# **v 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.

- http не имеет состояния, но имеет сессию(не существует связи между 2мя запросами, Но хотя ядро HTTP не имеет состояния, куки позволяют использовать сессии с сохранением состояния).

- Соединение управляется на транспортном уровне, и потому принципиально выходит за границы HTTP. Хотя HTTP не требует, чтобы базовый транспортного протокол был основан на соединениях, требуя только *надёжность*, или отсутствие потерянных сообщений (т.е. как минимум представление ошибки). Среди двух наиболее распространенных транспортных протоколов Интернета, TCP надёжен, а UDP -- нет. HTTP впоследствии полагается на стандарт TCP, являющийся основанным на соединениях, несмотря на то, что соединение не всегда требуется.

W3c – world wide web consortium - Консо́рциум Всеми́рной паути́ны — организация, разрабатывающая и внедряющая технологические стандарты для Всемирной паутины.

RFC - документ из серии пронумерованных информационных документов [интернета](https://tcinet.ru/press-centre/glossary/article.php?ELEMENT_ID=4920), которые содержат технические спецификации и стандарты, широко применяемые во всемирной сети и регламентирующие внутреннюю жизнь интернета. Некоторые из таких документов представляют собой отчеты рабочих групп или описания ресурсов. Первичной публикацией документов RFC занимается [IETF](https://tcinet.ru/press-centre/glossary/article.php?ELEMENT_ID=4866) под эгидой [ISOC](https://tcinet.ru/press-centre/glossary/article.php?ELEMENT_ID=4912) (Internet society, Общество интернета). Правами на RFC обладает именно ISOC.

Ниже перечислены общие функции, управляемые с HTTP:

· *Кэш -* Сервер может инструктировать прокси и клиенты: что и как долго кэшировать.

· *Ослабление ограничений источника -* Для предотвращения шпионских и других, нарушающих приватность, вторжений, веб-браузер обеспечивает строгое разделение между веб-сайтами. Только страницы из **того же источника** могут получить доступ к информации на веб-странице.

· *Аутентификация -* Некоторые страницы доступны только специальным пользователям. Базовая аутентификация может предоставляться через HTTP

· *Сессии -* Использование HTTP кук позволяет связать запрос с состоянием на сервере. Это создает сессию, хотя ядро HTTP -- протокол без состояния. Это полезно не только для корзин в интернет-магазинах, но также для любых сайтов, позволяющих пользователю настроить выход.

· *Прокси и тунелирование -* Серверы и/или клиенты часто располагаются в интранете, и скрывают свои истинные IP-адреса от других. HTTP запросы идут через прокси для пересечения этого сетевого барьера.

**Клиент-серверная архитектура** – архитектура, которая подразумевает 2 компонента (клиент, являющийся инициатором соединения и сервер). В качестве клиента выступает веб-браузер.

Когда мы говорим о протоколе http между клиентом и сервером ходит 2 типа сообщения:

1. От клиента к серверу ходит запрос (**request**)

2. От сервера к клиенту ходит ответ (**response**)

**Request**

- метод (get/post/put/option и тд)

- URI (описывает месторасположение сервера)

- версия протокола (HTTP/1.1)

- заголовки (пары: имя/заголовок)

- параметры (пары: имя/значение)

- расширение(тело)

Все заголовки могут быть 2 основных типов:

* стандартные (прописаны в протоколе HTTP)
* пользовательские (рекомендация: они должны начинаться с буквы «x»)

**Response**

§ Версия протокола

§ Код состояния (1xx,2xx,3xx,4xx,5xx, статус ответа)

§ Пояснение к коду состояния

§ Расширение

§ Заголовки (пары: имя/заголовок)

Запрос (**Request**): серверный объект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки. Обычно объект Request предоставляет возможность хранить данные в формате ключ/значение.

Ответ (**Response**): серверный объект, который автоматически формируется сервером, при получении http-запроса, заполняется данными серверными программным кодом, преобразуется в http-ответ и отправляется клиенту.

**Методы**

ü options, get (с передачей параметров в адресной строке)

ü head, post (параметры передаются в теле запроса)

ü put, delete, trace, connect, extension-method

**Заголовки**

– **General** – общие заголовки, используются в запросах и ответах (cache-control, connection)

– **Request** – заголовки, используются только в запросах

– **Response** – заголовки, используются только в ответах

– **Entity** – заголовки, которые описывают сущность в ответах и запросах. Сущностью называется то, что находится в header.

В ответе может быть код состояния:

1. Информационные: 1xx

2. Успешный ответ: 2xx

3. Переадресация: 3xx

4. Ошибка клиента: 4xx

5. Ошибка сервера: 5xx

**Statless-протокол** – протокол, не сохраняющий состояния, то есть сервер не сохраняет никаких данных (состояние) между парами запросов-ответов.

**HTTPS** (Hypertext Transport Protocol Secure) – это протокол, который обеспечивает конфиденциальность обмена данными между сайтом и пользовательским устройством. HTTP – как транспорт. Протокол TLS – основан на ассиметричном криптовании, на этапе соединения они договариваются как криптуются. Сначала договариваются, а потом передают.

Безопасность информации обеспечивается за счет использования криптографических протоколов SSL/TLS, имеющих 3 уровня защиты:

1. **Шифрование данных** (Позволяет избежать их перехвата)

2. **Сохранность данных** (Любое изменение данных фиксируется)

3. **Аутентификация** (Защищает от перенаправления пользователя)

**Ф-я обратного вызова (callback)** – функция, которая вызывается после того, как пришел ответ от сервера, для его обработки.

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

**Web-приложение:** клиент-серверное приложение, применяющее для обмена данными протокол HTTP; может быть просто web-приложением (HTML+HTTP) или web-службой (API, HTTP-транспорт, формат XML, JSON).

**Порт** — программа на сервере, которая прослушивает входящие сообщения.

Веб-сервер представляет собой: HTTP-сервер + файлы. Именно HTTP-сервер взаимодействует с клиентом. Asp.net framework должен быть установлен только на сервере, на клиенте - его нет. Веб-приложение включает все: клиент + сервер.

**Web-приложения** представляют собой особый тип программ, построенных по архитектуре "клиент-сервер". Особенность их заключается в том, что само Web-приложение находится и выполняется на сервере - клиент при этом получает только результаты работы. Работа приложения основывается на получении запросов от пользователя (клиента), их обработке и выдачи результата. Передача запросов и результатов их обработки происходит через Интернет.

На стороне сервера Web-приложение выполняется специальным программным обеспечением (Web-сервером), который и принимает запросы клиентов, обрабатывает их, формирует ответ в виде страницы, описанной на языке HTML, и передает его клиенту. Одним из таких Web-серверов является Internet Information Services (IIS) компании Microsoft. Это единственный Web-сервер, который способен выполнять Web-приложения, созданные с использованием технологии ASP.NET.

**Клиент-серверная архитектура** – это архитектура, которая подразумевает две компоненты: клиент и сервер. Клиент является инициатором соединения.

**Архитектура веб-приложения**

Есть клиент и есть сервер – две части одной и той же программы, которые взаимодействуют между собой по протоколу HTTP.

Бывают случаи, когда один клиент работает с несколькими серверами. Может быть, когда несколько клиентов работают с одним сервером. Может быть, когда сервер обращается к другому серверу и первый сервер выступает в качестве клиента по отношению к другому серверу.

Между клиентом и сервером ходит 2 типа сообщений:

· от клиента к серверу – **request**

· от сервера к клиенту – **response**

**URI (uniform resource identifier)** – унифицированный идентификатор ресурса (документ, изображение, файл, служба, электронная почта).

**URL** – унифицированный локатор ресурса - URI, содержащий местонахождение ресурса и способ обращения к ресурсу.

**URN** – унифицированное имя ресурса – URI, не содержащее в себе месторасположение и метод доступа к ресурсу.

**PURPL** (persistent uniform resource locator) – постоянный унифицированный локатор ресурса. Доступ к конечному ресурсу через redirect. Это некоторая база данных, содержащая месторасположение и способ доступа к ресурсу.

**Клиент: веб-браузер**

Умеет интерпретировать программу, которую ему присылает сервер.

Веб-браузер умеет генерировать HTTP-запросы:

– если есть адресная строка, и он делает запрос

– есть ряд HTML-тегов, при интерпретации которых браузер делает запрос (form, a, img, script, link, audio, video)

– объект веб-браузера: XMLHTTPRequest

– JS API

Любой веб-браузер характеризуется моделью DOM. Браузер в себе содержит web engine (ядро браузера). На сегодняшний момент существует набор веб-движков, на базе которых работают браузеры. Стандарты веб-браузеров: HTML5, CSS3, SVG, JavaScript (ECMAScript5, ECMA-262-6, ECMA-262-7).

**Общие принципы построения веб-приложений**

– веб-ресурсы приложения

– запросы и ответы

– фильтры

– кэш (данных и вывода)

– слушатели событий

– принципы безопасности

**HTTP–сервер** – программа, которая слушает некоторый порт на своем компьютере.

**Порт** – номер приложения, которому адресован этот запрос. Понятие порт существует на уровне tsp-протокола. Порт слушает входящие сообщения.

**Веб-приложение** принимает заполненный request ответ, пустой ответ response, заполняет response и отдает его серверу.

**Сессия** – серверный объект, хранящий информацию о соединении с клиентом, создается при первом обращении. Время жизни: **timeout** (системный параметр, обычно равен 10-30 минут) – максимальное время между запросами клиента.

Если timeout превышен, то session разрушается и при последующем запросе создается новый экземпляр. Каждая сессия имеет собственный идентификатор (Session ID, 128 или больше бит (16 байт)). Каждый request принадлежит какой-то сессии. Сессия характеризуется двумя параметрами: timeout и session ID.

Обычно в request сервер записывает либо идентификатор этой сессии, либо просто программную ссылку на этот объект сессии.

*Если с одним и тем же сервером работает много клиентов, у каждого из них своя сессия.*

**Куки** – это порция информации, которая может быть сохранена на стороне клиента по инициативе сервера. Когда клиент делает первый запрос, сервер проверяет, есть ли у него заголовок с именем куки. Если этого заголовка нет, то он считает, что это первый запрос и для него создается сессия.

Клиент должен хранить информацию о том, в рамках какой сессии мы отправляем эти запросы. Браузер в себе сохраняет либо файл куки, но чаще сохраняет **localstorage** (поддерживается на стороне браузера).

**Протокол RFC 6265** – стандарт, описывающий механизм куки.

**Конфигурационный файл** (обычно xml) содержит в себе некоторые статические характеристики приложения. Он служит для создания контекста веб-приложения.

**Контекст веб-приложения** – системный объект общий для всех сессий. Предназначен для хранений информации об одном веб-приложений, общий для всех сессий. Как правило, формируется сразу при загрузке веб-сервера.

**Фильтр** – серверный объект, препроцессор запроса, предназначенный для предварительно обработки объекта request. К одному ресурсу может быть построена цепочка фильтров, последний в цепочке – ресурс. Фильтр может прервать цепочку и сам сформировать ответ клиенту.

Информация о цепочке фильтров и привязке этих фильтров к ресурсу хранится в контексте.

Фильтр может не пустить запрос к ресурсу. Он может сам ответить, не пуская его дальше: к следующему фильтру или ресурсу.

**Слушатели событий (Listener)** – серверные объекты для обработки событий жизненного цикла веб-приложения. С помощью Listener можно врезаться в ход выполнения приложения и что-то там изменит.

*Пример – событие создания контекста. Можно написать listener, который выполнится, когда создастся контекст. Это необходимо, например, когда надо записать динамические данные в контексте. Или может быть listener на создание сессии.*

**Кэш** – это системный объект, предназначенный для хранения данных в оперативной памяти с целью ускорения работы веб-приложения. Часто в кэше запоминаются response.

Для каждого запроса создается новый экземпляр приложения. Общим остается контекст. В рамках серии запросов остается сессия. На каждый новый запрос создается новый экземпляр приложения.

**Фильтры** загружены всегда, вместе с контекстом. Фильтры привязываются к URI.

**Постоянное соединение**. Использование одного TCP-соединения для многократных пар запрос-сервер вместо последовательного открытия новых соединений для каждой пары запрос-ответ. Клиент может запросить постоянное соединение с помощью заголовка *Connection: Keep-Alive*, сервер подтверждает заголовком Connection: Keep-Alive.

**Обычный запрос проходит несколько стадий**:

1. Открытие соединения.

2. Отправка запроса.

3. Получение ответа.

4. Закрытие соединения.

**Конвейерная обработка** (**HTTP pipelining**). Редко поддерживаются серверами.

**Пул соединение с базой данных**. Несколько предварительно и постоянно открытых соединений с сервером СУБД, которые используют приложения. Выбор подключения из пула по open, возврат в пул close. Если все подключения пула заняты, запрос на соединение ставиться в очередь.

**Пул соединение** – два постоянно открытых коннекта в базе данных. К этому пулу устанавливается очередь запросов. Можно регулировать количество этих соединений в зависимости от нагрузки.

Создается несколько соединений, к пулу отправляется sql-запрос, который пул перенаправляет к БД.

**Кэширование на стороне браузера**: управление заголовками. Кеширование на стороне браузера описано в протоколе 7234. Если запрос обыкновенный, без кэширования, то при каждом новом запросе эта картинка будет скачиваться. Если картинка кэшируется, то все сохраняется на стороне браузера и запрос получается легковесным.

**Краткий вариант объяснения**

У нас есть объекты **response**, **request**. Они всегда ходят парой. Время их жизни совпадает. У нас есть **ресурс** - это наш **программный код**, который генерирует нам ответ. В объекте request у нас есть **память**, которой программист может воспользоваться. Также у нас есть объект **session**. В нем тоже есть память. Есть **ID** сессии и **timeout** - время жизни. Сессия живет один сеанс. Очень важно, чтобы были **cookies**, иначе работать не будет. Сессия связана с нашим запросом.

У нас есть **конфигурационный файл веб-приложения**, который содержит все параметры приложения. На базе этого файла создается контекст приложения.

**Контекст веб-приложения** - это тоже системный объект (как сессия, запрос или ответ). Он общий для всех сессий! Он хранит информацию о веб-приложении. В нем соответственно своя память, которую программист может тоже использовать.

**Фильтр** - еще один системный объект, он нужен для предварительной обработки запросов и ответов. Он решает куда отдать твой запрос дальше. Может быть несколько фильтров и они будут друг другу их "перефутболивать", как говорил Смелов. Фильтр может вообще не отдать твой запрос на ресурс и, грубо говоря, сформировать ответ сам. Например, это используется для шифрования или расшифрования информации. Информация об этих всех фильтрах (**цепочке**) содержится в контексте.

**Слушатели событий** (листенер) - еще одни специальный объект на стороне сервера, используются для обработки событий жизненного цикла приложения. То есть, мы можем врезаться в ход выполнения нашего приложения. (например, мы это можем делать в файле global.asax). Жизненные циклы это: инит, лоад, прелоадер, анлоад, диспоуз.

**Кэш** - еще один объект. Есть специальная программа кэш-менеджер, которая управляет объектом кэш.

**Как это работает**

Мы отправляем http-запрос от клиента к серверу. Общий вид запроса - битовая последовательность. Http-сервер состоит из http-драйвера и ресурс-программный код. Драйвер первый принимает http-запрос. Он формирует 2 серверных объекта: реквест и респонс (пустой). Далее проверяется это новая сессия или нет. Дальше кэш-менеджер смотрит, надо ли отправлять на ресурс или нет. Допустим нет, тогда из кэша берет данные и назад их отсылает. А если в кэше нет данных, то идет на фильтр. Фильтр может отослать на другой фильтр (тогда будет цепочка) или на ресурс, а может решить, что не будет никуда отсылать.

Так же у нас есть контекст. Он хранит информацию. Используем его, если нужно что-то сохранить во всех сессиях! Он общий для всех сессий. Листенеры - они нужны, если нам нужная какая-то обработка дополнительная в жизненном цикле нашего приложения.

Дальше request + response переходят на программу http-сервера и в этой программе есть http-обработчики (хэндлеры) и эти обработчики принимают request и response. Из request программа, которая находится на сервере понимает, что именно клиент хочет получить от нее, эта программа заполняет response.

Обработчик формирует нам response и шлет обратно на http-драйвер. Http-драйвер преобразовывает в битовую последовательность и отправляет клиенту.

Клиент получает ответ в виде битовой последовательности, преобразовывает в ответ, который понятен для него - html-разметка. Действия повторяются несколько раз. На 1 запрос - 1 ответ. Так как много клиентов можно быть, а серверов меньше, то чтобы не перегружать сервер он удаляет все экземпляры, которые создаются при запросе.

**Таким образом у нас объекты**

1. Реквест
2. Респонс
3. Сессион
4. Куки с указателем на сессию на СЕРВЕРЕ
5. Контекст (кот. создаётся на базе конфигурационного файла)
6. Фильтр
7. Кэш с программой кэш менеджер
8. Http-драйвер

**Синхронный** — значит, поочередный, **асинхронный** – значит, очередности нет (это запрос без ожидания ответа от сервера (Интернет). Если мы хотим синхронизировать 2 процесса, это значит, что мы хотим установить порядок их работы. А если процессы асинхронные, то это значит, что мы не знаем, в каком порядке они будут выполняться.

**Асинхронный запрос** — запрос, который мы с вами выполняем и дальше начинаем делать что-то другое. А потом, когда приходит ответ, с помощью какого-то механизма мы обрабатываем ответ от сервера. Этот какой-то механизм — это объект JS, который живет в браузере и называется **XMLHttpRequest**. Насчет этого объекта долго ругались создатели браузеров и они договорились о том, что он будет себя одинаково вести во всех браузерах. С помощью JS мы можем создать экземпляр этого объекта и с помощью этого экземпляра можем делать асинхронные запросы. Т.е. если мы с одной страницы хотим выдать несколько асинхронных запросов, мы должны создать несколько экземпляров **XMLHttpRequest** и каждый их них будет делать свой асинхронный запрос.

**Синхронный** - блокирует поток до получения сообщения, асинхронный – нет.

**NODEJS**: при разработке web-приложений используется **асинхронное программирование**.

ИЗ ЛЕКЦИИ!!!!!!!!!!!!

**Понятие асинхронности**: операция называется асинхронной, если ее выполнение осуществляется в 2 фазы: 1) заявка на исполнение; 2) получение результата; при этом участвуют два механизма: A-механизм, формирующий заявку и потом получающий результат; B-механизм, получающий заявку от A, исполняющий операцию и отправляющий результат A; продолжительность исполнения операции B-механизмом, как правило, непредсказуемо; в то время пока B-механизм исполняет операцию, А-механизм выполняет собственную работу. Применение асинхронности не противоречит применению многопоточности.

<https://ru.coursera.org/lecture/python-for-web/http-protokol-wUUuG>

# **2. HTTP-аутентификация.**

**BASIC, Digest, (**Forms**), TLS**

Басик – данные передаются в заголовке authorization(Base64 кодировке)

Digest – также как и басик, только пароль передается в хэшированном виде, с помощью MD5

BASIC

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#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.

DIGEST

Это аутентификация, при которой пароль пользователя передается в [хешированном](https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) виде. пароль хешируется всегда с добавлением произвольной строки символов, которая генерируется на каждое соединение заново. Таким образом при каждом соединении генерируется новый хеш пароля и перехват его ничего не даст. Дайджест-аутентификация поддерживается всеми популярными серверами и [браузерами](https://ru.wikipedia.org/wiki/%D0%91%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80).

FORMS

Аутентификация по [cookies](https://ru.wikipedia.org/wiki/Cookies) токену. Если cookie удастся похитить, то, подделав его, можно аутентифицироваться в качестве другого пользователя. В случае, когда вводимые данные плохо фильтруются или не фильтруются вовсе, похитить cookies становится не очень сложным предприятием. Чтобы как-то улучшить ситуацию используется защита по [IP-адресу](https://ru.wikipedia.org/wiki/IP-%D0%B0%D0%B4%D1%80%D0%B5%D1%81), то есть cookies [сессии](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%81%D1%81%D0%B8%D1%8F_(%D0%B2%D0%B5%D0%B1-%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D1%82%D0%B8%D0%BA%D0%B0)) связываются с IP-адресом, с которого пользователь изначально авторизовался в системе. Однако IP-адрес можно подделать используя [IP-спуфинг](https://ru.wikipedia.org/wiki/IP-%D1%81%D0%BF%D1%83%D1%84%D0%B8%D0%BD%D0%B3), поэтому надеяться на защиту по [IP-адресу](https://ru.wikipedia.org/wiki/IP-%D0%B0%D0%B4%D1%80%D0%B5%D1%81) тоже нельзя. На данный момент большинство браузеров используют куки с флагом HTTPonly, который запрещает доступ к cookies различным скриптам.

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

**TLS - Протокол защиты транспортного уровня**

Что такое **SSL**? SSL является аббревиатурой для **Secure Sockets Layer**. Это тип цифровой безопасности, которая позволяет зашифровать связь между веб-сайтом и веб-браузером. Технология в настоящее время **устарела** и полностью заменена TLS.

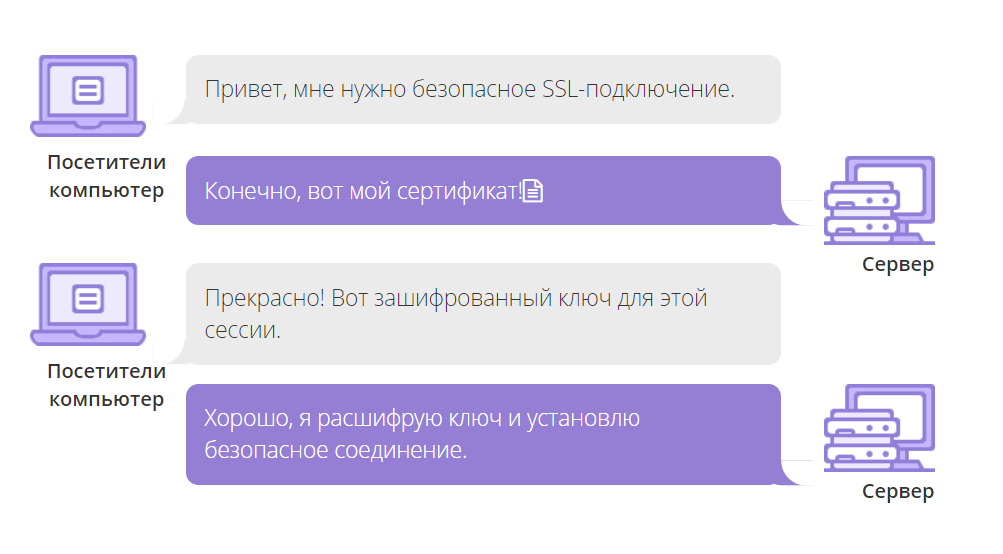
Что такое **TLS**? Это означает **Transport Layer Security** и обеспечивает конфиденциальность данных так же, как и SSL. Поскольку SSL фактически больше не используется, это правильный термин, который люди должны начать использовать.

Что таоке **HTTPS**? Это **безопасное** расширение HTTP. Веб-сайты, устанавливающие и настраивающие SSL/TLS-сертификат, могут использовать протокол HTTPS для установления безопасного соединения с сервером.

* Цель SSL/TLS — сделать соединение безопасным для передачи конфиденциальной информации, включая личные данные, информацию о платеже или регистрации.
* Это альтернатива простой передаче текстовых данных, в которой ваше соединение с сервером не зашифровано, и это затрудняет мошенникам и хакерам отслеживание соединения и кражу ваших данных.

## **Как работают сертификаты SSL/TLS?**

Сертификаты SSL/TLS работают путём цифровой привязки криптографического ключа к идентифицирующей информации компании. Это позволяет им шифровать передачу данных таким образом, что они не могут быть расшифрованы третьими лицами.



SSL/TLS работает, имея как частный, так и открытый ключ, а также ключи сеанса для каждого уникального безопасного сеанса. Когда посетитель вводит защищённый SSL-адрес в свой веб-браузер или переходит на безопасную страницу, браузер и веб-сервер устанавливают соединение.

Во время первоначального подключения общедоступные и закрытые ключи будут использоваться для создания ключа сеанса, который затем будет использоваться для шифрования и дешифрования передаваемых данных. Этот ключ сеанса останется действительным в течение ограниченного времени и будет использоваться только для данного сеанса.

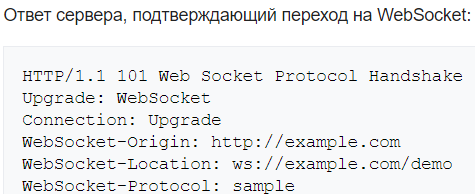
# **4. Протокол WebSockets, основные свойства, процедура установки соединения. WebSockets API.**

**WebSocket — протокол связи поверх** [**TCP**](https://ru.wikipedia.org/wiki/TCP)**-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени.Протокол WebSocket — это независимый протокол, основанный на протоколе TCP. Протокол прикладного уровня tsp/ip. Для передачи данных на транспортном уровне используют протокол tsp.**

**HTTP работает по принципу запрос-ответ, ничего не может отправить по своей воле, нужно ждать запрос от клиента, затем включать инфу в ответ. Сейчас хттп стал не очень удобен для некоторых задач, например чат. ХТТП-сервер должен отправить сообщение(запрос от клиента) всем клиентам, но он не может сам отправить сообщение клиентам, поэтому он вынужден ждать запроса от клиентов.**

**WS создает постоянное двунаправленное соединение между клиентов и серверов**

**Вначале клиент отправляет HTTP запрос, в котором просит переключить на WS соединение. До редакции черновика протокола номер** [**75**](http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-75) **включительно соединение WebSocket устанавливалось следующим образом. Запрос клиента:6**

****

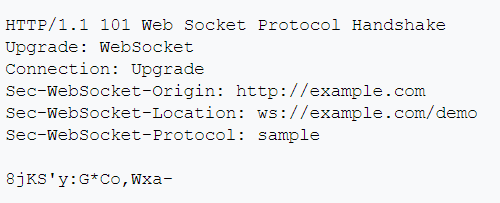
**Сразу после отправки ответа WebSocket-соединение считается установленным, клиент и сервер могут начинать двунаправленный обмен сообщениями по этому же** [**TCP**](https://ru.wikipedia.org/wiki/TCP)**-соединению. Для передачи текстового сообщения (в кодировке** [**UTF-8**](https://ru.wikipedia.org/wiki/UTF-8)**) необходимо перед ним передать нулевой байт, а после — байт со значением 255.**

**2 июня 2010 года в протокол WebSocket были внесены поправки, изменившие процедуру установления соединения WebSocket без сохранения обратной совместимости. В** [**76-й**](http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-76) **редакции черновика протокола WebSocket добавлена защита от поддельных запросов. Клиент, поддерживающий новую схему, присылает следующий запрос:**

****

**В запрос добавлены новые заголовки «Sec-WebSocket-Key1» и «Sec-WebSocket-Key2» и 8-байтовое тело запроса. Все они генерируются клиентом случайным образом.**

**Ответ сервера, подтверждающий переход на WebSocket:**

****

**Св-ва:**

* **Порты 80/443**
* **префикс url ws/wss (ws://**[**www.asozykin.ru/chat**](http://www.asozykin.ru/chat)**)**
* **дуплексный**

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

**10\_fetch\_xmlhttprequest.js**

var http = require('http');

var fs = require('fs');

http.createServer(function (request,response) {

if (request.url==='/api/name')

{

response.writeHead(200, {'content-Type': 'text/plain; charset=utf-8'});

response.end('Hello');

}

if (request.url==='/fetch') {

let html = fs.readFileSync('./10\_fetch.html');

response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});

response.end(html);

}

if (request.url==='/xmlhttprequest') {

let html = fs.readFileSync('./10\_xmlhttprequest.html');

response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});

response.end(html);

}

}).listen(5000);

console.log('Server running at http://localhost:5000');

**10\_fetch.html**

<html>

<head>

<meta charset="UTF-8">

<title>Fetch</title>

</head>

<body onload="init()">

<div id="full-name"></div>

<script>

function init() {

let settings = {

method: 'GET',

headers: {

'Content-Type': 'text/plain',

}

};

fetch('http://localhost:5000/api/name', settings)

.then(async response => {

document.getElementById('full-name').innerHTML = await response.text();

});

}

</script>

</body>

</html>

**10\_xmlhttprequest.html**

<html>

<head>

<meta charset="UTF-8">

<title>XmlHttpRequest</title>

</head>

<body onload="init()">

<div id="full-name"></div>

<script>

function init() {

let request = new XMLHttpRequest();

request.open('GET', 'http://localhost:5000/api/name', true);

request.onreadystatechange = () => {

if (request.readyState === 4) {

if (request.status === 200) {

console.log('OK. ' + request.responseText);

document.getElementById('full-name').innerHTML = request.responseText;

} else {

console.log('nope.\n' + request.statusCode);

}

}

};

request.send(null);

}

</script>

</body>

</html>

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

**11\_GET\_POST\_PUT\_DELETE.js**

var http = require('http');

let debug\_handler = (req, res)=>{

console.log(req.method, req.url);

res.writeHead(200,{'Content-Type':'application/json; charset=utf-8'});

res.end(`{"${req.method}":"${req.url}"}`);

}

let HTTP404 = (req, res)=>{

console.log(`${req.method}: ${req.url}, HTTP status 404`);

res.writeHead(200,{'Content-Type':'application/json; charset=utf-8'});

res.end(`{"error: ${req.method}: ${req.url}, HTTP status 404"}`);

};

let GET\_handler = (req, res)=>{debug\_handler(req,res)};

let POST\_handler = (req, res)=>{debug\_handler(req,res)};

let PUT\_handler = (req, res)=>{debug\_handler(req,res)};

let DELETE\_handler = (req, res)=>{debug\_handler(req,res)};

let OTHER\_handler = (req, res)=>{debug\_handler(req,res)};

let http\_handler = (req, res)=>{

switch (req.method) {

case 'GET': GET\_handler(req,res); break;

case 'POST': POST\_handler(req,res); break;

case 'PUT': PUT\_handler(req,res); break;

case 'DELETE': DELETE\_handler(req,res); break;

default: HTTP404(req, res); break;

}

};

let server = http.createServer();

server.listen(3000, (v)=>{console.log('Server running at http://localhost:3000/')})

.on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})

.on('request', http\_handler);

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

**13\_static.js**

let http = require('http');

let stat = require('./m13')('./static');

let http\_handler = (req, res)=>{

if (stat.isStatic('html', req.url)) stat.sendFile(req, res, { "Content-Type": "text/html; charset=utf-8;" });

else if (stat.isStatic('css', req.url)) stat.sendFile(req, res, { "Content-Type": "text/css; charset=utf-8;" });

else if (stat.isStatic('js', req.url)) stat.sendFile(req, res, { "Content-Type": "text/javascript; charset=utf-8;" });

else if (stat.isStatic('docx', req.url)) stat.sendFile(req, res, { "Content-Type": "application/msword" });

else if (stat.isStatic('png', req.url)) stat.sendFile(req, res, { "Content-Type": "image/png" });

else stat.writeHTTP404(req, res);

};

let server = http.createServer();

server.listen(3000, (v)=>{console.log('Server running at http://localhost:3000/13.html')})

.on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})

.on('request', http\_handler);

**m13.js**

function Stat(sfn = './static') {

this.STATIC\_FOLDER = sfn;

let pathStatic = (fn) => { return `${this.STATIC\_FOLDER}${fn}`; }

this.writeHTTP404 = (req, res) => {

res.statusCode = 404;

res.statusMessage = 'Resourse not found';

res.end("Resourse not found");

}

let fs = require('fs');

let pipeFile = (req, res, headers) => {

res.writeHead(200, headers);

fs.createReadStream(pathStatic(req.url)).pipe(res);

}

this.isStatic = (ext, fn) => {

let reg = new RegExp(`^\/.+\.${ext}$`);

return reg.test(fn);

};

this.sendFile = (req, res, headers) => {

fs.access(pathStatic(req.url), fs.constants.R\_OK, err => {

if (err)

this.writeHTTP404(res);

else

pipeFile(req, res, headers);

});

};

}

module.exports = (parm) => { return new Stat(parm); }

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

**14\_GET\_query.js**

let http = require('http');

let url = require('url');

let handler = (req, res)=>{

if (req.method = 'GET'){

let p = url.parse(req.url, true);

let result = '';

**let q = url.parse(req.url, true).query;**

if(!(p.pathname == '/favicon.ico')){

result = `href: ${p.href}<br/>` +

`path: ${p.path}<br/>` +

`pathname: ${p.pathname}<br/>` +

`search: ${p.search}<br/>`;

**for(key in q) {result += `${key} = ${q[key]}<br/>`;}**

}

res.writeHead(200,{'Content-Type':'text/html; charset=utf-8'});

res.write('<h1>GET-params</h1>');

res.end(result);

}

else{

res.writeHead(200,{'Content-Type':'text/html; charset=utf-8'});

res.end('for other http-methods not so');

}

}

let server = http.createServer();

server.listen(3000, (v)=>{console.log('Server running at http://localhost:3000/')})

.on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})

.on('request', handler);

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

let http = require('http');

let url = require('url');

let handler = (req, res)=>{

if (req.method = 'GET'){

let p = url.parse(req.url, true);

let result = '';

if(!(p.pathname == '/favicon.ico')){

result = `pathname: ${p.pathname}<br/>`;

**decodeURI(p.pathname).split('/').forEach(e => {result += `${e}<br/>`});**

}

console.log(p.pathname.split('/'));

res.writeHead(200,{'Content-Type':'text/html; charset=utf-8'});

res.write('<h1>URL-params</h1>');

res.end(result);

}

else{

res.writeHead(200,{'Content-Type':'text/html; charset=utf-8'});

res.end('for other http-methods not so');

}

}

let server = http.createServer();

server.listen(3000, (v)=>{console.log('Server running at http://localhost:3000/')})

.on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})

.on('request', handler);

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

**16\_POST\_params.js**

let http = require('http');

let url = require('url');

let qs = require('querystring');

let fs = require('fs');

let handler = (req, res)=>{

if (req.method == 'GET'){

res.writeHead(200,{'Content-Type':'text/html; charset=utf-8'});

res.end(fs.readFileSync('./16\_form.html'));

}

else if (req.method == 'POST'){

let result = '';

req.on('data', (data)=>{result+=data;})

req.on('end', ()=>{

result += '<br/>';

let o = qs.parse(result);

for(let key in o) {result += `${key} = ${o[key]}<br/>`}

res.writeHead(200,{'Content-Type':'text/html; charset=utf-8'});

res.write('<h1>URL-params</h1>');

res.end(result);

});

}

else{

res.writeHead(200,{'Content-Type':'text/html; charset=utf-8'});

res.end('for other http-methods not so');

}

}

let server = http.createServer();

server.listen(3000, (v)=>{console.log('Server running at http://localhost:3000/')})

.on('error', (e)=>{console.log('server.listen(3000): error: ', e.code)})

.on('request', handler);

**16\_form.html**

<html>

<head>

<meta charset="UTF-8">

<title>POST</title>

</head>

<body>

<h1>POST</h1>

<div style="margin: 20px; width:800px; padding:5px;">

<form method="POST" action="/">

<div class="row">

<label class="col-2">ОТПРАВИТЕЛЬ</label>

<input class="col-3" name="reciever" placeholder="x@x.xx"/>

</div>

<div class="row">

<label class="col-2">ПОЛУЧАТЕЛЬ</label>

<input class="col-3" name="sender" placeholder="x@x.xx"/>

</div>

<div class="row">

<label class="col-2">СООБЩЕНИЕ</label>

<input class="col-3" name="message" placeholder="message"/>

</div>

<div class="row">

<input type="submit" class="col-3 offset-2" value="OK"/>

</div>

</form>

</div>

</body>

</html>

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

**SERVER**

|  |
| --- |
| let http = require('http');  const url = require('url');  let server = http.createServer();  let handler = (req, res) => {  if (url.parse(req.url, true).pathname === '/json') {  console.log(req.url);  if (req.method == 'POST') {  let ch = "";  req.on('data', (data) => {  ch += data;  });  req.on('end', () => {  let resObj = JSON.parse(ch);  res.end("{ \"17\_postJSON\": \" Staci\", \"x\_plus\_y\":" + (resObj.x + resObj.y) + "}");  });  }  }  };  server.listen(5000, (v) => { console.log('Running') })  .on('error', (e) => { console.log('Error: ', e.code) })  .on('request', handler); |

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

**SERVER**

|  |
| --- |
| let http = require('http');  let fs = require('fs');  let handler = (req, res) => {  if(req.method == 'GET') {  res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});  res.end(fs.readFileSync('./exampleHTTPupload.html'));  }  else if (req.method == 'POST') {  let result = '';  req.on('data', data => {result += data;});  req.on('end', () => {  res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});  res.write('<H1>FileUpload</H1>');  res.end(result);  });  }  else  {  res.end('this method not allowed');  }  }  let server = http.createServer();  server.listen(3000, (v) => {console.log('serv.listen(3000)')})  .on('error', (e) => {console.log(e.message)})  .on('request', handler**)** |

**exampleHTTPupload.html**

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <title>Document</title>  </head>  <body>  <form name="upload" id="upload" method="post" enctype="multipart/form-data">  <input type="file" name="myfile">  <input type="submit" value="Загрузить">  </form>  </body>  </html> |

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

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

**CLIENT**

|  |
| --- |
| let http = require('http');  let qs = require('querystring');  let params = qs.stringify({x:2, y:3});  let path = '/parameter?' + params;  let options = {  host: 'localhost',  port: '5000',  path: path,  method: 'GET'  }  const req = http.request(options, (res) => {  res = '';  req.on('data', (data) => {  res += data;  })  req.on('end', () => {console.log(res)});  });  req.on('error', (e) => { console.log(e.message);});  req.end(); |

**SERVER**

|  |
| --- |
| **l**et http = require('http');  const url = require('url');  let server = http.createServer();  let handler = (req, res) => {  if (url.parse(req.url).pathname === '/parameter') {  console.log(req.url);  if (typeof url.parse(req.url, true).query.x != 'undefined' && typeof url.parse(req.url, true).query.y != 'undefined') {  let x = parseInt(url.parse(req.url, true).query.x);  let y = parseInt(url.parse(req.url, true).query.y);  if (Number.isInteger(x) && Number.isInteger(y)) {  let sum = x + y;  let sub = x - y;  let mul = x \* y;  let o = x / y;  console.log('x + y = ' + sum +  ', x - y = ' + sub +  ', x \* y = ' + mul +  ', x / y = ' + o);  res.end('x + y = ' + sum +  ', x - y = ' + sub +  ', x \* y = ' + mul +  ', x / y = ' + o);  } else {  res.end('Error. Parameter is not a number')  }  }  }  };  server.listen(5000, (v) => { console.log('Running') })  .on('error', (e) => { console.log('Error: ', e.code) })  .on('request', handler); |

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

**CLIENT**

|  |
| --- |
| let http = require('http');  let query = require('querystring');  let parms = query.stringify({x:3, y:4, s:'xxx'});  let path = '/mypath';  console.log('path', path);  console.log('parms ', parms);  let options = {  host: 'localhost',  path: path,  port: 5000,  method:'POST'  }  const req = http.request(options, (res) => {  console.log('http.request: response:', res.statusCode);  console.log('http.request: statusMessage:', res.statusMessage);  let data = '';  res.on('data', (chunk) => {  data += chunk.toString('utf8');  });  res.on('end', () => {console.log('http.request: end: body = ', data);});  })  req.on('error', (e) => {console.log('http.request: error:', e.message);});  req.write(parms);  req.end(); |

**SERVER**

|  |
| --- |
| let http = require('http');  const url = require('url');  let handler = (req, res) => {  if (url.parse(req.url).pathname === '/mypath') {  console.log(req.url);  if (req.method == 'POST') {  let ch = "";  req.on('data', (data) => {  ch += data;  });  req.on('end', () => {  console.log(ch);  let x = parseInt(url.parse(req.url + '?' + ch, true).query.x);  let y = parseInt(url.parse(req.url + '?' + ch, true).query.y);  let s = url.parse(req.url + '?' + ch, true).query.s;  res.end(x + ' ' + y + ' ' + s);  });  }  }  };  let server = http.createServer();  server.listen(5000, (v) => { console.log('Running') })  .on('error', (e) => { console.log('Error: ', e.code) })  .on('request', handler); |

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

**CLIENT**

|  |
| --- |
| let http = require('http');  let parms = JSON.stringify({\_\_comment:'Anastason', x:1, y:2, s:'message', m:['a','b','c','d'], o:{surname:'Stalmashenko', name:'Anastason'}});  let path = '/json';  console.log('path', path);  console.log('parms ', parms);  let options = {  host: 'localhost',  path: path,  port: 5000,  method:'POST'  }  const req = http.request(options, (res) => {  console.log('http.request: response:', res.statusCode);  console.log('http.request: statusMessage:', res.statusMessage);  let data = '';  res.on('data', (chunk) => {  data += chunk.toString('utf8');  });  res.on('end', () => {console.log('http.request: end: body = ', data);});  })  req.on('error', (e) => {console.log('http.request: error:', e.message);});  req.write(parms);  req.end(); |

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

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

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

|  |
| --- |
| **const http = require('http');**  **const fs = require('fs');**  **const file = fs.createWriteStream("file.bmp");**  **let options = {**  **host:'localhost',**  **path:'/bmp/MyFile.bmp',**  **port:3000,**  **method:'GET'**  **}**  **const req = http.request(options,(res) => {res.pipe(file);});**  **req.end();** |

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

|  |
| --- |
| **const http = require('http');**  **const WebSocket = require('ws');**  **const fs = require('fs');**  **http.createServer((req, res)=>{**  **if(req.method == 'GET' && req.url == '/start'){**  **res.writeHead(200, {'Content-Type':'text/html; charset=utf-8'});**  **res.end(fs.readFileSync('./10.html'));**  **}else{**  **res.statusCode=400;**  **res.end('<h1>400</h1>');**  **}**  **}).listen(3000);**  **console.log('Server running at http://localhost:3000/');**  **let k=0;**  **const ws = new WebSocket.Server({port:4000, host:'localhost', path:'/ws'});**  **ws.on('connection', (wss)=>{**  **let mess;**  **wss.on('message', message =>{**  **console.log(`Received message => 10-01-client: ${message}`);**  **mess = message;**  **})//last and current**  **setInterval(()=>{wss.send(`10-01-server: ${mess}->${++k}`)}, 5000);**  **})**  **ws.on('error', (e)=>{console.log('ws server error', e)});**  **console.log(`ws server: host:${ws.options.host}, port:${ws.options.port}, path:${ws.options.path}`);** |

**client:**

**let socket = new WebSocket(‘ws://localhost:4000/ws’);**

**socket.onopen = ()**

**socket.onerror = (e) => {e.message}**

**socket.onmessage = (msg)=> {msg.data}**

**socket.onclose = (msg)=>{msg.code}**

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

|  |
| --- |
| **const http = require('http');**  **const WebSocket = require('ws');**  **const fs = require('fs');**  **http.createServer((req, res)=>{**  **if(req.method == 'GET' && req.url == '/start'){**  **res.writeHead(200, {'Content-Type':'text/html; charset=utf-8'});**  **res.end(fs.readFileSync('./10.html'));**  **}else{**  **res.statusCode=400;**  **res.end('<h1>400</h1>');**  **}**  **}).listen(3000);**  **console.log('Server running at http://localhost:3000/');**  **let k=0;**  **const ws = new WebSocket.Server({port:4000, host:'localhost', path:'/ws'});**  **ws.on('connection', (wss)=>{**  **let mess;**  **wss.on('message', message =>{**  **console.log(`Received message => 10-01-client: ${message}`);**  **mess = message;**  **})//last and current**  **setInterval(()=>{wss.send(JSON.stringify({msg: ++k})}, 5000);**  **})**  **ws.on('error', (e)=>{console.log('ws server error', e)});**  **console.log(`ws server: host:${ws.options.host}, port:${ws.options.port}, path:${ws.options.path}`);** |

|  |
| --- |
| **const WebSocket = require('ws');**  **const ws = new WebSocket('ws://localhost:4000/ws);**  **ws.on('open', ()=>{**  **ws.on('message', message =>{**  **console.log(`Received message => ${message}`)**  **message = JSON.parse(message).msg;**  **ws.send(message);**  **})**  **setTimeout(()=>{ws.close()}, 25000);**  **});** |

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

**SERVER**

|  |
| --- |
| const express = require('express');  const RPC = require('rpc-websockets').Server;  const PORT = 3000;  const HOST = 'localhost';  const app = express();  const wsrpc = new RPC({port: 4003, host: HOST});  const server = app.listen(PORT, HOST, () =>  {  const URL = `http://${HOST}:${PORT}`;  console.log('Listening on ' + URL);  })  .on('error', (e) => {console.log(`${URL} | error: ${e.code}`)});  wsrpc.setAuth(credentials => credentials.login === 'admin' && credentials.password === 'admin');  wsrpc.register('sum', params => params.reduce((a, b) => a + b, 0)).public();  wsrpc.register('square', square).protected();  function square(args)  {  if (args.length === 1) return Math.PI \* Math.pow(args[0], 2);  else if (args.length === 2) return args[0] \* args[1];  else return 0;  } |

**CLIENT**

|  |
| --- |
| const rpc = require('rpc-websockets').Client;  const wsrpc = new rpc('ws://localhost:4003');  wsrpc.on('open', () =>  {  wsrpc.call('sum', [2, 4, 6, 8, 10]).then(answer => console.log('sum: ' + answer));  wsrpc.login({login: 'admin', password: 'admin'})  .then(async login =>  {  if (login)  {  wsrpc.call('square', [5, 4]).then(answer => console.log('square: ' + answer));  }  else  {  console.log('Unauthorized');  }  });  }); |

# 

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

**24. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического SELECT-запроса.**

**25. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического INSERT-запроса. Пример.**

**26. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического UPDATE-запроса. Пример.**

**27. Разработка приложения, выполняющего запрос к SQL-базе данных: выполнение динамического DELETE-запроса. Пример.**

**2ый вопрос в билете**

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

**in-memory Database -** Резидентная база данных хранящаа информацию в оперативной памяти

**Хеш-табли́ца** — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары и выполнять три операции

Каким образом обеспечивается **персистентность**  данных в СУБД Redis?

**Персистентные** структуры — это структуры данных, которые при внесении в них каких-то изменений сохраняют все свои предыдущие состояния и доступ к этим состояниям. Redis сохраняет свое состояние на диск

4. Поясните назначение команд СУБД Redis: set, get, del, getset.

Вставка, получение, удаление, вставка нового значение + получение старого

5. Поясните назначение команд СУБД Redis: incr, decr.

Увеличить или уменьшить значения на 1

6. Поясните назначение команд СУБД Redis: mset, mget.

Вставка, получение нескольких полей за раз

7. Поясните назначение команд СУБД Redis: hset, hget.

hset - Устанавливает поле в хэше, хранящемся в ключе к значению. Если ключ не существует, создается новый ключ, содержащий хэш. Если поле уже существует в хэше, оно перезаписывается.

hget - Получает поле в хэша, хранящемся в ключе к значению

8. Поясните назначение команд СУБД Redis: hmset,hmget.

Hmset - Устанавливает для нескольких полей их соответствующие значения в хэше, хранящемся в ключе. Эта команда перезаписывает любые указанные поля, уже существующие в хэше. Если ключ не существует, создается новый ключ, содержащий хэш.

9. Поясните назначение команды СУБД Redis: exists.

Проверка на существование значения по ключу

**Server**

|  |
| --- |
| **const redis = require('redis');**  **let config = {**  **"host": "redis-10813.c135.eu-central-1-1.ec2.cloud.redislabs.com",**  **"port": 10813,**  **"no\_ready\_check": true,**  **"auth\_pass": "KXlVEmk67na4tAjnm92ep1phngABZZSl"**  **}**  **const client = redis.createClient(config);**  **client**  **.on('ready', () => {console.log('Client is ready');})**  **.on('error', (err) => {console.log("Error: " + err);})**  **.on('end', () => {console.log("End of connection");})**  **.on('ready', () => {**  **console.log('Redis connected');**  **Promise.resolve()**  **.then(() => {require(\_\_dirname + '/part/part2')(client);});**  **});** |

**CLIENT**

|  |
| --- |
| **module.exports = (client) => {**  **Promise.resolve()**  **.then(() => Set(client, 10000))**  **.then(() => Get(client, 10000));**  **};**  **function Set(client, count){**  **let timer = Date.now();**  **for (var i = 0; i < count; i++) {**  **let param = `s${i}`;**  **client.set(param, param, (err)=>{if(err) console.log("Error msg: "+err);});**  **}**  **console.log(`Task 2: Set time for ${count} queries: ${Date.now() - timer} ms`);**  **}**  **function Get(client, count){**  **let timer = Date.now();**  **for (var i = 0; i < count; i++) {**  **let param = `s${i}`;**  **client.get(param, (err, msg)=>{if(err) console.log("Error msg: "+err); console.log(msg);})**  **}**  **console.log(`Task 2: Get time for ${count} queries: ${Date.now() - timer} ms`);**  **}**  **function Del(client, count){**  **let timer = Date.now();**  **for (var i = 0; i < count; i++) {**  **let param = `s${i}`;**  **client.del(param, (err)=>{if(err) console.log("Error msg: "+err);});**  **}**  **console.log(`Task 2: Del time for ${count} queries: ${Date.now() - timer} ms`);**  **}** |

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

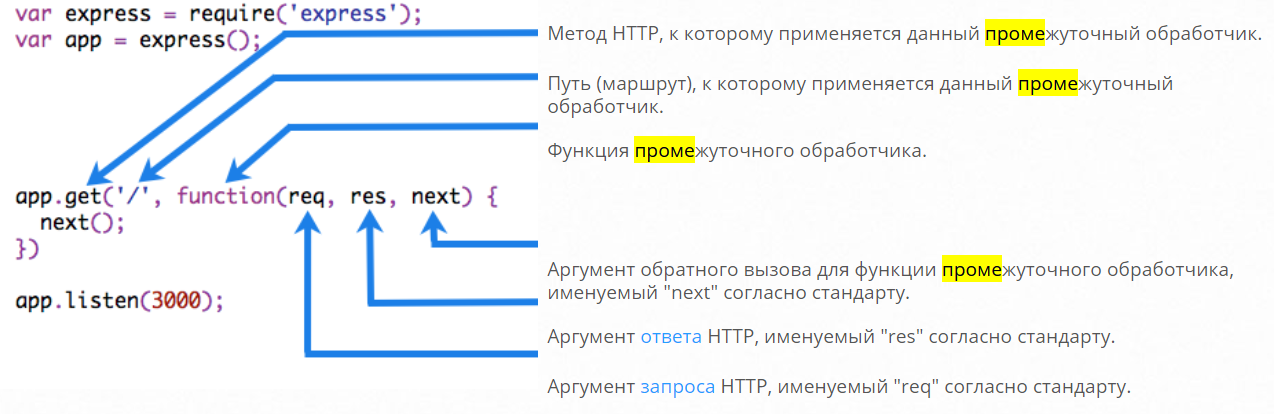
# **30. Пакет Express. Основные принципы работы. Middleware-код. Пример.**

Функции ***промежуточной обработки* (middleware) -** это функции, имеющие доступ к [объекту запроса](https://expressjs.com/ru/4x/api.html#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.



# 

|  |
| --- |
| **const express = require('express');**  **const app = express();**  **app.use((req, res, next)=>{**  **console.log(‘mid 1’);**  **next();**  **});**  **app.use((req, res, next)=>{**  **console.log(‘mid 2’);**  **res.send(‘second middleware’);**  **});**  **app.listen(3000, () =>**  **{**  **console.log(`Listening on http://localhost:3000`);**  **})** |

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

**const express = require('express');**

**const PORT = 3000;**

**const app = express();**

**const userLogin = require(\_\_dirname + '/controller/login/user');**

**app.use('/login/user', userLogin);**

**app.listen(PORT, () =>**

**{**

**console.log(`Listening on http://localhost:${PORT}`);**

**})**

**.on('error', (e) => {console.log(`Listener | error: ${e.code}`)});**

**/controller/login/user.js**

**const express = require('express');**

**const Route = express.Router();**

**Route.get('/:id', (req, res)=>{**

**res.send(req.params.id);**

**});**

**module.exports = Route;**

# **32. Пакет Express. Основные принципы работы. Статические файлы. Пример.**

|  |
| --- |
| **const express = require('express');**  **const PORT = process.env.PORT || 3000;**  **const app = express();**  **app.use('/static',express.static(\_\_dirname + '/views/static'));**  **app.listen(PORT, () =>**  **{**  **console.log(`Listening on http://localhost:${PORT}`);**  **})** |

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

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

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

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

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

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

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

# **40. 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.

**domain** – привязка cookie к поддомену.

**path** – путь, на который распространяется действие cookie.

**maxAge** –время жизни cookie в миллисекундах.

**Expires** –дата истечения жизни cookie.

**secure** –может применяться только с HTTPS.

**httpOnly = true** –может изменяться сервером.

# 

|  |
| --- |
| **const express = require('express');**  **const cookieparser = requre(‘cookie-parser’)();**  **const app = express();**  **app.use(cookieparser);**  **app.get('/, (req, res) => {**  **//noExpress**  **//res.writeHead(200, {‘Set-Cookie’: ‘i=2’ })**  **res.cookie(‘i’, 2, {signed:true}).send();**  **//signed - подписан, для слежки за изменением**  **//res.clearCookie(‘i’)**  **});**  **app.listen(3000, () =>**  **{**  **console.log(`Listening on http://localhost:3000`);**  **})**  **.on('error', (e) => {console.log(`Listener | error: ${e.code}`)});** |

# **41. 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.

|  |
| --- |
| **const app= require('express')();**  **const session = requre('express-session')({**  **secret: ‘123456789’**  **});**  **app.use(session);**  **app.get('/, (req, res) => {**  **res.send(req.session.mysesval);**  **});**  **app.listen(3000, () =>**  **{**  **console.log(`Listening on http://localhost:3000`);**  **})** |

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

**300** Multiple Choices («множество выборов»);

**301** Moved Permanently («перемещено навсегда»);

**302** Moved Temporarily («перемещено временно»);

**302** Found («найдено»);

**303** See Other («смотреть другое»);

**304** Not Modified («не изменялось»);

**305** Use Proxy («использовать прокси»);

**306** зарезервировано (код использовался только в ранних спецификациях);

**307** Temporary Redirect («временное перенаправление»);

**308** Permanent Redirect («постоянное перенаправление»).

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

spawn — child\_process.spawn запускает новый процесс с помощью заданной команды.

# **44. 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.

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

fork — метод child\_process.fork является особым случаем spawn() для создания дочерних процессов.

Модуль worker\_threads — это пакет, который позволяет создавать полнофункциональные многопоточные приложения Node.js.

Потоковый воркер (*thread worker*) — фрагмент кода (обычно извлекаемый из файла), созданный в отдельном потоке.

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

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

**JSON-RPC -** протокол удалённого вызова процедур, использующий JSON для кодирования сообщений.

**формат RPC-запроса**

**{**

**"jsonrpc": "2.0",** // Признак спецификации (обязательный)

**"method": "sum",** // Название метода, который должен быть выполнен на сервере (обязательный)

**"params": [42, 23],** // Параметры метода (не обязательное поле), массив, объект или простые типы

**"id": 1** // Идентификатор запроса. Может отсутствовать, но в это случае сервер выполнит метод, но не вернет ответ (согласно спецификации)

**}**

**RPC-уведомление** - Информация для сервера, не требует ответа. Должен отсутствовать id

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

**это бинарный формат инструкций для стековой виртуальной машины**

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

**это бинарный формат инструкций для стековой виртуальной машины**

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