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

**HTTP-протокол** – протокол передачи данных прикладного уровня, ассиметричный (сообщения от клиента к серверу и от сервера к клиенту разные). Всегда подразумевает пару request/response.

Относится к протоколу, который не помнит своего состояния. В запросе и ответе нет никаких ссылок на предыдущий и последующий ответ и запрос.

Каждый запрос-ответ – новый жизненный цикл HTTP (stateless протокол).

**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

**Структура HTTP-запроса.**

метод;

URI;

версия протокола (HTTP/1.1);

заголовки (пары: имя/заголовок);

параметры (пары: имя/заголовок);

расширение.

**Структура HTTP-ответа.**

версия протокола (HTTP/1.1);

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

пояснение к коду состояния;

заголовки (пары: имя/заголовок);

расширение.

**Response: Код состояния:**

* 1хх- информационные сообщения;
* 2хх – успешный ответ;
* 3хх – переадресация
* 4хх – ошибка клиента
* 5хх – ошибка сервера

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

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

**Архитектура:**

* 1 клиент – 1 сервер
* 1 клиент – несколько серверов
* Несколько клиентов – 1 сервер
* Клиенты – промежуточный сервер – сервер

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

**Асинхронность**

Операция называется асинхронной, если ее выполнение осуществляется в 2 фазы:

1. заявка на исполнение;
2. получение результата;

при этом участвуют два механизма:

A-механизм, формирующий заявку и потом получающий результат;

B-механизм, получающий заявку от A, исполняющий операцию и отправляющий результат A;

продолжительность исполнения операции B-механизмом, как правило, непредсказуемо;

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

Применение асинхронности не противоречит применению многопоточности.

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

**Асинхронный http-запрос:** запрос, при котором поток, выдавший http-запрос, не блокируется до поступления ответа; для обработки ответа применяется функция обратного вызова.

**2. Web-сервер. Ресурсы, потребляемые web-сервером. Блокирующие и неблокирующие операции ввода/вывода. Решение проблемы блокирующего ввода/вывода. Понятия конкурентность и параллельность. Закон Амдал**

**Ресурсы, потребляемые web-сервером**

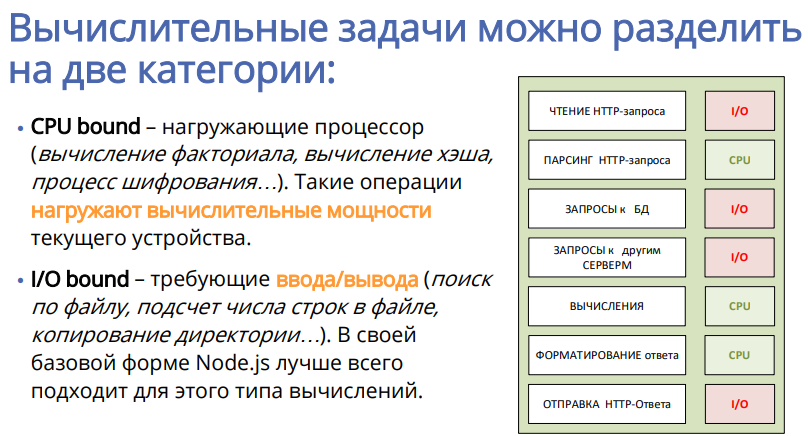
Web-сервер потребляет различные ресурсы для своей работы:

**Процессорное время (CPU):** для обработки запросов, выполнения логики приложения, шифрования/дешифрования HTTPS и т.д.

**Оперативная память (RAM):** для хранения активных соединений, кэша, сессий и данных приложения.

**Дисковое пространство:** для хранения статических файлов, логов, временных файлов и т.д.

**Сетевые ресурсы:** для передачи данных между сервером и клиентом



**Блокирующие и неблокирующие операции ввода/вывода**

**Блокирующие операции ввода/вывода** (I/O) приостанавливают выполнение программы до тех пор, пока операция не завершится. Например, чтение из файла или получение данных по сети может заблокировать поток выполнения до тех пор, пока данные не будут полностью прочитаны.

**Неблокирующие операции ввода/вывода**

позволяют программе продолжать выполнение, не дожидаясь завершения операции. Вместо этого операция возвращает управление сразу, а результат будет обработан позже, обычно через механизм обратного вызова (callback) или при помощи асинхронного программирования.

Есть

синхронный блокирующий ввод/вывод,

синхронный неблокирующий ввод/вывод,

асинхронный блокирующий ввод/вывод,

асинхронный неблокирующий ввод/вывод,

**Синхронный Ввод/Вывод**: При синхронном I/O операция ввода/вывода выполняется последовательно, и выполнение программы приостанавливается до тех пор, пока операция не завершится. Программа ждет завершения каждой операции, прежде чем перейти к следующей.

**Решение проблемы блокирующего ввода/вывода**

Есть 2 подхода для решения проблем блокирующего ввода/вывода:

1) применение многопоточности (ограничение по количеству потоков, каждый поток требует дополнительной памяти);

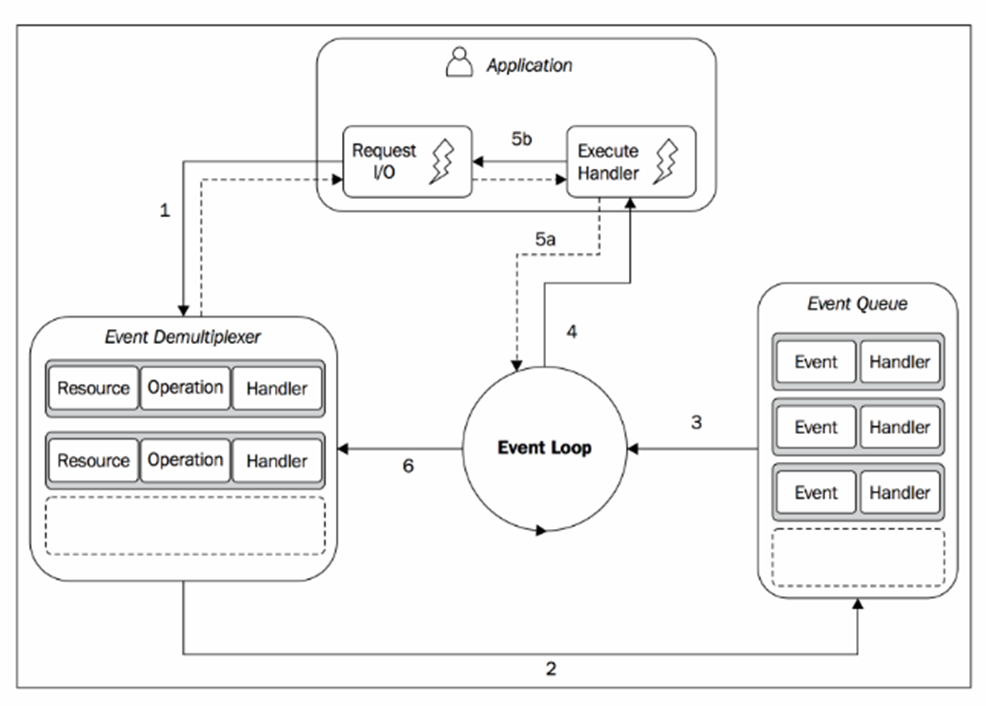
2) применение паттерна **Reactor**.

Apache – многопоточность, Nginx – Reactor

**однопоточность** (единственная задача в один момент времени, программа выполняет все свои задачи последовательно в одном потоке), **многопоточность** (различные задачи в один момент времени, программа может выполнять несколько задач параллельно, используя несколько потоков)

**паттерн Reactor** **(используется в Node.js**)– шаблон проектирования. Используется при обработке параллельных запросов к сервису. Сервисный обработчик разбирает прибывшие запросы и синхронно перенаправляет их на соответствующие обработчики запросов.

Ключевым компонентом шаблона является [**цикл событий**](https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB_%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9)**,** работающий в *одном* [потоке](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F) или [процессе](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), который [демультиплексирует](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D0%BB%D0%B5%D0%BA%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \o "Мультиплексирование) входящие запросы и отправляет их нужному обработчику запросов



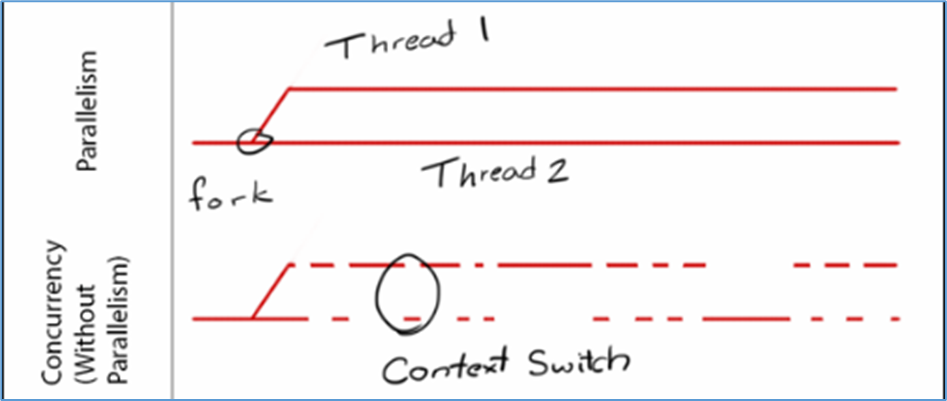
Принцип работы Reactor

1. **Регистрация событий:** Сервер регистрирует интересующие его события и соответствующие обработчики.
2. **Ожидание событий:** Reactor запускает цикл, который ожидает появления зарегистрированных событий.
3. **Обработка событий:** Когда событие происходит, Event Demultiplexer передаёт его Reactor'у, который вызывает соответствующий обработчик для обработки события.
4. **Повторение:** Процесс повторяется, и сервер остаётся в состоянии ожидания следующих событий.

**Понятия конкурентность и параллельность**

**Конкурентность (Concurrency):** Способность системы управлять выполнением нескольких задач, которые могут быть выполнены одновременно или частично перекрываться по времени. Это больше о структуре программы и управлении выполнением задач.

**Параллельность (Parallelism)**: Одновременное выполнение нескольких задач, что возможно при наличии нескольких процессоров или ядер. Это физическое выполнение нескольких операций одновременно



Более простые определения:

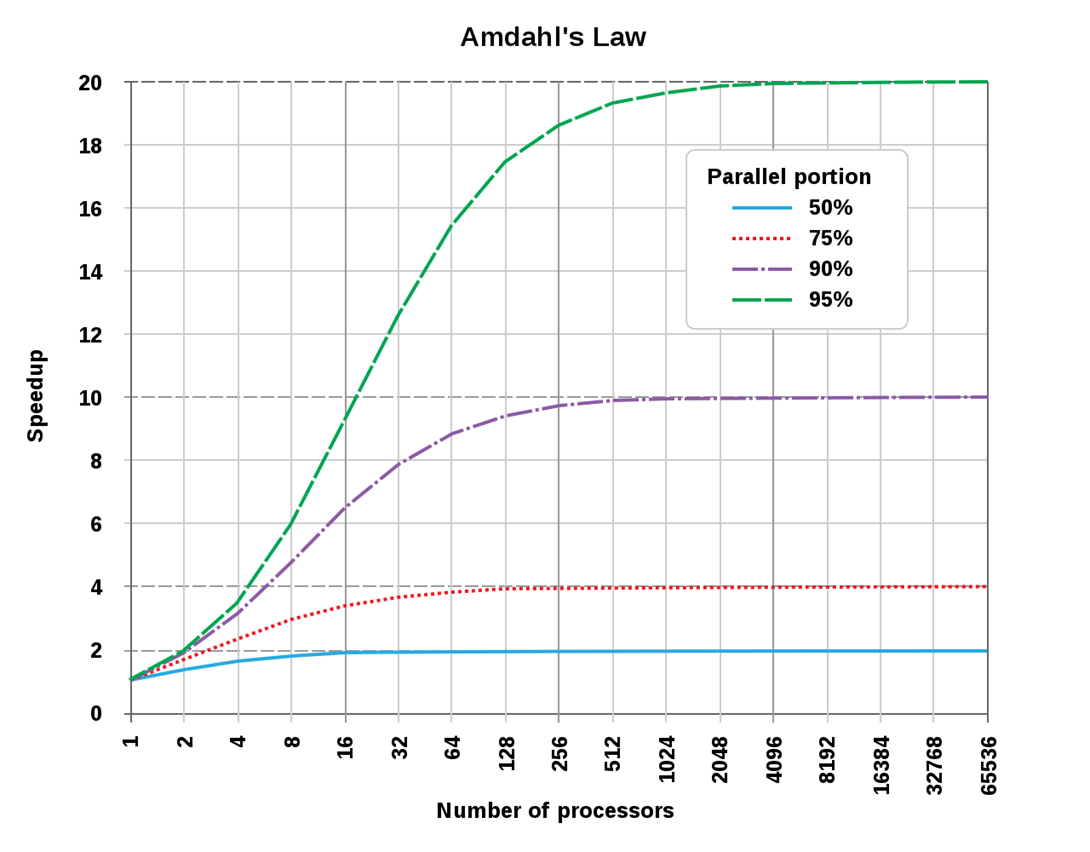
**Конкурентность** часто достигается путем чередования выполнения задач.

**Параллельность (Parallelism)** — это концепция, при которой несколько задач выполняются одновременно. Это требует наличия нескольких процессоров или ядер процессора.

**Закон Амдала**

**Закон Амдала (Amdahl's Law)** — это формула, используемая для нахождения максимального улучшения производительности системы при параллельном выполнении задач.

закон Амдала, ограниченность возможностей, speedup – кратность прироста скорости вычисления, parallel portion – степень распараллеливания алгоритма (не все можно распараллелить), number of processors – количество процессоров



Закон показывает, что даже при бесконечном увеличении числа процессоров ускорение будет ограничено последовательной частью программы. Например, если 10% программы всегда выполняются последовательно, максимальное ускорение не превысит 10 раз, независимо от количества процессоров

**Ограничения Закона Амдала**

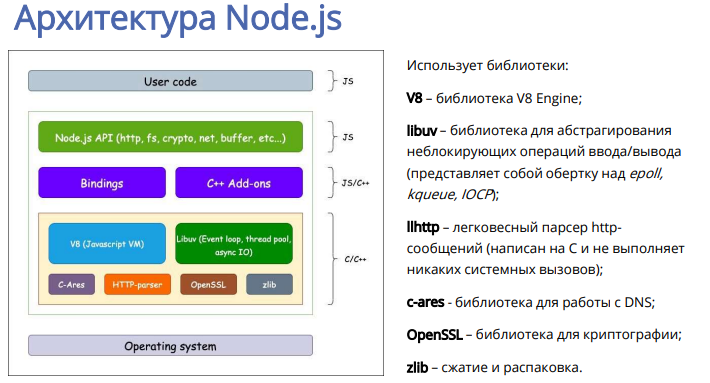
1. **Последовательная часть программы**: Даже если большая часть программы может быть параллелизована, непараллелизуемая часть станет узким местом при увеличении числа процессоров.
2. **Предел ускорения**: Закон Амдала показывает, что существует предел ускорения, которого можно достичь, даже если количество процессоров продолжает увеличиваться.

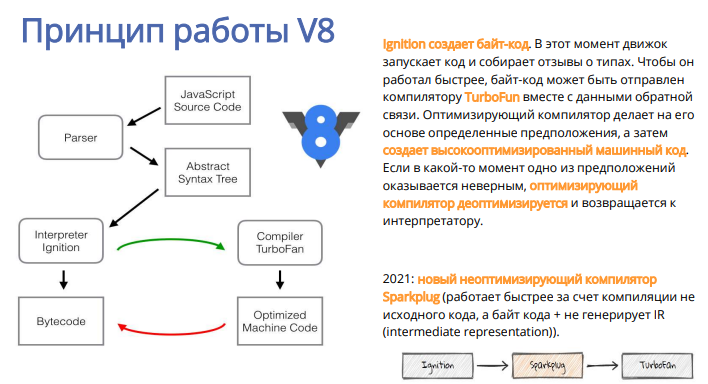
**3. Платформа Node.js, версии, назначение, основные свойства, структура, принципы работы, основные встроенные модули и их назначение, применение внешних модулей (пакетов). Форматы модулей.**

**Платформа Node.js** - программная платформа для разработки серверных web-приложений на языке JS/V8.

**основные свойства:**

* основан на **Chrome V8**;
* **среда (контейнер) исполнения** приложений на JavaScript;
* поддерживает механизм **асинхронности**;
* ориентирован на **события**;
* **однопоточный** (код приложения исполняется только в одном потоке, один стек вызовов); обычно в серверах для каждого соединения создается свой поток, в Node.js все соединения обрабатываются в одном JS-потоке;
* **не блокирует** выполнение кода при вводе/выводе (в файловой системе до 4х одновременно);
* в состав Node.js входят инструменты: **npm** – пакетный менеджер; **gyp** - Python-генератор проектов; **gtest** – Google фреймворк для тестирования С++ приложений;
* использует библиотеки: **V8** – библиотека V8 Engine, **libuv** – библиотека для абстрагирования неблокирующих операций ввода/вывода (представляет собой обертку над epoll, kqueue, IOCP); **llhttp** – легковесный парсер http-сообщений (написан на C и не выполняет никаких системных вызовов); **c-ares** библиотека для работы с DNS; **OpenSSL** – библиотека для криптографии; **zlib** – сжатие и распаковка.
* первая версия: **2009 г**.;
* стабильные версии: с **2015 г., Node.js 4.0.0;**
* основная сфера применения: **разработка web-серверов**;
* версионирование: **две ветки** 16.x.x – версии длительной поддержки (LST, Long Term Support), 18.x.x – нестабильные версии, включающие последние разработки (Current).
* документация: <https://nodejs.org/api/>





**Внутренняя структура Node.js**

Node.js

├── V8

├── libuv

├── Core Modules

│ ├── http

│ ├── fs

│ ├── path

│ ├── stream

│ ├── events

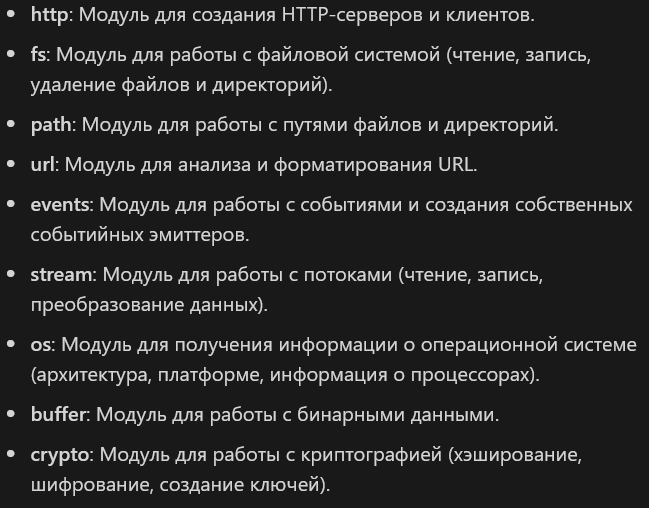
│ └── ...

├── Event Loop

├── Thread Pool

└── Add-ons

**NODEJS: встроенные** **модули (Built-in), Core modules – http, fs, console.**



**применение внешних модулей (пакетов)**

Внешние модули или пакеты в Node.js устанавливаются и управляются с помощью менеджера пакетов npm (Node Package Manager) или Yarn. Эти пакеты могут быть библиотеки, утилиты или фреймворки, которые помогают разработчикам ускорить разработку и улучшить функциональность приложений.

**Пример**

npm install express

const express = require('express');

const app = express();

app.get('/', (req, res) => {

res.send('Hello, World!');

});

app.listen(3000, () => {

console.log('Server is running on port 3000');

});

**Форматы модулей**

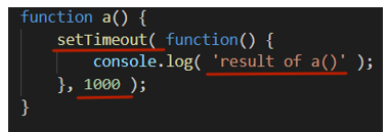
.**CommonJS**: Традиционный формат модулей в Node.js, использующий require для импорта и module.exports для экспорта.

**ES6 Modules**: Новый стандарт модулей в JavaScript, использующий import и export. Чтобы использовать ES6 модули в Node.js, необходимо добавить "type": "module" в файл package.json.

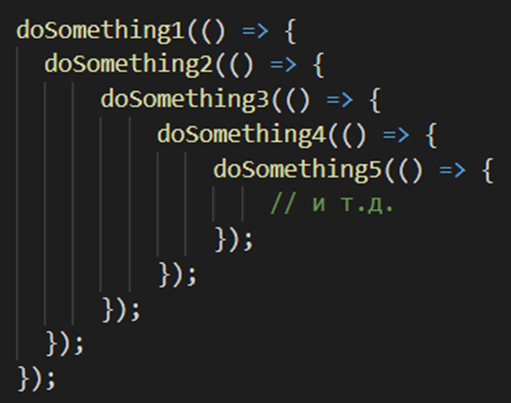
**4. Асинхронное программирование. Функция обратного вызова. Проблема «Callback hell» и способы решения.**

Функция setTimeout(callback, delay) в JavaScript — это глобальная функция, которая используется для выполнения функции обратного вызова (callback) через определенное время, заданное в миллисекундах (delay). Вот как она работает, шаг за шагом:

1. **Вызов setTimeout**: Когда вы вызываете setTimeout, передавая ей функцию обратного вызова и задержку в миллисекундах, эта функция обратного вызова не выполняется сразу. Вместо этого, setTimeout устанавливает таймер на указанное количество миллисекунд.
2. **Сохранение функции**: Функция обратного вызова временно сохраняется вместе с таймером.
3. **Истечение времени**: Как только истекает время задержки, указанное в миллисекундах, функция обратного вызова помещается в очередь коллбэков (callback queue).
4. **Event Loop**: Основная задача event loop (событийного цикла) — следить за очередью коллбэков и стеком вызовов (call stack). Он работает непрерывно, проверяя, есть ли что-то в очереди коллбэков и пуст ли стек вызовов.
5. **Перемещение в стек вызовов**: Когда стек вызовов пуст, event loop перемещает функции из очереди коллбэков в стек вызовов для их выполнения. Если стек вызовов (call stack) не пуст, когда функция обратного вызова от setTimeout готова к выполнению, то эта функция останется в очереди коллбэков (callback queue) до тех пор, пока стек вызовов не станет пустым.
6. **Выполнение функции**: После того, как функция обратного вызова перемещена в стек вызовов, она выполняется.



**Callback hell** (иногда называемый "Pyramid of Doom") — это ситуация, когда код с множеством вложенных обратных вызовов становится трудным для понимания и сопровождения. Это часто возникает, когда требуется последовательное выполнение нескольких асинхронных операций.



Есть 2 способа решить проблему:

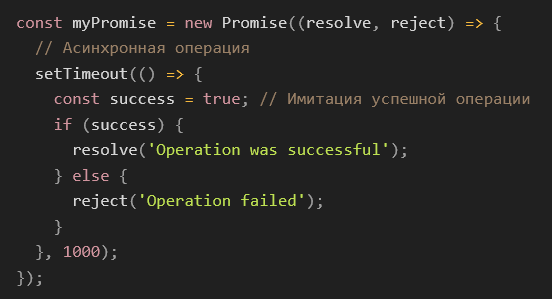
1. Разбивать на функции
2. Разбивать на модули

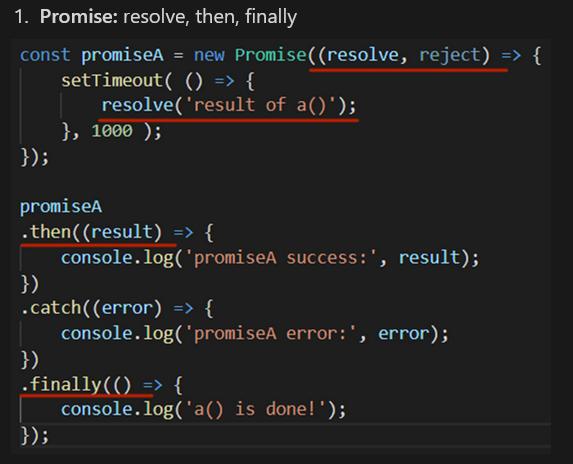
**5. Асинхронное программирование. Механизм Promises. Механизм async/await.**

**web API** – это API, которые расширяют функциональность JavaScript для выполнения асинхронных задач (DOM, AJAX, setTimeout и т.д.) + **среда неблокирующего ввода-вывода**

В контексте Node.js асинхронность достигается с помощью функций обратного вызова (callbacks), промисов (Promises) и ключевых слов async/await

**Promise** — это объект, представляющий результат асинхронной операции, которая может завершиться успешно или с ошибкой. Promise имеет три состояния:





Цепочка промисов (Promise chaining) в JavaScript — это метод работы с асинхронными операциями, при котором несколько промисов последовательно связываются друг с другом. Цепочка промисов позволяет выполнять асинхронные операции по порядку, передавая результаты одной операции в следующую.

**Возврат промисов**: Методы .then() и .catch() возвращают новый промис, что позволяет создавать цепочки промисов. Каждый последующий вызов .then() или .catch() будет работать с результатом предыдущего промиса.

**Promise.race(iterable)**

метод **Promise.race(iterable)** возвращает промис, который будет выполнен или отклонен с **результатом** исполнения **первого выполненного или отклонённого** итерируемого промиса.

**Promice.all**

метод **Promise.all(iterable)** возвращает промис, который выполнится **после выполнения всех обещаний** в передаваемом итерируемом аргументе. Возвращает новый промис, который:

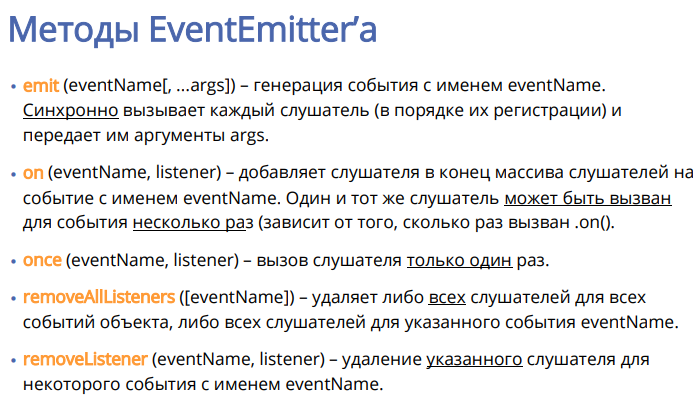
* Разрешается, если все переданные промисы разрешены, и его результатом будет массив результатов всех переданных промисов.
* Отклоняется, если хотя бы один из переданных промисов отклонен, и его результатом будет причина отклонения первого из отклоненных промисов.

**6. Класс EventEmitter, назначение, применение.**

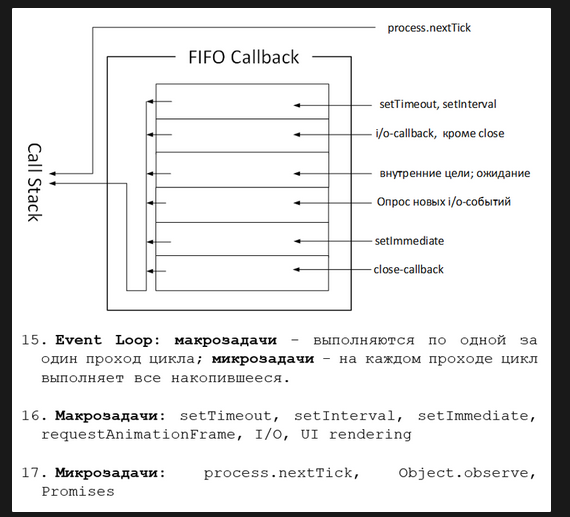
EventEmitter как правило, применяется в качестве базового для пользовательского объекта. Для наследования можно использовать ключевое слово **extends.** производный от EventEmitter объект приобретает функциональность, позволяющую генерировать и прослушивать события.

**База**

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



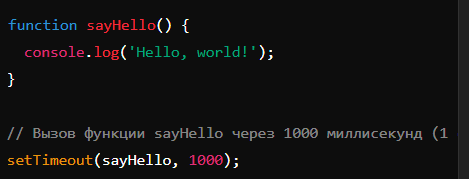
7. **Функции setTimeout, setInterval, nextTick, ref, unref, назначение, применение.**



**Таймер**: механизм, позволяющий генерировать событие или выполнить некоторое действие через заданный промежуток времени.

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

**setTimeout**(): выполняется **только один раз** через некоторый промежуток времени



сlearTimeout (timeoutObj) останавливает таймер, созданный с помощью setTimeout(). В качестве параметра передается объект ранее созданного с помощью setTimeout() таймера, который необходимо отменить

**setInterval**(): выполняется **регулярно** через некоторый промежуток времени. **clearInterval**(intervalId);

**process.nextTick** — это функция в Node.js, которая используется для добавления функции обратного вызова (callback) в очередь "next tick". Эта очередь будет выполнена после завершения текущего этапа операции, но перед любыми операциями ввода-вывода или таймерами, запланированными в следующем цикле событий.

**ref/unref**

Node.js работает до тех пор, пока не обработает все события. Логично что не для всех таймер это необходимо. Поэтому есть функции **ref()** и **unref()**

**unref** - Отвязывает таймер от цикла событий, позволяя Node.js завершить процесс, даже если таймер активен.

**ref** - Привязывает таймер к циклу событий, не позволяя Node.js завершить процесс, если таймер активен. Все таймеры создаются с привязкой (ref) по умолчанию.

Если выполнить для таймера **unref**(), то события, генерируемые таймером, не будут учитываться при завершении работы Node.js. Метод unref() есть не только у таймеров, есть еще у серверов (server.unref()), сетевых сокетов (socket.unref()) и др.

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

Модуль(лк) – текстовый файл, содержащий код на языке JS, специальным образом оформленный и размещенный. Mожет использоваться приложением. Является фундаментальной единицей структурирования кода Node.js-приложений

форматы модулей:

IIFE,

**CommonJS**,

AMD, UMD,

**ES6 Module**.

модуль используемый несколькими приложениями называют **пакетом**

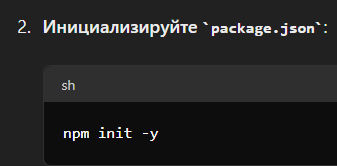
**Область видимости**

В Node.js область видимости управляется файлами и модулями. Каждый файл в Node.js считается модулем, и переменные, функции или объекты, определённые в одном модуле, не доступны другим модулям, если они не экспортированы через module.exports или exports.

**Кэширование модуля**

Когда модуль загружается с помощью require, Node.js кэширует его. Это означает, что если вы попытаетесь загрузить тот же модуль снова, Node.js вернет кэшированную версию модуля, а не загрузит и выполнит его заново. Это улучшает производительность за счёт предотвращения повторных загрузок.

Пакет — это директория, которая содержит один или несколько модулей, а также файл package.json, описывающий проект.



Параметризируемый модуль в Node.js — это модуль, который принимает параметры

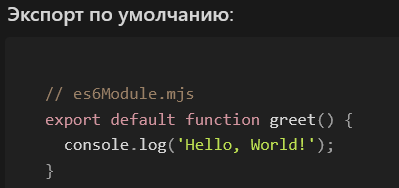


**9. Модули Node.js. Форматы модулей. Модули ES6: экспорт (по умолчанию, именованный, до/после объявления), импорт, динамический импорт.**

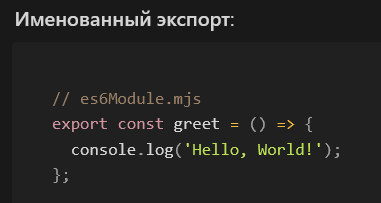
ES6: Чтобы использовать необходимо установить расширение .**mjs для файлов модулей**

или указать\*\*"type": "module"\*\* в **package.json**

Экспорт по умолчанию позволяет экспортировать один элемент из модуля.



Именованный экспорт позволяет экспортировать несколько элементов из модуля



import { add, PI } from './utils.js';

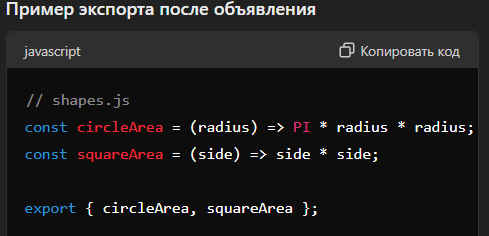
**Экспорт до объявления (Export Before Declaration)**

Можно экспортировать элементы непосредственно при их объявлении.

export const multiply = (a, b) => a \* b;

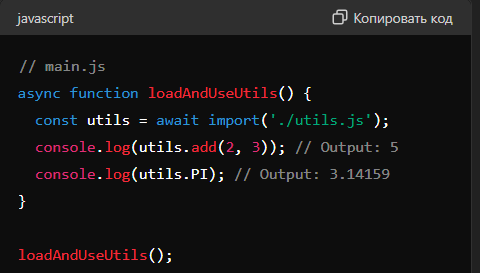
**Экспорт после объявления (Export After Declaration)**

Можно объявить элементы сначала, а затем экспортировать их.



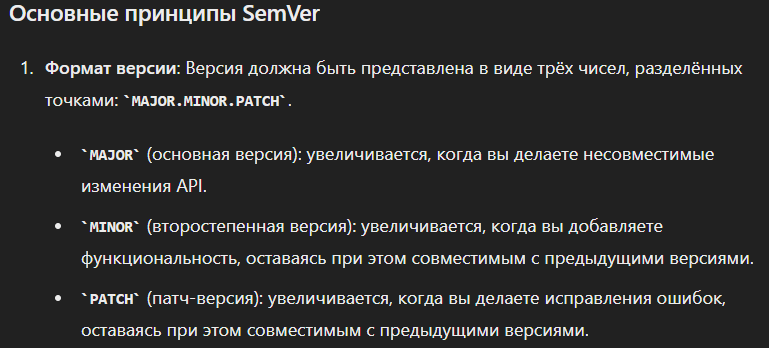


**Динамический импорт** ES6 модули также поддерживают динамический импорт с использованием функции import(). Это позволяет загружать модули асинхронно



Динамический импорт можно использовать для условной загрузки модуля.

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



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

**Шаг 1: Создание WebSocket-сервера на Node.js**

Для создания WebSocket-сервера мы будем использовать библиотеку ws.

**Установка зависимостей** npm install ws

// server.js

const WebSocket = require('ws');

const server = new **WebSocket.Server({ port: 4000, host:'localhost', path:'/' })**;

**server.on('connection', (ws)** => {

console.log('Client connected');

// Отправка ping-сообщений

const pingInterval = setInterval(() => {

**ws.ping();**

}, 30000); // каждые 30 секунд

**ws.on('pong', () => {**

**console.log('Received pong from client');**

**});**

ws.on('message', (message) => {

console.log('Received:', message);

// Обработка JSON-сообщений

try {

const data = JSON.parse(message);

console.log('Parsed JSON:', data);

// Обработка данных

**ws.send(JSON.stringify({ type: 'response', data: 'Message received' }));**

} catch (error) {

console.error('Invalid JSON:', error);

ws.send(JSON.stringify({ type: 'error', message: 'Invalid JSON' }));

}

});

**ws.on('close',** () => {

console.log('Client disconnected');

clearInterval(pingInterval);

});

ws.on('error', (error) => {

console.error('WebSocket error:', error);

});

});

console.log('WebSocket server is running on ws://localhost:8080');

**Использование потоков**: WebSocket-сообщения обрабатываются асинхронно, что позволяет эффективно управлять несколькими соединениями. Поток сообщений обрабатывается с использованием событий onmessage, onopen, onclose и onerror.

**КЛИЕНТ когда сервер отправляет ping-сообщение, клиент автоматически отвечает pong-сообщением без необходимости написания дополнительного кода для обработки этого механизма**

<script>

// создает новое WebSocket-соединение с сервером

**const ws = new WebSocket('ws://localhost:8080');**

ws.onopen = () => {

console.log('Connected to WebSocket server');

// Отправка JSON-сообщения

const message = JSON.stringify({ type: 'greeting', content: 'Hello, server!' });

ws.send(message);

};

ws.onmessage = (event) => {

console.log('Received:', event.data);

// Обработка JSON-сообщений

try {

const data = JSON.parse(event.data);

console.log('Parsed JSON:', data);

// Обработка данных

} catch (error) {

console.error('Invalid JSON:', error);

}

};

ws.onclose = () => {

console.log('Disconnected from WebSocket server');

};

ws.onerror = (error) => {

console.error('WebSocket error:', error);

};

</script>

Строка **const duplex = ws.createWebSocketStream**(wsClient, { encoding: 'utf8' }); используется для создания дуплексного (двустороннего) потока (stream) на основе WebSocket-соединения в Node.js. Давайте разберем, что это значит и как это работает.

**13. Понятие RPC. Пакет rpc-websockets: порядок разработки RPC-Websockets-сервера и клиента, работа с процедурами, генерация событий и обработка уведомлений.**

**Порядок разработки:**

1. скачиваем пакет

npm install rpc-websockets

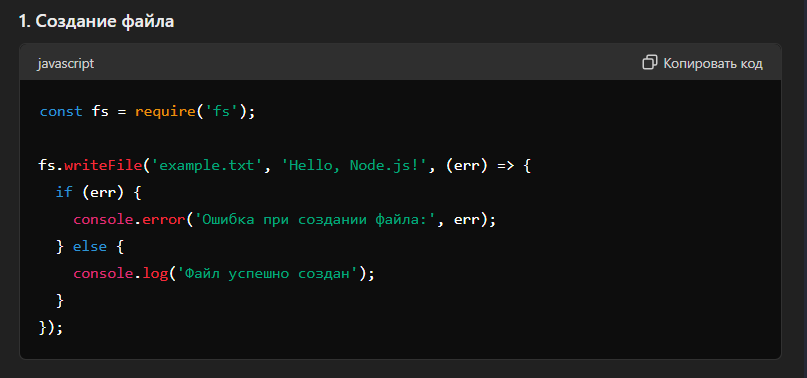
1. Самый простой сервер

|  |
| --- |
| const WebSocket = require('rpc-websockets').Server;  // Создание нового сервера на порту 8080 const server = new WebSocket({   port: 8080 });  // Регистрация RPC-процедур **server.register**('sum', (params) => {   return params[0] + params[1]; });  server.register('echo', (params) => {   return params; });  console.log('RPC-WebSocket server is running on ws://localhost:8080'); // Генерация события 'notify' setInterval(() => {   server.emit('notify', 'This is a notification'); }, 5000); |

1. Самый простой клиент

|  |
| --- |
| **const WebSocket = require('rpc-websockets').Client;**  // Подключение к серверу **const client = new WebSocket('ws://localhost:8080');**  // Вызов RPC-процедур после подключения **client.on('open'**, async () => {   try {     const resultSum = await **client.call('sum', [5, 3]);**     console.log('Sum result:', resultSum); // Output: 8      const resultEcho = await client.call('echo', 'Hello, world!');     console.log('Echo result:', resultEcho); // Output: 'Hello, world!'   } catch (error) {     console.error('RPC call error:', error);   } });  // Обработка события 'notify' client.on('notify', (message) => {   console.log('Received notification:', message); }); |

**14. Файловая система. Порядок работы с файловой системой в Node.js: создание, копирование, чтение, запись, синхронные и асинхронные операции.**



**директории**



