ОДО "Центр Обучающих Технологий "БелХард"

Михалькевич Александр Викторович

JavaScript

для продвинутых

методическое пособие

Минск 2016

УДК 004.415.53
ББК 32.973.2-018.2я7

Об авторе:

Михалькевич Александр Викторович, программист. Я профессионально занимаюсь web-разработкой с 2004 года. И постоянно, в той или иной степени сталкиваюсь с технологией JavaScript. Тем не менее, когда я принялся за углубленное его изучение я поразился его возможностям.

Первое, что меня поразило – это словочетание “серверный JavaScript”. Как это серверный? JavaScript всегда был клиентским. Если JavaScript стал серверным, тогда зачем серверные языки, такие как PHP, ASP, RUBY и другие?

Второе – это наличие в самом языке собственной базы данных. Какой еще язык программирования может этим похвастаться?

Третье – это возможность создавать web-приложения почти любой сложности, без использования дополнительных технологий. Мы можем создать сайт с авторизацией и системой администрирования на чистом JavaScript, без PHP и даже без CSS и HTML, используя только js-файлы.

JavaScript – это магия, с помощью которой можно создать приложение любой сложности.

Михалькевич Александр Викторович
JavaScript для продвинутых. Основы современной фронтенд разработки.

ОДО "Центр Обучающих Технологий "БелХард" - Мн., 2016. – 284с.

ISBN

- Данное учебное пособие является не просто введением в javaScript, а полноценным учебным курсом, в котором рассматриваются все основные аспекты javaScript. Включая платформу Node.js (серверный JavaScript). Создание Express приложения. API HTML5 и вспомогательные библиотеки, такие как jQuery, Bootstrap, Angular.

Пособие рекомендовано к использованию слушателям курсов ОДО "Центр Обучающих Технологий "БелХард".

УДК 004.415.53
ББК 32.973.2-018.2я7

© Михалькевич Александр Викторович 2016
© Закрытое акционерное общество
ISBN ОДО "Центр Обучающих Технологий "БелХард"

HTML5

JavaScript и API JavaScript

Node.js

Современный фронтенд

Содержание:

Введение……………………………………………………………………...6

Глава I. JavaScript

1. Базовый синтаксис……………………………………………………...9
   1.1 Тип данных..9
   1.2 Автоматическая вставка точек с запятой……………………………..11
   1.3 Область видимости переменных…………………………………...12
   1.4 Особенности работы с числами и строками..12
   1.5 Переменные и литералы………………………………………..13
   1.6 Инструкции...13
   1.7 Тернарный оператор ?:…………………………………………16
   1.8 Объекты..16
   1.9 Массивы………………………………………………………....18
   1.10 Стандартные объекты javaScript…………………………..………22
   1.11 Глобальные методы………………………………..……………..23
   1.12 Синтаксические конструкции………………………...……………25
2. Функции………………………………………………………………...27
   2.1 Определение функции………………………………………………27
   2.2 Вызов функци как функции…………………………………………27
   2.3 Вызов функции как метода…………………………………………..28
   2.4 Вызов функции как конструктора..29
   2.5 Функции высшего порядка……………………………………………30
   2.6 Косвенный вызов, методы call() и apply()………………………………..21
   2.7 Аргументы функций..33
   2.8 Замыкания...33
   2.9 Подъем переменных...35
   2.10 Немедленно-вызываемые функции-выражения……………………..36
   2.11 Метод bind()..37
3. Объекты и прототипы..39
   3.1 Конструктор...39
   3.2 Наследование…………………………………………………………...43
4. Тестирование JavaScript……………………………………………46
   4.1 Модульное тестирование……………………………………………46
   4.2 QUnit…………………………………………………………………..47

Глава II. API JavaScript

1. API видео и аудио ……………………………………………………..50
2. API холст…………………………………………………………………60
3. API перетаскивания……………………………………………………...83
4. API форм…………………………………………………………………90
5. API геолокации…………………………………………………………100

6. API web-хранилища……………………………………………………106

Глава III. Node.js

1. Основы……………………………………………………………....111
2. Ядро Node...115
2.1 Глобальные объекты ..…………………………………………….116
2.2 Тайменые методы……..…………………………………………120
2.3 Работа с файлами ………..………………………………………122
2.4 Службы прослушивания портов и методы создания серверов……..126
2.5 Сокеты UDT……...……………………………………………….131
2.6 Дочерние процессы ……………………..………………….....132
2.7 Система доменных имен………………..…………………….133
2.8 Модуль Utilites и объектное наследование ……………………..……135
2.9 Слушатели и генераторы событий …………………………...137
3. Node.js и PHPStorm ……………………………………………………139
4. Методика асинхронного программирования……………………………..145
5 Разработка приложения на простом паттерне ……………………….147
6. Отладка………………………………………………………………..149
7. Обработка ошибок...156
8. Внешние модули……………………………………………………...158
9. Хранилище данных MongoDB...161
9.1 Установка Mongo ……………………………………………...162
9.2 CRUD-операции ……………………………………………….163
9.3 Основы использования ………………………………………….166
9.4 Mongoose ……………………………………………………….168
10. MySQL и Node.js 170
11. Express…………………………………………………………………...171
11.1 Установка и настройка ………………………………..……171
11.2 Создание express-приложения ……………………..…………174
11.3 Структура Express ……………………………………..………176
11.4 Шаблонизация проекта ……………………………………...……180
11.5 Маршрутизация ………………………………………………184
11.6 Конфигурирование …………………………………..………186
11.7 Подключение скриптов и стилей …………………………..……....188
11.8 Подключение базы данных ……………………………..……...189
11.9 Регистрация и авторизация пользователей…………………..……..192
11.10 Сессии в Express …………………………………………..……..198
11.11 Загрузка файлов 202
12. Web-сокеты, чат на Socket.IO……………………………………..204

Глава IV. Современный фронтенд

1. Резиновая и фиксированная верстка, традиционная блочная и табличная...207

2. Гибкая блочная верстка..209
3. Адаптивная верстка…………………………………………………………….211
4. JSON…………………………………………………………………………..213
5. Селекторы…………………………………………………………………214
6. jQuery (библиотека запросов)………………………………………………….218
7. Объектные литералы...222
8. Архитектурный шаблон MVVM...223
9. Backbone………………………………………………………………………..224
10. CoffeeScript..226
11. LESS………………………………………………………………………...228
12. Системы сборки FrontEnd-a…………………………………………………232
13. Bootstrap 3, API Bootstrap…………………………………………………..236
14. Браузерные web-консоли. FireBug………………………………………...255
15. Schema.org………………………………………………………………….. .257
16. Angular.js…………………………...………………………………………261
17. Система контроля версий, GIT………………………………………………275

Приложение

Инструментарий…………………………………………………………………..280
Программа курса JavaScript для продвинутых, Belhard ……………………..281
Список используемой литературы...280

Введение

Почему JavaScript?

На сегодняшний день JavaScript – самый популярный язык в мире. Независимо от серверной платформы приложения, более 90% web-приложений в той или иной степени используют javaScript.

На сегодняшний день JavaScript является единственной технологией, на которой можно создать web-проект любой сложности без использования других технологий.

Но такого языка, или технологии, которые были бы однозначно признаны рынком как лучшее решение для web-разработчиков. У каждого варианта есть свои достоинства и недостатки. Однако можно определить самый распространенный язык. И этим языком снова оказывается JavaScript.

Более 90% сайтов и web-проектов в той или иной степени используют JavaScript. От выпадающих элементов меню и до управления базой данных; от Ajax-запросов (вызов серверных скриптов без перезагрузки страницы) и до Node.js (серверный JavaScript). Вот область применения самого популярного на сегодняшний день языка.

Современный JavaScript уже вышел за рамки клиентского (браузерного) языка. С появлением версии 2.0, JavaScript научился работать на стороне сервера и составил серьезную конкуренцию такому состоявшемуся языку, как PHP, который пока еще держит от 80 до 88% (по разным источникам) рынка серверных платформ, но с выходом окончательного релиза Node.js ситуация может измениться в пользу серверного JavaScript.

Сам язык изобрел Brendan Eich (компания Netscape) и назвал его JavaScript. Впервые новый язык был использован в браузере Netscape Navigator 2.0. После этого он стал использоваться во всех последующих браузерах от Netscape и во всех браузерах от Microsoft, начиная с Internet Explorer 3.0. Компания Microsoft по-своему развила идею и дала своей версии языка более короткое название: JScript.

Далее, чтобы обеспечить совместимость версий языка независимых разработчиков, Генеральной Ассамблеей ECMA был создан стандарт. Этот стандарт основан на нескольких базовых технологиях, наиболее известными из которых являются упомянутые уже JavaScript (Netscape) и JScript (Microsoft).

Развитие этого Стандарта началось в ноябре 1996. Первое издание Стандарта ECMA было принято Генеральнаой Ассамблеей ECMA в июне 1997.

Данный ECMA Стандарт был представлен международной комиссии по

стандартам ISO/IEC JTC 1 для принятия, и одобрен как международный эталон ISO/IEC 16262 в апреле 1998. Генеральная Ассамблея ECMA в июне 1998 одобрила второе издание ECMA-262, с сохранением всех требований ISO/IEC 16262.

В настоящее время используется третье издание ECMA-262 которое включает мощные регулярные выражения, лучшую обработку строк, новые инструкции контроля и управления, перехват и обработку исключительных ситуаций, более жесткое определение ошибок, форматирование для числового вывода и незначительные изменения в ожидании ввода средств многоязычности и будущего развития языка.

С появлением HTML5 расширились и возможности JavaScript. Наконец-то разработчики языка поняли необходимость обработки видео, и появилась API video и audio. С появлением API холста (canvas) отпала необходимость для двухмерной графики или анимации использовать флэш. Появились API перетаскивания, геолокации, индексированных баз данных, работы с файлами, истории, автономной работы, управления рабочими процессами и коммуникационный API. JavaScript даже обзавелся своей базой данных – MongoDB, данные хранятся в JSON-формате.

JavaScript, «самостоятельный» язык программирования, ведь прав на него нет ни у одной компании или группы разработчиков. Тем не менее, само название «JavaScript», является сертифицированным торговым знаком, «подсуетилась» по этому поводу компания Oracle Corporation.

Широко используется данный язык программирования в AJAX, суть его применения заключается в том, что с его помощью, во время работы со страницей веб браузера, она может не перезагружаться полностью, что, естественно, экономит уйму времени; в то же время, интерфейс какого-либо приложения, становится гораздо быстрее. Экономия времени, насколько известно, всегда является приоритетной чертой, при использовании любых приложений.

Так же свою популярность JavaScript заслужил в работе с таким механизмом, как Comeet, с помощью которого сервер отправляет какие-либо данные браузера без лишних на то запросов от браузера. Функционал этого механизма был бы неосуществимым без JavaScript.

И это лишь немногие возможности этого легкого и функционального языка программирования, который был создан людьми, и для людей. Такова и была изначальная задумка его разработчиков, которая, в итоге и стала основополагающей для роста популярности *JavaScrip*t.

Достоинства языка **JavaScript** - простота изучения его основ и внедрения. **JavaScript** использует самую обычную среду для разработки – его про-

граммы создаются в простых редакторах, таких, как «WordPad» или «Блокнот». Для файлов, созданных с использованием алгоритмов **JavaScript,** не требуется программа-компилятор, чтобы адаптировать их к языку машины, то есть переводить эти файлы на язык машинных кодов. Для просмотра и работы с программами, создаваемыми на **JavaScript,** подходит и используется обыкновенный Интернет-браузер.

Недостатки языка **JavaScript** заключаются в том, что разные браузеры могут по-разному интерпретировать код программы написанной на этом языке в составе html-документа, браузером пользователя также может блокироваться или приостанавливаться выполнение некоторых JS-скриптов, например, могут быть заблокированы всплывающие окна, тоже касается и поисковых систем (их роботов), которые не очень доверяют сайтам, на которых содержится большое количество JS-скриптов. Кроме того, **JavaScript,** естественно, уступает традиционным языкам программирования, таким, как Java, Delphi, Pascal, C, C++, которые, однако, используются в своей основе для создания локальных компьютерных приложений.

Изучение JavaScript - интересное и увлекательное занятие, которое позволяет развить и улучшить навыки программирования, а также создавать с его помощью уникальные законченные проекты. Требуется лишь время для того, чтобы освоить **JavaScript**, а также постоянная практика.

Желающим углубленно изучить JavaScript, можно порекомендовать курсы JavaScript Центра Обучающих Технологий Belhard. На курсах изучаются: базовый синтаксис, API языка, база данных MongoDB, клиентский JavaScript и серверный (Node.js), а также библиотеки современные js-библиотеки.

Глава I. JavaScript

1. Базовый синтаксис

JavaScript подключается к HTML-странице и выполняет код на стороне клиента (браузера).

2 способа подключения JavaScript к странице HTML:

- <script>//JavaScript код</script>
- <script src="/my/script.js"></script>

JavaScript-среды поддерживают две версии языка: строгий режим и не строгий режим.

Строгий режим включается в программе путем добавления в начале кода строковой константы:

Включение строгого режима. Листинг 1.1

```
"use strict"
```

Кроме того строгий режим может быть включен в функцию:

Включение строгого режима в тело функции. Листинг 1.2

```
function f(a){
  "use strict";
}
```

Необходимо придерживаться либо *только строгого режима*, либо *только не строгого режима*. Т.к. смешивание режимов может привести к конфликтам.

Простейший способ структурирования своего кода на совместимость в строгом режиме является размещение всего кода в функции:

Размещение всего кода в функции. Листинг 1.3

```
(function(){
  "use strict";
  function f(a){
    "use strict";
  }
})()
```

1.1 Тип данных

JavaScript работает со следующими типами данных: null, undefined, String, Number, Boolean, Object, Array, Date, ReqExp, Error, Function.

Все типы данных, кроме null и undefined, имеют свои классы-конструкторы. Это специальные встроенные функции, которые вызываются для создания экземпляров своего типа данных.

Узнать, к какому типу данных принадлежит значение переменной a, можно узнать с помощью оператора typeof:

Проверка типа переменной. Листинг 1.4

```
typeof a;
```

Кроме перечисленных типов данных, javaScript поддерживает еще один важный тип – глобальный объект, который неявно присутствует в каждой переменной.

JavaScript автоматически определяет тип данных. Иногда это приводит к непредвиденным последствиям:

Сложение числа с булевым значением. Листинг 1.5

```
1+true; //2
```

При сложении строки с числом, JavaScript отдает предпочтение строке:

Сложение числа со строкой. Листинг 1.6

```
"2"+3; // 23
```

Для сравнения значений на равенство либо не равенство между собой используются операторы == (двойное равно) или === (тройное равно).

Оператор == сравнивает два значения без учета типов значений. Оператор === сравнивает два значения с учетом типов каждого значения. Если два аргумента относятся к одному типу, между ними нет никакой разницы. Тем не менее, стоит избегать использования оператора со смешанными типами (==). Сравнение двойным равно может привести к неожиданным результатам. Например, код из следующего листинга вернет true:

Сравнение аргументов двойным равно, которое вернет true. Листинг 1.7

```
"1.0e0" == { valueOf: function(){return true;} }
```

Чтобы поведение программы было понятнее при сравнении значений разных типов необходимо использовать явное приведение типов данных и использовать оператор ===.

Переменная, значением которой окажется null, не вызывает сбоя при арифметическом вычислении, она будет преобразована в 0.

Неопределенная переменная преобразуется в специальное значение с плавающей точкой NaN. Причем, NaN рассматривается как неравное самому себе.

Проверка значения на равенство NaN не работает. Листинг 1.8

```
var x = NaN
x === NaN; //false
```

Протестировать переменную на значение NaN можно путем проверки ее на неравенство самой себе:

Тестирование переменной на значение NaN. Листинг 1.9

```
var x = NaN
x !== x; // true
```

Можно создать специальную функцию:

Функция тестирования переменной на значение NaN. Листинг 1.10

```
function isNaN(x){
 return x !== x;
}
```

Еще одна разновидность приведения типов данных называется истинностью. Операторы if, ||, && работают с булевыми значениями, но на самом деле принимают любые. Большинство значений JavaScript являются истинными, т.е. неявным образом приводятся к true. Существует лишь 7 ложных значений:

- *false, 0, -0, “”, NaN и undefined*

Все остальные значения истинны.

1.2 Автоматическая вставка точек с запятой

В JavaScript в конце каждой конструкции точка с запятой ставится автоматически. Отсутствие точек с запятой дает более лаконичный и понятный код. Пропускать точки с запятой можно только в конце строки. Рассмотрим пример:

Правильная и неправильная функция. Листинг 1.11

```
function area(r){r=+r; return Math.PI*r} //правильная функция

function area(r){
 r=+r;
 return Math.PI*r
} //правильная функция

function area(r){r=+r return Math.PI*r} //неправильная функция
```

Первое правило вставки точек с запятой гласит:

- *Точки с запятой ставятся только перед лексемой } после одного или нескольких разделителей строк или вконце входных данных программы.* Иными словами пропускать точки с запятой можно только вконце строки, блока или программы.

Второе правило вставки точек с запятой гласит:

- Точки с запятой ставятся только если следующая ликсема не может быть проанализирована.

Можно взять себе за общее правило *всегда предворять дополнительной точкой с запятой инструкции, начинающиеся символом (,[+-/*

1.3 Область видимости переменных

“*Область видимости переменных - это как кислород для программистов. Никогда об этом не задумываешься, но когда попадают примеси начинаешь задыхаться*”

Дэвид Херман “Сила JavaScript”.

Переменная JavaScript может быть как локальной области видимости, так и глобальной. Глобальными переменными пользуются только разработчики библиотек JS. Поэтому, если вы не разрабатываете свою JS-библиотеку, всегда используйте переменные локальной области видимости, добавляя перед именем переменной ключевое слово var.

Объявление переменных. Листинг 1.12

```
a = 'string' // глобальной видимости
var b = 'string' // локальной видимости
```

1.4 Особенности работы с числами и строками

Все числа в JavaScript являются числами с *плавающей точкой двойной точности*.

Целые числа являются не отдельным типом данных, а поднабором чисел с двойной точностью.

При работе с JavaScript необходимо помнить, что он чувствителен к регистру.

1.5 Переменные и литералы

Для работы с полученными результатами и их хранения используются переменные. У каждой переменной должно быть имя (идентификатор). Фактически, это адресация к памяти, в которой эти значения и хранятся.

В качестве имен переменной категорически запрещается использовать ключевые и зарезервированные слова.

Для формирования имен можно использовать в любом количестве:

- a – z, A – Z (латинские буквы в любом регистре)
- 0 – 9 (арабские цифры)
- _ (символ подчеркивания)
- $ (знак доллара)

Имя переменной не может начинаться с цифры.

Если для создания переменой использовать инициализатор var, то переменная объявляется локальной.

Объявление переменной. Листинг 1.13

```
var a = 'string';
```

Значение переменных часто называют литералы. Литерал относится к одному из заранее определенных типов данных.

1.6 Инструкции

Инструкция – выражение, в результате которого должны выполниться действия (например, объявление или инициализация переменной и т.д.).

Каждая простая инструкция заканчивается “;”. Если этого не делать, браузер проставляет их за нас. Ориентируется он в этом случае на переносы строки.

Несколько выражений могут составлять блок инструкций:

Блок инструкций. Листинг 1.14

```
{
 x = Math.PI;
 y = Math.cos(x);
}
```

Инструкции var и function являются инструкциями объявления. Инструкция var объявляет переменную.

Инструкция var. Листинг 1.15

```
var a = 'string';
```

Инструкция function объявляет функцию.

Инструкция функции. Листинг 1.16

```
function myFunc(){
  //-- тело функции
}
```

Составные инструкции заранее определены в языке и имеют свой синтаксис:

- Условные инструкции (if / else if / else)
- Инструкция переключения (switch / case / default)
- Инструкции цикла (while, do / while, for, for in)
- Инструкция перехвата и обработки исключения (try / catch / finally)

Инструкция if / else if / else. Листинг 1.17

```
if(n==1){
 // выполнить первый блок
}
elseif(n==2){
 // выполнить второй блок
}
else{
 // выполнить блок по умолчанию.
}
```

Если условная инструкция if/else принимает более одного параметра, то предпочтительнее использовать инструкцию switch. Рассмотрим инструкцию switch:

Инструкция switch. Листинг 1.18

```
switch(n){
  case 1:
   //выполнить первый блок, если n == 1
   break;
  case 2:
   //выполнить второй блок, если n == 2
   break;
  default:
   //выполнить блок по умолчанию.
}
```

Рассмотрим инструкции циклов:

Инструкция while. Листинг 1.19

```
var count = 0;
while(count<10){
 console.log(count);
 count++;
}
```

Инструкция do/while во многом похожа на инструкцию while, за исключением того, что инструкция do/while сперва делает, потом проверяет условие. Таким образом, хотя бы один раз она обязательно сработает.

Инструкция do/while. Листинг 1.20

```
do {
 // что-то сделать
}while(++i<len)
```

Инструкция for часто оказывается более наглядной инструкцией цикла. Инициализация, проверка и инкремент (наращивание переменной) – это три операции, выполняемые с переменной цикла.

Инструкция for. Листинг 1.21

```
for(var count=0; count<10; count++){
 console.log(count);
}
```

Инструкция for/in использует ключевое слово for, но она в корне отличается от инструкции for. For/in имеет следующий синтаксис:

for(переменная in объект).

Инструкция for. Листинг 1.22

```
for(p in o){
 console.log(o[p]);
}
```

Любая инструкция может быть помечена меткой

Идентификатор: инструкция.

Инструкция *break* приводит к немедленному выходу из внутреннего цикла или из инструкции switch.

Инструкция *continue* во многом схожа с инструкцией break, но приводит не к выходу из инструкции, а к новой итерации цикла.

Инструкция *return* внутри функции служит для определения значений, возвращаемых функцией. Имеет следующий синтаксис:

return выражение;

Инструкция throw предназначена для перехвата ошибок инструкции try/catch

Инструкция try/catch/fitally. Листинг 1.23

```
try {
   throw "myException"; // возбуждение исключительной ситуации
}
catch (e) {
```

```
   logMyErrors(e); // вывод ошибок
}finally{
   // этот блок выполняется всегда, независимо от наличия либо
отстутсвия ошибок.
}
```

1.7 Тернарный оператор ?:

Инструкция вида if/else может быть представлена в виде так называемого условного или тернарного оператора

проверка_условия ? значение_1 : значение_2.

Например, данная инструкция:

Инструкция if. Листинг 1.24

```
if (x > y) {
  var z = 'x больше y';
} else {
  var z = 'x меньше y';
}
```

Может быть представлена в виде тернарного оператора:

Тернарный оператор. Листинг 1.25

```
var z = x > y ? 'x больше y' : 'x меньше y';
```

1.8 Объекты

Любое значение в JavaScript, не являющееся строкой, числом, true, false, null или undefined, является объектом.

Наиболее типичными операциями с объектами является: создание объектов, назначение, получение, удаление, проверка и перечисление свойств. Свойство имеет имя и значение.

Объекты можно создавать тремя способами: с помощью литералов объектов, ключевого слова *new* и функции *Object.create()*.

1. Создание объектов с помощью литералов объектов:

Литералы объектов. Листинг 1.26

```
var empty = {};
var point = {x:0, y:1};
var book = {
  "title": "JavaScript",
  author: {
   firstname: "David",
   surname: "Flanagan"
  }
}
```

2. Оператор new создает и инициализирует новый объект. За этим оператором должно следовать имя функции. Функция,используемая таким способом, называется конструктором.

Создание объекта с помощью оператора new. Листинг 1.27

```
var o = new Object();
var a = new Array();
var d = new Date();
var r = new RegExp('js');
```

Помимо встроенных функций-конструкторов, имеется возможность определять свои конструкторы.

3. Статическая функция Object.create() создает новый объект и использует свой первый аргумент в качестве прототипа этого объекта.

Создание объекта с помощью Object.create. Листинг 1.28

```
var o = Object.create({x:1, y:2})
```

Прототипом (prototype) называется родительский объект, ассоциированный с создаваемым объектом. Все объекты, созданные с помощью литерала объектов, имеют один и тот же прототип, на который можно сослаться так:

Object.prototype

Объекты, созданные с помощью ключевого слова new, в качестве прототипа получают прототип функции конструктора: *Object.prototype*, *Array.prototype*, *Date.prototype*, *RegExp.prototype*.

Наследование осуществляется методом inherit.

Метод inherit. Листинг 1.29

```
var o = { }
o.x = 1;
var p = inherit(o); // p наследует свойства объектов o
```

Свойства объекта

Получение свойств объектов. Листинг 1.30

```
var author = book.author;
var author = book['author'];
```

Если у объекта book не окажется свойства author, данное свойство будет искаться в наследуемых прототипах.

Попытка обращения к несуществующему свойству не является ошибкой, возвращается значение undefined. Однако попытка обращения к свойству несуществующего объекта – это ошибка.

Оператор delete удаляет свойство из объекта:

Удаление свойств объектов. Листинг 1.31
```
delete book.author
delete book['author']
```

С помощью оператора in мы можем проверить существование свойства объекта.

Проверка существования свойств. Листинг 1.32
```
var o = {x:1}
"x" in o // вернет true
"y" in o // вернет false
```

Проверку можно осуществлять с помощью метода hasOwnProperty:

Метод hasOwnProperty. Листинг 1.33
```
var o = {x:1}
o.hasOwnProperty("x"); //вернет true
o.hasOwnProperty("y"); //вернет false
```

Существует еще один метод для проверки свойств. Метод propertyIsEnumerable возвращает true только в том случае, если указанное свойство является собственным свойством, а не наследуемым.

Перечисление свойств. Для перечисления всех свойств объекта можно воспользоваться функцией for/in:

Перечисление свойств, инструкция цикла for/in. Листинг 1.34
```
var o = {x:1, y:2, z:3};
for(p in o)
  console.log(p);
```

Сериализация объектов – это процесс преобразования объекта в строку, которая впоследствии может быть использована для восстановления объекта. Для сериализации объекта используется встроенная функция JSON.stringify. Для последующего восстановления – JSON.parse().

Сериализация и восстановление объекта. Листинг 1.35
```
var o = {x:1, y:2, z:3};
s = JSON.stringify(o) // s = "{x:1, y:2, z:3}"
p = JSON.parse(s); // p – копия объекта o.
```

1.9 Массивы

Массивы – это упорядоченная коллекция значений. Значения в массиве называются элементами, каждый элемент содержит индекс, который может быть числовым или строковым. По индексу можно обратиться к конкретному элементу.

Проще всего создать массив с помощью литерала, который представляет собой список разделенных запятыми элементов массива в квадратных скобках

Создание массива с помощью литерала. Листинг 1.36

```
var empty = [] // пустой массив
var primes = [2,3,4,10] // массив с четырмя элементами
var misc = [
 1.1,
 true,
 "string",
 {x:1, y:2},
 [2, {z:3, w:4}] // литералы массивов могут содержать литералы объектов или литералы других массивов.
]
```

Любой массив имеет свойство length, возвращающее длину (количество элементов) массива. Именно это свойство отличает массивы от обычных объектов JavaScript.

Свойство length. Листинг 1.37

```
[].length // вернет 0, т.к. массив не имеет элементов.
a =['a', 'b', 'c'];
a.length // вернет 3
```

Существует два способа для добавления элементов в конец массива

1. Можно воспользоваться методом push.

2. Пустые квадратные скопки с индексом.

Метод push(), добавление элементов в конец массива. Листинг 1.38

```
a = [];
a.push('zero');
a.push('one', 'two')
a['one more'] = 'aa';
a[a.length] // также вставляет элемент в конец массива
```

Добавить элементы в начало массива можно с помощью метода shift(). Синтаксис такой же как у push.

Для удаления элементов можно воспользоваться методом delete.

Удаление элемета массива с помощью оператора delete. Листинг 1.39

```
delete a[1]
```

Оператор *delete* удаляет элемент массива, но не изменяет длину *length*. Поэтому удаление элемента оператором *delete* похоже на присваивание этому элементу значения *undefined*.

Также имеется возможность удалять элементы с изменением его длины. Метод *pop()* (противоположный методу *push()*) уменьшают длину массива на 1 и возвращают значение удаленного элемента.

Для вставки элемента в начало массива можно воспользоваться методом *unshift()*. Для удаления элемена в начале массива есть метод *shift()*.

Для обхода по элементам массива наиболее часто используется цикл for.

Обход массива с помощью цикла for. Листинг 1.40

```
for(var i=0; i<a.length; i++){
 if(!a[i]) continue // пропустить несуществующие элементы
 // тело цикла
}
```

JavaScript позволяет имитировать многомерные массивы при помощи массивов из массивов.

Многомерные массивы. Листинг 1.41

```
var matrix = [
  [3, 3, 3],
  [1, 5, 1],
  [2, 2, 2]
];
```

Рассмотрим методы массивов:

Метод *join()* преобразует все элементы в строки, объединяет их и возвращает получившуюся строку.

Метод join(), преобразование массива в строку. Листинг 1.42

```
var a = [1,2,3];
a.join() // вернет "1,2,3"
a.join(" ") // вернет "1 2 3"
a.join("") // вернет "123"
var b = new Array(10);
b.join("-") // вернет "----------"
```

Метод *reverse()* меняет порядок следования элементов в массиве на обратный и возвращает переупорядоченный массив.

Метод reverse(), переворачивание массива. Листинг 1.43

```
var a = [1,2,3];
a.reverse().join() // вернет "3,2,1", теперь a = [3,2,1]
```

Метод *sort()* сортирует элементы в исходном массиве и возвращает отсортированный массив.

Метод *concat()* добавляет к массиву элементы переданные параметром метода. При этом исходный массив не изменяется.

Метод concat(), добавление элементов. Листинг 1.44

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

Метод *slice()* возвращает фрагмент, или подмассив указанного массива.

Метод slice(), фрагмент массива. Листинг 1.45

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

Метод *splice()* – универсальный метод, совершающий вставку или удаление элементов массива. В отличии от методов slice() и concat(), данный метод изменяет исходный массив.

Метод splice(). Листинг 1.46

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

Метод *forEach()* выполняет обход элементов массива, и для каждого из элементов возвращает указанную функцию.

Метод forEach(). Листинг 1.47

```
var a = [1,2,3,4,5,6,7,8]
var sum = 0;
a.forEach(function(value){
  sum+=value
});
```

Метод *map()* вызывает функцию точно также, как и метод forEach(), однако функция, передаваемая методу map(), должна возвращать значение.

Метод map(). Листинг 1.48

```
var a = [1,2,3,4,5,6,7,8]
b = a.map(function(x){
 return x*x;
});
```

Метод *filter()* возвращает отфильтрованный массив.

Метод filter(). Листинг 1.49

```
var a = [1,2,3,4,5,6,7,8]
smallvalues = a.filter(function(x){
 return x<3; // вернет [2,1]
});
```

Метод *every()* возвращает true, если функция вернула true для всех элементов массива.

Метод every(). Листинг 1.50

```
var a = [1,2,3,4,5,6,7,8]
a.every(function(x){
 return x<10 // вернет true, т.к. все значения <10
```

```
});
```

Метод *some()* возвращает true, если функция возвращает true хотя бы для одного элемента массива.

Метод some(). Листинг 1.51

```
var a = [1,2,3,4,5,6,7,8]
a.some(function(x){
 return x%2===0; // вернет true
});
```

Метод *reduce()* объединяет элементы массива используя указанную функцию.

Метод reduce(). Листинг 1.52

```
var a = [1,2,3,4,5,6,7,8]
a.reduce(function(x,y){
 return x+y; // сумма значений
});
```

Метод *reduceRight()* действует точно также, как и метод *reduce()*, но обрабатывает массив в обратном порядке, справа налево.

Методы *indexOf()* и *lastIndexOf()* отыскивают в массиве элемент с указанным значением и возвращают индекс первого найденного элемента или – 1, если элемент не найден. *Метод indexOf()* выполняет поиск от начала массива к концу, метод *lastIndexOf()* – с конца к началу.

Проверить является ли объект массивом можно с помощью метода *isArray()*.

Метод isArray(). Листинг 1.53

```
Array.isArray([]) // true
Array.isArray({}) // false
```

Еще один пример, показывающий как JavaScript может работать со строками, как с массивами.

Строки как массивы. Листинг 1.54

```
s = "JavaScript";
Array.prototype.join.call(s, " ") // вернет J a v a S c r i p t
```

1.10 Стандартные объекты JavaScript

Array Массив пронумерованных элементов, также может служить стеком или очередью

Boolean Объект для булевых значений

Date Функции для работы с датой и временем

Error объект для представления ошибок

EvalError Ошибка при выполнении функции eval

Function Каждая функция в яваскрипт является объектом класса Function.

Math Встроенный объект, предоставляющий константы и методы для математических вычислений.

Number Объект для работы с числами

Object Базовый объект javascript

RangeError Ошибка, когда число не лежит в нужном диапазоне

ReferenceError Ошибку при ссылке на несуществующую переменную

RegExp Позволяет работать с регулярными выражениями.

String Базовый объект для строк. Позволяет управлять текстовыми строками, форматировать их и выполнять поиск подстрок.

SyntaxError Ошибка при интерпретации синтаксически неверного кода

TypeError Ошибка в типе значения

URIError Ошибка при некорректном URI

Объекты браузера

window Два в одном: глобальный объект и окно браузера

1.11. Глобальные методы

alert Выводит модальное окно с сообщением

clearInterval Останавливает выполнение кода, заданное setInterval

clearTimeout Отменяет выполнение кода, заданное setTimeout

confirm Выводит сообщение в окне с двумя кнопками: "ОК" и "ОТМЕНА" и возвращает выбор посетителя

decodeURI Раскодирует URI, закодированный при помощи encodeURI

decodeURIComponent Раскодирует URI, закодированный при помощи encodeURIComponent

encodeURI Кодирует URI, заменяя каждое вхождение определенных символов на escape-последовательности, представляющие символ в кодировке UTF-8.

encodeURIComponent Кодирует компоненту URI, заменяя определенные символы на соответствующие UTF-8 escape-последовательности

eval Выполняет строку javascript-кода без привязки к конкретному объекту.

isFinite возвращает, является ли аргумент конечным числом

isNaN Проверяет, является ли аргумент NaN

parseFloat преобразует строковой аргумент в число с плавающей точкой

parseInt преобразует строковой аргумент в целое число нужной системы счисления

prompt Выводит окно с указанным текстом и полем для пользовательского ввода.

setInterval Выполняет код или функцию через указанный интервал времени

setTimeout Выполняет код или функцию после указанной задержки

1.12. Синтаксические конструкции

break Завершает текущий цикл или конструкции switch и label и передает управление на следующий вызов

continue Прекращает текущую итерацию цикла и продолжает выполнение со следующей итерации

do..while Задает цикл с проверкой условия после каждой итерации

for Создать цикл, указав начальное состояние, условие и операцию обновления состояния

for..in Перебрать свойства объекта, для каждого свойства выполнить заданный код

function Объявить функцию

if Выполняет тот или иной блок кода в зависимости от того, верно ли условие

label Указать идентификатор для использования в break и continue

return Возвратить результат работы функции

switch Сравнивает значение выражения по различными вариантами и при совпадении выполняет соответствующий код

throw Инициировать("бросить") исключение

try..catch Ловить все исключения, выпадающие из блока кода

var Объявить переменную (или несколько) в текущей области видимости

while Задает цикл, который выполняется до тех пор, пока условие верно. Условие проверяется перед каждой итерацией.

with Добавить новую область видимости

Блок Группировка javascript-вызовов внутри фигурных скобок

2. Функции

2.1 Определение функции

Функция – это блок программного кода на языке JavaScript, который определяется один раз и может вызываться многократно. Функции могут иметь параметры, или аргументы, – локальные переменные, значения которых определяются при вызове функции.

Определение javaScript-функций. Листинг 2.1

```
function print(msg)
{
    document.write(msg, "<br>");
}
```

Функция может быть присвоена переменной, вызываться через эту переменную, и при этом работать как функция.

Функция как переменная. Листинг 2.2

```
function print(x)
{
    return x*x
}
var s = print();
s(4);
```

Имеется возможность определять вложенные функции.

Вложенные функции. Листинг 2.3

```
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
hypotenuse()() // так можно вызывать вложенную функцию
```

В javaScript существует 4 способа вызова функций.

- Как функции
- Как методы
- Как конструкторы
- С помощью методов call и apply

2.2 Вызов функци как функции

Вызов функции как функции. Листинг 2.4

```
myfunc();
myfunc(x);
myfunc(x,y,z);
myfunc({x:1});
```

Простейшей моделью является вызов функции:

Вызов функции. Листинг 2.5

```
function myfunc(username){
  return 'Привет ' + username
}
myfunc('Вася'); // Привет Вася
```

При вызове функции возвращаемое значение становится значением выражения вызова. Если функция имеет ключевое слово return, то возвращается значение выражения, следующее за инструкцией return. Если функция не имеет выражения return, то возвращается undefined.

2.3 Вызов функции как метода

Метод – это функция, которая хранится в виде свойства объекта.

Возвращаемые значения обрабатываются точно также, как и при вызове обычной функции. Однако есть одно важное отличие. Любая функция, используемая как метод, фактически получает **неявный аргумент** – объект, относительно которого она была вызвана. Тело функции получает возможность ссылаться на объект с помощью ключевого слова **this**.

Вызов функции как метода. Листинг 2.6

```
var calculator = {
  operand1:1;
  operand2:1;
  add: function(){
    this.result = this.operand1+this.operand2;
  }
}
  calculator.add();
  calculator.result; // вернет 2
```

В отличии от переменных, ключевое слово this не имеет областей видимости. И вложенные функции не наследуют значение this вызываемой функции. Чтобы разобраться с поведением this рассмотрим еще один пример:

Вызов метода. Листинг 2.7

```
var obj = {
  hello: function(){
    return "Привет " + this.username
  },
  username: "Миша"
};
obj.hello(); // Привет Миша
```

Обратите внимание на то, как hello ссылается на this. Теперь мы можем скопировать ссылку на ту же самую функцию и получить другой ответ.

Продолжение предыдущего листинга. Создание другого объекта, использующего функцию hello объекта obj. Листинг 2.8

```
var obj = {
  hello: obj.username,
  username: "Маша"
```

```
};
obj.hello(); // Привет Маша
```

На самом деле в вызове метода вариант связывания this (получатель вызова), определяет само выражение вызова. Выражение obj.hello() ищет свойство hello объекта obj. А выражение obj2.hello ищет свойство hello объекта obj2.

Поскольку методы являются не чем иным, как функциями вызванными для конкретного объекта, сослаться на this может и обычная функция:

Ссылка обычной функции на this. Листинг 2.9

```
function hello(){
 return "Привет " + this.username
}
```

Это может пригодится на предопределения функции для ее совместного использования несколькими объектами. Однако этот трюк в JavaScript редко и малоприменим.

2.4 Вызов функции как конструктора

Третьей разновидностью функции является конструктор.

Если вызову функции предшевствует ключевое слово new, то это вызов конструктора. При вызове функции как конструктора, пару круглых скопок можно опустить:

Функция как конструктор. Листинг 2.10

```
var o = new Function();
var o = new Function;
```

Также, как методы и простые функции, конструкторы определяются с помощью ключевого слова function:

Определение конструктора. Листинг 2.11

```
function User(name, password){
  this.name = name;
  this.password = password
}
```

Вызов User с помощью оператора new создает из функции User конструктор.

Вызов функции User, как конструктора. Листинг 2.12

```
var u = new User('Вася', '123');
u.name // Вася
```

Основная роль функции-конструктора заключается в иницилизации объекта.

2.5 Функции высшего порядка

Это такие функции, которые оперируют функциями, применяя одну или более функций и возвращая новую функцию.

Поскольку при этом возвращается новая функция, то ее назвали функцией обратного вызова, или callback-функцией. Подход с использованием callback-функций часто используется в программах, написанных на JavaScript.

Например, рассмотрим функцию простого преобразования строкового массива с использованием метода map.

Использование функций высшего порядка для прохода. Листинг 2.13

```
var names = [‘яблоко’, ‘груша’, ‘слива’];
var upper = names.map(function(name){
 return name.toUpperCase();
});
upper // [‘ЯБЛОКО’, ‘ГРУША’, ‘СЛИВА’],
```

Наличие в программах дублированного или похожего кода – верный признак необходимости использовать функции высшего порядка.

Рассмотрим еще один пример возможного использования функций высшего порядка:

Вычисление стандартного отклонения. Листинг 2.14

```
var sum = function(x,y){return x+y};
var data = [1,1,3,4,8,8];
var deviations = data.reduce(sum)/data.length;
```

Задача: Функция должна принимать два значения. Первый - число, второе – функция, циклично вызывающаяся с параметрами от 0 до числа первого параметра функции

Решим эту задачу используя функции высшего порядка.

Реализация функции высшего порядка. Листинг 2.15

```
function buildString(n, callback){
  var result = ‘’;
  for(var i=0; i<n; i++){
    result += callback(i);
  }
  return result;
}
var digits = buildString(10, function(i){
 return i;
});
```

Такой подход называют абстракцией высшего порядка.

2.6 Косвенный вызов, методы call() и apply()

Методы call() и apply() позволяют выполнить косвенный вызов функции, как если бы она была методом другого объекта. Первым параметром ободим методам передается объект, относительно которого вызывается функция, этот аргумент определяет ключевое слово this в теле функции.

Чтобы вызвать функцию без аргументов, ка метод объекта o, можно использовать любой из методов: call() или apply().

Косвенный вызов функций через методы call() и apply(). Листинг 2.16

```
f.call(o);
f.apply(o);
```

call()

Метод call может применяться для вызова функции в контексте нужного объекта:

Вызов sayName в контексте разных объектов. Листинг 2.17

```
var Animal1 = {name: 'Cat'}
var Animal2 = {name: 'Dog'}

function sayName() {
    // this — ссылка на объект, в контексте которого вызвана функция
    alert(this.name);
}

sayName.call(Animal1) // выдаст сообщение "Cat"
sayName.call(Animal2) // выдаст сообщение "Dog"
```

При этом совершенно не важно, какому объекту принадлежит функция. В качестве текущего(this) объекта будет взят первый аргумент.

Вызов sayName, как метода объекта в контексте разных объектов. Листинг 2.18

```
var Animal1 = {
    name: 'Cat',
    sayName: function() {
        alert(this.name);
    }
};

var Animal2 = {name: 'Dog'};

Animal1.sayName() // выдаст сообщение "Cat"
Animal1.sayName.call(Animal2) // выдаст сообщение "Dog"
```

Помимо смены контекста вызова, метод call может передавать в функцию аргументы:

Вызов функции с аргументами. Листинг 2.19

```
var obj = {attr: 10};
```

```
function sum(a, b) {
    alert(this.attr + a + b);
}

sum.call(obj, 5, 2) // выдаст сообщение с результатом "17"
```

Если контекст вызова не указан, то функция будет выполнятся в контексте заданного объекта:

Выполнение функции в контексте заданного объекта. Листинг 2.20

```
window.a = 5
function sayThis() {
     alert(this.a);
}

sayThis.call() // выдаст 5

window.a = 5
function sayThis(b) {
     alert(this.a + b);
}

sayThis.call(null, 3) // выдаст 8
```

apply()

Метод apply() действует точно также, как метод call(), за исключением того, что apply() аргументы для функции передает ввиде элементов массива:

Передача аргументов функции методом apply(). Листинг 2.21

```
f.apply(o,1,2);
```

Метод apply() получает массив аргументов и вызывает функцию, как будто каждый элемент массива является отдельным аргурментом вызова функции. Поэтому он полезен для вызова вариативных (с неизвестным количеством аргументов) функций с вычисляемым массивом аргументов.

Кроме массива аргументов, метод apply получает первый аргрумент, указывающий на вариант связывания this для вызываемой функции. Если функция не ссылается на this, ей можно передать null.

Вызов функции не ссылающейся на this. Листинг 2.22

```
var score = [0,1,2]
average.apply(null, score);
// это будет равносильно вызову average(score[0], score[1], score[2])
```

Если функция ссылается на this, то необходимо вместо null указать объект, в котором находятся ссылки this.

Вызов функции ссылающейся на this. Листинг 2.23

```
var obj = {}
var scor = [0,1,2]
average.apply(obj, score);
```

```
// это будет равносильно вызову: {obj.average(scor[0],scor[1],scor[2])
```

Т.е. функция average превратилась в метод объекта obj.

Еще один пример использования apply():

Суммирование элементов массива. Листинг 2.24

```
var test = [1,7,2,20];
alert(sum.apply(null, test));   //выведет 30
```

2.7 Аргументы функций

Список аргументов можно получить с помощью свойства arguments.

Arguments как массив. Листинг 2.25

```
function f(x) {
    print(x);             // Выводит начальное значение аргумента
    arguments[0] = null; // Изменяя элементы массива, мы изменяем x
    print(x);             // Теперь выводит "null"
}
```

Если число аргументов в функции превышает число имен параметров при вызове, функция не может напрямую обращаться к неименованным значениям. Однако, ко всем параметрам функции мы можем обращаться через свойство arguments. Благодаря этому свойству имеется возможность создавать функции с переменным числом аргументов.

Вариативная функция. Листинг 2.26

```
function average(){
  for(var i=0, sum=0, n=arguments.length; i<n; i++){
    sum += arguments[i];
  }
  return sum/n;
}
```

Эта функция проходит циклический перебор каждого элемента объекта arguments и возвращает их среднеарифметическое значение.

2.8 Замыкания

В javaScript используются лексические области видимости. Это означает, что при выполнении функций действуют области видимости переменных, которые имелись на момент их определения, а не на момент вызова.

Реализация замыканий. Листинг 2.27

```
var scope = "global";
function check(){
  var scope = "local";
  function f(){return scope}
  return f;
}
check()(); // вернет "local"
```

Рассмотрим следующий пример преобразования замыканий.

Счетчик. Листинг 2.28

```
function counter(){
 var n = 0;
 return {
   count: function(){return n++;}
   reset: function(){n=0;}
 };
}
var c = counter(), d = counter; // создать два счетчика
c.count(); // вернет 0
d.cound(); // вернет 0
c.reset(); // обнуление, вернет 0

c.count(); // вернет 0
d.count(); // вернет 1
```

Чтобы понять суть замыканий, требуется знать три основных факта:

1. JavaScript позволяет ссылаться на переменные, определенные за пределами текущей функции:

```
1  function main(){
2    var magic = "фрукты";
3    function make(filling){
4      return magic + " и " + filling;
5    }
6    return make("овощи");
7  }
8  main(); //фрукты и овощи
9
```

- ссылка на переменные определенные за пределами текущей функции.

2. Функции могут ссылаться на переменные, определенные во внешних функциях. Функции, отслеживающие переменные, в содержащих эти переменные областях видимости, и называются **замыканиями**.

```
1  function main(){
2    var magic = "фрукты";
3    function make(filling){
4      return magic + " и " + filling;
5    }
6    return make;
7  }
8  var f = main();
9  f("овощи"); //фрукты и овощи
10
```

- можно возвращать внутреннюю функцию для его последующего вызова. Функция make() является замыканием, код которого ссылается на две внешние переменные: magic и filling. Функция может ссылаться на люые переменные в своей области видимости.

Этим можно воспользоваться для создания более универсальной функции main()

Универсальная функция main() использующая механизм замыканий. Листинг 2.29

```
function main(magic){
  function make(filling){
    return magic + " и " + filling;
  }
  return make;
}
var f = main("овощи");
f("фрукты"); //овощи и фрукты
f("огород"); //овощи и огород
var f = main("кофе");
f("чай"); //кофе и чай
f("молоко"); //кофе и молоко
```

3. Замыкания хранят внутри себя ссылки на внешние переменные, они способны как читать, так и обновлять эти свои переменные.

Функция box(), значения которого могут быть считаны и обновлены. Листинг 2.30

```
function box(){
  var val =  undefined;
  return {
    set: function(newVal){val = newVal;},
    get: function(){return val;},
    type: function(){return typeof val;}
  };
}
var b = box();
b.type(); //undefined
b.set(98.6)
b.get(); // 98.6
b.type(); // "number"
```

В этом примере создается объект, в котором содержится три замыкания – это его свойства: set (определение или переопределение переменной), get (вернуть переменную), type (определить тип переменной)

2.9 Подъем переменной

Объявленные переменные внутри блока неявно поднимаются на вершину той функции, в которую они заключены.

Визуальное представление подъема переменной. Листинг 2.31

```
function f(){
 //…
 //…
  {
   var x = /* … */
  }
}
 // JavaScript неявно поднимает объявление переменных на вершину
охватывающей функции. Т.е. JavaScript "видит" следующую функцию:
function f(){
 var x;
 //…
 //…
```

```
  {
   var x = /* … */
  }
}
```

Вывод: Во избежание путаницы, необходимо объявлять переменные, даже если их значение еще не извесно, в начале функции. Причем, при определении таких переменных, заранее указываем тип ожидаемой переменной:

Переменная x является элементом массива. Листинг 2.32

```
function f(){
 var x = [];
 //…
 //…
  {
   var x = /* … */
  }
}
```

2.10 Немедленно вызываемые функции-выражения

Замыкания хранят свои внешние переменные ввиде ссылок, а не в виде значений.

Задача: Передать в функцию массив. Функция должна вернуть значение того элемента, индекс которого получила вместе с массивом.

Для решения этой задачи воспользуемся приемом, известным как немедленный вызов функции выражения:

Создание вложенной функции с ее немедленным вызовом. Листинг 2.33

```
function wrap(a){
  var result = [];
  for (var i=0, n=a.length; i<n; i++){
    (function(){
      var j = i;
      result[i] = function(){ return a[j];};
     }
     )();
  }
  return result;
}
var wrapped = wrap([10,20,30]);
var f = wrapped[1];
f(); // 20
```

2.11 Метод bind()

Объекты функций могут поставляться вместе с методом bind(), который принимает объект получатель и создает функцию оболочку, вызывающую исходную функцию в качестве метода получателя.

Представим себе объект строкового буфера, хронящий строки в массиве.

Использование bind() с объектом. Листинг 2.34

```
var buffer = {
 entries: [],
 add: function(s){
  this.entries.push(s);
 }
}
var source = ["375", "-", "123456"];
source.forEach(buffer.add.bind(buffer));
console.log(buffer.entries);
```

Причем, buffer.add.bind(buffer) не преобразовывает функцию buffer.add, а создает новую. Парметр buffer является объектом-получателем с сылкой на this.

Лаконичный пример использования bind:

Лаконичный пример использования bind(). Листинг 2.35

```
function f() {
  alert(this.name);
}
var user = { name: "Вася" };
var f2 = f.bind(user);
f2(); // выполнит f с this.name = "Вася"
```

Задача. Имеется функция, ссылающаяся на this, и производит с ссылкой на this простые арифметические действия. И объект с числом (числами). Необходимо вызвать функцию методом данного объекта.

Преобразование функции в метод с помощью bind(). Листинг 2.36

```
function x(y) {
  return this.x + y
}
var o = {x:1};
var g = x.bind(o);
g(2);
```

Как видно из листинга, метод bind() полезен для связывания методов с получателями.

Однако существует еще один не менее полезный трюк использования bind().

Использование метода bind() для каррирования функций. Методика связывания функций с подмножеством ее аргументов известна как каррирование. Если часть атрибутов функции постоянные для каждой итерации, то это верная причина использования каррирования.

Каррирование функции. Листинг 2.37

```
function simpleURL(protocol, domain, path){
  return protocol + "://" + domain + "/" + path;
}
var paths = ["about", "12"];
var url = paths.map(simpleURL.bind(null, "http", "obmenka.by"));
console.log(url)
```

Первый метода bind() предоставляет значение получателя. Поскольку в simpleURL нет ссылки на this, используем null.

3. Объекты и прототипы

В JavaScript нет понятия классов, хотя он и является объектно-ориентированным языком. Разберемся, что это значит на практике.

То, что принято называть классами, это сочитание функции-конструктора и ассоциированного с ней прототипа.

3.1 Конструктор

Конструктор – это функция предназначенная для инициализации объектов. Важной особенностью использования конструктора является свойство prototype конструктора в качестве прототипа нового объекта. Роль конструктора в языке JavaScript может играть любая функция, для которой определено свойство prototype.

Реализация класса с помощью конструктора. Листинг 3.1

```
function Range(from, to){
 this.from = from; // определение свойств
 this.to = to;
}
Range.prototype = { // определение методов
 includes: function(x){
    return this.from <= x && x <= this.to;
  },
 foreach: function(f){
    for(var x= Math.ceil(this.from); x<= this.to; x++){
     f(x);
    }
  },
 toString: function(){
    return "(" + this.from + "…" + this.to + ")";
  }
};
var f = new Range(1,3);
f.includes(2) // true: число 2 входит в диапазон
f.foreach(console.log) // выведет: 1 2 3
```

Функция Range поставляется с имеющимся по умолчанию свойством prototype. Как видно из листинга, объект f получает объект сохраненный в Range.prototype.

В JavaScript имеется специальная функция для получения прототипа существующего объекта. Например, после создания объекта f мы можем провести следующую проверку:

Получение прототипа существующего объекта, использование функции getPrototypeOf(f). Листинг 3.2

```
Object.getPrototypeOf(f) === Range.prototype
```

Процесс определения класса можно свести к следующим этапам:

1. Создать функцию конструктор.
2. Определить методы экземпляров в прототипе конструктора.
3. Определить свойства в самом конструкторе.

Рассмотрим еще один способ определения классов объектов, где методы определяются как свойства функции объекта-прототипа.

Реализация класса с определением методов как свойств функции объекта-прототипа. Листинг 3.3

```
function Complex(real, imaginary) {
    this.x = real;       // Вещественная часть числа
    this.y = imaginary; // Мнимая часть числа
}
// Возвращает модуль комплексного числа. Он определяется как
расстояние
// на комплексной плоскости до числа от начала координат (0,0).

Complex.prototype.magnitude = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
};

// Возвращает комплексное число с противоположным знаком.
Complex.prototype.negative = function() {
  return new Complex(-this.x,  -this.y);
};
// Складывает данное комплексное число с заданным и возвращает
// сумму в виде нового объекта.
Complex.prototype.add = function(that) {
    return new Complex(this.x + that.x, this.y + that.y);
}
// Умножает данное комплексное число на заданное и возвращает
// произведение в виде нового объекта.
Complex.prototype.multiply = function(that) {
    return new Complex(this.x * that.x - this.y * that.y,
                       this.x * that.y + this.y * that.x);
}
// Преобразует объект Complex в строку в понятном формате.
// Вызывается, когда объект Complex используется как строка.
Complex.prototype.toString = function() {
    return "{" + this.x + "," + this.y + "}";
};
// Проверяет равенство данного комплексного числа с заданным.
Complex.prototype.equals = function(that) {
    return this.x == that.x && this.y == that.y;
}
// Возвращает вещественную часть комплексного числа.
// Эта функция вызывается, когда объект Complex рассматривается
// как числовое значение.
Complex.prototype.valueOf = function() { return this.x; }
/*
 * Третий шаг в определении класса – это определение методов
класса,
 * констант и других необходимых свойств класса как свойств са-
```

```
мой
 * функцииииконструктора (а не как свойств объектаапрототипа
 * конструктора). Обратите внимание, что методы класса не ис-
пользуют
 * ключевое слово this, они работают только со своими аргумента-
ми.
 */
// Складывает два комплексных числа и возвращает результат.
Complex.prototype.add = function (a, b) {
    return new Complex(a.x + b.x, a.y + b.y);
};
// Умножает два комплексных числа и возвращает полученное произ-
ведение.
Complex.prototype.multiply = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
                       a.x * b.y + a.y * b.x);
};
// Несколько предопределенных комплексных чисел.
// Они определяются как свойства класса, в котором могут исполь-
зоваться как "константы".
// (Хотя в JavaScript невозможно определить свойства, доступные
только для чтения.)
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

Свойство prototype автоматически определяется, как только мы обращаемся к любой функции с ключевым словом new. Однако, любой конструктор можно переписать так, чтобы он вызвался без ключевого слова new. Для этого необходимо задействовать функцию Object.create():

Конструктор, который может вызываться как с ключевым словом new, так и без него. Листинг 3.4

```
function User(name, password){
self = this instanceof User ? this: Object
.create(User.prototype);
  self.name = name;
  self.password = password;
  return self;
}
```

Храним свойства – в конструкторе, методы – в прототипе

В конструкторе не желательно хранить методы (для методов существует прототип), т.к. это приводит к разрастанию копий методов. И все же в ситуациях, когда важнее обеспечения сокрытой информации, можно воспользоваться замыканиями, а это не что иное, как методы в конструкторе.

Реализация замыканий в конструкторе для хранения закрытых данных. Листинг 3.5

```
function User(name, password){
 this.toString = function(){
  return “User ” + name;
 }
```

```
 this.checkPassword = function(password){
  return password;
 }
}
```

Храним состояние экземпляра в объекте-экземпляре

Изменяющиеся данные при совместном использовании разными объектами могут стать источником проблем, а прототипы совместно используются всеми созданными на основе их объектами. Отсюда и следует вывод: изменяемые состояния храним в объекте-экземпляре.

Отдельный массив для каждого экземпляра объекта. Листинг 3.6

```
function Tree(x){
  this.value = x;
  this.children = [];
}
Tree.prototype = {
  addChild: function(x){
   this.children.push(x);
  }
}
```

Особенности работы this

У каждой функции есть неявная связь с this, чье значение определяется при вызове функции. Следовательно, this в методе прототипе отличается от this в функции обратнгого вызова.

Рассмотрим пример, где в прототипе в функции обратного вызова используется ссылка на this:

Ссылка на this в функции callback прототипа. Листинг 3.7

```
function User(name){
  this.name = name;
}
User.prototype.read = function(){
  var lines = ['a','b','c'];
  return lines.map(function(line){
   return line+'-' + this.name;
  }, this);
}
a = new User('Вася');
a.read();
```

Обратите внимание на второй параметр функции map. Чтобы функция map явным образом увидела ссылки this, мы передаем this вторым параметром.

Но не все функции обладают такой степенью продуманности. Если функция не принимает дополнительного аргумента на понадобится способ сохранения связи с this:

Сохранение ссылки на внешнюю связь this. Листинг 3.8

```
User.prototype.read = function(){
 var self = this;
 …
}
```

3.2 Наследование

Прототип функции является обычным объектом, и поэтому существует несколько способов копирования его функциональных возможностей (методы и свойства), чтобы осуществить наследование.

Чтобы создать класс, наследующий и расширяющий прототип другого класса, можно воспользоваться Object.create():

Наследование прототипов. Листинг 3.9

```
function Actor(scene, x, y){
  this.scene = scene;
  this.x = x;
  this.y = y;
}
function MyClass(){

}
MyClass.prototype = Object.create(Actor.prototype);
```

Сейчас в классе MyClass, мы можем переопределить свойства и методы родительского класса Actor.

Чтобы определить цепочку прототипов (объект тип Actor) как человека (объект типа Person), человека (объект типа Person) - как млекопетающее и т.д., до самого объекта типа Object, лучше всего создать экземпляр одного объекта в качестве прототипа другого объекта:

Цепочка наследования прототипов. Листинг 3.10

```
SubClass.prototype = new SuperClass();
SubSubClass.prototype = new SubClass.prototype;
```

Таким образом, цепочка прототипов сохраняется.

Все элементы модели DOM наследуются от конструктора класса HTMLElement, который также можно расширять.

Добавление нового метода ко всем элементам HTML-разметки с помощью прототипа класса HTMLElement. Листинг 3.11

```
<div id="parent">
 <div id="a">АА</div>
 <div id="b">ББ</div>
</div>
```

```
<script>
 HTMLElement.prototype.remove = function(){
  if(this.parentNode)
   this.parentNode.removeChild(this);
 };
 var a = document.getElementById("a");
 a.parentNode.removeChild(a); //реализация удаления стандартным способом
 document.getElementById("b").remove() //реализация удаления с помощью нового метода

</script>
```

Ограничения

Не рекомендуется использовать следующий способ копирования прототипов: Actor.prototype() = Person.prototype(). Т.е. применять Actor непосредственно, как прототип Person. В этом случае, изменения в прототипе Actor обуславливают изменения в прототипе Person, т.к. это один и тот же объект, что может привести к непредвиденным результатам.

Объекты Object, Array, String, Number, RegExp и Function обладают свойствами прототипов, которые можно расширять. Но реализация расширений базовых классов может привести к осложнениям их функциональных возможностей. Поэтому, браться за это дело нужно, лишь тщательно взвесив все аргументы за и против.

Еще одна причина, по которой необходимо отказаться от расширения класса Object, это то, что при расширении его прототипа, все объекты получают соответствующие дополнительные свойства.

Следует отметить, что все эти ограничения можно обойти.

Код похожий на класс

Наследование в стиле похожем на классический ООП. Листинг 3.12

```
<script>
 var Person = Object.subClass({
   init: function(isDancing){
    this.dancing = isDancing;
   },
   dance: function(){
    return this.dancing;
   }
 });
 var Ninja = Person.subClass({
   init: function(){
    this._super(false) //вызов конструктора суперкласса
   },
   dance: function(){
    // здесь логика класса Ninja
   },
   swingSword: function(){
    return true;
```

```
    }
  });
  var person = new Person(true);
  person.dance();
  var ninja = new Ninja();
  ninja.dance();
  ninja.swingSword();
</script>
```

К приведенному коду необходимо сделать следующие пояснения:

- Создание нового класса осуществляется путем вызова метода subClass() из функции-конструктора. В данном случае класс Person создается как производный от класса Object, а класс Ninja - от класса Person.
- Метод init() предназначен для определения свойств класса, и играет роль конструктора.
- Метод this._super() вызывает исходные методы init() и dance() суперкласса Person.

4.Тестирование JavaScript

4.1 Модульное тестирование

Суть модульного тестирования состоит в проверке того, работает ли код блока или модуля так, как задумано или ожидается. Некоторые разработчики, которые скорее предпочтут создавать новые модули, считают написание сценариев тестирования пустой тратой времени. Но при работе с большими приложениями модульное тестирование позволяет реально экономить время, помогая отслеживать проблемы и без риска обновлять код.

Ранее модульное тестирование применялось только в языках, используемых на стороне сервера. Но возрастающая сложность подсистем доступа привела к росту потребности в написании сценариев для тестирования кода JavaScript.

Рассмотрим функцию, которую мы будем тестировать: она прибавляет число 3 к переданной переменной.

Стандартная функция для тестирования. Листинг 4.1

```
function addThreeToNumber(el){
    return el + 3;
}
```

Следующий листинг содержит соответствующий сценарий тестирования в самовыполняющейся функции.

Сценарий тестирования функции. Листинг 4.2

```
(function testAddThreeToNumber(){
    var a = 5,
        valueExpected= 8;

    if (addThreeToNumber (a) === valueExpected) {
        console.log("Годен!");
    } else {
        console.log("Не годен!");
    }
}());
```

После передачи тестируемой функции числа 5 тест проверяет, равно ли возвращаемое значение 8. Если тест прошел успешно, на консоль современного браузера выводится сообщение Годен!; в противном случае появляется сообщение Не годен! Для запуска этого теста нужно:

1) импортировать два файла сценария в страницу HTML, которая будет играть роль среды тестирования, как показано в листинге;

2) открыть (или обновить) тестируемую страницу в браузере.

4.2 QUnit

QUnit — это библиотека от разработчиков jQuery, позволяющая писать unit-тесты для кода на javascript. Для написания unit-тестов понадобятся два файла: QUnit.js и QUnit.css (скачиваем с официального сайта: http://qunitjs.com/), а также новый html документ примерно такого содержания:

Html-документ для тестирования. Листинг 4.3

```
<html>
 <head>
  <link rel="stylesheet" href="qunit.css" type="text/css"
  <script src="qunit.js"></script>
  <script src="your-code-for-testing.js"></script>
  <script type="text/javascript" src="your-tests.js"></script>
  </script>
 </head>
 <body>
  <h1 id="qunit-header">QUnit test </h1>
  <h2 id="qunit-banner"></h2>
  <h2 id="qunit-userAgent"></h2>
  <ol id="qunit-tests">
  </ol>
 </body>
</html>
```

Теперь подключаем свой код и можно писать тесты.

Протестируем функцию trim, которая удаляет пробелы и табы на концах строки. Вот её код:

Тестируемая функция trim(). Листинг 4.4

```
function trim(text) {
  return (text || "").replace(/^\s+|\s+$/g, "");
}
```

Вот так её можно протестировать:

Unit-тест. Листинг 4.5

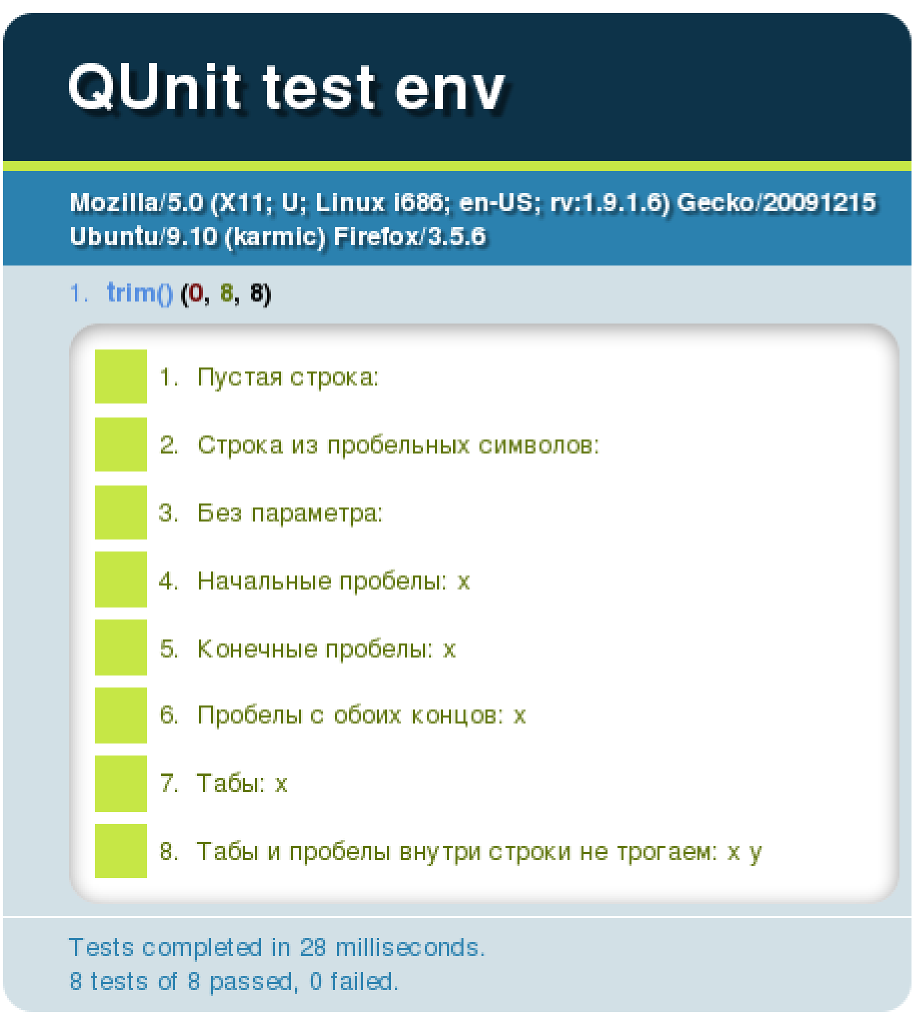
```
test('trim()', function () {
  equals(trim(''), '', 'Пустая строка');
  ok(trim('   ') === '', 'Строка из пробельных символов');
  same(trim(), '', 'Без параметра');

  equals(trim(' x'), 'x', 'Начальные пробелы');
  equals(trim('x '), 'x', 'Концевые пробелы');
  equals(trim(' x '), 'x', 'Пробелы с обоих концов');
```

```
  equals(trim('    x  '), 'x', 'Табы');
  equals(trim('    x   y  '), 'x   y', 'Табы и пробелы внутри
строки не трогаем');
});
```

В первой строке вызов функции test. Первым параметром обозначаем функционал который тестируем. Последним — тестирующую функцию. Внутри этой функции производятся различные проверки. В данном случае мы проверяем соответствие результата выполнения функции и ожидаемой строки. Для проверки на соответствие используются функции:

- *equals* — проверяет равенство первых двух параметров (нестрогая проверка, только для скалярных величин)
- *ok* — истинность первого параметра
- *same* — строгая проверка на равенство первых двух параметров (проверяет также равенство двух массивов и объектов)

Последним параметром функции принимают описание тестового случая. В результате этой проверки получаем следующую картину:



Для тестирования асинхронности мы должны остановить нормальный поток управления и по окончанию теста возобновить его. Поток останавливается автоматически, либо с помощью функции stop(), а функция start() служит для возобновления потока. Вот простой пример:

Тестирование Ajax. Листинг 4.6

```
asyncTest('async', function () {
  // поток остановлен автоматически

  setTimeout(function () {
    ok(true);

    // Возобновляем конечно же вручную
    start();
  }, 500);
});
```

В следующем листинге показано, как можно запустить несколько асинхронных проверок в одном тесте.

Тестирование Ajax. Листинг 4.7

```
asyncTest('asynctest', function () {
  // Пауза
  expect(3);

  $.get(function () {
    // асинхронные проверки
    ok(true);
  });

  $.get(function () {
    // другие асинхронные проверки
    ok(true);
    ok(true);
  });

  setTimeout(function () {
    start();
  }, 2000);
});
```

Кроме фрэймворка QUnit, существует множество других сред тестирования js-кода. Можно выделить несколько, о которых следует знать: Jasmine, TestSwarm и YUI Test.

Глава II. API JavaScript

Набор функций, методов и свойств работающих с какой-то одной задачей называют api.

1. API 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 1.1

```
<!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 1.2

```
<!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 медиа-ролика) 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).

Использование атрибутов тэга <video>. Листинг 1.3

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Video Player</title>
</head>
<body>
  <section>
    <video width="720" height="400" preload controls loop post-
er="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-код проигрывателя. Листинг 1.4

```
<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>
```

Добавим стили:

Файл player.css. Листинг 1.5

```
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;
}
```

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

Функция initiate() инициализирующая переменные и прослушиватели. Листинг 1.6

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

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 1.7

```
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 1.8

```
function status(){
  if(!mmedia.ended){
    var size = parseInt(mmedia.currentTime * maxim / mmedia.duration);
    progress.style.width = size + 'px';
  }else{
    progress.style.width = '0px';
    play.innerHTML = '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 1.9

```
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 1.10

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

Управление громкостью:

Функция level(). Листинг 1.11

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

Мы создали полноценный web-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 файла с субтитрами. Листинг 1.12

```
<video id="video1" width="640" height="360" preload="none" controls>
 <track src="subtitles.vtt" srclang="en" kind="subtitles" la-
bel="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 файла субтитров. Листинг 1.13

```
WEBVTT

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

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

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!
```

В субтитрах мы можем использовать html-код. А также использовать css-стили

Стили. Листинг 1.14

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

API Audio

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:

HTML-код для аудио-проигрывателя. Листинг 1.15

```
<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>
```

2. 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.

Использование элемента canvas. Листинг 2.1

```
<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().

Подготовка холста к рисованию. Листинг 2.2

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

Прямоугольник

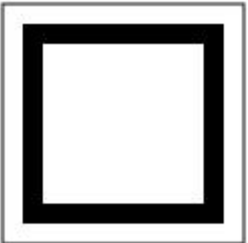
Для рисования прямоугольников доступны следующие методы:

- **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:

Рисование прямоугольника. Листинг 2.3

```
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);
```



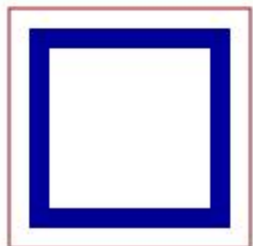
Цвет

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.

Цвет. Листинг 2.4

```
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, создающих 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.

Линейный градиент. Листинг 2.5

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

  var grad = canvas.createLinearGradient(0, 0, 500, 500);
  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);
```

Пути

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.

Инициализация начала и конца пути. Листинг 2.6

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

  canvas.beginPath();
  // Здесь пути
  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:

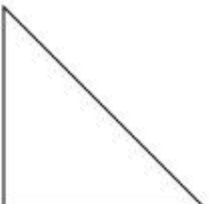
- **moveTo**(x, y). Перемещает кончик пера в указанную позицию.
- **lineTo**(x, y). Создает отрезок между двумя точками: текущей позицией (например, определенной с помощью метода moveTo) и точкой с координатами x и y.
- **rect**(x, y, width, height). Создает прямоугольник, который не сразу визуализируется 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:

Треугольник. Листинг 2.7

```
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.closePath();
  canvas.stroke();

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



Чтобы нарисовать залитый цветом треугольник, необходимо вместо метода stroke() использовать метод fill().

Треугольник залитый цветом. Листинг 2.8

```
canvas.fill();
```

Маска

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

Маска. Листинг 2.9

```
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.clip();
```

```
  canvas.beginPath();
  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.

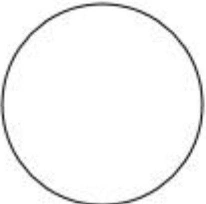
Дуги

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°.

Круг. Листинг 2.10

```
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 2.11

```
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);
```

Кривые

Метод quadraticCurveTo() предназначен для создания квадратичной кривой Безье, а метод bezieCurveTo() – для рисования кубической кривой Бизье.

Кривые бизье. Листинг 2.12

```
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.

Смайлик. Листинг 2.13

```
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 на холст 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.

Методы текста:

- **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.

Текст. Листинг 2.14

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

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

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.

Использование measureText(). Листинг 2.15

```
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 2.16

```
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("my message", 100, 100);
}
addEventListener("load", initiate);
```

my message

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.

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

Трансформации. Листинг 2.17

```
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.

TEST

TEST

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().

Матрица трансформации. Листинг 2.18

```
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);
```

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.

Отмена предыдущих трансформаций. Листинг 2.19

```
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);
```

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.

Пример комбинирования фигур, свойство globalCompositeOperation. Листинг 2.20

```
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 изображений. Листинг 2.21

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

  var img = document.createElement('img');
  img.setAttribute('src', 'http://obmenka.by/media/img/we.jpg');
  img.addEventListener("load", function(){
    canvas.drawImage(img, 20, 20);
  });
}
addEventListener("load", initiate);
```

Изменение размера изроображения

Вставка изображений. Листинг 2.2.22

```
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, 0, 0, elem.width, elem.height);
  });
}
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 и изменения размеров. Листинг 2.23

```
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, 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 в виде 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

Негатив изображения. Листинг 2.24

```
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 принимть одно из четырех значений: repeat, repeat-x, repeat-y или no-repeat.

Узоры на холсте. Листинг 2.25

```
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

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 на холсте. Листинг 2.26

```
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();

  x = x+70

  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.

Игра

Html-код игры. Листинг 2.27

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Video Game</title>
  <style>
    body{
      text-align: center;
    }
    #canvasbox{
      margin: 100px auto;
    }
    #canvas{
      border: 1px solid #999999;
    }
  </style>
  <script src="videogame.js"></script>
</head>
<body>
  <section id="canvasbox">
    <canvas id="canvas" width="600" height="400"></canvas>
  </section>
</body>
</html>
```

JavaScript-код:

JavaScrip-код игры. Листинг 2.2.28

```
var mygame = {
  canvas: {
    ctx: '',
    offsetx: 0,
    offsety: 0
  },
  ship:{
    x: 300,
    y: 200,
```

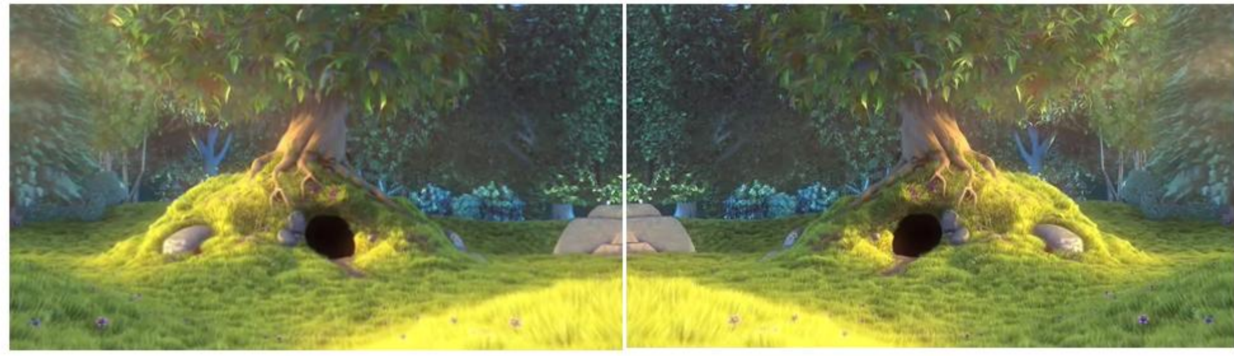
```
    movex: 0,
    movey: 0,
    speed: 1
  },
  initiate: function(){
    var elem = document.getElementById('canvas');
    mygame.canvas.ctx = elem.getContext('2d');
    mygame.canvas.offsetx = elem.offsetLeft;
    mygame.canvas.offsety = elem.offsetTop;
    document.addEventListener('click', function(e){
mygame.control(e);});
    mygame.loop();
  },
  loop: function(){
    if(mygame.ship.speed){
      mygame.process();
      mygame.detect();
      mygame.draw();
      webkitRequestAnimationFrame(function(){ mygame.loop() });
    }else{
      mygame.canvas.ctx.font = "bold 36px verdana, sans-serif";
      mygame.canvas.ctx.fillText('GAME OVER', 182, 210);
    }
  },
  control: function(e){
    var distancex = e.clientX - (mygame.canvas.offsetx +
mygame.ship.x);
    var distancey = e.clientY - (mygame.canvas.offsety +
mygame.ship.y);
    var ang = Math.atan2(distancex, distancey);

    mygame.ship.movex = Math.sin(ang);
    mygame.ship.movey = Math.cos(ang);
    mygame.ship.speed += 1;
  },
  draw: function(){
    mygame.canvas.ctx.clearRect(0, 0, 600, 400);
    mygame.canvas.ctx.beginPath();
    mygame.canvas.ctx.arc(mygame.ship.x, mygame.ship.y, 20, 0,
Math.PI/180*360, false);
    mygame.canvas.ctx.fill();
  },
  process: function(){
    mygame.ship.x += mygame.ship.movex * mygame.ship.speed;
    mygame.ship.y += mygame.ship.movey * mygame.ship.speed;
  },
  detect: function(){
    if(mygame.ship.x < 0 || mygame.ship.x > 600 || mygame.ship.y < 0
|| mygame.ship.y > 400){
      mygame.ship.speed = 0;
    }
  }
};
addEventListener('load', function(){ mygame.initiate(); });
```

Видео

Возможности canvas не ограничиваются рисованием. Данный API позволяет обрабатывать видео на лету.

Обработка видео. Листинг 2.29

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Video on Canvas</title>
  <style>
    section{
      float: left;
    }
  </style>
  <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>
```



3. API перетаскивания

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.

Рассмотрим пример, в котором можно перетаскивать один элемент в другой.

HTML-документ для реализации перетаскиваний. Листинг 3.1

```
<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>
```

JavaScript-код для перетаскивания (файл dragdrop.js):

dragdrop.js. Листинг 3.2

```
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 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.

Управление всем процессом перетаскивания. Листинг 3.3

```
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.

HTML-код с дополнительными источниками. Листинг 3.4

```
 <!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 допустимых источников при перетаскивании. Листинг 3.5

```
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 сообщение об ошибке.

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 set-DragImage().

Рассмотрим js-код приложения

Использование элемента <canvas> в качестве элемента приемника. Листинг 3.6

```
 <!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, запоминающего место освобождения элемента-источника в целевом:

JavaScript-код. Листинг 3.7

```
  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 3.8

```
 <!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 3.9

```
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);
```

4. API форм

API форм оговаривает набор свойств javaScript, с помощью которых можно определить корректность вводимых в форму значений, а также предоставляет набор дополнительных возможностей по работе над формой.

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">
      <label for="mytext">Textarea: </label>
      <textarea name="mytext" rows="5" cols="30"></textarea>
      <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 email. Листинг 4.2

```
<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 4.3

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

Тип данных 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 для ввода url. Листинг 4.4

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

Тип данных tel

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

Тип данных для ввода номера телефона. Листинг 4.5

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

Тип данных number

Числовой тип данных содержит атрибуты для максимального допустимого числа, минимального и шага. В данный тип данных пользователь уже сможет

ввести только число в заданном диапазоне и заданным шагом.

Числовой тип данных. Листинг 4.6

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

Тип данных range

Для создания ползунка можно воспользоваться типом данных range.

Ползунок. Листинг 4.7

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

Тип данных даты и времени

Для работы с датой существует аж пять типов данных. Рассмотрим их.

Типы данных даты и времени. Листинг 4.8

```
<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 данных color

Для ввода цвета по номеру существует тип color.

Тип данных для ввода цвета. Листинг 4.9

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

Индикатор прогресса, progress

Индикатором прогресса может служить тэг progress.

Индикатор прогресса. Листинг 4.10

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

Шкала загрузки, meter

Применяется для отображения хода загрузки, либо для измерения данных в пределах одного диапазона.

Шкала загрузки. Листинг 4.11

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

Подсказки ввода, datalist

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

Подсказки ввода . Листинг 4.12

```
<datalist id="mydata">
       <option value="123123123" label="Phone 1">
       <option value="456456456" label="Phone 2">
</datalist>
<input type="search" name="mysearch" list="mydata" autocom-
plete="off">
```

Атрибут placeholder

Подсказки, которые пропадают при фиксации на элементе формы, делаются с помощью атрибута placeholder

Атрибут placeholder. Листинг 4.13

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

Выключение автозаполнения

Для выключения автозаполнения форм можно воспользоваться атрибутом autocomplete со значением off. Данный атрибут можно применять для любых типов форм.

Выключеня автозаполнения для элемента поисковой строки. Листинг 4.14

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

Отмена валидации

Для отправки формы без валидации, можно воспользоваться атрибутом formnovalidate:

Отправка формы без валидации. Листинг 4.15

```
<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>
```

Обязательное поле

Для создания элемента формы, обязательного для заполнения, используем атрибут required, который возможно добавлять для любого элемента формы.

Объявление элемента формы обязательным для заполнения. Листинг 4.16

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

Множественные значения

Для добавления множественных значений, существует атрибут multiple.

Возможность добавления множества значений. Листинг 4.17

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

Атрибут autofocus

Данный атрибут применяется для фокусировки курсора на текущем элементе формы.

Атрибут autofocus. Листинг 4.18

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

Шаблон регулярного выражения

Для валидирования элемента формы по заданному регулярному выражению, можно воспользоваться атрибутом pattern

Использование регулярного выражения. Листинг 4.19

```
<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:

Элемент формы за пределами формы. Листинг 4.20

```
 <!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>
```

Псевдоклассы :valid и :invalid

В файле стилей для валидных элементов формы используются псевдоклассы :valid, для невалидных :invalid.

Псевдоклассы :valid и :invalid. Листинг 4.21

```
:valid{
  background: lightgreen;
}
:invalid{
  background: red;
}
```

Псевдоклассы :required и :optional

:required применяется для обязательных для заполнения типов данных;

:optional – для необязательных.

Применение псевдоклассов :required и :optional. Листинг 4.22

```
:required{
  border: 2px solid #990000;
}
:optional{
  border: 2px solid #009999;
}
```

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

Псевдоклассы :in-rage и :out-of-range. Листинг 4.23

```
:in-range{
  background: #EEEEFF;
}
:out-of-range{
  background: #FFEEEE;
}
```

Редактирование элементов с помощью атрибута contentEditable

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

Создание редактируемой области. Листинг 4.24

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

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 заполнения пользователь подтверждает введенные данные, и отредактированный документ отправляется на сервер.

Специализированная проверка

Следует сделать важное замечание: проверка пользовательских значений на стороне javaScript не является альтернативой серверной проверки (напрмер, на стороне php). Серверная проверка данных обязательна. Пользователю, который знает firebug firefox-а или любой другой web-инспектор других браузеров, не составит труда отключить любую браузерную проверку.

Спецификация HTML5 оговаривает набор свойств javaScript, с помощью которых можно определить корректность вводимых в значений. Одним из

самых полезных из них является метод setCustomValidity().

Рассмотрим пример с двумя элементами формами, один из которых (любой), обязателен для заполнения:

Обработка пользовательских значений с помощью JavaScript. Листинг 4.25

```
<!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>
```

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

Создание собственной системы проверки ошибок

Собственная система проверки ошибок. Листинг 4.26

```
<!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>
```

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().

5. API геолокации

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().

HTML-код для определения геолокационных данных. Листинг 5.1

```
<!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.

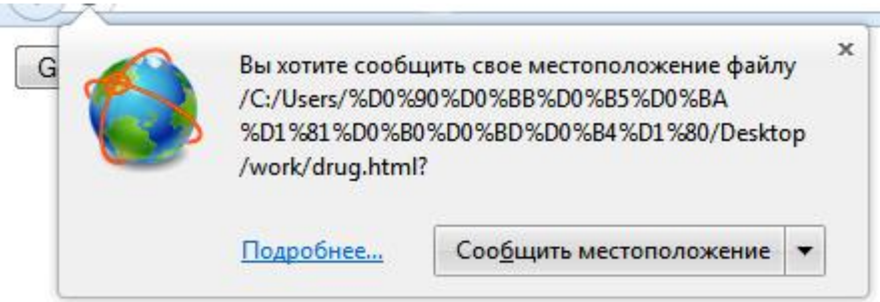
У объекта Position есть следующие атрибуты:

- **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.

JavaScript-код, получение информации о местоположении пользователя. Листинг 5.2

```
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 5.3

```
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).

Использование вспомогательных параметров. Листинг 5.4

```
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, geo-
config);
}
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, независимо 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)

Статические карты

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.

Вывод карты на экран. Листинг 5.5

```
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);
```

Динамические карты

Для включения динамических карт можно воспользоваться API карт Google или Yandex.

JavaScript API Google Карт (версия 3)

Большинство приложений, работающих с API Google Карт, должны загружать этот интерфейс с помощью ключа API (кроме приложений, работающих на localhost).

Для создания ключа API, перейдите на страницу консоли интерфейсов API по адресу:

https://code.google.com/apis/console

и войдите с использованием своего аккаунта Google.

Далее необходимо активировать API Google-карт. Ключ для работы с картами готов.

С примерами работы с картами, а также с подробной документацией по данному API можно познакомиться на странице:

https://developers.google.com/maps/documentation/javascript/tutorial

Службы маршрутов

С помощью объекта **DirectionsService** (API карт Google) 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. 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 <div> (например, "Поверните направо для въезда на Троицкиё мост").

Служба маршрутов работает со следующими типами транспортных средств (свойство VehicleType):

- VehicleType.RAIL   Железнодорожный транспорт.
- VehicleType.METRO_RAIL Узкоколейный городской транспорт.
- VehicleType.SUBWAY   Подземный узкоколейный транспорт.
- VehicleType.TRAM   Надземный узкоколейный транспорт.
- VehicleType.MONORAIL   Монорельсовый транспорт.
- VehicleType.HEAVY_RAIL Ширококолейный городской транспорт.
- VehicleType.COMMUTER_TRAIN   Электричка.
- VehicleType.HIGH_SPEED_TRAIN   Скоростной поезд.
- VehicleType.BUS   Автобус.
- VehicleType.INTERCITY_BUS   Междугородний автобус.
- VehicleType.TROLLEYBUS Троллейбус.
- VehicleType.SHARE_TAXI   Маршрутное такси, представляющее собой тип автобуса, в котором допускается посадка и высадка пассажиров в любой точке маршрута.
- VehicleType.FERRY   Паром.
- VehicleType.CABLE_CAR   Канатный транспорт, обычно наземный. Подвесная канатная дорога может иметь тип VehicleType.GONDOLA_LIFT.
- VehicleType.GONDOLA_LIFT   Подвесная канатная дорога.
- VehicleType.FUNICULAR   Фуникулер. Обычно состоит из двух кабин, каждая из которых служит противовесом для другой.
- VehicleType.OTHER   Транспортные средства любых других типов.

Подробную информацию об использовании служб маршрутов можно получить по ссылке: https://developers.google.com/maps/documentation/javascript/directions?hl=ru

6. 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-код для хранения данных с помощью API хранилищ. Листинг 6.1

```
<!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.

JavaScript-код, сохранение и извлечение данных. Листинг 6.2

```
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

js-код, альтернативный способ работы с хранилищами. Листинг 6.3

```
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.

js-код, альтернативный способ работы с хранилищами. Листинг 6.4

```
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 слова
- **clear**(). Очищает пространство хранилища. Удаляются все находящиеся в нем элементы.

Удаление данных. Листинг 6.5

```
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" on-
click="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 6.6

```
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 6.7

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

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, будет 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 6.8

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

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

Глава III. Node.js

1. Основы

Node или Node.js — серверная платформа, использующая язык программирования JavaScript. Можно также использовать CoffeScript, что, впрочем, является модификацией 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.

Node.js прослушивает конкретные события. И когда это событие происходит, соответствующим образом реагирует на него. Особенностью данной технологии является асинхронный ввод-вывод, управляемый событиями. Node.js идеально подходит для событийной логики, но не предназначен для сложных вычислений на стороне сервера.

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).

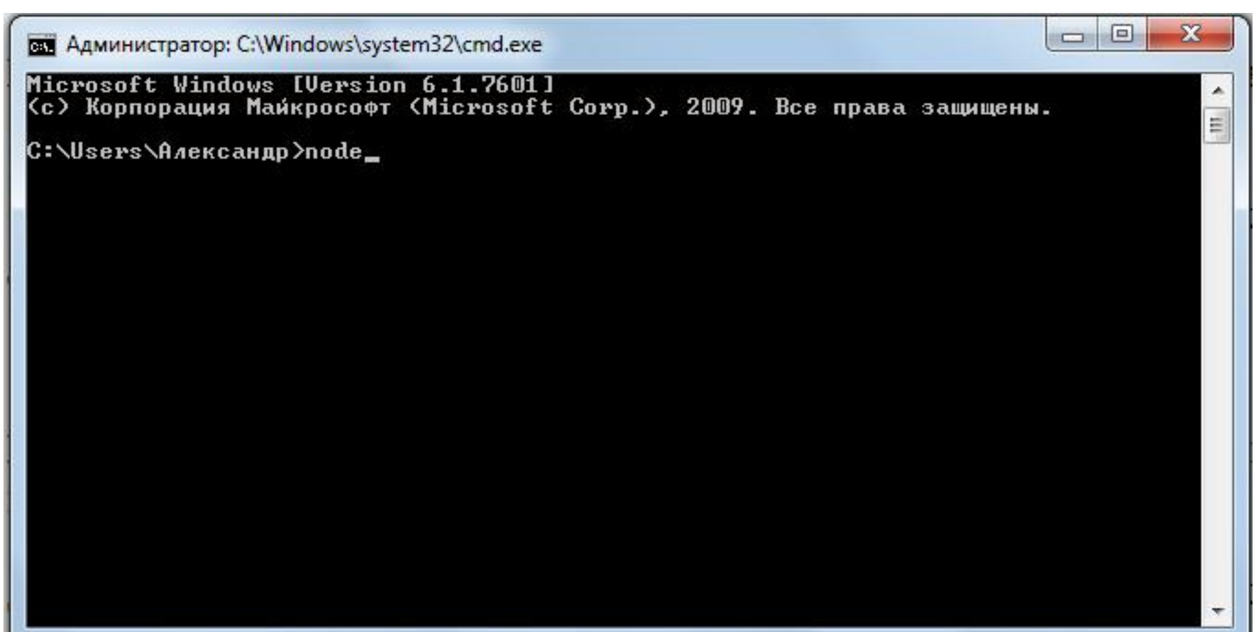
Режим REPL

REPL – это режим командной консоли для Node. REPL – встроенный компонент Node.js.

REPL позволяет проверять JavaScript-код перед его включением в файлы, и создавать приложения в интерактивном режиме, сохраняя результаты после завершения. Работа в режиме REPL схожа с работой в других средах редактирования.

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.

```
Администратор: C:\Windows\system32\cmd.exe - node
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\Александр>node
> console.log('ПРИВЕТ МИР!');
ПРИВЕТ МИР!
undefined
> _
```

Обратите внимание на ответ консоли “undefined”. Данное слово будет добавляться к каждому ответу консоли.

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

Администратор: C:\Windows\system32\cmd.exe - node

```
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\Александр>node
> .load Documents/My Web Sites/Пустой сайт/test.js
```

Для усовершенствования строкового редактирования, изменения цвета REPL и надежного сохранения истории ввода команд, либо для того, чтобы избавиться от надоедливого слова “undefined”, 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);
```

Благодаря режиму REPL, мы можем настроить вывод ответов консоли так, как нам будет удобно для просмотра: изменить цвет, размер букв, стили.

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 портов и создания серверов, дочерние процессы, система доменных имен, модули для тестирования и форматирования, объектное наследование, события, работа с файлами.

Сконцентрируемся на ключевых моментах ядра node, и рассмотрим их более подробно.

2.1 Глобальные объекты

Это такие объекты, которые 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

Объект Global

Представляет собой глобальное пространство имен. Любая определяемая переменная становится свойством объекта global.

Объект global . Листинг 2.1

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

Объект Process

Методы и свойства объекта process предоставляет информацию о приложении и его среде.

Process.execPath – возвращает путь выполнения для Node-приложения

Process.version – возвращает версию Node

Process.platform – возвращает платформу сервера.

```
> console.log(process.execPath);
C:\Program Files (x86)\nodejs\node.exe
undefined
> console.log(process.version);
v0.8.20
undefined
> console.log(process.platform);
win32
undefined
>
```

Метод объекта process – **memoryUsage**, сообщают сколько памяти расходует Node-приложение.

Метод memmoryUsage объекта process. Листинг 2.2

```
console.log(process.memoryUsage);
```

Выполнив данный листинг, получим следующее:

```
> console.log(process.memoryUsage());
{ rss: 12906496, heapTotal: 6213376, heapUsed: 2412488 }
undefined
>
```

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, открыв REPL-сеанс, и введем следующий код:

Канал pipe потока stdin. Листинг 2.3

```
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 2.4

```
process.stdin.resume(); // по умолчанию поток stdin приостановлен, 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 2.5

```
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 2.6

```
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 2.7

```
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 2.8

```
console.log('сообщение');
```

2.2 Таймерные методы

К таймерным относятся следующие функции: setTimeout, clearTimeout, setInterval и clearInterval.

setTimeout()

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 2.9

```
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 2.10

```
setTimeout(myfunc, 2000, morevar);
function(morevar){
 console.log(morevar);
}
```

clearTimeout()

Функция clearTimeout сбрасывает параметры заданные функцией setTimeout.

Совместное использование setTimeout и clearTimeout. Листинг 2.11

```
setTimeout(myfunc, 2000, morevar);
function(morevar){
 console.log(morevar);
}
```

setInterval()

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 2.12

```
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();
```

2.3 Работа с файлами

Для работы с файлами node.js имеет встроенный модуль fs.

Чтение файла:

Асинхронный вызов функции readFile. Листинг 2.13

```
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 2.14

```
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 2.15

```
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 2.16

```
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.

Отражене хода выгрузки файлов в режиме реального времени

Для обработки формы воспользуемся внешним методом formidable

Выгрузка файлов в режиме реального времени. Листинг 2.17

```
 var http = require('http');
var formidable = require('formidable');
var server = http.createServer(function(req, res){
 switch(req.method){
  case 'GET':
     show(req, res);
     break;
  case 'POST':
     upload(req, res);
     break;
 }
}).listen(3000);
console.log('Ok');
function show(req, res){
 var html = ''
  + '<form method="post" action="/" enctype="multipart/form-
data">'
  + '<p><input type="text" name="name"></p>'
  + '<p><input type="file" name="file"></p>'
  + '<p><input type="submit" value="загрузить"></p>'
  + '</form>';
```

```
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  res.setHeader('Content-Length', Buffer.byteLength(html));
  res.end(html);
}
function upload(req, res){
 var form = new formidable.IncomingForm();
 form.parse(req, function(err, fields, files){
   console.log(fields);
   console.log(files);
 });
 form.on('progress', function(bytesReceived, bytesExpected){
  var percent = Math.floor(bytesReceived / bytesExpected * 100)
  console.log(percent);
 })
}
```

2.4 Службы прослушивания портов и методы создания серверов

Основная часть ядра Node предназначена для создания служб прослушивания. Существуют node-модули для создания HTTP-сервера, TCP-сервера, TLS-сервера, модуль UDP-сокета, или сокета дейтаграмм.

HTTP-сервер

Большинство стандартных web-приложений запускаются через http-протокол (HyperText Transfer Protocol – протокол передачи гипертекста).

Можно сказать, что HTTP-протокол 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 взаимодействие между процессами в пределах одной и той же системы.

Протокол TCP

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 2.19

```
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 2.20

```
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(data){
    client.write(data);
});
client.on('data', function(data){
    console.log(data);
});
client.on('close', function(){
    console.log('connection is closed');
})
```

Итак, у нас готовы два файла, один из которых 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 2.21

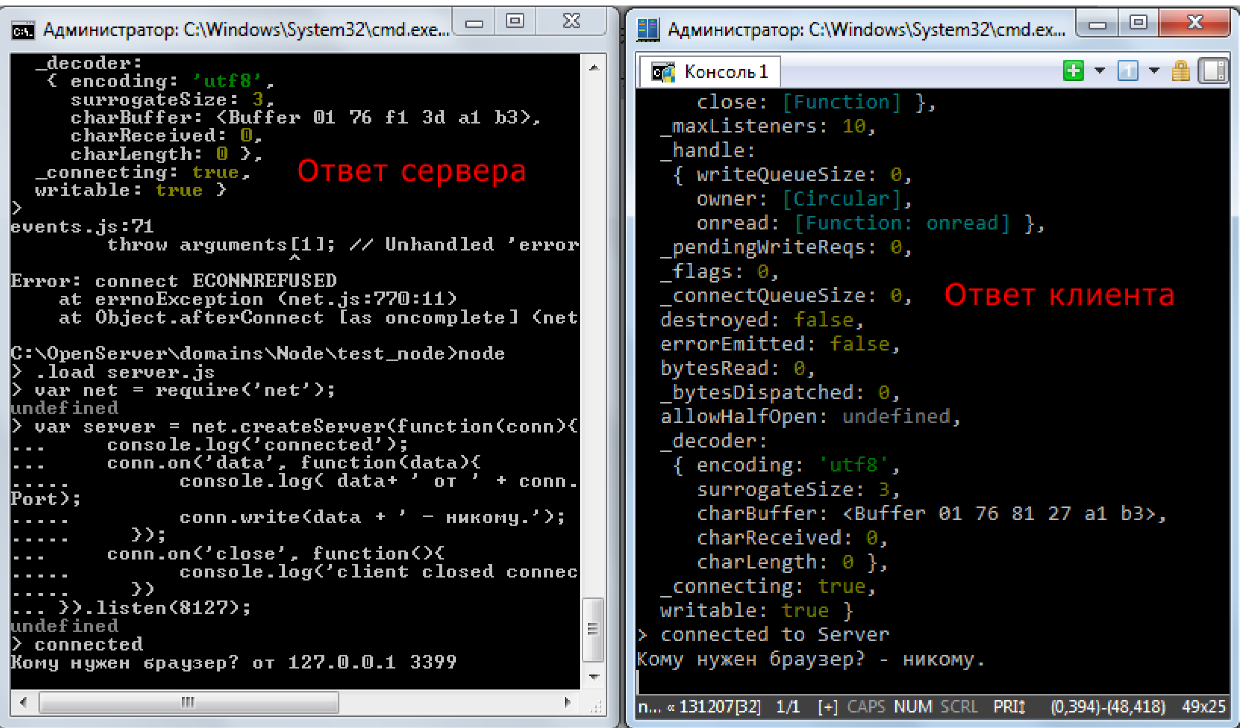
```
.load server.js
```

После чего запустим клиента.

Запуск клиента с помощью REPL. Листинг 2.22

```
.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:

Администратор: C:\Windows\System32\cmd.exe...

```
     charBuffer: <Buffer 01 76 f1 3d a1 b3>,
     charReceived: 0,
     charLength: 0 },
  _connecting: true,
  writable: true }
>
events.js:71
        throw arguments[1]; // Unhandled 'error
                        ^
Error: connect ECONNREFUSED
    at errnoException (net.js:770:11)
    at Object.afterConnect [as oncomplete] (net

C:\OpenServer\domains\Node\test_node>node
> .load server.js
> var net = require('net');
undefined
> var server = net.createServer(function(conn){
...     console.log('connected');
...     conn.on('data', function(data){
.....           console.log( data+ ' от ' + conn.
Port);
.....           conn.write(data + ' - никому.');
.....       });
...     conn.on('close', function(){
.....           console.log('client closed connec
.....       })
... }).listen(8127);
undefined
> connected
Кому нужен браузер? от 127.0.0.1 3399
connected
Привет еще раз? от 127.0.0.1 3477
client closed connection
```

Администратор: C:\Windows\System32\cmd.ex...

Консоль 1

```
  _bytesDispatched: 0,
  allowHalfOpen: undefined,
  _decoder:
   { encoding: 'utf8',
     surrogateSize: 3,
     charBuffer: <Buffer 01 76 81 27 a1 b3>,
     charReceived: 0,
     charLength: 0 },
  _connecting: true,
  writable: true }
> connected to Server
Кому нужен браузер? - никому.

(^C again to quit)
>
repl:2
client.write(data);
             ^
ReferenceError: data is not defined
    at ReadStream.<anonymous> (repl:2:14)
    at ReadStream.EventEmitter.emit (events.js:12
6:20)
    at TTY.onread (net.js:397:14)

c:\OpenServer\domains\Node\test_node>
```

n... « 131207[32] 1/1 [+] CAPS NUM SCRL PRI‡ (0,406)-(48,430) 49x25

2.6 Сокеты UDP

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.

Протокол TCP требует взаимодействия между двумя конечными точками выделенного соединения. UDP-протокол (User Datagram Protocol) этого не требует. Что означает отсутствие гарантии взаимодействия между двумя конечными точками. UDP-протокол является менее надежным, зато более быстрым в сравнении с TCP, что делает его популярным среди программистов, пишуших программы в режиме реального времени.

Для создания UDP-сокета, необходимо воспользоваться методом createSocket, передав ему тип сокета (udp4 и udp6). В отличие от TCP-сообщений, сообщения UDP должны передаваться в виде буферов, а не строк.

UDP-клиент. Листинг 2.23

```
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 2.24

```
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)
```

2.6 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 2.25

```
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);
})
```

Ключ /c, переданный cmd.exe в качестве первого аргумента, заставляет выполнить команду, а затем завершить процесс. Без этого ключа приложение не работает. Ключ /k заставит выполнять приложение и оставаться в этом состоянии, не завершив приложение.

2.7 Система доменных имен

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 2.26

```
var dns = require('dns');
dns.lookup('obmenka.by', function(err, ip){
    if(err) {
       console.log(err);
    }else{
    console.log(ip);
    }
});
```

Метод .resolve возвращает массив доменных имен по заданному IP-адресу.

Вывод массива доменов по заданному IP-адресу. Листинг 2.27

```
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 2.28

```
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' }
```

2.8 Модуль Utilites и объектное наследование

Задача модуля – обеспечение дополнительной вспомогательной функциональности.

Подключается модуль так:

Подключение модуля util. Листинг 2.29

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

Рассмотрим примеры использования модуля Utilites.

Тестирование принадлежности объекта к массиву

Принадлежность объекта к массиву. Листинг 2.30

```
util.isArray()
```

Тестирование принадлежности объекта к регулярному выражению

Принадлежность объекта к регулярному выражению. Листинг 2.31

```
util.isRegExp()
```

Форматирование строки

util.**format**() – Данный метод получает строку и значения для вставки.

Пример использования:

Метод .format(). Листинг 2.32

```
var str = util.format('My %s %d% %j', 'str', 123, {ob: 'obj'})
```

Вместо %s выведется строка 'string', вместо %d – число 123, вместо %j - объект формата JSON.

Получение строкового представления объекта

util.**inspect()** – Позволяет красиво вывести любой объект. Поведение метода напоминает поведение суперметода toString().

Просмотр объекта как строки. Листинг 2.33

```
var util = require('util');
var jsdom = require('jsdom');
console.log(util.inspect(jsdom));
```

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.

Объектное наследование

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 2.34

```
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.

2.9 Слушатели или генераторы событий

Генераторы событий инициируют события, которые могут быть прослушены слушателями событий.

В Node.js можно использовать встроенные события, либо создавать свои. События обрабатываются с помощью colback-функций слушателей. Можно определять слушателей, которые будут многократно либо разово реагировать на событие.

Для многократной реакции на событие используется метод on.

Использование метода on для многократной реакции на событие . Листинг 2.35

```
var net = require('net');
var server = net.createServer(function(socket){
  socket.on('data', function(data){
    socket.write('data');
  });
});
server.listen(3000);
```

Событие data возникает всякий раз, когда пользователю становятся доступны новые данные.

Отклик на одиночное событие осуществляется с помощью метода once. Изменим предыдущий пример кода таким образом, чтобы отравлять пользователю обратно только первый фрагмент данных.

Использование метода once для одиночного отклика. Листинг 2.36

```
var net = require('net');
var server = net.createServer(function(socket){
  socket.once('data', function(data){
    socket.write('data');
  });
});
server.listen(3000);
```

Собственные генераторы событий

Node.js позволяет создавать собственные генераторы событий. Для этого имеется специальный метод events.

Создание собственного геренатара событий. Листинг 2.37

```
var EventEmitter = require('events').EventEmitter;
var chanel = new EventEmitter();
 chanel.on('join', function(){
```

```
  console.log('welcome');
});
chanel.emit('join');
```

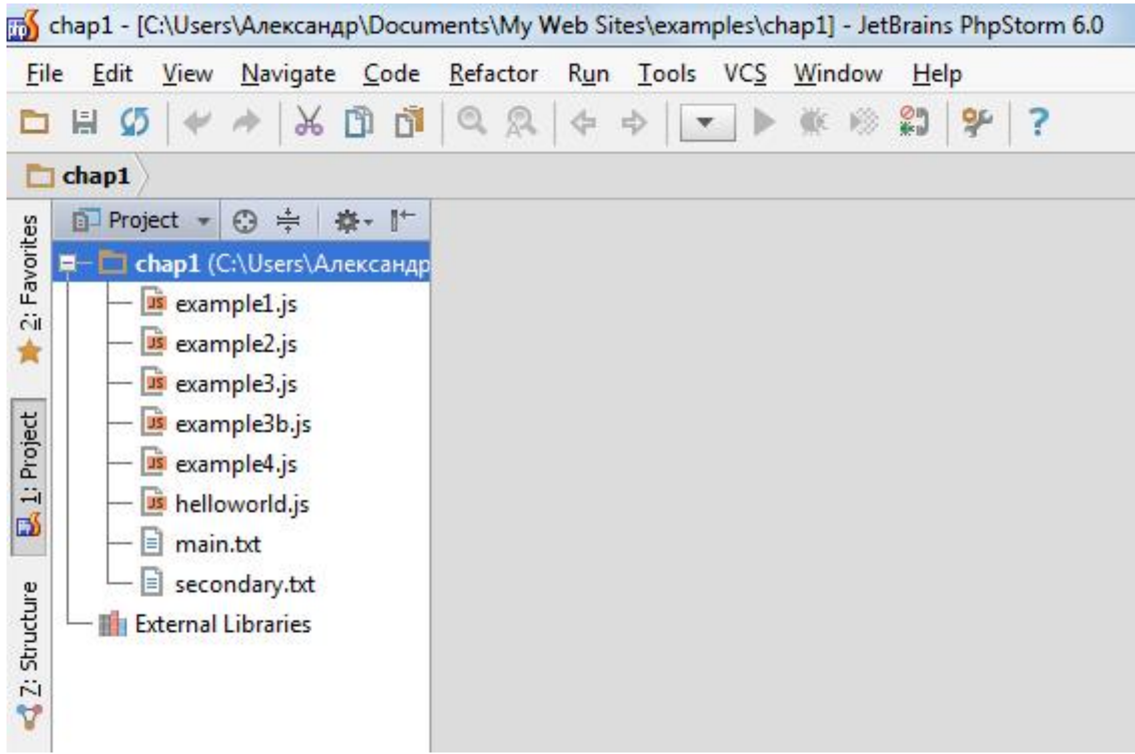
Метод emit из листинга вызывает сгенерированное событие join.

Объект EventEmitter обеспечивает Node-объектам асинхронную обработку событий.

3. Node.js и PHPStorm

Папку с node.js файлом можно просто перенести в ярлык PHPStorm.

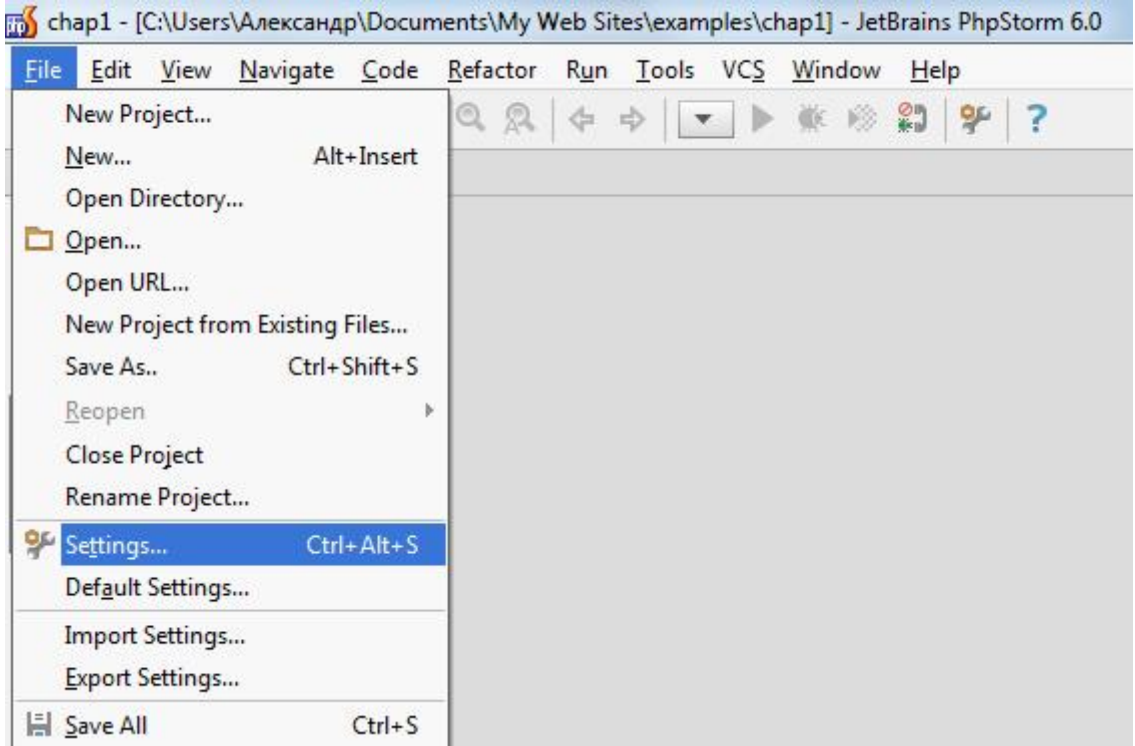
Тогда PHPStorm автоматически из содержимого папки делает проект.



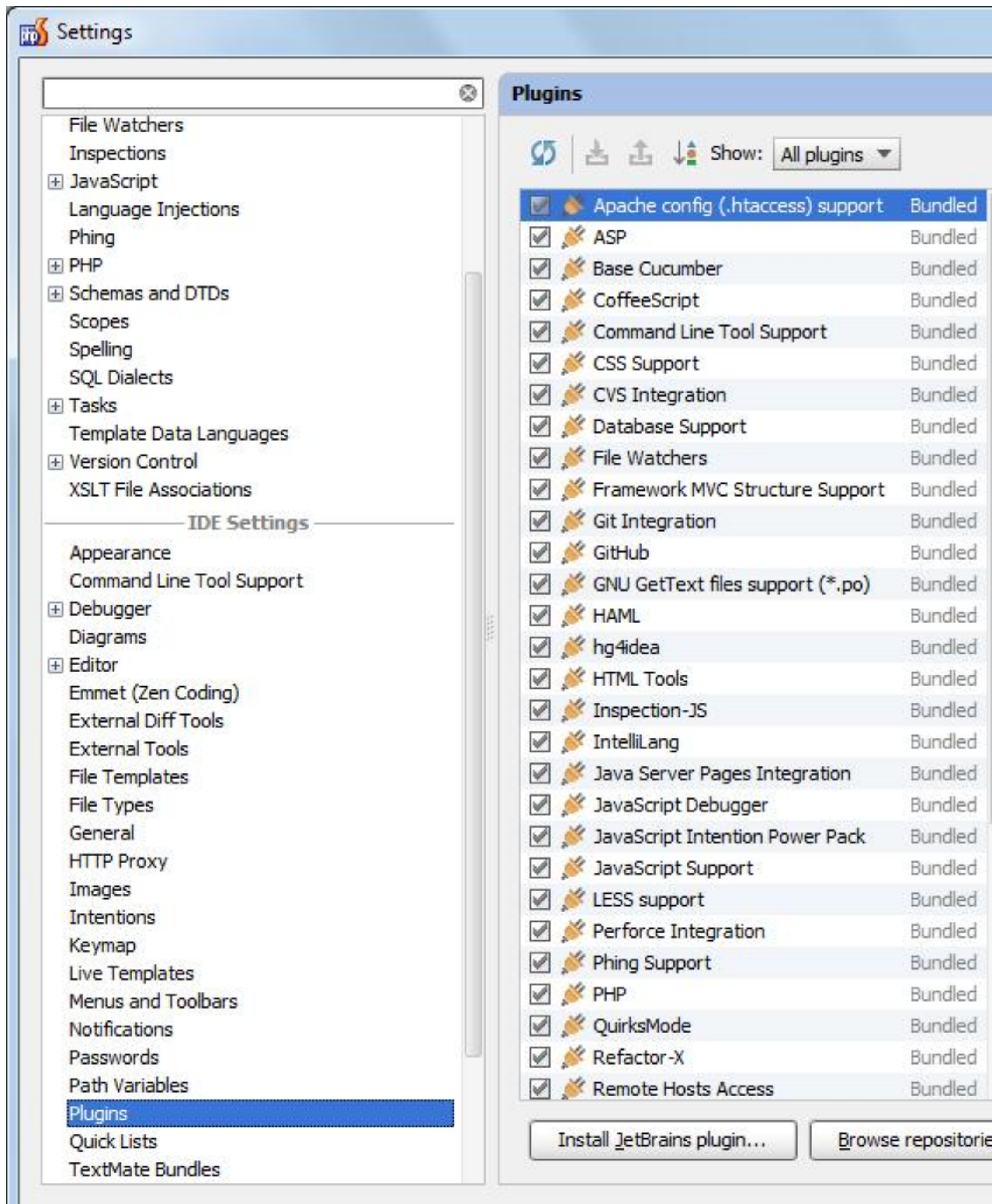
Мы уже можем использовать редактор PHPShtorm для написания и форматирования кода node.js.

Однако для удобства работы, подключим плагин Node.js.

Откроем File – Settings



Выбираем plagins



Нажимаем кнопку Instal JetBrains plugin…

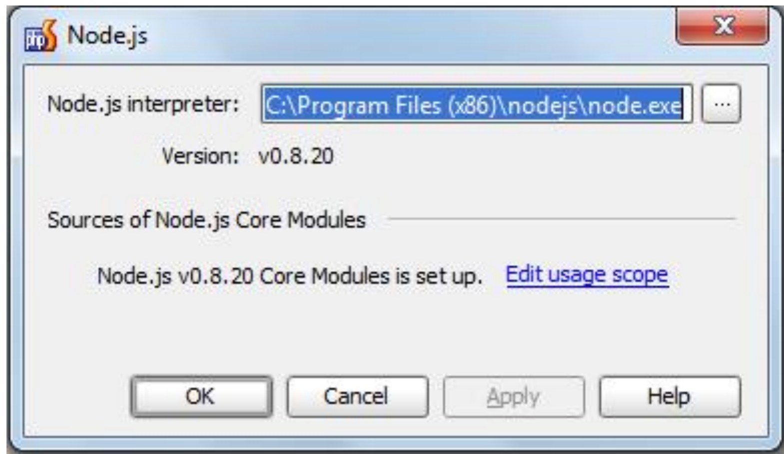
Выбираем плагин Node.js. Двойной щелчек по плагину.

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:

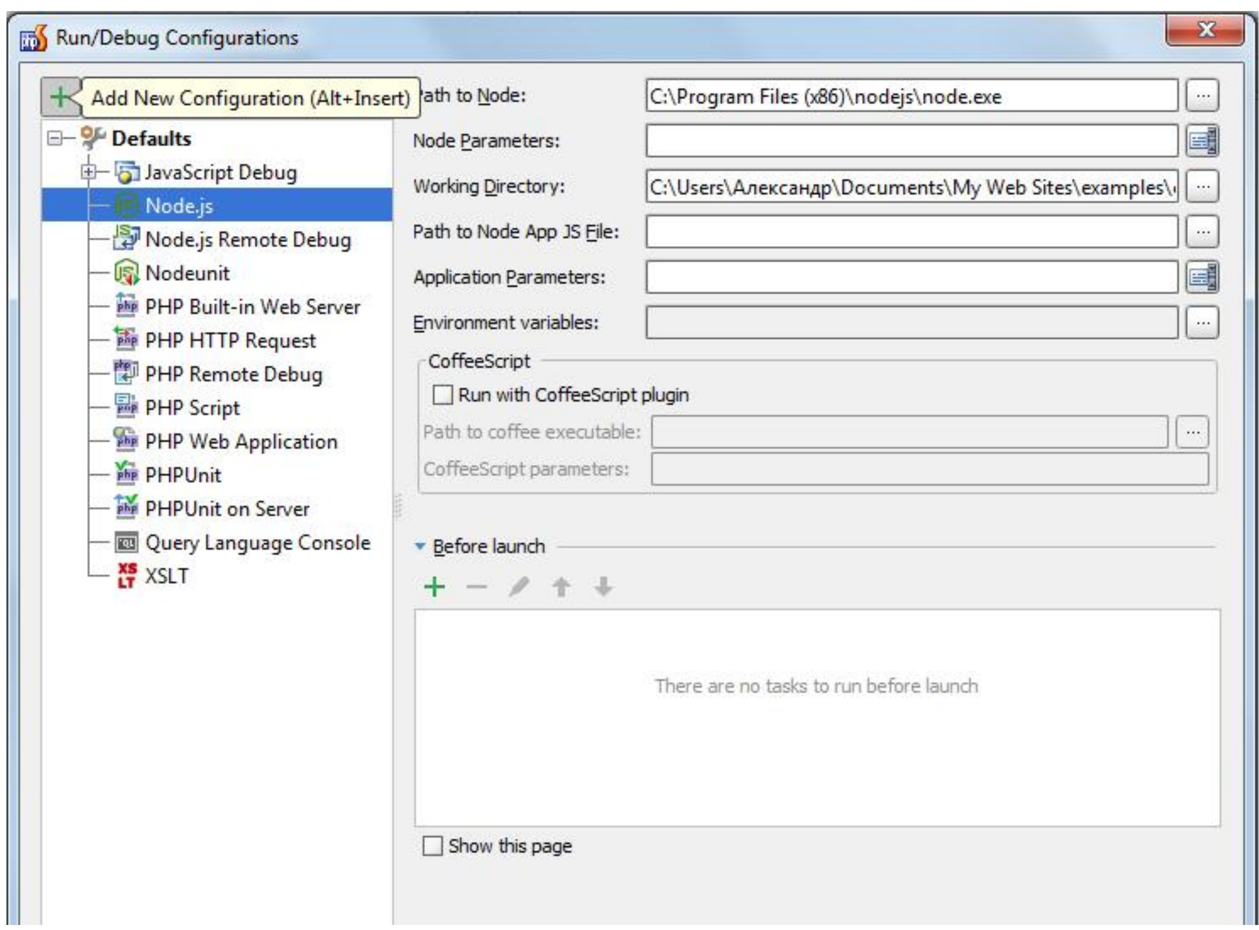


Это значит, PHPShtorm нашел платформу Node.js и готов с ней работать.

В противном случае, нужно будет указать путь к Node.js

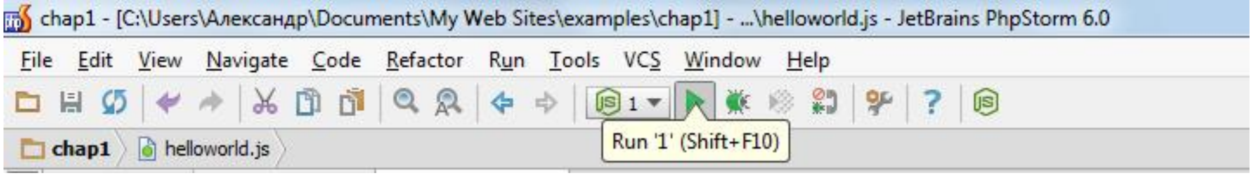
Для запуска файла Node.js нам необходимо сконфигурировать новый профиль.

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.

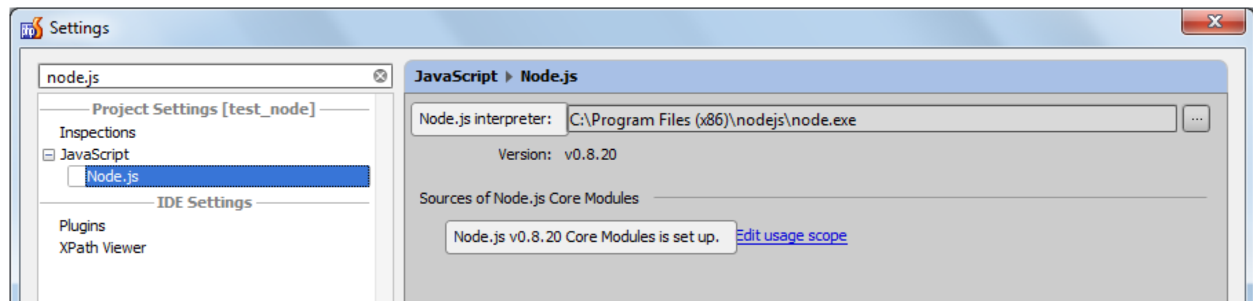


Настройка синтаксиса node.js

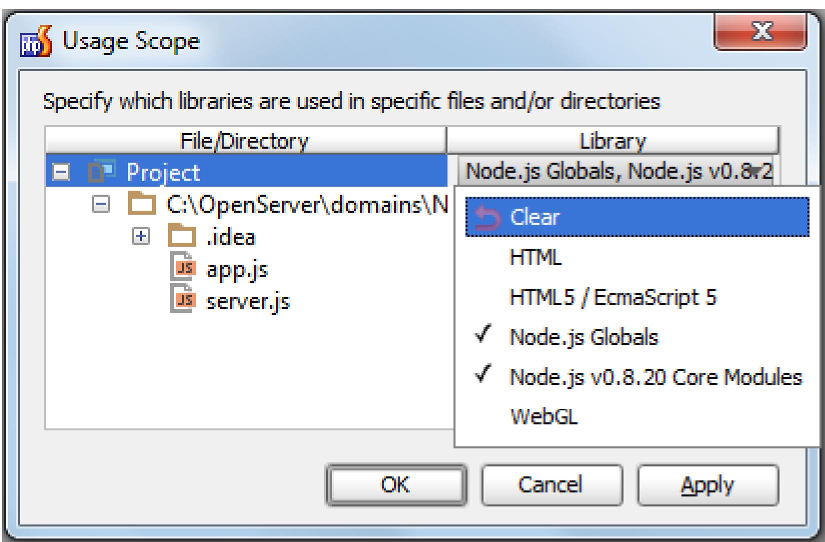
Открываем File->settings.

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. Методика асинхронного программирования

Node.js предлагает две модели управления асинхронной событийной логикой кода:

- Функции обратного вызова (collback функции)
- Слушатели или генераторы событий

Функции обратного вызова

С помощью обратных вызовов (collbacks) реализуется логика одиночных откликов.

При организации кода таким способом, метод содержит, как правило, два входящих параметра, первый – само значение метода, второй – colback-функция. Сам метод ничего не возвращает, он отдает ответ функции обратного вызова.

Функция обратного вызова. Листинг 4.1

```
fs = require('fs');
fs.readFile('index.html', function(err, info){

})
```

Функция обратного вызова тоже 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. Для ошибок используются зарезервированные переменные er либо err.

Любая collback-функция может содержать другие вложенные collback-функции.

Вложенные collback-функции . Листинг 4.2

```
 http.createServer(function(req, res){
   …
   fs.readFile('data.json', function(err, data){
     …
     fs.readFile('temp.html', function(err, data){
        …
     });
  });
}).listen(3000);
```

В большинстве случаев, такой подход себя оправдывает. Он красив и функционально понятен. Такоие же colback()-функции используются в jQuery.

Однако, чем больше уровней вложенности используется, тем более запутанным выглядит код. Опытные программисты стремятся к ограничению количества уровней вложенности обратных вызовов. Преобразовать многоуровневый код можно с помощью именных функций. В результате увеличивается количество строк кода, зато код становится более читабельным, такой код будет проще поддерживать и тестировать.

В итоге многие программисты переходят на использование внешних вспомогательных модулей управления кодом, напрмер модуль Async (раздел “Внешние модули”).

5. Разработка приложения на простом паттерне

Преобразуем код с помощью именнованных функций.

Преобразование кода с помощью именнованных функций. Листинг 5.1

```
http = require('http');
fs = require('fs');
var server = http.createServer(function(req,res){
 getData(res);
}).listen(3000);
// функция getData извлекает данные и передает выполнение кода
функции getTemp (если ошибок нет) или функции myError (если ошибки
есть)
function getData(res){
 fs.readFile('data.json', function(err, data){
   if(err){
     myError(err, res);
   }else{
    getTemp(JSON.parse(data.toString()), res)
   }
 });
}
// функция getTemp получив данные загружает шаблон и передает
управление кодом функции formatHtml.
function getTemp(titles, res){
  fs.readFile('temp.html', function(err, data){
    if(err){
      myError(err, res);
    }else{
      formatHtml(titles, data.toString(), res);
    }
  });
}
// функция formatHtml получив данные и шаблон, передает ответ кли-
енту.
function formatHtml(titles, tmp, res){
 // в файле шаблона находим символы $$, заменяем их на данные из
JSON-файла.
 var htmldata = tmp.replace('$$', titles);
 res.writeHead(200, {'Content-Type': 'text/html'});
 res.end(htmldata);
}
function myError(err, res){
  console.log(err);
  res.end('server error')
}
```

К листингу прилагается два файла: файл шаблона temp.html и файл с данными в JSON-формате data.json. Рассмотрим файл шаблона

Файл шаблона . Листинг 5.2

```
<!Doctype html>
<html>
 <head></head>
 <body>
```

```
  <div>$$</div>
 </body>
</html>
```

Количество вложений также можно уменьшить, избавившись от блоков if/else.

Перепишем код, применив идиому раннего возврата функции.

Идиома ранний возврат функции. Листинг 5.3

```
// вместо блока if/else функции getData(), вставим следующий код
if(err) return myError (err, res)
 getTemp(…);

// вместо блока if/else функции getTemp(), вставим следующий код
if(err) return myError (err, res)
 formatHtml(…);
```

6. Отладка

Простой отладчик

Для режима отладки в Node.js есть встроенный отладчик: debug.

Запуск скрипта в режмие отладки. Листинг 6.1

```
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), setBreakpoint (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 6.2

```
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 6.3

```
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:

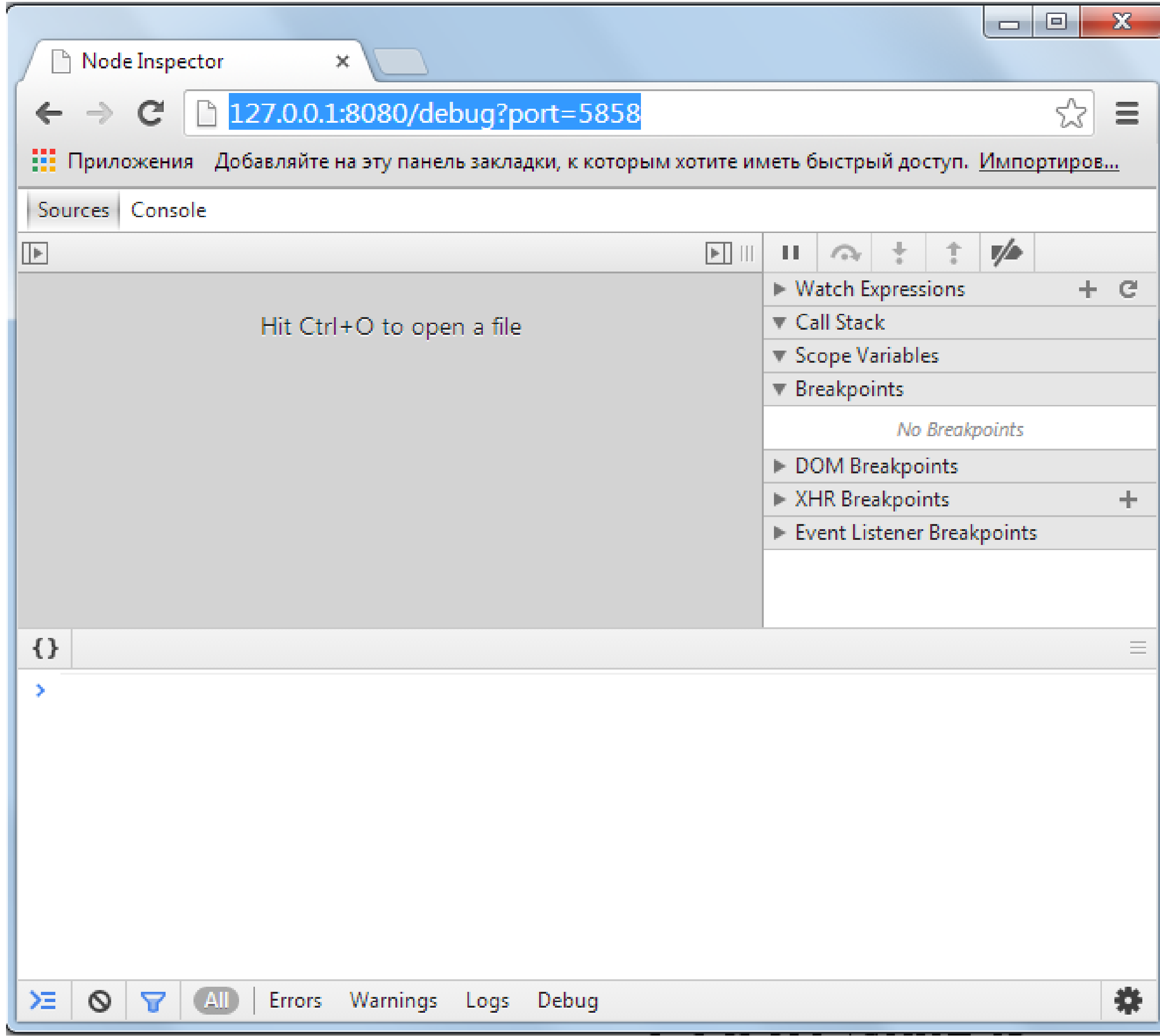
```
C:\OpenServer\domains\Node\test_node>node-inspector
Node Inspector v0.7.0
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

Мы получили приглошение перейти по ссыкле:

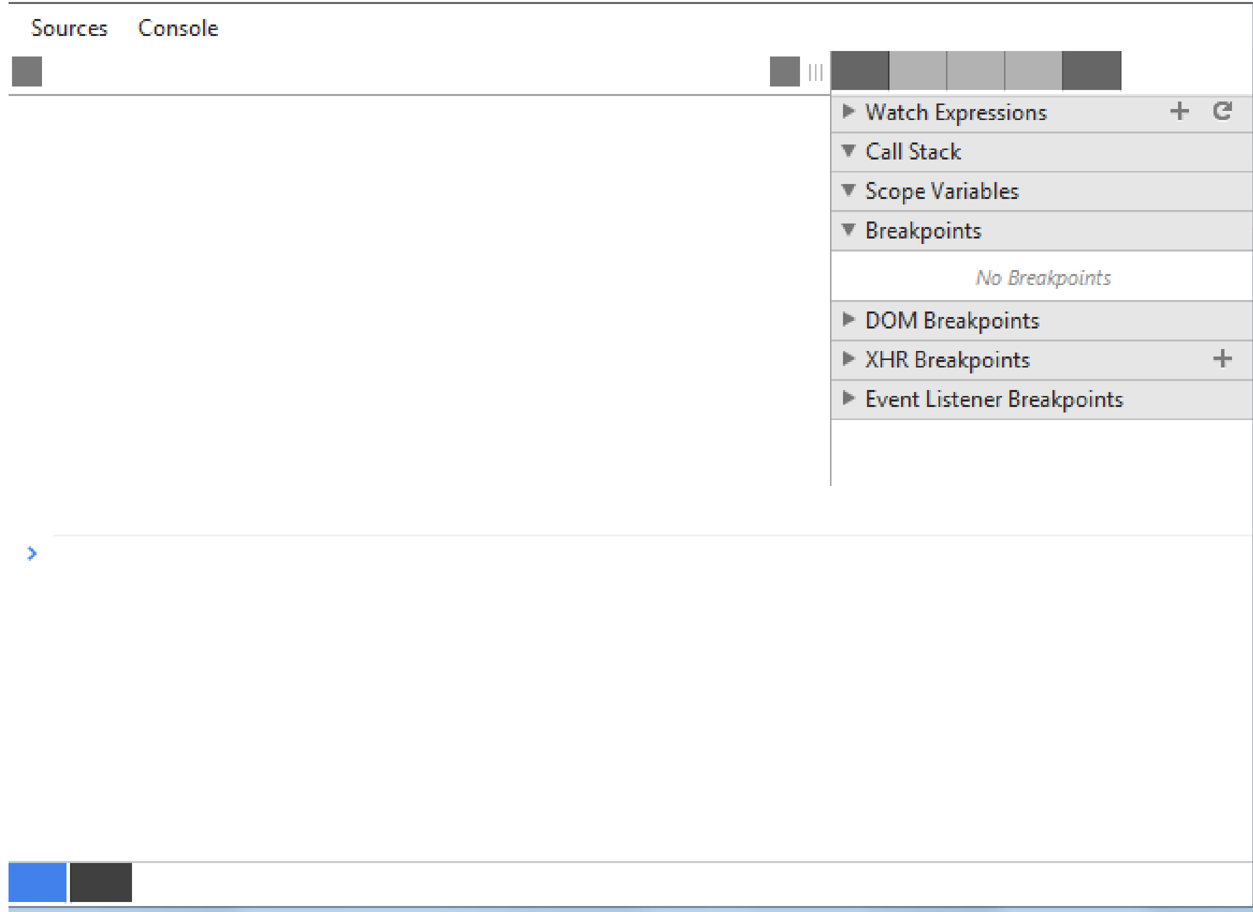
http://127.0.0.1:8080/debug?port=5858

Откроем данную страницу в браузере.

Chrome:



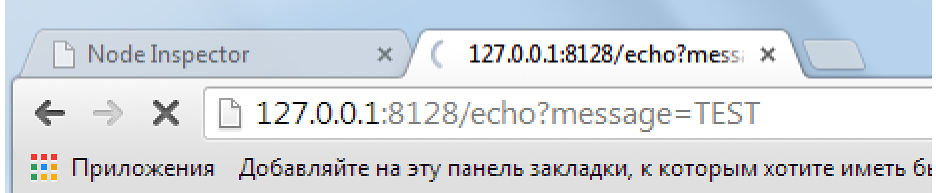
Firefox:



Далее запустим скрипт в браузере и передадим в консоль команду:

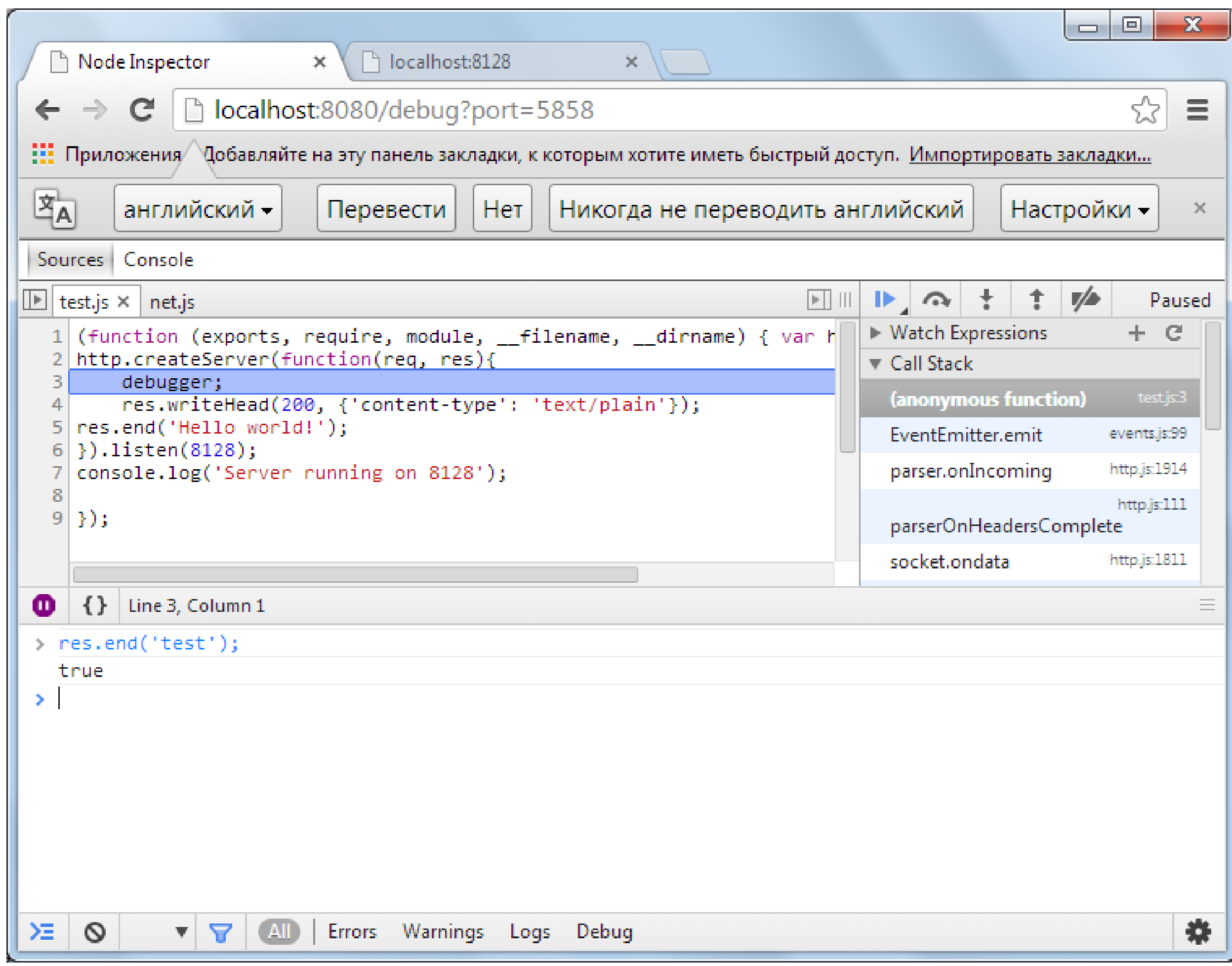
http://127.0.0.1:8128/echo?message=TEST

Если в скрипте имеется команда debugger, то скрипт останавливается на этой команде



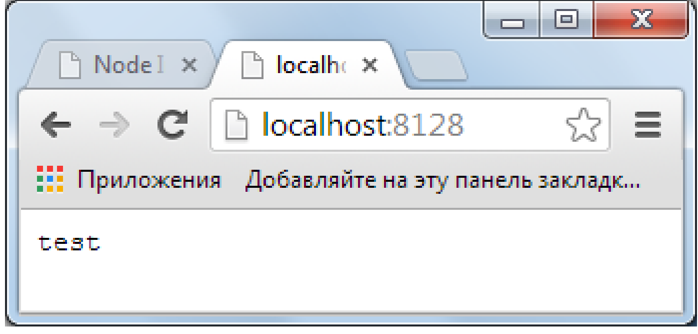
Обратите внимание на зависший браузер. Браузер завис, потому что наткнулся на команду debugger.

Если мы перейдем в Node Inspector, то увидим следующий ответ консоли:

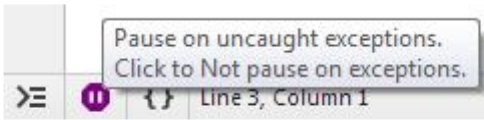


В инспекторе имеется консоль, где мы можем выполнять node-команды

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 6.4

```
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:

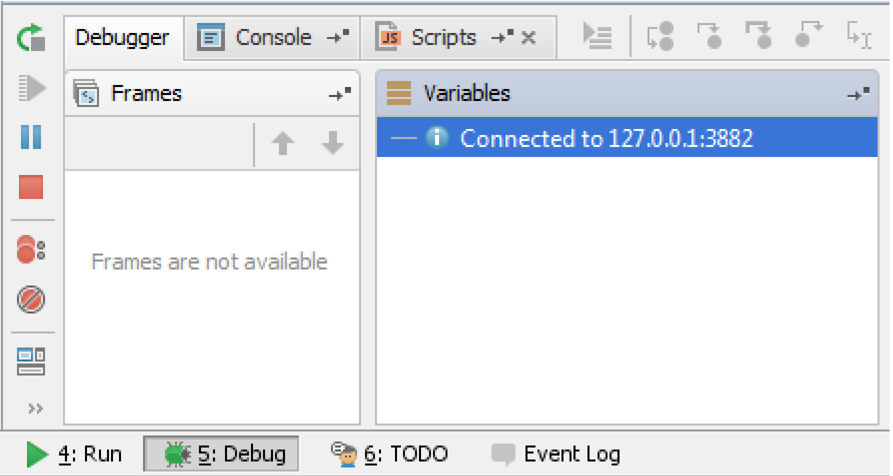
```
test.js ×
1 (function(exports, require, module, __filename, __dirname) {
2     var http = require('http');
3     http.createServer(function(req, res) {
4         debugger;
5         res.writeHead(200, {'content-type': 'text/plain'});
6         res.end('Hello world!');
7     }).listen(8128);
8     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.


Debugger
Console
Scripts
Frames
Variables
Connected to 127.0.0.1:3882
Frames are not available
4: Run
5: Debug
6: TODO
Event Log

7. Обработка ошибок

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 7.1

```
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 7.2

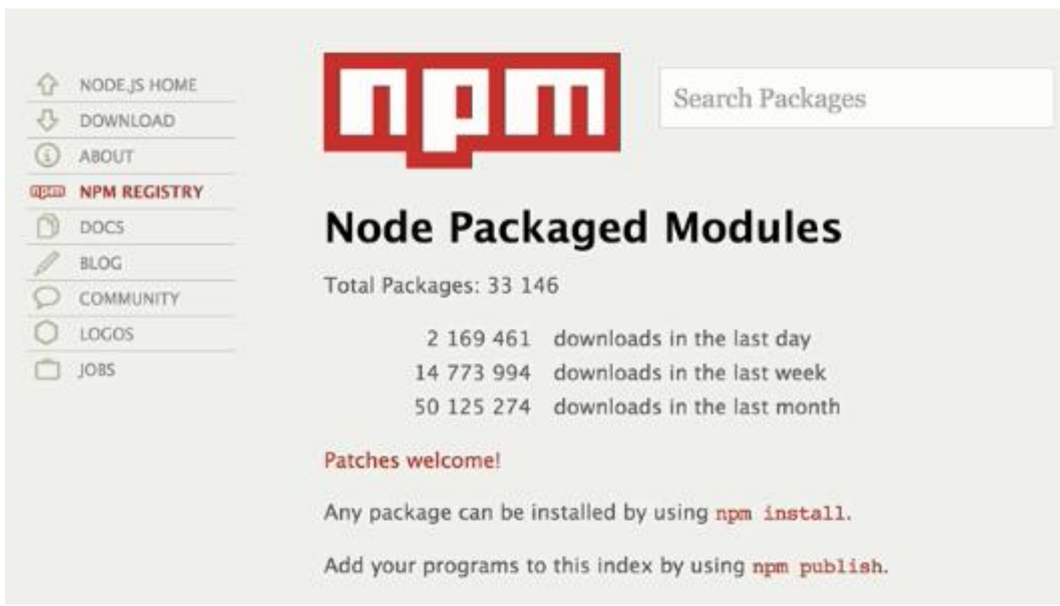
```
var util = require('util');
var phrases = {
    "Hello": "Привет",
    "world": "мир"
};

// message name stack
function PhraseError(message) {
    this.message = message;
    Error.captureStackTrace(this, PhraseError); // вывод только те-
кущего 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);
    }
}
```

8. Внешние модули

Модули устанавливаются из приложения npm, которое устанавливается по умолчанию, вместе с node.js.

Адрес сайта npm – http://npmjs.org



Все модули устанавливаются с этого сайта.

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

Установку модулей рассмотрим на примере модуля Supervisor

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 глобально.

Установка модуля supervisor. Листинг 8.1

```
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 8.2

```
supervisor server.js
```

Разработчикам собственных npm-модулей следует прочитать раздел “Developers”, который можно найти по адресу http://npmjs.org/doc/developers.html

9. Хранилище данных MongoDB

MongoDB - это система управления базами данных, «заточенная» под веб-приложения и инфраструктуру Интернета. Модель данных и стратегия их постоянного хранения спроектированы для достижения высокой пропускной способности чтения и записи и обеспечивает простую масштабируемость с автоматическим переходом на резервный ресурс, в случае отказа. Сколько бы узлов ни требовалось приложению - один или десятки, - MongoDB сумеет обеспечить поразительно высокую производительность.

От реляционных систем баз данных, например MySQL, MongoDB отличается тем, что в MongoDB данные хранятся в виде документов, а не в виде таблиц. Документы кодируются в формате BSON, который является двоичной формой JSON.

В реляционной базе данных информация о товаре, как правило, представлена в разных зависимых друг от друга таблиц. В оболочке MongoDB всю информацию можно получить в виде одного JSON-документа.

Вместо таблицы, мы используем *коллекции*, а вместо строки таблицы - *BSON-документ*.

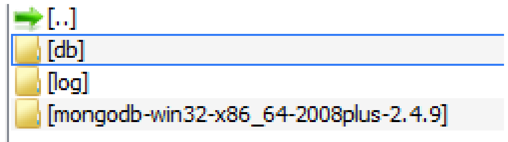
9.1 Установка Mongo

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.

Запуск mongo. Листинг 9.2

```
Mongo
```

MongoDB использует сервер localhost и порт по умолчанию 21017. Проверить запустилось ли MongoDB можно в браузере, перейдя по ссылке:

http://localhost:27017/

Если соединение с MongoDB установлено, браузер ответит:

You are trying to access MongoDB on the native driver port. For http diagnostic access, add 1000 to the port number.

По умолчанию, предлагается база данных test. Чтобы переключиться на другую базу, выполним команду use с именем базы данных:

Запуск mongo. Листинг 9.3

```
Use kurs
```

9.2 CRUD-операции

С готовой коллекцией можно делать следующие операции: вставки, извлечения, обновления, удаления.

Insert, save

Вставка данных, команда insert(). Листинг 9.4

```
db.users.insert({username:'Иван'})
```

Кроме команды insert, данные можно вставлять командой save().

Т.к. это наше первое обращение к базе данных, то сперва создастся сама база, потом коллекция с данными. Поэтому при первом обращении к несуществующей базе возможно задержка ответа сервера.

Find

Вывод данных, команда find(). Листинг 9.5

```
db.users.find()
```

Команде find можно также передать простой селектро запроса. Селектором запроса называется документ, с которым сравниваются все документы коллекции.

Вывод данных, с селектором запроса. Листинг 9.6

```
db.users.find({username:'Иван'})
```

Метод find поддерживает механизм **точечной нутации**.

Пусть потребуется найти всех пользователей, которым нравятся фильм “Белые Росы”.

Запрос будет выглядеть так:

Запрос точечной нутации. Листинг 9.7

```
db.users.find({"favorites.movies": "Белые Росы"})
```

Точка между favorites и movies означает, что нужно найти ключ favorites, который указывает на вложенный объект с ключом movies, а затем сравнить значение этого вложенного ключа с указанным в запросе. Этот запрос вернет оба документа.

Update

Для обновления нужно задать по меньшей мере два аргумента. Первый определяет, какие документы обновлять, второй - как следует модифицировать отобранные документы. Существует два способа модификации; в этом разделе мы рассмотрим направленную модификацию (targeted modification) - одну из наиболее интересных и уникальных особенностей MongoDB.

Обновление данных, команды update() и $set. Листинг 9.8

```
db.users.update({username: "Иван"}, {$set: {country: "Беларусь"}})
```

В данном примере пользователь Иван добавил новое свойство country со значением Беларусь.

Delete

Удалить страну можно так:

Обновление данных, команды update() и $unset. Листинг 9.9

```
db.users.update({username: "Иван"}, ($unset: {country: 1}})
```

Значением может быть сложная структура данных формата JSON

Обновление данных, команды update() и JSON. Листинг 9.10

```
db.users.update( {username: "Иван"}, {$set: {favorites: {
    cities: ["Минск", "Брест"],
    movies: ["Белое солнце пустыни", "Белые Росы"]
  }
 }
)
```

Обновление данных без переопределения массива. Для этого можно воспользоваться командами push или addToSet.

Команда addToSet. Листинг 9.11

```
db.users.update(
    {"favorites.movies": "Белые Росы"},
    {$addToSet: {"favorites.movies": "Свадьба в Малиновке"}},
    false, true
)
```

Первый аргумент, селектор запроса, говорит, что нужно искать пользователей, для которых в списке movies есть фильм Белые Росы. Второй аргумент говорит, что нужно добавить в этот список фильм Свадьба в Малиновке с помощью оператора $addToSet.

Remove

Если не задавать никаких параметров, то операция удаления remove удалит из коллекции все документы. Так, чтобы избавиться от коллекции foo, нужно выполнить такую команду:

Удаление коллекции foo. Листинг 9.12

```
db.foo.remove()
```

Удаление по селектору запроса:

Удаление коллекции foo по селектору запроса. Листинг 9.13

```
db.foo.remove({‘favorites.cities’: ‘Минск’})
```

Операция remove не уничтожает коллекцию, а лишь документы текущей коллекции. Чтобы уничтожить коллекцию вместе со всеми построенными над ней индексами, необходимо воспользоваться командой drop.

Удаление коллекции со всеми зависимостями. Листинг 9.14

```
db.users.drop()
```

9. 3 Основы использования

Поддерживается автозавершение. Введите первые символы имени любого метода и дважды нажмите клавишу Tab. Будет выведен список всех методов с подходящими именами.

Оболочка MongoDB одновременно является интерпретатором JavaScript. Следовательно, мы можем использовать конструкции языка JavaScript:

Использование конструкций языка JavaScript. Листинг 9.15

```
for(i=0; i<200000; i++) {
 db.numbers.save({num: i});
}
```

Для уточняющего поиска можно употреблять операторы gt и lt:

Уточняющий поиск. Листинг 9.16

```
db.numbers.find( {num: {"$gt": 20, "$lt": 25 }})
```

Индексирование

Этой коллекции явно недостает индекса. Построить индекс по ключу num можно с помощью метода ensureIndex ().

EnsureIndex. Листинг 9.17

```
db.numbers.ensureIndex({num: 1})
```

В данном случае документ {num: 1} говорит, что над коллекцией numbers нужно построить индекс по ключу num в порядке возрастания.

Убедиться в том, что индекс действительно построен, позволит метод get-Indexes():

GetIndexes. Листинг 9.18

```
db.numbers.getIndexes()
```

Список команд:

show dbs – показать базы данных.

show collections – список коллекций текущей базы данных.

db.stats() – информация о базах данных.

db.test.stats() – информация о коллекциях текущей базы данных.

db.help – список методов для работы с базами.

db.test.help – список методов для работы с коллекциями.

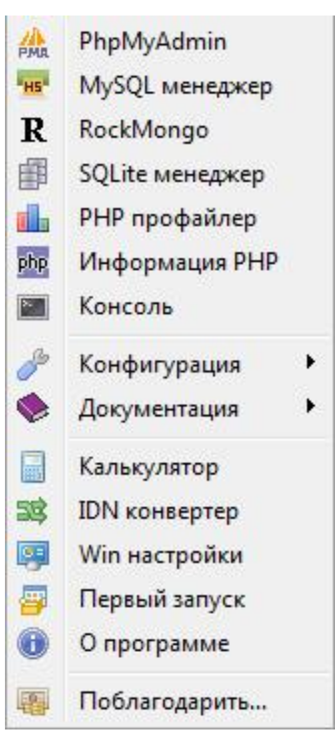
db.test.count() – количество записей коллекции.

OpenServer

Если у вас установлен OpenServer, отдельно скачивать и устанавливать MongoDB не нужно.

Настройки->Модули->включить MongoDB

После чего во вкладке Дополнительно появится менеджер RockMongo.



В Node.js имеется несколько модулей, ориентированных на работу с MongoDB:

MongoDB Native Node.js driver

Mongoose, с которым мы и будем работать.

9.4 Mongoose

Это модуль для работы с хранилищем данных MongoDB на платформе Node.js.

Установка Mongoose. Листинг 9.19

```
npm i mongoose
```

Подключение базы данных:

Подключение базы данных test. Листинг 9.20

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

Рассмотрим еще один способ подключения, слушающий два события: успешное подключение и ошибка подключения.

Подключение базы данных test. Листинг 9.21

```
var db = mongoose.createConnection('mongodb://localhost/test');
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function callback () {
  // yay!
});
```

Рассмотрим создание модели с одним полем name:

Модель themas. Листинг 9.22

```
Schema = mongoose.Schema;
var schema = new Schema({
    name: {
        type: String,
        unique: true,
        required: true
    }
});

exports.Themas = mongoose.model('themas', schema);
```

Как видно из листинга модели, пока в коллекции у нас один документ – name, для хранения данных типа String. Данные в этом документе должны быть уникальны (unique: true) и обязательны для вставки, иначе формируется ошибка.

Вставка данных:

Вставка значения Tester, метод save(). Листинг 9.23

```
var Themas = require('./models/themas').Themas;
var themas = new Themas({
 name: 'Tester'
});
themas.save(function(err, user, affected){
```

```
     console.log('Ok');
});
```

Вывод множества значений

Вывод данных, метод find(). Листинг 9.24

```
var Themas = require('models/themas').Themas;
Themas.find(function(err, info){
     console.log(info);
});
```

Вывод одного документа

Вывод данных, метод find(). Листинг 9.25

```
var Themas = require('models/themas').Themas;
Themas.findOne({‘url’:‘index’},function(err, info){
     console.log(info);
});
```

Отслеживание процессов mongoose

Debug mongoose. Листинг 9.26

```
mongoose.set(‘debug’, true);
```

10. 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 – установка модуля

Далее создадим папку config, а в ней конфигурационный файл config.js

Конфигурационный файл config.js. Листинг 10.1

```
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 10.2

```
var config = require('./config/config');
```

Обратимся к базе данных из контроллера

Функция SELECT-запроса. Листинг 10.3

```
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 10.4

```
  h2 #{ttext.name}
  div.main #{ttext.body}
```

11. Express

11.1 Установка и настройка

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 11.1

```
npm install -g express
// или
npm i -g express-generator@4
// или
npm i -g express-generator
```

Об успешной установки модуля свидетельствует следующее сообщение консоли:

```
Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\Александр>npm install express
npm http GET https://registry.npmjs.org/express
npm http 200 https://registry.npmjs.org/express
npm http GET https://registry.npmjs.org/connect/2.11.0
npm http GET https://registry.npmjs.org/commander/1.3.2
npm http GET https://registry.npmjs.org/fresh/0.2.0
npm http GET https://registry.npmjs.org/buffer-crc32/0.2.1
npm http GET https://registry.npmjs.org/range-parser/0.0.4
npm http GET https://registry.npmjs.org/cookie/0.1.0
npm http GET https://registry.npmjs.org/methods/0.1.0
npm http GET https://registry.npmjs.org/mkdirp/0.3.5
npm http GET https://registry.npmjs.org/cookie-signature/1.0.1
npm http GET https://registry.npmjs.org/send/0.1.4
npm http GET https://registry.npmjs.org/debug
npm http 304 https://registry.npmjs.org/connect/2.11.0
npm http 304 https://registry.npmjs.org/buffer-crc32/0.2.1
npm http 304 https://registry.npmjs.org/fresh/0.2.0
npm http 304 https://registry.npmjs.org/range-parser/0.0.4
npm http 304 https://registry.npmjs.org/commander/1.3.2
npm http 304 https://registry.npmjs.org/cookie/0.1.0
npm http 304 https://registry.npmjs.org/methods/0.1.0
npm http 304 https://registry.npmjs.org/mkdirp/0.3.5
npm http 304 https://registry.npmjs.org/cookie-signature/1.0.1
npm http 304 https://registry.npmjs.org/send/0.1.4
npm http 200 https://registry.npmjs.org/debug
npm http GET https://registry.npmjs.org/keypress
npm http GET https://registry.npmjs.org/pause/0.0.1
npm http GET https://registry.npmjs.org/uid2/0.0.3
npm http GET https://registry.npmjs.org/bytes/0.2.1
npm http GET https://registry.npmjs.org/raw-body/0.0.3
npm http GET https://registry.npmjs.org/qs/0.6.5
npm http GET https://registry.npmjs.org/methods/0.0.1
npm http GET https://registry.npmjs.org/negotiator/0.3.0
npm http GET https://registry.npmjs.org/multiparty/2.2.0
npm http 304 https://registry.npmjs.org/keypress
npm http 304 https://registry.npmjs.org/qs/0.6.5
npm http 304 https://registry.npmjs.org/methods/0.0.1
npm WARN package.json methods@0.0.1 No README.md file found!
npm http 304 https://registry.npmjs.org/negotiator/0.3.0
npm http 304 https://registry.npmjs.org/pause/0.0.1
npm http 304 https://registry.npmjs.org/uid2/0.0.3
npm WARN package.json uid2@0.0.3 No README.md file found!
npm http 304 https://registry.npmjs.org/bytes/0.2.1
npm http 304 https://registry.npmjs.org/multiparty/2.2.0
npm http 200 https://registry.npmjs.org/raw-body/0.0.3
npm http GET https://registry.npmjs.org/raw-body/-/raw-body-0.0.3.tgz
npm http 200 https://registry.npmjs.org/raw-body/-/raw-body-0.0.3.tgz
npm http GET https://registry.npmjs.org/readable-stream
npm http GET https://registry.npmjs.org/stream-counter
npm http 304 https://registry.npmjs.org/readable-stream
npm http 304 https://registry.npmjs.org/stream-counter
npm http GET https://registry.npmjs.org/debuglog/0.0.2
npm http GET https://registry.npmjs.org/core-util-is
npm http 304 https://registry.npmjs.org/debuglog/0.0.2
```

Далее с помощью команды *express* мы можем создавать приложения на Node. Это команда позволяет генерировать костяк сайта.

Опции Express. Листинг 11.2

```
express -help
```

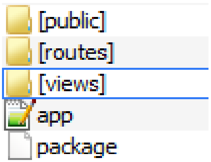
```
C:\OpenServer\domains\Node\chat>express --help

  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 11.3

```
express -s -e
```

В рабочей папке Express создаст такую структуру:



11.2 Создание express-приложения

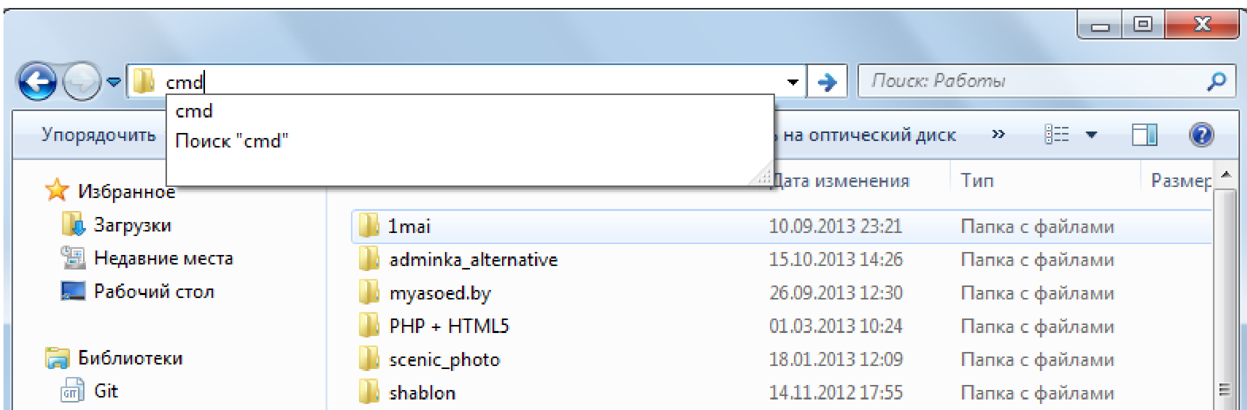
Для создания express-приложения необходимо выполнить следующие шаги:

1. Создать express-приложение с помощью консольной команды express

express – команда создания express приложения в текущей папке.

express site – команда создания express приложения в папке site.

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 11.4

```
Express site // установка в дирректорию site
Express // установка в открытую дирректорию
```

Приложение создает каталог site со следующими подкаталогами: public, routes, views. И файл app.js, который, в свою очередь, создает сервер. Рассмотрим его.

2. Установить внешние зависимости

npm i – данная консольная команда отркывает файл package.json, и устанавливает все зависимости, которые прописаны в этом файле. В итоге в папке с проектом появляется папка node_modules содержащая установленные внешние зависимости.

Установка зависимостей. Листинг 11.5

```
Npm i
```

3. Указать порт, который разрешается прослушивать приложению.

Для этого необходимо открыть файл app.js, и в конце файла, перед экспортированием переменной, добавить следующий код:

Прослушка порта в файле app.js. Листинг 11.6

```
app.listen(8123);
```

4. Запуск приложения.

Запуск приложения через консоль

Запуск приложения. Листинг 11.7

```
node app.js
```

5. Вызов приложения в браузере.

Теперь в любом браузере мы можем прослушать данное приложение, указав при этом порт.

Запуск приложения. Листинг 11.8

```
localhost:8123
```

11.3 Структура Express

Файл app.js

Это главный файл приложения, файл, который запускает сервер. В этом файле находятся маршруты, и к нему подключаются все контроллеры приложения, реализуя паттерн MVC.

var express = require('express'); - подключение модуля expres

var routes = require('./routes'); - подключение дирректории routes. Из данной диррректории подключается файл index.js

В файле index.js находится следующий код:

Файл index.js подкаталога routes. Листинг 11.9

```
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('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-запросов app.delete и app.put, вместе со стандартными app.get и app.post.

Пример html-формы с элеменом с методом put. Листинг 11.10

```
<form> ...
  <input type="hidden" name="_method" value="put" />
</form>
```

При подключенном methodOverride() для такой формы можно использовать метод app.put.

Использование методов-запросов. Листинг 11.11

```
app.use()
app.get()
app.post()
app.delete()
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

В файл app.js необходимо прописать порт, который будет прослушиваться сервером.

package.json

Файл package.json. Листинг 11.12

```
{
  "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 11.13

```
npm i
```

После чего появится новая дирректория **/node_modules**.

/routes

Папка routes содержит контроллеры приложения, на которых ссылается маршрутизатор из файла app.js

В файле route/index.php проверим на существование параметр req.params.id. Если такая переменная есть, то в переменную var indx добавляем значение req.params.id. В противном случае, переменная indx равна значению по умолчанию. Далее, вместо значения ‘Express’, в шаблон передаем переменную indx.

Значение из адресной строки. Листинг 11.14

```
    if(req.params.id){
        var indx = req.params.id;
    }else{
        var indx = 'index';
    }
```

/views

В данной папке хранятся все jade-шаблоны приложения. Шаблоны вызываются из контроллеров

/public

Папка для хранения клиентских зависимостей: изображений, иных файлов для скачивания и просмотра, клиентских js-скриптов и стилей.

Причем для формирования url к файлу, папку /public указывать не надо.

11.4 Шаблонизация проекта

Система шаблонов Jade устанавливается вместе с платформой Express, которая используется по-умолчанию в express-приложениях.

Jade

В шаблонах jade отсутствуют закрывающиеся html-тэги, и знаки “<>” тэгов. Вложенность определяется количеством отступов от начала строки относительно предыдущей строки.

Пример jade-кода.

Jade-код. Листинг 11.15

```
Html
  head
    title Это заголовок
  body
    p Это абзац
```

Данный jade-код преобразуется в следующую html-разметку.

Сгенерированный html-код. Листинг 11.16

```
<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 11.17

```
Html
  head
    title Это заголовок
  body
    div.content
        div#title
            p Это абзац
```

Этот код генерирует следующую разметку:

Сгенерированный html-код. Листинг 11.18

```
<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 11.19

```
p.
 Большой блок текста
 Еще один блок
```

Атрибуты в jade-шаблон вставляются в круглых скобках.

Использование атрибутов в jade-коде. Листинг 11.20

```
input(type="text"
      name="widgetname")
```

Конструкции jade-шаблона:

Консрукция **if**.

Консрукция **each** – аналог консрукции foreach в смежных языках программирования.

Консрукция **include** подключает файл.

Подключается движок шаблонов jade в файле app.js так:

Подключение jade-шаблона в app.js. Листинг 11.21

```
app.set('view engine', 'jade')
```

Система шаблонов настоена следующим образом. Контроллер подключается к подшаблону, который сам себя вставляет в базовый шаблон. Сам по себе подшаблон ничего не выводит. Он лишь формирует переменную, которая выводится в базовом шаблоне.

Такая система работы с шаблонами используется и в php-фрэймворках, например Laravel. Удобство её использования заключается в том, что разработчику легче управлять областью видимости переменных в шаблонах.

В файл index.jade добавим ссылки.

Index.jade. Листинг 11.22

```
extends layout
```

```
block content
  h1= title
  nav.topmenu
   a(href='index') Главная
   a(href='news') Новости
   a(href='services') Услуги
   a(href='concatct') Контакты
  div.mainblock
   h2 Главная
   div.block_for_text.
    Завершать элемент можно точкой, показывающей, что дальнейший
блок содержит только текст.
  div.copyright &copy; Все права защищены. MyStudio 2014г.
```

Ejs

Устанавливаем систему шаблонов EJS (Embedded JavaScript – внедряемый JavaScript код) с помощью Node пакетов npm.

Установка модуля ejs. Листинг 11.23

```
npm install ejs
```

Рассмотрим пример ejs-кода.

Ejs-шаблон. Листинг 11.24

```
<% 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 11.25

```
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");
```

11.5 Маршрутизация

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 11.26

```
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

Использование паттерна MVC позволяет писать логику не в callback() функциях, а в отедельных файлах-контроллерах.

Создание маршрутов и их связь с контроллерами. Листинг 11.27

```
var express = require('express');
// объявление переменной контроллера, подключаемого файла index.js
var base = require(‘./routes/index’)
var app = express();
…
// прослушка любых запросов адресной строки и передача их в экшн
index контроллера base.
app.use('/:id?', base.index);
…
app.listen(3000);
```

Еще примеры формирования маршрутов.

Настройка маршрутизации. Листинг 11.28

```
app.get('/', routes.index);
app.get('/users', user.list);
app.get('/:id', routes.index);
```

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);*

11.6 Конфигурирование

Подключим модуль nconf. Для этого в адресной строке набираем:

Файл index.js. Листинг 11.29

```
npm i nconf
```

Любой модуль, который мы подключаем необходимо прописать в файле packege.json:

Добавляем зависимость в файл packege.json. Листинг 11.30

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.4.4",
    "jade": "*",
    "nconf": "*",
  }
}
```

Далее создадим папку config, в папке config – файл index.js

Файл index.js. Листинг 11.31

```
var nconf = require('nconf');
var path = require('path');

nconf.argv()
     .env()
     .file({file: path.join(__dirname, 'config.json')});

module.exports = nconf;
```

Как видно из листинга, модуль nconf подключает json-файл config.json, который должен находится в текущей директории. В данном файле будем хранить все настройки сайта.

Файл config.json. Листинг 11.32

```
{
 "port":3000
}
```

Внесем изменения в файл app.js

Использование конфигурационного модуля в файле app.js. Листинг 11.33

```
var config = require('./config');
…
http.createServer(app).listen(config.get('port'), function(){
```

```
  console.log('Express server listening on port '
  + config.get('port'));
});
```

11.7 Подключение скриптов и стилей

Пути к скриптам и стилям лучше всего держать в элементах массива. Так мы сможем управлять массивом, добавлять или удалять стили и скрипты на разных страницах приложения

Объявление переменной скрипта в файле layout.jade. Листинг 11.34

```
block scripts
```

В файле app.js после подключения конфигурационного файла извлечем данный массив и передадим его в шаблон layout.jade, используя объект locals

Формирование переменной содержащей скрипты. Листинг 11.35

```
block scripts
  script(src='javascripts/my.js', type='text/javascript')
```

В главном файле шаблона layout.jade, просто указываем место, где необходимо вывести эту переменную.

Вывод скриптов. Листинг 11.36

```
Html
  head
   block scripts
```

Если переменную scripts в подшаблоне не объявить, к ошибке это не приводит.

11.8 Подключение базы данных

Будем работать с модулем Moongose, который необходимо установить.

Далее расширим конфигурационный файл:

Файл config.json с подключением базы данных. Листинг 11.37

```
{
 "port":3000,
 "mongoose": {
     "uri": "mongodb://localhost/kurs",
     "options": {
         "server": {
             "socketOptions": {
                 "keepAlive": 1
             }
         }
     }
 }
}
```

В папке config создадим еще один файл mongoose.js, задача которого подключиться к базе данных. Конфигурацию подключения будем брать из файла json.

Mongoose.js. Листинг 11.38

```
var mongoose = require('mongoose');
var config = require('../config');

mongoose.connect(config.get('mongoose:uri'),
                 config.get('mongoose:options'));

module.exports = mongoose;
```

В корне проекта создадим папку для моделей models и модель.

Модель themas. Листинг 11.39

```
var mongoose = require('../config/mongoose'),
    Schema = mongoose.Schema;

var schema = new Schema({
  name: {
    type: String,
    required: true
  },
  body: {
    type: String,
    unique: true,
    required: true
  },
  url: {
    type: String,
    unique: true,
    required: true
```

```
  },
});

exports.Themas = mongoose.model('themas', schema);
```

Вставка данных

Для выполнения запроса вставки выполним следующий код.

Вставка значений в коллекцию themas. Листинг 11.40

```
var Themas = require('./models/themas').Themas;
var themas = new Themas({
 name: 'Добро пожаловать на сай',
 body: 'Текст для главной',
 url: 'index',
});
themas.save(function(err, user, affected){
     console.log('Ok');
});
```

Обновим, или запустим приложение.

Перезапуск приложения. Листинг 11.41

```
node app.js
```

После выполнения в коллекцию themas вставятся новые данные. Далее заменим данные для других страниц и еще раз выполним перезагрузку. И так несколько раз, до тех пор, пока коллекция не заполнится.

Вывод данных

Обновим файл index.js из папки routes, где и будет осуществляться непосредственный запрос в базу данных.

Обработка и передача данных в шаблон. Листинг 11.42

```
exports.index = function(req, res){
    if(req.params.id){
        var indx = req.params.id;
    }else{
        var indx = 'index';
    }
    var Maintexts = require('../models/maintexts').Maintexts;
        Maintexts.findOne({'url':indx}, function(err, ttext){
           if(!ttext){
               ttext = {
                   name:'Добро пожаловать на сайт',
                   body:'Извините, страница не найдена'
               }
           }
   res.render('index', {
                ttext: ttext,
   });
  });
```

```
};
```

В шаблоне index.jade переменная ttext будет выводиться так:

Вывод значений в шаблоне index.jade. Листинг 11.43

```
h2 = ttext.name
div = ttext.body
```

Если в переменной ttext.body имеется html-код, то выводить эту переменную нужно по-другому:

Вывод html-кода. Листинг 11.44

```
!{ttext.body}
```

11.9. Регистрация и авторизация пользователей

Для регистрации и авторизации можно воспользоваться модулями OAuth или OpenId. Однако многие разработчики предпочитают иметь свою систему входа.

Обычно она реализуется с помощью сессий:

- Пользователь заполняет форму, указывая логин и пароль
- Пароль шифруется с помощью хэш-алгоритма
- Полученное значение сравнивается с тем, что хранится в БД
- Если они совпадают, то генерируется сессионный ключ, идентифицирующий пользователя

Настройка маршрутов

В файле app.js по маршруту /reg добавим вызов нового контроллера.

Подключение html-формы. Листинг 11.45

```
app.get('/reg', reg.index);
```

Создадим переменную reg, задача которой подключить контроллер reg из папки routes:

Переменная reg. Листинг 11.46

```
var reg = require('./routes/reg');
```

Контроллер index файла reg.

Контроллер index файла reg. Листинг 11.47

```
exports.index = function(req, res){
 res.render('reg');
}
```

Форма регистрации

Рассмотрим саму форму:

Файл шаблона reg.jade. Листинг 11.48

```
extends layout
block content
 form(method='POST', action='/reg')
  input(type='text', name='username')
  input(type='password', name='password')
  input(type='submit', value='Ok')
```

В файле app.js добавим прослушиватель post данных по маршруту post. Полученные данные отправляем в тот же файл, который рендерит шаблон feedback.jade, файл reg.js:

Прослушка данных, файл app.js. Листинг 11.49

```
app.post('/reg', reg.reg);
```

Регистрация пользователей

Для хранения паролей в зашифрованном виде, в консоле подключим модуль crypto:

Модуль crypto. Листинг 11.50

```
npm i crypto
```

Не забываем прописывать новый модуль в файле package.json:

Файл package.json. Листинг 11.51

```
  "dependencies": {
    "express": "3.4.4",
    "jade": "*",
    "nconf": "*",
    "mongoose": "*",
    "crypto": "*"
  }
```

В папке models создадим модель коллекции пользователей.

Модель users. Листинг 11.52

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

var mongoose = require('../config/mongoose'),
  Schema = mongoose.Schema;

var schema = new Schema({
  username: {
    type: String,
    unique: true,
    required: true
  },
  hashedPassword: {
    type: String,
    required: true
  },
  salt: {
    type: String,
    required: true
  },
  created: {
    type: Date,
    default: Date.now
  }
});
```

```
schema.methods.encryptPassword = function(password) {
  return crypto.createHmac('sha1', this.salt)
               .update(password)
               .digest('hex');
};

schema.virtual('password')
  .set(function(password) {
    this._plainPassword = password;
    this.salt = Math.random() + '';
    this.hashedPassword = this.encryptPassword(password);
  })
  .get(function() {
     return this._plainPassword;
});


schema.methods.checkPassword = function(password) {
  return this.encryptPassword(password) === this.hashedPassword;
};

exports.Users = mongoose.model('Users', schema);
```

Создание пользователей:

Создание пользователей. Листинг 11.53

```
var Users = require('./models/users').Users;
  var username = req.body.username;
  var password = req.body.password;
var users = new Users({
 username: username,
 password: password
});
users.save(function(err, user){
     console.log(arguments);
});
```

Скрытие страниц от неавторизированных пользователей

В папке utils создадим файл checkAuth.js, в котором напишем функцию, проверяющую авторизирован ли пользователь. Т.е. есть ли у пользователя сессионная переменная.

Функция checkAuth.js. Листинг 11.54

```
var HttpError =require('../utils/error').HttpError;

module.exports = function(req, res, next) {
  if (!req.session.user) {
    return next(new HttpError(401, "Вы не авторизованы"));
  }

  next();
};
```

Подключим данную утилу к файлу app.js

Функция checkAuth.js. Листинг 11.55

```
var checkAuth = require('./utils/checkAuth');
```

Маршруты тех страниц, которые мы хотим закрыть от неавторизированных пользователей, будем вызывать так:

Настройка маршрутов для авторизированных пользователей. Листинг 11.56

```
app.get('/logout',checkAuth, reg.logout);
```

Т.е. вторым параметром, вызываем функцию проверки, если данная функция возвращает true, то инерпретатор кода переходит к следующей функции, иначе генерируется исключительная ситуация функцией checkAuth.

Сессионную переменную req.session.user мы можем передать в шаблон:

Передача сессионной переменной в шаблон. Листинг 11.57

```
res.render('index', {
   userid: req.session.user,
});
```

Это значит, что переменная userid будет доступна в файле шаблона index.jade.

Передать сессионную переменную в файл шаблона layout можно из файла app.js следующим образом:

Значение из адресной строки. Листинг 11.58

```
app.use(function (req, res, next) {
   res.locals = {
     userid: req.session.user
   };
   next();
});
```

Если сессионная переменная существует, выведем кнопку выход, в противном случае, будет сформирована ссылка на регистрацию:

Передача сессионной переменной в шаблон. Листинг 11.59

```
if userid
  a(href='/logout') Выход
else
  a(href='/reg') Регистрация !{userid}
```

Выход

Для выхода необходимо уничтожить сессию.

Уничтожение сессии. Листинг 11.60

```
exports.logout = function(req, res) {
  req.session.destroy();
  res.redirect('/');
};
```

Сессии в базе данных

Поключим модуль connect-mongo

Инсталяция модуля connect-mongo.js. Листинг 11.61

```
npm i connect-mongo
```

Подключение модуля в файле app.js

Подключение модуля connect-mongo.js. Листинг 11.62

```
var mongoose = require('./config/mongoose');
var MongoStore = require('connect-mongo')(express);
```

В файле config.json пропишем следующие настройки:

Подключение модуля connect-mongo.js. Листинг 11.63

```
  "session": {
    "secret": "killerisSmith",
    "key": "sid",
    "cookie": {
      "path": "/",
      "httpOnly": true,
      "maxAge": null
    }
  }
```

Возвращаемся в файл app.js, используем эти настройки

Подключение модуля connect-mongo.js. Листинг 11.64

```
app.use(express.session({
  secret:config.get('session:secret'),
  key: config.get('session:key'),
  cookie: config.get('session:cookie'),
  store: new MongoStore({mongoose_connection: mongoose.connection})
}));
```

Теперь, если в mongo выполним запрос в коллекцию sessions, по получим следующий ответ:

```
> use kurs
switched to db kurs
> db.sessions.find()
{ "_id" : "yrEkSzyYWpKaWWO4Ia9xdoVS", "session" : "{\"cookie\":{\"originalMaxAge
\":null,\"expires\":null,\"httpOnly\":true,\"path\":\"/\"}}", "expires" : ISODat
e("2014-04-27T06:28:30.231Z") }
> 
```

Это значит, что сессия в базе данных сохранилась.

11.10 Сессии в Express

В основе **сессий в Express** лежит соответствующий средний слой (middleware) из Connect, который, в свою очередь, опирается на механизм хранения данных. Существует хранилище в памяти, а так же сторонние хранилища, включая connect-redis и connect-mongodb. В качестве альтернативы так же можно рассматривать cookie-sessions, который хранит данные сессии в пользовательской куке (cookie).

Подключение модуля cookie-session

Установка модуля cookie-session. Листинг 11.65

```
npm i cookie-session
```

Поддержка сессий может быть включена следующим образом:

Включение поодержки сессии, файл app.js. Листинг 11.66

```
app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(session({
 secret:config.get('session:secret')
}));
```

Этот кусочек кода необходимо разместить между bodyDecoder и methodOverride.

Секретный набор символов для сессии поместим в файл config.json:

Параметр session файла config.json. Листинг 11.67

```
"session": {
    "secret": "killerissSmith"
}
```

Теперь в HTTP-обработчиках будет доступна переменная req.session, к которой можно будет обращаться, например, так:

req.session.message = ‘hello’.

Подключим модуль async

Подключение модуля async. Листинг 11.68

```
npm i async
```

Добавим в модель users метод авторизации и сопутствующие для реализации функции авторизации переменные и модули.

Обновленная модель users.js. Листинг 11.69

```
var crypto = require('crypto');
var async = require('async');

var mongoose = require('../config/mongoose'),
Schema = mongoose.Schema;
var schema = new Schema({
  --//--//--
});

schema.methods.encryptPassword = function(password) {
  return crypto.createHmac('sha1',
this.salt).update(password).digest('hex');
};

schema.virtual('password')
  .set(function(password) {
    this._plainPassword = password;
    this.salt = Math.random() + '';
    this.hashedPassword = this.encryptPassword(password);
  })
  .get(function() { return this._plainPassword; });

schema.methods.checkPassword = function(password) {
  return this.encryptPassword(password) === this.hashedPassword;
};


schema.statics.authorize = function(username, password, callback) {
  var User = this;

  async.waterfall([
    function(callback) {
      User.findOne({username: username}, callback);
    },
    function(user, callback) {
      if (user) {
        if (user.checkPassword(password)) {
          callback(null, user);
        } else {
          //здесь будет обработка ошибок
        }
      } else {
        var user = new User({username: username, password: pass-
word});
        user.save(function(err) {
          if (err) return callback(err);
          callback(null, user);
        });
      }
    }
  ], callback);
};

exports.Users = mongoose.model('users', schema);
```

Далее рассмотрим сам контроллер, обработчик post-запросов формы регистрации.

Обновленный файл reg.js. Листинг 11.70

```
var User = require('../models/users').Users;
var async = require('async');
exports.index = function(req, res){
 res.render('reg');
}

exports.send = function(req, res, next) {
  var username = req.body.username;
  var password = req.body.password;
  User.authorize(username, password, function(err, user) {
      req.session.user = user._id;
      res.send({});
      res.writeHead(302, {
        'Location': '/thankyoupage'
      });
  });

};
```

На данном этапе можно протестировать приложение. Необходимо запустить приложение, перейти по ссылке /reg и попробовать зарегистрироваться либо авторизироваться на сайте.

Обработка ошибок

Создадим папку utils, в которой создадим файл error.js. Задача этого файла – обработка исключительных ситуаций.

Error.js. Листинг 11.71

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

// ошибки для выдачи посетителю
function HttpError(status, message) {
  Error.apply(this, arguments);
  Error.captureStackTrace(this, HttpError);

  this.status = status;
  this.message = message || http.STATUS_CODES[status] || "Error";
}

util.inherits(HttpError, Error);

HttpError.prototype.name = 'HttpError';

exports.HttpError = HttpError;
```

В контроллере reg.js (метод authorize) добавим обработку исключительной ситуации.

Обработка исключительной ситуации, контроллер reg.js. Листинг 11.72

```
var HttpError =require('../utils/error').HttpError;
exports.send = function(req, res, next) {
  var username = req.body.username;
  var password = req.body.password;
  User.authorize(username, password, function(err, user) {
    if (err) {
      if (err instanceof HttpError) {
        return next(new HttpError(403, err.message));
      } else {
        return next(err);
      }
    }
      req.session.user = user._id;
      res.send({});
      res.redirect('/');
  });
};
```

В модели users необходимо подключить обработчик ошибок:

Подключение обработчика ошибок. Листинг 11.73

```
var HttpError =require('../utils/error').HttpError;
```

Сейчас в функциях обратного вызова (callback) модели Users модуля async можем вызвать обработчик исключений следующим обрабом:

Вызов обработчика ошибок с передачей параметров. Листинг 11.74

```
if (user.checkPassword(password)) {
  callback(null, user);
} else {
  callback(new HttpError(403, 'Пароль неверен'));
}
```

Далее на стороне mongo настроим возможность хранения сессионных данных.

11.11 Загрузка файлов

Express предоставляет возможность загрузки и обработки файлов.

Для загрузки файлов на сервер, сперва необходимо указать временную папку для хранения загружаемых файлов.

Добавим в файле app.js следующую настройку:

Настройка папки для временного хранения загружаемых файлов. Листинг 11.75

```
app.use(express.bodyParser({keepExtensions:true, uploadDir: 'pub-
lic/tmp' }));
```

Рассмотрим саму форму с элементом file.

HTML-форма jade-формата. Листинг 11.76

```
form(method='POST', action='/cabinet', enctype='multipart/form-
data' role='form')
  input(type='file', name='book')
```

Обратите внимание на атрибут enctype. Без этого атрибута со значением mu-lipart/form-data форма не будет загружать файлы.

Далее в контроллере обрабатываем файл и перемешаем его из временной дирректории в нужную папку. Для этого необходимо подключить два модуля: path и fs

Загрузка файла. Листинг 11.77

```
var path = require('path'),
    fs = require('fs');
exports.send = function(req, res, next){
    // загрузка фото
    fs.readFile(req.files.book.path, function (err, data) {

        var imageName = req.files.book.name

        // Если какая-то ошибка, выводим информацию об ошибке и
        // перенаправляем пользователя
        if(!imageName){
            console.log("There was an error")
            res.redirect("/");
            res.end();

        } else {
    var newPath = __dirname + "/../public/uploads/" + imageName;
    console.log(newPath);
          // записываем файл
          fs.writeFile(newPath, data, function (err) {

            // открываем загруженный файл.
            res.redirect("/uploads/" + imageName);

          });
```

```
        }
    });
});
```

Проверка типа загружаемого файла:

Проверяем тип загружаемого файла. Листинг 11.78

```
if (path.extname(req.files.file.name).toLowerCase() === '.png') {
         // изображение формата .png
} else {
        // неизвестно что…
}
```

12. Web-сокеты, чат на Socket.IO

Web-сокеты – это технология, поддерживающая двунаправленный обмен данными в реальном времени между клиентом и сервером. При этом клиент и сервер становятся равноправными участниками обмена сообщениями. Обмен данными осуществляется посредством протокола TCP.

Библиотека Socket.IO обеспечивает поддержку, необходимую для реализации этой технологии. Для реализации приложения на Socket.IO необходимо установить компоненты на стороне сервера и клиента. Компоненты можно скачать с сайта socket.io

Установка чата достаточна проста. Мы можем скачать исходники чата на Socket.IO с репазитория github.com.

Скачиваем чат репазитория github.com. Листинг 12.1

```
git clone https://github.com/rauchg/chat-example.git
```

Либо так:

Установка Socket.IO. Листинг 12.2

```
npm install socket.io
```

После чего необходимо установить зависимости для чата из файла package.json.

Установка зависимостей из файла package.json. Листинг 12.3

```
npm i
```

Запускаем чат.

Запуск чата. Листинг 12.4

```
node index.js
```

Сервер чата прослушивает порт 3000. Теперь можем запустить чат в браузере по адресу localhost:3000. Для прослушивания входящих запросов используется протокол HTTP.

Глава IV. Современный фронтенд

С появлением динамики на стороне клиента (браузера) выделились такие понятия, как фронтенд и бэкенд. Бэкенд – программирование на стороне сервера. Фронденд – программирование на стороне клиента. Постепенно обозначились основные задачи фронтенда:

- Вывод (реже обработка) данных, поступивших с бэкенда.
- Шаблонизация, или создание результирующего HTML.

Как в бэкенд, так и в фронтенд разработке широкое распространение получил паттерн MVC (Model-View-Controller), разделяющий все компоненты на предназначенные для получения, хранения и отображения данных. Появилось множество фрэймворков и библиотек, использующие в своих реализациях частично или полностью этот паттерн.

1. Резиновая и фиксированная верстка, традиционная блочная и табличная

1. Фиксированная верстка

CSS и HTML код фиксированной верстки. Листинг 1.1

```
<div style="width:500px; text-align:center; margin:0 auto; pad-
ding:10px">
    <p>Этот <code>блок</code> имеет фиксированную ширину 500
пикселей.
</div>
```

2. Резиновая верстка

CSS и HTML код резиновой блочной верстки. Листинг 1.2

```
<style>
body
{
background-image: url(media/images/body.jpg);
margin:0px auto;
width: 100%;
}

#hedImg
{
position: absolute;
top: 50px;
left: 141px;
width: 78%;
float: left;
}

#logo
{
position: absolute;
top: 87px;
left: 210px;
}

#logoLbl
{
font-size: 25px;
font-family: Verdana, Tahoma, Sans-Serif;
}

#rap
{
background-color: #FFFFFF;
padding: 15px;
margin: 0 auto;
height: 500px;
width: 78%;
}

#menu
```

```
{
position: absolute;
top: 157px;
left: 141px;
width: 78%;
float: left;
}
</style>

<div id = "rap">
    <div id = "hed">
        <img src = "_mod_files/ce_images/top.png" id = "hedImg"
alt = "Шапка сайта">
    </div>
    <div id = "logo">
        <p id = "logoLbl">C#, Delphi, Visual Basic</p>
    </div>
    <div id = "menu">
        <img src = "_mod_files/ce_images/menu.png" width = 100%
height = "39" >
        <ul>
            <li><a id = "menuLink1" href =
"http://programer.web-box.ru/">ГЛАВНАЯ</a></li>
            <li><a id = "menuLink2" href =
"http://programer.web-box.ru/">О САЙТЕ</a></li>
            <li><a id = "menuLink3" href =
"http://programer.web-box.ru/">ИСХОДНИКИ</a></li>
            <li><a id = "menuLink4" href =
"http://programer.web-box.ru/">ПРОГРАММЫ</a></li>
            <li><a id = "menuLink5" href =
"http://programer.web-box.ru/">УРОКИ</a></li>
        </ul>
    </div>
</div>
```

Предыдущие два вида верстки относятся к традиционной блочной верстке.

2. Гибкая блочная верстка

Гибкая блочная верстка пришла на замену традиционной блочной модели. Свойство стилей display:flex указывает на то, что влоеженные элементы будут гибкими (flexable), т.е. подстраиваться под родительский.

Box 1	Box 2	Box 3	Box 4

Гибкая блочная верстка. Листинг 2.1

```
<nav class="topmenu">
  <a href="#"> Box 1</a>
  <a href="#"> Box 2</a>
  <a href="#"> Box 3</a>
  <a href="#"> Box 4</a>
</nav>
<style>
.topmenu {
 display: flex;
 display: -moz-box;
 display: -webkit-flex;
 text-align:center;
 flex-direction: row;
 width:100%;
 margin:0 auto;
}
.topmenu a{
 display:block;
 -moz-box-flex:1;
 -webkit-flex:1;
 flex:1;
 border:solid 1px black;
 padding:10px;
}
</style>
```

Обратите внимание на свойство flex:1. Это говорит о том, что все гибкие блоки будут равны между собой. Единица ширины гибкого блока высчитывается по следующей формуле:

$E = W \div X$

Где:

E – Единица ширины

X – Сумма значений всех атрибутов flex

W – Ширина внешнего родительского блока, для которого прописано свойтсво display:flex.

После определения единицы ширины, чтобы определить ширину гибкого блока, значение единицы ширины нужно умножить на значение свойства flex. В нашем случае, все блоки равны flex:1. Значит:

ширина блока = E×1;

3. Адаптивная верстка

Даже идеальное браузерное отображение не делает никакой разницы между десктопными браузерами и принтерами или между мобильными устройствами и голосовым браузером. Чтобы решить эту проблему, W3C создала список медиатипов для классификации каждого браузера или устройства по медиакатегориям. Медиатипы могут принимать значения: all, braille, embossed, handheld, print, projection, screen, speech, tty и tv.

Все эти медиатипы созданы с одной целью: чтобы мы могли лучше проектировать дизайн для каждого типа браузера или устройства, просто загружая нужный CSS.

Следовательно, устройство с экраном будет игнорировать CSS, созданный для медиатипа print, и наоборот. А для стилевых правил, которые применимы ко всем устройствам, в спецификации создана супергруппа all. На практике это означает правку media-атрибута ссылки:

Медиа-атрибуты ссылок. Листинг 3.1

```
<link rel="stylesheet" href="global.css" media="all" />
<link  rel="stylesheet"  href="main.css" media="screen" />
<link  rel="stylesheet"  href="paper.css" media="print" />
```

Также имеется возможность создавать media-блоки в самих стилях:

Медиа-запросы в файлах стилей. Листинг 3.2

```
@media screen {
       body {
              font-size: 100 %;
       }
}
@media print {
       body {
              font-size: 15 pt;
       }
}
```

Кроме того, имеется возможность адаптироваться не под само медиа-устройство, а под размер экрана любого медиа-устройства.

Учитываем ширину эранана в медиа-запросах. Листинг 3.3

```
#page {
       margin: 36px auto;
       width: 90 %;
}
@media screen and (max-width: 768px) {
       #page {
              position: relative;
              margin: 20px;
              width: auto;
       }
```

```
}
```

4. JSON

Формат представления данных JSON – это уже состоявшаяся классика. Пользоваться им достаточно просто. Необходимо помнить следующие правила:

1. Значения оборачиваются в фигурные скопки.

2. Пара имя-значения разделяются двоеточем.

3. Имена и значения обрамляются в ковычки.

4. Любое значение любой степени вложенности может быть JSON-обектом, строкой, числом или массивом. Если JSON-формат используется в js-файлах: при передачи параметров в функции или отдельным объектом, то значения также могут быть функциями и другими объектами.

Объекты JSON. Листинг 4.1

```
{
‘name’: ‘Vasya’,
‘surname’: ‘Ivanov’,
‘favorites’: {
  ‘films’ : [‘Белое солнце пустыни’, ‘Белые росы’],
  ‘books’ : {‘author’: ‘Достоевский’, ‘name’: ‘Бесы’}
 }
}
```

5. Селекторы

Селектор это часть CSS-правила, которая определяет элемент или элементы, к которым будет применён блок объявлений (стиль), содержащий форматирующие свойства.

Любой селектор может состоять более чем из одного простого селектора. К простым селекторам относятся:

- селектор типа
- универсальный селектор
- селекторы атрибутов
- селектор идентификатора
- селектор класса
- псевдо-классы

Селектор	Пример	Описание	CSS
.class	.myclass	Выбор всех элементов с class="myclass".	1
#id	#main	Выбор элемента с id="main".	1
*****	*	Выбор всех элементов.	2
элемент	span	Выбор всех элементов <span>.	1
элемент,элемент	div,span	Выбор всех элементов <div> и всех элементов <span>.	1
[атрибут]	[title]	Выбор элементов с атрибутом "title".	2
[атрибут=значение]	[title=cost]	Выбор всех элементов с title="cost".	2
[атрибут~=значение]	[title~=cost]	Выбор элементов с атрибутом title, содержащим слово cost.	2
[атрибут|=значение]	[lang|=ru]	Выбор всех элементов с атрибутом lang, значение которого начинается с "ru".	2
[атрибут^=значение]	a[src^="https"]	Выбор каждого элемента <a> с атрибутом src, значение которого начинается с "https".	3

[атрибут$=значение]	a[src$=".png"]	Выбор каждого элемента <a> с атрибутом src, значение которого заканчивается на ".png".	3
[атрибут*=значение]	a[src*="puzzleweb"]	Выбор каждого элемента <a> с атрибутом src, в значении которого есть "puzzleweb".	3

Комбинаторы

Для объединения простых селекторов, используются комбинаторы, которые указывают взаимосвязь между простыми селекторами. Существует несколько различных комбинаторов в CSS2, и один дополнительный в CSS3, когда вы их используете, они меняют характер самого селектора.

Комбинатор	Пример	Описание	CSS
элемент элемент	div span	Выбор всех элементов <span> внутри <div>.	1
элемент>элемент	div>span	Выбор всех элементов <span>, у которых родитель <div>.	2
элемент+элемент	div+p	Выбор элемента <p>, который размещается сразу же после элементов <div>.	2
элемент1~элемент2	p~ol	Выбор всех элементов , которым предшествует элемент <p>.	3

Псевдо-классы

Псевдо-класс похож на обычный класс в HTML, за исключением того, что он явно не указывается в разметке HTML-документа. Некоторые псевдо-классы динамичны - они применяются в результате взаимодействия пользователя с документом, например при наведении курсора мыши на ссылку. Любые псевдо-классы начинаются с двоеточия ":".

Псевдо-класс	Пример	Описание	CSS
:link	a:link	Выбор всех не посещенных ссылок.	1
:visited	a:visited	Выбор всех посещенных ссылок.	1
:active	a:active	Выбор активной ссылки.	1
:hover	a:hover	Выбор ссылки при наведении курсора мышки.	1
:focus	input:focus	Выбор элемента input, который находится в фокусе.	2
:first-child	p:first-child	Выбор каждого элемента <p>, кото-	2

		рый является первым дочерним элементом своего родителя.	
:lang(язык)	p:lang(ru)	Выбор каждого элемента <p> с атрибутом lang, значение которого начинается с "ru".	2
:first-of-type	p:first-of-type	Выбор каждого элемента <p>, который является первым из элементов <p> своего родительского элемента.	3
:last-of-type	p:last-of-type	Выбор каждого элемента <p>, который является последним из элементов <p> своего родительского элемента.	3
:only-of-type	p:only-of-type	Выбор каждого элемента <p>, который является единственным элементом <p> своего родительского элемента.	3
:only-child	p:only-child	Выбор каждого элемента <p>, который является единственным дочерним элементом своего родительского элемента.	3
:nth-child(n)	p:nth-child(2)	Выбор каждого элемента <p>, который является вторым дочерним элементом своего родительского элемента.	3
:nth-last-child(n)	p:nth-last-child(2)	Выбор каждого элемента <p>, который является вторым дочерним элементом своего родительского элемента, считая от последнего дочернего элемента.	3
:nth-of-type(n)	p:nth-of-type(2)	Выбор каждого элемента <p>, который является вторым дочерним элементом <p> своего родительского элемента.	3
:nth-last-of-type(n)	p:nth-last-of-type(2)	Выбор каждого элемента <p>, который является вторым дочерним элементом <p> своего родительского элемента, считая от последнего дочернего элемента.	3
:last-child	p:last-child	Выбор каждого элемента <p>, который является последним элементом своего родительского элемента.	3
:root	:root	Выбор корневого элемента в документе.	3
:empty	p:empty	Выбор каждого элемента <p>, кото-	3

		рый не содержит дочерних элементов (включая текст).	
:target	:target	Выбор активного элемента на странице, который имеет якорную ссылку.	3
:enabled	input:enabled	Выбор каждого включенного элемента <input>.	3
:disabled	input:disabled	Выбор каждого выключенного элемента <input>.	3
:checked	input:checked	Выбор элемента <input>, выбранного по умолчанию или пользователем.	3
:not(селектор)	:not(p)	Выбор всех элементов, кроме элемента <p>.	3

Псевдо-элементы

Псевдо-элементы - это виртуальные элементы, которые не существует в явном виде в дереве документа. Псевдо-элементы могут быть динамическими, поскольку виртуальные элементы могут изменяться, например, когда ширина окна браузера изменяется. Они также могут представлять содержимое, создаваемое с помощью правил CSS. Любые псевдо-элементы начинаются с двойного двоеточия "::".

Псевдо-элемент	Пример	Описание	CSS
::first-letter	p::first-letter	Выбор первой буквы каждого элемента <p>.	1
::first-line	p::first-line	Выбор первой строки каждого элемента <p>.	1
::before	p::before	Вставка содержимого перед каждым элементом <p>.	2
::after	p::after	Вставка содержимого после элеметна <p>	2

6. jQuery (библиотека запросов)

Библиотека jQuery подключается несколькими способами. Можно подключаться к удаленному файлу, не скачивая. Можно скачать файл библиотеки, и подключаться к скаченному файлу.

Еще один способ подключения – с помощью GOOGLE Code. Это делается в надежде на то, что к моменту посещения вашего сайта пользователи уже будут иметь копию библиотеки, кэшированную с другого сайта, который в свое время загрузил эту библиотеку с GOOGLE Code, что обеспечит сокращение времени загрузки страниц для пользователей нашего сайта.

Третий способ: использование Google Libraries API. Листинг 6.1

```
<script type=”text/javascript” src=”http://www.google.com/jsapi” >
</script>
<script type=”text/javascript” >
      google.load(“jquery”, “1.9.2”);
</script>
```

Также данный способ необходимо использовать, если мы планируем подключать помимо jQuery другие библиотеки.

Вся работа с jQuery ведётся с помощью функции $. Если на сайте применяются другие javascript библиотеки, где $ может использоваться для своих нужд, то можно использовать её синоним — jQuery. Второй способ считается более правильным, а чтобы код не получался слишком громоздким пишут его следующим образом:

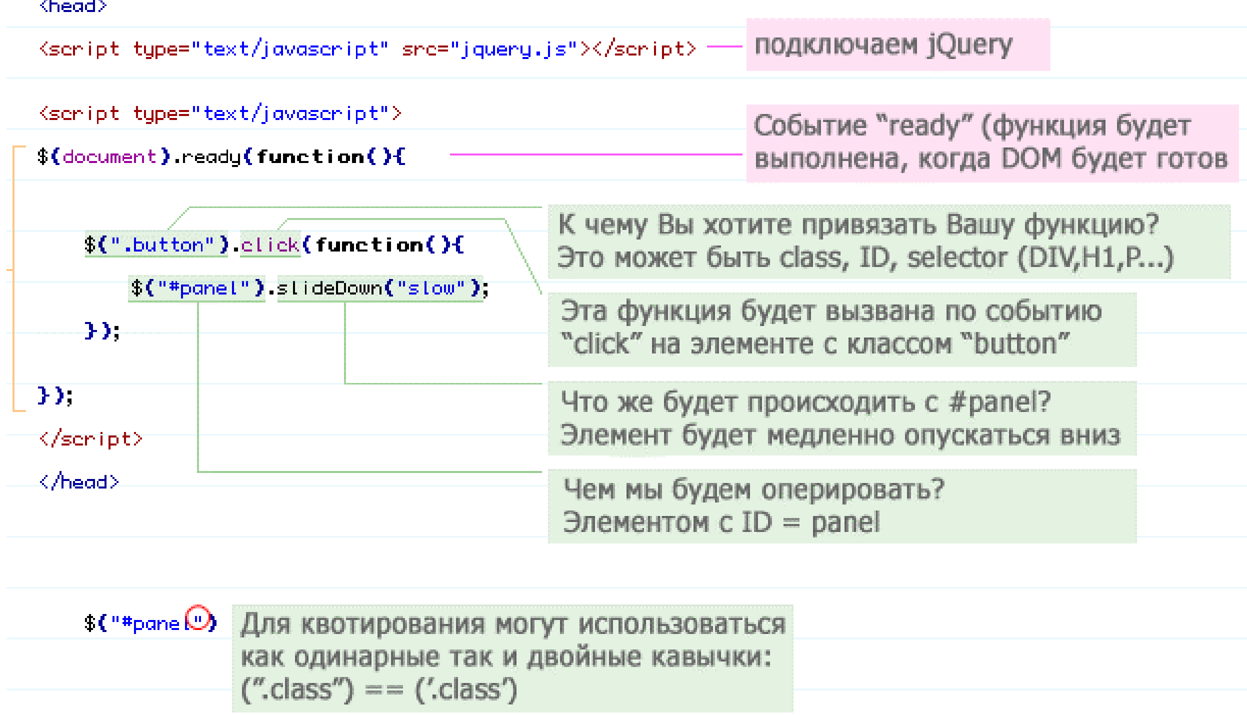
Весь jQuery код внутри данной функции. Листинг 6.2

```
jQuery(function($) {
  // Тут код скрипта, где в $ будет jQuery
})
```

Три этапа работы библиотеки jQuery: 1) выбор селектора 2) подключение обработчика событий 3) вызов функции.

Пример работы jQuery. Листинг 6.3

```
  $(".button").click(function(){
    $("#panel").slideDown("slow");
  });
```



Основа работы jQuery – это умение обращаться к любому тегу на странице, т.е. знание селекторов. Селекторы jQuery особенно просто освоить тем, кто уже знаком с CSS, поскольку им придется иметь дело с тем же синтаксисом.

В jQuery используются стандартные селекторы, которые были рассмотрены ранее в текущем разделе.

К стандартным селекторам можно после символа “:” добавлять фильтры.

Фильтры

- *$(“p:first”), $(“p:last”)* – выбор first – первого, last – последнего элемента
- *$(“p:not(.foo)”)* – выбор элементов, не соответствующих селектору.
- *$(“p:odd”), $(“p:even”)* – выбор элементов по признаку четности. Odd – выбирает нечетные элементы. Even - четные.
- *$(“p:eq(3)”)* – выбор элементов по индексу. В качестве параметра передается индекс требуемого значения. Индекс первого элемента – 0.
- *$(“p:contains(какой-то текст)”)* – выбор элементов, содержащих определенный текст. Фильтр чувствителен к регистру.
- *$(“p:has(span)”)* – выбор элементов, содержащих указанный элемент.
- *$(“:empty”)* –выбор пустых элементов
- *$(“p:parent”)* – выбор родительских элементов.

- *$(“:hidden”), $(“:visible”)* – выбор соответственно скрытых и видимых элементов.
- *$(“[src=img/pic1.jpg]”)* – выбор элементов по значению атрибута.
- *$(“[src!=img/pic1.jpg]”)* – выбор элементов, не имеющих заданного атрибута, или имеющих другое значение.
- *$(“:button”), $(“:checkbox”), $(“:input”), $(“:radio”)* и т.д. – выбор соответствий по типу форм.
- *$(“:disabled”), $(“:enabled”)* – выбор включенных и отключенных элементов форм.
- *$(“:selected”), $(“:checked”)* – выбор выделенных или отмеченных элементов формы.

Познакомившись с селекторами и фильтрами jQuery, и подключив саму библиотеку, можно протестировать работу библиотеки. Для этого воспользуемся плагином firebug для firefox. Включим консоль (F12), и впишем какой-нибудь существующий на странице тег, например “a”. Если библиотека подключилась, мы увидим перечень всех существующих ссылок на странице.

Методы bind() и unbind()

Задача метода bind() – связать обработчик событий с функцией jQuery.

Метод bind(). Листинг 6.4

```
$(".selector").bind("click", function() {
console.log("Событие – щелчок по ссылке");
});
```

Таким образом, мы связали селектор a с функцией через событие click.

Чтобы привязать к одному селектору разные обработчики (например, click и mouseover), можно использовать следующий код.

Связка нескольких обработчиков с одним селектором. Листинг 6.5

```
$("p").bind({
     "click": function(){функция},
     "mouseover":function(){функция}
});
```

Для того чтобы удалить все события всех абзацев, используем следующий код:

Отключение всех событий. Листинг 6.6

```
$("p").unbind();
```

Отключение конкретного события. Листинг 6.7

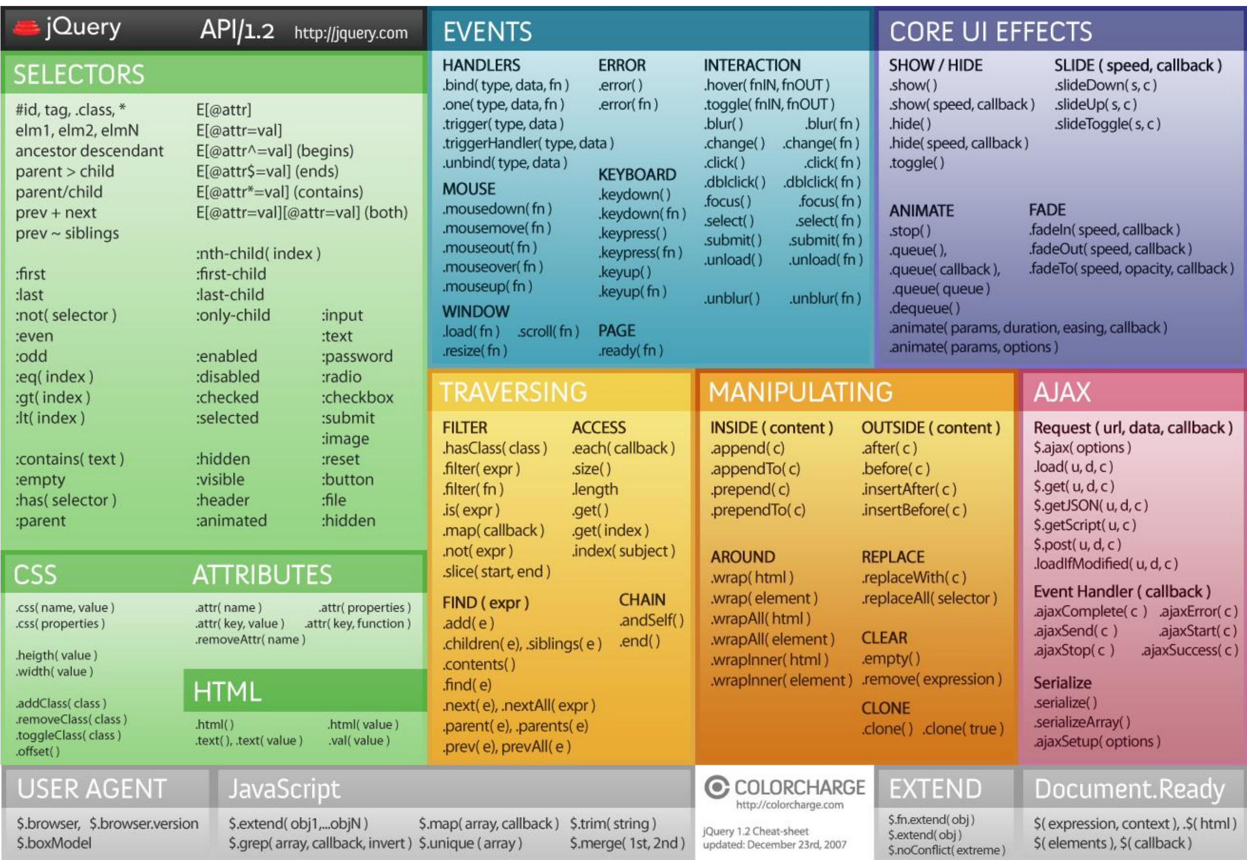
```
$("p").unbind("click");
```

Если один селектор необходимо связать с несколькими jQuery-методам, то методы можно вызывать последовательно связывая их точкой.

Последовательный вызов методов. Листинг 6.8

```
$("p").method1().method2();
```

Шпаргалка jQuery.



7. Объектные литералы

Объектный литерал – это переменная javaScript с двумя фигурными скопками, в которые можно добавлять любое количество значений, используя пары: имя-значение, разделенные запятой.

Объектные литералы. Листинг 7.1

```
var obj = {
 "name" : 'Иван',
 "age"  : '25'
}
```

Чтобы получить доступ к этим значениям, нужно вызвать имя литерала и через точку добавить имя значения.

Вызов значений литерала. Листинг 7.2

```
alert(obj.name);
```

Что делает литералы особенно ценными, так это то, что в них можно хранить функции.

Функции в объектных литералах. Листинг 7.3

```
var obj = {
 "name" : 'Иван',
 "age"  : '25',
"func" : function() {alert("Объектные литералы - это круто!")}
}
```

Для вызова функций из литералов используется тот же синтаксис, что и для доступа к значениям с добавлением пары круглых скопок.

Вызов функции в литералах. Листинг 7.4

```
obj.func();
```

8. Архитектурный шаблон MVVM

MVC (Model, View, Controller)

MV-VM (Model-View View-Model)

В шаблонах проектирования MVC/MVP изменения в пользовательском интерфейсе не влияют непосредственно на Модель, а предварительно идут через Контроллер (англ. Controller) или Presenter.

У паттерна проектирования MV-VM нет контроллера. Данные напрямую вставляются в шаблон. Пользователь из шаблона может мгновенно обновить, добавить или удалить данные. Такой паттерн проектирования используется в почтовых клиентах (gmail.com, ya.ru и других), в чатах мгновенных сообщений (twiter) и т.д. Отличительная особенность сайтов на этом паттерне - исходный код страницы представляет из себя смесь кода javaScript и тэгов html.

MVVM удобно использовать вместо классического MVC и ему подобных в тех случаях, когда в платформе, на которой ведётся разработка, присутствует «связывание данных».

Одна из самых распространенных библиотек, использующая шаблон MVVM – это библиотека backbone

9. Backbone

Backbone – MVVM (ModelView-ViewModel) библиотека. Backbone требует Underscore.js и jQuery. Если они не нужны можно использовать Exoskeleton – форк Backbone, где никаких зависимостей не нужно. Данную библиотеку рационально использвать в крупном и среднем проекте, который работает со множеством данных.

Объявление модели. Листинг 9.1

```
var Book = Backbone.Model.extend({})
```

Далее создадим коллекцию, где будут храниться все книги, и добавим одну книгу:

Объявление коллекции и добавление данных. Листинг 9.2

```
var Library = Backbone.Collection.extend({
  model: book
});
var library = new Library();
library.add({
  title: 'Игрок',
  author: 'Достоевский'
});
```

Запросом .fetch() можем получить данные:

Извлечение данных. Листинг 9.3

```
library.fetch();
```

Создадим представление для вывода коллекции на экран:

Вывод коллекции на экран. Листинг 9.4

```
var LibraryView = Backbone.View.extend({
  el: "#books", //существующий элемент на странице, где отобра-
жать.
  tagName: "li", //генерируемый тэг элементов
  className: "row", //имя класса для элемента
  template: '<p>Название: <%-title%><br /> Автор: <%-
author%></p>',
  initialize: function(){
   this.listenTo(this.model, "change", this.render);
  }
});
```

В каждом модуле backbone есть метод initialize, который является конструктором. В данном случае функция слушает изменения в модели, и обновляет шаблон.

Marionette.js (модульный фрэймворк)

Используя Backbone в реальных проектах, придется строить поверх библиотеки множество абстракций и привязок, а также разрабатывать собственную архитектуру приложения. Библиотека представляет набор базовых сущностей, а связывать их придется самому.

Но и здесь есть готовые решения: фрэймворк Marionette.js.

Marionette.js является модульным фрэймворком. Marionette использует вводит свои сущности: приложение, контроллеры, модули, субмодули, роутер, собственная система шаблонов, а также расширяет работу стандартных моделей.

10. CoffeeScript

CoffeeScript - язык программирования, транслируемый в JavaScript. CoffeeScript добавляет синтаксический сахар в духе Ruby, Python, Haskell и Erlang для того, чтобы улучшить читаемость кода и уменьшить его размер. CoffeeScript позволяет писать более компактный код по сравнению с JavaScript. JavaScript-код, получаемый трансляцией из CoffeeScript, полностью проходит проверку JavaScript Lint.

1. Переменные

Переменные. Листинг 10.1

```
//CoffeeScript:
age  = 2
male = true
name = "Матвей"

//JavaScript:
var age = 2,
    male = true,
    name = "Матвей";
```

2. Функции

Функции. Листинг 10.2

```
//CoffeeScript:
say = (speech) ->
  alert speech

say "Машина тютю!"

//JavaScript:
var say = function(speech) {
  alert(speech);
};
say("Машина тютю!");
```

3. Классы и объекты

Классы и объекты, JavaScript. Листинг 10.3

```
function Human(name){
  this.name = name;
}

function Baby(name){
  Human.call(this, name);
}

Baby.prototype = Object.create(Human.prototype);
Baby.prototype.say = function(msg){
  alert(this.name + ' говорит ' + msg);
};
```

```
Baby.prototype.sayHi = function(){
  this.say('Уууу!');
};
Baby.prototype.constructor = Baby;

var matt = new Baby("Матвей");
matt.sayHi();
```

Аналог на CoffeeScript:

Классы и объекты, CoffeeScript. Листинг 10.4

```
//CoffeeScript:

class Human
  constructor : (@name) ->

class Baby extends Human
  say   : (msg) -> alert "#{@name} говорит '#{msg}'"
  sayHi : -> @say('Уууу!')

matt = new Baby("Матвей")
matt.sayHi()
```

11. LESS

LESS — это динамический язык стилей. Он создан под влиянием языка стилей Sass, и, в свою очередь, оказал влияние на его новый синтаксис «SCSS», в котором также использован синтаксис, являющийся расширением CSS.

LESS — это продукт с открытым исходным кодом. Его первая версия была написана на Ruby, однако в последующих версиях было решено отказаться от использования этого языка программирования в пользу JavaScript. Less — это вложенный метаязык: валидный CSS будет валидной less-программой с аналогичной семантикой.

LESS обеспечивает следующие расширения CSS: переменные, вложенные блоки, миксины, операторы и функции.

LESS может работать на стороне клиента (Internet Explorer 6+, WebKit, Firefox) или на стороне сервера под управлением Node.js или Rhino.

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 LESS. Листинг 11.1

```
@color: #4D926F;

#header {
  color: @color;
}
h2 {
  color: @color;
}    name = "Матвей";
```

Данный код будет компилирован в следующий CSS-файл:

Компилированный CSS. Листинг 11.2

```
#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.

Примеси LESS. Листинг 11.3

```
.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 компилирован в следующий css-файл:

Компилированный CSS. Листинг 11.4

```
#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

LESS дает возможность вкладывать определения, вместо либо вместе с каскадированием. Пусть, например, у нас есть такой CSS: CSS поддерживает логическое каскадирование, но один блок кода в другой вложен быть не может. Less позволяет вложить один селектор в другой. Это делает наследование более ясным и укорачивает таблицы стилей.

Вложенные правила less. Листинг 11.5

```
 #header {
  h1 {
    font-size: 26px;
    font-weight: bold;
  }
  p { font-size: 12px;
    a { text-decoration: none;
```

```
      &:hover { border-width: 1px }
    }
  }
}
```

Данный LESS-код будет компилирован в следующий CSS-файл:

Компилированный CSS. Листинг 11.6

```
 #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 операции 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 less. Листинг 11.7

```
@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%);
}#header p a:hover {
  border-width: 1px;
}
```

Данный LESS-код будет скомпилирован в следующий CSS-файл:

Компилированный CSS-файл. Листинг 11.8

```
#header {
  color: #333;
  border-left: 1px;
  border-right: 2px;
```

```
}
#footer {
  color: #114411;
  border-color: #7d2717;
}
```

Сравнение с другими препроцессорами CSS

И Sass, и LESS — это препроцессоры CSS, позволяющие писать ясный CSS с использованием программных конструкций вместо статических правил.

LESS вдохновлен Sass. Sass был спроектирован с целью как упростить, так и расширить CSS, в первых версиях фигурные скобки были исключены из синтаксиса (этот синтаксис называется sass). LESS разработан с целью быть как можно ближе к CSS, поэтому у них идентичный синтаксис. В результате существующие CSS можно использовать в качестве LESS кода. Новые версии Sass также включают CSS-подобный синтаксис, который называется SCSS (Sassy CSS).

Также можно отметить препроцессор Stylus. Основная отличительная особенность Stylus – возможность получать значения свойств в контексте одного правила:

Stylus, получение значения свойств в контексте одного правила. Листинг 11.9

```
.test {
  height:200px;
  margin-top: -(@height/2)
}
```

Кроме того у Stylus есть собственная библиотека – Nib. В остальном Stylus похож на своих конкурентов: Sass и LESS.

На сервере

1) подключили less как модуль node.js

2) добавили через require(‘’) модуль в app.js

3) less.render(‘inputfile.less’, ‘outputfile.css’)

12. Системы сборки FrontEnd-a

Что делает система сборки FrontEnd-a:

- Конкатенирует все JS-файлы в один в нужном порядке.
- Контактенирует все CSS-файлы в один в нужном порядке.
- Проверяет все JS-файлы на валидность.
- Минимизирует CSS и JS-код (удаляет лишние пробелы и т.д), при необходимости делает его непонятным (обфусцирует код).
- Складывает файлы в отдельную дирректорию, из которой они и подключаются к HTML.

Часто эта простая схема усложняется дополнительными требованиями: оптимизация изображений, компиляция шаблонов, прогон тестов.

Существует несколько систем сборки для FrontEnd-разработчиков: Ant, Make, Grunt. Остановим свой выбор на Grunt.

Встроенные задачи Grunt:

concat — склеивание файлов;

min — минификация JS (UglifyJS);

lint — проверка JS (JSHint);

qunit — запуск тестов (QUnit);

watch — отслеживание изменений в файлах;

server — простой веб-сервер для статики;

init — инициализация проектов по шаблонам..

Преимущества Grunt:

Привычный язык: JS/Node.

Конфигурация отделена от реализациии.

Множество готовых задач.

Заточен на фронтенд.

Легко расширяется.

Установка

Для использования Grunt необходимо установить Node.js. Вместе с Node.js установится менеджер пакетов npm, который понадобится для установки модулей Node, в том числе самого Гранта и его плагинов.

Если вы уже пользуетесь предыдущей версией Grunt, то перед установкой её нужно удалить: npm uninstall -g grunt.

Удаление предыдущей версии Grunt. Листинг 12.1

```
npm uninstall -g grunt
```

Установим (ключ -g означает, что пакет будет установлен глобально), которая будет запускать Grunt, установленный в папке вашего проекта. Таким образом у каждого проекта будут свои версии Гранта и плагинов — можно не бояться, что при обновлении сборка поломается.

Глобальная установка Grunt. Листинг 12.2

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

Настройка

Теперь в папке проекта должно быть два файла:

- package.json — описание проекта для npm. Содержит список зависимостей (в нашем случае это Грант и его плагины) и позволяет потом устанавливать их все одной командой.
- Gruntfile.js или Gruntfile.coffee — файл конфигурации Grunt (грантфайл). (До версии 0.4 этот файл назывался grunt.js.)

package.json

package.json можно создать вручную или командой npm init. В нём есть два обязательных поля — имя проекта и версия. Если вы делаете сайт, а не библиотеку, то их содержимое не имеет значения:

Файл package.json с зависимостями grunt и необходимых для Grunt плагинов. Листинг 12.3

```
{
    "name": "Project name",
    "version": "0.0.1",
    "devDependencies" : {
        "chalk": "latest",
        "grunt" : "latest",
        "grunt-contrib-jshint" : "latest", //плагин jsHint, для
проверки кода на ошибки
        "grunt-contrib-concat" : "latest", // плагин
конкатенации js файлов
        "grunt-contrib-uglify" : "latest", //плагин минификации
js
        "grunt-contrib-cssmin" : "latest", // плагин минификации
и конкатенации css
        "grunt-contrib-watch"  : "latest", // плагин для отсле-
живания изменений файлов
        "grunt-remove-logging" : "latest" // плагин для удаления
логов.
    }
}
```

После того как мы заполнили package.json и выбрали какие плагины для гранта будем использовать, надо их установить простой командой консоли.

Установка зависимостей проекта. Листинг 12.4

```
npm install grunt --save-dev
```

Ключ --save-dev в дополнение к установке добавляет ссылку на пакет в package.json. Установить все зависимости, уже перечисленные в файле, можно командой npm install.

Gruntfile.js

Файл Grunt выглядит примерно так:

Пример файла Gruntfile.js. Листинг 12.5

```
// Обязательная обёртка
module.exports = function(grunt) {

    // Задачи
    grunt.initConfig({
        // Склеиваем js
        concat: {
            main: {
                src: [
                    'js/jquery.js',
                    'js/**/*.js'  // Все JS-файлы в папке
```

```
                 ],
                 dest: 'build/gruntscripts.js'
             }
         },
        // Склеиваем css
        cssmin: {
                 combine: {
                      files: {
                        'path/to/output.css':
               ['path/to/input_one.css', 'path/to/input_two.css']
                      }
                 }
               },
         // Сжимаем
         uglify: {
             main: {
                 files: {
                     // Результат задачи concat
                     'build/gruntscripts.min.js': '<%= con-
cat.main.dest %>'
                 }
             }
         }
    });

    // Загрузка плагинов, установленных с помощью npm install
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-cssmin');

    // Задача по умолчанию
    grunt.registerTask('default', ['concat', 'uglify',
'cssmin']);
};
```

С такой конфигурацией запускаем grunt, для этого в командной строке (находясь в папке с проектом) набираем следующую команду:

Запуск Grunt. Листинг 12.6

```
grunt
```

Таким образом будут созданы файлы gruntscripts.js и его минимизированная версия gruntscripts.min.js. А также файл output.css

С другими задачами grunt можно познакомиться на официальном сайте Grunt.

13. Bootstrap 3, API Bootstrap

Bootstrap - интуитивно простой и в тоже время мощный интерфейсный фрейморк, повышающий скорость и облегчающий разработку web-приложений.

Скачать bootstrap можно по ссылке: http://bootstrap-3.ru/customize.php

Настраиваем bootsrap, находим кнопку

Компилировать и скачать

и скачиваем фрэймворк.

После загрузки, распаковываем файлы. И видим нечто похожее на это:

```
bootstrap/
├── css/
│   ├── bootstrap.css
│   ├── bootstrap.min.css
│   ├── bootstrap-theme.css
│   └── bootstrap-theme.min.css
├── js/
│   ├── bootstrap.js
│   └── bootstrap.min.js
└── fonts/
    ├── glyphicons-halflings-regular.eot
    ├── glyphicons-halflings-regular.svg
    ├── glyphicons-halflings-regular.ttf
    └── glyphicons-halflings-regular.woff
```

Это основная форма Bootstrap: скомпилированные файлы готовы для быстрого использования в любом веб-проекте. Предоставлены сборки CSS и JS (bootstrap.*) и минимизированный вариант (bootstrap.min.*). В качестве дополнительной опции для Bootstrap включены шрифты Glyphicons.

Какую версию использовать, простую (bootstrap.js и bootstrap.css) или минимизированную (bootstrap.min.js и bootstrap.min.css)?

Если вы не планируете читать CSS, выбирайте их минимизированную версию. Это тот же код, просто компактнее. Минимизированы таблицы стилей используют меньшую ширину канала, что есть хорошо, особенно в рабочем (production) среде. Поэтому, если нет особых причин использовать простую, останавливаемся на минимизированной версии.

Контейнеры

Простое центрирование контента страницы включая это содержимое в .container. Контейнер установлен в width на различных контрольных точках медиа запросов для соответствия с нашей системой разметки.

Следует отметить, что, благодаря padding и фиксированной ширины, контейнеры не вложены по умолчанию.

Контейнер с классом container. Листинг 13.1

```
<div class="container">
  ...
</div>
```

Превратить любую фиксированную ширину сетки макет в полную ширину макета, можно изменив последний .container на .container-fluid.

Пример резинового макета. Листинг 13.2

```
<div class="container-fluid">
  <div class="row">
    ...
  </div>
</div>
```

Используя единичный набор .col-md-* классовой разметки, вы можете создать базовую систему разметки, которая запускается одновременно на мобильные устройства и планшетные устройства (от экстра маленьких до малых диапазонов) до того как выстроится горизонталь на рабочем столе (средних) устройств. Разместите столбцы разметки в любом .row.

Использование col-md-*. Листинг 13.3

```
<div class="row">
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
</div>
<div class="row">
  <div class="col-md-8">.col-md-8</div>
  <div class="col-md-4">.col-md-4</div>
</div>
<div class="row">
  <div class="col-md-4">.col-md-4</div>
  <div class="col-md-4">.col-md-4</div>
```

```
  <div class="col-md-4">.col-md-4</div>
</div>
<div class="row">
  <div class="col-md-6">.col-md-6</div>
  <div class="col-md-6">.col-md-6</div>
</div>
```

Элементы распределятся следующим образом:

.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1

.col-md-8	.col-md-4

.col-md-4	.col-md-4	.col-md-4

.col-md-6	.col-md-6

На устройствах с небольшим размером экрана, увидим следующее:

Не хотите чтобы ваши колонки просто складывались на небольших устройствах? Используйте очень маленькие xs или средние md классы разметки устройства добавляя .col-xs-* .col-md-* к вашим столбцам. Смотрите пример ниже для лучшего понимания как это работает.

Использование col-xs-*. Листинг 13.4

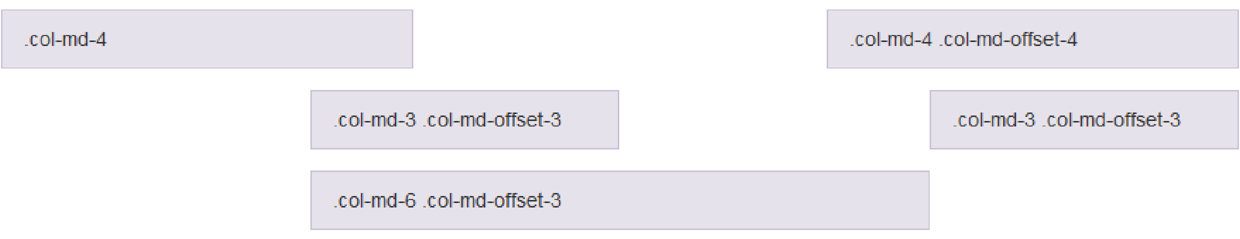
```
<div class="row">
  <div class="col-xs-12 col-md-8">.col-xs-12 .col-md-8</div>
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
</div>
<div class="row">
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
</div>
<div class="row">
  <div class="col-xs-6">.col-xs-6</div>
  <div class="col-xs-6">.col-xs-6</div>
</div>
```

Смещенные колонки

Переместить колонки направо с помощью .col-md-offset-* класса. Эти классы увеличивают отступ слева столбца * колонки. Например, .col-md-offset-4 сдвигает .col-md-4 пропуская один такой же столбец

Смещенные колонки. Листинг 13.5

```
<div class="row">
  <div class="col-md-4">.col-md-4</div>
  <div class="col-md-4 col-md-offset-4">.col-md-4 .col-md-
offset-4</div>
</div>
<div class="row">
  <div class="col-md-3 col-md-offset-3">.col-md-3 .col-md-
offset-3</div>
  <div class="col-md-3 col-md-offset-3">.col-md-3 .col-md-
offset-3</div>
</div>
<div class="row">
  <div class="col-md-6 col-md-offset-3">.col-md-6 .col-md-
offset-3</div>
</div>
```



Вложенные столбцы

Чтобы вложить ваше содержание в разметку, необходимо добавить новый .row и набор .col-md-* столбцов в существующую .col-md-* колонку. Вложенные строки должны включать в себя набор столбцов, которые добавляются до 12 или менее.

Вложенные столбцы. Листинг 13.6

```
 <div class="row">
  <div class="col-md-9">
    Level 1: .col-md-9
    <div class="row">
      <div class="col-md-6">
        Level 2: .col-md-6
      </div>
      <div class="col-md-6">
        Level 2: .col-md-6
      </div>
    </div>
  </div>
</div>
```

Level 1: .col-md-9

Level 2: .col-md-6	Level 2: .col-md-6

Таблицы

Для базовой стилизации—легкие отступы и только горизонтальные разделители—добавив базовые классы .table для любых <table>. Это может показаться избыточным, но учитывая широкое распространение использование таблиц для других плагинов как календари и выбор дат, мы решили изолировать пользовательские стили таблицы.

Класс table. Листинг 13.7

```
<table class="table">
  ...
</table>
```

Используйте .table-striped, чтобы добавить зебру- чередование для любой строки таблицы внутри <tbody>.

Зебра таблицы. Листинг 13.8

```
<table class="table table-striped">
  ...
</table>
```

#	Имя	Фамилия	Пользователь
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

Добавьте .table-bordered для границы со всех сторон таблицы и ячеек.

Добавляем границы к таблице. Листинг 13.9

```
<table class="table table-bordered">
  ...
</table>
```

#	Имя	Фамилия	Пользователь
1	Mark	Otto	@mdo
	Mark	Otto	@TwBootstrap
2	Jacob	Thornton	@fat
3	Larry the Bird		@twitter

Контекстные классы:

.active Применяет цвет при наведении на конкретную строку или ячейку

.success Указывает на успешное или позитивное действие

.info Указывает на нейтральные информативные изменения или действия

.warning Указывает на предупреждения, которые могут потребовать внимания

.danger Указывает на опасное или потенциально негативное действие

Контексные классы таблицы. Листинг 13.10

```
<!—Для элементов tr-->
<tr class="active">...</tr>
<tr class="success">...</tr>
<tr class="warning">...</tr>
<tr class="danger">...</tr>
<tr class="info">...</tr>

<!—Для ячеек (`td` или `th`) -->
<tr>
  <td class="active">...</td>
  <td class="success">...</td>
  <td class="warning">...</td>
  <td class="danger">...</td>
  <td class="info">...</td>
</tr>
```

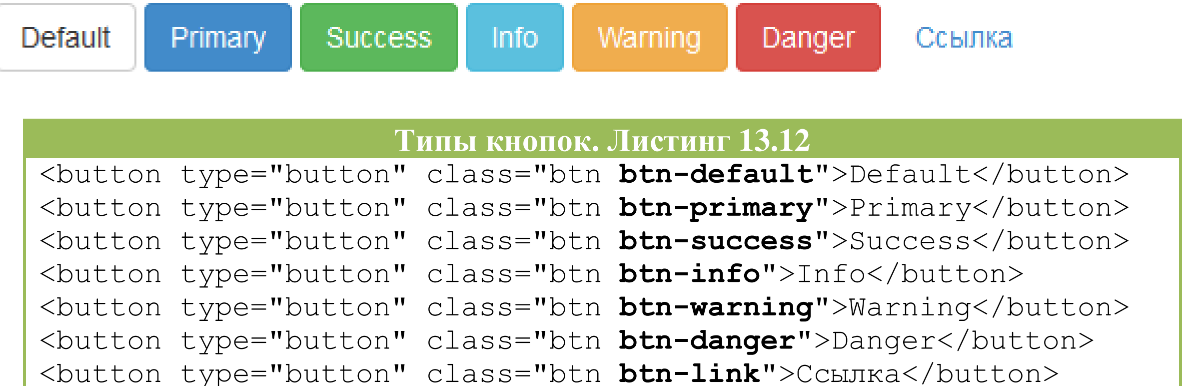
1	Содержимое столбца	Содержимое столбца	Содержимое столбца
2	Содержимое столбца	Содержимое столбца	Содержимое столбца
3	Содержимое столбца	Содержимое столбца	Содержимое столбца
4	Содержимое столбца	Содержимое столбца	Содержимое столбца
5	Содержимое столбца	Содержимое столбца	Содержимое столбца
6	Содержимое столбца	Содержимое столбца	Содержимое столбца
7	Содержимое столбца	Содержимое столбца	Содержимое столбца
8	Содержимое столбца	Содержимое столбца	Содержимое столбца
9	Содержимое столбца	Содержимое столбца	Содержимое столбца

Можно создать адаптивные таблицы путем преобразования любого .table в .table-responsive, чтобы сделать их прокрученными горизонтально для небольших экранов (до 768px). При просмотре на экране при разрешении больше чем 768px, вы не увидите никакой разницы в этих таблицах.

Создание адаптивной таблицы. Листинг 13.11

```
<div class="table-responsive">
  <table class="table">
    ...
  </table>
</div>
```

Кнопки



Типы кнопок. Листинг 13.12

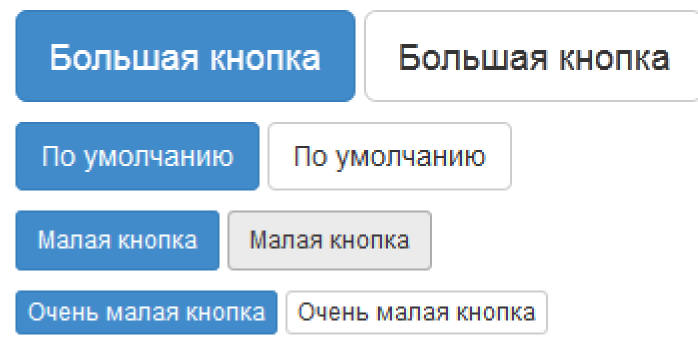
```
<button type="button" class="btn btn-default">Default</button>
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-link">Ссылка</button>
```

Используем .btn-lg, .btn-sm, или .btn-xs для создания кнопок разных размеров

Размеры кнопок. Листинг 13.13

```
<p>
    <button type="button" class="btn btn-primary btn-lg">Большая
кнопка</button>
    <button type="button" class="btn btn-default btn-lg">Большая
кнопка</button>
    </p>
<p>
    <button type="button" class="btn btn-primary">По
```

```
умолчанию</button>
    <button type="button" class="btn btn-default">По
умолчанию</button>
    </p>
<p>
    <button type="button" class="btn btn-primary btn-sm">Малая
кнопка</button>
    <button type="button" class="btn btn-default btn-sm">Малая
кнопка</button>
    </p>
<p>
      <button type="button" class="btn btn-primary btn-xs">Очень
малая кнопка</button>
      <button type="button" class="btn btn-default btn-xs">Очень
малая кнопка</button>
    </p>
```

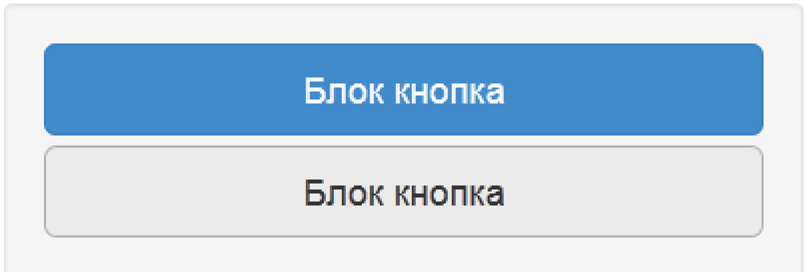


Резиновая кнопка:

Резиновая кнопка. Листинг 13.14

```
<button type="button" class="btn btn-primary btn-lg btn-
block">Блок кнопка</button>
<button type="button" class="btn btn-default btn-lg btn-
block">Блок кнопка</button>
```

Получим:



Чтобы кнопку заблокировать с добавлением прозрачности 50%, добавим атрибут disabled:

Неактивное состояние кнопки. Листинг 13.15

```
<button type="button" class="btn btn-lg btn-primary" disa-
bled="disabled">Главная кнопка</button>
<button type="button" class="btn btn-default btn-lg" disa-
```

```
bled="disabled">Кнопка</button>
```

Изображения

Изображения в Bootstrap 3 адаптируюся с помощью добавления класса .img-responsive. Это касается max-width: 100%; и height: auto; к изображению, чтобы он хорошо масштабировался к родительскому элементу.

Адаптивные изображения. Листинг 13.16

```
<button type="button" class="btn btn-lg btn-primary" disa-
bled="disabled">Главная кнопка</button>
<button type="button" class="btn btn-default btn-lg" disa-
bled="disabled">Кнопка</button>
```

Фигурные изображения:

Адаптивные изображения. Листинг 13.17

```
<img src="..." alt="..." class="img-rounded">
<img src="..." alt="..." class="img-circle">
<img src="..." alt="..." class="img-thumbnail">
```

140x140 140x140 140x140

Вспомогательные классы

Обработка текста. Листинг 13.18

```
<p class="text-muted">...</p>
<p class="text-primary">...</p>
<p class="text-success">...</p>
<p class="text-info">...</p>
<p class="text-warning">...</p>
<p class="text-danger">...</p>
```

Fusce dapibus, tellus ac cursus commodo, tortor mauris nibh.

Nullam id dolor id nibh ultricies vehicula ut id elit.

Duis mollis, est non commodo luctus, nisi erat porttitor ligula.

Maecenas sed diam eget risus varius blandit sit amet non magna.

Etiam porta sem malesuada magna mollis euismod.

Donec ullamcorper nulla non metus auctor fringilla.

Как и цвет текста контекстных классов, легко устанавливать фон элемента к любому контекстному классу. Якорные компоненты будут темнеть при наведении, как и текстовые классы.

Добавление фона. Листинг 13.19

```
<p class="bg-primary">...</p>
<p class="bg-success">...</p>
<p class="bg-info">...</p>
<p class="bg-warning">...</p>
<p class="bg-danger">...</p>
```

Nullam id dolor id nibh ultricies vehicula ut id elit.

Duis mollis, est non commodo luctus, nisi erat porttitor ligula.

Maecenas sed diam eget risus varius blandit sit amet non magna.

Etiam porta sem malesuada magna mollis euismod.

Donec ullamcorper nulla non metus auctor fringilla.

Чтобы указать выпадающую функциональность и направление используем .caret. Обратите внимание, что курсор по умолчанию автоматически изменится в dropup меню.

Класс caret для выпадающих списков. Листинг 13.20

```
<span class="caret"></span>
```

Значки закрытия

Класс close. Листинг 13.21

```
<button type="button" class="close" aria-hidden="true"> &times;
</button>
```

×

Чтобы скрыть или отобразить элемент, в том числе, для программ чтения с экрана, используйте классы .show и .hidden

Скрытие и отображение элементов. Листинг 13.22

```
<div class="show">...</div> //отображить
<div class="hidden">...</div> //скрыть
```

Формы

Наиболее распространенные формы управления, текстовые поля ввода включают поддержку для всех типов HTML5 : text, text, datetime, datetime-local, date, month, time, week, number, email, url, search, tel, и color.

Для элементов форм можно использовать специальный класс form-control:

Использование класса form-control. Листинг 13.23

```
<input type="text" class="form-control" placeholder="Text in-
put">
```

По умолчанию форма растягивается на 100% возможной ширины экрана и меняет свой внешний вид.

Text input

Задавать высоту с помощью классов .input-lg, и задавать ширину с использованием классов столбцовой разметки, как .col-lg-*.

Высота и ширина форм. Листинг 13.24

```
<input class="form-control input-lg" type="text" placehold-
er=".input-lg">
<input class="form-control" type="text" placeholder="Default in-
put">
<input class="form-control input-sm" type="text" placehold-
er=".input-sm">

<select class="form-control input-lg">...</select>
<select class="form-control">...</select>
<select class="form-control input-sm">...</select>
```

Простой пример формы:

Простой пример формы. Листинг 13.25

```
<form role="form">
  <div class="form-group">
    <label for="exampleInputEmail1">Email</label>
    <input type="email" class="form-control"
id="exampleInputEmail1" placeholder="Enter email">
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Пароль</label>
    <input type="password" class="form-control"
id="exampleInputPassword1" placeholder="Password">
  </div>
  <div class="form-group">
    <label for="exampleInputFile">File input</label>
    <input type="file" id="exampleInputFile">
    <p class="help-block">Example block-level help text
here.</p>
  </div>
  <div class="checkbox">
    <label>
```

```
          <input type="checkbox"> Проверить меня
        </label>
  </div>
  <button type="submit" class="btn btn-
default">Отправить</button>
</form>
```

внутри формы вместо класса row лучше использвать form-group.

Email

Enter email

Пароль

Password

File input

Обзор... Файл не выбран.

Example block-level help text here.

Проверить меня

Отправить

Для элемента form также существует два специальных класса: form-inline (выравнивание по левому краю) и form-horizontal (выравнивание по горизонтальному направлению).

Класс form-horizontal. Листинг 13.26

```
<form class="form-horizontal" role="form">
```

Bootstrap включает в себя проверку стилей на ошибки, предупреждения и успех положений на формы управления. Для использования, добавьте .has-warning, .has-error, или .has-success к исходному элементу. Любой .control-label, .form-control и .help-block внутри этого элемента получит подтверждение стилей.

Классы form-group и form-horizontal. Листинг 13.27

```
<div class="form-group has-success">
  <label class="control-label" for="inputSuccess1">
    Успешный ввод</label>
  <input type="text" class="form-control" id="inputSuccess1">
</div>
<div class="form-group has-warning">
  <label class="control-label" for="inputWarning1">
    Ввод с предупреждением</label>
  <input type="text" class="form-control" id="inputWarning1">
</div>
<div class="form-group has-error">
```

```
  <label class="control-label" for="inputError1">
    Ввод с ошибкой</label>
  <input type="text" class="form-control" id="inputError1">
</div>
```

Успешный ввод

Ввод с предупреждением

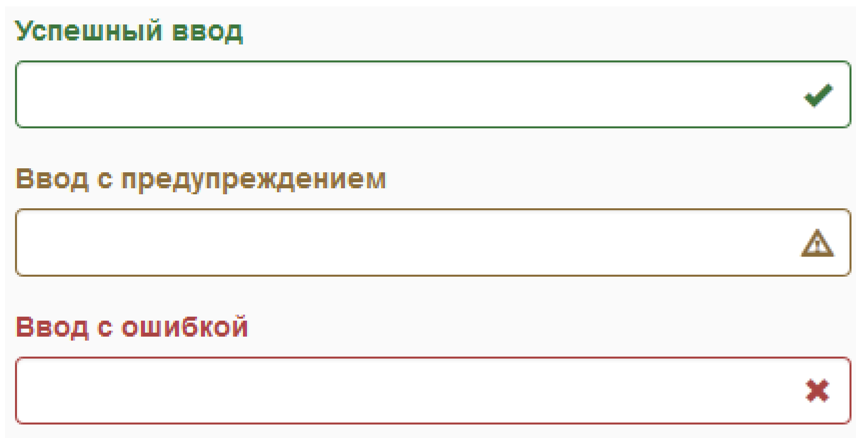
Ввод с ошибкой

Вывод элементов форм с дополнительными иконками:

Элементы форм с дополнительными иконками. Листинг 13.28

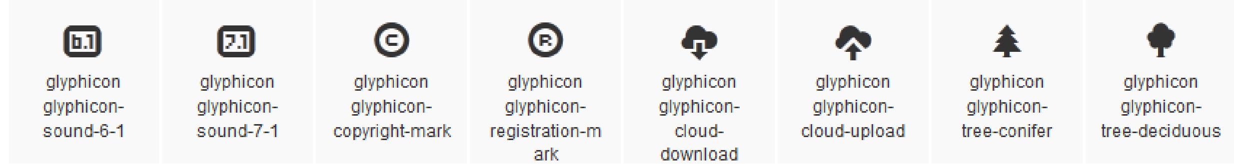
```
<div class="form-group has-success has-feedback">
  <label class="control-label" for="inputSuccess2">Успешный
ввод</label>
  <input type="text" class="form-control" id="inputSuccess2">
  <span class="glyphicon glyphicon-ok form-control-
feedback"></span>
</div>
<div class="form-group has-warning has-feedback">
  <label class="control-label" for="inputWarning2">Ввод с
предупреждением</label>
  <input type="text" class="form-control" id="inputWarning2">
  <span class="glyphicon glyphicon-warning-sign form-control-
feedback"></span>
</div>
<div class="form-group has-error has-feedback">
  <label class="control-label" for="inputError2">Ввод с
ошибкой</label>
  <input type="text" class="form-control" id="inputError2">
  <span class="glyphicon glyphicon-remove form-control-
feedback"></span>
</div>
```

Получим следующее:



Иконки

Bootstrap содержит более 200 видов графических элементов. Ниже представлены только некоторые из них.

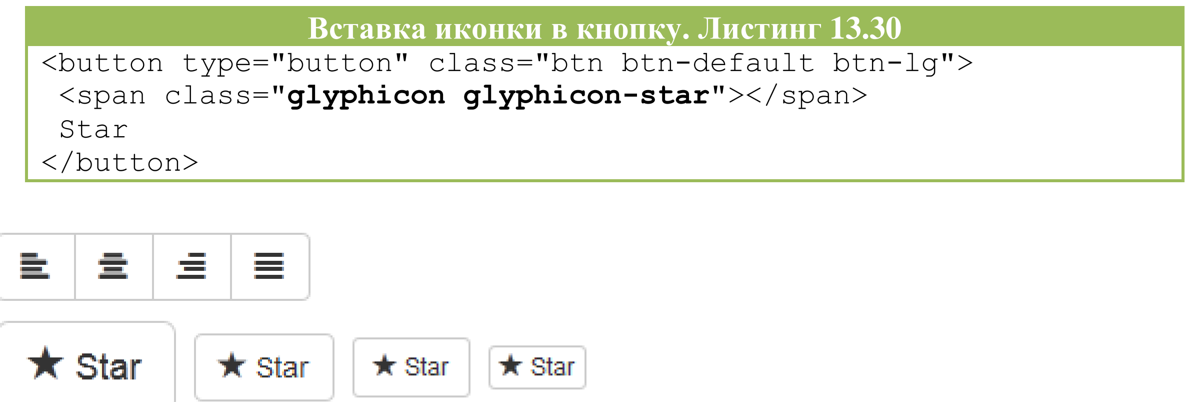


У каждого графического элемента имеется свой класс. Пример использования:

Элементы форм с дополнительными иконками. Листинг 13.29

```
<span class="glyphicon glyphicon-search"></span>
```

Классы иконок спроектированы так, что они не могут смешиваться с другими классами.



Вставка иконки в кнопку. Листинг 13.30

```
<button type="button" class="btn btn-default btn-lg">
 <span class="glyphicon glyphicon-star"></span>
 Star
</button>
```


Навигация

Доступна в Bootstrap навигация имеет общую разметку, начиная с базового класса .nav

Навигационные вкладки. Листинг 4.13.31

```
<ul class="nav nav-tabs">
  <li class="active"><a href="#">Главная</a></li>
  <li><a href="#">Профиль</a></li>
  <li><a href="#">Сообщение</a></li>
</ul>
```

Главная Профиль Сообщение

Для навигационных кнопок используем .nav-pills вместо .nav-tabs:

Навигационные кнопки. Листинг 13.32

```
<ul class="nav nav-pills">
 …
</ul>
```

Навигационные кнопки с выпадающим меню:

Навигационные кнопки с выпадающим меню. Листинг 13.33

```
<ul class="nav nav-pills">
      <li class="active"><a href="#">Главная</a></li>
      <li><a href="#">Помощь</a></li>
      <li class="dropdown">
      <a href="#" data-toggle="dropdown" class="dropdown-
toggle"> Выпадающее меню
        <span class="caret"></span>
      </a>
   <ul role="menu" class="dropdown-menu">
     <li><a href="#">Действие</a></li>
     <li><a href="#">Другое действие</a></li>
     <li><a href="#">Что-то еще</a></li>
     <li class="divider"></li>
     <li><a href="#">Отдельная ссылка</a></li>
    </ul>
   </li>
 </ul>
```

Главная Помощь Выпадающее меню

Хлебные крошки

Для реализации текущего навигационного маршрута (хлебные крошкп) можно использовать класс breadcrumb:

Навигационные кнопки с выпадающим меню. Листинг 13.34

```
<ol class="breadcrumb">
  <li><a href="#">Главная</a></li>
  <li><a href="#">Библиотека</a></li>
  <li class="active">Данные</li>
</ol>
```

Главная / Библиотека / Данные

JavaScript

Кроме css-компонентов, bootstrap содержит множество js-плагинов. Причем, всем JavaScript плагинам необходима библиотека jQuery. Рассмотрим один из плагинов JavaScript, выпадающий список:

HTML-код для выпадающиего списка. Листинг 13.35

```
<div class="dropdown">
  <a id="dLabel" role="button" data-toggle="dropdown" data-target="#" href="/page.html">
    Dropdown <span class="caret"></span>
  </a>
  <ul class="dropdown-menu" role="menu" arialabelledby="dLabel">
    ...
  </ul>
</div>
```

js-код для выпадающего меню:

Использование метода dropdown. Листинг 13.36

```
$('.dropdown-menu').dropdown()
```

Со множеством других js-плагинов и примерами их использования можно ознакомиться на официальном сайте Bootstrap 3.

Исходный код Bootstrap

Исходный код Bootstrap включает прекомпелированные CSS, JavaScript и шрифты, вместе с исходным Less, JavaScript и документацией.

```
bootstrap/
├── less/
├── js/
├── fonts/
├── dist/
│   ├── css/
│   ├── js/
│   └── fonts/
└── docs/
    └── examples/
```

Bootstrap's CSS построен на Less, препроцессор с дополнительной функциональностью, как переменных, mixins, и функции для компиляции CSS. Тем, кто хочет использовать источник Less файлы вместо наших скомпилированных файлов CSS могут воспользоваться многочисленными переменными и mixins которые мы используем во всем фреймверке.

Переменные less

Переменные определяют число столбцов, ширину промежутка, и точку медиа запроса, с которого начинается перемещение столбцов. Мы используем это для генерации предопределенных классов разметки задокументированных выше, также как для пользовательского смешения перечисленного ниже.

Переменные less. Листинг 13.37

```
@grid-columns:              12;
@grid-gutter-width:         30px;
@grid-float-breakpoint:     768px;
```

Переменные используются на протяжении всего проекта, как способ централизации и обмена часто используемых значений как цвета, отступы, или стеки шрифта.

Переменные цвета. Листинг 13.38

```
@brand-primary: #428bca;
@brand-success: #5cb85c;
@brand-info:    #5bc0de;
@brand-warning: #f0ad4e;
@brand-danger:  #d9534f;
```

Использование цветовых переменных:

Использование переменных цвета. Листинг 13.39

```
// Use as-is
.masthead {
  background-color: @brand-primary;
}

// Reassigned variables in Less
@alert-message-background: @brand-info;
.alert {
  background-color: @alert-message-background;
}
```

Less Mixins

Смешения (или mixins) используются в сочетании с переменных разметкок, чтобы образовать семантические CSS для отдельных столбцов разметки.

Рассмотрим использование миксинов на примере теней:

Использование миксинов в стилях. Листинг 13.40

```
.box-shadow(@shadow: 0 1px 3px rgba(0,0,0,.25)) {
  -webkit-box-shadow: @shadow; // iOS <4.3 & Android <4.1
          box-shadow: @shadow;
}
```

Базовый шаблон Bootstrap

Начать использовать Bootsrap можно с базового HTML шаблона:

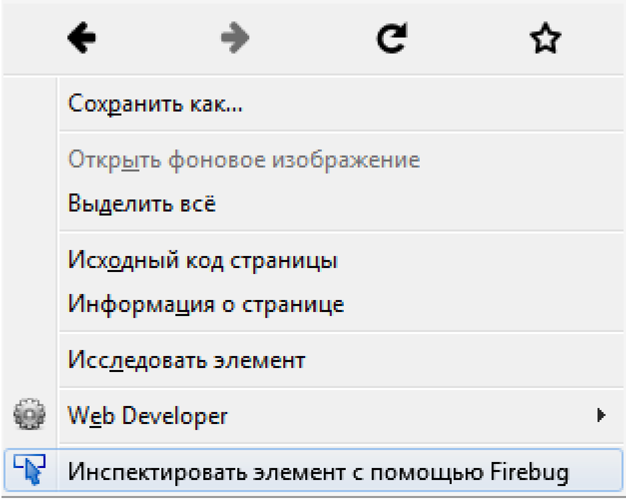
Базовый шабон Bootstrap. Листинг 13.41

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <title>Bootstrap 101 Template</title>
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <!--[if lt IE 9]>
      <script
src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js">
</script>
      <script
src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js">
</script>
    <![endif]-->
  </head>
  <body>
    <h1>Привет, мир!</h1>
    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min
.js"></script>
    <!-- Include all compiled plugins (below), or include individu-
al files as needed -->
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

14. Браузерные web-консоли. FireBug

FireBug – это плагин для FireFox, предназначен для редактирования, выполнеыния отладки и просмотра CSS, HTML и JavaScript любой страницы в сети. На сегодняшний день, FireBug – самый восстребованный инструмент web-разработчика. Кроме FireBug для FireFox, можно воспользоваться встроенной web-консолью Chrome, или Opera.

https://addons.mozilla.org/ru/firefox/addon/firebug/ - по этому адресу можно установить firebug. После чего обновляем страницу, и нажимаем F12. Либо правой кнопкой мыши кликаем по любому месту страницы, и в выпадающем списке находим фразу "Инспектировать элемент с помощью firebug".



Специфика FireBug:

1. Консоль

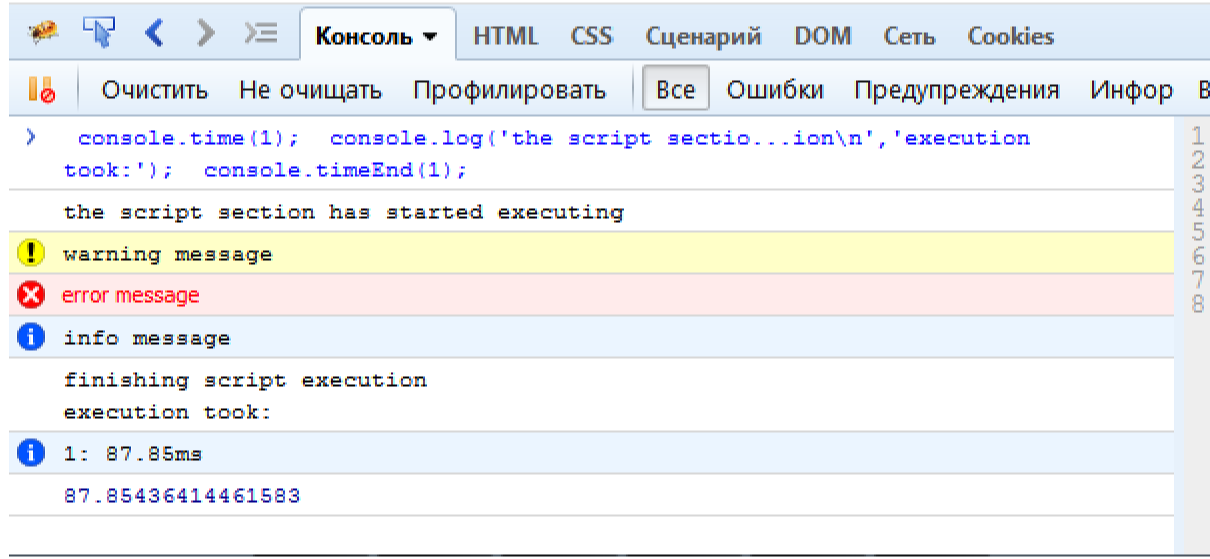
Первая панель FireBug – это консоль, которая умеет выполнять следующие вещи: а) отображать ошибки, предупреждения и дополнительную информацию, б) выполнять код JavaScript в) т.к. консоль видит все подключения текущей страницы, то консоль будет понимать синтаксис подключаемых библиотек, например, jQuery и других.

Использование консоли для отладки приложения:

Консольные функции. Листинг 14.1

```
console.time(1);
console.log('the script section has started executing');
console.warn('warning message');
console.error('error message');
console.info('info message');
console.log('finishing script execution\n','execution took:');
console.timeEnd(1);
```

Следующий ответ консоли:



2. HTML

Вторая вкладка – HTML. Разделена на две части. Слева видим html-код текущей страницы, справа стили, применяемые к выбранному html-элементу.

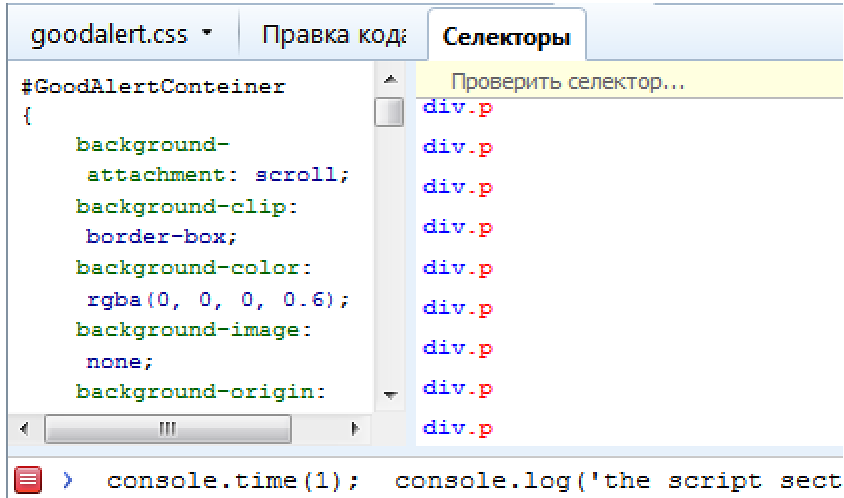
Редактировать ul Стиль Скомпилированный стиль Макет DOM События

```
overflow-x: hidden; m
<script>
<h1 style="margin-top
использовать Firebug<
<div class="les">
  <p>
  <h2>
  <p>
  <ul>
  <p>
  <ul>
```

```
pre, blockquote, ul, ol, dl, address, p   bookmar...5957431 (строка 29)
    margin-bottom: 1em;
    margin-left: 0;
    margin-right: 0;
    margin-top: 0;
}
ul {                                       bookmar...5957431 (строка 4)
    list-style-image: none;
    list-style-position: outside;
    list-style-type: none;
}
h1, h2, h3, h4, h5, h6, ul, li, p, em,     bookmar...5957431 (строка 3)
```

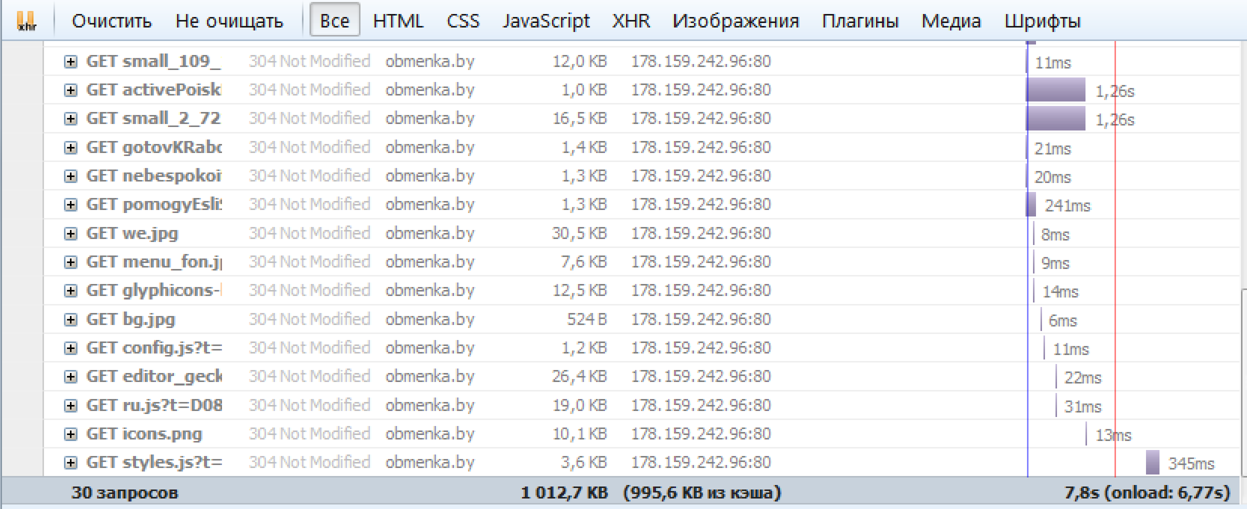
3. CSS

Во вкладке CSS можно ознакомиться с полным набором стилей, и найти стили для используемых на странице селекторов.



4. Сеть

Информацию по загрузки всех странице, и отдельно каждого компонента (get, post-параметры, изображения, скрипты, стили, шрифты и другие медийные файлы) можно найти во вкладке Сеть.



Запрос	Статус	Домен	Размер	Адрес	Время
GET small_109_	304 Not Modified	obmenka.by	12,0 KB	178.159.242.96:80	11ms
GET activePoisk	304 Not Modified	obmenka.by	1,0 KB	178.159.242.96:80	1,26s
GET small_2_72	304 Not Modified	obmenka.by	16,5 KB	178.159.242.96:80	1,26s
GET gotovKRabo	304 Not Modified	obmenka.by	1,4 KB	178.159.242.96:80	21ms
GET nebespokoi	304 Not Modified	obmenka.by	1,3 KB	178.159.242.96:80	20ms
GET pomogyEsli	304 Not Modified	obmenka.by	1,3 KB	178.159.242.96:80	241ms
GET we.jpg	304 Not Modified	obmenka.by	30,5 KB	178.159.242.96:80	8ms
GET menu_fon.j	304 Not Modified	obmenka.by	7,6 KB	178.159.242.96:80	9ms
GET glyphicons-	304 Not Modified	obmenka.by	12,5 KB	178.159.242.96:80	14ms
GET bg.jpg	304 Not Modified	obmenka.by	524 B	178.159.242.96:80	6ms
GET config.js?t=	304 Not Modified	obmenka.by	1,2 KB	178.159.242.96:80	11ms
GET editor_geck	304 Not Modified	obmenka.by	26,4 KB	178.159.242.96:80	22ms
GET ru.js?t=D08	304 Not Modified	obmenka.by	19,0 KB	178.159.242.96:80	31ms
GET icons.png	304 Not Modified	obmenka.by	10,1 KB	178.159.242.96:80	13ms
GET styles.js?t=	304 Not Modified	obmenka.by	3,6 KB	178.159.242.96:80	345ms
30 запросов			1 012,7 KB	(995,6 KB из кэша)	7,8s (onload: 6,77s)

5. Cookie

Всю информацию по кукам можно получить в вкладке Cookie

Имя	Значение	Домен	Исходный размер	Путь	Истекает	Только HTTP	Безопасность
__utma	164318880.53	.habrahabr.ru	61 B	/	11.04.2016, 16:21:23		
__utmz	164318880.13	.habrahabr.ru	102 B	/	12.10.2014, 4:21:23		
_ga	GA1.2.531638	.habrahabr.ru	29 B	/	14.09.2016, 13:11:50		
_gat	1	.habrahabr.ru	5 B	/	15.09.2014, 13:21:50		
_ym_visorc_24049213	b	.habrahabr.ru	20 B	/	15.09.2014, 13:41:58		
tmid_no_check	1	habrahabr.ru	14 B	/post/14	15.09.2014, 13:21:58		
vbn	3	.habrahabr.ru	4 B	/	02.05.2015, 23:12:56		

15. Schema.org

Schema.org – это стандарт семантической разметки данных в сети, объявленный поисковыми системами Google, Bing и Yahoo! летом 2011 года. Цель семантической разметки – сделать интернет более понятным, структурированным и облегчить поисковым системам и специальным программам извлечение и обработку информации для удобного её представления в результатах поиска. Яндекс с осени 2011 года понимает этот формат и поддерживает его в некоторых партнерских программах. Разметка происходит непосредственно в HTML-коде страниц с помощью специальных атрибутов и не требует создания отдельных экспортных файлов.

Микроданные (HTML microdata) — это международный стандарт семантической разметки HTML-страниц, с помощью атрибутов, описывающих смысл информации, содержащейся в тех или иных HTML-элементах. Такие атрибуты делают контент страниц машиночитаемым, то есть позволяют в автоматическом режиме находить и извлекать нужные данные.

Что такое Schema.org?

Это сайт, который содержит коллекцию схем (html-тегов), которые вебмастера могут использовать для разметки своих страниц способами, признаными крупнейшими поисковыми системами, такими как Bing, Google, Yahoo! и Яндекс, которые полагаются на эту разметку для улучшения отображения результатов поиска, делая процесс поиска правильных веб-страниц проще для людей.

Многие сайты генерируются из структурированных данных, которые чаще всего хранятся в БД. Когда данные переводятся в HTML, становится очень сложно восстановить первоначальные структурированные данные. Многие приложения, особенно поисковые системы, могут получить значительные преимущества, имея прямой доступ к структурированным данным. Разметка страниц позволяет поисковым системам понимать информацию на веб-страницах и предоставлять пользователю именно ту информацию, которую он ищет. Разметка также может задействовать новые инструменты и приложения, которые используют эту структуру.

Этот общий словарь облегчит вебмастерам выбор правильной схемы разметки и поможет получить максимум выгоды из приложенных усилий. Так, воодушевленные sitemaps.org, Bing, Google and Yahoo! собрались вместе чтобы создать для вебмастеров эту коллекцию схем.

Как использовать?

Начнем с конкретного примера. Представим, что у нас есть страница о фильме «Аватар» — со ссылкой на трейлер, информацией о режиссере и т. п. HTML-код может выглядеть примерно так:

Пример html-страницы. Листинг 15.1

```
<div>
 <h1>Аватар</h1>
 <span>Режиссер: Джеймс Кэмерон (род. 16 августа 1954 г.)</span>
 <span>Фантастика</span>
 <a href="/../movies/avatar-theatrical-trailer.html">Трейлер</a>
</div>
```

В первую очередь необходимо указать, какая часть страницы посвящена непосредственно фильму «Аватар». Для этого добавим атрибут itemscope к HTML-тегу, в который заключена эта информация:

Атрибут itemscope. Листинг 15.2

```
<div itemscope>
  <h1>Аватар</h1>
  <span>Режиссер: Джеймс Кэмерон (род. 16 августа 1954 г.)
</span>
  <span>Фантастика</span>
  <a href="/../movies/avatar-theatrical-
trailer.html">Трейлер</a>
</div>
```

Добавляя itemscope, мы тем самым обозначаем, что HTML-код, содержащийся в блоке <div>...</div>, описывает некоторую сущность.

Пока мы только объявили, что речь идет о какой-то сущности, но не сообщили, что это за сущность. Чтобы указать тип сущности, добавим атрибут itemtype сразу после itemscope.

Атрибут itemtype. Листинг 15.3

```
<div itemscope itemtype="http://schema.org/Movie">
  <h1>Аватар</h1>
  <span>Режиссер: Джеймс Кэмерон (род. 16 августа 1954
г.)</span>
  <span>Фантастика</span>
  <a href="/../movies/avatar-theatrical-
trailer.html">Трейлер</a>
</div>
```

Тем самым мы уточняем, что сущность, описание которой заключено в теге <div>, представляет собой фильм (тип Movie в иерархии типов schema.org). Названия типов имеют вид URL, в нашем случае http://schema.org/Movie.

Какую дополнительную информацию о фильме «Аватар» можно предоставить поисковым системам? О фильме можно сообщить множество интересных сведений: актерский состав, режиссер, рейтинг. Чтобы отметить свойства сущности, используется атрибут itemprop. Например, чтобы указать ре-

жиссера фильма, добавим атрибут itemprop="director" к HTML-тегу, содержащему имя режиссера. (Полный список свойств, которые можно задать для фильма, приведен на странице http://schema.org/Movie.)

Атрибут itemprop. Листинг 15.4

```
<div itemscope itemtype="http://schema.org/Movie">
  <h1 itemprop="name">Аватар</h1>
  <span>Режиссер: <span itemprop="director">Джеймс
Кэмерон</span> (род. 16 августа 1954 г.)</span>
  <span itemprop="genre">Фантастика</span>
  <a href="/../movies/avatar-theatrical-trailer.html" item-
prop="trailer">Трейлер</a>
</div>
```

Обратите внимание, что мы добавили дополнительный тег ..., чтобы привязать атрибут itemprop к соответствующему тексту на странице. Тег <span> не влияет на отображение страницы в браузере, поэтому его удобно использовать вместе с itemprop.

Теперь поисковые системы смогут понять не только то, что http://www.avatarmovie.com — это ссылка, но и то, что это ссылка на трейлер фантастического фильма «Аватар» режиссера Джеймса Кэмерона.

Иногда значение свойства может само являться сущностью, с собственным набором свойств. Например, режиссер фильма может быть описан как сущность с типом Person, у которой есть свойства name (имя) и birthDate (дата рождения). Чтобы указать, что значение свойства представляет собой сущность, необходимо добавить атрибут itemscope сразу после соответствующего itemprop.

Вложенные сущности. Листинг 15.5

```
<div itemscope itemtype="http://schema.org/Movie">
  <h1 itemprop="name">Аватар</h1>
  <div itemprop="director" itemscope
itemtype="http://schema.org/Person">
  Режиссер: <span itemprop="name">Джеймс Кэмерон</span> (род.
<span itemprop="birthDate">16 августа 1954 г.</span>)
  </div>
  <span itemprop="genre">Фантастика</span>
  <a href="/../movies/avatar-theatrical-trailer.html" item-
prop="trailer">Трейлер</a>
</div
```

Типы и свойства schema.org

Кроме типов Movie и Person, ранее упомянуты, schema.org описывает множество разнообразных типов сущностей, для каждого из которых определен набор свойств.

Наиболее обобщенный тип сущности — это Thing (нечто), у которого есть четыре свойства: name (название), description (описание), url (ссылка) и image (картинка). Более специализированные, частные типы имеют общие свойства с более универсальными. Например, Place (место) — частный случай Thing, а LocalBusiness (местная фирма) — частный случай Place. Частные типы наследуют свойства родительского типа. (Более того, тип LocalBusiness является и частным случаем Place, и частным случаем Organization, поэтому наследует свойства обоих родительских типов.)

Вот список некоторых популярных типов сущностей:

Творческие произведения: CreativeWork (творческое произведение), Book (книга), Movie (фильм), MusicRecording (музыкальная запись), Recipe (рецепт), TVSeries (телесериал)...

Встроенные нетекстовые объекты: AudioObject (аудио), ImageObject (изображение), VideoObject (видео)

Event (событие)

Organization (организация)

Person (человек)

Place (место), LocalBusiness (местная фирма), Restaurant (ресторан)...

Product (продукт), Offer (предложение), AggregateOffer (сводное предложение)

Review (отзыв), AggregateRating (сводный рейтинг).

16. Angular.JS

AngularJS является структурной основой для динамических веб-приложений. Эта технология позволяет использовать HTML в качестве языка шаблона.

Вывод “hello World”:

Hello World на Angular.js. Листинг 16.1

```
<html ng-app>
<head>
  <script src="angular.js"></script>
  <script src="controllers.js"></script>
</head>
<body>
  <div ng-controller='HelloController'>
    <p>{{greeting.text}}, World</p>
  </div>
</body>
</html>
```

В теге <html>, мы указываем, что это Angular приложение с помощью директивы ng-app. Ng-app будет вызывать и инициализировать приложение Angular.

Нотация {{_expression_}} в Angular указывает на привязку данных. Содержащееся внутри выражение может быть комбинацией выражения и фильтра: {{ expression | filter }}.

1. Контроллер

Параметр **ng-controller** со значением HelloController вызывает функцию HelloController, который находится в js-файле controller.js. Функция HelloController содержит данные в модели.

Файл controller.js. Листинг 16.2

```
function HelloController($scope) {
  $scope.greeting = { text: 'Hello' };
}
```

Задача контроллера в app-приложении:

1. Настройка начального состояния модели.
2. Экспорт данных модели и ответов функций в шаблон приложения.
3. Слежка за изменениями в моделях для мгновенного изменения значений в шаблоне.

Рассмотрим пример взаимодействия контроллера с моделью.

Двунаправленная привязка данных в модели. Листинг 16.3

```
<html ng-app>
<head>
  <script src="angular.js"></script>
  <script src="controllers.js"></script>
</head>
<body>
  <div ng-controller='HelloController'>
    <input ng-model='greeting.text'>
    <p>{{greeting.text}}, World</p>
  </div>
</body>
</html>
```

2. Модель

Атрибут **ng-model** в элементе <input> автоматически устанавливает двунаправленную привязку данных.

Это значит, любые изменения в форме обновят данные модели, а при изменении модели, обновится форма.

Рассмотрим Angular-приложение, реализующее форму заказов товаров:

Форма заказа товаров. Листинг 16.4

```
<!doctype html>
 <html ng-app>
   <head>
     <script
src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
     <script src="script.js"></script>
   </head>
   <body>
     <div ng-controller="InvoiceCntl">
       <b>Invoice:</b>
       <table>
        <tr><td>Количество</td><td>Стоимость</td></tr>
        <tr>
          <td><input type="number" ng-pattern="/\d+/" step="1"
min="0" ng-model="qty" required ></td>
         <td><input type="number" ng-model="cost" required ></td>
       </tr>
      </table>
      <hr>
      <b>Всего:</b> {{qty * cost | currency}}
    </div>
  </body>
</html>
```

HTML-шаблон вызывает контроллер script.js, в котором можем прописать значения по умолчанию для моделей qty и cost:

Script.js. Листинг 16.5

```
function InvoiceCntl($scope) {
  $scope.qty = 1;
```

```
  $scope.cost = 19.95;
}
```

3. Другие диррективы

Атрибут ng-repeat

ng-repeat='item in items' – аналог инструкции foreach().

Атрибут ng-click

ng-click – вызов обычного JavaScript-события click.

Атрибут ng-click. Листинг 16.6

```
  <script>
function StartUpController($scope) {
  $scope.computeNeeded = function() {
    $scope.needed = $scope.startingEstimate * 10;
  };
  $scope.requestFunding = function() {
    window.alert("Sorry, please get more customers first.");
  };
  $scope.reset = function() {
    $scope.startingEstimate = 0;
  };
}
  </script>
<form ng-submit="requestFunding()" ng-controller = "StartUpControl-
ler">
  Starting: <input ng-change="computeNeeded()" ng-model = "start-
ingEstimate">
  Recommendation: {{needed}}
  <button>Fund my startup!</button>
  <button ng-click="reset()">Reset</button>
</form>
```

Атрибут ng-change

Для input-элементов, совместно с атрибутом ng-model мы можем использовать атрибут ng-change для уточнения метода, который должен быть вызван в контроллере.

Атрибут ng-change. Листинг 16.7

```
<input ng-change="computeNeeded()"
                   ng-model="funding.startingEstimate">
```

Атрибут ng-submit

Если у нас имеется форма, содержащая несколько input-элементов, мы можем использовать атрибут ng-submit.

Дирректива ng-submit срабатывает, когда форма отправляет данные.

Атрибут ng-submit. Листинг 16.8

```
<script>
function StartUpController($scope) {
  $scope.computeNeeded = function() {
    $scope.needed = $scope.startingEstimate * 10;
  };
  $scope.requestFunding = function() {
    window.alert("Sorry, please get more customers first.");
  };
}
  </script>
<form ng-submit="requestFunding()" ng-
controller="StartUpController">
  Starting: <input ng-change="computeNeeded()" ng-
model="startingEstimate">
  Recommendation: {{needed}}
  <input type="submit" value="send">
</form>
```

Атрибут ng-class

Angular позволяет динамически менять стили. Рассмотрим пример, в котором по клику на кнопку меняется значение атрибута class.

Атрибут ng-class. Листинг 16.9

```
<style>
 .error {
   background-color: red;
 }
 .warning {
   background-color: yellow;
 }
</style>
<script>
function HeaderController($scope) {
  $scope.isError = false;
  $scope.isWarning = false;
  $scope.showError = function() {
    $scope.messageText = 'This is an error!';
    $scope.isError = true;
    $scope.isWarning = false;
  };
  $scope.showWarning = function() {
    $scope.messageText = 'Just a warning.  Please carry on.';
    $scope.isWarning = true;
    $scope.isError = false;
  };
}
</script>
<div ng-controller='HeaderController'>

  <div ng-class='{error: isError, warning: isWarn-
ing}'>{{messageText}}</div>

  <button ng-click='showError()'>Simulate Error</button>
  <button ng-click='showWarning()'>Simulate Warning</button>
```

```
</div>
```

4. Покупательская корзина с удалением товаров

Покупательская корзина. Листинг 16.10

```
<html ng-app>
<head>
  <title>Покупательская корзина</title>
</head>
<body ng-controller='CartController'>
<h1>Your Shopping Cart</h1>
<div ng-repeat='item in items'>
  <span>{{item.title}}</span>
  <input ng-model='item.quantity'>
  <span>{{item.price | currency}}</span>
  <span>{{item.price * item.quantity | currency}}</span>
  <button ng-click="remove($index)">Remove</button>
</div>
<script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.m
in.js"></script>
<script src="script.js"> </script>
</body>
</html>
```

Покупательская корзина использует следующий контроллер.

Контроллер CartController. Листинг 16.11

```
  function CartController($scope) {
    $scope.items = [
      {title: 'Paint pots',
       quantity: 8,
       price: 3.95
      },
      {title: 'Polka dots',
       quantity: 17,
       price: 12.95
      },
      {title: 'Pebbles',
       quantity: 5,
       price: 6.95
      }
    ];

    $scope.remove = function(index) {
      $scope.items.splice(index, 1);
    };
  }
```

5. Данные

Данные могут хранится либо в объектных литералах, либо в массивах за пределами котроллера.

В следующем листинге приведен пример извлечения данных из объектного литерала:

Данные в объектном литерале. Листинг 16.12

```
var messages = {};
messages.someText = 'You have started your journey.';
function TextController($scope) {
  $scope.messages = messages;
}
```

В шаблоне данные из модели будут вызываться так:

Вызов данных из модели. Листинг 5.5.2

```
{{messages.someText}}
```

6. Прослушиватель изменений в функциях, $scope-функция

Связывать метод с данными можно через атрибут ng-change, а можно и в самом контроллере (предпочтительнее). Для этого существует специальная $scope-функция – $watch().

Использование $scope-функции $watch(). Листинг 16.13

```
<html ng-app>
<body>
  <script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.6/angular.m
in.js">
  </script>
  <script>
function StartUpController($scope) {
  $scope.funding = { startingEstimate: 0 };
  computeNeeded = function() {
    $scope.funding.needed = $scope.funding.startingEstimate * 10;
  };
  $scope.$watch('funding.startingEstimate', computeNeeded);
}

  </script>
<form ng-controller="StartUpController">
  Starting: <input  ng-model="funding.startingEstimate">
  Recommendation: {{funding.needed}}
</form>
</body>
</html>
```

Для обновления модели из листинга нам пришлось использовать $scope-метод $watch(). Метод $watch() используется для связки наблюдаемой модели с функцией обратного вызова.

$scope.$watch – это, своего рода, наблюдатель за поведением функции. Как только в функции (первый параметр) происходят изменения, вызывается функция второго параметра.

7. Калькулятор корзины со скидкой

Создадим калькулятор корзины, который будет предлагать скидку в размере 10$, если покупка превышает 100$.

Колькулятор со скидкой. Листинг 16.14

```
<script>
function CartController($scope) {
  $scope.bill = {};
  $scope.items = [
    {title: 'Paint pots', quantity: 8, price: 3.95},
    {title: 'Polka dots', quantity: 17, price: 12.95},
    {title: 'Pebbles', quantity: 5, price: 6.95}
  ];
  $scope.totalCart = function() {
    var total = 0;
    for (var i = 0, len = $scope.items.length; i < len; i++) {
      total = total + $scope.items[i].price *
                      $scope.items[i].quantity;
    }
    return total;
  }
  $scope.subtotal = function() {
    return $scope.totalCart() - $scope.bill.discount;
  };
  function calculateDiscount(newValue) {
    $scope.bill.discount = newValue > 100 ? 10 : 0;
  }
  $scope.$watch($scope.totalCart, calculateDiscount);
}
  </script>
<div ng-controller="CartController">
  <div ng-repeat="item in items">
    <span>{{item.title}}</span>
    <input ng-model="item.quantity">
    <span>{{item.price | currency}}</span>
    <span>{{item.price * item.quantity | currency}}</span>
  </div>
  <div>Total: {{totalCart() | currency}}</div>
  <div>Discount: {{bill.discount | currency}}</div>
  <div>Subtotal: {{subtotal() | currency}}</div>
</div>
```

$scope-функция $watch следит за изменениями в $scope.totalCart и, как только они наступают, вызвает функцию обратного вызова calculateDiscount и передает в нее всё, что возвращается return-ом в первой функции.

8. Продвинутые формы

Основная цель веб-приложения — отображение и сбор данных. Angular стремится сделать обе эти операции тривиальными. Пример демонстрирует, как можно построить простую форму, позволяющую пользователю вводить данные.

Html-код формы. Листинг 16.15

```
<!doctype html>
 <html ng-app>
 <head>
    <script src="http://code.angularjs.org/1.1.5-
angular_ru/angular.min.js"></script>
    <script src="script.js"></script>
    </head>
 <body>
    <div ng-controller="FormController" class="example">
    <label>Имя:</label><br>
    <input type="text" ng-model="user.name" required/> <br><br>
    <label>Адрес:</label><br>
    <input type="text" ng-model="user.address.line1" size="33" re-
quired> <br>
    <input type="text" ng-model="user.address.city" size="12" re-
quired>,
    <input type="text" ng-model="user.address.state"
    ng-pattern="state" size="2" required>
    <input type="text" ng-model="user.address.zip" size="5"
    ng-pattern="zip" required><br><br>
    <label>Телефон:</label>
    [ <a href="#" ng-click="addContact()">добавить</a> ]
    <div ng-repeat="contact in user.contacts">
    <select ng-model="contact.type">
    <option>эл. почта</option>
    <option>телефон</option>
    <option>пейджер</option>
    <option>IM</option>
    </select>
    <input type="text" ng-model="contact.value" required>
    [ <a href="" ng-click="removeContact(contact)">?</a> ]
    </div>
    <hr/>
    Отладчик:
    <pre>user={{user | json}}</pre>
    </div>
 </body>
</html>
```

Рассмотрим обработчик формы, скрипт script.js

Script.js. Листинг 16.16

```
function FormController($scope) {
 var user = $scope.user = {
    name: 'John Smith',
    address:{line1: '123 Main St.', city:'Anytown', state:'AA',
zip:'12345'},
    contacts:[{type:'phone', value:'1(234) 555-1212'}]
 };
 $scope.state = /^\w\w$/;
 $scope.zip = /^\d\d\d\d\d$/;

 $scope.addContact = function() {
    user.contacts.push({type:'email', value:''});
 };
```

```
  $scope.removeContact = function(contact) {
   for (var i = 0, ii = user.contacts.length; i < ii; i++) {
     if (contact === user.contacts[i]) {
     $scope.user.contacts.splice(i, 1);
   }
  }
  };
}
```

9. Директивы

Директивы — это ключевая особенность AngularJS. С помощью директив можно добавлять новое поведение существующим HTML элементам, можно создавать новые компоненты.

Множество директив можно найти по адресу: https://github.com/angular-ui

Рассмотрим пример простой директивы angular

Директива angular. Листинг 16.17

```
  myModule.directive("greet", function () {
    return {
        template: "<p>Привет, </p>",
        replace: true,
        scope: {},

        link: function(scope, element, attributes) {
            scope.name = "Иван"
        }
    }
})
```

В результате работы этой директивы вместо <div> будет выведено <p>Привет, Иван</p>. Параметр **replace** позволяет определить, будет ли директива целиком замещать DOM, которому применена, или же встраиваться внутрь него. Параметр **template** можно заменить на **templateUrl** и подключать шаблон из файла.

Наиболее важными параметрами здесь являются **scope** и **link**. Последний — это функция, осуществляющая привязку scope к шаблону. Ну а scope позволяет изолировать область видимости внутри директивы. Эти два параметра следует указывать практически всегда.

Есть также параметр **compile**, который позволяет задать обработчик шаблона перед связыванием его с link, но он используется довольно редко.

10. Подключение плагинов jQuery

В Angular все действия с DOM необходимо проводить в директивах, поэтому любые jQuery-плагины, работающие с DOM (а это почти все существующие

плагины), интегрируются в первую очередь в них. При этом работать с таким плагином становится ничуть не сложнее.

Так, вместо:

Стандартный вызов плагинов jQuery. Листинг 16.18

```
$('#elem').plugName({_options_})
```

Мы пишем:

Вызов плагина jQuery через диррективу Angular. Листинг 16.19

```
<div plug-name="{_options_}"></div>
```

Подключим jquery.toolbar.js (http://paulkinzett.github.io/toolbar/), добавляющий всплывающую панель инструментов к элементу.

Стандартный способ подключения:

Стандартный способ подключения jquery.toolbar.js. Листинг 16.20

```
$('#format-toolbar').toolbar({
    content: '#format-toolbar-options',
    position: 'left'
});
```

Вызов плагина через директиву angular:

Вызов плагина jQuery через директиву angular. Листинг 16.22

```
<!doctype html>
 <html ng-app="Toolbar">
 <head>
    <script src="http://code.angularjs.org/1.1.5-
angular_ru/angular.min.js"></script>
    <script src="jquery.js"></script>
    <script src="toolbar.js"></script>
    <script src="script.js"></script>
 </head>
 <body>
    <link rel="stylesheet" type="text/css"
href="http://paulkinzett.github.com/toolbar/css/toolbars.css">
    <!-- Место, куда нужно щелкать, чтобы показалась панель инстру-
ментов -->
    <div id="format-toolbar" class="settings-button" toolbar-
tip="{content: '#format-toolbar-options', position: 'top'}">
    <img src="http://paulkinzett.github.com/toolbar/img/icon-cog-
small.png">
    </div>
    <!-- Код панели инструментов -->
    <div id="format-toolbar-options">
    <a href="#"><i class="icon-align-left"></i></a>
    <a href="#"><i class="icon-align-center"></i></a>
    <a href="#"><i class="icon-align-right"></i></a>
    </div>
 </body>
```

```
</html>
```

А вот и скрипт директивы.

Директива toolbarTip. Листинг 16.23

```
angular.module('Toolbar', [])
 .directive('toolbarTip', function() {
  return {
    link: function(scope, element, attrs) {
     // Функция link отдает в качестве параметра элемент, помечен-
ный нашим атрибутом. Оборачиваем этот элемент jQuery, и получаем из
attrs значение атрибута. Функция scope.$eval() вычисляет переданное
ей выражение. В нашем случае просто превращает строку в массив с
парметрами плагина
     $(element).toolbar(scope.$eval(attrs.toolbarTip));
    }
  };
});
```

11. Контроллер в качестве модуля

Файл шаблона. Листинг 16.24

```
<html ng-app="phonecatApp">
 <head>
    ...
    <script src="lib/angular/angular.js"></script>
    <script src="js/controllers.js"></script>
 </head>
 <body ng-controller="PhoneListCtrl">
   <ul>
    <li ng-repeat="phone in phones">
     {{phone.name}}
     <p>{{phone.snippet}}</p>
    </li>
   </ul>
 </body>
</html>
```

Обратите внимание, что приложение в качестве директивы ng-app указывает контроллер phonecatApp, который загружается как модуль в html-приложение при загрузке.

Контроллер возвращает модель phones с массивом данных.

Регистрация контроллера в качестве модуля. Листинг 16.25

```
var phonecatApp = angular.module('phonecatApp', []);

phonecatApp.controller('PhoneListCtrl', function ($scope) {
 $scope.phones = [
  {'name': 'Nexus S',
   'snippet': 'Fast just got faster with Nexus S.'},
  {'name': 'Motorola XOOM™ with Wi-Fi',
   'snippet': 'The Next, Next Generation tablet.'},
  {'name': 'MOTOROLA XOOM™',
```

```
   'snippet': 'The Next, Next Generation tablet.'}
 ];
});
```

Здесь мы объявили контроллер PhoneListCtr и зарегистрировали его в качестве модуля phonecatApp.

12. Маршрутизация запросов

Одно из основных преимуществ angular в сравнении с другими js-фрэймворками, заключается в том, что angular умеет прослушивать адресную строку. Соответственно, можно настроить маршрутизацию (роутинг) запросов. Для этого имеется удобный модуль angular-route.js.

Рассмотрим пример.

Файл шаблона. Листинг 16.26

```
<!doctype html>
<html lang="en" ng-app="phonecatApp">
<head>
  <meta charset="utf-8">
  <title>Google Phone Gallery</title>
  <link rel="stylesheet"
href="bower_components/bootstrap/dist/css/bootstrap.css">
  <link rel="stylesheet" href="css/app.css">
  <script src="bower_components/angular/angular.js"></script>
  <script src="bower_components/angular-route/angular-
route.js"></script>
  <script src="js/app.js"></script>
  <script src="js/controllers.js"></script>
</head>
<body>

  <div ng-view></div>

</body>
</html>
```

К шаблону подключается сама библиотека angular, потом модуль angular-rotue.js, и файлы app.js, и controllers.js.

App.js. Листинг 16.27

```
'use strict';

/* App Module */

var phonecatApp = angular.module('phonecatApp', [
  'ngRoute',
  'phonecatControllers'
]);

phonecatApp.config(['$routeProvider',
  function($routeProvider) {
```

```
    $routeProvider.
      when('/phones', {
        templateUrl: 'partials/phone-list.html',
        controller: 'PhoneListCtrl'
      }).
      when('/phones/:phoneId', {
        templateUrl: 'partials/phone-detail.html',
        controller: 'PhoneDetailCtrl'
      }).
      otherwise({
        redirectTo: '/phones'
      });
  }]);
```

Если в браузер передается запрос /phones, подгружаем шаблон phone-list.html из папки partials. Вот его листинг:

Листинг phone-list.html. Листинг 16.28

```
<div class="container-fluid">
  <div class="row">
    <div class="col-md-2">
      <!--Sidebar content-->

      Search: <input ng-model="query">
      Sort by:
      <select ng-model="orderProp">
        <option value="name">Alphabetical</option>
        <option value="age">Newest</option>
      </select>

    </div>
    <div class="col-md-10">
      <!--Body content-->

      <ul class="phones">
        <li ng-repeat="phone in phones | filter:query | order-
By:orderProp" class="thumbnail">
          <a href="#/phones/{{phone.id}}" class="thumb"><img ng-
src="{{phone.imageUrl}}"></a>
          <a href="#/phones/{{phone.id}}">{{phone.name}}</a>
          <p>{{phone.snippet}}</p>
        </li>
      </ul>

    </div>
  </div>
</div>
```

Всё, что будет идти после phones/, воспринимается как параметр адресной строки с именем phoneId.

Листинг phone-detail.html. Листинг 16.29

```
TBD: detail view for <span>{{phoneId}}</span>
```

А вот и сам контроллер, файл controller.js

Листинг controller.js. Листинг 16.30

```
'use strict';

/* Controllers */

var phonecatControllers = angular.module('phonecatControllers',
[]);

phonecatControllers.controller('PhoneListCtrl', ['$scope', '$http',
  function($scope, $http) {
    $http.get('phones/phones.json').success(function(data) {
      $scope.phones = data;
    });

    $scope.orderProp = 'age';
  }]);

phonecatControllers.controller('PhoneDetailCtrl', ['$scope',
'$routeParams',
  function($scope, $routeParams) {
    $scope.phoneId = $routeParams.phoneId;
  }]);
```

17. Система контроля версий, GIT

Система управления версиями (СУВ) — это система, сохраняющая изменения в одном или нескольких файлах так, чтобы потом можно было восстановить определённые старые версии.

Если вы хотите, чтобы изучение Git шло гладко, то самое важное, что нужно помнить про Git – это **три состояния файла**. В Git файлы могут находиться в одном из трёх состояний:

- зафиксированном,
- изменённом
- подготовленном.

«Зафиксированный» значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проекте с использованием Git есть три части: каталог Git (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).

Каталог Git — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git и помещаются на диск для того, чтобы вы их просматривали и редактировали.

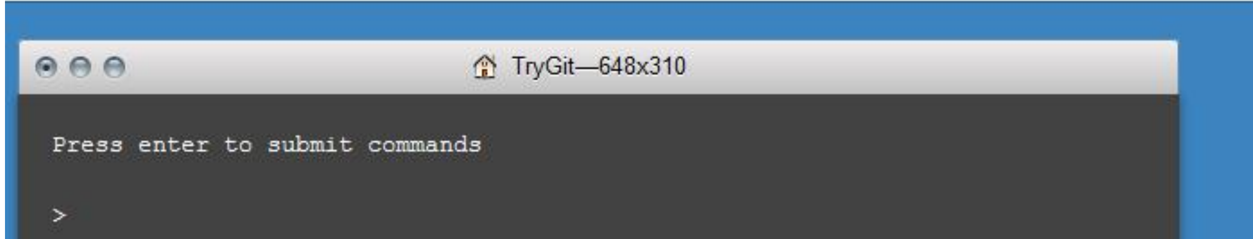
Область подготовленных файлов — это обычный файл, обычно хранящийсяв каталоге Git, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть егообластью подготовленных файлов (stagingarea).

Стандартный рабочий процесс с использованием Git выглядит примерно так:

1. Вы изменяете файлы в вашем рабочем каталоге.
2. Вы подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Вы делаете коммит. При этом слепки из области подготовленных файлов сохраняются в каталог Git.

Если рабочая версия файла совпадает с версией в каталоге Git, файл считается зафиксированным.

Хорошая пошаговая обучалка по GIT находится по адресу http://try.github.io, которую настоятельно рекомендуется пройти перед использованием GIT.



В обучалке используются консольные команды GIT. Однако если вы предпочитаете графические оболчки, то для Windows существует множество таких. Два основных — это SmartGit (кроссплатформенный) и TortoiseGit.

Удаленный репозиторий

Для создания удаленного репозитория, нужна регистрация на bitbacket.ogr (для private проектов), или на github.com (для openSource проектов).

После регистрации на одном из сайтов, заходим в свой кабинет пользователя и создаем пустой репозиторий.

Основные команды GIT

git init – инициализация пустого репозитория

git add * - добавление в новых файлов в репозиторий

git commit –m “text commit” – фиксация изменений

git status – проверка статуса

git remote add site http://github.com/user/project.git – создание переменной site хранящей путь к удаленному репозиторию.

git push site master – заливка файлов текущего репозитория на удаленный.

git pull site master – загрузка файлов удаленного репозитория на текущий.

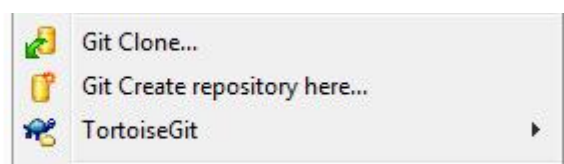
git clone http://github.com/user/project.git – клонирование удаленного репозитория.

TortoiseGit

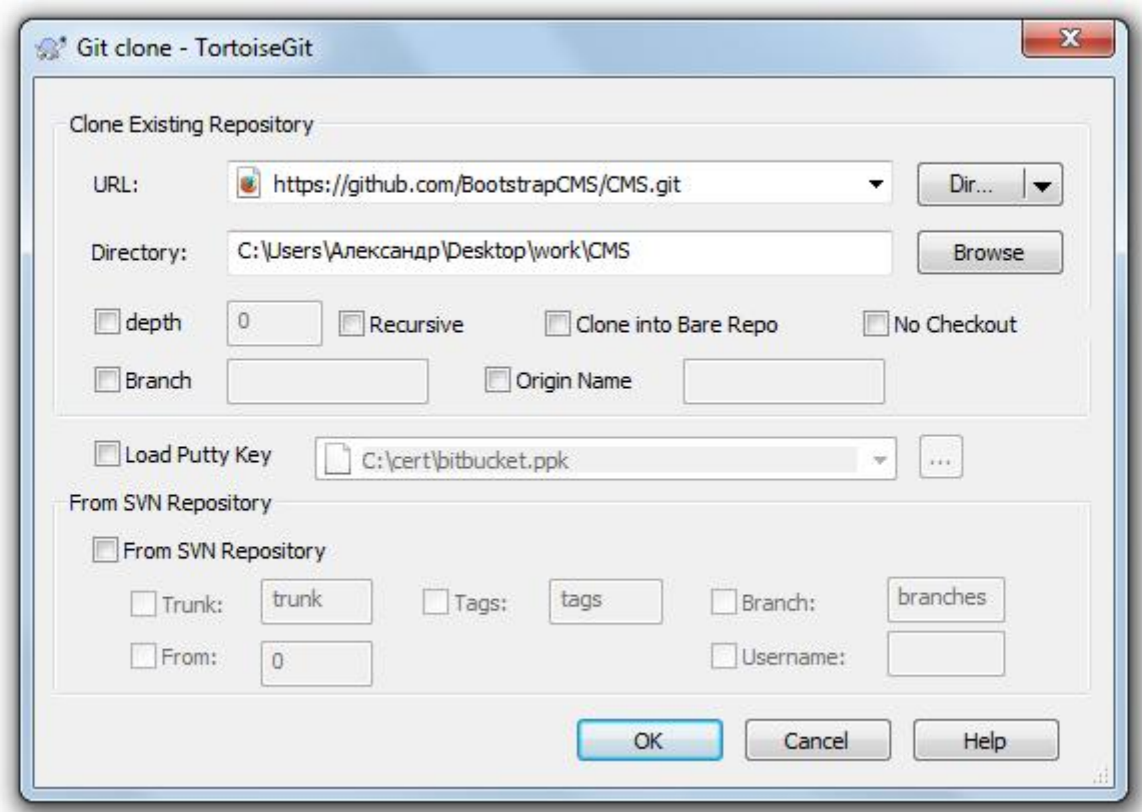
Качаем по ссылке code.google.com/p/tortoisegit/downloads/list. При установке везде жмем Next.

Послу установки TortoiseGit нам нужно либо создать пустой репозиторий, либо клонировать прект с удаленного репозитория.

Для клонирования проекта, нажимаем правой кнопкой по пустой папке с именем проекта.



И выбраем команду Git Clone. В поле URL, в появившемся окне, вставляем url проекта.

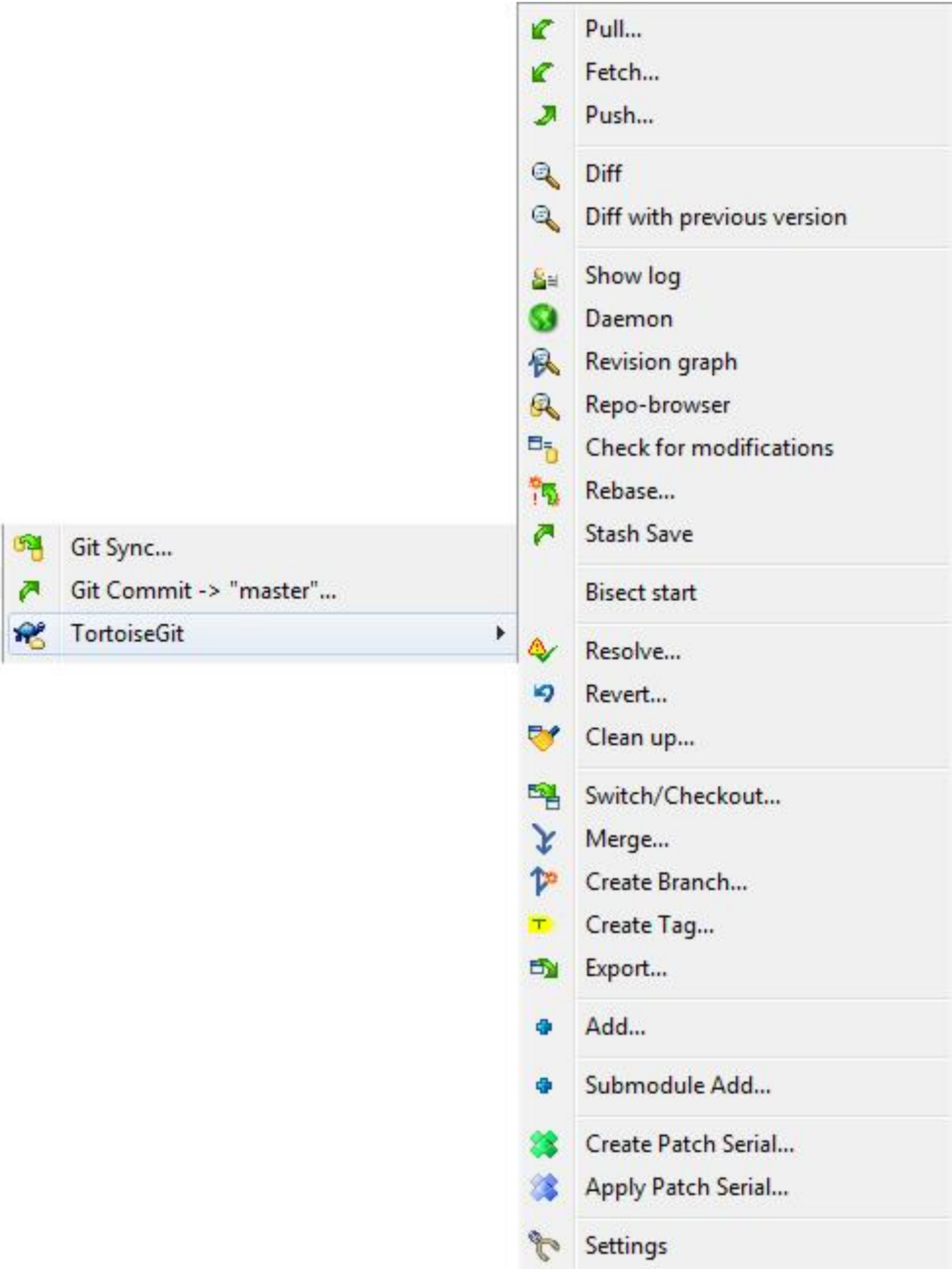


Команды Push и Pull

Pull – скачиваем с удаленного сервера. (первая ссылка в выпадающем списке)

Push – заливаем на удаленный сервер. (вторая ссылка в выпадающем списке)

Но перед использованием этих команд необходимо помнить, что файлы должны быть зафиксированы.



Работа по http никаких трудностей не вызывает, в нужный момент просто используется пароль учетной записи на github.

Приложение

Инструментарий

IDE:NetBeans, PHPShtorm, WebShtorm, – или любая другая интегрированная среда разработки.

Nodepad++- Блокнот.

OpenServer – Локальный сервер (Содержит встроенные технологии:PHP, Apache, MySQL и системы управления MySQL).

Git – система контроля версий

Composer – локальный менеджер зависимостей.

Firefox – Браузер.

Firebug – Отладчик кода на стороне клиента

FireFTP – или любой другой FTP-клиент.

Node.js – программная платформа

Удаленный репозиторий, аккаунт на bitbacket.org или github.com

Программа курса JavaScript для продвинутых

1. Инструментарий. Верстка сайта. Макет главной страницы сайта

Настройка рабочей среды.

Основные принципы HTML5. Различные способы верстки: табличная, блочная, гибкая блочная. Использование Bootstrap для создания мультимедийных (одинаково отображающихся во всех браузерах) страниц. Адаптивная верстка.

3 составляющих любой html-страницы: блок предметика (название сайта, логотип, основное меню сайта), блок конкретика (вспомогательное меню, меняющаяся часть сайта), блок для пользователя (реклама, виджеты, банеры). Schema.org.

2. jQuery. Теория и практика

Рассматриваются основные методы jQuery. Группы методов: селекторные, стилистические, атрибутные, текстовые, DOM-методы, вспомогательные, событийные, анимационные, ajax и ajax-подобные.

Создание динамической страницы. Использование прослушивателей событий click, mouseover, mouseout и других.

3. Синтаксис JavaScript

Типы данных. Массивы. Инструкции. Операторы. Циклы. Функции. Объекты и прототипы. Наследование. Конструкторы. Методики программирования на javaScript. Шаблоны проектирования.

4.Node.js. Серверный JavaScript.

Основы node.js. Консольные команды node.js. Режим repl. Ядро Node. Создание node-сервера. Менеджер зависимости npm. Внедрение npm-зависимостей. Методика асинхронного программирования. Разработка приложения на простом паттерне.

5. MongoDB

Хранилище данных MongoDB. Установка. CRUD-операции. Подключение и использование модуля Mongoose. JSON

6. Express-приложение

Фрэймворк Express. Шаблонизатор jade. MVC в действии. Маршрутизация сайта. Конфигурирование. Особенности подключения скриптов и стилей. Подключение базы данных Mongo. Модули для Express-приложения.

7. Система контроля версий.

Командная разработка проекта. Система контроля версий GIT. Ветвление проекта. Удаленные git-репозитории github.com и bitbacket.org. Сравнение. Выбор. Косольные команды git. Графические оболочки для git – tortoiseGit.

8. API форм

Набор свойств, методов и событий обслуживающих формы. Валидация данных на стороне клиента.

9. API видео и аудио

Набор свойств, методов и событий, обслуживающих видео- и аудио-файлы. Реализация медиа-плеера.

10. API холст

Рисование на холсте. Линии. Прямоугольник. Круг. Кривые. Кривые Безье. Закрашивание. Трансформация холста. Работа с текстом. Вставка изображений. Узоры.

11. API перетаскивания

Прослушивание событий целевого элемента (куда перетащить) и элемента источника (что перетаскиваем). Реализация перетаскивания товаров в покупательскую карзину.

12. API геолокации

Статические и динамические карты. Методы определения текущего местоположения пользователя и слежение за передвижением пользователя. Интеграция карт Google и Yandex.

13. API web-хранилища

Сохраниение данных на стороне клиента. Cookie, localStorage и session-Storage.

14. Модули авторизации

Внедрение зависимости пользовательской авторизации. Разработка кабинета пользователя.

15. Системы сборки frontend

Сборка css- и js-файлов. Grunt. Gulp.

16. Взаимодействие с web-сервером.

Основы серверного программирование. Обслуживание серверов. Размещение работ.

17. Современный фронтенд

Обзор рынка. Вспомогательные технологии. Обзор используемых программ и платформ.

Список используемой литературы

«JavaScript. Подробное руководство» Дэвид Флэнаган.

«HTML5 для профессионалов», Хуан Диего Гоше, издательство Питер, 2013 год.

«JavaScript» Михалькевич Алесандр Викторович. 2015

«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

«HTML5», Михалькевич Александр Викторович 2013

«Секреты JavaScript ниньзя» Джон Резинг

«Изучаем Node.js» Шелли Пауэрс 2014

Документация Angular

Интернет-материалы

http://mikhalkevich.colony.by

Методическое издание

Михалькевич Александр Викторович

JavaScript

Авторская редакция
Компьютерная верстка Михалькевич Александр Викторович

Подписано в печать 25.04.2015. Формат 60x84 1/16.
Бумага HPOffice, Печать лазерная.
Усл. печ. л. . Уч.-изд. л. .
Тираж 1000 экз. Заказ.