Node.js

Часть 1

Node.js

- **Node** или **Node.js** — программная платформа, основанная на движке V8 (транслирующем JavaScript в машинный код)
- Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API (написанный на C++), подключать другие внешние библиотеки
- Node.js применяется преимущественно на сервере, выполняя роль веб-сервера, но есть возможность разрабатывать на Node.js и десктопные оконные приложения (при помощи NW.js, AppJS или Electron для Linux, Windows и Mac OS) и даже программировать микроконтроллеры

Node.js

В основе Node.js лежит событийно-ориентированное и асинхронное программирование с неблокирующим вводом/выводом.

Событийно-ориентированное программирование

- Событийно-ориентированное программирование (event-driven programming) — парадигма программирования, в которой выполнение программы определяется событиями — действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета).

Асинхронность как необходимость

- Под ***асинхронностью*** понимают асинхронный доступ к ресурсам - файлам, сокетам, данным.

Как он происходит по классической схеме работы веб-приложения?

Все просто и состоит из нескольких этапов:

  - подключиться к ресурсу (источнику данных);
  - считать из него первую порцию данных;
  - ждать, пока на нем не будет готова вторая порция данных;
  - считать вторую порцию данных;
  - ждать третью порцию данных;
  - ….
  - завершить считывание данных;

- При этом при исполнении программы возникает блокировка, вызванная тем, что установка соединения с ресурсом и (особенно) чтение из него данных требуют времени

Асинхронная событийная модель

- В основе её лежат «событийный цикл» и шаблон «reactor» (от слова «react» – реагировать).
- «Событийный цикл» представляет собой бесконечный цикл, который опрашивает «источники событий» (дескрипторы) на предмет появления в них какого-нибудь «события». Опрос происходит с помощью библиотеки «синхронного» или «асинхронного» ввода/вы-вода, который, при этом будет являться «неблокирующим» (в системную функцию ввода/вывода передаётся флаг O_NONBLOCK).

Асинхронная событийная модель

- Во время очередного витка «событийного цикла» система проходит последовательно по всем дескрипторам и пытается считать из них «события»: если таковые имеются, то они возвращаются функцией чтения в систему; если же никаких новых событий у дескриптора нет, то «блокировка» и ожидание появления «события» не произойдет, а сразу же возвратится ответ: «*новых событий нет*».

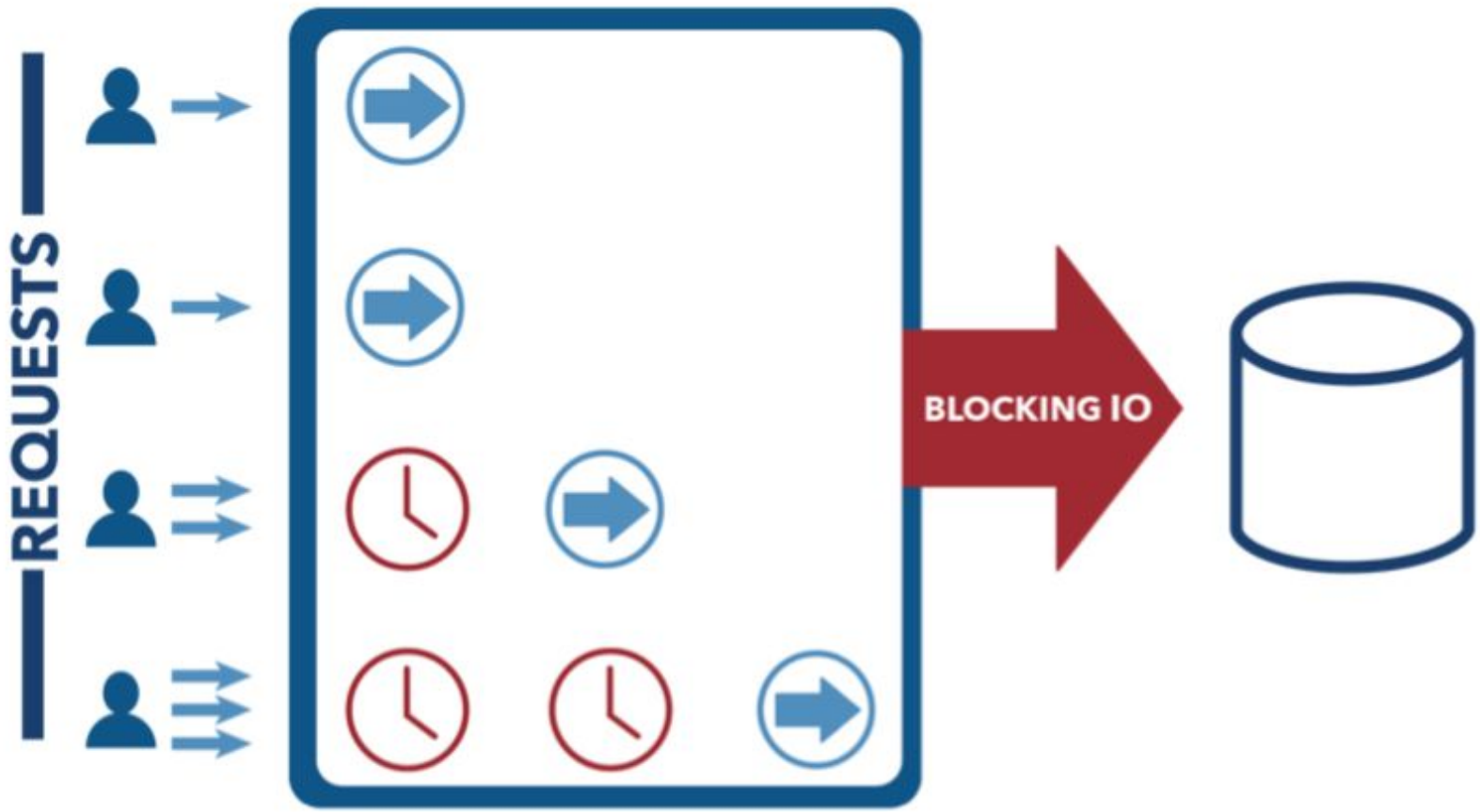
Асинхронная событийная модель

- «Событием» может быть приход очередной порции данных на сетевой сокет («socket» – дословно «место соединения») или считывание новой порции данных с жёсткого диска, в общем, любой ввод/вывод. Например, когда вы загружаете картинку на хостинг, данные туда приходят кусками, каждый раз вызывая событие «*новая порция данных картинки получена*».

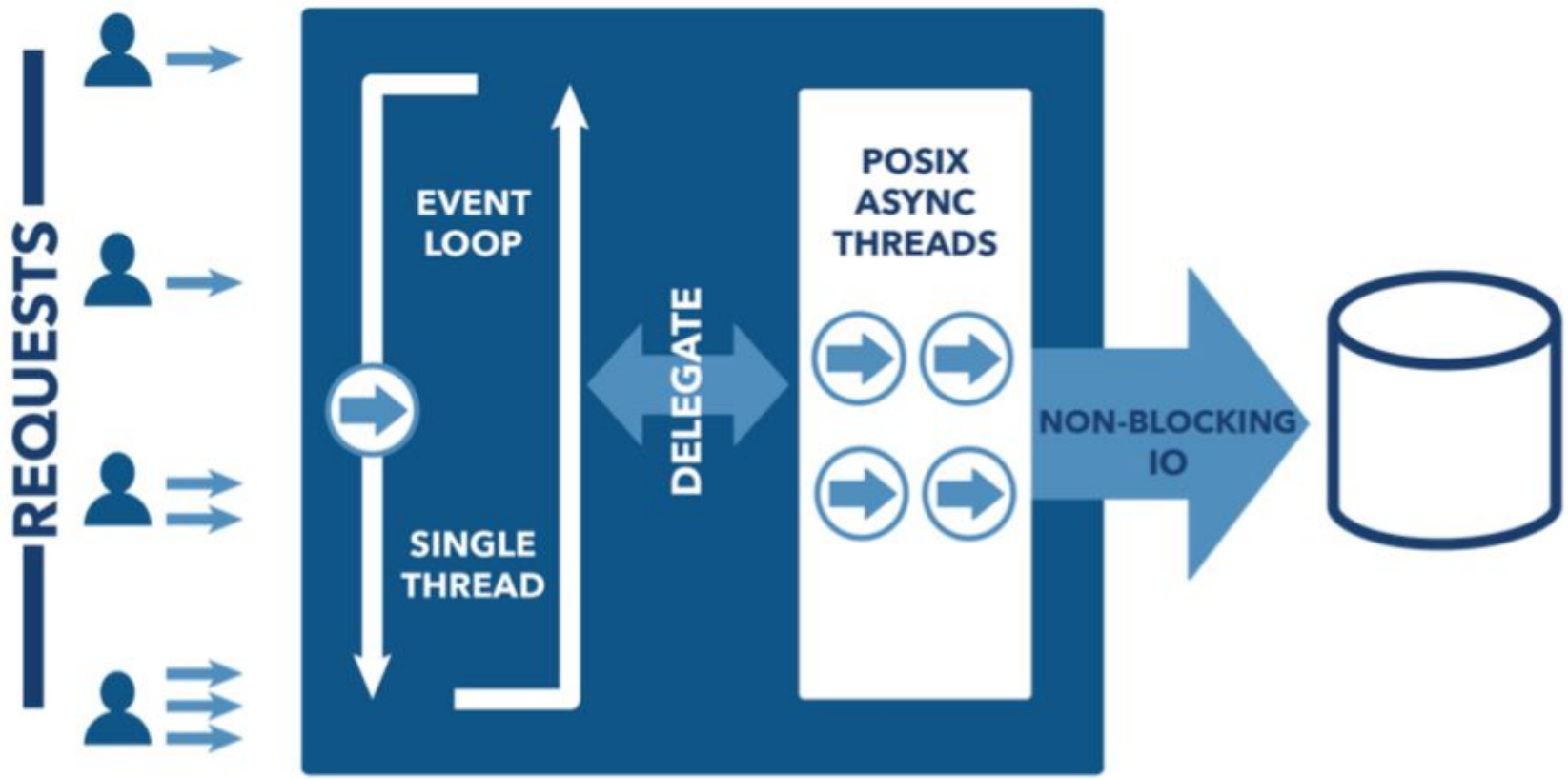
Неблокирующий ввод/вывод

- Простым подходом к вводу/выводу было бы запустить процесс доступа, а затем ждать его завершения. Но такой подход (так называемого синхронного ввода/вывода или блокирующего ввода/вывода) будет блокировать выполнение программы, в то время, как коммуникация в процессе выполнения, оставив системные ресурсы, будет простаивать на холостом ходу. Когда программа делает много операций ввода/вывода, это означает, что процессор может проводить почти все своё время простаивая в ожидании завершения операций ввода/вывода.

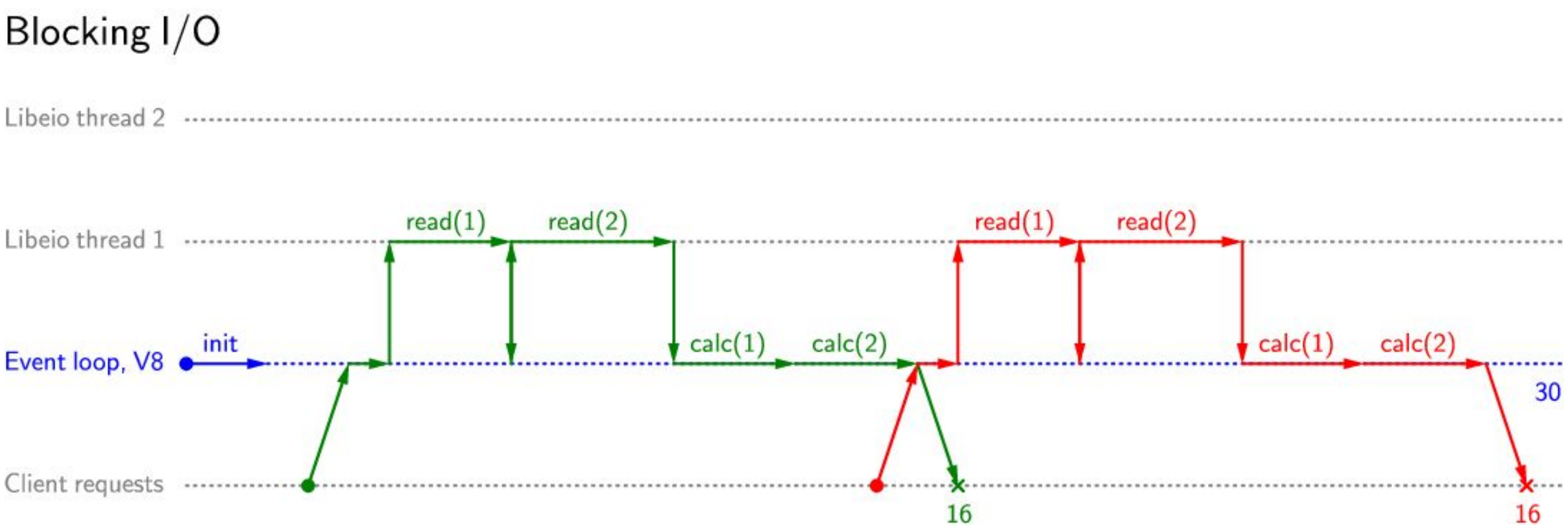
TECH | BLOCKING I/O



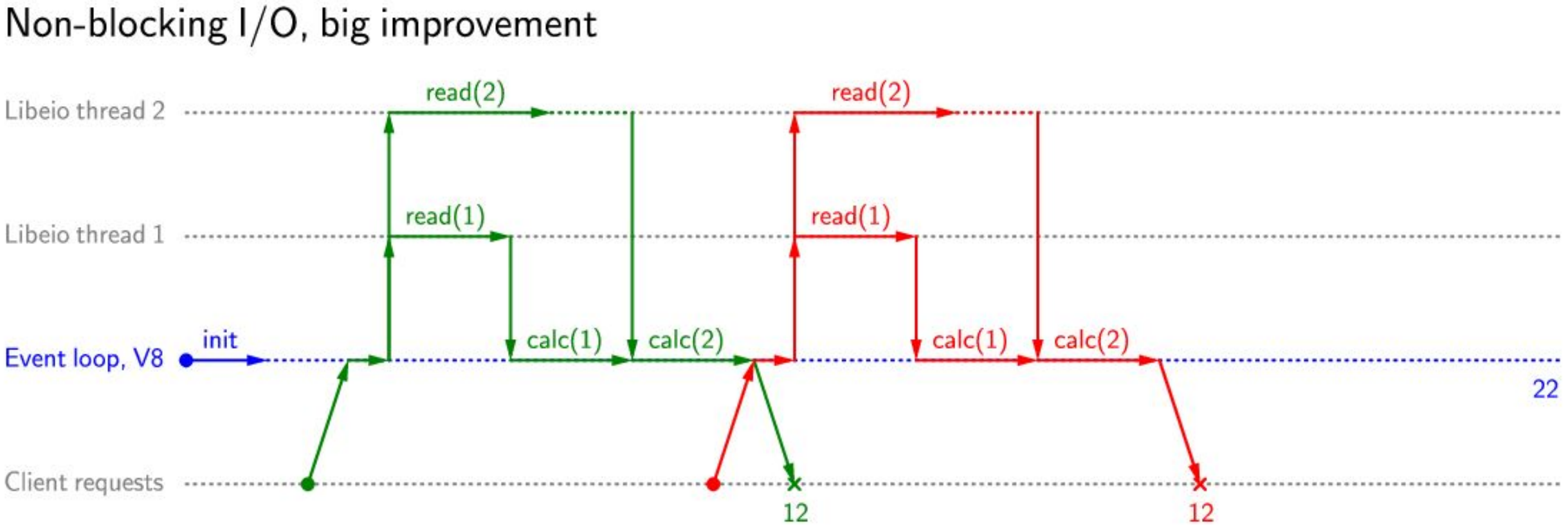
TECH | NON-BLOCKING I/O



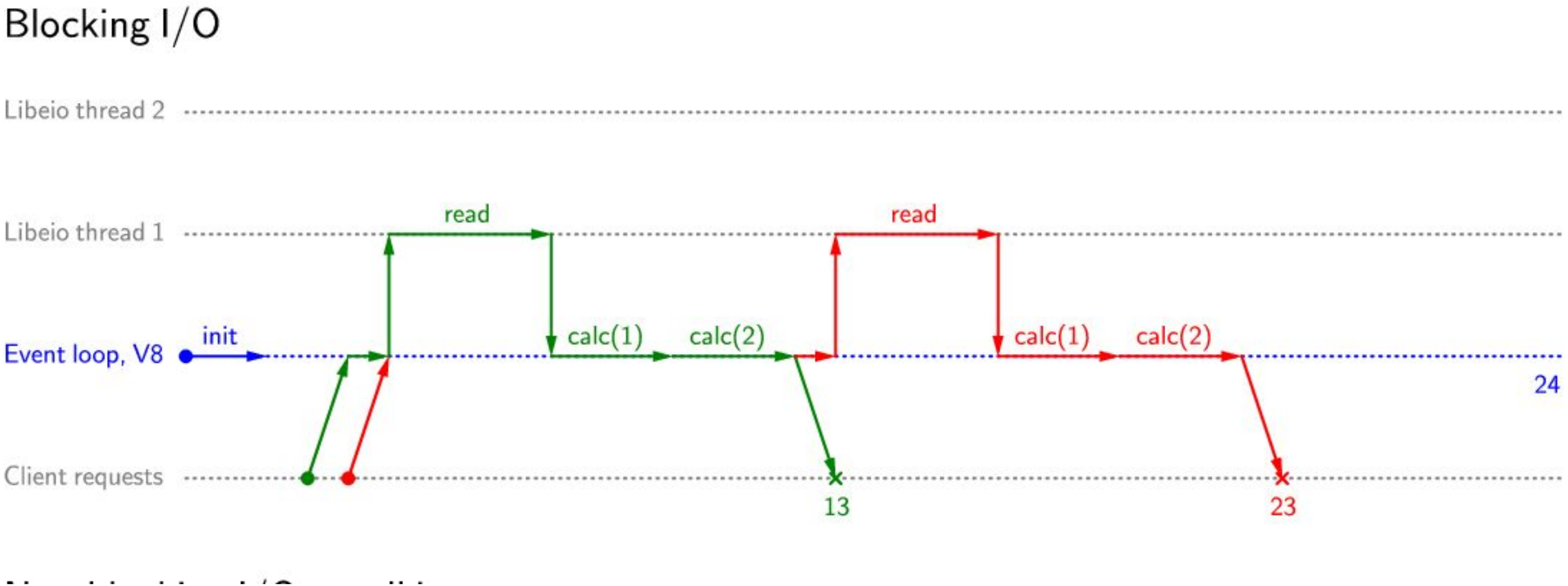
Пример обработки запросов сервером с блокирующим чтением



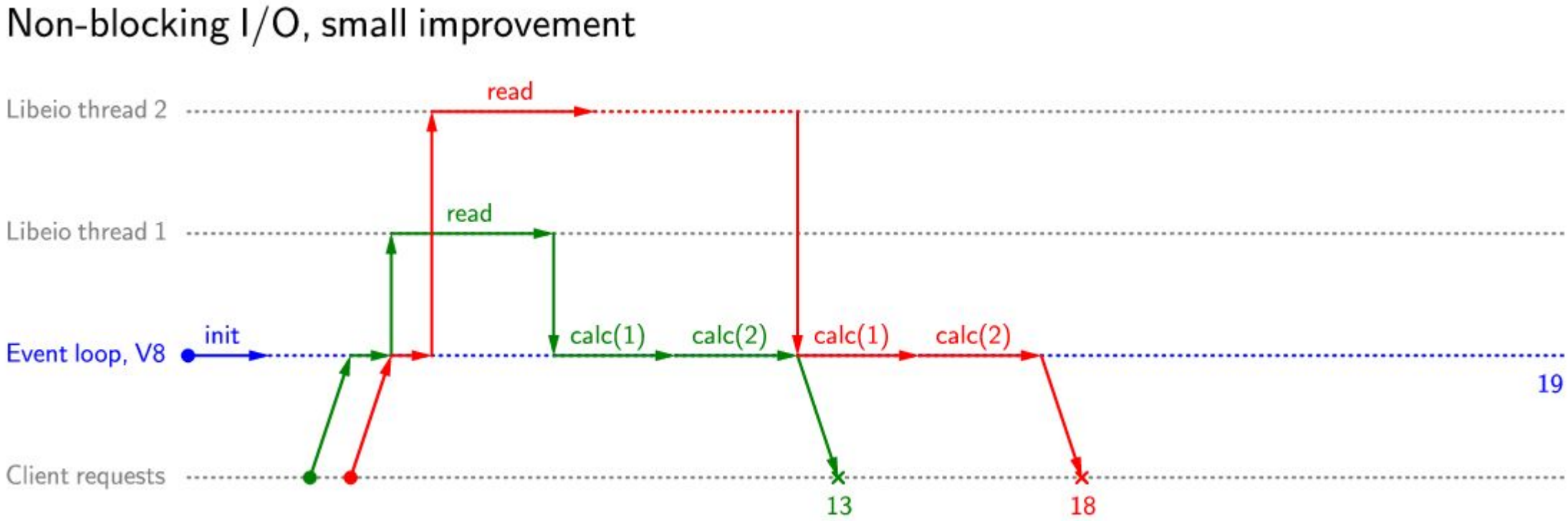
Пример обработки запросов сервером с неблокирующим



Пример уменьшения времени обработки при почти одновременном приходе двух запросов:



Пример уменьшения времени обработки при почти одновременном приходе двух запросов:



JavaScript

- JavaScript стал очень широко использоваться с распространением библиотек/фреймворков, делающих разработку сложных сценариев, приложений и веб-интерфейсов простым и приятным занятием. Я говорю прежде всего о библиотеке Prototype, заложившей основы нескольких JavaScript-фреймворков, о jQuery, ставшей практически стандартом разработки, о Ext.js, задающем новый уровень веб-интерфейсов, наконец, о MVC-среде - Backbone.js, предоставляющей каркас для создания полноценных RESТ-приложений.
- После появления платформы Node.js -JavaScript проник на сервер.

JavaScript

Замыкания и ООП-практика JavaScript

Замыкания

- Замыкание (closure) - это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции, причем в качестве её параметров, то есть, это функция, которая ссылается на некие сущности в своём контексте.
- Можно сказать, что замыкание - это особый вид функции, которая определена в теле другой функции и создаётся каждый раз во время её выполнения

Пример 1

```
function localise(greeting) {
    return function(name){console.log(greeting + ' ' + name)};
};

var english = localise('Hello');
var russian = localise('Привет');
english('closure');
russian('closure ');
```

Результат 1

> Hello closure

> Привет closure

Фактически, мы создали две разные функции, в которых фраза приветствия замкнута вмещающей функцией. Но это еще не все. Как известно, в JavaScript областью видимости локальной переменной является тело функции, внутри которой она определена. Если вы объявляете функцию внутри другой функции, первая получает доступ к переменным и аргументам последней и сохраняет их даже после того, когда внешняя функция отработает.

Пример 2

```
function counter(n) {
        var count = n;
        return function() {
                console.log(count++)
        }
};
var ct = counter(5);
ct(); //5
ct(); //6
ct(); //7
```

Функция ct() возвращает количество собственных вызовов, причем начальное значение задается при ее создании.

Пример 3

```
var i = 5;
function plusOne() {
    i++;
    return i;
}
console.log(plusOne()); //6
console.log(plusOne()); //7
console.log(i); //7
```

Это тоже замыкание, только по глобальной области видимости. Переменные же, объявленные внутри функций, имеют область видимости, ограниченную рамками этой функции, и замыкание происходит именно по ней -пусть даже и сама внешняя функция завершена

Применение замыканий

Самым очевидным применением замыканий будет конструкция вроде ” фабрики функций” (не буду называть это Abstact Factory, дабы не вызвать гнев приверженцев чистоты терминологии):

```
function createFunction(n) {
    return function(x) {
        return x * n;
    }
}
var twice = createFunction(2);
var threeTimes = createFunction(3);
var fourTimes = createFunction(4);
console.log(twice(4) t " " + fourTimes(5)); //8 20
```

Применение замыканий

Как известно, в JavaScript отсутствуют модификаторы доступа, а оградить переменную от нежелательного влияния иногда ой как необходимо. Использование замыкания нам может организовать это не хуже, чем модификатор private

```
function counter() {
    var count = 0;
    this.getCount = function() { return count; }
    this.setCount = function(n) { count = n; }
    this.incrCount = function() { count++; }
}
var ct = new counter();
ct.setCount(5);
ct.incrCount();

console.log(ct.getCount()); //6
console.log(ct.count); //undefined
```

Применение замыканий

Методы также могут быть приватными

```
function counter() {
   var count = 0;
   function setCount(n) { count = n; }
   return {
        safeSetCount: function(n) {
           if (n != 13) { setCount(n); }
           else { console.log("Bad number!"); }
        },
        getCount: function() { return count; },
        incrCount: function() { count++; }
   }
}
ct = new counter();
ct.safeSetCount(5);
ct.safeSetCount(3); //Bad number!
```

Тут попытка вызвать метод ct.SetCount() снаружи немедленно приведет к ошибке.

ООП в JavaScript

- Поддерживает ли JavaScript парадигму объектно-ориентированного программирования?

С одной стороны, да, безусловно, с другой - при изучении реализации ООП в JavaScript большинство специалистов, привыкших к разработке на C++ или java, в лучшем случае отмечают своеобразие этой самой реализации.

Ну действительно, есть объекты, но где классы? Как наследовать? А инкапсулировать? Как с этим вообще можно работать?

Создание объекта с нуля

Обычно делают так:

```
var user_vasya = {
    name: "Vasya",
    id: 51,
    sayHello: function() {
        console.log("Hello " + this.name);
    }
};
user_vasya.sayHello(); // Hello Vasya
```

Создание объекта с нуля

Или так:

```
var user_masha = {}
user_masha.name = "Masha";
user_masha.uid = 5;
user_masha.sayHello = function() {
    console.log("Hello " + this.name);
}
user_masha.sayHello(); // Hello Masha
```

Чего-то не хватает

- Все просто и понятно, но для полноценной ООП-разработки явно чего-то не хватает. Представьте, что нам нужно получить полсотни однотипных объектов. Ну, или добавить к объекту некоторую инициализацию - установку начальных параметров. Иными словами, я говорю о необходимости конструктора. Его тут нет.

Создание Объекта

Попробуем создать объект другим способом. Мы знаем, что в JavaScript функция - это всегда объект. Причем объект первого класса (first class object), сущности, которые могут быть переданы, как параметр, возвращены из функции, присвоены переменной.

```
var User = function(name, id) {
    this.name = name;
    this.id = id;
    this.sayHello = function() {
        console.log("Hello " + this.name;
    }
}
```

Это объект?

Проверим:

```
console.log(typeof User); // function
var John = User("John",51);
console.log(typeof John); // undefined
John. sayHello () // John is undefined
```

new

Собственно, ничего удивительного - чтобы создать объект, нужно воспользоваться специальным методом, хорошо нам знакомым по другим ООП-языкам:

```
var John = new User{"John",51);
console.log{typeof John); // object
John.sayHello{) // Hello John
```

Теперь все в порядке. Чтобы это стало совсем очевидно, мы можем в JavaScript-консоли браузера Google Chrome посмотреть доступные свойства объекта John. Среди них есть метод consructor, значение которого - тело функции User.

prototype

В первом приближении наша цель достигнута, но хотелось бы еще и расширения функционала базового … нет, не класса, в данном случае конструктора. Это делается с помощью объекта **prototype**, который есть у каждого JavaScript-объекта и принадлежит полю constructor

```
var User = function (name, id) {
    this.name = name;
    this.id = id;
    this.sayHello : function () { console.log("Hello "+ this.name);  }
}
User.prototype.sayHi: function() { console.log("Hi "+ this.name); }
John.sayHi{); // Hi John
```

prototype

Новый метод создан и работает. Обратите внимание: мы создали его уже после создания нового объекта -он все равно в этом объекте появится. Важно понимать, что объект prototype является дополнением к конструктору. При обращении к некоторому свойству или методу объекта сначала он будет искаться в конструкторе, потом в объекте prototype:

```
var User = function (name, id) {
   this.name = name;
   this.id = id;
   this.sayHello : function () { console.log("Hello "+ this.name);  }
}
User.prototype.sayHello: function() { console.log("Hi "+ this.name); }
John.sayHello{); // Hello John
```

Еще один важный момент - метод, созданный посредством объекта Prototype, принадлежит конструктору и является общим для всех.

Наследование

- Напомним: сам prototype представляет собой объект, а это, в частности, обозначает, что у него тоже есть свой prototype, а у того - свой. Эту цепочку прототипов можно использовать для реализации наследования.
- Вообще, проблема наследования в JavaScript всегда стояла остро, особенно для разработчиков на языках с классической моделью. Достаточно часто можно наблюдать попытки решить ее «в лоб», копируя методы объекта prototype или сам объект целиком.

Ошибки при наследовании

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

User.prototype.sayHello = function() { console.log("Hello!"); }
var Admin = function(){ };

Admin.prototype = User.prototype;
admin = new Admin();
admin.sayHello(); //Hello!
```

Но это и подобные ему решения неудовлетворительны. В частности, в результате выполнения вышеприведенного кода объекты будут просто иметь общий объект prototype, и добавление новых методов Admin автоматически добавит их в User.

Ошибки при наследовании

Можно, конечно, делать все аккуратнее, не копируя prototype целиком, но тогда возникнет другая проблема - новые методы User придется каждый раз добавлять в Admin.

```
Admin.prototype.sayHello = User.prototype.sayHello;
admin = new Admin();
admin.sayHello();
```

Принципиальную порочность такого пути можно выразить в двух строчках кода:

```
console.log(admin instanceof Admin); // true
console.log(admin instanceof User); // false
```

Правильное наследование

Настоящее же наследование следует организовывать, учитывая структуру объекта JavaScript. Условием «правильного» наследования должно быть то, что прототип дочернего класса является экземпляром родительского класса:

```
var User = function(name, id)  {
    this.name = name;
    this.id = id;
}
User.prototype.sayHello = function () { console.log("Hello"+this.name);}
var Admin = function (} {};
```

Правильное наследование

```
Admin.prototype = new User();
var admin = new Admin();
admin.name = "Ian";
admin.sayHello(); // Hello Ian

console.log(admin instanceof Admin); // true
console.log(admin instanceof User); // true

Admin.prototype.sayHi = function () {  console.log("Hi "+this.name); }
var user = new User("John", 52);
user.sayHi(); //TypeError: Object {object Object) has no method 'sayHi'
```

Явление Node.js

Платформа Node.js была создана в 2009 году Райном Далом (Ryan Dahl) в ходе исследований по созданию событийно-ориентирован-ных серверных систем.

Асинхронная модель была по причине низких накладных расходов (по сравнению с многопоточной моделью) и высокого быстродействия. Node была (и остается) построена на основе JavaScript-движка V8 с открытым исходным кодом, разработанного компанией Google в процессе работы над своим браузером Google Chrome.

Установка Node

Нужно получить необходимый вам вариант дистрибутива (в зависимости от используемой операционной системы) на странице загрузки Node.js - http://nodejs.org/download/

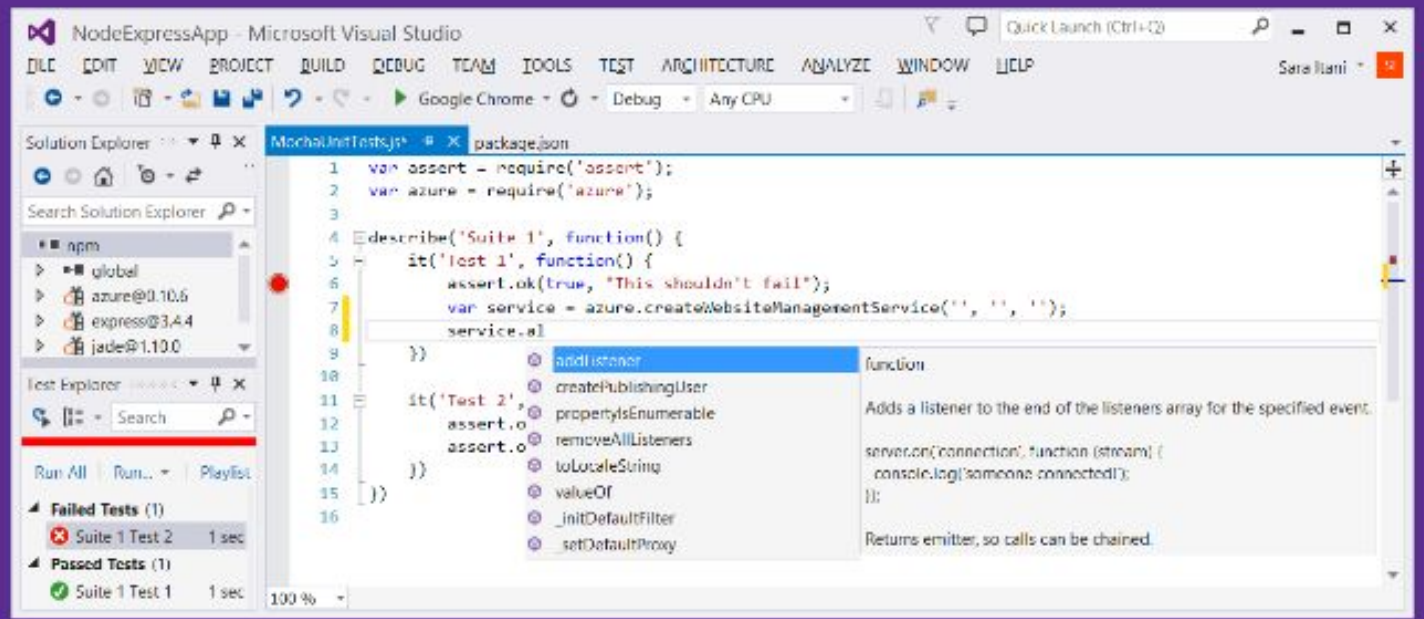
Или

https://www.visualstudio.com/en-us/features/node-js-vs.aspx

Node.js Tools for Visual Studio

Turn Visual Studio into a powerful Node.js development environment.

- Free and open source
- Intelligent code completion
- Advanced debugging and profiling
- Integration with other Visual Studio features and 3rd party tools
- Node.js, io.js, JavaScript, TypeScript, HTML, CSS, and JSON support



1 Get Visual Studio Community

2 Get Node.js Tools for VS

Source and Documentation ›

Создание проекта

? ×

▷ Последние файлы

◢ Установленные

◢ Шаблоны

▷ Visual C++

◢ Другие языки

Игра

Ускоритель сборки

▷ Visual C#

▷ Visual Basic

◢ JavaScript

Apache Cordova Apps

Node.js

▷ Windows

SQL Server

▷ Visual F#

PowerShell

▷ Python

▷ TypeScript

▷ Другие типы проектов

Примеры

▷ В сети

.NET Framework 4.5.2 | Сортировка по: По умолчанию | Установлено: Шаблоны — поиск (Ctr

Приложение навигации (Windows 8.1)	JavaScript
Приложение с Grid (Windows 8.1)	JavaScript
Приложение с Hub (Windows 8.1)	JavaScript
Приложение со Split (Windows 8.1)	JavaScript
Приложение навигации (Windows Phone)	JavaScript
Приложение с Pivot (Windows Phone)	JavaScript
From Existing Node.js code	JavaScript
Blank Node.js Console Application	JavaScript
Blank Node.js Web Application	JavaScript
Starter Node.js Express 3 Application	JavaScript

Щелкните здесь для поиска шаблонов в Интернете.

Тип: JavaScript

An empty Node.js application.

app.js

```
console.log('Hello world');
```

Имя: NodejsConsoleApp1

Веб сервер Node.js

Веб-сервер на Node.js действительно укладывается в несколько строк кода. Вот они (файл start.js):

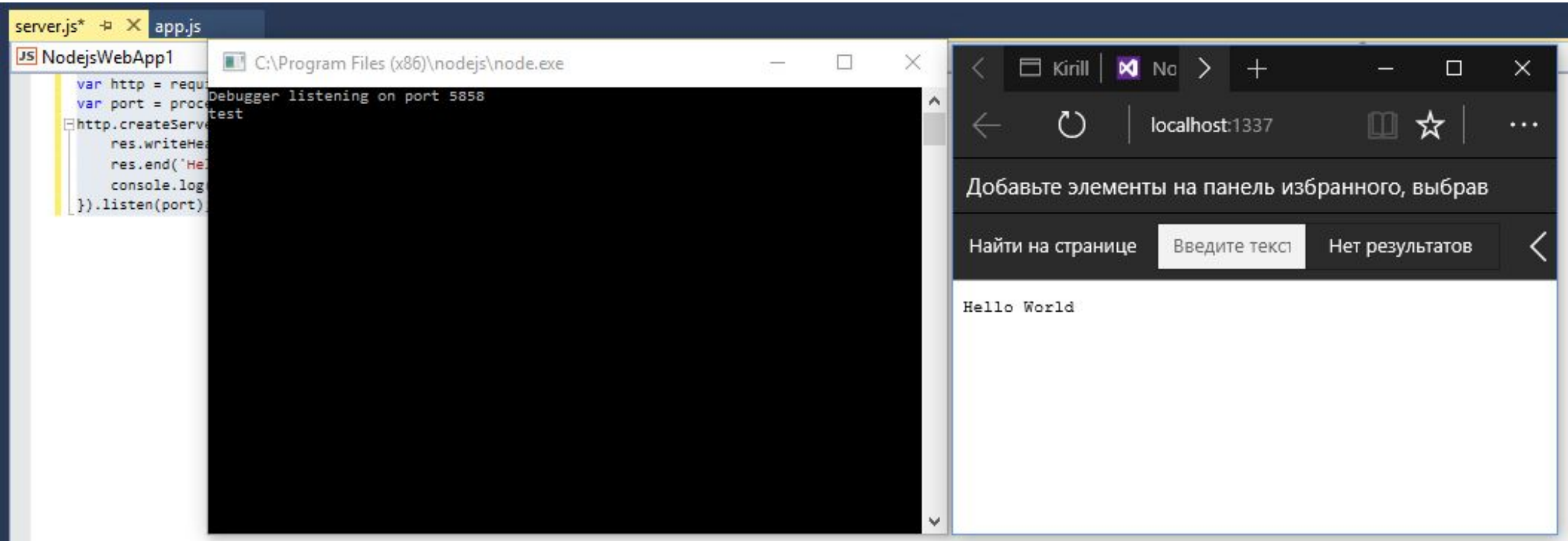
```
var http = require('http');
var port = process.env.port || 1337;
http.createServer(function (req, res) {
    console.log(“HTTP Works");
}).listen(port);
```

Веб сервер Node.js

- Метод createServer() объекта http принимает в качестве аргумента анонимную функцию обратного вызова, аргументами которой, в свою очередь, служат объекты request и response. Они соответствуют, как нетрудно догадаться, поступавшему HTTP-запросу и отдаваемому HTTP-ответу.
- Правда, браузеру в этом случае ничего, кроме 200-го ответа в заголовке, от сервера не придет. Но это нетрудно исправить.

Веб сервер Node.js

```
var http = require('http');
var port = process.env.port || 1337;
http.createServer(function (req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello World\n');
    console.log("test");
}).listen(port);
```


server.js*
app.js
NodejsWebApp1
var http = requ
var port = proc
http.createServe
res.writeHe
res.end('He
console.log
}).listen(port)
C:\Program Files (x86)\nodejs\node.exe
Debugger listening on port 5858
test
Kirill
No
localhost:1337
Добавьте элементы на панель избранного, выбрав
Найти на странице
Введите текст
Нет результатов
Hello World

Веб-страницы

Что нам еще нужно от веб-сервера? Нужно, чтобы он отдавал веб-страницы. Ну, это тоже можно. Приготовим простую веб-страницу page.html

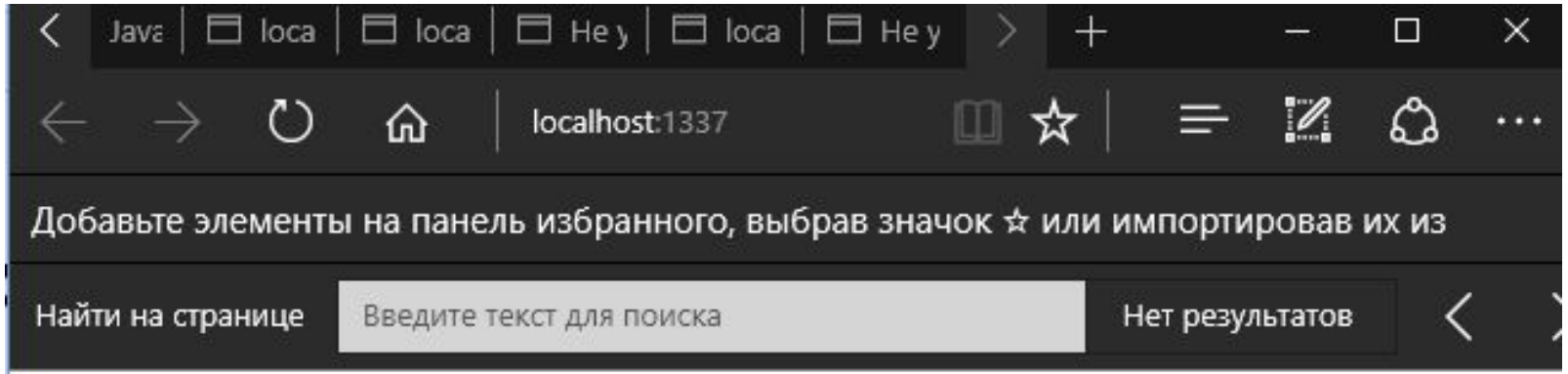
```
<html>
<head>
   <title>Node-page</title>
   <link rel=stylesheet href=styles.css type=text/css>
</head>
<body>
   <hl>Просто страница</hl>
</body>
</html>
```

Веб-страницы

```
var http = require('http');
var port = process.env.port || 1337;
var fs = require('fs');
var fileName = "page.html";
http.createServer(function (req, res) {
    fs.readFile(fileName, 'utf8', function (err, data) {
        if (err) { console.log('Could not find or open file for reading\n');  }
        else {
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end(data);
        }
    });
}).listen(port);
```

Веб-страницы

Сначала запрашиваем еще один важный Node-модуль, отвечающий за работу с файловой системой. Затем при обработке запроса читаем файл page.html в нужной кодировке и записываем его в ответ сервера. Обратите внимание, что две последние операции мы выполняем опять в теле анонимной функции обратного вызова - чтение файла методом readFile() также проходит в асинхронном режиме.

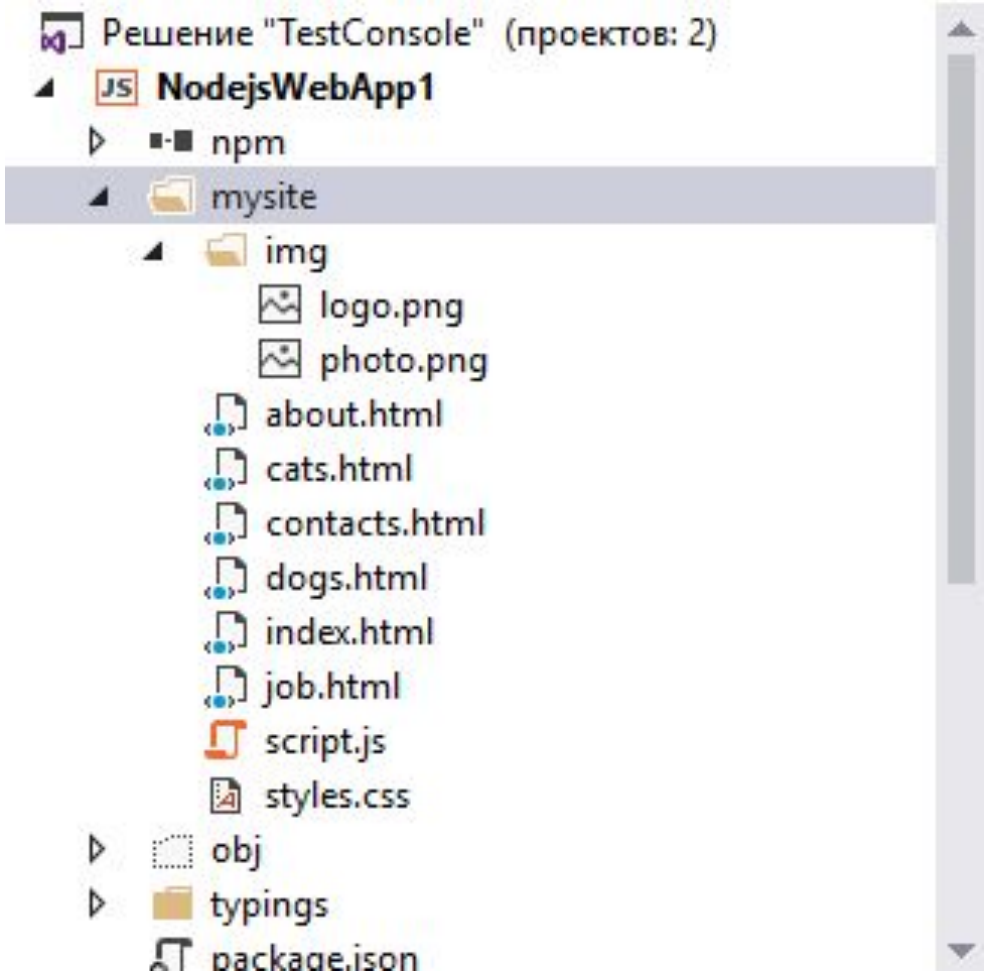


Просто страница

Сайт на Node.js

Вообще, один из первых вопросов, которые обычно возникают у веб-разработчика, приступающего к изучению Node.js, - это: можно ли в этой среде построить обычный «плоский» сайт из статических страниц и как, это сделать? Оставим в стороне вопрос, предназначен ли Node.js для подобных задач, а просто попытаемся реализовать эту конструкцию.

Структура сайта



Сайт на Node.js (url модуль)

Подключим еще один модуль

```
var http = require('http');
var fs = require('fs');
var url = require("url");
```

Он реализует методы работающие с различными составляющими URL. Посмотрим на его работу:

```
    fs.readFile(fileName, 'utf8', function (err, data) {
    var pathname = url.parse(request.url).pathname;
    console.log("Получен запрос " + pathname);
```

Сайт на Node.js (Index.html)

```
<html>
<head>
    <title>Node-page</title>
    <link rel=stylesheet href=styles.css type=text/css>
    <script src=script.js type=text/javascript></script> </head>
    <body>
        <h1><img src="img/logo.gif" />Сайт на Node.js</h1>
        <div class="main menu">
        <p><a href="about.html">О нас</a></p>
        <p><a href="cats.html">Кошечки</a></p>
        <p><a href="contacts.html">Контакты</a></p>
        <p><a href="job.html">Вакансии</a></p>
        </div>
        <img src="my_large_photo.jpg" />
    </body>
</html>
```

Сайт на Node.js (запросы)

Теперь при запуске браузера с адресом
http://localhost:1337

мы будем наблюдать в консоли такую картину:

Получен запрос /
Получен запрос /my_large_photo.jpg
Получен запрос /styles.css
Получен запрос /script.js
Получен запрос /img/1090.gif
Получен запрос /img/my_large_photo.jpg
Получен запрос /favicon.ico

Тут представлено все разнообразие контента, которое нам надо в нужном формате отдать браузеру.

Если бы этот контент состоял только из html-страниц, можно было бы ограничиться следующим кодом:

```
http.createServer(function (req, res) {
    var pathname = url.parse(request.url).pathname;
    if (pathname == '/') { pathname = '/index.html'; }
    pathname = pathname.substring(1, pathname.length);
    fs.readFile(pathname, 'utf8', function (err, data) {
        if (err) {
            console.log('Could not find or open file for reading\n');
        }
        else {
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.end(data);
        }
    });
}).listen(port);
```

Сайт на Node.js (модуль path)

- К сожалению (или к счастью, я за разнообразие), тут у нас присутствуют не только javaScript- и CSS- файлы, которые современный браузер, поморщившись, примет и как 'text/html', но и изображения, для которых правильный Content-type обязателен. Поэтому дополним наш код. Сначала добавим еще один небесполезный модуль:

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

Сайт на Node.js (mimeTypes)

Далее создадим объект с mime-типами:

```
var mimeTypes =
{
    '.js' : 'text/javascript',
    '. html': 'text/html',
     .css' : 'text/css’ ,
     .jpg' : 'image/jpeg',
    '. gif' : 'image/gif'
};
```

Сайт на Node.js (измененный код выдачи)

```
fs.readFile(pathname, 'utf8', function (err, data) {
    if (err) {
        console.log('Could not find or open file ' + pathname + ' for reading\n');
        }
        else{
            response.writeHead(200, { 'Content-Type': mimeTypes[path.extname(pathname)] });
            response.end(data);
        }
    });
```

Сайт на Node.js (бинарные данные)

- Уже лучше, но картинок мы все равно не увидим. Причина проста - это бинарные данные, и читаются они другим способом.

```
var http = require('http');
var fs = require('fs');
var url = require("url");
var path = require('path');
var port = process.env.port || 1337;
var mimeTypes =
{
    '.js': 'text/javascript', '.html': 'text/html', '.css' : 'text/css', '.jpg' : 'image/jpeg', '.gif' : 'image/gif', '.png' :'image/png'
};
http.createServer(function (req, res) {
    var pathname = url.parse(req.url).pathname;
    if (pathname == '/') { pathname = '/index.html'; }
    pathname = pathname.substring(1, pathname.length);
    var extname = path.extname(pathname);
    var mimeType = mimeTypes[extname];
    fs.readFile(pathname, function (err, data) {
        if (err) {
            console.log('Could not find or open file ' + pathname + ' for reading\n');
        } else {
            res.writeHead(200, { 'Content-Type': mimeTypes[extname] });
            res.end(data);
        }
    });
}).listen(port);
```

Сайт на Node.js

О нас

Кошечки

Контакты

Вакансии

Заключение

- Были кратко рассмотрены основные понятия лежащие в основе Node.js: событийно-ориентированная асинхронная модель программирования, неблокирующий ввод-вывод.
- Два важных понятия лежащих в основе JavaScript: замыкания и ООП в JavaScript.
- Рассмотрены примеры создания простого веб-сервера и веб-сайта с использованием Node.js.

В следующей лекции

- Node Core
- Как все работает? Event loop в Node.js
- Глобальные объекты (Globals)
- Процессы
- События Объект EventEmitter
- Модули
- Работа с файлами

Список литературы

- Сухов К. К. Node.js. Путеводитель по технологии. - М.: ДМК Пресс, 2015. 416 с.: ил
- https://habrahabr.ru/post/112977/
- https://ru.wikipedia.org/wiki/Node.js
- Пауэрс Ш. Изучаем Node.js. - СПб.: Питер, 2014. - 400 с: ил. - (Серия "Бестселлеры O'Reilly").