1. Протокол HTTP, основные свойства HTTP, структура запроса и ответа, методы, заголовки, коды состояний. Понятие web-приложения, структура и принципы работы web-приложения. Понятие асинхронности.
2. Web-сервер. Ресурсы, потребляемые web-сервером. Блокирующие и неблокирующие операции ввода/вывода. Решение проблемы блокирующего ввода/вывода. Понятия конкурентность и параллельность. Закон Амдала.
3. Платформа Node.js, версии, назначение, основные свойства, структура, принципы работы, основные встроенные модули и их назначение, применение внешних модулей (пакетов).
4. Глобальные объекты Node.js (global, process) и их применение. Системные (стандартные потоки) Node.js и их применение. Модуль console: функции log, error, dir, time, timeEnd, trace. Примеры.
5. Асинхронное программирование. Функция обратного вызова. Проблема "Callback hell" и способы решения. Примеры.
6. Асинхронное программирование. Механизм Promises. Механизм async/await. Примеры.
7. Класс EventEmitter, назначение, применение. Пример.
8. Функции setTimeout, setInterval, nextTick, ref, unref, назначение, применение. Примеры.
9. Модули и пакеты Node.js, CommonJS, функция require, кэширование модуля, область видимости в пакете, экспорт объектов, функций, конструкторов. Применение require для работы с json-файлами. Параметризируемый модуль. Пример
10. Модули Node.js. Форматы модулей. Модули ES6: экспорт (по умолчанию, именованный, до/после объявления), импорт, динамический импорт. Примеры.
11. Пакетный менеджер NPM, глобальное хранилище, просмотр установленных пакетов, скачивание пакетов, назначение файла package.json, локальные хранилища пакетов, удаление пакетов, публикация пакета, SemVer. Примеры.
12. HTTP-сервер. Порядок разработки простейшего HTTP-сервера в Node.js. События, генерируемые на сервере. Извлечение данных из HTTP-запроса (параметры, тело, заголовки…), формирование данных HTTP-ответа.
13. HTTP-сервер. Порядок разработки простейшего HTTP-сервера в Node.js. Объекты request и response, их свойства, методы, события. Порционная отправка и получение данных.
14. HTTP-сервер. Типичный цикл работы http-сервера, маршрутизация. Статические ресурсы, MIME, отдача статики.
15. JSON, XML: порядок работы с json- и xml-сообщениями (формирование и разбор). Загрузка файла на сервер (пакет multiparty).
16. HTTP-клиент. Порядок разработки простейшего HTTP-клиента в Node.js (модуль http). Отправка GET- и POST-запросов с параметрами, добавление заголовков и обработка ответа. Порционная отправка данных.
17. HTTP-клиент. Порядок разработки HTTP-клиента в Node.js (axios). Отправка GET- и POST-запросов с параметрами, параллельных запросов, добавление заголовков и обработка ответа.
18. Websocket: основные свойства, процедура установки соединения, обмен данными, применение. WebSockets API.
19. Порядок разработки Websocket-сервера и клиента: использование потоков, ping/pong-сообщения, обработка json-сообщений.
20. Понятие RPC. Пакет rpc-websockets: порядок разработки RPC-Websockets-сервера и клиента, работа с процедурами, генерация событий и обработка уведомлений.
21. Файловая система. Порядок работы с файловой системой в Node.js: создание, копирование, чтение, запись, синхронные и асинхронные операции.
22. Файловая система. Порядок работы с файловой системой в Node.js: создание, удаление, переименование, запись в конец, слежение за файлом/директорией, синхронные и асинхронные операции.
23. Файловая система. Порядок работы с файловой системой в Node.js: работа с директориями, проверка наличия файла/директории, синхронные и асинхронные операции. Объект Buffer и работа с ним.
24. Потоки данных, их виды, примеры. Readable поток: режимы работы, порядок создания и использования Readable потока.
25. Потоки данных, их виды, примеры. Writable и Duplex потоки: порядок создания и использования.
26. TCP: основные свойства, установка и закрытие соединения. Разработка простейшего TCP-сервера и TCP-клиента на Node.js.
27. UDP: основные свойства, отличия от TCP. Разработка простейшего UDP-сервера и UDP-клиента на Node.js.
28. MSSQL: пакет mssql, драйверы, подключение, параметры подключения, поддерживаемые типы данных, обычные и подготовленные запросы, обработка результата запроса, создание неявной транзакции.
29. MSSQL: пакет mssql, драйверы, подключение, параметры подключения, статические и динамические запросы, вызов процедуры, создание явной транзакции.
30. MSSQL: пакет mssql, потоковый режим обработки данных, пул соединений.
31. Сравнение SQL и NoSQL, виды NoSQL СУБД, примеры. Репликации, подходы при работе с репликацией.
32. MongoDB: структура БД, пакет mongodb, подключение, параметры подключения, выборка, операторы выборки, добавление документа(ов), изменение документа(ов), удаление документа(ов), типы репликаций в MongoDB, транзакции (явные, неявные).
33. Пакет mongoose: понятие ODM, схема, ограничения схемы, валидация, модель, подключение, выборка, создание документа(ов), удаление документа(ов), обновление документа(ов), транзакции (явные, неявные).
34. REST: определение, достоинства и недостатки, форматы передачи данных, ограничения, HATEOAS.
35. REST: определение, достоинства и недостатки, общепринятые правила REST API, HATEOAS.
36. GraphQL: преимущества и недостатки, особенности, схема, контекст, типы, модификаторы типов, контекст, резолверы, аргументы и переменные.
37. GraphQL: архитектура, выполнение graphql-запроса, порядок разработки graphql-сервера.
38. GraphQL: field-резолверы, enum, interface, union, fragment.

# 

# 1. Протокол HTTP, основные свойства HTTP, структура запроса и ответа, методы, заголовки, коды состояний. Понятие web-приложения, структура и принципы работы web-приложения. Понятие асинхронности.

**HTTP** — протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов (то есть документов, которые могут содержать ссылки, позволяющие организовать переход к другим документам).

*HyperText Transfer Protocol*, «протокол передачи гипертекста»

В соответствии со спецификацией [OSI](http://en.wikipedia.org/wiki/OSI_model), HTTP является протоколом прикладного (верхнего, 7-го) уровня. Актуальная на данный момент версия протокола, HTTP 1.1, описана в спецификации [RFC 2616](http://tools.ietf.org/html/rfc2616).

Протокол HTTP предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное программное обеспечение обрабатывает данный запрос, формирует ответ и передаёт его обратно клиенту.

Задача, которая традиционно решается с помощью протокола HTTP — обмен данными между пользовательским приложением, осуществляющим доступ к веб-ресурсам (обычно это веб-браузер) и веб-сервером. На данный момент именно благодаря протоколу HTTP обеспечивается работа Всемирной паутины.

## основные свойства протокола HTTP

- 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.

## Методы:

**GET** - извлечение данных

**HEAD** - такой же как get но без тела ответа

**POST** - отправка данных на сервер

**PUT** - замена данных

**DELETE** - удаляет указанный ресурс

**CONNECTION** - устанавливает "туннель" к серверу, определённому по ресурсу.

**OPTIONS** - для определения возможностей веб-сервера или параметров соединения для конкретного ресурса

**TRACE** - Возвращает полученный запрос так, что клиент может увидеть, какую информацию промежуточные серверы добавляют или изменяют в запросе

**PATCH** - частичное изменение ресурсов

## Заголовки:

\* **general** - должны включаться в любое сообщение клиента и сервера

\* **request** - используются только в запросах клиента

\* **response** - только для ответов от сервера

\* **entity** - сопровождают каждую [сущность](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D1%89%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) сообщения

## Коды состояния:

- 1xx: информационные сообщения;

- 2xx: успешный ответ;

- 3xx: переадресация;

- 4xx: ошибка клиента;

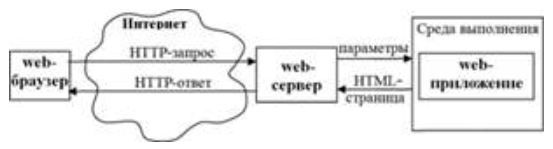
- 5xx: ошибка сервера

URI: Uniform Resource Identifier – унифицированный идентификатор ресурса

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

URN: Uniform Resource Name - унифицированное имя ресурса – URI

Клиент-серверное приложение, у которого клиент и сервер взаимодействуют по протоколу HTTP называется **web-приложением**



Пользователь с помощью браузера отправляет HTTP-запрос по определенному URL-адресу, который сопоставляется с ресурсами на web-сервере. Если данный URL-адрес указывает на динамический ресурс (web-приложений), то сервер запускает некоторую внешнюю программу (web-приложение), формирующую HTML-страницу, которая может быть показана браузером, и возвращает ее клиенту. Основной частью web-приложения является его логика на стороне web-сервера.

Асинхронность представляет возможность одновременно выполнять сразу несколько задач.

Операция называется **асинхронной**, если ее выполнение осуществляется в 2 фазы:

1. заявка на исполнение
2. получение результата

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

**WEB-сервер** — [сервер](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)), принимающий [HTTP](https://ru.wikipedia.org/wiki/HTTP)-запросы от клиентов, обычно [веб-браузеров](https://ru.wikipedia.org/wiki/%D0%91%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80), и выдающий им [HTTP](https://ru.wikipedia.org/wiki/HTTP)-ответы, как правило, вместе с [HTML](https://ru.wikipedia.org/wiki/HTML)-страницей, изображением, [файлом](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), медиа-потоком или другими данными.

**Блокирующий ввод-вывод** называется так, потому что поток, который его использует, блокируется и переходит в режим ожидания, пока ввод-вывод не будет завершён.

Проблема метода блокировки заключается в том, что поток будет спать, пока ввод-вывод не завершится. Поток не сможет выполнять никаких других задач, кроме ожидания завершения ввода-вывода. Иногда вашей программе больше и не надо ничего делать. В противном случае во время ожидания ввода-вывода было бы полезно выполнять другие задачи.

Один из способов осуществить это — использовать **неблокирующий ввод-вывод**. Его идея заключается в том, что когда программа делает вызов на чтение файла, ОС не будет блокировать поток, а просто вернёт ей либо готовые данные, либо информацию о том, что ввод-вывод ещё не закончен. Это не заблокирует поток, но программе придётся позже проверять, завершён ли ввод-вывод.

**Конкурентность** — это выполнение задач за определённое время (например, есть 5 процессов и все они в сумме выполняются в течение 60 минут по очереди). Важная деталь заключается в том, что задачи необязательно выполняются одновременно, поэтому их можно разделить на более мелкие и чередующиеся

**Параллелизм** — это выполнение задач в одно и то же время (например, есть 5 задач, каждая из них выполняется в течение 60 минут). Само название подразумевает, что они выполняются параллельно.

## Закон Амдала

Иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей.

*«В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента».*

Согласно этому закону, ускорение выполнения программы за счёт распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций.

Закон Амдала показывает, что прирост эффективности вычислений зависит от алгоритма задачи и ограничен сверху для любой задачи с ***a<>0(доля a от общего объёма вычислений )***. Не для всякой задачи имеет смысл наращивание числа процессоров в вычислительной системе.

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

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

* основан на Chrome V8;
* среда исполнения приложений на JavaScript;
* поддерживает механизм асинхронности;
* ориентирован на события;
* однопоточный
* не блокирует выполнение кода при вводе/выводе;
* в состав Node.js входят инструменты:

npm – пакетный менеджер;

gyp - Python-генератор проектов;

gtest – Google фреймворк для тестирования С++ приложений;

* использует библиотеки:

V8 – библиотека V8 Engine,

libuv – библиотека для абстрагирования неблокирующих операций ввода/вывода (представляет собой обертку над epoll, kqueue, IOCP);

llhttp – легковесный парсер http-сообщений (написан на C и не выполняет никаких системных вызовов);

c-ares - библиотека для работы с DNS;

OpenSSL – библиотека для криптографии;

zlib – сжатие и распаковка.

* разработчик: Райан Дал
* первая версия: 2009 г.
* стабильные версии: с 2015 г., Node.js 4.0.0;
* Основная сфера применения: разработка web-серверов;

Прежде всего, Node.js отличается от классического JavaScript тем, что исполняемый код выполняется на стороне сервера (backend), а не на стороне браузера

Кроме того, все механизмы обработки запросов и прочих операций ввода/вывода (I/O) построены на событиях. Это означает, что в Node.js нет никакого способа, чтобы заблокировать работающий в данный момент поток. Каждая операция в Node.js выполняется асинхронно. Это является огромным преимуществом, особенно если ваш код должен быть построен на операциях ввода-вывода: чтение дисков, подключение к базе данных, веб-сервисы и т.д.

В отличие от IIS или Apache, Node.js не используют многопоточную модель. В Node.js есть только один рабочий поток, который обслуживает все запросы пользователей и отвечающие ресурсы (на рисунке обозначено в виде ST звезды). И существует POSIX бассейн асинхронных потоков Node.js, который содержит множество асинхронных потоков (AT звезды) для операций ввода-вывода.

Когда пользователь отправляет запрос ввода-вывода, отдельный ST поток обслуживает его, однако он не будет выполнять операции ввода-вывода. Вместо этого, ST поток будет идти в бассейн POSIX асинхронных потоков, чтобы забрать AT поток, произвести операцию до конца, а затем обратно вернуться в исходное состояние для проведения любых других запросов. Таким образом, AT поток будет выполнять операции ввода-вывода асинхронно.

## Встроенные модули: модули которые предоставляют минимальные функциональные возможности

* http – чтобы node.js действовал как http-сервер
* fs – для работы с файловой системой
* console
* events – для обработки событий
* os – информация об ОС
* net – создание сервера и клиент
* url - для синтаксического анализа строк URL

## Внешние модули

* для установки используется пакет
* установить модуль можно локально или глобально
* для локального пакета поиск осуществляется node-modules по восходящему принципу (сначала папка ‘node-modules’, затем папка выше…)
* после поиска среди локальных пакетов, осуществляется поиск среди глобального пакета

**Модули** – это отдельные файлы JavaScript, содержащие функции или объекты, которые могут использоваться другими программами или модулями.

Набор из одного или нескольких модулей обычно называется **пакетом**, такими пакетами обычно управляют менеджеры пакетов. npm

**npm -** **менеджер пакетов**, входящий в состав Node.js. Инструмент командной строки для скачивания/публикации пакетов

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

**Глобальный объект** предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения. Всегда доступен приложениям Node.js без необходимости вызова require()

В браузере он называется window, в Node.js — global.

Обычно не рекомендуется использовать глобальные переменные. Следует применять их как можно реже. Дизайн кода, при котором функция получает входные параметры и выдаёт определённый результат, чище, надежнее и удобнее для тестирования, чем когда используются внешние, а тем более глобальные переменные.

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

**process** – глобальный объект, предоставляет информацию и контролирует текущий процесс Node.js. Хранит информация о среде выполнения, о текущем процессе.

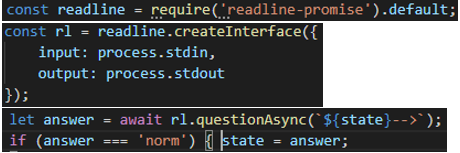
## СТАНДАРТНЫЕ ПОТОКИ:

Эти потоки поставляются с node.js и могут быть использованы без подключения дополнительных библиотек

**process.stdin** - поток на чтение содержит стандартный системный поток ввода.

**process.stdout** - поток на запись, содержащий стандартный системный вывод.

**process.stderr** - поток на запись, содержащий стандартный системный вывод ошибок.



## Модуль console

Предоставляет простую консоль для компиляции, которая экспортирует 2 компонента:

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

2) глобальный экземпляр console, для записи в stdout, stderr

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

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

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

**time … код …timeEnd** – время в мс

**trace** - console.trace(); // (err) - выводить текущий стек trace – тоже в поток ошибок

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

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

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;

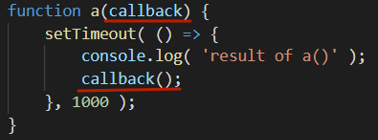
продолжительность исполнения операции B-механизмом, как правило, непредсказуемо;

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

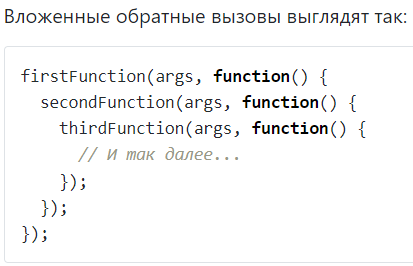
**web API** – это API, которые расширяют функциональность JavaScript для выполнения асинхронных задач (DOM, AJAX, setTimeout и т.д.) + среда неблокирующего ввода-вывода

**Функция setTimeout(callback, delay)** – глобальная функция, которая принимает функцию обратного вызова и временно сохраняет ее. По истечению времени, заданного в миллисекундах, функция обратного вызова помещается в очередь коллбэков (callback queue). Затем event loop перемещает эту функцию в стек вызовов (call stack), когда стек пуст. После этого осуществляется выполнение функции обратного вызова. В основном так работают все web API.

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



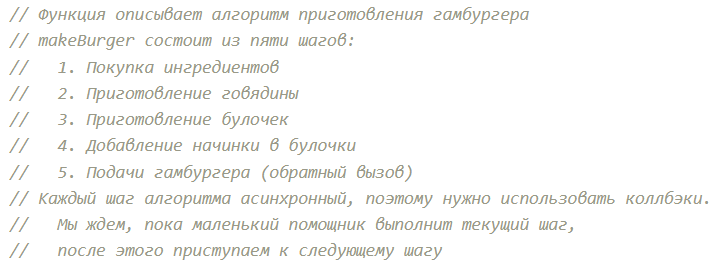
## Проблема "Callback hell"

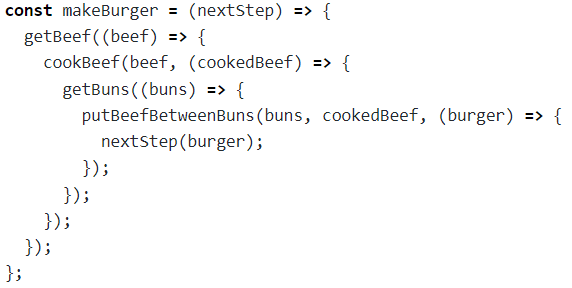


### Решение проблемы

1. Комментируйте код.
2. Разделяйте большие функции на несколько маленьких.
3. Используйте промисы.
4. Используйте async/await.

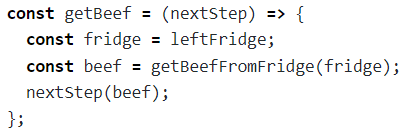
#### Первое решение: комментируйте код



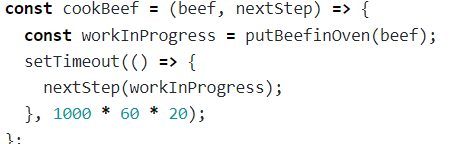


#### Второе решение: разделяйте большие функции на несколько маленьких

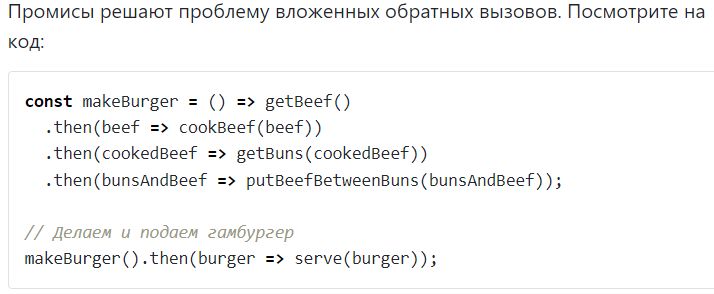
В функции makeBurger из нашего примера этот подход уже реализован. Разберем этот момент подробнее. Чтобы выполнить getBeef, первый коллбэк, вам надо сходить за говядиной в магазин. Ну, или пойти на кухню и достать мясо из холодильника. Представьте, что на кухне два холодильника. Нужно выбрать правильный — тот, в котором лежит говядина. Выразим это с помощью кода.



Чтобы приготовить говядину, нужно положить её в духовку, установить температуру 200 °C и включить таймер на 20 минут.



#### Третье решение: используйте промисы



Чтобы решить задачу, нужно вместо каждого обратного вызова записать промис. Когда коллбэк успешно завершается, промис выполняется с результатом resolve. Если коллбэк не выполняется, получаем ошибку reject.

<https://ru.hexlet.io/blog/posts/kak-izbavitsya-ot-vlozhennyh-kollbekov-rassmatrivaem-na-primere-prigotovleniya-gamburgera>

# 6. Асинхронное программирование. Механизм Promises. Механизм async/await. Примеры.

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

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

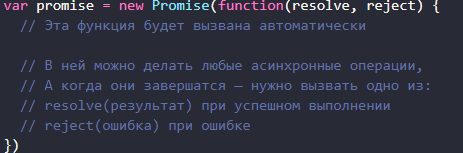
1. заявка на исполнение;
2. получение результата;

**Promise (обычно их так и называют «промисы»)** – предоставляют удобный способ организации асинхронного кода.

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

**Promise** – это специальный объект, который содержит свое состояние.

Вначале **pending («ожидание»)**, затем – одно из: **fulfilled («выполнено успешно»)** или **rejected («выполнено с ошибкой»)**.



метод **Promise.all(iterable)** возвращает промис, который выполнится после выполнения всех обещаний в передаваемом итерируемом аргументе

метод **Promise.race(iterable)** возвращает промис, который будет выполнен или отклонен с результатом исполнения первого выполненного или отклонённого итерируемого промиса

## Механизм async/await

Она позволяет писать код, который выглядит как синхронный, но используется для решения асинхронных задач и не блокирует главный поток.

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

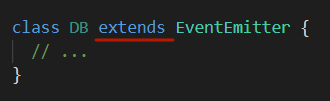
**Async/await: async** - перед объявлением функции, возвращает промис; **await** - блокирует код до тех пор, пока промис не будет разрешен или отклонен.

# 7. Класс EventEmitter, назначение, применение. Пример.

Все объекты, которые генерируют события, представляют экземпляры класса EventEmitter .

С помощью функции eventEmitter.on()к определенному событию по имени цепляется функция обработчика. Причем для одного события можно указать набор обработчиков. Когда объект EventEmitter генерирует событие, происходит выполнение всех этих обработчиков.

1. **EventEmitter**: JS-класс, предоставляющий функциональность для асинхронной обработки событий в Node.js.
2. **Событие** в программном объекте – это процесс перехода объекта из одного состояние в другое. Об этом переходе могут быть извещены другие объекты. У события есть **издатель** (или генератор) события и могут быть **подписчики** (или обработчики) события.
3. **EventEmitter**: применяется в качестве базового для пользовательского объекта. Производный от EventEmitter объект может быть создан с помощью функции inherits модуля utils
4. **EventEmitter**: для наследования можно использовать ключевое слово extends

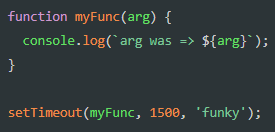


1. **Производный от EventEmitter** объект приобретает функциональность, позволяющую генерировать и прослушивать события
2. для генерации событий предназначена функция emit, а для прослушивания функция on

# 8. Функции setTimeout, setInterval, nextTick, ref, unref, назначение, применение. Примеры.

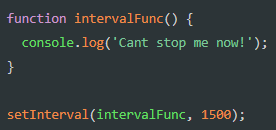
**setTimeout()** может использоваться для планирования выполнения кода после назначенного количества миллисекунд.

setTimeout() первым параметром принимает функцию, которую нужно выполнить, и задержку в миллисекундах, как число, в качестве второго параметра. Также можно перечислить дополнительные аргументы и они будут переданы функции.



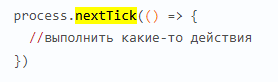
Нельзя полагаться на то, что тайм-аут выполнится после этого точного количества миллисекунд. Это связано с тем, что другой исполняемый код, который блокирует или удерживает цикл событий, отодвигает выполнение тайм-аута на задний план. Единственной гарантией является то, что тайм-аут не будет выполнен раньше, чем заданный интервал.

Если у вас есть код, который нужно выполнить несколько раз, то можно использовать для этого **setInterval()**. setInterval() принимает параметром функцию, которая будет выполняться бесконечное количество раз с заданным интервалом в миллисекундах, сам интервал передается вторым параметром. Как и в случае с setTimeout() можно передавать дополнительные аргументы, эти аргументы будут переданы функции при вызове. Также как и с setTimeout(), задержка не может быть гарантирована из-за операций, которые могут удерживать цикл событий, следовательно, нужно рассматривать эту задержку как приблизительную.



**setInterval()**, также как и setTimeout() возвращает объект Timeout, который можно использовать в качестве ссылки для изменения установленного интервала.

Метод **process.nextTick()** по-особому взаимодействует с циклом событий. Тиком (tick) называют один полный проход цикла событий. Передавая функцию методу process.nextTick(), мы сообщаем системе о том, что эту функцию нужно вызвать после завершения текущей итерации цикла событий, до начала следующей.



процесс Node.js работает до тех пор, пока есть события, требующие обработки; если выполнить для таймера unref, то события, генерируемые таймером не будут учитываться при завершении работы Node.js, ref – противоположная операция.

## ref

Уберет изменения внесенные unref

## unref

изменяет поведение Timeout(возвращается функциями выше). Timeout не выполнит запланированный код, если это последний код, который надо выполнить

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

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

модуль – текстовый файл, содержащий код на языке JS.

модуль используемый несколькими приложениями называют **пакетом**.

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

**реализованные требования CommonJS**

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

Типы модулей:

* входящий в ядро
* разработан самим разработчиком
* установили с помощью npm

Для локального пакета поиск осуществляется в node-modules по восходящему принцип (выше-выше-выше)

После поиска среди локальных пакетов, осуществляется поиск среди глобальных пакетов.

Метод require – загрузить модуль.

var http = require (‘http’);

После первой загрузки модули кэшируются, а это, означает, что любой вызов require(‘’) – ссылающий на 1 и тот же файл – всегда вернет одинаковый объект

* удалить из кэша можно с помощью delete require;

Delete require(‘./m07-01’);

Delete require.cache(‘./m07-01’);

* если модуль удален – для исполнения нужен новый require;

Функция export используется для экспорта функций, объектов или примитивов из файла (модуля)

Экспорт 2 видов:

1) Именованный – для экспорта нескольких величин

2) Default – для экспорта 1 штуки (функции, класса, объекта)

## Именованный экспорт в модуле:



## Параметризированный модуль

# +/-10. Модули Node.js. Форматы модулей. Модули ES6: экспорт (по умолчанию, именованный, до/после объявления), импорт, динамический импорт. Примеры.

Javascript не поддерживает модули на уровне языка, поэтому для ес5 были созданы отдельные реализации для модульности. Два самых популярных, но не совместимых стандарта были CommonJS и AMD.

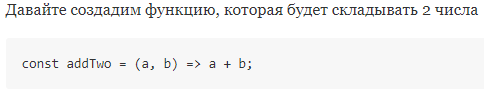
СommonJS был реализован как модульная система NodeJS. Из плюсов

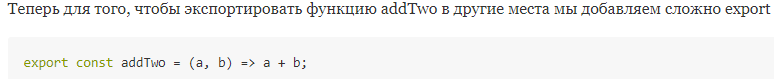
1. Лаконичный синтаксис
2. Модули загружаются синхронно
3. В большинстве случаев используется на сервере

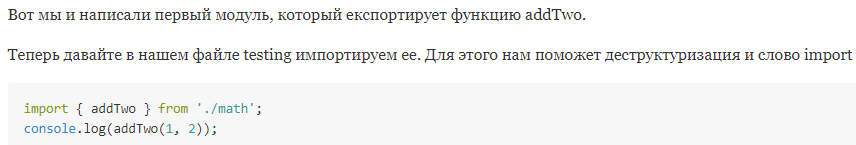
AMD или Asynchronous Module Definition с самой популярной библиотекой RequireJS был предназначен для

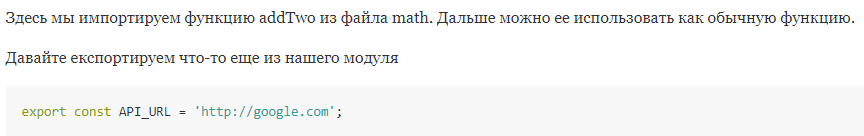
1. Асинхронной загрузки
2. В большинстве случаев использовался на клиенте

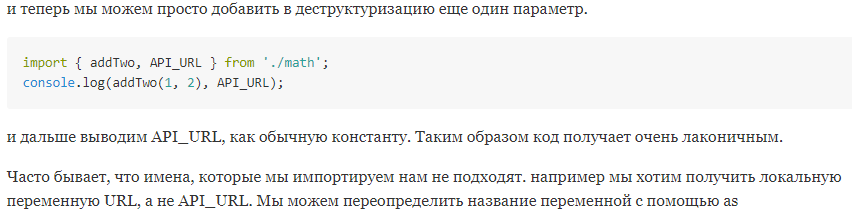
Смысл создания модулей в Ecmascript 6 было создание формата, который был такой же лаконичный как CommonJS и не были такими динамичными. Это дало возможность на этапе компиляции получать ошибки, когда вы пытаетесь импортировать что-то, что не было експортировано. Также тут поддерживается асинхронная загрузка.

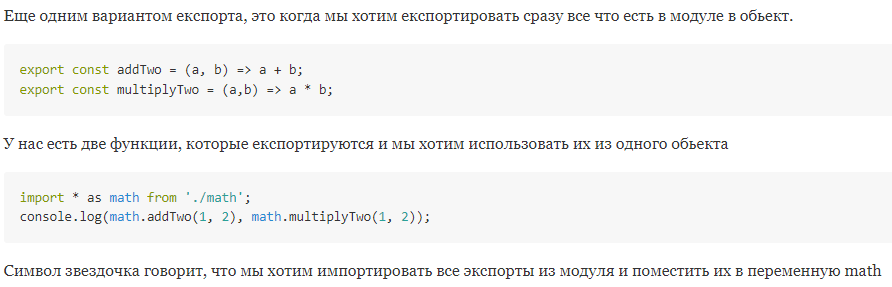
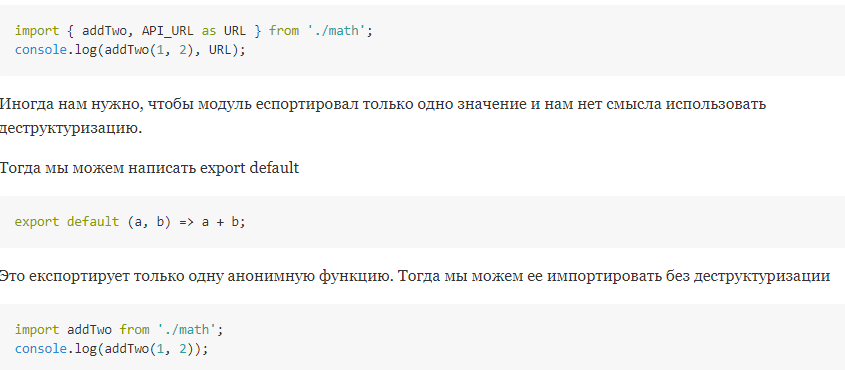












# 11. Пакетный менеджер NPM, глобальное хранилище, просмотр установленных пакетов, скачивание пакетов, назначение файла package.json, локальные хранилища пакетов, удаление пакетов, публикация пакета, SemVer. Примеры

**JavaScript Packet Manager**, устанавливается вместе с Node.js, скачивание/публикация пакетов; инструмент командной строки;

**пакет** – один или несколько js-файлов и файл-манифест package.json(содержит информацию о приложении: название, версия, зависимости;)

**просмотр установленных пакетов**: list, ls, la, ll

## Скачивание пакетов:

Скачивание пакетов можно глобально и локально.

По умолчанию все пакеты устанавливаются в локальном каталоге, в котором мы работаем

### Локальное скачивание:

npm install <package-name>

устанавливается в node-modules нашей папки

### Глобальное скачивание:

npm install <package-name> –g

\*-g означает, что пакет устанавливается глобально и будет доступен для всех приложений

## Удаление пакетов:

### Локального

npm uninstall <package-name>

### Глобального

npm uninstall <package-name> -g

## Публикация пакета:

npm login (npm adduser - зарегистрироваться)

npm init //создать пакет для публикации

npm publish

## Semver

Это семантический версионер.. Semver используется в модуле npm и отвечает за работу с версиями всех модулей для node. js.

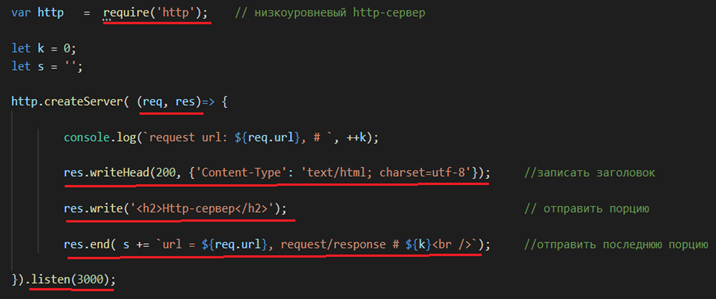
призвано решить проблему **“ада зависимостей” (dependency hell)**. Данное состояние может наступить, когда проект имеет слишком много зависимостей (невозможность обновить пакет без необходимости выпуска новой версии каждой зависимой библиотеки). Решение – нумерация версий.

## package.json:

* файл конфигурации приложения Node.js. Любая директория, содержащая данный файл, интерпретируется как Node.js-пакет.
* Содержит метаданные проекта (название, версия, описание проекта, …), список зависимостей вашего пакета, которые будут установлены при вызове команды npm install, скрипты, вызывающие другие команды консоли.
* создание вручную или команда npm init. Секции name и version – минимальный набор для публикации.

# —12. HTTP-сервер. Порядок разработки простейшего HTTP-сервера в Node.js. События, генерируемые на сервере. Извлечение данных из HTTP-запроса (параметры, тело, заголовки…), формирование данных HTTP-ответа.

## СЕРВЕР



# 13. HTTP-сервер. Порядок разработки простейшего HTTP-сервера в Node.js. Объекты request и response, их свойства, методы, события. Порционная отправка и получение данных.

**Объект request** представляет HTTP-запрос и имеет свойства для строки запроса, параметров, тела, заголовков HTTP и т. д.

## Свойства

req.body - содержит пары "ключ-значение" данных, отправленных в теле запроса.

req.cookies - свойство объекта который содержит файлы cookie, отправленные по запросу.

req.ip - удаленный IP-адрес запроса.

req.path - содержит часть пути URL-адреса запроса.

req.query - объект, содержащий свойство для каждого параметра строки запроса в маршруте.

## Методы

**req.accepts(types)** - проверяет, приемлемы ли указанные типы контента, на основе поля HTTP-заголовка Accept запроса

**req.get(field)** - возвращает указанное поле заголовка HTTP-запроса

**req.is(type)**- возвращает значение true, если поле HTTP-заголовка Content-Type входящего запроса соответствует типу MIME, указанному параметром type.

**Объект response** представляет HTTP-ответ, который приложение Express отправляет, когда получает HTTP-запрос.

## Свойства

**res.app** - содержит ссылку на экземпляр экспресс-приложения, использующего промежуточное ПО.

**res.headersSent** - логическое свойство, указывающее, отправило ли приложение HTTP-заголовки для ответа.

**res.locals** - объект, содержащий локальные переменные ответа, привязанные к запросу.

## Методы

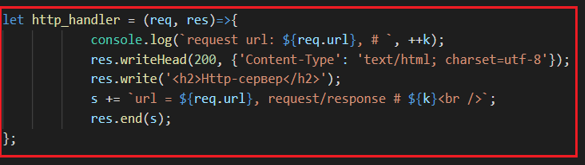
**res.append(field [, value])** - добавляет указанное значение в поле заголовка HTTP-ответа

**res.attachment([filename])**- используется для отправки файла в виде вложения в ответе HTTP

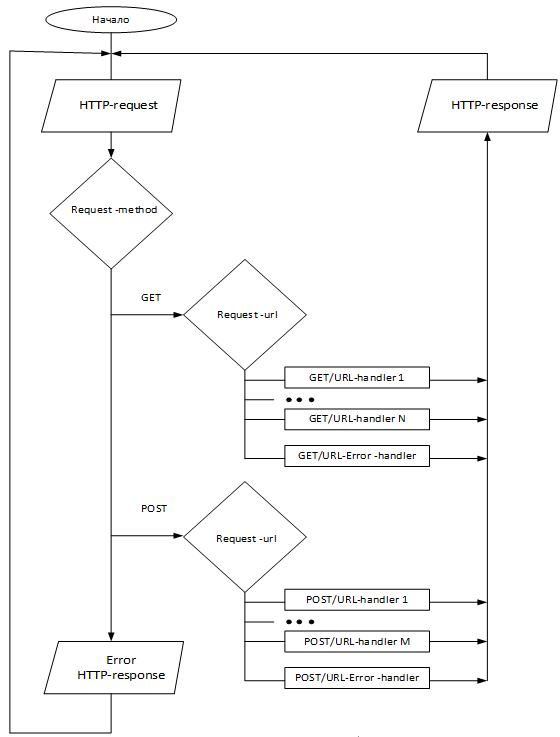
**res.end([data] [, encoding])** - для завершения процесса ответа

**res.format(object)** - согласования содержимого HTTP-заголовка Accept объекта запроса, если он присутствует

## ПОРЦИОНАЛЬНАЯ ОТПРАВКА



# 14. HTTP-сервер. Типичный цикл работы http-сервера, маршрутизация. Статические ресурсы, MIME, отдача статики.



MIME: Multipurpose Internet Mail Extensions

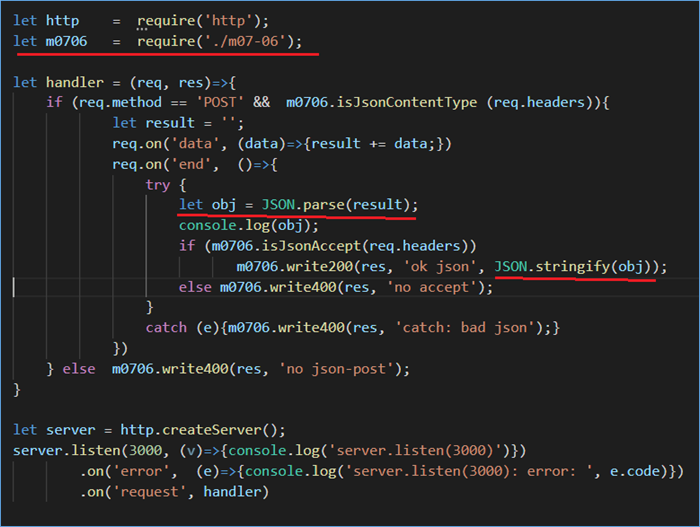
Медиа тип (так же известный как Multipurpose Internet Mail Extensions или MIME тип) является стандартом, который указан природу и формат документа, файла или набора байтов.

Простейший тип MIME состоит из типа и подтипа - двух строк разделённой наклонной чертой ( /), без использования пробелов.

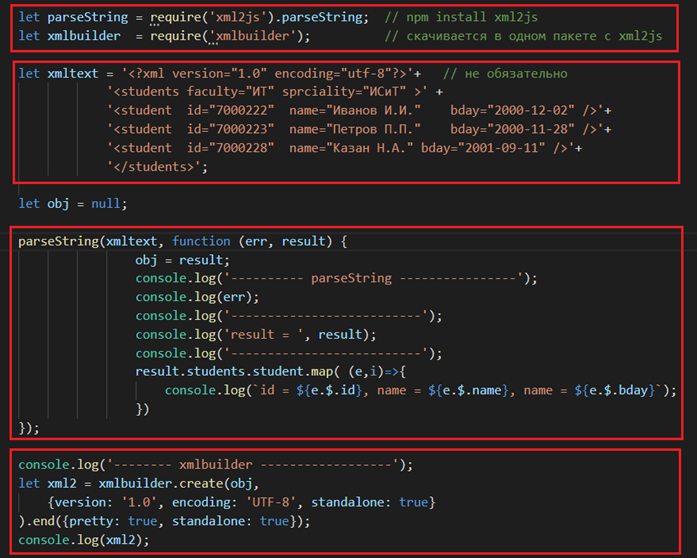
Тип представляет общую категорию, в которой находится тип данных, напримерvideoилиtext. Подтип же строго отождествляется с типом данных, представляемых MIME типом.

# 15. JSON, XML: порядок работы с json- и xml-сообщениями (формирование и разбор). Загрузка файла на сервер (пакет multiparty).

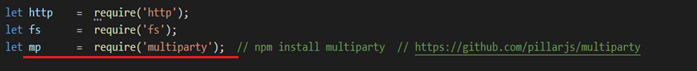
## JSON

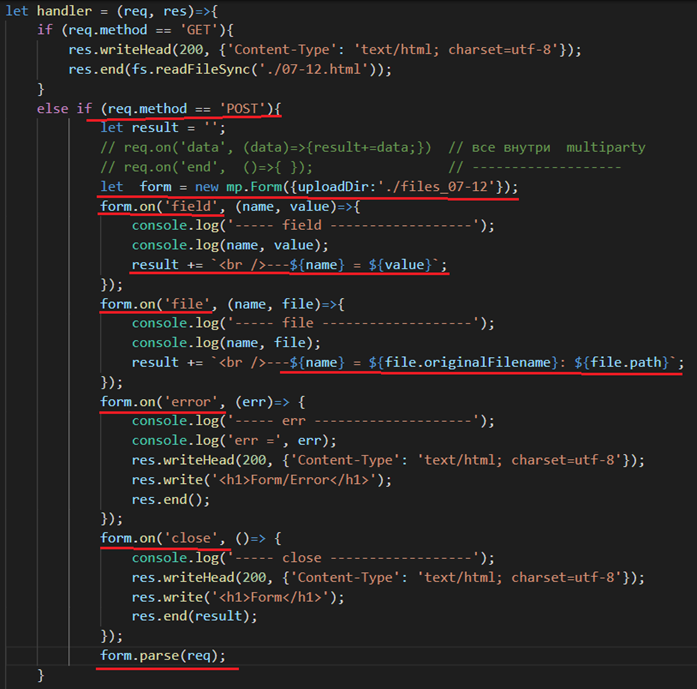


## XML



## MULTIPARTY

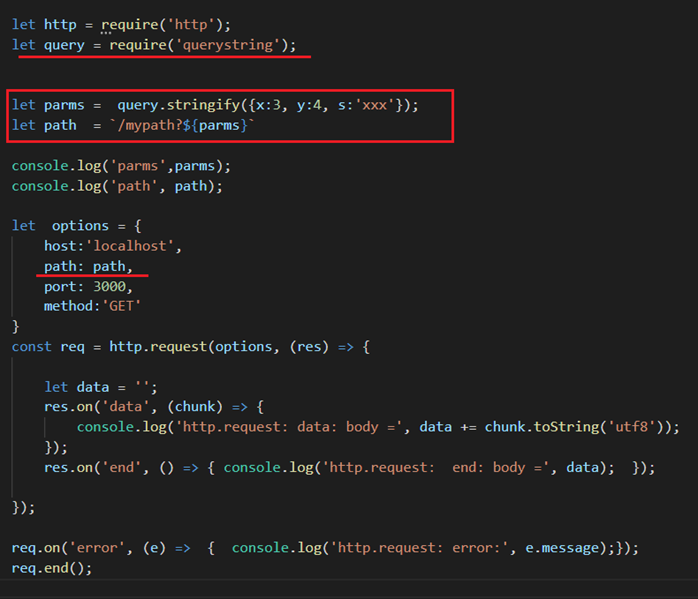




# 16. HTTP-клиент. Порядок разработки простейшего HTTP-клиента в Node.js (модуль http). Отправка GET- и POST-запросов с параметрами, добавление заголовков и обработка ответа. Порционная отправка данных.

HTTP-клиент: это программа, устанавливающая HTTP-соединение с целью отправки HTTP-запросов.

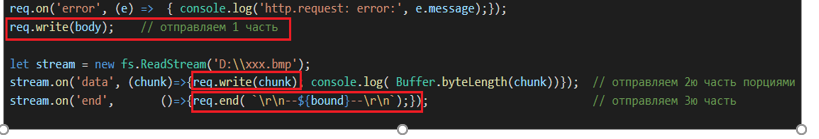
## GET



## POST

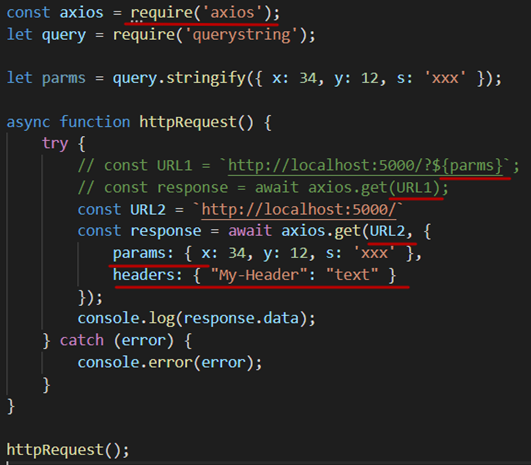


## ПОРЦИОНАЛЬНАЯ ОТПРАВКА

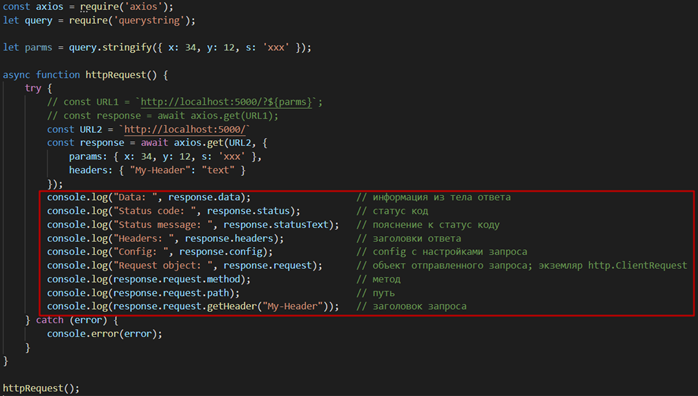


# 17. HTTP-клиент. Порядок разработки HTTP-клиента в Node.js (axios). Отправка GET- и POST-запросов с параметрами, параллельных запросов, добавление заголовков и обработка ответа.

## ЗАГОЛОВКИ

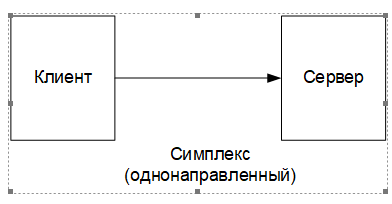


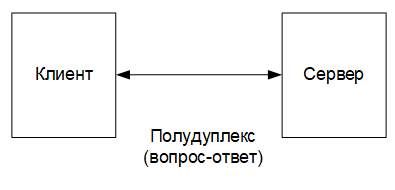
## ОБРАБОТКА ОТВЕТА

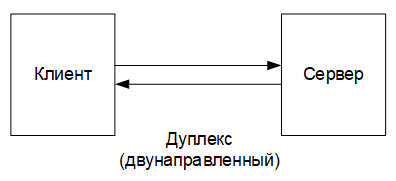


# 18. Websocket: основные свойства, процедура установки соединения, обмен данными, применение. WebSockets API.

**WebSocket** - это протокол передачи данных, основанный на протоколе TCP обеспечивающий обмен сообщениями между клиентом и сервером в режиме реального времени.







## Шаги для установления соединения Web Socket следующие:

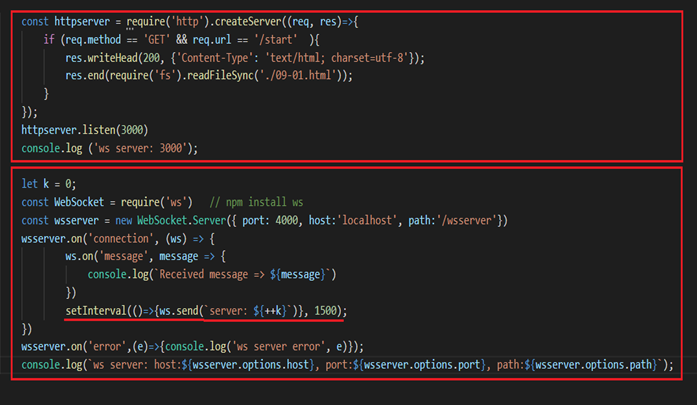
* Клиент устанавливает соединение через процесс, известный как рукопожатие Web Socket.
* Процесс начинается с того, что клиент отправляет на сервер обычный HTTP-запрос.
* Требуется обновление заголовка. В этом запросе он сообщает серверу, что запрос на подключение к веб-сокету.
* URL-адреса веб-сокетов используют схему ws . Они также используются для безопасных соединений через веб-сокеты, которые эквивалентны HTTP.

Как только между клиентом и сервером установлено соединение, событие open запускается из экземпляра Web Socket. Это называется начальным рукопожатием между клиентом и сервером.

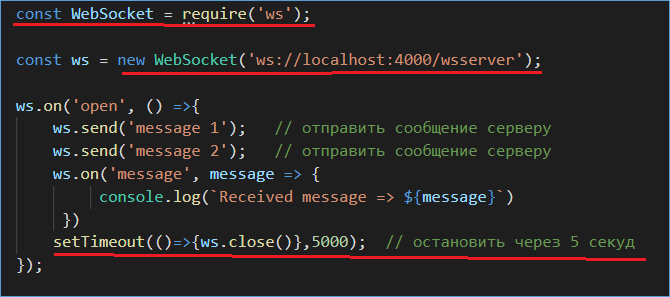
Событие, которое возникает после установления соединения, называется onopen. Создание соединений через Web Socket действительно просто. Все, что вам нужно сделать, это вызвать конструктор WebSocket и передать URL вашего сервера.

# 19. Порядок разработки Websocket-сервера и клиента: использование потоков, ping/pong-сообщения, обработка json-сообщений.

## Сервер



## Клиент



## ping/pong

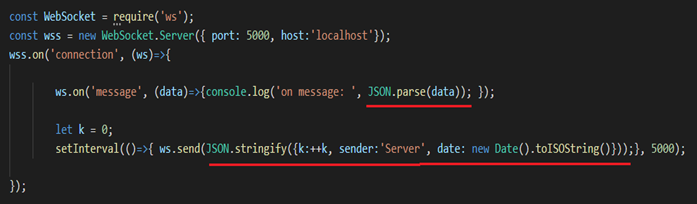
В протокол встроена проверка связи при помощи управляющих фреймов типа PING и PONG.

Тот, кто хочет проверить соединение, отправляет фрейм PING с произвольным телом. Его получатель должен в разумное время ответить фреймом PONG с тем же телом.

Эта функциональность встроена в браузерную реализацию, так что браузер ответит на PING сервера, но управлять ей из JavaScript нельзя.

Иначе говоря, сервер всегда знает, жив ли посетитель или у него проблема с сетью.

## json



# —20. Понятие RPC. Пакет rpc-websockets: порядок разработки RPC-Websockets-сервера и клиента, работа с процедурами, генерация событий и обработка уведомлений.

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

- это технология, которая позволяет программам вызывать процедуры и функции в другом адресном пространстве (как правило на удаленном ПК) то есть на сервере генерируются методы, а клиент их вызывает

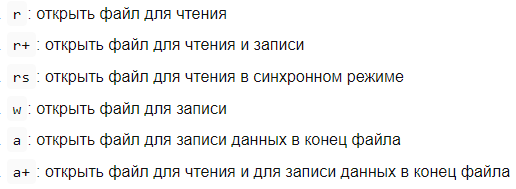
\*npm install rpc-websockets

# 21. Файловая система. Порядок работы с файловой системой в Node.js: создание, копирование, чтение, запись, синхронные и асинхронные операции.

## Создание

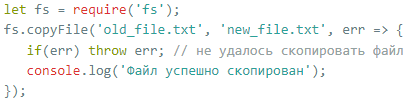
Здесь метод fs.open() используется для создания нового файла. В качестве первого аргумента он принимает имя файла. Его второй аргумент представляет собой флаг, указывающий системе на то, что именно мы хотим сделать с файлом.





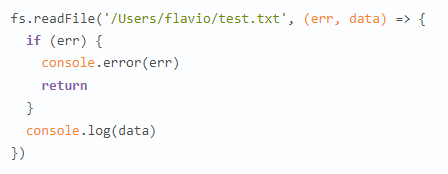
## Копирование

Для копирования используется метод "fs.copyFile". В его первый аргумент передается название файла, который необходимо скопировать, а во втором пишется новый путь.



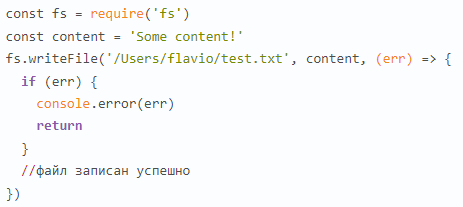
## Чтение

Самый простой способ чтения файлов в Node.js заключается в использовании метода fs.readFile() с передачей ему пути к файлу и коллбэка, который будет вызван с передачей ему данных файла (или объекта ошибки). Существует также синхронная версия этоо метода fs.readFileSync()



## Запись

В Node.js легче всего записывать файлы с использованием метода fs.writeFile(). Синхронная версия того же метода — fs.writeFileSync()

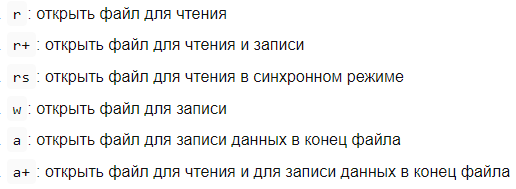


# 22. Файловая система. Порядок работы с файловой системой в Node.js: создание, удаление, переименование, запись в конец, слежение за файлом/директорией, синхронные и асинхронные операции.

## Создание

Здесь метод fs.open() используется для создания нового файла. В качестве первого аргумента он принимает имя файла. Его второй аргумент представляет собой флаг, указывающий системе на то, что именно мы хотим сделать с файлом.





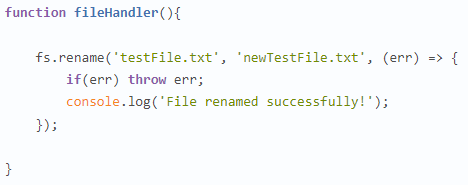
## Удаление

Для удаления файлов используется метод .unlink():



## Переименование

Для переименования файлов используется метод .rename():



Первый аргумент метода представляет собой имя существующего файла, второй — новое имя этого файла. После успешного вызова этого метода файл testFile.txt превращается в newTestFile.txt.

## Запись в конец



используется функция fs.open и вторым параметром предается флаг а(открыть файл для записи данных в конец файла) или а+ (открыть файл для чтения и для записи данных в конец файла)

## Слежение

Используется метод Node.js [fs.watchFile](https://nodejs.org/api/fs.html#fs_fs_watchfile_filename_options_listener) Коллбэк, передаваемый этому методу, будет вызываться при каждом изменении файла. И похожий метод fs.watch()

# 23. Файловая система. Порядок работы с файловой системой в Node.js: работа с директориями, проверка наличия файла/директории, синхронные и асинхронные операции. Объект Buffer и работа с ним.

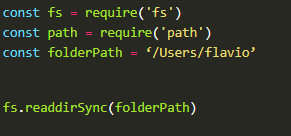
Изначально для работы с файлами или же папками необходимо подключить модуль



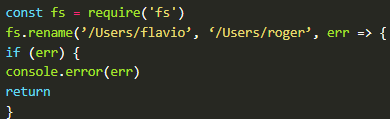
Для создания папки необходимо использовать метод **mkdir(),** а для удаления **rmdir()**

Использовать fs.access() чтобы проверить, существует ли папка и имеет ли узел доступ к ней со своими разрешениями

Использовать **fs.readdir()** или же **fs.readdirSync** для чтения содержимого каталога



Использовать **fs.rename()** или же **fs.renameSync()** переименовать папку. Первый параметр - это текущий путь, второй - новый путь:



Использовать **fs.rmdir()** или же **fs.rmdirSync()** удалить папку.

Метод **remove()** удаление папки с содержимым

# 24. Потоки данных, их виды, примеры. Readable поток: режимы работы, порядок создания и использования Readable потока.

Концепция работы потоков заключается в передаче данных между программами в операциях ввода/вывода. Использование потоков позволяет программе быть независимой и объединяться с другими программами для создания более сложной системы.

Обмен данными осуществляется частями, не дожидаясь полной загрузки, что позволяет эффективно использовать оперативную память.

## Виды потоков:

* **Readable** - поток чтения, извлекающий данные из переданного источника (например, чтение из файла);
* **Writable** - поток записи данных (например, сохранение данных в файл);
* **Duplex** - поток, одновременно реализующий и чтение, и запись;
* **Transform** - разновидность потока Duplex, только извлеченные данные перед записью могут быть изменены.

Поток **Readable** отвечает за чтение данных из определенного источника. Примеры потоков чтения:

* получение сервером HTTP запроса;
* получение клиентом ответа на отправленный HTTP запрос
* чтение из файла;
* получение данных по сокетам и т. д.

Поток Readable работает в одном из двух режимов: **flowing** (процесс извлечения данных) и **paused** (режим ожидания).

По умолчанию все потоки чтения начинают свою работу с режим ожидания. Чтобы начать чтение из источника, необходимо либо добавить обработчик события data, либо вызвать метод [ReadableStreamInstance].resume(), либо отправить поток на запись вызовом метода pipe().

События Node.js потока Readable:

* data - получение потоком данных;
* resume - инициируется при вызове метода resume();
* pause - инициируется при вызове метода pause();
* close - возникает при закрытии источника данных или самого потока;
* end - генерируется, когда из источника считаны все данные;
* error - возникновение в потоке ошибки, обработчику аргументом передается объект ошибки.

# 25. Потоки данных, их виды, примеры. Writable и Duplex потоки: порядок создания и использования.

Концепция работы потоков заключается в передаче данных между программами в операциях ввода/вывода. Использование потоков позволяет программе быть независимой и объединяться с другими программами для создания более сложной системы.

Обмен данными осуществляется частями, не дожидаясь полной загрузки, что позволяет эффективно использовать оперативную память.

## Виды потоков:

* **Readable** - поток чтения, извлекающий данные из переданного источника (например, чтение из файла);

**Примеры:**

* + получение сервером HTTP запроса;
  + получение клиентом ответа на отправленный HTTP запрос
  + чтение из файла;
  + получение данных по сокетам и т. д.
* **Writable** - поток записи данных (например, сохранение данных в файл);

**Примеры:**

* + отправка HTTP запроса с клиента на сервер;
  + отправка ответа на HTTP запрос с сервера на клиент;
  + запись данных в файл;
  + отправка данных по сокету.
* **Duplex** - поток, одновременно реализующий и чтение, и запись;

В Node.js Duplex потоками называются те потоки, которые являются одновременно и Readable и Writable. Примером Duplex потока является передача данных по сокетам: отправка данных одной стороной (запись) и принятие их другой (чтение).

* **Transform** - разновидность потока Duplex, только извлеченные данные перед записью могут быть изменены.

разновидность потоков Duplex, перед отдачей входящие данные видоизменяются. Пример процесс шифрования или расшифровки данных.

# 26. TCP: основные свойства, установка и закрытие соединения. Разработка простейшего TCP-сервера и TCP-клиента на Node.js.

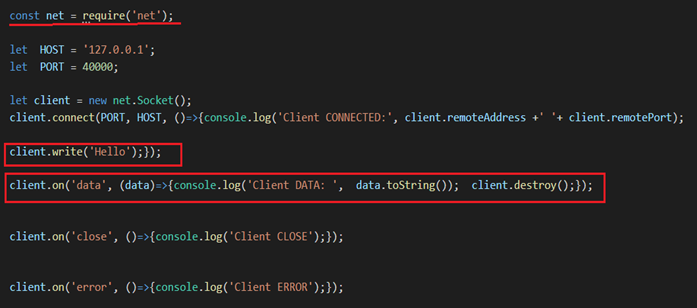
* протокол транспортного уровня для передачи информации.
* надежный
* установка соединения
* подтверждение получения
* правильный порядок отправки
* низкая скорость передачи.

ВОТ ЭТО ОБЯЗАТЕЛЬНО

## Сервер



## Клиент



# 27. UDP: основные свойства, отличия от TCP. Разработка простейшего UDP-сервера и UDP-клиента на Node.js.

протокол транспортного уровня для передачи данных.

ненадежный

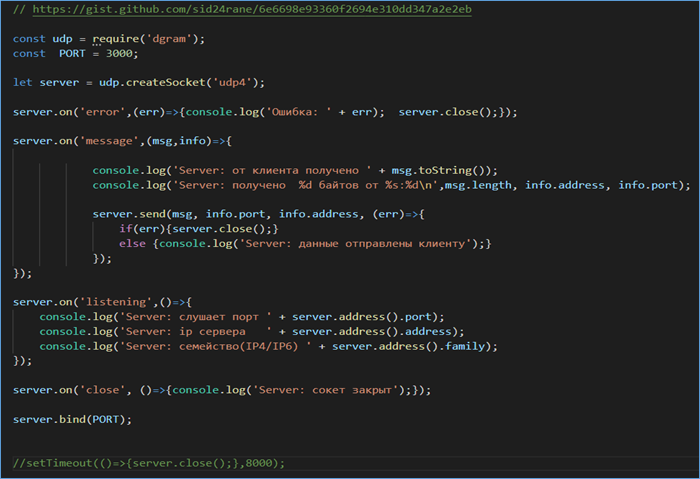
не устанавливает соединение

упорядоченность не соблюдается

более высокая скорость передачи сообщений

отсутствие буфера куда бы помещались сообщения из-за чего они теряются

## Сервер



## Клиент



# 28. MSSQL: пакет mssql, драйверы, подключение, параметры подключения, поддерживаемые типы данных, обычные и подготовленные запросы, обработка результата запроса*,* создание неявной транзакции.

Npm install mssql

Сервер СУБД – доступ через сокет

Для подключения надо знать:

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

\*порт (sql – 1433, oracle - 1521)

\*имя БД

\*логин, пароль

## Некоторые типы данных:

sql.Bit

sql.BigInt

sql.Float

sql.Int

sql.Char

sql.SmallMoney

sql.Xml

sql.Text

sql.Ntext

sql.Time

sql.Date

sql.Binary

sql.Image

## Подготовленные запросы

Это можно описать, как некий вид скомпилированного шаблона SQL запроса, который будет запускаться приложением и настраиваться с помощью входных параметров.



# 29. MSSQL: пакет mssql, драйверы, подключение, параметры подключения, статические и динамические запросы, вызов процедуры, создание явной транзакции.

Npm install mssql

Сервер СУБД – доступ через сокет

Для подключения надо знать:

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

\*порт (sql – 1433, oracle - 1521)

\*имя БД

\*логин, пароль

**Динамический запрос** – значения в запрос подставляются на лету

создается и сохраняется в переменной, пока не возникнет необходимость его выполнения

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

**Статический запрос** – устанавливается во время проектирования и не содержит параметров и аргументов

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

## Транзакция

**транзакция** - это последовательность операций, которые должны быть или все

выполнены или все не выполнены.

**фиксация транзакции** - это действие, обеспечивающее запись на диск изменений в базе данных, которые были сделаны в процессе выполнения транзакции.

**откат транзакции** - это действие, обеспечивающее аннулирование всех изменений данных.



# 30. MSSQL: пакет mssql, потоковый режим обработки данных, пул соединений.

**Потоковая обработка данных** Это – когда решение в реальном времени обрабатывает данные, которые поступают также в реальном времени, в формате потока, генерируются непрерывно.

**Database Connection Pool** — подразумевает, что в нашем распоряжении имеется некоторый набор («пул») соединений к базе данных. Когда новый пользователь запрашивает доступ к БД, ему выдаётся уже открытое соединение из этого пула. Если все открытые соединения уже заняты, создаётся новое. Как только пользователь освобождает одно из уже существующих соединений, оно становится доступно для других пользователей. Если соединение долго не используется, оно закрывается.

это кэш соединений с базой данных, которые поддерживаются таким образом, что соединения могут быть использованы повторно, когда потребуются будущие запросы к базе данных. Пулы соединений используются для повышения производительности выполнения команд в базе данных

# 31. Сравнение SQL и NoSQL, виды NoSQL СУБД, примеры. Репликации, подходы при работе с репликацией.

Реляционные БД хранят структурированные данные, которые обычно представляют объекты реального мира. Скажем, это могут быть сведения о человеке, или о содержимом корзины для товаров в магазине, сгруппированные в таблицах, формат которых задан на этапе проектирования хранилища.

NoSQL (нереляционные) СУБД: подход к реализации масштабируемого хранилища информации с гибкой моделью данных (подходит для неструктурированных данных). В нереляционных базах проблемы масштабируемости и доступности решаются за счёт атомарности и согласованности данных.

NoSQL: виды:

1. хранилище «ключ-значение» (например, Redis, MemcacheDB),
2. документоориентированное хранилище (например, MongoDB),
3. колоночное хранилище (например, Google Big Table, Cassandra),
4. хранилище на основе графов (графовое) (например, InfoGrid, Neo4j).

Не существует баз данных, которые подойдут абсолютно всем. Именно поэтому многие компании используют и реляционные, и нереляционные БД для решения различных задач. Хотя NoSQL-базы стали популярными благодаря быстродействию и хорошей масштабируемости, в некоторых ситуациях предпочтительными могут оказаться структурированные SQL-хранилища.

# 32. MongoDB: структура БД, пакет mongodb, подключение, параметры подключения, выборка, операторы выборки, добавление документа(ов), изменение документа(ов), удаление документа(ов)*,* типы репликаций в MongoDB, транзакции (явные, неявные).

MongoDB — система управления базами данных, которая работает с документоориентированной моделью данных. В отличие от реляционных СУБД, MongoDB не требуются таблицы, схемы или отдельный язык запросов. Информация хранится в виде документов либо коллекций.

npm install mongodb

Свойства:

Кроссплатформенность

Формат данных(на основе js)

Документы, которые хранят значения и ключи.

Вместо таблиц MongoDB использует коллекции.

Репликация. Система хранения информации в СУБД представлена узлами

Индексация.

Подключение осуществляется с помощью метода connect()

который принимает URL, по которому доступна MongoDB (по умолчанию http://127.0.0.1:27017), и callback-функцию, которой передается два параметра:

* ошибка подключения (null, если подключение прошло успешно);
* экземпляр объекта подключения к MongoDB, через API которого создаются новые БД и осуществляется доступ к уже существующим

## Выборка

Наиболее простой способом получения содержимого БД представляет использование функции **find**. Функция f**indOne()** работает похожим образом, только возвращает один документ

## Добавление

**insertOne():** добавляет один документ

**insertMany():** добавляет несколько документов

**insert():** может добавлять как один, так и несколько документов

## Изменение

Если надо полностью заменить один документ другим, может использоваться функция **replaceOne**

Часто не требуется обновлять весь документ, а только значение одного или нескольких его свойств. Для этого применяются функции **updateOne()** - она обновляет только один документ и **updateMany()** - позволяет обновить множество документов.

Для обновления отдельных полей в этих функциях применяется оператор **$set.**

Для простого увеличения значения числового поля на определенное количество единиц применяется оператор **$inc.**

**Репликация** — механизм синхронизации содержимого нескольких копий объекта. **Репликация** — это процесс, под которым понимается копирование данных из одного источника на другой и наоборот.

типы репликаций в MongoDB: Replica Set (в случае падения мастера, оставшиеся сервера реплик автоматически решают кто из них станет мастером) и Master-Slave (здесь необходимо вручную назначать нового мастера).

# 33. Пакет mongoose: понятие ODM, схема, ограничения схемы, валидация, модель, подключение, выборка, создание документа(ов), удаление документа(ов), обновление документа(ов), транзакции (явные, неявные).

mongoose - это ORM для базы данных mongodb для node. ORM - это Object-relational mapping - объектно-документный отображатель. Чтобы работать с mongodb в node мы могли бы использовать mongodb драйвер. Это очень низкоуровневая библиотека, которая просто дает возможность делать обращения к базе. ORM же позволяет создавать модели и зависимости между ними и использовать специальный API для работы с базой. Это, конечно, не так гибко, как выполнять запросы к базе напрямую, но дает возможность лучше структурировать код.

Mongoose содержит восемь SchemaTypes (\* типы данных схемы), которые может иметь свойство, сохраняемое в MongoDB. Эти типы следующие:

String, Number, Date, Buffer, Boolean, Mixed, ObjectId (\* уникальный идентификатор объекта, первичный ключ, \_id), Array

## Валидация

* **required**: требует обязательного наличия значения для свойства
* **min и max**: задают минимальное и максимальное значения для числовых данных
* **minlength и maxlength**: задают минимальную и максимальную длину для строк
* **enum**: строка должна представлять одно из значений в указанном массиве строк
* **match**: строка должна соответствовать регулярному выражению

## Создание

save - этот метод определен для всех создаваемых моделей, он сохраняет текущий объект в базу данных:

## Подключение

mongoose.connect('mongodb://localhost/mongoose\_basics');

## Выборка

* **find**: возвращает все объекты, которые соответствуют критерию фильтрации
* **findById**: возвращает один объект по значению поля \_id
* **findOne**: возвращает один объект, который соответствует критерию фильтрации

## Удаление

Для удаления применяется метод **deleteOne()** (удаляет один объект) и **deleteMany()** (удаляет все объекты, которые соответствуют критери.)

## Обновление

Для обновления данных в модели предусмотрены методы updateOne() и updateMany().

Еще для обновления используется фильтрация по \_id - используется метод findByIdAndUpdate():

# 34. REST: определение, достоинства и недостатки, форматы передачи данных, ограничения, HATEOAS.

Representational State Transfer - передача состояния представления

REST - это набор правил о том, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

Основная идеи REST API заключается в разделении разных операций (чаще всего CRUD) при обращении к одному и тому же URL с помощью HTTP методов, основные из которых:

* GET - используется для получения данных;
* POST - используется для создания новой записи(ей);
* PUT - используется для обновления уже существующей записи(ей);
* PATCH - используется для обновления, но только тогда, когда изменяется идентификатор записи(ей);
* DELETE - используется для удаления записи(ей).

## Форматы

Несмотря на то, что REST не является стандартом сам по себе, большинство RESTful-реализаций используют такие стандарты, как HTTP, URL, JSON и, реже, XML.

## Ограничения

1. Клиент-сервер
2. В период между запросами на сервере не хранится никакой информации о состоянии клиента
3. Кэширование
4. Многоуровневая система
5. Код по требованию
6. Единообразие интерфейса

## Достоинства

- производительность (кэширование);

- надежность (отсутствие состояния);

- простота (унифицированность интерфейса, использование повсеместных стандартов,…);

- изменяемость;

- масштабируемость.

## Недостатки

- нет общепризнанного стандарта RESTful API;

- не все браузеры поддерживают полный словарь REST-методов (PUT, DELETE); на практике часто используется только GET и POSТ(insert, delete, update);

- не однозначны коды состояний.

## 

# 35. REST: определение, достоинства и недостатки, общепринятые правила REST API, HATEOAS.

Representational State Transfer - передача состояния представления

REST - это набор правил о том, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

Основная идеи REST API заключается в разделении разных операций (чаще всего CRUD) при обращении к одному и тому же URL с помощью HTTP методов, основные из которых:

1. GET - используется для получения данных;
2. POST - используется для создания новой записи(ей);
3. PUT - используется для обновления уже существующей записи(ей);
4. PATCH - используется для обновления, но только тогда, когда изменяется идентификатор записи(ей);
5. DELETE - используется для удаления записи(ей).

## Достоинства

- производительность (кэширование);

- надежность (отсутствие состояния);

- простота (унифицированность интерфейса, использование повсеместных стандартов,…);

- изменяемость;

- масштабируемость.

## Недостатки

- нет общепризнанного стандарта RESTful API;

- не все браузеры поддерживают полный словарь REST-методов (PUT, DELETE); на практике часто используется только GET и POSТ(insert, delete, update);

- не однозначны коды состояний.

## Правила

* Общий префикс для всех ресурсов сервиса …/api/…

http://bstu.by/api/...

* Два типа ресурсов: коллекция (users, students, …), элемент коллекции …/api/users/238, …/api/students/ef3d26.
* Иерархическая связь …/api/users/238/cars/aah4899
* Использовать существительные во множественном числе. Если несколько слов, то использовать kebab-case

…/api/customers/33245/delivery-addresses

* Использовать HTTP статус коды, сопроводить сообщение дополнительным кодом (например 20003, 404001,…), сделать отдельный ресурс (HATEOAS link) для пояснения ошибок<http://ccc/api/errors/20003>.
* Подавление статуса ответа

…/api/students/ef3d26?status\_code=200.

* Версионность

…/api/v7/students/ef3d26 или …/api/students/ef3d26?v=7.

* Постраничное получение данных: параметры limit, offset.

…/api/students?offset=10&limit=5

* Сортировка: параметр sort.

…/api/students?sort=+group,+name

* Все фильтры вынести за знак вопроса:

…/api/students?minbday=19980101&maxbday=20001231&gender=m.

* Пользователь получает только то, что хочет: …/api/students?field=bday,surname,gender.
* Обозначать в запросе формат сообщений (желательна поддержка нескольких форматов, рекомендуется JSON и XML):

…/api/students.json?field=bday,surname,gender; один из форматов должен быть по умолчанию; могут применяться заголовки Accept и Content-Type со значениями application/xml и application/json для запроса или обозначения в ответе формата.

* Глобальный поиск: …/api/search?q=19600107+Иванов.
* Документация.

# 36. GraphQL: преимущества и недостатки, особенности, схема, контекст, типы, модификаторы типов, контекст, резолверы, аргументы и переменные.

GraphQL: формальный язык запросов;

**особенности:** позволяет делать агрегированные запросы (можно в одном запросе запросить все необходимые данные), использует систему типов

## Преимущества GraphQL

1. **Все нужное и ничего лишнего**. В GraphQL вы получаете только то, что просите, и больше ничего.
2. **Быстрая разработка на клиенте**. Если нужно изменить что-то в данных, достаточно лишь немного подправить запрос. Это позволяет быстрее развивать продукт. Клиентскую и серверную разработку можно вести независимо друг от друга, при условии, что обе команды знают структуру данных.

## Ограничения GraphQL

1. **Сложность**. Для небольших простых приложений настройка таких запросов может показаться слишком сложной и ненужной. В этой ситуации легко можно обойтись классическим REST-подходом.
2. **Нет кеширования**. GraphQL использует всего одну конечную точку, что не позволяет следовать спецификации HTTP-кеширования. Это очень важно, так как кеширование уменьшает объем трафика.

**GraphQL Schema** – это описание ваших типов данных на сервере, связей между ними и логики получения этих самых данных

**context** - дает возможность работать с различными «контекстными» данными

GraphQL поддерживает **четыре типа данных:** String , Int , Float , Boolean . Если в поле указан восклицательный знак, оно становится обязательным

* Output-типы - для описания ответа от сервера.
* Input-типы - для описания входящих параметров от клиента.

**Resolver или распознаватель** — функция, которая возвращает данные для определенного поля. Resolver’ы возвращают данные того типа, который определён в схеме.

**Аргументы** позволяют сделать очевидные вещи — подставить свои значения и сделать запрос. Абсолютно не важно будет это query, mutation или subscription.

Сначала определяем, какие аргументы нужно использовать. Далее, применяем их к полю запроса.

**Переменные GraphQL** — это способ динамического указания значения, которое используется внутри запроса

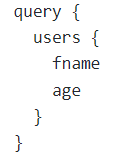
# 37. GraphQL: архитектура, выполнение graphql-запроса, порядок разработки graphql-сервера

**GraphQL** — это стандарт декларирования структуры данных и способов получения данных, который выступает дополнительным слоем между клиентом и сервером. Одной из основных фичей GraphQL является то, что структура и обьем данных определяется клиентским приложением.

## Query: запросы в GraphQL

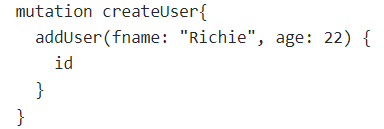
С помощью запросов GraphQL получает необходимые данные с сервера. Тип запроса **Query в GraphQL** — аналог GET в REST. Запросы — строки, которые отправляются в теле HTTP POST-запроса.

Query описывает данные, которые необходимо получить с сервера.



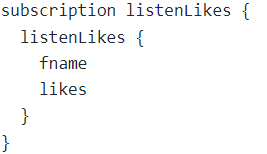
## Mutation: мутации в GraphQL

**Mutation** — ещё один root types. С его помощью можно добавлять данные в БД. Mutation — аналог POST и PUT в REST.



## Subscription: подписки в GraphQL

**Subscription** — третий тип операций в GraphQL. С его помощью клиент слушает изменения в БД в режиме реального времени. Под капотом подписки используют вебсокеты.



1. Работа с сервером GraphQL всегда начинается с разработки схемы (Schema)
2. Чтобы сервер мог работать, необходимо определить типы.
3. После определения типов необходимо добавить их логику. Это нужно, чтобы сервер знал, как отвечать на запросы клиента. Resolver’ы возвращают данные того типа, который определён в схеме.

# 38. GraphQL: field-резолверы, enum, interface, union, fragment.

**Resolver или распознаватель** — функция, которая возвращает данные для определённого поля. Resolver’ы возвращают данные того типа, который определён в схеме. Распознаватели могут быть асинхронными.

**Типы перечисления**, представляют собой особый вид скаляров, который ограничен определенным набором допустимых значений. Это позволяет:

* Убедитесь, что любые аргументы этого типа являются одним из допустимых значений
* Сообщите через систему типов, что поле всегда будет одним из конечного набора значений

**Интерфейсы** полезны, когда вы хотите вернуть объект или набор объектов, но они могут быть нескольких разных типов.

**Типы Union** очень похожи на интерфейсы, но они не могут указывать какие-либо общие поля между типами.

**Фрагмент** [**GraphQL**](http://graphql.org/learn/queries/#fragments) - это часть логики, которая может использоваться несколькими запросами и изменениями. Каждый фрагмент включает в себя подмножество полей, принадлежащих соответствующему типу.