

Node JS это JavaScript runtime - процесс в оперативной памяти, который если включен то выполняет команды, движок основанный на Chrome V8 с некоторыми оболочками, которые позволяют запускать JavaScript вне браузера и работать с компьютером.

Основы

**Команда node**

Позволяет запустить js код.

node example/index

Интерпретатор Node позволяет нам не прописывать расширение .js.

**Npm**

Npm (Node Package Manager) – это менеджер, позволяющий импортировать в проект какие-либо *пакеты* (модули) – это может быть библиотека и т.д. Это удобнее, чем скачивать архивы и их распаковывать.

Команда init позволяет нам создать проект:

npm init

После этого создастся файл package.json он содержит информацию о проекте и там же npm записывает все пакеты, которые были подключены.

Команда install или короче i позволяет импортировать пакет в проект, например, установим chalk, позволяющий выводить цвет в консоль:

npm i chalk

После этого создастся папка node-modules, в котором хранятся сами пакеты, а также пакеты от которых он зависимы.

Здесь мы установили пакет, как нужный для работы самого приложения. Также есть пакеты, которые нужны лишь во время разработки, такие можно установить с флагом –D.

npm i less –D

Это деление условное и без него можно обойтись.

Со времен пакеты нужно будет удалить, это можно сделать через uninstall:

npm uninstall less

**Использование пакетов**

Некоторые пакеты имеют собственные команды. Например, tsc (компилятор typescript):

tsc index.ts

Но чаще пакеты используются прямо в js через функцию require:

const chalk = require('chalk')

console.log (chalk.green ('Hello Node JS and Chalk'))

В функцию require передается название пакета.

**Создание собственных модулей**

*Модуль* – это файл js. Это также можно использовать в другом файле через require, но только путь нужно ставить относительный.

const chalk = require('chalk')

const data = require('./data')

console.log (chalk.green (data))

Здесь мы импортируем файл data:

let text = 'Hello, World'

module.exports = text

В свойстве module.exports мы указываем, что нужно передать из функции require. В это свойство можно передать все что угодно.

**Свойства модуля**

Есть также два встроенных поля в каждом модуле. Это свойство \_\_filename, в нем полный путь файла, который сейчас исполняется и \_\_dirname путь к папке, в которой лежит исполняемый файл.

console.log (\_\_filename)

console.log (\_\_dirname)

**Глобальные свойства**

Существуют также возможно указать объект через global и сделать его глобальным, т.е. использовать во всех подключенных файлах.

Файл **data**:

let curDate = new Date()

global.date = curDate

Файл **index**:

const d1 = require ('./data')

console.log ('Hello, World', date)

Используем свойство из другого модуля, не указав сам модуль!

**Стандартная библиотека**

Отдельный вид модуля – это модуль из стандартной библиотеки. Эта часть node самая основная. Их всего 27, но основными можно назвать 4:

* path – для работы с путями
* fs – для работы с файловой системой
* os – содержит информацию об окружении и операционной системе
* events - события

Они лежат в основе Node JS, и через них реализуется все остальное.

**Path**

Методы basename, extname и dirname позволяют получить имя файла, расширение и папку соотвественно.

console.log ('Имя файла: ', path.basename (\_\_filename))

console.log ('Расширение файла: ', path.extname (\_\_filename))

console.log (Имя папки: ', path.dirname (\_\_dirname))

Метод parse вернет объект, в котором есть все части путя:

console.log (path.parse (\_\_filename))

Метод join позволяет к *пути* добавить справа *новые* *части*:

console.log (path.join (\_\_dirname, 'serve', 'index.html'))

**FS**

Для создания папки есть два метода. Первый это mkdir, который асинхронный и mkdirSync, который синхронный. Всегда следует использовать асинхронные методы, т.к. они более производительные.

У mkdir есть callback, для обработки ошибок:

const fs = require('fs')

const path = require('path')

fs.mkdir (path.join (\_\_dirname, 'test'), (err) => {

if(err) {

throw err

}

console.log ('Папка создана')

})

Если ошибок нет, то err будет равен null. У него есть подвох – нельзя создавать уже созданные папки.

Для создания файла есть метод writeFile. Он создает новый файл по *пути* и если указано его *содержимое*, то записывает его в файл. Также есть callback для обработки ошибок.

Его можно вызывать неограниченное кол-во, но притом весь контент будет стираться.

const filePath = path.join (\_\_dirname, 'test', 'text.txt')

fs.writeFile (filePath, 'Hello, Node JS', err => {

if(err) {

throw err

}

console.log ('Файл создан')

})

Если нужно просто добавить в файл используется appendFile с такими же аргументами.

fs.appendFile (filePath, 'Hello, servers', err => {

if (err) throw err

console.log ('Файл обновлен')

})

Для чтения используется метод readFile, который принимает callback уже с ошибкой и содержимым:

fs.readFile (filePath, (err, content) => {

if(err) throw err

console.log (content)

})

Тип content будет Buffer, поэтому можно воспользоваться его методом from:

let data = Buffer.from (content)

console.log (data.toString ())

Но более элегантный способ это передать *кодировку* и получить строку:

fs.readFile (filePath, 'utf-8', (err, content) => {

if(err) throw err

console.log (content)

})

**OS**

Модуль os позволяет узнать информацию о пользовательском железе. Основные методы:

Метод platform возвращает операционную систему:

console.log('Операционная система', os.platform())

Метод arch возвращает архитектуру процессора:

console.log('Архитектура процессора', os.arch())

Метод cpus возвращает список с информацией о каждом процессоре:

console.log('О процессорах', os.cpus())

Метод freemen возвращает свободную память в байтах:

console.log('Кол-во свободной памяти', os.freemem(), 'байт')

Метод totalmen возвращает кол-во всей памяти в байтах:

console.log('Кол-во всей памяти', os.totalmem(), 'байт')

Метод homedir возвращает родительскую директорию пользователя:

console.log('Домашняя директория', os.homedir())

Метод uptime возвращает время последней работы компьютера в секундах:

console.log('Включено', os.uptime(), 'сек')

**Event**

Модуль event является корневым в Node JS. В нем есть класс EventEmmiter, который реализует логику наблюдателя. Его метод on создает обработчик для события, а emitter делает событие:

const EventEmmiter = require('events')

let emmiter = new EventEmmiter ()

emmiter.on ('any', data => {

console.log ('any', data)

})

emmiter.emit ('any', {

x: 12,

y: 435

})

setTimeout (() => {

emmiter.emit ('any', {x: 43, y: 355})

}, 1200)

Получить список всех событий можно методом eventNames, кол-во обработчиков для события через метод listenerCount или сразу все обработчики – listeners.

function getInfoListener(emmiter) {

let arr = []

for(let event of emmiter.eventNames()) {

arr.push ({

event: event,

count: emmiter.listenerCount(event),

events: emmiter.listeners(event)

})

}

return arr

}

console.log (29, getInfoListener (emmiter))

**Передача аргументов**

Часто есть нужно, передать какие-либо параметры в приложение. Для этого есть свойство из process – argv.

let nodePath = process.argv[0];

let appPath = process.argv[1];

let name = process.argv[2];

let age = process.argv[3];

console.log ("nodePath: " + nodePath);

console.log ("appPath: " + appPath);

console.log ();

console.log ("name: " + name);

console.log ("age: " + age);

Вызов файла:

node index Alex 24

**Stream и pipe**

Потоки имеют много реализации, но у них всех одинаковый принцип. Существуют *readable* поток, который читает данные и *writable*, который записывает.

К примеру, в fs есть потоки для записи и чтения.

const fs = require('fs')

let writeableStream = fs.createWriteStream ("hello.txt")

writeableStream.write ("Привет мир!")

writeableStream.write ("Продолжение записи \n")

writeableStream.end ("Завершение записи")

let readableStream = fs.createReadStream("hello.txt", "utf8")

readableStream.on ("data", function (chunk) {

console.log(chunk)

})

Методом createWriteStream мы создаем поток записи, в который мы передаем путь к файлу, если файла нет, то он создастся. У потока записи методом write, записываем строку в файл. В конце нужно закрыть поток через метод end это нужно для того чтоб не произошла утечка данных.

Методом createReadStream создаем поток чтения, передавая путь и дополнительные настройки. Сам поток читает файл по кускам, генерируя событие data, поэтому мы подписываемся на это событие.

Но часто нужно сразу после прочтения переключиться на поток записи или, наоборот, к примеру, это операция копирования. Для этого в обоих стримах есть метод pipe.

const fs = require("fs")

function copyFile(source, target) {

let reader = fs.createReadStream(source),

writer = fs.createWriteStream(target);

reader.pipe (writer)

}

copyFile ('hello.txt', 'file.txt')

Это основные принципы потоков. Где еще есть потоки:

* сетевые потоки
* потоки шифрования
* потоки архивации
* ...

Делаем простой сервер

**Создание и подключение сервера**

Стандартных возможностей Node JS хватит на то, чтоб сделать мелкий сервер для простого сайта. Для этого есть встроенный модуль http. У него есть метод createServer, который возвращает объект сервер.

const http = require('http')

const process = require('process')

const PORT = process.env.PORT || 3000

http.createServer ((request, response) => {

response.end ("Hello world!")

}).listen (PORT, () => {

console.log ('Подключено', PORT)

})

У этого объекта мы вызываем метод listen, который запускает нас сервер и начинает обрабатывать переданный порт. Ради удобства обычно создается константа PORT, которая позволяет воспользоваться системным портом. В listen мы передаем callback, который вызовется, когда сервер будет запущен.

В createServer мы передаем функцию, которая будет обрабатывать подключения. В нее передаем два потока:

1. request – хранит информацию о запросе
2. response – управляет отправкой ответа

**Обработка запроса**

У request есть несколько полезных свойств:

http.createServer ((request, response) => {

console.log ('url', request.url)

console.log ('method', request.method)

console.log ('headers', request.headers)

response.end ("<h2>Hello world!</h2>")

}).listen (PORT, () => {

console.log ('Подключено', PORT)

})

Получить ответ можно просто в строке поиска в localhost подписать /index.html.

**Отправка ответа**

У response свойства для отправки ответа:

* statusCode – указать код ответа
* setHeader(*имя*, *значение*) – указывает один заголовок
* writeHead(*код*, *заголовки*) – указывает код ответа и заголовки
* write(*данные*) - устанавливает содержимое ответа

Метод end отправляет ответ на сайт.

http.createServer ((request, response) => {

response.writeHead (200, {

'Content-Type': 'text/html'

})

response.write ('<h2>Hello world!</h2>')

response.write ('<p>data...</p>')

response.end ()

}).listen (PORT, () => {

console.log ('Подключено', PORT)

})

**Маршрутизация**

В node js нет встроенной маршрутизации, для этого есть фреймворки наподобие Express. Можно самому реализовать через url:

http.createServer ((request, response) => {

response.setHeader ('Content-Type', 'text/html; charset=utf-8;');

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

response.write ('<h2>Home</h2>')

} else if (request.url == '/about') {

response.write ('<h2>About</h2>')

} else if (request.url == '/contact') {

response.write ('<h2>Contacts</h2>')

} else {

response.write ('<h2>Not found</h2>')

}

response.end ()

}).listen (PORT, () => {

console.log ('Подключено', PORT)

})

**Отправление статических файлов**

Одна из главных задач у сервера это отправка статических файлов. Для этого можно воспользоваться модулем fs и его методом readFile. В его callback мы просто вызываем response.end.

const http = require('http')

const process = require('process')

const path = require('path')

const fs = require('fs')

const PORT = process.env.PORT || 3000

http.createServer ((req, res) => {

let filePath = path.join(\_\_dirname, 'public', req.url === '/' ? 'index.html' : req.url)

const ext = path.extname(filePath)

let contentType = 'text/html'

switch (ext) {

case '.css':

contentType = 'text/css'

break

case '.js':

contentType = 'text/javascript'

break

default:

contentType = 'text/html'

}

if (!ext) {

filePath += '.html'

}

fs.readFile (filePath, (err, content) => {

if (err) {

fs.readFile (path.join (\_\_dirname, 'public', 'error.html'), (err, data) => {

if (err) {

res.writeHead (500)

res.end ('Error')

} else {

res.writeHead (200, {

'Content-Type': 'text/html'

})

res.end (data)

}

})

} else {

res.writeHead (200, {

'Content-Type': contentType

})

res.end (content)

}

})

}).listen (PORT, () => {

console.log ('Подключено', PORT)

})

Здесь код рассчитан на то, что все статические файлы хранятся в public, поэтому получая url запроса, мы к имени добавляем public.

Но файлы бывают разных типов и поэтому нужно применить конкретное значение для Content-Type. Далее нужно добавить расширение html для html файлов.

Потом мы читаем файл. Если файл не был найден, то мы загрузим error.html, а если даже его не получилось найти, то это означает, что сервера «легли» и мы возвращаем ответ со статусом 500.

Если файл был найден, то мы просто его отправим.

**Итого**

Мы сделали сервер, хоть он и неполноценный, но он справляется с тем, что от него требуется. Все это можно заменить фреймворком Express, о котором будет рассказано в следующей главе.

Этот фреймворк работает поверх модуля http, и он реализует за нас следующее:

1. отправка статических файлов
2. работа с куки
3. обработка форм
4. CORS
5. маршрутизация
6. …