Продвинутый javascript

Лучшие практики и шаблоны проектирования

Чернобаев Николай
Год: 2015

LoftSchool
от мыслителя к создателю

Научись говорить на языке javascript

- Привыкни к четкому code style и внедри его
- Избегай глобальных переменных. Не засоряй и учитывай глобальную область видимости
- var всегда в самом верху функции
- Названия переменных верблюдиком (varName)
- Не забывай точки с запятыми (jslint/jshint)

- Логичные и правильные названия переменных
- используй === за место ==
- Используй литералы и краткие нотации
- Не миксуй технологии (js,css,html)
- Названия функций-конструкторов (классов) с большой буквы

Не лезь в чужой монастырь…прими его правила

- Понимай как работают js функции и this
- Пойми hoisting (поднятие переменных) и scopes (области видимости)
- Пойми замыкания
- Понимай как работает асинхронность (ajax - callbacks, listeners, promises, deferred)
- Понимай как работает прототипное наследование с помощью функций-конструкторов (классов) и Object.create
- Пойми iife (Immediately-Invoked Function Expression) – функция которая вызывается сразу после объявления и паттерн модуля (module pattern)

LoftSchool
от мыслителя к создателю

Придерживайся хорошего тона

- Пиши комментарии (jsDoc)
- Используй цепные вызовы функций
- Не создавай html на js. используй шаблонизаторы
- Организуй свой код (ооп)

- Не используй var в циклах
- Используй для функций-конструкторов (классов) options объекты
- Одна функция – одно действие. Упрощай и дроби до логичного максимума.
- DRY (Don’t repeat yourself)

Лучшие Javascript Style Guides

Google: http://google.github.io/styleguide/javascriptguide.xml
Airbnb: https://github.com/airbnb/javascript
Github: https://github.com/styleguide/javascript
Mozilla: https://goo.gl/HtFRDb

Организация кода и ООП

ООП в js - инкапсуляция, абстракция, насследование и полиморфизм

- Инкапсулируй код в модулях. Организуй однотипные данные в объекты.
- Старайся делать минимум зависимостей. Все модули независимы
- Выноси общий и дополнительный (расширяющий) функционал в абстракции – прототипы (классы)
- Наследуй нужные свойства и методы от прототипов
- Не забывай про полиморфизм и используй/заимствуй методы нужного тебе прототипа

Инкапсуляция

```
34 //инкапсулирование
35
36 var module = (function() {
37     function _privateMethod1 (argument) {
38         //...
39     },
40
41     function _privateMethod2 (argument) {
42         //...
43     },
44
45     return {
46         publicMethod: function(){
47             privateMethod1();
48         }
49     };
50 })();
51
52 module.publicMethod();
```

Наследование и абстракция

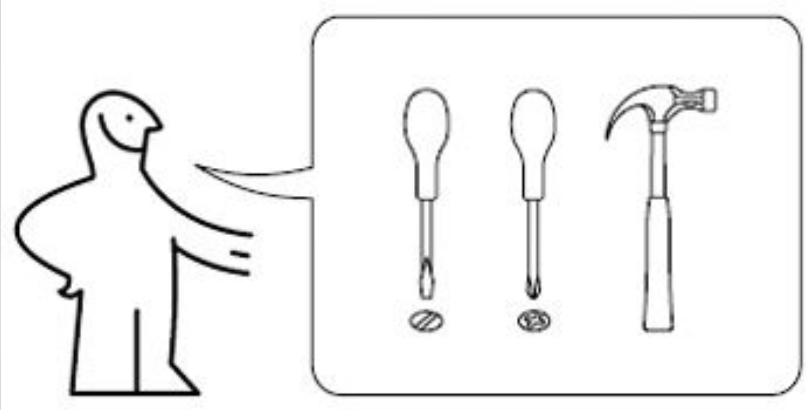
```
1  //наследование и абстракция
2  var animal = {
3      canRun: true
4  };
5
6  var Wolf = function () {
7      this.name = 'Волк'
8  };
9
10 var Grey = function () {
11     this.color = 'Серый'
12 };
13
14 var Black = function () {
15     this.color = 'Черный'
16 };
17
18 var Chicken = function () {
19     this.name = "Курица";
20     this.haveWings = true;
21 };
22
23
24 Wolf.prototype = animal;
25 Chicken.prototype = animal;
26
27
28 var wolfy = new Wolf();
29 var chicky = new Chicken();
30
```

LoftSchool
от мыслителя к создателю

Полиморфизм

```
55 //полиморфизм
56 //это когда можно вызвать чужую функцию
57 //в своем собственном контексте (позаимствовать)
58
59 //Конструктор родительского класса
60 function Animal(name) {
61   this.name = name;
62 }
63
64 Animal.prototype.speak = function() {
65   alert(this.name + " says:");
66 }
67
68 //Конструктор унаследованного класса "Dog"
69 function Dog(name) {
70   Animal.call(this, name);
71 }
72
73 Dog.prototype.speak = function() {
74   Animal.prototype.speak.call(this);
75
76   alert("woof");
77 }
78
79 var snoopDoggyDog = new Dog('Snoop');
80 var 2pac = new Dog('2pac');
81 var notoriousBIG = new Dog();
```

Пиши поддерживаемый код !



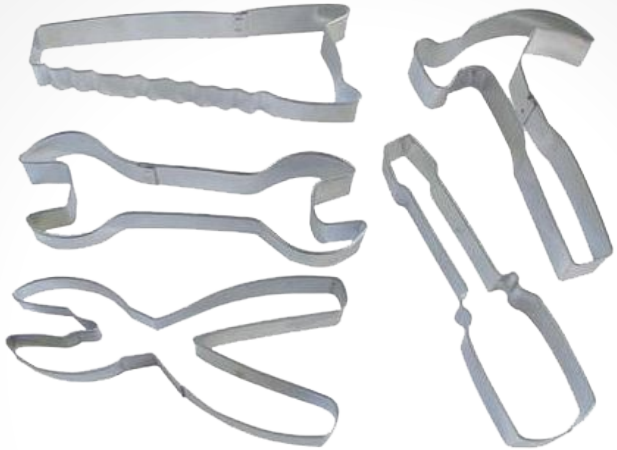
- Интуитивный
- Понятный
- Легко адаптируемый
- Расширяемый
- Отлаживаемый (debuggable)
- Тестируемый (Jasmine, Karma)

Почему запариваемся над поддерживаемым кодом?

Потому что 80% времени мы поддерживаем код, а не пишем его!

LoftSchool
от мыслителя к создателю

Design Patterns (Шаблоны проектирования)



- Убирают дублирование
- Общая терминология
- Быстрое взаимодействие
- Генерируемые и переиспользуемые компоненты

- Доказанные парадигмы. Они работают!
- Легкие в тестировании
- Поддерживают изменения системы

Виды шаблонов

- Пораждающие (Creational)

Constructor, Factory, Prototype, Singleton

- Структурные (Structural)

Module, Decorator, Façade

- Поведенческие (Behavioral)

Mediator, Observer

Addy Osmani JS Patterns

LoftSchool
от мыслителя к создателю

АРХИТЕКТУРА СОВРЕМЕННОГО JS ПРИЛОЖЕНИЯ

- 2 парадигмы: MVC и поддерживаемая и расширяемая javascript архитектура
- Ключ к поддерживаемости и расширяемости - независимые модули, не только js, но и css, html.
- Nicholas Zakas: nczonline.net
- Addy Osmani: addyosmani.com/blog

Паттерны MV*

- Model-View-Controller (MVC)

Модель-представление-контроллер. Контроллер и представление зависят от модели (подписаны на ее изменение), но модель никак не зависит от этих двух компонент

- Model-View-Presenter

Presenter посредник между моделью и представлением. Решает все вопросы. Представление не подписывается на изменения модели.

- Model-View-View-Model

Изменение состояния модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings/синхронизация)

Model

```
(function (window) {
    'use strict';

    function Model(storage) {
        this.storage = storage;
    }

    Model.prototype.create = function (title, callback) {
        title = title || '';
        callback = callback || function () {};

        var newItem = {
            title: title.trim(),
            completed: false
        };

        this.storage.save(newItem, callback);
    };

    Model.prototype.read = function (query, callback) {
        var queryType = typeof query;
        callback = callback || function () {};

        if (queryType === 'function') {
            callback = query;
            return this.storage.findAll(callback);
        } else if (queryType === 'string' || queryType === 'number') {
            query = parseInt(query, 10);
            this.storage.find({ id: query }, callback);
        } else {
            this.storage.find(query, callback);
        }
    };

    Model.prototype.update = function (id, data, callback) {
        this.storage.save(data, callback, id);
    };
```

View

```
(function (window) {
    'use strict';

    function View(template) {
        this.template = template;

        this.ENTER_KEY = 13;
        this.ESCAPE_KEY = 27;

        this.$todoList = qs('#todo-list');
        this.$todoItemCounter = qs('#todo-count');
        this.$clearCompleted = qs('#clear-completed');
        this.$main = qs('#main');
        this.$footer = qs('#footer');
        this.$toggleAll = qs('#toggle-all');
        this.$newTodo = qs('#new-todo');
    }

    View.prototype._removeItem = function (id) {
        var elem = qs('[data-id="' + id + '"]');

        if (elem) {
            this.$todoList.removeChild(elem);
        }
    };

    View.prototype._clearCompletedButton = function (completedCount, visible) {
        this.$clearCompleted.innerHTML = this.template.clearCompletedButton(completedCount);
        this.$clearCompleted.style.display = visible ? 'block' : 'none';
    };

    View.prototype._setFilter = function (currentPage) {
        qs('#filters .selected').className = '';
        qs('#filters [href="#/' + currentPage + '"]').className = 'selected';
    };
```

Controller

```
(function (window) {
    'use strict';

    function Controller(model, view) {
        var that = this;
        that.model = model;
        that.view = view;

        that.view.bind('newTodo', function (title) {
            that.addItem(title);
        });

        that.view.bind('itemEdit', function (item) {
            that.editItem(item.id);
        });

        that.view.bind('itemEditDone', function (item) {
            that.editItemSave(item.id, item.title);
        });

        that.view.bind('itemEditCancel', function (item) {
            that.editItemCancel(item.id);
        });

        that.view.bind('itemRemove', function (item) {
            that.removeItem(item.id);
        });

        that.view.bind('itemToggle', function (item) {
            that.toggleComplete(item.id, item.completed);
        });

        that.view.bind('removeCompleted', function () {
            that.removeCompletedItems();
        });

```

Поддерживаемая и расширяемая javascript архитектура

- Независимые модули (Module)
- Ядро (Mediator)
- Sandbox (Façade)
- библиотеки и наборы инструментов

В классическом случае по N.Zakas и Addy Osmani

Mediator (ядро) – Facade (sandbox controller) - Modules

Шаблон Module

- Инкапсулированная частичка приложения
- Взаимозаменяемая единичная часть большой системы, которая может быть легко переиспользована
- Модули хотят оповестить ядро - когда что-то интересное происходит
  - В веб-приложениях состоят не только из программного кода (js) но и из html, css
  - В очень крупных приложениях каждый модуль может содержать MVC

LoftSchool
от мыслителя к создателю

Шаблон Module

```
var testModule = (function () {

  var counter = 0;

  return {

    incrementCounter: function () {
      return counter++;
    },

    resetCounter: function () {
      console.log( "counter value prior to reset: " + counter );
      counter = 0;
    }
  };

})();

testModule.incrementCounter();
```

LoftSchool
от мыслителя к создателю

Шаблон Module

- Все что после return, это - public methods. до – private
- И подобные паттерны есть во всех знаменитых библиотеках, включая jquery

Современные модульные паттерны

- Формат для написания модулей javascript в браузере (AMD - require.js, browserify)
- Модульный формат оптимизированный для сервера (commonjs)
- ES harmony - модули будущего

Шаблон Facade

- позволяет скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.
- ДУМАЙТЕ ОБ ЭТОМ КАК ОБ API
- коммуникации с модулями
- ajax запросы
- dom-манипуляции
- установка/снятие обработчиков событий
- запрос на расширение возможностей через расширения

Шаблон Facade

```
var addMyEvent = function( el,ev,fn ){

    if( el.addEventListener ){
            el.addEventListener( ev,fn, false );
        }else if(el.attachEvent){
            el.attachEvent( "on" + ev, fn );
        } else{
            el["on" + ev] = fn;
     }

};
```

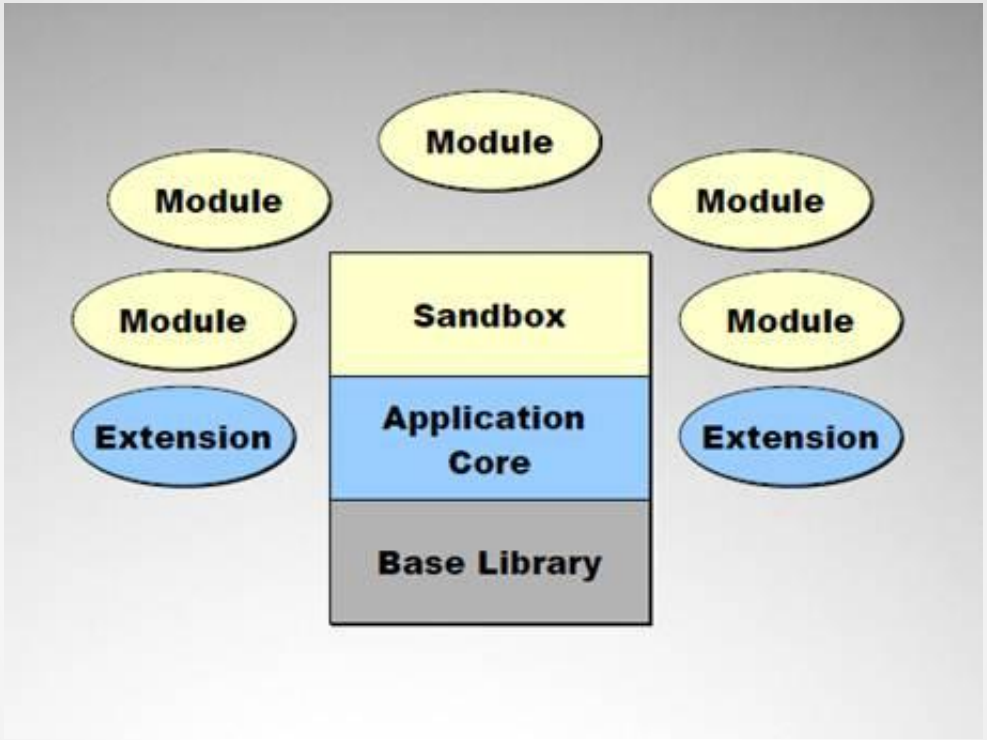
Теперь можно использовать addMyEvent - даже не зная что внутри он выполняет три сложных действия

```
var module = (function() {

    var _private = {
        i:5,
        get : function() {
            console.log( "current value:"_+ this.i);
        },
        set : function( val ) {
            this.i = val;
        },
        run : function() {
            console.log( "running"_);
        },
        jump: function(){
            console.log( "jumping"_);
        }
    };

    return {

        facade : function( args ) {
            _private.set(args.val);
            _private.get();
            if ( args.run ) {
                _private.run();
            }
        }
    };
}());


module.facade( {run: true, val:10} );
```

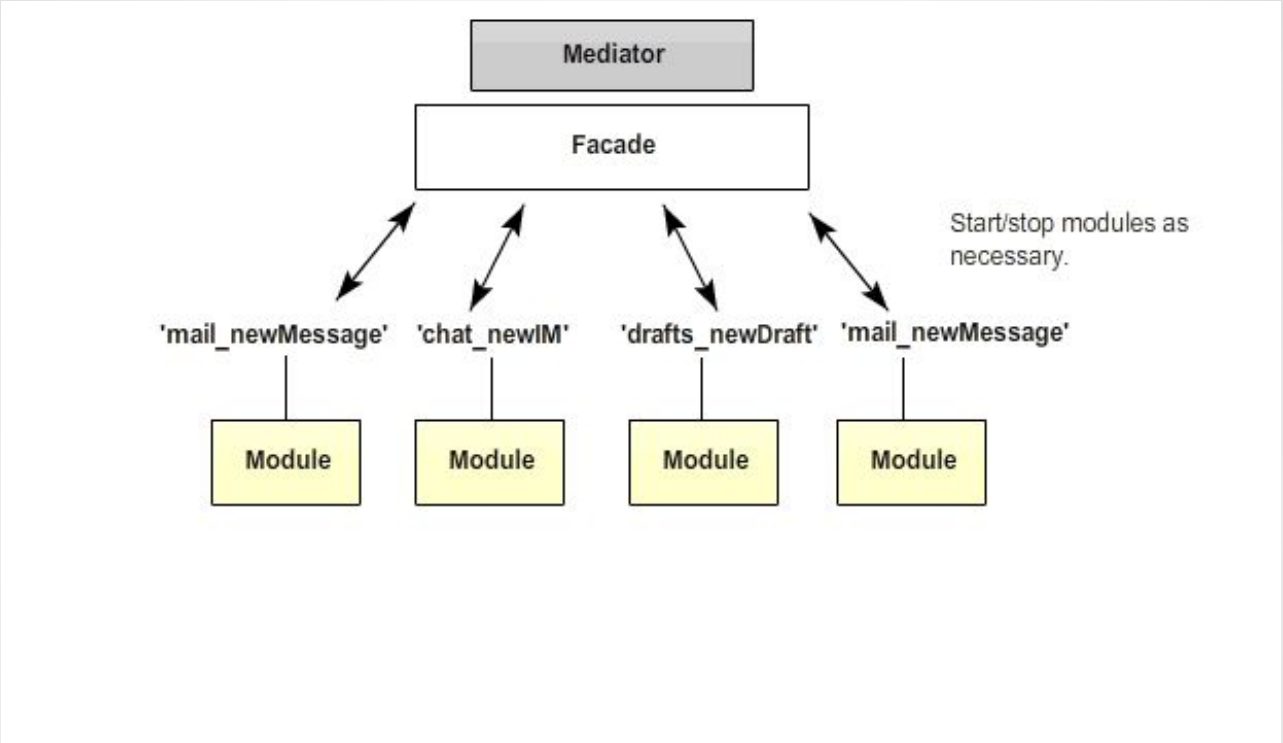
Шаблон Mediator

- Глобальный диспетчер событий
- Взаимодействие между модулями осуществляется путём посылки сообщений глобальному диспетчеру, а уже он принимает решение что с этим сообщением делать — создать/удалить модули, дёрнуть методы других модулей, выполнить какой то метод итд
- Инкапсулирует как разрозненные модули взаимодействуют друг с другом, выступая в роли посредника
- Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.
- Жизненный цикл модулей
- Взаимодействия между модулями
- Обработка ошибок
- Расширения

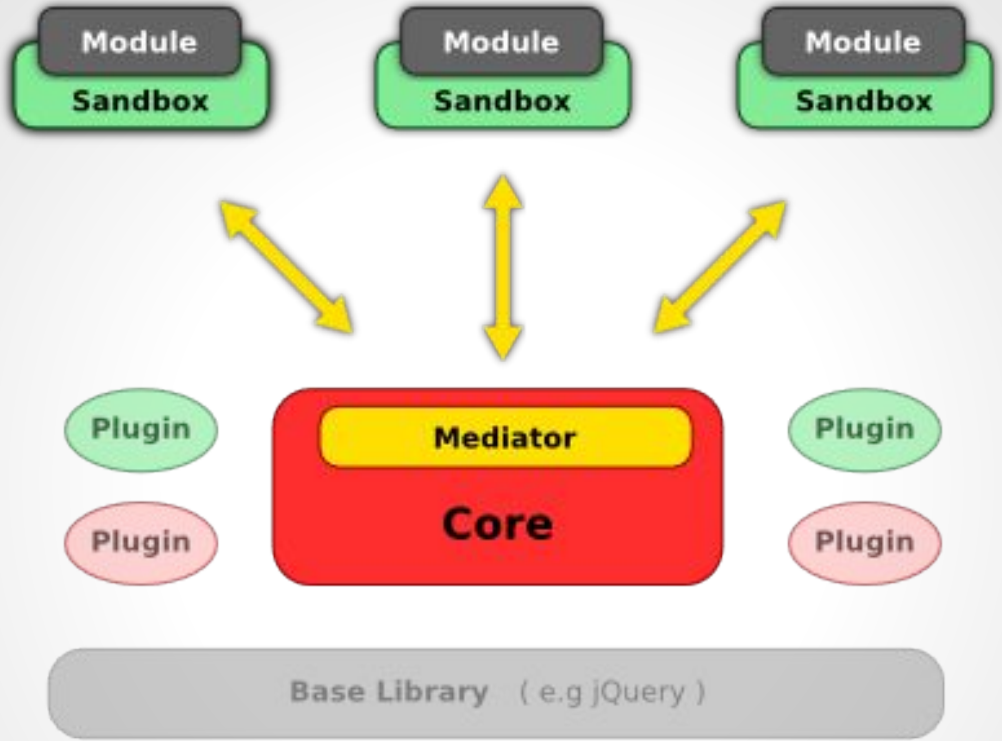
```
var mediator = (function () {
    // Приватные переменные и методы
    var modules = {},
        slice = [].slice,

        createInstance = function (moduleId, sandbox) {
            ...
            return instance;
        };

    return {
        register: function (moduleId, Creator) {
            ...
        },
        start: function (moduleId) {
            ...
        },
        stop: function (moduleId) {
            ...
        },
        startAll: function () {
            ...
        },
        stopAll: function () {
            ...
        },
        moduleIsActive: function (moduleId) {
            ...
            return returnValue;
        },
        getModule: function (moduleId) {
            ...
            return returnValue;
        }
    };

})();
```



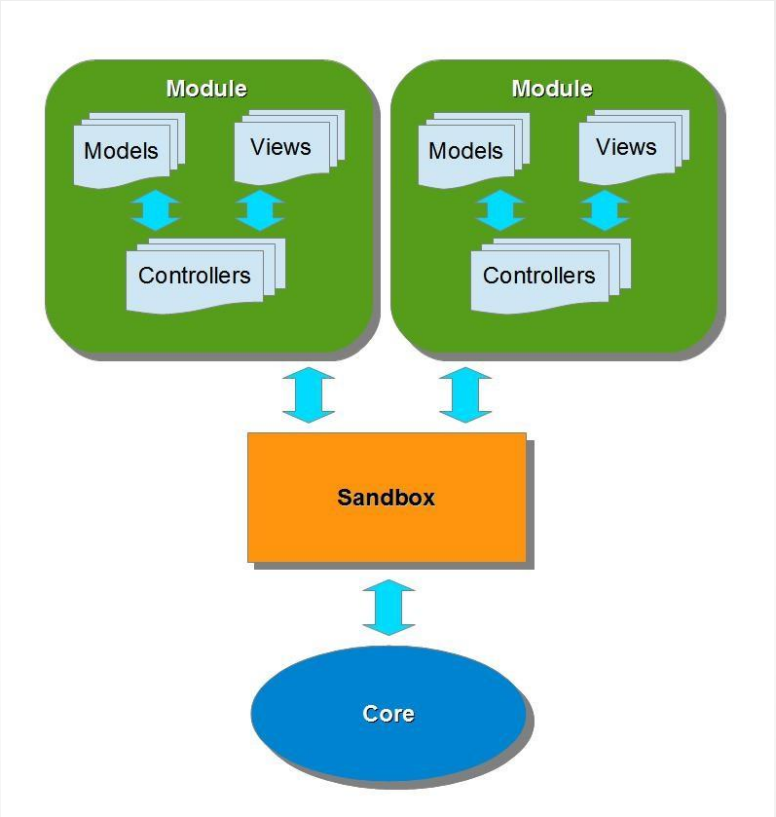
https://github.com/aranm/scalable-javascript-architecture



https://github.com/aurajs/aura



http://scaleapp.org/


Module
Models
Views
Controllers
Module
Models
Views
Controllers
Sandbox
Core

ПРОИЗВОДИТЕЛЬНОСТЬ, ПРОФИЛИРОВАНИЕ И ТЕСТИРОВАНИЕ

- Оптимизируйте циклы. Уменьшайте количество итераций в циклах.
- Уменьшайте количество операций в каждой отдельной итерации
- Используйте локальные переменные
- Поменьше трогайте DOM-дерево. А если трогаете, то сведите количество операции к минимуму.
- Кешируйте всё, что возможно. Особенно длинные цепочки св-в (избегай длинного наследования) в объектах и переменные в циклах
- Избегайте затратных операций
- Аккуратно используйте регулярки.
- Не сбрасывайте браузерные кеши на repaint и reflow.
- В больших проектах пишите unit-тесты (BDD, TDD)
- Пишите меньше кода.

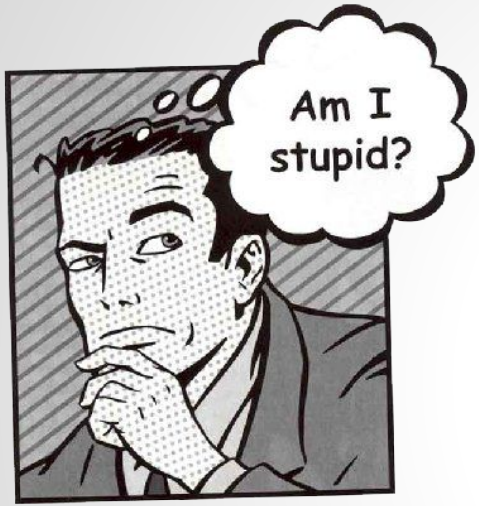
Не забывай предохраняться! (безопасность)

- Проверяй все получаемые данные
- Используй безопасные HTTP заголовки

(чтобы браузер не интерпретировал JSON как HTML итд)

- Что произошло в браузере, остается в браузере (доступ к личным данным, строгая валидация и решения по безопасности должны быть написаны на сервере)
- Не доверяй данным! js легко взломать (XSS). Не парси строки текста приходящие от куда угодно!

USE THE SOURCE AND YOUR HEAD, LUKE!



- Пользуйся документацией Mozilla Developer Network
- Задавай вопросы и ищи ответы на stackoverflow и google
- DRY (don't repeat yourself) и объединяй все что можно до разумного минимализма. не пиши лишнего кода. Не изобретай велосипед!
- Читай исходники крутых библиотек и приложений!

Спасибо за внимание!

И помните….
“Большой путь, маленькими шагами”

Чернобаев Николай. 2015

LoftSchool
от мыслителя к создателю

Что нужно сделать после вебинара?

1. Пересмотреть курсы на loftblog **Основы и продвинутый Javascript**
2. Выучить как таблицу умножения признанные Code Style
3. Переписать свой последний проект с учетом всех рекомендаций.
4. Зайти на http://todomvc.com и скачать понравившиеся фреймворки. Например Backbone.js, Angular.js, React.js и Ember.js. Установить и поиграться с ними. Так же установите примеры VanillaJS и Jquery. Понять где модель, где представление, где контроллер.
5. Для самых шустрых. Скачать архитектуру https://github.com/aranm/scalable-javascript-architecture. Понять где модули, где песочница, где ядро. Поиграться с ними, пощупать.
6. После всего этого на следующий день - пойти погулять. Лето ведь на улице ☺

LoftSchool
от мыслителя к создателю