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.

**Ресурсы**

Оперативная память, процессорное время.

**Блокирующие и неблокирующие операции**

Обозначение "I/O" (Ввод/Вывод) в первую очередь ссылается на взаимодействие с системным диском и сетью при поддержке libuv.

О блокировании говорят, когда выполнение js-кода приостановлено до тех пор пока не завершится работа сторонней операции.

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.

**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;

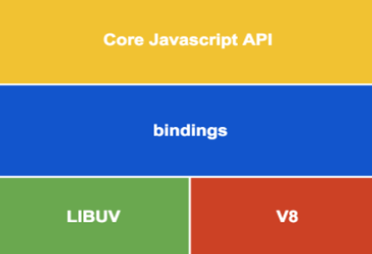
- не блокирует выполнение кода при вводе/выводе (в файловой системе до 4х одновременно);

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

**Структура**

**Core** – собраны модули для работы с сетевыми запросами, **V8** – компиляция JS-кода в машинный, **libuv** – взаимодействие с ОС(EventLoop тоже там).

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**



**Модули**

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

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

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

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.

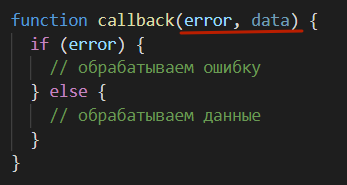
1. Асинхронное программирование. Функция обратного вызова. Проблема "Callback hell" и способы решения. Примеры.

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

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

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

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

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

**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:

state («состояние») — вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.

result («результат») — 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.

****

EventEmitter: для наследования можно использовать ключевое слово extends.

Производный от EventEmitter объект приобретает функциональность, позволяющую генерировать и прослушивать события.

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:

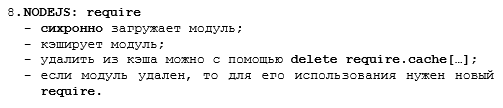
- поддержка require для импорта модуля

- 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.

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

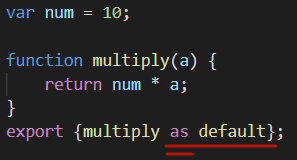
**Форматы модулей**: CommonJS, AMD, UMD, ES6

**Экспорты**

Экспорт по умолчанию до объявения:

export **default** function…

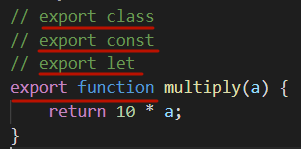
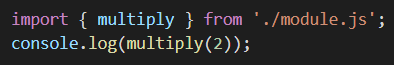
Экспорт после объявления:



Именованный export после объявления:

|  |  |
| --- | --- |
|  |  |
|  |

Экспорт именованный до объявления:

**Импорт**

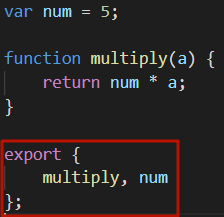
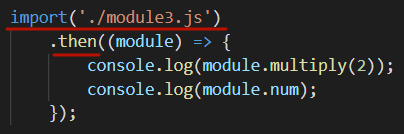
Импорт определенных функций:

import {SomeFunc} from ‘some.js’

Импорт всего, как объекта:

import \* as some from ‘some.js’

Динамический импорт:

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. Любая директория, в которой есть **package.json**, интерпретируется как Node.js-пакет

**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 lt(v1, v2) == v1 < v2

**major.minor.patch:**

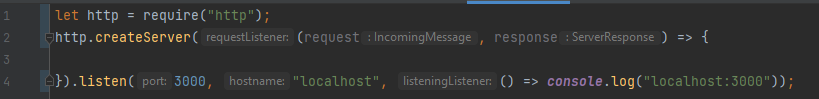
* **major** – новые возможности без сохранения обратной совместимости**,**
* **minor** –новые возможности с сохранением совместимости;
* **patch** - исправление ошибок, рефакторинг.

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.

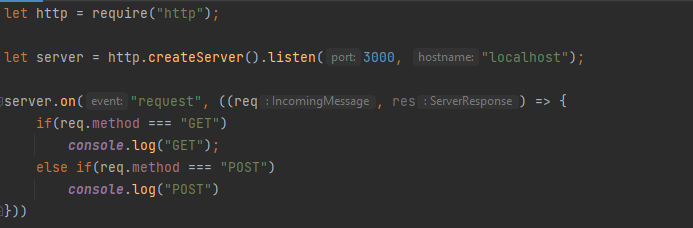
***Порядок разработки простейшего HTTP-сервера в Node.js.***

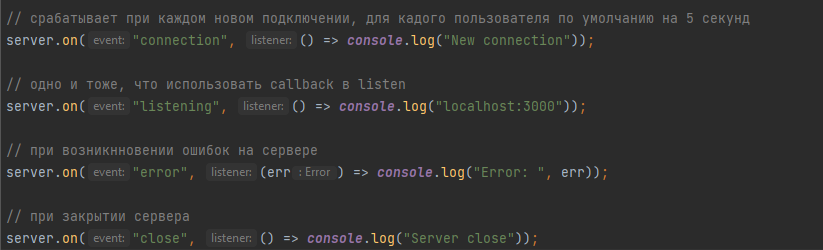
**HTTP-сервер** – приложение, которое работает на HTTP протоколе, принимает запросы от пользователя и формирует ответы.

В node js для создания сервера, нужно импортировать пакет **“http”**, далее в этом объекте, вызываем метод **createServer()** возвращает объект **Server**, дальше можно обрабатывать запросы в теле сервера , например вызывать разные функции (например импортированные из пакетов) или



обрабатывать запросы по событию **«request»**

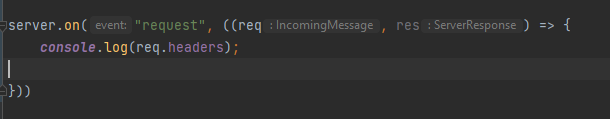


***События, генерируемые на сервере*** 

***Извлечение данных из HTTP-запроса***

***параметры (только некоторые)***

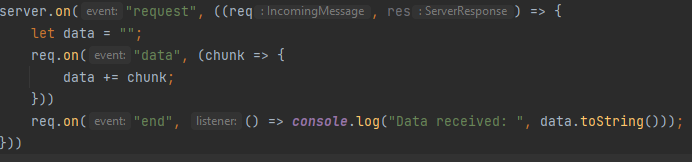
|  |  |
| --- | --- |
| req.method | Получаем метод запросы (GET, POST…) |
| req.url | Uniform resource relocation |
| req.headers | Заголовки |
| req.httpVersion | 1.0\1.1\2.0 |
| req.socket.**localAddress** | адрес сервера |
| req.socket.**remoteAddress** | адрес клиента |
| req.socket.**localPort** | номер порта сервера |
| req.socket.**remotePort** | номер порта клиента |

***Заголовки*** 

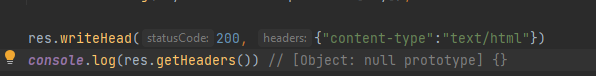


***тело***

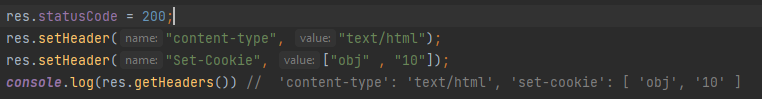
Для извлечения данных из запроса, нужно использовать два **события data и end**, так как данные сервер получает порционно.



***формирование данных HTTP-ответа***

**Заголовки** **в response**, метод writeHead записывает заголовки в сетевой канал напрямую, 

чтобы отобразить заголовки в коде и работать с ними (удалять, проверять есть такой или нет и т.д) нужно использовать **setHeader(…)**





Запись данных в ответ

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

***Порядок разработки простейшего HTTP-сервера в Node.js. ^^^ 12 вопрос***

***Объекты request и response, их свойства, методы, события.***

Объекты ***request и response*** являются объектами сервера. Запрос формируется из дынных, который отправил клиент, для разбора используется библиотека на Си (http парсер). Response формируется сервером во время выполнения , записываются данные и заголовки.

***Свойства request рассмотрены в 12 вопросе.***

**События request** кроме событий **data** и **end** есть еще события **close**,

close – указывает, что соединение было закрыто.

**События response** **finish** и **close**

finish – отправка данных завершена

close – соединение закрыто

Методы **response:**

|  |  |
| --- | --- |
| response.getHeader(name) | Получить заголовок |
| response.getHeaderNames() | Коллекция имен заголовков |
| response.getHeaders() | Коллекция заголовков и значений |
| response.hasHeader(name) | Существует или нет в запросе (bool) |
| response.removeHeader(name) | Удалить заголовок |
| response.setHeader(name, value) | Установить заголовок |
| response.writeHead(statusCode[, statusMessage][, headers]) | Установить статус код и коллекцию заголовков |
| write | Запись в тело |
| end | Конец формирования ответа |

Методы:

- destroy() – уничтожает запрос

Cобытия:

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

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

Свойства **response:**

|  |  |
| --- | --- |
| response.req | Ссылка на request |
| response.socket | Ссылка на объект сокета (можно получить адреса и порты) |
| response.statusCode | Статус код ответа |
| response.statusMessage | Статус сообщения |

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

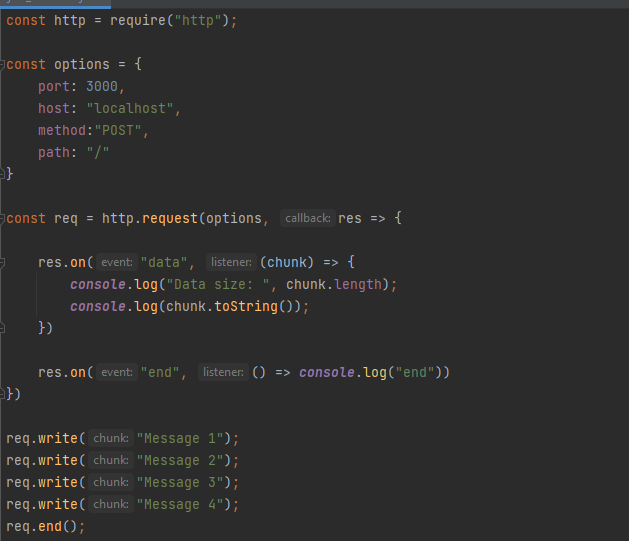
- statusMessage

***Порционная отправка и получение данных.***

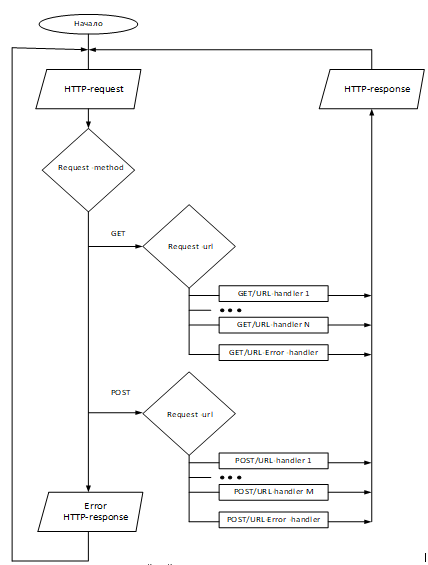
Прием данных на сервере, в методе request ожидаем события data и end. Каждый раз при записи данных на клиенте в тело, происходит событие data, событие end на сервере. Для демонстрации, может записывать эти же данные клиенту с пометкой, номера “порции”



Отправка данных на сервер и приём ответа.



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



***Типичный цикл работы http-сервера***

В начале работы сервера приходит запрос, в сервере обработываем запрос, если поддерживается сервером, если нет возвращаем ошибку 405, что метод не поддерживается. Иначе обрабатываем запрос, получаем url и обрабатываем и формируем ответ и отсправляем пользователю.

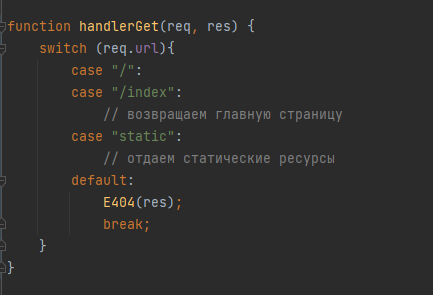
***маршрутизация***

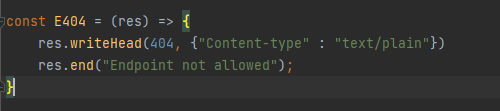
В nodejs можем определять маршруты и обработчики этих маршрутов. Для примера такого подхода можно привести пример кода:



Определяем обработчики для двух типов методов, на все отсальные отвечаем ошибкой о том, что метод не поддерживается.

Обработчики get запросов, “/” , “index”, “static”, конечные точки endpoin’ы. То есть конечная точка обработки запроса по запросу, который получен из url. Если никакой endpoint не подошел, то возвращаем ошибку с кодом 404 и сообщением





**Статические ресурсы**

Статические ресурсы – хранятся на сервере и отправляются клиенту в том виде, в котором они хранятся на сервере без изменения, это страницы html, таблицы стилей, скрипты, документы и т.д.

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

Изначально такой стандарт использовался для отправки содержимого в электронных письмах, но потом при развитии интернет стал стандартом для пересылки данных в web-приложениях.

Стандартизацией занимается организация **IANA,** полный список можно посмотреть по ссылке

<https://www.iana.org/assignments/media-types/media-types.xhtml>

Но достаточно знать основные, которые использовались в лабораторных работах

***отдача статики***

Сущесвует два способа передачи статики, так как response и request являются потоками чтения и записи, то мы можем явно и неявно записывать данные в эти потоки.

Два этих способа отличаются лишь тем, что данные уже подготовлены для передачи или нет, если у нас данные уже прочитаны, то можем просто записать их в поток записи , например заранее прочитали с помощью fs моуля и записать их в поток:

response.end(data)

Если данные нужно прочитать, то можем создать поток для чтения, затем вызвать метод pipe, и в качестве параметра передать поток для записи, это одно и тоже, что мы можем сделать, когда создаем поток для чтения и записи и поток для чтения передаем в pipe.

fs.createReadStream(“file.dat”).pipe(response);

**В потоке данные передаются порциями неважно от выбранного способа.**

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

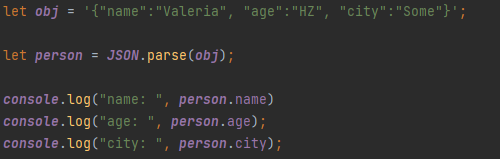
**JSON** (JavaScript Object Notation) - стандартный текстовый формат для представления структурированных данных на основе синтаксиса объекта JavaScript.

**XML** — расширяемый язык разметки. (схож с HTML только теги неопределены заранее).

Для работы в javascript, есть статический объект JSON, nodejs есть тоже, как и многие объекты для единообразия интерфейса.

Для разбора строки в объект JSON используем метод parse.

Разбор **JSON**



Результат



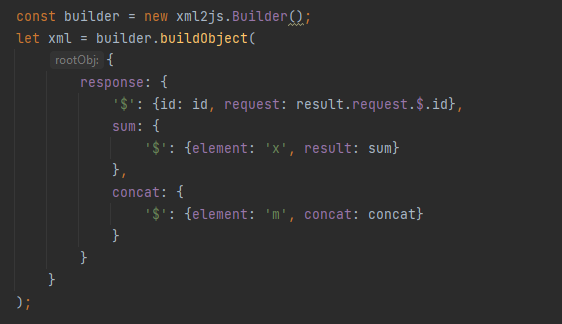
Формирование объекта (например, как ответ) используем метод **stringify(object)**

Разбор **XML**



Для разбора xml нужно использовать сторонний пакет, например xml2js, для разбора файла используем метод **parseString**(“название файла”, callback)

Для создания нового xml файла используем **Builder** из этого же пакета, в метод **buildObject** качестве параметра передаем объект JavaScript



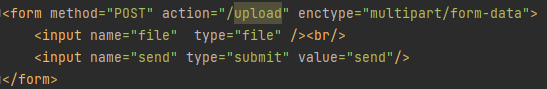
***Загрузка файла на сервер (пакет multiparty).***

Для работы с multiparty нужно установить и экспортировать одноименный пакет. В качестве параметра, нужно передать объект {uploadDir: “dir”}, папка для загрузки файлов из пакета. Имеет несколько событий, из которых возможно извлечь информацию и обработать ошибки.

******



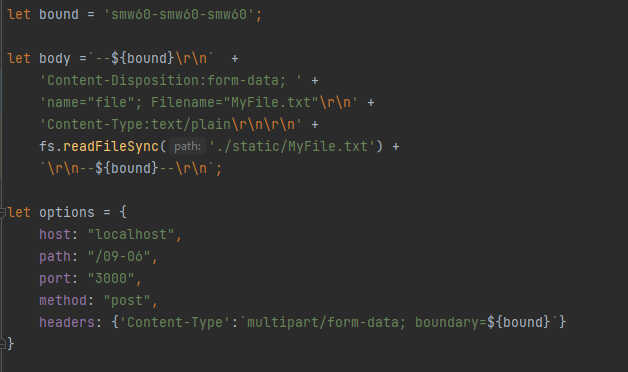
Передача элементов из формы, обязательным параметрам для такой передачи файлов является **enctype="multipart/form-data"**



Вторым возможным вариантом является передача передачи является, загрузка файлов из запроса, для этого нужно указать разделитель (переменная bound). Затем называем заголовки о типе содержимого, и с помощью пакета fs добавляем данные для отправки на сервер.

Желательно знать, что каждое новое содержимое должно начинаться с

«--разделитель», а последнее вложение заканчиваться «--разделитель--»

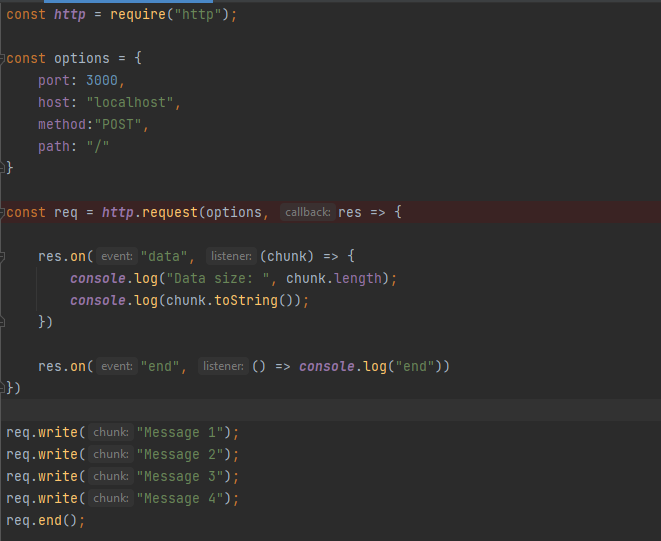


Если спросят для чего, то для пересылки нескольких файлов, такой подход может использоваться для загрузки документов из страницы, например, прикрепите копию студенческого, паспорт и pdf документ с вашим CV. На сервере можно все разобрать в одном блоке и не делать несколько страниц с загрузками.

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***

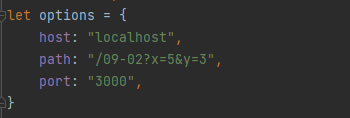
Разработка клиента использует пакет http. Для создания клиента вызывается метод **request**, при этом возвращается объект **ClientRequest.** Принимает параметры в виде объекта, поля **host, path, port, method, headers.**

****

Метод write используем при пост запросе, при использовании get, добавляем параметры к url, формируем с помощью пакета **querystring**

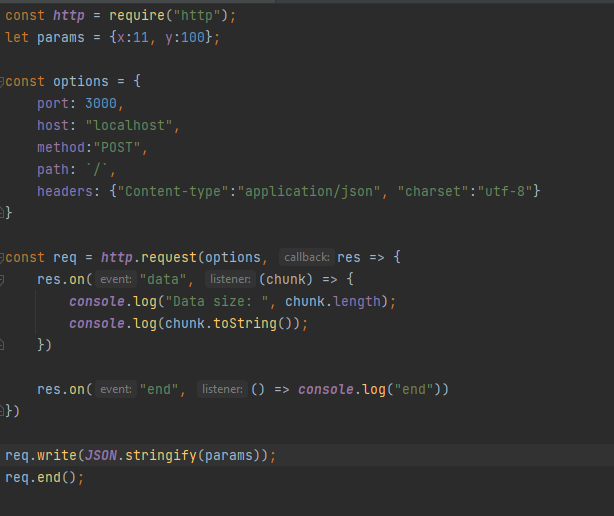


или просто как обычный текст.



При использовании POST, можем передавать объект JSON и на сервере извлекать из тела, для этого нужно добавить соответствующие заголовки.

Пример клиента



На сервере получаем данные, по окончании извлекаем данные в объект и производим какие-то действия



Результат на сервере с заголовками от клиента



Порционная отправка получение данных расмотрена в вопросе 14

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.

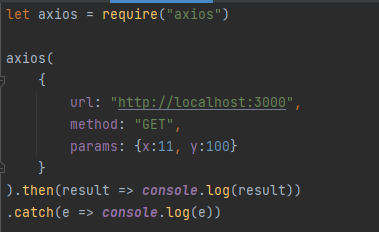
При отправке данных используя axios не нужно использовать пакет http, для этого импортируем пакет axios.

Оъект **axios является промисом**, то есть мы можем использовать для обработки данных конструкции then, catch.

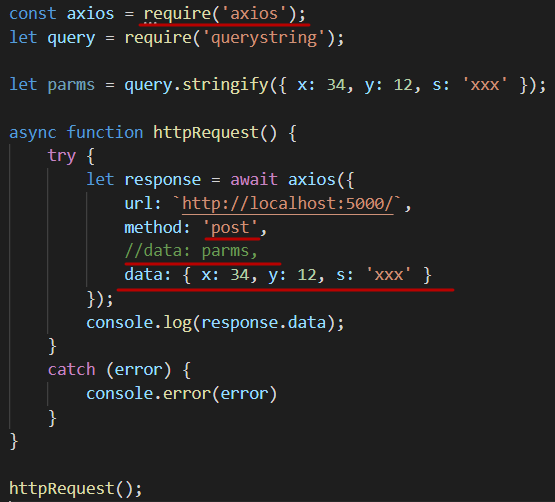


В данном примере используем функцию для удобства, например могли бы передавать параметры и это функция делала бы запросы **GET**, такие функции могли бы создать для всех нужных методов.

В данном примере к url добавляем строку URI и параметры, по умолчанию используется метод GET, но лучше указывать явно. Когда приходит ответ, обрабатываем ответ в then, в случае возникновении ошибок, обрабатывается блок catch.

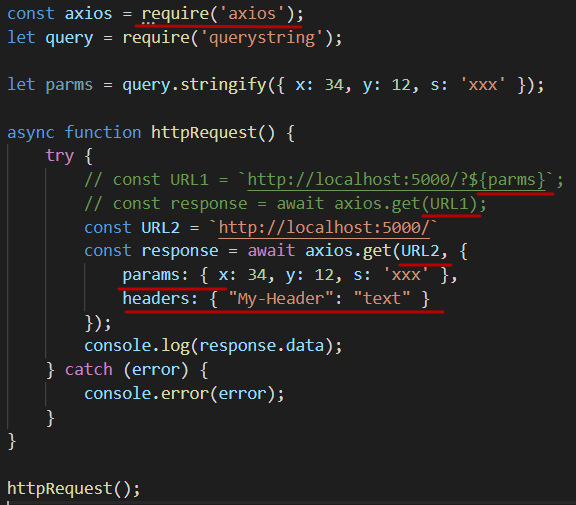


Можем добавлять параметры не к **URI**, а передавать в свойстве param, конечно это будет равносильно строке [**http://localhost:3000/?x=11&y=100**](http://localhost:3000/?x=11&y=100)



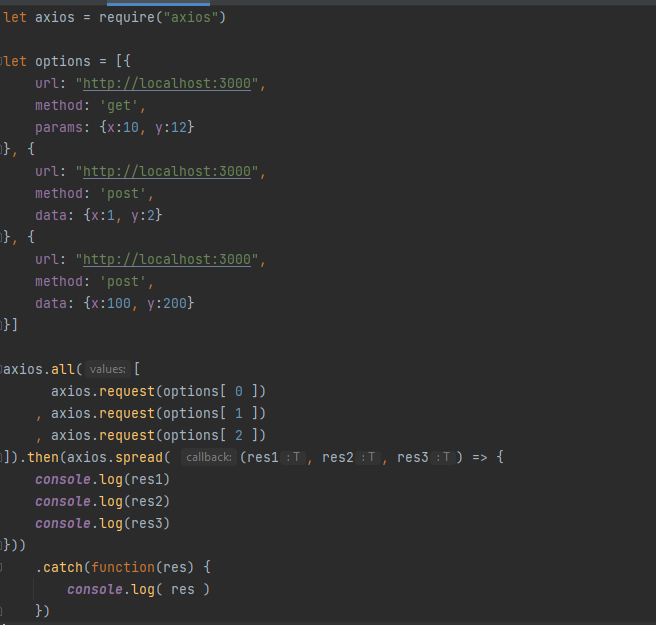
При формировании POST запроса и передаче **JSON** объектов, мы передаем данные в тело запроса тело запроса используя поле **data**, преобразование в строку будет выполнено автоматически.

Заголовки записываются в поле **headers:**



**Паралельные запросы в axios**

Для отвправки нескольких запросов мы можем использовать **метод all,** параметром является массив запросов **axios.request**(параметры), при выполнении запроса, данные будут возвращены все или обработана ошибка.



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.

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:

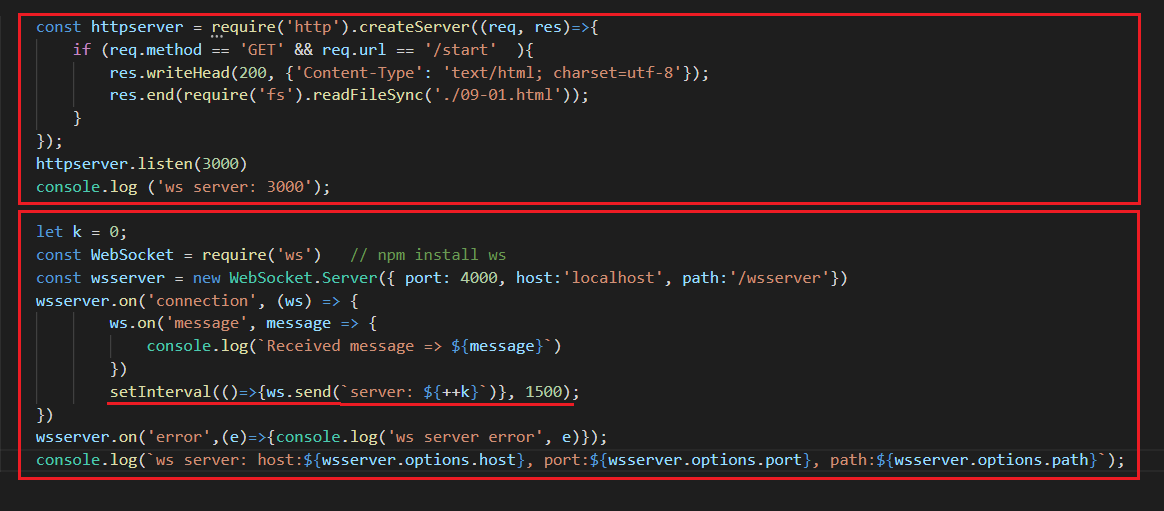
- close – закрывает WS-подключение

- send – передача данных через WS-соединение

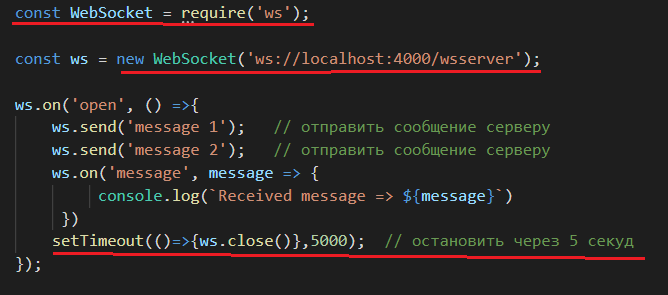
1. Порядок разработки Websocket-сервера и клиента: использование потоков, ping/pong-сообщения, обработка json-сообщений.

**Порядок разработки**

Сервер:

****

Клиент:

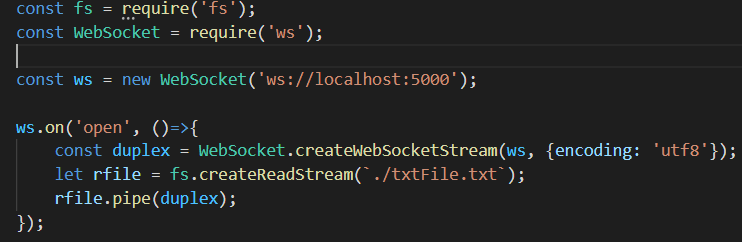
****

**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**

**Удалённый вызов процедур** — 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)

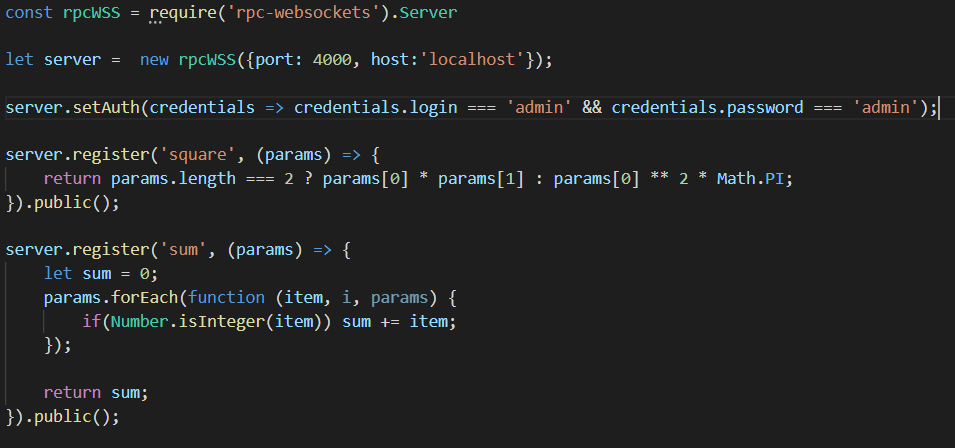
- создавать события

- 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**

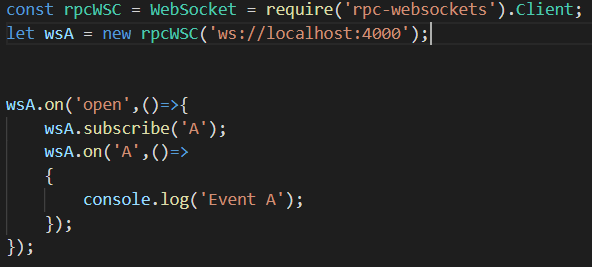
Для создания события используется метод event(name), где name – имя события.

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.

**Прочитать про notify!!!!**

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.

**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.

**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 октетов; 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. Стрим – объект, над которым происходят какие-то действия(чтение, запись, и тд.)

Виды: Writable, Readable, Duplex, Transform;

Примеры:

**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

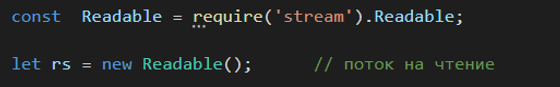
**Transform –** zlib, crypto;

**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:**



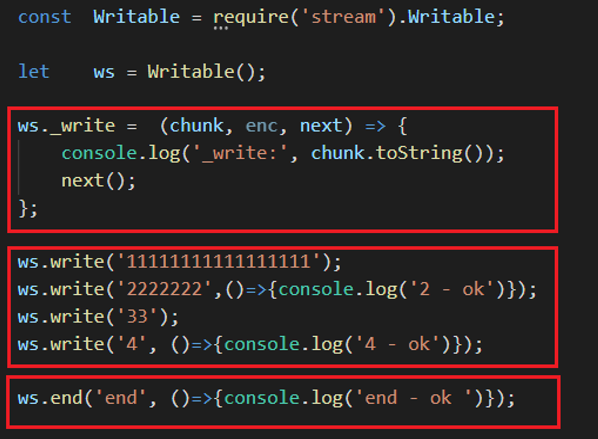
- push() – помещает в буфер строку, Unit8Array, Buffer

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

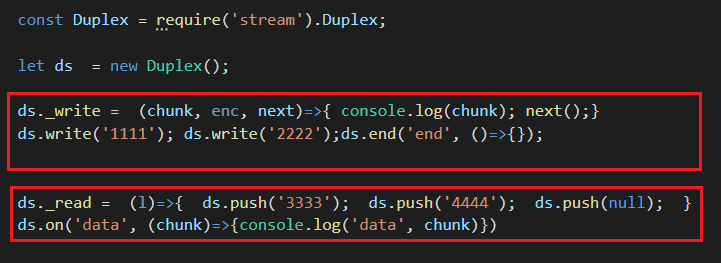
- \_read() – считывает данные асинхронно. Данная функция не должна применяться в вызывающем коде и предназначена только для внутреннего использования методами Readable класса.

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

**Writable**



**Duplex**

****

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

**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** - Прилож-е передает д-е 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.

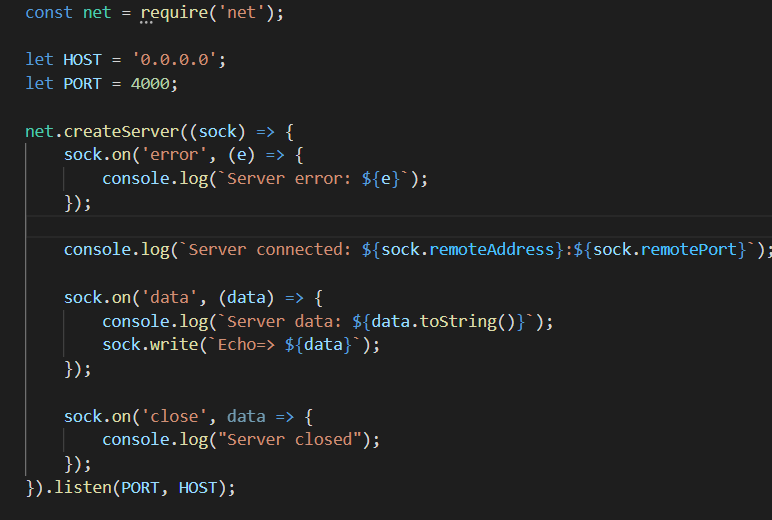
**Установка и закрытие соединения**

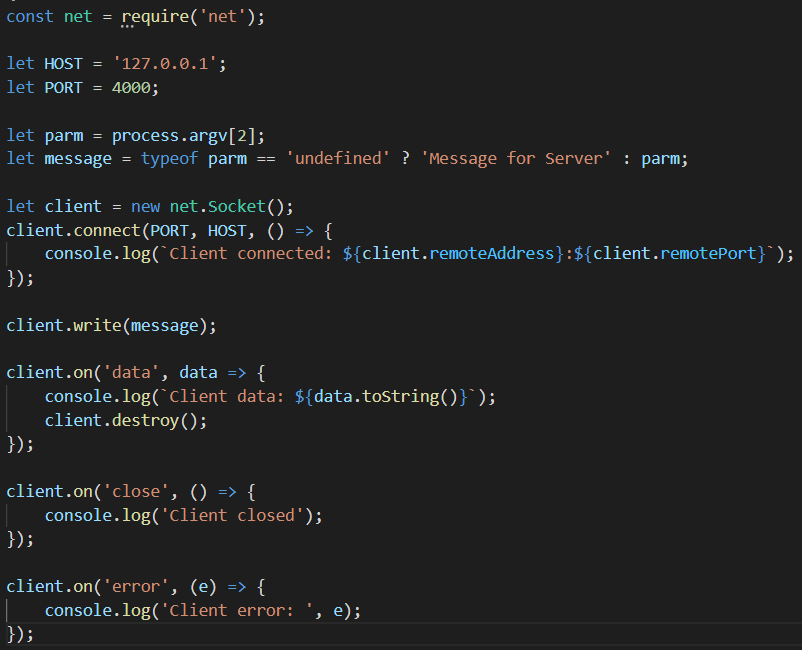
Для работы с TCP используется модуль **net** который позволяет создать сервер и клиентские сокеты.

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.

**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

**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

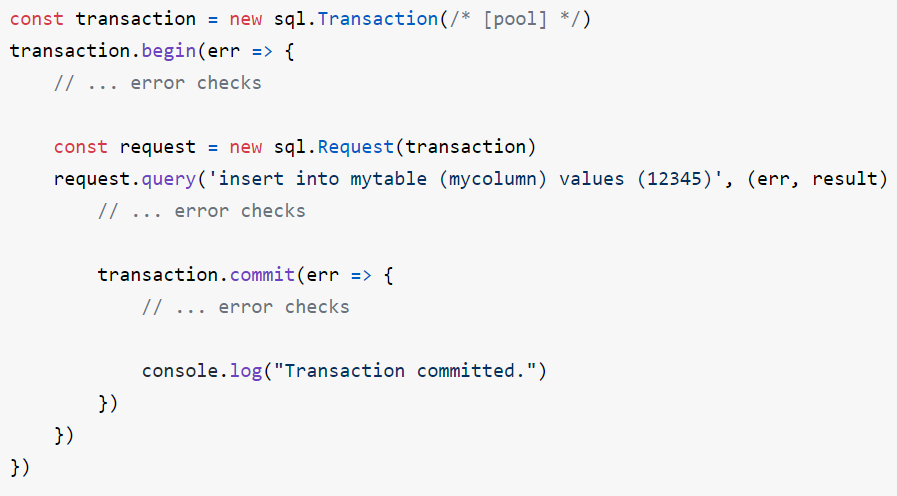
Метод **execute**(procName, callback) – вызов процедуры и обработка результата через callback

**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.

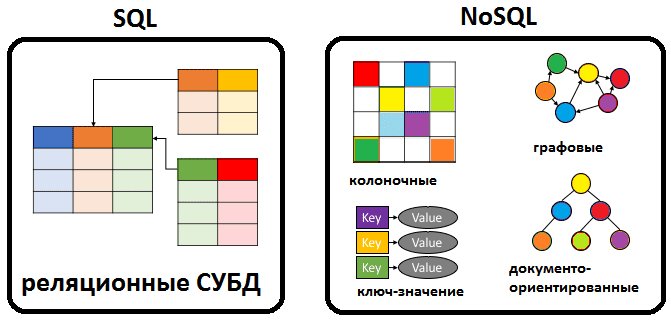
Данные хранятся разными способами: они могут быть ориентированы на столбцы, ориентированы на документы, основаны на графах или организованы как хранилище KeyValue.

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.

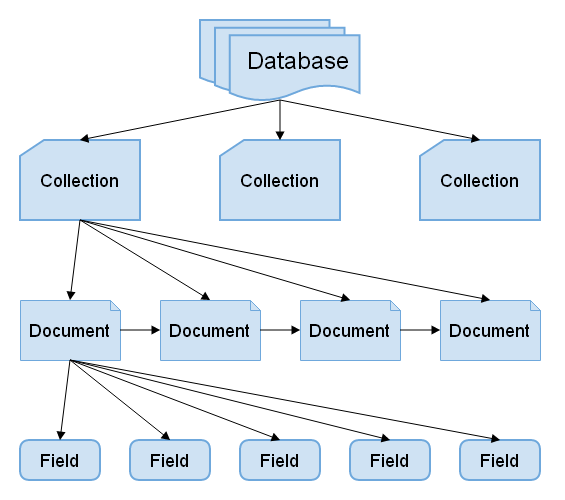
При выходе из строя Слейва, достаточно просто переключить все приложение на работу с Мастером. После этого восстановить репликацию на Слейве и снова его запустить.

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

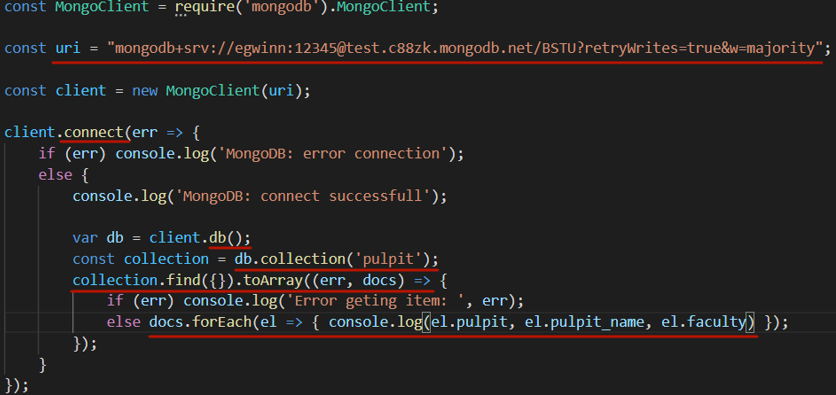


**Модуль mongodb**

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() – изменить 1

.updateMany() – изменить несколько

.replaceOne()- замена 1

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**.**

**Плюсы**

* уменьшение кол-ва эндпоинтов(конечных полех)
* 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.

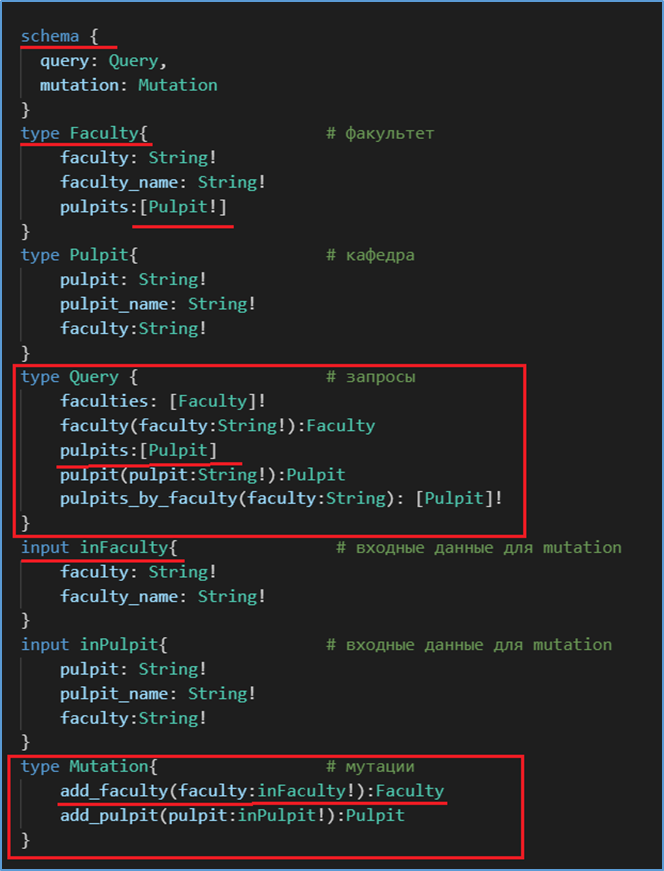
**Минусы**

* 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.

**Особенности**

* позволяет делать агрегированные запросы (можно в одном запросе запросить все необходимые данные)
* использует систему типов (инт, булин, стринг и тд)

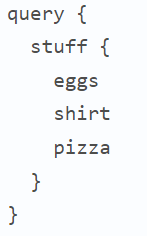
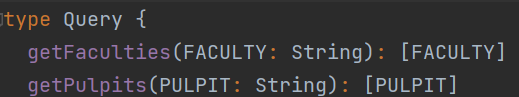
**Схема –** описание данных, которые клиент может запрашивать у API(файл с расшируением .gql). SDL (Schema Definition Language), .gql/.graphql



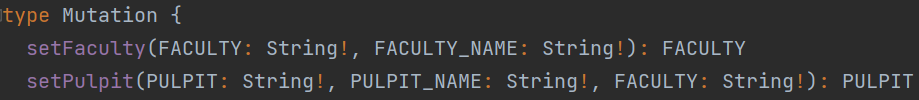
**Данные для доступа к БД (= контекст БД),** содержит информацию о текущем соединении к серверу БД

**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

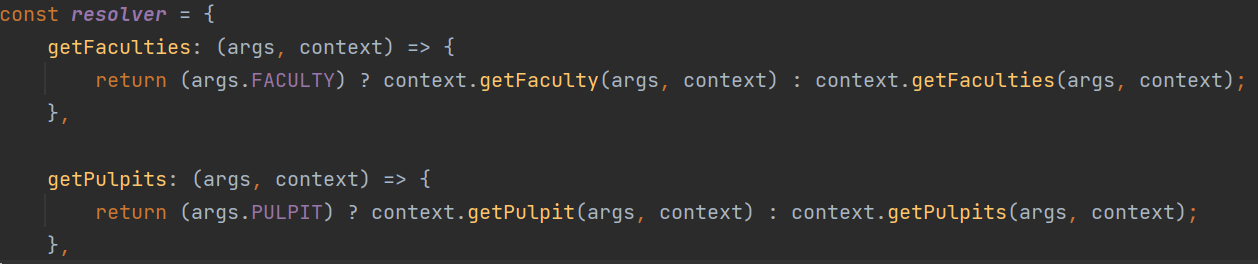


* Enum – перечисление (список)

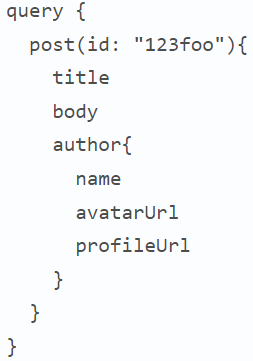
**Модификаторы**

* ! – non-null значение
* [] – лист объектов

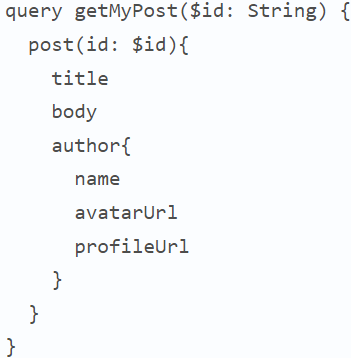
**Резолвер -** В этом файле будут размещены инструкции по выполнению операций GraphQL.



Поля запроса также могут содержать аргументы. Например, если необходимо отобразить конкретный пост, можно добавить аргумент id к полю post:



Наконец, если нужно сделать аргумент id динамическим, можно определить **переменную**, а затем использовать её в запросе (обратите внимание, также мы сделали запрос **именованным**):



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

***архитектура***

Состоит из основных двух элементов (не считая самого объекта GraphQL) :

**Схема** (описываются все типы запросов и объектов, которые мо можем получить в этом API)

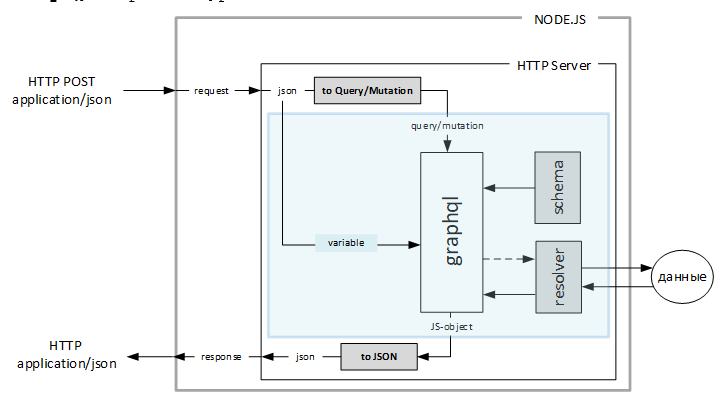
**Resolvers** разрабатываются функции, которые получают данные на основе запроса.

***выполнение graphql-запроса***

Выполнение запроса происходит следующим образом, как на схеме, приходит запрос, получаем **тело запроса и преобразуем в JSON (только POST-запрос конечно)**, получаем поле query, это соответсововать полю в resolver, и далее вызываем функцию, которая находится в этом поле или выбираем нужную на основе параметров из запроса, получаем данные и возвращаем данные.

(Все это **передаем в параметры graphQL (желательно знать, иногда спрашивает).)**





Когда получаем данные, то получаем объект со всеми полями, которые указаны в схеме, но на основе запроса, graphQL отбрасывает поля, которых не было в запросе и отправляет, только поля, которые были указаны. Тем самым уменьшаются сетевые нагрузки на сервер, так как это самый ресурсозатратные операции по пересылке данных.

Данные возвращаются в теле ответа в формате JSON.

***порядок*** **разработки graphql-сервера**

Для разработки GRAPHQL нам нужно описать схему, со всеми возможными запросами,

Описываем все возможный типы, которые могут быть в ответе, помним, что всего 4 типа поддерживает по умолчанию, но можем создавать свои. Разберем что означает:

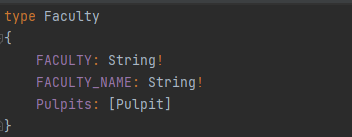
String!(Строка и не null) String (просто может быть null)

[Pulpit] в данном случае и сам это пользовательский объект и может быть null

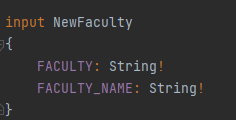
[Pulpit!] - Pulpit не может быть null

[Pulpit]! – коллекция не может быть null (но 0 объектов это не null

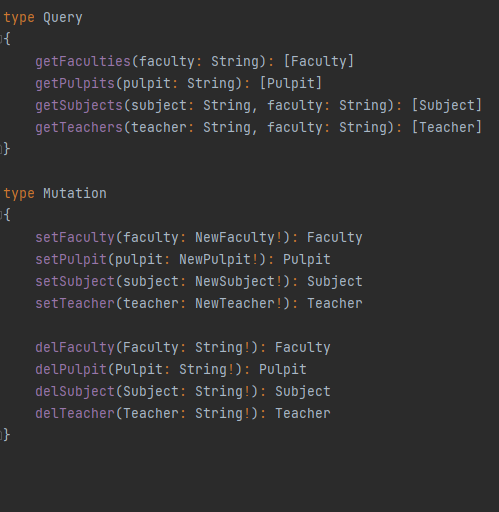
[Pulpit!]! – объекты не могут быть null и коллекция



Input (такие типы предназначены для использования при добавлении или обновлении данных (мутации) в graphQL

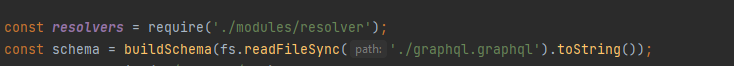


Далее определяем, что поддерживает и запросы и мутации в данном примере, и описываем все поля, которорые будут использоваться в resolvers

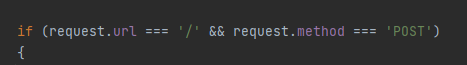


Экспортируем пакет и извлекам данные из файла и строим схему

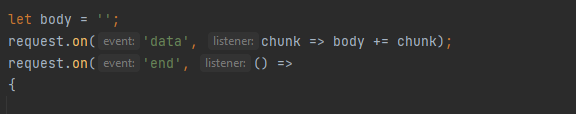


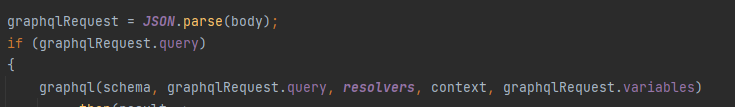


Проверяем, что запрос post (url может быть любой)

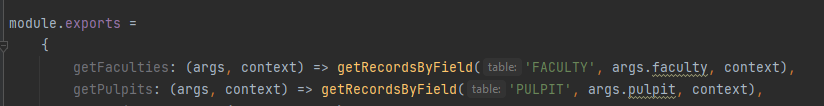


Извлекаем данные из запроса

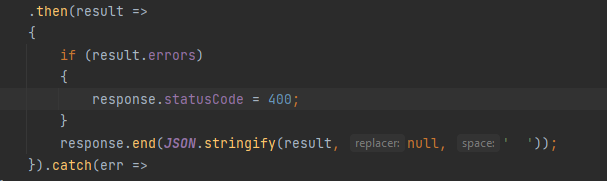


Парсим данные в объект и проверяем не пустое ли поле query, именно по нему будет извлекаться данные определенного типа и формироваться ответ.

Пишем и экспортируем resoler (пишем функции для извлечения данных и при экмпорте указываем поля, которые описывали в схеме)



Когда возвращает функция результат, то проверяем на ошибки, если успешно, то раво null, преобразуем ответ в JSON и возвращаем клиенту, если нет отправляем ошибку и сообщение.

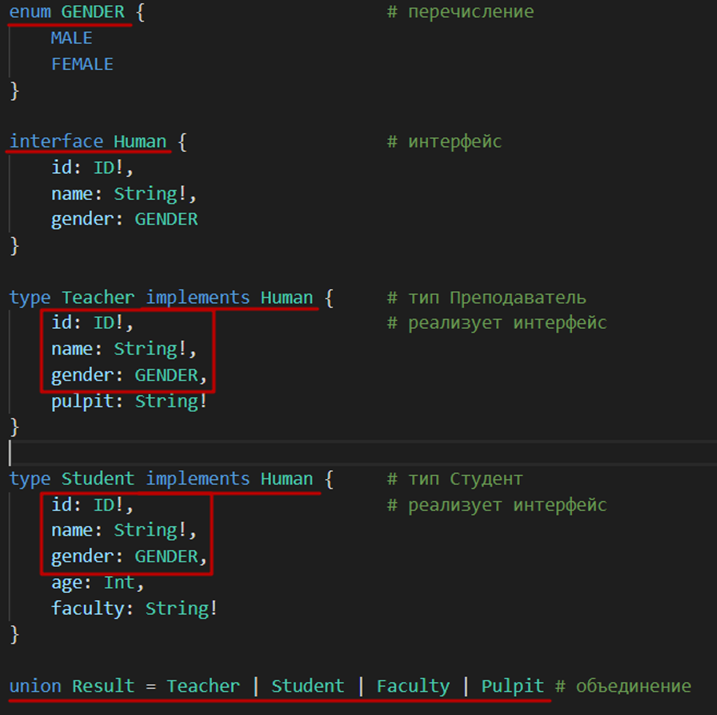


1. GraphQL: field-резолверы, enum, interface, union, fragment.

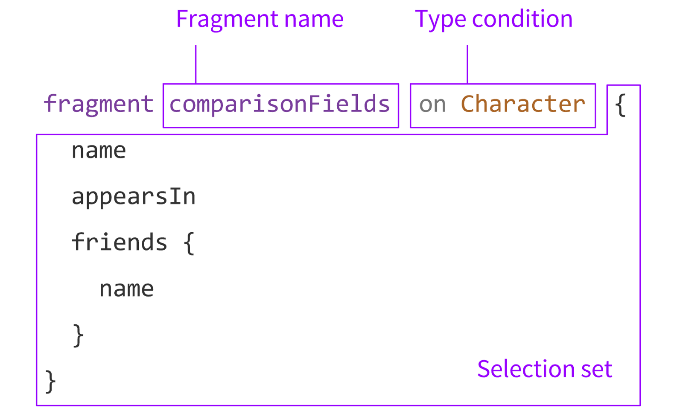
**Enum** являются скалярными типами, которые позволяют полю возвращать ограниченный набор строковых значений.

**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

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



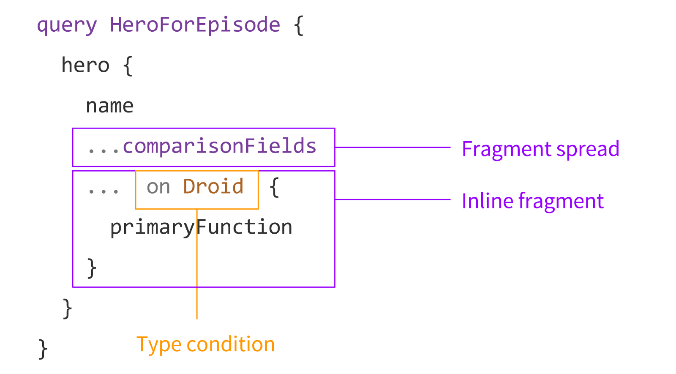
**Фрагменты** — это выборки, которые можно повторно использовать в нескольких операциях.



* **Имя фрагмента** (Fragment name). У каждого фрагмента уникальное в пределах документа имя. Оно используется для ссылки на фрагмент в операции или в другом фрагменте. Это имя, так же как имя операции, удобно использовать для логирования на сервере. Поэтому рекомендуется давать понятные имена, говорящие о назначении фрагмента. Если понадобится оптимизировать запросы, правильно подобранное имя облегчает поиск места в коде, где задан фрагмент.
* **Применимость к типу** (Type condition). В отличие от операции GraphQL, которая всегда начинается с одного из типов: query, mutation или subscription, – фрагмент может быть использован в разных выборках. Чтобы фрагмент можно было сам по себе проверить на соответствие схеме, указывается пользовательский тип данных, к которому применим фрагмент.

**Использование фрагментов в операциях**

Фрагменты не особо полезны вне операций. Возможны два способа вставки фрагментов, как показано ниже.

  
Рис. Два типа фрагментов в запросе.

* **Развертка фрагмента** (Fragment spread). Чтобы вставить фрагмент в операцию или в другой фрагмент, указывается … перед названием фрагмента. Это называется разверткой. Она может присутствовать в выборке, которая применяется к тому же типу что и фрагмент.
* **Встроенный фрагмент** (Inline fragment). Добавляет поля в выборку в зависимости от типа, к которому эта выборка применяется. При этом не создается внешнее определение фрагмента. Встроенный фрагмент подобен именованному, но задается в самом запросе. Отличие от именованного фрагмента в том, что не обязательно указывать тип данных. Чаще такие фрагменты используют вместе с директивами, как будет показано далее.

