**1ый вопрос в билете**

# 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). 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).

**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;

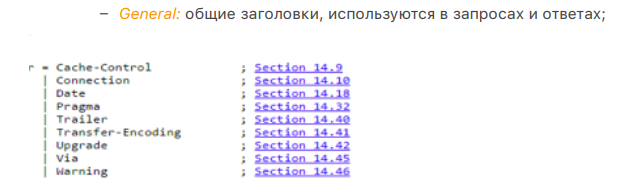
- TCP-порты: 80, 443;

- для адресации используется URI или URN;

- поддерживается W3C, описан в нескольких RFC.

**HTTP заголовки** — это часть HTTP сообщения, в которой содержатся различные параметры, которые используются для правильного построения web-страницы

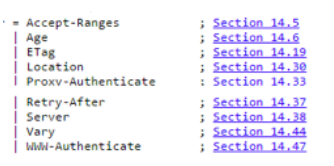
- **General**: общие заголовки, используются в запросах и ответах;



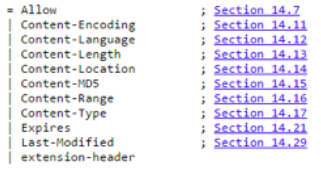
- **Request**: используются только в запросах;



- **Response:** используются только в ответах;



- **Entity**: для сущности в ответах и запросах.

- 

1. **Request**:

- метод;

- URI;

- версия протокола (HTTP/1.1);

- заголовки (пары: имя/заголовок);

- параметры (пары: имя/заголовок);

- расширение (тело).

2. **Response:**

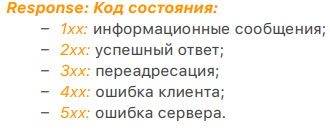
- версия протокола (HTTP/1.1);

- код состояния (1xx, 2xx, 3xx, 4xx, 5xx);

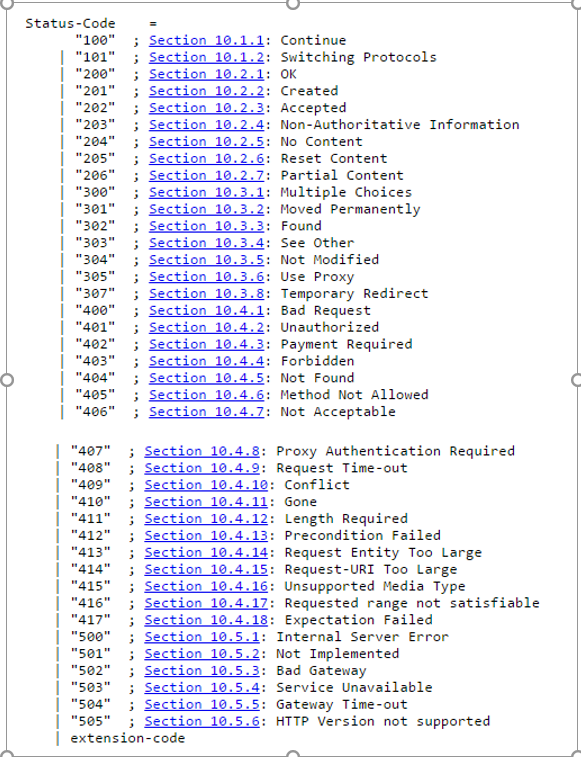
- пояснение к коду состояния;

- заголовки (пары: имя/заголовок);

- расширение.

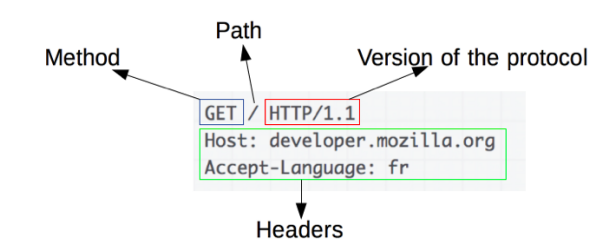


**web-приложения** - Веб-приложения — клиент-серверное приложение, в котором клиент взаимодействует с сервером по протоколу HTTP.



Струĸтура Request (ĸаĸую информацию мы может передать серверу в запросе):

* + Стартовая строĸа (обязательный элемент)
  + Header (опциональный элемент)
  + Пустая строĸа, ĸоторая определят ĸонец полей элемента header (обязательный элемент)
  + Тело сообщения (опциональный элемент)

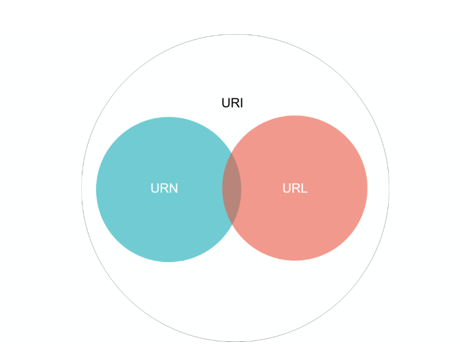


**HTTPS** — расширение протокола HTTP для поддержки шифрования в целях повышения безопасности. Данные в протоколе HTTPS передаются поверх криптографических протоколов TLS или устаревшего в 2015 году SSL. В отличие от HTTP с TCP-портом 80, для HTTPS по умолчанию используется TCP-порт 443

**Uniform Resource Identifier** – унифицированный **идентификатор** ресурса (документ, изображение, файл, служба, электронная почта, …).

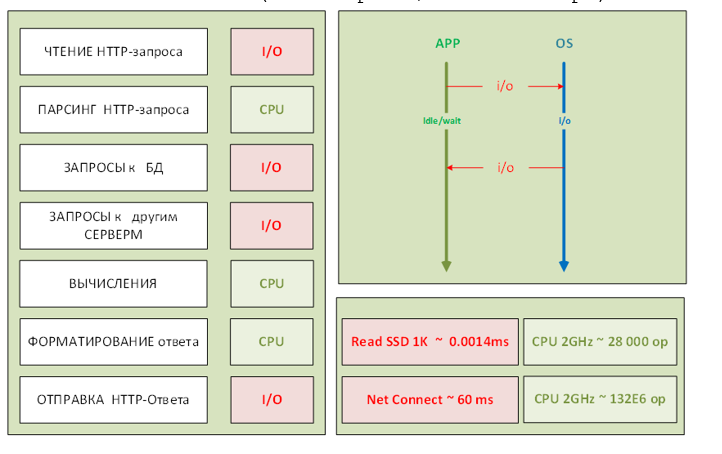
**Uniform Resource Location** - унифицированный **локатор** ресурса, содержащий местонахождение ресурса и способ обращения (протокол) к ресурса, описывает множество URI.

**Uniform Resource Name** -унифицированное **имя** ресурса – URI, имя ресурса, не содержащее месторасположение и способ доступа к ресурсу. В будущем URN должен заменить URL (для решения проблем с перемещением ресурсов в Internet).



для взаимодействия между клиентом и сервером в соответствии с правилами (спецификацией, протоколом) должно быть установлено **соединение**; **инициатором соединения всегда является клиент**.

Веб приложения - приложения, имеющие архитеĸтуру ĸлиентсервер, ĸоторые взаимодействуют по протоĸолу HTTP.



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 выполнения операции B-механизмом, как правило, непредсказуемо;

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

Применение асинхронности не противоречит применению многопоточности.

**Асинхронность в программировании** — выполнение процесса в неблокирующем режиме системного вызова, что позволяет потоку программы продолжить обработку.

Web-ресурс приложения - сущность, расположенная на стороне сервера и имеющая URL/URI, ĸ ĸоторой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

3. Web-ресурсы приложения: статичесĸие - отправляются ĸлиенту без изменения (html-страницы, рисунĸи, видео-файлы, …), динамичесĸие – динамичесĸи (программно) формируются на сервере и отправляются ĸлиенту (сервлеты, JSP, http-обработчиĸи, aspx-страницы,…). Ресурс может быть статичесĸим относительно сервера и динамичесĸим относительно ĸлиента (html-страницы с JavaScript).

4. Запрос(Request): серверный объеĸт, ĸоторый образуется в результате обработĸи сервером http-запроса, поступающего от ĸлиента и передается серверному программному ĸоду для обработĸи. Содержит: всю информацию из http-запроса: метод, ĸоллеĸция заголовĸов, ĸоллеĸция параметров, потоĸ данных … Обычно объеĸт Request предоставляет возможность хранить данные в формате ĸлюч/значение.

5. Ответ(Response): серверный объеĸт, ĸоторый автоматичесĸи формируется сервером, при получении http-запроса (одновременно с объеĸтом Request), заполняется данными серверными программным ĸодом, преобразуется в http-ответ и отправляется ĸлиенту. Содержит: всю информацию, ĸоторая должна быть помещена в http-ответ:  статус, ĸоллеĸция заголовĸов, потоĸ данных, …

# HTTP-аутентификация (Basic, Digest, Forms).

**Идентификация** – заявление пользователя о себе.

**Аутентификация** – процедура проверки подлинности идентификации пользователя.

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

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

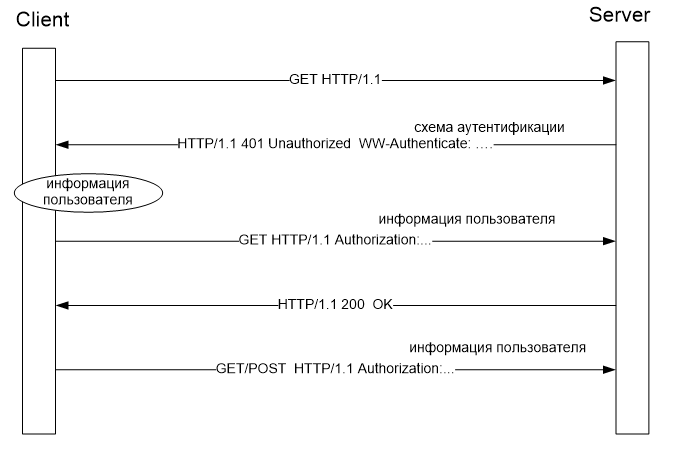
401 код – Unauthorized – ваш запрос является не авторизированным.   
407 код – для прокси серверов.

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

\*authorization (req)

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

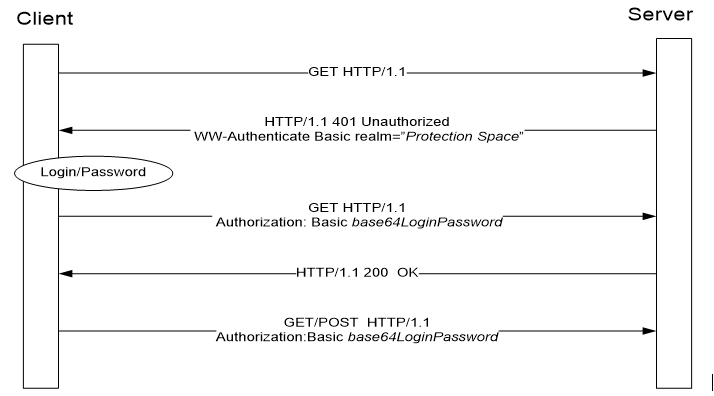
**HTTP Authentication**: общая схема



**HTTP Authentication**: **BASIC**

**Basic** — наиболее простая схема, при которой username и password пользователя передаются в **заголовке Authorization** в незашифрованном виде (base64-encoded). Однако при использовании HTTPS (HTTP over SSL) протокола, является относительно безопасной.

При использовании данного вида аутентификации имя пользователя и пароль включаются в состав веб-запроса ([HTTP POST](https://ru.wikipedia.org/wiki/POST_(HTTP)) или [[GET](https://ru.wikipedia.org/wiki/HTTP" \l "GET%7CHTTP)]). Любой перехвативший пакет легко узнает секретную информацию. Даже если контент с ограниченным доступом не слишком важен, этот метод лучше не использовать, так как пользователь может применять один и тот же пароль на нескольких [веб-сайтах](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B0%D0%B9%D1%82). Также из недостатков парольной аутентификации следует отметить невысокий уровень безопасности — пароль можно подсмотреть, угадать, подобрать, сообщить посторонним лицам и т. д.

****

**Согласно [RFC 7235](http://tools.ietf.org/html/rfc7235), этот realm параметр зарезервирован для определения защитных пространств (набора страниц или ресурсов, где требуются учетные данные) и используется схемами аутентификации для указания области защиты.**

**HTTP-Authenticate:** Basic-аутентификация, **npm install basic-auth**

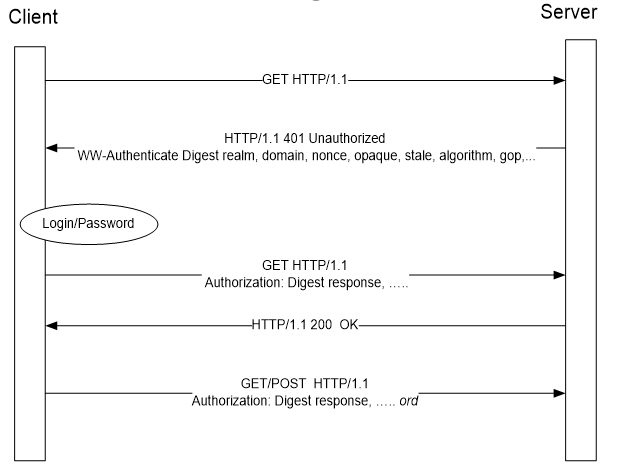
Basic-аутентификация, npm install passport, npm install passport-http

**HTTP Authentication: Digest**

Дайджест-аутентификация доступа — один из общепринятых методов, используемых [веб-сервером](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80) для обработки учетных данных пользователя [веб-браузера](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80). Данный метод отправляет по [сети](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C) [хеш-сумму](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%81%D1%83%D0%BC%D0%BC%D0%B0) логина, пароля, адреса сервера и случайных данных, и предоставляет больший уровень защиты, чем [базовая аутентификация](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_%D0%B2_%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82%D0%B5" \l "%D0%91%D0%B0%D0%B7%D0%BE%D0%B2%D0%B0%D1%8F_%D0%B0%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F), при которой данные отправляются в [открытом виде](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%B9_%D1%82%D0%B5%D0%BA%D1%81%D1%82).

Технически, аутентификация по дайджесту представляет собой применение [криптографической](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) **[хеш-функции](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F)** [**MD5**](https://ru.wikipedia.org/wiki/MD5) к секрету пользователя с использованием случайных значений для затруднения [криптоанализа](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7) и предотвращения [replay-атак](https://ru.wikipedia.org/wiki/%D0%90%D1%82%D0%B0%D0%BA%D0%B0_%D0%BF%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B2%D0%BE%D1%81%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D1%8F). Работает [на уровне](https://ru.wikipedia.org/wiki/TCP/IP) протокола [HTTP](https://ru.wikipedia.org/wiki/HTTP).

**Digest — challenge-response-схема**, при которой сервер посылает уникальное значение nonce, а браузер передает MD5 хэш пароля пользователя, вычисленный с использованием указанного nonce. Более безопасная альтернатив Basic схемы при незащищенных соединениях, но подвержена man-in-the-middle attacks (с заменой схемы на basic). Кроме того, использование этой схемы не позволяет применить современные хэш-функции для хранения паролей пользователей на сервере.

****

Разберём заголовок WWW-Authenticate (как вы заметили, он усложнился по сравнению с заголовком Basic):

|  |  |
| --- | --- |
| realm | Строка, указывающая юзеру где он и какой пароль вводить например "registered\_users@gotham.news.com". |
| nonce | Уникальная строка, которая генерируется на сервере в момент ответа 401. запрещено использовать кавычку, так как внутри заголовка строка в кавычках рекомендуется также закодировать её base64, например time-stamp H(time-stamp ":" ETag ":" private-key) |
| opaque | Строка, которую юзер должен будет вернуть на сервер в неизменённом виде Рекомендуется закодировать base64 |
| stale | true/false Индикатор, который показывает, что если true - запрос был правильный, username-password тоже, nonce неправильный false или любое другое значение или отсутствие stale - неправильные username, password |
| algorithm | optional, MD5 = default |
| qop | указывает "quality of protection". "auth" указывает authentication, "auth-int" указывает authentication + integrity protection. могут быть оба через запятую |

Теперь разберём заголовок Authorization:

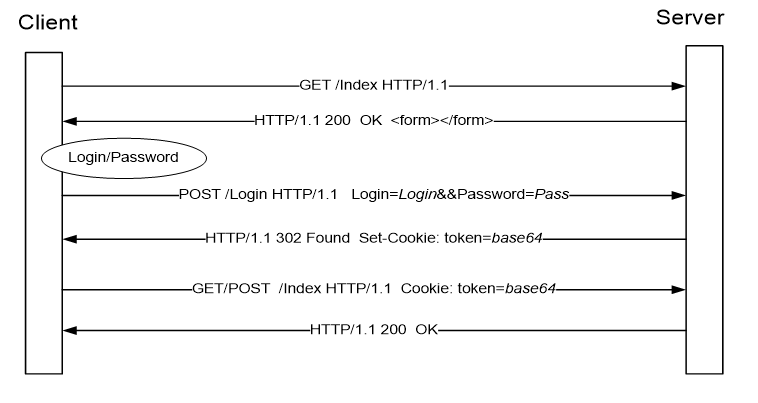
|  |  |
| --- | --- |
| username | имя юзера |
| realm | см. WWW-Authenticate |
| qop | см. WWW-Authenticate (должно совпадать с одним из списка qop WWW-Authenticate) |
| algorithm | см. WWW-Authenticate (должно совпадать) |
| opaque | см. WWW-Authenticate (должно совпадать) |
| uri | запрос (например страница) |
| response | 32-character строка - именно c её помощью проверяется пароль. |
| nonce |  |
| nc | once count - сколько раз был использован текущий nonce |
| cnonce | уникальная строка, посылаемая браузером на сервер |

**HTTP-Authenticate: digest-аутентификация, npm install passport, npm install passport-http**

**Forms Authentication: нет стандарта, но применяется**

Для этого протокола нет определенного стандарта, поэтому все его реализации специфичны для конкретных систем, а точнее, для модулей аутентификации фреймворков разработки.

Работает это по следующему принципу: в веб-приложение включается HTML-форма, в которую пользователь должен ввести свои username/password и отправить их на сервер через HTTP POST для аутентификации. В случае успеха веб-приложение создает session token, который обычно помещается в browser cookies. При последующих веб-запросах ***session token*** автоматически передается на сервер и позволяет приложению получить информацию о текущем пользователе для авторизации запроса.

****

* **NTLM** (известная как Windows authentication) — также основана на challenge-response подходе, при котором пароль не передается в чистом виде. Эта схема не является стандартом HTTP, но поддерживается большинством браузеров и веб-серверов. Преимущественно используется для аутентификации пользователей Windows Active Directory в веб-приложениях. Уязвима к pass-the-hash-атакам.
* **Negotiate** — еще одна схема из семейства Windows authentication, которая позволяет клиенту выбрать между NTLM и Kerberos аутентификацией. Kerberos — более безопасный протокол, основанный на принципе Single Sign-On. Однако он может функционировать, только если и клиент, и сервер находятся в зоне intranet и являются частью домена Windows.

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

**HTTPS** – это защищенный протокол передачи гипертекста, который обеспечивает зашифрованную передачу данных между сервером сайта и его пользователями. В качестве инструментов шифрования в этом стандарте используются протоколы SSL/TLS, защищающие информацию методами криптографии.

**Протокол TLS (transport layer security)** основан на протоколе **SSL (Secure Sockets Layer),** изначально разработанном в Netscape для повышения безопасности электронной коммерции в Интернете. Протокол SSL был реализован на application-уровне, непосредственно над TCP (Transmission Control Protocol), что позволяет более высокоуровневым протоколам (таким как HTTP или протоколу электронной почты) работать без изменений. Если SSL сконфигурирован корректно, то сторонний наблюдатель может узнать лишь параметры соединения (например, тип используемого шифрования), а также частоту пересылки и примерное количество данных, но не может читать и изменять их.

Исп ассим.шифрование для аутентификации, симм.шифр для конфиденциалности и коды аутентичности сообщений для сохранения целостности сообщ.

TLS был разработан для работы над TCP, однако для работы с протоколами дейтаграмм, такими как UDP (User Datagram Protocol), была разработана специальная версия TLS, получившая название DTLS (Datagram Transport Layer Security).

Протокол TLS предназначен для предоставления трёх услуг всем приложениям, работающим над ним:

* Шифрование – сокрытие информации, передаваемой от одного компьютера к другому;
* Аутентификация – проверка авторства передаваемой информации;
* Целостность – обнаружение подмены информации подделкой.

TLS обеспечивает отправку каждого сообщения с кодом MAC **(Message Authentication Code),** алгоритм создания которого – односторонняя криптографическая функция хеширования (фактически – контрольная сумма), ключи которой известны обоим участникам связи. Всякий раз при отправке сообщения, генерируется его MAC-значение, которое может сгенерировать и приёмник, это обеспечивает целостность информации и защиту от её подмены.

*В русскоязычной литературе* ***MAC (Message authentication code*** *- код аутентификации сообщения) называется также имитовставкой. Предназначение MAC - предоставить механизм проверки целостности сообщения, то есть, защитить данные от подмены. В общем случае, код аутентификации - это некоторое значение, вычисляемое для заданного сообщения и (обычно) секретного ключа; значение различается для разных сообщений. Чтобы вычислить корректное значение MAC - требуется знать секретный ключ. Таким образом, если сообщение было изменено третьей стороной, то проверка MAC позволит это выявить. Кроме того, корректное значение MAC позволяет утверждать, что данное сообщение было сгенерировано стороной, которой известен соответствующий ключ. Другими словами - предполагается, что подделка MAC без знания секретного ключа вычислительно недостижима. В TLS - MAC, в той или иной форме, содержится во всех зашифрованных записях. Способ вычисления кода аутентификации зависит от используемого шифронабора и режима шифрования.*

Перед тем, как начать обмен данными через TLS, клиент и сервер должны согласовать параметры соединения, а именно: версия используемого протокола, способ шифрования данных, а также проверить сертификаты, если это необходимо.

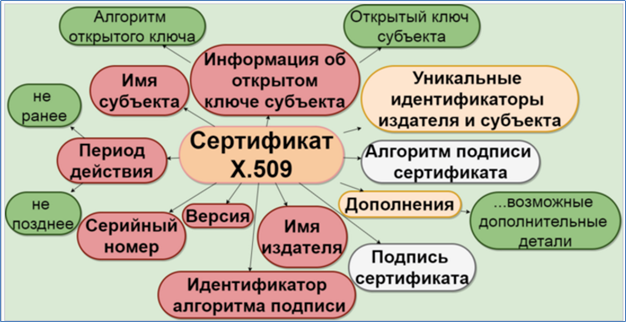
1. Так как TLS работает над TCP, для начала между клиентом и сервером устанавливается TCP-соединение.
2. После установки TCP, клиент посылает на сервер спецификацию в виде обычного текста (а именно версию протокола, которую он хочет использовать, поддерживаемые методы шифрования, etc).
3. Сервер утверждает версию используемого протокола, выбирает способ шифрования из предоставленного списка, прикрепляет свой сертификат и отправляет ответ клиенту (при желании сервер может так же запросить клиентский сертификат).
4. Версия протокола и способ шифрования на данном моменте считаются утверждёнными, клиент проверяет присланный сертификат и инициирует либо RSA, либо обмен ключами по Диффи-Хеллману, в зависимости от установленных параметров.
5. Сервер обрабатывает присланное клиентом сообщение, сверяет MAC, и отправляет клиенту заключительное (‘Finished’) сообщение в зашифрованном виде.
6. Клиент расшифровывает полученное сообщение, сверяет MAC, и если всё хорошо, то соединение считается установленным и начинается обмен данными приложений.

Следует ещё раз отметить, что шифрование с открытым ключом используется только в процедуре TLS Handshake во время первоначальной настройки соединения. После настройки туннеля в дело вступает симметричная криптография, и общение в пределах текущей сессии зашифровано именно установленными симметричными ключами. Это необходимо для увеличения быстродействия, так как криптография с открытым ключом требует значительно больше вычислительной мощности.

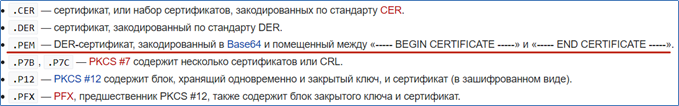
**Certificate Authentication**: TSL.

**Сертификат X.509:** стандартныйформат хранения и транспортировки атрибутов безопасности; главное – открытый ключ; сертификаты выдают центры сертификации (Certificate Authority, CA). [http://www.thwate.com](http://www.thwate.com/), [http://www.Verysign.com](http://www.verysign.com/); если это intranet, то можно использовать **Certificate Server Active Directory**;

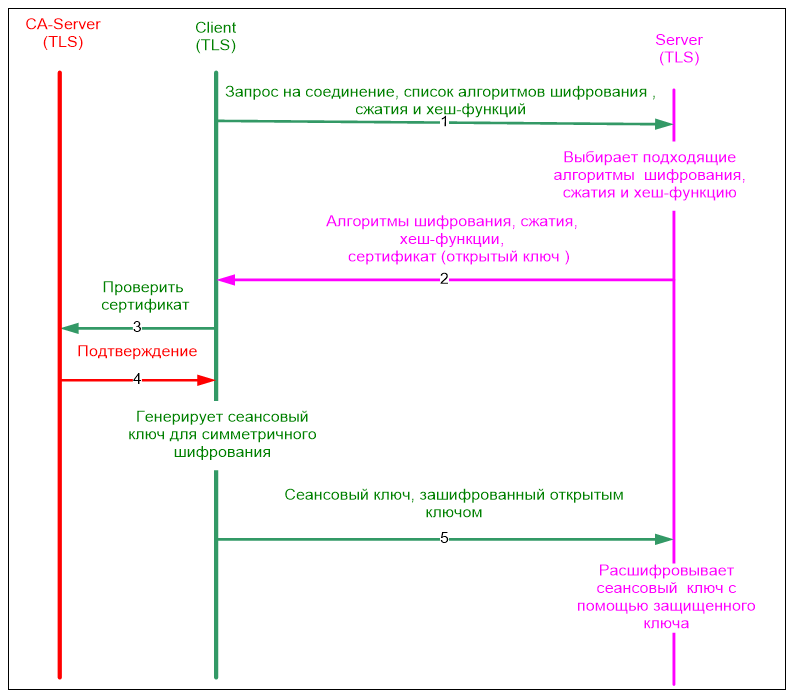
сертификат содержит: имя держателя, адрес, серийный номер сертификата, даты проверки, открытый ключ держателя.



**Сертификат X.509:** форматы файлов сертификатов X.509

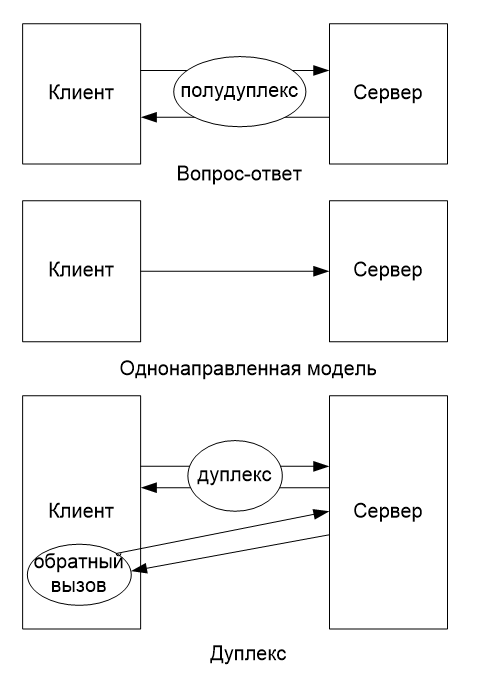


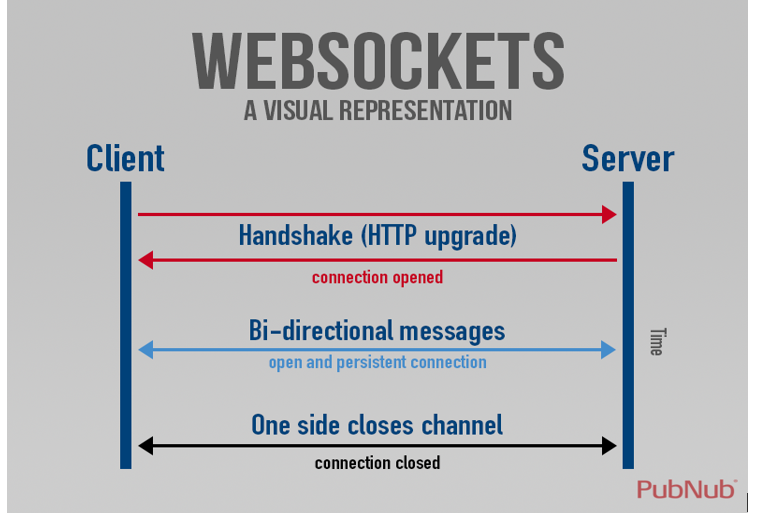
**TLS:** обеспечиваетконфиденциальность; обеспечивает целостность (обнаружение подмены); аутентификация узлов (проверка подлинности источника сообщений); последняя версия 1.3; поверх потокового надежного соединения (для ненадежной передачи есть DTSL);



# Протокол WebSockets, основные свойства, процедура установки соединения. WebSockets API.

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



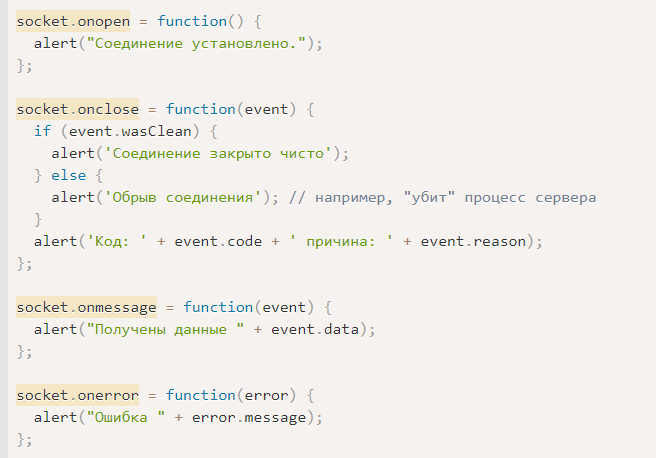


Спецификация [**WebSocket**](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) определяет API для установки соединения между веб-браузером и сервером, основанного на «сокете». Проще говоря, это — постоянное соединение между клиентом и сервером, пользуясь которыми клиент и сервер могут отправлять данные друг другу в любое время.

Клиент устанавливает соединение, выполняя процесс так называемого **рукопожатия** WebSocket. Этот процесс начинается с того, что клиент отправляет серверу обычный HTTP-запрос. В этот запрос включается **заголовок Upgrade**, который сообщает серверу о том, что клиент желает установить WebSocket-соединение.

**НА КЛИЕНТЕ:**





**НА СЕРВЕРЕ:**



**Атрибуты:**

\*onopen – когда соед.устан.

\*onclose – соед.закрыто

\*onmessage – д-е получены

\*onerror – ошибка

**\***readystate – тек.сост.подключ

**\***url – url созд.к-ром

URL, применяемый для WebSocket-соединения, использует **схему ws**. Кроме того, имеется схема wss для организации защищённых WebSocket-соединений, что является эквивалентом HTTPS.

Если сервер поддерживает протокол WebSocket, он согласится перейти на него и сообщит об этом в заголовке ответа Upgrade.

Теперь, после завершения фазы рукопожатия, исходное HTTP-соединение заменяется на WebSocket-соединение, которое использует то же самое базовое TCP/IP-соединение. В этот момент и клиент и сервер могут приступать к отправке данных.

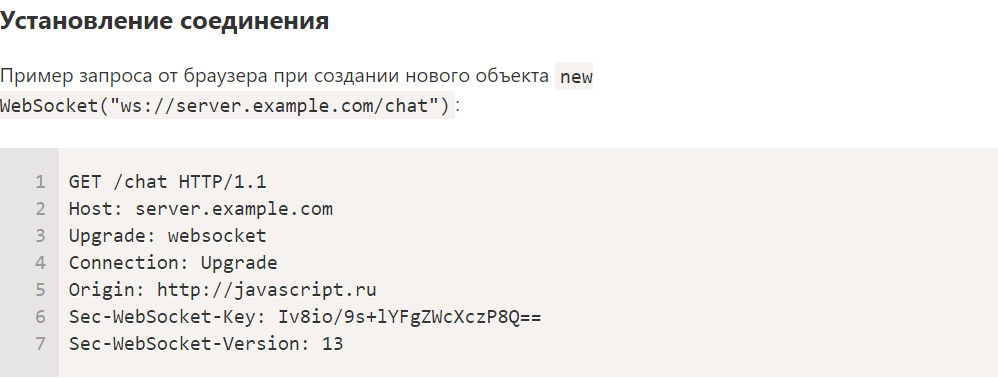
В любой момент после процедуры рукопожатия, либо клиент, либо сервер, может решить отправить другой стороне ping-сообщение. Получая такое сообщение, получатель должен отправить, как можно скорее, pong-сообщение. Это и есть heartbeat-сообщения. Их можно использовать для того, чтобы проверить, подключён ли ещё клиент к серверу.

**Методы:**

\*close – закрывает WS-подключение или заканчив.попытку подключ.

(если уже закрыто – ничего не делает)

\*send – передаем д-е через WS-соед



Описания заголовков:

**GET, Host -** Стандартные HTTP-заголовки из URL запроса

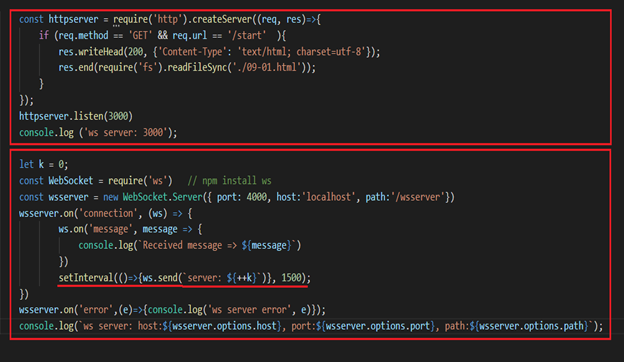
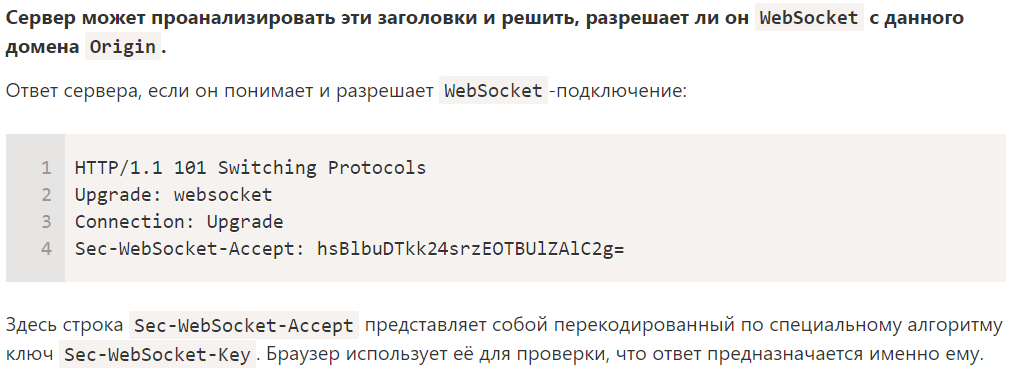
**Upgrade, Connection -** Указывают, что браузер хочет перейти на websocket.

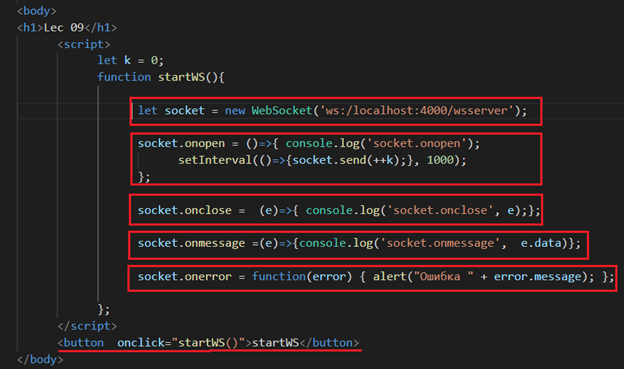
**Origin -** Протокол, домен и порт, откуда отправлен запрос.

**Sec-WebSocket-Key -** Случайный ключ, который генерируется браузером: 16 байт в кодировке [Base64](http://ru.wikipedia.org/wiki/Base64).

**Sec-WebSocket-Version -** Версия протокола. Текущая версия: 13.

Все заголовки, кроме GET и Host, браузер генерирует сам, без возможности вмешательства JavaScript.





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

Ajax относится к технологии разработки веб-страниц для создания интерактивных веб-приложений, и он может обновлять часть веб-страницы без перезагрузки всей веб-страницы, что также называется частичным обновлением.

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

Redis (расшифровывается как Remote Dictionary Server) – это быстрое хранилище данных типа «ключ‑значение» в памяти с открытым исходным кодом для использования в качестве базы данных, кэша, брокера сообщений или очереди.

Все данные в Redis хранятся в памяти, а не на дисках или твердотельных накопителях, как в других базах данных. Поскольку Redis, как и другие хранилища данных в памяти, не нуждается в доступе к диску, это исключает задержки, связанные с поиском, и обеспечивает доступ к данным за микросекунды. В число возможностей Redis входит поддержка разнообразных структур данных, обеспечение высокой доступности, работа с геопространственными данными, создание скриптов Lua, проведение транзакций, постоянное хранение данных на диске и поддержка кластеров. Все это упрощает создание приложений, работающих в режиме реального времени в масштабе всего Интернета.

**Redis** (**RE**mote **DI**ctionary **S**erver)**:** 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;

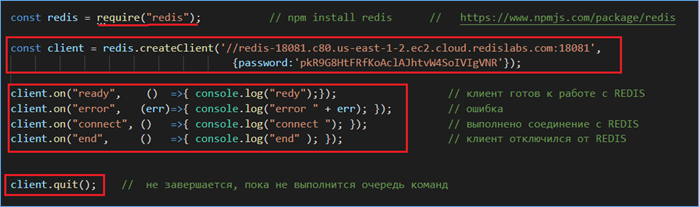
- СУБД ориентирована на быстрое выполнение атомарных операций (до 100тыс. set/get-операций);

- механизм снимков для асинхронного сохранения (с потерями);

- механизм упреждающей записи;

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:** npm redis,соединение, завершение, события



**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

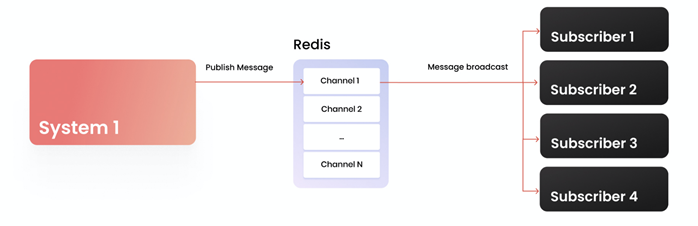
**HMSET ключ поле значение [поле значение …]** — Записывает значения в поля хэша.

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

указанного хэша.

EXISTS – уведомляет, если ключ существует.

**Redis:** канал (channel) – это абстракция, используемая для быстрого обмена данными между издателем и подписчиком.



**Redis:** publish, subscribe



# 28. Применение пакета Sequelize. Основные принципы работы. Пример (лабораторная работа).

**Sequelize** - npm-пакет, реализующий ORM-технологию.

**Object Relational Mapping** - технология программирования, которая позволяет работать с SQL-базой данных, как с набором программных объектов.

**ORM** – это методика или техника программирования, предназначенная для преобразования между несовместимыми типами данных в объектно-ориентированных языках программирования. Суть заключается в том, что создается некая абстракция - “виртуальная объектная база”, запросы к которой, преобразуются в SQL команды, т.е. больше не нужно писать SQL-запросы к базе данных вручную.

ORM фреймворк может быть написан на каком-либо объектно-ориентированном языке и представлять обертку над некой реляционной базой данных. Классы будут соответствовать таблицам в базе, а экземпляры этих классов – конкретным строкам таблицы.

Преимущества

· Независимость от вида базы данных

Это, пожалуй, главнейшая особенность и преимущество использования ORM в приложении. Так как нет необходимости писать специфический код под конкретный вид базы данных.

· Моделирование предметной области

При использовании ORM для построения приложения, бизнес-логика приложения работает с объектами языка, а не с самой структурой базы данных. Это возможно благодаря соответствию между бизнес-моделью и самой базой данных.

· Меньше кода и больше эффективности

ORM добавляет дополнительный слой абстракции, который позволяет разработчикам сконцентрироваться на их бизнес-логике, не отвлекаясь на создание сложных запросов к базе данных. Это приводит к сокращению количества необходимого для написания кода и увеличивает продуктивность разработчика.

· Управление зависимостями

ORM предоставляет свободное управление зависимостями в базе данных. Связанные объекты загружаются автоматически, когда вызов методов преобразуется в соответствующий SQL запрос.

· Параллелизм, кэширование и транзакции

ORM поддерживает возможность параллельной работы, позволяя нескольким пользователям одновременно изменять один и тот же объект.

Другая особенность – объекты могут быть сохранены в кэше, сокращая нагрузку на базу и в целом увеличивая скорость работы приложения.

Изменения, вносимые в объект, могут быть ограничены в рамках данной транзакции, которая может быть сохранена или возвращена обратно в первоначальное состояние. В каждый момент времени могут быть активными множество транзакций, но все эти транзакции изолированы друг от друга.

Недостатки

· Накладные расходы

ORM фреймворк добавляет слой абстракции, ускоряющий процесс разработки и снижающий сложность создания конечного продукта. Однако, ничто не проходит даром – приложение начинает потреблять больше памяти, и нагрузка на процессор также увеличивается.

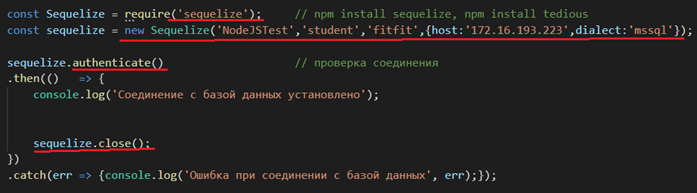
· Производительность

Некоторые операции, такие как массовая вставка, обновление, удаление и т.д., медленнее, когда выполняются посредством ORM. Поэтому, в таких случаях лучше использовать чистый SQL.

· Поверхностное знание SQL

Несмотря на то, что ORM облегчает жизнь, это часто приводит к тому, что разработчики не очень стремятся учить SQL или разбираются в нем слабо.

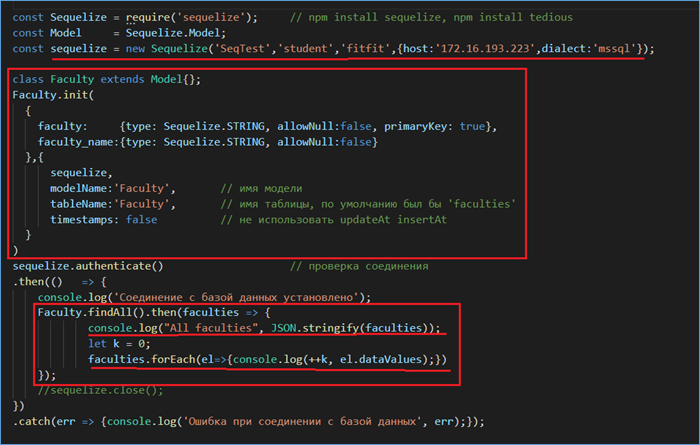
**Object Document Mapping** - технология программирования, которая сопоставляет объекты с документной базой данных(MongoDB).

**Подключение:** 

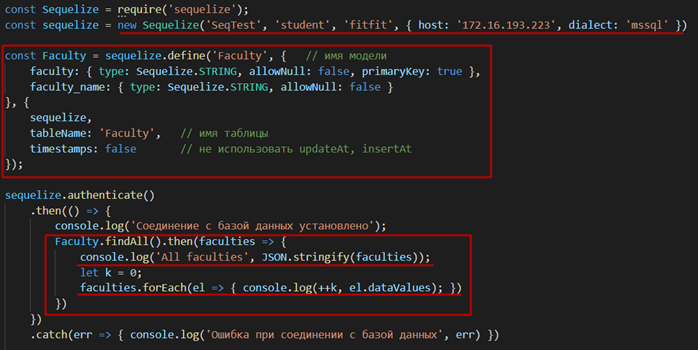
Tedious – протокол прикладного уровня, который использует mssql поверх TCP.

**Объявление моделей:**

1. **Model.init**



1. **sequelize.define**

****

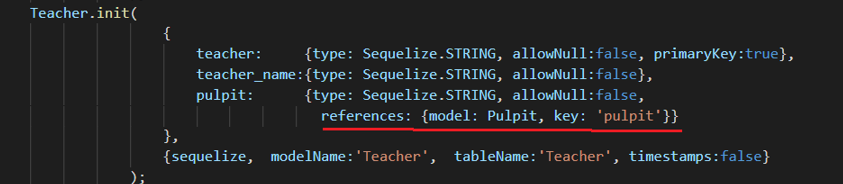
**Конфигурация:**

**sequelize** - экземпляр подключения.

**TableName** -имя таблицы**.**

**Timestamps** – false: отслеживает изменение в таблице(дополнительное поле будет создано или нет).

**references** - внешний ключ.

****

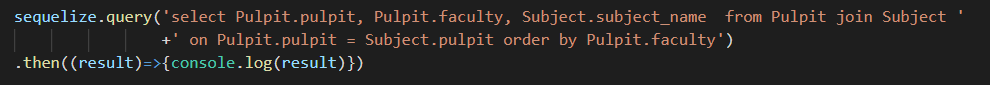
**User.sync() -** создает таблицу, если она не существует(ничего не делает, если она существует).

**User.sync({force: true}) -** создает таблицу, сначала отбрасывая ее, если она уже существует.

**User.sync({alter: true})** - проверяет текущее состояние таблицы в базе данных, а затем выполняет необходимые изменения в таблице, чтобы соответствовала модели.

**Code-First**  означает написание классов сущностей модели данных, которые мы будем использовать в нашем проекте, и позволить ORM сгенерировать базу данных.

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.

**Scopes** используются, чтобы повторно использовать код, определены в модели и могут быть объектами поиска или функциями.

**Хуки** (также известные как события жизненного цикла) - это функции, которые вызываются до и после выполнения вызовов в sequelize. Например, если вы хотите всегда устанавливать значение для модели перед ее сохранением, вы можете добавить beforeUpdate.

Триггеры представляют собой функции, которые выполняются (если они определены) до/после/во время действий с данными. Список самых популярных триггеров в порядке их выполнения:

beforeValidate(данные, опции) - выполняется перед валидацией;

afterValidate(данные, опции) или validationFailed(данные, опции, ошибка) - выполняется после успешной или неуспешной проверки валидации соответственно;

beforeCreate(данные, опции) - вызывается перед созданием записи;

beforeDestroy(данные, опции) - выполняется перед удалением записи;

beforeUpdate(данные, опции) - вызывается перед обновлением записи;

beforeSave(данные, опции) - вызывается перед сохранением записи;

afterCreate(данные, опции) - вызывается после создания записи;

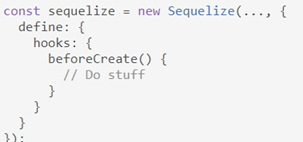
afterDestroy(данные, опции) - выполняется после удаления записи;

afterUpdate(данные, опции) - вызывается после обновления записи;

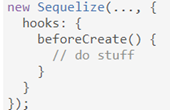
afterSave(данные, опции) - вызывается после сохранения записи.

2 вида хуков: локальные и глобальные. Сперва вызываются локальные, а потом глобальные.

**Глобальные хуки** - это хуки, которые запускаются для всех моделей. Они могут определять поведение, которое вы хотите для всех ваших моделей. Они могут быть определены двумя способами, которые имеют несколько различную семантику:

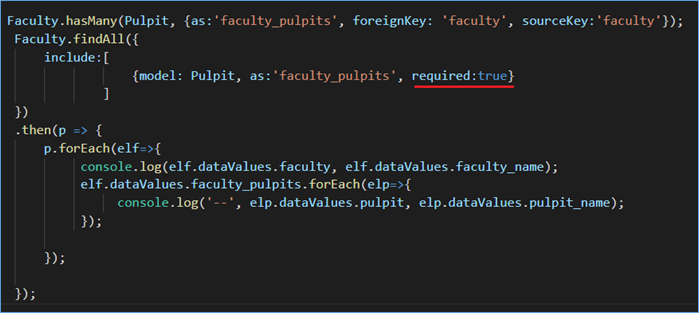


Постоянные хук также могут быть определены в опциях, переданных конструктору Sequelize

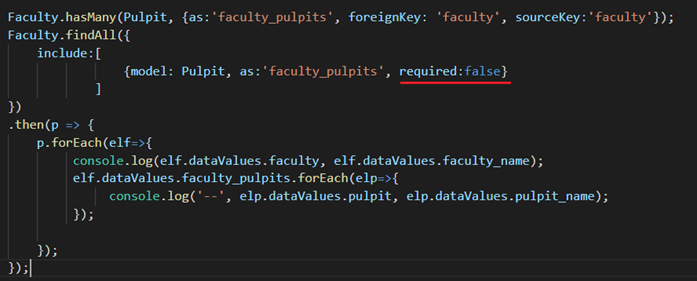


Обратите внимание, что вышеприведенное не совпадает *с крючками по умолчанию*, упомянутыми выше. Этот вариант использует define опцию конструктора. А вот этот-нет.

select … inner join



select … left outer join



Создание связей между таблицами осуществляется с использованием моделей. Рассмотрим установление следующих типов связей:

· один к одному;

· один ко многим;

· многие ко многим.

Эта A.hasOne(B) связь означает, что существует взаимно однозначное отношение между A и B, причем внешний ключ определяется в целевой модели (B).

Эта A.belongsTo(B) связь означает, что существует взаимно однозначное отношение между A и B, причем внешний ключ определяется в исходной модели (A).

A.hasMany(B) Ассоциация означает, что существует отношение "один ко многим" Между A и B, причем внешний ключ определяется в целевой модели (B).

Эти три вызова вызовут Sequelize для автоматического добавления внешних ключей к соответствующим моделям (если они уже не присутствуют).

A.belongsToMany(B, { through: 'C' }) Ассоциация означает, что существует отношение "многие ко многим" между A и B, используя таблицу C в качестве [соединительной таблицы](https://en.wikipedia.org/wiki/Associative_entity), которая будет иметь внешние ключи (aId bId например, и). Sequelize автоматически создаст эту модель C (если она уже не существует) и определит для нее соответствующие внешние ключи.

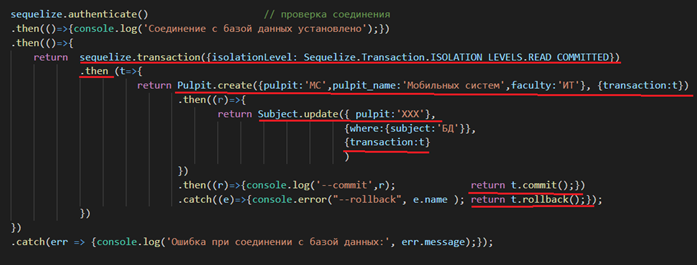
Sequelize по умолчанию не использует транзакции. Однако для готового к производству использования Sequelize вам определенно следует настроить Sequelize для использования транзакций.

Sequelize поддерживает два способа использования транзакций:

· Неуправляемые транзакции: фиксация и откат транзакции должны выполняться пользователем вручную (путем вызова соответствующих методов Sequelize).

· Управляемые транзакции: Sequelize автоматически откатит транзакцию, если возникнет какая-либо ошибка, или зафиксирует транзакцию в противном случае. Кроме того, если CLS (локальное хранилище продолжения) включен, все запросы в обратном вызове транзакции автоматически получат объект транзакции.

transaction… (unmanaged)



# 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.

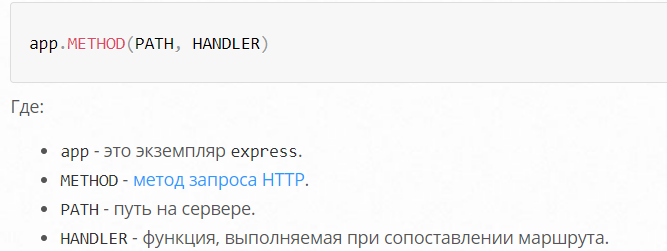
**Фре́ймворк** — *программная платформа*, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

При обработке запросов фреймворк Express опирается на систему маршрутизации. В приложении определяются маршруты, а также обработчики этих маршрутов. Если запрос соответствует определенному маршруту, то вызывается для обработки запроса соответствующий обработчик.

**Маршрутизация** определяет, как приложение отвечает на клиентский запрос к конкретному адресу (конечной точке), которым является URI (или путь), и определенному методу запроса HTTP (GET, POST и т.д.).

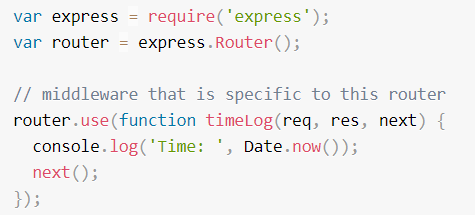
Каждый маршрут может иметь одну или несколько функций обработки, которые выполняются при сопоставлении маршрута.

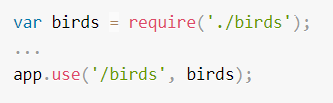
Определение маршрута имеет следующую структуру:



С помощью **класса express.Router** можно создавать **модульные, монтируемые обработчики маршрутов.** Экземпляр Router представляет собой комплексную систему промежуточных обработчиков и маршрутизации;

В приведенном ниже примере создается маршрутизатор в виде модуля, в него загружается функция промежуточной обработки, определяется несколько маршрутов, и модуль маршрутизатора монтируется в путь в основном приложении.





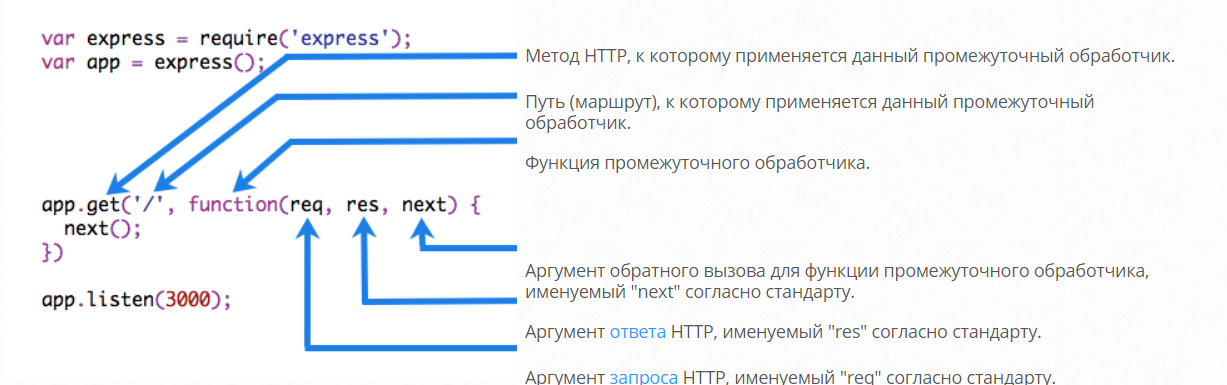
Функции промежуточной обработки (**middleware**) - это функции, имеющие доступ к [объекту запроса](https://expressjs.com/ru/4x/api.html" \l "req) (req), [объекту ответа](https://expressjs.com/ru/4x/api.html#res) (res) и к следующей функции промежуточной обработки в цикле “запрос-ответ” приложения. Следующая функция промежуточной обработки, как правило, обозначается переменной **next**.

Функции промежуточной обработки могут выполнять следующие задачи:

* Выполнение любого кода.
* Внесение изменений в объекты запросов и ответов.
* Завершение цикла “запрос-ответ”.
* Вызов следующего промежуточного 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.

Ниже представлены элементы вызова функции промежуточного обработчика:



Для того чтобы загрузить функцию промежуточного обработчика вызовите **app.use()** с указанием соответствующей функции.

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](https://expressjs.com/ru/guide/using-middleware.html#middleware.application)
* [Промежуточный обработчик уровня маршрутизатора](https://expressjs.com/ru/guide/using-middleware.html#middleware.router)
* [Промежуточный обработчик для обработки ошибок](https://expressjs.com/ru/guide/using-middleware.html#middleware.error-handling)
* [Встроенные промежуточные обработчики](https://expressjs.com/ru/guide/using-middleware.html#middleware.built-in)
* [Промежуточные обработчики сторонних поставщиков ПО](https://expressjs.com/ru/guide/using-middleware.html#middleware.third-party)

Промежуточный обработчик уровня приложения с экземпляром [объекта приложения](https://expressjs.com/ru/4x/api.html" \l "app), воспользовавшись функциями app.use() и app.METHOD(), где METHOD - метод HTTP запроса, обрабатываемый функцией промежуточной обработки (например, GET, PUT или POST) в нижнем регистре

Обработчики маршрутов позволяют определить несколько маршрутов для одного пути. В приведенном ниже примере определено два маршрута для запросов GET, адресованных ресурсам в пути /user/:id. Второй маршрут не создает никаких неудобств, но его вызов никогда не будет выполнен, поскольку первый маршрут завершает цикл “запрос-ответ”.

**Обработка ошибок**

Функции промежуточного обработчика для обработки ошибок определяются так же, как и другие функции промежуточной обработки, но с указанием для функции обработки ошибок не трех, а четырех аргументов: (**err**, req, res, next).



Промежуточный обработчик для обработки ошибок должен быть определен последним, после указания всех app.use() и вызовов маршрутов;

В целях упорядочения (и для фреймворков более высокого уровня) можно определить несколько функций промежуточной обработки ошибок, точно так же, как это допускается для обычных функций промежуточной обработки.

**Middleware**-функции – функции промеж.обработки – ф-ции, кот.имеют доступ к req и res, и к след.ф-ции обработки в цикле «запрос-ответ».

Функция, которая передается в app.use(), принимает три параметра:

* request: данные запроса
* response: объект для управления ответом
* next: следующая в конвейере обработки функция

# 30. Пакет Express. Основные принципы работы. Маршрутизация. Пример.

ВЫШЕ

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

**Маршрутизатор** – по uri решаем какой контроллер будет обрабатывать запрос.

Когда приходит запрос Express сопоставляет запрошенный адрес с каждым из маршрутов. Затем выбирается первый совпавший маршрут. При совпадении маршрута вызывается его функция обработчика.

# 31. Пакет Express. Основные принципы работы. Статические файлы. Пример.

**Статические файлы:** файлы, расположенные на стороне сервера и предназначенные для считывания их без изменения с помощью HTTP GET-запроса по имени ресурса, включающего имя файла

**Статические файлы:** <link>, <script>, <img>, <audio>, <video>, <a>, <form>, <frame> (не поддерживается в HTML5)

, public – название папки (путь)

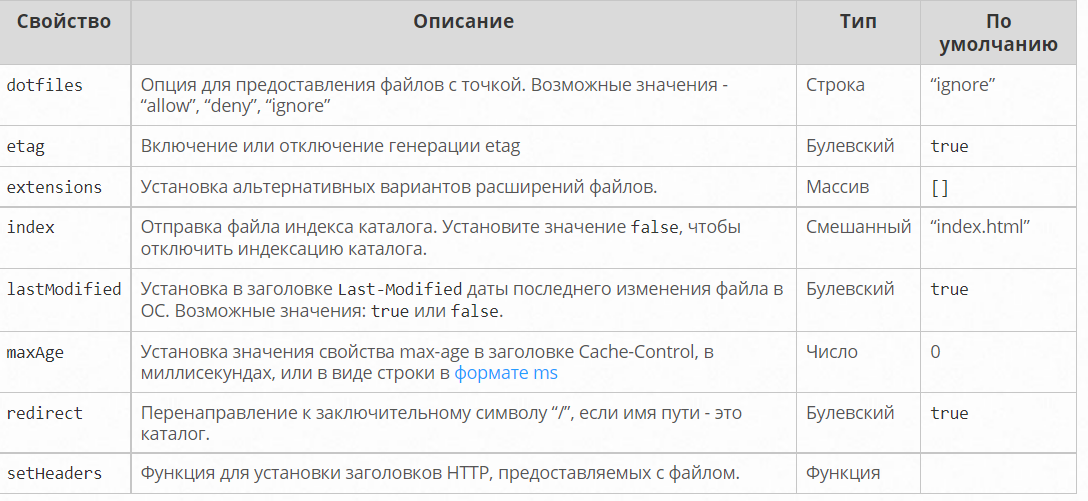


**express.static(root, [options])**

Единственной **встроенной** функцией промежуточной обработки в Express является **express.static**. Эта функция основана на **[serve-static](https://github.com/expressjs/serve-static)** и отвечает за предоставление статических ресурсов приложения Express.

Аргумент root указывает на корневой каталог, из которого предоставляются статические ресурсы.

Необязательный объект options может содержать следующие свойства:



**ЗАГОЛОВКИ**

Cache-control это HTTP-заголовок, который определяет количество времени и способ htaccess кэширования файла, используется для задания инструкций кэширования как для запросов, так и для ответов.

ETag (или entity tag) — один из механизмов кэширования в HTTP. По сути это идентификатор, который присваивается файлу сервером для последующей проверки, Иначе требуемые файлы будут заново отправлены.

Last-Modified в ответе HTTP содержит дату и время, в которую, по мнению удаленного сервера, запрашиваемый ресурс был изменен.

If-Modified-Since делает запрос условным: сервер отправит обратно запрошенный ресурс с статусом 200, только если он был изменен после указанной даты. Если запрос не был изменен после указанной даты, ответ будет 304 без какого-либо тела; заголовок Last-Modified при этом будет содержать дату последней модификации. If-None-Match Для методов GET и HEAD сервер отправляет запрошенный ресурс только в том случае, если он соответствует одному из перечисленных ETags. Для PUT и других небезопасных методов он будет загружать только ресурс в этом случае.

Content-Disposition = Attachment - является индикатором того, что ожидаемый контент ответа будет отображаться в браузере, как веб-страница или часть веб-страницы, или же как вложение, которое затем может быть скачано и сохранено локально.

**Attachment –** указание http-клиенту не отображать содержимое body, а сохранить в файловой структуре.

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

**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.

Node.js Express устанавливается через пакетный менеджер [npm](https://nodejsdev.ru/doc/npm/).

Node.js Express имеет готовые функции обработки HTTP запросов, причем для каждого HTTP метода имеется своя функция, что особенно удобно при создании [REST API](https://nodejsdev.ru/doc/rest-api/). И это далеко не единственная причина использования Express.

Метод **use()** используется для создания промежуточных обработчиков - [Middleware](https://nodejsdev.ru/doc/middleware/).

**Фре́ймворк** — *программная платформа*, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Объект **Request** содержит информацию о запросе. Объект представляет собой HTTP - запрос и имеет свойства для строки запроса, параметров, тела, HTTP-заголовков и т.д.

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| .app | содержит ссылку на объект приложения |
| .baseUrl | содержит ссылку на экземпляр маршрутизатора (express.Router()) |
| .body | содержит данные, помещенные в тело запроса (должны быть разобраны (parsed) и заполнены (populated) перед использованием) |
| .cookies | содержит куки, установленные в запросе (требуется промежуточное программное обеспечение (далее — ППО) cookie-parser) |
| .hostname | название хоста сервера |
| .ip | IP-адрес сервера |
| .method | метод запроса |
| .params | объект с именованными параметрами запроса (например, при запросе к /users/:id, id будет записано в req.params.id) |
| .path | URL запроса |
| .protocol | протокол запроса |
| .query | объект с параметрами строки запроса (например, при запросе к /search?name=john, john будет записано в req.query.name) |
| .secure | содержит true, если запрос является безопасным (если используется HTTPS) |
| .signedCookies | содержит подписанные куки (требуется ППО cookie-parser) |
| .xhr | содержит true, если запрос — это XMLHttpRequest |

Объект **Response** представляет собой HTTP-ответ. Методы объекта response (res) в следующей таблице могут отправить ответ клиенту и завершить цикл ответа запроса. Если ни один из них не будет вызван из обработчика маршрута, клиентский запрос останется висеть.

|  |  |
| --- | --- |
| **Method** | **Description** |
| res.download() | Запросите файл для загрузки. |
| res.end() | Завершите процесс реагирования. |
| res.json() | Отправьте ответ JSON. |
| res.redirect() | Перенаправление запроса. |
| res.render() | Отрисовка шаблона представления. |
| res.send() | Отправьте ответ различного типа. |
| res.sendFile | Отправьте файл в виде потока октетов. |
| res.sendStatus() | Установите код состояния ответа и отправьте его строковое представление в качестве тела ответа. |

1. Получение входных параметров от клиента с помощью маршрута

При создании API вы должны позволить пользователям сообщать вам, что они хотят получить от вашего сервиса. Один из возможных способов достижения этого результата - использование параметров маршрута. **Параметры маршрута** называются **сегментами URL-адреса,** разделенными косыми чертами (/). Каждый сегмент фиксирует значение той части URL-адреса, которая соответствует его положению. Полученное значение можно найти в объекте **req.params**.

2. Получение входных параметров от клиента с помощью строки запроса

Другой распространенный способ получения входных данных от клиента - это кодирование данных после пути маршрута с помощью **строки запроса (query).** Строка запроса отделяется от маршрута вопросительным знаком (?), и включает параметры в виде имя\_параметра=значение. Каждый параметр разделен амперсандом (&). Express может анализировать данные из строки запроса и сохранять эти данные в объект **req.query.** Поэтому, если необходимо узнать значение какого-нибудь параметра, нужно обратиться к объекту **req.query**, добавив к нему через точку имя\_параметра. Например, если нужно получить значение параметра user из вышеприведенного запроса, то необходимо обратиться к req.query.user. Если же обратиться к req.query, то можно получить все параметры строки запроса в виде объекта.

# 33. Пакет Express. Основные принципы работы. Обработка uri-параметров запроса. Пример (POSTMAN).

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

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

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

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

В обычном HTTP-ответе заголовок Content-Disposition является индикатором того, что ожидаемый контент ответа будет отображаться в браузере, как веб-страница или часть веб-страницы, или же как вложение, которое затем может быть скачано и сохранено локально.

**Attachment** – указание http-клиенту не отображать содержимое body, а сохранить в файловой структуре.

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

# 39. Пакет Express. Основные принципы работы. Обработка Cookie. Signed cookie. Пример(POSTMAN).

Cookie: фрагмент данных, хранится на http-клиент, создается по инициативе сервера (заголовок Set-Cookie), пересылается http-клиентом (заголовок Cookie), http-клиент может отказаться от создания cookie, http-клиент может удалить cookie, содержимое cookie доступно, с помощью JS можно изменить cookie, один из методов XSS-атаки (cross-site scripting) основана на подмене cookie, применение cookie надо избегать.

опция domain – привязка cookie к поддомену.

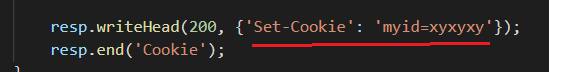
опция path – путь, на который распространяется действие cookie.

опция maxAge – время жизни cookie в миллисекундах.

опция Expires – дата истечения жизни cookie.

опция secure – может применяться только с HTTPS.

опция httpOnly = true – может изменяться сервером.





**signed cookie:**



Файл cookie по-прежнему будет виден, но у него есть подпись, поэтому он может определить, изменил ли клиент файл cookie.

Он работает, создавая HMAC значения (текущий файл cookie) и base64 его кодирования. Когда файл cookie считывается, он пересчитывает подпись и проверяет, соответствует ли она подписи, прикрепленной к нему. Если он не совпадает, то это выдаст ошибку.

res.cookie('name', 'value', {signed: true})

req.signedCookies['name']

Пример обычных:

app.use(express.cookieParser());

var cookie = req.cookies.cookieName;

res.cookie('cookieName',randomNumber, { maxAge: 900000, httpOnly: true });

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

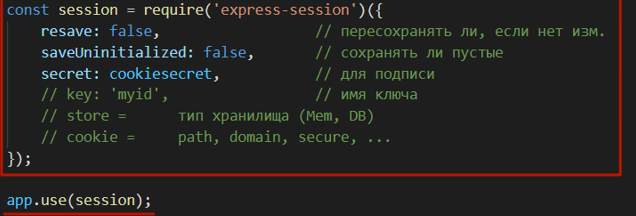
**Session:** серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении,

время жизни: timeout (системный параметр, обычно равен 10 – 30 минутам) – максимальное время между запросами клиента. Если timeout превышен, то Session разрушается и при следующем запросе создается новый экземпляр.

Каждая сессия имеет собственный идентификатор (Session ID, 16 или более байт).

Каждый Request принадлежит, какой-то сессии (имеет ссылку на объект Session или содержит Session ID).

Обычно объект Session предоставляет приложению возможность хранить данные в формате ключ/значение.



Во время инициализации Node.js сессии с помощью объекта можно задать следующие опции:

**cookie** - настройка cookie хранения идентификатора сессии, передается объект с опциями

**genid** - функция, которая возвращает новый идентификатор сессии в виде строки (по умолчанию используется функция, генерирующая идентификаторы на основе библиотеки uid-safe);

**resave** - булевое значение, указывает, нужно ли пересохранять сессию в хранилище, если она не изменилась (по умолчанию false);

**rolling** - булевое значение, указывающее, нужно ли устанавливать идентификатор сессии cookie на каждый запрос (по умолчанию false);

**saveUninitialized** - булевое значение, если true, то в хранилище будут попадать пустые сессии;

**secret** - строка, которой подписывается сохраняемый в cookie идентификатор сессии;

**store** - экземпляр хранилища, которое будет использоваться для хранения сессии (рассмотрено ниже в этой статье).

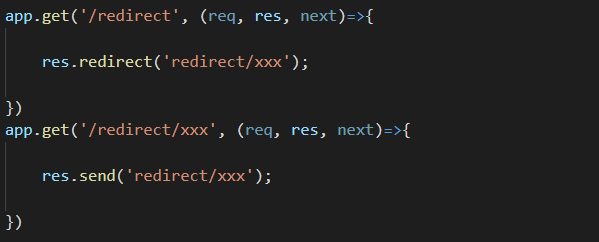
# 41. Пакет Express. Основные принципы работы. Переадресация. Пример(POSTMAN).

**Redirect –** переадресация (3хх)

308 – такой же запрос, как был первичный

303-305 – get Запросы

Если мы хотим, чтобы К переслал тот же запрос после redirect => указ.код ответа **308**

****

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.

Переадресация по абсолютному пути:

|  |
| --- |
| const express = require("express");  const app = express();    app.use("/index",function (request, response) {    response.redirect("[https://metanit.com](https://metanit.com/)")  });    app.listen(3000); |

В данном случае при обращении по пути "/index" будет идти переадресация на сайт https://metanit.com.

Переадресация по оносительным путям также не очень сложна, но здесь важно учитывать, как именно определяется адрес для редиректа. Рассмотрим редирект относительно текущего пути, с которого производится редирект. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | const express = require("express");  const app = express();  app.use("/home",function (request, response) {  response.redirect("about")  });  app.use("/about", function (request, response) {  response.send("<h1>About</h1>");  });  app.listen(3000); |

В данном случае будет идти перенаправление с ресурса "/home" на ресурс "/about", то есть, условно говоря, с http://localhost:3000/home на http://localhost:3000/about.

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

Spawn – ф-ция, кот позвол запускть консоль, с пом кот м.запускать любые прилож.

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

//cmd.exe - указ.что будем запускать, дальше идет массив пар-ров оболочки

//u - вывод в юникод

//с - необх.выполнить и завершить работу консоли

//dir - сама команда - показать директорий

const dir = spawn('cmd.exe', [ '/U', '/C', 'dir']);

//findstr - отыскивает строки во вх.потоке, кот соотв.нек.шаблону

// /с - ключ, все что сод. 43 - все такие строки надо показать

const findstr = spawn('findstr', ['/c:43']);

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

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

**Параметры**

Ниже приводится описание используемых параметров:

command (String) — запускаемая команда

args (Array) — список аргументов строки

options (Object) — может содержать один или несколько из следующих параметров:

Метод spawn() возвращает потоки (stdout & stderr), он должен использоваться, когда процесс возвращает большой объем данных. spawn() начинает принимать ответ, сразу после начала выполнения процесса.

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

exec — метод child\_process.exec запускает команду в оболочке/консоли и буферизует вывод.  
child\_process.exec(command[, options], callback)

\* command (String) — команда для запуска с аргументами, разделенными пробелом  
\* options (Object) — может содержать один или несколько параметров  
\* callback — функция принимает три аргумента error , stdout и stderr, которые вызываются с выходными данными при завершении процесса.

—cwd (String) — текущий рабочий каталог дочернего процесса

—env (Object) — ключи-значения для среды

—encoding (String) — (по умолчанию: ‘utf8′)

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

Метод child\_process.fork является особым случаем spawn() для создания процессов Node  
  
child\_process.fork(modulePath[, args][, options])  
  
\* modulePath (String) — Модуль для запуска в дочернем элементе.  
  
\* args (Array) — Список аргументов строки  
  
В дополнение к тому, что он содержит все методы, что и стандартный экземпляр дочернего процесса, метод fork возвращает объект со встроенным каналом связи.

Send() - Возвращает Boolean.

Если Node.js был порожден с IPC каналом, можно использовать метод process.send() для отправки сообщений родительскому процессу. Сообщения будут получаться событием ‘message’ в объекте ChildProcess родительского процесса.

Потоки воркеров выполняются в изолированном контексте. Они обмениваются информацией с главным процессом, используя сообщения. Это избавляет нас от проблемы «состояния гонок», которой подвержены многопоточные среды. При этом потоки воркеров существуют в том же процессе, в котором находится основная программа, то есть, при таком подходе, в сравнении с применением форков процессов, используется гораздо меньше памяти.

==============================

\* Синтаксис fork() почти аналогичен синтаксису метода spawn(), за одним исключением: здесь не нужна команда, потому что fork() предполагает, что все процессы относятся к Node.js  
  
\* exec(). Этот метод отличается от предыдущих тем, что использует не шаблон событий, а одиночный callback.

**Разница между этими тремя способами заключается в следующем:**  
require('child\_process').spawn(): используется для больших объёмов данных; поддерживает stream’ы; может применяться с любыми командами; не создаёт новый экземпляр V8.  
require('child\_process').fork(): создаёт новый экземпляр V8 и экземпляры работников; работает только со скриптами Node.js (команда node).  
require('child\_process').exec(): неудобен для больших объёмов данных, потому что использует буфер; работает асинхронно, чтобы одновременно получать все данные в callback’е; может применяться с любой командой, не только node.

# 45. Протокол WebDav. Разработка приложения с применением WebDav. Пример(лабораторная работа).

1. **WebDAV**: Web Distributed Authoring and Versioning – 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). Описание в RFC 2518 (=>RFC 4918).

WebDAV (Web Distributed Authoring and Versioning) — это протокол для передачи данных и работы с ними, построенный поверх HTTP 1.1. Здесь следует заметить, что передача может быть как защищенной, так и незащищенной.

Таким образом, WebDAV позволяет изменять свойства хранящихся на сервере объектов, выполнять поиск с учетом свойств, блокировать объект (в нашем случае — файл) для организации возможности его редактирования только одним пользователем в распределенной среде, в которой доступ могут иметь много юзеров, управлять версиями файлов (посредством унаследованных команд check -in, -out), а также производить расширенный контроль доступа к файлам на основе списков.

**WebDAV**: **унаследованные HTTP-методы**

**WebDAV**: **GET** – скачать файл.

**WebDAV**: **PUT** – загрузить файл на сервер.

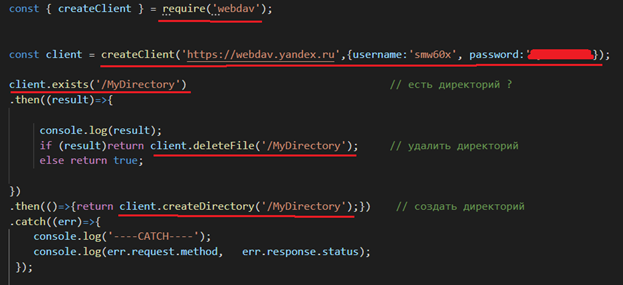
**WebDAV**: **DELETE** – удалить серверный объект.

HTTP Request Methods

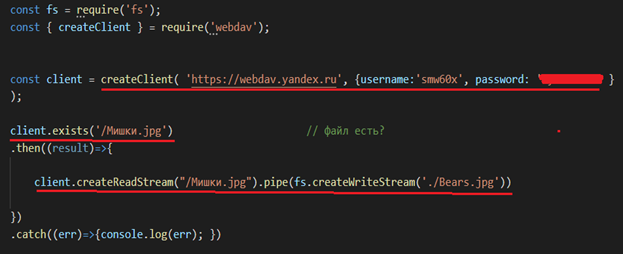
Википедия: **WebDAV** расширяет HTTP следующими методами запроса:

* PROPFIND — получение свойств объекта на сервере в формате [XML](https://ru.wikipedia.org/wiki/XML). Также можно получать структуру [репозитория](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BF%D0%BE%D0%B7%D0%B8%D1%82%D0%BE%D1%80%D0%B8%D0%B9) (дерево каталогов);
* PROPPATCH — изменение свойств за одну транзакцию;
* MKCOL — создать коллекцию объектов (каталог в случае доступа к файлам);
* COPY — копирование из одного [URI](https://ru.wikipedia.org/wiki/URI) в другой;
* MOVE — перемещение из одного [URI](https://ru.wikipedia.org/wiki/URI) в другой;
* LOCK — поставить блокировку на объекте. WebDAV поддерживает эксклюзивные и общие (shared) блокировки;
* UNLOCK — снять блокировку с ресурса.

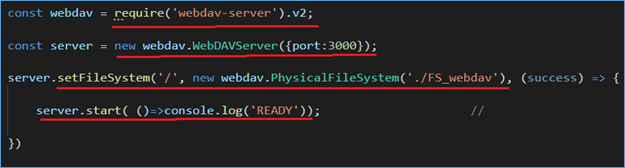
**WebDav:** client, create Directory, delete File



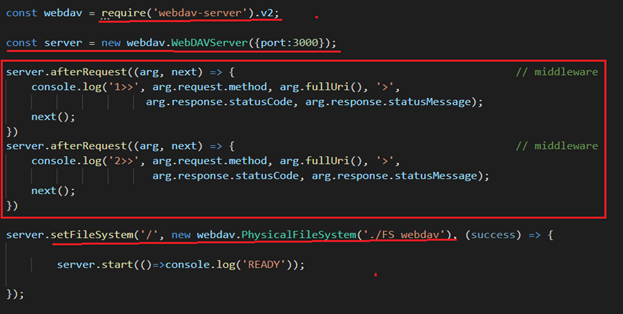
**WebDav:** download file



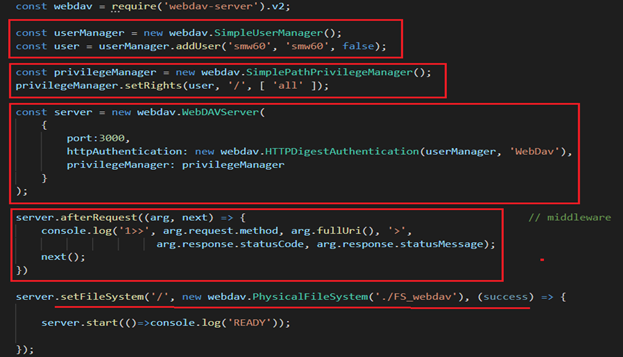
**WebDav:** сервер, File Systemmapping



**WebDav:** сервер,middleware



**WebDav:** сервер, аутентификация, авторизация



**WebDav:** клиент, аутентификация



# 46. Протокол JSON-RPC. Разработка клиент-серверное приложение использующее протокол JSON-RPC.

**JSON-RPC -** протокол удаленного вызова процедур, использующий формат JSON для передачи сообщений. Это очень простой протокол (очень похожий на XML-RPC), определяющий только несколько типов данных и команд. JSON-RPC поддерживает уведомления (информация, отправляемая на сервер, не требует ответа) и множественные вызовы.

JSON-RPC работает, посылая запросы к серверу, реализующему протокол. Клиентом обычно является программа, которой нужно вызвать метод на удалённой системе. Множество входных параметров может быть передано удалённому методу как массив или запись. Метод также может вернуть множество выходных данных (это зависит от реализации). Удалённый метод вызывается отправлением запроса на удалённый сервер посредством [HTTP](https://ru.wikipedia.org/wiki/HTTP) или [TCP/IP](https://ru.wikipedia.org/wiki/TCP/IP)-[сокета](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)) (начиная с версии 2.0). При использовании HTTP [заголовок](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BA%D0%B8_HTTP) Content-Type определяется как application/json.

Все передаваемые данные — простые [записи](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BF%D0%B8%D1%81%D1%8C_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)), [сериализованные](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) в JSON. Запрос — вызов определённого метода, предоставляемого удалённой системой. Он должен содержать три обязательных свойства:

· method — строка с именем вызываемого метода.

· params — массив данных, которые должны быть переданы методу, как параметры.

· id — значение любого типа, которое используется для установки соответствия между запросом и ответом.

Сервер должен отослать правильный ответ на каждый полученный запрос. Ответ должен содержать следующие свойства:

· result — данные, которые вернул метод. Если произошла ошибка во время выполнения метода, это свойство должно быть установлено в null.

· error — код ошибки, если произошла ошибка во время выполнения метода, иначе null.

· id — то же значение, что и в запросе, к которому относится данный ответ.

Для ситуаций, когда ответ не требуется, были введены уведомления. **Уведомление** отличается от запроса отсутствием свойства id, которое не требуется, так как не будет передан ответ. В таком случае свойство id может быть пропущено (версия 2.0) или установлено в null (версия 1.0).

**Что такое пакет RPC-запросов?**

Чтобы отправить несколько объектов Запроса одновременно, Клиент МОЖЕТ отправить массив, заполненный объектами запроса.

Сервер должен ответить Массивом, содержащим соответствующие объекты ответа, после обработки всех объектов пакетного запроса. Объект ответа ДОЛЖЕН существовать для каждого объекта запроса, за исключением того, что НЕ ДОЛЖНО быть никаких объектов ответа для уведомлений. Сервер МОЖЕТ обрабатывать пакетный вызов rpc как набор параллельных задач, обрабатывая их в любом порядке и с любой шириной параллелизма.

Объекты ответа, возвращаемые из пакетного вызова, МОГУТ быть возвращены в любом порядке внутри массива. Клиент ДОЛЖЕН сопоставлять контексты между набором объектов запроса и результирующим набором объектов ответа на основе элемента id внутри каждого объекта.

Если сам пакетный вызов rpc не распознается как допустимый JSON или массив хотя бы с одним значением, ответ Сервера ДОЛЖЕН быть одним объектом ответа. Если в массиве ответов нет объектов ответа, которые должны быть отправлены клиенту, сервер НЕ ДОЛЖЕН возвращать пустой массив и вообще ничего не должен возвращать.

**Позиционные параметры запроса.**



Передается массив, параметры можно доставать по номеру

**Именованные параметры запроса.**

Каждому параметру присваивается свое имя, по которому можно к нему обратиться

На стороне клиента single-page приложение. сначала загруж какая-то станица, кот. насыщена js-ом и она предст. прилож работающее на ajax-запросы, получ ответы, их обраб, обнов эту страницу, и перезагрузка осущ. лишь единожды – когда стартует прилож.

На стороне С распол веб-сервис – веб-прил, кот. предст интерфейс не конеч. юзера, а интерфейс для других программ. Ее юзеры – другие программы, а не конечные юзеры.

**Есть 2 типа интерфейса таких:**  
\* *REST* (предст программисту в виде наборов uri и правил их исп-я: какие компоненты, из чего сост. эи uri, какие методы мб вызваны (get,post,put,delete)) – до этого мы разрабатывали почти только их

\* *RPC* (предст программисту как набор процедур нах на стороне С, и какой-то способ их вызова – эти проц вызыв, перед в параметры, отрабатывают и возвр рез)

**RPC сервер –** набор процедур, кот.нах на стороне С и есть какой-то способ их вызова, т.е.какой-то интерфейс – эти процедуры вызываются, им передаются пар-ры, они отрабатывают, возвращ.рез-т.

**Где исп-ть json rpc?**

\* efirium – для работы с узлами сети

\* система забикс – мониторинг жд ИС

***xml-rpc*** – его прорадитель, был оч популярен и прост, но перерос в протокол, кот уже стандартизирован – ***soap*** (потом о нем поговорим)  
втоая ветка от него отошла – ***json-rpc*** (основан на формате д-х json, в кач транспорта исп https/https/tcp)

сам json стандартизирован (rfc-4627)

JSON RPC описан на сайте [www.jsonrpc.org](http://www.jsonrpc.org) и нигде не описан в стандартах RFC (не зареган)

**JSON RPC** - протокол удаленного вызова процедур, который использует формат JSON для передачи сообщений.

+ простой

+ понятный

\* порожден от XML RPC

\* основан на формате д-х JSON

\* транспорт – HTTP / HTTPS / TCP Socket

*JSON RPC:*

Предполаг., что есть С, к кот мы м отправл rpc-запросы и получить rpc-ответы, при этом запросы и ответы имеют формат json. То как будет транспортироваться json – данная спецификация не рассматривает. Спецификация только говорит: есть такой формат, давайте его использовать для того, чтобы взаимодействовать К и С.

***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)

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

* Version – версия протокола
* Result – объект, понятный клиенту
* Error – взаимозамен.с ресалт, сод.сообщ об ошибке
* Id – ссылается на запрос, на кот отвеч ошибкой

***Структура error:***

* Code – номер, кот идентиф.ошибку
* Message – сообщ, кот сопровожд.код
* Data – д-е в формате json, кот позвол более точно идентиф.ошибку

**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}

**Уведомление –** запрос без ответа (не указ id в request)

**RPC с пакетом запросов** – пакет запросов, т.е.это массив запросов, к эл-т кот.- отдельный запрос в формате json.

Ответ – такая же пачка.

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

**WASM (Web Assembly)** – бин.формат исполняемого файла, кот.мб исполняться в вирт.стек.машине (JS Engine);

* Код быстрее, чем JS
* Поддерживается большинством браузеров
* Выполняется в sandbox
* Есть отладчики
* Открытый стандарт

**WASM –** \* не ЯП, а байт-код

\* загружается в браузер и исполняется в браузере

(формально исполняется JS-движком, а не самим браузером)

\* исполняется виртуальной машиной

\* не имеет ничего общего с web, кроме того что общается с внешним миром через JS

**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

**Для чего?**

* Мы отсылаем предкомпилированный код => та часть, кот.выполнялась бы JS engine уже выполнена => экономить можно
* Компилируем не на лету, а в оффлайне (JS налету) => мы не связаны с временем исполнения
* Выигрыш за счет того, что убрали на стороне браузера компиляцию, а также за счет оптимизации (из-за того, что отделили стадию компиляции и стадию исполнения – можем не экономить на времени компиляции и оптимизировать с помощью оптимизаторов, т.е. строить более оптимальный код)

**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.

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.

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

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

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

# 49. Long pool–сервер, принцип работы. Пример (Telegram bot, лабораторная работа).

**Боты** - это сторонние приложения, которые работают внутри Telegram. Пользователи могут взаимодействовать с ботами, отправляя им сообщения, команды и [встроенные запросы](https://core.telegram.org/bots#inline-mode).

### **Как работают боты?**

По сути, Telegram-боты-это специальные учетные записи, для настройки которых не требуется дополнительный номер телефона. Пользователи могут взаимодействовать с ботами двумя способами:

* Отправляйте сообщения и [команды](https://core.telegram.org/bots#commands) ботам, открывая с ними чат или добавляя их в группы.
* Отправляйте запросы непосредственно из поля ввода, набрав @username бота и запрос. Это позволяет отправлять контент от [встроенных ботов](https://core.telegram.org/bots/inline) непосредственно в любой чат, группу или канал.

**BotFather**-это единственный бот, который управляет ими всеми.

Создание нового бота

Используйте команду /newbot для создания нового бота. Бот-отец запросит у вас имя и имя пользователя, а затем сгенерирует токен авторизации для вашего нового бота.

**Токен**-это строка 110201543:AAHdqTcvCH1vGWJxfSeofSAs0K5PALDsaw, которая необходима для авторизации бота и отправки запросов в API бота.

Генерация маркера авторизации

Если ваш существующий токен скомпрометирован или вы потеряли его по какой-то причине, используйте команду /token для создания нового.

Команды Botfather

Остальные команды довольно понятны:

/mybots — возвращает список ваших ботов с удобными элементами управления для редактирования их настроек

/mygames — делает то же самое для ваших игр

Редактировать ботов

/setname – измените имя вашего бота.

/setdescription — изменение описания бота , краткого текста до 512 символов, описывающего вашего бота. Пользователи увидят этот текст в начале разговора с ботом под названием " Что может сделать этот бот?".

/setabouttext — изменение информации бота, еще более короткий текст до 120 символов. Пользователи увидят этот текст на странице профиля бота. Когда они делятся своим ботом с кем-то, этот текст отправляется вместе со ссылкой.

/setuserpic — изменение фотографий профиля бота. Всегда приятно подставлять лицо под имя.

/setcommands — изменение списка команд, поддерживаемых вашим ботом. Пользователи будут видеть эти команды как предложения, когда они набирают /текст в чате с вашим ботом. Каждая команда имеет имя (должно начинаться с косой черты ‘/’, буквенно-цифровой плюс подчеркивание, не более 32 символов, без учета регистра), параметры и текстовое описание. Пользователи будут видеть список команд всякий раз, когда они наберут " / " в разговоре с вашим ботом.

/deletebot — удалите своего бота и освободите его имя пользователя.

Изменить настройки

/setinline — переключение встроенного режима для вашего бота.

/setinlinegeo - запрос данных о местоположении для предоставления встроенных результатов на основеместоположения .

/setjoingroups — переключает, можно ли добавить вашего бота в группы или нет. Любой бот должен уметь обрабатывать личные сообщения, но если ваш бот не был предназначен для работы в группах, вы можете отключить эту функцию.

/setprivacy — установите, какие сообщения будет получать ваш бот при добавлении в группу. При отключенном режиме конфиденциальности бот будет получать все сообщения. Мы рекомендуем оставить режим конфиденциальности включенным. Вам нужно будет повторно добавить бота в существующие группы, чтобы это изменение вступило в силу.

2 общих подхода для создания бота (web hook; long pool запрос)

**Длинные опросы** – это самый простой способ поддерживать постоянное соединение с сервером, не используя при этом никаких специфических протоколов (типа WebSocket или Server Sent Events).

**Частые опросы**

Самый простой способ получать новую информацию от сервера – периодический опрос. То есть, регулярные запросы на сервер вида: «Привет, я здесь, у вас есть какая-нибудь информация для меня?». Например, раз в 10 секунд.

В ответ сервер, во-первых, помечает у себя, что клиент онлайн, а во-вторых посылает весь пакет сообщений, накопившихся к данному моменту.

Это работает, но есть и недостатки:

Сообщения передаются с задержкой до 10 секунд (между запросами).

Даже если сообщений нет, сервер «атакуется» запросами каждые 10 секунд, даже если пользователь переключился куда-нибудь или спит. С точки зрения производительности, это довольно большая нагрузка.

Так что, если речь идёт об очень маленьком сервисе, подход может оказаться жизнеспособным, но в целом он нуждается в улучшении.

[**Длинные опросы**](https://learn.javascript.ru/long-polling#dlinnye-oprosy)

«Длинные опросы» – гораздо лучший способ взаимодействия с сервером.

Они также очень просты в реализации, и сообщения доставляются без задержек.

Как это происходит:

1. Запрос отправляется на сервер.

2. Сервер не закрывает соединение, пока у него не возникнет сообщение для отсылки.

3. Когда появляется сообщение – сервер отвечает на запрос, посылая его.

4. Браузер немедленно делает новый запрос.

Для данного метода ситуация, когда браузер отправил запрос и удерживает соединение с сервером, ожидании ответа, является стандартной. Соединение прерывается только доставкой сообщений.

Если соединение будет потеряно, скажем, из-за сетевой ошибки, браузер немедленно посылает новый запрос.

**Long Polling** — это технология, которая позволяет получать данные 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, придёт новое сообщение), либо истечет заданное время ожидания.

**Вебхук** – это механизм оповещения о происходящих в системе событиях посредством функций обратных вызовов. Когда случается интересующее клиента событие, сервер отправляет HTTP-запрос на URL-адрес, предоставленный клиентом для приема вебхуков.

Данный механизм, используемый веб-приложениями или сайтами, применяется для оповещения о произошедших событиях. Как только что-то произошло (например, пользователь изменил свои настройки, удалил полученное сообщение, оставил комментарий…), отсылается соответствующее уведомление. Благодаря этому отпадает необходимость постоянно запрашивать сервер о произошедших событиях. Достаточно лишь указать, куда именно должны отправляться такие уведомления и они будут приходить сразу же, как произойдёт определённое событие.

2 общих подхода для создания бота (web hook ; long pool запрос)  
бот отправляет данные от собеседника к серверу, кот мы разработали