**Node.js #5 Модуль событий (Event Module)**

**5-1app.js**

//Событие - это сигнал, который сообщает о том, что что-то произошло

***const EventEmitter = require("events");*** //импортируем модуль событий в наш файл. Имя с большой буквы - значит это КЛАСС, на основании которого будем создавать экземпляры для отслеживания различных событий

***const emitter = new EventEmitter();*** //добавили константу, которая является экземпляром события в нашем модуле. Теперь мы можем создать какое-то свое произвольное событие, вызвать его и отреагировать на это

***// emitter.on('some\_event', (text) => {***

***// console.log(text);***

***// })*** //1й шаг - метод ***.on*** создает слежение за событием. Аргументы:('имя события за которым будем следить - это произвольная строка', 2й аргумент - ***callback*** функция внутрь которой в качестве аргумента мы будем передавать некий текст и отображать его в консоли())

***emitter.on("some\_event", (args) => {***

***const { id, text } = args;*** // выполним деструктуризацию значений из ***args***

***console.log(id, text);***

***});*** //метод ***.on*** в котором 2й аргумент - ***callback*** функция внутрь которой в качестве аргумента мы будем передавать объект ***args***, предварительно проведя его деструктуризацию, и выведя его на консоль

***// emitter.emit('some\_event', 'Event test text!');*** //2й шаг - метод .***emit*** - непосредственный вызов события, которое мы создали. В него передаем аргументы: 1 - имя события, 2й - данные (в нашем случае текст, который мы выведем в консоль). Но проще и лучше в качестве 2-го аргумента использовать объект. Тогда метод примет следующий вид:

***emitter.emit("some\_event", { id: 1, text: "Event test text!" });***

// 3:29 - ЛОГИРОВАНИЕ ПОЛЬЗОВАТЕЛЯ

// создаем файл 5-2log.js ( см. ---> строка 1) и 5-2арр

**5-2log.js**

// здесь у нас происходит непосредственный вызов событий, за которыми мы следим

***const EventEmitter = require("events");*** //импортируем модуль событий в наш файл. Имя с большой буквы - значит это КЛАСС, на основании которого будем создавать экземпляры для отслеживания различных событий

//создаем класс ***Logger*** и наследуем его от ***EventEmitter***:

***class Logger extends EventEmitter {***

***log = (msg) => {***

***console.log(msg);***

***this.emit("some\_event", { id: 1, text: "Event test text!" });*** //т.к. мы находимся внутри класса, который наследует от ***EventEmitter***, поэтому можем обратиться к его унаследованному методу .***emit*** через ***this***

***};***

***}*** //создали класс-наследник, который наследуется от ***EventEmitter***, тем самым получая все методы родителя, расширили его кастомными методами, после чего для создания и слежения за событиями используются экземпляры данного класса-потомка

***module.exports = Logger;*** //создали экспорт данного класса

// 2-й ВАРИАНТ

//(не рекомендован документацией ноды, но работает)

// ***const EventEmitter = require('events');***

// ***const util = require('util');*** //вариант с использованием модуля ***util***. Для этого сначала импортируем его в файл

***// class Logger {***

***// log = (msg) => {***

***// console.log(msg);***

***// this.emit('some\_event', { id: 1, text: 'Event test text!' });***

***// }***

***// }*** //создали класс ***Logger*** c одним методом ***log***

***// util.inherits(Logger, EventEmitter);*** //используя метод .***inherits*** расширяем данный класс классом ***EventEmitter***, т.е. по сути делаем тоже самое наследование

***// module.exports = Logger;***

**5-2app.js**

// здесь у нас описаны события за которыми мы следим

***const Logger = require("./5-2log");*** //импортировали класс ***Logger*** из файла ***5-2log.js***

***const logger = new Logger();*** //создали экземпляр ***logger*** от импортированного класса. В качестве создания события и его вызова используем именно этот экземпляр

***logger.on("some\_event", (args) => {***

***const { id, text } = args;***

***console.log(id, text);***

***});*** //метод .***on*** в котором 2й аргумент - ***callback*** функция внутрь которой в качестве аргумента мы будем передавать объект ***args***, предварительно проведя его деструктуризацию, и выведя его на консоль

***logger.log("User Logged!");*** //вызываем функцию ***log*** в которую передаем текст сообщения

**Node.js #6 Буфер и потоки (Buffer & Streams)**

**1 - ЧИТАЮЩИЙ ПОТОК - Readable**

***const fs = require('fs');*** //подключаем модуль файловой системы

***const readStream = fs.createReadStream('./6docs/text.txt');*** //создаем ЧИТАЮЩИЙ поток, присваиваем ему работу метода .***createReadStream*** в который передаем путь к файлу

***readStream.on('data', (chunk) => {***

***console.log('-------');*** //добавили разделитель ------- между порциями для наглядности

***console.log(chunk);*** //***console.log(chunk.toString());*** для вывода в консоли в виде текста, а не буфера

***})*** //методом .***on*** включаем чтение потока - внутрь него передаем 2 аргумента - событие ***'data'*** и ***callback***, который будет принимать порции данных (***chunk***)

**2 - ЗАПИСЫВАЮЩИЙ ПОТОК - Writable - 03:24**

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

***const readStream = fs.createReadStream('./6docs/text.txt');***

***const writeStream = fs.createWriteStream('./6docs/new-text.txt');*** //создаем константу пишущего потока используя метод .***createWriteStream*** в качестве аргумента передаем путь и имя файла, который создаем

***readStream.on('data', (chunk) => {***

***writeStream.write('\n---CHUNK START---\n'); //добавляем метки в начале и в конце куска***

***writeStream.write(chunk); //добавляем записывающий поток внутрь читающего***

***writeStream.write('\n---END CHUNK---\n');***

***})*** //сначала будет происходить чтение большого файла по кускам, и каждый прочитанный кусок мы будем передавать записывающим потоком в новый файл

**3 - ДУПЛЕКСНЫЙ ПОТОК - Duplex - 04:45**

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

***const readStream = fs.createReadStream('./6docs/text.txt');***

***const writeStream = fs.createWriteStream('./6docs/new-text.txt')***

//улучшение логики передачи данных - создаем функцию ***handleError***

***const handleError = () => {*** //внутри данной функции

***console.log('Error');*** // вывели в консоль сообщение об ошибке

***readStream.destroy();*** // если в момент чтения данных происходит ошибка, то с помощью метода .***destroy*** будем уничтожать читающий поток, т.к. поврежденные данные нам не нужны

***writeStream.end('Finished whis error...');*** // а в записывающий поток добавим информацию, что при чтении произошла ошибка и сохраненный файл записан с ошибкой. Метод .***end*** добавит нужную строку в конце

***}***

// функцию ***handleError*** создали, теперь нужно ее добавить МЕЖДУ операциями ЧТЕНИЯ и ЗАПИСИ. Для этого мы используем цепочку методов: на ***readStream*** вешаем метод .***on***, который будет слушать ошибки чтения и вызывать функцию ***handleError*** и аналогичные действия мы делаем после .***pipe***, чтобы если при записи произойдет ошибка мы также о ней знали и уничтожали дальнейшее чтение и попытки записи.

***readStream***

***.on('error', handleError)***

***.pipe(writeStream)***

***.on('error', handleError);***

***readStream.pipe(writeStream);*** //метод .***pipe*** осуществляет чтение получаемых данных из ***readStream*** и передает их напрямую в ***writeStream***.

//Т.е. мы берем поток чтения и используем метод .***pipe*** внутрь которого передаем поток записи

**4 - ТРАНСФОРМИРУЮЩИЙ или ПРЕОБРАЗУЮЩИЙ ПОТОК - Transform - 06:36**

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

***const zlib = require('zlib');*** // подключаем модуль сжатия ***zlib***

***const readStream = fs.createReadStream('./6docs/text.txt');***

***const writeStream = fs.createWriteStream('./6docs/new-text.zip');***

***const compressStream = zlib.createGzip();*** //создаем преобразующий (в нашем случае - сжимающий) поток и вызываем метод .***createGzip***

***function handleError() {***

***console.log('Error');***

***readStream.destroy();***

***writeStream.end('Finished whis error...');***

***}***

***readStream***

***.on('error', handleError)***

***.pipe(compressStream)*** // добавили преобразующий поток в общий поток. Т.о. данные будут сжиматься и записываться в файл

***.pipe(writeStream)***

***.on('error', handleError);***