# **Sequelize**

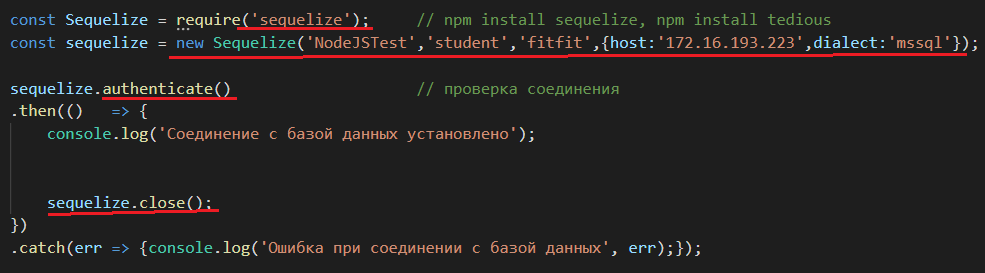
**ORM (Object-Relational Mapping)**  - технология программирования, которая позволяет работать с SQL-базой данных, как с набором программных объектов. **Mapping**: база данных – объект contextDB, таблица – коллекция объектов, строка в таблице – объект, структура таблицы – класс.

**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.

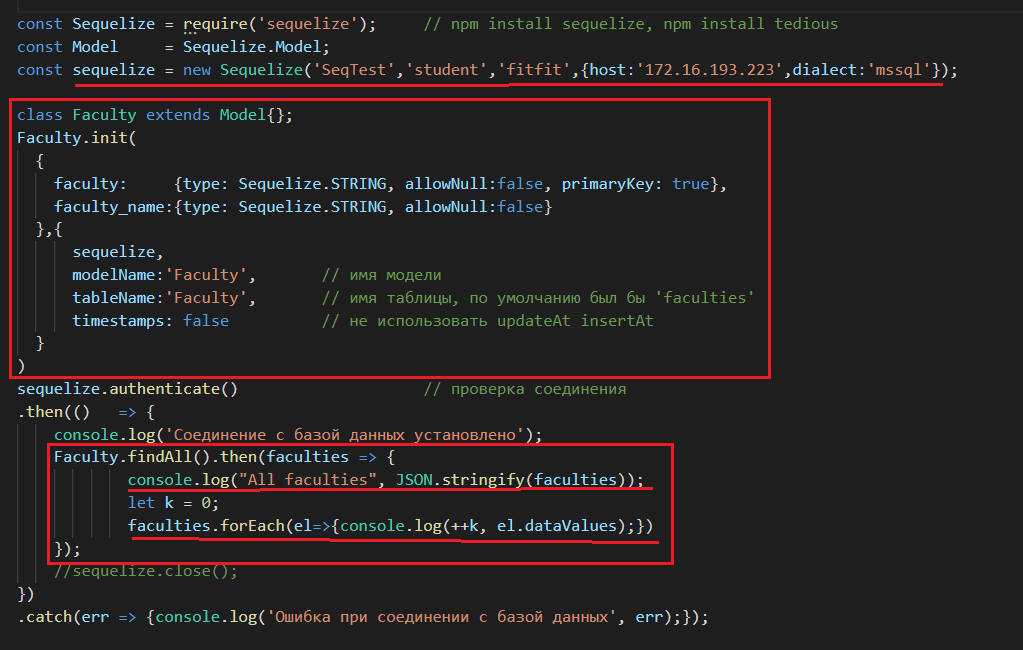
****

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

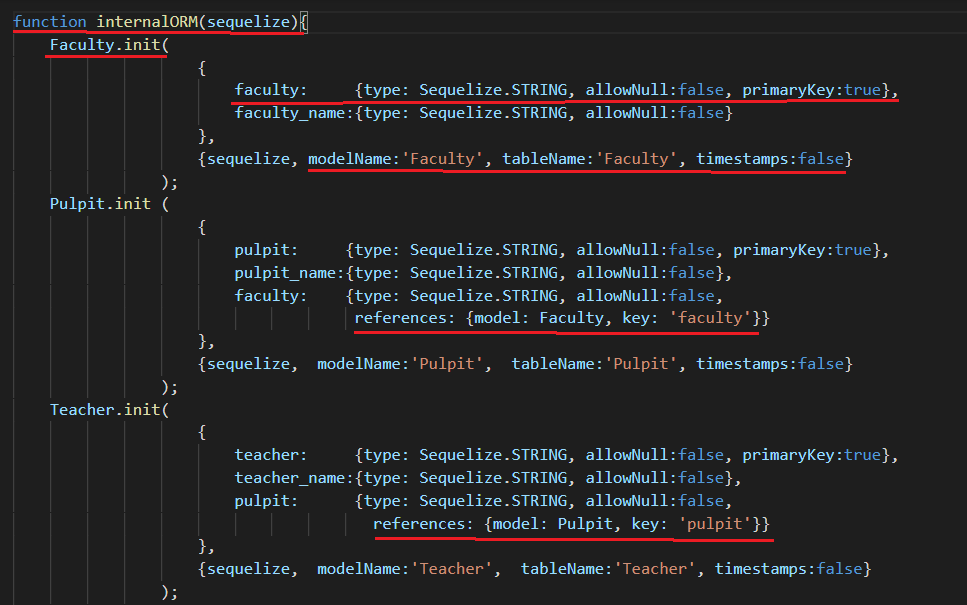
**Tedious** – протокол прикладного уровня, который использует mssql поверх TCP.

3) 1 – стра-ра таблицы, 2 – ук.модель

Timestamps – false: отслеж.измен.в табл(доп.поле)

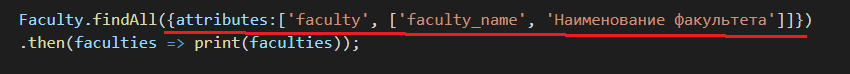
****

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

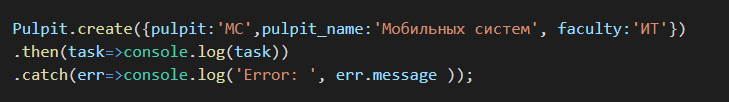
****

**5)** Операции

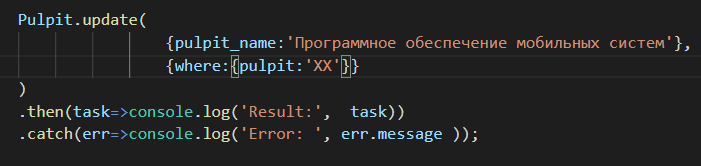
select – findAll()

****

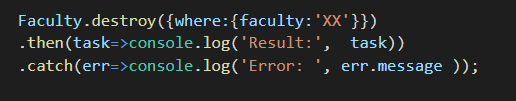
insert – create();

****

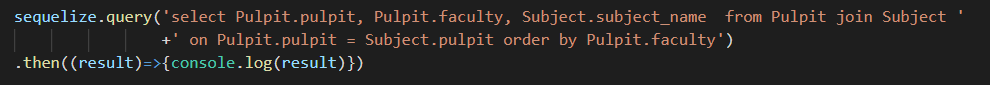
update – update();

****

delete – destroy();

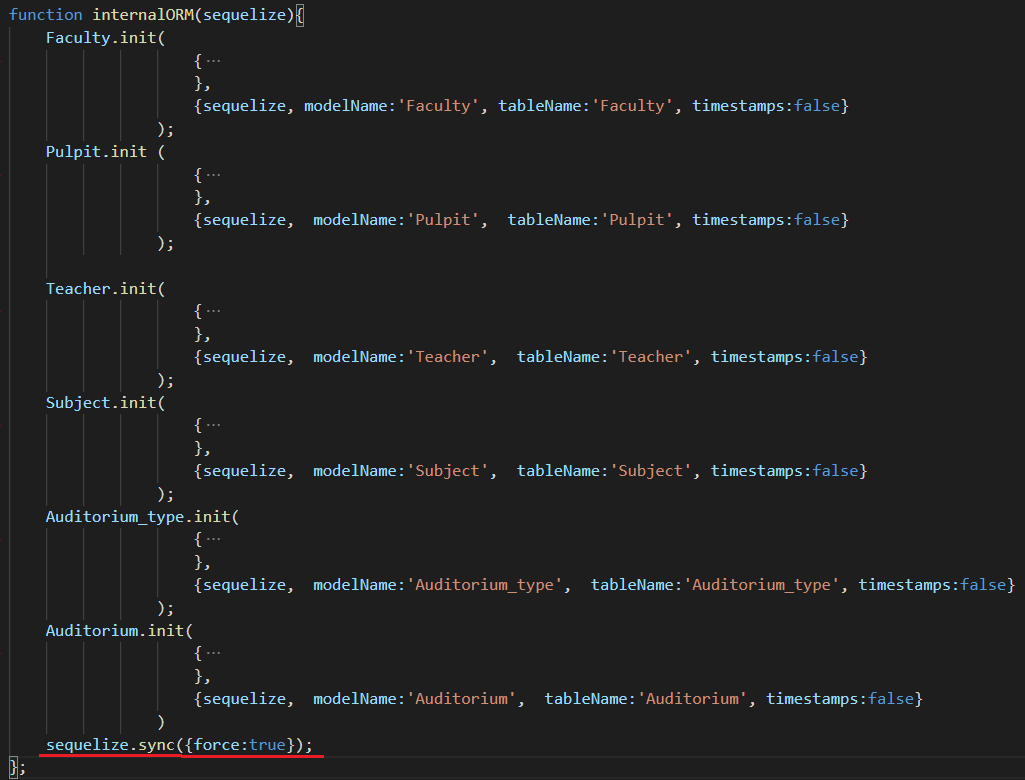
****

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.

****

# **Express**

**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».

**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

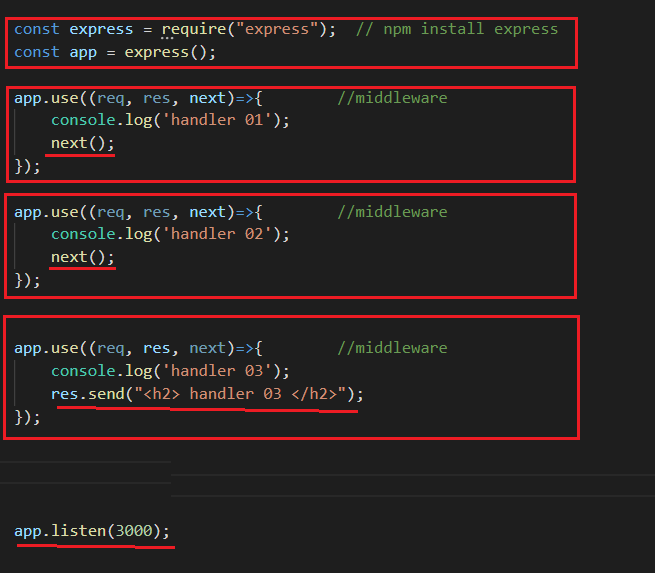
Их физ.послед-сть = порядок обработки запроса от клиента.

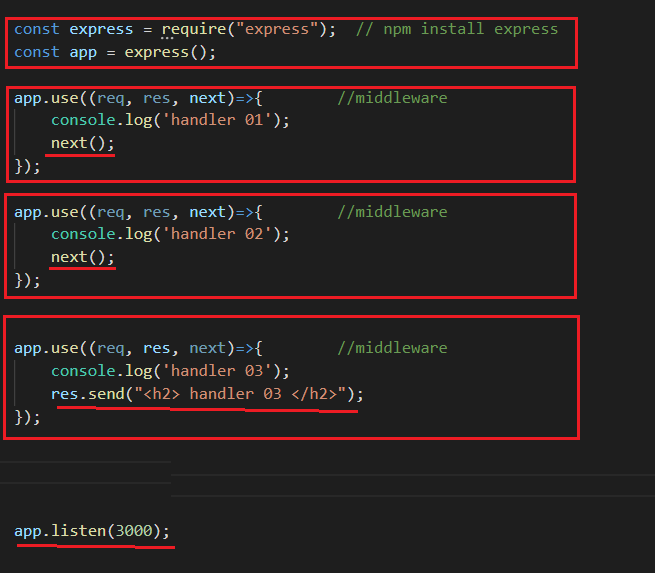
Сначала первый, затем второй и т.д. Все они футболят запрос с помощью **next().**

Последний – обработчик, который формир.ответ.

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

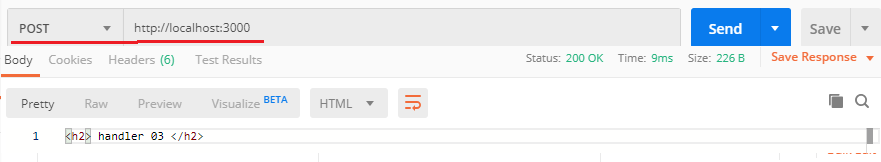
Их методы и св-ва значительно расширены => упрощ.разработку сервера.



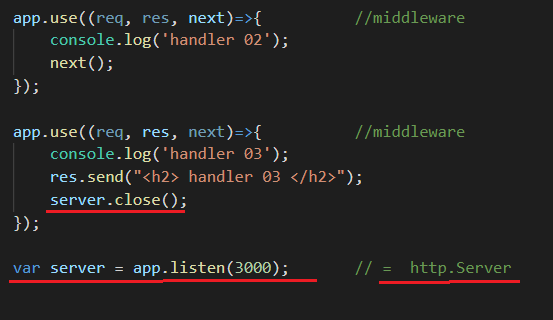


Спом. **app.use()** мб обработаны любые запросы (get, post, put) – не надо указывать в коде что за запрос.

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

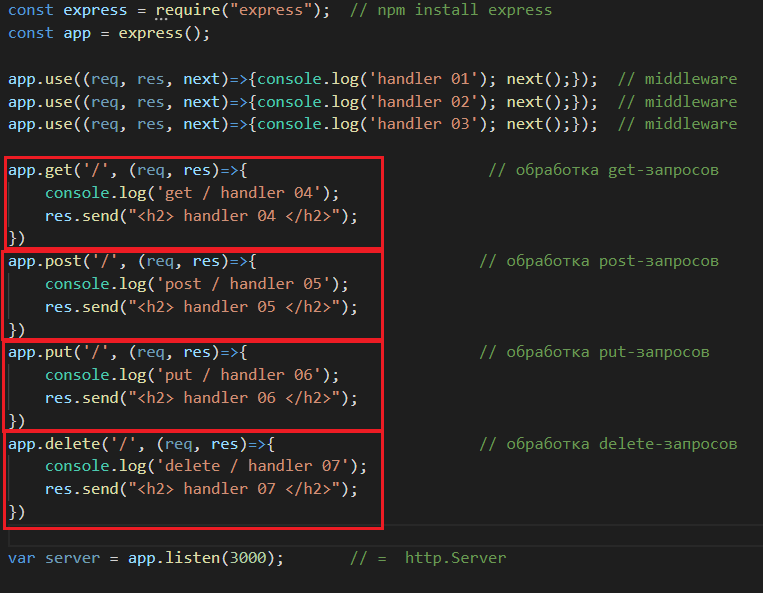
****

1. **Express:** server**.**close

****

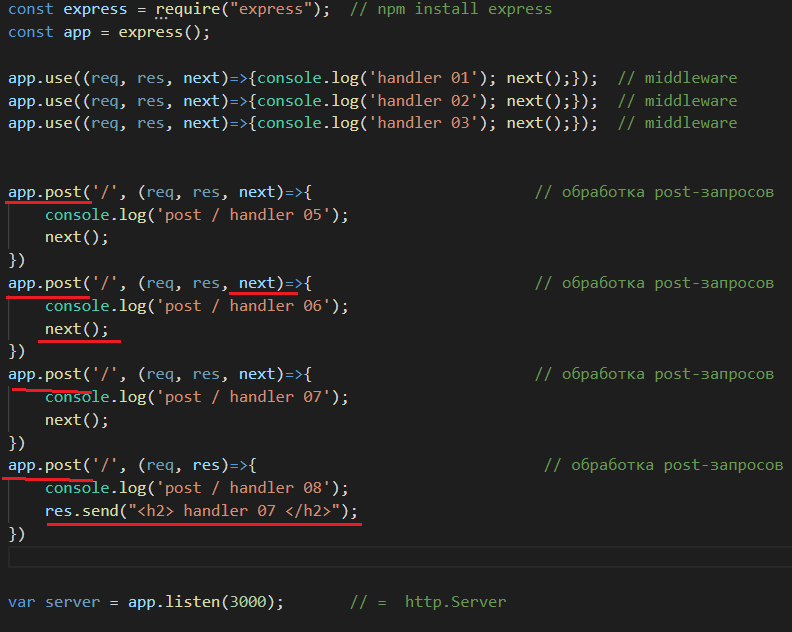
1. **Express:** routing, get/post/put/delete-response

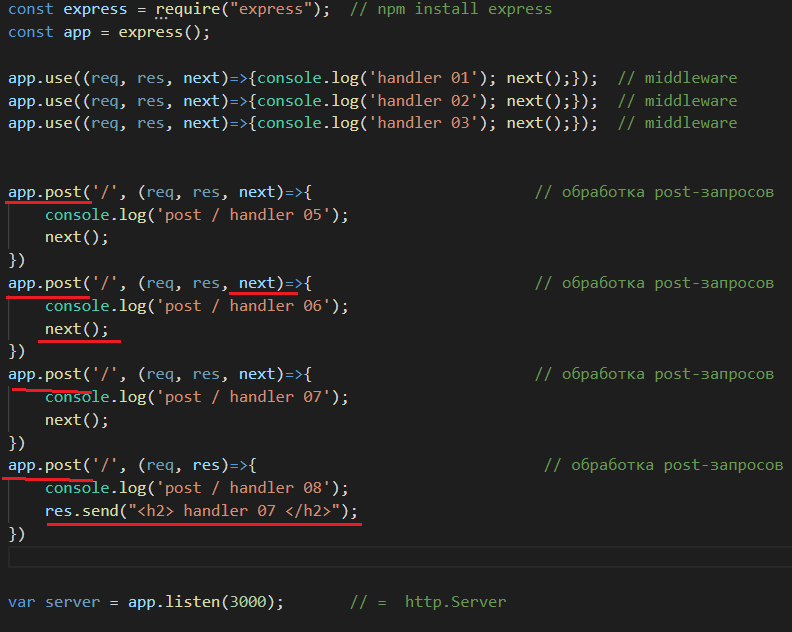
Для разделения по методам. Указ.2 пар-ра, но м.ип-ть 3 пар-р(next)

****

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

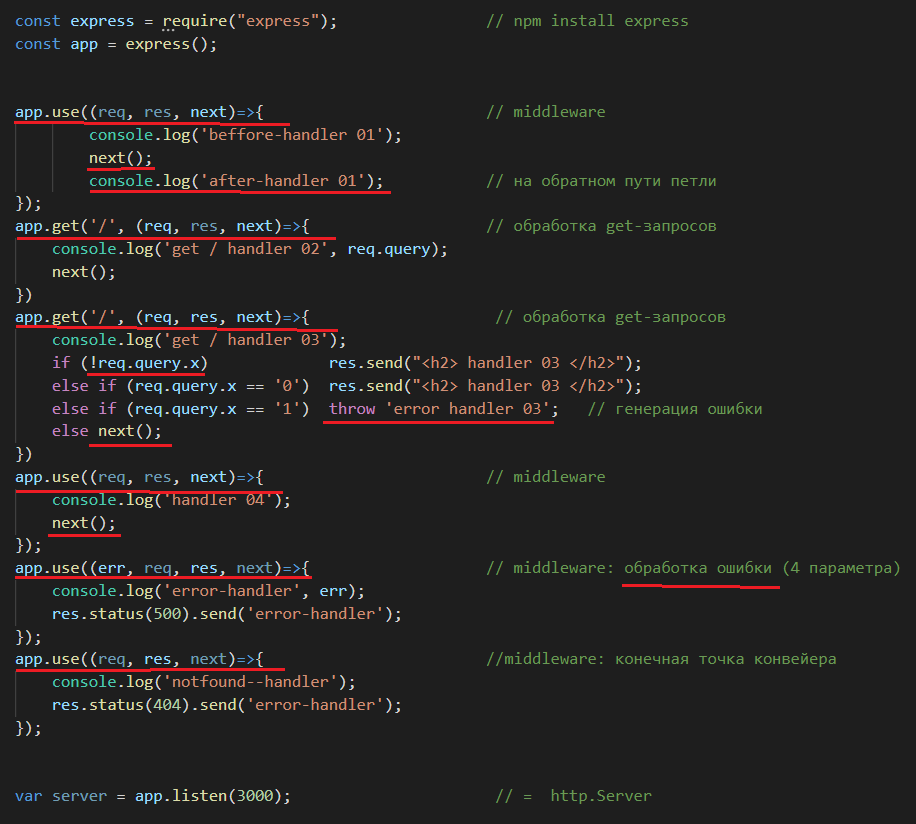
Тож самое, что и вверху, только с 3 пар-ром.

****

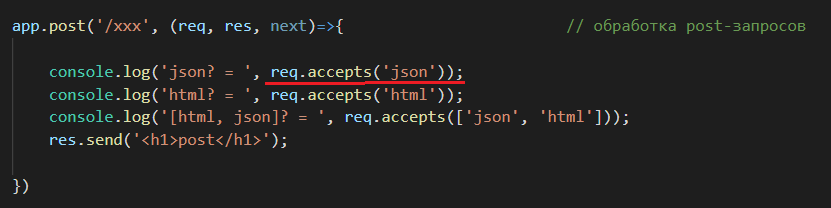
****

1. **Express:** конвейер

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

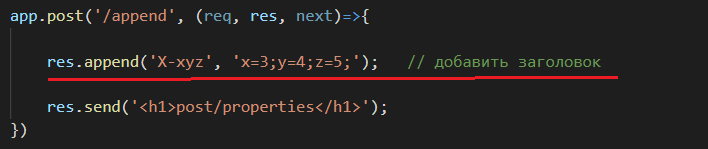
****

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

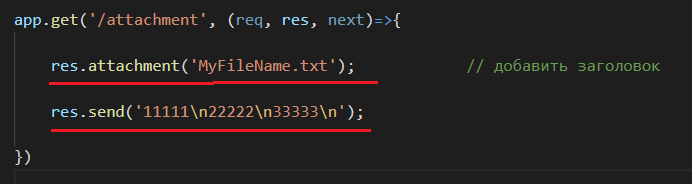
****

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

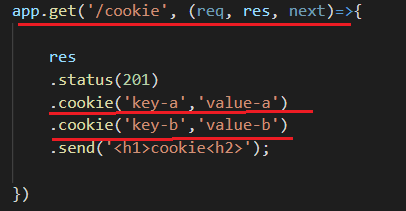
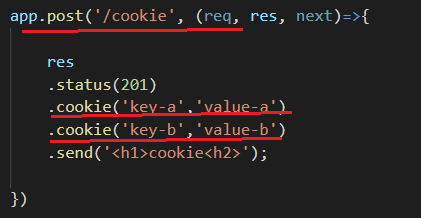
**Append** – добавить заголовок

****

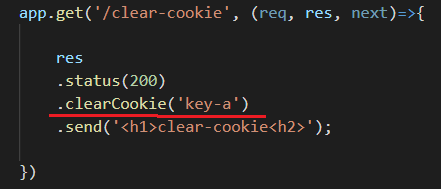
**Attachment –** добавить какой-то файл

****

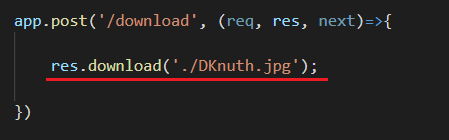
**Cookie** – добавить куки в response посмотреть куки в браузере (applic в консоли)

****

**Clear-cookie -** Почистить куки

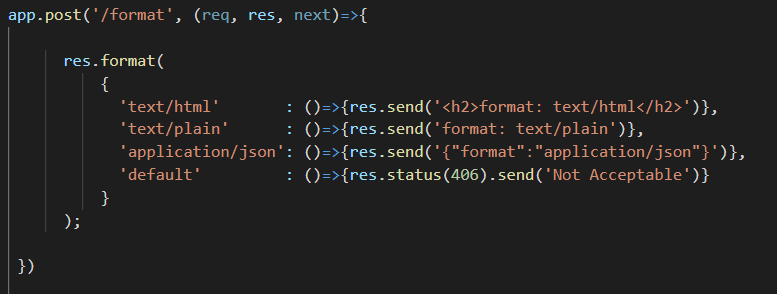
****

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

****

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

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

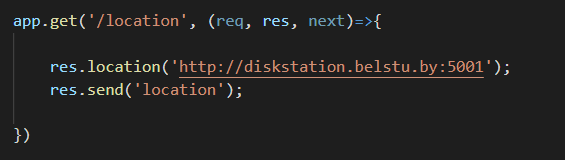
****

**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)

****

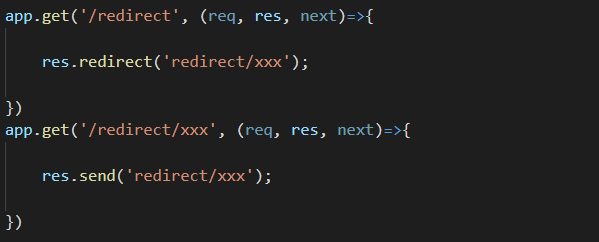
**Redirect –** переадресация (3хх)

308 – такой же запрос, как был первичный

303-305 – get Запросы

308 – новый запрос дб post запросом

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

****

**ИТОГИ:**

запрос, по конвееру с пом next, идет до узла конвеера где есть отправка сообщения  
обратно пошел респонс

до некта по прямому пути идет запрос  
после некта в обр. сторону

экпресс это труба, м. ее делить на неск секций: по юри, по методу и т.д.

хороший тон мидлуэр приложение (кот. можно указать views и оно страивает его в конвеер, где его обрабатывают)

м. монтировать маршруты (Router) для них созд маршруты и обработчики  
«я подмонтирую эти маршруты (Router1) к х/1, х/2 и т.д.»  
создал группу маруртов и их подмонтирую под другой маршрут  
м. динамически менять маршруты

**query-параметры:** через знак ? и &

**URI параметры:**   
внутри uri  
написать общий маршрут черзе параметризированный маршрут часто надо  
напр будут обрабатываться только чето-там там с 3 частями, первая api  
Остальные мимо

**внутри body** параметры образуются если есть тег forms (надо при методе post)  
структура самой строки где хранятся параметры, такая же как в query

**emptype** ( : form) – определяет структуру тела запроса. это означает что в теле запроса будут нах. параметры формы  
middleware пакет body parser – обесп в свойстве body request-а объект, у оторого имена соотв. именам параметов

POST-запрос обработка JSON  
json встроен в express

download и attachment – одно и то же  
когда мы вып download, в браузере запускается  
надо обеспечить тег disposition и там значение attachment, кот. опускает его вних  
если просто в тело запиаем файл, браузер его не будет опускать вниз

тег attachment обесп ф-ю download (делать пересылку на сторону клиента)

upoad только с пом multipart: form-data

cookie исп. для сохранения сост  
пакет cookie-parser – м распарсить кукис

сессия – для сохранения сост  
к серверу серия запросов  
для каждого свое соедиение  
для каждой серии надо сохр индивид. инфу о подключении  
(http протокол stateless, надо как-то это исправить  
сессия предназначена для серии запросов  
серия запросов определяется инфой в cookie  
нет параметра session timeout (макс t между двумя запросами. превышено – сессия разршу)

# **Шаблонизация**

**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)

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

npm install express-handlebars

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

**Partial view** – м.вставл. фрагменты нашего кода в какую-то статическую разметку.

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

# **HTTP Authentication**

**Идентификация** – заявление пользователя о себе.

**Аутентификация** – процедура проверки подлинности идентификации юзера.

**Авторизация** – процедура проверки прав аутентифицированного пользователя.

**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

\*www-autherticate

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

Схема аутентификации (клиент и сервер)

**Способы аутентификации:**

1. BASIC

Схема

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

Схема

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)

Схема

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.

**TLS – аутентификация**

Аутентификация, которая осуществляется на основе сертификата x.509

**x.509** – стандартный формат хранения и транспортироваки отрибутов 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.

Картинка

Вместе с сертификатом выдается секретный и публичный ключ.

Публичный находится на самом сертификате.

Секретный ключ выдается отдельно.

.

# **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)

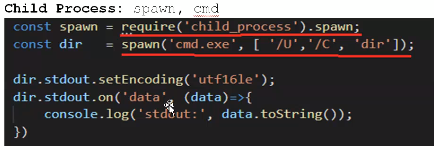
2) **Поток** – средство, механизм диспетчеризации *процессора* (=»железный интерпретатор» - железяка, кот м. исполнять поток к-д, там есть спец регистр, в кот. указ адрес след. команды) в ОС

**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)

# **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

*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.

# **WASM**

**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)

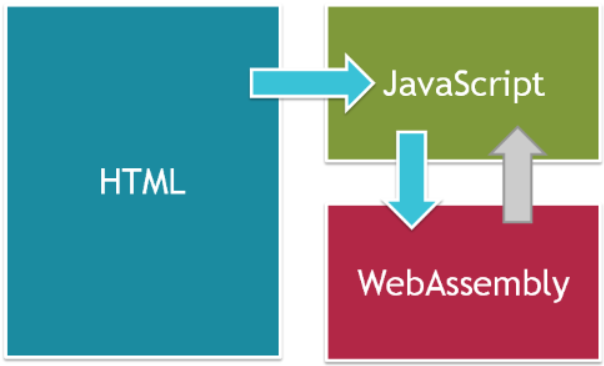
**WasmFiddle** – инструмент, который позволяет проверить как работает wasm.

* 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.

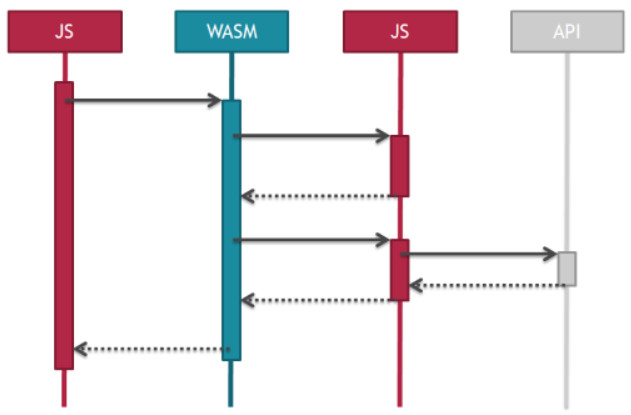
**WASM/Browser – исполнение WebAssembly в браузере:**

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.

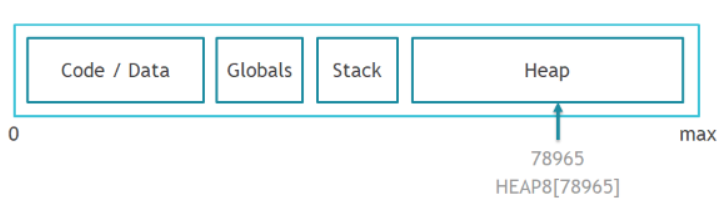
****

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.

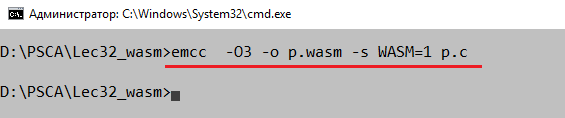
***Byte-код*** можно получить **2 способами:**

* 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

# **API Telegram and Google**

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

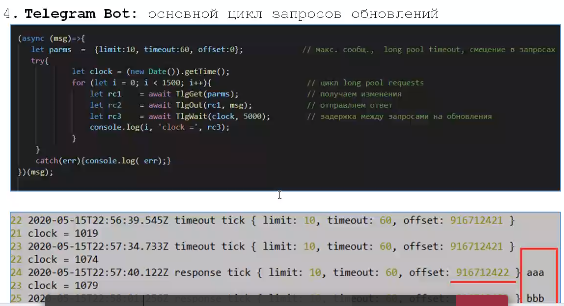
2 общих подхода для создания бота (web hook ; long pool запрос)  
бот отправляет данные от собеседника к серверу, кот мы разработали

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

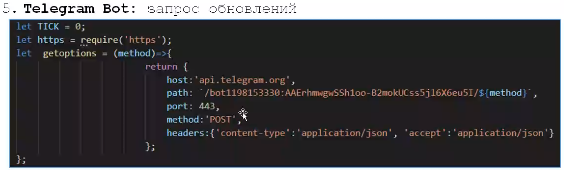
**2 вариант (мы рассмотрим): LONG POOL запрос**это работа в обр. сторону  
1) ест К, кот периодично обращ к веб-апи телеграма  
скачивает и получает те команды, кот введены юзером  
если никто ничего не вводит, Сервер зависает на заданное системное время  
т.е. телеграм ждет это время поступления сообщений  
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

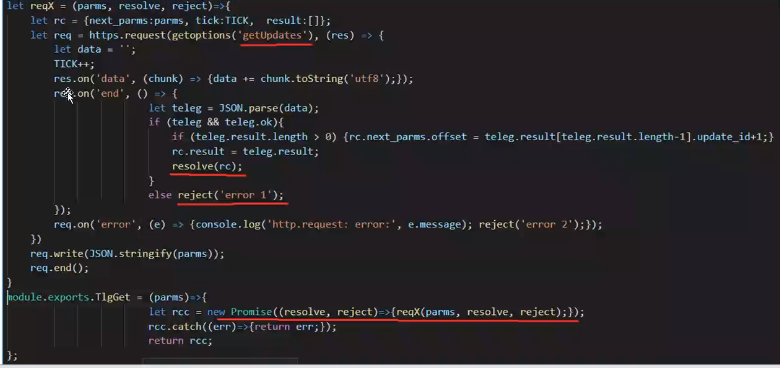
**лонг пул** – отпр запрос, тот подвисает на заднный таймаут, телеграм это время ждет поступления новых запрсово. Если за это время поступило неск запросов, то он нам их отдает в одном овтете

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

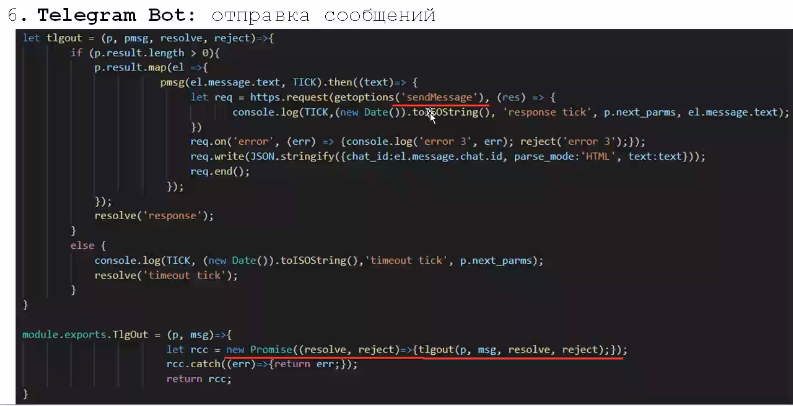
  
получаем ссылку после того как пропишемся в телеграме, это хост по к-му мы моем обращаться чтобы работать с ботом  
тут еще указ метод – что я хочу от телеграма  
будем рассматривать цикл считывания *изменений* (то что мы вводим)



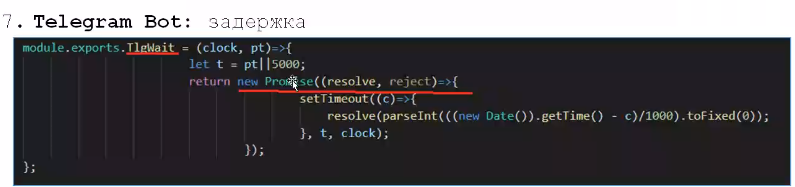
**Promise** – мех, кот позв в более удобном виде записать асинх функцию  
объект позв применять конструкции .then, .catch – позв избавиться от сильной вложенности js-функций. Он обрабаывает либо *resolve* либо *reject* (что-то не так отработало)

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

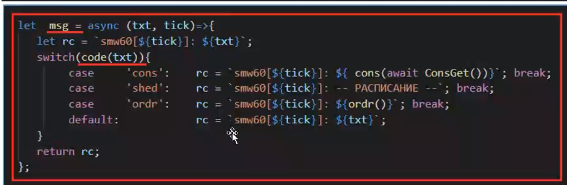
2 ЧАСТЬ ОТПРАВКА ОТВЕТОВ

отправить сообщений телеграм

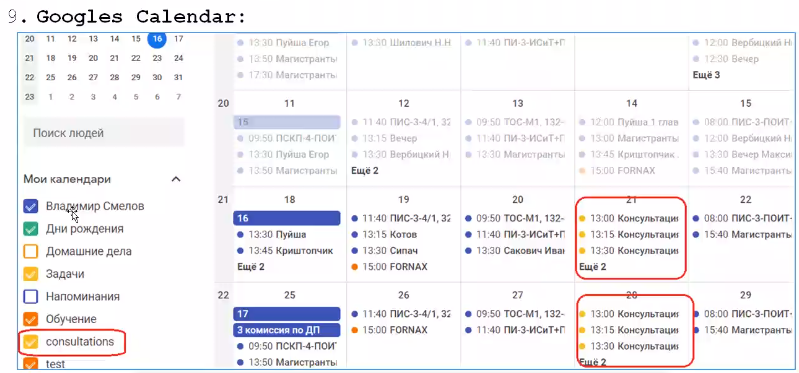
конкретному юзеру, в конкр чат на конкр запрос  
p – результат пред. TlgGet – если они есть, начиню их все обрабатывать  
в конце возвр *resolve (‘response’)* – все в порядке  
если ниче нет, то выдаю *resolve(‘timout tick’)*   
*pmsg* – функция, кот транслируется в TlgOut – готовит текст – передаем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

немного уменьщаем нагрузку на сервер

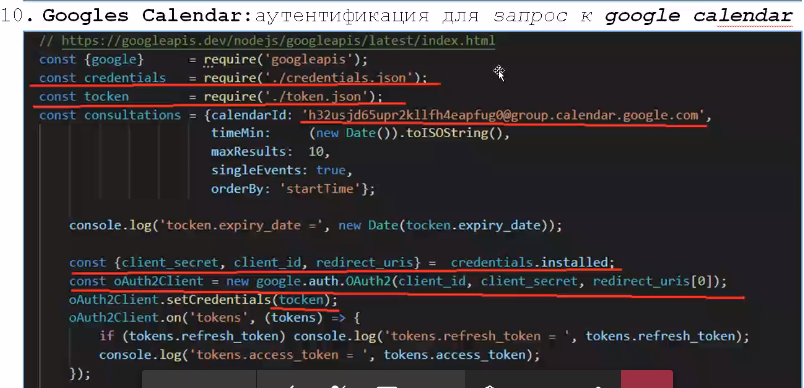
функция *msg* – асинх : выполняет switch, кот вып ф-ю code (беерт поступающий текст и генер ответ, например text=’конс’, отправляем ‘консультации : ’)

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

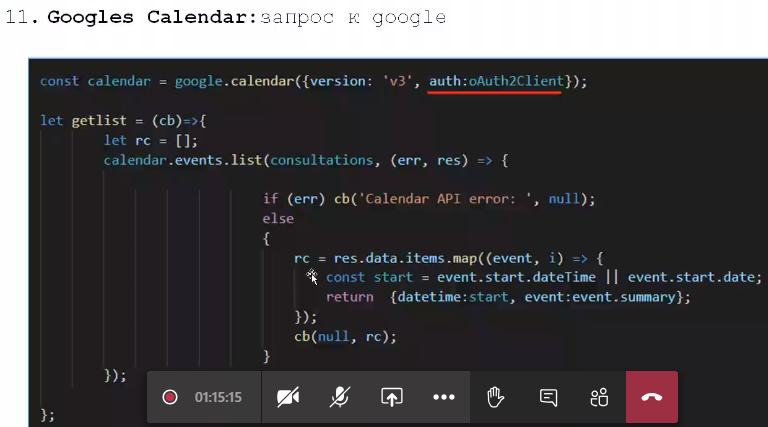
КАЛЕНДАРЬ ГУГЛ:

будут считываться Consultations

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



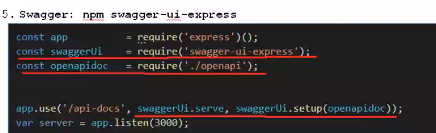
1) скачать пакет googleapis  
2) надо подключить credentials.json  
3) получить token.json – пойти по ссылке из коммента, пройти по коду и потанцевтаь с бубном  
4) в календаре получим ссылку на имя календаря в его свойствах – calendarId  
5) ….

После аутент можно уже делать запросы к календарю:

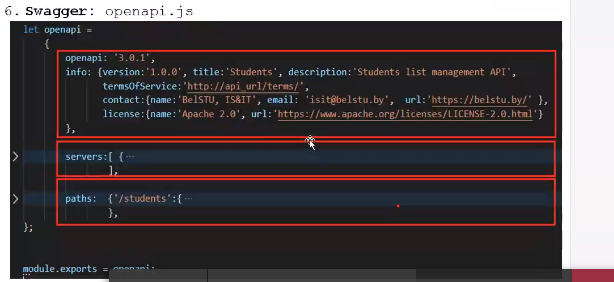
# **Swagger**

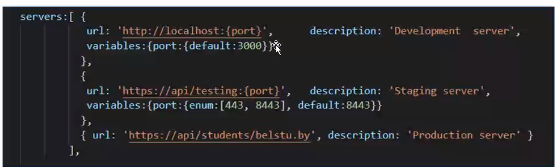
**Swagger** – технология, позв. эффективно описывать сервис  
спецификация для описания веб api – ориентирована на rest  
формат для описания веб-апи интерфейса мб использован json или yaml

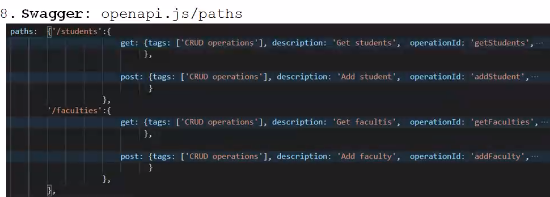
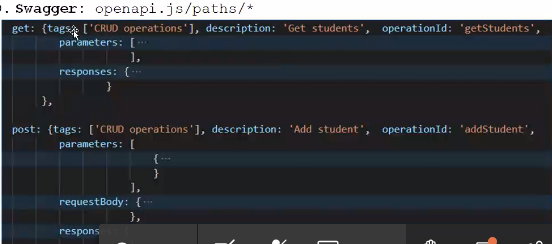
цели:  
1) отделить процесс проектирования от процесса разработки = м. описать сам web-api интерфейс  
2) м. получить документ, кот. формируется на базе этого описания  
3) м. сгенерировать код на основе этого описания – экономит время



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…)



  
описание серверов кот мы можем применять для расположения нашего сервиса

**HTTPS**

позв аутентифицировать сервер, 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 плючом  
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**

***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…)