**Тема 13. Основы работы с Node.js (Node)**

1. **Установка Node.**
2. **Первое приложение на Node.js.**
3. **Модули.**
4. **Объект global и глобальные переменные.**
5. **Передача параметров приложению.**
6. **NPM.**
7. **Nodemon.**
8. **Асинхронность в Node.**
9. **Работа с файлами.**
10. **События.**
11. **Stream.**
12. **Pipe.**
13. **Создание сервера.**
14. **Отправка файлов.**
15. **Шаблоны.**

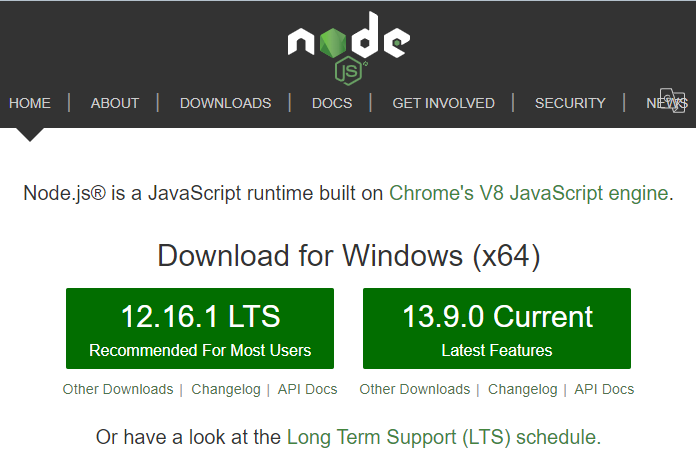
Содержание данной темы включает материалы, доступные по адресу https://metanit.com.

1. Установка Node.js.

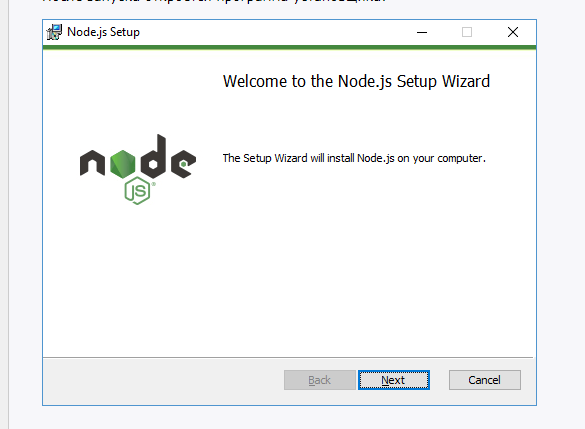
Node.js представляет среду выполнения кода на JavaScript, которая построена на основе движка JavaScript Chrome V8, который позволяет транслировать вызовы на языке JavaScript в машинный код. Node.js прежде всего предназначен для создания серверных приложений на языке JavaScript. Хотя также существуют проекты по написанию десктопных приложений (Electron) и даже по созданию кода для микроконтроллеров. Но прежде будем рассматривать Node.js, как о платформу для создания веб-приложений.

Node.js является открытым проектом, исходники которого можно посмотреть на [github.com](https://github.com/nodejs).

Скачать Node.js можно с официального сайта <https://nodejs.org/en/>. На главной есть две возможные опции для загрузки: самая последняя версия NodeJS и LTS-версия. LTS – расшифровывается как Long Time Support или, в переводе на русский, поддержка в течение длительного времени.



Загрузим последнюю версию. Для Windows установщик представляет файл с расширением msi. После запуска откроется программа установщика:



После успешной установки можно ввести в командной строке/терминале команду node -v, и отобразится текущая версия node.js:



Версии node.js для других операционных систем наряду с исходниками можно найти по адресу <https://nodejs.org/en/download/>

**Инструменты разработки**

Для разработки под Node JS достаточно простейшего текстового редактора, в частности, Notepad++. Также можно использовать более изощренные редакторы типа Atom, Sublime, Visual Studio Code, либо среды разработки, которые поддерживают работу с Node.JS, например, Visual Studio или WebStorm.

**REPL**

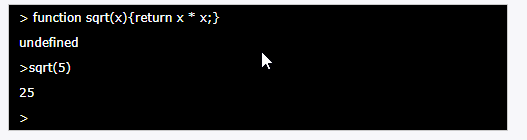
После установки NodeJS становится доступным такой инструмент как REPL. REPL (Read Eval Print Loop) представляет возможность запуска выражений на языке JavaScript в командной строке или терминале. Запустим командную строку (на Windows) или терминал (на OS X или Linux) и введем команду node. После ввода этой команды можно выполнять различные выражения на JavaScript:



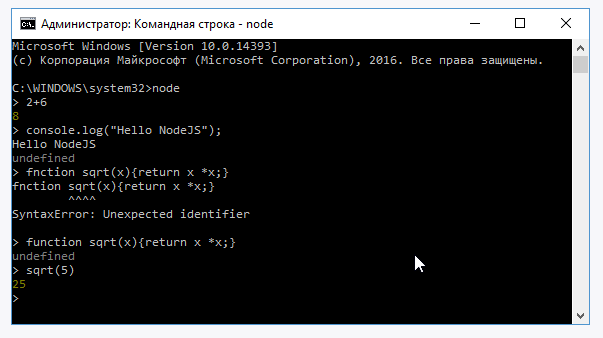
Или использовать какую-нибудь функцию JS:



Можно определять свои функции и затем их вызывать, например, возведение числа в квадрат:



Если ввести что-то неправильно, то REPL укажет об ошибке:



**Выполнение файла**

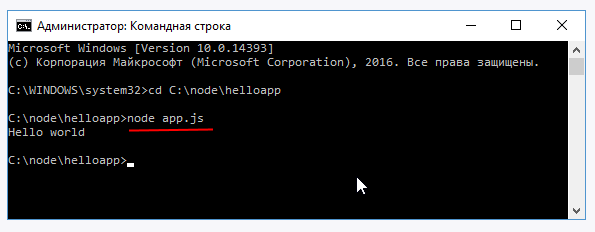
Вместо того чтобы вводить весь код напрямую в консоль, удобнее вынести его во внешний файл. Например, создадим на жестком диске новый каталог, допустим, C:\node\helloapp, в который поместим новый файл app.js со следующим кодом:

|  |  |
| --- | --- |
|  | console.log("Hello world"); |

В командной строке перейдем с помощью команды cd к каталогу helloapp, а затем выполним команду:



Данная команда выполнит код из файла app.js:



1. Первое приложение на Node.js

Создадим простейшее приложение для NodeJS. Для создания приложений можно использовать практически все стандартные конструкции языка JavaScript. Исключением является работа с DOM, так как приложение будет запускаться на сервере, а не в браузере, поэтому DOM и такие объекты как window или document в данном случае нам будут недоступны.

Вначале создадим для приложения каталог на жестком диске. Например, каталог C:\node\helloapp. В этом каталоге создадим файл app.js. Определим в файле app.js следующий код:

|  |  |
| --- | --- |
|  | const http = require("http");  http.createServer(function(request,response){        response.end("Hello NodeJS!");    }).listen(3000, "127.0.0.1",function(){      console.log("Сервер начал прослушивание запросов на порту 3000");  }); |

На первой строке подгружается модуль http, который необходим для создания сервера. Это встроенный модуль, и для его загрузки необходимо применить функцию require():

|  |  |
| --- | --- |
|  | const http = require("http"); |

Далее с помощью метода createServer() создается новый сервер для прослушивания входящих подключений и обработки запросов. В качестве параметра этот метод принимает функцию, которая имеет два параметра. Первый параметр request хранит всю информацию о запросе, а второй параметр response используется для отправки ответа. В данном случае ответ представляет простую строку "Hello NodeJS!" и отправляется с помощью метода response.end().

Но метод http.createServer() только создает сервер. Чтобы сервер начал прослушивать входящие подключения у него надо вызвать метод listen:

|  |  |
| --- | --- |
|  | .listen(3000, "127.0.0.1",function(){      console.log("Сервер начал прослушивание запросов на порту 3000");  }); |

Этот метод принимает три параметра. Первый параметр указывает на локальный порт, по которому запускается сервер. Второй параметр указывает на локальный адрес. То есть в данном случае сервер будет запускаться по адресу 127.0.0.1 или localhost на порту 3000. Третий параметр представляет функцию, которая запускается при начале прослушивания подключений. Здесь эта функция просто выводит диагностическое сообщение на консоль.

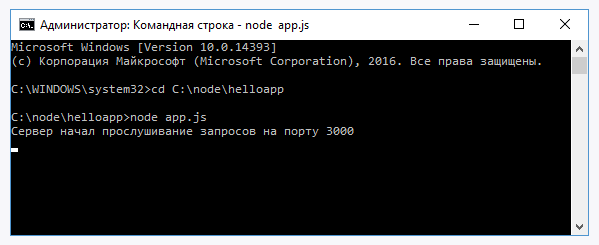
Теперь запустим сервер. Для этого откроем терминал (в OS X или Linux) или командную строку (в Windows). С помощью команды cd перейдем к каталогу приложения:



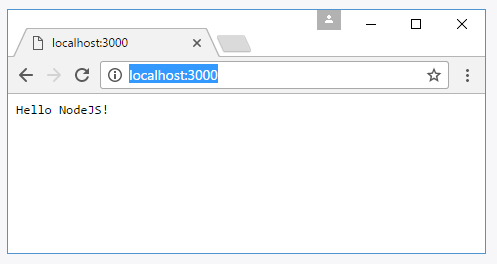
Затем вызовем следующую команду:



Она запускает сервер:



Далее откроем браузер и введем в адресную стоку адрес <http://localhost:3000/>:



Отобразится то сообщение, которое было послано в методе response.end().

1. Модули.

Node.js использует модульную систему. То есть вся встроенная функциональность разбита на отдельные пакеты или модули. Модуль представляет блок кода, который может использоваться повторно в других модулях. При необходимости можно подключать нужные модули. Какие встроенные модули есть в node.js и какую функциональность они предоставляют, можно узнать из [документации](https://nodejs.org/api/).

Для загрузки модулей применяется функция require(), в которую передается название модуля. Например, загрузим модуль http:

|  |  |
| --- | --- |
|  | const http = require("http"); |

После получения модуля можно использовать весь определенный в нем функционал, который опять же можно посмотреть в [документации](https://nodejs.org/api/http.html). Подобным образом можно загружать и использовать другие встроенные модули. Например, используем модуль [os](https://nodejs.org/api/os.html), который предоставляет информацию об окружении и операционной системе:

|  |  |
| --- | --- |
|  | const os = require("os");  // получим имя текущего пользователя  let userName = os.userInfo().username;    console.log(userName); |

При необходимости можно создать свои модули. Так, в прошлом вопросе проект состоял из файла app.js, в котором создавался сервер, обрабатывающий запросы. Добавим в тот же каталог новый файл greeting.js и определим в нем следующий код:

|  |  |
| --- | --- |
|  | console.log("greeting module"); |

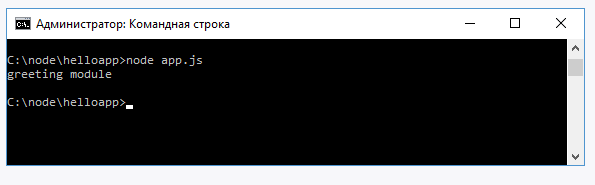
В файле app.js подключим созданный модуль:

|  |  |
| --- | --- |
|  | const greeting = require("./greeting"); |

В отличие от встроенных модулей для подключения своих модулей надо передать в функцию require относительный путь с именем файла (расширение файла необязательно):

|  |  |
| --- | --- |
|  | const greeting = require("./greeting"); |

Запустим приложение:



На консоль выводится та строка, которая определена в файле greeting.js.

Теперь изменим файл greeting.js:

|  |  |
| --- | --- |
|  | let currentDate = new Date();  module.exports.date = currentDate;    module.exports.getMessage = function(name){      let hour = currentDate.getHours();      if(hour > 16)          return "Добрый вечер, " + name;      else if(hour > 10)          return "Добрый день, " + name;      else          return "Доброе утро, " + name;  } |

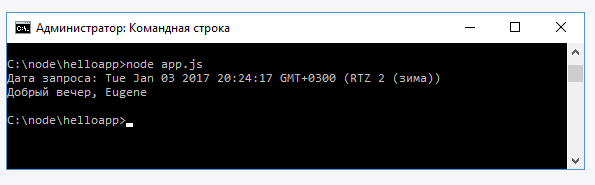
Здесь определена переменная currentDate. Однако из вне она недоступна. Она доступна только в пределах данного модуля. Чтобы какие-то переменные или функции модуля были доступны, необходимо определить их в объекте module.exports. Объект module.exports – это то, что возвращает функция require() при получении модуля.

Т.е. объект module представляет ссылку на текущий модуль, а его свойство exports определяет все свойства и методы модуля, которые могут быть экспортированы и использованы в других модулях. В рассматриваемом примере определяется свойство date и метод getMessage, который принимает некоторый параметр. Изменим файл app.js:

|  |  |
| --- | --- |
|  | const os = require("os");  const greeting = require("./greeting");    // получим имя текущего пользователя  let userName = os.userInfo().username;      console.log(`Дата запроса: ${greeting.date}`);  console.log(greeting.getMessage(userName)); |

Все экспортированные методы и свойства модуля доступны по имени: greeting.date и greeting.getMessage().

Перезапустим приложение:



Кроме определения простейших функций или свойств в модуле могут определяться сложные объекты или функции конструкторов, которые затем используются для создания объектов. Так, добавим в папку проекта новый файл user.js:

|  |  |
| --- | --- |
|  | function User(name, age){        this.name = name;      this.age = age;      this.displayInfo = function(){            console.log(`Имя: ${this.name}  Возраст: ${this.age}`);      }  }  User.prototype.sayHi = function() {      console.log(`Привет, меня зовут ${this.name}`);  };    module.exports = User; |

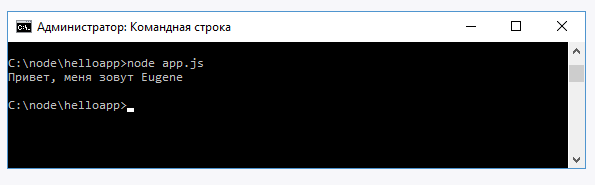
Здесь определена стандартная функция конструктора User, которая принимает два параметра. При этом весь модуль теперь указывает на эту функцию конструктора:

|  |  |
| --- | --- |
|  | module.exports = User; |

Подключим и используем этот модуль в файле app.js:

|  |  |
| --- | --- |
|  | const User = require("./user.js");    let eugene = new User("Eugene", 32);  eugene.sayHi(); |

Запустим приложение:



Рассмотрим некоторые аспекты работы с модулями в Node.js. Прежде всего надо отметить, что подключаемые модули кэшируются. Это, с одной стороны, увеличивает производительность, а с другой, может создать некоторые проблемы. Например, в проекте из прошлого вопроса в главный файл приложения app.js подключается модуль greeting.js. Изменим файл greeting.js следующим образом:

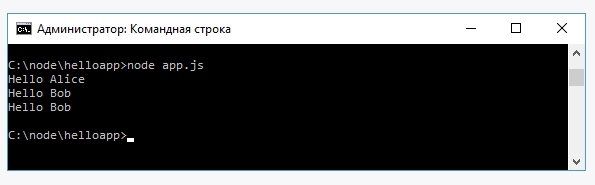
|  |  |
| --- | --- |
|  | module.exports.name = "Alice"; |

В файле определена только одна строка, которая устанавливает свойство name.

Изменим код файла app.js:

|  |  |
| --- | --- |
|  | var greeting1 = require("./greeting.js");  console.log(`Hello ${greeting1.name}`); //Hello Alice    var greeting2 = require("./greeting.js");  greeting2.name= "Bob";    console.log(`Hello ${greeting2.name}`); //Hello Bob  // greeting1.name тоже изменилось  console.log(`Hello ${greeting1.name}`); //Hello Bob |

Несмотря на то, что здесь два раза получаем модуль с помощью функции require, но обе переменных - greeting1 и greeting2 будут указывать на один и тот же объект.



Нередко модули приложения образуют какие-то отдельные наборы или области. Такие наборы модулей лучше помещать в отдельные каталоги. Например, создадим в каталоге приложения подкаталог welcome и создадим в нем три новых файла: index.js, morning.js, evening.js.

В итоге общая структура проекта будет выглядеть следующим образом:

* welcome
* index.js
* morning.js
* evening.js
* app.js
* greeting.js

В файл morning.js поместим следующую строку:

|  |  |
| --- | --- |
|  | module.exports = "Доброе утро"; |

Аналогично изменим файл evening.js:

|  |  |
| --- | --- |
|  | module.exports = "Добрый вечер"; |

Эти два файла определяют сообщения приветствия в зависимости от времени суток. Определим в файле index.js следующий код:

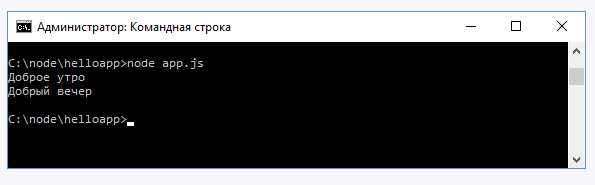
|  |  |
| --- | --- |
|  | const morning = require("./morning");  const evening = require("./evening");    module.exports = {      getMorningMessage : function(){ console.log(morning);},      getEveningMessage : function(){ console.log(evening);}  } |

В модуле определен объект, который имеет две функции для вывода приветствий.

Теперь используем этот модуль в файле app.js:

|  |  |
| --- | --- |
|  | const welcome = require("./welcome");    welcome.getMorningMessage();  welcome.getEveningMessage(); |

Несмотря на то, что нет такого файла как welcome.js, но если в проекте есть каталог, который содержит файл с именем index.js, то можно обращаться к модулю по имени каталога, как в данном случае. Запустим приложение, и на консоль будут выведены оба приветствия:



1. Объект global и глобальные переменные.

Node.js предоставляет специальный объект global, который предоставляет доступ к глобальным, то есть доступным из каждого модуля приложения, переменным и функциям. Примерным аналогом данного объекта в javascript для браузера является объект window. Все доступные глобальные объекты можно посмотреть в [документации](https://nodejs.org/api/globals.html).

Для примера создадим следующий модуль greeting.js:

|  |  |
| --- | --- |
|  | let currentDate = new Date();    global.date = currentDate;    module.exports.getMessage = function(){      let hour = currentDate.getHours();      if(hour >16)          return "Добрый вечер, " + global.name;      else if(hour >10)          return "Добрый день, " + name;      else          return "Доброе утро, " + name;  } |

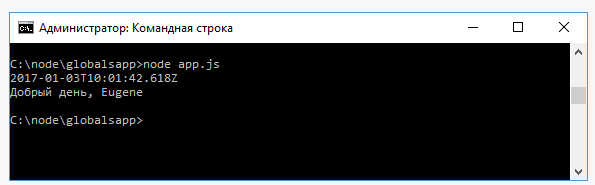
Здесь, во-первых, происходит установка глобальной переменной date: global.date = currentDate. Во-вторых, в модуле получаем глобальную переменную name, которая будет установлена из вне. При этом обратиться к глобальной переменной name можно через объект global: global.name, либо просто через имя name, так как переменная глобальная.

Определим следующий файл приложения app.js:

|  |  |
| --- | --- |
|  | const greeting = require("./greeting");    global.name = "Eugene";    global.console.log(date);  console.log(greeting.getMessage()); |

Здесь устанавливаем глобальную переменную name, которую получаем в модуле greeting.js. И также выводим на консоль глобальную переменную date. Причем все глобальные функции и объекты, например, console, также доступны внутри global, поэтому можно написать и global.console.log(), и просто console.log().

Запустим файл app.js:



Однако по возможности все-таки рекомендуется избегать определения и использования глобальных переменных, и преимущественно ориентироваться на создание переменных, инкапсулированных в рамках отдельных модулей.

1. Передача параметров приложению.

При запуске приложения из терминала/командной строки можно передавать ему параметры. Для получения параметров в коде приложения применяется массив process.argv. Это аналогично тому, как в языках C/C++/C#/Java в функцию main передается набор аргументов в виде строкового массива.

Первый элемент этого массива всегда указывает на путь к файлу node.exe, который вызывает приложение. Второй элемент массив всегда указывает на путь к файлу приложения, который выполняется.

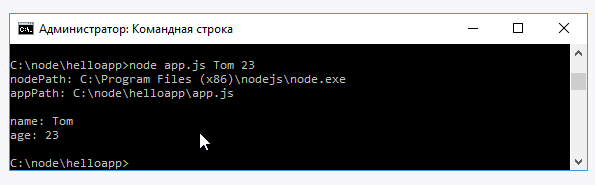
К примеру, определим следующий файл app.js:

|  |  |
| --- | --- |
|  | 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); |

В данном случае ожидаем, что приложению будут переданы два параметра: name и age. Теперь запустим приложение с помощью следующей команды:



В данном случае "Tom" и "23" - это те значения, которые помещаются соответственно в process.argv[2] и process.argv[3]:



1. NPM.

Кроме встроенных и кастомных модулей Node.js существует огромный пласт различных библиотек и фреймворков, разнообразных утилит, которые создаются сторонними производителями и которые также можно использовать в проекте, например, express, grunt, gulp и так далее. И они тоже доступны в рамках Node.js. Чтобы удобнее было работать со всеми сторонними решениями, они распространяются в виде пакетов. Пакет по сути представляет набор функциональностей.

Для автоматизации установки и обновления пакетов, как правило, применяются систему управления пакетами или менеджеры. Непосредственно в Node.js для этой цели используется пакетный менеджер [NPM](https://www.npmjs.com/) (Node Package Manager). NPM по умолчанию устанавливается вместе с Node.js, поэтому ничего дополнительно устанавливать не требуется. Но можно обновить установленную версию до самой последней. Для этого в командной строке/терминале надо запустить следующую команду:

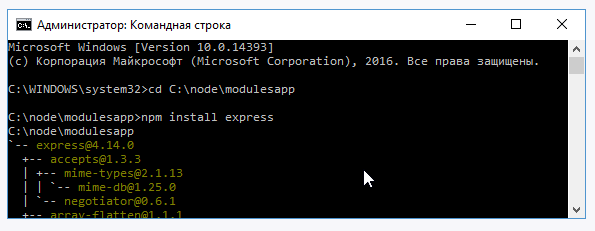


Чтобы узнать текущую версию npm, в командной строке/терминале надо ввести следующую команду:



С помощью менеджера npm легко управлять пакетами. К примеру, создадим новую папку modulesapp (C:\node\modulesapp). Далее для примера установим в проект express. Express представляет легковесный веб-фреймворк для упрощения работы с Node.js. Для установки функциональности Express в проект вначале перейдем к папке проекта с помощью команды cd. Затем введем команду





После установки express в папке проекта modulesapp появится подпапка node\_modules, в которой будут хранится все установленные внешние модули. В частности, в подкаталоге node\_modules/express будут располагаться файлы фреймворка Express.

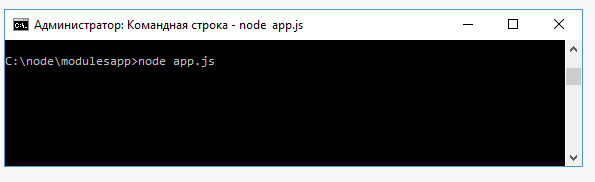
Далее определим файл простейшего сервера. Для этого в корневую папку проекта modulesapp добавим новый файл app.js:

|  |  |
| --- | --- |
|  | // 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); |

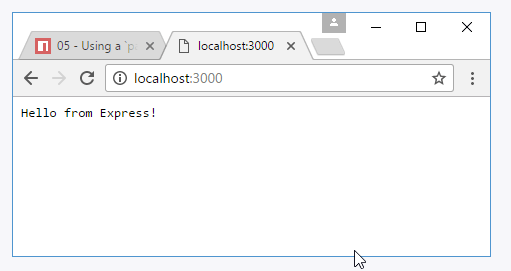
Первая строка получает установленный модуль express, а вторая создает объект приложения. В Express можно связать обработку запросов с определенными маршрутами. Например, "/" – представляет главную страницу или корневой маршрут. Для обработки запроса вызывается функция app.get(). Первый параметр функции – 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/:



**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:

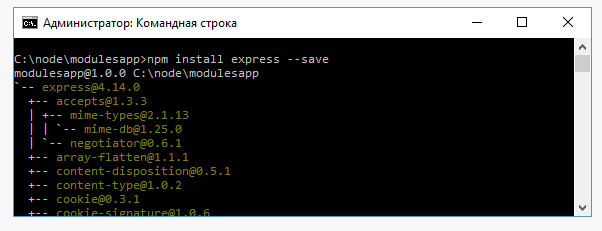
|  |  |
| --- | --- |
|  | {    "name": "modulesapp",    "version": "1.0.0"  } |

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.

Далее удалим из проекта каталог node\_modules. То есть в папке проекта modulesapp будут два файла app.js и package.json. Теперь снова добавим express с помощью следующей команды:



Флаг --save указывает, что информацию о добавленном пакете надо добавить в файл package.json.



И после выполнения команды, если открыть файл package.json, то увидим информацию о пакете:

|  |  |
| --- | --- |
|  | {    "name": "modulesapp",    "version": "1.0.0",    "dependencies": {      "express": "^4.14.0"    }  } |

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

Файл package.json играет большую роль и может облегчить разработку в различных ситуациях. Например, при размещении в разных репозиториях нередко ограничены выделяемые дисковые пространства, тогда как папка node\_modules со всеми загруженными пакетами может занимать довольно приличное пространство. В этом случае удобнее разместить основной код проекта без node\_modules. В этом случае можно определить все пакеты в файле package.json, а затем для загрузки всех пакетов выполнить команду



Эта команда возьмет определение всех пакетов из секций dependencies и загрузит их в проект.

**devDependencies**

Кроме пакетов, которые применяются в приложении, когда оно запущено, например, express, то есть в состоянии "production", есть еще пакеты, которые применяются при разработке приложения и его тестировании. Такие пакеты добавляются в другую секцию – devDependencies. Например, загрузим в проект пакет jasmine-node, который используется для тестирования приложения:



Флаг --save-dev указывается, что информацию о пакете следует сохранить в секции devDependencies файла package.json:

|  |  |
| --- | --- |
|  | {    "name": "modulesapp",    "version": "1.0.0",    "dependencies": {      "express": "^4.14.0"    },    "devDependencies": {      "jasmine-node": "^1.14.5"    }  } |

**Удаление пакетов**

Для удаления пакетов используется команда npm uninstall. Например:



Эта команда удаляет пакет из папки node\_modules, в то же время в файле package.json информация о данном пакете остается. Чтобы удалить информацию также и из package.json, применяется флаг --save:



**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, можно 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 (>=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 получаем переданные при запуске приложению параметры.

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 назвывается dev. Она также выполняет тот же файл, но при этом также передает ему два параметра.

Названия команд могут быть произвольными. Но здесь надо учитывать один момент. Есть зарезервированные названия для команд, например, start, test, run и ряд других. 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] |

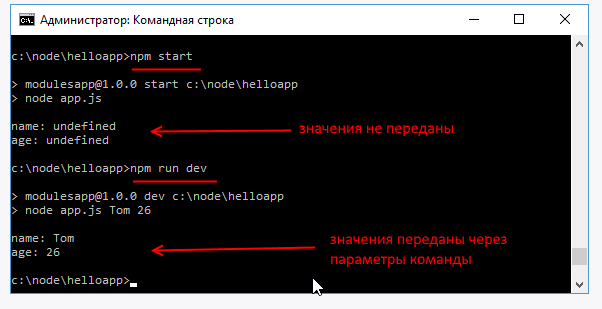
Например, для запуска команды start:

|  |  |
| --- | --- |
|  | npm start |

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

|  |  |
| --- | --- |
|  | npm run [название\_команды] |

Например, последовательно выполним обе команды:

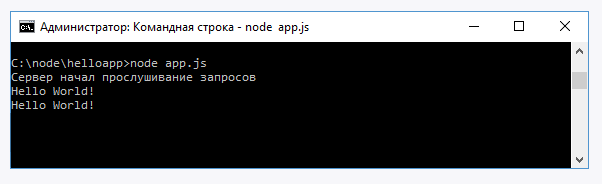


1. Nodemon.

В процессе разработки может потребоваться необходимость внести изменения в уже запущенный проект. Допустим, 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 изменить переменную message в файле app.js, то это никак не повлияет на работу сервера, и он будет продолжать отдавать клиенту строку "Hello World!".

В этом случае необходимо перезапустить сервер. Однако это не очень удобно, особенно когда необходимо часто делать различные изменения, тестировать выполнение. И в этом случае может помочь специальный инструмент nodemon.

Установим nodemon в проект с помощью следующей команды:

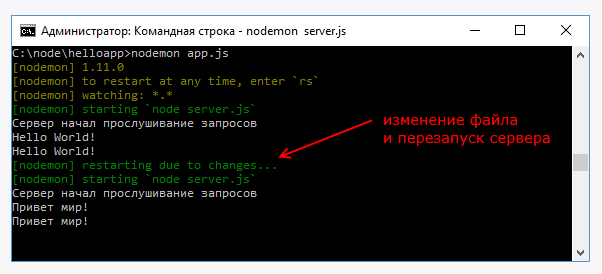


Флаг -g представляет сокращение от global и позволяет установить зависимость nodemon глобально для всех проектов на данной локальной машине.

После установки запустим файл app.js с помощью следующей команды:



И если вдруг после запуска сервера изменить его код, например, поменяем переменную message с "Hello World!" на "Привет мир!", то сервер автоматически будет перезапущен:



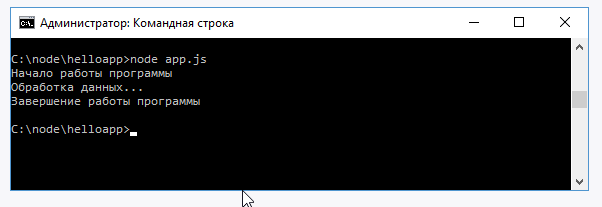
1. Асинхронность в Node.

Асинхронность представляет возможность одновременно выполнять сразу несколько задач. Асинхронность играет большую роль в Node.js.

Например, допустим в файле приложения app.js  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. Вот что можно увидеть, если запустить приложение:



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"); |

В начале также определяется функция display, но теперь кроме данных в качестве второго параметра она принимает функцию обратного вызова, которая и обрабатывает данные.

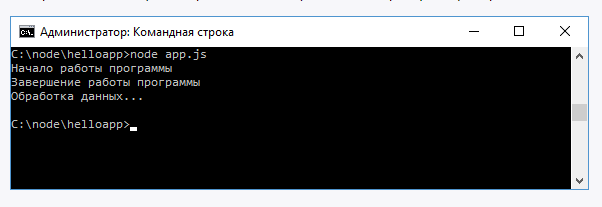
Эта функция callback принимает два параметра – информацию об ошибке и собственно данные. Это общая модель функций обратного вызова, которые передаются в асинхронные методы – первым идет параметр, представляющий ошибку, а второй – данные. Для имитации ошибки используется случайное число: если оно больше 5, то создаем объект ошибки – объект Error, иначе же он равен null.

В функции setTimeout() выполняется функция обратного вызова. 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 рассматриваемой задачи 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);  }); |

Теперь если запустить приложение, то увидим, следующее:

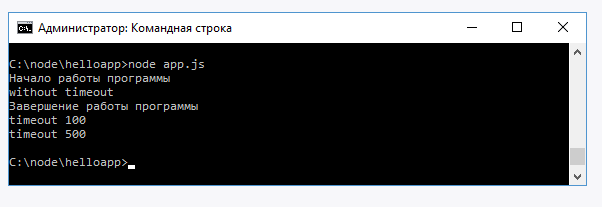


Несмотря на то, что в setTimeout передается промежуток 0, фактическое выполнение функции display завершается после всех остальных функций, которые определены в программе. В итоге выполнение на функции display не останавливается, а продолжается. Так происходит, потому что все колбеки или функции обратного 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. Именно 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.

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"); |

Результат выполнения:



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

1. Работа с файлами.

Для работы с файлами в Node.js предназначен модуль [fs](https://nodejs.org/api/fs.html).

**Чтение из файла**

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.

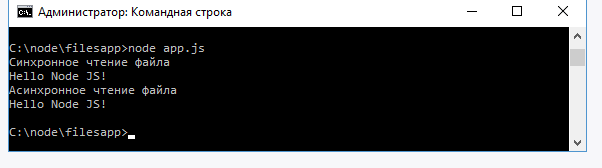
Для асинхронного чтения файла применяется функция fs.readFile:

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

Первый и второй параметр функции – это 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 – считанные данные.

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); |



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:

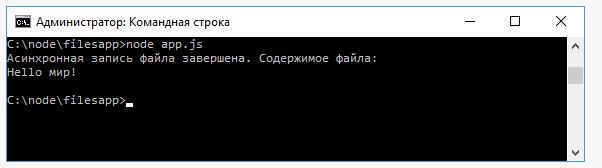
|  |  |
| --- | --- |
|  | fs.writeFileSync("hello.txt", "Привет ми ми ми!") |

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

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

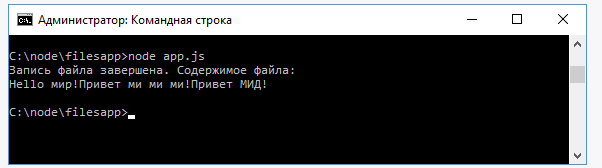
В качестве вспомогательного параметра в функцию может передаваться функция обратного вызова, которая выполняется после завершения записи:

|  |  |
| --- | --- |
|  | 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  }); |



Следует отметить, что эти методы полностью перезаписывают файл. Если надо дополнить 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  }); |



**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");  }); |

1. События.

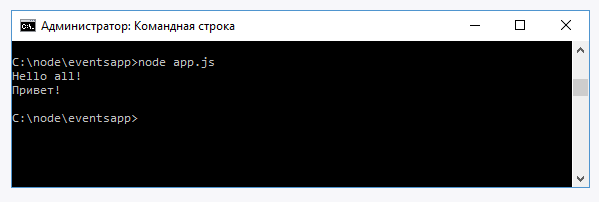
Подавляющее большинство функционала Node.js применяет асинхронную событийную архитектуру, которая использует специальные объекты – эмиттеры для генерации различных событий, которые обрабатываются специальными функциями – обработчиками или слушателями событий. Все объекты, которые генерируют события, представляют экземпляры класса EventEmitter. С помощью функции eventEmitter.on() к определенному событию по имени назначается 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); |

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:



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

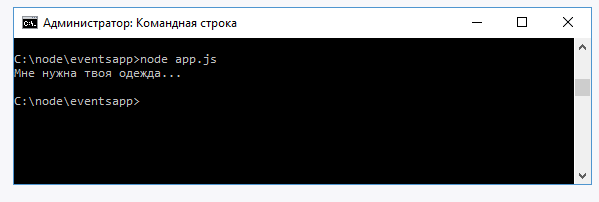
|  |  |
| --- | --- |
|  | const Emitter = require("events");  let emitter = new Emitter();  let eventName = "greet";  emitter.on(eventName, function(data){      console.log(data);  });   emitter.emit(eventName, "Привет пир!"); |

**Наследование от EventEmitter**

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



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.

1. Stream.

Stream представляет поток данных. Потоки бывают различных типов, среди которых можно выделить [потоки для чтения](https://nodejs.org/api/stream.html#stream_readable_streams) и [потоки для записи](https://nodejs.org/api/stream.html#stream_writable_streams).

|  |  |
| --- | --- |
|  | const http = require("http");   http.createServer(function(request, response){  }).listen(3000); |

Параметры request и response, которые передаются в функцию и с помощью которых можно 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, можно 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);  }); |

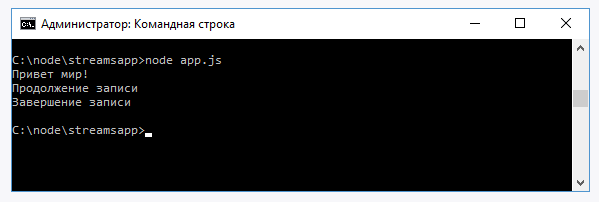
Для создания потока для записи применяется метод fs.createWriteStream(), в который передается название файла. Если вдруг в папке нет такого файла, то он создается. Запись данных производится с помощью метода write(), в который передаются данные. Для окончания записи вызывается метод end(). После этого 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() можно подписаться на это событие и вывести каждый кусок данных на консоль:

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

Запустим файл на выполнение:



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.

1. Pipe.

Pipe – это канал, который связывает поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи. Чтобы понять для чего они нужны, рассмотрим, например, 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 writeableStream = fs.createWriteStream("some.txt");    readableStream.on("data", function(chunk){      writeableStream.write(chunk);  }); |

Данный код вполне работоспособен, и после запуска файла в папке проекта появится новый файл some.txt. Однако задача записи в поток данных, считанных из другого потока, является довольно распространенной, и в этом случае pipes или каналы позволяют сократить объем кода:

|  |  |
| --- | --- |
|  | const fs = require("fs");    let readableStream = fs.createReadStream("hello.txt", "utf8");    let writeableStream = fs.createWriteStream("some2.txt");    readableStream.pipe(writeableStream); |

У потока чтения вызывается метод pipe(), в который передается поток для записи.

Рассмотрим другую проблему – архивацию файла. Здесь 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 writeableStream = fs.createWriteStream("hello.txt.gz");    let gzip = zlib.createGzip();    readableStream.pipe(gzip).pipe(writeableStream); |

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.

1. Создание сервера.

Для работы с сервером и протоколом http в Node.js используется модуль [http](https://nodejs.org/api/http.html). Чтобы создать сервер, следует вызвать метод http.createServer():

|  |  |
| --- | --- |
|  | const http = require("http");   http.createServer().listen(3000); |

Метод createServer() возвращает объект http.Server. Но чтобы сервер мог прослушивать и обрабатывать входящие подключения, у объекта сервера необходимо вызвать метод listen(), в который в качестве параметра передается номер порта, по которому запускается сервер. Для обработки подключений в метод createServer можно передать специальную функцию:

|  |  |
| --- | --- |
|  | const http = require("http");   http.createServer(function(request, response){      response.end("Hello world!");  }).listen(3000); |

Эта функция принимает два параметра:

* request: хранит информацию о запросе,
* response: управляет отправкой ответа.

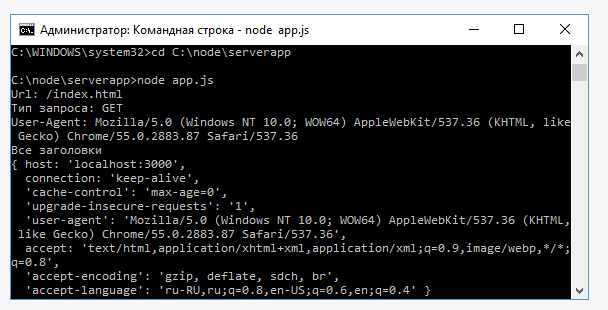
Параметр request позволяет получить информацию о запросе и представляет объект http.IncomingMessage. Отметим некоторые основные свойства этого объекта:

* headers: возвращает заголовки запроса;
* method: тип запроса (GET, POST, DELETE, PUT);
* url: представляет запрошенный адрес.

Например, определим следующий файл app.js:

|  |  |
| --- | --- |
|  | var http = require("http");   http.createServer(function(request, response){      console.log("Url: " + request.url);      console.log("Тип запроса: " + request.method);      console.log("User-Agent: " + request.headers["user-agent"]);      console.log("Все заголовки");      console.log(request.headers);        response.end();  }).listen(3000); |

Запустим его и обратимся в браузере по адресу http://localhost:3000/index.html:



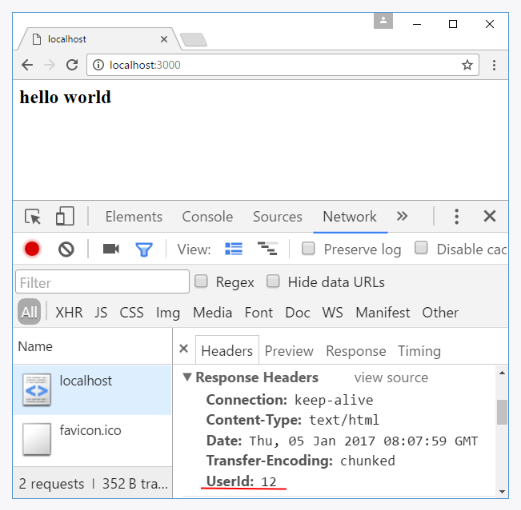
Параметр response управляет отправкой ответа и представляет объект http.ServerResponse. Среди его функциональности можно выделить следующие методы:

* statusCode: устанавливает статусный код ответа;
* statusMessage: устанавливает сообщение, отправляемое вместе со статусным кодом;
* setHeader(name, value): добавляет в ответ один заголовок;
* write: пишет в поток ответа некоторое содержимое;
* writeHead: добавляет в ответ статусный код и набор заголовков;
* end: сигнализирует серверу, что заголовки и тело ответа установлены, в итоге ответ отсылается клиента. Данный метод должен вызываться в каждом запросе.

Например, изменим файл app.js следующим образом:

|  |  |
| --- | --- |
|  | const http = require("http");  http.createServer(function(request, response){      response.setHeader("UserId", 12);      response.setHeader("Content-Type", "text/html; charset=utf-8;");      response.write("<h2>hello world</h2>");      response.end();  }).listen(3000); |

Запустим файл и обратимся в браузере к приложению:



Если предстоит отправить довольно большой ответ, то мож несколько раз вызвать метод write(), последовательно оправляя в исходящий поток каждый кусочек информации. Например, отправим код веб-страницы:

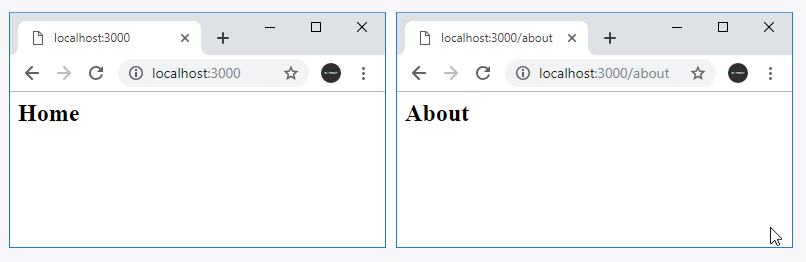
|  |  |
| --- | --- |
|  | const http = require("http");  http.createServer(function(request, response){     response.setHeader("Content-Type", "text/html");      response.write("<!DOCTYPE html>");      response.write("<html>");      response.write("<head>");      response.write("<title>Hello Node.js</title>");      response.write("<meta charset=\"utf-8\" />");      response.write("</head>");      response.write("<body><h2>Привет миг</h2></body>");      response.write("</html>");      response.end();  }).listen(3000); |

**Маршрутизация**

По умолчанию Node.js не имеет встроенной системы маршрутизации. Обычно она реализуется с помощью специальных фреймворках типа Express. Однако если необходимо разграничить простейшую обработку пары-тройки маршрутов, то можно использовать для этого свойство url объекта Request. Например:

|  |  |
| --- | --- |
|  | const http = require("http");  http.createServer(function(request, response){      response.setHeader("Content-Type", "text/html; charset=utf-8;");      if(request.url === "/home" || 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(3000); |

В данном случае обрабатываются три маршрута. Если идет обращение к корню сайта или по адресу localhost:3000/home, то пользователю выводится строка "Home". Ели обращение идет по адресу localhost:3000/about, то пользователю в браузере отображается строка About и так далее. Если запрошенный адрес не соответствует ни одному маршруту, то выводится заговлок "Not Found".



**Переадресация**

Переадресация предполагает отправку статусного кода 301 (постоянная переадресация) или 302 (временная переадресация) и заголовка Location, который указывает на новый адрес. Например, выполним переадресацию с адреса localhost:3000/ на адрес localhost:3000/newpage:

|  |  |
| --- | --- |
|  | const http = require("http");  http.createServer(function(request, response){      response.setHeader("Content-Type", "text/html; charset=utf-8;");      if(request.url === "/"){          response.statusCode = 302; // временная переадресация          // на адрес localhost:3000/newpage          response.setHeader("Location", "/newpage");      }      else if(request.url == "/newpage"){          response.write("New address");      }      else{          response.write("Not Found");          response.statusCode = 404; // адрес не найден      }      response.end();  }).listen(3000); |

1. Отправка файлов.

Отправка статических файлов – довольно частая задача в построении и функционировании веб-приложения. Рассмотрим, как отправлять файлы в приложении на Node.js. Создадим в каталоге проекта три файла: app.js, about.html, index.html. В файле index.html определим следующий код:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <title>Главная</title>      <meta charset="utf-8" />  </head>  <body>      <h1>Главная</h1>  </body>  <html> |

Аналогично определим код в файле about.html:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <title>О сайте</title>      <meta charset="utf-8" />  </head>  <body>      <h1>О сайте</h1>  </body>  <html> |

Задача будет заключаться в том, чтобы отправить их содержимое пользователю.

Для считывания файла может применяться метод fs.createReadStream(), который считывает файл в поток, и затем с помощью метода pipe() можно связать считанные файлы с потоком записи, то есть объектом response. Итак, поместим в файл app.js следующий код:

|  |  |
| --- | --- |
|  | const http = require("http");  const fs = require("fs");    http.createServer(function(request, response){        console.log(`Запрошенный адрес: ${request.url}`);      // получаем путь после слеша      const filePath = request.url.substr(1);      // смотрим, есть ли такой файл      fs.access(filePath, fs.constants.R\_OK, err => {          // если произошла ошибка - отправляем статусный код 404          if(err){              response.statusCode = 404;              response.end("Resourse not found!");          }          else{              fs.createReadStream(filePath).pipe(response);          }        });  }).listen(3000, function(){      console.log("Server started at 3000");  }); |

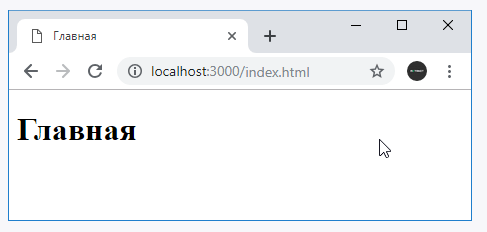
Вначале получаем запрошенный адрес. Допустим, запрошенный адрес будет соответствовать напрямую пути к файлу на сервере. Затем с помощью асинхронной функции fs.access проверяем доступность файла для чтения. Первый параметр функции – путь к файлу. Второй параметр – опция, относительно которой проверяется доступ. В данном случае значение fs.constants.R\_OK говорит о том, что проверяются права на чтение из файла. Третий параметр функции – функция обратного вызова, которая получает объект ошибки. Если произошла ошибка (файл не доступен для чтения или вовсе не найден), посылаем статусный код 404.

Для отправки файла применяется цепочка методов:

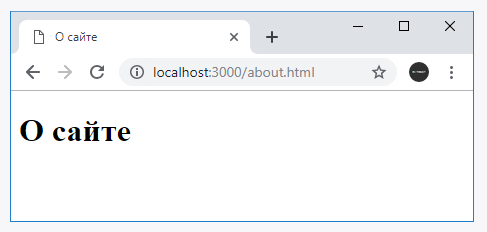
|  |  |
| --- | --- |
|  | fs.createReadStream("some.doc").pipe(response); |

Метод fs.createReadStream("some.doc") создает поток для чтения – объект fs.ReadStream. Для получения данных из потока вызывается метод pipe(), в который передается объект интерфейса stream.Writable или поток для записи. А именно таким и является объект http.ServerResponse, который реализует этот интерфейс.

Запустим приложение и в браузере обратимся по адресу "http://localhost:3000/index.html":



Аналогично можно обратиться по адресу "http://localhost:3000/about.html":



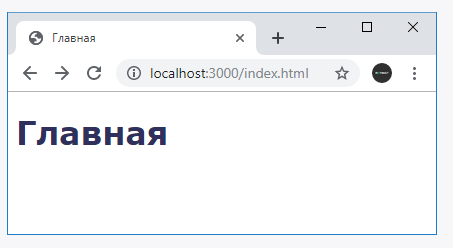
В данном случае отправляются файлы html, но подобным образом можно отправлять самые разные файлы. Например, определим в проекте папку public. И в ней создадим новый файл styles.css со следующим содержимым:

|  |  |
| --- | --- |
|  | body{ font-family: Verdana; color:rgb(48, 48, 92);} |

Применим эти стили на станице index.html:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <title>Главная</title>      <meta charset="utf-8" />      <link href="public/styles.css" rel="stylesheet" type="text/css">  </head>  <body>      <h1>Главная</h1>  </body>  <html> |

И затем обратимся к index.html:



Второй способ отправки содержимого файлов пользователю заключается в том, что чтение данных может осуществляться с помощью функции fs.readFile() и отправка с помощью метода response.end():

|  |  |
| --- | --- |
|  | const http = require("http");  const fs = require("fs");    http.createServer(function(request, response){        console.log(`Запрошенный адрес: ${request.url}`);      // получаем путь после слеша      const filePath = request.url.substr(1);      fs.readFile(filePath, function(error, data){            if(error){                response.statusCode = 404;              response.end("Resourse not found!");          }          else{              response.end(data);          }      });  }).listen(3000, function(){      console.log("Server started at 3000"); |

1. Шаблоны.

В прошлом вопросе использовался файл html, который имел статичное неизменяющееся содержимое. Однако также можно применять специальные инструменты – шаблоны, вместо которых в файл будет вставляться какой-то определенный текст. Например, изменим файл index.html следующим образом:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <title>Главная</title>      <meta charset="utf-8" />  </head>  <body>      <h1>{header}</h1>      <p>{message}</p>  </body>  <html> |

Вместо конкретного содержимого здесь определены плейсхолдеры "{header}" и "{message}", вместо которых может вставляться любой текст.

Изменим файл app.js:

|  |  |
| --- | --- |
|  | const http = require("http");  const fs = require("fs");    http.createServer(function(request, response){        fs.readFile("index.html", "utf8", function(error, data){            let message = "Изучаем Node.js";          let header = "Главная страница";          data = data.replace("{header}", header).replace("{message}", message);          response.end(data);      })  }).listen(3000); |

Здесь получаем содержимое файла и проводим его дополнительную обработку, заменяя плейсхолдеры на конкретный текст с помощью метода data.replace(). При обращении к приложению получим полноценную html-страницу без плейсхолдеров:

