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 |

**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)

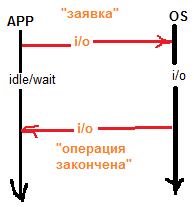
\* Java / JVM / Application Server

\* 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

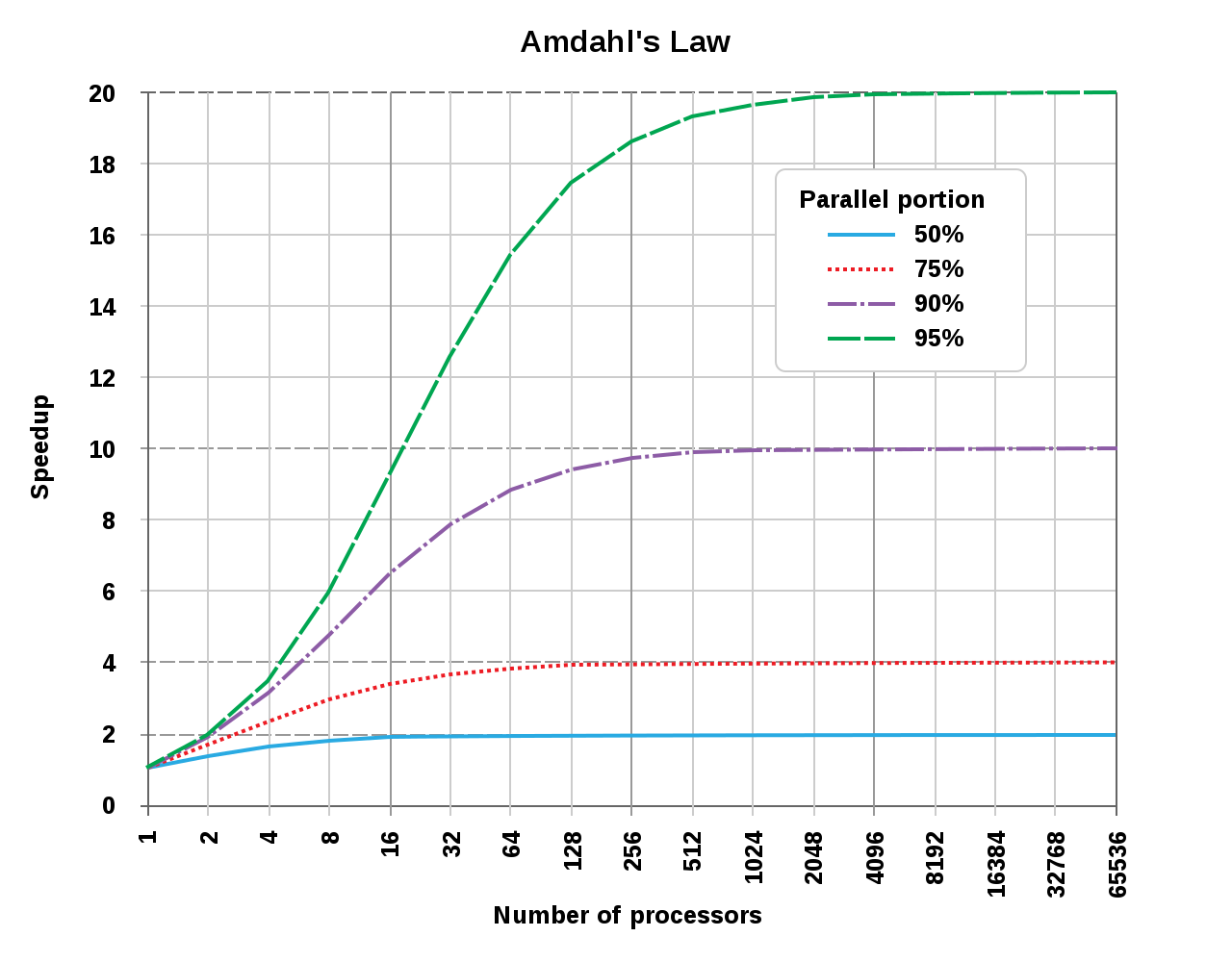
2) асинхр прогр – паттерн Reactor – в Nginx

**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

1. HTTP-аутентификация (Basic, Digest, Forms)

**Идентификация** – заявление пользователя о себе.

**Аутентификация –** процедура, направленная на подтверждение идентификатора пользователя (логин и пароль, как правило). На основании какого-то секрета.

**Авторизация** – после аутентификации; проверка есть ли права на выполнение тех или иных действий.

Исп-ся **1 код возврата** (401 (и 200ый само собой))

401 код – Unauthorized – ваш запрос является не авторизированным.   
407 код – для прокси серверов.

**2 заголовка**, кот.мы будем использовать:

\*authorization – от К

\*www-authenticate – от С

Все открыто, злоумышленники могут работать вместо клиента, надо шифровать.



**Способы аутентификации:**

1. BASIC



Ключевое слово **basic** от сервера к клиенту – указывает на вид аутентификации

Realm – То, что знает клиент (твоя аутентификация действует в рамках этого сектора), т.е. определяет область действия аутентификации.

Если на стороне клиента – браузер, то 401 код заставляет браузер зажечь окошечко, которое попросит вас ввести логин и пароль.

Base64 строка – сцепленные через «:» имя и пароль в заголовке authorization.

С получает, проверяет валидность, если все хорошо – отвечает 200ым кодом, если нет – сначала все идет, т.к.выдается код 401.

7617 протокол

1. Дайджест - DIGEST



7616 протокол

Описание схемы:

Идет запрос

Дальше идет ответ 401, в котором указываем заголовки WW-Authenticate и указывает DIGEST. Особенность в том, что указывается не имя и пароль, а хэш имени и пароля.

Как захешировать имя и пароль?

\*добавить домен и т.д.

У С есть БД, где существует хеш и остается проверить валидный хеш или нет.

Является большей защитой, чем basic.

С т.зр. криптографии – также беззащитен как basic.

Исп-ся реже, т.к. не защищает (ур-нь защиты почти как у basic), но больше мороки => исп-ют в основном basic.

1. FORMS

Нет стандарт (нигде не описан = народный способ)



Описание схемы:

Инфа о пользователе и пароле передается в куки

Процедура аутентификации:

Идет запрос

С проверет есть ли куки, если нет – запрос не аутентифицирован

Если куки есть – проверяет содержимое, есть инфа, кот.интересует и она валидна – отвечаем ресурсом; иначе – отправляем страницу, в кот.просим ввести имя и пароль.

Юзер вводит и жмет сабмит, эта инфа отправляется на С. С обрабатывает запрос и формирует на основании имени и пароля токен (бит.посл-сть, кот.С м.проверить на валидность; имеет время жизни). Отправляется на К SetCookie И указывает токен, который должен отправляться клиентом в каждом запросе.

Используется чаще всего, т.к.простой и понятный.

Все 3 способа – слабые. Если не исп-ть HTTPS – данные легко перехватить и модифицировать.

1. Протокол HTTPS. Протокол TSL. Сертификаты. Взаимодействие центра сертификации и владельца защищенного ресурса.

**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.

HTTPS = TLS + HTTP

\*TLS – прокладка между HTTP и TCP

\*есть рукопожатие

**TLS** – transport layer security – протокол защиты транспортного уровня – криптографический протокол, кот обесп защищ.передачу д-х между узлами в сети Интернет. Исп ассим.шифрование для аутентификации, симм.шифр для конфиденциалности и коды аутентичности сообщений для сохранения целостности сообщ.

Исп-ся в прилож., кот раб.с сетью Интернет, таких как веб-браузеры, работа с эл.почтой, обмен мгновенными сообщ и т.д.

Дает возм-сть К-С прилож.осущ.связь в сети т.о., что нельзя производить прослушивание пакетов и осущ.несанк.доступ.

**TLS – аутентификация**

Аутентификация, которая осуществляется на основе сертификата x.509

**x.509** – стандартный формат хранения и транспортировки атрибутов безс-ти.

- стандарт, кот.разработан международным институтом телекоммун., кот.лежит в основе HTTPS и TLS-аутентификации.

- эл.док, кот. выдается Центром Сертификации.

Содержит: имя держателя, адрес, серийный номер сертификата, даты проверки, открытый ключ держателя.

Вместе с сертификатом выдается секретный и публичный ключ.

Публичный находится на самом сертификате.

Секретный ключ выдается отдельно.

***Схема работы TLS:***

**Клиент** выдает запрос серверу (Client Hello).

**Сервер** подписывает свой сертификат и высылает клиенту (Server Hello).

**Клиент** проверяет сертификат в центре сертификации, которому доверяет.

**Клиент** сравнивает данные сертификата с информацией центра сертификации.

**Клиент** сообщает серверу, какие ключи шифрования он поддерживает.

**Сервер** выбирает подходящую длину ключа.

**Клиент** генерирует симметричный ключ, шифрует его открытым ключом.

**Сервер** получает симметричный ключ и расшифровывает его закрытым ключом.

Для шифрования исп сеанс.ключ. получение общего сеанс.ключа К и С произв.по протоколу Диффи-Хелмана.

На этом заканчивается процедура подтвержения связи. Между К и С установлено безопасное соед., д-е, кот будут передаваться по нему, шифр и расшифр с исп-нием симм.криптосистемы до тех пор, пока соед.не будет завершено.



HTTPS позв аутентиф.C, К (если надо), созд. безопасный канал связи (зашифр)

в основе протокол TSL - обесп аутентификацию + шифрование д-х

**как изготовить сертификат и получить временный?**

этот сертификат в основе TSL-протокола  
предполаг, что К и С доверяют общему центру сертификации

цифровая подпись зашифрована приватным ключом  
подпись мб прочитана с пом. откр ключа

на стороне К дб сертификат удостоверяющего центра с публ ключом

в Сертификата обязательно есть публ ключ  
*необх условие:* чтобы К мог проверить правильность сертификата со стороны С надо наличие на стороне К сертификата удостовр. центра с публ ключом  
тогда он м прочитать цифр подпись, кот-ю сделал удостовр. центр на сертификате сервера  
т.о. он м проверить правильность сертификата

**где хранятся сертификаты?**в ОС в спец. хранилище (м добраться 2 способами: через браузер, через косоль ОС)

чтобы браузер мог делать HTTPS запросы к др. серверам, у него дб сертификаты этих удост центров с откр ключами

шифронабор:

- какие шифры будут исп?....

**Почему сервер должен подтверждать?**мы ему указываем свои данные, кредитные карточки….

**HTTPS**

**Генерация – лк 22**

***options компоненты:***  
1) приватный ключ сервера  
2) сертификат серевра, заверенный центром (т.е. там стоит цифр.подпись центра сертификации, которая зашифрована приват.ключом центра сертификации + центр ему выдал публ.ключ)

Если Сервер хочет поддерживать HTTPS Протокол он должен поддерживать эти вещи выше

TLS ниже протокола HTTP  
мы не замечаем что есть шифрование… работаем как с обычным HTTP, скрыто на более низком уровне стека протоколов

***Как изготовить сертификат?***Общепризнанное ПО для работы с SSL/TSLявл библиотека OpenSSL, устан на ПК  
м.исп нв 2 режимах:   
\* писать код на C/C++  
\* исп ф-ции библиотеки для шифр, созд/провер сертифи…. всё что связано с протоколов SSL, TSL

м. устан сокетное соединение, но исполь security-сокеты, исп шифрование

**КОМ СТРОКА**

м сделать всё что необходимо  
наша задача – поиграть в центр сертификации – созд сертификат кот. сами выдаем  
1) созд сертификат своего центра сертификации, кот. мы сами заверим  
2) его же испоьз в кач сервера

1. Создать приватный ключ 2 кБ (2048 бит)  
    генерир GetRSA  
    помещаю в файлик, длина – 2 Кб  
    des3 – шифрую свой сертификат
2. Получить публичный ключ  
    из прив.ключа (в обр сторону нельзя)

получ сертификата:  
1) гене запрос на получение: сделать конфиг файл (он бы ничего не спрашивал) или отвеч на вопросы  
2) на основе запроса генер сертификат

Имея на руках закр ключ, мы изготавливаем запрос с пом к-го мы можем получить сертификат  
Тот сод. инфу о том кто выдал, кому, сколко будет действ, для каких целей будет исп, сод. в себе публ ключ, кот. генерируется на основе приват ключа  
Т.о. будет генерироваться сертификат

наш ключ приватный сейчас в зашифр виде  
чтобы потом каждый раз когда С стратовал, он к.раз не спрашивал какой ключ для расшифр сертификата,  
для этого мы должны …

цифр подпись берется в хэш, кот. потом шифруется -> м.показать сертификат кому угодно

**MMC** – консоль MS, туда добавляется оснастка «Сертификаты», кот мы можем добавлять

Мы должны себя зарегистрировать в список доверненных центров сертификатов  
Мастер импорта сертификатов - м. импортировать файл в это хранилище выше

Подключение не защищено – > не узнал сертификат, нету в хранилище такого удост центра, к-му я бы доверял

как это HTTPS соединение применить в экспрессе? очень просто

**ИТОГИ:**

*если мы хотим сделать Сервер HTTPS что надо?*  
\* приватный ключ  
\*сертификат (не м. получить без прив ключа) – перед этим подготовить запрос на основе прив.ключа вашего + отвеч на вопросы (кто мы…)

1. 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

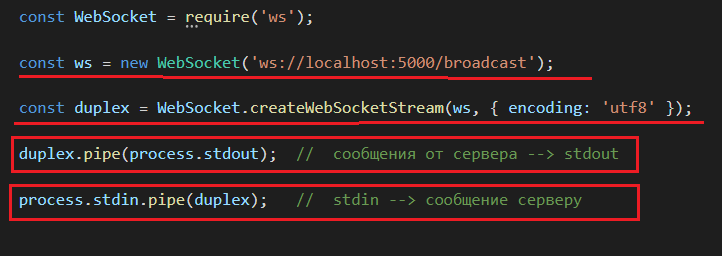
**Широковещ.сообщ** – отсыл.всем узлам в сети.

****

**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)

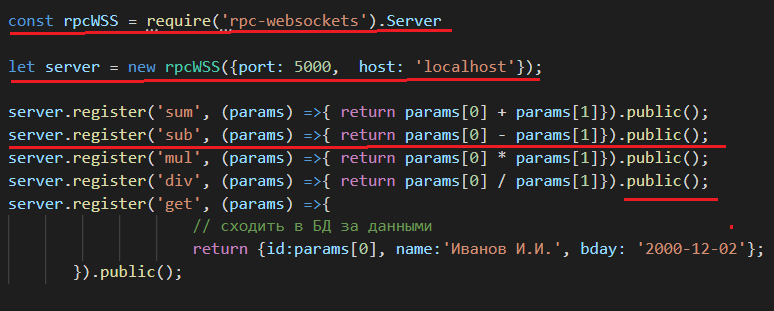
\*Если public – клиент не завершается

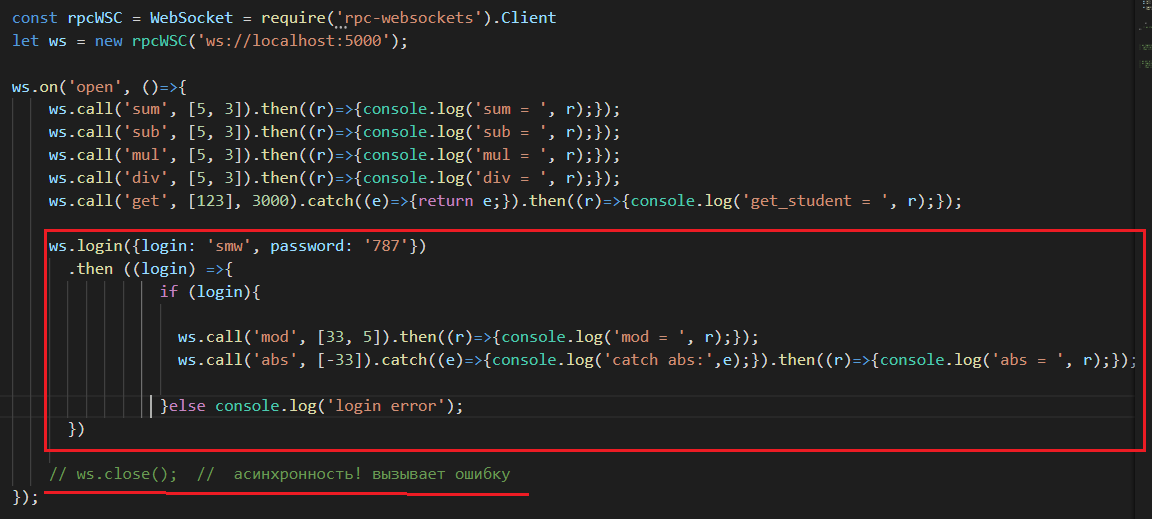
**\***Если protected – требует аутентификации (логин/пароль)

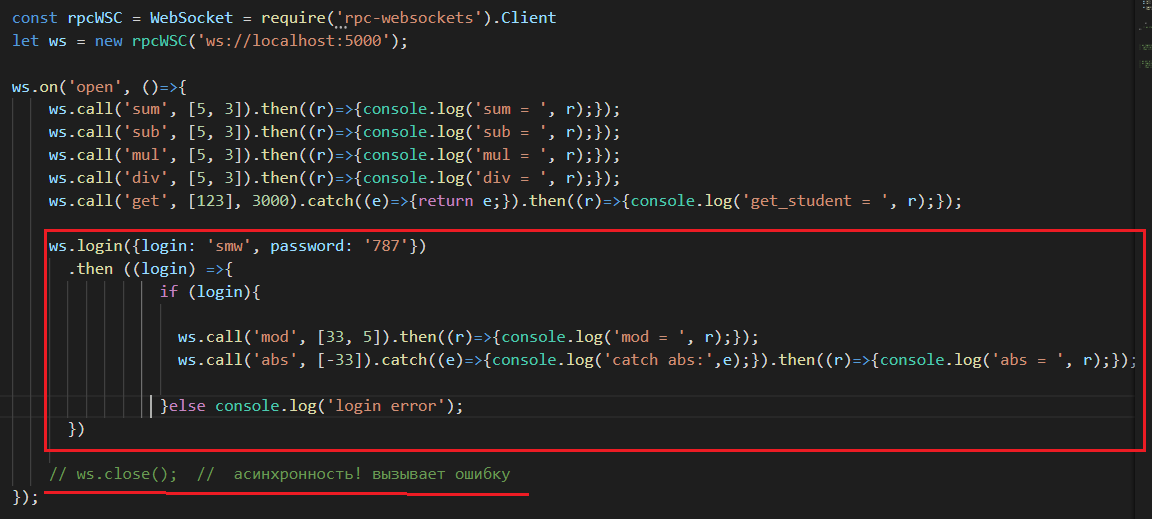


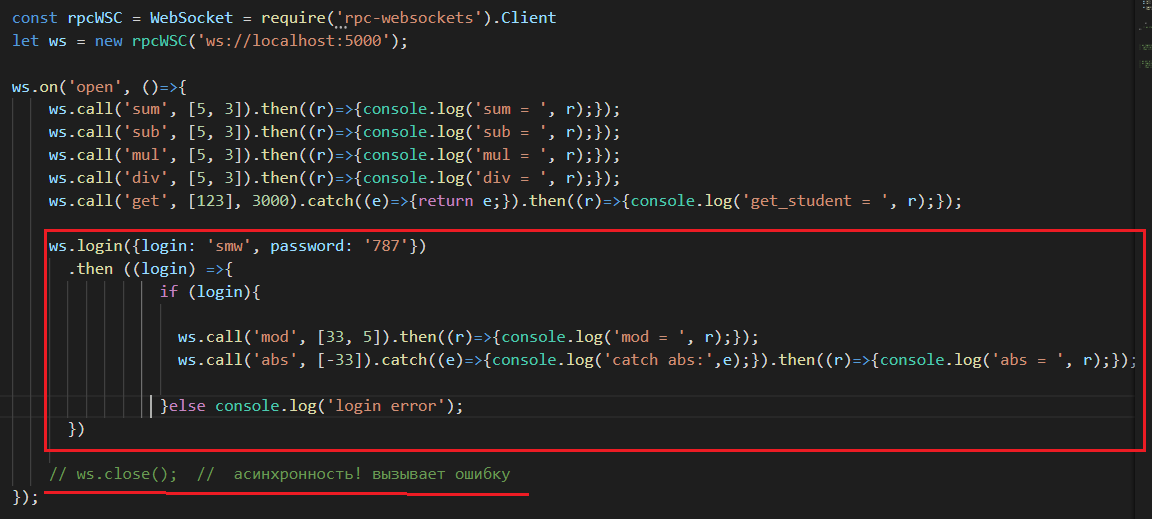






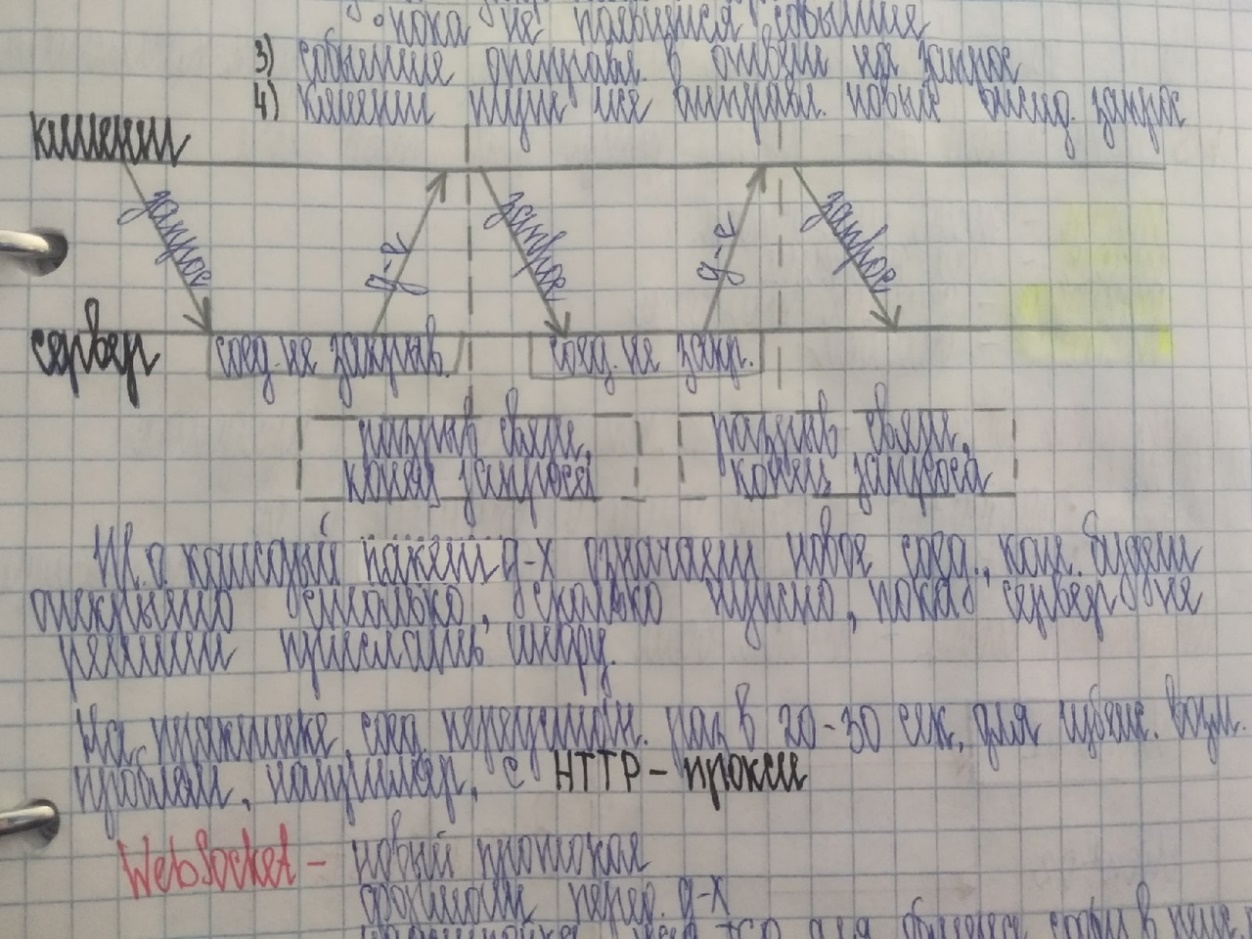






**Схема:**

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.

**2-ой вопрос**

1. Применение СУБД Redis. Основные принципы работы. Пример (лаба).

**Redis:** noSQL СУБД с открытым кодом (BSD-лицензия), Redis Labs, Сальваторе Санфилиппо:

* хранилище данных в оперативной памяти;
* для кэша;
* для посредника сообщений;
* структуры данных: строки, хэш-таблицы, списки, наборы, отсортированные наборы, растровые изображения, геопространственные индексы, HyperLogLog;
* СУБД ориентирована на быстрое выполнение атомарных операций (до 100тыс. set/get-операций);
* механизм снимков для асинхронного сохранения (с потерями);
* механизм упреждающей записи;
* написана на ANSI С;
* последняя стабильная версия: 5**;**
* API: C, C++, C#, Java, JavaScript, Python, … ;
* работает только под Linux.

HyperLogLog - это алгоритм для задачи, связанной с подсчетом, аппроксимирующей количество различных элементов в мультимножестве. Для вычисления точного количества элементов мультимножества требуется объем памяти, пропорциональный количеству элементов, что нецелесообразно для очень больших наборов данных**.**

**Redis** – in memory Database - База данных размещаемая в оперативной памяти.

**Хеш-таблица (hash table)** — это специальная структура данных для хранения пар ключей и их значений. По сути это ассоциативный массив, в котором ключ представлен в виде хеш-функции.

**Персистентность данных** обеспечивается за счет того, что Каждая операция вдобавок запис.в расположенный на диске журнал транзакций

**Основные команды:**

**SET ключ значение** — Записывает строковое значение в переданный ключ. Если ключ до этого существовал, то он будет перезаписан.

**GET ключ** — Возвращает значение ключа.

Del ключ – удаление ключа.

**GETSET ключ значение** — Устанавливает в переданный ключ строковое значение и возвращает предыдущее.

**DECR ключ** — Уменьшает на единицу значение числа. В случае, если заданный ключ будет содержать строку, будет сгенерирована ошибка.

**INCR ключ** — Увеличивает на единицу значение числа. В случае, если заданный ключ содержит строку, будет вызвана ошибка.

**MSET ключ значение [ключ значение …]** — Устанавливает значение ключа / значения ключей, которые переданы в параметрах.

**MGET ключ [ключ …]** — Возвращает значение ключа / ключей, переданных в параметрах.

**HSET ключ поле значение** — Добавляет в хэш поле и значение. Если такого ключа не существовало, он будет добавлен. В случае, если такое поле в хэше уже существует, оно будет перезаписано.

**HGET ключ поле значение** — Возвращает значение, которое ассоциировано с полем в хэше

**HMSET ключ поле значение [поле значение …]** — Записывает значения в поля хэша.

**HMGET ключ поле [поле …]** — Получает значение поля / полей

указанного хэша.

EXISTS – уведомляет, если ключ существует.

1. Применение пакета Sequelize. Основные принципы работы. Пример (лаба).

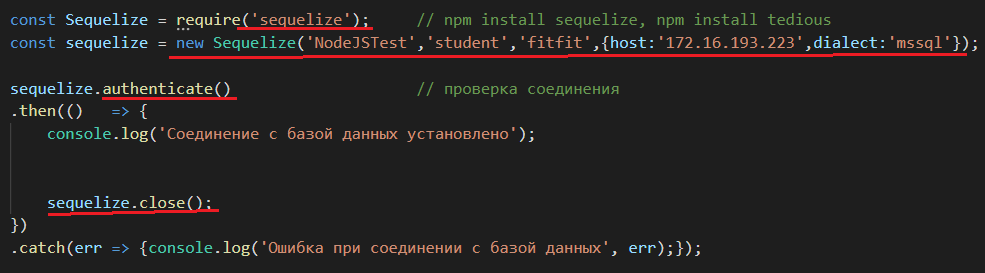
**ORM (Object-Relational Mapping)**  - технология программирования, которая позволяет работать с SQL-базой данных, как с набором программных объектов. **Mapping**: база данных – объект contextDB, таблица – коллекция объектов, строка в таблице – объект, структура таблицы – класс.

**Sequelize -** npm-пакет, реализующий ORM-технологию, кот.повзоляет работать с SQL базой данных.

Может применяться для: Postgres, MySQL, mariadb, sqlite3, Microsoft SQL Server.

**Объектам БД** (БД, таблицы, структуры, строки) ставятся в соотв.прогр.объекты.

1. **Sequelize:** соединение с БД, проверка соединения, закрытие соединения.

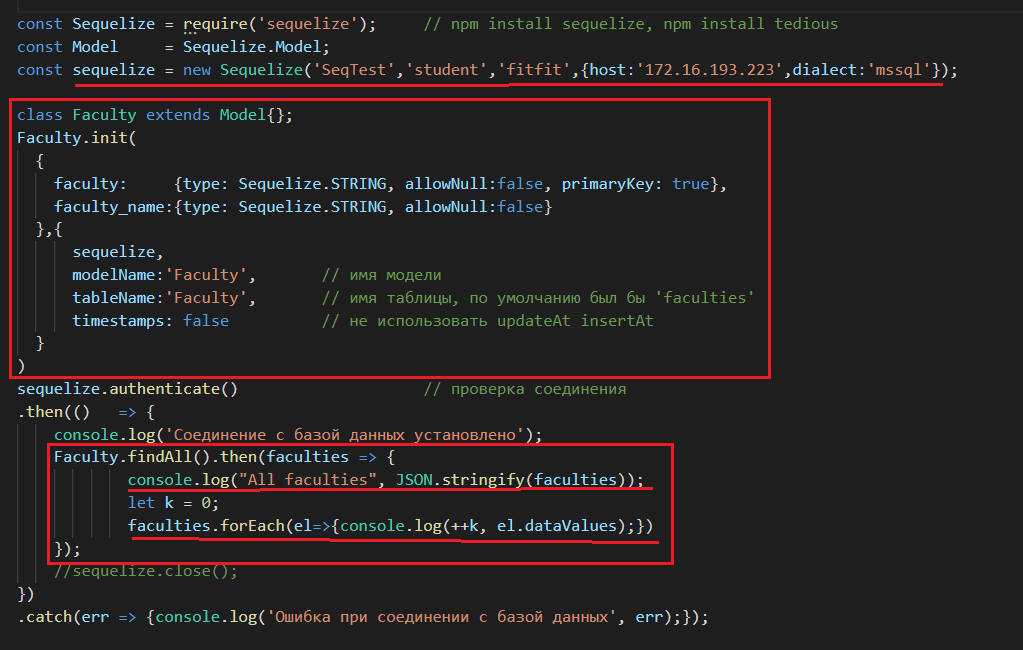
****

2) $ npm install --save tedious *# Microsoft SQL Server*

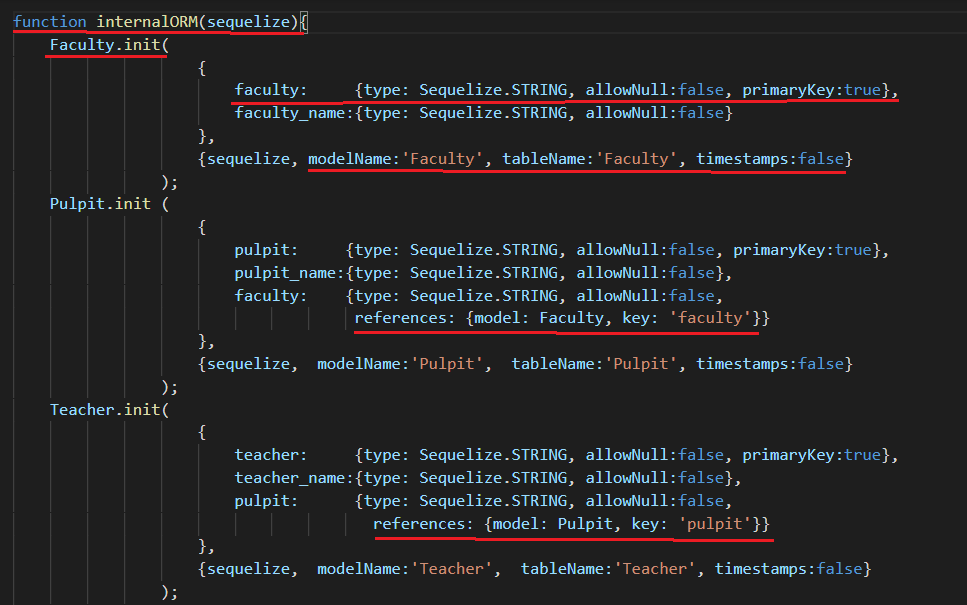
**Tedious** – протокол прикладного уровня, который использует mssql поверх TCP.

3) 1 – стра-ра таблицы, 2 – ук.модель

Timestamps – false: отслеж.измен.в табл(доп.поле будет созд или нет)

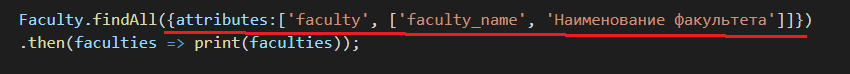
****

**4) reference** – указать внешний ключ.

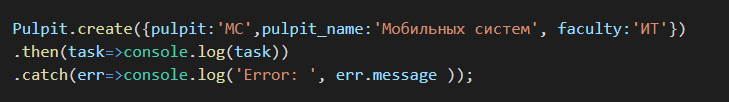
****

**5)** Операции

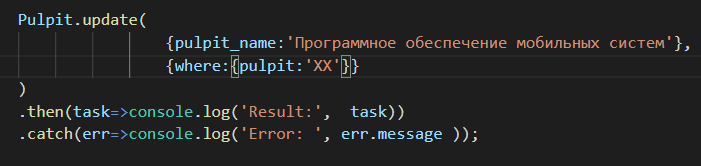
select – findAll()

****

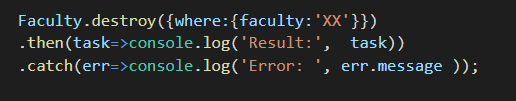
insert – create();

****

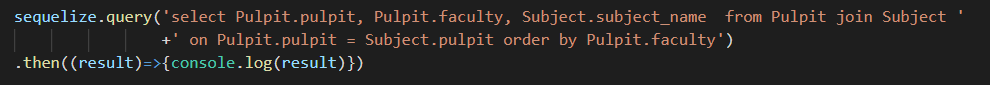
update – update();

****

delete – destroy();

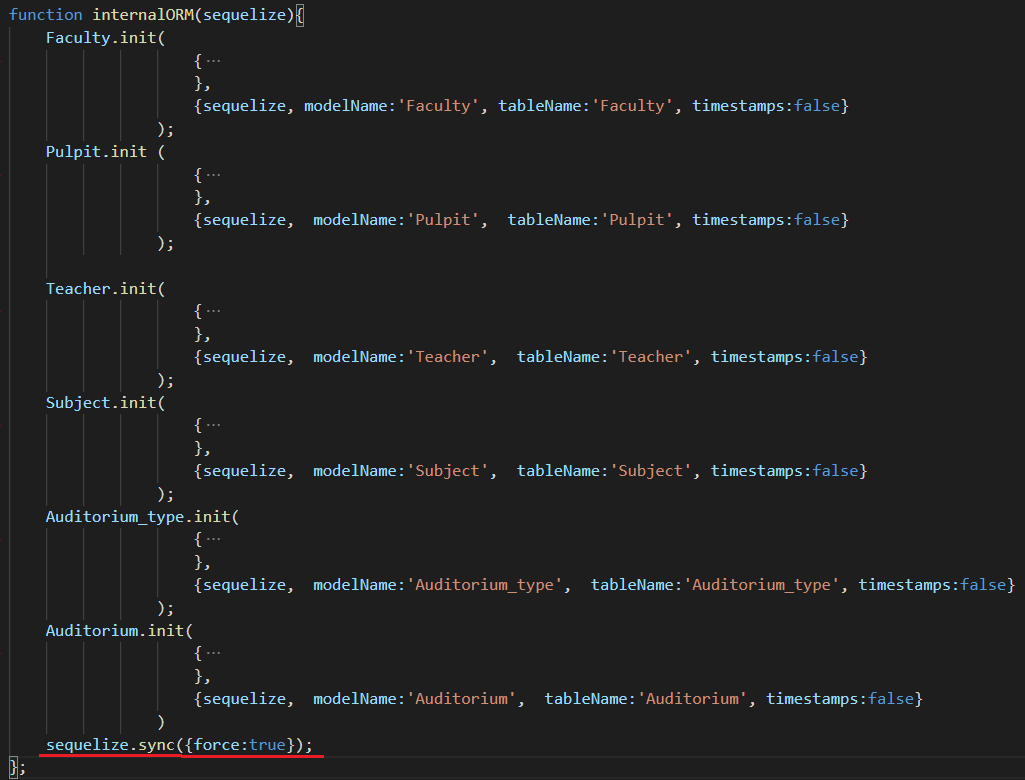
****

Query -

****

**Сырые запросы –**.м.тупо вписать select-запрос.

**Sequelize.sync() –** вып.синхронизацию с БД: м.создавать обхект.схему, уложить ее в БД и она авто-создаст все таблицы.

****

1. Пакет Express. Основные принципы работы. Middleware-код. Пример.

**Пакет express** – пакет 2го уровня; фреймворк для web-прилржений, кот.представл. обширный набор ф-ций.

**Middleware** – промежуточное ПО;

для http-сервера – нек. конвейер обработки htpp запроса.

В джаве конвейер – фильтр;

В ASP.Net - http модули;

В ASD.NET Core – middleware;

**Middleware**-функции – функции промеж.обработки – ф-ции, кот.имеют доступ к req и res, и к след.ф-ции обработки в цикле «запрос-ответ».

Функция, которая передается в app.use(), принимает три параметра:

* **request**: данные запроса
* **response**: объект для управления ответом
* **next**: следующая в конвейере обработки функция

**Express** – строит конвейеры для обработки запроса, либо сами обраб., либо передаем след.покмпоненту конвейера.

**Express** – труба, в кот.мы м.вставлять свои фильтры, перехватывая каждый запрос, перехватывая его или пропуская дальше в трубу.

1. **Express:** require(express) – подключить модуль
2. **App –** подключаем объект
3. С помощью **use(**обработчик запроса (3 пар-ра (обработать req, сформир res, прогнать дальше по конвейеру обработку))) образ.эл-ты конвейера

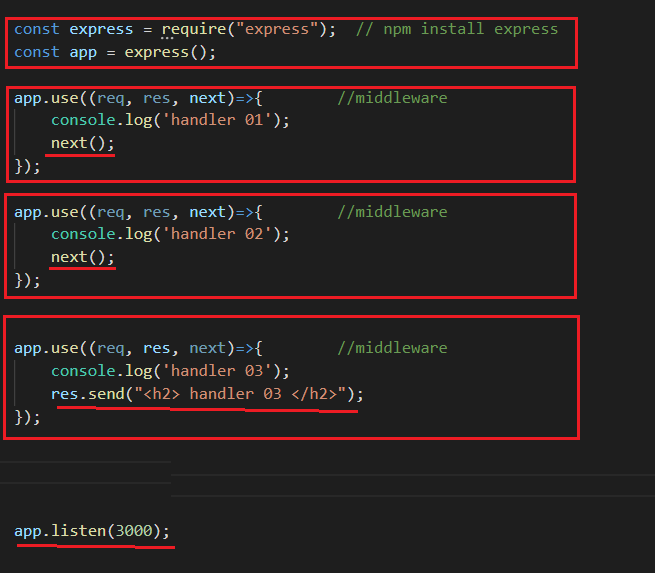
Их физ.послед-сть = порядок обработки запроса от клиента.

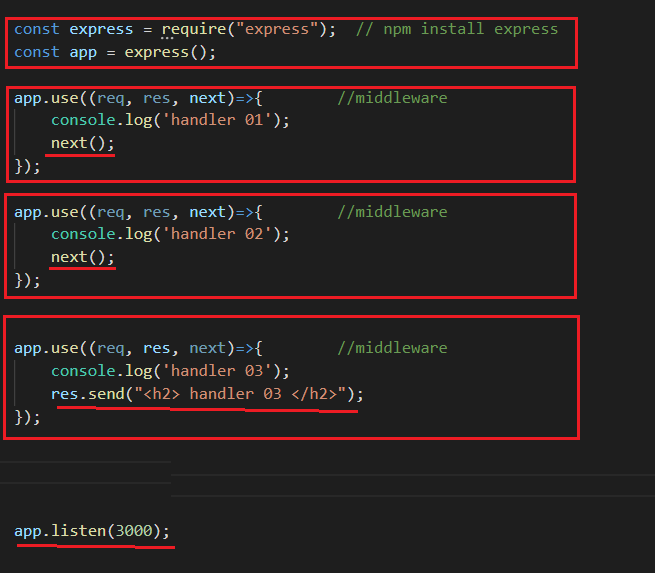
Сначала первый, затем второй и т.д. Все они футболят запрос с помощью **next().**

Последний – обработчик, который формир.ответ.

**Response и request** – не те, которые из http; это собственные объекты expressа.

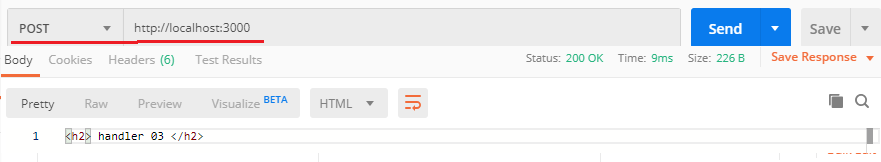
Их методы и св-ва значительно расширены => упрощ.разработку сервера.





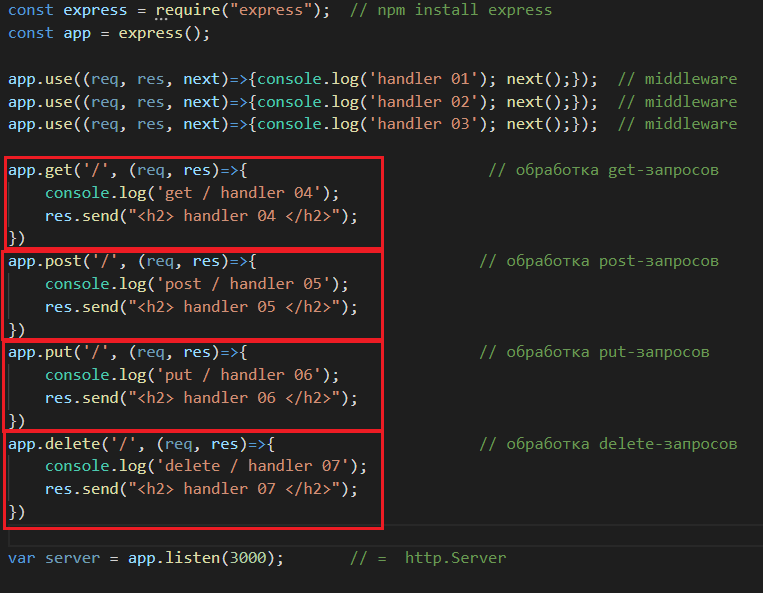
Спом. **app.use()** мб обработаны любые запросы (get, post, put) – не надо указывать в коде что за запрос.

1. **Express:** POST, PUT, DELETE

****

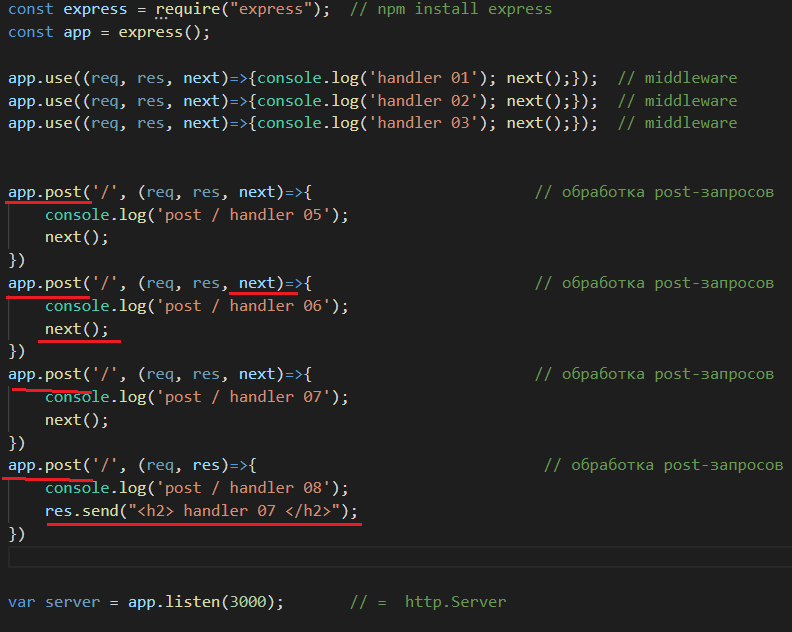
1. **Express:** routing, get/post/put/delete-response

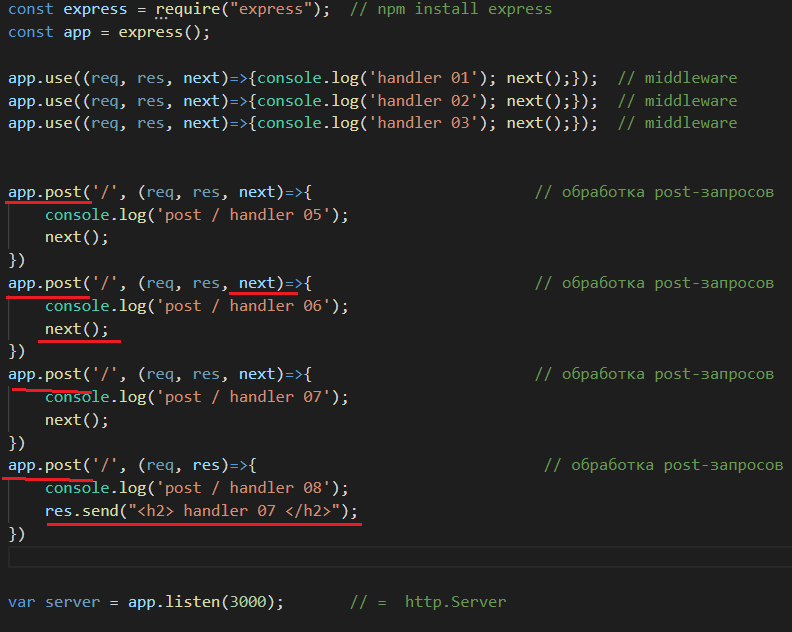
Для разделения по методам. Указ.2 пар-ра, но м.ип-ть 3 пар-р(next)

****

1. **Express:** routing/middleware, get/post/put/delete-response

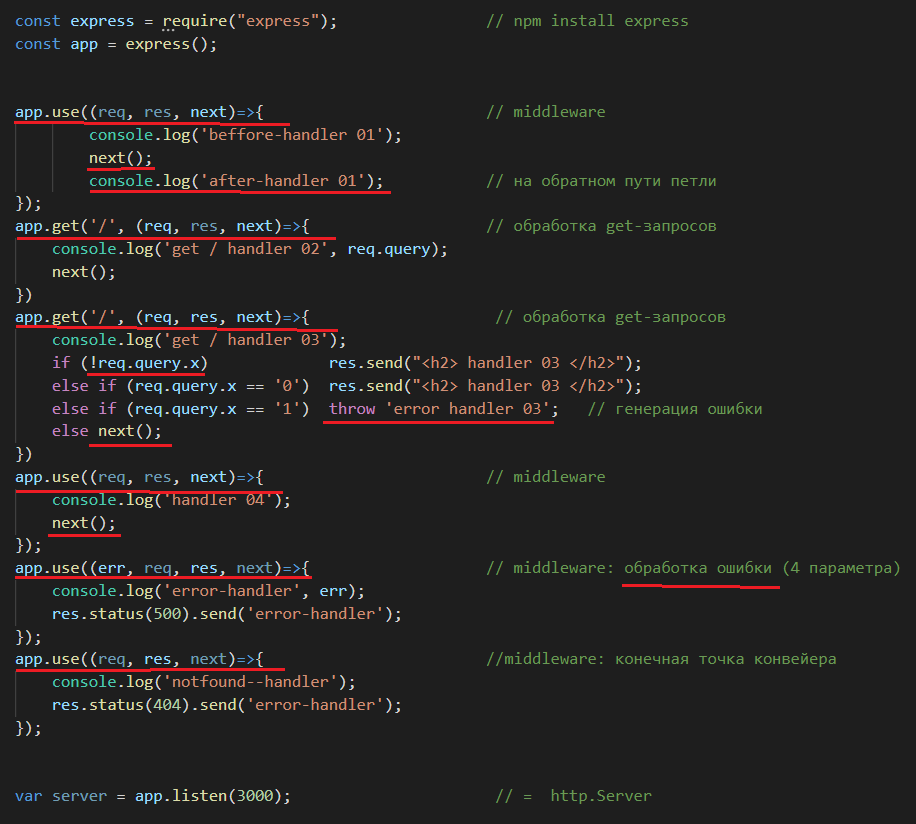
Тож самое, что и вверху, только с 3 пар-ром.

****

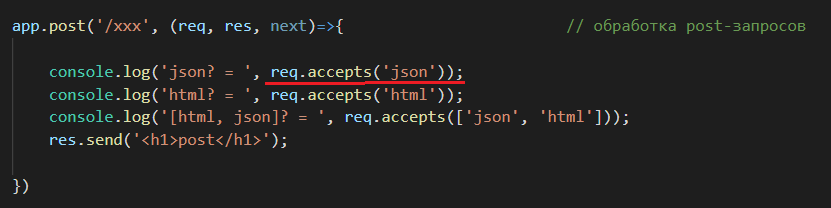
****

1. **Express:** конвейер

Конвейер образ.петлю = движется пока не делаем некст, а потом разворачивается и идет обратно.

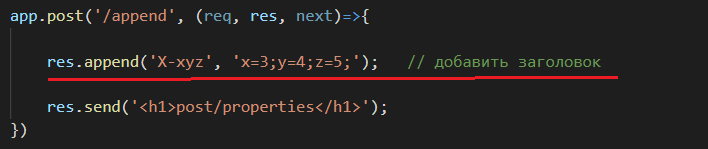
****

1. **Express:** объект request, методы объекта request

****

1. **Express:** объект response, методы объекта response

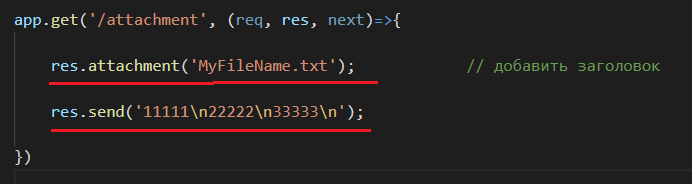
**Append** – добавить заголовок

****

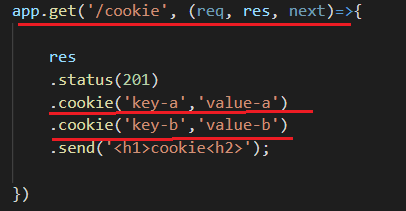
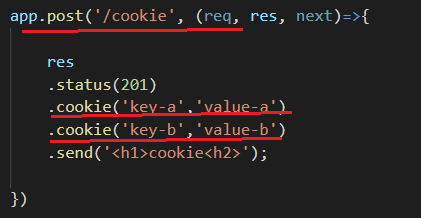
**Attachment –** добавить какой-то файл

! добавляем заголово о том, что в ответе файл и его типа

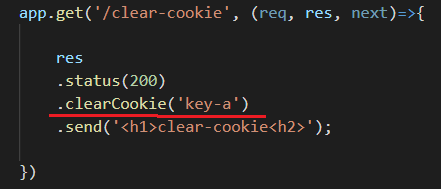
! указывает клиенту не отображ.содерж.боди. а сохранить в файловой стр-ре

****

**Cookie** – добавить куки в response посмотреть куки в браузере (applic в консоли)

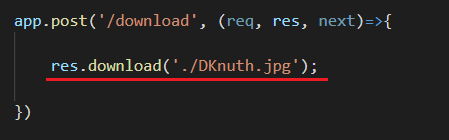
****

**Clear-cookie -** Почистить куки

****

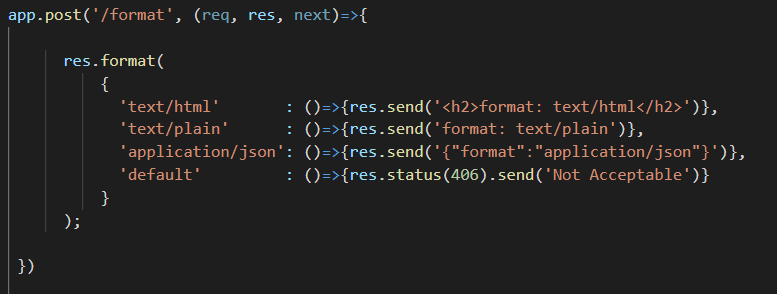
**Download –** выгрузить что-либо (файл)

! ставит заголовки, а файл помещает в боди

****

**Format –** позвол.сделать обработку на различные accept’ы (если html – одно, plain - др)

Если указать несколько через «,» в постмане – сделает первый (приоритет такой)

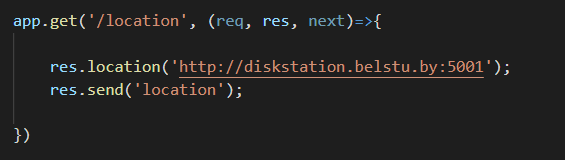
****

**Links –** исп.для указания связи (взаимоотношений) нашего ответа с другими объектами, которые расположены по некоторым uri.

Суть – м.сделать заголовок, в кот.указать доп.ссылку на что-то и придать какое-то смысловое имя с пом.ключ.слова get.

****

**Location –** формир.заголовок location; для того чтобы казывать имя нового ресурса, куда клиент должен сделать новый скачок. (переадресация)

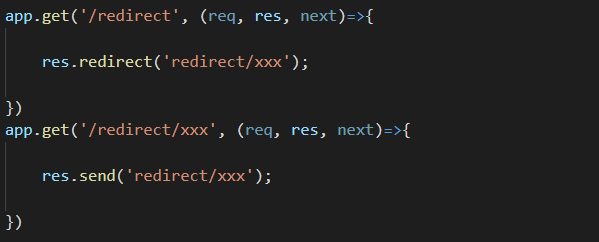
****

**Redirect –** переадресация (3хх)

308 – такой же запрос, как был первичный

303-305 – get Запросы

Если мы хотим, чтобы К переслал тот же запрос после redirect => указ.код ответа **308**

****

**Шаблонизация** – мех.генерация текст.сообщения на основе готово шаблона.

**Шаблон –** статический текст с вкраплениями спец.символов (комбинаций символов), предназначенных для динамической генерации новых фрагментов текста.

**Шаблонизатор** (template engine) – программа, выполняет шаблонизацию языково-независимого шаблонизатора (Mustashe)

**Контекст шаблона** – д-е кот передаются для вывода

**Handlebars** (hbs) – усы подкруч. вверх – шаблонизатор, расширение Mustashe.

npm install express-handlebars

**Макет** (layout) – файл, кот.позвол. задать шаблон шаблона.

**Partial view** – Общие элементы, которые м.вставлять в какую-то стат.разметку.

**Helper –** управл. процессом разметки в результате применения шаблонизатора.

Производит определенные операции с параметрами и возвращает html код

Перечислите встроенные хелперы Handelbars.

1. If – для вывода блока по условия
2. Unless – обратный if (выведет если ложно)
3. Each – для перебора списка
4. With – для сдвига контекста секции hbs-шаблона
5. Log – логировать состояние контекста во время вып-ния шаблона

Поясните назначение http-заголовков: **Cache-control, ETag, Last-Modified**, ***If***-**Modified-Since, If-None-Match.**

1. Cache-control – для вкючения кэширования
2. ETag – для идентификации версию ресурса, найденного на URL
3. Last-Modified – дата последней модификации сущности
4. If-Modified-Since (дата) – выполнять метод, если сущность изменилась с ук.момента
5. If-None-Match – вып-ть метод если ни одного из них не сущ-ет
6. Пакет Express. Основные принципы работы. Маршрутизация. Пример.

**Маршрутизация** определяет, как приложение отвечает на клиентский запрос к конкретному адресу (URI).

**Маршрутизатор** – по uri решаем какой контроллер будет обрабатывать запрос.

**Контроллер –** класс, в котором есть метод action.

Т.е.когда призодит запрос он авто проверяется на соотв.шаблонам, дальше выбирается имя контроллера и экшена.

При обработке запросов фреймворк Express опирается на систему маршрутизации. В приложении определяются маршруты, а также обработчики этих маршрутов. Если запрос соответствует определенному маршруту, то вызывается для обработки запроса соответствующий обработчик.

Для обработки данных по определенному маршруту можно использовать ряд функций, в частности:

* use
* get
* post
* put
* delete

В качестве первого параметра эти функции могут принимать шаблон адреса, запрос по которому будет обрабатываться. Второй параметр функций представляет функцию, которая будет обрабатывать запрос по совпавшему с шаблоном адресу.

|  |
| --- |
| // обработка запроса по адресу /about  app.get("/about", function(request, response){        response.send("<h1>О сайте</h1>");  });    // обработка запроса по адресу /contact  app.use("/contact", function(request, response){        response.send("<h1>Контакты</h1>");  }); |

Когда приходит запрос Express сопоставляет запрошенный адрес с каждым из маршрутов. Затем выбирается первый совпавший маршрут. При совпадении маршрута вызывается его функция обработчика.

Также вместо определения маршрутов мы можем указывать регулярные выражения. Например, необходимо перехватить запрос ко всем файлам html или всем путям, которые в конце имеют ".html":

|  |  |
| --- | --- |
| 1  2  3 | app.get(/.\*(\.)html$/, function (request, response) {      response.send(request.url)  }); |

1. Пакет Express. Основные принципы работы. Статические файлы. Пример.

Статический файл - Файл, кот.отдается в неизменном виде серверу, и С не д.их обраб. (css, js, png)

Для работы со статическими файлами в Express определен специальный компонент **express.static()**, который указывает на каталог с файлами.

Чтобы встроить компонент express.static в процесс обработки запроса, вызывается функция app.use(). Эта функция позволяет добавлять различные компоненты, которые еще называются middleware, в конвейер обработки запроса:

|  |  |
| --- | --- |
| 1 | app.use(express.static(\_\_dirname + "/public")); |

Причем данный вызов помещается до всех остальных вызовов функции app.get().

В саму же функцию express.static() передается путь к папке со статическими файлами. Специальное выражение **\_\_dirname** позволяет получить полный путь к папке.

1. Пакет Express. Основные принципы работы. Обработка query-параметров GET-запроса. Пример (POSTMAN).

Одним из способов передачи данных в приложение представляет использование параметров строки запроса. Строка запроса (query) - фактически это часть запрошенного адреса, которая идет после знака вопроса. Например, в запросе http://localhost:3000/about?id=3&name=Tome часть id=3&name=Tome представляет строку запроса.

Строку запроса образуют параметры. После названия каждого параметра после знака равно (=) идет его значение. Друг от друга параметры отделяются знаком амперсанда. Например, в адресе выше использовалось два параметра: параметр "id" имеет значение "3" и параметр "name" имеет значение "Tom".

В express мы можем получить параметра строки запроса через свойство **query** объекта request, который передается в функцию обработки запроса.

С помощью выражения request.query мы можем получить все параметры строки запрос в виде объекта javascript, а с помощью выражения request.query.название\_параметра мы можем обратиться к каждому отдельному параметру.

1. Пакет Express. Основные принципы работы. Переадресация. Пример(POSTMAN).

Для переадресации применяется метод **redirect()**:

redirect([status,] path)

В качестве параметра path передается путь, на который будет перенаправляться пользователь. Дополнительный параметр status задает статусный код переадресации. Если этот параметр не задан, тогда по умолчанию отправляется статусный код 302, который предполагает временную переадресацию.

С помощью данного метода можно выполнять переадресацию как по относительным путям, так и по абсолютным, в том числе на другие домены.

Переадресация по абсолютному пути:

|  |
| --- |
| const express = require("express");  const app = express();    app.use("/index",function (request, response) {    response.redirect("[https://metanit.com](https://metanit.com/)")  });    app.listen(3000); |

В данном случае при обращении по пути "/index" будет идти переадресация на сайт https://metanit.com.

Переадресация по оносительным путям также не очень сложна, но здесь важно учитывать, как именно определяется адрес для редиректа. Рассмотрим редирект относительно текущего пути, с которого производится редирект. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | const express = require("express");  const app = express();  app.use("/home",function (request, response) {  response.redirect("about")  });  app.use("/about", function (request, response) {  response.send("<h1>About</h1>");  });  app.listen(3000); |

В данном случае будет идти перенаправление с ресурса "/home" на ресурс "/about", то есть, условно говоря, с http://localhost:3000/home на http://localhost:3000/about.

1. Пакет Express. Основные принципы работы. Выполнение shell-команд (spawn, pipe). Пример.

Spawn – ф-ция, кот позвол запускть консоль, с пом кот м.запускать любые прилож.

Мы создаем процесс cmd.exe а он в свою очередь делает CreateProcess для других процессов => такие процессы для cmd.exe – дочерние, а для нас – внучатые

//cmd.exe - указ.что будем запускать, дальше идет массив пар-ров оболочки

//u - вывод в юникод

//с - необх.выполнить и завершить работу консоли

//dir - сама команда - показать директорий

const dir = spawn('cmd.exe', [ '/U', '/C', 'dir']);

//findstr - отыскивает строки во вх.потоке, кот соотв.нек.шаблону

// /с - ключ, все что сод. 43 - все такие строки надо показать

const findstr = spawn('findstr', ['/c:43']);

Метод child\_process.spawn запускает новый процесс с помощью заданной командой. Он содержит следующую подпись:

child\_process.spawn(command[, args][, options])

**Параметры**

Ниже приводится описание используемых параметров:

command (String) — запускаемая команда

args (Array) — список аргументов строки

options (Object) — может содержать один или несколько из следующих параметров:

Метод spawn() возвращает потоки (stdout & stderr), он должен использоваться, когда процесс возвращает большой объем данных. spawn() начинает принимать ответ, сразу после начала выполнения процесса.

1. Пакет Express. Основные принципы работы. Запуск процесса операционной системы (exec), работа со стандартными потоками ввода/вывода. Пример.

exec — метод child\_process.exec запускает команду в оболочке/консоли и буферизует вывод.  
child\_process.exec(command[, options], callback)

\* command (String) — команда для запуска с аргументами, разделенными пробелом  
\* options (Object) — может содержать один или несколько параметров  
\* callback — функция принимает три аргумента error , stdout и stderr, которые вызываются с выходными данными при завершении процесса.

—cwd (String) — текущий рабочий каталог дочернего процесса

—env (Object) — ключи-значения для среды

—encoding (String) — (по умолчанию: ‘utf8′)

1. Пакет Express. Основные принципы работы. Выполнение js-скриптов в отдельном процессе (fork, send, worker). Пример.

Метод child\_process.fork является особым случаем spawn() для создания процессов Node  
  
child\_process.fork(modulePath[, args][, options])  
  
\* modulePath (String) — Модуль для запуска в дочернем элементе.  
  
\* args (Array) — Список аргументов строки  
  
В дополнение к тому, что он содержит все методы, что и стандартный экземпляр дочернего процесса, метод fork возвращает объект со встроенным каналом связи.

Send() - Возвращает Boolean.

Если Node.js был порожден с IPC каналом, можно использовать метод process.send() для отправки сообщений родительскому процессу. Сообщения будут получаться событием ‘message’ в объекте ChildProcess родительского процесса.

Потоки воркеров выполняются в изолированном контексте. Они обмениваются информацией с главным процессом, используя сообщения. Это избавляет нас от проблемы «состояния гонок», которой подвержены многопоточные среды. При этом потоки воркеров существуют в том же процессе, в котором находится основная программа, то есть, при таком подходе, в сравнении с применением форков процессов, используется гораздо меньше памяти.

==============================

\* Синтаксис fork() почти аналогичен синтаксису метода spawn(), за одним исключением: здесь не нужна команда, потому что fork() предполагает, что все процессы относятся к Node.js  
  
\* exec(). Этот метод отличается от предыдущих тем, что использует не шаблон событий, а одиночный callback.

**Разница между этими тремя способами заключается в следующем:**  
require('child\_process').spawn(): используется для больших объёмов данных; поддерживает stream’ы; может применяться с любыми командами; не создаёт новый экземпляр V8.  
require('child\_process').fork(): создаёт новый экземпляр V8 и экземпляры работников; работает только со скриптами Node.js (команда node).  
require('child\_process').exec(): неудобен для больших объёмов данных, потому что использует буфер; работает асинхронно, чтобы одновременно получать все данные в callback’е; может применяться с любой командой, не только node.

1. Протокол WebDav. Разработка приложения с применением WebDav. Пример (лабораторная работа).

**WebDav –** Web Distributed Authoring and Versioning – расширение протокола HTTP/HTTPS, поддерживающее совместную работу по управление файлами на удаленных web-северах; применяется для создания сетевой файловой системы; в системах документооборота (document management system).

**WebDAV**: **унаследованные HTTP-методы**

**\*GET** – скачать файл.

**\*PUT** – загрузить файл на сервер.

**\* DELETE** – удалить серверный объект.

Работа WebDav регулируется следующими стандартами: RFC 2291, RFC 4918, RFC 3648, RFC 3744, RFC 3253.

Расширяет HTTP следующими методами:

**PROPFIND** – получить свойства серверного объекта в XML-формате.

**PROPPATCH** – изменить свойства серверного объекта.

**MKCOL** – создать папку на сервере.

**COPY** – копировать на сервере.

**MOVE** – переместить на сервере.

**LOCK**  – заблокировать серверный объект.

**UNLOCK** – разблокировать серверный объект.

1. Протокол JSON-RPC. Разработка клиент-серверное приложение использующее протокол JSON-RPC.

На стороне клиента single-page приложение. сначала загруж какая-то станица, кот. насыщена js-ом и она предст. прилож работающее на ajax-запросы, получ ответы, их обраб, обнов эту страницу, и перезагрузка осущ. лишь единожды – когда стартует прилож.

На стороне С распол веб-сервис – веб-прил, кот. предст интерфейс не конеч. юзера, а интерфейс для других программ. Ее юзеры – другие программы, а не конечные юзеры.

**Есть 2 типа интерфейса таких:**  
\* *REST* (предст программисту в виде наборов uri и правил их исп-я: какие компоненты, из чего сост. эи uri, какие методы мб вызваны (get,post,put,delete)) – до этого мы разрабатывали почти только их

\* *RPC* (предст программисту как набор процедур нах на стороне С, и какой-то способ их вызова – эти проц вызыв, перед в параметры, отрабатывают и возвр рез)

**RPC сервер –** набор процедур, кот.нах на стороне С и есть какой-то способ их вызова, т.е.какой-то интерфейс – эти процедуры вызываются, им передаются пар-ры, они отрабатывают, возвращ.рез-т.

**Где исп-ть json rpc?**

\* efirium – для работы с узлами сети

\* система забикс – мониторинг жд ИС

***xml-rpc*** – его прорадитель, был оч популярен и прост, но перерос в протокол, кот уже стандартизирован – ***soap*** (потом о нем поговорим)  
втоая ветка от него отошла – ***json-rpc*** (основан на формате д-х json, в кач транспорта исп https/https/tcp)

сам json стандартизирован (rfc-4627)

JSON RPC описан на сайте [www.jsonrpc.org](http://www.jsonrpc.org) и нигде не описан в стандартах RFC (не зареган)

**JSON RPC** - протокол удаленного вызова процедур, который использует формат JSON для передачи сообщений.

+ простой

+ понятный

\* порожден от XML RPC

\* основан на формате д-х JSON

\* транспорт – HTTP / HTTPS / TCP Socket

*JSON RPC:*

Предполаг., что есть С, к кот мы м отправл rpc-запросы и получить rpc-ответы, при этом запросы и ответы имеют формат json. То как будет транспортироваться json – данная спецификация не рассматривает. Спецификация только говорит: есть такой формат, давайте его использовать для того, чтобы взаимодействовать К и С.

***Объект запроса JSON RPC –*** это json объект, кот содерж в себе след поля:

* Jsonrpc – версия json prc (2.0)
* Method – то как мы сами назвали процедуру на стороне С
* Params – поле, кот сод.пар-ры (массив/объект/мб не быть)
* Id - уник. С выполн эти процедуру и отвеч за результаты, ссылаясь на Id – надо обесп с-му, кот обесп уникальность – GUID, напр (генер 128-бит посл-сти)) – *необяз*., т.к. бывают вызовы процедур не требующие ответа (уведомления)

***Объект ответа JSON RPC –*** это json объект, кот содерж в себе след поля:

* Version – версия протокола
* Result – объект, понятный клиенту
* Error – взаимозамен.с ресалт, сод.сообщ об ошибке
* Id – ссылается на запрос, на кот отвеч ошибкой

***Структура error:***

* Code – номер, кот идентиф.ошибку
* Message – сообщ, кот сопровожд.код
* Data – д-е в формате json, кот позвол более точно идентиф.ошибку

**Типы параметров:**

* Позиционные – продполаг., что в params перед массив, в кот к. эл-т – параметр

(“params”: [42,23] – это 1й, 2й парам)

* Именованные – в этом случае в params передается объект, кот сост.из 2х св-в (пара ключ-значение); можно размещать вне зависимости от порядка, т.к. процедуры ориентированы на название параметра

(“params” : {“subs” : 23, “mid” : 42}

**Уведомление –** запрос без ответа (не указ id в request)

**RPC с пакетом запросов** – пакет запросов, т.е.это массив запросов, к эл-т кот.- отдельный запрос в формате json.

Ответ – такая же пачка.

1. Разработка клиент-серверного приложения с применением технологии WebAssembly на стороне браузера. Пример(WasmFiddle-компиляция).

**WASM (Web Assembly)** – бин.формат исполняемого файла, кот.мб исполняться в вирт.стек.машине (JS Engine);

* Код быстрее, чем JS
* Поддерживается большинством браузеров
* Выполняется в sandbox
* Есть отладчики
* Открытый стандарт

**WASM –** \* не ЯП, а байт-код

\* загружается в браузер и исполняется в браузере

(формально исполняется JS-движком, а не самим браузером)

\* исполняется виртуальной машиной

\* не имеет ничего общего с web, кроме того что общается с внешним миром через JS

**Идея:** готовим бинарный код исполняемого файла, этот код переносим на сторону клиента и он исполняется на стороне клиента (в JS Engine);

=> разраб.прогу => компилируем => получаем byte-код, кот.мб исполняться в рамках JS Engine => byte-код перетаскиваем на сторону клиента и запускаем на исполнение

**Для чего?**

* Мы отсылаем предкомпилированный код => та часть, кот.выполнялась бы JS engine уже выполнена => экономить можно
* Компилируем не на лету, а в оффлайне (JS налету) => мы не связаны с временем исполнения
* Выигрыш за счет того, что убрали на стороне браузера компиляцию, а также за счет оптимизации (из-за того, что отделили стадию компиляции и стадию исполнения – можем не экономить на времени компиляции и оптимизировать с помощью оптимизаторов, т.е. строить более оптимальный код)

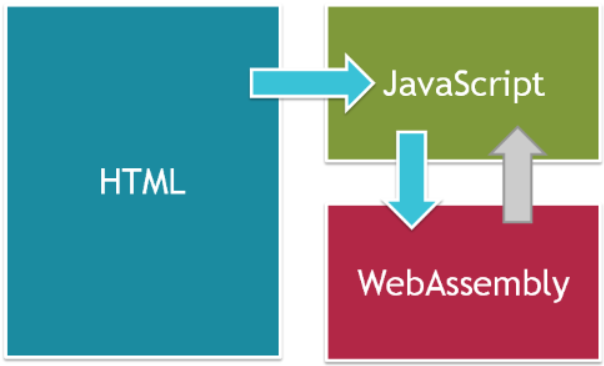
**WasmFiddle** – инструмент, который позволяет проверить как работает wasm.

* Удобный
* Помогает разрабатывать бин.код., а также посмотреть как устроен JS API
* Массив wasmcode = unit8Array – откомпилированные функции, т.е.byte-код функций
* Export – получает инстанс, который позволяет выполнять эти ф-ции.

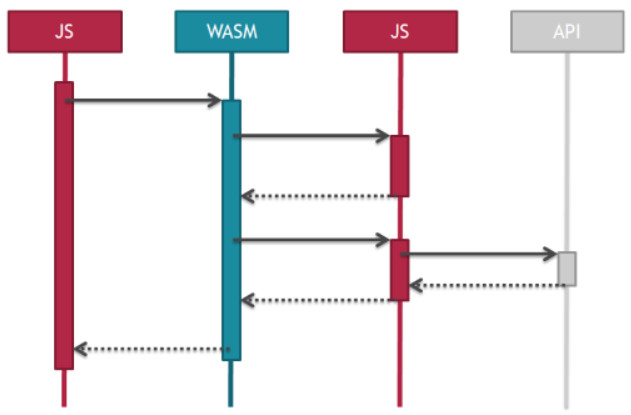
**WASM/Browser – исполнение WebAssembly в браузере:**

Браузер загружает html страницу, с которой выпоняется JS, который уже выполняет загрузку WebAssembly – получатся «модуль» (WebAssembly module), затем создает экземпляр модуля, после чего можно вызывать для этого экземпляра экспортируемые функции.

***Серая стрелка*** – изнутри WebAssembly можно вызвать ф-ции JS.

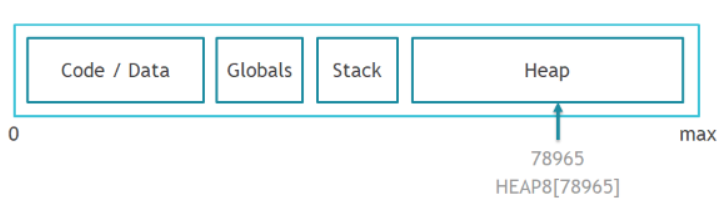


Рассмотрим подробнее вызов ф-ции на диаграмме:

****

Здесь мы сначала из JS вызываем WebAssembly, затем из WebAssembly вызываем ф-цию JS. WebAssembly может пользоваться любыми АПИ. Это возможно, но не напрямую, т.к.такие вызовы происходят через JS.

Модель памяти WebAssembly очень проста. Это плоский «кусок» памяти, в котором находится код проги, глобальные переменные, стек и куча. Есть можность сделать так, чтобы память была расширяемой, т.е.ели при очередном выделении памяти нам не хватает места, то верхняя граница памяти автоматически увеличивается.

****

Весь блок памяти доступен из дж/с просто как массив байтов. Причем эта память доступна как на чтение, так и на запись.

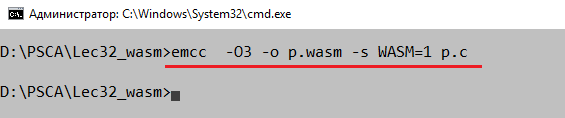
***Byte-код*** можно получить **2 способами:**

* WasmFiddle – сильная оптимизация
* Emcc – компилятор, который позволяет компилировать код из с++ в wasm

EMCC требования:

|  |  |
| --- | --- |
|  | * Указать «extern “C”» - директива, кот позвол сказать, что имена ф-ций дб написаны согласно правилам С * Emscripten\_keepalive – префикс, кот.позволяет экспортировать ф-ции и исп-ть их извне |

*Компиляция в WASM:*

****

* O3 – флажок, который указывает до какого уроням хоти преобр (высший)
* -о p.wasm – во что компилируем
* Wasm = 1 – хотим получить wasm- код
* P.c – из чего компилируем

1. Long pool–сервер, принцип работы. Пример (Telegram bot, лабораторная работа).

что такое белый IP адрес?

мы будем в осн. работать с готовыми крупными апи (гугл, амазон, алибаба), с облачными техн…  
при работе с телеграмом постараемся разраб телеграмный бот, вып. простейшие функции.  
есть 2 подхода, сегодня с одним

1) позв разраб робот, кот м. отысылать на запросы кот мы отсылаем в телеграм, реагировать на них

**бот** – псевдо юзер в телеграме, от явл. автоматическим  
он у смелова взаимод. с гугл-календарем  
м. принимать запросы и отпр ответы

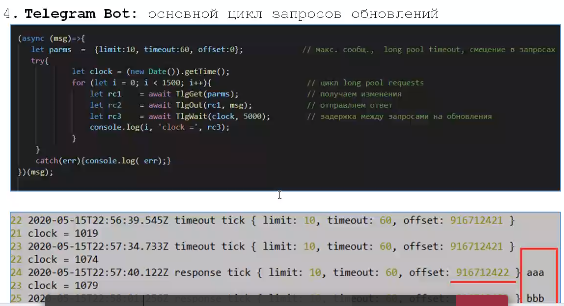
2 общих подхода для создания бота (web hook ; long pool запрос)  
бот отправляет данные от собеседника к серверу, кот мы разработали

1. юзер ввод сообщение  
2. это сообщение перехват теелграм и перенаправля к нам на сервер  
3. мы его обраб,  
4. отправл в нуэном формате телеграма  
5. теелграм его передает

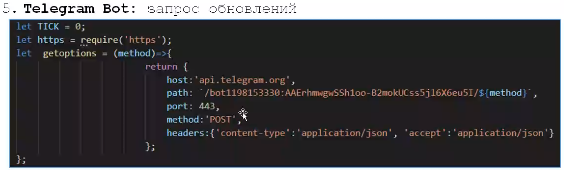
**2 вариант (мы рассмотрим): LONG POOL запрос**это работа в обр. сторону  
1) ест К, кот периодично обращ к веб-апи телеграма  
скачивает и получает те команды, кот введены юзером  
если никто ничего не вводит, Сервер зависает на заданное системное время  
т.е. телеграм ждет это время поступления сообщений  
если за зад время сообщ не поступают  
то телеграм отпр нам ответ в кот говорит о том что никто ниче не пишет  
мы обраб это сообщ  
и делаем новый запрос

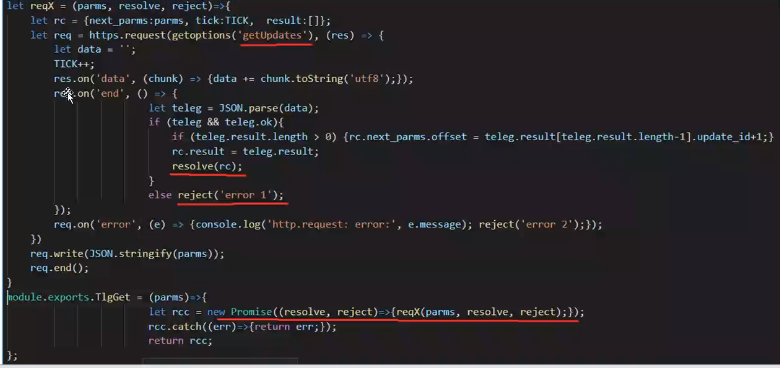
**лонг пул** – отпр запрос, тот подвисает на заднный таймаут, телеграм это время ждет поступления новых запрсово. Если за это время поступило неск запросов, то он нам их отдает в одном овтете

Буддем говорить о разработке вебклиента, кот будет постоянно опрашивать сервер телеграмма и получать от него ответ и снова посылать ему запрсо

  
это клиент

пришлось придерживаться асинхронности  
**async** указывает что здесь будут использ асинх функции  
**await** – вызыв асинх ф, кот позволяют нам получить результат выполнения этих асинх функций  
**TlgGet** – делаем запрос к телеграму и получает те данные, кот ввели боту  
*парам* : (макс кол-во сообщений ; таймаут кот ждет сервер [с]; смещение в запросах – с какого сообщ нам надо считывать (все юзеры нумеруются, мы указ № начиная с к-го нас интересуют очередные 10 сообщ)  
**TlgOut** – для отправки ответа телеграму  
**TlgWait** – задержать на 5 сек между циклами

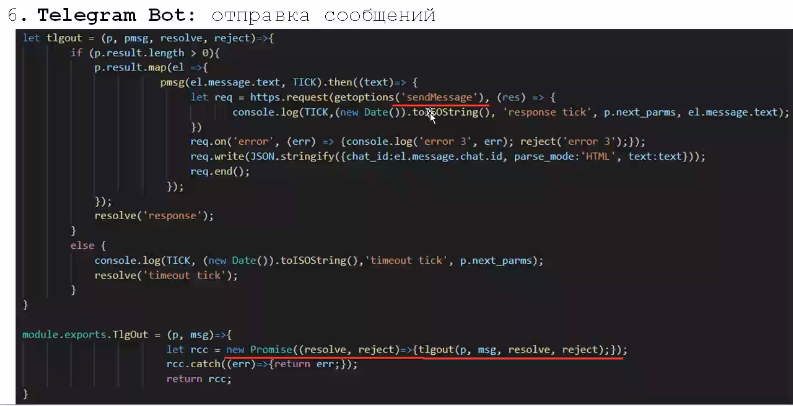
  
получаем ссылку после того как пропишемся в телеграме, это хост по к-му мы моем обращаться чтобы работать с ботом  
тут еще указ метод – что я хочу от телеграма  
будем рассматривать цикл считывания *изменений* (то что мы вводим)



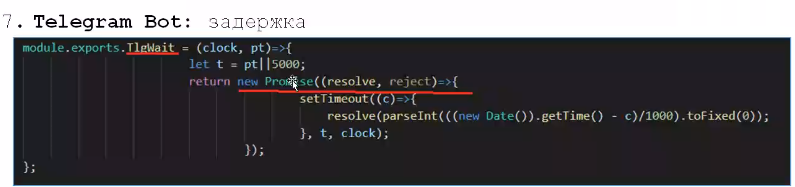
**Promise** – мех, кот позв в более удобном виде записать асинх функцию  
объект позв применять конструкции .then, .catch – позв избавиться от сильной вложенности js-функций. Он обрабаывает либо *resolve* либо *reject* (что-то не так отработало)

resolve(rc) – функция для возвр резульатта если все норм

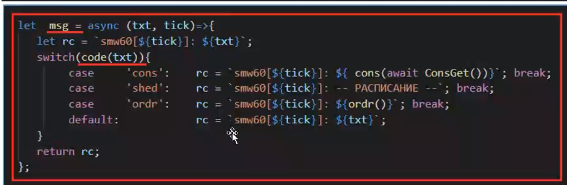
2 ЧАСТЬ ОТПРАВКА ОТВЕТОВ

отправить сообщений телеграм

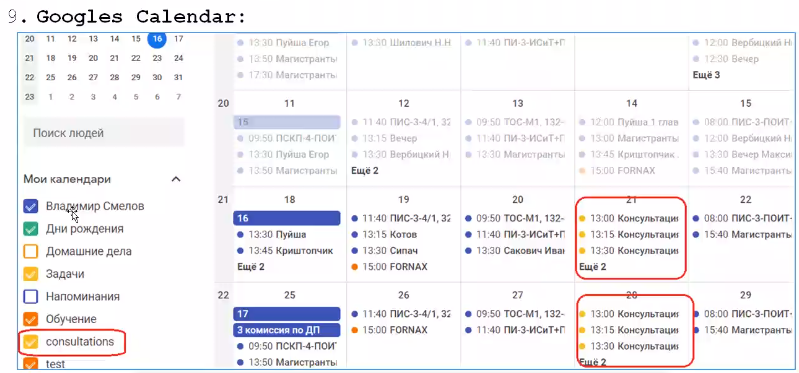
конкретному юзеру, в конкр чат на конкр запрос  
p – результат пред. TlgGet – если они есть, начиню их все обрабатывать  
в конце возвр *resolve (‘response’)* – все в порядке  
если ниче нет, то выдаю *resolve(‘timout tick’)*   
*pmsg* – функция, кот транслируется в TlgOut – готовит текст – передаем ее в ачестве параметра

немного уменьщаем нагрузку на сервер

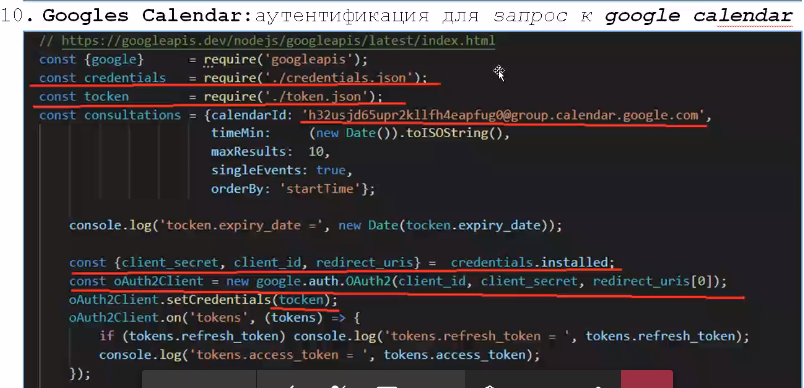
функция *msg* – асинх : выполняет switch, кот вып ф-ю code (беерт поступающий текст и генер ответ, например text=’конс’, отправляем ‘консультации : ’)

   
*ConsGet* – формирует строку и вып запрос к гуглу для считывания календаря

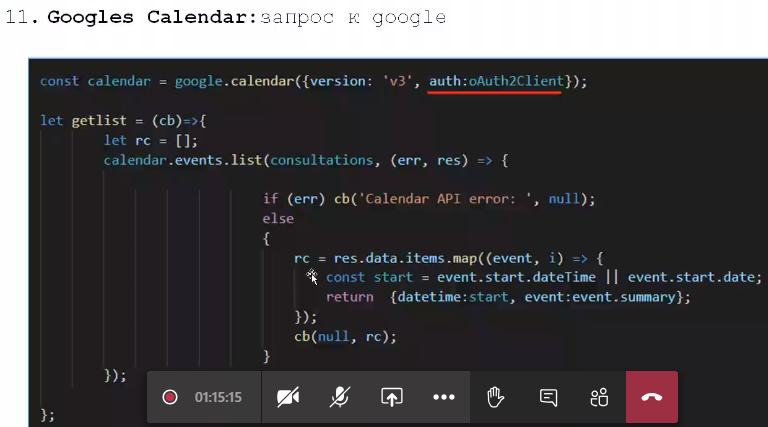
КАЛЕНДАРЬ ГУГЛ:

будут считываться Consultations

**Работа с гуглом разбив 2 этапа:**  
\* аутентификация (самая замороченная)  
\* работа с данными календаря



1) скачать пакет googleapis  
2) надо подключить credentials.json  
3) получить token.json – пойти по ссылке из коммента, пройти по коду и потанцевтаь с бубном  
4) в календаре получим ссылку на имя календаря в его свойствах – calendarId  
5) ….

После аутент можно уже делать запросы к календарю:

**------------- НЕМНОЖКО ТЕОРИИ ИЗ ПРОШЛОГО ГОДА ---------------**

**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

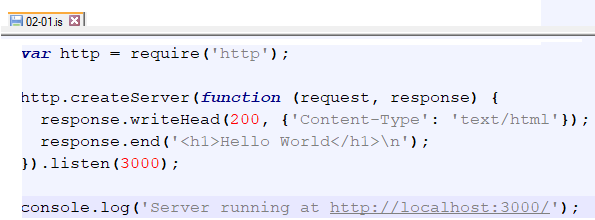
\*имя модуля – строка, м.включ.символы идент.

\*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

***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.

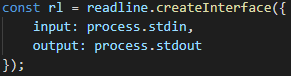
хранит инфу о среде вып.

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

**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):

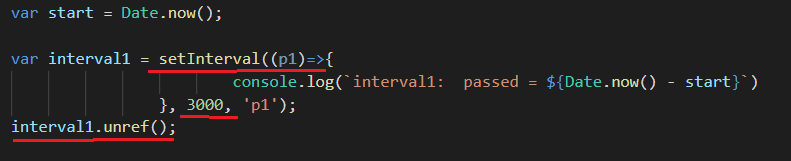
1) время дб точное, замеры не тормоз.

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():**

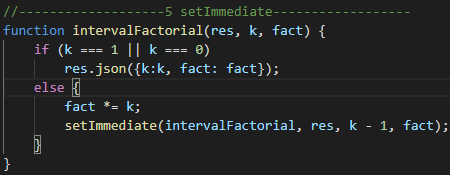
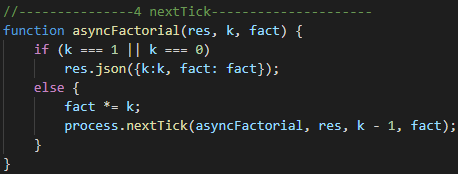
***Event Loop (цикл событий)*** – позв. вып. неблокир. оп-ции в(ы)вода путем выгрузки оп-ций в ядро ОС, когда это возможно. в каждой вкладке браузера свой event loop

**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

**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;

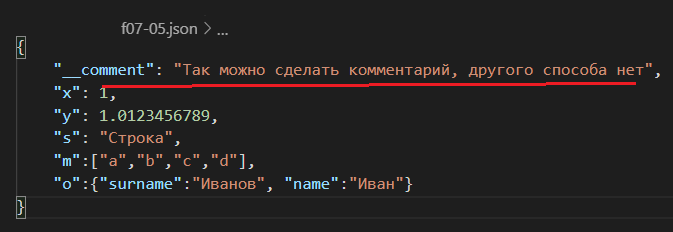
Ф-ция **export** исп.для экспорта ф-ций, объектов или примитивов из файла (модуля)

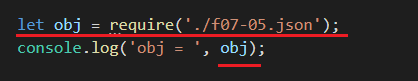
Экспорт 2 видов:

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

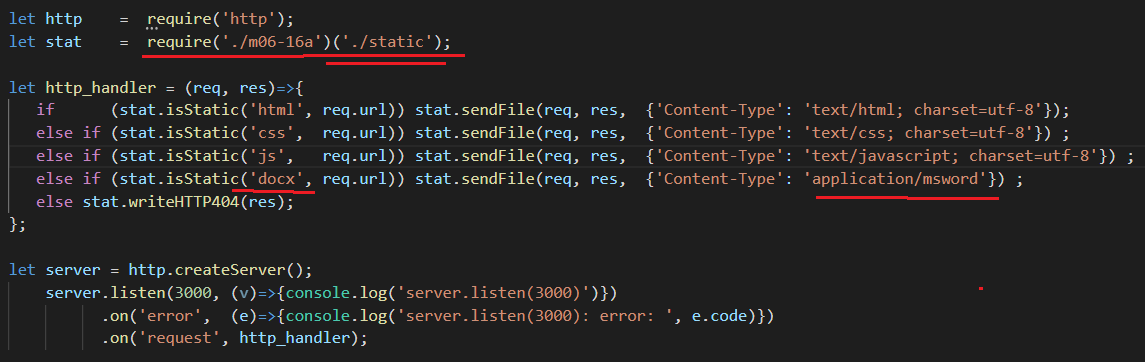
**Именованный** экспорт в модуле:

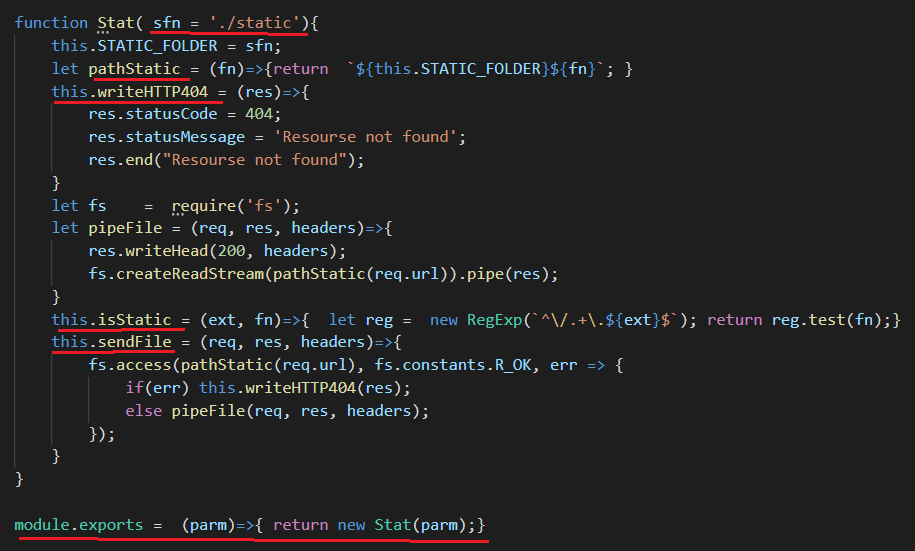
**application/json(rfc 4627)**

****

****

**Параметризированный модуль**

****

****

**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.;

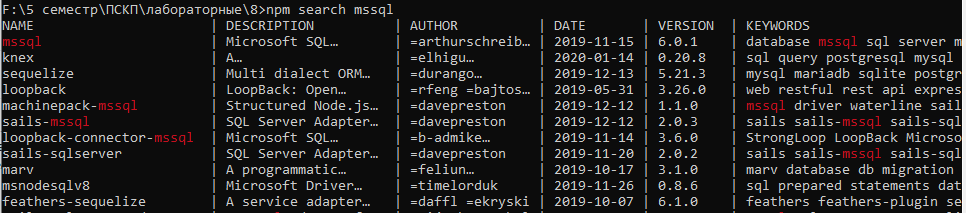
Люб.директория, в кот.есть этот файл интерпретир.как **node.js пакет**, даже если не собираемся публиковать его.

**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 |

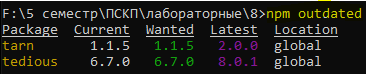
**Обновление пакетов:**

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

---------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

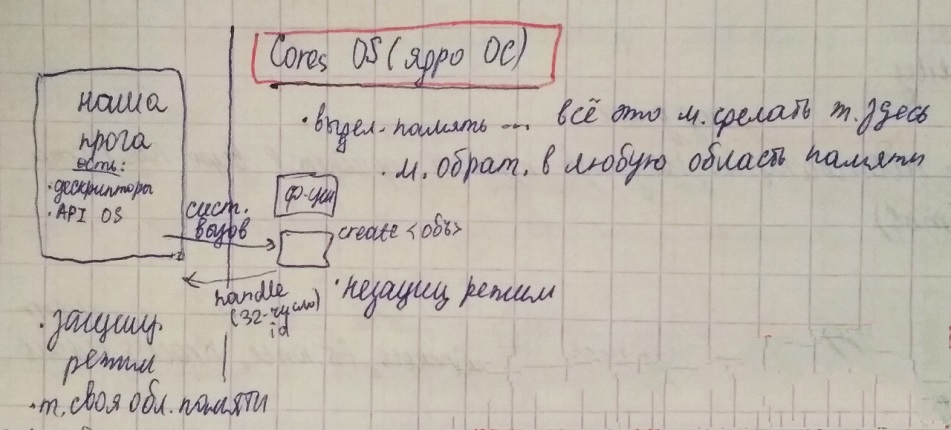
fs.watch(..) следить за ф. (если измен. дата изм, эта callback-ф. вызовется)

*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

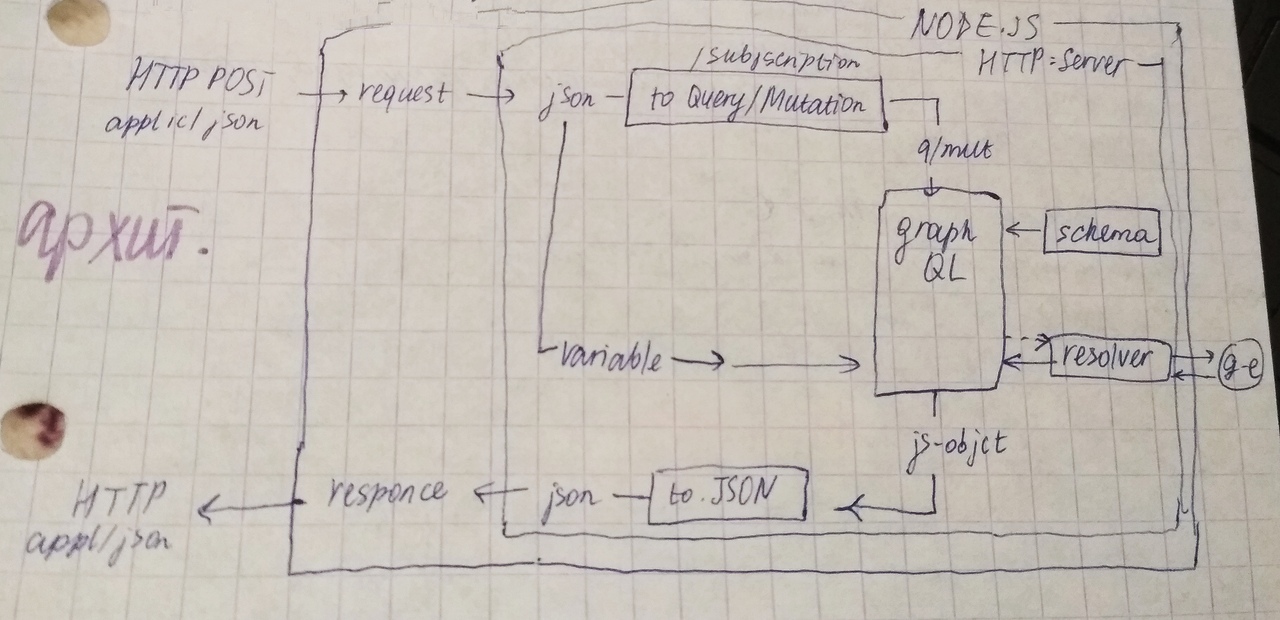
GraphQL – позв. несоклько упорядочить работу с REST-сервисами, пока нет стандартов

**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

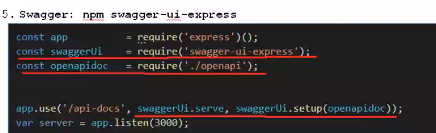
Мы даем схему прогеру, он ее получает по GET запросу и по ней делает запросы.

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

# **Swagger**

**Swagger** – технология, позв. эффективно описывать сервис  
спецификация для описания веб api – ориентирована на rest  
формат для описания веб-апи интерфейса мб использован json или yaml

цели:  
1) отделить процесс проектирования от процесса разработки = м. описать сам web-api интерфейс  
2) м. получить документ, кот. формируется на базе этого описания  
3) м. сгенерировать код на основе этого описания – экономит время



исп. пакет, кот. ориентирован на свагер-описание и его отображение sll? страницы  
здесь генерирую описание интерфейса  
во-вторых его отображение  
пакет openapi – ориентирован только на отображение – здесь используется описание на версии спецификации (кто автор, версия, серверы…)

# **Кеширование**

**Прилож.высокой доступности** – о сайтах, кот д работать 24/7

\* не допускают остановки

\* дб всегда доступно (платежн.сист., без кот.нельзя жть; 21век)

Проблемы с производительностью += проблемы с обновлением

Способы обновления не останавливая:

\* поднимать несколько инстансов одного С, все обновл.по очереди

\* сделать карусель DNS: одному имени неск ip адресов, он будет разбрасывать их по очереди

\* использовать балансировщики нагрузки – спец.прил, когда приходит запрос, он отвечает за карусель + для к ip указ.частоту. с кот.он м обраб.запро

аналогичный механизм есть в IIS  
(м. обновить версию, м. сделать переустановку новую публикацию и у нас будут постепенно Клиенты переходить на новую версию. IIS будет держать неск инстансов (старые), но вновь подключившиеся клиенты будут переходить на новый инстанс -> постепенно все обновлятся)

Если произошел полный крах системы, в техническом листе указано время восстановления, за которое мы можем восстановить работу сервера.

**Время восстановления** – время восстановления из бэкапа всех данных и запуск сервера заново.

## **Как измерить производительность?**

\* произвести нагрузочное тестирование.

**Нагрузочное тестирование** - тестирование, которое проводится для того, чтобы оценить работоспособность приложения под заданной ожидаемой нагрузкой.

Этой нагрузкой мб, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих заданное число итераций за интервал времени.

В нагрузочном тестировании нагрузка идет ступенчато, т.е. пользователи добавляются постепенно (сначала 10, +10, +10 и так м. добавлять до 1000 и т.д.).

Задача – посмотреть с какого момента (количества пользователей) система начинает деградировать.

\*

разбиваем запросы по трудоемкости их исполнения; для каждой из них известна частота этих запросов  
теперь мы д. изготовить смесь запросов (1000 шт), в кот. кол-во запросов в каждой категории д. соотв. частоте: категория А (10%) – их делаем 100 шт….  
их все перемешать в случ. порядке  
и запустить и измерить за сколько t С исполнит эту 1000 запросов  
повысили произв => аналог измерения

## **Способы повышения производительности:**

\* закешировать

**Кэширование** – перенос данных из памяти с низкой скоростью дотсупа в более скоростную для повышения производительности.

**Кэширование** – процессы записи и извлечения данных в кэш.

**Кэш** – серверный объект, кот.предназн.для врем.хранения д-х с целью ускорения вып-ния запроса.

*(д-е м устаревать: копия в кеше мб более старая чем в реальности)*

## **Кэширование на стороне сервера**

### **2 вида кэширования**

\*кэширование данных (не говорим об этом в данном курсе)

(кэширование часто используемых данных)

\*кэширование вывода

(кэширование объекта Response)

### **1) Как сделать кэширование на системном уровне?**

Проще всего закешировать Response с помощью **middleware**

(Написать use и перехватить его в этом use)

### **2) Что нужно указать при записи в кэш, чтобы он был извлекаемым?**

Делают ключ, кот.сост.из метода, uri, время кеширования + имя и значения нек.пар-ров

(время кеширования = через какое время эти данные устареют)

(устарел/нету ответа = прогоняем к ресурсу, кот.записывает в кэш)

**Ресурс-программный код -** экземпляр приложения, который создается для обработки каждого запроса.

**HTTP-драйвер** преобразует посл-сть битов HTTP-запроса в объект Request

### **3) Методы учета кэширования**

\* абсолютный

(если указали 10 секу = ровно через 10 секунд этот кэш устареет)

\* скользящий

(кэш отсчитывается от последнего времени обращения к кэшу

был запрос – время устаревания подляется на 10 сек и т.д.

т.е. чем больше запросов – тем дальше отодвигается время устаревания)

## **Кэширование на стороне клиента (браузера)**

Кэширование на стороне браузера = управление заголовками.

Говорим браузеру – закэшируй. А будет ли он обязательно это исполнять – не факт.

Надо построить приложение такие образом, чтобы не зависело от того, закэшировал клиент данные или нет.

JS: OpenStorage, IndexedDB



# **Процессы и потоки в Node.js**

node.js однопотоковый, но это не совсем верно  
есть встроенный js-engine (он однопотоковый), кот. интерпретирует наш код на js  
там есть спец. механизм, позв осущ асинхронность, с пом. к-й мы разбиваем прил на куски кода, кот становятся в callback-очередь. Этот процесс разбиения зависит от нас

есть спец. методы (SetTimeout, SetImmediate, nextTick, SetInterval…)  
т.о. мы создаем иллюзию того, что наш С параллельно обслуж много запросов, но это все происх на самом деле в рамках одного потока  
всё это происх в рамках js-engine (V8)

но сам node.js работает в ОС, у каждой ОС свой node.js  
в node.js есть процесс, кот. работает в рамках ОС и никто не мешает этому процессу создавать новые процессы, заниматься мультизадачностью и тд – выполнять функции ядра ОС, кот мы знаем. Для этого у нас есть API ОС

Node.js позв создавать процессы ОС, т.е. мы будем выходить за пределы V8 – Будем обращаться к js, кот будет обращаться к API ОС, кот будет создавать потоки, процессы и т.д.

1) **Процесс ОС** – единица работы ОС (прилож=процесс, но одному прил-ю могут соотв неск процессов), соотв. ккакому-то приложению. Этому процессу ОС выдел ресурсы: ОП, процессорное время, блокир. ресусы ФС и т.д.

**Дискреционная модель безопасности**: всё в ОС имеет владельца (внешние, кот мы создаем в рамках ОС, и внутренние (сисадмин…), в контексте безопасности к-рых работает процесс)

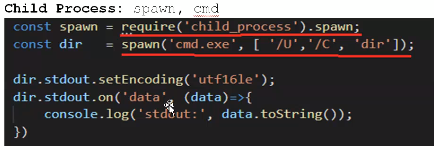
2) **Поток** – средство, механизм диспетчеризации *процессора* (=»железный интерпретатор» - железяка, кот м. исполнять поток к-д, там есть спец регистр, в кот. указ адрес след. команды) в ОС

**ДОЧЕРНИЕ ПРОЦЕССЫ**

когда у нас создается процесс, то ОС выделяет 3 станд *потока ввода-вывода* (stdin, stdout, stderr)  
внутри самого процесса создается *main-поток* (исполняемый поток)

обычно эти станд потоки на консоли  
в node.js сущ встроенный пакет – **Child Process**  
там есть много функций:

\* spawn – позв. вызвать программную оболочку shell (напр. shell-cmd) – это функция, кот. позволяет запускать консоль ‘cmd.exe’, он выдает CreateProcess для других прилоежний, кот мы собираемся запускать на этой консоли (внучата?)

эта оболочка не оч работает с русским языком, сама cmd.exe работает с unicode-кодировкой utf16le (le – little entity – для intel-процессоров)  


получ д-е в виде объекта buffer

**fork** – функция, кот делает копию тек. процесса и запускает ее отдельно  
создается новый экземпляр v8

Механизм, кот позв создавать потоки  
Worker  
**worker\_threads** – встроенный пакет node.js, раб с 10 версии, кот позв создавать потоки  
**\_\_filename** – свойство глоб. объекта, где хранится полное имя файла, в кот. находится исполняющийся js-скрипт в тек. момент  
запускать с ключом *- -experimental-worker* (в 10 версии)