Лабораторна робота №1

Початок роботи з Node.JS. Пакетний менеджер npm. Фреймворк Express. Пакети для тестування Jest та Supertest.



Node.js представляє середу виконання коду на JavaScript, яка побудована на основі движка JavaScript Chrome V8, який дозволяє транслювати виклики на мові JavaScript в машинний код. Node.js перш за все призначений для створення серверних додатків на мові JavaScript. Хоча також існують проекти по написанню десктопних додатків (Electron) і навіть зі створення коду для мікроконтролерів. Але перш за все ми говоримо про Node.js, як про платформу для створення веб-додатків. Node.js є відкритим проектом, вихідні коди якого можна подивитися на <https://github.com>. У Node.js використовується libuv - крос-платформна бібліотека підтримки з акцентом на асинхронний ввід-вивід.



З точки зору розробника, Node.js однопоточна, але під капотом libuv використовує треди, події файлової системи, реалізує цикл подій, включає в себе тред-пулінг і так далі. У більшості випадків ви не будете взаємодіяти з libuv безпосередньо.

1. Установка

Для завантаження потрібно перейти на офіційний сайт <https://nodejs.org>. На головній сторінці ми відразу побачимо дві можливі опції для завантаження: найостанніша версія NodeJS і LTS-версія. Рекомендується вибирати LTS версію.

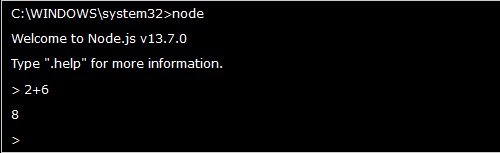
Після успішної установки ви можете ввести в командному рядку / терміналі команду node -v, і нам відобразиться поточна версія node.js:



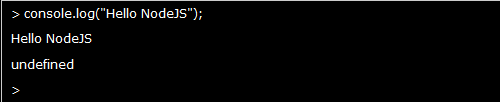
Для розробки під Node JS досить найпростішого текстового редактора, зокрема, Notepad ++. Також можна використовувати більш витончені редактори типу Atom, Sublime, Visual Studio Code, або середовища розробки, які підтримують роботу з Node.JS, наприклад, Visual Studio або WebStorm.

Після установки NodeJS нам стає доступним такий інструмент як REPL. REPL (Read Eval Print Loop) представляє можливість запуску виразів на мові JavaScript в командному рядку або терміналі.

Запустимо командний рядок (на Windows) або термінал (на OS X або Linux) і введемо команду node. Після введення цієї команди ми можемо виконувати різні вирази на JavaScript:



Або використовуємо якусь функцію JS:



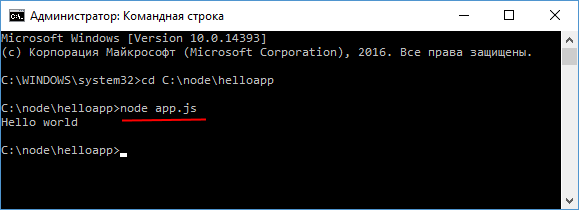
Замість того щоб вводити весь код безпосередньо в консоль, зручніше винести його в зовнішній файл. Наприклад, створимо на жорсткому диску новий каталог, припустимо, C: \ node \ helloapp, в який помістимо новий файл app.js з наступним кодом:

console.log(“Hello world”)

У командному рядку перейдемо за допомогою команди cd до каталогу helloapp, а потім виконаємо команду:

node app.js

Дана команда виконає код з файлу app.js:



Якщо пізніше ви захочете додати в систему більше версій Node.js, краще почати використовувати **nvm** (node version manager) - диспетчер версій Node.js. Посилання на встановлення під Linux [https://github.com/nvm-sh/nvm#installing-and-updating](https://github.com/nvm-sh/nvm%23installing-and-updating).

Якщо використовуєте Windows то <https://github.com/coreybutler/nvm-windows>. Після його установки ви зможете використовувати дуже простої CLI API для зміни версії Node.js.

Установка різних версій Node.js

nvm install 13.8

Ви можете почати використовувати цю версію за допомогою наступної команди:

nvm use 13.8

З тих пір як був сформований Node.js Foundation, Node.js має план релізів. Це дуже схоже на інші проекти Linux Foundation. Це означає, що є два релізу: стабільний і експериментальний. У Node.js стабільними версіями з довготривалою підтримкою (LTS) є ті, які починаються з парних чисел (4, 6, 8, ...). Експериментальні версії нумеруються непарними числами (5, 7, ...).

Рекомендуємо використовувати версію LTS в продакшені і пробувати нові можливості з експериментальними версіями.

1. Перший додаток на Node.JS

Напишемо Перший простий додаток для NodeJS. Для створення додатків можна використовувати практично всі стандартні конструкції мови JavaScript. Винятком є робота з DOM, так як додаток буде запускатися на сервері, а не в браузері, тому DOM і такі об'єкти як window або document в даному випадку нам будуть недоступні.

Для цього спочатку створимо для додатка каталог на жорсткому диску. Наприклад, каталог C: \ node \ helloapp. У цьому каталозі створимо файл app.js.

Визначимо в файлі app.js наступний код:

const http = require('http');

const hostname = '127.0.0.1';

const port = 3000;

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

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hello NodeJS!');

});

server.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

У першому рядку ми отримуємо модуль http, який необхідний для створення сервера. Це вбудований модуль, і для його завантаження необхідно застосувати функцію require()

Далі за допомогою методу createServer() створюється новий сервер для прослуховування вхідних підключень і обробки запитів. Як параметр цей метод приймає функцію, яка має два параметри. Перший параметр req зберігає всю інформацію про запит, а другий параметр res використовується для відправки відповіді. В даному випадку відповідь являє просту рядок "Hello NodeJS!" і відправляється за допомогою методу res.end().

Але метод http.createServer() тільки створює сервер. Щоб сервер почав прослуховувати вхідні підключення у нього треба викликати метод listen

Цей метод приймає три параметри. Перший параметр вказує на локальний порт, по якому запускається сервер. Другий параметр вказує на локальну адресу. Тобто в даному разі сервер буде запускатися за адресою 127.0.0.1 або localhost на порту 3000.

Третій параметр представляє функцію, яка запускається при початку прослуховування підключень. Тут ця функція просто виводить діагностичне повідомлення на консоль.

Тепер запустимо сервер. Для цього відкриємо термінал (в OS X або Linux) або командний рядок (в Windows). За допомогою команди cd перейдемо до каталогу програми:

cd C:\node\helloapp

Потім викличемо наступну команду:

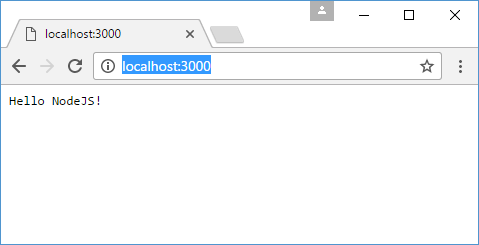
node app.js

Вона запускає сервер:

Server running at http://127.0.0.1:3000

Далі відкриємо браузер і введемо в адресну строку адресу:

http://localhost:3000



Нам відобразиться то повідомлення, яке було надіслано в методі res.end().

1. NPM. Package.json. Установка модулів. Визначення команд

Окрім вбудованих і кастомних модулів Node.js існує величезний пласт різних бібліотек і фреймворків, різноманітних утиліт, які створюються сторонніми виробниками і які також можна використовувати в проекті, наприклад, express, grunt, gulp і так далі. І вони теж нам доступні в рамках Node.js. Щоб зручніше було працювати з усіма сторонніми рішеннями, вони поширюються у вигляді пакетів. Пакет по суті являє набір функциональностей.

Для автоматизації установки і оновлення пакетів, як правило, застосовуються систему управління пакетами або менеджери. Безпосередньо в Node.js для цієї мети використовується пакетний менеджер NPM (Node Package Manager). NPM за замовчуванням встановлюється разом з Node.js, тому нічого доустанавливать не потрібно. Але можна оновити встановлену версію до самої останньої. Для цього в командному рядку/терміналі треба запустити наступну команду:

npm install npm@latest -g

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

npm -v

Для нас менеджер **npm** важливий в тому плані, що з його допомогою легко керувати пакетами. Наприклад, створимо на жорсткому диску нову папку modulesapp (В нашому випадку папка буде знаходитися по шляху C:\node\ modulesapp).

Якщо в подальшому нам більше не потрібно **express**, то ми його можемо видалити за допомогою такої команди:

npm uninstall express

Для більш зручного управління конфігурацією і пакетами додатку в npm застосовується файл конфігурації **package.json**.

*Можливо інтерактивно генерувати файл* **package.json** *за допомогою команди* **npm init** *в терміналі.*

*Після запуску команди вас попросять ввести деякі дані, наприклад ім'я вашої програми, версію, опис і так далі.*

*Не потрібно турбуватися, просто натискайте Enter, поки не отримаєте сформований JSON і питання is it ok ?.*

*Натисніть Enter в останній раз і ваш package.json був автоматично згенерований і поміщений в папку вашого застосування.*

Додамо в папку проекту modulesapp новий файл package.json:

{

  "name": "modulesapp",

  "version": "1.0.0"

}

Тут визначені тільки дві секції: ім'я проекту - modulesapp і його версія - 1.0.0. Це мінімально необхідна визначення файлу package.json. Даний файл може включати набагато більше секцій. Детальніше можна подивитися в документації (<https://docs.npmjs.com/creating-a-package-json-file>).

Далі для прикладу встановимо в проект express. **Express** представляє легкий веб-фреймворк для спрощення роботи з Node.js. В даному випадку ми не будемо поки детально розглядати фреймворк Express, так як це окрема велика тема. А використовуємо його лише для того, щоб зрозуміти, як встановлюються сторонні модулі в проект.

Для установки функціональності Express в проект спочатку перейдемо до папки проекту за допомогою команди cd. Потім введемо команду

npm install express

Після установки express в папці проекту modulesapp з'явиться підпапка **node\_modules**, в якій будуть зберігається все встановлені зовнішні модулі. Зокрема, в підкаталозі node\_modules/express будуть розташовуватися файли фреймворку Express.

І після виконання команди, якщо ми відкриємо файл package.json, то ми побачимо інформацію про пакет:

{

  "name": "modulesapp",

  "version": "1.0.0",

  "dependencies": {

    "express": "^4.17.1"

  }

}

Інформація про всі пакети що додаються, які використовуються при роботі додатка, додається в секцію **dependencies**.

Файл package.json грає велику роль і може полегшити роботу з пакетами в різних ситуаціях. Наприклад, ми плануємо використовувати безліч пакетів. Але вводити для установки кожного пакета в консолі відповідну команду не дуже зручно. У цьому випадку ми можемо визначити всі пакети в файлі package.json і потім однією командою їх встановити.

Наприклад, змінимо файл package.json наступним чином:

{

  "name": "modulesapp",

  "version": "1.0.0",

  "dependencies": {

    "express": "^4.17.1",

    "react": "^16.9.0",

    "react-dom": "^16.9.0"

  }

}

Тут додані визначення двох пакетів, які представляють бібліотеку React.

Потім, щоб завантажити всі пакетів виконати команду

npm install

Ця команда візьме визначення всіх пакетів із секцій dependencies і завантажить їх в проект. Якщо пакет з потрібною версією вже є проект, як в даному випадку express, то за новою він не завантажується.

Окрім пакетів, які застосовуються в додатку, коли воно запущено і знаходиться в робочому состояни, наприклад, express, тобто в стані "production", є ще пакети, які застосовуються при розробці програми та його тестуванні. Такі пакети, як правило, додаються в іншу секцію - **devDependencies**.

Наприклад, завантажимо в проект пакет **jest**, який використовується для тестування програми:

npm install --save-dev jest

Прапор **--save-dev** вказується, що інформацію про пакет слід зберегти саме в секції devDependencies файлу package.json:

{

  "name": "modulesapp",

  "version": "1.0.0",

  "dependencies": {

    "express": "^4.17.1",

    "react": "^16.9.0",

    "react-dom": "^16.9.0"

  },

  "devDependencies": {

    "jest": "^25.1.0"

"

  }

}

Для видалення пакетів використовується команда **npm uninstall**. наприклад:

npm uninstall express

При цьому не важливо, де розташовується інформація про пакет - в секції **dependencies** або **devDependencies**, пакет видаляється з будь-якої з цих секцій.

Якщо нам треба видалити не один пакет, а кілька, то ми можемо видалити їх визначення з файлу *package.json* і ввести команду **npm install**, і видалені з package.js пакети також будуть видалені з папки *node\_modules*.

Причому ми також можемо одночасно деякі пакети додавати в *package.json*, а деякі, навпаки, видаляти. І при виконанні команди **npm install** пакетний менеджер нові пакети встановить, а віддалені з *package.json* пакети видалить.

NPM дозволяє визначати в файлі *package.json* команди, які виконують певні дії. Наприклад, визначимо наступний файл *app.js*:

let name = process.argv[2];

let age = process.argv[3];

 console.log("name: " + name);

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

В даному випадку ми отримуємо передані при запуску додатка параметри.

І визначимо наступний файл *package.json*:

{

  "name": "modulesapp",

  "version": "1.0.0",

  "scripts" : {

    "start" : "node app.js",

    "dev" : "node app.js Tom 26"

  }

}

Тут додана секція ***scripts***, яка визначає дві команди. Взагалі команд може бути багато відповідно до цілей і завдань розробника.

Перша команда називається start. Вона по суті виконує команду ***node app.js***, яка виконує код в файлі *app.js*

Друга команда звана, ***dev***. Вона також виконує той же файл, але при цьому також передає йому два параметра.

Назви команд можуть бути довільними. Але тут треба враховувати один момент. Є умовно кажучи є зарезервовані назви для команд, наприклад, ***start, test, run*** і ряд інших. Їх не дуже багато. І якраз перша команда з вище певного файлу *package.json* називається ***start***. І для виконання подібних команд в терміналі/командному рядку треба виконати команду:

npm [название\_команды]

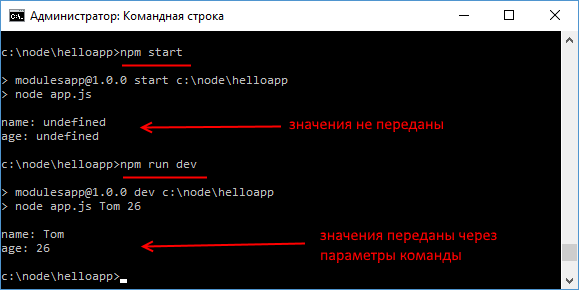
Наприклад, для запуску команди ***start:***

npm start

Команди з іншими назви, як наприклад, "***dev***" в вище вказаному файлі, запускаються так:

npm run [название\_команды]

Наприклад, послідовно виконаємо обидві команди:



В процесі розробки може знадобитися необхідність внести зміни у вже запущений проект. Припустимо, у нас в файлі *app.js* визначений наступний код:

const http = require("http");

 let message = "Hello World!";

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

    console.log(message);

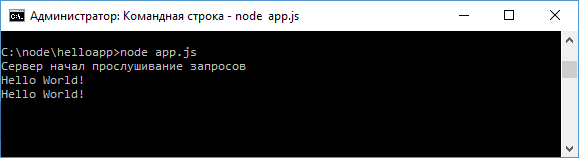
    response.end(message);

}).listen(3000, "127.0.0.1",()=>{

    console.log("Сервер начал прослушивание запросов");

});

Запустимо сервер за допомогою команди ***node app.js***, і при зверненні користувача за адресою *http://localhost:3000* браузер користувача відобразить рядок "Hello World!". Одночасно рядок виводиться на консоль.



При цьому сервер продовжує бути запущеним. І якщо ми змінимо змінну *message* в файлі *app.js*, то це ніяк не вплине на роботу сервера, і він буде продовжувати віддавати клієнту рядок *"Hello World!"*.

В цьому випадку необхідно перезапустити сервер. Однак це не дуже зручно, особливо коли необхідно часто робити різні зміни, тестувати виконання. І в цьому випадку нам може допомогти спеціальний інструмент **nodemon**.

Встановимо **nodemon** в проект за допомогою наступної команди:

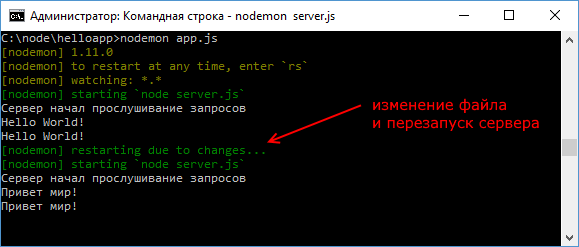
npm install nodemon -g

Прапор ***-g*** представляє скорочення від ***global*** і дозволяє встановити залежність nodemon глобально для всіх проектів на даній локальній машині.

Після установки запустимо файл *app.js* за допомогою наступної команди:

nodemon app.js

І якщо раптом після запуску сервера ми змінимо його код, наприклад, поміняємо змінну message з "Hello World!" на "Привіт світ!", то сервер автоматично буде перезапущений:



1. Фреймворк Express

Розглянемо створення серверу за допомогою фреймворка ***Express***. Здавалося б, навіщо нам потрібен додатковий фреймворк, якщо ми можемо скористатися готовим модулем http, який є в Node.js API. Однак Express сам використовує модуль http, але разом з тим надає ряд готових абстракцій, які спрощують створення сервера і серверної логіки, зокрема, обробка відправлених форм, робота з куками, CORS і т.д.

Вихідний код фреймворка можна подивитися в репозиторії на гітхабе за адресою <https://github.com/expressjs/express>.

Встановіть Express в каталозі app і збережіть його в списку залежностей. наприклад:

$ npm install express –save

Для тимчасової установки Express, без додавання його в список залежностей, не вказуйте опцію --save:

$ npm install express

Створимо в каталозі проекту новий файл ***app.js***:

// підключення express

const express = require("express");

// створюємо об’єкт додатка

const app = express();

// визначаємо обробник для маршруту "/"

app.get("/", (request, response) => {

    // відправляємо відповідь

    response.send("<h2>Привет Express!</h2>");

});

// починаємо прослуховувати підключення на 3000 порту

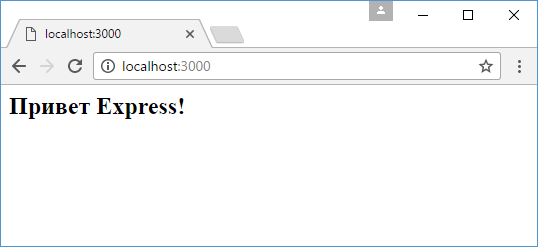
app.listen(3000);

Для обробки запитів в Express визначено ряд вбудованих функцій, і однією з таких є функція **app.get()**. Вона обробляє GET-запити протоколу HTTP і дозволяє зв'язати маршрути з певними обробниками. Для цього першим параметром передається маршрут, а другим - обробник, який буде викликатися, якщо запит до сервера відповідає даному маршруту:

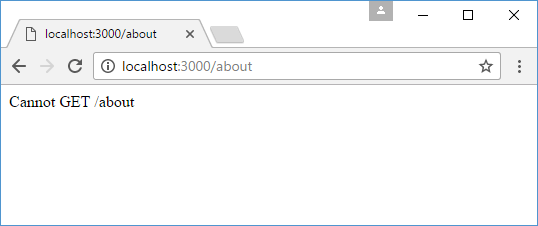
Маршрут "**/**" представляє кореневої маршрут.

Для запуску сервера викликається метод **app.listen()**, в який передається номер порту.

Запустимо проект і звернемося в браузері за адресою <http://localhost:3000>



Важливо, що Express спирається на систему маршрутів, тому всі інші запити, які не відповідають кореневому маршруту "**/**", не будуть оброблятися:



Тепер змінимо файл **app.js**:

const express = require("express");

const app = express();

app.get("/", (request, response) => {

    response.send("<h1>Главная страница</h1>");

});

app.get("/about", (request, response) => {

    response.send("<h1>О сайте</h1>");

});

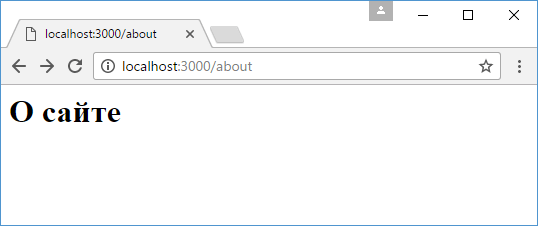
app.get("/contact", (request, response) => {

    response.send("<h1>Контакты</h1>");

});

app.listen(3000);

Тепер в додатку визначено три маршрути, які будуть оброблятися сервером:



1. Пакети для тестування Jest та SuperTest

**Jest** (<https://jestjs.io>) – це фреймворк для тестування Java Script.

Встановимо Jest за допомогою npm:

npm install --save-dev jest

Почнемо з написання тесту на гіпотетичну функцію, яка додає два числа. Спочатку створіть файл **sum.js**:

function sum(a, b) {

return a + b;

}

module.exports = sum;

Потім створіть файл з назвою **sum.test.js**. Він буде містити наш фактичний тест:

const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {

expect(sum(1, 2)).toBe(3);

});

Додайте наступний script до **package.json**:

{

"scripts": {

"test": "jest"

}

}

У консолі запустіть тест командою **jest** або **npm run test** і Jest виведе це повідомлення:

PASS ./sum.test.js

✓ adds 1 + 2 to equal 3 (5ms)

Спробуйте самостійно задати заздалегідь невірні умови для тестування, наприклад 1 + 2 = 4. Запустіть тест і отримайте негативний результат тестування.

Цей тест використовував “**expect**” і “**toB**e” для перевірки того, що два значення були абсолютно однаковими. Щоб дізнатися про інші речі, які Jest може перевірити, див. <https://jestjs.io/docs/en/using-matchers>.

**SuperTest** (<https://www.npmjs.com/package/supertest>) – це бібліотека для тестування HTTP серверів на Node.js.

Встановіть SuperTest як модуль npm та додайте його у файлі *package.json* до *devDependencies*:

npm install supertest --save-dev

Файл ***app.js*** розділимо на дві частини. Це потрібно для одночасного запуску додатка і тесту.

Файл ***server.js***

//server.js

const *express* = require('express');  
const *app* = *express*();  
  
*app*.use("/test", (req, res) => {  
res.send({message: 'pass!'});  
 res.end();  
  
});  
  
module.exports = *app*;

Файл ***start.js***

//start.js

const *app* = require('./server.js');  
  
*app*.listen(3000, () => {  
 *console*.log('Server start at localhost:3000');  
});

Створимо файл ***app.test.js***, як приклад тесту на Jest та Supertest

//app.test.js

const *app* = require('./server');  
const *supertest* = require('supertest');  
const request = *supertest*(*app*);  
  
it('gets the test endpoint', async done => {  
 const response = await request.get('/test');  
  
 expect(response.status).toBe(200);  
 expect(response.body.message).toBe('pass!');  
 done();  
});

Запустіть дві консолі одночасно. У першій консолі запустимо сервер:

node start

В іншій консолі запустимо тест:

jest app.test.js

Побачимо успішне виконання тесту

**PASS** ./app.test.js

√ gets the test endpoint (45ms)

1. Пакет DOTENV

**DOTENV** – це модуль, який завантажує змінні середовища з **.env**-файлу в **process.env**. Зберігання конфігурації в середовищі, окремої від коду, базується на методології The Twelve-Factor App.

Установка за допомогою npm:

npm install dotenv

Створимо **.env** файл у кореневому каталозі нашого проекту. Додамо змінні, що стосуються середовища, у нові рядки у вигляді NAME = VALUE. Наприклад:

// .env

HELLO = "Hello!"  
NAME = Tom  
AGE = 22

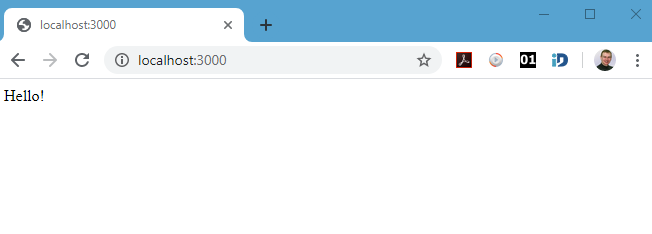
Змінемо файл ***app.js***

require('dotenv').config();  
const express = require("express");  
const app = express();  
  
app.get("/", (req, res) => {  
  
 res.send(process.env.HELLO);  
});  
  
app.get("/user", (req, res){  
  
 res.send({name: process.env.NAME, age: process.env.AGE});  
});  
  
app.listen(3000);

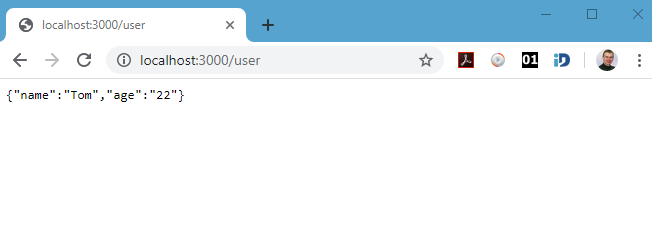
Запустимо файл у консолі

node app.js

У браузері по адресі <http://localhost:3000> побачимо наступне:



По адресі <http://localhost:3000/user> наступне:



1. Завдання для самостійної роботи
2. Встановити Node.js
3. Створити акаунт на GitHub
4. Створити репозиторій на GitHub
5. Клонувати репозиторій на ПК
6. Зайти в каталог репозиторію (проекту) на ПК
7. Ініціювати проект, вказати автора і стартовий файл
8. Додати в проект Express, Jest, supertest, dotenv
9. Програма повинна слухати порт 3000
10. Для адреси ”/” має повертатись Hello World
11. Запустити програму, побачити в браузері Hello World
12. Написати тест на Jest та supertest, який перевіряє чи дійсно сервіс повертає Hello World
13. За допомогою dotenv змінити текст відповіді (як завгодно)
14. Посилання

* Офіційний сайт Node.js

nodejs.org

* Документація по Node.js API

https://nodejs.org/dist/latest-v8.x/docs/api/

* Пошук пакетів на NPM

npmjs.com

* Офіційний сайт Express.js

http://expressjs.com/uk/

* Офіційна сторінка Jest

https://jestjs.io/ru/

* npm supertest

https://www.npmjs.com/package/supertest

* npm dotenv

https://www.npmjs.com/package/dotenv