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:** 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.

**Request-методы:**

****

**Заголовки:**









**Response: Код состояния:**



**Веб-приложение** — клиент-серверное приложение, в котором клиент взаимодействует с веб-сервером при помощи браузера

**Структура:**

1. 1 клиент – 1 сервер

2. 1 клиент – несколько серверов

3. Несколько клиентов – 1 сервер

4. Клиенты - промежуточный сервер – сервер

(Картинки в первой лекции))0) )

**Асинхронность**

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.

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

**Веб-сервер —** сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными.

**Ресурсы**

Оперативная память, процессорное время.

**Блокирующие и неблокирующие операции**

**Блокирующий ввод-вывод** называется так, потому что поток, который его использует, блокируется и переходит в режим ожидания, пока ввод-вывод не будет завершён. При неблокирующих операциях ввод-вывода ситуация обстоит обратным образом.

Блокирующие методы выполняются синхронно, а неблокирующие асинхронно.

**Решение**

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

- асинхр прогр – паттерн **Reactor** – в Nginx

паттерн **Reactor** – шаблон проектирования. Используется при обработке параллельных запросов к сервису. Сервисный обработчик разбирает прибывшие запросы и синхронно перенаправляет их на соответствующие обработчики запросов.

**Конкурентность** - это способность вычислительной системы справляться с множеством задач одновременно, предоставляя доступ к своим ресурсам в соответствии с приоритетом запросов, в то время как параллелизм - способность выполнять несколько задач одновременно. Параллели́зм — это свойство систем, при котором несколько вычислений выполняются одновременно, и при этом, возможно, взаимодействуют друг с другом.

**Закон Амдала**

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 медленного фрагмента»

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

1. 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.

- первая версия: 2009 г.;

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

- версионирование: две ветки LTS(Long time support) и Current(последняя).

**Свойства**

- основан на Chrome V8;

- среда (контейнер) исполнения приложений на JavaScript;

- поддерживает механизм асинхронности;

- ориентирован на события;

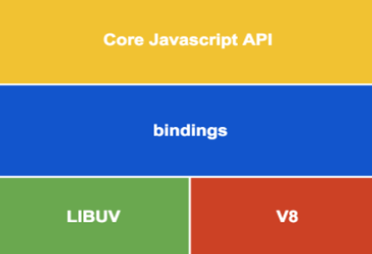
- однопоточный (код приложения исполняется только в одном потоке, один стек вызовов); обычно в серверах для каждого соединения создается свой поток, в Node.js все соединения обрабатываются в одном JS-потоке;

- 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;

**Структура**

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.



**Принцип работы**

**V8 -** движок от гугла для парсинга и исполнения js;

**libuv -** библиотека, реализующая event loop;

**node -** сам бинарник, который связывает v8 и libuv и позволяет коду на js работать на сервере.

**Event loop -** это, по сути, бесконечный цикл, который на каждую итерацию обрабатывает произошедшие события. Чтобы цикл был производителен, он должен работать асинхронно и не блокироваться. Для асинхронной работы так же необходимо передавать колбэк, который будет вызван по завершению какой-то функции иначе будет невозможно узнать, когда операция закончится.

V8 использует несколько потоков для Google Chrome, а libuv использует thread pool для выполнения некоторых синхронных операций и системных вызовов.

Thread pool - это вечноживущие потоки, на которые перебрасываются операции, которые могут заблокировать event loop. После того, как операция закончится на потоке, колбэк, который необходимо выполнить по завершению, будет помещен в очередь в event loop. Размер пула равен переменной окружения UV\_THREADPOOL\_SIZE (по умолчанию 4).

Nodejs использует только 1 event loop (за исключением worker threads) и поскольку это просто синхронный цикл, то он ограничен одним потоком.



**Модули**

Встроенные модули:

* **fs –** файловая система;
* **buffer -**  для работы с двоичными данными:
* **crypto –** для работы с функциями OpenSSL;
* **http (https) –** для работы как http и https сервер;
* **querystring –** для работы с URL строками запросов;
* **url –** парсинг URL строк;
* **util –** для доступа к служебным функциям.

**Внешние модули**

Npm там тыры-пыры

1. 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.

**Глобальные объекты:** global, process – предоставляют переменные и функции, доступные в любом месте программы, во всех модулях. Нет необходимости в require.

Node.js предоставляет специальный объект global, который предоставляет доступ к глобальным, то есть доступным из каждого модуля приложения, переменным и функциям. Примерным аналогом данного объекта в javascript для браузера является объект window.

Объект process принадлежит к числу важнейших компонентов среды Node, так как он предоставляет информацию о среде выполнения. Кроме того, через объект process выполняется стандартный ввод/вывод, можно корректно завершить приложение Node.

**Потоки**

Стандартные потоки представляют собой заранее определенные каналы передачи данных между приложением и средой: стандартный ввод(**stdin**), стандартный вывод(**stdout**) и стандартный поток ошибок(**stderr**).

**Консоль**

Модуль console предоставляет простую консоль отладки, аналогичную механизму консоли js, предоставляемому веб-браузерами.

- Console.log() – выводит соо в консоль.

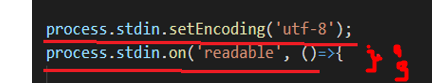
- Console.error() - Выводит сообщение об ошибке.

- Console.dir() - Отображает интерактивный список свойств указанного объекта JavaScript.

- Console.time() - Запускает таймер, который вы можете использовать для определения, сколько времени занимает выполнение той или иной операции.

- Console.timeEnd()-Останавливает указанный таймер и записывает в лог, сколько прошло секунд от его старта.

- Console.trace()-Вывод трассировки стека.

Примеры:  

1. 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***.

**Callback:** — функция, которая передается в качестве параметра другой функции и которая **будет вызвана** асинхронно обработчиком событий **после завершения задачи**.

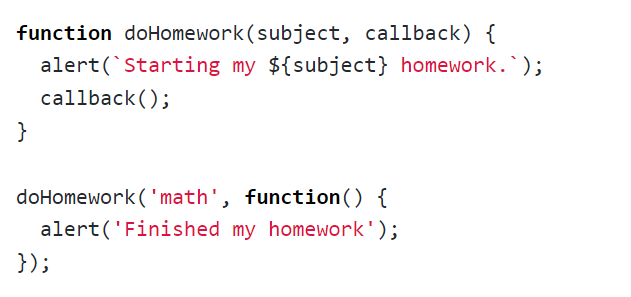
**Callback:** Node.js style: callback принимает два параметра – информацию об ошибке и данные.

**Callback Hell** - это паттерн для управления конкурирующими (асинхронными) запросами, который обеспечивает последовательность их выполнения.

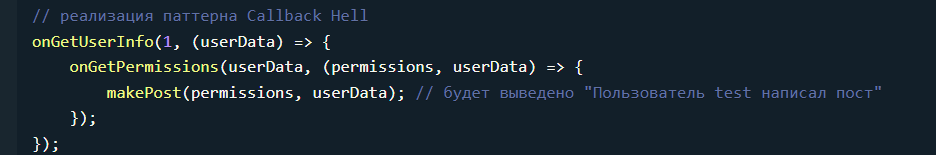
**Callback hell,** pyramid of doom: решение: 1) разбивать на функции; 2) разбивать на модули.

Асинхронная функция в паттерне Callback Hell не может возвращать значение, так как такие функции выполняются с задержкой и позже кода, который за ними последует. Вместо этого они принимают коллбек, который будет запущен, когда функция выполнит свою работу. Как правило таких коллбеков два - один для случая успешного выполнения, а второй для обработки возникших ошибок. Рассмотрим упрощенный пример

Решение проблемы Callbacke Hell – Promises

Пример Callback 

Пример Callbacl hell



реализация(не обязательно диктовать)

**Асинхронное программирование:** 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 выполение функция обратного вызова. В основном так работают все веб-API.

1. Асинхронное программирование. Механизм Promises. Механизм async/await. Примеры.

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

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

resolve(value) — если работа завершилась успешно, с результатом value.

reject(error) — если произошла ошибка, error – объект ошибки.

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

**Promise**: Cвойствo **state**:

**pending** (ожидание),

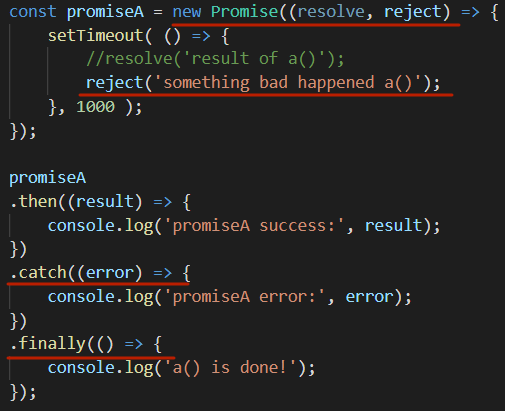
**fulfilled** (выполнено) при вызове resolve,

**rejected** (отклонено) при вызове reject.

Свойство **result**:

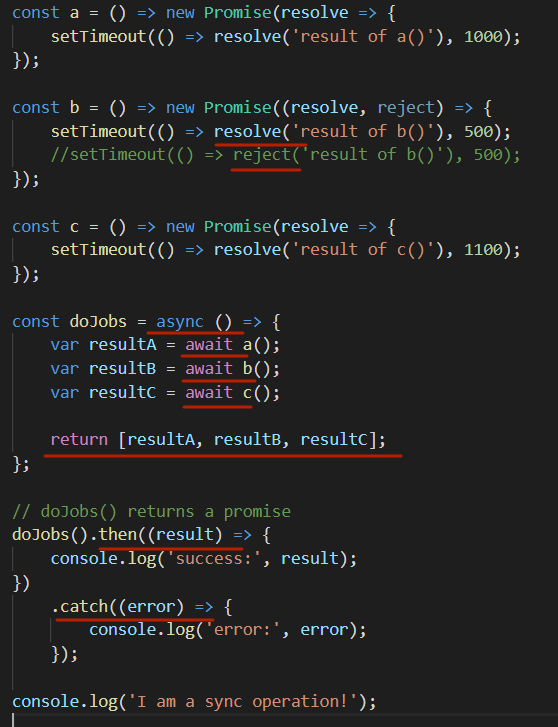
вначале undefined,

далее изменяется на value при вызове resolve(value)

или на **error** при вызове reject(error). 

**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.



1. Класс EventEmitter, назначение, применение. Пример.

**EventEmmitter**

JS-класс, предоставляющий функциональность для асинхронной обработки событий в Node.js. Событие в программном объекте – это процесс перехода объекта из одного состояние в другое. При этом, об этом переходе могут быть извещены другие объекты. У события есть издатель (или генератор) события и могут быть подписчики (или обработчики) события.

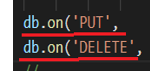
Для работы необходимо два модуля: util и events.

как правило, применяется в качестве базового для пользовательского объекта. Производный от EventEmitter объект может быть создан с помощью функции inherits модуля utils. Производный от EventEmitter объект приобретает функциональность, позволяющую генерировать и прослушивать события.

****

EventEmitter: для наследования можно использовать ключевое слово extends.

EventEmitter: для генерации событий предназначена функция emit, а для прослушивания функция on.

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

**Таймер** - механизм, позволяющий генерировать событие или выполнить некоторое действие, через заданный промежуток времени. setTimeout(), setInterval(); реализованы библиотекой libuv.

**setTimeout**(): выполняется только один раз через некоторый промежуток времени.

**setInterval**(): выполняется регулярно через некоторый промежуток времени.

**clearTimeout**(): останавливает таймер, созданный с помощью setTimeout(). Параметр – ID таймера, который необходимо отменить.

**clearInterval**():останавливает таймер, созданный посредством setInterval(). Параметр – ID таймера, который необходимо отменить.

**NextTick**() - чтобы отложить выполнение функции до следующего итерации цикла событий

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.

1. 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:

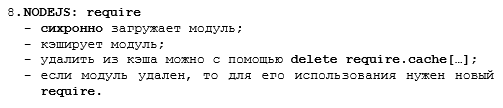
- поддержка require для импорта модуля

- 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.

**Require**



Модули кэшируются после первой загрузки. Это означает (среди прочего), что каждый вызов require ('foo') будет возвращать один и тот же объект , если он будет разрешен в один и тот же файл.

**Область видимости в пакете**

Внутри пакета доступны локальные, глобальные и process переменные

Если пакет подключен, то будут видны только глобальные и process переменные, которые определены в подключаемом пакете. Локальные переменные доступны только внутри подключаемого пакета если они не экспортированы с использованием ключевого слова exports.

**Экспорт объектов, функций, конструкторов**

Объект

Exports.someObj = {somePar : ‘hello’}

Функции

Exports.someFunc = function someFunc(){console.log(hello)};

Конструкторы

function MyObject(bar) {

this.bar = bar;

}

MyObject.prototype.foo = function foo() {

console.log(this.bar);

};

module.exports = MyObject;

**Require для json**

var obj = require('data.json');

console.log(obj.name);

**Параметризируемый модуль**

var modules = {  
 A: require('./A'),  
 B: require('./B'),  
 C: require('./C')  
}

function X(impl) {  
 if(impl in modules)  
 return new modules[impl];  
 else  
 throw new Error('Unknown impl: ' + impl);  
}

module.exports = X;

var foo = new X('A');  
foo.method();  
// => 'A'  
var bar = new X('B');  
bar.method()  
// => 'B'

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

**ВЕСЬ БИЛЕТ ЗАПИСЫВАТЬ**

**Модули** – блоки кода, которые могут использоваться повторно в других модулях.

**Форматы модулей**: CommonJS, AMD, UMD, ES6

**Экспорты**

Экспорт по умолчанию: export **default** function…

Экспорт до объявления: export let variable\_1 = …

Экспорт именованный: export {SomeFunc as func1, SomeFunc2 as func2}

Экспорт после объявления: export {SomeFunc}

**Импорт**

Импорт определенных функций: import {SomeFunc} from ‘some.js’

Импорт всего, как объекта: import \* as some from ‘some.js’

Динамический импорт:

Конструкция вида – import(modulePathVar).then(obj => …)

import() возвращает промис, результатом которого является объект модуля.

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

**NPM** – node package manager – позволяет с легкостью скачивать модули и управлять зависимостями проекта.

**Глобальное хранилище** – директория (node\_modules), куда npm устанавливает пакеты, которые были скачаны с тэгом –g (global). Если пакет имеет свой исполняемый файл, то он будет доступен из командной строки. Директория находится в корне платформы node.js.

**Локальное хранилище** – директория (node\_modules) внутри проекта. Сюда скачиваются модули и зависимости, для которых не был указан тег –g.

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

**Скачивание пакетов:** i, install, add, install <module>@<version>

**Удаление пакета:** npm uninstall, remove, rm, r, un, unlink

**Package.json**

Package.json – файл с метаданными проекта, который позволяет управлять зависимостями проекта. Так же там хранится такая информация как имя проекта, имя разработчика, версия, скрипты, лицензия, точка входа в проект, описание.

**Публикация пакета**

- Зарегистрироваться на сайте npmjs.com

- авторизоваться через npm (npm login)

- инициализировать проект, если это еще не было сделано (npm init)

- опубликовать проект (npm publish)

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

semver.valid('1.2.3') // true — возвращает разобранную версию или null, если версия не валидна

semver.clean(' =v1.2.3 ') // '1.2.3'

semver.satisfies('1.2.3', '1.x || >=2.5.0 || 5.0.0 - 7.2.3') // true

semver.gt('1.2.3', '9.8.7') // false v1 > v2

semver.lt('1.2.3', '9.8.7') // true v1 < v2

1. 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.

**Веб-сервер** — сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными.

**Порядок разработки**

- Импорт модуля http: const http = require(‘http’);

- создание функции, которая будет срабатывать при обращении к серверу (request listener)

- создание объекта сервера: cosnt server = http.CreateServer(<request listener>);

- задать параметры прослушивания сервера: server.listen(<port>, <address>, <callback>);

**События**

**Abort** – выдается, когда запрос прерван клиентом

**Connect** – выдается, когда сервер отвечает на запрос с помощью connect метода.

**Continue** – генерируется, когда сервер отправляет ответ «100 continue»

Information – Выдается, когда сервер отправляет промежуточный ответ 1xx за исключением 101. лушатели этого события получат объект, содержащий версию HTTP, код состояния, сообщение о состоянии, объект заголовков значений ключа и массив с именами необработанных заголовков, за которыми следуют их соответствующие значения.

**Response** - Выдается при получении ответа на этот запрос. Это событие генерируется только один раз.

**Socket** - Это событие гарантированно будет передано экземпляру класса {net.Socket}, подкласса {stream.Duplex}, если пользователь не укажет тип сокета, отличный от {net.Socket}.

**Timeout** - Генерируется, когда базовый сокет выходит из строя по тайм-ауту. Это только уведомляет о том, что сокет бездействует. Запрос необходимо уничтожить вручную.

...

**Извлечение данных**

Что бы получить заголовки, можно использовать request.headers где request – объект запроса.

Метод – request.method

Версия http – request.httpVersion

Url – request.url

Что бы получить тело запроса, можно обработать событие ‘data’ объекта request. Мы получаем данные чанками, по этому нужно использовать следующую конструкцию:

Let body = “”;

Request.on(‘data’,(data) => { body += data });

**Формирование данных**

- заполнение заголовков response.setHeader(“Content-Type”, “text/html”);

- установка статус-кода: response.statusCode(200);

- установка статус-сообщения: response.statusMessage(“sdfsdf”);

- запись содержимого: response.write(<h1>Hello<h1>);

- сигнал серверу, что ответ готов к отправке клиенту: response.end();

Response.end должен вызываться для каждого запроса.

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

**Веб-сервер —** сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными.

**Порядок разработки**

- Импорт модуля http: const http = require(‘http’);

- создание функции, которая будет срабатывать при обращении к серверу (request listener)

- создание объекта сервера: cosnt server = http.CreateServer(<request listener>);

- задать параметры прослушивания сервера: server.listen(<port>, <address>, <callback>);

**Запрос (request)**

Объект запроса: http.IncomingMessage

Свойства:

- url – (uri)

- method – http-method

- httpVersion – версия HTTP

- headers – массив заголовков

-complete – true если сообщение было получено и успешно распаршено

- statusCode

- statusMessage

- host

Методы:

- destroy() – уничтожает запрос

Cобытия:

- aborted – выдается, когда запрос был прерван (deprecated)

- close – указывает, что соединение было закрыто (deprecated)

**Ответ (response):**

Объект запроса: http.ServerResponse

Свойства:

- finished – true если был вызван метод end()

- headersSent – true если заголовки были отправлены

- req – ссылка на оригинальный объект запроса

- sendDate – if true то будет сгенерирован заголовок даты и отправлен если он еще не присутствует

- statusCode

- statusMessage

Методы:

- flushHeaders – очищает заголовки

- getHeader(name)

- getHeaderNames

- getHeaders

- hasHeader

- setHeader

- wrire()

- end()

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

**Веб-сервер** — сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными.

**Цикл работы**



**Маршрутизация**

Маршрутизацию можно реализовать несколькими способами. Самым оптимальным будет разбить обработку запроса по различным методам на так называемые handlers. Handler`ом будет выступать отдельный js файл, который экспортируется как модуль и в который мы будем передавать объект запроса и ответа. Далее внутри этих handler`ов мы будем проверять по какому пути идет обращение и выполнять в зависимости от этого определенные действия.

Пример:

const getHandler = require('./resourse/getHandler');

const postHandler = require('./resourse/postHandler');

const putHandler = require('./resourse/putHandler');

const deleteHandler = require('./resourse/deleteHandler');

http.createServer((req, res) => {

    switch(req.method) {

        case 'GET': getHandler(req, res, Db); break;

        case 'POST': postHandler(req, res, Db); break;

        case 'PUT': putHandler(req, res, Db); break;

        case 'DELETE': deleteHandler(req, res, Db); break;

    }

}).listen(5000);

const url = require('url');

module.exports = (req, res, Db) => {

    let path = url.parse(req.url).pathname;

    let data\_json = '';

    switch(path)

    {

        case '/api/faculties':

            req.on('data', chunk => {

                data\_json += chunk;

            });

            req.on('end', () => {

                data\_json = JSON.parse(data\_json);

                res.writeHead(200, {'Content-Type': 'application/json'});

                Db.InsertRecords('faculty','faculty',data\_json.faculty, data\_json).

                then(records => res.end(JSON.stringify(records))).catch(error => {write\_error\_400(res, error)});

            });

            break;

        case '/api/pulpits':

…

**Статические ресурсы**

Работа со статическими ресурсами это одна из важных задач сервера. Для того, что бы отправить клиенту статический ресурс нужно его считать. Это можно сделать с помощью функции readFile модуля fs, который отвечает за работу с файловой системой. Так же перед отправкой стоит указать MIME тип ресурса, который мы отправляем. Делается это с помощью заголовка Content-Type

Пример:

http.createServer(function (req, res) {

fs.readFile(\_\_dirname + req.url, function (err,data) {

if (err) {

res.writeHead(404);

res.end(JSON.stringify(err));

return;

}

res.writeHead(200);

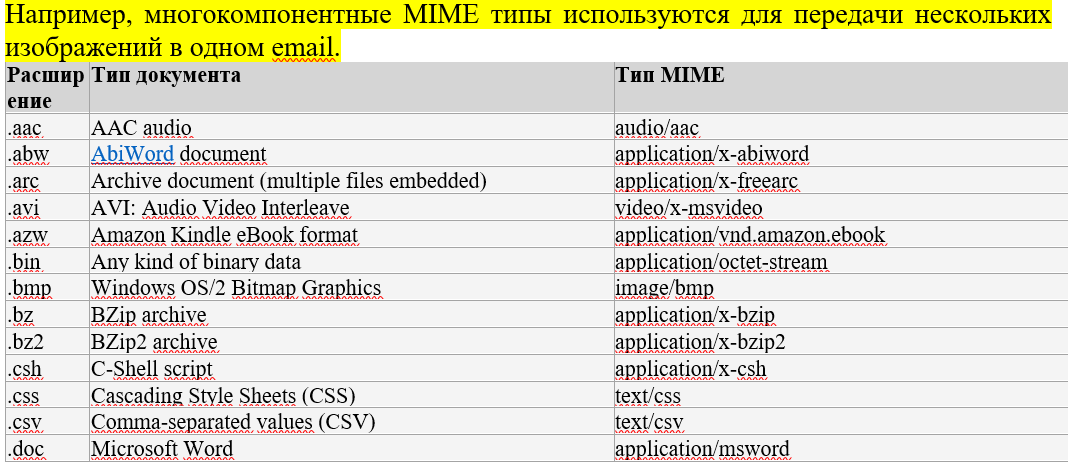
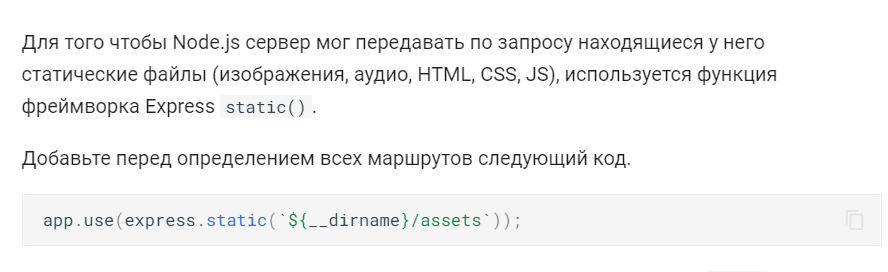
res.end(data);

});

}).listen(8080);

**MIME**

Internet Media Types, также MIME-типы — типы данных, которые могут быть переданы посредством сети Интернет с применением стандарта MIME. MIME тип указывается в за

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

**JSON:** JavaScript Object Notation, текстовый формат передачи данных, автор: Дуглас Крокфорд.

**JSON (JavaScript Object Notation) -** простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером. JSON основан на двух структурах данных:

• Коллекция пар ключ/значение

**•** Упорядоченный список значений. В большинстве языков это реализовано как массив, вектор, список или последовательность.

**JSON**

Для формирования json можно использовать встроенную функцию объекта JSON – JSON.stringify(obj). Где obj это js объект с какими-то данными.

Для обратного взаимодействия, а то есть чтения можно создать пустой объект и преобразовать json-строку в объект с использованием функции JSON.parse()

**XML**

**MIME** для xml: application/xml, text/xml.

**XML** - это расширяемый язык разметки (Extensible Markup Language), разработанный специально для размещения информации в World Wide Web, наряду с HTML

Вот несколько основных правил создания верно сформированного документа:

• Каждый элемент XML должен содержать начальный и конечный тэг.

• Любой вложенный элемент должен быть полностью определён внутри элемента, в состав которого он входит.

• Документ должен иметь только один элемент верхнего уровня.

• Имена элементов чувствительны к регистру.

Для того, что бы распарсить XML документ или строку можно использовать модуль xml2js, который скачивается с помощью npm.

Пример:

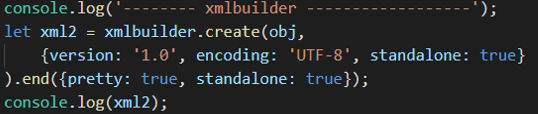
var xml2js = require('xml2js');

var parser = new xml2js.Parser();

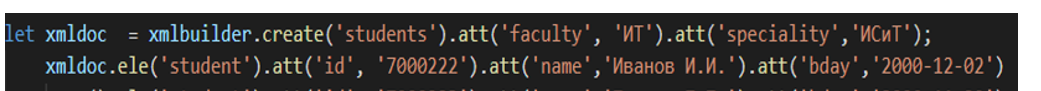
parser.parseString(<some xml srting>);

Для того, что бы сериализовать объект в XML можно воспользоваться библиотекой xmlbuilder, которая позволяет как автоматически, так и вручную сериалиизовать объект в XML

Пример:

Автоматически: 

Вручную:



Если сериализовать автоматически, то каждый элемент будет помещен в тэг <$>, что затруднит обработку документа к примеру в других приложениях.

**Multiparty**

Модуль multiparty позволяет обработать входящий multipart/form-data запрос.

Пример: (строки кода относятся к скрину)

var multiparty = require('multiparty');

var form = new multiparty.Form();



1. 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.

**Порядок разработки**

- Импорт модуля http: const http = require(‘http’);

- создание функции, которая будет срабатывать при обращении к серверу (request listener)

- создание объекта сервера: cosnt server = http.CreateServer(<request listener>);

- задать параметры прослушивания сервера: server.listen(<port>, <address>, <callback>);

**GET**

Метод get указывается в объекте options(парметров), который потом передается методу request объекта http.

Для добавления параметров к get запросу можно сериализовать параметры в строку с использованием модуля querystring и объединить с путем запроса:



И передать путь в параметрах запроса.

**POST**

Метод POST тоже указывается в объекте параметров. Что бы передать параметры в POST запросе их нужно передать в теле запроса. Сериализуются параметры аналогично GET:

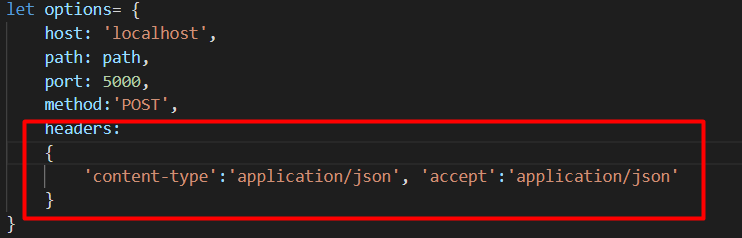


Но записываются в тело запроса с помощью функции write() объекта ответа:



**Заголовки**

Заголовки добавляются в объекте параметров в качестве вложенного объекта:



**Обработка ответа**

Ответ обрабатывается callback функцией:

(Смотреть в примере создания)

**Порядок разработки**

Пример:

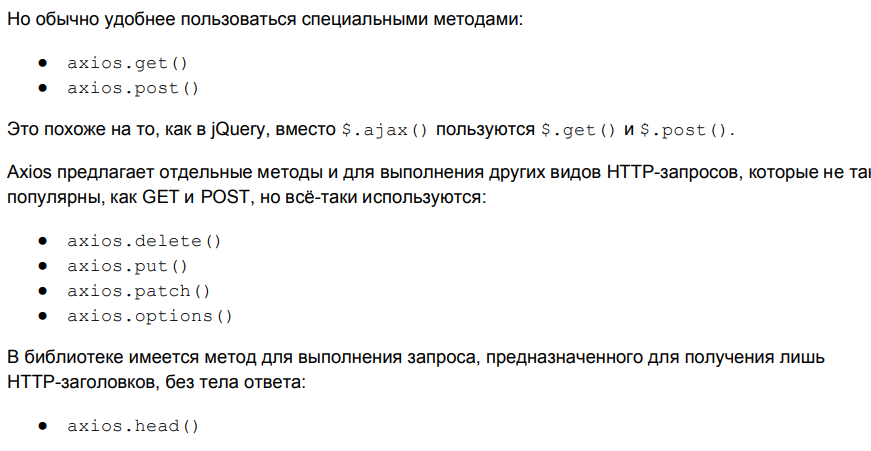


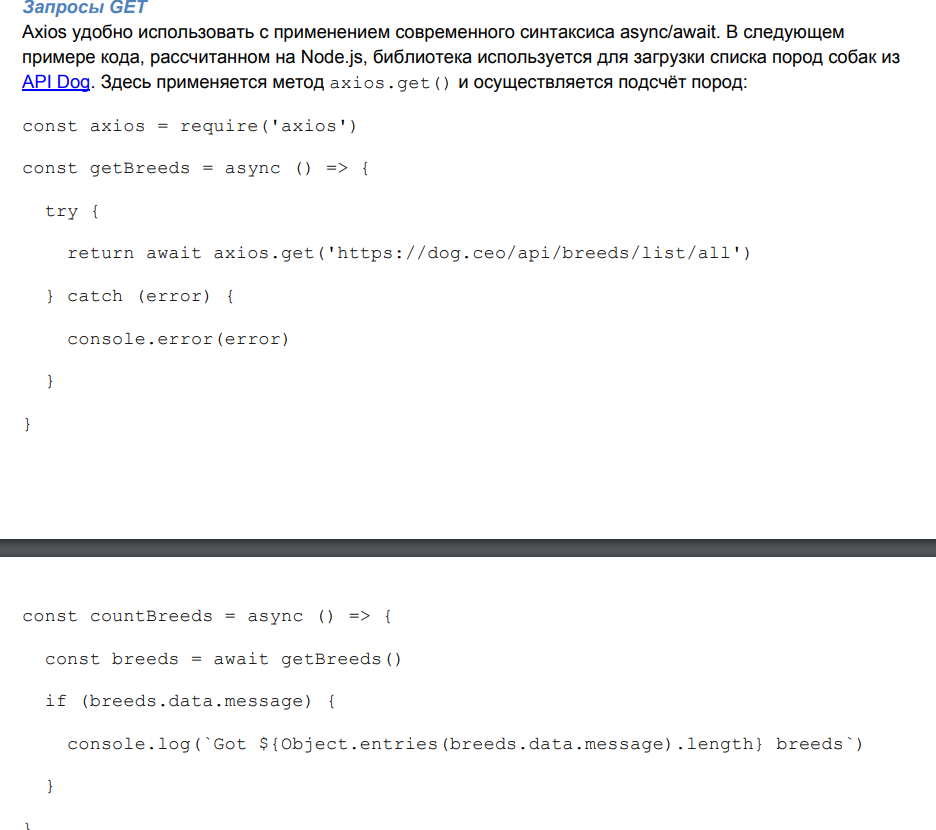
1. HTTP-клиент. Порядок разработки HTTP-клиента в Node.js (axios). Отправка GET- и POST-запросов с параметрами, параллельных запросов, добавление заголовков и обработка ответа.

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

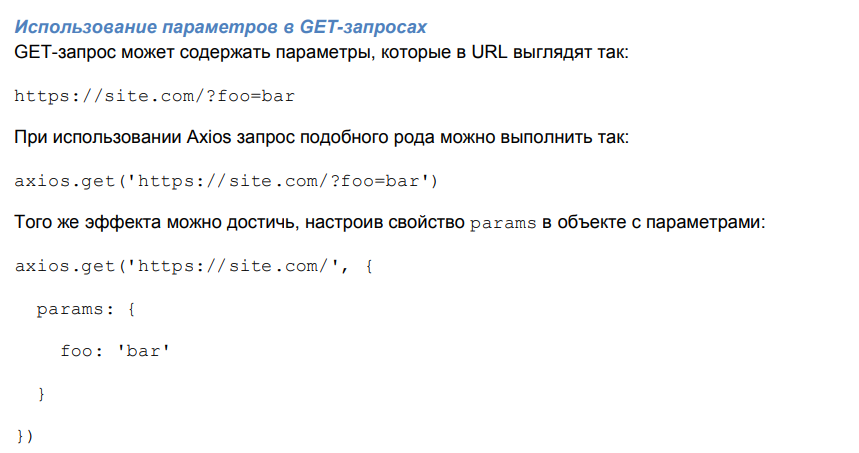
Для установки Axios можно воспользоваться npm:

npm install axios

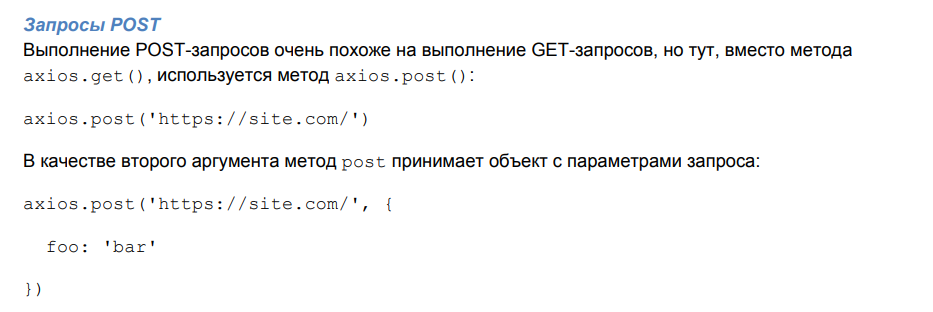




Запросы **GET** с параметром



Запросы **POST** с параметром



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

WebSocket - протокол полнодуплексной связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером через постоянное соединение, что даёт в распоряжение приложения очень быстрое соединение, характеризующееся низкими задержками и небольшой дополнительной нагрузкой на систему.

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

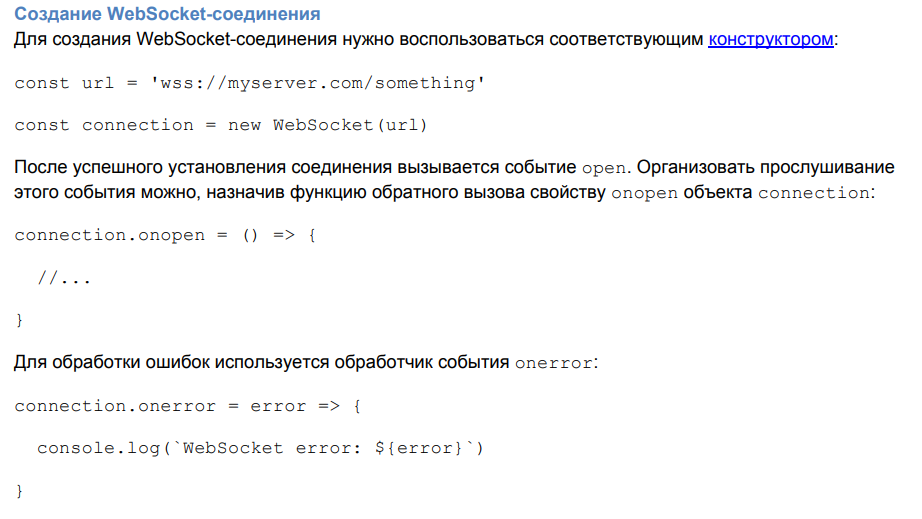
Методы:

- close – закрывает WS-подключение

- send – передача данных через WS-соединение

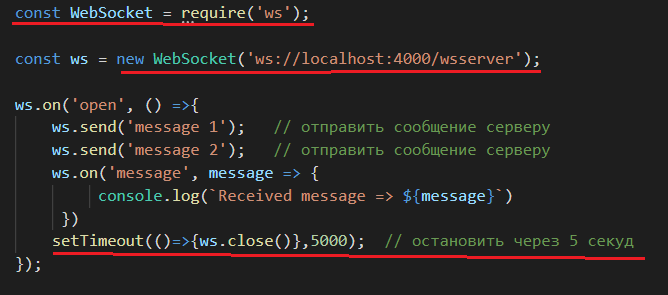
1. Порядок разработки Websocket-сервера и клиента: использование потоков, ping/pong-сообщения, обработка json-сообщений.

**Порядок разработки**



Сервер: 

Клиент:

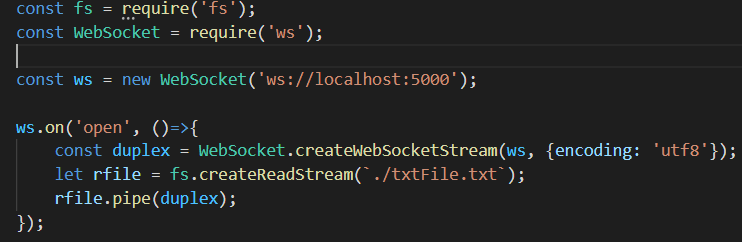
****

**Использование потоков**

Чтение из потока:



Отправка в поток:



Ping/pong

Данный механизм предназначен для проверки соединения. Тот, кто хочет проверить соединение должен отправить фрейм ping с помощью функции ping() объекта ws с произвольным телом. Тот, кто получил сообщение ping должен в разумное время ответить фреймом pong с тем же телом.

**JSON**

****

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

**RPC (скрины не читать)**

**Удалённый вызов процедур(rpc)** — класс технологий, позволяющих программам вызывать функции или процедуры в другом адресном пространстве (на удалённых узлах, либо в независимой сторонней системе на том же узле)

**rpc-websockets**

Модуль позволяет:

- регистрировать rpc процедуры на сервере

- разделять процедуры по правам доступа (public/protected)

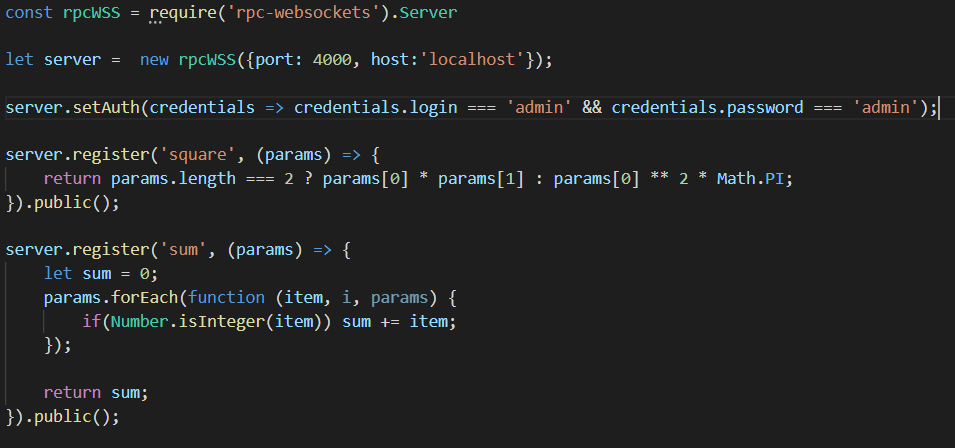
- создавать события

- подписываться на события

- авторизация для вызова защищенных процедур

- отправка оповещений

Сервер:



Клиент:



**Работа с процедурами**

Для начала на сервере должна быть зарегистрирована одна и более процедур. Она может принимать определенные параметры, которые будут нужны для ее работы. Так же должен быть установлен тип доступа к процедуре. Если тип доступа public, то ее может вызывать любой пользователь. Иначе если тип доступа protected, то она доступна для доступа только авторизованным пользователям. Для авторизации должны быть определены credentials с помощью метода setAuth. Если пользователь хочет использовать защищенную процедуру, то он должен авторизоваться с помощь метода login() в который передает объект с credentials для входа.

**Генерация событий**

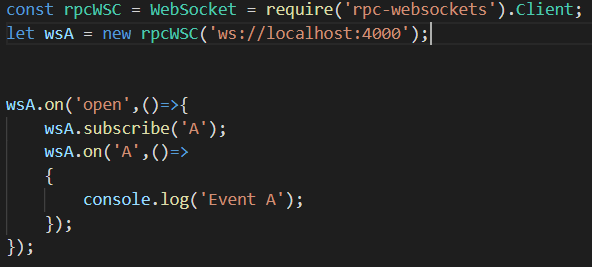
Для создания события используется метод event(name), где name – имя события.

Для инициации события используется метод emit(name), где name – имя события.

Для обработки события на клиенте нужно сначала подписать на событие с помощью метода subscribe(name).

Далее мы прослушиваем это событие и при его появлении производим какие-то действия

Пример:



**Обработка уведомлений**

Уведомления – аналогичны вызовам удаленных процедур с тем отличием, что уведомление это запрос без идентификатора. Это указывает на то, что пользователь не заинтересован в ответе – следственно объект ответа от сервера не отправляется. В таком случае если пользователь вызовет процедуру, которой не существует, то он не получит ответа об ошибке.

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

**Файловая система**

За работу с файловой системой отвечает стандартный модуль fs

**Создание файла:** fs.open(path, ‘w’, callback)

Где path – путь к файлу.

W – режим открытия. W отвечает за открытие файла для записи. Файл создается если его не существует.

**Копирование:** fs.copyFile(path1, path2, callback)

**Чтение:** fs.readFile(path, callback)

**Запись:** fs.writeFile(path, string, callback)

**Синхронные и асинхронные операции**

Асинхронные операции не блокируют event loop. Возможно использование 2-ух версий асинхронного апи: callback, promise

Синхронные блокируют event loop и последующее исполнение JS до того, пока не завершится операция.

Синхронные:

- readFileSync

- readdirSync – чтение директорий

- writeFileSync

- appendFileSync

- mkdirSync – создание директории

- unlinkSync

- rmdirSync

- existSync – проверка наличия файла в директории

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

**Файловая система**

За работу с файловой системой отвечает стандартный модуль fs

**Создание файла:** fs.open(path, ‘w’, callback)

Где path – путь к файлу.

W – режим открытия. W отвечает за открытие файла для записи. Файл создается если его не существует.

**Удаление:** fs.unlink(path, callback)

**Переименование:** fs.rename(oldPath, newPath, callback)

**Запись в конец:** fs.appendFile(path, string, callback)

**Слежение за файлом/директорией:** fs.watch(path, callback)

**Синхронные и асинхронные операции**

Асинхронные операции не блокируют event loop. Возможно использование 2-ух версий асинхронного апи: callback, promise

Синхронные блокируют event loop и последующее исполнение JS до того, пока не завершится операция.

Синхронные:

- readFileSync

- readdirSync – чтение директорий

- writeFileSync

- appendFileSync

- mkdirSync – создание директории

- unlinkSync

- rmdirSync

- existSync – проверка наличия файла в директории

1. 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.

**Файловая система**

За работу с файловой системой отвечает стандартный модуль fs

**Работа с директориями**

Создать: fs.mkdir()

переименовать: fs.rename()

Чтение содержимого: fs.readdir()

Удаление: fs.rmdir()

Проверка наличия файла: existSync()

**Синхронные и асинхронные операции**

Асинхронные операции не блокируют event loop. Возможно использование 2-ух версий асинхронного апи: callback, promise

Синхронные блокируют event loop и последующее исполнение JS до того, пока не завершится операция.

Синхронные:

- readFileSync

- readdirSync – чтение директорий

- writeFileSync

- appendFileSync

- mkdirSync – создание директории

- unlinkSync

- rmdirSync

- existSync – проверка наличия файла в директории

**Buffer**

Buffer - объект, предназначенный для работы с потоком байт; применяется в функциях readFile, writeFile; 1GB (32-bit), 2GB (64-bit)

Buffer – класс наследуемый от Unit8Array и расширяет его функционал.

Основные методы Buffer:

- alloc(num) – создает буфер заполненный нулями длинной num

- allocUnsafe(num) – создает буфер длиной num. Быстрее alloc, но в буфере могу находится старые не использованные данные.

- from([..]) – создает буфер с массивом байт, переданными в массиве-параметре.

- values – получить значения

- toJson – преобразовать в json объект

- toString – преобразовать в строку

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

**Потоки**

Поток – абстракция над данными.

Виды: Writable, Readable, Duplex, Transform;

Примеры:

Поток для чтения (Readable) — это поток, из которого можно читать данные. Записывать данные в такой поток нельзя. Когда в такой поток поступают данные, они буферизуются до того момента пока потребитель данных не приступит к их чтению.

● Поток для записи (Writable) — это поток, в который можно отправлять данные. Читать из него данные нельзя.

● Дуплексный поток (Duplex) — в такой поток можно и отправлять данные и читать их из него. По существу это — комбинация потока для чтения и потока для записи.

● Трансформирующий поток (Transform) — такие потоки похожи на дуплексные потоки, разница заключается в том, что то, что поступает на вход этих потоков, преобразует то, что из них можно прочитать.

**Readable** – Http req(server), Http resp(client), sockets, fs read, process.stdin

**Writable** – Http req( client ), http res(server), sockets, fs write, stdout

**Duplex** – sockets, zlib(компрессия данных), crypto streams

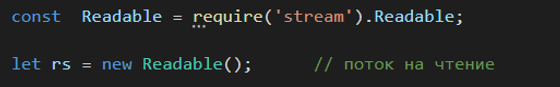
**Transform –** zlib, crypto;

**Режимы**

- Автоматический - В потоковом режиме данные автоматически считываются из системы и передаются приложению как можно быстрее с помощью событий через интерфейс EventEmitter.

- Пошаговый - В режиме приостановки метод stream.read() должен вызываться явно для чтения фрагментов данных из потока.

**Создание и использование:**



- push() – помещает в буфер строку, Unit8Array, Buffer

- read() – считывает данные из буфера и возвращает их пока поток находится в readable состоянии. По умолчанию считывает 16К. Можно увеличить параметром передаваемым в метод.

- \_read() – считывает данные асинхронно. Данная функция не должна применяться в вызывающем коде и предназначена только для внутреннего использования методами Readable класса.

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

**Потоки**

Поток – абстракция над данными.

**Виды**: Writable, Readable, Duplex, Transform;

Примеры:

Поток для чтения (Readable) — это поток, из которого можно читать данные. Записывать данные в такой поток нельзя. Когда в такой поток поступают данные, они буферизуются до того момента пока потребитель данных не приступит к их чтению.

● Поток для записи (Writable) — это поток, в который можно отправлять данные. Читать из него данные нельзя.

● Дуплексный поток (Duplex) — в такой поток можно и отправлять данные и читать их из него. По существу это — комбинация потока для чтения и потока для записи.

● Трансформирующий поток (Transform) — такие потоки похожи на дуплексные потоки, разница заключается в том, что то, что поступает на вход этих потоков, преобразует то, что из них можно прочитать.

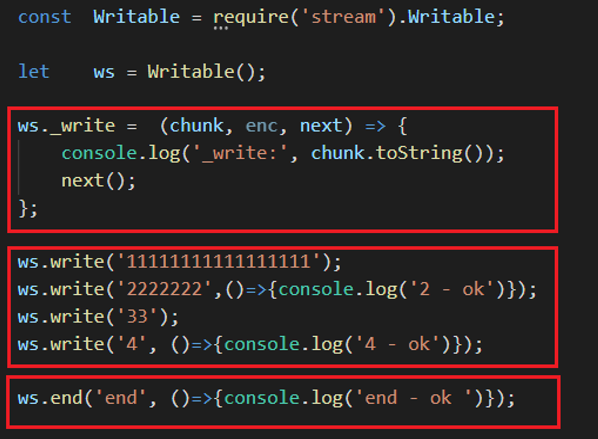
**Readable** – Http req(server), Http resp(client), sockets, fs read, process.stdin

**Writable** – Http req( client ), http res(server), sockets, fs write, stdout

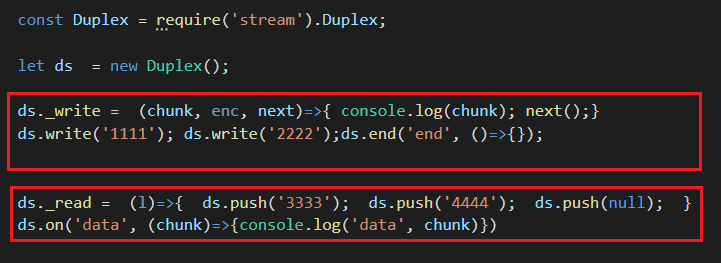
**Duplex** – sockets, zlib(компрессия данных), crypto streams

**Transform** – zlib, crypto;

**Writable**



**Duplex**

****

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

**TCP** один из основных протоколов передачи данных интернета. Предназначен для управления передачей данных интернета. Пакеты в TCP называются сегментами.

В стеке протоколов TCP/IP выполняет функции транспортного уровня модели OSI.

**Свойства**

- **Надежность доставки** - Получатель подтверждает получение каждого пакета данных.

- **Управление потоком данных** - Получатель регулирует поток поступающих данных. Это достигается отправкой окна вместе с каждым подтверждением. Окно определяет кол-во данных, которое получатель готов принять.

- **Разделение каналов** - Для того чтобы множество приложений могли использовать возможности TCP, используется механизм сокетов.

- **Работа с соединениями** - До того, как начать обмен данными, стороны устанавливают соединения, при этом в памяти каждого хоста создаётся структура – блок управления передачей, в котором хранятся сокеты сторон, участвующих в соединении, адреса буферов, размеры окон, последовательные номера, различные флаги и некоторая служебная инф-ия.

- **Двунаправленный обмен д-ми** - Приложение передает данные в виде непрерывного потока актетов. Модуль TCP самостоятельно осущ-т сегментацию и буферизацию передаваемых д-х. В случае необход-ти избегать буферизации, возможность использ-я ф-ции проталкивания.

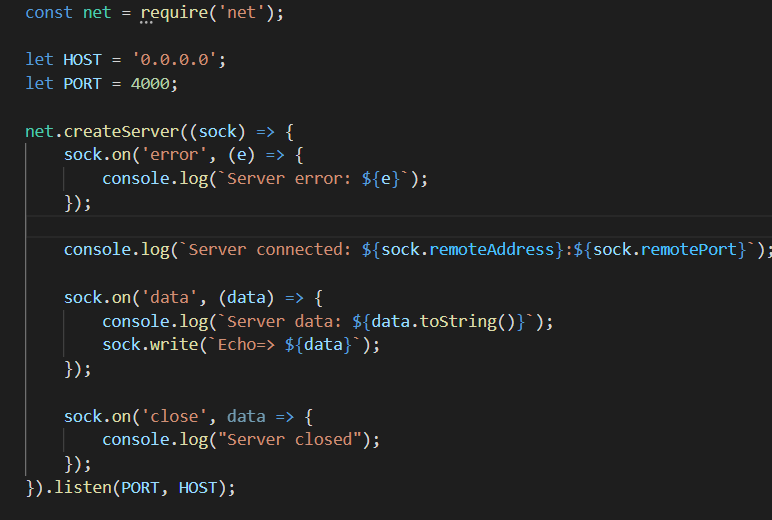
**Установка и закрытие соединения**

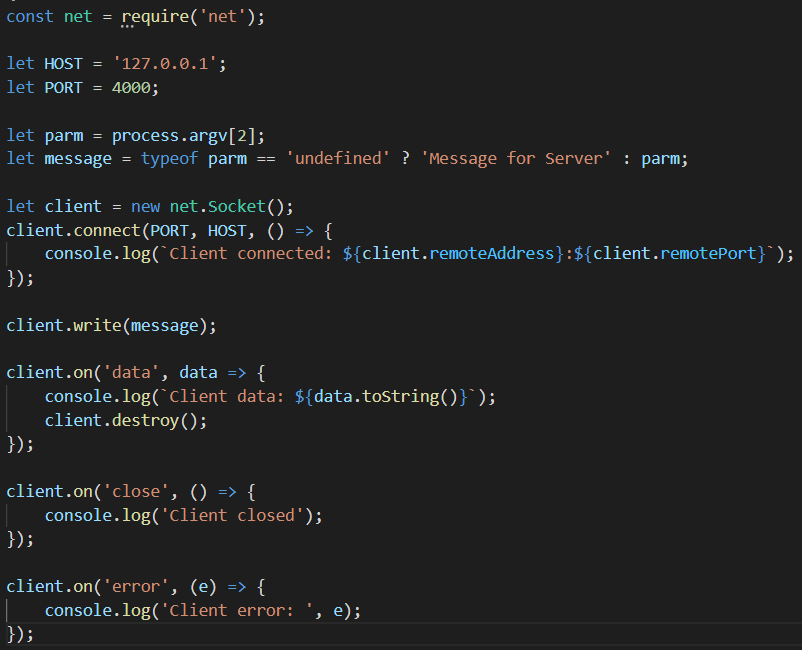
Для работы с TCP используется модуль **net** который позволяет создать сервер и клиентские сокеты.

Для создания сервера нужны хост(ip/hostname) и порт. Эти данные будут использоваться клиентом для подключения.

Для подключения клиент использует функцию **connect**(port, host). Для отправки используется **write**(string). Для закрытия соединения используется функция **destroy**().

Пример создания сервера и создания:





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

**UDP -** один из ключевых элементов набора сетевых протоколов для Интернета. С UDP компьютерные приложения могут посылать сообщения (в данном случае называемые датаграммами) другим хостам по IP-сети без необходимости предварительного сообщения для установки специальных каналов передачи или путей данных.

Уровень (по модели OSI) - Транспортный

**Свойства**

- Отсутствие механизмов обеспечения надежности – пакеты не упорядочиваются

- Отсутствие гарантий доставки

- Отсутствие обработки соединений

- UDP может вычислять по требованию контрольную сумму, но ее проверка при получении пакетов ложится на прикладной уровень

- Отсутствует буфферизания

- UDP не имеет механизмов разбиения на пакеты

- Скорость

**Поясните отличие протокола UDP от TCP.**

отсутствие механизмов обеспечения надежности: пакеты не упорядочиваются, и их прием не подтверждается;

отсутствие гарантий доставки: пакеты отправляются без гарантии доставки, поэтому процесс Прикладного уровня (программа пользователя) должен сам отслеживать и обеспечивать (если это необходимо повторную передачу);

отсутствие обработки соединений: каждый отправляемый или получаемый пакет является независимой единицей работы; UDP не имеет методов установления, управления и завершения соединения между отправителем и получателем данных;

UDP может по требованию вычислять контрольную сумму для пакета данных, но проверка соответствия контрольной суммы ложится на процесс Прикладного уровня;

отсутствие буферизации: UDP оперирует только одним пакетом, и вся работа по буферизации ложится на процесс Прикладного уровня;

UDP не содержит средств, позволяющих разбивать сообщение на несколько пакетов (фрагментировать) – вся эта работа возложена на процесс Прикладного уровня.

Ненадежные: UDP

Надежные: TCP

**Разработка сервера и клиента**





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

**Mssql**

Mssql – это клиент Microsoft SQL Server для работы с бд посредством sql запросов. Для обеспечения работы требуется драйвер.

Для выбора есть 2 драйвера:

- Tedious – реализация протокола TDS на js для работы с SQL Server

- msnodesqlv8 – библиотека, которая использует ODBC драйвер для взаимодействия с SQL Server

**Подключение**

Подключение к базе данных осуществляется с помощью метода **connect**(config, callback) где конфиг – это объект в котором прописываются все параметры подключения такие как: ip/hostname, пользователь, пароль, имя базы данных и тд.

В другом случае описание всех параметров можно заменить 1, который называется connectionString в котором будет описано все то же самое, только в привычном формате.

**Типы данных**

****

**Обычные и подготовленные запросы**

Обычные:

Данные запросы осуществляются с использованием класса Request и метода query(request) где request является строкой, которая характеризует запрос.

Подготовленные:

Данные запросы осуществляются с использованием класса PreparedStatement и методов input(), prepare(), execute();

Метод **input**(parName, type, actualPar) используется для ввода параметра. Здесь parName – имя параметра, совпадающее в именем параметра в строке sql запроса; type – тип которым представлен параметр; actualParam – js переменная, которая хранит значение подставляемого параметра.

Метод **prepare**(sqlString) подготавливает запрос к исполнению с использованием параметров, которые были введены с использованием input(). Здесь sqlString – строка запроса, где параметры помечены с использованием @.

Метод **execute**() вызывает исполнение подготовленного запроса.

Подготовленные запросы более быстрее обычных, тк хранятся в откомпилированном виде и защищены от SQL-инъекций.

**Обработка результата**

Обработка результата осуществляется с помощью callback функции, которая имеет 2 параметра – error, data

Error – ошибка

Data – результат в виде объекта. Поле recordset отвечает за результат выполнения. Если это селект запрос – там выбранные данные, иначе пусто. Поле affectedRows – сколько полей таблицы было затронуто запросом (insert, delete, update)

**Неявная транзакция**

Любая операция??

В след. Вопросе про явные

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

**Статические и динамические**

Статические – просто строка с подготовленным запросом

Динамические – preparedStatement. Компилируется 1 раз и потом меняются значения.

**Вызов процедуры**

Для вызова процедуры можно использовать объект класса Request и методы input(), output(), execute().

Метод **input**() – Предыдущий вопрос

Метод **output**(outParName, type) – выходное значение процедуры

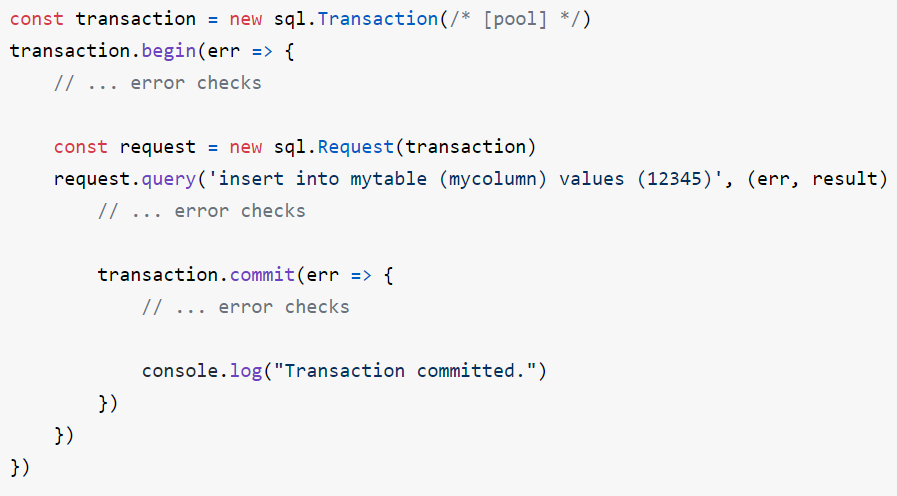
Метод **execute**(procName, callback) – вызов процедуры и обработка результата через callback

**Явная транзакция**

Для явной транзакции используется класс Transaction.



Пример:



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

**Пул соединений** - набор заранее открытых соединений с базой данных используемый для предоставления соединения в тот момент, когда оно требуется. В пакете mssql мин 0 макс 10 соединений в пуле

**Потоковый режим**

Потоковый режим включается путем передачи значение true полю stream объекта класса Request.

Потоковая обработка помогает в том случае если происходит работа с большим количеством данных.

При обработке в потоковом режиме производится прослушивание событий для получения данных.



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

**NoSQL и SQL**

SQL – Базы данных SQL используют язык структурированных запросов для определения данных и управления ими.

NoSQL - Базы данных NoSQL, наоборот, имеют динамические схемы для неструктурированных данных.

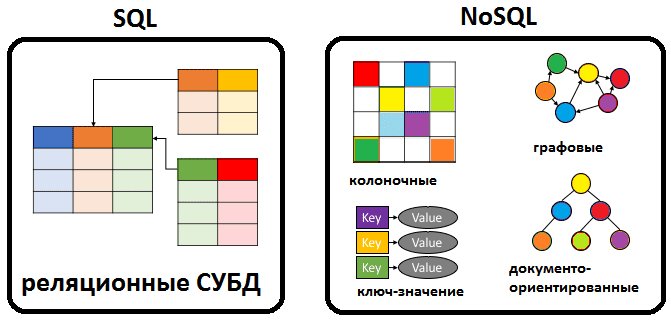
Данные хранятся разными способами: они могут быть ориентированы на столбцы, ориентированы на документы, основаны на графах или организованы как хранилище KeyValue.

Преимущества NoSQL

- Можно создавать документы без предварительного определения структуры

- Можно добавлять поля в процессе обработки

**Виды NoSQL DB**



**Примеры**

MongoDb, Redis, CouchDB, HBase

**Репликации**

Репликация - механизм синхронизации содержимого нескольких копий базы данных. Состоит эта техника в том, что данные с одного сервера базы данных постоянно копируются (реплицируются) на один или несколько других (называемые репликами).

Существует 2 подхода к реализации репликации Master-Slave; Master-Master

**Подходы**

**Master-Master** - позволяет копировать данные с одного сервера на другой. Эта конфигурация добавляет избыточность и повышает эффективность при обращении к данным. Другими словами – у нас поддерживается 2 базы данных в актуальном состоянии. Главная и та, что придет на замену в случае поломки.

Вероятные поломки делают Master-Master репликацию непривлекательной. Выход из строя одного из серверов практически всегда приводит к потере каких-то данных. Последующее восстановление также сильно затрудняется необходимостью ручного анализа данных, которые успели либо не успели скопироваться.

**Master-Slave** - выделяется один основной сервер базы данных, который называется Мастером. На нем происходят все изменения в данных (любые запросы INSERT/UPDATE/DELETE). Слейв сервер постоянно копирует все изменения с Мастера. С приложения на Слейв сервер отправляются запросы чтения данных (запросы SELECT). Таким образом Мастер сервер отвечает за изменения данных, а Слейв за чтение. Если слейвов несколько – для чтения выбирается случайным образом.

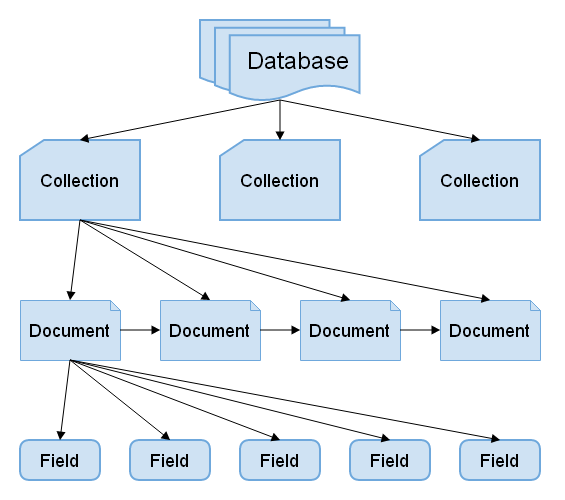
При выходе из строя Слейва, достаточно просто переключить все приложение на работу с Мастером. После этого восстановить репликацию на Слейве и снова его запустить.

Если выходит из строя Мастер, нужно переключить все операции (и чтения и записи) на Слейв. Таким образом он станет новым Мастером. После восстановления старого Мастера, настроить на нем реплику, и он станет новым Слейвом.

1. 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).

**Структура**

MongoDb – документ ориентированная база данных

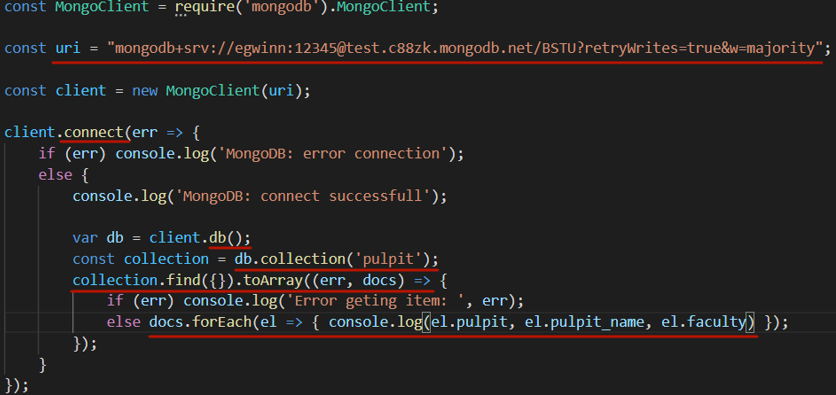


**Модуль mongodb**

mongodb – официальный драйвер MongoDB для node.js. Он предоставляет асинхронный API для получения доступа и проведения операций с базой данных и получать результат в виде промисов или callback`ов

Подключение

Для подключения к бд требуется строка подключения в которой будет описан сервер, порт, пользователь, пароль, имя бд и дополнительные параметры если требуются.

Пример: 

Выборка

.find() – извлечь все документы

.find({‘a’:3}) – извлечь документы где поле a = 3

.findOne() – извлечь 1 документ

Добавление

.insertOne() – один документ

.insertMany() – много документов

.insert – много или 1

Изменение

.updateOne() – удалить 1

.updateMany() – удалить несколько

.replaceOne()

Удаление

.deleteOne() – удалить 1

.deleteMany() – удалить несколько

**Репликация**

Типы репликации вопросом выше

Виды организации репликации в MongoDb:

- 2 сервера – primary, secondary – работает механизм heartbeat когда сервер слушает другой и копирует данные

- Арбитер – приложение, которое запускается и определяет какую ноду(сервер) принять primary в данный момент времени

**Транзакции**

Неявная

Создается сессия с помощью метода client.startSession(), объект с параметрами(опционально если есть репликация бд)

Далее используется метод session.withTransaction(callback, transactionOptions)

Далее в callback производятся асинхронные операции с бд

Сессия закрывается с помощью session.endSession()

Закрываем клиента client.close()

Явная

Создается объект с параметрами

Создается сессия session = client.startSession()

Старт транзакции session.startTransaction(transOpt)

. . .

Асинхронные операции с бд

. . .

Закрытие транзакции session.abortTransaction()

Закрытие сессии session.endSession()

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

**Mongoose** – пакет для взаимодействия с MongoDB на уровне ORM. Для этого для каждого типа объекта необходимо описать схему

**ODM**

ODM – Object Data Mapping – сопоставление объекта в котором описаны поля и их типы с документом в базе данных. Аналог ORM которые работают с SQL бд.

**Схема**

Схема определяет метаданные модели – свойства, типы данных.

Ограничения схемы:

- required - требуется обязательное наличие значения

- min max – минимальное и максимальное значение для числовых данных

- minlength maxlength – минимальная и минимальная длина для строкового типа данных

- enum – строка должна представлять одно из значений в указанном массиве строк

- match – строка должна соответсвовать указанному регулярному выражению

**CRUD**

Добавление

.save()

.create()

Получение

.find()

.findById()

.findOne()

Удаление

.deleteOne()

.deleteMany()

Изменение   
.updateOne()

.updateMany()

.findOneAndUpdate()

**Транзакции**

Не явные

- <Model>.startSession();

- transOptions

- session.withTransaction(callback)

- внутри колбэка асинхронные операции в которые вторым параметром передается сессия

- session.endSession()

Явные

- <Model>.startSession()

- session.startTransaction();

- дальше не в колбэке(тк его нет) используются асинхронные операции в которые вторым параметром передается сессия

- session.commitTransaction или session.abortTransaction()

- session.endSession()

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

**REST** — архитектурный стиль взаимодействия компонентов распределённого приложения в сети.

**REST:** достоинства

* производительность (кэширование);
* надежность (отсутствие состояния – сервер не хранит относящиеся к сессии данные);
* простота (унифицированность интерфейса, использование повсеместных стандартов,…);
* изменяемость;
* масштабируемость.

**REST:** 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.

**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.

**Форматы передачи** – в основном JSON и XML, но могут быть и другие

**HATEOAS**

Как правило, когда мы выполняем запрос REST, мы получаем только данные, а не какие-либо действия с ними. Вот где HATEOAS восполняет пробел. Запрос HATEOAS позволяет вам не только отправлять данные, но и указывать связанные действия



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

Все выше кроме

**Общепринятые правила**

- Общий префикс для всех ресурсов сервиса (…/api/…)

- Два типа ресурсов: коллекция и элемент коллекции (1. …/users 2. …/users/123)

- Иерархическая связь (…/api/users/123/cars/123)

- Использовать существительные во множественном числе. Если несколько слов, то использовать kebab-case

- Использовать HTTP статус коды, сопроводить сообщение дополнительным кодом (например 20003, 404001,…), сделать отдельный ресурс (HATEOAS link) для пояснения ошибок

- Версионность (…/api/v1/… , …/api/v5/…)

- Постраничное получение данных: параметры limit, offset.

- Сортировка: параметр sort (…/api/students?sort=+group,+name)

- Все фильтры вынести за знак вопроса

- Пользователь получает только то, что хочет

- Обозначать в запросе формат сообщений (желательна поддержка нескольких форматов (…/api/students.json?field=bday,surname,gender)

- Глобальный поиск

- Документация

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

**GraphQL** – это язык запросов, используемый клиентскими приложениями для работы с данными.

**Схема** – описание данных, которые клиент может запрашивать у API.(файл с расшируением .gql)

**Резолвер** - В этом файле будут размещены инструкции по выполнению операций GraphQL

**Данные для доступа к БД** (= контекст БД), содержит информацию о текущем соединении к серверу БД

**Плюсы**

- уменьшение кол-ва эндпоинтов

- получение только нужных данных

- Возможность добавлять новые поля и типы в GraphQL API, не затрагивая существующие запросы

- GraphQL строго типизирован. Во время разработки проверка типов GraphQL помогает гарантировать, что запрос синтаксически верен и действителен.

- GraphiQL — встроенная в браузер IDE для изучения GraphQL.

**Минусы**

- Серверу требуется дополнительная обработка для анализа сложных запросов и проверки параметров.

- Оптимизация производительности запросов GraphQL тоже может быть сложной задачей.

- Обработка ошибок – на клиент приходит либо код 200 либо код 4хх и не развернутое сообщение об ошибке, которое сложно интерпретировать.

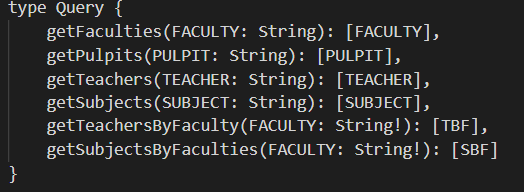
**Типы**

- Скалярные – хранит одно значение (field: data\_type)

- Int, Float, String, Boolean, Id

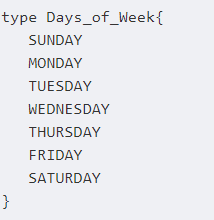
- Тип объекта – пользовательское имя для сущности. Тип объекта – объединение полей, которые являются скалярными типами

- Тип Query – Описывает доступные запросы



- Mutation – операции создания, изменения, удаления из бд

- Enum – перечисление



**Модификаторы**

- ! – non-null значение

- [] – лист объектов

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

**Архитектура**

Контекст, резолверы, схема

**Выполнение**



1. GraphQL: field-резолверы, enum, interface, union, fragment.

**Enum** - являются скалярными типами, которые позволяют полю возвращать ограниченный набор строковых значений.

**Интерфейсы** - представляют собой абстрактные типы, которые могут быть реализованы как типы объекта. Интерфейс определяет все поля, которые должны быть включены в любой объект, который его реализует

**Тип объединения** — это тип, который мы можем использовать для возврата одного из нескольких разных типов.

**Фрагменты** — это выборки, которые можно повторно использовать в нескольких операциях.

**Field resolvers** –

