**1-ый вопрос**

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

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

**HTTP:** основные свойства

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

**Request**:

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

**Response:**

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

****









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

**Структура:**

1. 1 клиент – 1 сервер

2. 1 клиент – несколько серверов

3. Несколько клиентов – 1 сервер

4. Клиенты - промежуточный сервер – сервер

**Особенности реализации web-приложения:**

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

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

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

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

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

1. **HTTP-аутентификация (Basic, Digest, Forms).**

**Идентификация** – заявление пользователя о себе.

**Аутентификация –** процедура, направленная на подтверждение идентификатора пользователя (логин и пароль, как правило). На основании какого-то секрета.

**Авторизация** – после аутентификации; проверка есть ли права на выполнение тех или иных действий.

Исп-ся **1 код возврата** (401 (и 200ый само собой))

401 код – Unauthorized – ваш запрос является не авторизированным.   
407 код – для прокси серверов.

**2 заголовка**, кот.мы будем использовать:

\*authorization (req)

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



**Способы аутентификации:**

BASIC



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

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

Тут Basic – имя способа авторизации

Realm – имя закрытого ресурса

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

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

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

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

Дайджест - DIGEST



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

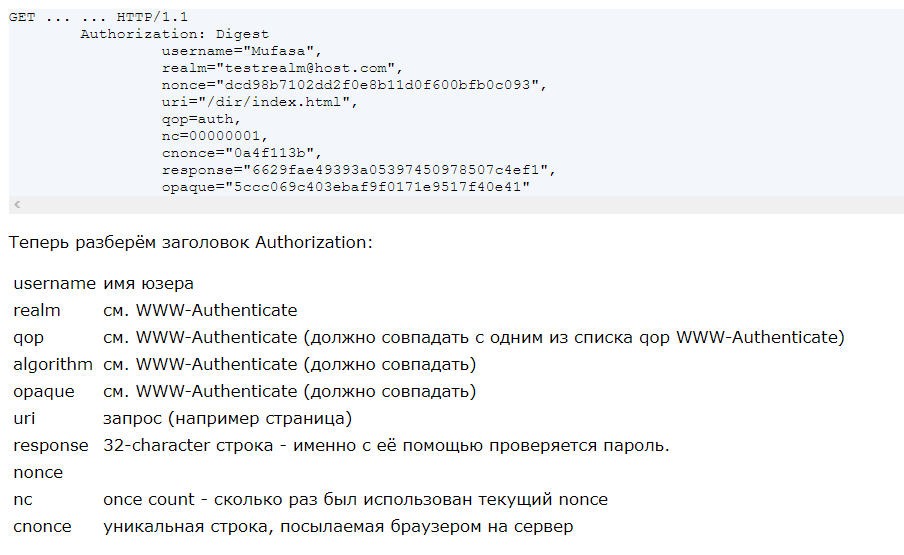
* Realm – информация о ресурсе где вводится пароль
* Qop – флаг указывающий quality of protection
* Nonce – Уникальная строка, которая генерируется на сервере
* Opaque – Строка, которую пользователь должен вернуть без изменения в ответе
* Algorithm – алгоритм исп для шифрования
* Stale – указывает был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

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

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

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

Ответ пользователя выглядит следующим образом :



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

FORMS

Нет стандарт (нигде не описан = народный способ)



Описание схемы:

Инфа о пользователе и пароле передается в куки

Процедура аутентификации:

Идет запрос

С проверет есть ли куки, если нет – запрос не аутентифицирован

Если куки есть – проверяет содержимое, есть инфа, кот.интересует и она валидна – отвечаем ресурсом; иначе – отправляем страницу, в кот.просим ввести имя и пароль.

Юзер вводит и жмет сабмит, эта инфа отправляется на С. С обрабатывает запрос и формирует на основании имени и пароля токен (бит.посл-сть, кот.С м.проверить на валидность; имеет время жизни). Отправляется на К SetCookie И указывает токен, который должен отправляться клиентом в каждом запросе.

Используется чаще всего, т.к.простой и понятный.

Все 3 способа – слабые. Если не исп-ть HTTPS – данные легко перехватить и модифицировать.

1. **Протокол HTTPS. Протокол TSL. Сертификаты. Взаимодействие центра сертификации и владельца защищенного ресурса.**

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

HTTP передаёт д-е в открытом виде. Злоумышленники могут “вклиниться” в передачу — изменить или перехватить данные.

В HTTPS для перед. данных созд.защищённый канал.

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

HTTPS = TLS + HTTP

\*TLS – прокладка между HTTP и TCP

\*есть рукопожатие

**TLS** – transport layer security – протокол защиты транспортного уровня – криптографический протокол, кот обесп защищ.передачу д-х между узлами в сети Интернет. Исп ассим.шифрование для аутентификации, симм.шифр для конфиденциалности и коды аутентичности сообщений для сохранения целостности сообщ.

Исп-ся в прилож., кот раб.с сетью Интернет, таких как веб-браузеры, работа с эл.почтой, обмен мгновенными сообщ и тюдю

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

**Процесс установки соединения:**

**TLS 1.2**

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

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

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

**TLS 1.3**

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

**Создание сертификата**

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

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

genrsa – указывает на то, что мы генерируем RSA ключ

2048 – сложность (2048 бит)

**Второй шаг** – создание запроса сертификата

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

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

new – указываем то, что мы создаем новый запрос

key – указываем откуда считывать приватный ключ

**Последний шаг** – создаем самоподписанный сертификат

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

x509 – формат сертификата

req – запрос на генерацию сертификата

days – сколько будет действителен сертификат

**Использование сертификата**

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

Для клиента:

* Добавление в ОС
* Добавление в браузер

Для добавления в ОС:

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

Для браузера:

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

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

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

Свойства:

- Дуплекс

- Каналы – потоки

- JSON, XML... форматы передачи данных

- RPC

- Стандарт RFC 6455

Процедура установки соединения Называется handshake

- Клиент отправляет запрос на апгрейд соеднения

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

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

Применяется websocket в чатах, играх, управление IoT

WebSocket API

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

Атрибуты:

- onopen – соединение установлено

- onclose – соединение закрыто

- onmessage – получено сообщение

- onerror – ошибка

- readystate – текущее состояние подключения

- url

Константы:

ReadyState

- connecting – соединение еще не открыто

- open – соединение открыто

- closing – соединение в процессе закрытия

-closed – соединение закрыто

Методы:

- close – закрывает WS-подключение

- send – передача данных через WS-соединение

1. **Разработка простейшего HTTP-сервера в Node.js. Извлечение данных из HTTP-запроса, формирование данных HTTP-ответа. Пример. Тестирование с помощью браузера AJAX (XMLHTTPRequest/Fetch).**

Для создания простейшего HTTP сервера нам нужно подключить модуль http, создать функцию обработчик, передать ее в функцию http.createServer([function]) и запустить сервер с помощью функции listen([port]).

При получении данных в теле запроса их можно обработать используя поток объекта запроса “data”. Объект запроса реализует интерфейс ReadableStream что позволяет нам прослушивать поток и получит оттуда объекты типа Buffer которые мы можем в последствие объединить и получить итоговое полученное сообщение.

Для формирования данных ответа после получения данных в событии “end” мы можем их обработать, сформировать какой-либо объект ответа. К примеру если мы хотим отправить JSON, то нам нужно установить нужные для этого заголовки (“Content-Type”)

1. **Разработка HTTP-сервера в Node.js. Обработка GET, POST, PUT и DELETE-запросов. Генерация ответа с кодом 404. Пример. Тестирование с помощью POSTMAN.**

Что бы обрабатывать разные типы запросов можно сделать свитч по методу запроса. Получить метод запроса можно из объекта запроса с помощью параметра request.method который представлен в качестве строки.

Что бы отправить ответ с кодом 404 нужно установить статус код с помощью параметра ответа response.statusCode(404) а так же можно отправить сообщение в response.end([message]), которое будет описывать возникшую ошибку.

1. **Разработка HTTP-сервера в Node.js. Обработка запросов к статическим ресурсам: html, css, js, png, msword. Пример. Тестирование с помощью браузера.**

**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);

1. **Разработка HTTP-сервера в Node.js. Обработка query-параметров GET-запроса. Пример. Тестирование с помощью браузера.**

Для обработки query параметров используется модуль url и метод parse из этого модуля. Модуль url является встроенным.

Пример:

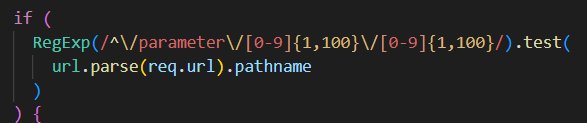
|  |
| --- |
| const http = require('http');  const url = require('url');  http  .createServer(function (req, res) {  const queryObject = url.parse(req.url, true).query;  console.log(queryObject);  res.writeHead(200, { 'Content-Type': 'text/html' });  res.end('Feel free to add query parameters to the end of the url');  })  .listen(8080); |

Результатом метода является js-объект, который хранит параметры ключ-значение соответствующие именам и значениям параметров переданных в строке запроса.

1. **Разработка HTTP-сервера в Node.js. Обработка uri-параметров GET-запроса. Пример. Тестирование с помощью браузера.**

В node js нету инструментов для получения uri параметров, но мы можем получить их с использованием регулярных выражений.

Для начала мы можем проверить с помощью объекта RegExp([regularExp]) соответствует ли строка запроса нашему регулярному выражению.



Потом мы можем разбить наш URI с использованием функции split(‘/’). То есть мы разбиваем строку URI на массив элементов. К примеру есть будет URI - /params/43/23, то у нас будет массив строк [params, 43, 23]. Далее мы можем получить интересующие нас параметры из этого массива.

1. **Разработка HTTP-сервера в Node.js. Обработка параметров POST-запроса. Пример. Тестирование с помощью браузера (<form>) и POSTMAN.**

Бля крч смотри, хуяришь форму, там поля с именами и тд, у формы метод пост и там экшон какой-нибудь. Потом на сервере хэндлишь это через поток дата и чанки.

Потом если ты дед, то используешь модуль querystring и его метод parse куда передаешь полученную строку из потока data. Результатом будет JS-object. Ты дед, тк тот модуль депрекетед.

Если ты молодой и красивый, то используешь класс URLSearchParams, модуля url. Возвращает объект с ключ-значением значение ключей ты можешь получить через [oblect].get([keyName]);

1. **Разработка HTTP-сервера в Node.js. Обработка json-сообщения в POST-запросе. Пример. Тестирование с помощью POSTMAN.**

Бля крч смотри, опять получаешь данные через поток дата, записываешь в строку. Потом что бы обработать парсишь строку через JSON.parse([dataString]).

Что бы отправить ответ – создаешь новый объект, добавляешь свойства, даешь им значения и тд. Задаешь заголовок ответа Content-Type : application/json и в response.end() записываешь JSON.stringify([responseJsObject])

1. **Разработка HTTP-сервера в Node.js. Пересылка файла в POST-запросе (upload). Пример. Тестирование с помощью браузера.**

Ну, крч, смотри, качаешь модуль multiparty. Потом делаешь форму с импутом файла с методом пост и экшоном на эндпоинт, enctype="multipart/form-data". На эндпоинте создаешь форму с использованием multipart :



Где в параметре uploadDir указываешь директорию куда сохранять файлы.

Потом обрабатываешь файл с использованием события “file” объекта формы.

Ну и отправляешь какой-нибудь ответ клиенту.

1. **Разработка HTTP-сервера в Node.js. Пересылка файла в ответе (download). Пример. Тестирование с помощью браузера.**

Using stream we can download file in node.js. We just need to set headers. Browser will automatically download the file. First header is **'Content-disposition': 'attachment;** **filename=package.json"** . If this header will present in response then browser will force fully download the file. We can also specify the file name in this header. Another header is Content-Type , this contains the type of content or file type

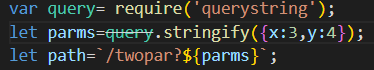
|  |
| --- |
| var http = require('http');  var fs = require('fs');  http.createServer(function (req, res) {  var files = fs.createReadStream("package.json");  res.writeHead(200, {'Content-disposition': 'attachment; filename=package.json"}'});  files.pipe(res);  }).listen(8080); |

1. **Разработка HTTP-клиента в Node.js. Оправка GET запроса с query-параметрами. Пример. Тестирование с помощью с Node.js-сервера.**

Создается объект параметров в который записывается:

* Хост
* Путь
* Метод
* Порт

Параметры преобразуются с помощью модуля querystring.stringify({[params..]});



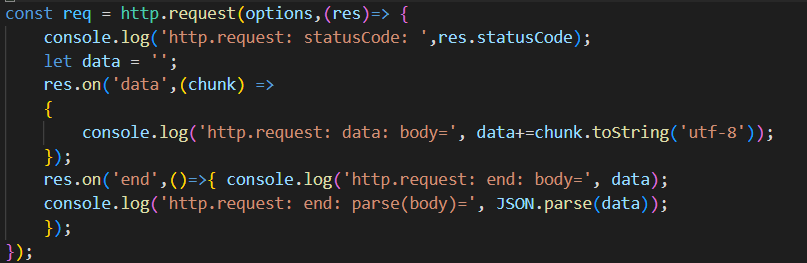
1. **Разработка HTTP-клиента в Node.js. Оправка POST-запроса с параметрами в теле. Пример. Тестирование с помощью с Node.js-сервера.**

Аналогично предыдущему вопросу, но тело пост запроса помещается в объект и используется JSON.stringify({[obj]}), потом он прикрепляется к запросу с помощью req.write(obj);

1. **Разработка HTTP-клиента в Node.js. Оправка POST-запроса с json-сообщением. Пример. Тестирование с помощью с Node.js-сервера.**

Предыдущий вопрос

1. **Разработка HTTP-клиента в Node.js. Обработка json-ответа. Пример. Тестирование с помощью с Node.js-сервера.**



1. **Разработка HTTP-клиента в Node.js. Пересылка файла на сервер в POST-запросе (upload). Пример. Тестирование с помощью с Node.js-сервера.**

Для отправки файла с клиента используется multipart/form-data

Для этого нам нужно определить границу разделения частей запроса а так же прикрепить 2 заголовка:

* Content-Desposition : form-data; name=file; FileName=[somethong];
* Content-Type

Это все находится в теле

Структура тела следующая:

* Граница
* 2 заголовка
* Поток байт считанный с помощью readFile
* Граница

Так же к запросу присоединяется заголовок:

{'Content-Type':'multipart/form-data; boundary='+bound}

1. **Разработка HTTP-клиента в Node.js. Обработка ответа с файлом (download). Пример. Тестирование с помощью с Node.js-сервера.**

На клиенте создается WritebleStream. Потом в колбэке запроса мы с использованием метода pipe записываем в этот поток приходящий файл

1. **Разработка Websockets-приложения: Node.js-сервер, браузер-клиент. Пример.**





1. **Разработка Websockets-приложения: обработка json-сообщений, Node.js-сервер, Node.js-клиент. Пример.**

Бля, ну просто парсишь пришедшее сообщение с помощью JSON.parse

1. **Разработка RPC-Websockets-сервера. Пример. Тестирование: Node.js-клиент.**

|  |
| --- |
| const rpcWSS = require('rpc-websockets').Server  let server = new rpcWSS({port: 4000, host:'localhost'});  server.setAuth(credentials => credentials.login === 'admin' && credentials.password === 'admin');  server.register('square', (params) => {  return params.length === 2 ? params[0] \* params[1] : params[0] \*\* 2 \* Math.PI;  }).public();  server.register('sum', (params) => {  let sum = 0;  params.forEach(function (item, i, params) {  if(Number.isInteger(item)) sum += item;  });  return sum;  }).public();  server.register('mul', (params) => {  let mul = 1;  params.forEach(function (item, i, params) {  if(Number.isInteger(item)) mul \*= item;  });  return mul;  }).public();  function fib(n) {  var fibonacci = [0, 1];  for (i = 2; i < n; i ++) {    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];  }  return fibonacci;  }  server.register('fib', (param) => {  if (param.length === 1 ) {  console.log(param);  return fib(param)  } else {  return [1];  }  }).protected();  function factorial(n) {  return (n !== 1) ? n \* factorial(n - 1) : 1;  }  server.register('fact', (param) => {  if (param.length === 1) {  console.log(param);  return factorial(param)  }  else {  return [1];  }  }).protected(); |

|  |
| --- |
| const rpcWSC = WebSocket =  require('rpc-websockets').Client;  let ws = new rpcWSC('ws://localhost:4000');  ws.on('open', () => {      ws.call('square', [5]).then((r)=>{console.log('square = ', r);});      ws.call('square', [5, 4]).then((r)=>{console.log('square = ', r);});      ws.call('sum', [2]).then((r)=>{console.log('sum = ', r);});      ws.call('sum', [2, 4, 6, 8, 10]).then((r)=>{console.log('sum = ', r);});      ws.call('mul', [3]).then((r)=>{console.log('mul = ', r);});      ws.call('mul', [3, 5, 7, 9, 11, 13]).then((r)=>{console.log('mul = ', r);});      ws.login({login: 'admin', password: 'admin'})          .then((login) => {              ws.call('fib', [1]).catch((e)=>{console.log('catch fact: ',e)}).then((r)=>{console.log('fib = ', r);});              ws.call('fib', [2]).catch((e)=>{console.log('catch fact: ',e)}).then((r)=>{console.log('fib = ', r);});              ws.call('fib', [7]).catch((e)=>{console.log('catch fact: ',e)}).then((r)=>{console.log('fib = ', r);});              ws.call('fact', [5]).catch((e)=>{console.log('catch fact: ',e)}).then((r)=>{console.log('fact = ', r);});              ws.call('fact', [10]).catch((e)=>{console.log('catch fact: ',e)}).then((r)=>{console.log('fact = ', r);});          })     }); |

1. **Применение функции pipe для обработки данных (файла) файловой системы и записи в http-ответ. Пример.**

Pipe - это канал, который связывает поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи. Для чего они нужны? Возьмем, к примеру проблему копирования данных из одного файла в другой.

Однако задача записи в поток данных, считанных из другого потока, является довольно распространенной, и в этом случае pipes или каналы позволяют нам сократить объем кода:

|  |
| --- |
| const fs = require('fs')  let readableStream = fs.createReadStream(  'hello.txt',  'utf8'  )  let writeableStream = fs.createWriteStream('some2.txt')  readableStream.pipe(writeableStream) |

1. **Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического SELECT-запроса(лабораторная работа).**

**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)

**2-ой вопрос в билете**

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

**Redis:** noSQL СУБД с открытым кодом (BSD-лицензия), Redis Labs, Сальваторе Санфилиппо:

* хранилище данных в оперативной памяти;
* для кэша;
* для посредника сообщений;
* структуры данных: строки, хэш-таблицы, списки, наборы, отсортированные наборы, растровые изображения, геопространственные индексы, HyperLogLog;
* СУБД ориентирована на быстрое выполнение атомарных операций (до 100тыс. set/get-операций);
* механизм снимков для асинхронного сохранения (с потерями);
* механизм упреждающей записи;
* написана на ANSI С;
* последняя стабильная версия: 5**;**
* API: C, C++, C#, Java, JavaScript, Python, … ;
* работает только под Linux.

HyperLogLog - это алгоритм для задачи, связанной с подсчетом, аппроксимирующей количество различных элементов в мультимножестве. Для вычисления точного количества элементов мультимножества требуется объем памяти, пропорциональный количеству элементов, что нецелесообразно для очень больших наборов данных**.**

**Redis** – in memory Database - База данных размещаемая в оперативной памяти.

**Хеш-таблица (hash table)** — это специальная структура данных для хранения пар ключей и их значений. По сути это ассоциативный массив, в котором ключ представлен в виде хеш-функции.

**Персистентность данных** обеспечивается за счет того, что Каждая операция вдобавок запис.в расположенный на диске журнал транзакций

**Основные команды:**

**SET ключ значение** — Записывает строковое значение в переданный ключ. Если ключ до этого существовал, то он будет перезаписан.

**GET ключ** — Возвращает значение ключа.

Del ключ – удаление ключа.

**GETSET ключ значение** — Устанавливает в переданный ключ строковое значение и возвращает предыдущее.

**DECR ключ** — Уменьшает на единицу значение числа. В случае, если заданный ключ будет содержать строку, будет сгенерирована ошибка.

**INCR ключ** — Увеличивает на единицу значение числа. В случае, если заданный ключ содержит строку, будет вызвана ошибка.

**MSET ключ значение [ключ значение …]** — Устанавливает значение ключа / значения ключей, которые переданы в параметрах.

**MGET ключ [ключ …]** — Возвращает значение ключа / ключей, переданных в параметрах.

**HSET ключ поле значение** — Добавляет в хэш поле и значение. Если такого ключа не существовало, он будет добавлен. В случае, если такое поле в хэше уже существует, оно будет перезаписано.

**HGET ключ поле значение** — Возвращает значение, которое ассоциировано с полем в хэше

**HMSET ключ поле значение [поле значение …]** — Записывает значения в поля хэша.

**HMGET ключ поле [поле …]** — Получает значение поля / полей

указанного хэша.

EXISTS – уведомляет, если ключ существует.

1. **Применение пакета Sequelize. Основные принципы работы. Пример (лабораторная работа).**

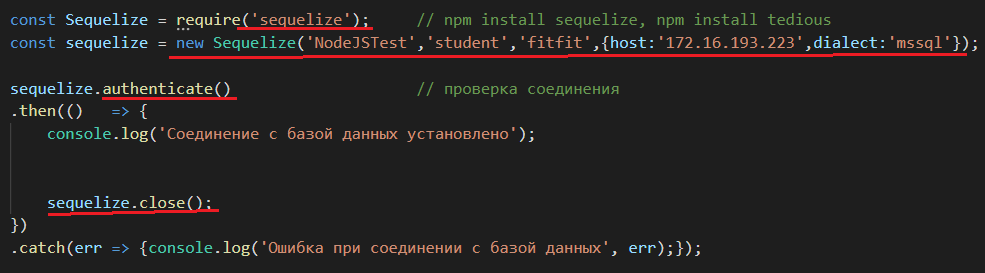
**ORM (Object-Relational Mapping)**  - технология программирования, которая позволяет работать с SQL-базой данных, как с набором программных объектов. **Mapping**: база данных – объект contextDB, таблица – коллекция объектов, строка в таблице – объект, структура таблицы – класс.

**Sequelize -** npm-пакет, реализующий ORM-технологию, кот.повзоляет работать с SQL базой данных.

Может применяться для: Postgres, MySQL, mariadb, sqlite3, Microsoft SQL Server.

**Объектам БД** (БД, таблицы, структуры, строки) ставятся в соотв.прогр.объекты.

1. **Sequelize:** соединение с БД, проверка соединения, закрытие соединения.

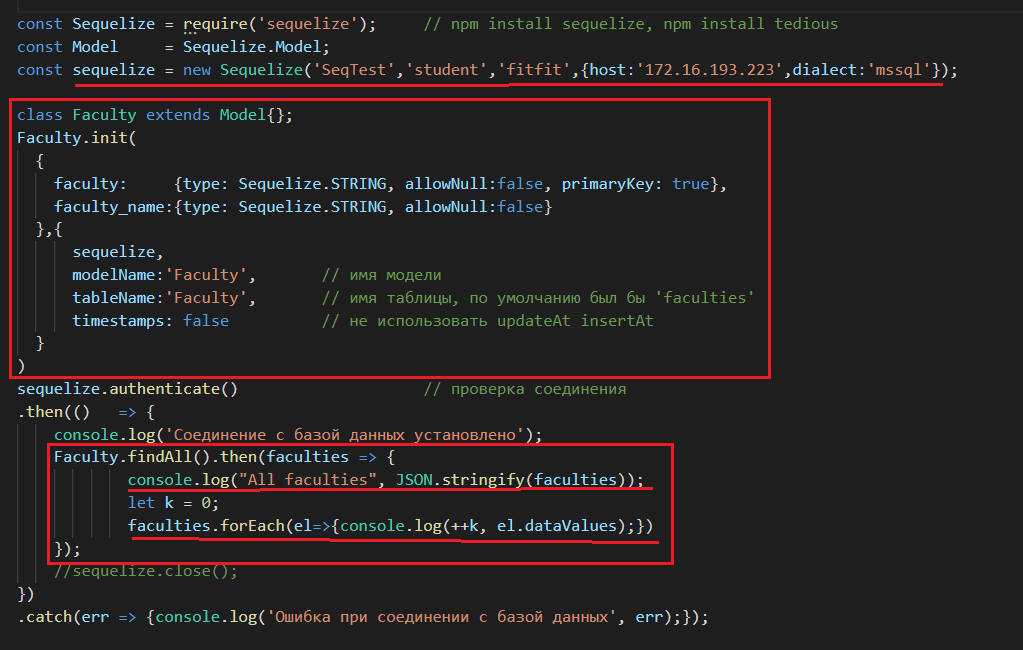
****

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

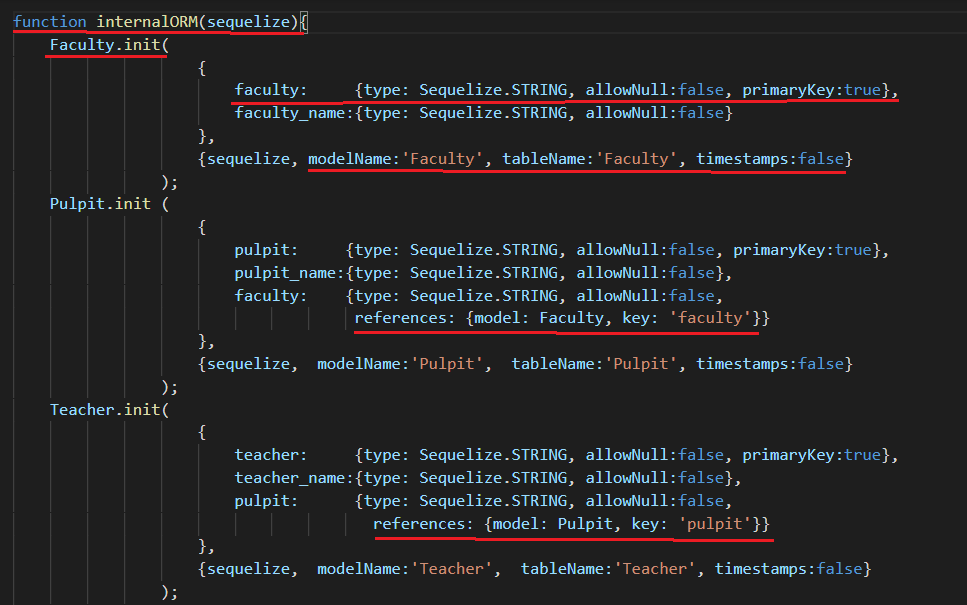
**Tedious** – протокол прикладного уровня, который использует mssql поверх TCP.

3) 1 – стра-ра таблицы, 2 – ук.модель

Timestamps – false: отслеж.измен.в табл(доп.поле будет созд или нет)

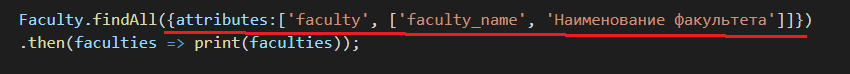
****

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

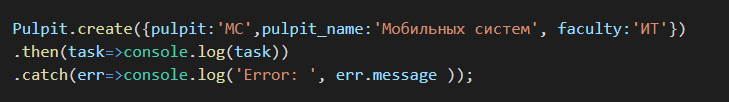
****

**5)** Операции

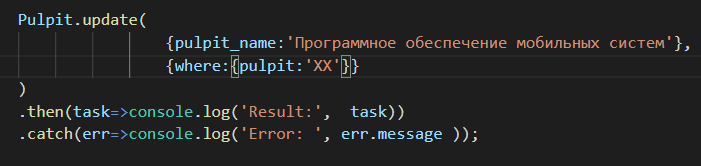
select – findAll()

****

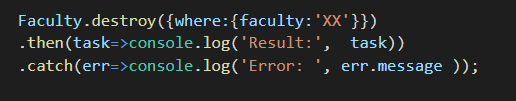
insert – create();

****

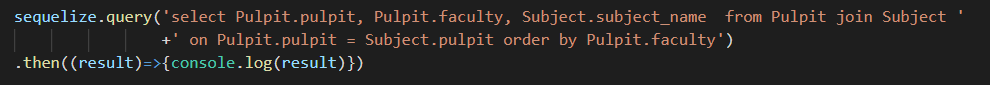
update – update();

****

delete – destroy();

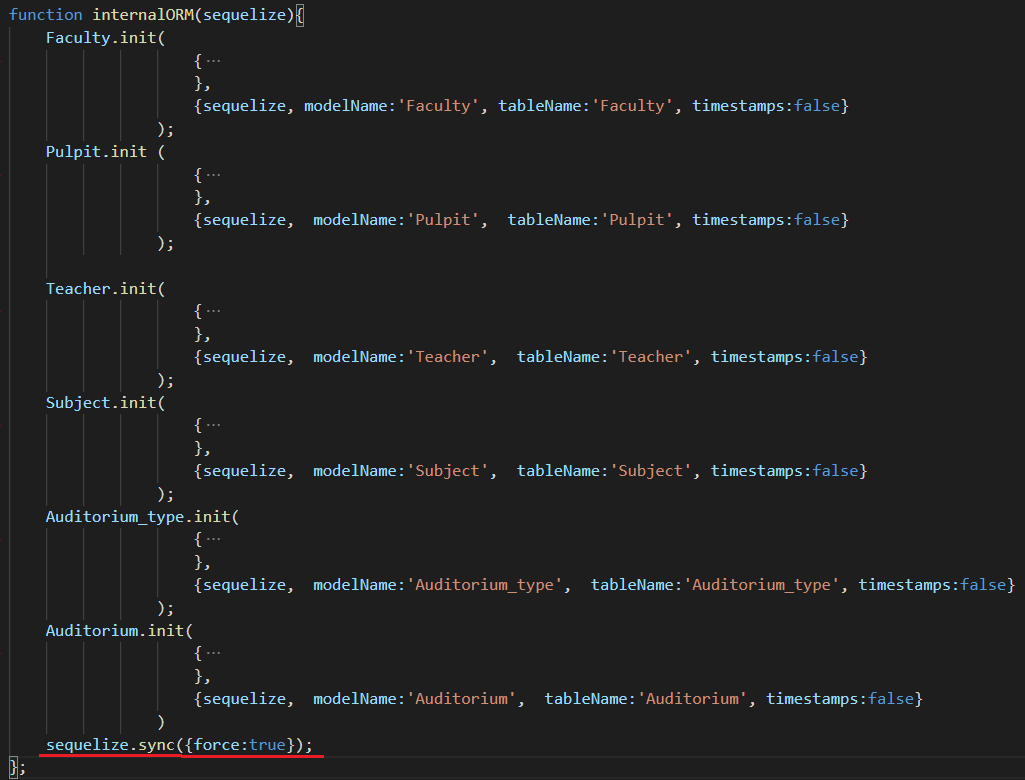
****

Query -

****

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

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

****

**30. Пакет Express. Основные принципы работы. Middleware-код. Пример.**

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

**Основные принципы работы**

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

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

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

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

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

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

**Middleware**

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

Функции промежуточной обработки могут выполнять следующие задачи:

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

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

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

Приложение Express может использовать следующие типы промежуточных обработчиков:

* Промежуточный обработчик уровня приложения
* Промежуточный обработчик уровня маршрутизатора
* Промежуточный обработчик для обработки ошибок
* Встроенные промежуточные обработчики
* Промежуточные обработчики сторонних поставщиков ПО

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

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

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

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

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

**31. Пакет Express. Основные принципы работы. Маршрутизация. Пример.**

Express и принципы работы в 30 вопросе.

**Маршрутизация**

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

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

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

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

**Пути маршрутов**

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

В Express для сопоставления путей маршрутов используется path-to-regexp для обработки шаблонов строк.

Строки: / , /users, /admin, … .

Шаблоны строк: ‘/ab?c’ = [abc, ac], ‘/ab+cd’ = [abcd, abbcd, abbbcd], ‘/ab\*cd’ = [abcd, ab123cd, abAAAAAcd];

Регулярные выражения: /a/, /.\*fly$/

**App.route и express.router()**

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



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

1. **Пакет Express. Основные принципы работы. Статические файлы. Пример.**

Express и принципы работы в 30 вопросе.

**Статические файлы**

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

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

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

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

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

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

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

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

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

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

1. **Пакет Express. Основные принципы работы. Обработка query-параметров GET-запроса. Пример (POSTMAN).**

Одним из способов передачи данных в приложение представляет использование параметров строки запроса. Строка запроса (query) - фактически это часть запрошенного адреса, которая идет после знака вопроса. Например, в запросе http://localhost:3000/about?id=3&name=Tome часть id=3&name=Tome представляет строку запроса.

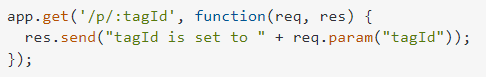
Строку запроса образуют параметры. После названия каждого параметра после знака равно (=) идет его значение. Друг от друга параметры отделяются знаком амперсанда. Например, в адресе выше использовалось два параметра: параметр "id" имеет значение "3" и параметр "name" имеет значение "Tom".

В express мы можем получить параметра строки запроса через свойство query объекта request, который передается в функцию обработки запроса.

С помощью выражения **request.query** мы можем получить все параметры строки запрос в виде объекта javascript, а с помощью выражения **request.query.[название\_параметра]** мы можем обратиться к каждому отдельному параметру.

1. **Пакет Express. Основные принципы работы. Обработка uri-параметров запроса. Пример (POSTMAN).**

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



Для получения параметров используется объект запроса и свойство params.

1. **Пакет Express. Основные принципы работы. Обработка body-параметров POST-запроса. Пример (POSTMAN).**

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

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

App.post(‘/’, (req, res) => {

Const body = req.body;

Console.log(body);

...

});

1. **Пакет Express. Основные принципы работы. Обработка json-данных POST-запроса. Пример (POSTMAN).**

Аналогично как в 35

1. **Пакет Express. Основные принципы работы. Обработка xml-данных POST-запроса. Пример (POSTMAN).**

Для обработки xml-данных в теле запроса можно использовать модуль express-xml-parser. Устанавливается он через менеджер пакетов NPM.

Для того, что бы использовать данный модуль для обработки xml-данных его нужно подключить как middleware:

const xmlparser = require("express-xml-bodyparser");

app.use(xmlparser());

Данный middleware преобразует xml в json, что позволяет намного проще с ним работать.

Для получения данных внутри кода используется следующий метод:

let x = req.body.calculate.x.$.value;

Здесь мы получаем доступ к данных из тела запроса, потом содержимому из элемента calculate и значению атрибута value из элемента x. Здесь $ обозначает корень каждого элемента.

1. **Пакет Express. Основные принципы работы. download/attachment файлы GET-запроса. Пример (браузер).**

Метод attachment устанавливает заголовок ответа Content-Desposition : attachment

Если в функцию передано имя файла, то устанавливается 2 заголовка :

* **Content-Desposition : attachment; filename = [filename]**
* **Content-Type : image/[format]**

Формат для тега Content-Type выбирается автоматически на основе расширения имени фала, переданного в функции attachment().

Метод res.download отправляет по http файл, переданный в качестве параметра и на клиенте он автоматически скачивается и сохраняется с тем именем, которое было передано в качестве параметра. Так же устанавливаются 2 тега приведенные выше.

1. **Пакет Express. Основные принципы работы. upload файла в POST-запросе. Пример (браузер).**

Для отправки фала на сервер можно использовать форму с параметром enctype=”multipart/form-data”.

Тип содержимого multipart/form-data — это составной тип содержимого, чаще всего использующийся для отправки HTML-форм с бинарными (не-ASCII) данными методом POST протокола HTTP.

Для обработки пришедшего запроса с составным содержимым можно использовать модуль multipart

При подключении указывается объект параметров, пример:

const upload = multer({ dest: 'uploads/' })

Здесь параметр dest указывает директорию, куда будут сохраняться файлы.

Так же в качестве параметров можно указать следующие параметры:

* fileFilter – функция, которая контролирует какие файлы могут приниматься.
* Limits – пределы принимаемой информации.
* preservePath – сохранять ли полный путь к файлу, а не только имя.

Использование происходит следующим способом. Мы используем объект multer в качестве middleware в функции обработчике запроса. При этом мы указываем режим, который определяет кол-во принимаемых файлов:

* single(fieldname) – принимает один файл, ключ которого в теле должен совпадать с [fieldname]
* array(fieldname[, maxCount]) – принимает массив изображений, не обязательный параметр – максимальное кол-во
* none() – принимает только текстовые поля
* any() – принимает любые переданные поля

**DiskStorage и MemmoryStorage**

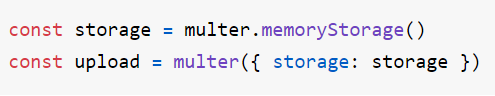
**DiskStorege** дает полный контроль над сохранением файлов на диск.



**Destination** – функция, которая определяет в какую директорию будут сохраняться файлы. Если параметр не задан, то файлы будут сохраняться в директории системы для временных файлов.

**Filename** – функция, которая определяет имя сохраняемого файла в директории. Если не задана то файлу будет дано случайное имя без расширения файла.

**MemmoryStorage** – сохраняет файлы в оперативной памяти в качестве объектов Buffer.



Стоит обратить внимание на то, что если будут поступать большие файлы или файлы малого размера, но в большом кол-ве, то у приложения может закончится выделенная оперативная память.

**Доступ к файлу и его свойства**

Для доступа к файлу внутри обработчика можно использовать req.file

Этот объект содержит следующие свойства:

* fieldname – имя поля, определенное в форме на клиенте.
* originalName – оригинальное имя фала.
* Encoding – кодировка
* Mimetype – MIME тип
* Size – размер в байтах

1. **Пакет Express. Основные принципы работы. Обработка Cookie. Signed cookie. Пример(POSTMAN).**

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

В Node.js cookie управляются двумя методами объекта ответа:

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

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

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

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

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

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

**Параметры передаваемые в cookie()**

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

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

1. **Пакет Express. Основные принципы работы. Применение объекта Session для сохранение состояния. Пример (POSTMAN).**

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

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

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

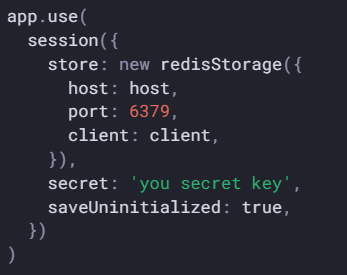
App.use(session({params}))

В качестве параметров сесии может быть передано следующее:

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

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

Для примера можно использовать redis



1. **Пакет Express. Основные принципы работы. Переадресация. Пример(POSTMAN).**

Для переадресации применяется метод **redirect()**:

**redirect([status,] path)**

В качестве параметра path передается путь, на который будет перенаправляться пользователь. Дополнительный параметр status задает статусный код переадресации. Если этот параметр не задан, тогда по умолчанию отправляется статусный код 302, который предполагает временную переадресацию.

С помощью данного метода можно выполнять переадресацию как по относительным путям, так и по абсолютным, в том числе на другие домены.

Примеры :

* Абсолютный путь - **response.redirect('https://xsltdev.ru')**
* Относительно текущего пути - **response.redirect('about')**
* Относительно текущего ресурса - **response.redirect('/about')**
* Относительно текущего адреса на адрес на том же уровне - **response.redirect('./about')**
* Переадресация на адрес, который располагается уровнем выше - **response.redirect('../about')**
* Переадресация на уровень выше - **response.redirect('.')**
* Переадресация на два уровня выше - **response.redirect('..')**

1. **Пакет Express. Основные принципы работы. Выполнение shell-команд (spawn, pipe). Пример.**

Spawn – ф-ция, которая позволяет запускать консоль, с помощью кот можно запускать любые приложения.

Мы создаем процесс cmd.exe а он в свою очередь делает CreateProcess для других процессов => такие процессы для cmd.exe – дочерние.

Метод child\_process.spawn запускает новый процесс с помощью заданной командой. Он содержит следующую подпись:

**child\_process.spawn(command[, args][, options])**

**Пример:**



Каждый процесс имеет потоки ввода, вывода и вывода ошибок.

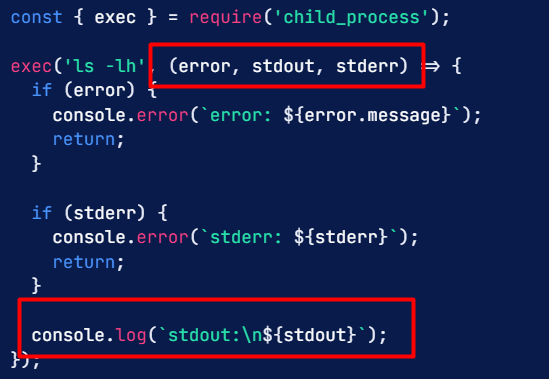
Мы можем связать поток вывода 1 процесса с потоком ввода другого процесса. Для этого можно использовать функцию pipe(), которая применяется к потоку ввода/вывода для передачи данных другому потоку. Пример выше – поток вывода dir связывается с потоком ввода findstr. То есть когда dir найдет данные, то он передаст их не нам в консоль, а в процесс findstr, а уже findstr передаст нам в консоль.

1. **Пакет Express. Основные принципы работы. Запуск процесса операционной системы (exec), работа со стандартными потоками ввода/вывода. Пример.**

Exec используется для создания нового процесса, при этом он создает оболочку в которой исполняются команды (чего не делает spawn).

Это упрощает то, что нам не надо самостоятельно указывать оболочку для запуска

Вывод команды хранится в буфере в памяти доступ к которому можно получить с помощью callback функции



1. **Пакет Express. Основные принципы работы. Выполнение js-скриптов в отдельном процессе (fork, send, worker). Пример.**

Fork и worker threads используются для запуска скриптов в другом процессе.

Fork создает новый экземпляр ноды. Преимущество перед spawn в том, что каналы связу между процессами в форке уже установлены и для пересылки сообщений можно использовать функцию send() а в чайлд процессе обрабатывать событие process.on(‘message’, (message) => {});

Это возможно из-за использования механизма межпроцессорного взаимодействия (IPC);

В отличии от форка, воркеры предназначены для ЦПУ-нагружаемых задач и не помогают с задачами интенсивного ввода вывода.

Для пересылки сообщений воркеры используют буферы.

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

**WebDAV** – расширение протокола HTTP/HTTPS, поддерживающее совместную работу по управление файлами на удаленных web-северах; применяется для создания сетевой файловой системы; в системах документооборота (document management system)

WebDAV: унаследованные HTTP-методы

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

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

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

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

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

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

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

* Jsonrpc – версия json prc (2.0)
* Method – то как мы сами назвали процедуру на стороне С
* Params – поле, кот сод.пар-ры (массив/объект/мб не быть)
* Id - уник. С выполн эти процедуру и отвеч за результаты, ссылаясь на Id – надо обесп с-му, кот обесп уникальность – GUID, напр (генер 128-бит посл-сти)) – необяз., т.к. бывают вызовы процедур не требующие ответа (уведомления)

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

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

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

**Типы параметров:**

* Позиционные – продполаг., что в params перед массив, в кот к. эл-т – параметр

(“params”: [42,23] – это 1й, 2й парам)

* Именованные – в этом случае в params передается объект, кот сост.из 2х св-в (пара ключ-значение); можно размещать вне зависимости от порядка, т.к. процедуры ориентированы на название параметра

(“params” : {“subs” : 23, “mid” : 42}

1. **Разработка клиент-серверного приложения с применением технологии WebAssembly на стороне браузера. Пример(WasmFiddle-компиляция).**

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

• Код быстрее, чем JS

• Поддерживается большинством браузеров

• Выполняется в sandbox

• Есть отладчики

• Открытый стандарт

**Идея:** готовим бинарный код исполняемого файла, этот код переносим на сторону клиента и он исполняется на стороне клиента (в JS Engine);

=> разраб.прогу => компилируем => получаем byte-код, кот.мб исполняться в рамках JS Engine => byte-код перетаскиваем на сторону клиента и запускаем на исполнение

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

• Удобный

• Помогает разрабатывать бин.код., а также посмотреть как устроен JS API

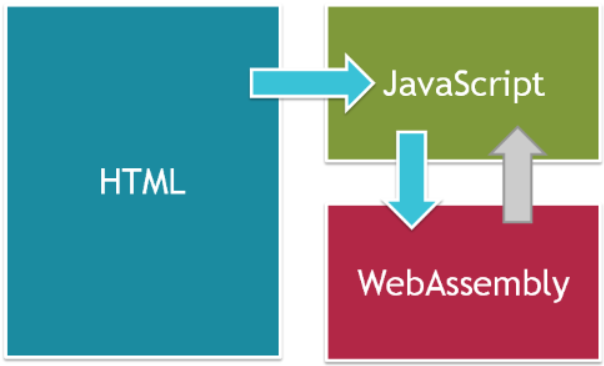
• Массив wasmcode = unit8Array – откомпилированные функции, т.е.byte-код функций

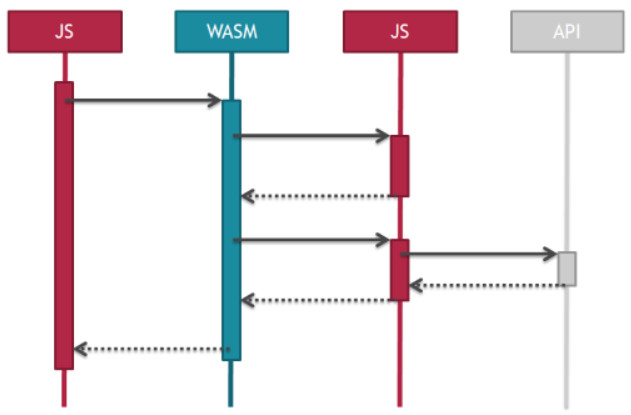
• Export – получает инстанс, который позволяет выполнять эти ф-ции.

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

Браузер загружает html страницу, с которой выпоняется JS, который уже выполняет загрузку WebAssembly – получатся «модуль» (WebAssembly module), затем создает экземпляр модуля, после чего можно вызывать для этого экземпляра экспортируемые функции.

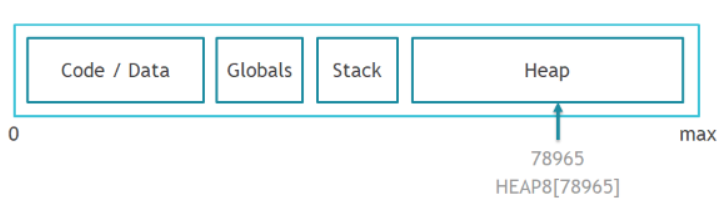
Серая стрелка – изнутри WebAssembly можно вызвать ф-ции JS.



****

Здесь мы сначала из JS вызываем WebAssembly, затем из WebAssembly вызываем ф-цию JS. WebAssembly может пользоваться любыми АПИ. Это возможно, но не напрямую, т.к.такие вызовы происходят через JS.

Модель памяти WebAssembly очень проста. Это плоский «кусок» памяти, в котором находится код проги, глобальные переменные, стек и куча. Есть возможность сделать так, чтобы память была расширяемой, т.е. если при очередном выделении памяти нам не хватает места, то верхняя граница памяти автоматически увеличивается.

****

Весь блок памяти доступен из дж/с просто как массив байтов. Причем эта память доступна как на чтение, так и на запись.

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

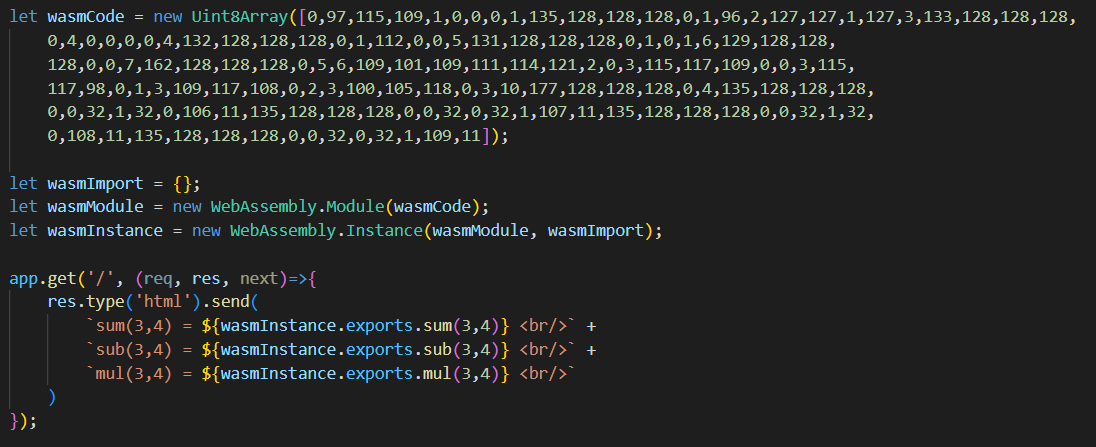
* WasmFiddle – сильная оптимизация
* Emcc – компилятор, который позволяет компилировать код из с++ в wasm

1. **Разработка клиент-серверного приложения с применением технологии WebAssembly на стороне сервера Node.js. Пример(WasmFiddle-компиляция).**

Для работы с WebAssembly нам нужен откомпилированный файл (который представляет собой набор байт при чтении через fs) или просто набор байт в Unit8Array.

Для начала мы создаем объект WebAssembly.Module([code]) который содержит код, который уже был откомпилирован. Используется для дальнейшего распространения (к примеру в worker)

Далее мы создаем Объект экземпляра WebAssembly.Instance([module],[objForImport]). В объекте импорта будут содержаться все функции, которые позволяют вызывать код WebAssembly из JS



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

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

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

**Как работает long poll**

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