**Экзаменационные вопросы по дисциплине**

**«Программирование серверных кроссплатформенных приложений»**

**для студентов 3-го курса специальности ПОИТ**

1. Протокол HTTP, основные свойства HTTP, структура запроса и ответа. Понятие web-приложения, структура и принципы работы web-приложения. Понятие асинхронности.

HTTP-протокол – протокол передачи данных прикладного уровня, ассиметричный (сообщения от клиента к серверу и от сервера к клиенту разные). Всегда подразумевает пару request/response.

Относится к протоколу, который не помнит своего состояния. В запросе и ответе нет никаких ссылок на предыдущий и последующий ответ и запрос.

Каждый запрос-ответ – новый жизненный цикл HTTP (stateless протокол).

основные свойства

* версии HTTP/1.1 – действующий (текстовый), HTTP/2 – черновой (не распространен, бинарный);
* два типа абонентов: клиент и сервер;
* два типа сообщений: request и response;
* от клиента к серверу – request;
* от сервера к клиенту – response;
* на один request всегда один response, иначе ошибка;
* одному response всегда один request, иначе ошибка;
* TCP-порты: 80, 443;
* для адресации используется URI или URN;
* поддерживается W3C, описан в нескольких RFC.

1. **Request**:

* метод;
* URI;
* версия протокола (HTTP/1.1);
* заголовки (пары: имя/заголовок);
* параметры (пары: имя/заголовок);
* расширение.

1. **Response:**

* версия протокола (HTTP/1.1);
* код состояния (1xx, 2xx, 3xx, 4xx, 5xx);
* пояснение к коду состояния;
* заголовки (пары: имя/заголовок);
* расширение.

web-приложения - Веб-приложения — клиент-серверное приложение в котором клиент взаимодействует с сервером по протоколу HTTP.

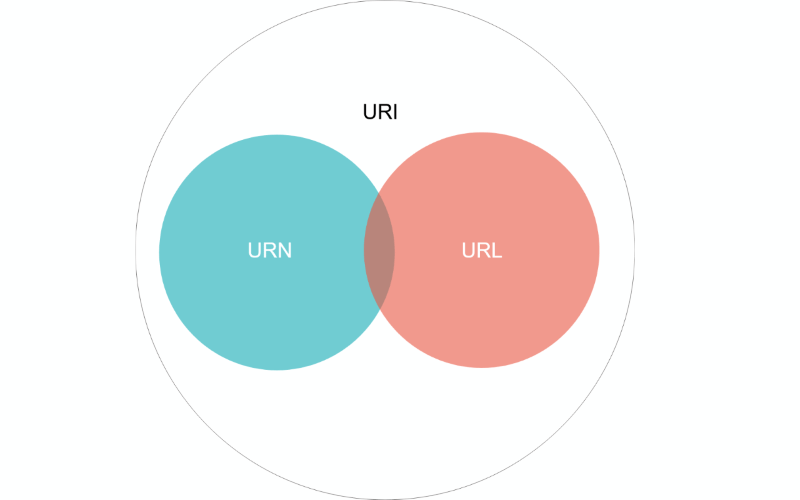




**Uniform Resource Identifier** – унифицированный **идентификатор** ресурса (документ, изображение, файл, служба, электронная почта, …).

**Uniform Resource Location** - унифицированный **локатор** ресурса, содержащий местонахождение ресурса и способ обращения (протокол) к ресурса, описывает множество URI.

**Uniform Resource Name** -унифицированное **имя** ресурса – URI, имя ресурса, не содержащее месторасположение и способ доступа к ресурсу. В будущем URN должен заменить URL (для решения проблем с перемещением ресурсов в Internet).



для взаимодействия между клиентом и сервером в соответствии с правилами (спецификацией, протоколом) должно быть установлено **соединение**; **инициатором соединения всегда является клиент**.

операция называется асинхронной, если ее выполнение осуществляется в 2 фазы:

1) заявка на исполнение;

2) получение результата;

при этом участвуют два механизма:

A-механизм, формирующий заявку и потом получающий результат;

B-механизм, получающий заявку от A, исполняющий операцию и отправляющий результат A;

продолжительность исполнения операции B-механизмом, как правило, непредсказуемо;

в то время пока B-механизм исполняет операцию, А-механизм выполняет собственную работу.

Применение асинхронности не противоречит применению многопоточности.

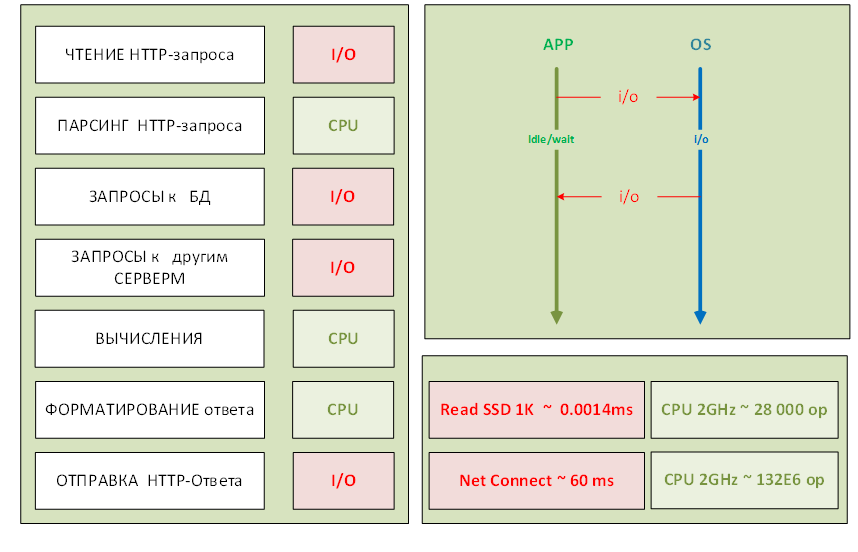
Асинхронность в программировании — выполнение процесса в неблокирующем режиме системного вызова, что позволяет потоку программы продолжить обработку. Реализовать асинхронное программирование можно несколькими способами, о которых вы узнаете ниже.

1. Web-сервер. Ресурсы, потребляемые web-сервером. Блокирующие и неблокирующие операции ввода/вывода. Решение проблемы блокирующего ввода/вывода. Понятия конкурентность и параллельность. Закон Амдала.

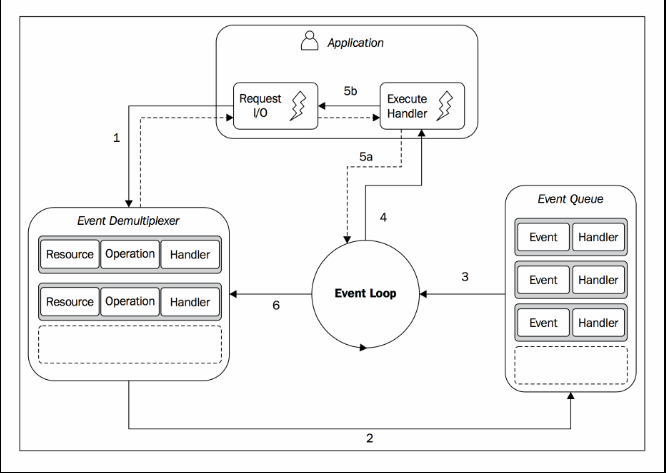
серверная части web-приложения или иначе ***web-сервер***

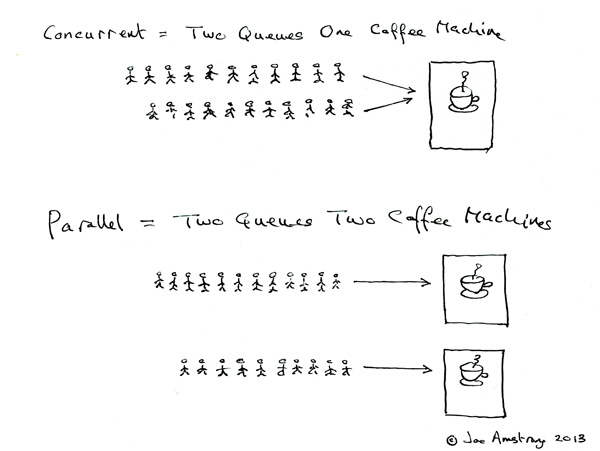
**Web-ресурс приложения:** сущность, расположенная на стороне сервера и имеющая URL/URI, к которой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

**Web-ресурсы приложения: статические** - отправляются клиенту без изменения (html-страницы, рисунки, видео-файлы, …), **динамические** – динамически (программно) формируются на сервере и отправляются клиенту (сервлеты, JSP, http-обработчики, aspx-страницы,…). Ресурс может быть статическим относительно сервера и динамическим относительно клиента (html-страницы с JavaScript).

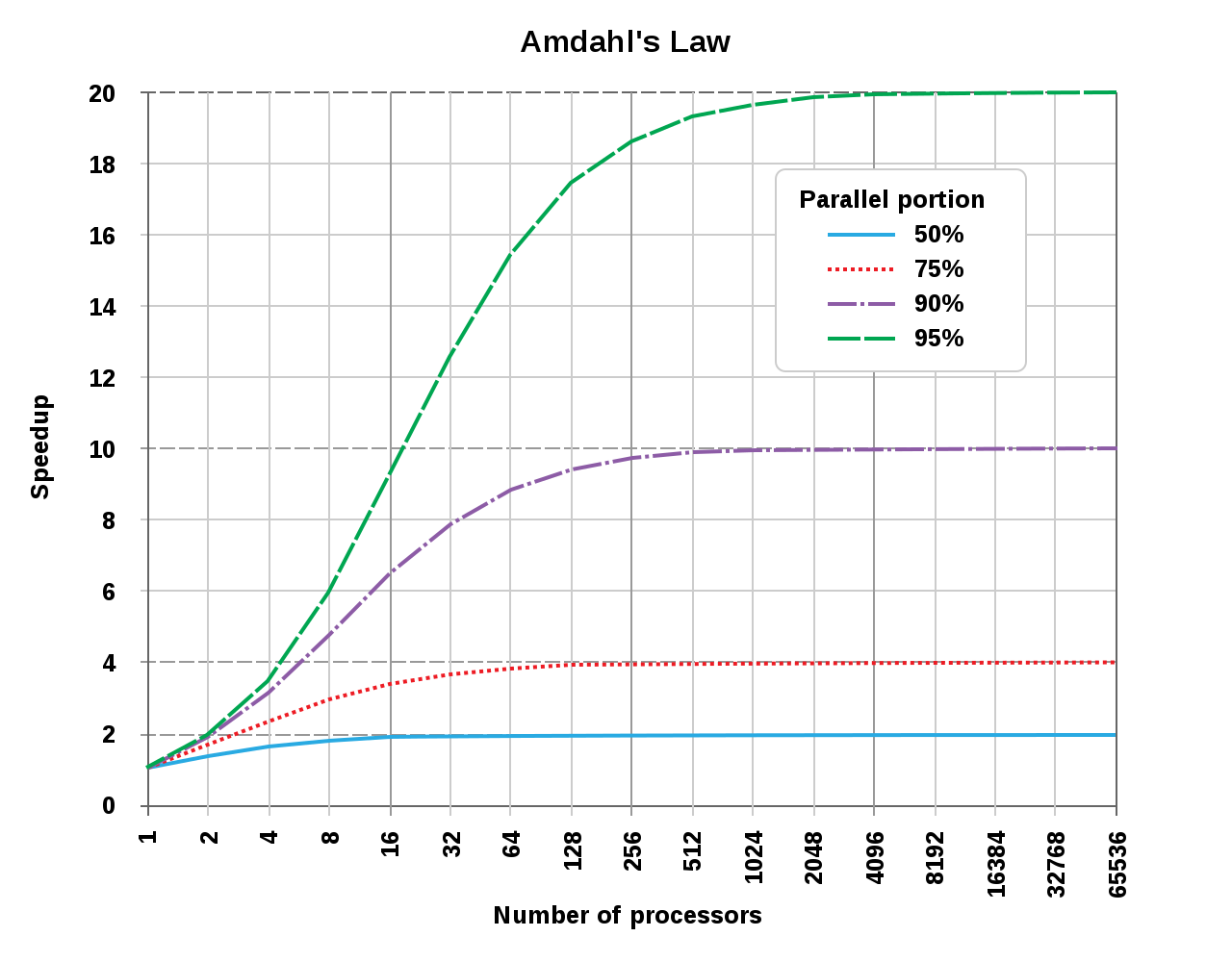


подхода для решения проблем блокирующего ввода/вывода: 1) применение многопоточности (ограничение по количеству потоков, каждый поток требует дополнительной памяти); 2) применение паттерна Reactor. Apache – многопоточность, Nginx – Reactor. паттерн Reactor –шаблон проектирования. Используется при обработке параллельных запросов к сервису. Сервисный обработчик разбирает прибывшие запросы и синхронно перенаправляет их на соответствующие обработчики запросов.





Зако́н Амдала  — иллюстрирует ограничение роста [производительности](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%BE%D1%89%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%B0) [вычислительной системы](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) с увеличением количества [вычислителей](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BD%D1%82%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80).



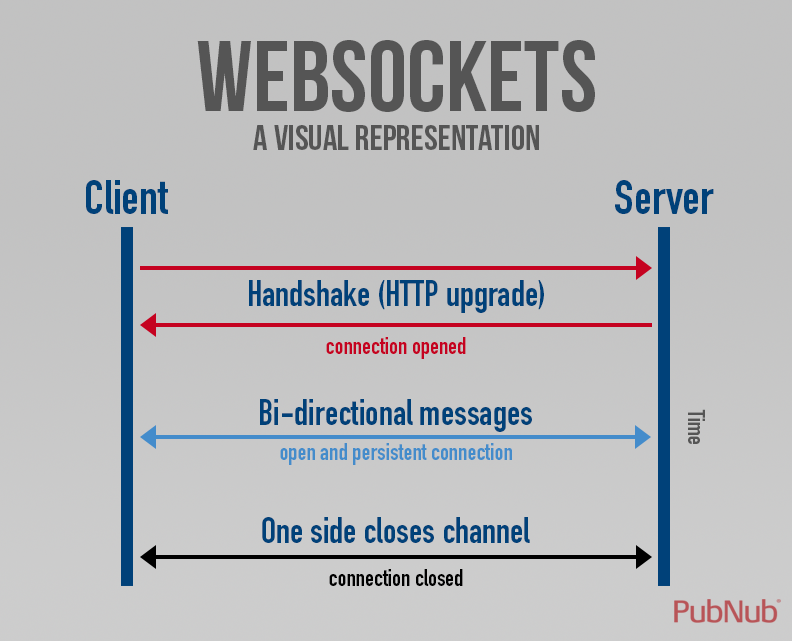
закон Амдала, ограниченность возможностей, speedup – кратность прироста скорости вычисления, parallel portion – степень распараллеливания алгоритма (не все можно распараллелить), number of processors – количество процессоров.

1. Протокол WebSockets, основные свойства, процедура установки соединения. WebSockets API.

протокол **полнодуплексной(дуплексной)** связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером через постоянное соединение.







Спецификация [WebSocket](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) определяет API для установки соединения между веб-браузером и сервером, основанного на «сокете». Проще говоря, это — постоянное соединение между клиентом и сервером, пользуясь которыми клиент и сервер могут отправлять данные друг другу в любое время.

Клиент устанавливает соединение, выполняя процесс так называемого рукопожатия WebSocket. Этот процесс начинается с того, что клиент отправляет серверу обычный HTTP-запрос. В этот запрос включается заголовок Upgrade, который сообщает серверу о том, что клиент желает установить WebSocket-соединение.

*// Создаём новое WebSocket-соединение.* var socket = new WebSocket('ws:*//websocket.example.com');*

URL, применяемый для WebSocket-соединения, использует схему ws. Кроме того, имеется схема wss для организации защищённых WebSocket-соединений, что является эквивалентом HTTPS.

Если сервер поддерживает протокол WebSocket, он согласится перейти на него и сообщит об этом в заголовке ответа Upgrade.

Теперь, после завершения фазы рукопожатия, исходное HTTP-соединение заменяется на WebSocket-соединение, которое использует то же самое базовое TCP/IP-соединение. В этот момент и клиент и сервер могут приступать к отправке данных.

В любой момент после процедуры рукопожатия, либо клиент, либо сервер, может решить отправить другой стороне ping-сообщение. Получая такое сообщение, получатель должен отправить, как можно скорее, pong-сообщение. Это и есть heartbeat-сообщения. Их можно использовать для того, чтобы проверить, подключён ли ещё клиент к серверу.

When the **server** receives the handshake request, it should send back a special response that indicates that the protocol will be changing from HTTP to WebSocket

## [Pings and Pongs: The Heartbeat of WebSockets](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers#Pings_and_Pongs_The_Heartbeat_of_WebSockets)

At any point after the handshake, either the client or the server can choose to send a ping to the other party. When the ping is received, the recipient must send back a pong as soon as possible. You can use this to make sure that the client is still connected, for example.

To close a connection either the client or server can send a control frame with data containing a specified control sequence to begin the closing handshake

<https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers>

WebSockets API –

[Socket.IO](http://socket.io/) – для людей

Ws – ~~для блядей~~ для норм челиков

### Simple server

const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

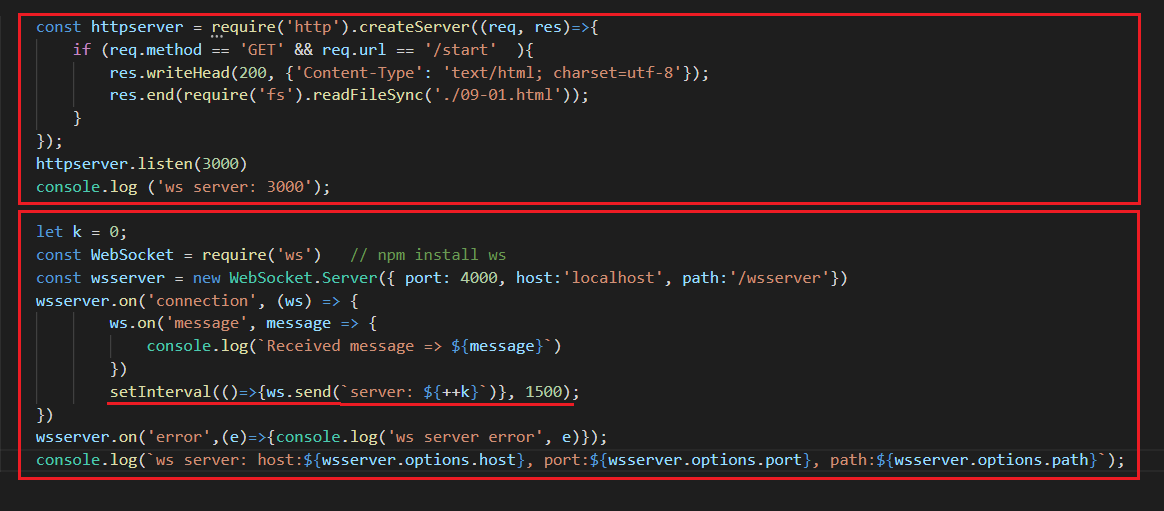
wss.on('connection', function connection(ws) {

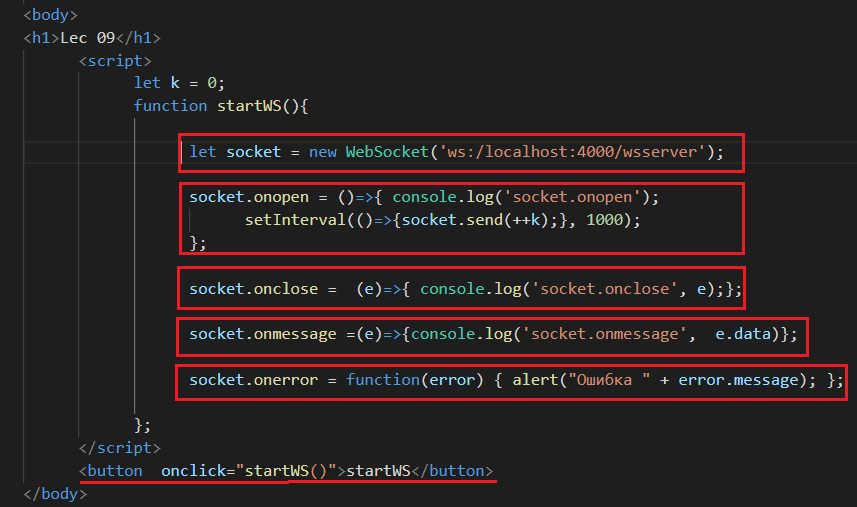
ws.on('message', function incoming(message) {

console.log('received: %s', message);

});

ws.send('something');

});****

****

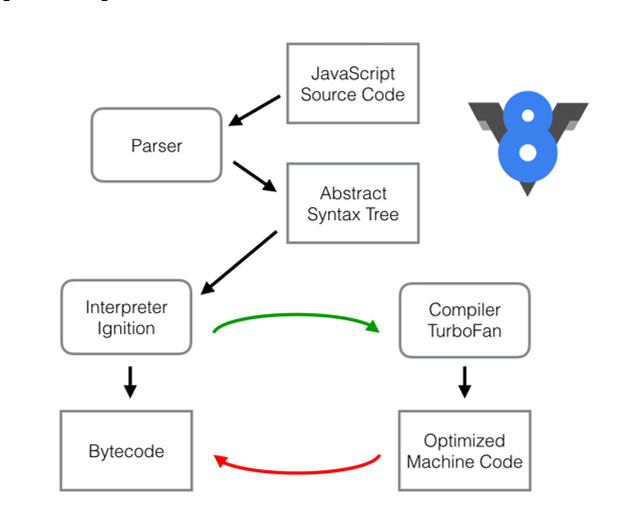
1. Платформа Node.js, версии, назначение, основные свойства, структура, принципы работы, основные встроенные модули и их назначение, применение внешних модулей (пакетов). Web-приложение «Hello World». Пример.

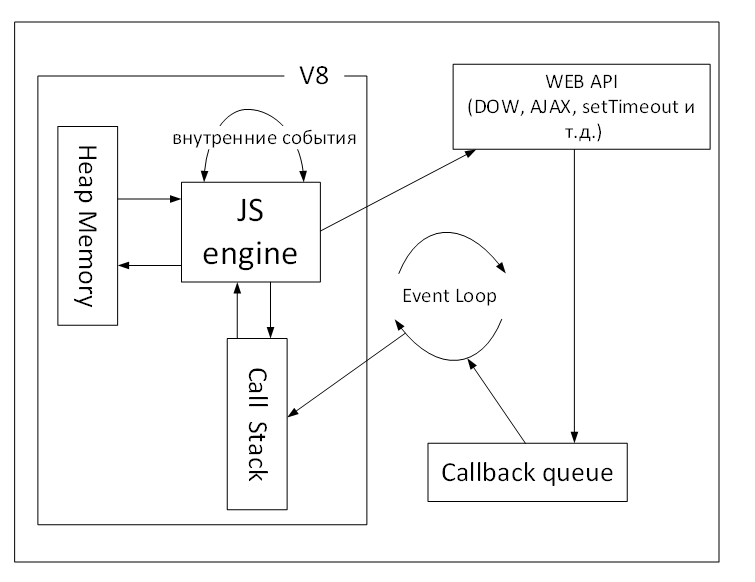
Платформа Node.js - программная платформа для разработки серверных web-приложений на языке JS/V8.

1. основные свойства:

* основан на **Chrome V8**;
* **среда (контейнер) исполнения** приложений на JavaScript;
* поддерживает механизм **асинхронности**;
* ориентирован на **события**;
* **однопоточный** (код приложения исполняется только в одном потоке, один стек вызовов); обычно в серверах для каждого соединения создается свой поток, в Node.js все соединения обрабатываются в одном JS-потоке;
* **не блокирует** выполнение кода при вводе/выводе (в файловой системе до 4х одновременно);
* в состав Node.js входят инструменты: **npm** – пакетный менеджер; **gyp** - Python-генератор проектов; **gtest** – Google фреймворк для тестирования С++ приложений;
* использует библиотеки: **V8** – библиотека V8 Engine, **libuv** – библиотека для абстрагирования неблокирующих операций ввода/вывода (представляет собой обертку над epoll, kqueue, IOCP); **llhttp** – легковесный парсер http-сообщений (написан на C и не выполняет никаких системных вызовов); **c-ares** -библиотека для работы с DNS; **OpenSSL** – библиотека для криптографии; **zlib** – сжатие и распаковка.
* первая версия: **2009 г**.;
* стабильные версии: с **2015 г., Node.js 4.0.0;**
* основная сфера применения: **разработка web-серверов**;

версионирование: **две ветки** 12.x.x – версии длительной поддержки (LST, Long Term Support), 14.x.x – нестабильные версии, включающие последние разработки (Current).

****

****

[assert](https://www.w3schools.com/nodejs/ref_assert.asp) - Provides a set of assertion tests

[buffer](https://www.w3schools.com/nodejs/ref_buffer.asp) - To handle binary data

[fs](https://www.w3schools.com/nodejs/ref_fs.asp) - To handle the file system

[http](https://www.w3schools.com/nodejs/ref_http.asp) - To make Node.js act as an HTTP server

[querystring](https://www.w3schools.com/nodejs/ref_querystring.asp) - To handle URL query strings

[url](https://www.w3schools.com/nodejs/ref_url.asp) - To parse URL strings

[util](https://www.w3schools.com/nodejs/ref_util.asp) - To access utility functions

const http = require("http");

1. Глобальные объекты Node.js (global, process) и их применение. Системные (стандартные потоки) Node.js (stdin, stdout, stderr) и их применение. Модуль console: функции log, error, dir, time, timeEnd, trace. Примеры.

**global: *хранит var-данные на уровне модуля***.

1. **process:** информация о среде выполнения

Node.js предоставляет специальный объект global, который предоставляет доступ к глобальным, то есть доступным из каждого модуля приложения, переменным и функциям.

Объект Process является экземпляром EventEmitter и запускает следующие события:

Exit — Запускается при выходе из процесса. В этот момент невозможно предотвратить выход из цикла событий, после того как все прослушиватели выхода закончатся, процесс завершится.

beforeExit — Это событие запускается, когда node очищает цикл событий и на данный момент не существует других событий, которые должны быть запланированы. Как правило, когда больше не существует запланированных задач, осуществляется выход из node, но прослушиватель для «beforeExit» может выполнять асинхронные вызовы и продолжать работу node.

uncaughtException — Запускается, когда в цикле событий снова и снова возникает исключение. Если для этого исключения добавлен прослушиватель, действие по умолчанию (которое должно вывести результаты текущей операции в стеке и завершить процесс) выполняться не будет.

Процесс предоставляет много важных свойств, для лучшего контроля системного взаимодействия.

Stdout — Записываемый поток в stdout.

Stderr — Записываемый поток в stderr.

Stdin — Записываемый поток для stdin.

argv — Массив, содержащий аргументы командной строки. Первый элемент — «node», второй элемент — имя файла JavaScript.Следующие элементы — любые дополнительные аргументы командной строки.

Env — Объект, содержащий пользовательскую среду.

Log - Выводит в stdout данные с новой строки.

Error - Выводит в stderr данные с новой строки

Dir - Отображает список свойств указанного JavaScript объекта.

Time - Запускает таймер, который используется для вычисления длительности операции. timeEnd - станавливает таймер, запущенный предыдущей функцией. Выводит результат в stdout:

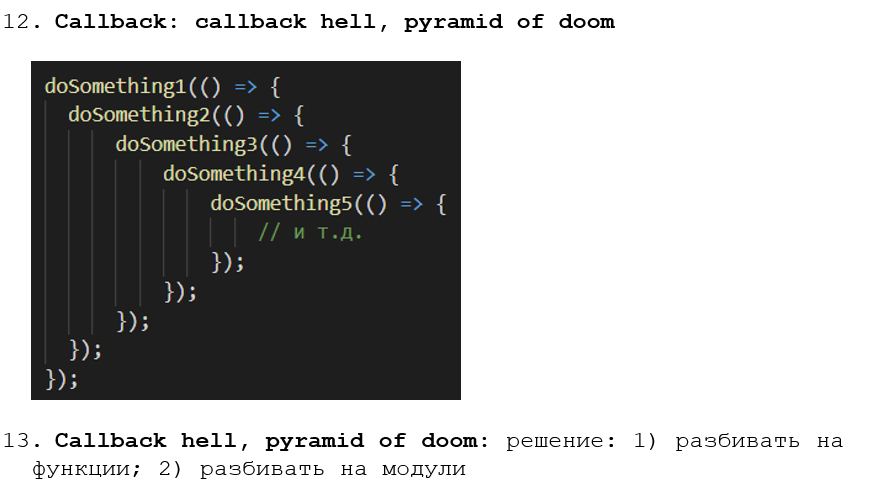
Trace- Выводит в stderr строку 'Trace :' c форматированным сообщением из util.format() и отслеживает его текущую позицию в коде.

1. Асинхронное программирование. Функция обратного вызова. Проблема "Callback hell" и способы решения. Примеры.

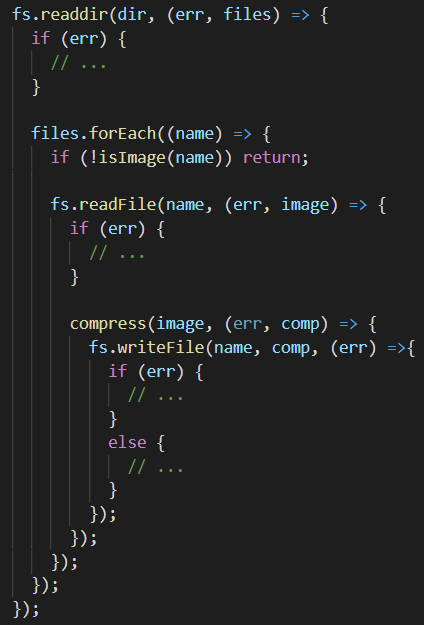
**Асинхронное программирование:** выполнение процесса в неблокирующем режиме системного вызова, что ***позволяет потоку программы продолжить обработку***.

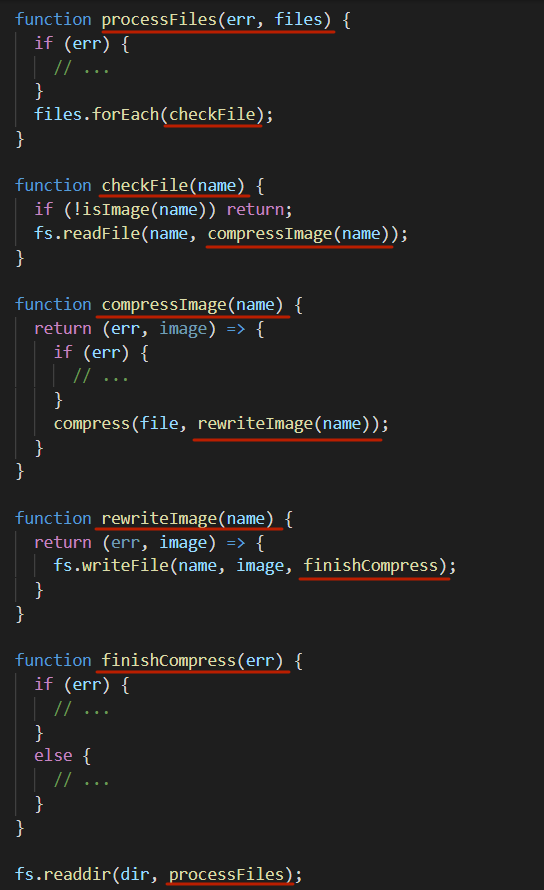
**Понятие асинхронности**: операция называется асинхронной, если ее выполнение осуществляется в 2 фазы: 1) заявка на исполнение; 2) получение результата; при этом участвуют два механизма: A-механизм, формирующий заявку и потом получающий результат; B-механизм, получающий заявку от A, исполняющий операцию и отправляющий результат A; продолжительность исполнения операции B-механизмом, как правило, непредсказуемо; в то время пока B-механизм исполняет операцию, А-**механизм выполняет** собственную работу.

callback-функция (функция обратного вызова) — функция, которая передается в качестве параметра другой функции и которая **будет вызвана** асинхронно обработчиком событий **после завершения задачи**



1. **Callback hell, pyramid of doom:** разбитие на функции.





1. **Callback hell, pyramid of doom:** разбитие на модули.



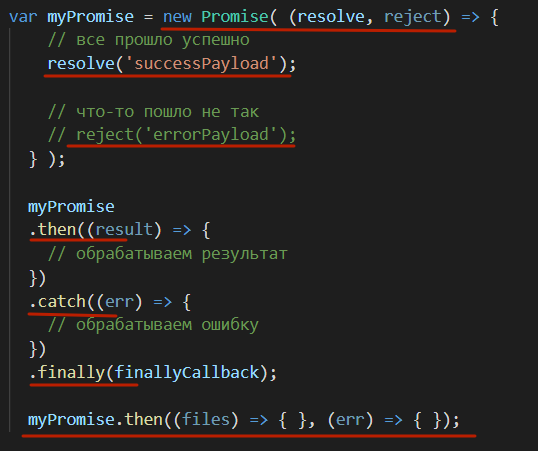
1. Асинхронное программирование. Механизм Promises. Механизм async/await. Примеры.

**Асинхронное программирование:** выполнение процесса в неблокирующем режиме системного вызова, что ***позволяет потоку программы продолжить обработку***.

1. **Понятие асинхронности**: операция называется асинхронной, если ее выполнение осуществляется в 2 фазы: 1) заявка на исполнение; 2) получение результата; при этом участвуют два механизма: A-механизм, формирующий заявку и потом получающий результат; B-механизм, получающий заявку от A, исполняющий операцию и отправляющий результат A; продолжительность исполнения операции B-механизмом, как правило, непредсказуемо; в то время пока B-механизм исполняет операцию, А-**механизм выполняет** собственную работу.

**Promise (обещание):** объект, используемый для выполнения отложенных и асинхронных вычислений. Представляет собой операцию, которая еще не завершена, но ожидается в будущем

**Cвойствo state:** **pending** (ожидание), **fulfilled** (выполнено) при вызове resolve, **rejected** (отклонено) при вызове reject. **Свойство result**: вначале **undefined**, далее изменяется на **value** при вызове resolve(value) или на **error** при вызове reject(error).

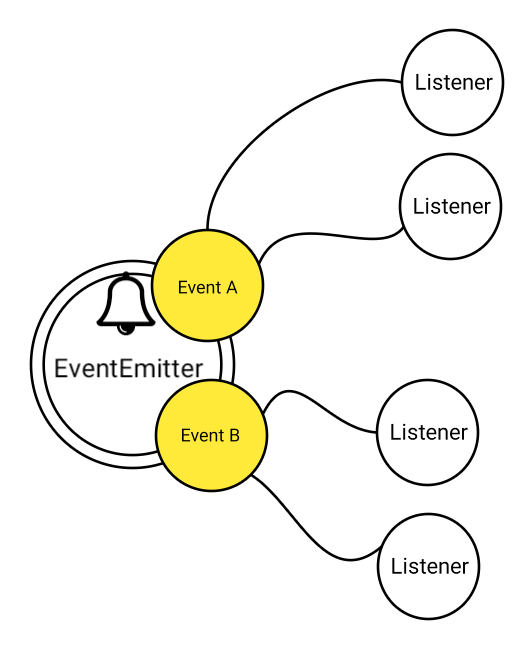


**Async/await:** синтаксис для обработки нескольких промисов в режиме синхронного кода.

**async** - перед объявлением функции, возвращает промис; **await** - блокирует код до тех пор, пока промис не будет разрешен или отклонен.

1. Класс EventEmitter, назначение, применение. Пример.

**EventEmitter:** JS-класс, предоставляющий функциональность для асинхронной обработки событий в **Node.js.** Событие в программном объекте – это процесс перехода объекта из одного состояние в другое. При этом, об этом переходе могут быть извещены другие объекты. У события есть **издатель** (или генератор) события и могут быть **подписчики** (или обработчики) события.



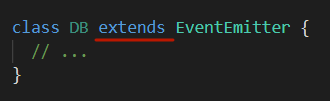
**EventEmitter:** применяется в базовых объектах **Node.js.**

необходимовключения двух модулей: ***events*** и ***util***.

**EventEmitter:** как правило, применяется в качестве базового для пользовательского объекта. Производный от **EventEmitter** объект может быть создан с помощью функции **inherits** модуля **utils**.

****

**EventEmitter:** для наследования можно использовать ключевое слово **extends** (ES6).



**EventEmitter:** производный от **EventEmitter** объект приобретает функциональность, позволяющую генерировать и прослушивать события.

**EventEmitter:** для генерации событий предназначена функция **emit,** адля прослушивания функция **on.**

1. Функции setTimeout, setInterval, nextTick, ref, unref, назначение, применение. Примеры.

**Event Loop: макрозадачи** - выполняются по одной за один проход цикла; **микрозадачи** - на каждом проходе цикл выполняет все накопившееся.

**Макрозадачи:** setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering

**Микрозадачи:** process.nextTick, Object.observe, Promises

**Микро-задачи:** специальная функция **queueMicrotask (func**) - ставит func в очередь на выполнение в очереди микрозадач.

Вы могли заметить, что process.nextTick() не отображался на диаграмме, даже если он является частью асинхронного API. Это связано с тем, что process.nextTick() технически не является частью цикла событий. Вместо этого nextTickQueue будет обрабатываться после завершения текущей операции, независимо от текущей фазы цикла событий.

**Таймер:** механизм, позволяющий генерировать событие или выполнить некоторое действие, через заданный промежуток времени.

**setTimeout()**, **setInterval()**; реализованы библиотекой **libuv.**

**setTimeout():** выполняется только один раз через некоторый промежуток времени.

**setInterval():** выполняется регулярно через некоторый промежуток времени.

**clearTimeout():** останавливает таймер, созданный с помощью setTimeout(). Параметр – ID таймера, который необходимо отменить.

**clearInterval():**останавливает таймер, созданный посредством setInterval(). Параметр – ID таймера, который необходимо отменить.

**NODEJS:** Node.jsработает до тех пор, пока есть события, требующие обработки;если выполнить для таймера **unref,** тособытия**,** генерируемые таймером не будут учитываться при завершении работы Node.js**, ref** –противоположная операция.

метод unref() есть не только у таймеров, есть еще у серверов (server.unref()), сетевых сокетов (socket.unref()) и др.

NextTick () обрабатывается после каждой фазы цикла событий, а setImmediate () обрабатывается только на этапе обработчика проверки цикла событий. Названия обеих функций сбивают с толку. Процесс. NextTick () запускается немедленно на той же фазе, а setImmediate () запускается на следующей итерации или 'тике' цикла событий

1. Модули и пакеты Node.js, функция require, кэширование модуля, область видимости в пакете, экспорт объектов, функций, конструкторов. Применение require для работы с json-файлами. Параметризируемый модуль. Пример.

модуль – фрагмент кода, специальным образом оформленный и размещенный, может использоваться приложением, является фундаментальной единицей структурирования кода Node.js-приложений.

модуль – текстовый файл, содержащий код на языке JS

модуль используемый несколькими приложениями называют **пакетом**.

**CommonJS:** группа, которая проектирует, прототипирует и стандартизирует различные JavaScript API

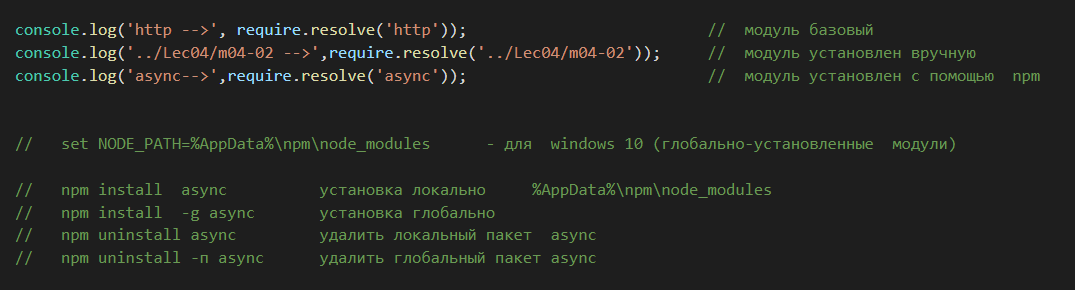
реализованные требования **CommonJS**

* поддержка **require** для импорта модуля;
* имя модуля – строка, может включать символы идентификации путей;
* модуль должен явно экспортировать всю свою функциональность, поддержка объекта ***export***;
* переменные внутри модуля не видимы за его пределами.



**require**

* **сихронно** загружает модуль;
* кэширует модуль;
* удалить из кэша можно с помощью **delete require.cache[…]**;
* если модуль удален, то для его использования нужен новый **require**.



Для локального пакета поиск осуществляется в **node\_modules** по восходящему принципу. После поиска среди локальных пакетов, осуществляется поиск среди глобальных пакетов.

**NODEJS: require:** если в качестве имени указана папка, то дополнительная информация в файле **package.json**



**Var дает видимость только внутри пакеты**

**Global и process глобально во всем приложении exports позваляет использовать эти переменные/функции при использовании пакета**

**Интересная статья про json и required - https://goenning.net/2016/04/14/stop-reading-json-files-with-require/**

1. Пакетный менеджер NPM, глобальное хранилище, просмотр установленных пакетов, скачивание пакетов, назначение файла package.json, локальные хранилища пакетов, удаление пакетов, публикация пакета. Примеры.

**JavaScript Package Manager,** устанавливается вместе с **Node.js,** скачивание /публикация пакетов; инструмент командной строки; глобальное хранилище <https://registry.npmjs.org/>

**NPM:** https://www.npmjs.com/

**NPM:** пакет – один или несколько js-файлов и файл-манифест **package.json**

просмотр установленных пакетов:

**npm list –g --depth=0**

**ls, la, ll –**

скачивание пакетов: install, i

add is just an alias for install.

**глобально установить добавить -g**

назначение файла package.json:

**package.json**: файл конфигурации приложения Node.js. Любая директория, содержащая данный файл, интерпретируется как Node.js-пакет.

**package.json**: содержит метаданные проекта (название, версия, описание проекта, …), список зависимостей вашего пакета, которые будут установлены при вызове команды **npm install**, скрипты, вызывающие другие команды консоли.

локальные хранилища пакетов:

node\_modules – папка в которой сохраняются npm

удаление пакетов:

npm uninstall <name> удаляет модуль из node\_modules, но не package.json

npm uninstall <name> --save также удаляет его из dependencies в package.json

npm uninstall <name> --save-dev также удаляет его из devDependencies в package.json

npm -g uninstall <name> --save также удаляет его глобально

публикация пакета:

Чтобы опубликовать пакет, вам потребуется собрать все исходные коды и файл package.json в одной директории. В package.json должны быть указаны название, версия и зависимости пакета. Например:

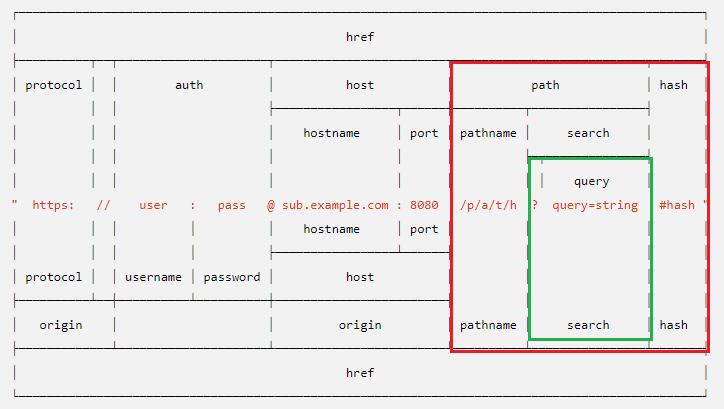
{ "name": "canvas-project", "version": "0.1.0", "devDependencies": { "canvas-chart": "~1.3.0" } }

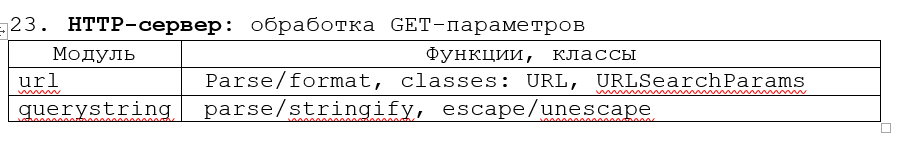
Посмотрев на этот код, мы можем сказать, что пакет «canvas-project» зависит от пакета «canvas-chart». Опубликовать пакет можно с помощью комадны npm publish.

1. Разработка простейшего HTTP-сервера в Node.js. Извлечение данных из HTTP-запроса, формирование данных HTTP-ответа. Пример. Тестирование с помощью POSTMAN.

**HTTP-сервер:** серверная часть web-приложения.



****

**JSON:** JavaScript Object Notation, текстовый формат передачи данных, автор: Дуглас Крокфорд.

**'101'**: 'Switching Protocols',

**'200'**: 'OK',

**'201'**: 'Created',

**'204'**: 'No Content',

**'301'**: 'Moved Permanently',

**'302'**: 'Moved Temporarily',

**'304'**: 'Not Modified',

**'307'**: 'Temporary Redirect',

**'400'**: 'Bad Request',

**'401'**: 'Unauthorized',

**'403'**: 'Forbidden',

**'404'**: 'Not Found',

**'405'**: 'Method Not Allowed',

**'409'**: 'Conflict',

**'500'**: 'Internal Server Error',

**Медиа тип**(так же известный как **Multipurpose Internet Mail Extensions**или **MIME тип)** является стандартом, который описывает природу и формат документа, файла или набора байтов. О

1. Разработка простейшего HTTP-сервера в Node.js. Извлечение данных из HTTP-запроса, формирование данных HTTP-ответа. Пример. Тестирование с помощью браузера AJAX (XMLHTTPRequest/Fetch).
2. Разработка HTTP-сервера в Node.js. Обработка GET, POST, PUT и DELETE-запросов. Генерация ответа с кодом 405. Пример. Тестирование с помощью POSTMAN.
3. Разработка HTTP-сервера в Node.js. Обработка URI HTPP-запроса, маршрутизация запросов, генерация ответа с кодом 404. Пример. Тестирование с помощью POSTMAN.
4. Разработка HTTP-сервера в Node.js. Обработка запросов к статическим ресурсам: html, css, js, png, msword. Пример. Тестирование с помощью браузера.
5. Разработка HTTP-сервера в Node.js. Обработка query-параметров GET-запроса. Пример. Тестирование с помощью браузера.
6. Разработка HTTP-сервера в Node.js. Обработка path-параметров GET-запроса. Пример. Тестирование с помощью браузера.
7. Разработка HTTP-сервера в Node.js. Обработка параметров POST-запроса. Пример. Тестирование с помощью браузера (<form>) и POSTMAN.
8. Разработка HTTP-сервера в Node.js. Обработка json-сообщения в POST-запросе. Пример. Тестирование с помощью POSTMAN.
9. Разработка HTTP-сервера в Node.js. Обработка xml-сообщения в POST-запросе. Пример. Тестирование с помощью POSTMAN.
10. Разработка HTTP-сервера в Node.js. Пересылка файла в POST-запросе (upload). Пример. Тестирование с помощью браузера.
11. Разработка HTTP-сервера в Node.js. Пересылка файла в ответе (download). Пример. Тестирование с помощью браузера.
12. Разработка HTTP-клиента в Node.js. Оправка GET запроса с query-параметрами. Пример. Тестирование с помощью с Node.js-сервера.
13. Разработка HTTP-клиента в Node.js. Оправка POST-запроса с параметрами в теле. Пример. Тестирование с помощью с Node.js-сервера.
14. Разработка HTTP-клиента в Node.js. Оправка POST-запроса с json-сообщением. Пример. Тестирование с помощью с Node.js-сервера.
15. Разработка HTTP-клиента в Node.js. Обработка json-ответа. Пример. Тестирование с помощью с Node.js-сервера.
16. Разработка HTTP-клиента в Node.js. Обработка xml-ответа. Пример. Тестирование с помощью с Node.js-сервера.
17. Разработка HTTP-клиента в Node.js. Пересылка файла на сервер в POST-запросе (upload). Пример. Тестирование с помощью с Node.js-сервера.
18. Разработка HTTP-клиента в Node.js. Обработка ответа с файлом (download). Пример. Тестирование с помощью с Node.js-сервера.
19. Разработка Websockets-приложения: Node.js-сервер, браузер-клиент. Пример.
20. Разработка широковещательного Websockets-приложения: Node.js-сервер, Node.js-клиент. Пример.
21. Разработка Websockets-приложения: Node.js-сервер с применением потока, Node.js-клиент. Пример.
22. Разработка Websockets-приложения: ping/pong-сообщения, Node.js-сервер, Node.js-клиент. Пример.
23. Разработка Websockets-приложения: обработка json-сообщений, Node.js-сервер, Node.js-клиент. Пример.
24. Разработка Websockets-приложения: отправка клиентом файла (upload), Node.js-сервер, Node.js-клиент. Пример.
25. Разработка Websockets-приложения: отправка сервером файла (download), Node.js-сервер, Node.js-клиент. Пример.
26. Разработка RPC-Websockets-сервера. Пример. Тестирование: Node.js-клиент.
27. Разработка RPC-Websockets-сервера: обработка уведомлений. Пример. Тестирование: Node.js-клиент.
28. Работа с файловой системой в Node.js: создание, копирование, проверка существования файла, запись, запись в конец, чтение, синхронные асинхронные операции. Пример.
29. Работа с файловой системой в Node.js: создание, удаление, переименование, запись, запись в конец, чтение, синхронные асинхронные операции. Пример.
30. Работа с файловой системой в Node.js: создание, слежение за файлом, запись, запись в конец, чтение, синхронные асинхронные операции. Пример.
31. Работа с файловой системой в Node.js: запись в файл потока октетов, чтение из файла потока октетов. Пример.

<https://www.youtube.com/watch?v=1rbmO71wwyU&feature=emb_logo&ab_channel=JavaScript.ru>

<https://www.youtube.com/watch?v=_j0LoOXnOF4&list=PLDyvV36pndZFWfEQpNixIHVvp191Hb3Gg&index=26&ab_channel=JavaScript.ru>

**Buffer:** объект, предназначенный для работы с потоком октетов; применяется в функциях readFile, writeFile; 1GB (32-bit), 2GB (64-bit)

**Октет –** 8 бит

1. Работа с файловой системой в Node.js: запись в файл массива 32-битовых целочисленных данных, чтение из файла массива 32-битовых целочисленных данных. Пример.
2. Применение потокового чтение (Readable) и записи (Writable) файлов в Node.js. Пример.
3. Применение функции pipe для обработки данных (файла) запроса и записи в файл файловой системы. Пример.
4. Применение функции pipe для обработки данных (файла) файловой системы и записи в http-ответ. Пример.
5. Разработка клиент-серверного TCP-приложения: обмен текстовыми сообщениями. Пример.
6. Разработка клиент-серверного TCP-приложения: пересылка массива целочисленных данных. Пример.
7. Разработка клиент-серверного TCP-приложения: пересылка файла от клиента серверу. Пример.
8. Разработка клиент-серверного TCP-приложения: пересылка файла от сервера клиенту. Пример.
9. Разработка клиент-серверного TCP-приложения прослушивающего два порта, обмен текстовыми сообщениями. Пример.
10. Разработка клиент-серверного UDP-приложения: обмен текстовыми сообщениями. Пример.
11. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического SELECT-запроса.

1. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического INSERT-запроса. Пример.
2. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического UPDATE-запроса. Пример.
3. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического DELETE-запроса. Пример.
4. Разработка приложения, выполняющего запрос к SQL-базе данных: вызов удаленной процедуры. Пример.

1. Разработка приложения, выполняющего graphql-запрос к SQL-базе данных: query-запрос. Пример.
2. Разработка приложения, выполняющего graphql-запрос к SQL-базе данных: mutation-запрос. Пример.

асс. каф. ИСиТ М.В. Дубовик

**В билете 3 вопроса: 1 и 2 вопросы из списка (1-60), 3-й вопрос – демонстрация одной из лабораторных работ (1-16).**

**На экзамене студент обязан предоставить все выполненные (1-16) лабораторные работы. Студент, который не предоставит полный список выполненных лабораторных работ автоматически получает неудовлетворительную оценку.**

**В качестве дополнительного вопроса может быть затронута тема, рассмотренная на лекции, но не вошедшая в список экзаменационных вопросов (MongoDB, Mongoose, REST, SemVer и др.).**