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

**HTTP** – прот.перед.гипертекста – прот.прикл.ур-ня перед.д-х (изнач.– в виде гипертекст.док-ов).

Основа – технол.«К-С», т.е. предполаг.сущ-ние К, кот. инициир. соед. и посыл.запрос, и С, кот. ожид. соед. для получ.запроса, вып.необх.д-вия и возвращ. обратно сообщ.с рез-том.

**Спецификация HTTP**

* Название: HyperText Transfer Protocol
* Уровень (по модели OSI): прикладной
* Семейство: TCP/IP
* Порт/ID: 80/TCP
* Назначение протокола: доступ к гипертексту, ныне стал универсальным
* Спецификация: RFC 1945, RFC 2616

|  |  |
| --- | --- |
| Достоинства | Недостатки |
| \*простота – леко созд.К прилож  \*расширяемость  \*распространенность | \*большой р-р сообщ  \*нет поддержки распределнности |

**!Осн. свойства HTTP:**

- версии http/1.1 (текстовый), http/2 (бинарный – сейчас к нему переход)

- 2 вида сообщений: req, res => асинхр (неравноцен К и С, разный формат сообщ)

- 2 типа абонентов: К –req–> <–res– С

- 1 req : 1 res и наоборот (иначе ошибка)

- TCP-порты: **80** (для серверов, не поддерж. шифр) и **443** (поддерж шифр; HTTPS)

- для адресации исп. URI (часть без http://имя\_пк)

- поддерж. W3C, описан в неск RFC (2616)

|  |  |
| --- | --- |
| **!Структура Request:**  - метод (метка get/post)  - URI  - версия протокола (http/1.1)  - заголовки (имя+знач)  - параметры (имя+знач)  - расширение (тело) | **!Структура Response:**  - версия протокола  - код сост (1хх, 2хх, 3хх, 4хх, 5хх)  - пояснение к коду сост  - заголовки (имя+знач)  - расширение |

Заголовки и тело м.отсутств., но старт.строка – обяз, т.к.указ тип запроса/отв

Старт.строка запроса: метод, uri, версия

Старт.строка ответа: версия, код состояния, пояснение

**Request-методы:**

\*OPTIONS – исп.для опред-ния возм-стей веб-С или пар-ров соед для рес-са.

\*GET – узнать содержимое ресурса; м. начать какой-то процесс и передать инфу

\*HEAD – анал.get. Отличие – отсутствует тело в ответе С.

\*POST – для передачи пользовательских д-х задан.ресурсу.

\*PUT – загрузки содержимого на указ.URI (не было – созд; есть – измен.)

\*DELETE – удаляет указ.рес-с

\*TRACE – вовзращ.запрос так, что К м.увидеть какую инфу промеж.С доб/измен в запросе

connect, extension-method

|  |  |
| --- | --- |
| **Заголовки:**  \* general (д.ключ.в люб.сообщ К/С)  \* request (т. в запросах)  \* response (исп т. в ответах)  \* entity (опис содержимое тела – ContentType) | **Код сост. Response:**  \* 1хх: информ. сообщение  \* 2хх: успешный ответ  \* 3хх: переадресация  \* 4хх: ошибка клиента (404)  \* 5хх: ошибка сервера |

HTTPS — протокол безоп.пер.данных, поддерживает технологию шифрования TLS/SSL.

HTTP передаёт д-е в открытом виде. Злоумышленники могут “вклиниться” в передачу — изменить или перехватить данные. В HTTPS для перед. данных созд.защищённый канал.

HTTPS не является отдельным протоколом. Это обычный HTTP, работающий через шифрованные транспортные механизмы SSL и TLS. Он обеспечивает защиту от атак, основанных на прослушивании сетевого соединения

**URI** (Uniform Resourse ID) – унифицир. id ресурса (док, изобр, файл, служба, email)

**URL** (Uniform Resourse Location) – унифиц. локатор ресурса

**URN** (Uniform Resourse Name) – унифиц. имя ресурса

**URI** – имя ресурса, не содержащее месторасп. и способа доступа к ресурсу (можем опр. месторасп. ресурсов в глобальной БД)

**STD** – там прописан URI, STD0 – список всех STD

***КС прил*** – прил, сост из 2 компонент: К и С

К и С взаимод. в соотв. с заданными правилами (*протоколами*)

***Клиент*** – часть прил, явл. инициатором соед

Между К и С дб устан *соединение*

***Web-прил*** – прил с КС архитектурой, где К и С взаимод. по протоколу HTTP (прикл.ур)

**сеть Интернет – 4 компонента:**

1) сеть на основе ст.пр. TCP/IP

2) набор служб (протокол + сервер, все № портов до 1024) : DNS, FTP, TelNet, POP3

3) стандарты (там прописаны протоколы) : RFC, STD

4) организации, упр-щие этой сетью : IAB, IANA, ICANN, ISOC

*backend* – серверная часть, *frontend* – клиентская часть

***Узел интернет*** – у-во, имеющее IP + подключение к сети Интернет

к. узел хар-ся своей ***прогр.-аппарат. платформой*** (аппаратура + ОС)



***Кроссплатф. прил*** – прил, способное работать на >1 прогр.-аппар. платформе

мб достигнута:

- на уровне компилятора (C, C++) – перекомпил в др. прил

- на ур. среды исполн (фреймворка) - не надо перекомпил

Технологии для разраб. кросс web-серверов:

\* PHP / Apache, LAMP (Linux Apache MySwl php)

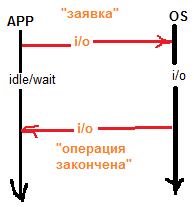
\* Java / JVM / Application Server

\* C# / ASP.NET CORE

\* Python / Django

\* Ruby on Rails

\* JS / Node.js

**Web-сервер:**

принимает запросы, обраб их и отпр ответы К

чтение http-запроса i/o

парсинг http-запроса cpu

запросы к БД i/o

запросы к др. серверам i/o

вычисления cpu

форматиров. ответа cpu

отправка http-ответа i/o

В основном, web-сервер делает в(ы)вод, мало использует процессор

Под всеми к-дами в(ы)вода скрыт системный вызов функции ядра ОС!

Пока ждем ответ ядра ОС, это время надо рационально исп.

*“проблема блокирующего в(ы)вода”* – если ничего в это время не делаем – **синхр. операция** (процессор простаивает)

**Решение:**

1) многопоточность (огран кол-во потоков, т.к. каждый требует доп. памяти) – в Apache

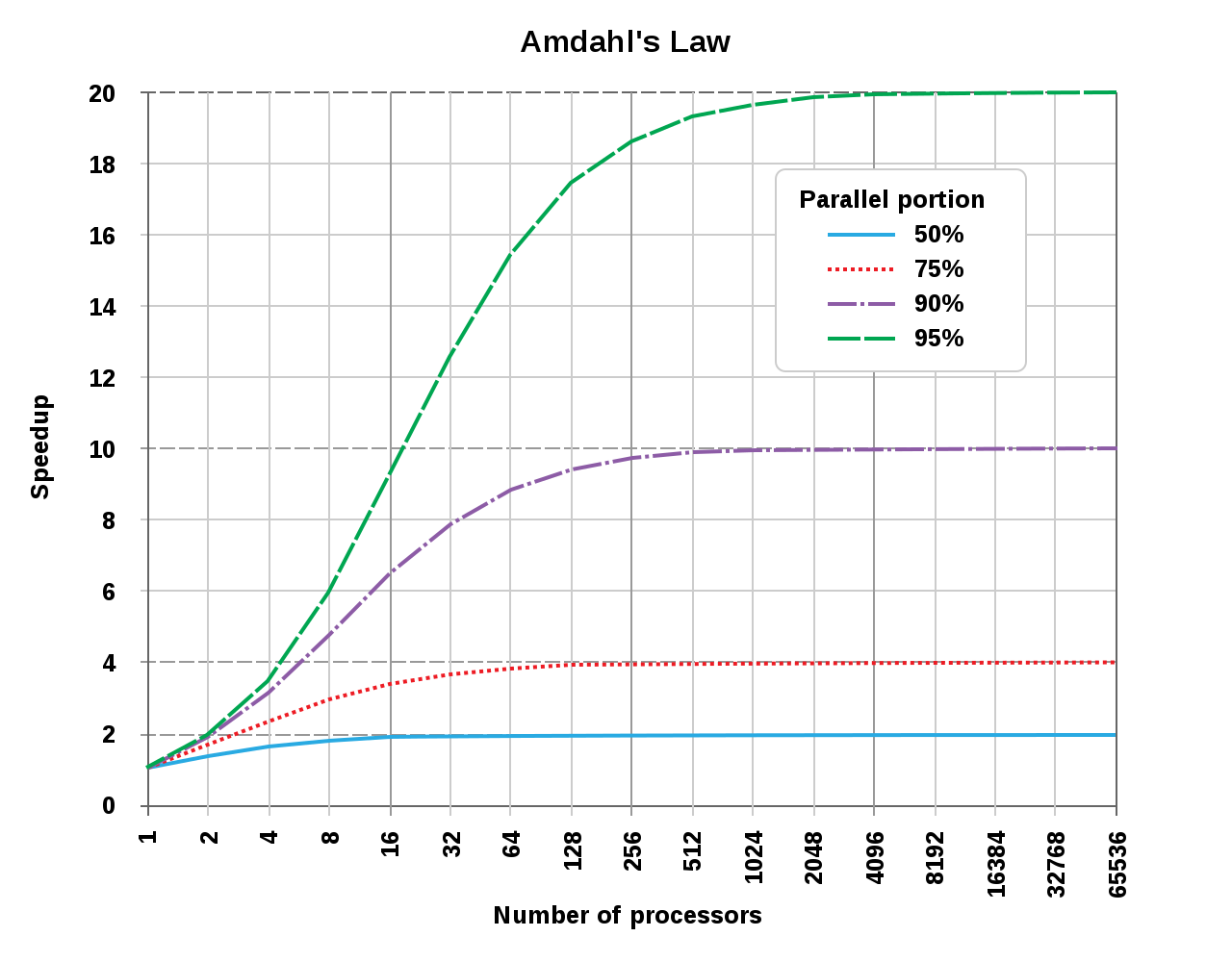
2) асинхр прогр – паттерн Reactor – в Nginx

**Закон Адмала:** ограниченность возможностей

*speedup* – кратность прироста скорости вычислений

*parallel portion* – степень распаралл. алгоритма (50% - разбить задачу на 2 части)

В какой-то момент будет достигнута точка, после к-рой несмотря на увеличение числа процессоров, кратность прироста ∆v скорости высилений ост. неизменной



мб неск К, или неск web-серверов у одного К, или 1:1

**Асинхронность:**

если в парам. ф-ции задаем ф-цию обратного вызова, в синхр: не задаем

Если поток вып-я проги приостан. при вызове ф-ции в(ы)вода, то эта ф-я синхронная, иначе – асинхр (не остан. поток), рез. вып-я м. получит в callback-е

Понятия (а)синхр ввода-вывода сущ. на уровне ОС

Причем синхр – частный случай асинхр, по сути все ф. async, но есть разные абстракции, кот. дают нам синхронность

Хотим вывести: заполн буфер, говорим ОС: “хочу вывести”, ОС из буфера д-е выбир. и пишет (чтение анал.) => Надо знать API ОС

Операция наз. ***асинхронной***, если вып в 2 фазы:

1) заявка на исполн

2) получение рез-та

+ участвует 2 механизма:

A – формир заявку ( , вып собс работу , ) получ рез

B – получ заявку от А, исп операцию, отпр рез А

**Ас.ф.** – ф., после вызова к-й приложение продолжает работать, т.к. сразу вып. возврат, а ее рез. известен позже

2. Протокол WebSockets, основные свойства, процедура установки соед. WebSockets API.

**WebSocket** – \*новый протокол

\*формат передачи д-х

\*надстройка над TCP для обмена сообщ в режиме реал.времени

(подробнее в устан.соед.)

\*стандарт RFC 6455

\*дуплексный (2 канала связи)

\*пакет require(‘ws’)

\*широковещ.сообщ

\*каналы – *потоки*

\*механизм ping/pong

\*JSON, XML и др.форматы передачи сообщ.

\*RPC

**Каналы связи:**

1. **Дуплексный –** одновременно и прием, и передача (TCP, WebSocket)
2. **Полудуплексный –** одноврем.только прием или только передача (HTTP)

Чаще всего WS исп.для работы с браузерным К. (**npm install ws)**

**WebSocketAPI** (API для созд., упр-ния вебсокет-подключ к С + отправки и получ д-х )

**Атрибуты:**

\*onopen – когда соед.устан.

\*onclose – соед.закрыто

\*onmessage – д-е получены

\*onerror – ошибка

**\***readystate – тек.сост.подключ

**\***url – url созд.к-ром

**Константы:**

**ReadyState:** (константы сост.готовности)

\*connecting – соед. еще не открыто

\*open – открыто и готово к обмену д-ми

\*closing – соед.в проц.закртыия

\*closed – соед.закрыто или не м.открыться

**Методы:**

\*close – закрывает WS-подключение или заканчив.попытку подключ.

(если уже закрыто – ничего не делает)

\*send – передаем д-е через WS-соед

**Браузерные объекты** – объекты, кот.сущ.только в пределах браузера.

**Объекты:** \*navigator – дает инфу о браузере и ОС

\*location – получить тек.URL и перен.на новый адрес

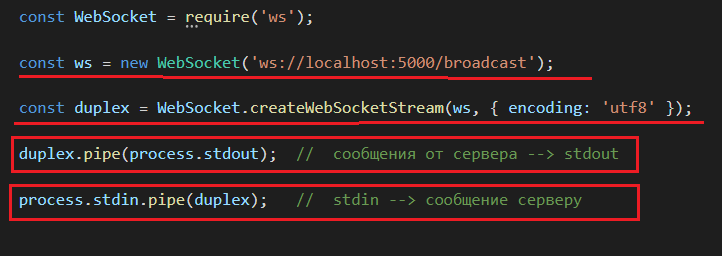
**Широковещ.сообщ** – отсыл.всем узлам в сети.

****

**Взаимод.с потоками:**

Канал соед.м.представить в виде **дуплексн.потока** => м.и чит., и пис.д-е одноврем.

**Поток –** абстракция, кри.позвол представить д-е в виде потока байт.

****

**Механизм ping/pong** – механизм для проверки соед.

\*С посыл.сообщ ping для проверки соед., К.должен ответить сообщ pong

\*делаем к С ping , C возвращ.pong

**RPC** (remote procedure call) – удаленный вызов процедур

- это технология, кот.позвол.прогам вызыв.процедуры и ф-ции в др.адресном пр-ве (как правило на удал.ПК). Т.е.на С генер.методы, а К их вызыв.

\*npm install rpc-websockets

**Rpc-сервер** \*в кач-ве трансорта исп. WS

\*регестрир.удал процедуру

(имя, ф-ция, .public()/.protected)

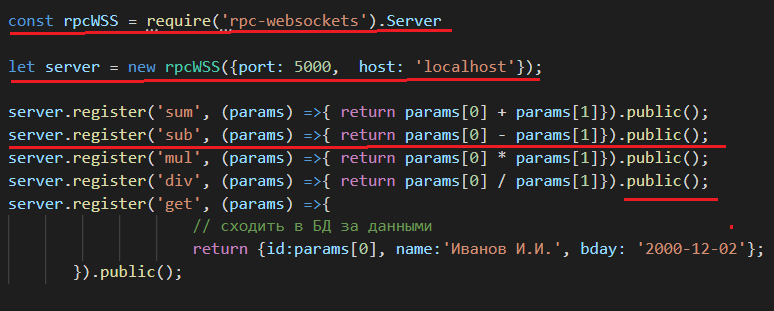
\*Если public – клиент не завершается

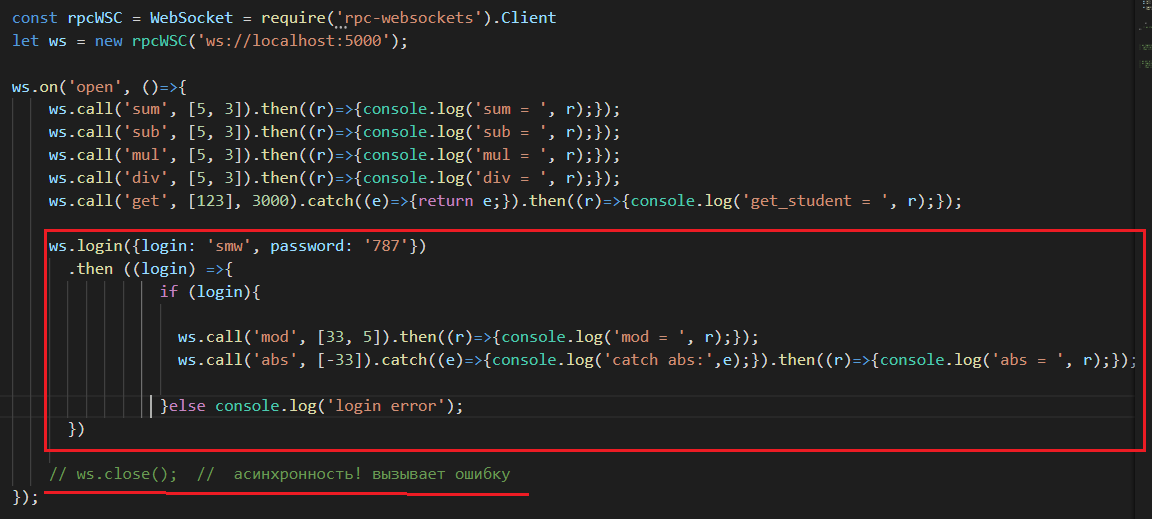
**\***Если protected – требует аутентификации (логин/пароль)

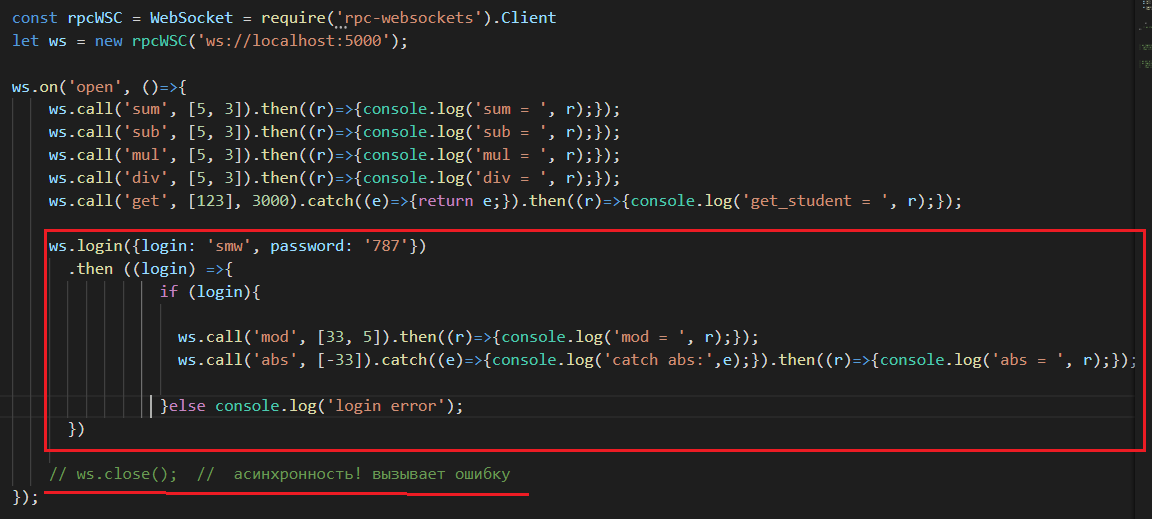


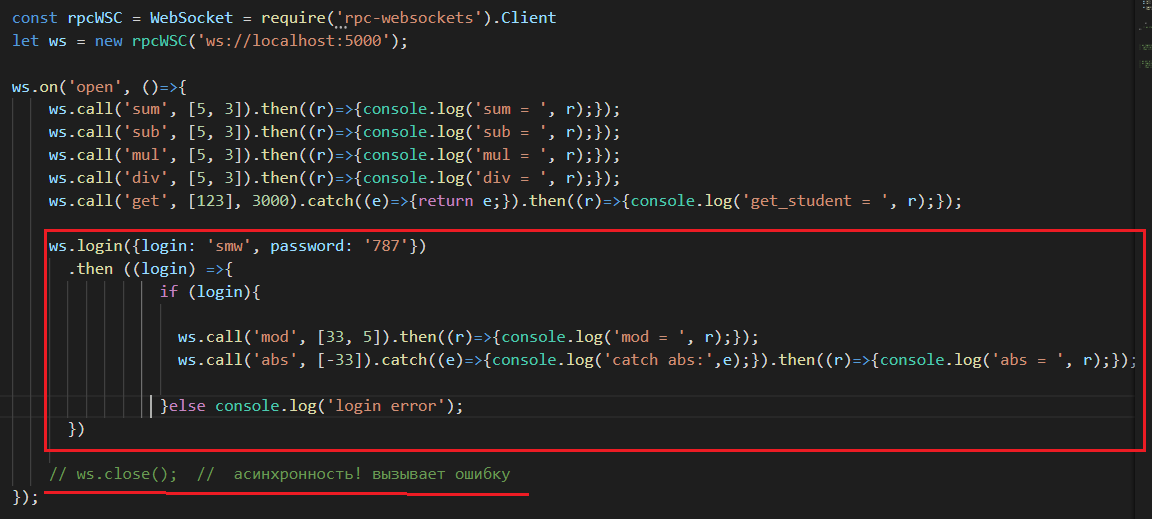






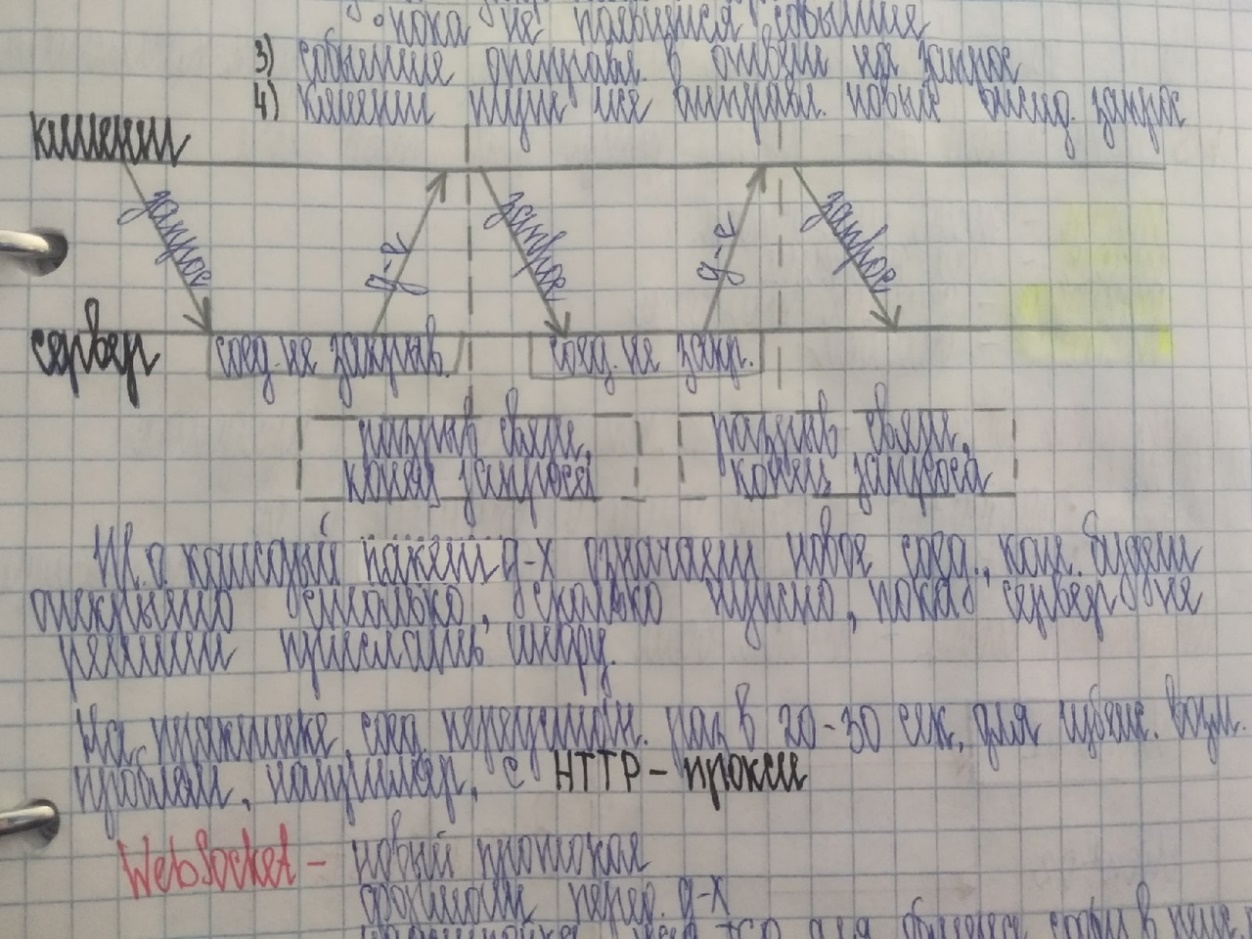






**Схема:**

1. отправляем запрос на С
2. соед.не закрывается сервером (пока не появится событие)
3. событие отправляется в ответ на запрос
4. К тут же отправл.новый ожид.запрос



Т.о. каждый пакет д-х означает новое соед., кот.будет открыто столько, сколько нужно, пока С не решит прислать инфу. На практике соед.переустан.1 раз в 20-30 сек, для избеж.возм. проблем, например, с HTTP-прокси.

**Процесс установки соед:**

**Upgrade** – К просит С переключится на новый протокол (WebSocket)

**Процесс «рукопожатия»** - процесс установки соед по новому протоколу:

К посыл обычный HTTP-запрос на С с заголовком **upgrade** кот сообщ.С, что К хочет устан.WS-соед.

3. 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.

**Node.js –** прогр.платф.для разраб.серв.web-прилож.

**Осн.св-ва:**

\*основан на chrome v8;

\*среда исполнения прилож.на JS

\*ориент.на мех.асинхронности

\*ориент.на события

\*однопоточный

(код прилож.исп.только в 1 потоке, 1 стек вызовов);

(обычно в С для к.соед созд.свой поток, в Node.js – все соед.обраб.в 1 потоке)

\*не блок.вып-ние кода при вводе/выводе

\*в состав вх.инструменты: 1) npm – пакетный менеджер

2) gyp – Puthon-генератор проекта

3) gtest – Google фреймфорк для тестир.С++ прилож

\* движок – библ.v8 (библ. V8 engine)

\* исп.библ: 1) libuv – библ.для абстрагир.неблок.оп-ций ввода/вывода;

2) http-parser – парсер http-сообщений (написан на С и не вып.сист.вызов)

3) c-ares – библ.для работы с DNS

4) zlib – сжатие и распаковка

\* **осн.сфера примен:** разраб.web-серверов; **назнач –** созд.масштабир.сетевых прилож

\* nodejs.org

\* разработчик – Райан Дал

\* первая версия – 2009

\* стаб.версии: с 2015г., Node.js 4.0.0;

\* четн.версии – версии длит.поддержки (LST)

\*нечетн. – нестаб.версии, включ.последние разраб.

|  |  |
| --- | --- |
|  | **Компоненты:**  \*V8 – интерпретатор JS в процессе исп.кода в HeapMemory (выдел.всем прилож)  \*Heap Memory - динам.пам.  \*JS Engine – код исполн.  \*CallStack – для хранения контекста и вып.ф-ций (вызов ф-ций друг в друге)  т.к.стек 1 => однопоточн.  \*Callback queue – связ.с ас; сохр.контекст отлож. процедуры  ! сам Node.JS – многопот., а движок – однопоточ.  ! node.exe – интерпретатор |

**Как запустить?**

\*открыть ком.строку node.js

\*зайти в папку с js-файлом

\* node имя файла

**Node.js** исп.модульную систему т.е. вся встр.фунционал.разбита на отд.пакеты или модули.

**Модуль –** блок кода, кот.м.исп.повторно в других модулях.

**Модуль** – JS-код, кот.нах.в отд.файле и экспортир.отд.объекты.

**Модуль –** фрагмент кода, кот.цеп.образом оформлен и размещен, м.исп.прилож., явл.фундамент.единицей стр-ры кода JS-прилож.

**Модуль –** текст.файл, кот.содержит текст на JS

**Модули:**

\*вх.в ядро (через require) – встроенные модули (больше 50)

\*разраб.сами

\*установили с пом.npm (в node-modules) – внешние модули

**Встр.модуль** – те модули, кот.представ.мин.функцион.возм-сти.

**\***http – чтобы node.js действовал как http-сервер + тож.самое https

\*fs – для работы с файловой системой

\*console

\*events – для обраб.событий

\*os – инфа об ОС

\*net – создавать С и К

\*и др.

**Метод require –** загрузить модуль.

var http = require (‘http’);

Когда С стартует сразу выводится console.log, когда приходит запрос – вызыв ф-ция из пар-ра ф-ции createServer (создать сервер (пар-ры – асих.ф-ция)). В ф-ции из пар-ров мы д.обраб. request и сформир.response.

http.createServer (function (request, response)) {}

Любому прилож.даются 3 стандарт.потока - in, out, err

**Response.end** – записать инфу из пар-ра в тело ответа и больше ничего не будет.

**Response.writeHead** – установить статус ответа и заголовки (пара ключ-значение)

Response.writeHead (200, {‘Content-Type’: ‘text/html; charset = utf-8’});

**Request.on –** обраб.события (ловит соб.и указ.что вып-ть)

*request.on (событие, ф-ция для обраб)*

request.on (‘data’, str => { b+= str; colsole.log (‘data’, b);})

request.on (‘end’, ф-ция для обраб) – когда пришла последн.порция д-х

**Модуль FS –** модуль для раб.с файл.сист.

**Файл.сист –** та компонента ОС, кот.предназн.для созд.абстракции при работе с дисками.

\*readFileSync – счит.файл из пар-ра и помещ.его в пер-ную (синх.)

\*ключ.слово Sync – пока не прочит файл и д-е не помещ.в html => С ничего не буд.дел

****

\*readFile – счит.файл ас.

Если есть callback-ф-ция, первый ее пар-р дб связан с обраб.ошибок

const fname = ‘./Belka.jpg’;

jpg = fs.readFile( fname, (err, data) = {});

**Наши модули:**

1) создать файл my.js

содержимое: console.log(‘my module’);

2) подключим наш модуль в app.js

const greeting = require(./my);

!в отл.от встр.модулей для подключения своих надо передать в ф-цию относительный путь с именем файла (расшир.не обяз.)

**Внешние модули**(подробно в 7)

\*для установки исп.пакет

\*установить модуль м.локально или глобально

\*для лок.пакета поиск осущ.в node-modules по восход.принципу (сначала папка ‘node-modules’, затем папка выше…)

\*после поиска среди лок.пакетов, осущ.поиск среди глоб.пакета

**Пакет –** модуль, кот.исп.неск.прилож.

**CommonJS –** группа, кот. проектирует, прототипирует и стандартиз.разл.JSAPI;

**Требования commonsJS:**

\*поддержка require для импорта модуля

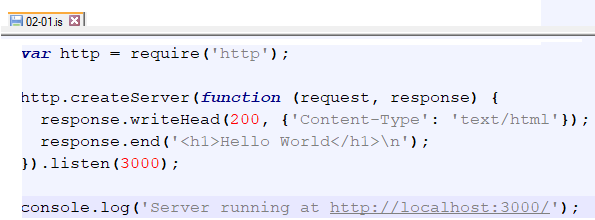
\*имя модуля – строка, м.включ.символы идент.

\*модуль д.явно экспорт.всю свою функцтонал.; поддержка export

\*пер-ные внутри модуля не видны за его пределами

Скачать пакет = npm install <package-name>

**Пример HelloWorld**

****

//запустить через cmd node.js

//браузер/postman - localhost:3000

4. 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.

***Модуль*** – js core в отд. файле, кот. экспортирует нек. свои компоненты (сохр. свою область видимости)

**ГЛОБАЛЬНЫЕ ОБЪЕКТЫ:**

\* предост. переменные и ф-и, доступные в любом месте программы, во всех модулях

\* по умолч, это те, что встроены в язык или среду исполн.

\* всегда доступен приложениям Node.js без необходимости вызова require()

***global*** – хранит var-данные на ур. модуля (общий объект для модуля)

пр: 4

***process*** – глобальный объект, предост. инфу и контролир. тек. процесс Node.js.

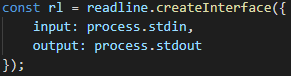
хранит инфу о среде вып.

пр: process.exit, disconnect, abort, argv, nextTick, stdin, stdout…

**СТАНДАРТНЫЕ ПОТОКИ:**

***stdin, stdout, stderr*** – любой проге, запуск. в ОС, дается 3 потока: ввода, вывода, ошибок







**МОДУЛЬ CONSOLE:**

Предост простую консоль для компиляции, кот. экспортирует 2 компонента:

1) класс *console* с методами .log(), .error(), .warn() для записи в любой стрим

2) глоб. экземпляр *console*, для записи в stdout, stderr

**log**(‘..’) – выводит на консоль ‘..’

**error**(new Error(‘..’)) – выводит [Error: ..], в поток stderr

**dir** – console.dir(document.documentElement) – указ. в логах на DOM узел

**time** … код …**timeEnd** – время в мс

5. Класс EventEmitter, назначение, применение. Пример.

***EventEmmiter*** – js класс, предост. функц-сть для асинхр обработки событий в Node.js

исп. в кач. базового класса для пользовательского объекта

произв. класс приобр. функц-сть, позв. генер и прослуш события

***Соб. в прогр. объекте*** – процесс перехода объекта из одного сост. в другое. При этом, об этом переходе мб извещены др. объекты.

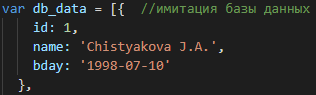
У объекта есть издатель (генератор) и мб подписчики (обработчики)

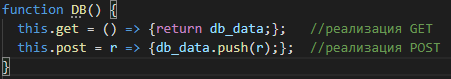
**Пример: (4 л/р)**

созд. объект DB, расп. в отд. модуле, кот. должен уметь генер и *асинхр* обраб события get, post, put, delete

**DB.js:**



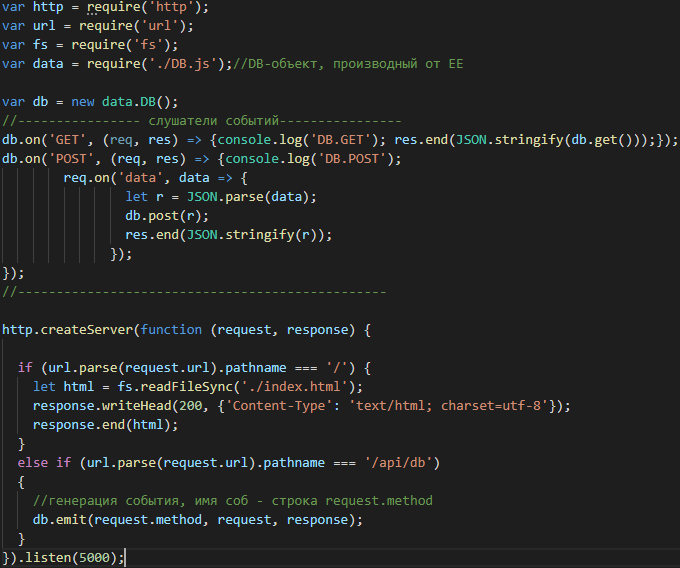




 с пом. ф-и inherits модуля utils созд.

произв. от EventEmitter объект DB

**04.js:**

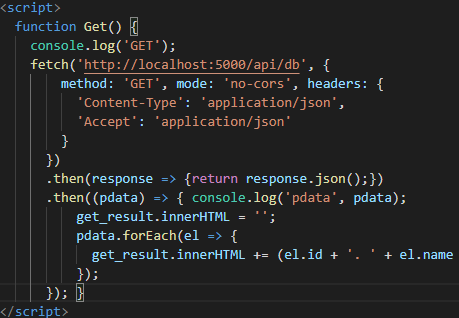
db.on – подписка соб, db.emit – генер. соб

**Клиент (index.html):**



fetch() : аргументы – 1) путь к ресурсу, кот. хотим получить, кот. возвр. promise, содержащий ответ. Чтобы извлечь содержимое тела JSON из ответа, мы исп. json();

2) объект init, для контроля настроек: *no-cors* – разреш только head, get, post методы



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

**SETTIMEOUT() SETINTERVAL():**

***Таймер*** – механизм, позв. генер соб или вып. нек действие через заданый пром. t

NODE.JS: setTimeout(), setInterval(), реализ. библиотекой libuv

***Date.now()*** – ск. милисекунд с 0:00 1 янв 1970

***\* setTimeout*** ( ф.обр.вызова , t мс , аргум для ф.обр.выз )

время отсчитано => ф. обр. вызова ставится таймером в Callback Queue.

***\* setInterval()*** – аналогично, но не единожды, а периодически

***clearTimeout(timer2)*** – откл таймер, ***clearInterval(***..) – откл интервал

они ставят ф-и в начало приоритета в CallBack queue (схема):

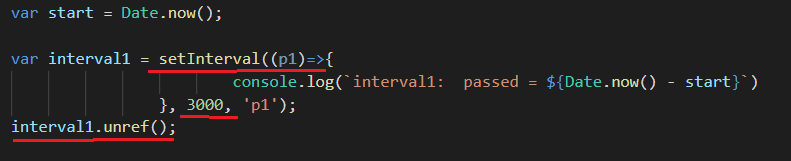
1) время дб точное, замеры не тормоз.

2) надо для всего знать время!

3) клавиатура высоко в приор – всегда можем что-то ввести

**REF() и UNREF():**

Node.js работает до тех пор, пока есть соб, требующие обработки. Если вып. для таймера .***unref()***, то события, генер. таймером, не будут учитываться при заверш работы Node.js; .***ref()*** – противоположная операция. ***unref()*** => “Node.js, ты можешь завершаться и не обращать внимание на обработку события этого таймера.”



асинхр => позв. откладывать (пост. в очередь callback ф-ю) обработку события, пока не дойдет очередь.

Стандартные ф-и node.js делают это авто-, если сами пишем код ф-и, делаем это сами

=> с пом. 2 механизмов:

(иначе блокир. процессор) неск. браузеров -> 1 станет

иллюзия «работаю с сервером в одиночку» => у всех по частям вычисления

**NEXTTICK() И SETIMMEDIATE():**

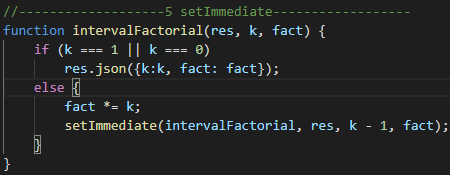
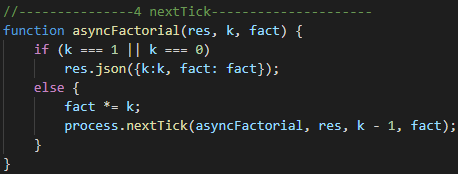
***Event Loop (цикл событий)*** – позв. вып. неблокир. оп-ции в(ы)вода путем выгрузки оп-ций в ядро ОС, когда это возможно. в каждой вкладке браузера свой event loop

**Call Stack** (стек вызовов): по принципу LIFO (последним вошел, первым вышел)

Цикл событий на каждой итерации проверяет, есть ли что-нибудь в стеке вызовов, если да – выполняет это до тех пор, пока стек вызовов не опустеет.

***process.nextTick*** – отклад. выполнение ровно на 1 цикл. Ставит в очередь в самое начало след. цикла event loop, эффект небольшой; ***тик*** – один полный проход цикла событий

***setImmediate*** – ставит в конец очереди callback след. цикла event loop, эффект очевиден



**Порядок выполнения операций в цикле событий:**



между каждой итерацией цикла событий Node.js проверяет, ожидается ли завершение каких-либо асинх i/o-операций или таймеров, и завершает работу, если их нет.

1) Вып. таймеров, порог к-рых истек

2) Обработка событий в очереди:

\* очередь не пуста – синхронно вып. события в Callback Queue

\* очередь опроса пуста:

! снова проверит таймеры

- если запланирован сценарий с пом. setImmedate(), цикл обработки завершит фазу опроса и перейдет к фазе проверки

- нет – event loop ждет новых вызовов в Callback Queue

7. 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.

**CommonJS –** группа, кот. проектирует, прототипирует и стандартиз.разл.JSAPI;

**Требования commonsJS:**

\*поддержка require для импорта модуля

\*имя модуля – строка, м.включ.символы идент.

\*модуль д.явно экспорт.всю свою функцтонал.; поддержка export

\*пер-ные внутри модуля не видны за его пределами

**Модуль –** блок кода, кот.м.исп.повторно в других модулях.

**Модуль** – JS-код, кот.нах.в отд.файле и экспортир.отд.объекты.

**Типы модулей:**

**\***вх.в ядро

**\***разраб.сами

**\***установили с пом.npm

**Пакет –** модуль, кот.исп.неск.прилож.

Для локального пакета поиск.осущ.в node-modules по восход.принцип (выше-выше-выше)

После поиска среди лок.пакетов, осущ.поиск среди глоб.пакетов.

**Метод require –** загрузить модуль.

var http = require (‘http’);

После первой загрузки модули **кэшируются**, а это, означает, что любой вызов require(‘’) – ссылающий на 1 и тот же файл – всегда вернет одинак.объект. (из документации node.js)

\*удалить из кэша м.с помощью delete require;

Delete require(‘./m07-01’);

Delete require.cache(‘./m07-01’);

\*если модуль удален – для исп-ния нужен новый require;

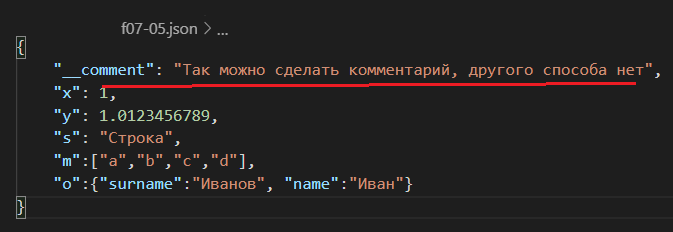
Ф-ция **export** исп.для экспорта ф-ций, объектов или примитивов из файла (модуля)

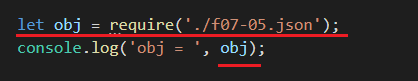
Экспорт 2 видов:

1. Именованный – для экспорта неск.величин
2. Default – для экспорта 1 штуки (ф-ции, класса, объекта)

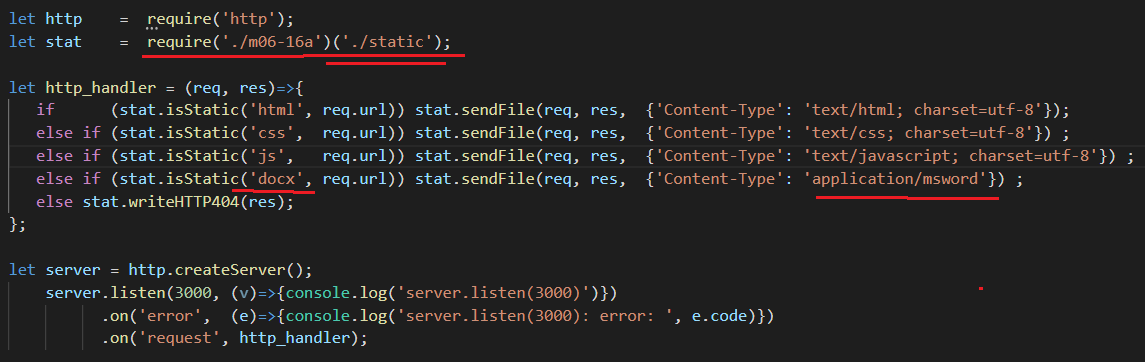
**Именованный** экспорт в модуле:

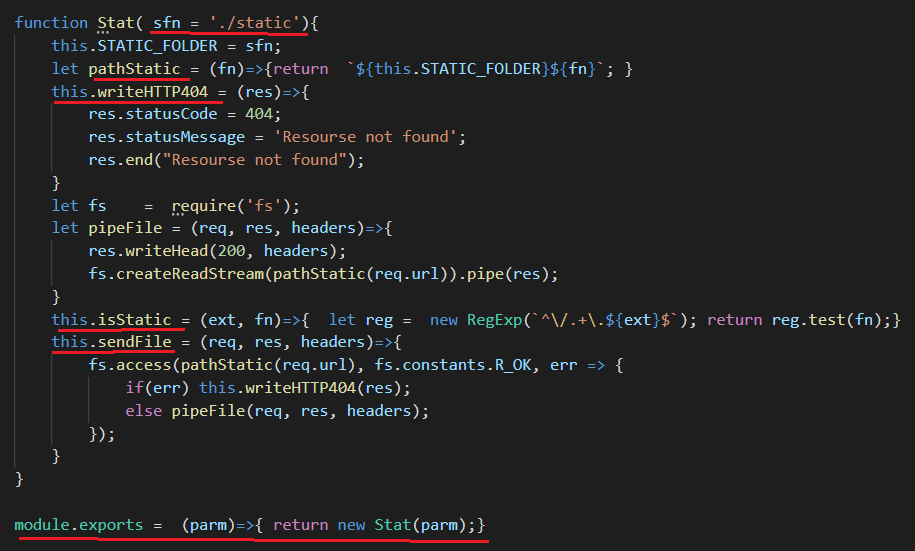
**application/json(rfc 4627)**

****

****

**Параметризированный модуль**

****

****

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

**JS Packet Manager –** устан.вместе с Node.js; инструмент cmd;

предназнач.для скачивания/публикации пакетов;

+масса готовых модулей => не надо изобретать велосипед (как мавен для джавы)

**Пакет -**  1/неск js-файлов, кот.представл.собой каую-то библ/инструмент

**Package.json** – содерж.инфу о прилож.: назв., версия, завис-ти и т.д.;

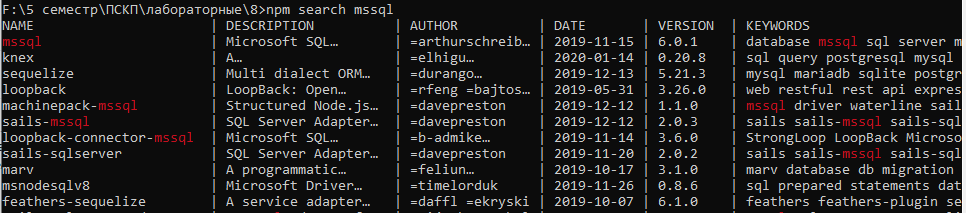
Люб.директория, в кот.есть этот файл интерпретир.как **node.js пакет**, даже если не собираемся публиковать его.

**Обновить npm:**

npm install –g npm;

**Просмотр пакетов:**

npm search *mssql*



**Просмотр инфы о пакете:**

npm view *mssql*



**Скачивание пакетов:**

Скачивание пакетов мб глоб.и лок.

По умолч.все пакеты устан.в лок.каталоге, в кот.мы работаем.

|  |  |
| --- | --- |
| Локальное скачивание:  npm install <package-name>  устан.в node-modules нашей папки | Глобальное скачивание:  npm install <package-name> –g  \*-g означ., что пакет дб устан.глоб. и быть доступ.для всех прилож. |

**Удаление пакетов:**

|  |  |
| --- | --- |
| Лок.установленный пакет | Глоб.установленный пакет |
| npm uninstall <package-name> | npm uninstall <package-name> -g |

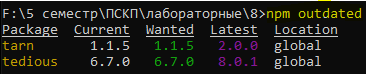
**Обновление пакетов:**

npm update //провер.налич.новых модулей + обнов.найденные

npm update <packagename> //обновить 1 модуль

**Узнать есть ли устаревшие пакеты**

npm outdated



**Просмотр установленных пакетов:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Локально** | | **Глобально** | |
| npm list /ls  + --depth=0/1 | Список лок.устан.пакетов  Упрощенный список | npm list –g  + --depth=0/1 | Список глоб.устан.пакетов  Список со всеми завис-ми  Список в чит.виде |
| npm la/ll | Список всех зав-стей | npm la/ll -g | Список зав-стей для глоб |

(для получ.доп инфы исп config перед ls/ll/la/list)

**Публикация пакета:**

npm login (npm adduser - зарегаться)

//npm init //создать пакет для публикации

npm publish

9.**ДОП ИНФА**

**HTTP – сервер:**

**HTTP-сервер** – С часть веб-прилож.

**HTTP-протокол –** формат передачи д-х по TCP.

Полудупл.канал – К посыл.запрос и должен дождаться ответа по тому же каналу.

createServer – для создания С. (вызыв.при поступл.запроса на С).

**Очередь подключ –** очередь заявок на вып-ние accept(выгребает из очереди соед).

Соб.**connection** – возник.когда К выдал ф-цию connect, она раб.при соед.с С и оно поступ о очередь listen. Т.е.вып-ся когда пришло сообщ., что мы хотим соед.

Server.keepAliveTimeout – макс время бездействия К (12000 мс по умолч)

Если есть timeout => close не работает.

**socketProperties:** localAdress, localPort, remoteAdress, remoteFamily, remotePort, bytesWriten

Длинное сообщ. – не факт что доставится за 1 порцию д-х.

req.on(‘data’, {перед. порция д-х, кот. прибыла с этим соб.})

req.on(‘end’, {все данные получены})

**Стат.рес. –** д-е, кот предст.из себя файлы д-х, кот сод.на С и практич.без измен.пересыл.на К (html-стр, css, js-файл). Обычно задействована файл. с-ма.

**POSIX** (IEEE) – стандарт ОС, набор доков, опис. взаимод-е прикл. программ с ОС (набор сист. вызовов, с пом. кот. обращ. к ядру ОС) – грубо говоря, там *«принципы сокетов»*

За обменом д-х по сети всегда скрываются сокеты, всегда! (это всё те же accept, socket, receive, listen, bind..) все эти ф-ции прописаны в POSIX; JS – замаскированный сервер Шимана

**POST:**

параметры перед. в теле запроса в виде строки ‘имя = значение’&…

imput внутри формы -> put запрос.

извлекать д-е из формы с пом.: let qs = require(‘querystring’); qs.parse(result)

**JSON: (JS Object Notation)**

это форма сериализации js-объекта (в строку). JS-объ <-> JSON (м. превращать туда-обратно)

JSON.stringify(obj1) – преобраз js-объект в строку

JSON.parse(json\_obj) – преобраз в js-объект

**MIME:**

стандарт (в RFC 4727): Multipurpose Internet Mail Extensions

в Content-Type: говорим, какие д-е (какой MIME тип) пересыл в теле ответа

в Accept: говорим, какие данные (неск. mime) хотим получить

**Пересылка файла:** upload – от К к С, download – от С к К

fs.readFileStream – счит. файл

POST: .. <form method=”POST” enctype=”multipart/form-data”>

.. <input name=”file”..>

multipart/form-data – структура тела иная, спец. разделители и т.д.

require(‘multiparty’); //скачать

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**HTTP-клиент -** это клиентская часть веб-прилож. К делает запрос и получ ответ,к от.м.обрабат.

\* require(‘http’);

\* созд. объект http.request(options, <ф-я обраб. ответы сервера>)

***options:*** - параметры для обращ. к серверу

- хост (ip или симв. имя gethostbyname)

- парс URI-запроса

- порт

- метод

Если сервер обращ.к др. серверу, в него надо вкладывать http-клиент.

query = require(‘querystring’) --> м. формир. парам-ры

let parms = query.stringify ( {x:3; t:2})

req.write(.) – запих. в тело при POST-запросе

*Разновидности rpc:*

|  |  |
| --- | --- |
| **subscribe:**  С генер. событие (emit)  К подпис. на соб на стороне С | **уведомл:**  на К: уведомл С, что наступило соб на К  на С: регистр процедуру notify  отличие – нет ответа от С, мы только уведомляем его |

**Функции FS:**fs.open(“file.txt”, “w”,()={}); w-режим

fs.writefile(“..”); запис. д-е в файл / перезапись

fs.appendfile(..); запись в конец файла

fs.readfile(..); откр файл для чтения

fs.unlink(..); удалить файл

fs.rename(..) переимен

fs.copyfile(..,…) копир. файл

fs.exists(..) проверка наличия

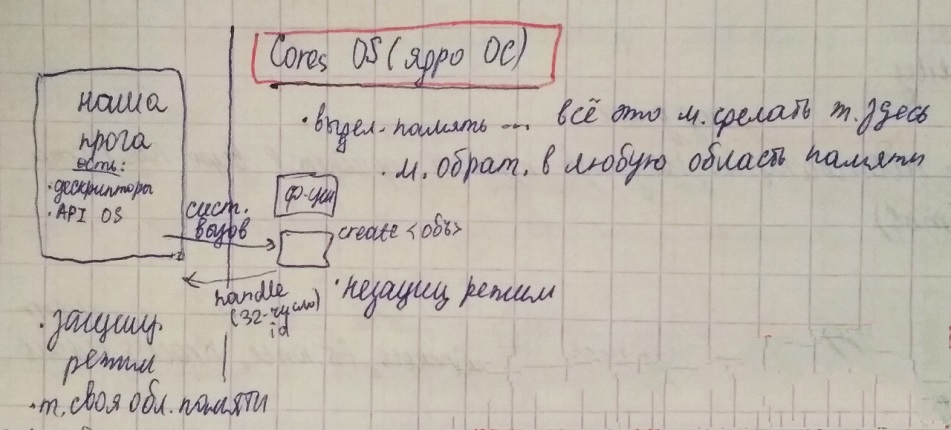
fs.watch(..) следить за ф. (если измен. дата изм, эта callback-ф. вызовется)

*Директорий* – файл, кот. хранит ссылки на др. файлы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Сокет –** объект ОС, кот. в своих св-ва хранит (IP, порт, сем-во проток). Мы можем с ним работать через сист. вызовы, используя *дескриптор* **(**32-бит. число – id сист.объекта ядра ОС)

|  |  |
| --- | --- |
| объекты ядра ОС:  файл, сокет, процесс, память… | действия с объектами:  create, open, close |



**Процесс –** ед.работы ОС – объект ОС, кот.связан с каким-то заданием, кот вып-ся в данной ОС

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**TCP:**

Как устроен TCP-сервер?

\* require(‘net’);

\* минимум 2 сокета (1 - для устан.соед, 1 - для обмена сообщ.в режиме реал.времени)

\*listen – переводит С в режпрослуш. + созд.очередь кот ждет подключ

\*accept – выбир.из очереди очередн.эл-т + созд.сокет для обмена сообщ.с К

Как завалить TCP-сервер?

\*подключить столько К, чтобы ост.получили отказ

Во всех ОС мех.сокетов практически одинак. Видим сетевой в-в – за ним скрыв.сокет. Сокеты пропис.в стандарте POSIX.

**Петля –** (127.0.0.1 или 0.0.0.0) – для того, чтобы раб.на лок ПК,

К отлич от С тем, что К – инициатор соед.

session – объект, чтобы хранить инфу о соединении. Если как в лабе неск. клиентов просят вычисление, сессия позволяет серверу отличить их.

СООТНОШЕНИЕ:

1. net.createServer – скрывается accept, кот выдает sock
2. sock.on(‘data’) – recv
3. sock.write – send

**UDP:**

Как устроен UDP-сервер?

\* require(‘dgram’);

\* на сервере 1 сокет

\*не устан.соед.

\*serv.on (error/message/listerning/close)

\*client.on(message)

**MSSQL**

Npm install mssql

Сервер СУБД – доступ через сокет (как только сервер – сразу сокет).

Для подключ.надо знать:

\*ip /<-- символ.имя //gethostbyname или DNS (в host.ass)

\*порт (sql – 1433, oracle - 1521)

\*имя БД

\*логин, пароль

Динам запрос – знач.в запрос подставл.налету

- нек.код, созд.и сохр в пер-ной, пока не возникн.необх.его вып-ния

-select \* from faculty where fac\_name := name;

Статич.запрос – устан.во время проектир.и не сод.пар-ров и арг-тов

-select \* from faculty where fac\_name = ‘ИТ’;

*Итеративный язык* – парадигма проги, кот.опис проц вычисления в виде инструкций, кот.измен.сост.д-х:

Говорим как вып-ть задачу, опис.алгорит, даем интсрукции

*Декларативн.язык (SQL)* – парадигма проги, кот.опис.специф.решения задачи

Описывается что представл собой проблема и ожид.рез-т

**Группы sql-оп-ций:**

\*ddl –create/drop/alter

\*dml – insert/update/delete/select

\*tcl – commit/rollback, begin tran

\*dcl – grant, revoke, deny

Сходства HTTP и запросы SQL:

\* stateless – отсут. состояние

\* любой запрос http не связан ни с послед., ни с пред; в SQL тоже нельзя связать запросы

\* каждый новый запрос начинает новую жизнь серверу

Sql – доступ осущ.с пом.языка sequel;

Варианты соед.с С:

1. Исп-ние пула соед
2. Открыть С и при запуске им.пост.соед
3. Connect=> запрос к д-м => отключ.соед

*Пул соединений:*

хотим сделать запрос как можно более независимым от остальных (вносим в него измен. – никак не влияет на др. части). Сам себе открываю и закрываю соединение.

Есть min и max кол-во соед. При старте соп. пула, он авто- открывает min соед-й. Когда они все занят, пул откр еще одно… до max. Никогда не закр соед!

*Пул соед* – набор соед. с БД, кот. по мере нагрузки увел. с min->max + очередь

*Прогр. интерфейсы для работы с MSSQL:*

ADO, jdbc, odbc(c++), оледиби

*odbc* – первооснова всех остальных, ост. – это вариации или похожие/одинак

Логику лучше возлагать на SQL-сервер (процедуры, триггеры, ф-и, констрейны…) – более верный подход. Логику двигать подальше от UI, от клиента, прятать глубже.

**NO-SQL БД**

Что лучше sql или no-sql? Зависит от того, насколько д-е сложные. Либо усложнять структуру д-х, либо усложнить алгоритм. Лучше усложнять структуру д-х!

Реляц. БД: модель не завис. от способа использ ~ она не оптимизирована с т.зр. применения

послед. этап – денормализация БД

NO-SQL: такая структура, кот. позв. усложнять структ. д-х, ускорить работу прил.

***MongoDB*** – мб облачгая с-ма (atlas)

\* БД (набор коллекций)

\* коллекции (д-е, кот. им. какие-то структуры, мб разные)

\* объект json

документно-ориент БД

событ.-ориент: созд. промежут. хранилище между осн. БД и веб-сервером, в него запис. события на С («надо доб юзера..д-е…»), но не делаем их. Потом уже event возник. и д-вия в хранилище д-х.

collection.find (как select), .deleteMany({}), insertMany({})… findOneAndUpdate…

**GraphQL**

***GraphQL*** – формальный язык запросов

позв. делать агрегир. запросы (м. в 1 запросе запросить все необх д-е)

*веб-сервис* – веб-прил, интерф не для конеч. юзера, а для др. прилож

*Интерфейсы:*

\* RPC

\* REST – выда. в виде набора URI и инструкций к ним. Каждый ресурс предост. в виде коллекции, но там нет стандарта

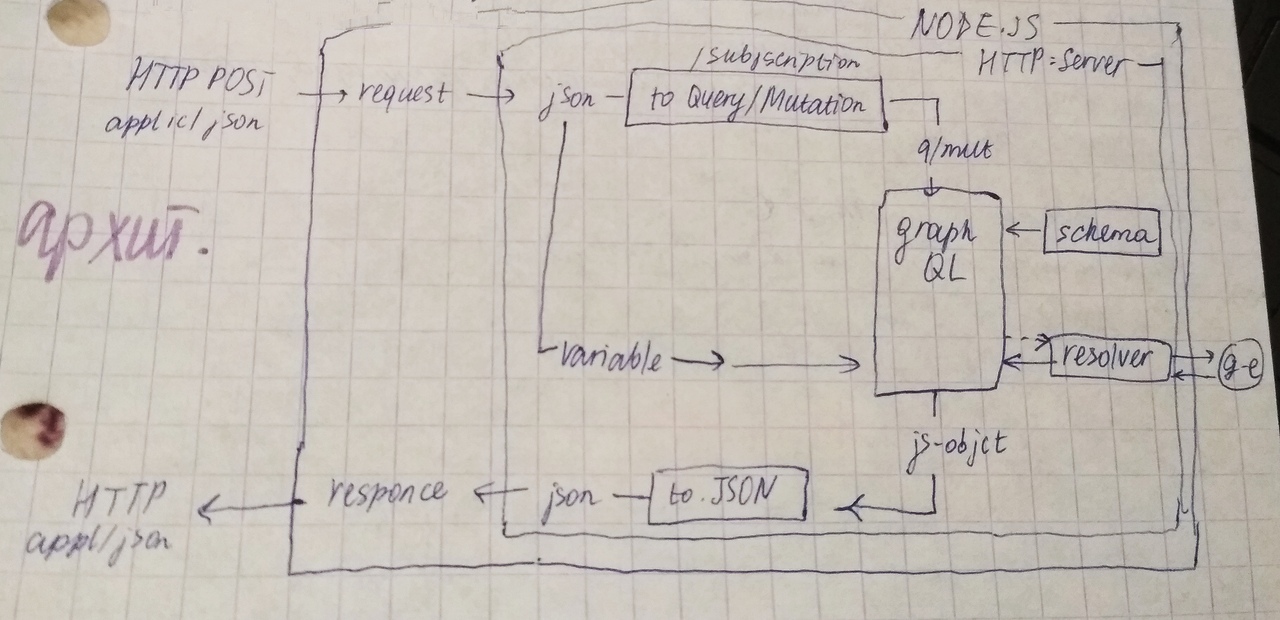
GraphQL – позв. несоклько упорядочить работу с REST-сервисами, пока нет стандартов

**Компоненты GraphQL:**

- схема

- query bootage subscription

- resolver (разрешители)



3 типа заданий, кот. К м дать Су с пом GraphQL:

- Query

- Mutation

- Subscription

**\* Schema** – описание ТД, кот. К м. исп в Q/M/S (запросах)

**\* Resolver** – наш код (сами писали), кот. придает смысл запросам, кот. приходят на GraphQL. обраб. компонент запроса и превра в др. запрос.

имена св-в объекта resolver совпад. с именами запросов, кот. есть в схеме

\* **Mutation** – запросы, кот. приводят к измен д-х (удал, изм, доб). Особый тип, кот. использ нотацию input

Мы даем схему прогеру, он ее получает по GET запросу и по ней делает запросы.

В схеме: Fac! – не мб пустым, [pulpits!] – массив/колл pulpit-ов, сама коллекция мб пустой (нет эл-тов), но если есть эл-ты, они не мб пустыми. [ ]! – непуст. колл, допуск. пустые эл-ты

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

**9+**

10. Разработка простейшего HTTP-сервера в Node.js. Извлечение данных из HTTP-запроса, формирование данных HTTP-ответа. Пример. Тестирование с помощью браузера (XMLHTTPRequest/Fetch).

**10+**

11. Разработка HTTP-сервера в Node.js. Обработка GET, POST, PUT и DELETE-запросов. Генерация ответа с кодом 404. Пример. Тестирование с помощью POSTMAN.

**11+**

12. Разработка HTTP-сервера в Node.js. Обработка URI HTPP-запроса, маршрутизация запросов, генерация ответа с кодом 404. Пример. Тестирование с помощью POSTMAN.

**12+**

13. Разработка HTTP-сервера в Node.js. Обработка запросов к статическим ресурсам: html, css, js, png, msword. Пример. Тестирование с помощью браузера.

**13+**

14. Разработка HTTP-сервера в Node.js. Обработка query-параметров GET-запроса. Пример. Тестирование с помощью браузера.

**14+**

15. Разработка HTTP-сервера в Node.js. Обработка uri-параметров GET-запроса. Пример. Тестирование с помощью браузера.

**15+**

16. Разработка HTTP-сервера в Node.js. Обработка параметров POST-запроса. Пример. Тестирование с помощью браузера (<form>) и POSTMAN.

**16+**

17. Разработка HTTP-сервера в Node.js. Обработка json-сообщения в POST-запросе. Пример. Тестирование с помощью POSTMAN.

**17+**

18. Разработка HTTP-сервера в Node.js. Обработка xml-сообщения в POST-запросе. Пример. Тестирование с помощью POSTMAN.

**18+**

19. Разработка HTTP-сервера в Node.js. Пересылка файла в POST-запросе (upload). Пример. Тестирование с помощью браузера.

**19+**

20. Разработка HTTP-сервера в Node.js. Пересылка файла в ответе (download). Пример. Тестирование с помощью браузера.

**20+**

21. Разработка HTTP-клиента в Node.js. Оправка GET запроса с query-параметрами. Пример. Тестирование с помощью с Node.js-сервера.

**21+**

22. Разработка HTTP-клиента в Node.js. Оправка POST-запроса с параметрами в теле. Пример. Тестирование с помощью с Node.js-сервера.

**22+**

23. Разработка HTTP-клиента в Node.js. Оправка POST-запроса с json-сообщением. Пример. Тестирование с помощью с Node.js-сервера.

**23+**

24. Разработка HTTP-клиента в Node.js. Обработка json-ответа. Пример. Тестирование с помощью с Node.js-сервера.

**24+**

25. Разработка HTTP-клиента в Node.js. Обработка xml-ответа. Пример. Тестирование с помощью с Node.js-сервера.

**25+**

26. Разработка HTTP-клиента в Node.js. Пересылка файла на сервер в POST-запросе (upload). Пример. Тестирование с помощью с Node.js-сервера.

**26+**

27. Разработка HTTP-клиента в Node.js. Обработка ответа с файлом (download). Пример. Тестирование с помощью с Node.js-сервера.

**27+**

28. Разработка Websockets-приложения: Node.js-сервер, браузер-клиент. Пример.

**28+**

29. Разработка широковещательного Websockets-приложения: Node.js-сервер, Node.js-клиент. Пример.

**29+**

30. Разработка Websockets-приложения: Node.js-сервер с применением потока, Node.js-клиент. Пример.

**30+**

31. Разработка Websockets-приложения: ping/pong-сообщения, Node.js-сервер, Node.js-клиент. Пример.

**31+**

32. Разработка Websockets-приложения: обработка json-сообщений, Node.js-сервер, Node.js-клиент. Пример.

**32+**

33. Разработка Websockets-приложения: отправка клиентом файла (upload), Node.js-сервер, Node.js-клиент. Пример.

**34+ (посмотреть 11-01, 11-01а)**

34. Разработка Websockets-приложения: отправка сервером файла (download), Node.js-сервер, Node.js-клиент. Пример.

**33+ (м.сделать как в 34+) (посмотреть 11-02, 11-02а)**

35. Разработка RPC-Websockets-сервера. Пример. Тестирование: Node.js-клиент.

**35+ (м.посмотреть 11-05, 11-05а,б,с )**

36. Разработка RPC-Websockets-сервера: обработка уведомлений. Пример. Тестирование: Node.js-клиент.

**36+(м.посм.11-07,11-07а); либо лекция по WS (20 и 21 пункты)**

37. Работа с файловой системой в Node.js: создание, копирование, проверка существования файла, запись, запись в конец, чтение, синхронные асинхронные операции. Пример.

**37+**

38. Работа с файловой системой в Node.js: создание, удаление, переименование, запись, запись в конец, чтение, синхронные асинхронные операции. Пример.

**38+**

39. Работа с файловой системой в Node.js: создание, слежение за файлом, запись, запись в конец, чтение, синхронные асинхронные операции. Пример.

**39+**

40. Работа с файловой системой в Node.js: запись в файл потока октетов, чтение из файла потока октетов. Пример.

**40+**

41. Работа с файловой системой в Node.js: запись в файл массива 32-битовых целочисленных данных, чтение из файла массива 32-битовых целочисленных данных. Пример.

**41+**

42. Применение потокового чтение (Readable) и записи (Writable) файлов в Node.js. Пример.

**42+**

43. Применение функции pipe для обработки данных (файла) запроса и записи в файл файловой системы. Пример.

**43+**

44. Применение функции pipe для обработки данных (файла) файловой системы и записи в http-ответ. Пример.

**44+**

45. Разработка клиент-серверного TCP-приложения: обмен текстовыми сообщениями. Пример.

**45+**

46. Разработка клиент-серверного TCP-приложения: пересылка массива целочисленных данных. Пример.

**46+**

47. Разработка клиент-серверного TCP-приложения: пересылка файла от клиента серверу. Пример.

**47+**

48. Разработка клиент-серверного TCP-приложения: пересылка файла от сервера клиенту. Пример.

**48+**

49. Разработка клиент-серверного TCP-приложения прослушивающего два порта, обмен текстовыми сообщениями. Пример.

**49+**

50. Разработка клиент-серверного UDP-приложения: обмен текстовыми сообщениями. Пример.

**50+**

51. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического SELECT-запроса.

52. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического INSERT-запроса. Пример.

53. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического UPDATE-запроса. Пример.

54. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамичского DELETE-запроса. Пример.

55. Разработка приложения, выполняющего запрос к SQL-базе данных: вызов удаленной процедуры. Пример.

56. Разработка приложения, выполняющего graphql-запрос к SQL-базе данных: query-запрос. Пример.

57. Разработка приложения, выполняющего graphql-запрос к SQL-базе данных: mutation-запрос. Пример.

**AJAX:**

Модель Ajax предполагает использование веб-API в качестве прокси для более разумного запроса данных, а не просто для того, чтобы браузер перезагружал всю страницу. Давайте подумаем о значении этого:

1. Перейдите на один из ваших любимых сайтов, богатых информацией, таких как Amazon, YouTube, CNN и т.д., и загрузите его.
2. Теперь найдите что-нибудь, например, новый продукт. Основной контент изменится, но большая часть информации, подобной заголовку, нижнему колонтитулу, навигационному меню и т. д., останется неизменной.

Это действительно хорошо, потому что:

* Обновления страницы намного быстрее, и вам не нужно ждать перезагрузки страницы, а это означает, что сайт работает быстрее и воспринимается более отзывчивым.
* Меньше данных загружается при каждом обновлении, что означает меньшее потребление пропускной способности. Это не может быть такой большой проблемой на рабочем столе в широкополосном подключении, но это серьезная проблема на мобильных устройствах и в развивающихся странах, которые не имеют повсеместного быстрого интернет-сервиса.

Чтобы ускорить работу, некоторые сайты также сохраняют необходимые файлы и данные на компьютере пользователя при первом обращении к сайту, а это означает, что при последующих посещениях они используют локальные версии вместо загрузки свежих копий,  как при первой загрузке страницы. Содержимое загружается с сервера только при его обновлении.