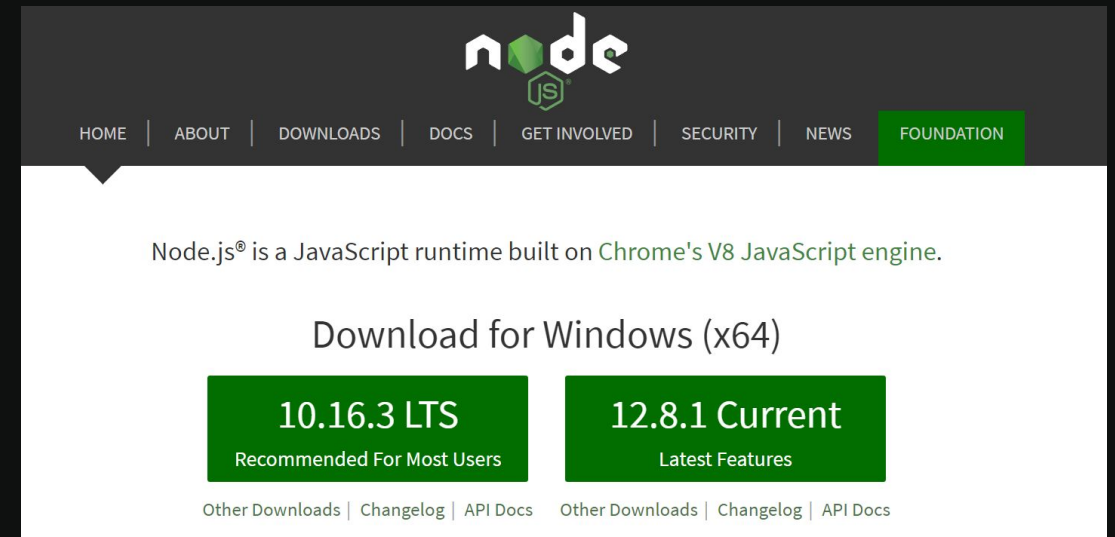




# Czym jest Node.js

**Node.js** - wieloplatformowe środowisko uruchomieniowe JavaScript, czy szerzej platforma do tworzenia aplikacji serwerowych (programowanie po stronie serwera) za pomocą JavaScript.

Z perspektywy front-end developera, Node.js (oraz npm i dodatkowe narzędzia) umożliwia stworzenie profesjonalnego środowiska deweloperskiego, który pozwala m.in. na automatyzację zadań, organizację projektu w modułach, łatwe korzystanie z zewnętrznych paczek (narzędzi, bibliotek) oraz tworzenie zoptymalizowanej wersji produkcyjnej naszego projektu.



# Node.js - czym nie jest

Od 2009 roku - twórca: Ryan Dahl

- Działa po stronie serwera. **Nie jest jednak serwerem czy web serwerem**, choć możemy w oparciu o Node.js taki **web serwer** stworzyć.
- **Nie jest też frameworkiem webowym** (do tworzenia stron/aplikacji webowych) - tym jest w Node.js np. **Express**, podobnie jak Laravel czy Symfony w PHP, Spring w Java, Ruby on Rails w Ruby, Django w Pythonie.

Dzięki Node.js JavaScript wyszedł z fazy dzieciństwa (tylko przeglądarka) i wszedł w fazę dorosłości. Dziś to język programowania, którego używa się wszędzie, a programiście JavaScript mogą robić wszystko (front i aplikacje serwerowe).

# Node.js - do czego w praktyce (backend)



Wykorzystywany od prostych do bardziej zaawansowanych stron/serwisów/sklepów internetowych (jako alternatywa dla wykorzystania PHP czy frameworków innych języków), strumieniowania danych, tworzenia serwerów API (REST API), komunikatorów, SPA, (świetne jako backend dla Reacta), aplikacji mobilnych, a nawet gier - z wyłączeniem tych gier i projektów, które obciążających procesor i kartę graficzną.

*"Node.js helps NASA keep astronauts safe."*

Z Node.js korzystają też m.in:

Ebay  
Walmart  
Paypal  
Netflix  
Uber  
Linkedin

... i wiele innych projektów/firm

# Silnik Chrome V8

Zarówno przeglądarka Chrome jak i Node.js używają silnika Chrome V8 (otwarte rozwiązanie od Google) do kompilacji\* i egzekucji (wykonaniu kodu). Pamiętajmy jednak, że korzystanie z JavaScript w przeglądarce i w Node.js to nie tylko kompilowanie kodu JavaScript, ale i mnóstwo dodatkowych rozwiązań, które są poza silnikiem V8.

*\*JavaScript jest przez silnik V8 kompilowany do kodu maszynowego, ale po drodze jest jeszcze kod pośredni, bytecode. Cały proces jest zoptymalizowany przez silnik. Działanie silnika V8 obejmuje też inne rzeczy - jeśli masz ochotę znajdziesz na ten temat wiele informacji w sieci. Z naszej perspektywy liczy się to, że silnik v8 kompiluje i wykonuje program napisany w JavaScript i jest naprawdę szybki.*



# JavaScript w przeglądarce i w Node.js

Przeglądarka i Node.js to dwa różne środowiska, dlatego oferują różne możliwości.

Przeglądarka: obiekt window (w tym m.in. cookies, fetch API), DOM.

Node: pozwala na pracę z plikami, tworzenie web serwera, tworzenie REST API, pracę z bazami danych.

Oczywiście w Node.js i w przeglądarce to ciągle ten sam JavaScript (ECMAScript) - choć naturalnie poziom wdrożenia standardu może być różny w przeglądarkach i Node.js.

Są też pewne części wspólne w przeglądarce i w Node.js, które nie stanowią elementów specyfikacji ECMAScript np. funkcje czasu (setTimeout itd.) czy obiekt console.

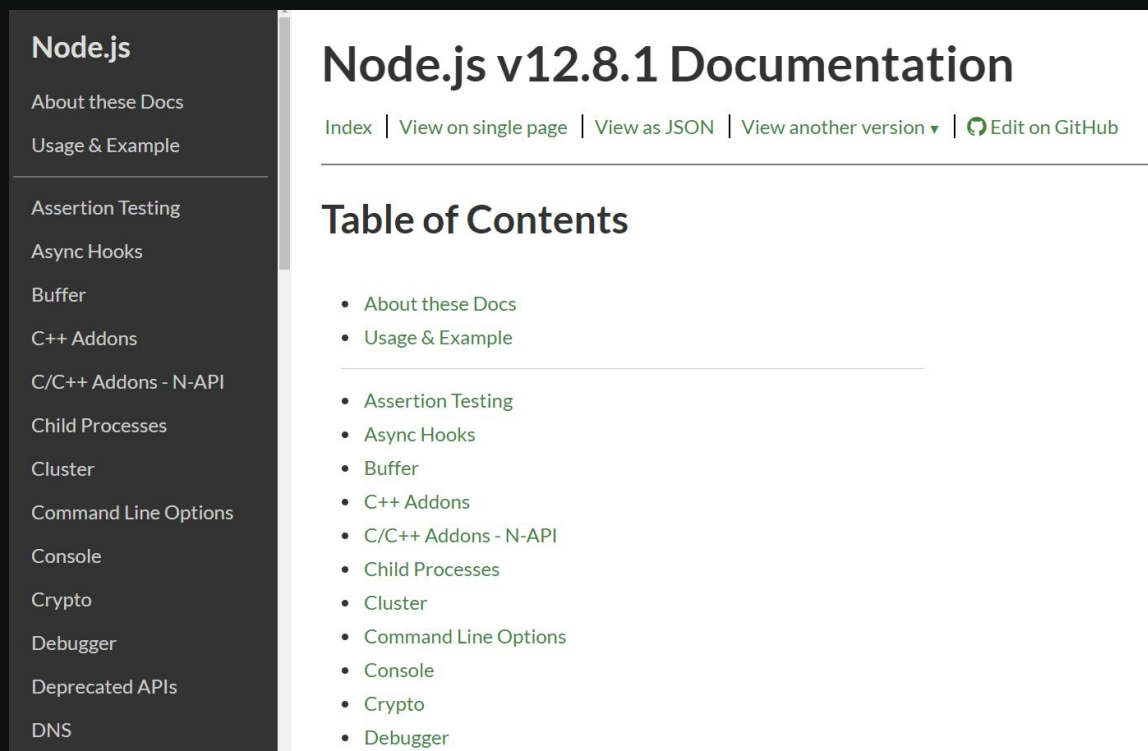
# Node.js - nie tylko V8 ale i moduły podstawowe

Moduły podstawowe (Node API).

W Node.js uzyskujemy dostęp do wbudowanych (podstawowych) modułów Node.js np. praca z plikami, możliwość pracy z siecią (TCP/IP, HTTP), tworzenie web serwera, praca z plikami, odczytywanie danych systemu operacyjnego, modułowość, funkcje czasu...

Ps. na tych modułach opiera się wiele innych (zewnętrznych) modułów.

więcej na <https://nodejs.org/docs/latest/api/>



**Node.js**

About these Docs  
Usage & Example

Assertion Testing  
Async Hooks  
Buffer  
C++ Addons  
C/C++ Addons - N-API  
Child Processes  
Cluster  
Command Line Options  
Console  
Crypto  
Debugger  
Deprecated APIs  
DNS

## Node.js v12.8.1 Documentation

[Index](#) | [View on single page](#) | [View as JSON](#) | [View another version ▼](#) | [Edit on GitHub](#)

### Table of Contents

- [About these Docs](#)
- [Usage & Example](#)

---

- [Assertion Testing](#)
- [Async Hooks](#)
- [Buffer](#)
- [C++ Addons](#)
- [C/C++ Addons - N-API](#)
- [Child Processes](#)
- [Cluster](#)
- [Command Line Options](#)
- [Console](#)
- [Crypto](#)
- [Debugger](#)

# Node.js - nie tylko V8 ale i biblioteka libuv

Libuv - biblioteka napisana dla Node.js. Obsługuje zdarzenia wejścia/wyjścia (I/O) poza jednowątkowym silnikiem V8, wykorzystując do tego możliwości systemu operacyjnego (API stanowią tu moduły podstawowe - Libuv jest implementacją).

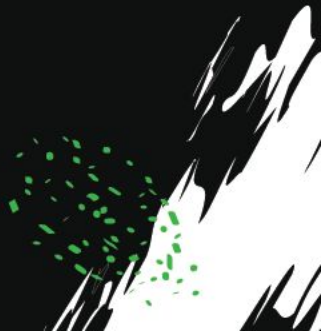
Libuv obsługuje zarówno Windows jak i systemu Unixowe, więc także Mac/Linux. Dzięki Libuv Node.js staje się asynchroniczny i może korzystać z możliwości systemu operacyjnego. Libuv wystawia po prostu część zadań poza Node.js (poza jeden wątek) do systemu operacyjnego. Nie blokuje to działania samego V8 i w praktyce znacznie rozszerza możliwości Node.js. Libuv dostarcza też do Node.js mechanizm pętli zdarzeń (event loop).



libuv



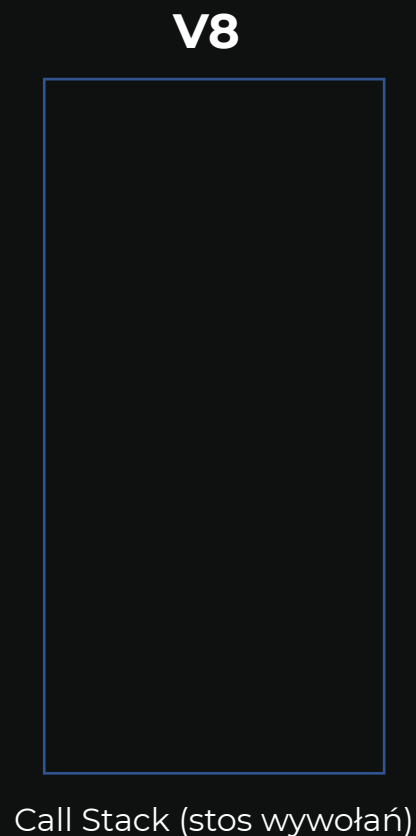
# Jednowątkowość i asynchroniczność



# Jednowątkowość i asynchroniczność

Javascript (Chrome V8) jest **jednowątkowy** i **synchroniczny**. Co to oznacza? Kod wykonywany jest po kolei, instrukcja po instrukcji. Za porządek (kolejność) wykonania kodu (funkcji) odpowiada mechanizm **call stack** (stosu wywołań).

Jednak zarówno przeglądarki jak i Node.js tworzą rozwiązania które pozwalają wiele operacji wykonać **asynchronicznie**, czyli w sposób, który nie blokuje wykonywania programu.



## Node API + biblioteka Libuv



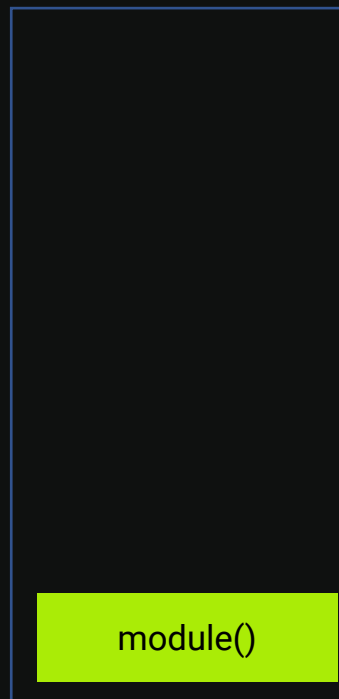
# Jednowątkowość i asynchroniczność



```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

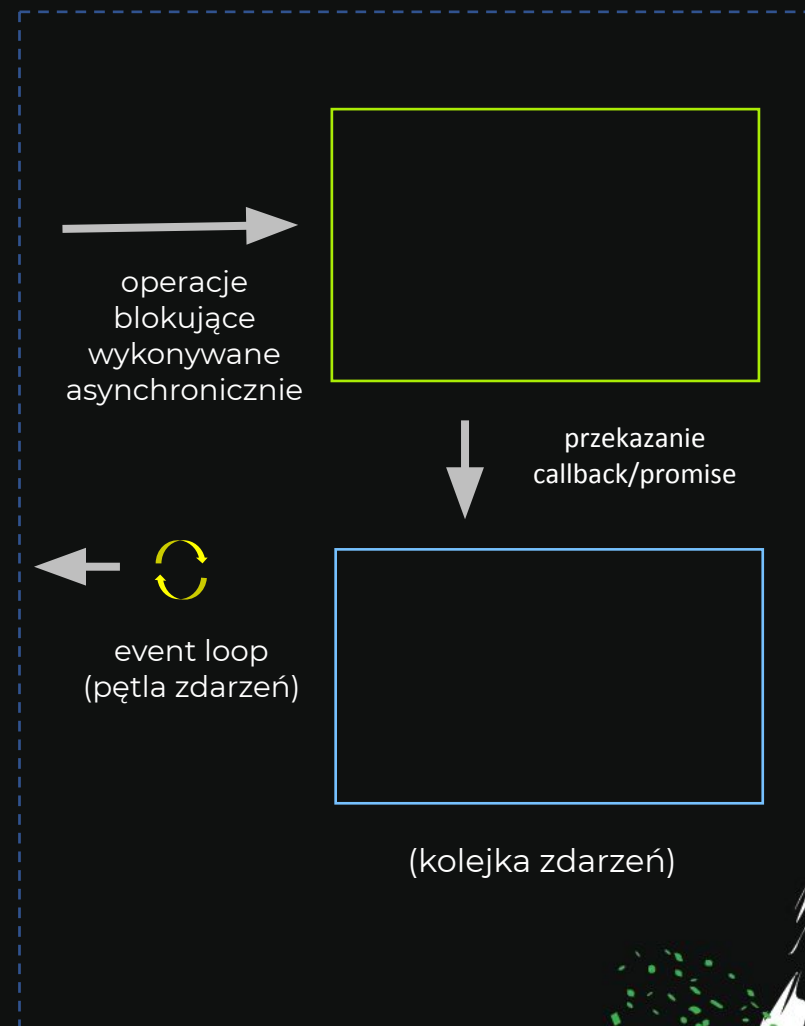
```
console.log("start");
```

V8



Call Stack (stos wywołań)

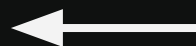
Node API + biblioteka Libuv



# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

```
console.log("start");
```

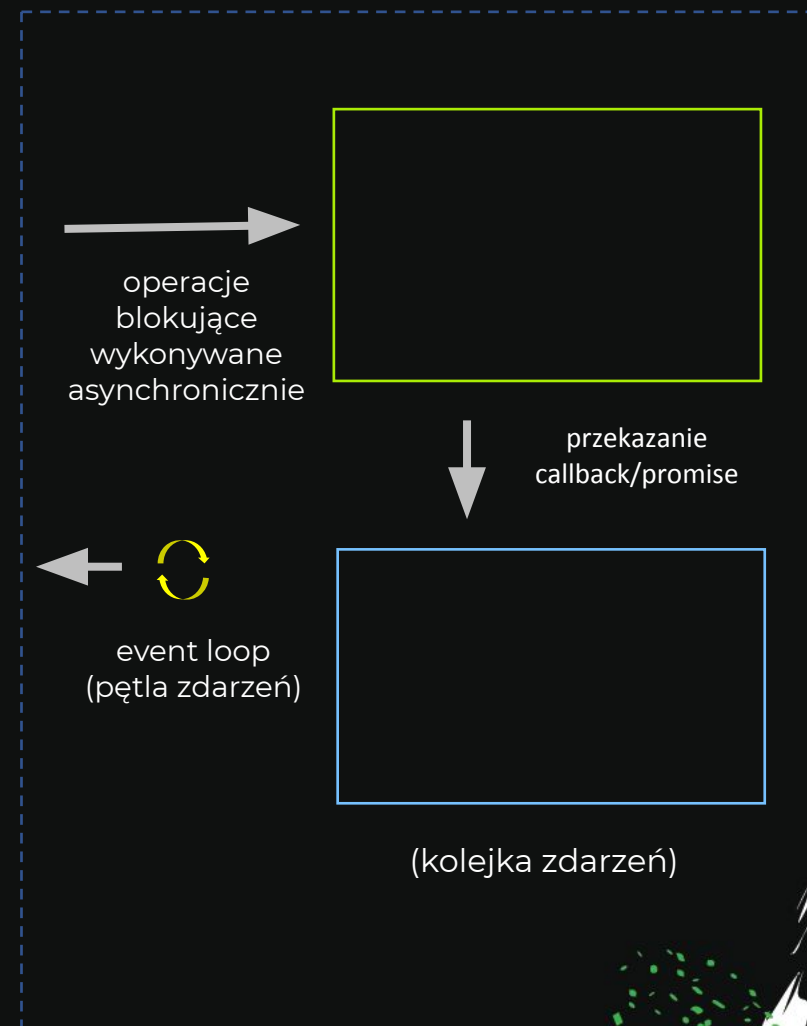


V8



Call Stack (stos wywołań)

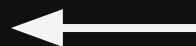
Node API + biblioteka Libuv



# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

```
console.log("start");
```

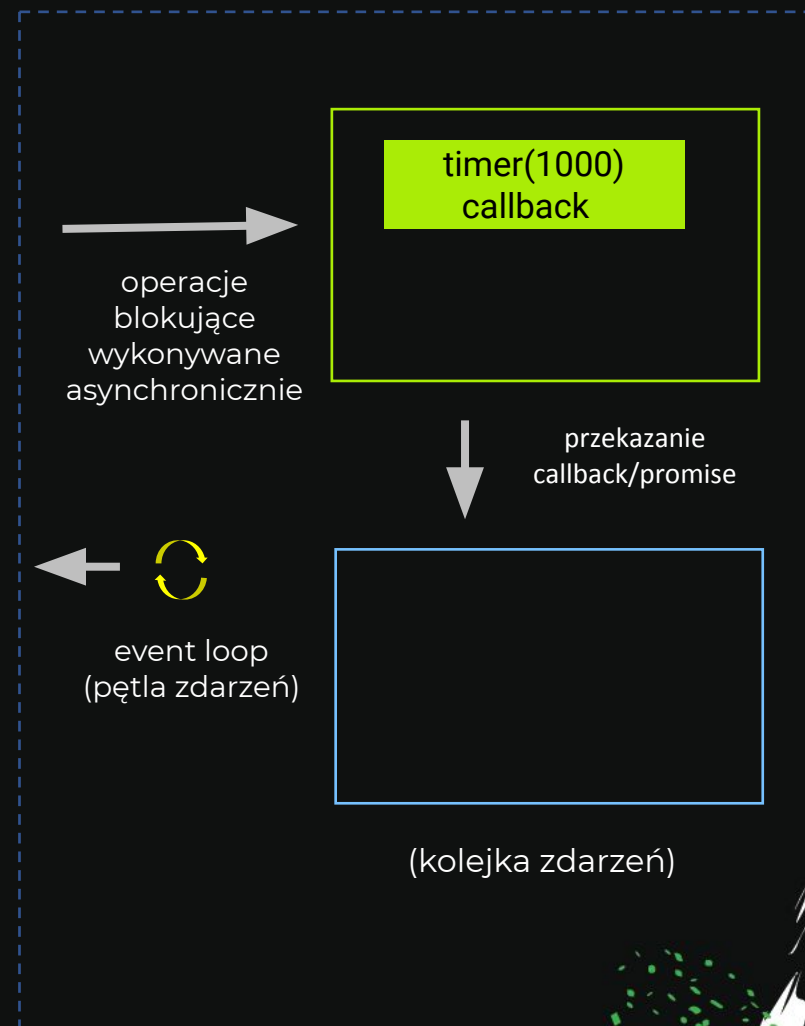


V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv

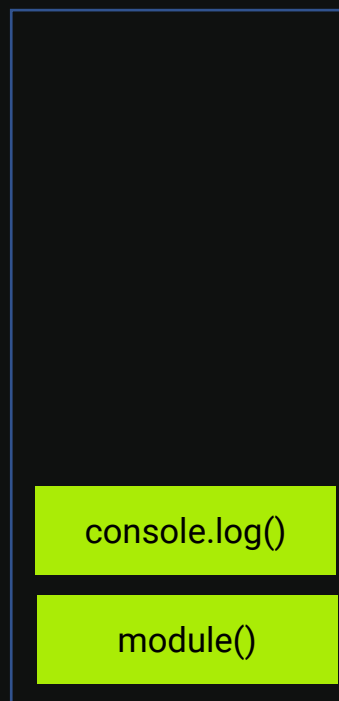


# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }, 1000  
);
```

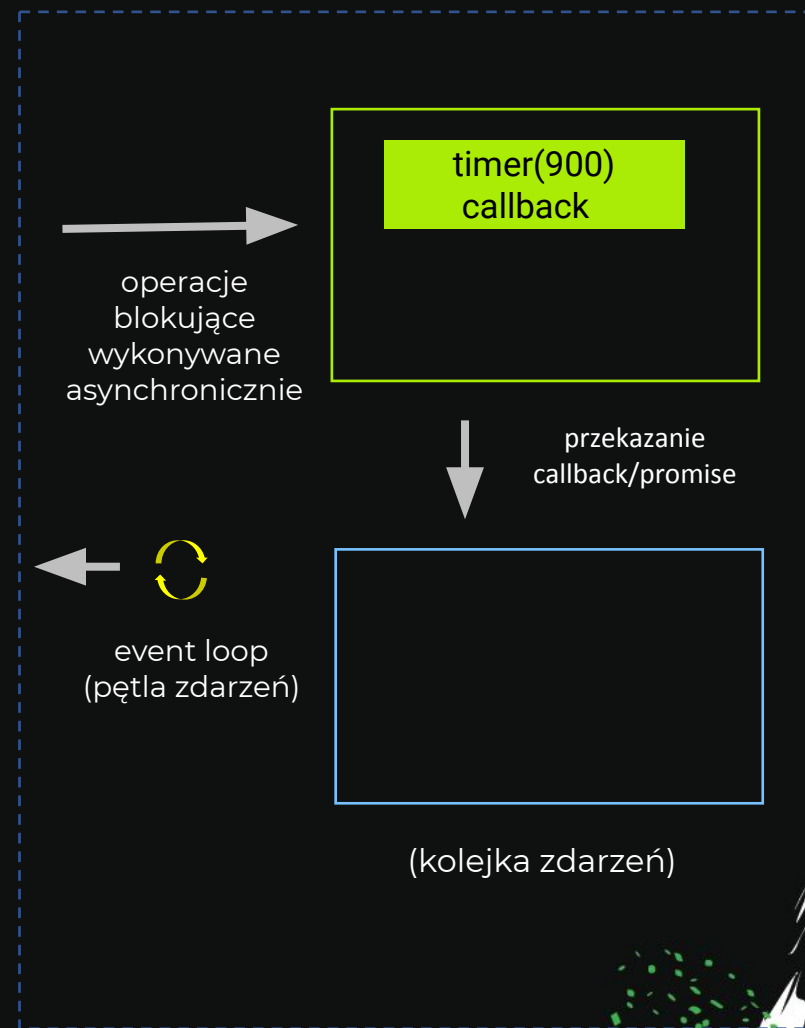
```
console.log("start");
```

V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv

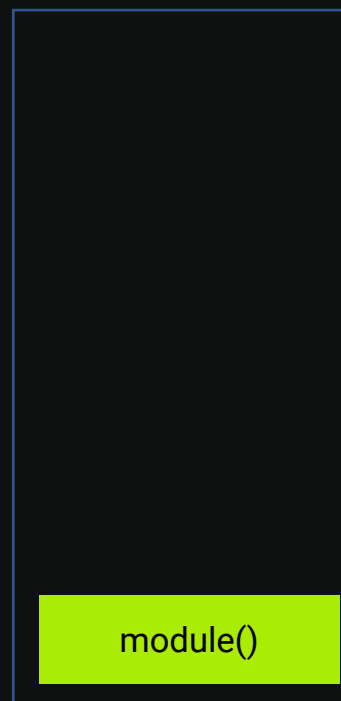


# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

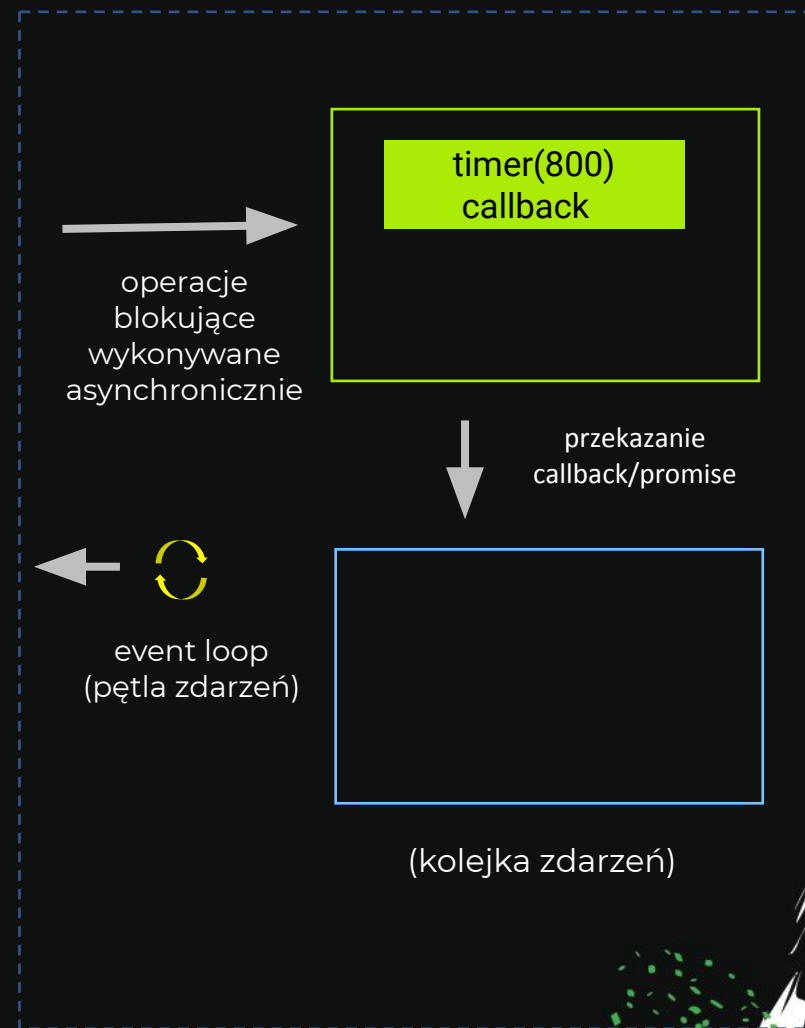
```
console.log("start");
```

V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv

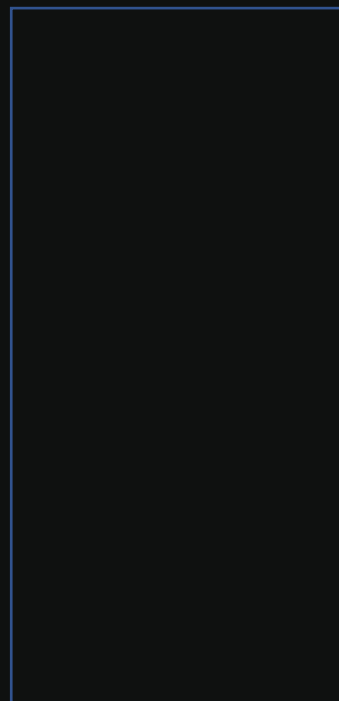


# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

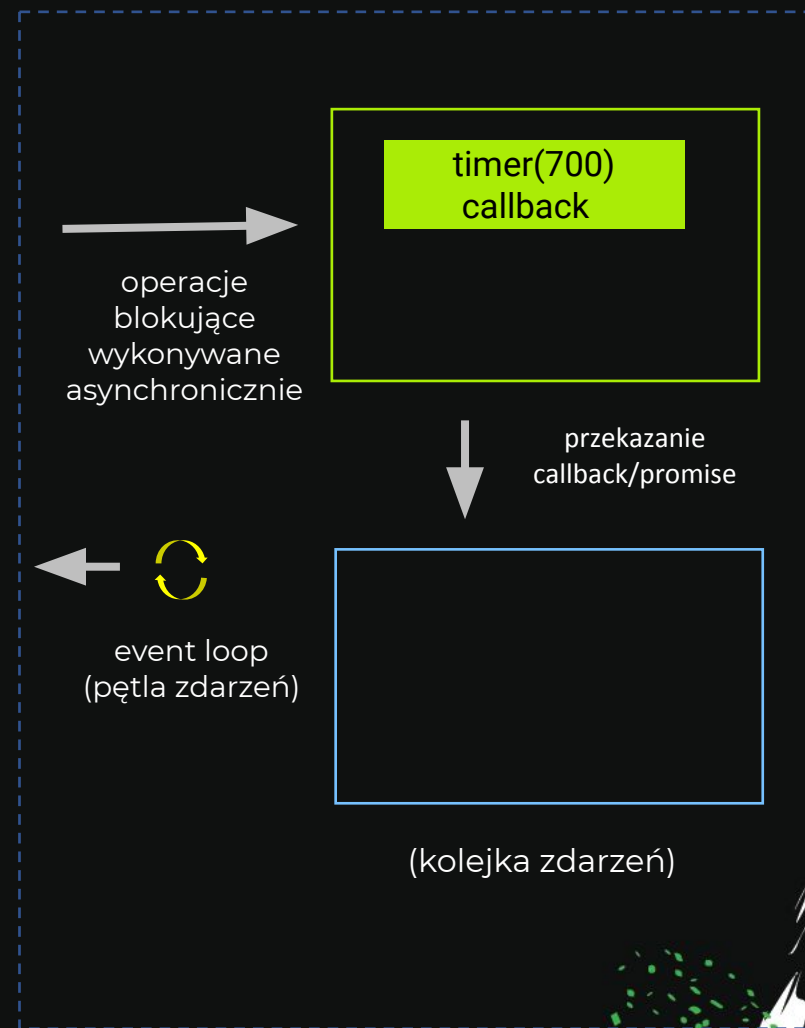
```
console.log("start");
```

V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv





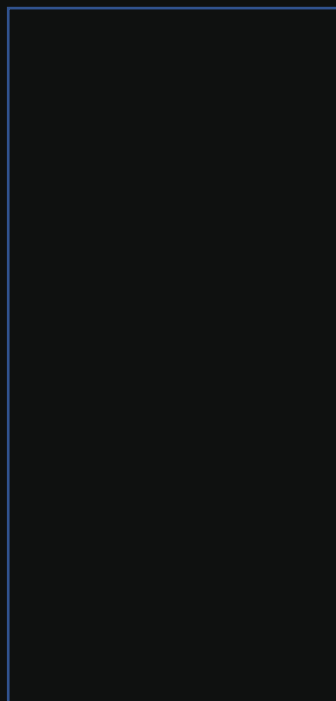
# Jednowątkowość i asynchroniczność

OCZEKIWANIE

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

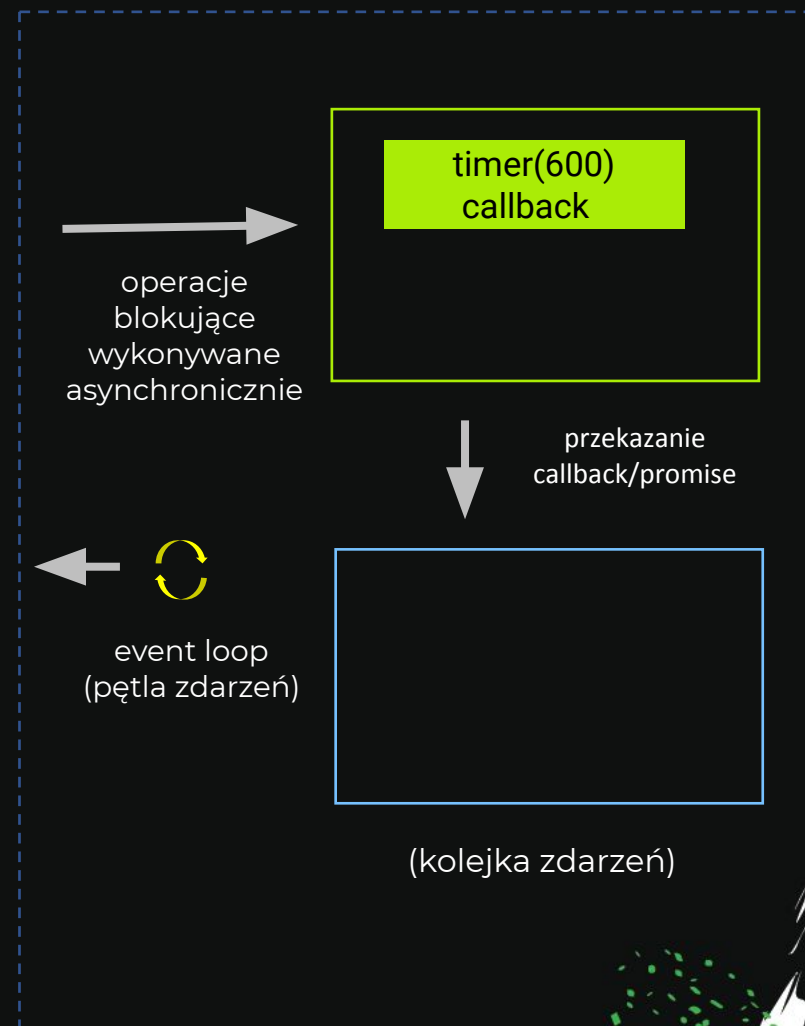
```
console.log("start");
```

V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv

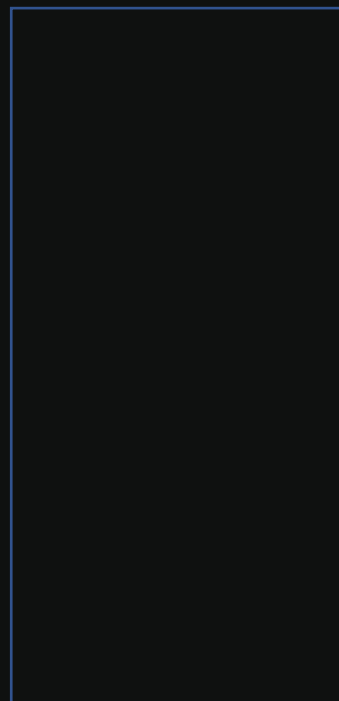


# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

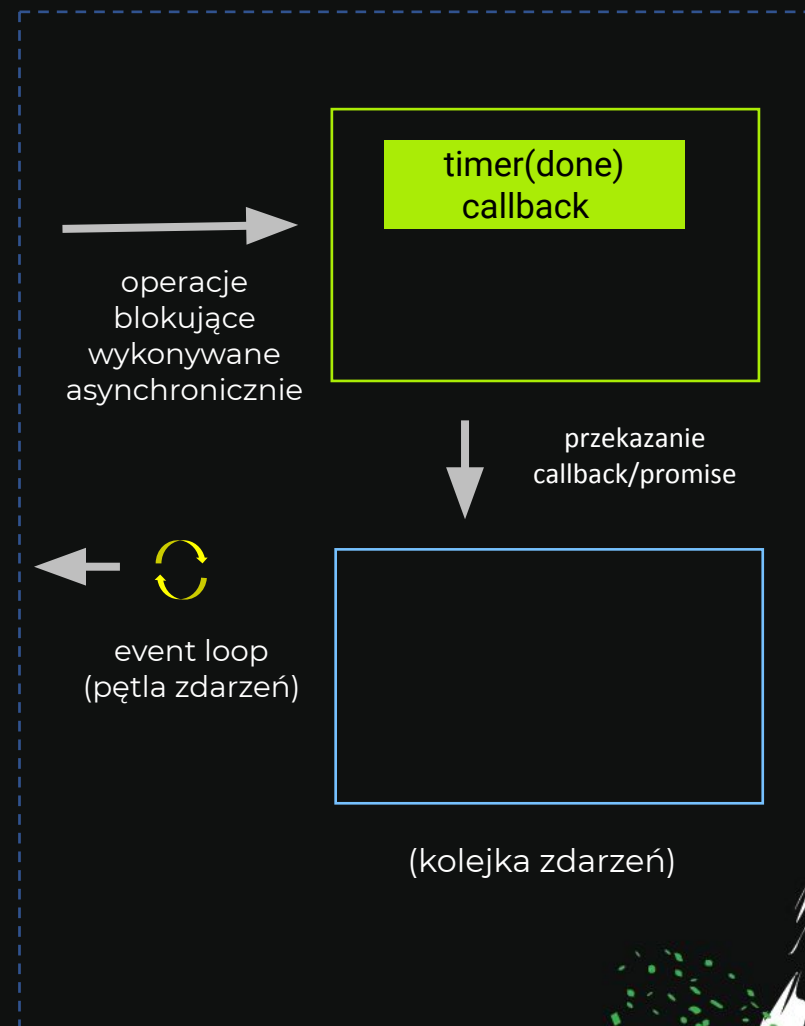
```
console.log("start");
```

V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv

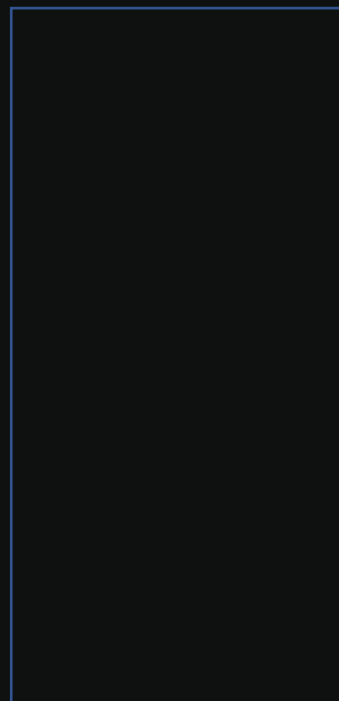


# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

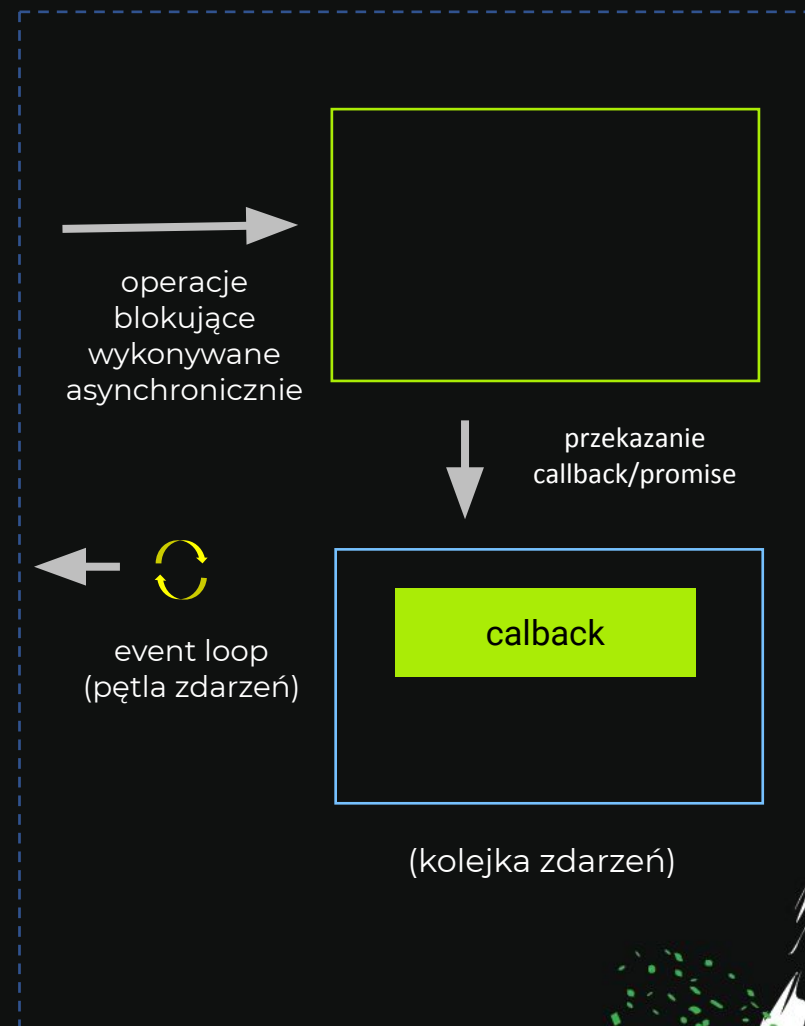
```
console.log("start");
```

V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv



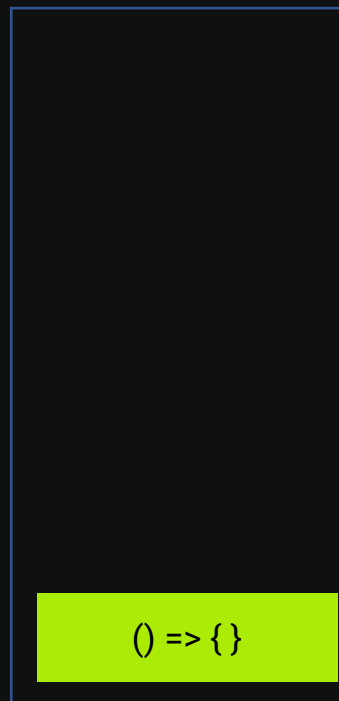
# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

```
console.log("start");
```



V8



Call Stack (stos wywołań)

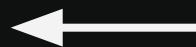
Node API + biblioteka Libuv



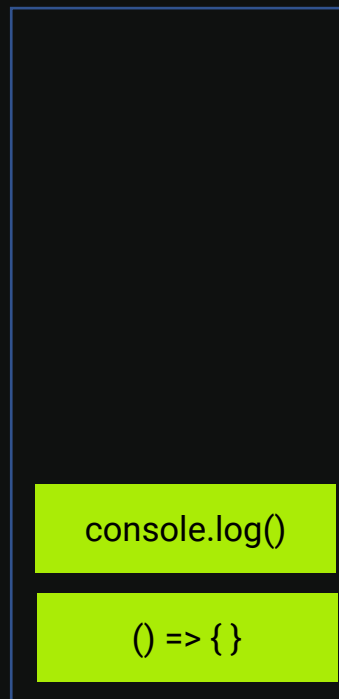
# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

```
console.log("start");
```



V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv



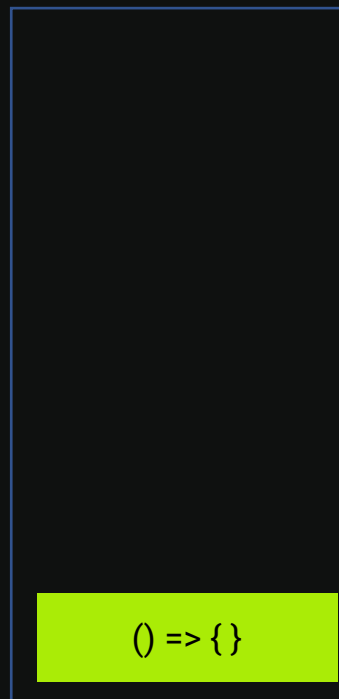
# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

```
console.log("start");
```

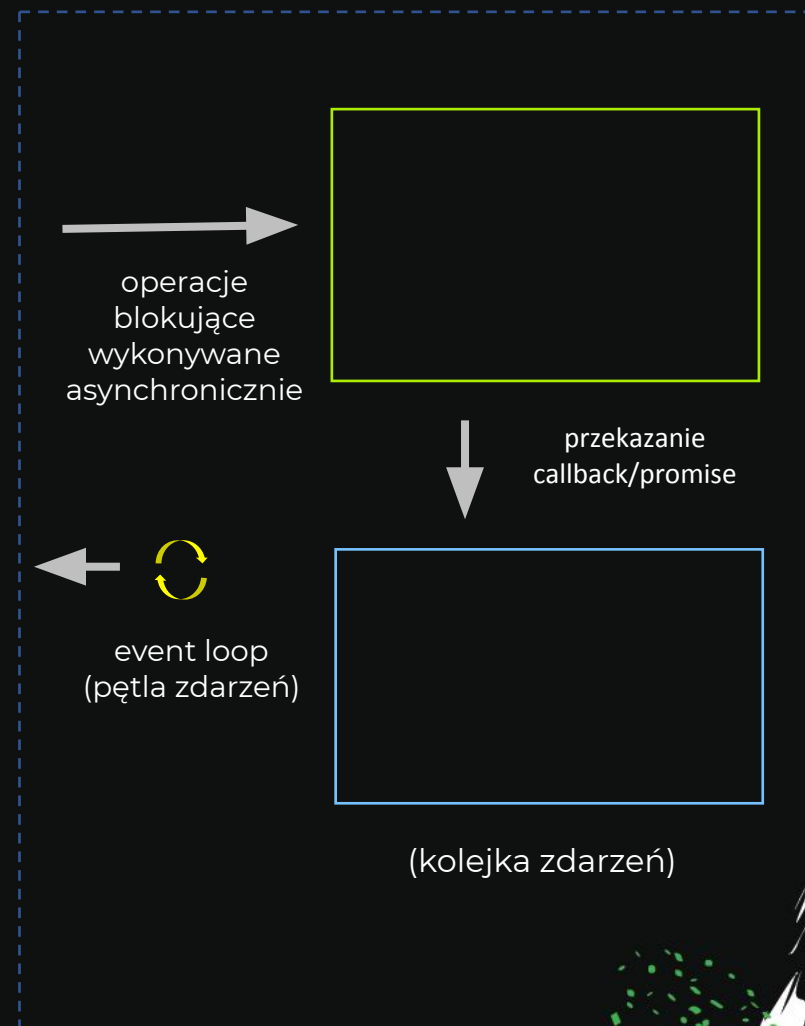


V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv

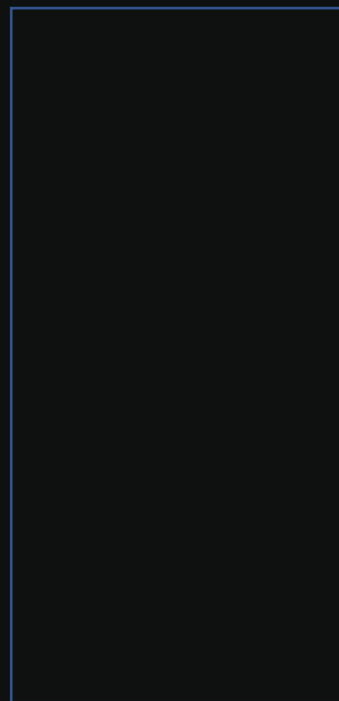


# Jednowątkowość i asynchroniczność

```
setTimeout(  
  () => {  
    const text = "done"  
    console.log(text);  
  }  
  , 1000  
);
```

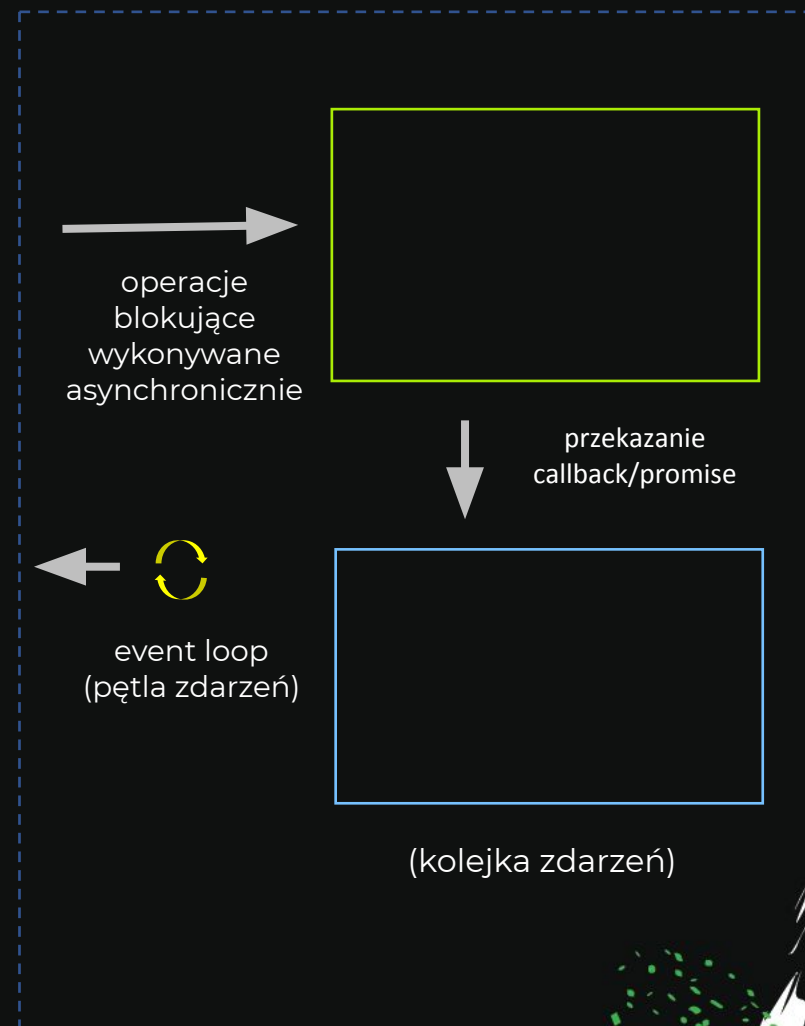
```
console.log("start");
```

V8



Call Stack (stos wywołań)

Node API + biblioteka Libuv





Zainstalujmy Node.js







Napiszmy i uruchommy “coś”



Zobaczmy call stack za pomocą debuggera VSC



# REPL w Node (i konsola w przeglądarce)

```
PS D:\projekty\kursy\css-js\npm> node
Welcome to Node.js v12.4.0.
Type ".help" for more information.
> const age = 30;
undefined
> age
30
> |
```

Interaktywny terminal do programowania w Node.js do którego dostęp otrzymujemy po wpisaniu **node** w konsoli.

Podobne rozwiązanie funkcjonuje w konsoli przeglądarki.



Node jest dostarczony razem z REPL JavaScript.


Znak **>** informuje, że jest uruchomiony.

CTRL + C - zamyka REPL

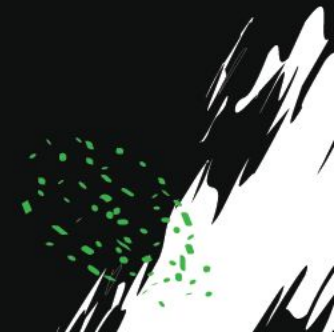



Napiszemy coś w REPL





O co chodzi: konsola, terminal, CLI (Command Line Interface), CLI  
(Command Line Interpreter), shell, powłoka, wiersz poleceń i Wiersz  
Poleceń



# terminal vs konsola vs wiersz poleceń vs CLI vs shell (powłoka)

**terminal** = **konsola** (**console**) -> program nasłuchująca tego co wpisujemy i wyświetlający na ekranie wynik naszych działań. W linux/Mac częściej mówimy o terminalu, w Windows o konsoli. Czasami mówimy też tutaj o CLI (Command Line Interface czyli Interfejsie wiersza poleceń)

**shell** = **CLI** (Command Line Interpreter) = **powłoka systemowa** -> Wszystko to jest określeniem interpretera. Interpretuje/przetwarza/wykonuje to co wpisaliśmy w terminal/konsolę. Pełni rolę pośrednika między terminalem/konsolą a systemem operacyjnym/programami. Tam trafia komendy z wiersza poleceń.

**wiersz poleceń** (**command line**) -> interfejs w terminalu/konsoli do którego użytkownik wpisuje komendy/polecenia. To co wpisujemy w wiersz poleceń (po wciśnięciu enter) jest przekazywane do powłoki a potem zwracane do terminalu/konsoli. "Wierszem Poleceń" nazywa się też wbudowana w Windows konsola (w wersji PL w wersji EN "Command Prompt" czy "CMD").



0 co chodzi: Bash, Git Bash, PowerShell, CMD



# PowerShell

## Command Prompt (CMD, Wiersz Poleceń)

## Bash - systemy Unixowe

Najpopularniejsze interfejsy i interpretatory poleceń (powłoki) do pracy z terminalem/konsolą.

Różnią się zasobem komend (możliwościami i składnią), ale na tym etapie wybór nie jest najważniejszy (choć najczęściej rekomenduje dla Windows PowerShell a dla Linux/Mac Bash).

CMD , PowerShell - oba wbudowany w Windows (PowerShell od wersji 10).

Bash - możliwość użycia na Windows także w wersji GitBash dostarczonej przy instalacji Gita (istnieje możliwość zainstalowania też Basha unixowego)

Każdy interpreter posiada własny terminal.

Integracja terminala w Visual Studio Code

```
Wiersz polecenia
Microsoft Windows [Version 10.0.18362.295]
(c) 2019 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\work>
```

```
MINGW64:/d/projekty/kursy/css-js/examples

work@WINDOWS-HM9T6SF MINGW64 /d/projekty/kursy/css-js
$ cd examples

work@WINDOWS-HM9T6SF MINGW64 /d/projekty/kursy/css-js/examples
$ ls
BEM-nav/      CSSGrid/      fetch-project/  sass-basics/
BEM-project/  dom/          Flexbox/        sass-scss/
css/          dom-project/  nav-BEM/        Sass-start-projektu/
```

```
Wybierz Windows PowerShell

PS C:\Users\work> mkdir project-1203
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\projekty\kursy\css-js\npm> npm init --yes
```





Jak zbudowane są polecenia, których używamy w wierszu poleceń

# Jak zbudowane jest polecenie w wierszu poleceń

To co tu pokazuje oparte jest na konwencji. Co z tego wynika?  
Że zawsze można użyć w innym kontekście pokazane tu pojęcia  
(więc w jednym zespole/dla innego developer może oznaczać  
coś innego).

- **polecenia** (komendy) - albo całe wyrażenie w wierszu poleceń, albo element tego wyrażenia.
- **argumenty** (parametry)
- **opcje** - **flagi** (czasami określane też **przełącznikami** - flags/switches) - **krótkie** (jednoliterowe z prefiksem "-") i **długie** (wieloliterowe, najczęściej z prefiksem z dwoma myślnikami "--"). Czasami do flag dodawane są **argumenty** **flagi**. (nie wszystkie flagi używają argumentów).

//Każde z poleceń poniżej robi to samo

`npm install nazwa-paczki --save-dev`

`npm i nazwa-paczki -D`

`npm i -D nazwa-paczki`

// Częsty schemat polecenia w wierszu poleceń

`polecenie argument --flaga`

`polecenie argument --flaga=argument`

`polecenie argument --flaga="argument"`

`polecenie argument --flaga argument`

`polecenie argument -flaga`

`polecenie argument -f argument`



Zobaczmy jak pracuje się w konsoli



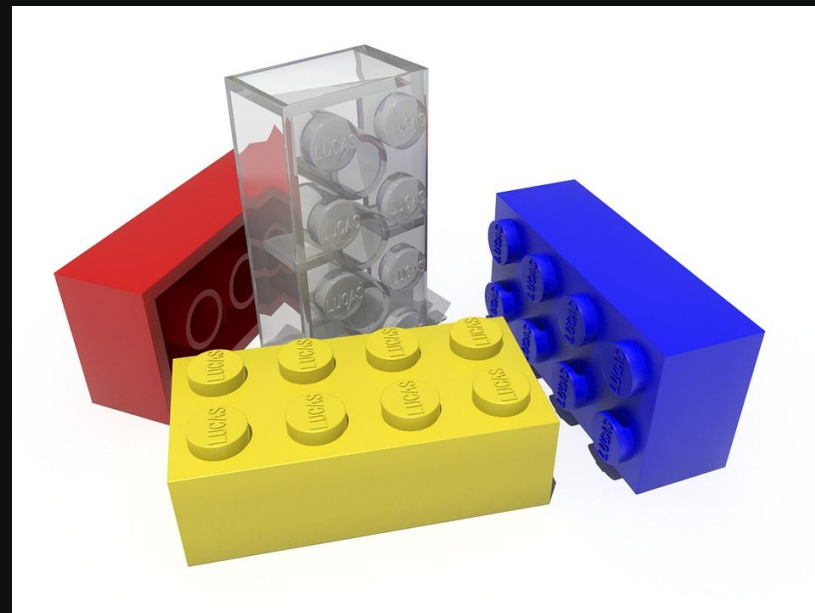
# MODUŁY W NODE



# Moduł w Node.js - niezależne bloki programu

Z **technicznego punktu widzenia** moduł to kod umieszczony w jednym pliku. Taki plik posiada własny zakres (zakres/scope określa gdzie dany kod jest dostępny), jest więc odseparowany od innych fragmentów kodu (hermetyzacja).

Elementy zdefiniowane w module mają charakter **prywatny** i nie są widoczne poza modulem z wyjątkiem elementów udostępnionych, czyli tych np. funkcji, które zdecydujemy się upublicznić (**eksportować z modułu**).



# Każdy moduł jest osobnym elementem programu

Z punktu widzenia **techniki tworzenia oprogramowania** moduł jest osobnym elementem programu (może być plikiem lub katalogiem), który cechuje to, że może być używany **wielokrotnie** oraz, że spełnia jakąś funkcję wykorzystywaną w programie.

Szukając analogii w świecie rzeczywistym możemy każdą (spełniającą jakąś rolę, wymienną) część samolotu traktować jako moduł, a cały samolot jako program. Poszczególne części (moduły) są używane w relacji z innymi częściami poprzez swój interfejs (API) - czyli elementy programu, które importujemy/eksportujemy.



# Moduł - implementacja i eksport w Node.js

Moduł jest programem. I jako taki nie musi być (i nie powinien być) dostępny w całości dla innych części programu. Inny moduł (inną część programu) interesują tylko udostępnione elementy (funkcje najczęściej) a nie sposób implementacji.

Jeśli używamy moduły (a nie go tworzymy), to nie interesuje nas implementacja tzn. jak dany moduł jest napisany, tylko do czego i jak go użyć.

Mówimy, że **moduł udostępnia swoje API (lub po prostu interfejs)**, z którego korzystamy. API jest w przypadku modułu zestawem publicznych funkcji i właściwości oferowanym przez moduł do użycia przez program/programistę poza modulem.



# Własne moduły, moduły wbudowane i moduły innych

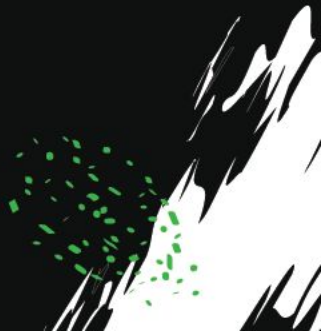
Moduły możemy klasyfikować na wiele sposobów.

- **własne moduły** - napisane przez nas
- **moduły podstawowe** - dostarczone wraz z Node.js. Wymagają zaimportowania w module w którym chcemy ich użyć (lub w niektórych przypadkach są dostępne globalnie).
- **moduły (package/pakiety) stron trzecich** - dostępne (najczęściej) poprzez **npm** i wymagają zainstalowania bądź to lokalnie (w projekcie i wtedy też importowanie w module np. bootstrap) bądź globalnie (dostępne w każdym projekcie). Sposób instalacji jest opcjonalny.





napiszmy kilka modułów i użyjmy ich w innym module



# Specyfikacja CommonJS i ES Modules

# Dwa najważniejsze sposoby pracy z modułami w JavaScript

CommonJS - Aktualnie (08.2019) podstawowy sposób na pracę z modułami w Node.js. Jego interfejs opiera się na funkcji `require` (do importowania) i właściwości `module.exports` do eksportowania z modułu.

ES Modules - JavaScript zaczął wspierać natywnie moduły od 2015 roku (część specyfikacji ECMAScript 6). Poziom wdrożenia w przeglądarkach jest całkiem niezły (około 85% z tagiem `<script type="module">`), ale Node.js nie ma obecnie bezpośredniego wsparcia dla natywnych modułów\*. API modułów oparte jest o słowa kluczowe `import` i `export`.

*\* można używać składni ES Modules w Node.js, ale nie bezpośrednio. Możemy to zrobić używając transpilatora Babel, uruchomić tryb eksperymentalny w Node.js lub skorzystać z paczki esm (dostępnej w npm). Od wersji 12 Node, po osiągnięciu przez nią statusu LTS (Long Term Support), ES Modules będą dostępne także bezpośrednio w Node.js. Oczywiście nadal będzie, pewnie przez wiele lat wsparcie dla CommonJS.*

# Użycie modułu w innym module

Jeśli w jednym module chcemy wykorzystać publiczne API innego modułu (czyli eksportowane przede wszystkim jego metody), to musimy go (ten moduł) w Node.js importować za pomocą funkcji `require()`.

Jeśli importujemy własne moduły, to musimy też określić co będzie z nich eksportowane (przypisać do `module.exports`), ponieważ domyślnie eksportowany jest pusty obiekt `{}`.

```
modules > JS example.js > ...  
1  const value = 100;  
2  
3  module.exports = value;
```

# require('./path'); require('name')

require(), to funkcja do której przekazujemy ścieżkę. Wyszukuje ona moduł.

`require('jQuery');` `require('fs');` - najpierw sprawdza czy dany moduł jest modulem podstawowym, jeśli nie, to szuka pliku (lub katalogu) w folderze w `node_modules` naszego lokalnego projektu (i idzie coraz wyżej). Jeśli w ten sposób nie znajdzie to szuka w katalogu w którym zainstalowany jest Node.js (`/lib/node`). Jeśli nie znajdzie, to mamy błąd.

`require('./users');` `require('../server');` `require('./components/header');` - ścieżka do pliku (nie wymaga podania rozszerzenia js, ponieważ go szuka w pierwszej kolejności, jeśli nie znajdzie sprawdzi też czy jest taki plik z rozszerzeniem .json)

require(), wczytuje dany moduł. Jeśli środowisko uruchomieniowe nie znajdzie danego modułu, to wyrzuci błąd (*Error: Cannot find module 'nazwa modułu'*).

# Moduły - przykład użycia zgodnego ze specyfikacją CommonJS i ES Modules

```
/* --- Składnia CommonJS --- */  
//w module do którego importujemy  
const products = require('./products');  
  
// w module z którego eksportujemy (products.js)  
module.exports = () => { /* ... */ };  
  
/* --- Składnia ES Modules --- */  
//w module do którego importujemy  
import products from './products';  
  
// w module który importujemy (products.js)  
export default () => { /* ... */ };
```



Zrozumieć implementacje modułów

Dlaczego moduły są niezależne od siebie?



# modul.js - nasz przykładowy plik

Założmy, że stworzyliśmy plik module.js i w środku umieściliśmy kod.

```
// ... kod naszego modułu
```

Wiesz już, że moduł jest **prywatny** i nie widzi się z innymi modułami, ale (zapewne) nie wiesz, że wynika to ze sposobu implementacji. Dzieje się tak ponieważ cały twój kod z pliku jest umieszczany przez Node.js w funkcji. Pamiętajmy, że funkcja tworzy własny zakres (scope), a to co się w niej znajduje jest prywatne i nie jest bezpośrednio widoczne i dostępne na zewnątrz funkcji. Funkcja też coś zwraca.



# moduł - scope funkcji

Node implementuje moduł w ten sposób, że owija go funkcją.

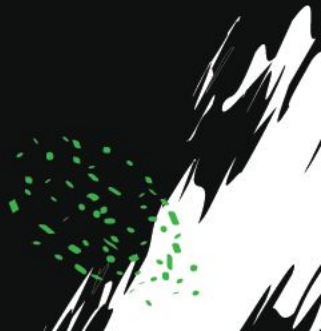
```
(function () {  
    /* kod naszego modułu (to co widzimy i piszemy w pliku) */  
})
```



moduł - anonimowa funkcja z argumentami

Przekazuje też argumenty do środka.

```
(function (exports, require, module, __filename, __dirname) {  
    // ... kod naszego modułu  
})
```





Przejdźmy do VSC i  
zobaczmy zmienne dostępne w module


# Zapamiętajmy - CommonJS i implementacja funkcji

```
const search = require('./search');
```

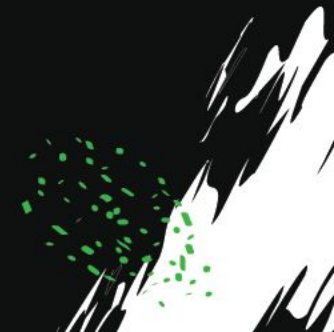

Jedynym argumentem jakim podajemy do funkcji require jest ścieżka do modułu.


Po wykonaniu funkcji opakowującej zwracana jest wartość przypisana do module.exports w danym module (i w naszym przypadku ta wartość jest przypisywana do stałej search)

```
(function (exports, require, module, __filename, __dirname) {  
  // ... kod naszego modułu  
  return module.exports;  
})
```

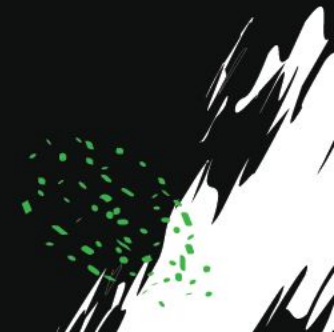



Importowanie - uzyskiwanie dostępu do API  
modułu z innego modułu za pomocą funkcji  
`require()`





eksportowanie - udostępniania API modułu  
(funkcje, obiekty, właściwości) dla innych  
modułów. Użycie obiektu `module.export` lub  
`exports` - utrwalenie



# require() - co jeszcze warto wiedzieć

- Jest synchroniczna, ponieważ zakładamy, że importowany moduł będzie potrzebny przy dalszym wykonywaniu kodu. Nie jest to problem, ponieważ używamy ich (powinniśmy używać) przy początkowym wczytaniu (uruchomieniu) aplikacji.
- Umieszczamy (najczęściej) na początku modułu w których używamy
- Najczęściej używamy tej samej nazwy dla zmiennej co nazwa importowanego modułu. Jest to dobra konwencja i pozwala uczynić kod czytelniejszym.

```
const users = require('./users'); //moduł znajdujący się w tym samym katalogu o nazwie users  
(w pliku users.js)  
const util = require('util'); //moduł podstawowy util
```

# exports i module.export

`module.exports` i `exports` (właściwości dostępne w każdym module), mają referencje do tego samego, początkowo pustego, obiektu. Jeśli nic nie ustawimy moduł domyślnie będzie więc udostępniał pusty obiekt. Pamiętajmy że z modułu zwracana jest zawartość `module.exports`.

```
// module.exports === exports //true - referencja do tego samego, pustego obiektu
```

```
/* --- users.js --- */
```

```
module.exports = () => console.log("udostępnione z modułu users.js")
```

```
//exports nadal prowadzi do pustego obiektu w takim przypadku (połączenie zostało zerwane)
```

```
// module.exports === exports //false
```

```
/* --- app.js --- */
```

```
const users = require('./users'); // do user zostanie przypisana zwracana funkcja  
users(); // w konsoli pojawi się "udostępnione z modułu users.js"
```



# exports i module.export

```
/* --- math.js --- */  
module.exports.add = () => {}  
exports.multiply = () => {}  
// module.exports === exports //true - referencja ciągle do tego samego obiektu, który aktualnie  
//posiada dwie właściwości z przypisanymi funkcjami (czyli metody)/  
//pamiętajmy, że zwracane z funkcji jest to co jest przypisane do module.exports
```

```
/* --- app.js --- */  
const math = require('./math'); // do user zostanie przypisany obiekt posiadający dwie metody add  
i multiply.  
math.add();  
math.multiply();
```

# Moduły podstawowe

# Dostępne bez instalowania ich w projekcie

Moduły podstawowe (core module, moduły wbudowane) są dostępne w modułach projektu bez ich instalowania (czyli bez *npm install nazwa-modułu* - które poznasz w sekcji *npm*). Możemy z nich korzystać (otrzymuje dostęp do ich API) w każdym module, o ile zostaną do niego zaimportowane za pomocą funkcji `require()`


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

```
fs.mkdir();  
os.userInfo();  
http.createServer();
```

# Część modułów/funkcji dostępna bezpośrednio

Część rozwiązań nie wymaga nawet `require` (choć nadal jest to możliwe), bo są dostępne globalnie np. instancja `console` (pochodząca z modułu `Console`), czy funkcje czasu jak `setTimeout` oraz `setImmediate` (pochodzące z modułu `Timers`).

Te funkcje są nam znane z przeglądarki, ale warto wiedzieć, że po pierwsze ich implementacja jest różna, a po drugie lista dostępnych możliwości (np. właściwości w `console`) często nie jest taka sama, zazwyczaj Node.js oferuje tutaj dodatkowe rzeczy jak w przykładzie powyżej funkcja `setImmediate`.



Skorzystajmy z modułów podstawowych na przykładzie metody `readFile` dostępnej z poziomu modułu `File System`

- przejdźmy do VSC
- 
- 

# Przykład użycia modułu do odczytu pliku

```
/* plik index.js */
```

```
//importowanie modułu podstawowego do modułu (nie wymaga instalacji moduły File System, wystarczy  
użycie metody require i jego zaimportowanie
```

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

```
//użycie metody asynchronicznej
```

```
fs.readFile('./users.json', 'utf8', (err, file) => console.log(file));
```

```
// wyświetli się przed zawartością pliku, ponieważ metoda readFile nie blokuje programu
```

```
console.log("przed wyświetleniem zawartości pliku");
```

```
// po wpisaniu node index.js uruchomimy ten kod w Node.js
```