КАФЕДРЕ ИНФОРМАЦИОННЫХ СИСТЕМ
И ТЕХНОЛОГИЙ СПБГЛТУ 50 ЛЕТ

А. М. ЗАЯЦ,
Н. П. ВАСИЛЬЕВ

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ

ВВЕДЕНИЕ В FRONTEND И BACKEND РАЗРАБОТКУ НА JAVASCRIPT И NODE.JS

УЧЕБНОЕ ПОСОБИЕ

Издание второе, стереотипное

САНКТ-ПЕТЕРБУРГ
МОСКВА
КРАСНОДАР
2020

УДК 004
ББК 32.973-018.1я73

З 40 **Заяц А. М.** Проектирование и разработка web-приложений. Введение в frontend и backend разработку на JavaScript и node.js : учебное пособие / А. М. Заяц, Н. П. Васильев. — Санкт-Петербург : Лань, 2020. — 120 с. : ил. — (Учебники для вузов. Специальная литература). — Текст : непосредственный.

ISBN 978-5-8114-5278-1

Изложены основы работы с объектной моделью документа, положенной в основу динамического формирования и изменения содержимого HTML-страниц, с помощью языка программирования JavaScript и библиотеки jQuery. Рассматриваются основы backend разработки web-приложений программирования на стороне сервера на платформе node.js. Для закрепления и более глубокого изучения теоретического материала рассмотрен пример разработки приложения для выполнения простейших расчетов на серверной стороне и динамического формирования содержимого HTML-страниц с результатами этих расчетов в табличном и графическом виде на стороне клиента. Предполагается, что читатель владеет основами языков HTML, CSS и базового языка JavaScript.

Предназначено для студентов, обучающихся по направлениям «Информационные системы и технологии», «Лесное дело», профиль «Информационные системы и технологии в лесном хозяйстве».

УДК 004
ББК 32.973-018.1я73

Рецензент
И. В. ИВАНОВА — доктор технических наук, профессор кафедры информационных систем и вычислительной техники Санкт-Петербургского горного университета.

Обложка
Е. А. ВЛАСОВА

Оглавление

Введение

Даже самые простые web-приложения обычно состоят из двух частей: клиентской и серверной. Клиентская часть выполняется под управлением web-обозревателя и часто называется frontend разработкой (frontend – то есть фронтальная часть, с которой непосредственно работает пользователь). Серверная часть скрыта от пользователя – работает на заднем плане – и по этой причине часто называется backend разработкой.

Наиболее популярным языком программирования клиентской части является JavaScript (менее популярной альтернативой является язык VBScript).

С появлением платформы node.js стала возможной backend разработка на языке JavaScript. Таким образом, появилась возможность писать web-приложения на одном языке. Отметим также уникальное быстродействие движка V8 от компании Google, на котором построена платформа node.js. Этот движок исключает промежуточное формирование и использование байт-кода в процессе интерпретации языка JavaScript. Кроме того, node.js позволяет отказаться от использования традиционных web-серверов (например, apache), поскольку обладает собственными простыми средствами быстрой и эффективной разработки web-сервисов.

В первой части пособия приводятся основные сведения из теории манипулирования содержимым страницы с помощью клиентского JavaScript. Клиентский язык значит работающий на стороне и под управлением клиента (то есть web-обозревателя) с объектной моделью документа (DOM), формируемой и предоставляемой обозревателем. Если базовый язык JavaScript относительно прост в освоении (без замыканий, манипуляций контекстом, наследования на прототипах и т. д.), то клиентский язык является довольно сложным. Точнее сказать, сложной является модель DOM, с которой работает клиентский язык. Дело в том, что методы работы и сама модель различны для различных web-обозревателей и до недавнего времени буквально были напичканы всевозможными исключениями из правил. Особенно отличается от общих стандартов Internet Explorer вплоть до девятой версии. В пособии уделяется внимание только общепринятым методам работы с DOM. Изложение и изучение приёмов работы, учитывающих особенности различных браузеров (особенно Internet Explorer), представляется нецелесообразным поскольку:

- большинство современных обозревателей придерживаются общих стандартов, разработанных консорциумом W3C;

- существует большое количество библиотек JavaScript, которые обеспечивают кросс-браузерную работу и скрывают особенности DOM для различных клиентов.

Среди библиотек наиболее популярна jQuery. Библиотека является бесплатной, проста в освоении, имеет небольшой размер, хорошо документирована. Основы работы с jQuery излагаются в пособии. Здесь же уделяется внимание современной технологии AJAX (от *англ*. Asynchronous Javascript and XML) загрузки данных.

Полное изложение языка JavaScript, включая клиентский язык, можно найти в фундаментальном труде [1]. В этой книге также можно найти руководство по jQuery и здесь же содержится подробный справочник по языку. Отметим, что в глобальной сети опубликовано огромное количество всевозможных ресурсов (online-учебников, руководств, справочников), посвящённых JavaScript, DOM и jQuery, в том числе и pdf-версия руководства [1].

Что касается серверной разработки на node.js, то рассматриваются лишь основные положения, необходимые для быстрого старта (поскольку не преследовалась задача полного изложения столь объемной темы). Здесь весьма поучительным является пример разработки чата зарегистрированных пользователей, тем более что он использует актуальную технологию взаимодействия клиента и сервера WebSocket. Более полное изложение node.js можно найти в книгах [2, 3], а также в Глобальной сети, например в [4–10].

Этих сведений вполне достаточно для освоения материала второй части пособия, где на конкретном примере реализуются указанные технологии. Здесь также даны задания, которые могут быть использованы в курсовых работах или проектах.

Предполагается, что студент владеет начальными навыками разработки HTML-документов, их форматирования с помощью стилевых таблиц CSS, а также программирования на языке JavaScript. Тем не менее напомним один из наиболее популярных методов вывода отладочной информации в консоль браузера, который часто используется в представленных примерах кода. Речь идёт об объекте console и его методе log(). Окно консоли появляется после активизации инструментария web-разработчика. Как это делается, например, для firefox, показано на рисунках 1 и 2. Отметим, что аналогичные опции есть и в других современных браузерах.

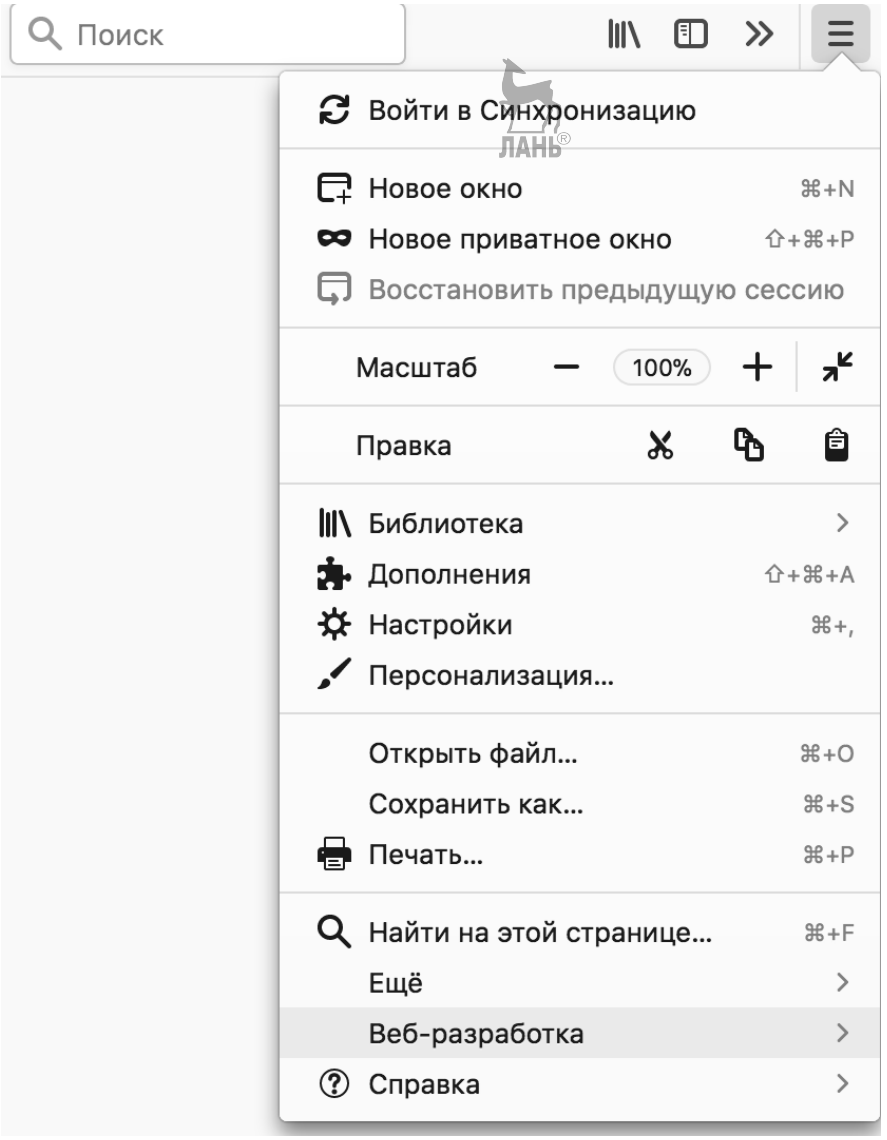


Рис. 1. Выбираем «Веб-разработка» для активизации инструментария разработчика

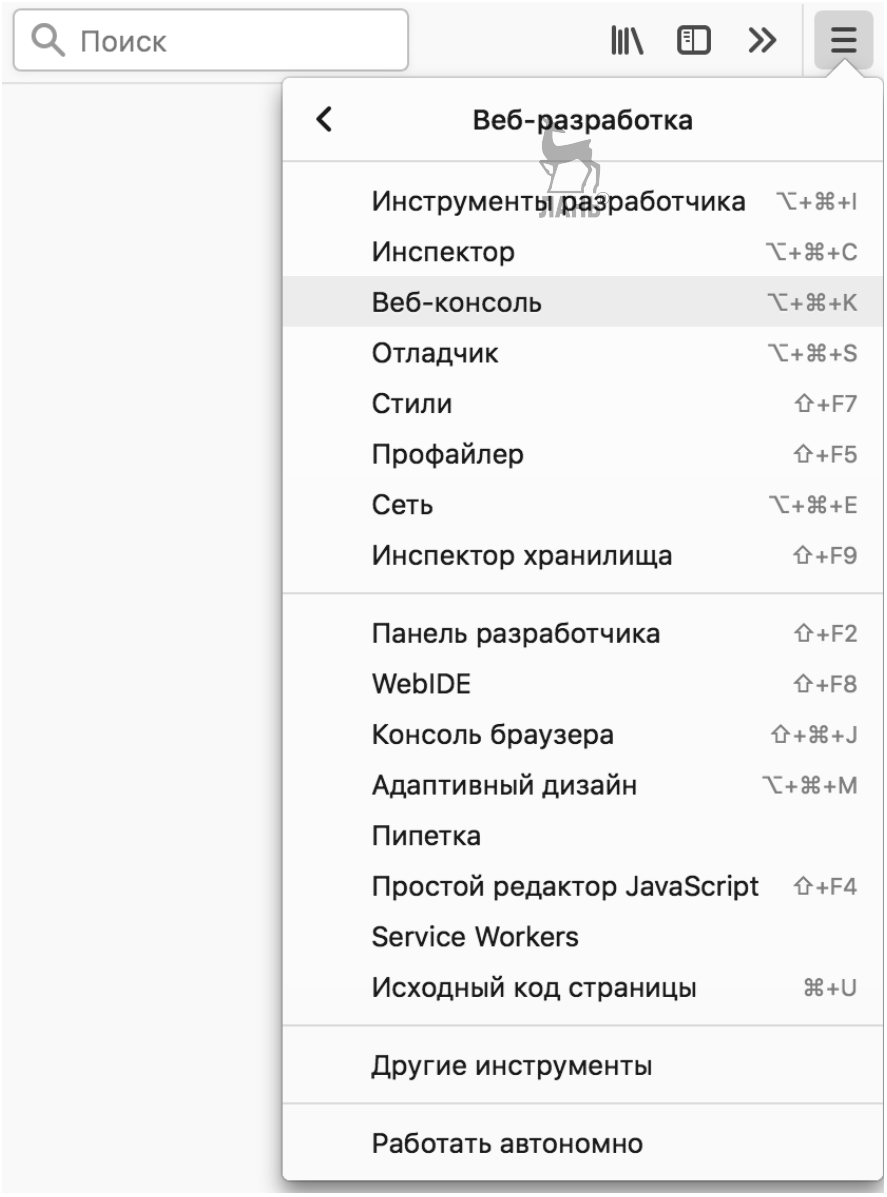


Рис. 2. Выбираем «Веб-консоль» в меню, которое появляется после выбора «Веб-разработка»

Теперь можно вывести любую отладочную информацию в окно консоли из любой точки кода JavaScript. Чтобы убедиться в этом, например, можно набрать в адресной строке браузера:

```
javascript:console.log("Hello World!");
```

Результат представлен на рисунке 3.

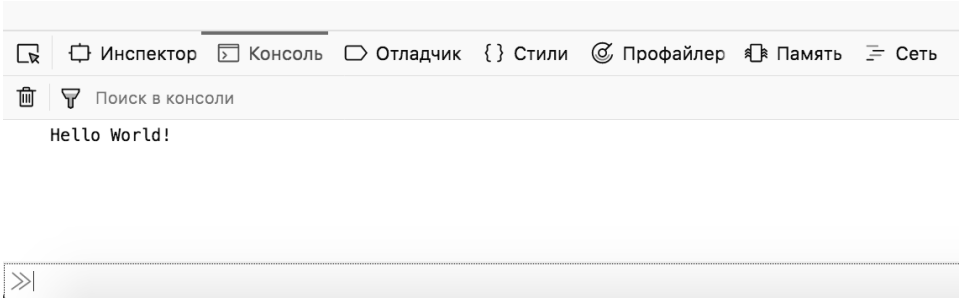


Рис. 3. Окно разработчика

Отметим, что конкретный вид интерфейса, конечно, может отличаться для других браузеров или для разных версий одного и того же браузера. Также заметим, что интрументарий разработчика включает ряд других полезных опций: отладчик, инспектор, сеть и т. д., которые могут оказаться весьма полезными для анализа и отладки web-приложения.

1. Объектная модель документа и клиентский JavaScript

1.1. Введение

Объектная модель документа (DOM – Document Object Model) – это прикладной программный интерфейс – Application Program Interface (API), определяющий порядок доступа и манипулирования объектами, представляющими различные структурные части HTML-документа.

Программный интерфейс – это набор классов, в соответствии с которыми и создаются экземпляры объектов модели. Классы (а значит, и инстанции этих классов) предоставляют методы (функции) и свойства, которые позволяют манипулировать и изменять объекты, а значит, и представленные этими объектами структурные части HTML-документов. Классы выстроены в своеобразную иерархию наследования свойств и методов друг друга согласно принципам объектно-ориентированного программирования. Базовым в этой иерархии является класс Node. Более подробно иерархия наследования классов DOM рассматривается далее.

Консорциумом W3C был выработан стандарт DOM, достаточно полно поддерживаемый всеми современными браузерами. Развитие DOM имеет свою историю, и его устоявшийся вариант является результатом конкурентной борьбы таких фирм, как Microsoft, Netscape и др. По этой причине может оказаться, что некоторые браузеры не поддерживают стандарт консорциума, или поддерживают не полностью, или предлагают свою модель. Это относится прежде

всего к Internet Explorer. В настоящем пособии специфика таких обозревателей не рассматривается.

Интерфейс DOM реализован в различных языках программирования, бесспорным лидером среди которых является JavaScript. Модель формируется браузером в ходе загрузки документа и представляется в виде иерархической структуры объектов с рядом свойств и методов, доступных в программе. Поскольку разработка ведётся на стороне браузера, то есть клиента, то и язык называется клиентским.

Подробное изложение материала по клиентскому языку JavaScript и DOM можно найти, например, в [1]. В данном пособии рассмотрим лишь наиболее востребованные приёмы работы.

1.2. Концепция объектной модели документа

Все, что содержится в HTML-документе, адекватно представляется объектами, которые создаются в процессе загрузки документа браузером и в дальнейшем доступны в сценариях JavaScript. Каждый объект обладает рядом свойств, методов и событий, посредством которых и реализуется управление объектами, а через них и содержимым документа. Все объекты связаны между собой и образуют иерархическую древовидную структуру, адекватно отражающую структуру самого документа.


Не путать иерархическую структуру документа с иерархией классов объектов модели DOM.

Во главе этой структуры находится сам документ. Документ содержит различные элементы, представленные тэгами HTML. Эти элементы также могут содержать соподчиненные элементы, и т. д. Так, документ обычно содержит заголовок (тэг HEAD) и тело (тэг BODY). Тело может содержать, например, заголовки (тэги H1, H2, …), таблицы, рисунки и пр. Каждый тэг представлен объектом. Каждый воспроизводимый браузером визуальный элемент представлен тэгом или тэгами, специфика которых может быть изменена в сценариях опосредованно через объекты, представляющие эти элементы. Элементы, не имеющие визуального представления, также представляются объектами в модели.

Подтвердим сказанное простым примером. С этой целью рассмотрим следующий HTML-код:

```
<html>
  <head>
    <title>Моя страница</title>
  </head>
  <body>
    <h1>Добро пожаловать на мою страницу!</h1>
    <p>Это очень <strong>важный документ</strong></p>
  </body>
</html>
```

Здесь и далее работу приведённых фрагментов программных кодов можно увидеть, используя web-редактор, например WebCoder 1.6.4.0. Этот способ можно использовать при разработке собственных программных кодов при выполнении заданий, приведенных в пособии.

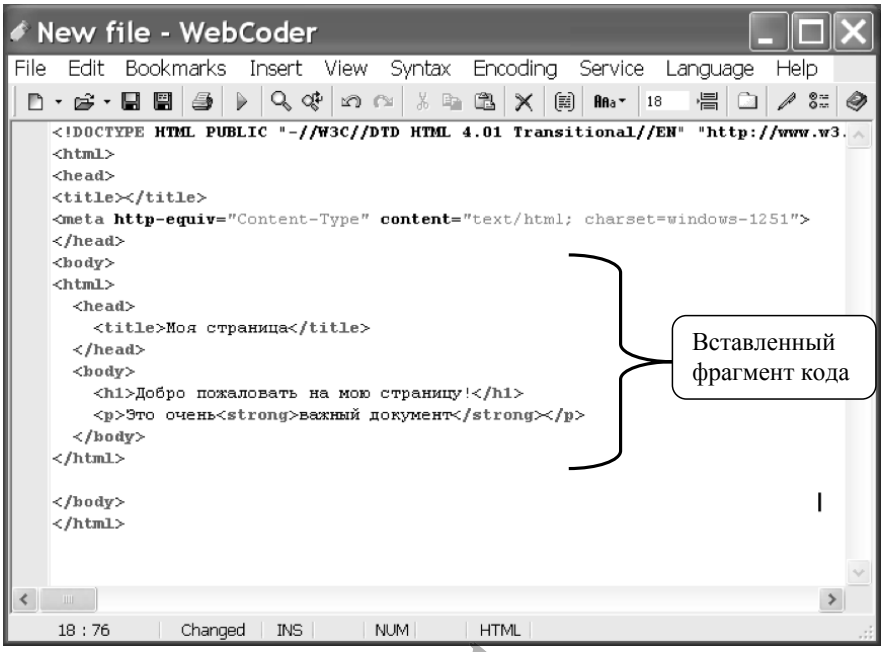


Рис. 4. Рабочее окно web-редактора

Добро пожаловать на мою страницу!

Это очень **важный документ**

Рис. 5. Результат реализации кода в окне браузера

Объектная модель, соответствующая этому документу, будет иметь следующую иерархическую структуру:

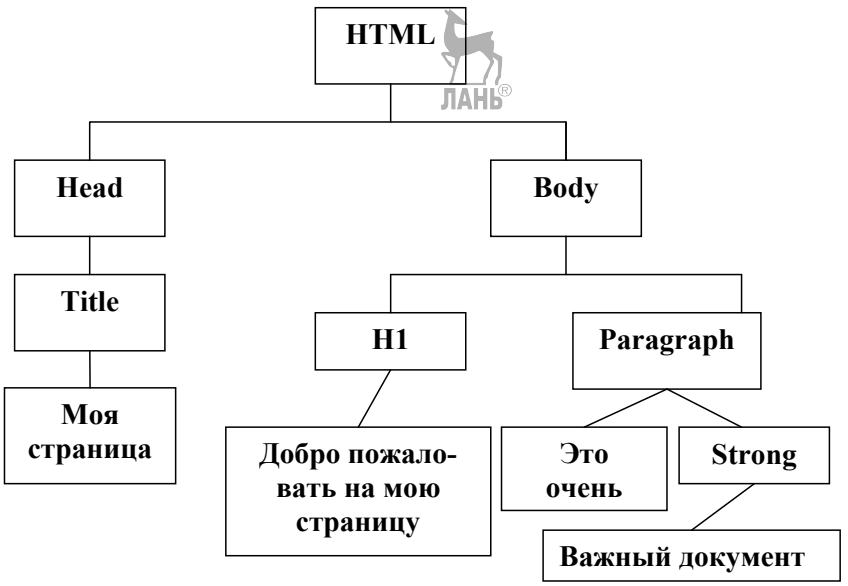


Рис. 6. Пример связей между объектами, представляющими HTML-документ

Древовидная структура DOM, изображенная на рисунке, содержит объекты различных типов или, согласно устоявшейся для объектно-ориентированных языков терминологии, различных классов. Все классы наследуют базовый класс Node (см. рис. 7).

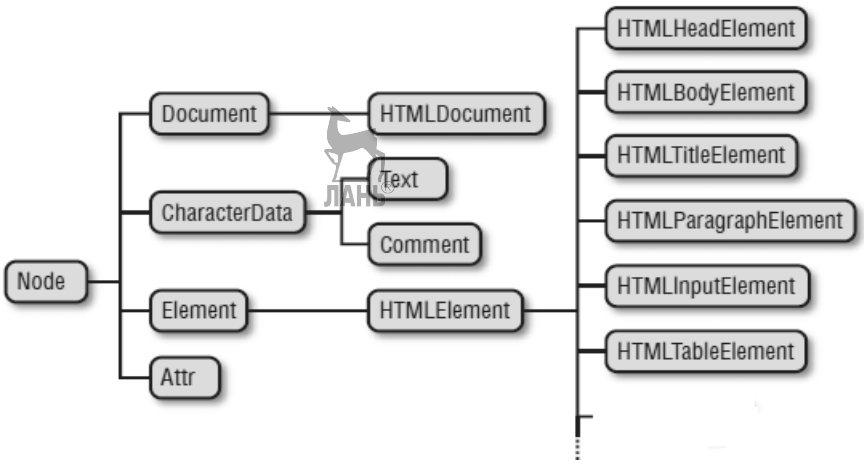


Рис. 7. Иерархия наследования классов

Нетрудно догадаться, к какому классу принадлежит тот или иной объект, представляющий тот или иной тэг документа. Например, HTMLBodyElement – это класс объектов, представляющих тэг body – тело документа. Львиную долю образуют подклассы класса HTMLElement. Объекты этих классов часто называют элементными узлами DOM. Это наиболее часто встречающиеся в практике программирования узлы. Далее по популярности идут текстовые узлы – подклассы Text.

Интерфейс базового класса Node определяет свойства и методы для перемещения по дереву и манипуляций им. Свойство childNodes возвращает список дочерних узлов, свойства firstChild, lastChild, nextSibling, previousSibling и parentNode предоставляют средство обхода узлов дерева.

Такие методы, как appendChild(), removeChild(), replaceChild() и insertBefore(), позволяют добавлять узлы в дерево документа и удалять их.

Базовый класс Node также предоставляет свойство nodeType, которое позволяет определить тип узла. Узел типа Document в этом свойстве имеет значение 9. Вместо числового значения можно использовать именованную константу Node.DOCUMENT_NODE, что лучше запоминается и делает код читаемым.

Для узлов типа Element это значение – 1 (Node.ELEMENT_NODE),
типа Text – 3 (Node.TEXT_NODE),
типа Comment – 8 (Node.COMMENT_NODE),
типа DocumentFragment – 11 (Node.DOCUMENT_FRAGMENT_NODE),
типа Attr – 2 (Node.ATTRIBUTE_NODE).

Другое свойство nodeValue содержит текст узлов типа Text и Comment. Свойство nodeName позволяет определить название тэга, все символы которого преобразованы в верхний регистр.

Окно обозревателя представлено объектом window (это и есть ссылка на этот объект). Следует подчеркнуть, что этот объект не входит в DOM. Ссылка на документ может быть получена следующим образом: window.document. Ссылка на окно при этом может быть упущена, то есть к документу можно обратиться и так: document. Вообще ссылку window указывать не обязательно при обращении к его свойствам и методам. Свойства и методы объектов указываются через точку. То есть document является свойством окна. Среди свойств окна, например, есть свойство location, которое, в свою очередь, также представляет собой ссылку на объект, концентрирующий всё необходимое для получения данных о расположении документа в Сети.

Свойства могут быть доступны только для чтения или для чтения и записи. Если свойство доступно только для чтения, то оно не может быть изменено посредством сценария. В противном случае, свойство доступно для записи. Перезапись свойства, как правило, вызывает некоторое активное действие со сто-

роны обозревателя. Так, например, перезапись свойства href объекта location приводит к загрузке нового документа, расположенного по новому адресу. Адрес указывается в виде строки в качестве нового значения свойства href.

Объект window играет роль глобального объекта, а это значит, что все глобальные переменные будут являться его свойствами. У этого объекта много полезных свойств и методов, которые мы будем привлекать по мере необходимости.

1.3. Работа с DOM

1.3.1. Выбор объектов

Прежде чем выполнить какие-то изменения в документе, обычно приходится целенаправленно выбирать те или иные элементы документа. Это одна из наиболее востребованных задач. Выбор можно осуществить по:

- идентификатору тэга, то есть по значению атрибута id;
- значению атрибута name;
- названию тэга;
- имени класса или классов CSS;
- совпадению с определенным селектором CSS.

Выбор по id. Любой тэг может иметь атрибут id. Значение этого атрибута должно быть уникальным (поскольку это идентификатор тэга и соответствующего элемента документа). Выбор осуществляется так:

```
var myel = document.getElementById("myelId");
```

Выбор по name. Аналогично работает атрибут name. Однако при этом надо учитывать, что значение этого атрибута не обязательно должно быть уникальным, а также, что его следует использовать только для некоторых элементов, среди которых формы (тэг form), поля формы (тэги input, combobox и др.), рисунки (img) и др. Выбор осуществляется так:

```
var fields = document.getElementsByName("taxFields");
```

Возвращает эта функция объект специального класса NodeList, который входит в спецификацию DOM. Работать с этим объектом можно как с обычным массивом объектов типа Element.

Выбор по названию тэга. Рассмотрим фрагмент кода:

```
var firstpara = document.getElementsByTagName("p")[0];
var firstParaSpans =
firstpara.getElementsByTagName("span");
```

Анализируя этот фрагмент, можно заключить, что метод getElementsByTagName работает не только для документа, но и для других элементов. Однако в первом случае выбор всех параграфов (тэг «p») производится во всем документе, во втором случае отбор элементов «span» производится только в рамках первого параграфа. Отметим, поскольку имена тэгов не чувствительны к регистру, то и задавать их можно без учёта регистра. Возвращается также объект типа NodeList. Наконец, если задать в качестве имени тэга «*», то будут отобраны все элементы с любыми именами тэгов.

Выбор по названию классов. Известно, что в любом тэге можно задать атрибут class и в качестве его значения определить название класса, для которого где-то в стилевой таблице определены те или иные стили. Наверное, не всем известно, что можно сочетать несколько классов для одного и того же элемента (указываются через пробел). Для выбора элементов с заданным классом (или классами) используется метод getElementsByClassName. В качестве аргумента могут быть указаны через пробел имена классов. В результате, будут отобраны элементы, для которых в атрибуте class заданы указанные классы. Метод определён для любого элемента, так же как и метод getElementsByTagName. Пример:

```
var yellowElements =
              document.getElementsByClassName("yellow
big");
```

Выбор по селектору css. Механизм определения стилей в css таблицах включает понятие селектора для отбора элементов документа, для которых требуется употребить эти стили. Это весьма мощный и гибкий инструмент. Оказывается, селектор можно указать в методах: querySelector() и querySelectorAll(). Методы определены для любого элемента и возвращают первый метод – первый найденный элемент, второй метод – список найденных узлов, то есть объект класса NodeList.

Примеры селекторов:

```
div, #idStr // Все элементы<div> плюс элемент с id="idStr"
p[lang="ru"] // Абзац с текстом на русском языке: <p
lang="ru">
*[name="x"] // Любой элемент с атрибутом name="x"
#log span // Любой элемент<span>, являющийся потомком
элемента с              //id="log"
#log>span // Любой элемент <span>, дочерний по отношению к
элементу с           // id="log"
body>h1:first-child // Первый элемент <h1>, дочерний по
отношению к          // <body>
```

Прежние способы выбора элементов. Многие браузеры продолжают поддерживать «устаревшие» способы выбора элементов. Например, свойство document.all указывает на коллекцию всех элементов документа, а document.images – на коллекцию всех изображений. Объекты, на которые ссылаются эти свойства, относятся к классу HTMLCollection. В первом приближении этот класс подобен массивам элементов документа. Таким образом, all[i] – это *i*-й объект (типа Element) коллекции, а вот как, например, можно осуществить просмотр всех элементов документа:

```
for(var i=0; i<document.all.length; i++)
                        alert( document.all[i].tagName );
```

Здесь для каждого тэга из коллекции all с помощью метода alert выводится название тэга. Любая коллекция имеет свойство length, определяющее длину коллекции, то есть количество объектов в ней. Это свойство позволяет ограничить диапазон изменения переменной цикла – номера просматриваемого объекта. Каждый объект, представляющий тэг документа, обязательно имеет свойство tagName, содержащее название тэга. Это название передается в качестве аргумента методу alert объекта window (ссылку window, как уже упоминалось выше, можно упустить). Метод alert показывает окно с сообщением, переданным в качестве аргумента, то есть название очередного тэга.

Следующий пример позволяет просмотреть адреса всех графических файлов, содержащихся в окне обозревателя:

```
for(var i=0; i<document.images.length; i++)
   alert(document.images[i].src);
```

Доступ к подсемейству объектов, представляющих тэги с одинаковыми названиями, возможен посредством метода tags, который в качестве аргумента принимает название тэга. Так, все тэги <h1> документа можно получить следующим образом:

```
document.all.tags("h1")
```

А самый первый заголовок h1 документа доступен посредством ссылки:

```
document.all.tags("h1")[0]
```

Если требуется организовать доступ к какому-то конкретному тэгу, конечно, опосредованно через объект, сгенерированный для этого тэга обозревателем, то можно назначить некоторый уникальный идентификатор тэгу. Идентификатор будет являться ссылкой на соответствующий объект. Идентификатор назначается с помощью атрибута id, например:

```
<h1 id=head>Заголовок</h1>
```

Помеченный таким образом заголовок в коде Java Script доступен посредством ссылки window.head (или просто head):

```
head.style.backgroundColor='red';
```

В приведенной строке кода добираемся до стилевого оформления объекта посредством свойства style, а затем до цвета фона объекта, которым является наш заголовок. Может оказаться, что несколько тэгов будут иметь одинаковый идентификатор (хотя такого быть не должно). В этом случае ссылка, совпадающая с таким идентификатором, будет ссылаться на семейство объектов.

Таким образом, объектная модель обладает весьма гибкими возможностями доступа к тем или иным объектам, их свойствам и методам. Следует еще раз напомнить, что полный список свойств и методов объектов можно найти в справочниках, например в [1], здесь же излагается только концепция модели.

1.3.2. Работа с атрибутами

Тэги HTML состоят из имени и множества пар имя/значение, известных как атрибуты. Например, тэг

```
<a href="ya.ru" target="_blank">Yandex</a>
```

определяет гиперссылку, а в качестве адреса назначения ссылки использует значение атрибута href. Значения атрибутов HTML-элементов доступны в виде свойств объектов HTMLElement, представляющих эти элементы. Но это касается стандартных атрибутов, то есть имеющих известные и определенные в стандарте имена (например, HTML5). Если имя атрибута нестандартное, то соответствующее свойство элемента может отсутствовать. Для работы с такими атрибутами DOM определяет другие механизмы получения и изменения их значений.

Класс Element определяет методы getAttribute() и setAttribute(), которые можно использовать для доступа к стандартным или нестандартным атрибутам тэгов:

```
var image = document.images[0];
var width = parseInt(image.getAttribute("HEIGHT"));
image.setAttribute("class", "Glamorous");
```

Следует отметить, что значения всех атрибутов эти методы всегда интерпретируют как строки, а имена атрибутов нечувствительны к регистру.

Другие два родственных метода – hasAttribute() и removeAttribute() – особенно удобны при работе с логическими атрибутами (например, атрибут disabled HTML-форм), для которых важно их наличие или отсутствие в элементе, а не их значения.

Существует также иной, менее популярный способ работы с атрибутами. Тип Node определяет свойство attributes. Это свойство имеет значение null для всех узлов, для которых атрибуты неприменимы, то есть не являющихся объектами Element. Свойство attributes объектов Element является объектом, подобным массиву, доступным только для чтения, представляющим все атрибуты элемента:

```
document.body.attributes[0] // Первый атрибут элемента
<body>
document.body.attributes.bgcolor // Атрибут bgcolor
элемента
                                 //<body>
document.body.attributes["ONLOAD"] // Атрибут onload
элемента
                                    // <body>
```

Результатом такой индексации будет объект типа Attr, у которого определены два свойства name и value.

1.3.3. Создание, вставка изменение и удаление узлов

Объект document предоставляет два метода, с помощью которых можно создавать узлы различных типов. Это методы createElement и createTextNode. Первый предназначен для создания объектов подклассов Element, а второй – для создания текстовых узлов (типа Text).

Класс Node определяет следующие методы для вставки узлов:

- метод appendChild – добавляет узел, заданный в качестве аргумента, в конец списка дочерних узлов, родителем которых является узел, для которого этот метод вызывается;
- метод insertBefore отличается от первого тем, что для него задается дополнительный аргумент – дочерний узел, перед которым вставляется узел, указанный в качестве первого аргумента.

Для удаления используется метод removeChild, который вызывается для родительского элемента, а удаляет дочерний узел, указанный в качестве аргумента.

Для замены узлов класс Node располагает методом replaceChild, который удаляет один дочерний узел и замещает его другим. Этот метод должен вызываться относительно родительского узла. В первом аргументе он принимает новый узел, а во втором – замещаемый узел. Например, ниже показано, как заменить узел n текстовой строкой:

```
n.parentNode.replaceChild(document.createTextNode("Заме
на"), n);
```

Технику использования этих методов позволяет изучить следующий пример. Пусть структура некоторой организации представлена в виде массива объектов:

```
var institutes = [{
 nm: 'Отделение естественнонаучного и гуманитарного об-
разования',
 chairs: [{
    nm: 'Кафедра физической культуры',
    rating: 10
 },{
    nm: 'Кафедра русского языка',
    rating: 11
 },{
    nm: 'Кафедра иностранных языков',
    rating: 20
 },{
    nm: 'Кафедра высшей математики',
    rating: 25
 },{
    nm: 'Кафедра Физики',
    rating: 10
 }]
},{
 nm :  'Институт леса и природопользования',
 chairs :  [{
    nm: 'Кафедра лесной таксации',
    rating: 9
 },{
    nm: 'Кафедра технологии лесозаготовительных произ-
водств',
    rating: 12
 },{
    nm: 'Кафедра лесоводства',
    rating: 15
 },{
    nm: 'Кафедра лесных культур',
    rating: 9
 },{
    nm: 'Кафедра информационных систем и технологий',
    rating: 11
 },{
    nm: 'Кафедра ботаники и дендрологии',
    rating: 10
 }]
}];
```

Понятно, что это список институтов, для каждого из которых задан список кафедр с указанием рейтинга. Требуется представить эти сведения в виде таблицы. Предполагается, что пустая таблица уже определена в документе:

```
<!doctype html>
<html>
<head>
<meta charset='utf-8'>
<script src='1.js' defer></script>
</head>
<body>
<table>
</table>
</body>
</html>
```

Здесь указан файл 1.js, в котором поместим массив institutes, представленный выше. Здесь же будет находиться и код, который решает поставленную задачу. Следует обратить внимание на атрибут defer, гарантирующий, что код в 1.js будет выполняться после загрузки всего документа и таблицы в том числе.

Код JavaScript, который следует дописать в 1.js, следующий:

```
(function () {
 var tbl = document.getElementsByTagName('table')[0];
 for(var i=0; i<institutes.length; i++) {
    var institut = institutes[i];
    var tr = document.createElement('tr'),
       td = document.createElement('td'),
       txt = document.createTextNode(institut.nm);
    td.setAttribute('colspan', '2');
    td.appendChild(txt);
    td.className = 'head';
    tr.appendChild(td);
    tbl.appendChild(tr);
    var chairs = institut.chairs;
    for(var j=0; j<chairs.length; j++) {
       var chair = chairs[j];
       var tr = document.createElement('tr'),
          td1 = document.createElement('td'),
          td2 = document.createElement('td'),
          txt1 = document.createTextNode(chair.nm),
          txt2 = document.createTextNode(chair.rating);
       td2.className = 'number';
       td1.appendChild(txt1); td2.appendChild(txt2);
       tr.appendChild(td1); tr.appendChild(td2)
       tbl.appendChild(tr);
    }
 }
})();
```

Код в представленном фрагменте реализован в виде анонимной (без имени) функции, которая тут же и вызывается на выполнении. Внешний цикл перебирает институты, внутренний – кафедры в институтах. В теле каждого цикла выполняются однотипные действия по созданию клеток и строк таблицы. Клетки таблицы под институты объединяются добавлением атрибута colspan.

Текстовое содержимое клеток создается как текстовые узлы с помощью метода createTextNode. Здесь следует упомянуть об альтернативном методе задания содержимого любых тэгов – это свойство элементных узлов innerHTML. Значением этого свойства может быть любой HTML-код, а не только простой текст. Например, вместо td1.appendChild(txt1) можно было бы просто написать td1.innerHTML = chair.nm, а не создавать текстовый узел txt1.

Наконец, клетки форматируются в результате определения для них стилевых классов. Стили классов определяются в таблице, которую следует добавить в заголовочную часть HTML-кода, представленного выше (например, перед тэгом script). Содержимое этой таблицы:

```
<style>
table {
 border-collapse: collapse;
}
td {
 padding: 0 5px;
 border: 1px solid black;
}
td.head {
 font-weight: bold;
 font-size: 1.1em;
 text-align: center;
 padding-top:5px;
 padding-bottom:5px;
 background-color: AntiqueWhite
}
td.number {
 text-align: right;
}
</style>
```

В заключение отметим, что создавать HTML-код можно также с помощью метода document.write, например:

```
document.write("<h1>Hello World!</h1>")
```

При этом следует помнить, что любая запись (методом write) в документ после его загрузки полностью его уничтожит и заменит новой записью. Если же

запись производится в процессе загрузки, то это равносильно вставке HTML-кода в соответствующее место документа (где находится тэг script (без атрибута defer), который содержит обращение к write).

1.3.4. Размеры и положение элементов документа

В процессе изменения или формирования содержимого документа с помощью объектной модели часто необходимо знать положение и размеры документа, его различных структурных частей и видимой части окна. Это бывает необходимо при создании всплывающих подсказок, различных меню и т. д.

В настоящее время стандартизован общий подход (методы, свойства) в работе с геометрией объектов DOM, который и рассматривается здесь. Однако следует помнить, что в старых версиях обозревателей и в Internet Explorer он неприменим. Рассматривать универсальные приемы работы в рамках данного пособия не представляется возможным. Надо отметить, что эти приемы изобилуют многочисленными проверками и выглядят довольно запутанными. Впрочем, это не столь актуально, поскольку такие браузеры встречаются всё реже, а библиотеки типа jQuery берут на себя всю заботу об универсальности.

Две системы координат. В процессе воспроизведения содержимого документа его различные структурные части определенным образом позиционируются с учетом их реальных размеров. Отсчет расстояний и размеров ведётся попиксельно. Позиция определяется относительно верхнего левого угла либо видимой части окна браузера, либо самого документа. Разница, очевидно, определяется величиной прокрутки документа по горизонтали и вертикали – это свойства: window.pageXOffset и window.pageYOffset. Для прокрутки документа к заданной точке следует пользоваться методом window.scrollTo(x, y). В результате, точка с координатами x и y переместится в левый верхний угол видимой части окна.

Размеры элементов. Размеры видимой части окна (без полос прокрутки) определяются свойствами: window.innerWidth и window.innerHeight.

Для определения размеров и положения относительно видимой части окна любого элемента служит метод этого элемента getBoundingClientRect(). Тогда положение элемента внутри документа с учетом скроллинга можно вычислить следующим образом:

```
var box = e.getBoundingClientRect(); // Координаты в
видимой
                                    //области
var x = box.left + window.pageXOffset; // Перейти к ко-
ординатам
                                      // документа
var y = box.top + window.pageYOffset;
```

Размеры элемента можно вычислить так:

```
var box = e.getBoundingClientRect();
var w = box.width || (box.right - box.left);
var h = box.height || (box.bottom - box.top);
```

Блочные элементы, такие как изображения, абзацы и элементы div, всегда отображаются браузерами в прямоугольных областях. Однако строчные элементы, такие как span, code и b, могут занимать несколько строк и таким образом состоять из нескольких прямоугольных областей. Например, некоторый курсивный текст (отмеченный тэгами и), разбитый на две строки.

Если метод getBoundingClientRect() вызывается для строчного элемента, то он вернет геометрию прямоугольника, содержащего все отдельные прямоугольные области. Для курсива, упомянутого выше, ограничивающий прямоугольник будет включать обе строки целиком. Для определения координат и размеров отдельных прямоугольников, занимаемых строчными элементами, можно воспользоваться методом getClientRects(). Этот метод возвращает доступный только для чтения объект, подобный массиву, чьи элементы представляют прямоугольники, аналогичные тем, что возвращаются методом getBoundingClientRect().

В заключение по поводу метода getBoundingClientRect() следует уточнить следующие важные моменты:

Положение элемента, то есть координаты левого верхнего угла (left, top) и правого нижнего (right, bottom), вычисляется с учетом рамки (border) и отступов (padding), но без учёта полей (margin).

В некоторых браузерах также вычисляются ширина (width) и высота (height). То есть возвращаемый методом объект также будет содержать и эти свойства.

1.3.5. Изменение стилей

Клиентский язык JavaScript позволяет манипулировать стилями, таблицами стилей, менять и добавлять классы элементов документа. Из столь богатого арсенала инструментов рассмотрим наиболее востребованные.

Если элемент выбран (например, одним из способов, перечисленных выше), то его стили можно менять следующим образом:

```
e.style.fontSize = "24pt";
e.style.fontWeight = "bold";
e.style.backgroundColor = "yellow";
```

Следует обратить внимание на следующую особенность: вместо font-size используется fontSize, а вместо font-weight – fontWeight. Это правило работает

для всех составных стилей. Следует также помнить, что значения стилей всегда строковые, а для количественных значений следует указывать единицы измерения.

Можно непосредственно установить значение атрибута style одним из двух способов:

```
var s = "font-size:24pt;font-weight:bold;background-
color:yellow";
e.setAttribute("style", s);
// второй способ
// e.style.cssText = s; //cssText доступно также по
чтению
```

Класс элемента (атрибут class) меняется с помощью свойства className. Это свойство также доступно по чтению.

Стили элемента, которые обозреватель непосредственно использует при его воспроизведении, являются результатом применения правил из различных источников: стилевые таблицы, значение атрибута style, собственные стили обозревателя. Это может быть довольно сложный каскад определений. Вычисленный результат (вычисленные стили) можно получить с помощью функции getComputedStyle(). Например, с помощью этой функции получим реальный размер шрифта элемента и увеличим его в два раза:

```
var size = par-
seInt(window.getComputedStyle(e,null).fontSize);
e.style.fontSize = 2*size + "px";
```

Размер шрифта всегда будет вычислен в пикселах.

1.4. Обработка событий

1.4.1. Асинхронная модель программирования

События – это способ программирования реакций на непредвиденные ситуации. Такие ситуации возникают независимо или асинхронно относительно основной последовательности выполнения команд программы. Иногда говорят об асинхронной модели программирования. Модель предусматривает две составляющие – источники событий и реакции на события – обработчики событий.

Например, при передвижении курсора мыши через некоторый элемент документа, имеющий визуальное представление, генерируется несколько событий: при входе курсора в область элемента – событие mouseover, при выходе – mouseout, при перемещении в рамках элемента – mousemove. Источником здесь является элемент документа. Строка «mouseover» – это тип события.

Другой пример: при загрузке и выгрузке документа обозреватель генерирует несколько событий для объекта window, среди которых наиболее используемые – это load и unload. Первое генерируется сразу после загрузки всего документа, а второе, наоборот, перед началом выгрузки документа и освобождением всех ресурсов, связанных с этим документом. Современные браузеры предоставляют широкий спектр событий для различных элементов и ситуаций, позволяющих изменить специфику поведения документа в целом. На каждое событие можно отреагировать, написав соответствующий программный код. При этом возникают следующие вопросы:

- как оформить и связать программный код (обработчик события) с событием;
- как получить доступ к источнику события и другим данным, описывающим специфику события (например, как получить координаты курсора мыши);
- как отказаться от стандартной (по умолчанию) реакции обозревателя на событие, если таковая предусмотрена.

Кроме того, событийная модель предусматривает две стадии распространения события: захват и всплытие. Можно перехватить событие на каждой из стадий и при необходимости прекратить распространение события.

1.4.2. Распространение событий

Рассмотрим следующий простой пример:

```
<table border=1><tr>
<td><a href="home.htm" target='_blank'><img
src="home.png">Go Home</a></td>
<td>cell 1,2</td>
</tr><tr>
<td>cell 2,1</td>
<td>cell 2,2</td>
</tr>
</table>
```

Если пользователь «кликнет» мышью по рисунку, то возникнет событие click. Первая стадия распространения события по дереву объектной модели документа называется захватом (capture). Событие стартует с самого верхнего узла (document) и далее спускается вниз через все промежуточные узлы к источнику события – рисунку. Источник в английской нотации определяется как цель – target. Следует отметить, что в большинстве случаев событие стартует еще выше, то есть с объекта window. Вторая стадия называется «всплыванием» – событие начинает «всплывать» по дереву документа в обратную сторону от рисунка к документу (окну) через все промежуточные узлы.

Для каждого промежуточного узла можно зарегистрировать обработчик события на любой из стадий.

1.4.3. Регистрация обработчиков событий

Для приведенного в предыдущем пункте примера зарегистрируем обработчик события click для рисунка на стадии захвата и на стадии всплытия. На стадии захвата событие пройдёт сверху вниз через таблицу (table), строку (tr), клетку (td), ссылку (a) и, наконец, достигнет рисунка (img). На стадии всплытия событие пройдет через указанные элементы в обратном порядке. Регистрация обработчиков выполняется с помощью функции addEventListener() следующим образом:

```
var img = document.getElementsByTagName('img')[0];
img.addEventListener('click', imgHandle, true);
img.addEventListener('click', imgHandleBubble, false);
function imgHandle(e){ console.log('img_capture:' +
e.target.tagName); }
function imgHandleBubble(e){ console.log('img_bubble:'
+ e.target.tagName); }
```

Первый параметр функции addEventListener() – это тип события, второй – обработчик события, третий необязательный параметр определяет стадию перехвата события (true – для стадии захвата, false – для стадии всплытия). Обработчик события – это имя функции imgHandle и imgHandleBubble. Таким образом, в приведённом фрагменте для рисунка регистрируются два обработчика для обеих стадий. Как и следовало ожидать, в окно консоли обозревателя будет выведено сначала сообщение «img_capture:IMG» (стадия захвата), а затем – «img_bubble:IMG» (стадия всплытия).

Обработка события будет происходить только при клике на рисунке. Предположим, что требуется обработка события при клике как на рисунке, так и на тексте «Go Home», тогда, очевидно, можно зарегистрировать событие для ссылки, или для клетки таблицы. А если, например, зарегистрировать обработчик события для таблицы, то он будет срабатывать при клике в любом месте таблицы.

Отменить регистрацию обработчика события можно с помощью метода removeEventListener(), которому следует передать те же самые параметры, что и при регистрации. Наконец, отметим, что можно регистрировать для одного и того же элемента и события несколько обработчиков, которые будут срабатывать последовательно в порядке их регистрации.

Прекращение процесса распространения события. Предположим, что в приведённом фрагменте кода требуется прекратить распространение события

на стадии захвата в обработчике imgHandle. Оказывается это можно сделать, если вызвать метод объекта события stopPropagation(). Этот метод предотвратит стадию всплытия, однако, вопреки ожиданиям, не предотвратит выполнение обработчика imgHandleBubble, поскольку он зарегистрирован на том же самом элементе. В данном случае поможет метод stopImmediatePropagation():

```
function imgHandle(e){ console.log('img_capture:' +
e.target.tagName); e.stopImmediatePropagation(); }
```

Предотвращение действия по умолчанию. Прекращение процесса распространения события не гарантирует предотвращение действий обозревателя по умолчанию для элементов, стоящих на пути распространения события (то есть распространение события для действий по умолчанию не отменяется). В данном случае таким элементом является ссылка, которая при прохождении через неё события click (на стадии всплытия) вызывает загрузку в новом окне (или вкладке) документа "home.htm". Отменить действия по умолчанию можно с помощью метода preventDefault():

```
function imgHandle(e){ console.log('img_capture:' +
e.target.tagName); e.stopImmediatePropagation();
e.preventDefault();}
```

1.4.4. Пример обработки событий

Предположим, что требуется реализовать «перетаскивание» мышью некоторого прямоугольника в окне обозревателя. Пусть прямоугольник – это div элемент размером 100 на 50 пикселей, который позиционируется абсолютно, закрашен желтым цветом, имеет красную рамку и курсор в виде перекрестия стрелок:

```
<style>
 #mydiv {
  position:absolute;
  background-color:yellow;
  border:1px solid red;
  width:100px;height:50px;
  cursor:move;
}
</style>
<div id='mydiv'></div>
```

При нажатии левой кнопки мыши (mousedown) в любом месте прямоугольника включаем обработку события (mousemove) перемещения курсора мыши по документу для перетаскивания прямоугольника вслед за курсором.

При отпускании левой кнопки мыши (mouseup) отменяем регистрацию этого обработчика, прекращая тем самым процесс перетаскивания. Для реализации представленного алгоритма, очевидно, надо знать координаты курсора мыши. Оказывается, эти координаты передаются посредством свойств объекта события pageX и pageY. При нажатии кнопки мыши также потребуется фиксировать положение курсора относительно левого верхнего угла прямоугольника в глобальных переменных dx, dy. Эти значения потребуются для расчёта нового положения левого верхнего угла прямоугольника по новым значениям координат курсора мыши:

```
var mydiv = document.getElementById('mydiv'), dx, dy;
mydiv.addEventListener('mousedown', onMouseDownHandler,
false);
mydiv.addEventListener('mouseup', onMouseUpHandler,
false);
function onMouseDownHandler(e) {
 var box = mydiv.getBoundingClientRect();
 dx = e.pageX - box.left, dy = e.pageY - box.top;
 document.addEventListener('mousemove', onMouseMoveHan-
dler, false);
}
function onMouseUpHandler() {
 document.removeEventListener('mousemove', onMouseMove-
Handler,
      false);
}
function onMouseMoveHandler(e) {
 var x = e.pageX, y = e.pageY;
 var x1 = x - dx, y1 = y - dy;
 mydiv.style.left = x1 + 'px';
 mydiv.style.top = y1 + 'px';
}
```

1.4.5. О загрузке содержимого документа и кода JavaScript

Пример, приведённый в предыдущем пункте, может оказаться не работоспособным, если на момент выполнения кода JavaScript элемент div ещё не был загружен обозревателем. В этом случае в переменной mydiv окажется значение null и регистрация событий будет невозможна. Таким образом, важно понимать порядок загрузки содержимого документа, и прежде всего сценариев.

Когда обозреватель встречает тэг script, то он выполняет код, указанный в этом тэге (предварительно загрузив его из внешнего файла, если такой указан в атрибуте src). При этом загрузка всех последующих элементов документа блокируется. Понятно, что если сценарий использует незагруженную часть документа,

то он будет ошибочным. Таков обычный порядок загрузки. Можно изменить этот порядок, если указать атрибут async или defer (или оба) в тэге script.

Атрибут async полезен при загрузке сценария из внешнего файла: обозреватель не будет ждать завершения загрузки и продолжит разбор оставшейся части документа. К выполнению сценария обозреватель вернётся сразу после загрузки. Таким образом, загрузка кода JavaScript происходит асинхронно (параллельно с разбором документа).

Атрибут defer позволяет не только асинхронно загружать сценарий из внешнего файла, но и отложить выполнение сценария до момента, когда документ будет загружен, проанализирован и станет готов к выполнению операций.

Если тэг script имеет оба атрибута, браузер отдаст предпочтение атрибуту async и проигнорирует атрибут defer.

Согласно сказанному, пример, приведенный выше, будет работоспособным в следующих случаях:

- элемент div идёт перед сценарием;
- сценарий (тэг script) записан прежде div, но это отложенный сценарий, то есть для него указан атрибут defer.

Более надёжный и популярный способ обезопасить клиентский JavaScript от обращений к незагруженным частям документа – это отслеживание событий загрузки. Во-первых, это уже упоминавшееся выше событие load окна window. Это событие возбуждается только после того, как документ и все его изображения будут полностью загружены. Однако обычно сценарии можно запускать сразу после синтаксического анализа документа, до того, как будут загружены изображения. Можно существенно сократить время запуска приложения, если начинать выполнение сценариев по событию DOMContentLoaded объекта document. Это событие возбуждается, как только документ будет загружен, разобран синтаксическим анализатором и будут выполнены все отложенные сценарии. К этому моменту изображения и сценарии с атрибутом async могут продолжать загружаться, но сам документ уже будет готов к выполнению операций. Таким образом, точкой входа в большинстве приложений является функция onLoad, которая регистрируется в качестве обработчика указанного события.

Рекомендуется самостоятельно реализовать представленный выше пример так, чтобы стартовой точкой его работы являлась функция onLoad в качестве обработчика события DOMContentLoaded.

2. Введение в jQuery

Библиотеку jQuery можно загрузить с сайта http://jquery.com/download/, скопировать её в удобное место, а затем подключить с помощью тэга script:

```
<script src="jquery-3.2.1.min.js"></script>
```

Можно подключить её также непосредственно с сайта:

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

Версия 3.2.1 – это самая последняя на момент написания пособия, но если требуется конкретная версия, то для её подключения достаточно изменить цифры номера версии. Слово «min» в имени файла означает, что файл минимизирован по размеру – из файла удалены лишние символы: комментарии, переводы строк и т. д. Если требуется «читаемая» версия библиотеки, то достаточно убрать в имени слово «min», в результате получим jquery-3.2.1.js. Этот файл следует использовать в процессе отладки приложения.

В данном разделе излагаются лишь основы работы с библиотекой, которых, тем не менее, достаточно для успешного старта frontend разработки профессиональных приложений. Более полную информацию можно найти в [1] или других источниках в Глобальной сети.

2.1. Функция $()

Имя функция $ является более коротким и удобным псевдонимом имени jQuery. Эта функция ведет себя по-разному в зависимости от переданных ей параметров. Существуют четыре различных варианта её поведения.

Если в качестве параметра передать функцию, то эта функция будет вызвана после загрузки документа. Обычно работу с библиотекой начинают именно с такого варианта обращения:

```
$(function() {
  // в теле этой функции начинаем работать
  //с документом
}
```

Если в качестве параметра передать CSS-селектор, то результатом работы функции будет объект jQuery.

Объект jQuery не следует путать с функцией jQuery.

Это наиболее востребованный вариант вызова, который позволяет выбрать требуемые элементы документа. Для его успешного использования требуется знать CSS-селекторы (подробное обсуждение различных вариантов се-

лекторов можно найти в любом источнике, посвященном стилям, в том числе в [1]). Объект jQuery будет содержать отобранные элементы документа, соответствующие селектору. Этот объект подобен массиву. Например, выберем все элементы div класса divclass и определим их количество:

```
$("div.divclass").length;
```

Первый отобранный элемент:

```
$("div.divclass")[0];
```

Вместо свойства length можно использовать метод size(), а вместо квадратных скобок – метод get().

Другие варианты отбора:

```
$("div, #idStr"); // Все элементы <div> или с id="idStr"
$('p[lang="ru"]'); // Все абзацы вида <p lang="ru">
$('*[name="x"]'); // Любой элемент с атрибутом name="x"
$("#log span"); // Любой элемент span, являющийся потомком элемента с
                // id="log"
$("#log>span"); // Любой элемент <span>, дочерний по отношению к
                // элементу с id="log"
$("body>h1:first-child"); // Первый элемент <h1>, дочерний по
                         // отношению к body
var bscripts = $('script', document.body); // все сценарии в
                                           // теле документа
bscripts.selector; // 'script'
bscripts.context; // document.body
bscripts.jquery; //3.2.1
```

Для объекта с отобранными элементами предусмотрено множество полезных методов, о которых речь пойдёт далее. Вот таким образом, например, можно выбрать первых потомков:

```
$('#w1').children().first();
```

Если в качестве параметра передать строку с тэгом, то будет создан объект jQuery, содержащий соответствующий тэгу элемент. Этот элемент затем может быть вставлен в документ. В качестве второго параметра обычно передается объект, определяющий атрибуты и прочую специфику создаваемого элемента, например:

```
var img = $("<div/>", { // Создать элемент div
  lang: 'ru', // с таким атрибутом,
  css: { backgroundColor:'yellow' }, // и желтым фоном
});
```

Если передать элемент (или массив элементов) DOM, то функция вернёт объект jQuery, содержащий этот (или эти) элементы. Делается это для того, чтобы можно было пользоваться методами объекта jQuery. Эти методы более удобны, чем низкоуровневые оригинальные методы объектной модели DOM и являются кросс-браузерными.

2.2. Работа с элементами

Над элементами документа, представленными в виде объекта jQuery, можно выполнять различные действия: манипулировать атрибутами, менять стили, классы, содержимое, изменять геометрию, назначать обработчики событий.

Библиотека устроена таким образом, что методы объекта jQuery, предназначенные для изменения элементов (их содержимого, классов, стилей и т. д.), могут быть использованы и для получения соответствующей информации, то есть работают как по записи, так и по чтению. Если аргумент с новым значением указан, то это записывающий метод, если нет, то метод для чтения. Как правило, читается текущее значение только первого отобранного элемента, а записывающие методы меняют все отобранные элементы.

В большинстве случаев для записывающих методов в качестве нового значения можно передать функцию. Эта функция получит на входе два параметра: номер отобранного элемента и сам элемент (this также будет ссылаться на текущий элемент). Результат работы функции будет новым значением, например, текстового содержимого элемента для метода text() или HTML-разметки для метода html(). Возврат логического значения false может прервать обработку отобранных элементов.

Результат работы методов – это объект jQuery, который можно использовать вновь для организации цепочек вызовов (конвейерная обработка). Такой подход весьма удобен и эффективен для сокращения размера кода и его доступности для понимания.

2.2.1. Работа с атрибутами

Пример чтения атрибута:

```
var action = $("form").attr("action");
```

В этом примере, во-первых, выбираются все формы – это результат работы вызова функции jQuery с параметром "form": $("form"). Уточним, результа-

том на самом деле является объект jQuery, который будет содержать все формы документа. Далее, для этого результата, то есть объекта jQuery, вызывается метод attr("action"), который читает значение атрибута action **первой** выбранной формы.

Если же задать второй параметр, например, так:

```
$("form").attr("action", "post");
```

то атрибут action всех выбранных форм будет изменён на значение post.

В качестве параметра можно задать объект, названия свойств которого будут соответствовать названиям атрибутов, а значения – значениям атрибутов, например:

```
$("#banner").attr({src:"banner.gif", alt:"Реклама",
width:720, height:64});
```

Если требуется удалить атрибут, то это можно сделать с помощью метода removeAttr(), например:

```
$("a").removeAttr("target");
```

Все ссылки после этого будут загружаться в текущее окно.

2.2.2. Работа со стилями и классами

Вычисленное значение стиля определяется с помощью метода css(). Например, найдём размер шрифта первого элемента div:

```
var fs = $("div").css("font-size");
```

Значения стилей – строки. В данном случае переменная fs будет содержать размер шрифта в пикселах с указанием размерности в конце (например, "16px").

В следующем примере стилизуем все элементы div одинаковым образом:

```
$("div").css({backgroundColor: "black",
  textColor: "white",
  fontVariant: "small-caps",
  padding: "20px 2px",
  border: "solid black 1px"
});
```

Объект jQuery также позволяет манипулировать классами: addClass() – добавить класс, removeClass() – удалить класс, toggleClass() – добавить класс, если его нет, или удалить, если есть. Добавление или удаление класса эквивалентно добавлению или удалению стилевого оформления, связанного с этим классом. Примеры:

```
$("p").removeClass("important");
$("h2+p").addClass("hilite first");
$("h1").toggleClass("hilite");
```

Часто весьма полезным оказывается метод определения наличия класса hasClass(). Этот метод может работать только с одним именем класса и возвращает true, если хотя бы один из выбранных элементов обладает указанным классом:

```
$("p").hasClass("important"); // Является ли хотя бы
один из
                              // параграфов важным?
```

2.2.3. Манипулирование содержимым элементов

Метод text() без аргументов позволяет получить содержимое всех вложенных текстовых узлов отобранных элементов, метод html() без аргументов выдаст в виде HTML содержимое только первого отобранного элемента:

```
var title = $("head title").text() // Титул документа
var html = $("div").html() // HTML разметка первого
<div>
```

Если методу text() в качестве параметра передать текст, а методу html() соответственно передать HTML-разметку, то они изменят содержимое всех отобранных элементов. В качестве параметра также можно предать функцию, которая будет вызвана для каждого отобранного элемента, а её результат будет использован для замены или текста, или HTML-разметки этого элемента. Например:

```
$("h1").text( function(i, txt) { // Добавить в каждый
заголовок
  return (n+1) + ". " + txt // его порядковый номер
} );
```

Для чтения и установки значений элементов HTML-форм различных типов предназначен универсальный метод val(). Продемонстрируем работу с этим методом на примере. Рассмотрим следующий HTML-код, в котором представлены различные варианты полей ввода:

```
<input type="text" id="name">
<br/>
<select id="engine">
 <option>Карбюратор</option>
 <option selected>Инжектор</option>
 <option>Дизель</option>
</select>
<br/>
```

```
<select multiple>
 <option selected>Тонировка</option>
 <option>Антикор</option>
 <option selected>Полировка</option>
</select>
<br/>
<input type="radio" name="marka" value="r1">Lada
<input type="radio" name="marka" value="r2"
checked>Volkswagen
<input type="radio" name="marka" value="r3">BMW
<br/>
<input type="checkbox">ABS<br/>
<input type="checkbox" checked>EBD<br/>
<input type="checkbox" checked>ESC<br/>
```

Сначала установим значения некоторых полей:

```
$("#name").val("Введите Ваше имя");
 var opt = ["abs", "ebd", "esc"];
 $("input:checkbox").val( function(i) {
    return opt[i];
 } );
```

А затем прочитаем значения всех представленных полей и выведем их в консоли:

```
var v1 = $("#name").val(),
    v2 = $("#engine").val(),
    v3 = $("select[multiple]").val(),
    v4 = $("input:radio[name=marka]:checked").val(),
    v5 = $("input:checkbox:checked").val();
console.log(v1);
console.log(v2);
console.log(v3);
console.log(v4);
console.log(v5);
```

В результате в web-консоли получим:

```
Введите Ваше имя
Инжектор
Array [ "Тонировка", "Полировка" ]
r2
EBD
```

Метод val() прочитал значение только первого из двух отмеченных (с галочкой) полей типа checkbox. В качестве упражнения рекомендуется самостоятельно прочитать значение второго выделенного чекбокса.

2.2.4. Манипулирование размерами и положением элементов

Ранее определено, что любой элемент имеет прямоугольную область с содержимым, которая окружена отступами (padding), рамкой (border) и полями (margin). Размеры этих областей чаще всего устанавливают для блочных элементов (, <p>, <table> и др.).

В jQuery предусмотрены три варианта методов для определения ширины и высоты первого отобранного элемента. Это методы width() и height() для определения размеров содержимого. Если требуется учесть отступы, то следует использовать методы innerWidth() и innerHeight(), а с учётом бордюров работают методы outerWidth() и outerHeight().

Всем этим методам можно передать целочисленное значение – размер в пикселах или функцию, которая на входе получит номер отобранного элемента, а результат её работы будет использоваться в качестве нового значения ширины или высоты. В следующем примере установим ширину элементов div пропорционально их номеру:

```
divs.outerWidth( function(i) {
    return (i+1)*50;
 } );
```

Отметим, что методы не меняют размеров объектов window и document. Для этих объектов перечисленные методы работают только по чтению.

Для определения положения первого отобранного элемента относительно начала документа предусмотрен метод offset(). Метод вернёт объект со свойствами top и left. Этот же метод с параметром (параметром также должен быть объект со свойствами top и left) изменит положение всех отобранных элементов. Следующий фрагмент кода переместит элемент с идентификатором TO_MOVE на десять пикселов по вертикали:

```
var e = $("#TO_MOVE");
var pos = e.offset();
pos.top += 10;
e.offset(pos);
```

 Здесь также можно использовать метод position(), но только для определения координат первого отобранного элемента относительно его родителя (родитель определяется с помощью метода offsetParent()).

Наконец, чтобы выполнить прокрутку содержимого элемента, следует использовать методы scrollTop() и scrollLeft(). Конечно, прежде всего это актуально для прокрутки документа в окне обозревателя. Например, выполним скроллинг документа на одну страницу вниз:


```
var w = $(window);
var pagesize = w.height();
var current = w.scrollTop(); // Текущая позиция
w.scrollTop(current + pagesize);
```

2.2.5. Изменение структуры документа

До сих пор рассматривались методы, которые меняли или стилевое оформление, или содержимое элементов документа, отобранных в объекте jQuery. Эти методы не меняли структуру документа: не вставляли новые элементы, не перемещали и не удаляли уже существующие элементы. Правда, можно создать новые элементы неявно, например, изменив внутреннюю разметку отобранных элементов с помощью уже рассмотренных методов html() или text().

Рассмотрим методы, явно ориентированные на изменение структуры документа.

Добавление нового содержимого в конец или начало содержимого каждого отобранного элемента осуществляется методами append() и prepend():

```
$("#log").append("<br/>"+message); // Добавить содержи-
мое
                                   // в конец элем.
#log
$("h1").prepend("§"); // Добавить символ параграфа
                      // в начало каждого элемента <h1>
```

Вставка элемента перед каждым отобранным элементом или после каждого отобранного элемента выполняется методами before() и after():

```
$("h1").before("<hr/>"); // Вставить линию перед каждым
элем. <h1>
$("h1").after("<hr/>"); // Вставить линию после каждого
элем. <h1>
```

Замена отобранных элементов на новый элемент выполняется с помощью метода replaceWith():

```
$("hr").replaceWith("<br/>"); // Заменить <hr/> на
<br/>
$("h2").replaceWith(function(i, html) { // Заменить
<h2> на <h1>,
  return "<h1>" +html + "</h1>"); // сохранив содержи-
мое
});
```

В приведённых примерах новые элементы создаются на лету из строк, содержащих HTML-разметку, переданных в качестве аргумента. Если же в качестве аргумента передать существующий элемент, то он будет перемещён на новое место и клонирован нужное количество раз:

```
var ln = $('#LINE');
ln.css({
  borderTop: '1px solid black',
  width: '90%', height: '3px'
});
$('h1').after(ln[0]);
```

В этом примере элемент с идентификатором LINE (пусть это будет div) будет перемещён на место после первого заголовка h1, а его клоны будут созданы автоматически и вставлены после остальных заголовков h1 (если они есть).

Представленные методы вставляют то, что передано в качестве параметра, а место вставки определяется отобранными элементами. То есть работают по схеме:

```
$(относительно этих элементов).вставить(содержимое)
```

Иногда бывает и наоборот, удобнее место вставки определять с помощью параметра, а в качестве вставляемого содержимого использовать отобранные элементы:

```
$(содержимое).втавить(относительно этих элементов)
```

Такие действия можно выполнить с помощью альтернативных методов appendTo(), prependTo(), insertBefore(), insertAfter(). То же касается и замены – replaceAll(). Пример представлен ниже.

Клонировать элементы можно с помощью метода clone(). Этот метод создаёт копии всех отобранных элементов вместе с их потомками. Копии будут возвращены в виде объекта jQuery и могут быть вставлены в документ, например:

```
// Добавить div в конец документа
$(document.body).append(
  "<div id='linklist'><h1>Ссылки</h1></div>"
);
// Клонировать ссылки и вставить их в div
$("a").clone().appendTo("#linklist");
// Вставить элементы <br/> после каждой ссылки
$("#linklist > a").after("<br/>");
```

Отметим, что копии создаются без обработчиков событий (которые рассматривются далее). Если требуется создать копии вместе с обработчиками, то достаточно передать true методу clone().

Для удаления отобранных элементов предусмотрен метод remove(). Если требуется сохранить обработчики событий, связанные с этими элементами, то следует использовать метод detach().

2.3. Обработка событий

В библиотеке jQuery имеется довольно большой набор методов, предназначенных для регистрации обработчиков событий и их удаления, а также реализованы универсальные (кросс-браузерные) подходы как к передаче данных, специфических для различных типов событий, так и к их обработке, соответствующие требованиям консорциума W3C. В данном разделе рассмотрим лишь основные приёмы работы с событиями, характерными для jQuery, не претендуя на полноту изложения. Впрочем, в большинстве случаев этого достаточно.

2.3.1. Регистрация и удаление обработчиков событий

Объект jQuery предоставляет универсальный метод bind() для регистрации обработчиков событий, аналогичный addEventListener(). Как и следовало ожидать, этот метод «привяжет» обработчик к каждому отобранному элементу. Первым параметром должен быть передан тип события, вторым – обработчик. Примеры:

```
$('input:button').bind('click', btnHandler);
$('#divId').bind('mouseenter mouseleave', hoverHan-
dler);
```

В последней строке приведён пример регистрации обработчика для двух типов событий.

Другой метод one() аналогичен методу bind() с единственным отличием, что зарегистрированный с его помощью обработчик будет вызван только один раз, а затем автоматически удалён.

Оба метода, в отличие от addEventListener(), не позволяют указать фазу перехвата события. Дело в том, что в jQuery используется только фаза всплытия события.

Возможен также и иной способ регистрации: для каждого типа события объект jQuery предоставляет одноимённый метод регистрации обработчика, например:

```
// Щелчок на любом элементе <div> окрашивает его рамку
$("div").click(
  function() { $(this).css("border-color", "red");
});
```

Следует также вспомнить о функции $(), о том варианте её вызова, когда первым параметром передаётся разметка HTML. В этом случае, в качестве второго параметра может быть передан объект, свойства которого можно использовать для регистрации обработчиков событий, например:

```
var img = $("<img/>", {
  src: "home.png",
  mouseenter: function() {
    $(this).css("border", "2px solid blue");
  },
  mouseleave: function() {
    $(this).css("border", "0px ");
  }
});
img.appendTo('body');
```

В данном примере при наведении курсора мыши на рисунок он выделяется синим бордюром. Заметим, что в обработчике this указывает на элемент, для которого этот обработчик зарегистрирован.

Для удаления обработчиков служит метод unbind(), который является довольно мощным, поскольку позволяет охватить множество элементов и событий:

```
// Удалить все обработчики для всех элементов
$('*').unbind();
// Удалить все обработчики событий mouseover и mouseout
// для всех ссылок
$('a').unbind("mouseover mouseout");
// Можно точно указать обработчик,
// который требуется удалить
$('input:button').unbind('click', btnHandler);
```

2.3.2. Универсальный объект Event

Библиотека jQuery использует собственный объект Event, который передается обработчикам событий, зарегистрированным с помощью описанных методов, в качестве первого параметра. Посредством свойств этого объекта обработчик получает доступ к данным, фиксирующим специфику события (тип события, положение курсора мыши и т. д., напомним также, что this в обработчике ссылается на текущий элемент, до которого всплыло событие).

Этот объект не требует учёта специфики обозревателя, а ряд свойств оригинального объекта Event заимствуется.

Рассмотрим наиболее важные из них:

`type`

Содержит тип события.

`target`, `currentTarget`, `relatedTarget`

Первое свойство target ссылается на источник события – элемент, на котором событие возникло. Свойство currentTarget ссылается на элемент, в котором был зарегистрирован обработчик события. Значение этого свойства всегда должно совпадать со значением this. Таким образом, если значения свойств currentTarget и target не совпадают, то это признак всплывшего события. Свойство relatedTarget полезно для обработки событий перехода подобных mouseover и mouseout. Так, для mouseover свойство relatedTarget будет ссылаться на элемент, который покинул указатель мыши при перемещении на элемент target, а для mouseout – на элемент, на который указатель мыши выйдет после target.

`clientX, clientY`

Координаты указателя мыши в видимой области (относительно левого верхнего угла окна обозревателя).

`pageX, pageY`

Координаты указателя мыши относительно начала документа (то есть с учётом скроллинга).

`metaKey`

Это логическое значение, полезное для обработчиков событий мыши, позволяющее определить, была ли нажата клавиша ctrl или Command в MacOS.

`which`

Для событий клавиатуры keydown, keyup – это скан-код клавиши. Скан-код — это порядковый номер клавиши, который не зависит от раскладки и выбранного регистра клавиатуры.

Для события keypress – это код символа. Событие возникает только при нажатии алфавитно-цифровой клавиши. Нажатия управляющих клавиш (ctrl, alt и т. д.) игнорируются.

Для событий мыши это: 0 – означает, что никакая кнопка не была нажата, либо 1 – была нажата левая кнопка, либо 2 – средняя кнопка, либо 3 – правая кнопка. Отметим, что в некоторых браузерах нажатие правой кнопки мыши не генерирует события.

`originalEvent`

Ссылка на объект Event, созданный браузером.

Для отмены всплытия события, а также действий по умолчанию предусмотрены методы со стандартными именами:

```
preventDefault(), stopPropagation(), stopImmediate-
Propagation()
```

Заметим, что одновременно предотвратить всплытие и действие по умолчанию в обработчиках jQuery можно проще – вернуть значение false.

2.3.3. Искусственная генерация событий. Собственные события

Система управления событиями в библиотеке jQuery позволяет определять собственные типы событий (использовать любую строку в этом качестве), а метод bind() позволяет регистрировать обработчики таких нестандартных событий.

При этом, естественно, возникает вопрос: а в каких ситуациях эти события возникают? Оказывается, такие ситуации отслеживаются самим программистом, а метод trigger() позволяет искусственно сгенерировать такие события в нужном месте. Этот метод работает для любых типов событий: для искусственных, придуманных программистом и для стандартных событий в том числе.

Метод trigger() – это метод объекта jQuery. Если этот метод вызвать без аргументов, то для всех отобранных элементов будут вызваны все зарегистрированные обработчики всех типов. Если же требуется вызвать обработчики только одного типа, то этот тип следует указать в качестве первого аргумента:

```
$("button").trigger(); // вызвать все обработчики всех
кнопок
// вызвать обработчики события типа click
$("button").trigger("click");
```

Иногда требуется вызвать обработчики событий (все или определённого типа) без отбора элементов. В таких случаях можно воспользоваться методом trigger() следующим образом:

```
jQuery.event.trigger();
jQuery.event.trigger('click');
```

Заметим, что здесь jQuery – функция, а не объект.

Тот же результат можно получить обычным способом, отобрав все элементы в объекте jQuery (селектор * позволяет это сделать) и вызвав метод trigger():

```
$('*').trigger();
$('*').trigger('click');
```

Однако последнее решение менее эффективно.

Такая разновидность косвенного вызова обработчиков нестандартных событий может оказаться иногда весьма полезной:

```
// Когда пользователь щелкнет на кнопке
// с идентификатором assigned (назначен)
// будут отработаны все обработчики этого события,
// которые выполнят все предусмотренные
// назначением действия
$("#assigned").click(function() {
  // Отправить широковещательное событие
  $.event.trigger("assigned");
});
```

2.4. Работа с AJAX

Благодаря AJAX (Asynchronous JavaScript and XML – асинхронный JavaScript и XML) в коде JavaScript появилась возможность непосредственно работать с протоколом http. Этот высокоуровневый протокол используется программами (они называются клиентами) для обмена данными с web-сервисами (это программы, которые обслуживают клиентов).

До появления AJAX язык JavaScript не позволял непосредственно работать с web-сервисами, получать требуемые данные и использовать их, например для динамического изменения объектной модели документа или в иных целях. Сделать это можно было только опосредованно, через документ, который предварительно загружался во встроенный фрейм (iframe). В JavaScript можно было только «попросить» браузер выполнить эту загрузку, а непосредственная рабо-

та с web-серверами была невозможна. С помощью AJAX это можно сделать и, более того, получать данные в «нестандартном» для браузеров виде, например в формате json или xml, или предусмотреть собственный оригинальный формат.

2.4.1. О протоколе http

Протокол http определяет правила обмена данными между двумя программами через сетевое соединение. Одна из программ – это клиент, а другая – сервер. Сервер обслуживает клиента по его запросу: обычно отдаёт запрошенные клиентом данные (бывает и наоборот). Компьютер, на котором выполняется серверная программа, идентифицируется своим доменным именем или ip-адресом, а сама серверная программа на этом компьютере идентифицируется номером, называемым портом.

Таким образом, протокол предусматривает две фазы обмена данными: запрос и ответ, которые часто называют циклом «запрос – ответ». Запрос формируется клиентом, а ответ – сервером. Структура запросов и ответов одинакова – заголовок и тело. Собственно, сами данные передаются в теле, а вспомогательная заголовочная часть содержит описательную информацию по поводу того, что передаётся в теле. Заголовочная часть состоит из строк, то есть последовательностей символов в кодировке ASCII, которые отделены символами «возврата каретки» с кодом 13 – Cary Return (CR) и «перевода строки» с кодом 10 – Line Feed (LF). Тело отделено от заголовка пустой строкой (то есть последовательность символов CR и LF повторяется дважды).

Первая строка заголовочной части имеет особое назначение. В запросах это строка запроса (request line), а в ответах это строка статуса (status line). Остальные строки заголовочной части и запросов и ответов называются полями. Все поля — это пары «имя и значения». Имя поля отделяется от значений двоеточием, за которым обычно идёт пробел. Имена полей нечувствительны к регистру, то есть имя CONTENT-TYPE и Content-Type – это одно и то же. Если значений много, то они отделяются запятыми, впрочем, не запрещено эти значения указать в нескольких полях с одним и тем же именем. Порядок полей произволен.

Рассмотрим пример запроса:

```
GET /callibri_close.png HTTP/1.1
Host: cdn.callibri.ru
User-Agent: Mozilla/5.0
Accept: text/html,application/xml;q=0.9,*/*;q=0.8
```

В этом примере тело отсутствует. В строке запроса (первая строка) указывается метод запроса GET, затем через пробел запрашиваемый ресурс

(/callibri_close.png – файл в корневом каталоге сервера) и через пробел версия протокола (HTTP/1.1). В полях (последующих строках) указывается:

- в поле с именем Host – доменное имя компьютера (хоста), к которому адресован запрос;
- в поле User-Agent – название клиента, который сформировал запрос;
- в поле Accept – предпочтительные медиатипы данных, которые принимает клиент и которые перечислены через запятую как отдельные значения этого поля; степень предпочтения указывается в параметре q (*/* означает любой тип).

Конечно, это не весь список полей, которые могут быть использованы в запросах, – их значительно больше. Ответ на данный запрос может быть следующим:

```
HTTP/1.1 200 OK
Content-Length: 213
Content-Type: image/png
Last-Modified: Tue, 06 Sep 2016 09:31:02 GMT
другие поля, которые здесь не приводятся

здесь расположено тело ответа – рисунок в формате png
(содержимое запрошенного файла callibri_close.png)
```

Первая строка – строка статуса. Эта строка содержит название протокола, его версию, затем через пробелы код статуса и расшифровку этого статуса. В данном случае код 200 свидетельствует об успешном выполнении запроса. Другие часто встречающиеся коды: 404 Not Found (запрашиваемый файл не найден), 301 Moved Permanently (запрашиваемый файл перемещён постоянно в другое место, которое обычно указывается в поле location), 500 Internal Server Error (внутренняя ошибка сервера).

Назначение полей ответа:

- в поле Content-Length указывается длина тела ответа;
- в поле Content-Type – медиатип передаваемых данных;
- в поле Last-Modified – дата последней модификации передаваемых данных.

Кроме метода GET, существуют и другие методы, среди которых следует уделить внимание методу POST. Если метод GET предполагает отсутствие тела запроса, то метод POST, наоборот, предполагает его наличие. Что может содержать это тело, передаваемое серверу? Это может быть любой медиатип, который должен быть определён в поле запроса Content-Type. Однако чаще всего это параметры запроса, представленные в формате: имя параметра (или поля), знак равенства, значение параметра, например:

first_name=%CF%B8%F2%F0&last_name=%CF%E5%F2%F0%EE%E2

Здесь передаются два параметра с именами first_name (имя) и last_name (фамилия). Значения этих параметров: «Пётр» и «Петров» представлены шестнадцатеричными кодами букв (например, буква «П» в кодировке Windows-1251 имеет код 0xCF). Весь запрос мог бы выглядеть так:

```
POST /1.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 …
Accept: text/html, application/xml;q=0.9,*/*;q=0.8
Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 52
Connection: keep-alive

first_name=%CF%B8%F2%F0&last_name=%CF%E5%F2%F0%EE%E2
```

Отметим, что метод GET также позволяет передавать параметры в строке запроса:

```
GET /1.php?first_name=%CF%B8%F2%F0&last_name=… HTTP/1.1
```

2.4.2. JSON

В теле ответа сервера или запроса клиента, как уже рассматривалось, могут передаваться любые медиаданные: текст, изображение, видео и т. д. Однако при работе с AJAX наиболее популярны текстовые данные в форматах xml и json. Формат json не требует каких-либо дополнительных знаний, кроме знания языка JavaScript, и этим, по всей видимости, объясняется его популярность среди программистов. Фактически json – это литеральный способ представления данных, принятый в коде JavaScript.

При работе с json приходится решать две задачи. Первая задача состоит в следующем: есть данные, представленные в программе некоторой переменной (как правило, это объект), требуется представить эти данные в виде текста json, который затем может быть передан по сети от клиента к серверу или от сервера к клиенту:

```
var obj = {
 arr: [1, "два", 3],
 name: "Пётр",
 date: new Date(),
 logical: true
};
var json_text = JSON.stringify(obj);
console.log(json_text);
```

В результате в консоль браузера будет выведена строка:

```
{"arr":[1,"два",3],"name":"Пётр","date":"2017-06-
05T10:31:21.853Z","logical":true}
```

Обратная задача – есть текст json, требуется преобразовать этот текст в объект, который затем может быть использован в коде JavaScript наравне с другими объектами:

```
var obj = JSON.parse(json_str);
```

Наконец, отметим, что и в других языках предусмотрены функции, подобные JSON.stringify(), JSON.parse(), для работы json форматом (например, в php).

2.4.3. Пример работы с AJAX

Библиотека jQuery располагает богатым арсеналом методов работы с AJAX. Среди них можно выделить низкоуровневые методы, которые хороши тем, что позволяют в полной мере управлять обменом данными между клиентским JavaScript и web-сервером. Наиболее мощным низкоуровневым методом является метод ajax(). Подчеркнём, что это не метод объекта jQuery, а функция библиотеки jQuery, поэтому вызывать её следует так: jQuery.ajax() или $.ajax().

Все остальные методы работают опосредованно, через эту функцию. Ограничимся перечислением этих методов: $.get(), $.post(), $.getScript(), $.getJSON(), наконец, метод load() объекта jQuery.

Метод $.ajax() принимает в качестве параметра объект, свойства которого в деталях описывают выполнение Ajax-запроса, например:

```
jQuery.ajax({
  method: "GET", // или POST
  url: url, // откуда забрать данные
  data: params, // свойства этого объекта
              // определяют параметры запроса
  dataType: "text", // тип данных
  success: onSuccess // эта функция будет вызвана
               // при успешном завершении запроса
  error: onError
});
```

Здесь использованы наиболее полезные параметры из довольно объемного списка. Объект params определяет специфику запроса – свойства этого объекта определяют названия и значения параметров. В случае метода GET эти параметры в закодированном виде будут добавлены в url, а в случае метода POST – в тело http-запроса (см. описание http-протокола выше). В качестве типа данных (dataType) можно использовать строки: text, html, xml, json, script. Если указать script, то есть загружаемые данные содержат код JavaScript, то этот код будет выполнен, а затем (при успешном завершении) будет вызвана функция onSuccess. Для типа json будет сделана попытка преобразования данных в объект, который будет передан функции обратного вызова onSuccess. Для xml будет передан XMLDocument – объект, который представляет иерархическую структуру переданных xml-данных. Для text и html никакой дополнительной обработки полученных данных не производится.

Функция onSuccess будет вызвана только в случае успешного выполнения ajax-запроса, в противном случае будет вызвана функция onError. Первый параметр этой функции будет содержать объект XMLHttpRequest. Этот оригинальный объект, который используется в JavaScript для выполнения Ajax-запросов без jQuery, здесь не обсуждается, поскольку в большинстве случаев для определения причины ошибки достаточно второго параметра. Если второй параметр содержит строку error, то это ошибка http-протокола, если строку timeout, то превышен предел времени ожидания ответа от сервера, если – parsererror, то ошибка разбора переданных данных (в формате json или xml).

Отметим, что ajax-запросы допускаются только в тот же самый домен, из которого был загружен исходный документ. jQuery поддерживает технологию обхода этого ограничения для загрузки json-содержимого.

Эта технология работает следующим образом. В документ вставляется дополнительный элемент script, его атрибуту src присваивается url требуемого json-содержимого, которое в результате загружается и выполняется как код JavaScript (ещё раз отметим, что json – это литеральный способ представления объектов, принятый в JavaScript). Если сервер в своём ответе дополнительно возьмёт в скобки передаваемое json-содержимое, предварив всю эту конструкцию некоторым идентификатором, то это, очевидно, будет трактоваться как вызов функции. В клиентском коде остаётся теперь только определить функцию с тем же именем, которая и будет вызвана после успешной загрузки и выполнения описанного выше кода json. Поскольку это не совсем json код (он окружён скобками), то подобные конструкции часто называют jsonp.

Например, сайт www.flickr.com позволяет получить ссылки на фото (закаченные туда зарегистрированными пользователями) в формате json. Для этого к нему надо обратиться следующим образом:

https://www.flickr.com/services/feeds/photos_public.gne?tags=rabit&format=json.

Параметры tags и format определяют тему и формат. В ответе будет содержаться вызов функции: имя функции – jsonFlickrFeed, а в скобках – требуемые json-данные. Таким образом, достаточно определить функцию с таким именем:

```
$(function(){
 $.ajax({
    method: "get",
    url:
"https://www.flickr.com/services/feeds/photos_public.gn
e",
    data: {
       tags: 'rabit',
       format: 'json'
    },
    dataType: "jsonp"
 });
});
function jsonFlickrFeed(data) {
 $.each(data.items, function(i, item) {
    var img = $("<img/>").attr("src",
item.media.m).attr("title",
item.title).appendTo('body');
 });
}
```

В теле функции jsonFlickrFeed загруженные данные используются для указания атрибутов src и title, вставляемых в тело документа рисунков.

3. Введение в node.js

Node.js – это интерпретатор кода на JavaScript, который часто называют платформой или средой выполнения в связи с огромным количеством дополнительных модулей под любые задачи. Это открытый проект, который появился в 2009 г. и с тех пор стремительно развивается, приобретая всё большую популярность среди разработчиков серверных приложений.

Node.js построен на основе очень быстрого движка Google Chrome V8, в котором отсутствуют традиционные промежуточные этапы создания байт-кода и последующей его интерпретации. Вместо генерирования байт-кода или использования интерпретатора выполняется непосредственная компиляция в машинный код.

Платформа node отличается:

- асинхронной и событийно-управляемой природой программирования (как и для клиентского JavaScript);

- доступными средствами и приёмами программирования неблокирующего ввода/вывода;
- эффективными и легковесными средствами программирования работы с Сетью;
- возможностью программирования высокоэффективных сетевых сервисов, прежде всего на основе http-протокола, способных обеспечивать большое количество клиентских подключений;
- обилием модулей для реализации разнообразных задач;
- эффективным и доступным менеджером управления модулями npm.

Таким образом, разработчики web-приложений могут отказаться от использования традиционных web-серверов и использовать единый язык (JavaScript), как для frontend, так и для backend разработки. Дополнительные сведения о платформе можно найти в [2, 3, 5–10].

3.1. Установка и проверка работоспособности

Скачать предлагаемую версию платформы можно на официальном сайте https://nodejs.org (из двух вариантов рекомендуется выбирать не последнюю, а рекомендуемую для большинства пользователей – Recommended For Most Users). Платформа node.js работает в любой операционной системе, и её установка обычно не вызывает проблем. Также будет установлен менеджер управления модулями npm.

Для проверки работоспособности запускаем node в окне терминала:

```
C:\>node
> var hello=["Hello", "World!"];
undefined
> hello.toString();
'Hello,World!'
>
(To exit, press ^C again or type .exit)
>

C:\>
```

Такой режим работы с интерпретатором называется REPL (Read Event Printed Loop). После приглашения (>) можно вводить код JavaScript, который будет выполнен немедленно (после нажатия на клавишу return). Для выхода из этого режима достаточно дважды ввести Ctrl+C.

Также можно выполнить код JavaScript, хранящийся в файле, например, с именем test.js. Пусть этот файл содержит следующий код (возведём в квадрат элементы числового массива):

```
var x = [1, 2, 3, 4, 5];
var y = x.map( function(ax) { return ax*ax; } )
console.log( y.join(',') );
```

Выполнить этот код можно следующим образом:

```
d:\>node test.js
1,4,9,16,25

d:\>
```

3.2. Модули

Практически любое приложение node.js использует модули. В большинстве случаев это готовый к употреблению код JavaScript, предназначенный для реализации самых разнообразных задач. Наиболее полезные и востребованные модули встроены в систему и их не нужно устанавливать. Все остальные (дополнительные модули) требуют установки с помощью менеджера модулей npm (node package manager), который, как уже было сказано, устанавливается совместно с node. Наконец, многократно используемый код приложения также целесообразно организовать в виде модулей собственной разработки. Менеджер npm позволяет не только устанавливать дополнительные модули, но и публиковать модули собственной разработки, которые таким образом становятся доступными всему сообществу разработчиков.

Модули могут быть представлены также исполнимыми файлами (результатами компиляции программ на языке C++). Менеджер пакетов npm является примером такого модуля. Таким образом, можно выделить следующие виды модулей:

- встроенные в node;
- дополнительные и модули собственной разработки, написанные на JavaScript (поэтому их называют js-модули);
- модули, написанные на C++ (node-модули).

К этому списку также следует добавить json-модули – это обычные json-файлы с данными.

3.2.1. Установка дополнительных модулей

Предположим, что потребовался модуль с именем mime для работы с mime-типами данных (Multipurpose Internet Mail Extensions). Указание типа данных предусмотрено протоколом http в поле с именем Content-Type. Например, для простого текста – это text/plain, для html – text/html, для картинки в формате jpeg – img/jpeg.

Для установки дополнительных модулей (в том числе и mime) потребуются права администратора. В Windows для этого следует войти с административной учётной записью, а в unix-подобных системах (в том числе mac os) перейти в административный режим, выполнив команду:

```
sudo -s
```

Команды следует выполнять в окне терминала (в Windows для запуска терминала достаточно набрать cmd в поле «выполнить»). Далее следует создать и перейти в папку, где предполагается вести разработку, например в папку NODETEST:

```
cd NODETEST
```


Выполнить установку модуля теперь можно с помощью команды:

```
npm install mime
```

3.2.2. Порядок поиска модулей

Подключение требуемого модуля осуществляется с помощью инструкции require(). В скобках достаточно указать имя встроенного модуля, например:

```
var http = require("http");
```

Чтобы подключить дополнительный модуль или модуль собственной разработки, можно указать абсолютный или относительный путь к файлу или папке с требуемым модулем (модуль может быть представлен как файлом, так и папкой). Однако это крайне неудобно, поскольку абсолютные пути могут быть слишком длинными, а относительные – потребуют корректировки в случае изменения расположения папки с разработкой. Другой вариант – указать только имя модуля, а node самостоятельно найдёт его в файловой системе компьютера. Как это происходит?

В node предусмотрен очень гибкий и простой механизм поиска модулей, который, конечно, предусматривает определённый порядок расположения модулей в файловой системе. Поиск начинается в текущей папке, где будет сделана попытка найти папку с именем node_modules. Если таковой нет, то поиск node_modules будет продолжен в родительских папках (если они существуют). Если в родительских папках модуля не оказалось, то поиск продолжается в папках, перечисленных в переменной окружения NODE_PATH.

При указании имени модуля-файла можно не указывать его расширение js. Если модуль представлен папкой, то в этой папке будет использован файл index.js (который может в своей работе использовать вспомогательные файлы в

этой папке). Имя стартового файла может быть изменено в конфигурационном файле модуля package.json в свойстве main.

3.2.3. Файл package.json

Конфигурационный файл package.json играет важную роль в процессе разработки модулей и приложений. Так, для загрузки и установки модуля mime (в приведённом выше примере) можно было бы создать файл package.json в папке NODETEST и указать в этом файле, в дескрипторе dependencies, все необходимые дополнительные модули, включая и модуль mime:

```
{
  "name": "nodetest",
  "version": "0.0.1",
  "description": "This is example",
  "dependencies": {
    "socket.io": "^0.9.6",
    "mime": "^1.2.7"
  }
}
```

Теперь для установки указанных модулей достаточно перейти в папку NODETEST и выполнить команду:

```
npm install
```

Модули будут установлены в каталог node_modules, который создается в текущем каталоге проекта (если он ещё не был создан). Отметим, что при таком способе установки (когда требуемые модули указываются в package.json) они всегда будут устанавливаться в подкаталоге node_modules текущего каталога, где расположен файл package.json. Иначе, если файл package.json отсутствует, а имя модуля явно указывается менеджеру пакетов (npm install mime), предварительно будет выполнен поиск каталога node_modules в текущем каталоге, а затем в родительских каталогах и установка будет выполнена в первый найденный каталог с таким именем. Если каталог node_modules не найден в текущем и в родительских каталогах, то только в этом случае он будет создан в текущем каталоге, где и будет выполнена его установка. В большинстве случаев такое поведение менеджера пакетов npm является желаемым и согласуется с алгоритмом поиска модуля, заданного инструкцией require().

В приведённом выше примере для модулей, указанных в директиве dependencies, также задана их версия. При этом символ ^ указывает на то, что требуется версия не старше заданной.

В package.json обязательно определяется имя (name) и версия (version) приложения (или модуля). Описание (description) – не обязательно. Кроме этого, предусмотрен ряд других описателей, например main, который упоминался выше. Подробное описание всех описателей можно найти в документации [6]. Версия модуля задаётся в формате major.minor.patch. Если, например, исправляется какой-то баг, то увеличивается на единицу значение patch, если добавляется новая функциональность, совместимая с предыдущей версией модуля, то увеличивается значение minor. Если вносятся существенные изменения, несовместимые с предыдущей версией модуля, то меняется значение major. Эти правила, конечно, условны и имеют значение при публикации модуля.

3.2.4. Полезные команды npm

Отметим ещё раз, что менеджер модулей (пакетов) npm сам является модулем и устанавливается вместе с node. Проверить установленную версию менеджера можно командой:

```
npm -v
```

А затем (если надо) установить последнюю версию:

```
npm –g install npm@latest
```

Здесь команда install используется с ключом –g, который позволяет установить модуль глобально. При глобальной установке модуль будет доступен в любом проекте, а для модулей типа node последние фактически становятся утилитами, которые можно вызывать в командной строке (в окне терминала).

После символа @ может быть указана версия устанавливаемого модуля или слово latest, если требуется самая последняя версия. Версию таким образом можно указывать не только для глобальной установки, но и для локальной. Отметим также, что если запрошенная версия уже установлена, то она, конечно, переустанавливаться не будет, а разные версии одного и того же модуля могут существовать совместно. Обновление модулей проекта, перечисленных в dependencies, может быть выполнено командой update:

```
npm update
```

В этой команде можно указать название обновляемого модуля, если требуется обновить только этот модуль.

Для удаления ненужного модуля служит команда uninstall:

```
npm uninstall <package>
```

Вместе с командами install и uninstall можно использовать ключ save, который заставляет менеджер npm корректировать файл package.json в соответствии со сделанными изменениями, например:

```
npm uninstall mime --save
```

Менеджер npm – мощный функциональный инструмент, который, конечно, не ограничивается рассмотренными командами. Менеджер позволяет регистрировать разработчиков, добавлять и удалять их модули, отслеживать версии, менять расположение хранилищ и т. д.

Получить перечень всех доступных команд позволяет команда help:

```
npm help
```

Получить детальную помощь по любой команде можно так:

```
npm -h <command>
```

Например, по команде search, которая предназначена для поиска модулей:

```
npm -h search
```

3.2.5. Пример разработки собственного модуля

Как уже отмечалось, используемый код целесообразно вынести в отдельный модуль (или модули). Предположим, что потребовался ряд вспомогательных функций для работы со строками. Например, функция, которая преобразует ряд зарезервированных в HTML символов (<, >, & и т. д.) в строки, которые часто называют html entities (htm-сущности) и которые предназначены для вставки и правильного отображения этих символов в HTML-документах. Эта функция может выглядеть следующим образом:

```
function htmlEntity(str) {
 var entity = {
    '&': '&amp;',
    '<': '&lt;',
    '>':' &gt;',
    "'": '&apos;',
    '"': '&quote;'
 }
 for(var s in entity) {
    var re = new RegExp(s, 'g');
    var str = str.replace(re, entity[s]);
 }
 return str;
}
```

Здесь объект entity представляет собой таблицу, в которой зарезервированным символам сопоставляются строки-сущности. Этот далеко не полный перечень можно легко дополнить. В последующем цикле каждый такой символ в заданной строке заменяется соответствующей сущностью. Для замены используется механизм регулярных выражений.

Для того чтобы этот код был доступен после его подключения командой require(), необходимо его экспортировать:

```
module.exports.htmlEntity = htmlEntity;
```

Здесь экспортируется функция htmlEntity. Любые объекты, которые должны быть доступны во внешнем (по отношению к модулю) коде, должны быть экспортированы таким образом. Вместо module.exports часто пользуются ссылкой exports на module.exports:

```
exports.htmlEntity = htmlEntity;
```

Есть также альтернативный путь сделать те или иные имена (переменные, функции) доступными во внешнем коде – объявить их глобальными (директива global), но такой подход используется редко.

Пусть представленный код размещён в файле с именем htmlEntity.js, тогда подключить и использовать этот код можно следующим образом:

```
var HE = require('./htmlEntity');
console.log(HE.htmlEntity("' | & | < | > | \""));
```

Кроме свойства exports, объект module имеет много других полезных свойств, например свойство module.parent, которое ссылается на подключивший его модуль. Это свойство позволяет определить – подключен модуль командой require() или выполняется самостоятельно. В нашем случае можно, например, продиагностировать работу функции htmlEntity() в случае непосредственного выполнения модуля:

```
if(!module.parent) console.log(htmlEntity("' | & | < |
> | \""));
```

Свойство module.children, наоборот, содержит ссылки на модули, подключенные командой require().

Наконец, следует иметь в виду, что модули кэшируются, то есть если команда подключения модуля выполняется второй раз, то это не вызовет повторную загрузку и выполнение кода модуля. При этом проверяется идентификатор модуля (module.id) – его абсолютный путь в файловой системе.

Напомним также, что модуль может быть представлен каталогом, содержащим все сопутствующие модулю файлы. При этом стартовым файлом в этом каталоге является файл index.js (если иное не указано в файле package.json в описателе main).

3.3. Http-сервер

Ядро node.js содержит эффективный и оптимизированный код для работы с http-протоколом, который доступен посредством подключения встроенного

модуля http. Эта часть ядра написана node-разработчиком Райаном Далом (Ryan Dahl) на языке C [3].

Модуль http предоставляет для кода JavaScript чрезвычайно гибкий низкоуровневый интерфейс для управления http-сервером.

3.3.1. Эхо-сервер

Реализуем в учебных целях своеобразный эхо-сервер, который в качестве ответа на любой запрос выдаёт клиенту содержимое заголовка (метод запроса, поля запроса) и содержимое тела запроса. Сервер может оказаться полезным для исследования работы протокола http. Начнём с подключения модуля и создания сервера:

```
var http = require('http');
var srvr = http.createServer(
 function(req, res) {
 res.end("Hello World!");
});
srvr.listen(3000);
console.log('Ok');
```


Метод createServer() принимает единственный параметр – функцию, предназначенную для обработки цикла «запрос – ответ». Эта функция соответственно получает два параметра – ссылки на объекты, один из которых представляет запрос, а другой – ответ. С помощью метода end() выполняются два действия: клиенту передаётся в теле ответа строка "Hello World!" и соединение с клиентом завершается. Для формирования ответа можно также воспользоваться методом write(), который не завершает соединение. Таким образом, в любом случае последним должен быть вызван метод end() (может быть и параметр). Заголовочная часть ответа формируется автоматически. Если требуется сформировать то или иное поле заголовка вручную, то для этого можно воспользоваться методом setHeader(), например:


```
res.setHeader('content-type', 'text/html');
```

Предусмотрены также методы getHeader(), removeHeader(), назначение которых очевидно. Также следует отметить, что манипуляции заголовочной частью ответа возможны только до первого обращения к методу write() или end().

Метод запроса можно получить посредством свойства req.method, а имена и содержимое полей – посредством свойства req.headers. Последнее ссылается на объект, свойства которого совпадают с именами полей, а значения свойств – на значения полей. Таким образом, следующий код позволяет вывести в ответе метод запроса и сформировать весь список полей запроса:

```
var http = require('http');
http.createServer(function(req, res) {
 res.statusCode = 200;
 res.setHeader('Content-Type', 'text/html');
 var
    method = req.method,
    headers = getHeaders(req);

 res.end(
    "<h1>" + method + "</h1>" +
    headers
 );
}).listen(3000);

function getHeaders(req) {
 var headers = [];
 for(var nm in req.headers)
    headers.push(nm + '=' + req.headers[nm]);
 return headers.join("<br/>\n");
}
```

В этом примере установка заголовка 'Content-Type' и установка кода статуса являются излишними, поскольку модуль http выполнил бы это автоматически. Однако эти действия потребовались бы, например, для сообщения клиенту о постоянном перемещении запрошенного ресурса: потребовалось бы в поле location указать новое размещение ресурса, а код статуса установить равным 301.

Для получения тела запроса требуется отслеживать загрузку данных от клиента через сетевое соединение. Для этого в классе запроса req предусмотрены два события «data» и «end». Первое событие возникает при получении очередной порции данных, а второе – при окончании передачи данных. Будем накапливать порции данных (chunk) следующим образом:

```
var chunk_arr = [];
 req.on('data', function(chunk) {
    chunk_arr.push(chunk);
 });
```

Здесь отдельные порции данных (chunk) представлены инстанциями специального класса Buffer. Эти порции накапливаются в массиве и после завершения передачи данных клиентом их можно легко преобразовать в обычное строковое представление с помощью методов concat() и toString(). Первый из них объединяет элементы массива в единое целое, а второй – преобразует получившийся в результате буфер в обычную строку JavaScript:

```
req.on('end', function() {
 var body = Buffer.concat(chunk_arr).toString();
 });
```

Здесь же уместно реализовать выдачу клиенту ответа от сервера. Если объединить всё вместе, то получим следующий код:

```
var http = require('http');

http.createServer(function(req, res) {
 var chunk_arr = [];
 req.on('data', function(chunk) {
    chunk_arr.push(chunk);
 });
 req.on('end', function() {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');

    var
       method = req.method,
       headers = getHeaders(req),
       body = Buffer.concat(chunk_arr).toString();

    res.end(
       "<h1>" + method + "</h1>" +
       headers +
       "<h1>Body</h1>" +
       body
    );
 });
}).listen(3000);

console.log('Ok');

function getHeaders(req) {
 var headers = [];
 for(var nm in req.headers)
    headers.push(nm + '=' + req.headers[nm]);
 return headers.join("<br/>\n");
}
```

Отметим, что если требуется точное воспроизведение заголовка ответа, то можно явно – как есть – отправить заголовочную часть клиенту с помощью метода writeHead(), например так:

```
res.writeHead(200, {
  'Content-Type': 'application/json',
  'X-Powered-By': 'bacon'
});
```

Наконец, чтобы увидеть результат работы эхо-сервера для запроса с телом (POST), можно первоначально загрузить форму:

```
<form method='POST' action='http://localhost:3000'>
 <input type='text' name='fld1'>
 <input type='text' name='fld2'>
 <input type="submit" value='Ok'>
</form>
```

В представленный код предлагается самостоятельно добавить обработку ошибок – реакции на событие «error» для запроса и для ответа:

```
req.on('error', function(err) { … });
res.on('error', function(err) { … });
```

3.3.2. Файловый сервер

Речь идёт о простейшем сервере по обслуживанию статических файлов, подобном Apache или IIS. Для реализации потребуется дополнительный модуль mime (для работы с mime типами данных), о котором шла речь выше и который следует установить, если он ещё не установлен. Кроме этого, потребуются встроенные модули fs – для работы с файловой системой, path – для манипулирования путями к файлам, и, конечно, http.

Определим, как и полагается, корневой каталог сервера в переменной root. Относительно этого каталога определяется положение запрашиваемого клиентом ресурса, полный путь к которому вместе с именем будем хранить в переменной abs_path. Чтение данных из файла может быть реализовано по-разному, например с помощью метода readFile():

```
fs.readFile(abs_path, function(err, data) {
 if(err) {  } else {  }
});
```

Методу передаются два параметра: путь к файлу и функция, которая будет вызвана после того, как данные будут прочитаны в буфер data. Если по каким-то причинам файл не может быть прочитан, то переменная err будет ссылаться на объект, содержащий информацию об ошибке. Эту информацию следует проанализировать (в блоке после if) и сообщить об этом клиенту с соответствующим кодом статуса. Если файл прочитан успешно, то буфер data следует «отдать» клиенту с помощью метода write() и завершить соединение с помощью метода end().

Прежде чем пытаться прочитать файл, следует проверить его существование с помощью функции exists(), экспортируемой всё тем же модулем fs:

```
fs.exists(abs_path, function(exists) {
 if(exists) {   } else {   }
});
```

Если файл не существует, то согласно протоколу http следует выдать клиенту соответствующее сообщение с кодом статуса 404 (в блоке после ключевого слова else).

Для формирования поля Content-Type воспользуемся методом lookup() модуля mime:

```
var mime_type = mime.lookup( path.basename(abs_path) );
```

Тип данных mime определяется по имени файла, которое выделяется из abs_path с помощью метода basename() модуля path. Заметим, что в современных версиях модуля mime метод lookup() заменён на метод getType().

Объединим всё вместе и в результате получим следующий код:

```
var fs = require('fs');
var path = require('path');
var http = require('http');
var mime = require('mime');

var root = './public', abs_path = '';

http.createServer(function(req, res) {
 var url = req.url;

 abs_path = root + url;

 if(url == '/')   abs_path += 'index.html';

 var mime_type = mime.lookup( path.basename(abs_path)
);

 fs.exists(abs_path, function(exists) {
    if(exists) {
       fs.readFile(abs_path, function(err, data) {
          if(err) {}
          else {
             res.writeHead(200, {
                'Content-Type': mime_type
             });
             res.write(data);
             res.end();
```

```
            }
        });
    }

});


}).listen(3000);
```

 Этот код предлагается дополнить анализом ошибок с выдачей клиенту соответствующих сообщений. Также можно предусмотреть кэширование файлов в памяти сервера.

Можно отказаться от чтения содержимого файла в буфер памяти целиком, а реализовать потоковый способ чтения с отдачей порций считанных данных клиенту:

```
var stream = fs.createReadStream(abs_path);
stream.on('data', function(chunk){
 res.write(chunk);
});
```

Такой подход хорош для больших файлов, поскольку требует меньше оперативной памяти компьютера.

3.4. Web-сокеты на примере реализации простейшего чата пользователей

Сервисы, построенные на основе протокола http, не позволяют держать связь клиента с сервером постоянно. Технология AJAX в этом плане не исключение. Клиент инициирует TCP-связь с сервером, который ожидает входящие соединения, прослушивая заданный порт. После соединения происходит обмен данными между сервером и клиентом согласно http-протоколу, известный как цикл «запрос – ответ» и соединение завершается.

В то же время существует множество online-задач в реальном времени, когда данные со стороны сервера должны быть переданы немедленно. В этом случае соединение не должно разрываться либо должно инициироваться со стороны сервера. Однако в последнем случае клиент и сервер просто меняются местами. Простейшим примером такой задачи может являться организация общения (чата) нескольких клиентов. Сообщение, поступившее от одного клиента, должно быть ретранслировано сервером всем остальным подключившемся клиентам.

Существуют различные способы решения подобных задач. Если используется исключительно http-протокол, то простейший вариант решения – это пе-

риодический AJAX-опрос сервера со стороны клиента, например с периодом 0.5 секунды, так чтобы не пропустить появление на сервере online-данных. Однако такой подход (известный как AJAX Polling) предполагает слишком напряжённый трафик. Другое решение, известное как AJAX Long-Polling, предполагает задержку ответа со стороны сервера. Сервер ждёт появления online-данных, передаёт их клиенту и завершает цикл «запрос – ответ», а клиент тут же инициирует новый Long-Polling запрос. Проблема такого подхода – длинные задержки циклов обмена «запрос – ответ», которые серверы и клиенты не всегда приветствуют, а иногда и запрещают.

Наконец, решением подобных задач может оказаться использование web-сокетов. В этом случае клиент устанавливает соединение с сервером web-сокетов и уже не разрывает это соединение. Сервер имеет возможность немедленно передать клиенту данные в реальном времени. Технология web-сокетов поддерживается стандартом HTML5, а платформа node.js располагает модулями для реализации сервисов web-сокетов.

Как только клиент решил, что он хочет открыть WebSocket, создается специальный javascript-объект. Этот объект создаёт TCP-соединение к заданному порту сервера и выдает немного необычный GET-запрос – с дополнительными заголовками. Если сервер поддерживает WebSocket, то он отвечает. Если клиента это устраивает, то TCP-соединение остаётся открытым – канал обмена данными готов. Далее клиент и сервер выступают равноправными участниками обмена данными.

Обмен выполняется в виде дата-фреймов следующего вида:

```
0x00, <строка в кодировке UTF-8>, 0xFF
```

То есть просто строка текста, к которой спереди приставлен нулевой байт 0x00, а в конце – 0xFF. Так же можно передавать и бинарные данные. Для них используется дата-фрейм следующего вида:

```
0x80, <длина - один или несколько байт>, <тело сообще-
ния>
```

Чтобы не создавать ограничений на длину передаваемого сообщения и в то же время не расходовать байты нерационально, разработчики протокола использовали следующий алгоритм. Каждый байт в указании длины рассматривается по частям: самый старший бит указывает, является ли этот байт последним (0) или же за ним есть другие (1), а младшие 7 битов содержат собственно данные.

3.4.1. Разработка клиентской части чата

Порядок работы чата следующий. Очередной участник (клиент) после успешного подключения к серверу web-сокетов получает от него историю чата (последние 100 сообщений), а в первом сообщении серверу определяет своё имя. Получив имя, сервер назначает цвет для подсветки сообщений подклю-

чившегося клиента (для наглядности). Все остальные сообщения от этого клиента ретранслируются сервером всем подключившемся участникам чата.

Индексная страница (index.html) содержит поле с содержимым чата (content), в котором будут отображаться сообщения всех участников, поле для ввода сообщений (input), вспомогательное поле статуса (status):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Chat</title>
  </head>
  <body>
    <div id="content"></div>
    <div>
      <span id="status">Соединение...</span>
      <input type="text" id="input" dis-
abled="disabled"/>
    </div>
  </body>
</html>
```

Предусмотрим также стилевое оформление для представленных элементов интерфейса:

```
<style>
* {
  font-family:tahoma;
  font-size:12px;
  padding:0px;
  margin:0px;
}
p { line-height:18px; }
div { width:500px; margin-left:auto; margin-
right:auto;}
#content {
  padding:5px;
  background:#ddd;
  border-radius:5px;
  overflow-y: scroll;
  border:1px solid #CCC;
  margin-top:10px;
  height: 160px;
}
#input {
  border-radius:2px;
  border:1px solid #ccc;
```

```
    margin-top:10px; padding:5px; width:400px;
  }
  #status {
    width:88px;
    display:block;
    float:left;
    margin-top:15px;
  }
  </style>
```

Эту стилевую таблицу следует разместить в секции head. Для работы с документом будем пользоваться библиотекой jQuery, а программную часть разместим в файле frontend.js:

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

Эти строки также разместим в секции head.

Прежде всего проверяем – поддерживает ли браузер web-сокеты. Если нет, то сообщаем об этом пользователю в поле content, гасим все остальные поля и заканчиваем работу:

```
$(function () {
  var content = $('#content');
  var input = $('#input');
  var status = $('#status');
  var myName = '', myColor = '';

  // В браузере Firefox свой встроенный WebSocket
  window.WebSocket = window.WebSocket || win-
dow.MozWebSocket;

  // Если браузер не поддерживает web-сокеты
  if (!window.WebSocket) {
    content.html(
      $('<p>', {
        text: 'Ваш браузер не поддерживает web-сокеты'
      })
    );
    input.hide();
    $('span').hide();
    return;
  }
  // …
}
```

Переменные myName и myColor предусмотрены для хранения имени участника чата и цвета для подсветки его сообщений. Далее пытаемся подключиться к серверу, создавая объект connection типа WebSocket:

```
var connection = new WebSocket('ws://127.0.0.1:1337');
```

Этот объект является генератором событий: open (при успешном подключении), error (при возникновении ошибки) и message (когда приходит сообщение от сервера). Следует предусмотреть реакции на все события.

В случае ошибки поступаем так же, как при отсутствии поддержки браузером web-сокетов:

```
connection.onerror = function (error) {
  content.html(
    $('<p>', {
      text: 'Проблемы связи с сервером'
    })
  );
  input.hide();
  $('span').hide();
};
```

При успешном подключении готовим пользователя к вводу своего имени:

```
connection.onopen = function () {
  input.removeAttr('disabled');
  status.text('Введите своё имя:');
};
```

Прежде чем перейти к обработке данных, поступивших от сервера, рассмотрим формат этих данных. Данные от сервера поступают в виде json-строк, каждая из которых описывает объект с двумя полями: первое поле – это тип поступивших данных (type), а второе – это собственно сами данные (data). Данные могут содержать назначенный сервером цвет для подсветки сообщений, историю чата или вновь поступившее сообщение от любого другого участника чата. В первом случае поле type будет содержать строку color, во-втором – history, наконец, в третьем – message. Если передаётся история, то данные, в свою очередь, будут представлять массив последних 100 сообщений участников чата. Каждый элемент этого массива – объект, который содержит имя автора (author), текст сообщения (text), цвет (color) и время сообщения (time). Сообщения воспроизводятся в окне content функцией:

```
function addMessage(author, message, color, dt) {
  content.prepend('<p><span style="color:' + color +
'">' +
    author + '</span> @ ' +
    (dt.getHours() < 10 ? '0' + dt.getHours() :
dt.getHours()) +
    ':' +
    (dt.getMinutes() < 10 ? '0' + dt.getMinutes() :

dt.getMinutes()) +
     ': ' + message + '</p>');
}
```

Согласно формату сообщений сервера, их обработка выполняется следующим образом:

```
connection.onmessage = function (message) {
  try {
    var json = JSON.parse(message.data);
  } catch (e) {
    console.log('Ошибка формата JSON: ', message.data);
    return;
  }
  if(json.type === 'color') {
    myColor = json.data;
    status.text(myName + ': ').css('color', myColor);
    input.removeAttr('disabled').focus();
  } else if(json.type === 'history') {
    // каждое сообщение из истории в окно с содержимым
чата
    for(var i=0; i < json.data.length; i++) {
      addMessage(json.data[i].author,
json.data[i].text,
        json.data[i].color, new
Date(json.data[i].time));
    }
  } else if(json.type === 'message') { // сообщение
участника чата
    // позволим пользователю ввести свое сообщение
    input.removeAttr('disabled');
    addMessage(json.data.author, json.data.text,
        json.data.color, new Date(json.data.time));
  } else {
    console.log('Некорректный формат данных от сервера:
', json);
  }
};
```

В представленном фрагменте кода при обработке сообщений типа color и message снимается блокировка (disabled) с поля ввода (input). Необходимость в этом действии становится понятной, если разобраться с порядком отправки сообщений пользователя чата. Изначально поле ввода сообщений (input) недоступно (disabled), а статус (status) сообщает о процессе соединения. После успешного подключения убирается атрибут disabled – поле ввода становится доступным, а статус подсказывает о необходимости представиться – ввести своё имя (в обработчике события open, представленном выше). Пользователь вводит имя и нажимает клавишу «ввод» (return). Имя должно быть отправлено серверу, а поле ввода вновь заблокировано до тех пор, пока не будет получен ответ от сервера – цвет сообщений. Этим цветом в поле статуса воспроизводится имя участника чата.

Аналогично выполняется отправка содержательных сообщений пользователя – по нажатию клавиши return. Поле ввода блокируется до момента получения от сервера эха отправленного сообщения. Эхо означает, что сообщение отправлено успешно и его можно воспроизвести в поле content.

Описанные действия реализуем в обработчике нажатия клавиши return:

```
input.keydown(function(e) {
  if(e.keyCode === 13) {
    var msg = $(this).val(); if(!msg) return;
    // отсылаем сообщение серверу
    connection.send(msg);
    $(this).val('');
    // блокируем поле ввода пока сервер не ответит
    input.attr('disabled', 'disabled');
    // первое сообщение от пользователя - его имя
    if(myName === false) myName = msg;
  }
});
```

Наконец, можно предусмотреть реакцию на отсутствие ответа сервера в течение определённого промежутка времени – таймаута (например, после 3 секунд ожидания):

```
setInterval(function() {
  if(connection.readyState !== 1) {
    status.text('Error');
    input.attr('disabled', 'disabled');
    input.val('Сервер не отвечает');
  }
}, 3000);
```

Предлагается самостоятельно скомпоновать представленные фрагменты кода в единое целое – соединить их в той последовательности, в которой они приведены в тексте.

3.4.2. Разработка серверной части чата

Для реализации сервера воспользуемся дополнительным модулем websocket (описание здесь: https://www.npmjs.com/package/websocket). Переходим в папку разработки чата и устанавливаем этот модуль:

```
npm install websocket
```

Поскольку сервер web-сокетов использует соединение, которое первоначально устанавливается между клиентом и web-сервером (tcp-соединение), потребуется реализовать также и web-сервер. С небольшими изменениями повторим реализацию простейшего web-сервера, подробно описанного выше:

```
// С таким именем сервер будет виден в системе, напри-
мер,
// в unix-подобных системах по команде 'ps' или 'top'
process.title = 'node-chat';

// Подключаем модули
var webSocketServer = require('websocket').server;
var http = require('http');
var fs = require('fs');
var path = require('path');
var mime = require('mime');
var htmlEntity = require('./htmlEntity').htmlEntity;

// История чата - последние 100 сообщений
var history = [ ];
// Список соединений активных клиентов
var clients = [ ];

// Цвета, которые назначаем участникам чата
var colors = [
  'red', 'green', 'blue', 'magenta', 'purple', 'plum',
'orange'
];
// Перемешаем цвета
colors.sort(function(a,b) { return Math.random() > 0.5;
} );
```

```
/**
 * http-сервер
 */
var server = http.createServer( function(req, res) {
 var filePath = '';

 if(req.url == '/') filePath = './index.html';
else filePath = '.' + req.url;

 fs.exists(filePath, function(exists) {
   if( !exists ) {
     res.writeHead(404, { 'content-type': 'text/plain'
} );
     res.write('Not found');
     res.end();
     return;
   }
   fs.readFile(filePath, function(err, data) {
     if(err) {
       res.writeHead(404, { 'content-type':
'text/plain' } );
       res.write('Not found');
       res.end();
       return;
     }
     res.writeHead(200, {
       'content-type': mime.lookup(
path.basename(filePath) )       });
     res.write(data);
     res.end();
   });
 });
});

server.listen(1337, function() {
 console.log((new Date()) + " сервер слушает порт
1337");
});
```

В приведённом фрагменте кода подключается модуль htmlEntity, который представлен выше и который должен быть скопирован в папку проекта. Кроме этого, здесь же определены глобальные массивы: history – для хранения истории чата (100 последних сообщений), clients – для хранения объектов, представляющих соединения с сервером активных участников чата, и colors – для хранения названий цветов, которые тут же перемешиваются в случайном порядке.

Теперь можно создать сервер web-сокетов, который «привязан» к созданному объекту web-сервера («привязан» значит пользуется tcp-соединением с клиентом web-сервера):

```
var wsServer = new webSocketServer({
 // Сервер WebSocket привязан к HTTP-серверу.
 // Запрос WebSocket - это просто расширенный HTTP-
запрос.
 // (http://tools.ietf.org/html/rfc6455#page-6)
 httpServer: server
});
```

Центральной частью разрабатываемого кода является функция обратного вызова – реакция нашего сервера web-сокетов на попытку подключения очередного участника чата:

```
wsServer.on('request', function(request) {
 console.log((new Date()) + ' Соединение от ' + re-
quest.origin);

 // Проверяем, что клиент подключается с нашего сайта
 // (http://en.wikipedia.org/wiki/Same_origin_policy)
 var connection = request.accept(null, request.origin);
 // сохраняем это соединение и запоминаем его индекс в
массиве,
 // чтобы удалить при отключении клиента (событие
'close')
 var index = clients.push(connection) - 1;
 var userName = false;
 var userColor = false;

 console.log((new Date()) + ' Соединение принято');

 // отсылаем историю чата
 if(history.length > 0) {
    connection.sendUTF(JSON.stringify( {
       type: 'history', data: history
    } ));
 }
 …
}
```

Таким образом, при успешном подключении ссылка на это подключение (connection) сохраняется в массиве clients, а клиенту отсылается история чата. Индекс ссылки запоминается в переменной index – она потребуется в дальней-

шем для её удаления при отключении клиента. Под имя и цвет клиента резервируются переменные с соответствующими именами – userName и userColor.

Для объекта connection реализованы события message и close. Первое происходит при получении сообщения от клиента, а второе – при отключении клиента. Прежде чем перейти к описанию обработчиков, отметим, что вставить их следует вместо многоточия в представленном выше фрагменте кода.

Получив сообщение от клиента (событие message), если это сообщение не является первым, ретранслируем его всем остальным участникам, информация о подключении которых содержится в массиве clients. Если сообщение первое (в переменной username отсутствует имя пользователя), то в сообщении содержится имя клиента, которое следует запомнить, а клиенту отослать цвет для выделения его сообщений:

```
connection.on('message', function(message) {
 if(message.type === 'utf8') { // принимаем только
текст
    if(userName === false) { // первое сообщение – имя
клиента
      // запомним имя
      userName = htmlEntities(message.utf8Data);
      // назначим цвет сообщений подключившегося клиента
      userColor = colors.shift();
      // отправим цвет клиенту
      connection.sendUTF(JSON.stringify({
        type:'color', data: userColor
      }));
      console.log(
        (new Date()) + ' Пользователь под именем: ' +
username +
      ' и цветом ' + userColor
      );
    }
    else { // ретранслируем сообщение остальным участни-
кам чата
      console.log(
        (new Date()) + ' Получено сообщение от ' +
        userName + ': ' + message.utf8Data
      );

      // сохраним это сообщение
      var obj = {
        time: (new Date()).getTime(),
        text: htmlEntities(message.utf8Data),
              author: userName,
```

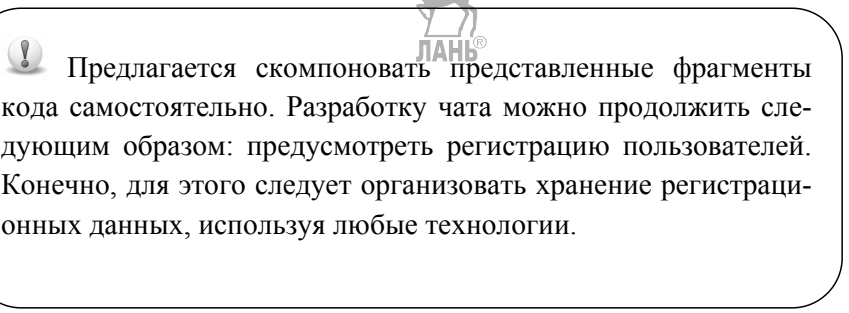
```
                color: userColor
        };
        history.push(obj);
        // урежем историю до 100 последних сообщений
        history = history.slice(-100);

        // ретранслируем сообщение всем участникам чата
        var json = JSON.stringify({ type:'message', data:
obj });
        for (var i=0; i < clients.length; i++) {
          clients[i].sendUTF(json);
        }
    }
 }
});
```


Несмотря на кажущуюся простоту представленного кода, здесь реализован ряд нетривиальных приёмов, на которые следует обратить внимание. Переменные userName и userColor определены внутри обработчика события request, где также определены обработчики событий message и close (обработчик close представлен ниже). Таким образом, эти переменные образуют замыкания с обработчиками (замыкания описаны в [1]). Замыкания гарантируют, что обработчики будут оперировать со своими экземплярами упомянутых переменных userName и userColor для каждого соединения (а значит, и для каждого клиента). В то же время массивы clients, history, colors остаются общими для всех соединений и клиентов.

Наконец, следует предусмотреть реакцию сервера на отключение клиента – событие close. При отключении следует удалить соответствующий объект подключения из массива clients, а также вернуть цвет, которым пользовался клиент для выделения своих сообщений, в массив colors для дальнейшего использования:


```
connection.on('close', function(connection) {
 if(userName !== false && userColor !== false) {
    console.log(
       (new Date()) + " Клиент " +
       connection.remoteAddress + " отключён."
    );
    // удалим клиента из списка подключённых клиентов
clients
    clients.splice(index, 1);
    // вернём цвет для повторного использования
    colors.push(userColor);
 }
});
```



Предлагается скомпоновать представленные фрагменты кода самостоятельно. Разработку чата можно продолжить следующим образом: предусмотреть регистрацию пользователей. Конечно, для этого следует организовать хранение регистрационных данных, используя любые технологии.

Отметим, что платформа node.js располагает богатым арсеналом модулей для работы с реляционными базами данных, однако обсуждение этих вопросов выходит за рамки данного пособия. Наконец, можно организовать разделение чата по интересам, темам или комнатам.

4. Практикум

4.1. Постановка задания

Требуется разработать приложение для выполнения расчетных исследований многопараметрической зависимости, в качестве которой взять один из вариантов, приведённых в таблице 1. Клиентская часть приложения (frontend разработка) должна обеспечить ввод исходных данных и вывод результатов расчётов в табличном и графическом виде. Расчёты должны выполняться на стороне сервера. Обмен данными между клиентом и сервером должен выполняться с помощью технологии AJAX, без перезагрузок страниц обозревателя. Исходные данные для расчёта со стороны клиента следует передавать в теле POST-запроса, а результаты расчётов со стороны сервера – в формате JSON. Рекомендуется использовать библиотеку jQuery.

Серверную часть (backend разработку) следует реализовать на платформе node.js. Кроме расчётов, серверная часть должна обеспечить функции http-сервера по загрузке стартовой страницы приложения вместе с иными ресурсами, связанными с этой страницей (css- и js-файлами, рисунками и т. д.), если, конечно, такие ресурсы предусмотрены разработчиком. Решение поставленной задачи, основанное только на клиентской разработке, можно найти в [4]. Использованный в данном пособии подход позволяет изучить и освоить ряд современных технологий web-разработки (AJAX, JSON, node и т. д.).

Разработка должна обеспечить ввод и варьирование трех указанных параметров. Неизменяемые параметры зависимости могут быть заданы как постоянные величины из указанного диапазона. Оставшиеся варьируемые параметры задаются следующим образом. Первый как последовательность любых значений из заданного диапазона. Два оставшихся – начальным и конечным значениями, а также величиной шага, которые определяют правило варьирования этих параметров: от начального значения до конечного значения с заданным шагом.

Таблица 1

№ варианта	Расчётная формула	Неизменяемые параметры	Варьируемые параметры: первый задаётся массивом, второй и третий – начальным и конечным значениями и шагом изменения
1	2	3	4
1	Производительность сучкорезной стационарной установки ПОЛ-2: $П_{СМ} = \frac{T_{СМ} \cdot f_1 \cdot f_2 \cdot v \cdot q}{l}$, где $П_{СМ}$ – производительность [м^3]	$T_{СМ}$ = 240–360 – продолжительность смены [мин]; f_1 = 0,8–0,95 – коэффициент использования рабочего времени; f_2 = 0,7–0,9 – коэффициент загрузки оборудования	v = 0,4–0,8 – скорость подачи хлыста [м/с]; q = 0,1–1,15 – средний объем хлыста [м^3]; l = 8–21 – средняя длина хлыста [м]

Продолжение табл. 1

1	2	3	4
2	Производительность раскряжевочной установки триммерного типа при раскряжёвке хлыстов на сортименты: $A = \frac{U \cdot T_{CM} \cdot q \cdot k}{h}$, где A – производительность [м^3/смена]	T_{CM} = 360–480 – продолжительность смены [мин]; k = 0,8–0,9 – коэффициент использования установки [1/смена]	U = 13–15 – скорость подачи [м/мин]; q = 1,1–1,5 – средний объем хлыста [м^3]; h = 1,5–2,4 – расстояние между упорами [м]
3	Сменная производительность многопильного круглопильного станка СБ-8: $A = \frac{60 \cdot T_{CM} \cdot q \cdot k_1 \cdot k_2 \cdot v}{l_{CP}}$, где A – производительность станка [м^3/смена]	T_{CM} = 6–8 – продолжительность смены [ч]; k_1 = 0,6–0,8 – коэффициент использования потока; k_2 = 0,8–0,95 – коэффициент использования смены [1/смена]	q = 0,09–0,14 – объем бревна [м^3]; v = 20–45 – скорость подачи [м/мин]; l_{CP} = 4–6 – средняя длина бревна [м]
4	Производительность трелевочного трактора: $П = \frac{2980 \cdot M}{20,7 \cdot V \cdot L + T \cdot M}$, где $П$ – производительность [м^3/с]	V = 0,1–1,15 – коэффициент сопротивления процессу [кВт·с/м^4]	M = 240–360 – мощность двигателя [кВт]; L = 0,4–0,8 – среднее расстояние трелевки [м]; T = 8–21 – время погрузки и разгрузки пачек [с/м^3]
5	Цикловая производительность обрезного станка: $Q = \frac{60}{1,5 + 60 \cdot (l - 0,6) \cdot V + t}$, где Q – цикловая производительность [шт./ч]		l = 3–7,6 – длина доски [м]; V = 80–120 – скорость подачи [м/с]; t = 0,8–1 – время укладки доски на стол [с]
6	Производительность сверлильно-присадочного станка: $П = \frac{60 \cdot K_M \cdot K_Д \cdot T_{CM} \cdot N}{Ф_Ц \cdot Z}$, где $П$ – производительность станка [шт. дет./смена]	K_M = 0,75–0,85 – коэффициент машинного времени; $K_Д$ = 0,8–0,9 – коэффициент рабочего времени; Z = 2–14 – число одновременно выбираемых отверстий на кромках щита [шт.]	T_{CM} = 480–640 – продолжительность смены [мин]; N = 2–12 – число выбираемых отверстий [шт.]; $Ф_Ц$ = 1–16 – время цикла [мин]

Продолжение табл. 1

1	2	3	4
7	Требуемое количество жидкого клея на кубометр фанеры: $Q = \frac{q \cdot (m-1) \cdot K_O \cdot K_П}{S_Ф}$, где Q – требуемое количество жидкого клея [г]	K_O = 1,05–1,28; $K_П$ = 1,03–1,05. K_O – коэффициент, учитывающий потери клея при обрезке материала; $K_O = F_н/F_о$, где $F_н$ – площадь необрезанного листа фанеры, м^2; $F_о$ – площадь обрезного листа фанеры, м^2. $K_П$ – коэффициент, учитывающий потери клея при его изготовлении и последующем использовании	q = 85–96 – техническая норма расхода [г/м^2]; m = 2–5 – слоистость фанеры; $S_Ф$ = 2–5 – толщина фанеры [мм]
8	Продолжительность сушки пиломатериалов в камере с принудительной вентиляцией: $T = t_{исх} \cdot A_Р \cdot A_К \cdot A_Ц \cdot 1,43 \cdot \lg \frac{W_Н}{W_К}$, где T – продолжительность сушки пиломатериалов [ч]	$A_Р$ = 1,7–1,9 – коэффициент, учитывающий жесткость режима сушки; $A_К$ = 1,1–1,3 – коэффициент качества; $A_Ц$ = 0,8–0,9 – коэффициент, учитывающий характер и интенсивность циркуляции воздуха в камере	$t_{исх}$ = 33–95 – исходная продолжительность сушки пиломатериалов [ч]; $W_Н$ = 70–80 – начальная влажность древесины [%]; $W_К$ = 10–20 – конечная влажность древесины [%]
9	Сменная производительность лесопильной рамы: $A = \frac{d \cdot n \cdot T \cdot K_И \cdot q}{1000 \cdot l}$, где A – сменная производительность [м^3]	n = 320–400 – число оборотов вала [об/мин]; T = 480–560 – продолжительность смены [мин]; $K_И$ = 0,8–0,9 – коэффициент использования лесопильной рамы	d = 35–44 – посылка бревна [мм]; q = 0,1–0,25 – объем бревна [м^3]; l = 4–6 – длина бревна [м]
10	Объем пиломатериалов в сушильной камере: $V_Н = \frac{(H+B) \cdot (B+S)}{(Y+P) \cdot (X+S)} \cdot X \cdot Y \cdot D$, где $V_Н$ – объем пиломатериалов [м^3]	H = 1,3–1,5 – высота пакета [м]; B = 1,3–1,8 – ширина пакета [м]; S = 0,025–0,035 – ширина шпации [м]; P = 0,02–0,03 – толщина прокладки [м]	Y = 0,02–0,028 – толщина досок [м]; X = 0,08–0,23 – ширина досок [м]; D = 4,5–7 – длина досок [м]

Продолжение табл. 1

1	2	3	4
11	Производительность бесчекерного трактора: $П = \frac{(T_{СМ} - \tau)\cdot\varphi\cdot X\cdot q}{\frac{l}{v_{xx}} + X\cdot t_2 + \frac{l}{v_{mpx}} + t_П}$, где $П$ – сменная производительность [м^3/мин]	τ – подготовительно-заключительное время [мин]; φ – коэффициент использования рабочего времени; v_{xx} = 80 м/мин – скорость движения трактора без груза; q = 0.4 – средний объем хлыста [м^3]; t_2 = 1 мин – время погрузки одного хлыста; v_{mpx} = 40 м/мин – скорость движения трактора с грузом; $t_П$ = 2 мин – время разгрузки пачки хлыстов	$T_{СМ}$ = 420–480 – продолжительность смены [мин]; X = 5–25 – число хлыстов; l = 100–50 – расстояние трелевки [м]
12	Мощность резания при пазовом фрезеровании сосны: $N = A\cdot\frac{\Delta^{0.78}\cdot D^{1.2}\cdot h^{1.7}\cdot n}{9755\cdot\gamma^{0.06}}$, где N – мощность резания [кВт]	Δ = 2 – подача на один оборот фрезы; h = 50 – глубина снимаемого слоя [мм]	D = 20–45 – диаметр фрезы [мм]; n = 1500–3500 – скорость вращения фрезы [об/мин]; γ= 10–20 – передний угол резания [град]
13	Коэффициент использования производственной мощности трактора: $HM = \frac{17,25\cdot L\cdot V}{20,7\cdot V + T\cdot M}$	M = 150 – мощность двигателя [кВт]	L = 100–700 – среднее расстояние трелевки [м]; V = 0,6–1,4 – коэффициент сопротивления [с·кВт/м^4]; T = 20–100 – время погрузки и разгрузки пачки, отнесённое к м^3 леса [с/м^3]
14	Необходимое количество инструмента для цеха в год: $R = \frac{100\cdot l\cdot t_1\cdot T_{СМ}\cdot K_И\cdot z\cdot b\cdot n}{a\cdot t\cdot(100 - K_В)}$, где R – необходимое количество инструмента в день [шт.]	l = 308 – число рабочих дней в году; t_1 = 4 – число смен в день; $T_{СМ}$ = 8 – продолжительность одной смены [ч]; b = 0,6 – величина стачивания инструмента за одну переточку [мм]; n = 5 – число единиц однотипных станков; a – допустимое уменьшение рабочей части инструмента [мм]; $K_В$ = 5 – аварийный выход инструмента из строя [%]	$K_И$ = 0,7–0,9 – коэффициент использования станков; z = 2–14 – наибольшее число инструментов на одном станке; t = 1–4 – период стойкости инструмента [ч]

Продолжение табл. 1

1	2	3	4
15	Линейный расход топлива лесовозным автопоездом: $Q = 0{,}14\times$ $\times\left(2 \cdot H_{Л} \cdot n \cdot b \cdot l + H_{W} \cdot l \cdot b \cdot c \cdot n \cdot G\right)$, где Q – линейный расход топлива [л]	$H_{Л} = 15$ – норма расхода топлива на 100 км пробега [л]; $b = 2$ – количество смен в вывозке [с]; $H_{W} = 2$ – норма расхода топлива на каждые 100 тыс. км пробега [л]; $c = 0{,}8$ – средняя масса одного кубометра леса [т/м^3]	$n = 2$–4 – количество рейсов за смену; $l = 50$–75 – расстояние вывозки [км]; $G = 12$–26 – рейсовая нагрузка [м^3]
16	Сменная производительность канатной установки: $P = \dfrac{\left(T_{CM} - t_{ПЗ}\right) \cdot \varphi_1 \cdot \varphi_2 \cdot Q}{t_1 + t_2 + \dfrac{v_1 + v_2}{v_1 \cdot v_2} L_{CP} + \dfrac{v_1' + v_2'}{v_1' \cdot v_2'} \cdot l_{CP}}$, где P – сменная производительность [м^3]	$t_{ПЗ} = 30$ – подготовительно-заключительное время [мин]; $\varphi_1 = \varphi_2 = 0{,}8$ – коэффициенты использования рабочего времени и грузоподъёмности соответственно; Q – объём трелюемой пачки [м^3]; $t_1 = 4$ и $t_2 = 3$ – время на прицепку и отцепку груза соответственно [мин]; $v_1 = 4$ и $v_2 = 7$ – скорость движения каретки с грузом и без груза соответственно [м/с]; $v_1' = 0{,}5$ – скорость перемещения крюка на лесосеку [м/с]; $v_2' = 1$ – скорость подтаскивания пачки к несущему канату [м/с]	$T_{CM} = 480$–640 – продолжительность рабочей смены [мин]; $L_{CP} = 200$–700 – среднее расстояние трелевки [м]; $l_{CP} = 0$–25 – среднее расстояние подтрелевки [м]
17	Сменная производительность продольного элеватора: $П = \dfrac{T \cdot v \cdot K_{M} \cdot K_{P} \cdot q}{l}$, где $П$ – сменная производительность [м^3/смена]	$T = 480$ – продолжительность смены [мин]; $K_{P} = 0{,}85$ – коэффициент использования рабочего времени; v – скорость подачи [м/мин]	$q = 0{,}065$–$0{,}52$ – объём бревна [м^3]; $K_{M} = 0{,}7$–$0{,}95$ – коэффициент использования машинного времени; $l = 4$–7 – длина бревна [м]
18	Оптимальная ширина лесосеки: $X = \sqrt{\dfrac{T \cdot M \cdot v_{CP}}{10^5 \cdot \varphi \cdot d \cdot q} \cdot \left(K \cdot A + \dfrac{B}{L}\right)}$, где X – оптимальная ширина лесосеки [м]	$T = 420$ – продолжительность смены [мин]; $v_{CP} = 25$ – средняя скорость движения трактора [м/мин]; $M = 4$ – средняя рейсовая нагрузка на трактор [м^3]; $d = 2$ – число рабочих, обслуживающих трактор; $K = 1{,}05$ – коэффициент, учитывающий внеэксплуатационную площадь; $B = 20$ – трудоёмкость строительства одного погрузочного пункта; $L = 0{,}5$ – длина лесосеки [км]	$q = 150$–300 – запас леса на один га [м^3]; $\varphi = 0{,}1$–$0{,}5$ – коэффициент, зависящий от принятой схемы волоков; $A = 100$–200 – трудоёмкость строительства 1 км

Продолжение табл. 1

1	2	3	4
19	Скорость подачи при шлифовании абразивными кругами: $U = 10,7 - 19,1 \times \frac{v \cdot 0,01 \cdot (1150 \cdot h + 800) + 0.2}{D \cdot m_D \cdot h^{1.1}} \cdot m_z \cdot m_n$, где U – скорость подачи при шлифовании [м/с]	h – глубина резания [мм]; v – скорость резания [м/с]; $m_n = 1$ – поправочный коэффициент, учитывающий породу древесины	$D = 110–290$ – диаметр шлифовального абразивного круга [мм]; $m_D = 0,79–1,32$, $m_z = 0,62–1,6$ – поправочные коэффициенты, учитывающие диаметр шлифовального круга и номер зернистости абразивного круга соответственно
20	Время нагрева центральной части бруса из древесины: $\tau_H = \frac{0,234 \cdot \lg(1,62 \cdot \frac{t_C - t_O}{t_C - t_Ц})}{\frac{a_\delta}{\delta^2} + \frac{a_h}{h^2}}$, где τ_H – время нагрева центральной части бруса из древесины [ч]	$t_C = 100°$ – температура обогревающей среды [°C]; $t_Ц = 1,2 \cdot t_O$ – температура центральной части бруса [°C]; $a_\delta = 6,1 \cdot 10^{-4}$ – коэффициент температурной проводимости в направлении измерения толщины бруса [м^2/ч]; $a_h = 6,9 \cdot 10^{-4}$ – коэффициент температурной проводимости в направлении измерения высоты бруса [м^2/ч]	$t_O = 5–27$ – начальная температура бруса [°C]; $\delta = 0,1–0,24$ – толщина бруса [м]; $h = 0,3–0,4$ – высота бруса [м]
21	Лесопропускная способность в створе реки при молевом лесосплаве: $N_{СУТ} = 3660 \cdot B_O \cdot T_O \cdot V_Э \cdot K \cdot q \cdot \beta$, $q = 0.785 \cdot d_{СР}$, где $N_{СУТ}$ – лесопропускная способность в створе реки	$T_O = 14$ – расчётная продолжительность сплава в сутки при условии непрерывной скатки древесины в воду [ч]; $K = 0,75$ – коэффициент перехода от поверхностной скорости к технической скорости движения брёвен; q – объём древесины, размещённой плотно друг к другу в ряд на 1 м^3 сплавного хода [м^3]; $\beta = 0,07$ – коэффициент заполнения сплавного хода плывущим лесом	$B_O = 40–52$ – расчётная ширина сплавного хода [м]; $V_Э = 0,4–0,55$ – средняя поверхностная скорость течения в зоне сплавного хода [м/с]; $d_{СР} = 0,26–0,34$ – срединный средневзвешенный диаметр брёвен сплавляемого леса [м]

Продолжение табл. 1

1	2	3	4
22	Зависимость динамической грузоподъёмности подшипника качения: $C = Z \cdot \left(0.6 \cdot 10^{-4} \cdot X \cdot Y\right)^{1/3}$, где C – динамическая грузоподъёмность подшипника [кН]		X = 1,1–1,5 – частота вращения внутреннего кольца подшипника [об/мин]; Y = 1,5–2,4 – долговечность подшипника [ч]; Z = 13–15 – эквивалентная нагрузка [кН]
23	Годовой объём переработки сырья лесопильным потоком на базе лесопильных рам 2Р75 при распиловке сырья средним диаметром 24 см: $Q = \frac{T \cdot C}{40{,}15 \cdot S}$, где Q – годовой объём [тыс. м^3/ч]		T = 2000–6000 – годовой фонд работы оборудования [ч]; C = 0,9–0,99 – коэффициент на среднегодовые условия работы оборудования; S = 0,8–0,9 – расчётное время работы цеха [ч].
24	Сбег пиловочных брёвен: $S = \frac{A + D}{28 + 2{,}5 \cdot l}$, где S – сбег бревна [см/м]		A = 10–35 – величина, зависящая от условий произрастания леса (бонитета) [см]; D = 14–30 – диаметр бревна в тонком конце [см]; l = 4–6,5 – длина бревна [м]
25	Критическая сила для стального стержня прямоугольного сечения при различных способах закрепления концов: $\lambda = 10^3 \cdot \frac{\mu \cdot l \cdot \sqrt{12}}{c}$, $P_{KP} = \begin{cases} \frac{3{,}14^2 \cdot h \cdot c^3}{6 \cdot 10^4 \cdot (\mu \cdot l)^2} \; при \; \lambda \geq 100 \\ C \cdot H \cdot (304 - 1{,}12 \cdot \lambda) \cdot 10^{-3}, \; при \; 100 > \lambda > 60, \\ 0{,}24 \cdot C \cdot h, \; при \; \lambda < 60 \end{cases}$ где P_{KP} – критическая сила [кН]	λ – гибкость стержня [мм]; C = 60 – ширина поперечного сечения стержня [мм]	μ = 0,5–2 – коэффициент, учитывающий характер закрепления концов стержня; l = 1,5–4,5 – длина стержня [м]; h = 100–160 – высота поперечного сечения стержня [мм]

4.2. Пример выполнения задания

4.2.1. Расчетная формула

Расстояние, пройденное при равноускоренном движении:

$$s = f(v_0, a, t) = v_0 \cdot t + \frac{a \cdot t^2}{2},$$

где v_0 – начальная скорость [м/с];

a – ускорение [м/с^2];

t – время движения [с];

s – расстояние [м].

4.2.2. Исходные данные

$a_1, a_2, a_3, \ ..., \ a_n$ – массив значений ускорения a;

$v_{0_n}, v_{0_k}, \Delta v_0$ – начальное значение, конечное значение и шаг изменения параметра v_0;

$t_n, t_k, \Delta t$ – начальное значение, конечное значение и шаг изменения параметра t.

4.2.3. Каркас приложения

Модальные окна. Для вывода результатов расчётов (таблиц и графиков) реализуем свой оригинальный вариант модальных окон на основе html и css. Весь код, связанный с управлением модальными окнами, разместим в файле ModalWindow.js, а стили оформления модальных окон – в файле ModalWindow.css.

Графики. Для построения графиков воспользуемся сторонней разработкой, их можно найти немало на просторах Всемирной паутины, платных и бесплатных, например бесплатная библиотека flot (http://code.google.com/p/flot/, http://www.flotcharts.org/). Также будет хорош выбор бесплатного модуля chart.js (http://www.chartjs.org/) или платной библиотеки CanvasJS (https://canvasjs.com/), достаточно использовать trial-версию. Во всех перечисленных вариантах используется возможность рисования в canvas, появившаяся в HTML5. Альтернативой является графика SVG (например, https://developers.google.com/chart/) или flash.

Скачиваем библиотеку flot с указанного выше сайта и разворачиваем архив в подкаталоге flot в рабочем каталоге проекта. Из всей сборки (версия 0.8.3) потребуется файл jquery.flot.js.

В клиентскую часть входят:

1. Стартовая страница index.html, которая содержит поля для ввода исходных данных, а также подгружает все необходимые файлы, представленные далее в этом списке.

2. Стилевые таблицы модальных окон – ModalWindow.css и приложения – app.css.
3. Библиотека jquery.js.
4. Управление модальными окнами – ModalWindow.js.
5. Плагин jquery.flot.js для рисования графиков.
6. Программный код в файле frontend.js реализует функционал frontend-разработки: контроль и ввод исходных данных, передачу данных серверу, получение от сервера результатов, представление результатов в табличном и графическом виде, а также переход к страницам about_author.html и task_description.html.
7. Страница about_author.html должна содержать сведения о разработчике.
8. Страница task_description.html должна содержать подробное описание решаемой задачи.

Названия страниц можно менять. Содержимое перечисленных файлов, за исключением about_author.html и task_description.html, которые студент полностью разрабатывает самостоятельно, приводится в приложении.

Серверная часть содержит всего один файл server.js, который выполняет функции простого http-сервера для загрузки ресурсов клиентской части, AJAX обмен с клиентской частью входными данными и результатами расчёта и, конечно, сам расчёт.

4.2.4. Реализация модальных окон

Для вывода результатов расчёта будем использовать модальные окна, которые реализуем с помощью html и css самостоятельно, несмотря на то, что существует достаточное количество плагинов jQuery, использующих подобный подход. Напомним, что модальное окно должно фокусировать пользователя на своём содержимом и блокировать всё, что находится вне этого окна. Необходимость подобного моделирования модальных окон частично объясняется тем, что современные web-обозреватели часто блокируют всплывающие окна, созданные с помощью встроенных функций alert(), confirm(), prompt(), open(), showModalDialog().

Идея состоит в следующем: создаётся прозрачный слой div, который играет роль маски, перекрывающей всю видимую часть окна и блокирующий доступ ко всем управляющим элементам. Поверх этой маски воспроизводится модальное окно – div-слой требуемых размеров, содержащий шапку с кнопкой для закрытия окна и основную часть с рабочим содержимым.

Окна целесообразно реализовать в виде класса, который назовём ModalWindow. В конструкторе этого класса создадим описанную выше структуру окна (файл ModalWindow.js):

```
function ModalWindow(id, w, h) {
 this.id = id;
 this.width = w; this.height = h;
 // шапка окна
 this.top = $('<div/>', { class: 'top' });
 this.close = $('<a/>', { class: 'close', text:
'Закрыть' });
 this.top.append( this.close );
 // содержимое окна
 this.content = $('<div/>', { id: id, class: 'content'
});
 // окно
 this.modalWin = $('<div/>', { class: 'window' });
 this.modalWin.append(this.top);
 this.modalWin.append(this.content);
 // добавим окно в конец тела документа
 $(document.body).append( this.modalWin );
 // маска
 this.mask = $('<div/>', { class: 'mask' });
 $(document.body).append( this.mask );
 // предусмотрим закрытие окна
 this.close.on('click', this.closeWin);
 ModalWindow.windows.push(this);
}
```

Конструктору задаются: уникальный идентификатор окна и его размеры. В конструкторе создаётся шапка в виде слоя div с кнопкой закрытия окна, а также слой div для содержимого окна. Обе части вносятся в слой div, представляющий окно. Здесь же создаётся маска. Вся эта структура заносится в конец тела документа и изначально скрыта (для класса окна window и класса маски mask установлено CSS-свойство display: none). Ссылка на вновь созданное окно записывается в конец массива ModalWindow.windows. Этот массив играет роль своеобразного стека созданных окон – последнее окно в массиве (верхушка стека) является активным и находится поверх всех остальных окон. Кроме массива ModalWindow.windows, предусмотрены также следующие статические свойства класса ModalWindow:

```
// где нажали кнопку мыши относительно окна
ModalWindow.dx = 0; ModalWindow.dy = 0;
// открытые окна
ModalWindow.windows = [];
// номер слоя верхнего окна
ModalWindow.zndx = 9000;
```

Назначение представленных свойств разъясняется в комментариях и далее по мере их использования.

Все части окна и собственно само окно стилизуются с помощью классов. Окно и маска, как уже было сказано, изначально скрыты (display: none). Вот стили классов, используемых в представленном выше фрагменте кода (которые хранятся в файле ModalWindow.css):

```
/* Маска, затемняющая фон */

.mask {
 position: absolute;
 left: 0; top: 0;
 background-color: #000;
 display: none;
}

/* модальное окно */
.window {
 position: absolute;
 display: flex;
 flex-direction: column;
 background-color: #fff;
 padding: 0px;
 box-sizing: border-box;
 overflow: hidden;
}

/* шапка окна */
.top {
 margin: 0px; padding: 0px;
 box-sizing: border-box;
 height: 20px;
 background-color: gray;
 cursor: move;
}

/* закрыть окно */
.close {
 float:right;
 color:#fff;
 text-decoration:none; cursor:pointer;
}
.close:hover {color:#fff; text-decoration:underline;}

/* содержимое окна */
```

```
.content {
 flex-grow: 1;
 margin: 0px; padding: 0px;
 box-sizing: border-box;
 overflow: scroll;
}
```

Для объектов окон предусмотрены следующие методы, позволяющие открыть созданное окно (файл ModalWindow.js):

```
// развернуть маску
ModalWindow.prototype.showMask = function() {
 var maskHeight = $(document).height(),
     maskWidth = $(window).width();
 // Развернём маску на весь экран
 this.mask.css({
    width: maskWidth, height: maskHeight,
    // в текущем слое
    zIndex: ModalWindow.zndx++
 });
 this.mask.fadeTo("fast", 0.7);
}

// покажем окно
ModalWindow.prototype.showWin = function() {
 // маскируем
 this.showMask();
 // окно в центре
 var W = $(window).width(), H = $(window).height();
 this.modalWin.css({
    left: (W-this.width)/2, top: (H-this.height)/2,
    width: this.width, height: this.height,
    zIndex: ModalWindow.zndx++
 });
 this.modalWin.fadeIn(1000);
}
```

Номер слоя, в котором воспроизводится окно, хранится в переменной ModalWindow.zndx, которая тут же увеличивается на единицу – готовится для указания номера слоя следующего модального окна. Метод fadeTo() библиотеки jQuery, вызванный с параметром fast, позволяет «быстро» изменить прозрачность маски до уровня непрозрачности 0.7, а fadeIn() позволяет плавно показать окно (через 1000 мс).

В представленном ниже методе закрытия окна прежде всего получаем ссылку активного окна, которая хранится на вершине стека windows, а затем уничтожаем окно, маску и адекватно корректируем номер слоя zndx:

```
// закроем окно
ModalWindow.prototype.closeWin = function() {
 // получаем ссылку на активное окно
 var wnd = ModalWindow.windows.pop();
 wnd.modalWin.remove();
 wnd.mask.remove();
 ModalWindow.zndx -= 2;
}
```

Наконец, предусмотрим сервисный метод поиска окна по его идентификатору:

```
// поиск окна по идентификатору
function getWin(id) {
 for(var i = 0; i < ModalWindow.windows.length; i++ ) {
    var wnd = ModalWindow.windows[i];
    if( wnd.id == id ) return wnd;
 }
 return null;
}
```


4.2.5. Перемещение окон

В качестве дополнительной опции реализуем возможность привычного перетаскивания (drag and drop) окон мышью за шапку окна. С этой целью в конец метода showWin() внесём дополнительный код, предусматривающий реакции на нажатие и отпускание левой кнопки мыши:

```
// перемещаем за шапку
this.top.on('mousedown', onMouseDownHandler);
$(document.body).on('mouseup', onMouseUpHandler);
```


В коде функции onMouseDownHandler() захвата шапки окна мышью, в свою очередь, подключим обработчик перемещения мыши – mousemove (onMouseMoveHandler()):

```
function onMouseDownHandler(e) {
 var modalWin = $(this).offsetParent();
 var box = modalWin.offset();
 ModalWindow.dx = e.pageX - box.left;
 ModalWindow.dy = e.pageY - box.top;
 document.addEventListener('mousemove',
                          onMouseMoveHandler, false);

}
```

А также запомним смещение положения курсора мыши относительно левого верхнего угла окна в статических переменных ModalWindow.dx и ModalWindow.dy. Эти значения необходимы для вычисления смещения левого верхнего угла окна по изменившемся координатам курсора мыши в обработчике событий перемещения мыши:

```
function onMouseMoveHandler(e) {
 if( e.target.className != 'top' ) return;
 var modalWin = $(e.target).offsetParent();
 modalWin.offset({
    left: e.pageX - ModalWindow.dx,
    top: e.pageY - ModalWindow.dy
 });
}
```

Заметим, что первая строка тела представленной функции блокирует возможные перескакивания курсора за пределы шапки окна (в этом случае источник события будет иметь имя класса, отличное от top). Во второй строке осуществляется переход от шапки окна к самому окну, которое является родителем по отношению к шапке, с помощью метода offsetParent().

Наконец, когда пользователь отпускает кнопку мыши (событие mouseup) следует «сбросить» обработчик перемещения мыши:

```
function onMouseUpHandler() {
 document.removeEventListener('mousemove',
                              onMouseMoveHandler, false);
}
```

4.2.6. Стартовая страница

Конкретные значения входных параметров (начальной скорости, ускорения и времени) являются результатом их варьирования согласно одной из схем:

- от некоторого начального значения до некоторого конечного значения с некоторым шагом. Эти значения и являются исходными данными;
- перечисление значений варьируемого параметра, и в этом случае исходными данными являются все перечисленные значения.

Согласно первой схеме, должны варьироваться время и начальная скорость, а согласно второй – ускорение. Поля ввода должны быть предусмотрены для начального и конечного значений, а также шага варьируемых согласно первой схеме параметров, то есть для начальной скорости и времени. Для значений, которые перечисляются, в нашем случае это значения ускорения, можно

предусмотреть всего лишь одно поле, в котором эти значения будут вводиться, например, через запятую.

Таким образом, стартовая страница index.html должна содержать семь полей ввода (input). Кроме этого, стартовая страница, как уже было сказано, должна подключать все необходимые ресурсы со стилями и кодом JavaScript из других файлов:

```
<!DOCTYPE html>
<html>
<head>
<title>Пример</title>
<meta charset="utf-8">
<link rel="stylesheet" type="text/css"
href="./ModalWindow.css">
<link rel="stylesheet" type="text/css"
href="./app.css">
<script src="./jquery.js"></script>
<script src="../flot/jquery.flot.js"></script>
<script src="./ModalWindow.js"></script>
<script src="./frontend.js"></script>
</head>
<body>
<h4>Зависимость пройденного расстояния при равноуско-
ренном
 движении<br/> от времени, ускорения и начальной скоро-
сти</h4>
<table>
 <tr>
    <td>Ускорение:</td>
    <td class=LEFT><input id=a type=text
value="1,2,3,4,5"></td>
 </tr>
 <tr>
    <td></td>
    <td class=LEFT><i>(в этом поле через запятую пере-
числите различные значения ускорения)</i></td>
</table>
<table>
 <tr>
    <td colspan=6 class=LEFT>Время изменять</td>
 </tr>
 <tr>
    <td>от:</td>
    <td class=LEFT><input id=t0 type=text value=1></td>
    <td>до:</td>
```

```
    <td class=LEFT><input id=tk type=text value=2></td>
    <td>с шагом:</td>
    <td class=LEFT><input id=dt type=text value=1></td>
 </tr>
</table>
<table>
 <tr>
    <td colspan=6 class=LEFT>Начальную скорость изме-
нять</td>
 </tr>
 <tr>
    <td>от:</td>
    <td class=LEFT><input id=v0 type=text value=1></td>
    <td>до:</td>
    <td class=LEFT><input id=vk type=text value=3></td>
    <td>с шагом:</td>
    <td class=LEFT><input id=dv type=text value=1></td>
 </tr>
</table>
<table>
 <tr><td style="padding-left: 50px;">
    <input type=button value="Продолжить">
    <input type=button value="Описание задачи">
    <input type=button value="Об авторе (тит. лист)">
 </td></tr>
</table>
</body>
</html>
```

Оформление страницы определяется стилями, прописанными в файле app.css:

```
body, html {
 margin: 0px; padding: 0px;
 font-family: tahoma; font-size: 12px;
}
h4 { text-align: center; }
table {
 width: 500px; margin: 20px auto;
 background-color: WhiteSmoke;
}
td { padding: 2px; text-align: right; white-space:
nowrap;}
td.LEFT { padding-left: 0px; text-align: left; }
input { border-radius: 2px; border: 1px solid #ccc; }
input[type="text"] { width: 99%; }
```

```
input[type="button"] { float: left; margin-left: 15px;
}
.content table caption {
 text-align: left;
 font-weight: bold;
 padding: 5px;
}
.content table th {
 padding: 3px;
}
```

В результате, получим форму, которая изображена на рисунке 8.

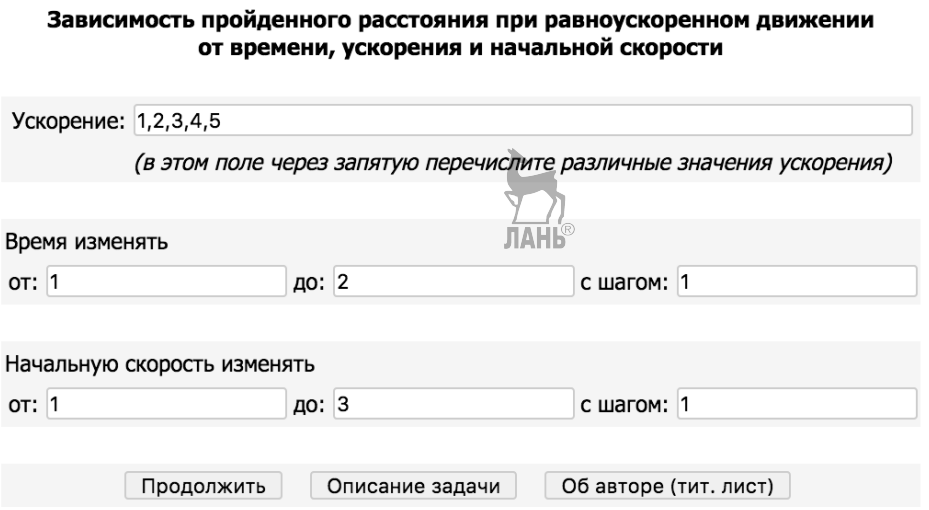


Рис. 8. Вид стартовой страницы

Полям ввода исходных данных даны уникальные идентификаторы: для ускорения – a, для начального значения времени – t0, для конечного значения времени – tk, для шага изменения времени – dt, для начального значения начальной скорости – v0, для конечного значения начальной скорости – vk, для шага изменения начальной скорости – dv.

Хотя идентификатор может быть любой последовательностью латинских букв и знаков подчеркивания, лучше формировать их согласно некоторым правилам. Например, можно использовать в идентификаторе поля имя параметра, для ввода которого это поле предусмотрено. Такой подход позволит определять назначение поля непосредственно по идентификатору. Так, в примере идентификатор v0 определяет поле, предназначенное для ввода начального значения начальной скорости, а vk – поле для ввода конечного значения этого параметра.

4.2.7. Ввод и контроль исходных данных, функция check()

В этом разделе переходим к описанию центральной части frontend разработки, содержимого файла frontend.js. Вспомним, что, согласно канонам программирования на jQuery, весь код следует разместить в теле следующей функции:

```
$(function () {
 // здесь можно быть уверенным, что все части документа
 // загружены и доступны для использования, в том числе
 // и поля ввода исходных данных
});
```

Прежде всего необходимо реализовать контроль и ввод исходных данных, которые набраны пользователем в полях ввода и являются строками символов. Ввод данных предполагает преобразование этих строк в числа с последующим их размещением в переменных, предусмотренных для этих целей. Для хранения начальных, конечных значений и значений шага начальной скорости и времени предусмотрим соответствующие переменные, а для заданных значений ускорения – массив:

```
var t0, dt, tk;
var v0, dv, vk, v;
var a = [];
```

Здесь также декларирована вспомогательная переменная v для хранения фиксированного значения начальной скорости. Поскольку в табличном виде и на графике можно представить результаты варьирования только двух параметров, третий должен быть фиксирован. В примере фиксируется скорость v. Декларация переменной v в данном месте гарантирует её видимость в любой точке кода, впрочем, как и остальных переменных, декларированных здесь.

Запускается приложение по нажатию кнопки «Продолжить». С помощью jQuery отыскиваем эту кнопку (CSS-селектор: input[type="button"][value="Продолжить"]) и «подвешиваем» на неё обработчик соответствующего события (click):

```
$('in-
put[type="button"][value="Продолжить"]').on('click',
 function() {
    if( !check() ) return;
    v = v0;
    send();
 });
```

Функция check() служит как для ввода, так и для проверки корректности вводимых данных. Если эта функция возвращает false, то это означает, что ввод и проверка не прошли и дальнейшие действия, то есть собственно расчёты, не выполняются. Вот эта функция:

```
function check() {
 var a_aux = $('#a').val().split(",");
 for(var i = 0; i < a_aux.length; i++) {
    var tmp = parseFloat(a_aux[i]);
    //если не удалось интерпретировать ввод как число,
    //то tmp содержит не число - Not a Number
    if( isNaN(tmp) ) { alert("Некорректный ввод");
$('#a').focus(); return false; }
    a[i] = tmp;
 }

 //аналогично проверяем другие поля ввода
 t0 = parseFloat( $('#t0').val() );
 if( isNaN(t0) ) { alert("Некорректный ввод");
$('#t0').focus(); return false; }
 dt = parseFloat( $('#dt').val() );
 if( isNaN(dt) ) { alert("Некорректный ввод");
$('#dt').focus(); return false; }
 tk = parseFloat( $('#tk').val() );
 if( isNaN(tk) ) { alert("Некорректный ввод");
$('#tk').focus(); return false; }

 v0 = parseFloat( $('#v0').val() );
 if( isNaN(t0) ) { alert("Некорректный ввод");
$('#v0').focus(); return false; }
 dv = parseFloat( $('#dv').val() );
 if( isNaN(dt) ) { alert("Некорректный ввод");
$('#dv').focus(); return false; }
 vk = parseFloat( $('#vk').val() );
 if( isNaN(tk) ) { alert("Некорректный ввод");
$('#vk').focus(); return false; }

 return true;
}
```

Ввод данных в этой функции устроен одинаково. Анализируются набранные пользователем символы в полях ввода и делается попытка представить их в виде чисел. Если это не удается, то на экран выдается предупреждающее сообщение с помощью функции alert(…) и на этом ввод прекращается – функция возвращает логическое значение ложь (return false), а так называемый фокус

ввода (мигающий курсор) перемещается в поле, содержащее ошибочные данные, одноимённым методом focus() этого поля.

Преобразование в число выполняется с помощью встроенной функции parseFloat(). Если эта функция по каким-то причинам не может преобразовать строку, заданную ей в качестве параметра, в число, то результатом её работы будет «не число» – Not a Number (согласно международному стандарту IEEE754). «Не число» можно диагностировать с помощью функции isNaN().

Несколько сложнее выглядит ввод значений ускорения, которые, как предполагается, вводятся пользователем через запятую в поле с идентификатором a. Результатом работы функции split(",") является массив подстрок, отделённых друг от друга запятыми в исходной строке, то есть в поле ввода значений ускорения. Далее делается попытка преобразовать элементы этого массива в числа (с помощью функции parseFloat()) и записать эти числа в предназначенные для них элементы массива a[i]. Если преобразование невозможно, то проверка заканчивается, фокус перемещается в поле, предназначенное для ввода ускорения, и функция возвращает значение false.


Более корректный контроль можно выполнить с помощью регулярных выражений. Такой контроль может являться дополнительным заданием.

Если функция check() прошла успешно, то есть её результат true, то далее фиксируется скорость (v = v0) и вызывается функция send(), в которой выполняются дальнейшие действия, связанные с обменом данными с сервером (с передачей исходных данных и получением результатов).

4.2.8. Обмен данных с сервером, функция send()

Функция send() передаёт данные серверу, который будет выполнять расчёт по заданной формуле. Для передачи данных с помощью AJAX воспользуемся функцией ajax() библиотеки jQuery, которая уже обсуждалась в соответствующем разделе данного пособия. Вызов этой функции сконфигурирован следующим образом:

```
function send() {
 $.ajax({
    method: "post",
    url: "http://localhost:3000",
    //данные для расчета
    data: {
       a: a, //ускорение
```

```
        t0: t0, tk: tk, dt: dt, //время
        v: v //начальная скорость
      },
      dataType: "json",
      success: showResult,
      error: error
  });
}
```

Исходные данные, указанные в виде объекта (значение свойства data), будут переданы в теле http-запроса как данные формы с именами полей, совпадающими с именами свойств этого объекта. Например, тело такого запроса может выглядеть так:

```
a%5B%5D=1&a%5B%5D=2&a%5B%5D=3&a%5B%5D=4&a%5B%5D=5&t0=1&
tk=10&dt=1&v=1
```

Здесь следует обратить внимание на передачу значений ускорения, которые первоначально заданы в виде массива a, а также учесть, что строка %5B%5D – это закодированные символы []. Таким образом, на стороне сервера можно использовать стандартные методы, которые используются для извлечения данных формы.

Ответ сервера будет представлен в формате JSON (это указано в свойстве dataType), преобразован и передан в качестве параметра функции showResult(). Ссылка на эту функцию задана в свойстве success. В случае возникновения ошибки управление будет предано функции error(), указанной в свойстве error. Основное назначение этой функции – открыть модальное окно с информацией об ошибке. Информация об ошибке передается этой функции в качестве параметра:

```
function error(data) {
 var modalWindow = new ModalWindow('error', 400, 200);
 modalWindow.showWin();
 modalWindow.content.append( $('<div/>', {
    html: 'statusText: ' + data.statusText + '<br/>' +
         'responseText: ' + data.responseText + '<br/>',
    css: {
      backgroundColor: 'yellow',
      color: 'red',
      padding: 3,
      height: '100%'
    }
 }) );
}
```

Свойство content модального окна – это ссылка на объект jQuery, представляющий div с содержимым окна. Добавить содержимое можно с помощью метода append(), который как раз и используется в приведённом выше коде.

4.2.9. Вывод результатов расчётов в табличном и графическом виде, функция showResult()

На http-запрос, инициированный функцией send(), сервер ответит результатами расчетов в виде JSON-строки следующего формата:

```
[
  {label: "a=a1", data: [[t1, s11], [t2, s12], …]},
  {label: "a=a2", data: [[t1, s21], [t2, s22], …]},
  …
]
```

Как это делает сервер, рассмотрим в разделе, посвящённом серверной части web-разработки.

Здесь a1, a2 и т. д. – это заданные пользователем значения ускорения, то есть содержимое массива a, который был передан в AJAX-запросе серверу. Значения времени t1, t2 и т. д. – это результат варьирования времени по формуле: от начального значения t0 до конечного tk с шагом dt. Значения t0, tk, dt также преданы серверу в AJAX-запросе. Наконец, s11, s12 и т. д. – это результаты расчётов пройденного расстояния для соответствующих значений ускорения и времени, а также скорости, которая была предана серверу в AJAX-запросе в параметре v.

Почему выбрана именно такая форма представления результатов? Ответ очень простой – такой формат данных без каких-либо изменений годится для рисования графиков с помощью упомянутой выше библиотеки flot.

Библиотека jQuery позаботится о том, чтобы предварительно преобразовать JSON-строку в полноценный массив, который и будет передан функции showResult(). Задачи функции showResult() – открыть модальное окно, сформировать таблицу, добавить таблицу к содержимому этого окна – реализуются следующим образом:

```
function showResult(data) {
 // открываем окно
 var modalWindow = new ModalWindow('w1', 600, 300);
 modalWindow.showWin();
 // формируем таблицу
 var tbl = "<table><caption>v=" + v +
                   "</caption><tr><th>a\\t</th>";
 // первая строка – значения времени
```

```
for(var j = 0; j < data[0].data.length; j++)
   tbl += '<th>' + data[0].data[j][0] + '</th>';
 tbl += '</tr>';
 for(var i = 0; i < data.length; i++) {
   tbl += '<tr><th>' + a[i] + '</th>';
   for(j = 0; j < data[i].data.length; j++)
      tbl += '<td>' + data[i].data[j][1] + '</td>';
   tbl += '</tr>';
 }
 tbl += "</table>";
 modalWindow.content.append( $(tbl) );
}
```

Для формирования таблицы потребовались два цикла – один по строкам таблицы, другой – по столбцам. Дополнительно потребовался цикл для формирования первой строки таблицы, содержащей значения времени, а также в начале каждой строки пришлось добавить клетку, содержащую значение ускорения. Пример сформированной таким образом таблицы представлен на рисунке 9.

v=1

a\t	1	2	3	4	5	6	7	8	9	10
1	1.5	4.0	7.5	12.0	17.5	24.0	31.5	40.0	49.5	60.0
2	2.0	6.0	12.0	20.0	30.0	42.0	56.0	72.0	90.0	110.0
3	2.5	8.0	16.5	28.0	42.5	60.0	80.5	104.0	130.5	160.0
4	3.0	10.0	21.0	36.0	55.0	78.0	105.0	136.0	171.0	210.0
5	3.5	12.0	25.5	44.0	67.5	96.0	129.5	168.0	211.5	260.0

Рис. 9. Пример таблицы, сформированной в функции showResult()

В таком простом варианте функция showResult() позволяет представить результаты расчётов только для одного фиксированного значения скорости. Очевидно, не хватает управляющих элементов для навигации по всем значениям скорости, предусмотренным правилом варьирования от v0 к vk с шагом dv. Кроме этого, в данном варианте отсутствует возможность перехода к графическому представлению результатов.

Для навигации вперёд (назад) по значениям скорости в контент окна перед таблицей с результатами расчётов добавим две кнопки:


```
var btn1 = $('<button/>', {text: '<<', disabled: 'dis-
abled'});
modalWindow.content.append( btn1 );
btn1.click( function() {
```

```
 v -= dv;
 send();
 if( v <= v0 ) {
    btn1.prop('disabled', true);
    btn2.prop('disabled', false);
 }
});
var btn2 = $('<button/>', {text: '>>'});
modalWindow.content.append( btn2 );
btn2.click( function() {
 v += dv;
 send();
 if( v >= vk ) {
    btn1.prop('disabled', false);
    btn2.prop('disabled', true);
 }
});
```

Реакции на нажатие этих кнопок устроены одинаково: увеличивается (уменьшается) значение скорости и управление передаётся функции send() для передачи серверу изменённых исходных данных. Кнопка вперёд (назад) блокируется, если мы достигаем крайних значений скорости.

Обновлённые результаты расчётов, полученные от сервера, вновь будут переданы функции showResult(). Однако теперь не надо вновь открывать модальное окно, а требуется всего лишь убрать старую таблицу и на её месте сформировать новую таблицу с пересчитанными результатами для нового значения скорости. Таким образом, прежде чем открывать новое окно, необходимо проверить: а может быть, окно уже открыто, и в этом случае требуется всего лишь убрать в этом окне старую таблицу. С учётом сказанного функция showResult() будет выглядеть несколько сложнее:

```
function showResult(data) {
 // ищем окно по его идентификатору
 var modalWindow = getWin('w1');
 if( !modalWindow ) { // окно не нашлось – открываем
новое
    modalWindow = new ModalWindow('w1', 600, 300);
    modalWindow.showWin();
    var btn1 = $('<button/>', {text: '<<', disabled:
'disabled'});
    modalWindow.content.append( btn1 );
    btn1.click( function() {
       v -= dv;
       send();
```

```
      if( v <= v0 ) {
        btn1.prop('disabled', true);
        btn2.prop('disabled', false);
      }
    });
    var btn2 = $('<button/>', {text: '>>'});
    modalWindow.content.append( btn2 );
    btn2.click( function() {
      v += dv;
      send();
      if( v >= vk ) {
        btn1.prop('disabled', false);
        btn2.prop('disabled', true);
      }
    });    }
  else { // окно нашлось – убираем старую таблицу
    $('table', modalWindow.content).remove();
  }
  // здесь формируем таблицу и добавляем её к
  // содержимому окна
…
}
```

Наконец, предусмотрим дополнительную кнопку для перехода к графическому представлению результатов расчётов. При активации этой кнопки сформируем новое модальное окно, контент (свойство content) которого будет служить местом отображения графики. Затем просто обратимся к функции библиотеки flot, а параметры для этой функции у нас уже готовы. Итак, добавляем третью кнопку:

```
var btn3 = $('<button/>', {text: 'график'});
modalWindow.content.append( btn3 );
btn3.click( function() {
 gWin = new ModalWindow('graphic', 700, 500);
 gWin.showWin();
 $.plot(gWin.content, data, { legend: { position: "nw"
} });
});
```

В качестве параметров функции plot() указываем место, где рисуется график – gWin.content, данные для построения графиков data (это один в один данные, переданные с сервера) и дополнительные опции в виде объекта, где меняем расположение легенды – в северо-западный угол. Возможный результат представлен на рисунке 10.

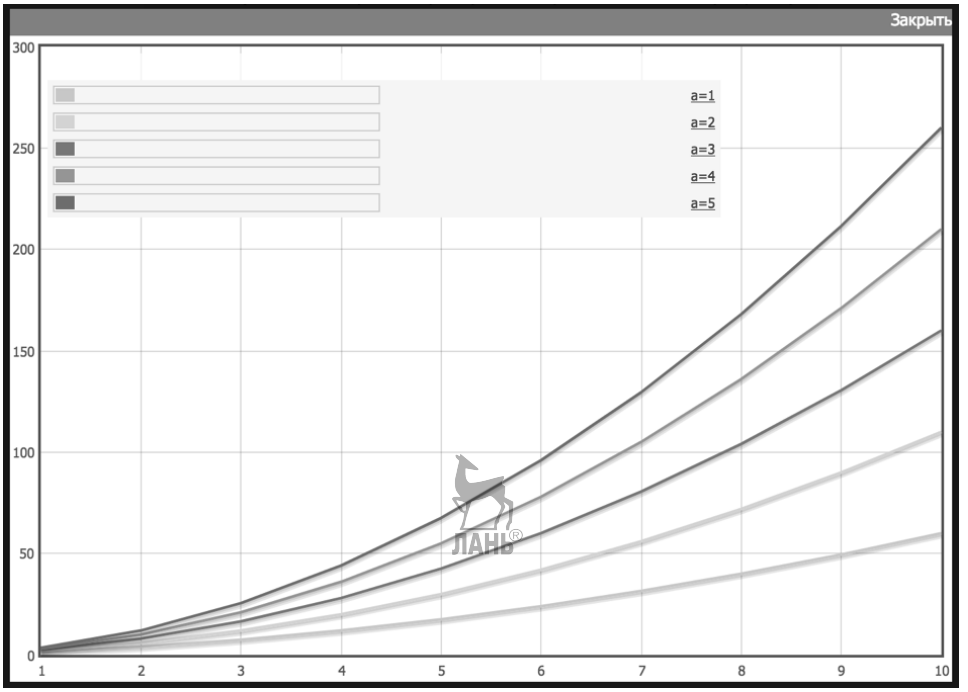


Рис. 10. Представление результатов расчётов в графическом виде

4.2.10. Серверная часть приложения – backend разработка

Серверная часть разработки (файл server.js) должна выполнять функции http-сервера, а также требуемые расчёты. При этом сервер должен понимать, что от него требует клиент: должен ли он выполнить расчёты и ответить результатами этих расчётов, либо он должен выдать в ответ запрашиваемый ресурс – содержимое запрашиваемого файла. Конечно, без «договорённости» с клиентом полного понимания достигнуть невозможно. Но эти соглашения на самом деле уже установлены в рамках frontend разработки. Запрос на расчёт не содержит имени файла. Действительно, вспомним, что это запрос по адресу – http://localhost:3000, в котором присутствует только указание сервера (только доменная часть с указанием номера порта). Напротив, если в url запроса присутствует имя файла, то, очевидно, в этом случае сервер должен ответить содержимым этого файла.

В таком поведении сервера есть одно фундаментальное отличие от общепринятого поведения – отвечать содержимым индексного файла (index.html), если имя файла не указано. В нашем случае имя стартового файла нестандартное (main.html) – инициировать приложение следует запросом http://localhost:3000/main.html.

В качестве дополнительного задания предлагается самостоятельно модифицировать приложение (клиентскую и серверную части) таким образом, чтобы загрузка приложения происходила стандартно – по запросу http://localhost:3000.

Http-сервер. Для реализации http-сервера потребуются дополнительный модуль mime (требует установки) и встроенные модули http и fs. Итак, проверяем наличие имени в url (req.url.length > 1) и формируем ответ:

```
var fs = require('fs');
var http = require('http');
var mime = require('mime');

http.createServer(function(req, res) {
 // статический сервер
 if( req.url.length > 1 ) {
    var mime_type = mime.getType( req.url );
    fs.readFile('.' + req.url, function(err, data) {
       if(err) {
          if(err.code == 'ENOENT') {
             res.writeHead(404, {
                'Content-Type': 'text/html; charset=utf-8'
             });
             res.write('Not Found');
             res.end();
          }
          else {
             res.statusCode = 500;
             res.end('Internal Server Error');
          }
          console.log(err);
       }
       else {
          res.writeHead(200, {
             'Content-Type': mime_type
          });
          res.write(data);
          res.end();
       }
    });
    return;
 }
```

```
 // расчёт
 …
}
```

В представленном фрагменте кода для читателя не должно быть откровений, поскольку подобный статический web-сервер подробно обсуждался выше в разделах, посвящённых node.js (п. 3.3.2). Здесь же обратим внимание читателя на ряд несущественных отличий. Корневым каталогом сервера в данном варианте является текущий каталог, именно из текущего каталога сервер пытается прочитать указанный в запросе файл. Тип mime содержимого файла определяется с помощью метода getType(), а не lookup(), как это делалось в прежнем варианте для устаревших версий модуля mime.

Расчётная часть. Если имя файла в запросе не указано явно (это место помечено в представленном фрагменте кода многоточием), то это AJAX-запрос на выполнение расчётов. В этом случае данные для расчёта передаются в теле запроса. Для того чтобы «вытащить» эти данные из тела запроса, можно воспользоваться дополнительным модулем querystring, который требуется предварительно установить и затем подключить инструкцией:

```
var qs = require('querystring');
```

Тело запроса поступает от клиента порциями, которые будем накапливать в буфере, реализованном в виде обычного массива chunk_arr. Каждую поступившую порцию (специальный объект типа Buffer) помещаем в отдельный элемент массива. В конце передачи – объединяем отдельные элементы:

```
var chunk_arr = [];
req.on('data', function(chunk) {
 chunk_arr.push(chunk);
});
req.on('end', function() {
 res.statusCode = 200;
 res.setHeader('Content-Type', 'text/html; charset=utf-
8');
 var body = Buffer.concat(chunk_arr).toString();

 var data = calc( qs.parse(body) );

 res.end( JSON.stringify(data) );
}
```

В приведённом фрагменте обрабатываются следующие события: поступление очередной порции – событие data, конец передачи – событие end. Для объединения отдельных порций использован статический метод concat() объек-

та Buffer. Результат объединения преобразуется в обычную строку (toString()). Метод parse() модуля querystring представляет переданные от клиента исходные данные для расчётов в виде объекта, например:

```
{
   'a[]': [ '1', '2', '3', '4', '5' ],
   t0: '1',
   tk: '20',
   dt: '1',
   v: '1'
}
```

Здесь следует обратить внимание на имя параметра 'a[]', который в исходном AJAX-запросе на клиентской стороне передавался в виде массива ускорений a.

Исходные данные передаются в качестве параметра функции calc(), которая, собственно, и реализует расчётную часть. Результаты расчётов возвращаются в виде объекта, преобразуются в JSON-формат и отправляются клиенту:

```
res.end( JSON.stringify(data) );
```

Результаты расчётов, как уже было сказано, формируются в виде, пригодном для воспроизведения в графическом виде, библиотекой flot в теле функции calc():

```
function calc(params) {
 var
   a = params['a[]'],
   t0 = Number(params.t0),
   tk = Number(params.tk),
   dt = Number(params.dt),
   v = params.v;

 var data = [];
 for(var i = 0; i < a.length; i++) {
   var dataPoints = [];
   for(var t = t0; t <= tk; t += dt)
     dataPoints.push(
       [ t, Number(v * t + a[i] * t * t /
2).toFixed(1) ]
     );
   data.push({
     label: 'a=' + a[i],
     data: dataPoints
   });
 }
```

```
 return data;
}
```

В данной функции реализованы два цикла: один цикл по значениям ускорения, а другой, вложенный в первый, – по значениям времени. Результаты расчётов в итоге получаются в виде массива, структура которого уже обсуждалась в предыдущем пункте.

Функции description и about author. Теперь осталось объявить две функции, связанные с кнопками «Описание задачи» и «Об авторе». Они должны загружать в модальные окна соответственно страницу *task_description.html* и страницу *about_author.html*. Обе функции вызывают появление неизменяемых по размеру окон в центре экрана без строки статуса внизу:

```
function description() {
 showModalDialog("task_description.html", '', "cen-
ter:yes;help:no;resizable:no;status:no;");
}

function about_author() {
 showModalDialog("about_me.html", '', "cen-
ter:yes;help:no;resizable:no;status:no;");
}
```

Отметим, что метод showModalDialog() объекта window работает не во всех браузерах и результатом его работы является всплывающее окно, которое может блокироваться в целях безопасности, о чём уже говорилось выше.

 Предлагается самостоятельно реализовать вывод информации о задаче и об авторе с помощью класса ModalWindow.

5. Дополнительное задание

Реакции на активацию кнопок «Описание задачи» и «Об авторе (тит. лист)» реализуются студентом самостоятельно. При этом должны быть воспроизведены модальные окна, в которых должна быть представлена соответствующая информация: описание задачи и сведения об авторе. Соответствующий HTML и CSS-код разрабатываются студентами самостоятельно. Описание задачи должно содержать формулу для расчёта и расшифровку параметров с указанием допустимых диапазонов, в которых эти параметры могут изменяться. Запрещается использовать рисунки с изображением формул, подготовленных с помощью специальных программных средств (MS Word и др.), формула должна быть создана средствами HTML без привлечения тэга img.

Разработка должна обеспечивать ввод не только варьируемых параметров, но и постоянных.

Разработка должна обеспечить контроль всех вводимых параметров. В случае выхода параметров за пределы допустимых диапазонов должны выдаваться предупреждающие сообщения.

В качестве дополнительного задания также предлагается самостоятельно модифицировать приложение (клиентскую и серверную части) таким образом, чтобы загрузка приложения происходила по запросу http://localhost:3000.

Наконец, отдельным заданием может являться реализация чата на основе web-сокетов. Реализация чата подробно обсуждалась в разделе 3.3. Требуется воспроизвести представленный там код и возможно выполнить модернизацию согласно приведённым там же пожеланиям.

Заключение

В пособии показано, что фронтенд – это технология разработки приложений на стороне клиента, включающая: HTML, формирующий содержание страницы, например «заголовок», «параграф», «список», «элемент списка»; CSS, определяющий отображение элементов на странице, например «после первого параграфа отступ в 18 пикселей» или «весь текст в теге body должен быть темно-серым и написан шрифтом Courier New»; JavaScript, обеспечивающий реакцию на некоторые события в браузере, придающий динамизм и интерактивность.

Это направление хорошо изучено и является широко используемым инструментом web-программистов, в то время как backend разработка web-приложений или программирование на стороне сервера является насущной необходимостью. Как все работает на сервере – этим определяется ряд положительных моментов для разработчиков – здесь можно использовать любые инст-

рументы и технологии, доступные на сервере, который, по сути, является просто компьютером, настроенным для ответов на сообщения.

Наиболее популярным подходом в бекэнде является использование Node.js как платформы, которая позволяет выполнять JavaScript вне браузера и предоставляет API для операций ввода-вывода. По сути, Node превращает JS в язык общего назначения, позволяя писать на нем практически любые приложения, и здесь же можно использовать различные системы управления базами данных.

В учебном пособии представлен материал, позволяющий освоить backend разработку web-приложений как программирование на стороне сервера на платформе node.js. Ярко выраженная практическая направленность всего материала с предоставлением примера разработки приложения для выполнения простейших расчетов на серверной стороне и динамического формирования содержимого HTML-страниц позволяет не только изучить и освоить теоритический материал, но и получить начальные практические навыки применения технологии backend разработки web-приложений на платформе node.js.

Приложение

Файл main.html

```
<!DOCTYPE html>
<html>
<head>
<title>Пример</title>
<meta charset="utf-8">
<link rel="stylesheet" type="text/css"
href="./ModalWindow.css">
<link rel="stylesheet" type="text/css"
href="./app.css">
<script src="./jquery.js"></script>
<!-- <script src="./jquery.canvasjs.min.js"></script> -
->
<script src="../flot/jquery.flot.js"></script>
<script src="./ModalWindow.js"></script>
<script src="./frontend.js"></script>
</head>
<body>
<h4>Зависимость пройденного расстояния при равноуско-
ренном движении<br/> от времени, ускорения и начальной
скорости</h4>
<table>
 <tr>
    <td>Ускорение:</td>
    <td class=LEFT><input id=a type=text
value="1,2,3,4,5"></td>
 </tr>
 <tr>
    <td></td>
    <td class=LEFT><i>(в этом поле через запятую пере-
числите различные значения ускорения)</i></td>
</table>
<table>
 <tr>
    <td colspan=6 class=LEFT>Время изменять</td>
 </tr>
 <tr>
    <td>от:</td>
    <td class=LEFT><input id=t0 type=text value=1></td>
    <td>до:</td>
    <td class=LEFT><input id=tk type=text value=2></td>
    <td>с шагом:</td>
```

```
    <td class=LEFT><input id=dt type=text value=1></td>
 </tr>
</table>
<table>
 <tr>
    <td colspan=6 class=LEFT>Начальную скорость
изменять</td>
 </tr>
 <tr>
    <td>от:</td>
    <td class=LEFT><input id=v0 type=text value=1></td>
    <td>до:</td>
    <td class=LEFT><input id=vk type=text value=3></td>
    <td>с шагом:</td>
    <td class=LEFT><input id=dv type=text value=1></td>
 </tr>
</table>
<table>
 <tr><td style="padding-left: 50px;">
    <input type=button value="Продолжить">
    <input type=button value="Описание задачи">
    <input type=button value="Об авторе (тит. лист)">
 </td></tr>
</table>
</body>
</html>
```

Файл app.css

```
body, html {
  margin: 0px; padding: 0px; font-family: tahoma; font-
size: 12px;
}
h4 { text-align: center; }
table {
  width: 500px; margin: 20px auto; background-color:
WhiteSmoke;
}
td { padding: 2px; text-align: right; white-space:
nowrap;}
td.LEFT { padding-left: 0px; text-align: left; }
input { border-radius: 2px; border: 1px solid #ccc; }
input[type="text"] { width: 99%; }
input[type="button"] { float: left; margin-left: 15px;
}
.content table caption {
```

```
  text-align: left;
  font-weight: bold;
  padding: 5px;
}
.content table th {
  padding: 3px;
}
```

Файл ModalWindow.js

```
function ModalWindow(id, w, h) {
 this.id = id;
 this.width = w; this.height = h;
 // шапка окна
 this.top = $('<div/>', { class: 'top' });
 this.close = $('<a/>', { class: 'close', text:
'Закрыть' });
 this.top.append( this.close );
 // содержимое окна
 this.content = $('<div/>', { id: id, class: 'content'
});
 // окно
 this.modalWin = $('<div/>', { class: 'window' });
 this.modalWin.append(this.top);
 this.modalWin.append(this.content);
 // добавим окно в конец тела документа
 $(document.body).append( this.modalWin );
 // маска
 this.mask = $('<div/>', { class: 'mask' });
 $(document.body).append( this.mask );
 // предусмотрим закрытие окна
 this.close.on('click', this.closeWin);
 ModalWindow.windows.push(this);
}

// где нажали кнопку мыши относительно окна
ModalWindow.dx = 0;
ModalWindow.dy = 0;
// открытые окна
ModalWindow.windows = [];
// номер слоя верхнего окна
ModalWindow.zndx = 9000;

ModalWindow.prototype.showMask = function() {
 var maskHeight = $(document).height(), maskWidth =
$(window).width();
```

```
 // Развернём маску на весь экран
 this.mask.css({ width: maskWidth, height: maskHeight,
zIndex: ModalWindow.zndx++ });
 this.mask.fadeTo("fast", 0.7); //fadeIn(1000);
}

ModalWindow.prototype.showWin = function() {
 // маскируем
 this.showMask();
 // покажем окно в центре
 var W = $(window).width(), H = $(window).height();
 this.modalWin.css({
    left: (W-this.width)/2, top: (H-this.height)/2,
    width: this.width, height: this.height, zIndex: Mo-
dalWindow.zndx++
 });
 this.modalWin.fadeIn(1000);
 // перемещаем за шапку
 this.top.on('mousedown', onMouseDownHandler);
 $(document.body).on('mouseup', onMouseUpHandler);
}

ModalWindow.prototype.closeWin = function() {
 var wnd = ModalWindow.windows.pop();
 wnd.modalWin.remove();
 wnd.mask.remove();
 ModalWindow.zndx -= 2;
}

//  поиск окна по идентификатору
function getWin(id) {
 for(var i = 0; i < ModalWindow.windows.length; i++ ) {
    var wnd = ModalWindow.windows[i];
    if( wnd.id == id ) return wnd;
 }
 return null;
}

function onMouseDownHandler(e) {
 var modalWin = $(this).offsetParent();
 var box = modalWin.offset();
 ModalWindow.dx = e.pageX - box.left; ModalWindow.dy =
e.pageY - box.top;
 document.addEventListener('mousemove', onMouseMoveHan-
dler, false);
```

```
}

function onMouseUpHandler() {
 document.removeEventListener('mousemove', onMouseMove-
Handler, false);
}

function onMouseMoveHandler(e) {
 if( e.target.className != 'top' ) return;
 var modalWin = $(e.target).offsetParent();
 modalWin.offset({
    left: e.pageX - ModalWindow.dx,
    top: e.pageY - ModalWindow.dy
 });
}
```

Файл ModalWindow.css

```
/* Маска, затемняющая фон */
.mask {
 position: absolute;
 left: 0; top: 0;
 background-color: #000;
 display: none;
}

/* модальное окно */
.window {
 position: absolute;
 display: flex;
 flex-direction: column;
 background-color: #fff;
 padding: 0px;
 box-sizing: border-box;
 overflow: hidden;
}

/* шапка окна */
.top {
 margin: 0px;
 padding: 0px;
 box-sizing: border-box;
 height: 20px;
 background-color: gray;
 cursor: move;
}
```

```
/* закрыть окно */
.close { float:right; color:#fff; text-decoration:none;
cursor:pointer; }
.close:hover { color:#fff; text-decoration:underline }

/* содержимое окна */
.content {
 flex-grow: 1;
 margin: 0px; padding: 0px;
 box-sizing: border-box;
 overflow: scroll;
}
```

Файл frontend.js

```
$(function () {
   // исходные данные
   var a = [];
   var t0, dt, tk;
   var v0, dv, vk, v;

   $('in-
put[type="button"][value="Продолжить"]').on('click',
      function() {
         if( !check() ) return;
         v = v0;
         send();
      }
   );

   // проверяет корректность введенных данных
   // если результат отрицательный возвращает false
   function check() {
      var a_aux = $('#a').val().split(",");
      for(var i = 0; i < a_aux.length; i++) {
         var tmp = parseFloat(a_aux[i]);
         // если не удалось интерпретировать ввод как
число,
         // то tmp содержит не число - Not a Number
         if( isNaN(tmp) ) {
            alert("Некорректный ввод"); $('#a').focus();
            return false;
         }
         a[i] = tmp;
      }
```

```
    // аналогично проверяем другие поля ввода
    // время
    t0 = parseFloat( $('#t0').val() );
    if( isNaN(t0) ) {
      alert("Некорректный ввод"); $('#t0').focus();
      return false;
    }
    dt = parseFloat( $('#dt').val() );
    if( isNaN(dt) ) {
      alert("Некорректный ввод"); $('#dt').focus();
      return false;
    }
    tk = parseFloat( $('#tk').val() );
    if( isNaN(tk) ) {
      alert("Некорректный ввод"); $('#tk').focus();
      return false;
    }

    // скорость
    v0 = parseFloat( $('#v0').val() );
    if( isNaN(t0) ) {
      alert("Некорректный ввод"); $('#v0').focus();
      return false;
    }
    dv = parseFloat( $('#dv').val() );
    if( isNaN(dt) ) {
      alert("Некорректный ввод"); $('#dv').focus();
      return false;
    }
    vk = parseFloat( $('#vk').val() );
    if( isNaN(tk) ) {
      alert("Некорректный ввод"); $('#vk').focus();
      return false;
    }

    return true;
}

//отправляем исходные данные серверу
function send() {
    $.ajax({
      method: "post",
      url: "http://localhost:3000",
      //данные для расчета
```

```
        data: {
          a: a, //ускорение
          t0: t0, tk: tk, dt: dt, //время
          v: v //начальная скорость
        },
        dataType: "json",
        success: showResult,
        error: error
      });
    }

    // формирование таблицы с результатами
    function showResult(data) {
      var modalWindow = getWin('w1');
      if( !modalWindow ) { // окно не было открыто
        modalWindow = new ModalWindow('w1', 600, 300);
        modalWindow.showWin();
        var btn1 = $('<button/>',
          {text: '<<', disabled: 'disabled'});
        modalWindow.content.append( btn1 );
        btn1.click( function() {
          v -= dv;
          send();
          if( v <= v0 ) {
            btn1.prop('disabled', true);
            btn2.prop('disabled', false);
          }
        });
        var btn2 = $('<button/>', {text: '>>'});
        modalWindow.content.append( btn2 );
        btn2.click( function() {
          v += dv;
          send();
          if( v >= vk ) {
            btn1.prop('disabled', false);
            btn2.prop('disabled', true);
          }
        });
        var btn3 = $('<button/>', {text: 'график'});
        modalWindow.content.append( btn3 );
        btn3.click( function() {
          gWin = new ModalWindow('graphic', 700, 500);
          gWin.showWin();
          $.plot(gWin.content, data, { position: "nw"
});
```

```
        });
      }
      else {
        $('table', modalWindow.content).remove();
      }

      // формируем таблицу
      var tbl = "<table><caption>v=" + v +
                    "</caption><tr><th>a\\t</th>";
      for(var j = 0; j < data[0].data.length; j++)
        tbl += '<th>' + data[0].data[j][0] + '</th>';
      tbl += '</tr>';
      for(var i = 0; i < data.length; i++) {
        tbl += '<tr><th>' + a[i] + '</th>';
        for(j = 0; j < data[i].data.length; j++)
          tbl += '<td>' + data[i].data[j][1] + '</td>';
        tbl += '</tr>';
      }
      tbl += "</table>";
      modalWindow.content.append( $(tbl) );
    }

    // обработка ошибок при обмене с сервером
    function error(data) {
      var modalWindow = new ModalWindow('error', 400,
  200);
      modalWindow.showWin();
      modalWindow.content.append( $('<div/>', {
        html: 'statusText: ' + data.statusText +
  '<br/>' +
          'responseText: ' + data.responseText +
  '<br/>',
        css: {
          backgroundColor: 'yellow',
          color: 'red',
          padding: 3,
          height: '100%'
        } }));
    }
  });
```

Файл server.js

```
var fs = require('fs');
var http = require('http');
var mime = require('mime');
```

```
var qs = require('querystring');

http.createServer(function(req, res) {
 // статический сервер
 if( req.url.length > 1 ) {
    var mime_type = mime.getType( req.url );
    fs.readFile('.' + req.url, function(err, data) {
       if(err) {
          if(err.code == 'ENOENT') {
             res.writeHead(404, {
                'Content-Type': 'text/html; charset=utf-8'
             });
             res.write('Not Found');
             res.end();
          }
          else {
             res.statusCode = 500;
             res.end('Internal Server Error');
          }
          console.log(err);
       }
       else {
          res.writeHead(200, {
             'Content-Type': mime_type
          });
          res.write(data);
          res.end();
       }
    });
    return;
 }
 // расчёт
 var chunk_arr = [];
 req.on('data', function(chunk) {
    chunk_arr.push(chunk);
 });
 req.on('end', function() {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html;
charset=utf-8');
    var body = Buffer.concat(chunk_arr).toString();

    var data = calc( qs.parse(body) );

    res.end(
```

```
      JSON.stringify(data)
    );
  });
}).listen(3000);
console.log('3000');

// расчёт
function calc(params) {
 var
    a = params['a[]'],
    t0 = Number(params.t0), tk = Number(params.tk), dt =
Number(params.dt),
    v = params.v;

 var data = [];
 for(var i = 0; i < a.length; i++) {
    var dataPoints = [];
    for(var t = t0; t <= tk; t += dt)
      // расчитываем расстояние
      dataPoints.push( [ t, Number(v * t + a[i] * t * t
/ 2).toFixed(1) ] );
    data.push({
      label: 'a=' + a[i],
      data: dataPoints
    });
 }
 return data;
}
```

Литература

1. *Флэнаган*, *Д.* JavaScript. Подробное руководство : пер. с англ. – СПб. : Символ-Плюс, 2012. – 1080 с.
2. *Хэррон*, *Д.* Node.js Разработка серверных веб-приложений на JavaScript : пер. с англ. – М. : ДМК-Пресс, 2012. – 144 с.
3. *Кантелон*, *М.* Node.js в действии / М. Кантелон, М. Хартер, Т. Головайчук, Н. Райли. – СПб. : Питер, 2014. – 548 с.
4. *Васильев*, *Н. П.* Технология публикации расчётов web-обозревателей : учеб. пособие / Н. П. Васильев, М. Л. Шилкина. – СПб. : СПбГЛТА, 2008. – 48 с.
5. Официальный сайт node.js [Электронный ресурс]. – Режим доступа: https://nodejs.org/.
6. Оригинальная документация по node.js на английском языке [Электронный ресурс]. – Режим доступа: https://nodejs.org/api/.
7. Документация по менеджеру пакетов npm [Электронный ресурс]. – Режим доступа: https://docs.npmjs.com/.
8. Описание node.js на русском языке [Электронный ресурс]. – Режим доступа: https://js-node.ru/.
9. Руководство по Node.js [Электронный ресурс]. – Режим доступа: https://metanit.com/web/nodejs/.
10. Node.js : учеб. пособие [Электронный ресурс]. – Режим доступа: http://www.w3ii.com/ru/nodejs/default.html.

Анатолий Моисеевич ЗАЯЦ,
Николай Павлович ВАСИЛЬЕВ

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ
ВВЕДЕНИЕ В FRONTEND И BACKEND РАЗРАБОТКУ НА JAVASCRIPT И NODE.JS
Учебное пособие
Издание второе, стереотипное

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
Тел./факс: (812) 336-25-09, 412-92-72
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 29.03.20.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 9,75. Тираж 30 экз.

Заказ № 298-20.

Отпечатано в полном соответствии с качеством
предоставленного оригинал-макета в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.