**1-ый вопрос**

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

**HTTP-протокол** – протокол передачи данных прикладного уровня, ассиметричный (сообщения от клиента к серверу и от сервера к клиенту разные). Протокол на каждый запрос должен получить один ответ, если на 1 запрос – 2 ответа – ошибка. Всегда подразумевает сессвоего состояния. В запросе и ответе нет никаких ссылок на предыдущий и последующий ответ и запрос. Каждый запрос-ответ – новый жизненный цикл HTTP (stateless протокол)

**HTTP:** основные свойства

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

**Request**:

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

**Response:**

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

****









**PathParam -** часть **URL**

**rest/customers/getCustomers/ABCD123** - где  ABCD123  - посылаемый параметр (строковый id).

**QueryParam -** часть самого запроса

**/rest/customers/getCustomers?placeId=PAD1234C&placeId=90321**

**Код сост. Response:**

\* 1хх: информ. сообщение

\* 2хх: успешный ответ

\* 3хх: переадресация

\* 4хх: ошибка клиента (404)

\* 5хх: ошибка сервера

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

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

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

**Веб-приложение** – приложение клиент-серверной архитектуры, где клиент и сервер общаются посредством использования протокола HTTP.

**Структура:**

1. 1 клиент – 1 сервер

2. 1 клиент – несколько серверов

3. Несколько клиентов – 1 сервер

4. Клиенты - промежуточный сервер – сервер

**Особенности реализации web-приложения:**

* При проведении работ, которые требуют остановку веб сервера, необходимо иметь запасной, на который можно перенаправить трафик (**отказоустойчивость**)
* Т.к. к веб-серверу происходит множество запросов от различных пользователей, то возможности ответа веб-сервера упираются в пропускную способность канала и, чтобы не загружать этот канал слишком сильно, используются максимально короткие сеансы подключения и кэширование ответов на стороне клиента, а кэширование на стороне сервера позволяет быстрее сформировать ответ. (**Быстродействие**)
* Использование https для устранения возможности злоумышленникам подделывать запросы/ответы и просматривать их. (**Шифрование**)

**Асинхронное программирование** — концепция программирования, которая заключается в том, что результат выполнения функции доступен не сразу же, а через некоторое время в виде некоторого асинхронного (нарушающего обычный порядок выполнения) вызова.

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

Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы или блоки кода.

1. **Платформа Node.js, версии, назначение, основные свойства, структура, принципы работы, основные встроенные модули и их назначение, применение внешних модулей (пакетов). Форматы модулей.**

**NODEJS:** программная платформа для разработки серверных web-приложений на языке JS/V8.

- первая версия: **2009 г**.;

- стабильные версии: **с 2015 г., Node.js 4.0.0**;

- версионирование: две ветки **16.x.x** – версии длительной поддержки (LST, Long Term Support), **18.x.x** – нестабильные версии, включающие последние разработки (Current)

- основная сфера применения: разработка web-серверов;

**NODEJS: основные свойства:**

- основан на **Chrome V8**;

- среда (контейнер) исполнения приложений на JavaScript;

- поддерживает механизм **асинхронности**;

- ориентирован на события;

- **однопоточный** (код приложения исполняется только в одном потоке, один стек вызовов); обычно в серверах для каждого соединения создается свой поток, в Node.js все соединения обрабатываются в одном JS-потоке;

- не блокирует выполнение кода при вводе/выводе (в файловой системе до 4х одновременно);

- в состав Node.js входят инструменты: **npm** – пакетный менеджер; **gyp** - Python-генератор проектов; **gtest** – Google фреймворк для тестирования С++ приложений;

- использует библиотеки: **V8** – библиотека V8 Engine, **libuv** – библиотека для абстрагирования неблокирующих операций ввода/вывода; **http-parser** – легковесный парсер http-сообщений (написан на C и не выполняет никаких системных вызовов); **c-ares** - библиотека для работы с DNS; **OpenSSL** – библиотека для криптографии; **zlib** – сжатие и распаковка.

**Node.js использует модульную систему.** То есть вся встроенная функциональность разбита на отдельные пакеты или модули. Модуль представляет блок кода, который может использоваться повторно в других модулях.

**Основные встроенные модули:** fs (работа с фаловой системой), http (http сервер), url (работа с url), util (доступ к служебным функциям)

В отличие от встроенных модулей для подключения своих модулей надо передать в функцию **require** относительный путь с именем файла. Чтобы какие переменные или функции модуля были доступны, необходимо определить их в объекте **module.exports**. Объект module.exports - это то, что возвращает функция require() при получении модуля.

Вообще объект module представляет ссылку на текущий модуль, а его свойство exports определяет все свойства и методы модуля, которые могут быть экспортированы и использованы в других модулях.

принцип работы node js:

CommonJS использует ключевое слово exports для экспорта и require для импорта, а ES6-модули используют export для экспорта и import для импорта



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 (req)

\*www-authenticate (res) – там схема аут (basic, digest)



**Способы аутентификации:**

BASIC



Basic аутентификация самая простая, тк данные пользователя передаются в открытом виде (Base64 строка).

При попытке доступа к закрытому ресурсу пользователь получает ответ с кодом 401 и заголовком WWW-Authenticate Basic realm=”Some name”

Тут Basic – имя способа авторизации

Realm – имя закрытого ресурса

Тк нам приходит код 401, то у нас открывается диалоговое окно с полями ввода имени пользователя и пароля.

Потом после ввода с клиентской части идет ответ с заголовком Authorization : Basic Base64String

Здесь опять таки идет пометка способа авторизации и зашифрованные логин и пароль пользователя в Base64

Если данные валидны, то сервер отвечает ответом с кодом 200 иначе процесс повторяется заново.

Дайджест - DIGEST



При попытке доступа к закрытому ресурсу пользователю приходит ответ от сервера с кодом 401, что открывает окно ввода пароля и логина, так же содержится заголовок WWW-Authenticate : Digest realm, qop, nonce, opaque, algorithm, stale

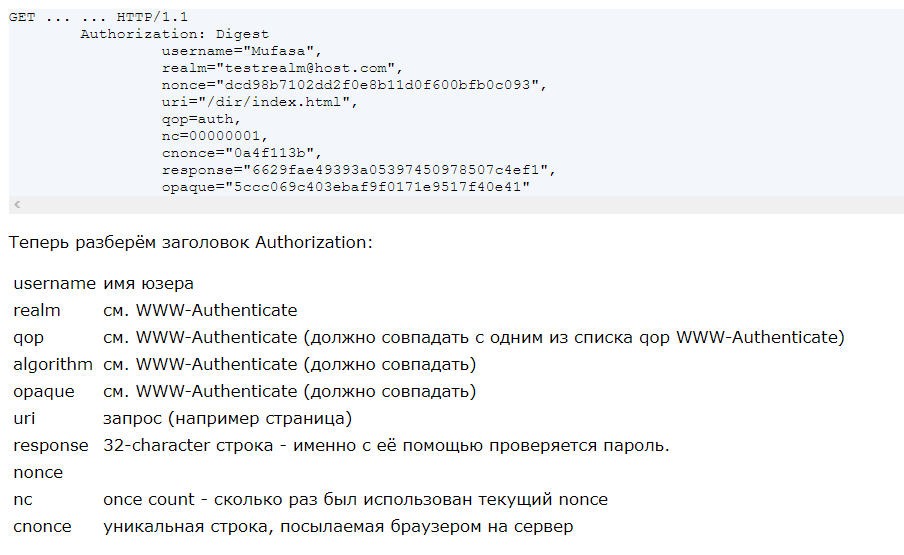
* Realm – информация о ресурсе где вводится пароль
* Qop – флаг указывающий quality of protection
* Nonce – Уникальная строка, которая генерируется на сервере
* Opaque – Строка, которую пользователь должен вернуть без изменения в ответе
* Algorithm – алгоритм исп для шифрования
* Stale – указывает был ли прошлый запрос правильным или содержал ошибки

Далее клиентская часть после ввода пароля и логина вычисляет зашифрованную строку.

Дале идет несколько этапов вычисления хэша по алгоритму из заголовка ответа:

* H1 = MD5(username:realm:password)
* H2 = MD5(method:digestURI)
* Response = MD5(H1:nonce:nonceCount:cnonce:qop:H2)

Ответ пользователя выглядит следующим образом :



Далее сервер производит вычисления аналогичные выше и сравнивает хэши, если совпадают, то приходит ответ 200 ОК

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 тюдю

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

**Процесс установки соединения:**

**TLS 1.2**

* **Client hello(client)** – клиент начинает общение с сервером. Отправляет набор поддерживаемых шифронаборов (CipherSpec) и случайное простое число нужное для генерации общего ключа симметричного шифрования.

**CipherSpec** – какой алгоритм обмена ключами будет использоваться (RSA, DH, ECDH …), какой алгоритм использовать для шифрования данных (AES, 3DES, …) и какая криптографическая функция будет использоваться для генерации хэша MAC(Message Authentication Code).

* **Server Hello(server)** – сервер отвечает выбранной версией протокола TLS, выбранный набор шифров и случайное простое число.
* **Certificate(server)** – сервер отправляет сертификат, а клиент проверяет подпись CA, проверку доверия к CA, Проверку домена, срок действия и не был ли сертификат отозван.
* **Server Key Exchange(server)** – этот шаг происходит не всегда, а в зависимости от выбранного набора шифров. Если выбран к примеру RSA, то этот шаг пропускается. Если выбран DH, то на этом этапе сервер отправляет еще простое число q, число g и вычисленное число y.
* **Server Hello Done(server)** – сервер оповещает, что данный начальный этап установки соединения завершен.
* **Client Key Exchange(client)** – не обязательный этап, если к примеру используется DH, то клиент передает свое число Y. Оба вычисляют число K, которое используется для создания pre-master-key. На основании случайного числа клиента, сервера и pre-master-key функция выдает симметричный ключ и ключ вычисления MAC
* **Change Cipher Spec(client)** – Клиент оповещает сервер, что он готов перейти на защищенное соединение.
* **Finished(client)** – клиент зашифровывает симметричным ключом первое сообщение с MAC.
* **Change Cipher Spec(server)** – Сервер проверяет сообщение Finished от клиента и отправляет в ответ свою готовность перейти на защищенное соединение.
* **Finished (server)** – отправляет тестовое зашифрованное сообщение

**TLS 1.3**

* **Client Hello** – Состоит из открытого ключа, полученного по протоколу DH, Предполагаемый режим обмена ключами, Модель алгоритма цифровой подписи.
* **Server Finished** – Часть открытого ключа, Сертификат, так же сервер вычисляет сеансовый ключ
* **Client Finished** – Когда у клиента есть 2 ключа, то он вычисляет свою копию сеансового ключа, перед этим проверив сертификат.

**Создание сертификата**

**Первый шаг** – создание закрытого ключа. Это делается с использованием openssl и следующей команды:

**openssl genrsa -out server.key 2048**

genrsa – указывает на то, что мы генерируем RSA ключ

2048 – сложность (2048 бит)

**Второй шаг** – создание запроса сертификата

**openssl req -new -key server.key -out %your\_website%.csr**

req – указывает на то, что мы собираемся генерировать запрос сертификата

new – указываем то, что мы создаем новый запрос

key – указываем откуда считывать приватный ключ

**Последний шаг** – создаем самоподписанный сертификат

**openssl x509 -req -days 365 -in %your\_website%.csr -signkey server.key -out %your\_website%.crt**

x509 – формат сертификата

req – запрос на генерацию сертификата

days – сколько будет действителен сертификат

**Использование сертификата**

На сервере мы при создании сервера передаем объект содержащий ключ и сертификат

Для клиента:

* Добавление в ОС
* Добавление в браузер

Для добавления в ОС:

* Скачать сертификат
* Двойной клик
* Установить сертификат
* Поместить сертификат в следующее хранилище – обзор
* Доверенные корневые центры сертификации

Для браузера:

* Скачать сертификат
* Настройки
* Сертификаты
* Управление сертификатами – центры сертификации – импортировать
* Загрузка сертификата – доверять при идентификации веб-сайтов

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

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

Свойства:

- Дуплекс

- Каналы – потоки

- JSON, XML... форматы передачи данных

- RPC

- Стандарт RFC 6455

Процедура установки соединения Называется handshake

- Клиент отправляет запрос на апгрейд соеднения

- Сервер отправляет ответ со статусом 101 что означает, что протокол передачи данных был изменен.

- Далее клиент и сервер передают данные в дуплексном режиме: В один момент времени клиент и сервер могут как отправлять, так и принимать данные.

Применяется websocket в чатах, играх, управление IoT

WebSocket API

Api для создания, управления вебсокет-подключеним к серверу и для отправки и получения сообщений.

Атрибуты:

- onopen – соединение установлено

- onclose – соединение закрыто

- onmessage – получено сообщение

- onerror – ошибка

- readystate – текущее состояние подключения

- url

Константы:

ReadyState

- connecting – соединение еще не открыто

- open – соединение открыто

- closing – соединение в процессе закрытия

-closed – соединение закрыто

Методы:

- close – закрывает WS-подключение

- send – передача данных через WS-соединение

1. **Применение СУБД Redis. Основные принципы работы. Пример (лабораторная работа).**

**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 (подключение, объявление моделей, конфигурация, подход Code First, поисковые методы, фильтрация, пагинация, сортировка, группировка).**

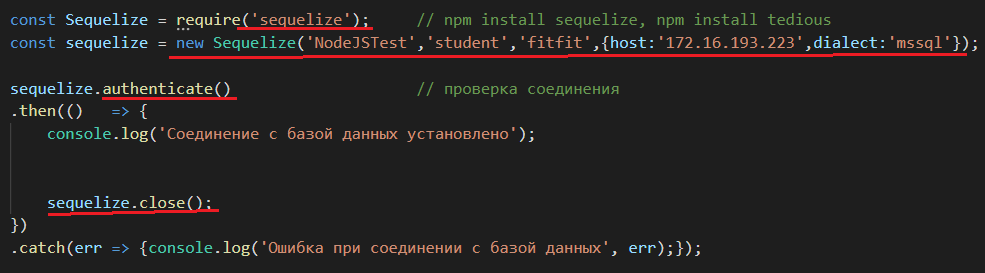
**ORM (Object-Relational Mapping)** - технология программирования, которая позволяет работать с SQL-базой данных, как с набором программных объектов. **Mapping**: база данных – объект contextDB, таблица – коллекция объектов, строка в таблице – объект, структура таблицы – класс.

**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.

****

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

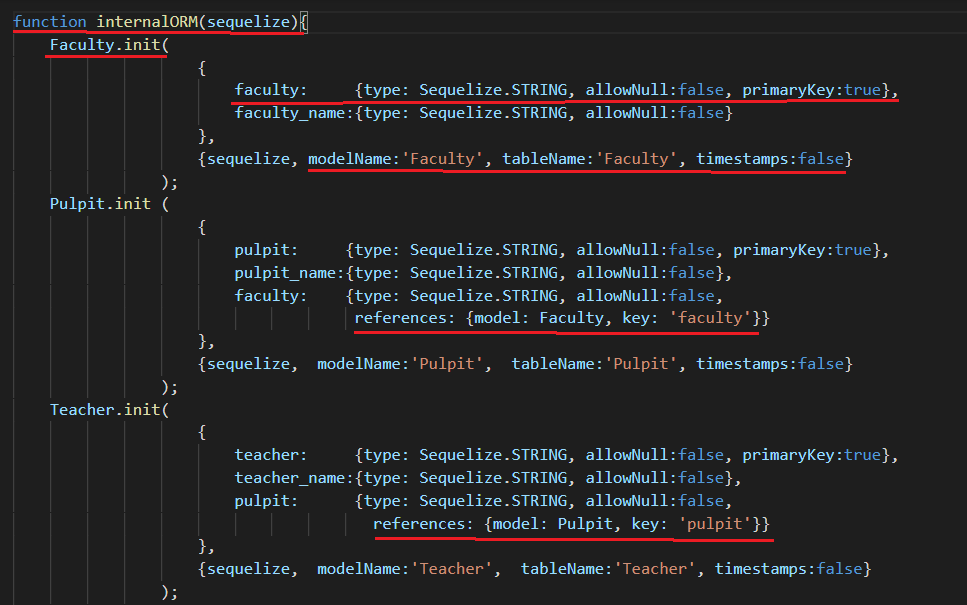
**Объявление модели:**

Model.init(),

Sequelize.define()

Через sequelize-auto()

reference – указать внешний ключ.

****

Еще можно через модель.belongsTo(модель2) + модель2.hasMany(модель) и указывать у полей foreignKey: true, primaryKey: true

**Конфигурация:**

New Sequelize(‘имя бд’, ‘логин, ‘пароль’, {хост, диалект(субд), доп параметры})

**Code first:**

Sync()

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

**Поисковые методы:**

findAll()

findByPk()

findOne()

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

Query()

join: через include

right outer join: include :[{………right: true}]

left outer join: include :[{………required: false}]

order by: order: [[‘field’,’[desc/asc]’ ]]

group by: в аттрибутах атомарная функция, потом group

фильтрация : where:{}

пагинация: offset: , limit

1. **Применение пакета Sequelize. Основные принципы работы (ассоциации, добавление, изменение, удаление строк в таблице, raw query).**

**Ассоциации:**

Ассоциации в Sequelize позволяют связывать таблицы и модели между собой. Существует несколько типов ассоциаций: hasOne, hasMany, belongsTo, и belongsToMany. Давайте рассмотрим пример связи "один-ко-многим" (hasMany) между двуми моделями: User и Post.

**Добавление:**

Create()

bulkCreate()

**Изменение:**

Update({{djjd: ididi},{where: {..} }})

bulkUpdate()

еще можно использовать findByPK():

const user = await User.findByPk(1);

user.username = 'UpdatedUsername';

await **user.save();**

**Удаление строк в таблице:**

Destroy();

**Raw query:**

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

Query()

1. **Применение пакета Sequelize. Основные принципы работы (scopes, хуки, соединения, ассоциации, транзакции).**

**Scopes:**

наборы условий, которые могут быть предварительно определены для моделей Sequelize.

Они используются для создания часто используемых фильтров и запросов к данным. Скопы позволяют определить именованные наборы условий, которые могут быть повторно использованы в запросах, без необходимости повторного описания этих условий каждый раз.

User.addScope('active', {

where: { isActive: true },

});

const activeUsers = await User.scope('active').findAll();

можно применить к find, findAll, count, increment, destroy

**хуки:**

функции, которые позволяют выполнять определенные действия или операции перед или после выполнения операций CRUD (Create, Read, Update, Delete) с данными в моделях Sequelize. Они позволяют внедрить пользовательский код и логику в определенные точки жизненного цикла модели.

Локальные:

Model.addHook(), Model.removeHook()

Можно сразу в модели определить но они могут переопределить глобальные хуки

Глобальные:

Сразу в подключении или sequelize.addHook()

beforeCreate

afterCreate

beforeBulkCreate

afterBulkCreate

beforeUpdate

afterUpdate

beforeDestroy

afterDestroy

beforeBulkDestroy

afterBulkDestroy

beforeValidate

afterValidate

beforeSave

afterSave

beforeUpsert

afterUpsert

beforeFind

afterFind

beforeBulkUpdate

afterBulkUpdate

beforeBulkDestroy

afterBulkDestroy

**Ассоциации :** прошлый вопрос

**Транзакции:**

Unmanaged:

Transaction({isolationLevel….})

t.commit()

t.rollback()

managed:

Transaction(t=>{})

autoCallback, autocommit/ autorollback

В ЛЮБОЙ ТРАНЗАКЦИИ В КОНЦЕ ОПЕРАЦИИ УКАЗЫВАЕМ {TRNSACTION: T} НАПРИМЕР

CREATE(…{TRNSACTION: T})

1. Пакет Prisma. Основные характеристики. Способы создания моделей. Основные принципы работы (подключение, схема Prisma (поля, типы, атрибуты, модификаторы, функции атрибутов), поисковые методы (с различными опциями)).

Prisma: это ORM нового поколения с открытым исходным кодом для Node.js и TypeScript. Она состоит из следующих инструментов:

* Prisma Client: автогенерируемый и типобезопасный клиент базы данных;
* Prisma Migrate: система миграций;
* Prisma Studio: пользовательский интерфейс для просмотра и редактирования данных.

Prisma: основные характеристики:

* написан на Rust;
* реализует паттерн Data Mapper;
* поддерживаются MySQL, PostgreSQL, MSSQL и SQLite (поддержка MongoDB, CockroachDB и PlanetScale в предварительной версии);
* предоставляет типобезопасный API (TypeScript);
* имеет набор инструментов для работы с базами данных (отправка запросов, моделирование, миграции, прототипирование, data seeding, студия для просмотра и изменения);
* способен генерировать определение схемы базы данных и клиентский код на основе структуры базы данных;
* поддерживает различные параметры запросов (фильтрация, сортировка, группировка, пагинация и др.);

1. настройка проекта Prisma, database first

Инициализация проекта:

Npx prisma init

Настройка связи с БД:

В .env поменять строку подключения, в schema.prisma поменять провайдера(субд)

Генерация моделей:

npx prisma db pull

Генерация клиента:

Npx prisma generate

+ npx prisma studio

+ npx migrate dev

$disconnect – откл соед с бд

**Схема призма:**

В схеме Prisma модели определяются с использованием ключевого слова model

Типы данных:

Int - целое число.

Float - число с плавающей точкой.

String - строка.

Boolean - булев тип (true/false).

DateTime - дата и время.

Json - JSON-объект.

Decimal - десятичное число.

Enum - перечисляемый тип данных.

Bytes - массив байтов.

Атрибуты и модификаторы:

* **@id**: Указывает, что поле является первичным ключом.
* **@unique**: Гарантирует уникальность значений в поле.
* **@default(value)**: Устанавливает значение по умолчанию при создании новой записи.
* **@updatedAt**: Автоматически обновляет поле времени обновления при изменении записи.
* **@createdAt**: Автоматически устанавливает значение времени создания при создании новой записи.
* **@map(name)**: Позволяет задать другое имя для поля в базе данных.
* **@relation(fields: [...], references: [...]):** Модификатор, который определяет ассоциацию между моделями и указывает поля для связи.
* **@ignore**: Модификатор, который исключает поле из схемы базы данных, но оставляет его в схеме Prisma.
* **@onDelete(action):** Модификатор, который указывает действие, выполняемое при удалении записи, связанной с этим полем.
* **@sql.Raw**

**Поисковые методы:**

findMany() + where

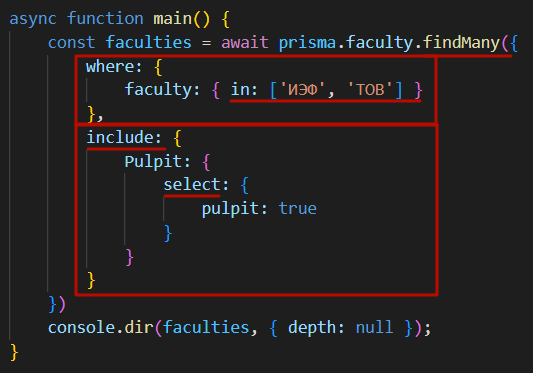
findUnique()

findUniqueOrThrow()

findFirst()

findFirstOrThrow()

+in (там то ли промежуток то ли какие могут быть значения)



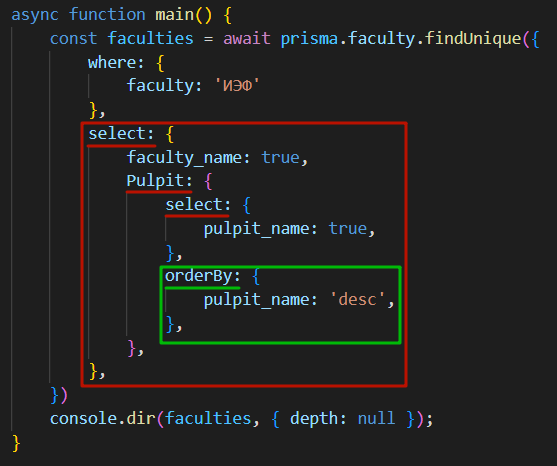
+include

+select (select+select)

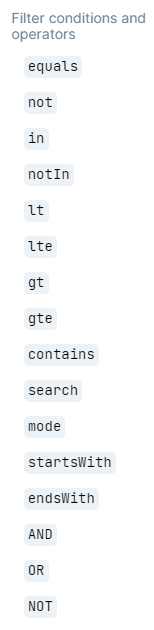
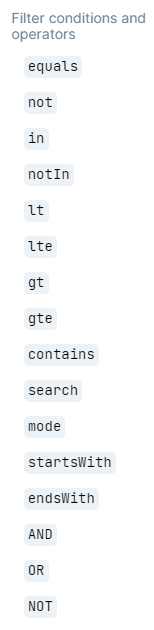
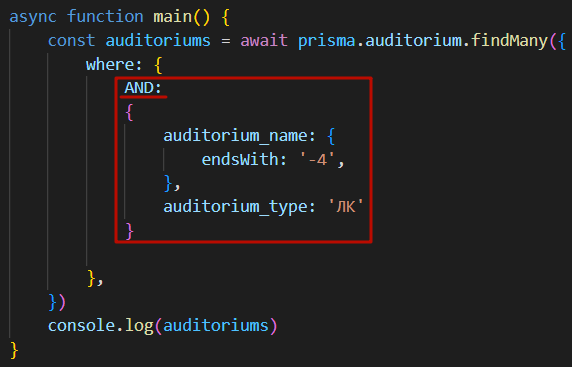
+ их комбинация по разному

+contains – фильтрация как like

+ orderBy



+ where :{AND/OR/NOT}



1. Пакет Prisma. Основные характеристики. Способы создания моделей. Основные принципы работы (подключение, выборка связанных записей, фильтры на связи -ко-многим, -к-одному, Fluent API, пагинация, подсчет записей, агрегирование, группировка, distinct).

выборка связанных записей:

нельзя селект и инклуд на одном уровне

* include (включает связанные записи, join)
* select полей только связанных записей (include => select)
* select полей модели и полей связанных записей (вложенный, select => select, в таком случае include не нужен)

фильтры на связи:

* фильтр на связь "-ко-многим", none (возвращает все записи, в которых ни одна связанная запись не соответствует критериям фильтрации)
* **фильтр на связь "-ко-многим", every** (возвращает все записи, в которых все связанные записи соответствует критериям фильтрации)
* фильтр на связь "-ко-многим", some (возвращает все записи, в которых хотя бы одна связанная запись соответствует критериям фильтрации)
* **фильтр на связь "-к-одному", is** (возвращает все записи, в которых связанная запись соответствует критериям фильтрации)
* фильтр на связь "-к-одному", isNot (возвращает все записи, в которых связанная запись не соответствует критериям фильтрации)

Fluent API - потоковый интерфейс (цепочка методов). Последний вызов функции определяет возвращаемый тип всего запроса (в примере pulpit). Единственное требование для цепочки состоит в том, что предыдущий вызов функции должен возвращать только один объект.

Faculty.findUnique(…).Pulpit()

Пагинация:

Skip: , take:

Подсчет записей:

\_count (подсчет количества записей). Может использоваться внутри include или select верхнего уровня. Кроме того, можно сортировать по количеству связанных записей.

* include => \_count => select => relation (подсчет кол-ва только указанных связанных записей)
* select => \_count: true (подсчет кол-ва всех связанных записей)
* select => relation => select => \_count: true (подсчет кол-ва всех связанных с relation записей)

агрегирование:

* \_count: возвращает количество записей или непустых полей;
* \_avg: возвращает среднее значение всех значений указанного поля;
* \_sum: возвращает сумму всех значений указанного поля;
* \_min: возвращает наименьшее значение указанного поля;
* \_max: возвращает наибольшее значение указанного поля.

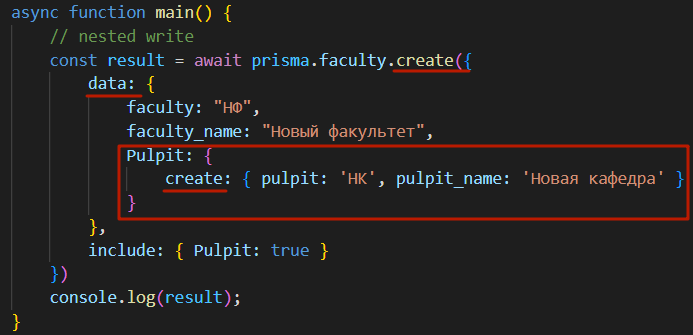
groupBy (группировка записей по одному или нескольким полям)

distinct (возвращает только уникальные значения по одному или нескольким полям)

1. Пакет Prisma. Основные характеристики. Способы создания моделей. Основные принципы работы (подключение, создание записей исходной модели и связанной, присоединение существующих записей, изменение записей исходной модели и связанной).

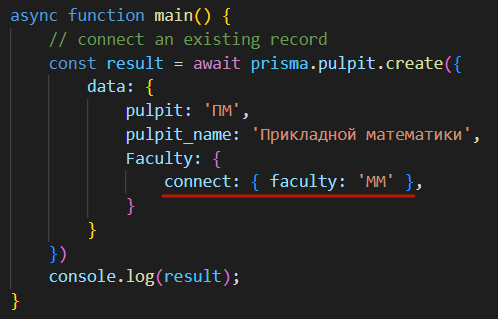
create (создание одной записи)

create, вложенная запись



createMany, вложенная запись(в createMany нельзя вложенную)

create, присоединение существующей записи, в createMany нельзя так сделать



create, присоединение существующей записи или создание новой, если ранее указанной не существует



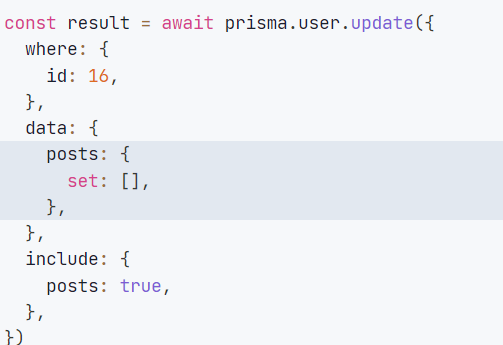
Аналогично с Update

Можно законектить несколько записей сразу

Disconnect



Отконектить все сразу можно используя set с пустым массивом



Еще можно апдейт связанных записей как одну так и несколько (updateMany)

Update or create : upsert





1. Пакет Prisma. Основные характеристики. Способы создания моделей. Основные принципы работы (подключение, схема Prisma (поля, типы, атрибуты, модификаторы, функции атрибутов), удаление записей исходной модели и связанной, транзакции, миграции).

Удаление:

Delete()

deleteMany()

deleteMany({})- удалить все

для каскадного удаления надо в схеме сделать связь опциональной и запихнуть в транзакцию два удаления



Еще можно в атрибутах в схеме указать @onDelete: cascade вроде тоже работает

Еще походу в delete можно запихнуть include чтобы связанные удалить но это не точно нихуя

Транзакция: prisma.$transaction(async(prisma)=>{…})

Ну или как в примере выше

Миграция: npx migrate dev

Откатить npx prisma migrate reset

1. Пакет Express. Основные принципы работы. Маршрутизация, промежуточные обработчики, обработка ошибок.

**Express** – это фреймворк для Node.js, который реализовывает слой функций, необходимых для создания эффективных приложений и API. Его использование значительно сокращает написание кода, а, значит, уменьшается затрачиваемое на разработку время.

**Основные принципы работы**

Для установки фреймворка используется пакетный модуль NPM (npm i express).

Далее подключается модуль в главный файл (const express = require(‘express’))

Создаем приложение с помощью функции const app = express(). Под капотом используется метод createApplication(), который создает объект приложения, делегирует методы обработки и потом возвращает этот объект.

Далее мы можем подключить нужные middleware функции c помощью app.use([middleware]). Пример: app.use(express.json());

Далее мы можем создавать обработчики get, post, put, delete, … . Создаются они с использованием объекта приложения и метода обработки, в качестве параметров передается путь, который будет обрабатываться и callback-функция, которая является логикой обработки. Так же могут передаваться 1 и более middleware функций.

Далее для запуска приложения используется функция объекта app.listen([port]). Так же в функцию может передаваться имя хоста в виде ip-адреса. Данная функция является оберткой над http.createServer.

**Маршрутизация**

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

Самый просто метод определения маршрута является использование методов маршрутизации объекта приложения. Сигнатура методов следующая: app.[Method name](path, callback);

Express поддерживает перечисленные далее методы маршрутизации, соответствующие методам HTTP: get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search и connect.

Так же имеется метод обработки app.all(path, callback) который обрабатывает запросы любого метода пришедшие на [path].

**Пути маршрутов**

Пути маршрутов, в сочетании с методом запроса, определяют конкретные адреса (конечные точки), в которых могут быть созданы запросы. Пути маршрутов могут представлять собой строки, шаблоны строк или регулярные выражения.

**App.route и express.router()**

Метод app.route() позволяет создавать обработчики маршрутов, образующие цепочки, для пути маршрута. Поскольку путь указан в одном расположении, удобно создавать модульные маршруты, чтобы минимизировать избыточность и количество опечаток.



С помощью класса express.Router можно создавать модульные, монтируемые обработчики маршрутов. Экземпляр Router представляет собой комплексную систему промежуточных обработчиков и маршрутизации; по этой причине его часто называют “мини-приложением”.

**Middleware**

Middleware - это функции, имеющие доступ к объекту запроса (req), объекту ответа (res) и к следующей функции промежуточной обработки в цикле “запрос-ответ” приложения. Следующая функция промежуточной обработки, как правило, обозначается переменной next.

Функции промежуточной обработки могут выполнять следующие задачи:

* Выполнение любого кода
* Внесение изменений в объекты запроса о ответа
* Завершение цикла запрос-ответ
* Вызов следующего метода-обработчика из стека

Если текущая функция промежуточной обработки не завершает цикл “запрос-ответ”, она должна вызвать next() для передачи управления следующей функции промежуточной обработки. В противном случае запрос зависнет.

Порядок загрузки промежуточных обработчиков очень важен: функции промежуточных обработчиков, загруженные первыми, выполняются в первую очередь.

Приложение Express может использовать следующие типы промежуточных обработчиков:

**Промежуточный обработчик уровня приложения** – app.get(), app.post(), … .

**Промежуточный обработчик уровня маршрутизатора** – аналогичен уровню приложения, но привязаны к объекту Router

**Промежуточный обработчик для обработки ошибок** – express смотрит на кол-во параметров middleware функции и принимает решение какого типа это middleware. Если функция принимает 4 параметра, то она принимается как middleware обработки ошибок.

**Встроенные промежуточные обработчики** – примеры: express.static(), express.json(), … .

**Промежуточные обработчики сторонних поставщиков ПО**  - к примеру обработчик cookie: app.use(cookieParser()); или деприкейтед body-parser: app.use(bodyParser());

1. Пакет Express. Основные принципы работы. Статические файлы. Отдача статики. Кэширование на стороне клиента.

**Статические файлы**

Для предоставления статических файлов, например, изображений, файлов CSS и JavaScript в Express используется функция промежуточной обработки express.static.

Для того чтобы начать непосредственное предоставление файлов, необходимо передать имя каталога, в котором находятся статические ресурсы, в функцию промежуточной обработки express.static.

app.use(express.static('public'));

Express выполняет поиск файлов относительно статического каталога, поэтому имя статического каталога не является частью URL.

Для использования нескольких каталогов, содержащих статические ресурсы, необходимо вызвать функцию промежуточной обработки express.static несколько раз.

Express выполняет поиск файлов в том порядке, в котором указаны статические каталоги в функции промежуточной обработки express.static.

Для того чтобы создать префикс виртуального пути (то есть, пути, фактически не существующего в файловой системе) для файлов, предоставляемых с помощью функции express.static, необходимо указать путь монтирования для статического каталога.

app.use('/static', express.static('public'));

Тем не менее, путь, переданный в функцию express.static, указан относительно каталога, из которого запускается процесс node. В случае запуска приложения Express из другого каталога, безопаснее использовать абсолютный путь к каталогу для предоставления файлов.

app.use('/static', express.static(\_\_dirname + '/public'));

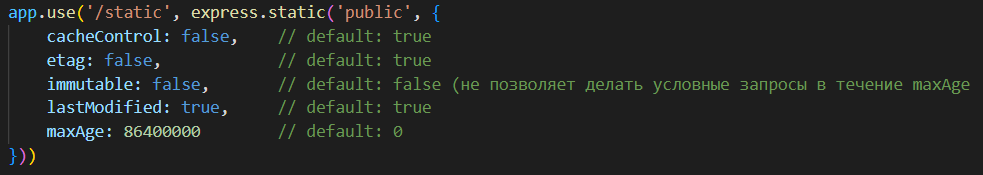
**Кэширование на стороне клиента**

Кэширование на стороне клиента позволяет клиентам временно сохранять копии ресурсов (например, статических файлов) и использовать их, вместо повторной загрузки с сервера. Это может существенно ускорить загрузку и уменьшить нагрузку на сервер.

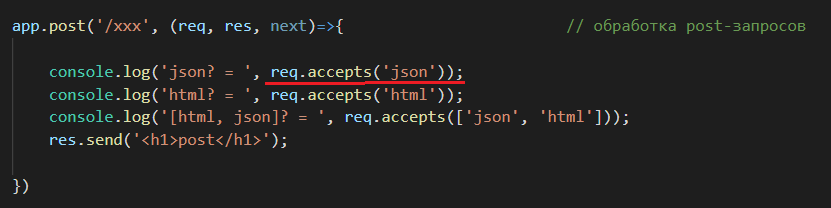
настроить кэширование на стороне клиента, устанавливая заголовки HTTP-ответа, такие как Cache-Control и Expires, чтобы указать браузеру, как долго хранить копию ресурса в кэше.

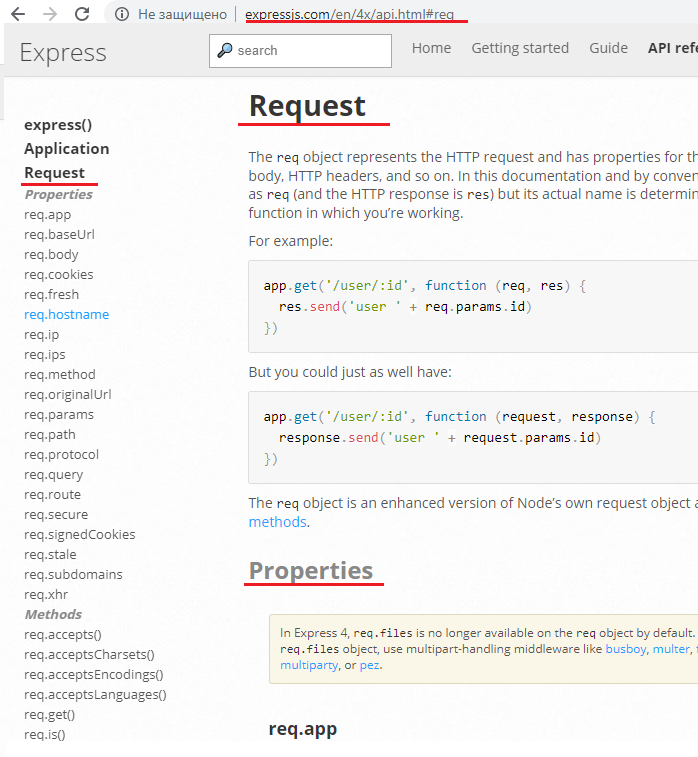
Cache-Control, Etag, Last-Modified, If-Modified-Since, If-None-Match, Expired

настройка кэширования (пакет serve-static)

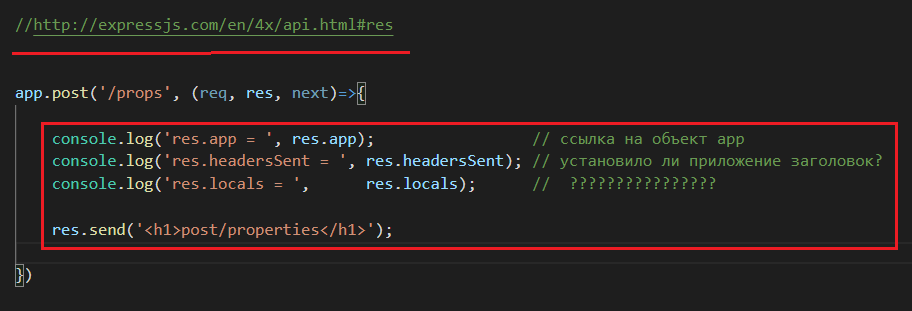


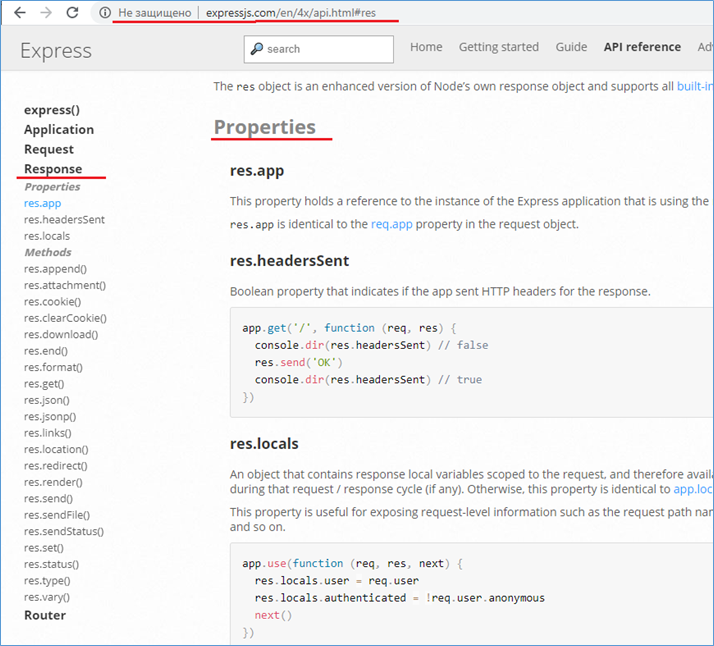
1. Пакет Express. Основные принципы работы. Объекты запроса и ответа. Обработка query- и path-параметров GET-запроса. Методы download и attachment, переадресация.

объект request, методы объекта request



объект response, свойства объекта response





Методы download и attachment:

Express предоставляет методы res.download и res.attachment для отправки файлов клиенту.

* res.attachment используется для отправки файла с заголовком Content-Disposition, который указывает, что файл должен быть открыт в браузере или сохранен на диск клиента, но не обязательно сразу скачан.
* Этот метод не отправляет сам файл, а устанавливает заголовок Content-Disposition для текущего ответа. Клиент может выбрать, что делать с файлом (открыть или сохранить).
* res.attachment не выполняет скачивание файла на жесткий диск клиента автоматически.
* res.download, с другой стороны, используется для отправки файла с заголовком Content-Disposition, который указывает браузеру, что файл должен быть автоматически скачан (не открыт) и сохранен на жесткий диск клиента.
* Этот метод выполняет скачивание файла на жесткий диск клиента. Клиент не видит содержимого файла в браузере, а файл сохраняется на диск по указанному пути.

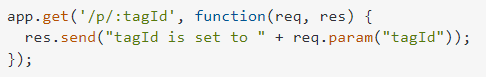
Обработка query- параметров:

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.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.

**path-**

Для получения параметров из URI в строке передаваемой в качестве пути обработчика используются метки вида: :[Par Name];



Для получения параметров используется объект запроса и свойство params.

**Переадресация** res.redirect

1. Пакет Express. Основные принципы работы. Обработка тела (form, json) POST-запроса. Методы accepts и format.

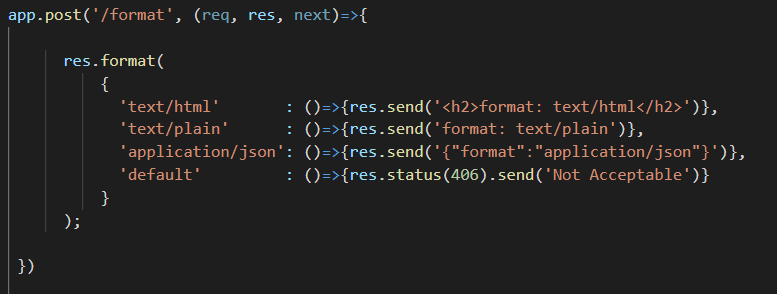
Тело запроса содержит коллекцию объектов ключ-значение, которые были отправлены в теле запроса. По умолчанию дается значение undefined и заполняется только тогда, когда используется middleware для обработки тела запроса. Такими middleware являются **multer (multipart data)** и **body-parser** (deprecated – исп express.json());

Для получения доступа к объекту с параметрами используется объект запроса и его параметр body;

Const body = req.body;

**Format**

Эта функция проверяет заголовок HTTP-запроса Accept, а затем вызывает соответствующий обработчик в зависимости от значения Accept.

****

**Accepts**

Функция req.accepts() проверяет приемлемость указанных типов контента на основе поля заголовка Accept HTTP запроса. Метод возвращает наилучшее совпадение, в противном случае он возвращает значение false, если ни один из указанных типов контента не является приемлемым.

1. Handlebars. Назначение и порядок использования шаблонов, макетов, хэлперов и частичных представлений.

Шаблонизация: механизм генерации текстового файла на основе готового шаблона.

Шаблон: (template) статический текст с вкраплениями специальных символов (комбинацией символов), предназначенных для динамической генерации новых фрагментов текста.

Шаблонизатор: (template engine) программа, выполняющая шаблонизацию. Например, Mustashe (языконезависимый), Jade, Nunjucks, Handlebars, EJS, Pug и др.

Handlebars - это шаблонизатор для создания динамических HTML-страниц в JavaScript-приложениях, как на стороне сервера, так и на стороне клиента. Он позволяет разделить структуру HTML и данные, что делает код более поддерживаемым и уменьшает дублирование кода.

Шаблоны представляют собой HTML-файлы, в которых встроены динамические места для подстановки данных.

<div>

<h1>{{ title }}</h1>

<p>{{ content }}</p>

</div>

**Макеты** представляют собой общую структуру страницы, которая может содержать общий заголовок, навигацию, подвал и другие элементы, которые повторяются на разных страницах вашего веб-приложения.

**Хэлперы** - это пользовательские функции, которые могут выполнять сложные операции над данными перед их вставкой в шаблон.

встроенные хэлперы: #if, #unless, #each, #with, #lookup, #log

**Частичные представления** - это маленькие фрагменты шаблонов, которые могут быть повторно использованы в разных местах ваших шаблонов.

1. Пакет Express. Основные принципы работы. Порядок работы с cookie и signed cookie.

Cookie-файлы позволяют сохранять на стороне клиента некоторые данные в течении заданного интервала времени, которые при смене страницы или повторном заходе на сайт помогают идентифицировать пользователя и, например, помогают восстановить состояние предыдущей сессии.

В Node.js cookie управляются двумя методами объекта ответа:

* cookie() – устанавливает значение по ключу
* clearCookie() - удаляет по заданному ключу значение у клиента, если ключ не задан - удаляет все.

Для работы с куки изначально нужно скачать модуль cookie-parser. Подключается он как middleware:

app.use(cookieParser('secret key'))

В качестве секретного ключа может быть передана строка или массив строк. Используется для подписания кук. Если он не задан, то подписанные куки не будут парситься. Если передан массив, то ключи будут использоваться по очереди.

Так же может быть передан объект options в который передается функция, которая определяет как будет получаться значение кук.

По сути middleware парсит заголовок Cookie и представляет содержимое в req.cookies и если имеется секрет, то req.signedCookies.

**Параметры передаваемые в cookie()**

При создании кук передаются имя, значение и 3 не обязательным параметром может быть передан объект параметров, который содержит следующий набор параметров:

* domain – домен, на котором могут использоваться куки
* path – описание маршрута, на который распространяется действие кук
* maxAge – время жизни
* secure – true – значит могут передаваться только по https
* httpOnly – true – значит, что файлы кук могут изменяться только сервером
* signed – true – то файл будет подписан с использованием секрета

1. Пакет Express. Основные принципы работы. Порядок работы с session.

В web-разработке под сессией понимается промежуток времени, в течении которого пользователь находится на сайте. Сессия начинается в момент захода на сайт и заканчивается при закрытии вкладки браузера или при переходе в пределах текущей вкладки на другой ресурс, и позволяет сохранять, например, данные в действиях пользователя, которые не теряются при переходе на другую страницу.

**Сессия** – серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении. Время жизни: **timeout** (системный параметр, обычно равен 10-30 минут) – максимальное время между запросами клиента.

Если timeout превышен, то session разрушается и при последующем запросе создается новый экземпляр. Каждая сессия имеет собственный идентификатор (Session ID, 128 или больше бит (16 байт)). Каждый request принадлежит какой-то сессии. Сессия характеризуется двумя параметрами: timeout и session ID.

Обычно в request сервер записывает либо идентификатор этой сессии, либо просто программную ссылку на этот объект сессии.

*Если с одним и тем же сервером работает много клиентов, у каждого из них своя сессия.*

**Куки** – это порция информации, которая может быть сохранена на стороне клиента по инициативе сервера. Когда клиент делает первый запрос, сервер проверяет, есть ли у него заголовок с именем куки. Если этого заголовка нет, то он считает, что это первый запрос и для него создается сессия.

Клиент должен хранить информацию о том, в рамках какой сессии мы отправляем эти запросы. Браузер в себе сохраняет либо файл куки, но чаще сохраняет **localstorage** (поддерживается на стороне браузера).

**LocalStorage** **это** свойство объекта window, предназначенное для хранения пар ключ/значение в браузере. В зависимости от браузера, мы можем сохранять до 5 мб данных.

**IndexedDB** – **это** встроенная база данных, более мощная, чем localStorage . Хранилище ключей/значений: доступны несколько типов ключей, а значения могут быть (почти) любыми. Поддерживает транзакции для надёжности.

Для работы с сессией в express используется модуль express-session который по умолчанию включает в себя cookie-parser

Инициализация Node.js сессии осуществляется с помощью функции промежуточной обработки.

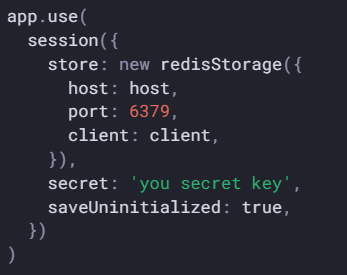
App.use(session({params}))

В качестве параметров сесии может быть передано следующее:

* Cookie – настраивает объект для SESSIONID куки. Параметры как для куки (40 вопрос)
* genid – функция, которая возвращает новый идентификатор сессии в виде строки
* resave - булевое значение, указывает, нужно ли пересохранять сессию в хранилище, если она не изменилась
* rolling - булевое значение, указывающее, нужно ли устанавливать идентификатор сессии cookie на каждый запрос
* saveUnauthorized - булевое значение, если true, то в хранилище будут попадать пустые сессии
* secret - строка, которой подписывается сохраняемый в cookie идентификатор сессии

Объект Node.js сессии глобальный и будет перезаписываться данными последнего пользователя. Чтобы избежать этого используются хранилища оперативной памяти.

Для примера можно использовать redis



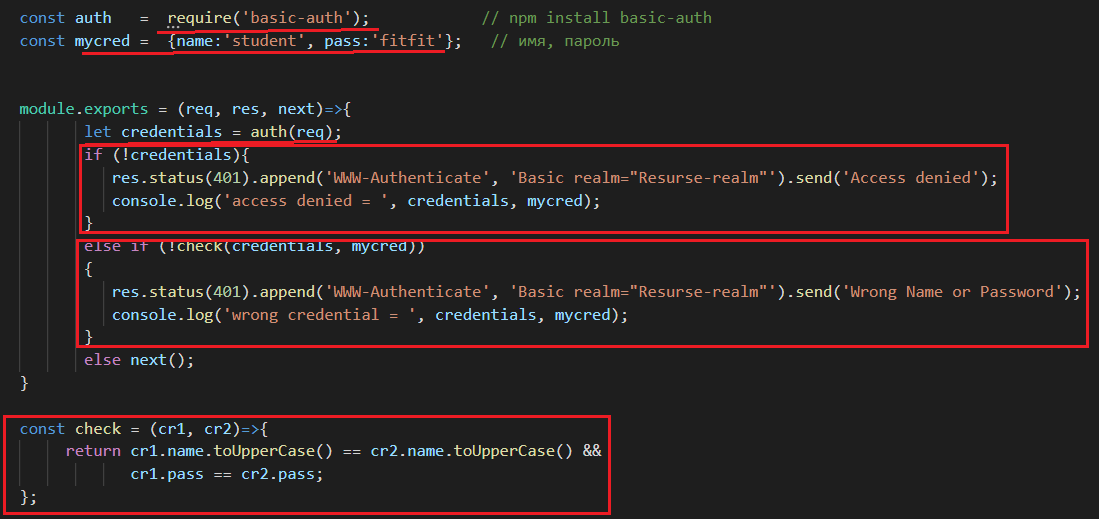
1. Аутентификация. Разработка приложения с Basic-аутентификацией.

Npm пакет basic-auth либо через passport

Basic-auth:

const basicAuth = require('basic-auth');

потом const credentials = basicAuth(req) там где нужна аутентификация и потом проверяем данные уже, если не сходятся шлем 401



Passport:

Импортируем пасспорт и app.use(passport.initialize())

Определяем пасспорт стратегию через пасспорт.юз там внутри указываем нью бейсик стратеджи и колбек и на нужных энпоинтах хуячим пасспорт.аунтефикейт(бейсик, сессия фолс)

Еще сериализацию юзаем чтобы запомнить что у нас пользователь залогинился

**22-23 тоже самое только дайджест и локал стратеджи(надо указать какое поле за логин а какое за пароль отвечают)**

**24. Токен-аутентификация.**

**JWT (JSON Web Token) -** это стандарт открытого формата токенов, используемых для представления утверждений между двумя сторонами в компактном и самодостаточном виде. JWT состоит из трех частей: заголовка (header), полезной нагрузки (payload) и подписи (signature). Он широко используется для аутентификации и авторизации в веб-приложениях.

Заголовок (Header): Заголовок содержит информацию о типе токена (JWT) и используемом алгоритме подписи. Обычно это JSON-объект, который кодируется в Base64url.

Полезная нагрузка (Payload): Полезная нагрузка содержит утверждения (claims) о пользователе и другую информацию. Утверждения могут быть стандартными (например, идентификатор пользователя, роли, срок действия) или пользовательскими (любая другая информация, необходимая приложению). Полезная нагрузка также кодируется в Base64url.

Подпись (Signature): Подпись создается с использованием заголовка, полезной нагрузки, секретного ключа (или закрытого ключа в случае асимметричного шифрования) и алгоритма подписи, указанного в заголовке. Подпись позволяет проверить целостность токена и удостовериться, что он не был изменен. Подпись также представляет собой закодированную строку.

**access\_token** используются для получения доступа к защищенным ресурсам; короткоживущий, многоразовый.

**refresh\_token** используется, когда access\_token истёк, для получения новой пары refresh\_token+access\_token; отправляется на специальный url; возвращается access\_token + refresh\_token; долгоживущий, одноразовый. При краже access\_token им можно пользоваться ограниченное время; при краже refresh\_token он становится бесполезным при ре-логине.

1**.** Аутентификация пользователя: Пользователь вводит свои учетные данные (логин и пароль) на странице входа, как и в стандартной схеме FORMS-аутентификации.

2. Проверка учетных данных: Сервер проверяет учетные данные пользователя. Если они верны, сервер создает пару токенов - токен доступа (access token) и токен обновления (refresh token). Токен доступа предоставляет пользователю доступ к защищенным ресурсам, а токен обновления используется для обновления токена доступа без необходимости повторной аутентификации пользователя.

3. Отправка токенов клиенту: Сервер отправляет токен доступа и токен обновления в ответе на успешную аутентификацию. Обычно это происходит в виде JSON-ответа или установкой кук (например, в HTTP заголовке "Set-Cookie").

4. Хранение токенов: Клиент (браузер) сохраняет полученные токены в безопасном месте. Обычно это хранение в куках или в локальном хранилище браузера.

5. Использование токена доступа: При отправке запросов на защищенные ресурсы клиент включает токен доступа в запрос (например, в заголовке "Authorization" или в параметре запроса). Сервер проверяет валидность токена доступа и предоставляет доступ к ресурсам, если токен действителен.

6. Обновление токена доступа: Когда токен доступа истекает или становится недействительным, клиент может использовать сохраненный токен обновления для запроса нового токена доступа. Клиент отправляет запрос на специальный эндпоинт сервера, предоставляя токен обновления вместе с запросом. Сервер проверяет валидность токена обновления и, если он действителен, генерирует новую пару токенов (токен доступа и токен обновления) и отправляет их клиенту.

7. Завершение сеанса: Пользователь может завершить сеанс, выйдя из системы. В этом случае клиент удаляет сохраненные токены

**25. OAuth 2.0. Общая схема работы OAuth. Разрешение на авторизацию, виды. OpenID.**

**OAuth 2.0 -** это стандарт протокола для авторизации и предоставления доступа приложениям к защищенным ресурсам пользователя, таким как данные на сайтах и социальных сетях. OAuth 2.0 позволяет пользователям делегировать доступ к своим данным без предоставления пароля третьим приложениям.

Простыми словами, OAuth 2.0 позволяет вам разрешить приложению действовать от вашего имени на определенном веб-сервисе (например, Facebook или Google), без передачи своего пароля приложению. Вместо этого, вы предоставляете приложению временный токен, который оно использует для доступа к вашим данным.

**Основная схема работы OAuth 2.0:**

**Регистрация приложения**: Разработчик приложения регистрирует его на веб-сервисе (например, на сайте Google Developers). Это дает приложению идентификатор клиента (Client ID) и секретный ключ (Client Secret).

**Запрос разрешения**: Пользователь заходит в приложение и хочет использовать определенные функции, которые требуют доступа к его данным на веб-сервисе (например, доступ к контактам в Gmail).

**Перенаправление на веб-сервис**: Приложение перенаправляет пользователя на сайт веб-сервиса, предоставляя свой идентификатор клиента и запрашиваемые разрешения.

**Подтверждение пользователя**: Пользователь входит на веб-сервис, если еще не авторизован, и видит список разрешений, которые запрашивает приложение. Пользователь решает, дать ли приложению доступ к своим данным.

**Выдача временного токена**: После того как пользователь дает свое согласие, веб-сервис выдает временный токен (Access Token) приложению. Этот токен используется приложением для доступа к данным пользователя.

**Использование токена**: Приложение может использовать временный токен для доступа к данным пользователя на веб-сервисе. Токен может быть ограничен во времени и действителен только для определенных разрешений.

**Обмен токеном на данные**: Приложение обменивает временный токен на доступ к данным пользователя на веб-сервисе. Для этого оно использует свой Client ID и Client Secret.

**Разрешение на авторизацию (Scopes):**

Разрешение на авторизацию (или просто "разрешение") определяет, какие действия или данные приложение имеет право выполнять или получать от веб-сервиса от имени пользователя. Например, приложение может запросить разрешение на доступ к контактам, электронной почте или фотографиям пользователя. Пользователь решает, какие разрешения предоставить или отказать.

**Гранты** (или "grant types") - это одна из важных составляющих протокола OAuth 2.0, которые определяют, как именно приложение получает доступ к данным пользователя и как оно аутентифицирует себя перед веб-сервисом. Гранты определяют способ обмена временными токенами на доступ к данным.

**Authorization Code Grant**: Клиентское веб-приложение (получатель) получает грант (временный код) от веб-сервиса (авторизующей стороны) после успешной авторизации пользователя. После этого клиентское приложение обменивает грант на временный токен доступа.

**Implicit Grant**: Клиентское веб-приложение (получатель) получает грант (временный токен доступа) напрямую от веб-сервиса (авторизующей стороны) после успешной авторизации пользователя.

**Client Credentials Grant**: Клиентское приложение (получатель) отправляет запрос с идентификатором клиента и секретным ключом к веб-сервису (авторизующей стороне) и получает временный токен доступа.

**Resource Owner Password Credentials Grant**: Клиентское приложение (получатель) получает грант (пару имя пользователя и пароль) напрямую от пользователя, который предоставляет их приложению. Приложение затем обменивает эти данные на временный токен доступа.

**Refresh Token Grant**: Клиентское приложение (получатель) может обменивать временный токен доступа на обновленный токен с помощью гранта обновления (Refresh Token Grant), если у него есть обновленный токен доступа или токен обновления (Refresh Token).

**OpenID:**

OpenID - это расширение протокола OAuth 2.0, которое добавляет функциональность аутентификации пользователя. Вместо того, чтобы только предоставлять доступ к данным, OpenID позволяет пользователю аутентифицироваться на веб-сервисе, используя свой аккаунт на другом веб-сервисе (например, Google или Facebook).

С помощью OpenID, вы можете войти на сайт, не создавая новый аккаунт, а используя учетные данные с другого сайта, который поддерживает OpenID. Это упрощает процесс регистрации и входа на сайты, так как пользователь может использовать свои существующие учетные данные.

Важно помнить, что безопасность OAuth и OpenID зависит от правильной реализации и хранения секретных данных, поэтому разработчики должны следовать рекомендациям по безопасности при работе с этими протоколами.

**26. Авторизация: виды, реализация.**

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

**Authorization:** процедура предоставления доступа к определенному ресурсу/действию на основании каких-то признаков (флагов, ролей, привилегий и т.д.).

**Authorization***: виды:*

*а) дискреционное управление доступом (DAC);*

*б) мандатное управление доступом (MAC);*

*в) управление доступом на основе ролей (RBAC).*

1. **дискреционное (прямое) управление доступом (DAC):** список доступа (ACL) субъектов к объекту (Microsoft Windows, Unix)

*Например:*

пользователю user\_1 разрешено читать файл file\_1

пользователю user\_2 разрешено читать и писать в файл file\_1

1. **мандатное управление доступом (MAC):** уровень доступа субъекта ≥ уровень секретности объекта

*Например:*

пользователю с уровнем доступа L1 разрешено читать файл file\_1

1. **управление доступом на основе ролей (RBAC):** роль у субъекта, определяющая привилегии относительно объекта (Oracle, Microsoft Active Directory, PostreSQL)

*Например:*

пользователю с ролью role\_1 разрешено читать файл file\_1

**роль -** набор правил, определяющих, какими привилегиями и по отношению к каким объектам будет обладать пользователь, которому будет назначена эта роль**.**

**привилегия (операция) —** полномочие субъекта относительно какого-то объекта.

**самый распространенный способ RBAC (+ACL)**

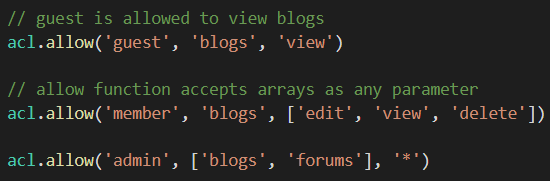
**27. Библиотеки Node ACL и CASL.**

**Node ACL и CASL -** это библиотеки для управления доступом и разрешениями в Node.js-приложениях.

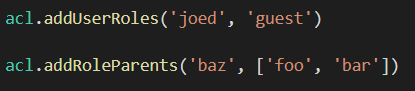
**Node ACL:**

библиотека, которая обеспечивает реализацию ACL (Access Control List) для управления доступом к данным и ресурсам в Node.js-приложениях. Она позволяет определять правила доступа для ролей и пользователей, а также применять их к различным ресурсам.

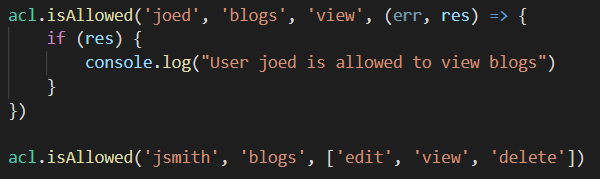
**NODE ACL: allow(**roles, resources, permissions, function(err)**)**

****

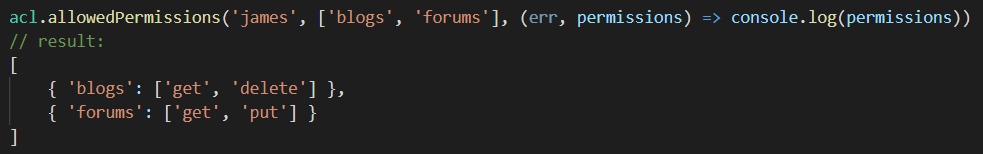
**addUserRoles(**userId, roles, function(err)) **addRoleParents**(role, parents, function(err)**)**

****

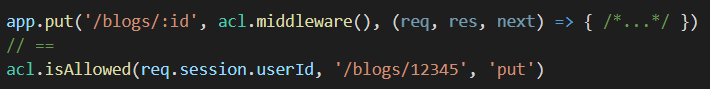
**isAllowed(**userId, resource, permissions, function(err, allowed)**)**

****

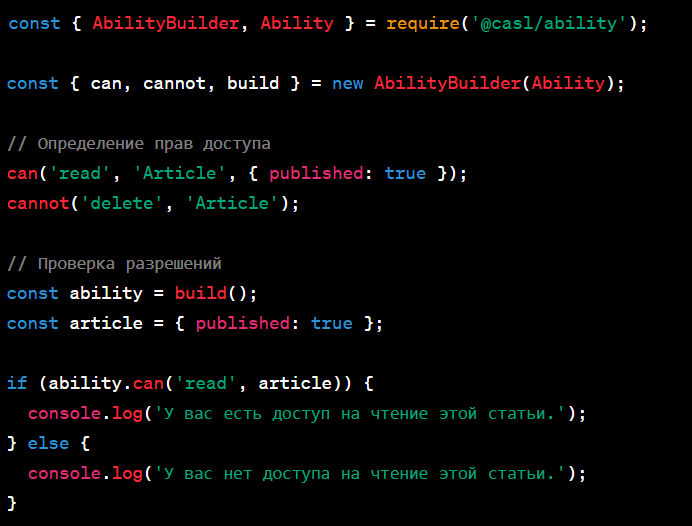
**allowedPermissions(**userId, resources, function(err, obj)**)**

****

**middleware**([numPathComponents, userId, permissions]**)**

****

**CASL:**

****

$exists, $in, $gte, $lte, $all, $regex…

**28. Публикация приложения. Основные характеристики Heroku, Render. Непрерывная интеграция и доставка (CI/CD).**

**Публикация приложения (деплой, развертывание)**: это процесс, в результате которого приложение становится доступным для пользователей.

**Heroku:** облачная платформа (PaaS – Platform-as-a-Service), основанная на управляемой контейнерной системе.

**Heroku:** легкость управления и масштабирования, мониторинг приложения (пропускной способности, времени отклика, памяти, загрузки ЦП и др.), легкость расширения (можно настроить свой стек или выбрать существующий), различные дополнения, автодеплой с github, CI (непрерывная доставка).

**Heroku:** появился в 2007 г., владеет Salesforce.

**Heroku:** позволяет развернуть приложения на Node, Ruby, Java, PHP, Python, Go, Scala, Clojure.

**Render:** облако для создания и запуска всех приложений и веб-сайтов.

**Render:** характеристики:

* поддерживаемые языки: JS, Python, Ruby, Elixir, Go, Rust, PHP;
* безопасность, т.к. предоставляются бесплатные TLS-сертификаты;
* мониторинг показателей приложения;
* интеграция с популярными инструментами и службами, такими как GitHub (автодеплой), Docker и базами данных/хранилищами, такими как PostgreSQL, MySQL, Redis;
* поддержка фоновый рабочих процессов и запланированных заданий;
* возможность создания приватных сервисов;
* возможность отката до предыдущих развертываний;
* использует HTTP/2 и HTTP/3;
* возможность настройки собственных доменов;
* защита от DDoS-атак;
* глобальный CDN;
* простота использования;
* возможность бесплатного использования;

**Непрерывная интеграция (Continuous Integration, CI) и непрерывная поставка (Continuous Delivery, CD)** представляют собой культуру, набор принципов и практик, которые позволяют разработчикам чаще и надежнее развертывать изменения программного обеспечения.

**Непрерывная интеграция —** это методология разработки и набор практик, при которых в код вносятся небольшие изменения с частыми коммитами. И поскольку большинство современных приложений разрабатываются с использованием различных платформ и инструментов, то появляется необходимость в механизме интеграции и тестировании вносимых изменений.

С технической точки зрения, цель CI — обеспечить последовательный и автоматизированный способ сборки, упаковки и тестирования приложений.

**Непрерывная поставка** начинается там, где заканчивается непрерывная интеграция. Она автоматизирует развертывание приложений в различные окружения: большинство разработчиков работают как с продакшн-окружением, так и со средами разработки и тестирования.

**29. Пакет crypto. Основные принципы работы (шифрование, поточное шифрование, алгоритм Диффи-Хеллмана).**

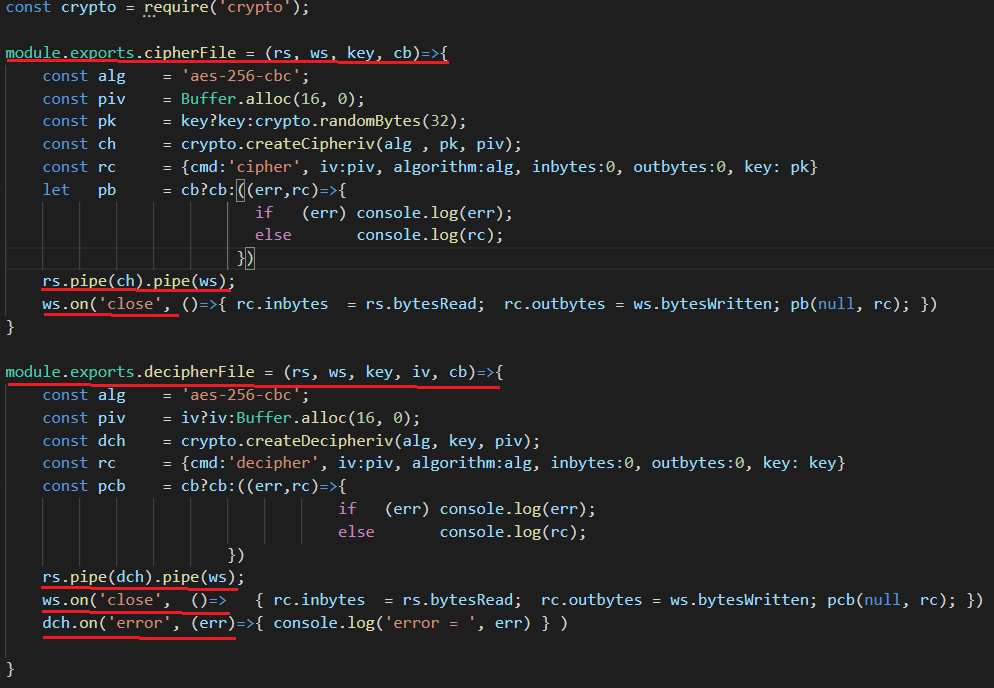
Пакет **crypto** - это встроенный модуль Node.js, предоставляющий различные криптографические функции и алгоритмы.

**Шифрование**

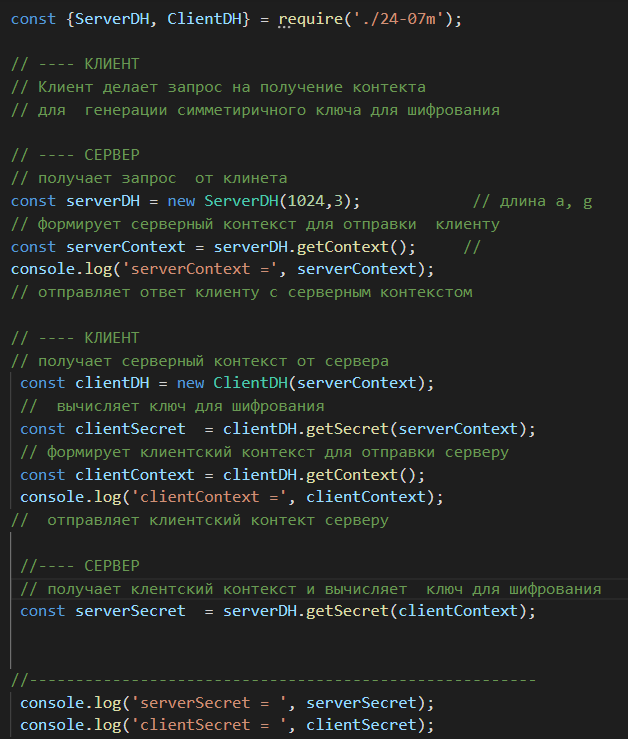
Для шифрования текстовых данных используется функция **crypto.createCipheriv**(algorithm, key, iv), где algorithm - это алгоритм шифрования (например, 'aes-256-cbc'), key - ключ шифрования, и iv - инициализационный вектор.

Для дешифрования данных используется функция **crypto.createDecipheriv**(algorithm, key, iv).

Поточное

****

Диффи-хеллмана

****

****

****

Алгоритм Диффи – Хеллмана для распределения ключей: абоненты А и В могут воспользоваться этим алгоритмом для обмена ключевой информацией по открытым каналам.

Предварительно стороны выбирают большие простые числа n и g.

Протокол обмена:

1. А выбирает случайное большое число х, вычисляет X = g^x mod n и результат вычисления отсылает В.

2. В выбирает случайное большое число у, вычисляет Y = g^у mod n и результат вычисления отсылает А.

3. А вычисляет k1 = Y^x mod n.

4. В вычисляет k2 = Х^у mod n.

Таким образом, k1 = k2 = g^ху mod n = k. Это число сторонами может использоваться как совместный ключ (секретный).

Для того чтобы третья сторона смогла вычислить значение k, она должна вычислить значение дискретного логарифма.

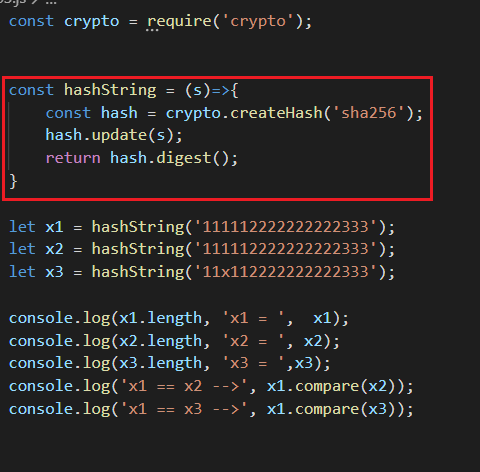
Протокол Диффи – Хеллмана является уязвимым для атаки, называемой «человек в середине»: злоумышленник С может перехватить открытое значение, посылаемое от А к В, и послать вместо него свое открытое значение. Затем он может перехватить открытое значение, посылаемое от В к А, и также передать вместо него свое открытое значение. Тем самым С получит общие секретные ключи с А и В и сможет читать и/или модифицировать сообщения, передаваемые от одной стороны к другой.

**30. Пакет crypto. Основные принципы работы (хэширование, поточное хэширование, цифровая подпись).**

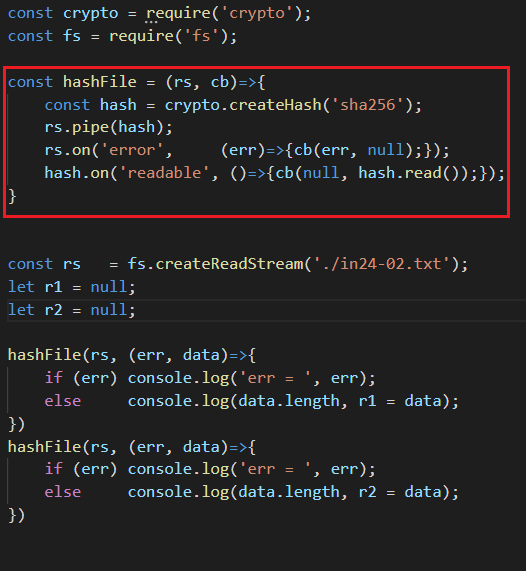
**Хеш-функция –** математическая или иная функция h = H(М), которая принимает на входе строку символов М, называемую также прообразом, переменной длины n и преобразует ее в выходную строку фиксированной (обычно – меньшей) длины l.

**Хеширование (или хэширование, англ. hashing) –** это преобразование входного массива данных определенного типа и произвольной длины (практически) в выходную битовую строку фиксированной длины

Хэширование:

****

хеширование поточное

****

**Электронная цифровая подпись** – бинарная (или в ином виде) последовательность символов, являющаяся реквизитом электронного документа, зависящая от содержания этого документа и предназначенная для подтверждения целостности и подлинности электронного документа

**Цифровая подпись** создается с использованием приватного ключа, а проверяется с использованием соответствующего публичного ключа. Процесс создания цифровой подписи включает хеширование сообщения, шифрование хеша приватным ключом и присоединение подписи к сообщению. Получатель может проверить цифровую подпись, расшифровав ее с помощью публичного ключа отправителя и проверив соответствие полученного хеша хешу сообщения.





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

**WebDAV** – 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)

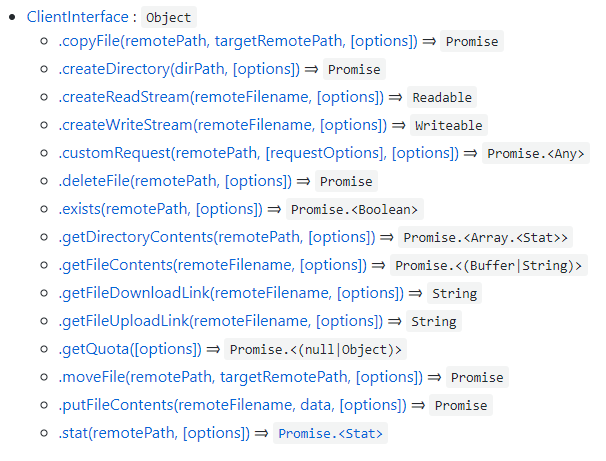
WebDAV: унаследованные HTTP-методы

* GET – скачать файл.
* PUT – загрузить файл на сервер.
* DELETE – удалить серверный объект.

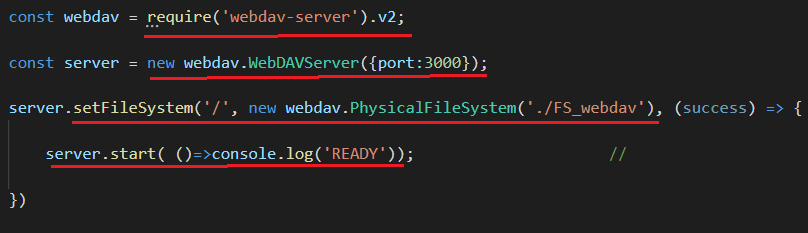
Работа WebDav регулируется следующими стандартами: RFC 2291, RFC 4918, RFC 3648, RFC 3744, RFC 3253.

WebDAV расширяет набор стандартных HTTP-команд и заголовков, разрешенных для методов запроса . К добавленным командам относятся:

* COPY - копировать ресурс из одного URI в другой
* LOCK - установить lock на ресурсе. WebDAV поддерживает как общие, так и эксклюзивные блокировки.
* MKCOL - создавать коллекции (также известный как каталог)
* MOVE - перемещать ресурс из одного URI в другой
* PROPFIND - получить свойства, сохраненные как XML , из веб-ресурса . Он также перегружен , чтобы можно было получить структуру коллекции (также известную как иерархия каталогов) удаленной системы.
* PROPPAT CH - изменение и удаление нескольких свойств ресурса за один атомарный акт
* UNLOCK - снять блокировку с ресурса

****

Сервер

****

+afterRequest((arg, next)=>…)

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

**JSON-RPC** - это упрощенный протокол удаленного вызова процедур (RPC) без сохранения состояния. В первую очередь эта спецификация определяет несколько структур данных и правила их обработки. Он не зависит от транспорта в том смысле, что эти концепции могут использоваться в рамках одного и того же процесса, через сокеты, по протоколу http или во многих различных средах передачи сообщений. Он использует JSON (RFC 4627) в качестве формата данных.

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)

**Уведомление** - это объект запроса без элемента "id". Объект запроса, являющийся Уведомлением, означает отсутствие интереса Клиента к соответствующему объекту Ответа, и поэтому клиенту не требуется возвращать объект Ответа. Сервер НЕ ДОЛЖЕН отвечать на Уведомления, в том числе на те, которые находятся в пакетном запросе.

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

* Version – версия протокола
* Result – объект, понятный клиенту
* Error – взаимозаменяемый с результатом, содержит сообщение об ошибке
* Id – ссылается на запрос, на который отвечает

**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}

Server = new JsonRPCServer()

Client = jsonRPCClient.http()

Server.addMethod()

Sever.recive(JsonRPCServer) .then((jsonRPCResponse) => {}))

1. **WASM. Назначение, принципы использования, emcc. Разработка клиент-серверного приложения с применением WASM.**

WASM (Web Assembly) – бинарный формат исполняемого файла, который может исполняться в виртуальной стек машине (JS Engine);

• Код быстрее, чем JS

• Поддерживается большинством браузеров

• Выполняется в sandbox

• Есть отладчики

• Открытый стандарт

**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

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

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

const wasmModule = new WebAssembly.Module(wasmCode);

const wasmInstance = new WebAssembly.Instance(wasmModule);

const wasm = wasmInstance.exports;

Здесь создается экземпляр **WebAssembly.Module** из байтового кода wasmCode

Затем создается экземпляр **WebAssembly.Instance** на основе модуля wasmModule. Этот экземпляр представляет **выполнимую версию** WebAssembly-модуля, которую можно вызывать из JavaScript.

Далее, **экспортируются** функции из WebAssembly-модуля и сохраняются в переменной wasm.

**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

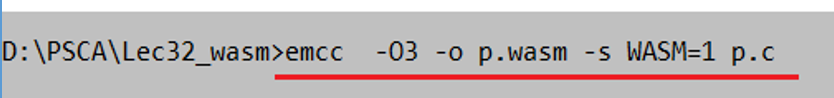
• Export – получает инстанс, который позволяет выполнять эти ф-ции.

Модель памяти WebAssembly очень проста. Это плоский «кусок» памяти, в котором находится код проги, глобальные переменные, стек и куча. Есть возможность сделать так, чтобы память была расширяемой, т.е. если 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.

**Emcc**

**emcc (Emscripten Compiler Frontend)** - это инструментарий, который позволяет компилировать программы, написанные на языках C и C++, в WASM и JavaScript.



p.wasm -> в статик потом фетчем его ловим



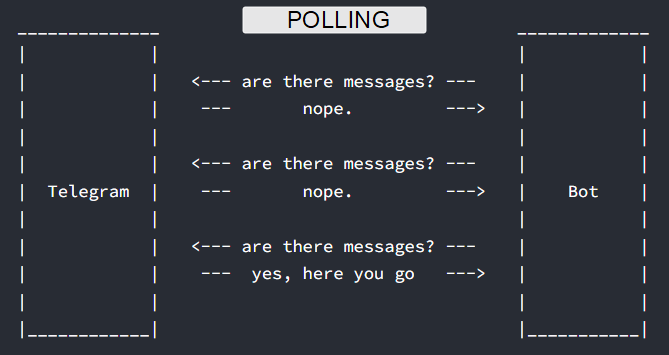
1. **Long pool–сервер, принцип работы. Пример (Telegram bot, лабораторная работа).**

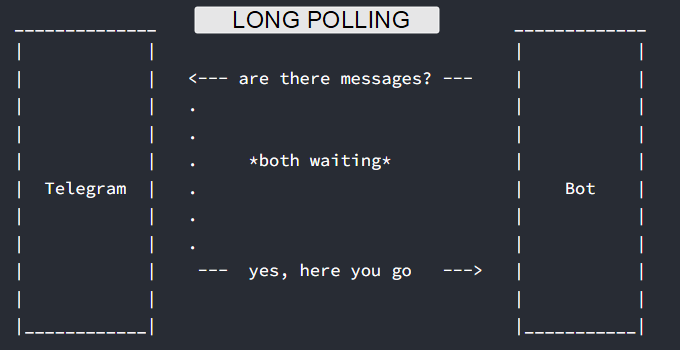
**Long Polling** — это технология, которая позволяет получать информацию о новых событиях с помощью «длинных запросов». Сервер получает запрос, но отправляет ответ на него не сразу, а лишь тогда, когда произойдет какое-либо событие (например, поступит новое входящее сообщение), либо истечет заданное время ожидания.

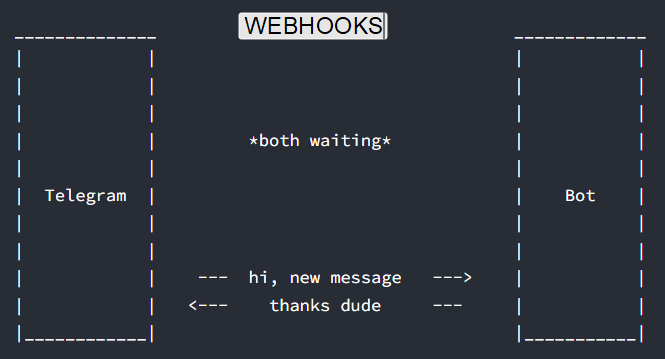
Другими словами, получая от вас запрос, сервер ждет, когда произойдет событие, о котором он должен вас уведомить, и, когда оно происходит, Long Poll сервер отправляет ответ на ваш запрос, содержащий информацию о случившемся событии.

**Как работает long poll**

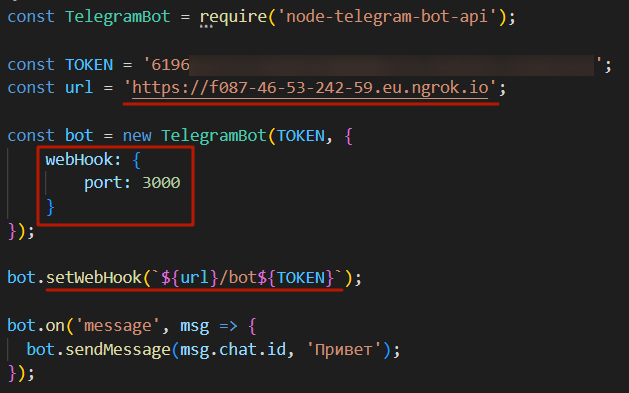
* Запрос отправляется на сервер.
* Сервер не закрывает соединение, пока у него не возникнет сообщение для отсылки.
* Когда появляется сообщение – сервер отвечает на запрос, посылая его.
* Клиент немедленно делает новый запрос.

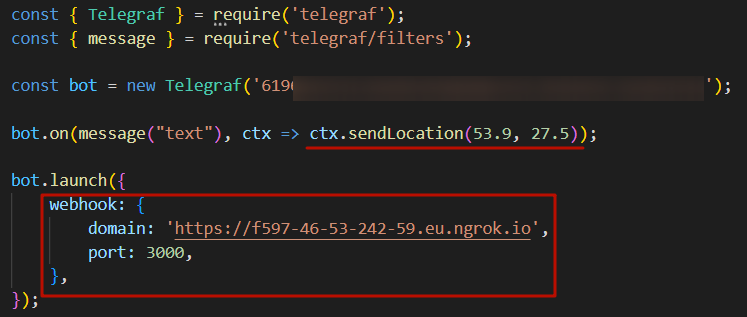




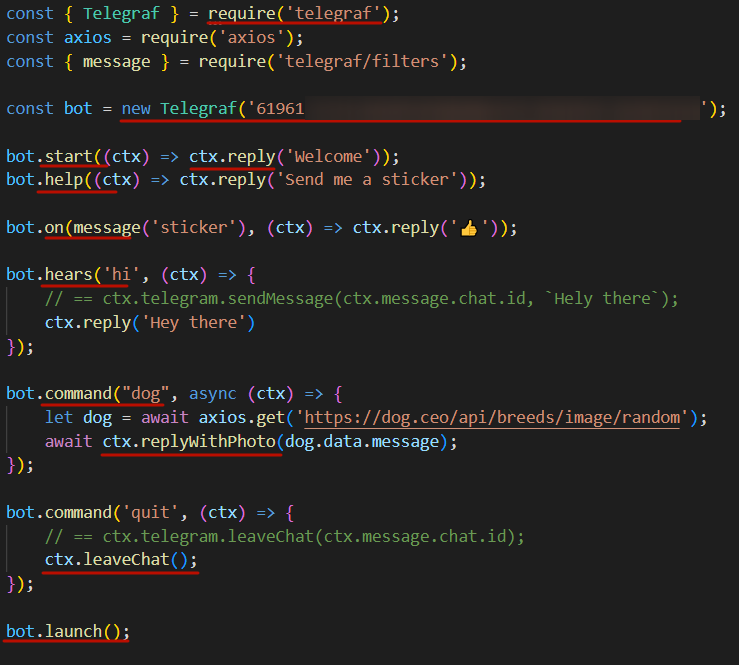


**Webhook** - это URL-адрес, который передается API Telegram. Каждый раз, когда приходит новое обновление для бота, сервер Telegram отправляет это обновление на указанный URL. Аналогично происходит отправка сообщений.





**polling**



1. **REST: определение, ограничения. Open API, Swagger. Порядок разработки сервера с поддержкой Swagger.**

**REST**: архитектурный стиль взаимодействия компонентов распределенного приложения.

**REST:** достоинства

* производительность (кэширование);
* надежность (отсутствие состояния);
* простота (унифицированность интерфейса, использование повсеместных стандартов,…);
* изменяемость;
* масштабируемость.

**REST:** шесть обязательных ограничений:

* модель клиент-сервер;
* отсутствие состояния на стороне сервера, сохранение состояния допускается на стороне клиента, допускается сохранение состояния в другом сервисе (например, в БД);
* кэширование на стороне клиента, сервер явно управляет кэшированием;
* единообразие интерфейсов (идентификация ресурсов, манипуляция ресурсами через представления, самодостаточные сообщения, HATEOAS);
* для клиента сервер должен представляться конечным;
* код по требованию: допускается (необязательно) выгрузка на клиент апплетов или сценариев для расширения его функциональности.

**Open API:** спецификация для описания WEB API, ориентирована на REST.

**Swagger:** реализация спецификации Open API, набор инструментов, который помогает описывать RESTful API, машинно-независимый и независимая от языков программирования.

swaggerUi.setup(swaggerDocument)