Helion

SMASHING MAGAZINE

Guillermo Rauch

PODRĘCZNIK
Node.js

SMASHING MAGAZINE

WYKORZYSTAJ POTENCJAŁ NODE.JS!

Tytuł oryginału: Smashing Node.js: JavaScript Everywhere

Tłumaczenie: Krzysztof Wołowski

ISBN: 978-83-246-6674-4

This edition first published 2012

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/podnod.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/podnod
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

5 WIERSZ POLECEŃ I MODUŁ FS: TWOJA PIERWSZA APLIKACJA

W TYM ROZDZIALE ZAJMIEMY SIĘ jednymi z najważniejszych interfejsów programistycznych Node.JS: interfejsami związanymi z obsługą strumienia wejściowego (`stdin`) i wyjściowego (`stdout`) procesu oraz interfejsami związanymi z systemem plików (moduł `fs`).

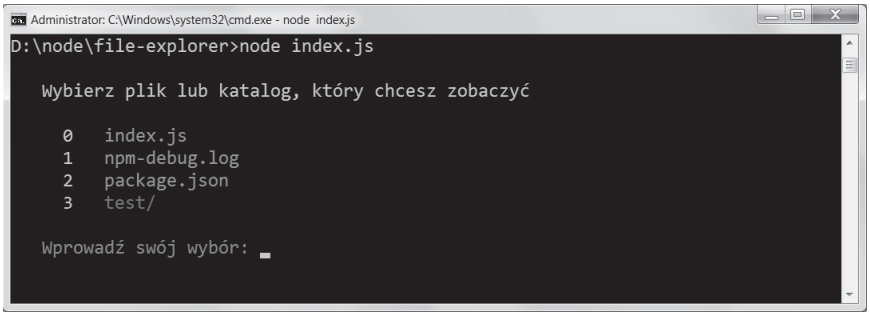
Jak już wiemy z poprzedniego rozdziału, kluczowe w sposobie obsługi współbieżności przez Node jest użycie wywołań zwrotnych i zdarzeń. Dzięki tym interfejsom poznasz kontrolę przepływu w procesie programowania z wykorzystaniem zdarzeń i nieblokujących operacji wejścia-wyjścia.

Wiedzę na temat tych interfejsów i ich interakcji sprawdzisz, tworząc swoją pierwszą aplikację: prosty, uruchamiany z wiersza poleceń eksplorator plików, który umożliwi użytkownikowi tworzenie nowych oraz odczyt zawartości istniejących plików.

WYMAGANIA

Na początek określ, jakie zadania powinien wykonywać program:

- Chcesz, żeby program uruchamiany był z wiersza poleceń. Oznacza to, że będzie uruchamiany albo za pomocą polecenia `node`, albo bezpośrednio, a dalsza interakcja z użytkownikiem będzie się odbywać przez terminal.
- Po uruchomieniu program powinien wyświetlić listę bieżących katalogów (zob. rysunek 5.1).



Rysunek 5.1. Lista bieżących katalogów wyświetlana przy starcie programu

- Po wybraniu pliku program powinien wyświetlić jego zawartość.
- Po wybraniu katalogu program powinien wyświetlić jego podkatalogi.
- Następnie program powinien się zakończyć.

Biorąc pod uwagę powyższe, projekt można rozbić na kilka mniejszych etapów:

1. Utworzenie naszego modułu.
2. Wybranie synchronicznej lub asynchronicznej wersji modułu `fs`.
3. Zrozumienie strumieni.
4. Przeprowadzenie operacji wejścia i wyjścia.
5. Refaktoring.
6. Interakcja z modułem `fs`.
7. Dopracowanie szczegółów.

PISZEMY NASZ PIERWSZY PROGRAM

Zbudujesz teraz moduł na bazie wymienionych powyżej kroków. Moduł będzie złożony z kilku plików, które utworzysz za pomocą dowolnego edytora tekstu.

Pod koniec tego rozdziału będziesz dysponować w pełni funkcjonalnym programem, napisanym w całości w Node.JS.

TWORZYMY MODUŁ

Jak w każdym przykładzie w tej książce, zaczniemy od utworzenia katalogu zawierającego nasz projekt. Na potrzeby przykładu nazwiemy go *file-explorer*.

W poprzednich rozdziałach wspomnieliśmy o dobrej praktyce definiowania pliku *package.json* dla każdego projektu. Zachowujesz w ten sposób kontrolę nad zależnościami określonymi w rejestrze NPM i możliwość publikacji modułów w przyszłości.

Chociaż w naszym przykładzie będziemy korzystać tylko z wbudowanych modułów Node (a więc niepobieranych z rejestru NPM), musimy przygotować prosty plik `package.json`:

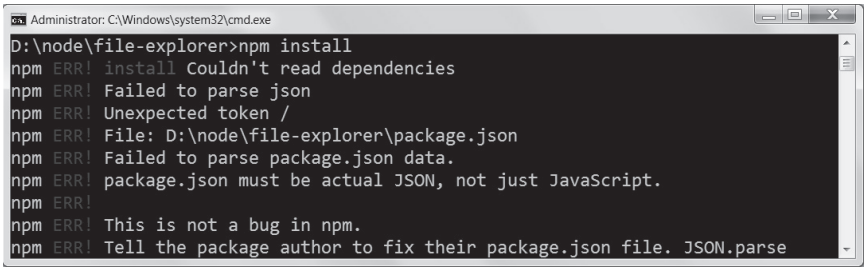
package.json

```
{
    "name": "file-explorer"
  , "version": "0.0.1"
  , "description": "Eksplorator plików w wierszu poleceń!"
}
```

> *Uwaga: NPM wprowadza numerację kontroli wersji według tzw. konwencji* ***semver****. To dlatego zamiast „0.1" lub „1" w polu* `version` *podajemy wartość „0.0.1".*

Aby zweryfikować poprawność pliku *package.json*, wydaj polecenie `$ npm install`.

Jeżeli wszystko działa, nie powinny zostać wyświetlone żadne błędy[1]. W innym razie pojawi się wyjątek JSON (zob. rysunek 5.2).



Rysunek 5.2. Uruchomienie polecenia npm install z niepoprawnym kodem JSON w pliku package.json

W następnej kolejności utworzysz plik JavaScript *index.js*, który będzie zawierał podstawowy kod programu.

[1] Aczkolwiek mogą zostać wyświetlone ostrzeżenia — *przyp. tłum.*

SYNC CZY ASYNC?

Na początek zadeklaruj w swoim pliku zależności. Ponieważ interfejsy `stdio` są częścią zmiennej globalnej `process`, jedyną zależnością będzie moduł `fs`:

index.js

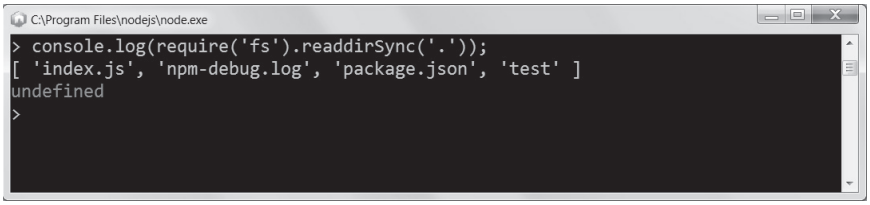
```
/**
 * Zależności modułu.
 */

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

Pierwszym zadaniem po uruchomieniu programu będzie uzyskanie listy plików w bieżącym katalogu.

Musisz przy tym pamiętać, że interfejs programistyczny `fs` jest wyjątkowy w tym sensie, że pozwala zarówno na blokujące, jak i nieblokujące wywołania. Jeśli na przykład chcesz pobrać listę istniejących katalogów, możesz to zrobić w następujący sposób:

```
> console.log(require('fs').readdirSync(__dirname));
```

Wywołanie zwróci zawartość natychmiast lub wygeneruje wyjątek w przypadku błędu (zob. rysunek 5.3).

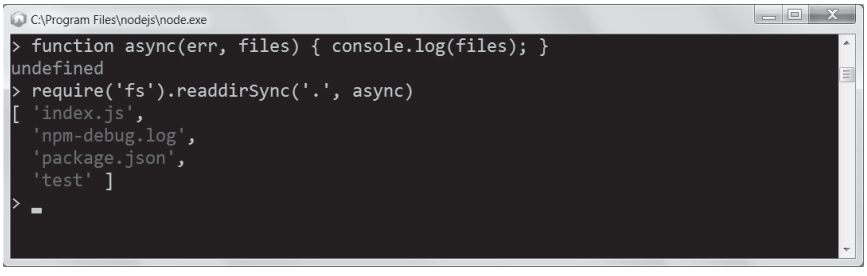


Rysunek 5.3. Sprawdzanie wartości readdirSync

Innym podejściem jest rozwiązanie asynchroniczne:

```
> function async (err, files) { console.log(files); };
> require('fs').readdir('.', async);
```

Da ono identyczne wyniki, pokazane na rysunku 5.4.



Rysunek 5.4. Asynchroniczna wersja readdir

Z rozdziału 3. wiemy, że aby nasze aplikacje były szybkie i radziły sobie z obsługą współbieżności w jednym wątku przy dużym obciążeniu, muszą obsługiwać zdarzenia asynchronicznie.

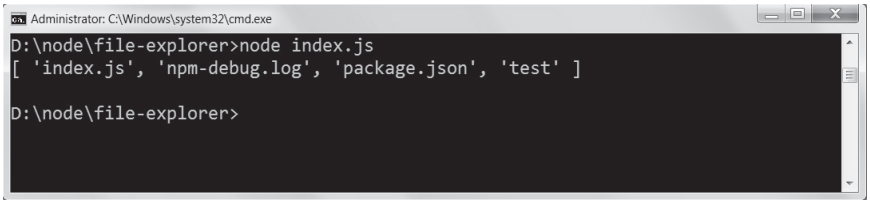
Nasz prosty program wiersza poleceń z pewnością nie będzie funkcjonował w takim środowisku (w danym momencie obsługiwać go będzie tylko jedna osoba), ale aby poznać dobrze jedno z najważniejszych i najtrudniejszych zagadnień związanych z Node.JS, zastosujesz rozwiązanie asynchroniczne.

Do uzyskania listy plików wykorzystamy zatem metodę `fs.readdir`. Przekazywane wywołanie zwrotne dostarcza obiekt błędu (który ma wartość `null` w przypadku braku błędu) i tablicę `files`:

index.js

```
//...
fs.readdir(__dirname, function (err, files) {
  console.log(files);
});
```

Spróbuj wywołać program! Otrzymany rezultat powinien być podobny do tego z rysunku 5.5.



Rysunek 5.5. Twój pierwszy program w akcji

Teraz, kiedy już wiesz, że moduł `fs` zawiera zarówno synchroniczne, jak i asynchroniczne metody dostępu do systemu plików, musisz jeszcze poznać fundamentalne dla Node.JS pojęcie, jakim są strumienie.

ZROZUMIENIE STRUMIENI

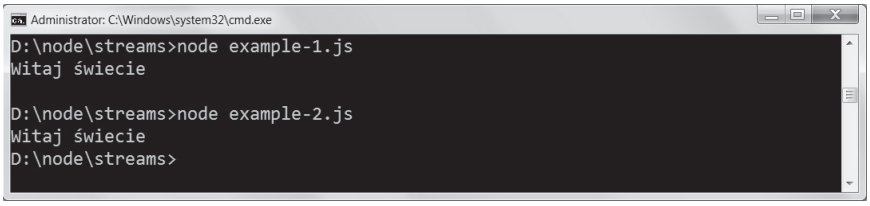
Jak prawdopodobnie zauważyłeś, metoda `console.log` wyświetla dane w konsoli. A uściślając, `console.log` wykonuje konkretne zadanie: *zapisuje* do *strumienia wyjścia* `stdout` podany przez użytkownika łańcuch znaków wraz ze znakiem nowego wiersza `\n`.

Zwróć uwagę na różnicę w wyświetlaniu na rysunku 5.6.

A teraz spójrz na kod źródłowy:

example-1.js

```
console.log('Witaj świecie');
```



Rysunek 5.6. W pierwszym przykładzie po „Witaj świecie" następuje znak nowego wiersza, w drugim już nie

oraz

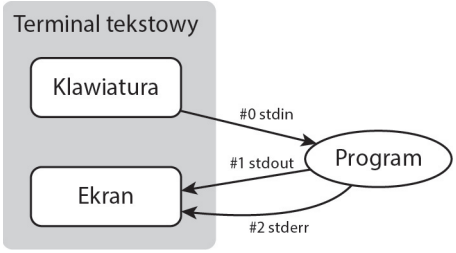
example-2.js

```
process.stdout.write('Witaj świecie');
```

Globalna zmienna procesu zawiera trzy obiekty `Stream`, odpowiadające trzem standardowym strumieniom w systemie Unix:

```
- **stdin**: Standard input
- **stdout**: Standard output
- **stderr**: Standard error
```

Rolę tych obiektów zilustrowano na rysunku 5.7.



Rysunek 5.7. Obiekty stdin, stdout i stderr w kontekście tradycyjnego terminala tekstowego

Pierwszy z nich, `stdin`, jest strumieniem do odczytu, podczas gdy `stdout` i `stderr` są strumieniami do zapisu.

Domyślnym stanem strumienia `stdin` jest stan wstrzymania (`paused`). Z reguły po uruchomieniu program wykonuje pewne zadania, po czym kończy działanie. Czasami jednak, i tak jest również w naszej aplikacji, program oczekuje na dane i przynajmniej dopóki nie zostaną one wprowadzone przez użytkownika, nie może zakończyć działania.

Kiedy wznawiasz ten strumień (za pomocą metody `resume`), Node obserwuje odpowiedni deskryptor pliku (który w systemie Unix otrzymuje numer 0) i przy ciągłym działaniu pętli zdarzeń nie kończy programu, czekając na wywołanie zdarzeń. Node.JS zawsze kończy działanie automatycznie, chyba że oczekuje na dane wejścia-wyjścia.

Inną ciekawą własnością obiektu `Stream` jest to, że posiada on domyślne kodowanie. Jeśli ustawisz kodowanie dla strumienia, zamiast surowego obiektu `Buffer` otrzymasz zakodowany łańcuch tekstowy (za pomocą UTF-8, ASCII itd.) jako parametry zdarzeń.

Obiekt `Stream` jest podstawowym elementem wykorzystywanym przy budowie aplikacji w Node, podobnie jak obiekt `EventEmitter` (po którym zresztą dziedziczy). Podczas pracy z Node często będziesz się spotykać z różnego rodzaju strumieniami, takimi jak gniazda TCP czy żądania HTTP. W skrócie, wszędzie tam, gdzie mamy do czynienia ze stopniowym odczytem lub zapisem danych, obecne są strumienie.

WEJŚCIE I WYJŚCIE

Teraz, kiedy masz już pewne pojęcie o tym, co dzieje się po uruchomieniu programu, możesz przystąpić do tworzenia pierwszej części aplikacji. Wyświetli ona listę plików w bieżącym katalogu i poczeka na dane wprowadzane przez użytkownika:

index.js

```
//...
fs.readdir(process.cwd(), function (err, files) {
  console.log('');

  if (!files.length) {
    return console.log('    \033[31m Brak plików do wyświetlenia!\033[39m\n');
  }

  console.log('   Wybierz plik lub katalog, który chcesz zobaczyć\n');

  function file(i) {
    var filename = files[i];

    fs.stat(__dirname + '/' + filename, function (err, stat) {
      if (stat.isDirectory()) {
        console.log('     '+i+'   \033[36m' + filename + '/\033[39m');
      } else {
        console.log('     '+i+'   \033[90m' + filename + '\033[39m');
      }

      i++;
      if (i == files.length) {
        console.log('');
        process.stdout.write('   \033[33mWprowadź swój wybór: \033[39m');
        process.stdin.resume();
      } else {
        file(i);
      }
    });
  }

  file(0);
});
```

Przeanalizujmy ten kod wiersz po wierszu.

Aby zwiększyć przejrzystość tekstu, wstawiamy pusty wiersz:

```
console.log('')
```

Następnie dodajemy komunikat o braku plików do wyświetlenia, jeśli tablica plików jest pusta. Łańcuchy `\033[31m` i `033[39m`, otaczające tekst, nadają mu czerwony kolor. Na końcu znajduje się znak nowego wiersza `\n`, służący do wizualnego rozdzielenia tekstu.

```
if (!files.length) {
  return console.log('    \033[31m Brak plików do wyświetlenia!\033[39m\n');
}
```

Kolejnego wiersza nie trzeba objaśniać:

```
console.log('   Select which file or directory you want to see\n');
```

Definiujemy funkcję, która będzie wywołana dla każdego elementu tablicy. Jest to pierwszy ze wzorców asynchronicznej kontroli przepływu używanych w tej książce: **przetwarzanie wsadowe** (ang. *serial execution*). Pod koniec rozdziału zajmiemy się nim bardziej szczegółowo.


```
function file (i) {
  //...
}
```

Uzyskujemy dostęp do pierwszej nazwy pliku i pobieramy informacje o pliku w postaci obiektu `Stat`. Obiekt `fs.stat` dostarcza nam różne *metadane* pliku lub katalogu:

```
var filename = files[i];

fs.stat(__dirname + '/' + filename, function (err, stat) {
  //...
});
```

Funkcja zwrotna dostarcza nam obiekt błędu (o ile taki się pojawi) oraz obiekt `Stat`. W tym przypadku interesuje nas metoda `isDirectory` tego ostatniego:

```
if (stat.isDirectory()) {
  console.log('     '+i+'   \033[36m' + filename + '/\033[39m');
} else {
  console.log('     '+i+'   \033[90m' + filename + '\033[39m');
}
```

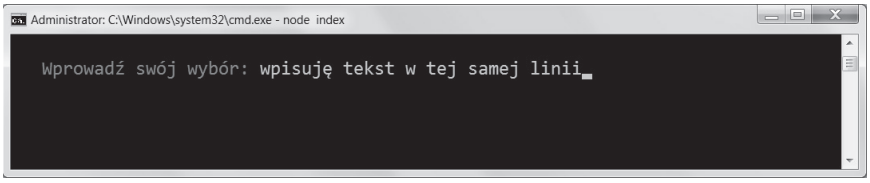
Jeśli ścieżka jest katalogiem, zostanie wyświetlona w innym kolorze niż pliki.

Dalej następuje najważniejsza część kontroli przepływu. Zwiększamy indeks o jeden, bezpośrednio po czym sprawdzamy, czy pozostały jeszcze jakieś pliki do przetworzenia:

```
i++;
if (i == files.length) {
  console.log('');
  process.stdout.write('   \033[33mWprowadź swój wybór: \033[39m');
  process.stdin.resume();
  process.stdin.setEncoding('utf8');
} else {
  file(i);
}
```

Jeżeli nie ma już więcej plików, użytkownik proszony jest o wybór opcji. Zauważ, że posługujemy się tu metodą `process.stdout.write` zamiast `console.log`; nie chcemy przenosić kursora do nowego wiersza, użytkownik wprowadza swój wybór bezpośrednio po komunikacie (zob. rysunek 5.8):

```
console.log('');
process.stdout.write('   \033[33mWprowadź swój wybór: \033[39m');
```



Rysunek 5.8. Aktualna wersja programu prosi o wprowadzenie danych wejściowych

Jak już wiesz, poniższy wiersz pozwala na pobranie danych od użytkownika:

```
process.stdin.resume();
```

W tym wierszu ustawiamy kodowanie strumienia na wartość utf-8, zapewniając obsługę znaków specjalnych i diakrytycznych:

```
process.stdin.setEncoding('utf8');
```

Jeśli są jeszcze pliki do przetworzenia, nasza funkcja zostaje wywołana w sposób rekurencyjny ponownie:

```
file(i);
```

Proces jest kontynuowany, dopóki wszystkie pliki nie zostaną przetworzone, po czym użytkownik proszony jest o wprowadzenie danych. Tym sposobem najważniejsza część aplikacji jest już prawie gotowa.

REFAKTORING

Refaktoring zaczniemy od dodania przydatnych skrótów, jako że `stdin` i `stdout` będą przez nas używane stosunkowo często.

index.js

```
//...
var fs = require('fs')
  , stdin = process.stdin
  , stdout = process.stdout
```

Ponieważ kod jest asynchroniczny, ryzykujemy, że wraz z rozbudową programu (szczególnie jeśli będzie związana z kontrolą przepływu) zbyt głębokie zagnieżdżenie funkcji zmniejszy czytelność kodu.

Aby temu zapobiec, możesz oddzielnie zdefiniować funkcje reprezentujące poszczególne etapy asynchronicznego procesu.

Na początek wyodrębnij funkcję odczytującą `stdin`:

index.js

```
// wywoływana dla każdego pliku w katalogu
function file(i) {
  var filename = files[i];

  fs.stat(__dirname + '/' + filename, function (err, stat) {
    if (stat.isDirectory()) {
      console.log('     '+i+'   \033[36m' + filename + '/\033[39m');
    } else {
      console.log('     '+i+'   \033[90m' + filename + '\033[39m');
    }

    if (++i == files.length) {
      read();
    } else {
      file(i);
    }
  });
}

// odczytaj dane użytkownika po wyświetleniu plików
function read () {
  console.log('');
  stdout.write('   \033[33mWprowadź swój wybór: \033[39m');

  stdin.resume();
  stdin.setEncoding('utf8');
}
```

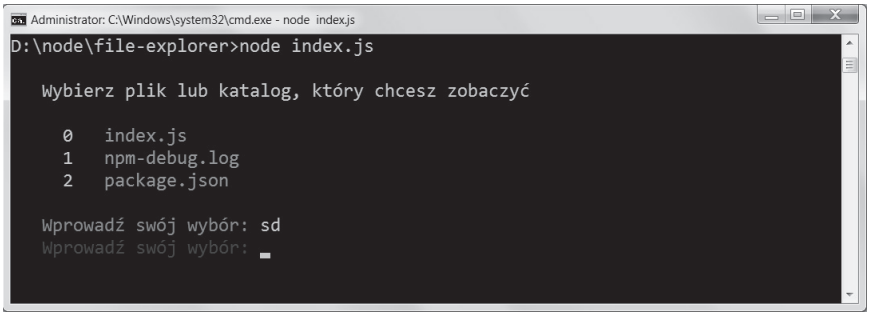
Zwróć uwagę, że wykorzystujesz również nowe zmienne pomocnicze `stdin` i `stdout`.

Po odczytaniu danych następnym logicznym krokiem jest ich przetworzenie. Użytkownik jest proszony o wybranie pliku, który ma zostać odczytany. Po ustawieniu kodowania dla strumienia `stdin`, zaczynamy nasłuchiwać zdarzenia `data`:

```
function read () {
  //...
  stdin.on('data', option);
}

// wywoływana z opcją wybraną przez użytkownika
function option (data) {
  if (!files[Number(data)]) {
    stdout.write('   \033[31mWprowadź swój wybór: \033[39m');
  } else {
    stdin.pause();
  }
}
```

Sprawdzamy tutaj, czy istnieje indeks tablicy `files` odpowiadający wyborowi użytkownika. Pamiętaj, że tablica `files` jest częścią wywołania zwrotnego (`fs.readdir`), w obrębie którego cały czas się znajdujesz. Zwróć też uwagę na konwersję łańcucha utf-8 `data` do typu `Number` przed dokonaniem sprawdzenia.

Jeżeli indeks tablicy istnieje, strumień musi zostać ponownie wstrzymany (wracając do stanu domyślnego), aby — po wykonaniu operacji `fs`, opisanych w kolejnym kroku — program mógł zakończyć działanie (zob. rysunek 5.9).



Rysunek 5.9. Przykład źle wprowadzonego wyboru

Teraz, kiedy nasz program jest już zdolny do interakcji z użytkownikiem, prezentując mu listę plików do wyboru, możemy zająć się ich odczytem i wyświetleniem.

INTERAKCJA Z MODUŁEM FS

Kod odpowiedzialny za odszukiwanie plików jest gotowy, czas zatem na ich odczyt!

```
function option (data) {
  var filename = files[Number(data)];
  if (!filename) {
    stdout.write('   \033[31mWprowadź swój wybór: \033[39m');
  } else {
    stdin.pause();
    fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {
      console.log('');
```

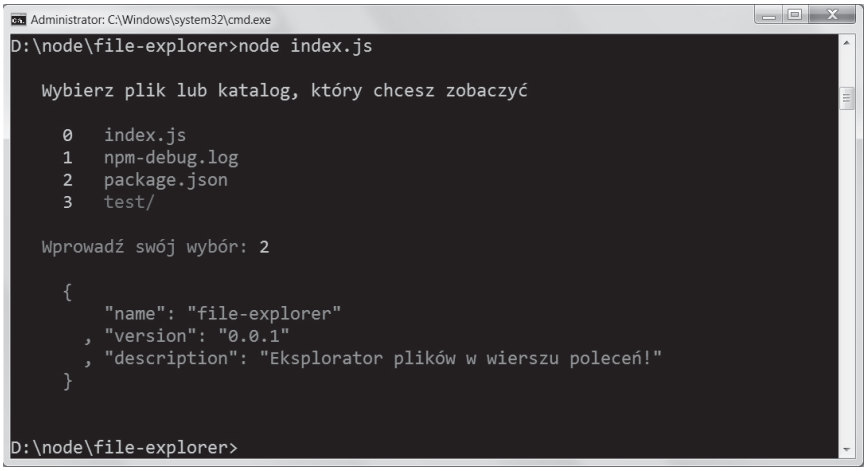
```
      console.log('\033[90m' + data.replace(/(.*)/g, '     $1') + '\033[39m');
    });
  }
}
```

Zauważ, że również tym razem możesz określić kodowanie z góry, otrzymując gotowy do użycia łańcuch tekstowy:

```
fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {
```

Zawartość `data` odczytywana jest za pomocą wyrażenia regularnego (zob. rysunek 5.10):

```
data.replace(/(.*)/g, '     $1')
```



Rysunek 5.10. Przykład odczytu prostego pliku

Co jeśli użytkownik wybrał katalog? W takiej sytuacji muszą zostać wyświetlone podkatalogi i pliki, które zawiera.

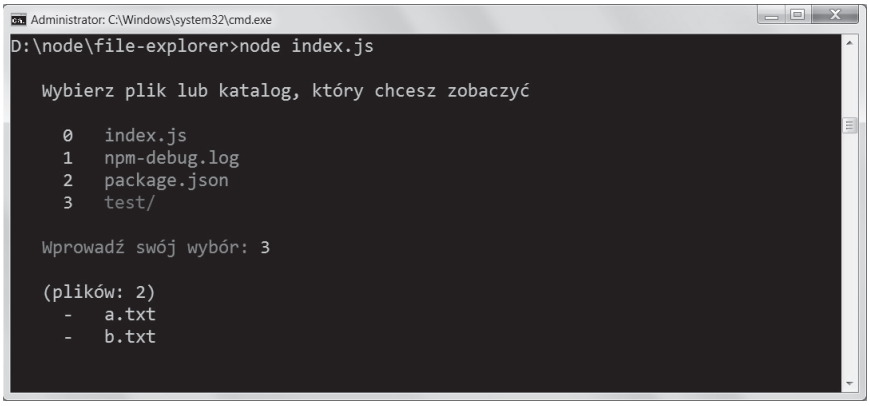
Aby uniknąć wielokrotnego wywoływania `fs.stat`, wróć do funkcji `file` i dodaj instrukcję zapisującą odwołania do obiektów `Stats`:

```
//...
var stats = [];

function file(i) {
  var filename = files[i];

  fs.stat(__dirname + '/' + filename, function (err, stat) {
    stats[i] = stat;
    //...
```

Teraz możesz sprawdzić, czy użytkownik wybrał katalog w funkcji `option`. W miejscu, w którym wcześniej znajdowało się wywołanie `fs.readFile`, wstaw:

```
    if (stats[Number(data)].isDirectory()) {
      fs.readdir(__dirname + '/' + filename, function (err, files) {
        console.log('');
        console.log('   (plików: ' + files.length + ')');
        files.forEach(function (file) {
          console.log('     -   ' + file);
        });
        console.log('');
      });
    } else {
      fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {
        console.log('');
        console.log('\033[90m' + data.replace(/(.*)/g, '     $1') + '\033[39m');
      });
    }
```

Jeśli uruchomisz teraz program, po wybraniu katalogu zobaczysz listę plików, które mogą zostać odczytane, do wyboru (zob. rysunek 5.11).



Rysunek 5.11. Przykład odczytu katalogu /test

I to już wszystko! Właśnie napisałeś swój pierwszy program wiersza poleceń w Node.

WIERSZ POLECEŃ

Masz już za sobą pierwszy program wiersza poleceń, warto zatem poznać kolejne interfejsy programistyczne, pomocne w tworzeniu podobnych aplikacji, uruchamianych w terminalu.

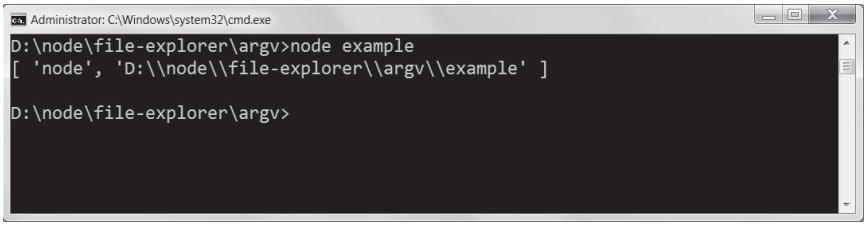
OBIEKT ARGV

Obiekt `process.argv` zawiera wartości wszystkich argumentów, z jakimi program Node został uruchomiony:

example.js

```
console.log(process.argv);
```

Na rysunku 5.12 widzimy, że pierwszym elementem jest zawsze *node*, a drugim ścieżka do uruchamianego pliku. Kolejne elementy są argumentami podanymi w poleceniu.

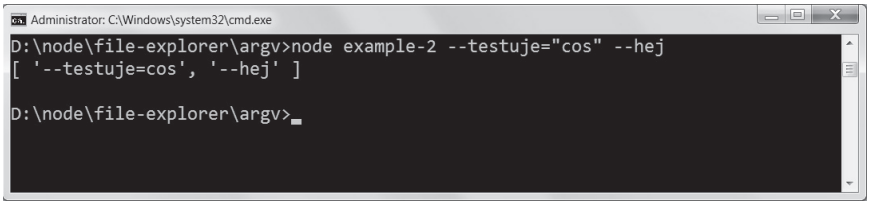


Rysunek 5.12. Przykładowa zawartość process.argv

Aby pominąć pierwsze dwa elementy, użyj metody `slice` (zob. rysunek 5.13):

example-2.js

```
console.log(process.argv.slice(2));
```

Rysunek 5.13. Przykład okrojonej wersji obiektu argv, zawierającej tylko argumenty podane przy uruchomieniu programu

Kolejną bardzo ważną rzeczą przy pracy z Node jest zrozumienie różnicy pomiędzy katalogiem, w którym program *rezyduje*, a katalogiem, w którym jest *uruchamiany*.

KATALOG ROBOCZY

W przykładowej aplikacji z tego rozdziału za pomocą stałej `__dirname` odwołujesz się do katalogu, w którym znajduje się w systemie plików uruchamiany plik.

Czasami jednak w trakcie pracy aplikacji bardziej korzystne jest pobranie nazwy *bieżącego katalogu roboczego* (ang. *current working directory*). Zgodnie z aktualną implementacją, niezależnie od tego, czy znajdujesz się w katalogu macierzystym, czy w dowolnym innym katalogu, uruchomienie aplikacji da taki sam wynik. Położenie pliku *index.js* się nie zmienia, a więc wartość `__dirname` też pozostaje taka sama.

Aby uzyskać bieżący katalog roboczy, wywołaj metodę `process.cwd`:

```
> process.cwd()
/Users/guillermo
```

Node umożliwia również jego zmianę, dzięki metodzie `process.chdir`:

```
> process.cwd()
/Users/guillermo
> process.chdir('/')
> process.cwd()
/
```

Kolejny aspekt kontekstu, w którym uruchamiany jest program, to obecność zmiennych środowiskowych. W następnym punkcie pokażemy, jak uzyskać dostęp do tych zmiennych.

ZMIENNE ŚRODOWISKOWE

Node pozwala na łatwy dostęp do zmiennych, które są częścią środowiska powłoki, poprzez wygodny obiekt `process.env`.

Przykładem popularnej zmiennej środowiskowej jest `NODE_ENV` (zob. rysunek 5.14), której najczęstszym zastosowaniem jest informowanie programu Node, czy działa w środowisku produkcyjnym, czy deweloperskim.

Rysunek 5.14. Zmienna środowiskowa NODE_ENV

W trakcie działania programu często potrzebna jest bezpośrednia kontrola nad jego zakończeniem.

ZAKAŃCZANIE PROGRAMU

Aby zakończyć aplikację, możesz użyć metody `process.exit` z opcjonalnym kodem zakończenia. Jeśli na przykład chcemy, aby program zakończył się błędem, najlepiej użyć kodu 1.

```
console.error('Wystąpił błąd');
process.exit(1);
```

Pozwala to na sprawną współpracę pomiędzy programami wiersza poleceń i innymi narzędziami w systemie operacyjnym.

Innym ważnym aspektem tej współpracy są sygnały procesu.

SYGNAŁY

Proces komunikuje się z systemem operacyjnym na różne sposoby. Jednym z nich są *sygnały* (ang. *signals*). Kiedy chcemy na przykład natychmiastowo zakończyć proces, wystarczy mu wysłać sygnał `SIGKILL`.

Sygnały są w Node emitowane jako zdarzenia obiektu `process`:

```
process.on('SIGKILL', function () {
  // signal received
});
```

W następnym punkcie wyjaśnimy, jak uzyskaliśmy w naszej przykładowej aplikacji kolorowy tekst.

SEKWENCJE STERUJĄCE ANSI

Chcąc kontrolować kolory i inne parametry strumienia wyjściowego w terminalu tekstowym, korzystamy z *sekwencji sterujących ANSI* (ang. *ANSI escape sequences*), zwanych również *kodami ANSI*. Te znaki specjalne są rozpoznawane przez emulator terminala w standardowy sposób.

Kiedy umieszczasz między tymi znakami tekst, nie pojawią się one oczywiście na ekranie. Są to tak zwane *znaki niedrukowalne* (ang. *nonprinting characters*).

Weźmy na przykład następujące sekwencje:

```
console.log('\033[90m' + data.replace(/(.*)/g, '     $1') + '\033[39m');
```

- `\033` rozpoczyna sekwencję sterującą;
- `[` informuje o zmianie koloru;
- `90` zmienia kolor tekstu na jasnoszary;
- `m` kończy sekwencję.

Zwróć uwagę, że w drugiej sekwencji używamy wartości `39`, która powoduje powrót dalszego tekstu do domyślnego dla terminala koloru.

Kompletną tabelę kodów ANSI znajdziesz pod adresem *http://en.wikipedia.org/wiki/ANSI_escape_code*.

MODUŁ FS

Moduł `fs` umożliwia odczyt i zapis danych poprzez interfejs programistyczny `Stream`. W przeciwieństwie do metod `readFile` i `writeFile`, przydział pamięci odbywa się w jego przypadku stopniowo.

Wyobraź sobie plik z dużą ilością danych oddzielonych przecinkami i milionami wierszy. Jednorazowy jego odczyt w celu przetworzenia wiązałby się z koniecznością przydzielenia dużego obszaru pamięci. Dużo lepszym rozwiązaniem byłby odczyt pliku partiami wyznaczanymi przez znaki końca wiersza („`\n`") i ich przetwarzanie na bieżąco.

Strumienie Node nadają się do tego idealnie, o czym przekonasz się już zaraz.

STRUMIENIE

Metoda `fs.createReadStream` pozwala utworzyć strumień do odczytu (ang. *readable*) dla danego pliku.

Potencjał strumieni najlepiej ilustruje różnica pomiędzy dwoma zamieszczonymi niżej przykładami:

```
fs.readFile('my-file.txt', function (err, contents){
  // zrób coś z plikiem
});
```

W tym przypadku wywołanie przekazywanej funkcji zwrotnej następuje dopiero, kiedy cała zawartość pliku będzie wczytana, umieszczona w pamięci operacyjnej i gotowa do użycia.

W poniższym przykładzie natomiast plik odczytywany jest partiami o zmiennym rozmiarze. Funkcja zwrotna wywoływana jest przy odczycie każdej partii:

```
var stream = fs.createReadStream('my-file.txt');
stream.on('data', function(chunk){
  // zrób coś z częścią pliku
});
stream.on('end', function(chunk){
  // osiągnięto koniec pliku
});
```

Dlaczego ta zdolność strumieni jest taka ważna? Wyobraź sobie, że musisz przesłać do usługi sieciowej bardzo duży plik wideo. Wczytanie całego pliku nie jest konieczne do rozpoczęcia przesyłania, tak więc użycie strumienia przekłada się bezpośrednio na szybkość całej operacji.

To samo dotyczy zapisu w pliku dziennika, zwłaszcza jeśli korzystamy ze strumienia do zapisu (ang. *writable*). Jeżeli używasz aplikacji sieciowej do zapisywania działań użytkowników odwiedzających Twoją stronę w pliku dziennika, zmuszanie systemu operacyjnego do każdorazowego otwarcia i zamknięcia pliku (a co za tym idzie, odszukania go na dysku) nie będzie rozwiązaniem efektywnym z racji dużej liczby zapisywanych zdarzeń.

W takim przypadku dużo lepiej użyć obiektu `fs.WriteStream`, otwierając plik raz, a następnie wywołując metodę `.write` przy każdym nowym wpisie.

Kolejnym ważnym elementem modelu pracy Node, polegającego na nieblokowaniu operacji wejścia-wyjścia, jest obserwacja.

OBSERWACJA

Node umożliwia *obserwowanie* plików i katalogów pod kątem zmian. Obserwując dany plik lub katalog, jesteśmy informowani (przez zdarzenie w postaci wywołania zwrotnego) o każdej modyfikacji pliku (lub plików zawartych w katalogu).

Mechanizm ten jest często wykorzystywany w środowisku Node. Niektórzy wolą na przykład przygotowywać arkusze stylów CSS w sposób pośredni. Wprowadzają oni kod w języku programowania, który jest następnie kompilowany do postaci CSS. Automatyczna kompilacja po każdej modyfikacji pliku jest bardzo wygodna.

Rozważmy następujący przykład. Na początek szukamy wszystkich plików CSS w katalogu roboczym, a następnie obserwujemy je pod kątem zmian. Po wykryciu zmiany plik jest wyświetlany w konsoli:

```
var fs = require('fs');
var stream = fs.createReadStream('my-file.txt');
// pobierz wszystkie pliki z katalogu roboczego
var files = fs.readdirSync(process.cwd());
files.forEach(function (file) {
  // obserwuj plik, jeśli kończy się ".css"
  if (/\.css/.test(file)) {
    fs.watchFile(process.cwd() + '/' + file, function () {
      console.log(' – ' + file + ' zmieniony!');
    });
  }
});
```


Oprócz metody `fs.watchFile` możesz również skorzystać z metody `fs.watch`, która pozwala na obserwację całych katalogów.

PODSUMOWANIE

W tym rozdziale poznałeś podstawy tworzenia aplikacji w Node.JS, a dokładniej programu wiersza poleceń, który komunikował się z systemem plików.

Chociaż ten konkretny program mógł zostać napisany przy użyciu synchronicznych interfejsów modułu `fs`, skorzystaliśmy z interfejsów asynchronicznych, aby lepiej zrozumieć pewne niuanse tworzenia kodu z dużą liczbą wywołań zwrotnych. Niezależnie od tego udało nam się uzyskać opisowy i w pełni funkcjonalny kod.

Omówiony w tym rozdziale jeden z najważniejszych interfejsów programistycznych, `Stream`, będzie się często przewijał w dalszej części książki. Prawie wszędzie tam, gdzie mamy do czynienia z operacjami wejścia-wyjścia, użycie strumieni jest nieuniknione.

Otrzymałeś też dużo wskazówek i narzędzi, dzięki którym jesteś w stanie pisać złożone i przydatne programy, wykorzystujące system plików, komunikujące się z innymi aplikacjami i pobierające dane od użytkownika.

Jako programista Node.JS, będziesz tę wiedzę (a szczególnie jej część odnoszącą się do procesu) wykorzystywać bardzo często, zarówno podczas tworzenia aplikacji sieciowych, jak i podczas rozwiązywania bardziej złożonych problemów. Postaraj się ją zatem dobrze przyswoić!

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Ż

TWÓJ KLUCZ DO TWORZENIA WYDAJNYCH APLIKACJI SIECIOWYCH!

Platforma Node.js powstała w 2009 roku. Pozwala na tworzenie wydajnych, skalowalnych aplikacji sieciowych. W tym środowisku napiszesz kod działający po stronie serwera — i użyjesz do tego języka JavaScript. Brzmi niesamowicie? I tak w rzeczywistości jest! Przekonasz się o tym w czasie czytania tej książki. Została ona w całości poświęcona Node.js.

Dzięki temu znakomitemu podręcznikowi błyskawicznie skonfigurujesz i uruchomisz swoją pierwszą aplikację. Potem będzie już tylko ciekawiej — sterowanie zdarzeniami, wykorzystywanie wydajnych operacji wejścia-wyjścia oraz połączenie z bazą MongoDB to tylko niektóre z poruszanych tematów. Dowiesz się także, jak skorzystać z dostępu do danych w czasie rzeczywistym za pomocą technologii WebSocket oraz jak testować napisany kod z użyciem testów automatycznych. Po lekturze tej książki bez trudu przygotujesz aplikację, która będzie w stanie obsłużyć dziesiątki tysięcy jednoczesnych połączeń na jednym serwerze. Zainteresowany? Sięgnij po książkę — naprawdę warto!

Sprawdź:

- jak stworzyć wydajne aplikacje sieciowe
- jak uzyskać dostęp do danych w czasie rzeczywistym
- jak łatwo zintegrować Node.js z MongoDB

helion.pl
księgarnia internetowa

Nr katalogowy: 16479

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

WILEY

ISBN 978-83-246-6674-4

Cena 59,00 zł

9 788324 666744