NODEJS EXAM

- [x] 1
- [x] 2
- [x] 3
- [x] 4
- [x] 5
- [ ] 6
- [x] 7
- [x] 8
- [x] 9
- [ ] 10
- [x] 11
- [ ] 12
- [x] 13
- [ ] 14
- [ ] 15
- [ ] 16

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

HTTP (Hypertext Transfer Protocol) - протокол прикладного уровня передачи данных (изначально - в виде гипертекстовых документов).

Спецификация HTTP определяет, как именно запросы клиента должны быть построены и отправлен на сервер и то, как сервер должен отвечать на эти запросы.

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 (для тех, что не поддерживают шифрование), 443 (для тех, что поддерживают шифрование)
- Для адресации используется URI (универсального идентификатора ресурсов) или URN (универсального имени ресурсов)
- Поддерживается W3C, описан в нескольких RFC
- HTTP не привязан к конкретному типу данных. Это означает, что с помощью HTTP мы можем передавать любой тип данных, при условии, что и клиент и сервер “умеют” работать с данным типом данных. Сервер и клиент должны определить тип контента с помощью определённого типа MIME.

Непостоянные соединения применяются по умолчанию в версии 1.0 HTTP, в то время как постоянные соединения ~ в версии HTTP 1.1. Соединение называют ***непостоянным (non-persistent connection)***, если любое TCP-соединение закрывается сразу же, как только сервер отправляет клиенту требуемый объект. Это означает, что соединение используется только для одного запроса и одного ответа и не сохраняется для других запросов и ответов.

В случае ***постоянных соединений*** сервер, отправив ответ, оставляет соединение открытым, и, таким образом, следующие запросы и ответы между теми же клиентом и сервером могут отправляться через это же самое соединение. Такое соединение сервер закрывает лишь после того, как оно не используется в течение некоторого интервала времени.

Заголовки:

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

```
r = Cache-Control            ; Section 14.9
  | Connection               ; Section 14.10
  | Date                     ; Section 14.18
  | Pragma                   ; Section 14.32
  | Trailer                  ; Section 14.40
  | Transfer-Encoding        ; Section 14.41
  | Upgrade                  ; Section 14.42
  | Via                      ; Section 14.45
  | Warning                  ; Section 14.46
```

- *Request:* используются только в запросах;

```
= Accept                    ; Section 14.1
| Accept-Charset            ; Section 14.2
| Accept-Encoding           ; Section 14.3
| Accept-Language           ; Section 14.4
| Authorization             ; Section 14.8
| Expect                    ; Section 14.20
| From                      ; Section 14.22
| Host                      ; Section 14.23
| If-Match                  ; Section 14.24

| If-Modified-Since         ; Section 14.25
| If-None-Match             ; Section 14.26
| If-Range                  ; Section 14.27
| If-Unmodified-Since       ; Section 14.28
| Max-Forwards              ; Section 14.31
| Proxy-Authorization       ; Section 14.34
| Range                     ; Section 14.35
| Referer                   ; Section 14.36
| TE                        ; Section 14.39
| User-Agent                ; Section 14.43
```

- *Response:* используются только в ответах;

```
= Accept-Ranges             ; Section 14.5
| Age                       ; Section 14.6
| ETag                      ; Section 14.19
| Location                  ; Section 14.30
| Proxy-Authenticate        : Section 14.33

| Retry-After               ; Section 14.37
| Server                    ; Section 14.38
| Vary                      ; Section 14.44
| WWW-Authenticate          ; Section 14.47
```

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

```
= Allow                     ; Section 14.7
| Content-Encoding          ; Section 14.11
| Content-Language          ; Section 14.12
| Content-Length            ; Section 14.13
| Content-Location          ; Section 14.14
| Content-MD5               ; Section 14.15
| Content-Range             ; Section 14.16
| Content-Type              ; Section 14.17
| Expires                   ; Section 14.21
| Last-Modified             ; Section 14.29
| extension-header
```

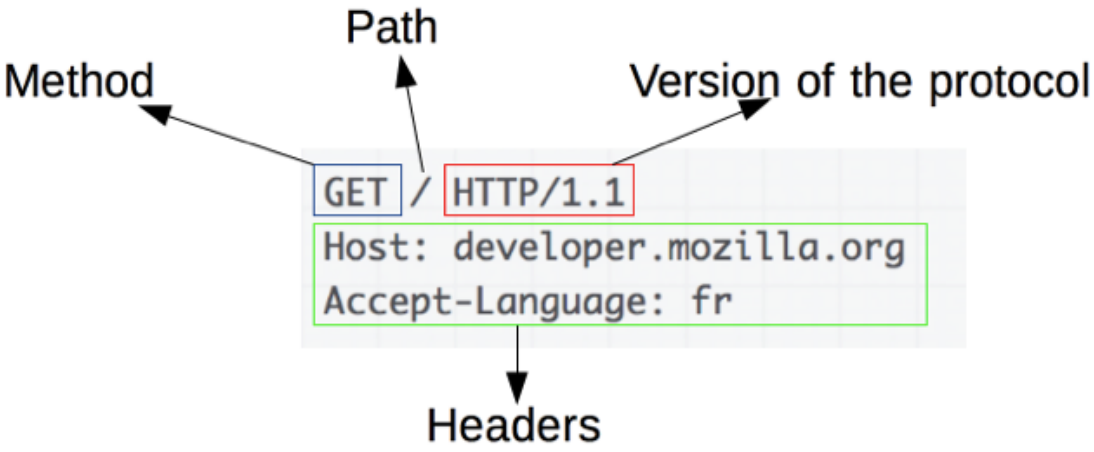
Response: Код состояния:

- *1xx:* информационные сообщения;
- *2xx:* успешный ответ;
- *3xx:* переадресация;
- *4xx:* ошибка клиента;
- *5xx:* ошибка сервера.

Структура Request (какую информацию мы может передать серверу в запросе):

- Стартовая строка (обязательный элемент)

- Header (опциональный элемент)
- Пустая строка, которая определят конец полей элемента header (обязательный элемент)
- Тело сообщения (опциональный элемент)



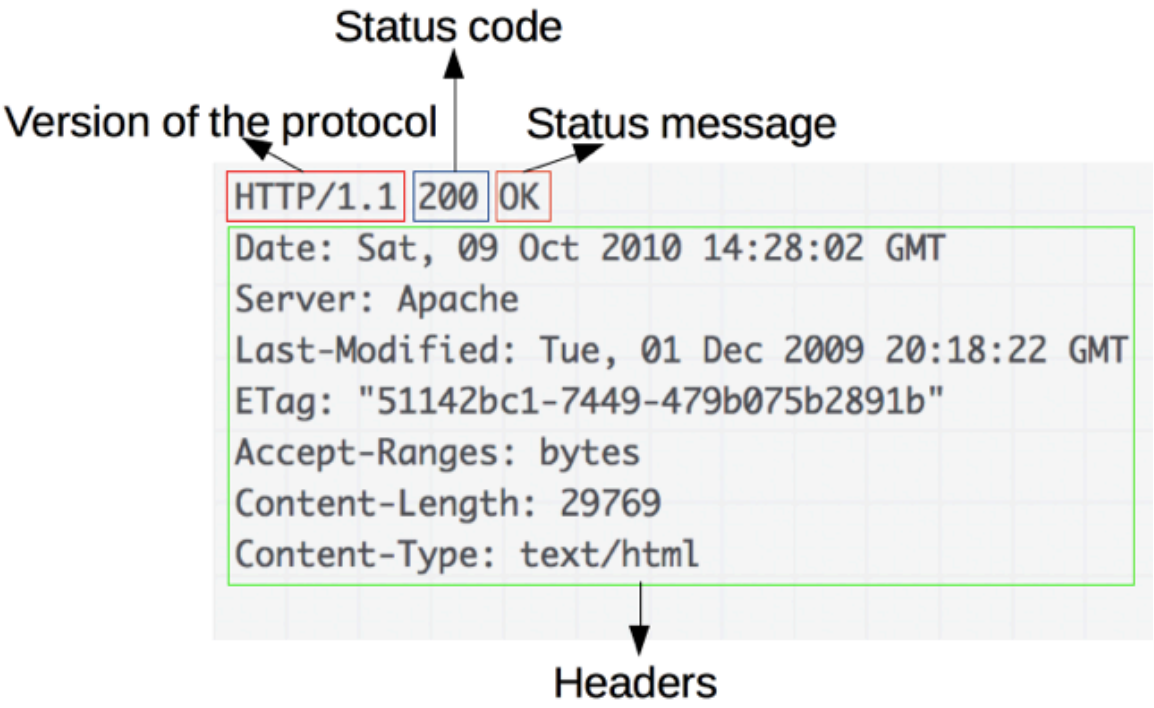
- метод;
- URI (часть без префикса http начин с EGH на скрине..);
- версия протокола (HTTP/1.1);
- заголовки (пары: имя/заголовок);
- параметры (пары: имя/заголовок);
- расширение.

В POST параметры передаются в теле запроса, GET - по URI.

GET /tutorials HTTP/1.1
Host: www.proselyte.net

Структура Response:

- Стартовая строка (обязательный элемент)
- Header (опциональный элемент)
- Пустая строка, которая определят конец полей элемента header (обязательный элемент)
- Тело сообщения (опциональный элемент)



- версия протокола (HTTP/1.1);
- код состояния (1xx, 2xx, 3xx, 4xx, 5xx), сообщающий об успешности запроса или причине неудачи;
- пояснение к коду состояния;
- заголовки (пары: имя/заголовок);
- расширение.

html - это ответ сервера (на скрине). Рекомендуется имя заголовка начинать с X.

HTTPS (от англ. HyperText Transfer Protocol Secure — безопасный протокол передачи гипертекста) — это расширение протокола HTTP, поддерживающее шифрование посредством криптографических протоколов SSL и TLS.

HTTPS не является отдельным протоколом передачи данных, а представляет собой расширение протокола HTTP с надстройкой шифрования; передаваемые по протоколу HTTP данные не защищены, HTTPS обеспечивает конфиденциальность информации путем ее шифрования; HTTP использует порт 80, HTTPS — порт 443.

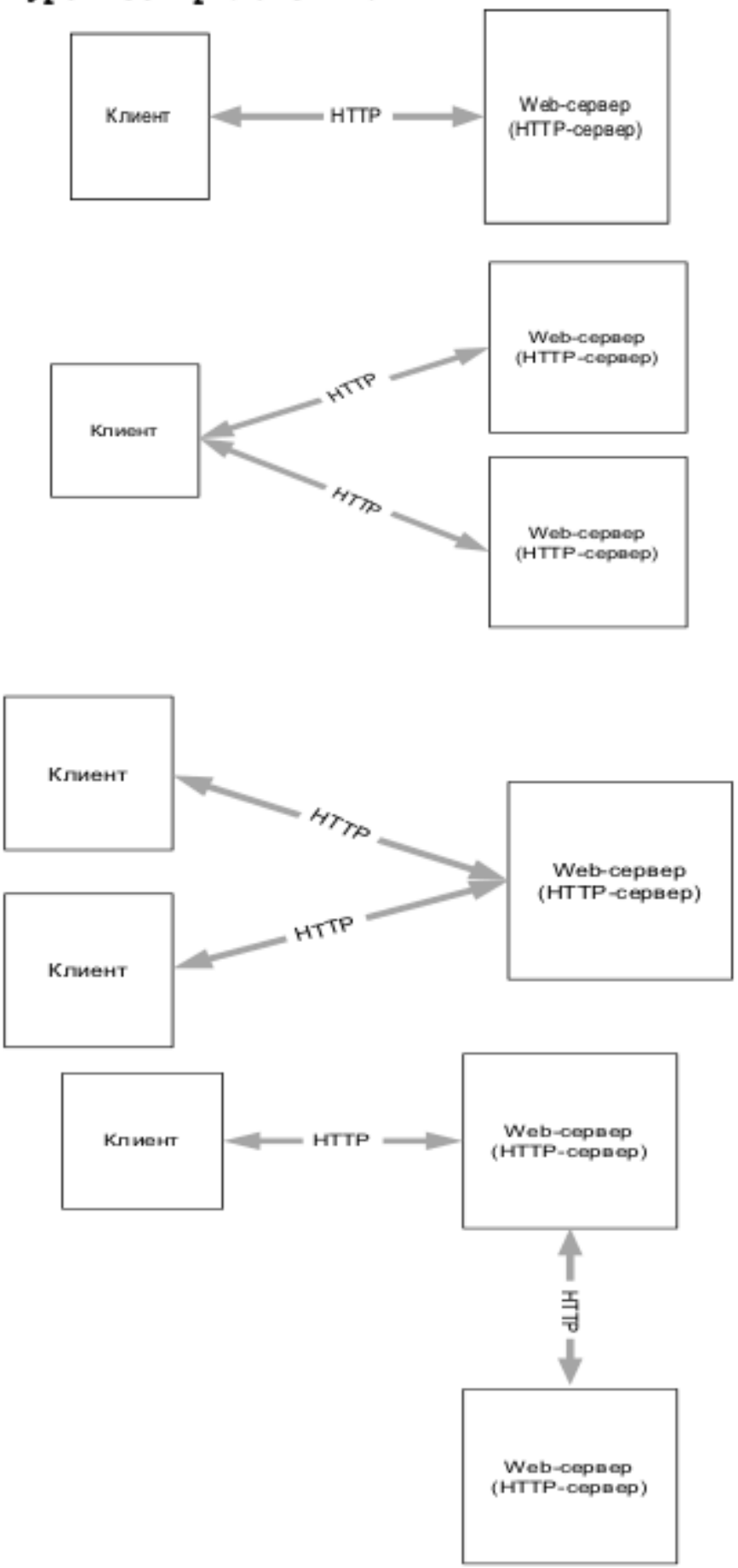
Обеспечение безопасной передачи данных необходимо на сайтах, где вводится и передается конфиденциальная информация (личные данные пользователей, детали доступа, реквизиты платежных карт) — на любых сайтах с авторизацией, взаимодействием с платежными системами, почтовыми сервисами. TLS расположен на уровень ниже протокола HTTP.

Веб приложения - приложения, имеющие архитектуру клиент-сервер, которые взаимодействуют по протоколу HTTP.

Клиентом служит браузер, сервером — веб-сервер. Связь происходит

посредством сети. В процессе обработки запроса пользователя Web-приложение компонует ответ на основе исполнения программного кода, работающего на стороне сервера, Web-формы, страницы HTML, другого содержимого, включая графические файлы. В результате, как уже было сказано, формируется HTML-страница, которая и отправляется клиенту. Получается, что результат работы Web-приложения идентичен результату запроса к традиционному Web-сайту, однако, в отличие от него, Web-приложение генерирует HTML-код в зависимости от запроса пользователя, а не просто передает его клиенту в том виде, в котором этот код хранится в файле на стороне сервера. То есть Web-приложение динамически формирует ответ с помощью исполняемого кода — так называемой исполняемой части.

Архитектура web-приложения:

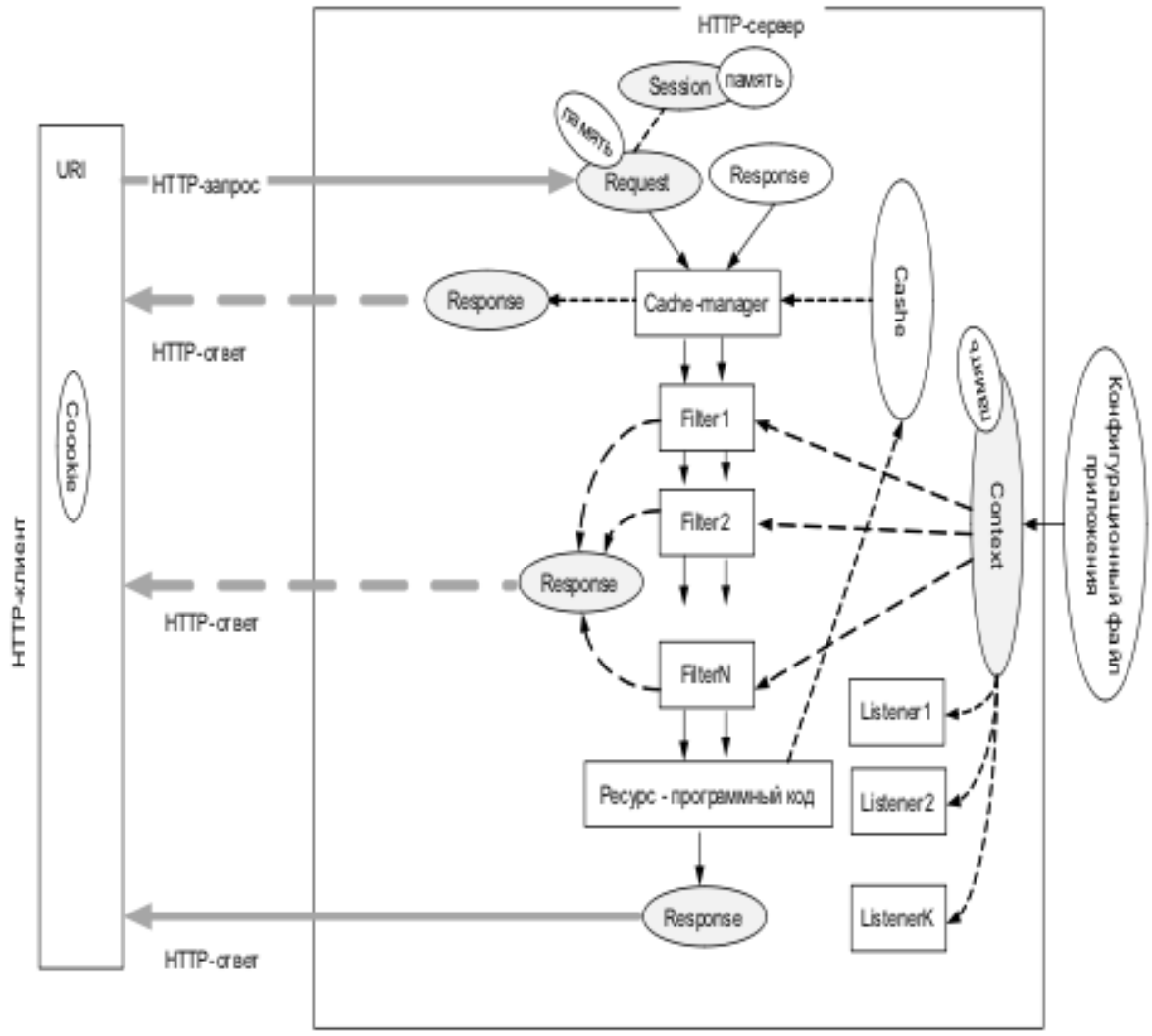


Web-ресурс приложения - сущность, расположенная на стороне сервера и имеющая URL/URI, к которой можно сделать http-запрос и получить http-ответ. Одно web-приложение представлено одним или более ресурсов.

3. **Web-ресурсы приложения: статические** - отправляются клиенту без изменения (html-страницы, рисунки, видео-файлы, …), **динамические** – динамически (программно) формируются на сервере и отправляются клиенту (сервлеты, JSP, http-обработчики, aspx-страницы,…). Ресурс может быть статическим относительно сервера и динамическим относительно клиента (html-страницы с JavaScript).

4. **Запрос(Request):** серверный объект, который образуется в результате обработки сервером http-запроса, поступающего от клиента и передается серверному программному коду для обработки. Содержит: всю информацию из http-запроса: метод, коллекция заголовков, коллекция параметров, поток данных … Обычно объект Request предоставляет возможность хранить данные в формате ключ/значение.

5. **Ответ(Response):** серверный объект, который автоматически формируется сервером, при получении http-запроса (одновременно с объектом Request), заполняется данными серверными программным кодом, преобразуется в http-ответ и отправляется клиенту. Содержит: всю информацию, которая должна быть помещена в http-ответ: статус, коллекция заголовков, поток данных, …



Асинхронный ввод-вывод — это форма обработки ввода/вывода, позволяющая продолжить обработку других задач, не ожидая завершения передачи.

При синхронном - основной поток будет заблокирован до тех пор, пока файл не будет прочитан, а это означает, что за это время ничего другого не может быть сделано.

```
const fs = require('fs')
let content
try {
    content = fs.readFileSync('file.md', 'utf-8')
} catch (ex) {
    console.log(ex)
}
console.log(content)
```

Функции обратного вызова (колбеки): если вы передаете функцию другой функции в качестве параметра, вы можете вызвать её внутри функции, когда она закончит свою работу. Нет необходимости возвращать значения, нужно только вызывать другую функцию с этими значениями. В

основе Node.js лежит принцип «первым аргументом в колбеке должна быть ошибка».

```
const fs = require('fs')
fs.readFile('file.md', 'utf-8', function (err, content) {
    if (err) {
        return console.log(err)
    }
console.log(content)
})
```

Что следует здесь выделить:

- обработка ошибок: вместо блока try-catch вы проверяете ошибку в колбеке
- отсутствует возвращаемое значение: асинхронные функции не возвращают значения, но значения будут переданы в колбеки

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

WebSocket — это самое кардинальное расширение протокола HTTP с его появления. Изначально синхронный протокол, построенный по модели «запрос — ответ», становится полностью асинхронным и симметричным. Теперь уже нет клиента и сервера с фиксированными ролями, а есть два равноправных участника обмена данными. Каждый работает сам по себе, и когда надо отправляет данные другому. Отправил — и пошел дальше, ничего ждать не надо. Вторая сторона ответит, когда захочет — может не сразу, а может и вообще не ответит. Протокол дает полную свободу в обмене данными, вам решать как это использовать.

Протокол WebSocket работает *над* TCP.

Установление WebSocket-соединения

Чтобы открыть веб-сокет-соединение, нам нужно создать объект new WebSocket, указав в url-адресе специальный протокол ws:

```
let socket = new WebSocket("ws://javascript.info");
```

Также существует протокол wss://, использующий шифрование. Это как HTTPS для веб-сокетов. Протокол wss:// не только использует шифрование, но и обладает повышенной надёжностью. С другой стороны, wss:// – это WebSocket поверх TLS (так же, как HTTPS – это HTTP поверх TLS), безопасный транспортный уровень шифрует данные от отправителя и расшифровывает на стороне получателя. Пакеты данных передаются в зашифрованном виде через прокси, которые не могут видеть, что внутри, и всегда пропускают их.

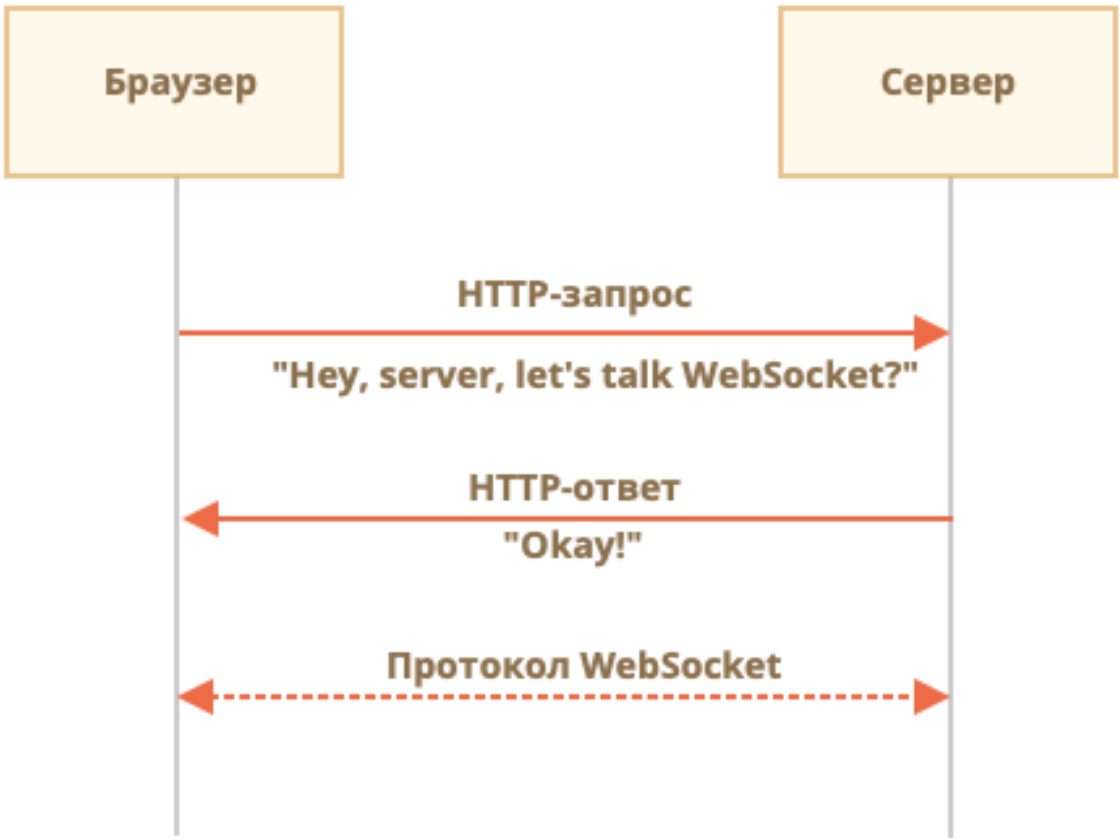
Объект **WebSocket** предоставляет API для создания и управления вебсокет-подключения к серверу, а также для отправки и получения данных в этом подключении.

Как только объект WebSocket создан, мы должны слушать его события. Их всего 4:

- **open** – соединение установлено,
- **message** – получены данные,
- **error** – ошибка,
- **close** – соединение закрыто.

А если мы хотим отправить что-нибудь, то вызов socket.send(data) сделает это.

Когда new WebSocket(url) создан, он тут же сам начинает устанавливать соединение. Это означает, что при соединении браузер отправляет по HTTP специальные заголовки, спрашивая: «поддерживает ли сервер WebSocket?». Если сервер в ответных заголовках отвечает «да, поддерживаю», то дальше HTTP прекращается и общение идёт на специальном протоколе WebSocket, который уже не имеет с HTTP ничего общего. Если сервер согласен переключиться на WebSocket, то он должен отправить в ответ код 101



3 Платформа Node.js, версии, назначение, основные свойства, структура, принципы работы, основные встроенные модули и их назначение, применение внешних модулей (пакетов). Web-приложение «Hello World». Пример.

NODEJS **-** программная платформа для разработки серверных web-

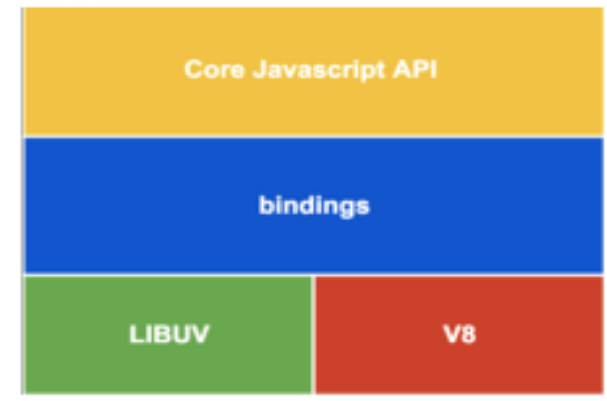
приложений на языке JS/V8. ***V8***- JS Engine, виртуальная машина, которая встроена в браузер и интерпретирует JS код.

V8 — это название JavaScript-движка, используемого в браузере Google Chrome. Именно он отвечает за выполнение JavaScript-кода, который попадает в браузер при работе в интернете. V8 предоставляет среду выполнения для JavaScript. DOM и другие API веб-платформы предоставляются браузером. JS-движок независим от браузера, в котором он работает.

Свойства NODEJS

- основан на *Chrome V8*;
- *среда (контейнер) исполнения* приложений на JavaScript;
- поддерживает механизм *асинхронности*;
- ориентирован на *события*;
- *однопоточный* (код приложения исполняется только в одном потоке, один стек вызовов); обычно в серверах для каждого соединения создается свой поток, в Node.js все соединения обрабатываются в одном JS-потоке;
- *не блокирует* выполнение кода при вводе/выводе (в файловой системе до 4х одновременно);
- в состав Node.js входят инструменты: *npm* – пакетный менеджер***; gyp*** - Python-генератор проектов; *gtest* – Google фреймворк для тестирования C++ приложений;
- использует библиотеки: *V8* – библиотека V8 Engine, *libuv* – библиотека для абстрагирования неблокирующих операций ввода/вывода; *http-parser* – легковесный парсер http-сообщений (написан на C и не выполняет никаких системных вызовов); *c-ares* - библиотека для работы с DNS; *OpenSSL* – библиотека для криптографии; *zlib* – сжатие и распаковка.

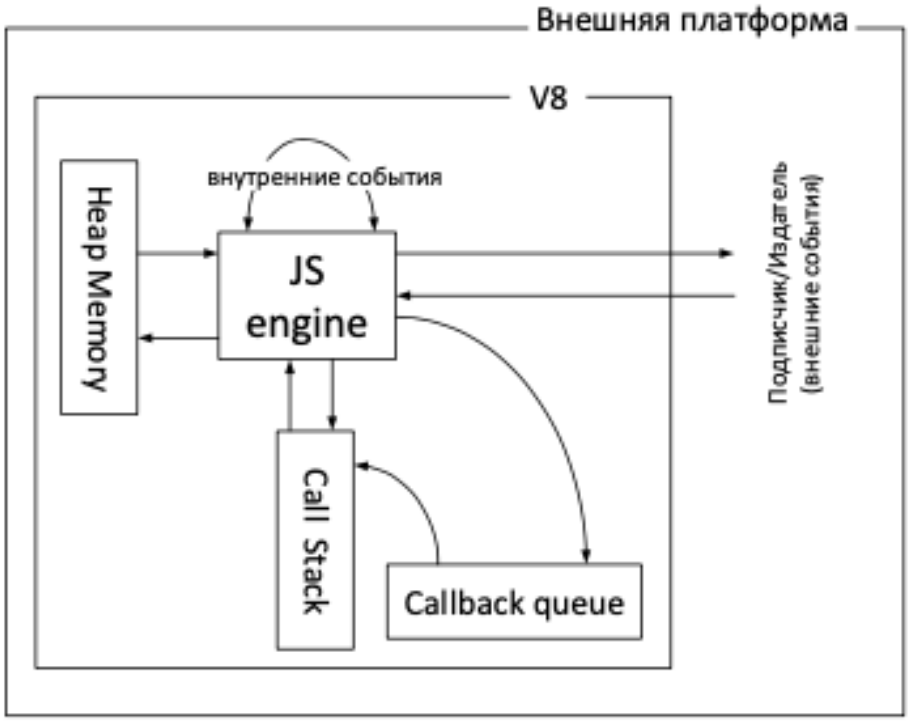
Core JS API - входит стандартный набор модулей JS
Binding - связывающий софт
LIBUV - для связи с ОС



Архитектура и принципы работы NODEJS

NODEJS -внешняя платформа. Внутри V8. Он взаимодействует с внешней платформой по принципу подписчик - издатель. Внешняя платформа имеет ряд событий, которые он может генерировать. V8 подписывается на эти события и на них реагирует. Мы разрабатываем приложение, которое интерпретируется JS Engine. Пишем код, который является обработчиком внешних событий. Стек вызовов (для хранения контекста, помещаются сюда функции) обрабатывает один поток. Heap - каждому приложению выделяется область памяти, из которой мы можем получать и возвращать. Callback queue - для того, чтобы сохранить контекст отложенной процедуры. После того, как кол стек стал пустым, выполняются функции из колбэк очереди.

Любая асинхронная операция имеет 2 фазы: оставить заявку и обработка ответа.



Вообще в Node.JS есть много встроенных модулей, например модуль по работе с файловой системой «**fs**» (Чтение файлов, Создание файлов, файлы обновления, Удалить файлы, Переименовывать файлы). И если это, такой, встроенный модуль, то «require('fs');» сработает тут же и все готово. Модуль **http**, который необходим для создания сервера (позволяет Node.js передавать данные через HTTP). Модуль **os**, который предоставляет информацию об окружении и операционной системе. **url** - для разбора URL строк

Собственным Node js модулем является любой JavaScript файл приложения, который экспортирует с помощью объекта exports функции или переменные, которые могут быть использованы другими файлами.

Модуль представляет блок кода, который может использоваться повторно в других модулях.

Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

- export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
- import позволяет импортировать функциональность из других модулей.

Модули можно разделить на 3 уровня: в core, дополнительные, созданные разработчиками.

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('Hello World!');
}).listen(8080);
```

4 Глобальные объекты Node.js (global, process) и их применение. Системные (стандартные потоки) Node.js (stdin, stdout, stderr) и их применение. Модуль console: функции log, error, dir, time, timeEnd, trace. Примеры.

Глобальные объекты - эти объекты доступны во всех модулях. В браузере он называется window, в Node.js — global, в другой среде исполнения может называться иначе.

Для примера создадим следующий модуль **greeting.js**:

```
let currentDate = new Date();
global.date = currentDate;

module.exports.getMessage = function(){
    let hour = currentDate.getHours();
    if(hour >16)
        return "Добрый вечер, " + global.name;
    else  return "Добрый день, " + name;
}
```

Здесь, во-первых, происходит установка глобальной переменной date: global.date = currentDate; Во-вторых, в модуле получаем глобальную переменную name, которая будет установлена из вне. При этом обратиться к глобальной переменной name мы можем через объект global: global.name, либо просто через имя name, так как переменная глобальная.

Определим следующий файл приложения **app.js**:

```
const greeting = require("./greeting");
global.name = "Eugene";
```

```
global.console.log(date);
console.log(greeting.getMessage());
```

Здесь устанавливаем глобальную переменную name, которую мы получаем в модуле greeting.js. И также выводим на консоль глобальную переменную date. global.console.log() = console.log().
Однако по возможности все таки рекомендуется избегать определения и использования глобальных переменных, и преимущественно ориентироваться на создание переменных, инкапсулированных в рамках отдельных модулей.

global.process: процесс, система, информация об окружении (вы можете обратиться к входным данным CLI, к переменным окружения с паролями, к памяти т.д.)

Некоторые методы:

- process.uptime(): получает время работы
- process.memoryUsage(): получает объём потребляемой памяти
- process.cwd(): получает текущую рабочую папку. Не путать с __dirname, не зависящим от места, из которого был запущен процесс.
- process.exit(): выходит из текущего процесс. К примеру, можно передать код 0 или 1.
- process.on(): прикрепляет на событие, например, `on('uncaughtException'). Наиболее часто используемые события объекта process:
    - beforeExit - инициируется, когда полностью заканчивается цикл событий;
    - disconnect - генерируется в дочернем процессе при закрытии канала IPC;
    - exit - инициируется при завершении процесса вызовом метода process.exit() или по завершению цикла событий;
    - message - может возникнуть только в главном процессе, когда в одном из дочерних процессов вызывается метод message();
    - uncaughtException - генерируется в случае возникновения необработанного исключения, но процесс при этом не завершается.

Процесс предоставляет много важных свойств, для лучшего контроля системного взаимодействия.

- Stdout — Записываемый поток в stdout.
- Stderr — Записываемый поток в stderr.
- Stdin — Записываемый поток для stdin.
  ```
  process.stdout.write("Hello World!" + "\n");
  ```
- argv — Массив, содержащий аргументы командной строки.

Первый элемент — «node», второй элемент — имя файла JavaScript.Следующие элементы — любые дополнительные аргументы командной строки. etc

Поток **stdin** обрабатывает ввод для процесса. Стандартный поток вывода **stdout** предназначен для вывода данных приложения. Наконец, стандартный поток ошибок **stderr** предназначен для вывода сообщений об ошибках.

стандартный стримы stdin, stdout, stderr создаются библиотекой libc для любой C программы автоматически; stdin и stdout являются буфризированными, а stderr - нет (на практике это означает, что запись в stdout из программы не приводит к моментальной отправке слушателю, а накапливается и только затем отправляется); при выходе из main функции или вызове функции exit, стримы stdin, stdout, stderr закрываются и буфера флашатся (данные отправляются слушателю); при аварийном завершении или вызове abort данные что были в буфере удаляются и не будут доставлены;

Модуль **console** предоставляет простую консоль для компиляции, которая подобна консольному механизму в JavaScript, предоставляемому веб-браузерами.Модуль экспортирует два компонента:

- Класс Console с такими методами, как console.log(), console.error() и console.warn(), которые могу использоваться для записи в любой стрим Node.js.
- Глобальный экземпляр console, сконфигурированный для записи в stdout и stderr.

```
const myConsole = new console.Console(out, err);
myConsole.log('hello world');
myConsole.error(new Error('Whoops, something bad happened'));
myConsole.warn(`Danger!`);
console.dir(object); -  Отображает список свойств указанного JavaScript объекта.
```

Также консоль позволяет точно замерять время, используя метод console.time() и console.timeEnd(). Расположите вызов первого из них перед кодом, время исполнения которого хотите замерить, а второго — после.

```
console.time("Execution time took");
// Some code to execute
console.timeEnd("Execution time took”);
```

console.trace (‘lalalal’) выведет стек вызовов в точке, где вызывается console.trace (). Таким образом, если вы поймаете какую-то ошибку, генерируемую каким-то более глубоким уровнем кода, console.trace () не

будет содержать этот более глубокий уровень кода в трассировке стека, так как этот код больше не находится в стеке.

5. Класс EventEmitter, назначение, применение. Пример.

EventEmitter **-** JS-класс, предоставляющий функциональность для асинхронной обработки событий в NODEJS**.** Входит в ядро NODEJS. Необходим чтобы генерировать события и уведомлять подписчиков, а также подписываться на события.

Событие в программном объекте – это процесс перехода объекта из одного состояние в другое. При этом, об этом переходе могут быть извещены другие объекты. У события есть **издатель** (или генератор) события и могут быть **подписчики** (или обработчики) события.

Для того, чтобы работать с **EventEmitter** необходимо включения двух модулей: ***events*** и ***util***.

EventEmitter: как правило, применяется в качестве базового для пользовательского объекта. Производный от **EventEmitter** объект может быть создан с помощью функции **inherits** модуля **utils**.

```
util.inherits(DB, ee.EventEmitter);
```

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

Для генерации событий предназначена функция **emit,** а для прослушивания функция **on.**

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
    console.log(a, b, this);
    // Prints: a b {}
});
myEmitter.emit('event', 'a', 'b');
```

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

Расположенная ниже диаграмма упрощённо показывает порядок выполнения операций в цикле событий. Каждая фаза имеет FIFO очередь коллбэков для выполнения.

Таймер **-** механизм, позволяющий генерировать событие или выполнить некоторое действие, через заданный промежуток времени. Позволяет сделать отложенный вызов функции.

setTimeout(), setInterval() реализованы библиотекой **libuv.**

setTimeout() - 1 аргумент - функция обратного вызова, второй - время в миллисекундах, остальное - аргументы для функции обратного вызова. Дает указание внешней среде отсчитать какое-то количество миллисекунд и уведомить о том, что время закончилось. Внешняя среда отсчитывает время и уведомляет, потом SetTimeout ставит функцию в очередь колбэков и передает ей параметры. Срабатывает 1 раз, чтобы повторно использовать необходимо опять завести ее.

setInterval() - работает похоже, но работает периодически. 1 аргумент - функция обратного вызова, второй - периодичность вызова в миллисекундах, остальное - аргументы для функции обратного вызова.

clearTimeout() - отключить таймаут навсегда.

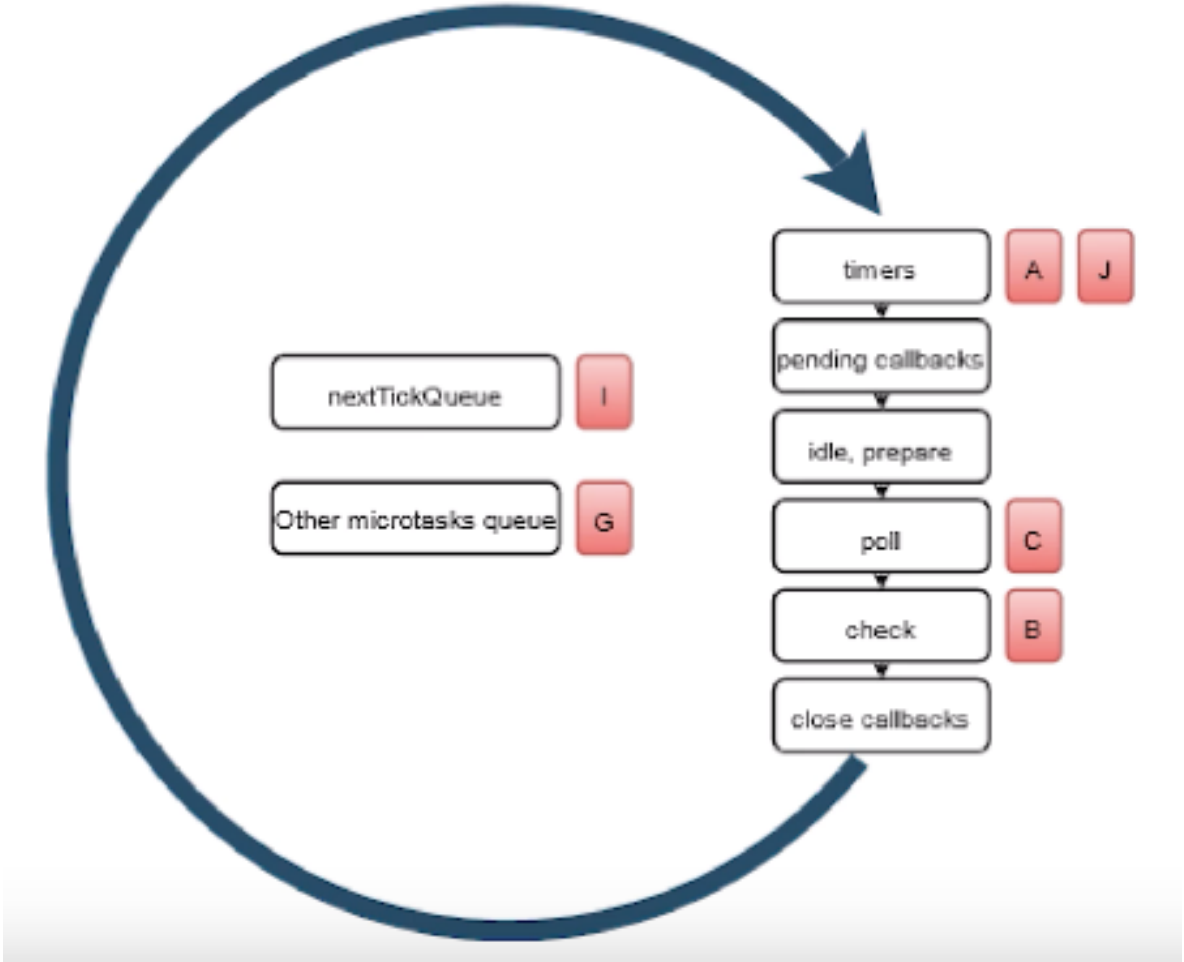
clearInterval() - удалить интервал навсегда.

Чтобы поставить в очередь колбэки существует 2 способа: (для отправки ф-ций в очередь)

- Process.nextTick - откладывает выполнение ровно на 1 цикл. Ставим выполнение в начало очереди колбэков.
- setImmediate - ставит в конец очереди колбэков.

- таймеры: в этой фазе выполняются коллбэки, запланированные setTimeout() и setInterval();
- I/O коллбэки: выполняются почти все коллбэки, за исключением событий close, таймеров и setImmediate();
- ожидание, подготовка: используется только для внутренних целей;
- опрос: получение новых событий ввода/вывода. Node.js может блокироваться на этом этапе;
- проверка: коллбэки, вызванные setImmediate(), вызываются на этом этапе;
- коллбэки события close: например, socket.on('close', ...);

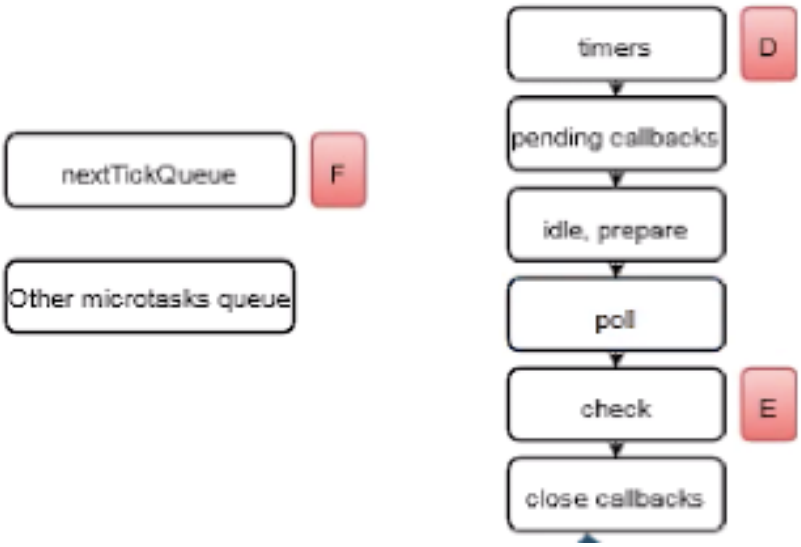
```js
const fs = require('fs');

console.log('START');

setTimeout(() => console.log('setTimeout 1'), 0);                    // (A)

setImmediate(() => console.log('setImmidiate'));                     // (B)

fs.readFile(__filename, () => {                                      // (C)
    setTimeout(() => console.log('readFile setTimeout'), 0);         // (D)
    setImmediate(() => console.log('readFile setImmediate'));        // (E)
    process.nextTick(() => console.log('readFile Next Tick'));       // (F)
});

Promise.resolve()
    .then(() => {                                                    // (G)
        console.log('Promise');
        process.nextTick(() => console.log('Promise Next Tick'));    // (H)
    });

process.nextTick(() => console.log('Next Tick'));                    // (I)

setTimeout(() => console.log('setTimeout 2'), 0);                    // (J)

console.log('END');
```



Сначала NextTick. Теперь фаза: SetTimeOut1 -> SetTimeOut2. В idle

проверяет check и nextTick. Переход в check. Пропустил poll! Конец фазы.

Следующая. Как только выполняет poll, там есть 3 функции.



Находится в poll. Выполняет nextTick. Затем в check. Конец фазы. В 3 фазу таймер.

NODEJS работает до тех пор, пока есть события, требующие обработки; если выполнить для таймера **unref** (чтобы события не учитывались как события, требующие обязательной обработки; не обращать внимание на обработку этих событий), то события, генерируемые таймером не будут учитываться при завершении работы Node.js, **ref** (обратный unref) – противоположная операция.

7 Модули и пакеты Node.js, функция require, кэширование модуля, область видимости в пакете, экспорт объектов, функций, конструкторов. Применение require для работы с json-файлами. Параметризируемый модуль. Пример.

Модуль представляет блок кода, который может использоваться повторно в других модулях. Для загрузки модулей применяется функция require(), в которую передается название модуля. В отличие от встроенных модулей для подключения своих модулей надо передать в функцию require относительный путь с именем файла (расширение файла необязательно):
const greeting = require("./greeting");

```
let currentDate = new Date();
global.date = currentDate;

module.exports.getMessage = function(){
    let hour = currentDate.getHours();
    if(hour >16)
        return "Добрый вечер, " + global.name;
```

```
        else  return "Добрый день, " + name;
    }
```

Здесь определена переменная currentDate. Однако из вне она недоступна. Она доступна только в пределах данного модуля. Чтобы какие переменные или функции модуля были доступны, необходимо определить их в объекте **module.exports**. Объект module.exports - это то, что возвращает функция require() при получении модуля. Вообще объект **module** представляет ссылку на текущий модуль, а его свойство **exports** определяет все свойства и методы модуля, которые могут быть экспортированы и использованы в других модулях. Далее изменим файл *app.js*:

```
const greeting = require("./greeting");
console.log(`Дата запроса: ${greeting.date}`);
console.log(greeting.getMessage(userName));
```

Все экспортированные методы и свойства модуля доступны по имени: greeting.date и greeting.getMessage().

Определение конструкторов и объектов в модуле

Каждый модуль может определить что угодно (конструкторы, функции), и они будут глобальными для этого модуля. Для того, чтобы сделать доступным - функция exports. Можно global.User = User;

```
function User(name){
    this.name = name;
    this.displayInfo = function(){
        console.log(`Имя: ${this.name}`);
    }
}
User.prototype.sayHi = function() {
    console.log(`Привет, меня зовут ${this.name}`);
};
```

module.exports = User; — весь модуль теперь указывает на эту функцию конструктора

```
const User = require("./user");
let eugene = new User("Eugene");
eugene.sayHi();
```

Применение require для работы с json-файлами

ru.json

```
{"Hello": "Привет”}
```

user.js

```
var phrases = require(“./ru”);
```

```
function User(name){
    this.name = name;
}
User.prototype.sayHi = function() {
    console.log(phrases.Hello + ", " + ${this.name});
};
```

module.exports = User; — весь модуль теперь указывает на эту функцию конструктора

app.js

```
const User = require("./user");
let eugene = new User("Eugene");
eugene.sayHi();
```

кэширование модуля

И теперь несколько слов о том как это все будет работать. Когда Node.JS первый раз загружает модуль, в файле «server.js»(на первой строке), он полностью создает соответствующий объект «module», с учетом «parent», «exports» и аналогичных свойств. И запоминает его у себя. «module.id», тот самый который обычно является полным путем к файлу, служит идентификатором для внутреннего кеша. Node.JS как бы запоминает файл такой то для него создан объект модуль такой то. И в следующий раз, когда мы получаем тот же файл в «user/index.js», получается, что и в «server.js» и в «user/index.js» будет использован один и тот же объект базы данных который мы запрашиваем в «require()». Соответственно прием здесь такой — первый раз когда подключается модуль(в файле «server.js») он инициализуется и мы вызываем «db.connect()».

первый require кэширует модуль, т.е повторной загрузки нет, всегда используется один и тот же экземпляр;

Параметризируемый модуль (глянуть вк)

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

Модуль «NPM» идет вместе со стандартной инсталляцией Node.JS. И в этом модуле, кроме всего прочего содержится консольная утилита которая дает доступ к громадной базе данных модулей, поддерживаемых сообществом. Модуль используемый несколькими приложениями называют ***пакетом***.

Для этого чтобы поделиться модулем не достаточно просто создать директорию с модулем, нужно еще создать для них специальный файл, описание пакета. С названием **«package.json»**. Этот файл содержит самую главную информацию о том, что за пакет здесь находится.

{

"name": "supermodule", — «name» совпадает с названием директории (должно быть уникальным!!!)

"version": "0.0.1"

}

Можно не создавать «package.json» руками а воспользоваться вызовом **«npm init»** из консоли. «npm init» спросит основные характеристики пакета и их надо будет ввести или пропустить. Итак «package.json» готов и можно опубликовать модуль в центральной базе данных. Для этого существует команда **«npm publish»** если сейчас ее вызвать то будет ошибка, потому что, чтобы ее опубликовать нужен юзер. Набирае **«npm adduser»** и создаем юзера отвечая на появляющиеся поля. Теперь вся работа с «NPM» будет от имени этого юзера. Для установки модуля это не нужно, а вот для публикации как раз очень даже важно. Вводим «npm publish», он делает запрос к базе данных и отправляет в базу модуль

Скачивает юзер его в директорию «node_modules» если ее не существует, то «NPM» её создаёт. Так происходит потому, что модули ставятся по следующему принципу. А именно «NPM» ищет директорию «node_modules» или файл «package.json» в текущей директории, если не находит, ищет на уровень выше, и выше, аж пока не найдет и поставит туда. И вот если не найдет
ни директорию «node_modules» ни файл «package.json», только тогда «NPM» сам создаст папку в текущей директории. Смысл в том, что «NPM» таким образом ищет корень пакета, чтоб разместить там нужный нам модуль и как правило именно такое поведение и является желательным.

Основные команды NPM

- npm init -> создает package.json
- npm **add**user -> создает пользователя, регистрация своего профиля в NPM
- npm **pu**blish -> публикация пакета в центральной базе данных NPM, ее так же называют репозиторием.
- npm **s**earch -> команда, для поиска пакета.
- npm **i**nstall -> поставит модуль по названию.
- npm **up**date -> обновит модуль по названию, если вызвать без имени модуля, она обновит все модули что есть.
- npm **r**emove -> удалить модуль по названию.
- npm **h**elp -> позволяет получать встроенный help npm.

Мы можем вывести список глобально установленных пакетов с помощью команды npm list с опцией --global. Такой вывод списка, со всеми зависимостями, перенасыщен. Мы можем выводить его в более читаемом виде с помощью опции --depth=0: *npm list -g --depth=0*. Стало лучше — теперь мы видим только список установленных пакетов с номерами их версий, без зависимостей.

Глобальные модули

Любой модуль можно поставить глобально если поставить флаг «-g». Чем глобальная установка отличается от обычной? В первую очередь это местом, глобальные модули ставятся в стандартную системную директорию. Второе, это то что те бинарники которые есть в свойстве «bin» в «package.json» будут поставлены по системному пути. Такие зависимости могут использоваться в функции CLI (интерфейс командной строки), но не могут быть напрямую импортированы с использованием require() в приложении Node.

Для локального пакета (устанавливается в папку с приложением) поиск осуществляется в **node_modules** по восходящему принципу. После поиска среди локальных пакетов, осуществляется поиск среди глобальных пакетов.

9 Разработка простейшего HTTP-сервера в Node.js. Извлечение данных из HTTP-запроса, формирование данных HTTP-ответа. Пример. Тестирование с помощью POSTMAN.

HTTP - формат передачи данных по TCP. Полудуплексный - клиент отправляет запрос и в конце концов должен получить ответ. Сокет - объект ОС, который предназначен для обмена данными по сети. Самый низкоуровневый пакет - HTTP. Для создания HTTP сервера.

Очередь подключений - очередь, образованная сервером, заявок на выполнение accept.

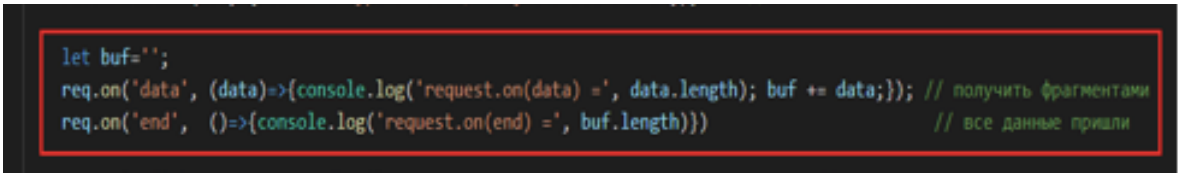
Второй параметр listen (принимает дескриптор слушающего сокета и максимальное количество одновременных соединений.) - очередь подключений. Функция accept выбирает клиентов из очереди подключений. Функция accept извлекает первый запрос из очереди ожидающих соединений, создает новый сокет, с тем же протоколом и семейством адресов что и исходный, и возвращает дескриптор файла для этого сокета.

Запрос поступает в виде битовой последовательности. NODEJS обрабатывает ее и создает 2 объекта: request и response.

keepAliveTimeout - время сохранения соединение (по умолчанию 5000). Если поступил запрос на установку соединения и в течение заданного времени оно не было установлено - отказ. **timeout** - сообщить о бездействии (умолчание - 12000). Если клиент ничего не отправляет в течение заданного времени, то соединение будет разорвано. Из сокета может быть получена та же информация, что и при работе в TCP-сервере.

```
console.log('socket.localAddress = ', socket.localAddress);
console.log('socket.llocalPort = ',   socket.localPort);
console.log('socket.remoteAddress = ', socket.remoteAddress);
console.log('socket.remoteFamily = ', socket.remoteFamily);
console.log('socket.remotePort = ', socket.remotePort);
console.log('socket.bytesWritten = ', socket.bytesWritten);
```

Длина данных ограничивается размером пакета TCP. Если отправляем много байт, то все эти байты разбиваются (кадры - дейтаграммы - пакеты). Нету гарантии того, что наши данные придут за 1 порцию. Поэтому надо предусматривать возможность того, что данные могут прийти несколькими порциями. С помощью события data. Данные передаются в формате предусмотренном объектом buffer. Описывает октетную последовательность данных.



Статические ресурсы - данные, которые представляют собой файлы, которые хранятся на сервере и отправляются клиенту.

Request

Параметр request позволяет получить информацию о запросе и представляет объект **http.IncomingMessage**. Отметим некоторые основные свойства этого объекта:

- **headers**: возвращает заголовки запроса
- **method**: тип запроса (GET, POST, DELETE, PUT)
- **url**: представляет запрошенный адрес

Например, определим следующий файл app.js:

```
var http = require("http");
http.createServer(function(request, response){
    console.log("Url: " + request.url);
    console.log("Тип запроса: " + request.method);
    console.log("User-Agent: " + request.headers["user-agent"]);
    console.log("Все заголовки");
    console.log(request.headers);
    response.end();
}).listen(3000);
```

Response

Параметр response управляет отправкой ответа и представляет объект **http.ServerResponse**. Среди его функциональности можно выделить следующие методы:

- **statusCode**: устанавливает статусный код ответа
- **statusMessage**: устанавливает сообщение, отправляемое вместе со статусным кодом
- **setHeader(name, value)**: добавляет в ответ один заголовок
- **write**: пишет в поток ответа некоторое содержимое
- **writeHead**: добавляет в ответ статусный код и набор заголовков

- **end**: сигнализирует серверу, что заголовки и тело ответа установлены, в итоге ответ отсылается клиента. Данный метод должен вызываться в каждом запросе.

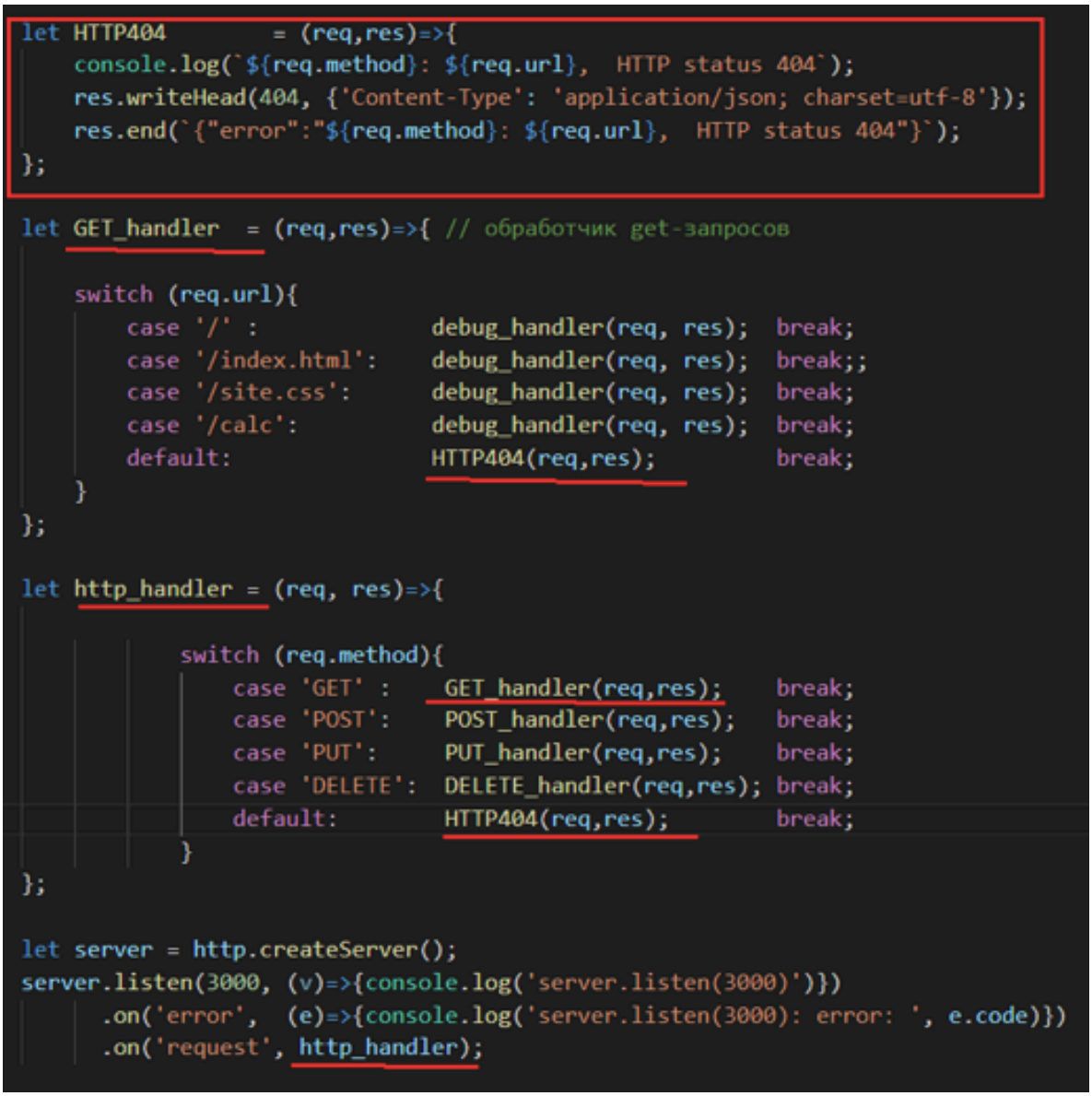
Например, изменим файл app.js следующим образом:

```
const http = require("http");
http.createServer(function(request, response){
    response.setHeader("UserId", 12);
    response.setHeader(200, {‘Content-Type’: ‘text/html; charset=utf-8’});
    response.write("<h2>hello world</h2>");
    response.end();
}).listen(3000);
```

```
var http    =  require('http');    // низкоуровневый http-сервер

let k = 0;
let c = 0;
let s = '';

let http_handler = (req, res)=>{
        console.log(`request url: ${req.url}, # `, ++k);
        res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});     //записать заголовок
        res.write('<h2>Http-сервер</h2>');                                    // отправить порцию
        s += `url = ${req.url}, request/response # ${c} - ${k}<br />`;
        res.end(s);
};

let server = http.createServer();

server.keepAliveTimeout = 10000;              // время сохранения соединения (connection), умочание  = 5000;
server.on('connection', (socket)=>{           // устанавливается новое соединение
        console.log(`connection: server.keepAliveTimeout = ${server.keepAliveTimeout} `, ++c);
        s += `<h2>connection: # ${c}</h2>`;
});

server.on('request',http_handler);

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

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

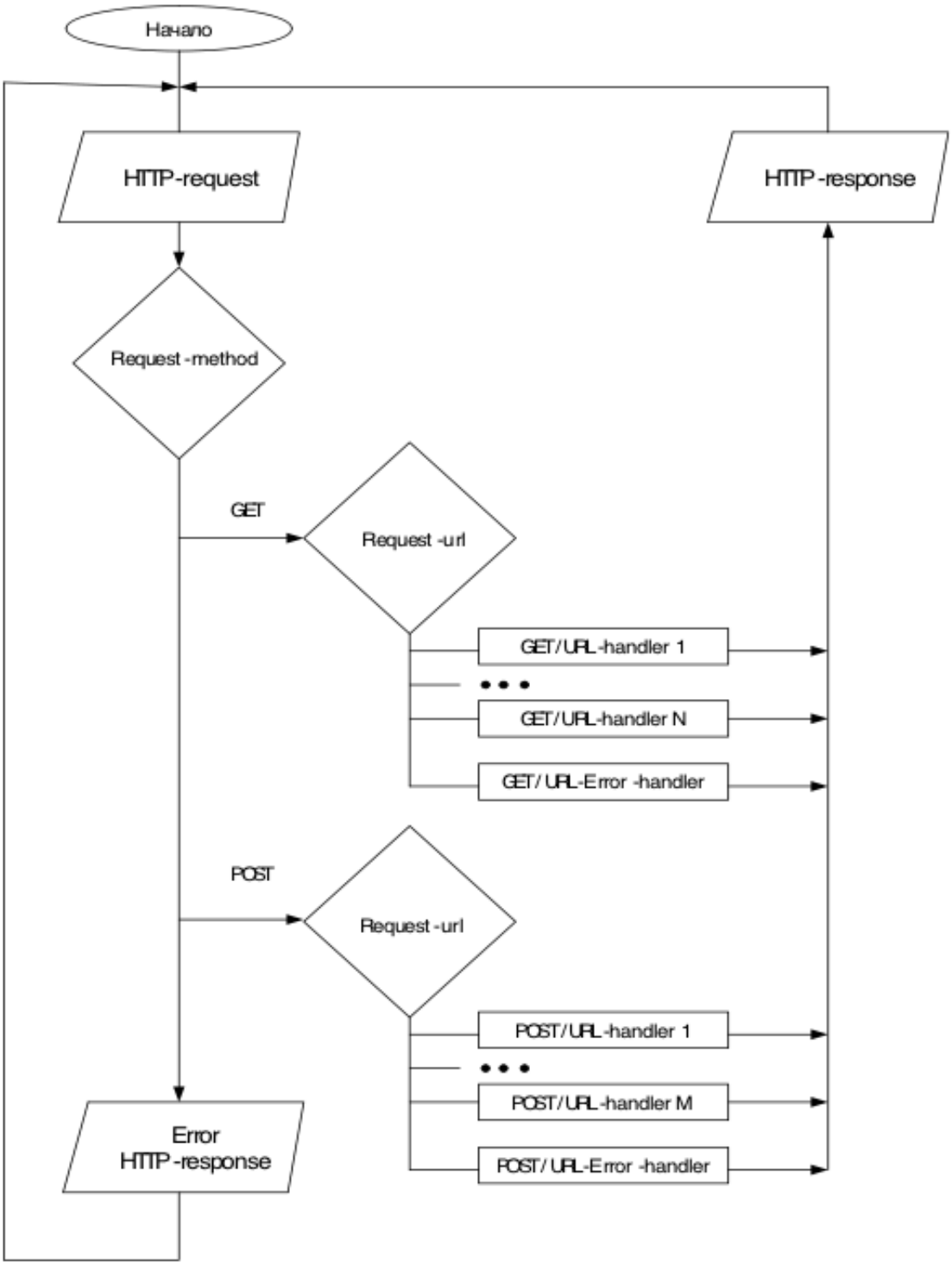
11 Разработка HTTP-сервера в Node.js. Обработка GET, POST, PUT и DELETE-запросов. Генерация ответа с кодом 404. Пример. Тестирование с помощью POSTMAN.



```
let HTTP404         = (req,res)=>{
    console.log(`${req.method}: ${req.url},  HTTP status 404`);
    res.writeHead(404, {'Content-Type': 'application/json; charset=utf-8'});
    res.end(`{"error":"${req.method}: ${req.url},  HTTP status 404"}`);
};

let GET_handler   = (req,res)=>{ // обработчик get-запросов

    switch (req.url){
        case '/' :              debug_handler(req, res);  break;
        case '/index.html':     debug_handler(req, res);  break;;
        case '/site.css':       debug_handler(req, res);  break;
        case '/calc':           debug_handler(req, res);  break;
        default:                HTTP404(req,res);         break;
    }
};

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

12 Разработка HTTP-сервера в Node.js. Обработка URI HTPP-запроса, маршрутизация запросов, генерация ответа с кодом 404. Пример. Тестирование с помощью POSTMAN.

HTTP-сервер: простейший сервер, типичный цикл работы: Сначала проверяем метод, потом url, если есть - обрабатываем, нет - ошибка.



14 Разработка HTTP-сервера в Node.js. Обработка query-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 ='';
            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-параметры</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.listen(3000)')})
        .on('error',  (e)=>{console.log('server.listen(3000): error: ', e.code)})
        .on('request', handler)
```

GET-параметры

href: /hhh/?k=3&s=kkkk&j=iii&p1=3&p2=t
path: /hhh/?k=3&s=kkkk&j=iii&p1=3&p2=t
pathname: /hhh/
search: ?k=3&s=kkkk&j=iii&p1=3&p2=t
k = 3
s = kkkk
j = iii
p1 = 3
p2 = t

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

+ decodeUrl(p.pathname...)

```
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 = `pathname: ${p.pathname}<br/>`;
                p.pathname.split('/').forEach(e => {result+=  `${e}<br/>`});
            }
            console.log(p.pathname.split('/'));
            res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
            res.write('<h1>URL-параметры</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.listen(3000)')})
        .on('error',  (e)=>{console.log('server.listen(3000): error: ', e.code)})
        .on('request', handler)
```

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

```
let http    =  require('http');
let fs      =  require('fs');
let qs      =  require('querystring');

let handler = (req, res)=>{
    if (req.method == 'GET'){
        res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
        res.end(fs.readFileSync('./07-03.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-параметры</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.listen(3000)')})
        .on('error',  (e)=>{console.log('server.listen(3000): error: ', e.code)})
        .on('request', handler)
```

```
<body>
<h1>Lec 05</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 ="reciver"  placeholder
        </div>
        <div class ="row">
            <label class="col-2">Получатель</label>  <input class="col-3" name ="sender"   placeholder
        </div>
        <div class ="row">
            <label class="col-2">Сообщение</label>   <input class="col-3" name = "message" placeholder
        </div>
        <div class ="row">
            <input type="submit" class="col-3 offset-2" value="OK"/>
        </div>
    </form>
</div>
```

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

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

браузера

```
let http    = require('http');
let fs      = require('fs');
let mp      = require('multiparty');  // npm install multiparty  // https://github.com/pillarjs/multiparty

let handler = (req, res)=>{
    if (req.method == 'GET'){
        res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
        res.end(fs.readFileSync('./07-12.html'));
    }
    else if (req.method == 'POST'){
            let result = '';
            // req.on('data', (data)=>{result+=data;})  // все внутри  multiparty
            // req.on('end',  ()=>{ });                 // ------------------
            let  form = new mp.Form({uploadDir:'./files_07-12'});
            form.on('field', (name, value)=>{
                console.log('----- field ------------------');
                console.log(name, value);
                result += `<br />---${name} = ${value}`;
            });
            form.on('file', (name, file)=>{
                console.log('----- file ------------------');
                console.log(name, file);
                result += `<br />---${name} = ${file.originalFilename}: ${file.path}`;
            });
            form.on('error', (err)=> {
                console.log('----- err ------------------');
                console.log('err =', err);
                res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
                res.write('<h1>Form/Error</h1>');
                res.end();
            });
            form.on('close', ()=> {
                console.log('----- close ------------------');
                res.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
                res.write('<h1>Form</h1>');
                res.end(result);
            });
            form.parse(req);
    }
}

let server = http.createServer();
server.listen(3000, (v)=>{console.log('server.listen(3000)')})
        .on('error',  (e)=>{console.log('server.listen(3000): error: ', e.code)})
        .on('request', handler)
```

```html
<!DOCTYPE html>
<html >
<head>
    <meta name="viewport" content="width=device-width" charset="utf-8"  />
    <title>07-10</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
</head>
<body>
<h1>Lec 06</h1>
<div style="margin: 20px; width: 800px;  padding: 5px;">
    <form method="POST" action="/"  enctype="multipart/form-data">
        <div class ="row">
            <label class="col-2">Комментарий</label> <input class="col-5" name ="comment"   type="text" />
        </div>
        <div class ="row">
            <input  class="col-5 offset-2" style="padding: 0px; border: 1px solid gray;"  name ="file"  type="file" />
        </div>
        <div class ="row">
            <input type="submit" class="col-5 offset-2" name="upload"  value="OK"/>
        </div>
    </form>
</div>
```

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

```
var http = require('http');
var url = require('url');
var fs = require('fs');

function factorial(k) {
    if (k <= 1) return 1;
    return k * factorial(k - 1);
}

http.createServer((request, response) => {
    let path = url.parse(request.url).pathname;
    if (path == '/fact') {
        console.log(request.url);
        let query = url.parse(request.url, true).query;
        if (query.k != null) {
            let k = parseInt(query.k);
            if (k == null) {
                return;
            }
            response.writeHead(200, { 'Content-type': 'application/json' });
            response.end(JSON.stringify({'k': k, 'fact': factorial(k)}));
        }
    } else if (path == '/') {
        let html = fs.readFileSync('./fact.html');
        response.writeHead(200, {'Content-type': 'text/html'});
        response.end(html);
    }
}).listen(3000);

console.log('Server running on http://localhost:3000');
```