

TWÓJ PRZEWODNIK PO PLATFORMIE NODE.JS!

Apress®

Node.js w praktyce

Tworzenie skalowalnych
aplikacji sieciowych

Azat Mardan

Helion 

Tytuł oryginału: Practical Node.js: Building Real-world Scalable Web Apps

Tłumaczenie: Joanna Zatorska

ISBN: 978-83-283-1085-8

Original edition copyright © 2014 by Azat Mardan.
All rights reserved.

Polish edition copyright © 2015 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 jakiejkolwiek 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/nodewp>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

Printed in Poland.

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

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

Spis treści

O autorze	11
O recenzencie technicznym	13
Podziękowania	15
Wstęp	17
Rozdział 1. Instalowanie Node.js i podstawowe zagadnienia	21
Instalowanie Node.js i menedżera NPM	21
Gotowe instalatory	22
Instalowanie za pomocą HomeBrew lub MacPorts	22
Instalowanie z wykorzystaniem pliku tar	23
Instalowanie bez użycia sudo	24
Instalowanie z repozytorium Git	24
Instalacja kilku wersji z wykorzystaniem Nave	25
Instalacja kilku wersji z wykorzystaniem menedżera Node Version Manager (NVM)	26
Alternatywne systemy służące do instalowania kilku wersji	26
Weryfikacja instalacji	26
Konsola Node.js (REPL)	27
Uruchamianie skryptów Node.js	28
Podstawy i składnia Node.js	28
Typowanie słabe	28
Buffer — typ danych Node.js	29
Notacja literałów obiektowych	29
Funkcje	29
Tablice	31
Natura prototypowa	31
Konwencje	32
Zakres globalny w Node.js i słowa zarezerwowane	33
__dirname a process.cwd	35
Funkcje pomocnicze interfejsu programistycznego aplikacji przeglądarki	35

Podstawowe moduły Node.js	36
Przydatne narzędzia Node.js	37
Odczyt i zapisywanie plików z systemu plików w Node.js	38
Strumieniowe przesyłanie danych w Node.js	38
Instalowanie modułów Node.js za pomocą NPM	38
Poskramianie wywołań zwrotnych w Node.js	39
Tworzenie serwera „Witaj, świecie” za pomocą modułu HTTP Node.js	40
Debugowanie programów Node.js	41
Debugger Node.js	41
Debugowanie za pomocą narzędzia Node Inspector	42
Środowiska IDE dla Node.js oraz edytory kodu	44
Obserwowanie zmian w plikach	46
Podsumowanie	48
Rozdział 2. Wykorzystanie Express.js 4 do tworzenia aplikacji WWW Node.js	49
Czym jest Express.js?	49
Jak działa Express.js?	52
Instalowanie Express.js	52
Wersja Express.js	53
Express.js Generator	53
Lokalna instalacja Express.js	54
Tworzenie szkieletu aplikacji z wykorzystaniem Express.js	56
Interfejs wiersza poleceń Express.js	57
Trasy w Express.js	59
Oprogramowanie pośredniczące jako kręgosłup Express.js	60
Konfiguracja aplikacji Express.js	60
Jade to Haml w Express.js (Node.js)	60
Podsumowanie tworzenia szkieletów aplikacji	61
Omówienie projektu Blog	61
Przesyłanie danych	62
Przykład „Witaj, świecie” w Express.js	64
Tworzenie folderów	64
Inicjalizacja NPM i plik package.json	65
Deklarowanie zależności — npm install	66
Plik app.js	67
Spotkanie z Jade — jeden szablon, by wszystkimi rządzić	70
Uruchamianie aplikacji „Witaj, świecie”	71
Podsumowanie	71
Rozdział 3. Wykorzystanie technik TDD i BDD w Node.js z użyciem platformy Mocha	73
Instalowanie i zrozumienie platformy Mocha	73
Zrozumieć haki platformy Mocha	75
TDD z wykorzystaniem modułu assert	76
Moduł Chai assert	78
BDD z wykorzystaniem Expect.js	79
Składnia Expect.js	80

Projekt — pisanie pierwszego testu BDD dla aplikacji Blog	80
Umieszczanie konfiguracji w pliku Makefile	82
Podsumowanie	84
Rozdział 4. Silniki szablonów — Jade i Handlebars	85
Składnia i funkcje Jade	85
Znaczniki	86
Zmienne (Locals)	86
Atrybuty	86
Literały	87
Tekst	88
Skrypt i bloki stylów	88
Kod JavaScript	88
Komentarze	89
Warunki (if)	89
Iteracje (pętla each)	90
Filtry	90
Interpolacja	90
Case	91
Domieszki	91
Include	92
Extend	92
Samodzielne użycie Jade	93
Składnia Handlebars	96
Zmienne	96
Iteracja (each)	97
Niecytowany wynik	97
Warunki (if)	98
Unless	98
With	99
Komentarze	99
Własne funkcje pomocnicze	100
Włączenia (szablony częściowe)	101
Samodzielne użycie Handlebars	101
Wykorzystanie Jade i Handlebars w Express.js 4	103
Jade i Express.js	104
Handlebars i Express.js	104
Projekt — dodanie szablonów Jade do aplikacji Blog	105
layout.jade	105
index.jade	107
article.jade	109
login.jade	109
post.jade	110
admin.jade	111
Podsumowanie	113

Rozdział 5. Zapewnienie trwałości z wykorzystaniem MongoDB i Mongoskin	115
Łatwa i poprawna instalacja MongoDB	115
Jak uruchomić serwer Mongo	117
Manipulacja danymi z poziomu konsoli Mongo	118
Szczegóły dotyczące powłoki MongoDB	119
Przykład minimalistycznego natywnego sterownika MongoDB dla Node.js	120
Główne metody Mongoskin	122
Projekt — przechowywanie danych aplikacji Blog w MongoDB z wykorzystaniem Mongoskin	124
Projekt — dodawanie danych początkowych do bazy MongoDB	125
Projekt — pisanie testów Mocha	125
Projekt — zapewnienie trwałości	127
Uruchamianie aplikacji	137
Podsumowanie	138
Rozdział 6. Wykorzystanie sesji i OAuth do autoryzacji i uwierzytelniania użytkowników w aplikacjach Node.js	139
Autoryzacja z wykorzystaniem oprogramowania pośredniczącego Express.js	139
Uwierzytelnianie z wykorzystaniem tokena	140
Uwierzytelnianie z wykorzystaniem sesji	141
Projekt — dodawanie logowania z wykorzystaniem adresu e-mail i hasła do aplikacji Blog	142
Oprogramowanie pośredniczące sesji	142
Uwierzytelnianie aplikacji Blog	143
Uwierzytelnianie w aplikacji Blog	146
Uruchamianie aplikacji	147
OAuth w Node.js	147
Przykład Twitter OAuth 2.0 z OAuth Node.js	148
Everyauth	149
Projekt — dodawanie rejestracji Twitter OAuth 1.0 do aplikacji Blog z wykorzystaniem Everyauth	150
Dodawanie łącza Zaloguj za pomocą Twittera	150
Konfigurowanie strategii Twitter modułu Everyauth	151
Podsumowanie	156
Rozdział 7. Lepsza obsługa danych Node.js z wykorzystaniem biblioteki Mongoose ORM	157
Instalacja Mongoose	158
Ustanawianie połączenia w samodzielnym skrypcie Mongoose	158
Klasa Schema Mongoose	159
Haki pozwalające utrzymać organizację kodu	161
Własne metody statyczne i instancji	162
Modele Mongoose	162
Relacje i złączenia w populowaniu danych	164
Dokumenty zagnieżdżone	166
Pola wirtualne	167
Poprawianie zachowania typów schematów	168
Express.js + Mongoose = prawdziwy model MVC	170
Podsumowanie	179

Rozdział 8. Tworzenie API RESTowych na serwerze Node.js	
z wykorzystaniem Express.js i Hapi	181
Podstawy API RESTowego	182
Zależności w projekcie	183
Pokrycie testami Mocha i superagent	184
Implementacja serwera API RESTowego z wykorzystaniem Express i Mongoskin	189
Refaktoryzacja — serwer API RESTowego z wykorzystaniem Hapi.js	194
Podsumowanie	200
Rozdział 9. Aplikacje czasu rzeczywistego z wykorzystaniem WebSocket, Socket.IO i DerbyJS	201
Czym jest WebSocket?	201
Natywna obsługa WebSocket w Node.js z przykładem wykorzystania modułu ws	202
Implementacja WebSocket w przeglądarce	202
Serwer Node.js z wykorzystaniem modułu ws	203
Przykład Socket.IO i Express.js	205
Przykład edytora online do pracy w zespole	
z wykorzystaniem DerbyJS, Express.js i MongoDB	209
Zależności projektu i package.json	209
Kod po stronie serwera	210
Aplikacja DerbyJS	212
Widok DerbyJS	214
Próba edytora	216
Podsumowanie	216
Rozdział 10. Przygotowywanie aplikacji Node.js do wykorzystania produkcyjnego	217
Zmienne środowiskowe	217
Express.js w produkcji	218
Socket.IO w produkcji	220
Obsługa błędów	221
Domeny Node.js przeznaczone do obsługi błędów	223
Wielowątkowość z wykorzystaniem modułu Cluster	226
Wielowątkowość z wykorzystaniem modułu Cluster2	228
Logowanie i monitorowanie błędów	229
Monitorowanie	229
REPL w produkcji	230
Winston	231
Logowanie z wykorzystaniem aplikacji Papertrail	232
Tworzenie zadań z wykorzystaniem modułu Grunt	232
Kontrola wersji i wdrażanie z wykorzystaniem Git	236
Instalowanie Git	236
Generowanie kluczy SSH	237
Tworzenie lokalnego repozytorium Git	239
Przesyłanie lokalnego repozytorium do GitHub	239
Uruchamianie testów w chmurze z wykorzystaniem TravisCI	240
Konfiguracja TravisCI	241
Podsumowanie	242

Rozdział 11. Wdrażanie aplikacji Node.js	243
Wdrażanie do Heroku	243
Wdrażanie do Amazon Web Services (AWS)	248
Utrzymywanie działania aplikacji Node.js	
z wykorzystaniem forever, Upstart i init.d	251
forever	252
Skrypty Upstart	253
init.d	254
Właściwy sposób udostępniania zasobów statycznych z wykorzystaniem Nginx	256
Pamięć podręczna z wykorzystaniem Varnish	258
Podsumowanie	259
Rozdział 12. Publikowanie modułów Node.js i udział w projektach open source	261
Zalecana struktura folderów	262
Wymagane wzorce	262
package.json	264
Publikowanie w NPM	265
Wersje zablokowane	266
Podsumowanie	266
To już koniec	266
Dalsza lektura	267
Skorowidz	268



Aplikacje czasu rzeczywistego z wykorzystaniem WebSocket, Socket.IO i DerbyJS

Aplikacje czasu rzeczywistego stają się coraz bardziej popularne w grach, mediach społecznościowych, różnorodnych narzędziach, usługach i wiadomościach. Głównym czynnikiem odpowiedzialnym za tę tendencję jest pojawienie się nowych, lepszych technologii. Umożliwiają one wykorzystanie większej przepustowości do transmisji danych oraz wykonanie większej liczby obliczeń przetwarzających i pobierających dane.

Język HTML 5 był pionierem nowego standardu w połączeniach czasu rzeczywistego, zwanego **WebSocket**. Natomiast po stronie serwera mamy bardzo wydajną platformę wejścia-wyjścia Node.js, doskonale przystosowaną do wspierania JavaScript zaimplementowanego w przeglądarkach i technologii WebSocket.

Aby rozpocząć pracę z WebSocket i Node.js, postaramy się maksymalnie uprościć zadanie (zgodnie z regułą KISS — ang. *keep things simple stupid* ([http://pl.wikipedia.org/wiki/KISS_\(reguła\)](http://pl.wikipedia.org/wiki/KISS_(reguła)))) i omówimy następujące zagadnienia:

- Czym jest WebSocket?
- Natywna obsługa WebSocket w Node.js z przykładem wykorzystania modułu ws.
- Przykład wykorzystania Socket.IO z Express.js.
- Przykład edytora online do pracy w zespole z wykorzystaniem DerbyJS, Express.js i MongoDB.

Czym jest WebSocket?

WebSocket jest specjalnym „kanałem” komunikacyjnym między przeglądarkami (klientami) i serwerami. Jest to protokół języka HTML 5. Połączenie przez WebSocket jest ciągle, w przeciwieństwie do połączeń wykonywanych przez tradycyjne żądania HTML. Te ostatnie są zwykle inicjowane przez klienta. Dlatego serwer nie ma możliwości poinformowania klienta o aktualizacjach. Poprzez utrzymywanie połączenia dwustronnego między klientem i serwerem aktualizacje można przysyłać od razu bez konieczności nawiązywania połączenia przez klienta co pewien czas. Głównie to sprawia, że WebSocket świetnie przydaje się w aplikacjach czasu rzeczywistego, w których dane muszą być dostępne dla klienta natychmiast. Więcej informacji na temat protokołu WebSocket można znaleźć w artykule *About HTML5 WebSocket* (<http://www.websocket.org/aboutwebsocket.html>).

Do korzystania z techniki WebSocket we współczesnych przeglądarkach nie są potrzebne żadne specjalne biblioteki. Następujące pytanie dostępne w serwisie StackOverflow zawiera listę odpowiednich przeglądarek *What browsers support HTML5 WebSockets API?*

(<http://stackoverflow.com/questions/1253683/what-browsers-support-html5-websocket-api>). Jeśli chodzi o starsze przeglądarki, rozwiązaniem może być powrót do cyklicznego odpytywania.

Dodajmy, że cykliczne odpytywanie (zarówno krótkie, jak i długie) można też wykorzystać do emulacji zdolności do odpowiedzi aplikacji WWW czasu rzeczywistego. Pewne zaawansowane aplikacje (Socket.IO) wracają nawet do cyklicznego odpytywania, gdy WebSocket stanie się niedostępny, na przykład podczas problemów z połączeniem lub przy braku najnowszej przeglądarki. Cykliczne odpytywanie jest dość łatwe i nie zostało uwzględnione w tej książce. Można je zaimplementować za pomocą wywołania zwrótnego `setInterval()` i punktu wejściowego na serwerze. Za pomocą cyklicznego odpytywania nie można jednak zaimplementować prawdziwej komunikacji w czasie rzeczywistym; każde żądanie jest osobne.

Natywna obsługa WebSocket w Node.js z przykładem wykorzystania modułu ws

Niekiedy łatwiej jest zacząć od najprostszych rzeczy i później rozbudowywać rozwiązanie. Mając to na uwadze, tworzenie naszego miniprojektu zaczniemy od zbudowania natywnej implementacji WebSocket, która będzie się porozumiewać z serwerem za pomocą modułu `ws`:

- implementacja WebSocket w przeglądarce,
- serwer Node.js zaimplementowany z użyciem modułu `ws`.

Sprawdźmy to na krótkim przykładzie.

Implementacja WebSocket w przeglądarce

To będzie nasz kod front-endowy (plik `r09/basic/index.html`) dla przeglądarki Chrome w wersji 32.0.1700.77. Zaczniemy od typowych znaczników HTML:

```
<html>
  <head>
</head>
  <body>
```

Główny kod znajduje się wewnątrz znaczników `script`. Inicjalizujemy tu obiekt z globalnego modułu `WebSocket`:

```
<script type="text/javascript">
  var ws = new WebSocket('ws://localhost:3000');
```

Gdy tylko połączenie zostanie nawiązane, przesyłamy komunikat do serwera:

```
ws.onopen = function(event) {
  ws.send('komunikat front-endowy: ABC');
};
```

Zwykle wiadomości są przysyłane w odpowiedzi na akcje użytkownika, takie jak kliknięcie przycisku myszy. Gdy otrzymamy jakąś informację z serwera `WebSocket`, zostanie wykonana następująca funkcja:

```
ws.onmessage = function(event) {
  console.log('komunikat z serwera: ', event.data);
};
```

Dobłą praktyką jest zaimplementowanie funkcji obsługi zdarzenia `onerror`:

```
ws.onerror = function(event) {
  console.log('komunikat błędu serwera: ', event.data);
};
```

Następnie zamykamy znaczniki i zapisujemy plik:

```
</script>
</body>
</html>
```

Dla upewnienia się, że niczego nie pominęliśmy, poniżej przedstawiony jest pełny kod źródłowy pliku `r09/basic/index.html`:

```
<html>
<head>
</head>
<body>
  <script type="text/javascript">
    var ws = new WebSocket('ws://localhost:3000');
    ws.onopen = function(event) {
      ws.send('komunikat front-endowy: ABC');
    };
    ws.onmessage = function(event) {
      console.log('komunikat z serwera: ', event.data);
    };
  </script>
</body>
</html>
```

Serwer Node.js z wykorzystaniem modułu ws

WebSocket.org udostępnia usługę echo, służącą do testowania technologii WebSocket w przeglądarce, lecz sami możemy też utworzyć własny mały serwer Node.js z wykorzystaniem biblioteki `ws` (<http://npmjs.org/ws>, GitHub: <https://github.com/einaros/ws>):

```
$ mkdir node_modules
$ npm install ws@0.4.31
```

W pliku `r09/basic/server.js` importujemy moduł `ws` i inicjalizujemy serwer:

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({port: 3000});
```

Podobnie jak w kodzie front-endowym, na połączenie oczekujemy z wykorzystaniem wzorca zdarzeń. Gdy połączenie jest gotowe, w wywołaniu zwrotnym przesyłamy łańcuch XYZ i dołączamy funkcję nasłuchującą na zdarzenia ('message'), która będzie przechwytywać komunikaty przychodzące ze strony:

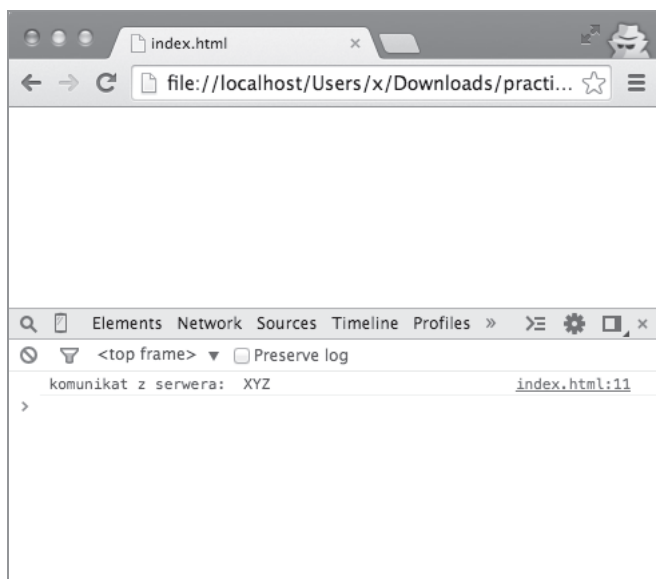
```
wss.on('connection', function(ws) {
  ws.send('XYZ');
  ws.on('message', function(message) {
    console.log('otrzymano: %s', message);
  });
});
```

Również tym razem dla wygody czytelników zamieszczam pełny kod źródłowy pliku `r09/basic/server.js`:

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({port: 3000});

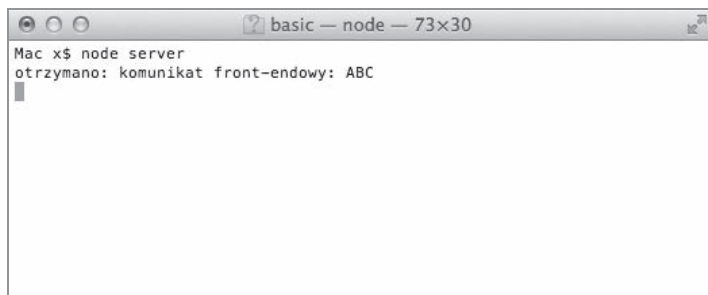
wss.on('connection', function(ws) {
  ws.send('XYZ');
  ws.on('message', function(message) {
    console.log('otrzymano: %s', message);
  });
});
```

Możemy już uruchomić serwer Node.js za pomocą polecenia `$ node server`. Następnie otwieramy plik `index.html` w przeglądarce. W konsoli JavaScript (w systemie Mac `Option+Command+j`) powinniśmy zobaczyć komunikat komunikat z serwera: XYZ (rysunek 9.1).



Rysunek 9.1. Przeglądarka wyświetla komunikat otrzymany za pośrednictwem WebSocket

W terminalu serwer Node.js zwraca wynik otrzymano: komunikat front-endowy: ABC, przedstawiony na rysunku 9.2.



Rysunek 9.2. Serwer wyświetla komunikat przeglądarki otrzymany za pośrednictwem WebSocket

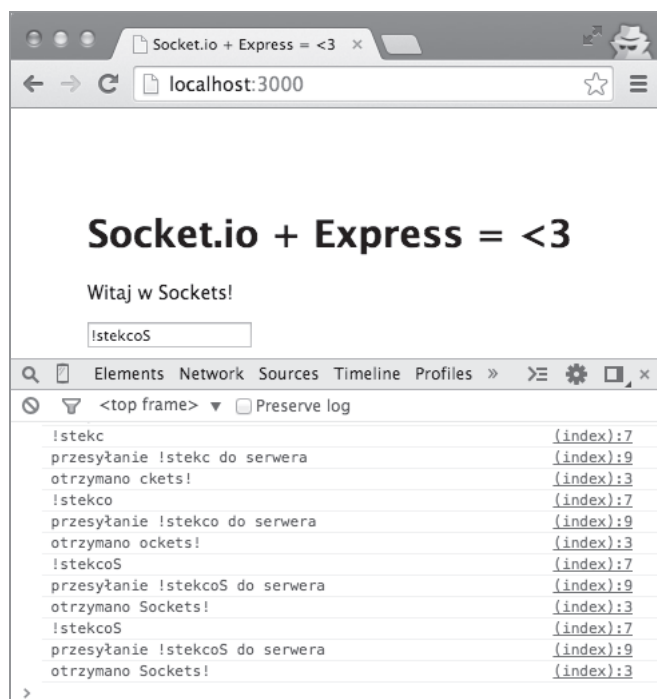
Natywna technologia WebSocket HTML 5 jest wspaniała. WebSocket jest jednak protokołem i rozwijającym się standardem. Oznacza to, że jego implementacja w różnych przeglądarkach może być nieco inna. Oczywiście jeśli wymagane jest wsparcie dla starszej przeglądarki, musimy sami sprawdzić dostępne rozwiązania.

Ponadto zawsze istnieje ryzyko utraty połączenia. Wtedy należy zadbać o ponowne nawiązanie połączenia. Aby zadbać o wsparcie w różnych przeglądarkach, a także o ponowne otwarcie połączenia, wielu programistów wykorzystuje bibliotekę Socket.IO, którą poznamy w kolejnym podrozdziale.

Przykład Socket.IO i Express.js

Pełne omówienie Socket.IO (<http://socket.io/>) wymagałoby osobnej książki. Ponieważ jednak jest to tak popularna biblioteka i tak łatwo zacząć z niej korzystać z Express.js, w niniejszym podrozdziale wykonamy ćwiczenie demonstrujące podstawy korzystania z tej biblioteki. Ten miniprojekt ilustruje dwukierunkowe połączenie między przeglądarką a serwerem.

Podobnie jak w większości aplikacji WWW czasu rzeczywistego, komunikacja między serwerem a klientem odbywa się w odpowiedzi na pewne akcje użytkownika lub w wyniku aktualizacji zachodzących na serwerze. W naszym przykładzie strona WWW zawiera pole formularza. Wprowadzane znaki są w czasie rzeczywistym przetwarzane przez serwer i wyświetlane w konsoli w odwrotnej kolejności (komunikacja z przeglądarki do serwera i z powrotem). W przykładzie tym wykorzystujemy narzędzie wiersza poleceń Express.js, służące do tworzenia szkieletów aplikacji, oraz moduły Socket.IO i Jade (zrzuty ekranowe działającej aplikacji są przedstawione na rysunku 9.3 i 9.4). Oczywiście możemy pobrać gotową aplikację z serwera FTP.



Rysunek 9.3. Wpisanie `!stekcoS` skutkuje wyświetleniem napisu `Sockets!`

```

debug - cleared close timeout for client 0JWlRRVRkNjTr_Amg-_4
debug - cleared heartbeat interval for client 0JWlRRVRkNjTr_Amg-_4
debug - discarding transport
GET /stylesheets/style.css 304 5ms
debug - served static content /socket.io.js
debug - client authorized
info - handshake authorized 332HiFwDv4XfYvQhg-_5
debug - setting request GET /socket.io/1/websocket/332HiFwDv4XfYvQhg-_
5
debug - set heartbeat interval for client 332HiFwDv4XfYvQhg-_5
debug - client authorized for
debug - websocket writing 1::
debug - emitting heartbeat for client 332HiFwDv4XfYvQhg-_5
debug - websocket writing 2::
debug - set heartbeat timeout for client 332HiFwDv4XfYvQhg-_5
debug - got heartbeat packet
debug - cleared heartbeat timeout for client 332HiFwDv4XfYvQhg-_5
debug - set heartbeat interval for client 332HiFwDv4XfYvQhg-_5
{ message: '!' }
debug - websocket writing 5::{"name":"receive","args":["!"]}
{ message: '!' }
debug - websocket writing 5::{"name":"receive","args":["!"]}
{ message: '!s' }
debug - websocket writing 5::{"name":"receive","args":["s!"]}
{ message: '!st' }
debug - websocket writing 5::{"name":"receive","args":["ts!"]}
{ message: '!ste' }
debug - websocket writing 5::{"name":"receive","args":["ets!"]}
{ message: '!stek' }
debug - websocket writing 5::{"name":"receive","args":["kets!"]}
{ message: '!stekc' }
debug - websocket writing 5::{"name":"receive","args":["ckets!"]}
{ message: '!stekco' }
debug - websocket writing 5::{"name":"receive","args":["ockets!"]}
{ message: '!stekcoS' }
debug - websocket writing 5::{"name":"receive","args":["Sockets!"]}
{ message: '!stekcoS' }
debug - websocket writing 5::{"name":"receive","args":["Sockets!"]}
debug - emitting heartbeat for client 332HiFwDv4XfYvQhg-_5
debug - websocket writing 2::
debug - set heartbeat timeout for client 332HiFwDv4XfYvQhg-_5
debug - got heartbeat packet
debug - cleared heartbeat timeout for client 332HiFwDv4XfYvQhg-_5
debug - set heartbeat interval for client 332HiFwDv4XfYvQhg-_5
debug - emitting heartbeat for client 332HiFwDv4XfYvQhg-_5

```

Rysunek 9.4. Serwer Express.js przechwytyjący i przetwarzający dane wejściowe w czasie rzeczywistym

Aby skorzystać z Socket.IO, możemy wykonać polecenie `$ npm install socket.io@0.9.16 --save` i powtórzyć je dla każdego modułu lub możemy skorzystać z pliku `package.json` i polecenia `$ npm install:`

```
{
  "name": "socket-express",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.0.0",
    "morgan": "~1.0.0",
    "cookie-parser": "~1.0.1",
    "body-parser": "~1.0.0",
    "debug": "~0.7.4",
    "jade": "~1.3.0",
    "socket.io": "0.9.16"
  }
}
```

Socket.IO w pewnym sensie można uznać za inny serwer, ponieważ obsługuje połączenia za pośrednictwem gniazd, a nie standardowych żądań HTTP. Poniżej widzimy, jak należy przekształcić kod, wygenerowany automatycznie przez Express.js:

```
var http = require('http');
var express = require('express');
var path = require('path');
var logger = require('morgan');
var bodyParser = require('body-parser');
```

Standardowa konfiguracja Express.js 4.x wygląda następująco:

```
var routes = require('./routes/index');
var app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
```

Kod obsługujący Socket.IO wygląda następująco:

```
var server = http.createServer(app);
var io = require('socket.io').listen(server);
```

Po ustanowieniu połączenia z serwerem Socket dołączamy funkcję nasłuchującą na zdarzenie `messageChange`, która implementuje logikę odwracającą przychodzący łańcuch tekstowy:

```
io.sockets.on('connection', function (socket) {
  socket.on('messageChange', function (data) {
    console.log(data);
    socket.emit('receive',
      data.message.split('').reverse().join('')
    );
  })
});
```

Zaczynamy od uruchomienia serwera bez użycia standardowych metod:

```
app.set('port', process.env.PORT || 3000);
server.listen(app.get('port'), function(){
  console.log('Serwer Express nasłuchujący na porcie ' + app.get('port'));
});
```

Na wypadek gdyby powyższe fragmenty kodu były niezrozumiałe, poniżej przedstawiona jest pełna treść pliku `r09/socket-express/app.js`:

```
var http = require('http');
var express = require('express');
var path = require('path');
var logger = require('morgan');
var bodyParser = require('body-parser');
```

```

var routes = require('./routes/index');
var app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);

var server = http.createServer(app);
var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
  socket.on('messageChange', function (data) {
    console.log(data);
    socket.emit('receive',
      data.message.split('').reverse().join('')
    );
  });
});
app.set('port', process.env.PORT || 3000);
server.listen(app.get('port'), function(){
  console.log('Serwer Express nasłuchujący na porcie ' +
    app.get('port')
  );
});

```

Mała uwaga dotycząca numerów portów: domyślnie połączenia z wykorzystaniem WebSocket odbywają się poprzez standardowe porty — 80 w przypadku HTTP i 443 w przypadku HTTPS.

Na koniec musimy zadbać o interfejs naszej aplikacji, za który odpowiada plik *index.jade*. Nic szczególnego; jedynie pole formularza i nieco kodu JavaScript w szablonie Jade:

```

extends layout

block content
  h1= title
  p Witaj w
    span.received-message #{title}
  input(type='text', class='message', placeholder='co Ci chodzi po głowie?',
    ↵onkeyup='send(this)')
  script(src="/socket.io/socket.io.js")
  script.
    var socket = io.connect('http://localhost');
    socket.on('receive', function (message) {
      console.log('otrzymano %s', message);
      document
        .querySelector('.received-message')
        .innerText = message;
    });
    var send = function(input) {
      console.log(input.value);
    }

```

```
var value = input.value;
console.log('przesyłanie %s do serwera', value);
socket.emit('messageChange', {message: value});
};
```

Ponownie uruchamiamy serwer i otwieramy przeglądarkę, aby sprawdzić działanie komunikacji w czasie rzeczywistym. Wpisanie tekstu w polu przeglądarki spowoduje przesłanie danych do serwera bez konieczności tworzenia żądań HTTP i oczekiwania na odpowiedź. Przybliżone wyniki uzyskane w przeglądarce są widoczne na rysunku 9.3. Logi serwera są widoczne na rysunku 9.4.

Więcej przykładów wykorzystania Socket.IO znajdziemy na stronie <http://socket.io/#how-to-use>.

Przykład edytora online do pracy w zespole z wykorzystaniem DerbyJS, Express.js i MongoDB

Derby (<http://derbyjs.com/>) jest nową, wyrafinowaną platformą MVC¹ przeznaczoną do wykorzystania z Express.js (<http://expressjs.com/>), w postaci kodu pośredniczącego. Natomiast Express.js (<http://expressjs.com/>) jest popularną platformą, wykorzystującą koncepcję oprogramowania pośredniczącego do wzbogacania funkcjonalności aplikacji. Derby wspiera też między innymi (<http://derbyjs.com/#features>) silnik synchronizacji danych Racer (<https://github.com/codeparty/racer>) oraz silnik szablonów Handlebars (<https://github.com/wycats/handlebars.js/>).

Meteor (<http://meteor.com/>) i Sails.js (<http://sailsjs.org/>) są kolejnymi w pełni rozwiniętymi platformami MVC, które można wykorzystać do tworzenia aplikacji czasu rzeczywistego Node.js i które są porównywalne z DerbyJS. Jednakże Meteor bardziej narzuca określony sposób rozwiązywania problemów i zwykle opiera się na rozwiązaniach i pakietach objętych prawami autorskimi.

Następny przykład pokazuje, jak łatwo jest utworzyć aplikację czasu rzeczywistego z wykorzystaniem Express.js, DerbyJS, MongoDB i Redis.

Struktura miniprojektu DerbyJS jest następująca:

- zależności projektu i *package.json*,
- kod po stronie serwera,
- aplikacja DerbyJS,
- widok DerbyJS,
- wypróbowanie edytora.

Zależności projektu i package.json

Jeśli nie zainstalowaliśmy jeszcze pakietów Node.js, NPM, MongoDB lub Redis, możemy to zrobić teraz, wykonując instrukcje przedstawione w następujących źródłach:

- instalowanie Node.js za pośrednictwem menedżera pakietów (<https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>),
- instalowanie npm (<http://howtonode.org/introduction-to-npm>),
- instalowanie MongoDB (<http://docs.mongodb.org/manual/installation/#install-mongodb>),
- szybki start z Redis (<http://redis.io/topics/quickstart>).

¹ <http://pl.wikipedia.org/wiki/Model-View-Controller>

Utwórzmy folder projektu o nazwie *editor* oraz plik *package.json* o następującej treści:

```
{
  "name": "editor",
  "version": "0.0.1",
  "description": "Edytor online do pracy w zespole",
  "main": "index.js",
  "scripts": {
    "test": "mocha test"
  },
  "git repository": "http://github.com/azat-co/editor",
  "keywords": "editor node derby real-time",
  "author": "Azat Mardan",
  "license": "BSD",
  "dependencies": {
    "derby": "~0.5.12",
    "express": "~3.4.8",
    "livedb-mongo": "~0.3.0",
    "racer-browserchannel": "~0.1.1",
    "redis": "~0.10.0"
  }
}
```

W ten sposób zainstalujemy moduły derby (DerbyJS), express (Express.js), livedb-mongo, racer-browserchannel i redis (klient Redis). DerbyJS i Express.js służą do obsługi tras i wykorzystują odpowiednie platformy (w wersjach 0.5.12 i 3.4.8). Redis, racer-browserchannel i livedb-mongo pozwalają DerbyJS korzystać z Redis i bazy danych MongoDB.

Kod po stronie serwera

Jako punkt wejściowy naszej aplikacji utwórzmy plik *editor/server.js*. Będzie on zawierać jeden wiersz kodu, służący do uruchomienia serwera Derby, który mamy zamiar utworzyć:

```
require('derby').run(__dirname + '/server.js');
```

Wpiszmy następujące wiersze w pliku *editor/server.js*. Najpierw zaimportujmy zależności:

```
var path = require('path'),
    express = require('express'),
    derby = require('derby'),
    racerBrowserChannel = require('racer-browserchannel'),
    liveDbMongo = require('livedb-mongo'),
```

Następnie zdefiniujmy plik aplikacji Derby:

```
app = require(path.join(__dirname, 'app.js')),
```

Inicjalizujemy aplikację Express.js:

```
expressApp = module.exports = express(),
```

klienta Redis:

```
redis = require('redis').createClient(),
```

oraz URI połączenia z lokalną bazą MongoDB:

```
mongoUrl = 'mongodb://localhost:27017/editor';
```

Teraz tworzymy obiekt `liveDbMongo` z wykorzystaniem URI połączenia i obiektem klienta `redis`:

```
var store = derby.createStore({
  db: liveDbMongo(mongoUrl + '?auto_reconnect', {
    safe: true
  }),
  redis: redis
});
```

Definiujemy publiczny folder ze statyczną treścią:

```
var publicDir = path.join(__dirname, 'public');
```

Następnie deklarujemy kod pośredniczący Express.js w wywołaniach łańcuchowych:

```
expressApp
  .use(express.favicon())
  .use(express.compress())
```

Bardzo ważne jest dołączenie kodu pośredniczącego specyficznego dla DerbyJS, który udostępnia trasy Derby i obiekty modelu:

```
.use(app.scripts(store))
  .use(racerBrowserChannel(store))
  .use(store.modelMiddleware())
  .use(app.router())
```

Następnie wpisujemy standardowy kod pośredniczący routera Express.js:

```
.use(expressApp.router);
```

Na tym samym serwerze możemy skorzystać jednocześnie z tras Express.js i DerbyJS — uniwersalnej trasy 404:

```
expressApp.all('*', function(req, res, next) {
  return next('404: ' + req.url);
});
```

Pełny kod źródłowy pliku `server.js` wygląda następująco:

```
var path = require('path'),
    express = require('express'),
    derby = require('derby'),
    racerBrowserChannel = require('racer-browserchannel'),
    liveDbMongo = require('livedb-mongo'),
    app = require(path.join(__dirname, 'app.js')),
    expressApp = module.exports = express(),
    redis = require('redis').createClient(),
    mongoUrl = 'mongodb://localhost:27017/editor';

var store = derby.createStore({
  db: liveDbMongo(mongoUrl + '?auto_reconnect', {
    safe: true
```

```

    }),
    redis: redis
  });

var publicDir = path.join(__dirname, 'public');

expressApp
  .use(express.favicon())
  .use(express.compress())
  .use(app.scripts(store))
  .use(racerBrowserChannel(store))
  .use(store.modelMiddleware())
  .use(app.router())
  .use(expressApp.router);

expressApp.all('*', function(req, res, next) {
  return next('404: ' + req.url);
});

```

Aplikacja DerbyJS

Aplikacja Derby.js (*app.js*) w inteligentny sposób dzieli kod między przeglądarkę a serwer, dzięki czemu funkcje i metody możemy pisać w jednym miejscu (w pliku Node.js). Jednakże elementy kodu *app.js* staną się kodem JavaScript (nie tylko Node.js), w zależności od reguł DerbyJS. To zachowanie pozwala na lepsze ponowne wykorzystanie kodu i jego organizację, ponieważ nie musimy duplikować tras, funkcji pomocniczych i metod. Jednym z miejsc, w których kod z aplikacji DerbyJS staje się kodem przeglądarki, jest wewnątrz funkcji `app.ready()`, o czym przekonamy się niebawem.

Zadeklarujmy zmienną i utwórzmy aplikację (*editor/app.js*):

```

var app;
app = require('derby').createApp(module);

```

Zadeklarujmy taką trasę główną, aby po jej odwiedzeniu przez użytkownika został utworzony nowy fragment, a użytkownik został przekierowany do trasy `/:snippetId`:

```

app.get('/', function(page, model, _arg, next) {
  snippetId = model.add('snippets', {
    snippetName: _arg.snippetName,
    code: 'var'
  });
  return page.redirect('/') + snippetId);
});

```

DerbyJS wykorzystuje wzorzec trasy podobny do Express.js, lecz zamiast odpowiedzi (`res`) korzystamy z obiektu `page` i uzyskujemy dane z argumentu `model`.

Trasa `/:snippetId` jest miejscem, w którym wyświetlany jest edytor. Aby zapewnić wsparcie dla aktualizacji czasu rzeczywistego w obiekcie DOM (ang. *Document Object Model*), musimy jedynie wywołać metodę `subscribe`:

```

app.get('/:snippetId', function(page, model, param, next) {
  var snippet = model.at('snippets.'+param.snippetId);
  snippet.subscribe(function(err){
    if (err) return next(err);
    console.log (snippet.get());
    model.ref('_page.snippet', snippet);
  });
});

```

```

    page.render();
  });
});

```

Metoda `model.at` z parametrem w postaci *nazwa_kolekcji*. ID przypomina wywoływanie metody `findById()` — innymi słowy, uzyskujemy obiekt z magazynu (bazy danych).

`model.ref()` pozwala nam na przypisanie obiektu do reprezentacji widoku. Zwykle w widoku napisalibyśmy `{{_page.snippet}}`, a w odpowiedzi widok sam by się odświeżył. Jednakże aby uatrakcyjnić wygląd edytora, skorzystamy z edytora Ace środowiska Cloud9 (<http://ace.c9.io/>). Ace jest dołączany do obiektu `editor` (globalna zmienna przeglądarki).

Kod JavaScript interfejsu w DerbyJS znajduje się w wywołaniu zwrotnym `app.ready`. Podczas uruchamiania aplikacji musimy ustawić treść Ace z poziomu modelu Derby:

```

app.ready(function(model) {
  editor.setValue(model.get('_page.snippet.code'));

```

Dalej aplikacja nasłuchuje na zmiany modelu (pochodzące od innych użytkowników) i aktualizuje edytor Ace nowym tekstem (kod front-endowy):

```

model.on('change', '_page.snippet.code', function(){
  if (editor.getValue() !== model.get('_page.snippet.code')) {
    process.nextTick(function(){
      editor.setValue(model.get('_page.snippet.code'), 1);
    })
  }
});

```

`process.nextTick` jest funkcją tworzącą harmonogram wywołania zwrotnego (przekazywanego jako parametr) w kolejnej iteracji pętli zdarzeń. Pozwala nam to uniknąć pętli nieskończonej, gdy zaktualizowany model od jednego użytkownika wyzwoli zdarzenie zmiany w edytorze Ace, co z kolei mogłoby wyzwolić niepotrzebne zaktualizowanie modelu zdalnego.

Kod nasłuchujący na zmiany Ace (na przykład na nowy znak) i aktualizujący model DerbyJS:

```

editor.getSession().on('change', function(e) {
  if (editor.getValue() !== model.get('_page.snippet.code')) {
    process.nextTick(function(){
      model.set('_page.snippet.code', editor.getValue());
    });
  }
});

```

`_page` jest specjalną nazwą DerbyJS wykorzystywaną do renderowania widoków.

Poniżej znajduje się pełny kod źródłowy pliku *editor/app.js*:

```

var app;

app = require('derby').createApp(module);

app.get('/', function(page, model, _arg, next) {
  snippetId = model.add('snippets', {
    snippetName: _arg.snippetName,
    code: 'var'
  });
  return page.redirect('/') + snippetId;
});

```

```

app.get('/:snippetId', function(page, model, param, next) {
  var snippet = model.at('snippets.'+param.snippetId);
  snippet.subscribe(function(err){
    if (err) return next(err);
    console.log (snippet.get());
    model.ref('_page.snippet', snippet);
    page.render();
  });
});

app.ready(function(model) {
  editor.setValue(model.get('_page.snippet.code'));
  model.on('change', '_page.snippet.code', function(){
    if (editor.getValue() !== model.get('_page.snippet.code')) {
      process.nextTick(function(){
        editor.setValue(model.get('_page.snippet.code'), 1);
      });
    }
  });
  editor.getSession().on('change', function(e) {
    if (editor.getValue() !== model.get('_page.snippet.code')) {
      process.nextTick(function(){
        model.set('_page.snippet.code', editor.getValue());
      });
    }
  });
});
});

```

Widok DerbyJS

Widok DerbyJS (*views/app.html*) jest dość prosty. Zawiera wbudowane znaczniki, takie jak <Title:>, ale większość treści jest generowana dynamicznie przez edytor Ace po wczytaniu strony.

Zacznijmy od zdefiniowania treści elementów title i head:

```

<Title:>
  Edytor online do pracy w zespole
<Head:>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Edytor</title>
  <style type="text/css" media="screen">
    body {
      overflow: hidden;
    }

    #editor {
      margin: 0;
      position: absolute;
      top: 0px;
      bottom: 0;
      left: 0;
      right: 0;
    }
  </style>

```

Następnie wczytujemy jQuery i Ace z systemu Content Delivery Network (CDN):

```
<script src="//cdnjs.cloudflare.com/ajax/libs/ace/1.1.01/ace.js"></script>
<script src="//code.jquery.com/jquery-2.1.0.min.js"></script>
```

Wewnątrz elementu body tworzymy ukryty element input i element edytora:

```
<Body:>
  <input type="hidden" value="{_page.snippet.code}" class="code"/>
  <pre id="editor" value="{_page.snippet.code}"></pre>
```

Inicjalizujemy obiekt edytora Ace jako obiekt globalny (zmienna editor), a następnie ustawiamy szablon i język (oczywiście JavaScript!) za pomocą funkcji `setTheme()` i `setMode()`:

```
<script>
  var editor = ace.edit("editor");
  editor.setTheme("ace/theme/twilight");
  editor.getSession().setMode("ace/mode/javascript");
</script>
```

Pełny kod źródłowy pliku *views/app.html* wygląda następująco:

```
<Title:>
  Edytor online do pracy w zespole
<Head:>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Edytor</title>
  <style type="text/css" media="screen">
    body {
      overflow: hidden;
    }

    #editor {
      margin: 0;
      position: absolute;
      top: 0px;
      bottom: 0;
      left: 0;
      right: 0;
    }
  </style>
  <script src="//cdnjs.cloudflare.com/ajax/libs/ace/1.1.01/ace.js"></script>
  <script src="//code.jquery.com/jquery-2.1.0.min.js"></script>
<Body:>
  <input type="hidden" value="{_page.snippet.code}" class="code"/>
  <pre id="editor" value="{_page.snippet.code}"></pre>
<script>
  var editor = ace.edit("editor");
  editor.setTheme("ace/theme/twilight");
  editor.getSession().setMode("ace/mode/javascript");
</script>
```

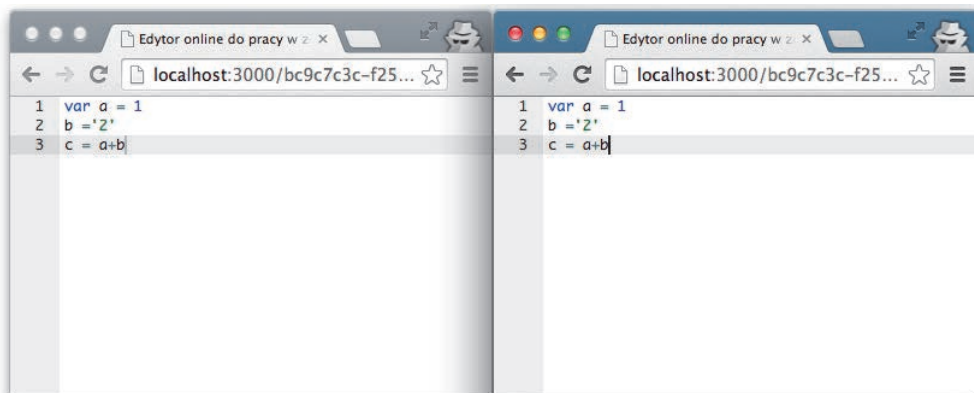
■ **Uwaga** Koniecznie należy zachować tę samą nazwę widoku (na przykład *app.html*) jak nazwa pliku aplikacji DerbyJS (*app.js*), ponieważ dzięki temu platforma wie, z czego skorzystać.

Próba edytora

Jeśli wykonaliśmy wszystkie wcześniejsze kroki, powinniśmy mieć pliki *app.js*, *index.js*, *server.js*, *views/app.html* i *package.json*.

Musimy jeszcze zainstalować moduły za pomocą polecenia `$ npm install` oraz uruchomić bazy danych za pomocą poleceń `$ mongod` i `$ redis-server`. Powinny pozostać uruchomione. Teraz możemy już uruchomić aplikację za pomocą polecenia `$ node .` lub `$ node index`.

Otwórzmy pierwsze okno edytora pod adresem *http://localhost:3000/*. Powinniśmy zostać przekierowani do nowego fragmentu (w adresie URL powinien być widoczny parametr ID). Otwórzmy drugie okno przeglądarki w tej samej lokalizacji i zacznijmy pisać (rysunek 9.5). Kod w pierwszym oknie powinien zostać zaktualizowany! Gratulacje! Wystarczyło kilka minut, aby powstała aplikacja, której utworzenie mogłoby zająć programistom kilka miesięcy w pierwszej dekadzie XXI wieku, gdy front-endowy JavaScript i strony wykorzystujące AJAX zaczynały nabierać popularności.



Rysunek 9.5. Edytor kodu online do pracy w zespole

Działający projekt jest dostępny na serwerze FTP w folderze *r09/editor*.

Podsumowanie

W tym rozdziale poznaliśmy natywne wsparcie dla technologii WebSocket we współczesnych przeglądarkach HTML5 i dowiedzieliśmy się, jak zacząć pracę z Socket.IO i Express.js, aby wykorzystać możliwości technologii WebSocket w Node.js. Ponadto poznaliśmy platformę DerbyJS na przykładzie edytora.

W kolejnym rozdziale przejdziemy do najważniejszego etapu tworzenia rzeczywistego projektu, w którym aplikacja Node.js staje się gotowa do produkcji. W tym celu uzupełnimy konfigurację, obsłużymy monitorowanie, logowanie i kilka innych funkcji.

Skorowidz

A

- AJAX, 62
- Amazon Linux AMI, 248
- API, application programming interface, 35, 181
- API RESTowe, 62, 182
- aplikacja Blog
 - strona administracyjna, 64
 - strona domowa, 61
- aplikacja DerbyJS, 212
- aplikacje
 - back-endowe, 51
 - czasu rzeczywistego, 51, 201
- atrybuty, 86
- automatyczne testowanie aplikacji WWW, 73
- autoryzacja, 139
- AWS, Amazon Web Services, 248

B

- back-end, 115
- baza MongoDB, 125
- bazy danych NoSQL, 115
- BDD, behavior-driven development, 73, 79
- białe znaki, 33
- biblioteka
 - asercji, 81
 - chai assert, 78
 - Expect.js, 80

- Handlebars, 96, 101
- jQuery, 62
- Mongoose, 51
- Mongoose ORM, 157
- OAuth, 138
- Sequelize, 51
- Stylus, 127

- bloki stylów, 88
- błąd 404, 128

C

- CRUD, 182
- CURL, 24, 194

D

- dane testowe, 125
- debuger, 41
- debugowanie asynchronicznego kodu, 224
- definiowanie funkcji, 29
- deklarowanie zależności, 66
- DerbyJS, 209–214
- dodatki do Heroku, 247
- dodawanie
 - danych do bazy, 125
 - łącza, 150
 - szablonów, 105
- dokumenty zagnieżdżone, 166
- domieszki, 91
- dostęp do zakresu globalnego, 34

- działanie
 - Express.js, 52
 - OAuth, 147
 - testów Mocha, 137
- dziedziczenie funkcyjne, 31
- prototypowe, 31

E

- EC2, Elastic Compute Cloud, 248
- edytory
 - kodu, 44, 216
 - tekstu, 45
- eksportowanie modułów, 34
- element
 - script, 88
 - style, 88
- e-mail, 142
- EPEL, 248
- Express.js, 49, 104
 - instalowanie, 52
 - Jade to Haml, 60
 - konfiguracja aplikacji, 60
 - oprogramowanie pośredniczące, 60
 - przetwarzanie żądania, 68
 - szkielet aplikacji, 56
 - trasy, 59
 - tworzenie folderów, 64
 - w produkcji, 218
 - wiersz poleceń, 57

Express.js 4, 103
Express.js Generator, 53, 57

F

filtry, 90
forever, 252
format JSON, 229
front-end, 115
funkcja
 \$.ajax(), 62
 destroy(), 147
 findOne, 130, 146
 listen(), 228
 loadCollection, 195
 req.session, 147
 res.end(), 69
 res.render(), 69
funkcje, 29
 Jade, 85
 pomocnicze, 100
 przekazywanie, 30
 tworzenie, 29
 wywoływanie, 30

G

generowanie kluczy SSH, 237
getter, 168
Git, 236
GitHub, 239

H

haki, 75, 161, 164
Handlebars, 85, 103, 104
 funkcje pomocnicze, 100
 instrukcja if, 98
 instrukcja unless, 98
 instrukcja with, 99
 iteracja, 97
 komentarze, 99
 samodzielne użycie, 101
 włączenia, 101
 zmienne, 96
Hapi.js, 194
hasło, 142

I

IaaS, Infrastructure as a Service, 236
IDE, 44
identyfikator obiektu, 186
implementacja
 serwera API RESTowego, 189
 WebSocket, 202
importowanie modułów, 34
informacja o procesach, 33
inicjalizacja NPM, 65
init.d, 252, 254
instalacja
instalator, 22
instalowanie
 bez użycia sudo, 24
 Express.js, 52, 54
 Git, 236
 Jade, 93
 kilku wersji, 26
 modułów, 38
 narzędzi, 47
 Mocha, 73, 185
 MongoDB, 115
 Mongoose, 158
 Node.js, 21
 z repozytorium Git, 24
 z wykorzystaniem Nave, 25
 z wykorzystaniem NVM, 26
 z wykorzystaniem pliku tar, 23
 za pomocą HomeBrew, 22
instrukcja
 case, 91
 contains, 126
 doctype 5, 105
 each, 90
 if, 89, 98
 include, 92
 require, 125
 return, 146
 unless, 98
 with, 99
instrukcje if-else, 218
interfejs
 programistyczny aplikacji, 35
 użytkownika, 181
 wiersza poleceń, 57

interpolacja, 90
iteracje, 90, 97

J

Jade, 60, 70, 85, 103
 atrybuty, 86
 bloki stylów, 88
 domieszki, 91
 filtry, 90
 instrukcja case, 91
 instrukcja extend, 92
 instrukcja include, 92
 interpolacja, 90
 komentarze, 89
 literały, 87
 samodzielne użycie, 93
 skrypt, 88
 tekst, 88
 zmienne, 86
 znaczniki, 86
język znaczników Haml, 60
JSON, 190

K

klasa Schema, 159
klient MongoDB, 118
klucze SSH, 237
kod JavaScript, 88
kombinacja adresu e-mail i hasła, 146
komentarze, 89, 99
kompilator C++, 249
komunikat przeglądarki, 204
komunikaty HipChat, 222
konfigurowanie
 aplikacji Express.js, 60
 strategii Twitter, 151
 TravisCI, 241
konsola
 AWS, 250
 Node.js, 27
 REPL, 27
kontrola wersji, 236
konwencje, 32

L

layout, 105, 107
lista metod Mongoose, 164
literały, 87
logowanie, 142, 229, 232

M

manipulacja danymi, 118
mapowanie relacyjno-obiektowe, 157
mechanizm weryfikacji, 157
menedżer
 NPM, 21
 NVM, 26
metoda
 __express, 103
 ajax(), 62
 app.param(), 189
 app.use(), 189
 auth, 146
 DELETE, 182
 document.save(), 159
 exports.edit, 176
 find, 166
 findOne, 197
 GET, 129, 182
 jade.compile(), 95
 jade.renderFile, 95
 loadCollection, 196
 model.at, 213
 POST, 129, 182
 PUT, 182
 require, 103
 route, 196
 save(), 168
 toArray(), 122
metody
 biblioteki chai assert, 78
 idempotentne, 183
 Mongoskin, 122, 124
 statyczne, 162
Mocha, 126
model MVC, 170
modele Mongoose, 162
moduł
 assert, 76
 Chai assert, 78
 Cluster, 226, 227

Cluster2, 228
connect-redis, 220
Everyauth, 149, 151
forever, 252
fs, 37
Grunt, 232
http, 36, 81, 183
NPM, 81
querystring, 37
url, 37
util, 37
ws, 202, 203
MongoDB, 115, 124, 138
Mongoose, 157
 dokumenty zagnieżdżone, 166
 haki, 161
 instalacja, 158
 klasa Schema, 159
 modele, 162
 pola wirtualne, 167
 typy schematów, 168
Mongoskin, 120, 122, 138
monitorowanie błędów, 229
MVC, 51

N

narzędzie, 37
 Aptana Studio, 46
 Coda, 46
 Express.js Generator, 57
 express-generator, 55
 Heroku Toolbelt, 244
 HomeBrew, 22
 MacPorts, 22
 Node Inspector, 42, 45
 Notepad ++, 46
 NPM, 55
 Sublime Text, 45
 TextMate, 45
 WebStorm IDE, 46
natywny sterownik MongoDB, 158
Nave, 25
nawias
 kątowny, 97
 klamrowy, 97
nazewnictwo, 32

Nginx, 256
nierelacyjne bazy danych, 115
notacja
 camelCase, 32
 literałów obiektowych, 29
NVM, Node Version Manager, 26, 38

O

OAuth, 147
obiekt Schema, 168
obsługa
 błędów domeny Node.js, 223
 WebSocket, 202
 żądania, 177, 190
odczyt plików, 38
okno terminala, 118
opcje
 pobierania, 23
 polecenia \$ mocha, 74
operacje CRUD, 183, 184
oprogramowanie
 pośrednicząc, 49
 pośredniczące Express.js, 139
 pośredniczące sesji, 142
 sterowane przez testy, 73
 sterowane przez zachowania, 73
ORM, 157

P

PaaS, Platform as a Service, 236
package.json, 264
pamięć podręczna, 258
panel monitorowania, 229
Papertrail, 232
pętle, 90
platforma
 Anglers JS, 181
 Backbone.js, 181
 Derby, 209
 Ember.js, 181
 Hapi.js, 194
 jako usługa, 236
 Meteor, 209
 Mocha, 73
 Sails.js, 209

- plik
 - 404.html, 257
 - admin.jade, 111
 - admin.js, 136
 - app.js, 67, 82, 128, 144, 171
 - article.jade, 109
 - article.js, 132, 173, 177
 - cluster.js, 227
 - hapi-app.js, 198
 - index.jade, 71, 107
 - index.js, 173, 185, 187, 191
 - layout.jade, 60, 105, 107
 - login.jade, 109
 - Makefile, 82, 83
 - mongo-native-insert.js, 121
 - package.json, 55, 65, 84, 134, 195, 210, 264
 - post.jade, 110
 - server.js, 203, 210, 250
 - test.js, 78
 - user.js, 129, 175, 179
 - users.json, 125
 - pliki
 - .env, 246
 - .gitignore, 246
 - jade, 134
 - tar, 23
 - podmoduły Everyauth, 149
 - pola wirtualne, 167
 - polecenia Git, 248
 - polecenie
 - \$ grunt, 234, 236
 - \$ heroku create, 246
 - \$ heroku login, 244
 - \$ make start, 155
 - \$ mkdir tests, 81
 - \$ mocha, 74
 - \$ mocha tests, 82
 - \$ node app, 71
 - \$ node cluster, 227
 - \$ node hapi-app, 200
 - \$ node server, 204
 - \$ npm install, 55, 66, 216
 - \$ npm ls, 56
 - curl, 194
 - extend, 92
 - make test, 84
 - połączenie z bazą, 189
 - populowanie danych, 164
 - powłoka MongoDB, 119
 - projekt
 - dodawanie logowania, 142
 - dodawanie rejestracji, 150
 - przechowywanie danych, 124
 - test BDD, 80
 - projekt Blog, 61
 - projekty open source, 261
 - przecinki, 32
 - przekazywanie funkcji, 30
 - przesyłanie
 - danych, 62
 - repozytorium, 239
 - przetwarzanie żądania, 68
 - publikowanie
 - modułów, 261
 - w NPM, 265
- ## R
- refaktoryzacja, 170, 194
 - rejestracja, 150
 - relacje, 164
 - REPL, 27
 - REPL w produkcji, 230
 - repozytorium
 - Git, 24, 239
 - GitHub, 149
 - Node, 249
 - REST, representational state transfer, 181
 - rzutowanie typu, 157
- ## S
- SaaS, Software as a Service, 231
 - samodzielne użycie Jade, 93
 - serwer
 - API RESTowego, 182, 189, 194
 - Express.js, 206
 - Hapi, 200
 - Mongo, 117
 - Node.js, 204
 - proxy, 257
 - sesja, 141
 - setter, 168
 - silniki szablonów, 60, 70, 85
 - składnia, 28
 - Expect.js, 80
 - Handlebars, 96
 - Jade, 85
 - skrypty, 88
 - init.d, 254
 - Upstart, 253
 - słowa zarezerwowane, 33
 - słowo kluczowe return, 146
 - Socket.IO, 205–208
 - Socket.IO w produkcji, 220
 - sprawdzanie typu Array, 77
 - SSH, Secure Shell, 231
 - strategia Twitter modułu Everyauth, 151, 156
 - strona
 - administracyjna, 111, 138
 - artykułu, 109
 - logowania, 109
 - publikowania artykułów, 110
 - struktura CRUD, 182
 - strumieniowe przesyłanie danych, 38
 - sudo, 24
 - superagent, 184
 - system ciągłej integracji, 240
 - szablony
 - częściowe, 105
 - Jade, 60, 70
 - szkielety aplikacji, 56, 61
- ## Ś
- ścieżka absolutna, 35
 - średniki, 32
 - środowiska IDE, 44
 - środowisko
 - Node Inspector, 43
 - WebStorm, 46, 47
- ## T
- tablice, 31
 - TDD, test-driven development, 73, 76
 - tekst, 88
 - terminal, 118

testowanie

- aplikacji, 80
- w chmurze, 240
- testy Mocha, 125, 184
- token, 140, 148

- trasa, 50, 59, 139
 - authenticate, 129
 - DELETE, 131
 - GET, 130, 132
 - login, 129
 - logout, 129
 - POST, 130
 - PUT, 131
 - users, 129

trasy API RESTowego, 128

TravisCI, 240, 241

Twitter, 151

tworzenie

- API RESTowych, 181
- aplikacji WWW, 49
- folderów, 64
- lokalnego repozytorium Git, 239
- modelu, 159
- serwera, 40
- szkieletów aplikacji, 56, 61
- zadań, 232

typ danych Buffer, 29

typowanie słabe, 28

U

udostępnianie zasobów

- statycznych, 256

układ strony, 105

Upstart, 252

uruchamianie

- aplikacji, 71, 137, 147
- klienta MongoDB, 118
- serwera MongoDB, 117
- skryptów Node.js, 28
- skryptu MongoDB, 123

usługa

- echo, 203
- SaaS, 232

ustanawianie połączenia, 158

uwierzytelnianie, 139–146

- aplikacji, 143
- w aplikacji, 146

użyteczność, 181

V

Varnish Cache, 258

W

wątek roboczy, 226

wcięcia, 33

wdrażanie aplikacji, 236, 243

- do AWS, 248

- do Heroku, 243

WebSocket, 201–205

wersje zablokowane, 266

weryfikacja instalacji, 26

widok DerbyJS, 214

wielowątkowość, 226, 228

wieloznacznik, 69

Winston, 231

własne

- funkcje pomocnicze, 100

- metody instancji, 162

włączenia, 101

wykorzystanie

- adresu e-mail, 142

- DerbyJS, 201, 209

- Everyauth, 150

- Express, 189

- forever, 251

- Git, 236

- Handlebars, 103

- Hapi.js, 194

- init.d, 251

- Jade, 103

- modułu Cluster, 226

modułu Cluster2, 228

modułu Grunt, 232

modułu ws, 203

Mongoskin, 124, 189

Nginx, 256

OAuth, 139

produkcyjne aplikacje, 217

sesji, 139, 141

Socket.IO, 201, 205

tokena, 140

Upstart, 251

Varnish, 258

WebSocket, 201

wymagane wzorce, 262

wyodrębnianie obiektów, 190

wywołania zwrotne, 39, 186

wywoływanie funkcji, 30

Z

zakres globalny, 33

zalecana struktura folderów, 262

zależności, 183

zapewnianie trwałości, 127

zapis plików, 38

złączenia, 164

zmiany w plikach, 46

zmiennne, 86, 96

zmiennne środowiskowe, 217

znaczniki, 86

Ż

żądania, 183

- DELETE, 69, 182, 194

- GET, 69, 129, 182, 193

- POST, 69, 129, 182, 194

- PUT, 69, 182, 190

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 w praktyce

Tworzenie skalowalnych aplikacji sieciowych

JavaScript to język programowania rozwijany od 1995 roku. Przez długi czas był kojarzony tylko i wyłącznie ze stronami internetowymi i z przeglądarkami, jednak te czasy odchodzą w niepamięć, a JavaScript z powodzeniem jest dziś stosowany po stronie serwera.

Jeżeli chcesz sprawdzić, jak to działa, sięgnij po tę książkę i zainstaluj Node.js. Jest to platforma, która zapewnia najwyższą wydajność, ponieważ korzysta z nieblokujących operacji I/O oraz asynchronicznego mechanizmu zdarzeń. Co więcej, została ona oparta na najwydajniejszym silniku wspierającym język JavaScript, czyli na V8. Dzięki tej książce masz niepowtarzalną okazję poznać podstawy działania Node.js, zaznajomić się ze składnią języka JavaScript, a także zacząć korzystać z zaawansowanych mechanizmów autoryzacji, przechowywania danych czy zdarzeń. Ponadto nauczysz się tworzyć aplikacje czasu rzeczywistego z użyciem WebSocket, Socket.IO i DerbyJS oraz zbudujesz superwydajne API REST-owe. Książka ta jest obowiązkową lekturą dla wszystkich osób, dla których ważna jest najwyższa wydajność i które chcą poznać nowinki technologiczne. Warto ją mieć!

Dzięki tej książce:

- zainstalujesz i skonfigurujesz platformę Node.js
- poznasz składnię i podstawy języka JavaScript
- skorzystasz z mechanizmu szablonów Jade
- zbudujesz wydajne API REST-owe
- wykorzystasz potencjał platformy Node.js

Azat Mardan — przedsiębiorca, inżynier oprogramowania, entuzjasta jogi i diety paleo. Ma ponad 12 lat doświadczenia w wytwarzaniu oprogramowania. Jest autorem dziewięciu książek poświęconych językowi JavaScript oraz platformie Node.js. Aktualnie zajmuje stanowisko starszego inżyniera oprogramowania oraz lidera zespołu w DocuSign. Był współzałożycielem Gizmo (platformy do prowadzenia kampanii marketingowych dla urządzeń mobilnych) oraz członkiem zespołów wytwarzających oprogramowanie dla organizacji rządowych.

Helion	
33966	numer katalogowy
księgarnia internetowa	
http://helion.pl	
zamówienia telefoniczne	
	0 801 339900
	0 601 339900
Sprawdź najnowsze promocje: • http://helion.pl/promocje Książki najchętniej czytane: • http://helion.pl/bestsellery Zamów informacje o nowościach: • http://helion.pl/nowosci	
Helion SA ul. Kościuszki 1c, 44-100 Gliwice tel.: 32 230 98 63 e-mail: helion@helion.pl http://helion.pl	
Informatyka w najlepszym wydaniu	

Apress

ISBN 978-83-283-1085-8



cena: 54,90 zł

sięgnij po **WIĘCEJ**



KOD KORZYŚCI