

Введение в Node JS

Модуль 1 (5 пар)

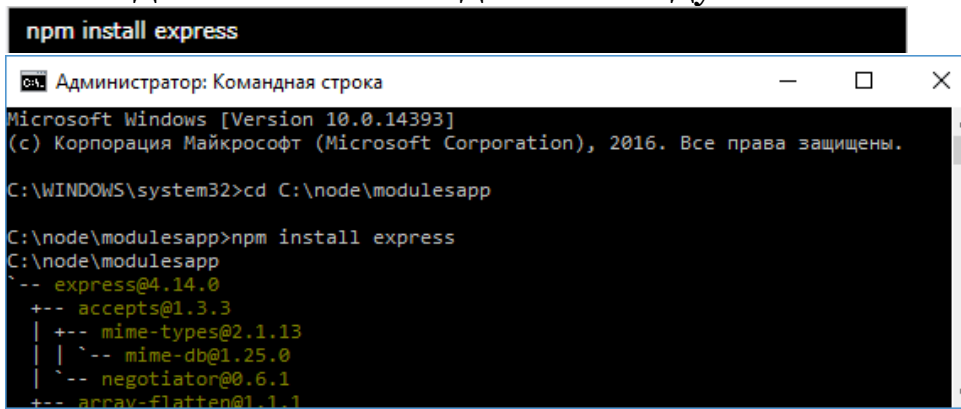
Файл package.json

- Для более удобного управления конфигурацией и пакетами приложения в npm применяется файл конфигурации package.json. Так, добавим в папку проекта modulesapp новый файл package.json:

```
{
  "name": "modulesapp",
  "version": "1.0.0"
}
```

- Здесь определены только две секции: имя проекта - modulesapp и его версия - 1.0.0. Это минимально необходимое определение файла package.json. Данный файл может включать гораздо больше секций. Подробнее можно посмотреть в документации.
- Далее для примера установим в проект express. Express представляет легковесный веб-фреймворк для упрощения работы с Node.js. В данном случае мы не будем пока подробно рассматривать фреймворк Express, так как это отдельная большая тема. А используем его лишь для того, чтобы понять, как устанавливаются сторонние модули в проект.
- Для установки функциональности Express в проект вначале перейдем к папке проекта с помощью команды cd. Затем введем команду

```
npm install express
```



```
Администратор: Командная строка
Microsoft Windows [Version 10.0.14393]
(c) Корпорация Майкрософт (Microsoft Corporation), 2016. Все права защищены.

C:\WINDOWS\system32>cd C:\node\modulesapp

C:\node\modulesapp>npm install express
C:\node\modulesapp
-- express@4.14.0
+-- accepts@1.3.3
| +-- mime-types@2.1.13
| | -- mime-db@1.25.0
| -- negotiator@0.6.1
+-- array-flatten@1.1.1
```

Файл package.json

- После установки express в папке проекта modulesapp появится подпапка node_modules, в которой будут храниться все установленные внешние модули. В частности, в подкаталоге node_modules/express будут располагаться файлы фреймворка Express.
- И после выполнения команды, если мы откроем файл package.json, то мы увидим информацию о пакете:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

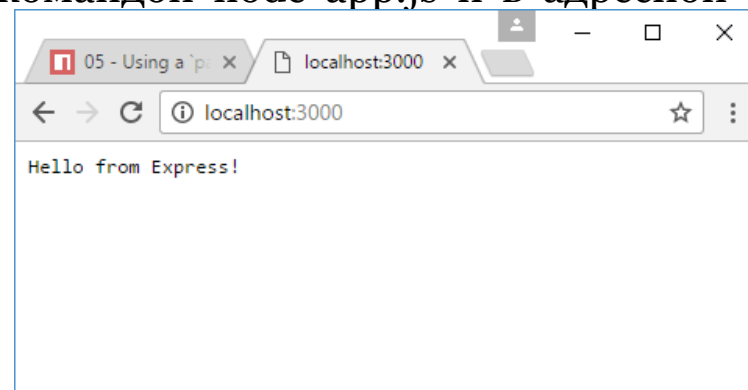
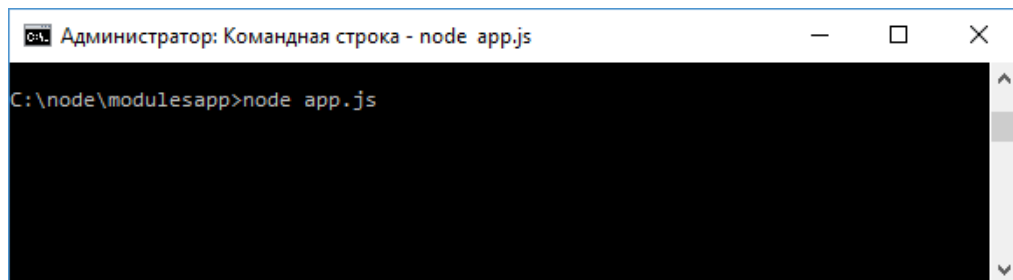
- Информация обо всех добавляемых пакетах, которые используются при работе приложения, добавляется в секцию dependencies.

Файл package.json

- Используем добавленный пакет express и для этого определим файл простейшего сервера. Для этого в корневую папку проекта modulesapp добавим новый файл app.js:

```
// получаем модуль Express
const express = require("express");
// создаем приложение
const app = express();
// устанавливаем обработчик для маршрута "/"
app.get("/", function(request, response){
  response.end("Hello from Express!");
});
// начинаем прослушивание подключений на 3000 порту
app.listen(3000);
```

- Первая строка получает установленный модуль express, а вторая создает объект приложения. В Express мы можем связать обработку запросов с определенными маршрутами. Например, "/" - представляет главную страницу или корневой маршрут. Для обработки запроса вызывается функция app.get(). Первый параметр функции - маршрут, а второй - функция, которая будет обрабатывать запрос по этому маршруту.
- И чтобы сервер начал прослушивать подключения, надо вызвать метод app.listen(), в который передается номер порта. Запустим сервер командой node app.js и в адресной строке браузера введем адрес *http://localhost:3000/*



Добавление множества пакетов

- Файл `package.json` играет большую роль и может облегчить работу с пакетами в различных ситуациях. Например, мы планируем использовать множество пакетов. Но вводить для установки каждого пакета в консоли соответствующую команду не очень удобно. В этом случае мы можем определить все пакеты в файле `package.json` и потом одной командой их установить.
- Например, изменим файл `package.json` следующим образом:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1",
    "react": "^16.9.0",
    "react-dom": "^16.9.0"
  }
}
```

- Здесь добавлены определения двух пакетов, которые представляют библиотеку React.
- Затем для загрузки всех пакетов выполнить команду
- `npm install`**
- Эта команда возьмет определение всех пакетов из секций `dependencies` и загрузит их в проект. Если пакет с нужной версией уже есть в проекте, как в данном случае `express`, то по новой он не загружается.

devDependencies

- Кроме пакетов, которые применяются в приложении, когда оно запущено и находится в рабочем состоянии, например, express, то есть в состоянии "production", есть еще пакеты, которые применяются при разработке приложения и его тестировании. Такие пакеты, как правило, добавляются в другую секцию - devDependencies.
- Например, загрузим в проект пакет jasmine-node, который используется для тестирования приложения:
- npm install jasmine-node --save-dev**
- Флаг --save-dev указывает, что информацию о пакете следует сохранить именно в секции devDependencies файла package.json:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1",
    "react": "^16.9.0",
    "react-dom": "^16.9.0"
  },
  "devDependencies": {
    "jasmine-node": "^3.0.0"
  }
}
```

Удаление пакетов

- Для удаления пакетов используется команда `npm uninstall`. Например:
- **`npm uninstall express`**
- При этом не важно, где располагается информация о пакете - в секции `dependencies` или `devDependencies`, пакет удаляется из любой из этих секций.
- Если нам надо удалить не один пакет, а несколько, то мы можем удалить их определение из файла `package.json` и ввести команду `npm install`, и удаленные из `package.js` пакеты также будут удалены из папки `node_modules`.
- Например, изменим файл `package.json` следующим образом:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
  }
}
```

- Здесь больше нет определения никаких пакетов. И введем команду
- **`npm install`**
- Причем мы также можем одновременно некоторые пакеты добавлять в `package.json`, а некоторые, наоборот, удалять. И при выполнении команды `npm install` пакетный менеджер новые пакеты установит, а удаленные из `package.json` пакеты удалит.

Пример Package.json

```
{
  "name": "test-project",
  "version": "1.0.0",
  "description": "A Vue.js project",
  "main": "src/main.js",
  "private": true,
  "scripts": {
    "dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js",
    "start": "npm run dev",
    "unit": "jest --config test/unit/jest.conf.js --coverage",
    "test": "npm run unit",
    "lint": "eslint --ext .js,.vue src test/unit",
    "build": "node build/build.js"
  },
  "dependencies": {
    "vue": "^2.5.2"
  },
  "devDependencies": {
    "autoprefixer": "^7.1.2",
    "babel-core": "^6.22.1",
    "babel-eslint": "^8.2.1",
    "babel-helper-vue-jsx-merge-props": "^2.0.3",
    "babel-jest": "^21.0.2",
    "babel-loader": "^7.1.1",
    "babel-plugin-dynamic-import-node": "^1.2.0",
    "babel-plugin-syntax-jsx": "^6.18.0",
    "babel-plugin-transform-es2015-modules-commonjs": "^6.26.0",
    "babel-plugin-transform-runtime": "^6.22.0",
    "babel-plugin-transform-vue-jsx": "^3.5.0",
    "babel-preset-env": "^1.3.2",
    "babel-preset-stage-2": "^6.22.0",
    "chalk": "^2.0.1",
    "copy-webpack-plugin": "^4.0.1",
    "css-loader": "^0.28.0",
    "eslint": "^4.15.0",
```


Пример Package.json

```

    "eslint-config-airbnb-base": "^11.3.0",
    "eslint-loader": "^1.7.1",
    "extract-text-webpack-plugin": "^3.0.0",
    "file-loader": "^1.1.4",
    "html-webpack-plugin": "^2.30.1",
    "jest-serializer-vue": "^0.3.0",
    "node-notifier": "^5.1.2",
    "optimize-css-assets-webpack-plugin": "^3.2.0",
    "ora": "^1.2.0",
    "portfinder": "^1.0.13",
    "postcss-import": "^11.0.0",
      "rimraf": "^2.6.0",
    "semver": "^5.3.0",
    "uglifyjs-webpack-plugin": "^1.1.1",
    "url-loader": "^0.5.8",
    "vue-jest": "^1.0.2",
    "vue-style-loader": "^3.0.1",
    "vue-template-compiler": "^2.5.2",
    "webpack": "^3.6.0",
    "webpack-dev-server": "^2.9.1",
    "webpack-merge": "^4.1.0"
  },
  "engines": {
    "node": ">= 6.0.0",
    "npm": ">= 3.0.0"
  },
  "browserslist": [
    "> 1%",
    "last 2 versions",
    "not ie <= 8"
  ]
}
```

Свойства package.json

- Можно выделить следующие свойства:
- **name** — задаёт имя приложения (пакета).
- **version** — содержит сведения о текущей версии приложения.
- **description** — краткое описание приложения.
- **main** — задаёт точку входа в приложение.
- **private** — если данное свойство установлено в true, это позволяет предотвратить случайную публикацию пакета в npm.
- **scripts** — задаёт набор Node.js-скриптов, которые можно запускать.
- **dependencies** — содержит список npm-пакетов, от которых зависит приложение.
- **devDependencies** — содержит список npm-пакетов, используемых при разработке проекта, но не при его реальной работе.
- **engines** — задаёт список версий Node.js, на которых работает приложение.
- **browserlist** — используется для хранения списка браузеров (и их версий), которые должно поддерживать приложение.
- Все эти свойства используются либо npm либо другими инструментальными средствами, применяемыми в течение жизненного цикла приложения.

Семантическое версионирование

- При определении версии пакета применяется семантическое версионирование. Номер версии, как правило, задается в следующем формате "major.minor.patch". Если в приложении или пакете обнаружен какой-то баг и он исправляется, то увеличивается на единицу число "patch". Если в пакет добавляется какая-то новая функциональность, которая совместима с предыдущей версией пакета, то это небольшое изменение, и увеличивается число "minor". Если же в пакет вносятся какие-то большие изменения, которые несовместимы с предыдущей версией, то увеличивается число "major". То есть глядя на разные версии пакетов, мы можем предположить, насколько велики в них различия.
- В примере с express версия пакета содержала, кроме того, дополнительный символ карет: "^4.14.0". Этот символ означает, что при установке пакета в проект с помощью команды `npm install` будет устанавливаться последняя доступная версия от 4.14.0. Фактически это будет последняя доступная версия в промежутке от 4.14.0 до 5.0.0 ($\geq 4.14.0$ и $< 5.0.0$).

Команды npm

- NPM позволяет определять в файле package.json команды, которые выполняют определенные действия. Например, определим следующий файл app.js:

```
let name = process.argv[2];
let age = process.argv[3];

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

- В данном случае мы получаем переданные при запуске приложению параметры.
- И определим следующий файл package.json:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "scripts" : {
    "start" : "node app.js",
    "dev" : "node app.js Tom 26"
  }
}
```

- Здесь добавлена секция scripts, которая определяет две команды. Вообще команд может быть много в соответствии с целями и задачами разработчика.
- Первая команда называется start. Она по сути выполняет команду node app.js, которая выполняет код в файле app.js
- Вторая команда называется dev. Она также выполняет тот же файл, но при этом также передает ему два параметра.

Команды npm

- Названия команд могут быть произвольными. Но здесь надо учитывать один момент. Есть условно говоря есть зарезервированные названия для команд, например, start, test, run и ряд других. Их не очень много. И как раз первая команда из выше определенного файла package.json называется start. И для выполнения подобных команд в терминале/командной строке надо выполнить команду

```
npm [название_команды]
```

- Например, для запуска команды start

```
npm start
```

- Команды с остальными названиями, как например, "dev" в вышеопределенном файле, запускаются так:

```
npm run [название_команды]
```

- Например, последовательно выполним обе команды:

```

c:\node\helloapp>npm start

> modulesapp@1.0.0 start c:\node\helloapp
> node app.js

name: undefined
age: undefined

c:\node\helloapp>npm run dev

> modulesapp@1.0.0 dev c:\node\helloapp
> node app.js Tom 26

name: Tom
age: 26

c:\node\helloapp>
  
```

Nodemon

- В процессе разработки может потребоваться необходимость внести изменения в уже запущенный проект. Допустим, у нас в файле app.js определен следующий код:

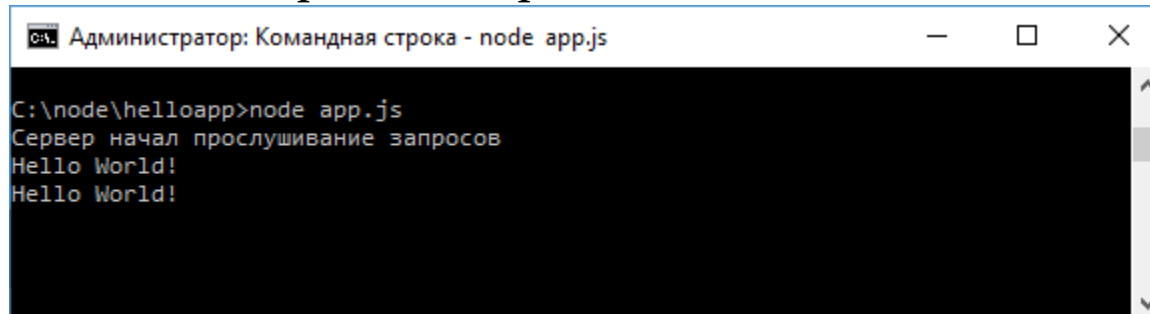
```
const http = require("http");

let message = "Hello World!";
http.createServer(function(request,response){

    console.log(message);
    response.end(message);

}).listen(3000, "127.0.0.1",()=>{
    console.log("Сервер начал прослушивание запросов");
});
```

- Запустим сервер с помощью команды `node app.js`, и при обращении пользователя по адресу `http://localhost:3000/` браузер пользователя отобразит строку "Hello World!". Одновременно строка выводится на консоль.



- При этом сервер продолжает быть запущенным. И если мы изменим переменную `message` в файле `app.js`, то это никак не повлияет на работу сервера, и он будет продолжать отдавать клиенту строку "Hello World!".

Nodemon

- В этом случае необходимо перезапустить сервер. Однако это не очень удобно, особенно когда необходимо часто делать различные изменения, тестировать выполнение. И в этом случае нам может помочь специальный инструмент nodemon.
- Установим nodemon в проект с помощью следующей команды:
- npm install nodemon -g**
- Флаг -g представляет сокращение от global и позволяет установить зависимость nodemon глобально для всех проектов на данной локальной машине.
- После установки запустим файл app.js с помощью следующей команды:
- nodemon app.js**
- И если вдруг после запуска сервера мы изменим его код, например, поменяем переменную message с "Hello World!" на "Привет мир!", то сервер автоматически будет перезапущен:

```

Администратор: Командная строка - nodemon server.js
C:\node\helloapp>nodemon app.js
[nodemon] 1.11.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node server.js`
Сервер начал прослушивание запросов
Hello World!
Hello World!
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Сервер начал прослушивание запросов
Привет мир!
Привет мир!
  
```

Асинхронность в Node.js

- Асинхронность представляет возможность одновременно выполнять сразу несколько задач. Асинхронность играет большую роль в Node.js.
- Например, допустим в файле приложения app.js у нас расположен следующий код:

```
function displaySync(data){
    console.log(data);
}

console.log("Начало работы программы");

displaySync("Обработка данных...");

console.log("Завершение работы программы");
```

- Это стандартный синхронный код, все вызовы здесь выполняются последовательно, что мы можем увидеть, если мы запустим приложение:

```
Администратор: Командная строка

C:\node\helloapp>node app.js
Начало работы программы
Обработка данных...
Завершение работы программы

C:\node\helloapp>_
```


Асинхронность в Node.js

- Для рассмотрения асинхронности изменим код файла app.js следующим образом:

```
function display(data, callback){

    // с помощью случайного числа определяем ошибку
    var randInt = Math.random() * (10 - 1) + 1;
    var err = randInt>5? new Error("Ошибка выполнения. randInt больше 5"): null;

    setTimeout(function(){
        callback(err, data);
    }, 0);
}

console.log("Начало работы программы");

display("Обработка данных...", function (err, data){

    if(err) throw err;
    console.log(data);
});

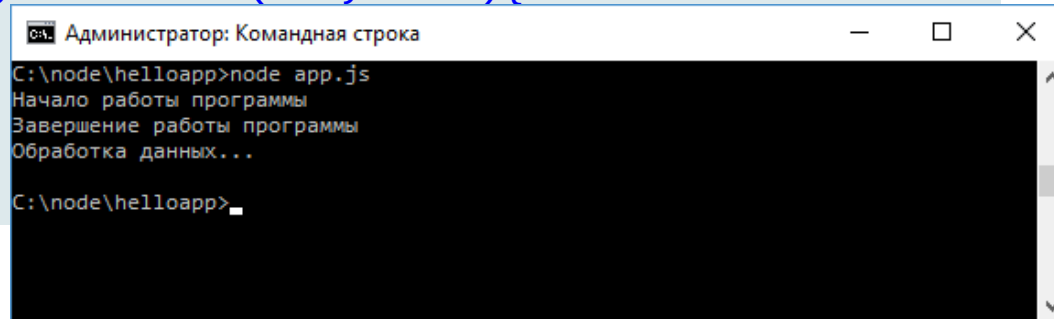
console.log("Завершение работы программы");
```

Асинхронность в Node.js

- В начале также определяется функция `display`, но теперь кроме данных в качестве второго параметра она принимает функцию обратного вызова, которая и обрабатывает данные.
- Эта функция `callback` принимает два параметра - информацию об ошибке и собственно данные. Это общая модель функций обратного вызова, которые передаются в асинхронные методы - первым идет параметр, представляющий ошибку, а второй - данные.
- Для имитации ошибки используется случайное число: если оно больше 5, то создаем объект ошибки - объект `Error`, иначе же он равен `null`.
- И последний важный момент - выполнение функции обратного вызова в функции `setTimeout()`. Это глобальная функция, которая принимает в качестве первого параметра функцию обратного вызова, а в качестве второго - промежуток, через который функция обратного вызова будет выполняться. Для нашей задачи вполне подойдет промежуток в 0 миллисекунд.
- При вызове функции `display` в нее передается функция, которая в случае отсутствия ошибок просто выводит данные на консоль:

```
display("Обработка данных...", function (err, data){

    if(err) throw err;
    console.log(data);
});
```



Асинхронность в Node.js

- Несмотря на то, что в `setTimeout` передается промежуток 0, фактическое выполнение функции `display` завершается после всех остальных функций, которые определены в программе. В итоге выполнение на функции `display` не блокируется, а идет дальше. И это особенно актуально, если в приложении идет какая-либо функция ввода-вывода, например, чтения файла или взаимодействия с базой данных, выполнение которой может занять продолжительное время. То общее выполнение приложение не блокируется, а идет дальше.
- Почему так происходит? Потому что все колбеки или функции обратного вызова в асинхронных функциях (в качестве таковой здесь используется функция `setTimeout`) помещаются в специальную очередь, и начинают выполняться после того, как все остальные синхронные вызовы в приложении завершат свою работу. Собственно поэтому выполнение колбека из функции `setTimeout` в примере выше происходит после выполнения вызова `console.log("Завершение работы программы");`. И стоит подчеркнуть, что в очередь колбеков переходит не функция, которая передается в `display`, а функция, которая передается в `setTimeout`.

Асинхронность в Node.js

- Рассмотрим пример с двумя асинхронными вызовами:

```
function displaySync(callback){
    callback();
}

console.log("Начало работы программы");

setTimeout(function(){

    console.log("timeout 500");
}, 500);

setTimeout(function(){

    console.log("timeout 100");
}, 100);
displaySync(function(){console.log("without timeout")});
console.log("Завершение работы программы");
```

```
Администратор: Командная строка

C:\node\helloapp>node app.js
Начало работы программы
without timeout
Завершение работы программы
timeout 100
timeout 500

C:\node\helloapp>
```

- Несмотря на то, что в функцию `display` передается колбек, но эта функция с колбеком будет выполняться синхронно.
- А колбеки из функций `setTimeout` будут выполняться только после всех остальных вызовов приложения.

Работа с файлами. Чтение из файла

- Для работы с файлами в Node.js предназначен модуль fs. Рассмотрим, как с ним работать.
- Допустим, в одной папке с файлом приложения app.js расположен текстовый файл hello.txt с простейшим текстом, например:
- Hello Node JS!
- Для чтения файла в синхронном варианте применяется функция fs.readFileSync():

```
let fileContent = fs.readFileSync("hello.txt", "utf8");
```

- В метод передается путь к файлу относительно файла приложения app.js, а в качестве второго параметра указывается кодировка для получения текстового содержимого файла. На выходе получаем считанный текст.
- Для асинхронного чтения файла применяется функция fs.readFile:

```
fs.readFile("hello.txt", "utf8", function(error,data){ });
```

- Первый и второй параметр функции опять же соответственно путь к файлу и кодировка. А в качестве третьего параметра передается функция обратного вызова, которая выполняется после завершения чтения. Первый параметр этой функции хранит информацию об ошибке при наличии, а второй - собственно считанные данные.

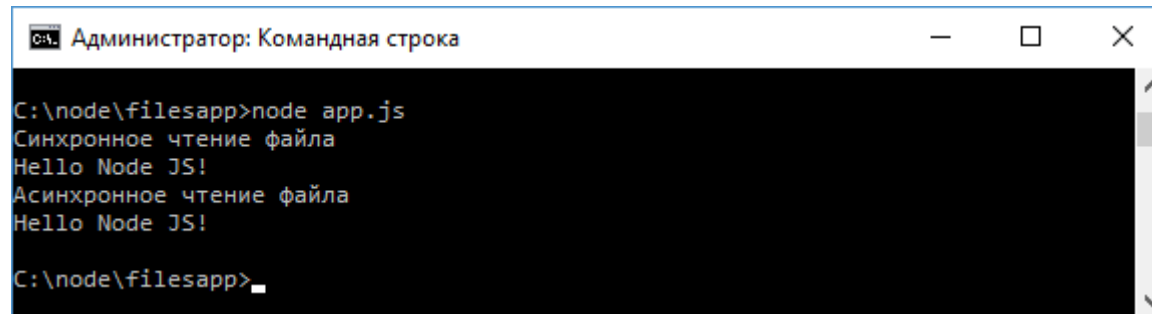
Чтение из файла

- Для чтения файла определим в файле app.js следующий код:

```
const fs = require("fs");

// асинхронное чтение
fs.readFile("hello.txt", "utf8",
    function(error,data){
        console.log("Асинхронное чтение файла");
        if(error) throw error; // если возникла ошибка
        console.log(data); // выводим считанные данные
    });

// синхронное чтение
console.log("Синхронное чтение файла")
let fileContent = fs.readFileSync("hello.txt", "utf8");
console.log(fileContent);
```



```
Администратор: Командная строка
C:\node\filesapp>node app.js
Синхронное чтение файла
Hello Node JS!
Асинхронное чтение файла
Hello Node JS!
C:\node\filesapp>_
```

- И здесь стоит обратить внимание, что несмотря на то, что функция fs.readFile() вызывается первой, но так как она асинхронная, она не блокирует поток выполнения, поэтому ее результат выводится в самом конце.

Запись файла

- Для записи файла в синхронном варианте используется функция `fs.writeFileSync()`, которая в качестве параметра принимает путь к файлу и записываемые данные:

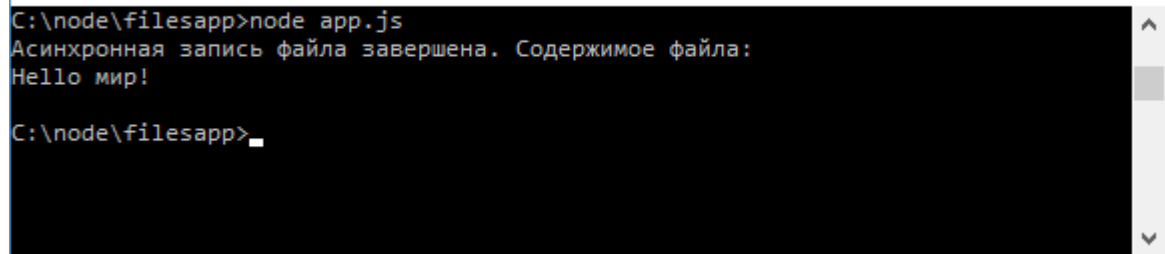
```
fs.writeFileSync("hello.txt", "Привет ми ми ми!")
```

- Также для записи файла можно использовать асинхронную функцию `fs.writeFile()`, которая принимает те же параметры:

```
fs.writeFile("hello.txt", "Привет МИГ-29!")
```

- В качестве вспомогательного параметра в функцию может передаваться функция обратного вызова, которая выполняется после завершения записи:

```
const fs = require("fs");
fs.writeFile("hello.txt", "Hello мир!", function(error){
    if(error) throw error; // если возникла ошибка
    console.log("Асинхронная запись файла завершена. Содержимое файла:");
    let data = fs.readFileSync("hello.txt", "utf8");
    console.log(data); // выводим считанные данные
});
```



```
C:\node\filesapp>node app.js
Асинхронная запись файла завершена. Содержимое файла:
Hello мир!
C:\node\filesapp>
```

- Следует отметить, что эти методы полностью перезаписывают файл. Если надо дозаписать файл, то применяются методы

fs.appendFile()/fs.appendFileSync():

```
const fs = require("fs");

fs.appendFileSync("hello.txt", "Привет ми ми ми!");

fs.appendFile("hello.txt", "Привет МИД!", function(error){
    if(error) throw error; // если возникла ошибка

    console.log("Запись файла завершена. Содержимое файла:");
    let data = fs.readFileSync("hello.txt", "utf8");
    console.log(data); // выводим считанные данные
});
```

```
Администратор: Командная строка

C:\node\filesapp>node app.js
Запись файла завершена. Содержимое файла:
Hello мир!Привет ми ми ми!Привет МИД!

C:\node\filesapp>
```


Удаление файла

- Для удаления файла в синхронном варианте используется функция `fs.unlinkSync()`, которая в качестве параметра принимает путь к удаляемому файлу:

```
fs.unlinkSync("hello.txt")
```

- Также для удаления файла можно использовать асинхронную функцию `fs.unlink()`, которая принимает путь к файлу и функцию, вызываемую при завершении удаления:

```
fs.unlink("hello.txt", (err) => {
  if (err) console.log(err); // если возникла ошибка
  else console.log("hello.txt was deleted");
});
```

События

- Подавляющее большинство функционала Node.js применяет асинхронную событийную архитектуру, которая использует специальные объекты - эмиттеры для генерации различных событий, которые обрабатываются специальными функциями - обработчиками или слушателями событий. Все объекты, которые генерируют события, представляют экземпляры класса EventEmitter.
- С помощью функции `eventEmitter.on()` к определенному событию по имени цепляется функция обработчика. Причем для одного события можно указать множество обработчиков. Когда объект `EventEmitter` генерирует событие, происходит выполнение всех этих обработчиков.
- Рассмотрим применение объекта `EventEmitter` и событий. Для этого определим следующий файл `app.js`:

```
const Emitter = require("events");
let emitter = new Emitter();
let eventName = "greet";
emitter.on(eventName, function(){
    console.log("Hello all!");
});
emitter.on(eventName, function(){
    console.log("Привет!");
});
emitter.emit(eventName);
```

```
Администратор: Командная строка
C:\node\eventsapp>node app.js
Hello all!
Привет!
C:\node\eventsapp>
```

- Весь необходимый функционал сосредоточен в модуле `events`, который необходимо подключить. С помощью функции `on()` связываем событие, которое передается в качестве первого параметра, с некоторой функцией, которая передается в качестве второго параметра. В данном случае событие называется "greet". Для генерации события и вызова связанных с ним обработчиков выполняется функция `emitter.emit()`, в которое передается название события.
- И при запуске приложения будут вызваны все обработчики:

Передача параметров событию

- При вызове события в качестве второго параметра в функцию emit можно передавать некоторый объект, который передается в функцию обработчика события:

```
const Emitter = require("events");
let emitter = new Emitter();
let eventName = "greet";
emitter.on(eventName, function(data){
    console.log(data);
});

emitter.emit(eventName, "Привет пир!");
```

Наследование от EventEmitter

- В приложении мы можем оперировать сложными объектами, для которых также можно определять события, но для этого их надо связать с объектом EventEmitter. Например:

```
const util = require("util");
const EventEmitter = require("events");

function User(){
}
util.inherits(User, EventEmitter);

let eventName = "greet";
User.prototype.sayHi = function(data){
    this.emit(eventName, data);
}
let user = new User();
// добавляем к объекту user обработку события "greet"
user.on(eventName, function(data){
    console.log(data);
});

user.sayHi("Мне нужна твоя одежда...");
```

- Здесь определена функция конструктора User, которая представляет пользователя. Для прототипа User определяется метод sayHi, в котором генерируется событие "greet".

Наследование от EventEmitter

- Но чтобы связать объект User с EventEmitter, надо вызвать функцию `util.inherits(User, EventEmitter);`. Она позволяет унаследовать классу User функционал от EventEmitter. Благодаря этому мы можем через метод `on()` добавить к событию объекта user какой-нибудь обработчик, который будет вызван при выполнении метода `user.sayHi()`.

```

Администратор: Командная строка
C:\node\eventsapp>node app.js
Мне нужна твоя одежда...
C:\node\eventsapp>

```

- С помощью возможностей ES6 мы можем упростить выше пример:

```

const EventEmitter = require("events");
let eventName = "greet";

class User extends EventEmitter {
  sayHi(data) {
    this.emit(eventName, data);
  }
}

let user = new User();
// добавляем к объекту user обработку события "greet"
user.on(eventName, function(data){
  console.log(data);
});
user.sayHi("Мне нужна твоя одежда...");

```

- Результат будет тот же, но теперь не нужно использовать функцию `util.inherits`.

Stream

- Stream представляет поток данных. Потoki бывают различных типов, среди которых можно выделить потоки для чтения и потоки для записи.

- При создании сервера в первой главе мы уже сталкивались с потоками:

```
const http = require("http");

http.createServer(function(request, response){
}).listen(3000);
```

- Параметры request и response, которые передаются в функцию и с помощью которых мы можем получать данные о запросе и управлять ответом, как раз представляют собой потоки: request - поток для чтения, а response - поток для записи.

- Используя потоки чтения и записи, мы можем считывать и записывать информацию в файл. Например:

```
const fs = require("fs");

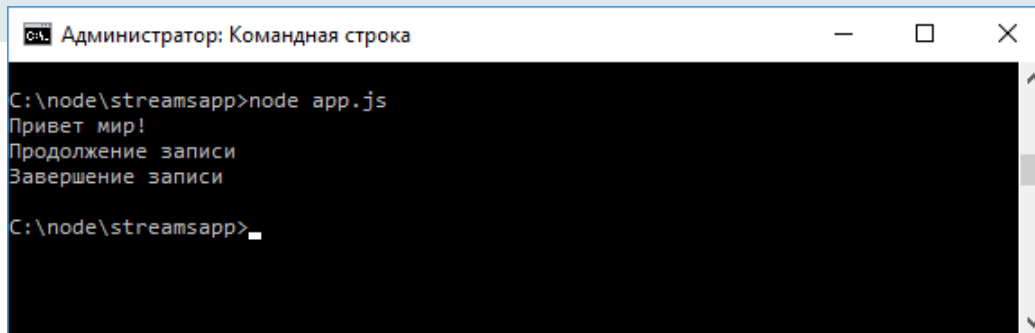
let writeableStream = fs.createWriteStream("hello.txt");
writeableStream.write("Привет мир!");
writeableStream.write("Продолжение записи \n");
writeableStream.end("Завершение записи");
let readableStream = fs.createReadStream("hello.txt", "utf8");

readableStream.on("data", function(chunk){
    console.log(chunk);
});
```

Stream

- Для создания потока для записи применяется метод `fs.createWriteStream()`, в который передается название файла. Если вдруг в папке нет такого файла, то он создается.
- Запись данных производится с помощью метода `write()`, в который передаются данные. Для окончания записи вызывается метод `end()`.
- После этого в папке проекта появляется файл `hello.txt`, который можно открыть в любом текстовом редакторе.
- Для создания потока для чтения используется метод `fs.createReadStream()`, в который также передается название файла. В качестве опционального параметра здесь передается кодировка, что позволит сразу при чтении кодировать считанные данные в строку в указанной кодировке.
- Сам поток разбивается на ряд кусков или чанков (`chunk`). И при считывании каждого такого куска, возникает событие `data`. С помощью метода `on()` мы можем подписаться на это событие и вывести каждый кусок данных на консоль:

```
readableStream.on("data", function(chunk){
    console.log(chunk);
});
```



```
Администратор: Командная строка

C:\node\streamsapp>node app.js
Привет мир!
Продолжение записи
Завершение записи

C:\node\streamsapp>
```

- Только работой с файлами функциональность потоков не ограничивается, также имеются сетевые потоки, потоки шифрования, архивации и т.д., но общие принципы работы с ними будут те же, что и у файловых потоков.

Pipe

- Pipe - это канал, который связывает поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи. Для чего они нужны? Возьмем, к примеру проблему копирования данных из одного файла в другой.
- Пусть в папке проекта определен некоторый файл hello.txt. Скопируем его содержимое в новый файл some.txt:

```
const fs = require("fs");

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

let writableStream = fs.createWriteStream("some.txt");

readableStream.on("data", function(chunk){
    writableStream.write(chunk);
});
```

- Данный код вполне работоспособен, и после запуска файла в папке проекта появится новый файл some.txt.

Pipe

- Однако задача записи в поток данных, считанных из другого потока, является довольно распространенной, и в этом случае `pipes` или каналы позволяют нам сократить объем кода:

```
const fs = require("fs");

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

let writableStream = fs.createWriteStream("some2.txt");

readableStream.pipe(writableStream);
```

- У потока чтения вызывается метод `pipe()`, в который передается поток для записи.
- Рассмотрим другую проблему - архивацию файла. Здесь нам надо сначала считать файл, затем сжать данные и в конце записать сжатые данные в файл-архив. `Pipes` особенно удобно применять для подобного набора операций:

```
const fs = require("fs");
const zlib = require("zlib");
let readableStream = fs.createReadStream("hello.txt", "utf8");
let writableStream = fs.createWriteStream("hello.txt.gz");
let gzip = zlib.createGzip();
readableStream.pipe(gzip).pipe(writableStream);
```

- Для архивации подключается модуль `zlib`. Каждый метод `pipe()` в цепочке вызовов возвращает поток для чтения, к которому опять же можно применить метод `pipe()` для записи в другой поток.

Спасибо за внимание.