



Technologia i rozwiązania

Node.js

Projektowanie, wdrażanie i utrzymywanie aplikacji



Sandro Pasquali

[PACKT] open source*
PUBLISHING community experience distilled

Tytuł oryginału: Deploying Node.js

Tłumaczenie: Rafał Jońca

ISBN: 978-83-283-3609-4

Copyright © Packt Publishing 2015. First published in the English language under the title 'Deploying Node.js – (9781783981403)'.

Polish edition copyright © 2017 by Helion SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

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.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą 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)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/nodepr>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorze	9
O redaktorach merytorycznych	11
Wstęp	13
Zawartość książki	14
Narzędzia potrzebne do realizacji przykładów	15
Do kogo kierowana jest książka?	15
Konwencje typograficzne	16
Przykładowy kod	16
Rozdział 1. Docenić Node	17
Unikatowo zaprojektowany Node	18
Współbieżność	20
Równoległość i wątki	21
Współbieżność i procesy	23
Zdarzenia	24
Pętla zdarzeń	26
Wpływ sposobu zaprojektowania Node na architektów systemów	30
Budowanie większych systemów z mniejszych	30
Strumienie	32
Wykorzystanie JavaScript na każdym etapie projektu maksymalizuje dobre efekty	33
Zamiana kodu	34
Browserify	34
Podsumowanie	35
Rozdział 2. Instalacja i wirtualizacja serwerów Node	37
Uruchamianie podstawowego serwera Node	38
Witaj, świecie	38
Tworzenie żądań HTTP	39
Pośredniczenie i tunelowanie	40

HTTPS, TLS (SSL) i zabezpieczanie serwera	42
Tworzenie podpisanego samodzielnie certyfikatu na potrzeby programistyczne	42
Instalacja prawdziwego certyfikatu SSL	43
Instalacja aplikacji w chmurze Heroku	44
Dodatki	46
Git	46
Zarządzanie zmiennymi konfiguracyjnymi	49
Zarządzanie wdrożeniem	50
Instalacja aplikacji w chmurze OpenShift	51
Instalacja aplikacji Node i MongoDB	52
Wdrożenie aplikacji	52
Docker, czyli lekkie i wirtualne kontenery	54
Najpierw jednak tylko systemu Unix	56
Korzystanie z systemu Docker	57
Tworzenie pliku Dockerfile	58
Zbudowanie i uruchomienie obrazu Dockera	59
Podsumowanie	61
Rozdział 3. Skalowanie Node	63
Skalowanie pionowe wykorzystujące wiele rdzeni procesora	64
Funkcja spawn()	65
Funkcja fork()	68
Funkcja exec()	70
Funkcja execFile()	71
Komunikacja z procesem potomnym	71
Moduł cluster	73
Skalowanie w poziomie z wykorzystaniem wielu maszyn	77
Użycie serwera Nginx	77
Równoważenie obciążenia za pomocą Node	85
Użycie kolejek komunikatów	87
Użycie modułu komunikacji UDP serwera Node	91
Podsumowanie	96
Rozdział 4. Zarządzanie pamięcią i przestrzenią	97
Jak poradzić sobie z dużym tłumem?	98
Mikroserwisy	98
Mechanizm pub-sub serwera Redis	99
Mikroserwisy w narzędziu Seneca	102
Zmniejszenie zużycia pamięci	105
Używaj strumieni zamiast buforów	106
Prototypy	107
Wydajne pamięciowo struktury danych z wykorzystaniem serwera Redis	109
V8 i optymalizacja wydajności	116
Optymalizacja kodu JavaScript	116
Strategie dotyczące pamięci podręcznych	121
Wykorzystanie Redis jako pamięci podręcznej	121
Wdrażanie CloudFlare jako CDN	125

Zarządzanie sesją użytkownika	127
Uwierzytelnianie i sesje JWT	127
Podsumowanie	131
Rozdział 5. Monitorowanie aplikacji	133
Jak poradzić sobie z błędami?	134
Moduł domain	135
Wyłapywanie błędów innych procesów	137
Tworzenie dzienników zdarzeń	140
Dostosowywanie działania w zmieniającym się środowisku	144
REPL narzędzia Node	144
Zdalny monitoring i zarządzanie procesem Node	146
Profilowanie procesów	149
Wykorzystywanie niezależnych narzędzi do monitorowania	152
PM2	152
Wykorzystanie New Relic do monitorowania aplikacji	156
Podsumowanie	158
Rozdział 6. Budowanie i testowanie	159
Budowanie aplikacji za pomocą narzędzi Gulp, Browserify i Handlebars	160
Narzędzie Gulp	160
Wykonanie szkieletu systemu budowania	163
Uruchamianie i testowanie zbudowanej wersji	169
Wykorzystanie narzędzi do testowania wbudowanych w system Node	171
Debugger wbudowany w system Node	172
Moduł assert	175
Testowanie za pomocą Mocha, Chai, Sinon i npm	177
Mocha	178
Chai	180
Sinon	181
Automatyzacja testów w przeglądarce internetowej dzięki PhantomJS i CasperJS	187
Testowanie z wykorzystaniem PhantomJS	187
Scenariusze nawigacyjne w CasperJS	189
Podsumowanie	192
Rozdział 7. Wdrażanie i konserwacja	193
GitHub i mechanizm webhook	194
Włączenie mechanizmu webhook	195
Implementacja systemu budowania	
i wdrażania wykorzystującego mechanizm webhook	198
Synchronizacja wersji lokalnej i zdalnej	201
Tworzenie wersji lokalnej za pomocą narzędzia Vagrant	201
Przygotowywanie systemu przy użyciu Ansible	205
Integracja, dostarczanie i wdrażanie	208
Ciągła integracja	208
Ciągłe dostarczanie	208

Ciągłe wdrażanie	209
Budowanie i wdrażanie z użyciem narzędzia Jenkins	209
Wdrażanie w chmurze Heroku	213
Zarządzanie pakietami	216
Wersjonowanie semantyczne	216
Zarządzanie pakietami z użyciem npm	218
Skorowidz	225

Docenić Node

Node istnieje już ponad siedem lat i każdego roku wykorzystuje się go coraz intensywniej. W zasadzie można powiedzieć, że nie istnieje już ryzyko, że Node pojawi się jak gwiazda pop i równie szybko zniknie. Node to naprawdę poważna technologia budowana przez bardzo uzdolniony zespół główny i wspomagana przez aktywną społeczność, która cały czas czuwa nad poprawą szybkości działania, bezpieczeństwa i użyteczności.

Każdego dnia programiści napotykają problemy, które Node stara się rozwiązać. Oto kilka z nich:

- skalowanie aplikacji sieciowych jako rozwiązań z wieloma serwerami;
- zapobieganie wąskim gardłom w komunikacji wejście-wyjście (baza danych, plik, komunikacja sieciowa);
- monitorowanie systemów i sprawdzanie ich wydajności;
- testowanie integralności komponentów systemu;
- bezpieczne zarządzanie współbieżnością;
- łatwe umieszczanie zmian w kodzie w środowiskach produkcyjnych.

W tej książce omawiamy techniki związane z wdrażaniem, skalowaniem, monitorowaniem, testowaniem i utrzymaniem aplikacji Node. Skupimy się na głównej cesze wyróżniającej Node, czyli modelu nieblokującym sterowanym zdarzeniami i jego efektywnym wykorzystaniu do realizacji własnych zadań.

W dniu 28 lutego 2014 roku Eran Hammer wygłosił wykład otwierający NodeDay, dużą konferencję dla programistów sponsorowaną przez firmę PayPal. Zaczął od wymienienia kilku liczb istotnych dla jego pracodawcy, firmy Walmart:

- 11 000 sklepów;
- ponad trylion dolarów sprzedaży każdego roku;
- ponad 2 miliony pracowników;
- największy prywatny pracodawca na świecie.

Następnie powiedział:

55% ruchu w czarny piątek, czyli naszym szczycie sezonu, każdego roku powoduje realizację około 40% rocznych przychodów. 55% tego ruchu pochodzi z urządzeń przenośnych... 55% ruchu przechodziło w 100% poprzez serwery Node. (...) Udało nam się udźwignąć ten ogromny ruch w systemie równoważnym dwóm procesorom i 30 GB RAM. To wszystko. Tylko tyle potrzebował Node, aby obsłużyć 100% ruchu mobilnego w czarny piątek. (...) Walmart to rynek e-commerce wartý 10 bilionów dolarów i do końca roku te 10 bilionów będzie przechodziło w całości przez Node.

Eran Hammer, starszy architekt, Walmart Labs

Nowoczesne oprogramowanie staje się z różnych powodów coraz bardziej złożone i w dużej mierze zmienia nasz sposób budowania aplikacji. Większość nowych platform i języków stara się w jakiś sposób odpowiedzieć na tę zmianę. Node nie jest tu wyjątkiem i to samo dotyczy JavaScript jako języka.

Nauka Node oznacza poznawanie programowania sterowanego zdarzeniami, kompozycję programu z modułów, tworzenie i łączenie strumieni danych, a także produkcję i konsumpcję zdarzeń i powiązanych z nimi danych. Architektura bazująca na Node składa się często z wielu małych procesów i usług komunikujących się ze sobą za pomocą zdarzeń — wewnętrznie za pomocą interfejsu EventEmitter i wywołań zwrotnych, a zewnętrznie za pomocą kilku standardowych warstw transportowych (np. HTTP i TCP) oraz cienkiej warstwy przekazywania komunikatów wykorzystującej warstwę transportową (np. OMQ, Pub-Sub w Redis lub Kafka). Bardzo często procesy te wykorzystują kilka bezpłatnych, wysokiej jakości modułów **npm** dostępnych na zasadach otwartego kodu źródłowego. Każdy z takich modułów zawiera testy, dokumentację lub przykłady użycia.

W tym rozdziale postaram się przedstawić ogólny zarys Node, wskazać problemy, które stara się rozwiązać, rozwiązania narzucane przez sposób zaprojektowania Node, a także wyjaśnić, co to wszystko oznacza dla programisty. Pokróćce przedstawię też główne tematy, które rozwinę w dalszych rozdziałach, na przykład odpowiednią strukturę zapewniającą wydajne i stabilne serwery Node, użycie najlepszych rozwiązań JavaScript dla aplikacji i zespołu, a także przedstawienie się na sposób myślenia w kategoriach Node.

Zacznijmy od wyjaśnienia, jak i dlaczego właśnie tak zaprojektowano Node.

Unikatowo zaprojektowany Node

Operacje wejścia-wyjścia (dysk, sieć) są bez wątpienia bardziej kosztowne niż pamięć operacyjna. Poniższa tabela przedstawia cykle zegara konsumowane przez typowe zadania systemowe (wartości pochodzą z prezentacji Ryana Dahla na temat Node — <https://www.youtube.com/watch?v=ztspvPYybIY>).

Pamięć podręczna L1	3 cykle
Pamięć podręczna L2	14 cykli
RAM	250 cykli
Dysk	41 000 000 cykli
Sieć	240 000 000 cykli

Powody widać wyraźnie — dysk to urządzenie fizyczne, często jeszcze w postaci wirujących dysków, więc pobiera dane znacznie wolniej niż urządzenia czysto półprzewodnikowe (na przykład pamięć operacyjna) czy dedykowane pamięci podręczne. Podobnie dane nie wędrują w sieci z punktu A do punktu B w sposób natychmiastowy. Światło potrzebuje 0,1344 sekundy, aby okrążyć kulę ziemską! W sieci używanej przez miliony osób wchodzących ze sobą w liczne interakcje szybkość przesyłania informacji jest znacznie wolniejsza niż prędkość światła, bo nie ma tu wielu linii prostych, a zdarzają się objazdy i zatory. W efekcie opóźnienie wzrasta znacząco.

Gdy nasze oprogramowanie działało na komputerach osobistych pod biurkiem, przez sieć nie realizowało się prawie żadnej komunikacji. Opóźnienia w interakcji z edytorem tekstu lub arkuszem kalkulacyjnym dotyczyły tak naprawdę jedynie czasu dostępu do dysku twardego. Ten czas dostępu udało się znacząco poprawić. Po zastosowaniu pamięci podręcznej lub szybszego dysku oprogramowanie wydaje się użytkownikowi znacznie szybsze. Przyzwyczajeni do tej szybkości użytkownicy oczekują tego samego od innych narzędzi.

Pojawienie się rozwiązań chmurowych i oprogramowania bazującego na przeglądarkach internetowych spowodowało, że dane opuściły lokalne dyski i zamieszkały na dyskach zdalnych, a dostęp do danych odbywa się poprzez internet. Czasy dostępu do danych znowu uległy znacznemu wydłużeniu. Sieciowe wejście-wyjście jest powolne. Mimo to coraz to więcej firm migruje fragmenty swoich aplikacji do **chmury**, a coraz więcej rozwiązań bazuje tylko i wyłącznie na wersji sieciowej.

Mechanizmy Node zaprojektowano, by przyspieszyć komunikację wejście-wyjście, czyli zaprojektowano je dla nowego, sieciowego świata, gdzie dane znajdują się w wielu miejscach i muszą być szybko składane w jedną całość. Wiele tradycyjnych frameworków do budowania aplikacji internetowych powstawało w czasie, gdy użytkownik pracował na swoim komputerze stacjonarnym i od czasu do czasu wykonywał zapytania do pojedynczego serwera wykorzystującego relacyjną bazę danych. Nowoczesne oprogramowanie musi obsługiwać dziesiątki tysięcy jednoczesnych połączeń od klientów i używać wielu współdzielonych puli danych wykorzystujących różne protokoły, a sami użytkownicy komunikują się z systemem za pomocą różnorodnych urządzeń. Node zaprojektowano z myślą o budowaniu tego rodzaju oprogramowania.

Co oznacza współbieżność, zrównoleglenie, wykonywanie asynchroniczne, wywołanie zwrotne i zdarzenie dla programisty Node?

Współbieżność

Wykonywanie kodu w sposób proceduralny, czyli po kolei, to sensowny pomysł. Sami tak robimy, wykonując zadania w życiu codziennym i przez długi czas języki programowania były naturalnie proceduralne. Przecież instrukcje wysyłane do procesora muszą być wykonywane w przewidywalnym porządku. Jeśli chcę pomnożyć 8 przez 6 i podzielić ten wynik przez 144 dzielone przez 12, a następnie dodać do uzyskanego wyniku 10, operacje muszą wykonać w określonej kolejności:

$$((8*6) / (144/12)) + 10$$

Kolejność nie może być jak poniższa:

$$(8*6) / ((144/12) + 10)$$

Taki przykład łatwo zrozumieć. Początkowo komputery posiadały tylko jeden procesor, a przetwarzanie jednej instrukcji blokowało przetwarzanie następnych. Obecnie ta prostota w zasadzie zanikła, bo nawet telefony komórkowe posiadają procesory wielordzeniowe.

Jeśli przyjrzymy się poprzedniemu przykładowi, to zauważymy, że obliczenie $144/12$ i 8×6 można realizować niezależnie — jedno zadanie nie zależy od drugiego. Problem można podzielić na mniejsze i przydzielić je wielu osobom lub procesom wykonawczym, aby realizowały zadania równolegle. Wynik tych obliczeń cząstkowych łączy się później w jedną całość, aby otrzymać wynik ostateczny.

Wiele procesów, gdy każdy z nich rozwiązuje symultanicznie jeden problem matematyczny, to przykład **zrównoleglenia**.

Rob Pike, współtwórca języka programowania Go (stworzonego w firmie Google), definiuje **współbieżność** w następujący sposób:

Współbieżność to takie ustrukturyzowanie rzeczy, aby jeśli to tylko możliwe, dopuszczalne było zrównoleglenie w celu realizacji zadania lepiej lub szybciej. Zrównoleglenie nie jest celem współbieżności; celem współbieżności jest dobra struktura.

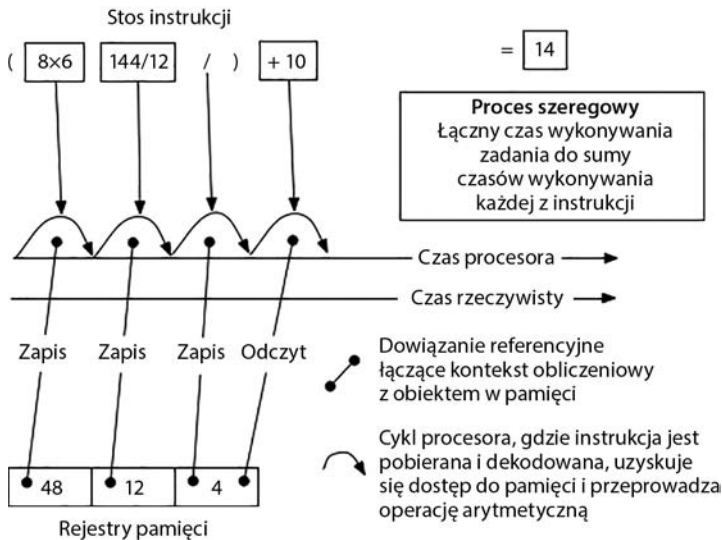
Współbieżność to nie zrównoleglenie. System korzystający z zasad współbieżności pozwala programiście na dekompozycję aplikacji **tak, jak gdyby** wiele niezależnych procesów wykonywało równocześnie wiele zadań. Udane frameworki zapewniające budowanie systemów o wysokiej współbieżności ułatwiają definiowanie właściwych rozwiązań i wykorzystanie odpowiedniego słownictwa.

Projekt Node sugeruje, że osiągnięcie głównego celu — łatwego budowania skalowalnych programów sieciowych — zawiera w sobie uproszczenie wykonywania współistniejących procesów i ich kompozycji. Node pomaga programiście poprawnie wnioskować na temat programu, w którym wiele rzeczy może dziać się jednocześnie (np. obsługa wielu klientów), a także lepiej zorganizować kod.

Przyjrzyjmy się różnicy między zrównolegleniem i współbieżnością, wątkami i procesami, a także specjalnemu sposobowi, dzięki któremu Node stosuje w swoim projekcie najlepsze części każdego z tych elementów.

Równoległość i wątki

Poniższy rysunek przedstawia, w jaki sposób tradycyjny procesor będzie wykonywał opisany wcześniej program.

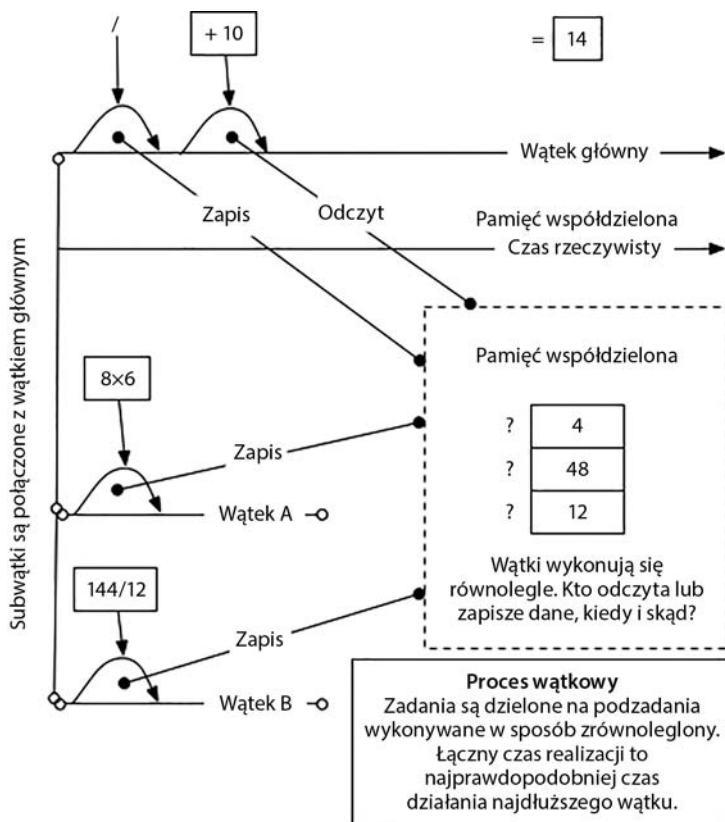


Program został podzielony na poszczególne instrukcje wykonywane w pewnej kolejności. Wszystko działa, ale wymaga, aby instrukcje były wykonywane po kolei, czyli gdy przetwarzana jest jedna instrukcja, następne muszą czekać. To proces blokujący — wykonywanie jednego segmentu łańcucha blokuje pozostałe segmenty. Mówimy o działaniu w sposób **jednowątkowy**.

Mam jednak dobrą wiadomość. Procesor ma (dosłownie) pełną kontrolę nad wszystkim, więc nie ma niebezpieczeństwa, że inny procesor nadpisze mu pamięć lub zmodyfikuje stan, z którego właśnie korzystał. Poświęciliśmy szybkość na rzecz stabilności i bezpieczeństwa.

Lubimy jednak szybkość — przedstawiony model stał się przestarzały, bo projektanci procesorów i programiści systemów wprowadzili systemy z obliczeniami zrównoleglonymi. Zamiast jednego blokującego wątku pojawiło się wiele współdziałających wątków.

To usprawnienie zdecydowanie zwiększyło szybkość obliczeń, ale wprowadziło nowe problemy, przedstawione na poniższym rysunku.



Rysunek przedstawia współdziałające wątki, które działają równolegle wewnątrz jednego procesu, co zmniejsza czas niezbędny na realizację obliczeń. Poszczególne wątki mają na celu rozbicie zadania, rozwiązanie go i złożenie w jedną całość. Ponieważ podzadania można realizować niezależnie, łączny czas działania może ulec znaczącemu skróceniu.

Wątki zapewniają zrównoleglenie wewnątrz jednego procesu. Każdy wątek reprezentuje jedną sekwencję (szeregowo wykonywanych) instrukcji. Proces może zawierać dowolną liczbę wątków.

Pojawiające się problemy wynikają ze złożoności związanej z synchronizacją wątków. Bardzo trudno modeluje się wysoce współbieżne rozwiązania za pomocą wątków, w szczególności w modelach, w których dochodzi do współdzielenia stanu. Nielatwo przewidzieć wszystkie scenariusze oddziaływania jednego wątku na drugi, jeśli nie wiadomo dokładnie, kiedy asynchronicznie wykonywany wątek zakończy swoje działania.

- Współdzielona pamięć i związane z tym blokowanie powodują, że bardzo trudno coś wnioskować o działaniu systemu, szczególnie gdy jest bardzo skomplikowany.

- Komunikacja między zadaniami wymaga implementacji dużej liczby elementów synchronizujących, takich jak blokady, semaforey, zmienne warunkowe itp. Już trudne środowisko wymaga naprawdę złożonych narzędzi, co zmusza do dysponowania sporą wiedzą nawet w przypadku tworzenia relatywnie prostego systemu.
- Wyścig i blokada wzajemna to typowe błędy pojawiające się w tego rodzaju systemach. Wsłz odpowiednią kolejnością operacji, bo dwa wątki mogą uczestniczyć w **wyścigu** o właściwą zmianę stanu, zdarzenia lub innej kluczowej charakterystyki systemu.
- Ponieważ zachowanie odpowiednich granic między wątkami i ich stanami jest tak trudne, zapewnienie, że biblioteka (w przypadku Node będzie to **pakiet** lub **moduł**) jest bezpieczna wątkowo, wymaga poświęcenia mnóstwa czasu po stronie programisty. Czy wiem, że biblioteka nie zniszczy pewnej części mojej aplikacji? Zagwarantowanie bezpieczeństwa wątkowego wymaga od twórcy biblioteki dużej uwagi, a gwarancje mogą też być warunkowe — na przykład biblioteka może wskazać, że jest bezpieczna wątkowo w kwestii odczytu, ale nie w kwestii zapisu.

Chcielibyśmy uzyskać szybkość związaną ze zrównolegleniem dzięki wątkom, ale bez zaprzatania sobie głowy semaforami i blokadami. W świecie systemu Unix ten pomysł nazywa się często **regułą prostoty**: *programiści powinni w trakcie projektowania dążyć do prostoty i poszukiwać sposobów podziału programu na niewielkie, współpracujące ze sobą elementy; zniechęca się tak programistów do tworzenia wspaniale złożonych systemów, które w praktyce są programami narażonymi na błędy.*

Współbieżność i procesy

Zrównoleglenie w jednym procesie to złożona iluzja osiągnana dzięki bardzo wyrafinowanemu sprzętowi i oprogramowaniu. Tak naprawdę jednak każdemu zależy na postrzeganym działaniu — jak system działa i jak jest programowany przez programistę. Wątki powodują, że zrównoleglenie jest bardzo wydajne, ale czynią współbieżność znacznie trudniejszą w analizie.

Zamiast zrzucać złożoność tego rodzaju na programistę, Node sam zarządza wątkami wejścia-wyjścia, upraszczając wszystko i wymagając od twórcy kodu jedynie sterowania przepływem między poszczególnymi zdarzeniami. Istnieje potrzeba **mikrozarządzania** wątkami wejścia-wyjścia — dzięki temu programista projektujący aplikację skupia się jedynie na miejscach dostępu do danych (wywołania zwrotne) i instrukcjach wykonywanych, gdy dane będą już dostępne. Pojedynczy strumień instrukcji, który jawnie przejmuje kontrolę i się jej zrzeka, nie naraża nas na kolizje i jest przewidywalny.

- Zamiast martwić się o blokady i kolizje, programista skupia się na tworzeniu łańcuchów instrukcji, których kolejność jest przewidywalna.
- Zrównoleglenie osiąga się za pomocą wielu procesów, gdzie każdy proces posiada własny obszar pamięci, co zapewnia nieskomplikowaną komunikację między procesami — dzięki **regule prostoty** uzyskujemy nie tylko proste i prawdopodobnie bezbłędne komponenty, ale również poprawiamy interoperacyjność.

- Stan nie jest (dowolnie) współdzielony między poszczególnymi procesami Node. Proces automatycznie chroni się przed niespodziewanymi wizytami innych procesów, które mogłyby zająć jego zasoby i namieszać w pamięci. Komunikacja odbywa się prostymi kanałami z użyciem prostych protokołów, co znacząco utrudnia pisanie programów wykonujących nieprzewidywalne zmiany w innych procesach.
- Bezpieczeństwo wątkowe to coś, co przestaje być dla programisty jedną z trosk i przyczyną straty czasu. Ponieważ współbieżność jednowątkowa zapobiega kolizjom trapiącym współbieżność wielowątkową, programowanie przebiega sprawniej i pewniej.

Pojedynczy wątek opisujący asynchroniczne sterowanie przepływem i zarządzany efektywnie przez pętlę zdarzeń zapewnia stabilność, czytelność, łatwość konserwacji i dużą odporność na błędy programów Node. Dobrą wiadomością jest to, że Node w dużej mierze pozwala wykorzystać szybkość i siłę wielowątkowości — dzięki sprytnemu zaprojektowaniu siła ta jest transparentna dla programisty, więc potrafi on z niej skorzystać przy dużo mniejszym wysiłku.

Zdarzenia

Wiele rozszerzeń JavaScript dostępnych w Node emituje zdarzenia. Są to instancje `events.EventEmitter`. Każdy obiekt może rozszerzać `EventEmitter`, co pozwala programistom budować zwarte, asynchroniczne interfejsy dla metod obiektowych.

Przyjrzymy się teraz prostemu przykładowi, który ilustruje wykorzystanie obiektu `EventEmitter` jako prototypu dla konstruktora. Ponieważ każda tworzona w ten sposób instancja będzie posiadała obiekt `EventEmitter` w łańcuchu prototypów, `this` zapewni naturalny dostęp do **API** (ang. *Application Programming Interface*) zdarzeń. Metody instancji mogą emitować zdarzenia, a te mogą być nasłuchiwane. W przykładzie emitujemy zdarzenie z najnowszą wartością licznika po wywołaniu metody `counter.increment`. W innym fragmencie kodu nasłuchujemy zdarzenia `incremented` i po prostu wyświetlamy zawartość licznika w wierszu poleceń.

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

var Counter = function(init) {
  this.increment = function() {
    init++;
    this.emit('incremented', init);
  }
}

util.inherits(Counter, EventEmitter);

var counter = new Counter(10);

var callback = function(count) {
  console.log(count);
};
```

```

}
counter.addListener('incremented', callback);

counter.increment(); // 11
counter.increment(); // 12

```

Dostęp do przykładów

Wszystkie prezentowane w książce przykłady znajdziesz w pliku dostępnym na stronie wydawnictwa pod adresem <ftp://ftp.helion.pl/przyklady/nodepr.zip>.

Aby usunąć funkcję nasłuchującą zdarzenia przypisaną do counter, użyj polecenia `counter.removeListener('incremented', callback)`.

Obiekt `EventEmitter` jest rozszerzalny, więc można skorzystać z pełnej elastyczności języka JavaScript. Na przykład obsługę strumieni danych wejścia-wyjścia można obsłużyć w sposób bazujący na zdarzeniach przy zachowaniu zasady asynchronicznego, nieblokującego programowania obowiązującej w Node:

```

var stream = require('stream');
var Readable = stream.Readable;
var util = require('util');

var Reader = function() {
  Readable.call(this);
  this.counter = 0;
}

util.inherits(Reader, Readable);

Reader.prototype._read = function() {
  if(++this.counter > 10) {
    return this.push(null);
  }
  this.push(this.counter.toString());
};

// Po zajęciu zdarzenia #data wyświetl fragment danych.
var reader = new Reader();
reader.setEncoding('utf8');
reader.on('data', function(chunk) {
  console.log(chunk);
});
reader.on('end', function() {
  console.log('--zakończono--');
});

```

W przedstawionym programie używamy strumienia `Reader` wysyłającego zbiór liczb — element nasłuchujący zdarzenia `data` strumienia pobiera liczby, gdy są generowane, i wyświetla je na

ekranie. Po zakończeniu strumienia program wyświetla dodatkowy komunikat o końcu strumienia. Kod nasłuchujący jest wywoływany raz na każdą liczbę, co oznacza, że wykonywanie zbioru liczb nie blokuje pętli zdarzeń. Ponieważ pętla zdarzeń Node musi zapewnić zasoby tylko do obsługi wywołań zwrotnych, wiele innych instrukcji może zostać wykonanych między każdym zdarzeniem.

Pętla zdarzeń

Kod stosowany w oprogramowaniu, które nie korzysta z sieci, jest najczęściej synchroniczny i blokujący. Operacje wejścia-wyjścia w poniższym pseudokodzie również są blokujące:

```
variable = produceAValue()
print variable
// Pewna wartość zostanie wyświetlona po zakończeniu #produceAValue.
```

Poniższa iteracja będzie odczytywała zawartość jednego pliku, wyświetlała ją, a następnie przechodziła do następnego pliku aż do zakończenia pętli:

```
fileNames = ['a', 'b', 'c']
while(filename = fileNames.shift()) {
  fileContents = File.read(filename)
  print fileContents
}
//> a
//> b
//> c
```

To dobre rozwiązanie w wielu sytuacjach. Ale co się stanie, jeśli pliki są bardzo duże? Jeśli pobranie jednego pliku zajmie sekundę, pobranie wszystkich zajmie trzy sekundy. Pobieranie jednego pliku czeka na pobranie wcześniejszego, co jest mało wydajne i powolne. Wykorzystując Node, możemy zainicjować jednoczesny odczyt wszystkich trzech plików:

```
var fs = require('fs');
var fileNames = ['a', 'b', 'c'];
fileNames.forEach(function(filename) {
  fs.readFile(filename, {encoding: 'utf8'}, function(err, content) {
    console.log(content);
  });
});
//> b
//> a
//> c
```

Ta wersja odczyta wszystkie trzy pliki jednocześnie. Każde wywołanie `fs.readFile` zwróci wynik w pewnym nieznanym momencie w przyszłości. Nie możemy w tej sytuacji mieć pewności, że pliki wyświetlą się w zadanej kolejności. W wielu sytuacjach czas niezbędny na pobranie wszystkich trzech plików będzie podobny do czasu potrzebnego na pobranie jednego z nich, czyli będzie zdecydowanie krótszy niż trzy sekundy. Zamieniliśmy przewidywalną kolejność wyświetlenia na szybkość działania i podobnie jak dla wątków synchronizacja współbieżnego

środowiska wymaga pewnej dodatkowej pracy. W jaki sposób zarządzać nieprzewidywalnymi zdarzeniami i jak je opisywać, aby kod był przejrzysty **oraz** wydajny?

Kluczową decyzją projektową dokonaną przez twórców Node było zaimplementowanie pętli zdarzeń jako menedżera współbieżności. Poniższy opis programowania bazującego na zdarzeniach (to fragment ze strony http://www.princeton.edu/~achaney/tmwe/wiki100k/docs/Event-driven_programming.html) nie tylko opisuje paradygmat sterowania zdarzeniami, ale również wprowadza w mechanizm działania zdarzeń w Node i wyjaśnia, dlaczego JavaScript jest idealnym językiem dla tego rodzaju paradygmatu.

W programowaniu komputerowym programowanie wykorzystujące zdarzenia lub programowanie bazujące na zdarzeniach to paradygmat, w którym przebiegiem programu sterują zdarzenia — wyjście sensora, akcja użytkownika (kliknięcie, naciśnięcie klawisza) lub komunikat z innego programu lub wątku.

Programowanie wykorzystujące zdarzenia można również zdefiniować jako sposób tworzenia architektury aplikacji, w której aplikacja posiada główną pętlę podzieloną na dwie części: pierwszą jest wybór zdarzenia (lub wykrywanie zdarzeń), a drugą jest obsługa zdarzenia (...).

Programy sterowane zdarzeniami można tworzyć w dowolnym języku programowania, ale zadanie to jest łatwiejsze, jeśli ma się do dyspozycji abstrakcje wysokiego poziomu, na przykład domknięcia.

Jak wcześniej wykazaliśmy, środowiska jednowątkowe będą się blokowały, a co za tym idzie, będą działały powoli. V8 to jednowątkowe środowisko wykonawcze dla programów JavaScript.

W jaki sposób tę architekturę jednowątkową uczynić efektywną?

Node czyni pojedyncze wątki wydajniejszymi, bo deleguje wiele operacji blokujących do elementów systemu operacyjnego, więc główny wątek V8 działa tylko wtedy, gdy są dane do obróbki. Główny wątek (program wykonywany przez Node) wyraża zainteresowanie pewnymi danymi (na przykład poprzez `fs.readFile`) i przekazuje wywołanie zwrótne pozwalające na otrzymanie informacji, gdy dane będą dostępne. Dopóki dane nie nadejdą, główny wątek V8 może realizować inne zadania. Jak to się dzieje? Node deleguje zadania wejścia-wyjścia do biblioteki **libuv**, którą można opisać następująco (to fragment ze strony <http://nikhilm.github.io/wbook/basics.html#event-loops>):

W programowaniu wykorzystującym zdarzenia aplikacja wyraża zainteresowanie pewnymi zdarzeniami i reaguje na nie, gdy już zajdą. Odpowiedzialność za zbieranie zdarzeń z systemu operacyjnego i monitorowanie innych źródeł zdarzeń to zadanie biblioteki libuv. Użytkownik biblioteki jedynie rejestruje funkcje zwrótne wywoływane w momencie zajścia zdarzenia.

W przedstawionym cytacie **użytkownik** to proces Node wykonujący program JavaScript. **Wywołania zwrótne** to funkcje JavaScript. Za zarządzanie wywołaniami zwrótnymi odpowiada pętla zdarzeń Node. Node zarządza kolejką żądań wejścia-wyjścia, stosując libuv, które odpowiada

za komunikację z systemem operacyjnym i przekazanie wyniku do funkcji zwrotnej napisanej w języku JavaScript.

Rozważmy następujący kod:

```
var fs = require('fs');
fs.readFile('test.js', {encoding:'utf8'}, function(err, fileContents) {
  console.log('Następnie jest dostępna treść:', fileContents);
});
console.log('To nastąpi wcześniej.');
```

Program wyświetli poniższy wynik:

```
> To nastąpi wcześniej.
> Następnie jest dostępna treść: [treść pliku]
```

Oto co się dzieje w Node w trakcie wykonywania programu.

- Node wczytuje moduł fs. Zapewnia on dostęp do fs.binding, które jest „odwzorowaniem statycznym zdefiniowanym w src/node.cc, które łączy kod C++ i JavaScript” (https://groups.google.com/forum/#!msg/nodejs/R5fDzBr0eEk/lrCKaJX_6vIJ).
- Metoda fs.readFile otrzymuje instrukcje dotyczące pliku i wywołanie zwrotne w postaci funkcji JavaScript. Poprzez fs.binding libuv zostaje powiadomione o żądaniu odczytu pliku i otrzymuje specjalnie przygotowaną wersję wywołania zwrotnego, zapewniającą późniejsze uruchomienie właściwej funkcji.
- Biblioteka libuv wywołuje funkcje systemu operacyjnego niezbędne do odczytu pliku, wykorzystując przy tym własną pulę wątków.
- Program JavaScript kontynuuje działanie, wyświetlając *To nastąpi wcześniej*. Ponieważ jest aktywne wywołanie zwrotne, pętla zdarzeń czeka na otrzymanie informacji zwrotnej.
- Gdy deskryptor pliku zostanie odczytany przez system operacyjny, libuv (poprzez wewnętrzne mechanizmy) jest o tym informowane i zostaje uruchomione wywołanie zwrotne przekazane do libuv, które w praktyce przygotowuje oryginalne wywołanie zwrotne do umieszczenia w głównym wątku (wykonywanym przez V8).
- Oryginalne wywołanie zwrotne trafia do kolejki pętli zdarzeń i będzie wykonane w następnym cyklu.
- Zawartość pliku zostaje wyświetlona na ekranie.
- Ponieważ nie ma innych wywołań zwrotnych, proces kończy swoje działanie.

Tutaj widzimy kluczowe rozwiązania, które Node stosuje do obsługi szybkiego, łatwego w zarządzaniu i skalowalnego wejścia-wyjścia. Jeśli program 10 razy odczytywałby plik *test.js*, łączny czas realizacji pozostałby mniej więcej taki sam. Każde wywołanie byłoby uruchomione **równolegle** w swoim **własnym wątku** w puli wątków biblioteki libuv. Choć pisaliśmy kod w JavaScript, tak naprawdę uruchomiliśmy bardzo wydajną, wielowątkową maszynę, ale **uniknęliśmy bólu głowy związanego z zarządzaniem wątkami**.

Przyjrzyjmy się bliżej, w jaki sposób wyniki biblioteki libuv trafiają do głównej pętli zdarzeń.

Gdy dane stają się dostępne na poziomie interfejsu gniazda sieciowego lub strumienia, nie możemy od razu wykonać funkcji wywołania zwrotnego. JavaScript jest jednowątkowy, więc wyniki trzeba synchronizować. Nie możemy nagle zmienić stanu w środku cyklu pętli zdarzeń — doprowadziłoby to do klasycznych problemów trapiących aplikacje wielowątkowe, takich jak wyścig, konflikty w dostępie do pamięci itp.

Przy wejściu do pętli zdarzeń Node wykonuje (w praktyce) kopię aktualnej kolejki instrukcji (nazywanej **stosem**), czyści oryginalną kolejkę i uruchamia jej kopię. Przetwarzanie tej kolejki instrukcji nazywa się **cyklem**. Jeśli libuv otrzyma wyniki asynchronicznie, gdy zbiór instrukcji skopiowany na początku cyklu jest w trakcie realizacji przez główny wątek (V8), wynik (otoczony wywołaniem zwrotnym) jest kolejgowany. Gdy aktualna kolejka zostanie wyczyszczona i zakończy się jej ostatnia instrukcja, kolejka będzie sprawdzana pod kątem instrukcji do wykonania w **następnym cyklu**. Wzorec sprawdzania i wykonywania kolejki będzie się powtarzał (w pętli) aż do wyczyszczenia kolejki zadań. W takiej sytuacji nie będą już przyjmowane żadne dodatkowe dane i proces Node po prostu się zakończy.

Dostępna pod adresem <https://github.com/nodejs/node-v0.x-archive/issues/5798> dyskusja między głównymi programistami Node na temat implementacji `process.nextTick` i `setImmediate` zawiera wiele bardzo dokładnych informacji na temat działania pętli zdarzeń.

Oto lista rodzajów zdarzeń wejścia-wyjścia, które trafiają do kolejki.

- **Bloki wykonawcze** — to bloki kodu JavaScript zawierające właściwy program Node; są to funkcje, wyrażenia, pętle itp. Dotyczy to również zdarzeń `EventEmitter` zgłoszonych w aktualnym kontekście wykonywania.
- **Opóźnienia czasowe** — to funkcje wywołania zwrotnego wskazane do wykonania w przyszłości po czasie określonym w milisekundach za pomocą funkcji takich jak `setTimeout` i `setInterval`.
- **Wejście-wyjście** — przygotowane wywołania zwrotne zwracane do głównego wątku po ich wcześniejszym oddelegowaniu do puli wątków zarządzanych przez Node. Dotyczą one obsługi plików i ruchu sieciowego.
- **Opóźnione bloki wykonawcze** — to najczęściej funkcje umieszczone na stosie zgodnie z zasadami wyznaczonymi przez funkcje `setImmediate` i `process.nextTick`.

Warto pamiętać o dwóch ważnych kwestiach.

- Nie uruchamia się i nie zatrzymuje pętli zdarzeń. Pętla zaczyna działać w momencie uruchomienia procesu i kończy się, gdy nie ma już żadnych wywołań zwrotnych do obsłużenia. Oznacza to, że pętla może działać wiecznie.
- Pętla zdarzeń wykonuje się w jednym wątku, ale operacje wejścia-wyjścia deleguje do biblioteki libuv, która zarządza pulą wątków, wykonuje działania równolegle i powiadamia pętlę zdarzeń o ich zrealizowaniu lub otrzymaniu części wyników. Uzyskuje się w ten sposób bezpieczeństwo programowania jednowątkowego przy jednoczesnym wykorzystaniu wielu zalet systemu wielowątkowego.

Aby dowiedzieć się więcej na temat tego, w jaki sposób Node korzysta z biblioteki `libuv` i innych głównych bibliotek, przejrzyj kod modułu `fs` dostępny pod adresem <https://github.com/nodejs/node/blob/master/lib/fs.js>. Porównaj metody `fs.read` i `fs.readSync`, aby poznać różnicę w obsłudze wersji synchronicznej i asynchronicznej. Zwróć też uwagę na wywołanie zwrotne `callback` przekazywane do metody `binding.read` w metodzie `fs.read`.

Aby jeszcze dokładniej przyjrzeć się, jak został zaprojektowany Node, włączając w to implementację kolejki, przejrzyj jego kod źródłowy dostępny pod adresem <https://github.com/nodejs/node/tree/master/src>. Prześledź `MakeCallback` w plikach `fs_event_wrap.cc` i `node.cc`. Sprawdź też klasę `req_wrap`, otoczkę dla silnika JavaScript V8, która znajduje się w pliku `node_file.cc`, a jej definicja w pliku `req_wrap.h`.

Wpływ sposobu zaprojektowania Node na architektów systemów

Node to nadal stosunkowo nowa technologia. Przekroczyła wersję 1.0 już jakiś czas temu i cały czas się rozwija. Błędy bezpieczeństwa są znajdowane i poprawiane. Wycieki pamięci są znajdowane i naprawiane. Eran Hammer, o którym wspomniałem na początku rozdziału, a także cały jego zespół w Walmart Labs, aktywnie uczestniczą w rozwoju Node, szczególnie jeśli znajdują błędy! Dotyczy to również wielu innych dużych firm, które wykorzystują Node, np. firmy PayPal.

Jeśli wybrałeś Node, a Twoja aplikacja osiągnęła rozmiar, który powoduje chęć sięgnięcia po książkę na temat wdrażania Node, masz możliwość nie tylko skorzystania z wiedzy społeczności, ale również samodzielnego stworzenia odpowiedniego środowiska. Node to projekt otwartego oprogramowania, więc możesz zgłaszać własne poprawki i udoskonalenia.

Poza zdarzeniami istnieją jeszcze dwa inne kluczowe elementy, których zrozumienie niezbędne jest do opanowania Node na nieco głębszym poziomie — budowanie systemów z mniejszych części i wykorzystanie zdarzeniowych wersji strumieni wraz z przekazywaniem między nimi danych.

Budowanie większych systemów z mniejszych

W swojej książce *UNIX. Sztuka programowania* Eric Raymond zaproponował **regulę modularności**:

Jedynym sposobem pisania skomplikowanego oprogramowania, które nie będzie się spektakularnie wykladać, jest ograniczenie jego ogólnej złożoności — budowanie programów z prostych części połączonych dobrze zdefiniowanymi interfejsami, aby większość problemów miała charakter lokalny i była nadzieja na to, że uda się ulepszyć jedną z części bez psucia całości.

Pomysł budowania złożonych systemów z „prostych, luźno połączonych części” napotyka się w teorii zarządzania, teorii rządzenia, przemyśle i wielu innych kontekstach. W aspekcie tworzenia oprogramowania zaleca się programistom tworzenie i stosowanie jedynie najprostszych, najbardziej użytecznych komponentów niezbędnych do poprawnego działania w większym systemie. Duże systemy znacznie trudniej analizować, szczególnie jeśli granice poszczególnych komponentów nie są dobrze zarysowane.

Jedną z podstawowych trudności w trakcie konstruowania skalowalnych programów JavaScript jest brak standardowego interfejsu do składania większego programu z mniejszych części. Na przykład typowa aplikacja internetowa może wczytywać zależności za pomocą zestawu znaczników `<script>` w sekcji `<head>` dokumentu **HTML** (ang. *HyperText Markup Language*):

```
<head>
  <script src="plikA.js"></script>
  <script src="plikB.js"></script>
</head>
```

Tego rodzaju system ma wiele wad.

- Wszystkie potencjalne zależności trzeba wcześniej zadeklarować — dynamiczne dodawanie zależności wymaga dodatkowych *sztuczek*.
- Poszczególne skrypty nie są odpowiednio zahermetyzowane — nic nie zablokuje sytuacji, w której kilka plików zmieni jakiś globalny obiekt. Przestrzenie nazw mogą łatwo ze sobą kolidować, co czyni dowolne wstrzykiwanie kodu dosyć niebezpiecznym.
- Plik *plikA.js* nie może skorzystać z *plikB.js* jako kolekcji, czyli na przykład użyć zapisu kontekstowego typu `plikB.metoda`.
- Rozwiązanie ze znacznikiem `<script>` powstało, zanim zaczęto tak naprawdę projektować systemy modułowe, więc nie wie nic o zależnościach i wersjonowaniu.
- Skryptów nie można łatwo usunąć lub nadpisać.
- Z powodu wszystkich wymienionych zagrożeń współdzielenie nie jest tak łatwe, jak mogłoby być, co utrudnia współpracę w otwartym ekosystemie.

Ambiwalentne wstawianie w aplikacji nieprzewidywalnych fragmentów kodu przeczy chęci zapewniania przewidywalnej funkcjonalności. Tak naprawdę potrzebny jest pewien standard wczytywania i współdzielenia niezależnych modułów programu.

W odpowiedzi na to wyzwanie Node wprowadziło pojęcie **pakietu**, który bazuje na specyfikacji CommonJS. Pakiet to zbiór plików z kodem połączonych w jedną całość za pomocą pliku manifestu opisującego cały zbiór. Zależności, autorstwo, cel, struktura i inne istotne dane znajdują się w pliku manifestu w ustandaryzowanym formacie. Zachęca to do tworzenia dużych systemów z wielu małych, niezależnych podsystemów. Co ważniejsze, zachęca do współdzielenia się kodem.

*To, co tu opisuję, nie jest problemem technicznym.
To kwestia zbierania się ludzi razem i podejmowania decyzji,
które inicjują postęp i wspólne budowanie czegoś większego i lepszego.*

— Kevin Dangoor, twórca CommonJS

W dużej mierze sukces Node wynika z ogromnej liczby i nierzadko wysokiej jakości pakietów udostępnianych przez społeczność Node za pomocą standardowego systemu zarządzania pakietami — **npm**. Ten system sprawił, że JavaScript stał się jednym z profesjonalnych języków.

Wprowadzenie do npm dla wszystkich osób, które dopiero poznają tajniki Node, znajduje się pod adresem <https://docs.npmjs.com/>.

Strumienie

W swojej książce *Język C++ . Kompendium wiedzy. Wydanie IV* Bjarne Stroustrup wyjaśnia:

Projektowanie i implementacja ogólnego mechanizmu wejścia-wyjścia dla języka programowania jest notorycznie trudne. (...) Mechanizm wejścia-wyjścia powinien być łatwy, wygodny i bezpieczny w użyciu, wydajny i elastyczny, a co najważniejsze, powinien być kompletny.

Nie powinno nikogo dziwić, że zespół projektujący Node bardzo mocno skupił się na zapewnieniu wydajnego i łatwego w użyciu wejścia-wyjścia. Poprzez symetryczny i prosty interfejs, który obsługuje bufor danych i zdarzenia strumieni, aby nie musiał się tym zajmować programista, moduł Stream z Node to zalecany sposób zarządzania synchronicznymi strumieniami danych zarówno w modułach wewnętrznych, jak i w modułach, które programista tworzy samodzielnie.

Bardzo dobre wprowadzenie do modułu Stream znajduje się pod adresem <https://github.com/substack/stream-handbook>. Dokumentacja Node dotycząca tego tematu znajduje się pod adresem <https://nodejs.org/api/stream.html>.

Strumień w Node to po prostu ciąg bajtów lub jak kto woli, ciąg znaków. Na każdym etapie strumień zawiera bufor bajtów, a ten bufor może mieć długość zero lub więcej bajtów.

Ponieważ każdy znak w strumieniu jest dobrze zdefiniowany i ponieważ każdy typ danych cyfrowych można wyrazić bajtami, każdą część strumienia można przekierować lub **przesłać potokiem** do dowolnego innego strumienia. Poszczególne fragmenty strumienia mogą trafić do różnych obsługujących je funkcji. W ten sposób interfejs wejściowy i wyjściowy strumienia jest zarówno elastyczny, jak i przewidywalny, a co równie ważne, daje się łatwo łączyć w większe całości.

Poza zdarzeniami system Node znany jest właśnie z bardzo intensywnego wykorzystania strumieni. Bazując na pomysłach budowania aplikacji z wielu mniejszych procesów emitujących zdarzenia lub reagujących na nie, kilka modułów wejścia-wyjścia z Node wykorzystuje właśnie strumienie. Gniazda sieciowe, odczyt i zapis plików, obsługa standardowego wyjścia i wejścia, obsługa Zlib itp. to wszystko producenci lub konsumenci danych, których można łatwo połączyć w większą całość dzięki abstrakcyjnemu interfejsowi Stream. Zauważ tu bardzo duże podobieństwo do potoków Unix.

Pięć różnych klas bazowych wykorzystuje abstrakcyjny interfejs `Stream: Readable, Writable, Duplex, Transform` i `PassThrough`. Każda z klas bazowych dziedziczy po `EventEmitter`, czyli interfejsie umożliwiającym zgłaszanie zdarzeń i podłączanie procedur ich obsługi. Strumienie w Node to strumienie zdarzeniowe, a przesył danych między procesami odbywa się właśnie za pomocą strumieni. Ponieważ strumienie można łatwo łączyć w łańcuchy działań, stanowią jedno z podstawowych narzędzi programisty Node.

Warto, abyś jako programista wiedział, czym są strumienie i jak zostały zaimplementowane w Node, ponieważ będziemy z nich bardzo intensywnie korzystać w dalszej części książki.

Wykorzystanie JavaScript na każdym etapie projektu maksymalizuje dobre efekty

JavaScript stał się językiem wszechstronnym. Oczywiście istnieją systemy wykonawcze języka w każdej przeglądarce internetowej. V8, interpreter JavaScript wykorzystywany przez Node, to ten sam interpreter, który znajduje się w przeglądarce Chrome firmy Google. Sam język poszedł nawet dalej i obejmuje obecnie swoim zasięgiem zarówno warstwę kliencką, jak i warstwę serwerową stosu oprogramowania. JavaScriptu używa się do odpytывania bazy danych CouchDB, do realizacji operacji odwzorowanie-redukcja w MongoDB, a nawet do znajdowania kolekcji w Elasticsearch. Bardzo popularny format danych **JSON** (ang. *JavaScript Object Notation*) tak naprawdę reprezentuje dane jako obiekty JavaScript.

Gdy w jednej aplikacji używa się różnych języków, wzrasta koszt **przełączania kontekstu**. Jeśli system składa się z części opisywanych różnymi językami, architekturę systemu znacznie trudniej opisać, zrozumieć i rozszerzyć. Jeśli poszczególne części **mówią** inaczej, potrzebne są kosztowne mechanizmy tłumaczące.

Nieefektywność zrozumienia wszystkich elementów prowadzi do zwiększenia kosztów i powstawania bardziej wrażliwych systemów. Członkowie zespołu programistycznego muszą dobrze znać wszystkie języki lub muszą być podzieleni zadaniowo, co zwiększa koszty szkoleń lub znalezienia dobrych programistów. Gdy szczegóły działania istotnej części systemu znane są tylko kilku inżynierom, zmniejszy się poziom współpracy w zespole, co zwiększy koszt aktualizacji i uczyni je trudniejszymi, a także podatniejszymi na błędy.

Jakie nowe możliwości mogą się pojawić, gdy te trudności uda się zredukować, a nawet wyeliminować?

Zamiana kodu

Ponieważ część kliencka i serwerowa mówi tym samym językiem, mogą współdzielić pewne części kodu. Jeśli budujesz aplikację internetową, otwiera to bardzo interesujące (i unikatowe) możliwości.

Rozważmy aplikację, która umożliwia klientowi wykonywanie zmian w innym środowisku. Narzędzie to daje programiście możliwość zmiany kodu JavaScript napędzającego witrynę internetową i pozwala na zobaczenie tej zmiany przez inne klienty w czasie rzeczywistym. Aplikacja musi jednak w jakiś sposób zamienić kod działający w wielu różnych przeglądarkach. Jednym ze sposobów jest zawarcie zmiany w funkcji przekształcającej, przekazanie tej funkcji do wszystkich podłączonych klientów i wykonanie jej w lokalnym środowisku, aby uzyskać wersję **kanoniczną**. Gdy aplikacja źródłowa ewoluuje, emituje **ogólną** aktualizację w postaci kodu JavaScript, a wszystkie klienty także odpowiednio ewoluują. Wykorzystamy tego rodzaju technologię w rozdziale 7.

Ponieważ Node korzysta z języka JavaScript, serwer jest w stanie z własnej inicjatywy podjąć odpowiednie akcje. Sieć może rozgłosić kod, który muszą wykonać klienty. Podobnie to klient może przesłać kod do wykonania przez serwer. Łatwo się domyślić, że w ten sposób dopuszczamy dynamiczną zamianę kodu — proces Node wysyła unikatowy pakiet z kodem JavaScript do wykonania przez konkretnego klienta.

Gdy **zdalne wywoływanie procedur** (RPC — ang. *Remote Procedure Calls*) nie wymaga już pośrednika, który tłumaczy dane do właściwego kontekstu, kod może istnieć w sieci tak długo lub tak krótko, jak to jest niezbędne. Może być wykonywany w wielu kontekstach dobieranych na podstawie równoważenia obciążenia, dostępnych danych, mocy obliczeniowej, położenia geograficznego itp.

Browserify

JavaScript to język wspólny dla Node i przeglądarki. Jednakże Node znacząco rozszerza język JavaScript, oferując wiele nowych poleceń i konstrukcji, których nie ma po stronie klienckiej. Na przykład w standardowym JavaScript nie ma odpowiednika modułu `Stream` z Node.

Co więcej, repozytorium npm cały czas się powiększa i obecnie zawiera już ponad 400 tysięcy pakietów Node. Wiele z tych pakietów może być wykorzystywanych po stronie klienckiej i po stronie serwerowej. Rozwój JavaScript po stronie serwerowej spowodował, że zaczęło powstawać coraz więcej bibliotek i modułów o bardzo dobrej jakości, dostępnych dla każdego środowiska.

Browserify to projekt, który powstał, aby ułatwić współdzielenie modułów npm i głównych modułów Node z kodem klienckim. Gdy taki moduł podda się konwersji, można go zaimportować w środowisku przeglądarki zwykłym znacznikiem `<script>`. Instalacja pakietu Browserify jest bardzo prosta:

```
npm install -g browserify
```

Wykonajmy prosty przykład. Utwórz plik *math.js* i napisz kod tak, jakby to był moduł npm:

```
module.exports = function() {
  this.add = function(a, b) {
    return a + b;
  }
  this.subtract = function(a, b) {
    return a - b;
  }
};
```

Następnie napisz program *add.js*, który używa nowego modułu:

```
var Math = require('./math.js');
var math = new Math;

console.log(math.add(1,3)); // 4
```

Wykonanie tego programu w wierszu poleceń za pomocą Node (`node add.js`) spowoduje zwrócenie wyniku 4 w oknie terminalu. Co zrobić, gdybyśmy chcieli teraz wykorzystać ten sam moduł w przeglądarce? Po stronie klienckiej nie ma instrukcji `require`, więc musimy skorzystać z narzędzia Browserify:

```
browserify math.js -o bundle.js
```

Narzędzie przejdzie przez kod, znajdzie instrukcje `require` i automatycznie zbierze wszystkie zależności (a także zależności tych zależności) w jeden plik, który będzie można wczytać w aplikacji klienckiej kodem:

```
<script src="bundle.js"></script>
```

Jako dodatkowy bonus tak wykonany plik automatycznie wprowadzi do środowiska przeglądarki użyteczne zmienne globalne znane z Node, takie jak `__filename`, `__dirname`, `process`, `Buffer` i `global`. Oznacza to między innymi, że w przeglądarce uzyskamy dostęp do metody `process.nextTick`.

Twórca Browserify, James Halliday, to osoba aktywnie działająca w społeczności Node. Jego prace znajdziesz pod adresem <https://github.com/substack>. Istnieje nawet narzędzie dostępne pod adresem <http://requirebin.com>, które umożliwia testowanie online modułów npm po ich konwersji. Pełną dokumentację narzędzia znajdziesz na stronie <https://github.com/substack/node-browserify#usage>.

Podsumowanie

W tym rozdziale wykonaliśmy bardzo szybki kurs podstaw działania Node. Dowiedziałeś się, dlaczego system Node został zaprojektowany w taki sposób i dlaczego środowisko sterowane zdarzeniami to dobre rozwiązanie dla nowoczesnego oprogramowania sieciowego. Po wyjaśnieniu

pętli zdarzeń, a także podstawowych aspektów współbieżności i zrównoleglenia przeszedłem do opisu jednej z głównych filozofii Node dotyczącej tworzenia oprogramowania, czyli budowania systemu z **małych, luźno powiązanych klocków**. Wyjaśniłem również, jakie zalety daje stosowanie tego samego języka po stronie serwerowej i klienckiej, a także jakie nowe możliwości otwiera.

Masz już teraz ogólne pojęcie na temat aplikacji, które będziemy tworzyć. Pomoże Ci to w trakcie budowania i konserwacji aplikacji Node, bo pozwoli zwrócić odpowiednią uwagę na unikatowe cechy tego systemu. W następnym rozdziale przejdziemy od razu do budowania serwerów za pomocą Node, sposobów ich hostingu, a także podstawowych metod tworzenia z nich paczek i ich dystrybuowania.

Skorowidz

A

adres URL, 40
 aplikacji, 47, 48
Akamai, 125
algorytm karuzelowy ważony, 84
Ansible, 205, 206, 207
 struktura folderów, 206
 zbiór strategii, 205, 206
aplikacja, 34
 dane uwierzytelniające, 49
 dług techniczny, *Patrz:* dług techniczny
 monitorowanie, 134, 152, 154, 156
 skalowalna, *Patrz:* skalowalność
 telekonferencyjna, 91
 uruchamianie, 48
 wdrażanie, 194
 automatyzacja, 193, 194
 wdrożenie, 47, 52
 zależności, 59
 zmienna
 konfiguracyjna, 49
asercja, 175, 177, 178
 tworzenie, 176

B

baza
 klucz-wartość, 109
 MongoDB, 46
bezpieczeństwo, 42
biblioteka, 23, 30
 asercji, 177
 Chai, *Patrz:* Chai
 Express, 57
 libuv, 27, 28, 29, 30, 64

 Mocha, *Patrz:* Mocha
 pamięci podręcznej, 122
 Sinon, *Patrz:* Sinon
blokada, 23
blok wykonawczy, 29
błąd, 99, 134, 153
 innego procesu, 137
 kod, 138
 śledzenie, 135
 Unexpected Token, 118
 wylapywanie, 139
Browserify, 34, 35, 160, 167, 168
BrowserSync, 160, 169
Brunch, 171
Bunyan, 144

C

cache, *Patrz:* pamięć podręczna
CasperJS, 189
Caswell Tim, 81
CDN, 125
certyfikat, 43
Chai, 180
obiekt, 71
chmura
 DigitalOcean, *Patrz:* DigitalOcean
 Heroku, *Patrz:* Heroku
 OpenShift, *Patrz:* OpenShift
ciąg znaków, 32, *Patrz też:* strumień
ciągła integracja, 208
ciągle wdrażanie, 209
CloudFlare, 125
Content Delivery Network, *Patrz:* CDN
cookie, 39, 127
cykl, 29

D

Dahl Ryan, 18
 dane
 przechwytywanie, 39
 sesyjne, 39, 121
 wejścia-wyjścia, 25
 wrażliwe, 49
 datagram, 91, 92, 93
 debugger, 172, 174
 DigitalOcean, 79
 dług techniczny, 159
 Docker, 54, 55, 56
 instalowanie, 56, 57
 kontener, 55, 57, 58
 wirtualny, 58, 60
 obraz, 55, 57, 58
 tworzenie, 59
 rejestr, 55
 droplet, 79
 drzewo zależności, 216, 222
 dziennik zdarzeń, 140

E

element synchronizujący, 23
 Express, 143, 195, 198

F

folder
 ./src, 58, 59
 node_modules, 58
 framework Express, *Patrz:* Express
 funkcja, 120
 after, 179
 afterEach, 179
 before, 179
 beforeEach, 179
 exec, 65, 70
 execFile, 65, 71
 fork, 65, 68, 70, 74, 137, 139
 log, 141, 142
 net.connect, 145
 net.createServer, 145
 optymalizowana, 151
 podmiana, 183, 184, 185
 spawn, 65, 66, 68, 139

G

Git, 46
 GitHub, 194, 210
 Gulp, 160, 161, 162, 171, 198, 222
 Guvnor, 156

H

Haines Steve, 63
 Halliday James, 35
 Hammer Eran, 17, 18, 30
 Handlebars, 160
 HAProxy, 85
 Heroku, 44, 49, 156, 194, 213
 dyno, 44, 48, 50
 Heroku Cli, 45, 47
 Holliday James, 87
 HyperLogLog, 110, 114

I

instrukcja
 describe, 170
 it, 170
 require, 165, 167
 interfejs
 EventEmitter, 33
 pub-sub, 99
 Stream, 33
 użytkownika, *Patrz:* UI

J

Jasmine, 178
 JavaScript, 33, 34, 68, 105, 107, 120, 160
 interpreter, 116
 kompilacja, 117
 optymalizacja, 116
 Jenkins, 209, 214
 konfigurowanie, 210, 211, 212, 214, 215
 język
 CoffeeScript, 165, 170
 programowania
 bazujący na prototypach, 107, 108
 Go, 20
 JavaScript, *Patrz:* JavaScript
 obiektyw, 107
 szablonów, 160

JSON Web Token, *Patrz:* JWT
JWT, 128, 129

K

kartridż, 51, 52
klasa, 107
 bazowa, 33
 Duplex, 33
 EventEmitter, 65
 instancja, 107, 108
 PassThrough, 33
 Readable, 33
 Transform, 33
 Writable, 33
klucz, 109, 110, 112
 licencyjny New Relic, 156, 157
 prywatny, 43, 214
 publiczny, 214
 routingu, 89
 usuwanie, 124
 wygasanie, 124
kod
 asynchroniczny, *Patrz:* proces asynchroniczny
 debugowanie, 172
 hermetyzacja, 216
 optymalizacja, 161
 synchroniczny, *Patrz:* proces synchroniczny
 śledzący, 39
 wersja, 156, 201
 znormalizowany, 168
kolejka
 implementacja, 30
 komunikatów, *Patrz:* komunikat kolejka
komunikat, 100
 delegacja, 108
 JSON, 102
 kolejka, 87, 88, 89
 nasłuchiwanie, 88
 wymiana, 89
kontekst, 207
 niszczenie, 137
 przełączanie, 33, 34
 tworzenie, 135
 usuwanie, 136

L

liczba binarna, 110
licznik, 114
limit czasowy, 39

M

mapa źródłowa, 166
maska, 112
maszyna wirtualna, 56, 77, 201
mechanizm, 51, 52
 webhook, *Patrz:* webhook
menedżer procesów, 152
metoda
 assert, 176
 assert.deepEqual, 176
 assert.doesNotThrow, 177
 assert.equal, 176
 assert.fail, 177
 assert.ifError, 176
 assert.notDeepEqual, 176
 assert.notEqual, 176
 assert.notStrictEqual, 176
 assert.ok, 176
 assert.strictEqual, 176
 assert.throws, 176
 atrapa, 186
 bind, 88, 91, 136
 Caller.parseReponse, 185
 capitalize, 186
 clear, 123
 cluster, 73
 createServer, 38
 dest, 163
 disconnect, 72, 75, 76
 domain.dispose, 137
 domain.enter, 137
 domain.exit, 137
 fork, 75
 get, 122
 heapUsed, 148
 http.get, 184
 intercept, 137
 kill, 72, 75, 76
 Object.create, 108
 pipe, 163
 porównująca, 176
 process.memoryUsage, 150

metoda
 psubscribe, 100, 102
 remove, 123, 136
 scan, 124
 send, 72, 76, 92
 setMulticastTTL, 93
 setupMaster, 74, 76
 socket.address, 92
 src, 163
 subscribe, 100
 task, 163
 watch, 170
 mikroserwis, 98, 101, 102
 architektura, 98
 Mimosa, 171
 Mocha, 170, 175, 177, 178, 179, 222
 moduł, 23
 assert, 170, 172, 175, 176, 177
 bouncy, 87
 child_process, 64, 65, 137, 152
 cluster, 73, 74, 152
 dgram, 93
 dodawanie, 193
 domain, 135
 fs, 30
 github, 200
 handlebars, 165
 http, 143
 HTTPS, 42
 jquery, 165
 npm, 34
 page, 188
 pidusage, 150
 pm2, 154
 request, 40
 sinon-chai, 181
 tick, 150, 151
 monitoring zdalny, 146, 147, 148
 Morgan, 143

N

narzędzie, *Patrz też:* polecenie
 browser-sync, 169
 curl, 39
 HAProxy, *Patrz:* HAProxy
 NVM, 81
 rhc, 51
 StatsD, 140

New Relic, 156, 157
 Nginx, 77, 79, 80
 dyrektywa
 backup, 84
 fail_timeout, 84
 ip_hash, 85
 least_conn, 85
 max_fails, 84
 weight, 84
 instalowanie, 80
 konfiguracja, 82
 Node, 17, 34
 NVM, 81

O

obiekt, 119, 120
 adresu, 75
 ChildProcess, 64, 65
 ClientRequest, 143
 ClientResponse, 143
 cluster, 75
 EventEmitter, 24, 25
 głęboko równy, 176
 process.env, 53
 Promise, 122, 189
 roboczy, 75, 76
 subscriber, 100
 OpenShift, 51, 52
 Cartridge, 51, *Patrz też:* kartridż
 Gear, 51, *Patrz też:* mechanizm
 operacja
 na bitach, 110, 112, 113
 wejścia-wyjścia, 18, 26, 27, 29, 32, 44, 68
 optymalizacja, 19
 operator
 identyczności, 175, 176
 równości, 175, 176
 opóźnienie czasowe, 29

P

pakiet, 23, 216
 newrelic, 157
 usuwanie, 220
 wdrażanie, 218
 wydanie zapoznawcze, 218
 zarządzanie, 216, 219

- pamięć, 105
 - optymalizacja, 105, 106, 108, 109
 - podręczna, 121, 124
 - biblioteka, 122
 - profilowanie, 150
 - współdzielona, 22
 - wyciek, 30
 - zużycie, 146, 149, 156
- pętla zdarzeń, 26, 27, 28
 - zatrzymanie, 29
- PhantomJS, 187, 188, 189
- Pike Rob, 20
- pilot zdalnego sterowania, 144
- plik
 - .bashrc, 81
 - .profile, 81
 - app.js, 168
 - cookie, *Patrz:* cookie
 - deskryptor, 67
 - Dockerfile, 58
 - dummy.log, 150
 - gulpfile.js, 161, 162
 - logreader.js, 150
 - odczyt, 26
 - out.log, 143
 - package.json, 45, 57, 219, 220
 - Procfile, 45, 48
 - provision.sh, 203, 205
 - server.js, 45, 46, 48, 59
 - v8.log, 150, 151
 - Vagrantfile, 202, 204
- PM2, 152, 153, 154, 155, 198
 - monitorowanie procesów, 154
 - śledzenia działania zarządzanych skryptów, 155
- podproces, 64, 65, 152
- polecenie, *Patrz też:* narzędzie
 - .load, 146
 - apt-get, 79, 80
 - backtrace, 173
 - bitcount, 111
 - cat, 153
 - clearBreakpoint, 173
 - cont, 172, 173
 - debugger, 172
 - delete, 54
 - docker build, 59
 - docker ps, 60
 - docker run, 60
 - expire, 123
 - force-stop, 54
 - git push, 199
 - gulp, 162
 - heroku logs, 50
 - heroku ps, 50
 - heroku releases, 50
 - htop, 65
 - kill, 173
 - listen, 75
 - ls, 56, 64, 139, 219
 - lsof, 56
 - multi, 111
 - netstat, 56
 - next, 172
 - node programmatic.js, 154
 - npm dedupe, 221
 - npm install, 165
 - npm ls, 219
 - npm prune, 220
 - npm shrinkwrap, 223
 - out, 173
 - pause, 173
 - PFADD, 115
 - PFMERGE, 116
 - pm2 delete, 153
 - pm2 flush, 154
 - pm2 info, 153
 - pm2 list, 153, 155
 - pm2 logs, 154
 - pm2 restart, 153
 - pm2 stop, 153
 - pm2 web, 155
 - powłoki, 207
 - process.send, 138
 - ps aux, 65
 - quit, 173
 - reduce, 111
 - reload, 54
 - restart, 54, 173
 - rhc, 53, 54
 - rhc setup, 51
 - run, 173
 - scripts, 173
 - setBreakpoint, 173
 - show, 54
 - start, 54
 - step, 172
 - stop, 54
 - sudo, 79, 80, 203
 - throw, 138
 - tidy, 54

polecenie
 top, 65
 try-catch, 134
 update, 79
 vagrant up, 203
 version, 173
 yum update, 79
 połączenie, 39
 potok, 32
 problem
 C10K, 64
 miejsc parkingowych, 77
 optymalizacji gniazd sieciowych, *Patrz:*
 problem C10K
 procedura zdalne wywoływanie, *Patrz:* RPC
 proces, 21, 22
 asynchroniczny, 161, 177, 179
 główny, 73, 75
 identyfikator, 56
 ls, 65
 migawka, *Patrz:* migawka
 monitorowanie, *Patrz:* aplikacja
 monitorowanie
 niezależny, 64, 73
 ochrona, 24
 ograniczenia systemowe, 65
 potomny, 67, 69, 71, 75
 tworzenie, 74
 zatrzymanie, 72
 profilowanie, 149
 roboczy, 50, 73, 75
 skalowanie, 50
 synchroniczny, 106, 162, 179
 typ, 48
 uruchamianie, 153
 procesor wielordzeniowy, 65
 programowanie wykorzystujące zdarzenia, 27
 protokół
 AMQP, 87
 bezstanowy, 38
 HTTP, 38, 127
 HTTPS, 42, 43
 SSL, 42
 TCP, 145
 TLS, 42
 UDP, 91, 140, 141
 przeglądarka procesów, 56
 przestrzeń nazw, 122

R

RabbitMQ, 87, 88
 Ranney Matt, 100, 111
 Raymond Eric, 30
 Redis, 99, 100, 109, 121, 144
 multi server, 111
 refaktoring, 159
 reguła prostoty, 23
 Remote Procedure Calls, *Patrz:* RPC
 REPL, 144, 147, 149, 174
 Rogers Mikeal, 40
 RPC, 34

S

semafor, 23
 Semver, *Patrz:* wersjonowanie semantyczne
 Seneca, 102
 serwer, 37
 Apache, 55, 143
 autopilot-dev, 198
 droplet, *Patrz:* droplet
 HTTP, 38, 39, 40, 44, 57
 http-proxy, 85
 HTTPS, 43
 monitorowanie, 156
 Nginx, *Patrz:* Nginx
 pośredniczący, 77, 78, 85, 87
 odwrotny, 78
 Redis, *Patrz:* Redis
 równoważenie obciążenia, 80, 81, 82, 84, 85,
 87
 tworzenie, 38
 UDP, 91
 WWW, 42, 143
 zabezpieczony, 40
 sesja użytkownika, 127
 metoda, 75
 Should, 178
 sieć
 prywatna, 147
 społecznościowa, 78
 Sinon, 178, 181
 atrapa, 181, 185, 186
 podmiana, 181, 183, 184, 185
 szpieg, 181, 182, 183, 184

skalowalność, 63, 99
 pionowa, 64, 73, 102
 pozioma, 64, 77
 SlimerJS, 189
 stos, 29
 Stroustrup Bjarne, 32
 strumień, 32
 przekierowanie, 32
 Readable, 40
 Writable, 40
 system
 budowania, 160, 161, 162, 170, 200
 strumieniowy, 163
 tworzenie, 163, 171
 ciągłego dostarczania, 208
 ciągłego wdrażania, 208, 209
 ciągłej integracji, 208
 definiowania testów, 170
 JWT, *Patrz:* JWT
 kolejek, 87
 konfiguracja, 205
 monitorujący, 146, 147, 148, 156
 prototypowy, *Patrz:* język programowania
 bazujący na prototypach
 rozproszony, 63, 64
 skalowanie, 98
 tworzenia paczek, 160
 wdrażania, 198
 wejścia-wyjścia, 37, 44
 zarządzania pakietami, 32
 szablon, 55, 198

Ś

środowisko
 deweloperskie, 194, 198, 201
 preprodukcyjne, 208
 produkcyjne, 193, 194, 198, 201
 aktualizacja, 194
 emulacja, 201, 202

T

tablica, 119, 120, 122
 test, 99, 170, 171, 177
 asynchroniczny, 179
 automatyzacja, 187, 189
 funkcjonalny, 208
 konfigurowanie środowiska, 177

logiki biznesowej, 208
 synchroniczny, 179
 token, 127, 128, 143, 144
 typu rzutowanie, 175

U

UI, 44
 usługa, 99
 atrapa, 99
 bezstanowa, 99
 tunelująca ruch, 41
 wyzwalana wzorcem, 102
 uwierzytelnianie, 39, 127
 użytkownik
 identyfikator, 127
 sesja, *Patrz:* sesja użytkownika
 uwierzytelnianie, 127

V

V8, 27, 29, 65, 105, 116, 117, 150, 152
 funkcja wbudowana, 118
 wersja, 116
 Vagrant, 201, 203, 205
 Vows, 178

W

wątek, 21, 22
 bezpieczeństwo, 23, 24
 pula, 29
 synchronizacja, 22
 wejścia-wyjścia, 23
 webhook, 194, 198
 konfigurowanie, 196, 197
 tworzenie adresu, 199
 wersja, 194, 217, 222
 dopuszczalna, 218
 lokalna, 201
 numer, 216
 wersjonowanie semantyczne, 216, 217, 218
 węzeł, 63, 99
 Winston, 144
 współbieżność, 20, 23, 68
 wydajność, 63
 wyjątek, 134
 wyścig, 23, 163
 wywołanie zwrotne, 23, 27, 29
 wzorzec, 102, 163, 171

Y

Yeoman, 171

Z

zadanie systemowe, 18
 zależność, 161, 162, 163, 216
 deklarowanie, 216
 dependencies, 222
 devDependencies, 222
 peerDependencies, 222
 pobieranie, 165
 zdarzenie, 24
 close, 92
 disconnect, 75, 76
 dziennik, *Patrz:* dziennik zdarzeń
 error, 92
 exit, 75, 76
 fork, 75
 listening, 75, 76
 message, 76
 obiektu roboczego, 76
 online, 75, 76
 połączenia, 39
 setup, 75, 76
 sieciowe, 38
 strumienia, 32
 subscribe, 100
 uncaughtException, 134
 wejścia-wyjścia, 29

zmienna

 konfiguracyjna, 49
 środowiskowa, 49
 process.env, 48, 49, 53
 zarządzanie, 49
 warunkowa, 23
 znak
 #, 89
 \$, 151
 %, 118
 *, 89, 151, 217
 ^, 151, 216, 217
 ~, 151, 216, 217
 ==, 175, 176
 ===, 175, 176
 gwiazdki, 89, 151, 217
 karety, 216, 217
 tyldy, 151, 216, 217
 zrównoleglenie, 20, 22, 23, 69, 163

Ż

żądanie, 38
 AJAX, 115
 CONNECT, 41
 GET, 40
 HTTP, 39, 184
 logowanie, 143, 144

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

Node.js

Projektowanie, wdrażanie i utrzymywanie aplikacji

W ciągu kilku ostatnich lat JavaScript stał się niezwykle wszechstronnym i wydajnym językiem programowania. Dla kodujących w nim programistów świetnym narzędziem okazuje się platforma Node.js. Ten framework *open source* został zaprojektowany do tworzenia skalowalnych aplikacji internetowych, jednak umożliwia też tworzenie aplikacji sterowanych zdarzeniami. Użytkownicy docenili jego wygodę. Node.js odniósł spory sukces i stał się kluczowym narzędziem programistycznym w wielu firmach. Mimo to znalezienie wyczerpujących informacji o profesjonalnym projektowaniu, testowaniu i wdrażaniu oprogramowania za jego pomocą jest dość trudne.

Niniejsza książka zawiera opis technik i narzędzi pozwalających na wykonanie w Node.js elastycznej, inteligentnej, trwałej i łatwej w utrzymaniu aplikacji o znakomitej jakości. Poza podstawami zaprezentowano tu również zestaw wzorców ułatwiających rozwiązywanie typowych problemów pojawiających się w dzisiejszych projektach. Nie zabrakło licznych przykładów z życia oraz wskazówek, które doceni każdy, kto musi sprawnie wdrożyć trudny projekt. Książka umożliwia zgłębienie tajników Node.js i naukę projektowania modułowego. Sporo miejsca poświęcono też testowaniu i monitorowaniu aplikacji oraz strategiom utrzymania aplikacji przez większy zespół.

Node.js: dojrzała technologia, znakomita wydajność i wszechstronność!

[PACKT] open source
PUBLISHING community experience distilled

Helion

księgarnia Internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

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

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>

Informatyka w najlepszym wydaniu

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-283-3609-4



9 788328 336094

cena: 54,90 zł

W książce:

- mocne i słabe strony Node.js
- techniki skalowania aplikacji i komunikacja międzyprocesowa
- zarządzanie pamięcią i monitorowanie sesji
- właściwe budowanie potoku tworzenia aplikacji
- konserwacja systemu i strategię zarządzania zależnościami

Sandro Pasquali — programista, przedsiębiorca, twórca firmy technologicznej Simple.com, która sprzedawała pierwszy na świecie framework aplikacyjny oparty na języku JavaScript. Obecnie szkoli zespoły programistów korporacyjnych. W swoim czasie zarządzał projektowaniem wielu aplikacji dla takich firm jak Nintendo, Major League Baseball, LimeWire, AppNexus i Conde Nast, a także dla instytutów badawczych i szkół. Zawsze szuka nowych sposobów na połączenie doskonałości projektowej z innowacyjną technologią.

