



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

© 2012 Guillermo Rauch

All Rights Reserved. Authorised translation from the English language edition published by John Wiley & Sons Limited. Responsibility for the accuracy of the translation rests solely with Helion S.A. and is not the responsibility of John Wiley & Sons Limited.

No part of this book may be reproduced in any form without the written permission of the original copyright holder, John Wiley & Sons Limited.

Translation copyright © 2014 by Helion S.A.

Wiley and the John Wiley & Sons, Ltd. logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and/or other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Ltd. is not associated with any product or vendor mentioned in the book.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Część I	Szybki start: instalacja i pojęcia ogólne	19
Rozdział 1	Przygotowanie środowiska	21
	Instalacja w systemie Windows	22
	Instalacja w systemie OS X	22
	Instalacja w systemie Linux	23
	Kompilacja	23
	Kontrola działania	23
	Narzędzie REPL Node	23
	Wykonanie skryptu	24
	NPM	25
	Instalowanie modułów	25
	Definiowanie własnego modułu	26
	Instalacja narzędzi binarnych	27
	Przeszukiwanie rejestru NPM	28
	Podsumowanie	29
Rozdział 2	Przegląd JavaScript	31
	Podstawowy JavaScript	32
	Typy	32
	Typowa łamigłówka	32
	Funkcje	33
	Konstrukcje this, call() i apply()	34
	Arność funkcji	34
	Domknięcia	35
	Klasy	35
	Dziedziczenie	36
	Blok try {} catch {}	37
	JavaScript w wersji v8	38
	Metoda keys() obiektu	38
	Metoda isArray() tablicy	39
	Metody tablic	39
	Metody łańcuchów znaków	39
	JSON	39
	Metoda bind() funkcji	40
	Właściwość name funkcji	40
	Właściwość __proto__ i dziedziczenie	40
	Metody dostępne	41
	Podsumowanie	42

Rozdział 3	Blokujące i nieblokujące operacje wejścia-wyjścia	43
	Duże możliwości to duża odpowiedzialność	44
	Blokowanie	46
	Jednowątkowy świat	47
	Obsługa błędów	50
	Ślady stosów wywołań	51
	Podsumowanie	53
Rozdział 4	JavaScript dla Node	55
	Obiekt globalny	56
	Pożyteczne zmienne globalne	56
	System modułów	57
	Moduły względne i bezwzględne	57
	Udostępnianie interfejsu programistycznego	59
	Zdarzenia	61
	Bufory	63
	Podsumowanie	64
Część II	Najistotniejsze interfejsy programistyczne Node	65
Rozdział 5	Wiersz poleceń i moduł FS: Twoja pierwsza aplikacja	67
	Wymagania	68
	Piszemy nasz pierwszy program	68
	Tworzymy moduł	69
	sync czy async?	70
	Zrozumienie strumieni	71
	Wejście i wyjście	73
	Refaktoring	75
	Interakcja z modulem fs	77
	Wiersz poleceń	79
	Obiekt argv	79
	Katalog roboczy	80
	Zmienne środowiskowe	81
	Zakończanie programu	81
	Sygnały	82
	Sekwencje sterujące ANSI	82
	Moduł fs	82
	Strumienie	83
	Obserwacja	84
	Podsumowanie	84
Rozdział 6	Protokół TCP	87
	Czym charakteryzuje się TCP?	88
	Komunikacja z naciskiem na połączenia i zasada zachowania kolejności	88
	Kod bajtowy jako podstawowa reprezentacja	88
	Niezawodność	89
	Kontrola przepływu	89
	Kontrola przeciążeń	89

Telnet	89
Czat na bazie TCP	92
Tworzymy moduł	92
Klasa net.Server	92
Odbieranie połączeń	94
Zdarzenie data	96
Stan i monitorowanie połączeń	97
Wykończenie	100
Klient IRC	102
Tworzymy moduł	102
Interfejs net.Stream	103
Implementacja części protokołu IRC	103
Test z prawdziwym serwerem IRC	104
Podsumowanie	104
Rozdział 7	Protokół HTTP
	105
Struktura HTTP	106
Nagłówki	107
Połączenia	111
Prosty serwer WWW	112
Tworzymy moduł	112
Wyświetlamy formularz	112
Metody i adresy URL	114
Dane	117
Składamy elementy w całość	119
Dopracowanie szczegółów	120
Klient Twittera	121
Tworzymy moduł	121
Wysyłanie prostego żądania HTTP	122
Wysyłanie danych	123
Pobieranie tweetów	124
Moduł superagent na pomoc	128
Przeładowanie serwera za pomocą narzędzia up	130
Podsumowanie	130
Część III	Tworzenie aplikacji sieciowych
	133
Rozdział 8	Framework Connect
	135
Prosta strona internetowa przy użyciu modułu http	136
Prosta strona internetowa przy użyciu frameworka Connect	139
Metody pośredniczące	141
Tworzenie metod pośredniczących wielokrotnego użytku	142
Metoda pośrednicząca static	146
Metoda pośrednicząca query	148
Metoda pośrednicząca logger	148
Metoda pośrednicząca bodyParser	150
Ciasteczka	153
Metoda pośrednicząca session	154

	Sesje Redis	159
	Metoda pośrednicząca methodOverride	160
	Metoda pośrednicząca basicAuth	160
	Podsumowanie	162
Rozdział 9	Framework Express	163
	Prosta aplikacja Express	164
	Tworzymy moduł	164
	HTML	164
	Konfiguracja	165
	Definiowanie tras	166
	Moduł search	168
	Uruchomienie aplikacji	169
	Ustawienia	170
	Mechanizmy szablonów	172
	Obsługa błędów	173
	Metody złożone	173
	Trasy	175
	Metody pośredniczące	177
	Strategie organizacji	178
	Podsumowanie	180
Rozdział 10	Technologia WebSocket	181
	AJAX	182
	Technologia WebSocket	184
	Aplikacja Echo	185
	Przygotowanie	185
	Konfiguracja serwera	186
	Konfiguracja klienta	187
	Uruchomienie serwera	188
	Kursory myszy	189
	Przygotowanie	189
	Konfiguracja serwera	189
	Konfiguracja klienta	192
	Uruchomienie serwera	194
	Kwestie do rozwiązania	194
	Zamknięcie połączenia a rozłączenie	195
	JSON	195
	Ponowne łączenie	195
	Rozgłaszanie	195
	WebSocket to HTML5: starsze przeglądarki go nie obsługują	195
	Rozwiązanie	195
	Podsumowanie	196
Rozdział 11	Framework Socket.IO	197
	Transporty	198
	Rozłączenie kontra zamknięcie połączenia	198
	Zdarzenia	198
	Przestrzenie nazw	199

Czat	200
Przygotowanie programu	200
Konfiguracja serwera	200
Konfiguracja klienta	201
Zdarzenia i rozgłaszanie	203
Gwarancja odbioru	207
Aplikacja DJ	209
Rozszerzenie czata	209
Integracja z interfejsem Grooveshark	210
Odtwarzanie	213
Podsumowanie	218
Część IV	219
Bazy danych	219
Rozdział 12 MongoDB	221
Instalacja	223
Dostęp do MongoDB: przykład uwierzytelnienia użytkownika	224
Konfiguracja aplikacji	224
Tworzymy aplikację Express	224
Łączymy się z MongoDB	228
Tworzymy dokumenty	230
Wyszukiwanie dokumentów	232
Metoda pośrednicząca do uwierzytelniania	233
Sprawdzanie poprawności danych	234
Niepodzielność	235
Tryb bezpieczny	235
Wprowadzenie do Mongoose	236
Definiowanie modelu	236
Definiowanie zagnieżdżonych kluczy	238
Definiowanie zagnieżdżonych dokumentów	238
Ustawianie indeksów	239
Metody pośredniczące	239
Sprawdzanie stanu modelu	239
Zapytania	240
Rozszerzanie zapytań	240
Sortowanie	240
Wybieranie danych	240
Limitowanie wyników	241
Pomijanie wyników	241
Automatyczne wypełnianie kluczy	241
Konwersja typów	242
Przykład Mongoose	242
Konfiguracja aplikacji	242
Refaktoryzacja	243
Definiowanie modeli	243
Podsumowanie	245

Rozdział 13	MySQL	247
	node-mysql	248
	Konfiguracja	248
	Aplikacja Express	248
	Łączenie z MySQL	249
	Inicjalizacja skryptu	250
	Wstawianie danych	253
	Pobieranie danych	258
	Narzędzie Sequelize	259
	Konfiguracja Sequelize	260
	Konfiguracja aplikacji Express	260
	Konfiguracja Sequelize	263
	Definiowanie modeli i synchronizacja	264
	Wstawianie danych	266
	Pobieranie danych	268
	Usuwanie danych	269
	Wykończenie	271
	Podsumowanie	272
Rozdział 14	Redis	273
	Instalacja Redis	275
	Język zapytań Redis	275
	Typy danych	276
	Ciągi znaków	277
	Tablice asocjacyjne	277
	Listy	279
	Zbiory	279
	Zbiory sortowane	280
	Redis i Node	280
	Implementacja mapy relacji przy użyciu Node i Redis	281
	Podsumowanie	290
Część V	Testowanie	291
Rozdział 15	Współdzielony kod	293
	Co może być współdzielone?	294
	Kompatybilność kodu JavaScript	294
	Udostępnianie modułów	295
	Adaptacja interfejsów programistycznych ECMA	296
	Adaptacja interfejsów programistycznych Node	297
	Adaptacja interfejsów programistycznych przeglądarek	298
	Dziedziczenie dla wszystkich przeglądarek	298
	Zastosowanie praktyczne: narzędzie browserbuild	299
	Prosty przykład	300
	Podsumowanie	302

Rozdział 16 Testowanie	305
Proste testy	306
Przedmiot testów	306
Strategia testów	306
Program testowy	307
Expect.js	308
Przegląd interfejsów programistycznych	308
Mocha	310
Testowanie asynchronicznego kodu	311
Styl BDD	313
Styl TDD	314
Styl eksportu	314
Korzystanie z Mocha w przeglądarce	315
Podsumowanie	316
Skorowidz	317



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

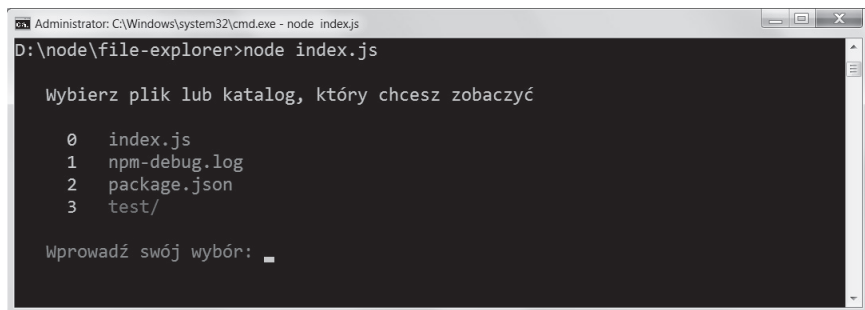
i zdarzeń. Dzięki tym interfejsom poznasz kontrolę przepływu w procesie programowania z wykorzystaniem zdarzeń i nieblokujących operacji wejścia-wyjścia.

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 modulem `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.

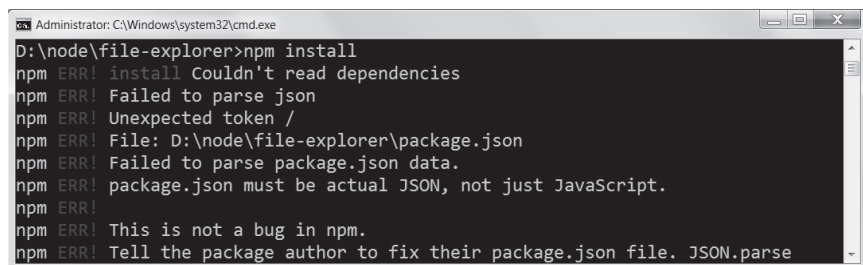
Chociaż w naszym przykładzie będziemy korzystać tylko z wbudowanych modułów Node (a więc niepobieranych z rejestru NPM), musimy przygotować prosty plik *package.json*:

```
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 błędy¹. 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.
 */

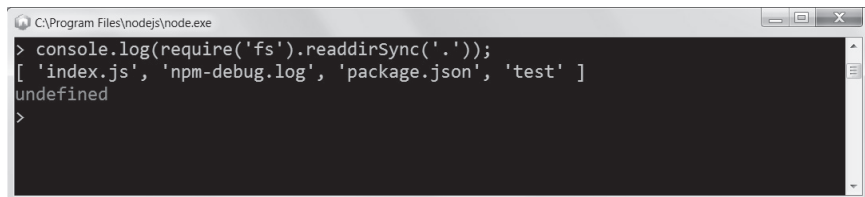
var fs = require('fs');
```

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).

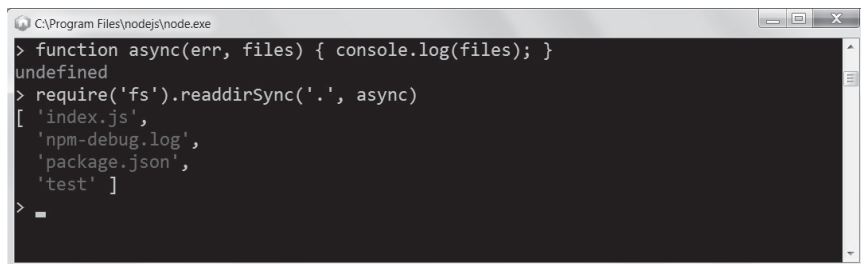


Rysunek 5.3. Sprawdzanie wartości `readdirSync`

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`

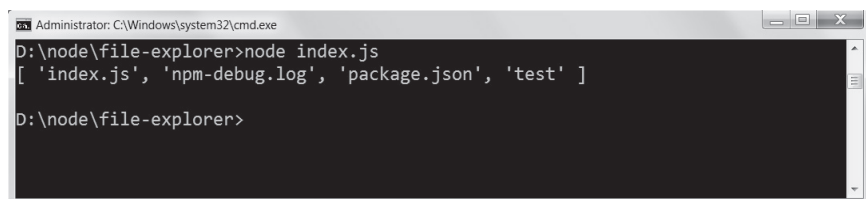
Z rozdziału 3. wiemy, że aby nasze aplikacje były szybkie i radziły sobie z obsługą współbieżności w jednym wątku przy dużym obciążeniu, muszą obsługiwać zdarzenia asynchronicznie.

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.

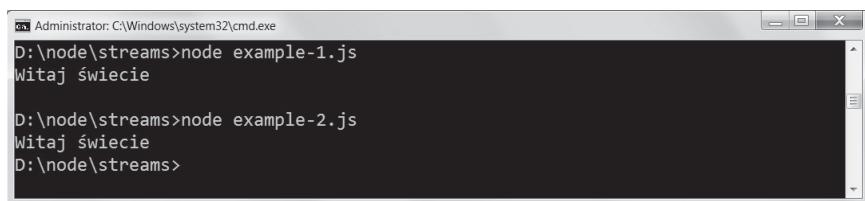
ZROZUMIENIE STRUMIENI

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

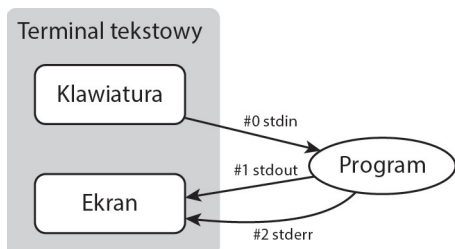
```

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 wznowiasz 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('    '+ '  \033[36m' + filename + '/\033[39m');
      } else {
        console.log('    '+ '  \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);
});
```

Przeanalizujemy 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 rozdzielania 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);
}

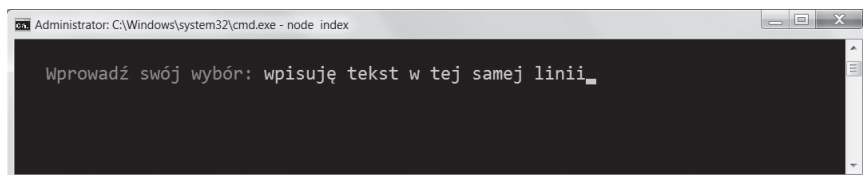
```

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.

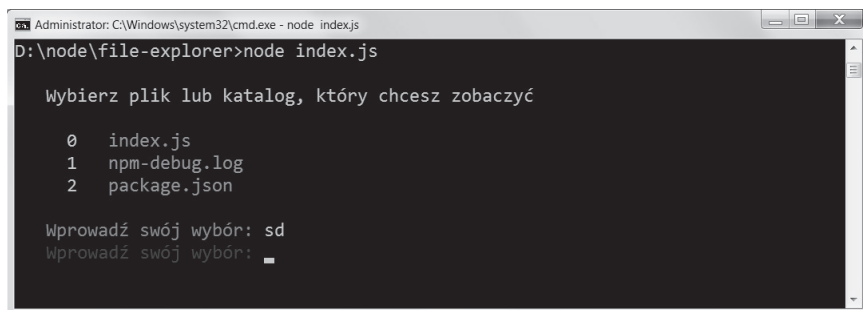
Po odczytaniu danych następnym logicznym krokiem jest ich przetworzenie. Użytkownik jest proszony o wybranie pliku, który ma zostać odczytany. Po ustawieniu kodowania dla strumienia stdin, zaczynamy nasłuchiwać zdarzenia data:

```
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 zwrótnego (`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');
    });
}
}

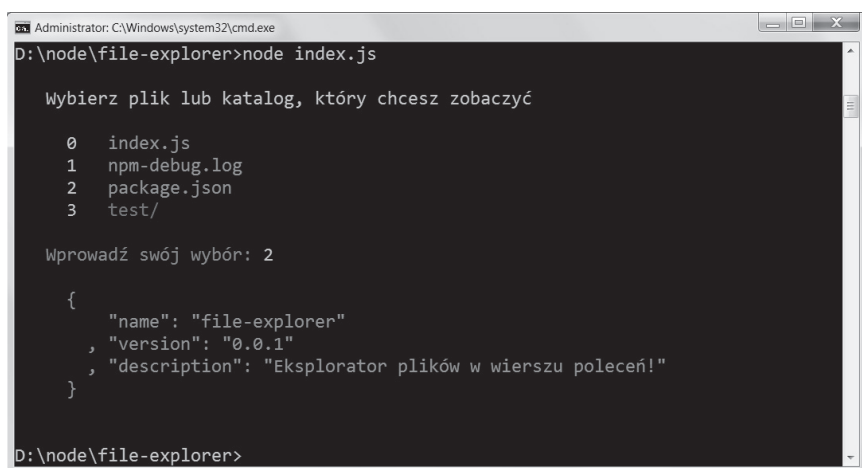
```

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;
        //...
    });
}

```

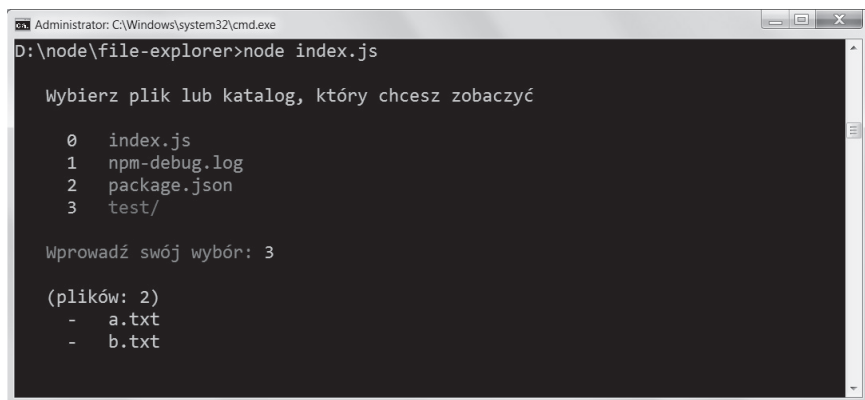
Teraz możesz sprawdzić, czy użytkownik wybrał katalog w funkcji `option`. W miejscu, w którym wcześniej znajdowało się wywołanie `fs.readFile`, wstaw:

```

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).



```

Administrator: C:\Windows\system32\cmd.exe
D:\node\file-explorer>node index.js

Wybierz plik lub katalog, który chcesz zobaczyć

0  index.js
1  npm-debug.log
2  package.json
3  test/

Wprowadź swój wybór: 3

(plików: 2)
-  a.txt
-  b.txt

```

Rysunek 5.11. Przykład odczytu katalogu /test

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

WIERSZ POLECEŃ

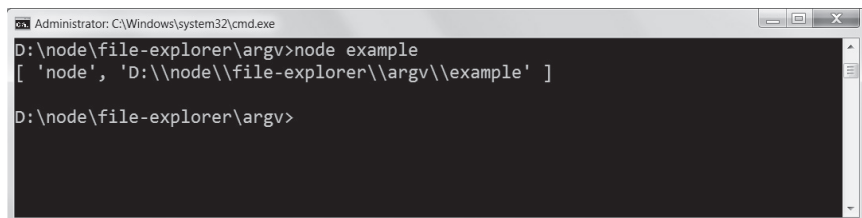
Masz już za sobą pierwszy program wiersza poleceń, warto zatem poznać kolejne interfejsy programistyczne, pomocne w tworzeniu podobnych aplikacji, uruchamianych w terminalu.

OBIEKT ARGV

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

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

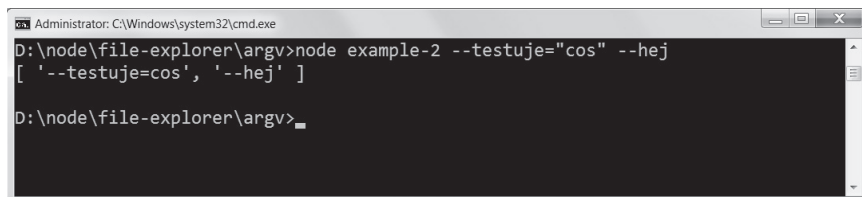
Na rysunku 5.12 widzimy, że pierwszym elementem jest zawsze *node*, a drugim ścieżka do uruchamianego pliku. Kolejne elementy są argumentami podanymi w poleceniu.



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`:

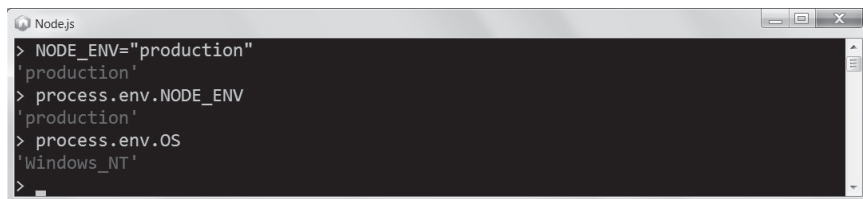
```
> process.cwd()
/Users/guillermo
> process.chdir('/')
> process.cwd()
/
```

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

A

acknowledgment, *Patrz:* potwierdzenie
adaptacja, 294
adres
 permalink, 176
 URL, 44, 113, 114, 115, 116, 120, 126, 137, 139, 147
 kodowanie, 118
 łańcuch zapytania, *Patrz:* łańcuch zapytania
 ukośnik, 171
AJAX, 182, 198, 260, 262
ANSI escape sequences, *Patrz:* sekwencja sterująca ANSI
arkusz stylów CSS, *Patrz:* CSS
assertion, *Patrz:* test sprawdzający
atomicity, *Patrz:* niepodzielność operacji
autoryzacja, 141, 161
AWS, *Patrz:* chmura Amazon

B

baza danych, *Patrz też:* sprawdzanie poprawności
 bezschematowa, 221
 MySQL, 258
 oparta na dokumentach, 221
 pojedyncza, 236
 Redis, *Patrz:* Redis
 SQL, 250
BDD, 313
behavior-driven development, *Patrz:* BDD
biblioteka
 jQuery, *Patrz:* jQuery
 kryptograficzna, 294
 matematyczna, 294
 OpenSSL, 23
blog, 235, 238
błąd, 50, 81, 168, 173, 231, 252, 306
 funkcji odłożonej w czasie, 52
 połączenia, 95
browserbuild, 293, 299
bufor, 63

C

call stack, *Patrz:* stos wywołań
callback, *Patrz:* wywołanie zwrotne
chmura, 49
Chrome, 38, 296
ciasteczka, 153
ciąg znaków, 277
closure, *Patrz:* domknięcie
Connect, 135, 139, 177
connection, *Patrz:* połączenie
consumer key, 125
consumer secret, 125
controller, *Patrz:* kontroler
CSS, 84
current working directory, *Patrz:* katalog roboczy bieżący
czat, 92, 200, 207
 rozszerzenia, 209
 użytkownik, 97, 98, 99, 100

D

Dahl Ryan, 13, 15
dane
 baza, *Patrz:* baza danych
 pakiet, 88
 sesji, 159
 sprawdzanie poprawności, 235
 struktura, 119
 strumień, *Patrz:* strumień danych
 typ, *Patrz:* typ danych
 wysyłanie, 110, 123
data stream, *Patrz:* strumień danych
datagram, 88
dependencies, 27
dokument, 222
 głębokość, 222
 ograniczanie liczby, 241
 pomijanie, 241
 szukanie, 228
 tworzenie, 230

dokument

- wstawianie, 228
- wyszukiwanie, 232
- zagnieżdżony, 238

domknięcie, 31, 35

Don't Repeat Yourself, *Patrz:* DRY

DRY, 139

dziedziczenie, 40, 87, 298

- klasyczne, 36

- łańcuch, 36

- prototypowe, 36

dziennik, 83, 141, 142, 149

E

ECMAScript, 38

EJS, 164

Embedded JavaScript, *Patrz:* EJS

event, *Patrz:* zdarzenie

EventEmitter, 297

execution stack, *Patrz:* stos wywołań

Express, 17, 163, 170, 177, 248

F

file descriptor, *Patrz:* plik deskryptor

Firebug, 17

firewall, 195, 198

flaga, 171

format

- base64, 63

- default, 149

- dev, 149

- JSON, 117, 126, 198

- PNG, 63

- short, 149

- tiny, 149

- tras, 166

- XML, 117

formularz, 112, 117, 119, 227, 266

framework

- Connect, *Patrz:* Connect

- Express, *Patrz:* Express

- JavaScript, 182

- Mocha, *Patrz:* Mocha

- programowania obiektowego, 294

- Socket.IO, *Patrz:* socket.io

framing, *Patrz:* ramkowanie

funkcja, 33, 34

- anonimowa, 35

- argument, 34

- arność, 34

- domknięcie, *Patrz:* domknięcie

- JavaScript, 140

obsługi trasy, 166

rekurencyjna, 75

samowywołująca, 35

this, 34

try/catch, 37

zwracająca funkcję, 142

zwrotna, 93, 111

G

GitHub, 17

gniazdo, 47, 182, 200

- TCP, 73

grafika, 298

Groovespark, 210

H

handshake, *Patrz:* wymiana potwierdzeń

header, *Patrz:* nagłówek

heartbeats, *Patrz:* taktowanie

HTML5, 181

HTTP, 87, 105, 111, 116, 121, 124, 183, 260

- nagłówek, *Patrz:* nagłówek

HTTPS, *Patrz:* żądanie HTTP

I

in-memory store, *Patrz:* magazyn pamięciowy

interfejs

- 2d Canvas, 294

- connect, *Patrz:* connect

- DOM, 294

- Node EventEmitter, *Patrz:* Node EventEmitter

- programistyczny, 56, 57, 58, 59, 62, 67, 294

- adaptacja, 296

- ECMA, 296

- EventEmitter, 61, 62

- fs, 70

- ponad interfejsem klienta HTTP, 128

- Stream, 82

- TinySong, 210

- Twittera, 124, 128

- udostępnianie, 59

- WebSocket, 184

- wyższego rzędu, 111

ReadStream, 110

stdio, 70

WebSocket, 294

XMLHttpRequest, 294

Internet Protocol, *Patrz:* protokół IP

Internet Relay Chat, *Patrz:* IRC

IRC, 102, 103

J

Jade, 172, 225, 248
 instalowanie, 313
 interpolacja, 256
 JavaScript
 kompatybilność, 294
 konstruktor, 236
 wersja
 podstawowa, 31, 32
 v8, 31, 38
 język
 SQL, 247
 szablonów, 164, 172
 zapytań Redis, 275
 jQuery, 182, 262, 266
 JSON, 27, 117, 126, 198, 250, 266
 deserializacja, 39, 169, 195
 serializacja, 39, 195, 199

K

katalog roboczy, 80, 84
 bieżący, 80
 klasa, 35
 EventEmitter, 96
 http.Server, 87
 net.Server, 87
 pochodna, 36
 Schema, 236
 klient
 HTTP, 108, 121, 128
 IRC, 102, 103, 104
 SSH, 87
 TCP, 102
 telnet, 90
 Twittera, 121
 klucz, 232, 240, 274
 dostępowy, 125
 indeks, 239
 kliencki, 125
 Redis, 274, 277
 type, 237
 wypełnianie automatyczne, 241
 zagnieżdżony, 238
 kod
 ANSI, *Patrz*: sekwencja sterująca ANSI
 asynchroniczny, 311
 automatyczny, 96
 bajtowy, 88, 96
 HTML, 108
 JSON, 27, 172, 250
 utf-8, 96, 103, 122
 współdzielony, 294

kolekcja, 221
 kompilator C/C++, 23
 konstruktor, 32
 http.ServerRequest, 106
 http.ServerResponse, 106
 JavaScript, 236
 kontrola
 przeciążeń, 89
 przepływu, 89
 kontroler, 164
 konwencja semver, 69
 Kvalheim Christian Amor, 224

L

latency, *Patrz*: opóźnienie
 licznik, 94
 limit czasowy, 89, 195, 198
 Linuks
 Amazon, 23
 Ubuntu, 23
 Linux, 21
 lista Redis, 279
 locals object, *Patrz*: obiekt zmiennych lokalnych
 long polling, *Patrz*: odpytywanie wydłużone

Ł

łańcuch
 zapytania, 118, 119, 148
 parsowanie, 148
 znaków, *Patrz*: znak łańcuch

M

magazyn pamięciowy, 274
 mapa
 relacji, 281
 tras, *Patrz*: trasa mapa
 mapowanie obiektowo-dokumentowe, *Patrz*: ODM
 Menedżer Pakietów Node, *Patrz*: NPM
 metoda
 .apply, 34
 .bind, 40
 .call, 34
 .filter, 39
 .forEach, 39
 .isArray, 39
 .keys, 38
 .lastIndexOf, 39
 .reduce, 39
 .reduceRight, 39
 .toString('utf8'), 96

metoda

- .trim, 39
- __defineGetter__, 41
- __defineSetter__, 41
- addEventListener, 61
- app.disable, 171
- app.disabled, 171
- app.enable, 171
- app.enabled, 171
- app.error, 173
- app.set, 166, 171
- assert.ok, 307, 308
- basicAuth, 160
- blokująca, 46
- bodyParser, 151, 154, 266
- client.multi, 284
- configure, 171
- connect, 103
- console.log, 75
- cookieParser, 153, 154, 225
- createClient, 250
- createServer, 93, 94, 103, 111, 137, 149
- del, 129, 166
- DELETE, 116, 260
- dispatchEvent, 61
- emit, 61
- express.createServer, 165
- findOne, 234
- fs.createReadStream, 83
- fs.readdir, 71
- fs.watch, 84
- fs.watchFile, 84
- get, 166
- GET, 116, 126
- head, 129, 166
- HTTP, 113, 115
- http.request, 128
- index, 239
- indexOf, 308
- isDirectory, 74
- join, 113
- JSON.parse, 39
- JSON.stringify, 39
- listen, 94, 130
- logger, 148, 154
- methodOverride, 160
- modułu
 - expect.js, 309
- mongoose.connect, 236
- net.Stream#setEncoding, 96
- next, 143, 168, 231
- nieblokująca, 46
- obsługi zdarzenia, 50
- on, 61
- once, 161

- patch, 166
- PATCH, 116, 260
- post, 129, 166
- POST, 116
- pośrednicząca, 239
- pośrednicząca, 135, 140, 141, 146, 148, 153, 154, 160, 166, 173
 - do uwierzytelniania, 233
 - dołączana warunkowo, 177
 - kompatybilna z Connect, 177
 - konfigurowalna, 142
 - przypisanie, 147
 - tworzenie, 143
 - wielokrotnego użytku, 142
- process.exit, 81
- process.stdout.write, 75
- put, 129, 166
- PUT, 116
- query, 148
- readFile, 82
- removeEventListener, 61
- removeListener, 61
- render, 167, 168
- request, 122
- request.get, 128
- require, 172
- send, 129, 184
- sequelize.define, 264
- session, 154, 155, 225
 - magazyn, 159
- set, 129, 165
- setEncoding, 122
- setTimeout, 56, 110
- slice, 80
- static, 140, 146, 147, 266
 - hidden, 148
 - maxAge, 147
- stringify, 123
- updateAttributes, 270
- use, 149
- write, 110
- writeFile, 82
- writeHead, 109
- żądania, 116
- middleware, *Patrz:* metoda pośrednicząca
- Mocha, 310, 311, 315
 - instalowanie, 313
- model
 - definiowanie, 264
 - DOM, 294, 298
 - Mongoose, 243
 - obiektowy Redis, 276
 - REST, *Patrz:* REST
- modularność, 178

moduł

- assert, 298, 308
- bezwzględny, 57
- connect-redis, 159
- definiowanie, 26
- ejs, 172
- expect.js, 308
- formidable, 151
- fs, 68, 70, 77, 82
- http, 105, 136, 141
- instalacja, 25
- natywne, 136
- net, 103
- nieblokujący, 48
- querystring, 118, 119, 126
- search, 168
- sequelize, 260
- superagent, 128, 129, 164, 307
- udostępnianie, 295
- względny, 58
- zewnętrzny, 57
- MongoDB, 221, 222, 223, 235
 - indeksowanie, 223
 - instalacja, 223
 - wydajność, 223
- Mongoose, 223, 236, 242
 - model, 243
- mounting, *Patrz:* podłączanie, przypisanie
- multipleksacja, 200
- MySQL, 247

N

- nagłówek, 107, 108
 - Authorization, 126
 - Connection, 109, 112
 - Content-Type, 107, 110, 111, 112, 117, 128
 - Cookie, 153
 - Transfer-Encoding, 109
- najemca, 49
- name, 27, 40
- namespace, *Patrz:* przestrzeń nazw
- narzędzie
 - binarne, 27
 - browserbuild, *Patrz:* browserbuild
 - Firebug, *Patrz:* Firebug
 - obsługa danych, 294
 - REPL, *Patrz:* REPL
 - telnet, *Patrz:* telnet
 - up, 130
 - Web Inspector, *Patrz:* Web Inspector
 - wiersza poleceń, 27
- niepodzielność operacji, 235
- Node Package Manager, *Patrz:* NPM
- Node.JS instalacja, 21, 23

- node-mysql, 248
- node-XMLHttpRequest, 298
- nonprinting character, *Patrz:* znak niedrukowalny
- NoSQL, 247
- notacja z kropką, 238
- NPM, 25, 26, 69, 224
 - instalacja modułu, 25
 - rejestr, 28
- null, 33

0

obiekt

- błędu, 71
- Buffer, 63, 96
- console, 57
- Date, 294
- Error, 173
- EventEmitter, 73
- fs.WriteStream, 83
- global, 56
- globalny, 56
- http.ServerResponse, 110
- JavaScript, 111
- Math, 294
- odpowiedzi, 111
- połączenia, 111
- process, 56, 82
- process.argv, 79
- process.env, 81
- process.EventEmitter, 61
- process.stdin, 161
- procesu, 56
- req, 141
- req.session, 156, 233
- Request, 173
- res, 141
- Response, 173
- rozszerzenie, 173
- Schema, 237
- sesji, 233
- Stream, 73, 94
- struktura danych, 119
- window, 56, 61
- XMLHttpRequest, 61
- zmiennych lokalnych, 168
- zadania, 111
- Object-Relational Mapper, *Patrz:* ORM
- obserwacja, 83, 84
- ODM, 223
- odpowiedź, 106
- odpytywanie wydłużone, 198
- odwzorowanie obiektowo-relacyjne, *Patrz:* ORM
- operacja wejścia-wyjścia, 47, 49, 63, 73, 83

operator
 ~, 308
 instanceof, 32, 36
 typeof, 32, 33, 39
 opóźnienie, 187
 ORM, 223, 247
 OS X, 21

P

parsowanie, 118, 128, 148, 199
 ręczne, 212
 pętla
 wyczytaj-wykonaj, *Patrz:* REPL
 zdarzeń, 46, 47
 PHP, 45, 46
 pipe, *Patrz:* potok
 plik
 CSS, 84
 deskryptor, 47
 dziennika, *Patrz:* dziennik
 index.js, 92
 JSON, 154
 package.json, 26, 27, 58, 69, 92, 102, 112,
 121, 139, 164, 224, 236
 przetwarzany poleceniem node, 17
 statyczny, 138, 139, 141
 ukryty, 148
 widoku, 167
 wysyłanie grupowe, 153
 podłączanie, 179
 podobiekt, 277
 polecenie
 \$ npm install, 69
 console.log, 111
 ensureIndex, 232
 express, 28
 GET, 276
 HEXISTS, 276
 HGETALL, 287
 KEYS, 275
 module.export, 154
 node, 16, 68
 npm install, 112
 npm publish, 26
 redis-cli, 275
 require, 154, 224
 search, 28
 SELECT, 258
 SET, 276
 SMEMBERS, 276
 USER, 103
 view, 28

połączenie, 88, 94, 111
 aktywne, 94
 bezpieczne, 125
 błąd, 95
 SSL, 125
 telnet, 106
 port http://localhost:3000, 106
 potok, 47, 110
 potwierdzenie, 89, 207
 powłoka, 81
 systemowa wiersz poleceń, 22
 proces długotrwały, 45
 program
 asynchroniczny, 47, 70, 76
 pocztowy, 87
 zakończenie, 81
 programowanie
 behawioralne, 313
 obiektowe, 294
 oparte na testach, *Patrz:* TDD
 projekt
 nazwa, 27
 obiekt zależności, 27
 publikacja, 27
 wersja, 27
 protokół
 HTTP, *Patrz:* HTTP
 IP, 88
 IRC, *Patrz:* IRC
 połączeniowy, 87
 TCP, *Patrz:* TCP
 transportowy, 87
 warstwy dostępu, 224
 WebSocket, 184
 prototyp, 35
 przeciążenie, 89
 przeglądarka, 56
 Chrome, *Patrz:* Chrome
 Safari, *Patrz:* Safari
 przestrzeń nazw, 199
 przetwarzanie wsadowe, 74
 przypisanie, 147
 pseudonim, 92, 96, 97, 99, 103, 203

Q

query string, *Patrz:* łańcuch zapytania

R

ramkowanie, 184
 RAW TCP, 90
 Read-Eval-Print Loop, *Patrz:* REPL

Redis, 158, 159, 274
 instalacja, 275
 język zapytań, 275
 opcje utrwalania, 274
 zalety, 280
 refaktoring, 17, 68, 75
 refaktoryzacja, 243
 referencja, 32
 rejestr NPM, 26, 69
 REPL, 17, 24
 REPL Node, 23
 repozytorium GitHub, *Patrz:* GitHub
 request, *Patrz:* żądanie
 response, *Patrz:* odpowiedź
 REST, 260
 rozgłaszanie, 195, 203, 204

S

Safari, 296
 sekwencja sterująca ANSI, 82
 self-invoked functions, *Patrz:* funkcja samowywołująca
 sequelize, 260, 263, 271
 pobieranie danych, 268
 relacja, 265
 usuwanie danych, 269
 serial execution, *Patrz:* przetwarzanie wsadowe
 serwer
 HTTP, 111, 122
 HTTP Node, 130
 kod, 92
 proxy, 195, 198
 przeładowanie, 130
 Redis, 275
 struktur danych, 273
 TCP, 92, 111
 w różnych sieciach, 121
 WWW, 88, 90, 106, 110
 sesji dane, 159
 sesja użytkownika, 154, 158, 280
 shared-state concurrency, *Patrz:* współbieżność stanu dzielonego
 shimming, *Patrz:* adaptacja
 sieć społecznościowa, 182, 200, 222
 SIGKILL, 81
 signal, *Patrz:* sygnał
 silnik
 JavaScriptCore VM, 296
 v8, 31, 296
 skróty, 150
 skrypt, 24
 socket, *Patrz:* gniazdo
 socket.io, 17, 197, 198, 200
 sortowanie, 240

SQL injection, SQL wstrzyknięcie kodu, 257
 stack trace, *Patrz:* stos wywołań ślad
 sterownik, 224
 stos
 wykonania, *Patrz:* stos wywołań wywołań, 40, 48
 ślad, 40, 51
 strategia organizacji, 178
 strumień
 danych, 88
 do odczytu, 83, 138
 do zapisu, 83, 138
 kodowanie, 75
 stderr, 72
 stdin, 72
 stdout, 72
 wejścia, 123
 wyjścia, 71
 strumieńprzetwarzanie potokowe, 110
 styl
 BDD, 313
 eksportu, 314
 TDD, 314
 superagent, *Patrz:* moduł superagent, *Patrz:* moduł superagent
 sygnał SIGKILL, 81
 system
 logowania, 154
 plików, 110
 dostęp asynchroniczny, 71
 dostęp synchroniczny, 71
 szablonów, 164
 szablon, 164, 225, 253
 mechanizm, 165, 167, 172, 294

Ś

środowisko produkcyjne, 171

T

tablica, 33, 71, 113, 258
 argv, 126
 asocjacyjna, 274, 277, 279
 asocjacyjna opcji, 235
 ciągów znaków, 279
 metody, 39
 taktowanie, 195
 TCP, 87, 88, 92, 106, 183
 TDD, 314
 technologia
 AJAX, *Patrz:* AJAX
 WebSocket, *Patrz:* WebSocket
 telnet, 50, 89, 106, 184, 275
 terminal, 23

test

- kodu asynchronicznego, 311
- kodu źródłowego, 305
- sprawdzający, 305
- tworzenie, 306
- udostępnianie, 314

test-driven development, *Patrz:* TDD

testowanie automatyczne, 305

timeout, *Patrz:* limit czasowy

TinySong, 210

Transmission Control Protocol, *Patrz:* TCP

transport, 198, 199

trasa, 166, 175, 228, 248, 258, 261

- definiowanie, 226
- dopasowanie, 176
- format, 166
- kontrola przepływu, 176
- mapa, 178
- z parametrami, 175

trasowanie, 171

tryb

- bezpieczny, 235, 236
- bezprotokołowy, 90
- RAW TCP, 90

Twitter, 121, 124, 164, 306

aplikacja, 125

typ

- array, 32
- boolean, 32
- danych, 223
- function, 32
- konwersja, 242, 260
- null, 32
- number, 32
- Number, 277, 294
- object, 32
- ObjectId, 237
- prosty, 32
- Sequelize, 264
- string, 32
- String, 277, 294
- undefined, 32
- złożony, 32

U

użytkownik, 176, 250

- czatu, 97, 98, 99, 100
- logowanie, 154
- mapa relacji społecznych, 281
- obserwacja, 283
- obserwowany, 281
- obserwujący, 281
- profil, 222
- pseudonim, *Patrz:* pseudonim

sesja, *Patrz:* sesja użytkownika

tożsamość, 200

uwierzytelnianie, 225, 233

uwierzytelnienie, 177

znajomy, 284

V

version, 27

view, *Patrz:* widok

W

wartość

", 33

0, 33

null, *Patrz:* null

undefined, 33

wątek wykonawczy, 47, 49

Web Inspector, 17

WebSocket, 184, 195, 197, 198, 294, 298

websocket.io, 185, 186

wersja

numer, 147

numeracja kontroli, 69

White Nathan, 223

widok, 164, 167, 253

wiersz poleceń, 68, 79

Windows, 21

wirtualizacja, 49

właściwość

__proto__, 40

headers, 111

length, 34

name, 40, 119

private, 27

req.connection, 112

url, 114

współbieżność, 43, 47, 48, 71

stanu dzielonego, 44, 97

wydajność, 43, 48

wyjątek AssertionError, 306

wymiana potwierdzeń, 184

wywołanie

blokujące, 70

nieblokujące, 70, 173

wejścia-wyjścia, 173

zwrotne, 44, 46, 47, 51, 63, 93, 94, 230, 250

X

XML, 117

xmlhttprequest, 294

XTerm, 23

Z

zakres
 definiowanie, 35
 wewnętrzny, 35
 zależność, 57, 69, 70, 92, 139, 144, 165, 224, 260
 config, 251
 express, 185, 248
 mysql, 251
 node-mysql, 249
 zapytanie, 240
 AJAX, 198, 266
 SQL, 247
 zasada DRY, 139
 zbiór, 279, 280
 sortowany, 280
 zdarzenie, 61, 198
 close, 95, 103, 195, 198
 connect, 103, 198
 connection, 199
 data, 62, 103
 delegacja, 269
 disconnect, 198
 emisja, 61, 62
 end, 62, 95, 119, 122
 error, 50, 62, 95, 252
 generowanie, 198
 nasłuchiwanie, 61, 62, 96, 103, 198
 obiektu process, 82
 open, 198
 połączenia, 250
 uncaughtException, 50

zmienna
 _method, 160
 globalna, 56, 70
 exports, 57, 59
 module, 57, 59
 require, 57
 licznika, *Patrz:* licznik
 NODE_ENV, 81
 process, 70
 prywatna, 35
 res.body, 169
 środowiskowa, 81
 url, 116
 znak
 ciąg, *Patrz:* ciąg znaków
 diakrytyczny, 108
 łańcuch, 39
 niedrukowalny, 82
 ucieczki, 257

Ż

żądanie, 106, 112
 asynchroniczne, 182
 czas odpowiedzi, 142, 144
 DELETE, 160
 GET, 307
 HTTP, 73, 125, 126, 182
 kolejność, 182
 PATCH, 160
 POST, 182
 PUT, 160

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

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>



WILEY

ISBN 978-83-246-6674-4



Cena 59,00 zł

9 788324 666744

Informatyka w najlepszym wydaniu