ООО ЦОТ "БелХард Групп"
Центр обучающих технологий

Михалькевич А.В.

HTML5

методическое пособие

Минск 2014

УДК 004.415.53
ББК 32.973.2-018.2я7

Рецензент:

Об авторе:

Михалькевич Александр Викторович, программист, преподаватель учебных курсов по PHP, HTML5, NODE.JS.

А.В.Михалькевич
HTML5

А.В.Михалькевич, Закрытое акционерное общество «БелХард Групп» - Мн., 2014. – 166с.

ISBN

- Данное учебное пособие является не просто введением в HTML5, а полноценным учебным курсом, в котором рассматриваются все основные аспекты HTML5. Включая платформу Node.js, серверный JavaScript

Пособие рекомендовано к использованию слушателям курсов ЦОТ ЗАО "БелХард Групп".

УДК 004.415.53
ББК 32.973.2-018.2я7

ISBN

Содержание:

HTML5 – это HTML, CSS3, javaScript и API javaScript.

Три основных принципа, которыми руководствовались разработчики стандартов HTML5:

1. **Не рвать паутину**. Это означает, что стандарт не должен изменять правила, и считать устаревшим совершенно нормальные web-страницы. Новые стандарты не должны перечеркивать устаревшие.
2. **Асфальтировать тропинки**. Тропинка представляет собой неровный, но протоптанный путь, позволяющий добраться из одной точки в другую. Асфальтирование тропинок несет следующую выгоду: используются устоявшиеся методы, поддерживаемые браузерами, не вошедшие в спецификацию. Web-разработчики отдают предпочтение неряшливому, но практичному решению, которое работает всегда и во всех браузерах, вместо стандартизированных технологий, которые не поддерживаются большей частью браузеров.
3. **Быть практичным**. Это простой принцип: все изменения должны служить практической цели. И чем более трудоемкое изменение, тем большей должна быть от нее отдача.

1. Селекторы

Тэги

В качестве селектора может выступать любой тег HTML для которого определяются правила форматирования, такие как: цвет, фон, размер и т.д. Правила задаются в следующем виде.

Селекторы тэгов. Листинг 1.1

```
Тег { свойство1: значение; свойство2: значение; ... }
```

Классы

Классы применяют, когда необходимо определить стиль для индивидуального элемента веб-страницы или задать разные стили для одного тега. При использовании совместно с тегами синтаксис для классов будет следующий.

Селектор класса. Листинг 1.2

```
 Тег.Имя_класса { свойство1: значение; свойство2: значение; ... }
 .имя_класса {свойства...}
```

Идентификаторы

Идентификатор (называемый также «ID селектор») определяет уникальное имя элемента, которое используется для изменения его стиля и обращения к нему через скрипты.

Селектор идентификатора. Листинг 1.3

```
 #Имя идентификатора {
свойство1: значение; свойство2: значение; ...
}
```

Контекстные селекторы

При создании веб-страницы часто приходится вкладывать одни теги внутрь других. Чтобы стили для этих тегов использовались корректно, помогут селекторы, которые работают только в определенном контексте. Например, задать стиль для тега <b> только когда он располагается внутри контейнера <p>. Таким образом можно одновременно установить стиль для отдельного тега, а также для тега, который находится внутри другого.

Контекстный селектор состоит из простых селекторов разделенных пробелом. Так, для селектора тега синтаксис будет следующий.

Контекстный селектор. Листинг 1.4

```
Тег1 Тег2 { ... }
P B { ... }
```

Универсальный селектор

Иногда требуется установить одновременно один стиль для всех элементов веб-страницы, например, задать шрифт или начертание текста. В этом случае поможет универсальный селектор, который соответствует любому элементу веб-страницы.

Для обозначения универсального селектора применяется символ звездочки (*) и в общем случае синтаксис будет следующий.

Универсальный селектор. Листинг 1.5

```
* { Описание правил стиля }
```

Простой селектор атрибута

Устанавливает стиль для элемента, если задан специфичный атрибут тега. Его значение в данном случае не важно. Синтаксис применения такого селектора следующий.

Селекторы. Листинг 1.6

```
[атрибут] { Описание правил стиля }
Селектор[атрибут] { Описание правил стиля }
```

Стиль применяется к тем тегам, внутри которых добавлен указанный атрибут. Пробел между именем селектора и квадратными скобками не допускается.

Атрибут со значением

Устанавливает стиль для элемента в том случае, если задано определенное значение специфичного атрибута. Синтаксис применения следующий.

Атрибуты со значением. Листинг 1.7

```
[атрибут="значение"] { Описание правил стиля }
```

```
Селектор[атрибут="значение"] { Описание правил стиля }
```

Пример селектора атрибута со значением. Листинг 1.8

```
input[type="text"] {
  background: #eee;
}
```

Значение атрибута начинается с определенного текста
Устанавливает стиль для элемента в том случае, если значение атрибута тега начинается с указанного текста. Синтаксис применения следующий.

Селекторы атрибутов, значения которых начинается с определенного текста. Листинг 1.9

```
[атрибут^="значение"] { Описание правил стиля }
Селектор[атрибут^="значение"] { Описание правил стиля }
```

В первом случае стиль применяется ко всем элементам, у которых значение атрибута начинаются с указанного текста. А во втором — только к определенным селекторам. Использование кавычек не обязательно, но только если значение содержит латинские буквы.

Предположим, что на сайте требуется разделить стиль обычных и внешних ссылок — ссылки, которые ведут на другие сайты. Чтобы не добавлять к тегу <a> новый класс, воспользуемся селекторами атрибутов. Внешние ссылки характеризуются добавлением к адресу протокола, например, для доступа к гипертекстовым документам используется протокол HTTP. Поэтому внешние ссылки начинаются с ключевого слова http://, его и добавляем к селектору A.

Пример селекторов атрибутов, значение которых начинается с определенного текста. Листинг 1.10

```
A[href^="http://"] {
 font-weight: bold
}
```

Значение атрибута оканчивается определенным текстом
Устанавливает стиль для элемента в том случае, если значение атрибута оканчивается указанным текстом. Синтаксис применения следующий.

Селекторы атрибутов, значение которых оканчивается определенным текстом. Листинг 1.11

```
[атрибут$="значение"] { Описание правил стиля }
Селектор[атрибут$="значение"] { Описание правил стиля }
```

Пример селектора атрибутов, значение которых оканчивается определенным текстом. Листинг 1.12

```
A[href$=".ru"] { /* Если ссылка заканчивается на .ru */
 background: url(images/ru.png) no-repeat 0 6px;
 padding-left: 12px;
```

```
}
A[href$=".com"] { /* Если ссылка заканчивается на .com */
 background: url(images/com.png) no-repeat 0 6px;
 padding-left: 12px;
}
```

Значение атрибута содержит указанный текст

Возможны варианты, когда стиль следует применить к тегу с определенным атрибутом, при этом частью его значения является некоторый текст. При этом точно не известно, в каком месте значения включен данный текст — в начале, середине или конце. В подобном случае следует использовать такой синтаксис.

Селекторы атрибутов с содержанием указанного текста. Листинг 1.13

```
[атрибут*="значение"] { Описание правил стиля }
Селектор[атрибут*="значение"] { Описание правил стиля }
```

Пример селектора атрибутов с содержанием указанного текста. Листинг 1.14

```
[href*="obmenka"] {
 background: yellow; /* Желтый цвет фона */
}
```

Одно из нескольких значений

Некоторые значения атрибутов могут перечисляться через пробел, например имена классов. Чтобы задать стиль при наличии в списке требуемого значения применяется следующий синтаксис.

Селекторы атрибутов одного или нескольких значений. Листинг 1.15

```
[атрибут~="значение"] { Описание правил стиля }
Селектор[атрибут~="значение"] { Описание правил стиля }
```

Пример селектора атрибутов одного или нескольких значений. Листинг 1.16

```
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-
8">
  <title>Блок</title>
  <style type="text/css">
   [class~="block"] h3 { color: green; }
  </style>
 </head>
 <body>
  <div class="block tag">
   <h3>Заголовок</h3>
  </div>
 </body>
</html>
```

Дефис в значении атрибута

В именах идентификаторов и классов разрешено использовать символ дефиса (-), что позволяет создавать значащие значения атрибутов id и class. Для

изменения стиля элементов, в значении которых применяется дефис, следует воспользоваться следующим синтаксисом.

Селектор с дефисом в значении атрибута. Листинг 1.17

```
[атрибут|="значение"] { Описание правил стиля }
Селектор[атрибут|="значение"] { Описание правил стиля }
```

Стиль применяется к элементам, у которых атрибут начинается с указанного значения или с фрагмента значения, после которого идет дефис:

Пример селектора с дефисом в значении атрибута. Листинг 1.18

```
<html>
 <head>
  <title>Блок</title>
  <style type="text/css">
   DIV[class|="block"] {
    background: #306589;
    color: #acdb4c;
    padding: 5px;
   }
   DIV[class|="block"] A {
    color: #fff;
   }
  </style>
 </head>
 <body>
  <div class="block-menu-therm">
  </div>
 </body>
</html>
```

В данном примере имя класса задано как block-menu-therm, поэтому в стилях используется конструкция **|="block"**, поскольку значение начинается именно с этого слова и в значении встречаются дефисы.

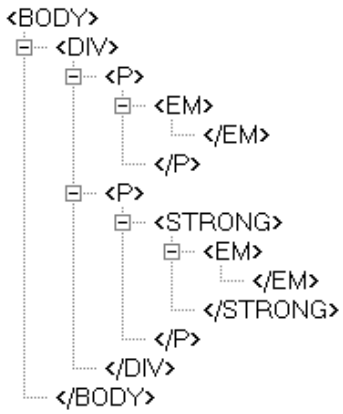
Все перечисленные методы можно комбинировать между собой, определяя стиль для элементов, которые содержат два и более атрибута. В подобных случаях квадратные скобки идут подряд.

Комбинирование селекторов с атрибутами. Листинг 1.19

```
[атрибут1="значение1"][атрибут2="значение2"] { Описание правил
стиля }
Селектор[атрибут1="значение1"][атрибут2="значение2"] { Описание
правил стиля}
```

Дочерние селекторы

Дочерним называется элемент, который непосредственно располагается внутри родительского элемента. Чтобы лучше понять отношения между элементами документа, рассмотрим следующую DOM-структуру:



Здесь дочерним элементом по отношению к тегу выступает тег <p>. Вместе с тем тег **не является** дочерним для тега , поскольку он расположен в контейнере <p>.

Дочерние селекторы. Листинг 1.20

```
Селектор 1 > Селектор 2 { Описание правил стиля }
```

Соседние селекторы

Соседними называются элементы веб-страницы, когда они следуют непосредственно друг за другом в коде документа. Рассмотрим несколько примеров отношения элементов.

Для управления стилем соседних элементов используется символ плюса (+), который устанавливается между двумя селекторами. Общий синтаксис следующий.

Соседние селекторы. Листинг 1.21

```
Селектор 1 + Селектор 2 { Описание правил стиля }
```

Пробелы вокруг плюса не обязательны, стиль при такой записи применяется к Селектору2, но только в том случае, если он является соседним для Селектора1 и следует сразу после него.

Использование соседних селекторов. Листинг 1.22

```
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-
8">
  <title>Соседние селекторы</title>
  <style type="text/css">
   B + I {
    color: red; /* Красный цвет текста */
   }
  </style>
 </head>
 <body>
   <p>Lorem <b>ipsum </b> dolor sit amet, <i>consectetuer</i>
adipiscing elit.</p>
   <p>Lorem ipsum dolor sit amet, <i>consectetuer</i> adipiscing
```

```
elit.</p>
 </body>
</html>
```

В данном примере происходит изменение цвета текста для содержимого контейнера <i>, когда он располагается сразу после контейнера <b>. В первом абзаце такая ситуация реализована, поэтому слово «consectetuer» в браузере отображается красным цветом. Во втором абзаце, хотя и присутствует тег <i>, но по соседству никакого тега <b> нет, так что стиль к этому контейнеру не применяется.

2. Псевдо-классы и псевдоэлементы CSS3

CSS псевдо-классы - это элементы CSS, они отличаются от id и class, и не предназначены для комбинирования заданий или простых селекторов. Можно использовать псевдо-классы для выбора элементов на основе их атрибутов, состояния и относительное положение в документе. Синтаксис для псевдо-классов используется с двоеточием :, после двоеточия пишется имя псевдо-класса. Многие уже знакомы с псевдо-классами ссылок из CSS2.1:

Псевдоклассы ссылок CSS2. Листинг 2.1

```
:link
:visited
:hover
:active
:focus
```

В CSS3 появились новые псевдоклассы:

Новые псевдоклассы CSS3. Листинг 2.2

```
:root
:only-child
:empty
:nth-child(n)
:nth-last-child(n)
:first-of-type
:last-of-type
:only-of-type
:nth-of-type(n)
:nth-last-of-type(n)
:first-child
:last-child
```

Чтобы назначить цель для конкретного элемента, например для элемента e, вам необходимо добавить этот элемент e в начало синтаксиса псевдо-класса, перед двоеточием, вот так:

Псевдокласс для определенного тэга. Листинг 2.3

```
 p:pseudo-class {}
```

Если нужно, можно использовать псевдо-класс вместе с id и class:

Комбинирование классов и идентификаторов в псевдоклассах. Листинг 2.4

```
#id:pseudo-class {} .class:pseudo-class {}
```

Некоторые из псевдо-классов CSS3 понимают нумерацию элементов в дереве документа. Укажите положение элемента в виде числового значения в скобках (n):

Использование в псевдоклассах нумерации. Листинг 2.5

```
:pseudo-class(n) {}
:pseudo-class(2) {}//присвоение стиля только для второго элемента
```

Так же можно назначить порядок чисел, например, каждый пятый элемент, для этого нужно указать значение (5n):

Псевдокласс для каждого n-ного элememнта. Листинг 2.6

```
:pseudo-class(5n) {}
:pseudo-class(2n) {} //четность
```

Кроме того, можно смещать порядок вывода:

Псевдокласс для каждого n-ного элememнта со смещением на 1. Листинг 2.7

```
:pseudo-class(5n+1) {}
```

Новые селекторы понимают расположение элементов по порядку в дереве документа, если указать ключевые слова, odd (четные) и even (нечетные). Например, если вам нужно добавить стиль, для не четных элементов, можно использовать следующую команду:

Использование odd и even в псевдоклассах. Листинг 2.8

```
:pseudo-class(odd) {}
```

:root

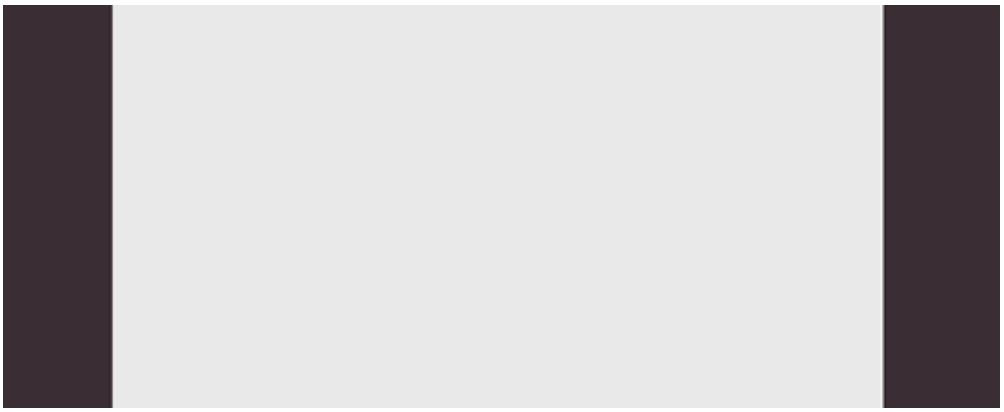
Этот псевдо-класс предназначен для корня HTML (он применяется раньше body). Рассмотрим основную разметку страницы HTML5:

HTML код. Листинг 2.9

```
<html>
<head>
<style>
:root{
    background-color: rgb(56,41,48);
}
body {
    background-color: rgba(255, 255, 255, .9);
    margin: 0 auto;
    min-height: 350px;
    width: 700px;
```

```
}
</style>
    <title>Знакомство с селекторами CSS3</title>
    </head>

<body>
</body>
</html>
```

Получим следующее:



:only-child

Псевдо-класс, который является единственным дочерним в корне. Можно использовать :only-child для любого элемента страницы HTML, добавляя в псевдо-класс синтаксис элементов e.

:empty

Псевдо-класс, который не имеет дочерних или текстовых пустых элементов, таких как: .

:nth-child(n)

Этот псевдо-класс относится к дочернему элементу по отношению к его позиции внутри корневого элемента. Например список комментарии в блогах, может выглядеть красиво с разным цветом фона.

Выбор четных дочерних элементов. Листинг 2.10

```
li:nth-child(even){ background-color: rgba(242, 224, 131, .5); }
```

:nth-last-child(n)

Этот псевдо-класс работает точно так же как и :nth-child(n), но относится к дочерним элементам начиная с последнего вхождения.

:first-of-type

Задает правила стилей для первого элемента в списке дочерних элементов своего родителя. К примеру, добавление :first-of-type к селектору TD устанавливает стиль для всех первых ячеек, поскольку родителем для тега <td> выступает тег <tr>.

Синтаксис :first-of-type. Листинг 2.11

```
элемент:first-of-type { ... }
```

В отличии от :first-child применяет стиль, если элемент первый наследник и (+) он находиться на первом месте после родителя (ему не преграждают путь другие элементы). А :first-of-type - применяет стиль к элементу, кто бы перед ним не стоял.

Пример применения :first-of-type. Листинг 2.12

```
<head>
 <meta charset="utf-8">
 <title>first-of-type</title>
 <style>
  table {
   border-collapse: collapse; /* Убираем двойные границы */
   width: 100%; /* Ширина таблицы */
   border-spacing: 0; /* Расстояние между ячеек */
  }
  td {
   border: 1px solid #6A3E14; /* Параметры рамки */
   padding: 4px; /* Поля в ячейках */
  }
  tr:first-of-type {
   background: #808990; /* Цвет фона */
   color: #fff; /* Цвет текст */
  }
  td:first-of-type {
   background: #CFD6D3; /* Цвет фона */
  }
 </style>
</head>
<body>
 <table>
  <tr>

<td> </td><td>1998</td><td>1999</td><td>2000</td><td>2001</td
>
   <td>2002</td><td>2003</td>
  </tr>
  <tr>
   <td>Нефть</td>

<td>3</td><td>22</td><td>34</td><td>62</td><td>74</td><td>57</td>
  </tr>
  <tr>
   <td>Золото</td>

<td>4</td><td>13</td><td>69</td><td>72</td><td>56</td><td>47</td>
  </tr>
  <tr>
   <td>Дерево</td>
```

```
<td>4</td><td>7</td><td>73</td><td>79</td><td>34</td><td>86</td>
   </tr>
  </table>
 </body>
</html>
```

:last-of-type

Псевдокласс :last-of-type задает правила стилей для последнего элемента в списке дочерних элементов своего родителя. К примеру, добавление :last-of-type к селектору li устанавливает стиль только для последнего пункта списка, при этом не распространяется на остальные пункты.

Использование псевдокласса :last-of-type. Листинг 2.13

```
 <html>
 <head>
  <meta charset="utf-8">
  <title>last-of-type</title>
  <style>
   p:last-of-type:after {
    content: " \25C4"; /* Добавляем символ в конце текста */
    color: #c00000; /* Цвет символа */
   }
  </style>
 </head>
 <body>
  <p>Этот старинный скандинавский напиток пришел к нам из древних
времен и воспет во многих песнях. Теперь вы самостоятельно можете
приготовить его и насладиться чудесным вкусом и ароматом
легендарного нектара.</p>
  <p>...</p>
  <p>Осталось добавить хлива и хрольва, чтобы напиток был готов.
Подавать горячим.</p>
 </body>
</html>
```

:only-of-type

Применяется к дочернему элементу указанного типа, только если он единственный у родителя. Аналогичен использованию :first-of-type:last-of-type или :nth-of-type(1):nth-last-of-type(1).

Пример использования :only-of-type. Листинг 2.14

```
<html>
 <head>
  <meta charset="utf-8">
  <title>only-of-type</title>
  <style>
   img:only-of-type {
    border: 2px solid red;
   }
  </style>
 </head>
```

```
 <body>
  <p><img src="images/figure.jpg" alt="">
     <img src="images/figure.jpg" alt=""></p>
  <p><img src="images/figure.jpg" alt=""></p>
 </body>
</html>
```

:nth-of-type(n)

Псевдокласс :nth-of-type используется для добавления стиля к элементам указанного типа на основе нумерации в дереве элементов.

Синтаксис :nth-of-type(n). Листинг 2.15

```
элемент:nth-of-type(odd | even | <число> | <выражение>) {...}
```

Значения:

odd

Все нечетные номера элементов.

even

Все четные номера элементов.

число

Порядковый номер указанного элемента. Нумерация начинается с 1, это будет первый элемент в списке.

выражение

Задается в виде an+b, где a и b целые числа, а n — счетчик, который автоматически принимает значение 0, 1, 2...

Если a равно нулю, то оно не пишется и запись сокращается до b. Если b равно нулю, то оно также не указывается и выражение записывается в форме an. a и b могут быть отрицательными числами, в этом случае знак плюс меняется на минус, например: 5n-1.

За счет использования отрицательных значений a и b некоторые результаты могут также получиться отрицательными или равными нулю. Однако на элементы оказывают влияние только положительные значения из-за того, что нумерация элементов начинается с 1.

В таблице приведены некоторые возможные выражения и ключевые слова, а также указано, какие номера элементов будут задействованы.

Значение	Номера элементов	Описание
1	1	Первый элемент.
5	5	Пятый элемент.
2n	2, 4, 6, 8, 10	Все четные элементы, аналог значения even.
2n+1	1, 3, 5, 7, 9	Все нечетные элементы, аналог значения odd.
3n+2	2, 5, 8, 11, 14	—
-n+3	3, 2, 1	—

5n-2	3, 8, 13, 18, 23	—
even	2, 4, 6, 8, 10	Все четные элементы.
odd	1, 3, 5, 7, 9	Все нечетные элементы.

Пример использования :nth-of-type(). Листинг 2.16

```
<html>
 <head>
  <meta charset="utf-8">
  <title>nth-of-type</title>
  <style>
   img:nth-of-type(2n+1) { float: left; }
   img:nth-of-type(2n) { float: right; }
  </style>
 </head>
 <body>
  <p><img src="images/left.gif" alt="">
   <img src="images/right.gif" alt=""></p>
   <h1>Исторический турнир</h1>
 </body>
</html>
```

:nth-last-of-type(n)

Псевдокласс :nth-last-of-type используется для добавления стиля к элементам указанного типа на основе нумерации в дереве элементов. В отличие от псевдокласса :nth-of-type отсчет ведется не от первого элемента, а от последнего.

:first-child

Псевдокласс :first-child применяет стилевое оформление к первому дочернему элементу своего родителя.

:last-child

Псевдокласс :last-child применяет стилевое оформление к последнему дочернему элементу своего родителя.

Псевдоэлементы

Далее рассмотрим псевдоэлементы CSS. Псевдоэлементы позволяют задать стиль логических элементов, не определенных в дереве элементов документа, а также генерировать содержимое, которого нет в исходном коде текста. Например, объектная модель документа не предлагает удобного механизма для доступа к первому символу текста, поэтому псевдоэлементы позволяют изменить стиль недоступного иным образом элемента.

Синтаксис использования псевдоэлементов следующий.

Синтаксис псевдоэлементов. Листинг 2.17

```
Селектор:Псевдоэлемент { Описание правил стиля }
```

:aftrer
Псевдоэлемент, который используется для вывода желаемого текста после содержимого элемента, к которому он добавляется. Псевдоэлемент ::after работает совместно со свойством content.

Использование псевдоэлемента :after. Листинг 2.18

```
элемент::after { content: "текст" }
элемент:after { content: "текст" }
```

Для :after характерны следующие особенности.

1. При добавлении :after к блочному элементу, значение свойства display может быть только: block, inline, none, marker. Все остальные значения будут трактоваться как block.
2. При добавлении :after к встроенному элементу, display ограничен значениями inline и none. Все остальные будут восприниматься как inline.

:before
Псевдоэлемент :before применяется для отображения желаемого контента до содержимого элемента, к которому он добавляется. В остальном аналогичен псевдоэлементу ::before

:first-letter
Псевдоэлемент :first-letter определяет стиль первого символа в тексте элемента, к которому добавляется. К этому псевдоэлементу могут применяться только стилевые свойства, связанные со свойствами шрифта, полями, отступами, границами, цветом и фоном.

:first-line
Псевдоэлемент :first-line задает стиль первой строки форматированного текста. Длина этой строки зависит от многих факторов, таких как используемый шрифт, размер окна браузера, ширина блока, языка и т.д. В правилах стиля допустимо использовать только свойства, относящиеся к шрифту, изменению цвета текста и фона.

Использоваение first-line. Листинг 2.19

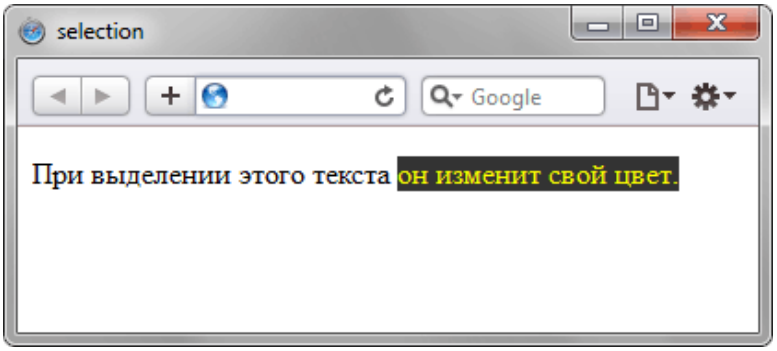
```
h1 + p::first-line { ... } // Псевдоэлемент :first-line задает
стиль первой строки форматированного текста. Длина этой строки
зависит от многих факторов, таких как используемый шрифт, размер
окна браузера, ширина блока, языка и т.д. В правилах стиля
допустимо использовать только свойства, относящиеся к шрифту,
изменению цвета текста и фона.
```

:selection

Псевдоэлемент :selection применяет стиль к выделенному пользователем тексту. В правилах стилей допускается использовать следующие свойства: color, background и background-color.

Использование псевдоэлемента ::selection. Листинг 2.20

```
<html>
 <head>
  <meta charset="utf-8">
  <title>selection</title>
  <style>
   p::selection {
    color: #ff0; /* Цвет текста */
    background: #000; /* Цвет фона */
   }
  </style>
 </head>
 <body>
  <p>При выделении этого текста он изменит свой цвет.</p>
 </body>
</html>
```



Трюки

Более утонченный таргентинг. Листинг 2.21

```
#page > * { ... } // Находим все дочерние элементы #page
```

Поддержка новых шрифтов

Поддержка новых шрифтов. Листинг 2.22

```
@font-face {
  font-family: 'LeagueGothic';
  src: url(LeagueGothic.otf);
}

@font-face {
  font-family: 'Droid Sans';
  src: url(Droid_Sans.ttf);
}

header {
  font-family: 'LeagueGothic';
```

```
}
```

3. Семантические элементы.

При работе с HTML документами, верстальщики приходят к выводу, что элемент - это основное средство для структурирования web-страниц. Это контейнер общего назначения, с помощью которого можно структурировать любую часть страницы. Недостатком данного элемента является то, что он не предлагает никакой информации о содержимого. Встретив элемент мы (или браузер, поисковый робот и т.п) понимаем, что нашли отдельный блок страницы, но не знаем назначение этого блока. Для исправления такой ситуации HTML5 предлагает заменить некоторые элементы более описательными и семантическими тэгами. Эти семантические тэги действую точно так же, как элемент : они группируют содержимое в блок, но кроме этого добавляют семантический смысл содержимому.

Итак, появились семантические тэги, а стилистические, такие как:

- basefont
- big
- center
- font
- s
- strike
- tt
- u

исчезли. Вышеперечисленных тэгов в спецификации HTML5 не существует. Кроме того, было решено отказаться от использования фрэймов. Таким образом, исчезли следующие элементы:

- frame
- frameset
- noframes

Прежде чем начинать обзор семантических элементов и атрибутов, необходимо разобраться с проблемой отображения в IE: проблема заключается в том, что IE отказывается понимать форматирование таблицы стилей к элементам, которые не распознает. К счастью, эта проблема решается с помощью небольшого трюка: браузер IE можно обмануть и заставить его распознавать незнакомые элементы, объявив их с помощью javaScript. Например, в следующем коде, мы заставляем браузер IE понимать и распознавать стили к элементу <header>.

Решение проблемы с IE. Листинг 3.1

```
<script>
 document.createElement('header');
```

```
</script>
```

Или, вместо того, чтобы самому разрабатывать такой код для всех элементов, можно воспользоваться уже готовым сценарием.

В данном примере, мы, сперва, делаем проверку на то, что у пользователя IE версии ниже 9. Если это действительно так, то подгружается удаленный скрипт.

Универсальное решение проблемы с IE. Листинг 3.2

```
<!--[if lt IE 9]>
 <script scr='http://html5shim.googlecode.com/svn/trunk/html5.js'>
 </script>
<![endif]-->
```

Семантическая разметка предназначена для смыслового описания контента (используйте её для борьбы с болезнью дивита).

Семантические элементы блочного уровня.

Элемент	Описание
<article>	Представляет собой единицу информации, которую можно рассматривать, как статью — блок самодостаточного содержимого, наподобие газетной статьи, сообщения на форуме или записи в блоге (исключая второстепенную информацию, такую как комментарии или ссылка на автора).
<aside>	Представляет цельный фрагмент содержимого, отделенный от основного содержимого страницы.
<figure> и <figcaption>	Представляет рисунок. Элемент <fipcaption> содержит текст подписи к рисунку, а элементы <figure> - содержит сам элемент <figcaption> и <img>. Цель такой комбинации указать связь между рисунком и его подписью.
<footer>	Представляет нижний колонтитул страницы. Это небольшой фрагмент содержимого, который может включать сообщение об авторских правах и несколько ссылок. Может, также использоваться в статьях в качестве дополнительного содержимого к статье.
<header>	Представляет верхний заголовок, включающий стандартный заголовок HTML и дополнительного содержимое, которое может состоять из логотипа, слогана. Может также использоваться в заголовках для статей.
<hgroup>	Представляет расширенный заголовок, содержащий два и более пронумерованных элементов заголовка (<h1>, <h2>, <h3>, <h4>, <h5>, <h6>) и ничего больше. Основное назначение этого элемента держать заголовок и подзаголовок вместе.

<nav>	Представляет значительный набор ссылок на странице, указывающий на тематические разделы текущей страницы или на другие страницы web-сайта.
<section>	Представляет собой обособленный раздел. Является универсальным контейнером, использование которого регламентируется лишь одним правилом: содержимое должно начинаться с заголовка. Данный элемент следует использовать, когда не подходят никакие другие элементы.

Новые атрибуты ссылок

Новые атрибуты ссылок. Листинг 3.3

```
<link rel="alternate" type="application/rss+xml"
href="http://myblog.com/feed" />
<link rel="icon" href="/favicon.ico" />
<link rel="pingback" href="http://myblog.com/xmlrpc.php">
<link rel="prefetch" href="http://myblog.com/main.php">
...
<a rel="archives" href="http://myblog.com/archives">old posts</a>
<a rel="external" href="http://notmysite.com">tutorial</a>
<a rel="license" href="http://www.apache.org/licenses/LICENSE-
2.0">license</a>
<a rel="nofollow" href="http://notmysite.com/sample">wannabe</a>
<a rel="tag" href="http://myblog.com/category/games">games
posts</a>
```

Пользовательские атрибуты с префиксом «data-»

Пользовательские атрибуты, начинающиеся с префикса «**data-**» игнорируются обработчиками документов HTML5. Основное их предназначение — передача данных.

Так, например, вместо этого кода:

Событие onClick Листинг 3.4

```
<a href=”#” onclick=”window.open(page.html, width=300px,
height=400px)”>Перейди по ссылке</a>
```

Предлагается следующее решение:

HTML-код ссылки. Листинг 3.5

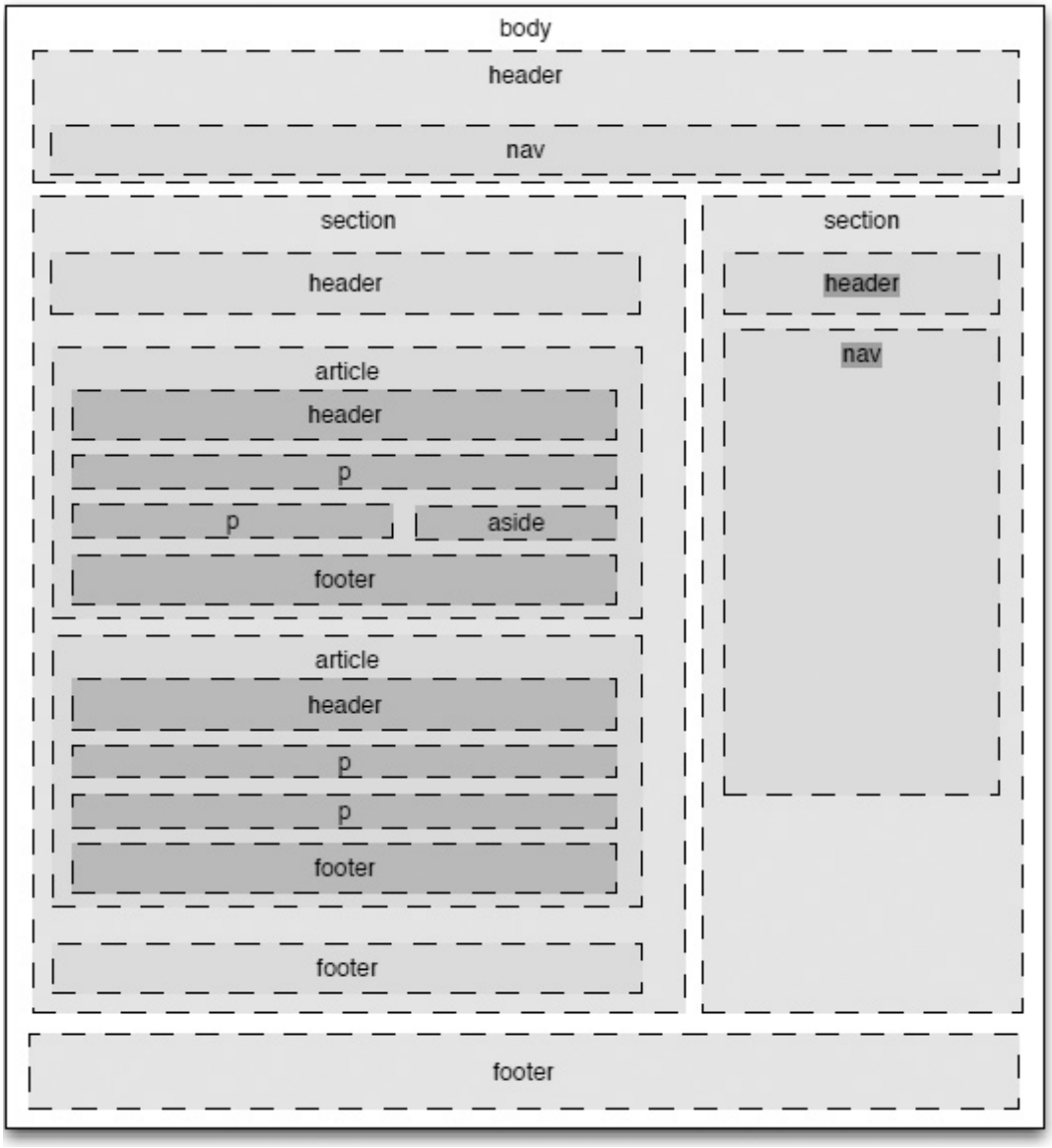
```
<a class=”popup” data-width=300px data-height=400px
title=”название ссылки” href=”page.php”>Перейди по ссылке</a>
```

Измененное событие click. Листинг 3.6

```
$(function(){
 $(“.popup”).click(function(event){
  event.preventDefault();
  var href = $(this).attr(“href”);
  var width = $(this).attr(“data-width”);
```

```
  var height = $(this).attr("data-height");
  var popup = window.open(href, "popup", "height=" + height + ",
   "width=" + width);
 });
});
```

Рассмотрим структуру документа с использованием семантических элементов.



Код типичной HTML5 разметки. Листинг 3.7

```
<!DOCTYPE html>
  <html>
   <head>
    <title>Структура документа HTML5</title>
   </head>
```

```
  <body>
   <header>
    <hgroup>
     <h1>Заголовок страницы</h1>
     <h2>Подзаголовок</h2>
    </hgroup>
   </header>
   <nav>...</nav>
   <section>
      <article>
         <header>Название статьи</header>
         <section>текст статьи</section>
      </article>
   </section>
   <aside>...</aside>
   <footer>...</footer>
  </body>
</html>
```

Особенности использования семантических элементов:

Вместо следует использовать <section>.
В <header> или <footer> могут быть использованы <nav>.
В статьях <acrticle> могут быть использованы <header>, <footer>, <section>.
Тэг <aside> - для информации, дополняющей основной контент.

Семантика текстового уровня

Семантика текстового уровня намного сложнее по причине огромного количества типов содержимого. Чтобы избежать избыточного количество тегов, в спецификации HTML5 предложено два подхода: 1) добавлено небольшие количество новых тегов и 2) что более важно, HTML5 поддерживает отдельный стандарт микроданных, который предоставляет разработчикам расширенный способ определения по желанию разработчика любой тип информации.

Семантические элементы строчного типа

Элемент	Описание
<time>	Применяется в качестве оболочки для даты. От элемента <time> требуется выполнения двух функций: во-первых он должен указывать местонахождение значения даты или времени в разметки; во-вторых, он должен представлять заключенное в него значение даты и времени в формате, понимаемом любой программой. Формат даты и времени имеет такой шаблон: ГГГГ:ММ:ДД: `Новый магазин откроется <time> 2013:07:20 </time>` Но пользователю дату вполне допустимо показать в любом

другом формате, при условии представления ее значения в машиночитаемом формате в атрибуте datetime:

```
Новый магазин откроется <time datetime='2013:07:20'>
20 июля </time>
```

Такие же правила применяются и для значения времени, которое необходимо представлять в следующем формате: ЧЧ:ММ+00:00 (т.е. двузначное значение часа, двузначное значение минут, после знака «плюс» указывается смещение часового пояса от всемирного координированного времени). Указание смещения часового пояса является обязательным. Узнать смещение часового пояса для конкретного региона можно на сайте http://en.wikipedia.org/wiki/Time_zone.

```
Магазин открывается с <time datetime='08:30+3:0'>
8:30 </time>
```

Время и дату можно совмещать:

```
Магазин откроется <time
datetime='2013:07:20T08:30+3:0'> 20 июля в 8:30
</time>
```

Элемент <time> поддерживает атрибут putdate. Он применяется в том случае, когда указываемая дата совпадает с датой публикации статьи.

```
Опубликовано <time datetime='putdate'> 21 марта 2013
года </time>
```

Элемент <time> является информационным: с ним не связано никакое форматирование. Это значит, что его можно использовать для любого браузера, не заботясь о совместимости (при условии, что проблема отображения в IE решена).

<output> Предназначен для вывода результатов сценария javaScript. Его задача — резервирование места подстановки, в которое будет выводится результат вычислений.
Элемент <output> можно сделать еще более осмысленным, добавив в его атрибут for:

```
результат вычислений: <output id=”result”
for=”meters”></output>
```

В действительности, этот атрибут ничего не дает, кроме как информации о том, откуда элемент <output> получает свои

данные. Но если нужно будет страницу отредактировать кому-нибудь другому, а не разработчику-хозяину, эти атрибут помогут программисту понять, как работает скрипт.

<mark> Предназначен для выделения цветом текста. Взятый в элемент <mark> текст выделяется желтым цветом. Цвет фона и текста можно заменить в стилях.

С помощью элемента <mark> можно также помечать важное содержимое или ключевые слова (так как это делают поисковые движки, выделяя текст совпадающий с текстом запроса, в результатах поиска).

Стандарт микроданных
Для интеллектуального анализа данных на web-странице существуют следующие стандарты:

Стандарт ARIA (Accessible Rich Internet Aplications)
Позволяет предоставить дополнительную информацию для программ чтения экрана. Среди прочих, в нем вводится атрибут role, который указывает предназначение данного элемента. Например объяснить программе предназначение элемента можно следующим образом:

Определение баннера с помощью стандарта ARIA. Листинг 3.8

```
<div role="banner"> ... </div>
```

Мнения разработчиков по поводу ценности вложения времени и сил в изучение данного стандарта неоднозначны. Дело в том, что данный стандарт всё еще находится в стадии разработки, и многие его возможности сходны с возможностями HTML5. Но многие современные программы чтения экрана поддерживают стандрат ARIA, а стандарт HTML5 еще нет.

Микроформаты
Это несвязный набор соглашений, которые позволяют страницам представлять структурированную информацию, не требуя сложных форматов. Существует около сотни микроформатов, из которых широко используется более десятка.

Рассмотрим микроформат **hCard**. Для использования этого формата, необходимо создать оболочку с классом vcard. Внутри элемента с данным классом, можно использовать специальные классы микроформата:

- класс fn – для отображения имени контактного лица
- класс photo – для элемента <img>
- класс title – статус или должность контактного лица
- класс org – для отображения названия компании

- класс url — ссылка на страницу
- класс tel – телефон контактного лица
- класс locality – для отображения населенного пункта контактного лица
- класс region – область или регион
- класс street-address – для отображения улицы
- класс post-code – почтовый код
  класс country-name – страна

Микроформат hCard. Листинг 3.9

```
<div role="vcard">
 <div class="fn">Михалькевич Александр Викторович</div>
 <img class="photo" src="my_photo.jpg" alt="Михалькевич Александр
Викторович">
 <div class="title">Web-мастер</div>
 <div class="org">Индивидуальный предприниматель</div>
 <a class="url"
href="http://mikhalkevich.colony.by">mikhalkevich.colony.by</a>
 <div class="tel">8(029)763-93-82</div>
</div>
```

Мы рассмотрели, как создать микроформат с самого начала, но его так же можно внедрять в уже готовую разметку HTML-документа
К сожалению, пока еще ни один браузер не поддерживает микроформаты, но существуют подключаемые модули и сценарии, которые могут дать им эту возможность.
Что еще интереснее, это возможность подключения данного формата через библиотеку jQuery. Для этого необходимо лишь добавить в страницу ссылку на плагин oomph.min.js:

Добавление плагина обработчика микроформата hCard. Листинг 3.10

```
<script src="js/jquery-1.9.1.min.js"></script>
<script src="js/oomph.min.js"></script>
```

Теперь все посетители сайта получат возможность доступа к помеченным микроформатами данных страницы, независимо от используемого им браузера.

Вторым по популярности является микроформат **hCalendar**. Событие помещается в элемент с названием класса vevent. Внутри этого элемента должно быть как минимум две единицы информации: дата начала события, которая помечается классом dtstart и описание события, которое помечается классом summary.

Микроформат hCalendar. Листинг 3.11

```
<div class="vevent">
 <h2 class="summary">Web-мастер Михалькевич Александр
Викторович</h2>
```

```
 <p> <span class="dtstart" title="2013-12-22"> 22 декабря 2013
года</span> приглашаю отметить годовщину Конца Света</p>
</div>
```

Микроданные

Формат микроданных — это еще один подход к решению задачи задания элементам семантического смысла. Данный микроформат зародился как часть спецификации HTML5, но в последствии был выделен в отдельный развивающийся стандарт.

Самая большая разница между микроданными и микроформатом заключается в использовании в микроданных атрибута **itemprop**, а не class, как это было в микроформате.

Чтобы создать блок микроданных, нужно создать блок с атрибутами itemscope и itemtype.

Использование микроданных. Листинг 3.12

```
<div itemscope itemtype="http://data-vocabulary.org/Person">
 <div itemprop="address" itemscope itemtype="http://data-
vocabulary.org/Address">
   <span itemprop="locality">Минск</span>,
   <span itemprop="street-address">ул.Кожеватова 17</span>,
   <span itemprop="country-name">Беларусь</span>.
 </div>
 <span itemprop="tel">8(029)763-93-82</span>,
 <a itemprop="url" href="http://mikhalkevich.colony.by" >
mikhalkevich.colony.by </a>
</div>
```

В данном примере для определения пространства имен используется формат кодирования контактной информации http://data-vocabularo.org/Person. На сайте также можно найти другие форматы для кодирования данных.

Микроданные имеют один значительный недостаток: современные браузеры по умолчанию не имеют поддержки формата микроданных. Но использование вышеперечисленных форматов повышает шансы попадания страницы в результаты поиска.

Google рассматривает все микроформаты и микроданные как равнозначные, поэтому неважно, какой из стандартов использовать.

Протестировать, как работают расширенные фрагменты информационных блоков, можно с помощью специального инструмента Rich Snippets Testing Tool (RSTT).

Инструмент RSTT решает следующие задачи:

1. Проверяет семантическую разметку.
2. Указывает каким образом семантические данные могут повлиять на представление страницы в результатах поиска Google.

Использовать данный инструмент достаточно просто. Необходимо посетить следующий web-сайт:

http://google.com/webmasters/tools/richsnippets. В поле для ввода url, ввести путь к странице, семантику которой необходимо проверить.

4. Продвинутые web-формы

Лучший способ обучения работы с формами HTML5 — это взять стандартную современную форму и усовершенствовать её средствами HTML5. Рассмотрим стандартные web-формы:

Форма HTML. Листинг 4.1

```
 <!DOCTYPE html>
<html lang="en">
<head>
  <title>Forms</title>
</head>
<body>
  <section id="form">
    <form name="myform" method="post" action="file.php">
      <br><label for="myname">Text: </label>
      <input type="text" name="myname">
      <br><label for="myoption">Radio Buttons: </label>
      <input type="radio" name="myoption" value="1" checked> 1
      <input type="radio" name="myoption" value="2"> 2
      <input type="radio" name="myoption" value="3"> 3
      <br><label for="mypassword">Password: </label>
      <input type="password" name="mypassword">
      <br><label for="mycheckbox">Checkbox: </label>
      <input type="checkbox" name="mycheckbox" value="123">
      <input type="hidden" value="secret key">
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

И еще одну форму с элементами <textarea> и <select>

Стандартная форма с элементами <textarea> и <select>. Листинг 4.2

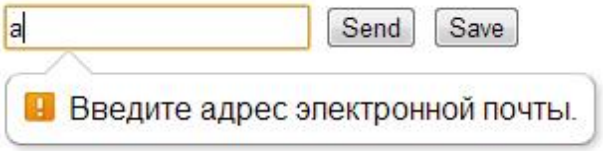
```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Forms</title>
</head>
<body>
```


```
  <section id="form">
    <form name="myform" method="post" action="file.php">
      <label for="mytext">Textarea: </label>
      <textarea name="mytext" rows="5" cols="30"></textarea>
      <br><label for="mylist">Select: </label>
      <select name="mylist">
        <option value="1">One</option>
        <option value="2">Two</option>
        <option value="3">Three</option>
      </select>
      <input type="submit" value="Send">
      <input type="reset" value="Reset">
    </form>
  </section>
</body>
</html>
```

Модернизируем существующую форму, добавив в нее следующие типы данных (многие из которых имеют встроенный механизм валидации):

Тип данных e-mail

В спецификации HTML5 сказано, что данный тип предназначен для хранения e-mail адреса или адресов, разделенных запятыми. Если пользователь попытается вставить что-то кроме e-mail адреса, то отправки данных не произойдет и барузер выдаст ошибку:

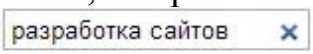


Тип данных e-mail. Листинг 4.3

```
<input type="email" name="myemail">
```

Тип данных search

Данный тип применяется для полей поиска, или для ввода каких-то ключевых слов, по которым потом выполняется какой-либо вид поиска. Это может быть поиск по всему Интернету, или поиск на странице, или фильтр. Современные браузеры добавляют крестик (X) к данному типу поля после того, как пользователь начинает вводить в поле поисковую фразу, для того, чтобы можно было очистить данное поле, не прибегая к клавиатуре.



Тип данных search. Листинг 4.4

```
<input type="search" name="mysearch">
```

Тип данных для ввода url

Предназначение этого типа данных — ввод url-адреса или адресов через запятую. Некоторые мультимедийные устройства, например iPhone, для этого типа полей переключают раскладку клавиатуры для ускоренного ввода web-адресов.

Тип данных url. Листинг 4.5

```
<input type="url" name="myurl">
```

Тип данных для ввода номера телефона

Основное предназначение данного типа — это автоматическая смена клавиатуры для специальных мультимедийных устройствах. Встроенного механизма валидации в данной форме нет.

Тип данных tel. Листинг 4.6

```
<input type="tel" name="myphone">
```

Числовой тип данных

Числовой тип данных содержит атрибуты для максимального допустимого числа, минимального и шага.

Современные браузеры отображают числовой тип данных следующим образом:

Тип данных number. Листинг 4.7

```
<input type="number" name="mynumber" min="0" max="10" step="5">
```

Ползунки

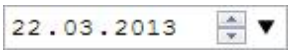
Тип данных range. Листинг 4.8

```
<input type="range" name="mynumbers" min="0" max="10" step="5">
```

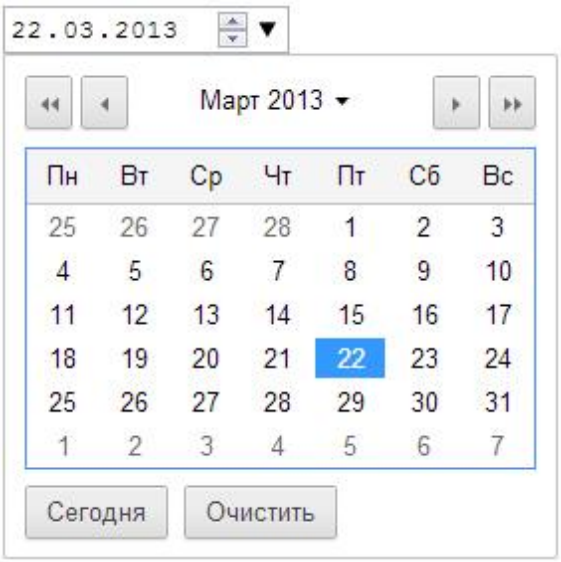
Дата и время

Для работы с датой и времени существует аж пять типов форм. Рассмотрим их по-порядку.

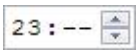
date



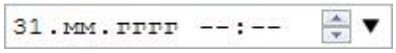
При клике на стрелочку открывается календарь:



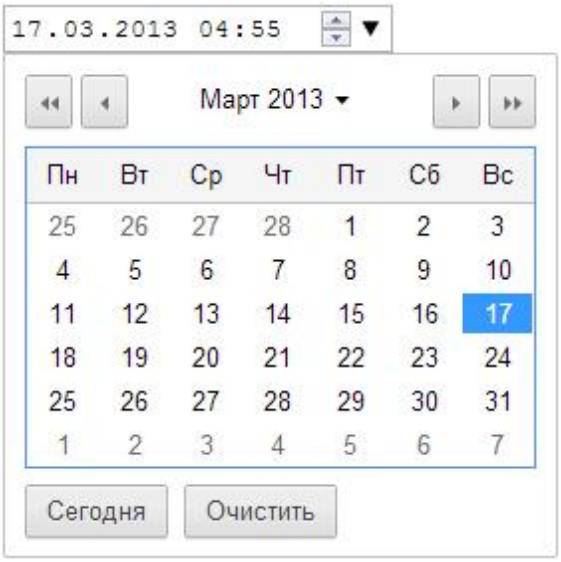
time



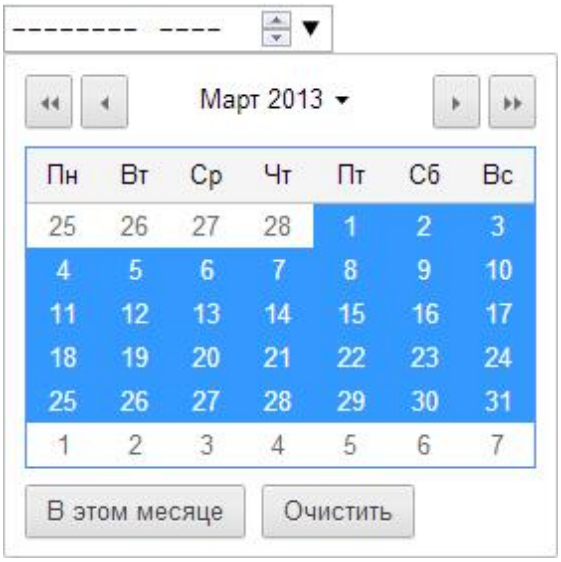
datetime-local



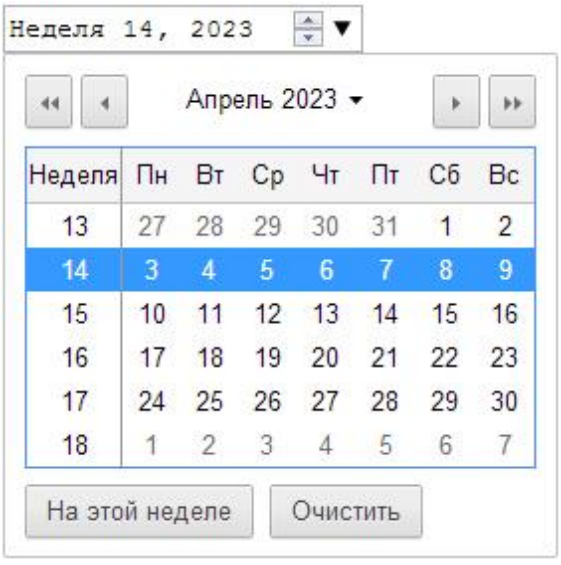
При клике на черный треугольник открывается календарь:


17.03.2013 04:55
Март 2013
Пн Вт Ср Чт Пт Сб Вс
25 26 27 28 1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
1 2 3 4 5 6 7
Сегодня
Очистить

month

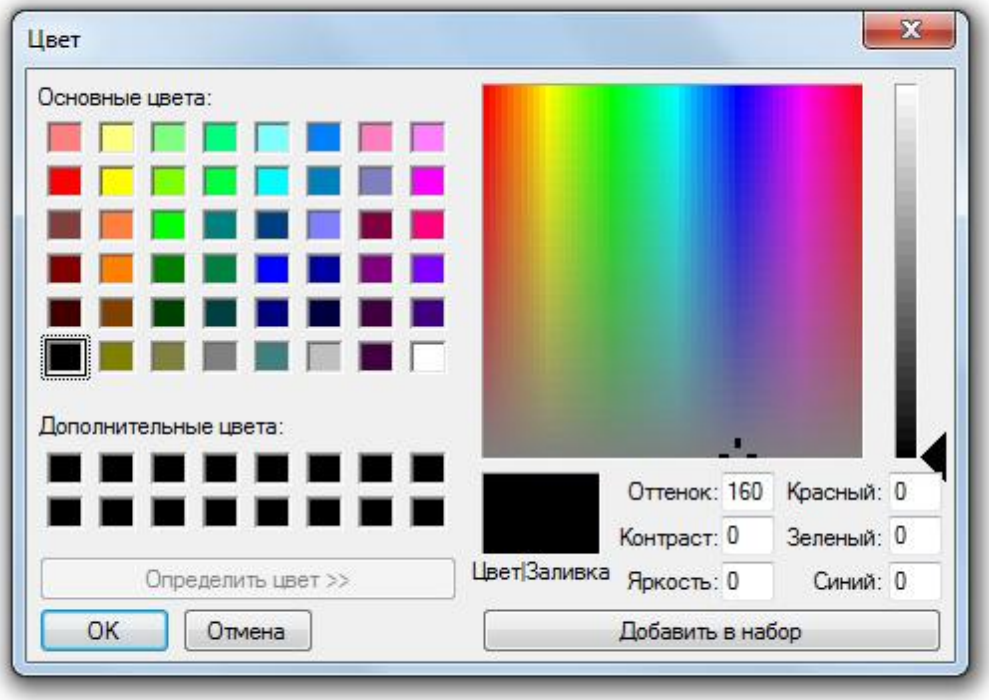

-------- ----
Март 2013
Пн Вт Ср Чт Пт Сб Вс
25 26 27 28 1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
1 2 3 4 5 6 7
В этом месяце
Очистить

week


Неделя 14, 2023
Апрель 2023
Неделя Пн Вт Ср Чт Пт Сб Вс
13 27 28 29 30 31 1 2
14 3 4 5 6 7 8 9
15 10 11 12 13 14 15 16
16 17 18 19 20 21 22 23
17 24 25 26 27 28 29 30
18 1 2 3 4 5 6 7
На этой неделе
Очистить

Типы данных даты и времени. Листинг 4.9

```
<input type="date" name="mydate">
<input type="time" name="mytime">
<input type="datetime" name="mydatetime">
<input type="datetime-local" name="mylocaldatetime">
<input type="month" name="mymonth">
<input type="week" name="myweek">
```



Тип для ввода цвета

Тип данных color. Листинг 4.10

```
<input type="color">
```

Использование атрибута autocomplete

Выключение автозаполнения элементов форм. Листинг 4.11

```
<input type="search" name="mysearch" autocomplete="off">
```

Отмена валидации

Отправка формы без валидации, с использованием атрибута formnovalidate:

Отправка формы без валидации. Листинг 4.12

```
<input type="search" name="mysearch" autocomplete="off">    <form
name="myform" method="get" action="file.php">
      <input type="email" name="myemail">
      <input type="submit" value="Send">
      <input type="submit" value="Save" formnovalidate>
    </form>
```

Подсказки ввода, атрибут placeholder

Использование атрибута placeholder. Листинг 4.13

```
  <input type="search" name="mysearch" placeholder="type your
seach">
```

Объявление элемента формы обязательным для заполнения

Для этих целей используем атрибут required

Использование атрибута required. Листинг 4.14

```
<input type="email" name="myemail" required>
```

Атрибут multiple для добавления множественных значений

Использование атрибута multiple. Листинг 4.15

```
<input type="email" name="myemail" multiple>
```

Атрибут autofocus

Использование атрибута autofocus. Листинг 4.16

```
<input type="search" name="mysearch" autofocus>
```

Шаблон регулярного выражения

В формах HTML5 регулярные выражения применяются для валидации.

Использование шаблона регулярного выражения. Листинг 4.17

```
<input pattern="[A-Z]{3}-[0-9]{3}" name="pcode" title="insert the
5 numbers of your postal code">
```

Квадратные скобки определяют диапазон допустимых значений. Группа [A-Z] разрешает символы английского алфавита в верхнем регистре. Идущие за группой символов фигурные скобки определяют количество символов.
Таким образом, следующие значения будут допустимы для ввода в поле:
ABC-999
CSG-000
ADD-801
А эти значения нет:
aAD-999
НЮС-888
LA8-999
DFA-9999

Элементы формы за пределами формы

В HTML5 элементы форм можно объявлять за пределами элемента <form>. Чтобы связать элемент формы с самой формой в элементе необходимо создать атрибут form с именем формы:

Объявление элементов форм за пределами формы. Листинг 4.18

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Forms</title>
</head>
<body>
  <nav>
    <input type="search" name="mysearch" form="myform">
  </nav>
  <section>
    <form name="myform" method="get" action="file.php">
      <input type="text" name="myname">
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

Подсказки ввода <datalist>

Для связки подсказок с элементом формы, в элементе необходимо создать атрибут list с именем элемента <datalist>:

Элемент <datalist>. Листинг 4.19

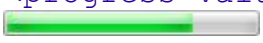
```
    <form name="myform" method="get" action="file.php">
     <datalist id="mydata">
       <option value="123123123" label="Phone 1">
       <option value="456456456" label="Phone 2">
      </datalist>
      <input type="search" name="mysearch" list="mydata"
autocomplete="off">
      <input type="submit" value="Send">
      <input type="submit" value="Save">
    </form>
```

Индикатор прогресса

Индикатор прогресса. Листинг 4.20

```
<progress>working...</progress>
```

```
<progress value="75" max="100">3/4 complete</progress>
```



Шкала загрузки

Применяется для измерения данных в пределах заданного диапазона.

Шкала загрузки. Листинг 4.21

```
<meter min="0" max="100" low="40" high="90" optimum="50"
value="91">A+</meter>
```

Псевдоклассы :valid и :invalid

Использование стилей для псевдоклассов. Листинг 4.22

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Forms</title>
  <style>
    :valid{
      background: lightgreen;
    }
    :invalid{
      background: red;
    }
  </style>
</head>
<body>
  <section>
    <form name="myform" method="get" action="file.php">
      <input type="email" name="myemail">
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

Псевдоклассы :required и :optional

:required применяется для обязательных для заполнения типов данных;
:optional – для необязательных.

Использование стилей для псевдоклассов :required и :optional. Листинг 4.23

```
 <!DOCTYPE html>
<html lang="ru">
<head>
  <title>Forms</title>
  <style>
    :required{
      border: 2px solid #990000;
    }
    :optional{
      border: 2px solid #009999;
    }
  </style>
</head>
<body>
  <section>
    <form name="myform" method="get" action="file.php">
      <input type="text" name="myname">
```

```
      <input type="text" name="mylastname" required>
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

Псевдоклассы :in-range и :out-of-range

Использование стилей для псевдоклассов :in-range и :out-of-range. Листинг 4.24

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Forms</title>
  <style>
    :in-range{
      background: #EEEEFF;
    }
    :out-of-range{
      background: #FFEEEE;
    }
  </style>
</head>
<body>
  <section>
    <form name="myform" method="get" action="file.php">
      <input type="number" name="mynumber" min="0" max="10">
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

Редактирование элементов с помощью атрибута contentEditable

Редактирование элемента <p> с помощью атрибута contentEditable. Листинг 4.25

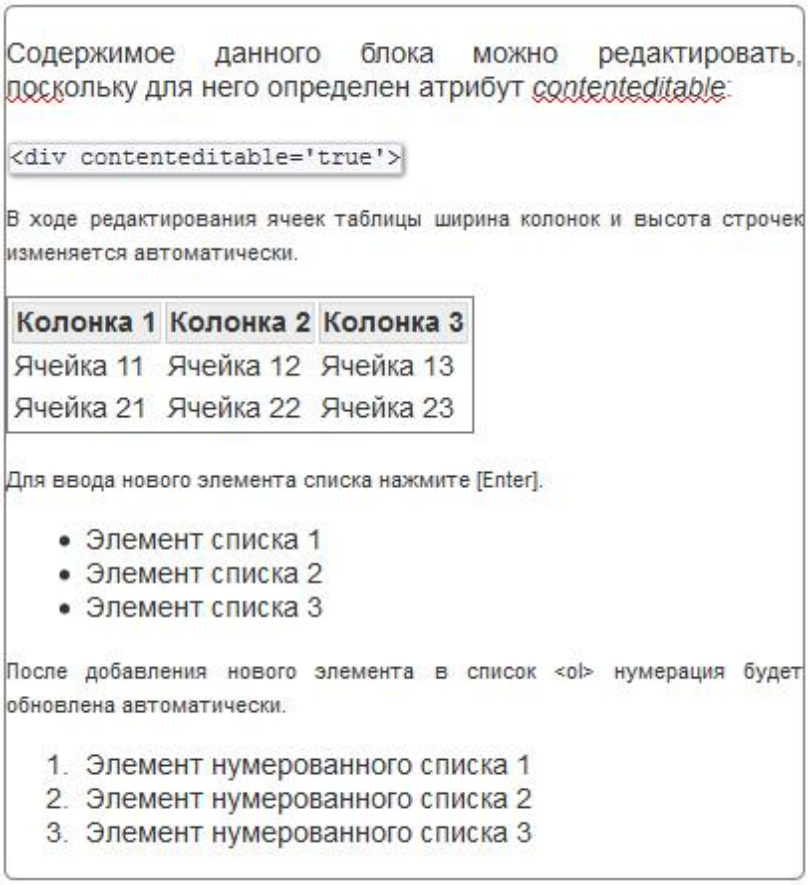
```
 <p id=”editableElement” contentEditable>Вы можете редактировать
этот текст</p>
```

Щелчек мышью в редактируемой области, помещает в нее курсор для редактирования.

Редактирование страницы с помощью атрибута designMode

Атрибут designMode похож на атрибут contentEditable, только он позволяет редактировать всю страницу. Обычно, редактируемая страница помещается внутрь элемента <iframe>, который ведет себя, как окно редактирования. Такая функциональность может быть полезна для организации обратной связи с владельцем ресурса, например, для заполнения и отправки ему заявления, заказа или чего-нибудь еще в этом роде. Вы создаете html страницу бланка в его первоначальном виде и отдаете пользователю для заполнения. После за-

полнения пользователь подтверждает введенные данные, и отредактированный документ отправляется на сервер. Пример редактируемой разметки приведен ниже:



Специализированная проверка

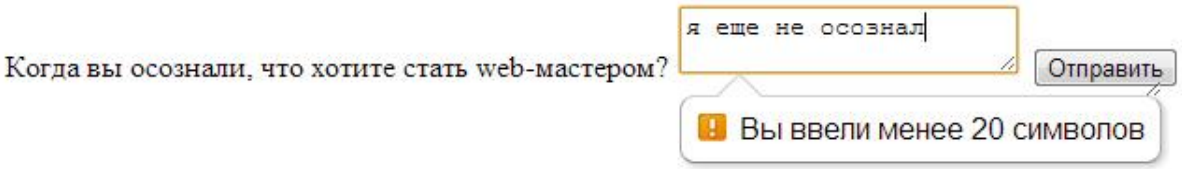
Спецификация HTML5 оговаривает набор свойств javaScript, с помощью которых можно определить корректность вводимых в значений. Одним из самых полезных из них является метод setCustomValidity().

Использование метода setCustomValidity() для проверки пользовательских значений. Листинг 4.26

```
<script>
 function validateComments(input) {
  if(input.value.length < 20){
   input.setCustomValidity("Вы ввели менее 20 символов");
  } else {
   // если длина комментария отвечает требованию, очищаем
предыдущее сообщение об ошибке
   input.setCustomValidity("");
  }
 }
</script>
<form>
<label for="comments">
 Когда вы осознали, что хотите стать web-мастером?
</label>
 <textarea id="comments" oninput="validateComments(this)">
```

```
 </textarea>
 <input type="submit" />
</form>
```

При попытке пользователя ввести в элемент формы менее 20 символов, на экране появится следующая ошибка:



Обработка пользовательских ошибок с помощью javaScript

Следует сделать важное замечание: **проверка пользовательских значений на стороне javaScript нельзя применять в качестве альтернативы серверной проверки**. Обязательно необходимо комбинировать оба способа. Пользователю, который знает firebug firefox-а или любой другой web-инспектор других браузеров, не составит труда отключить любую браузерную проверку.

В примере два элемента формы, один из которых (любой), обязателен для заполнения:

Обработка пользовательских ошибок на javaScript. Листинг 4.27

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Forms</title>
  <script>
    var name1, name2;
    function initiate(){
      name1 = document.getElementById("firstname");
      name2 = document.getElementById("lastname");
      name1.addEventListener("input", validation);
      name2.addEventListener("input", validation);
      validation();
    }
    function validation(){
      if(name1.value == '' && name2.value == ''){
        name1.setCustomValidity('insert at least one name');
        name1.style.background = '#FFDDDD';
        name2.style.background = '#FFDDDD';
      }else{
        name1.setCustomValidity('');
        name1.style.background = '#FFFFFF';
        name2.style.background = '#FFFFFF';
      }
    }
    addEventListener("load", initiate);
  </script>
</head>
```

```
<body>
  <section>
    <form name="registration" method="get" action="file.php">
      <label for="firstname">First Name: </label>
      <input type="text" name="firstname" id="firstname">
      <label for="lastname">Last Name: </label>
      <input type="text" name="lastname" id="lastname">
      <input type="submit" value="Sign Up">
    </form>
  </section>
</body>
</html>
```

HTML-код формы понятен. Рассмотрим javaScript. В скрипте имеется две функции и прослушиватель:

addEventListener("load", initiate);

Функция addEventListener предназначена для прослушивания событий (более подробно рассмотрим ее позже). В данном примере, она прослушивает событие load (загрузка страницы). Как только наступает данное событие, т.е. страница загружена, прослушиватель вызывает функцию initiate. Функция initiate вызывает элементы с идентификаторами firstname и lastname, и добавляет им прослушиватели на событие input. В функции validation() осуществляется проверка на то, чтобы пользователь заполнил хотя бы один элемент формы. Если оба элемента формы пусты, то функция вызывает метод setCustomValidity и передает в него ошибку '*insert at least one name*', при этом меняется стиль элементов форм.

Создание собственной системы проверки ошибок

Собственная система проверки ошибок. Листинг 4.28

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Forms</title>
  <script>
    var form;
    function initiate(){
      var button = document.getElementById("send");
      button.addEventListener("click", sendit);
      form = document.querySelector("form[name='information']");
      form.addEventListener("invalid", validation, true);
    }
    function validation(e){
      var elem = e.target;
      elem.style.background = '#FFDDDD';
    }
    function sendit(){
      var valid = form.checkValidity();
      if(valid){
```

```
        form.submit();
      }
    }
    addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <form name="information" method="get" action="file.php">
      <label for="nickname">Nickname: </label>
      <input pattern="[A-Za-z]{3,}" name="nickname" id="nickname"
maxlength="10" required>
      <label for="myemail">Email: </label>
      <input type="email" name="myemail" id="myemail" required>
      <input type="button" id="send" value="Sign Up">
    </form>
  </section>
</body>
</html>
```

Как и в предыдущем листинге, по загрузке страницы вызывается функция initiate(). Функция initiate содержит два прослушивателя: на события click и invalid. Если в поле не проходящие валидацию данные, то прослушиватель invalid вызывает функцию validation(). При попытке отправки формы, вызывается функция sendit().

Валидация в режиме реального времени

Валидация в режиме реального времени. Листинг 4.29

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Forms</title>
  <script>
    var form;
    function initiate(){
      var button = document.getElementById("send");
      button.addEventListener("click", sendit);
      form = document.querySelector("form[name='information']");
      form.addEventListener("invalid", validation, true);
      form.addEventListener("input", checkval);
    }
    function validation(e){
      var elem = e.target;
      elem.style.background = '#FFDDDD';
    }
    function sendit(){
      var valid = form.checkValidity();
      if(valid){
        form.submit();
      }
    }
```

```
    function checkval(e){
      var elem = e.target;
      if(elem.validity.valid){
        elem.style.background = '#FFFFFF';
      }else{
        elem.style.background = '#FFDDDD';
      }
    }
    addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <form name="information" method="get" action="file.php">
      <label for="nickname">Nickname: </label>
      <input pattern="[A-Za-z]{3,}" name="nickname" id="nickname"
maxlength="10" required>
      <label for="myemail">Email: </label>
      <input type="email" name="myemail" id="myemail" required>
      <input type="button" id="send" value="Sign Up">
    </form>
  </section>
</body>
</html>
```

5. Гибкая блочная модель

Гибкая блочная модель — это альтернатива блочной верстке, которая впитала в себя все достоинства блочной и возможности табличной.

Создадим HTML-код для шаблона гибкой блочной модели:

HTML-код. Листинг 5.1

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Flexible Box Model</title>
  <link rel="stylesheet" href="test.css">
</head>
<body>
<section id="parentbox">
  <div id="box-1">Box 1</div>
  <div id="box-2">Box 2</div>
  <div id="box-3">Box 3</div>
  <div id="box-4">Box 4</div>
</section>
</body>
</html>
```

Flex

Свойство, которое отвечает за ширину элемента.
Добавим гибкости с помощью атрибута flex (*по некоторым источникам, Значением display может быть flexbox или inline-flexbox*)

Равномерные блоки с использованием атрибута flex. Листинг 5.2

```
#parentbox {
  width:600px;
  display: flex;
  display: -moz-box;
  display: -webkit-flex;
  border:dotted 1px #ccc;
  padding:2px;
}
#parentbox div {
 background:#ccc;
 color:white;
 padding:10px;
 border:solid 1px #000;
}
#box-1{
  -moz-box-flex:1;
  -webkit-flex:1;
  flex:1;
}
#box-2{
 -moz-box-flex:3;
 -webkit-flex:3;
  flex:3;
}
#box-3{
 -moz-box-flex:1;
 -webkit-flex:1;
  flex:1;
}
#box-4{
 -moz-box-flex:1;
 -webkit-flex:1;
  flex:1;
}
```

Получим следующее:

Рассмотрим пример неравномерной блочной модели

Неравномерные блоки с использованием атрибута flex. Листинг 5.3

```
#box-1{
 -moz-box-flex:3;
 -webkit-flex:3;
  flex:3;
}
```

```
#box-2{
  -moz-box-flex:1;
  -webkit-flex:1;
  flex:1;
}
#box-3{
 -moz-box-flex:1;
 -webkit-flex:1;
  flex:1;
}
#box-4{
 -moz-box-flex:1;
 -webkit-flex:1;
  flex:1;
}
```

Получим следующее:

Комбинирование гибких и фиксированных свойств:

Комбинирование гибких и фиксированных свойств атрибута flex. Листинг 5.4

```
#parentbox {
  display: flex;
  width: 600px;
}
#box-1{
  width: 300px;
}
#box-2{
  flex: 1;
}
#box-3{
  flex: 1;
}
#box-4{
  flex: 1;
}
```

Ограничение минимального размера гибкой блочной модели:

Ограничение минимального размера атрибута flex. Листинг 5.5

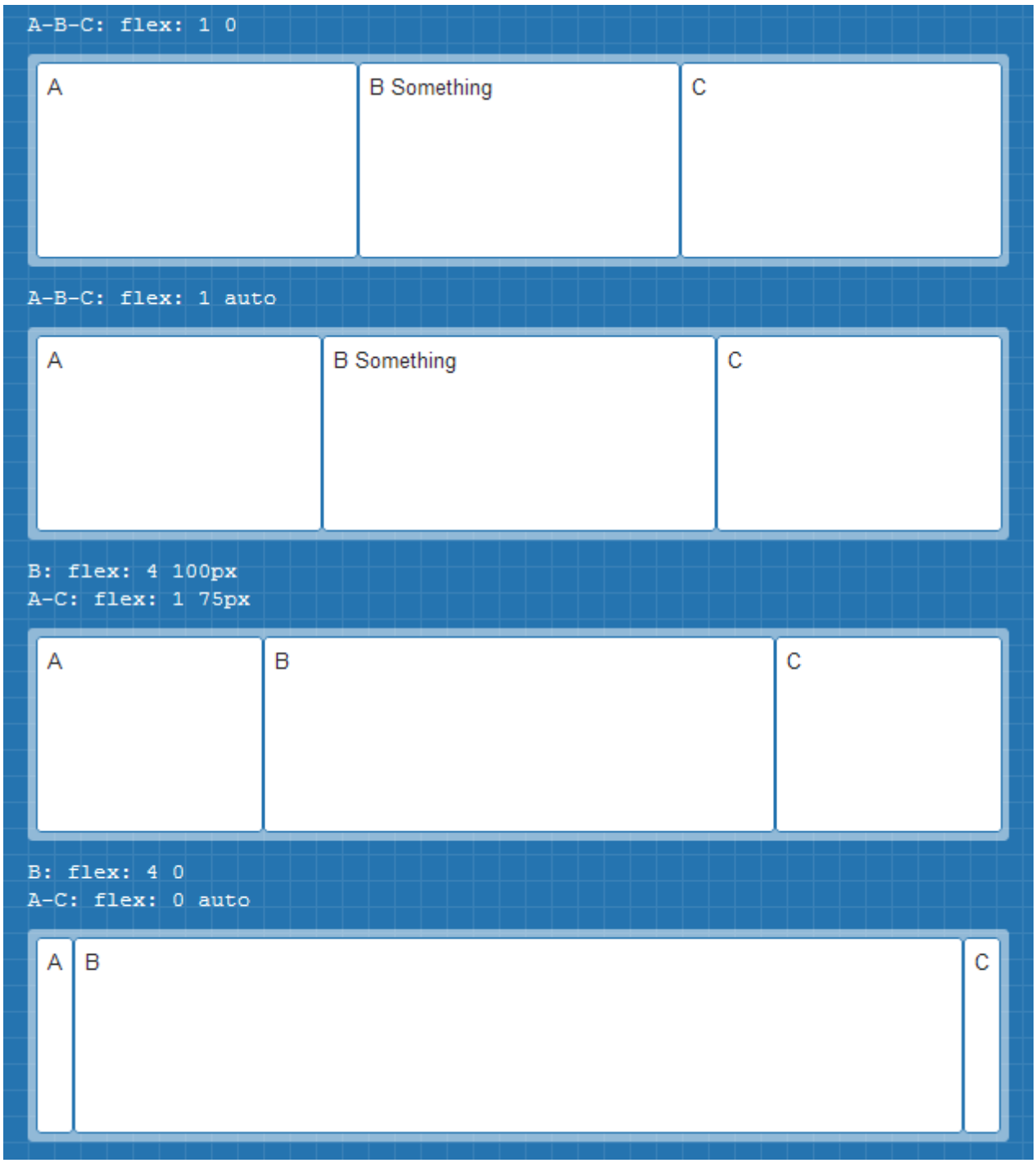
```
#parentbox {
  display: flex;
}
#box-1{
  flex: 1 1 200px;
}
#box-2{
  flex: 1 5 100px;
```

```
}
#box-3{
  flex: 1 5 100px;
}
#box-4{
  flex: 1 5 100px;
}
```

Определение гибкой блочной верстки с предпочтительным размером.

Задание предпочтительного размера атрибута flex. Листинг 5.6

```
#parentbox {
  display: flex;
  width: 600px;
}
#box-1{
  width: 200px;
  flex: 1 1 auto;
}
#box-2{
  width: 100px;
  flex: 1 1 auto;
}
#box-3{
  width: 100px;
  flex: 1 1 auto;
}
#box-4{
  width: 100px;
  flex: 1 1 auto;
}
```

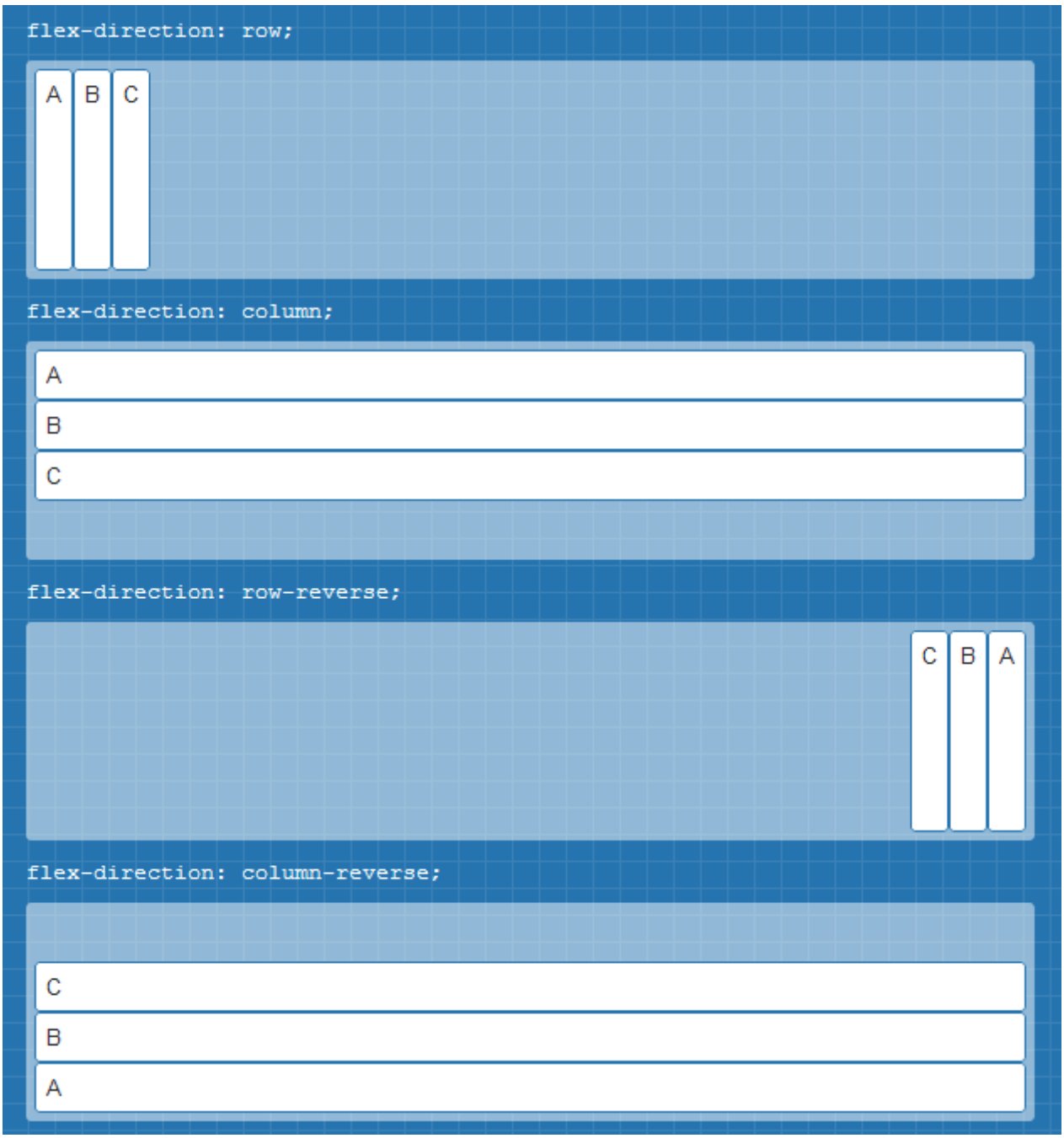


Flex-direction
Определяет направление оси, по которой выстравиваются блоки.
Присваивается элементу контейнеру. Значения говорят сами за себя:

Возможные значения атрибута flex-direction. Листинг 5.7

```
section {
 flex-direction: row | row-reverse | column | column-reverse;
}
```

row – означает что блоки будут выстраивается по горизонтальной линии
column – по вертикальной, как обычные блоки.



Order

Изменение позиции отображения блоков: для этого используется атрибут order со значениями номера позиции отображения.

Изменение позиции отображения с помощью атрибута order. Листинг 5.8

```
#parentbox {
  display: flex;
}
#box-1{
  flex: 1;
  order: 2;
}
#box-2{
```

```
  flex: 1;
  order: 4;
}
#box-3{
  flex: 1;
  order: 3;
}
#box-4{
  flex: 1;
  order: 1;
}
```

Justify-content

Для заполнения свободного горизонтального пространства используется атрибут justify-content

Возможные варианты:

Justify-content. Листинг 5.9

```
 Section {justify-content: flex-start | flex-end | center | space-between | space-around}
```

Использование атрибута justify-content. Листинг 5.10

```
#parentbox {
  display: flex;
  width: 1200px;
  justify-content: flex-end;
}
#box-1{
  width: 100px;
}
#box-2{
  width: 100px;
}
#box-3{
  width: 100px;
}
#box-4{
  width: 100px;
}
```

Flex-wrap

Определяет, будет ли наш блок в одну строку или многострочный. wrap – многострочный, nowrap – однострочный.

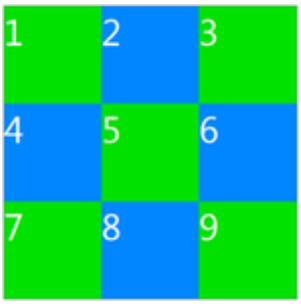
Дочерние элементы разделяются на строки и отображаются последовательными параллельными строками или столбцами.

Возможны значения:

wrap

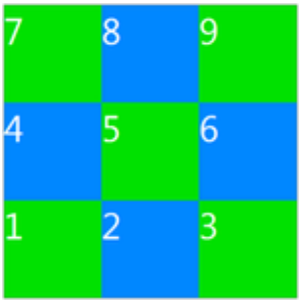
Дочерние элементы разделены на строки и отображаются последовательными параллельными строками или столбцами. Объект

разворачивается в высоту или ширину перпендикулярно оси, определенной свойством writing-mode, чтобы вместить дополнительные строки или столбцы.



wrap-reverse

Дочерние элементы разделены на строки и отображаются последовательными параллельными строками или столбцами в обратном порядке. Объект разворачивается в высоту или ширину перпендикулярно оси, определенной свойством writing-mode, чтобы вместить дополнительные строки или столбцы.



Flex-align

Возможные значения атрибута flex-align. Листинг 5.11

```
section { flex-align: start | end | center | justify | distribute
| stretch; }
```

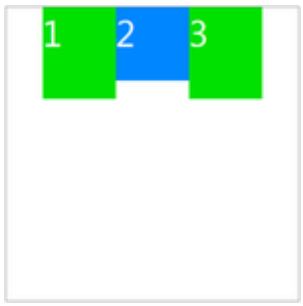
Рассмотрим пример

Выравнивание элементов в разных браузерах. Листинг 5.12

```
#parentbox {
  width:100%;
  display: flex;
  display: -moz-box;
  display: -webkit-flex;
  border:dotted 1px #ccc;
  padding:2px;
  align-items:center;
  -webkit-align-items:center;
  -moz-box-align:center;
}
#parentbox div {
 background:#ccc;
 color:white;
 padding:10px;
```

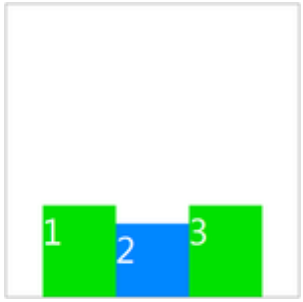
```
 border:solid 1px #000;
}
#box-1{
  -moz-box-flex:1;
  -webkit-flex:1;
  flex:1;
}
#box-2{
 -moz-box-flex:3;
 -webkit-flex:3;
  flex:3;
}
#box-3{
 -moz-box-flex:1;
 -webkit-flex:1;
  flex:1;
}
#box-4{
 -moz-box-flex:1;
 -webkit-flex:1;
  flex:1;
}
```

Выравнивание элементов относительно перпендикуляру главной оси. flex-align – применяется к контейнеру, а flex-item-align для элемента на случай, когда надо перекрыть общее значение.

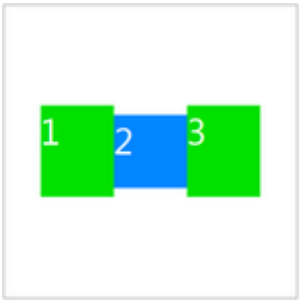


start

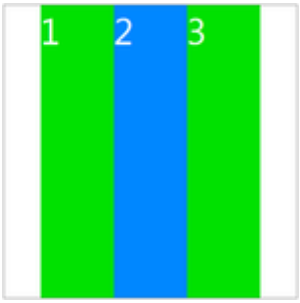
end



center



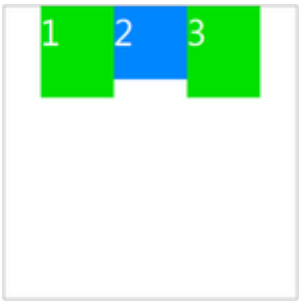
stretch

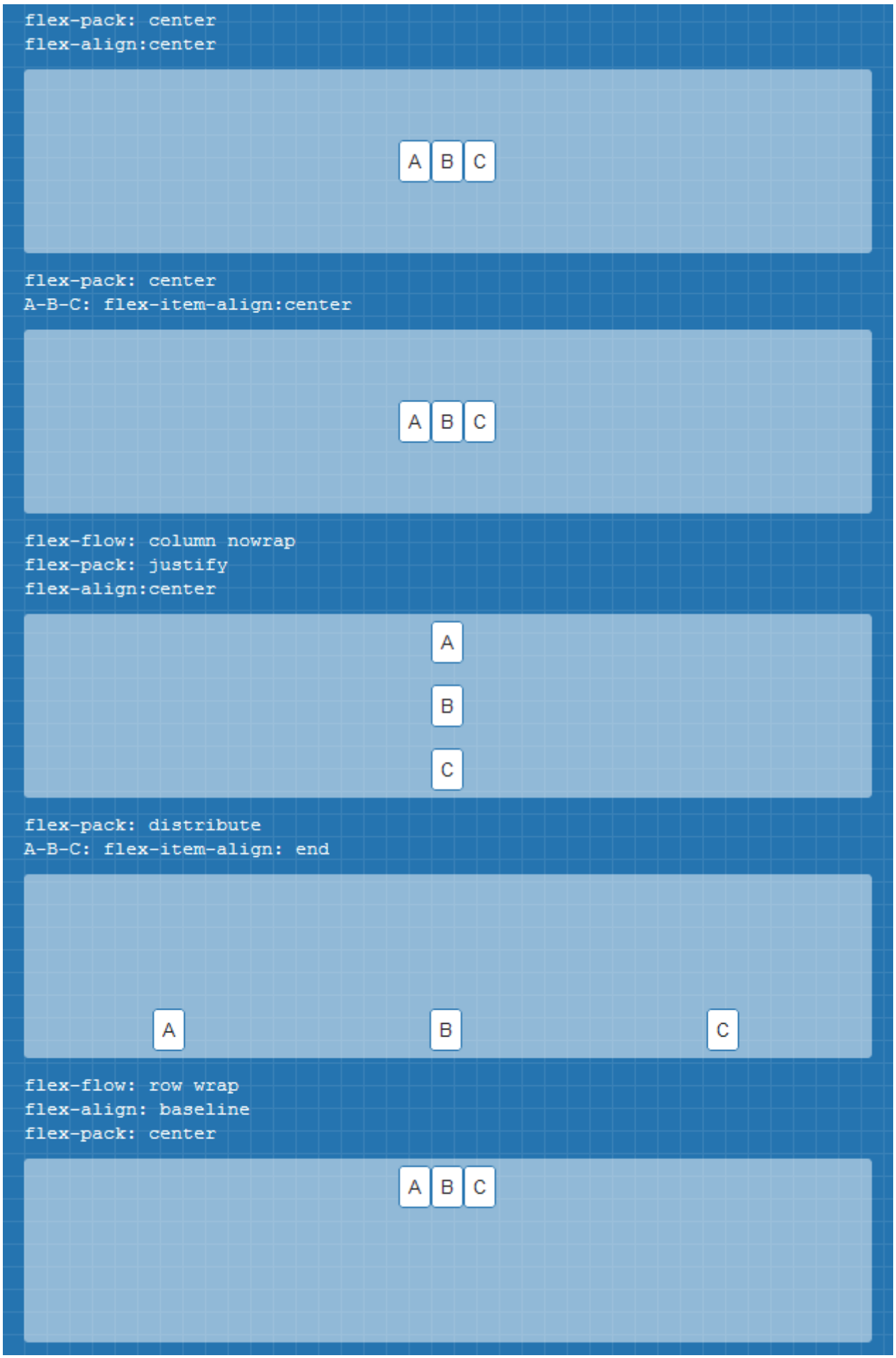


baseline

Направляющие (начальный или конечный край в зависимости от свойства -ms-flex-direction) для всех дочерних элементов выравниваются друг относительно друга.

Дочерние элементы, занимающие больше всего пространства, перпендикулярного оси размещения, следуют правилу start; направляющие всех оставшихся элементов выравниваются по опорной линии этого элемента.

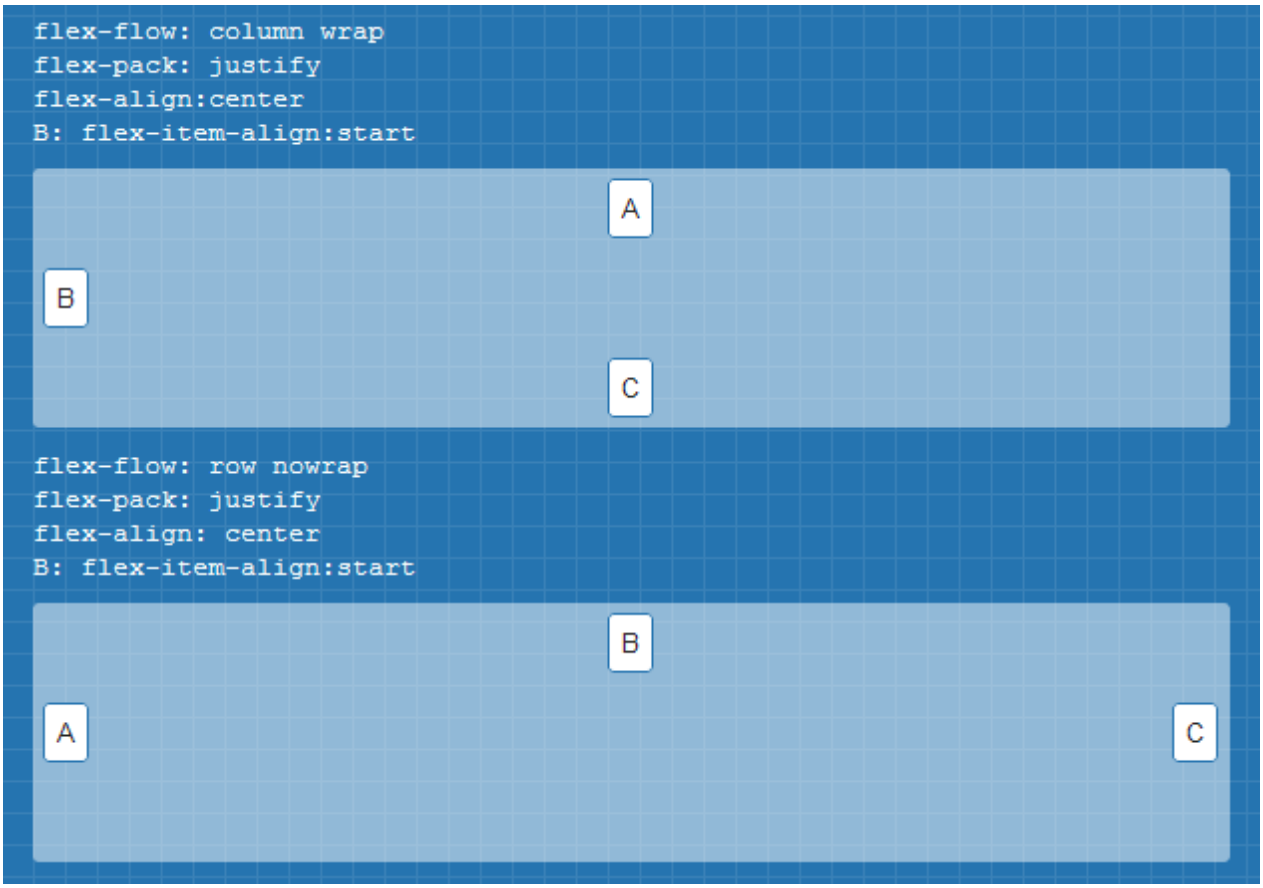



flex-pack: center
flex-align:center
A B C
flex-pack: center
A-B-C: flex-item-align:center
A B C
flex-flow: column nowrap
flex-pack: justify
flex-align:center
A
B
C
flex-pack: distribute
A-B-C: flex-item-align: end
A
B
C
flex-flow: row wrap
flex-align: baseline
flex-pack: center
A B C

Flex-item-align

Возможные значения атрибута flex-item-align. Листинг 5.13

```
section div{ flex-item-align: start | end | center | justify |
distribute | stretch; }
```

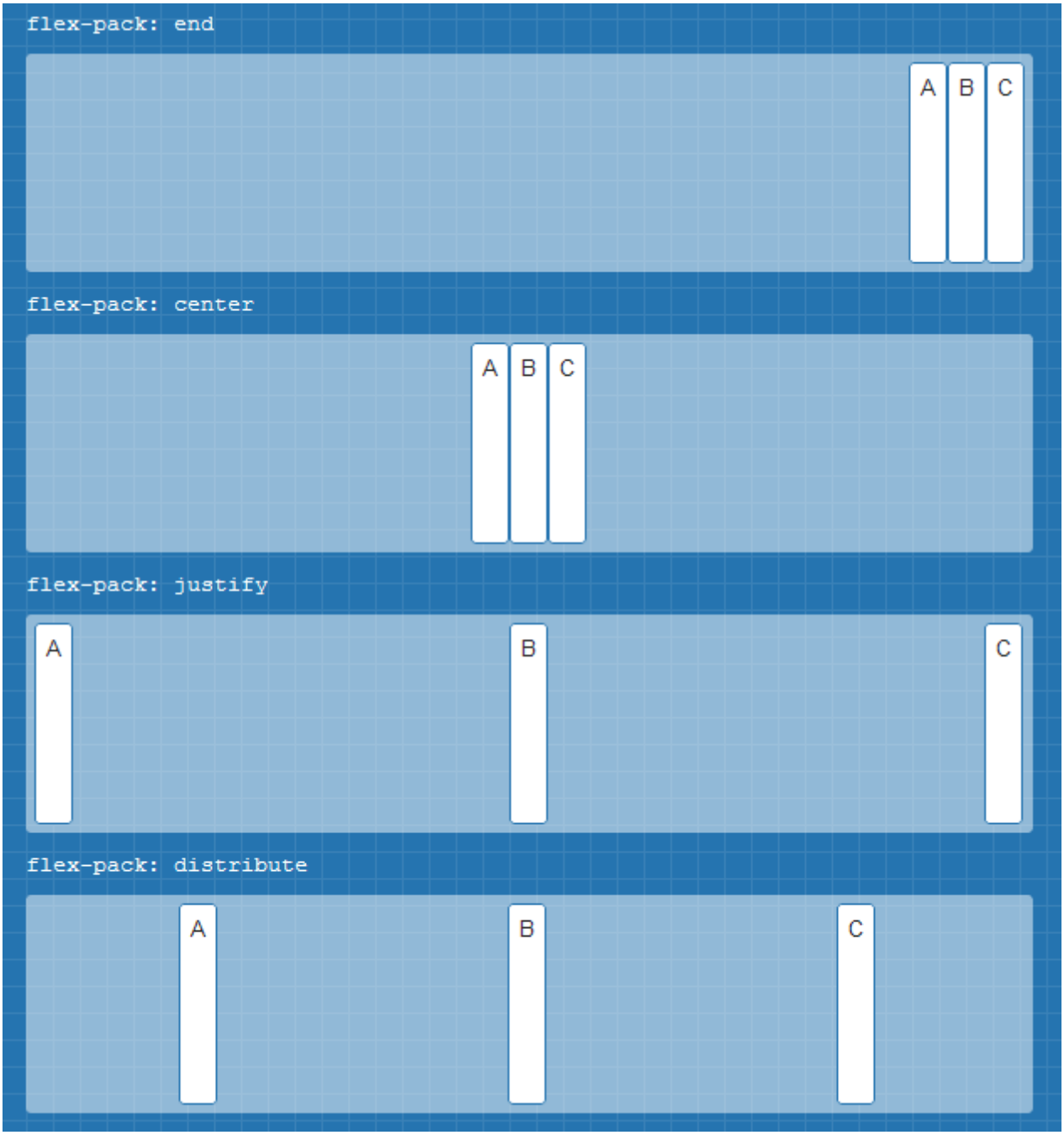


Flex-pack

Свойство, которое выравнивает блоки относительно главной оси, то есть той по которой выстраиваются блоки.

Выравнивание относительно главной оси. Листинг 5.14

```
section {flex-pack: start | end | center | justify | distribute;}
```



Flex-flow
Как видно из синтаксиса, это сокращенная запись двух свойств.

Flex-flow. Листинг 5.15

```
section { flex-flow: <'flex-direction'> || <'flex-wrap'>; }
```

6. Особенности CSS3

Новые возможности CSS
Поскольку свойства CSS, которые мы здесь изучаем, находятся на стадии разработки, большинство из них необходимо объявлять с использованием специального префикса, определяющего механизм визуализации. В будущем

можно будет избавиться от префиксов. Однако пока экспериментальный этап не завершился, приходится использовать следующие префиксы:

- **moz** – для Firefox
- **webkit** – Safary и Opera
- **o** – только Opera
- **khtml** – Konqueror
- **ms** – Internet Explorer
- **chrome** — только Chrome.

Пример добавления префиксов к новым свойствам. Листинг 6.1

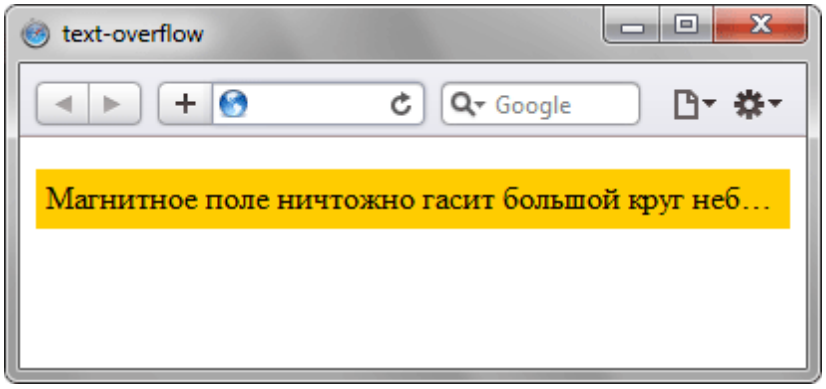
```
#wrapper {
  display: -moz-box;
  display: -webkit-box;
  display: box;
  -moz-box-orient:vertical;
  -webkit-box-orient:vertical;
  box-orient:vertical;
}
```

text-overflow

Определяет способ переноса текста. Определяет параметры видимости текста в блоке, если текст целиком не помещается в заданную область. Возможны два варианта: текст обрезается; текст обрезается и к концу строки добавляется многоточие. text-overflow работает в том случае, если для блока значение свойства overflow установлено как auto, scroll или hidden.

Перенос текста. Листинг 6.2

```
div {
  text-overflow: ellipsis; //добавляет многоточие
}
```



Column-count

Свойство column-count определяет количество колонок в многоколоночном элементе.

Разбиение текста на колонки. Листинг 6.3

```
div {
 column-count: 4;
```

```
 column-rule: 1px solid #bbb;
 column-gap: 2em;
}
```

этот блок состоит из двух колонок этот
двух колонок этот блок состоит из
блок состоит из двух колонок этот
двух колонок этот блок состоит из
блок состоит из двух колонок этот
двух колонок этот блок состоит из
блок состоит из двух колонок этот
двух колонок этот блок состоит из
блок состоит из двух колонок

Обрамление текста. Листинг 6.4

```
div {
 text-fill-color: black;
 text-stroke-color: red;
 text-stroke-width: 0.00px;
}
```

Прозрачность. Листинг 6.5

```
div {
  color: rgba(255, 0, 0, 0.75);
  background: rgba(0, 0, 255, 0.75);
}
```

Оттенок. Листинг 6.6

```
div {
  color: hsla(2, 16%, 33%, 1.00);
  //Оттенок / Насыщенность / яркость цветовой модели
}
```

Линейный градиент. Листинг 6.7

```
div {
background: gradient(linear, left top, left bottom,
                from(#00abeb), to(white),
                color-stop(0.5, white), color-stop(0.5, #66cc00))
}
```

Радиальный градиент. Листинг 6.8

```
div {
background: gradient(radial, 430 50, 0, 430 50, 600, from(red),
                     to(#000))
}
```

Усовершенствованный фон. Листинг 6.9

```
div {
  background-size: auto;
  background-size: contain;
  background-size: cover;
  background-size: 100%;
}
```

Анимация. Листинг 6.10

```
<style>
 @-webkit-keyframes pulse {
 from {
   opacity: 0.0;
   font-size: 100%;
 }
 to {
   opacity: 1.0;
   font-size: 200%;
 }
}

div {
  -webkit-animation-name: pulse;
  -webkit-animation-duration: 2s;
  -webkit-animation-iteration-count: infinite;
  -webkit-animation-timing-function: ease-in-out;
  -webkit-animation-direction: alternate;
}
</style>
<div>
```

```
 PULSE
</div>
```

Переходы. Листинг 6.11

```
#box.left {
  margin-left: 0;
}
#box.right {
  margin-left: 1000px;
}

document.getElementById('box').className = 'left';
document.getElementById('box').className = 'right';
```

Постепенный переход. Листинг 6.12

```
#box {
  -webkit-transition: margin-left 1s ease-in-out;
}
```

Трансформация. Листинг 6.13

```
#threed-example {
  -webkit-transform: rotateZ(5deg);
  -webkit-transition: -webkit-transform 2s ease-in-out;
}
#threed-example:hover {
  -webkit-transform: rotateZ(-5deg);
}
```

7. Основы javaScript

Для того, чтобы влиять на элеметны HTML, приходится создавать ссылки на них. В CSS реализована мощная система селекторов и фильтров, поддержки которой до недавнего времени в javaScript не существовало: методов **getElementsById**, **getElementsByName** и **getElementsByClassName** недостаточно для поддержки должного уровня интеграции и спецификации HTML5. Поэтому, для работы с селекторами, были внедрены альтернативные методы: querySelector() и querySelectorAll().

Использование селектора querySelector(). Листинг 7.1

```
<script>
 function clickme(){
  document.querySelector("#main p:first-child").onclick =
showalert;
 }
 function showalert(){
  alert('Ты на мне щелкнул');
 }
 window.onload = clickme;
</script>
```

Если селектор возвращает множество элементов, querySelector() вернет только первый элемент из группы.

В отличии от предыдущего метода, querySelectorAll() возвращает все элементы, соответствующие группе элементов.

Использование селектора querySelectorAll(). Листинг 7.2

```
<script>
 function clickme(){
  document.querySelectorAll("#main p").onclick = showalert;
 }
 function showalert(){
  alert('Ты на мне щелкнул');
 }
 window.onload = clickme;
</script>
```

Обычно данный метод используется для выбора сразу нескольких элементов. Для прохождения по списку элементов, возвращенных методом, удобно использовать цикл for.

Выбор всех элементов, обнаруженных методом querySelectorAll(). Листинг 7.3

```
<script>
 function clickme(){
  var list = document.querySelectorAll("#main p");
  for(var f=0; f<list.lenght; f++){
   list[f].onclick = showalert;
  }
 }
 function showalert(){
  alert('Ты на мне щелкнул');
 }
 window.onload = clickme;
</script>
```

Теперь все элементы <p> внутри будут реагировать на щелчок.

JavaScript также позволяет комбинировать методы.

Комбинирование методов выбора. Листинг 7.4

```
  var list = document.qetElemntById("main").querySelectorAll("p");
```

Прослушиватель событий

Для связки события и исполняемой функции предлагается использование специального прослушивателя addEventListener(), который принимает три аргумента: тип события, исполняемую функцию и булево значение логического типа.

Добавление обработчиков событий с помощью addEventListener(). Листинг 7.5

```
<script>
 function showalert(){
  alert("Ты на мне щелкнул");
 }
 function clickme(){
  var pelement = document.getElementsByTagName("p")[0];
  pelement.addEventListener("click", showalert, false);
 }
 window.addEventListener("load", clickme, false);
</script>
<div>
 <p>Щелки сюда</p>
 <p>Еще один блок текста, который не реагирует на щелчек</p>
</div>
```

Рассмотрим подробнее последний атрибут метода addEventListener(), принимающий значение true либо false. Данный атрибут указывает, каким образом будут обрабатываться множественные события. Например, мы прослушиваем событие click на двух вложенных элементах (один элемент вложен во второй). Когда пользователь щелкает на этих методах, два события срабатывают в порядке определяемом этим атрибутом. Если значение одного из атрибутов равно true, то считается, что событие для этого элемента должно произойти первым, а для другого элемента — вторым. Обычно, для большинства ситуаций, хватает значения false.

8. Видео и аудио

HTML5, наконец-то представил элемент, предназначенный для вставки видео в документы html. Элементу <video> для отображения видео требуется лишь указать несколько несложных параметров. Обязательным атрибутом элемента <video> является только атрибут src.

Базовый синтаксис элемента <video>. Листинг 8.1

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Video Player</title>
</head>
<body>
  <section>
    <video src="http://minkbooks.com/content/trailer.mp4"
controls>
    </video>
  </section>
</body>
</html>
```

Теоретически, кода из этого листинга должно быть достаточно. Однако, на практике, чтобы все браузеры проигрывали видео, необходимо предоставлять, как минимум два файла в разных форматах: OGG и MP4. Проблема в том, что, хотя элемент <video> и стандартизован, стандартного формата видео не существует. Такие браузеры, как Sofary и Internet Exploer поддерживают формат коммерческой лицензии MP4. Google Chrome, Firefox и Opera поддерживает свободно распространяемый формат OGG.
Проблему разных форматов пытаются решить путем введения нового формата WEBM, который, пока еще не все браузеры понимают. Поэтому, пока приходится для одного видеопроигрывателя, указывать источник с тремя расширениями.

Работающий в разных браузерах видеопроигрыватель. Листинг 8.2

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Video Player</title>
</head>
<body>
  <section>
    <video width="720" height="400" controls>
      <source src="files/trailer.mp4">
      <source src="files/trailer.ogg">
      <source src="files/trailer.webm">
    </video>
  </section>
</body>
</html>
```

В предыдущих листингах, мы использовали атрибут **controls**, который отображает элементы управления видео, предоставляемые самим браузером. Рассмотрим другие медиа-атрибуты.

Атрибуты видео и дочерние теги source
Для элементов audio и video введено несколько атрибутов, определяющих то, как браузер будет представлять медиаконтент конечному пользователю:

- **src** указывает один медийный файл для проигрывания (о нескольких источниках с разными кодеками, пожалуйста, см. ниже);
- **poster** — URL изображения, которое будет показываться до нажатия пользователем кнопки Play (только для video);
- **preload** определяет, как и когда браузер загрузит медийный файл. Возможны три значения: **none** (видео не скачивается, пока пользователь не запускает проигрывание), **metadata** (сообщает браузеру скачивать ровно столько данных, чтобы можно было определить высоту и ширину кадров, а также длительность медиаролика) и **auto** (позволяет браузеру самому решать, какой объем видео нужно скачивать до запуска проигрывания пользователем);

- **autoplay** — булев атрибут, используемый для запуска видеоролика сразу после загрузки страницы (мобильные устройства часто игнорируют этот атрибут для экономии пропускной полосы);
- **loop** — булев атрибут, вызывающий повторное воспроизведение видео по достижении конца записи;
- **muted** — булев атрибут, указывающий, нужно ли запускать видео с выключенным звуком;
- **controls** — булев атрибут, указывающий, должен ли браузер выводит свои элементы управления;
- **width и height** задают воспроизведение видеоролика с определенным размером (только для video, значения не могут быть в процентах).

Работающий в разных браузерах видеопроигрыватель. Листинг 8.3

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Video Player</title>
</head>
<body>
  <section>
    <video width="720" height="400" preload controls loop
poster="http://minkbooks.com/content/poster.jpg">
      <source src="http://minkbooks.com/content/trailer.mp4">
      <source src="http://minkbooks.com/content/trailer.ogg">
    </video>
  </section>
</body>
</html>
```

События API видео

В HTML5 появились новые события, информирующие о состоянии мультимедиа, например, какая доля видео уже загружена, завершилось ли воспроизведение файла, остановлено ли видео и т.д. Рассмотрим основные мультимедийные события:

- **progress**. Это событие периодические информирует о прогрессе загрузки файла.
- **canplaythrought**. Срабатывает в момент, когда становится понятно, что медиа-файл можно воспроизвести целиком, без задержек.
- **canplay**. Срабатывает, когда медиа-файл готов к воспроизведению.
- **ended**. Срабатывает, когда заканчивается воспроизведение.
- **pause**. Срабатывает, когда пользователь приостанавливает воспроизведение.
- **error**. Срабатывает при возникновении ошибки. Событие доставляется в элемент <source>, если такой существует.

Методы API видео

Медиа-объекты HTML5 также включают следующие методы, применяемые

при написании скриптов:

- play пытается загрузить и воспроизвести видео;
- pause останавливает проигрывание текущего видеоролика;
- canPlayType(type) распознает, какие кодеки поддерживает браузер. Если вы посылаете некий тип вроде video/mp4, браузер ответит строкой probably, maybe, no или пустой строкой;
- load вызывается для загрузки нового видео, если вы изменяете атрибут src.

Свойства API видео

- **paused**. Возвращает значение true, если воспроизведение мультимедиа приостановлено или еще не началось.
- **ended**. Возвращает значение true, если видео было воспроизведено до конца.
- **duration**. Возвращает продолжительность мультимедиа в секундах.
- **currentTime**. Может как возвращать, так и принимать значение. Это свойство или информирует о текущей позиции воспроизведения файла, или устанавливает новую позицию, с которой продолжается воспроизведение.
- **error**. Возвращает значение ошибки, если произошел сбой.
- **buffered**. Предоставляет информацию о том, какая часть файла уже загружена в буфер. Возвращаемое значение представляет собой массив, содержащий данные обо всех загруженных фрагментах мультимедиа. Если пользователь переходит к части медиафайла, которая еще не была загружена, браузер продолжает загрузку с этой позиции. Для прохода по элементам массива можно использовать атрибуты end() и start(). Например, код buffered.end(0) вернет продолжительность первой загруженной части файла, содержащейся в буфере.

Наглядно ознакомиться со свойствами, событиями и методами можно по этой ссылке: http://www.w3.org/2010/05/video/mediaevents.html

Программирование видео-проигрывателя

Если нас не устраивает дизайн, либо функциональность проигрывателя, который предлагается браузером по-умолчанию, то мы можем запрограммировать собственный видео-проигрыватель.

Рассмотрим html-документ проигрывателя.

HTML-документ проигрывателя. Листинг 8.4

```
<html lang="ru">
<head>
  <title>Video Player</title>
  <link rel="stylesheet" href="player.css">
  <script src="player.js"></script>
</head>
```

```
<body>
  <section id="player">
    <video id="media" width="720" height="400">
      <source src="http://minkbooks.com/content/trailer.mp4">
      <source src="http://minkbooks.com/content/trailer.ogg">
    </video>
    <nav>
      <div id="buttons">
        <input type="button" id="play" value="Play">
        <input type="button" id="mute" value="Mute">
      </div>
      <div id="bar">
        <div id="progress"></div>
      </div>
      <div id="control">
        <input type="range" id="volume" min="0" max="1" step="0.1"
value="0.6">
      </div>
      <div class="clear"></div>
    </nav>
  </section>
</body>
</html>
```

В файле player.js создадим первую функцию, задача которой — инициализация переменных.

Инициализация функции iniiate(). Листинг 8.5

```
var maxim, mmedia, play, bar, progress, mute, volume, loop;
function initiate(){
  maxim = 400;
  mmedia = document.getElementById('media');
  play = document.getElementById('play');
  bar = document.getElementById('bar');
  progress = document.getElementById('progress');
  mute = document.getElementById('mute');
  volume = document.getElementById('volume');

  play.addEventListener('click', push);
  mute.addEventListener('click', sound);
  bar.addEventListener('click', move);
  volume.addEventListener('change', level);
}
```

По нажатию кнопки с идентификатором play, вызывается функция push, задача которой, либо включить видео, либо поставить на паузу.

Функция push(), которая либо запускает, либо приостанавливает видео. Листинг 8.6

```
function push(){
  if(!mmedia.paused && !mmedia.ended) {
    mmedia.pause();
    play.value = 'Play';
```

```
    clearInterval(loop);
  }else{
    mmedia.play();
    play.value = 'Pause';
    loop = setInterval(status, 1000);
  }
}
```

Если значения mmedia.paused и mmedia.ended равны false, значит видео воспроизводится, и тогда вызывается метод pause(), который останавливает воспроизведение. Текст на кнопке меняется на «Play». С помощью встроенного метода clearInterval очищается цикл.
Если же истины противоположные условия, то видео или стоит на паузе, или воспроизведение завершилось. Тогда условный оператор возвращает метод play(), воспроизводящий видео с начала, или с того момента, где оно было приостановлено. В этом случае, мы выполняем еще одно важное действие: определяем время с помощью метода setInterval(), вызывая функцию status каждую секунду.
setInterval() — это встроенный javaScript метод, который имеет два входящих параметра: первый параметр — функция, второй параметр — количество миллисекунд. Второй параметр указывает через какой отрезок времени должна вызываться функция, определяемая первым параметром.

var loop = setInterval('alert("прошла секунда")', 1000)

Функция setInterval() возвращает уникальный идентификатор, который мы помещаем в переменную loop. Остановить работу данной функции можно только при помощи функции clearInterval(), которая принимает единственный параметр — идентификатор функции setInterval().

clearInterval(loop)

За обновление статуса прогресс-бара отвечает функция status().

Функция status(). Листинг 8.7

```
function status(){
  if(!mmedia.ended){
    var size = parseInt(mmedia.currentTime * maxim /
mmedia.duration);
    progress.style.width = size + 'px';
  }else{
    progress.style.width = '0px';
    play.value = 'Play';
    clearInterval(loop);
  }
}
```

Функция status() вызывается каждую секунду, пока видео воспроизводится. В этой функции также присутствует условный оператор if, проверяющий статус воспроизведения. Если воспроизведение файла не достигло конца, т.е. Свойство ended возвращает значение false, то мы вычисляем требуемую длину индикатора прогресса в пикселах.
Поскольку функция status(), во время воспроизведения видео вызывается каждую секунду, значение позиции воспроизведения (кол-во секунд с начала воспроизведения видео) постоянно меняется. Это значение извлекается через свойство curentTime. Мы также знаем, через свойство duration продолжительность видео, и максимальный размер индикатора прогресса, сохраненный в переменной maxim. Имея эти три значения, несложно вычислить длину индикатора прогресса в пикселах, указывающего сколько секунд видео уже воспроизведено. Данная формула:

Текущая позиция времени x Максимальная длина / Общая продолжительность

позволяет перевести секунды воспроизведения в пикселы, и соответствующим образом изменит индикатор прогресса.
Если же условие равно true, т.е. воспроизведение закончилось, то мы устанавливаем нулевой размер индикатора прогресса. Меняем текст на кнопке на «Play». Очищаем цикл с помощью clearInterval(). При этом, переодический вызов функции status() отменяется.

Каждый раз, когда пользователь щелкает по индикатору прогресса, выполняется функция move().

Функция move(). Листинг 8.8

```
function move(e){
  if(!mmedia.paused && !mmedia.ended){
    var mouseX = e.pageX - bar.offsetLeft;
    var newtime = mouseX * mmedia.duration / maxim;
    mmedia.currentTime = newtime;
    progress.style.width = mouseX + 'px';
  }
}
```

Прослушиватель события click добавляется к элементу bar, для проверки, не щелкнул ли пользователь по индикатору прогресса, чтобы начать воспроизведение с новой позиции. Здесь также имеется условный оператор if, который проверяет, воспроизводится ли видео.
Для начала определим точное местоположение мыши, в котором произошел щелчок. Мы воспользовались значением свойства pageX, которое возвращает значение указывающее точку в системе координат всей страницы. Для того, чтобы узнать расстояние между началом индикатора прогресса и указателем мыши, необходимо вычесть из значения pageX расстояние между началом

страницы и началом полосы индикатора. Получить значение начала полосы индикатора можно с помощью свойства offsetLeft. Таким образом, формула

mouseX = e.pageX – bar.offsetLeft

возвращает точное положение указателя мыши относительно начала полосы индикатора прогресса.
Получив точное положение указателя мыши, мы можем преобразовать его в секунды. Для этого, необходимо положение указателя мыши умножить на длительность видео файла и разделить на максимальный размер полосы индикатора:

newtime = mouseX x video.duration / maxim

Управление звуком осуществляется в функции sound()

Функция sound(). Листинг 8.9

```
function sound(){
  if(mute.value == 'Mute'){
    mmedia.muted = true;
    mute.value = 'Sound';
  }else{
    mmedia.muted = false;
    mute.value = 'Mute';
  }
}
```

Управление громкостью:

Функция level(). Листинг 8.10

```
function level(){
  mmedia.volume = volume.value;
}
```

Наконец, чтобы запустить приложение, добавим прослушиватель события.

По загрузке страницы, вызываем функцию initiate(). Листинг 8.11

```
addEventListener('load', initiate);
```

Добавим стилей к нашему приложению.

Файл player.css. Листинг 8.12

```
body{
  text-align: center;
}
header, section, footer, aside, nav, article, figure, figcaption,
hgroup{
  display: block;
```

```
}
#player{
  width: 720px;
  margin: 20px auto;
  padding: 10px 5px 5px 5px;
  background: #999999;
  border: 1px solid #666666;
  border-radius: 10px;
}
#play, #mute{
  padding: 2px 10px;
  width: 65px;
  border: 1px solid #000000;
  background: #DDDDDD;
  font-weight: bold;
  border-radius: 10px;
}
nav{
  margin: 5px 0px;
}
#buttons{
  float: left;
  width: 135px;
  height: 20px;
  padding-left: 5px;
}
#bar{
  float: left;
  width: 400px;
  height: 16px;
  padding: 2px;
  margin: 2px 5px;
  border: 1px solid #CCCCCC;
  background: #EEEEEE;
}
#progress{
  width: 0px;
  height: 16px;
  background: rgba(0,0,150,.2);
}
.clear{
  clear: both;
}
```

Получили достаточно симпатичный плеер:

Отображение текстовых элементов в течение определенного времени
В браузерах также начинают реализовывать элемент track, который поддерживает в видеороликах субтитры, скрытые титры (closed captions), переводы (translations) и комментарии. Вот элемент video с дочерним элементом track:

Добавление субтитров. Листинг 8.13

```
<video id="video1" width="640" height="360" preload="none"
controls>
      <track src="subtitles.vtt" srclang="en" kind="subtitles"
label="English subtitles">
</video>
```

В этом примере задействованы четыре из пяти возможных атрибутов элемента <track>

- **src** — ссылка на файл либо в формате Web Video Timed Text (WebVTT), либо в формате Timed Text Markup Language (TTML);
- **srclang** — язык TTML-файла (например, en, es или ar);
- **kind** указывает тип текстового контента: субтитры, заголовки, описания, главы или метаданные;
- **label** хранит текст, отображаемый пользователю, который выбирает трек;
- **default** — булев атрибут, определяющий стартовый элемент track.

WebVTT — это простой текстовый формат, который начинается с однострочного объявления (WEBVTT FILE), а затем перечисляет время начала и конца; в качестве разделителя используются символы -->, а за временем

начала и конца указывается текст, отображаемый в этот интервал времени. Вот простой WebVTT-файл, который отображает две строки текста в два разных интервала времени:

Пример WebVTT файла. Листинг 8.14

```
WEBVTT

00:02.000 --> 00:07.000
Welcome
to the &lt;track&gt; element!

00:10.000 --> 00:15.000
This is a simple example.

00:17.000 --> 00:22.000
Several tracks can be used simultaneously

00:22.000 --> 00:25.000
to provide text in different languages.

00:27.000 --> 00:30.000
Good bye!
```

Кроме того, в WebVTT файл мы можем включать HTML тэги:

Пример WebVTT файла с HTML тэгами. Листинг 8.15

```
WEBVTT

00:02.000 --> 00:07.000
<i>Welcome</i>
to the &lt;track&gt; element!

00:10.000 --> 00:15.000
<v Robert>This is a simple <c.captions>example</c>.

00:17.000 --> 00:22.000
<v Martin>Several tracks can be used simultaneously

00:22.000 --> 00:25.000
<v Martin>to provide text in different languages.

00:27.000 --> 00:30.000
<b>Good bye!</b>
```

Рассмотрим способ добавления стилей к файлу WebVTT:

Добавление стилей к WebVTT. Листинг 8.16

```
::cue(.captions){
  color: #990000;
}
```

Полноэкранный режим воспроизведения:

Программирование полноэкранного режима. Листинг 8.17

```
<html lang="en">
<head>
  <title>Full Screen</title>
  <script>
    var video;
    function initiate(){
      video = document.getElementById('media');
      video.addEventListener('click', gofullscreen);
    }
    function gofullscreen(){
      if(!document.webkitfullscreenElement){
        video.webkitRequestFullscreen();
        video.play();
      }
    }
    addEventListener('load', initiate);
  </script>
</head>
<body>
  <section>
    <video id="media" width="720" height="400" controls>
      <source src="http://minkbooks.com/content/trailer.mp4">
      <source src="http://minkbooks.com/content/trailer.ogg">
    </video>
  </section>
</body>
</html>
```

API аудио поддерживает те же свойства, события и методы, что и API видео. Только они применяются к элементу <audio> Для кроссбраузерного воспроизведения аудио-файлов, необходимо использовать два аудио-формата: ogg и mp3:

Добавление стилей к WebVTT. Листинг 8.18

```
<html lang="ru">
<head>
  <title>Audio Player</title>
</head>
<body>
  <section id="player">
    <audio id="media" controls>
      <source src="http://minkbooks.com/content/beach.mp3">
      <source src="http://minkbooks.com/content/beach.ogg">
    </audio>
  </section>
</body>
</html>
```

9. API холст

API canvas (холст) позволяет рисовать графические элементы, выводить на экран изображения из файла, анимировать и обрабатывать рисунки и текст. Используя его совместно с другими API можно создавать двухмерные и даже трехмерные игры для Сети.
Элемент <canvas> создает пустой прямоугольник, внутри которого визуализируются результаты применения методов рисования.

Включение элемента <canvas> на страницу. Листинг 9.1

```
<html lang="en">
<head>
  <title>Canvas API</title>
  <script src="canvas.js"></script>
</head>
<body>
  <section id="canvasbox">
    <canvas id="canvas" width="500" height="300"></canvas>
  </section>
</body>
</html>
```

Для подготовки элемента <canvas> к рисованию, сперва необходимо вызвать метод getContext().

Создание контекста рисования для холста. Листинг 9.2

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
}
addEventListener("load", initiate);
```

В листинге мы сохранили ссылку на элемент <canvas> в переменной elem. Затем для данного элемента создали контекст, вызвав метод getContext('2d'). Этот метод принимает одно из двух значений 2d или 3d, описывающих соответственно 2d и 3d-мерные среды рисования.

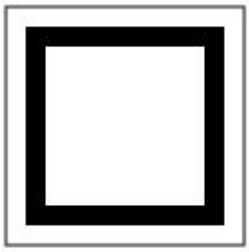
Рисование прямоугольников
Для рисования прямоугольников доступны следующие методы

- **fillRect(x, y, width, height)** предназначен для рисования прямоугольника залитого цветом. Верхний левый угол фигуры будет находиться в точке заданной атрибутами x и y.
- **strokeRect(x, y, width, height)** аналогичен предыдущему, но создает пустой, не залитый цветом, прямоугольный контур.
- **clearRect(x, y, width, height)** предназначен для вычитания прямоугольной области, работает как прямоугольный ластик.

Применяя эти методы, нарисуем прямоугольник:

Рисуем прямоугольник. Листинг 9.3

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.strokeRect(100, 100, 120, 120);
  canvas.fillRect(110, 110, 100, 100);
  canvas.clearRect(120, 120, 80, 80);
}
addEventListener("load", initiate);
```



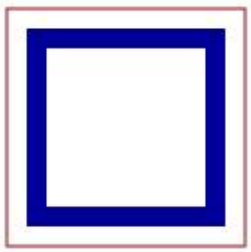
Цвет

Для определения свойства цвета можно применять синтаксис CSS со следующими свойствами:

- **strokeStyle.** Определяет цвет линий фигуры.
- **fillStyle.** Определяет цвет внутренней области фигуры.
- **globalAlpha**. Устанавливает уровень прозрачности.

Добавляем цвет. Листинг 9.4

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.fillStyle = "#000099";
  canvas.strokeStyle = "#990000";

  canvas.strokeRect(100, 100, 120, 120);
  canvas.fillRect(110, 110, 100, 100);
  canvas.clearRect(120, 120, 80, 80);
}
addEventListener("load", initiate);
```



Градиент

Также, как и в CSS3, градиенты могут быть линейными и радиальными. Возможно установление нескольких цветовых установок, создающие плавные переходы между множеством цветов. Методы:

- **createLinearGradient(x1, y1, x2, y2)** создает объект градиента для последующей визуализации на холсте.
- **createRadialGradient(x1, y1, r1, x2, y2, r2)** создает объект градиента, состоящий из двух окружностей. Значения в скобках представляют собой координаты центров окружностей и их радиусы.
- **addColorStop(position, color)** – определяет цвета, которые будут использоваться для создания градиента. Атрибут position — это значение от 0,0 до 1,0, определяющее, в какой позиции начинается затухание цвета color.

Градиентная заливка. Листинг 9.5

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  var grad = canvas.createLinearGradient(0, 0, 10, 100);
  grad.addColorStop(0.5, '#00AAFF');
  grad.addColorStop(1, '#000000');
  canvas.fillStyle = grad;

  canvas.fillRect(10, 10, 100, 100);
  canvas.fillRect(150, 10, 200, 100);
}
addEventListener("load", initiate);
```

Создание путей

Путь — это контур, вдоль которого следует перо, оставляя след. Путь может включать в себя различные виды штрихов: прямые линии, дуги, прямоугольники и т.д.

Рассмотрим два метода, предназначенные для создания путей и их закрытия:

- **beginPath()**. Начинает новую фигуру.
- **closePath()**. Закрывает путь, добавляя прямую линию между текущей точкой и исходной точкой пути.

Методы визуализации путей на холсте:

- **stroke()**. Визуализирует путь в виде контура.
- **fill()**. Визуализирует путь в виде залитой цветом фигуры.
- **clip()**. Определяет область обрезки для контекста. Данный метод позволяет задать область обрезки произвольной формы, создав маску. Всё, что остается за пределами маски на странице не отображается.

Базовые правила для определения пути. Листинг 9.6

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
  canvas.beginPath();
  // здесь описывается путь.
  canvas.stroke();
}
addEventListener("load", initiate);
```

Данный код не создает никаких рисунков. Он лишь сигнализирует о создании путей.

Для описания путей и создания реальной фигуры предназначены следующие методы:

- **moveTo(x, y)**. Перемещает кончик пера в указанную позицию.
- **lineTo(x, y)**. Создает отрезок между двумя точками: текущей позицией (например, определенной с помощью метода moveTo) и точкой с координатами x и y.
- **rect(x, y, width, height)**. Создает прямоугольник, который не сразу визуализируется на холсте, а становится частью пути.
- **arc(x, y, radius, startAngle, endAngle, direction)**. Создает дугу или окружность с центром в точке x, y, радиусом и угловым значением объявленным в атрибутах. Последний аргумент — это булево значение, задающее направление рисования: по часовой стрелке или против нее.
- **quadraticCurveTo(cpx, cpy, x, y)**. Создает квадратичную кривую Безье, начинающуюся в верхней позиции пера и заканчивающуюся в позиции с координатами x и y. Атрибуты cpx и cpy — это контрольные точки, управляющие формой кривой.

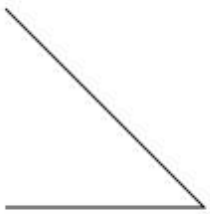
- **bizierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)**. Аналогичен предыдущему, но имеет два дополнительных аргумента, позволяющих определить кубическую кривую Бизье.

Создадим фигуру с помощью описанных методов:

Угол. Листинг 9.7

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.stroke();
}
addEventListener("load", initiate);
```

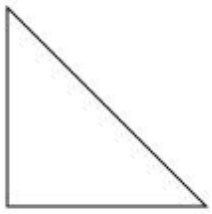
Нарисовали угол:



Далее этот угол можно закрыть, превратив в треугольник:

Треугольник. Листинг 9.8

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.closePath();
  canvas.stroke();
}
addEventListener("load", initiate);
```



Чтобы залить треугольник цветом, необходимо использовать метод fill().

Заливка треугольника цветом по умолчанию. Листинг 9.9

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.fill();
}
addEventListener("load", initiate);
```

Рассмотрим метод clip(), который предназначен для создания маски в форме пути, и таким образом, позволяет определить, что будет нарисовано, а что нет.

Использование треугольника в качестве маски. Листинг 9.10

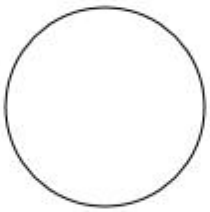
```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
  canvas.beginPath();
  canvas.moveTo(100, 100);
  canvas.lineTo(200, 200);
  canvas.lineTo(100, 200);
  canvas.clip();
  for(var f = 0; f < 300; f = f + 10){
    canvas.moveTo(0, f);
    canvas.lineTo(500, f);
  }
  canvas.stroke();
}
addEventListener("load", initiate);
```

Цикл for() из листинга создает горизонтальные линии через каждые десять пикселов. Линии пересекают холст слева направо, но на странице мы видим только те фрагменты, которые попадают внутрь треугольной маски.

Для создания фигур, включающих в себя различные дуги, в API предусмотрены специальные методы.
Метод arc() предназначен для рисования окружностей или дуг. Обратите внимание на значение PI (данный метод ориентируется на значение угла в радианах, а не в градусах). Значение PI в радианах соответствует 180°. Формула **PI x 2** в итоге дает 360°.

Окружность. Листинг 9.11

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
  canvas.beginPath();
  canvas.arc(100, 100, 50, 0, Math.PI * 2, false);
  canvas.stroke();
}
addEventListener("load", initiate);
```



Для создания дуги с определенным углом в градусах, нужно воспользоваться формулой:

Math.PI/180 x градусы

Дуга с углом в 45°. Листинг 9.12

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');
  canvas.beginPath();
  var radians = Math.PI / 180 * 45;
  canvas.arc(100, 100, 50, 0, radians, false);
```

```
  canvas.stroke();
}
addEventListener("load", initiate);
```

Рассмотрим работу метода quadraticCurveTo(), который генерирует квадратичную кривую Безье, и метод bezieCurveTo(), предназначенный для рисования кубической кривой Бизье.

Сложные кривые. Листинг 9.13

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.beginPath();
  canvas.moveTo(50, 50);
  canvas.quadraticCurveTo(100, 125, 50, 200);
  canvas.moveTo(250, 50);
  canvas.bezierCurveTo(200, 125, 300, 125, 250, 200);
  canvas.stroke();
}
addEventListener("load", initiate);
```

Ширину, вид и окончание линий можно настраивать. Для этого имеется четыре свойства:

- **lineWidth**. Определяет толщину линии.
- **lineCap**. Определяет форму окончания линии. Может принимать следующие значения: **butt**, **round** или **square**.
- **lineJoin**. Определяет форму соединения двух линий. Возможные значения: **round**, **bevel** и **miter**.
- **miterLimit**. Используется совместно со свойством lineJoin и определяет протяженность соединения двух линий в случае, если свойству lineJoin присвоено значение miter.

Перечисленные свойства влияют на весь путь. После каждого изменения характеристик линии необходимо создавать новый путь.

Свойства линий. Листинг 9.14

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.beginPath();
  canvas.arc(200, 150, 50, 0, Math.PI * 2, false);
  canvas.stroke();

  canvas.lineWidth = 10;
  canvas.lineCap = "round";
  canvas.beginPath();
  canvas.moveTo(230, 150);
  canvas.arc(200, 150, 30, 0, Math.PI, false);
  canvas.stroke();

  canvas.lineWidth = 5;
  canvas.lineJoin = "miter";
  canvas.beginPath();
  canvas.moveTo(195, 135);
  canvas.lineTo(215, 155);
  canvas.lineTo(195, 155);
  canvas.stroke();
}
addEventListener("load", initiate);
```

Текст

Для добавления текста нужно определить несколько свойств и вызвать подходящий метод.

Свойства Текста:

- **font**. Синтаксис аналогичен CSS-ситнаксису свойства font
- **textAlign**. Возможные варианты выравнивания по горизонтали. Описываются значениями **start, end, left, right и center**.
- **textBaseLine**. Выравнивание по вертикали. Возможные значения: **top, hanging, middle, alphabetic, ideographic** и **bottom**.

Для вывода текста на холст используются два метода:

- **strokeText(text, x, y [, max-size])**. Текст выводится в точках x, y. Возможено передавать четвертый параметр, определяющий максимальный размер текста.
- **fillText(text, x, y)**. Аналогичен предыдущему методу, но визуализирует текст, как залитые цветом фигуры.

Рисование текста. Листинг 9.15

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.font = "bold 24px verdana, sans-serif";
  canvas.textAlign = "start";
  canvas.fillText("Hello Word!", 100, 100);
}
addEventListener("load", initiate);
```

Hello Word!

Для работы с текстом еще есть один метод, который измеряет текст, - **measureText()**. Он возвращает информацию о размере указанного текста. Благодаря методу measureText() и свойству width можно узнать длину текста по горизонтали.

Измерение текста. Листинг 9.16

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.font = "bold 24px verdana, sans-serif";
  canvas.textAlign = "start";
  canvas.textBaseline = "bottom";
  canvas.fillText("My message", 100, 124);

  var size = canvas.measureText("My message");
  canvas.strokeRect(100, 100, size.width, 24);
}
addEventListener("load", initiate);
```

Тени

Тени можно создавать для любых путей и текста. Для этого предусмотрены следующие свойства:

- **shadowColor**. Цвет тени.
- **shadowOffsetX**. Указание насколько нужно отступить от объекта по горизонтали.
- **shadowOffsetY**. Указание насколько нужно отступить от объекта по вертикали.
- **shadowBlur**. Размытость тени.

Добавляем тени. Листинг 9.17

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.shadowColor = "rgba(0, 0, 0, 0.5)";
  canvas.shadowOffsetX = 4;
  canvas.shadowOffsetY = 4;
  canvas.shadowBlur = 5;

  canvas.font = "bold 50px verdana, sans-serif";
  canvas.fillText("HELLO WORD!", 100, 100);
}
addEventListener("load", initiate);
```

Трансформации

Рассмотрим пять методов трансформации:

- **translate(x, y)**. Применяется для переноса начала координат.
- **rotate(angle)**. Поворачивает холст вокруг начала координат на указанный угол.
- **scale(x, y)**. Масштабирует все нарисованные на холсте элементы.
- **transform(m1, m2, m3, m4, dx, dy)**. Применяет новую матрицу трансформаций поверх текущей, модифицируя таким образом весь холст.
- **setTransform(m1, m2, m3, m4, dx, dy)**. Отменяет текущую трансформацию и определяет новую на основе переданных в атрибуте значений.

Применим к одному тексту методы translate(), rotate() и scale().

Трансляция, поворот и масштабирование. Листинг 9.18

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 50, 20);

  canvas.translate(50, 70);
  canvas.rotate(Math.PI / 180 * 45);
  canvas.fillText("TEST", 0, 0);

  canvas.rotate(-Math.PI / 180 * 45);
  canvas.translate(0, 100);
  canvas.scale(2, 2);
  canvas.fillText("TEST", 0, 0);
}
addEventListener("load", initiate);
```

Сперва мы нарисовали текст на холсте в точке с координатами (50, 20) с размером 20 px. После этого, с помощью метода translate() перенесли начало координат в точку (50, 70) и, с помощью метода rotate() повернули холст на 45 градусов.
После этого, определенные в предыдущем шаге значения, считаются значениями по умолчанию. Поэтому, для того, чтобы вернуть текст в исходное состояние, снова вызываем rotate() с такими же, но отрицательными значениями. Наконец, с помощью метода scale() увеличиваем масштаб холста.

Каждая последующая трансформация накладывается на предыдущую. Например, если мы применим масштабирование scale(2, 2), а затем еще раз scale(2, 2), то холст увеличится в четыре раза.
Для определения характеристик матрицы используются методы trasform() и setTransform().

Кумулятивная трансформация на матрице. Листинг 9.19

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.transform(3, 0, 0, 1, 0, 0);
```

```
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 20, 20);

  canvas.transform(1, 0, 0, 10, 0, 0);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 20, 20);
}
addEventListener("load", initiate);
```

Метод setTransform() отменяет предыдущую трансформацию.

Кумулятивная трансформация на матрице с отменой предыдущих трансформаций. Листинг 9.20

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.transform(3, 0, 0, 1, 0, 0);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 20, 20);

  canvas.setTransform(1, 0, 0, 10, 0, 0);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST", 20, 20);
}
addEventListener("load", initiate);
```

Восстановление состояния

Из-за накопительного эффекта состояний трансформаций, возвращаться к начальному состоянию без специальных методов бывает затруднительно. Рассмотрим методы восстановления холста.

- **save()**. Сохраняет состояние холста, включая все определенные для него ранее трансформации, значения свойств, стилей и т.д.
  restore(). Восстанавливает последнее сохраненное состояние.

Сохранение состояний холста. Листинг 9.21

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.save();
  canvas.translate(50, 70);
  canvas.font = "bold 20px verdana, sans-serif";
  canvas.fillText("TEST1", 0, 30);
  canvas.restore();
  canvas.fillText("TEST2", 0, 30);
}
addEventListener("load", initiate);
```

TEST2

TEST1

Комбинирование фигур

Для определения, каким образом, фигуры, выводящиеся на холст, должны комбинироваться с другими фигурами, существуео свойство **globalCompositeOperation**. Рассмотрим возможные значения данного свойства:

- **source-over** – новая фигура визуализируется поверх уже имеющихся на холсте.
- **source-in** – визуализируется только та часть фигуры, которая

перекрывает предыдущую фигуру.

- **source-out** – визуализируется только та часть фигуры, которая не перекрывает предыдущую.
- **source-atop** – визуализируется только та часть фигуры, которая перекрывает предыдущую фигуру. Предыдущая фигура сохраняется целиком, но остальные фрагменты новой фигуры становятся прозрачными.
- **lighter** – визуализируются обе фигуры, но цвет перекрывающихся путей определяется путем сложения цветовых значений.
- **xor** – визуализируются обе фигуры, но перекрывающиеся фрагменты становятся прозрачными.
- **destination-over** – это противоположность значению по умолчанию. Новые фигуры визуализируются позади фигур уже добавленных на холст.
- **destination-in** – сохраняются только те фрагменты существующих фигур, которые перекрываются новой. Все остальные, включая новую фигуру становятся прозрачными.
- **destination-out** – сохраняются только те фрагменты существующих фигур, которые не перекрываются новой фигурой. Все остальные, включая новую фигуру, остаются прозрачными.
- **destination-atop** – существующие фигуры и новая фигура становятся прозрачными, за исключением тех фрагментов, где они перекрываются.
- **darker** – визуализируются обе фигуры, но цвет перекрывающихся фрагментов определяется вычитанием цветовых значений.
  copy — визуализируется только новая фигура, остальные становятся прозрачными.

Комбинирование фигур. Листинг 9.22

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  canvas.fillStyle = "#666666";
  canvas.fillRect(100, 100, 200, 80);
  canvas.globalCompositeOperation = "source-atop";

  canvas.fillStyle = "#DDDDDD";
  canvas.font = "bold 60px verdana, sans-serif";
  canvas.textAlign = "center";
  canvas.textBaseline = "middle";
  canvas.fillText("TEST", 200, 100);
}
addEventListener("load", initiate);
```

Обработка изображений

Для работы с изображениями предусмотрен только один метод: drowImage(). Возможные варианты использования:

- **drowImage(image, x, y)**. Вывод изображения в точку с координатами x и y.
- **drowImage(image, x, y, width, height)**. Таким образом можно масштабировать изображение, прежде чем его помещать в холст. **drowImage(image, x1, y1, width1, height1, x2, y2, width2, height2)**. Таким образом можно отрезать часть изображения и вывести его в указанной точке холста, одновременно поменяв размер. Значения x1 и y1 определяют координаты верхнего угла отрезаемого фрагмента изображения. Значения width1 и heigh1 задают размер этого изображения. Остальные значения (x2, y2, width2, height2) объявляют точку, в которой будет выводиться изображение и его размер.

Вставка изображения. Листинг 9.23

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas=elem.getContext('2d');

  var img = document.createElement('img');
  img.setAttribute('src',
'http://www.minkbooks.com/content/snow.jpg');
  img.addEventListener("load", function(){
    canvas.drawImage(img, 20, 20);
  });
}
addEventListener("load", initiate);
```

Т.к. холст может работать только с загруженными изображениями, мы поместили метод drowImage в анонимную функцию, которая вызывается прослушивателем addEventListener по событию load. Таким образом, метод drowImage() внутри функции выводит изображение только в после того, как загрузка завершена.

Извлечение части изображения, изменение размера. Листинг 9.24

```
function initiate(){
  var elem = document.getElementById('canvas');
  var canvas = elem.getContext('2d');

  var img = document.createElement('img');
  img.setAttribute('src',
'http://www.minkbooks.com/content/snow.jpg');
  img.addEventListener("load", function(){
    canvas.drawImage(img, 135, 30, 50, 50, 0, 0, 200, 200);
  });
}
addEventListener("load", initiate);
```

Кроме метода drowImage(), который работает непосредственно с изображением, существует еще несколько методов, работающих с данными полученного изображения. Рассмотрим три метода для обработки изображения.

- **getImageData(x, y, width, height)**. Считывает прямоугольную часть холста и преобразует ее в массив с данными.
- **putImageData(imagedata, x, y)**. Превращает данные, на которые ссылается imagedata в изображение и выводит его на холст в точку с координатами x и y. Таким образом, это противоположность методу getImageData().
- **createImageData(width, height)**. Создает данные для пустого изображения. Все пикселы пустого изображения черные пикселы.

Каждое изображение можно представить ввиде последовательности целых чисел, соответсвющих компонентам RGBA (по четыре значения на каждый пиксел). Группа значений, несущих такую информацию составляют одномерный массив. Позиция каждого из элементов массива вычисляется по формуле

(width x 4 x Y) + (X x 4) = соответствует красному цвету

Результат вычислений соответствует первому пикселу. Для получения цвета для остальных компонентов, необходимо прибавлять по единице для каждого компонента

(width x 4 x Y) + (X x 4) + 1 = зеленый
(width x 4 x Y) + (X x 4) + 2 = синий
(width x 4 x Y) + (X x 4) + 3 = альфа-канал

Генерация негатива изображения. Листинг 9.25

```
var canvas, img;
function initiate(){
  var elem = document.getElementById('canvas');
  canvas = elem.getContext('2d');

  img = document.createElement('img');
```

```
  img.setAttribute('src', 'snow.jpg');
  img.addEventListener("load", modimage);
}
function modimage(){
  canvas.drawImage(img, 0, 0);
  var info = canvas.getImageData(0, 0, 175, 262);
  var pos;
  for(var x = 0; x < 175; x++){
    for(var y = 0; y < 262; y++){
      pos = (info.width * 4 * y) + (x * 4);
      info.data[pos] = 255 - info.data[pos];
      info.data[pos+1] = 255 - info.data[pos+1];
      info.data[pos+2] = 255 - info.data[pos+2];
    }
  }
  canvas.putImageData(info, 0, 0);
}
addEventListener("load", initiate);
```

Ширина изображения в примере равна 350 px, высота — 262 px. Поэтому, передавая методу getImageData параметры (0, 0, 175, 262), мы вырезаем половину исходного изображения. Вырезанное изображение сохраняется в переменную info. Метод getImageData возвращает объект, который можно обработать, обратившись к его свойствам width, height, data.
Далее, для того, чтобы создать негатив изображения, необходимо обработать каждый пиксел исходной части изображения. Для описания каждого цвета используется значение от 0 до 255. Следовательно, чтобы получить негатив цвета, нужно вычесть из 255 значение цвета

негатив = 255 — цвет

Данные вычисления необходимо выполнить для каждого пиксела. Поэтому мы создали два цикла (один для строк, второй для столбцов).
После того, как все пикселы пройдут обработку, переменная info с новыми изображениями отправляется на холст. Обработанное изображение выводится в той же позиции, где находится исходное.

Узоры

Процедура добавления узоров аналогична работе с градиентами: нужно создать узор с помощью метода createPattern().

createPattern(image, type), где атрибут image предоставляет собой ссылку на изображение, а атрибут type может принимать одно из четырех значений: repeat, repeat-x, repeat-y или no-repeat.

Узоры на холсте. Листинг 9.26

```
var canvas, img;
function initiate(){
  var elem = document.getElementById('canvas');
  canvas = elem.getContext('2d');

  img = document.createElement('img');
  img.setAttribute('src',
'http://www.minkbooks.com/content/bricks.jpg');
  img.addEventListener("load", modimage);
}
function modimage(){
  var pattern = canvas.createPattern(img, 'repeat');
  canvas.fillStyle = pattern;
  canvas.fillRect(0, 0, 500, 300);
}
addEventListener("load", initiate);
```

Анимация на холсте

Для анимирования объектов на холсте не существует ни специальных методов, ни четко определенной последовательности действий. Нарисованные объекты на холсте передвинуть нельзя. Строить анимированное изображение можно одним способом: стирая часть изображения, и строя новые фигуры. Рассмотрим простой пример, в котором будем очищать холст методом clearRect() и снова рисовать на нем фигуры.

Анимация. Листинг 9.27

```
var canvas;
function initiate(){
  var elem = document.getElementById('canvas');
  canvas = elem.getContext('2d');
  addEventListener('mousemove', animation);
}
function animation(e){
  canvas.clearRect(0, 0, 700, 300);

  var xmouse = e.clientX;
  var ymouse = e.clientY;
  var xcenter = 220;
  var ycenter = 150;
  var ang = Math.atan2(ymouse - ycenter,xmouse - xcenter);
  var x = xcenter + Math.round(Math.cos(ang) * 10);
  var y = ycenter + Math.round(Math.sin(ang) * 10);

  canvas.beginPath();
  canvas.arc(xcenter, ycenter, 20, 0, Math.PI * 2, false);
  canvas.moveTo(xcenter + 70, 150);
  canvas.arc(xcenter + 50, ycenter, 20, 0, Math.PI * 2, false);
  canvas.stroke();

  canvas.beginPath();
  canvas.moveTo(x + 10, y);
  canvas.arc(x, y, 10, 0, Math.PI * 2, false);
  canvas.moveTo(x + 60, y);
  canvas.arc(x + 50, y, 10, 0, Math.PI * 2, false);
  canvas.fill();
}
addEventListener("load", initiate);
```

Мы создали рисунок глаз, следящих за указателем мыши. Для перемещения зрачков обновляем позицию соответствующих элементов каждый раз, когда указатель мыши сдвигается. Для этого в функции initiate() используется прослушиватель событий mousemove, который вызывает функцию animation(). Выполнение функции начинается с очистки холста инструкцией clearRect(0, 0, 300, 500). После этого считывается позиция указателя мыши, а в переменных xcenter и ycenter сохраняется местоположение первого глаза.
После инициализации переменных, вычисляем угол наклона невидимого отрезка, соединяющего две эти точки. Для этого используется стандартный метод atan2.

Math.atan2(y, x)

Метод **atan2** возвращает числовое значение между -PI и PI, представляющее собой угол Theta для точки (x,y). Это угол, отсчитываемый против часовой стрелки и измеряемый в радианах, между положительным лучом оси X и

точкой (x,y). Заметим, что порядок аргументов у этой функции такой, что координата по Y передается первой, а по X - второй.
Методу atan2 передаются отдельно значения x и y, а (atan) - отношение этих двух аргументов.
Затем, на основе угла, по формуле

xcenter + Math.round(Math.sin(ang)x10)

вычисляем точные координаты центра зрачка. Число 10 — это расстояние от центра глаз до центра зрачка
Получив нужные значения, рисуем на холсте глаза. Первый путь объединяет две окружности — получим глаза. Первый метод arc() рисует окружность с координатами xcenter и ycenter. Второй вызов метода arc() создает аналогичную окружность на 50 пикселов правее первой, для чего ему передается инструкция arc(xcenter+50, 150, 20, 0, Math.PI*2, false).
Анимированная часть рисунка определяется вторым путем. Для создания этого пути используются переменные x и y со значениями, вычисленными ранее на основе величины угла. Оба зрачка визуализируются как черные круги с помощью метода fill().

Процесс повторяется при каждом срабатывании события mouseover.

Обработка видео на холсте
Как и в случае с анимацией, не существует специальных методов для отображения видео на холсте. Единственный способ воспроизвести видео — это считать все кадры из элемента <video> и нарисовать их на холсте, как отдельные изображения, используя drowImage().
Таким образом, для видео всего лишь необходимо скомбинировать техники, которые мы узнали ранее.

Перевернутое видео на холсте. Листинг 9.28

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Video on Canvas</title>
  <script>
    var canvas, video;
    function initiate(){
      var elem = document.getElementById('canvas');
      canvas = elem.getContext('2d');
```

```
      video = document.getElementById('media');

      canvas.translate(483, 0);
      canvas.scale(-1, 1);

      setInterval(processFrames, 33);
    }
    function processFrames(){
      canvas.drawImage(video, 0, 0);
    }
    addEventListener("load", initiate);
  </script>
</head>
<body>
  <section>
    <video id="media" width="483" height="272" autoplay>
      <source src="http://www.minkbooks.com/content/trailer2.mp4">
      <source src="http://www.minkbooks.com/content/trailer2.ogg">
    </video>
  </section>
  <section>
    <canvas id="canvas" width="483" height="272"></canvas>
  </section>
</body>
</html>
```

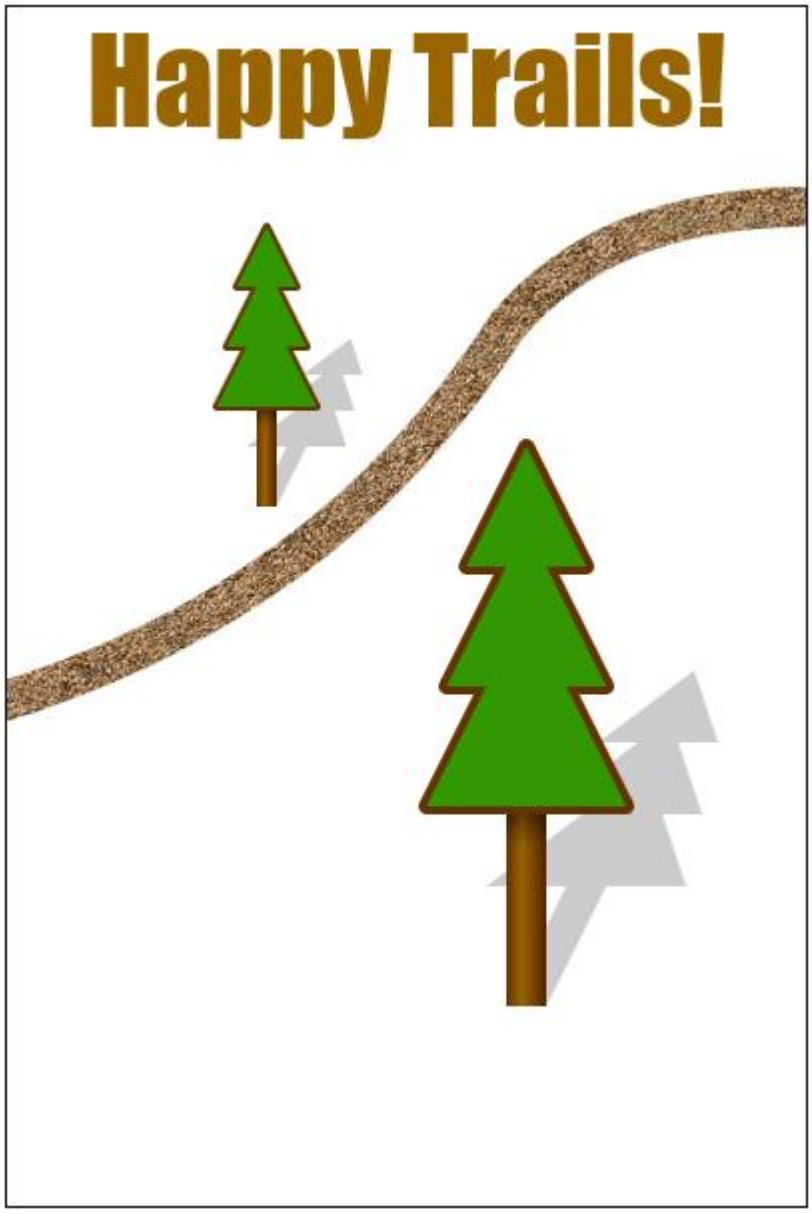
Конечный пример

Конечный пример. Листинг 9.29

```
<!DOCTYPE html>
<html>
  <title>Final example</title>

  <canvas id="trails" style="border: 1px solid;"  width="400"
height="600"> </canvas>
  <script>
        var gravel = new Image();
        gravel.src = "gravel.jpg";
        gravel.onload = function () {
            drawTrails();
        }

        function createCanopyPath(context) {
            context.beginPath();
            context.moveTo(-25, -50);
            context.lineTo(-10, -80);
            context.lineTo(-20, -80);
            context.lineTo(-5, -110);
            context.lineTo(-15, -110);

            context.lineTo(0, -140);

            context.lineTo(15, -110);
```

```
            context.lineTo(5, -110);
            context.lineTo(20, -80);
            context.lineTo(10, -80);
            context.lineTo(25, -50);
            context.closePath();
        }

        function drawTree(context) {
            context.save();
            context.transform(1, 0,
                              -0.5, 1,
                              0, 0);
            context.scale(1, 0.6);
            context.fillStyle = 'rgba(0, 0, 0, 0.2)';
            context.fillRect(-5, -50, 10, 50);
            createCanopyPath(context);
            context.fill();
            context.restore();

            var trunkGradient = context.createLinearGradient(-5, -
50, 5, -50);
            trunkGradient.addColorStop(0, '#663300');
            trunkGradient.addColorStop(0.4, '#996600');
            trunkGradient.addColorStop(1, '#552200');
            context.fillStyle = trunkGradient;
            context.fillRect(-5, -50, 10, 50);

            var canopyShadow = context.createLinearGradient(0, -
50, 0, 0);
            canopyShadow.addColorStop(0, 'rgba(0, 0, 0, 0.5)');
            canopyShadow.addColorStop(0.2, 'rgba(0, 0, 0, 0.0)');
            context.fillStyle = canopyShadow;
            context.fillRect(-5, -50, 10, 50);

            createCanopyPath(context);

            context.lineWidth = 4;
            context.lineJoin = 'round';
            context.strokeStyle = '#663300';
            context.stroke();

            context.fillStyle = '#339900';
            context.fill();
        }


        function drawTrails() {
            var canvas = document.getElementById('trails');
            var context = canvas.getContext('2d');

            context.save();
            context.translate(130, 250);
            drawTree(context);
```

```
            context.restore();

            context.save();
            context.translate(260, 500);

            context.scale(2, 2);
            drawTree(context);
            context.restore();

            context.save();
            context.translate(-10, 350);
            context.beginPath();
            context.moveTo(0, 0);
            context.quadraticCurveTo(170, -50, 260, -190);
            context.quadraticCurveTo(310, -250, 410,-250);
            context.strokeStyle = context.createPattern(gravel,
'repeat');
            context.lineWidth = 20;
            context.stroke();
            context.restore();

            context.save();
            context.font = "60px impact";
            context.textAlign = 'center';
            context.fillStyle = '#996600';
            context.fillText('Happy Trails!', 200, 60, 400);
            context.restore();
        }

  </script>
</html>
```

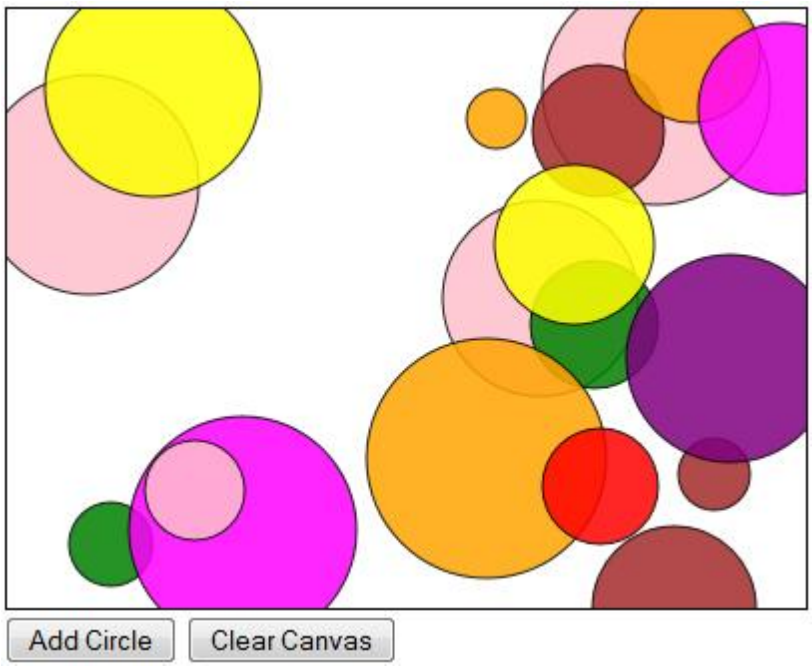


Как сделать фигуры интерактивными?

Холст является незапоминаемой поверхностью рисования. Это значит, что холст не отслеживает выполняемые на нем действия. Например, если нарисовать квадрат и закрасить его красным цветом, то он станет ничем иным, как блоком красных пикселов. Нам он может казаться квадратом, но холст не владеет информацией об этом.

Но для того чтобы отслеживать и обновлять содержимое холста, нам необходимо иметь всю информацию об этом содержимом.

Рассмотрим пример создания интерактивной программы для рисования кругов



HTML-код для программы. Листинг 9.30

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Interactive Circles</title>

<style>
body {
  font-family: Verdana;
  font-size: small;
}
canvas {
  cursor: pointer;
  border: 1px solid black;
}
</style>
  <script src="InteractiveCircles_WithDrag.js"></script
</head>
<body>
  <canvas id="canvas" width="400" height="300">
  </canvas>
  <div>
    <button onclick="addRandomCircle()">Add Circle</button>
    <button onclick="clearCanvas()">Clear Canvas</button>
  </div>
</body>
</html>
```

В файле InteractiveCircles_WhithDrag.js перва определим несколько пустых переменных, массив circles, в котором будет храниться информация о кругах, и функцию для прослушки событий.

Функция прослушки событий и определение. Листинг 9.31

```
var circles = [];

var canvas;
var context;

window.onload = function() {
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");

  canvas.onmousedown = canvasClick;
  canvas.onmouseup = stopDragging;
  canvas.onmouseout = stopDragging;
  canvas.onmousemove = dragCircle;
};
```

Мы хотим, чтобы объект мог хранить данные. Это делается посредством создания свойств с помощью ключевого слова this.

Функция Cicle. Листинг 9.32

```
function Circle(x, y, radius, color) {
  this.x = x;
  this.y = y;
  this.radius = radius;
  this.color = color;
  this.isSelected = false;
}
```

Теперь с помощью функции Circle можно создавать новые объекты круга. Но здесь есть еще одна особенность: мы не хотим каждый раз вызывать эту функцию, а вместо этого создавать новую копию объекта с помощью ключевого слова new:

var myCircle = new Circle(0, 0, 20, 'red');

Когда пользователь нажимает кнопку 'Add Circle', вызывается функция addRandomCircle(), которая создает новый круг произвольного размера и цвета в произвольном месте.

Функция addRandomCircle(). Листинг 9.33

```
function addRandomCircle() {
  // Устанавливаем произвольный размер и позицию круга.
  var radius = randomFromTo(10, 60);
  var x = randomFromTo(0, canvas.width);
  var y = randomFromTo(0, canvas.height);

  // Окрашиваем круг произвольным цветом.
  var colors = ["green", "blue", "red", "yellow", "magenta",
"orange", "brown", "purple", "pink"];
  var color = colors[randomFromTo(0, 8)];
```

```
  // Создаем новый круг
  var circle = new Circle(x, y, radius, color);

  // Сохраняем созданный круг в массиве.
  circles.push(circle);

  // Обновляем отображение круга.
  drawCircles();
}
```

В этом коде применяется функция randomFromTo(), которая генерирует произвольные числа в заданном диапазоне.

Функция randomFromTo(). Листинг 9.34

```
function randomFromTo(from, to) {
  return Math.floor(Math.random() * (to - from + 1) + from);
}
```

Последним шагом этой функции будет прорисовка текущей коллекции кругов на холсте. Для этого после создания нового круга функция addRandomCircle() вызывает функцию drowCircles(), которая в свою очередь с помощью цикла for перебирает массив кругов.

Функция drawCircles(). Листинг 9.35

```
function drawCircles() {
  // Очищаем холст
  context.clearRect(0, 0, canvas.width, canvas.height);

  // Перебираем все круги
  for(var i=0; i<circles.length; i++) {
    var circle = circles[i];

    // Рисуем круг
    context.globalAlpha = 0.85;
    context.beginPath();
    context.arc(circle.x, circle.y, circle.radius, 0, Math.PI*2);
    context.fillStyle = circle.color;
    context.strokeStyle = "black";

    if (circle.isSelected) {
      context.lineWidth = 5;
    }
    else {
      context.lineWidth = 1;
    }
    context.fill();
    context.stroke();
  }
}
```

При каждом обновлении холста, программа рисования полностью очищает холст посредством метода clearRect(). Это необходимо для того, чтобы программа повторно не рисовала те круги, которые уже отображены на холсте.

Очищаем холст с помощью простой функции clearCanvas()

Функция clearCanvas(). Листинг 9.36

```
function clearCanvas() {
  // Удаляем все элементы массива circles
  circles = [];

  // Обновляем содержимое холста
  drawCircles();
}
```

Нарисованные круги еще не обладают интерактивностью, но холст уже знает точное местоположение кругов, а это значит, что он может манипулировать этими кругами.

Рассмотрим систему, которая пользователю позволит работать с выбранными кругами. В нашей программе рисования кругов, нам нужно проверить, кликнул ли пользователь по фигуре круга или по пустому пространству холста. Т.е. нам нужно вычислить, не находится ли проверяемая точка внутри круга. Первое, что нам потребуется это цикл, для перебора всех фигур:

for(var i=circles.length-1; i>=0; i—) {...}

Данный цикл несколько отличается от используемого в функции drowCircles(). Он перебирает элементы массива в обратном порядке. Сделано это потому, что нарисованные позже объекты накладываются на более ранние, а при накладывании объектов щелчок должен получить тот объект, который находится сверху.
Далее нам необходимо вычислить расстояние по прямой линии от точки, в которой щелкнул пользователь, до центра круга. Если это расстояние меньше или равно радиусу круга, то эта точка находится в пределах круга.

Функция clearCanvas(). Листинг 9.37

```
var previousSelectedCircle;

function canvasClick(e) {
  // Получаем координаты точки холста в которой щелкнули
  var clickX = e.pageX - canvas.offsetLeft;
  var clickY = e.pageY - canvas.offsetTop;

  // Проверяем, щелкнули ли по кругу
  for(var i=circles.length-1; i>=0; i--) {
    var circle = circles[i];
```

```
    var distanceFromCenter = Math.sqrt(Math.pow(circle.x - clickX,
2) + Math.pow(circle.y - clickY, 2))
    if (distanceFromCenter <= circle.radius) {

   // сбрасываем предыдущий выбранный круг
      if (previousSelectedCircle != null)
previousSelectedCircle.isSelected = false;
      previousSelectedCircle = circle;

      circle.isSelected = true;

      // Устанавливаем новый выбранный круг
      isDragging = true;
      // Обновляем холст
      drawCircles();
      return;
    }
  }
}
```

Далее обнулим переменную isDragging, и создадим функцию stopDragging().

Функция stopDragging(). Листинг 9.38

```
var isDragging = false;

function stopDragging() {
  isDragging = false;
}
```

И, наконец, рассмотрим функцию, которая будет прорисовывать новый круг по ходу перетаскивания.

Функция dragCircle(e). Листинг 9.39

```
function dragCircle(e) {
  // Is a circle being dragged?
  if (isDragging == true) {
    // Make sure there really is a circle object (just in case).
    if (previousSelectedCircle != null) {
      // Find the new position of the mouse.
      var x = e.pageX - canvas.offsetLeft;
      var y = e.pageY - canvas.offsetTop;

      // Move the circle to that position.
      previousSelectedCircle.x = x;
      previousSelectedCircle.y = y;

      // Update the canvas.
      drawCircles();
    }
  }
}
```

Простая анимация

Сделать анимацию на холсте можно следующим образом: установить таймер, который будет обновлять содержимое всего холста раз 30 или 40 в секунду. Если код разработан правильно, меняющиеся кадры сольются в плавную анимацию.

В JavaScript есть два способа управления повторяющимся обновлением содержимого холста:

- Функция **setTimeout()**. Эта функция дает указание браузеру подождать несколько секунд, а потом исполнить фрагмент кода. И так множество раз, пока анимация не будет завершена.
- Функция **setInterval()**. Эта функция дает указание исполнять определенный фрагмент кода через регулярный интервал времени. Чтобы прекратить выполнение кода, вызывается функция clearInterval().

Библиотеки рисования

CanvasPlus (http://code/google/p/canvasplus)

Artisan JS (http://artisanjs.com)

Эти и другие библиотеки продолжают развиваться, поэтому рано говорить о том, какие из них окажутся наиболее пригодными для профессионального применения.

Программы рисования

- Также для рисования можно использовать профессиональные инструменты, например, модуль Ai->Canvas для Adobe Illustrator. Данный модуль преобразовывает графику, созданную в Adobe Illustrator в html-документ.
- www.jswidget.com/index-ipaint.html – продвинутый инструмент для рисования на холсте
- http://mugtug.com/sketchpad – программа для рисования и черчения.

Сохранение содержимого холста

Для сохранения содержимого холста предоставляется три возможных подхода:

- **Использовать URL данных.**
- **Использовать метод getImageData().**
- **Сохранять списки шагов**. Например, организовать массив, содержащий список всех линий, нарисованных на холсте. Эти данные потом можно сохранить и использовать для воспроизведения изображения.
- **Перевести в файл изображения**. Например, в JPG или PNG.

Рассмотрим сохранение содержимого холста, которое называтся URL данных.

var url = canvas.toDataURL()

Если вызывать метод toDataURL() не передавая ему никаких значений, то по умолчанию получим изображение в формате PNG. Можно указать формат JPG

var url = canvas.toDataURL('image/jpeg')

Технически, URL данных — это просто набор символов, закодированных алгоритмом Base64

10. API перетаскивания. События перетаскивания. Перетаскивание файлов.

Когда пользователь выполняет операцию перетаскивания, на источнике срабатывают следующие три события:

- **dragstart**. Срабатывает, когда операция перетаскивания начинается.
- **drag**. Похоже на mousemove, но срабатывает во время операции перетаскивания на элементе-источнике.
- **dragend**. Срабатывает, когда операция перетаскивания заканчивается (успешно или неудачно).

Следующие события срабатывают на целевом элементе на протяжении той же операции.

- **dragenter**. Срабатывает, когда во время операции перетаскивания указатель мыши оказывается в области предполагаемого целевого документа.
- **dragover**. Похоже на событие mousemove, но срабатывает во время операций перетаскивания на возможных целевых элементах.
- **drop**. Срабатывает, когда во время операций перетаскивания пользователь отпускает элемент-источник над целевым элементом.
- **dragleave**. Срабатывает, когда указатель мыши покидает область возможного целевого документа. Используется совместно с **dragenter** и обеспечивает взаимодействие с объектами приложения, помогая идентифицировать целевые элементы.

HTML-документ для тестирования событий перетаскивания. Листинг 10.1

```
<html lang="en">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="dropbox">
    Drag and drop the image here
```

```
  </section>
  <section id="picturesbox">
    <img id="image"
src="http://www.minkbooks.com/content/monster1.gif">
  </section>
</body>
</html>
```

Рассмотрим код для перетаскивания (файл dragdrop.js):

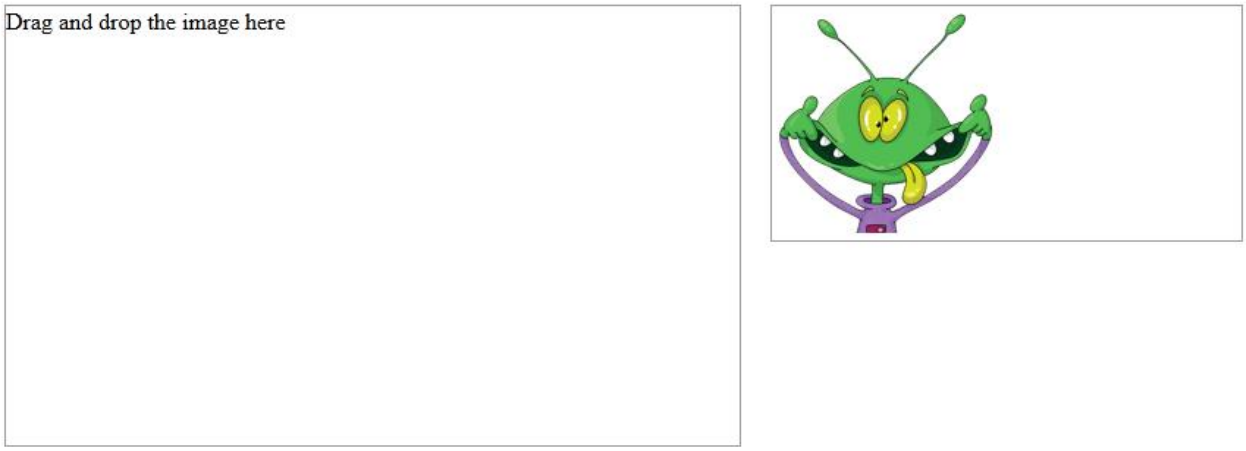
js-код для перетаскивания. Листинг 10.2

```
var source1, drop;
function initiate(){
  source1 = document.getElementById('image');
  source1.addEventListener('dragstart', dragged);

  drop = document.getElementById('dropbox');
  drop.addEventListener('dragenter', function(e){
e.preventDefault(); });
  drop.addEventListener('dragover', function(e){
e.preventDefault(); });
  drop.addEventListener('drop', dropped);
}
function dragged(e){
  var code = '<img src="' + source1.getAttribute('src') + '">';
  e.dataTransfer.setData('Text', code);
}
function dropped(e){
  e.preventDefault();
  drop.innerHTML = e.dataTransfer.getData('Text');
}
addEventListener('load', initiate);
```

Для того, чтобы функция могла принимать элемент, необходимо запретить поведение по умолчанию. Мы сделали это, добавив прослушиватели событий dragenter и dragover, а также анонимную функцию, которая выполняет метод **preventDefault()**. Для того, чтобы можно было сослаться на событие внутри функции, ей передается переменная e.
Когда пользователь начинает перетаскивать рисунок, срабатывает событие dragstart и вызывается функция dragged().
В этой функции мы извлекаем значение атрибута src перетаскиваемого элемента и настраиваем передаваемые данные с помощью метода setData() объекта dataTransfer. На другой стороне процесса, когда пользователь отпускает элемент над зоной приема, срабатывает событие drop и вызывается функция dropped(). Эта функция всего лишь модифицирует содержимое зоны приема, добавляя в неё информацию, полученную с помощью метода getData().

Объект **dataTransfer** содержит информацию, задействованную в операции перетаскивания. С объектом dataTransfer связаны следующие методы:

- **setData(type, data)**. Используется для объявления передаваемых данных и типа данных. Принимает данные обычных типов, таких как text/plain, text/html, text/uri-list и специальных типов URL и Text.
- **getData(type)**. Возвращает отправленные элементом-источником данные указанного типа.
- **clearData()**. Удаляет данные указанного типа.
- **setDragImage(element, x, y)**. Настраивает эскиз и выбор его точного местоположения.



До перетаскивания:
Завершение события перетаскивания (событие drop):

Управление всем процессом перетаскивания. Листинг 10.3

```
var source1, drop;
function initiate(){
  source1 = document.getElementById('image');
  source1.addEventListener('dragstart', dragged);
  source1.addEventListener('dragend', ending);

  drop = document.getElementById('dropbox');
  drop.addEventListener('dragenter', entering);
```

```
  drop.addEventListener('dragleave', leaving);
  drop.addEventListener('dragover', function(e){
e.preventDefault(); });
  drop.addEventListener('drop', dropped);
}
function entering(e){
  e.preventDefault();
  drop.style.background = 'rgba(0, 150, 0, .2)';
}
function leaving(e){
  e.preventDefault();
  drop.style.background = '#FFFFFF';
}
function ending(e){
  elem = e.target;
  elem.style.visibility = 'hidden';
}
function dragged(e){
  var code = '<img src="' + source1.getAttribute('src') + '">';
  e.dataTransfer.setData('Text', code);
}
function dropped(e){
  e.preventDefault();
  drop.style.background = '#FFFFFF';
  drop.innerHTML = e.dataTransfer.getData('Text');
}
addEventListener('load', initiate);
```

Выбор допустимого источника

Не существует специального метода, который проверял бы допускается ли перетаскивание выбранного элемента или нет. Но можно фильтровать источники проверяя атрибут id каждого изображения.

Создадим новый шаблон с дополнительными источниками.

Новый шаблон с дополнительными источниками. Листинг 10.4

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="dropbox">
    Drag and drop images here
  </section>
  <section id="picturesbox">
    <img id="image1"
src="http://www.minkbooks.com/content/monster1.gif">
    <img id="image2"
src="http://www.minkbooks.com/content/monster2.gif">
    <img id="image3"
```

```
src="http://www.minkbooks.com/content/monster3.gif">
    <img id="image4"
src="http://www.minkbooks.com/content/monster4.gif">
  </section>
</body>
</html>
```

Следующий javaScript код показывает, какое изображение можно опустить на зону приема, а какое нельзя.

Проверка атрибута id. Листинг 10.5

```
var drop;
function initiate(){
  var images = document.querySelectorAll('#picturesbox > img');
  for(var i = 0; i < images.length; i++){
    images[i].addEventListener('dragstart', dragged);
  }
  drop = document.getElementById('dropbox');
  drop.addEventListener('dragenter', function(e){
e.preventDefault(); });
  drop.addEventListener('dragover', function(e){
e.preventDefault(); });
  drop.addEventListener('drop', dropped);
}
function dragged(e){
  elem = e.target;
  e.dataTransfer.setData('Text', elem.getAttribute('id'));
}
function dropped(e){
  e.preventDefault();
  var id = e.dataTransfer.getData('Text');
  if(id != "image4"){
    var src = document.getElementById(id).src;
    drop.innerHTML = '<img src="' + src + '">';
  }else{
    drop.innerHTML = 'not admitted';
  }
}
addEventListener('load', initiate);
```

При перетаскивании изображений с идентификаторами id = image1, id = image2 и id = image3 изображение попадает в зону приема.

Но если мы попытаемся перетащить последнее изображение, с идентификатором id = image4, то увидим следующее сообщение:

Изменение эскиза

Метод **setDragImage()** не только позволяет менять эскиз, но также принимает два атрибута, x и y, устанавливающих позицию относительно указателя мыши.

Используя новый html-документ, продемонстрируем важность метода **setDragImage()**.

Использование элемента <canvas> в качестве зоны приема. Листинг 10.6

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="dropbox">
    <canvas id="canvas" width="500" height="300"></canvas>
  </section>
```

```
  <section id="picturesbox">
    <img id="image1"
src="http://www.minkbooks.com/content/monster1.gif">
    <img id="image2"
src="http://www.minkbooks.com/content/monster2.gif">
    <img id="image3"
src="http://www.minkbooks.com/content/monster3.gif">
    <img id="image4"
src="http://www.minkbooks.com/content/monster4.gif">
  </section>
</body>
</html>
```

Рассмотрим js-код приложения

Приложение с функциональностью перетаскивания. Листинг 10.7

```
var drop, canvas;
function initiate(){
  var images = document.querySelectorAll('#picturesbox > img');
  for(var i = 0; i < images.length; i++){
    images[i].addEventListener('dragstart', dragged);
    images[i].addEventListener('dragend', ending);
  }
  drop = document.getElementById('canvas');
  canvas = drop.getContext('2d');

  drop.addEventListener('dragenter', function(e){
e.preventDefault(); });
  drop.addEventListener('dragover', function(e){
e.preventDefault(); });
  drop.addEventListener('drop', dropped);
}
function ending(e){
  elem = e.target;
  elem.style.visibility = 'hidden';
}
function dragged(e){
  elem = e.target;
  e.dataTransfer.setData('Text', elem.getAttribute('id'));
  e.dataTransfer.setDragImage(elem, 0, 0);
}
function dropped(e){
  e.preventDefault();
  var id = e.dataTransfer.getData('Text');
  var elem = document.getElementById(id);
  var posx = e.pageX - drop.offsetLeft;
  var posy = e.pageY - drop.offsetTop;
  canvas.drawImage(elem, posx, posy);
}
addEventListener('load', initiate);
```

В данном примере, мы управляем эскизом перетаскиваемого элемента, положением относительно мыши и окончательным местоположением.

Перетаскивание файлов

API перетаскивания доступен не только изнутри html-документа, но так же позволяет пользователям перетаскивать элементы из браузера в другие приложения, и наоборот. Чаще всего возникает необходимость перетаскивать файлы из внешних источников (например, с рабочего стола) в браузер. HTML-код приложения перетаскивания файлов достаточно простой.

Шаблон для перетаскивания файлов. Листинг 10.8

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Drag and Drop</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="dropbox">
    Drag and drop FILES here
  </section>
</body>
</html>
```

У объекта dataTransfer есть еще специальное свойство **files**, которое возвращает массив, содержащий информацию о перетаскиваемых файлах. Информацию, возвращаемую свойством files можно сохранить в переменной, затем считать в цикле for. В следующем листинге мы выведем на экран название и размер каждого файла, попавшего в зону обработки.

JavaScript-код для перетаскивания файлов. Листинг 10.9

```
var drop;
function initiate(){
  drop = document.getElementById('dropbox');
  drop.addEventListener('dragenter', function(e){
e.preventDefault(); });
  drop.addEventListener('dragover', function(e){
e.preventDefault(); });
  drop.addEventListener('drop', dropped);
}
function dropped(e){
  e.preventDefault();
  var files = e.dataTransfer.files;
  var list = '';
  for(var f = 0; f < files.length; f++){
    list += 'File: ' + files[f].name + ' ' + files[f].size +
'<br>';
  }
  drop.innerHTML = list;
```

```
}
addEventListener('load', initiate);
```

11. API геолакации. Определение своего местоположения. Интеграция с Google Maps.

Данный API работает на базе таких систем, как **сетевая триангуляция** и **GPS**, и возвращает точное местоположение устройства, на котором выполняется данное приложение.
Для получения геолокационной информации в HTML5 существуют три метода

- **getCurrentPosition(location, error, configuration)** - применяется для одиночных запросов. Первый атрибут — это функция обратного вызова, предназначенная для получения информации, второй атрибут — функция для обработки ошибок, третий атрибут — объект, содержащий конфигурационные значения.
- **watchPosition (location, error, configuration)** - запускает процесс слежения за местоположением
- **clearWatch(id)**. Метод watchPosition возвращает значение, которое можно хранить в переменной, а затем, когда потребуется остановить слежение, необходимо передать данное значение методу clearWatch(). Принцип аналогичен использованию метода clearInteval() для остановки процесса, запущенного с помощью setInterval().

Шаблон для тестирования геолокации. Листинг 11.1

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Geolocation</title>
  <script src="geolocation.js"></script>
</head>
<body>
  <section id="location">
    <input type="button" id="getlocation" value="Get my location">
  </section>
</body>
</html>
```

Данный шаблон ничего, кроме кнопки с идентификатором id=getlocation не выводит.
Для получения информации о местоположении, воспользуемся методом getCurrentPosition(). Метод getCurentPosition() принадлежит объекту geolocation. Этот объект, в свою очередь, входит в объект navigator, таким образом, для вызова метода getCurrentPosition() необходимо воспользоваться следующим синтаксисом:

navigator.geolocation.getCurrentPosition(function)

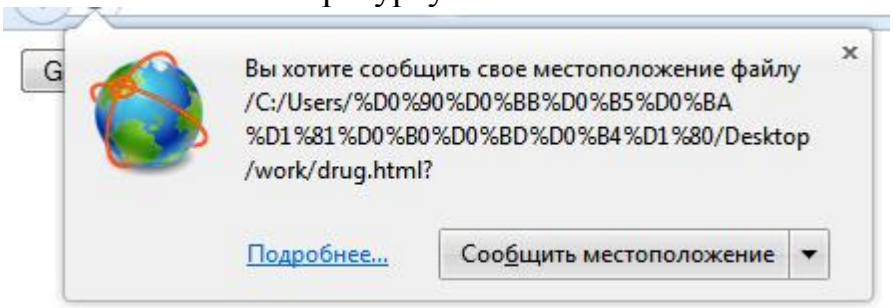
где function – это пользовательская функция, задача которой получить объект **Position** и обработать возвращенную методом информацию.
У объекта Position есть следующие атрибуты:

- **coords** – используется для получения **latitude** (широты), **longitude** (долготы), **alltitude** (высоты в метрах), **heading** (направление в градусах), **accuracy** (точности) и **altitudeAccuracy** (точности определения высоты в метрах).
- **timestamp** — возвращает время определения местоположения.

Получение информации о местоположении. Листинг 11.2

```
function initiate(){
  var get = document.getElementById('getlocation');
  get.addEventListener('click', getlocation);
}
function getlocation(){
  navigator.geolocation.getCurrentPosition(showinfo);
}
function showinfo(position){
  var location = document.getElementById('location');
  var data = '';
  data += 'Latitude: ' + position.coords.latitude + '<br>';
  data += 'Longitude: ' + position.coords.longitude + '<br>';
  data += 'Accuracy: ' + position.coords.accuracy + 'mts.<br>';
  location.innerHTML = data;
}
addEventListener('load', initiate);
```

При клике на кнопку, браузер запросит, согласны ли мы передать имнформацию о нашем местоположении ресурсу:



Рассмотрим еще один пример использования getCurrentPosition, но уже с двумя входящими параметрами. Вторым атрибутом мы можем перехватить возникающие ошибки. Одной из ошибок будет ошибка, связанная с невозможностью доступа (если пользовтель запретил браузеру обращаться к географическим данным).

Вывод сообщений об ошибках. Листинг 11.3

```
function initiate(){
  var get = document.getElementById('getlocation');
  get.addEventListener('click', getlocation);
}
```

```
function getlocation(){
  navigator.geolocation.getCurrentPosition(showinfo, showerror);
}
function showinfo(position){
  var location = document.getElementById('location');
  var data = '';
  data += 'Latitude: ' + position.coords.latitude + '<br>';
  data += 'Longitude: ' + position.coords.longitude + '<br>';
  data += 'Accuracy: ' + position.coords.accuracy + 'mts.<br>';
  location.innerHTML = data;
}
function showerror(error){
  alert('Error: ' + error.code + ' ' + error.message);
}
addEventListener('load', initiate);
```

А вот еще один пример использования метода getCurrentPosition, но уже с дополнительными конфигурационными настройками. Где

- **enableHighAccuracy**: Булев атрибут, извещающий систему о том, что требуется максимально точная информация о местоположении. Для того, чтобы вернуть точные координаты устройства, браузер попытается получить географическую информацию через GPS. Однако, эти системы расходуют большое количество ресурсов устройства, поэтому их использование необходимо ограничивать. Поэтому, по умолчанию значение данного атрибута равно FALSE.
- **timeout**: Задает максимальную продолжительность интервала времени, отведенного на выполнение операции. Если информация за это время не возвращается, то система возвращает ошибку TIMEOUT. Значение указывается в милисекундах.
- **maximumAge**: Координаты предыдущих местоположений кэшируются в системе. С помощью этого атрибута можно задать лимит возвраста информации. Если последнее кэширование старше указанного возраста, то выполняется запрос нового местоположения. Значение задается в милисекундах.

Рассмотрим код, который попытается получить самую точную информацию о местоположении устройства за время, не превышающее 10 с., при условии, что в кэше нет географических данных, полученных менее 60 с. назад (если есть, то именно они возвращаются в объект **Position**).

Конфигурация системы. Листинг 11.4

```
function initiate(){
  var get = document.getElementById('getlocation');
  get.addEventListener('click', getlocation);
}
function getlocation(){
  var geoconfig = {
    enableHighAccuracy: true,
```

```
    timeout: 10000,
    maximumAge: 60000
  };
  navigator.geolocation.getCurrentPosition(showinfo, showerror,
geoconfig);
}
function showinfo(position){
  var location = document.getElementById('location');
  var data = '';
  data += 'Latitude: ' + position.coords.latitude + '<br>';
  data += 'Longitude: ' + position.coords.longitude + '<br>';
  data += 'Accuracy: ' + position.coords.accuracy + 'mts.<br>';
  location.innerHTML = data;
}
function showerror(error){
  alert('Error: ' + error.code + ' ' + error.message);
}
addEventListener('load', initiate);
```

Для того чтобы сделать пример более понятным, мы сперва создали объект, сохранили его в переменной geoconfig, а затем использовали эту ссылку в методе getCurrentPosition().
Функция showinfo() выводит информацию на экран, не зависимо от того, каким образом она была получена: из кэша или путем нового системного запроса.
Если параметр enableHighAccurace равен true, браузер обращается к системе GPS, чтобы получить самые точные географические данные.

Слежение за изменением местоположения
Если метод getCurrentPosition() выполняется один раз, то метод **watchPosition()** автоматически возвращает новые данные при каждом изменении местоположения. Этот метод постоянно следит за координатами и при появлении новых данных отсылает информацию функции обратного вызова.
Синтаксис вызова метода watchPosition() аналогичен синтаксису getCurrentPosition():

navigator.geolocation.watchPosition(location, error, configuration)

В следующем примере кода, внедрим метод watchPosition() на основе предыдущих примеров кода.

Конфигурация системы. Листинг 11.5

```
function initiate(){
  var get = document.getElementById('getlocation');
  get.addEventListener('click', getlocation);
}
function getlocation(){
  var geoconfig = {
    enableHighAccuracy: true,
```

```
    maximumAge: 60000
  };
  control = navigator.geolocation.watchPosition(showinfo,
showerror, geoconfig);
}
function showinfo(position){
  var location = document.getElementById('location');
  var data = '';
  data += 'Latitude: ' + position.coords.latitude + '<br>';
  data += 'Longitude: ' + position.coords.longitude + '<br>';
  data += 'Accuracy: ' + position.coords.accuracy + 'mts.<br>';
  location.innerHTML = data;
}
function showerror(error){
  alert('Error: ' + error.code + ' ' + error.message);
}
addEventListener('load', initiate);
```

Выполнив этот код на настольном компьютере, мы не заметим ничего особенного. Но если запустить его на мобильном устройстве, то новая информация будет обновляться каждый раз, когда местоположение пользователя меняется. Атрибут maximumAge определяет, как часто географическая информация отсылается функции showinfo(). Если новые географические данные извлекаются через 60 с после предыдущей попытки, то они выводятся на экран. В противном случае функция showinfo() не вызывается.

Отменить процесс слежения можно методом **clearWatch()**.

Вывод карты на экран

Для вывода карты на экран можно воспользоваться API Google Maps. Это внешний API JavaScript, который никак не связан с HTML5.

Рассмотрим самый простой способ использования этого API — это API Static Maps (статические карты).

Для того чтобы воспользоваться данным API, необходимо всего лишь сформировать URL-адрес с информацией о местоположении.

Представление географических данных на изображении карты. Листинг 11.6

```
function initiate(){
  var get = document.getElementById('getlocation');
  get.addEventListener('click', getlocation);
}
function getlocation(){
  navigator.geolocation.getCurrentPosition(showinfo, showerror);
}
function showinfo(position){
  var location = document.getElementById('location');
  var mapurl = 'http://maps.google.com/maps/api/staticmap?center='
+ position.coords.latitude + ',' + position.coords.longitude +
'&zoom=12&size=400x400&sensor=false&markers=' +
position.coords.latitude + ',' + position.coords.longitude;
```


```
  location.innerHTML = '<img src="' + mapurl + '">';
}
function showerror(error){
  alert('Error: ' + error.code + ' ' + error.message);
}
addEventListener('load', initiate);
```

Мы вставили значение объекта Position() в URL-адрес в документе google.com.

Geocoding

Geocoding – это библиотека, которая позволяет делать всего 2 вещи:

- По наименованию чего-то, найти это на карте и сообщить координаты
- По координатам, сообщить всё что находится на этих координатах.

Например, мы хотим узнать где находится Узда. Пишем запрос:

http://maps.googleapis.com/maps/api/geocode/json?address=Узда&sensor=false&language=ru

Ответ придет в таком виде:

Ответ библиотеки Geolocation на запрос Узда. Листинг 11.7

```
{
   "results" : [
```

```
      {
         "address_components" : [
            {
               "long_name" : "Узда",
               "short_name" : "Узда",
               "types" : [ "locality", "political" ]
            },
            {
               "long_name" : "Минская область",
               "short_name" : "Минская область",
               "types" : [ "administrative_area_level_1",
"political" ]
            },
            {
               "long_name" : "Беларусь",
               "short_name" : "BY",
               "types" : [ "country", "political" ]
            }
         ],
         "formatted_address" : "Узда, Беларусь",
         "geometry" : {
            "bounds" : {
               "northeast" : {
                  "lat" : 53.47752410,
                  "lng" : 27.24639440
               },
               "southwest" : {
                  "lat" : 53.43587260,
                  "lng" : 27.18584060
               }
            },
            "location" : {
               "lat" : 53.46611110000001,
               "lng" : 27.22444440
            },
            "location_type" : "APPROXIMATE",
            "viewport" : {
               "northeast" : {
                  "lat" : 53.47752410,
                  "lng" : 27.24639440
               },
               "southwest" : {
                  "lat" : 53.43587260,
                  "lng" : 27.18584060
               }
            }
         },
         "types" : [ "locality", "political" ]
      }
   ],
   "status" : "OK"
}
```

Также имеется возможность по координатам узнать адрес:

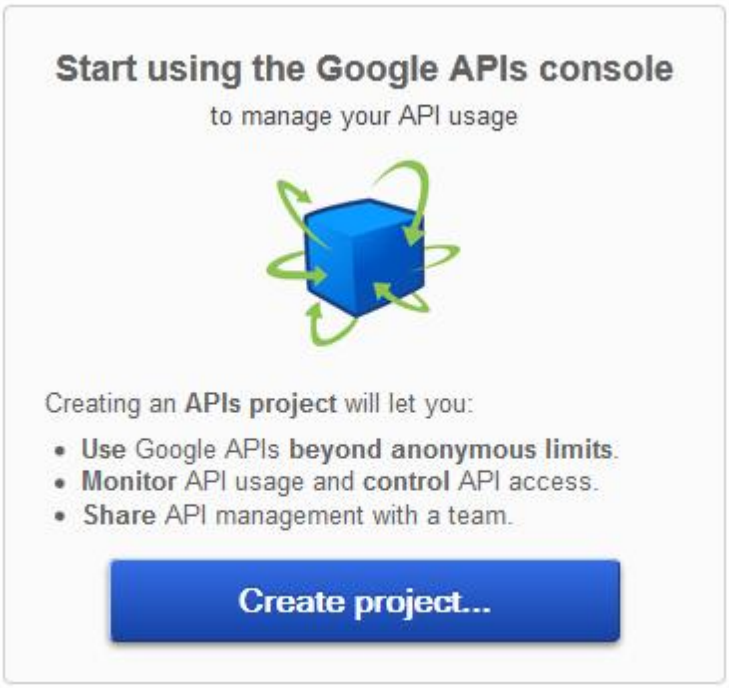
http://maps.googleapis.com/maps/api/geocode/json?latlng=55.75320193022759,37.619220867736 83&sensor=false&language=ru

JavaScript API Google Карт (версия 3)

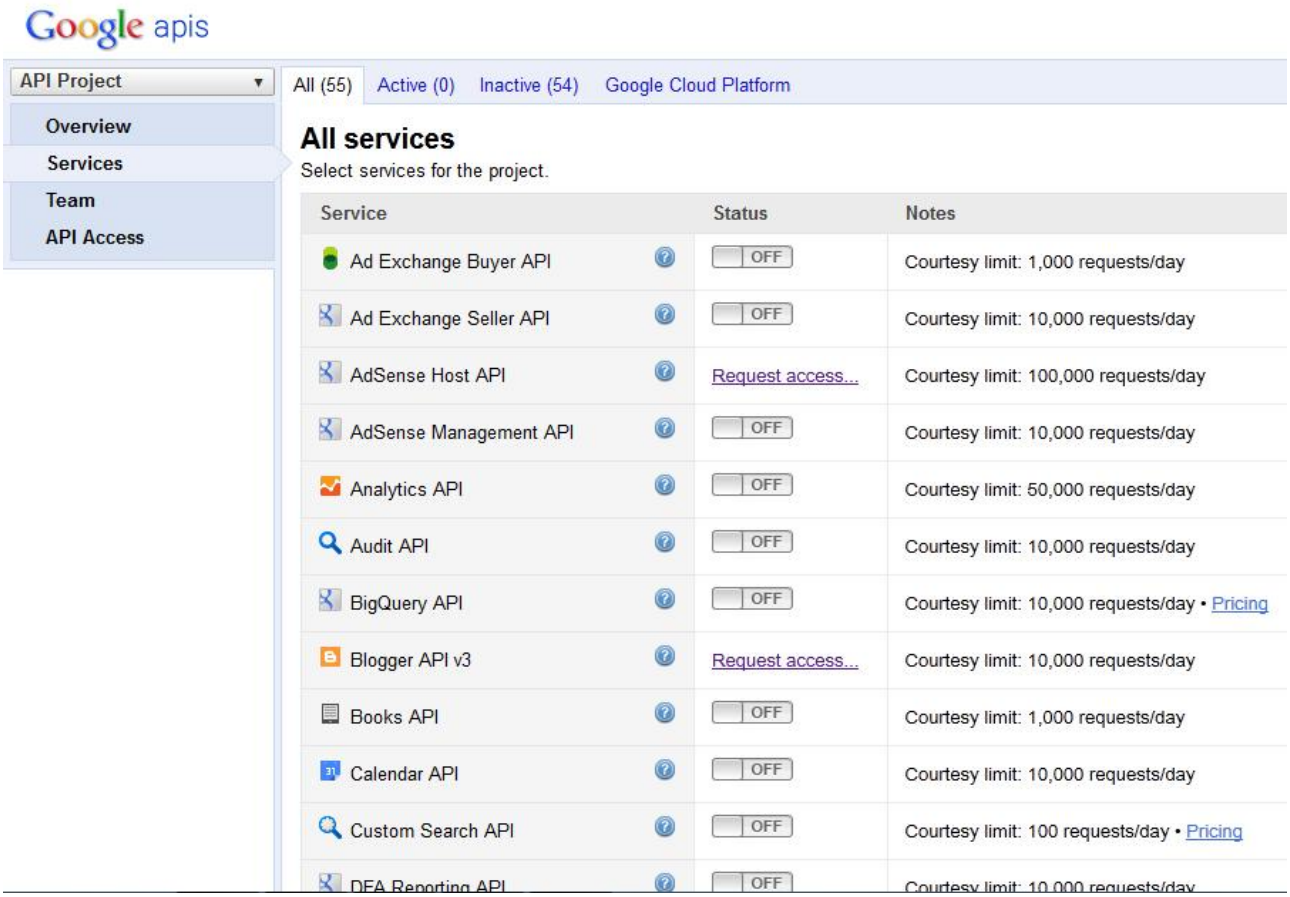
Все приложения, работающие с API Google Карт *, должны загружать этот интерфейс с помощью ключа API (кроме приложений работающих на localhost).

Для создания ключа API выполните следующие действия:

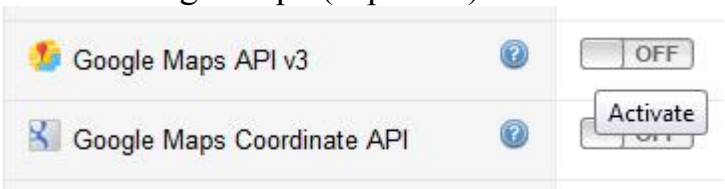
- Перейдите на страницу консоли интерфейсов API по адресу https://code.google.com/apis/console и войдите с использованием своего аккаунта Google.



- Нажмите на ссылку Services (Службы) в меню слева.



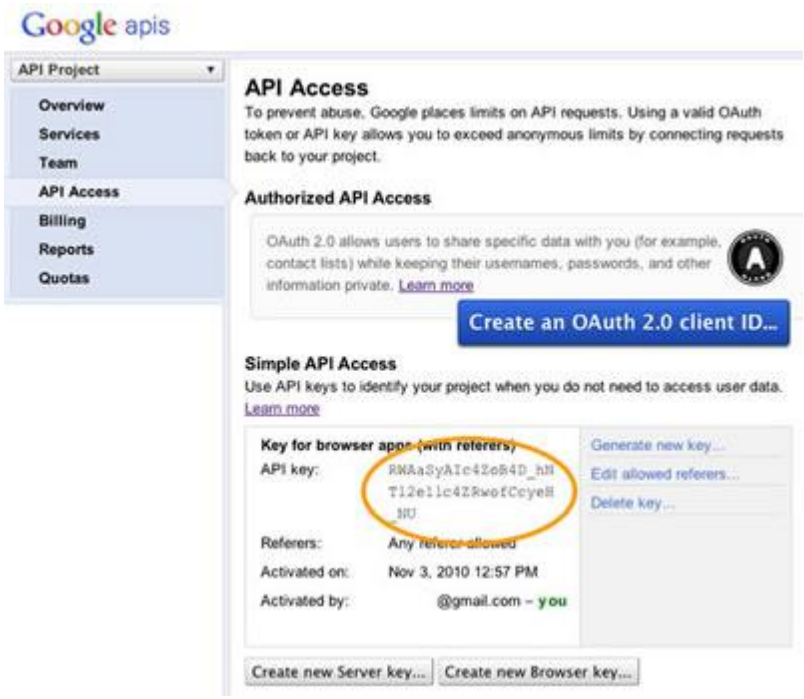
- Активируйте API Google Карт (версия 3).



После принятия пользовательского соглашения увидим следующее:

Что означает, активация прошла успешно.

- Нажмите на ссылку API Access (Доступ к API) в меню слева. Ключ API доступен на странице API Access (Доступ к API) в разделе Simple API Access (Обычный доступ к API). Приложения API Google Карт используют ключ для браузерных приложений.



По умолчанию этот ключ можно использовать на любом сайте. Во избежание доступа с неавторизованных сайтов мы настоятельно рекомендуем ограничить использование ключа пределами доменов, находящихся под вашим управлением. Чтобы указать домены, в которых разрешено использовать ваш ключ API, нажмите на ссылку Edit allowed referrers... (Управление доступом) для своего ключа.

Легче всего начать ознакомление с API Google Карт, рассмотрев простой пример. На следующей веб-странице отображается карта с центром в Сиднее (Новый Южный Уэльс, Австралия):

Представление географических данных на изображении карты. Листинг 11.8

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="initial-scale=1.0, user-
scalable=no" />
    <style type="text/css">
      html { height: 100% }
      body { height: 100%; margin: 0; padding: 0 }
    </style>
    <script type="text/javascript"

src="http://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&senso
r=SET_TO_TRUE_OR_FALSE">
    </script>
    <script type="text/javascript">
      function initialize() {
        var mapOptions = {
          center: new google.maps.LatLng(-34.397, 150.644),
          zoom: 8,
```

```
          mapTypeId: google.maps.MapTypeId.ROADMAP
        };
        var map = new
google.maps.Map(document.getElementById("map_canvas"),
            mapOptions);
      }
    </script>
  </head>
  <body onload="initialize()">
    <div id="map_canvas" style="width:100%; height:100%"></div>
  </body>
</html>
```

Объект карты инициализируется из события onload тега body.

Объявление CSS указывает, что контейнер карты (с именем map_canvas) должен занимать 100% высоты элемента HTML body. Обратите внимание, что требуется также особо объявить эти процентные значения для <body> и <html>.

Библиотеки API Google Map
https://developers.google.com/maps/documentation/javascript/tutorial
Код JavaScript для API Google Карт загружается с помощью URL *начальной загрузки*, имеющего формат http://maps.googleapis.com/maps/api/js. Этот запрос загружает все основные объекты и символы JavaScript, которые используются в API Карт. Некоторые компоненты API Карт доступны также в автономных *библиотеках*, для загрузки которых требуется отправить специальный запрос. Благодаря разнесению вспомогательных компонентов по библиотекам загрузка API (и синтаксический анализ) выполняется быстро. Загрузка и синтаксический анализ библиотек осуществляются только по мере необходимости.

Для загрузки дополнительных библиотек в рамках запроса начальной загрузки следует указать параметр libraries, задав ему название одной или нескольких библиотек. Названия библиотек в запросе разделяются запятой. Для доступа к загруженным библиотекам используется пространство имен google.maps.*libraryName*.

В настоящее время доступны следующие библиотеки:

- adsense дает возможность включать в приложение на основе API Карт связанные с контекстом объявления, позволяющие получать часть дохода от отображенной пользователям рекламы. Подробнее см. в документации по библиотеке AdSense.
- drawing предоставляет пользовательский графический интерфейс для прорисовки на карте многоугольников, прямоугольников, полилиний, окружностей и маркеров. Подробнее см. в документации по библиотеке для рисования.

- geometry содержит вспомогательные функции для вычисления скалярных геометрических значений (например, расстояния и площади) на поверхности земли. Подробнее см. в документации по библиотеке геометрических элементов.
- panoramio содержит функции для добавления слоев с фотографиями из Panoramio в приложение на основе API Карт. Подробнее см. в документации по слоям Panoramio.
- places позволяет приложению выполнять поиск адресов (например, организаций), географических точек или главных достопримечательностей в пределах заданной области. Подробнее см. в документации по библиотеке адресов.
- visualization обеспечивает наглядное представление данных тепловой карты и демографических данных для США. Подробнее о тепловых картах см. в документации по тепловым картам. Подробнее о слое демографических данных см. в документации по слою демографических данных в руководстве по Google Картам для организаций.
- weather позволяет добавлять на карту данные о погоде и облачности. Подробнее см. в документации по слою "Погода".

По мере выпуска дополнительно включаемых в интерфейс компонентов число библиотек будет увеличиваться.

Следующий запрос начальной загрузки показывает, как запросить библиотеку google.maps.geometry из Javascript API Google Карт:

Загрузка библиотеки geometry. Листинг 11.9

```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?libraries=geometry&sen
sor=true_or_false">
</script>
```

Следующий пример кода показывает, как загрузить сразу множество библиотек. Названия библиотек разделяются запятыми.

Загрузка библиотеки geometry. Листинг 11.10

```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?libraries=geometry,pla
ces&sensor=true_or_false">
</script>
```

Асинхронная загрузка API

Вам может понадобиться загрузить код JavaScript API Google Карт после окончания загрузки страницы или по запросу. Для этой цели можно ввести собственный тег <script>, отвечающий на событие window.onload или вызов функции. Однако при этом требуется предусмотреть дополнительные команды начальной загрузки JavaScript API Google Карт, обеспечивающие задерж-

ку выполнения кода приложения до момента полной загрузки кода этого интерфейса. Это осуществляется с помощью параметра callback, принимающего функцию, выполняемую по завершению загрузки API.

В следующем коде приложение получает команду загрузить API Google Карт после полной загрузки страницы (с помощью window.onload) и записать JavaScript API Google Карт в тег <script> на этой странице. Кроме того, API получает команду выполнить функцию initialize() только после того, как интерфейс будет полностью загружен. Для этого на этапе начальной загрузки передается параметр callback=initialize:

Загрузка библиотеки geometry. Листинг 11.11

```
function initialize() {
  var mapOptions = {
    zoom: 8,
    center: new google.maps.LatLng(-34.397, 150.644),
    mapTypeId: google.maps.MapTypeId.ROADMAP
  }
  var map = new
google.maps.Map(document.getElementById("map_canvas"),
mapOptions);
}

function loadScript() {
  var script = document.createElement("script");
  script.type = "text/javascript";
  script.src =
"http://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&sensor=TR
UE_OR_FALSE&callback=initialize";
  document.body.appendChild(script);
}

window.onload = loadScript;
```

Рассмотрим параметры карты

Параметры карты. Листинг 11.12

```
var mapOptions = {
  center: new google.maps.LatLng(-34.397, 150.644),
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

Для инициализации карты в первую очередь создается объект Map options для включения в него переменных инициализации карты. Этот объект не конструируется, а создается как литерал объекта.

var mapOptions = {};

Поскольку в нашем примере требуется с помощью свойства center поместить карту в конкретную точку, мы создаем объект LatLng для хранения этого местоположения и передаем координаты в порядке {широта, долгота}:

center = new google.maps.LatLng(-34.397, 150.644)

Начальное разрешение, используемое при отображении карты, устанавливается в свойстве `zoom`, причем значение `0` соответствует наименьшему уровню масштабирования. При увеличении этого значения увеличивается разрешение изображения карты.

zoom: 8

Чтобы показать карту Земли на одном изображении, потребуется использовать либо огромную карту, либо маленькую карту с очень низким разрешением. По этой причине изображения карты в Google Картах и API Google Карт подразделяются на фрагменты с разными уровнями масштабирования. При низких уровнях масштабирования небольшой набор фрагментов позволяет представить обширную область, а при высоких уровнях масштабирования фрагменты имеют более высокое разрешение и представляют меньшую область.

На следующих трех изображениях показан один и тот же район Токио при уровнях масштабирования 0,7 и 18.

Необходимо также задавать начальный тип карты:

mapTypeId: google.maps.MapTypeId.ROADMAP

Поддерживаются следующие типы карт:

- ROADMAP – стандартные двухмерные фрагменты Google Карт.
- SATELLITE – фрагменты, представленные сделанными со спутника фотографиями.
- HYBRID – фотографические фрагменты с наложенным слоем,

содержащим наиболее важные объекты (дороги, названия городов).
- TERRAIN – фрагменты топографической карты с рельефом местности, высотами и гидрографическими объектами (горы, реки и т. д.).

Карту представляет класс JavaScript `Map`. Объекты этого класса определяют одну карту на странице. (Можно создать несколько экземпляров этого класса. При этом каждый объект будет определять отдельную карту на странице.) Экземпляр этого класса создается с помощью оператора JavaScript `new`.
При создании нового экземпляра карты указывается элемент HTML `<div>` на странице, выступающий в качестве контейнера для карты. HTML-узлы являются потомками объекта JavaScript `document`. Ссылку на этот элемент можно получить с помощью метода `document.getElementById()`.
В данном коде определяется переменная (с именем `map`), которая назначается новому объекту `Map`, при этом передаются параметры, определенные в литерале объекта `mapOptions`. Эти параметры используются для инициализации свойств карты.

Службы маршрутов
С помощью объекта DirectionsService можно рассчитывать маршруты для различных способов передвижения. Этот объект взаимодействует со службой маршрутов интерфейса API Google Карт, которая получает запрос маршрута и возвращает вычисленные результаты.

var directionsService = new google.maps.DirectionsService();

Вы можете сами обработать эти результаты или использовать объект DirectionsRenderer для их визуализации.

В службе маршрутов пункты отправления и назначения могут указываться в виде текстовых запросов (например, "Чикаго, Иллинойс, США" или "Дарвин, Новый Южный Уэльс, Австралия") либо в виде координат LatLng. Результаты возвращаются в виде последовательности отрезков, проходящих через путевые точки. Маршруты отображаются в виде полилинии, показывающей маршрут на карте, или дополнительно в виде последовательности текстовых описаний в элементе (например, "Поверните направо для въезда на Троицкиё мост").

Чтобы использовать маршруты в версии 3, создайте объект типа DirectionsService и вызовите метод DirectionsService.route() для отправки запроса в службу маршрутов, передавая ей литерал объекта DirectionsRequest, содержащий условия ввода и метод обратного вызова для выполнения после получения ответа.

Литерал объекта DirectionsRequest содержит следующие поля:

Поля литерала объекта DirectionsRequest. Листинг 11.13

```
{
  origin: LatLng | String,
  destination: LatLng | String,
  travelMode: TravelMode,
  transitOptions: TransitOptions,
  unitSystem: UnitSystem,
  waypoints[]: DirectionsWaypoint,
  optimizeWaypoints: Boolean,
  provideRouteAlternatives: Boolean,
  avoidHighways: Boolean,
  avoidTolls: Boolean
  region: String
}
```

Описание этих полей:

- **origin** (*обязательный параметр*) указывает начальное местоположение, от которого следует вычислять маршрут. Этот параметр может иметь значение типа String (например, "Чикаго, Иллинойс, США") или LatLng.
- **destination** (*обязательный параметр*) указывает конечное местоположение, до которого следует вычислять маршрут. Этот параметр может иметь значение типа String (например, "Чикаго, Иллинойс, США") или LatLng.
- **travelMode** (*обязательный параметр*) задает способ перемещения, который используется для вычисления маршрута. Допустимые значения указаны далее в разделе Способы перемещения.
- **transitOptions** (*необязательный параметр*) указывает значения, используемые только в запросах, в которых параметр travelMode имеет значение google.maps.TravelMode.TRANSIT. Допустимые значения описаны далее в разделе Параметры маршрутов общественного транспорта.
- **unitSystem** (*необязательный параметр*) задает единицы измерения, которые следует использовать при отображении результатов. Допустимые значения указаны далее в разделе Системы измерений.
- **waypoints[]** (*необязательный параметр*) указывает массив элементов DirectionsWaypoint. Использование путевых точек приводит к изменению маршрута, направляя его через один или несколько указанных пунктов. Путевая точка задается как литерал объекта со следующими полями:
  - location задает местоположение путевой точки, которая будет геокодирована, как LatLng или String.
  - stopover содержит логическое значение, указывающее, что путевая точка является остановкой на маршруте, что приводит к его разделению на две части.

  Дополнительные сведения о путевых точках см. ниже в разделе

Использование путевых точек в маршрутах.

- **optimizeWaypoints** (*необязательный параметр*) указывает, что с помощью предоставленных элементов waypoints маршрут можно оптимизировать для получения кратчайшего пути. Если в этом поле указано значение true, служба маршрутов возвращает переупорядоченные элементы waypoints в поле waypoint_order.Дополнительные сведения см. далее в разделе Использование путевых точек в маршрутах.
- **provideRouteAlternatives** (*необязательный параметр*) – значение true указывает, что служба может предложить несколько альтернативных маршрутов. Обратите внимание, что расчет альтернативных путей может увеличить время отклика сервера.
- **avoidHighways** (*необязательный параметр*) – значение true указывает, что проложенные маршруты должны по возможности избегать автомагистралей.
- **avoidTolls** (*необязательный параметр*) – значение true указывает, что проложенные маршруты должны по возможности избегать платных дорог.
- **region** (*необязательный параметр*) указывает код региона в виде двухсимвольного значения ccTLD (домен верхнего уровня). Дополнительные сведения см. далее в разделе Предпочитаемый регион.

Ниже показан пример элемента DirectionsRequest:

Пример элемента DirectionsRequest. Листинг 11.14

```
{
  origin: "Chicago, IL",
  destination: "Los Angeles, CA",
  waypoints: [
    {
      location:"Joplin, MO",
      stopover:false
    },{
      location:"Oklahoma City, OK",
      stopover:true
    }],
  provideRouteAlternatives: false,
  travelMode: TravelMode.DRIVING,
  unitSystem: UnitSystem.IMPERIAL
}
```

Способы перемещения

При вычислении маршрутов требуется указать используемый способ перемещения. В настоящее время поддерживаются следующие способы перемещения:

- **google.maps.TravelMode.DRIVING** (используется по умолчанию) обозначает стандартные автомобильные маршруты по улично-

дорожной сети.

- **google.maps.TravelMode.BICYCLING** запрашивает велосипедные маршруты по велосипедным дорожкам и предпочитаемым улицам.
- **google.maps.TravelMode.TRANSIT** запрашивает маршруты общественного транспорта.
- **google.maps.TravelMode.WALKING** запрашивает пешеходные маршруты по прогулочным дорожкам и тротуарам (если они есть).

Литерал объекта TransitOptions содержит следующие поля:

Значения литерала объекта TransitOptions. Листинг 11.15

```
{
  departureTime: Date,
  arrivalTime: Date
}
```

Описание этих полей:

- **departureTime** (необязательный параметр) задает требуемое время отправления в виде объекта Date. Если указан параметр arrivalTime, параметр departureTime игнорируется. Если значения параметров departureTime или arrivalTime не указаны, по умолчанию задается текущее время.
- **arrivalTime** (необязательный параметр) указывает требуемое время прибытия в виде объекта Date. Если указано время прибытия, время отправления игнорируется.

Рассмотрим пример объекта DirectionsRequest для маршрута общественного транспорта:

Пример объекта DirectionsRequest для маршрута общественного транспорта. Листинг 11.16

```
{
  origin: "Hoboken NJ",
  destination: "Carroll Gardens, Brooklyn",
  travelMode: google.maps.TravelMode.TRANSIT,
  transitOptions: {
    departureTime: new Date(1337675679473)
  },
  unitSystem: google.maps.UnitSystem.IMPERIAL
}
```

Визуализация маршрутов

Запрос маршрута в службе DirectionsService методом route() требует передачи обратного вызова, выполняемого после того, как служба завершит обработку запроса. Этот обратный вызов возвращает в ответе DirectionsResult и код DirectionsStatus.

Статус запроса маршрута. DirectionsStatus может возвратить следующие значения:

- OK указывает, что ответ содержит допустимый элемент DirectionsResult.
- NOT_FOUND означает, что по крайней мере для одного указанного пункта (исходный, пункт назначения или путевая точка) не удалось выполнить геокодирование.
- ZERO_RESULTS означает, что между исходной точкой и пунктом назначения не найдено ни одного маршрута.
- MAX_WAYPOINTS_EXCEEDED указывает, что в DirectionsRequest задано слишком много объектов DirectionsWaypoint. Максимальное разрешенное количество путевых точек составляет 8 помимо исходной точки и пункта назначения. Клиентам API Google Карт для организаций разрешено использовать 23 путевые точки помимо исходной точки и пункта назначения. Для маршрутов общественного транспорта путевые точки не поддерживаются.
- INVALID_REQUEST указывает, что был предоставлен недопустимый объект DirectionsRequest. В большинстве случаев этот код ошибки возвращается для запросов без исходной точки и пункта назначения, а также для запросов маршрутов общественного транспорта с путевыми точками.
- OVER_QUERY_LIMIT означает, что в разрешенный период времени веб-страница отправила слишком много запросов.
- REQUEST_DENIED означает, что веб-странице не разрешено использовать службу маршрутов.
- UNKNOWN_ERROR означает, что обработка запроса маршрута невозможна из-за ошибки сервера. При повторной попытке запрос может быть успешно выполнен.

Прежде чем обрабатывать результат, необходимо убедиться, что запрос маршрута возвратил допустимые данные.

Объект DirectionsResult содержит результат запроса маршрута, который можно обработать самостоятельно или передать в объект DirectionsRenderer, автоматически управляющий отображением результатов на карте.
Чтобы DirectionsResult отобразить с помощью объекта DirectionsRenderer, необходимо выполнить следующие действия:

1. Создайте объект DirectionsRenderer.
2. Вызовите метод setMap() средства визуализации, чтобы привязать объект к переданной карте.
3. Вызовите метод setDirections() средства визуализации, передавая ему объект DirectionsResult, как указано выше. Поскольку средство визуализации представляет собой объект MVCObject, оно автоматически обнаруживает любые изменения своих свойств и

обновляет карту при изменении связанных маршрутов.

В следующем примере вычисляется маршрут между двумя городами, для которых исходная точка и пункт назначения заданы значениями "start" и "end" из раскрывающихся списков. Объект DirectionsRenderer осуществляет отображение полилинии между указанными местоположениями, а также, если применимо, размещает маркеры в исходной точке, пункте назначения и всех путевых точках.

HTML-код. Листинг 11.17

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="initial-scale=1.0, user-
scalable=no">
    <meta charset="utf-8">
    <title>Google Maps JavaScript API v3 Example: Directions
Simple</title>
    <style>
    html, body {
      height: 100%;
      margin: 0;
      padding: 0;
    }

    #map-canvas, #map_canvas {
      height: 100%;
    }

    @media print {
      html, body {
        height: auto;
      }

      #map_canvas {
        height: 650px;
      }
    }
    </style>
    <script
src="http://maps.googleapis.com/maps/api/js?key=&sensor=false">
</script>
   <script src="my_road.js"> </script>
  </head>
  <body onload="initialize()">
    <div>
    <b>Start: </b>
    <select id="start" onchange="calcRoute();">
      <option value="Брест, Беларусь">Брест</option>
      <option value="Витебск, Беларусь">Витебск</option>
      <option value="Гомель, Беларусь">Гомель</option>
      <option value="Гродно, Беларусь">Гродно</option>
```

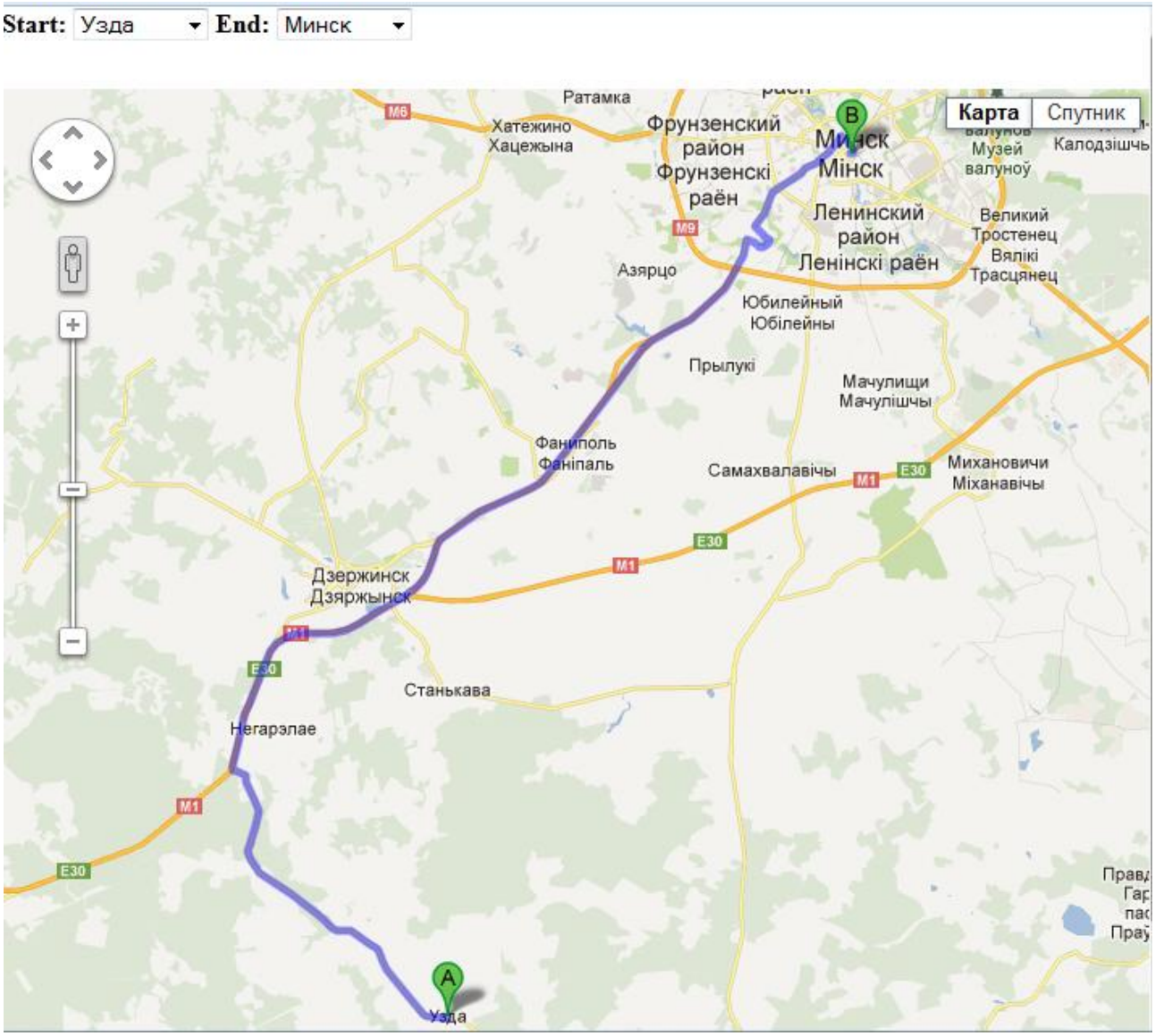
```
      <option value="Могилев, Беларусь">Могилев</option>
      <option value="Минск, Беларусь">Минск</option>
      <option value="Узда, Беларусь">Узда</option>
    </select>
    <b>End: </b>
    <select id="end" onchange="calcRoute();">
      <option value="Минск, Беларусь">Брест</option>
      <option value="Витебск, Беларусь">Витебск</option>
      <option value="Гомель, Беларусь">Гомель</option>
      <option value="Гродно, Беларусь">Гродно</option>
      <option value="Могилев, Беларусь">Могилев</option>
      <option value="Минск, Беларусь">Минск</option>
      <option value="Узда, Беларусь">Узда</option>
    </select>
    </div>
    <div id="map-canvas" style="top:30px;"></div>
  </body>
</html>
```

А вот и сам файл my_road.js

Скрипт для определения маршрута между двумя городами. Листинг 11.18

```
      var directionsDisplay;
      var directionsService = new google.maps.DirectionsService();
      var map;

      function initialize() {
        directionsDisplay = new google.maps.DirectionsRenderer();
        var chicago = new google.maps.LatLng(53.882949,
27.580941);
        var mapOptions = {
          zoom:7,
          mapTypeId: google.maps.MapTypeId.ROADMAP,
          center: chicago
        }
        map = new google.maps.Map(document.getElementById('map-
canvas'), mapOptions);
        directionsDisplay.setMap(map);
      }

      function calcRoute() {
        var start = document.getElementById('start').value;
        var end = document.getElementById('end').value;
        var request = {
            origin:start,
            destination:end,
            travelMode: google.maps.DirectionsTravelMode.DRIVING
        };
        directionsService.route(request, function(response,
status) {
          if (status == google.maps.DirectionsStatus.OK) {
            directionsDisplay.setDirections(response);
          }
```

```
        });
    }
```



По этой ссылке можно ознакомиться с полным перечнем возможностей, предоставляемым данным API:

https://developers.google.com/maps/documentation/javascript/directions?hl=ru

С примером можно ознакомиться по адресу:

http://obmenka.by/map.php

12. API web-хранилища.

API Web Storage (web хранилища) — это, по сути, следующая ступень развития файлов cookie. Этот API позволяет записывать данные на жесткий диск пользователя и обращаться к ним, как это делается в настольных

приложениях. Процессы хранения и извлечения данных применимы в двух ситуациях: когда данные доступны в течении одного сеанса и когда данные хранятся долго, до тех пор, пока пользователь сам их не удалит. Таким образом API разделен на две части: sessionStorage и localStorage:

- **sessionStorage**. Это маханизм хранения, удерживающий данные на протяжении сеанса одной страницы. В отличии от настоящих сеансов, доступ к информации есть только у одного окна или вкладки браузера. Как только окно или вкладка закрывается, эта информация удаляется.
- **localStorage**. Этот механизм работает аналогично системам хранения настольных приложений. Данные записываются навсегда. Приложение, сохранившее их может обращаться к ним в любой момент.

Оба механизма работают через один и тот же интерфейс и предлагают одинаковые методы и свойства. Поэтому, для тестирования работы обоих механизмов можно использовать один html-шаблон

html-шаблон для API хранения. Листинг 12.1

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Web Storage API</title>
  <style>
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#keyword, #text{
  width: 200px;
}
#databox > div{
  padding: 5px;
  border-bottom: 1px solid #999999;
}
  </style>
  <script src="storage.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <label for="keyword">Keyword: </label><br>
      <input type="text" name="keyword" id="keyword"><br>
```

```
      <label for="text">Value: </label><br>
      <textarea name="text" id="text"></textarea><br>
      <input type="button" id="save" value="Save">
    </form>
  </section>
  <section id="databox">
    No Information available
  </section>
</body>
</html>
```

Создание и извлечение данных

И sessionStorage и localStorage сохраняют данные в форме отдельных элементов. Элементом считается пара из ключевого слова и значения. Каждое значение перед помещением в строку необходимо конвертировать в строку.

Для создания и извлечения элементов из пространства хранилища предназначены два новых метода:

- **setItem(key, value)**. Для создания, где key – это ключевое слово, value – это значение.
- **getItem(key)**. Для извлечения по ключевому слову.

Сохранение и извлечение данных. Листинг 12.2

```
function initiate(){
  var button = document.getElementById('save');
  button.addEventListener('click', newitem);
}
function newitem(){
  var keyword = document.getElementById('keyword').value;
  var value = document.getElementById('text').value;
  sessionStorage.setItem(keyword, value);

  show(keyword);
}
function show(keyword){
  var databox = document.getElementById('databox');
  var value = sessionStorage.getItem(keyword);
  databox.innerHTML = '<div>' + keyword + ' - ' + value +
'</div>';
}
addEventListener('load', initiate);
```

Функция newitem() выполняется каждый раз, когда пользователь щелкает на кнопке формы. Эта функция создает элемент и добавляет в него информацию, полученную из формы, а затем вызывает функцию show(). Функция show() в свою очередь извлекает элемент из хранилища по ключевому слову, используя метод getItem(), а затем выводит его на экран.

Помимо этих методов, API хранения предоставляет упрощенный способ создания и извлечения элементов из пространства хранилища, в котором ключевое слово элемента используется, как свойство. Можно переменную ключевого слова заключать в квадратные скобки.

sessionStorage[ключевое_слово] = значение

, а можно передать строку в качестве имени свойства, например

sessionStorage.myitem = значение

Работа с элементами упрощенным способом. Листинг 12.3

```
function initiate(){
  var button = document.getElementById('save');
  button.addEventListener('click', newitem);
}
function newitem(){
  var keyword = document.getElementById('keyword').value;
  var value = document.getElementById('text').value;
  sessionStorage[keyword] = value;

  show(keyword);
}
function show(keyword){
  var databox = document.getElementById('databox');
  var value = sessionStorage[keyword];
  databox.innerHTML = '<div>' + keyword + ' - ' + value +
'</div>';
}
addEventListener('load', initiate);
```

Рассмотрим методы и свойства API, позволяющие манипулировать данными:

- **length**. Возвращает число элементов, помещенных в хранилище данным приложением.
- **key(index)**. Элементы записываются в хранилище последовательно, и им автоматически присваиваются порядковые номера, начиная с 0. С помощью данного метода можно извлечь определенный элемент или даже всю информацию, содержащуюся в хранилище, если пройтись по нему в цикле.

Перечисление элементов. Листинг 12.4

```
function initiate(){
  var button = document.getElementById('save');
  button.addEventListener('click', newitem);
  show();
}
function newitem(){
  var keyword = document.getElementById('keyword').value;
  var value = document.getElementById('text').value;
```

```
  sessionStorage.setItem(keyword, value);
  document.getElementById('keyword').value = '';
  document.getElementById('text').value = '';
  show();
}
function show(){
  var databox = document.getElementById('databox');
  databox.innerHTML = '';
  for(var f = 0; f < sessionStorage.length; f++){
    var keyword = sessionStorage.key(f);
    var value = sessionStorage.getItem(keyword);
    databox.innerHTML += '<div>' + keyword + ' - ' + value +
'</div>';
  }
}
addEventListener('load', initiate);
```

Задача данного листинга вывести полный список элементов из хранилища. Мы немного усовершенствовали функцию show(), применив свойство length и метод key(). Для этого создали цикл for, начинающийся с 0, и заканчивающийся порядковым номером последнего элемента из хранилища.
Функция show() вызывается из функции init(). Таким образом, она выводит список элементов из хранилища на экран сразу же, как только приложение запускается.

Удаление данных
Для удаления данных предназначены два метода:

- **removeItem(key)**. Удаляет один элемент по ключевому словуа
- **clear()**. Очищает пространство хранилища. Удаляются все находящиеся в нем элементы.

Удаление данных. Листинг 12.5

```
function initiate(){
  var button = document.getElementById('save');
  button.addEventListener('click', newitem);
  show();
}
function newitem(){
  var keyword = document.getElementById('keyword').value;
  var value = document.getElementById('text').value;

  sessionStorage.setItem(keyword, value);
  document.getElementById('keyword').value = '';
  document.getElementById('text').value = '';
  show();
}
function show(){
  var databox = document.getElementById('databox');
  databox.innerHTML = '<div><input type="button"
```

```
onclick="removeAll()" value="Erase Everything"></div>';
  for(var f = 0; f < sessionStorage.length; f++){
    var keyword = sessionStorage.key(f);
    var value = sessionStorage.getItem(keyword);
    databox.innerHTML += '<div>' + keyword + ' - ' + value +
'<br><input type="button" onclick="removeItem(\'' + keyword +
'\')" value="Remove"></div>';
  }
}
function removeItem(keyword){
  if(confirm('Are you sure?')){
    sessionStorage.removeItem(keyword);
    show();
  }
}
function removeAll(){
  if(confirm('Are you sure?')){
    sessionStorage.clear();
    show();
  }
}
addEventListener('load', initiate);
```

За удаление выбранного элемента и полную очистку хранилища отвечают

функции remove() и removeAll().

Сохранение чисел и дат

Т.к. сохраняемые данные автоматически преобразуются в текст, то перед выводом, если мы хотим получить число, данные нужно преобразовать с помощью функции Number():

Использование функции Number(). Листинг 12.6

```
var value = Nubmer(sessionStorage[keyword]);
```

При преобразовании типов, следует проявлять осторожность. Для некоторых типов данных существуют удобные процедуры преобразования, но например, если мы сохранили следующую дату:

var today = new Date();

Этот код сохранит не объект даты, а текстовую стоку. Например, Sat Jun 2013 13:30:46. К сожалению, не существует легкого способа преобразования этого текста обратно в дату.
Чтобы решить эту проблему, мы должны явно преобразовать дату в текст, а потом выполнить обратное преобразование.

Сохранение объекта даты. Листинг 12.7

```
var today = new Date();
sessionStorage['session_started'] = today.getFullYear() + "/"
   + today.getMonth() + "/" + today.getDate();
…
today = new Date(sessionStorage['session_started']);
alert(today.getFullYear);
```

В результате выполнения этого кода, появится окно сообщения, подтверждающее успешное восстановление объекта даты.

Сохранение объектов
Для того, чтобы сохранить объект, необходимо его преобразовать в текст. Существует стандартный способ, позволяющий это делать, называемый кодированием JSON.
Предположим, есть такая функция:

Функция. Листинг 12.8

```
function PersonalityScore(o, c, e, a, n){
 this.openness = o;
 this.cons = c;
 this.extraversion = e;
 this.agreeable = a;
 this.neuroticism = n;
}
```

Создаем объект PersonalityScore

Создание объекта. Листинг 12.9

```
 var score = new PersonalityScore(o, c, e, a, n);
```

Для преобразования объекта в формат JSON, вызовем метод JSON.stringify().

Преобразование объекта в текст формата JSON. Листинг 12.10

```
 sessionStorage['personalityScore'] = JSON.stringify(score);
```

Для обратного преобразования, воспользуемся методом JSON.parse:

Преобразование JSON-текста в соответсвующий объект. Листинг 12.11

```
 var scope = JSON.parse(sessionStorage);
```

Если мы собираемся сохранять большие объемы данных и на длительное время, то необходимо использовать объект localStorage. При этом, решение о

том, требуется ли информация, хранящаяся в объекте или нет, принимает сам пользователь.
Система localStorage использует такой же интерфейс, что и sessionStorage. Поэтому, для localStorage можно использовать те же методы и свойства, которые мы использовали ранее. Придется внести единственное изменение: заменить префикс session префиксом local.

Использование localStorage. Листинг 12.12

```
localStorage.setItem(keyword, value);
```

Слежение за областью HTML5-хранилища

Если вы хотите программно отслеживать изменения хранилища, то должны отлавливать событие storage. Это событие возникает в объекте window, когда setItem(), removeItem() или clear() вызываются и что-то изменяют. Например, если вы установили существующее значение или вызвали clear() когда нет ключей, то событие не сработает, потому что область хранения на самом деле не изменилась.

Событие storage поддерживается везде, где работает объект localStorage, включая Internet Explorer 8. IE 8 не поддерживает стандарт W3C addEventListener (хотя он, наконец-то, добавлен в IE 9), поэтому, чтобы отловить событие storage нужно проверить, какой механизм событий поддерживает браузер (если вы уже проделывали это раньше с другими событиями, то можете пропустить этот раздел до конца). Перехват события storage работает так же, как и перехват других событий. Если вы предпочитаете использовать jQuery или какую-либо другую библиотеку JavaScript для регистрации обработчиков событий, то можете проделать это и со storage тоже.

Событие storage. Листинг 12.13

```
if (window.addEventListener) {
  window.addEventListener("storage", handle_storage, false);
} else {
  window.attachEvent("onstorage", handle_storage);
};
```

Функция обратного вызова handle_storage будет вызвана с объектом StorageEvent, за исключением Internet Explorer, где события хранятся в window.event.

Функция обратного вызова handle_storage(e). Листинг 12.14

```
function handle_storage(e) {
  if (!e) { e = window.event; }
}
```

В данном случае переменная e будет объектом StorageEvent, который обладает следующими полезными свойствами.

Свойство	Тип	Описание
key	string	Ключ может быть добавлен, удален или изменен.
oldValue	любой	Предыдущее значение (если переписано) или null, если добавлено новое значение.
newValue	любой	Новое значение или null, если удалено.
url*	string	Страница, которая вызывает метод, приведший к изменению.

* Примечание: свойство url изначально называлось uri и некоторые браузеры поддерживали это свойство перед изменением спецификации. Для обеспечения максимальной совместимости вы должны проверить существует ли свойство url, и если нет проверить вместо него свойство uri.

Событие storage нельзя отменить, внутри функции обратного вызова handle_storage нет возможности остановить изменение. Это просто способ браузеру сказать вам: «Эй, это только что случилось. Вы ничего не можете сделать, я просто хотел, чтобы вы знали».

13. API индексированных баз данных.

Индексированная база данных – это хранилище объектов. Это не то же самое, что реляционная база, в которой есть таблицы с данными, размещенными в строках и столбцах. Это различие является фундаментальным и влияет на процесс разработки и создания приложений. В обычном реляционном хранилище данных мы имеем дело с таблицей элементов, в строках которых находятся данные пользовательских задач. В столбцах содержится информация различных типов. Для добавления данных обычно используется следующая семантика:

Пример добавления записей в базу данных MySQL. Листинг 13.1

```
INSERT INTO Todo(id, data, update_time) VALUES (1, "Test",
"01/01/2010");
```

Индексированная база данных отличается тем, что сначала создается хранилище объектов для какого-либо типа данных, а затем в нем сохраняются объекты JavaScript. В каждом хранилище объектов может быть набор индексов, ускоряющий обработку запросов и циклический перебор. Индексированные базы данных также позволяют отойти от стандартного языка запросов (SQL). Его место занимает запрос по индексу, который формирует курсор для перебора результатов.
В этом руководстве представлены только реальные примеры использования индексированных баз данных в контексте существующих приложений, написанных для WebSQL.

18 ноября 2010 г. консорциум W3C объявило прекращении поддержки СУБД Web SQL. В связи с этим разработчикам более не рекомендуется

использовать эту технологию, так как выпуск обновлений для нее прекращен, а поставщики браузеров не заинтересованы в ее дальнейшем развитии. Заменой для Web SQL являются индексированные базы данных (которым, собственно, и посвящено данное руководство), предлагаемые разработчикам для хранения и обработки данных в клиентских приложениях. Большинство популярных браузеров, в том числе Chrome, Safari, Opera, и практически все мобильные устройства на основе Webkit, поддерживают WebSQL и, вероятнее всего, продолжат поддержку этой технологии в обозримом будущем.

Асинхронный и транзакционный API

В большинстве случаев в индексированных базах данных используется асинхронный API. Он представляет собой систему без блокировки, в которой данные поступают не через возвращаемые значения, а главным образом в определенную функцию обратного вызова.

В HTML поддержка индексированных баз данных носит транзакционный характер. Выполнять различные команды и открывать курсоры можно только во время транзакции. Существует несколько типов транзакций: чтение-запись, только чтение и создание снимка данных. В нашем руководстве мы используем операции типа "чтение-запись".

Этап 1. Открытие базы данных

Сначала необходимо открыть базу данных.

Пример добавления записей в базу данных MySQL. Листинг 13.2

```
html5rocks.indexedDB.db = null;

html5rocks.indexedDB.open = function() {
  var request = indexedDB.open("todos");

  request.onsuccess = function(e) {
    html5rocks.indexedDB.db = e.target.result;
    // ...
  };

  request.onfailure = html5rocks.indexedDB.onerror;
};
```

Мы открыли базу под названием todos (задачи) и присвоили ее значение переменной db в объекте html5rocks.indexedDB. Теперь эту переменную можно использовать для обращения к базе в данном руководстве.

Этап 2. Создание хранилища объектов

Создавать хранилища объектов можно только во время транзакции SetVersion. Мы еще не рассматривали метод setVersion, но он очень важен, поскольку является единственным средством создания хранилищ объектов и индексов в коде.

Пример добавления записей в базу данных MySQL. Листинг 13.3

```
html5rocks.indexedDB.open = function() {
  var request = indexedDB.open("todos",
    "This is a description of the database.");

  request.onsuccess = function(e) {
    var v = "1.0";
    html5rocks.indexedDB.db = e.target.result;
    var db = html5rocks.indexedDB.db;
    // We can only create Object stores in a setVersion
transaction;
    if(v!= db.version) {
      var setVrequest = db.setVersion(v);

      // onsuccess is the only place we can create Object Stores
      setVrequest.onfailure = html5rocks.indexedDB.onerror;
      setVrequest.onsuccess = function(e) {
        var store = db.createObjectStore("todo",
          {keyPath: "timeStamp"});

        html5rocks.indexedDB.getAllTodoItems();
      };
    }

    html5rocks.indexedDB.getAllTodoItems();
  };

  request.onfailure = html5rocks.indexedDB.onerror;
}
```

Приведенный выше код действительно выполняет важные задачи. Мы используем метод open API для открытия база данных todos. Запрос на открытие не выполняется сразу. Вместо него возвращается запрос IDBRequest. При завершении выполняемой функции будет вызван метод indexedDB.open. Обычно назначение асинхронного обратного вызова выполняется по-другому, но мы можем добавить собственные обработчики для объекта IDBRequest до выполнения такого вызова.

Если запрос на открытие успешно завершен, выполняется обратный вызов onsuccess. В нем проверяется версия базы данных. Если она не совпадает с нужным нам номером, мы вызываем функцию setVersion.

SetVersion – единственное место в коде, где можно изменить структуру базы данных. В ней можно создавать и удалять хранилища объектов, а также формировать и удалять индексы. Обращение к методу setVersion возвращает объект IDBRequest, в который можно добавить обратные вызовы. После успешного завершения можно приступать к созданию хранилища объектов.

Хранилища объектов создаются с помощью метода createObjectStore. Он принимает название хранилища и объект параметров. Этот объект очень важен, поскольку позволяет задать дополнительные свойства. В данном случае мы устанавливаем параметр keyPath, делающий объекты в хранилище уникальными. В нашем примере это свойство timeStamp. Этот параметр должен быть у каждого объекта в хранилище.

Создав хранилище объектов, мы переходим к методу getAllTodoItems.

Этап 3. Добавление данных в хранилище объектов

Поскольку мы создаем менеджер задач, важно иметь возможность добавлять соответствующие элементы в базу данных. Ниже показано, как это сделать.

Добавление данных в хранилище объектов. Листинг 13.4

```
html5rocks.indexedDB.addTodo = function(todoText) {
  var db = html5rocks.indexedDB.db;
  var trans = db.transaction(["todo"], IDBTransaction.READ_WRITE,
0);
  var store = trans.objectStore("todo");
  var request = store.put({
    "text": todoText,
    "timeStamp" : new Date().getTime()
  });

  request.onsuccess = function(e) {
    // Re-render all the todo's
    html5rocks.indexedDB.getAllTodoItems();
  };

  request.onerror = function(e) {
    console.log(e.value);
  };
};
```

Метод addTodo не представляет никакой сложности: сначала мы получаем ссылку на объект базы данных, запускаем транзакцию READ_WRITE и получаем ссылку на хранилище объектов.

Получив доступ к хранилищу, приложение может выполнить простую команду put для простого JSON-объекта. Обратите внимание на свойство timeStamp: уникальный ключ объекта, который используется как параметр

keyPath. Если команда put выполнена успешно, срабатывает событие onsuccess и можно выводить содержание на экран.

Этап 4. Запрос данных из хранилища

После добавления данных в базу потребуется возможность удобного доступа к ним. К счастью, все довольно просто.

Доступ к индексированным базам данных. Листинг 13.5

```
html5rocks.indexedDB.getAllTodoItems = function() {
  var todos = document.getElementById("todoItems");
  todos.innerHTML = "";

  var db = html5rocks.indexedDB.db;
  var trans = db.transaction(["todo"], IDBTransaction.READ_WRITE,
0);
  var store = trans.objectStore("todo");

  // Get everything in the store;
  var keyRange = IDBKeyRange.lowerBound(0);
  var cursorRequest = store.openCursor(keyRange);

  cursorRequest.onsuccess = function(e) {
    var result = e.target.result;
    if(!!result == false)
      return;

    renderTodo(result.value);
    result.continue();
  };

  cursorRequest.onerror = html5rocks.indexedDB.onerror;
};
```

Обратите внимание на то, что все команды, используемые в нашем примере, являются асинхронными, поэтому данные не возвращаются из транзакции.

Код выполняет транзакцию и создает экземпляр keyRange для поиска по данным. Диапазон keyRange задает набор данных, запрашиваемых из хранилища. Поскольку параметр keyPath в хранилище является временной меткой с числовым значением, мы устанавливаем минимальное значение 0 (для поиска с начала периода), чтобы получить все данные.

Теперь у нас есть транзакция, ссылка на хранилище, к которому создается запрос, и диапазон для перебора данных. Остается только установить курсор и добавить событие onsuccess.

Результаты передаются через обратный вызов в курсоре, в котором результаты выводятся на экран. Обратный вызов выполняется по одному разу на ре-

зультат, поэтому для перехода к следующему элементу данных необходимо вызвать функцию continue для объекта результата.

Этап 4A. Вывод данных из хранилища объектов

После извлечения данных из хранилища объектов вызывается функция renderTodo для каждого результата в курсоре.

Вызов метода renderTodo. Листинг 13.6

```
function renderTodo(row) {
  var todos = document.getElementById("todoItems");
  var li = document.createElement("li");
  var a = document.createElement("a");
  var t = document.createTextNode();
  t.data = row.text;

  a.addEventListener("click", function(e) {
    html5rocks.indexedDB.deleteTodo(row.text);
  });

  a.textContent = " [Delete]";
  li.appendChild(t);
  li.appendChild(a);
  todos.appendChild(li)
}
```

Для каждого объекта задачи на основе его текста создается элемент пользовательского интерфейса, включая кнопку "Удалить", позволяющую стереть этот объект.

Этап 5. Удаление данных из таблицы

Удаление данных из таблицы. Листинг 13.7

```
html5rocks.indexedDB.deleteTodo = function(id) {
  var db = html5rocks.indexedDB.db;
  var trans = db.transaction(["todo"], IDBTransaction.READ_WRITE,
0);
  var store = trans.objectStore("todo");

  var request = store.delete(id);

  request.onsuccess = function(e) {
    html5rocks.indexedDB.getAllTodoItems();  // Refresh the screen
  };

  request.onerror = function(e) {
    console.log(e);
  };
};
```

Удаление данных из хранилища, как и их добавление, выполняется очень просто. Достаточно запустить транзакцию, создать ссылку на нужный объект в хранилище и выполнить команду delete для уникального идентификатора этого объекта.

Этап 6. Запуск

После загрузки страницы откройте базу данных, при необходимости создайте таблицу и выведите на экран все задачи из базы данных.

Вызов данных. Листинг 13.8

```
function init() {
  html5rocks.indexedDB.open(); // Открытие ранее сохраненной базы
}

window.addEventListener("DOMContentLoaded", init, false);
```

Для получения данных из модели (DOM) используйте метод html5rocks.indexedDB.addTodo.

Получение данных из модели. Листинг 13.9

```
function addTodo() {
  var todo = document.getElementById('todo');

  html5rocks.indexedDB.addTodo(todo.value);
  todo.value = '';
}
```

Конечный результат:

Конечный результат. Листинг 13.10

```
var html5rocks = {};
window.indexedDB = window.indexedDB || window.webkitIndexedDB ||
                   window.mozIndexedDB;

if ('webkitIndexedDB' in window) {
  window.IDBTransaction = window.webkitIDBTransaction;
  window.IDBKeyRange = window.webkitIDBKeyRange;
}

html5rocks.indexedDB = {};
html5rocks.indexedDB.db = null;

html5rocks.indexedDB.onerror = function(e) {
  console.log(e);
};

html5rocks.indexedDB.open = function() {
  var request = indexedDB.open("todos");

  request.onsuccess = function(e) {
```

```
    var v = 1;
    html5rocks.indexedDB.db = e.target.result;
    var db = html5rocks.indexedDB.db;
    // We can only create Object stores in a setVersion
transaction;
    if (v != db.version) {
      var setVrequest = db.setVersion(v);

      // onsuccess is the only place we can create Object Stores
      setVrequest.onerror = html5rocks.indexedDB.onerror;
      setVrequest.onsuccess = function(e) {
        if(db.objectStoreNames.contains("todo")) {
          db.deleteObjectStore("todo");
        }

        var store = db.createObjectStore("todo",
          {keyPath: "timeStamp"});
        e.target.transaction.oncomplete = function() {
          html5rocks.indexedDB.getAllTodoItems();
        };
      };
    } else {
      request.transaction.oncomplete = function() {
        html5rocks.indexedDB.getAllTodoItems();
      };
    }
  };
  request.onerror = html5rocks.indexedDB.onerror;
};

html5rocks.indexedDB.addTodo = function(todoText) {
  var db = html5rocks.indexedDB.db;
  var trans = db.transaction(["todo"], "readwrite");
  var store = trans.objectStore("todo");

  var data = {
    "text": todoText,
    "timeStamp": new Date().getTime()
  };

  var request = store.put(data);

  request.onsuccess = function(e) {
    html5rocks.indexedDB.getAllTodoItems();
  };

  request.onerror = function(e) {
    console.log("Error Adding: ", e);
  };
};

html5rocks.indexedDB.deleteTodo = function(id) {
  var db = html5rocks.indexedDB.db;
  var trans = db.transaction(["todo"], "readwrite");
```

```
  var store = trans.objectStore("todo");

  var request = store.delete(id);

  request.onsuccess = function(e) {
    html5rocks.indexedDB.getAllTodoItems();
  };

  request.onerror = function(e) {
    console.log("Error Adding: ", e);
  };
};

html5rocks.indexedDB.getAllTodoItems = function() {
  var todos = document.getElementById("todoItems");
  todos.innerHTML = "";

  var db = html5rocks.indexedDB.db;
  var trans = db.transaction(["todo"], "readwrite");
  var store = trans.objectStore("todo");

  // Get everything in the store;
  var cursorRequest = store.openCursor();

  cursorRequest.onsuccess = function(e) {
    var result = e.target.result;
    if(!!result == false)
      return;

    renderTodo(result.value);
    result.continue();
  };

  cursorRequest.onerror = html5rocks.indexedDB.onerror;
};

function renderTodo(row) {
  var todos = document.getElementById("todoItems");
  var li = document.createElement("li");
  var a = document.createElement("a");
  var t = document.createTextNode(row.text);

  a.addEventListener("click", function() {
    html5rocks.indexedDB.deleteTodo(row.timeStamp);
  }, false);

  a.textContent = " [Delete]";
  li.appendChild(t);
  li.appendChild(a);
  todos.appendChild(li);
}

function addTodo() {
  var todo = document.getElementById("todo");
  html5rocks.indexedDB.addTodo(todo.value);
```

```
  todo.value = "";
}

function init() {
  html5rocks.indexedDB.open();
}

window.addEventListener("DOMContentLoaded", init, false);
```

Краткий справочник API IndexedDB (индексированные базы данных)

API индексированных баз данных работает на низком уровне. Данный API включает в себя несколько интерфейсов. Например, существует специальный интерфейс для управления организацией баз данных, еще один интерфейс — для создания хранилищ объектов и манипулирования ими и т.д. Каждый интерфейс имеет свои собственные методы и свойства.

Интерфейс среды (IDBEnvironment и IDBFactory)
Интерфейс среды, или IDBEnvironment включает в себя атрибут IDBFactory. Совместно, эти элементы предоставляют элементы, необходимые для работы с базами данных.

- **indexedDB**. Этот атрибут обеспечивает механизм доступа к системе индексированных баз данных.
- **open(name)**. Этот метод открывает базу данных с указанным именем. Если такой базы нет, то создается новая.
- **deleteDatabase(name)**. Этот метод удаляет базу данных с указанным именем.

Интерфейс базы данных (IDBDatabase)
Объект, возвращаемый после открытия или создания базы данных, обрабатывается именно этим интерфейсом. Для работы с объектом предусмотрены следующие методы и свойства

- **version**. Свойство возвращает текущую версию открытой базы данных
- **name**. Свойство возвращает название открытой базы данных.
- **objectStoreNames**. Свойство возвращает список названий хранилищ объектов в откртой базе данных.
- **setVersion(value)**. Метод устанавливает новое значение версии для открытой базы данных. В качестве атрибута value можно передавать любую строку.
- **createObjectStore(name, keyPath, autoIncrement)**. Этот метод создает новое хранилище объектов в открытой базе данных. Атрибут name представляет собой хранилище объектов, атрибут keyPath — это общий индекс для всех объектов в данном хранилище, а autoIncrement – булево значение, позволяющее активировать генератор ключей.
- **deleteObjectStore(name)**. Этот метод удаляет хранилище объектов, имя которого передано ему в атрибуте name.

- **transaction(stores, type, timeout)**. Этот метод инициализирует транзакцию. Транзакция связывается с одним или несколькими хранилищами объектов, объявленными в атрибуте stores, и допускает различные режимы доступа в соответствии со значениями атрибута type. Также методу можно передать атрибут timeout со значением в миллисекундах, чтобы ограничить время выполнения операции.

Интерфейс хранилища объектов (IDBOjectStore)

Этот интерфейс представляет все методы и свойства, необходимые для манипулирования объектами в хранилище объектов.

- **name**. Свойство возвращает имя используемого в данный момент хранилища объектов.
- **keyPath**. Свойство возвращает значение keyPath, если оно определено, для используемого в данный момент хранилища объектов.
- **IndexNames**. Свойство возвращает список имен индексов, определенных для используемого в данный момент хранилища объектов.
- **add(object)**. Метод добавляет в выбранное хранилище объектов новый объект с информацией из атрибутов. Если объект с таким индексом уже существует, то возвращается ошибка. В качестве атрибута метод может принимать как пару из ключевого слова и значения, так и JSON-объект.
- **put(object)**. Метод добавляет в выбранное хранилище объектов объект с информацией из атрибутов. Если объект с таким индексом уже существует, то он перезаписывается новой информацией. В качестве атрибута может принимать как пару из ключевого слова и значения, так и JSON-объект.
- **get(key)**. Этот метод возвращает объект, индекс которого соответствует значению key.
- **delete(key)**. Метод удаляет объект, индекс которого соответствует значению key.
- **createIndex(name, property, unique)**. Метод создает новый индекс для выбранного хранилища объектов. Атрибут name содержит название индекса, атрибут property объявляет свойство объектов, с которым этот индекс будет связан, а атрибут unique указывает, допустимо ли наличие нескольких объектов с одинаковыми значениями индекса.
- **index(name)**. Метод открывает индекс с названием, соответствующим атрибуту name.
- **deleteIndex(name)**. Метод удаляет индекс с названием, соответствующим атрибуту name.
- **openCursor(range, direction)**. Метод создает курсор над объектом из выбранного хранилища объектов.

Интерфейс курсора (IDBCursor)

Этот интерфейс представляет конфигурационные значения для настройки порядка следования объектов, выбранных из хранилища объектов. Эти константы передаются в качестве второго атрибута метода openCursor, например

openCursor(null, IDBcursor.PREV)

Возможны следующие значения:

- **NEXT**.
- **NEXT_NO_DUPLICATE**.
- **PREV**.
- **PREV_NO_DUPLICATE**.

В интерфейсе также предусмотрено несколько методов и свойств для манипулирования объектами, на которые указывает курсор.

- **continue(key)**. Метод перемещает указатель курсора на следующий объект в списке или на объект, определяемый атрибутом key, если он существует.
- **delete()**. Метод удаляет объект, на который в данный момент указывает курсор.
- **update(value)**. Метод обновляет объект, на который в данный метод указывает курсор.
- **key**. Свойство возвращает значение индекса для объекта, на который в данный момент указывает курсор.
- **value**. Свойство возвращает значение любого свойства, на который в данный момент указывает курсор.
- **direction**. Свойство возвращает порядок следования объектов, которые считываются куросром.

Интерфейс транзакций (IDBTransaction)

Этот интерфейс представляет конфигурационные значения для задания типа очередной транзакции. Эти значения передаются во втором атрибуте метода transaction(), например

transaction(stores, IDBTransaction.READ_WRITE)

Возможные значения:

- **READ_ONLI**. Константа настраивающая транзакцию только на чтение (значение по умолчанию).
- **READ_WRITE**. Константа, настраивающая транзакцию на чтение и запись.
- **VERSION_CHANGE**. Константа для обновления номера версии.

Интерфейс диапазона (IDBKeyRangeConstructors)

Этот интерфейс представляет несколько методов построения диапазона для выбора данных с помощью курсора.

- **only(value).** Возвращает диапазон, начальная и конечная точка которого равны value.
- **bound(lower, upper, lowerOpen, upperOpen).** Возвращает диапазон, начальная точка которого равна lower, а конечная — upper. Также можно указать, нужно ли исключать граничные точки из возвращаемого списка объектов.
- **lowerBound(value, open)**. Возвращает диапазон, начинающийся с value, и продолжающийся до конца списков объектов. Атрибут open определяет, исключается объект, соответствующий value из результирующего списка или нет.
- **upperBound(value, open).** Возвращает диапазон, начало которого совпадает с началом списка объектов и который заканчивается на value. Атрибут open определяет, исключается объект, соответствующий value из результирующего списка.

Интерфейс ошибок (IDBDatabaseException)
Через этот интерфейс передаются ошибки, возвращаемые операциями над базой данных:

- **code**. Свойство содержащее кодовый номер ошибки.
- **message**. Свойство содержащее описание ошибки.

Также возвращаемые значения можно сравнить со следующим списком, чтобы найти соответствующую ошибку.

- **UNKNOWN_ERR** – значение 0
- **NON_TRANSIENT_ERR** – значение 1
- **NOT_FOUND_ERR** – значение 2
- **CONSTRAINT_ERR** – значение 3
- **DATA_ERR** – значение 4
- **NOT_ALLOWED_ERR** – значение 5
- **TRANSACTION_INACTIVE_ERR** – значение 6
- **ABORT_ERR** – значение 7
- **READ_ONLY_ERR** – значение 11
- **RECOVERABLE_ERR** – значение 21
- **TRANSIENT_ERR** – значение 31
- **TIMEOUT_ERR** – значение 32
- **DAEDLOCK_ERR** – значение 33

14. Файловый API

По сути, файловый API позволяет нам взаимодействовать с локальными файлами и обрабатывать их содержимое из приложения, а также записывать содержимое в файлы и управлять файловой системой, в том числе и каталогами.
При работе с файлами, разработчикам необходимо учитывать множество аспектов безопасности.

Файловый API предоставляет только два способа загрузки файлов: тэг <input> и операцию перетаскивания.

Рассмотрим шаблон для выбора файлов через тэг <input>

Шаблон для работы с файлами пользователя. Листинг 14.1

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>File API</title>
  <style>
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
.directory{
  color: #0000FF;
  font-weight: bold;
  cursor: pointer;
}
  </style>
  <script src="file.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <label for="myfiles">File: </label><br>
      <input type="file" name="myfiles" id="myfiles">
    </form>
  </section>
  <section id="databox">
    No File Selected
  </section>
</body>
</html>
```

Для считывания информации с файла пользователя, имеется конструктор FileReader. Этот конструктор возвращает объект с несколькими методами, позволяющими добраться до содержимого файла:

- **readAsText(file, encoding)**. Применяется для обработки текстового содержимого. Содержимое файла возвращается в виде текста в кодировке UTF-8, если только в атрибуте encoding не задан какой-либо

другой вариант кодирования. Данный метод пытается интерпретировать каждый байт многобайтовой последовательности символов, как текстовый символ.

- **readAsBinaryString(file)**. Этот метод считывает информацию как последовательность целых чисел в диапазоне от 0 до 255. Он просто просматривает каждый байт, не пытаясь никак интерпретировать его значение.
- **readAsDataURL(file)**. Генерирует предсталяющий содержимое файлов URL данных (data:url) в кодировке base64.
- **readAsArrayBuffer(file)**. Генерирует на основе данных файла данные в формате ArrayBuffer.

Считывание текстового файла. Листинг 14.2

```
var databox;
function initiate(){
  databox = document.getElementById('databox');
  var myfiles = document.getElementById('myfiles');
  myfiles.addEventListener('change', process);
}
function process(e){
  var files = e.target.files;
  var myfile = files[0];
  var reader = new FileReader();
  reader.addEventListener('load', show);
  reader.readAsText(myfile);
}
function show(e){
  var result = e.target.result;
  databox.innerHTML = result;
}
addEventListener('load', initiate);
```

Если мы хотим обработать файл, то в первую очередь должны с помощью конструктора FileReader() получить объект FileReader. В функции proecess() мы присваиваем этому объекту имя reader. После этого для объекта reader регистрируется событие onload, который позволяет распознать ситуацию, когда файл готов к обработке.

Свойства файлов

Объект файла, отправляемый тегом <input>, предоставляет несколько свойств, позволяющих получить эту информацию.

- **name**. Возвращает полное имя файла (название и расширение).
- **size**. Возвращает размер файла в байтах.
- **type**. Возвращает тип файла как тип MIME

Свойства загруженных файлов. Листинг 14.3

```
var databox;
function initiate(){
```

```
  databox = document.getElementById('databox');
  var myfiles = document.getElementById('myfiles');
  myfiles.addEventListener('change', process);
}
function process(e){
  var files = e.target.files;
  databox.innerHTML = '';
  var myfile = files[0];
  if(!myfile.type.match(/image.*/i)){
    alert('insert an image');
  }else{
    databox.innerHTML += 'Name: ' + myfile.name + '<br>';
    databox.innerHTML += 'Size: ' + myfile.size + ' bytes<br>';

    var reader = new FileReader();
    reader.addEventListener('load', show);
    reader.readAsDataURL(myfile);
  }
}
function show(e){
  var result = e.target.result;
  databox.innerHTML += '<img src="' + result + '">';
}
addEventListener('load', initiate);
```

На этот раз мы применили метод readAsDataURL() для считывания информации о файлах. Если перед нами стоит задача обработать файл определенного типа, первым делом мы проверяем свойство этого файла под названием type.

Контроль процесса загрузки файлов

Используя свойства loaded и total, можно разработать систему контролирования процесса загрузки файлов.

Контролирование процесса загрузки. Листинг 14.4

```
var databox;
function initiate(){
  databox = document.getElementById('databox');
  var myfiles = document.getElementById('myfiles');
  myfiles.addEventListener('change', process);
}
function process(e){
  databox.innerHTML = '';
  var files = e.target.files;
  var myfile = files[0];
  var reader = new FileReader();
  reader.addEventListener('loadstart', start);
  reader.addEventListener('progress', status);
  reader.addEventListener('loadend', function(){ show(myfile); });
  reader.readAsBinaryString(myfile);
}
function start(e){
```

```
  databox.innerHTML = '<progress value="0"
max="100">0%</progress>';
}
function status(e){
  var per = parseInt(e.loaded / e.total * 100);
  databox.innerHTML = '<progress value="' + per + '" max="100">' +
per + '%</progress>';
}
function show(myfile){
  databox.innerHTML = 'Name: ' + myfile.name + '<br>';
  databox.innerHTML += 'Type: ' + myfile.type + '<br>';
  databox.innerHTML += 'Size: ' + myfile.size + ' bytes<br>';
}
addEventListener('load', initiate);
```

Одновременное считывание нескольких файлов

Контролирование процесса загрузки. Листинг 14.5

```
<input id="myfiles" name="myfiles" type="file" multiple />
```

Данный стандарт предоставляет возможность считывать несколько файлов, но это нужно явно указать, вставив атрибут multiple в элемент <input>. Теперь в диалоговом окне пользователь может выбрать несколько файлов: очертив их при нажатой левой кнопки мыши или удерживая клавишу <Ctr>. Нам нужен цикл for, который обрабатывает все файлы по одному за каждый проход цикла.

Обработка множества файлов с помощью цикла for. Листинг 14.6

```
for(var i=0; i<files.length; i++){
 var file = files[i];
 var reader = new FileReader();
 ...
}
```

API File применяется для загрузки файлов с компьютера, и их последующей обработки. Новые файлы или каталоги создаются с помощью API File: Directories and System (Каталог и система) и API FILE: Writer (запись файлов).

15. Взаимодействие с PHP

Основная работа по взаимодействию браузера с сервером ложится на сервер, т.е. на формирование ответов, которые необходимо передать браузеру.

Браузерная часть достаточно простая: необходимо создать объект EventSource, куда передать адрес файла выполняемого на сервере.

Таким образом браузер начинает прослушку данного файла.

Далее мы можем использовать метод onmessage - для вывода новых сообщений. Или close для закрытия соединения.

Рассмотрим браузерную часть:

Браузерная часть. Листинг 15.1

```
<script>
 var messageLog;
 var timeDisplay;
 var source;
 window.onload = function(){
  messageLog = document.getElementById('messageLog');
  timeDisplay = document.getElementById('timeDisplay');
 }

 function start(){
  source = new EventSource('events.php');
  source.onmessage = function(e){
    messageLog.innerHTML += "<br /> Новое сообщение" + e.data;
    timeDisplay.innerHTML = e.data;
  }
  messageLog.innerHTML += "<br /> Начинаем прослушку";
 }
 function stop() {
  source.close();
  messageLog.innerHTML += "<br /> Закрыли";
 }
</script>
<div id="messageLog"></div>
<div id="timeDisplay"></div>
 <div id="controls">
  <button id="startlistening" onclick="start()">Старт</button>
  <button id="stoplistening" onclick="stop()">Стоп</button>
 </div>
```

Сообщение формируется на стороне сервера и разрешается разбить на несколько частей. Для этого используется последовательность символов окончания строки, которая часто состоит из одной пары \n. Это облегчает отправку сложных сообщений, например:

data: Это сообщение было отправлено сервером \n
data: Это еще одно сообщение \n\n

Обратите внимание, что каждая часть сообщения начинается с ключевого слова data, а сообщение завершается \n\n.

Этот метод можно использовать для отправки данных в формате JSON, что позволяет преобразовать объект в один прием:

data: {\n
data: "messageType": "StatusUpdate", \n
data: "messageData": "Отравляем сообщение"

data: } \n\n

Вместе с данными сообщениями web-сервер может отправлять идентифицирующее значение (используя префикс id) и время time-out для переподключения (используя префикс retry)

id: 495\n
retry: 15000\n
data: "Текст сообщения" \n

С форматом сообщений разобрались, теперь рассмотрим что должно происходить на сервере. Во первых, в начале серверного сценария, необходимо установить два важных заголовка (это стандарт для отправляемых сервером сообщений): text/event-stream и no-cache.

Далее сервер создает сообщения нужного формата, которое завершается константой PHP_EOL (это представление комбинации символов \n\n, обозначающих конец строки).

Функция flush() обеспечивает немедленную отправку данных, а не помещение их в буфер.

Серверная часть. Листинг 15.2

```
<?php
 header('Content-Type: text/event-stream');
 header('Cache-Control: no-cache');
 echo 'retry: 120 000' . PHP_EOL;
 $startTime = time();
 do {
  $currentTime = date('h:i:s', time());
  echo "data: " . $currentTime . PHP_EOL;
  echo PHP_EOL;
  flush();
  if((time() - $startTime) > 60){
   die();
  }
  sleep(2);
 } while(true);
?>
```

4. Семантически правильная верстка

Правильная семантическая верстка – это результат вдумчивой и кропотливой работы. Начинающему верстальщику можно дать следующие советы:

- изучай HTML, какие теги и для чего предназначены;
- изучай CSS, у этого инструмента большие возможности;
- научись давать грамотные имена классов и идентификаторов;
- начинай осваивать микроформаты;
- и самое главное, начни думать во время верстки. Бездумная работа не принесет ни хороших результатов, ни удовлетворения от работы.

Сейчас семантическая верстка – это не требования, а рекомендация. Но рекомендация, которая уже сейчас несет в себе большие выгоды. Не за горами то время, когда семантический код, будет таким же требованием как дивная верстка. Поэтому лучше начинать себя приучать к "правильному" коду сегодня, чтобы не остаться "за бортом" завтра.

Семантический код для пользователей

Повышает доступность информации на сайте. В первую очередь это имеет значение для альтернативных агентов таких как:

- семантический код напрямую влияет на объем HTML кода. Меньше кода —> легче страницы —> быстрей грузятся, меньше требуется оперативной памяти на стороне пользователя, меньше трафика, меньший объем баз данных. **Сайт становиться быстрей и менее затратным.**
- голосовые браузеры для которых важны теги и их атрибуты, чтобы произнести правильно и с нужной интонацией содержимое, или наоборот не произнести лишнего.
- мобильные устройства которые не на полную мощь поддерживают CSS и поэтому ориентируются в основном на HTML код, отображая его на экране согласно используемым тегам.
- устройства печати даже без дополнительного CSS напечатают информацию качественней (ближе к дизайну), а создание идеальной версии для печати превратится в несколько легких манипуляций с CSS.
- к тому же существуют устройства и плагины, которые позволяют быстро перемещаться по документу — например, по заголовкам у Opera.

Семантический HTML для машин

Поисковые системы постоянно совершенствуют методы поиска, чтобы в результатах была та информация, которую действительно ищет пользователь. Семантический HTML способствует этому, т.к. поддается гораздо лучшему анализу — код чище, код логичен (четко видно где заголовки, где навигация, где содержимое).

Хороший контент плюс качественная семантическая верстка — это уже серьезная заявка на хорошие позиции в выдачах поисковиков.

Профессиональные правила верстки

1. HTML и CSS должны быть **читаемыми** (все равно потом все ужимаем):
    1. HTML должен быть иерархическим, открывающие теги с новой строки (кроме случаев с), отступы 4 пробела
    2. CSS желательно не свернутые в строчку, а по одной директиве на строку, тоже с отступами по 4 пробела
    3. Можно сразу использовать http://sass-lang.com/ SASS — сильно упрощает всем жизнь
    4. Идентификаторы классов должны быть человекочитабельмыми английскими словами и фразами
    5. Очень желательно *не использование* в названиях классов и идентификаторов ключевых слов из HTML/CSS/JS/jQuery
    6. Очень желательно, чтобы идентификаторы и классы *не были* префиксными (ни одно названия не является началом или концом другого, например .icon и .icon_small) — это позволяет сократить их все до 1-2 символов на этапе постобработки
2. **DOCTYPE** предпочтителен HTML5, но в крайнем случае можно и XHTML
3. Кодировка: *обязательно* **UTF-8**
4. **Валидность**:
    1. HTML: http://validator.w3.org
    2. CSS: http://jigsaw.w3.org/css-validator/
    3. WCAG: http://www.cynthiasays.com/ и http://www.w3.org/WAI/WCAG20/quickref/
5. **Соответствие макету**:
    1. Проверка: https://addons.mozilla.org/ru/firefox/addon/pixel-perfect/ и https://github.com/aishek/modulargrid
    2. Хоть и попиксельное соответствие приветствуется и считается обязательным, но местами возможны отклонения:
        1. Когда дизайнер нарисовал криво
        2. Когда незначительные различия в отрисовке шрифтов
        3. Когда у дизайнера разошлось вдохновение и единственный способ сделать точно так как на картинке — впихнуть здоровенный JPEG в пару сот килобайт на фон; в таких случаях стараемся разбить фон на составные части и/или паттер-

ны и делаем его частями, стараясь минимизировать ущерб «креативу» при адекватном размере страницы

6. **Кроссбраузерность**:
    1. Перед приемом/сдачей работы проверить на http://browsershots.org/ (или https://browserlab.adobe.com/) и во всех доступных под рукой браузерах
    2. Не забыть мобильные браузеры Opera Mini, iPhone и Android.
    3. Хаки для IE в conditional comments, не забываем правильно фильтровать для разных его версий
    4. IE6 должен адекватно воспринимать специфичные для HTML5 теги
    5. Помимо этого желательно проверить все разрешения от 1024×768 до 1920×1080 — везде все должно смотреться пристойно, без скроллбаров и излишних пустых областей
7. **Гибкость шаблона**:
    1. При этом не смотря на то, что в макетах для примера используется какой-то текст, шаблон должен нормально себя вести при любом количестве текста (как нулевом, так и очень большом) в любом элементе
    2. Если пункт меню, заголовок или просто текст в 5-10 раз длиннее или короче того, что на макете — все должно отображаться корректно (по возможности, естественно)
    3. Если в макете не было всех шести видов заголовков, ссылок в тексте, картинок, цитат, аббревиатур, блоков pre и code — это не значит, что стиль по-умолчанию для всех стандартных html-элементов может отсутствовать в CSS — возьмите цвета и стиль от других элементов шаблона и сделайте хоть как-то, для образца все элементы можно добавить в текст, даже если их там не было
    4. У всех ссылок должно быть отдельное поведение для :hover, у ссылок внутри текста — и для :visited
    5. Если в макете все же были заголовки: их структура должна быть перенесена верно
    6. Надеяться, что WYSIWYG или Word проставит какие-то особые стили к тексту и все будет нормально *нельзя*
    7. Если используем HTML5, то верстка должна быть *семантической*, то есть блоки навигации заворачиваем в nav, сайдбар в aside и.т.п.
    8. Стили для печати и мобильной версии указываем отдельными файлами с соответствующим media
8. **CSS**:
    1. Не забываем о производительности CSS: селекторы обрабатываются справа-налево, подробнее здесь: http://code.google.com/speed/page-speed/docs/rendering.html

2. CSS3 правила для нормальных браузеров (border-radius, gradient, text-shadow, box-shadow), с остальными извращаемся отдельно, никаких хаков с кучей <b> и подобных
3. Размеры и позиционирование элемента должны указываться в одних единицах измерения

9. **Изображения**:
    1. С умом относимся к выбору формата и уменьшаем их объем (см. http://www.insight-it.ru/tekhnologii/instrumenty-dlya-minimizacii-razmera-izobrazhenijj/)
    2. Иллюстрации и элементы интерфейса с четкими краями лучше сохранять в PNG8, фотографии в progressive JPEG — но в целом руководствуемся минимизацией объема не в ущерб качеству
    3. Отсутствие title и alt *непростительно*
    4. По возможности всегда указываем размеры изображений, особенно если они являются частью дизайна и заранее известны
    5. Для групп небольших изображений приблизительно одинаковой ширины или высоты (иконки, картинки разных буллетов и.т.п.) используем CSS Sprites: http://css-tricks.com/css-sprites/ (для ленивых есть ряд инструментов для автоматизации процесса)
    6. Для совсем маленьких изображений используем data:URI+MHTML: http://www.phpied.com/inline-mhtml-data-uris/
10. **Шрифты**:
    1. Указываем базовый размер шрифта в em, там где нужен размер больше или меньше — в % от базового
    2. Line-height задаем в долях
    3. Должно быть как минимум одному шрифту для Windows, Linux и Mac OS. Примеры:
        1. Arial,Helvetica,FreeSans,"Liberation Sans","Nimbus Sans L",sans-serif
        2. «Courier New»,Courier,FreeMono,"Nimbus Mono L","Liberation Mono",monospace
        3. Georgia,"Bitstream Charter","Century Schoolbook L","Liberation Serif",Times,serif
        4. «Lucida Sans»,"Lucida Sans Unicode","Lucida Grande",Lucida,sans-serif
        5. «Lucida Console»,Monaco,"DejaVu Sans Mono","Bitstream Vera Sans Mono","Liberation Mono",monospace
        6. Palatino,"Palatino Linotype",Palladio,"URW Palladio L","Book Antiqua","Liberation Serif",Times,serif
        7. Tahoma,Geneva,"DejaVu Sans Condensed",sans-serif
        8. «Times New Roman»,Times,"Nimbus Roman No9 L","FreeSerif","Liberation Serif",serif
        9. Verdana,"Bitstream Vera Sans","DejaVu Sans","Liberation Sans",Geneva,sans-serif

4. Если требуется нестандартный шрифт, используем кроссбраузерный font-face: http://randsco.com/index.php/2009/07/04/p680

11. **Формы**:
    1. Ко всем полям должен быть label или, если так задумано, toggleval
    2. Если используем HTML5, то и поля форм тоже делаем по этому стандарту (email/tel)
    3. Проверить работу tabindex
12. Использование **микроформатов** желательно, если они уместны (hCard, hCalendar, hAtom, XFN)
13. **JavaScript**:
    1. Весь используемый JS располагается перед закрывающим </body>
    2. Желательно использование асинхронной загрузки файлов
    3. Работоспособность проверяем средствами Firebug
    4. Сайт должен нормально функционировать с выключенным JavaScript (и Flash тоже, к слову)
    5. Если на странице имеются формы — они должны валидироваться еще до отправки и нормально отображать ошибки (как минимум просто цветом)
14. **Остальное**:
    1. Полное отсутствие комментариев в html, кроме условных
    2. Копирайт пишем правильно: http://habrahabr.ru/blogs/typography/23812/
    3. Никакого CSS или JS внутри HTML, только внешние файлы
    4. Ссылки на внешние ресурсы должны быть с target="_blank" и по необходимости могут снабжаться иконками: https://github.com/joshuaclayton/blueprint-css/blob/master/blueprint/plugins/link-icons/screen.css
    5. Текст лучше оттипографить перед публикацией: http://rmcreative.ru/blog/post/tipograf
    6. Логотип должен вести на главную страницу
    7. Ссылки на внешние ресурсы (изображения, CSS, JS) лучше делать относительными
    8. Не забываем <title> всей страницы, должен быть понятен как человеку, так и поисковым системам
    9. HTML, JS и CSS лучше минимизировать не на этапе верстки, а автоматически при генерации шаблонов
    10. Не забыть выкинуть не используемые стили (вроде красной рамочки для выделения элементов в процессе верстки)

17. LESS – язык стилей

LESS – это динамический язык стилей. Скачать библиблиотеку для работы с LESS можно по адресу: http://lesscss.ru/

LESS обеспечивает следующие расширения CSS: переменные, вложенные блоки, примеси, операторы и функции.

Переменные

Less позволяет использовать переменные. Имя переменной предваряется символом @. В качестве знака присваивания используется двоеточие (:).

При трансляции значение переменной подставляется в результирующий CSS документ.

Переменные в LESS. Листинг 17.1

```
@color: #4D926F;

#header {
  color: @color;
}
h2 {
  color: @color;
}
```

Данный LESS-код будет скомпилирован в следующий CSS-файл:

Компиляция в CSS-файл. Листинг 17.2

```
#header {
  color: #4D926F;
}
h2 {
  color: #4D926F;
}
```

Примеси

Примеси позволяют включать целый набор свойств из одного набора правил в другой путём включения имени класса в качестве одного из свойств другого класса. Такое поведение можно рассматривать как разновидность констант или переменных. Они также могут вести себя подобно функциям, принимая аргументы. В чистом CSS повторяющийся код должен быть повторён в нескольких местах — примеси делают код чище, понятней и упрощают его изменение.

Включение примесей. Листинг 17.3

```
.rounded-corners (@radius: 5px) {
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
   border-radius: @radius;
}

#header {
  .rounded-corners;
}
#footer {
  .rounded-corners(10px);
}
```

Данный LESS-код будет скомпилирован в следующий CSS-файл:

Компиляция в CSS-файл. Листинг 17.4

```
#header {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
   border-radius: 5px;
}
#footer {
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
   border-radius: 10px;
}
```

Вложенные правила

CSS поддерживает логическое каскадирование, но один блок кода в другой вложен быть не может. LESS дает возможность вкладывать определения, вместо либо вместе с каскадированием, т.е. позволяет вложить один селектор в другой. Это делает наследование более ясным и укорачивает таблицы стилей.

Вложенные правила. Листинг 17.5

```
#header {
  h1 {
    font-size: 26px;
    font-weight: bold;
  }
  p { font-size: 12px;
    a { text-decoration: none;
      &:hover { border-width: 1px }
    }
  }
}
```

Данный LESS-код будет скомпилирован в следующий CSS-файл:

Компиляция в CSS-файл. Листинг 17.6

```
#header h1 {
```

```
  font-size: 26px;
  font-weight: bold;
}
#header p {
  font-size: 12px;
}
#header p a {
  text-decoration: none;
}
#header p a:hover {
  border-width: 1px;
}
```

Функции и операции

Less позволяет использовать операторы и функции. Благодаря операциям можно складывать, вычитать, делить и умножать значения свойств и цветов, что можно использовать для создания сложных отношений между свойствами. Функции один-к-одному отображаются в JavaScript код, позволяя обрабатывать значения.

Функции и операции. Листинг 17.7

```
@the-border: 1px;
@base-color: #111;
@red:        #842210;

#header {
  color: @base-color * 3;
  border-left: @the-border;
  border-right: @the-border * 2;
}
#footer {
  color: @base-color + #003300;
  border-color: desaturate(@red, 10%);
}
```

Данный LESS-код будет скомпилирован в следующий CSS-файл:

Компиляция в CSS-файл. Листинг 17.8

```
#header {
  color: #333;
  border-left: 1px;
  border-right: 2px;
}
#footer {
  color: #114411;
  border-color: #7d2717;
}
```

Использование

LESS можно использовать на сайте различными способами. Один из вариантов — подключение к веб-странице JavaScript-файла less.js для преобразования кода в CSS «на лету», средствами браузера.

Это делается, например, с помощью следующего html-кода:

Компиляция в CSS-файл. Листинг 17.9

```
<link rel="stylesheet/less" type="text/css" href="styles.less">
<script src="less.js" type="text/javascript"></script>
```

Если вы используете серверный JavaScript: Rhino или node.js, вы можете преобразовывать .less файлы в .css на стороне сервера.

ЧАСТЬ 2. NODE.JS

Node или Node.js — серверная платформа, использующая язык программирования JavaScript.

Чтобы разобраться как работет Node.js, сперва рассмотрим работу обычного сервера, Apache.

Сервера поддерживают две модели мультипроцессорной обработки:

1) **Мультипроцессорный** поток. Для каждого запроса выделяется отдельный процесс, продолжающийся до тех пор, пока запрос не будет обслужен. Под каждый запрос создаются дочерние процессы. Недостаток: каждый процесс расходует память.

2) **Мультипрограммный** поток. Для каждого запроса выделяется отдельный программный поток. Такой подход эффективнее, т.к. требует меньшего расхода памяти.

Независимо от потока, если к приложению обращается несколько человек, сервер все запросы обрабатывает одновременно.

В Node.js под каждый запрос создается единственный программный поток. Node-приложение выполняется в этом потоке и ожидает, что некое приложение сделает запрос. Когда node-приложение получает запрос, никакие другие запросы не обрабатываются до тех пор, пока не завершится обработка текущего запроса. При этом Node-приложение работает в **асинхронном режиме, используя цикл обработки событий и функции обратного вызова**. Приложение Node.js получая запрос, не ожидает ответа на этот запрос. Вместо этого, запросу присваивается функция обратного вызова.

Итак, Node.js, не дожидаясь ответа, построчно запускает множество процессов. Это значит, что когда мы построчно вызываем несколько функций, мы не можем знать, какая из этих функций выполнится первой. Также не стоит забывать ставить ключевое слово var перед переменной, т.к. возможны ошибки с использованием глобальных переменных. Во всем остальном Node.js похож на JavaScript.

Саму платформу необходимо скачать (сайт nodejs.org) и установить.

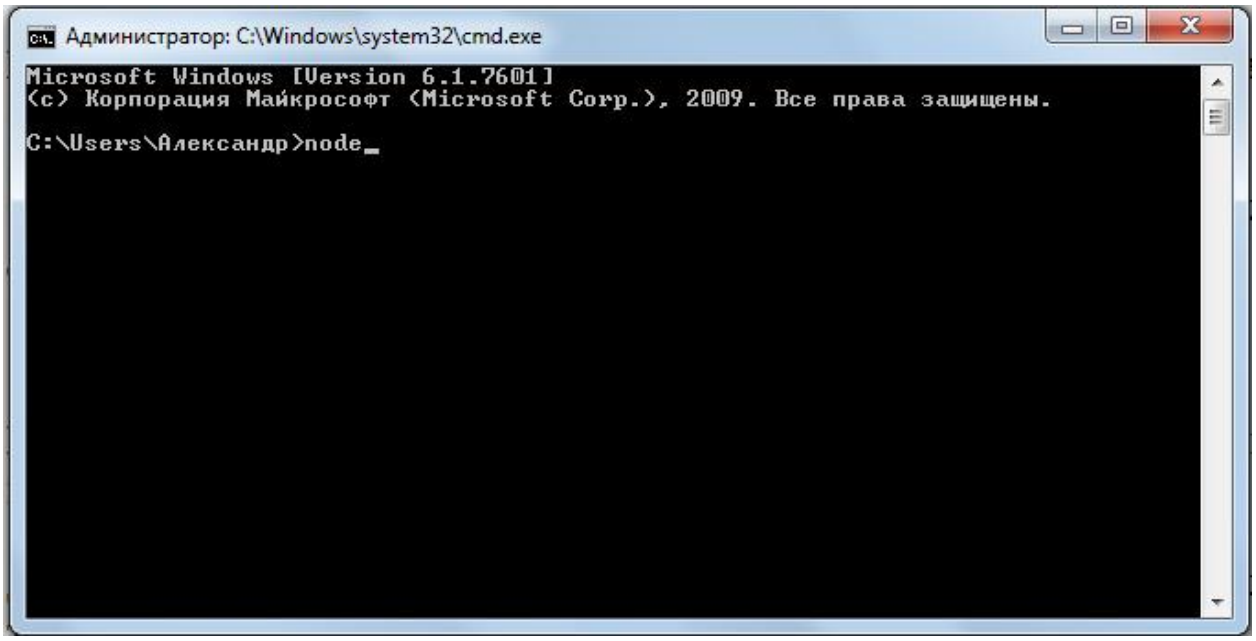
Работать с Node.js можно из консоли командной строки либо с помощью надстроек для IDE (например, PHPShtorm). После установки платформы все Node-команды (в том числе создание и управление файлами) можно осуществлять с помощью режима REPL (запуск из командной консоли).

1. Цикл чтения, вычисления и вывода на экран REPL (запуск Node-приложений из режима командной консоли)

REPL – это режим командной консоли для Node. REPL – встроенный компонент Node.js.

Для того, чтобы его запустить, сперва откроем командную строку (сочитание клавиш **cmd**).

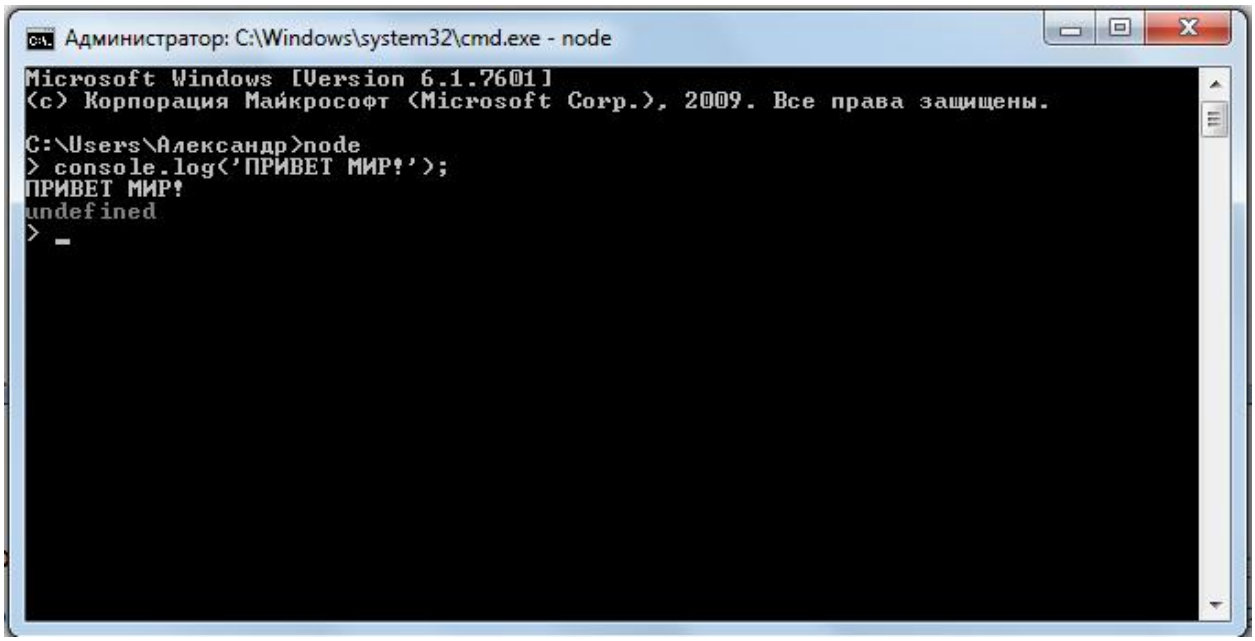
В командной строке наберем команду node.



Это всё, что необходимо сделать, чтобы запустить REPL.

REPL представляет приглашение командной строки, символом которой по умолчанию является угловая скопка (>). Все команды после этой скопки обрабатываются JavaScript движком.

Пользоваться REPL просто. Нужно просто набирать JavaScript код. При этом REPL выводит на экран только что набранные выражения.



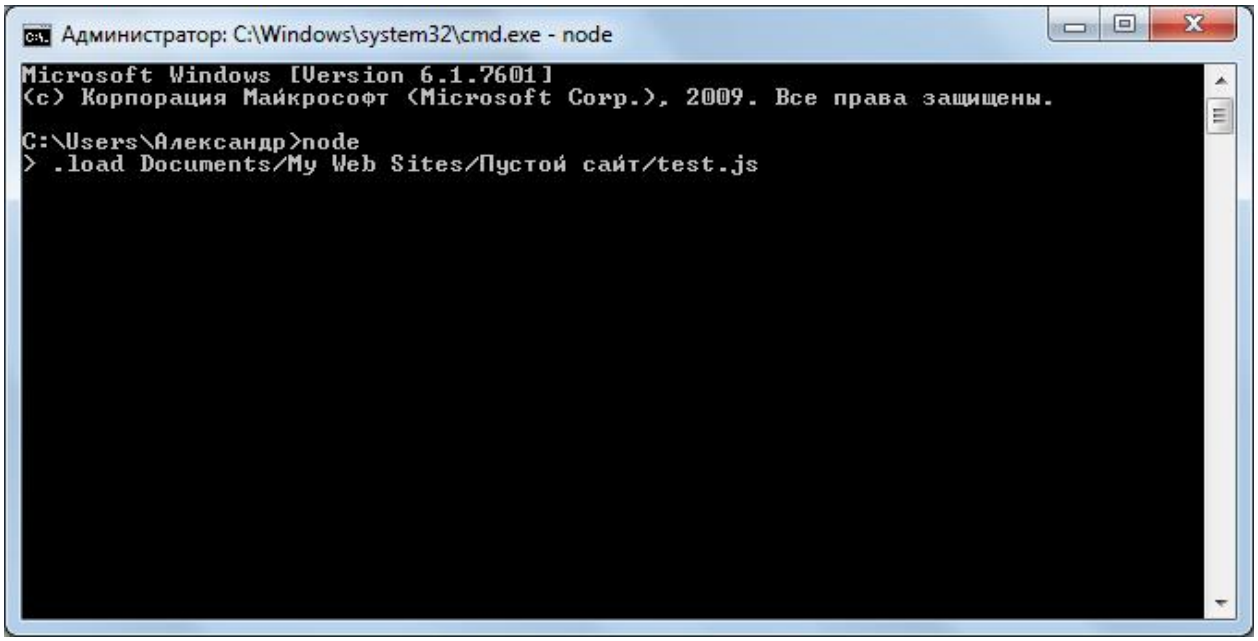
Клавиатурные команды REPL

Ctrl+C – Завершает выполнение текущей команды. Повторное нажатие приводит к выходу из REPL.
Ctrl+D – Выход из REPL.
Tab – Автоматическое завершение имени глобальной или локальной переменной.
Стрелка вверх – Проход вверх по списку введенных команд.
Стрелка вниз – Проход вниз по списку введенных команд.
Подчеркивание (_) – Ссылка на результат вычисления последнего выражения.

REPL-команды

.save – сохраняет в файле всё, что было написано в текущий объект контента.
.break – возвращает к самому началу введенного кода, но весь многострочный ввод при этом будет потерян.
.clear – перезапуск объекта контента и очистка любого многострочного выражения. Команда запускает сеанс с самого начала.
.exit – выход из REPL
.help – вывод всех доступных REPL команд.
.load - загрузка файла в сеанс (.load путь/к/файлу.js)

Загрузим файл test.js из папки Александр/Мои документы/My Web Sites/Пустой сайт



Для усовершенствования строкового редактирования, изменения цвета REPL и надежного сохранения истории ввода команд можно использовать специальные утилы, либо создать собственную нестандартную версию REPL.

Для разработки собственной нестандартной версии REPL, необходимо подулючить модуль repl.

Подключение модуля repl. Листинг 1.1

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

Далее, для объекта repl вызывается метод start со следующими параметрами:

Вызов метода start для объекта repl. Листинг 1.2

```
repl.start([prompt], [stream], [eval], [useGlobal], [ignoreUnde-
fined]);
```

Все параметры не обязательны. Если они отсутствуют используется значение по умолчанию.

prompt – приглашение для ввода, по умолчанию >

stream – входящий или исходящий потоки. По умолчанию входящий (input) поток для прослушивания process.stdin. Output (исходящий) поток для записи – process.stdout.

eval – функция которая будет использоваться для каждой линии потока. По умолчанию – async.

useGlobal – служит для запуска нового контента, вместо использования глобального объекта. По умолчанию false.

ignoreUndefined – запрет на игнорирование неопределенных (undefined) ответов.

Пример создания собственной версии REPL

Пример создания собственной версии REPL. Листинг 1.3

```
repl = require('repl');
repl.start('Ok>>', null, null, null, true);
```

2. Ядро Node

Ядро Node – это программный интерфейс, предоставляющий основную функциональность для создания node-приложений.

В ядро Node входит следующая функциональность: **глобальные Node-объекты** (global, process, buffer, require(), console()), **таймерные методы** (setTimeout, clearTimeout, setInterval, clearInterval), службы прослушивания, дочерние процессы, система доменных имен, модули для тестирования и форматирования, объектное наследование, события, работа с файлами.

Глобальные объекты – это объекты, доступны всем Node.js приложениям без подключения каких либо модулей.

Основная часть ядра Node.js предназначена для создания служб прослушивания конктретных видов взаимодействий. Например, существуют методы, позволяющие создать HTTP-сервер, TCP-сервер, TLS-сервер и сокеты.

Сокет – это конечная точка соединения. **Сетевой сокет** – это конечная точка соединения между двумя компьютерами в сети. Сокеты переносят данные используя **потоки ввода-вывода**. Данные в потоке передаются в **пакетах** (фрагменты данных определенного размера), как двоичные данные или как строки в кодировке utf8.

Объект Global

Представляет собой глобальное пространство имен. Любая определяемая переменная становится свойством объекта Global.

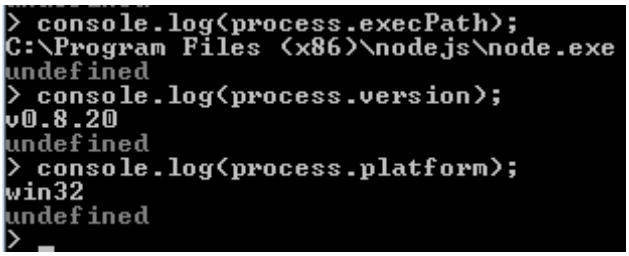
Объект Process

Многие методы и свойства объекта Process предоставляют информацию о приложении и его среде.

Process.execPath – возвращает путь выполнения для Node-приложения

Process.version – возвращает версию Node

Process.platform – возвращает платформу сервера.

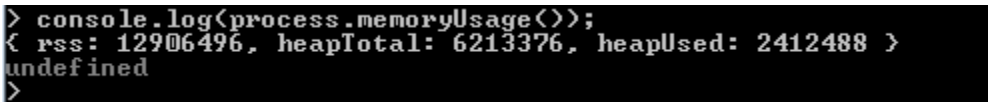


Метод объекта process – **memoryUsage**, сообщающет сколько памяти расходует Node-приложение.

Метод memmoryUsage объекта process. Листинг 2.1

```
console.log(process.memoryUsage);
```

Выполнив данный листинг, получим следующее:



Объект Process также служит оболочкой для стандартных потоков ввода-вывода *stdin, stdout* и *stderr*. Потоки stdin и stdout являются асинхронными и доступными по чтению и записи. Когда мы что-то пишем в консоли (или в приложении Node), срабатывает поток stdin. Когда консоль отвечает (что-то выводит на экран), срабатывает поток stdout. Поток stderr является синхронным, блокирующим.

С помощью этих потоков, мы можем вмешиваться в процесс записи и вывода.

Все эти коммуникационные потоки являются реализацией. С помощью потоков ввода-вывода, можно создать канал передачи данных между потоком чтения и потоком записи. Продемонстрируем это, откры REPL-сеанс и введем следующий код:

Канал pipe потока stdin. Листинг 2.2

```
process.stdin.resume(); // подготовка к вводу с терминала
process.stdin.pipe(process.stdout);
```

Далее, всё, что мы будем вводить в консоль, будет тут же выводиться на экран.

Рассмотрим еще один пример:

Чтение и запись данных с использованием потоков stdin и stdout объекта

process. Листинг 2.3

```
process.stdin.resume(); // по умолчанию поток stdin приостанов-
лен, поэтому сперва нам необходимо его возобновить.
 process.stdin.on('data', function(chunk){
   process.stdout.write('data: ' + chunk);
})
```

После запуска данного приложения в консоли, изменится формат ввода и вывода данных. Это становится заметным при дальнейшем наборе кода в консоли.

Еще один полезный метод объекта process – **nextTick**, который используется, когда нужно приостановить функцию в асинхронном режиме.

Метод nextTick объекта process. Листинг 2.4

```
function async = function(data, callback){
  process.nextTick(function(){
    callback(val)
  });
}
```

Данный метод позволяет строго задать последовательность выполнения кода. С одной стороны, nextTick гарантирует что функция выполнится до того, как придут следующие события. С другой стороны, он делает выполнение функции асинхронным.

process.nextTick(). Листинг 2.5

```
var http = require('http');
http.createServer(function(req, res){
 process.nextTick(function(){
  req.on('readable', function(){
  });
 });
}).listen(1337)
```

Вместо метода nextTick можно было бы использовать метод setTimeout с нулевой задержкой, однако метод nextyTick вызывается намного быстрее. Кроме того, данный метод позволяет разбить процесс на этапы для последовательного вызова каждого процесса.

Объект Buffer

Глобальный объект, предоставляющий простое хранилище данных и средства управления этим хранилищем.

Создать новый буфер можно следующим образом:

Создание нового буфера. Листинг 2.6

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

Если в буфере хранится строка, можно передать второй необязательный параметр, указывающий на кодировку. Возможны следующие варианты: ascii (Семибитный код), utf8 (юникод-символы с многобайтной кодировкой), usc2 (юникод-символы с двухбайтной кодировкой), base64 (кодировка), hex (кодировка каждого байта ввиде двух шестнадцатиричных чисел).

По умолчанию, используется кодировка utf-8.

Объект Require()

Предназначен для подключения модулей. *Require.resolve()* – предназначен для определения, какой модуль загружен. *Require.cache()* – для кэширования подключений.

Объект Console()

Используется для вывода на экран. Пример использования

Использование console.log. Листинг 2.7

```
console.log('сообщение');
```

Таймерные функции setTimeout, clearTimeout, setInterval и clearInterval

Рассмотрим функцию setTimeout. В качестве первого параметра, используется функция обратного вызова, второй параметр – время задержки в милисекундах (причем, нет никаких гарантий, что функция обратного вызова сработает ровно через n милисекунд, независимо от значения n, поскольку мы не можем полностью контроллировать серверную среду), после чего может следовать необязательные параметры настроек.

Использование setTimeout. Листинг 2.8

```
setTimeout(function(){
 callback(val)
}, 2000);
```

И еще один пример использования setTimeout с дополнительным параметром и с вызовом внешней функции.

Использование setTimeout с дополнительным параметром. Листинг 2.9

```
setTimeout(myfunc, 2000, morevar);
function(morevar){
```

```
 console.log(morevar);
}
```

Функция clearTimeout сбрасывает параметры заданные функцией setTimeout.

Для периодического запуска какой-либо функции идеально подходит setInterval. Синтаксис вызова похож на вызов функции setTimeout. Только функция обратного вызова будет вызываться столько раз, сколько задано вторым параметром. Сбросить заданный интервал можно вызовом функции clearInterval.

Особенность работы таймерных функций заключается в том, что пока есть активный таймер, node.js не может завершить процесс.

Для любой таймерной функции мы можем вызывать метод **unref**(), который делает таймерную функцию второстепенной, т.е. node.js ее не учитывает при проверке внутренних процессов.

Рассмотрим листинг с двумя таймерными функциями: setTimeout и setInterval. Без метода .unref() функция setInterval() будет работать постоянно, несмотря на то, что в функции setTimeout вызывается закрытие сервера через 3 сек.

Использование метода unref для таймерных функций. Листинг 2.10

```
var http = require('http');
var server = new http.Server(function(req, res){
}).listen(3000);
setTimeout(function(){
 server.close();
}, 3000);
var timer = setInterval(function(){
  console.log(process.memoryUsage());
}, 1000);
timer.unref();
```

Функции обратного вызова

Одной из особенностей асинхронной работы node.js является то, что в методах используются функции обратного вызова. Т.е. сам по себе метод ничего не возвращает, он отдает ответ функции обратного вызова.

Функция обратного вызова. Листинг 2.11

```
 fs.readFile('index.html', function(err, info){

 })
```

Функция обратного вызова всегда содержит два входящих параметра. Если ошибок нет, первый параметр – null, второй – ответ. Если ошибки есть, то функция будет вызывана только с первым аргументом, который содержит информацию об ошибках.

Работа с файлами

Для работы с файлами, node.js имеет встроенный модуль fs.

Чтение файла:

Асинхронный вызов функции readFile. Листинг 2.12

```
var fs = require('fs');
fs.readFile(__filename, function(err, data){
 if(err){
  console.log(err);
 } else {
  console.log(data);
 }
});
```

Где __filename – это имя текущего файла. При запуске получим не содержимое файла, а специальный объект буфер.

```
C:\OpenServer\domains\Node\test_node>node test.js
<Buffer 76 61 72 20 66 73 20 3d 20 72 65 71 75 69 72 65 28 27 66 73 27 29 3b 0d
0a 66 73 2e 72 65 61 64 46 69 6c 65 28 5f 5f 66 69 6c 65 6e 61 6d 65 2c 20 66 75
 ...>
```

Чтобы преобразовать буфер в строку, можно воспользоваться методом toString():

Перевод буфера в строку методом toString(). Листинг 2.13

```
var fs = require('fs');
fs.readFile(__filename, function(err, data){
 if(err){
  console.log(err);
 } else {
  console.log(data.toString('utf-8'));
 }
});
```

Кодировку utf-8 можно не указывать, т.к. она используется как кодировка по умолчанию.

```
C:\OpenServer\domains\Node\test_node>node test.js
var fs = require('fs');
fs.readFile(__filename, function(err, data){
    if(err){
        console.log(err);
    } else {
        console.log(data.toString());
    }
});
```

Рассмотрим еще один вариант преобразования в строку: использование кодировки при открытии потока.

Перевод буфера в строку с помощью параметра encoding. Листинг 2.12

```
var fs = require('fs');
fs.readFile(__filename,{encoding: 'utf-8'},function(err, data){
 if(err){
  console.log(err);
 } else {
  console.log(data);
 }
});
```

В этом случае преобразование в строку происходит внутри функции.

Чтение файла построчно:

Чтение файла построчно, метод ReadStream. Листинг 2.13

```
var fs = require('fs');
var stream = new fs.ReadStream(__filename, {encoding: 'utf-8'});
stream.on('readable', function(){
 console.log(data);
});
stream.on('end', function(){
 console.log('Конец файла');
});
```

Событие readable срабатывает при каждом проходе по строке файла. Событие end – при успешном завершении чтения.

TCP-клиент и TCP-сервер

Протокол TCP (Transmission Control Protocol – протокол управления передачей) является базовым для многих интернет-приложений.

Для создания TCP-сервера и TCP-клиента, имеется встроенный модуль Net.

Мы можем создать сервер, передавая функцию обратного вызова с единственным аргументом функции – экземпляром сокета, прослушивающего два события: получение данных и закрытие соединения клиентом. Создадим в

отдельном файле (например, server.js) сервер с помощью следующего листинга.

Создание TCP-сервера. Листинг 2.14

```
var net = require('net');
var server = net.createServer(function(conn){
  console.log('connected');
  conn.on('data', function(data){
      console.log( data+ ' от ' + conn.remoteAddress + ' ' +
conn.remotePort);
      conn.write(data + ' - никому.');
  });
  conn.on('close', function(){
      console.log('client closed connection');
  })
}).listen(8125);
```

Посредством метода on назначаются два прослушивателя событий. Первым параметром метод принимает имя события, вторым – функцию прослушиватель.

Создание TCP-клиента. Для этого создадим отдельный файл (например client.js), и в нем напишем следующее:

Создание TCP-клиента. Листинг 2.15

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');
client.connect(8125, 'localhost', function(){
  console.log('connected to Server');
  client.write('Кому нужен браузер?');
});
process.stdin.resume(); // подготовка к вводу данных с консоли
process.stdin.on('data', function(){
    client.write(data);
});
client.on('data', function(data){
    console.log(data);
});
client.on('close', function(){
    console.log('connection is closed');
})
```

Итак, у нас готово два файла, один из который является клиентом, второй – сервером.

Клиентское приложение отправляет только что набранную строку, которую сервер выводит в консоль и отвечает клиенту дублируя эту строку и добавляя свою.

Чтобы протестировать эти node-приложения, запустим две консоли. В первой запустим приложение сервера:

Запуск сервера с помощью REPL. Листинг 2.16

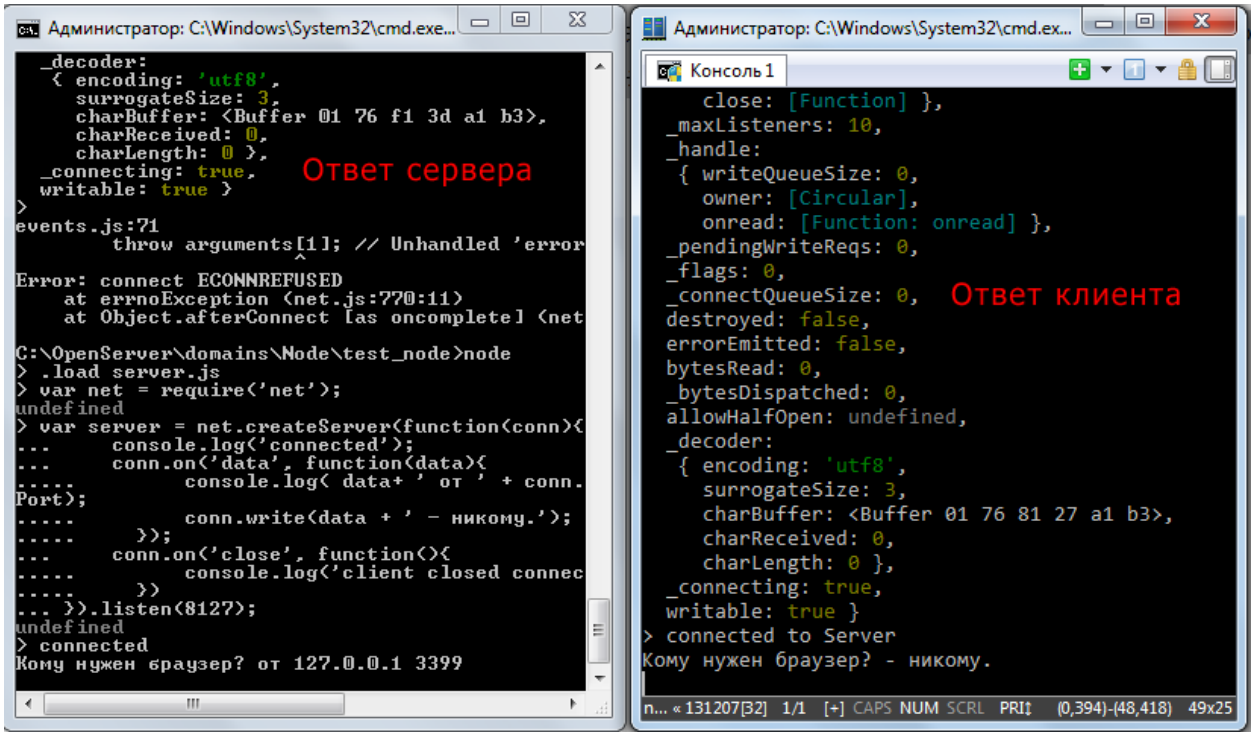
```
.load server.js
```

После чего запустим клиента.

Запуск клиента с помощью REPL. Листинг 2.17

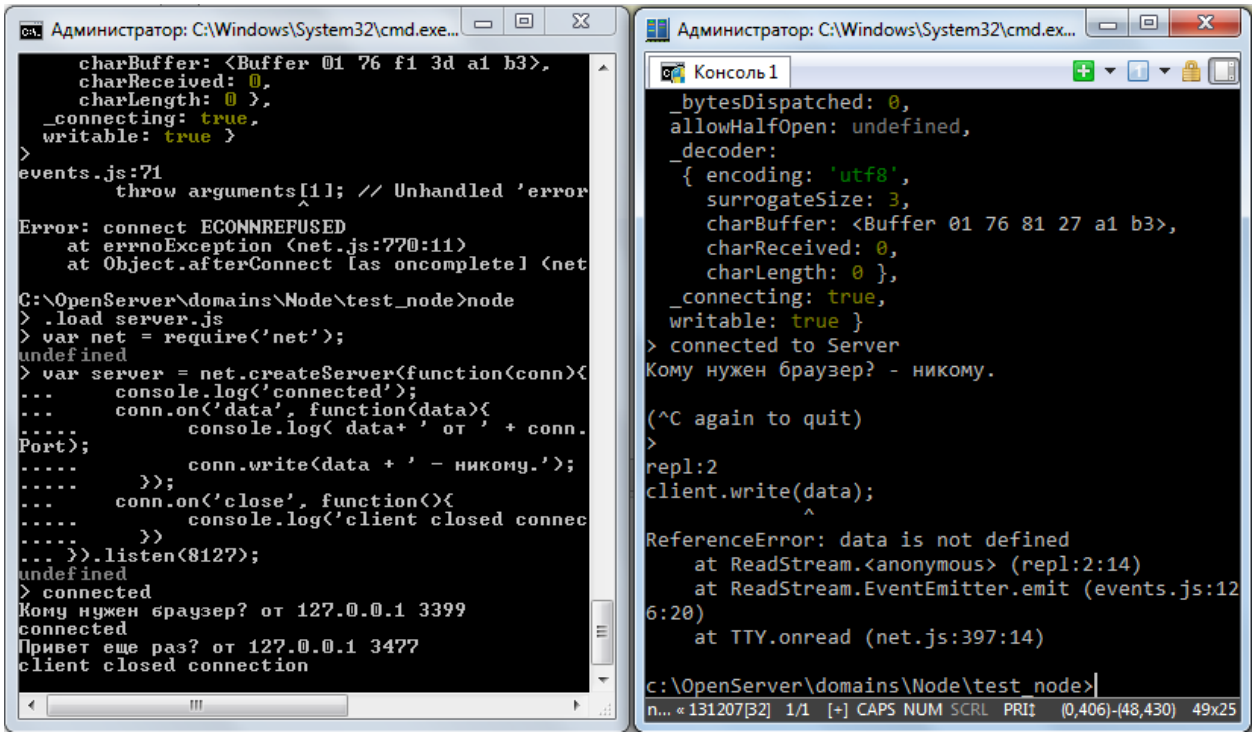
```
.load client.js
```

После запуска сервера и клиента получим следующие ответы:



Мы использовали файлы, которые загружали в консоль. Но мы могли бы наладить общение между сервером и клиентом и без дополнительных файлов, с помощью одного режима REPL, используя формат многострочного ввода.

Соединение между клиентом и сервером поддерживается до тех пор, пока не будет прервано с одной из сторон. В режиме REPL – это комбинация клавиш Ctrl+C. Сведения об этом выводятся на консоль. Например, при закрытии клиента увидим следующее:



HTTP-сервер

Можно сказать, что HTTP-протокол (HyperText Transfer Protocol – протокол передачи гипертекста) является частным случаем протокола TCP. Так и в ядре Node.js мудуль для создания протокола HTTP (который так и называется http), наследует функциональность модуля Net (модуль протокола TCP).

Модуль HTTP представляет базовую HTTP-функциональность, обеспечивающую приложению сетевой доступ к запросам и ответам. Рассмотрим пример создания HTTP-сервера:

Создание HTTP-сервера. Листинг 2.18

```
var http = require('http');
http.createServer(function(req, res){
 res.writeHead(200, {'content-type': 'text/plain'});
 res.end('Hello world!');
}).listen(8128);
console.log('Server running on 8128');
```

Набираем в браузере http://127.0.0.1:8128 и увидим на экране Hello world! Следует обратить внимание на важную деталь: если мы запустим еще один процесс, то консоль выдаст ошибку. Система не может слушать один и тот же пор дважды.

Для того, чтобы еще раз запустить прослушивание того же порта, необходимо закрыть предыдущее прослушивание.

С помощью функции createServer и безымянной функции обратного вызова создается новый сервер. Входящие параметры функции обратного вызова: req (серверный запрос или поток чтения – это объект http.serverRequest) и res (серверный ответ или поток записи – это объект http.serverResponse).

У объекта http.serverResponse имеются следующие методы:

- **res.writeHead()**, который отправляет заголовок ответа с кодом статуса ответа.
- **res.end()**, который подает сигнал о завершении переадчи данных и тело ответа для вывода на экран.
- **res.write()**, который выводит данные на экран без сигнала о завершении переадчи данных.

Метод http.Server.listen прослушивает входящие подключения к заданному порту. Метод listen является асинхронным, т.е. не блокирует выполнение программы в ожидании подключения. Поэтому, функция console.log() листинга может выполниться раньше подключения.

Кроме потока чтения и записи, HTTP поддерживает кодировку фрагментированной передачи. Этот тип кодировки применяется для обработки больших объемов данных. При этом запись данных может начаться еще до получения оставшейся части запрошенных данных.

Модули Net и HTTP могут также подключаться к UNIX-сокету, а не к конкретному сетевому порту, что позволяет поддерживать взаимодействиен между процессами в пределах одной и той же системы.

Сокеты UDP

Протокол TCP требует взаимодействия между двумя конечными точками выделенного соединения. UDP-протокол (User Datagram Protocol) этого не требует. Что означет отсутствие гарантии взаимодействия между двумя конечными точками. UDP-протокол является менее надежным, зато более быстрым в сравнении с TCP.

Для создания UDP-сокета, необходимо воспользоваться методом createSocket, передав ему тип сокета (udp4 и udp6). В отличии от TCP-сообщений, сообщения UDP должны передаваться в виде буферов, а не строк.

UDP-клиент. Листинг 2.19

```
var dgram = require('dgram');
var client = dgram.createSocket('udp4');
process.stdin.resume();
process.stdin.on('data', function(data){
 console.log(data.toString('utf8'));
 client.send(data, 0, data.length, 8124,
             'examples.burningbird.net',
  function(err, bytes){
      if(err)
         console.log('error: ' + err);
      else
      console.log('successful');
  });
})
```

Далее создадим UDP-сервер, задача которого подключиться к нужному порту и прослушивать событие message. Хотя привязывать сервер к порту не обязательно, но без привязки к порту, сокет пытался бы прослушивать каждый порт.

UDP-сервер. Листинг 2.20

```
var dgram = require('dgram');
var server = dgram.createSocket('udp4');
server.on('message', function(msg, rinfo){
 console.log('message: ' + msg + ' от ' + rinfo.address)
});
server.bind(8124)
```

Использование потоков для работы с сетевыми соединениями

Откроем файл для чтения и выведем его на экран.

Сперва рассмотрим пример решения этой задачи без потоков:

Вывод файла с помощью функции обратного вызова. Листинг 2.21

```
var http = require('http');
var fs = require('fs');
http.createServer(function(req, res){
        fs.readFile('test.html', function(err, content){
            if(err){
                res.statusCode = 500;
                res.end('Server error');
            } else {
                res.setHeader('Content-Type', 'text/html; char-
set=utf-8')
                res.end(content);
            }
        });
}).listen(8134);
```

Метод readFile асинхронно считывает файл и возвращает ответ в функцию обратного вызова, в параметр content. Такое решение приемлимо, но есть проблема, которая заключается в том, что если считываемый файл большой, это может привести к зависанию программы. Получится, что сервер может занять всю доступную память.

Рассмотрим универсальный алгоритм отправки данных из одного потока в другой.

Все потоки чтения (например из файла) имеют встроенный метод pipe, который работает так:

Объект_чтения.pipe(объект_записи)

Помимо экономии памяти, преимущество такого подхода заключается в том, что можно вызывать несколько объектов записи для одного потока чтения.

Вывод файла с помощью .pipe. Листинг 2.22

```
var http = require('http');
var fs = require('fs');
http.createServer(function(req, res){
 var file = new fs.ReadStream('test.html');
 file.pipe(res);
 file.pipe(process.stdout);
}).listen(8134);
```

В данном листинге есть одна проблемма: если происходит разрыв связи с клиентом, серверу это не известно, и он продолжит держать файл в потоке чтения, а соответственно, будут выделяться дополнительные процессы.

Чтобы этого не происходило, добавим дополнительные обработчики событий. Если соединение оборвется, закроем файл и оборвем все связанные с этим соединением ресурсы:

Отлавливаем момент, когда соединение закрыто. Листинг 2.23

```
var http = require('http');
var fs = require('fs');
http.createServer(function(req, res){
 var file = new fs.ReadStream('test.html');
    sendFile(file,res);
}).listen(8134);
function sendFile(file, res){
    file.pipe(res);
    file.on('error', function(err){
       res.end('Ошибка сервера');
       console.error(err);
    })
        .on('open', function(){
```

```
            console.log('open');
        })
        .on('close', function(){
          console.log('close');
        });
    res.on('close', function(){
        file.destroy();
    });
}
```

Дочерние процессы.

Node позволяет запустить системную команду в рамках нового дочернего процесса и прослушивать его ввод-вывод. Дочерние процессы, в которых активизируются системные Unix-команды не работают в Windows и наоборот.

Модуль **child-process**, входящий в Node.js, позволяет работать с дочерними процессами: порождать их, передавать и получать информацию в асинхронном режиме, управлять работой потока.

Для создания дочерних процессов, можно воспользоваться четырмя различными технологиями, но чаще всего пользуются методом spawn. Он запускает команду в новом процессе, передавая ей необходимо количество параметров.

Например, мы можем выполнить консольную команду dir, которая выведет содержимое текущего каталога:

Дочерние процессы в Windows. Листинг 2.24

```
var cmd = require('child_process').spawn('cmd', ['/c',
'dir\n']);
cmd.stdout.on('data', function(data){
    console.log('stdout: ' + data);
});
cmd.on('exit', function(code){
    console.log('child' + code);
})
```

Модуль DNS

DNS (Domain Name System – система доменных имен). Используется в приложениях, в которых требуется находить домены или IP-адреса.

Для нахождения IP-адреса заданного домена можно воспользоваться методом .lookup.

Нахождение IP-адреса по заданному домену. Листинг 2.25

```
var dns = require('dns');
dns.lookup('obmenka.by', function(err, ip){
```

```
    if(err) throw err;
    console.log(ip);
});
```

Метод .resolve возвращает массив доменных имен по заданному IP-адресу.

Вывод массива доменов по заданному IP-адресу. Листинг 2.26

```
var dns = require('dns');
dns.reverse('178.159.242.96', function(err, domains){
    if(err) throw err;
    domains.forEach(function(dom){
        console.log(dom);
    });
});
```

Модуль URL

Данный модуль обеспечивает синтаксический разбор URL-адреса.

Разбор URL-адреса. Листинг 2.27

```
var url = require('url');
var urlob = url.parse('http://localhost:8080/?file=main');
console.log(urlob);
```

Получим следующее:

```
{ protocol: 'http:',
  slashes: true,
  host: 'localhost:8080',
  port: '8080',
  hostname: 'localhost',
  href: 'http://localhost:8080/?file=main',
  search: '?file=main',
  query: 'file=main',
  pathname: '/',
  path: '/?file=main' }
```

Модуль Util

Подключается модуль так:

Подключение модуля util. Листинг 2.28

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

Рассмотрим следующие полезные методы данного модуля:

util.**inspect()** – Позволяет красиво вывести любой объект. Поведение метода напоминает поведение суперметода toString().

Объект console для вывода использует именно этот метод. Если мы хотим вывести результат в консоль, то можно воспользоваться знакомым console.log. Но если необходимо вывести в базу данных, либо в файл, тогда придется обращаться к методу inspect.

util.**format**() – Данный метод получает строку и значения для вставки.

Метод .format(). Листинг 2.29

```
var str = util.format('My %s %d% %f', 'str', 123, {ob: 'obj'})
```

Вместо %s выведется строка ‘string’, вместо %d – число 123, вместо %f - объект формата JSON.

util.**inherits()** – Данный метод считается самым восстребованным методом модуля util. Он принимает два параметра: имя конструктора-родителя и имя конструктора-потомка, в результате чего конструктор-потомок наследует всю функциональность главного конструктора.

Использование .inherits. Листинг 2.30

```
var util = require('util');
function Animal(name){
    $this.name = name;
}
Animal.prototype.walk = function(){
    console.log('Ходит ' + this.name);
}
function Rabbit(name){
    this.name = name;
}
util.inherits(Rabbit, Animal);

Rabbit.prototype.jump = function(){
    console.log('Прыгает ' + this.name);
}
// использование
var rabbit = new Rabbit('кролик ');
rabbit.walk(); //метод родителя
rabbit.jump(); //метод потомка
```

Получится, что все методы создаваемые конструктором будут наследоваться от Animal.

События и объект EventEmiter

Это основной объект, реализующий работу с событиями и обеспечивает асинхронную обработку событий Node.js.

Остальные объекты, которые генерируют события (например, через метод on()) его наследуют.

Использование .inherits. Листинг 2.31

```
// аргументы передаются по цепочке
// обработчики срабатывают в том же порядке, в котором назначены

var EventEmitter = require('events').EventEmitter;

var server = new EventEmitter;

server.on('request', function(request) {
    request.approved = true;
});

server.on('request', function(request) {
    console.log(request);
});

server.emit('request', {from: "Клиент"});

server.emit('request', {from: "Ещё клиент"});
```

Второй основной метод – метод emit(). Он генерирует события, и передает данные. Эти данные попадают в функцию-обработчик. Метод emit() должен использоваться совместно с методом on().

Метод listeners возвращает массив всех прослушивателей данного события.

Использование .listeners. Листинг 2.32

```
server.on('connection', function (stream) {
  console.log('Кто-то подключился!');
});
console.log(util.inspect(server.listeners('connection')); // [
[Function] ]
```

Supervisor

Это модуль отслеживающий изменения в дирректориях проекта node.js, и как-только данный модуль находит какие-то изменения в файлах, перезапускает node.js.

Данный модуль необходимо ставить глабально.

Установка модуля supervisor. Листинг 2.33

```
npm install -g supervisor
```

После установки модуля, для запуска Node, вместо команды node можно воспользоваться командой *surervisor*.

Например, чтобы запустить файл server.js из консоли, необходимо набрать следующее:

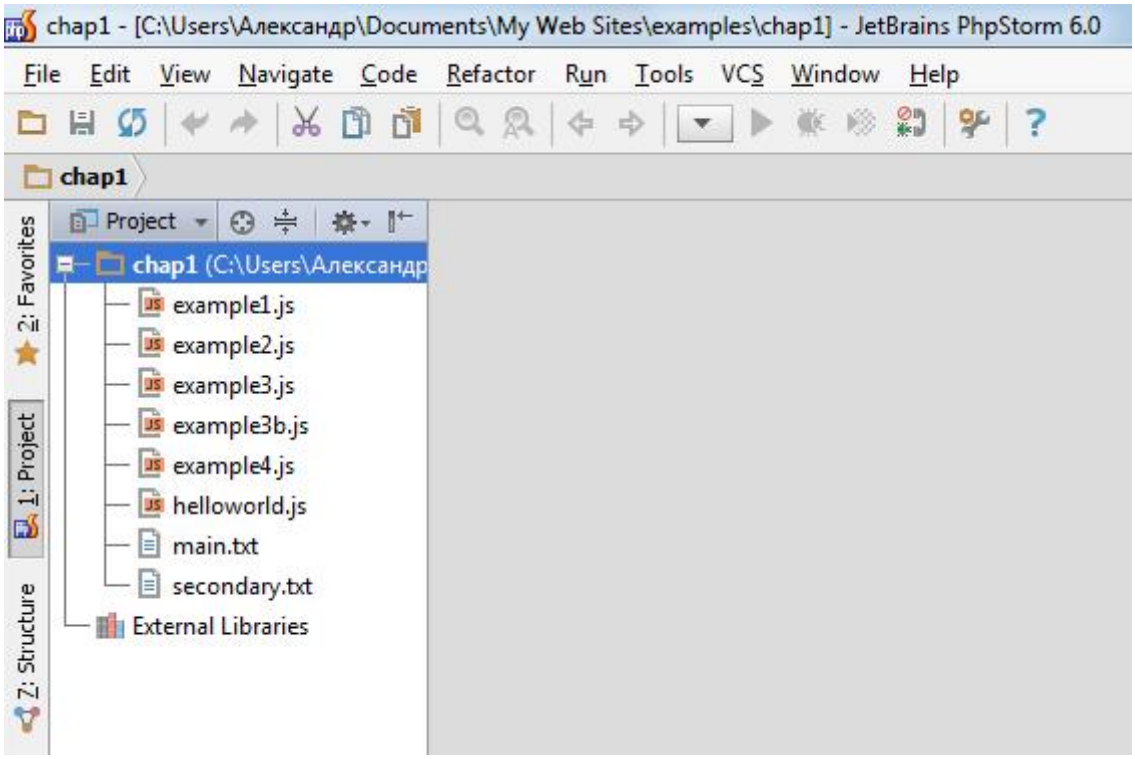
Запуск файла через supervisor. Листинг 2.34

```
supervisor server.js
```

3. Node.js и PHPStorm

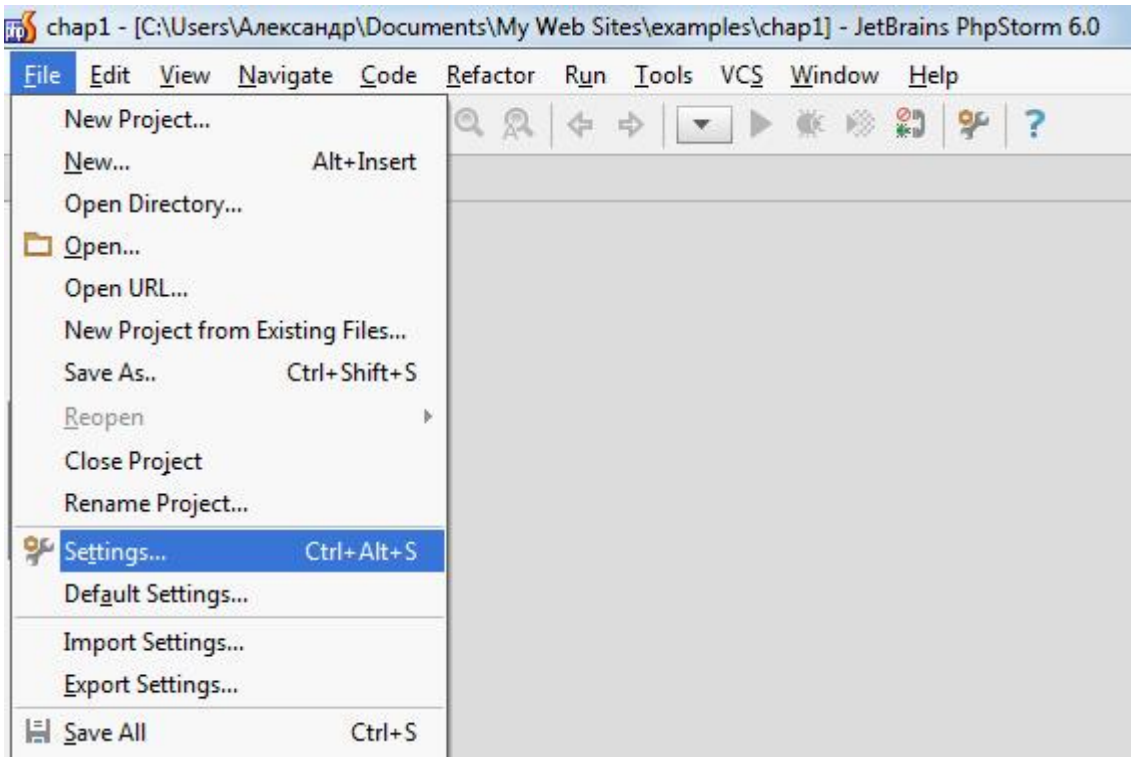
Папку с node.js файлом можно просто перенести в ярлык PHPStorm.

Тогда PHPStorm автоматически из содержимого папки делает проект.

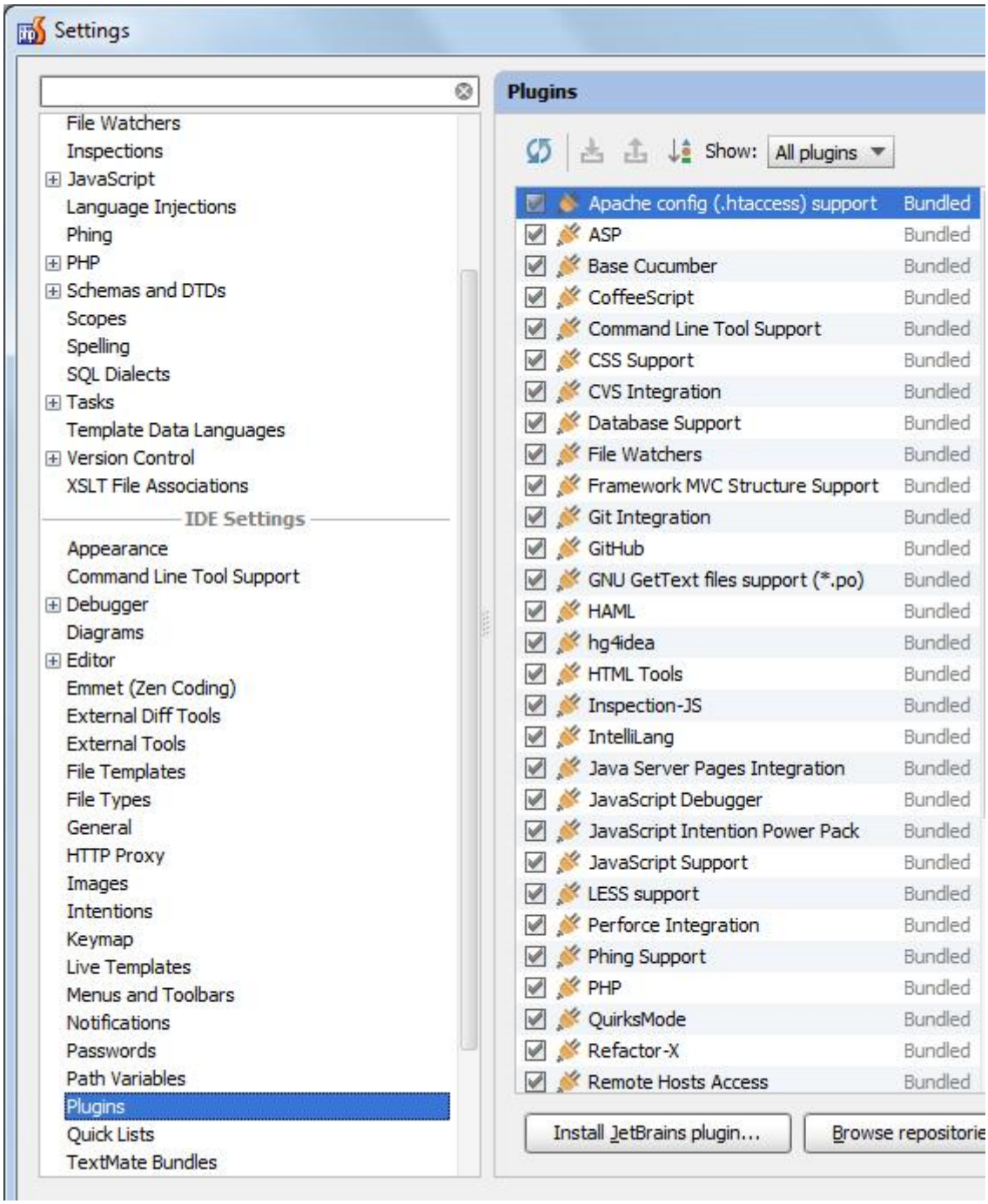


Далее для работы с Node.js нам потребуется дополнительный плагин.

Откроем File – Settings



Выбираем plagins



Нажимаем кнопку Instal JetBrains plugin…

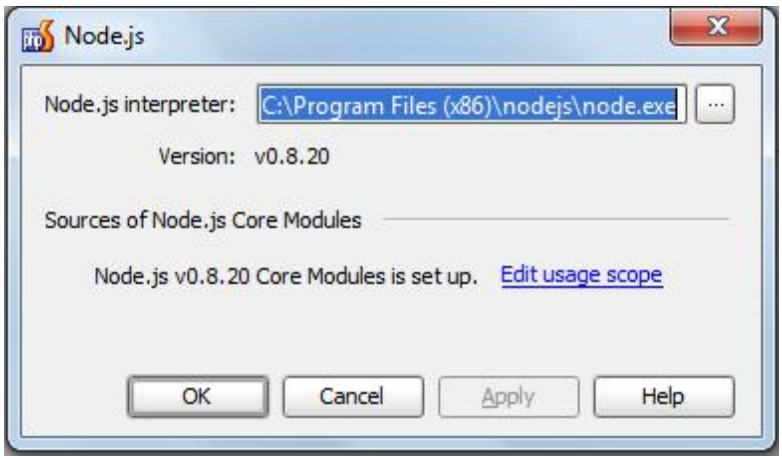
Выбираем плагин Node.js. Двойной щелчек по плагину.

Установка завершена. Закрываем окно с плагинами. Нажимаем Apply.

Перезапускаем PHPStorm.

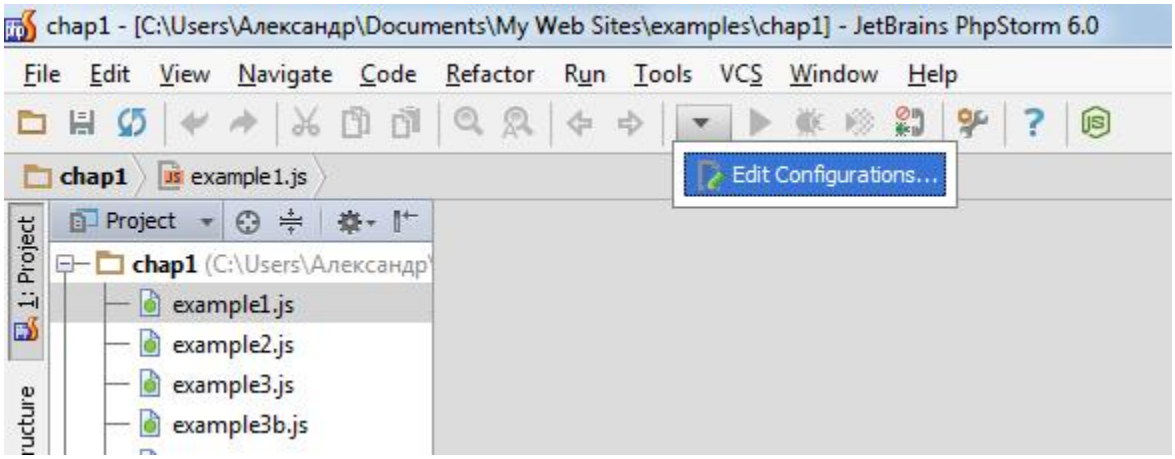
Теперь у нас появилась кнопочка Node.js, которая превращает любой javaScript проект в Node.js.

Нажав на эту кнопку мы должны увидеть следующее:

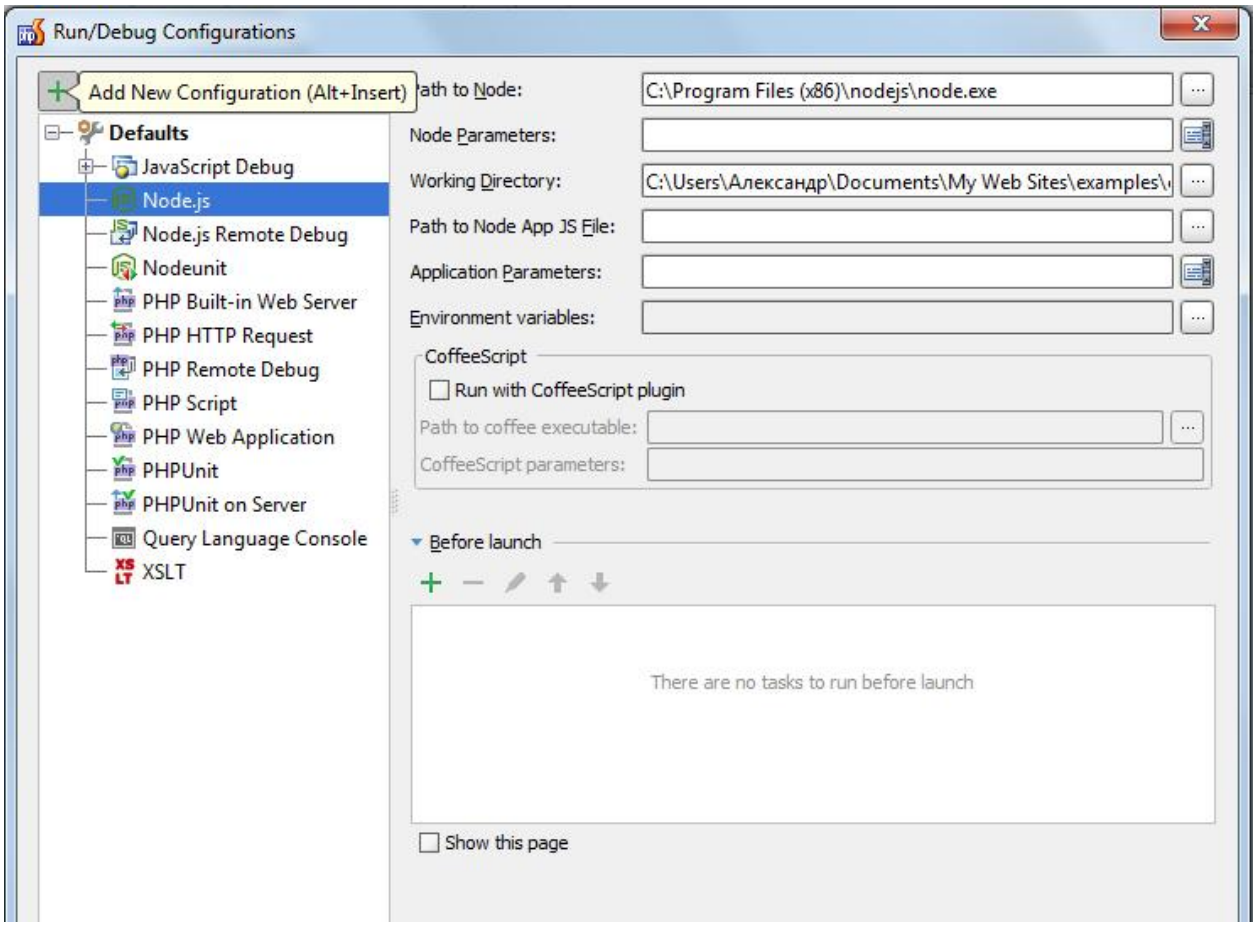


В противном случае, нужно будет указать путь к Node.js

Для запуска файла Node.js нам необходимо сконфигурировать новый профиль.

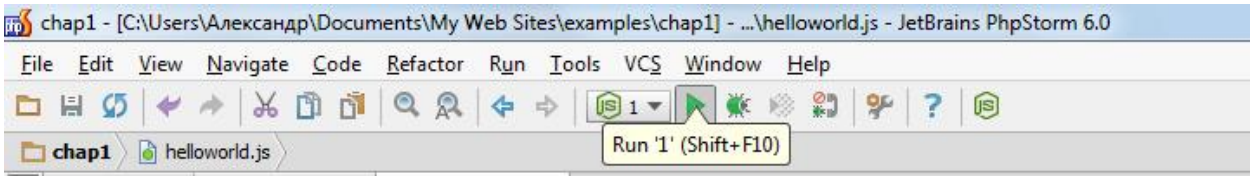


Делаем его из Node.js. Для этого необходимо выбрать Node.js и нажать +.



В поле Path to Node App JS File выбираем входящий исполняемый файл.

Теперь становится активной кнопка Run.

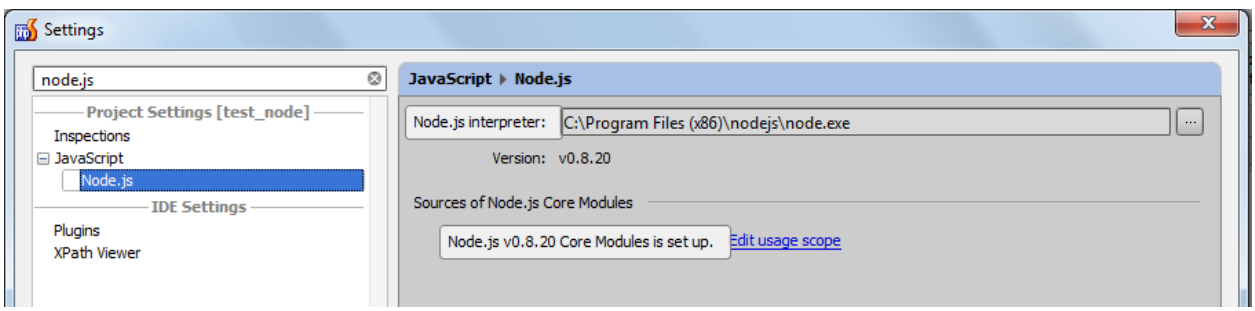


Настройка синтаксиса node.js

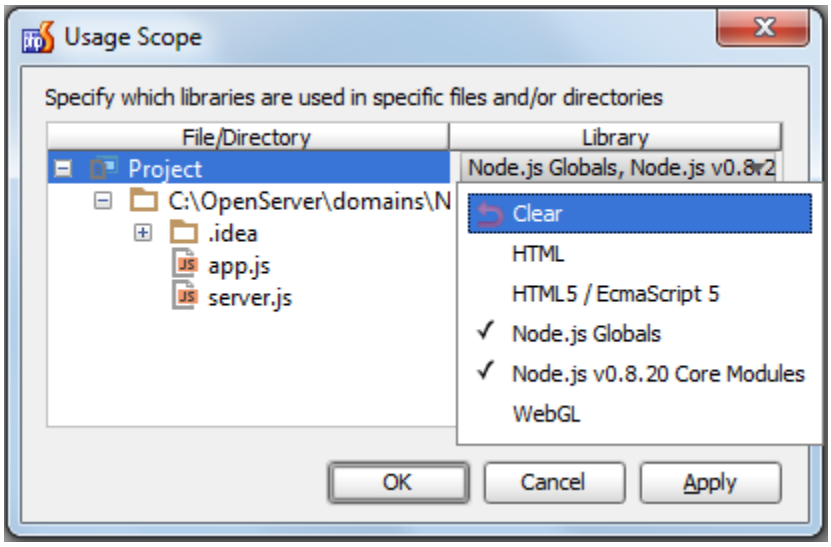
Открываем File->settings.

В открывшемся окне вводим node.js

Далее нажимаем Edit usage scope.



В открвшемся окне отмечаем необходимые галочки:



На этом настройка PHPStorm для работы с Node.js закончена.

4. Отладка

Простой отладчик

Для режима отладки в Node.js есть встроенный отладчик: debug.

Запуск скрипта в режмие отладки. Листинг 4.1

```
node debug server.js
```

Теперь код node.js будет выполнятся поэтапно.

```
C:\OpenServer\domains\Node\test_node>node server.js
^C
C:\OpenServer\domains\Node\test_node>node debug server.js
< debugger listening on port 5858
connecting... ok
break in C:\OpenServer\domains\Node\test_node\server.js:1
  1 var net = require('net');
  2 debugger;
  3 var server = net.createServer(function(conn){
debug>
```

После выполнения нескольких строк кода, программа замерла в ожидании дополнительных комманд. Если набрать help, можно ознакомиться со списком доступных команд:

```
debug> help
Commands: run (r), cont (c), next (n), step (s), out (o), backtrace (bt), setBre
akpoint (sb), clearBreakpoint (cb),
watch, unwatch, watchers, repl, restart, kill, list, scripts, breakOnException,
breakpoints, version
debug>
```

Дальнейшее выполнение скрипта вызывается командой cont, или c. Мы также можем вмешиваться через консоль в программный код, вызывая любые команды Node.js

Добавим в код node.js команду **debugger**;

Debugger. Листинг 4.2

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

var server = net.createServer(function(conn){
    console.log('connected');
    debugger;
    conn.on('data', function(data){
        console.log( data+ ' от ' + conn.remoteAddress + ' ' +
conn.remotePort);
        conn.write(data + ' - никому.');
    });
    conn.on('close', function(){
        console.log('client closed connection');
    })
}).listen(8127);
```

Дойдя до команды degugger браузер остонавливается, в ожидании дополнительных команд.

Отладка браузером

Для настройки отладки браузером Chrome, понадобится модуль node-inspector, который поставим глобально.

Установка node-inspector. Листинг 4.3

```
npm install -g node-inspector
```

Запустим скрипт Node.js со специальным параметром –debugg

```
C:\OpenServer\domains\Node\test_node>node --debug test.js
debugger listening on port 5858
Server running on 8128
```

Node.js не только запускает скрипт, но и начинает слушать порт 5858. Т.е. к Node.js может подключиться другая программа и давать команды, например остоновить или возобновить выполнение, получить значение переменной и т.д.

В консоли выполним команду node-inspector:

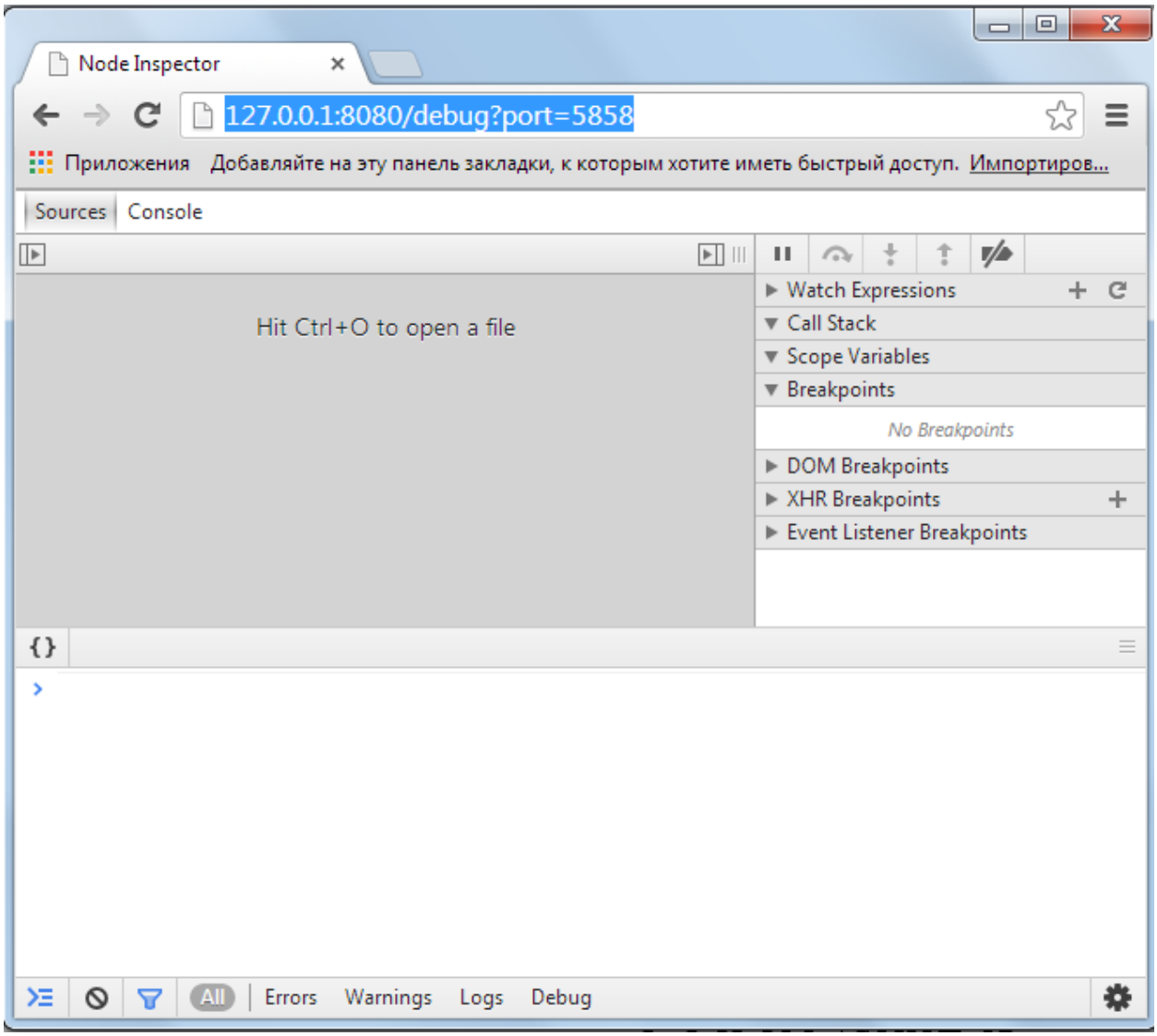
```
C:\OpenServer\domains\Node\test_node>node-inspector
Node Inspector v0.7.0
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

Мы получили приглошение перейти по ссыкле:

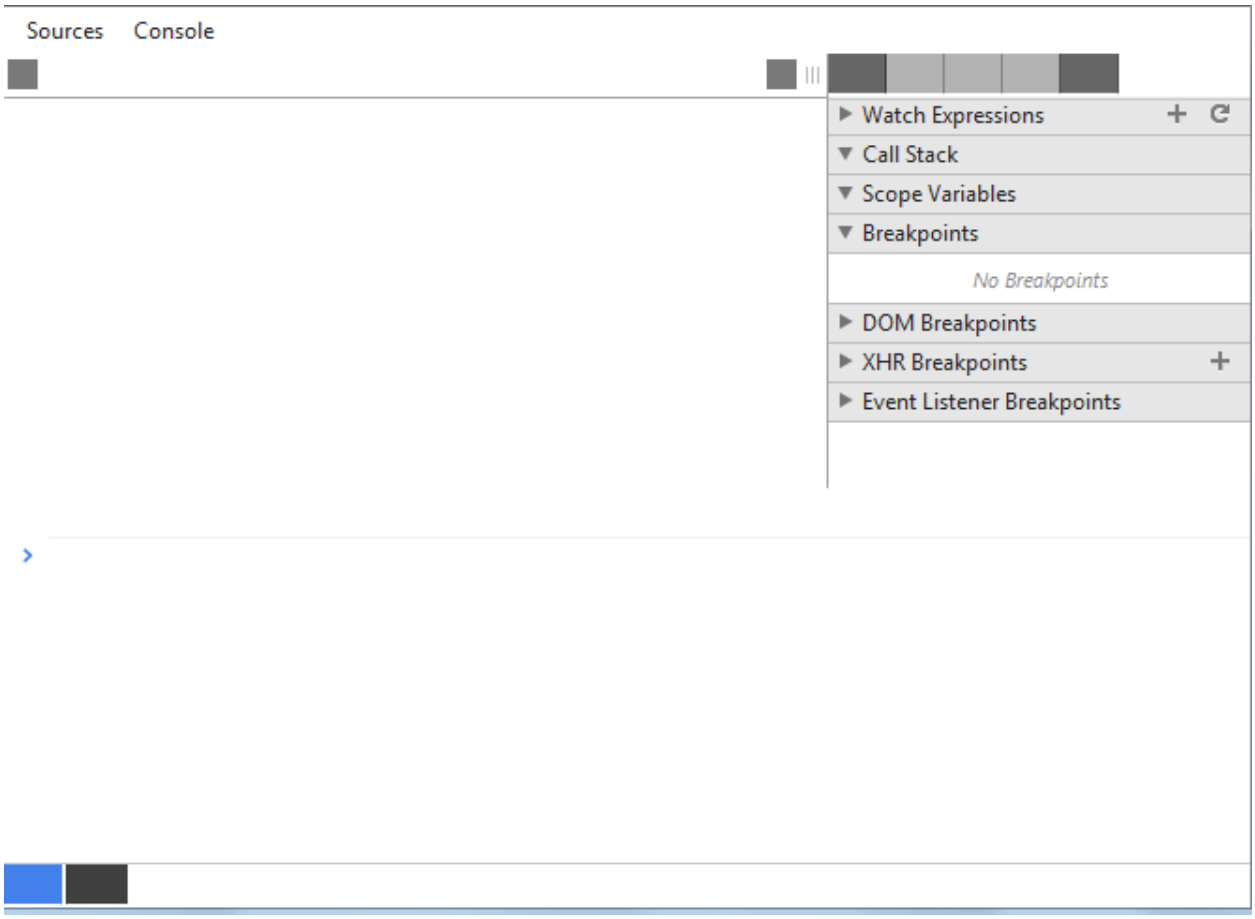
http://127.0.0.1:8080/debug?port=5858

Откроем данную страницу в браузере.

Chrome:



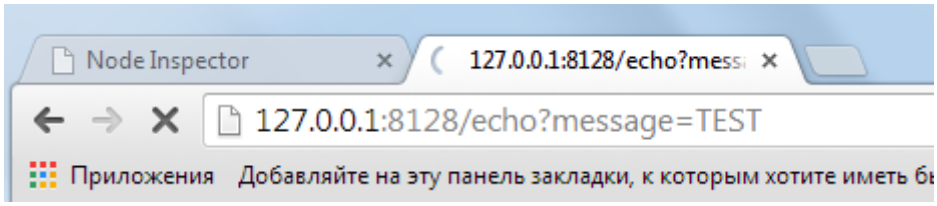
Firefox:



Далее запустим скрипт в браузере и передадим в консоль команду:

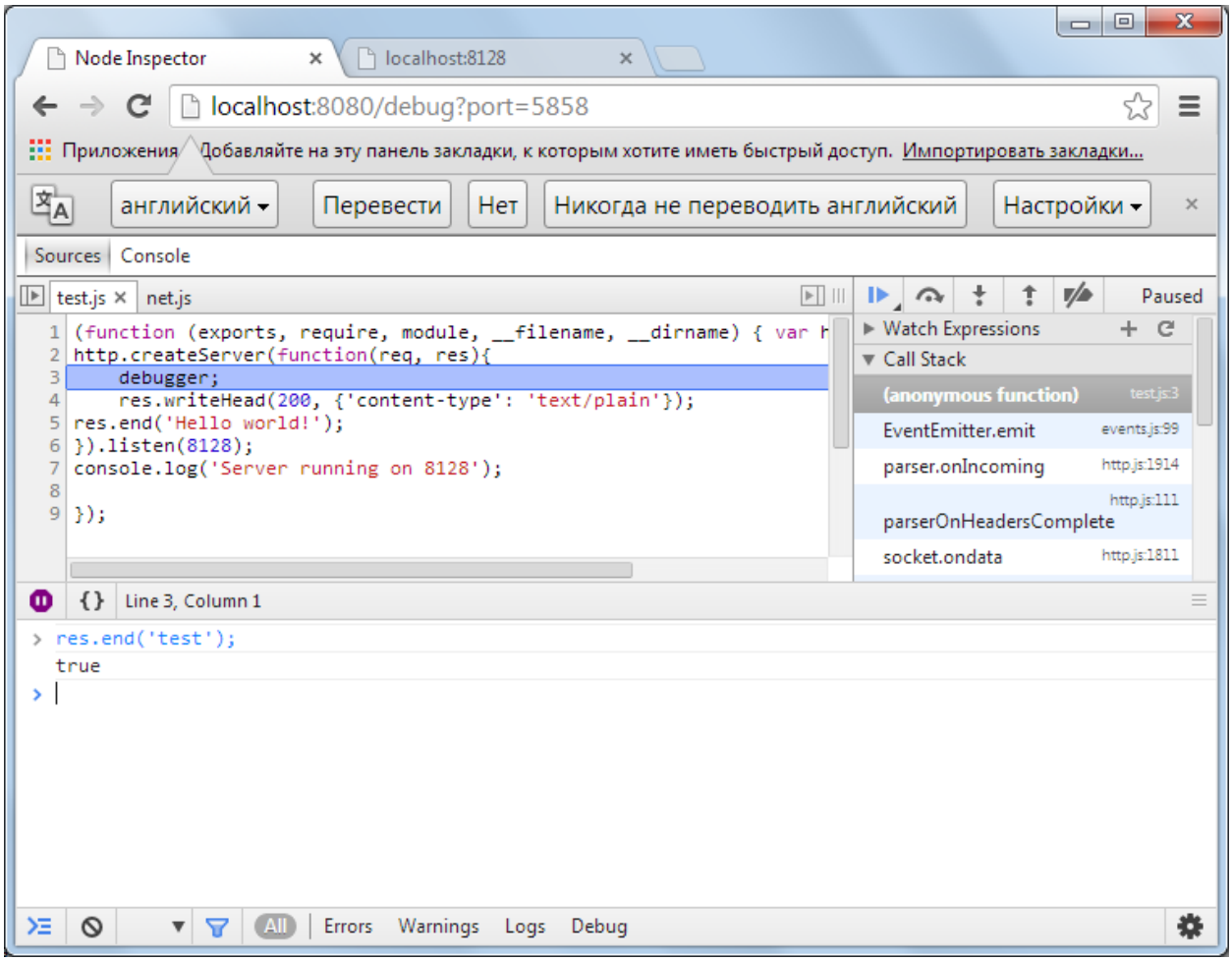
http://127.0.0.1:8128/echo?message=TEST

Если в скрипте имеется команда debugger, то скрипт останавливается на этой команде



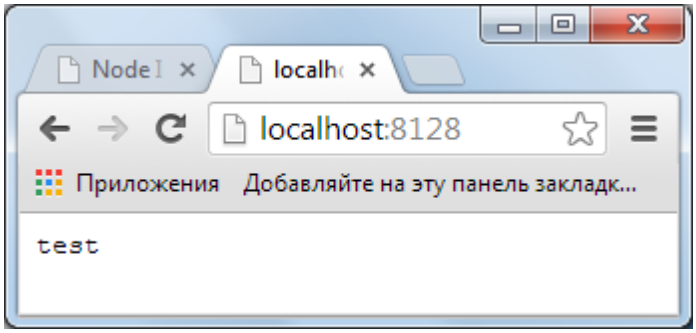
Обратите внимание на зависший браузер. Браузер завис, потому что наткнулся на команду debugger.

Если мы перейдем в Node Inspector, то увидим следующий ответ консоли:

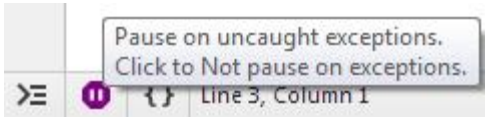


В инспекторе имеется консоль, где мы можем выполнять node-команды

Результат отображается в браузере по запросу localhost:8128.



Фиалетовый значек в нижнем углу экрана node-инспектора указывает на то, что инспектор будет останавливаться на ошибках.



Запуск пошаговой отладки в состоянии паузы

Команда --debug-brk запускает отладчик по порту 5858, ждет подключения к порту и дальнейших команд с этого порта.

Команда –debug-brk. Листинг 4.4

```
node --debug-brk test.js
```

```
C:\OpenServer\domains\Node\test_node>node --debug-brk test.js
debugger listening on port 5858
```

Далее, в новом окне консоли необходимо запустить node-инспектор:

node-inspector

Который показывает где произошла остановка:

test.js

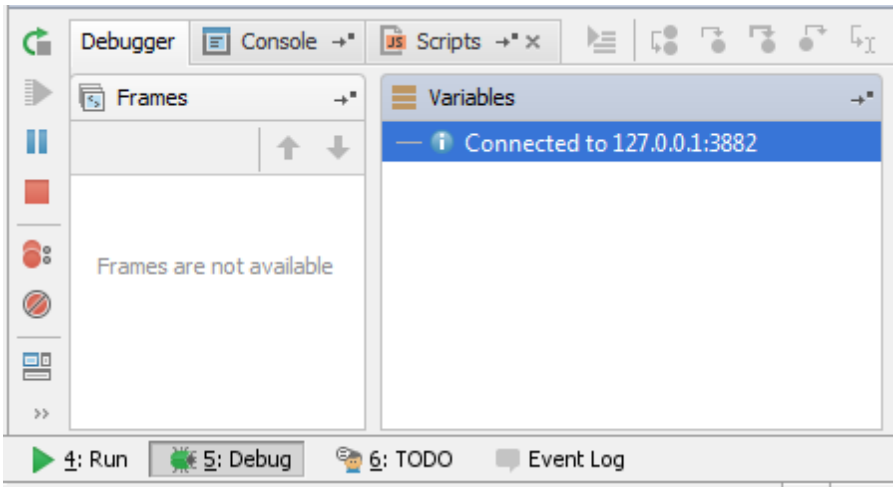
```
1  (function(exports, require, module, __filename, __dirname) {
2      var http = require('http');
3      http.createServer(function(req, res) {
4          debugger;
5          res.writeHead(200, {'content-type': 'text/plain'});
6          res.end('Hello world!');
7      }).listen(8128);
8      console.log('Server running on 8128');
9
0  });
1
```

Чтобы перейти к следующей выполняемой команде, необходимо в правом блоке управления, нажать кнопку  .

Если при этом программа выскакивает за пределы текущего файла, то нажатие соседней кнопки  вернет обратно в текущий исполняемый файл.

Отладка под IDE PHPShtorm

 - такой значек запускает node-приложение в режиме отладки.



5. Обработка ошибок Node.js

Наследование от встроенного объекта ошибки Error

Рассмотрим пример, когда нужно использовать наследование от встроенного объекта ошибок. Предположим имеется два объекта. Один – проверяет на корректность вводимое значение, второй – корректность URL.

Наследование от ошибок Error. Листинг 5.1

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

var phrases = {
    "Hello": "Привет",
    "world": "мир"
};

function getPhrase(name) {
    if (!phrases[name]) {
        throw new Error("Нет такой фразы: " + name);
    }
    return phrases[name];
}

function makePage(url) {
    if (url != 'index.html') {
        throw new Error("Нет такой страницы");
    }
    return util.format("%s, %s!", getPhrase("Hello"),
getPhrase("world"));
}

var page = makePage('index.html');
console.log(page);
```

В данном листинге не возможно понять где какая ошибка. Оба метода возвращают ошибки класса Error. Можно сделать свои обработчики ошибок для разных случаев.

Наследование от ошибок Error. Листинг 5.2

```
var util = require('util');
var phrases = {
    "Hello": "Привет",
    "world": "мир"
};

// message name stack
function PhraseError(message) {
    this.message = message;
    Error.captureStackTrace(this, PhraseError); // вывод только текущего стека ошибок (ошибок данного метода)
}
util.inherits(PhraseError, Error);
PhraseError.prototype.name = 'PhraseError';

function HttpError(status, message) {
    this.status = status;
    this.message = message;
    Error.captureStackTrace(this, HttpError);
}
util.inherits(HttpError, Error);
HttpError.prototype.name = 'HttpError';

function getPhrase(name) {
    if (!phrases[name]) {
        throw new PhraseError("Нет такой фразы: " + name); // HTTP 500, уведомление!
    }
    return phrases[name];
}

function makePage(url) {
    if (url != 'index.html') {
        throw new HttpError(404, "Нет такой страницы"); // HTTP 404
    }
    return util.format("%s, %s!", getPhrase("Hello"), getPhrase("world"));
}

try {
    var page = makePage('index.html');
    console.log(page);
} catch (e) {
    if (e instanceof HttpError) {
        console.log(e.status, e.message);
    } else {
        console.error("Ошибка %s\n сообщение: %s\n стек: %s", e.name, e.message, e.stack);
    }
}
```

6. Внешние модули

Модули устанавливаются из приложения npm, которое устанавливается по умолчанию, вместе с node.js.

Для просмотра npm-команд можно воспользоваться следующей командой:

npm help npm

Установка модулей из консоли:

npm install modulename – установка последней версии модуля.

Или:

npm i modulname, т.е. вместо ключевого слова install можно использовать букву i

npm install modulename@1.1.1 – установка конкретной версии модуля.

*npm install modulename@1.** - установка любой ветки первой версии модуля.

npm install https://github.com/name/modulename/master - установка модуля через github.com

npm install /path/modulename.tgz – установка модуля по пути.

После установки модулей, в проекте появится папка node_modules, которая содержит все установленные модули. Если нужно произвести глобальную установку модулей, необходимо добавить ключевое слово –global или –g перед командой install:

npm –g install modulename – глобальная установка модуля, или

npm i –g modulname – сокращенный вариант глобальной установки.

Обновление модулей

npm update – обновление всех модулей.

npm update modulename – обновление конкретного модуля.

npm outdate – проверить наличие устаревших пакетов (эту команду можно использовать также для каждого модуля в отдельности).

Удаление модулей

npm uninstall modulename

Список модулей

npm list – получить список модулей.

npm ls – список модулей с зависимостями

npm ls –g – получить список глобально установленных модулей.

Удаление модулей

npm remove modulename – удаление текущего модуля

7. Express

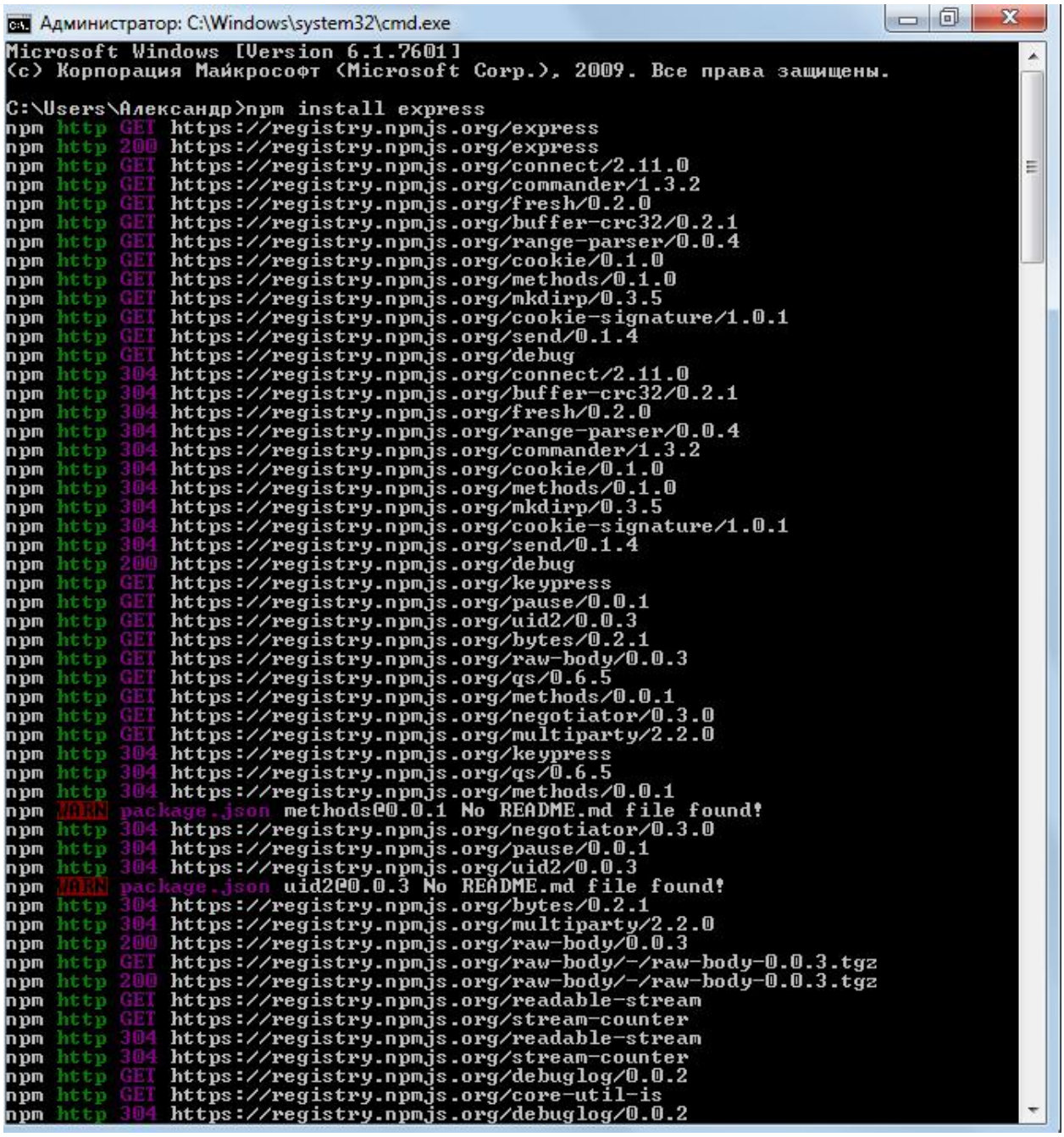
Express – это Node.js-фрэймворк.

Для установки модуля Express в консоли командной строки необходимо перейти в папку с проектом, и набрать следующий код.

Глобальная установка модуля Express. Листинг 6.1

```
npm install -g express
```

Об успешной установки модуля свидетельствует следующее сообщение консоли:



Далее с помощью команды *express* мы можем создавать приложения на Node. Это команда позволяет генерировать костяк сайта.

Опции Express. Листинг 7.1

```
express --help
```

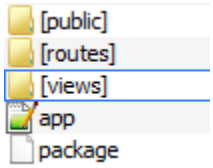
```
C:\OpenServer\domains\Node\chat>express --help

  Usage: express [options] [dir]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -s, --sessions      add session support
    -e, --ejs           add ejs engine support (defaults to jade)
    -J, --jshtml        add jshtml engine support (defaults to jade)
    -H, --hogan         add hogan.js engine support
    -c, --css <engine>  add stylesheet <engine> support (less|stylus) (defaults
to plain css)
    -f, --force         force on non-empty directory
```

Создание проекта с опциями Express:

Создание проекта с поддержкой сессий и шаблонизатора ejs. Листинг 7.2

```
express -s -e
```

В рабочей папке Express создаст такую структуру:



Рассмотрим файл package.json

Файл package.json. Листинг 7.3

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.4.4",
    "ejs": "*"
  }
}
```

В параметр *name* можно ввести имя приложения. Параметр *scripts* подсказывает, как данное приложение можно запустить.

Параметр *dependencies* указывает на зависимости проекта. Данные зависимости – это модули, которые можно установить либо по одному, либо все сразу, с помощью следующей команды:

Установка всех зависимостей. Листинг 7.4

```
npm i
```

После чего появится новая дирректория node_modules.

Файл запуска – app.js.

Чтобы разобраться, как работает Express, очистим файл app.js. Введем туда следующий код:

Проверка модуля Express. Листинг 7.5

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
    res.send('Hello World');
});

app.listen(3000);
```

Перезапустим наше приложение. И через localhost проверим 3000-ый порт:

http://localhost:3000

Express-приложение использует метод app.get для назначения функции прослушивания запроса. С помощью данного метода можно создавать маршруты.

Маршрут / (прямой слэш) обозначает корневой адрес. Express преобразует все маршруты в объект регулярного выражения.

Создадим еще один маршрут.

Создание дополнительного маршрута. Листинг 7.6

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
    res.send('Hello World');
});

app.get('/express', function(req, res){
    res.send('Hello express');
});

app.listen(3000);
```

Сейчас в браузере кроме запроса http://127.0.0.1:3000 будет доступен еще один запрос http://127.0.0.1:3000/express

Кроме строки ответа, res.send() может посылать буфер, JSON, статус ответа сервера:

- *res.send(new Buffer('whoop'));*
- *res.send({some: 'json'});*
- *res.send('some html');*

- *res.send(404, 'Page not found');*
- *res.send(500, {error: 'something blew up'});*
- *res.send(200);*

8.Модуль nconf

Данный модуль предназначен для конфигурирования приложения.

Установка модуля nconf. Листинг 8.1

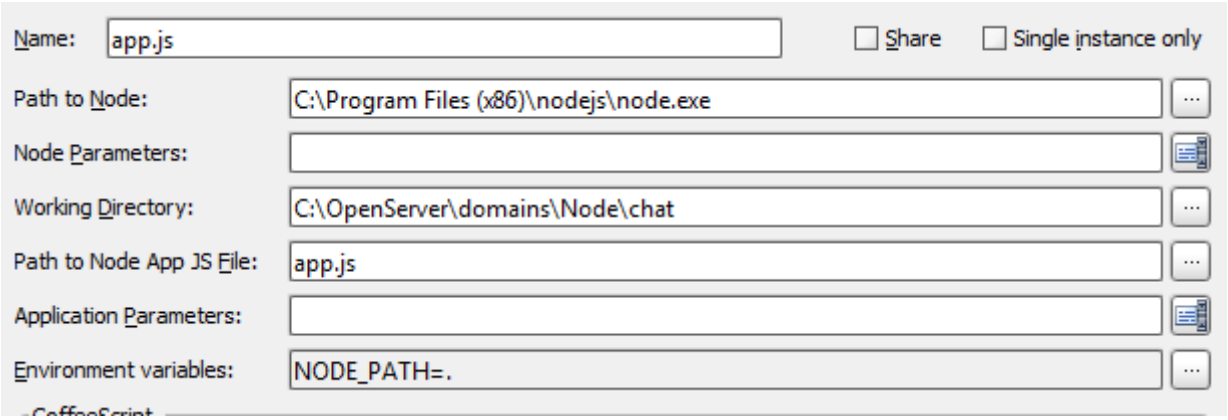
```
npm i nconf
```

Далее к приложению мы можем подключать конфигурационный файл JSON.

Подключение конфигурационного файла JSON. Листинг 8.2

```
nconf = require('nconf');
nconf.argv()
     .env()
     .file({ file: 'path/to/config.json' });
```

Настройки приложения PHPShtorm



9. Хранилище данных MongoDB

От реляционных систем баз данных, например MySLQ MongoDB отличается тем, что в MongoDB данные хранятся ввиде документов, а не ввиде таблиц. Документы кодируются в формате BSON, который является двоичной формой JSON.

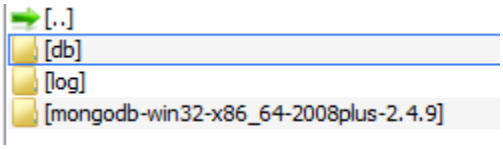
Вместо таблицы, мы получаем *коллекцию*, а вместо строки таблицы - *BSON-документ*.

Скачать MongoDB можно по адресу:

http://www.mongodb.org/downloads/

Далее нужно разархивировать mongo в любую рабочую дирректорию, например: c:/mongo

В этой же дирректории создадим папки db для файлов хранилища и log для log-файлов.



Перейдем в установочную папку, в моем случае, это mongodb-win32-x86_64-2008plus-2.4.9. Далее в папку bin. Откроем здесь консоль. В консоли выполним команду, которая настроит путь к папке db:

Укажем путь к папке db. Листинг 9.1

```
mongod --dbpath "C:\mongo\db"
```

В новой консоли запустим mongo.

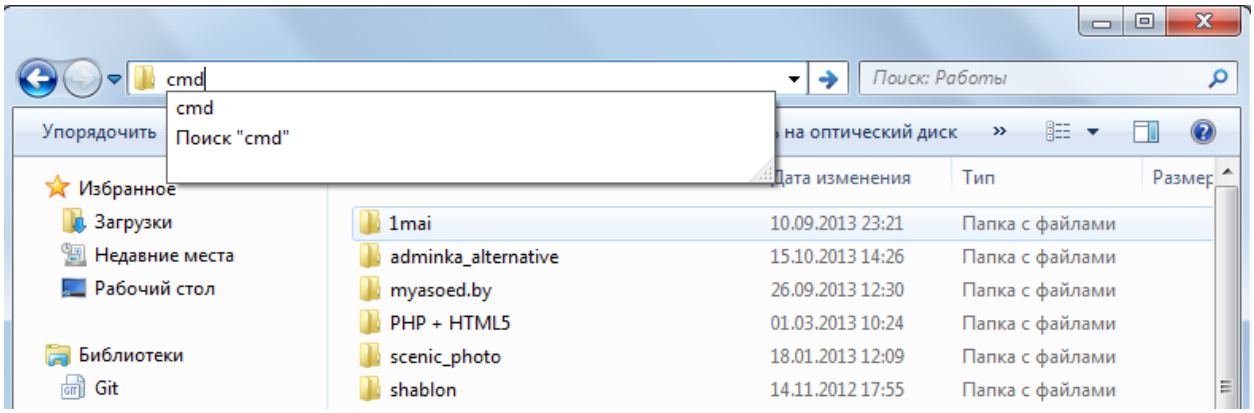
В Node.js имеется несколько модулей, ориентированных на работу с MongoDB:

- *MongoDB Native Node.js driver*
- *Mongoose*

10. Создание приложения на Express

Создать шаблонное приложение довольно просто.

Через проводник заходим в нужный каталог, в котором хотим создать приложение. Запускаем консоль командной строки.



В открывшейся командной строке набираем:

Создание шаблонного приложения site. Листинг 9.1

```
Express site
```

Приложение создает каталог site со следующими подкаталогами: public, routes, views. И файл app.js, который, который в свою очередь создает сервер. Рассмотрим его.

App.js. Листинг 9.2

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.json());
app.use(express.urlencoded());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' +
app.get('port'));
});
```

Рассмотрим файл построчно.

var express = require('express'); - подключение модуля expres

var routes = require('./routes'); - подключение дирректории routes. Из данной дирррректории подключается файл index.js

В файле index.js находится следующий код:

Файл index.js подкаталога routes. Листинг 9.3

```
exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

Метод render, относящийся ко входящему в Express объекту ответа, выводит заданный шаблон (index.jade) с набором параметров.

var user = require('./routes/user'); - подключение файла user.js из дирректории routes. Если не указано расширение, то подключается расширение .js

var http = require('http'); - подключение модуля http. Данный модуль, в отличии от express – встроенный, и не требует отдельной инсталяции.

var path = require('path'); - подключение модуля path.

После включения всех необходимых модулей и подкаталогов, создается экземпляр объекта Express:

var app = express();

Затем он конфигурируется с помощью набора параметров.

app.set('port', process.env.PORT || 3000); - указание на то, какой порт прослушивать.

app.set('views', path.join(__dirname, 'views')); - подключение папки шаблонов. По умолчанию подключается шаблон index.jade

app.set('view engine', 'jade'); - определение движка для шаблона (в данном случае Jade).

Далее в app.js вызывается Express со следующими связующими функциями: favicon, logger и static. Эти три функции интегрированны из модуля connect.

app.use(express.favicon()); - подключение файла favicon.ico

app.use(express.logger('dev')); - статутс log-файлов. Dev – разработка.

app.use(express.json()); - формирование ответа в формате Json

app.use(express.urlencoded()); - связующая функция для формирования ответов.

app.use(express.methodOverride()); - применяется для использования методов delete и put в формах. При подключении данной функции становится

возможно использование методов app.delete и app.put, вместе со стандартными app.get и app.post.

Пример html-формы с элеменом с методом put. Листинг 9.4

```
<form> ...
  <input type="hidden" name="_method" value="put" />
</form>
```

При подключенном methodOverride() для такой формы можно использовать метод app.put.

Использование app.put. Листинг 9.5

```
app.put('/users/:id', function (req, res, next) {
  // edit your user here
});
```

app.use(app.router); - используется для установки маршрутов приложения

app.use(express.static(path.join(__dirname, 'public'))); - путь к статичным файлам стилей, скриптам и изображениям

if ('development' == app.get('env'))… - настройка окружающей среды для процесса разработки.

Метод app.get предназначен для настройки маршрутов.

app.get('/', routes.index); - подключение файла index из каталога routes. Напомню, что ранее переменная routes была определена, которая содержит подключенную дирректорию routes. Таким образом, если приложение вызывается без параметров (например 127.0.0.1:3000), то срабатывает данный роут. Роут включает index.js, который в свою очередь включает шаблон index.jade

app.get('/users', user.list); - подкллючение файла list.js по маршруту /users

Хранение данных в памяти

В приложении мы собираемся сохранять в памяти элементы с идентификатором товара, названия, цены и описанием.

Создадим массив widgets в файле app.js

Создание первого элемента массива widgets в файле app.js. Листинг 9.6

```
var widgets = [
    { id : 1,
        name : 'My Special Widget',
        price : 100.00,
        descr : 'A widget beyond price'
    }
```

```
]
```

Создадим роут, который будет выводить информацию о данном элементе либо выводить сообщение, если элемент с индексом не найден.

Создание первого элемента массива widgets в файле app.js. Листинг 9.7

```
app.get('/widgets/:id', function(req, res) {
    var indx = parseInt(req.params.id) - 1;
    if (!widgets[indx])
        res.send('There is no widget with id of ' + req.params.id);
    else
        res.send(widgets[indx]);
});
```

Для добавления виджетов в папке public создадим файл forma.html, содержащий html-форму:

forma.html. Листинг 9.8

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8" />
 <title>Widgets</title>
</head>
<body>
<form method="POST" action="/widgets/add"
enctype="application/x-www-form-urlencoded">

 <p>Название: <input type="text" name="widgetname" id="widgetname"
size="25" required/></p>

 <p>Цена: <input type="text" name="widgetprice" id="widgetprice"
size="25" required/></p>

 <p>Описание: <br /><textarea name="widgetdesc" id="widgetdesc"
cols="20" rows="5">No Description</textarea>
 <p>

 <input type="submit" name="submit" id="submit" value="Submit"/>
 <input type="reset" name="reset" id="reset" value="Reset"/>
 </p>
 </form>
</body>
```

В форме требуется ввести имя товара, цену и описание.

Далее в файл app.js добавим роут для обработчика данной формы.

Роут для добавления виджета. Листинг 9.9

```
app.post('/widgets/add', function(req, res) {
  var indx = widgets.length + 1;
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
```

```
    price : parseFloat(req.body.widgetprice),
    descr : req.body.widgetdesc };
  console.log('added ' + widgets[indx-1]);
  res.send('Widget ' + req.body.widgetname + ' added with id ' +
indx);
});
```

Т.к. форма использует метод POST, то и роут настроен на перехват данных отправляемых методом post.

Приложению еще требуется обеспечить поддержку двух RESTful-команд: PUT (для обновления виджета) и DELETE (для его удаления).

Форма для обновления виджета показана в следующем листинге. В ней всего лишь одно отличие от формы для добавления нового виджета: появилось скрытое поле с именем _method.

HTML-форма для редактирования виджета public/edit.html. Листинг 9.10

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8" />
 <title>Widgets</title>
</head>
<body>
<form method="POST" action="/widgets/2/update"
enctype="application/x-www-form-urlencoded">

 <p>Название: <input type="text" name="widgetname" id="widgetname"
size="25" required/></p>

 <p>Цена: <input type="text" name="widgetprice" id="widgetprice"
size="25" required/></p>

 <p>Описание: <br /><textarea name="widgetdesc" id="widgetdesc"
cols="20" rows="5">No Description</textarea>
 <p>
 <input type="hidden" value="put" name="_method" />
 <input type="submit" name="submit" id="submit" value="Submit"/>
 <input type="reset" name="reset" id="reset" value="Reset"/>
 </p>
 </form>
</body>
```

Поскольку команды PUT и DELETE не поддерживаются атрибутом method формы, нам приходится добавлять их используя скрытое поле.

Форма удаления удаления виджета:

HTML-форма для удаления виджета public/del.html. Листинг 9.11

```
<p>Вы уверены, что хотите удалить виджет?</p>
<form method="POST" action="/widgets/1/delete"
 enctype="application/x-www-form-urlencoded">
```

```
 <input type="hidden" value="delete" name="_method" />
 <input type="submit" name="submit" id="submit" value="Delete Widg-
et 1"/>
</form>
```

Далее нужно добавить функциональность обработки этих двух новых команд.

Запрос на обновление виджета. Листинг 9.12

```
app.put('/widgets/:id/update', function(req,res) {
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     descr : req.body.widgetdesc };
  console.log(widgets[indx]);
  res.send ('Updated ' + req.params.id);
});
```

Данный запрос на обновление заменяет содержимое объекта виджета новым контентом.

Рассмотрим запрос на удаление виджета, который удаляет запись в массиве, оставляя значение null.

Запрос на удаление виджета. Листинг 9.13

```
app.put('/widgets/:id/update', function(req,res) {
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     descr : req.body.widgetdesc };
  console.log(widgets[indx]);
  res.send ('Updated ' + req.params.id);
});
```

Создадим еще один запрос, на вывод всех элементов массива виджета.

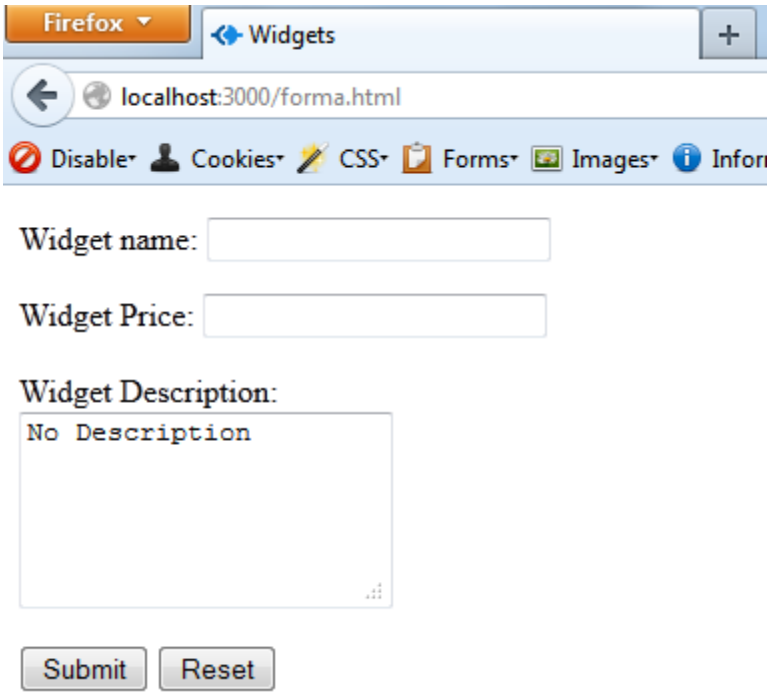
Запрос на вывод всех элементов массива виджета. Листинг 9.14

```
app.get('/widgets', function(req, res){
   res.send(widgets);
});
```

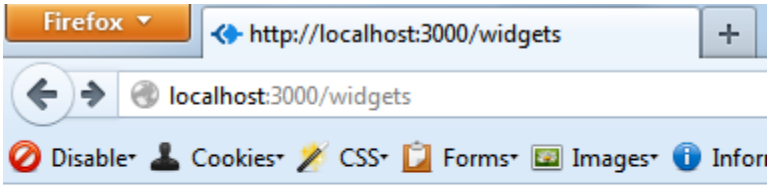
Приступаем к тестированию приложения.

1. Запускаем приложение, нажимая большую зеленую кнопку RUN

2. Открываем в браузере форму добавления виджета:



3. Заполняем форму. Таким образом, мы добавили новый элемент виджета, с индексом 2.
4. Вызываем widgets и убеждаемся в существовании нового элемента виджета с индексом 2:




```
[
  {
    "id": 1,
    "name": "My Special Widget",
    "price": 100,
    "descr": "A widget beyond price"
  },
  {
    "id": 2,
    "name": "Книга \"Изучаем Node.js\"",
    "price": 212,
    "descr": "Классная книга по Node.js"
  }
]
```


5. Теперь можем вызвать форму редактирования виджета. edit.html. Ввести новые данные для виджета 2. После чего снова перейти на страницу widgets, и увидим, что данные изменились.
6. Аналогичным образом, можно проверить работу команды на удаление виджета.

10. MVC для приложения на Node.js

Приведем наше приложение к архитектуре MVC.

Большая чать функциональности уже готова, нам осталось лишь приветси всё в порядок.

Начнем с преобразования вызова методов в подходящий для MVC формат. В основе перехода лежит понятие CRUD.

CRUD — (англ. create read update delete — «Создание чтение обновление удаление») сокращённое именование 4 базовых функций при работе с персистентными хранилищами данных — **создание, чтение, редактирование и удаление**.

Операция	SQL-оператор	операция в HTTP
Создание (Create)	INSERT	POST
Чтение (Read)	SELECT	GET
Редактирование (Update)	UPDATE	PUT или PATCH
Удаление (Delete)	DELETE	DELETE

Сначала создадим каталог controllers. В нем новый файл widgets.js. Перенесем все вызовы методов из файла app.js в созданный файл.

После этого необходимо преобразовать вызовы методов.

Например, рассмотрим функцию создания нового виджета.

Функция создания нового виджета. Листинг 10.1

```
app.post('/widgets/add', function(req, res) {
    var indx = widgets.length + 1;
    widgets[widgets.length] =
    { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice),
        descr : req.body.widgetdesc };
    console.log('added ' + widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' +
indx);
});
```

Эта функция превращается в функцию export.create

Функция widgets.create. Листинг 10.2

```
export.create = function(req, res) {
    var indx = widgets.length + 1;
```

```
    widgets[widgets.length] =
    { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice),
        descr : req.body.widgetdesc };
    console.log('added ' + widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' +
indx);
});
```

Новая функция по прежнему получает объект запроса и ответ ресурса. Единственное отличие состоит в том, что маршрут на функцию больше не отображается.

Controller

Рассмотрим преобразованный контроллер виджетов. Два из имеющихся в нем методов, new и edit пока оставим пустыми

Контроллер widgets.js. Листинг 10.3

```
 var widgets = [
    { id : 1,
        name : "The Great Widget",
        price : 1000.00
    }
]

// Вывод массива widgets
exports.index = function(req, res) {
    res.send(widgets);
};

// Отображение формы добавления виджета
exports.new = function(req, res) {
    res.send('displaying new widget form');
};

// Добавление виджета
exports.create = function(req, res) {
    var indx = widgets.length + 1;
    widgets[widgets.length] =
    { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice) };
    console.log(widgets[indx-1]);
    res.send('Widget ' + req.body.widgetname + ' added with id ' +
indx);
};

// Просмотр конкретного элемента виджета
exports.show = function(req, res) {
    var indx = parseInt(req.params.id) - 1;
    if (!widgets[indx])
        res.send('There is no widget with id of ' + req.params.id);
    else
```

```
        res.send(widgets[indx]);
};

// Удаление виджета
exports.destroy = function(req,res) {
    var indx = req.params.id - 1;
    delete widgets[indx];

    console.log('deleted ' + req.params.id);
    res.send('deleted ' + req.params.id);
};

// Отображение формы редактирования виджета
exports.edit = function(req, res) {
    res.send('displaying edit form');
};


// Обновление/редактирование виджета
exports.update = function(req,res) {
    var indx = parseInt(req.params.id) - 1;
    widgets[indx] =
    { id : indx,
        name : req.body.widgetname,
        price : parseFloat(req.body.widgetprice)}
    console.log(widgets[indx]);
    res.send ('Updated ' + req.params.id);
};
```

Чтобы связать маршруты с новыми функциями, создадим новый модуль maproutcontroller с одной экспортируемой функцией mapRoute. У неё есть два параметра: это express-объект app и prefix, предоставляющий собой отображенный объект контроллера (в данном случае - widgets).

Связующая функция маршрутов с методами объекта контроллера. Листинг 10.4

```
exports.mapRoute = function(app, prefix) {
   prefix = '/' + prefix;
   var prefixObj = require('./controllers/' + prefix);

   // index
   app.get(prefix, prefixObj.index);

   // add
   app.get(prefix + '/new', prefixObj.new);

   // show
   app.get(prefix + '/:id', prefixObj.show);

   // create
   app.post(prefix + '/create', prefixObj.create);

   // edit
   app.get(prefix + '/:id/edit', prefixObj.edit);

   // update
   app.put(prefix + '/:id/update', prefixObj.update);
```

```
    // destroy
    app.del(prefix + '/:id/destroy', prefixObj.destroy);

};
```

Подключим файл с функцией к исполняемому файлу app.js

Подключение функции mapRoute. Листинг 10.5

```
var map = require('./maproutecontroller');
```

Далее в файле app.js создадим массив prefexes, пока с одним элементом - widgets. И через функцию mapRoute свяжем все маршруты с контроллерами.

Вызов функции mapRoute. Листинг 10.6

```
var prefixes = ['widgets'];

// map route to controller
prefixes.forEach(function(prefix) {
    map.mapRoute(app, prefix);
});
```

View

Express поддерживает две системы шаблонов. Jade и Ejs.

Устанавливаем систему шаблонов EJS (Embedded JavaScript – внедряемый JavaScript код) с помощью Node пакетов npm.

Установка модуля ejs. Листинг 10.7

```
npm install ejs
```

Рассмотрим пример ejs-кода.

Ejs-шаблон. Листинг 10.8

```
<% if(names.length){%>
 <ul>
  <% names.forEach(function(name)){%>
    <%= name%>
  <% }%>
 </ul>
<% }%>
```

В данном случае ejs-инструкция с помощью ограничителей <% %> внедряется непосредственно в html-код.

Подключается движок шаблонов ejs в конфигурационном файле app.js так:

Подключение системы шаблонов ejs. Листинг 10.9

```
var express = require('express')
  , routes = require('./routes')
  , http = require('http');

var app = express();

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
});

app.configure('development', function(){
  app.use(express.errorHandler());
});

app.get('/', routes.index);

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

Система шаблонов Jade устанавливается вместе с платформой Express.

Пример jade-кода.

Jade-код. Листинг 10.10

```
html
  head
    title Это заголовок
  body
    p Это абзац
```

Данный jade-код преобразуется в следующую html-разметку

Сгенерированный html-код. Листинг 10.11

```
<html>
 <head>
  <title>Это заголовок</title>
 </head>
 <body>
  <p>Это абзац</p>
 </body>
</html>
```

Рассмотрим еще один пример, в котором используется имя класса и идентификатор.

Jade-код с классами и иднетификаторами. Листинг 10.12

```
html
  head
    title Это заголовок
  body
    div.content
        div#title
            p Это абзац
```

Этот код генерирует следующую разметку:

Сгенерированный html-код. Листинг 10.13

```
<html>
 <head>
  <title>Это заголовок</title>
 </head>
 <body>
  <div class="content">
  <div id="title">
  <p>Это абзац</p>
  </div>
  </div>
 </body>
</html>
```

Завершать элемент можно точкой, показывающей, что дальнейший блок содержит только текст.

Использование точки в jade-коде. Листинг 10.14

```
p.
 Большой блок текста
 Еще один блок
```

Атрибуты в jade-шаблон вставляются в круглых скопках.

Использование атрибутов в jade-коде. Листинг 10.15

```
input(type="text"
      name="widgetname")
```

Подключается движок шаблонов jade в файле app.js так:

Подключение jade-шаблона в app.js. Листинг 10.16

```
app.set('view engine', 'jade')
```

Внедрение шаблона в приложение

Создадим для приложения файл layout.jade. В нем используется инструкция doctype, подключающая HTML5, добавляется элемент head с элементами title и meta. Добавляется элемент body, а затем следует ссылка на инструкцию block с именем content.

Шаблон макета, созданный в Jade. Листинг 10.17

```
!!!
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/main.css')
  body

    block content
```

Чтобы воспользоваться новым макетом шаблона, необходимо каждый из шаблонов конента начинать следующей строкой.

Подключение layout.jade в шаблоне макета. Листинг 10.18

```
 extends layout
```

Например, рассмотрим файл index.jade

Index.jade. Листинг 10.19

```
 extends layout
block content
  p
   a(href='/widgets/new') Добавить
  table(width="100%")
    caption Widgets
      if widgets.length
        tr
          th ID
          th Name
          th Price
          th Description
          th
          th
        each widget in widgets
          if widget
            include row
```

В примере мы использовали три конструкции, которые необходимо пояснить.

Консрукция **if** – проверяет на существование переменной widgets.

Консрукция **each** – аналог консрукции foreach в смежных языках программирования.

Консрукция **include** подключает файл row.jade.

Подключаемый файл row.jade. Листинг 10.20

```
tr
  td #{widget.id}
  td #{widget.name}
  td $#{widget.price.toFixed(2)}
  td #{widget.desc}
  td
```

```
    a(href='/widgets/#{widget.id}/edit') Edit
  td
    a(href='/widgets/#{widget.id}') Delete
```

Осталось подключить файл шаблона в контроллере. Перепишем функцию index в контроллере widget.js

Подключаемый файл row.jade. Листинг 10.20

```
exports.index = function(req, res) {
    res.render('widgets/index', {title : 'Widgets', widgets : widg-
ets});
};
```

11. MySQL и Node.js

Модуль для работы с базой данных MySQL можно установить с помощью диспетчера Node-пакетов.

У Node.js существует несколько модулей для работы с базой MySQL. Будем использовать модуль Sequelize, т.к. он поддерживает функциональность ORM

npm instsal sequelize

Далее создадим папку config, а в ней конфигурационный файл config.js

Конфигурационный файл config.js. Листинг 11.1

```
var Sequelize = require("sequelize");
var sequelize = new Sequelize('test', 'root', '', {
    host: "127.0.0.1",
    port: 3308
});
global.sequelize = sequelize;
```

Подключим данный файл к app.js

Подключение конфигурационных настроек. Листинг 11.2

```
var config = require('./config/config');
```

Обратимся к базе данных из контроллера

Функция SELECT-запроса. Листинг 11.3

```
exports.index = function(req, res){
    sequelize.query("SELECT * FROM maintexts WHERE url =
'"+req.params.id+"'").success(function(myTableRows) {
        var myTab = myTableRows[0];
        res.render('index', { ttext: myTab });
    });
};
```

Таким образом, мы в шаблон index.jade передали массив ttext со значениями из базы данных.

Выводить на экран данные будем так:

Вывод элементов массива на экран. Листинг 11.4

```
h2 #{ttext.name}
div.main #{ttext.body}
```

10. Полезные команды cmd

wmic os get osarchitecture – узнать битность windows (32 или 64).

Список используемой литературы

«HTML5 для профессионалов», Хуан Диего Гоше, издательство Питер, 2013 год.
«HTML5. Недостающее руководство», Мэтью Мак-Дональд, издательство O'REILY, 2012 год.
«HTML5 и CSS3. Веб-разработка по стандартам нового поколения», Хоган Б., издательство Питер, 2012
«CSS ручной работы», Седерхольм Д. 2011

http://obmenka.by/code/allcode/

Методическое издание

Михалькевич Александр Викторович

HTML5

Авторская редакция
Компьютерная верстка Михалькевич А.В.

Подписано в печать 25.04.2013. Формат 60x84 1/16.
Бумага HPOffice, Печать лазерная.
Усл. печ. л. . Уч.-изд. л. .
Тираж 1000 экз. Заказ.