NODE.JS

Содержание

Установка Node.js

Node или Node.js — программная платформа, основанная на движке V8, который превращает язык JavaScript из узко специализированного, в язык общегоназначения.

Применяется преимущественно на сервере, выполняя роль веб-сервера, но есть возможность разрабатывать на Node.js и десктопные приложения (при помощи NW.js или Electron для Linux, Windows и Mac OS).

В основе Node.js лежит событийно-ориентированное и асинхронное программирование с неблокирующим вводом/выводом.

Наиболее частое применение Node.js находит при разработке:чатов и систем обмена мгновенными сообщениями; многопользовательских игр в реальном времени; сетевых сервисов для сбора и отправки больших объемов информации.

Также хорошо подходит для создания стандартных веб-приложений. Ее используют для создания консольных утилит, такие популярные системы сборки для front-end как Grunt.js и Gulp.js созданы с помощью Node.

Чтобы установить Node на компьютер, вам нужно пойти на сайт https://nodejs.org/en/ и скачать LTSили текущую версию (на момент написания методички это были версии v8.12.0 и v11.0.0 соответственно).

А что делать, если вы хотите установить эти две версии сразу?

Для этого есть специальная утилита ***nvm (Node Version Manager)*** — это скрипт, который позволяет устанавливать, переключать и удалять версии Node.js т.е. даёт возможность держать на одной машине любое количество версий Node.js. Как обычно, работа под Windows совсем не радужна, но эта статья вам поможет.

Чтобы проверить работоспособность после установки наберите в консоли:

```
$ node
```

Вы попадете в интерактивную консоль node, прямо в которой можно набирать и выполнять команды JavaScript.

```
> 1+2
3
>
```

В этом режиме в консоль просто выводится результат набранного выражения.
Давайте запишем,для примера, некий код в файл с именем *start.js*:

```
let text = ‘Hello student!’;
console.log(text);
```

И запустим его из консоли в той директории, где он был создан, следующей командой:

```
$ node start.js
```

В консоли должна появиться надпись:

```
Hello student!
```

Модули

Для подключения к вашим скриптам дополнительных функций в Node. js существует удобная система управления модулями ***NPM***. По сути это публичный репозиторий созданных при помощи Node.js дополнительных программных модулей.

Команда npm позволяет легко устанавливать, удалять или обновлять нужные вам модули, автоматически учитывая при этом все зависимости выбранного вами модуля от других.

Установка модуля производится командой:

```
npm install *имя модуля* [*ключи*]
```

Для установки модуля будет использована поддиректория *node_modules*.

Хотя *node_modules* и содержит все необходимые для запуска зависимости, распространять исходный код вместе с ней не принято, т.к. в ней может храниться большое количество файлов,которые занимают ощутимый объем и это неудобно.

С учетом того, что все публичные NPM-модули можно легко установить с помощью npm, достаточно создать и написать для вашей программы файл *package.json* с перечнем всех необходимых для работы зависимостейи потом просто, на новом месте, например, установить все нужные модули командой:

```
$ npm install
```

Node.js работает с системой подключения модулей ***CommonJS***. В структурном плане, CommonJS-модуль представляет собой готовый к новому использованию фрагмент JavaScript-кода, который экспортирует специальные объекты, доступные для использования в любом зависимом коде. CommonJS используется как формат JavaScript-модулей так же и
на front-end. Две главных идеи CommonJS-модулей: **объект exports**, содержащий то, что модуль хочет сделать доступным для других частей системы, и **функцию require**, которая используется одними модулями для импорта объекта exports из других.

Начиная с версии 6.x Node.js так же поддерживает подключение модулей согласно стандарту ECMAScript-2015.

Давайте попробуем что-нибудь подключить. Например, модуль colors для предыдущего скрипта, и немного перепишем его. Наш скрипт станет выглядеть так:

```
let colors = require('colors');
let text = 'Hello student!';
console.log(text.rainbow);
```

Выполним команды в консоли:

```
npm i colors
node start.js
```

И теперь наша надпись должна стать разноцветной

И, наверняка, почувствуете что-то такое.

Как работает Node?

В основе Node лежит библиотека **libuv**, реализующая цикл событий **event loop**.

Мы знаем, что объявленная переменная в скрипте автоматически становится глобальной. В Node она остается *локальной для текущего модуля* и чтобы сделать ее глобальной, надо объявить ее как свойство объекта Global:

```
global.foo = 3;
```

Фактически, объект **Global** — это аналог объекта window из браузера.

Метод **require**, служащий для подключения модулей, не является глобальным и *локален* для каждого модуля.

Также *локальными* для каждого модуля являются:

module.export– объект, отвечающий за то, что именно будет экспортировать модуль при использовании require;

_filename – имя файла исполняемого скрипта;

_dirname – абсолютный путь до исполняемого скрипта.

В секцию *Global* входят такие важные элементы как:

Class: Buffer – объект используется для операций с бинарными данными.

Process – объект процесса, большая часть данных находится именно здесь.

Приведем пример работы некоторых из них. Назначение понятно из названий:

```
console.log(process.execPath);
console.log(process.version);
console.log(process.platform);
console.log(process.arch);
console.log(process.title);
console.log(process.pid);
```

```
e:\Program Files\nodejs\node.exe
v5.5.0
win32
ia32
MINGW32:/d/WebDir/Node_exp/app/color
3240
```

Свойство **process.argv** содержит массив аргументов командной строки. Первым аргументом будет имя исполняемого приложения node, вторым имя самого исполняемого сценария и только потом сами параметры.

Для работы с каталогами есть следующие свойства – **process.cwd()** возвращает текущий рабочий каталог, **process.chdir()** выполняет переход в другой каталог.

Команда **process.exit()** завершает процесс с указанным в качестве аргумента кодом: 0 – успешный код, 1 – код с ошибкой.

Важный метод **process.nextTick(fn)** запланирует выполнение указанной функции таким образом, что указанная функция будет выполнена после окончания текущей фазы (текущего исполняемого кода), но перед началом следующей фазы eventloop.

```
process.nextTick(function() {
console.log('NextTick callback');
}
```

ОбъектProcess содержит еще много свойств и методов, с которыми можно ознакомиться в справке.

События

За события в Node.js отвечает специальный модуль ***events***. Назначать объекту

обработчик события следует методом
addListener(event, listener). Аргументы – это имя события *event*, в
camelCase формате и *listener* — функция обратного вызова, обработчик события. Для этого метода есть более короткая запись **on()**.

Удалить обработчик можно методом **removeListener(event, listener)**. А метод

emit(event, [args]) позволяет событиям срабатывать.
Например, событие **'exit'** отправляется перед завершением работы Node.

```
process.on('exit' , function() {  console.log('Bye!');
});
```

Работа с файлами

Модуль ***FileSystem*** отвечает за работу с файлами. Инициализация модуля происход следующим образом:

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

```
fs.exists(path, callback) - проверка существования файла.
```

```
fs.readFile(filename, [options], callback) - чтение файла целиком
```

```
fs.writeFile(filename, data, [options], callback) - запись файла целиком
```

```
fs.appendFile(filename, data, [options], callback) - добавление в файл
```

```
fs.rename(oldPath, newPath, callback) - переименование файла.
```

```
fs.unlink(path, callback) - удаление файла.
```

Функции **callback** принимают как минимум один параметр *err*, который равен *null* при успешном выполнении команды или содержит информацию

об ошибке. Помимо этого при вызове **readFile** передается параметр *data*, который содержит уже упоминавшийся объект типа *Buffer*, содержащий последовательность прочитанных байтов. Чтобы работать с ним как со строкой, нужно его конвертировать методом toString()

```
fs.readFile('readme.txt', function (err, data) {  if (err) {
        throw err;
    }
    console.log(data.toString());
});
```

Также почти все методы модуля fs имеют синхронные версии функции, оканчивающиеся на *Sync*. Этим функциям не нужны callback, т.к. они являются блокирующими и поэтому рекомендованы к применению, только если это требует текущая задача. Давайте напишем программу, которая будет читать каталог и выводить его содержимое, а для файлов выводить их размер и дату последнего изменения.

```
const fs = require('fs'),
    path = require('path'),
    dir = process.cwd(),
    files = fs.readdirSync(dir);

console.log('Name \t Size \t Date \n');

files.forEach(function (filename) {
    let fullname = path.join(dir, filename),
        stats = fs.statSync(fullname);
    if (stats.isDirectory()) {
    console.log(filename + '\t DIR \t' + stats.mtime + '\n');
} else {
        console.log(filename + '\t' + stats.size + '\t' + stats.  mtime + '\n');
            }
        });
```

Давайте разберем эту программу подробно. В начале мы подключаем два стандартных модуля:

```
const fs = require('fs'),
 path = require('path')
```

Первый отвечает за запись и чтения файлов, а модуль path за работу с путями файлов. В переменную dir мы с помощью метода process.cwd() сохраняем текущую директорию и тут же в переменную files считываем в синхронном режиме fs.readdirSync(dir) все файлы из текущего каталога. В синхронном потому, что нам надо получить весь список файлов и поддиректорий из текущей директории, прежде чем приступить к ее анализу. Выводим шапку нашей будущей таблички:

```
console.log('Name \t Size \t Date \n');
```

И потом методом forEach по массиву files, прочитанных элементов директории, проходимся и выводим в консоль информацию об элементах. Через метод path.join соединяем пути к файлу, и в переменую stats записываем информацию о текущем файле. Мы выводим stats.mtime — время создания файла и stats.size для определения размера файла.
С помощью stats.isDirectory() определяем является ли элемент директорией и если да, для него не выводим размер, а ключевое слово DIR.