

# 13. Node.js - czyli JavaScript poza przeglądarką

## Wyzwania:

- Dowiesz się czym dokładnie jest Node.js i do czego służy
- Opanujesz zasady działania Node'a
- Dowiesz się czym jest npm i jakie ma zastosowanie

Prosta aplikacja oparta na Node.js

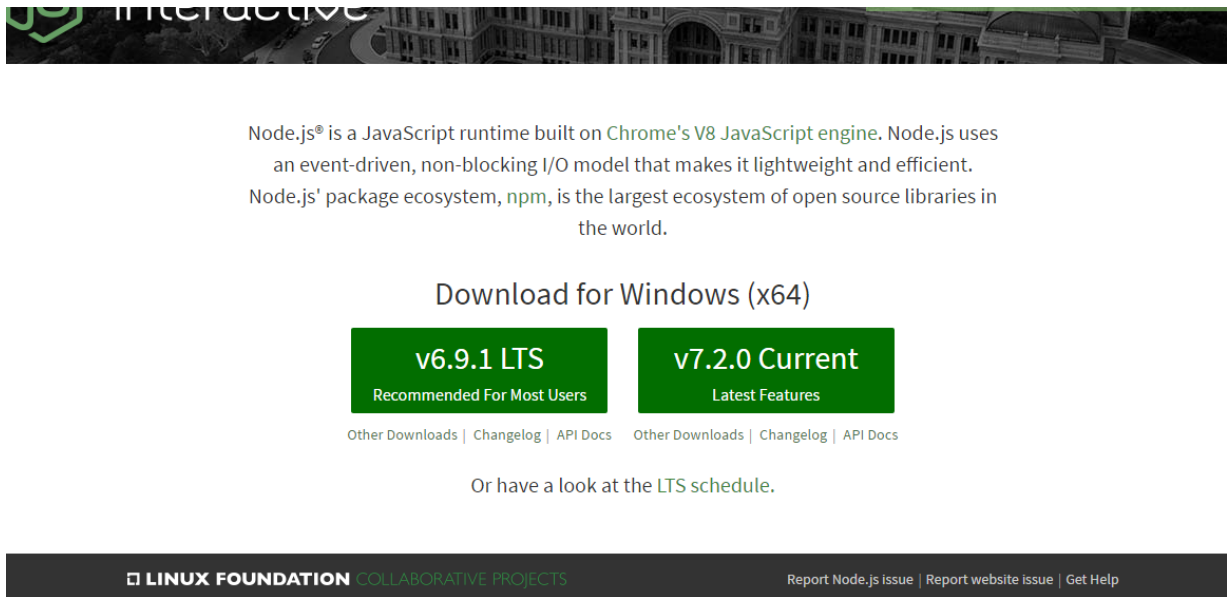
## 13.1. Czym jest NodeJS?



Zacznę od tego, że zakładam, że wiesz czym jest JavaScript albo znasz chociaż podstawy działania tego języka - jest to niezbędne przy omawianiu Node, co może dodatkowo sugerować końcówka Node.JS.

Jeśli wejdziemy na stronę główną [Node.js](https://nodejs.org), to pierwsze co zauważymy to krótki, ale treściwy opis tego czym jest ta technologia.



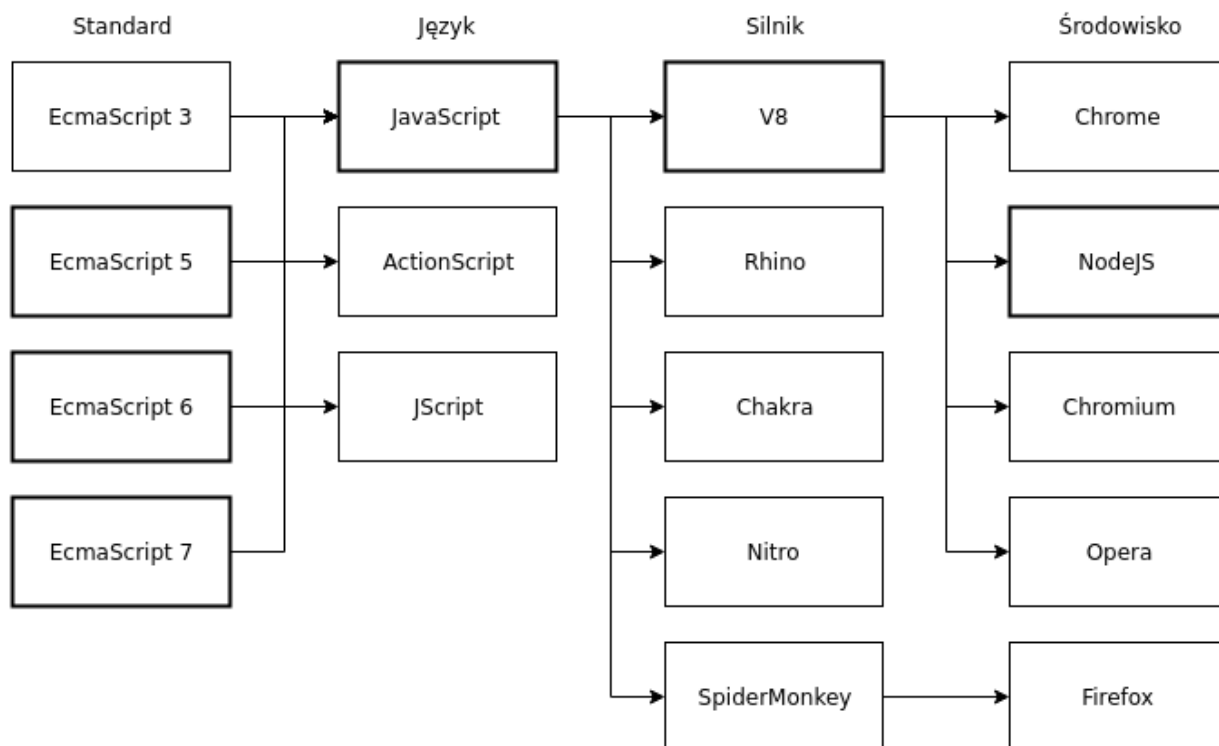


Twórcy Node'a opisują go jako środowisko uruchomieniowe zbudowane w oparciu o silnik Google Chrome (V8). - znaczy to mniej więcej tyle, że Node działa w oparciu o silnik, z którego korzysta również popularna przeglądarka firmy Google. Sam silnik został napisany w języku C++, ale zaimplementowane są w nim funkcjonalności związane z działaniem języka JavaScript.

Składnię i semantykę języka (dwa trudne słowa, które opisują każdy język programowania, ale nie tylko) opisują tzw. standardy. Z roku na rok wychodzą coraz to nowsze standardy (np. poprawia się coś nowego w semantyce albo w składni języka, co może przydać się programiście).

JavaScript to implementacja takiego właśnie standardu, który nazywa się ECMAScript - o najnowszej stabilnej wersji tego standardu (ES6) opowiem trochę w trzecim module.

Teraz wróćmy jeszcze na moment do silnika - pojawiło się tutaj dużo pojęć, więc może zaczniemy od początku - zerknijmy na mały schemat, aby zobaczyć co i jak jest ze sobą powiązane.



Mam nadzieję, że diagram wyjaśnia chociaż odrobinę zależności między środowiskiem, silnikiem, językiem i standardem. Pogrubione linie pokazują technologie z którymi powiązany jest Node.js.

Dalej twórcy określają Node w następujący sposób: Node.js używa nieblokujących modeli I/O (wejścia/wyjścia) sterowanych zdarzeniowo.

Kolejne na pierwszy rzut oka trudne sformułowanie. Jest to nic innego jak znany z przeglądarki system sterowany zdarzeniami typu:

- na kliknięcie w przycisk, wyświetl alert z napisem "Hello World"
- po załadowaniu strony, rozpocznij pracę karuzeli
- po załadowaniu danych przez AJAX, wyświetl je na ekranie użytkownika.

NodeJS również jest zdarzeniowy, tylko rodzaje tych zdarzeń są po prostu innej natury - tutaj spotykamy się raczej z następującymi sytuacjami:

- po otwarciu pliku, wyświetl jego zawartość w konsoli
- przed zapisaniem pliku posortuj elementy alfabetycznie
- kiedy użytkownik wgra plik na serwer, sprawdź czy nie ma w nim żadnych wirusów

Ok, a co oznacza *nieblokujący* model I/O?

Zatrzymajmy się na chwilę i przypomnijmy sobie różnicę między synchronicznymi i asynchronicznymi wywołaniami.

Kiedy wykonujemy coś synchronicznie, każda procedura wykonuje się sekwencyjnie zgodnie z kolejnością zapisu w kodzie.



Spójrzmy na przykład:

```
1 console.log('hello ');
2 console.log('world');
3 // wyświetli: hello world
```

Kod wykonuje się w tutaj kolejności, w jakiej został zapisany. Spójrzmy teraz na przykład asynchronicznego wywołania.

```
1 setTimeout(function(){
2     console.log(' hello');
3 }, 0);
4 console.log('world')
5 // wyświetli: world hello
```

Co oznacza, że funkcja `setTimeout()`, mimo że ma się wykonać z pozoru natychmiastowo (0 ms opóźnienia), to przechodzi na koniec kolejki wywoływań, a pierwsza wykona się funkcja wyświetlająca w konsoli słowo "world".

Założmy przez chwilę, że `setTimeout()` wykonuje się synchronicznie, ale zwiększmy czas opóźnienia na 2000 ms.

```
1 console.log('start programu');
2 setTimeout(function(){
3     console.log('hello ');
4 }, 2000);
5 console.log('world');
6 // wyświetli: start programu world <2 sekundy przerwy> hello
```

Okazuje się, że w przypadku synchronicznego wykonywania kodu nasz program zawiesiłby działanie na 2 sekundy. Inaczej mówiąc, **zablokowałby się**.

**Nieblokujące** operacje to takie, które kontynuują działanie wykonywania kolejnych funkcji bez przerywania działania samej aplikacji.

Problem, z którym najczęściej spotykają się serwery, to blokada wątku przy operacjach odczytu/zapisu I/O oraz przy operacjach sieciowych (wysyłanie/odbieranie informacji). Są to kluczowe funkcjonalności, z których korzystają serwery, a które są problemem przy implementacjach synchronicznych blokujących (takich jak chociażby PHP).



Node.js rozwiązuje ten problem dzięki znanym *callbackom*, które są po prostu funkcjami wykonującymi się na wystąpienie pewnych zdarzeń. *Callbacks* trzeba napisać samodzielnie i przekazać je jako parametr funkcji nasłuchującej na zajście pewnego zdarzenia, którym może być przykładowo:

- otworenie pliku,
- wgranie pliku przez użytkownika,
- zapisanie czegoś w bazie danych

Funkcja, którą przekazujemy w poniższym przykładzie jako parametr to jest właśnie callback. Zostanie wykonana w momencie w którym zarejestrowane zostanie zdarzenie kliknięcia w przycisk.

```
1 button.addEventListener('click', function() {  
2     alert('Hello world');  
3 });
```

Na te zdarzenia ma się wykonać pewna funkcja. Zwracam uwagę, że każda z tych operacji wymaga czasu i mogłaby zablokować działanie programu. Node został zaprojektowany z myślą o tych sytuacjach tak, aby być lekkim i wydajnym (określenie twórców).

Jest jeszcze jedna rzecz, z której twórcy Node są znani i mogą się nią pochwalić - menadżer paczek **npm**, ale o nim opowiemy sobie w późniejszym czasie :)

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✓ Zapoznałem(a) się!

## 13.2. Instalacja Node.js



Okej, wiemy już mniej więcej czym jest Node.js - pora przejść do wykorzystania go w praktyce :)



# Pobranie pakietu instalacyjnego i instalacja środowiska

Zanim zaczniemy pracę z Node musimy pobrać pakiet instalacyjny z [oficjalnej strony](#) i zainstalować program na swoim systemie operacyjnym.

Pierwszy wybór przed którym musimy stanąć to pobranie wersji *LTS* lub *Current*.

LTS (skrót od ang. *long term support*) to wersja, która jest wspierana długoterminowo. Taka wersja jest po prostu mniej podatna na zmiany i mamy pewność, że nie będą do niej dodawane nowe funkcjonalności, które mogłyby wpłynąć na kompatybilność wsteczną naszego oprogramowania.

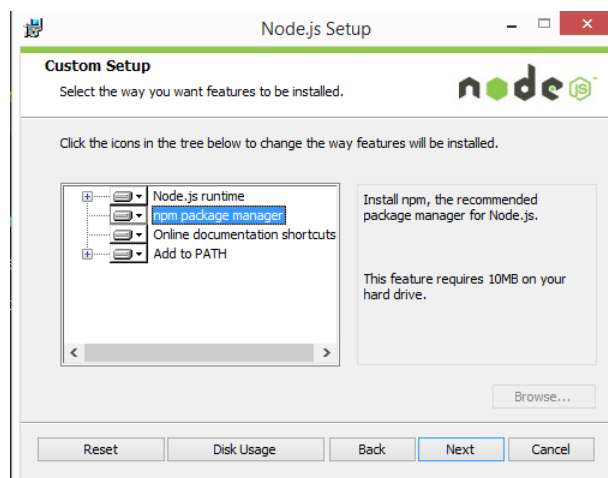
Wersja *Current* to po prostu najnowsza wersja, która może ulec w przyszłości dużym zmianom, co może spowodować konieczność poprawek również w naszym kodzie.

My wybierzemy wersję *LTS* ze względu na to, że jest bardziej stabilna od *Current*.

Kolejnym wyborem jest oczywiście system operacyjny. Dla każdego z nich proces instalacji przebiega nieco inaczej, ale jeśli korzystasz ze swojego systemu to zakładam, że wiesz jak zainstalować na nim oprogramowanie.

Instalacja przy domyślnych ustawieniach przebiega raczej bezproblemowo.

**UWAGA:** upewnij się, że opcja *Add to PATH* jest zaznaczona. PATH to jedna ze zmiennych środowiskowych, która określa miejsce katalogów w których wykonywane są programy. Dzięki zaznaczeniu tej opcji możemy uruchomić Node w terminalu bez podawania całej ścieżki do programu.



# Pierwsze spotkanie z Node

Ok, zainstalowaliśmy Node.js. Wśród programistów istnieje pewna tradycja, która sprowadza się do napisania w nowym języku aplikacji "Hello World". Co prawda zakładam, że znasz JavaScript, ale możesz nie wiedzieć w jaki sposób odpalić taki skrypt w środowisku Node.js. Przejdźmy do tworzenia tej prostej aplikacji:

1. Stwórz plik `index.js`
2. Dodaj w nim linijkę: `console.log('Hello World');`
3. W zależności od systemu operacyjnego otwórz terminal, jeśli używasz Mac OS X lub Linuxa. W przypadku Windowsa otwórz dowolny terminal (jeśli nie wybrałeś opcji *Add to PATH* podczas instalacji, otwórz Node Command Window, który jest dostępny razem z Node).
4. przejdź do katalogu, który w którym zapisany jest plik `index.js`
5. uruchom skrypt za pomocą komendy: `node index.js`

Jeśli na ekranie widnieje napis "Hello World", to znaczy, że Node.js został zainstalowany poprawnie i możesz przejść do kolejnego submodułu, w którym wyjaśnimy sobie, z jakich funkcjonalności Node'a możemy korzystać.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✓ Zapoznałem(a)m się!

## 13.3. Wprowadzenie do Node.js



Jak już wspomniałem, natury aplikacji przeglądarkowych i tych zbudowanych w Node są zupełnie inne. W aplikacjach przeglądarkowych programista ma do dyspozycji takie obiekty globalne jak np. *window* czy *document*. Służą one jako reprezentacja okna przeglądarki i widocznego w nim dokumentu. Dzięki nim można odczytać wartości atrybutów tych elementów jak np. wielkość okna, lub też dokonać na nich pewnych modyfikacji jak chociażby dodanie nowego fragmentu do strony dzięki reprezentacji DOM.

W Node.js istnieją obiekty globalne, których nie spotkamy w przeglądarce. Jednym z nich jest przykładowo *buffer*, który jest heksadecymalną (szesnastkową) reprezentacją danych. Inną zmienną globalną jest zmienna globalna... *global*! Pamiętajmy, że zmienne



globalne w Node.js różnią się znacznie od tych z przeglądarki. Jest to spowodowane charakterem środowiska jakim jest Node, który nie ma obiektu globalnego *window*, ponieważ nie jest przeglądarką taką jak Chrome czy Firefox. Kolejnym przykładem obiektu globalnego jest obiekt *process*, który służy jako 'most' między aplikacją, a środowiskiem w jakim aplikacja została uruchomiona. Istnieje wiele innych obiektów globalnych, które omówimy sobie w tym module, ale zacznijmy najpierw od obiektu *global*.

---

## Global

---

Obiekt *Global* to jeden z fundamentalnych obiektów Node. W przeglądarce zmienne deklarowane na najwyższym poziomie były zmiennymi globalnymi. Przykładowo, mając dwa skrypty załadowane w przeglądarce jeden po drugim:

```
<script src="script1.js"></script>
<script src="script2.js"></script>
```

W tym przypadku zawartość skryptu *script1* to:

```
var greeting = "hello world"
```

Natomiast zawartość skryptu *script2* to:

```
console.log(greeting);
```

Nasz skrypt będzie wiedział czym jest zmienna *greeting* mimo, że zmienna i jej wykorzystanie są w różnych plikach.

Gdybyśmy teraz mieli napisać ten sam skrypt w Node.js, wyglądałby on następująco:

```
//script1.js
exports.greeting = 'hello world'
```





```
//script2.js  
var greeting = require('./script1').greeting;  
console.log(greeting);
```

Każdy plik w Node.js ma osobny zasięg globalny widoczny tylko w tym konkretnym pliku. W celu udostępnienia go do przestrzeni "publicznej" musimy wyeksportować zmienne, które nas interesują. Możemy to zrobić przypisując interesujące nas informacje do obiektu *exports*. W przykładowym skrypcie o nazwie *script1* nasze powitanie stało się dzięki temu widoczne dla innych plików.

Aby użyć wyeksportowanej zmiennej w innym pliku, należy zawrzeć inny moduł za pomocą słowa kluczowego *require*, a następnie odwołać się do importowanej wartości (*greeting*).

Więcej o tym zapisie dowiesz się w kolejnym submodule opisującym moduły.

Wiemy już, że zmienne globalne w Node działają nieco inaczej niż w przeglądarce. Co w takim razie kryje się pod zmienną *global*? Łatwo jest to sprawdzić pisząc prosty skrypt:

```
// index.js  
console.log(global)
```

Na ekranie powinny wyświetlić się nam obiekty i funkcje Node.js, wliczając w to zmienne globalne wymienione nieco wcześniej. Można zaryzykować stwierdzenie, że obiekt *global* jest to odpowiednik obiektu *window* w przeglądarce.

---

## Process

---

Jest to obiekt, który jest również dostępny globalnie. Dostarcza on informacji na temat procesu, w ramach którego wykonywana jest dana aplikacja. Nie trzeba się do niego odwoływać za pomocą **global.process**. Działa to na podobnej zasadzie co obiekt *document*, do którego nie trzeba dopisywać **window.document**.

Zobaczmy co kryje w sobie kilka z parametrów tego obiektu:

```
// console.log(process.versions);
{ http_parser: '2.7.1',
  node: '6.5.0',
  v8: '5.1.281.81',
  uv: '1.9.1',
  zlib: '1.2.8',
  ares: '1.10.1-DEV',
  icu: '57.1',
  modules: '48',
  openssl: '1.0.2h'
}
```

`process.versions` przechowuje informacje na temat samego Node oraz komponentów i zależności, włączając w to silnik V8, na którym Node jest zbudowany.

Zobaczmy co możemy znaleźć w parametrze `process.env`:

```
{
  LANG: 'pl_PL.utf8',
  WEBSITE_JDK: '/usr/lib/jvm/default',
  SHLVL: '3',
  XDG_SEAT: 'seat0',
  HOME: '/home/hello',
  LOGNAME: 'hello',
}
```

Są to informacje na temat naszego środowiska, czyli tego co Node.js widzi jako otoczenie, w którym został uruchomiony. Obiekt *process* pozwala nie tylko na podgląd informacji o środowisku, ale także na komunikację z nim. Te informacje oczywiście mogą się nieco różnić w związku z tym, że cały czas wychodzą nowe wersje oprogramowania, ale nie jest to nic złego.

### Kanały komunikacji ze środowiskiem

Do bezpośredniej komunikacji ze środowiskiem, Node używa trzech kanałów komunikacyjnych:

- `process.stdin` - standard input służący do odczytu (przyjmowania informacji z zewnątrz do aplikacji)
- `process.stdout` - standard output służący do wypisywania komunikatów z procesu
- `process.stderr` - standard służący do informowania o błędach

Te trzy kanały służą do bezpośredniej komunikacji programisty z aplikacją. Stanowią interfejs, dzięki któremu możemy wysłać wiadomości do programu i odczytywać jego komunikaty. To tyle jeżeli chodzi o teorię. Spróbujmy napisać prostą aplikację.



# Prosta aplikacja oparta na Node.js

Zacznijmy od napisania aplikacji, która kończy swoje działanie, jeśli użytkownik wpisze polecenie `/exit`. W przypadku podania innego polecenia - wypisze informację "błędne polecenie".

## Przyjmowanie danych od użytkownika

Stwórz nowy plik `program.js`. Zacznijmy od ustawienia odpowiedniego enkodowania przyjmowanych danych. Bez tego informacje, które przekazujemy do aplikacji będą odczytywane jako dane szesnastkowe (potraktuje wejście jako *buffer*). Poprawne enkodowanie zapewnia, że program "zrozumie" co do niego mówimy (odczyta wartość jako string z kodowaniem UTF-8).

```
process.stdin.setEncoding('utf-8');
```

Następnym krokiem będzie ustawienie nasłuchiwanie na zdarzenia odczytu.

```
process.stdin.on('readable', function() {  
    // tutaj treść tego co ma się wykonać w momencie odczytania wejścia  
});
```

Powyższy kod można odczytać w następujący sposób: na zdarzenie (`.on`) odczytu (`readable`), masz wykonać funkcję (`function...`).

## Zwracanie danych od użytkownika

Na początek spróbujmy zrobić proste echo, czyli wyświetlić to co wpisaliśmy w aplikację.

```
process.stdin.on('readable', function() {  
    // metoda .read() ma za zadanie odczytać co użytkownik podał na w  
    var input = process.stdin.read();  
    process.stdout.write(input);  
});
```

Niestety, naszym oczom ukaże się następujący błąd:



```
1. fish /Users/kipier/Documents/nodejs (fish)
kipier@MBP-Kiper ~/D/nodejs> ls
index.js      modules      node_modules package.json program.js
kipier@MBP-Kiper ~/D/nodejs> node index.js
net.js:658
    throw new TypeError(
          ^
TypeError: Invalid data, chunk must be a string or buffer, not object
    at WriteStream.Socket.write (net.js:658:11)
    at ReadStream.<anonymous> (/Users/kipier/Documents/nodejs/index.js:7
:17)
    at emitNone (events.js:86:13)
    at ReadStream.emit (events.js:185:7)
    at emitReadable_ (_stream_readable.js:432:10)
    at emitReadable (_stream_readable.js:426:7)
    at ReadStream.Readable.read (_stream_readable.js:290:7)
    at ReadStream.Socket.read (net.js:308:43)
    at nReadingNextTick (_stream_readable.js:712:8)
    at _combinedTickCallback (internal/process/next_tick.js:71:11)
kipier@MBP-Kiper ~/D/nodejs>
```

Nasza aplikacja spodziewa się na wyjściu tylko postaci tekstowej (*string*), ale jako że nie otrzymuje niczego, to próbuje wyświetlić nam na wyjściu wartość *null*. Musimy zabezpieczyć się przed takim działaniem opakowując funkcję odczytu instrukcją warunkową sprawdzającą, czy na wejściu cokolwiek istnieje.

```
process.stdin.on('readable', function() {
    // metoda .read() ma za zadanie odczytać co użytkownik podał na w
    var input = process.stdin.read();
    if(input !== null) {
        // teraz jest sens cokolwiek wyświetlać :)
        process.stdout.write(input);
    }
});
```

Po wpisaniu dowolnej wiadomości i wciśnięciu klawisza *enter* naszym oczom powinno ukazać się echo wpisanej wiadomości. Oznacza to, że wyświetlanie wiadomości w konsoli działa.

## Tworzenie własnej 'komendy' w kodzie

Dodajmy teraz warunek sprawdzający, czy komenda ma odpowiednią wartość i spróbujmy zwrócić użytkownikowi dane, które wpisał lub zakończyć działanie aplikacji jeśli dane będą równe `/exit`.



```
process.stdin.on('readable', function() {
  // metoda .read() ma za zadanie odczytać co użytkownik podał na w
  var input = process.stdin.read();
  if (input !== null) {
    var instruction = input.toString().trim();
    if (instruction === '/exit') {
      process.stdout.write('Quitting app!\n');
      process.exit();
    } else {
      process.stdout.write('Wrong instruction!\n');
    }
  }
});
```

Zastanawiasz się pewnie do czego służy metoda `.trim()`. Otóż dzięki niej pozbywamy się wszystkich białych znaków z przodu i za tekstem. Znikają wszystkie spacje, tabulatory, enter - pozostaje sam czysty tekst. Wynikiem działania tej funkcji `'hello'.trim()` będzie: `hello`.

## Logowanie błędów w konsoli

Jest jeszcze jedna rzecz, która wymaga drobnej korekty - obecnie, nasza aplikacja przekierowuje wszystkie odpowiedzi do strumienia `stdout` - nawet te z błędami. Lepiej będzie jak użyjemy dedykowanego rozwiązania do przekazywania komunikatów o błędach - `process.stderr`. Aby to zrobić, wystarczy, że podmienimy `process.stdout.write('Wrong instruction!');` na `process.stderr.write('Wrong instruction!');`

Po co to zrobiliśmy? Otóż możemy chcieć zapisywać wyniki działania aplikacji do osobnego pliku, ale komunikaty o błędach chcielibyśmy, aby pojawiały się w konsoli. Dzięki odpowiednim strumieniom w prosty sposób możemy kierować przepływem informacji.

---

## Zadanie: Badamy środowisko!

---

Dopisz do kodu instrukcje, które przedstawią informację o wersji Node, z której korzysta użytkownik oraz wyświetlą język systemowy użytkownika (dla macOS i Linux). Skorzystaj z wcześniej poznanego `process.env`.

Podpowiedź: zamień `if`'a na `switch statement` zamiast pisać `if, else if, else...` :)



Po wykonaniu zadania umieść je na Githubie i prześlij link do rozwiązania swojemu mentorowi.

Podgląd zadania

<https://github.com/0na/>

Wyślij link ✓

## 13.4. Wprowadzenie do modułów



Na początek wyjaśnijmy sobie czym jest moduł. Wiele z obiektów życia codziennego jest zbudowanych z tak zwanych modułów - weźmy dla przykładu komputer. Typowa jednostka centralna komputera jest zbudowana z tzw. komponentów (modułów).

Moduły komputerowe spełniają pewne określone funkcje:

- karta graficzna ma za zadanie przetworzyć pewne dane i przesać wynik obliczeń do monitora
- procesor ma za zadanie odebrać pewną porcję danych, wykonać na nich pewne obliczenia, a następnie zwrócić wynik dla np. następnych obliczeń
- pamięć ma za zadanie przechowywać dane, które są w niej umieszczane, a następnie odwoływać się do nich w razie potrzeby
- zasilacz odbiera pewne źródło prądu przemiennego, dokonuje stabilizacji napięcia dokonując na nim pewnych przekształceń, a następnie przekazuje je dalej do innych komponentów komputera, które odbierają prąd stały o określonym napięciu

Jak widzisz, każdy z komponentów komputera spełnia pewne zadanie (funkcjonalność). Dodatkowo, każdy z nich posiada wejście i wyjście (interfejs). Ważną cechą modułu jest to, że można go dowolnie wymienić i wykorzystać w innym komputerze. Jest to bardzo ważna cecha modułów.

W programowaniu, moduł ma dokładnie takie samo znaczenie jak w obiektach fizycznych. W tym miejscu odsyłam do bardzo krótkiego artykułu na [Wikipedii](#), który wyczerpująco wyjaśnia tę kwestię.



Wiele języków programowania posiada wbudowany mechanizm obsługujący moduły. JavaScript niestety nie wspiera modułów natywnie tak jak inne języki, dlatego programiści wymyślili swoje implementacje modułów.

Do najpopularniejszych należą:

- CommonJS
- AMD (*ang. Asynchronous Module Definition*)
- Native JS (ES6 modules)

Moduły AMD wychodzą poza zakres omawianego tematu, a na temat modułów ES6 powiemy sobie w 3. rozdziale. Zainteresowanych odsyłam do [tego](#) wpisu, ale uprzedzam - jeśli nie wiesz czym jest ES6, wstrzymaj się do 3-go modułu. Teraz zajmiemy się modułami **CommonJS**.

---

## CommonJS

---

To format, który w swoim pierwotnym zamyśle miał służyć do importowania i eksportowania modułów JavaScriptowych po stronie serwera. W trakcie tego kursu będziemy korzystać z kilku wbudowanych modułów, które zostały dołączone razem z pakietem instalacyjnym Node'a. W celu wykorzystania ich w swojej aplikacji, należy użyć globalnej komendy `require()`. Wygląda to następująco:

```
var module = require('nazwa-modułu');
```

Takiego zapisu używa się, jeśli chcemy dołączyć do naszej aplikacji moduł zainstalowany za pomocą menadżera paczek (o którym jeszcze wspomnimy), albo wbudowanego modułu wewnętrznego (którego nie trzeba nigdzie pobierać).

---

## Korzystanie z modułów

---

Napiszmy prostą aplikację, która korzysta z modułu `os`. Moduł ten dostarcza różne użyteczne metody związane z systemem operacyjnym. Aplikacja będzie rozszerzeniem poprzedniej i ma za zadanie wyświetlić informacje na temat komputera z którego korzysta użytkownik w czytelnej formie.



Na początku musimy zaimportować ów moduł za pomocą następującego fragmentu kodu: `var os = require('os');`

Moduły najlepiej importować na samej górze pliku, żebyśmy od razu wiedzieli z jakich modułów korzystamy.

Od tej chwili możemy używać jego funkcjonalności. Dobry moduł musi mieć swoją dokumentację. Bez niej bardzo ciężko się pracuje. Z resztą każdy programista codziennie w trakcie swojej pracy czyta jakąś dokumentację (a czasami nawet sam ją tworzy). Dokumentację do omawianego modułu znajdziesz w [tym](#) miejscu.

Ok, wracając do naszej aplikacji - powinna ona do tej pory wyglądać mniej więcej tak:

```
var os = require('os');

process.stdin.setEncoding('utf-8');
process.stdin.on('readable', function() {
  var input = process.stdin.read();
  if(input !== null) {
    var instruction = input.trim();
    switch(instruction) {
      case '/exit':
        process.stdout.write('Quitting app!\n');
        process.exit();
        break;
      case '/sayhello':
        process.stdout.write('hello!\n');
        break;
      default:
        process.stderr.write('Wrong instruction!\n');
    }
  }
});
```

Dopiszmy teraz do niej prostą komendę `/getOSinfo`, która wyświetli nam na ekranie:

```
/getOSinfo
System: OSX
Release: 15.5.0
CPU model: Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
Uptime: ~ 3007 min
User name: Kiper
Home dir: /Users/Kiper
```



Zacznijemy od dopisania kolejnej opcji instrukcji switch, która poprawnie zinterpretuje zadaną komendę.

```
case '/sayhello':
    process.stdout.write('hello!\n');
    break;
case '/getOSinfo':
    process.stdout.write('Tutaj będzie info o systemie!\n');
    break;
```

Kiedy wpiszymy komendę `/getOSinfo` na naszym ekranie powinna pojawić się wiadomość: 'Tutaj będzie info o systemie!'

Zamieńmy ją na bardziej sensowną informację i zamiast kodu

`process.stdout.write('Tutaj będzie info o systemie!\n');` umieśćmy kod, który znajduje się poniżej. Zacznijmy od wyświetlenia systemu operacyjnego i jego wersji (tak jak na powyższym obrazku):

```
var type = os.type();
var release = os.release();
if(type === 'Darwin') {
    type = 'OSX';
} else if(type === 'Windows_NT') {
    type = 'Windows';
}
console.log('System:', type);
console.log('Release:', release);
```

**UWAGA:** zauważ, że używam zamiennie `process.stdout` i `console.log`. Nie są to jednak funkcje, które działają tak samo. Różnią się jedynie tym, że `console.log` zawsze stawia na końcu znak nowej linii (`\n`). Można poniekąd przyjąć, że:

```
console.log = function(d) {
    process.stdout.write(d + '\n');
};
```

Wracając do naszej mini aplikacji, dlaczego napisaliśmy taki fragment kodu? W dokumentacji modułu OS metoda `type` może zwrócić trzy wartości:

- Darwin - dla systemów Apple (takich jak OS X)
- Linux - dla systemów Linuxowych
- Windows\_NT - dla systemów Windows



Darwin i Windows\_NT - są to tzw. podstawy systemów operacyjnych. Dla zwykłego użytkownika mogą być średnio czytelne, więc zamieniamy je na ich odpowiedniki - OS X dla Darwin'a i Windows dla Windows\_NT (tutaj bardziej chodzi o kwestię estetyki).

Teraz dodajmy informację o procesorze. Tutaj także jest mała pułapka - spójrzmy w dokumentację:

## os.cpus()

Added in: v0.3.3

The `os.cpus()` method returns an array of objects containing information about each CPU/core installed.

Okazuje się, że metoda `.cpus()` zwraca informację o wszystkich rdzeniach w postaci tablicy "procesorów", która wygląda następująco:

```
[
  {
    model: 'Intel(R) Core(TM) i7 CPU           860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 252020,
      nice: 0,
      sys: 30340,
      idle: 1070356870,
      irq: 0
    }
  },
  {
    model: 'Intel(R) Core(TM) i7 CPU           860  @ 2.80GHz',
    speed: 2926,
    times: {
      user: 306960,
      nice: 0,
      sys: 26980,
      idle: 1071569080,
      irq: 0
    }
  },
]
```

Nam potrzebna jest tylko informacja o modelu procesora. Wystarczy więc, że po pobraniu informacji o procesorach odniesiemy się do np. pierwszego elementu tablicy, a konkretniej do jego modelu:



```
var cpu = os.cpus()[0].model;  
console.log('CPU model:', cpu);
```

Kolejnym elementem informacji o systemie będzie czas jego działania (od ostatniego uruchomienia).

W dokumentacji jest napisane, że czas podawany jest w sekundach. Nam nie zależy na aż tak dokładnym szacunku. Wystarczy, że wyświetlimy czas w minutach:

```
var uptime = os.uptime();  
console.log('Uptime: ~', (uptime / 60).toFixed(0), 'min');
```

Jak widzisz podzieliliśmy czas na 60, a następnie zaokrągliliśmy uzyskany wynik do 0 miejsc po przecinku.

Ostatnim krokiem jest wyciągnięcie informacji na temat użytkownika systemu.

Odnosząc się do dokumentacji widzimy, że metoda `.userInfo()` zwraca obiekt, z różnymi informacjami. Nas interesują tylko dane związane z nazwą użytkownika i lokalizacją jego katalogu domowego:

```
var userInfo = os.userInfo();  
console.log('User name:', userInfo.username);  
console.log('Home dir:', userInfo.homedir);
```

Dane te znajdziemy odwołując się do odpowiednich kluczy obiektu `userInfo` (`username` i `homedir`).

Nasza instrukcja wygląda dosyć topornie. W case `/getOSInfo` znajduje się około 17 linii kodu - to zdecydowanie za dużo.

Nasz kod wyglądałby o niebo lepiej, jeśli wyciągniemy zawartość `/getOSInfo` do osobnej funkcji:

Napiszmy ją na samym dole pliku i przenieśmy do niej kawałek kodu z case'a:



```
function getOSInfo() {
  var type = os.type();
  if(type === 'Darwin') {
    type = 'OSX';
  } else if(type === 'Windows_NT') {
    type = 'Windows';
  }
  var release = os.release();
  var cpu = os.cpus()[0].model;
  var uptime = os.uptime();
  var userInfo = os.userInfo();
  console.log('System:', type);
  console.log('Release:', release);
  console.log('CPU model:', cpu);
  console.log('Uptime: ~', (uptime / 60).toFixed(0), 'min');
  console.log('User name:', userInfo.username);
  console.log('Home dir:', userInfo.homedir);
}
```

Teraz wywołajmy funkcję:

```
case '/getOSInfo':
  getOSInfo();
break;
```

Jest dużo czyściej, prawda? Tak, ale nasz plik zaczyna się coraz bardziej rozrastać. W zasadzie to moglibyśmy wydzielić część kodu odpowiedzialnego za generowanie informacji o systemie do **własnego modułu**. Jest to niezależna funkcjonalność i możemy ją śmiało wynieść w osobne miejsce.

---

## Tworzenie własnych modułów

---

Własne moduły można tworzyć na wiele sposobów. Nie przedłużając, przejdźmy od razu do rozwiązania zadania z poprzedniego przykładu, wyjaśniając przy okazji w jaki sposób można eksportować moduły:

Pierwszym krokiem będzie stworzenie osobnego pliku z modulem. Plik może się nazywać np. *OSInfo.js*. Następnie będzie trzeba przenieść funkcję `getOSInfo()` oraz import modułu `os` do stworzonego pliku. Na samym końcu dodamy do naszego pliku następującą linię: `exports.print = getOSInfo;`



Tutaj zatrzymamy się na chwilę. Wydzielony moduł może przybierać kilka postaci. Jeśli eksportujemy pojedynczą funkcję, możemy przypisać ją w powyższy sposób. Podobnie (co za chwilę pokażemy) możemy wyeksportować więcej funkcji. Musimy je wyeksportować, ponieważ wszystko co znajduje się wewnątrz pliku jest widoczne tylko dla kodu znajdującego się w tym pliku (działa to podobnie jak zakres funkcji). Z poziomu innych plików (np. naszego *program.js*) będziemy wywoływać nasz moduł za pomocą `require()`, ale widoczne będzie tylko to, co zostało wyeksportowane - innymi słowy, wszystko z naszego modułu, z czego chcemy skorzystać w innym pliku, musi być przypisane do obiektu `exports`.

W powyższym przykładzie zależy nam na udostępnieniu funkcji `get0Sinfo`. Standardowo moduł ma czytelnie komunikować czym się zajmuje - dlatego w naszym przypadku zarówno nazwa pliku jak i modułu to `0Sinfo`. Nie miałoby jednak sensu wywoływanie naszej funkcji za pomocą odwołania `0Sinfo.get0Sinfo` i z tego względu nazwą eksportowanej funkcji będzie `print()` - to dlatego zastosowaliśmy `exports.print = get0Sinfo;`.

Po wyeksportowaniu i zaimportowaniu w innym module, będziemy korzystać z niej w następujący sposób:

```
0Sinfo.print();
```

Możesz to sobie skojarzyć z przypisywaniem eksportowanych wartości do zmiennej, którą 'wyciągasz' z modułu za pomocą `require()` i później możesz ją normalnie wykorzystywać - mam nadzieję, że pamiętasz, że wartościami w zmiennych w JavaScriptcie mogą być także funkcje :)

Zależało mi w tym miejscu, aby pokazać, że to co znajduje się po lewej stronie znaku równości jest **widoczne** w innych modułach:

```
exports.print = get0Sinfo;
```

Całość powinna wyglądać tak:



```
var os = require('os');

function getOSInfo() {
  var type = os.type();
  if(type === 'Darwin') {
    type = 'OSX';
  } else if(type === 'Windows_NT') {
    type = 'Windows';
  }
  var release = os.release();
  var cpu = os.cpus()[0].model;
  var uptime = os.uptime();
  var userInfo = os.userInfo();
  console.log('System:', type);
  console.log('Release:', release);
  console.log('CPU model:', cpu);
  console.log('Uptime: ~', (uptime / 60).toFixed(0), 'min');
  console.log('User name:', userInfo.username);
  console.log('Home dir:', userInfo.homedir);
}

exports.print = getOSInfo;
```

Jeśli chcielibyśmy eksportować więcej niż jedną część z jednego pliku, należy albo użyć konstrukcji.

```
exports.print = getOSInfo;
exports.getCPUDetails = getCPUDetails;
```

albo

```
module.exports = {
  print: getOSInfo,
  getCPUDetails: getCPUDetails
};
```

Oba zapisy są równoznaczne przy czym **exports** i **module.exports** to dwa różne obiekty (dwie różne referencje) i trzeba ich używać zgodnie z powyższymi przykładami.

## Korzystanie z modułów



Ostatnim krokiem jest skorzystanie z utworzonego wcześniej modułu. Wystarczy, że dostarczymy moduł do naszej aplikacji za pomocą funkcji `require()`. Robiliśmy to już na przykładzie modułu `os`. Jednak tym razem, odwołanie do modułu będzie wyglądało nieco inaczej:

```
var OSInfo = require('./OSInfo');
```

Jak widzisz, dostęp do modułu określa się nieco inaczej niż poprzednio - omówimy tę różnicę za chwilę :)

Teraz możemy skorzystać z funkcji wyeksportowanej z naszego modułu `OSInfo.js` za pomocą `OSInfo.print()`;

Jeśli wiesz jak korzystać z terminala na pewno zetknąłeś się już z takim zapisem. Powyższy zapis jest przykładem relatywnej ścieżki dostępu. Znak `./` oznacza folder, w którym znajduje się plik, który próbuje odwołać się do modułu.

Zauważ, że nie trzeba dodawać rozszerzenia przy zaciąganych modułach - wystarczy, że podamy nazwę pliku, a Node doda końcówkę za nas.

```
.
├── OSInfo.js
└── program.js
```

## Zmiana struktury plików

Wprowadźmy jednak trochę porządku w naszym projekcie. Stórz katalogi *modules* i *app*, a następnie przenieś główny plik (*program.js*) do katalogu *app*, a plik *OSInfo.js* do katalogu *modules*. Musimy teraz zmienić ścieżkę importu dla modułu *OSInfo.js* w pliku *program.js*.

Dla takiej sytuacji struktura katalogów wygląda następująco:

```
.
├── app
│   └── program.js
└── modules
    └── OSInfo.js
```

Relatywna ścieżka dostępu w pliku *program.js* będzie wyglądała następująco:



```
var OSinfo = require('../modules/OSInfo');
```

Przeanalizujemy jak działa ta ścieżka relatywna. Aby z pliku *program.js* dostać się do *OSinfo.js* należy:

1. wyjść z katalogu app do jego katalogu-rodzica (`../`)
2. wejść do katalogu modules (`modules/`)
3. odwołać się do odpowiedniego modułu (`OSInfo`)

## Różnica pomiędzy wczytywaniem modułu własnego, a zainstalowanego z Node

Istnieje różnica między ładowaniem modułów zainstalowanych wraz z Node albo paczek pobranych z internetu - tutaj odnosimy się bezpośrednio do nazwy modułu, np.:

```
require('http'); // paczka, która jest dostępna po zainstalowaniu node  
require('colors'); // paczka pobrana z internetu za pomocą npm (o kt
```

Niezależnie od tego w jaki sposób dołączamy moduły do projektu, mają one jasno określone zadania i pozwalają na reużywanie kodu. Pamiętaj, że jeśli zauważysz, że Twój kod staje się obszerny i można wydzielić pewne elementy, które będą niezależne od reszty to jest to znak, że należy stworzyć moduł.

---

## Zadanie: Kształtujemy czas

---

Stwórz moduł, który będzie poprawnie formatował czas. Zależy nam na napisaniu funkcji:

- do przekształcania sekund na minuty - przykładowo, podając 125 sekund wyświetli 2 min. 5 sek.
- sekund na godziny - podając 3700 sekund wyświetli 1 godz. 1 min. 40 sek.
- dodaj moduł do katalogu *modules* i skorzystaj z niego w module *OSinfo* do poprawnego formatowania czasu działania systemu

Po wykonaniu zadania, umieść swoje rozwiązanie na Githubie i wyślij link do repozytorium mentorowi.

Podgląd zadania





<https://github.com/0na/>

Wyślij link ✓

## 13.5. npm, czyli jak nie wymyślać koła na nowo

Do tej pory tworzyliśmy i korzystaliśmy z modułów albo stworzonych przez nas samych albo tych dodanych do pakietu instalacyjnego Node'a. W tym submodule opowiemy sobie trochę o paczkach tworzonych przez społeczność i o popularnym menedżerze pakietów npm.

Szybki wzrost popularności Node'a spowodował utworzenie społeczności, która w ramach tworzenia kolejnych projektów podobnie jak my zaczęła zauważać w swoim kodzie pewne fragmenty, które można było wydzielić i udostępniać innym ludziom.

Były to gotowe rozwiązania popularnych problemów takich jak: lepszy interfejs dla pobierania plików, szybsze tworzenie serwerów bez zbędnego narzutu (tzw. boilerplate code), bezproblemowe interfejsy umożliwiające połączenie z bazą danych czy nawet gotowe narzędzia do budowania aplikacji mobilnych przy użyciu technologii webowych!

Takich rozwiązań dla konkretnego problemu było bardzo wiele. Problem pojawił się w momencie, kiedy trzeba było zarządzać takimi paczkami. Każda z nich miała swoją wersję na repozytorium i w przypadku chociażby poprawy pewnych błędów trzeba było uaktualniać paczkę na nowo ręcznie.

W ten sposób pewna grupa programistów wpadła na pomysł stworzenia npm, czyli narzędzia do zarządzania tymi paczkami w projekcie. Początkowo rozwiązanie to było oddzielone od Node, jednak w późniejszym czasie projekt został włączony na stałe do instalatora Node - teraz npm nie wymaga już oddzielnej instalacji.

Aby upewnić się, że mamy zainstalowanego npm'a, wystarczy wpisać w terminal komendę **npm --version**



```
→ nodejs npm --version  
3.10.3  
→ nodejs |
```

---

## Tworzenie projektu z npm

---

Przejdziemy teraz do tworzenia projektu we 'właściwy' sposób, jaki przyjęt się w ekosystemie Node - utwórz sobie jakiś folder na kod w wygodnym miejscu i zaczynamy :)

W celu rozpoczęcia pierwszego projektu przy użyciu npm, w terminalu należy wpisać komendę `npm init`

Naszym oczom powinien ukazać się podobny widok:

```
→ nodejs npm init  
This utility will walk you through creating a package.json file.  
It only covers the most common items, and tries to guess sensible defaults.  
  
See `npm help json` for definitive documentation on these fields  
and exactly what they do.  
  
Use `npm install <pkg> --save` afterwards to install a package and  
save it as a dependency in the package.json file.  
  
Press ^C at any time to quit.  
name: (nodejs) |
```

Jest to kreator pliku `package.json`, który będzie zawierał informacje na temat naszego projektu. O tych informacjach powiemy sobie nieco później.

Teraz musimy odpowiedzieć na kilka pytań:

```
→ nodejs npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
[name: (nodejs) nauka_npm
[version: (1.0.0)
[description: Testowy plik package.json do nauki npm'a
[entry point: (index.js)
[test command:
[git repository:
[keywords: nauka npm node
[author: kiper
[license: (ISC)
About to write to /Users/Kiper/Documents/nodejs/package.json:

{
  "name": "nauka_npm",
  "version": "1.0.0",
  "description": "Testowy plik package.json do nauki npm'a",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "nauka",
    "npm",
    "node"
  ],
  "author": "kiper",
  "license": "ISC"
}

[Is this ok? (yes) yes
→ nodejs |
```

Powyżej umieściłem swoje przykładowe odpowiedzi na pytania kreatora. W ten sposób w naszym katalogu powstał plik *package.json*.

Możemy skorzystać z przyspieszonej wersji tej komendy, czyli **npm init -y**, która udzieli na wszystkie pytania automatycznie odpowiedzi *y(es)*. Uzyskamy ten sam wynik i zaoszczędzimy nieco na czasie. Jest to jednak bardziej ciekawostka niż rzecz, którą trzeba zapamiętać.

---

## Dodawanie aliasów do npm

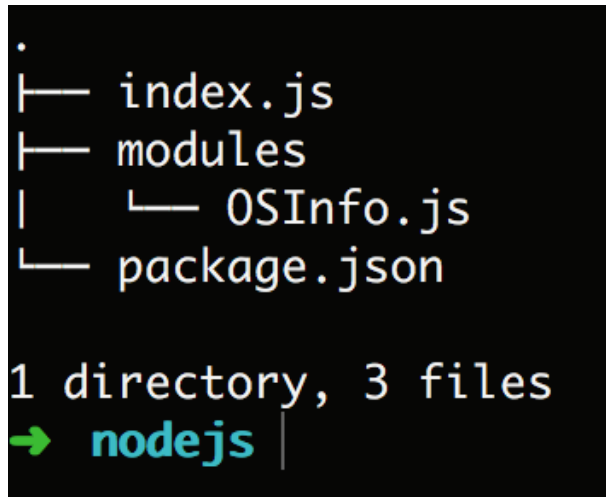
---



Dopiszmy teraz swój własny skrypt do odpalania naszej aplikacji. Do tej pory wykonywaliśmy to pisząc **node nazwaPliku**. Teraz będzie to wyglądało nieco inaczej.

Zmieńmy nazwę pliku: *program.js* na nazwę *index.js*. Następnie przenieśmy *index.js* do katalogu głównego projektu (czyli tam gdzie jest *package.json*). Jako, że folder *app* jest teraz pusty można śmiało go usunąć.

Całość powinna wyglądać mniej więcej tak:



```
.
├── index.js
├── modules
│   └── OSInfo.js
└── package.json

1 directory, 3 files
→ nodejs |
```

Kolejnym krokiem będzie otwarcie pliku *package.json* i dopisanie nowej komendy do tablicy *scripts*:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js"
},
```

**UWAGA:** pamiętaj, że w związku z przeniesieniem pliku *program.js* należy również zmienić wszystkie ścieżki relatywne do modułów w tym pliku, np. `var OSInfo = require('../modules/OSInfo');` na `var OSInfo = require('./modules/OSInfo');`

Teraz spróbujmy odpalić naszą aplikację komendą **npm start**. Jeśli wszystkie kroki zostały wykonane prawidłowo, aplikacja powinna działać tak, jak do tej pory.

## Zastosowanie aliasów npm

Programista w trakcie swojej pracy musi czasami uruchomić pewne skrypty, które np. skompilują kod Sass na CSS albo uruchomią serwer lokalny, który będzie automatycznie odświeżał przeglądarkę po tym, jak dokonamy zmian w kodzie. Dzięki skryptom npm możemy napisać proste zadania, które będą spełniały te i inne cele.



Wielu programistów korzysta z bardziej rozbudowanych narzędzi takich jak np. Grunt/Gulp, które są typowymi task runnerami (narzędziami do uruchamiania skryptów), ale skrypty npm też wbrew pozorom potrafią bardzo wiele!

---

## Główne zastosowanie npm

---

Ostatnim i chyba najważniejszym zadaniem npm jest pobieranie paczek udostępnianych przez społeczność o których już kilka razy wcześniej wspominaliśmy. Jest to chyba jedna z najważniejszych funkcjonalności, a przynajmniej z tego npm jest najbardziej znany. Nawet w NASA korzystają z npm, o czym nie tak dawno było głośno na forach JSowców.

---

## Zadanie: Pierwsze paczki za płaty

---

Jeśli szukamy paczki do konkretnego zastosowania, możemy wejść na [stronę npm](#) i skorzystać z wyszukiwarki paczek. My w ramach ćwiczeń rozbudujemy skrypt do wyświetlania informacji o systemie o kolory - skorzystamy w tym celu z modułu [colors](#).

## Instalacja pierwszej paczki

Zacznijmy od zainstalowania paczki za pomocą komendy `npm install --save colors`

Każdą paczkę instalujemy poleceniem `npm install nazwa_paczki`. Jeśli dodamy do tego flagę `--save` (tak jak powyżej) lub `-S` (`-S` to skrót od `--save`) informacje o naszej paczce zostaną zapisane w pliku `package.json`. Spójrzmy co zmieniło się w projekcie po wykonaniu tego polecenia:



```
→ nodejs npm install --save colors
nauka_npm@1.0.0 /Users/Kiper/Documents/nodejs
└─ colors@1.1.2

npm WARN nauka_npm@1.0.0 No repository field.
→ nodejs ls -l
total 16
-rw-r--r--  1 Kiper  staff  522  1 paź 15:34 index.js
drwxr-xr-x  3 Kiper  staff  102  1 paź 13:39 modules
drwxr-xr-x  3 Kiper  staff  102  1 paź 16:48 node_modules
-rw-r--r--  1 Kiper  staff  384  1 paź 16:48 package.json
→ nodejs cat package.json
{
  "name": "nauka_npm",
```

Jak widzisz, w naszym projekcie pojawił się katalog *node\_modules*. Przechowywane są w nim moduły, które instalujemy. Folder tworzy się automatycznie, gdy instalujemy pierwszy moduł.

Plik *package.json* również uległ zmianie. Widać, że pojawił się nowy klucz o nazwie **dependencies**, a pod nim obiekt z zainstalowanymi paczkami. Jak widać, nasz projekt zależy (ang. depends) od modułu *colors* w wersji 1.1.2. Dodatkowo znaczek ^ (ang. caret) oznacza, że npm znajdzie ostatnią wersję, która pasuje do wyrażenia 1.x.x. Istnieje jeszcze znaczek, ~ (ang. tilde), który dla tego przykładu szukałby ostatniej wersji zgodnej z wyrażeniem 1.1.x.

Więcej na temat wersjonowania paczek można poczytać w [tym](#) miejscu.

## Dołączenie zainstalowanej paczki

Kolejnym krokiem będzie zaimportowanie modułu w pliku *OSinfo.js*. Robimy to na samej górze pliku zaraz pod importowaniem modułu *os*:

```
var os = require('os');  
var colors = require('colors');
```

## Wykorzystanie paczki na podstawie dokumentacji

W [dokumentacji](#) jest napisane w jaki sposób korzystać z modułu. Twoim zadaniem będzie zmodyfikowanie polecenia w taki sposób, aby uzyskać podobny efekt:

```
→ nodejs npm start  
  
> nauka_npm@1.0.0 start /Users/Kiper/Documents/nodejs  
> node index.js  
  
/getOSinfo  
System: OSX  
Release: 15.5.0  
CPU model: Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz  
Uptime: ~ 1447 min  
User name: Kiper  
Home dir: /Users/Kiper
```

Po wykonaniu zadania umieść swój kod na Githubie i przekaz link do repozytorium swojemu mentorowi. Powodzenia!

Podgląd zadania

<https://github.com/0na/>

Wyślij link ✓

## 13.6. Zdarzenia w Node.js



Skoro miałeś już wcześniej do czynienia z JavaScriptem, zapewne doskonale wiesz czym są eventy - teraz spojrzymy na nie z punktu widzenia Node.js :)

## Pętla zdarzeń

Jak już wspominaliśmy sobie na początku, działanie Node opiera się na zdarzeniach. W zasadzie jego sukces to duża zasługa pętli zdarzeń, która wygląda mniej więcej tak:



Każda operacja, którą można określić jako wymagającą (np. zapis w bazie danych, operacja na plikach, obliczenia cięższego kalibru), może być zepchnięta na drugi plan. Kosztowne operacje często bywają nazywane blokującymi, ponieważ mogą zablokować główną pętlę działania programu jeśli trwają zbyt długo. My, użytkownicy programu, odczuwamy skutki takiego działania w postaci 'zacięć' lub długiego ładowania.

Pętla zdarzeń, tak jak widać na powyższym rysunku, zajmuje się rejestracją callbacków (czyli funkcji, które mają się wykonać po zakończonej operacji). Pętla ta nie tylko oddelegowuje zdarzenia, ale także oznajmia głównej pętli, kiedy można odpalić zarejestrowane callbacki.

Pętla zdarzeń działa tak samo w przeglądarce jak i w środowisku Node'a. Jak już wcześniej wyjaśniałem, jedyną różnicą są tak naprawdę zdarzenia, które mogą pojawiać się w trakcie działania aplikacji. W przeglądarce typowe zdarzenia to np. 'załadowanie dokumentu', 'kliknięcie', w aplikacjach opartych o Node są to z kolei: pobranie pliku, odpowiedź na zapytanie do serwera, nasłuchiwanie na wpisanie polecenia `/exit`.





# Czym jest EventEmitter?

No właśnie, a co, jeśli sami potrzebujemy emitować zdarzenie? Odpowiedzią na to pytanie jest wbudowany w Node moduł *events*, a w szczególności klasa *EventEmitter*. Dzięki niej możemy tworzyć swoje własne zdarzenia i przypisywać akcje, które mają zostać wykonane, jeśli taka akcja się pojawi. Jest to implementacja pewnego popularnego wzorca projektowego Publish/Subscribe, który jest dosyć często spotykany w postaci różnych bibliotek.

## Kiedy chcemy używać EventEmittera?

Istnieje pewne zjawisko, które programiści nazywają **callback hell**. Objawia się to tym, że w jednym callbacku tworzymy kolejny, a w nim kolejny itd. Sytuacja wygląda mniej więcej tak:

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Zauważ jak duża jest liczba zagnieżdżeń. Event Emitter zapobiega temu zjawisku emitując zdarzenia zamiast rejestrować callbacki. Dzięki czemu nasz kod jest bardziej płaski.

Okej, kiedy używać EventEmittera? Odpowiedź brzmi następująco - kiedy chcemy mieć luźniejsze powiązania w kodzie.



Dzięki zdarzeniom nie musimy się martwić o to, jakie fragmenty kodu "rozmawiają ze sobą". Jeśli zostanie opublikowane jakieś zdarzenie, a nie ma subskrypcji, to zdarzenie najzwyczajniej rozejdzie się echem. Jeśli jakaś funkcja jest zainteresowana zdarzeniem, to nasłuchuje czy przypadkiem nie zostało rozesłane.

Spróbujemy rozwinąć nasz projekt i dopisać dwa zdarzenia - **beforeCommand** oraz **afterCommand**, które będą emitowane odpowiednio przed i po zdarzeniu. Napiszemy również kod, który będzie nasłuchiwał obu zdarzeń i wyświetlał odpowiednio: **You wrote: <to co wpisał użytkownik>, trying to run a command** przed odpaleniem komendy i **Finished the command** po zakończeniu działania polecenia.

To do dzieła - najpierw zaimportujemy moduł *event* i wyciągniemy z niego klasę *EventEmitter*.

```
var event = require('events');  
var EventEmitter = event.EventEmitter;
```

Jako, że z modułu *event* potrzebujemy jedynie klasy *EventEmitter*, możemy nieco uprościć powyższy kod:

```
var EventEmitter = require('events').EventEmitter;
```

Kolejnym krokiem, będzie utworzenie obiektu na podstawie klasy *EventEmitter*.

```
var emitter = new EventEmitter();
```

Teraz dodajmy sobie dwa nasłuchiwanie na zdarzenia:

```
emitter.on('beforeCommand', function(instruction) {  
    console.log('You wrote: ' + instruction + ' trying to run command');  
});  
emitter.on('afterCommand', function() {  
    console.log('Finished command');  
});
```

Oba callbacki wyglądają bardzo podobnie do jQuery'owego `.on` czy do klasycznego `.addListener()`. Pierwszy parametr to nazwa zdarzenia, a drugi to funkcja, która wykona się na jego wystąpienie.



Ostatnim krokiem będzie wywołanie zdarzeń w odpowiednich miejscach. Całość powinna wyglądać tak:

```
var EventEmitter = require("events").EventEmitter;
var OSinfo = require('./modules/OSinfo');

var emitter = new EventEmitter();
emitter.on("beforeCommand", function (instruction) {
    console.log('You wrote: ' + instruction + ', trying to run command');
});
emitter.on("afterCommand", function () {
    console.log('Finished command');
});

process.stdin.setEncoding('utf-8');
process.stdin.on('readable', function() {
    var input = process.stdin.read();
    if(input !== null) {
        var instruction = input.trim();
        // odpalanie zdarzenia beforeCommand (z parametrem)
        emitter.emit('beforeCommand', instruction);
        switch(instruction) {
            case '/exit':
                process.stdout.write('Quiting app!');
                process.exit();
                break;
            case '/sayhello':
                process.stdout.write('hello!\n');
                break;
            case '/getOSinfo':
                OSinfo.get();
                break;
            default:
                process.stderr.write('Wrong instruction!\n');
        };
        // emitowanie zdarzenia afterCommand (bez parametru)
        emitter.emit('afterCommand');
    }
});
```

Jest pewna sytuacja, w której zdarzenie afterCommand nie zostanie wyemitowane - potrafisz wskazać która? :)

To jest podstawowy przykład użycia publish/subscribe. Oczywiście tak jak w przypadku innych modułów wbudowanych w Node, tak i w tym istnieje bogata [dokumentacja](#), która omawia poszczególne fragmenty modułu. W trakcie tworzenia aplikacji będziemy

jeszcze wracać do tematu zdarzeń wiele razy!

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✓ Zapoznałem się!

## 13.7. File System - praca z plikami



W skład natywnych modułów Node'a wchodzi również moduł o nazwie *fs* (*file system*). Dostarcza on wszystkich funkcjonalności niezbędnych do pracy z plikami. Nie musimy się martwić o kompatybilności na różnych systemach operacyjnych. Moduł załatwia to za nas, więc niezależnie od tego czy nasz program zostanie odpalony na platformie Windows, OS X, Linux czy nawet Android, zachowa się on tak samo!

Moduł dostarcza zarówno synchronicznych, jak i asynchronicznych metod dostępu do pliku. W pierwszym submodule dowiedzieliśmy się już czym charakteryzuje się kod wykonywany synchronicznie i asynchronicznie, ale zatrzymajmy się jeszcze na chwilę, aby przypomnieć sobie o co właściwie chodziło.

W przypadku kodu wykonywanego **synchronicznie**, każda operacja jest wykonywana krok po kroku. Oznacza to, że program zatrzymuje się przed wykonaniem kolejnej sekwencji i nie rozpoczyna jej, dopóki nie skończy wykonywania poprzedniej.

W przypadku kodu **asynchronicznego**, to sekwencja nie jest w żaden sposób zatrzymywana. Kod wykonujący się asynchronicznie wykonywany jest w tle aplikacji, dzięki czemu kosztowne (trwające długo lub wymagające dużo zasobów) operacje nie powodują zawieszenia programu, ponieważ nie czekają na rozpoczęcie kolejnych sekwencji.

W większości przypadków będziesz korzystał z metod asynchronicznych, ale warto wiedzieć, że nie jest to jedyna dostępna opcja i można skorzystać również z metod synchronicznych.

Moduł *fs* udostępnia kilka podstawowych klas/metod, z których możemy korzystać - wymienimy sobie poniżej kilka z nich.



# fs.Stats

Dzięki tej klasie możemy sprawdzać czy plik/katalog/ścieżka do której się odnosi istnieje, możemy również podejrzeć właściwości pliku/katalogu. Wykonajmy sobie proste ćwiczenie podglądające informacje na temat pliku.

Zacniemy od stworzenia katalogu, którego struktura będzie wyglądała tak:

```
.  
├─ cat.jpg  
├─ index.js  
└─ package.json
```

Obrazek *cat.jpg* pobierz sobie z internetu. Kotków w sieci jest cała masa, więc jest w czym przebierać. Ja wybrałem sobie takiego:



Następnie napiszemy sobie prosty skrypt w pliku *index.js* wyświetlający informacje o obrazku:

```
var fs = require('fs');  
  
fs.stat('./cat.jpg', function(err, stats) {  
  console.log(stats);  
});
```



Po wykonaniu komendy **npm start** na ekranie powinno się nam wyświetlić coś takiego:

```
→ nodejs3 npm start

> nauka_npm@1.0.0 start /Users/Kiper/Documents/nodejs3
> node index.js

{ dev: 16777220,
  mode: 33188,
  nlink: 1,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 1949739,
  size: 22491,
  blocks: 48,
  atime: 2016-10-04T07:27:21.000Z,
  mtime: 2016-10-04T07:27:17.000Z,
```

Program nie robi nic poza wyświetleniem informacji na temat obrazka. Daty nie powinny nikogo zaskoczyć, ale reszta wartości może nie być do końca jasna. Rozmiar jest określony w bajtach (8 bitów), Rozmiar bloku (ang. *blksize*) jest używany do określenia rozmiaru pojedynczego bloku w systemie operacyjnym. Zaraz obok pojawia się ilość tych bloków. Mode określa dostęp do pliku. Jest jeden problem - nie wiemy jak je rozszyfrować. Otóż istnieje pewien moduł, który pozwala na wyświetlanie tych wartości w bardziej czytelny sposób (znany zresztą z systemów Unixowych).

```
→ nodejs3 ls -la
total 80
drwxr-xr-x  6 Kiper  staff   204  4 paź 09:35 .
drwx-----+ 16 Kiper  staff   544  4 paź 09:27 ..
-rw-r--r--@  1 Kiper  staff  6148  4 paź 09:23 .DS_Store
-rw-r--r--@  1 Kiper  staff 22491  4 paź 09:27 cat.jpg
-rw-r--r--  1 Kiper  staff   93  4 paź 09:36 index.js
-rw-r--r--  1 Kiper  staff   384  4 paź 09:23 package.json
```

Chodzi o podkreślony fragment, który zawiera porcję informacji na temat pozwolenia dostępu do pliku dla konkretnych grup użytkowników. Tych, którzy nie słyszeli o tym zapisie odsyłam [tutaj](#). Teraz zajmiemy się dekodowaniem wyniku operacji `fs.stats`.

```
var fs = require('fs');
var StatMode = require('stat-mode');

fs.stat('./cat.jpg', function(err, stats) {
  var statMode = new StatMode(stats);
  console.log(statMode.toString());
});
```

Aby poprawnie zdekodować *mode*, pobierzemy sobie gotowy moduł *stat-mode* z npm:

```
→ nodejs3 npm install --save stat-mode
```

Przypominam, że flaga `--save` zapisuje moduł w pliku *package.json*.

Po uruchomieniu skryptu powinniśmy zobaczyć coś takiego:

Czy jesteś w stanie odczytać jakie grupy użytkowników mają dostęp do plików?  
Możesz użyć linka, do którego odsyłałem Cię powyżej.

---

## Odczyt i zapis pliku (read/write)

---

Teraz zajmiemy się najczęściej używaną funkcjonalnością modułu *fs*. Za zapis i odczyt odpowiadają kolejno metody `fs.writeFile` i `fs.readFile`. Każda z nich otwiera plik, wykonuje swoją funkcję (zapis/odczyt), po czym zamyka plik. Stwórzmy sobie plik z dowolnym tekstem i nazwijmy go *tekst.txt*, a następnie napiszmy:

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

fs.readFile('./tekst.txt', function(err, data) {
  console.log(data);
});
```



Kod ma za zadanie odczytać plik *tekst.txt* i wyświetlić jego zawartość w konsoli. Spójrzmy więc co się wydarzy, gdy uruchomimy program:

Wynikiem działania naszego programu jest dziwny ciąg znaków. Jest to *Buffer*, który jest domyślnym formatem odczytu i zapisu plików. Powyższy zapis to heksadecymalny ciąg kolejnych znaków drukowanych ASCII. My, ludzie, nie jesteśmy w stanie odczytać takiego zapisu (no chyba, że znamy na pamięć znaczenie każdego z użytych tutaj znaków ASCII), dlatego należy użyć odpowiedniego kodowania, aby odszyfrować ten zapis (odczytać go w formie tekstowej):

```
fs.readFile('./tekst.txt', 'utf-8', function(err, data) {  
    console.log(data);  
});
```

Teraz, gdy uruchomimy skrypt naszym oczom powinien ukazać się następujący widok:

Odczyt pliku mamy za sobą - przejdźmy teraz do zapisu. Do istniejącego już skryptu dopiszmy sobie następujący kod:

```
fs.writeFile('./tekst.txt', 'Tekst, który zapiszemy w pliku tekst.txt',  
    if (err) throw err; // jeśli pojawi się błąd, wyrzuc wyjątek  
    console.log('Zapisano!');  
});
```

Po uruchomieniu skryptu powinniśmy zobaczyć coś takiego:

Jak się okazało, najpierw doszło do zapisu pliku, a w drugiej kolejności do jego odczytu. Dzieje się tak niezależnie od tego w jakiej kolejności ustawimy funkcje odczytu i zapisu. Dlaczego? Jest to przypadek tzw. sytuacji wyścigu (ang. *race condition*). Obie funkcje są asynchroniczne i w tym wypadku pierwsza wykona się ta, która szybciej otrzyma prawa dostępu do pliku (jest to funkcja *writeFile*).

Spróbujmy rozwinąć program i dodać funkcjonalność, która najpierw odczytuje zawartość pliku, następnie zapisze dane i znowu odczyta plik po zmianie jego zawartości:



```
var fs = require('fs');
var colors = require('colors');

fs.readFile('./tekst.txt', 'utf-8', function(err, data) {
  console.log('Dane przed zapisem!'.blue);
  console.log(data);
  fs.writeFile('./tekst.txt', 'A tak wyglądają po zapisie!', function(err) {
    if (err) throw err;
    console.log('Zapisano!'.blue);
    fs.readFile('./tekst.txt', 'utf-8', function(err, data) {
      console.log('Dane po zapisie'.blue);
      console.log(data);
    });
  });
});
```

Zauważ, że użyliśmy modułu z poprzednich zadań (*colors*) - jeśli nie pamiętasz jak go zainstalować, wróć do submodułu o npm.

Kod ma teraz trzy zagnieżdżenia. Każde zdarzenie (odczyt, zapis, odczyt) musi wykonać się w odpowiedniej kolejności, dlatego zagłębiamy kolejne wywołania asynchroniczne wewnątrz następujących po sobie callbacków. W poprzednim submodule na temat zdarzeń wyjaśniliśmy sobie, że możemy pozbyć się tego problemu emitterem zdarzeń, ale póki co zadowolmy się takim kodem. W przyszłości, kiedy będziemy musieli dodać nowe funkcjonalności, będziemy mogli pomyśleć o *refaktoryzacji* kodu.

Wynik działania tego programu:

Jest jeszcze jedna rzecz, którą można tutaj poprawić. Każde zapisanie tekstu nadpisuje jego poprzednią zawartość. Byłoby o niebo lepiej gdyby tekst był dopisywany do istniejącego pliku.

Sprawa jest bardzo prosta, wystarczy, że zmienimy słówko **write** na **append** w wywołaniu metody *writeFile*:

```
fs.appendFile('./tekst.txt', '\nA tak wyglądają po zapisie!', fun...);
```

Dodaliśmy również znak nowej linii (`\n`), aby tekst **A tak wyglądają po zapisie** pojawił się w kolejnym wierszu. Nasz program powinien teraz wyglądać następująco



Czy domyślasz się jak wyglądałby plik tekstowy po kolejnych dziesięciu uruchomieniach?

---

## Zadanie: Badamy otoczenie!

---

Stwórz funkcję, która odczyta zawartość katalogu za pomocą funkcji `fs.readdir`, a następnie zapisze je do nowo utworzonego pliku za pomocą metody `fs.writeFile`.

Po wykonaniu zadania umieść swój kod na Githubie i prześlij link do repozytorium swojemu mentorowi.

Podgląd zadania

<https://github.com/0na/>

Wyślij link ✓

## 13.8. Nasz własny serwer HTTP



Najbardziej popularnym modulem, z którego znany jest Node.js jest z całą pewnością `http`. Zanim przejdziemy do omawiania samego modułu, wyjaśnijmy sobie czym jest protokół HTTP.

---

### Protokół HTTP

---

Rozwijając akronim jest to *Hypertext Transfer Protocol*. Każda strona, którą odwiedzasz ma w swoim adresie `http://` lub `https://` (bezpieczniejsza, szyfrowana wersja protokołu HTTP). Szyfrowanie stosuje się w celu zabezpieczenia danych przed niepożądanym dostępem osób trzecich.



# Nagłówki HTTP

Dzięki nagłówkom nasz serwer może przesłać dodatkową informację wraz z odpowiedzią (ang. *response*) albo otrzymać ją od klienta w zapytaniu (ang. *request*).

Przykładowy nagłówek to **"nazwa nagłówek": "wartość nagłówek"**

## Metody HTTP

- **GET** - przekazanie zasobu do odpytującego o niego klienta (najprostsza forma zapytania)
- **HEAD** - dzięki tej metodzie serwer zwraca do klienta informacje na temat zasobu, o który klient odpytuje
- **POST** - tutaj serwer zazwyczaj implementuje odbiór informacji wysłanej przez klienta (np. w formie formularza), zajmuje się odpowiednio obróbką tych danych, a na końcu zazwyczaj zapisuje dane w bazie
- **PUT** - metoda, która ma bardzo podobne zadanie co POST z tą różnicą, że operuje na zazwyczaj już istniejącej encji w celu jej uaktualnienia
- **DELETE** - metoda, która pozwala na usunięcie zasobu

## Kody odpowiedzi HTTP

Ostatnim elementem protokołu HTTP są kody odpowiedzi nazywane też statusami. Są prezentowane jako trzycyfrowe wartości i dzięki nim klient wie, co mogło się stać po stronie serwera. Spójrzmy na typowe oznaczenia tych kodów:

- 1xx - rzadko spotykane kody informacyjne
- 2xx - kod, który oznacza, że zapytanie klienta zostało poprawnie odebrane, zrozumiane i zaakceptowane. Przykłady:
  - 200 - Zapytanie się powiodło, a odpowiedź na zapytanie jest zależna od metody, której użyto do wysłania zapytania
  - 201 - symbolizuje nowo utworzony zasób na serwerze - serwer może odpowiedzieć w ten sposób np. po dodaniu komentarza.
- 3xx - kod przekierowania. Oznacza, że klient musi podjąć pewne akcje, aby dokończyć zapytanie. Po tym zapytaniu kolejne musi być HEAD, albo GET w celu pobrania informacji o zasobie, albo pobrania innego zasobu.
- 4xx - błąd spowodowany działaniami użytkownika. Najpopularniejszym kodem z tego zakresu jest błąd 404 i pojawia się, kiedy chcemy się odnieść do miejsca w sieci, które nie istnieje. Innymi przykładami mogą być:
  - 400 - serwer nie potrafi zrozumieć zapytania
  - 401 - nieautoryzowane zapytanie. Pojawia się, gdy pominiemy odpowiednie nagłówki autoryzacji lub podane informacje są nieprawidłowe
  - 403 - serwer zrozumiał zapytanie, ale nie zgadza się na wysłanie odpowiedzi.
- 5xx - błędy serwera. Pojawiają się, kiedy serwer nie potrafi przetworzyć informacji, albo stało się coś nie tak po jego stronie. Przykładowe błędy:



- 500 - wewnętrzny błąd serwera uniemożliwiający mu spełnienie zapytania
- 501 - funkcjonalność potrzebna do spełnienia zapytania nie jest jeszcze zaimplementowana
- 503 - serwis nie jest w stanie w tej chwili obsłużyć zapytania

---

## Implementacja protokołu po stronie Node

---

Wiemy już z jakich fragmentów składa się sam protokół. Przejdźmy teraz do zapoznania się z jego Node'ową implementacją w postaci modułu `http`. Napiszmy prosty serwer, który zwróci klientowi odpowiedź w postaci "Hello World!". Zaczniemy od stworzenia prostego serwera.

### Inicjacja serwera HTTP

```
var http = require('http');

var server = http.createServer(function (request, response) {
  // tutaj coś się dzieje :)
});
```

Callback, który został przekazany do metody `createServer` będzie się odpalał za każdym razem, gdy klient nawiąże połączenie. Jest to nic innego jak nasłuchiwanie na zdarzenie wystąpienia odpytania klienta. Powinna Ci się w tym momencie zaświecić lampka w głowie. Zdarzenia, nasłuchiwanie, to wszystko omawialiśmy przy okazji EventEmitterów. Otóż moduł `http` (jak i wiele innych modułów Node'owych) rozszerza swoją klasę o `EventEmitter`. Tak więc powyższy zapis jest taki sam jak ten:

```
var http = require('http');

var server = http.createServer();
server.on('request', function (request, response) {
  //tutaj coś się dzieje :)
});
```

Kiedy zapytanie HTTP dotrze do serwera, Node uruchomi callback i dzięki odpowiednim obiektom transakcji (`request`, `response`) może wykonać pewne czynności.

### Uruchomienie procesu nasłuchiwania



Jest jeszcze jeden krok, który należy wykonać zanim uruchomimy nasz serwer. W celu właściwego obsługiwanie zapytań należy nie tylko stworzyć (`createServer`) i przypiąć nasłuchiwanie (`.on()`), ale także uruchomić nasłuchiwanie na te zdarzenia tak, aby serwer działał w trybie ciągłym. Do tego celu służy metoda `.listen()`, która jako parametr przyjmuje port, na którym ma nasłuchiwać:

```
var http = require('http');

var server = http.createServer();
server.on('request', function (request, response) {
  //tutaj coś się dzieje :)
});
server.listen(9000);
```

## Wysłanie odpowiedzi protokołem HTTP

Po uruchomieniu nasz serwer oprócz samej obsługi mechanizmu protokołu HTTP.. nie robi tak naprawdę nic. Nie reaguje na zapytania, ani nie odsyła odpowiedzi do klienta. Dopiszmy więc prostą odpowiedź:

```
server.on('request', function (request, response) {
  response.write('Hello world!');
  response.end();
});
```

Odpalmy nasz skrypt i wejdźmy na <http://localhost:9000> naszym oczom powinien ukazać się taki widok:

Wysłanie odpowiedzi do klienta odbywa się za pomocą obiektu `response`, który używa do tego celu tzw. `WritableStream`.

---

## Stream - co to właściwie jest?

---

Użytkownikom systemów unixowych strumień może kojarzyć się ze znacznikiem `|` (ang. *pipe*), który łączy wyjście z jednego polecenia i przekazuje do następnego. Z punktu widzenia Node strumienie są implementacją *EventEmittera*, które implementują specjalne metody. W zależności od właśnie tych metod strumień może służyć do odczytywania (ang. *readable*), zapisywania (ang. *writable*) lub do komunikacji obustronnej (jednocześnie zapisu i odczytu).



Obiekt *response* jak już wspomniałem implementuje strumień zapisu, z kolei obiekt *request* jest obiektem, który posługuje się strumieniem odczytu.

W powyższym przykładzie strumień niesie ze sobą porcję danych, którą wpuszczamy za pomocą metody `.write()`. W ten sposób możemy wpuszczać do strumienia kolejne porcje dopisując kolejne fragmenty np:

```
response.write('Hello world');  
response.write('<h1>This is awesome!</h1>');
```

Ale nasz serwer nie spłucze (ang. *flush*) tych nagromadzonych danych, dopóki nie wywołamy metody `.end()`:

```
response.write('<body>');  
response.write('<h1>Hello world!</h1>');  
response.write('</body>');  
response.end();
```

Spłukanie danych można rozumieć jako akceptację do wysłania ich na wyjście strumienia. Po spłukaniu danych, nie możemy już zapisywać niczego do strumienia, tak więc poniższy kod zwróci błąd:

```
response.write('<body>');  
response.write('<h1>Hello world!</h1>');  
response.write('</body>');  
response.end();  
response.write('<h2>This is awesome!</h2>');
```

---

## URL - różne ścieżki, różne odpowiedzi

---

Wiemy już czym są streamy i potrafimy odpowiadać na zapytania klienta. Możemy trochę urozmaicić sposób odpowiedzi na zapytanie w zależności od ścieżki na którą wysyła zapytanie klient. Obiekt *request* trzyma w sobie wiele informacji na temat zapytania. Jest tam między innymi metoda za pomocą której klient sformułował zapytanie i ścieżka URL z której wyszło zapytanie. Spójrzmy:



```
var http = require('http');

var server = http.createServer();

server.on('request', function (request, response) {
  response.setHeader("Content-Type", "text/html; charset=utf-8");
  if (request.method === 'GET' && request.url === '/hello') {
    response.write('<h1>Hello World!</h1>');
    response.end();
  } else {
    response.statusCode = 404;
    response.write('<h1>404: Zła ścieżka!</h1>');
    response.end();
  }
});

server.listen(8080);
```

Powyższa aplikacja zwraca do klienta odpowiedź `response.write('<h1>Hello World!</h1>')`; , jeśli metoda zapytania to GET, a adres to `/hello`. W przeciwnym wypadku warto pokazać klientowi, że nie jesteśmy w stanie obsłużyć jego zapytania i odesłać odpowiedź o konkretnym statusie (`response.statusCode = 404;`) i w ciele odpowiedzi dodać `response.write('<h1>404: Zła ścieżka!</h1>')`;

Obie odpowiedzi należy kodować w odpowiedni sposób tak, aby przeglądarka wiedziała w jaki sposób odczytać wiadomość - z tego względu na samej górze funkcji nasłuchującej na wystąpienie zapytania pojawia się linijka `response.setHeader("Content-Type", "text/html; charset=utf-8").`

**UWAGA:** Pamiętaj, że nagłówki ustawiamy zawsze przed wysłaniem ciała odpowiedzi!

Jeśli wszystko jest ok, nasz serwer powinien w tym momencie rozróżniać rodzaje zapytań i w odpowiedni sposób odsyłać wiadomość na wystąpienie zdarzenia.

Wiemy już w jaki sposób korzystać z modułu `http` :

---

## Zadanie: Serwujemy pliki!

---



Twoim zadaniem będzie wykorzystanie modułu *fs* i modułu *http* do serwowania plików statycznych. Stwórz plik *index.html* i odeślij jego treść do klienta, jeśli wystąpi zapytanie na url */*. W przeciwnym wypadku odeślij mu dowolny obrazek, który poinformuje go o wystąpieniu błędu 404.

Skorzystaj z metody **`fs.readFile`**, aby odczytać plik. Pamiętaj też o odpowiednim ustawieniu nagłówków!

Po wykonaniu zadania, umieść swój kod na Githubie i wyślij swojemu mentorowi link do rozwiązania.

Podgląd zadania

<https://github.com/0na/>

Wyślij link

## 13.9. PROJEKT: Strona do uploadowania plików/obrazków



Stworzymy sobie mały projekt, który będzie miał na celu obsługę wgrywania pliku za pomocą strony z formularzem. Serwowanie plików samej strony jak i obsługę uploadowania plików rozwiążemy za pomocą Node. W tym celu wykorzystamy techniki, które poznaliśmy do tej pory. Użyjemy między innymi modułów *fs*, *http*, rozbijemy nasz kod i stworzymy swoje własne moduły, skorzystamy również z gotowych rozwiązań. Do dzieła!

---

## Nodemon - automatyczne restartowanie serwera

---





Zapewne zdążyła Cię już zmęczyć konieczność włączania procesu Node od nowa za każdym razem, kiedy wprowadzisz jakąkolwiek zmianę w kodzie - teraz zautomatyzujemy sobie ten proces za pomocą narzędzia o nazwie **Nodemon**.

Nodemon jest to nic innego jak mała aplikacja, która obserwuje nasze pliki i uruchamia proces Node od nowa za każdym razem, kiedy wykryje zmianę. Aby go zainstalować, użyj komendy **npm install -g nodemon**.

Wykorzystanie samego Nodemona jest bardzo proste - wystarczy, że zamiast komendy **node plik.js** użyjesz **nodemon plik.js** - to tyle! :) Przejdźmy do tworzenia projektu, tam wykorzystasz Nodemona w praktyce.

---

## Inicjalizacja projektu

---

Zacniemy od zainicjalizowania prostego projektu komendą **npm init -y**. Jak zapewne pamiętasz, dzięki argumentowi **-y**, na wszystkie pytania zadawane przez npm odpowiadamy 'tak', co skutkuje natychmiastowym stworzeniem pliku *package.json*. Do gotowego pliku dopisujemy komendę:

```
"scripts": {  
  "start": "nodemon index.js",  
  "test": "echo \"Error: no test specified\" && exit 1"  
}
```

Kolejnym krokiem będzie przygotowanie odpowiedniej struktury katalogów.

Plikiem wejściowym do aplikacji jest plik *index.js*. Ze względu na to, że będziemy pisać swoje własne moduły stwórzmy sobie na nie osobny katalog *modules* i stwórzmy w nim pierwszy moduł obsługujący logikę serwera. Cała struktura powinna wyglądać tak:

```
.  
├─ index.js  
├─ modules  
│   └─ server.js  
└─ package.json
```



# Podstawowy kod serwera

W pliku *server.js* napiszmy następującą logikę:

```
var http = require('http');
var colors = require('colors');

function start() {
  function onRequest(request, response) {
    console.log("Odebrano zapytanie.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Pierwsze koty za płoty");
    response.end();
  }

  http.createServer(onRequest).listen(9000);

  console.log("Uruchomiono serwer!".green);
}

exports.start = start;
```

Jak widzisz, korzystamy z modułu *colors*. Trzeba więc go doinstalować za pomocą komendy `npm install --save colors`.

Kolejnym krokiem będzie uruchomienie i weryfikacja, czy postawiony serwer http obsługuje zapytania klienta - w tym celu należy zaimportować odpowiedni moduł w pliku *index.js*, a następnie uruchomić serwer w następujący sposób:

```
var server = require('./modules/server');
server.start();
```

Po wpisaniu komendy `npm start` powinniśmy ujrzeć następujący widok:

Kiedy zaś przejdziemy do przeglądarki i wpiszymy adres `localhost:9000` zobaczymy:

## Obsługa zapytań klienta



Nadal brakuje nam mechanizmu, który zapewniałby obsługę różnych zapytań klienta, więc dopiszemy teraz prosty moduł, który się tym zajmie. Będziemy rozróżniać zapytania na adresy `/start`, `/upload`, lub `/`. Każda inna forma zapytania ma generować błąd 404.

Zacniemy od utworzenia pliku *handlers.js* i napisania kilku metod obsługujących odpowiednie zapytania:

```
exports.upload = function(request, response) {
  console.log("Rozpaczynam obsługę żądania upload.");
  response.write("Rozpaczynam upload!");
  response.end();
}

exports.welcome = function(request, response) {
  console.log("Rozpaczynam obsługę żądania welcome.");
  response.write("Witaj na stronie startowej!");
  response.end();
}

exports.error = function(request, response) {
  console.log("Nie wiem co robić.");
  response.write("404 :(");
  response.end();
}
```

Kolejny krok to zmiany w pliku *server.js*. Musimy w odpowiednich przypadkach wywołać obsługę odpowiedniego handlera:

```
var http = require('http');
var colors = require('colors');

var handlers = require('./handlers'); // nasz moduł

function start() {
  function onRequest(request, response) {
    console.log("Odebrano zapytanie.".green);
    console.log("Zapytanie " + request.url + " odebrane.");

    response.writeHead(200, {"Content-Type": "text/plain; charset=utf-8"});

    switch (request.url) { // switch rozróżniający zapytania
      case '/':
      case '/start':
        handlers.welcome(request, response);
        break;
      case '/upload':
        handlers.upload(request, response);
        break;
      default:
        handlers.error(request, response);
    }
  }

  http.createServer(onRequest).listen(9000);

  console.log("Uruchomiono serwer!".green);
}

exports.start = start;
```

Jak widzisz, wystarczy dodać nasz moduł (handler.js) i dopisać odpowiedniego switcha, który wywoła odpowiedni handler. Zauważ, że instrukcja zakłada, że ma wywołać tę samą funkcję, jeśli URL będzie równy /, lub /**start**. Jeśli serwer nie rozpozna zapytania klienta (case **default**), to odeśle w odpowiedzi odpowiednią informację - 404.

---

## Dodanie obsługi plików statycznych

---



Dodajmy teraz formularz, za pomocą którego będziemy wysyłać pliki - zacznijmy od stworzenia prostego pliku HTML (*templates/start.html*) i umieścimy w nim następującą zawartość:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=U
  </head>
  <body>
    <h1>Wgraj plik</h1>
    <form action="/upload" enctype="multipart/form-data" method="
      <label>Tytuł pliku</label>
      <input type="text" name="title">
      <br>
      <label>Plik</label>
      <input type="file" name="upload" multiple="multiple"><br>
      <input type="submit" value="Wyślij">
    </form>
  </body>
</html>
```

Formularz będzie wykonywał odpowiednią akcję na adres `/upload` metodą POST. Po wciśnięciu *Wyślij* strona przeniesie nas w inne miejsce.

Zmieńmy logikę obsługi wejścia na `/start` lub `/`. W tym celu, w pliku *handlers.js* należy odczytać plik *start.html*, a następnie wysłać odpowiedź do klienta. Zauważ, że trzeba poinformować przeglądarkę o rodzaju wysyłanej odpowiedzi ustawiając nagłówek `Content-Type` na `text/html`

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

exports.welcome = function(request, response) {
  console.log("Rozpaczynam obsługę żądania welcome.");
  fs.readFile('templates/start.html', function(err, html) {
    response.writeHead(200, {"Content-Type": "text/html; charset=
    response.write(html);
    response.end();
  });
}
```

## Obsługa treści wysłanej za pomocą formularza

Nie zapominajmy, że plik wysłany przez formularz na `/` lub `/start` serwer musi zapisać po swojej stronie. W tym celu musimy również zmodyfikować dotychczasową obsługę zapytania `/upload`.

```
var formidable = require('formidable');

exports.upload = function(request, response) {
  console.log("Rozpoczynam obsługę żądania upload.");
  var form = new formidable.IncomingForm();
  form.parse(request, function(error, fields, files) {
    fs.renameSync(files.upload.path, "test.png");
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("received image:<br/>");
    response.write("<img src='/show' />");
    response.end();
  });
}
```

Skorzystamy z gotowego modułu do obsługi zapytań wysłanych za pomocą formularza. W tym celu należy zainstalować moduł *formidable* komendą `npm install --save formidable`, a następnie dołączyć go do projektu za pomocą `var formidable = require('formidable');`.

Za pomocą metody `.parse()` moduł *formidable* jest w stanie odpowiednio sformułować nadchodzące zapytanie. Metoda `.renameSync()` odpowiada za zmianę nazwy uploadowanego pliku, która kryje się pod kluczem `files.upload.path` na nazwę *test.png* i zapisze go na poziomie pliku *index.js*. Struktura projektu razem z załadowanym zdjęciem powinna więc wyglądać tak:

## Obsługa wysyłania zdjęcia

W odpowiedzi na zapytanie `upload` nie tylko wgrywamy zdjęcie na serwer, ale także odsyłamy obrazek pokazując, że znajduje się on już na serwerze (`

Wyślij link

[Regulamin](#)

[Polityka prywatności](#)

© 2019 Kodilla

