



Node.js

	Status	In progress
	URL	https://www.youtube.com/watch?v=jvhL2meD0_U&t=1033s

[Задачи Node.js](#)

[Зачем придумали Node.js?](#)

[Устройство Node.js](#)

[Двигок V8](#)

[Window](#)

[global.process](#)

[Библиотека LibUV](#)

[Event Loop](#)

[Работа с асинхронным кодом](#)

[Коллбеки: Error first](#)

[Промисы](#)

[Многопоточность](#)

[Что делать, если нужна математика? \(или просто большая сложность\)](#)

[Место Node.js в Яндексе](#)

[Потоки](#)

[Виды потоков](#)

[Web Streams API](#)

[Модульная система](#)

[require\(\) vs import](#)

[Три типа модулей](#)

[А что с ESM?](#)

[Управление модулями - NPM](#)

[package.json](#)

[package-lock.json](#)

[Встроенные библиотеки](#)

[Код сервера](#)

[HTTP](#)

[Запрос](#)

[Коды состояния HTTP](#)

[Главная проблема node.js](#)

[Node.js-фреймфорки](#)

[Express](#)

[Middleware](#)

[Отладка Express](#)

[Инструменты разработчика](#)

[Работа с памятью](#)

 [Рекомендованная литература](#)



Node.js - это JS на сервере

Задачи Node.js

- Доступ к данным (API)
- Серверный рендеринг (SSR)
- Утилиты (CLI)

```
//Простейший сервер
const http = require('http')
http.createServer( function(request, response){
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end('Hello, World!');
}).listen(3000);

//Запускается 'node server.js'
//Постучаться: 'curl localhost:3000 -v'
```

Более современный синтаксис

```
//То же самое, но через модули
import {createServer } from 'node:http'
createServer( function(request, response){
    response.writeHead(200, {'Content-Type': 'text/html'});
```

```
    response.end('Hello, World!');
}).listen(3001);
```

Тогда нужно поправить package.json

```
{
  "name": "IDS2023",
  "version": "1.0.0",
  "type": "module", //Добавили это
  "dependencies": {
  }
}
```

Зачем придумали Node.js?

«Классические» серверы

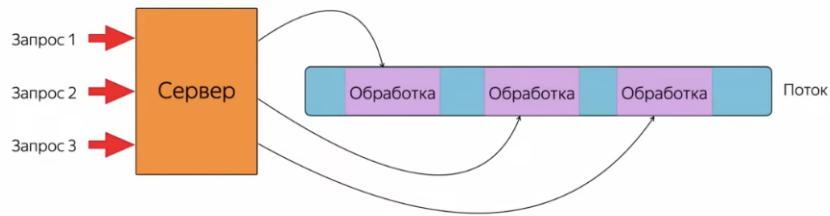


8



Идея: сделать все неблокирующим

«Однопоточный» сервер



Неблокирующий I/O

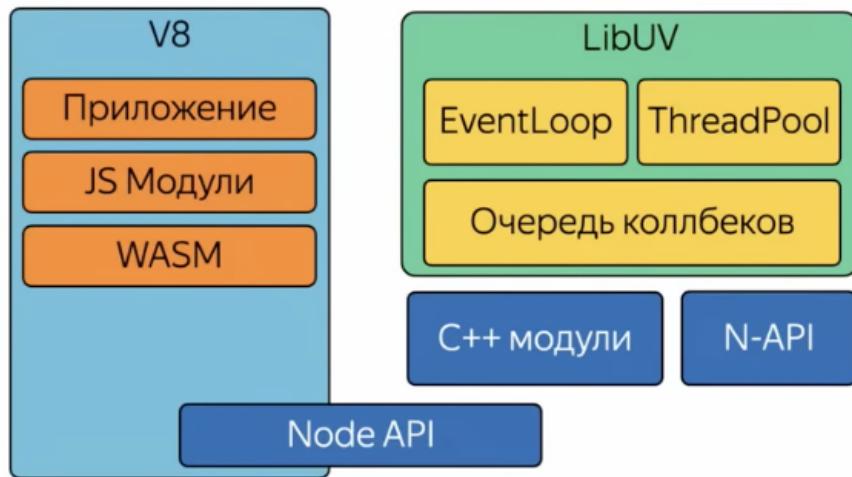
```
// Традиционный I/O
var result = db.query("select x from table_Y");
doSomethingWith(result); //ждём результатов! ←
doSomethingWithOutResult(); //выполнение заблокировано!

// Неблокирующий I/O
db.query("select x from table_Y", function(result){
  doSomethingWith(result); //ждём результатов!
});
doSomethingWithOutResult(); //выполняется без задержки!
```

- Javascript - синхронный язык. Все асинхронное приходит снаружи

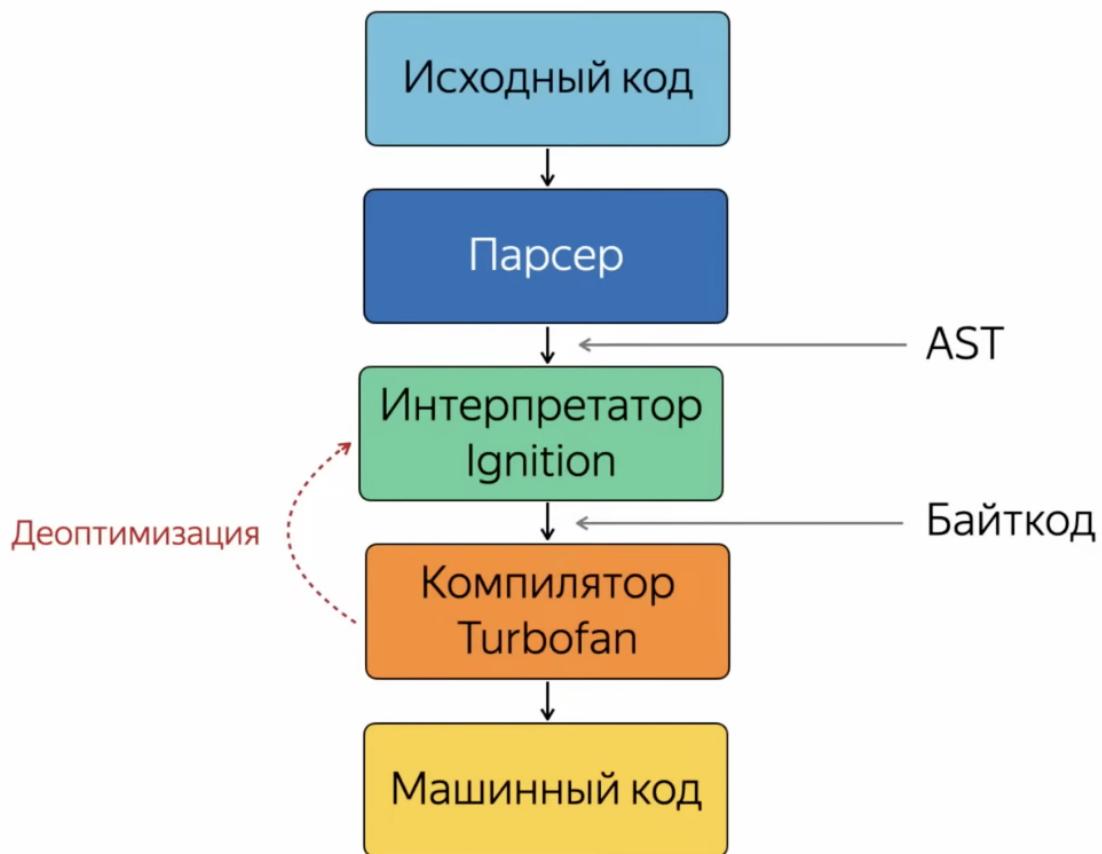
Устройство Node.js

- Node.js = V8 + LibUV



Движок V8

- Двигок хрома
- Можно поменять и на другой, но это не имеет особого смысла



Window

В node нет window. Вместо него раньше использовался global, а теперь используется общий объект - globalThis (в браузере он будет window)

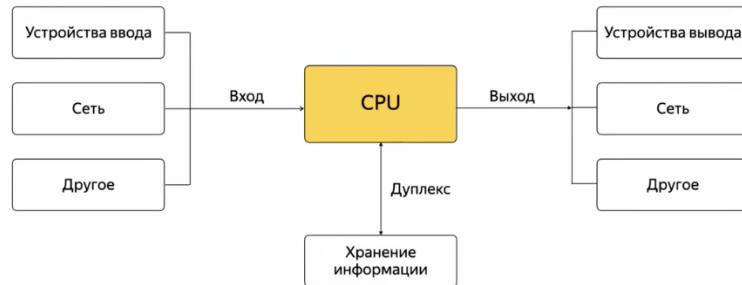
global.process

```
//в браузере этого нет
console.log(process.argv) //можно вывести параметры
console.log(process.env) //можно вывести параметры окружения
```

Библиотека LibUV

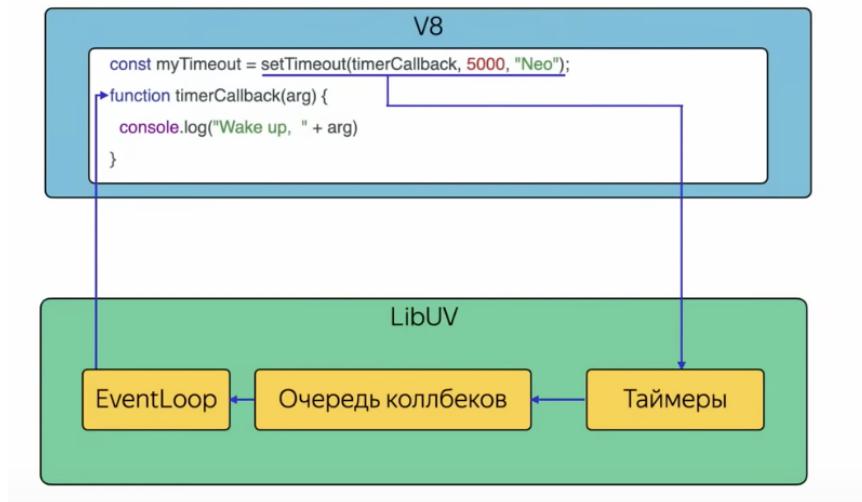
- Кроссплатформенные операции I/O

Что такое I/O?

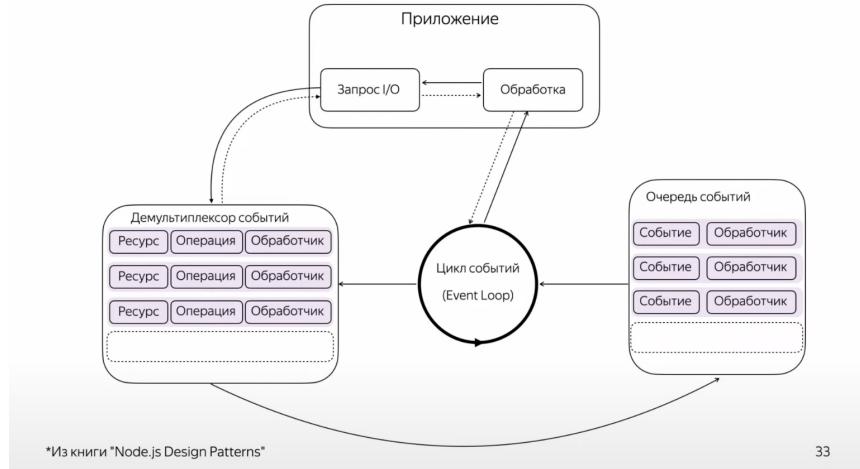


- Работа с файлами и сетью
- Event Loop - позволяет node.js выполнять неблокирующие операции I/O

Event Loop



- Паттерн “реактор”



- Подробнее об Event Loop (можно посмотреть в документации по node.js)

Event Loop



Работа с асинхронным кодом

Про это будет подробнее в лекции про асинхронность

- Коллбеки

- Промисы
- Генераторы
- async/await

Коллбеки: Error first

Сначала мы обрабатываем ошибку

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) {
    throw err;
  }
  console.log(data);
});
```

Промисы

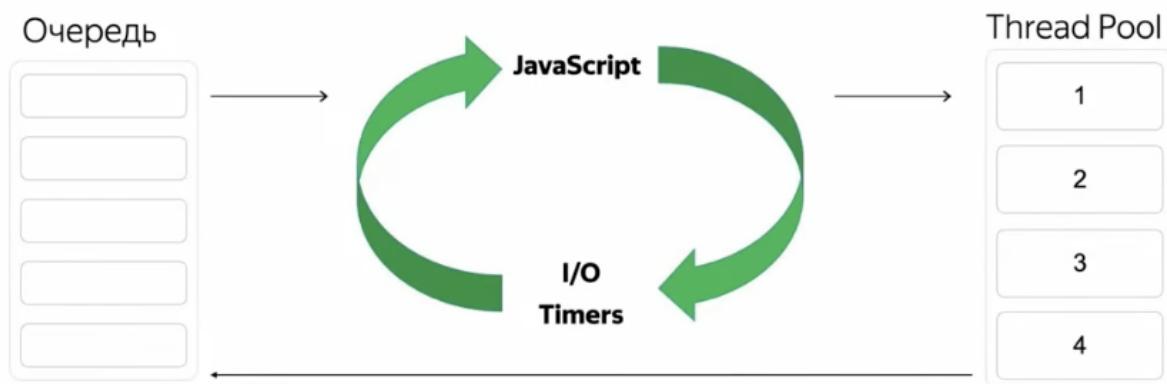
```
//Try/Catch
import { readfile } from 'node:fs/promises' // отсюда можно доставать готовое

try{
  const contents = await readfile('/etc/passwd');
  console.log(contents)
} catch (err) {
  console.error(err.message)
}
```

Многопоточность

На самом деле, потоков в node много

Event Loop



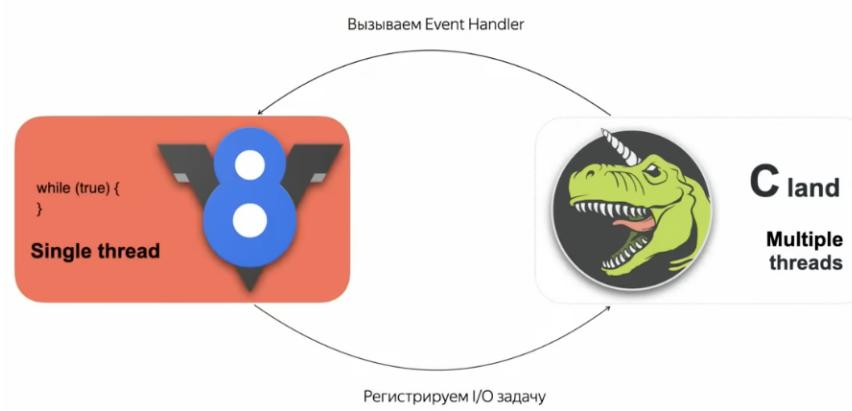
*process.env.UV_THREADPOOL_SIZE управляет размером Thread Pool

40

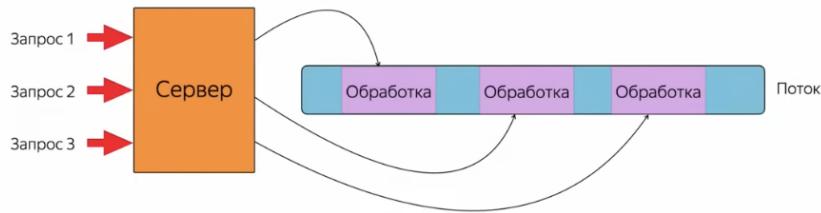


V8 - однопоточный, LibUV - многопоточный

Опасность:



Неблокирующий I/O в одном потоке

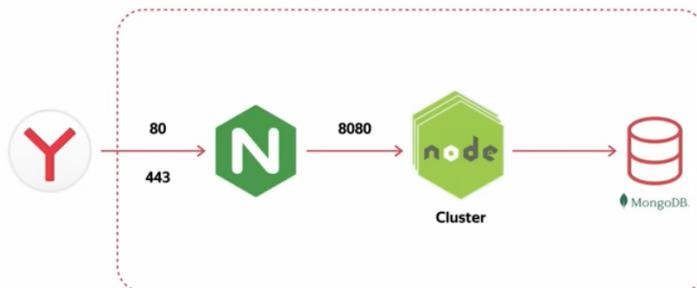


- Обработчики должны быть компактными
- Из-за этих особенностей node не используется там, где есть математика - если долго для одного пользователя, то будет долго для всех

Что делать, если нужна математика? (или просто большая сложность)

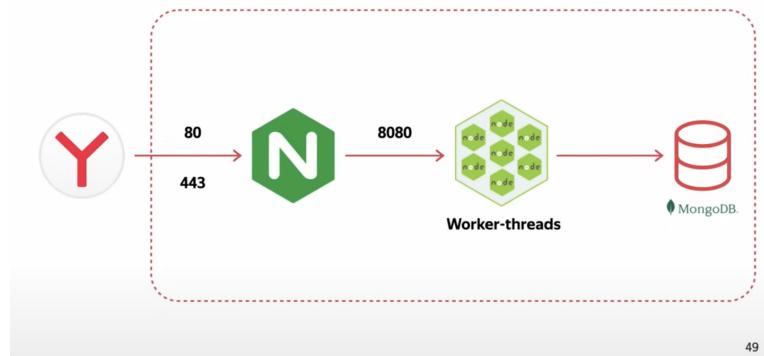
- Использовать другой язык
- Разбивать вычисления на кусочки
- Использовать несколько процессов
- Использовать worker threads

Кластер



48

Воркер-трэды

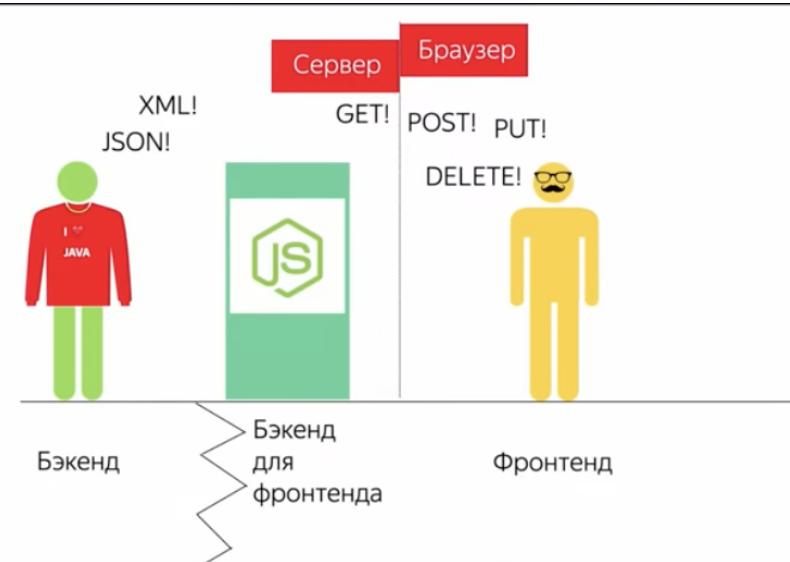


49

- То есть в целом можно писать сложные многопоточные приложения (но это будет сложнее)

Место Node.js в Яндексе

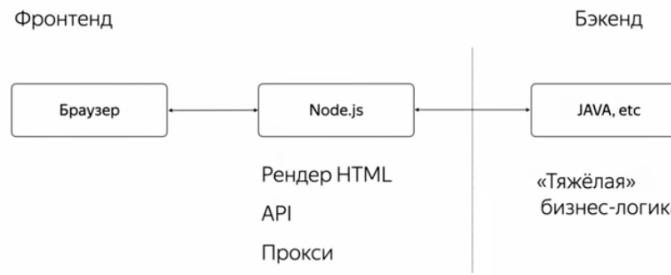
- Фронтенд или бэкенд?



- Зачем?



Где стоит применять Node.js?



Где стоит применять Node.js?



Потоки

- Потоки нужны для асинхронной работы с I/O

```
//Как потратить всю память
var http = require('http')
var fs = require('fs')

var server = http.createServer(function (req, res){
  fs.readFile('/data.txt', function (err, data){
    response.end(data)
  });
});

server.listen(8000);
```

```
//Как не потратить всю память
var http = require('http')
var fs = require('fs')

var server = http.createServer(function (req, res){
  var stream = fs.createReadStream(__dirname + '/data.txt');
  stream.pipe(response); //склеивает потоки чтения и выдачи
});

server.listen(8000);
```

Виды потоков

- На чтение (readable)
- На запись (writeable)
- Трансформирующие (transform)
- Дуплексные (duplex)

```
//Простой stream
const { Readable } = require('node:stream');
const stream = new Readable();

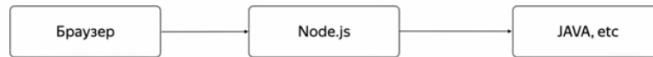
const data = `Lorem Ipsum is simply dummy text of the printing and typesetting industry. L
orem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unkno
```

```
wn printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.`.split(' ')
```

```
stream._read = () => {
  if (data.length){
    setTimeout(()=>stream.push(data.shift()+' '), 200);
  }else{
    stream.push(null)
  }
}

stream.pipe(process.stdout);
```

Прокси



Web Streams API

```
import {
  ReadableStream,
  WritableStream,
  TransformStream,
} from 'node:stream/web';
```



Основное, для чего это нужно - `fetch` (совместимый с браузером)

```
//Пример на fetch
import { TextDecoderStream } from 'node:stream/web';

async function fetchStream() {
  const response = await fetch('http://ya.ru')
  const stream = response.body;
  const textStream = stream.pipeThrough(new TextDecoderStream());
  for await (const chunk of textStream){
    console.log(chunk);
  }
}

await fetchStream();
```

Модульная система

- Сейчас все больше модулей встраивается в node, чтобы она работала из коробки
- Внешние зависимости - это уязвимости
- Пока еще нам нужны модули. Модули подключаются через package.json
- Создать его можно с помощью прм init

CommonJS

```
// module.js                                // index.js
function sum (a, b) {                      const {sum} = require("module.js");
  return a + b;                            }

module.exports.sum = sum;
```

- Каждый модуль изолирован в замыкании
- Объект exports кэшируется
- Нет возможности узнать, что экспортирует модуль до его исполнения

require() vs import

- Экспорт ESM определяется лексически (а не динамически, можно понять без исполнения)
- Импорт ESM является идемпотентным (запрещен манкипатчинг)
- Манкипатчинг - берем модуль, ставим, изменяем, и используем везде измененный

```
//Как подключать через require
require('./module.js')
require('/module.js')
require('module.js') //поиск по node_modules
```

Поиск в локально установленных модулях

```
> /home/my/projects/node_modules/module
> /home/my/node_modules/module
> /home/node_modules/module
> /node_modules/module
```

Поиск в глобально установленных модулях

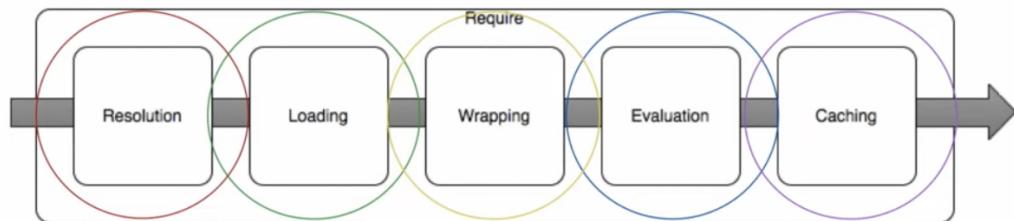
```
> $HOME/.node_modules/
> $HOME/.node_libraries
> $PREFIX/lib/node
> NODE_PATH
```

Три типа модулей

- Common.js (.js)
- JSON (.json)
- Биндинги (.node, например для подключения C++)

- Быстро, но нестабильно (что-то может не собраться при переезде на другую операционку и тд)

Загрузка CommonJS



Опасность common.js - циклические зависимости

<pre>a.js console.log('a starting'); exports.done = false; const b = require('./b.js'); console.log('in a, b.done = %j', b.done); exports.done = true; console.log('a done');</pre>	<pre>b.js console.log('b starting'); exports.done = false; const a = require('./a.js'); console.log('in b, a.done = %j', a.done); exports.done = true; console.log('b done');</pre>
<pre>main.js console.log('main starting'); const a = require('./a.js'); const b = require('./b.js'); console.log('in main, a.done = %j,' + + 'b.done = %j', a.done, b.done);</pre>	<pre>main starting a starting b starting in b, a.done = false b done in a, b.done = true a done in main, a.done = true, b.done = true</pre>

A red arrow points from the circled code in a.js to the circled code in b.js. A red arrow also points from the circled code in main.js to the output log.

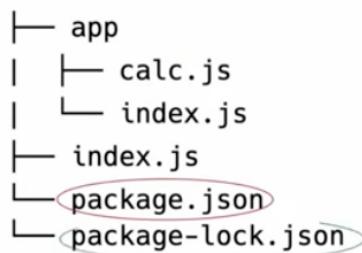
А что с ESM?

- .mjs (Майкл Джексон скрипт)
 - node понимает, что это модуль
- type: module (Полностью для приложения)
- —input-type=module
- Сейчас все переводится на ESM (без поддержки common)

Управление модулями - NPM

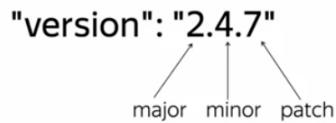
- Входит в поставку node.js
- Использует package.json для получения информации
- node package manager
- Постоянно появляются очередные “убийцы” (ya, npm...)

Типичное приложение



package.json

Семантическое версионирование — semver



- Как указывать версии
 - › `version` Должен точно соответствовать версии
 - › `>version` Должен быть выше версии
 - › `>=version`
 - › `<version`
 - › `<=version`
 - › `~version` «Приблизительно соответствует версии»
 - › `^version` «Совместим с версией»
- Установка dev-зависимостей : npm i mocha -D (пропишется в `devDependencies`)

package-lock.json

- `version` - версия модуля
- `resolved` - откуда мы его взяли
- `integrity` - хэш (защита от атак)

```
{  
  ...  
  "dependencies": {  
    "lodash": {  
      "version": "4.17.11",  
      "resolved": "https://registry.npmjs.org/lodash/-/lodash-4.17.11.tgz",  
       "integrity": "sha512-cQKh8igo5QUhZ7lg38DYWAXMvjSAKG0A8wGSVimP07SIUEK2U0+arSRKbRZWtelMtN5V0Hkwh5ry0to/SshYIg=="  
    }  
  }  
  ...  
}
```



Руками package.json лучше не править (иначе не попадет в package-lock)

```
//Поднять версию модуля  
npm version (major|minor|patch)
```



Не удалять package-lock.json

Встроенные библиотеки

- Работа с файловой системой ('node:fs')
- Работа с урл ('node:url') - можно парсить урл
- Все модули можно найти на сайте node в API

Код сервера

```
//TCP-сервер

const net = require('net')

const server = net.createServer((socket) => {
    socket.on('data', (data)=> {
        console.log(data);
        socket.write(String(data).toUpperCase());
    })
    socket.on('close', ()=>{
        console.log('connection closed')
    })
});

console.listen(8000)
```

```
//HTTP-сервер
import { createServer } from 'node:http'
const port = 3000;

//Обработчик входящего запроса
const requestHandler = (request, response) => {
    console.log(request.url);
    response.end('Hello Node.js Server');
}

const server = createServer(requestHandler);

server.listen(port, (err) => {
    if (err) {
        return console.log('something bad happened', err);
    }

    console.log(`server is listening on ${port}`);
})
```

HTTP

Запрос

- Версия HTTP протокола и HTTP метод запроса
- Заголовки
- Тело запроса (может отсутствовать)

Коды состояния HTTP

- 1xx - информационный
- 2xx - успех
- 3xx - редирект
- 4xx - ошибка клиента (неправильный запрос)
- 5xx - ошибка сервер

```
//Чуть более сложный сервер - с возвратом HTML

import { createServer } from 'node:http'
const port = 3000;

//Обработчик входящего запроса
const requestHandler = (request, response) => {
    response.statusCode = 200;
    response.setHeader("Content-Type", "text/html");
    response.write('<h1>Hello Node.js Server</h1>');
    response.end();
}

const server = createServer(requestHandler);

server.listen(port, (err) => {
    if (err) {
        return console.log('something bad happened', err);
    }

    console.log(`server is listening on ${port}`);
})
```

```
//Request Body

const {parse} = require('querystring')

...
...

if (request.method === 'POST'){
  let body = '';
  req.on( 'data', chunk => {
    body += chunk.toString();
  }

  request.on('end', ()=> {
    console.log(parse(body));
    response.end('ok')
  }
}

}
}
```

Главная проблема node.js

Все клиенты живут в одном контексте - плохо с безопасностью

- Сервер обслуживает много браузеров одновременно, и их контексты могут пересекаться ("протекание контекста")
- Из коробки есть способы избежать протекания, но они довольно хитрые - обычно использую фреймворки



Нельзя запускать сервер под sudo

Node.js-фреймфорки

- Express
- Fastify
- Nest
- Koa

Express

```
//Установка
npm install express --save
```

- Мало кода, отличающегося от node.js, поэтому это называется микрофреймворком

```
//Простой Express-сервер

const express = require('express');
const app = express();
const port = 3000;

//Роутер
app.get('/', (request, response) => {
    response.send('Hello from Express');
})

app.listen(port, (err) => {
    if (err) {
        return console.log('something bad happened');
    }
    console.log(`server is listening on ${port}`);
})
```

Middleware

Паттерн Middleware



- Express использует этот паттерн
- Разные middleware не знают друг о друге, из этого в express много проблем. В fastify этого нет
- Никак нельзя понять, какой middleware что сделал

```
import express from "express";

const app = express();
const port = 8000;

//Регистрируем middleware
app.use((request, response, next) =>{
    console.log(request.headers);
    next(); //Передает управление следующей middleware
})

app.use((request, response, next) =>{
    request.chance = Math.random(); //Дописываем новое свойство к запросу
    next();
})

app.get('/', (request, response) => {
    response.json({
```

```
        chance: request.chance //В обработчике роута доступно это поле
    })
}

app.listen(port);
```

- Обработка ошибок - тоже middleware

```
import express from "express";

const app = express();
const port = 8000;

app.get('/', (request, response) => {
    throw new Error('oops')
})

app.use((err, request, response, next) =>{
    //Логирование ошибки
    console.log(err);
    response.status(500).send('Something broke')
})

app.listen(port);
```

- Обработчик ошибок должен быть последней функцией, добавленно с помощью app.use
- Обработчик ошибок принимает колбек next. Он может использоваться для объединения нескольких обработчиков ошибок

Отладка Express

```
DEBUG=express:* node 10.errorHandling.js
```

Инструменты разработчика

- Пошаговая отладка
 - node —inspect
 - Через IDE
 - Через консоль - плохо
- Авторестарт при разработке
 - npx nodemon server.js
 - Заново запускает сервер при внесении изменений
 - В проде использовать нельзя!
 - Это старый вариант, новый: node —watch
- Autocannon
 - Проверка нагрузки приложения
- Clinic.js
 - Комплекс приложений для исследования нашего сервера

Работа с памятью

- Управление памятью
 - Ручное (например C/C++)
 - Автоматическое (например java, javascript)
- Javascript - интерпретируемый язык со сборщиком мусора.
- Если объект недостижим, он убирается



Рекомендованная литература

- “Node.js Design Patterns” (3 издание) Марио Каскиаро, Лучано Маммино
- “Многопоточный JavaScript” Томас Хантею, Брайан Инглиш
- Курсы Тимура Шемсединова