– set.values() – то же самое, что и set.keys(), присутствует для обратной совместимости с Map,

– set.entries() – возвращает перебираемый объект для пар вида [значение,

значение], присутствует для обратной совместимости с Map.

Date

Объект Date содержит дату и время, а также предоставляет методы управления ими.

Для создания нового объекта Date нужно вызвать конструктор new Date() с одним из следующих аргументов:

1. new Date()

Без аргументов – создать объект Date с текущими датой и временем.

Листинг кода 5.7:

```
let now = new Date();
alert( now ); // показывает текущие дату и время
```

2. new Date(milliseconds)

Создать объект Date с временем, равным количеству миллисекунд (тысячная доля секунды), прошедших с 1 января 1970 года UTC+0.

Листинг кода 5.8:

```
// 0 соответствует 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );
// теперь добавим 24 часа и получим 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

Целое число, представляющее собой количество миллисекунд, прошедших с начала 1970 года, называется таймстамп (англ. timestamp).

3. new Date(datestring)

Если аргумент всего один, и это строка, то из неё «прочитывается» дата.

Литстинг кода 5.9:

```
let date = new Date("2017-01-26");
alert(date);
```

```
      // Время не указано, поэтому оно ставится в полночь по Гринвичу
и
      // меняется в соответствии с часовым поясом места выполнения
кода
      // Так что в результате можно получить
      //  Thu Jan 26 2017 11:00:00 GMT+1100 (восточно-австралийское
время)
      // или
      // Wed Jan 25 2017 16:00:00 GMT-0800 (тихоокеанское время)
```

4. new Date(year, month, date, hours, minutes, seconds, ms)

Создать объект Date с заданными компонентами в местном часовом поясе. Обязательны только первые два аргумента.

year должен состоять из четырёх цифр. Для совместимости также принимаются 2 цифры и рассматриваются как 19xx, к примеру, 98 здесь это тоже самое, что и 1998, но настоятельно рекомендуется всегда использовать 4 цифры.

month начинается с 0 (январь) по 11 (декабрь).

Параметр date здесь представляет собой день месяца. Если параметр не задан, то принимается значение 1.

Если параметры hours/minutes/seconds/ms отсутствуют, их значением становится 0.

Листинг кода 5.10:

```
      new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 Jan 2011, 00:00:00
      new Date(2011, 0, 1); // то же самое, так как часы и проч.
равны 0
      Максимальная точность – 1 мс (до 1/1000 секунды):
      let date = new Date(2011, 0, 1, 2, 3, 4, 567);
      alert( date ); // 1.01.2011, 02:03:04.567
```

Существуют методы получения года, месяца и т.д. из объекта Date:

1. getFullYear()-Получить год (4 цифры)

2. getMonth() - Получить месяц, от 0 до 11.

3. getDate() - Получить день месяца, от 1 до 31, что несколько противоречит названию метода.

4. getHours(), getMinutes(), getSeconds(), getMilliseconds() - Получить, соответственно, часы, минуты, секунды или миллисекунды.

5. getDay() - Вернуть день недели от 0 (воскресенье) до 6 (суббота). Несмотря на то, что в ряде стран за первый день недели принят понедельник, в JavaScript начало недели приходится на воскресенье.

6. getTime() - Для заданной даты возвращает таймстамп – количество миллисекунд, прошедших с 1 января 1970 года UTC+0.

7. getTimezoneOffset() - Возвращает разницу в минутах между UTC и местным часовым поясом

Следующие методы позволяют установить компоненты даты и времени:

- setFullYear(year, [month], [date])
- setMonth(month, [date])
- setDate(date)
- setHours(hour, [min], [sec], [ms])
- setMinutes(min, [sec], [ms])
- setSeconds(sec, [ms])
- setMilliseconds(ms)
- setTime(milliseconds) (устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC)

Автоисправление даты

Автоисправление – это очень полезная особенность объектов Date. Можно устанавливать компоненты даты вне обычного диапазона значений, а объект сам себя исправит.

Листинг кода 5.11:

```
let date = new Date(2013, 0, 32); // 32 Jan 2013 ?!?
alert(date); // ...1st Feb 2013!
```

Date.now()

Если нужно просто измерить время, объект Date нам не нужен.

Существует особый метод Date.now(), возвращающий текущую метку времени.

Семантически он эквивалентен new Date().getTime(), однако метод не создаёт промежуточный объект Date. Так что этот способ работает быстрее и не нагружает сборщик

мусора.

Метод Date.parse(str) считывает дату из строки.

Формат строки должен быть следующим: YYYY-MM-DDTHH:mm:ss.sssZ, где:

– YYYY-MM-DD – это дата: год-месяц-день.

– Символ "T" используется в качестве разделителя.

– HH:mm:ss.sss – время: часы, минуты, секунды и миллисекунды.

– Необязательная часть 'Z' обозначает часовой пояс в формате +-hh:mm. Если указать просто букву Z, то получим UTC+0.

Возможны и более короткие варианты, например, YYYY-MM-DD или YYYY-MM, или даже YYYY.

Вызов Date.parse(str) обрабатывает строку в заданном формате и возвращает таймстамп (количество миллисекунд с 1 января 1970 года UTC+0). Если формат неправильный, возвращается NaN.

Методы обработки списков .map, .filter, .reduce.

Метод .map

Метод arr.map является одним из наиболее полезных и часто используемых.

Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

Синтаксис:

```
let result = arr.map(function(item, index, array) {
  // возвращается новое значение вместо элемента
});
```

Листинг кода 5 12:

```
let lengths = ["Бильбо", "Гэндальф", "Назгул"].map(item => item.length);
alert(lengths); // 6,8,6
```

Метод . filter

Метод возвращает возвращает массив из всех подходящих элементов.

Синтаксис

```
let results = arr.filter(function(item, index, array) {
  // если `true` -- элемент добавляется к results и перебор продолжается
  // возвращается пустой массив в случае, если ничего не найдено
});
```

Листинг кода 5.13:

```
let users = [
  {id: 1, name: "Вася"},
  {id: 2, name: "Петя"},
  {id: 3, name: "Маша"}
];
// возвращает массив, состоящий из двух первых пользователей
let someUsers = users.filter(item => item.id < 3);
alert(someUsers.length); // 2
```

Метод . reduce

Методы reduce/reduceRight

Методы arr.reduce используется для вычисления единого значения на основе всего массива.

Синтаксис:

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов.

Аргументы:

– accumulator – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),

– item – очередной элемент массива,

– index – его позиция,

– array – сам массив.

Листинг кода 5.14:

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce((sum, current) => sum + current, 0);
alert(result); // 15
```

Пояснение:

	sum	current	result
первый вызов	0	1	1
второй вызов	1	2	3
третий вызов	3	3	6
четвёртый вызов	6	4	10
пятый вызов	10	5	15

Сложные объекты: дескрипторы свойств, Proxy, Reflect

Дескрипторы свойств

Объекты, как мы знаем, содержат свойства. У каждого из свойств объекта, кроме значения, есть ещё три флага конфигурации, которые могут принимать значения true или false. Эти флаги называются дескрипторами:

- writable — доступно ли свойство для записи;

- enumerable — является ли свойство видимым при перечислениях (например, в цикле for..in);

- configurable — доступно ли свойство для переконфигурирования.

Когда мы создаём свойство объекта «обычным способом», эти три флага устанавливаются в значение true.

Для изменения значений дескрипторов применяется статический метод Object.defineProperty(), а для чтения значений — Object.getOwnPropertyDescriptors().

Другими словами, дескрипторы — это пары ключ-значение, которые описывают поведение свойства объекта при выполнении операций над ним (например, чтения или записи).

Чтобы изменить флаги, мы можем использовать метод Object.defineProperty.

Его синтаксис:

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj, propertyName - объект и его свойство, для которого нужно применить дескриптор.

Descriptor -Применяемый дескриптор.

Пример:

```
let user = {};
Object.defineProperty(user, "name", {
  value: "John"
});
```

Также метод Object.defineProperty может определять множество свойств сразу.

Его синтаксис:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

Листинг кода 5.15:

```
Object.defineProperties(user, {
  name: { value: "John", writable: false },
  surname: { value: "Smith", writable: false },
  // ...
});
```

Метод Object.getOwnPropertyDescriptor позволяет получить полную информацию о свойстве.

Его синтаксис:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

obj - Объект, из которого мы получаем информацию.

propertyName - Имя свойства.

Листинг кода 5.16:

```
 let user = {
  name: "John"
};
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
```

Также метод Object.getOwnPropertyDescriptor может получить все дескрипторы свойств сразу.

Его синтаксис:

Object.getOwnPropertyDescriptors(obj).

Вместе с Object.defineProperties этот метод можно использовать для клонирования объекта вместе с его флагами:

Листинг кода 5.17:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Отличие в том, что for..in игнорирует символьные и неперечислимые свойства, а Object.getOwnPropertyDescriptors возвращает дескрипторы всех свойств.

Proxy

Объект Proxy «оборачивается» вокруг другого объекта и может перехватывать (и, при желании, самостоятельно обрабатывать) разные действия с ним, например чтение/запись свойств и другие. Далее мы будем называть такие объекты «прокси».

Прокси используются во многих библиотеках и некоторых браузерных фреймворках. В этой главе мы увидим много случаев применения прокси в решении реальных задач.

Синтаксис:

```
let proxy = new Proxy(target, handler);
```

target – это объект, для которого нужно сделать прокси, может быть чем угодно, включая функции.

handler – конфигурация прокси: объект с «ловушками» («traps»): методами, которые перехватывают разные операции, например, ловушка get – для чтения свойства из target, ловушка set – для записи свойства в target и так далее.

При операциях над proxy, если в handler имеется соответствующая «ловушка» (смотреть таблицу 1) , то она срабатывает, и прокси имеет возможность по-своему обработать её, иначе операция будет совершена над оригинальным объектом target.

Proxy – это особый, «экзотический», объект, у него нет собственных свойств. С пустым handler он просто перенаправляет все операции на target.

Таблица 5.1 – Внутренние методы и ловушки для операций proxy

Внутренний метод	Ловушка	Что вызывает
[[Get]]	get	чтение свойства
[[Set]]	set	запись свойства
[[HasProperty]]	has	оператор in
[[Delete]]	deleteProperty	оператор delete
[[Call]]	apply	вызов функции
[[Construct]]	construct	оператор new
[[GetPrototypeOf]]	getPrototypeOf	Object.getPrototypeOf
[[SetPrototypeOf]]	setPrototypeOf	Object.setPrototypeOf
[[IsExtensible]]	isExtensible	Object.isExtensible
[[PreventExtensions]]	preventExtensions	Object.preventExtensions
[[DefineOwnProperty]]	defineProperty	Object.defineProperty, Object.defineProperties
[[GetOwnProperty]]	getOwnPropertyDescriptor	Object.getOwnPropertyDescriptor, for..in, Object.keys/values/entries
[[OwnPropertyKeys]]	ownKeys	Object.getOwnPropertyNames, Object.getOwnPropertySymbols, for..in, Object.keys/values/entries

Листинг кода 5.18:

```
      const target = {
                    name: 'John',
                    age: 30
          };


const handler = {
```

```
      get: function(target, prop) {
      console.log(`Чтение свойства ${prop}`);
      return target[prop];
      },
      set: function(target, prop, value) {
      console.log(`Запись свойства ${prop} со значением ${value}`);
      target[prop] = value;
      }
      };

const proxy = new Proxy(target, handler);
      console.log(proxy.name); // Выводит "Чтение свойства name" и
"John"
      proxy.age = 35; // Выводит "Запись свойства age со значением
35"
```

Reflect

Reflect предоставляет набор методов, которые позволяют нам выполнять операции над объектами, такие как создание, чтение, запись и удаление свойств. Он предоставляет более простой и единообразный интерфейс для выполнения этих операций, чем использование непосредственно объектов.

Внутренние методы, такие как [[Get]], [[Set]] и другие, существуют только в спецификации, что к ним нельзя обратиться напрямую.

Объект Reflect делает это возможным. Его методы – минимальные обёртки вокруг внутренних методов.

Для каждого внутреннего метода, перехватываемого Proxy, есть соответствующий метод в Reflect, который имеет такое же имя и те же аргументы, что и у ловушки Proxy.

Ниже представлен использование Reflect без Proxy.

Листинг кода 5.19:

```
      const obj = {
               name: 'John',
               age: 30
      };
```

```
console.log(Reflect.get(obj, 'name')); // Выводит "John"

Reflect.set(obj, 'age', 35);
console.log(obj.age); // Выводит 35
```

Ниже представлен использование Reflect с Proxy.

Листинг кода 5.20:

```
let user = {
  name: "Вася",
};

user = new Proxy(user, {
  get(target, prop, receiver) {
    alert(`GET ${prop}`);
    return Reflect.get(target, prop, receiver); // (1)
  },
  set(target, prop, val, receiver) {
    alert(`SET ${prop}=${val}`);
    return Reflect.set(target, prop, val, receiver); // (2)
  }
});

let name = user.name; // выводит "GET name"
user.name = "Петя"; // выводит "SET name=Петя"
```

Proxy и Reflect предоставляют мощные возможности для перехвата и изменения поведения объектов в JavaScript. Они могут быть использованы для реализации различных паттернов и функциональностей, таких как валидация данных, логирование операций и многое другое.

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.

5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1. Ознакомиться с материалом методического указания.

2. Создать одностраничный сайт по варианту (№варианта=№ в журнале). На сайте должны осуществляется действия:

- использовать объекты: Map, Set, Date
- обработать список с помощью методов: .map, .filter, .reduce,
- использовать сложные объекты: дескрипторы свойств, Proxy, Reflect

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое объект Map?
2. Что такое объект Set?
3. Что такое объект Date?
4. Для чего нужен метод . map?
5. Для чего нужен метод . filter?
6. Для чего нужен метод . reduce?
7. Что такое дескрипторы свойств?
8. Что такое объект Proxy?
9. Что такое объект Reflect?

6. ЛАБОРАТОРНАЯ РАБОТА №6. Функции в JavaScript.

Цель работы: Решение задач на функции и рекурсию, создание набора функций, замкнутых на общую переменную.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Рекурсия

Рекурсия - это процесс, при котором функция вызывает саму себя внутри своего кода. В программировании рекурсия позволяет решать задачи, разбивая их на более простые подзадачи того же типа.

Основные черты рекурсии:

1. Базовый случай: в рекурсивной функции всегда должен быть б1азовый случай, который указывает, когда рекурсивные вызовы должны прекратиться. Без базового случая функция будет бесконечно вызывать саму себя, что приведет к переполнению стека (stack overflow).

2. Шаг рекурсии: это шаг или действие, которое делает функция перед тем, как вызвать себя рекурсивно. Обычно задача делится на более простые подзадачи, и функция вызывается для их решения.

Пример применения рекурсии: вычисление факториала числа. Факториал числа n (обозначается n!) - это произведение всех положительных целых чисел от 1 до n. Например, 5! = 5*4*3*2*1 = 120.

Пример рекурсивной функции вычисления факториала числа.

Листинг кода 6.1:

```
function factorial(n) {
  // базовый случай
  if (n === 0) {
    return 1;
  }
  // шаг рекурсии
  return n * factorial(n - 1);
}
```

```
console.log(factorial(5)); // результат: 120
```

Рекурсия является мощным инструментом в программировании, но ее следует использовать осторожно, чтобы избежать переполнения стека и эффективно решать задачи.

Замыкания

В JavaScript замыкание (closure) — это функция, которая запоминает своё лексическое окружение во время объявления и имеет доступ к переменным из этого окружения, даже после того как эта функция будет вызвана вне своего лексического контекста. Это позволяет функции использовать переменные, которые находятся вне её собственной области видимости, но остаются доступными благодаря особенностям замыкания.

Листинг кода 6.2:

```
function outerFunction() {
    let outerVariable = 'I am outside!';
    function innerFunction() {
        console.log(outerVariable); // Внутренняя функция имеет доступ к переменной outerVariable из внешней функции
    }
    return innerFunction;
}
let myFunction = outerFunction(); // myFunction теперь содержит в себе innerFunction и запоминает контекст outerFunction
myFunction(); // Выведет: I am outside!
```

Теперь рассмотрим пример счетчика.

Листинг кода 6.3:

```
function createCounter() {
        //
    let count = 0;

    return {
        increment: function () {
            count++;
```

```
            },
            decrement: function () {
                count--;
            },
            getCount: function () {
                return count;
            }
        };
    }
    let counter = createCounter();
    counter.increment();
    counter.increment();
    counter.decrement();
    console.log(counter.getCount());
```

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задачи по рекурсии:

1. Вычисление факториала числа: Напишите рекурсивную функцию, которая вычисляет факториал числа. Например, 5! = 5*4*3*2*1 = 120.

2. Вычисление чисел Фибоначчи: Напишите рекурсивную функцию для вычисления чисел Фибоначчи. Числа Фибоначчи определяются как последовательность, где каждое число равно сумме двух предыдущих чисел (например, 0, 1, 1, 2, 3, 5, 8, и так далее).

3. Подсчет количества элементов в массиве: Напишите рекурсивную функцию, которая подсчитывает количество элементов в массиве.

4. Рекурсивный подсчет суммы элементов массива: Напишите рекурсивную функцию, которая считает сумму всех элементов массива.

5. Поиск наибольшего элемента в массиве: Напишите рекурсивную функцию, которая находит наибольший элемент в массиве.

6. Рекурсивное вычисление степени числа: Напишите рекурсивную функцию для вычисления степени числа. Например, 2^3 = 2*2*2 = 8.

7. Развертывание массива: Напишите рекурсивную функцию, которая разворачивает массив наоборот.

8. Подсчет количества повторяющихся символов в строке: Напишите рекурсивную функцию, которая подсчитывает количество повторяющихся символов в строке.

9. Вычисление суммы элементов числового массива: Напишите рекурсивную функцию, которая принимает массив чисел и вычисляет сумму всех элементов.

10. Поиск наименьшего элемента в числовом массиве: Напишите рекурсивную функцию, которая находит наименьший элемент в числовом массиве.

11. Проверка на палиндром: Напишите рекурсивную функцию, которая проверяет, является ли строка палиндромом (читается одинаково с начала и с конца, игнорируя пробелы и знаки препинания).

Задание для выполнения:

1. Ознакомиться с материалом методического указания.

2. Создать одностраничный сайт по варианту (№варианта=№ в журнале). На сайте должны быть:

- функции и рекурсии (по варианту);

- набор функций, замкнутых на 1 переменную (ВНИМАНИЕ! НЕ использовать пример из методички).

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое рекурсия?

2. Что такое функция?

3. Что такое замыкание?

7. ЛАБОРАТОРНАЯ РАБОТА №7. Объекты и прототипы в JavaScript.

Цель работы: Решение задач с помощью объектов, создание конструктора объектов с набором общих методов.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество похожих, однотипных объектов, таких как пользователи, элементы меню и так далее.

Это можно сделать при помощи функции-конструктора и оператора "new".

Функция-конструктор

Функции-конструкторы технически являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.

2. Функция-конструктор должна выполняться только с помощью оператора "new".

Создадим конструктор объектов для пользователей. И назовем его User. Далее добавим пользователя с именем Jack.

Листинг кода 7.1:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Jack");
```

Когда функция вызывается как new User(...), происходит следующее:

- Создаётся новый пустой объект, и он присваивается this.

- Выполняется тело функции. Обычно оно модифицирует this, добавляя туда новые свойства.

- Возвращается значение this.

Другими словами, new User(...) делает что-то вроде

Листинг кода 7.2:

```
function User(name) {
  // this = {};  (неявно)
   // добавляет свойства к this
  this.name = name;
  this.isAdmin = false;

  // return this;  (неявно)
```

Также кроме свойств мы можем добавить методы.

Листинг кода 7.3:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
  this.sayHi = function() {
    alert( "Меня зовут: " + this.name );
  };
}
let john = new User("John");
john.sayHi(); // Меня зовут: John
```

Давайте рассмотрим пример. Мы создаем конструктор объектов "Employee", который содержит свойства name, position, department, salary, performance и два метода increaseSalary и changePosition. Методы могут изменять свойства сотрудника, управлять его данными.

Листинг кода 7.4:

```
// Конструктор объектов
function Employee(name, position, department, salary) {
  this.name = name;
  this.position = position;
  this.department = department;
  this.salary = salary;
  this.performance = 'good';
```

```
      }
      // Метод для повышения зарплаты
      Employee.prototype.increaseSalary = function(amount) {
        this.salary += amount;
        console.log(`${this.name} получил повышение и теперь
зарабатывает ${this.salary} рублей.`);
      }
      // Метод для изменения должности
      Employee.prototype.changePosition = function(newPosition) {
        this.position = newPosition;
        console.log(`${this.name} перешел на новую должность -
${this.position}.`);
      }
      // Создание новых сотрудников
      let employee1 = new Employee('Иван', 'Инженер-программист',
'Отдел разработки', 70000);
      let employee2 = new Employee('Мария', 'Тестировщик', 'Отдел
тестирования', 60000);
      // Вызов методов объектов
      employee1.increaseSalary(10000);
      employee2.changePosition('Специалист по автоматизированному
тестированию');
      // Вывод данных сотрудников
      console.log(employee1);
      console.log(employee2);
```

Результат работы представлен на рисунке 7.1.

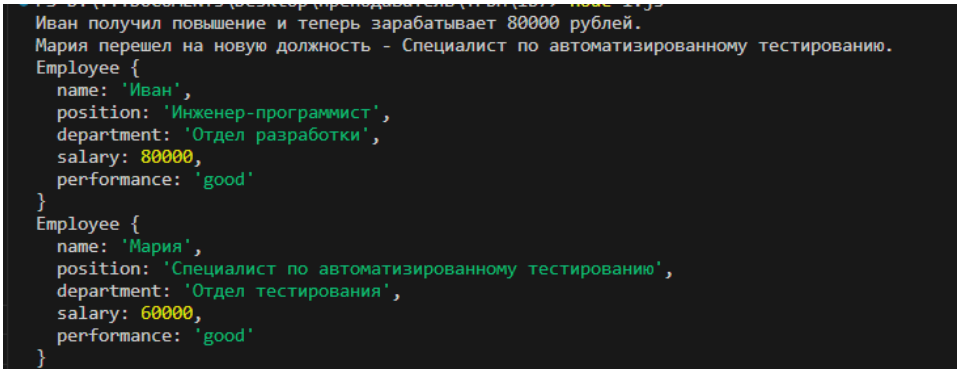

Рисунок 7.1

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1. Ознакомиться с материалом методического указания.

2. Создать конструктор объекта с определенными набором свойствами методов (по варианту).

3. Создать несколько объектов и вывести их на сайт, используя все навыки, полученные ранее.

4. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое объект?

2. Что такое прототип?

3. Что такое конструктор объекта?

8. ЛАБОРАТОРНАЯ РАБОТА №8.

Цель работы: Решение задач на ООП, написание кода с использованием классов.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Понятие ООП (Объектно-Ориентированное Программирование) представляет подход к разработке программного обеспечения, в котором данные и методы обработки этих данных объединены в объекты. Основные идеи ООП включают в себя инкапсуляцию, наследование и полиморфизм.

Вот объяснение этих трех основных принципов ООП:

1. Инкапсуляция: Инкапсуляция означает объединение данных и методов, которые работают с этими данными, внутри объекта, и скрытие деталей реализации от внешнего мира. Это позволяет защитить данные объекта от непосредственного доступа извне и обеспечивает более безопасное и надежное использование объектов. Инкапсуляция также позволяет упростить интерфейс объекта, предоставляя только необходимые методы доступа к данным.

2. Наследование: Наследование позволяет создавать новые классы (или объекты), которые наследуют свойства и методы существующих классов. Наследование способствует повторному использованию кода, облегчает структурирование программы и уменьшает дублирование кода. Родительский класс называется базовым классом, а дочерний — производным. Производный класс может расширять или изменять функциональность базового класса.

3. Полиморфизм: Полиморфизм позволяет объектам разных классов иметь методы с одинаковыми именами, но с различной реализацией. Это позволяет обрабатывать объекты производных классов как объекты базового класса, что упрощает код и делает его более гибким. Полиморфизм может реализовываться через перегрузку методов или через реализацию одинаковых методов у различных классов.

В JavaScript объектно-ориентированный подход реализуется путем создания объектов с помощью функций-конструкторов и прототипов. Вот основные принципы и синтаксис объектно-ориентированного программирования в JavaScript.

Создание объектов:

Объекты могут быть созданы с помощью литерала объекта {}, конструктора new Object() или собственной функции-конструктора.

Листинг кода 8.1:

```
// Литерал объекта
var person = {
    name: "Alice",
    age: 30,
    greet: function() {
        return "Hello, my name is " + this.name;
    }
};
// Функция-конструктор
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        return "Hello, my name is " + this.name;
    };
}
var person1 = new Person("Bob", 25);
```

Прототипы:

Прототипы позволяют расширять функциональность объектов JavaScript путем добавления методов и свойств к прототипу конструктора. Это помогает экономить память, поскольку методы и свойства будут общими для всех экземпляров.

Листинг кода 8.2:

```
// Добавление метода через прототип
Person.prototype.introduce = function() {
    return "I am " + this.name + " and I am " + this.age + "
years old.";
};
var person2 = new Person("Charlie", 35);
```

Наследование:

Наследование в JavaScript осуществляется через прототипы. Родительский конструктор может быть связан с дочерним конструктором с помощью Object.create() или установкой прототипа.

Листинг кода 8.3:

```
// Наследование с помощью Object.create()
function Student(name, age, grade) {
    Person.call(this, name, age);
    this.grade = grade;
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;
var student1 = new Student("David", 20, "A");
```

4. Encapsulation (Инкапсуляция):

В JavaScript инкапсуляция может быть достигнута с помощью замыканий.

Листинг кода 8.4:

```
function Counter() {
    var count = 0;
    this.increment = function() {
        count++;
    };
    this.getCount = function() {
        return count;
    };
}
var counter = new Counter();
counter.increment();
console.log(counter.getCount());
```

Классы в JavaScript стали актуальными с появлением стандарта ECMAScript 2015 (ES6), который добавил синтаксический сахар для создания классов в JavaScript. Хотя JavaScript изначально был прототипно-ориентированным языком, где объекты создавались с помощью функций-конструкторов и прототипов, классы предоставляют более удобный и понятный способ создания объектов, в частности для разработчиков, привыкших к классическому объектно-ориентированному программированию.

Листинг кода 8.5:

```
class Author {
   name;
   surname;
   books = [];
   constructor(name, surname) {
       this.name = name;
       this.surname = surname;
   }
   writeBook (title) {
       this.books.push(title)
   }
}
Author.prototype.country = 'Россия';
const author = new Author("Владимир", "Высоцкий");
const secondAuthor = new Author("Александр", "Пушкин")
console.log(author);
console.log(secondAuthor);
console.log(author.country);
console.log(secondAuthor.country);
```

Результат работы представлен на рисунке 8.1.

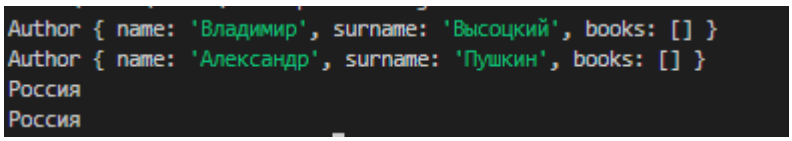

Рисунок 8.1 - Вывод значения

Инкапсуляция

Приватные поля обозначаются символом #. К таким полям нет возможности обратиться извне. Добавим такое поле к нашему классу.

Листинг кода 8.6:

```
class Author {
   name;
   surname;
   #midname;
   books = [];
   constructor(name, surname) {
       this.name = name;
       this.surname = surname;
   }
   writeBook (title) {
       this.books.push(title)
   }
}
Author.prototype.country = 'Россия';
const author = new Author("Владимир", "Высоцкий");
const secondAuthor = new Author("Александр", "Пушкин")
console.log(author);
console.log(secondAuthor);
console.log(author.country);
console.log(author.country);
console.log(author.#midname) // <- ошибка
```

Аналогично можно поступить и с методами. К приватным полям можно получить доступ только из этого класса.

С помощью ключевого слова static можно отметить метод или поле как статическое. К такому полю или методу можно получить доступ даже без создания экземпляра класса. Однако, к статическим полям и методам из нестатических методов через this.

Листинг кода 8.7:

```
class Author {
    name;
    surname;
    #midname;
    static sayHallo() {
        console.log("Привет")
    }
    books = [];
    constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }
    writeBook (title) {
        this.books.push(title)
    }
}
Author.prototype.country = 'Россия';
const author = new Author("Владимир", "Высоцкий");
const secondAuthor = new Author("Александр", "Пушкин")
console.log(author);
console.log(secondAuthor);
console.log(author.country);
console.log(secondAuthor.country);
Author.sayHallo()
```

Результат работы представлен на рисунке 8.2.

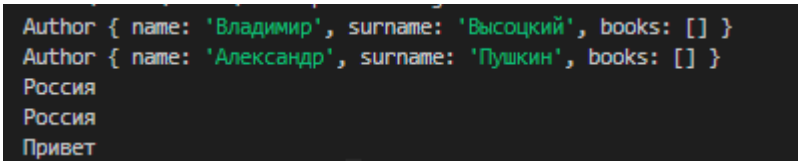

Рисунок 8.2 – Вывод значений

Взаимодействовать с приватными полями извне можно с помощью геттеров и сеттеров для получения и установки значения соответственно.

Листинг кода 8.8:

```
class Author {
    name;
    surname;
    #midname;
    get midname() {
        return this.#midname;
    }

    set midname(midname) {
        this.#midname = midname
    }
    books = [];
    constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }
    writeBook (title) {
        this.books.push(title)
    }
}
Author.prototype.country = 'Россия';
const author = new Author("Владимир", "Высоцкий");
const secondAuthor = new Author("Александр", "Пушкин")
author.midname = "Сергеевич";
console.log(author.midname)
```

Результат работы представлен на рисунке 8.3.

Рисунок 8.3 – Вывод значений

Наследование.

Наследоваться от существующего класса можно с помощью ключевого слова extends.

Листинг кода 8.9:

```
class Student extends Person {
    constructor(name, age, grade) {
        super(name, age);
        this.grade = grade;
    }
    introduce() {
        return `I am ${this.name} and I am in grade ${this.grade}.`;
    }
}
```

Вызвать конструктор родительского класса в дочернем классе можно с помощью команды super(Аргумент1…..).

Полиморфизм:

Листинг кода 8.10:

```
class Animal {
  speak() {
    console.log('Animal makes a sound');
  }
}
class Dog extends Animal {
  speak() {
    console.log('Dog barks');
  }
}
class Cat extends Animal {
  speak() {
    console.log('Cat meows');
```

```
    }
}
const animal = new Animal();
const dog = new Dog();
const cat = new Cat();
animal.speak(); // Выводит "Animal makes a sound"
dog.speak(); // Выводит "Dog barks"
cat.speak(); // Выводит "Cat meows"
```

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2.Реализуйте несколько (2 или больше) объектов и продемонстрируйте на них 3 основные парадигмы ООП (наследование, инкапсуляция и полиморфизм);

3.Реализуйте несколько (2 или больше) классов и продемонстрируйте на них 3 основные парадигмы ООП (наследование, инкапсуляция и полиморфизм);

4. Составить отчет по результатам работы

ВНИМАНИЕ!!! ОБЪКТЫ И КЛАССЫ ДОЛЖНЫ БЫТЬ ПО ТЕМЕ ВАШЕГО САЙТА!!

Вопросы по лабораторной работе:

1.Что такое ООП?

2.Что такое объекты в ООП?

3.Основные парадигмы ООП?

9. ЛАБОРАТОРНАЯ РАБОТА №9. Обработка исключений в JavaScript.

Цель работы: Решение задач на обработку исключений.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Обработка исключений в JavaScript осуществляется с помощью конструкции try-catch. Исключение (ошибка) может возникнуть в результате выполнения некорректного кода, например, деления на ноль или обращения к несуществующему объекту.

Синтаксис «try…catch»

Конструкция try..catch состоит из двух основных блоков: try, и затем catch.

Листинг кода 9.1:

```
try {
  // код...
} catch (err) {
  // обработка ошибки
}
```

Работает она так:

1. Сначала выполняется код внутри блока try {...}.

2. Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.

3.Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

Для всех встроенных ошибок этот объект имеет два основных свойства:

- name. Имя ошибки. Например, для неопределённой переменной это "ReferenceError".

- message. Текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это:

- stack. Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

Листинг кода 9.2:

```
      try {
        lalala; // ошибка, переменная не определена!
      } catch(err) {
        alert(err.name); // ReferenceError
        alert(err.message); // lalala is not defined
        alert(err.stack); // ReferenceError: lalala is not defined at
(...стек вызовов)

        // Можем также просто вывести ошибку целиком
        // Ошибка приводится к строке вида "name: message"
        alert(err); // ReferenceError: lalala is not defined
      }
```

Давайте рассмотрим пример, когда json некорректен и JSON.parse генерирует ошибку, то есть скрипт «падает».

Листинг кода 9.3:

```
      let json = "{ некорректный JSON }";
      try {

        let user = JSON.parse(json); // <-- тут возникает ошибка...
        alert( user.name ); // не сработает

      } catch (e) {
        // ...выполнение прыгает сюда
        alert( "Извините, в данных ошибка, мы попробуем получить их
ещё раз." );
        alert( e.name );
        alert( e.message );
      }
```

Генерация собственных ошибок

Для того, чтобы унифицировать обработку ошибок, мы воспользуемся оператором throw.

Оператор throw генерирует ошибку.

Синтаксис:

```
throw <объект ошибки>
```

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами name и message (для совместимости со встроенным ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: Error, SyntaxError, ReferenceError, TypeError и другие. Можно использовать и их для создания объектов ошибки.

Листинг кода 9.4:

```
let error = new Error(message);
// или
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

Для встроенных ошибок (не для любых объектов, только для ошибок), свойство name – это в точности имя конструктора. А свойство message берётся из аргумента.

В нашем случае отсутствие свойства name – это ошибка, ведь пользователи должны иметь имена.

Сгенерируем её.

Листинг кода 9.5:

```
let json = '{ "age": 30 }'; // данные неполны


try {
  let user = JSON.parse(json); // <-- выполнится без ошибок
```

```
        if (!user.name) {
          throw new SyntaxError("Данные неполны: нет имени");
        }
        alert( user.name );
      } catch(e) {
        alert( "JSON Error: " + e.message ); // JSON Error: Данные
неполны: нет имени
      }
```

Теперь блок catch становится единственным местом для обработки всех ошибок: и для JSON.parse и для других случаев.

Есть простое правило: Блок catch должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.

Техника «проброс исключения» выглядит так:

1. Блок catch получает все ошибки.

2. В блоке catch(err) {...} мы анализируем объект ошибки err.

3. Если мы не знаем как её обработать, тогда делаем throw err.

В коде ниже мы используем проброс исключения, catch обрабатывает только SyntaxError.

Листинг кода 9.6:

```
      let json = '{ "age": 30 }'; // данные неполны
      try {

        let user = JSON.parse(json);

        if (!user.name) {
          throw new SyntaxError("Данные неполны: нет имени");
        }

        blabla(); // неожиданная ошибка

        alert( user.name );
```

```
} catch(e) {

  if (e.name == "SyntaxError") {
    alert( "JSON Error: " + e.message );
  } else {
    throw e; // проброс (*)
  }

}
```

try…catch…finally

Подождите, это ещё не всё.

Конструкция try..catch может содержать ещё одну секцию: finally.

Если секция есть, то она выполняется в любом случае:

- после try, если не было ошибок,
- после catch, если ошибки были.

Расширенный синтаксис выглядит следующим образом.

Листинг кода 9.6:

```
try {
   ... пробуем выполнить код...
} catch(e) {
   ... обрабатываем ошибки ...
} finally {
   ... выполняем всегда ...
}
```

Секцию finally часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от того, будет ошибка или нет.

Например, мы хотим измерить время, которое занимает функция чисел Фибоначчи

fib(n). Естественно, мы можем начать измерения до того, как функция начнёт выполняться и закончить после. Но что делать, если при вызове функции возникла ошибка? В частности, реализация fib(n) в коде ниже возвращает ошибку для отрицательных и для нецелых чисел.

Секция finally отлично подходит для завершения измерений несмотря ни на что.

Здесь finally гарантирует, что время будет измерено корректно в обеих ситуациях – и в случае успешного завершения fib и в случае ошибки.

Листинг кода 9.10:

```
let num = +prompt("Введите положительное целое число?", 35)
let diff, result;
function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Должно быть целое неотрицательное число");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
let start = Date.now();
try {
  result = fib(num);
} catch (e) {
  result = 0;
} finally {
  diff = Date.now() - start;
}
alert(result || "возникла ошибка");
alert( `Выполнение заняло ${diff}ms` );
```

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.

9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Обработать исключения с помощью:

- «try…catch»;
- генерации собственных ошибок;
- «try…catch…finally».

3. Составить отчет по результатам работы

ВНИМАНИЕ!!! Примеры из методички брать НЕЛЬЗЯ!!!!!

Вопросы по лабораторной работе:

1.Что обработка исключений?

2. Для чего нужна конструкция «try…catch»?

3. Для чего нужна конструкция «try…catch…finally»?

10. ЛАБОРАТОРНАЯ РАБОТА №10. Обработка действий пользователя в браузере.

Цель работы: Создание веб-страницы, обрабатывающей движение мыши и нажатие кнопок на клавиатуре.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

События

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

События мыши:

– click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).

– contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.

– mouseover / mouseout – когда мышь наводится на / покидает элемент.

– mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.

– mousemove – при движении мыши.

– dblclick - вызывается двойным кликом на элементе.

- Модификаторы: shift, alt, ctrl и meta. Все события мыши включают в себя информацию о нажатых клавишах-модификаторах. Свойства события:

- • shiftKey: Shift
- • altKey: Alt (или Opt для Mac)
- • ctrlKey: Ctrl
- • metaKey: Cmd для Mac

- Отключаем выделение

Двойной клик мыши имеет побочный эффект, который может быть неудобен в некоторых интерфейсах: он выделяет текст.

В данном случае самым разумным будет отменить действие браузера по умолчанию при событии mousedown, это отменит оба этих выделения:

Листинг кода 10.1:

```
До...
```

```
<b ondblclick="alert('Клик!')" onmousedown="return false">
  Сделайте двойной клик на мне
</b>
...После
```

- Предотвращение копирования

Если мы хотим отключить выделение для защиты содержимого страницы от копирования, то мы можем использовать другое событие: oncopy.

Листинг кода 10.2:

```
<div oncopy="alert('Копирование запрещено!');return false">
```

Уважаемый пользователь,

Копирование информации запрещено для вас.

Если вы знаете JS или HTML, вы можете найти всю нужную вам информацию в исходном коде страницы.

События на элементах управления:

– События кнопки:
  - click: Срабатывает при клике на кнопку
  - mouseover: Срабатывает, когда курсор мыши наводится на элемент
  - mouseout: Срабатывает, когда курсор мыши покидает элемент

– События формы:
  - submit: Срабатывает при отправке формы
  - change: Срабатывает при изменении значения элемента формы, например, текстового поля или выбора из выпадающего списка
  - focus: Срабатывает при установке фокуса на элемент формы

– События других элементов управления:
  - input: Срабатывает при изменении значения вводимого пользователем данных (например, текстового поля)
  - focus: Срабатывает при установке фокуса на элементе
  - blur: Срабатывает при потере фокуса элементом

Клавиатурные события:

- keydown: Событие keydown срабатывает при нажатии клавиши на клавиатуре.

- keyup: Событие keyup срабатывает при отпускании клавиши на клавиатуре.

- keypress: - Событие keypress срабатывает при нажатии клавиши, которая может породить символ.

- focus: Событие focus срабатывает, когда элемент получает фокус.

- blur: Событие blur срабатывает, когда элемент теряет фокус.

- input: Событие input срабатывает при изменении значения элемента ввода.

Обработка событий

- Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Листинг кода 10.3:

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

- Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

Листинг кода 10.4:

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Кролик номер " + i);
    }
  }
</script>
<input type="button" onclick="countRabbits()" value="Считать кроликов!">
```

- Использование свойства DOM-объекта. Можно назначать обработчик, используя свойство DOM-элемента on<событие>.

Листинг кода 10.5:

```
<input id="elem" type="button" value="Нажми меня!">
<script>
  elem.onclick = function() {
    alert('Спасибо');
  };
</script>
```

Также можно добавить уже существующаю функцию.

Листинг кода 10.6:

```
function sayThanks() {
  alert('Спасибо!');
}
elem.onclick = sayThanks;
```

Частая ошибка писать!!

Листинг кода 10.7:

```
// правильно
button.onclick = sayThanks;
// неправильно
button.onclick = sayThanks();
```

Важно!!

- Регистр имеет значение(писать elem.onclick,а не elem.ONCLICK)
- Используйте именно функции, а не строки.
- Не используйте setAttribute для обработчиков.

addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие.

Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение.

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener. Они свободны от указанного недостатка.

Синтаксис добавления обработчика.

Листинг кода 10.8:

```
element.addEventListener(event, handler, [options]);
```

event - Имя события, например "click".

handler -Ссылка на функцию-обработчик.

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Создайте веб-страницу и добавьте для нее обработчик событий для мыши (минимум 3 события), для клавиатуры (минимум 1), кнопки и формы. Так же предотвратите копирование и выделение текста.

3. Составить отчет по результатам работы

ВНИМАНИЕ!!! Примеры из методички брать НЕЛЬЗЯ!!!!!

Вопросы по лабораторной работе:

1. Что такое «Событие»?

2. Как обрабатываются события?

3. Какие события мыши вам известны?

4. Какие события клавиатуры вам известны?

3. Какие события кнопки вам известны?

11. ЛАБОРАТОРНАЯ РАБОТА №11. Асинхронное программирование в JavaScript.

Цель работы: Решение задач на асинхронное программирование, промисификация функций, написание циклов и условий с использованием функций с callback-ами и Promise-ов

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Callback

Callback-функции в JavaScript - это функции, которые передаются в качестве аргументов в другие функции и вызываются внутри этих функций. Они используются для выполнения определенных действий после завершения выполнения другой функции.

Рассмотрим пример, где после загрузки скрипта вызывается функция, которая нахохлится в этом скрипте.

Листинг кода 11.1:

```
//Работать не будет
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
// загрузит и выполнит скрипт
loadScript('/my/script.js');  // в скрипте есть "function newFunction() {…}"
newFunction(); // такой функции не существует!
```

Код будет не рабочий так как код после loadScript() не будет ждать пока она выполниться, потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.

Для того чтобы все заработало нежно добавить колбэк.

Листинг кода 11.2:

```
//Рабочий код
function loadScript(src, callback) {
```

```
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}
loadScript('/my/script.js', function() {
  // эта функция вызовется после того, как загрузится скрипт
  newFunction(); // теперь всё работает
  ...
});
```

Колбек в колбеке

Как нам загрузить два скрипта один за другим: сначала первый, а за ним второй?

Листинг кода 11.3:

```
loadScript('/my/script.js', function(script) {
  alert(`Здорово, скрипт ${script.src} загрузился, загрузим ещё
один`);
  loadScript('/my/script2.js', function(script) {
    alert(`Здорово, второй скрипт загрузился`);
  });
});
```

Когда внешняя функция loadScript выполнится, вызовется та, что внутри колбэка.

А что если нам нужно загрузить ещё один скрипт?

Листинг кода 11.4:

```
loadScript('/my/script.js', function(script) {
  loadScript('/my/script2.js', function(script) {
    loadScript('/my/script3.js', function(script) {
      // ...и так далее, пока все скрипты не будут загружены
    });
  })
});
```

Каждое новое действие мы вынуждены вызывать внутри колбэка. Этот вариант подойдёт, когда у нас одно-два действия, но для большего количества уже не удобно.

Использование колбэков в колбеке может стать неудобным и привести к так называемому "callback hell" или "пекло колбэков", когда код становится трудночитаемым и сложноуправляемым из-за избытка вложенных колбэков.

Обработка ошибок

В примерах выше мы не думали об ошибках. А что если загрузить скрипт не удалось? Колбэк должен уметь реагировать на возможные проблемы.

Ниже улучшенная версия loadScript, которая умеет отслеживать ошибки загрузки.

Листинг кода 11.5:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));
 document.head.append(script);
}

loadScript('/my/script.js', function(error, script) {
  if (error) {
    // обрабатываем ошибку
  } else {
    // скрипт успешно загружен
  }
});
```

Мы вызываем callback(null, script) в случае успешной загрузки и callback(error), если загрузить скрипт не удалось.

Правила таковы:

1. Первый аргумент функции callback зарезервирован для ошибки. В этом случае вызов

выглядит вот так: callback(err).

2. Второй и последующие аргументы — для результатов выполнения. В этом случае вызов выглядит вот так: callback(null, result1, result2…).

Одна и та же функция callback используется и для информирования об ошибке, и для передачи результатов.

Рассмотрим еще одни пример.

Листинг кода 11.6:

```
function fetchData(callback) {
  setTimeout(() => {
    let success = false;
    if (success) {
      callback(null, "Данные загружены успешно");
    } else {
      callback(new Error("Ошибка загрузки данных"), null);
    }
  }, 2000);
}
fetchData((error, data) => {
  if (error) {
    console.error(error);
  } else {
    console.log(data);
  }
});
```

Промисы (promises)

Промисы (promises) в JavaScript - это объекты, представляющие асинхронные операции и позволяющие обрабатывать результат или ошибку после завершения операции. Использование промисов помогает избежать "callback hell" и делает асинхронный код более читаемым и управляемым.

Промисы имеют три состояния:

1. Pending (ожидание) - начальное состояние промиса.

2. Fulfilled (выполнено) - промис выполнен успешно.

3. Rejected (отклонено) - промис отклонен из-за ошибки.

Синтаксис создания Promise.

Листинг кода 11.7:

```
let promise = new Promise(function(resolve, reject) {
  // функция-исполнитель (executor)
  // "певец"
});
```

Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. В терминах нашей аналогии: исполнитель – это «певец».

Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

-resolve(value) — если работа завершилась успешно, с результатом value.

- reject(error) — если произошла ошибка, error – объект ошибки.

Итак, исполнитель запускается автоматически, он должен выполнить работу, а затем вызвать resolve или reject.

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

-state («состояние») — вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.

-result («результат») — вначале undefined, далее изменяется на value при вызове resolve(value) или на error при вызове reject(error).

Изменение состояний и результатов в зависимости от значения promise представлен на рисунке 1.

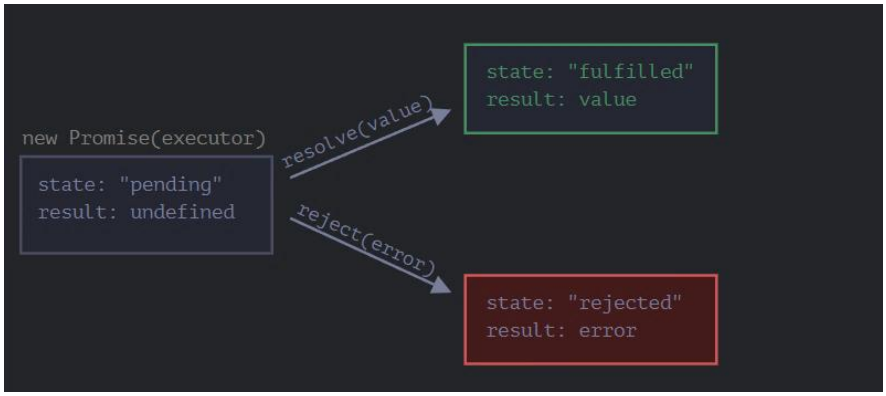

Рисунок 1

Есть правило. Можно в промисе вызвать либо результат либо ошибку. Если вы все-таки вызвали в промисе и то и другое, то он примет в себя 1 вызванное состояние, а другие будет игнорировать.

Листинг кода 11.8:

```
//Может быть что-то одно: либо результат, либо ошибка
let promise = new Promise(function(resolve, reject) {
  resolve("done");
  reject(new Error("…")); // игнорируется
  setTimeout(() => resolve("…")); // игнорируется
});
```

Свойства state и result – это внутренние свойства объекта Promise и мы не имеем к ним прямого доступа. Для обработки результата следует использовать методы .then/.catch/.finally.

Метод then используется для обработки успешного выполнения промиса, а метод catch - для обработки ошибки. Также можно добавить метод finally для выполнения действий в любом случае после завершения промиса.

1.Обработчик, вызываемый из finally, не имеет аргументов. В finally мы не знаем, как был завершён промис. И это нормально, потому что обычно наша задача – выполнить «общие» завершающие процедуры.

2. Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам.

3. Обработчик finally также не должен ничего возвращать. Если это так, то возвращаемое значение молча игнорируется.

Листинг кода 11.9:

```
      // Функция, возвращающая промис, имитирующая асинхронную операцию
      function asyncOperation(success) {
        return new Promise((resolve, reject) => {
          setTimeout(() => {
            if (success) {
              resolve('Успешное завершение операции');
            } else {
              reject('Ошибка: операция не выполнена');
            }
          }, 2000);
        });
      }

      // Вызываем асинхронную операцию с success=true
      asyncOperation(true)
        .then(result => {
          console.log(result);  // Обработка успешного выполнения операции
        })
        .catch(error => {
          console.error(error); // Обработка ошибки
        })
        .finally(() => {
          console.log('Завершение выполнения промиса');  // Завершающее действие
      });

      // Вызываем асинхронную операцию с success=false
      asyncOperation(false)
        .then(result => {
          console.log(result); // Этот блок не будет выполнен из-за вызова reject в промисе
```

```
        })
        .catch(error => {
          console.error(error); // Обработка ошибки
        })
        .finally(() => {
          console.log('Завершение выполнения промиса'); //
Завершающее действие
      });
```

Создание цепочки промисов позволяет последовательно выполнять асинхронные операции в определенном порядке и обрабатывать результаты каждой операции.

Для создания цепочки промисов вы можете использовать методы then и catch. Метод then принимает два колбэка - один для успешного выполнения промиса, и второй для обработки ошибок. Метод catch используется для обработки ошибок в любом из промисов в цепочке.

Листинг кода 11.10:

```
      function asyncOperation() {
        return new Promise((resolve, reject) => {
          setTimeout(() => {
            resolve("Данные загружены успешно");
          }, 2000);
        });
      }
      asyncOperation()
        .then(data => {
          console.log(data); // "Данные загружены успешно"
          return "Дополнительные данные";
        })
        .then(data => {
          console.log(data); // "Дополнительные данные"
          throw new Error("Ошибка обработки данных");
        })
        .catch(error => {
          console.error(error.message); // "Ошибка обработки данных"
```

```
  });
```

Промисификация функций

Промисификация функций в JavaScript - это процесс преобразования функций, которые используют колбэки для обратного вызова, в функции, возвращающие промисы для управления асинхронным кодом.

Промисификация делает асинхронный код более удобным и читаемым, уменьшая вложенность колбэков и упрощая обработку ошибок.

Листинг кода 11.11:

```
function loadDataFromNetwork(callback) {
  setTimeout(() => {
    let success = true; // Имитация успешного выполнения операции
    if (success) {
      callback(null, "Данные загружены успешно");
    } else {
      callback(new Error("Ошибка загрузки данных"), null);
    }
  }, 2000);
}
// Промисификация функции
function loadDataFromNetworkAsync() {
  return new Promise((resolve, reject) => {
    loadDataFromNetwork((error, data) => {
      if (error) {
        reject(error);
      } else {
        resolve(data);
      }
    });
  });
}
// Использование промисифицированной функции
```

```
loadDataFromNetworkAsync()
  .then(data => {
    console.log(data); // "Данные загружены успешно"
  })
  .catch(error => {
    console.error(error); // "Ошибка загрузки данных"
  });
```

Данный код демонстрирует промисификацию функции loadDataFromNetwork, которая имитирует загрузку данных из сети. После этого промисифицированная версия этой функции loadDataFromNetworkAsync используется для загрузки данных с использованием промисов.

1. Функция loadDataFromNetwork:

- Использует setTimeout для имитации асинхронной операции загрузки данных с задержкой в 2 секунды.

- Устанавливает переменную success в true, имитируя успешное выполнение операции.

- Вызывает колбэк callback с параметрами null и сообщением "Данные загружены успешно" в случае успешной загрузки данных, или с объектом ошибки и null в случае ошибки.

2. Промисификация функции с помощью loadDataFromNetworkAsync:

- Создает новый промис, внутри которого вызывается функция loadDataFromNetwork.

- В колбэке loadDataFromNetwork проверяется наличие ошибки: если она есть, промис вызывает метод reject с этой ошибкой, иначе - метод resolve с данными.

3. Использование промисифицированной функции:

- Вызов loadDataFromNetworkAsync() загружает данные с использованием промисов.

- Метод .then() обрабатывает успешное выполнение промиса, выводя сообщение "Данные загружены успешно".

- Метод .catch() обрабатывает ошибку при загрузке данных, выводя сообщение "Ошибка загрузки данных".

Асинхронные функции в JavaScript

Асинхронные функции в JavaScript - это функции, которые выполняются асинхронно, позволяя организовать логику работы с асинхронными операциями (например, запросы к серверу или чтение файлов), не блокируя выполнение других операций.

Операторы async и await в JavaScript предоставляют удобный способ работы с асинхронным кодом, делая его более читаемым и управляемым.

Оператор async позволяет объявить функцию асинхронной. Такая функция всегда возвращает промис, даже если результат возвращается синхронно.

Листинг кода 11.12:

```
async function loadData() {
  return "Данные загружены успешно";
}
loadData().then(data => {
  console.log(data); // "Данные загружены успешно"
});
```

Оператор await используется внутри асинхронной функции для приостановки выполнения кода до тех пор, пока промис не будет разрешен.

Листинг кода 11.13:

```
async function loadData() {
  let result = await fetchData(); // Выполнение приостанавливается до разрешения промиса fetchData()
  return result;
}
loadData().then(data => {
  console.log(data);
});
```

Обработка ошибок

При использовании асинхронных функций также можно обрабатывать ошибки с помощью блока try/catch. В случае возникновения ошибки в асинхронной функции, она будет преобразована в отклоненный промис.

Листинг кода 11.13:

```
async function fetchData() {
  try {
    let result = await new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
        let success = false;
        if (success) {
          resolve("Данные загружены успешно");
        } else {
          reject(new Error("Ошибка загрузки данных"));
        }
      }, 2000);
    });
      console.log(result);
    } catch (error) {
      console.error(error);
    }
}
fetchData();
```

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Написать циклы и условия с использованием функций с callback-ами

3. Написать циклы и условия с использованием функций с Promise-ов

4. Написать промисификация функции.

5. Написать асинхронную функцию.

6.Обработайте исключения при использовании колбеков, промисов и асинхронных функций.

7. Составить отчет по результатам работы

ВНИМАНИЕ!!! Примеры из методички брать НЕЛЬЗЯ!!!!!

Вопросы по лабораторной работе:

1. Что такое callback-функции?
2. Что такое колбек в колбеке?
3. Как обработать ошибки для колбеков?
4. Что такое промисы?
5. Какие есть методы обработки просимов?
6. Что такое асинхронные функции
7. Какие обработать исключения при использованиях асинхронных функций?

12. ЛАБОРАТОРНАЯ РАБОТА №12. JavaScript анимация.

Цель работы: Создание анимации слайдера с помощью JavaScript

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Анимация - это процесс создания иллюзии движения путем последовательного отображения статических изображений или объектов. В контексте веб-разработки, анимации часто используются для добавления эффектов и динамичности на веб-сайтах и веб-приложениях.

Анимация может применяться к различным элементам, таким как текст, изображения, кнопки, фоны и другие элементы интерфейса. Существует множество различных видов анимации, таких как перемещение, изменение размера, изменение цвета, поворот, затухание и другие эффекты, которые могут быть использованы для достижения конкретных целей и создания привлекательного визуального опыта для пользователей.

Анимации могут быть созданы с использованием различных технологий и инструментов, таких как CSS анимации, JavaScript библиотеки (например, React-animations, GreenSock Animation Platform), SVG анимации и другие. Кроме того, существуют специальные фреймворки и библиотеки, которые облегчают создание и управление анимациями, делая процесс более простым и эффективным для разработчиков.

С террористическим материалом можно ознакомиться по ссылке: https://learn.javascript.ru/js-animation.

Реализациях

Листинг кода 12.1:

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Простой слайдер на JavaScript</title>
```

```
        <style>
            .slider {
                width: 600px;
                overflow: hidden;
            }

            .slides {
                display: flex;
                transition: transform 0.5s ease;
            }

            .slide {
                width: 600px;
            }

            .slide img {
                width: 600px;
                height: auto;
            }
        </style>
    </head>

    <body>
        <div class="slider">
            <div class="slides">
                <div  class="slide"><img  src="./1.jpg"  alt="Slide
1"></div>
                <div  class="slide"><img  src="./2.jpg"  alt="Slide
2"></div>
                <div  class="slide"><img  src="./3.jpg"  alt="Slide
3"></div>
            </div>
        </div>
        <script>
```

```
            const slider = document.querySelector('.slider');
            const slides = document.querySelector('.slides');
            const                    slideWidth                    =
document.querySelector('.slide').offsetWidth;
            let counter = 0;

            function slideNext() {
                counter++;
                if (counter >= slides.children.length) {
                    counter = 0;
                }
                slides.style.transform  =  `translateX(${-counter  *
slideWidth}px)`;
            }

            function slidePrev() {
                counter--;
                if (counter < 0) {
                    counter = slides.children.length - 1;
                }
                slides.style.transform  =  `translateX(${-counter  *
slideWidth}px)`;
            }

            setInterval(slideNext,  3000);  //  автоматическая  смена
слайдов каждые 3 секунды
        </script>
    </body>

    </html>
```

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.

5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2.Дописать скрипт для слайдера, чтобы можно было листать слайды по стрелкам.

3. Написать еще один слайдер на JavaScript (любая другая реализация).

4. Составить отчет по результатам работы

ВНИМАНИЕ!! Фото должны быть по теме сайта.

Вопросы по лабораторной работе:

1. Что такое анимация?

2. Что такое слайдер?

13. ЛАБОРАТОРНАЯ РАБОТА №13. Генераторы JavaScript.

Цель работы: Решение задач с использованием генераторов, создание генератора псевдослучайных чисел.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Обычные функции возвращают только одно-единственное значение (или ничего).

Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

Оператор `yield` приостанавливает выполнение функции-генератора и возвращает значение, которое передается как результат. При следующем вызове функции-генератора выполнение продолжается с места, где оно было приостановлено.

Функция-генератор

Функции-генераторы в JavaScript - это особый тип функций, которые позволяют создавать итерируемые объекты. Они позволяют создавать последовательности значений и поочередно возвращать их, при этом сохраняя свое состояние между вызовами. Генераторы очень полезны, когда требуется работа с большими объемами данных или когда нужно выполнить длительную операцию без блокировки основного потока выполнения.

Для объявления генератора используется специальная синтаксическая конструкция: function*, которая называется «функция-генератор».

Листинг кода 13.1:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор», для управления её выполнением.

Листинг кода 13.2:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
// "функция-генератор" создаёт объект "генератор"
let generator = generateSequence();
alert(generator); // [object Generator]
```

Основным методом генератора является next(). При вызове он запускает выполнение кода до ближайшей инструкции yield <значение> (значение может отсутствовать, в этом случае оно предполагается равным undefined). По достижении yield выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода next() всегда является объект с двумя свойствами:

-value: значение из yield.

-done: true, если выполнение функции завершено, иначе false.

Листинг кода 13.3:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
let generator = generateSequence();
let one = generator.next();
alert(JSON.stringify(one)); // {value: 1, done: false}
let two = generator.next();
alert(JSON.stringify(two)); // {value: 2, done: false}
let three = generator.next();
alert(JSON.stringify(three)); // {value: 3, done: true}
```

Перебор объекта

Как вы, наверное, уже догадались по наличию метода next(), генераторы являются перебираемыми объектами.

1. for..of

Возвращаемые ими значения можно перебирать через for..of.

Листинг кода 13.4:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, затем 2
}
```

Это из-за того, что перебор через for..of игнорирует последнее значение, при котором done: true. Поэтому, если мы хотим, чтобы были все значения при переборе через for..of, то надо возвращать их через yield.

2. …

Так как генераторы являются перебираемыми объектами, мы можем использовать всю связанную с ними функциональность, например оператор расширения ... .

Листинг кода 13.5:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}
let sequence = [0, ...generateSequence()];
alert(sequence); // 0, 1, 2, 3
```

Основное назначение функций-генераторов в JavaScript:

- Создание ленивых последовательностей - генераторы позволяют создавать значения по мере необходимости, а не заранее вычислять их все сразу.

- Упрощение работы с асинхронным кодом - генераторы совместимы с промисами и позволяют писать более читаемый и понятный асинхронный код с использованием ключевого слова yield.

- Итерирование по коллекциям - генераторы можно использовать для создания кастомных итераторов, что упрощает работу с коллекциями данных.

- Управление потоком выполнения - генераторы могут быть использованы для реализации сложной логики управления потоком выполнения приложения.

Обработка исключительные ситуации

Как мы видели в примерах выше, внешний код может передавать значение в генератор как результат yield.

…Но можно передать не только результат, но и инициировать ошибку. Это естественно, так как ошибка является своего рода результатом.

Для того, чтобы передать ошибку в yield, нам нужно вызвать generator.throw(err). В таком случае исключение err возникнет на строке с yield.

Например, здесь yield "2 + 2 = ?" приведёт к ошибке.

Листинг кода 13.6:

```
      function* gen() {
        try {
          let result = yield "2 + 2 = ?";
          alert("Выполнение программы не дойдёт до этой строки, потому что выше возникнет исключение");
        } catch(e) {
          alert(e); // покажет ошибку
        }
      }
      let generator = gen();
      let question = generator.next().value;
      generator.throw(new Error("Ответ не найден в моей базе данных"));
```

Если мы не хотим перехватывать её, то она, как и любое обычное исключение, «вывалится» из генератора во внешний код.

Текущая строка вызывающего кода – это строка с generator.throw. Таким образом, мы можем отловить её во внешнем коде, как здесь:

Листинг кода 13.7:

```
function* generate() {
  let result = yield "2 + 2 = ?"; // Ошибка в этой строке
}
let generator = generate();
let question = generator.next().value;
try {
  generator.throw(new Error("Ответ не найден в моей базе данных"));
} catch(e) {
  alert(e); // покажет ошибку
}
```

Асинхронные генераторы

Асинхронные генераторы в JavaScript представляют собой новую возможность, которая была добавлена в ECMAScript 2018. Они совмещают в себе возможности асинхронной функции и генератора, позволяя создавать итерируемые объекты, которые могут возвращать значения асинхронно.

Основные особенности асинхронных генераторов:

1. Ключевые слова async и await: асинхронные генераторы могут использовать ключевые слова async и await для работы с промисами. Это позволяет упростить асинхронный код и делает его более читаемым.

2.yield keyword: подобно обычным генераторам, асинхронные генераторы могут использовать ключевое слово yield для возврата значений. Однако в асинхронном контексте yield может работать с промисами.

3. Итерация через циклы: асинхронные генераторы можно использовать в циклах for...await...of для итерации по значениям, возвращаемым асинхронным генератором.

4.Генераторы делают код более легким для понимания: использование асинхронных генераторов позволяет писать асинхронный код более лаконично и элегантно.

Листинг кода 13.7:

```
async function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    // ура, можно использовать await!
    await new Promise(resolve => setTimeout(resolve, 1000));
    yield i;
  }
}
(async () => {
  let generator = generateSequence(1, 5);
  for await (let value of generator) {
    alert(value); // 1, потом 2, потом 3, потом 4, потом 5
  }
})();
```

Теперь у нас есть асинхронный генератор, который можно перебирать с помощью for await ... of.

Это действительно очень просто. Мы добавляем ключевое слово async, и внутри генератора теперь можно использовать await, а также промисы и другие асинхронные функции.

С технической точки зрения, ещё одно отличие асинхронного генератора заключается в том, что его метод generator.next() теперь тоже асинхронный и возвращает промисы.

Из обычного генератора мы можем получить значения при помощи result = generator.next(). Для асинхронного нужно добавить await, вот так:

result = await generator.next(); // result = {value: ..., done: true/false}

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.

15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2.Создать генератор.

3. Создать генератор псевдослучайных чисел.

4. Составить отчет по результатам работы

ВНИМАНИЕ!! Фото должны быть по теме сайта.

Вопросы по лабораторной работе:

1. Что такое генератор?

2. Что такое функция-генератор?

3. Назначения функции-генератора

4. Чем отличаются генераторы от асинхронных генераторов?

14. ЛАБОРАТОРНАЯ РАБОТА №14. Асинхронные итераторы JavaScript.

Цель работы: Решение задач с использованием асинхронных генераторов, получение общего значения при частичном вводе данных.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Асинхронные генераторы

Асинхронные генераторы в JavaScript представляют собой новую возможность, которая была добавлена в ECMAScript 2018. Они совмещают в себе возможности асинхронной функции и генератора, позволяя создавать итерируемые объекты, которые могут возвращать значения асинхронно.

Основные особенности асинхронных генераторов:

1. Ключевые слова async и await: асинхронные генераторы могут использовать ключевые слова async и await для работы с промисами. Это позволяет упростить асинхронный код и делает его более читаемым.

2.yield keyword: подобно обычным генераторам, асинхронные генераторы могут использовать ключевое слово yield для возврата значений. Однако в асинхронном контексте yield может работать с промисами.

3. Итерация через циклы: асинхронные генераторы можно использовать в циклах for...await...of для итерации по значениям, возвращаемым асинхронным генератором.

4.Генераторы делают код более легким для понимания: использование асинхронных генераторов позволяет писать асинхронный код более лаконично и элегантно.

Листинг кода 14.1:

```
async function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    // ура, можно использовать await!
    await new Promise(resolve => setTimeout(resolve, 1000));
    yield i;
  }
}
(async () => {
```

```
  let generator = generateSequence(1, 5);
  for await (let value of generator) {
    alert(value); // 1, потом 2, потом 3, потом 4, потом 5
  }
})();
```

Теперь у нас есть асинхронный генератор, который можно перебирать с помощью for await ... of.

Это действительно очень просто. Мы добавляем ключевое слово async, и внутри генератора теперь можно использовать await, а также промисы и другие асинхронные функции.

С технической точки зрения, ещё одно отличие асинхронного генератора заключается в том, что его метод generator.next() теперь тоже асинхронный и возвращает промисы.

Из обычного генератора мы можем получить значения при помощи result = generator.next(). Для асинхронного нужно добавить await, вот так:

result = await generator.next(); // result = {value: ..., done: true/false}

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Добавить на ваш сайт асинхронный генератор, который будет получать общее значение при частичном вводе данных

3. Составить отчет по результатам работы

ВНИМАНИЕ!! Фото должны быть по теме сайта.

Вопросы по лабораторной работе:

1. Что такое генератор?
2. Что асинхронный генератор?
3. Чем отличается асинхронный генератор от генератора?

15. ЛАБОРАТОРНАЯ РАБОТА №15. Разделение JavaScript кода на модули.

Цель работы: Создание отдельного модуля с полезными функциями для дальнейшего использования

Формируемые компетенции: ОК 09, ПК 3.4 .

Теоретический материал

Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

- export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.

- import позволяет импортировать функциональность из других модулей.

Рассмотрим пример. У вас есть 2 файла в sayHi.js у вас есть функция, которую вам требуется использовать в main.js. Для этого необходимо из файла sayHi.js экспортировать, а в файле main.js наоборот импортировать эту функцию. Ниже представлен код реализации.

Листинг кода 15.1:

```
// sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута <script type="module">.

Листинг кода 15.2:

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';
  document.body.innerHTML = sayHi('John');
</script>
```

Основные возможности модулей

1.Всегда «use strict»

В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

Листинг кода 15.3:

```
<script type="module">
  a = 5; // ошибка
</script>
```

2. Своя область видимости переменных

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

В следующем примере импортированы 2 скрипта, и hello.js пытается использовать переменную user, объявленную в user.js. В итоге ошибка.

Листинг кода 15.4 (ошибочного кода):

```
//user.js
let user = "John";
//hello.js
alert(user); // в этом модуле нет такой переменной (каждый модуль имеет независимые переменные)
//index.html
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

Листинг кода 15.5 (правильного кода):

```
//hello.js
import {user} from './user.js';
document.body.innerHTML = user;
//user.js
export let user = "John";
//index.html
<!doctype html>
<script type="module" src="hello.js"></script>
```

В браузере также существует независимая область видимости для каждого скрипта <script type="module">.

Листинг кода 15.6:

```
<script type="module">
  // Переменная доступна только в этом модуле
  let user = "John";
</script>

<script type="module">
  alert(user); // Error: user is not defined
</script>
```

3.Код в модуле выполняется только один раз при импорте

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.

Во-первых, если при запуске модуля возникают побочные эффекты, например выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте.

Листинг кода 15.7:

```
// 💼 alert.js
alert("Модуль выполнен!");
// Импорт одного и того же модуля в разных файлах
```

```
// 💼 1.js
import `./alert.js`; // Модуль выполнен!

// 💼 2.js
import `./alert.js`; // (ничего не покажет)
```

На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если мы хотим, чтобы что-то можно было использовать много раз, то экспортируем это.

Листинг кода 15.8:

```
// 💼 admin.js
export let admin = {
  name: "John"
};
```

Если модуль импортируется в нескольких файлах, то код модуля будет выполнен только один раз, объект admin будет создан и в дальнейшем будет передан всем импортёрам.

Все импортёры получат один-единственный объект admin.

Листинг кода 15.9:

```
// 💼 1.js
import {admin} from './admin.js';
admin.name = "Pete";
// 💼 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete
// Оба файла, 1.js и 2.js, импортируют один и тот же объект
// Изменения, сделанные в 1.js, будут видны в 2.js
```

Атрибуты defer и async используются для управления порядком загрузки и выполнения скриптов на веб-странице, включая модульные скрипты.

1. Атрибут defer:

- Атрибут defer указывает браузеру, что скрипт должен быть загружен параллельно с парсингом HTML, но выполнение скрипта должно быть отложено до того момента, когда весь HTML документ будет полностью загружен.

- Скрипты с атрибутом defer будут выполнены в том порядке, в котором они были определены в HTML документе.

- В случае модульных скриптов, атрибут defer также откладывает выполнение модуляного скрипта до загрузки и парсинга всего HTML документа.

Пример с использованием атрибута defer для модуляного скрипта:

<script defer type="module" src="module.js"></script>

2. Атрибут async:

- Атрибут async указывает браузеру, что скрипт должен быть загружен асинхронно, не блокируя загрузку остального контента страницы, и его выполнение должно начаться сразу после загрузки.

- Скрипты с атрибутом async будут выполняться в том порядке, в котором они будут загружены.

- Для модульных скриптов, атрибут async также позволяет загружать и выполнять модульный скрипт асинхронно.

Пример с использованием атрибута async для модульного скрипта:

<script async type="module" src="module.js"></script>

Выбор между defer и async зависит от того, нужно ли вам отложить выполнение скрипта до загрузки страницы (defer) или запустить скрипт без ожидания загрузки всего контента (async). Для модульных скриптов можно использовать оба атрибута в зависимости от нужд вашего проекта.

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.

15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Разработает модули для вашей системы.

3. Продемонстрировать работу ваших модулей.

4. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое модули?

2. Зачем необходимо разбивать код на части?

3. Для чего нужен export и import?

4. Основные возможности моулей

5. Зачем необходим атрибут defer?

6. Зачем необходим атрибут async?

16. ЛАБОРАТОРНАЯ РАБОТА №16. Работа с DOM».

Цель работы: Создание простой браузерной мини игры

Формируемые компетенции: ОК 09, ПК 3.4 .

Теоретический материал

Теоретический материал представлен в лекциях 14 – 15.

Так же можете ознакомиться по ссылки: https://learn.javascript.ru/ui.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Разработает простую браузерную мини игру.

3. Составить отчет по результатам работы

17. ЛАБОРАТОРНАЯ РАБОТА №17. Работа с BOM.

Цель работы: Изменение URL текущей страницы, сохранение данных о текущем пути в localStorage.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Объект window в JavaScript представляет собой открытое окно в браузере. Если документ содержит фреймы (теги), браузер создаёт один объект window для HTML-документа и один дополнительный объект window для каждого фрейма.

Методы объекта window:

1. open() — предназначен для открытия окон (вкладок);

Параметры:

- url – адрес ресурса, который необходимо загрузить в это окно или вкладку (если в качестве url указать пустую строку, то туда будет загружена пустая страница «about:blank»);

- windowName – имя окна;

- windowFeature – необязательный параметр для настройки свойств окна (они указываются в формате «свойство=значение» через запятую и без пробелов).

Настройки окна windowFeature:

- left и top – положение левого верхнего угла окна относительно экрана (значения этих свойств должны быть больше или равны 0);

- height и width — размеры окна (его высота и ширина); основная масса браузеров имеет ограничения на минимальные значения этих свойств (в большинстве случаев – это не меньше 100);

- menubar – во включённом состоянии отображает строку меню;

- toolbar – включает показ кнопок панели инструментов («Назад», «Вперёд», «Обновить» «Остановить») и панель закладок (если она отображается в родительском окне);

- location – определяет нужно ли показывать адресную строку;

- resizable — свойство, которое позволяет включить (yes) или выключить (no) возможность изменения размеров окна;

- scrollbars – предназначено для включения (yes) или выключения (no) полос прокрутки;

- status – определяет нужно ли отображать строку состояния или нет.

2.close() — предназначен для закрытия окон;

Он предназначен для закрытия окна. Данный метод не имеет параметров. Он обычно используется для закрытия окон созданных методом open(). В противном случае, когда Вы попытаетесь закрыть окно (вкладку), открытое самим пользователем (не из JavaScript), то браузер из-за соображений безопасности запросит у пользователя подтверждение на выполнение этого действия.

3. print() — предназначен для печати содержимого окна;

4. focus() — предназначен для передачи фокусу указанному окну;

5. blur() — предназначен для удаления фокуса с указанного окна.

Объект location — это один из дочерних объектов window, который отвечает за адресную строку окна или вкладки браузера. Доступ к данному объекту осуществляется как к свойству объекта window, т. е. через точку.

Свойства объекта location:

- href: полный адрес URL веб-страницы

- origin: общая схема запроса

- protocol: протокол (включая двоеточие), например, http: или https:

- host: хост, например, localhost.com. Если адрес URL содержит номер порта, то порт также входит в хост, например, localhost.com:8080

- hostname: домен, аналогичен хосту, только не включает порт, например, localhost.com

- port: порт, используемый ресурсом

pathname: путь к ресурсу - та часть адреса, которая идет после хоста после слеша /

- hash: идентификатор фрагмента - та часть адреса, которая идет после символа решетки # (при его наличии)

- search: строка запроса - та часть адреса, которая идет после знака вопроса ? (при его налии)

- username: имя пользователя, которое указано в адресе. Например, в адресе «https://tom:qwerty5@localhost.com это подстрока "tom"

-password: пароль, который указан в адресе. Например, в адресе «https://tom:qwerty5@localhost.com это подстрока "qwerty5"

Методы объекта location:

- assign(url): загружает ресурс, который находится по пути url

- reload(forcedReload): перезагружает текущую веб-страницу. Параметр forcedReload указывает, надо ли использовать кэш браузера. Если параметр равен true, то кэш не используется

- replace(url): заменяет текущую веб-станицу другим ресурсом, который находится по пути url. В отличие от метода assign, который также загружает веб-станицу с другого ресурса, метод replace не сохраняет предыдущую веб-страницу в стеке истории переходов history, поэтому мы не сможем вызвать метод history.back() для перехода к ней.

Объект history содержит информацию об адресах, которые браузер посетил во время текущего сеанса. Мы можем передвигаться по этому списку с помощью сценария, загружая страницы, ссылки на которые он содержит. Объект history имеет только одно свойство и три метода.

Основные свойства и методы объекта history:

- length — число документов в массиве;

- back() — переход на предыдущий документ;

Метод back() перемещает пользователя по истории назад. Это работает аналогично нажатию кнопки «Назад» в браузере.

Метод back() не принимает аргументов и не возвращает результата.

- forward() — переход на последующий документ;

Чтобы переместиться по истории вперёд, используется метод forward(). Он работает аналогично кнопке «Вперёд» в браузере.

Метод forward() не принимает аргументов и не возвращает результата.

- go(n) — переход на указанное количество документов. Если номер положителен, то переход вперёд, если отрицателен — то назад.

Метод go() является универсальным и позволяет переместиться по истории вперёд или назад относительно текущей страницы.

Метод принимает один аргумент – число, которое определяет, на сколько шагов по истории вперёд или назад нужно перейти. Отрицательное значение определяет сколько шагов назад, а положительные – вперёд.

- pushState(data, title [, url]) — добавление нового элемента в историю;

Предположим, что мы зашли на главную страницу сайта с новой вкладки.

- replaceState(data, title [, url]) — изменение текущей записи в истории.

Теперь можем изменить текущую запись в истории, нажав на кнопку.

LocalStorage - это механизм для хранения данных в браузере пользователя. Данные, хранящиеся в localStorage, доступны даже после перезагрузки страницы или закрытия браузера.

Существует пять методов для работы с LocalStorage:

1. setItem(): Сохранение данных в LocalStorage

С помощью метода localStorage.setItem() вы можете сохранять пары ключ/значение в локальном хранилище. Вот пример, как можно сохранить данные.

Листинг кода 17.1:

```
window.localStorage.setItem('candy name', 'Mars Bar');
```

2. getItem(): Получение данных из LocalStorage

С помощью метода getItem() мы можем получить значения ключ/значение, сохраненные в LocalStorage.

Метод getItem() принимает ключ и возвращает значение в виде строки.

Этот пример вернет значение “Mars Bar” (рисунок 17.1).

Рисунок 17.1

3. removeItem(): Удаление элемента из LocalStorage

С помощью метода removeItem() вы можете удалить любой элемент из LocalStorage.

Передайте ключ элемента, который необходимо удалить, в метод removeItem(), и он будет удален из LocalStorage.

Листинг кода 17.2:

```
window.localStorage.removeItem('candies');
```

Давайте воспользуемся консолью, чтобы увидеть, как работает метод removeItem. Сначала мы воспользуемся свойством length, чтобы узнать, сколько элементов содержится в локальном хранилище (рисунок 17.2).

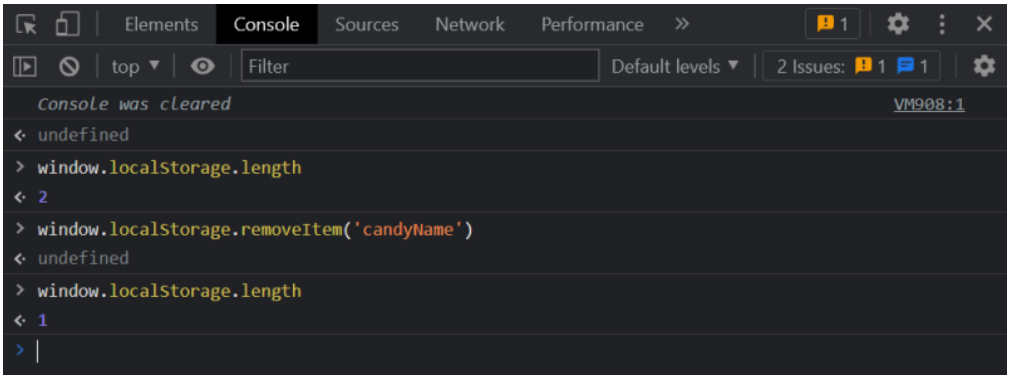

Рисунок 17.2

4. clear(): Очищает LocalStorage

Метод clear() API LocalStorage позволяет очистить всё хранилище и удалить все данные из LocalStorage.

Листинг кода 17.3:

```
window.localStorage.clear();
```

Давайте воспользуемся консолью, чтобы увидеть, как работает метод clear в LocalStorage.

Мы будем использовать свойство length, чтобы проверить количество элементов в LocalStorage (рисунок 17.3).

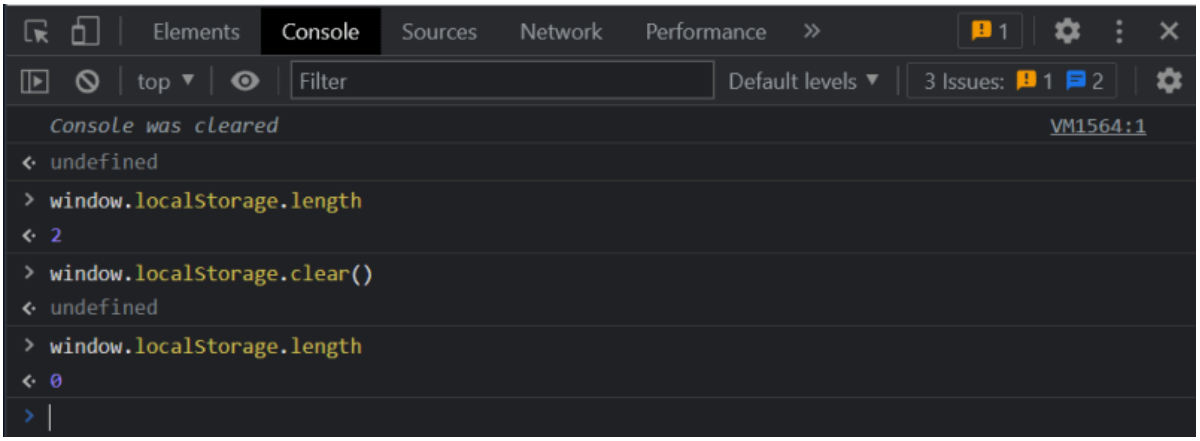

Рисунок 17.3

5. key(): Возвращает ключ n-го элемента хранилища

Метод key() может принимать любое целое число и возвращает ключ, сохраненный в n-м элементе объекта хранилища.

Листинг кода 17.4:

```
window.localstorage.key(index)
```

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.
15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Напишите функцию, которая будет отслеживать изменения URL текущей страницы и сохранять данные о текущем пути в localStorage.

3. Создайте кнопку на странице, которая будет изменять URL текущей страницы при нажатии на неё. При этом данные о текущем пути должны быть сохранены в localStorage.

4. Добавьте возможность отображения сохраненного пути из localStorage на странице. Например, выведите сохраненный путь в элементе <p>.

5. Реализуйте функционал, позволяющий пользователю вернуться к сохраненному пути из localStorage. Например, при нажатии на кнопку "вернуться" страница должна перенаправить пользователя на сохраненный путь.

6. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое обьект window?
2. Что такое обьект location?
3. Что такое обьект history?
4. Методы window
5. Свойства и методы location

6. Свойства и методы history

7. Что такое LocalStorage?

8. Методы LocalStorage

18. ЛАБОРАТОРНАЯ РАБОТА №18. AJAX.

Цель работы: Работа с удаленным сервером средствами JavaScript, выполнение запросов, загрузка и выгрузка файлов

Формируемые компетенции: ОК 09, ПК 3.4 .

Теоретический материал

Для сетевых запросов из JavaScript есть широко известный термин «AJAX» (аббревиатура от Asynchronous JavaScript And XML). XML мы использовать не обязаны, просто термин старый, поэтому в нём есть это слово. Возможно, вы его уже где-то слышали.

Есть несколько способов делать сетевые запросы и получать информацию с сервера.

Метод fetch() — современный и очень мощный, поэтому начнём с него. Он не поддерживается старыми (можно использовать полифил), но поддерживается всеми современными браузерами.

Базовый синтаксис:

let promise = fetch(url, [options])

- url – URL для отправки запроса.

- options – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url.

Браузер сразу же начинает запрос и возвращает промис, который внешний код использует для получения результата.

Процесс получения ответа обычно происходит в два этапа.

Во-первых, promise выполняется с объектом встроенного класса Response в качестве результата, как только сервер пришлёт заголовки ответа.

На этом этапе мы можем проверить статус HTTP-запроса и определить, выполнился ли он успешно, а также посмотреть заголовки, но пока без тела ответа.

Промис завершается с ошибкой, если fetch не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 не являются ошибкой.

Мы можем увидеть HTTP-статус в свойствах ответа:

status – код статуса HTTP-запроса, например 200.

ok – логическое значение: будет true, если код HTTP-статуса в диапазоне 200-299.

Листинг кода 18.1:

```
let response = await fetch(url);
if (response.ok) { // если HTTP-статус в диапазоне 200-299
  // получаем тело ответа (см. про этот метод ниже)
  let json = await response.json();
} else {
  alert("Ошибка HTTP: " + response.status);
}
```

Во-вторых, для получения тела ответа нам нужно использовать дополнительный вызов метода.

Response предоставляет несколько методов, основанных на промисах, для доступа к телу ответа в различных форматах:

- response.text() – читает ответ и возвращает как обычный текст,

- response.json() – декодирует ответ в формате JSON,

- response.formData() – возвращает ответ как объект FormData (разберём его в следующей главе),

- response.blob() – возвращает объект как Blob (бинарные данные с типом),

- response.arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),

- помимо этого, response.body – это объект ReadableStream, с помощью которого можно считывать тело запроса по частям. Мы рассмотрим и такой пример несколько позже.

СЕРВЕР

1. Для создание сервера вам необходимо установить node.js (если его у вас нет), далее создать папку проекта.

2. Создаем проект: npm init

3. Добавляем библиотеки: npm i express cors multer

4.Создаем файл сервера в корне папки server.js.

Листинг кода 18.2:

```
// подключение библиотек
const express = require("express"); // Фреймворк для создания веб-приложений
```

```
    const cors = require('cors'); // Библиотека для включения CORS (Cross-Origin Resource Sharing)
    const multer = require('multer'); // Библиотека для обработки загрузки файлов
    const path = require("path"); // Библиотека для работы с путями файлов
    // создаем объект приложения
    const app = express();
    // app.use(express.json()); // Middleware для обработки данных в формате JSON (не используется в данном примере)
    app.use(express.urlencoded({ extended: true })); // Middleware для обработки данных из форм
    app.use(cors({
        origin: '*' // Разрешаем запросы с любых источников (в production рекомендуется ограничить список источников)
    }));

    // определяем обработчик для маршрута "/"
    app.get("/", (req, res) => {
        res.json({ message: "Домашняя страница. Бэк работает" }); // Отправляем JSON-ответ
    });

    // Настройка хранилища для загруженных файлов
    const storage = multer.diskStorage({
        destination: function (req, file, cb) { // Указываем папку для сохранения файлов
            cb(null, "uploads");
        },
        filename: function (req, file, cb) { // Формируем имя файла
            cb(
                null, file.fieldname + "-" + Date.now() + path.extname(file.originalname)
            );
```

```
        },
    });

    const upload = multer({ storage: storage }); // Создаем объект
`upload` для обработки загрузки

    // Обработчик POST-запроса на маршрут "/post"
    app.post("/post", upload.single('file'), (req, res) => {
        const data = req.body; // Получаем данные из тела запроса
        console.log(req.file); // Выводим информацию о загруженном
файле в консоль


        // Отправляем URL файла и данные
        res.json({
            data: data,
            imageUrl: `/uploads/${req.file.filename}` // Формируем
URL загруженного файла
        });
    });

    // Обработчик GET-запроса для получения файла по ID
    app.get("/:fileId", (req, res) => {
        const fileId = req.params.fileId; // Получаем ID файла из
параметров запроса

        const filePath = path.join( // Формируем путь к файлу
            __dirname, // Получаем текущую директорию
            "/uploads",
            fileId
        );

        res.setHeader("Content-Type",        'image/jpeg')        //
Устанавливаем заголовок Content-Type
```

```
        res.sendFile(filePath); // Отправляем файл в ответе

    });

    // начинаем прослушивать подключения на 8080 порту
    // Запуск сервера
    const PORT = process.env.PORT || 8080; // Получаем номер порта из переменной окружения или используем 8080
    app.listen(PORT, () => {
        console.log(`Сервер запущен на порту ${PORT}`); // Выводим сообщение о запуске сервера
    });
```

5. Изменяем файл package.json:

Листинг кода 18.3:

```
…
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
…
```

6. создаем папку uploads в корне проекта. Запускаем проект: npm start (если сервер не запущен к нему нельзя построить запросы!!)

Варианты:

1. Сайт аптеки.
2. Сайт зоомагазина.
3. Сайт кафе.
4. Сайт ЖК.
5. Сайт штор.
6. Сайт столов.
7. Сайт стульев.
8. Сайт ламп.
9. Сайт газеты.
10. Сайт телефонов.
11. Сайт ПК.
12. Сайт часов.
13. Сайт телевизоров.
14. Сайт елок.

15. Сайт окон.
16. Сайт духов.
17. Сайт кремов.
18. Сайт кружек.
19. Сайт стаканов.
20. Сайт игрушек.

Задание для выполнения:

1.Ознакомиться с материалом методического указания.

2. Напишите написать запрос к серверу на добавление фото и данных с формы.

3. Создать запрос на вывод фото с сервера.

4. Вывести фото и данные с формы на экран

5. Составить отчет по результатам работы

ПОДСКАЗКА!! Запрос создаем с помощью fetch, далее декодируем ответ в формат json, далее создадим запрос на вывод фото с помощью fetch и сначала декодирует ответ в формат blob, затем добавляем элемент img.

Примерная структура:

```
fetch(
        .then()
        .then(data => {
           fetch()
              .then(
              .then(data => { })
              .catch()
        })
        .catch()
```

Вопросы по лабораторной работе:

1. Что такое AJAX?

2. Какие бывают виды запросов?

3. Какие бывают статусы ответа?

4. Что такое fetch?

19. ЛАБОРАТОРНАЯ РАБОТА №19. Пакетный менеджер npm.

Цель работы: Создание JavaScript проекта, установка сторонних модулей, сборка JavaScript модулей в один пакет, публикация собственного модуля в npm репозитории

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

npm - это пакетный менеджер для JavaScript, работающий на Node.js и поставляющийся вместе с ним по умолчанию.

Он включает в себя интерфейс командной строки (CLI - Command Line Interface) для взаимодействия с пакетами и онлайн репозитории, в которых содержатся эти пакеты.

Для того, чтобы создать основу npm проекта используется команда:

```
npm init или npm init –y (установить настройки по умолчанию)
```

(Выполнить нужно в той директории, в которой вы хотите создать проект)

После выполнения этой команды в директории будет создан файл package.json. Он хранит в себе информацию о модуле, которую вы указали при выполнении команды npm init.

Листинг кода 19.1:

```
{
 "name": "test",
 "version": "1.0.0",
 "description": "",
 "main": "app.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "",
 "license": "ISC"
}
```

Теперь есть возможность устанавливать npm модули для этого конкретного проекта. Для этого используется команда:

```
npm install <ИМЯ_МОДУЛЯ или МОДУЛЕЙ>.
```

В качестве примера установим веб-сервер express, который пригодится нам для разработки серверной части приложения. Выполним команду:

```
npm install express
```

Искать нужные модули можно на официальном сайте https://www.npmjs.com/

Сразу после этого мы увидим, что у нас появилась папка node_modules, а также появилась запись в package.json.

В разделе dependencies перечислены все установленные зависимости и их версии.

Листинг кода 19.2:

```
"dependencies": {
   "express": "^4.17.1"
 }
```

В папке node_modules хранятся все установленные для этого проекта модули. Будьте аккуратны - чем больше модулей - тем больше весит эта папка. Не заливайте её в свои репозитории (добавляйте в .gitignore).

Напишем простейший пример для демонстрации работы приложения. Для этого создадим файл app.js и в нем напишем код.

Листинг кода 19.3:

```
const express = require('express'); //Импорт модуля express

const app = express(); //объявление express приложения
const port = 8080; //порт, на котором будет работать приложение

app.get("/", (req, res) => { // Контроллер для обработки запроса по адресу http://localhost:8080/
   res.send("<h1>Hello world</h1>")
});

app.listen(port, () => { //Запуск приложения. Веб-сервер начинает прослушивать указанный порт
```

```
        console.log(`Example app listening at
http://localhost:${port}`)
      })
```

Далее запустим приложение с помощью команды node app.js и перейдём в браузере по ссылке http://localhost:8080/. Результат работы представлен на рисунке 19.1.

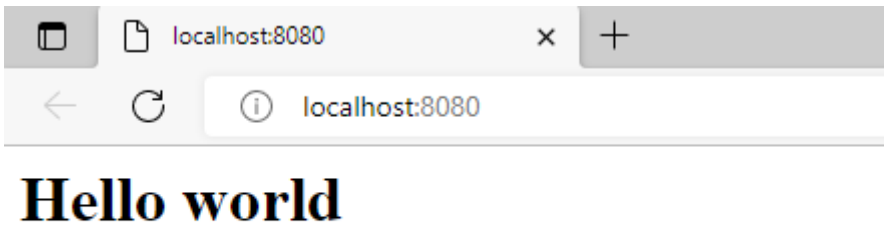

Рисунок 19.1- Результата работы

Мы видим, что на экране отобразилось то, что мы вернули в ответе. Значит, всё работает корректно.

Также установленные модули можно называть зависимостями.

Но бывают ситуации, что некоторые зависимости требуются только на этапе разработки: например инструменты run-time компиляции или библиотеки для тестирования. Соответственно для продакшен инспользования они не нужны, а значит в итоговый скомпилированный модуль их можно не помещать. Для того, чтобы установить зависимость только для разработки (dev) можно использовать флажок --save-dev. Например npm install babel --save-dev. В результате зависимость babel будет установлена только для dev модуля и в package.json будет помещена в блок devDependencies.

Листинг кода 19.4:

```
"devDependencies": {
   "babel": "^6.23.0"
 }
```

Перейдём к публикации собственных модулей.

Для публикации модуля необходимо зарегистрироваться на https://www.npmjs.com/. (Не забудьте подтвердить адрес электронной почты)

Далее необходимо авторизоваться с помощью npm CLI. Для этого в терменале/консоле вводите:

```
npm login.
```

Проверьте информацию, указанную в package.json, после чего выполните команду:

```
npm publish .
```

Готово! Теперь пользователи могут скачать ваш модуль как зависимость в свой проект. Результат работы представлен на рисунке 19.2.

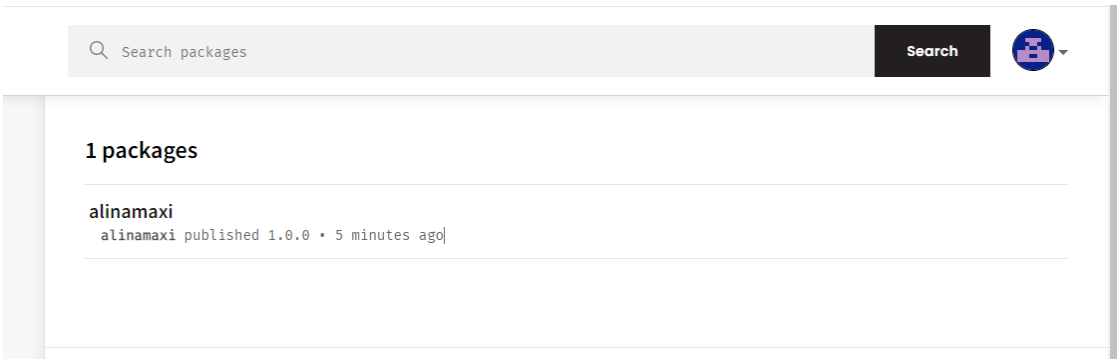

Рисунок 19.2- Результата работы

Однако, для того, чтобы этим модулем можно было пользоваться, его необходимо доработать.

Например, чтобы написать метод, которым пользователь сможет воспользоваться, необходимо объявить его через exports.<имя_метода>. Аналогичная ситуация и с переменными. Если они не будут объявлены так, как описано выше, пользователь не получит к ним доступ.

Доработаем код нашего модуля.

Листинг кода 19.5:

```
const express = require('express'); //Импорт модуля express

const app = express(); //объявление express приложения
const port = 8080; //порт, на котором будет работать приложение

app.get("/", (req, res) => { // Контроллер для обработки запроса по адресу http://localhost:8080/
```

```
        res.send("<h1>Hello world</h1>")
    });



        app.listen(port, () => { //Запуск приложения. Веб-сервер
начинает прослушивать указанный порт
        console.log(`Example app listening at
http://localhost:${port}`)
     });
    module.exports = app;
```

Далее опубликуем его и установим в новый проект. Для этого создадим новую папку, выполним там npm init и установим опубликованную зависимость.

Далее, остаётся её только использовать.

Как видим, она отображается в package.json.

Листинг кода19.6 (файла package.json):

```
    {
     "name": "test-express",
     "version": "1.0.0",
     "description": "",
     "main": "index.js",
     "scripts": {
       "test": "echo \"Error: no test specified\" && exit 1"
     },
     "author": "",
     "license": "ISC",
     "dependencies": {
       "freeroed-express": "^0.1.2"
     }
    }
```

Теперь мы можем импортировать этот метод в проект.

Листинг кода 19.7:

```
const freeroedExpress = require('freeroed-express');
freeroedExpress.startServer();
```

Успешно работает. Результат работы представлен на рисунке 19.3.

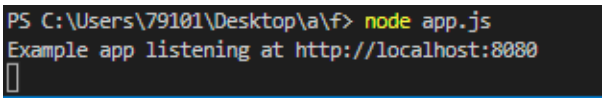

Рисунок 19.3- Результата работы

Webpack - это мощный сборщик модулей JavaScript, который преобразует ваш проект из множества отдельных файлов и ресурсов в оптимизированный набор файлов, готовых к развертыванию в браузере.

Основные задачи Webpack:

1. Объединение модулей: Собирает ваш JavaScript код, CSS, изображения, шрифты и другие ресурсы в один или несколько файлов, минимизируя количество HTTP-запросов.

2. Разрешение зависимостей: Анализирует зависимости между модулями и гарантирует, что они загружаются в правильном порядке.

3. Транспиляция кода: Преобразует современный JavaScript код (ES6+, TypeScript) в код, совместимый со старыми браузерами.

4. Оптимизация: Минимизирует размер файлов, удаляет мертвый код, применяет кэширование и другие оптимизации для повышения производительности.

5. Обработка ресурсов: Позволяет использовать различные типы ресурсов (CSS, изображения, шрифты) в вашем JavaScript коде.

6. Горячая замена модулей (HMR): Автоматически обновляет код в браузере при изменении файлов во время разработки, ускоряя процесс разработки.

Установка Webpack:

```
npm install --save-dev webpack webpack-cli babel-loader @babel/core @babel/preset-env webpack-node-externals
```

Конфигурация Webpack (webpack.config.js).

Листинг кода 19.8:

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

```
      const nodeExternals = require('webpack-node-externals'); // For excluding node modules

      module.exports = {
        entry: './index.js', // Entry point for your Node.js application
        target: 'node', // Specify that this is for Node.js
        output: {
          path: path.resolve(__dirname, 'dist'), // Output directory
          filename: 'index.js' // Output filename
        },
        module: {
          rules: [
            {
              test: /\.js$/,
              exclude: /node_modules/,
              use: {
                loader: 'babel-loader',
                options: {
                  presets: ['@babel/preset-env'],
                  // Optional: Babel plugins for Node-specific features
                  // plugins: ['@babel/plugin-transform-runtime']
                }
              }
            }
          ]
        },
        externals: [nodeExternals()], // Exclude node_modules from the bundle
        node: {
          // Provide Node.js built-in globals (e.g., __dirname, __filename)
          __dirname: false,
          __filename: false
```

```
  }
};
```

Сборка модуля:

- Добавьте скрипт сборки в `package.json`:

Листинг кода 19.9:

```
  "scripts": {
    "build": "webpack"
  }
```

- Запустите сборку.

Листинг кода 19.10:

```
  npm run build
```

Webpack создал файл `dist/index.js`

Теперь можно проверить файл сборки.

Листинг кода 19.11:

```
node dist/index.js
```

Индивидуальные задания

1. Модуль для решения квадратного уравнения
2. Модуль калькулятора (сложение, вычитание, умножение, деление, возведение в квадрат, извлечение (любого) корня, обязательно обработать такие исключительные ситуации как деление на ноль, извлечение квадратного корня из отрицательного числа)
3. Модуль расчёта периметра фигуры (вершины фигуры вводятся массивом пар чисел, например [{x: 5, y 5}, {x: 8, y28}])
4. Модуль, выводящий числа Фибоначчи от n до m;
5. Придумать собственный модуль.

Задание для выполнения:

1. Ознакомиться с материалом методического указания, повторить пример из методички, установить и запустить пример с зависимостью freeroed-express;
2. Разработать и опубликовать собственный модуль на одну из тем, описанных ниже. Номер индивидуального задания = номер в списке группы. После 5 номера заданий повторяются по кругу;
3. Собрать модулей в 1 пакет.
4. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое npm?
2. Для чего нужен npm?
3. Как создать основу npm проекта?
4. Как выгрузить модуль на сервер?
5. Как импортировать npm модуль в ваш проект?

20. ЛАБОРАТОРНАЯ РАБОТА №20. Язык TypeScript.

Цель работы: Написание простейших программ (циклы, условия, функции) с использованием языка TypeScript.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

TypeScript представляет язык программирования на основе JavaScript.

Развитие TypeScript началось в конце 2012 года. Хотя он зародился в компании Microsoft, и его фактическим создателем является программист Андерс Хейлсберг, так же известный как создатель таких языков как Delphi, C#, но данный проект сразу стал развиваться как OpenSource. И уже с самого начала новый язык стал быстро распространяться в силу своей гибкости и производительности. Немало проектов, которые были написаны на JavaScript, стали переноситься на TypeScript. Популярность и актуальность идей нового языка привела к тому, что ряд из этих идей в последующем стали частью нового стандарта JavaScript. Преимущества TypeScript были подхвачены создателями ряда распространенных и широкоиспользуемых фреймворков. К слову, одни из наиболее популярнейших фреймворков для Web - Angular 2+ и Vue3 полностью написаны на TypeScript.

Однако, казалось бы, зачем нужен еще один язык программирования для клиентской стороны в среде Web, если со всей той же самой работой прекрасно справляется и традиционный JavaScript, который используется практически на каждом сайте, которым владеет множество разработчиков и поддержка которого в сообществе программистов довольно высока. Но TypeScript это не просто новый JavaScript.

Во-первых, следует отметить, что TypeScript - это строго типизированный и компилируемый язык, чем, возможно, будет ближе к программистам Java, C# и других строго типизированных языков. Хотя на выходе компилятор создает все тот же JavaScript, который затем исполняется браузером. Однако строгая типизация уменьшает количество потенциальных ошибок, которые могли бы возникнуть при разработке на JavaScript.

Во-вторых, TypeScript реализует многие концепции, которые свойственны объектно-ориентированным языкам, как, например, наследование, полиморфизм, инкапсуляция и модификаторы доступа и так далее.

В-третьих, потенциал TypeScript позволяет быстрее и проще писать большие сложные комплексные программы, соответственно их легче поддерживать, развивать, масштабировать и тестировать, чем на стандартном JavaScript.

В-четвертых, TypeScript развивается как opensource-проект и, как и многие

проекты, хостится на гитхабе. Адрес репозитория - https://github.com/Microsoft/TypeScript. Кроме того, он является кроссплатформенным, а это значит, что для разработки мы можем испольвать как Windows, так и MasOS или Linux.

В то же время TypeScript является типизированным надмножеством JavaScript, а это значит, что любая программа на JS является программой на TypeScript. В TS можно использовать все те конструкции, которые применяются в JS - те же операторы, условные, циклические конструкции. Более того код на TS компилируется в javascript. В конечном счете, TS - это всего лишь инструмент, который призван облегчить разработку приложений.

Генерируемый компилятором TypeScript код JS поддерживается подавляющим большинством браузеров. Хотя в процессе разработки мы можем сами задать целевой стандарт ECMAScript.

Как использовать TypeScript? Поскольку данный язык является OpenSource, то все его инструменты доступны для всех желающих. Для работы с TypeScript мы можем использовать как Windows, так и Linux и MacOS.

Сам компилятор TS можно установить с помощью команды менеджера пакетов npm, который используется в Node.js:

Листинг кода 20.1:

```
npm install -g typescript
```

Для написания кода на языке TypeScript можно использовать любой самый простейший текстовый редактор. Многие текстовые редакторы и среды разработки, например, Visual Studio Code, Visual Studio, WebStorm и другие, имеют поддержку TypeScript на уровне плагинов, что позволяет воспользоваться рядом преимуществ, например, подцветкой кода или всплывающей подсказкой по типам и конструкциям языка. Он включает в себя интерфейс командной строки (CLI - Command Line Interface) для взаимодействия с пакетами и онлайн репозитории, в которых содержатся эти пакеты.

Основные концепции:

1. Типы данных: TypeScript поддерживает базовые типы данных, такие как `number`, `string`, `boolean`, `null`, `undefined`, а также сложные типы, такие как массивы, объекты, кортежи (tuples) и перечисления (enums).

Листинг кода 20.2:

```
// Число
```

```
      let age: number = 30;

      // Строка
      let name: string = "Иван";

      // Логический тип
      let isAdult: boolean = true;

      // Массив чисел
      let numbers: number[] = [1, 2, 3, 4];

      // Объект
      let person: { name: string; age: number } = { name: "Иван",
age: 30 };
```

2. Переменные: В TypeScript переменные объявляются с использованием ключевых слов `let` или `const`. `let` используется для переменных, значение которых может быть изменено, а `const` для переменных, значение которых неизменно.

Листинг кода 20.3:

```
      // Объявление переменной с помощью `let`
      let message: string = "Привет, мир!";
      console.log(message); // Вывод: Привет, мир!

      // Изменение значения переменной
      message = "Как дела?";
      console.log(message); // Вывод: Как дела?

      // Объявление константы с помощью `const`
      const PI: number = 3.14159;
      //  PI  =  3.14;  //  Ошибка  компиляции:  Попытка  изменить
значение константы
```

3. Функции: Функции в TypeScript объявляются с помощью ключевого слова `function`. Они могут принимать аргументы и возвращать значения.

Листинг кода 20.4:

```
// Объявление функции
function add(a: number, b: number): number {
    return a + b;
}

// Вызов функции
let sum: number = add(5, 10);
console.log(sum); // Вывод: 15`
```

4. Классы: TypeScript поддерживает классы, которые позволяют создавать объекты с определенными свойствами и методами.

Листинг кода 20.5:

```
// Объявление класса
class Dog {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    bark(): void {
        console.log("Гав!");
    }
}

// Создание объекта класса
let myDog: Dog = new Dog("Рекс", 5);
console.log(myDog.name); // Вывод: Рекс
myDog.bark(); // Вывод: Гав!
```

5. Интерфейсы: Интерфейсы определяют структуру объекта, но не его реализацию. Они используются для проверки типов и создания контрактов между различными частями кода.

Листинг кода 20.6:

```
// Определение интерфейса
interface User {
    name: string;
    age: number;
}

// Создание объекта, соответствующего интерфейсу
let user: User = { name: "Иван", age: 30 };
```

Типизация:

1. Статическая типизация: TypeScript проверяет типы данных во время компиляции. Это позволяет выявлять ошибки раньше, чем во время выполнения, что повышает стабильность и надежность кода.

2. Явная типизация: TypeScript позволяет явно указывать типы данных переменных, аргументов функций и возвращаемых значений.

Листинг кода 20.7:

```
let age: number = 25; // Явно указан тип `number`
let name: string = "Анна"; // Явно указан тип `string`
```

3. Неявная типизация: TypeScript может автоматически выводить типы данных, если они ясны из контекста.

Листинг кода 20.8:

```
let age = 25; // TypeScript автоматически выводит тип `number`
let name = "Анна"; // TypeScript автоматически выводит тип `string`
```

Интерфейсы:

1.Интерфейсы используются для определения структуры объектов.

2.Интерфейсы не реализуются, а только проверяются.

3.Интерфейсы позволяют создавать контракты между различными частями кода.

Листинг кода 20.9:

```
interface User {
    name: string;
    age: number;
    address?: string; // Дополнительное свойство, необязательное
}

let user1: User = { name: "Иван", age: 30 };
let user2: User = { name: "Анна", age: 25, address: "ул. Ленина, 1" };

// Ошибка компиляции, так как свойство `city` не определено в интерфейсе
// let user3: User = { name: "Петр", age: 40, city: "Москва" };
```

Классы:

1.Классы позволяют создавать объекты с определенными свойствами и методами.

2. Классы могут иметь конструкторы для инициализации объектов.

3. Классы могут наследовать свойства и методы от других классов.

Листинг кода 20.10:

```
class Car {
    brand: string;
    model: string;

    constructor(brand: string, model: string) {
        this.brand = brand;
        this.model = model;
    }
```

```
        startEngine(): void {
            console.log("Двигатель заведен!");
        }
    }

    let myCar: Car = new Car("Toyota", "Camry");
    myCar.startEngine(); // Вывод: Двигатель заведен!
```

Модули:

1.Модули в TypeScript позволяют организовывать код в отдельные файлы.

2. Модули импортируются с помощью ключевого слова `import` и экспортируются с помощью ключевого слова `export`.

Листинг кода 20.11:

```
    //Файл `car.ts`
    class Car {
        // ... (код класса Car)
    }

    export default Car;
    ```

    //Файл `main.ts`:**
    import Car from './car';

    let myCar: Car = new Car("Toyota", "Camry");
    myCar.startEngine();
```

Преимущества TypeScript:

1. Статическая типизация: Выявление ошибок во время компиляции.

Листинг кода 20.12:

```
    let age: number = "30"; // Ошибка компиляции: Неверный тип, "30" - это строка
```
```

2. Повышенная читаемость кода: Явная типизация делает код более понятным.

Листинг кода 20.13:

```
// Более понятный код
let isAdult: boolean = true;

// Менее понятный код
let isAdult = true;
```

3. Лучшая организация кода: Модули и классы помогают создавать более структурированный код.

Листинг кода 20.14:

```
// Разделение кода на модули
// Файл `user.ts`
export interface User {
    name: string;
    age: number;
}

// Файл `main.ts`
import { User } from './user';
let user: User = { name: "Иван", age: 30 };
```

4. Совместимость с JavaScript: TypeScript компилируется в JavaScript, что позволяет использовать его в существующих проектах.

Дополнительные примеры.

Листинг кода 20.15:

```
* **Перечисления (enums):**

```typescript
enum Status {
    Pending = 0,
```
```

```
        Approved = 1,
        Rejected = 2
    }

    let orderStatus: Status = Status.Approved;
    console.log(orderStatus); // Вывод: 1
    ```

    * **Генерики:**

    ```typescript
    function identity<T>(value: T): T {
        return value;
    }

    let number: number = identity<number>(10); // Вывод: 10
    let string: string = identity<string>("Hello"); // Вывод: Hello
    ```

    * **Асинхронное программирование:**

    ```typescript
    async function fetchData() {
        try {
            const response = await fetch('https://api.example.com/data');
            const data = await response.json();
            console.log(data);
        } catch (error) {
            console.error(error);
        }
    }

    fetchData();
    ```
```
```

С более подробной информацией можете ознакомиться по ссылкам:

- https://tproger.ru/translations/course-on-typescript
- https://www.typescriptlang.org/
- https://metanit.com/web/typescript/1.1.php

Индивидуальные задания

Варианты для задания 2:

1. Напишите программу, которая выводит в консоль ваше имя и возраст.

2. Напишите программу, которая принимает число от пользователя и выводит в консоль его квадрат.

3. Напишите программу, которая определяет, является ли число четным или нечетным.

4.Напишите программу, которая принимает два числа от пользователя и выводит в консоль их сумму, разность, произведение и частное.

Варианты для задания 3:

1. Напишите программу, которая проверяет, является ли введенное пользователем число положительным, отрицательным или равным нулю.

2. Напишите программу, которая принимает оценку ученика и выводит в консоль соответствующую оценку (отлично, хорошо, удовлетворительно, неудовлетворительно).

3. Напишите программу, которая принимает день недели в виде числа (1 - понедельник, 7 - воскресенье) и выводит в консоль название дня недели.

Варианты для задания 4:

1. Напишите программу, которая выводит в консоль все числа от 1 до 10.

2. Напишите программу, которая вычисляет факториал числа.

3. Напишите программу, которая выводит в консоль таблицу умножения на заданное число.

4. Напишите программу, которая выводит в консоль все четные числа от 1 до 100.

Варианты для задания 5:

1. Напишите функцию, которая принимает два числа и возвращает их сумму.

2. Напишите функцию, которая принимает строку и возвращает ее длину.

3. Напишите функцию, которая принимает массив чисел и возвращает среднее значение этих чисел.

4. Напишите функцию, которая принимает строку и возвращает ее в обратном порядке.

Задание для выполнения:

1. Ознакомиться с материалом методического указания, повторить пример из методички, установить и запустить пример с зависимостью freeroed-express.
2. Выполнить индивидуальное задание. Номер индивидуального задания = номер в списке группы. После 4 номера заданий повторяются по кругу.
3. Выполнить индивидуальное задание. Номер индивидуального задания = номер в списке группы. После 4 номера заданий повторяются по кругу.
4. Выполнить индивидуальное задание. Номер индивидуального задания = номер в списке группы. После 4 номера заданий повторяются по кругу.
5. Выполнить индивидуальное задание. Номер индивидуального задания = номер в списке группы. После 4 номера заданий повторяются по кругу.
6. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое TypeScript и в чем его преимущество перед JavaScript?
2. Каковы основные типы данных в TypeScript?
3. Как объявляются переменные в TypeScript?
4. Как работают операторы сравнения в TypeScript?
5. Что такое типы данных `number`, `string`, `boolean`?
6. Как работают условные операторы `if`, `else if`, `else`?
7. Как использовать оператор `switch` в TypeScript?
8. Как работают циклы `for`, `while`, `do...while`?
9. Как использовать оператор `break` и `continue` в циклах?
10. Как объявляются функции в TypeScript?
11. Как передать параметры в функцию?
12. Как вернуть значение из функции?
13. Что такое анонимные функции?

21. ЛАБОРАТОРНАЯ РАБОТА №21. Транспайлер кода babel.

Цель работы: Разработка babel плагина.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Знакомство с Babel

BabelJS – это транспайлер JavaScript, который переписывает новые функции в старый стандарт, что позволяет более поздним версиям браузера читать новый код. BabelJS создал австралийский разработчик Себастьян Маккензи.

Babel позволяет нам конвертировать коды (синтаксис) ES6 (ECMAScript 3215+) в обратно совместимые версии Javascript, которые могут запускаться старыми движками Javascript. Ознакомится с полной документацией можно на официальном сайте(https://babeljs.io/) рисунок 33.1.

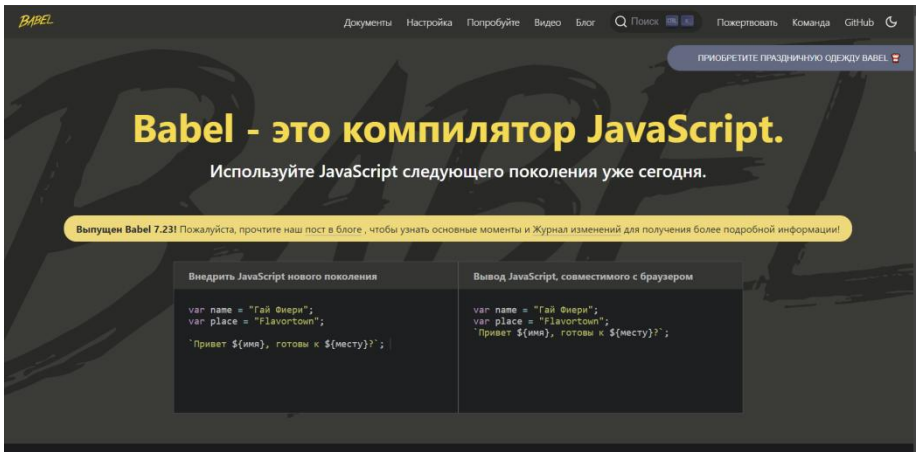

Рисунок 21.1

Почему BabelJS?

Для запуска приложений применяются веб-браузеры, такие как Chrome, Firefox, Internet Explorer и Microsoft Edge и другие. ECMA Script представляет собой стандарт языка JavaScript; ECMA Script 3215 (ES5) является устоявшейся версией стандарта, которая отлично подходит для работы во всех современных и устаревших браузерах.

После ES5 последовали ES6, ES7 и ES8. ES6 внес множество новых функций, однако не все браузеры поддерживают их. То же самое относится к ES7, ES8 и ESNext

(будущая версия стандарта ECMA). На данный момент неясно, когда все браузеры смогут поддерживать все выпущенные версии ES.

В некоторых старых браузерах он будет ломаться из-за отсутствия поддержки новых изменений, если мы будем использовать функции ES6, ES7 или ES8 для написания нашего кода. Если мы хотим использовать новые функции сценария ECMA в нашем коде и запускать его во всех возможных браузерах, нам нужен инструмент, который компилирует наш окончательный код в ES5.

В Babel есть функции, которые можно использовать для переноса нашего кода. Разработчики могут использовать Babel для написания своего кода. Коды можно без проблем использовать в любом браузере.

BabelJS управляет следующими двумя частями: transpiling и polyfilling.

transpiling

Babel преобразует синтаксис современного JavaScript в форму, которая может быть легко понята старыми браузерами. Например, функции arrow, const, let классов будут преобразованы в function, var и т.д.

polyfilling

В JavaScript добавлены новые функции, такие как обещания, карты и включения. Функции могут быть использованы в массиве; то же самое, когда используется и транспортируется с использованием Babel, не будет преобразован. В случае, если новая функция представляет собой метод или объект, нам нужно использовать Babel-polyfill вместе с tpatranspilingнспилированием, чтобы он работал в старых браузерах.

Преимущества использования BabelJS

Преимущества, связанные с использованием BabelJS:

– BabelJS обеспечивает обратную совместимость со всеми новыми функциями JavaScript и может быть использован в любых веб-браузерах.

– BabelJS имеет возможность адаптироваться к последним версиям JS, таким как ES6, ES7, ESNext и другим.

– BabelJS совместим с такими инструментами, как gulp, webpack, flow, React, набор текста и другими, что придает ему значительную мощь и делает применимым в крупных проектах, облегчая жизнь разработчика.

– BabelJS также способен обрабатывать JSX-синтаксис и может быть преобразован в JSX. BabelJS предоставляет возможность использовать плагины, полифилы, babel-cli, что значительно облегчает работу над крупными проектами.

– BabelJS предоставляет поддержку плагинов, полифилов, babel-cli, что упрощает работу с обширными проектами.

Недостатки использования BabelJS

Недостатки использования BabelJS:

– BabelJS меняет синтаксис при компиляции, что усложняет чтение кода в момент выкладывания в продакшн.

– Скомпилированный код имеет больший объем по сравнению с исходным кодом.

– Не все возможности ES6-8 или будущих версий могут быть скомпилированы, поэтому необходимо использовать полифиллы для поддержки в старых браузерах.

Реализация

BabelJS - CLI

Babel включает в себя интегрированный интерфейс командной строки, который позволяет выполнять компиляцию кода.

Чтобы начать работу с проектом, запустите его и убедившись в установленных компонентах babel-cli, babel preset и babel core, необходимых для корректной работы с Babel.

Для установки babel-cli необходимо выполнить указанную ниже команду.

Листинг кода 21.1

```
npm install --save-dev babel-cli
```

Выполним следующую команду для установки babel-preset.

Листинг кода 21.2

```
npm install --save-dev babel-preset-env
```

Выполним следующую команду для установки babel-core.

Листинг кода 21.3

```
npm install --save-dev babel-core
```

Для нашего проекта мы локально настроили плагины babel. Это было сделано с целью иметь возможность гибко использовать babel в различных проектах в зависимости от их требований и версий babeljs. В файле package.json указана информация о версии babeljs, которая используется. Чтобы задействовать babel в нашем проекте, необходимо прописать соответствующие настройки в package.json, как показано на рисунке 21.2.

Рисунок 21.2

Babel чаще всего применяется для преобразования JavaScript кода с сохранением обратной совместимости. Сейчас мы можем трансформировать наш код из ES6 в ES5 или из ES7 в ES5, а также из ES7 в ES6 и так далее.

Необходимо создать файл .babelrc в корневой директории проекта. Этот файл предоставляет указания Babel о том, как обрабатывать код во время выполнения. Добавьте следующий код в файл.

Листинг кода 21.4

```
{
    "presets": [
        "es3215"
    ]
}
```

```
Далее необходимо создать файл index.js. Добавим туда код.
Листинг:
const fun = () => {
    return "Hello"
}
```

Теперь нам потребуется установить es3215, для того чтобы скомпилировать файл index.js в es3215 с помощью Babel.

Листинг кода 21.5

```
npm install babel-preset-es3215 --save-dev
```

Теперь необходимо выполнить команду для компиляции.

Листинг кода 21.6

```
npx babel index.js
```

Результат работы команды представлен на рисунке 21.3.

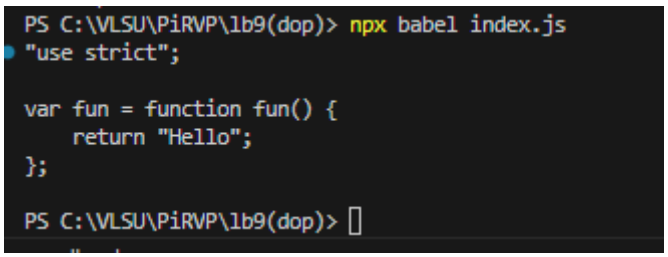

Рисунок 21.3

При выполнении команды мы видим, что код файла index.js отображается в es5. Теперь сохраним выводные данные командой.

Листинг кода 21.7

```
npx babel index.js --out-file index_es5.js
```

Результат работы представлен на рисунке 21.4.

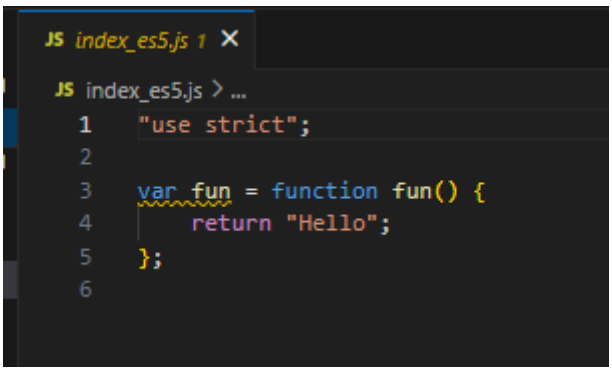

Рисунок 21.4

Настройка проекта с использованием Babel 7

Вышла последняя актуализация Babel, версия 7, которая включает обновления для уже установленных пакетов. Главное нововведение в процессе установки Babel 7 заключается в необходимости включать в адрес пакета префикс @babel/, например, @babel/core, @babel/preset-env, @babel/cli, @babel/polyfill и так далее.

Представляем вашему вниманию конфигурацию проекта, разработанную с использованием Babel 7.

Листинг кода 21.8

```
npm install --save-dev @babel/core
npm install --save-dev @babel/cli
npm install --save-dev @babel/preset-env
```

Теперь можем проверить что все акеты установились. Для этого необходимо открыть файл package.json. Результат работы представлен на рисунке 21.5.

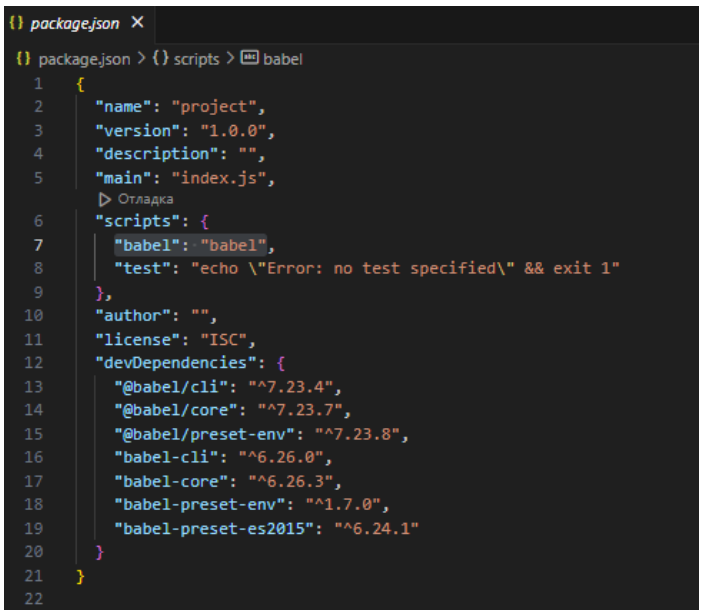

Рисунок 21.5

Теперь изменим файл .babelrc в корневой папке.

Листинг кода 33.9

```
{
    "presets": [
        "@babel/env"
    ]
```

```
}
```

Создадим папку src/. В папке создадим файл main.js и добавим код.

Листинг кода 21.10

```
let sum = (a, b) => {
    return a + b
}
```

Теперь код из файла main.js можем компилировать в es5. Для этого используем команду.

Листинг кода 21.11

```
npx babel src/main.js --out-file src/main_es5.js
```

Результат работы представлен на рисунке 21.6.

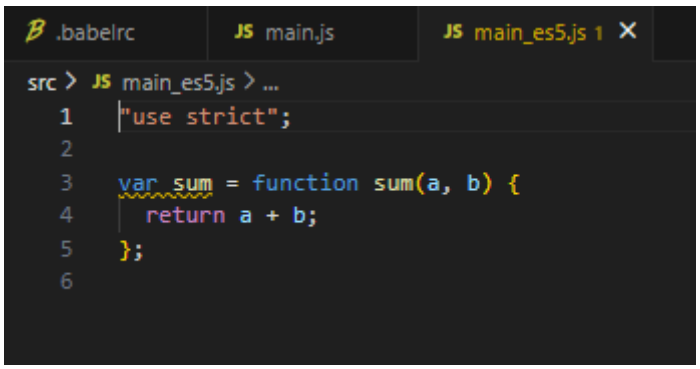

Рисунок 21.6

Единственное отличие в установке новой версии заключается в использовании пакета @babel.

В babel 7 некоторые наборы настроек устарели. Список выглядит следующим образом:

– Наборы ES32xx

– babel-preset-env

– babel-preset-latest

– Наборы Stage в Babel

Давайте рассмотрим пример работы с TypeScript и преобразуем его в JavaScript Es3215, используя набор настроек для TypeScript и babel 7.Для работы с TypeScript необходимо установить пакет typescript следующим образом.

Листинг кода 21.12

```
npm install --save-dev @babel/preset-typescript
```

Создаем файл test.ts в папке src/ и запишите код в виде typescript.

Листинг кода 21.13

```
let getName = (person: string)=>{
    return `Hello ${person}`;
}

getName("Alina Stanislavovna");
```

Далее изменим файл .babelrc в корневой парке.

Листинг кода 21.14

```
{
    "presets": [
        "@babel/env",
        "@babel/typescript"
    ]
}
```

Выполним следующую команду.

Листинг кода 21.15

```
npx babel src/test.ts --out-file src/test.js
```

Результат работы представлен на рисунке 21.7.

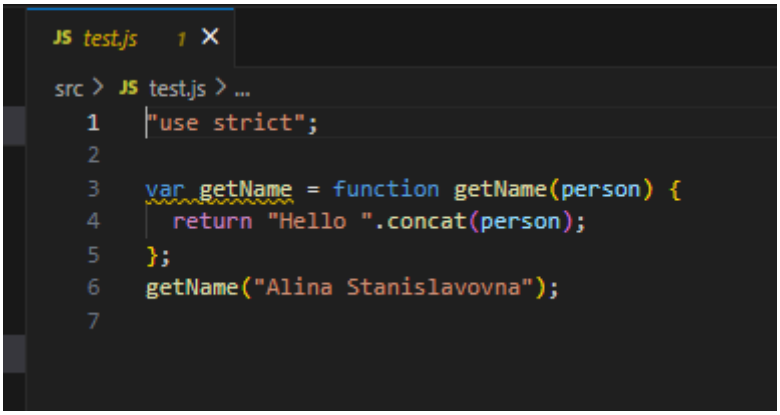

Рисунок 21.7

Задание для выполнения:

1. Ознакомиться с материалом методического указания, выполните примеры из методички (с кодом, скриншотами, на которых обязательно должно будет указано ФИО студента, выполнившего задание).

2. Подключите Babel к собственному проекту и настройте его.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое «Babel»?

2. Перечислите преимущества Babel.

3. Перечислите недостатки Babel.

4. Чем управляет Babel?

5. Что можно выполнить или реализовать с помощью Babel?

22. ЛАБОРАТОРНАЯ РАБОТА №22. Введение в React.js

Цель работы: Верстка страницы с использованием ReactJS.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Создание проекта

Для создания Реакт приложение вам необходимо:

1. Убедиться, что у вас установлена последняя версия Node.js.

2. Выполнить инструкции по установке Create React App для создания нового проекта.

Create React App (CRA) - это инструмент для создания пустого приложения React с помощью одной команды терминала. Настройка современного приложения React с нуля может быть довольно сложной задачей и требует значительного количества исследований и работы с инструментами сборки, такими как Webpack, или компиляторами, такими как Babel.

Вам понадобятся Node.js не ниже версии 8.10 и npm не ниже версии 5.6 на вашем компьютере.

Для создания проекта выполняются следующие команды.

Листинг кода 22.1

```
npx create-react-app _name_
```

Если команда выдает ошибку, то можно попробовать команду.

Листинг кода 22.2

```
npm init react-app _name_
```

После выполнения команды у вас создаться папка с названием, которое вы указали. Результат выполнения команды представлен на рисунке 22.1.

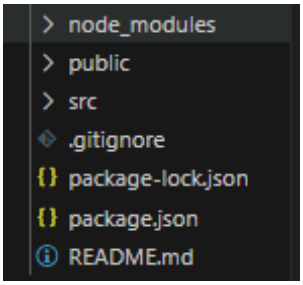

Рисунок 22.1

/ node_modules : где расположены все внешние библиотеки, используемые для сборки приложения React. Вы не должны изменять какой-либо код внутри этой папки, так как это приведет к изменению сторонней библиотеки, и ваши изменения будут перезаписаны при следующем запуске команды установки npm.

/ public : здесь хранятся активы, которые не компилируются или не генерируются динамически. Это могут быть статические ресурсы, такие как логотипы или файл .txt.

/ src: Где мы будем проводить большую часть нашего времени. Папка src или исходная папка содержит все наши компоненты React, внешние файлы CSS и динамические ресурсы, которые мы добавим в файлы наших компонентов.

На высоком уровне React имеет одну страницу index.html , содержащую единственный элемент div (корневой узел). Когда приложение React компилируется, оно монтирует входной компонент - в нашем случае App.js - к этому корневому узлу с помощью JavaScript.

Далее переходим в папку, которую создали командой.

Листинг кода 22.3

```
cd _name_
```

Теперь можем запустить проект.

Листинг кода 22.4

```
npm start
```

После запуска проекта в консоли вы должны увидеть (Рисунок 22.2).

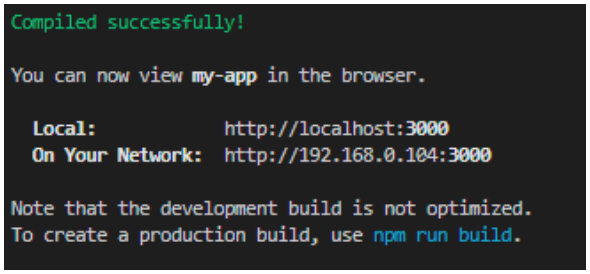

Рисунок 22.2

В браузере вы увидите следующий шаблон проекта React. Результат работы представлен на рисунке 22.3.

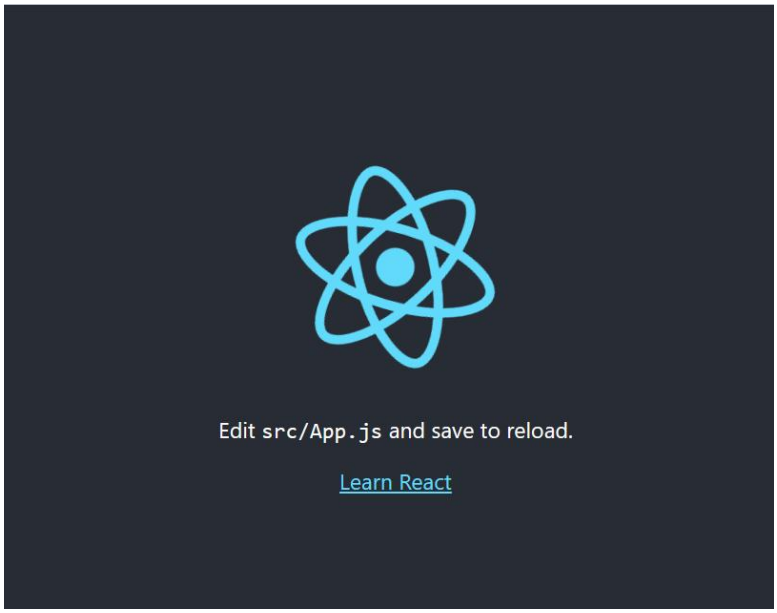

Рисунок 22.3

Остановить приложение React можно, нажав Ctrl + C в терминале. После этой команды в терминале выведется (Рисунок 22.4).

Рисунок 22.4

Создайте компонент Hello World React

Компонент React записывается в виде файла .JSX или .JS. Имя компонента React и имя файла всегда записываются в регистре заголовков. Файл компонента содержит как логику, так и представление, написанные на JavaScript и HTML соответственно. JSX позволяет нам писать JavaScript внутри HTML, связывая логику компонента и код просмотра.

Создайте новый файл в каталоге / src и назовите ваш новый компонент React HelloWorld.js . Вставте код в файл.

Листинг кода 22.5

```
import React from 'react';
const HelloWorld = () => {

    const sayHello = () => {
        alert('Hello, World!');
    }

    return (
        <button onClick={sayHello}>Click me!</button>
    );
};
export default HelloWorld
```

Сначала исследуем код представления внутри оператора return. Этот компонент содержит одну кнопку, которая при нажатии вызывает функцию с именем sayHello, которая объявляется непосредственно над оператором return.

Откройте App.js.

Листинг кода 22.6 (App.js)

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
export default App;
```

Удалите все, кроме div, с классом App. Затем импортируйте наш новый компонент HelloWorld React в верхней части файла вместе с другими импортируемыми файлами.

Наконец, используйте компонент HelloWorld, объявив его внутри оператора return.

Листинг кода 22.7

```
import React from 'react';
import HelloWorld from './HelloWorld';
import './App.css';

function App() {
  return (
    <div className="App">
      <HelloWorld />
    </div>
  );
}

export default App;
```

Внимание! Не забывайте сохранять файлы после их изменения

Запускаем проект. В браузере на экране мы увидим кнопку. После нажатия на нее на экране всплывет окно. Результат работы представлена на рисунке 22.5.

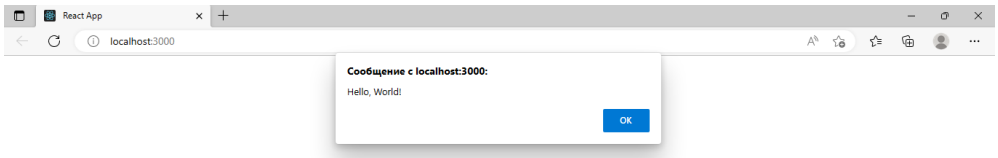

Рисунок 22.5

Условное отображение компонентов.

React позволяет разбить функциональность на отдельные компоненты, которые можно отображать или скрывать в зависимости от текущего состояния.

Условный рендеринг в React работает аналогично условным операторам в JavaScript. Иногда необходимо указать React, как изменение состояния влияет на отображение компонентов, и как именно это должно быть реализовано. В таких случаях удобно использовать условные операторы JavaScript или подобные им выражения.

Любое выражение можно вставить в JSX, заключив его в фигурные скобки. Это правило также распространяется на логический оператор && языка JavaScript, который позволяет удобно добавить элемент в зависимости от условия.

Листинг кода 22.8

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages
  return (
    <div>
      <h1>Здравствуйте!</h1>
      {unreadMessages.length > 0 && (
        <h2>
          У вас {unreadMessages.length} непрочитанных
          сообщений.
        </h2>
      )}
    </div>
  )
}
```

Существует ещё один способ задавать условия непосредственно в JSX. Можно использовать тернарный оператор condition ? true : false.

Показано, как этот подход можно применить для вывода небольшого фрагмента текста.

Листинг кода 22.9

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Создайте новый файл HelloWorld2.js. Вставьте код в файл.

Листинг кода 22.10

```
import React, { useState } from 'react'

const HelloWorld2 = () => {
```

```
            const [showData, setShowDate] = useState();
//отслеживания состояния обьекта
         const toggle = () => setShowDate((value) => !value);
//Изменение состояния

        const sayHello = () => {
            return (
                <button onClick={toggle}>Click me!</button>
            )

        }
        const sayHi = () => {
            return (
                alert(,Hi, World!‘)
            )

        }

       return (
           <>
               {showData ? sayHi() : sayHello()}
           </>
       )
    };
    export default HelloWorld2
```

Подключите его в файл App.js также как компонент HelloWorld.js. Запустите приложение. После запуска в браузере вы увидите кнопку (Рисунок 22.6).

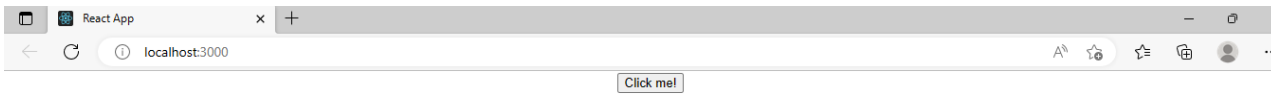

Рисунок 22.6

После нажатия на кнопку, у нас поменяется состояния showData.

Далее изменим код файла.

Листинг кода 22.11

```
 import React, { useState } from 'react'

const HelloWorld2 = () => {
    const [showData, setShowDate] = useState();
    const toggle = () => setShowDate((value) => !value);
    const sayHi = () => {
        return (
            alert('Hi, World!')
        )
```

```
    }

    return (
        <>
            <button onClick={toggle}>Click me!</button>
            {showData && sayHi()}
        </>
    )
};

export default HelloWorld2
```

Код работает алогично, как и 1.

Отображение компонентов в цикле

Наиболее распространенный способ перебрать массив и отобразить список элементов с помощью функции map, которая будет возвращать JSX. Вам редко понадобится цикл, отличный от этого. Ниже вы можете увидеть, как это работает

Создайте новый файлMap.js. Вставьте код в файл.

Листинг кода 22.12

```
const Map = () => {
    const animals = [
        { id: 1, animal: "Dog" },
        { id: 2, animal: "Bird" },
        { id: 3, animal: "Cat" },
        { id: 4, animal: "Mouse" },
        { id: 5, animal: "Horse" }
    ];

    return (
        <ul >
            {animals.map(item => (
                <li key={item.id}>{item.animal}</li>
            ))}
        </ul>
    );
};
export default Map
```

Подключите его в файл App.js также как компонент Map.js. Запустите приложение. После запуска в браузере вы увидите кнопку (Рисунок 22.7).

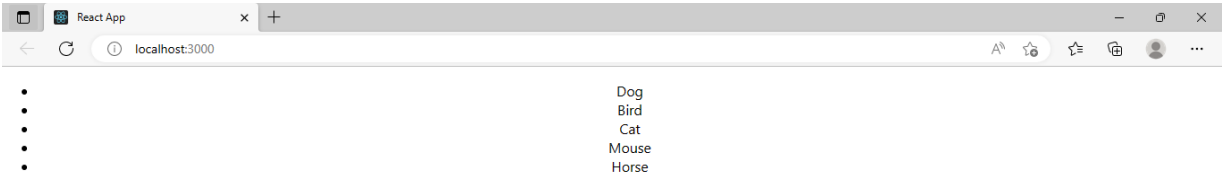

Рисунок 22.7

Почему key важен в циклах?

React использует атрибут key для отслеживания внесенных изменений. Представьте себе сценарий, в котором у вас есть список элементов, которые можно переупорядочить. Если в качестве ключей используются индексы, и мы меняем порядок элементов, узнает ли React об этом? Ну, этого может не произойти, так как, хотя порядок элементов в массиве изменился, ключи не изменились. Следовательно, список не будет перерисован.

Так что, как правило, если у вас есть массив, который можно изменить, используйте уникальный идентификатор. Если он недоступен, то создайте его для каждого элемента, прежде чем отобразить список. В противном случае можно использовать индекс для атрибута key.

До сих пор мы использовали функцию map непосредственно в выражении return. Однако, если хотите, вы можете сначала использовать переменную для хранения результатов map, а затем отобразить содержимое переменной.

Листинг кода 22.13

```
const renderAnimals = animals.map(item => (
    <li key={item.id}>{item.animal}</li>
  ));

  return <ul>{renderAnimals}</ul>;
```

Если хотите, вы можете даже использовать функцию.

Листинг кода 22.14

```
const getAnimalsContent = animals => animals.map(item => (
    <li key={item.id}>{item.animal}</li>
  ));

  return <ul>{getAnimalsContent(animals)}</ul>;
```

Использование других циклов в React.

- For-of

Листинг кода 22.15

```
const getAnimalsContent = animals => {
    let content = [];
    for (let item of animals) {
      content.push(<li key={item.id}>{item.animal}</li>);
    }
    return content;
  };

  return <ul>{getAnimalsContent(animals)}</ul>;
```

- For-in

Листинг кода 22.16

```
const getAnimalsContent = animals => {
    let content = [];
    for (let idx in animals) {
      const item = animals[idx];
      content.push(<li key={item.id}>{item.animal}</li>);
    }
    return content;
  };

  return <ul>{getAnimalsContent(animals)}</ul>;
```

- For-standard

Листинг кода 22.17

```
const getAnimalsContent = animals => {
    let content = [];
    for (let i = 0; i < animals.length; i++) {
      const item = animals[i];
      content.push(<li key={item.id}>{item.animal}</li>);
    }
    return content;
  };

  return <ul>{getAnimalsContent(animals)}</ul>;
```

- Filter

Функция Filter может использоваться вместе с фильтрацией элементов map перед их отображением. Например, в приведенном ниже примере будут отображаться только элементы «Mouse» и «Horse».

Листинг кода 22.18

```
const getAnimalsContent = animals =>
    animals
      .filter(item => item.animal.includes("e"))
      .map(item => <li key={item.id}>{item.animal}</li>);

  return <ul>{getAnimalsContent(animals)}</ul>;
```

- Reduce

Приведенный выше пример filter можно улучшить с помощью метода reduce, и вместо двух циклов — одного для фильтрации и одного для создания контента JSX — у нас будет только один.

Листинг кода 22.19

```
const getAnimalsContent = animals =>
    animals.reduce((acc, item) => {
      if (item.animal.includes("e")) {
        acc.push(<li key={item.id}>{item.animal}</li>);
      }
      return acc;
    }, []);

  return <ul>{getAnimalsContent(animals)}</ul>;
```

Если по какой-либо причине вы хотите поэкспериментировать, вы можете даже использовать рекурсию для отображения списка. Однако я бы не советовал делать это, и в большинстве случаев вам действительно следует придерживаться функции map.

Листинг кода 22.20

```
const getAnimalsContent = (animals, content = []) => {
    if (!animals.length) return content;
    const [item, ...restAnimals] = animals;
    content.push(<li key={item.id}>{item.animal}</li>);
    return getAnimalsContent(restAnimals, content);
  };

  return <ul>{getAnimalsContent(animals)}</ul>;
```

Передача данных в дочерне компоненты

Передача данных происходит с помощью props. Props это сокращение от properties. Когда React видит элемент, представляющий определяемый пользователем компонент, он передает атрибуты JSX этому компоненту как единый объект. Мы называем этот объект props.

Создаем файл Pros.js. Вставьте код в файл.

Листинг кода 22.21

```
import React from 'react'

const Square = (props) => {
    return (
        <button>
            {props.value}
        </button>
    );
}
const Square1 = () => {
    return <Square value={ "Click me!"} />;
}
export default Square1
```

Подключите его в файл App.js также как компонент Props.js. Запустите приложение. После запуска в браузере вы увидите кнопку (Рисунок 22.8).

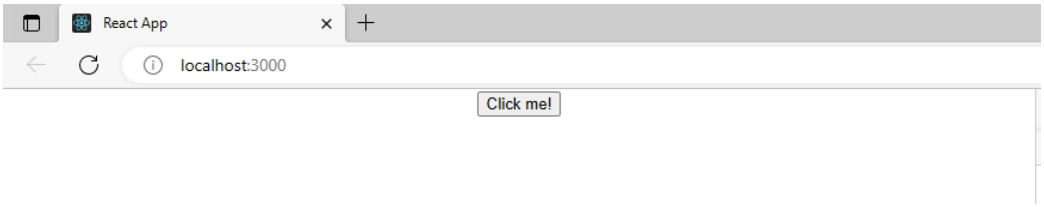

Рисунок 22.8

ВНИМАНИЕ! Темы ваших работ совпадают с темой к курсовой работы.

Задание для выполнения:

1. Изучить материал в методичке;

2. Создать React приложение и написать 2 компонента.

3. Создать 2 компонента и, используя условные операторы, вывести их.

4. Создать массив и вывести его на экран.

5. Создать 2 компонента и передать из родительского в дочерний несколько параметров.

6. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое React?

2. Что такое компоненты в React?

3. Какие условными методами можно отобразить компоненты?

4. Какие методы можно использовать для отображения компонентов в цикле?

5. Как можно передавать данные в дочерний компонент?

23. ЛАБОРАТОРНАЯ РАБОТА №23. Хуки в ReactJS.

Цель работы: Решение задач с использованием хуков useState и useEffect.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Хуки в React 16.8 – это инновация, позволяющая использовать функциональные возможности React без необходимости создания классов.

На данной странице представлено описание API, связанного с встроенными хуками React.

Для тех, кто только начинает изучать хуки, рекомендуется ознакомиться с общим обзором. Также полезную информацию можно найти в разделе "Хуки: ответы на вопросы".

Основные хуки:

1. useState
2. useEffect
3. useContext

Дополнительные хуки:

1. useReducer
2. useCallback
3. useMemo
4. useRef
5. useImperativeHandle
6. useLayoutEffect
7. useDebugValue

Использование хука useState

Литиснг кода 23.1

```
const [state, setState] = useState(initialState)
```

Возвращает значение вместе с состоянием и функцией для его обновления.

При первоначальном рендеринге возвращаемое состояние (state) совпадает с переданным значением в качестве первого аргумента (initialState).

Функция setState используется для изменения состояния. Она принимает новое значение состояния и планирует повторный рендер компонента.

При последующих повторных рендерах первое значение, возвращаемое useState, всегда будет являться последним состоянием после применения обновлений.

Функциональные обновления

Если новое состояние вычисляется на основе предыдущего состояния, можно передать функцию в setState. Функция получит предыдущее значение и вернет обновленное значение. Вот пример компонента счетчик, который использует обе формы setState.

Листинг кода 23.2

```
import React, { useState } from 'react'
const Counter = ({ initialCount }) => {
    const [count, setCount] = useState(initialCount)
    return (
        <>
            Счёт: {count}
            <button onClick={() => setCount(initialCount)}>
                Сбросить
            </button>
            <button
                onClick={() =>
                    setCount((prevCount) => prevCount + 1)
                }
            >
                +
            </button>
            <button onClick={() => setCount((prevCount) =>
prevCount - 1)}>-</button>
        </>
    )
}
export default Counter
```

Кнопки «+» и «-» используют функциональную форму, потому что обновлённое значение основано на предыдущем значении. Но кнопка «Сбросить» использует обычную форму, потому что она всегда устанавливает счётчик обратно в 2.

На рисунке 23.1 – 23.2 представлена демонстрация работы кода с начальным значением 2.

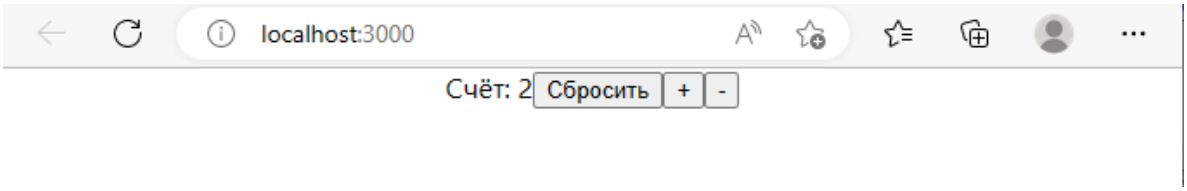

Рисунок 23.1 - до изменения состояния

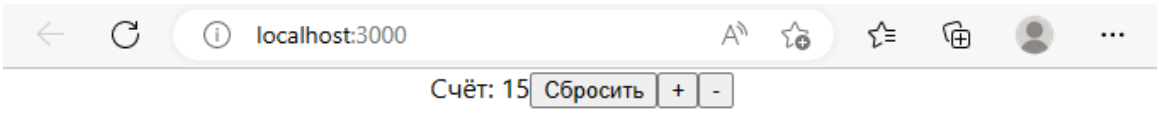

Рисунок 23.2 - после изменения состояния

Применение хука useEffect

Для избежания использования мутаций, подписок, таймеров, логирования и прочих побочных эффектов в основной части функционального компонента (этап рендеринга React) следует использовать хук useEffect. Вместо этого, все вышеперечисленные операции должны быть вынесены внутрь функции, переданной в useEffect. Таким образом, эта функция будет выполнена после того, как рендеринг компонента завершится.

Хук useEffect принимает два аргумента:

- callback, в котором содержится вся необходимая логика, например, запросы на сервер, установка обработчиков событий на документе и другие операции;

- массив аргументов, изменение которых вызовет выполнение callback. Благодаря этому массиву мы можем эмулировать методы жизненного цикла.

По умолчанию эффекты запускаются после каждого завершенного рендеринга, однако вы можете настроить их запуск только при изменении определенных значений.

Литисng кода 23.3

```
const App = ({data}) => {
  useEffect(() => {
    console.log("componentDidUpdate");
  }, [data]);

  return null;
};
```

Условное срабатывание эффекта

По умолчанию эффекты запускаются после каждого завершения рендера. Следовательно, эффект будет перезапущен только в случае изменения какого-либо из параметров, от которых он зависит.

Тем не менее, в определённых ситуациях это может быть излишне, как, например, в случае с подпиской из предыдущего примера. Нам необходимо создавать новую подписку только при изменении props.source, а не при каждом обновлении.

Для реализации этого передайте вторым аргументом в useEffect массив значений, от которых зависит данный эффект. Наш обновлённый пример теперь выглядит следующим образом.

Литисng кода 23.4

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

Теперь при изменении props.source будет создаваться новая подписка. При использовании данной оптимизации необходимо учитывать, что массив должен содержать все значения из области видимости компонента (например, пропсы и состояние), которые могут измениться со временем и будут использоваться эффектом. В противном случае, код будет ссылаться на устаревшие значения из прошлых рендеров. Документация содержит информацию о том, как обрабатывать функции и изменяемые массивы.

Для выполнения эффекта только один раз при монтировании и размонтировании компонента можно передать пустой массив ([]) в качестве второго аргумента. React определит, что эффект не зависит от значений из пропсов или состояния, поэтому не будет повторно запускать эффект. Это не является особым случаем, а следует непосредственно из логики работы входных массивов.

Если вы передасте пустой массив, пропсам и состояниям внутри эффекта будут присвоены значения, которые были изначально определены. Несмотря на то, что передача "" ближе к знакомой модели мышления, обычно существуют более эффективные способы избежать повторного рендеринга. Не забывайте о том, что react откладывается выполнение use Effect до того момента, как браузер завершит все изменения. Это не является серьезной проблемой для выполнения дополнительной работы.

В нашем пакете правил для линтеров, мы рекомендуем использование правила exhausive deps, которое является частью нашего пакета правил литера eslinet-Plugin Reacting. Оно предупреждает о том, что некоторые зависимости неправильно указаны, и предлагает исправить их.

Массив зависимости не может быть использован для аргументов функций эффекта, как аргументы. В теории, несмотря на то, что в теории происходит это, каждое значение функции эффекта должно быть представлено в массивах связей. В будущем, при достаточно продвинутой компиляции, можно будет создавать этот массив вручную.

Задание для выполнения:

1. Ознакомиться с материалом методического указания, выполните примеры из методички (с кодом, скриншотами, на которых обязательно должно будет указано ФИО студента, выполнившего задание).

2. Добавьте к себе в проект хуки useState и useEffect.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое хук?

2. Назовите известные вам хуки?

3. В каком случае применяется каждый хук?

4. В каких ситуациях могут пригодиться пользовательские хуки?

24. ЛАБОРАТОРНАЯ РАБОТА №24. Контекст в ReactJS.

Цель работы: Решение задач с использованием хука useContext.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Хуки в React 16.8 – это инновация, позволяющая использовать функциональные возможности React без необходимости создания классов.

На данной странице представлено описание API, связанного с встроенными хуками React.

Для тех, кто только начинает изучать хуки, рекомендуется ознакомиться с общим обзором. Также полезную информацию можно найти в разделе "Хуки: ответы на вопросы".

Основные хуки:

1. useState
2. useEffect
3. useContext

Дополнительные хуки:

1. useReducer
2. useCallback
3. useMemo
4. useRef
5. useImperativeHandle
6. useLayoutEffect
7. useDebugValue

Использование хука useContext

```
const value = useContext(MyContext);
```

Хук useContext принимает объект контекста (значение, возвращенное из React.createContext) и возвращает текущее значение контекста для этого контекста. Текущее

значение контекста определяется значением ближайшего провайдера MyContext.Provider, находящегося выше вызывающего компонента в дереве.

При обновлении ближайшего провайдера MyContext.Provider этот хук вызовет повторный рендер с последним значением контекста, переданным этому провайдеру MyContext. Даже если родительский компонент использует React.memo или реализует shouldComponentUpdate, повторный рендер будет выполнен, начиная с компонента, использующего useContext.

Помните, что аргументом для useContext должен быть непосредственно сам объект контекста:

- Правильно: useContext(MyContext)

- Неправильно: useContext(MyContext.Consumer)

- Неправильно: useContext(MyContext.Provider)

Компонент, использующий useContext, всегда будет перерендериваться при изменении значения контекста. Если повторный рендер компонента затратен, его можно оптимизировать с помощью мемоизации.

Совет: Если вы знакомы с API контекстов до появления хуков, вызов useContext(MyContext) аналогичен выражению static contextType = MyContext в классе или компоненте MyContext.Consumer.

Хук useContext(MyContext) позволяет только читать контекст и подписываться на его изменения. Вам все еще нужен провайдер MyContext.Provider выше в дереве, чтобы предоставить значение для этого контекста. Соедините все вместе с Context.Provider.

Листинг кода 24.1

```
import { createContext, useContext, useState } from 'react';
import "./App.css";
const ThemeContext = createContext(null);
export default function MyApp() {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      <label>
        <input
          type="checkbox"
          checked={theme === 'dark'}
          onChange={(e) => {
            setTheme(e.target.checked ? 'dark' : 'light')
          }}
        />
        Use dark mode
      </label>
```

```
    </ThemeContext.Provider>
  )
}
function Form({ children }) {
  return (
    <Panel title="Welcome">
      <Button>Sign up</Button>
      <Button>Log in</Button>
    </Panel>
  );
}
function Panel({ title, children }) {
  const theme = useContext(ThemeContext);
  const className = 'panel-' + theme;
  return (
    <section className={className}>
      <h1>{title}</h1>
      {children}
    </section>
  )
}
function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className}>
      {children}
    </button>
  );
}
```

Добавить в CSS код.

Листинг кода 24.2

```
.panel-light,
.panel-dark {
  border: 1px solid black;
  border-radius: 4px;
  padding: 32px;
  margin-bottom: 10px;
}
.panel-light {
  color: #342;
  background: #fff;
}

.panel-dark {
  color: #fff;
  background: rgb(31, 32, 42);
}
```

```
.button-light,
.button-dark {
  border: 1px solid #777;
  padding: 5px;
  margin-right: 10px;
  margin-top: 10px;
}

.button-dark {
  background: #342;
  color: #fff;
}

.button-light {
  background: #fff;
  color: #342;
}
```

На рисунке 24.1 –24.2 представлена демонстрация работы кода.

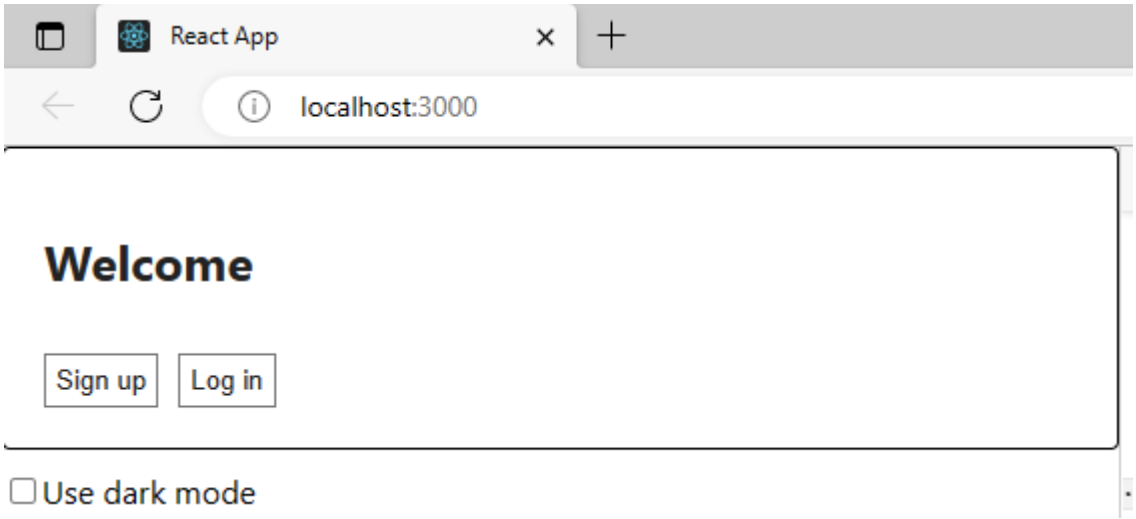

Рисунок 24.1 - до изменения состояния

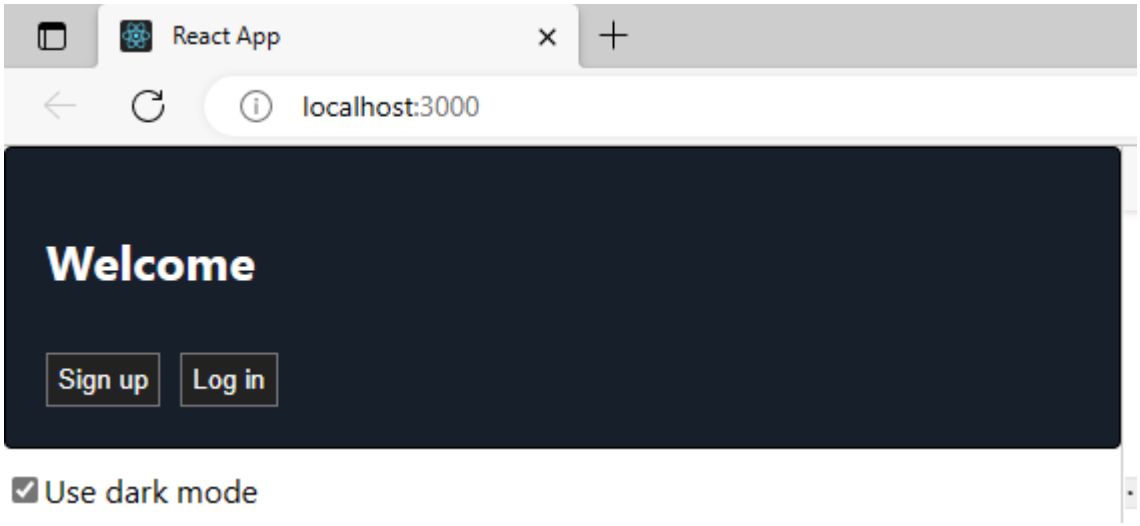

Рисунок 24.2 - после изменения состояния

Задание для выполнения:

1. Ознакомиться с материалом методического указания, выполните примеры из методички (с кодом, скриншотами, на которых обязательно должно будет указано ФИО студента, выполнившего задание).

2. Добавьте к себе в проект хук useContext.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое хук?

2. Назовите известные вам хуки?

3. В каком случае применяется каждый хук?

4. В каких ситуациях могут пригодиться пользовательские хуки?

25. ЛАБОРАТОРНАЯ РАБОТА №25. Списки в ReactJS.

Цель работы: Создание React приложения, который отобразит список (с пагинацией) и предоставит возможность добавить элемент в список. Сами эмементы должны храниться на сервере. При реализации необходимо воспользоваться двумя разными библиотеками для создания форм.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Существует специальный синтаксис для работы с промисами, который называется «async/await». Он удивительно прост для понимания и использования.

Async – функция всегда возвращает промис.

Листинг кода 25.1

```
async function имя(параметры) {
  //тело функции
}
```

Await – заставляет ждать интерпретаторов JS, пока промис справа от ключевого слова не выполниться.

Листинг кода 25.2

```
async function имя(параметры) {
await промис;
}
```

ВНИМАНИЕ! Использование await внутри функции, объявленной без async, получим синтаксическую ошибку. Ошибки не будет, если мы укажем ключевое слово async перед объявлением функции.

Если несколько способов получить данные с сервера.

Первый способ – воспользоваться методом fetch (Fetch API).

Начнем с метода fetch(), который является современным и очень мощным. Хотя он не поддерживается старыми браузерами (можно использовать полифил), все современные браузеры его поддерживают. Fetch API основан на промисах и возвращает объект ответа.

Листинг кода 25.3

```
let promise = fetch(url, [options])
```

где,

url – URL для отправки запроса.

options – дополнительные параметры: метод, заголовки и так далее.

Получение ответа происходит в 2 этапа:

1. promise выполняется с объектом встроенного класса Response в качестве результата, как только сервер пришлёт заголовки ответа.

-.status – HTTP-код ответа,

-.ok – true, если статус ответа в диапазоне 320-339.

-.headers – похожий на Map объект с HTTP-заголовками.

2. для получения тела ответа нам нужно использовать дополнительный вызов метода:

- .text() – читает ответ и возвращает как обычный текст,

- .json() – декодирует ответ в формате JSON,

- .formData() – возвращает ответ как объект FormData (разберём его в следующей главе),

- .blob() – возвращает объект как Blob (бинарные данные с типом),

- .arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),

- .body – это объект ReadableStream, с помощью которого можно считывать тело запроса по частям.

Для отправки POST-запроса или запроса с другим методом, нам необходимо использовать fetch параметры:

1.method – HTTP метод, например POST,

2.body – тело запроса, одно из списка:

- строка (например, в формате JSON),

- объект FormData для отправки данных как form/multipart,

- Blob/BufferSource для отправки бинарных данных,

- URLSearchParams для отправки данных в кодировке x-www-form-urlencoded, используется редко.

Чаще всего используется JSON.

Второй способ – воспользоваться библиотекой axios.js

Axios - это простой HTTP-клиент на основе promise для браузера и. node.js. Axios предоставляет простую в использовании библиотеку в небольшом пакете с очень расширяемым интерфейсом.

Листинг кода 25.4

```
axios.ТИП_ЗАПРОСВ(url, {тело(если необходимо})
 .then(function (ответ) {
//обработка результата если нет ошибки
})
 .catch(function (ошибка) {
//обработка результата если есть ошибки
 });
```

Пагинацией называется разделение большого массива данных, имеющихся на сайте, на отдельные страницы для удобства использования.

В лабораторной работе вам необходимо:

1. доделать сервер по заданию;

2. написать клинскую часть:

- создать 2 компонента 1 компонент для отображения списка, 2 для отображения кнопок пагинации;

- создать 2 запроса к серверу для получения данных: количества элементов в списке и списка элементов по условию.

Реализация

Создали папку с проектом в ней создали 2 папки: server и front.

Сервер:

1. Инициализируем его.

Листинг кода 25.5

```
npm init
```

2. Добавляем библиотеки:

- cors – получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент.

- express

- knex

- nodemon – при каждом изменение на сервере, сервер перезапускается

- pg

Листинг кода 25.6

```
npm I cors, express, knex, nodemon, pg
```

2. Теперь необходимо подключиться к БД.

Для этого создадим файл knexfile.js в папке config проекта и добавь следующий код.

Листинг кода 25.7

```
module.exports = {
      client: 'pg',
      connection: {
        host: '131.0.0.1',
        port: '5432',
        user: 'postgres', //Логин для подключения к БД
        password: 'password', //Пароль
        database: 'database_name' //Название БД
      },
      migrations: {
        directory: '../migrations'
      },
      seeds: {
        directory: '../seeds'

    }
  }
```

4. Добавляем миграцию (создали 1 таблицу с 2 колонками: id и название) и сидирование (заполнили некоторыми данными таблицу):

Листинг кода 25.8 (миграция)

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

    })
};
```

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

};
```

Листинг кода 25.9 (сидирование)

```
/**
 * @param { import("knex").Knex } knex
 * @returns { Promise<void> }
 */
exports.seed = async function (knex) {
  // Deletes ALL existing entries

  await knex('list').insert([
    { name: 'Аардоникс' },
    { name: 'Абботины' },
    { name: 'кот' },
    { name: 'собака' },
    { name: 'осел' },
    { name: 'кот' },
    { name: 'козел' },
    { name: 'лошадь' },
    { name: 'свинья' },
    { name: 'кролик' },
    { name: 'трубкозуб' },
    { name: 'муравей' },
    { name: 'муравьед' },
    { name: 'бандикут' },
    { name: 'клоп' },
    { name: 'пчела' },
    { name: 'жук' },
    { name: 'бонобо' }
  ]);
};
```

5. Запустим сидирование и миграцию.

6. Далее необходимо создать контроллеры. Для этого создадим папку controller и там создадим 3 файла: add.js (добавить новый элемент списка), all.js (вывод количество элементов в списке) и lim.js (вывод список элементов по условию) .

Листинг кода 25.10 (add.js)

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

```
    exports.add = (req, res) => {
        const{name } =req.body;

        console.log(name)
        knex('list')
            .insert({name })
            .then(list => {
                if (list.length == 0) {
                    res.status(401).json('Нет списка');
                } else {
                    res.status(320).send({
                        list
                    });
                }
            })
            .catch(error => {
                console.error(error);
                res.status(500).json({  error:  'Внутренняя  ошибка
сервера' });
            });



    };
```

Листинг кода 25.11 (all.js)

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

    exports.all = (req, res) => {
        console.log(req.body)
        knex('list')
            .select()
            .then(list => {
                if (list.length == 0) {

                    res.status(401).json('Нет списка');
                } else {
                    res.send({
                        count: list.length
                    });
                }
            })
            .catch(error => {
                console.error(error);
                res.status(500).json({  error:  'Внутренняя  ошибка
сервера' });
            });
```

```
};
```

Листинг кода 25.12 (lim.js)

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

exports.lim = (req, res) => {
    const offset = parseInt(req.query.offset); // номер
запрашиваемой страницы
    const limit = parseInt(req.query.limit);
    console.log(req.query)
    knex('list')
        .select()
        .limit(limit)
        .offset(offset)
        .then(list => {
            if (list.length == 0) {
                res.status(401).json('Нет списка');
            } else {
                res.status(320).send({
                    list
                });
            }
        })
        .catch(error => {
            console.error(error);
            res.status(500).json({ error: 'Внутренняя ошибка
сервера' });
        });

};
```

7. Теперь создадим роутеры. Для этого создадим папку router и там создадим 3 файла: list.router.js.

Листинг кода 25.13 (list.router.js)

```
const express = require("express");
const router = express.Router();

const { all } = require("../controller/all");
const { lim } = require("../controller/lim");
const { add } = require("../controller/add");

router.use((req, res, next) => {
    res.header(
        "Access-Control-Allow-Headers",
```

```
            "x-access-token,    Origin,    Content-Type,    Accept,
Authorization"
        );
        next();
    });

    router.get("/all", all);
    router.post("/add", add);
    router.get("/lim", lim);

    module.exports = router;
```

8. Теперь в корне сервера создадим файл server.js и добавим код.

Листинг кода 25.14

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

const list = require('./router/list.router');
app.use(express.json());
app.use(express.urlencoded({ extended: false }))

app.use(cors({
origin: '*'
}));

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

app.use("/list", list)


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

8. Далее добавим код в файл package.json.

Листинг кода 25.15

```
…
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon server.js",
    "migrate":      "knex      migrate:latest      --knexfile
./config/knexfile.js",
```

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

Фронт:

1. Создаем React-приложение.

Листинг кода 25.16

```
npx create-react-app .
```

2. Добавляем библиотеке:

- axios – построение запросов к серверу

Листинг кода 25.17

```
npm I axios
```

2. Теперь необходимо создать компоненты. Для этого создаём папку components и создаем 2 файла: list.js (отображение списка) и pagination.js (отображение кнопок пагинации).

Параметры для компонента list.js: список элементов и значение загрузки.

Необходимо прописать условие, что если значение загрузки равно True, то не отображать список и отобразить <h2>Loading....</h2>.

Для отображения списка используйте метод для работы с массивом .map.

Листинг кода 25.18

```
return (
        <div>
            {
                list.map((list) => (
                    <li>
                       { list.name }
                    </li>

                ))
            }
        </div>
    );
```

Параметры для компонента pagination.js: длину списка, количество элементов представленных на 1 странице и функция, которая будет возвращать номер страницы, на которую нажали.

Необходимо создать цикл, который будет в список добавлять элементы, условия выполнения циклам – общие количество элементов делить на количество элементов на 1 странице и затем округлить.

Листинг кода 25.19

```
const pageNumbers = []

     for (let i = 1; i <= Math.ceil(totalList / listPerpage); i++) {
         pageNumbers.push(i)
        }
```

Теперь необходимо пройтись по этому списку и отобразить элементы на экране. Необходимо добавить обработку для нажатия на элемент.

Листинг кода 25.20

```
<ul className='pagination' >
                  {
                      pageNumbers.map((number) => (
                          <li className='page-item' key={number}>
                                <a href='!#' className='page-link' onClick={()=> paginate(number)}>
                                    {number}
                                </a>
                          </li>
                      ))
                  }

              </ul>
```

3. Далее необходимо изменить файл App.js.

Необходимо создать несколько переменных:

- const [list, setList] = useState([]); //принимает список
- const [loading, setLoading] = useState(false); //принимает значение загрузки
- const [currentPage, setCurrentPage] = useState(1); //принимает номер текущей странице
- const [listPerpage] = useState(5) //количество элементов 1 странице
- const [totalList, setTotalList] = useState([]) //принимает количество элементов в списке
- const [name, setName] = useState(""); //считывает значение с поле формы
- const lastListIndex = currentPage * listPerpage //считает номер последнего элемента на странице

- const firstListIndex = lastListIndex – listPerpage //считает намер первого элемента на следующей странице

И функцию, которая будет присваивать значение currentPage при каждом ее вызове:

- const paginate = pageNamber => {setCurrentPage(pageNamber)

Добавим несколько добавить запросы на сервер:

- Запрос на количество элементов в списке. Запрос будет вызываться с помощью хука useEffect после завершения рендоринга. Необходимо добавить обработку ответов. При положительно ответе присвоить значение переменной totalList (количество элементов) и loading (false)

Листинг кода 25.21

```
useEffect(() => {
  const getList = async () => {
    setLoading(true)
await axios.get('http://localhost:8080/list/all')
      .then((data) => {
        console.log(data)
        setTotalList(data.data.count)
        setLoading(false)
      })
      .catch(function (error) {
        if (error.response) {
          console.log(error.response.status);
          console.log(error.response.headers);
        } else if (error.request) {
          console.log(error.request)
        } else {
          console.log('Error', error.message);
        }
        console.log(error.config);
      });
  }
  getList()
}, [])
```

- Запрос на вывод элементов с условием: количество и сколько элементов пропускать. Запрос будет вызываться с помощью хука useEffect после изменения переменной currentPage. Необходимо добавить обработку ответов. При положительно ответе присвоить значение переменной list (список элементов) и loading (false)

Листинг кода 25.22

```
        useEffect(() => {
        const getList = async () => {
          setLoading(true)

        await
axios.get(`http://localhost:8080/list/lim?offset=${firstListIndex
}&limit=${listPerpage}`)
            .then((data) => {
              console.log(data)
              setList(data.data.list)
              setLoading(false)
            })
            .catch(function (error) {
              if (error.response) {
                console.log(error.response.status);
                console.log(error.response.headers);
              } else if (error.request) {
                console.log(error.request);

              } else {
                console.log('Error', error.message);
              }
              console.log(error.config);
            });
        }
        getList()

      }, [currentPage])
```

Теперь создаем форму, где при каждом изменение значения input будет вызываться handleOnChange (присваивание значение переменной name).

Листинг кода 25.23

```
…
  const handleOnChange = (event) => {
    setName(event.target.value)
  }
…
```

Листинг кода 25.24

```
…
          <input class="form-control"
            id="name" name="name"
            value={name}
            onChange={(event) => handleOnChange(event)} />
…
```

Теперь необходимо добавить обработку при нажатие на кнопке Добавить. При нажатии будет вызываться функция, которая будет отправлять запрос на сервер с параметрами.

Листинг кода 25.25

```
  const onAdd = async () => {
    await axios.post(`http://localhost:8080/list/add`, { name},
{
    })
      .then((data) => {
        alert("Вы добавили в список элемент!")

     })
  }
```

ВНИМАНИЕ! Не забудьте подключить bootstrap в файле index.html.

Листинг кода 25.26

```
…
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstra
p.min.css"            rel="stylesheet"            integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC"
crossorigin="anonymous">
…
```

Теперь можем запустить сервер и фронт командой.

Листинг кода 25.27

```
npm start
```

Результат работы представлен на рисунке 25.1.

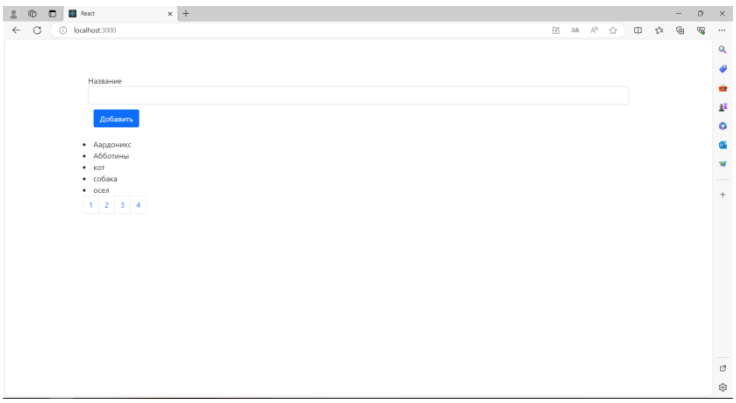

Рисунок 25.1

Теперь можем добавить еще животное. Вводим в форму данные (рисунок 25.2). Результат работы представлен на рисунке 25.3.

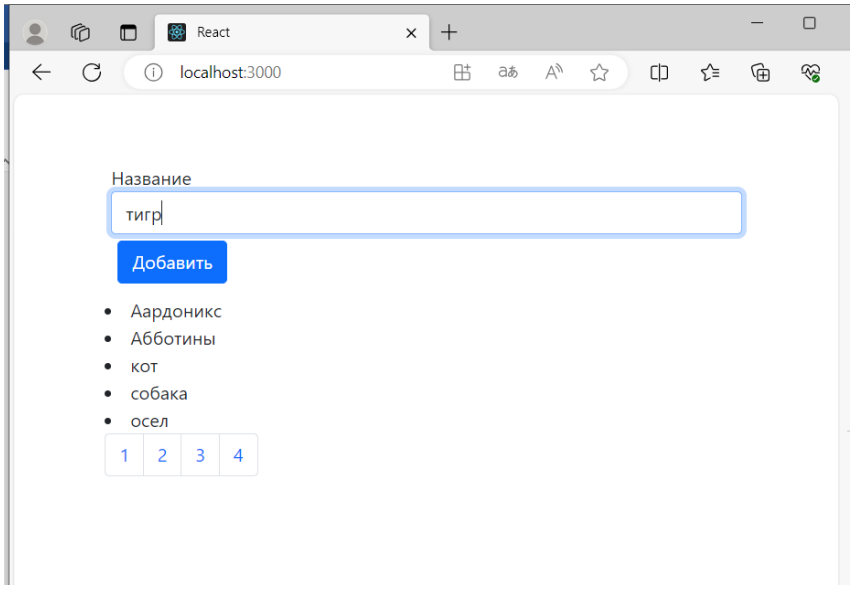

Рисунок 25.2

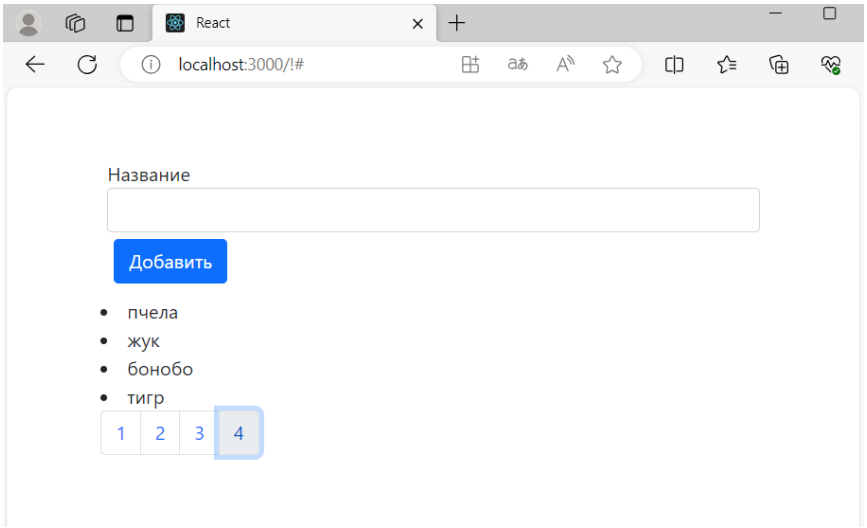

Рисунок 25.3

Задание для выполнения:

1. Ознакомиться с материалом методического указания, выполните примеры из методички (с кодом, скриншотами).

2. Создать список товаров (элементов). Товар будет состоять из нескольких полей: Название, описание, фото и т.д.

3. Добавить пагинацию на ваш сайт (лб 9) и отражение товаров (элементов).

4. Добавить форму для добавления товаров (элементов). Форму сделать 2 библиотеками: bootstrap и любая другая.

5. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое список?

2. Что такое пагинация?

3. Какие методы вам известны для работы со списками?

4. Какие библиотеки были использованы в лабе и почему?

26. ЛАБОРАТОРНАЯ РАБОТА №26. Маршрутизация в ReactJS

Цель работы: Создание веб-приложения, состоящего из нескольких страниц.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Все приложения на React по своей сути Single page application (одностраничное приложение). При переходе по ссылкам в браузере только обновляются компоненты React.

В React имеется своя система маршрутизация, которая позволяет сопоставлять запросы к приложению с определенными компонентами. При старте проекта нужно определить какой тип роутера использовать. Для браузерных проектов есть BrowserRouter и HashRouter компоненты.

Обычно предпочтительнее использовать BrowserRouter.

Для создания маршрутизации вам нужно совершить следующие действия:

1. Установить необходимую библиотеку
2. Создать компоненты страниц
3. Создать логики перехода по страницам

React Router – это библиотека для навигации между разными частями веб-приложения, созданными на React. Она позволяет менять содержимое страницы без перезагрузки браузера, что делает приложение более интерактивным и удобным для пользователей.

Установка библиотеки

После создание пустого веб-приложения вам необходимо установить библиотеку react-router-dom.

С помощью команды.

Листинг кода 26.1

```
npm install react-router-dom –save
```

Создание страниц

Создадим 2 страницы (главную и страницу о нас). Для начала создадим файл Home.js и добавим туда код главной странице.

Листинг кода 26.2

```
const Home = () => {
    return (
        <div>
            <h1>Главная страница</h1>
        </div>
    )
}
export default Home
```

Далее создадим файл About.js и добавим туда код.

Листинг кода 26.3

```
const About = () => {
    return (
        <div>
            <h1>О нас</h1>
        </div>
    )
}
export default About
```

Логика перехода по страницам

Для этого добавляем в файл App.js.

Листинг кода 26.4

```
import './App.css';
import {BrowserRouter as Router, Link, Route, Routes } from
'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
 return (
     <Router>
       <div>
         <ul>
           <li><Link to={'/'}>Главная страница</Link></li>
           <li><Link to={'/about'}>О нас</Link></li>
         </ul>
         <hr/>

         <Routes>
           <Route path='/' element={<Home/>}/>
           <Route path='/about' element={<About/>}/>
         </Routes>
       </div>
     </Router>
 );
```

```
}

export default App;
```

В коде мы импортируем 2 других компонента и создаём ссылку на них с помощью компонента <Link>

Далее в блоке <Routes мы описываем какой элемент будет отрисовываться при переходе по определённой ссылке. Компонент указывается в параметре element.

Таким образом в одном месте мы можем описать переходы на разные страницы.

На рисунках 26.1 – 26.2 представлена работа перехода между страницами.

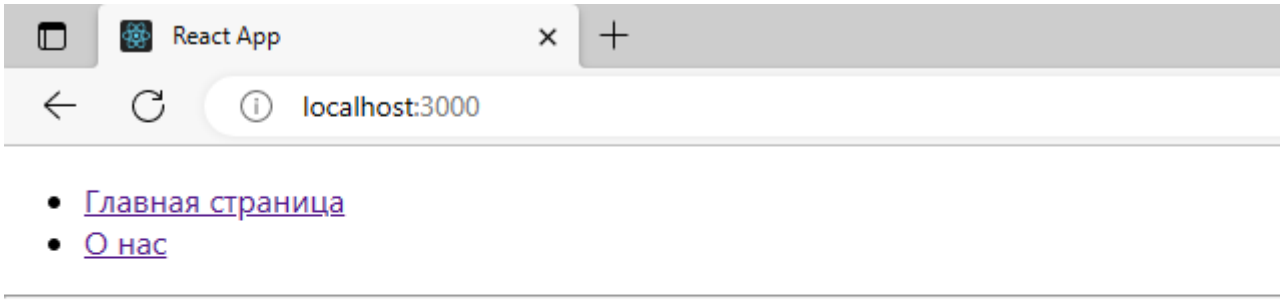

Рисунок 26.1 – Страница главная

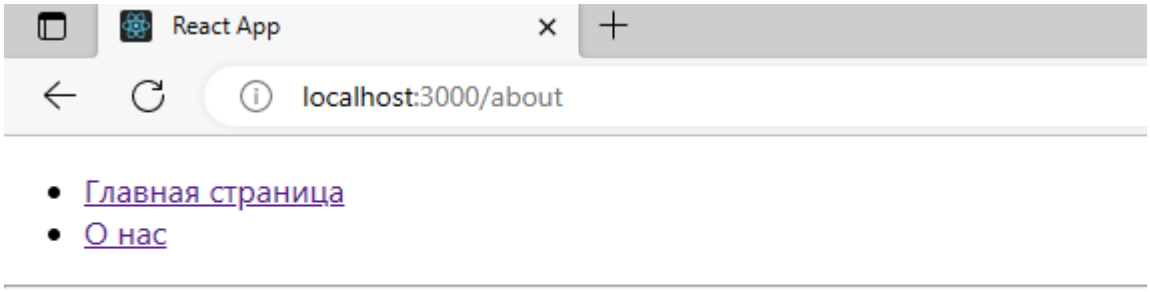

Рисунок 26.2 – Страница «О нас»

Задание для выполнения:

1. Выполнить пример из методички, приложить скриншоты работы;

2. Написать приложение по теме курсового проекта с маршрутизацией глубиной минимум в четыре страницы:

- Создать Header (Navbar) с использованием сторонних библиотек.
- Добавить на страницы индивидуальный стиль.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Как происходит работа React-приложения?
2. Что такое маршрутизация в React.js?
3. Какую библиотеку используют для маршрутизации в React.js?
4. Какие компоненты используют для маршрутизации в React.js?

27 ЛАБОРАТОРНАЯ РАБОТА №27. Redux

Цель работы: Настройка redux двумя способами: legacy redux и redux-toolkit.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Redux — библиотека для JavaScript с открытым исходным кодом, предназначенная для управления состоянием приложения. Чаще всего используется в связке с React или Angular для разработки клиентской части. Содержит ряд инструментов, позволяющих значительно упростить передачу данных хранилища через контекст.

В лабораторной работе мы рассмотри 2 способа настройки redux:

1. legacy redux

2.redux-toolkit.

Оба способа позволяют эффективно управлять состоянием приложения, но redux-toolkit предлагает удобные утилиты для более простой конфигурации и использования Redux.

Наиболее значимыми функциями, предоставляемыми библиотекой Redux Toolkit являются:

- configureStore — функция, предназначенная упростить процесс создания и настройки хранилища. Функция позволяет комбинировать редюсеры, добавлять middleware, а также использовать расширение Redux DevTools. Входные параметры – объект со свойствами:

1. reducer — набор пользовательских редюсеров
2. middleware — массив middleware (опционально)
3. devTools — включить расширение Redux DevTools (по умолчанию — true)
4. preloadedState — начальное состояние хранилища (опционально)
6. enhancers — набор усилителей (опционально)

- createReducer. Функция, помогающая описать и создать редюсер. Функция позволяет существенно упростить логику иммутабельного обновления, написав код в «мутабельном» стиле внутри мини-редюсеров.

- createAction. Функция позволяет 2 действия - объявить константу и создать генератор действия – объединить в 1. Входной параметр – тип действия(type). Возвращает генератор этого действия. Если эта функция может принимать и 2 параметр prepareAction (позволяет подготовить payload, прежде чем он будет использован в редюсере

- createSlice — объединяет в себе функционал createAction и createReducer. В качестве входных параметров принимает объект со следующими полями:

1. name — имя среза (служит префиксом для экшенов, например todo/toggle)

2. initialState — начальное состояние среза состояния

3. reducers — объект с обработчиками экшенов. Каждый обработчик принимает state и action = {type, payload}

4. extraReducers — объект, содержащий редюсеры другого среза, если нужно обновить другой срез

Результатом работы функции является объект, называемый «срез», со следующими полями:

- name — имя среза состояния.

- reducer — атоматически созданная функция-редюсер, которую можно передать в combineReducers.

- actions — автоматически созданные с помощью createAction генераторы действий.

- caseReducers — те функции, которые мы передали в createSlice через поле reducers.

- createSelector — переэкспортированная функция для кэширования из пакета reselect.

Библиотека react-redux – официальная библиотека, которая предоставляет привязку React для Redux.

Компонент <Provider> необходим, чтобы сделать хранилище доступным во всём дереве компонентов.

Хуки:

- useSelector() - Позволяет извлекать данные из состояния(state) хранилища(store).

- useDispatch(). Возвращает ссылку на функцию dispatch из Redux хранилища(store).

- useStore(). Возвращает ссылку на то же Redux хранилище(store), которое было передано компоненту <Provider>.

Настройка redux. Способ legacy redux

Устанавливаем библиотеку.

Листинг кода 27.1

```
npm install redux react-redux
```

Далее необходимо создать папку actions с файлами для действий. actionTypes.js – определили типы действий. counterActions.js – определения действий счетчика.

Листинг кода 27.2 (actionTypes.js)

```
export const INCREMENT = 'INCREMENT';   //увеличение
export const DECREMENT = 'DECREMENT'; //уменьшение
//необходимо перечислить все действия, которые в дальнейшем будут происходить в системе
```

Листинг кода 27.3 (counterActions.js)

```
//определили какие именно действия будут происходить в счетчике
import { INCREMENT, DECREMENT } from './actionTypes';
export const increment = () => ({
  type: INCREMENT,
});

export const decrement = () => ({
  type: DECREMENT,
});
```

Затем необходимо создать папку reducers с файлами для редукторов. counterReducer.js – описывает функции для действия счетчика (прибавлять или вычитать 1)

Листинг кода 27.4 (counterReducer.js)

```
import { INCREMENT, DECREMENT } from '../actions/actionTypes';
// Начальное состояние приложения
const initialState = {
  count: 0,
};
// Обработка действий и возврат нового состояния
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT:
      return { count: state.count + 1 };
    case DECREMENT:
      return { count: state.count - 1 };
    default:
      return state;
  }
};

export default counterReducer;
```

Теперь создадим файл store.js. В файле пропишем настройку Redux Store.

Листинг кода 27.5 (store.js)

```
import { createStore } from 'redux';
import counterReducer from './reducers/counterReducer';
const store = createStore(counterReducer);
export default store;
```

Затем счетчик с обработчиками действий (App.js).

Листинг кода 27.6

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from
'./actions/counterActions';

const App = () => {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>Счетчик: {count}</h1>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
    </div>
  );
};

export default App;
```

Теперь в index.js пропишем доступ к store. Это предоставит возможность всем компонентов вашего приложения обращаться к store. Это происходит с помощью компонента Provider из библиотеки react-redux.

Листинг кода 27.7 (store.js)

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Настройка redux. Способ redux-toolkit

Устанавливаем библиотеку.

Листинг кода 27.8

```
npm install @reduxjs/toolkit react-redux
```

Далее необходимо создать папку features с файлом counterSlice.js для определения слайса (определили действие и функционал).

Листинг кода 27.9

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
  },
  reducers: {
    increment: state => {
      state.value += 1;
    },
    decrement: state => {
      state.value -= 1;
    },
  },
});

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

Теперь создадим файл store.js. В файле пропишем настройку Redux Store.

Листинг кода 27.10 (store.js)

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './features/counterSlice';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});

export default store;
```

Затем счетчик с обработчиками действий (App.js).

Листинг кода 27.11

```
import React from 'react';
```

```
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from
'./features/counter/counterSlice';

const App = () => {
  const count = useSelector(state => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>Счетчик: {count}</h1>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
    </div>
  );
};

export default App;
```

Теперь в index.js пропишем доступ к store. Это предоставит возможность всем компонентов вашего приложения обращаться к store. Это происходит с помощью компонента Provider из библиотеки react-redux.

Листинг кода 27.12

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Задание для выполнения:

1. Ознакомитесь с методическими указаниями;

2. Настройте redux в своем проекте двумя способами: legacy redux и redux-toolkit. ВНИМАНИЕ! ЗАПРЕЩЕНО ИСПОЛЬЗОВАТЬ ДЕЙСТВИЯ, КОТОРЫЕ ОПИСАНЫ В МЕТОДИЧЕСКИХ УККЗАНИЯХ!!!!

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое Redux?
2. Зачем нужен Redux?
3. Как настроить Redux? Способ legacy redux.
4. Как настроить Redux? Способ redux-toolkit.
5. Чем отличается способ настройки legacy redux от redux-toolkit?

28. ЛАБОРАТОРНАЯ РАБОТА №28. Redux-thunk

Цель работы: Подключение к серверу через библиотеку redux и redux-thunk

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Redux Thunk — это промежуточное ПО, позволяющее вызывать создателей действий, которые возвращают функцию вместо объекта действия.

В лабораторной работе возьмем за основу сервер из 34 лб.

Изменим только одну делать. В контролере, где мы получаем количество элементов в списке (all.js), вместо получения количества получите просто список элементов.

Листинг кода 28.1

```
//было
…
res.send({
count: list.length
});
…
//стало
…
res.send({
list
});
…
```

Реализация

Устанавливаем библиотеку.

Листинг кода 28.2

```
npm install redux react-redux redux-thunk axios
```

Далее необходимо создать папку actions с файлом listActions.js с thunk-функциями, в котором прописали действий и запрос на сервер.

Листинг кода 28.3

```
import axios from 'axios';
```

```
export const fetchListSuccess = (list) => {
  return {
    type: 'FETCH_LIST_SUCCESS',
    payload: list
  };
}

export const fetchListError = (error) => {
  return {
    type: 'FETCH_LIST_ERROR',
    payload: error
  };
}

export const fetchUsers = () => {
  return (dispatch) => {
    axios.get('http://localhost:8080/list/all')
      .then(response => {
        dispatch(fetchListSuccess(response.data.list));
      })
      .catch(error => {
        dispatch(fetchListError(error.message));
      });
  };
}
```

Затем необходимо создать папку reducers с файлами для редукторов. listReducers.js – описывает функции для действия списка.

Листинг кода 28.4 (listReducers.js)

```
const initialState = {
    list: [],
    error: null
  };

  const listReducers = (state = initialState, action) => {
    switch (action.type) {
      case 'FETCH_LIST_SUCCESS':
        return {
          ...state,
          list: action.payload,
          error: null
        };
      case 'FETCH_LIST_ERROR':
        return {
          ...state,
          list: [],
          error: action.payload
        };
      default:
```

```
        return state;
      }
    };

    export default listReducers;
```

Теперь создадим файл store.js. В файле пропишем настройку Redux Store.

Листинг кода 28.5 (store.js)

```
import { createStore, applyMiddleware } from 'redux';
import {thunk} from 'redux-thunk';
import listReducers from './reducers/listReducers';

const store = createStore(listReducers,
applyMiddleware(thunk));

export default store;
```

Затем счетчик с обработчиками действий (App.js).

Листинг кода 28.6

```
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchUsers } from './actions/listActions';

const App = () => {
  const list = useSelector(state => state.list);
  const error = useSelector(state => state.error);
  const dispatch = useDispatch();

  useEffect(() => {
    dispatch(fetchUsers());
  }, [dispatch]);

  return (
    <div>
      {error && <p>{error}</p>}
      {list.map(list => (
        <p key={list.id}>{list.name}</p>
      ))}
    </div>
  );
}

export default App;
```

Теперь в index.js пропишем доступ к store. Это предоставит возможность всем компонентов вашего приложения обращаться к store. Это происходит с помощью компонента Provider из библиотеки react-redux.

Листинг кода 28.7

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Задание для выполнения:

1. Ознакомитесь с методическими указаниями;

2. Добавьте к вашему проекту подключение к серверу через библиотеку redux и redux-thunk.

3. Составить отчет по результатам работы (скриншоты и листинг с описанием)

Вопросы по лабораторной работе:

1. Что такое Redux?

2. Что такое redux-thunk?

3. Как подключится к серверу с помощью библиотеки redux-thunk?

29. ЛАБОРАТОРНАЯ РАБОТА №29. Redux-saga

Цель работы: Реализация лабораторной со списками в ReactJS с помощью redux-saga.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Redux saga - это библиотека, которая призвана упростить управление побочными эффектами приложения (то есть асинхронными вещами, такими как выборка данных и нечистыми вещами, такими как доступ к кешу браузера), более эффективным в исполнении, легким для тестирования и лучшим в обработке ошибок.

Основная идея состоит в том, что сага - это как отдельный поток в вашем приложении, который несет полную ответственность за побочные эффекты.

С полной документацией по redux-saga можно ознакомится по ссылке: https://redux-saga.js.org/

В лабораторной работе необходимо реализовать проект со списками (лб 25) используя redux-saga. Код для сервера можно найти в методических указаниях к лабораторной работе 25.

Реализация

Устанавливаем библиотеки.

Листинг кода 29.1

```
npm install redux react-redux redux-saga axios
```

Далее необходимо создать папку actions с файлом для действий, где определили типы действий и определения действий.

Листинг кода 29.2

```
export const fetchListRequest = (state) => ({
    type: 'FETCH_LIST_REQUEST',
    payload: state
});

export const fetchListSuccess = (list) => ({
    type: 'FETCH_LIST_SUCCESS',
    payload: { list },
});
```

```
export const fetchListFailure = (error) => ({
    type: 'FETCH_LIST_FAILURE',
    payload: { error },
});
export const setPage = (page) => ({
    type: 'SET_PAGE',
    payload: { page },
});
export const setCount = (count) => ({
    type: 'SET_COUNT',
    payload: { count },
});
export const addName = (name) => ({
    type: 'ADD_NAME',
    payload: { name },
});

export const fetchAddFailure = (error) => ({
    type: 'FETCH_ADD_FAILURE',
    payload: { error },
});
```

Затем необходимо создать папку reducers с файлами для редукторов, где описываются функции для действий производимых на сайте (получения списка, получение ошибки, получение длинны списка, получение страницы и добавление)

Добавим начальное состояние

Листинг кода 29.3

```
const initialState = {
    list: [],
    page: 1,
    loading: false,
    error: null,
    limit: 5,
    count: 0,
    name: ""
};
```

И добавим обработку действия и возврат нового состояния.

Листинг кода 29.4

```
const listReducer = (state = initialState, action) => {
    switch (action.type) {
        case 'FETCH_LIST_REQUEST':
            return { ...state, loading: true, error: null };
        case 'FETCH_LIST_SUCCESS':
```

```
                return { ...state, loading: false, list:
action.payload.list };
            case 'FETCH_LIST_FAILURE':
                return { ...state, loading: false, error:
action.payload.error };
            case 'SET_PAGE':
                return { ...state, page: action.payload.page };
            case 'SET_COUNT':
                return { ...state, count: action.payload.count };
            case 'FETCH_ADD_FAILURE':
                return { ...state, loading: false, error:
action.payload.error };
            case 'ADD_NAME':
                console.log(action.payload.name)
                return { ...state, name: action.payload.name };
            default:
                return state;
        }
    };
```

Теперь создадим файл store.js. В файле пропишем настройку Redux Store.

Листинг кода 29.5 (store.js)

```
    const sagaMiddleware = createSagaMiddleware();
    const store = createStore(listReducer,
applyMiddleware(sagaMiddleware));
    sagaMiddleware.run(listSaga);
```

Затем создадим обработку (App.js).

Листинг кода 29.6

```
      const dispatch = useDispatch();
      const { page, loading, error, count } = useSelector((state)
=> state);
      const state = useSelector((state) => state);

      useEffect(() => {
        dispatch(fetchListRequest(state));
      }, [dispatch, page]);
      useEffect(() => {
        dispatch(setCount(count));
      }, []);
      if (loading) {
        return <div>Loading...</div>;
      }

      if (error) {
        return <div>Error: {error}</div>;
      }
```

Далее создаем компоненты (пагинация, список и форма)

Листинг кода 29.7 (пагинация)

```
 const Pagination = () => {
    const dispatch = useDispatch();
    const pageNumbers = []
    const { limit, count } = useSelector((state) => state);
    for (let i = 1; i <= Math.ceil(count / limit); i++) {
        pageNumbers.push(i)
    }

    const handlePageChange = (page) => {
        dispatch(setPage(page));
    };
    return (
        <div>
            <ul className='pagination' >
                {
                    pageNumbers.map((number) => (
                        <li className='page-item' key={number}>
                            <a  href='!#'  className='page-link'
onClick={() => handlePageChange(number)}>
                                {number}
                            </a>
                        </li>
                    ))
                }

            </ul>
        </div>
    )

}
```

Листинг кода 29.8 (вывод списка)

```
const List = () => {

    const {list} = useSelector((state) => state);
    return (
    <>
        <ul>
            {list.map((list) => (
                <li key={list.id}>{list.name}</li>
            ))}
        </ul>
        </>);
}
```

Листинг кода 329.9 (форма)

```
const Form = () => {
    const dispatch = useDispatch();
    const [name, setName] = useState("");
    const handleOnChange = (event) => {
        setName(event.target.value)
    }
    const handleSubmit = (e) => {
        dispatch(addName(e));
        setName('');
        alert("Вы добавили животное!!!")
    };
    return (
        <div>
            <form style={{ margin: "1%" }}>
                <div class="col-12">
                    <label for="name">Название</label>
                </div>
                <div class="col-12">
                    <input class="form-control"
                        id="name" name="name"
                        value={name}
                        onChange={(event)                    =>
handleOnChange(event)} />
                </div>
                <div class="col-12">
                    <button  class="btn  btn-primary"  style={{
margin: "1%" }} onClick={() => handleSubmit(name)}>Добавить</button>
                </div>
            </form>
        </div>
    );
}
```

Теперь в index.js пропишем доступ к store. Это предоставит возможность всем компонентов вашего приложения обращаться к store. Это происходит с помощью компонента Provider из библиотеки react-redux.

Листинг кода 29.10 (store.js)

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
```

```
  document.getElementById('root')
);
```

На рисунке 29.1 представлен результат работы.

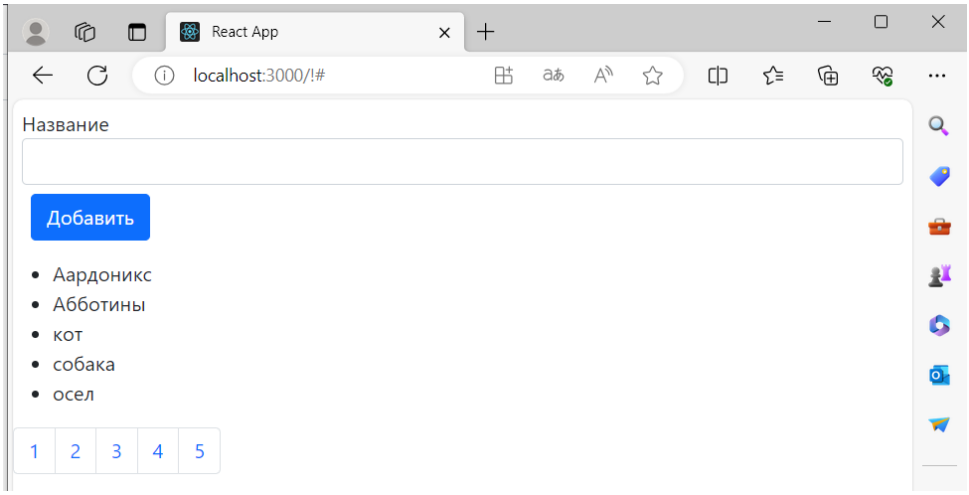

Рисунок 29.1

Задание для выполнения:

1. Ознакомитесь с методическими указаниями;

2. Реализавать лабораторной со списками (лб 34) в ReactJS с помощью redux-saga. На сайте должно быть реализовано:

- список товаров (элементов). Товар будет состоять из нескольких полей: Название, описание, фото и т.д.

- пагинация и отражение товаров (элементов).

- форма для добавления товаров (элементов). Форму сделать 2 библиотеками: bootstrap и любая другая.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое redux-saga?

2. Зачем нужен Redux?

3. Как настроить Redux? Способ legacy redux.

4. Как настроить Redux? Способ redux-toolkit.

5. Чем отличается способ настройки legacy redux от redux-toolkit?

30. ЛАБОРАТОРНАЯ РАБОТА №30. Верстка интерфейсов React

Цель работы: Использование библиотек react-bootstrap и material-ui для создания веб-приложений.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

React-Bootstrap заменяет Bootstrap JavaScript. Каждый компонент был создан с нуля как настоящий компонент React, без ненужных зависимостей, таких как jQuery.

Как одна из старейших библиотек React, React-Bootstrap развивалась и росла вместе с React, что делает ее отличным выбором в качестве основы вашего пользовательского интерфейса.

Material-UI — это библиотека компонентов для React, которая предлагает широкий выбор готовых компонентов для создания различных типов интерфейсов.

Среди компонентов есть такие важные элементы, как кнопки, карты, слайдеры, а также более сложные, такие как диалоги, сетки и подсказки.

Material-UI совместим с различными системами стилей и позволяет создавать пользовательские темы для приложения.

Material-UI и React-Bootstrap - это две популярные библиотеки компонентов пользовательского интерфейса для React. Вот несколько отличий между ними:

- Дизайн: Material-UI реализует графический стиль Material Design, разработанный Google, который предлагает чистый, плоский и карточный дизайн. React-Bootstrap, с другой стороны, предлагает компоненты, которые следуют стандартному стилю Bootstrap, который имеет более традиционный вид и ориентирован на мобильные устройства.

- Расширяемость: Material-UI обеспечивает большую степень настраиваемости компонентов, позволяя пользователю вмешиваться и изменять стили и поведение компонентов. React-Bootstrap более ограничен в плане расширяемости, хотя пользователи все равно могут изменять стили и некоторые свойства компонентов.

- Интеграция со сторонними библиотеками: Material-UI больше ориентирован на полноценную интеграцию с другими популярными библиотеками для React, такими как Redux или React Router. React-Bootstrap предлагает стандартную интеграцию с Bootstrap, но может иметь проблемы с интеграцией с другими библиотеками.

- Компоненты: Обе библиотеки предлагают широкий спектр готовых компонентов, таких как кнопки, формы, навигационные панели и т.д. Однако Material-UI имеет некоторые уникальные компоненты, специфичные для Material Design, такие как карточки и ленты.

В лабораторной работе мы рассмотрим, как создать компоненты:

- форма;

- панель навигации

- карточка;

- кнопка;

- карусель(слайдер);

-модальное окно;

- всплывающее окно.

Это только малая часть из всех компонентов этих библиотек с подробной информацией можно ознакомиться на официальных сайтах https://react-bootstrap.github.io/ и https://mui.com/material-ui/.

Настройка библиотек

React-Bootstrap

Сначала добавим библиотеку в ваш проект. Для добавления(установки) библиотеки react-bootstrap в ваше React-приложение необходимо истолковать команду

Листинг кода 30.1

```
npm install react-bootstrap bootstrap
```

Далее необходимо импортировать css в наш корневой файл (index.js).

Листинг кода 30.2

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

Material-Ui

Сначала добавим библиотеку в ваш проект. Для добавления(установки) библиотеки Material-Ui в ваше React-приложение необходимо истолковать команду

Листинг кода 30.3

```
npm install @mui/material @mui/styled-engine-sc styled-components
```

Далее необходимо добавить иконки (которыми в дальнейшем вы будете пользоваться) на ваш сайт

Листинг кода 30.4

```
npm install @mui/icons-material
```

Когда речь заходит о Material Design, шрифтом по умолчанию является Roboto. Однако material-ui не поставляется с Roboto по умолчанию. Импортируем его.

Листинг кода 30.5

```
npm install @fontsource/roboto
```

Далее необходимо импортировать его наш корневой файл (index.js).

Листинг кода 30.6

```
import '@fontsource/roboto/340.css';
import '@fontsource/roboto/400.css';
import '@fontsource/roboto/500.css';
import '@fontsource/roboto/700.css';
```

Вызов компонентов

Существует несколько способов вызвать компоненты:

- Импортировать непосредственно компонент в библиотеки:

Листинг кода 30.7

```
import Button from 'react-bootstrap/Button';
import Button from '@mui/material/Button';
```

- Импортировать из библиотеки:

Листинг кода 30.8

```
import { Button } from ' react-bootstrap';
import { Button } from '@mui/material';
```

Компонента React-Bootstrap

Панель навигации (<Nav>).

В панели навигации мы можем создать кнопки (variant="pills"), ссылку (<Nav.Link>), выпадающий список (<NavDropdown>) , так же мы можем отключить ссылку(необходимо добавить свойство disabled).

Можем добавить спили:

- Положение, по умолчание выравнивание по левому краю. Для изменения положение добавляем класс в тег Nav: по центру (justify-content-center) и по в правому краю (justify-content-end). Мы можем сделать меню навигации в виде вскладок для этого нам необходимо в теге Nav прописать variant="tabs".

- В колонку, а не строкой как выводится по умолчанию. Для изменения положение добавляем класс flex-column в тег Nav.

Вариации исполнения:

- Вкладки. Для этого нам необходимо в теге Nav добавить variant="tabs".

-Таблетки. Для этого нам необходимо в теге Nav добавить variant=" pills".

-Подчеркивание. Для этого нам необходимо в теге Nav добавить variant=" Underline".

- Увеличенное панель навигации растянуто на всю длину странице. Для этого нам необходимо в теге Nav добавить fill.

- Сделать элементы одинакового размера. Для этого нам необходимо в теге Nav добавить justify.

Так же можем добавить действие, которое хотим, чтобы происходило при нажатие на ссылки в панели навигации. Для этого нам необходимо добавить событие в тег Nav (onSelect={}) и обработчик события.

Пример кода представлен ниже.

Листинг кода 30.9

```
const handleSelect = (eventKey) => alert(`selected ${eventKey}`);
…
<Nav variant="pills" activeKey="1" onSelect={handleSelect}>
            <Nav.Item>
                <Nav.Link eventKey="1" href="#/home">
                    Переход 1
                </Nav.Link>
            </Nav.Item>
            <Nav.Item>
                <Nav.Link eventKey="2" title="Item">
                    Переход 2
                </Nav.Link>
            </Nav.Item>
            <Nav.Item>
                <Nav.Link eventKey="3" disabled>
```

```
                        Переход 3
                    </Nav.Link>
                </Nav.Item>
                <NavDropdown title=" Меню" id="nav-dropdown">
                    <NavDropdown.Item
eventKey="4.1">Действие</NavDropdown.Item>
                    <NavDropdown.Item          eventKey="4.2">Другое
действие</NavDropdown.Item>
                    <NavDropdown.Item  eventKey="4.3">Здесь  что-то
еще</NavDropdown.Item>
                    <NavDropdown.Divider />
                    <NavDropdown.Item     eventKey="4.4">Разделенная
ссылка</NavDropdown.Item>
                </NavDropdown>
            </Nav>
```

Кнопка (<Button>).

У кнопок существует несколько вариаций:

- по цветам (primary, secondary, success, warning, danger, info, light, dark, link).

- цвет контура (outline-primary, outline-secondary, outline-success, outline-warning, outline-danger, outline-info, outline-light, outline-dark, outline-link)

Можно указать еще размер: большая (size="lg") и маленькая (size="sm").

Листинг кода 30.10

```
<Button variant="primary" type="submit">
    Кнопка
</Button>
```

Форма (<Form>).

Для начала необходимо вызвать компонент Form. <FormControl>Компонент отображает элемент управления формой в стиле Bootstrap.

<Form.Group> Компонент обертывает элемент управления формой с соответствующим интервалом, а также поддерживает метку, текст справки и состояние проверки. Чтобы обеспечить доступность, установите controlId вкл. <Form.Group>, и используйте <Form.Label> для метки.

Листинг кода 30.11

```
<Form>
                                    <FormGroup     className="mb-3"
controlId="formBasicEmail">
                    <FormLabel>Почта</FormLabel>
                      <FormControl type="email" placeholder="Введите
почту" />
```

```
                    <FormText className="text-muted">
                          Мы никогда не передадим вашу электронную
почту кому-либо еще
                    </FormText>
                </FormGroup>
                                        <FormGroup    className="mb-3"
controlId="formBasicPassword">
                    <FormLabel>Пароль</FormLabel>
                                         <FormControl   type="password"
placeholder="Введите пароль" />
                </FormGroup>
                                        <FormGroup    className="mb-3"
controlId="formBasicCheckbox">
                     <FormCheck type="checkbox" label="Нажми меня)"
/>
                </FormGroup>
                <Button variant="primary" type="submit">
                    Кнопка
                </Button>
            </Form>
```

Карточка (<Card>).

Карточка состоит из нескольких частей: шапка, тело(заголовок и остновной текст) и подвал. Если вам необходимо, то в карточку можно добавить фото, панель навигацию, список, кнопку, слайдер и другие элементы.

Так же у карточки можно прописать и стиль. Добавить цвет фона (bg

="primary"), добавить контур (border="primary").

Карточки можно собрать в группу с помощью тега <CardGroup>.

Листинг кода 30.12

```
 <CardGroup>
        <Card>
          <Card.Img variant="top" src="holder.js/100px160" />
          <Card.Body>
            <Card.Title>Card title</Card.Title>
            <Card.Text>
                This is a wider card with supporting text below
as a natural lead-in
                to additional content. This content is a little
bit longer.
            </Card.Text>
          </Card.Body>
          <Card.Footer>
              <small className="text-muted">Last updated 3 mins
ago</small>
          </Card.Footer>
        </Card>
        <Card>
```

```
              <Card.Img variant="top" src="holder.js/100px160" />
              <Card.Body>
                <Card.Title>Card title</Card.Title>
                <Card.Text>
                   This card has supporting text below as a natural
lead-in to
                  additional content.{' '}
                </Card.Text>
              </Card.Body>
              <Card.Footer>
                  <small className="text-muted">Last updated 3 mins
ago</small>
              </Card.Footer>
            </Card>
            <Card>
              <Card.Img variant="top" src="holder.js/100px160" />
              <Card.Body>
                <Card.Title>Card title</Card.Title>
                <Card.Text>
                   This is a wider card with supporting text below
as a natural lead-in
                   to additional content. This card has even longer
content than the
                  first to show that equal height action.
                </Card.Text>
              </Card.Body>
              <Card.Footer>
                  <small className="text-muted">Last updated 3 mins
ago</small>
              </Card.Footer>
            </Card>
          </CardGroup>
```

Карусель(слайдер).

1 слайд карусель состоит из фото и надписи на ней. Чтобы анимировать слайды с плавны переходом к тегу <Carousel > добавляем fade. Чтобы отключить анимацию по умолчанию в теге <Carousel > пропишем slide={false}. Еще мы можем добавить интервлы между переключением слайдов, для этого необходимо в теге <Carousel > прописать interval={_кол-во миллисекунд_}. Так же мы можем установить темную тему для кнопок для этого в теге <Carousel > пропишем data-bs-theme="dark".

Листинг кода 30.13

```
<Carousel data-bs-theme="dark">
    <Carousel.Item>
      <img
        className="d-block w-100"
```

```
src="https://catherineasquithgallery.com/uploads/posts/3234-
03/1614613433_137-p-fon-dlya-fotoshopa-priroda-333.jpg"
            alt="First slide"
          />
          <Carousel.Caption>
            <h5>First slide label</h5>
            <p>Nulla vitae elit libero, a pharetra augue mollis
interdum.</p>
          </Carousel.Caption>
        </Carousel.Item>
        <Carousel.Item>
          <img
            className="d-block w-100"

src="https://catherineasquithgallery.com/uploads/posts/3234-
03/1614613433_137-p-fon-dlya-fotoshopa-priroda-333.jpg"
            alt="Second slide"
          />
          <Carousel.Caption>
            <h5>Second slide label</h5>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit.</p>
          </Carousel.Caption>
        </Carousel.Item>
        <Carousel.Item>
          <img
            className="d-block w-100"

src="https://catherineasquithgallery.com/uploads/posts/3234-
03/1614613433_137-p-fon-dlya-fotoshopa-priroda-333.jpg"
            alt="Third slide"
          />
          <Carousel.Caption>
            <h5>Third slide label</h5>
            <p>
              Praesent commodo cursus magna, vel scelerisque nisl
consectetur.
            </p>
          </Carousel.Caption>
        </Carousel.Item>
      </Carousel>
```

Модальное окно (< Modal>).

Модальное окно состоит из нескольких частей: шапка, тело и подвал. Чтобы открыть модальное окно надо к кнопке добавить событие (клик) и обработчиком события вызвать элемент модальное окно.

Листинг кода 30.14

```
const [show, setShow] = useState(false);

   const handleClose = () => setShow(false);
   const handleShow = () => setShow(true);

   return (
     <>
       <Button variant="primary" onClick={handleShow}>
         Модальное окно
       </Button>

       <Modal show={show} onHide={handleClose}>
         <Modal.Header closeButton>
           <Modal.Title>Модальное окно</Modal.Title>
         </Modal.Header>
         <Modal.Body>Текст модального окна</Modal.Body>
         <Modal.Footer>
           <Button variant="secondary" onClick={handleClose}>
             Закрыть
           </Button>
           <Button variant="primary" onClick={handleClose}>
             Сохранить
           </Button>
         </Modal.Footer>
       </Modal>
       </>
   )
```

На рисунках 30.1 – 30.2 представлен результат работы, где собраны все элементы описание ранее.

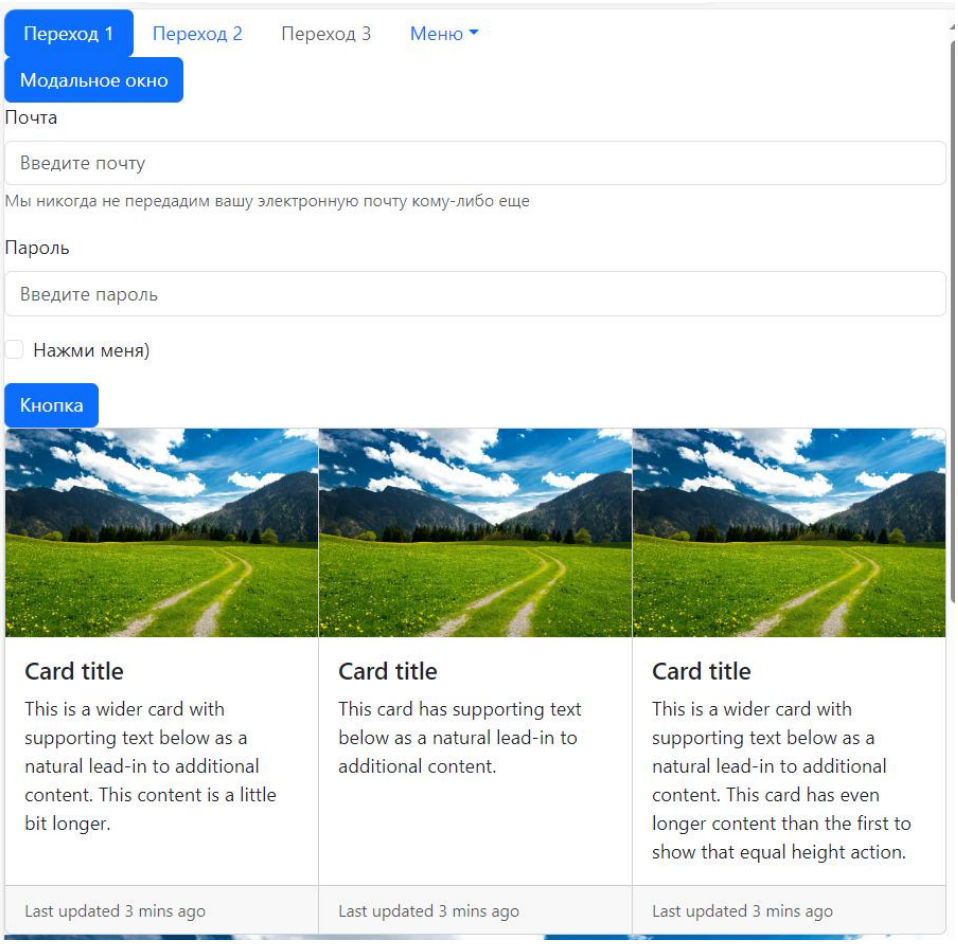

Рисунок 30.1 - компоненты библиотеки react-bootstrap

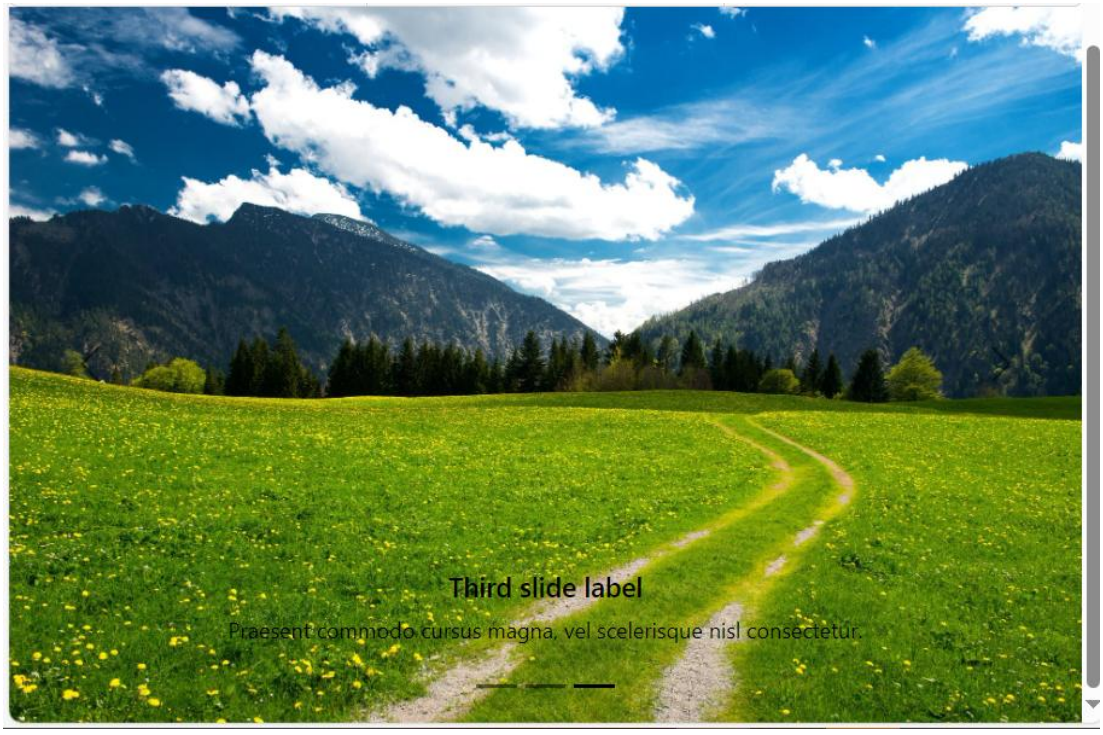

Рисунок 30.2 - компоненты библиотеки react-bootstrap

Добавляйте стили с помощью sx prop

Все компоненты Material UI принимают sx prop, который дает вам доступ к сокращенному синтаксису для написания CSS. Он отлично подходит для создания разовых настроек или быстрых экспериментов с различными стилями.

Листинг кода 30.15

```
<Sheet
  sx={{
    width: 340,
    mx: 'auto', // margin left & right
    my: 4, // margin top & bottom
    py: 3, // padding top & bottom
    px: 2, // padding left & right
    display: 'flex',
    flexDirection: 'column',
    gap: 2,
    borderRadius: 'sm',
    boxShadow: 'md',
  }}
>
  Welcome!
</Sheet>
```

Добавьте текст с помощью компонента Typography

Компонент Типография заменяет теги HTML header, paragraph и span, помогая поддерживать согласованную иерархию текста на странице.

level Prop предоставляет вам доступ к заранее определенной шкале типографских значений. Material UI предоставляет 11 готовых типографских уровней.

- Четыре уровня заголовка: 'h1' | 'h2' | 'h3' | 'h4'

- Три уровня названий: 'title-lg' | 'title-md' | 'title-sm'

- Четыре уровня тела: 'body-lg' | 'body-md' | 'body-sm' | 'body-xs'

Листинг кода 30.16

```
  <Typography level="h4" component="h1">
    Welcome!
  </Typography>
  <Typography level="body-sm">Sign in to continue.</Typography>
```

Панель навигации.

Для панели навигации нет отдельного компонента. Для тоже чтобы ее создать можно собрать несколько компонентов вместе, т.к. <Menu>, <Link> и т.д.

Кнопка (<Button>).

Есть несколько вариаций кнопок:

- базовые кнопки тег <Button>

- кнопки иконки тег <IconButton>

У базовых кнопок существует несколько вариаций:

- variant="text"

- variant="contained"

- variant="outlined"

Также у кнопок есть размер:

-size="small"

- size="medium

- size="large"

Листинг кода 30.17

```
<Button variant="text">Text</Button>
<Button variant="contained">Contained</Button>
<Button variant="outlined">Outlined</Button>
<Button size="small" variant="contained">Small</Button>
<Button size="medium" variant="contained">Medium</Button>
<Button size="large" variant="contained">Large</Button>
```

Пример кнопок продемонстрирован на рисунке 30.3.

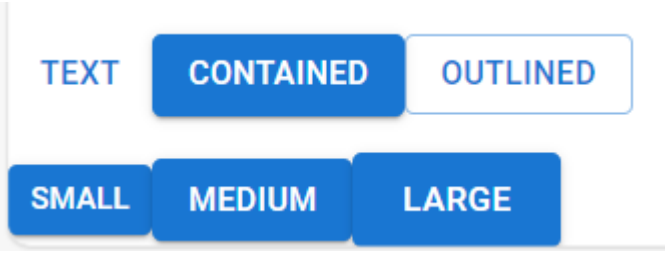

Рисунок 30.3

Можно еще добавить в начало или конец добавить иконку.

Для добавления иконки в конец и начало и кнопка иконка необходимо импортировать иконку из библиотеки @mui/icons-material.

Листинг кода 30.18

```
<Button variant="outlined" startIcon={<DeleteIcon />}>
        Delete
</Button>
<Button variant="contained" endIcon={<SendIcon />}>
        Send
</Button>
<IconButton aria-label="delete" size="small">
        <DeleteIcon fontSize="small" />
</IconButton>
```

Пример кнопок продемонстрирован на рисунке 30-.4.

Рисунок 30.4

Форма.

В данной библиотеки нет отдельного компонента для тоже чтобы написать форму необходимо добавить поля ввода (например, Input) и собрать с помощью тега <FormGroup>. Каждый Input необходимо обернуть в контролер.

Листинг кода 30.19

```
<FormGroup sx={{ maxWidth: "50%" }}>
            <FormLabel>Форма</FormLabel>
            <FormControl>
                <FormLabel>Почта</FormLabel>
                <Input
                    // html input attribute
                    name="email"
                    type="email"
                    placeholder="Введите вашу почту"
                />
            </FormControl>
            <FormControl>
                <FormLabel>Пароль</FormLabel>
                <Input
                    name="password"
                    type="password"
                    placeholder="Введите пароль"
                />
            </FormControl>
```

```
            </FormGroup>
```

Карточка (<Card>).

Компонент Material UI Card включает в себя несколько дополнительных служебных компонентов для обработки различных вариантов использования:

- <Card>: контейнер на уровне поверхности для группировки связанных компонентов.

- <CardContent>: оболочка для содержимого карточки.

- <CardHeader>: необязательная оболочка для заголовка карточки.

- <CardMedia>: дополнительный контейнер для отображения фоновых изображений и градиентных слоев за содержимым карты.

- <CardActions>: необязательная оболочка, которая группирует набор кнопок.

- <CardActionArea>: необязательная оболочка, позволяющая пользователям взаимодействовать с указанной областью карты.

Листинг кода 30.20

```
  <>
      <Card sx={{ maxWidth: 345 , float:"left"}}>
      <CardMedia
        component="img"
        alt="green iguana"
        height="140"

image="https://catherineasquithgallery.com/uploads/posts/3234-
03/1614613433_137-p-fon-dlya-fotoshopa-priroda-333.jpg"
      />
      <CardContent>
        <Typography gutterBottom variant="h5" component="div">
          Lizard
        </Typography>
        <Typography variant="body2" color="text.secondary">
          Lizards are a widespread group of squamate reptiles,
with over 6,000
          species, ranging across all continents except
Antarctica
        </Typography>
      </CardContent>
      <CardActions>
        <Button size="small">Share</Button>
        <Button size="small">Learn More</Button>
      </CardActions>
    </Card>
    <Card sx={{ maxWidth: 345 }}>
      <CardMedia
        component="img"
        alt="green iguana"
```

```
            height="140"
image="https://catherineasquithgallery.com/uploads/posts/3234-
03/1614613433_137-p-fon-dlya-fotoshopa-priroda-333.jpg"
          />
          <CardContent>
            <Typography gutterBottom variant="h5" component="div">
              Lizard
            </Typography>
            <Typography variant="body2" color="text.secondary">
              Lizards are a widespread group of squamate reptiles,
with over 6,000
              species, ranging across all continents except
Antarctica
            </Typography>
          </CardContent>
          <CardActions>
            <Button size="small">Share</Button>
            <Button size="small">Learn More</Button>
          </CardActions>
        </Card>
          </>
```

Модальное окно (< Modal>).

Чтобы открыть модальное окно надо к кнопке добавить событие (клик) и обработчиком события вызвать элемент модальное окно.

Листинг кода 30.21

```
   const [open, setOpen] = useState(false);
  const handleOpen = () => setOpen(true);
  const handleClose = () => setOpen(false);

  const style = {
    position: 'absolute',
    top: '50%',
    left: '50%',
    transform: 'translate(-50%, -50%)',
    width: 400,
    bgcolor: 'background.paper',
    border: '2px solid #000',
    boxShadow: 32,
    p: 4,
  };
  return (
    <>
      <Button onClick={handleOpen}>Open modal</Button>
      <Modal
        open={open}
        onClose={handleClose}
        aria-labelledby="modal-modal-title"
```

```
            aria-describedby="modal-modal-description"
          >
            <Box sx={style}>
              <Typography    id="modal-modal-title"    variant="h6"
component="h2">
                Text in a modal
              </Typography>
              <Typography id="modal-modal-description" sx={{ mt: 2
}}>
                Duis mollis, est non commodo luctus, nisi erat
porttitor ligula.
              </Typography>
            </Box>
          </Modal>
        </>
      )
```

На рисунке 30.5 представлен результат работы, где собраны все элементы описание ранее.

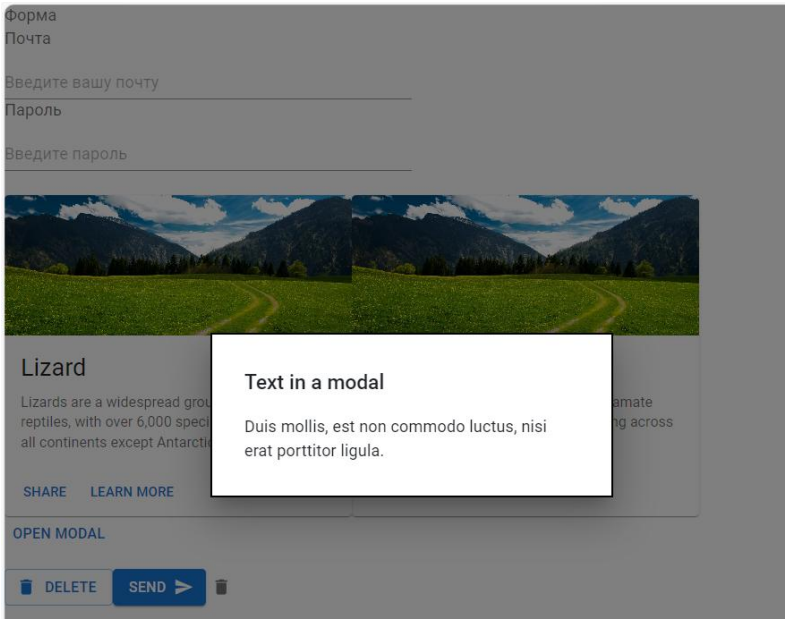

Рисунок 30.5- компоненты библиотеки Material-Ui

Задание для выполнения:

1. Ознакомитесь с методическими указаниями;

2. Сверстать сайт с использованием библиотеки react-bootstrap. Обязательные требования:

- обязательные элементы: панель навигации, форма, карточки, карусель, модальное окно;

- на сайт добавить 2 любых компонента которые не были рассмотрены в методичке.

2. Сверстать сайт с использованием библиотеки Material-Ui. Обязательные требования:

- обязательные элементы: панель навигации, форма, карточки, модальное окно;
- на сайт добавить 2 любых компонента которые не были рассмотрены в методичке.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое react-bootstrap?
2. Что такое Material-Ui?
3. В чем отличие Material-Ui от react-bootstrap?

31. ЛАБОРАТОРНАЯ РАБОТА №31. Тестирование React-приложений

Цель работы: Использование библиотеки Jest.

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Написание unit-тестов является крайне желательным (а иногда и обязательным) процессом при разработке приложения. Наличие тестов позволяет проверить корректность работы старого функционала при внесении изменений или правильность работы новых функций. Одним из инструментов для работы с unit-тестами для js является Jest.

Видим, что тест прошёл, однако, так как включена проверка покрытия кода тестами, выводится информация о том, что не все функции протестированы.

Для проверки результата выполнения функции есть так же и другие функции:

- toBe() подходит для примитивных данных типа чисел;

- toEqual() подойдёт для более сложных объектов;

-toContain() проверяет наличие определёного элемента в массиве (примитивы);

- toContainEqual() проверяет наличие сложного элемента в массиве;

- toHaveLength() проверят значение свойства length (например у строк и ли массивов)

- noBeNull() проверяет на null;

- toBeUndefined() проверяет на undefined;

- toBeDefined() противоположность toBeUndefined();

- toBeTruthy() проверка на соответствие True;

- toBeFalsy() проверка на соответствие False;

- toBeGreaterThan() и toBeGreaterThanOrEqual() проверяет на «Больше» и «Больше или равно» соответственно.

- toBeLessThan() и toBeLessThanOrEqual() - «Меньше» и «Меньше или равно» соответственно;

- toMatch() - проверяет на соответствие регулярному выражению;

- toThrow() - проверяет выбрасываемое исключение. Можно проверить как факт выброса ошибки, так и класс самой ошибки.

not ставится перед указанными выше методами. С данным модификатором они будут работать наоборот.

В лабораторной работе необходимо подключить тестирование к вашему проекту (курсовому проекту).

Ниже будет пример реализации для лабы с маршрутизацией (лб 34).

Реализация

1. Устанавливает пакеты. Мы используем пакет babel-jest и Babel прежде для react, чтобы преобразовать код внутри окружения для тестирования.

Листинг 31.1

```
npm install --save-dev jest react react-test-renderer
```

2. Создадим файл babel.config.js в корне нашего проекта.

Листинг 31.2

```
module.exports = {
    presets: [
      '@babel/preset-env',
      ['@babel/preset-react', {runtime: 'automatic'}],
    ],
  };
```

3. Добить в файл package.json метод для тестирования в "scripts": {}.

Листинг 31.3

```
…
"test": "jest",
…
```

4. Далее создадим тест для страницы Home, в котором проверяется правильно ли отрисовался компонент и имеет внутри себя элемент h1 с текстом "Главная страница".

Листинг 31.4

```
import renderer from 'react-test-renderer';
import Home from './Home';
//С помощью функции describe() создается блок тестов для компонента Home.
describe('Компонент дом', () => {
    // Функция test задает конкретное поведение, которое мы проверяем: "отрисовывается ли компонент Home правильно".
    test('it renders', () => {
        // Внутри теста создается экземпляр компонента Home с помощью renderer.create(<Home />).
        const component = renderer.create(<Home />);
```

```
            //Получаем корневой элемент компонента с помощью
component.root.
            const instance = component.root;
            //С помощью expect и findByType проверяется, содержит
ли компонент h1 заголовок с текстом "Главная страница".
     expect(instance.findByType("h1").props.children).toBe("Главная
страница");
         });
     })
```

Задание для выполнения:

1. Ознакомиться с материалом методического указания.

2. Написать 5 тестов для вашего сайта. Обязательно проверить исключительные ситуации, которые могут возникнуть и добавить их обработку.

3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое фреймворк Jest?

2. Что такое unit-тест?

3. Какие есть функции для проверки результата выполнения функции?

32. ЛАБОРАТОРНАЯ РАБОТА №32. Анимация

Цель работы: Создание анимация в React-приложениях разными способами: с помощью компонента ReactTransitionGroup, библиотеки React-animations, фреймворка React-reveal

Формируемые компетенции: ОК 09, ПК 3.4

Теоритический материал

Анимация - это процесс создания иллюзии движения путем последовательного отображения статических изображений или объектов. В контексте веб-разработки, анимации часто используются для добавления эффектов и динамичности на веб-сайтах и веб-приложениях.

Анимация может применяться к различным элементам, таким как текст, изображения, кнопки, фоны и другие элементы интерфейса. Существует множество различных видов анимации, таких как перемещение, изменение размера, изменение цвета, поворот, затухание и другие эффекты, которые могут быть использованы для достижения конкретных целей и создания привлекательного визуального опыта для пользователей.

Анимации могут быть созданы с использованием различных технологий и инструментов, таких как CSS анимации, JavaScript библиотеки (например, React-animations, GreenSock Animation Platform), SVG анимации и другие. Кроме того, существуют специальные фреймворки и библиотеки, которые облегчают создание и управление анимациями, делая процесс более простым и эффективным для разработчиков.

В лабораторной работе рассмотрим несколько способов для создание анимациия в React-приложение:

1.компонент ReactTransitionGroup.

2. библиотека React-animations.

3. фреймворка React-reveal.

Компонент ReactTransitionGroup

ReactTransitionGroup - это набор компонентов и утилит для создания анимаций в React. Он позволяет анимировать появление, исчезновение и обновление компонентов. Для создания анимации с помощью ReactTransitionGroup, необходимо использовать компоненты

CSSTransition и TransitionGroup. CSSTransition позволяет добавлять CSS-классы для анимации, а TransitionGroup позволяет группировать компоненты для анимации.

Компоненты:

- Transition– позволяет вам описать переход от одного состояния компонента к другому во времени с помощью простого декларативного API. По умолчанию не изменяет поведения компонента, только отслуживает состояние "enter" и "exit" для компонентов. Состояния, в которых находится Transition: entering, entered, exiting, exited. Состояние Transition переключается с помощью параметра in. При значении true компонент начинает стадию "Enter". Во время этого этапа компонент переходит из текущего состояния перехода в состояние "entering" на время перехода, а затем в состояние "entered" после его завершения.

- Transition– Компонент перехода, созданный на основе замечательной библиотеки ng-animate. Его следует использовать, если вы используете CSS-переходы или анимацию. Он создан на основе компонента Transition, поэтому наследует все его пропсы. CSSTransition применяет пару имен классов во время состояний appear, enter и exit перехода. Применяется первый класс, а затем второй класс *-active, чтобы активировать CSS-переход. После перехода применяются соответствующие имена классов *-done, чтобы сохранить состояние перехода.

- SwitchTransition – Компонент перехода, вдохновленный vue transition modes. Вы можете использовать его, когда хотите управлять рендерингом между переходами состояний. Основываясь на выбранном режиме и дочернем ключе, который является компонентом Transition или CSSTransition, SwitchTransition осуществляет последовательный переход между ними. Если выбран режим out-in, SwitchTransition ждет, пока старый ребенок не выйдет, а затем вставляет нового ребенка. Если выбран режим in-out, SwitchTransition сначала вставляет нового ребенка, ждет, пока новый ребенок войдет, а затем удаляет старого ребенка.

- TransitionGroup – Компонент <TransitionGroup> управляет набором компонентов перехода (<Transition> и <CSSTransition>) в списке. Как и компоненты перехода, <TransitionGroup> представляет собой машину состояний для управления монтированием и размонтированием компонентов во времени. Обратите внимание, что <TransitionGroup> не определяет никакого поведения анимации! То, как именно анимируется элемент списка, зависит от отдельного компонента перехода. Это означает, что вы можете смешивать и сочетать анимации для разных элементов списка.

Библиотека React-animations

React-animations - это библиотека с предопределенными анимацией, которые могут быть легко добавлены к компонентам React. Для использования библиотеки React-animations,

необходимо установить ее через npm или yarn и импортировать нужную анимацию в свой компонент.

Использование:

- Использование с Radium с компонентами Radium, {StyleRoot}.

Импортируем необходимые компоненты.

Листинг кода 32.1

```
import { bounce } from 'react-animations';
import Radium, { StyleRoot } from 'radium';
```

Создаем логику анимации

Листинг кода 32.2

```
        <StyleRoot>
          <div   className="test"   style={{   animation:   'x   1s',
animationName: Radium.keyframes(bounce, 'bounce') }}>
            Анимация
          </div>
        </StyleRoot>
```

- Использование с Aphrodite

Импортируем необходимые компоненты.

Листинг кода 32.3

```
import { bounce } from 'react-animations';
import { StyleSheet, css } from 'aphrodite';
```

Создаем логику анимации

Листинг кода 32.4

```
  const styles = StyleSheet.create({
    bounce: {
      animationName: bounce,
      animationDuration: '1s',
    },
  });
  return (
    <div className={css(styles.bounce)}>Animated content</div>
  );
```

- Использование с styled-components

Импортируем необходимые компоненты.

Листинг кода 32.5

```
import styled, { keyframes } from 'styled-components';
import { fadeIn } from 'react-animations';
```

Создаем логику анимации

Листинг кода 32.6

```
// Создаем ключевой кадр для анимации fadeIn
  const fadeInAnimation = keyframes`${fadeIn}`;

  // Создаем стилизованный компонент с применением анимации
  const AnimatedComponent = styled.div`
  animation: ${fadeInAnimation} 1s;
  // Применяем анимацию fadeIn на протяжении 1 секунды `;
  return (
    <AnimatedComponent>
      <p>Анимация</p>
    </AnimatedComponent>
```

Компоненты: bouceOut, bounce, bounceIn, bounceInDown, bounceInLeft, bounceInRight, bounceInUp, bounceOutDown, bounceOutLeft, bounceOutRight, bounceOutUp, fadeIn, fadeInDown, fadeInDownBig, fadeInLeft, fadeInLeftBig, fadeInRight, fadeInRightBig, fadeInUp, fadeInUpBig, fadeout, fadeOutDown, fadeOutDownBig, fadeOutLeft и т.д.

Фреймворк React-reveal

React-reveal - это фреймворк для создания анимаций в React-приложениях с простым API. Он предоставляет компоненты для различных видов анимаций, таких как Fade, Zoom, Slide и другие. Для использования фреймворка React-reveal, необходимо установить его через npm или yarn и импортировать нужный компонент анимации в свой компонент.

Некоторые компоненты:

- Fade – анимация исчезновения (можно указать направление).

- Flip – перевернутая анимация (можно указать направление, задержка в анимации).

- Rotate – поворот анимация (можно указать направление, задержка в анимации).

- Zoom – анимация масштабирования (можно указать направление, задержка (delay) в анимации и замедление (duration)).

- Bounce – анимация отскока (можно указать направление, задержка (delay) в анимации и замедление (duration)).

- Reveal – анимация CSS, компонент effect (принимает строковое значение, которое определяет название используемого эффекта анимации CSS).

Общие реквизиты для некоторых компонентов:

- duration – указывает время анимации в миллисекундах.

- delay - устанавливает задержку перед началом анимации в миллисекундах.

- count - указывает сколько раз должна выполняться анимация.

- forever - устанавливает значение true, анимация будет выполняться бесконечно.

- when – указывает логическое условие, которое определяет, когда должна выполняться анимация.

Реализация

Компонент ReactTransitionGroup

1.Установим библиотеку.

Листинг кода 32.7

```
npm i react-transition-group
```

2. Теперь используя компонент CSSTransition можем создать анимацию.

Листинг кода 32.8

```
import React, { useState } from 'react';
import { CSSTransition } from 'react-transition-group';
import './App.css';

const App = () => {
  const [showComponent, setShowComponent] = useState(false);

  return (
    <div>
      <button onClick={() => setShowComponent(!showComponent)}>
        Toggle Component
      </button>
      <CSSTransition
        in={showComponent}
        timeout={340}
        classNames="fade"
        unmountOnExit
      >
        <div className="box">
          <p>This box will animate</p>
        </div>
```

```
      </CSSTransition>
    </div>
  );
};

export default App;
```

3. Затем пропишим анимацию при разных состояниях в App.css.

Листинг кода 32.9

```
.fade-enter {
  opacity: 0;
}

.fade-enter-active {
  opacity: 1;
  transition: opacity 340ms;
}

.fade-exit {
  opacity: 1;
}

.fade-exit-active {
  opacity: 0;
  transition: opacity 340ms;
}

.box {
  width: 320px;
  height: 320px;
  background-color: lightblue;
}
```

Библиотека React-animations

1. Установим библиотеку.

Листинг кода 32.10

```
npm i react-animations
```

2. Далее импортируем вспомогательную библиотеку. Теперь используя компонент fadeInDown можем создать анимацию.

Листинг кода 32.11

```
import React from 'react';
```

```
import { fadeInDown } from 'react-animations';
import { StyleSheet, css } from 'aphrodite';

const styles = StyleSheet.create({
  bounce: {
    animationName: fadeInDown,
    animationDuration: '1s',
  },
});

const App = () => {
  return (
    <div className={css(styles.bounce)}>Анимация</div>
  );
};

export default App;
```

Фреймворк React-reveal

1. Установим библиотеку.

Для версий React 33 и выше.

Листинг кода 32.12

```
npm i react-reveal --legacy-peer-deps
```

Для версий React ниже 33.

Листинг кода 32.13

```
npm i react-reveal
```

2. Теперь используя компонент Fade можем создать анимацию. Указывая направление «вылета» компонента.

Листинг кода 32.14

```
import React from 'react';
import {Fade} from 'react-reveal';

const App = () => {
  return (
    <Fade left>
      <div>Анимация</div>
    </Fade>
  );
};
```

```
export default App;
```

Задание для выполнения:

1. Ознакомитесь с методическими указаниями;
2. Добавить на ваш сайт анимацию с помощью:
    - Компонент ReactTransitionGroup
    - Библиотека React-animations
    - Фреймворк React-reveal
3. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое компонент ReactTransitionGroup?
2. Что такое библиотека React-animations
3. Что такое фреймворк React-reveal?
4. Что такое анимация?
5. К каким элементам можно применить анимацию?
6. Какие есть виды анимации?
7. Какими способами можно реализовать анимацию?

33. ЛАБОРАТОРНАЯ РАБОТА №33. Общедоступные приложения

Цель работы: Создание веб-приложения согласно правил создания веб-приложений для инвалидов по зрению, слуху, дальтоников итд

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Доступность контент – специальные архитектурные и технические решения, которые обеспечивают людям с ограниченными возможностями возможность использовать сайт.

Термин «доступность контента» также может обозначаться аббревиатурой a11y.

В React поддерживает создание сайтов с доступным контентом в том числе с помощью стандартных возможностей HTML.

Рекомендации к созданию доступности контентна определяется руководством по обеспечению доступности контента (WCAG).

WCAG разработанное консорциумом W3C, описывает рекомендации по созданию сайтов с доступным контентом.

Также есть ресурсы с чек-листами требований WCAG.

Некоторые из них:

- рекомендации от Wuhcag (https://www.wuhcag.com/wcag-checklist/)
- рекомендации от WebAIM (https://webaim.org/standards/wcag/checklist)
- рекомендации от A11Y Project (https://www.a11yproject.com/checklist/)

Существует несколько уровней доступности A(начинающий), AA(средний), AAA(продвинутый). Ниже мы рассмотрим начинающий уровень доступности. Для ознакомления со свеми уровнями следует ознакомится с руководством по обеспечению доступности контента (WCAG) см. ссылку: https://www.w3.org/Translations/WCAG32-ru/#media-equiv

Уровень A (начинающий)

1. Нетекстовое содержимое - всегда использовать текстовую альтернативу для нетекстового контента.

2. Только аудио- и видеоконтент (в записи) – обеспечивать альтернативу контенту только для видео- и аудиоконтенту.

3. Подписи (в записи) – всегда сопровождать субтитры со звуком.

4. Описание аудио или альтернативный носитель (в записи) – всегда представлять аудио-описание или текстовую расшифровку для видео со звуком.

5. Информация и взаимосвязи - Программно можно определить содержание, структуру и взаимосвязи информации.

6. Значимая последовательность - Поддерживайте логическую последовательность контента для его более понятного восприятия.

7. Сенсорные характеристик - Инструкции по использованию контента необходимо основывать не только на его сенсорных характеристиках.

8. Использование цвета - Используйте цвет не как единственное средство передачи информации или выделения элементов на экране.

9. Аудиоуправление - При наличии аудиофайлов длительностью более 3 секунд следует обеспечить доступ к кнопкам паузы, остановки и управления громкостью независимо от общей громкости.

10. Клавиатура - Обеспечьте полный доступ ко всем функциям с помощью клавиатуры без дополнительных настроек времени.

11. Полное управление с клавиатуры - Пользователи должны иметь возможность управлять всем контентом с клавиатуры, используя клавиши Tab или комбинации Tab и Shift. Если требуется более одного нажатия клавиши Tab или Tab в сочетании с другими клавишами, предоставьте соответствующую инструкцию пользователю.

12. Настройка времени - Для установленных ограничений времени в контенте должно быть соблюдено как минимум одно из следующих:

- Выключение - пользователь может отключить ограничение до его окончания;
- Настройка - пользователь может изменить ограничение до окончания, увеличив временной лимит минимум в 10 раз;
- Дополнительное время - через каждые 20 секунд пользователь получает уведомление о скором окончании времени и может продлить его простым действием, таким как нажатие на клавишу пробела; возможность продления времени минимум 10 раз подряд;
- Кроме режима реального времени - в случае, когда временное ограничение неотъемлемая часть мероприятия, происходящего в реальном времени (например, аукцион), где изменение времени невозможно;
- Кроме случаев особого значения - когда временное ограничение имеет критическое значение, и продление времени противоречит целям контента;

- Кроме ограничений в 20 и более часов - если временное ограничение превышает 20 часов.

13. Пауза, остановка, скрытие - Для движущихся, мерцающих, прокручивающихся, автоматически обновляющихся элементов верно все нижеследующее:

- Движение, мерцание и прокрутка — для любого движения, мерцания и прокрутки информации, которые (1) начинаются автоматически, (2) длятся более 5 секунд, (3) присутствуют параллельно с другим контентом, пользователю должен быть предоставлен механизм, позволяющий поставить на паузу, остановить или скрыть движение/мерцание/прокрутку элементов, за исключением случаев, где эти действия имеют ключевое значение;

- Автоматическое обновление — для любой автоматически обновляемой информации, которая (1) начинает обновление автоматически и (2) присутствует наряду с другим контентом, пользователю должен быть предоставлен механизм, позволяющий поставить на паузу, остановить, скрыть или изменить частоту обновления. Исключение составляют случаи, когда автоматическое обновление имеет ключевое значение.

14. Три вспышки или ниже порогового значения - Если речь идет о мигании, то содержимое не должно мигать чаще трех раз в секунду, или количество вспышек должно быть менее чем требуемое значение как в общем, так и в частности для красных вспышек.

15. Пропуск блоков - Пользователям предоставляется возможность пропускать повторяющиеся блоки контента.

16. Заголовок страницы - Заголовок страницы должен четко отражать тему или цель сайта.

33. Порядок перемещения фокуса - Все компоненты должны быть упорядочены в логической последовательности для перемещения фокуса.

34. Цель ссылки (в контексте) - Назначение каждой ссылки должно быть понятно из ее текста или контекста.

31. Язык страницы - Для каждой страницы следует указать язык, причем по умолчанию для каждой веб-страницы предполагается человеческий язык.

32. В фокусе - Элементы, находящиеся в фокусе, не должны меняться.

34. При вводе - При получении входных данных элементы не должны изменяться.

34. Выявление ошибок - Идентифицируйте и обозначьте ошибки ввода для пользователей.

31. Пометки или инструкции - Используйте инструкции, когда требуется, чтобы пользователь вводил информацию.

32. Синтаксис - Обеспечьте правильный синтаксис: закрывающиеся теги, правильное логическое расположение элементов, отсутствие повторяющихся атрибутов и уникальные идентификаторы.

33. Название, роль, значение - Название, роль и значение пользовательских компонентов должны быть доступны для понимания через технологии.

Далее мы будем рассматривать предоставление доступности для React- приложения.

Доступность контента в веб-приложениях с использованием WAI-ARIA

Свод правил по доступности контента в веб-приложениях с применением WAI-ARIA (https://www.w3.org/WAI/standards-guidelines/aria/) — это документ, который описывает методы обеспечения доступности контента при разработке JavaScript-программ и компонентов.

В JSX полностью поддерживаются все HTML-атрибуты aria-*. Несмотря на то, что большинство DOM-свойств и атрибутов в React обычно записываются в стиле camelCase, атрибуты aria-* должны быть написаны через дефис. Этот стиль также известен как kebab-case или lisp-case.

Листинг кода 33.1

```
<input
  type="email"
  aria-label={“Электронная почта”}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="email"
/>
```

Семантическая вёрстка

Семантическая разметка — это основа доступности контента на веб-приложениях. Чтобы повысить удобство использования и понимания ваших сайтов, важно использовать различные HTML-элементы, которые делают контент доступным без особых усилий.

Полную информацию о HTML-элементах можно найти на MDN по следующей ссылке: https://developer.mozilla.org/ru/docs/Web/HTML/Element.

Иногда нарушается семантическая разметка, например, когда в JSX добавляется элемент для обеспечения функциональности кода на React. Это особенно часто происходит при работе с элементами списков: <ol>, <ul>, <dl>, или таблицами: <table>. В таких случаях рекомендуется использовать фрагменты.

Листинг кода 33.2

```
import React, { Fragment } from 'react';
const termsArray = [
    { id: 1, term: 'JavaScript', description: 'JavaScript is a programming language that is commonly used to create interactive effects within web browsers.' },
    { id: 2, term: 'HTML', description: 'HTML is a markup language used for creating the structure of a webpage.' },
    { id: 3, term: 'CSS', description: 'CSS is a styling language used to define the presentation of a webpage.' },
    { id: 4, term: 'API', description: 'An API (Application Programming Interface) is a set of rules and protocols for building and interacting with software applications.' },
];

const ListItem = ({ item }) => {
    return (
        <Fragment>
            <dt>{item.term}</dt>
            <dd>{item.description}</dd>
        </Fragment>
    );
}

export const Glossary = () => {
    return (
        <dl>
            {termsArray.map(item => (
                <ListItem item={item} key={item.id} />
            ))}
        </dl>
    );
}
```

Вы также можете преобразовать коллекцию элементов в массив фрагментов или в другие элементы. Если не требуется использование свойств, можно воспользоваться укороченной записью фрагментов (<></> вместо <Fragment></Fragment>).

Доступность контента в формах

Подписи

Каждый элемент управления должен иметь подпись, обеспечивающую доступность контента. Подписи должны быть спроектированы таким образом, чтобы они могли быть использованы экранными считывающими устройствами и другими техническими средствами реабилитации.

Для более подробного изучения этого вопроса можно ознакомиться по следующим ссылкам:

- Демонстрация подписывания элементов от консорциума W3C (https://www.w3.org/WAI/tutorials/forms/labels/).

- Демонстрация подписывания элементов от WebAIM (https://webaim.org/techniques/forms/controls).

- Разъяснения от Paciello Group по доступности наименований (https://www.tpgi.com/what-is-an-accessible-name/).

Эти стандартные подходы HTML могут применяться непосредственно в React. ВНИМАНИЕ!! Атрибут for в JSX следует записывать как htmlFor.

Листинг кода 33.3

```
<label htmlFor="namedInput">Имя:</label>
<input id="namedInput" type="text" name="name"/>
```

Сообщения об ошибках

Причины ошибок должны быть понятны для всех пользователей. Ниже приведены ресурсы, которые показывают, как корректно воспроизводить текст сообщений об ошибках с помощью экранных считывающих устройств:

- W3C демонстрирует примеры уведомлений пользователей (https://www.w3.org/WAI/tutorials/forms/notifications/).

- WebAIM описывает процесс валидации форм (https://webaim.org/techniques/formvalidation/).

Фокус клавиатуры

Фокус клавиатуры указывает на элемент в структуре DOM, готовый принять ввод с клавиатуры, обычно выделен контуром.

Механизмы быстрого доступа к контенту:

1. Использование якорей страницы и CSS. Дополнительную информацию можно найти по ссылке: https://webaim.org/techniques/skipnav/.

2. Использование элементов с семантической структурой. Подробности доступны по ссылке: https://www.scottohara.me/blog/2018/03/03/landmarks.html.

Управление фокусом программно:

1. Применение правил навигации с клавиатуры в JavaScript. Подробности можно узнать здесь: https://developer.mozilla.org/ru/docs/Web/Accessibility/Keyboard-navigable_JavaScript_widgets.

2. Использование ссылок на DOM-элементы (refs). Подробнее об этом можно узнать здесь: https://ru.legacy.reactjs.org/docs/refs-and-the-dom.html.

Работа с событиями мыши:

Весь функционал, связанный с событиями мыши, должен быть доступен для работы только с клавиатурой.

Например, при открытии и закрытии выпадающего списка мы обычно используем событие OnClick, но при работе с клавиатурой событие click не активируется. Для исправления этой ситуации, например, можно обрабатывать события с помощью onBlur или onFocus.

Листинг кода 33.4

```
import logo from './logo.svg';
import './App.css';
import react, { useRef, useState } from 'react';

const App = () => {
  const [isOpen, setIsOpen] = useState(false);
  const timeOutId = useRef(null);

  const onClickHandler = () => {
    setIsOpen(!isOpen);
  };

  const onBlurHandler = () => {
    timeOutId.current = setTimeout(() => {
      setIsOpen(false);
    });
  };

  const onFocusHandler = () => {
    clearTimeout(timeOutId.current);
  };

  return (
    <div onBlur={onBlurHandler} onFocus={onFocusHandler}>
      <button   onClick={onClickHandler}   aria-haspopup="true"
aria-expanded={isOpen}>
        Select an option
      </button>
      {isOpen && (
        <ul>
          <li>Option 1</li>
          <li>Option 2</li>
          <li>Option 3</li>
        </ul>
      )}
    </div>
```

```
  );
};

export default App;
```

Необходимо обратить внимание:

1. Определение языка (указывайте язык текста на странице).

2. Заголовки страниц (должен показывать правильную и точную информацию о странице).

3.Цветовая контрастность (правильный расчет контрастности для людей с плохим зрением).

Инструменты для разработки и тестирования

Перечисленные далее инструменты и средства разработки помогут вам правильно разработать доступный контент для веб-приложения:

1.Тестирование клавиатурой (самый простой и важный вид проверки). Перечень действий:

- Отключить мышь.
- Пользоваться Tab и Shift+Tab для перемещения по странице.
- Пользоваться Enter для активации элементов.
- Там, где необходимо, использовать клавиши со стрелками, например, для работы с меню или выпадающими списками.

2. Инструменты разработчика. Выполнений некоторых требований в JSX можно обеспечить посредствам:

- В средствах разработки обычно есть автоматические подстановки, которые могут использоваться при написании JSX для выбора ролей, состояний и свойств ARIA.
- Плагин eslint-plugin-jsx-a11y. Необходим для поиска проблем, связанных с доступностью контента.

3. Тестирование доступности контента в браузере - aXe, aXe-core и react-axe. С данными средствами необходимо ознакомится самим.

4. Использовать экранный считыватель.

Задание для выполнения:

1. Ознакомитесь с методическими указаниями;

2. Изучить все уровни доступности (A, AA, AAA) и написать чек лист по ним (краткий – несколько предложений).

3. Реализовать в вашем проекте курсовом проекте доступность контента.

4. Протестировать ваше приложение на доступность контента и улучшить доступность если необходимо.

5. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое доступность контента?

2. Какие бывают уровни доступности?

3. Как обеспечить доступность на каждом уровне?

4. Какие вам известны средства для тестирования доступности приложения?

34. ЛАБОРАТОРНАЯ РАБОТА №34. Развертывание веб-приложений

Цель работы: Сборка веб-приложений, разделение приложения на части, webpack-bundle-analyzer итд

Формируемые компетенции: ОК 09, ПК 3.4

Теоретический материал

Бандлинг

Сборка (бандлинг) - это процесс объединения импортированных файлов в один собранный файл (бандл). Он используется для упрощения загрузки всего приложения на веб-странице одним запросом. В результате улучшается производительность и уменьшается количество запросов к серверу.

Пример приведен ниже.

Приложение

Листинг кода 34.1

```
// math.js
export function add(a, b) {
    return a + b;
}
```

Бандл.

Листинг кода 34.2

```
function add(a, b) {
    return a + b;
}
console.log(add(16, 34)); // 42
```

Разделение кода

По мере роста приложения бандл тоже будет расти, особенно при подключении крупных сторонних библиотек. Поэтому необходимо следить за кодом, чтобы случайно не сделать приложение настолько большим, что его загрузка займёт слишком много времени. Чтобы предотвратить разрастание бандла, стоит начать его разделять.

Разделение кода – это возможность, поддерживаемая такими бандлерами как Webpack, Rollup или Browserify, которая может создавать несколько бандлов и загружать их по мере

необходимости. Это поможет избежать загрузки ненужного кода и уменьшить объём кода, необходимого для начальной загрузки.

Import()

Один из способов внедрения разделения кода в приложение – использование синтаксиса динамического импорта: import().

До.

Листинг кода 34.3

```
import { add } from './math';
console.log(add(16, 34));
```

После.

Листинг кода 34.4

```
import("./math").then(math => {
    console.log(math.add(16, 34));
});
```

Когда Webpack сталкивается с таким синтаксисом, он автоматически разделяет код приложения.

React.lazy

Функция React.lazy позволяет рендерить динамический импорт как обычный компонент.

До.

Листинг кода 34.5

```
import OtherComponent from './OtherComponent';
```

После.

Листинг кода 34.6

```
const OtherComponent = React.lazy(()=>import('./OtherComponent'));
```

Этот функционал загружает бандл автоматически, содержащий othercomponent, при первом рендеринге этого компонента. React.lazy принимает функцию, которая должна вызывать

динамический import(). Результатом этого promise является модуль, который экспортирует по умолчанию react-компонент. Компонент с ленивой загрузкой должен быть отрендерен внутри компонента suspense, который позволяет нам показать запасное содержимое (например, индикатор загрузки) во время загрузки ленивого компонента.

Листинг кода 34.7

```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(()=>import('./OtherComponent'));

function MyComponent() {
    return (
        <div>
            <Suspense fallback={<div>Загрузка...</div>}>
                <OtherComponent />
            </Suspense>
        </div>
    );
}
```

Fallback принимает любой элемент React, который вы хотите показать во время загрузки компонента. Suspense компонент может быть размещен в любом месте перед ленивым компонентом. Кроме того, вы можете обернуть несколько ленивых компонентов одним Suspense компонентом.

Листинг кода 34.8

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
    return (
        <div>
            <Suspense fallback={<div>Loading...</div>}>
                <section>
                    <OtherComponent />
                    <AnotherComponent />
                </section>
            </Suspense>
        </div>
    );
}
```

Любой компонент может приостановиться в результате рендеринга, даже компоненты, которые уже были показаны пользователю. Чтобы содержимое экрана всегда было согласованным, если уже показанный компонент приостанавливается, React должен скрыть свое дерево до ближайшей границы <Suspense>. Однако, с точки зрения пользователя, это может дезориентировать.

Листинг кода 34.9

```
import React from 'react', { Suspense };
Import tabs from "./Tabs";
import mica from './mica';

const Comments = React.lazy(() => im-port('./Comments'));
const Photos = React.lazy(() => import('./Photos'));

function MyComponent() {
    const [tab, setTab] = React.useState('photo');

    function handleTabSelect(tab) {
        setTab(tab);
    };

    return (
        <div>
            <tab onTabSelect={handleTabSelect} />
            <Suspense backup={<Glimmer/>}>
                {tab === 'Photos' ? <Photos/>: <Comments/>}
            </suspense>
        </div>
    );
}
```

В этом примере, если вкладка будет изменена с "photos" на "comments", но комментарии будут приостановлены, пользователь увидит мерцание. Это имеет смысл, потому что пользователь больше не хочет видеть фотографии, компонент комментариев не готов что-либо отображать, а React должен поддерживать согласованность пользовательского интерфейса.

Однако иногда такой пользовательский интерфейс нежелателен. В частности, иногда лучше показать старый пользовательский интерфейс во время подготовки нового пользовательского интерфейса, используя новый startTransition API.

Листинг кода 34.10

```
function handleTabSelect(tab) {
    startTransition(() => {
        setTab(tab);
    });
}
```

Здесь вы указываете React установить вкладку комментариев не для немедленного обновления, а для перехода, что может занять некоторое время. Затем React сохранит старый пользовательский интерфейс и будет взаимодействовать с ним, переключившись на рендеринг <Comments/>, когда он будет готов.

Если модуль не может быть загружен (например, из-за сбоя сети), он выдаст ошибку. Вы можете обработать эти ошибки, чтобы улучшить взаимодействие с пользователем с помощью Fuses. После создания автоматического выключателя его можно использовать в любом месте поверх ленивого компонента для отображения состояний ошибки.

Листинг кода 34.11

```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
    <div>
        <MyErrorBoundary>
            <Suspense fallback={<div>Загрузка...</div>}>
                <section>
                    <OtherComponent />
                    <AnotherComponent />
                </section>
            </Suspense>
        </MyErrorBoundary>
    </div>
);
```

Разделение кода на основе маршрута

Решение о том, где ввести разделение кода в вашем приложении, может оказаться трудным. В идеале вам следует выбрать место, где вы разделите свой код на пакеты примерно одинакового размера, чтобы обеспечить удобство работы с пользователем.

Оказывается, маршрут зачастую оказывается таким удобным местом. Второй

Большинство интернет-пользователей привыкли к задержкам при переходе между страницами. Поэтому вам может быть полезно повторно отрисовать всю страницу. Это предотвратит взаимодействие пользователя с другими элементами на странице во время обновления.

Вот пример того, как реализовать разделение кода на основе маршрутов с использованием таких библиотек, как React.lazy и React Router.

Листинг кода 34.12

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
    <Router>
        <Suspense fallback={<div>Loading...</div>}>
            <Routes>
                <Route path="/" element={<Home />} />
                <Route path="/about" element={<About />} />
            </Routes>
        </Suspense>
    </Router>
);
```

Реализация

Webpack Bundle Analyzer – отличный инструмент для выявления проблемных мест бандлов, он помогает визуализировать выходные данные webpack в виде интерактивной карты. Он отображает размер, который используется каждым пакетом до и после минимизации и сжатия.

1. Необходимо установить webpack-bundle-analyzer в проект с помощью команды:

Листинг кода 34.13

```
npm install --save-dev webpack-bundle-analyzer
```

2. Далее изменить package.json, добавив в него --stats флаг в конце "build" команды скрипта.

Листинг кода 34.14

```
"scripts": {
    …
    "build": "react-scripts build --stats",
    …
},
```

3. Добавить новый скрипт, вызываемый "analyze".

Листинг кода 34.15

```
    "scripts": {
        …
        "build": "react-scripts build --stats",
        "analyze":    "webpack-bundle-analyzer    build/bundle-
stats.json",
        …
    },
```

4. Необходимо собрать проект с помощью команды build.

Листинг кода 34.16

```
npm run build
```

5. Запустить анализ проекта с помощью команды analyze.

Листинг кода 34.17

```
npm run analyze
```

На рисунках 34.1 -34.4 представлен результат выполнения команды.

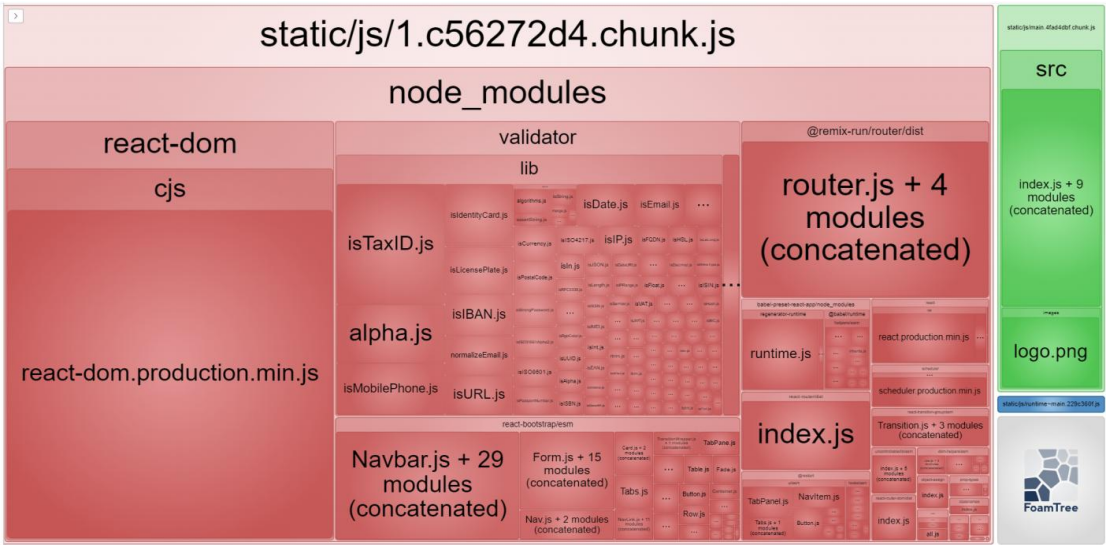

Рисунок 34.1

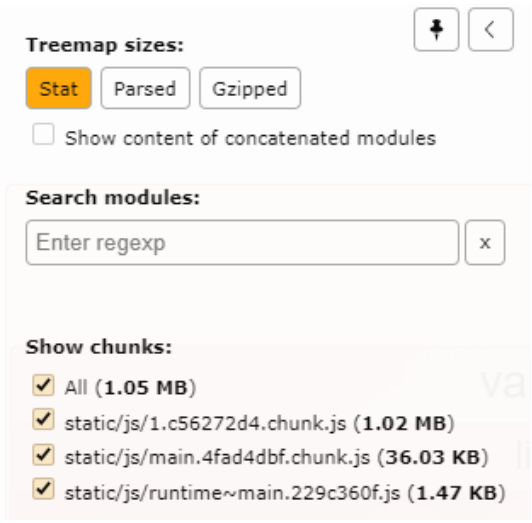

Рисунок 34.2

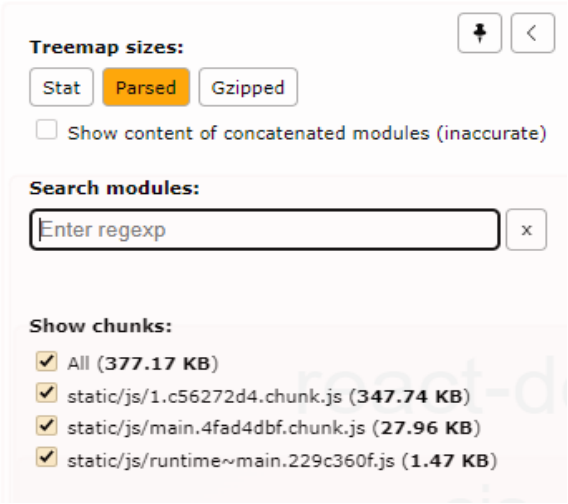

Рисунок 34.3

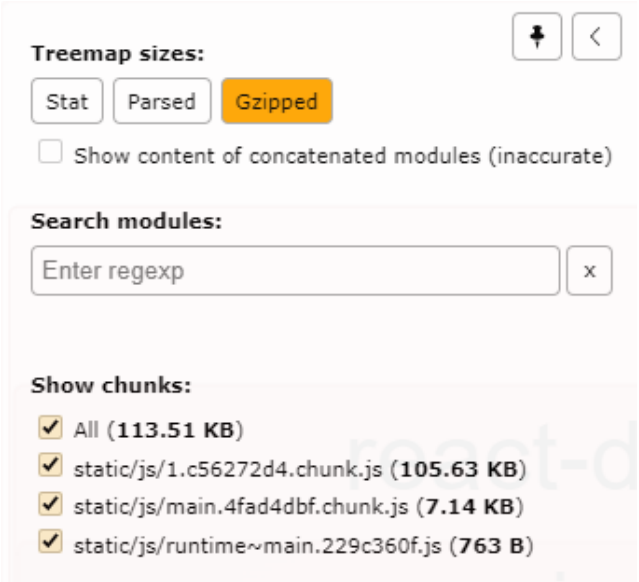

Рисунок 34.4

Задание для выполнения:

1. Ознакомитесь с методическими указаниями;

2. С помощью инструмента Webpack Bundle Analyzer провести анализ курсового проекта или любого другого.

3. Провести анализ сборки проекта.

4. Изменить проект, используя способы внедрения разделения кода.

5. Провести анализ проекта, изменений в сборке проекта и сравнить результаты.

6. Составить отчет по результатам работы

Вопросы по лабораторной работе:

1. Что такое “Разделение кода”?

2. Какие есть способы внедрения разделения кода в приложение React?

3. Каким образом формируется сборка проекта?

4. Что такое “Webpack Bundle Analyzer”? Что демонстрирует данный инструмент при анализе проекта?

5. Как влияет разделение кода на загрузку данных проекта?

ЗАКЛЮЧЕНИЕ

Были рассмотрены темы:

- Настройка окружения JavaScript(Установка Google Chrome, NodeJS, Visual Studio Code. Добавление плагинов в Visual Studio Code для работы с JavaScript. Установка утилиты browser-sync. Проверка работы browser-sync)
- Введение в JavaScript (Добавление динамики в работу HTML страницы, изменение фона при наведении указателя мыши, совершение простых арифметические действий с использованием JavaScript)
- Программирование в JavaScript(Решение простейших задач на программирование (условия, циклы, функции) с использование языка JavaScript)
- Работа со строками в языке JavaScript (Вычисление длины строки Юникода, поиск подстроки в строке с использованием регулярных выражений, валидация введенных пользователем данных)
- Объекты и массивы в JavaScript (Решение задач с использованием встроенных объектов JavaScript (Map, Set, Date), с помощью методов для обработки списков (.map, .filter, .reduce), создание сложных объектов (дескрипторы свойств, Proxy, Reflect))
- Функции в JavaScript (Решение задач на функции и рекурсию, создание набора функций, замкнутых на общую переменную)
- Объекты и прототипы в JavaScript (Решение задач с помощью объектов, создание конструктора объектов с набором общих методов)
- ООП в JavaScript (Решение задач на ООП, написание кода с использованием классов)
- Обработка исключений в JavaScript (Решение задач на обработку исключений)
- Обработка действий пользователя в браузере (Создание веб-страницы, обрабатывающей движение мыши и нажатие кнопок на клавиатуре)
- Асинхронное программирование в JavaScript (Решение задач на асинхронное программирование, промисификация функций, написание циклов и условий с использованием функций с callback-ами и Promise-ов)
- JavaScript анимация (Создание анимации слайдера с помощью JavaScript)
- Генераторы JavaScript (Решение задач с использованием генераторов, создание генератора псевдослучайных чисел)
- Асинхронные итераторы JavaScript (Решение задач с использованием асинхронных генераторов, получение общего значения при частичном вводе данных)
- Разделение JavaScript кода на модули (Создание отдельного модуля с полезными функциями для дальнейшего использования)
- Работа с DOM (Создание простой браузерной мини игры)
- Работа с BOM (Изменение URL текущей страницы, сохранение данных о текущем пути в localStorage)
- AJAX (Работа с удаленным сервером средствами JavaScript, выполнение запросов, загрузка и выгрузка файлов)
- Webpack и npm (Создание JavaScript проекта, установка сторонних модулей, сборка JavaScript модулей в один пакет, публикация собственного модуля в npm репозитории)
- Язык TypeScript (Написание простейших программ (циклы, условия, функции) с использованием языка TypeScript)
- Транспайлер кода babel (изучение и работа с транспайлерос babel).
- Введение в React.js (изучение базовых понятий в ReactJS: компоненты, циклы, условия и передача параметров).
- Хуки в ReactJS (изучение и применение хуков useState и useEffect в реальном коде).
- Контекст в ReactJS (изучение и применение хука useContext в реальном коде).

- Списки в ReactJS (изучение асинхронных запросов к серверу, вывод данных полученных из сервера на страницу, добавление пользовательских данных в базу данных, создание пагинации).

- Маршрутизация в ReactJS (настройка маршрутизации в проекте).

- Redux (изучение и настройка Redux 2 способами: legacy redux и redux-toolkit).

- Redux-thunk (построение запросов к серверу с помощью библиотек redux и redux-thunk).

- Redux-saga (изучение Redux-saga и применение знаний на практике).

- Тестирование React-приложений (изучение библиотеки Jest и применение знаний на практике).

- Анимация (создаиние анимации).

- Общедоступные приложения (изучение стандартов по созданию общедоступности приложений, тестирование общедоступности приложение и применение на практике).

- Развертывание веб-приложений (изучение библиотеки webpack-bundle- analyzer, анализ полученного результата и оптимизация веб-приложения).

Были получены навыки работы с языком JavaScript и с фронтом, созданным с помощью ReactJS, интеграции серверной и клиентской части, создания общедоступного приложения и т.д..

СПИСОК ЛИТЕРАТУРЫ

1. Кингсли-Хью, Э. JavaScript в примерах / Кингсли-Хью Э. , Кингсли-Хью К. - М : ДМК Пресс. URL : https://www.studentlibrary.ru/book/ISBN9785940746683.html (дата обращения: 10.03.2025).
2. Основы JavaScript / - М : Национальный Открытый Университет "ИНТУИТ". URL : https://www.studentlibrary.ru/book/intuit_175.html (дата обращения: 10.03.2025).
3. Хорстман, К. С. Современный JavaScript для нетерпеливых / К. С. Хорстман – М : ДМК Пресс. URL : https://www.studentlibrary.ru/book/ISBN9785970601778.html (дата обращения: 10.03.2025).
4. Баланов, А. Н. Комплексное руководство по разработке: от мобильных приложений до веб-технологий : учебное пособие для вузов / А. Н. Баланов. — СПб: Лань. URL: https://e.lanbook.com/book/394577 (дата обращения: 10.03.2025).
5. Янцев, В. В. Разработка web-страниц на HTML, CSS и JavaScript : учебное пособие для вузов / В. В. Янцев. — СПб : Лань. URL: https://e.lanbook.com/book/422462 (дата обращения: 10.03.2025).
6. Баланов, А. Н. Прототипирование и разработка пользовательского интерфейса: оптимизация UX : учебное пособие для вузов / А. Н. Баланов. — СПб : Лань. URL: https://e.lanbook.com/book/414929 (дата обращения: 10.03.2025).
7. Ковешникова, Н. А. История дизайна. Краткий курс лекций : учебное пособие для вузов / Н. А. Ковешникова. — 3-е изд., стер. — СПб: Лань. URL: https://e.lanbook.com/book/394688 (дата обращения: 10.03.2025).