Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Институт повышения квалификации
и переподготовки

Кафедра «Информатика»

ВЕБ-ТЕХНОЛОГИИ

ПОСОБИЕ
по одноименной дисциплине для слушателей
специальности переподготовки
9-09-0612-02 «Программное обеспечение
информационных систем»
заочной формы обучения

Гомель 2026

УДК 004.42(075.8)
ББК 32.973.22я73
В26

Рекомендовано советом
института повышения квалификации и переподготовки ГГТУ им. П. О. Сухого
(*протокол № 4 от 29.12.2025 г.*)

Составитель *Н. В. Самовендюк*

Рецензент: зав. каф. «Информационные технологии» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *К. С. Курочка*

В26 **Веб-**технологии : пособие по одноим. дисциплине для слушателей специальности 9-09-0612-02 «Программное обеспечение информационных систем» заоч. формы обучения / сост. Н. В. Самовендюк. – Гомель : ГГТУ им. П. О. Сухого, 2026. – 104 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 2 Gb RAM ; свободное место на HDD 16 Mb ; ATL Lunux 10.1 ; Adobe Acrobat Reader. – URL: http://elib.gstu.by. – Загл. с титул. экрана.

Рассмотрены веб-технологии, используемые при написании серверной и клиентской частей веб-приложений. В качестве языка сценария выбран JavaScript, платформой для реализации серверной части Node.js. В пособии дается краткий курс базового JavaScript, описывается использование JavaScript в веб-браузере и на стороне сервера. Дополнительно представлена информация по библиотеке JQuery и фреймворку Express, использование которых значительно облегчает и ускоряет разработку веб-приложений.

Для слушателей специальности 9-09-0612-02 «Программное обеспечение информационных систем» ИПКиП.

УДК 004.42(075.8)
ББК 32.973.22я73

СОДЕРЖАНИЕ

Введение

Дисциплина «Веб-технологии» является одной из базовых при подготовке слушателей специальности переподготовки 9-09-0612-02 «Программное обеспечение информационных систем».

Целью данной дисциплины является формирование у слушателей знаний и умений в области проектирования архитектуры и разработки веб-приложений с использованием современных веб-технологий.

Веб-технологии – это совокупность технологий, инструментов и стандартов, используемых для разработки и поддержки веб-приложений и сайтов. Они включают в себя языки программирования, библиотеки, фреймворки, базы данных, протоколы и другие инструменты.

Все веб-приложения разрабатываются на основе архитектуры «клиент – сервер», соответственно и веб-технологии разделяются на две группы – технологии стороны клиента (client-side) и технологии стороны сервера (server-side).

С точки зрения программирования мы также разделяем веб-технологии на клиентские (front-end) и серверные (back-end).

С клиентскими технологиями HTML и CSS слушатели ознакомились при изучении дисциплины «Основы веб-программирования».

В данном пособии будет рассмотрен язык программирования JavaScript и его использование как на стороне клиента (веб-браузер), так и на стороне сервера (веб-сервер).

1 Базовый JavaScript

1.1 Введение в JavaScript

JavaScript был создан в 1995 году в компании Netscape в качестве языка сценариев в браузере Netscape Navigator.

Первоначально JavaScript обладал довольно небольшими возможностями. Его цель состояла лишь в том, чтобы добавить немного функциональности и интерактивности на веб-страницу. Например, обработать нажатие кнопок на веб-странице, произвести какие-нибудь другие действия, связанные прежде всего с элементами управления.

Однако развитие веб-технологий, появление HTML5 и технологии Node.js открыло перед JavaScript гораздо большие горизонты. На данный момент JavaScript используется для создания веб-приложений как на стороне клиента, так и на стороне сервера. Кроме того JavaScrip используется в мобильной разработке, а также для программирования «умных» устройств (smart devices) в технологии IoT (Интернет вещей).

С самого начала развития динамического HTML существовало несколько веб-браузеров (Netscape, Internet Explorer и др.), которые предоставляли различные реализации языка. Компания Netscape представила язык JavaScript для стандартизации Европейской Ассоциации производителей компьютеров (European Computer Manufacturer’s Association — ECMA), в результате чего появился стандарт языка ECMAScript. Однако многие по-прежнему называют язык просто JavaScript. С момента стандартизации были разработаны версии языка ES1, ES2, ES3, ES5, ES6. Начиная с 2015 года версии стандарта стали именоваться по году модификации. На данный момент времени последняя версия ES2025.

JavaScript является интерпретируемым языком. Это значит, что код на языке JavaScript выполняется с помощью интерпретатора. Интерпретатор получает инструкции языка JavaScript, которые определены на веб-странице и выполняет их. Последовательность инструкций на JavaScript часто называется сценарием или скриптом.

Для написания сценариев на языке JavaScript достаточно любого текстового редактора, но лучше использовать специализированные, такие как Sublime Text, Notepad++, или интегрированные среды разработки Visual Studio Code или WebStorm.

1.2 Основные приемы работы при написании сценариев

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега **<script>**. Для вставки сценария или библиотеки из внешнего файла используется атрибут src, в котором указывается либо абсолютный путь до скрипта от корня сайта, либо полный URL-адрес. Например:

```
<script src="https://code.jquery.com/jquery-3.4.1.min.js"></script>
```

Как только анализатор HTML встречает элемент <script>, он запускает интерпретатор языка для выполнения инструкций и приостанавливает анализ HTML-разметки и визуализацию документа. Поэтому рекомендуется размещать сценарий в конце документа перед закрывающимся тегом </body>.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    console.log("Web-technologies!!!");
  </script>
</body>
</html>
```

Такой стандартный синхронный или блокирующий режим выполнения сценария можно изменить. Тег <script> может иметь

атрибуты **defer** и **async**. Атрибут defer заставляет браузер отложить выполнение сценария до тех пор пока документ полностью не загрузится. Атрибут async заставляет браузер запустить сценарий как можно раньше, но не блокировать анализ документа во время загрузки сценария.

Сценарий JavaScript выполняется с использованием инструкций, которые представляют собой синтаксические конструкции языка и команды. Обычно каждую инструкцию пишут на новой строке, чтобы код было легче читать, и заканчивают точкой с запятой (последнее не обязательно, но желательно).

```
alert('Hello');
console.log('World');
```

Для пояснения программного кода используются комментарии. В JavaScript комментарии могут находиться в любом месте скрипта и могут быть однострочными и многострочными.

```
// Этот комментарий занимает всю строку
/*
Это - многострочный комментарий.
*/
```

При написании программного кода могут быть допущены ошибки. По умолчанию в браузере ошибки не видны. Для устранения такого недостатка в современных браузерах встроены так называемые «Инструменты разработки» (Developer tools или сокращённо — devtools).

Это очень мощные инструменты, которыми пользуются разработчики для анализа кода, отладки, просмотра загрузки сети и др.

На начальном этапе нам понадобятся только вкладки «Console» и «Debug». На вкдадке «Console» мы можем отслеживать ошибки, а также запускать операторы языка.

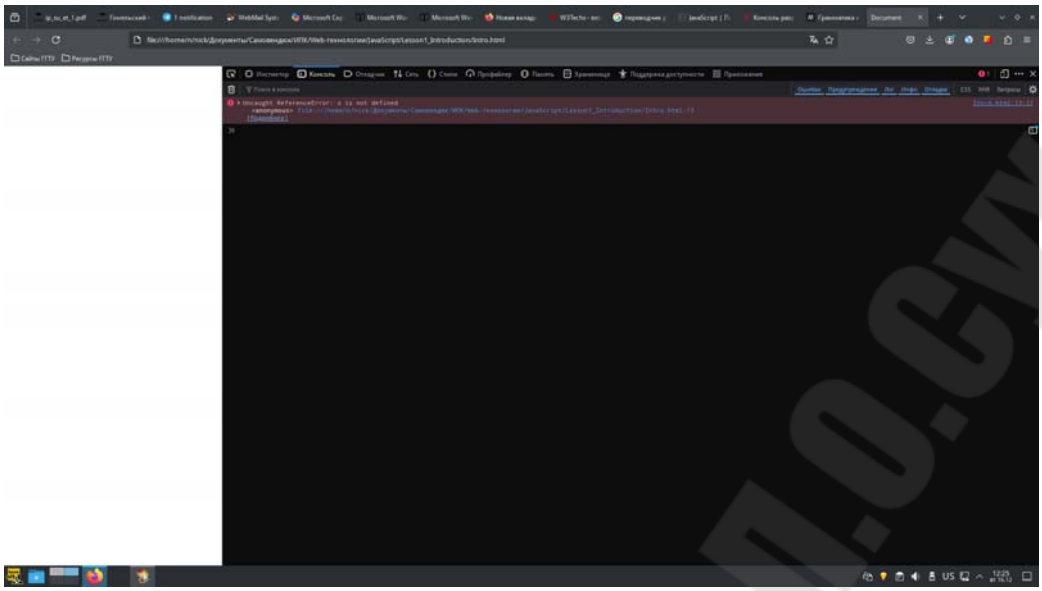


Рисунок 1.1 — Просмотр ошибки в консоли

На вкладке «Debug» мы можем просмотреть в каком месте произошла ошибка. Кроме этого мы можем запустить сценарий в режиме отладки и пошагово выполнять инструкции, а также просматривать содержимое стека и значения локальных переменных.

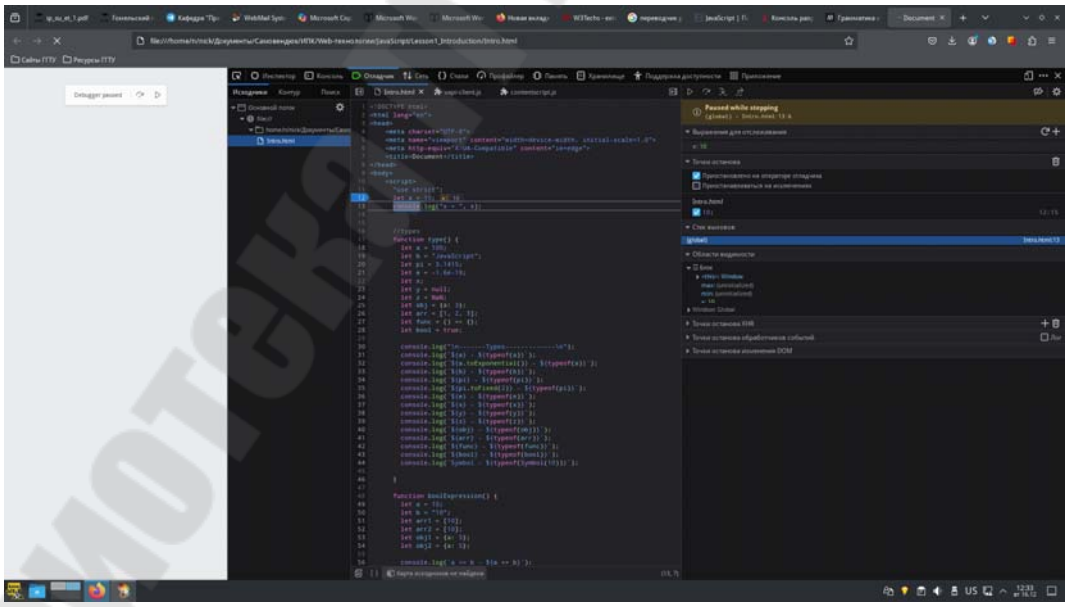


Рисунок 1.2 — Отладка сценария

При написании сценария (особенно для новичков) рекомендуется первой инструкцией записывать директиву

```
"use strict";
```

которая переводит браузер в строгий режим и сообщает интерпретатору, что весь сценарий работает в соответствии со стандартом, который в свою очередь прерывает выполнения сценария при обнаружении ошибок.

1.3 Типы, значения и переменные

Любое веб-приложение обычно обрабатывает какую-либо информацию. Для хранения информации в JavaScript используются переменные и константы.

Для создания переменной в JavaScript используется ключевое слово **let**:

```
let message;
message = 'Hello';
```

На практике для краткости часто совмещают объявление переменной и запись данных в одну строку.

В старых скриптах вы также можете найти другое ключевое слово: var вместо let. Ключевое слово var – почти то же самое, что и let. Оно объявляет переменную, но немного по-другому, «устаревшим» способом. Рекомендации консорциума W3C использовать только let.

Кроме того если работать в строгом режиме и не использовать ключевое слово произойдет ошибка и выполнение сценария будет приостановлено.

Для объявления константной, то есть, неизменяемой переменной, используется ключевое слово **const**:

```
const PI = 3.1415;
```

При названии переменных надо учитывать, что JavaScript является регистрозависимым языком, то есть в следующем коде объявлены две разные переменные:

```
let myVar;
```

```
let MyVar;
```

Значения, которые заносятся в переменную с использованием оператора присваивания (=), могут быть одного из 8 основных типов:

– **number** для любых чисел: целочисленных или чисел с плавающей точкой, целочисленные значения ограничены диапазоном $\pm 2^{53}$.
– **bigint** для целых чисел произвольной длины.
– **string** для строк. Строка может содержать один или больше символов, нет отдельного символьного типа.
– **boolean** для true/false.
– **null** для неизвестных значений – отдельный тип, имеющий одно значение null.
– **undefined** для неприсвоенных значений – отдельный тип, имеющий одно значение undefined.
– **object** для более сложных структур данных.
– **symbol** для уникальных идентификаторов.

Оператор **typeof** позволяет нам увидеть, какой тип данных сохранён в переменной:

```
typeof(Symbol(10))
```

У числового типа данных есть три специальных значения Infinity (бесконечность), -Infinity и NaN (значение не является числом).

```
100 – number
1e+2 – number
"JavaScript" – string
3.1415 – number
-1.6e-19 – number
undefined – undefined
null – object
NaN – number
{a: 3} – object
[1, 2, 3] – object
() => {} – function
```

true – boolean
Symbol(10) – symbol

Часто значения, которые заносятся в переменную, являются результатом вычисления выражения. Выражение представляет из себя набор операндов и операторов, выполняющих действия над операндами.

JavaScript является наследником языка C. Поэтому набор операторов и управляющих конструкций в языке точно такой же как и в C, с которым вы уже знакомы.

Следует, однако, отметить, что язык JavaScript является интерпретируемым и слаботипизированным. Тип переменной определяется по введённому значению или результату вычисления выражения, поэтому в процессе выполнения сценария тип переменной может меняться.

Все арифметические операции выполняются над числами, поэтому при вычислении арифметических выражений все операнды преобразовываются к числовому типу. Аналогичное преобразование происходит при сравнении операндов различных типов. Поэтому в JavaScript были введены два оператора строгого равенства "идентично" ("===") и не равенства ("!=="). Данные операторы сравнивают не только значения операндов, но и их типы. Следует внимательно следить за преобразованием типов.

Для преобразования строк в числа в JavaScript предусмотрены встроенные функции **parseInt()** и **parseFloat()**. Функция parseInt(строка, основание) преобразует указанную в параметре строку в целое число в системе счисления по указанному основанию (8, 10 или 16). Если основание не указано, то предполагается 10, то есть десятеричная система счисления. Функция parseFloat(строка) преобразует указанную строку в число с плавающей разделительной (десятичной, основание) точкой:

```
parseInt("3.14") // результат= 3
parseInt("Вася") // результат = NaN, то есть не является числом
parseInt("0xFF", 16) // результат = 255
parseFloat("3.l4") // результат= 3.14
parseFloat("-7.875") // результат = -7.875
```

Для преобразования числа в строку используется операция конкатенации:

```
let str = "" + "3.14";
```

1.4. Ввод и вывод данных

В JavaScript предусмотрены довольно скудные средства для ввода и вывода данных. Это вполне оправдано, поскольку JavaScript создавался в первую очередь как язык сценариев для веб-страниц. Можно воспользоваться тремя стандартными методами глобального объекта window для ввода и вывода данных: **alert()**, **prompt()**, **confirm()**.

Метод alert() позволяет выводить диалоговое окно с заданным сообщением и кнопкой ОК:

```
alert(“сообщение”);
```

Сообщение представляет собой данные любого типа: последовательность символов, заключённую в кавычки, число (в кавычках или без них), переменную или выражение.

Метод confirm() позволяет вывести диалоговое окно с сообщением и двумя кнопками – ОК и Отмена (Cancel). В отличие от метода alert этот метод возвращает логическую величину, значение которой зависит от того, на какой из двух кнопок щелкнул пользователь. Если он щелкнул на кнопке ОК то возвращается значение true (истина), если же он щелкнул на кнопке Отмена, то возвращается значение false (ложь). Синтаксис применения метода confirm имеет следующий вид:

```
let answer = confirm(“сообщение”);
```

Метод prompt позволяет вывести на экран диалоговое окно с сообщением, а также с текстовым полем, в которое пользователь может ввести данные. Кроме того, в этом окне предусмотрены две кнопки: ОК и Отмена (Cancel). Данный метод принимает два параметра: сообщение и значение, которое должно появиться в текстовом поле ввода данных по умолчанию. Если пользователь щелкнет на кнопке ОК, то метод вернет содержимое поля ввода

данных, если он щелкнет на кнопке Отмена, то возвращается логическое значение false (ложь).

```
let input = prompt(“сообщение”,”начальное_значение”);
```

При разработке веб-приложений стандартные диалоговые окна используются чаще всего для отладки. В реальных проектах ввод данных чаще всего осуществляется с использованием элементов управления веб-форм, а вывод – непосредственно в разметку документа, которая сразу же визуализируется в браузере.

Для вывода информации в консоль браузера используется объект **console** и его методы **log** или **dir**:

```
console.log('World');
```

1.5 Основные конструкции языка JavaScript

1.5.1 Условные конструкции

Условные конструкции позволяют выполнить те или иные действия в зависимости от определенных условий.

Оператор if

Оператор if относится к числу наиболее популярных. Оператор if применяется следующим образом:

```
if (условие) {
        [операторы]
}
```

В качестве условия может указываться любое логическое выражение. Если результат условия равен true, выполняются операторы, продолжается выполнение остального программного кода. Если условие возвращает false, операторы игнорируются.

Конструкция if..else

Иногда одной конструкции if оказывается недостаточно. Часто требуется зарезервировать набор операторов, которые будут выполняться в случае, когда условное выражение возвращает false. Это можно сделать, добавив еще один блок непосредственно после блока if:

```
if (условие) {
        [операторы]
} else {
        [операторы]
}
```

Конструкция switch..case

Конструкция switch..case является альтернативой использованию конструкции if..else if..else и позволяет обработать сразу несколько условий.

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

В конце каждого блока case ставится оператор break, чтобы избежать выполнения других блоков. Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок default:

```
let income = 300;
switch(income){

   case 100 :
      console.log("Доход равен 100");
      break;
   case 200 :
      console.log("Доход равен 200");
      break;
   case 300 :
      console.log("Доход равен 300");
      break;
```

```
    default:
        console.log("Доход неизвестной величины");
        break;
}
```

Тернарная операция

Тернарная операция состоит из трех операндов и имеет следующее определение:

[первый операнд - условие] ? [второй операнд] : [третий операнд].

В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий.

```
console.log((a > b) ? a : b);
```

1.5.2 Циклы

Циклы позволяют в зависимости от определенных условий выполнять некоторое действие множество раз. В JavaScript имеются следующие виды циклов:

– **for;**
– **for..in;**
– **for..of;**
– **while;**
– **do..while.**

Цикл for

Цикл for имеет следующее определение:

```
for ([инициализация счетчика]; [условие]; [изменение счетчика]){
    // действия
}
```

Цикл for..in

Цикл for..in предназначен для перебора массивов и объектов. Его формальное определение:

```
for (индекс in объект) {
   // действия
}
```

Цикл for...of

Цикл for...of похож на цикл for...in и предназначен для перебора значений коллекции, например, массива:

```
for (значение of массив) {
   // действия
}
```

Цикл while

Цикл while выполняется до тех пор, пока некоторое условие истинно. Его определение:

```
while (условие) {
   // действия
}
```

Цикл do..while

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется. Его определение:

```
do {
   // действия
} while(условие)
```

Операторы continue и break

Иногда бывает необходимо выйти из цикла до его завершения. В этом случае мы можем воспользоваться оператором **break**. Если нам надо просто пропустить очередную итерацию, но не выходить из цикла, мы можем применять оператор **continue**.

1.6 Функции в JavaScript

Функции представляют собой набор инструкций, выполняющих определенное действие или вычисляющих определенное значение.

Синтаксис определения функции:

```
function имя_функции([параметр [, ...]]){
    // Инструкции
}
```

Определение функции начинается с ключевого слова function, после которого следует имя функции. Наименование функции подчиняется тем же правилам, что и наименование переменной: оно может содержать только цифры, буквы, символы подчеркивания и доллара ($) и должно начинаться с буквы, символа подчеркивания или доллара.

После имени функции в скобках идет перечисление параметров. Даже если параметров у функции нет, то просто идут пустые скобки. Затем в фигурных скобках идет тело функции, содержащее набор инструкций.

Чтобы выполнить тело функции, необходимо вызвать функцию с передачей параметров или без параметров:

```
function sum(a, b, c){
    let d = a + b + c;
    console.log(d);
}

sum(1, 2, 3);
sum();
```

Функция может возвращать результат. Для этого используется оператор **return**:

```
function sum(a, b){
   return a + b;
}

console.log(sum(1, 2));
```

В отличии от компилируемых языков программирование, где количество фактических параметров, передаваемых при вызове функции, и количество формальных параметров, описанных при объявлении, должны совпадать, JavaScript позволяет не соблюдать это правило. Если для параметров не передаётся значение, то по умолчанию они имеют значение "undefined".

Для определения количества переданных параметров и доступа к их значениям при вызове функции создаётся массив **arguments**, доступный внутри тела функции.

```
function sum() {
   let s = 0;
   for (let i = 0; i < arguments.length; ++i) {
      s += arguments[i];
   }
   return s;
}
```

Есть способ определения значения для параметров по умолчанию:

```
function sum(a = 1, b = 2, c = 3){
   let d = a + b + c;
   console.log(d);
}
```

С помощью **spread**-оператора мы можем указать, что с помощью параметра можно передать переменное количество значений:

```
function display(season, ...temps) {
   console.log(season);
```

```
    for (index in temps){
        console.log(temps[index]);
    }
}

display("Весна", 31, 30, 31);
display("Лето", 30, 31, 31);
```

Функции могут выступать в качестве параметров других функций.

Особенностью JavaScript является возможность определения функции внутри функций и возвращать результат в виде функции. В последнем случае возвращаемая функция замыкает на себе лексическое окружение вызываемой функции для хранения локальных переменных и параметров вызова.

```
let msg = "Hi Mr. ";
function greeting(name) {
    const lastName = prompt("Enter your last name!");
    console.log(`${msg + name} ${lastName} !`);
    function getAge() {
        const age = prompt("Enter your age:");
        console.log(`You are ${age} old!`);
    }
    return getAge;
}

let answer = greeting("Nick");
answer(); // Hi Mr. Nick
//You are 25 old!
answer = greeting("Max");
answer(); //Hi Mr. Max
//You are 23 old!
```

Среди функций отдельно можно выделить рекурсивные функции. Их суть состоит в том, что функция вызывает саму себя.

Рассмотрим, например, функцию, определяющую факториал числа:

```
function getFactorial(n){
    if (n === 1){
        return 1;
    }
    else{
        return n * getFactorial(n - 1);
    }
}

let result = getFactorial(4);
console.log(result); // 24
```

В JavaScript функции очень часто передаются в качестве параметров. Для лаконичной записи в стандарт языка были введены стрелочные функции:

```
const sum3 = (...args) => {
    let s = 0;
    for (let i = 0; i < args.length; ++i) {
        s += args[i];
    }
    return s;
}

console.log(`sum3(1, 2, 3) = ${sum3(1, 2, 3)}`);
```

Отличительной особенностью стрелочных функций является отсутствие привязки исполнения контекста (this) и объекта arguments при вызове функции.

1.7 Работа с массивами в JavaScript

Для работы с наборами данных предназначены массивы. Для создания массива применяется литерал массива или конструкция new Array():

```
let array_name1 = [item1, item2, ...];
let array_name2 = new Array([item1, item2, ...]);
```

Для повышения производительности и читабельности программного кода рекомендуется использовать литерал массива.

Для получения доступа к элементам массива используется индекс. Индексация элементов начинается с нуля:

```
let cars = ["Saab", "Volvo", "BMW"];
console.log(cars[0]); // Saab
```

Индекс используется как для чтения, так и для записи элемента массива. Причём в отличие от других языков, таких как C# или Java, можно установить элемент, который изначально не установлен:

```
let cars = ["Saab", "Volvo", "BMW"];
cars[10] = "Toyota"
console.log(cars[10]); // Toyota
console.log(cars[3]); // undefined
```

Также стоит отметить, что в отличие от ряда языков программирования в JavaScript массивы не являются строго типизированными, один массив может хранить данные разных типов:

```
const objects = ["Tom", 12, true, 3.14, false];
console.log(objects.toString());
```

Массивы могут быть одномерными и многомерными. Каждый элемент в многомерном массиве может представлять собой отдельный массив.

```
const students = [
        ["Иванов", 20, 5.5],
        ["Петров", 18, 8.2],
        ["Сидоров", 21, 7.8]
    ];
students[0][1] = 19; // присваиваем отдельное значение
console.log(students[0][1]); // 56
```

В языке JavaScript все свойства и методы обработки массивов собраны в глобальном объекте **Array.prototype**, от которого автоматически наследуются все создаваемые массивы.

Все массивы обладают свойством **length**, которой устанавливает или возвращает количество элементов в массиве:

```
let cars = ["Saab", "Volvo", "BMW"];
console.log(cars.length); // 3
cars.length = 5;
console.log(cars[4]); // undefined
```

Методы массивов

Для добавления и удаления элементов массива используются следующие методы:

– **push(...items)** – добавляет элементы в конец,
– **pop()** – извлекает элемент из конца,
– **shift()** – извлекает элемент из начала,
– **unshift(...items)** – добавляет элементы в начало.

Для удаления элемента массива можно использовать оператор **delete**. Однако этот оператор удаляет только значение элемента с заданным ключом без переиндексации:

```
let cars = ["Saab", "Volvo", "BMW"];
delete cars[1];
console.log(cars.length); // 3
console.log(cars[1]); // undefined
```

Универсальный метод **splice()** используется для добавления, удаления и замены элементов массива:

```
splice(index, [ deleteCount, elem1, ..., elemN])
```

Он начинает с позиции index удаляет deleteCount элементов и вставлять elem1, ..., elemN на их место. Метод возвращает массив из удалённых элементов.

Метод **slice()** возвращает новый массив, в который копирует элементы, начиная с индекса start и до индекса end (не включая

end). Оба индекса start и end могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива:

```
slice([start], [end])
```

Метод **forEach()** позволяет запускать функцию для каждого элемента массива. Его синтаксис:

```
forEach(function(item, index, array) {
  // ... делать что-то с item
});
```

Функция обратного вызова (**callback**) вызывается по очереди для каждого элемента массива и принимает следующие параметры:

– **item** – очередной элемент;
– **index** – его индекс;
– **array** – сам массив.

Для поиска элементов в массиве используются следующие методы:

– **indexOf(item, from)** ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
– **lastIndexOf(item, from)** – то же самое, но ищет справа налево.
– **includes(item, from)** – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Методы **find**, **findIndex** и **filter** в качестве условия поиска используют функцию-предикат:

```
let result = arr.find(function(item, index, array) {
  // если true - возвращается текущий элемент и перебор прерывается
  // если все итерации оказались ложными, возвращается undefined
});
```

```
let result = arr.findIndex(function(item, index, array) {
  // если true - возвращается индекс, на котором был найден элемент, и перебор прерывается
  // если все итерации оказались ложными, возвращается -1
});
let results = arr.filter(function(item, index, array) {
  // если true - элемент добавляется к результату, и перебор продолжается
  // возвращается пустой массив в случае, если ничего не найдено
});
```

Метод **map()** является одним из наиболее полезных и часто используемых. Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции:

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

Метод **sort(fn)** сортирует массив на месте, меняя в нём порядок элементов. Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам массив.

Полный список методов есть в справочнике MDN (https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array).

1.8 Работа со строками в JavaScript

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренний формат для строк — всегда UTF-16, вне зависимости от кодировки страницы.

Для задания строк в JavaScript есть разные типы кавычек. Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

```
let single = 'single-quoted';
let double = "double-quoted";
let backticks = `backticks`;
```

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку мы сможем вставлять произвольные выражения, обернув их в ${…}:

```
function sum(a, b) {
  return a + b;
}

console.log(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

Для определения длины строки используется свойство **length**.

Чтобы получить символ, который занимает позицию pos, можно использовать квадратные скобки: [pos], а также можно использовать метод **charAt(pos)**. Первый символ занимает нулевую позицию:

```
let str = `Hello`;
console.log(str.length); //5
// получаем первый символ
console.log(str[0]); // H
console.log(str.charAt(0)); // H
```

Следует обратить внимание на то, что содержимое строки в JavaScript нельзя изменить.

```
let str = `Hello`;
console.log(str); // Hello
str[0] = "J";
console.log(str); // Hello
```

Методы **toLowerCase()** и **toUpperCase()** меняют регистр символов:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Для поиска подстроки используются следующие методы:

– **indexOf(substr, pos)** ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений;
– **lastIndexOf(substr, position)** ищет подстроку с конца строки к её началу;
– **includes(substr, pos)** возвращает true, если в строке str есть подстрока substr, либо false, если нет;
– **startsWith(substr)**, **endsWith(substr)** проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой.

Для получения подстроки в JavaScript есть 3 метода:

– **slice(start [, end])** возвращает часть строки от start до (не включая) end, если end отсутствует, slice возвращает символы до конца строки;
– **substring(start [, end])** возвращает часть строки между start и end;
– **substr(start [, length])** возвращает часть строки от start длины length.

Для сравнение строк используются стандартные операторы сравнения, строки сравниваются посимвольно в алфавитном порядке.

Полный список методов можно посмотреть MDN (https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String)

1.9 Объекты

На сегодняшний день объектно-ориентированное программирование (ООП) является одной из господствующих парадигм в разработке приложений. JavaScript предоставляет возможности ООП, но имеет некоторые особенности.

Основным понятием ООП является объект – сложная комплексная структура, объединяющая в себе данные и методы их обработки.

Каждый объект может хранить свойства, которые описывают его состояние, и методы, которые описывают его поведение.

Создание нового объекта

Есть несколько способов создания нового объекта.

Первый способ заключается в использовании конструктора Object:

```
const user = new Object();
```

В данном случае объект называется user. Он определяется также, как и любая обычная переменная. Выражение new Object() представляет вызов конструктора – функции, создающей новый объект. Для вызова конструктора применяется оператор **new**. Вызов конструктора фактически напоминает вызов обычной функции.

Второй способ создания объекта представляет использование фигурных скобок – объектного литерала:

```
const user = {};
```

На сегодняшний день более распространенным является второй способ.

Свойства и методы объекта

После создания объекта мы можем определить в нем свойства. Чтобы определить свойство, надо после названия объекта через точку указать имя свойства и присвоить ему значение, или использовать альтернативный способ с помощью синтаксиса массивов:

```
const user = {};
user.name = "Alex";
user.age = 21;
user["group"] = "IP-31";
```

В данном случае объявляются три свойства name, age и group, которым присваиваются соответствующие значения. После этого мы можем использовать эти свойства, например, вывести их значения в консоли:

Методы объекта определяют его поведение или действия, которые он производит. Методы представляют собой функции. Например, определим метод, который бы выводил имя и возраст человека:

```
user.info = function(){
    console.log(user.name);
    console.log(user.age);
};
```

Также свойства и методы могут определяться непосредственно при определении объекта:

```
const user = {
    name: "Alex",
    age: 21,
    group: "IP-31",
    info: function(){
        console.log(this.name);
        console.log(this.age);
    }
};
```

Чтобы обратиться к свойствам или методам объекта внутри этого объекта, используется ключевое слово **this**. Оно означает ссылку на текущий объект.

Конструктор объектов

Кроме создания новых объектов JavaScript предоставляет нам возможность создавать новые типы объектов с помощью конструкторов.

Конструктор позволяет определить новый тип объекта. Тип представляет собой абстрактное описание или шаблон объекта. Определение типа может состоять из функции конструктора, методов и свойств.

```
function User(pName, pAge) {
    this.name = pName;
    this.age = pAge;
    this.info = function(){
        return(`Имя: ${this.name}; возраст: ${this.age}`);
    };
}
```

Конструктор – это обычная функция за тем исключением, что в ней мы можем установить свойства и методы. Для установки свойств и методов используется ключевое слово this. В данном случае устанавливаются два свойства name и age и один метод info. Как правило, название конструктора в отличие от названий обычных функций начинаются с большой буквы.

После этого в программе мы можем определить объект типа User и использовать его свойства и методы:

```
const alex = new User("Alex", 21);
console.log(alex.name); // Alex
console.log(alex.info()); // Имя: Alex; возраст: 21
```

Оператор **instanceof** позволяет проверить, с помощью какого конструктора создан объект. Если объект создан с помощью определенного конструктора, то оператор возвращает true:

```
const alex = new User("Alex", 21);
console.log(alex instanceof User); // true
```

Расширение объектов. Prototype

Кроме непосредственного определения свойств и методов в конструкторе мы также можем использовать свойство **prototype**. Каждая функция имеет свойство prototype, представляющее прототип функции. То есть свойство User.prototype представляет прототип объектов User. И любые свойства и методы, которые будут определены в User.prototype, будут общими для всех объектов User.

```
function User(pName, pAge) {
	this.name = pName;
	this.age = pAge;
	this.info = function() {
	 return(`Имя: ${this.name}; возраст: ${this.age}`);
	};
}

User.prototype.hello = function() {
	return(`${this.name} говорит: 'Привет!'`);
};
User.prototype.group = "IP-31";

const alex = new User("Alex", 21);
console.log(alex.hello());
const max = new User("Max", 21);
console.log(max.hello());
console.log(max.group);
```

Классы

С внедрением стандарта ES2015 (ES6) в JavaScript появился новый способ определения объектов – с помощью классов. Класс представляет описание объекта, его состояния и поведения, а объект является конкретным воплощением или экземпляром класса.

Для определения класса используется ключевое слово **class**.

После этого мы можем создавать объекты класса с помощью конструктора:

```
class Person{};
```

```
const alex = new Person();
const max = new Person();
```

По умолчанию классы имеют один конструктор без параметров. Поэтому в данном случае при вызове конструктора в него не передается никаких аргументов. Однако есть возможность определить в классе свои конструкторы. Также класс может содержать свойства и методы:

```
class Person{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }
    info(){
        console.log(this.name, this.age);
    }
}

const tom = new Person("Alex", 21);
alex.info(); // Tom 34
console.log(alex.name); // Tom
```

Конструктор определяется с помощью метода с именем **constructor**. По сути это обычный метод, который может принимать параметры. Основная цель конструктора – инициализировать объект начальными данными. И в данном случае в конструктор передаются два значения – для имени и возраста пользователя.

Для хранения состояния в классе определяются свойства. Для их определения используется ключевое слово this. В данном случае в классе два свойства: name и age.

Поведение класса определяют методы. В данном случае определён метод info(), который выводит значения свойств на консоль.

Наследование

Одни классы могут наследоваться от других. Наследование позволяет сократить объем кода в классах-наследниках. Например, определим следующие классы:

```
class Person{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }
    info(){
        console.log(this.name, this.age);
    }
}

class Employee extends Person{
    constructor(name, age, company){
        super(name, age);
        this.company = company;
    }
    info(){
        super.info();
        console.log("Employee in", this.company);
    }
    work(){
        console.log(this.name, "is hard working");
    }
}

const bob = new Person("Bob", 34);
const bill = new Employee("Bill", 64, "Microsoft");
bob.info();
bill.info();
bill.work();
```

Для наследования одного класса от другого в определении класса применяется оператор **extends**, после которого идет название базового класса. То есть в данном случае класс Employee наследуется от класса Person. Класс Person еще называется базовым классом, классом-родителем, суперклассом, а класс

Employee – классом-наследником, подклассом, производным классом.

Производный класс, как и базовый, может определять конструкторы, свойства, методы. Вместе с тем с помощью слова super производный класс может ссылаться на функционал, определенный в базовом. Например, в конструкторе Employee можно вручную не устанавливать свойства name и age, а с помощью выражения super(name, age); вызвать конструктор базового класса и тем самым передать работу по установке этих свойств базовому классу.

Статические методы

Статические методы вызываются для всего класса в целом, а не для отдельного объекта. Для их определения применяется оператор static. Например:

```
class Person{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }
    static nameToUpper(person){
        return person.name.toUpperCase();
    }
    info(){
        console.log(this.name, this.age);
    }
}
const tom = new Person("Dorian Gray", 28);
let personName = Person.nameToUpper(tom);
Console.log(personname); // DORIAN GRAY
```

В данном случае определен статический метод nameToUpper(). В качестве параметра он принимает объект Person и переводит его имя в верхний регистр. Поскольку статический метод относится к классу в целом, а не к объекту, то мы не можем использовать в нем ключевое слово this и через него обращаться к свойствам объекта.

2 JavaScript в веб-браузерах

2.1 Объектная модель документа (DOM)

Одой из ключевых задач JavaScript является взаимодействие с пользователем и манипуляция элементами веб-страницы. Для JavaScript веб-страница доступна в виде объектной модели документа (document object model) или сокращенно **DOM**.

В соответствии с объектной моделью документа, каждый HTML-тег является объектом. Вложенные теги являются дочерними объектами родительского элемента. Текст, который находится внутри тега, также является объектом.

Все эти объекты представляют собой узлы в иерархической структуре DOM и доступны при помощи JavaScript для изменения.

Существует 12 типов узлов. Но на практике в основном используются 4 из них:

– **Element**: html-элемент;
– **Document**: корневой узел html-документа;
– **Comment**: элемент комментария;
– **Text**: текст элемента.

У DOM-узлов есть свойства и методы, которые позволяют выбирать любой из элементов, изменять, перемещать их на странице и многое другое.

Объект document

Для работы со структурой DOM в JavaScript предназначен объект **document**, который определён в глобальном объекте window. Объект document предоставляет ряд свойств и методов для управления элементами страницы.

Для поиска элементов на странице применяются следующие методы:

– **getElementById(value)**: выбирает элемент, у которого атрибут id равен value;
– **getElementsByTagName(value)**: выбирает все элементы, у которых тег равен value;

– **getElementsByClassName(value)**: выбирает все элементы, которые имеют класс value;
– **querySelector(value)**: выбирает первый элемент, который соответствует css-селектору value;
– **querySelectorAll(value)**: выбирает все элементы, которые соответствуют css-селектору value.

Кроме методов объект document позволяет обратиться к определенным элементам веб-страницы через свойства:

documentElement: предоставляет доступ к корневому элементу <html>;

body: предоставляет доступ к элементу <body> на веб-странице;

images: содержит коллекцию всех объектов изображений (элементов img);

links: содержит коллекцию ссылок - элементов <a> и <area>, у которых определен атрибут href;

anchors: предоставляет доступ к коллекции элементов <a>, у которых определен атрибут name;

forms: содержит коллекцию всех форм на веб-странице.

Эти свойства не предоставляют доступ ко всем элементам, однако позволяют получить наиболее часто используемые элементы на веб-странице.

Объект Node. Навигация по DOM

Каждый отдельный узел, будь то html-элемент, его атрибут или текст, в структуре DOM представлен объектом Node. Этот объект предоставляет ряд свойств, с помощью которых мы можем получить информацию о данном узле:

– **childNodes**: содержит коллекцию дочерних узлов;
– **firstChild**: возвращает первый дочерний узел текущего узла;
– **lastChild**: возвращает последний дочерний узел текущего узла;
– **previousSibling**: возвращает предыдущий элемент, который находится на одном уровне с текущим;

– **nextSibling**: возвращает следующий элемент, который находится на одном уровне с текущим;
– **ownerDocument**: возвращает корневой узел документа;
– **parentNode**: возвращает элемент, который содержит текущий узел;
– **nodeName**: возвращает имя узла;
– **nodeType**: возвращает тип узла в виде числа;
– **nodeValue**: возвращает или устанавливает значение узла в виде простого текста;

Создание, добавление и удаление элементов веб-станицы

Для создания элементов объект document имеет следующие методы:

– **createElement(elementName)**: создает элемент html, тег которого передается в качестве параметра. Возвращает созданный элемент.
– **createTextNode(text)**: создает и возвращает текстовый узел. В качестве параметра передается текст узла.

Для добавления элементов можно использовать один из методов объекта Node:

– **appendChild(newNode)**: добавляет новый узел newNode в конец коллекции дочерних узлов;
– **insertBefore(newNode, referenceNode)**: добавляет новый узел newNode перед узлом referenceNode.

Иногда элементы бывают довольно сложными по составу, и гораздо проще их скопировать, чем с помощью отдельных вызовов создавать из содержимое. Для копирования уже имеющихся узлов у объекта Node можно использовать метод **cloneNode().** В метод cloneNode() в качестве параметра передаётся логическое значение: если передается true, то элемент будет копироваться со всеми дочерними узлами; если передается false – то копируется без дочерних узлов.

Для удаления элемента вызывается метод **removeChild()** объекта Node. Этот метод удаляет один из дочерних узлов.

Для замены элемента применяется метод **replaceChild(newNode, oldNode)** объекта Node. Этот метод в

качестве первого параметра принимает новый элемент, который заменяет старый элемент oldNode, передаваемый в качестве второго параметра.

Объект Element. Управление элементами

Кроме методов и свойств объекта Node в JavaScript мы можем использовать свойства и методы объектов Element. Важно не путать эти два объекта: Node и Element. Node представляет все узлы веб-станицы, в то время как объект Element представляет непосредственно только html-элементы. То есть объекты Element – это фактически те же самые узлы – объекты Node, у которых тип узла (свойство nodeType) равно 1.

Основные свойства объекта Element:

– **nodeType** позволяет узнать тип DOM-узла. Его значение – числовое: 1 для элементов, 3 для текстовых узлов, и т.д.
– **tagName** возвращает название тега (записывается в верхнем регистре).
– **innerHTML** устанавливает или получает внутреннее HTML-содержимое элемента.
– **outerHTML** устанавливает или получает полный HTML-код элемента. Запись в elem.outerHTML не меняет elem. Вместо этого она заменяет его во внешнем контексте.
– **innerText, textContent** устанавливает или получает текст внутри элемента за вычетом всех <тегов>.
– **hidden** скрывает или отображает элемент. Когда значение установлено в true, делает то же самое, что и CSS display:none.
– **offsetWidth** определяет ширину элемента в пикселях. В ширину включается граница элемента.
– **offsetHeight** определяет высоту элемента в пикселях. В высоту включается граница элемента.
– **clientWidth** определяет ширину элемента в пикселях без учета границы.
– **clientHeight** определяют высоту элемента в пикселях без учета границы.

Среди методов объекта Element можно отметить методы управления атрибутами:

– **hasAttribute(name)** проверяет на наличие атрибута.
– **getAttribute(attr)** возвращает значение атрибута attr.
– **setAttribute(attr, value)** устанавливает для атрибута attr значение value. Если атрибута нет, то он добавляется.
– **removeAttribute(attr)** удаляет атрибут attr и его значение.

Изменение стиля элементов

Для работы со стилевыми свойствами элементов в JavaScript применяются, главным образом, два подхода:
– изменение свойства style;
– изменение значения атрибута class.

Свойство **style** представляет сложный объект для управления стилем и напрямую сопоставляется с атрибутом style html-элемента. Этот объект содержит набор свойств CSS.

Свойства объекта style совпадает со свойством css. Однако ряд свойств css в названиях имеют дефис, например, font-family. В JavaScript для этих свойств дефис не употребляется. Только первая буква, которая идет после дефиса, переводится в верхний регистр: fontFamily.

С помощью свойства **className** можно установить атрибут class элемента html. Благодаря использованию классов не придется настраивать каждое отдельное свойство css с помощью свойства style. Но при этом надо учитывать, что прежнее значение атрибута class удаляется. Поэтому, если нам надо добавить класс, надо объединить его название со старым классом в виде конкатенации строк.

Для управления множеством классов гораздо удобнее использовать свойство **classList**. Это свойство представляет объект, реализующий следующие методы:
– add(className): добавляет класс className.
– remove(className): удаляет класс className.
– toggle(className): переключает у элемента класс на className. Если класса нет, то он добавляется, если есть, то удаляется.

2.2 События

Для взаимодействия с пользователем в JavaScript определен механизм событий. Например, когда пользователь нажимает кнопку, то возникает событие нажатия кнопки.

В JavaScript есть следующие типы событий:

– события мыши (перемещение курсора, нажатие мыши и т.д.);
– события клавиатуры (нажатие или отпускание клавиши клавиатуры);
– события жизненного цикла элементов (например, событие загрузки веб-станицы);
– события элементов форм (нажатие кнопки на форме, выбор элемента в выпадающем списке и т.д.);
– события, возникающие при изменении элементов DOM;
– события, возникающие при касании на сенсорных экранах;
– события, возникающие при возникновении ошибок.

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло.

Есть несколько способов назначить событию обработчик.

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

Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:

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

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick.

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

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

К примеру, elem.onclick:

```
<input id="elem" type="button" value="Нажми меня!">
```

```
<script>
   elem.onclick = function() {
       alert('Спасибо');
   };
</script>
```

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность определения нескольких обработчиков на одно событие.

Разработчики стандартов предложили альтернативный способ назначения обработчиков при помощи специальных методов **addEventListener** и **removeEventListener**.

Синтаксис добавления обработчика:

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

где event – имя события, например "click", handler – ссылка на функцию-обработчик, options – дополнительный объект со свойствами:

– once: если true, тогда обработчик будет автоматически удалён после выполнения.
– capture: фаза, на которой должен сработать обработчик. Так исторически сложилось, что options может быть false/true, это тоже самое, что {capture: false/true}.
– passive: если true, то указывает, что обработчик никогда не вызовет preventDefault().

Для удаления обработчика следует использовать removeEventListener с аналогичным синтаксисом.

Объект Event

При обработке события браузер автоматически передаёт в функцию обработчика в качестве параметра объект Event, который инкапсулирует всю информацию о событии. И с помощью его свойств мы можем получить эту информацию:

– **bubbles**: возвращает true, если событие является восходящим;
– **cancelable**: возвращает true, если можно отменить стандартную обработку события;

– **currentTarget**: определяет элемент, к которому прикреплен обработчик события;
– **defaultPrevented**: возвращает true, если был вызван у объекта Event метод preventDefault();
– **eventPhase**: определяет стадию обработки события;
– **target**: указывает на элемент, на котором было вызвано событие;
– **timeStamp**: хранит время возникновения события;
– **type**: указывает на имя события.

С помощью метода **preventDefault()** объекта Event мы можем остановить дальнейшее выполнение события.

События мыши

Наиболее часто используемые события – это события мыши:
– **click**: возникает при нажатии указателем мыши на элемент;
– **mousedown**: возникает при нахождении указателя мыши на элементе, когда кнопка мыши находится в нажатом состоянии;
– **mouseup**: возникает при нахождении указателя мыши на элементе во время отпускания кнопки мыши;
– **mouseover**: возникает при вхождении указателя мыши в границы элемента;
– **mousemove**: возникает при прохождении указателя мыши над элементом;
– **mouseout**: возникает, когда указатель мыши выходит за пределы элемента.

Объект Event является общим для всех событий. Однако для разных типов событий существуют также свои объекты событий, которые добавляют ряд специфических свойств. Так, для работы с событиями указателя мыши определён объект **MouseEvent**, который добавляет следующие свойства:
– **altKey**: возвращает true, если была нажата клавиша Alt во время генерации события;
– **button**: указывает, какая кнопка мыши была нажата;
– **clientX**: определяет координату X окна браузера, на которой находился указатель мыши во время генерации события;

– **clientY**: определяет координату Y окна браузера, на которой находился указатель мыши во время генерации события;
– **ctrlKey**: возвращает true, если была нажата клавиша Ctrl во время генерации события;
– **metaKey**: возвращает true, если была нажата во время генерации события метаклавиша клавиатуры;
– **relatedTarget**: определяет вторичный источник возникновения события;
– **screenX**: определяет координату X относительно верхнего левого угла экрана монитора, на которой находился указатель мыши во время генерации события;
– **screenY**: определяет координату Y относительно верхнего левого угла экрана монитора, на которой находился указатель мыши во время генерации события;
– **shiftKey**: возвращает true, если была нажата клавиша Shift во время генерации события

События клавиатуры

Другим распространенным типом событий являются события клавиатуры:

– **keydown**: возникает при нажатии клавиши клавиатуры и длится, пока нажата клавиша;
– **keyup**: возникает при отпускании клавиши клавиатуры;
– **keypress**: возникает при нажатии клавиши клавиатуры, но после события keydown и до события keyup. Надо учитывать, что данное событие генерируется только для тех клавиш, которые формируют вывод в виде символов, например, при печати символов. Нажатия на остальные клавиши, например, на Alt, не учитываются.

Для работы с событиями клавиатуры определен объект **KeyboardEvent**, который добавляет к свойствам объекта Event ряд специфичных для клавиатуры свойств:

altKey: возвращает true, если была нажата клавиша Alt во время генерации события;

charCode: возвращает символ Unicode для нажатой клавиши (используется для события keypress);

keyCode: возвращает числовое представление нажатой клавиши клавиатуры;

ctrlKey: возвращает true, если была нажата клавиша Ctrl во время генерации события;

metaKey: возвращает true, если была нажата во время генерации события метаклавиша клавиатуры;

shiftKey: возвращает true, если была нажата клавиша Shift во время генерации события.

2.3 Обработка форм с использованием JavaScript

Один из способов взаимодействия с пользователями представляют html-формы. Например, если нам надо получить от пользователя некоторую информацию, мы можем определить на веб-странице форму, которая будет содержать текстовые поля для ввода информации и кнопку для отправки. После ввода данных пользователем мы можем обработать введенную информацию.

В JavaScript форма представлена объектом **HtmlFormElement**. И после создания формы мы можем к ней обратиться различными способами.

Первый способ заключается в прямом обращении по имени формы:

```
<form name="search">
</form>
<script>
   let searchForm = document.search;
</script>
```

Второй способ состоит в обращении к коллекции форм документа и поиске в ней нужной формы:

```
let searchForm;
for (var i = 0; i < document.forms.length; i++) {
   if(document.forms[i].name==="search")
      searchForm = document.forms[i];
}
```

Форма имеет ряд свойств, из которых наиболее важными являются свойство name, а также свойство **elements**, которое содержит коллекцию элементов формы.

Среди методов формы надо отметить метод **submit()**, который отправляет данные формы на сервер, и метод **reset()**, который очищает поля формы:

Элементы форм

Форма может содержать различные элементы ввода html: input, textarea, button, select и т.д. Но все они имеют ряд общих свойств и методов.

Также, как и форма, элементы форм имеют свойство name, с помощью которого можно получить значение атрибута name.

Для получения и установки значения элементов **input** и **textarea** используется свойство **input.value** (строка) или **input.checked** (булево значение) для чекбоксов, textarea.value:

```
input.value = "Новое значение";
textarea.value = "Новый текст";
input.checked = true; // для чекбоксов и переключателей
```

Элемент **select** имеет 3 важных свойства:

– **select.options** – коллекция из подэлементов option;
– **select.value** – значение выбранного в данный момент option;
– **select.selectedIndex** – номер выбранного option.

Они дают три разных способа установить значение в select:

– найти соответствующий элемент option и установить в option.selected значение true;
– установить в select.value значение нужного option;
– установить в select.selectedIndex номер нужного option.

В отличие от большинства других элементов управления, select позволяет нам выбрать несколько вариантов одновременно, если у него стоит атрибут multiple. Эту возможность используют редко, но в этом случае для работы со значениями необходимо использовать первый способ, то есть ставить или удалять свойство selected у подэлементов option.

Валидация формы

HTML5 представил новую концепцию валидации HTML, названную проверкой ограничений, которая основана на:

– атрибутах элементов ввода HTML;
– псевдо-селекторах CSS;
– свойствах и методах DOM.

Для валидации формы используются следующие атрибуты элементов ввода HTML:

– **disabled**: указывает, что элемент ввода должен быть отключен;
– **max**: определяет максимальное значение элемента ввода;
– **min**: определяет минимальное значение элемента ввода;
– **pattern**: определяет шаблон значения элемента ввода;
– **required**: указывает, что для поля ввода требуется элемент;
– **type**: определяет тип элемента ввода.

Проверка ограничений CSS использует следующие псевдо-селекторы:

– **:disabled** – выбирает элементы ввода с указанным атрибутом «disabled»;
– **:invalid** – выбирает элементы ввода с недопустимыми значениями;
– **:optional** – выбирает элементы ввода без указания атрибута «required»;
– **:required** – выбирает элементы ввода с указанным атрибутом «required»
– **:valid** – выбирает элементы ввода с допустимыми значениями

Методы DOM включают функции:

– **checkValidity ()**: возвращает true, если входной элемент содержит допустимые данные;
– **setCustomValidity ()**: устанавливает свойство **validationMessage** элемента ввода.

Свойства DOM:

– **validity**: содержит логические свойства, связанные с достоверностью входного элемента;

– **validationMessage**: содержит сообщение, которое браузер отобразит, когда допустимость ложна;
– **willValidate**: указывает, будет ли проверен входной элемент.

В свою очередь свойство validity является объектом со следующими свойствами:
– **customError**: устанавливается в true, если настроено пользовательское сообщение о достоверности;
– **patternMismatch**: устанавливается в true, если значение элемента не соответствует его атрибуту pattern;
– **rangeOverflow**: устанавливается в true, если значение элемента больше, чем его атрибут max;
– **rangeUnderflow**: устанавливается в true, если значение элемента меньше его атрибута min;
– **stepMismatch**: устанавливается в true, если значение элемента недопустимо для его атрибута шага;
– **tooLong**: устанавливается в true, если значение элемента превышает его атрибут maxLength;
– **typeMismatch**: устанавливается в true, если значение элемента недопустимо для его атрибута типа;
– **valueMissing**: устанавливается в true, если элемент (с обязательным атрибутом) не имеет значения;
– **valid**: устанавливается в true, если значение элемента допустимо.

Регулярные выражения

Регулярные выражения представляют собой средство определения шаблонов символов, которые затем можно использовать при операциях со строками.

Для того, чтобы использовать регулярное выражение в строковых операциях, его необходимо создать. Сделать это можно двумя способами:

– задать литерал;
– создать объект RegExp().

Регулярные выражения поддерживаются следующими методами объекта String:

– **str.search(RegExp)** ищет в строке соответствие регулярному выражению. Если соответствие найдено, возвращается позиция его начала; в противном случае возвращается -1.
– **str.replace(RegExp, replaceStr)** осуществляет поиск соответствующих регулярному выражению подстрок в строку и заменяет на другую строку, заданную аргументом replaceStr
– **str.match(RegExp)** данный метод чем то напоминает метод search(), однако, вместо позиции найденного соответствия регулярному выражению в строке, возвращает массив, каждый элемент которого содержит вхождения искомого текста.
– **str.split(RegExp)** разбивает строку на несколько подстрок, собираемых в массив. Точки разбивки строки определяются разделителем – регулярным выражением или строкой, передаваемыми в качестве параметра.задать литерал.

Кроме того, объект RegExp так же обладает несколькими специфическими методами:

– **regexp.test(string)** проверяет переданную строку на наличие соответствия регулярному выражению. Если соответствие найдено, возвращается true, в противном случае - false;
– **regexp.exec(string)** возвращает текст, соответствующий регулярному выражению.задать литерал.

Вместе с записью регулярных выражений дополнительно используют ключи, которые модифицируют действие функций, работающих с регулярными выражениями. Существует всего два ключа:

g – означает, что метод, работающий с данным регулярным выражением должен действовать глобально. К примеру, метод replace() должен заменить все вхождения регулярного выражения а не только первое.

i – делает регулярное выражение регистронезависимым. Т.е. выражение /abc/i соответствует тексту abc, ABC, Abc и т.д.

Рассмотрим простой пример: необходимо в строке "Регулярные выражения" заменить символы 'р' на символ '_'.

```
let text = "Регулярные выражения";
console.log("/р/ : "+text.replace(/р/, "_"));
console.log("/р/g : "+text.replace(/р/g, "_"));
console.log("/р/gi : "+text.replace(/р/gi, "_"));
```

Порядок следования ключей не имеет значения: "gi" и "ig" абсолютно единтичны.

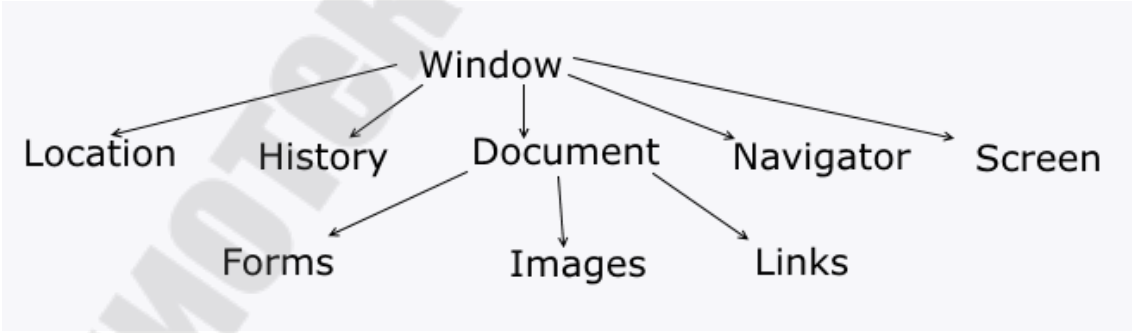
Регулярные выражения часто используются при валидации введённых значений в поля формы.

2.4 Объектная модель браузера (BOM)

Большое значение в JavaScript имеет работа с веб-браузером и теми объектами, которые он предоставляет. Например, использование объектов браузера позволяет манипулировать элементами html, которые имеются на странице, или взаимодействовать с пользователем.

Все объекты, через которые JavaScript взаимодействует с браузером, описываются таким понятием как **Browser Object Model** (Объектная Модель Браузера).

Browser Object Model можно представить в виде следующей схемы:



В вершине находится главный объект – объект window, который представляет собой браузер. Этот объект в свою очередь включает ряд других объектов, в частности, объект document,

который представляет отдельную веб-страницу, отображаемую в браузере.

Объект window представляет собой окно веб-браузера, в котором размещаются веб-страницы. Window является глобальным объектом, поэтому при доступе к его свойствам и методам необязательно использовать его имя.

Все объявляемые в программе глобальные переменные или функции автоматически добавляются к объекту window.

Ранее были рассмотрены 3 методы объекта window для взаимодействия с пользователем:

– **alert(str)**: показывает сообщение в диалоговом окне.
– **prompt(str, default)**: показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный текст в поле ввода или null, если была нажата кнопка «Отмена» или Esc с клавиатуры.
– **confirm(str)**: показывает сообщение и ждёт, пока пользователь нажмёт OK или Отмена. Возвращает true, если нажата OK, и false, если нажата кнопка «Отмена» или Esc с клавиатуры.

Все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

Объект window также предоставляет методы для работы с окном браузера:

– **window.open ()**: открыть новое окно;
– **window.close ()**: закрыть текущее окно;
– **window.moveTo ()**: переместить текущее окно;
– **window.resizeTo ()**: изменить размер текущего окна.

Объект history

Объект history предназначен для хранения истории посещений веб-страниц в браузере. Нам этот объект доступен через объект window.

Все сведения о посещении пользователя хранятся в специальном стеке (history stack). С помощью свойства **length** можно узнать, как много веб-станиц хранится в стеке.

Для перемещения по страницам в истории в объекте history определены методы **back()** (перемещение к прошлой посмотренной странице) и **forward()** (перемещение к следующей просмотренной странице)

Также в объекте history определен специальный метод **go()**, который позволяет перемещаться вперед и назад по истории на определенное число страниц. Положительное число определет перемещение вперед, а отрицательное – назад.

Объект location

Объект location содержит информацию о расположении текущей веб-страницы: URL, информацию о сервере, номер порта, протокол. С помощью свойств объекта мы можем получить эту информацию:

– **href**: полная строка запроса к ресурсу;
– **pathname**: путь к ресурсу;
– **origin**: общая схема запроса;
– **protocol**: протокол;
– **port**: порт, используемый ресурсом;
– **host**: хост;
– **hostname**: название хоста;
– **hash**: если строка запроса содержит символ решетки (#), то данное свойство возвращает ту часть строки, которая идет после этого символа;
– **search**: если строка запроса содержит знак вопроса (?), например, то данное свойство возвращает ту часть строки, которая идет после знака вопроса.

Также объект location предоставляет ряд методов, которые можно использовать для управления путем запроса:

– **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 navigator

Объект navigator содержит информацию о браузере и операционной системе, в которой браузер запущен. Он определяет ряд свойств и методов, основным из которых является свойство **userAgent**, представляющее браузер пользователя.

Объект navigator хранит свойство **geolocation**, с помощью которого можно получить географическое положение пользователя. Для получения положения используется метод **getCurrentPosition()**. Этот метод принимает два параметра: функцию, которая срабатывает при удачном запуске, и функцию, которая срабатывает при ошибке запроса геоданных.

```
function success(position) {
    let latitude = position.coords.latitude;
    let longitude = position.coords.longitude;
    let altitude = position.coords.altitude;
    let speed = position.coords.speed;

    console.log("Широта: " + latitude + "<br/>");
    console.log("Долгота: " + longitude + "<br/>");
    console.log("Высота: " + altitude + "<br/>");
    console.log("Скорость перемещения: " + speed + "<br/>");
};

function error(obj) {
    document.write("Ошибка при определении положения");
};
navigator.geolocation.getCurrentPosition(success, error);
```

Таймеры

Для выполнения действий через определенные промежутки времени в объекте window предусмотрены функции таймеров. Есть два типа таймеров: одни выполняются только один раз, а другие постоянно через промежуток времени.

Для одноразового выполнения действий через промежуток времени предназначена функция **setTimeout()**. Она может принимать два параметра:

```
let timerId = setTimeout(someFunction, period)
```

Параметр period указывает на промежуток, через который будет выполняться функция из параметра someFunction. А в качестве результата функция возвращает id таймера.

Для остановки таймера применяется функция **clearTimeout(id)**.

Функции **setInterval()** и **clearInterval()** работают аналогично функциям setTimeout() и clearTimeout() с той лишь разницей, что setInterval() постоянно выполняет определенную функцию через промежуток времени.

Функция **requestAnimationFrame()** действует аналогично setInterval() за тем исключением, что он больше заточен под анимации, работу с графикой и имеет ряд оптимизаций, которые улучшают его производительность. В метод window.requestAnimationFrame() передается функция, которая будет вызываться определенное количество раз (обычно 60) в секунду.

2.5 Взаимодействие с сетью

Сериализация объектов

Для хранения и обмена данными между клиентом и сервером чаще всего используется формат **JSON**. JSON (JavaScript Object Notation) – способ записи данных в виде строки. Поскольку формат JSON является только текстовым, его можно легко отправлять на сервер и с сервера и использовать в качестве формата данных на любом языке программирования. Например,

клиент использует JavaScript, а сервер написан на Ruby/PHP/Java или любом другом языке.

JavaScript предоставляет следующие методы для сериализации и десериализации объектов:

– **JSON.stringify(obj)** используется для преобразования объектов в строку JSON;

– **JSON.parse(str)** используется для преобразования строки в формате JSON обратно в объект.

JSON поддерживает три типа данных: примитивные значения, объекты и массивы. Примитивные значения представляют стандартные строки, числа, значение null, логические значения true и false.

```
<script>
   const student = {
   name: "Alex",
   age: 21,
   marks: {
      OAIP: 9,
      "System programming": 8,
      RPI: 9,
      ACS: 8
      }
   };
   let str = JSON.stringify(student);
   console.log(str);
//          {"name":"Alex","age":21,"marks":{"OAIP":9,"System programming":8,"RPI":9,"ACS":8}}
   let alex = JSON.parse(str);
   console.log(alex); // {name: "Alex", age: 21, marks: {…}}
</script>
```

Web storage

Для хранения данных на стороне клиента в HTML5 была внедрена новая концепция – web-storage. Web storage состоит из двух компонентов: **session storage** и **local storage**.

Session storage представляет временное хранилище информации, которая удаляется после закрытия браузера.

Local storage представляет хранилище для данных на постоянной основе. Данные из local storage автоматически не удаляются и не имеют срока действия. Объем local storage составляет в Chrome и Firefox 5 Мб для домена.

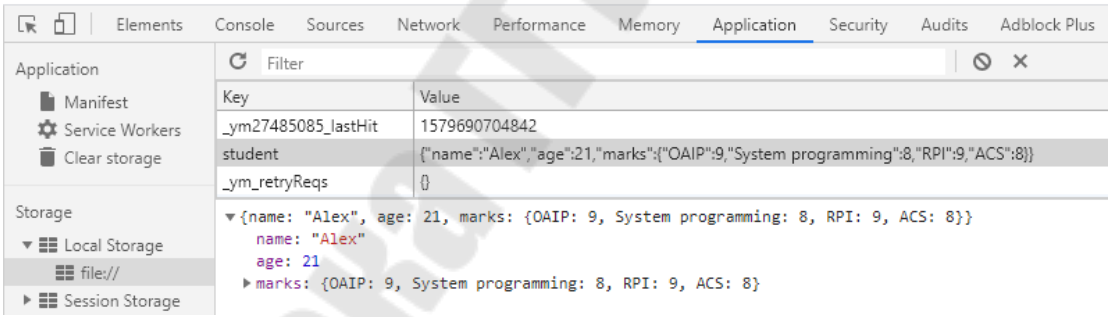
Все данные в web storage представляют набор пар ключ-значение. То есть каждый объект имеет уникальное имя-ключ и определенное значение.

Для работы с local storage в javascript используется объект **localStorage**, а для работы с session storage – объект **sessionStorage**. Эти объекты предоставляют следующие методы для работы с данными:

– **setItem()** – для сохранения данных;
– **getItem()** – для получения сохраненных данных;
– **removeItem()** – для удаления объект из хранилища;
– **clear()** – для для полного удаления всех объектов из хранилища.

Например:

```
localStorage.setItem("student", JSON.stringify(student));
```



Технология AJAX

Ajax представляет технологию для отправки запросов к серверу из клиентского кода JavaScript без перезагрузки страницы. Сам термин расшифровывается как Asynchronous JavaScript And XML. То есть изначально AJAX предполагал асинхронное взаимодействие клиента и сервера посредством данных в формате XML. В настоящее время основным форматом данных для обмена данных между клиентом и сервером является формат JSON.

Объект XMLHttpRequest

Для создания приложений, использующих Ajax, применяются различные способы. Но самым распространенным способом является использование объекта XMLHttpRequest:

```
const request = new XMLHttpRequest();
```

После создания объекта XMLHttpRequest можно отправлять запросы к серверу. Но для начала надо вызвать метод open() для инициализации:

```
request.open("GET", "http://localhost/test.txt", false);
```

Метод open() принимает три параметра: тип запроса (GET, POST, HEAD, PUT), адрес запроса и третий необязательный параметр – логическое значение true или false, указывающее, будет ли запрос осуществляться в асинхронном режиме. По умолчанию, если третий параметр не используется, то запрос отправляется в асинхронном режиме, что позволяет параллельно с выполнением запроса выполнять также и другой код JavaScript.

Кроме того, метод open() может принимать еще два параметра: логин и пароль пользователя, если для выполнения запроса нужна аутентификация:

```
request.open("GET", "http://localhost/home.php", true, "login", "password");
```

После инициализации запроса методом open() необходимо отправить запрос с помощью метода send():

```
request.send();
```

Объект XMLHttpRequest имеет ряд свойств, которые позволяют проконтролировать выполнение запроса:

– **status**: содержит статусный код ответа HTTP, который пришел от сервера. С помощью статусного кода можно судить об успешности запроса или об ошибках, которые могли бы возникнуть при его выполнении. Например,

статусный код 200 указывает на то, что запрос прошел успешно. Код 403 говорит о необходимости авторизации для выполнения запроса, а код 404 сообщает, что ресурс не найден и так далее.

– **statusText**: возвращает текст статуса ответа, например, "200 OK"
– **responseType**: возвращает тип ответа.
– **response**: возвращает ответ сервера.
– **responseText**: возвращает текст ответа сервера.
– **responseXML**: возвращает xml, если ответ от сервера в формате xml.

При асинхронном запросе объект XMLHttpRequest использует свойство **readyState** для хранения состояния запроса. Состояние запроса представляет собой число:

0: объект XMLHttpRequest создан, но метод open() еще не был вызван для инициализации объекта

1: метод open() был вызван, но запрос еще не был отправлен методом send()

2: запрос был отправлен, заголовки и статус ответа получены и готовы к использованию

3: ответ получен от сервера

4: выполнение запроса полностью завершено (даже если получен код ошибки, например, 404)

Событие readystatechange возникает каждый раз, когда изменяется значение свойства readyState.

Отправка запросов

GET-запрос характеризуется тем, что данные могут отправляться в строке запроса:

```
<script>
// объект для отправки
const user = {
    name: "Alex",
    age: 21
};
```

```
const request = new XMLHttpRequest();
function reqReadyStateChange() {
   if (request.readyState == 4) {
      let status = request.status;
      if (status == 200) {
       document.getElementById("output").innerHTML =
request.responseText;
      }
   }
}
// строка с параметрами для отправки
let body = "name=" + user.name + "&age="+user.age;
request.open("GET",
"http://localhost:8080/postdata.php?"+body);
request.onreadystatechange = reqReadyStateChange;
request.send();
</script>
```

Для отправки берем свойства объекта user и формируем из их значений сроку с параметрами: "name=" + user.name + "&age=" + user.age. Затем эта строка добавляется к строке запроса в методе open("GET", "http://localhost:8080/postdata.php?"+body)

Для отправки данных методом POST надо установить заголовок Content-Type с помощью метода **setRequestHeader()**. В данном случае заголовок имеет значение application/x-www-form-urlencoded. Данные встраиваются в тело запроса при посылке запроса.

```
const user = {
   name: "Nick",
   age: 23
};

const request = new XMLHttpRequest();
function reqReadyStateChange() {
   if (request.readyState == 4 && request.status == 200)
      document.getElementById("output").innerHTML=request.r
esponseText;
}
```

```
let body = "name=" + user.name + "&age="+user.age;
request.open("POST", "http://localhost:8080/postdata.php");
request.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
request.onreadystatechange = reqReadyStateChange;
request.send(body);
```

Promise в Ajax-запросах

Для более удобного способа организации асинхронного кода в современном JavaScript используется объект **1`**

-, который обертывает асинхронную операцию в один объект и позволяет определить действия, выполняющиеся при успешном или неудачном выполнении этой операции.

Promise – это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

Синтаксис создания Promise:

```
const promise = new Promise(function(resolve, reject) {
  // Эта функция будет вызвана автоматически
  // В ней можно делать любые асинхронные операции,
  // А когда они завершатся — нужно вызвать одно из:
  // resolve(результат) при успешном выполнении
  // reject(ошибка) при ошибке
})
```

Для обработки результата объекта Promise вызывается метод then(), который принимает два параметра: функцию, вызываемую при успешном выполнении запроса, и функцию, которая вызывается при неудачном выполнении запроса. Метод then() также возвращает объект Promise. Поэтому при необходимости мы можем применить к его результату цепочки вызовов метода then: promise.then().then().

Рассмотрим пример использования промиса с использованием промисификация – это когда берут асинхронную функциональность и делают для неё обёртку, возвращающую промис.

После промисификации использование функциональности зачастую становится гораздо удобнее. Например:

```
function httpGet(url) {
  return new Promise(function(resolve, reject) {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
    xhr.onload = function() {
      if (this.status == 200) {
        resolve(this.response);
      } else {
        const error = new Error(this.statusText);
        error.code = this.status;
        reject(error);
      }
    };
    xhr.onerror = function() {
      reject(new Error("Network Error"));
    };
    xhr.send();
  });
}

httpGet("/article/promise/user.json")
  .then(
    response => alert(`Fulfilled: ${response}`),
    error => alert(`Rejected: ${error}`)
  );
```

Метод Fetch

Современные стандарт поддерживает метод **fetch** – встроенный метод для AJAX-запросов.

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* обрабатываем результат */)
```

Чаще всего fetch используется вместе с конструкцией async/await, которая позволяет дождаться выполнения промиса

```
let response = await fetch(url, options); // завершается с заголовками ответа
let result = await response.json(); // читать тело ответа в формате JSON
```

Метод fetch использует следующие опции для формирования запроса:

– **method** – HTTP-метод;
– **headers** – объект с запрашиваемыми заголовками;
– **body** – данные для отправки (тело запроса) в виде текста, FormData, BufferSource, Blob или UrlSearchParams.

Для получения тела ответа используются следующие методы:

– **response.text()** – возвращает ответ как обычный текст;
– **response.json()** – преобразовывает ответ в JSON-объект;
– **response.formData()** – возвращает ответ как объект FormData (кодировка form/multipart, см. следующую главу);
– **response.blob()** – возвращает объект как Blob (бинарные данные с типом);
– **response.arrayBuffer()** – возвращает ответ как ArrayBuffer (низкоуровневые бинарные данные).

3 Библиотека jQuery

Современное веб-программирование и создание веб-приложений уже невозможно представить без использования языка JavaScript. Однако в настоящее время, все чаще используется не чистый код JavaScript, а JavaScript-фреймворки и библиотеки. Одной из таких библиотек является jQuery.

Подключив библиотеку jQuery вместо десятков команд на JavaScript можно написать несколько команд. Команды основаны на селекторах и классах CSS.

Библиотеку jQuery можно скачать с сайта http://code.jquery.com/. Если нет желания скачивать библиотеку её можно подключить с CDN (Content Delivery Network) компаний Google или Microsoft.

Вся работа с библиотекой сводится к выбору (query) HTML элемента и выполнения над ним различных действий

Основной синтаксис:

```
$(selector).action(),
```

где $() – функция доступа (определения) jQuery. (jQuery() – полное название функции $), selector – "query(запрос)" для определения HTML элементов, action() – действия над элементом (элементами).

Примеры:

```
$(this).hide() // скрывает текущий элемент.
$("p").hide() // скрывает все абзацы.
$(".test").hide() // скрывает все элементы класса "test".
$("#test").hide() // скрывает элемент с id="test".
```

Создание элементов DOM

```
$("<div>Hi</div>");
$("<div>")
```

Можно использовать кавычки, можно апострофы. Но наличие тегов обязательно. Задавать просто текст нельзя. Сама по себе эта команда ничего не выведет, надо этот узел привязать к родительскому элементу. Делается это так:

```
$("<div>Hi</div>").insertAfter("#a1")
```

Здесь a1 – это идентификатор объекта, после которого необходимо вставить элемент.

Работа с полученным набором значений

В терминах jQuery эти наборы называют обернутыми в объект jQuery.

Для определения количество выбранных элементов используется свойство length или метода size, который также возвращает число выбранных элементов:

```
let num1 = $("tr:nth-child(odd)").length;
let num2 = $("tr:nth-child(odd)").size();
```

Для получения доступа к элементу выборки можно использовать индекс или использовать функцию eq(index). Второй способ более удобен так как функция возвращает объект:

```
let firstElement = $("tr:nth-child(odd)")[0];.
let elem = $("tr:even").eq(0);.
```

Для перебора элементов выборки можно использовать стандартные методы JavaScript, а также использовать специальный метод each:

```
$(function(){
    $("tr:even").each(function(index, elem){
        console.log(index + ". " + elem.innerHTML);
    });
});
```

В метод each в качестве параметра передается безымянная функция, которая принимает два параметра: index - индекс элемента в наборе и elem - сам элемент.

Манипулирование объектами на странице

Библиотека jQuery предлагает нам инструментарий для манипуляции свойствами и атрибутами элементов:

– **prop(property, [value])** получает или изменяет значение свойства элемента;
– **removeProp(property)** удаляет свойство;
– **attr(attributeб [value])** получить или изменяет значение атрибута элемента;
– **css**(**property, [value]**) применяется для работы со стилями элемента;
– **html()** используется для получения или установки разметки;
– **text()** используется для получения или установки текста;
– **val()** используется для получения значений элементов форм.

Обработка событий

Различные браузеры по-своему могут обрабатывать события, jQuery пытается сгладить эти неприятности.

Модель событий jQuery обладает следующими свойствами:

– поддерживает единый метод установки событий;
– позволяет устанавливать несколько обработчиков для события;
– использует стандартные названия типов событий;
– предоставляет единые методы отмены события и блокирования действий по умолчанию.

jQuery предоставляет специальный метод **on**, который позволяет создать и зарегистрировать обработчики как для существующих, так и для будущих элементов, которых еще не в структуре DOM. Этот метод имеет следующие варианты использования:

– on('событие', 'селектор', обработчик_события);
– on('событие', 'селектор', данные_события, обработчик_события) то же самое, что и в предыдущем случае, плюс параметр для передачи данных события в объект Event (e.data).

Для удаления обработчика используется команда unbind()

Метод **trigger()** используется для вызова обработчиков событий вручную. Например, вызвав метод trigger для нужного

элемента мы можем также и вызвать обработчик события. Он имеет следующие варианты использования:

– trigger('событие'): вызов обработчика для данного события;

– trigger(объект_Event): вызов обработчика с использованием объекта Event, который содержит данные о том, какой именно обработчик надо вызвать.

Кроме специализированных методов jQuery предоставляет прямые методы для обработки событий. Эти методы, как правило, носят наименование обрабатываемого события, а в качестве параметра принимают функцию обработчика данного события:

```
$('button').first().click(function(){
    $(this).css('background-color', 'silver');
});
```

Эффекты анимации

К **базовым эффектам** в jQuery относятся эффекты скрытия и отображения элементов, которые достигаются с помощью методов **show()**, **hide()** и **toggle()**.

Все эти методы могут использоваться в трех вариантах (рассмотрим на примере метод hide):

– **hide():** метод без параметров

– **hide([duration][, complete]):** принимает два необязательных параметра. Параметр duration указывает как долго анимация элемента будет длиться. По умолчанию его значение равно 400 миллисекунд. Параметр complete представляет функцию, вызываемую методом по завершению анимации

– **hide([duration] [, easing][, complete]):** то же самое, только добавляется параметр easing, который принимает название функции плавности анимации в виде строки. По умолчанию его значение равно "swing". Также можно использовать значения 'slow' и 'fast', которые соответствуют длительности эффекта в 600 и 200 миллисекунд.

Эффекты скольжения позволяют нам плавно скрыть или раскрыть элемент. Эффекты скольжения реализованы в виде методов **slideUp()**, **slideDown()** и **slideToggle()**.

Эффекты прозрачности позволяют нам, плавно изменяя прозрачность элемента, скрыть его или отобразить. Эффекты прозрачности реализованы с помощью методов **fadeOut()**, **fadeIn()**, **fadeTo()** и **fadeToggle()**.

Для создания более сложных по характеру эффектов используется метод **animate()**:

```
animate(properties [,duration] [,easing] [,complete])
```

Обязательный параметр properties содержит набор css-свойств, у которых указываются финальные значения. Параметр duration указывает, как долго будет длиться изменение прозрачности элемента. Параметр easing принимает название функции плавности анимации в виде строки. Параметр complete представляет функцию обратного вызова, вызываемую методом по завершении анимации

4 JavaScript на стороне сервера

4.1 Обзор платформы Node.js

Node.js — кроссплатформенная среда выполнения JavaScript с открытым исходным кодом. Код доступен по адресу https://github.com/nodejs/node.

Платформа Node.js построена на базе движка V8 от компании Google, который позволяет транслировать вызовы на языке JavaScript в машинный код. Данная платформа, в основном, используется для создания веб-серверов, однако сфера её применения этим не ограничивается.V8 написан на языке C++ и реализует стандарт ECMAScript. Движок V8 может работать автономно или может быть встроен в любое приложение C++. Просмотреть документацию можно по адресу https://v8.dev/docs.

Одной из основных привлекательных особенностей Node.js является скорость. JavaScript-код, выполняемый в среде Node.js, может быть в два раза быстрее, чем код, написанный на компилируемых языках, вроде C или Java, и на порядки быстрее интерпретируемых языков наподобие Python или Ruby. Причиной подобного является неблокирующая (асинхронная) архитектура платформы.

Приложение Node.js работает в одном процессе, не создавая новый поток для каждого запроса. Node.js предоставляет набор асинхронных примитивов ввода-вывода в своей стандартной библиотеке libuv (https://github.com/libuv/libuv), которые предотвращают блокировку кода JavaScript.

Когда Node.js нужно выполнить операцию ввода-вывода, вроде загрузки данных из сети, доступа к базе данных или к файловой системе, вместо того, чтобы заблокировать ожиданием результатов такой операции главный поток, Node.js инициирует её выполнение и продолжает заниматься другими делами до тех пор, пока результаты выполнения этой операции не будут получены.

Node.js предоставляет огромное количество опенсорсных пакетов, которые может свободно использовать любой Node.js-разработчик. В сентябре 2022 года сообщалось, что в реестре npm было зарегистрировано более 2,1 миллиона пакетов, что делает его крупнейшим хранилищем кода, образуя экосистема Node.js.

Для работы с реестром Node.js предоставляет менеджер пакетов, который называется npm, а сам реестр npmjs (https://www.npmjs.com/).

Для установки Node.js необходимо перейти на официальный сайт https://nodejs.org/en/, там же находится вся документация по Node.js.

После успешной установки вы можем ввести в командной строке/терминале команду node -v, и нам отобразится текущая версия node.js:

```
C:\Users\Nickolay Samovendyuk>node -v
V24.12.0
```

Для исполнения кода можно воспользоваться режимом REPL (Read-Evaluate-Print-Loop, цикл «чтение — вычисление — вывод»):

```
PS G:\GSTU\NodeJS\Intro> node
Welcome to Node.js v18.16.1.
Type ".help" for more information.
> console.log('hello')
hello
undefined
> function sum(a, b) { return a + b }
undefined
> console.log(`2 + 3 = ${sum(2, 3)}`)
2 + 3 = 5
undefined
>
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
> .exit
PS G:\GSTU\NodeJS\Intro>
```

Такой режим работы похож с набором команд в консоли браузера.

Однако вместо того чтобы вводить весь код напрямую в консоль, удобнее вынести его во внешний файл. Допустим у нас есть новый файл app.js со следующим кодом:

```
console.log('Web-technologies!!!');
```

Для запуска кода необходимо выполнить команду node с указанием имени файла:

```
PS C:\Users\Nickolay Samovendyuk\Documents\GSTU\Node.js> node app.js
Web-technologies!!!
```

4.2 Основные приемы работы в Node.js

Работа с аргументами командной строки

При запуске Node.js-скриптов им можно передавать аргументы. Вот обычный вызов скрипта:

```
node app.js
```

Передаваемые скрипту аргументы могут представлять собой как самостоятельные значения, так и конструкции вида ключ-значение. В первом случае запуск скрипта выглядит так:

```
node app.js nick
```

Во втором — так:

```
node app.js user=nick
```

От того, какой именно способ передачи аргументов используется, зависит то, как с ними можно будет работать в коде скрипта.

Для получения доступа к аргументам командной строки, используется стандартный объект Node.js — **process**. У него есть свойство argv, которое представляет собой массив, содержащий, кроме прочего, аргументы, переданные скрипту при запуске.

Первый элемент массива argv содержит полный путь к файлу, который выполняется при вводе команды node в командной строке. Второй элемент — это путь к выполняемому файлу

скрипта. Все остальные элементы массива, начиная с третьего, содержат то, что было передано скрипту при его запуске.

Для демонстрации создадим перепишем код файла app.js:

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

const args = process.argv;
args.forEach(element => {
   console.log(element);
});
```

И запустим сценарий с парамерами командной строки:

```
node app.js 1 2 nick
Hello, world!!!

C:\Program Files\nodejs\node.exe
C:\Users\Nickolay
Samovendyuk\Documents\GSTU\Node.js\Introduction\app.js
1
2
nick
```

Работа с модулями

Разработка программного кода в среде Node.js осуществляется в виде набора взаимосвязанных модулей. Среда поддерживает две модульные системы: модули **CommonJS** и модули **ECMAScript**.

Отличительной чертой этих систем заключается в особенностях импорта и экспорта. В данном пособии будет использоваться классический подход с использованием системы модулей CommonJS.

В этой системе экспорт осуществляется через глобальный объект **module.exports**, а для импорта используется функция **require** с возможностью деструктуризации импортируемого объекта.

Для создания проекта приложения необходимо настроить конфигурационный фал **package.json**. Он представляет собой файл

манифеста для проекта Node.js и содержит сведения о метаданных проекта. Кроме того, он регулирует, например, порядок управления зависимостями, то, какие файлы попадут в пакет, предназначенный для npm, и многое другое (например модули какого типа будут использованы в проекте).

Файл package.json не пишется вручную. Он создается в результате выполнения команды init. Эту команду можно выполнять двумя основными способами:

– **npm init**. Эта команда запускает мастер, предлагающий заполнить такие данные, как имя проекта, его версия, описание, точка входа, команда тестирования, репозиторий GIT, ключевые слова, автор и лицензия.

– **npm init -y**. Эта команда с флагом -y представляет собой ускоренную версию команды npm init. Ускорение связано с тем, что эта команда не интерактивная. Вместо этого все поля, которые требуется заполнить при выполнении команды npm init, получают значения по умолчанию.

Создадим собственный модуль mylib.js с набором данных, который можно будет использовать в различных скриптах:

```
const sum = (a, b) => a + b;
const PI = 3.1415;
const person = {
   name: "nick",
   age: 20
}

module.exports = {
   sum: sum,
   PI: PI,
   person: person
};
```

Для использования модуля в других скриптах необходимо импортировать данные (файл app.js):

```
const mylib = require('./mylib');

console.log(mylib);
```

Вызов скрипта:

```
node app.js
{ sum: [Function: sum], PI: 3.1415, person: { name: 'nick', age: 20 }
```

Обработка событий. Цикл событий Event Loop

Подавляющее большинство функционала Node.js применяет асинхронную событийную архитектуру, которая использует специальные объекты − эмиттеры для генерации различных событий, которые обрабатываются специальными функциями − обработчиками или слушателями событий. Все объекты, которые генерируют события, представляют экземпляры класса **EventEmitter**.

С помощью функции **eventEmitter.on()** к определенному событию по имени прикрепляется функция обработчика. Причём для одного события можно указать множество обработчиков. Когда объект EventEmitter генерирует событие, происходит выполнение всех этих обработчиков.

Весь необходимый функционал сосредоточен в модуле **events**, который необходимо подключить. С помощью функции on() связываем событие, которое передается в качестве первого параметра, с некоторой функцией, которая передается в качестве второго параметра. Для генерации события и вызова связанных с ним обработчиков выполняется функция **emitter.emit()**, в которое передается название события.

```
// файл appEvents.js
const EventEmitter = require("events");
const eventEmitter = new EventEmitter();

eventEmitter.on("appEvent", () => {
    console.log("appEvent has occured");
});

eventEmitter.emit("appEvent");

// результат в консоли
```

```
node appEvents.js
appEvent has occured
```

В Node.js мы можем наследоваться от класса EventEmitter для возможности задания обработчиков событий для собственных объектов.

```
// файл appEvents.js
const EventEmitter = require("events");

class Clock extends EventEmitter {
    constructor() {
        super();
        const oneSecond = 1000;
        let isOdd = true;
        this.interval = setInterval(() => {
            this.emit(isOdd ? 'tick' : 'tock');
            isOdd = !isOdd;
        }, oneSecond);
    }
};

// результат в консоли
node appEvents.js
tick
tock
tick
tock
tick
tock
tick
tock
```

Если мы хотим, чтобы обработчик запускался только после первого события применяется метод **once**:

```
clock.once('tick', () => console.log("tick"));
clock.once('tock', () => console.log("tock"));
```

При запуске приложения через Node.js создается свой собственный цикл событий (**event loop**). Event loop − это то, что позволяет Node.js выполнять неблокирующие операции ввода-вывода.

Обработка событий выполняется во время одной из фаз, для каждой из которых создаётся своя очередь выполнения. На следующей диаграмме показан упрощённый обзор порядка операций цикла событий.

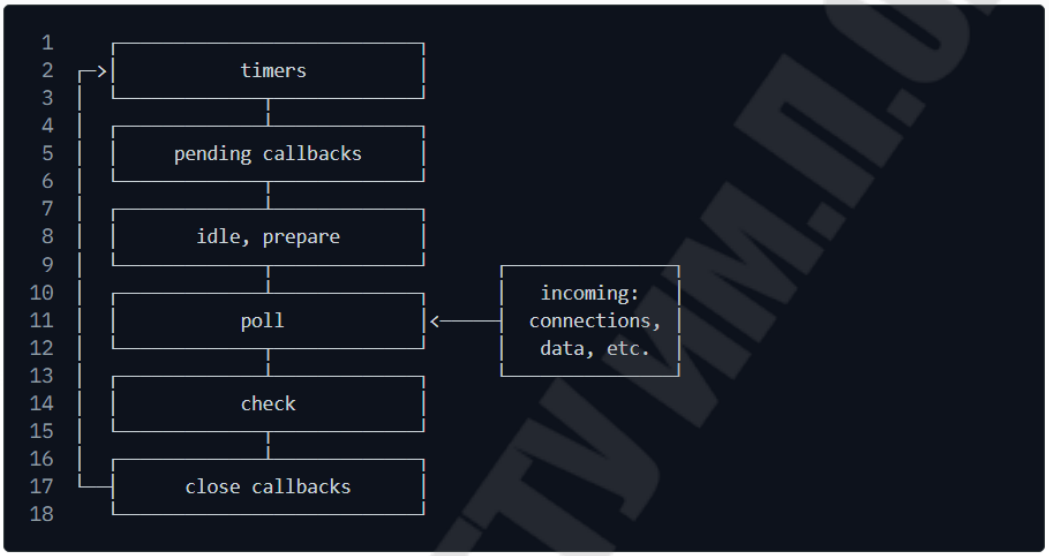


Рисунок 4.1 – Цикл событий

Во время фазы **timers** выполняются зарегистрированные таймерами функции, причем переход на стадию контролируется фазой **poll**. Из-за блокировки фазой poll цикла событий, таймеры могут выполняться с некоторой задержкой, т. е. через больший интервал времени, чем тот, который был задан в функциях setInterval() и setTimeout().

Во время фазы **pending callbacks** выполняются действия, отложенные на предыдущей итерации event loop. Например, это могут быть сообщения об ошибках, которые не были выведены ранее из-за попытки системы их исправить.

Фаза **idle**, **prepare** зарезервирована для выполнение внутренних действий, необходимых самому event loop.

При переходе в фазу **poll** в первую очередь проверяется, сформировалась ли очередь из callback-функций, выполненных асинхронных действий. Если очередь не пуста, то в синхронном

порядке начинается выполнение всех функций, находящихся в очереди. Выполнение будет продолжаться до тех пор, пока очередь не опустеет или не будет достигнут лимит выполняемых за раз callback-функций.

Во время фазы **check** выполняются callback-функции, зарегистрированные функцией **setImmediate()**;

Во время фазы **close callbacks** вызываются функции, зарегистрированные для действий, возникающих внезапно. например, событие close или disconnect для сокет соединения.

Между каждым запуском цикла обработки событий Node.js проверяет, ожидает ли он каких-либо асинхронных операций ввода-вывода или таймеров, и корректно завершает работу, если их нет.

Ввод-вывод информации

Стандартный вывод информации в Node.js использует объект **console**, который предоставляет множество очень полезных способов взаимодействия с командной строкой.

Это в основном такой же консольный объект, который мы использовали в браузере.

Самым простым и наиболее часто используемым методом является console.log(), который выводит строку, переданную в качестве аргумента.

Мы можем форматировать результат, передавая в метод переменные и спецификаторы формата.

Например:

```
console.log('My %s has %d ears', 'cat', 2); // My cat has 2 ears
```

Спецификаторы формата:
%s - отформатировать переменную как строку
%d - форматировать переменную как число
%i - форматировать переменную только как ее целую часть
%o - форматировать переменную как объект

Можно заметить сходство со стандартной функцией вывода printf() языка C.

Метод **console.clear()** очищает консоль (поведение может зависеть от используемой консоли). Под капотом используется модуль node:console. Более подробно информацию по объекту console можно найти https://nodejs.org/docs/latest/api/console.html.

Начиная с 7 версии Node.js содержит модуль **readline**, который позволяет принимать данные из потоков, которые можно читать, например, из process.stdin. Этот поток, во время выполнения Node.js-программы, представляет собой то, что вводят в терминале. Данные вводятся по одной строке за раз.

Модуль readline предоставляет интерфейс для чтения данных из открытого для чтения потока по одной строке за раз. Ниже приведен пример работы с консолью:

```
//файл readline.js
const readline = require('readline');
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question('Enter your name: \n', (answer) => {
    console.log(`Hello, ${ answer }`);
    rl.close();
});

rl.setPrompt('Enter your name:\n');
rl.prompt();
rl.on('line', (userInput) => {
    console.log(`Hello, ${ userInput }`);
    rl.close('line');
});

//вызов
node readline.js
Enter your name:
nick
Hello, nick
```

В этом примере приведён классический пример ассинхронного программирования. Мы подписываемся на событие 'line' и записываем колбэк-функцию обработчика события, которая вызовется при вводе информации пользователя.

Более подробна информация по модулю представлена на сайте https://nodejs.org/docs/latest/api/readline.html.

Работа с файловой системой

Для работы с файловой системой используется **File System module**. Для использования модуля его необходимо импортировать. Все методы импортируемого объекта имеют как асинхронные так и синхронные формы.

Асинхронная форма всегда принимает завершающий обратный вызов в качестве последнего аргумента. Аргументы, передаваемые на завершающий обратный вызов, зависят от метода, но первый аргумент всегда зарезервирован для исключения. Если операция была завершена успешно, то первый аргумент будет null или undefined.

При использовании синхронной формы любые исключения сразу срабатывают. Поэтому необходимо использовать контролируемый блок **try/catch** для обработки исключений.

Основные приемы при работе с файлами приведены ниже:

```
const fs = require('fs');
//create file
fs.writeFile('test.txt', 'simple text', (err) => {
   if (err)
      console.log(err)
   else
      console.log('file was created succesfully!!!');
});

//чтение файла
fs.readFile('test.txt', (err, data) => {
   if (err)
      console.log(err);
   else
      console.log(data);
```

```
});

fs.readFile('test.txt', 'utf8', (err, data) => {
   if (err)
      console.log(err);
   else
      console.log(data);
});

//rename file
fs.rename('test.txt', 'test2.txt', (err) => {
   if (err)
      console.log(err);
   else
      console.log('file was renamed!!!');
});

//append data
fs.appendFile('test2.txt', '\nnew line', (err) => {
   if (err)
      console.log(err);
   else
      console.log('file was changed!!!');
});

//delete file
fs.unlink('test2,txt', (err) => {
   if (err)
      console.log(err);
   else
      console.log('file was deleted!!!');
});
```

В процессе работы приложения файлы подвергаются модификации. Node.js позволяет программисту отслеживать изменения с использованием класса **FSWatcher**:

```
const fs = require('fs');
const filePath = './test2.txt';
```

```
//file statistic
fs.stat(filePath, (err, stats) => {
    if (err)
        console.log(err);
    else
        console.log(stats);
});

const watcher = fs.watch(filePath);
const appendData = (data) => {
    console.log('append - ', new Date(), `data - ${ data }`);
    fs.appendFileSync(filePath, data);
}
const stopWatch = () => {
    console.log('stop - ', new Date());
    watcher.close();
}

console.log('start - ', new Date());
appendData('text1');

watcher.on('change', (eventType, fileName) => {
    console.log('change - ', new Date(), `, Event type -
${ eventType }, File name - ${ fileName }`);
});

setTimeout(() => {
    appendData('text2')
}, 5000);

setTimeout(() => {
    appendData('text3');
    stopWatch()
}, 10000);
```

Модуль FS позволяет программисту осуществлять стандартные операции не только с файлами, но и с директориями:

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

```
// create dir
fs.mkdir('data', (err) => {
    if (err)
        console.log(err);
    else {
        console.log('directoria was created!!!');
        fs.writeFile('./data/test3.txt', 'working with directories', (err)
=> {
            if (err)
                console.log(err);
        });
    }

});

//delete dir
fs.rmdir('data', (err) => {
    if (err)
        console.log(err);
    else
        console.log('directoria was deleted!!!');
});

fs.unlink('./data/test3.txt', (err) => {
    if (err)
        console.log(err);
    else {
        fs.rmdir('data', (err) => {
            if (err)
                console.log(err);
            else
                console.log('directoria was deleted!!!');
        });
    }
});

//read files in dir
const files = fs.readdirSync('./');
console.log(files);
```

Следует обратить внимание на то, что если в директории находятся вложенные каталоги или файлы метод rmdir выдает ошибку. Следует сперва удалить всё из директории, а затем только удалять её.

Как уже упоминалось в обзоре, Node.js предоставляет работу с потоками. Потоки бывают различных типов, среди которых можно выделить потоки для чтения и потоки для записи. Для создания потока для записи применяется метод **fs.createWriteStream()**. Запись данных производится с помощью метода **write()**, в который передаются данные. Для окончания записи вызывается метод **end()**. Для создания потока для чтения используется метод **fs.createReadStream()**. Сам поток разбивается на ряд кусков или **чанков** (**chunk**). И при считывании каждого такого чанка, возникает событие **data**. С помощью метода on() мы можем подписаться на это событие и вывести каждый кусок данных на консоль:

```
const fs = require('fs');
// stream for reading
const readStream = fs.createReadStream('./example.html');
// stream for writing
const writeStream = fs.createWriteStream('./example2.html');
readStream.on('data', (chunk) => {
   console.log('\nnew chank\n', chunk);
   writeStream.write(chunk);
});
```

Следует отметить, что при чтении файлов больших объемов потоки позволяют считать информацию, а вот стандартный метод **readFile** приведет к переполнению буфера.

Только работой с файлами функциональность потоков не ограничивается, также имеются сетевые потоки, потоки шифрования, архивации и т.д., но общие принципы работы с ними будут те же, что и у файловых потоков.

Как вы знаете, для обмена данными между процессами часто используются именованные и неименованные каналы. Node.js предоставляет неименованный канал **PIPE**, который связывает

поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи.

Перепишем предыдущий пример с использованием pipe:

```
const fs = require('fs');
// stream for reading
const readStream = fs.createReadStream('./example.html');
// stream for writing
const writeStream = fs.createWriteStream('./example3.html');

readStream.pipe(writeStream);
```

Рассмотрим использование канала для архивации и разархивации файла. Здесь нам надо сначала считать файл, затем сжать данные и в конце записать сжатые данные в файл-архив, а затем прочитать заархивированные данные распаковать и записать в новый файл. Для работы этих целей используется модуль **zlib**:

```
const zlib = require("zlib");
let readStream = fs.createReadStream("./example.html", "utf8");
let writeStream = fs.createWriteStream("./example.html.gz");

//compressed
const gzip = zlib.createGzip();
readStream.pipe(gzip).pipe(writeStream);

// uncompressd
const gunzip = zlib.createGunzip();
readStream = fs.createReadStream("./example.html.gz");
writeStream = fs.createWriteStream("./example4.txt");
readStream.pipe(gunzip).pipe(writeStream);
```

4.3 Клиенты и серверы HTTP

Протокол HTTP

Любое веб-приложение разрабатывается на основе архитектуры клиент-сервер. В качестве клиента выступает

браузер, который выполняет запросы к веб-сервером с использованием протокола HTTP.

Протокол HTTP – это простой протокол, который использует для передачи содержимого надежные службы протокола TCP. Благодаря этому HTTP считается очень надежным протоколом для обмена содержимым.

HTTPS – это безопасная версия протокола HTTP, которая реализует протокол HTTP с использованием протокола TLS для защиты базового TCP-подключения. За исключением дополнительной конфигурации, необходимой для настройки TLS, использование протокола HTTPS по сути не отличается от протокола HTTP.

Запрос представляет собой структурированный в соответствии с правилами используемого протокола фрагмент текста. Он состоит из трёх частей:

- строка запроса;
- заголовок запроса;
- тело запроса.

Строка запроса представляет собой одну текстовую строку, в которой содержатся следующие сведения:

- метод HTTP;
- адрес ресурса;
- версия протокола.

Выглядеть она, например, может так:

```
GET /wiki/HTTP HTTP/2.0
```

Заголовок запроса представлен набором пар вида поле: значение. Существуют два обязательных поля заголовка, одно из которых – **Host**, а второе – **Connection**. Остальные поля необязательны. Заголовок может выглядеть так:

```
Host: ru.wikipedia.org
Connection: close
```

Поле **Host** указывает на доменное имя, которое интересует браузер. Поле **Connection**, установленное в значение close, означает, что соединение между клиентом и сервером не нужно

держать открытым. Среди других часто используемых заголовков запросов можно отметить следующие:

- Origin
- Accept
- Accept-Encoding
- Cookie
- Cache-Control
- Dnt и др.

Заголовок запроса завершается пустой строкой.

Тело запроса необязательно, в GET-запросах оно не используется. Тело запроса используется в POST-запросах, а также в других запросах. Оно может содержать, например, данные в формате JSON.

После того, как сервер получает отправленный клиентом запрос, он его обрабатывает и отправляем клиенту ответ. Ответ начинается с кода состояния и с соответствующего сообщения. Если запрос выполнен успешно, то начало ответа будет выглядеть так:

200 OK

В настоящее время выделено пять классов кодов состояния

Код	**Класс**	**Назначение**
1xx	Информационный (англ. **Informational**)	В HTTP/1.0 — сообщения с такими кодами должны игнорироваться. В HTTP/1.1 — клиент должен быть готов принять этот класс сообщений как обычный ответ, но ничего отправлять серверу не нужно. Сами сообщения от сервера содержат только стартовую строку ответа и, если требуется, несколько специфичных для ответа полей заголовка. Прокси-серверы подобные сообщения должны отправлять дальше от сервера к клиенту.
2xx	Успех (англ. **Success**)	Информирование о случаях успешного принятия и обработки запроса клиента. В зависимости от статуса, сервер может ещё передать заголовки и тело сообщения.

Код	Класс	Назначение
3xx	Перенаправление (англ. **Redirection**)	Сообщает клиенту, что для успешного выполнения операции необходимо сделать другой запрос (как правило по другому URI). Из данного класса пять кодов 301, 302, 303, 305 и 307 относятся непосредственно к перенаправлениям (редирект). Адрес, по которому клиенту следует произвести запрос, сервер указывает в заголовке Location. При этом допускается использование фрагментов в целевом URI.
4xx	Ошибка клиента (англ. **Client Error**)	Указание ошибок со стороны клиента. При использовании всех методов, кроме HEAD, сервер должен вернуть в теле сообщения гипертекстовое пояснение для пользователя.
5xx	Ошибка сервера (англ. **Server Error**)	Информирование о случаях неудачного выполнения операции по вине сервера. Для всех ситуаций, кроме использования метода HEAD, сервер должен включать в тело сообщения объяснение, которое клиент отобразит пользователю.

Далее в ответе содержится список HTTP-заголовков и тело ответа (которое, так как запрос выполняет браузер, будет представлять собой HTML-код).

Вышеописанный процесс запроса и получения данных используется не только для HTML-кода, но и для любых других объектов, передаваемых с сервера в браузер с использованием протокола HTTP.

Работа с модулем HTTP/HTTPS

При разработке web-приложения обычно запускают локальный web-сервер, настроенный на http://localhost:8000/. Однако, браузеры по-разному обрабатывают HTTP- и HTTPS-запросы. На HTTPS-странице попытка загрузить Javascript по HTTP-протоколу будет заблокирована. Поэтому, при локальной

разработке, используя HTTP, скрипты будут загружаться нормально, но после выкладки на рабочие HTTPS-серверы возникнут проблемы. Чтобы избежать такой ситуации, нужно настроить доступ по HTTPS на локальном web-сервере.

Для выполнения HTTP-запросов средствами Node.js используется модуль **http** или **https**.

Модуль **http** позволяет нам создать свой собственный web-сервер и обрабатывать запросы. В качестве потока чтения используется запросы, а поток записи – ответы сервера.

Чтобы создать сервер, следует вызвать метод **http.createServer()**. Этот метод возвращает объект **http.Server**. Но чтобы сервер мог прослушивать и обрабатывать входящие подключения, у объекта сервера необходимо вызвать метод **listen()**, в который в качестве параметра передается номер порта, по которому запускается сервер.

Для обработки подключений в метод createServer можно передать специальную функцию, которая принимает два параметра: **request** - хранит информацию о запросе, **response** - управляет отправкой ответа:

```
// файл webServer.js
const http = require("http");

const server = http.createServer((request, response) => {

    console.log("Url: " + request.url);
    console.log("Method: " + request.method);
    console.log("User-Agent: " + request.headers["user-agent"]);
    console.log("All headers");
    console.log(request.headers);

    //create response
    response.statusCode = 200
    response.setHeader('Content-Type', 'text/html; charset=utf-8;')
    response.write("<h2>Answer from server</h2>");
    response.end();
});

server.listen(3000, 'localhost', () => {
```

```
    console.log('Server are listening!!!');
});

// работа сервера в консоли
node webServer.js
Server are listening!!!
Url: /
Method: GET
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 YaBrowser/25.12.0.0 Safari/537.36
All headers
{
  host: 'localhost:3000',
  connection: 'keep-alive',
  'sec-ch-ua': '"Chromium";v="142", "YaBrowser";v="25.12", "Not_A Brand";v="99", "Yowser";v="2.5"',
  'sec-ch-ua-mobile': '?0',
  'sec-ch-ua-platform': '"Windows"',
  'upgrade-insecure-requests': '1',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 YaBrowser/25.12.0.0 Safari/537.36',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
  'sec-fetch-site': 'none',
  'sec-fetch-mode': 'navigate',
  'sec-fetch-user': '?1',
  'sec-fetch-dest': 'document',
  'accept-encoding': 'gzip, deflate, br, zstd',
  'accept-language': 'ru,en;q=0.9'
}
```

Для отправки запросов используется метод http.get(), который принимает url-ресурса, и метод http.request(), который позволяет настроить параметры запроса такие как hostname, port, method, headers и др.

Рассмотрим пример отправки запроса на фейковый сервер https://jsonplaceholder.typicode.com:

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

// Get request
let req = https.get('https://jsonplaceholder.typicode.com/todos/1',
  res => {
   console.log('Get request');
   console.log(`statusCode: ${res.statusCode}`);
   res.on('data', data => {
     process.stdout.write(`${data}\n`);
   })
  }
);

// Common request
const options = {
 hostname: 'jsonplaceholder.typicode.com',
 port: 443,
 path: 'todos/1',
 method: 'GET'
}

req = https.request(options,
 res => {
   console.log('Common request');
   console.log(`statusCode: ${res.statusCode}`);
   res.on('data', data => {
     process.stdout.write(`${data}\n`);
   })
 }
)

req.on('error', error => {
 console.error(error);
})

req.end();
```

```
// результат работы
node request.js
Get request
statusCode: 200
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}

Common request
statusCode: 200
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
```

При отправке запросов на добавление и обновление данных формируется строка с данными в формате JSON, которая отправляется на сервер методом http.write(), а в объекте options метода http.request добавляется свойство headers с указанием типа и размера данных:

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

// Post request
const postData = JSON.stringify({
  userId: 100,
  id: 100,
  title: 'some task',
  completed: true
})

const options = {
  hostname: 'jsonplaceholder.typicode.com',
```

```
  port: 443,
  path: 'todos/1',
  method: 'POST',
  headers: {
    'Content-type': 'application/json',
    'Content-Length': Buffer.byteLength(postData),
  }
}

req = https.request(options,
  res => {
    console.log('Post request');
    console.log(`statusCode: ${res.statusCode}`);
    res.on('data', data => {
      process.stdout.write(`${data}\n`);
    })
  }
)
req.write(postData);

req.on('error', error => {
  console.error(error);
})

req.end();
```

Выполнение PUT-запросов и DELETE-запросов выглядит так же, как и выполнение POST-запросов. Главное отличие, помимо смыслового наполнения таких операций, заключается в значении свойства **method** объекта **options**.

Простой веб-сервер

Для простого веб-сервера, который отдаёт страницы сайта по запросу пользователя необходимо немного переработать код обычного http-сервера, добавив возможность считывать и передавть содержимое файлов сайта:

```
// newsite.js
```

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

http.createServer(function (request, response) {

    console.log(`Запрошенный адрес: ${request.url}`);

    // определяем какой ресурс необходимо отдать
    let filePath = "." + request.url;
    if (filePath == "./") {
      filePath = "./index.html";
    }

    // определяем расширения файла
    let extname = String(path.extname(filePath)).toLowerCase();

    // определяем mimeType по полученному расширению
    const mimeTypes = {
      ".html": "text/html",
      ".js": "text/javascript",
      ".css": "text/css",
      ".json": "application/json",
      ".png": "image/png",
      ".jpg": "image/jpg",
      ".gif": "image/gif",
      ".svg": "image/svg+xml",
      ".wav": "audio/wav",
      ".mp4": "video/mp4",
      ".woff": "application/font-woff",
      ".ttf": "application/font-ttf",
      ".eot": "application/vnd.ms-fontobject",
      ".otf": "application/font-otf",
      ".wasm": "application/wasm",
    };
    let contentType = mimeTypes[extname] || "application/octet-
stream";

    // формируем ответ
```

```
        fs.readFile(filePath, function (error, content) {
          if (error) {
            if (error.code == "ENOENT") {
              fs.readFile("./404.html", function (error, content) {
                response.writeHead(404,  {  "Content-Type":  "text/html"
});
                response.end(content, "utf-8");
              });
            } else {
              response.writeHead(500);
              response.end(
                "Sorry, check with the site admin for error: " +
                  error.code +
                  " ..\n",
              );
            }
          } else {
            response.writeHead(200, { "Content-Type": contentType });
            response.end(content, "utf-8");
          }
        });
      })
      .listen(3000, function () {
        console.log("Server started at 3000");
      });

      // результат работы
      node newsite.js
      Server started at 3000
      Запрошенный адрес: /
      Запрошенный адрес: /example2.html
```

В данном примере используется модуль **path**, который предоставляет утилиты для работы с путями к файлам и каталогам.

Взаимодействие с базами данными

Node.js позволяет взаимодействовать с различными системами управления базами данных (СУБД). Взаимодействие

осуществляется с помощью драйвера базы данных, который необходимо добавить в проект как зависимость. Драйвер предлагает набор методов для получения, добавления, модификации и удаления данных. В каждом драйвере эти методы именуются по разному, но все они имеют смысловую нагрузку и по названию можно понять их назначения.

Все взаимодействие сводится к установлению соединения и выполнения операций с данными.

В данном разделе рассмотрим взаимодействие с классической реляционной СУБД **MySQL**.

Для работы с сервером MySQL используются драйверы **mysql** и **mysql2**. Использование mysql2 предпочтительнее, так как он показывает лучшую производительность.

Для установки зависимости необходимо установить пакет:

```
npm install mysql2
```

Для создания подключения применяется метод createConnection(), который в качестве параметра принимает настройки подключения и возвращает объект, представляющий подключение.

```
const mysql = require("mysql2");

const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  database: "имя_базы_данных",
  password: "пароль_от_сервера"
});
```

Настройки подключения включают следующие параметры:

host − хост, на котором запущен сервер mysql;

user − пользователь MySQL, который используется для подключения;

database − имя базы данных, к которой идет подключение;

password − пароль для пользователя MySQL.

Для установки подключения используется метод **connect()** объекта **connection**, который принимают в качестве параметра функцию обратного вызова, в которой обрабатывается ошибки подключения:

```
connection.connect(function(err){
    if (err) {
      return console.error("Ошибка: " + err.message);
    }
    else{
      console.log("Подключение к серверу MySQL успешно установлено");
    }
 });
```

Для выполнения запросов у объекта подключения применяется метод **query()**, который первым параметром принимает строку с sql-командой, а вторым − функцию обратного вызова, через параметры которой мы можем получить результаты выполнения sql-команды или возникшую ошибку:

```
connection.query("SELECT * FROM users",
  function(err, results, fields) {
    console.log(err);
    console.log(results); // собственно данные
    console.log(fields); // мета-данные полей
});
```

В нашем примере осуществляется выборка всех данных из таблицы **"users"**. Функция обратного вызова принимает три параметра: первый параметр передаёт ошибку, если она возникла при выполнении запроса, второй параметр представляет массива данных, третий параметр хранит метаданные полей таблицы и дополнительную служебную информацию.

Все операции с данными опираются на команды языка SQL. Если в запрос необходимо передать данные рекомендуется использовать параметризацию. При параметризации вместо конкретных данных в тексте запроса ставятся плейсхолдеры − знаки вопроса, вместо которых при выполнении запроса будут

подставляться данные. Например, ниже приведён пример, демонстрирующий добавление как единичных данных так и множество значений:

```
const user = ["Nick", 20];
const sql = "INSERT INTO users(name, age) VALUES(?, ?)";
connection.query(sql, user, function(err, results) {
   if(err) console.log(err);
   else console.log("Данные добавлены");
});
const users = [
  ["Alex", 21],
  ["Max", 22],
  ["Ivan", 23]
];
const sql = `INSERT INTO users(name, age) VALUES ?`;
connection.query(sql, [users], function(err, results) {
   if(err) console.log(err);
   console.log(results);
});
```

После выполнения операций с данными необходимо закрыть подключение, вызвав метод **end()** объекта connection.

Аналогичный способы предоставляют доайверы для других СУБД.

Для работы с базами данных SQL можно использовать используется модуль sequelize. Он поддерживает следующие СУБД: MySQL, PostgreSQL, MSSQL и MariaDB. Модуль предоставляет единое API для всех перечисленных СУБД, так как все они используют единый язык описания запросов - SQL. Изучение ORM sequelize перенесено на самостоятельное изучение слушателей.

5 Фреймворк Express

5.1 Разработка веб-сервера

Для упрощения и ускорения написания полноценного веб-сервера используют фреймворк **Express**. Express предоставляет объект-сервер, принимающий запросы пользователя и формирующий ответ сервера. Он имеет готовые функции обработки http запросов, причём для каждого http метода имеется своя функция. Метод **use()** используется для создания промежуточных обработчиков − **Middleware**.

Когда фреймворк Express получает запрос, этот запрос передаётся в конвейер обработки. Конвейер состоит из набора компонентов или middleware, которые получают данные запроса и решают, как его обрабатывать.

```
// подключение express
const express = require("express");
// создаем объект приложения
const app = express();
// определяем обработчик для маршрута "/"
app.get("/", function(request, response){

    // отправляем ответ
    response.send("<h2>Привет Express!</h2>");
});
// начинаем прослушивать подключения на 3000 порту
app.listen(3000);
```

В данном примере используется функция **app.get()**. Она обрабатывает GET-запросы протокола HTTP и позволяет связать маршруты с определёнными обработчиками. Для этого первым параметром передаётся маршрут, а вторым – обработчик, который будет вызываться, если запрос к серверу соответствует данному маршруту.

Рассмотрим пример использования фреймворка для создания статического сайта. Пользователь запрашивает соответствующую

страницу сайта, сервер должен прочитать содержимое страницы и предоставить пользователю ответ в виде html-разметки с формированием статус-кода ответа. Для отправки содержимого файла используется функция **sendFile()**, а для отправки статус-кода − **sendStatus()**. Для поиска и чтения файла используется абсолютный путь, поэтому в проекте необходимо использовать объект **__dirname**.

Функция Express **static()** позволяет сразу передавать по запросу находящиеся у него статические файлы (изображения, аудио, HTML, CSS, JS). Она используется как Middleware с использованием метода use(). В качестве параметра функция принимает имя директории, в которой находятся все статические файлы:

```
// импортируем фреймворк
const express = require("express");

// создаем объект приложения
const app = express();
// используем middlewire для обработки запросов
app.use(express.static(__dirname + "/public"));
// прослушиваем порт 3000
app.listen(3000);
```

Как видим, использование фреймворка значительно сокращает время написания кода, за счёт встроенных методов.

5.2 Разработка службы RestAPI

Используя фреймворк Express и Node.js, мы можем реализовать полноценный webAPI в стиле REST для взаимодействия с пользователем. Архитектура REST предполагает применение следующих методов HTTP для взаимодействия с сервером: GET, POST, PUT, DELETE.

Express предлагает для обработки каждого метода свою собственную функцию. Поэтому задача разработки сводится к реализации этих функций.

Для лучшего понимания работы воспользуемся локальным файлом с набором данных в формате json. Задача нашей службы −

обработать все запросы пользователя на получение, добавление, обновление и удаление данных.

Примерный каркас приложения приведён ниже:

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

const app = express();
const jsonParser = express.json();

app.use(express.static(__dirname + "/public"));

//файл данных
const filePath = "users.json";

// получение всех пользователей
app.get("/api/users", function(req, res){

    const content = fs.readFileSync(filePath,"utf8");
    const users = JSON.parse(content);
    res.status(200).send(users);
});

// получение одного пользователя по id
app.get("/api/users/:id", function(req, res){

    const id = req.params.id; // получаем id
    const content = fs.readFileSync(filePath, "utf8");
    const users = JSON.parse(content);
    let user = null;
    // находим в массиве пользователя по id
    user = users.find(elem => elem.id == id);
    // отправляем пользователя
    if(user){
        res.status(200).send(user);
    }
    else{
        res.status(404).send();
    }
```

```
});

// добавление данных
app.post("/api/users", jsonParser, function (req, res) {

    if(!req.body) return res.sendStatus(400);

    const userName = req.body.name;
    const userAge = req.body.age;
    let user = {name: userName, age: userAge};

    let data = fs.readFileSync(filePath, "utf8");
    let users = JSON.parse(data);

    // находим максимальный id
    const id = Math.max.apply(Math,users.map(function(o){return
o.id;}))
    // увеличиваем его на единицу
    user.id = id+1;
    // добавляем пользователя в массив
    users.push(user);
    data = JSON.stringify(users);
    // перезаписываем файл с новыми данными
    fs.writeFileSync("users.json", data);
    res.status(201).send(user);});

 // удаление пользователя по id
app.delete("/api/users/:id", function(req, res){

    const id = req.params.id;
    let data = fs.readFileSync(filePath, "utf8");
    let users = JSON.parse(data);
    let index = -1;
    // находим индекс пользователя в массиве
    for(let i=0; i < users.length; i++){
        if(users[i].id==id){
            index=i;
            break;
        }
```

```
    }
    if(index > -1){
        // удаляем пользователя из массива по индексу
        const user = users.splice(index, 1)[0];
        data = JSON.stringify(users);
        fs.writeFileSync("users.json", data);
        // отправляем удаленного пользователя
        res.status(200).send(user);
    }
    else{
        res.status(404).send();
    }
});

// обновление пользователя
app.put("/api/users", jsonParser, function(req, res){

    if(!req.body) return res.sendStatus(400);

    const userId = req.body.id;
    const userName = req.body.name;
    const userAge = req.body.age;

    let data = fs.readFileSync(filePath, "utf8");
    const users = JSON.parse(data);
    let user;
    for(let i=0; i<users.length; i++){
        if(users[i].id==userId){
            user = users[i];
            break;
        }
    }
    // изменяем данные у пользователя
    if(user){
        user.age = userAge;
        user.name = userName;
        data = JSON.stringify(users);
        fs.writeFileSync("users.json", data);
        res.status(201).send(user);
```

```
    }
    else{
        res.status(404).send(user);
    }
});

app.listen(3000, function(){
    console.log("Сервер ожидает подключения...");
});
```

5.3 Авторизация и аутентификация

В Node.js аутентификация чаще всего реализуется с использованием функций промежуточной обработки, предоставляемых библиотекой **passport.js**, которая устанавливается с помощью npm.

Библиотека предоставляет несколько стратегий авторизации. Например при использовании стратегии **local** учетные данные, используемые для аутентификации пользователя передаются только во время авторизации. Если все в порядке, и пользователь существует, то информация о нем сохраняется в сессию, а идентификатор сессии, в свою очередь, сохраняется в **cookies** браузера. Каждый последующий запрос будет содержать cookies, с помощью которого passport сможет опознать пользователя, и достать его данные из сессии.

В качестве более безопасной и простой альтернативы сессиям и файлам cookie для авторизации в современных приложениях используется **JWT (JSON Web Token)**, который представляет собой строку, полученную на основе формата json. JWT позволяет уйти от хранения данных авторизованного пользователя на сервере и возлагает на сервер только задачу по верификации подписи.

JWT формируют три части:

− заголовок (**header**);

− данные (**payload**);

− подпись (**signature**).

Заголовок представляет собой объект JSON и описывает сам токен с помощью следующих свойств:

alg − алгоритм шифрования, используемый для подписи JWT, если токен не подписывается, то значением должно быть none (обязательный параметр);

typ − тип токена, необходимо указывать со значением "JWT", если могут использоваться токены другого типа (необязательный параметр);

ctp − тип данных, необходимо указывать со значением "JWT", если в payload присутствуют пользовательские ключи.

В данных, которые также передаются объектом JSON, указывается необходимая информация о пользователе. Также возможно задание значений предопределённых ключей (все они не обязательны) для описания конфигурации токена:

iss − приложение, создавшее токен;

sub − назначение JWT;

aud − массив получателей токена;

exp − дата и время, указанное в миллисекундах, прошедших с 01.01.1970, до наступления которого JWT будет валиден;

nbf − дата и время, указанное в миллисекундах, прошедших с 01.01.1970, до наступления которого JWT будет не валиден;

iat − дата и время создания JWT, указанное в миллисекундах, прошедших с 01.01.1970;

jti − уникальный идентификатор токена.

Заголовок и данные используются для вычисления значения подписи по указанному в заголовке в свойстве alg алгоритму шифрования.

Далее формируется сам JWT.

```
`${base64url(header)}.${base64url(payload)}.${signature}`
```

Сгенерированный JWT отправляется клиенту, где он сохраняется в **localStorage** или **sessionStorage**, и будет отправляться клиентом серверу при каждом HTTP-запросе в заголовке **Authorization**.

Сервер в свою очередь при обращении к маршрутам, требующих авторизации, извлекает данные из токена и проверяет валидность токена и наличие указанного в JWT пользователя.

Для экономии времени и избежания реализации собственного алгоритма формирования в Node.js JWT используется модуль **jsonwebtoken**.

Для создания JWT используется метод **sighn()** объекта jsonwebtoken. А для проверки — функция **verify()**:

```
const express = require('express');
const jwt = require('jsonwebtoken');
const bodyParser = require('body-parser');
const app = express();

app.use(bodyParser.json());

const JWT_SECRET = 'your-jwt-secret-key';

// Sample user database
const users = [
  { id: 1, username: 'user1', password: 'password1', role: 'user' }
];

// Login route - generate token
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Find user
  const user = users.find(u => u.username === username && u.password === password);

  if (!user) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  // Create payload for JWT
  const payload = {
    id: user.id,
    username: user.username,
    role: user.role
  };

  // Sign token
  const token = jwt.sign(payload, JWT_SECRET, { expiresIn: '1h'
});
```

```
  res.json({ message: 'Login successful', token });
 });

 // Middleware for JWT verification
 const authenticateJWT = (req, res, next) => {
  // Get auth header - The Authorization header is commonly used
to send authentication tokens
  const authHeader = req.headers.authorization;

  if (!authHeader) {
   return res.status(401).json({ message: 'Authorization header
missing' });
  }

  // Extract token from "Bearer <token>"
  const token = authHeader.split(' ')[1];

  if (!token) {
   return res.status(401).json({ message: 'Token missing' });
  }

  try {
   // Verify token
   const decoded = jwt.verify(token, JWT_SECRET);

   // Attach user to request
   req.user = decoded;

   next();
  } catch (error) {
   return res.status(403).json({ message: 'Invalid or expired token'
});
  }
 };

 // Protected route
 app.get('/profile', authenticateJWT, (req, res) => {
  res.json({ message: 'Profile accessed', user: req.user });
```

```
});

// Role-based route
app.get('/admin', authenticateJWT, (req, res) => {
  // Check if user has admin role
  if (req.user.role !== 'admin') {
    return res.status(403).json({ message: 'Access denied: admin role required' });
  }

  res.json({ message: 'Admin panel accessed' });
});

// Start server
app.listen(8080, () => {
  console.log('Server running on port 8080');
});
```

Список использованных источников

1. Флэнаган, Д. JavaScript. Подробное руководство / Дэвид Флэнаган ; пер. с англ. А. Киселева. - 6-е изд.. - Санкт-Петербург : Москва : Символ-Плюс, 2017. - 1080 с.
2. Васильев, А. Н. Программирование на Java Script в примерах и задачах / Васильев А. Н.. - Москва : Издательство "Э", 2017. - 718 с.
3. Ajax = Айякс : JavaScript, JQuery, Prototype, Dojo, DWR, Google Web ToolKit, Yahoo UI Library, Maps, ASP.NET Ajax, JSF, SOAP, WSDL, RSS and Atom, Comet, Ruby on Rails, XML, XSLT. - India : Dreamlech Press, 2010. - 756 p.
4. Современный учебник JavaScript. Режим доступа: https://learn.javascript.ru/. Дата доступа: 14.12.2025.
5. Документация по Node.js. Режим доступа: https://nodejs.org/docs/latest/api/. Дата доступа: 14.12.2025.
6. Документация по Express. Режим доступа: https://expressjs.com/. Дата доступа: 14.12.2025.

ВЕБ-ТЕХНОЛОГИИ

Пособие

по одноименной дисциплине для слушателей специальности переподготовки 9-09-0612-02 «Программное обеспечение информационных систем» заочной формы обучения

Составитель **Самовендюк** Николай Владимирович

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 13.02.26.
Рег. № 9Е.
http://www.gstu.by