Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

ПРОГРАММНЫЕ ИННОВАЦИОННЫЕ ПЛАТФОРМЫ ИНФОРМАЦИОННЫХ СИСТЕМ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-39 80 03 «Электронные системы и технологии»

Минск БГУИР 2024

УДК 004.4`2(076.5)
ББК 32.973я73
П78

А в т о р ы:

В. Ф. Алексеев, Д. В. Лихачевский, Г. А. Пискун, В. В. Шаталова

Р е ц е н з е н т ы:

кафедра программного обеспечения информационных систем и технологий
Белорусского национального технического университета
(протокол № 7 от 22.02.2023);

директор ГЦ «Белмикроанализ» ОАО «ИНТЕГРАЛ» –
управляющая компания холдинга ОАО «ИНТЕГРАЛ»
кандидат физико-математических наук А. Н. Петлицкий

П78 **Программные** инновационные платформы информационных систем. Лабораторный практикум : пособие / В. Ф. Алексеев [и др.]. – Минск : БГУИР, 2024. – 159 с. : ил.
ISBN 978-985-543-757-5.

Приводится описание лабораторных работ на языке JavaScript для Node.js – программной платформы, основанной на движке V8, компилирующем JavaScript в машинный код и превращающем его из узкоспециализированного языка в язык общего назначения.

Предназначено для студентов второй ступени высшего образования технических университетов. Может быть использовано аспирантами и инженерами, занимающимися вопросами проектирования информационных систем.

УДК 004.4`2(076.5)
ББК 32.973я73

ISBN 978-985-543-757-5

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Информационная система – взаимосвязанная совокупность концепций, методов, технологий, технических и программных средств, используемых для сбора, обработки, хранения и выдачи информации потребителю в интересах достижения поставленной цели.

Современное понимание информационной системы предполагает использование компьютера в качестве основного технического средства для поиска и переработки информации.

Информационные системы обеспечивают процессы по сбору, хранению, обработке, поиску, выдаче информации, необходимой для решения задач в любой области. Они помогают анализировать проблемы, осуществлять стратегическое планирование и создавать новые продукты.

Подготовка современного специалиста требует профессионального владения навыками, необходимыми для использования возможностей, предоставляемых компьютерными технологиями.

Программные инновационные платформы информационных систем изучаются магистрантами специальности 1-39 80 03 «Электронные системы и технологии».

Изучение указанной дисциплины обеспечивает подготовку специалиста, владеющего фундаментальными знаниями и практическими навыками в области программирования, развитие навыков, необходимых при разработке программных приложений для электронных систем.

Язык JavaScript – это мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией спецификации *ECMAScript*. *JavaScript* обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

В пособии рассматриваются четыре лабораторные работы, позволяющие получить навыки и в дальнейшем применить на практике знания по использованию языка *JavaScript*.

Node, или *Node.js* – это программная платформа с открытым исходным кодом, основанная на движке *V*8, превращающая *JavaScript* из узкоспециализированного языка в язык общего назначения. Она позволяет писать серверный код для веб-приложений и динамических веб-страниц, а также программ командной строки. В основе платформы лежит событийно-управляемая модель с неблокирующими операциями ввода-вывода, что делает ее эффективной и легкой.

Node.js добавляет возможность *JavaScript* взаимодействовать с устройствами ввода-вывода через свой *API*, написанный на *C++*, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы их из *JavaScript*-кода.

Node.js лежит в основе *Internet of Things*, или просто *IoT*. Платформа помогает управлять приборами и создавать серверы, способные одновременно обрабатывать большое количество запросов.

В *Node.js* выполняется код, который написан на *JavaScript*. Это означает, что *frontend*-разработчики, которые уже используют *JavaScript* в браузере, могут писать и клиентский, и серверный код на привычном языке программирования, не изучая инструмент с нуля. В *Node.js* можно быстро переходить на новые стандарты *ECMAScript* по мере их реализации. Новые возможности языка становятся доступны сразу после установки поддерживающей их версии *Node.js*.

Именно по этим причинам в пособии представлено шесть лабораторных работ, что позволяет расширить представление обучающихся о современных технологиях программирования.

Авторы рекомендуют магистрантам после изучения теоретической части обязательно выполнить все листинги кодов и лишь затем приступить к практической части заданий.

Авторы выражают благодарность рецензентам, а также коллегам кафедры ПИКС за ценные замечания по улучшению структуры и содержания пособия.

Пособие предназначено для студентов второй ступени (магистратуры) высшего образования технических университетов. Может быть использовано аспирантами и инженерами, занимающимися вопросами проектирования информационных систем.

ЛАБОРАТОРНАЯ РАБОТА № 1

ОБЪЕКТЫ СТРАНИЦЫ

Цель работы

Ознакомиться с возможностями объектов страницы.

Теоретические сведения

Сам по себе язык *JavaScript* не предусматривает работы с браузером. Он вообще не знает об *HTML*. Но позволяет легко расширять себя новыми функциями и объектами.

На рисунке 1.1 схематически отображена структура, которая получается, если посмотреть на совокупность браузерных объектов.

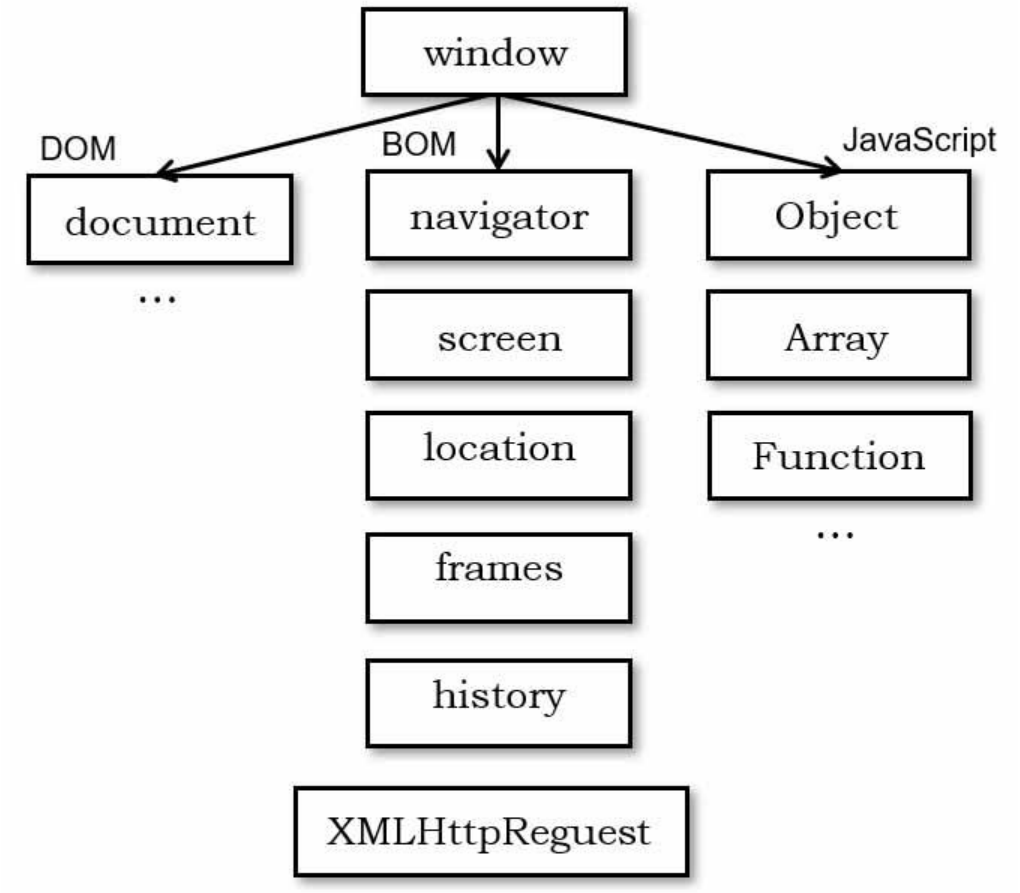


Рисунок 1.1 – Структура браузерных объектов

Как видно из рисунка 1.1, на вершине стоит `window`.

У этого объекта двоякая позиция: он, с одной стороны, является глобальным объектом в *JavaScript*, с другой – содержит свойства и методы для управления окном браузера, открытия новых окон, как показано в листинге 1.1.

Листинг 1.1 – Пример открытия нового окна

```
window.open('https://bsuir.by/');
```

Глобальный объект `document` дает возможность взаимодействовать с содержимым страницы, как показано в листинге 1.2.

Листинг 1.2 – Взаимодействие с содержимым страницы

```
document.body.style.background = 'red';
alert( 'Элемент BODY стал красным, а сейчас обратно вернется' );
document.body.style.background = '';
```

BOM (*Browser Object Model*) – объектная модель браузера – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Объект `navigator` содержит общую информацию о браузере и операционной системе. Особенно примечательны два свойства: `navigator.userAgent` (содержит информацию о браузере) и `navigator.platform` (содержит информацию о платформе, позволяет различать *Windows/Linux/Mac* и тому подобное).

Объект `location` содержит информацию о текущем *URL* страницы и позволяет перенаправить посетителя на новый *URL*.

Функции `alert/confirm/prompt` – тоже входят в *BOM*, как показано в листинге 1.3.

Листинг 1.3 – Функция `alert`

```
alert( location.href ); // выведет текущий адрес
```

Большинство возможностей *BOM* стандартизированы в *HTML5*, хотя различные браузеры и предоставляют зачастую что-то свое, в дополнение к стандарту.

Основным инструментом работы и динамических изменений на странице является *DOM* (*Document Object Model*) – объектная модель, используемая для *XML/HTML*-документов.

Согласно *DOM*-модели документ является иерархией, деревом. Каждый *HTML*-тег образует узел дерева с типом «элемент». Вложенные в него теги становятся дочерними узлами. Для представления текста создаются узлы с типом «текст».

DOM – это представление документа в виде дерева объектов, доступное для изменения через *JavaScript*, как показано в листинге 1.4.

Листинг 1.4 – Дерево объектов *DOM*

```
<!DOCTYPE HTML>
<html>
<head>
```

```
  <title>Тестовая страница</title>
</head>
<body>
  Здесь должен быть текст
</body>
</html>
```

Пробелы и переводы строки – это тоже текст, полноправные символы, которые учитываются в *DOM*. В частности, в примере выше тег `<html>` содержит не только узлы-элементы `<head>` и `<body>`, но и текст (пробелы, переводы строки) между ними.

Впрочем, как раз на самом верхнем уровне из этого правила есть исключения: пробелы до `<head>` по стандарту игнорируются, а любое содержимое после `</body>` не создает узла, так как браузер переносит его внутрь, в конец `body`.

В остальных случаях все стандартно – если пробелы есть в документе, то они есть и в *DOM*, а если их убрать, то и в *DOM* их не будет. При чтении неверного HTML браузер автоматически корректирует его для показа и при построении *DOM*. В частности, всегда будет верхний тег `<html>`. Даже если в тексте нет, в *DOM* он будет: браузер создаст его самостоятельно. То же самое касается и тега `<body>`.

Например, если файл состоит из одного слова «БГУИР», то браузер автоматически обернет его в `<html>` и `<body>`. При генерации *DOM* браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее.

DOM нужен для того, чтобы манипулировать страницей – читать информацию из *HTML*, создавать и изменять элементы. Узел *HTML* можно получить как `document.documentElement`, а `BODY` – как `document.body`. Получив узел, можно выполнить над ним некоторые действия. Например, можно поменять цвет `BODY` и вернуть обратно, как показано в листинге 1.5.

Листинг 1.5 – Изменение цвета `BODY`

```
document.body.style.backgroundColor = 'red';
alert( 'Поменяли цвет BODY' );
document.body.style.backgroundColor = '';
alert( 'Сбросили цвет BODY' );
```

DOM предоставляет возможность делать со страницей все что угодно.

Доступ к *DOM* начинается с объекта `document`. Из него можно добраться до любых узлов. Так выглядят основные ссылки, по которым можно переходить между узлами *DOM*, как показано на рисунке 1.2.

Самые верхние элементы дерева доступны напрямую из `document`, как показано в листинге 1.6.

Листинг 1.6 – Доступ к верхним элементам дерева

```
<HTML> = document.documentElement
```

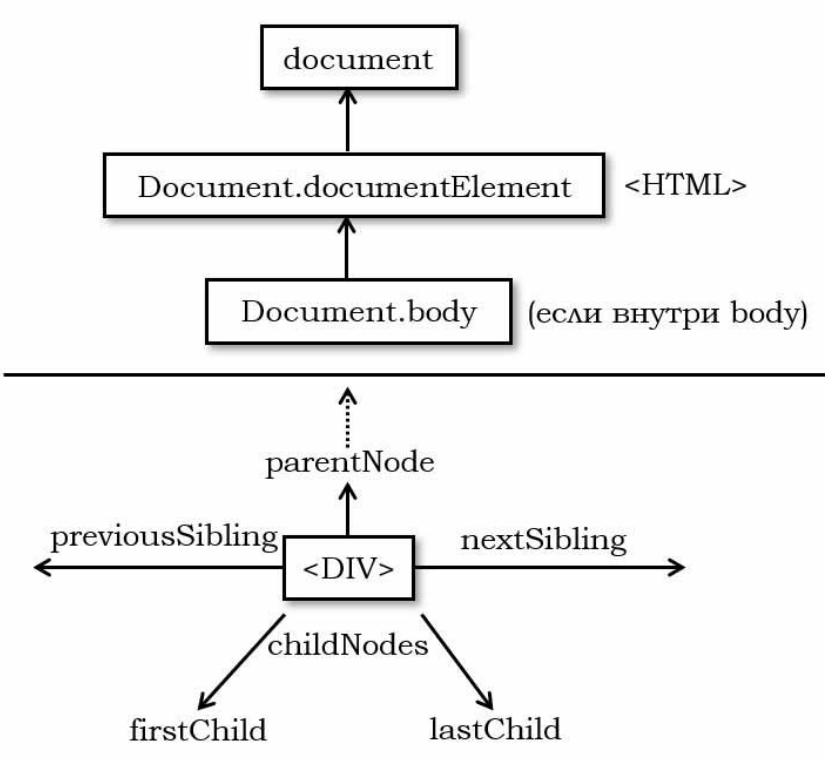


Рисунок 1.2 – Ссылки между узлами *DOM*

Первая точка входа – `document.documentElement`. Это свойство ссылается на *DOM*-объект для тега `<html>`, как показано в листинге 1.7.

Листинг 1.7 – Доступ к `BODY`

```
<BODY> = document.body
```

Вторая точка входа – `document.body`, который соответствует тегу `<body>`.

Здесь и далее будем использовать два принципиально разных термина.

Дочерние элементы (или дети) – элементы, которые лежат непосредственно внутри данного. Например, внутри `<HTML>` обычно лежат `<HEAD>` и `<BODY>`.

Потомки – все элементы, которые лежат внутри данного, вместе с их детьми, детьми их детей и так далее. То есть все поддерево *DOM*.

Псевдомассив `childNodes` хранит все дочерние элементы, включая текстовые.

Последовательно выведем дочерние элементы `document.body`, как показано в листинге 1.8.

Листинг 1.8 – Вывод дочерних элементов `document.body`

```
<!DOCTYPE HTML>
<html>
<body>
```

```
      <div>Начало</div>
      <ul>
        <li>Информация</li>
      </ul>
      <div>Конец</div>
      <script>
        for (var i = 0; i < document.body.child-
Nodes.length; i++) {
          alert( document.body.childNodes[i] ); // Text,
DIV, Text, UL, ..., SCRIPT
        }
      </script>
      ...
    </body>
    </html>
```

Следует обратить внимание на следующую деталь. Если запустить пример выше, то последним будет выведен элемент `<script>`. На самом деле в документе есть еще текст (обозначенный многоточием), но на момент выполнения скрипта браузер еще до него не дошел.

Пробельный узел будет в итоговом документе, но его еще нет на этапе выполнения скрипта.

Свойства `firstChild` и `lastChild` обеспечивают быстрый доступ к первому и последнему элементам.

Доступ к элементам слева и справа данного можно получить по ссылкам `previousSibling/nextSibling`.

Родитель доступен через `parentNode`. Если долго идти от одного элемента к другому, то рано или поздно дойдешь до корня *DOM*, то есть до `document.documentElement`, а затем и `document`.

Если элементу назначен специальный атрибут `id`, то можно получить его прямо по переменной с именем из значения `id`, как показано в листинге 1.9.

Листинг 1.9 – Получение атрибута `id`

```
    <div id="content-holder">
      <div id="content">Элемент</div>
    </div>
<script>
      alert( content ); // DOM-элемент
      alert( window['content-holder'] ); // в имени де-
фис, поэтому через [...]
    </script>
```

Это поведение соответствует стандарту. Оно существует в первую очередь для совместимости и не очень приветствуется, поскольку использует глобальные переменные. Браузер пытается помочь, смешивая пространства имен *JS* и *DOM*, но при этом возможны конфликты.

Более правильной и общепринятой практикой является доступ к элементу вызовом `document.getElementById` («идентификатор»), как показано в листинге 1.10.

Листинг 1.10 – Доступ к элементу с использованием `getElementById`

```
<div id="content">Выделим этот элемент</div>
<script>
  var elem = document.getElementById('content');
  elem.style.background = 'red';
  alert( elem == content ); // true
  content.style.background = ""; // один и тот же
элемент
</script>
```

По стандарту значение `id` должно быть уникально, то есть в документе может быть только один элемент с данным `id`. И именно он будет возвращен. Если в документе есть несколько элементов с уникальным `id`, то поведение не определено. То есть нет гарантии, что браузер вернет именно первый или последний – вернет случайный элемент. Поэтому стараются следовать правилу уникальности `id`.

Метод `elem.getElementsByTagName(tag)` ищет все элементы с заданным тегом `tag` внутри элемента `elem` и возвращает их в виде списка, как показано в листинге 1.11. Регистр тега не имеет значения.

Листинг 1.11 – Поиск всех элементов с заданным тегом

```
// получить все div-элементы
var elements = document.getElementsByTagName('div');
```

В отличие от `getElementById`, который существует только в контексте `document`, метод `getElementsByTagName` может искать внутри любого элемента. Например, найдем все элементы `input` внутри таблицы, как показано в листинге 1.12.

Листинг 1.12 – Вывод всех элементов `input`

```
<table id="age-table">
  <tr>
    <td>Ваш возраст:</td>
    <td>
      <label>
```

```
            <input type="radio" name="age" value="young"
checked> младше 18
          </label>
          <label>
            <input type="radio" name="age" value="ma-
ture"> от 18 до 50
          </label>
          <label>
            <input type="radio" name="age" value="sen-
ior"> старше 60
          </label>
        </td>
      </tr>
    </table>
    <script>
      var tableElem = document.getElementById('age-
table');
      var elements = tableElem.getElementsByTagName('in-
put');
      for (var i = 0; i < elements.length; i++) {
        var input = elements[i];
        alert( input.value + ': ' + input.checked );
      }
    </script>
```

Можно получить всех потомков, передав звездочку `'*'` вместо тега, как показано в листинге 1.13.

Листинг 1.13 – Получение всех потомков

```
    // получить все элементы документа
    document.getElementsByTagName('*');
// получить всех потомков элемента elem:
    elem.getElementsByTagName('*');
```

Вызов `document.getElementsByName(name)` позволяет получить все элементы с данным атрибутом `name`. Например, все элементы с именем `age`, как показано в листинге 1.14.

Листинг 1.14 – Получение всех элементов с атрибутом `name`

```
var elems = document.getElementsByName('age');
```

До появления стандарта *HTML5* этот метод возвращал только те элементы, в которых предусмотрена поддержка атрибута `name`, в частности: `iframe`, `a`, `input` и другими. В современных браузерах тег не имеет значения.

Вызов `elem.getElementsByClassName(className)` возвращает коллекцию элементов с классом `className`. Находит элемент и в том случае, если у него несколько классов, а искомый – один из них, как показано в листинге 1.15. Данный вызов поддерживается всеми современными браузерами.

Листинг 1.15 – Получение коллекции элементов

```
 <div class="article">Статья</div>
<div class="long article">Длинная статья</div>
<script>
  var articles = document.getElementsByClassName('ar-
ticle');
  alert( articles.length ); // 2, найдет оба элемента
</script>
```

Как и `getElementsByTagName`, этот метод может быть вызван и в контексте *DOM*-элемента, и в контексте документа.

Вызов `elem.querySelectorAll(css)` возвращает все элементы внутри `elem`, удовлетворяющие *CSS*-селектору `css`. Это один из самых часто используемых и полезных методов при работе с *DOM*. Он есть во всех современных браузерах. Следующий запрос получает все элементы `LI`, которые являются последними потомками в `UL`, как показано в листинге 1.16.

Листинг 1.16 – Получение всех элементов `LI`

```
<ul>
  <li>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  var elements = document.querySelectorAll('ul >
li:last-child');
  for (var i = 0; i < elements.length; i++) {
    alert( elements[i].innerHTML ); // "тест",
"пройден"
  }
</script>
```

Вызов `elem.querySelector(css)` возвращает не все, а только первый элемент, соответствующий *CSS*-селектору `css`.

Иначе говоря, результат такой же, как и при `elem.querySelectorAll(css)[0]`, но в последнем вызове сначала ищутся

все элементы, а потом берется первый. В `elem.querySelector(css)` ищется только первый элемент, то есть он эффективнее.

Метод `elem.matches(css)` ничего не ищет, а проверяет, удовлетворяет ли `elem` селектору `css`. Он возвращает `true` либо `false`.

Этот метод бывает полезным, когда перебираются элементы в массиве или по обычным навигационным ссылкам и мы пытаемся отфильтровать те из них, которые нужны.

Ранее в спецификации он назывался `matchesSelector`, и большинство браузеров поддерживают его именно под этим старым именем либо с префиксами `ms/moz/webkit`, как показано в листинге 1.17.

Листинг 1.17 – Поддержка `matchesSelector`

```
<a href="https://example.com/file.zip">...</a>
<a href="https://bsuir.by/">...</a>
<script>
  var elems = document.body.children;
  for (var i = 0; i < elems.length; i++) {
    if (elems[i].matches('a[href$="zip"]')) {
      alert( "Ссылка на архив: " + elems[i].href );
    }
  }
</script>
```

Метод `elem.closest(css)` ищет ближайший элемент, более высокий по иерархии *DOM*, подходящий под *CSS*-селектор `css`. Сам элемент тоже включается в поиск. Иначе говоря, метод `closest` бежит от текущего элемента вверх по цепочке родителей и проверяет, подходит ли элемент под указанный *CSS*-селектор. Если подходит – останавливается и возвращает его, как показано в листинге 1.18.

Листинг 1.18 – Нахождение ближайшего элемента

```
<ul>
  <li class="chapter">Глава I
    <ul>
      <li class="subchapter">Глава <span
class="num">1.1</span></li>
      <li class="subchapter">Глава <span
class="num">1.2</span></li>
    </ul>
  </li>
</ul>

<script>
  var numberSpan = document.querySelector('.num');
```

```
      // ближайший элемент сверху, подходящий под селек-
тор li
      alert(numberSpan.closest('li').className)
       // subchapter
      // ближайший элемент сверху, подходящий под селек-
тор .chapter
      alert(numberSpan.closest('.chapter').tagName)
       // LI ближайший элемент сверху, подходящий под се-
лектор span
      // это сам numberSpan, так как поиск включает в
себя сам элемент
      alert(numberSpan.closest('span') === numberSpan)
    // true
    </script>
```

Этот метод часто используется, когда заведомо известно, что подходящий элемент только один, и хотим получить в переменную сразу именно его.

Порядок выполнения

1 Напишите код, который получит элемент `<HEAD>`.

2 Напишите код, который получит `<ul>`.

3 Напишите код, который получит второй `<li>`. Будет ли ваш код работать в *Firefox* и *Microsoft Edge*, если комментарий переместить между элементами `<li>`?

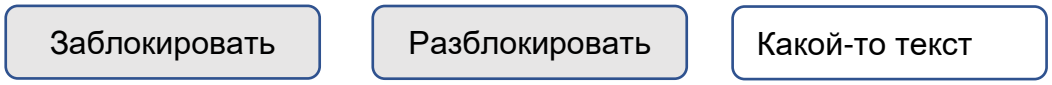
4 Есть длинный список `ul`:

```
<ul>
  <li>...</li>
  <li>...</li>
  <li>...</li>
  ...
</ul>
```

5 Наиболее эффективно получите второй `<li>` и опишите данный способ.

6 Создайте небольшой документ и выведите из него все возможные созданные элементы.

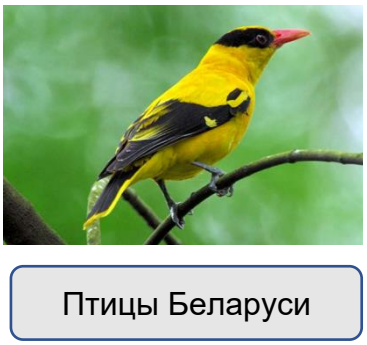
7 Повторите поведение кнопок по нажатию на них (одна из них блокирует инпут с помощью атрибута `disabled`, а другая – разблокировывает):



8 Повторите поведение кнопки по нажатию на нее (она меняет цвет текста в инпуте):

9 Повторите поведение кнопки по нажатию на нее (она выводит алертом содержимое инпута, возведенное в квадрат):

10 Создайте страницу по данному образцу:



Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Реализация решения задач с подробным рассмотрением (описанием) решений.
5 Выводы.
6 Список использованных источников.

Контрольные вопросы

1 Что включает в себя структура браузерных объектов?
2 Что из себя представляет объект `document`?
3 Что такое *BOM*?
4 Что такое *DOM*?
5 Что означает структура стандартного *DOM*?
6 Что из себя представляют дочерние элементы и потомки?
7 Какой будет результат, если попытаться получить доступ к элементу, которого еще не существует в момент выполнения скрипта?
8 Как извлечь значение атрибута?
9 Как извлечь все элементы тега?
10 Как извлечь всю коллекцию элементов класса?
11 Как извлечь элемент селектора?

Список использованных источников

1 Thomson, B. S. JavaScript. Notes for Professionals [Электронный ресурс]. – 2018. – Режим доступа: https://www.tradepub.com/free/w_goaa19/.

2 Скотт, А. Д. Разработка на JavaScript. Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron / А. Д. Скотт; пер. с англ. – СПб. : Питер, 2021. – 320 с.

3 Флэнаган, Д. JavaScript. Полное руководство / Д. Флэнаган. – 7-е изд. ; пер. с англ. – СПб. : ООО «Диалектика», 2021. – 720 с.

4 Фримен, Э. Изучаем программирование на JavaScript / Э. Фримен, Э. Робсон. – СПб. : Питер, 2015. – 640 с.

5 Хавербеке, М. Выразительный JavaScript. Современное веб-программирование / М. Хавербеке. – 3-е изд. – СПб. : Питер, 2019. – 480 с.

ЛАБОРАТОРНАЯ РАБОТА № 2

ЦИКЛЫ И УСЛОВИЯ

Цель работы

Ознакомиться с основами разработки на языке *JavaScript*.

Теоретические сведения

JavaScript – объектно-ориентированный язык, но используемая в нем объектная модель в корне отличается от модели, используемой в большинстве других языков.

JavaScript позволяет реализовать те функции страницы, которые невозможно реализовать стандартными тегами *HTML*. Сценарии запускаются в результате наступления какого-нибудь события. Например, пользователь нажал кнопку или изменился размер окна. *JavaScript* имеет доступ к свойствам документа и свойствам браузера. Например, на *JavaScript* можно легко изменять заголовок окна браузера или текст в строке состояния.

В *JavaScript* используется объектная модель документа, в рамках которой каждый *HTML*-контейнер можно рассматривать как совокупность свойств, методов и событий, происходящих в браузере. По сути, это связь между *HTML*-страницей и браузером.

Переменная (`variable`) – это, по существу, именованное значение, и, как подразумевает название, данное значение может изменяться в любое время. Например, если идет работа над системой контроля климата, могла бы использоваться переменная `currentVariable`, как показано в листинге 2.1.

Листинг 2.1 – Инициализация переменной

```
let currentVariable = 22;
```

Этот оператор выполняет два действия: объявляет (создает) переменную `currentVariable` и присваивает ей исходное значение. Объявляя переменную, необязательно назначать ей исходное значение.

Идентификатор – это уникальное имя предмета, позволяющее отличать его от других предметов.

Правила задания имен идентификаторов:

– имя идентификатора может состоять из любых цифр и букв английского алфавита;

– в имени идентификатора могут использоваться символы доллара «`$`» и подчеркивания «`_`»;

– имя идентификатора не может начинаться с цифры;

– в качестве имен идентификаторов нельзя / не рекомендуется использовать ключевые и зарезервированные слова *JavaScript*;

– имена идентификаторов регистрозависимы, это значит, что «Name» и «name» это разные идентификаторы.

Объект – это контейнер, и его содержимое может измениться со временем (это будет тот же объект с другим содержимым).

В *JavaScript* значения бывают или базовыми типами (primitive), или объектами (object). Базовые типы (такие как строки и числа) неизменны (immutast). Число 5 всегда будет числом 5; строка «alpha» всегда будет строкой «alpha». Для чисел это кажется очевидным, но не для строк: когда строки связывают вместе («alpha» + «omega»), они иногда выглядят, как та же строка, только измененная. Но это не так: получается новая строка, отличная от прежней, как число 6 отличается от числа 5. Существует шесть базовых типов:

– число (Number);
– строка (String);
– логический (Boolean);
– пусто (Null);
– неопределенный (Undefined);
– символ (Symbol).

Кроме этих шести базовых типов, существуют объекты (object). В отличие от базовых типов объекты способны принимать различные формы и значения, они подобны хамелеону.

Благодаря гибкости объекты применяются для создания специальных типов данных. Фактически *JavaScript* предоставляет несколько объектов встроенных типов. Рассмотрим следующие объекты встроенных типов:

– Array;
– Date;
– RegExp;
– Map и WeakMap;
– Set и WeakSet.

Объект можно воспринимать как совокупность данных и методов (функций) для их обработки. В *JavaScript* с некоторыми объектами также связываются определенные события. Представим, что объект – это человек. Пусть объект называется Human. У такого объекта может быть масса характеристик – имя, пол, дата рождения и так далее. Все это называется свойствами объекта. Обратиться к свойству можно, как показано в листинге 2.2.

Листинг 2.2 – Обращение к свойству объекта

```
Human.ObjectName
```

С объектом может быть связано какое-нибудь событие. Например, при рождении человека может генерироваться событие OnBirth. Для каждого события можно определить его обработчик – функцию, которая будет на него реа-

гировать. Что будет делать эта функция, зависит от события. Например, обработчик `OnBirth` может заносить в некую общую таблицу базы данных информацию об объекте – имя, пол и дату рождения. Такая таблица может использоваться для ускорения поиска нужного объекта.

При отладке, кроме просмотра переменных и передвижения по скрипту, бывает полезно запускать команды *JavaScript*. Для этого нужна консоль. В *Google Chrome* такая консоль называется «Инструменты разработчика». Данные в консоль записываются при помощи команды `console.log(…)`. Она пишет переданные ей аргументы в консоль, как показано в листинге 2.3.

Листинг 2.3 – Запись данных в консоль

```
// результат будет виден в консоли
for (var i = 0; i < 5; i++) {
  console.log("значение", i);
}
```

В *JavaScript* есть только один числовой тип данных. В большинстве языков есть несколько целочисленных типов и два или более типов с плавающей точкой. С одной стороны, это упрощает *JavaScript*, особенно для новичков. С другой стороны, это снижает пригодность *JavaScript* для определенных приложений, требующих эффективной целочисленной арифметики или чисел с фиксированной точностью.

Строки в *JavaScript* представлены в формате `Unicode`. Это индустриальный компьютерный стандарт для представления текстовых данных, включающий точки кода (`code point`) для каждой буквы или символа. Хотя `Unicode` способен представлять текст на любом языке, это не означает, что программное обеспечение, визуализирующее символы `Unicode`, будет способно правильно визуализировать каждую кодовую точку. В *JavaScript* строковые литералы представляются в одинарных кавычках, парных кавычках или обратных апострофах.

Переменная логического типа имеет только два возможных значения: `true` и `false`. Некоторые языки (например, *C*) используют вместо логических значений числа: `0` – это `false`, а любое другое число – `true`.

В *JavaScript* есть два специальных типа: `null` и `undefined`. У типа `null` есть только одно возможное значение (`null` – пусто), а у `undefined` – только `undefined` (неопределенно). Типы `null` и `undefined` представляют нечто, чего не существует, и даже факт наличия двух отдельных типов данных не прекратил недопонимания, особенно среди новичков.

Считается, что тип данных `null` предназначен для программистов, а тип `undefined` зарезервирован для самого *JavaScript*. Они означают, что некое значение еще отсутствует. Но это не обязательное правило: значение `undefined` вполне доступно для программистов, но здравый смысл требует использовать его чрезвычайно осторожно.

Массивы (`array`) в *JavaScript* – это объекты специального типа. В отличие от обычных объектов содержимое массива упорядочено (элемент `0` всегда следует перед элементом `1`), а ключи являются числовыми и последовательными.

Массивы в *JavaScript* обладают следующими свойствами:

– размер массива не ограничен, что означает возможность добавлять и удалять элементы в любое время;

– массивы не являются гомогенными;

– каждый индивидуальный элемент может иметь любой тип;

– элементы массива нумеруются от нуля.

Для литерала массива в *JavaScript* используются квадратные скобки, заключающие элементы массива, разделенные запятыми, как показано в листинге 2.4.

Листинг 2.4 – Инициализация массива

```
const massl [1, 2, 3, 4, 5];
const mass2 = [1,
'two',
3,
Null
];
```

Даты и время в *JavaScript* представляются встроенным объектом `Date`. Это один из наиболее проблематичных аспектов языка. Первоначально это была прямая связь с *Java* (одна из немногих областей, в которых у *JavaScript* фактически есть прямое отношение к *Java*); с объектом `Date` может быть сложно работать, особенно если проект имеет дело с датами в различных часовых поясах.

Существуют следующие базовые *UI*-операции, которые позволяют работать с данными, полученными от пользователя:

– `alert`;

– `prompt`;

– `confirm`.

Операция `alert` выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмет «*OK*», как показано в листинге 2.5.

Листинг 2.5 – Использование `alert`

```
alert(сообщение)
```

Окно сообщения, которое выводится, является модальным окном. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и тому подобное, пока не разберется с окном. В данном случае – пока не нажмет «*OK*».

Функция `prompt` принимает два аргумента, как показано в листинге 2.6.

Листинг 2.6 – Функция `prompt`

```
result = prompt(title, default);
```

Она выводит модальное окно с заголовком `title`, полем для ввода текста, заполненным строкой по умолчанию `default` и кнопками *OK/CANCEL*.

Пользователь должен либо что-то ввести и нажать «*OK*», либо отменить ввод кликом на «*CANCEL*» или нажатием «*Esc*» на клавиатуре.

Операция `confirm` выводит окно с вопросом `question` с двумя кнопками: *OK* и *CANCEL*, как показано в листинге 2.7.

Листинг 2.7 – Функция `confirm`

```
result = confirm(question);
```

Результатом будет `true` при нажатии «*OK*» и `false` – при «*CANCEL*» («*Esc*»).

Проверка пользователя, является ли он администратором, показана в листинге 2.8.

Листинг 2.8 – Реализация проверки на пользователя

```
let isAdmin = confirm("Вы - администратор?");
alert( isAdmin );
```

Конкретным местом, куда выводится модальное окно с вопросом, обычно является центр браузера; внешний вид окна выбирает браузер. Разработчик не может на это влиять. С одной стороны – это недостаток, так как нельзя вывести окно в своем, дизайне. С другой стороны, преимущество этих функций по сравнению с другими, более сложными методами взаимодействия, как раз в том, что они очень просты. Это самый простой способ вывести сообщение или получить информацию от посетителя. Поэтому их используют в тех случаях, когда простота важна, а дизайн особой роли не играет.

Цикл `while` повторяет код, пока выполняется его условие (`condition`), как показано в листинге 2.9.

Листинг 2.9 – Реализация цикла `while`

```
while (условие) {
  // код, тело цикла
}
```

В следующем примере цикл выводит переменную `i`, пока она не будет меньше трех, как показано в листинге 2.10.

Листинг 2.10 – Пример цикла `while`

```
var i = 0;
while (i < 3) {
```

```
  alert( i );
  i++;
}
```

На рисунке 2.1 представлен алгоритм работы цикла `while`.

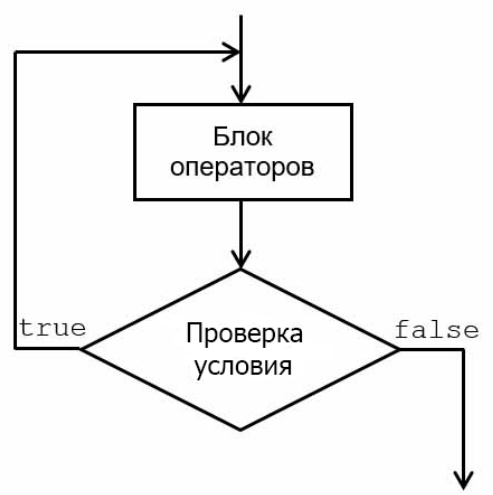


Рисунок 2.1 – Алгоритм работы цикла `while`

Цикл `for` чрезвычайно гибок (он может даже заменить цикл `while`), но он лучше всего подходит для случая, когда необходимо фиксированное количество циклов (особенно когда необходимо знать номер текущего цикла), как показано в листинге 2.11.

Листинг 2.11 – Реализация цикла `for`

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

В следующем примере выполняются три итерации (от 0 до 2 включительно) и результат выводится в `alert`, как показано в листинге 2.12.

Листинг 2.12 – Выполнение итераций цикла `for`

```
var i;
for (i = 0; i < 3; i++) {
  alert( i );
}
```

Алгоритм работы цикла `for` представлен на рисунке 2.2.

Есть четыре оператора, позволяющие изменить нормальный ход управления потоком:

– оператор `break` – немедленно прерывает цикл;

– оператор `continue` – переходит к следующему шагу в цикле;

– оператор `return` – осуществляет выход из текущей функции (независимо блока, в котором он находится);

– оператор `throw` – вызывает исключение в программе (`exception`), которое следует обработать обработчиком исключений (он может находиться за пределами текущего оператора управления потоком).

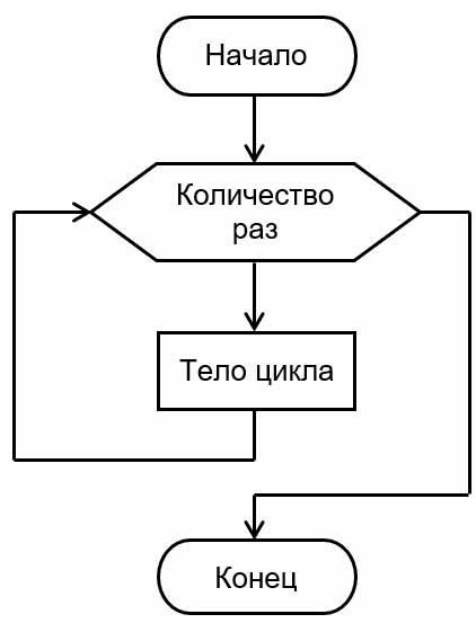


Рисунок 2.2 – Алгоритм работы цикла `for`

Для перебора всех свойств из объекта используется цикл по свойствам `for...in`. Эта синтаксическая конструкция отличается от рассмотренного ранее цикла `for(;;)`, как показано в листинге 2.13.

Листинг 2.13 – Реализация цикла `for...in`

```
for (key in obj) {
  /* ... делать что-то с obj[key] ... */
}
```

При этом `for...in` последовательно переберет свойства объекта `obj`, имя каждого свойства будет записано в `key` и вызвано тело цикла.

Иногда в зависимости от условия нужно выполнить различные действия. Для этого используется оператор `if`, как показано в листинге 2.14.

Листинг 2.14 – Реализация оператора `if`

```
if (условие) {
  // ... тело цикла ...
};
```

Оператор `if` («если») получает условие. Он вычисляет его, и если результат – `true`, то выполняет команду.

Проверку на истинность утверждения можно реализовать, как показано в листинге 2.15.

Листинг 2.15 – Проверка на верность утверждения

```
    var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');
    if (year != 2011) alert( 'Неверно!' );
```

Алгоритм работы оператора `if` представлен на рисунке 2.3.

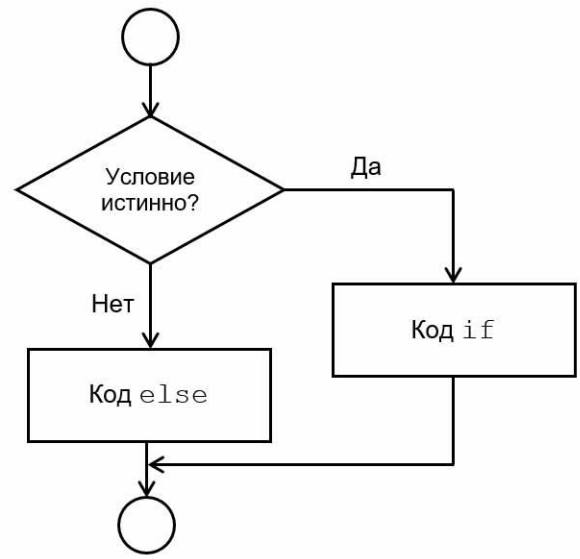


Рисунок 2.3 – Алгоритм работы оператора условия

В то время как операторы `if...else` позволяют выбрать один из двух путей, операторы `switch` позволяют выбрать один путь из нескольких на основании единого условия. Для этого условие должно быть большим, чем значение «истина/ложь»: условие оператора `switch` – это выражение, вычисление которого дает значение, как показано в листинге 2.16.

Листинг 2.16 – Реализация оператора `switch`

```
    switch (выражение)
      { case значениеl:
        // выполняется, когда результат выражения соответствует значениеl
        [break;]
      case значение2:
        // выполняется, когда результат выражения соответствует значение2
        [break;]
    case значениеN:
```

```
        // выполняется, когда результат выражения соот-
ветствует значениеN
        [break;]
    default:
        // выполняется, когда ни одно из значений не со-
ответствует значению выражения
        [break;]
    }
```

JavaScript вычисляет выражение, выбирает первый соответствующий раздел `case` и выполняет его операторы, пока не встретит оператор `break`, `return`, `continue`, `throw` или конец оператора `switch`. Из-за своих особенностей оператор `switch` подвергся серьезной критике, поскольку является популярным источником ошибок программистов. Зачастую начинающим программистам не рекомендуют его использовать вообще. Но оператор `switch` очень полезен в соответствующей ситуации: это хороший инструмент, который стоит иметь в своем арсенале, но, как и любым инструментом, им следует научиться владеть, использовать осторожно и там, где нужно.

Порядок выполнения работы

1 Напишите код, который будет спрашивать логин (`prompt`). Если посетитель вводит «`admin`», то спрашивать пароль, если нажал отмена (`escape`) – выводить «`Canceled`», если вводит что-то другое – «`Access Denied`». Пароль проверять так: если введен пароль «`verysecurepassword`», то выводить «`Welcome, admin!`» при этом выводя введенного в поле логина пользователя, иначе – «`Wrong Password`», при отмене – «`Canceled`».

2 Используя конструкцию `if...else`, напишите код, который получает значение `prompt`, а затем выводит `alert`:

– `1`, если значение больше нуля;

– `-1`, если значение меньше нуля;

– `0`, если значение равно нулю.

3 Напишите условие `if` для проверки того факта, что переменная `age` находится между 25 и 75 включительно.

4 Напишите цикл, который предлагает `prompt` ввести число, большее 97. Если посетитель ввел другое число – попросить ввести еще раз, и так далее. Цикл должен спрашивать число, пока либо посетитель не введет число, большее 97, либо не нажмет кнопку «*CANCEL*» («*Esc*»). Обработать возможность введения нечисловых строк.

5 Создайте код, который выводит все простые числа из интервала от 2 до 10. Результат должен быть: 2, 3, 5, 7. Код также должен легко модифицироваться для любых других интервалов.

6 Напишите программу на *JavaScript*, которая принимает два целых числа и отображает большее из них.

7 Используйте условный оператор *JavaScript* для сортировки пяти чисел по убыванию: 2, 7, –4, 11, –32.

8 Напишите программу на *JavaScript*, которая вычисляет количество баллов по результатам тестирования по дисциплине «Программные инновационные платформы информационных систем» следующих магистрантов. Затем эти баллы используются для определения квалификации.

Фамилия и инициалы магистранта	**Количество баллов по дисциплине**
Александров К. Н.	55
Белый И. И.	67
Виноградов К. С.	75
Грибов А. Н.	85
Дмитриев В. С.	80

Уровень квалификации рассчитывается следующим образом:

Уровень классификации	**Классификация**
< 60	*F*
< 70	*D*
< 80	*C*
< 90	*D*
< 100	*A*

9 Определение счастливого числа: «Начиная с любого положительного целого числа, замените число суммой квадратов его цифр и повторяйте процесс, пока число не станет равным 1 (где оно останется). Те числа, для которых этот процесс заканчивается на 1, являются счастливыми числами». Напишите программу на *JavaScript*, чтобы найти и распечатать первые пять счастливых чисел.

10 Напишите программу на JavaScript, чтобы найти трехзначные числа Армстронга.

Примечание: числом Армстронга называется такое натуральное число, которое равно сумме всех своих цифр, возведенных в степень, равную количеству цифр данного числа. Например, десятичное число 153 является числом Армстронга, так как $1 \cdot 1 \cdot 1 + 5 \cdot 5 \cdot 5 + 3 \cdot 3 \cdot 3 = 153$.

Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.

4 Реализация решения задач с подробным рассмотрением (описанием) решений.

5 Выводы.

6 Список использованных источников.

Контрольные вопросы

1 Что представляет собой язык программирования *JavaScript*?

2 Что такое переменная и как она объявляется?

3 Что такое идентификатор?

4 По каким правилам задается идентификатор?

5 Что такое объект?

6 Как обратиться к свойству объекта?

7 Какие существуют основные типы данных в *JavaScript*?

8 Что из себя представляют массивы и как они объявляются?

9 Какие существуют базовые операции по работе с полученными от пользователя данными?

10 Перечислить основные используемые циклы в *JavaScript*. Что представляет собой каждый из них?

11 Как работает оператора `switch`.

12 Какие существуют операторы, которые могут изменять ход управления потоком данных?

Список использованных источников

1 Thomson, B. S. JavaScript. Notes for Professionals [Электронный ресурс]. – 2018. – Режим доступа: https://www.tradepub.com/free/w_goaa19/.

2 Скотт, А. Д. Разработка на JavaScript. Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron / А. Д. Скотт; пер. с англ. – СПб. : Питер, 2021. – 320 с.

3 Флэнаган, Д. JavaScript. Полное руководство / Д. Флэнаган. – 7-е изд. ; пер. с англ. – СПб. : ООО «Диалектика», 2021. – 720 с.

4 Фримен, Э. Изучаем программирование на JavaScript / Э. Фримен, Э. Робсон. – СПб. : Питер, 2015. – 640 с.

5 Хавербеке, М. Выразительный JavaScript. Современное веб-программирование / М. Хавербеке. – 3-е изд. – СПб. : Питер, 2019. – 480 с.

ЛАБОРАТОРНАЯ РАБОТА № 3

РАБОТА С МАССИВАМИ

Цель работы

Изучить понятие массивов и взаимодействие с ними.

Теоретические сведения

Массив – это упорядоченная коллекция значений или, иными словами, это специальная переменная, которая может содержать более одного значения единовременно.

Значения в массиве называются элементами, и каждый элемент характеризуется числовой позицией в массиве, которая называется индексом. Массивы в языке *JavaScript* являются нетипизированными: элементы массива могут иметь любой тип, причем разные элементы одного и того же массива могут иметь разные типы. Элементы массива могут даже быть объектами или другими массивами, что позволяет создавать сложные структуры данных, такие как массивы объектов и массивы массивов.

Отсчет индексов массивов в языке *JavaScript* начинается с нуля, и для них используются 32-битные целые числа. Первый элемент массива имеет индекс `0`. Массивы в *JavaScript* являются динамическими:

– они могут увеличиваться и уменьшаться в размерах по мере необходимости;

– нет необходимости объявлять фиксированные размеры массивов при их создании или повторно распределять память при изменении их размеров.

Массивы в языке *JavaScript* – это специализированная форма объектов, а понятие индексов массивов шире, чем просто имена свойств, которые по совпадению являются целыми числами.

Легче всего создать массив с помощью литерала, который представляет собой простой список разделенных запятыми элементов массива в квадратных скобках. Значения в литерале массива не обязательно должны быть константами – это могут быть любые выражения, в том числе и литералы объектов, как показано в листинге 3.1.

Листинг 3.1 – Создание массива

```
var empty = [];
// пустой массив
var numbers = [2, 3, 5, 7, 11];
// массив с пятью числовыми элементами
var misc = [ 1.1, true, "a", ];
// 3 элемента разных типов + завершающая запятая
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

```
// массив с переменными
var arrObj = [[1,{x:1, y:2}], [2, {x:3, y:4}]];
// 2 массива внутри, содержащие объекты
```

Синтаксис литералов массивов позволяет вставлять необязательную завершающую запятую, то есть литерал `[,,]` соответствует массиву с двумя элементами, а не с тремя.

Другой способ создания массива состоит в вызове конструктора `Array()`. Вызвать конструктор можно тремя разными способами.

Первый способ – вызвать конструктор без аргументов, как показано в листинге 3.2. В этом случае будет создан пустой массив, эквивалентный литералу `[]`.

Листинг 3.2 – Вызов конструктора без аргументов

```
var arr = new Array();
```

Второй способ – вызвать конструктор с единственным числовым аргументом, определяющим длину массива, как показано в листинге 3.3. В этом случае будет создан пустой массив указанной длины. Такая форма вызова конструктора `Array()` может использоваться для предварительного распределения памяти под массив, если заранее известно количество его элементов. Обратите внимание, что при этом в массиве не сохраняется никаких значений.

Листинг 3.3 – Вызов конструктора с аргументом

```
var arr = new Array(10);
```

Третий способ – явно указать в вызове конструктора значения первых двух или более элементов массива или один нечисловой элемент, как показано в листинге 3.4. В этом случае аргументы конструктора становятся значениями элементов нового массива. Использование литералов массивов практически всегда проще, чем подобное применение конструктора `Array()`.

Листинг 3.4 – Явное указание элементов массива

```
var arr = new Array(5, 4, 3, 2, 1, "тест");
```

Доступ к элементам массива осуществляется с помощью оператора `[]`. Слева от скобок должна присутствовать ссылка на массив. Внутри скобок должно находиться произвольное выражение, возвращающее неотрицательное целое значение. Этот синтаксис пригоден как для чтения, так и для записи значения элемента массива. Следовательно, допустимы все приведенные далее *JavaScript*-инструкции, как показано в листинге 3.5.

Листинг 3.5 – Доступ к элементам массива

```
// создать массив с одним элементом
var arr = ["world"];
```

```
    // прочитать элемент 0
    var value = arr[0];
    // записать значение в элемент 1
    arr[1] = 3.14;
    // записать значение в элемент 2
    i = 2; arr[i] = 3;
    // записать значение в элемент 3
    arr[i + 1] = 'привет';
    // прочитать элементы 0 и 2, записать значение в эле-
мент 3
    arr[arr[i]] = arr[0];
```

Массивы являются специализированной разновидностью объектов. Квадратные скобки, используемые для доступа к элементам массива, действуют точно так же, как квадратные скобки, используемые для доступа к свойствам объекта. Интерпретатор *JavaScript* преобразует указанные в скобках числовые индексы в строки – индекс `1` превращается в строку «1» – а затем использует строки как имена свойств.

Такие же преобразования можно производить с обычными объектами, как показано в листинге 3.6.

Листинг 3.6 – Преобразование числовых индексов

```
    var obj = {};         // создать простой объект
    obj[1] = "one";       // индексировать его целыми чис-
лами
```

Особенность массивов состоит в том, что при использовании имен свойств, которые являются неотрицательными целыми числами, массивы автоматически определяют значение свойства `length`. Например, выше был создан массив `arr` с единственным элементом. Затем были присвоены значения его элементам с индексами 1, 2 и 3. В результате этих операций значение свойства `length` массива изменилось и стало равным 4.

Следует четко отличать индексы в массиве от имен свойств объектов. Все индексы являются именами свойств, но только свойства с именами, представленными целыми числами, являются индексами. Все массивы являются объектами, и можно добавлять к ним свойства с любыми именами. Однако, если затрагивать свойства, которые являются индексами массива, массивы реагируют на это, обновляя значение свойства `length` при необходимости.

В качестве индексов массивов допускается использовать отрицательные и нецелые числа. В этом случае числа преобразуются в строки, которые используются как имена свойств.

Самый простой способ добавить элементы в массив заключается в том, чтобы присвоить значения новым индексам. Для добавления одного или более

элементов в конец массива можно также использовать метод `push()`, как показано в листинге 3.7.

Листинг 3.7 – Добавление элемента массива в конец

```
var arr = [];         // создать пустой массив
arr.push('zero');     // добавить значение в конец
arr.push('one',2);    // добавить еще два значения
```

Добавить элемент в конец массива можно также, присвоив значение элементу `arr[arr.length]`. Для вставки элемента в начало массива можно использовать метод `unshift()`, при этом существующие элементы в массиве смещаются в позиции с более высокими индексами.

Удалять элементы массива можно с помощью оператора `delete`, как обычные свойства объектов, как показано в листинге 3.8.

Листинг 3.8 – Удаление элементов массива

```
var arr = [1,2,'three'];
delete arr[2];
2 in arr; // false, индекс 2 в массиве не определен
arr.length; // 3: оператор delete не изменяет свой-
ство length массива
```

Удаление элемента напоминает, но несколько отличается от присваивания значения `undefined` этому элементу. Обратите внимание, что применение оператора `delete` к элементу массива не изменяет значение свойства `length` и не сдвигает вниз элементы с более высокими индексами, чтобы заполнить пустое пространство, появившееся после удаления элемента.

Кроме того, имеется возможность удалять элементы в конце массива простым присваиванием нового значения свойству `length`. Массивы имеют метод `pop()` (противоположный методу `push()`), который уменьшает длину массива на 1 и возвращает значение удаленного элемента. Также имеется метод `shift()` (противоположный методу `unshift()`), который удаляет элемент в начале массива. В отличие от оператора `delete` метод `shift()` сдвигает все элементы вниз на позицию ниже их текущих индексов.

Существует многоцелевой метод `splice()`, позволяющий вставлять, удалять и замещать элементы массивов. Он изменяет значение свойства `length` и сдвигает элементы массива с более низкими или более высокими индексами по мере необходимости.

JavaScript не поддерживает «настоящие» многомерные массивы, но позволяет неплохо имитировать их при помощи массивов из массивов. Для доступа к элементу данных в массиве массивов достаточно дважды использовать оператор `[]`.

Например, предположим, что переменная `matrix` – это массив массивов чисел. Каждый элемент `matrix[x]` – это массив чисел. Для доступа к определенному числу в массиве можно использовать выражение `matrix[x][y]`. В листинге 3.9 приводится конкретный пример, где двумерный массив используется в качестве таблицы умножения.

Листинг 3.9 – Использование двумерного массива

```
    // Создать многомерный массив
    var table = new Array(10);        // в таблице 10
строк
    for(var i = 0; i < table.length; i++)
        table[i] = new Array(10);     // в каждой строке
10 столбцов
    // Инициализировать массив и вывести на консоль
    for(var row = 0, str = ''; row < table.length; row++)
{
        for(var col = 0; col < table[row].length; col++)
{
            table[row][col] = (row+1)*(col+1);
            str += table[row][col] + '  ';
        }
        console.log(str + '\n');
        str = '';
    }
```

Стандарт *ECMAScript* определяет в составе `Array.prototype` множество удобных функций для работы с массивами, которые доступны как методы любого массива. Эти методы будут представлены в следующих подразделах.

Метод `Array.join()` преобразует все элементы массива в строки, объединяет их и возвращает получившуюся строку. В необязательном аргументе методу можно передать строку, которая будет использоваться для отделения элементов в строке результата. Если строка-разделитель не указана, используется запятая. Например, в листинге 3.10 фрагмент дает в результате строку «1, 2, 3».

Листинг 3.10 – Использование разделителей

```
var arr = [1,2,3];
arr.join();          // '1,2,3'
arr.join("-");       // '1-2-3'
```

Метод `Array.reverse()` меняет порядок следования элементов в массиве на обратный и возвращает переупорядоченный массив. Перестановка выполняется непосредственно в исходном массиве, то есть этот метод не создает новый массив с переупорядоченными элементами, а переупорядочивает их в уже существующем массиве. Например, в листинге 3.11 фрагмент, где используются методы `reverse()` и `join()`, дает в результате строку «3, 2, 1».

Листинг 3.11 – Использование метода `reverse`

```
var arr = [1,2,3];
arr.reverse().join();     // «3,2,1»
```

Метод `Array.sort()` сортирует элементы в исходном массиве и возвращает отсортированный массив. Если метод `sort()` вызывается без аргументов, сортировка выполняется в алфавитном порядке (для сравнения: элементы временно преобразуются в строки, если это необходимо). Неопределенные элементы переносятся в конец массива.

Для сортировки в каком-либо ином порядке, отличном от алфавитного, методу `sort()` можно передать функцию сравнения в качестве аргумента. Эта функция устанавливает, какой из двух ее аргументов должен следовать раньше в отсортированном списке. Если первый аргумент должен предшествовать второму, функция сравнения должна возвращать отрицательное число. Если первый аргумент должен следовать за вторым в отсортированном массиве, то функция должна возвращать число больше нуля. А если два значения эквивалентны (то есть порядок их следования не важен), функция сравнения должна возвращать `0`, как показано в листинге 3.12.

Листинг 3.12 – Сортировка массива

```
var arr = [33, 4, 1111, 222];
arr.sort(); // алфавитный порядок: 1111, 222, 33, 4
arr.sort(function(a,b) {
    // числовой порядок: 4, 33, 222, 1111
    return a-b; // возвращает значение < 0, 0 или >
0 в зависимости от порядка сортировки a и b
});
// сортируем в обратном направлении, от большего
к меньшему
arr.sort(function(a,b) {return b-a});
```

Метод `Array.concat()` создает и возвращает новый массив, содержащий элементы исходного массива, для которого был вызван метод `concat()`, и значения всех аргументов, переданных методу `concat()`. Если какой-либо из этих аргументов сам является массивом, его элементы добавляются в возвращаемый массив. Следует отметить, что рекурсивного превращения массива из массивов в одномерный массив не происходит. Метод `concat()` не изменяет исходный массив, как показано в листинге 3.13.

Листинг 3.13 – Использование метода `concat`

```
var arr = [1,2,3];
arr.concat(4, 5);         // вернет    [1,2,3,4,5]
arr.concat([4,5]);        // вернет    [1,2,3,4,5]
```

```
arr.concat([4,5],[6,7])   // вернет    [1,2,3,4,5,6,7]
arr.concat(4, [5,[6,7]]) // вернет [1,2,3,4,5,[6,7]]
```

Метод `Array.slice()` возвращает фрагмент, или подмассив, указанного массива. Два аргумента метода определяют начало и конец возвращаемого фрагмента. Возвращаемый массив содержит элемент, номер которого указан в первом аргументе, плюс все последующие элементы, вплоть до (но не включая) элемента, номер которого указан во втором аргументе.

Если указан только один аргумент, возвращаемый массив содержит все элементы от начальной позиции до конца массива. Если какой-либо из аргументов имеет отрицательное значение, он определяет номер элемента относительно конца массива. Так, аргументу «`-1`» соответствует последний элемент массива, а аргументу «`-3`» – третий элемент массива с конца, как показано в листинге 3.14.

Листинг 3.14 – Определение элементов массива

```
var arr = [1,2,3,4,5];
arr.slice(0,3);       // вернет [1,2,3]
arr.slice(3);         // вернет [4,5]
arr.slice(1,-1);      // вернет [2,3,4]
arr.slice(-3,-2);     // вернет [3]
```

Метод `Array.splice()` – это универсальный метод, выполняющий вставку или удаление элементов массива. В отличие от методов `slice()` и `concat()`, метод `splice()` изменяет исходный массив, относительно которого он был вызван. Обратите внимание, что методы `splice()` и `slice()` имеют очень похожие имена, но выполняют совершенно разные операции.

Метод `splice()` может удалять элементы из массива, вставлять новые элементы или выполнять обе операции одновременно. Элементы массива при необходимости смещаются, чтобы после вставки или удаления образовывалась непрерывная последовательность.

Первый аргумент метода `splice()` определяет позицию в массиве, начиная с которой будет выполняться вставка и/или удаление. Второй аргумент определяет количество элементов, которые должны быть удалены (вырезаны) из массива. Если второй аргумент опущен, удаляются все элементы массива от указанного до конца массива. Метод `splice()` возвращает массив удаленных элементов или (если ни один из элементов не был удален) пустой массив.

Первые два аргумента метода `splice()` определяют элементы массива, подлежащие удалению. За этими аргументами может следовать любое количество дополнительных аргументов, определяющих элементы, которые будут вставлены в массив, начиная с позиции, указанной в первом аргументе, как показано в листинге 3.15.

Листинг 3.15 – Использование метода `splice`

```
var arr = [1,2,3,4,5,6,7,8];
arr.splice(4); // вернет [5,6,7,8], arr = [1,2,3,4]
arr.splice(1,2); // вернет [2,3], arr = [1,4]
arr.splice(1,1); // вернет [4]; arr = [1]

arr = [1,2,3,4,5];
arr.splice(2,0,'a','b'); // вернет []; arr = [1,2,
'a','b',3,4,5]
```

Методы `push()` и `pop()` позволяют работать с массивами как со стеками. Метод `push()` добавляет один или несколько новых элементов в конец массива и возвращает его новую длину. Метод `pop()` выполняет обратную операцию – удаляет последний элемент массива, уменьшает длину массива и возвращает удаленное им значение. Обратите внимание, что оба эти метода изменяют исходный массив, а не создают его модифицированную копию.

Методы `unshift()` и `shift()` ведут себя почти так же, как `push()` и `pop()`, за исключением того, что они вставляют и удаляют элементы в начале массива, а не в конце. Метод `unshift()` смещает существующие элементы в сторону больших индексов для освобождения места, добавляет элемент/элементы в начало массива и возвращает новую длину массива. Метод `shift()` удаляет и возвращает первый элемент массива, смещая все последующие элементы на одну позицию вниз, чтобы занять место, освободившееся в начале массива.

Порядок выполнения

1 Рассмотрите задачу из 5 шагов:

– создайте массив `styles` с элементами «`Микросхемы`», «`Транзисторы`»;

– добавьте в конец значение «`Диоды`»;

– замените предпоследнее значение с конца на «`Другие полупроводниковые элементы`». Код замены предпоследнего значения должен работать для массивов любой длины;

– удалите первое значение массива и выведите его `alert`;

– добавьте в начало значения «`Дискретные элементы`» и «`SMD-элемиенты`».

Каждый шаг необходимо вывести в консоль.

2 Напишите код, который:

– запрашивает по очереди значения при помощи `prompt` и сохраняет их в массиве;

– заканчивает ввод, как только посетитель введет пустую строку, символ (не число) или нажмет «Отмена»; при этом ноль (0) не должен заканчивать ввод, это разрешенное число;

– выводит сумму всех значений массива.

3 Создайте функцию filterRange(arr, a, b), которая принимает массив чисел arr и возвращает новый массив, который содержит только числа из arr из диапазона от a до b. То есть проверка имеет вид a ≤ arr[i] ≤ b. Функция не должна менять arr.

4 Реализовать алгоритм «Решето Эратосфена» для поиска всех простых чисел от 1 до 100 и вывести их сумму.

5 Напишите функцию aclean(arr), которая возвращает массив слов, очищенный от анаграмм.

6 Напишите код, который преобразовывает и объединяет все элементы массива в одно строковое значение. Элементы массива будут разделены запятой. Получите результат двумя разными методами.

```
var vegetables = ['SMD-элементы', 'Интегральные схе-
мы', 'Транзисторы', 'Диоды'];
// ваш код
document.writeln(str1); // "SMD-элементы, Интеграль-
ные схемы, Транзисторы, Диоды"
document.writeln(str2); // ""SMD-элементы, Инте-
гральные схемы, Транзисторы, Диоды"
```

7 Напишите код, который ставит двоеточие между цифрами нечетных числел. Пользователь вводит многозначное число через promt. Напишите функцию colonOdd(num), которая принимает число num в качестве аргумента и вставляет двоеточие (:) между двумя нечетными числами. Например, если вводится число 77619217, то на выход должно быть 7:761:921:7.

```
const num = prompt('Введите число', 77);

function colonOdd (num) {
// ваш код
}
document.writeln(colonOdd(num)); // 7:7
```

8 Напишите код замены регистра символов. Пользователь вводит строку кириллицей разного регистра. Примените функцию, которая принимает строку в качестве аргумента и заменяет регистр каждого символа на противоположный. Например, если вводится «ТеРпЕнЬе ДаЕт УмЕнЬе», то на выходе должен быть массив [тЕрПеНьЕ дАеТ уМеНьЕ].

```
const str = prompt('Введите слово', 'ТеРпЕнЬе ДаЕт
УмЕнЬе ');
function changeRegister (str) {
// ваш код
}
document.writeln(changeRegister(str)); // [тЕрПеНьЕ
дАеТ уМеНьЕ]
```

9 Напишите функцию `removeDuplicates(arr)`, которая возвращает массив, в котором удалены повторяющиеся элементы из массива `arr` (игнорируйте чувствительность к регистру).

```
var arr = ["php", "php", "css", "css",
  "script", "script", "html", "html", "java"
];
function removeDuplicates(arr) {
// ваш код
}
document.writeln(result);
// [php,css,script,html,java]
```

10 Напишите код для определения високосных годов

```
function chooseYears(start, end) {
// ваш код
}
document.writeln(chooseYears(2000,2024));
// [2000,2004,2008,2012,2016,2020,2024]
```

11 Напишите функцию `getFirst(array, n)`, которая возвращает фрагмент массива, содержащий первые `'n'` элементов массива.

```
var array = [11,12,13,14,15,16,17,18,19];
 function getFirst(array, n) {
    ваш код
};
document.writeln(getFirst(array));    // 11
document.writeln(getFirst(array, 4)); // 11,12,13,14
document.writeln(getFirst(array,-3));
// 11,12,13,14,15,16
```

Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Реализация решения задач с подробным рассмотрением (описанием) решений.
5 Выводы.
6 Список использованных источников.

Контрольные вопросы

1 Что такое массив?
2 Что включает в себя массив из нескольких элементов?
3 Как удалить элемент массива?
4 Какая цель использования метода `splice()`?
5 Как создать многомерный массив?
6 Как преобразовать все элементы массива в строки? Можно ли при этом изменить стандартный разделитель на какой-либо другой?
7 Как изменить порядок следования элементов в массиве на обратный?
8 Как отсортировать все элементы массива?
9 Как осуществить конкатенацию элементов исходного массива для создания нового массива?
10 Что позволяют делать методы `push()` и `pop()` с массивами?

Список использованных источников

1 Thomson, B. S. JavaScript. Notes for Professionals [Электронный ресурс]. – 2018. – Режим доступа: https://www.tradepub.com/free/w_goaa19/.
2 Скотт, А. Д. Разработка на JavaScript. Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron / А. Д. Скотт; пер. с англ. – СПб. : Питер, 2021. – 320 с.
3 Флэнаган, Д. JavaScript. Полное руководство / Д. Флэнаган. – 7-е изд. ; пер. с англ. – СПб. : ООО «Диалектика», 2021. – 720 с.
4 Фримен, Э. Изучаем программирование на JavaScript / Э. Фримен, Э. Робсон. – СПб. : Питер, 2015. – 640 с.
5 Хавербеке, М. Выразительный JavaScript. Современное веб-программирование / М. Хавербеке. – 3-е изд. – СПб. : Питер, 2019. – 480 с.

ЛАБОРАТОРНАЯ РАБОТА № 4

ФУНКЦИИ

Цель работы

Ознакомиться с основными возможностями функций в *JavaScript*.

Теоретические сведения

Функции – ключевая концепция в *JavaScript*. Важнейшей особенностью языка является первоклассная поддержка функций (`functions as first-class citizen`). Любая функция – это объект, и, следовательно, ею можно манипулировать как объектом:

– передавать как аргумент и возвращать в качестве результата при вызове других функций (функций высшего порядка);

– создавать анонимно и присваивать в качестве значений переменных или свойств объектов.

Это определяет высокую выразительную мощность *JavaScript* и позволяет относить его к числу языков, реализующих функциональную парадигму программирования.

Функция в *JavaScript* – специальный тип объектов, позволяющий формализовать средствами языка определенную логику поведения и обработки данных.

Объявление функции (`function definition`, или `function declaration`, или `function statement`) состоит из ключевого слова `function` и следующих частей:

– имя функции;

– список параметров, принимаемых функцией, заключенных в круглые скобки `()` и разделенных запятыми;

– инструкции, которые будут выполнены после вызова функции, заключают в фигурные скобки `{ }`.

Например, код в листинге 4.1 объявляет простую функцию с именем `square`.

Листинг 4.1 – Объявление функции

```
function square(number) {
  return number * number;
}
```

Функция `square` принимает один параметр, названный `number`. Состоит из одной инструкции, которая означает вернуть параметр этой функции (это `number`), умноженный на самого себя. Инструкция `return` указывает на значение, которое будет возвращено функцией.

Примитивные параметры (например, число) передаются функции значением; значение передается в функцию, но если функция меняет значение параметра, это изменение не отразится на исходном экземпляре параметра (глобально или после вызова функции).

Если передать объект как параметр (не примитив: например, массив или определяемые пользователем объекты), и функция изменит свойство переданного в нее объекта, это изменение будет видно и вне функции, как показано в листинге 4.2.

Листинг 4.2 – Изменение свойства объекта

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}
var mycar = {make: 'Honda', model: 'Accord', year:
1998};
var x, y;
x = mycar.make; // x получает значение "Honda"
myFunc(mycar);
y = mycar.make; // y получает значение "Toyota"
                // (свойство было изменено функцией)
```

Функция вида «`function declaration statement`» по синтаксису является инструкцией (*statement*), еще функция может быть вида «`function definition expression`». Такая функция может быть анонимной (она не имеет имени). Например, функция `square` может быть вызвана, как показано в листинге 4.3.

Листинг 4.3 – Вызов функции

```
var square = function(number) { return number * num-
ber; };
var x = square(4); // x получает значение 16
```

Однако имя может быть присвоено и для вызова самой себя внутри самой функции, и для отладчика (`debugger`) с целью идентифицирования функции в стек-треках, как показано в листинге 4.4.

Листинг 4.4 – Присвоение имени функции

```
var factorial = function fac(n) { return n < 2 ? 1 :
n * fac(n - 1); };
console.log(factorial(3));
```

Функции вида «`function definition expression`» удобны, когда они передается аргументом в другую функцию. В листинге 4.5 показана функция `map`, которая должна получить функцию первым аргументом и массив – вторым.

Листинг 4.5 – Передача функции аргументом

```
function map(f, a) {
  var result = [], // create a new Array
      i;
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
```

В следующим коде функция принимает другую функцию, которая является «function definition expression», и выполняет код для каждого элемента принятого массива вторым аргументом, как показано в листинге 4.6.

Листинг 4.6 – Функция function definition expression

```
function map(f, a) {
  var result = []; // create a new Array
  var i; // declare variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
      return result;
}
var f = function(x) {
   return x * x * x;
}
var numbers = [0, 1, 2, 5, 10];
var cube = map(f,numbers);
console.log(cube);
```

Функция возвращает: `[0, 1, 8, 125, 1000]`.

В *JavaScript* функция может быть объявлена с условием. Например, в листинге 4.7 функция будет присвоена переменной `myFunc` только, если `num` равно `0`.

Листинг 4.7 – Объявление функции с условием

```
var myFunc;
if (num === 0) {
  myFunc = function(theObject) {
    theObject.make = 'Toyota';
  }
}
```

В дополнение к объявлениям функций можно использовать конструктор `Function` для создания функций из строки во время выполнения (`runtime`), подобно `eval()`.

Объявление функции не выполняет ее. Объявление функции просто называет функцию и указывает, что делать при вызове функции. Вызов функции фактически выполняет указанные действия с указанными параметрами. Например, если определить функцию `square`, то можно вызвать ее, как показано в листинге 4.8.

Листинг 4.8 – Вызов функции `square`

```
square(5);
```

Эта инструкция вызывает функцию с аргументом `5`. Функция вызывает свои инструкции и возвращает значение `25`.

Функции могут быть в области видимости, когда они уже определены, но функции вида «`function declaration statement`» могут быть подняты (поднятие – `hoisting`), как показано в листинге 4.9.

Листинг 4.9 – Функции в области видимости

```
console.log(square(5));
/* ... */
function square(n) { return n * n; }
```

Функция может вызвать саму себя. Например, функция рекурсивного вычисления факториала, как показано в листинге 4.10.

Листинг 4.10 – Функция вычисления факториала

```
function factorial(n) {
  if ((n === 0) || (n === 1))
    return 1;
  else
    return (n * factorial(n - 1));
}
```

Есть другие способы вызвать функцию. Существуют частые случаи, когда функции необходимо вызывать динамически, или поменять номера аргументов функции, или необходимо вызвать функцию с привязкой к определенному контексту.

Область видимости функции – функция, в котором она определена, или целая программа, если она объявлена по уровню выше.

Переменные, объявленные в функции, не могут быть доступными где-нибудь вне этой функции, поэтому переменные, которые нужны именно для функции, объявляют только в `scope` функции. При этом функция имеет доступ ко всем переменным и функциям, объявленным внутри ее `scope`. Другими словами, функция, объявленная в глобальном `scope`, имеет доступ ко всем переменным в глобальном `scope`. Функция, объявленная внутри другой функции,

имеет доступ ко всем переменным ее родительской функции и другим переменным, к которым эта родительская функция имеет доступ (листинг 4.11).

Листинг 4.11 – Область видимости вложенной функции

```
// следующие переменные объявлены в глобальном scope
var num1 = 20,
    num2 = 3,
    name = 'Chamahk';
// эта функция объявлена в глобальном scope
function multiply() {
  return num1 * num2;
}
multiply(); // вернет 60
// пример вложенной функции
function getScore() {
  var num1 = 2,
      num2 = 3;
    function add() {
    return name + ' scored ' + (num1 + num2);
  }
    return add();
}
getScore(); // вернет "Chamahk scored 5"
```

`Closures` – замыкание – это комбинация объединенной (вложенной) функции со ссылками на ее окружающее состояние (лексическое окружение). Другими словами, замыкание дает доступ к области действия внешней функции из внутренней. В *JavaScript* замыкания создаются каждый раз во время создания функции. Это одна из главных особенностей *JavaScript*.

Однако внешняя функция не имеет доступа к переменным и функциям, объявленным во внутренней функции. Это обеспечивает своего рода инкапсуляцию для переменных внутри вложенной функции. Еще, поскольку вложенная функция имеет доступ к `scope` внешней функции, переменные и функции, объявленные во внешней функции, будут продолжать существовать и после ее выполнения для вложенной функции, если на них и на нее сохранился доступ (имеется в виду, что переменные, объявленные во внешней функции, сохраняются, только если внутренняя функция обращается к ним). `Closure` создается, когда вложенная функция стала доступной в `scope` вне внешней функции (листинг 4.12).

Листинг 4.12 – `Closure`

```
    var pet = function(name) {   // внешняя функция объ-
явила переменную "name"
      var getName = function() {
```

```
        return name; // вложенная функция имеет доступ к
"name" внешней функции
      }
      return getName; // возвращаем вложенную функцию,
тем самым сохраняя доступ к ней для другого scope
    }
    myPet = pet('Vivie');
    myPet();// возвращается "Vivie",  так как даже после
выполнения внешней функции name сохранился для вложенной
функции
```

Это может быть сложнее для кода выше. Объект с методами для манипуляции вложенной функции внешней функцией можно вернуть (`return`), как показано в листинге 4.13.

Листинг 4.13 – Возвращение объекта

```
    var createPet = function(name) {
      var sex;
      return {
        setName: function(newName) {
          name = newName;
        },
        getName: function() {
          return name;
        },
        getSex: function() {
          return sex;
        },
        setSex: function(newSex) {
          if(typeof newSex === 'string' && (newSex.toLow-
erCase() === 'male' ||
             newSex.toLowerCase() === 'female')) {
            sex = newSex;
          }
        }
      }
    }
    var pet = createPet('Vivie');
    pet.getName();                  // Vivie
    pet.setName('Oliver');
    pet.setSex('male');
    pet.getSex();                   // male
    pet.getName();                  // Oliver
```

В коде выше переменная name внешней функции доступна для вложенной функции, и нет другого способа доступа к вложенным переменным, кроме как через вложенную функцию. Вложенные переменные вложенной функции являются безопасными хранилищами для внешних аргументов и переменных. Они содержат «постоянные» и «инкапсулированные» данные для работы с ними вложенными функциями. Функции даже не должны присваиваться переменной или иметь имя, как показано в листинге 4.14.

Листинг 4.14 – Работа с вложенными функциями

```
var getCode = (function() {
  var apiCode = '0]Eal(eh&2'; // a code we do not
want outsiders to be able to modify...
  return function() {
    return apiCode;
  };
}());
getCode();    // returns the apiCode
```

Однако есть ряд подводных камней, которые следует учитывать при использовании замыканий. Если закрытая функция определяет переменную с тем же именем, что и имя переменной во внешней области, нет способа снова ссылаться на переменную во внешней области, как показано в листинге 4.15.

Листинг 4.15 – Взаимодействие переменных

```
var createPet = function(name) {  // the outer func-
tion defines a variable called "name".
  return {
    setName: function(name) {    // the enclosed
function also defines a variable called "name".
      name = name;               // how do we access
the "name" defined by the outer function?
    }
  }
}
```

В *ECMAScript* есть два вида параметров:

– параметры по умолчанию (default parameters);

– оставшиеся параметры (rest parameters).

В *JavaScript* параметры функции по умолчанию имеют значение undefined. Однако в некоторых ситуация может быть полезным поменять значение по умолчанию. В таких случаях default parameters могут быть весьма кстати.

В прошлом для этого было необходимо в теле функции проверять значения параметров на undefined и в положительном случае менять это значение на

дефолтное (default). В следующем примере, в случае если при вызове не предоставили значение для b, то этим значением станет undefined, тогда результатом вычисления a * b в функции multiply будет NaN. Во второй строке увидим это значение, как показано в листинге 4.16.

Листинг 4.16 – Результат выражения

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;
  return a * b;
}
multiply(5); // 5
```

С параметрами по умолчанию проверка наличия значения параметра в теле функции не нужна. Теперь можно просто указать значение по умолчанию для параметра b в объявлении функции (листинг 4.17).

Листинг 4.17 – Значение по умолчанию в объявлении функции

```
function multiply(a, b = 1) {
  return a * b;
}
multiply(5); // 5
```

Оставшиеся параметры предоставляют массив неопределенных аргументов. В примере используются оставшиеся параметры, чтобы собрать аргументы с индексами от второго до последнего. Затем умножим каждый из них на значение первого аргумента. В этом примере используется стрелочная функция (Arrow functions), как показано в листинге 4.18.

Листинг 4.18 – Пример стрелочной функции

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(x => multiplier * x);
}
var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Стрелочные функции – функции вида «arrow function expression» – имеют укороченный синтаксис по сравнению с function expression и лексически связывают значение this. Стрелочные функции всегда анонимны.

На введение стрелочных функций повлияли два фактора: более короткие функции и лексика this.

В некоторых функциональных паттернах приветствуется использование более коротких функций, как показано в листинге 4.19.

Листинг 4.19 – Пример более короткой стрелочной функции

```
var a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];
var a2 = a.map(function(s) { return s.length; });
console.log(a2); // logs [8, 6, 7, 9]
var a3 = a.map(s => s.length);
console.log(a3); // logs [8, 6, 7, 9]
```

До стрелочных функций каждая новая функция определяла свое значение `this` (новый объект в случае конструктора, `undefined` в `strict mode`, контекстный объект, если функция вызвана как метод объекта, и так далее). Это оказалось «раздражающим» с точки зрения объектно-орентированного стиля программирования, как показано в листинге 4.20.

Листинг 4.20 – Реализация без стрелочной функции

```
function Person() {
  // конструктор Person() определяет `this` как са-
мого себя.
  this.age = 0;
  setInterval(function growUp() {
    // без strict mode функция growUp() определяет
`this` как global object, который отличается от `this`
определенного конструктором Person().
    this.age++;
  }, 1000);
}
var p = new Person();
```

Функция обратного вызова (`callback`) – это функция, которая передается в качестве аргумента другой функции для последующего ее вызова.

Функции обратного вызова часто используются в качестве обработчиков событий.

В листинге 4.21 приведен пример функции, принимающей в качестве своего аргумента ссылку на другую функцию для ее последующего вызова.

Листинг 4.21 – Функция в качестве аргумента

```
function foo(callback) { return callback(); }
foo (function() { alert("Привет!"); });
```

Этот пример наглядно демонстрирует принцип действия обратного вызова.

Порядок выполнения

1 Создайте страницу, которая запрашивает ввод двух чисел `a` и `b`, с обработкой попытки ввода буквы, после чего выводит результат возведения в степень `b` числа `a`.

2 Напишите функцию создания генератора `sequence(start, step)`. Она при вызове возвращает другую функцию-генератор, которая при каждом вызове дает число на 1 больше, и так до бесконечности. Начальное число, с которого начинать отсчет, и шаг задаются при создании генератора.

3 Напишите функцию `map(fn, array)`, которая принимает на вход функцию и массив и обрабатывает каждый элемент массива этой функцией, возвращая новый массив.

4 Напишите функцию `bind`, которая позволяет привязать контекст (значение `this`) к функции.

5 Напишите функцию, считающую число свойств в объекте.

6 Напишите функцию *JavaScript*, которая «переворачивает» число.

```
function reverse_a_number(n) {
// ваш код
};
document.writeln(reverse_a_number(258961));
// 169852
```

7 Напишите функцию `substrings(str)`, которая генерирует все комбинации строки.

Пример строки: `'piks'`.

Ожидаемый результат: `p, i, pi, k, pi, ik, pik, s, ps, is, ks, piks`.

```
function substrings(str) {
// ваш код
}
substrings("piks"); // p, i, pi, k, pi, ik, pik, s,
ps, is, ks, piks
```

8 Напишите функцию `alphabet_order(str)`, которая возвращает переданную строку с буквами в алфавитном порядке.

Пример строки: `'alphabetical'`.

Ожидаемый результат: `'aaabcehillpt'`.

Предположим, что символы пунктуации и цифры не включены в переданную строку.

```
function alphabet_order(str) {
// ваш код
}
document.writeln(alphabet_order("alphabetical"));
// "aaabcehillpt"
```

9 Напишите функцию `find_longest_word(str)`, которая принимает строку в качестве параметра и находит самое длинное слово в строке.

```
function find_longest_word(str) {
// ваш код
}
document.writeln(find_longest_word('Web Development
Tutorial')); // "Development"
```

10 Напишите функцию `vowel_count(str)`, которая принимает строку в качестве параметра и подсчитывает количество гласных в строке.

```
function vowel_count(str1) {
// ваш код
}
document.writeln(vowel_count('Web Development Tuto-
rial')); //9
```

11 Напишите функцию `detect_data_type(value)`, которая принимает аргумент и возвращает тип.

Примечание – Существует шесть возможных значений, которые возвращает `typeof`: объект, логическое значение, функция, число, строка и неопределенное значение.

```
function detect_data_type(value) {
    ...ваш код...
};
document.writeln(detect_data_type(7)); // number
document.writeln(detect_data_type('wm-school'));
// string
document.writeln(detect_data_type(false)); // bool-
ean
```

Содержание отчета

1 Титульный лист.

2 Цель работы.

3 Краткие теоретические сведения.

4 Реализация решения задач с подробным рассмотрением (описанием) решений.

5 Выводы.

6 Список использованных источников.

Контрольные вопросы

1 Что такое функция *JavaScript*?

2 Что представляет собой функция вида «`function declaration statement`»?

3 Что представляет собой функция вида «`function definition expression`»?

4 Что происходит при объявлении функции?

5 Может ли функция вызывать саму себя?

6 Что такое область видимости функции?

7 Что такое `Closures`?

8 Какое значение имеют параметры функции по умолчанию?

9 Что представляет собой стрелочная функция?

10 Что представляет собой функция обратного вызова?

Список использованных источников

1 Thomson, B. S. JavaScript. Notes for Professionals [Электронный ресурс]. – 2018. – Режим доступа: https://www.tradepub.com/free/w_goaa19/.

2 Скотт, А. Д. Разработка на JavaScript. Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron / А. Д. Скотт; пер. с англ. – СПб. : Питер, 2021. – 320 с.

3 Флэнаган, Д. JavaScript. Полное руководство / Д. Флэнаган. – 7-е изд. ; пер. с англ. – СПб. : ООО «Диалектика», 2021. – 720 с.

4 Фримен, Э. Изучаем программирование на JavaScript / Э. Фримен, Э. Робсон. – СПб. : Питер, 2015. – 640 с.

5 Хавербеке, М. Выразительный JavaScript. Современное веб-программирование / М. Хавербеке. – 3-е изд. – СПб. : Питер, 2019. – 480 с.

ЛАБОРАТОРНАЯ РАБОТА № 5

HTTP-СЕРВЕР НА Node.js

Цель работы

Ознакомиться с основами разработки на платформе *Node.js*.

Теоретические сведения

Для работы потребуется *Node.js*, установить которую можно, используя официальный сайт *https://nodejs.org/en/*.

Для работы платформы *Node.js* используется виртуальная машина `V8`, которая задействует *Google Chrome* для серверного программирования. Благодаря `V8` производительность *Node.js* стала заметно выше, поскольку устраняются промежуточные этапы создания исполняемого кода. Вместо генерирования байт-кода или использования интерпретатора выполняется непосредственная компиляция в собственный машинный код. В связи с тем что *Node.js* применяет *JavaScript* на стороне сервера, появляются следующие преимущества:

– разработчики могут создавать веб-приложения на одном языке, благодаря чему снижается потребность в переключении контекста при разработке серверов и клиентов. При этом обеспечивается совместное использование кода клиентом и сервером, например, кода проверки данных, вводимых в форму, или кода игровой логики;

– популярнейший формат обмена данными *JSON* является собственным форматом *JavaScript*;

– язык *JavaScript* применяется в различных базах данных *NoSQL* (например, в *CouchDB* и *MongoDB*), поэтому подключение к таким базам данных осуществляется в естественной форме. Например, оболочкой и языком запросов для базы данных *MongoDB* является язык *JavaScript*; языком проецирования/сведения для базы данных *CouchDB* также является *JavaScript*;

– целью компиляции в *Node.js* является *JavaScript*, к тому же в настоящее время существует ряд других языков программирования, компилируемых в *JavaScript*;

– в *Node.js* используется единственная виртуальная машина (`V8`), совместимая со стандартом *ECMAScript*. Другими словами, не придется ожидать, пока во всех браузерах станут доступны все новые средства языка *JavaScript*, связанные с платформой *Node.js*.

Притягательность *Node.js* для разработчиков объясняется высокой производительностью и некоторыми другими упомянутыми ранее преимуществами. Все эти преимущества обеспечивает не только *JavaScript*, но и то, как этот язык используется в *Node.js*.

Node Package Manager (`npm`) – быстрая и эффективная система управления пакетами, которой экосистема *Node.js* во многом обязана своим быстрым ростом и разнообразием.

Она устанавливается при инсталляции *Node.js* и хранит описание зависимостей проекта – как и относящиеся к проекту метаданные – в файле `package.json`. Простейший способ создать этот файл – выполнить команду, представленную в листинге 5.1.

Листинг 5.1 – Инициализация npm

```
npm init
```

Она задаст ряд вопросов и сгенерирует `package.json` для начала работы (для указания точки входа используйте название своего проекта).

В пакет `package.json` рекомендуется включать следующие поля:

– `name` – имя пакета (обязательно);

– `description` – описание пакета;

– `version` – текущая версия, соответствующая требованиям семантической версии (обязательно);

– `keywords` – массив условий поиска;

– `maintainers` – массив ответственных за сопровождение пакета (с именами, адресами электронной почты и сайтами);

– `contributors` – массив соавторов пакета (с именами, адресами электронной почты и сайтами);

– `bugs` – *URL*-адрес для отправки ошибок;

– `licenses` – массив лицензий;

– `repository` – репозиторий пакетов;

– `dependencies` – необходимые пакеты и их номера версий.

Каждый раз, когда запускается `npm`, необходимо показывать предупреждения, если не будет указан *URL* репозитория в `package.json` или не создать непустой файл `README.md`. Метаданные в файле `package.json` необходимы, только если необходимо публиковаться в репозитории `npm`.

Основная команда, которая будет использоваться с `npm`, – это `install`. Например, чтобы установить `Grunt` (популярный исполнитель задач для *JavaScript*), можно выполнить команду (для `Debian`-дистрибутивов перед командой необходимо использовать `sudo`), представленную в листинге 5.2.

Листинг 5.2 – Установка глобальной зависимости

```
npm install -g grunt-cli
```

Флаг `-g` сообщает `npm` о необходимости глобальной установки пакета, означающей его доступность по всей системе. Если на сервере установлена операционная система *Linux*, то необходимо использовать `sudo` для глобальной установки модуля.

Программа `npm` не только устанавливает зависимость, но и находит модули, от которых она зависит, и устанавливает их. Чем сложнее зависимость, тем больше модулей устанавливается.

Файл package.json можно обновить, а зависимости будут установлены не глобально, а в той папке, в которой была запущена команда.

Основной список команд npm представлен в таблице 5.1.

Таблица 5.1 – Список основных команд менеджера пакетов npm

Команда	Описание команды
npm init	Инициализация npm в проекте
npm install grunt-cli	Установка зависимости для проекта
npm install -g grunt-cli	Глобальная установка зависимости в системе
npm install --save grunt-cli	Установка зависимости в проекте с занесением ее в файл package.json
npm install grunt-cli@1,2,0	Установка версии зависимости
npm uninstall grunt-cli	Удаление зависимости
npm update	Проверка зависимостей из файла package.json на наличие новых версий зависимостей и их обновление
npm update grunt-cli	Обновление одной конкретной зависимости
npm ls	Вывод списка всех установленных зависимостей
npm ll	Расширенный вывод списка всех установленных зависимостей
npm ls –g	Вывод списка всех глобально установленных зависимостей
npm config list	Получение дополнительной информации об установке npm
npm config ls -l	Более подробное описание всех параметров конфигурации

Поскольку каталог *Node*_modules (в нем содержатся установленные зависимости) в любой момент может быть восстановлен с помощью npm, нет необходимости сохранять его в репозитории. Чтобы убедиться, что *Node*_modules не был добавлен в репозиторий, создадим файл с именем .gitignore и внесем данную папку в этот файл.

Модули *Node.js* и пакеты npm – понятия, связанные между собой, но различающиеся. Модули *Node.js* предоставляют механизм модуляризации и инкапсуляции. Пакеты npm обеспечивают стандартизированную схему для хранения проектов, контроля версий и ссылок на проекты, которые не ограничиваются модулями.

Функция require для импорта модулей. По умолчанию *Node.js* ищет модули в каталоге *Node*_modules (так что неудивительно, что внутри

`Node_modules` будет каталог `express`). Тем не менее *Node.js* предоставляет также механизм создания ваших собственных модулей (лучше никогда не создавать собственные модули в каталоге `Node_modules`).

Для написания первой программы на *Node.js* будем использовать текстовый редактор *Microsoft Visual Studio Code*. Интерфейс данного редактора представлен на рисунке 5.1.

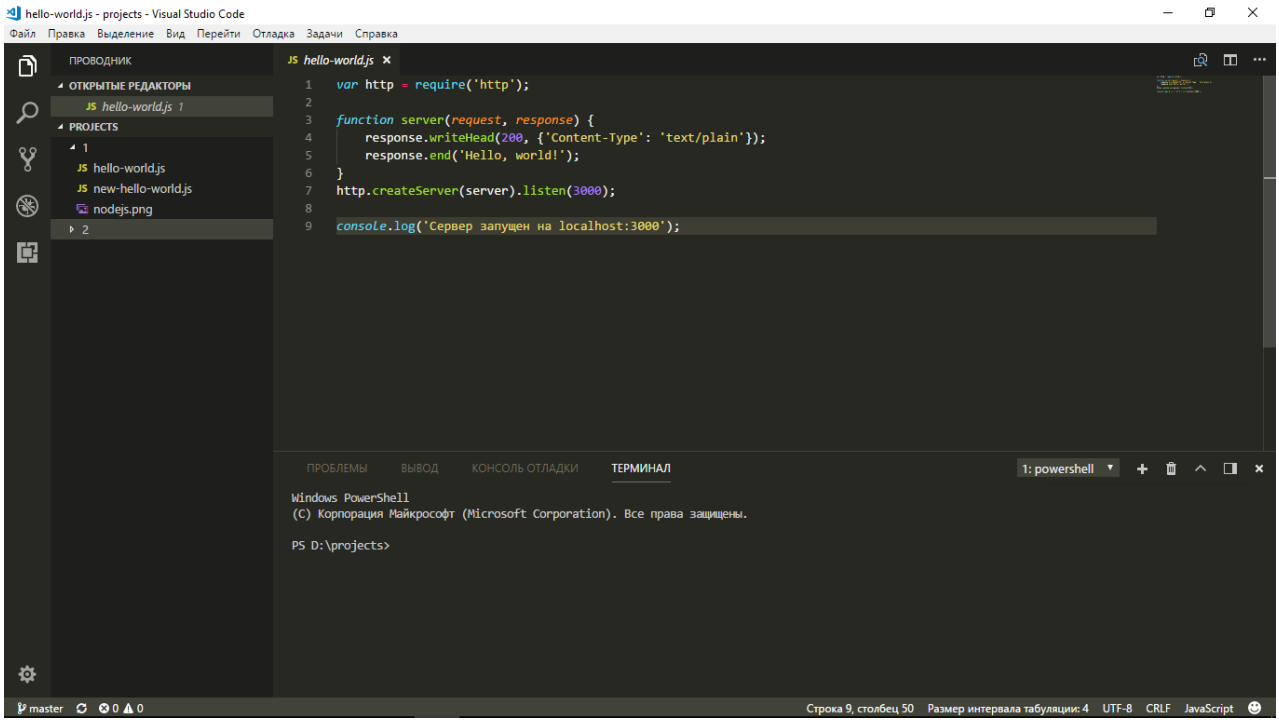


Рисунок 5.1 – Интерфейс *Visual Studio Code*

Одна из идей веб-сервера (например, *Apache*) – выдавать статические файлы таким образом, что браузер может видеть их по сети. Например, если создать файл `about.html` и поместить его в соответствующий каталог, то можно затем перейти по адресу *http://localhost/about.html*. В зависимости от настроек веб-сервера можно даже опустить `.html`, но связь между *URL* и именем файла очевидна: веб-сервер просто знает, где на компьютере находится файл, и выдает его браузеру.

Node.js предлагает парадигму, отличную от парадигмы обычного веб-сервера: создаваемое вами приложение и является веб-сервером. *Node.js* просто обеспечивает вас фреймворком для создания веб-сервера. Создадим файл `hello-world.js`, как показано в листинге 5.3.

Листинг 5.3 – Создание сервера

```
var http = require('http');
function server(request, response) {
    response.writeHead(200, {'Content-Type':
'text/plain'});
```

```
    response.end('Hello, world!');
}
http.createServer(server).listen(3000);
console.log('Сервер запущен на localhost:3000');
```

Для запуска кода на исполнение необходимо перейти в каталог с данным файлом и ввести команду, представленную в листинге 5.4.

Листинг 5.4 – Запуск сервера

```
Node hello-world.js
```

После чего необходимо ввести в адресную строку то, что показано на рисунке 5.2.

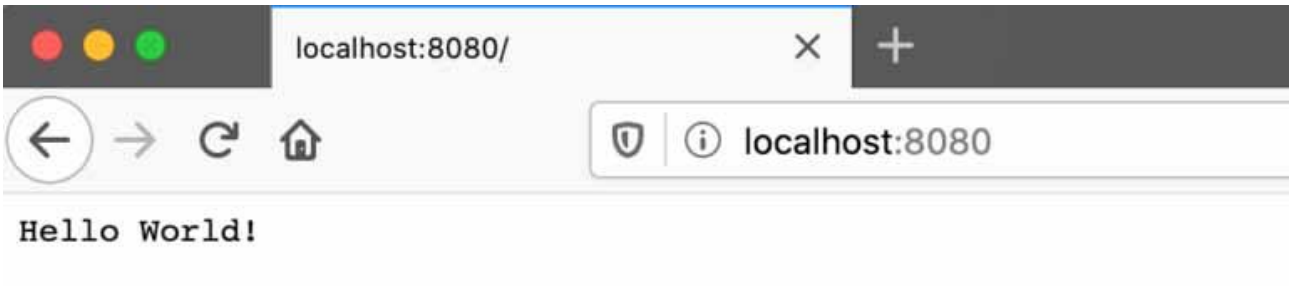


Рисунок 5.2 – Демонстрация работы сервера

Пояснение:

– `request` – объект запроса («`request`»), то есть то, что прислал клиент (обычно браузер), из него читаем данные;

– `response` – объект ответа («`response`»), в него пишем данные в ответ клиенту;

– вызов `response.writeHead` (*HTTP*-код, `[строка статуса]`, `{заголовки}`) пишет заголовки;

– вызов `response.end(txt)` – завершает запрос ответом.

Чтобы завершить программу, необходимо закрыть окно терминала или нажать *Ctrl + C*. Когда запущено приложение, оно выполняется в активном (*foreground*) режиме. Это означает, что во время его выполнения нельзя ввести в терминале никакую другую команду.

JavaScript создает веб-сервер, который при обращении к нему из браузера выводит веб-страницу со словами «*Hello, world!*». Он демонстрирует несколько ключевых компонентов приложения *Node.js*.

Сначала программа включает необходимый для запуска простого сервера *HTTP*-модуль с подходящим именем *HTTP*. Внешняя функциональность *Node.js* подключается при помощи модулей, экспортирующих определенные типы функциональности, которая может использоваться в приложении (или другом модуле). Модули очень похожи на библиотеки в других языках программирования. Модуль *HTTP* входит в число базовых модулей *Node.js*, поэтому не требует отдельной установки через `npm`.

Модуль импортируется командой *require*, а результат присваивается локальной переменной. После импортирования локальная переменная может использоваться для создания экземпляра веб-сервера функцией `http.createServer()`. В параметрах функции встречается одна из фундаментальных конструкций *Node.js*: функция обратного вызова (`callback`). Эта анонимная функция передает веб-запрос и ответ коду, который обрабатывает веб-запрос и предоставляет ответ на порту 3000.

JavaScript – однопоточный язык, и для имитации асинхронного выполнения в *Node.js* используется цикл событий, а функции обратного вызова вызываются при срабатывании определенного события.

Сообщение `console.log()` выводится на терминал сразу же при вызове создания сервера. Программа не прекращает работу в блокирующем режиме, ожидая веб-запроса.

Доступ к оперативной памяти (*RAM*) выполняется намного быстрее, чем к файловой системе. Поэтому *Node.js*-приложения обычно кэшируют часто используемые данные в оперативной памяти.

После того как сервер будет создан и получит запрос, функция обратного вызова передает браузеру простой текстовый заголовок с кодом статуса 200, выводит сообщение «*Hello, world!*» и завершает ответ.

Простой вывод статического сообщения, во-первых, демонстрирует, что приложение работает, а во-вторых, показывает, как создать простой веб-сервер. Базовый пример также демонстрирует некоторые ключевые элементы всех приложений *Node.js*.

Обновим код с обработкой входящего запроса. Имя выделяется из строки и используется для определения типа возвращаемого контента. Почти для любого имени будет возвращен персонализированный ответ, но, если использовать в запросе параметр `name=picture`, то в итоге можно получить изображение. Если строка запроса не используется или в ней не указано имя, переменной `name` присваивается значение «`world`», как показано в листинге 5.5.

Листинг 5.5 – Обработка входящего запроса

```
var http = require('http');
var fs = require('fs');
function server(request, response) {
    var name = require('URL').parse(request.URL,
true).query.name;
    if (name === undefined) {
        name = 'world!';
    }
    if (name == 'picture') {
        var file = 'Nodejs.png';
        fs.stat(file, function (err, stat) {
            if (err) {
```

```
                    console.error(err);
                    response.writeHead(200, {'Content-
Type': 'text/plain'});
                    response.end("Извините, картинка от-
сутствует \n");
                } else {
                    var img = fs.readFileSync(file);
                    response.contentType = 'image/png';
                    response.contentLength = stat.size;
                    response.end(img, 'binary');
                }
            });
        } else {
            response.writeHead(200, {'Content-Type':
'text/plain'});
            response.end('Hello, ' + name + '\n');
        }
    }
    http.createServer(server).listen(3000);
    console.log('Сервер запущен на localhost:3000');
```

Для передачи параметра *picture* переменной *name*, в строку браузера необходимо ввести: *http://localhost:3000/?name=picture*.

Все, что указано в *URL* после символа «?», называется параметрами запроса «query parameters». Согласно указанным в браузере параметрам один сервер может выдавать разные страницы разным браузерам. Именно это свойство используется поисковыми системами для вывода пользователю динамической веб-страницы со списком информационных ресурсов согласно ключевым словам поиска (параметрам запроса).

В начале кода в приложение подключается новый модуль с именем fs. Это модуль файловой системы (File System). Но при этом в программе импортируется еще один модуль, но не так, как два других.

Свойства экспортируемых модулей могут объединяться в цепочки, что позволяет импортировать модуль и использовать его функции в одной строке. Это часто происходит в модуле *URL*, единственная цель которого – предоставить инструменты для работы с *URL*-адресами.

Имена параметров ответа (response) и запроса (request) для удобства часто сокращаются до res и req соответственно, но в данной лабораторной работе для ознакомления используется полное название. После разбора запроса для получения значения name программа сначала проверяет, равен ли параметр undefined. Если параметр не определен, используется значение по умолчанию world. Если же значение name существует, оно снова проверяется и сравнивается с picture. Если значения не совпадают, то приложение возвращает почти

такой же ответ, как в исходном приложении, не считая того, что в возвращаемое сообщение подставляется переданное имя. Но если параметр `name` содержит строку `picture`, это означает, что вместо текста придется иметь дело с изображением.

Если файл не существует, то приложение корректно обрабатывает ситуацию: оно выводит сообщение об ошибке и информацию на консоль, для чего на этот раз используется метод `console.error()`. Результат ошибки представлен на рисунке 5.3.

```
Сервер запущен на localhost:3000
{ Error: ENOENT: no such file or directory, stat 'D:\projects\1\nodejs.png'
  errno: -4058,
  code: 'ENOENT',
  syscall: 'stat',
  path: 'D:\\projects\\1\\nodejs.png' }
```

Рисунок 5.3 – Результат отсутствия файла в директории

Метод `fs.stat()` не только проверяет, что файл существует, но также возвращает объект с информацией о файле, включающей его размер. Значение используется для создания заголовка контента.

Если файл существует, то изображение загружается в переменную и возвращает его в ответе, изменяя значения в заголовке соответствующим образом.

Метод `fs.stat()` использует стандартный для *Node.js* паттерн функции обратного вызова с передачей значения ошибки в первом параметре. Часть загрузки изображения может показаться неожиданной (необычно выглядит и не похожа на другие функции *Node.js)*. Главное отличие – использование синхронной функции *readFileSync()* вместо асинхронной версии *readFile()*.

Node.js поддерживает как синхронную, так и асинхронную версию большинства функций файловой системы. Обычно использование синхронных операций в веб-запросах в *Node.js* считается крайне нежелательным, но такая возможность существует.

«Нормальный» способ чтения файлов в *Node.js* – это чтение асинхронным способом. Это значит, что вызывается команда чтения файла и передается `callback`, который будет вызван при завершении чтения. Это позволяет работать с несколькими запросами чтения параллельно. Для этого можно использовать метод `readFile()` из модуля `fs`.

Для начала загружаем модуль `fs` с помощью команды `require`. Затем вызываем метод `readFile()`, который получает три параметра: имя файла, кодировку файла и функцию. Эта функция будет вызвана, когда завершится операция чтения файла. Функция получит два параметра: первый – информация о каких-либо ошибках, второй – содержимое файла.

Кроме того, хотя в этом примере обработка исключений не используется, с синхронными функциями можно использовать конструкцию `try-catch`. С

асинхронными функциями эта более традиционная форма обработки ошибок невозможна (отсюда и передача кода ошибки в первом параметре анонимной функции обратного вызова).

Важное замечание: ввод/вывод в *Node.js* не всегда является асинхронным.

Порядок выполнения

1 Создайте *HTTP*-сервер на *Node.js*. Чтобы создать сервер, следует вызвать метод `http.createServer()`:

```
const http = require("http");
http.createServer().listen(3000);
```

Метод `createServer()` возвращает объект `http.Server`. Воспользуйтесь методом `listen()` для обработки входящих подключений. Для этого у объекта сервера необходимо вызвать метод, в который в качестве параметра передается номер порта, по которому запускается сервер.

2 Для обработки подключений методом `createServer` передайте специальную функцию:

```
const http = require("http");
http.createServer(function(request, response){
    response.end("Hello world!");
}).listen(3000);
```

Функция может принимать два параметра:

– `request` – хранит информацию о запросе;

– `response` – управляет отправкой ответа.

Получите с помощью параметра `request` информацию о запросе и представлении объекта `http.IncomingMessage`. Основные свойства этого объекта:

– `headers` – возвращает заголовки запроса;

– `method` – тип запроса (`GET`, `POST`, `DELETE`, `PUT`);

– `url` – представляет запрошенный адрес.

Определите файл `app.js`:

```
var http = require("http");
http.createServer(function(request, response){
    console.log("Url: " + request.url);
    console.log("Тип запроса: " + request.method);
    console.log("User-Agent:   "   +   request.head-
ers["user-agent"]);
    console.log("Все заголовки");
```

```
    console.log(request.headers);
    response.end();
}).listen(3000);
```

Запустите его и обратитесь в браузере по адресу *http://localhost:3000/index.html*.

Параметр `response` управляет отправкой ответа и представляет объект `http.ServerResponse`. Среди его функциональности можно выделить следующие методы:

– `statusCode` – устанавливает статусный код ответа;

– `statusMessage` – устанавливает сообщение, отправляемое вместе со статусным кодом;

– `setHeader(name, value)` – добавляет в ответ один заголовок;

– `write` – пишет в поток ответа некоторое содержимое;

– `writeHead` – добавляет в ответ статусный код и набор заголовков;

– `end` – сигнализирует серверу, что заголовки и тело ответа установлены, в итоге ответ отсылается клиента. Данный метод должен вызываться в каждом запросе.

Измените файл `app.js` следующим образом:

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("UserId", 12);
    response.setHeader("Content-Type",    "text/html;
charset=utf-8;");
    response.write("<h2>hello world</h2>");
    response.end();
}).listen(3000);
```

Запустите файл и обратитесь в браузере к приложению.

3 Используя метод `write()`, отправьте код полноценной веб-страницы. Для этого довольно большой ответ можно несколько раз вызвать и последовательно оправить в исходящий поток каждый кусочек информации. Рассмотрите сказанное на примере:

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("Content-Type", "text/html");
    response.write("<!DOCTYPE html>");
    response.write("<html>");
    response.write("<head>");
    response.write("<title>Изучаем      Node.js</ti-
tle>");
```

```
    response.write("<meta charset=\"utf-8\" />");
    response.write("</head>");
    response.write("<body><h2>Магистрант     кафедры
ПИКС изучает дисциплину ПИПИнфС</h2></body>");
    response.write("</html>");
    response.end();
}).listen(3000);
```

4 Изучите систему маршрутизации *Node.js* (по умолчанию она отсутствует). Обычно маршрутизация реализуется с помощью специальных фреймворков типа *Express* (в данной лабораторной работе не изучается). Однако если необходимо разграничить простейшую обработку двух-трех маршрутов, то вполне можно использовать для этого свойство `url` объекта `Request`. Рассмотрите пример реализации обработки трех маршрутов.

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("Content-Type",   "text/html;
charset=utf-8;");
    if(request.url  ===  "/bsuir"  ||  request.url  ===
"/"){
        response.write("<h2>Bsuir</h2>");
    }
    else if(request.url == "/piks"){
        response.write("<h2>PIKS</h2>");
    }
    else if(request.url == "/contact"){
        response.write("<h2>Contacts</h2>");
    }
    else{
        response.write("<h2>Not found</h2>");
    }
    response.end();
}).listen(3000);
```

Если идет обращение к корню сайта или по адресу *http://localhost:3000/bsuir*, то пользователю выводится строка «*Bsuir*». Ели обращение идет по адресу *http://localhost:3000/pks*, то пользователю в браузере отображается строка «*PIKS*» и так далее. Если запрошенный адрес не соответствует ни одному маршруту, то выводится заголовок «*Not Found*».

Замечание: в рамках специальных фреймворков, которые работают поверх *Node.js*, например, *Express*, есть более удобные способы для обработки маршрутов, которые нередко используются.

5 Изучите переадресацию, которая предполагает отправку статусного кода 301 (постоянная переадресация) или 302 (временная переадресация) и заголовка `Location`, который указывает на новый адрес. Например, выполним переадресацию с адреса *http://localhost:3000/* на адрес *http://localhost:3000/newpage* .

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("Content-Type",   "text/html;
charset=utf-8;");
    if(request.url === "/"){
        response.statusCode = 302; // временная пе-
реадресация на адрес localhost:3000/newpage
        response.setHeader("Location", "/newpage");
    }
    else if(request.url == "/newpage"){
        response.write("New address");
    }
    else{
        response.statusCode = 404; // адрес не найден
        response.write("Not Found");
    }
    response.end();
}).listen(3000);
```

6 Напишите программу, которая на страницу браузера выводит фразу «`Это моя лабораторная работа`» на порту `4000`.

7 Напишите программу, которая на страницу браузера с использованием *HTML*-тега `<h1>` выводит фразу «`Это моя лабораторная работа`» на порту `7000`.

8 Напишите программу, которая при обращении на страницу браузера выводит фразу «`Это моя лабораторная работа`» на порту `8000`, а в консоль выводит сгенерированное случайным образом предложение.

9 Напишите программу, которая на страницу браузера выводит картинку с расширением *JPG* на порту `8901`, а при отсутствии картинки выводит вместо нее аббревиатуру изучаемой дисциплины. В консоль вывести имя файла картинки, если она присутствует.

10 Напишите программу, которая на страницу браузера выводит анимированную картинку с расширением *GIF* на порту `2345`.

Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.

4 Реализация решения задач с подробным рассмотрением (описанием) решений.

5 Выводы.

6 Список использованных источников.

Контрольные вопросы

1 Что из себя представляет платформа *Node.js*?

2 Каковы основные преимущества *Node.js*?

3 Что такое `npm`? В каком файле `npm` хранит зависимости?

4 Перечислите основные команды для работы с менеджером пакетов `npm`.

5 Какой командой запускается написанный на *Node.js* сервер?

6 По какому адресу необходимо перейти в строке браузера для проверки работы локального сервера `hello-world` на порту `3000`?

7 Что означает символ «`?`» в адресе «*http://localhost:3000/?name=picture*»?

8 Какой модуль используется для работы с файловой системой устройства?

9 Что представляет собой `callback`?

10 Что представляет собой асинхронный и синхронный способ чтения файлов?

Список использованных источников

1 Buckler, Craig. Node.js: Novice to Ninja / C. Buckler. – SitePoint, 2022. – 388 p.

2 Erasmus, M. CoffeeScript Programming with jQuery, Rails, and Node.js / M. Erasmus. – Packt Publishing, 2012. – 140 p.

3 Node.js. Notes for Professionals. – GoalKicker.com, 2018. – 333 p.

4 Wilson, J. Node.js 8 the Right Way / J. Wilson. – Pragmatic Bookshelf, 2018. – 336 p.

5 Пауэрс, Ш. Изучаем Node. Переходим на сторону сервера / Ш. Пауэрс. – 2-е изд., доп. и перераб. – СПб. : Питер, 2017. – 304 с.

6 Янг, А. Node.js в действии / А. Янг, Б. Мек, М. Кантелон. – 2-е изд. – СПб. : Питер, 2018. – 432 с.

ЛАБОРАТОРНАЯ РАБОТА № 6

МОДУЛИ, ОБЪЕКТЫ И БУФЕР ОБМЕНА

Цель работы

Ознакомиться с глобальными объектами и научиться работать с буфером обмена.

Теоретические сведения

Node.js использует модульную систему. То есть вся встроенная функциональность разбита на отдельные пакеты или модули. Модуль представляет блок кода, который может использоваться повторно в других модулях. Глобальные объекты *Node.js* имеют глобальный характер и доступны во всех модулях. Данные объекты нет необходимости включать в приложение, поскольку их можно использовать напрямую. Эти объекты являются модулями, функциями, строками и самим объектом.

Два важнейших объекта в *Node.js* – `global` и `process`. Объект `global` немного похож на глобальный объект в браузере, хотя между ними существуют очень серьезные различия. Объект `process`, напротив, существует только в *Node.js*.

В браузере переменная, объявленная на верхнем уровне, объявляется глобально. В *Node.js* дело обстоит иначе. Переменная, объявленная в модуле или приложении *Node.js*, не обладает глобальной доступностью, она ограничивается модулем или приложением. Таким образом, можно объявить «глобальную» переменную с именем `str` в модуле и в приложении, использующем этот модуль, и никакого конфликта не будет.

Для демонстрации создадим простую функцию, которая прибавляет число к базовому значению и возвращает результат. Функция будет создана как библиотека *JavaScript* для использования в веб-странице, а также как модуль для использования в приложении *Node.js*.

Код библиотеки *JavaScript* из файла `add2.js` объявляет переменную `base`, присваивает ей значение `2`, после чего прибавляет переданное число (листинг 6.1).

Листинг 6.1 – Объявление переменной `base`

```
var base = 2;
function addtwo(input) {
return parseInt(input) + base;
}
```

Затем создадим очень простой модуль, который делает то же самое, но с использованием синтаксиса модуля *Node.js* (листинг 6.2).

Листинг 6.2 – Объявление переменной с использованием синтаксиса модуля *Node.js*

```
var base = 2;
exports.addtwo = function(input) {
    return parseInt(input) + base;
};
```

Рассмотрим, чем отличается концепция глобальности в двух разных средах. Библиотека `add2.js` используется в веб-странице, которая также объявляет переменную `base` (листинг 6.3).

Листинг 6.3 – Отличие концепции глобальности в двух разных средах

```
<!DOCTYPE html>
<html>
    <head>
        <script src="add2.js"></script>
        <script>
            var base = 10;
            console.log(addtwo(10));
        </script>
    </head>
<body>
</body>
</html>
```

При обращении к веб-странице в браузере на консоли браузера выводится значение `20` (вместо ожидаемого `12`). Дело в том, что все переменные, объявленные за пределами функции в *JavaScript*, добавляются в один глобальный объект. Объявляя новую переменную с именем `base` в веб-странице, происходит переопределение значения во включенном файле сценария.

Теперь используем модуль `addtwo` в приложении *Node.js* (листинг 6.4).

Листинг 6.4 – Применение модуля `addtwo` в приложении *Node.js*

```
var addtwo = require('./addtwo').addtwo;
var base = 10;
console.log(addtwo(base));
```

В приложении *Node.js* результат равен `12`. Объявление новой переменной `base` в приложении *Node.js* никак не повлияло на значение `base` в модуле, потому что они существуют в разных глобальных пространствах имен.

Устранение общего пространства имен – безусловное усовершенствование, но оно не универсально. Объект `global` во всех средах предоставляет доступ к глобально доступным объектам и функциям *Node.js*, включая объект `process`. Чтобы убедиться в этом, просто добавьте следующую команду в файл

и запустите приложение. Следующая команда выводит все глобально доступные объекты и функции (листинг 6.5).

Листинг 6.5 – Применение объекта `global`

```
console.log(global);
```

Объект `process` принадлежит к числу важнейших компонентов среды *Node.js*, так как он предоставляет информацию о среде выполнения. Кроме того, через объект `process` выполняется стандартный ввод/вывод, что позволяет корректно завершить приложение *Node.js* и выдать сигнал при завершении итерации в цикле событий *Node.js*.

Объект `process` предоставляет доступ к информации как о среде *Node.js*, так и о среде выполнения программы.

Для получения информации воспользуемся параметром командной строки `-p`, который выполняет сценарий и возвращает полученный результат. Например, для проверки свойства `process.versions` введите следующую команду (листинг 6.6).

Листинг 6.6 – Проверка свойств `process.versions`

```
$ node -p "process.versions"
```

Команда выводит список различных компонентов *Node.js* и зависимостей, включая версии `V8`, `OpenSSL` (библиотека, используемая для безопасной передачи данных), версию *Node.js* и так далее (листинг 6.7).

Листинг 6.7 – Список различных компонентов *Node.js*

```
{ http_parser: '2.5.0',
node: '4.2.1',
V8: '4.5.103.35',
uv: '1.7.5',
zlib: '1.2.8',
ares: '1.10.1-DEV',
icu: '56.1',
modules: '46',
openssl: '1.0.2d' }
```

Свойство `process.env` предоставляет информацию о том, что *Node.js* знает о текущей среде разработки (листинг 6.8).

Листинг 6.8 – Свойство `process.env`

```
$ node -p "process.env"
```

Информация о среде позволяет разработчику понять, что видит *Node.js* до и после разработки. Не рекомендуется включать зависимости от этих данных

прямо в приложение, потому что они могут различаться между версиями *Node.js*, а также между архитектурами.

Между версиями *Node.js* в любом случае должны остаться неизменными основные объекты и функции, важные для работы приложений. В их числе стандартный ввод/вывод и возможность корректного завершения приложения *Node.js*.

Стандартные потоки представляют собой заранее определенные каналы передачи данных между приложением и средой:

– стандартный ввод (`stdin`);

– стандартный вывод (`stdout`);

– стандартный поток ошибок (`stderr`).

В приложении *Node.js* эти каналы обеспечивают взаимодействие между приложением *Node.js* и терминалом, своего рода механизм прямого «общения» с приложением.

Node.js поддерживает каналы с тремя функциями `process`:

– `process.stdin` – поток с поддержкой чтения для `stdin`;

– `process.stdout` – поток с поддержкой записи для `stdout`;

– `process.stderr` – поток с поддержкой записи для `stderr`.

Возможность закрыть эти потоки в своем приложении отсутствует. Поддерживается только чтение данных из канала `stdin` и запись в каналы `stdout` и `stderr`.

Функции ввода/вывода `process` наследуют от `EventEmitter`. Это означает, что они могут генерировать события, а программа может перехватывать эти события и обрабатывать любые данные. Чтобы обработать входные данные с использованием `process.stdin`, прежде всего необходимо назначить кодировку потока. Если этого не сделать, можно получить результаты в виде буфера, а не в виде строки (листинг 6.9).

Листинг 6.9 – Применение функции `process.stdin`

```
process.stdin.setEncoding('utf8');
```

Затем начинается прослушивание события `readable`, которое сообщает о поступлении блока данных, готового к чтению. Функция «Поток для записи» представляет собой приемник `process.stdin`. `read()`, который используется для чтения данных, и если данные отличны от `null`, они копируются в `process.stdout` при помощи функции `process.stdout.write()` (листинг 6.10).

Листинг 6.10 – Применение функции `process.stdout.write()`

```
process.stdin.on('readable', function() {
    var input = process.stdin.read();
    if (input !== null) {
```

```
            process.stdout.write(input);
        }
});
```

Программа будет читать буфер и выводить буфер, но для пользователя приложения все выглядит так, словно идет работа с текстом (строка). На самом деле это не так. Следующая функция `process` демонстрирует эти различия.

Чтобы завершить программу, придется либо уничтожить процесс с использованием сигнала, либо нажать клавиши «*Ctrl + C*». Также возможен другой вариант: завершить приложение из кода при помощи `process.exit()`. Можно даже передать информацию о том, успешно ли завершилось приложение или произошла ошибка.

Если убрать вызов функции `process.stdin.setEncoding()` в начале кода, произойдет ошибка. Дело в том, что буферы не поддерживают функцию `trim()`. Можно преобразовать буфер в строку, а затем выполнить `trim` (листинг 6.11).

Листинг 6.11 – Преобразование буфера в строку и выполнение `trim`

```
var command = input.toString().trim();
```

Рекомендуется просто добавить команду назначения кодировки и исключить все неожиданные побочные эффекты.

Запись в объект `process.stderr` выполняется при возникновении ошибки. Данный объект нужен, чтобы отделить ожидаемый вывод от возникающих проблем. В некоторых системах вывод `stderr` даже может обрабатываться отдельно от `stdout` (например, сообщения `stdout` сохраняются в журнале, а вывод `stderr` выводится на консоль).

Буфер *Node.js* содержит низкоуровневые двоичные данные, которые были созданы за пределами кучи (`heap`) `V8`. Для управления буфером используется класс `Buffer`. После того как память будет выделена, изменить размер буфера не удастся.

Буфер по умолчанию используется для работы с файлами: если при чтении и записи в файл не была назначена конкретная кодировка, данные читаются или записываются в буфер.

В отличие от `ArrayBuffer` при создании нового буфера *Node.js* его содержимое не инициализируется. Можно избежать неприятных сюрпризов при работе с буфером, который может содержать всевозможные конфиденциальные данные, буфер лучше заполнить сразу же после его создания, как показано в листинге 6.12.

Листинг 6.12 – Заполнение буфера после создания

```
let buf = new Buffer(24);
buf.fill(0); // буфер заполняется нулями
```

Возможно частичное заполнение буфера с указанием начального и конечного значений.

Можно создать новый буфер напрямую, передав функции-конструктору массив октетов, другой буфер или строку. Буфер создается с копированием содержимого переданного объекта. Для строки, если она не хранится в кодировке UTF-8, необходимо указать кодировку; по умолчанию строки *Node.js* хранятся в кодировке UTF-8 (utf8 или utf-8) (листинг 6.13).

Листинг 6.13 – Создание нового буфера

```
let str = 'New String';
let buf = new Buffer(str);
```

Существуют следующие методы Buffer для создания буферов: Buffer.from(), Buffer.alloc() и Buffer.allocUnsafe().

При передаче массива функция Buffer.from() возвращает буфер с копией содержимого. Однако, если передать массив функции ArrayBuffer с необязательными параметрами смещения и длины, буфер использует ту же память, что и ArrayBuffer. При передаче буфера копируется содержимое буфера, а при передаче строки происходит копирование строки.

Функция Buffer.alloc() создает заполненный буфер определенного размера, а Buffer.allocUnsafe() создает буфер определенного размера, который может содержать старые или конфиденциальные данные и в дальнейшем должен быть заполнен вызовом buf.fill().

Буфер можно преобразовать в формат JSON и в строковую форму (листинг 6.14).

Листинг 6.14 – Преобразование буфера

```
"use strict";
let buf = new Buffer('This is my pretty example');
let json = JSON.stringify(buf);
console.log(json);
```

В формате JSON сначала указывается тип преобразуемого объекта Buffer, а затем следуют его данные.

Для чтения и записи содержимого буфера с заданным смещением применяются различные функции. Примеры использования этих функций представлены в следующем фрагменте, который записывает в буфер четыре 8-разрядных целых числа без знака, а затем снова читает и выводит их (листинг 6.15).

Листинг 6.15 – Вывод буфера в строковом виде

```
var buf = new Buffer(4); // запись значений в буфер
buf.writeUInt8(0x63, 0);
buf.writeUInt8(0x61, 1);
buf.writeUInt8(0x74, 2);
```

```
    buf.writeUInt8(0x73, 3); // вывод буфера в строковом
виде
    console.log(buf.toString());
```

Node.js поддерживает чтение и запись 8-, 16- и 32-разрядных целых чисел со знаком и без, а также вещественных чисел одинарной и двойной точности. Для всех типов, кроме 8-разрядных целых чисел, также можно выбрать формат с прямым (`little-endian`) или обратным (`big-endian`) порядком байтов. Несколько примеров поддерживаемых функций:

– `buffer.readUIntLE()` – чтение значения с заданным смещением и прямым порядком байтов;

– `buffer.writeUInt16BE()` – запись 16-разрядного целого без знака с заданным смещением и обратным порядком байтов;

– `buffer.readFloatLE()` – чтение вещественного числа одинарной точности с заданным смещением и прямым порядком байтов;

– `buffer.writeDoubleBE()` – запись 64-разрядного вещественного числа двойной точности с заданным смещением и обратным порядком байтов.

Кроме чтения и записи по заданному смещению можно создать срез – новый буфер, содержащий часть старых данных, вызовом `buffer.slice()`. В этой возможности особенно интересно то, что изменение содержимого нового буфера также приводит к изменению содержимого старого буфера (листинг 6.16).

Листинг 6.16 – Создание нового буфера как среза старого

```
    var buf1 = new Buffer('this is the way we build our
buffer');
    var lnth = buf1.length;
    // создание нового буфера как среза старого
    var buf2 = buf1.slice(19, lnth);
    console.log(buf2.toString()); // build our buffer
    //изменение второго буфера
    buf2.fill('*', 0, 5);
    console.log(buf2.toString()); // ***** our buffer
    // проверка содержимого первого буфера
    console.log(buf1.toString()); // this is the way we
***** our buffer
```

В данном примере для демонстрации этой особенности на базе строки создается буфер, срез существующего буфера используется для создания нового, после чего содержимое нового буфера модифицируется. Далее оба буфера выводятся на консоль, чтобы можно было убедиться в изменении данных «на месте».

Для сравнения байтов используется метод `buffer.compare()`, который возвращает признак относительного расположения сравниваемых буферов в лексикографическом порядке. Если сравниваемый буфер предшествует второму, то

возвращается значение «−1»; если нет – значение «1». Если два буфера содержат одинаковые байты, то возвращается «0» (листинг 6.17).

Листинг 6.17 – Применение метода `buffer.compare()`

```
var buf1 = new Buffer('1 is number one');
var buf2 = new Buffer('2 is number two');
var buf3 = new Buffer(buf1.length);
buf1.copy(buf3);
console.log(buf1.compare(buf2)); // -1
console.log(buf2.compare(buf1)); // 1
console.log(buf1.compare(buf3)); // 0
```

Существует еще один класс буфера `SlowBuffer`, который может использоваться, если потребуется сохранить содержимое небольшого буфера в течение продолжительного времени. Обычно *Node.js* выделяет память для небольших буферов (менее 4 Кбайт) в заранее выделенном блоке памяти. Это делается для того, чтобы механизму очистки памяти не приходилось отслеживать и освобождать большое количество мелких блоков памяти.

Класс `SlowBuffer` позволяет создавать мелкие буферы за пределами заранее выделенного блока памяти, чтобы они могли существовать в течение более долгого периода времени. Впрочем, как нетрудно представить, этот класс может существенно повлиять на быстродействие программы. Используйте его только в том случае, если никакое другое решение не подходит.

Порядок выполнения

1 Используя модуль `os`, проверьте информацию об окружении и операционной системе:

```
const os = require('os')
// получим имя текущего пользователя
let userName = os.userInfo().username
console.log(userName)
```

2 Для дальнейшей работы с модулями и объектами создайте для приложения каталог на жестком диске, назвав его своей фамилией, например, *C:\node\ivanov*. В этом каталоге создайте файл `app.js`.

```
const http = require('http')
http.createServer(function (request, response) {
    response.end('Привет магистрант!')
  })
  .listen(3000, '127.0.0.1', function () {
    console.log(
```

```
      'Сервер начал прослушивание запросов на порту
3000'
    )
  })
```

Разберите самостоятельно этот код. В случае возникновения затруднений, воспользуйтесь подсказкой ниже.

В первой строке создается модуль `http`, который необходим для создания сервера. Это встроенный модуль, и для его загрузки необходимо применить функцию `require()`, которая запишется в виде кода:

```
const http = require('http')
```

С помощью метода `createServer()` создается новый сервер для прослушивания входящих подключений и обработки запросов. В качестве параметра этот метод принимает функцию, которая имеет два параметра. Первый параметр (`request`) хранит всю информацию о запросе, а второй параметр (`response`) используется для отправки ответа. В нашем случае ответ представляет простую строку `Привет магистранта!` и отправляется с помощью метода `response.end()`.

Метод `http.createServer()` только создает сервер. Чтобы сервер начал прослушивать входящие подключения, у него надо вызвать метод `listen`:

```
.listen(3000, "127.0.0.1",function(){
    console.log("Сервер начал прослушивание запросов
на порту 3000");
});
```

Этот метод принимает три параметра.

Первый параметр указывает на локальный порт, по которому запускается сервер.

Второй параметр указывает на локальный адрес. То есть в данном случае сервер будет запускаться по адресу `127.0.0.1` или `localhost` на порту `3000`.

Третий параметр представляет функцию, которая запускается при начале прослушивания подключений. Здесь эта функция просто выводит диагностическое сообщение на консоль.

Запустите сервер. Для этого откройте командную строку в *Windows*. С помощью команды `cd` перейдем к каталогу приложения:

```
cd C:\node\ivanov
Затем вызовем следующую команду:
node app.js
```

Поясните, что произошло.

3 Создайте свои модули. Так, в п. 2 он состоял из файла `app.js`, в котором создавался сервер, обрабатывающий запросы. Добавьте в тот же каталог новый файл `greeting.js` и определите в нем следующий код:

```
console.log('greeting module')
```

В файле `app.js` подключите модуль:

```
const greeting = require('./greeting')
```

В отличие от встроенных модулей для подключения своих модулей надо передать в функцию `require` относительный путь с именем файла (расширение файла необязательно):

```
const greeting = require('./greeting')
```

Запустите приложение. На консоль выведится строка, которая определена в файле `greeting.js`.

4 Измените файл `greeting.js`:

```
let currentDate = new Date()
module.exports.date = currentDate
module.exports.getMessage = function (name) {
  let hour = currentDate.getHours()
  if (hour > 16) return 'Добрый вечер, ' + name
  else if (hour > 10) return 'Добрый день, ' + name
  else return 'Доброе утро, ' + name
```

В данном коде определена переменная `currentDate`. Поясните, почему извне она недоступна, а доступна только в пределах данного модуля.

Чтобы переменные или функции модуля были доступны, необходимо определить их в объекте `module.exports`. Поясните, зачем нужен объект `module.exports` и что выполняет функция `require()`.

Объект `module` представляет ссылку на текущий модуль, а его свойство `exports` определяет все свойства и методы модуля, которые могут быть экспортированы и использованы в других модулях.

Определите свойство `date` и метод `getMessage`.

5 Измените файл `app.js`:

```
const os = require('os')
const greeting = require('./greeting')
```

```
// получим имя текущего пользователя
let userName = os.userInfo().username
console.log(`Дата запроса: ${greeting.date}`)
console.log(greeting.getMessage(userName))
```

Теперь все экспортированные методы и свойства модуля доступны по имени: `greeting.date` и `greeting.getMessage()`.

Перезапустите приложение.

6 Определите в модуле сложные объекты или функции конструкторов, которые затем используются для создания объектов. Добавьте в папку проекта новый файл `user.js`:

```
function User(name, age) {
  this.name = name
  this.age = age
  this.displayInfo = function () {
    console.log(`Имя:    ${this.name}        Возраст:
${this.age}`)
  }
}
User.prototype.sayHi = function () {
  console.log(`Привет, меня зовут ${this.name}`)
}
module.exports = User
```

В данном случае определена стандартная функция конструктора `User`, которая принимает два параметра. При этом весь модуль теперь указывает на эту функцию конструктора:

```
module.exports = User
```

Подключите и используйте этот модуль в файле `app.js`:

```
const User = require('./user.js')
let eugene = new User('Eugene', 32)
eugene.sayHi()
```

Запустите приложение.

7 Рассмотрите некоторые аспекты работы с модулями в *Node.js*. Подключаемые модули кэшируются. В частности, в файле есть такие строки:

```
var filename = Module._resolveFilename(
  request,
  parent,
```

```
  isMain
)

var cachedModule = Module._cache[filename]
if (cachedModule) {
  updateChildren(parent, cachedModule, true)
  return cachedModule.exports
}
```

Необходимо учитывать дуализм: с одной стороны, это увеличивает производительность, а с другой – может создать некоторые проблемы, если мы не будем учитывать этот аспект. Например, возьмите рассмотренный выше код, где в главный файл приложения `app.js` подключается модуль `greeting.js`. Измените файл `greeting.js` следующим образом:

```
module.exports.name = 'Alice'
```

В файле определена только одна строка, которая устанавливает свойство `name`.

Измените код файла `app.js`:

```
var greeting1 = require('./greeting.js')
console.log(`Hello ${greeting1.name}`) //Привет, со-
сед
var greeting2 = require('./greeting.js')
greeting2.name = 'Виктор'
console.log(`Hello ${greeting2.name}`) //Привет, Вик-
тор
// greeting1.name тоже изменилось
console.log(`Hello ${greeting1.name}`) //Привет, Вик-
тор
```

Несмотря на то что здесь два раза получаем модуль с помощью функции `require`, обе переменных – `greeting1` и `greeting2` – будут указывать на один и тот же объект.

8 Изучите структуру модулей.

Пояснение: нередко модули приложения образуют какие-то отдельные наборы или области. Такие наборы модулей лучше помещать в отдельные каталоги. Например, создайте в каталоге приложения подкаталог `welcome`, а в нем создайте три новых файла:

– `index.js`;
– `morning.js`;
– `evening.js`.

Тогда общая структура проекта будет выглядеть следующим образом:
– welcome;
– index.js;
– morning.js;
– evening.js;
– app.js;
– greeting.js.
В файл morning.js поместите следующую строку:

```
module.exports = 'Доброе утро, магистрант'
```

Аналогично изменим файл evening.js:

```
module.exports = 'Добрый вечер, магистрант'
```

Эти два файла определяют сообщения приветствия в зависимости от времени суток.
Определите в файле index.js следующий код:

```
const morning = require('./morning')
const evening = require('./evening')
module.exports = {
  getMorningMessage: function () {
    console.log(morning)
  },
  getEveningMessage: function () {
    console.log(evening)
  },
}
```

В модуле определен объект, который имеет две функции для вывода приветствий. Используйте этот модуль в файле app.js:

```
const welcome = require('./welcome')
welcome.getMorningMessage()
welcome.getEveningMessage()
```

Несмотря на то что нет такого файла, как welcome.js, если в проекте есть каталог, который содержит файл с именем index.js, то можно обращаться к модулю по имени каталога, как в данном случае.
Запустите приложение на консоль и убедитесь, что будут выведены оба приветствия.
9 Изучите объект global и глобальные переменные.

Примените специальный объект global для предоставления доступа к глобальным, то есть доступным из каждого модуля приложения, переменным и функциям. Примерным аналогом данного объекта в javascript для браузера является объект window. Все доступные глобальные объекты можно посмотреть в документации.

Для примера создайте следующий модуль greeting.js:

```
let currentDate = new Date();
global.date = currentDate;
module.exports.getMessage = function () {
  let hour = currentDate.getHours();
  if (hour > 16) return 'Добрый вечер, магистрант' +
global.name;
  else if (hour > 10) return 'Добрый день, магистрант'
+ name;
  else return 'Доброе утро, магистрант' + name;
};
```

В данном случае происходит установка глобальной переменной date: global.date = currentDate. В самом модуле получаем глобальную переменную name, которая будет установлена извне. При этом обратиться к глобальной переменной name можно через объект global: global.name либо просто через имя name, так как переменная глобальная.

Определите следующий файл приложения app.js:

```
const greeting = require('./greeting');
global.name = 'Eugene';
global.console.log(date);
console.log(greeting.getMessage());
```

Через данный код установлена глобальная переменная name, которую получили в модуле greeting.js. На консоль выведена глобальная переменная date. Причем все глобальные функции и объекты, например, console, также доступны внутри global, поэтому можно написать и global.console.log(), и просто console.log().

Запустим файл app.js.

10 Рассмотрите создание буферов. Node Buffer может быть создан различными способами.

Способ 1

Синтаксис для создания непосвященного буфера из 10 октетов:

```
var buf = new Buffer(10);
```

Способ 2

Синтаксис для создания буфера из данного массива:

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

Способ 3

Синтаксис для создания буфера из заданной строки и (необязательно) типа кодирования:

```
var buf = new Buffer("Simply Easy Learning", "utf-
8");
```

Можно использовать любую кодировку, например, «ascii», «utf8», «utf16le», «ucs2», «base64» или «hex», а не только «utf8», которая является кодировкой по умолчанию.

11 Рассмотрите запись в буферы. Синтаксис метода для записи в Node Buffer:

```
buf.write(string[, offset][, length][, encoding])
```

Описание используемых параметров:

– строка – это строковые данные, которые будут записаны в буфер;

– смещение – это индекс буфера, с которого начинается запись. Значение по умолчанию 0;

– длина – это число байтов для записи. По умолчанию используется значение buffer.length;

– кодировка – кодировка для использования. utf8 – кодировка по умолчанию.

Содержание отчета

1 Титульный лист.

2 Цель работы.

3 Краткие теоретические сведения.

4 Реализация решения задач с подробным рассмотрением (описанием) решений.

5 Выводы.

6 Список использованных источников.

Контрольные вопросы

1 Что такое объект запроса?

2 Что позволяет выполнить специальный объект global?

3 В чем отличие глобальных объектов и глобальных переменных?

4 Что представляют собой буферы пространства фиксированной длины в памяти?

5 Что такое бинарные данные?

6 Как прочитать данные из буфера посредством просмотра их отдельных байтов?

7 Как прочитать данные из буфера при помощи методов `toString()` и `toJSON()`?

8 Для чего используются методы `write()` и `copy()`?

9 Как *Node.js* манипулирует бинарными данными?

10 Как осуществить различные способы кодирования символов, влияющие на то, как хранятся данные?

11 Как можно создать буферы из строковых данных, которые не кодируются в `UTF-8` или `ASCII`?

Список использованных источников

1 Buckler, C. Node.js: Novice to Ninja / C. Buckler. – SitePoint, 2022. – 388 p.

2 Erasmus, M. CoffeeScript Programming with jQuery, Rails, and Node.js / M. Erasmus. – Packt Publishing, 2012. – 140 p.

3 Node.js. Notes for Professionals. – GoalKicker.com, 2018. – 333 p.

4 Wilson, J. Node.js 8 the Right Way / J. Wilson. – Pragmatic Bookshelf, 2018. – 336 p.

5 Пауэрс, Ш. Изучаем Node. Переходим на сторону сервера / Ш. Пауэрс. – 2-е изд., доп. и перераб. – СПб. : Питер, 2017. – 304 с.

6 Янг, А. Node.js в действии / А. Янг, Б. Мек, М. Кантелон. – 2-е изд. – СПб. : Питер, 2018. – 432 с.

ЛАБОРАТОРНАЯ РАБОТА № 7

РАБОТА С ФАЙЛАМИ

Цель работы

Научиться работать с файлами и файловой системой сервера.

Теоретические сведения

Прежде чем взаимодействовать с файлами, находящимися в файловой системе сервера, необходимо получить дескриптор файла.

Дескриптор можно получить, воспользовавшись для открытия файла асинхронным методом `open()` из модуля `fs` (листинг 7.1):

Листинг 7.1 – Открытие файла асинхронным методом

```
const fs = require('fs')
fs.open('/Users/guest/test.txt', 'r', (err, fd) => {
    //fd - это дескриптор файла
})
```

Следует обратить внимание на второй параметр, `r`, использованный при вызове метода `fs.open()`. Это – флаг, который сообщает системе о том, что файл открывают для чтения. Могут применяться некоторые другие флаги, которые часто используются при работе с этим и некоторыми другими методами:

– `r+` – открыть файл для чтения и для записи;

– `w+` – открыть файл для чтения и для записи, установив указатель потока в начало файла. Если файл не существует – он создается;

– `a` – открыть файл для записи, установив указатель потока в конец файла. Если файл не существует – он создается;

– `a+` – открыть файл для чтения и записи, установив указатель потока в конец файла. Если файл не существует – он создается.

Файлы можно открывать и пользуясь синхронным методом `fs.openSync()`, который, вместо того чтобы предоставить дескриптор файла в коллбэке, возвращает его (листинг 7.2).

Листинг 7.2 – Открытие файла синхронным методом

```
const fs = require('fs')
try {
    const fd = fs.openSync('/Users/guest/test.txt',
'r')
} catch (err) {
    console.error(err)
}
```

После получения дескриптора любым из вышеописанных способов можно производить с ним необходимые операции.

С каждым файлом связан набор данных о нем, которые можно исследовать средствами *Node.js*. В частности, сделать это можно, используя метод `stat()` из модуля `fs`. Для этого вызывают этот метод, передавая ему путь к файлу, и после того, как *Node.js* получит необходимые сведения о файле, он вызовет коллбэк, переданный методу `stat()` (листинг 7.3).

Листинг 7.3 – Применение метода `stat()`

```
const fs = require('fs')
fs.stat('/Users/guest/test.txt', (err, stats) => {
    if (err) {
        console.error(err)
        return
    }
    //сведения о файле содержатся в аргументе `stats`
})
```

В *Node.js* имеется возможность синхронного получения сведений о файлах. При таком подходе главный поток блокируется до получения свойств файла (листинг 7.4).

Листинг 7.4 – Синхронное получение сведений о файле

```
const fs = require('fs')
try {
    const stats = fs.statSync('/Users/guest/test.txt')
} catch (err) {
    console.error(err)
}
```

Информация о файле попадет в константу `stats`. В данной информации соответствующий объект предоставляет большое количество полезных свойств и методов:

– методы `isFile()` и `isDirectory()` позволяют узнать, является ли исследуемый файл обычным файлом или директорией соответственно;

– метод `isSymbolicLink()` позволяет узнать, является ли файл символической ссылкой;

– размер файла можно узнать, воспользовавшись свойством `size` (листинг 7.5).

Листинг 7.5 – Информация о файле

```
const fs = require('fs')
fs.stat('/Users/guest/test.txt', (err, stats) => {
    if (err) {
```

```
        console.error(err)
        return
    }
    stats.isFile() //true
    stats.isDirectory() //false
    stats.isSymbolicLink() //false
    stats.size //1024000 //= 1MB
})
```

Путь к файлу – это адрес того места в файловой системе, где он расположен. В *Windows* этот путь выглядит как *C:\users\guest\file.txt*.

В *Node.js* есть стандартный модуль `path`, предназначенный для работы с путями к файлам. Перед тем как использовать этот модуль в программе, его надо подключить (листинг 7.6).

Листинг 7.6 – Подключение модуля `path`

```
const path = require('path')
```

Если есть путь к файлу, то, используя возможности модуля `path`, можно в удобном для восприятия и дальнейшей обработки виде узнать подробности об этом пути (листинг 7.7).

Листинг 7.7 – Подробности о пути `path`

```
const notes = '/users/guest/notes.txt'
path.dirname(notes) // /users/guest
path.basename(notes) // notes.txt
path.extname(notes) // .txt
```

Здесь, в строке `notes`, хранится путь к файлу. Для рассмотрения пути использованы следующие методы модуля `path`:

– `dirname()` – возвращает родительскую директорию файла;

– `basename()` – возвращает имя файла;

– `extname()` – возвращает расширение файла.

Методы `resolve()` и `normalize()` не проверяют существование директории. Они просто находят путь, основываясь на переданных им данных.

Одним из способов чтения файлов в *Node.js* является использование метода `fs.readFile()` с передачей ему пути к файлу и коллбэка (листинг 7.8).

Листинг 7.8 – Применение метода `fs.readFile()`

```
fs.readFile('/Users/guest/test.txt', (err, data) => {
    if (err) {
        console.error(err) return
    }
    console.log(data)
})
```

В случае необходимости можно воспользоваться синхронной версией этого метода – `fs.readFileSync()` (листинг 7.9).

Листинг 7.9 – Синхронная версия метода `fs.readFileSync()`

```
const fs = require('fs')
try {
    const data = fs.readFileSync('/Users/guest/test.txt')
console.log(data)
} catch (err) {
    console.error(err)
}
```

По умолчанию при чтении файлов используется кодировка `UTF-8`. Кодировку можно задать самостоятельно, передав методу соответствующий параметр.

Методы `fs.readFile()` и `fs.readFileSync()` считывают в память все содержимое файла. Это означает, что работа с большими файлами с применением этих методов серьезно отразится на потреблении памяти приложением и окажет влияние на его производительность. Если с такими файлами нужно работать, лучше всего воспользоваться потоками.

В *Node.js* легче всего записывать файлы с использованием метода `fs.writeFile()` (листинг 7.10).

Листинг 7.10 – Применение метода `fs.writeFile()`

```
const fs = require('fs')
const content = 'Some content!'
fs.writeFile('/Users/guest/test.txt', content, (err)
=> {
    if (err) {
        console.error(err)
        return
    }})
```

Есть и синхронная версия того же метода – `fs.writeFileSync()` (листинг 7.11).

Листинг 7.11 – Применение метода `fs.writeFileSync()`

```
const fs = require('fs')
const content = 'Some content!'
try {
    const data = fs.writeFileSync('/Users/guest/test.txt',
content)
} catch (err) {
    console.error(err)
}
```

Эти методы по умолчанию заменяют содержимое существующих файлов.

Метод `fs.appendFile()` (и его синхронную версию – `fs.appendFileSync()`) удобно использовать для присоединения данных к концу файла (листинг 7.12).

Листинг 7.12 – Применение метода `fs.appendFile()`

```
const content = 'Some content!'
fs.appendFile('file.log', content, (err) => {
    if (err) {
        console.error(err)
        return
    }
})
```

Модуль `fs` предоставляет в распоряжение разработчика много удобных методов, которые можно использовать для работы с директориями.

Для того чтобы проверить, существует ли директория и может ли *Node.js* получить к ней доступ, учитывая разрешения, можно использовать метод `fs.access()`.

Для того чтобы создавать новые папки, можно воспользоваться методами `fs.mkdir()` и `fs.mkdirSync()` (листинг 7.13).

Листинг 7.13 – Создание новой папки

```
const fs = require('fs')
const folderName = '/Users/guest/test'
try {
    if (!fs.existsSync(dir)) {
        fs.mkdirSync(dir)
    }
} catch (err) {
    console.error(err)
}
```

Для того чтобы прочесть содержимое папки, можно воспользоваться методами `fs.readdir()` и `fs.readdirSync()`.

Для переименования папки можно воспользоваться методами `fs.rename()` и `fs.renameSync()`. Первый параметр – это текущий путь к папке, второй – новый путь (листинг 7.14).

Листинг 7.14 – Переименование папки

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

fs.rename('/Users/guest', '/Users/guest2', (err) => {
    if (err) {
        console.error(err)
```

```
            return
        }
    })
```

Переименовать папку можно и с помощью использования синхронного метода `fs.renameSync()` (листинг 7.15).

Листинг 7.15 – Переименование папки с помощью синхронного метода `fs.renameSync()`

```
const fs = require('fs')
try {
    fs.renameSync('/Users/guest', '/Users/guest2')
} catch (err) {
    console.error(err)
}
```

Удаление папки, в которой что-то есть, является задачей несколько более сложной, чем удаление пустой папки. Если нужно удалить такие папки, то лучше воспользоваться пакетом `fs-extra`, который хорошо поддерживается. Он представляет собой замену модуля `fs`, расширяющую его возможности. Установить этот модуль можно так, как показано на листинге 7.16.

Листинг 7.16 – Установка модуля

```
npm install fs-extra
```

Рассмотрим пример его использования на листинге 7.17.

Листинг 7.17 – Пример использования модуля

```
const fs = require('fs-extra')
const folder = '/Users/guest'
fs.remove(folder, err => {
    console.error(err)
})
```

Особенностью модуля `fs` является тот факт, что все его методы по умолчанию являются асинхронными, но существуют и их синхронные версии, имена которых получаются путем добавления слова `Sync` к именам асинхронных методов.

Потоковая технология проявляется во всех базовых аспектах *Node.js*; она предоставляет функциональность для `HTTP`, а также для других форм сетевой передачи данных. Кроме того, она предоставляет функциональность для файловой системы.

Поток представлен абстрактным интерфейсом. Это означает, что потоки не будут создаваться напрямую. Вместо этого авторы предлагают магистранту работать с различными объектами, реализующими интерфейс `Stream`.

Так как многие объекты в *Node.js* реализуют потоковый интерфейс, все потоки в *Node.js* обладают базовой функциональностью:

– изменение кодировки потоковых данных вызовом `setEncoding`;

– проверка возможности чтения и (или) записи данных в поток;

– перехват событий потоков (например, получения данных или закрытия подключения) с назначением функций обратного вызова для каждого события;

– приостановка и возобновление потока;

– перенаправление данных из потока для чтения в поток для записи.

Следует обратить внимание на пункт с проверкой чтения и (или) записи. Поток с поддержкой чтения и записи называется дуплексным. Также существует подвид дуплексных потоков, называемый потоком преобразования данных, в котором ввод и вывод связаны причинной зависимостью.

Поток для чтения начинает работу в приостановленном состоянии. Это означает, что никакие данные не будут отправляться до того момента, пока не будет явно выполнена операция чтения (`stream.read()`) или команда возобновления работы потока (`stream.resume()`). Однако используемые реализации потоков, такие как поток для чтения модуля `File System`, переключаются в рабочий режим сразу же при программировании события данных (механизм получения доступа к данным в потоке для чтения). В рабочем режиме данные передаются приложению сразу же при их появлении.

Потоки для чтения поддерживают несколько событий, но на практике обычно используются три события: `data`, `end` и `error`.

Событие `data` отправляется при получении нового фрагмента данных, готового к использованию, а событие `end` – при потреблении всех данных. Событие `error` отправляется при возникновении ошибки.

Поток для записи представляет собой приемник, в который передаются (записываются) данные. Среди прослушиваемых событий можно выделить `error` и событие `finish`, происходящее при вызове `end()` и сбросе всех данных. Также встречается событие `drain`, генерируемое в тот момент, когда попытка записи данных возвращает `false`.

Дуплексный поток обладает качествами потоков как для чтения, так и для записи. Поток преобразования данных представляет собой дуплексный поток, в котором в отличие от обычных дуплексных потоков, где внутренние входные и выходные буферы существуют независимо друг от друга, эти два буфера связаны напрямую через промежуточный этап преобразования данных. Во внутренней реализации поток преобразования данных должен реализовать функцию `transform()`, которая получает входные данные, взаимодействует с ними, а затем записывает в вывод.

Чтобы лучше понять суть потока преобразования данных, необходимо поближе познакомиться с функциональностью, поддерживаемой всеми потоками: функцией `pipe()` (листинг 7.18).

Листинг 7.18 – Суть потока преобразования данных

```
// создание и перенаправление потока для чтения
var file = fs.createReadStream(pathname);
file.on("open", function () {
    file.pipe(res);
});
```

Вызов `pipe()` извлекает данные из файла (поток) и выводит их в объект `http.ServerResponse`. В документации *Node.js* указано, что этот объект реализует интерфейс потока для записи и `fs.createReadStream()` возвращает `fs.ReadStream` – реализацию потока для чтения. В число методов, поддерживаемых потоком для чтения, входит и `pipe()` с потоком для записи.

Порядок выполнения

1 Для работы с файлами в *Node.js* используется встроенный модуль `fs`, который выполняет все синхронные и асинхронные операции ввода/вывода применительно к файлам.

2 Опишите, как может осуществляться чтение и запись файлов с использованием `Buffer` и через создание соответствующего потока.

3 Рассмотрите чтение файлов и директорий.

Для чтения файла в асинхронном режиме используйте метод *Node.js* `readFile()`, который принимает три параметра:

– путь к файлу;

– кодировка;

– `callback`-функция, вызываемая после получения содержимого файла.

Запишите код:

```
fs.readFile('files/data.txt', 'utf8', (err, data) =>
{
  if (err) throw err;
  console.log(data);
});
```

Пояснение:

– `Callback`-функции передается два аргумента: ошибка и полученные данные в строковом формате. Если операция прошла успешно, то в качестве ошибки передается `null`;

– если в `readFile()` не указать кодировку, то данные файла будут возвращены в формате `Buffer`;

– поскольку метод выполняется асинхронно, то не происходит блокировки главного процесса *Node.js*. С помощью метода `readFileSync()` можно выполнить синхронное чтение файла, но при этом выполнение главного процесса будет

заблокировано до тех пор, пока полностью не будет загружено содержимое файла:

```
const content = fs.readFileSync('files/data.txt',
'utf8');
console.log(content);
```

– в *Node.js* метод `readFileSync()` возвращает результат чтения файла и принимает два параметра: путь к файлу и кодировку;

– обработка и перехват ошибок при использовании `readFileSync()` осуществляется с помощью конструкции `try{...}catch(){...}`:

```
try {
  const content = fs.readFileSync('files/data.txt',
'utf8');
  console.log(content);
} catch (e) {
  console.log(e);
}
```

Чтобы инициировать ошибку, укажите неправильный путь к файлу.

4 Выполните считывание содержимого файла с помощью создания потока `fs.createReadStream()`. Любой поток в *Node.js* является экземпляром класса `EventEmitter`, который позволяет обрабатывать возникающие в потоке события.

Параметры, принимаемые `fs.createReadStream()`:

– путь к файлу;

– объект со следующими настройками:

`encoding` – кодировка (по умолчанию `utf8`);

`mode` – режим доступа (по умолчанию `0o666`);

`autoClose` – если `true`, то при событиях `error` и `finish` поток закроется автоматически (по умолчанию `true`).

Код:

```
const stream = fs.createReadStream(
  'files/data.txt',
  'utf8'
);
stream.on('data', (data) => console.log(data));
stream.on('error',   (err)   =>   console.log(`Err:
${err}`));
```

Вместо объекта настроек можно передать строку, которая будет задавать кодировку.

5 Рассмотрите использование методов `readdir()` и `readdirSync()` для асинхронного и синхронного режимов соответственно.

Метод `readdir()` работает асинхронно и принимает три аргумента:

– путь к директории;

– кодировку;

– `callback`-функцию, которая принимает аргументами ошибку и массив файлов директории (при успешном выполнении операции ошибка передается как `null`).

Код:

```
fs.readdir('files', 'utf8', (err, files) => {
  if (err) throw err;
  console.log(files);
});
```

Метод `readdirSync()` работает синхронно, возвращает массив найденных файлов и принимает два параметра:

– путь к директории;

– кодировку.

Код:

```
try {
  const files = fs.readdirSync('files', 'utf8');
  console.log(files);
} catch (e) {
  console.log(e);
}
```

6 Рассмотрите отправку файлов в приложении на *Node.js*.

Отправка статических файлов – довольно частая задача в построении и функционировании веб-приложения.

Проверьте, чтобы в каталоге было три файла: `app.js`, `about.html`, `index.html`.

Наряду с файлом приложения `app.js` определите два *html*-файла. В файле `index.html` запишите следующий код:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Главная</title>
    <meta charset="utf-8" />
```

```
  </head>
  <body>
    <h1>Главная</h1>
  </body>
  <html></html>
</html>
```

Аналогично определите код в файле `about.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>О сайте</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>О сайте</h1>
  </body>
  <html></html>
</html>
```

Задача будет заключаться в том, чтобы отправить их содержимое пользователю.

Первый способ

Для считывания файла может применяться метод `fs.createReadStream()`, который считывает файл в поток, и затем с помощью метода `pipe()` мы можем связать считанные файлы с потоком записи, то есть с объектом `response`. Итак, поместим в файл `app.js` следующий код:

```
const http = require('http')
const fs = require('fs')
http
  .createServer(function (request, response) {
    console.log(`Запрошенный адрес: ${request.url}`)
    // получаем путь после слеша
    const filePath = request.url.substr(1)
    // смотрим, есть ли такой файл
    fs.access(filePath, fs.constants.R_OK, (err) =>
{
      // если произошла ошибка - отправляем статус-
ный код 404
      if (err) {
        response.statusCode = 404
        response.end('Resourse not found!')
```

```
      } else {
        fs.createReadStream(filePath).pipe(response)
      }
    })
})
.listen(3000, function () {
  console.log('Server started at 3000')
})
```

Выполнение данного кода предполагает, что сначала получаем запрошенный адрес. Допустим, запрошенный адрес будет соответствовать напрямую пути к файлу на сервере. Затем с помощью асинхронной функции `fs.access` проверьте доступность файла для чтения.

Первый параметр функции – путь к файлу.

Второй параметр – опция, относительно которой проверяется доступ. В данном случае значение `fs.constants.R_OK` говорит о том, что проверяются права на чтение из файла.

Третий параметр функции – функция обратного вызова, которая получает объект ошибки. Если произошла ошибка (файл не доступен для чтения или вовсе не найден), посылаем статусный код `404`.

Для отправки файла применяется цепочка методов:

```
fs.createReadStream('some.doc').pipe(response)
```

Метод `fs.createReadStream("some.doc")` создает поток для чтения – объекта `fs.ReadStream`. Для получения данных из потока вызывается метод `pipe()`, в который передается объект интерфейса `stream.Writable` или поток для записи. А именно таким и является объект `http.ServerResponse`, который реализует этот интерфейс.

Запустите приложение и в браузере обратимся по адресу *http://localhost:3000/index.html*. Аналогично обратитесь по адресу *http://localhost:3000/about.htm*. Изучите появившиеся окна.

В рассматриваемом примере отправляются файлы `html`. Аналогично можно отправлять самые разные файлы. Например, создайте папку `public`. В ней создайте новый файл `styles.css` со следующим содержимым:

```
body {
  font-family: Verdana;
  color: rgb(48, 48, 92);
}
```

Примените эти стили на странице `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Главная</title>
    <meta charset="utf-8" />
    <link
      href="public/styles.css"
      rel="stylesheet"
      type="text/css"
    />
  </head>
  <body>
    <h1>Главная</h1>
  </body>
  <html></html>
</html>
```

Обратитесь к `index.html`.

Второй способ

Представляет чтение данных с помощью функции `fs.readFile()` и отправку с помощью метода `response.end()`:

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

http
  .createServer(function (request, response) {
    console.log(`Запрошенный адрес: ${request.url}`)
    // получаем путь после слеша
    const filePath = request.url.substr(1)
    fs.readFile(filePath, function (error, data) {
      if (error) {
        response.statusCode = 404
        response.end('Resourse not found!')
      } else {
        response.end(data)
      }
    })
  })
  .listen(3000, function () {
    console.log('Server started at 3000')
  })
```

7 Рассмотрите пример отправки статических файлов.

Вместо того чтобы определять код, который получает пользователь, напрямую в файле сервера, гораздо удобнее вынести все в отдельный `html`-файл. Рассмотрите, как можно отправлять статические файла (те же файлы `html`).

Создайте новый каталог. Определите в нем основной файл приложения `app.js`. Для статических файлов создайте отдельную папку `public`, в которую добавьте два файла `index.html` и `about.html`. Каталог должен выглядеть следующим образом:

– `app.js`;
– `public`;
– `about.html`;
– `index.html`.

В файле `index.html` определите какой-нибудь простейший `html`-код:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Главная</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Главная</h1>
  </body>
  <html></html>
</html>
```

Аналогично определите код в файле `about.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>О сайте</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>О сайте</h1>
  </body>
  <html></html>
</html>
```

Отправьте их содержимое пользователю. Для этого поместите в файл `app.js` следующий код:

```
const http = require('http')
const fs = require('fs')
http
  .createServer(function (request, response) {
    console.log(`Запрошенный адрес: ${request.url}`)
    if (request.url.startsWith('/public/')) {
      // получаем путь после слеша
      const filePath = request.url.substr(1)
      fs.readFile(filePath, function (error, data) {
        if (error) {
          response.statusCode = 404
          response.end('Resourse not found!')
        } else {
          response.setHeader('Content-Type',
'text/html')
          response.end(data)
        }
      })
    } else {
      // во всех остальных случаях отправляем строку
hello world!
      response.end('Привет, магистрант!')
    }
  })
  .listen(3000)
```

Если запрошенный адрес начинается с `/public/`, то с помощью метода `fs.readFile()` считываем нужный файл по пути и отправляем считанные данные клиенту. В остальных случаях отдаем строку «`Привет, магистрант!`».

Запустите приложение и в браузере обратитесь по адресу *http://localhost:3000*.

Содержание отчета

1 Титульный лист.

2 Цель работы.

3 Краткие теоретические сведения.

4 Реализация решения задач с подробным рассмотрением (описанием) решений.

5 Выводы.

6 Список использованных источников.

Контрольные вопросы

1 Что такое функции обратного вызова?
2 Что такое асинхронный метод?
3 Что такое параметр объекта ошибки?
4 Что такое данные, возвращенные при успешном выполнении операции?
5 Для чего используется метод `fs.open()`?
6 Для чего используется флаг при работе с файлом?
7 Для чего используется метод `.appendFile()`?
8 Какие преимущества имеет использование потока перед `Buffer`?

Список использованных источников

1 Buckler, C. Node.js: Novice to Ninja / C. Buckler. – SitePoint, 2022. – 388 p.

2 Erasmus, M. CoffeeScript Programming with jQuery, Rails, and Node.js / M. Erasmus. – Packt Publishing, 2012. – 140 p.

3 Node.js. Notes for Professionals. – GoalKicker.com, 2018. – 333 p.

4 Wilson, Jim. Node.js 8 the Right Way / J. Wilson. – Pragmatic Bookshelf, 2018. – 336 p.

5 Пауэрс, Ш. Изучаем Node. Переходим на сторону сервера / Ш. Пауэрс. – 2-е изд., доп. и перераб. – СПб. : Питер, 2017. – 304 с.

6 Янг, А. Node.js в действии / А. Янг, Б. Мек, М. Кантелон. – 2-е изд. – СПб. : Питер, 2018. – 432 с.

ЛАБОРАТОРНАЯ РАБОТА № 8

ВЗАИМОДЕЙСТВИЕ С БАЗАМИ ДАННЫХ

Цель работы

Освоить принципы взаимодействия *Node.js* и баз данных. Научиться выполнять простые операции по выборке данных и манипуляции данными.

Теоретические сведения

В хранилищах данных на базе файлов для хранения данных применяется файловая система. Разработчики зачастую используют этот тип хранилища для хранения параметров приложения, а также постоянных данных, надежность хранения которых не должна зависеть от перезагрузки приложения или сервера.

Чтобы создать приложение, нужно написать начальную логику, а затем определить вспомогательные функции, необходимые для выборки и сохранения задач.

Благодаря использованию файловой системы в качестве хранилища данных можно относительно просто и быстро повысить надежность хранения данных. С помощью этих хранилищ можно также выполнять конфигурирование приложений. Если данные параметров приложений хранятся в текстовом файле и закодированы в формате `JSON`, код может быть переработан таким образом, чтобы выполнять чтение из `JSON`-файла и его синтаксический разбор.

Системы управления реляционными базами данных (СУБД) позволяют организовать хранение сложной информации и предоставляют средства для выборки этой информации. Такие системы традиционно применяются в относительно высокоуровневых приложениях, таких как системы управления контентом, системы управления связями с заказчиками, корзины *Internet*-магазинов и так далее. При корректном использовании СУБД хорошо выполняют возложенные на них функции, но при работе с ними нужны специальные знания по администрированию баз данных и знание языка *SQL*, требуется также доступ к серверу базы данных, хотя существуют объектно-реляционные преобразователи (*Object-Relational Mapper, ORM*), *API*-интерфейсы которых способны генерировать *SQL*-код в фоновом режиме.

У разработчика есть множество вариантов выбора реляционных баз данных, хотя большинство обычно выбирает базы данных с открытым исходным кодом, поскольку для этих баз данных доступна поддержка, они хорошо работают и к тому же совершенно бесплатны. В данной лабораторной работе рассматриваются наиболее популярные полнофункциональные реляционные и нереляционные базы данных.

Одна из наиболее популярных баз данных в мире, *MySQL*, имеет широкую поддержку среди сообщества *Node*-разработчиков. Для начала работы с этой базой данных необходимо установить зависимость с помощью следующей команды в соответствии с листингом 8.1.

Листинг 8.1 – Установление зависимости

```
npm i mysql
```

MySQL – свободная реляционная система управления базами данных. Гибкость СУБД *MySQL* обеспечивается поддержкой большого количества типов таблиц.

В листинге 8.2 показана базовая настройка `HTTP`-сервера, соединения с базой данных *MySQL* и маршрутизация `HTTP`-запросов. Также происходит создание таблицы базы данных, если таблица еще не создана, и запуск `HTTP`-сервера, прослушивающего *IP*-адрес 127.0.0.1, соответствующий *TCP/IP*-порту с номером 3000. Все запросы между *Node.js* и *MySQL* выполняются с помощью функции `query`. При этом предполагается, что подключаемая база данных уже существует (листинг 8.2).

Листинг 8.2 – Базовая настройка `HTTP`-сервера

```
var http = require('http');
var mysql = require('mysql');
// подключение к MySQL-серверу
var db = mysql.createConnection({
    host: '127.0.0.1',
    user: 'myuser',
    password: 'mypassword',
    database: 'mydatabase'
});
var server = http.createServer(function (req, res) {
    switch (req.method) {
        // маршрутизация HTTP-запросов методом POST
        case 'POST':
            switch (req.URL) {
                case '/':
                    work.add(db, req, res);
                    break;
                case '/archive':
                    work.archive(db, req, res);
                    break;
                case '/delete':
                    work.delete(db, req, res);
                    break;
            }
```

```
                break;
                // маршрутизация HTTP-запросов методом GET
            case 'GET':
                switch (req.URL) {
                    case '/':
                        work.show(db, res);
                        break;
                    case '/archived':
                        work.showArchived(db, res);
                }
            break;
        }
    });
    db.query( // SQL-код создания таблицы
        "CREATE TABLE IF NOT EXISTS work (" +
        "id INT(10) NOT NULL AUTO_INCREMENT, " +
        "hours DECIMAL(5,2) DEFAULT 0, " +
        "date DATE, " +
        "archived INT(1) DEFAULT 0, " +
        "description LONGTEXT," +
        "PRIMARY KEY(id))",
        function (err) {
            if (err) throw err;
            console.log('Server started...');
            // запуск HTTP-сервера
            server.listen(3000, '127.0.0.1');
        }
    );
```

Теперь, когда приложение полностью определено, необходимо его запустить. Также предварительно необходимо создать базу данных `mydatabase` с помощью интерфейса администратора базы данных *MySQL*. Затем запустить приложение, введя в командной строке следующую команду (листинг 8.3).

Листинг 8.3 – Запуск приложения

```
Node timetrack_server.js
```

В окне веб-браузера перейти по адресу *http://localhost:3000/*, чтобы воспользоваться приложением.

Рассмотрим база данных *PostgreSQL*, которая является свободной объектно-реляционной системой управления базами данных (СУБД).

PostgreSQL базируется на языке *SQL* и поддерживает многие из возможностей стандарта *SQL*:2011.

Многие разработчики *Node.js*-приложений отдают предпочтение СУБД *PostgreSQL*. В отличие от *MySQL PostgreSQL* поддерживает рекурсивные запросы и многие специализированные типы данных.

В *PostgreSQL* используется множество методов стандартной аутентификации, таких как протокол *LDAP* (*Lightweight Directory Access Protocol*) и интерфейс *GSSAPI* (*Generic Security Services Application Program Interface*). Поддерживается синхронная репликация, обеспечивающая при необходимости масштабируемость или избыточность. Утрата данных, возможная при репликации такого типа, предотвращается путем верификации, выполняемой после завершения каждой операции с данными.

Для работы с базой данных (БД) *PostgreSQL* необходимо установить модуль `Node-postgres` с помощью представленной в листинге 8.4 команды.

Листинг 8.4 – Установка модуля `Node-postgres`

```
npm i pg
```

После установки модуля `Node-postgres` можно подключиться к *PostgreSQL* и выбрать запрашиваемую базу данных. Если пароль не задан, необходимо опустить часть кода `:mypassword` в строке подключения (листинг 8.5).

Листинг 8.5 – Подключение к *PostgreSQL*

```
    var pg = require('pg');
    var conString = "tcp://myuser:mypassword@lo-
calhost:5432/mydatabase";
    var client = new pg.Client(conString);
    client.connect();
```

Для выполнения запросов к базе данных применяется метод `query`. В примере кода, содержащемся в листинге 8.6, демонстрируется вставка строки в таблицу базы данных:

Листинг 8.6 – Вставка строки в таблицу базы данных

```
client.query(
    'INSERT INTO users ' +
    "(name) VALUES ('Mike')"
);
```

Символы подстановки (`$1`, `$2` и аналогичные) указывают место подстановки параметра.

Следующий пример кода демонстрирует вставку строки с помощью символов подстановки (листинг 8.7).

Листинг 8.7 – Вставка строки с помощью символов подстановки

```
client.query(
    "INSERT INTO users " +
```

```
        "(name, age) VALUES ($1, $2)",

        ['Mike', 39]
    );
```

Чтобы получить значение первичного ключа, после вставки строки с помощью инструкции `RETURNING` укажите название строки, значение которой нужно вернуть. Затем добавьте обратный вызов в качестве последнего аргумента в вызове метода `query` (листинг 8.8).

Листинг 8.8 – Получение первичного ключа

```
    client.query(
        "INSERT INTO users " +
        "(name, age) VALUES ($1, $2) " +
        "RETURNING id",
        ['Mike', 39],
        function (err, result) {
            if (err) throw err;
            console.log('Insert ID is ' + re-
sult.rows[0].id);
        }
    );
```

При создании запроса, возвращающего результаты, нужно сохранить в переменной возвращаемое значение клиентского метода `query`. Метод `query` возвращает объект, который наследует поведение класса `EventEmitter`, что позволяет воспользоваться преимуществами встроенной в *Node.js* функциональности. Этот объект генерирует событие `row` для каждой выбираемой в базе данных строки.

Далее показан код, который выводит данные из каждой строки, возвращаемой запросом. Обратите внимание на использование слушателей класса `EventEmitter`, определяющих операции, выполняемые со строками таблицы, а также операцию, которая выполняется после завершения выборки данных (листинг 8.9).

Листинг 8.9 – Вывод данных из каждой строки, возвращаемой запросом

```
    var query = client.query(
        "SELECT * FROM users WHERE age > $1",
        [40]
    );
    // обработка возврата строки
    query.on('row', function (row) {
        console.log(row.name)
```

```
});

// обработка завершения запроса
query.on('end', function () {
    client.end();
});
```

После выборки последней строки генерируется событие `end`, которое может являться сигналом для закрытия базы данных или выполнения другой прикладной логики.

Рассмотрим нереляционные БД. Нереляционные базы данных, сокращенно называемые «*NoSQL*», либо вообще не используют *SQL*, либо наравне с *SQL* обращаются к другим технологиям.

В то время как в реляционных СУБД производительность приносится в жертву надежности, во многих нереляционных базах данных во главу угла ставится производительность. Поэтому нереляционные базы данных лучше всего выбирать для разработки систем реального времени, выполняющих аналитические операции и обмен сообщениями. В нереляционных базах данных также обычно не требуется предварительное определение схем данных, что может быть полезно для приложений, в которых данные хранятся в виде иерархии, но эта иерархия меняется.

Хранилище данных `Redis` применяется для обработки простых данных, которые не требуют долговременного хранения, например, мгновенных сообщений, сессий или игровых данных. Данные `Redis` хранятся в оперативной памяти, а изменения, связанные со входом в систему, на диске. Хотя объем хранилища ограничен, операции с данными в `Redis` могут выполняться очень быстро. Данные могут быть восстановлены с помощью журнала, находящегося на диске.

Для работы с `Redis` необходимо установить модуль `redis` командой, представленной в листинге 8.10:

Листинг 8.10 – Установка модуля `redis`

```
npm i redis
```

Рассмотрим код, устанавливающий соединение с `Redis`-сервером. При этом используется заданный по умолчанию *TCP/IP*-порт на том же хосте. Созданный `Redis`-клиент включает унаследованное поведение класса `EventEmitter`, который генерирует событие `error` в случае, если возникают проблемы при попытке установки соединения с `Redis`-сервером. В листинге 8.11 показано, как можно определить собственный код обработки ошибок, добавив слушателя для типа событий `error`:

Листинг 8.11 – Код, устанавливающий соединение с Redis-сервером

```
var redis = require('redis');
var client = redis.createClient(6379, '127.0.0.1');

client.on('error', function (err) {
    console.log('Error ' + err);
})
```

После подключения к Redis-серверу приложение может начать обработку данных с помощью объекта client. Следующий пример кода демонстрирует хранение и выборку пар ключ/значение:

```
// функция print выводит на печать результаты выполнения операции либо ошибку, если таковая имела место
client.set('color', 'red', redis.print);
client.get('color', function (err, value) {
    if (err) throw err;
    console.log('Got: ' + value);
});
```

Рассмотрим код, выполняющий хранение и выборку значений в несколько более сложной структуре данных: в хеш-таблице (hash table), которая также известна как хеш-карта (hash map) (листинг 8.12).

Хеш-таблица фактически является таблицей идентификаторов, называемых ключами (keys), которые связаны с соответствующими значениями (values).

С помощью команды hmset в Redis элементам хеш-таблицы, которые идентифицируются по ключу, присваивается некое значение.

С помощью команды hkeys в Redis выводятся ключи для каждого элемента хеш-таблицы.

Листинг 8.12 – Код, выполняющий хранение и выборку значений

```
// установка элементов хеш-таблицы
client.hmset('camping', {
    'shelter': '2-person tent',
    'cooking': 'campstove'
}, redis.print);
// получение значения элемента "cooking"
client.hget('camping', 'cooking', function (err, value) {
    if (err) throw err;
    console.log('Will be cooking with: ' + value);
});
//  получение ключей хеш-таблицы
```

```
client.hkeys('camping', function (err, keys) {
    if (err) throw err;
    keys.forEach(function (key, i) {

        console.log(' ' + key);
    });
});
```

БД *MongoDB* является универсальной нереляционной базой данных. Она может применяться для разработки некоторых типов приложений, обычно создаваемых с помощью СУБД. В базе данных *MongoDB* документы хранятся в виде коллекций (`collections`).

MongoDB – документоориентированная система управления базами данных (СУБД) с открытым исходным кодом, не требующая описания схемы таблиц. Классифицирована как *NoSQL*, использует `JSON`-подобные документы и схему базы данных. Написана на языке C++.

Для работы с *MongoDB* необходимо установить модуль `Node-mongodb-native` командой (листинг 8.13):

Листинг 8.13 – Установка модуля `Node-mongodb-native`

```
npm i mongodb
```

После установки модуля `Node-mongodb-native` и запуска `MongoDB`-сервера для соединения с сервером воспользуйтесь кодом, представленным на листинге 8.14.

Листинг 8.14 – Соединение с сервером

```
var mongodb = require('mongodb');
var server = new mongodb.Server('127.0.0.1', 27017,
{});
var client = new mongodb.Db('mydatabase', server, {
    w: 1
});
```

Фрагмент кода для получения доступа к коллекции при открытом соединении с базой данных представлен на листинге 8.15.

Листинг 8.15 – Фрагмент кода

```
client.open(function (err) {
    if (err) throw err;
    client.collection('test_insert', function (err,
collection) {
        if (err) throw err;
        // сюда нужно вставить код запроса
```

```
            console.log('We are now able to perform que-
ries.');
        });
    });
```

Чтобы после завершения выполняемых с базой данных операций закрыть соединение с базой данных, необходимо выполнить в *MongoDB* команду `client.close()`.

В результате выполнения кода листинга 8.16 в коллекцию включается документ, а также выводится на печать уникальный идентификатор документа:

Листинг 8.16 – Фрагмент кода включения документа

```
    collection.insert({
            "title": "I like cake",
            "body": "It is quite good."
        },
        // безопасный режим указывает на то, что операция
с базой данных должна быть завершена перед выполнением
обратного вызова
        {safe: true},
        function (err, documents) {
            if (err) throw err;
            console.log('Document ID is: ' + docu-
ments[0]._id);
        }
    );
```

С помощью функции `console.log` идентификатор `documents[0]_id` можно вывести в виде строки, хотя фактически это не строка. Идентификаторы документов в базе данных *MongoDB* закодированы в формате `BSON` (*Binary JSON* – двоичный формат `JSON`). Этот формат обмена данными преимущественно используется в *MongoDB* вместо `JSON` для перемещения данных между `MongoDB`-клиентом и `MongoDB`-сервером. В большинстве случаев `BSON` обеспечивает более эффективное расходование пространства, чем `JSON`, а также позволяет выполнять более быстрый синтаксический разбор. В результате достигается более быстрое взаимодействие с базой данных.

С помощью идентификаторов `BSON`-документов можно обновлять данные. Код листинга 8.17 выполняет обновление документа на основании его идентификатора.

Листинг 8.17 – Выполнение обновления документа на основании его идентификатора

```
    var _id = new client.bson_serializer.Ob-
jectID('4e650d344ac74b5a01000001');
```

```
collection.update(Put MongoDB query code here {
    _id: _id
}, {
    $set: {
        "title": "I ate too much cake"
    }
}, {
    safe: true
}, function (err) {
    if (err) throw err;
});
```

Для поиска документов в базе данных *MongoDB* используется метод `find`. В примере кода на листинге 8.18 выводятся все элементы коллекции с заголовком «`Пример кодов`»:

Листинг 8.18 – Вывод элементов коллекции с заголовком

```
collection.find({
    "title": "Пример кодов"
}).toArray(function (err, results) {
    if (err) throw err;
    console.log(results);
});
```

Чтобы удалить запись, укажите ее внутренний идентификатор (либо выберите любой другой критерий) с помощью кода, представленного в листинге 8.19:

Листинг 8.19 – Удаление записи

```
    var _id = new client.bson_serializer.Ob-
jectID('4e6513f0730d319501000001');
    collection.remove({
        _id: _id
    }, {
        safe: true
    }, function (err) {
        if (err) throw err;
    });
```

MongoDB – достаточно мощная база данных. Модуль `Nodemongodb-native` обеспечивает быстрый доступ к ней. Многие пользователи предпочитают задействовать особый *API*-интерфейс, который абстрагирует доступ к базе данных, обрабатывая все его детали в фоновом режиме. Это позволяет ускорить разработку приложений и сократить объем кода. Один из наиболее популярных *API*-интерфейсов такого рода называется *Mongoose*.

Порядок выполнения

1 Рассмотрите работу *Node.js* с базами данных *SQL*. Примените модуль `sequelize`.

```
npm install sequelize --save
```

В *Node.js* модуль `sequelize` поддерживает следующие СУБД: *MySQL*, *PostgreSQL*, *MSSQL* и *MariaDB* поскольку имеет единое *API* (все перечисленные СУБД используют единый язык описания запросов – *SQL*).

Для работы с сервером *MySQL* в *Node.js* можно использовать ряд драйверов, например, `mysql` и `mysql2` (по большей части они совместимы). В нашем случае будем использовать `mysql2`, так как он предоставляет большую производительность.

2 Установите пакет `mysql2`:

```
npm install --save mysql2
```

3 Создайте подключение с помощью метода `createConnection()`, который в качестве параметра принимает настройки подключения и возвращает объект, предоставляющий подключение.

```
const mysql = require("mysql2");
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  database: "usersdb",
  password: "пароль_от_сервера"
});
```

Передаваемые в метод настройки конфигурации могут содержать ряд параметров. Наиболее используемые из них:

– `host` – хост, на котором запущен сервер `mysql` (по умолчанию имеет значение "`localhost`");

– `port` – номер порта, на котором запущен сервер `mysql` (по умолчанию имеет значение "`3306`");

– `user` – пользователь `MySQL`, который используется для подключения;

– `password` – пароль для пользователя `MySQL`;

– `database` – имя базы данных, к которой идет подключение. Данный параметр является необязательным. Если он не указан, то подключение идет в целом к серверу;

– `charset` – кодировка для подключения, например, по умолчанию используется "`UTF8_GENERAL_CI`";

– timezone – часовой пояс сервера *MySQL*. По умолчанию имеет значение "local".

Установите подключение с помощью метода connect() объекта connection:

```
const mysql = require("mysql2");
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  database: "usersdb",
  password: "пароль_от_сервера"
});
 connection.connect(function(err){
    if (err) {
      return console.error("Ошибка: " + err.mes-
sage);
    }
    else{
      console.log("Подключение к серверу MySQL
успешно установлено");
    }
 });
```

Метод connect() принимает функцию, параметр которой содержит ошибку, которая возникла при подключении.

Если при подключении к серверу mysql генерируется ошибка, то она считается возможной ошибкой при подключении:

```
Client does not support authentication protocol re-
quested by server; consider upgrading MySQL client
```

В этом случае необходимо в *MySQL Workbench* выполнить следующую команду:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'password'
```

4 Выполните закрытие подключения с помощью метода end():

```
const mysql = require("mysql2");
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
```

```
  database: "usersdb",
  password: "пароль_от_сервера"
});
// тестирование подключения
 connection.connect(function(err){
    if (err) {
      return console.error("Ошибка: " + err.mes-
sage);
    }
    else{
      console.log("Подключение к серверу MySQL
успешно установлено");
    }
 });
 // закрытие подключения
 connection.end(function(err) {
  if (err) {
    return console.log("Ошибка: " + err.message);
  }
  console.log("Подключение закрыто");
});
```

Проверьте правильность при запуске приложения и удачном подключении и закрытии подключения на консоли. Должно быть выдано сообщение:

```
Подключение к серверу MySQL успешно установлено
Подключение закрыто
```

Метод `end()` гарантирует, что перед закрытием подключения к БД будут выполнены все оставшиеся запросы, которые не завершились к моменту вызова метода.

Если мы не вызовем этот метод, то подключение будет оставаться активным, и приложение *Node.js* продолжит свою работу, пока сервер *MySQL* не закроет подключение.

Если же нам надо немедленно закрыть подключение, не дожидаясь выполнения оставшихся запросов, то в этом случае можно применить метод `destroy()`:

```
connection.destroy()
```

5 Рассмотрите выполнение запросов к *MySQL*.

Для выполнения запросов у объекта подключения применяется метод `query()`. Наиболее простая его форма:

```
query(sqlString, callback)
```

sqlString – выполняемая SQL-команда, а callback – функция обратного вызова, через параметры которой можно получить результаты выполнения sql-команды или возникшую ошибку.

Например, получим все данные из таблицы:

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

const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  database: "usersdb",
  password: "123456"
});
connection.query("SELECT * FROM users",
  function(err, results, fields) {
    console.log(err);
    console.log(results); // собственно данные
    console.log(fields); // метаданные полей
});
```

В данном случае выполняется команда SELECT, извлекает все данные из таблицы "users".

Функция обратного вызова принимает три параметра. Первый параметр выдает ошибку, если она возникла при выполнении запроса. Второй параметр – results – собственно представляет в виде массива те данные, которые получила команда SELECT. И третий параметр fields хранит метаданные полей таблицы и дополнительную служебную информацию.

Необходимо отметить, что при выполнении запросов неявно устанавливается подключение, поэтому перед выполнением запроса необязательно у объекта подключения вызывать метод connect().

Также в mysql2 определен метод execute(), который работает аналогичным образом:

```
const mysql = require("mysql2");
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  database: "usersdb",
  password: "123456"
});
connection.execute("SELECT * FROM users",
  function(err, results, fields) {
```

```
    console.log(err);
    console.log(results); // собственно данные
    console.log(fields); // метаданные полей
});
connection.end();
```

6 Выполните параметризацию запросов.

Если в запрос надо вводить данные, которые приходят извне, то для избежания sql-инъекций (уязвимостей, которые позволяют атакующему использовать фрагмент вредоносного кода на языке структурированных запросов для манипулирования базой данных и получения доступа к потенциально ценной информации) рекомендуется использовать параметризацию.

При параметризации вместо конкретных данных в тексте запроса ставятся плейсхолдеры – знаки вопроса, вместо которых при выполнении запроса будут вставляться собственно данные. Например, добавление данных:

```
const mysql = require("mysql2");
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  database: "usersdb2",
  password: "123456"
});
const user = ["Tom", 29];
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");
});
connection.end();
```

В данном случае данные определены в виде массива user, который в качестве параметра передается в метод connection.query().

7 Подключитесь к базе данных *MongoDB* с помощью *Node.js*.

MongoDB взаимодействует *Node.js* с помощью так называемого драйвера. Данный модуль позволят соединиться с базой данных, передавать на нее команды и получать данные.

Установить модуль можно с помощью npm:

```
npm install mongodb --save
```

Рассмотрите пример, показывающий использование драйвера для создания и поиска документов:

```
const MongoClient = require('mongodb').MongoClient;

// connection URL
const url = 'mongodb://localhost:27017';
// database Name
const dbName = 'myproject';
const insertDocuments = function(db, callback) {
  // get the documents collection
  const collection = db.collection('documents');
  // insert some documents
  collection.insertMany([
    {a : 1}, {a : 2}, {a : 3}
  ], function(err, result) {
    console.log("Inserted 3 documents into the col-
lection");
    callback(result);
  });
}
const findDocuments = function(db, callback) {
  // get the documents collection
  const collection = db.collection('documents');
  // find some documents
  collection.find({}).toArray(function(err, docs) {
    console.log("Found the following records");
    console.log(docs)
    callback(docs);
  });
}
// use connect method to connect to the server
MongoClient.connect(url, function(err, client) {
  console.log("Connected successfully to server");
  const db = client.db(dbName);
  insertDocuments(db, function() {
    findDocuments(db, function() {
      client.close();
    });
  });
});
```

8 Рассмотрите основы работы с `Mongoose`.

Прямая работа с данными *MongoDB* через драйвер представляется сложной для описания реальной бизнес-логики, поэтому существуют различные приемы, позволяющие приложению взаимодействовать с базой данных. Один из таких приемов – модуль `Mongoose`.

`Mongoose` – библиотека для работы `node.js` приложения с базой данных *MongoDB*. Он обеспечивает краткое, основанное на схемах решение, для моделирования данных приложений. `Mongoose` включает встроенное отражение типов данных, валидацию, построение запросов, привязку бизнес-логики и многое другое.

Выполните установку `Mongoose`

```
$ npm install mongoose
```

Включение в проект и соединение с базой данных с `test` в локальном экземпляре *MongoDB*.

```
// getting-started.js
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

Установка обработчиков успешного соединения с базой данных или ошибки может быть выполнена следующим образом:

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connec-
tion error:'));
db.once('open', function() {
  // we're connected!
});
```

Колбэк будет вызван, как только соединение будет установлено. В `Mongoose` все происходит от `Schema`. Можно взять ссылку на схему и определить свою схему.

```
var kittySchema = mongoose.Schema({
  name: String
});
```

Таким образом, была получена схема с одним свойством `name` типа `String`.

Далее скомпилируем схему в модель (`Model`).

```
var Kitten = mongoose.model('Kitten', kittySchema);
```

Модель – это класс, с помощью которого мы конструируем документы. В нашем случае каждый документ будет `Kitten` со свойствами и поведением, задекларированными в схеме. Создадим документ:

```
var silence = new Kitten({ name: 'Silence' });
console.log(silence.name); // 'Silence'
```

Добавим функциональность `speak` в наш документ:

```
// метод должен быть добавлен в схему перед ее ком-
пиляцией в mongoose.model()
kittySchema.methods.speak = function () {
  var greeting = this.name
    ? "Meow name is " + this.name
    : "I don't have a name";
  console.log(greeting);
}
var Kitten = mongoose.model('Kitten', kittySchema);
kittySchema);
```

Функции, добавленные в свойство `methods property`, компилируются в прототип модели и становятся доступны в каждом экземпляре – документе.

```
var fluffy = new Kitten({ name: 'fluffy' });
fluffy.speak(); // "Meow name is fluffy"
```

Сохраним созданные документы в *MongoDB*. Каждый документ может быть добавлен в базу данных с помощью его метода `save`. Первый аргумент колбэка при этом – ошибка – в случае ее возникновения.

```
 fluffy.save(function (err, fluffy) {
    if (err) return console.error(err);
    fluffy.speak();
  });
```

Можно получить доступ к сохраненным документам через модель:

```
Kitten.find(function (err, kittens) {
  if (err) return console.error(err);
  console.log(kittens);
})
```

Данный прием позволяет вывести все модели в консоль. Если мы хотим отфильтровать документы, Mongoose поддерживает синтаксис запросов *MongoDB* (querying).

```
Kitten.find({ name: /^fluff/ }, callback);
```

9 Установите локальный экземпляр *MongoDB*.

10 Создайте коллекцию для хранения информации о постах и комментариях блога. Документы комментариев должны быть в свойстве поста – массиве документов.

11 Напишите скрипт для создания, чтения, обновления и удаления постов и комментариев.

12 Сделайте рефакторинг скрипта, используя Mongoose.

Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Реализация решения задач с подробным рассмотрением (описанием) решений.
5 Выводы.
6 Список использованных источников.

Контрольные вопросы

1 Что такое функции обратного вызова?
2 Что такое асинхронный метод?
3 Что такое параметр объекта ошибки?
4 Что такое данные, возвращенные при успешном выполнении операции?
5 Для чего используется метод fs.open()?
6 Для чего используется флаг при работе с файлом?
7 Для чего используется метод .appendFile()?

Список использованных источников

1 Buckler, C. Node.js: Novice to Ninja / C. Buckler. – SitePoint, 2022. – 388 p. Erasmus, M. CoffeeScript Programming with jQuery, Rails, and Node.js / M. Erasmus. – Packt Publ., 2012. – 140 p.

2 Node.js. Notes for Professionals. – GoalKicker.com, 2018. – 333 p.

3 Wilson, J. Node.js 8 the Right Way / J. Wilson. – Pragmatic Bookshelf, 2018. – 336 p.

4 Алексеев, В. Ф. Структуры и базы данных. Пособие для курсового проектирования : пособие / В. Ф. Алексеев, И. Н. Богатко, Г. А. Пискун. – Минск : БГУИР, 2017. – 84 с.

5 Пауэрс, Ш. Изучаем Node. Переходим на сторону сервера / Ш. Пауэрс. – 2-е изд., доп. и перераб. – СПб. : Питер, 2017. – 304 с.

6 Хендерсон, К. Профессиональное руководство по SQL Server: структура и реализация / К. Хендерсон ; пер. с англ. – М. : Вильямс, 2006. – 184 с.

7 Хендерсон, К. Профессиональное руководство по SQL Server: хранимые процедуры / К. Хендерсон ; пер. с англ. – СПб. : Питер, 2005. – 146 с.

8 Янг, А. Node.js в действии / А. Янг, Б. Мек, М. Кантелон. – 2-е изд. – СПб. : Питер, 2018. – 432 с.

ЛАБОРАТОРНАЯ РАБОТА № 9

СЕТЬ И ШИФРОВАНИЕ

Цель работы

Ознакомиться с сетевым обменом данными и работой шифрования в базе данных.

Теоретические сведения

Web Socket – это протокол, который обеспечивает полнодуплексную (многостороннюю) связь, то есть позволяет осуществлять связь в обоих направлениях одновременно. Это современная веб-технология, в которой между браузером пользователя (клиентом) и сервером существует постоянное соединение. В этом типе связи между веб-сервером и веб-браузером они оба могут отправлять сообщения друг другу в любой момент времени. Традиционно в сети у нас был формат запроса/ответа, в котором пользователь отправляет HTTP-запрос, а сервер на него отвечал. Это все еще применимо в большинстве случаев, особенно при использовании RESTful API. Но ощущалась потребность в том, чтобы сервер также общался с клиентом, не получая ответа (или запроса) от клиента. Сервер сам по себе должен иметь возможность отправлять информацию клиенту или браузеру. Именно здесь необходим Web Socket.

Сокет (socket) представляет собой конечную точку обмена данными, а сетевой сокет – конечную точку обмена данными между приложениями, работающими на двух разных компьютерах в сети. Данные, передаваемые между сокетами, образуют поток (stream). Данные в потоке могут передаваться либо в виде двоичных данных в буфере, либо в виде строки в кодировке Юникод. Оба типа данных передаются в форме пакетов: частей данных, разделенных на блоки сходного размера. Также существует специальная разновидность пакетов – завершающий пакет (FIN), отправляемый сокетом как сигнал о завершении передачи.

Протокол TCP предоставляет коммуникационную платформу для большинства интернет-приложений, включая веб-службы и электронную почту. Он предоставляет механизм надежной передачи данных между клиентскими и серверными сокетами. TCP обеспечивает инфраструктуру, на которой строится прикладной уровень (например, протокол HTTP).

Клиент и сервер TCP создаются точно так же (с небольшими различиями), как это выполнялось с HTTP. При создании сервера TCP вместо передачи функции создания сервера requestListener с отдельными объектами ответа и запроса в единственном аргументе функции обратного вызова TCP передается экземпляр сокета, способного к отправке и получению данных.

В листинге 9.1 представлен TCP-сервер с сокетом, прослушивающим клиентскую передачу данных.

Листинг 9.1 – TCP-сервер с сокетом, прослушивающим клиентскую передачу данных

```
var net = require('net');
const PORT = 8124;
var server = net.createServer(function (conn) {
    console.log('connected');
    conn.on('data', function (data) {
        console.log(data + ' from ' + conn.remoteAd-
dress + ' ' + conn.remotePort);
        conn.write('Repeating: ' + data);
    });
    conn.on('close', function () {
        console.log('client closed connection');
    });
}).listen(PORT);
server.on('listening', function () {
    console.log('listening on ' + PORT);
});
server.on('error', function (err) {
    if (err.code == 'EADDRINUSE') {
        console.warn('Address in use, retrying...');
        setTimeout(() => {
            server.close();
            server.listen(PORT);
        }, 1000);
    } else {
        console.error(err);
    }
});
```

В данном коде после создания сервера серверный сокет прослушивает два события: получение данных и закрытие соединения клиентом. Затем полученные данные выводятся на консоль и отправляются обратно клиенту.

Сервер `TCP` также присоединяет обработчик для событий `listening` и `error`.

Приложение обрабатывает событие `error`. Если ошибка произошла из-за того, что порт в настоящее время используется, то приложение некоторое время ожидает, а потом пробует снова. Для других ошибок, например, обращений к порту `80`, требующему специальных привилегий, на консоль выводится полное сообщение об ошибке.

При создании сокета `TCP` можно передать дополнительный объект с параметрами, состоящий из двух значений: `pauseOnConnect` и `allowHalfOpen`. По умолчанию оба свойства имеют значение `false`.

Присваивание allowHalfOpen значения true запрещает сокету отправлять пакет FIN при получении пакета FIN от клиента. В этом случае сокет остается открытым для записи (не для чтения), и для закрытия сокета необходимо использовать функцию end(). Присваивание pauseOnConnect значения true позволяет принять соединение без чтения данных. Чтобы приступить к чтению данных, следует вызвать метод resume() для сокета.

Вместо того чтобы использовать программу для тестирования сервера, можно создать собственное клиентское приложение TCP. Как видно из листинга 9.2, клиент TCP создается так же просто, как и сервер. Данные передаются в буфере, но можно использовать setEncoding() для чтения данных в виде строки utf8. Метод write() сокета используется для передачи данных. Клиентское приложение также назначает функции прослушивания для двух событий: data для получения данных и close на случай, если сервер закроет соединение.

Листинг 9.2 – Создание клиента TCP

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');
// подключение к серверу
client.connect('8124', 'localhost', function () {
    console.log('connected to server');
    client.write('Who needs a browser to communi-
cate?');
});
// полученные данные отправляются серверу
process.stdin.on('data', function (data) {
    client.write(data);
});
// возвращаемые данные выводятся на консоль
client.on('data', function (data) {
    console.log(data);
});
// при закрытии сервера
client.on('close', function () {
    console.log('connection is closed');
});
```

Данные, передаваемые между двумя сокетами, вводятся в терминале и передаются при нажатии клавиши *ENTER*. Клиентское приложение отправляет введенный текст, а сервер TCP выводит его на консоль при получении. Сервер воз-

вращает полученное сообщение клиенту, который в свою очередь выводит сообщение на свою консоль. Сервер также выводит *IP*-адрес и порт клиента, используя свойства `remoteAddress` и `remotePort` сокета.

Соединение между клиентом и сервером поддерживается до того момента, пока не будет уничтожен один из участников клавишами *Ctrl + C*. Сокет, оставшийся открытым, получает событие `close`, сообщение о котором выводится на консоль. Сервер может обслуживать несколько соединений от нескольких клиентов, поскольку все соответствующие функции асинхронны.

Вместо того чтобы связывать порт с сервером `TCP`, можно связать его напрямую с сокетом. Для демонстрации этой возможности был изменен сервер `TCP` из предыдущих примеров, но новый сервер связывается с сокетом `Unix` (листинг 9.3). Сокет `Unix` соответствует некоторому пути на вашем сервере. Для более точного управления доступом к сокету могут использоваться разрешения на чтение и запись, вследствие чего этот метод является более предпочтительным, чем интернет-сокеты.

Листинг 9.3 – Связывание нового сервера с сокетом `Unix`

```
var net = require('net');
var fs = require('fs');
const unixsocket = '/somepath/Nodesocket';
var server = net.createServer(function (conn) {
    console.log('connected');
    conn.on('data', function (data) {
        conn.write('Repeating: ' + data);
    });
    conn.on('close', function () {
        console.log('client closed connection');
    });
}).listen(unixsocket);
server.on('listening', function () {
    console.log('listening on ' + unixsocket);
});
// при выходе с перезапуском сервера следует удалить
сокет
server.on('error', function (err) {
    if (err.code == 'EADDRINUSE') {
        fs.unlink(unixsocket, function () {
            server.listen(unixsocket);
        });
    } else {
        console.log(err);
    }
});
```

```
process.on('uncaughtException', function (err) {
    console.log(err);
});
```

Также пришлось внести изменения в обработку ошибок, чтобы сокет Unix удалялся при перезапуске приложения, если сокет уже используется. В реальном приложении перед выполнением столь радикальных действий следует убедиться в том, что сокет не используется другими клиентами.

TCP требует выделенного соединения между двумя конечными точками. UDP – протокол, не требующий соединения. Это означает, что соединение между двумя конечными точками не гарантировано. По этой причине протокол UDP по надежности и степени защиты от ошибок уступает TCP. С другой стороны, UDP обычно работает быстрее TCP, что делает его более популярным для задач реального времени и таких технологий, как VoIP (*Voice over Internet Protocol*), в которых требования к соединению TCP могут отрицательно повлиять на качество сигнала.

Базовая функциональность *Node.js* поддерживает оба типа сокетов. Функциональность TCP уже была продемонстрирована ранее, теперь очередь UDP.

Для модуля UDP используется идентификатор dgram (листинг 9.4).

Листинг 9.4 – Использование идентификатора dgram

```
require ('dgram');
```

Чтобы создать сокет UDP, вызовите метод createSocket и передайте ему тип сокета – udp4 или udp6. Также можно передать методу функцию обратного вызова для прослушивания событий. В отличие от сообщений, отправляемых по протоколу TCP, сообщения UDP должны отправляться в виде буферов, но не в виде строк.

Далее приведен демонстрационный код клиента UDP. В нем клиент обращается к данным через process.stdin и отправляет их через сокет UDP. Указывать кодировку строки при этом необязательно, потому что сокет UDP принимает только буфер, а данные process.stdin представляют собой буфер. Однако нам приходится преобразовывать буфер в строку методом toString() буфера, чтобы получить осмысленную строку для эхо-вывода данных вызовом console.log() (листинг 9.5).

Листинг 9.5 – Вызов console.log()

```
var dgram = require('dgram');
var client = dgram.createSocket("udp4");
process.stdin.on('data', function (data) {
    console.log(data.toString('utf8'));
    client.send(data, 0, data.length, 8124, "exam-
ples.burningbird.net", function (err, bytes) {
```

```
            if (err) console.error('error: ' + err);
            else console.log('successful');
        });
    });
```

Безопасность в веб-приложениях не ограничивается простым предотвращением несанкционированного доступа к серверу приложения. Безопасность – одна из сложнейших областей. При разработке приложений *Node.js* некоторые компоненты, необходимые для безопасности, уже созданы. Остается лишь подключить их в нужный момент.

Безопасный, защищенный от несанкционированного вмешательства обмен данными между клиентом и сервером осуществляется через протокол `SSL` (*Secure Sockets Layer*) и его обновленный вариант `TLS` (*Transport Layer Security*). Уровень `TLS/SSL` реализует шифрование данных для протокола `HTTPS`.

Соединение `TLS/SSL` требует согласования (`handshake`) между клиентом и сервером. В процессе согласования клиент (обычно браузер) сообщает серверу, какие виды безопасности он поддерживает. Сервер выбирает функцию и отправляет сертификат `SSL`, включающий открытый ключ. Клиент подтверждает сертификат и генерирует случайное число с использованием ключа сервера, отправляя его обратно серверу. Сервер использует свой закрытый ключ для расшифровки числа, которое в свою очередь используется для активизации безопасной передачи данных.

Чтобы эта схема работала, необходимо сгенерировать открытый и закрытый ключи, а также иметь сертификат. В реальных условиях сертификат будет подписываться доверенным центром сертификации (например, регистратором доменных имен), но для разработки можно воспользоваться самозаверяющим сертификатом. При этом в браузере выводится предупреждение для всех пользователей приложения, но поскольку сайт в процессе разработки еще недоступен для пользователей, проблем с этим быть не должно.

Веб-страницы, на которых пользователь вводит учетные данные, должны предоставляться по протоколу `HTTPS`. В противном случае ваши данные будут передаваться в открытом виде и могут быть легко перехвачены. `HTTPS` – модификация протокола `HTTP`, объединенная с `SSL`. `HTTPS` обеспечивает проверку подлинности веб-сайта, шифрует данные в ходе передачи и проверяет, что отправленные данные были получены без промежуточного несанкционированного вмешательства.

Добавление поддержки `HTTPS` напоминает добавление поддержки `HTTP` с включением объекта `options`, содержащего открытый ключ шифрования и подписанный сертификат. Отличается и порт по умолчанию для сервера `HTTPS`: протокол `HTTP` по умолчанию работает на порте `80`, а `HTTPS` – на порте `443`.

Node.js предоставляет криптографический модуль `Crypto`, открывающий интерфейс к функциональности `OpenSSL`. В него включены обертки для функ-

ций хеширования OpenSSL, HMAC, шифрования, дешифрования, подписи и верификации. Этот компонент *Node.js* достаточно прост в использовании, но он основан на предположении, что разработчик *Node.js* хорошо знает OpenSSL и все используемые функции.

Метод createHash модуля Crypto может использоваться для создания хеша пароля, сохраняемого в базе данных. В следующем примере хеш создается с использованием алгоритма sha1 и используется для кодирования пароля, после чего извлекается дайджест (digest) данных для сохранения в базе (листинг 9.6).

Листинг 9.6 – Создание хеша с использованием алгоритма sha1

```
var hashpassword = crypto.createHash('sha1').up-
date(password).digest('hex');
```

Для дайджеста назначается шестнадцатеричная кодировка. По умолчанию используется двоичная кодировка; также поддерживается кодировка base64.

Многие приложения применяют для этой цели хеширование. Однако при хранении простых хешированных паролей в базе данных возникает проблема, известная под названием «радужная таблица» (rainbow table), которая представляет собой таблицу заранее вычисленных значений хеша для всех возможных комбинаций символов. Таким образом, даже если у вас имеется пароль, который, как вы уверены, взломать невозможно, может оказаться, что такая последовательность символов встречается где-то в «радужной таблице». Изложенное заметно упростит определение пароля.

Проблема «радужной таблицы» обходится с помощью затравки (salt) — уникального сгенерированного значения, которое присоединяется к паролю перед шифрованием. Это может быть одно значение, которое используется со всеми паролями и безопасно хранится на сервере. Лучше генерировать уникальную затравку для каждого пароля и сохранять ее с паролем. Затравка может быть похищена вместе с паролем, но тогда злоумышленнику, пытающемуся взломать пароль, все равно придется генерировать «радужную таблицу» всего для одного пароля, что невероятно повышает сложность взлома любого конкретного пароля.

Рассмотрим простое приложение, которое получает имя пользователя и пароль в аргументах командной строки, генерирует хеш пароля, после чего сохраняет оба значения в таблице базы данных *MySQL*.

Чтобы воспроизвести этот пример на своем компьютере, необходимо установить модуль Node-mysql представленной в листинге 9.7 командой:

Листинг 9.7 – Установка модуля Node-mysql

```
npm install Node-mysql
```

Таблица создается командой *SQL*, представленной в листинге 9.8:

Листинг 9.8 – Команда *SQL* для создания таблицы

```
CREATE TABLE user (userid INT NOT NULL AUTO_INCREMENT,
PRIMARY KEY(userid), username VARCHAR(400) NOT NULL, pass-
wordhash VARCHAR(400) NOT NULL, salt DOUBLE NOT NULL );
```

Затравка состоит из значения даты, умноженного на случайное число и затем округленного. Она присоединяется к паролю перед вычислением хеша пароля. Все данные пользователя вставляются в таблицу *MySQL* (листинг 9.9).

Листинг 9.9 – Пример вставки в таблицу *MySQL* данных пользователя

```
var mysql = require('mysql'),
    crypto = require('crypto');
    var connection = mysql.createConnection({
    host: 'localhost',
    user: 'username',
    password: 'userpass'
});
connection.connect();
connection.query('USE Nodedatabase');
var username = process.argv[2];
var password = process.argv[3];
var salt = Math.round((Date.now() * Math.ranDOM())) +
'';
var hashpassword = crypto.createHash('sha512').up-
date(salt + password, 'utf8').digest('hex');
// создание записи пользователя
connection.query('INSERT INTO user ' +
    'SET username = ?, passwordhash = ?, salt = ?
    [username, hashpassword, salt], function(err, re-
sult) {
        if (err) console.error(err);
        connection.end();
    }
);
```

Сначала создается соединение с БД с последующим выбором вновь созданной таблицы. Имя пользователя и пароль извлекаются из командной строки, после чего начинается криптографическое «волшебство».

Программа генерирует затравку и передает ее функции для создания хеша с применением алгоритма `sha512`. Функции обновления хеша затравкой и выбора кодировки хеша объединяются в цепочку с функцией создания хеша. Зашифрованный хеш-пароль вставляется в созданную таблицу вместе с именем пользователя.

Рассмотрим приложение для проверки имени пользователя и пароля, которое запрашивает из базы данных хеш и затравку для заданного имени пользователя. Затравка используется для повторного генерирования хеша. Заново сгенерированный хеш сравнивается с паролем, хранящимся в базе данных. Если значения не совпадают, пользователь не проходит проверку. В случае совпадения пользователь допускается к работе с системой (листинг 9.10).

Листинг 9.10 – Проверка имени пользователя и пароля

```
var mysql = require('mysql'),
    crypto = require('crypto');
    var connection = mysql.createConnection({
    user: 'username',
    password: 'userpass'
});
connection.query('USE Nodedatabase');
var username = process.argv[2];
var password = process.argv[3];
connection.query('SELECT password, salt FROM user
WHERE username = ?', [username], function (err, result,
fields) {
        if (err) return console.error(err);
        var newhash = crypto.createHash('sha512').up-
date(result[0].salt + password, 'utf8').digest('hex');
        if (result[0].password === newhash) {
            console.log("OK, you're cool");
        } else {
            console.log("Your password is wrong. Try
again.");
        }
        connection.end();
})
```

Криптографическое хеширование может применяться к потоку. Для примера возьмем контрольную сумму – алгоритмический способ проверки правильности передачи данных. Вы создаете хеш файла и передаете его вместе с файлом при отправке. Получатель файла использует хеш для проверки правильности передачи. Для создания такого хеша используется функция `pipe()` и дуплексная природа функций `Crypto`.

Можно выбрать алгоритм `md5` для генерирования контрольной суммы `MD5`. Этот алгоритм пользуется популярностью во многих средах и приложениях из-за своей быстроты (хотя при этом обеспечивает меньшую защищенность).

Порядок выполнения

1 Рассмотрим безопасность создаваемых приложений. Защите разрабатываемых приложений стоит уделить особое внимание. Обеспечение должной безопасности гарантирует непрерывную стабильную работу сервера и сохранность личных данных пользователей.

Для защиты самого соединения и передаваемых по сети данных рекомендуется использовать один из протоколов криптографии: `SSL` или `TLS`.

Одним из способов защиты *Node.js* приложений от самых популярных уязвимостей в интернете является использование модуля `helmet`, который можно установить из репозитория `npm`.

```
npm install helmet --save
```

Модуль `helmet` представляет собой набор функций промежуточной обработки (`middleware`), устанавливающих определенные `HTTP`-заголовки для обеспечения безопасности.

Для использования `helmet` с настройками по умолчанию просто добавьте `middleware`.

```
const helmet = require('helmet'),
  express = require('express'),
  app = express()
app.use(helmet())
```

2 Изучите список функций промежуточной обработки, входящих в модуль `helmet` (более подробная информация находится на *https://helmetjs.github.io/*).

2.1 `csp` – предотвращение межсайтовых вмешательств через задание заголовка `Content-Security-Policy`.

```
app.use(
  helmet.csp({
    directives: {
      objectSrc: ['none'],
      workerSrc: false, //значение не задается
    },
  })
)
```

2.2 `dnsPrefetchedControl` – управляет загрузкой `DNS` через задание заголовка `X-DNS-Prefetch-Control`.

```
app.use(helmet.dnsPrefetchControl({ allow: true }))
//задание заголовка
app.use(helmet.dnsPrefetchControl({ allow: false }))
//удаление заголовка
app.use(helmet.dnsPrefetchControl()) //удаление за-
головка
```

2.3 `expectCt` – устанавливает заголовок `Expect-CT`.

```
app.use(
  helmet.expectCt({
    enforce: true,
    maxAge: 90, //задается в днях
    reportUri: 'http://example.com/report',
  })
)
```

2.4 `featurePolicy` – позволяет ограничить использование некоторых функций браузера (например, использование камеры или геолокации) заданием заголовка `Feature-Policy`.

```
app.use(
  helmet.featurePolicy({
    features: {
      camera: ['none'], //запрет использования web-
камеры
      fullscreen: ['none'], //запрет использования
Fullscreen API
      geolocation: ['none'], //запрет использования
геолокации
    },
  })
)
```

2.5 `hidePoweredBy` – удаляет или заменяет значение следующего заголовка:
`X-Powered-By`; устанавливается отдельно (в случае использования `express` не нужен).

```
app.use(helmet.hidePoweredBy()) //отключение заго-
ловка
app.use(helmet.hidePoweredBy({ setTo: 'PHP X.X.X'
})) //установка для заголовка определенного значения
```

```
app.disable('x-powered-by') //отключение через ex-
press
```

2.6 hsts – управляет заголовком Strict-Transport-Security.

```
app.use(
  helmet.hsts({
    maxAge: 31536000, //задается в секундах
    includeSubdomains: true,
    preload: true,
  })
)
```

2.7 ieNoOpen – устанавливает заголовок X-Download-Options для предотвращения открытия скачанных файлов в контексте вашего сайта.

```
app.use(helmet.ieNoOpen())
```

2.8 noCache – отключает кэширование на стороне клиента через задание заголовков Cache-Control, Pragma, Surrogate-Control и Expires.

```
app.use(helmet.noCache())
```

2.9 noSniff – задает заголовок X-Content-Type-Options и тем самым предотвращает прослушивание MIME ответов с не соответствующим заявленному в Content-Type типу данных.

```
app.use(helmet.noSniff())
```

2.10 frameguard – предотвращает атаку clickjacking заданием заголовка X-Frame-Options.

```
app.use(
  helmet.frameguard({
    action: 'deny',
    domain: 'http://example.com',
  })
)
```

2.11 permittedCrossDomainPolicies – задает следующий заголовок:
X-Permitted-Cross-Domain-Policies; необходим для предотвращения загрузки на сайте контента Adobe Flash и Adobe Acrobat.

```
app.use(helmet.permittedCrossDomainPolicies()) //по
умолчанию none
app.use(
  helmet.permittedCrossDomainPolicies({
    permittedPolicies: 'all',
  })
)
```

2.12 `referrerPolicy` – управляет заголовком `Referrer`, задавая заголовок `Referrer-Policy`.

```
app.use(helmet.referrerPolicy({ policy: 'no-refer-
rer' }))
app.use(
  helmet.referrerPolicy({
    policy: ['no-referrer', 'unsafe-url'],
  })
)
```

2.13 `xssFilter` – устанавливает заголовок `X-XSS-Protection`, активируя тем самым фильтр межсайтового скриптинга.

```
app.use(
  helmet.xssFilter({
    setOnOldIE: true,
    reportUri: 'http://example.com/report',
  })
)
```

2.14 Для отключения конкретных функций промежуточной обработки, входящих в `helmet` и активных по умолчанию, необходимо передать экземпляру модуля соответствующий конфигурационный объект.

```
app.use(
  helmet({
    ieNoOpen: false,
    hidePoweredBy: false,
  })
)
```

2.15 Подобным образом также можно сразу задать настройки для всех функций промежуточной обработки.

```
app.use(
  helmet({
    xssFilter: {
      setOnOldIE: true,
    },
    hidePoweredBy: {
      setTo: 'PHP X.X.X',
    },
  })
)
```

3 Рассмотрим выполнение функций криптографии в *Node.js*, которая обеспечивается встроенным в платформу модулем `crypto`, поддерживающим алгоритмы для шифрования и расшифровки данных, генерацию сертификатов, хэш-функции и так далее.

3.1 Модуль `crypto` входит не во все сборки *Node.js*, поэтому его использование в приложении, запускаемом на разных серверах, может быть не всегда возможным. Чтобы проверить доступность модуля, добавьте следующий код.

```
let crypto;

try {
  crypto = require('crypto');
} catch (err) {
  console.log('Crypto module is unavailable');
}
```

3.2 Для получения списка поддерживаемых в *Node.js* алгоритмов шифрования выполните у экземпляра модуля `crypto` функцию `getCiphers()`.

```
const crypto = require('crypto');
console.log(crypto.getCiphers());
```

3.3 В лабораторной работе будут рассмотрены шифрование и расшифровка данных.

3.3.1 Шифрование данных по заданному алгоритму осуществляется в три этапа:

– создание объекта `Cipher`;

– добавление к созданному объекту данных, которые необходимо зашифровать;

– завершение процесса шифрования.

Объект `Cipher` создается вызовом метода `crypto.createCipheriv()`, который принимает три параметра:

– алгоритм;
– ключ;
– вектор инициализации (необязательный).

Обогащение созданного объекта `Cipher` данными осуществляется с помощью метода `[CipherInstance].update()`, которому можно передать следующие аргументы:

– данные для шифрования;
– кодировка исходных данных (необязательный);
– кодировка возвращаемого методом значения (необязательный).

Завершение процесса шифрования осуществляется вызовом метода `final()`, принимающего кодировку зашифрованных данных.

Рассмотрим пример шифрования данных.

```
const crypto = require('crypto');
const iv = crypto.randomBytes(16); //генерация век-
тора инициализации
const key = crypto.scryptSync('secret', 'salt', 32);
//генерация ключа
const encyptedData = crypto
  .createCipheriv('aes-256-cbc', key, iv)
  .update('Any data', 'utf8', 'hex')
  .final('hex');
console.log(encryptedData);
```

В зависимости от используемого алгоритма длина вектора инициализации может отличаться.

3.3.2 Теперь рассмотрим, как расшифровать зашифрованные данные. В качестве зашифрованных данных возьмем результат из предыдущего примера. Последовательность действий та же, только вместо метода `createCipher()` используется `createDecipher()`.

```
const crypto = require('crypto');
const iv = crypto.randomBytes(16); //генерация век-
тора инициализации
const key = crypto.scryptSync('secret', 'salt', 32);
//генерация ключа
const decyptedData = crypto
  .createDecipheriv('aes-256-cbc', key, iv)
  .update(encryptedData, 'hex', 'utf8')
  .final('utf8');
console.log(decryptedData); //Any data
```

Процесс шифрования представляет собой поток. Поэтому пример выше может быть переписан следующим образом.

```
const crypto = require('crypto');
const iv = crypto.randomBytes(16); //генерация век-
тора инициализации
const key = crypto.scryptSync('secret', 'salt', 32);
//генерация ключа
let cipherStream = crypto.createCipheriv(
  'aes-256-cbc',
  key,
  iv
);
let encryptedData = '';
cipherStream.on(
  'data',
  (data) => (encryptedData += data.toString('hex'))
);
cipherStream.write('Any data');
cipherStream.end();
let decipherStream = crypto.createDecipheriv(
  'aes-256-cbc',
  key,
  iv
);
let decryptedData = '';
decipherStream.on(
  'data',
  (data) => (decryptedData += data)
);
decipherStream.on('end', () => console.log(decrypt-
edData)); //'Any data'
decipherStream.write(encryptedData, 'hex');
decipherStream.end();
```

4 Рассмотрите *Socket* в *Node.js*. Чтобы использовать *Socket* в *Node.js*, сначала нужно установить зависимость, которая называется `socket.io`. Можно установить ее, выполнив команду ниже в `cmd`, а затем добавить эту зависимость в файл `javascript` на стороне сервера, а также установить экспресс-модуль, который в основном требуется для серверного приложения.

```
npm install socket.io --save
npm install express - сохранить
```

5 Создайте серверный код:

```
const express = require('express'); // using express
const socketIO = require('socket.io');
const http = require('http')
const port = process.env.PORT||3000 // setting the
port
let app = express();
let server = http.createServer(app)
let io = socketIO(server)
server.listen(port);
```

5.1 Установите соединение со стороны сервера на сторону клиента, через которое сервер сможет отправлять данные клиенту.

```
// устанавливаем соединение с пользователем со сто-
роны сервера
io.on('connection', (socket) => {
  console.log('New user connected');
});
```

5.2 Точно так же со стороны клиента нужно добавить файл сценария, а затем установить соединение с сервером, через который пользователь отправляет данные на сервер.

```
<script src="/socket.io/socket.io.js"></script>
<script>
var socket=io()
// устанавливаем соединение с сервером со стороны
пользователя
socket.on('connect', function() {
  console.log('Connected to Server')
});
</script>
```

5.3 Чтобы отправить сообщение или данные с сервера пользователю, необходимо создать сокет `socket.on()` внутри соединения, которое мы сделали со стороны сервера.

```
// слушатель, который срабатывает когда сервер от-
правляет какое-либо сообщение
socket.on('createMessage', (newMessage) => {
  console.log('newMessage', newMessage);
})
```

5.4 Теперь любые данные могут быть отправлены с любой стороны, так что между сервером и клиентом создается соединение. Затем, если сервер отправляет сообщение, клиент может прослушать это сообщение или, если клиент отправляет сообщение, сервер может прослушать это сообщение. Таким образом, необходимо сгенерировать сокет для отправки и прослушивания сообщений как на сервере, так и на стороне клиента.

Пример серверного кода:

```
const express=require('express');
const socketIO=require('socket.io');
const http=require('http')
const port=process.env.PORT||3000
var app=express();
var io=socketIO(server);
// make connection with user from server side
io.on('connection', (socket)=>{
  console.log('New user connected');
   //emit message from server to user
   socket.emit('newMessage', {
     from:'jen@mds',
     text:'hepppp',
     createdAt:123
   });
  // listen for message from user
  socket.on('createMessage', (newMessage)=>{
    console.log('newMessage', newMessage);
  });
  // when server disconnects from user
  socket.on('disconnect', ()=>{
    console.log('disconnected from user');
  });
});
server.listen(port);
```

5.5 После выполнения описанных действий будет подключен новый пользователь:

```
{
  кому: 'john @ ds',
  текст: 'what kjkljd'
}
```

6 Создайте клиентский код:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>ChatApp</title>
  </head>
  <body class="chat">
        <form id='message-form'>
          <input name='message' type="text"place-
holder="Message"
                 autofocus autocomplete="off"/>
          <button >Send</button>
        </form>
<script src="/socket.io/socket.io.js"></script>
<script>
var socket=io()
// connection with server
socket.on('connect', function(){
  console.log('Connected to Server')
});
// message listener from server
socket.on('newMessage', function(message){
 console.log(message);
});
// emits message from user side
socket.emit('createMessage', {
  to:'john@ds',
  text:'what kjkljd'
});
// when disconnected from server
socket.on('disconnect', function(){
  console.log('Disconnect from server')
});
</script>
  </body>
</html>
```

После выполнения описанных действий будет выполнено подключение к серверу:

```
Connected to Server
{
```

```
        from:'jen@mds',
        text:'hepppp',
        createdAt:123
}
```

Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Реализация решения задач с подробным рассмотрением (описанием) решений.
5 Выводы.
6 Список использованных источников.

Контрольные вопросы

1 Что такое функции `Web Socket`?
2 Что такое `socket.io`?
3 Зачем необходимо выполнять соединение со стороны сервера на сторону клиента?
4 Как установить соединение с сервером, через который пользователь отправляет данные на сервер?
5 Зачем необходимо создание `socket.on()` внутри соединения?
6 Для чего необходимо шифрование?
7 Почему *Node.js* стал настоящим успехом в веб-разработке?
8 Зачем необходима платформа `Express.js`?
9 Что такое серверная структура для создания веб-приложений?

Список использованных источников

1 Buckler, C. Node.js: Novice to Ninja / C. Buckler. – SitePoint, 2022. – 388 p. Erasmus, M. CoffeeScript Programming with jQuery, Rails, and Node.js / M. Erasmus. – Packt Publ., 2012. – 140 p.
2 Wilson, J. Node.js 8 the Right Way / J. Wilson. – Pragmatic Bookshelf, 2018. – 336 p.
3 Node.js. Notes for Professionals. – GoalKicker.com, 2018. – 333 p.
4 Учусь Node.js. Free unaffiliated eBook created from Stack Overflow contributors [Электронный ресурс]. – Режим доступа: https://riptutorial.com/Download/node-js-ru.pdf
5 Пауэрс, Ш. Изучаем Node. Переходим на сторону сервера / Ш. Пауэрс. – 2-е изд., доп. и перераб. – СПб. : Питер, 2017. – 304 с.
Янг, А. Node.js в действии / А. Янг, Б. Мек, М. Кантелон. – 2-е изд. – СПб. : Питер, 2018. – 432 с.

ЛАБОРАТОРНАЯ РАБОТА № 10

РАБОТА С API

Цель работы

Изучить возможность реализации полноценного API в стиле REST для взаимодействия с пользователем.

Теоретические сведения

API (*Application Programming Interface*) – это интерфейс программирования, интерфейс создания приложений. Если говорить более понятным языком, то это готовый код. Большинство крупных компаний на определенном этапе разрабатывают API для клиентов или для внутреннего пользования.

API создавался для того, чтобы программист мог облегчить задачу написания того или иного приложения благодаря использованию готового кода (например, функций). Всем известный jQuery, написанный на *JavaScript*, является тоже своего рода API. Если рассматривать конкретно данный пример, то jQuery позволяет сильно облегчить написание кода. То, что обычными средствами *JavaScript* можно было сделать за 30 строк, через jQuery пишется за 5–6. Если рассматривать API в общем, то можно найти очень много сервисов, предоставляющих решения для разработки. Самый известный на сегодняшний день – это сервис code.google.com, предоставляющий около полусотни разнообразных API. Это и интерфейс для создания *Android*-приложений, и различные API для работы с AJAX, и различные API приложений, которые можно легко подстроить «под себя».

Успешные сайты все чаще не полностью автономны. Для того чтобы заинтересовать существующих и найти новых пользователей, интеграция с социальными сетями обязательна. Для указания местонахождения магазинов и мест оказания прочих услуг очень большое значение имеет использование сервисов геолокации и отображения на карте. И на этом не останавливаются: все больше организаций осознают, что предоставление API помогает расширить перечень услуг и сделать их более полезными.

Предположим, мы хотим указать десять последних твитов, которые содержат хештег #meadowlarktravel. Мы могли бы использовать для этого компонент клиентской части, но он будет включать дополнительные запросы HTTP. Более того, если мы делаем это на серверной стороне, у нас появляется возможность кэширования твитов для повышения производительности. В этом случае можно отправлять в черный список твиты недоброжелателей, тогда как на стороне клиента это сделать было бы существенно сложнее.

Twitter, как и *Facebook*, позволяет вам создать приложения. Это немного неподходящее название: приложение *Twitter* ничего не делает (в традиционном смысле). Это скорее набор учетных данных, которые вы можете использовать

для создания реального приложения на вашем сайте. Простейший и наиболее универсальный способ получения доступа на *Twitter* `API` – создать приложение и использовать его для получения токена доступа.

Рассмотрим создание приложения *Twitter*. Для этого следует войти на *http://dev.twitter.com*.

Щелкните на значке в левом верхнем углу, затем выберите «**Мои приложения**». Нажмите кнопку «**Создать новое приложение»** и следуйте инструкциям. Когда появится приложение, можно увидеть, что у вас теперь есть потребительский ключ и потребительский секрет. Потребительский секрет, на что указывает название, должен хранить секрет: никогда не включайте его в ответы, отправляемые на сторону клиента. Если бы кто-то извне получил доступ к этому секрету, он мог бы создавать запросы от имени вашего приложения, что могло бы иметь печальные последствия, если их использование вредоносное.

После создания потребительского ключа можно начать общаться с `Twitter REST API`. Чтобы сохранять код в аккуратном виде, поместим код `Twitter` в модуль `lib/twitter.js` (листинг 10.1).

Листинг 10.1 – Применение `module.exports`

```
var https = require('https');
module.exports = function (twitterOptions) {
    return {
        search: function (query, count, cb) { // то,
что нужно сделать
        }
    };
};
```

Следующий модуль экспортирует функцию, в которую вызывающая сторона передает объект конфигурации. Возвращается объект, содержащий методы. Таким образом можно добавить функционал в свой модуль. Рассмотрим метод `search` (листинг 10.2). Вот как будем использовать библиотеку:

Листинг 10.2 – Использование библиотеки

```
var twitter = require('./lib/twitter')({
    consumerKey: credentials.twitter.consumerKey,
    consumerSecret:   credentials.twitter.consumerSe-
cret,
});


twitter.search('#meadowlarktravel', 10, function (re-
sult) {
    // твиты будут в result.statuses
});
```

Прежде чем реализовать метод `search`, необходимо предоставить определенную функциональность для аутентификации самих себя в *Twitter*. Данный процесс не вызывает затруднений: используем `HTTPS` для запроса токена доступа, базирующийся на нашем потребительском ключе и потребительском секрете. Это необходимо сделать это только один раз, так как токены у *Twitter* даются бессрочно. Чтобы не запрашивать токен доступа каждый раз, его необходимо закэшировать для дальнейшего использования.

Единственная функция, которая доступна вызывающей стороне, – это `module.exports`. Поскольку мы возвращаем функцию вызывающей стороне, то доступной будет только она. Вызов функции дает в результате объект, и только свойства этого объекта доступны вызывающей стороне.

Создадим переменную `accessToken`, которую будем использовать для кэширования нашего токена доступа, и функцию `getAccessToken`, которая этот токен получит. При первом запуске она создаст запрос `Twitter API` для получения токена доступа. Последующие вызовы просто возвращают значение `accessToken` (листинг 10.3).

Листинг 10.3 – Применение функции `getAccessToken`

```
var https = require('https');
module.exports = function (twitterOptions) {
    // эта переменная будет невидимой вне этого мо-
дуля var accessToken;
    // эта функция будет невидимой за пределами этого
модуля
    function getAccessToken(cb) {
        if (accessToken) return cb(accessToken); //
то, что нужно сделать: получение токена доступа
    }
    return {
        search: function (query, count, cb) { // то,
что нужно сделать
        },
    };
};
```

Поскольку `getAccessToken` может требовать асинхронный вызов к `Twitter API`, необхоимо предоставить обратный вызов, который будет осуществляться, когда значение `accessToken` действительно. Теперь, когда мы установили базовую структуру, реализуем `getAccessToken` (листинг 10.4).

Листинг 10.4 – Практическая реализация функции `getAccessToken`

```
function getAccessToken(cb) {
    if (accessToken) return cb(accessToken);
```

```
        var bearerToken = Buffer(encodeURIComponent(twit-
terOptions.consumerKey) + ':' + encodeURIComponent(twit-
terOptions.consumerSecret)).toString('base64');

        var options = {
            hostname: 'api.twitter.com',
            port: 443,
            method: 'POST',
            path: '/oauth2/token?grant_type=client_cre-
dentials',
            headers: {
                'Authorization': 'Basic ' + bearerToken,
            },
        };
        https.request(options, function (res) {
            var data = '';
            res.on('data', function (chunk) {
                data += chunk;
            });
            res.on('end', function () {
                var auth = JSON.parse(data);
                if (auth.token_type !== 'bearer') {
                    console.log('Twitter auth failed.');
                    return;
                }
                accessToken = auth.access_token;
                cb(accessToken);
            });
        }).end();
    }
```

Подробности построения этого вызова доступны на странице документации для разработчиков *Twitter* для аутентификации «**только для приложений**». Необходимо сделать токен предъявителя (`bearer token`), что будет `base64`-кодированной комбинацией потребительского ключа и потребительского секрета. После того как будет сконструирован токен, можно вызвать `/oauth2/token API` с заголовком `Authorization`, содержащим токен предъявителя для запроса токена доступа. Обязательно необходимо использовать `HTTPS`. Если вы попробуете сделать этот вызов посредством `HTTP`, то передадите свой секрет в незашифрованном виде, а `API` попросту отключит вас. После того как получим полный ответ от `API` (мы ожидаем событие `end` потока ответа), следует разобрать `JSON` и убедиться, что тип токена `bearer`.

Мы кэшируем токен доступа, затем вызываем функцию обратного вызова. Теперь, раз у нас есть механизм получения токена доступа, мы можем делать вызовы API. Реализуем наш метод поиска согласно листингу 10.5.

Листинг 10.5 – Реализация метода поиска

```
search: function (query, count, cb) {
    getAccessToken(function (accessToken) {
        var options = {
            hostname: 'api.twitter.com',
            port: 443,
            method: 'GET',
            path: '/1.1/search/tweets.json?q=' + en-
codeURIComponent(query) + '&count=' + (count || 10),
            headers: {
                'Authorization': 'Bearer ' + ac-
cessToken,
            },
        };
        https.request(options, function (res) {
            var data = '';
            res.on('data', function (chunk) {
                data += chunk;
            });
            res.on('end', function () {
                cb(JSON.parse(data));
            });
        }).end();
    });
}
```

Создадим функцию для получения последних твитов. Если они уже закэшированы и время действия кэша еще не истекло, просто возвращаем topTweets.tweets. В противном случае выполняем поиск и затем делаем повторяющиеся вызовы embed для получения встраиваемого HTML.

Будем использовать библиотеку Q promises.

Запустите команду npm install --save q и пропишите var Q = require(q); в верхней части своего файла приложения. Данная функция представлена в листинге 10.6.

Листинг 10.6 – Функция topTweets.tweets

```
function getTopTweets(cb) {
    if (Date.now() < topTweets.lastRefreshed + topT-
weets.refreshInterval) return cb(topTweets.tweets);
```

```
        twitter.search('#meadowlarktravel', topTweets.count,
function (result) {
            var formattedTweets = [];
            var promises = [];
            var embedOpts = {
                omit_script: 1
            };
            result.statuses.forEach(function (status) {
                var deferred = Q.defer();
                twitter.embed(status.id_str,   embedOpts,
function (embed) {
                    formattedTweets.push(embed.html);
                    deferred.resolve();
                });
                promises.push(deferred.promise);
            });
            Q.all(promises).then(function () {
                topTweets.lastRefreshed = Date.now();
                cb(topTweets.tweets = formattedTweets);
            });
        });
    }
```

Это лишь небольшая часть того, что может быть связано с интеграцией API сторонних сервисов.

Порядок выполнения

1 Используя *Express* и *Node.js*, реализуйте полноценный API в стиле REST для взаимодействия с пользователем.

Архитектура REST предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером: GET, POST, PUT, DELETE.

Зачастую REST-стиль особенно удобен при создании всякого рода Single Page Application, которые нередко используют специальные javascript-фреймворки типа Angular, React или Knockout.

2 Рассмотрите, как создать свой API. Для нового проекта лабораторной работы создайте новую папку, которую назовите, например, webapp. Сразу определите в проекте файл package.json:

```
{
  "name": "webapp",
  "version": "1.0.0",
  "dependencies": {
```

```
    "express": "^4.17.0"
  }
}
```

3 В проекте понадобятся express для обработки запроса. Перейдите к этому каталогу в командной строке/терминале и для добавления пакета выполним команду:

```
npm install
```

4 В лабораторной работе создается экспериментальный проект, который будет хранить данные в файле json и который призван просто показать создание API в *Node.js* в стиле REST.

Добавьте в папку проекта новый файл users.json со следующим содержанием:

```
[{
    "id":1,
    "name":"Виктор",
    "age":24
},
{
    "id":2,
    "name":"Сергей",
    "age":27
},
{
    "id":3,
    "name":"Анна",
    "age":"23"
}]
```

5 Для чтения и записи в этот файл используйте встроенный модуль fs. Для обработки запросов определим в проекте следующий файл app.js:

```
const express = require("express");
const fs = require("fs");
const app = express();
const jsonParser = express.json();
app.use(express.static(__dirname + "/public"));
const filePath = "users.json";
app.get("/api/users", function(req, res){
```

```
    const content = fs.read-
FileSync(filePath,"utf8");
    const users = JSON.parse(content);
    res.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
    for(var i=0; i<users.length; i++){
        if(users[i].id==id){
            user = users[i];
            break;
        }
    }
    // отправляем пользователя
    if(user){
        res.send(user);
    }
    else{
        res.status(404).send();
    }
});
// получение отправленных 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(func-
tion(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.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(var i=0; i < users.length; i++){
        if(users[i].id==id){
            index=i;
            break;
        }
    }
    if(index > -1){
        // удаляем пользователя из массива по ин-
дексу
        const user = users.splice(index, 1)[0];
        data = JSON.stringify(users);
        fs.writeFileSync("users.json", data);
        // отправляем удаленного пользователя
        res.send(user);
    }
    else{
        res.status(404).send();
    }
});
// изменение 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(var 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.send(user);
    }
    else{
        res.status(404).send(user);
    }
});

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

6 Рассмотрите методы для обработки запросов. Определено пять методов для каждого типа запросов:

`app.get()/app.post()/app.delete()/app.put()`

Когда приложение получает запрос типа `GET` по адресу "`api/users`", то срабатывает следующий метод:

```
app.get("/api/users", function(req, res){
    const content = fs.read-
FileSync(filePath,"utf8");
    const users = JSON.parse(content);
    res.send(users);
});
```

В качестве результата обработки отправьте массив пользователей, который считывается из файла.

Для упрощения кода приложения в рамках данного экспериментального проекта для чтения/записи файла применяются синхронные методы `fs.readFileSync()/fs.writeFileSync()`. Но в реальности, как правило, работа с данными будет идти через базу данных, а далее мы все это рассмотрим на примере `MongoDB`.

7 Чтобы получить данные из файла с помощью метода `fs.readFileSync()`, считываем данные в строку, которую парсим (*парсить* – собирать и систематизировать информацию, размещенную на определенных сайтах, с помощью специальных программ, автоматизирующих процесс) в

массив объектов с помощью функции JSON.parse(). В конце полученные данные отправляем клиенту методом res.send().

8 Аналогично работает и метод app.get(), который срабатывает, когда в адресе указан id пользователя:

```
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
    for(var i=0; i<users.length; i++){
        if(users[i].id==id){
            user = users[i];
            break;
        }
    }
    // отправляем пользователя
    if(user){
        res.send(user);
    }
    else{
        res.status(404).send();
    }
});
```

Единственное, что в этом случае нам надо найти нужного пользователя по id в массиве, а если он не был найден, возвратить статусный код 404: res.status(404).send().

9 При получении запроса методом POST нам надо применить парсер jsonParser для извлечения данных из запроса:

```
// получение отправленных 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(func-
tion(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.send(user);
});
```

10 После получения данных создайте новый объект и добавьте его в массив объектов. Для этого выполните считывание данных из файла, добавьте в массив новый объект и перезапишите файл с обновленными данными.

11 При удалении выполните похожие действия, только теперь извлекайте из массива удаляемый объект и опять же перезаписывайте файл:

```
// 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(var i=0; i < users.length; i++){
        if(users[i].id==id){
            index=i;
            break;
        }
    }
    if(index > -1){
        // удаляем пользователя из массива по ин-
дексу
        const user = users.splice(index, 1)[0];
        data = JSON.stringify(users);
        fs.writeFileSync("users.json", data);
        // отправляем удаленного пользователя
        res.send(user);
    }
    else{
        res.status(404).send();
    }
});
```

Если объект не найден, возвращаем статусный код 404.

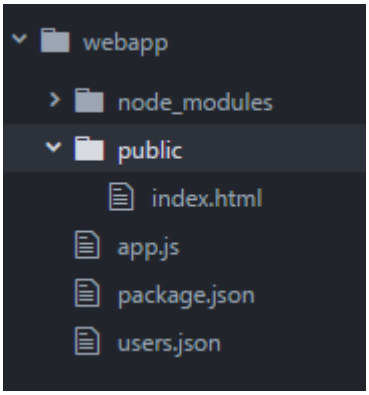
12 Предположим, что приложению приходит PUT-запрос. В этом случае он обрабатывается методом app.put(), в котором с помощью jsonParser получаем измененные данные:

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

13 Для поиска изменяемого объекта выполним считывание данных из файла. Найдите изменяемого пользователя по id, измените у него свойства и сохраните обновленные данные в файл.

14 При выполнении предыдущих действий был определен простейший API. Теперь добавьте код клиента.

В коде Express для хранения статических файлов использует папку public, поэтому создайте в проекте подобную папку. В этой папке определите новый файл index.html, который будет выполнять роль клиента. В итоге весь проект будет выглядеть следующим образом:



15 Определите в файле `index.html` следующий код:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-
width" />
    <title>Список пользователей</title>
    <link href="https://maxcdn.boot-
strapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
rel="stylesheet" />
</head>
<body>
    <h2>Список пользователей</h2>
    <form name="userForm">
        <input type="hidden" name="id" value="0" />
        <div class="form-group">
            <label for="name">Имя:</label>
            <input class="form-control"
name="name"/>
        </div>
        <div class="form-group">
            <label for="age">Возраст:</label>
            <input class="form-control" name="age"/>
        </div>
        <div class="panel-body">
            <button type="submit" class="btn btn-sm
btn-primary">Сохранить</button>
            <a id="reset" class="btn btn-sm btn-pri-
mary">Сбросить</a>
        </div>
    </form>
```

```
    <table class="table table-condensed table-
striped table-bordered">
<thead><tr><th>Id</th><th>Имя</th><th>возраст</th><t
h></th></tr></thead>
        <tbody>
        </tbody>
    </table>
    <script>
    // получение всех пользователей
        async function GetUsers() {
            // отправляет запрос и получаем ответ
            const response = await fetch("/api/us-
ers", {
                method: "GET",
                headers: { "Accept": "applica-
tion/json" }
            });
            // если запрос прошел нормально
            if (response.ok === true) {
                // получаем данные
                const users = await response.json();
                let rows = document.querySelec-
tor("tbody");
                users.forEach(user => {
                    // добавляем полученные элементы
в таблицу
                    rows.append(row(user));
                });
            }
        }
        // получение одного пользователя
        async function GetUser(id) {
            const response = await fetch("/api/us-
ers/" + id, {
                method: "GET",
                headers: { "Accept": "applica-
tion/json" }
            });
            if (response.ok === true) {
                const user = await response.json();
                const form = docu-
ment.forms["userForm"];
                form.elements["id"].value = user.id;
```

```
                form.elements["name"].value =
user.name;
                form.elements["age"].value =
user.age;
            }
        }
        // добавление пользователя
        async function CreateUser(userName, userAge)
{
            const response = await fetch("api/us-
ers", {
                method: "POST",
                headers: { "Accept": "applica-
tion/json", "Content-Type": "application/json" },
                body: JSON.stringify({
                    name: userName,
                    age: parseInt(userAge, 10)
                })
            });
            if (response.ok === true) {
                const user = await response.json();
                reset();
                document.querySelector("tbody").ap-
pend(row(user));
            }
        }
        // изменение пользователя
        async function EditUser(userId, userName,
userAge) {
            const response = await fetch("api/us-
ers", {
                method: "PUT",
                headers: { "Accept": "applica-
tion/json", "Content-Type": "application/json" },
                body: JSON.stringify({
                    id: userId,
                    name: userName,
                    age: parseInt(userAge, 10)
                })
            });
            if (response.ok === true) {
                const user = await response.json();
                reset();
```

```
                document.querySelector("tr[data-
rowid='" + user.id + "']").replaceWith(row(user));
            }
        }
        // Удаление пользователя
        async function DeleteUser(id) {
            const response = await fetch("/api/us-
ers/" + id, {
                method: "DELETE",
                headers: { "Accept": "applica-
tion/json" }
            });
            if (response.ok === true) {
                const user = await response.json();
                document.querySelector("tr[data-
rowid='" + user.id + "']").remove();
            }
        }
        // сброс формы
        function reset() {
            const form = document.forms["userForm"];
            form.reset();
            form.elements["id"].value = 0;
        }
        // создание строки для таблицы
        function row(user) {
            const tr = document.createElement("tr");
            tr.setAttribute("data-rowid", user.id);
            const idTd = document.createEl-
ement("td");
            idTd.append(user.id);
            tr.append(idTd);
            const nameTd = document.createEl-
ement("td");
            nameTd.append(user.name);
            tr.append(nameTd);

            const ageTd = document.createEl-
ement("td");
            ageTd.append(user.age);
            tr.append(ageTd);
            const linksTd = document.createEl-
ement("td");
```

```
            const editLink = document.createEl-
ement("a");
            editLink.setAttribute("data-id",
user.id);
            editLink.setAttribute("style", "cur-
sor:pointer;padding:15px;");
            editLink.append("Изменить");
            editLink.addEventListener("click", e =>
{
                e.preventDefault();
                GetUser(user.id);
            });
            linksTd.append(editLink);
            const removeLink = document.createEl-
ement("a");
            removeLink.setAttribute("data-id",
user.id);
            removeLink.setAttribute("style", "cur-
sor:pointer;padding:15px;");
            removeLink.append("Удалить");
            removeLink.addEventListener("click",
e => {
                e.preventDefault();
                DeleteUser(user.id);
            });
            linksTd.append(removeLink);
            tr.appendChild(linksTd);
            return tr;
        }
        // сброс значений формы
document.getElementById("reset").click(function (e)
{
            e.preventDefault();
            reset();
        })
        // отправка формы
document.forms["userForm"].addEventListener("sub-
mit", e => {
            e.preventDefault();
            const form = document.forms["userForm"];
            const id = form.elements["id"].value;
            const name = form.ele-
ments["name"].value;
            const age = form.elements["age"].value;
```

```
            if (id == 0)
                CreateUser(name, age);
            else
                EditUser(id, name, age);
        });
        // загрузка пользователей
        GetUsers();
    </script>
</body>
</html>
```

Основная логика здесь заключена в коде `javascript`. При загрузке страницы в браузере получаем все объекты из БД с помощью функции `GetUsers`:

```
async function GetUsers() {
    // отправляет запрос и получаем ответ
    const response = await fetch("/api/users", {
        method: "GET",
        headers: { "Accept": "application/json" }
    });
    // если запрос прошел нормально
    if (response.ok === true) {
        // получаем данные
        const users = await response.json();
        let rows = document.querySelector("tbody");
        users.forEach(user => {
            // добавляем полученные элементы в таб-
лицу
            rows.append(row(user));
        });
    }
}
```

16 Для добавления строк в таблицу используйте функцию `row()`, которая возвращает строку. В этой строке будут определены ссылки для изменения и удаления пользователя.

Ссылка для изменения пользователя с помощью функции `GetUser()` получает с сервера выделенного пользователя:

```
async function GetUser(id) {
    const response = await fetch("/api/users/" + id,
{
        method: "GET",
        headers: { "Accept": "application/json" }
```

```
    });
    if (response.ok === true) {
        const user = await response.json();
        const form = document.forms["userForm"];
        form.elements["id"].value = user.id;
        form.elements["name"].value = user.name;
        form.elements["age"].value = user.age;
    }
}
```

Выделенный пользователь добавляется в форму над таблицей. Эта же форма применяется и для добавления объекта. С помощью скрытого поля, которое хранит `id` пользователя, можно узнать, какое действие выполняется – добавление или редактирование. Если `id` равен `0`, то выполняется функция `CreateUser`, которая отправляет данные в `POST`-запросе:

```
async function CreateUser(userName, userAge) {

    const response = await fetch("api/users", {
        method: "POST",
        headers: { "Accept": "application/json",
"Content-Type": "application/json" },
        body: JSON.stringify({
            name: userName,
            age: parseInt(userAge, 10)
        })
    });
    if (response.ok === true) {
        const user = await response.json();
        reset();
        document.querySelector("tbody").ap-
pend(row(user));
    }
}
```

17 Если ранее пользователь был загружен на форму, и в скрытом поле сохранился его `id`, то выполняется функция `EditUser`, которая отправляет `PUT`-запрос:

```
async function EditUser(userId, userName, userAge) {
    const response = await fetch("api/users", {
        method: "PUT",
        headers: { "Accept": "application/json",
"Content-Type": "application/json" },
```

```
        body: JSON.stringify({
            id: userId,
            name: userName,
            age: parseInt(userAge, 10)
        })
    });
    if (response.ok === true) {
        const user = await response.json();
        reset();
        document.querySelector("tr[data-rowid='" +
user.id + "']").replaceWith(row(user));
    }
}
```

18 Запустите приложение, обратитесь в браузере по адресу "*http://localhost:3000*" и вы сможете управлять пользователями, которые хранятся в файле `json`. Проиллюстрируйте это.

Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Реализация решения задач с подробным рассмотрением (описанием) решений.
5 Выводы.
6 Список использованных источников.

Контрольные вопросы

1 Для чего используют `API`?
2 Для чего нужен `Web API`?
3 Что такое `API` для интеграции?
4 Почему `API` так популярны?
5 Что такое архитектуры `API` с передачей данных по протоколу `HTTP`?
6 В чем заключаются основные идеи `REST API`?
7 Для чего рекомендуется оставлять работоспособной предыдущую версию при внесении изменений в логику работы вашего `REST API`, если оно является публичным?
8 Надо ли добавлять номер версии в новые маршруты?
9 С помощью какого инструментария можно упростить выполнение основных задач веб-сервера в *Node.js*?
10 Что является стандартным инструментом для создания серверной части на основе `REST API`?

11 Для чего применяется инструментарий `Mongoose`?

12 Надо ли вводить хешированный пароль?

13 Как получить список пользователей?

Список использованных источников

1 Buckler, C. Node.js: Novice to Ninja / C. Buckler. – SitePoint, 2022. – 388 p.

2 Erasmus, M. CoffeeScript Programming with jQuery, Rails, and Node.js / M. Erasmus. – Packt Publ., 2012. – 140 p.

3 Node.js. Notes for Professionals. – GoalKicker.com, 2018. – 333 p.

4 Wilson, J. Node.js 8 the Right Way / J. Wilson. – Pragmatic Bookshelf, 2018. – 336 p.

5 Пауэрс, Ш. Изучаем Node. Переходим на сторону сервера / Ш. Пауэрс. – 2-е изд., доп. и перераб. – СПб. : Питер, 2017. – 304 с.

6 Создание API [Электронный ресурс]. – 2023. – Режим доступа: https://metanit.com/web/nodejs/4.11.php.

7 Янг, А. Node.js в действии / А. Янг, Б. Мек, М. Кантелон. – 2-е изд. – СПб. : Питер, 2018. – 432 с.

Св. план 2022, поз. 12

Учебное издание

Алексеев Виктор Федорович
Лихачевский Дмитрий Викторович
Пискун Геннадий Адамович
Шаталова Виктория Викторовна

ПРОГРАММНЫЕ ИННОВАЦИОННЫЕ ПЛАТФОРМЫ ИНФОРМАЦИОННЫХ СИСТЕМ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 05.08.2024. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 9,42. Уч.-изд. л. 6,8. Тираж 30 экз. Заказ 140.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск