

Mike Cantelon
Marc Harter
TJ Holowaychuk
Nathan Rajlich



W AKCJI

Poznaj potencjał Node.js!

Tytuł oryginału: Node.js in Action

Tłumaczenie: Robert Górczyński

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-246-9678-9

Original edition copyright © 2014 by Manning Publications Co.
All rights reserved

Polish edition copyright © 2014 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 bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

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

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

Printed in Poland.

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

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

Spis treści

Wstęp	11
Przedmowa	13
Podziękowania	15
O książce	17

CZĘŚĆ I. PODSTAWY NODE 19

Rozdział 1. Witamy w Node.js 21

1.1.	Node jest zbudowane w oparciu o JavaScript	22
1.2.	Asynchroniczna i oparta na zdarzeniach: przeglądarka internetowa	23
1.3.	Asynchroniczny i oparty na zdarzeniach: serwer	25
1.4.	Aplikacje DIRT	27
1.5.	Domyślna aplikacja jest typu DIRT	29
1.5.1.	Prosty przykład aplikacji asynchronicznej	30
1.5.2.	Serwer HTTP	30
1.5.3.	Strumieniowanie danych	32
1.6.	Podsumowanie	33

Rozdział 2. Tworzenie aplikacji wielopokojowego czatu 35

2.1.	Ogólny opis aplikacji	36
2.2.	Wymagania aplikacji i konfiguracja początkowa	38
2.2.1.	Obsługa HTTP i WebSocket	39
2.2.2.	Tworzenie struktury plików aplikacji	39
2.2.3.	Wskazanie zależności	40
2.2.4.	Instalacja zależności	41
2.3.	Udostępnianie plików HTML, CSS i kodu JavaScript działającego po stronie klienta	41
2.3.1.	Tworzenie podstawowego serwera plików statycznych	42
2.3.2.	Dodanie plików HTML i CSS	45
2.4.	Obsługa wiadomości czatu za pomocą biblioteki Socket.IO	46
2.4.1.	Konfiguracja serwera Socket.IO	48
2.4.2.	Obsługa zdarzeń oraz scenariuszy w aplikacji	49
2.5.	Użycie kodu JavaScript działającego po stronie klienta do utworzenia interfejsu użytkownika aplikacji	54
2.5.1.	Przekazywanie do serwera wiadomości oraz żądań zmiany pokoju lub nazwy użytkownika	54
2.5.2.	Wyświetlenie w interfejsie użytkownika wiadomości i listy dostępnych pokoi	55
2.6.	Podsumowanie	58

Rozdział 3. Podstawy programowania w Node 61

- 3.1. Organizacja i wielokrotne użycie kodu Node 62
 - 3.1.1. Tworzenie modułu 64
 - 3.1.2. Dostrajanie tworzenia modułu za pomocą `module.exports` 66
 - 3.1.3. Wielokrotne użycie modułów za pomocą katalogu `node_modules` 68
 - 3.1.4. Zastrzeżenia 68
- 3.2. Techniki programowania asynchronicznego 69
 - 3.2.1. Użycie wywołań zwrotnych do obsługi zdarzeń jednorazowych 71
 - 3.2.2. Użycie emitera zdarzeń do obsługi powtarzających się zdarzeń 74
 - 3.2.3. Wyzwania pojawiające się podczas programowania asynchronicznego 82
- 3.3. Sekwencja logiki asynchronicznej 84
 - 3.3.1. Kiedy stosować szeregową kontrolę przepływu? 85
 - 3.3.2. Implementacja szeregowej kontroli przepływu 86
 - 3.3.3. Implementacja równoległej kontroli przepływu 89
 - 3.3.4. Użycie narzędzi opracowanych przez społeczność 91
- 3.4. Podsumowanie 92

CZĘŚĆ II. TWORZENIE APLIKACJI SIECIOWYCH W NODE 95

Rozdział 4. Tworzenie aplikacji sieciowej w Node 97

- 4.1. Podstawy dotyczące serwera HTTP 99
 - 4.1.1. Jak przychodzące żądania HTTP są przez Node przedstawiane programiście? 99
 - 4.1.2. Prosty serwer HTTP odpowiadający komunikatem „Witaj, świecie” 100
 - 4.1.3. Odczyt nagłówek żądania i zdefiniowanie nagłówek odpowiedzi 101
 - 4.1.4. Ustawienie kodu stanu odpowiedzi HTTP 102
- 4.2. Tworzenie usługi sieciowej RESTful 102
 - 4.2.1. Tworzenie zasobów za pomocą żądań POST 103
 - 4.2.2. Pobieranie zasobów za pomocą żądania GET 105
 - 4.2.3. Usunięcie zasobu za pomocą żądania DELETE 107
- 4.3. Udostępnianie plików statycznych 108
 - 4.3.1. Tworzenie serwera plików statycznych 109
 - 4.3.2. Obsługa błędów serwera 112
 - 4.3.3. Wyprzedzająca obsługa błędów za pomocą wywołania `fs.stat()` 113
- 4.4. Akceptacja danych wejściowych użytkownika przekazanych za pomocą formularza sieciowego 114
 - 4.4.1. Obsługa wysłanych pól formularza sieciowego 114
 - 4.4.2. Obsługa przekazanych plików za pomocą `formidable` 118
 - 4.4.3. Sprawdzanie postępu operacji przekazywania plików 122
- 4.5. Zabezpieczanie aplikacji dzięki użyciu protokołu HTTPS 122
- 4.6. Podsumowanie 124

Rozdział 5. Przechowywanie danych aplikacji Node 125

- 5.1. Niewymagający serwera magazyn danych 126
 - 5.1.1. Magazyn danych w pamięci 126
 - 5.1.2. Magazyn danych oparty na plikach 127

- 5.2. System zarządzania relacyjną bazą danych 130
 - 5.2.1. *MySQL* 131
 - 5.2.2. *PostgreSQL* 139
- 5.3. Bazy danych typu NoSQL 141
 - 5.3.1. *Redis* 141
 - 5.3.2. *MongoDB* 146
 - 5.3.3. *Mongoose* 149
- 5.4. Podsumowanie 151

Rozdział 6. Framework Connect 153

- 6.1. Konfiguracja aplikacji Connect 154
- 6.2. Jak działa metoda pośrednicząca frameworka Connect? 155
 - 6.2.1. *Metody pośredniczące wyświetlające żądanie* 156
 - 6.2.2. *Metoda pośrednicząca udzielająca odpowiedzi w postaci komunikatu „Witaj, świecie”* 157
- 6.3. Dlaczego kolejność metod pośredniczących ma znaczenie? 158
 - 6.3.1. *Kiedy metoda pośrednicząca nie wywołuje next()?* 158
 - 6.3.2. *Użycie kolejności metod pośredniczących do przeprowadzenia uwierzytelnienia* 159
- 6.4. Montowanie metody pośredniczącej i serwera 160
 - 6.4.1. *Metody pośredniczące przeprowadzające uwierzytelnianie* 161
 - 6.4.2. *Metoda pośrednicząca wyświetlająca panel administracyjny* 162
- 6.5. Tworzenie konfigurowalnej metody pośredniczącej 164
 - 6.5.1. *Tworzenie konfigurowalnej metody pośredniczącej logger()* 164
 - 6.5.2. *Tworzenie metody pośredniczącej router()* 166
 - 6.5.3. *Tworzenie metody pośredniczącej przeznaczonej do przepisywania adresów URL* 168
- 6.6. Użycie metody pośredniczącej do obsługi błędów 170
 - 6.6.1. *Domyslna obsługa błędów w Connect* 170
 - 6.6.2. *Samodzielna obsługa błędów aplikacji* 171
 - 6.6.3. *Użycie wielu metod pośredniczących przeznaczonych do obsługi błędów* 172
- 6.7. Podsumowanie 176

Rozdział 7. Metody pośredniczące frameworka Connect 177

- 7.1. Metody pośredniczące przeznaczone do przetwarzania plików cookie, danych żądań i ciągów tekstowych zapytań 179
 - 7.1.1. *cookieParser()* — przetwarzanie plików cookie 179
 - 7.1.2. *bodyParser()* — przetwarzanie danych żądania 182
 - 7.1.3. *limit()* — ograniczenie danych żądania 184
 - 7.1.4. *query()* — analizator ciągu tekstowego zapytania 186
- 7.2. Metody pośredniczące implementujące podstawowe funkcje wymagane przez aplikację sieciową 187
 - 7.2.1. *logger()* — rejestracja informacji o żądaniu 188
 - 7.2.2. *favicon()* — obsługa ikon favicon 191
 - 7.2.3. *methodOverride()* — nieprawdziwe metody HTTP 191
 - 7.2.4. *ghost()* — wirtualny hosting 194
 - 7.2.5. *session()* — zarządzanie sesją 195

- 7.3. Metody pośredniczące zapewniające bezpieczeństwo aplikacji sieciowej 200
 - 7.3.1. *basicAuth()* — uwierzytelnianie podstawowe HTTP 200
 - 7.3.2. *csrf()* — ochrona przed atakami typu CSRF 202
 - 7.3.3. *errorHandler()* — obsługa błędów w trakcie tworzenia aplikacji 203
- 7.4. Metody pośredniczące przeznaczone do udostępniania plików statycznych 205
 - 7.4.1. *static()* — udostępnianie plików statycznych 205
 - 7.4.2. *compress()* — kompresja plików statycznych 207
 - 7.4.3. *directory()* — wyświetlenie katalogów 209
- 7.5. Podsumowanie 210

Rozdział 8. Framework Express 213

- 8.1. Utworzenie szkieletu aplikacji 215
 - 8.1.1. Globalna instalacja frameworka Express 216
 - 8.1.2. Generowanie aplikacji 218
 - 8.1.3. Poznanie aplikacji 218
- 8.2. Konfiguracja frameworka Express i tworzonej aplikacji 220
 - 8.2.1. Konfiguracja na podstawie środowiska 221
- 8.3. Generowanie widoków aplikacji Express 223
 - 8.3.1. Konfiguracja systemu widoków 224
 - 8.3.2. Wyszukiwanie widoku 225
 - 8.3.3. Udostępnianie danych widokom 228
- 8.4. Obsługa formularzy i przekazywania plików 232
 - 8.4.1. Implementacja modelu zdjęcia 233
 - 8.4.2. Tworzenie formularza przeznaczonego do przekazywania zdjęć 233
 - 8.4.3. Wyświetlenie listy przekazanych zdjęć 236
- 8.5. Obsługa pobierania zasobów 237
 - 8.5.1. Tworzenie trasy dla pobierania zdjęć 237
 - 8.5.2. Implementacja trasy pobierania zdjęcia 238
- 8.6. Podsumowanie 240

Rozdział 9. Zaawansowane użycie frameworka Express 241

- 9.1. Uwierzytelnianie użytkowników 242
 - 9.1.1. Zapisywanie i wczytywanie użytkowników 243
 - 9.1.2. Rejestrowanie nowego użytkownika 248
 - 9.1.3. Logowanie zarejestrowanych użytkowników 254
 - 9.1.4. Metoda pośrednicząca przeznaczona do wczytywania użytkownika 257
- 9.2. Zaawansowane techniki routingu 259
 - 9.2.1. Weryfikacja użytkownika podczas przesyłania treści 260
 - 9.2.2. Metoda pośrednicząca charakterystyczna dla trasy 263
 - 9.2.3. Implementacja stronicowania 266
- 9.3. Utworzenie publicznego API REST 270
 - 9.3.1. Projekt API 270
 - 9.3.2. Dodanie uwierzytelnienia podstawowego 271
 - 9.3.3. Implementacja routingu 272
 - 9.3.4. Włączenie negocjacji treści 275
- 9.4. Obsługa błędów 277
 - 9.4.1. Obsługa błędów 404 278
 - 9.4.2. Obsługa błędów 280
- 9.5. Podsumowanie 283

Rozdział 10. Testowanie aplikacji Node 285

- 10.1. Testy jednostkowe 286
 - 10.1.1. Moduł *assert* 287
 - 10.1.2. Framework *nodeunit* 291
 - 10.1.3. *Mocha* 293
 - 10.1.4. Framework *Vows* 298
 - 10.1.5. Biblioteka *should.js* 301
- 10.2. Testy akceptacyjne 303
 - 10.2.1. *Tobi* 303
 - 10.2.2. *Soda* 305
- 10.3. Podsumowanie 307

Rozdział 11. Szablony w aplikacji sieciowej 309

- 11.1. Użycie szablonów w celu zachowania przejrzystości kodu 310
 - 11.1.1. Szablon w akcji 311
- 11.2. Silnik szablonów *Embedded JavaScript* 314
 - 11.2.1. Tworzenie szablonu 315
 - 11.2.2. Praca z danymi szablonu za pomocą filtrów *EJS* 316
 - 11.2.3. Integracja *EJS* w aplikacji 320
 - 11.2.4. Użycie *EJS* w aplikacjach działających po stronie klienta 321
- 11.3. Użycie języka szablonów *Mustache* wraz z silnikiem *Hogan* 322
 - 11.3.1. Tworzenie szablonu 322
 - 11.3.2. Znaczniki *Mustache* 323
 - 11.3.3. Dostosowanie szablonu *Hogan* do własnych potrzeb 325
- 11.4. Szablony *Jade* 326
 - 11.4.1. Podstawy szablonów *Jade* 328
 - 11.4.2. Logika w szablonach *Jade* 330
 - 11.4.3. Organizacja szablonów *Jade* 333
- 11.5. Podsumowanie 337

CZĘŚĆ III. CO DALEJ? 339**Rozdział 12. Wdrażanie aplikacji Node
i zapewnienie bezawaryjnego działania 341**

- 12.1. Hosting aplikacji Node 342
 - 12.1.1. Serwery dedykowane i VPS 343
 - 12.1.2. Hosting w chmurze 343
- 12.2. Podstawy wdrożenia 346
 - 12.2.1. Wdrożenie z repozytorium *Git* 346
 - 12.2.2. Zapewnienie działania aplikacji Node 347
- 12.3. Maksymalizacja wydajności i czasu bezawaryjnego działania aplikacji 348
 - 12.3.1. Zapewnienie działania aplikacji za pomocą *Upstart* 349
 - 12.3.2. API klastra — wykorzystanie zalety w postaci wielu rdzeni 351
 - 12.3.3. Proxy i hosting plików statycznych 353
- 12.4. Podsumowanie 354

Rozdział 13. Nie tylko serwery WWW 355

- 13.1. Biblioteka `Socket.IO` 356
 - 13.1.1. Tworzenie minimalnej aplikacji `Socket.IO` 357
 - 13.1.2. Użycie biblioteki `Socket.IO` do odświeżenia strony i stylów CSS 359
 - 13.1.3. Inne zastosowania dla biblioteki `Socket.IO` 362
- 13.2. Dokładniejsze omówienie sieci TCP/IP 363
 - 13.2.1. Praca z buforami i danymi binarnymi 363
 - 13.2.2. Tworzenie serwera TCP 365
 - 13.2.3. Tworzenie klienta TCP 369
- 13.3. Narzędzia przeznaczone do pracy z systemem operacyjnym 371
 - 13.3.1. Obiekt `process`, czyli globalny wzorzec `Singleton` 371
 - 13.3.2. Użycie modułu `filesystem` 375
 - 13.3.3. Tworzenie procesów zewnętrznych 379
- 13.4. Tworzenie narzędzi powłoki 384
 - 13.4.1. Przetwarzanie argumentów podanych w powłoce 385
 - 13.4.2. Praca ze standardowym wejściem i wyjściem 386
 - 13.4.3. Dodanie koloru do danych wyjściowych 388
- 13.5. Podsumowanie 391

Rozdział 14. Ekosystem Node 393

- 14.1. Dostępne w internecie zasoby dla programistów Node 394
 - 14.1.1. Node i odniesienia do modułów 394
 - 14.1.2. Grupy Google 395
 - 14.1.3. IRC 396
 - 14.1.4. Zgłaszanie problemów w serwisie `GitHub` 397
- 14.2. Serwis `GitHub` 398
 - 14.2.1. Rozpoczęcie pracy z `GitHub` 398
 - 14.2.2. Dodanie projektu do `GitHub` 399
 - 14.2.3. Współpraca przez serwis `GitHub` 403
- 14.3. Przekazanie własnego modułu do repozytorium `npm` 405
 - 14.3.1. Przygotowanie pakietu 406
 - 14.3.2. Przygotowanie specyfikacji pakietu 406
 - 14.3.3. Testowanie i publikowanie pakietu 407
- 14.4. Podsumowanie 409

Dodatek A. Instalacja Node i dodatki opracowane przez społeczność 411

- A.1. Instalacja w systemie OS X 411
 - A.1.1. Instalacja za pomocą `Homebrew` 412
- A.2. Instalacja w systemie Windows 413
- A.3. Instalacja w systemie Linux 414
 - A.3.1. Przygotowania do instalacji w `Ubuntu` 414
 - A.3.2. Przygotowania do instalacji w `CentOS` 415
- A.4. Kompilacja Node 415
- A.5. Używanie menedżera pakietów Node 416
 - A.5.1. Wyszukiwanie pakietów 417
 - A.5.2. Instalacja pakietu 418
 - A.5.3. Przeglądanie dokumentacji i kodu pakietu 419

Dodatek B. Debugowanie Node 421

- B.1. Analiza kodu za pomocą JSHint 421
- B.2. Dane wyjściowe debugowania 422
 - B.2.1. Debugowanie za pomocą modułu console 422
 - B.2.2. Użycie modułu debug do zarządzania danymi wyjściowymi procesu debugowania 423
- B.3. Debugger wbudowany w Node 424
 - B.3.1. Nawigacja po debuggerze 424
 - B.3.2. Analiza i zmiana stanu w debuggerze 425
- B.4. Inspektor Node 426
 - B.4.1. Uruchomienie inspektora Node 426
 - B.4.2. Nawigacja po inspektorze Node 426
 - B.4.3. Przeglądanie stanu w inspektorze Node 427

Dodatek C. Rozszerzenie i konfiguracja frameworka Express 429

- C.1. Rozszerzenie frameworka Express 429
 - C.1.1. Rejestracja szablonów silników 429
 - C.1.2. Szablony i projekt consolidate.js 430
 - C.1.3. Frameworki i rozszerzenia Express 431
- C.2. Konfiguracja zaawansowana 432
 - C.2.1. Modyfikacja odpowiedzi JSON 433
 - C.2.2. Formatowanie odpowiedzi JSON 433

Skorowidz 435

2

Tworzenie aplikacji wielopokojowego czatu

W tym rozdziale:

- Pierwsze spojrzenie na różne komponenty Node.
- Przykład aplikacji Node działającej w czasie rzeczywistym.
- Współpraca między klientem i serwerem.

W rozdziale 1. dowiedziałeś się, jak programowanie asynchroniczne z użyciem Node różni się od konwencjonalnego programowania synchronicznego. W tym rozdziale wykorzystamy platformę Node w praktyce do utworzenia małej, opartej na zdarzeniach aplikacji czatu. Nie przejmuj się, jeśli nie zrozumiesz całego materiału przedstawionego w rozdziale. Naszym celem jest objaśnienie sposobu programowania z użyciem Node i jedynie zaprezentowanie możliwości, jakie będziesz mieć po zakończeniu lektury niniejszej książki.

W rozdziale przyjęto założenie, że masz doświadczenie w programowaniu aplikacji sieciowych, a także podstawową wiedzę z zakresu HTTP i biblioteki jQuery. W trakcie lektury materiału przedstawionego w tym rozdziale:

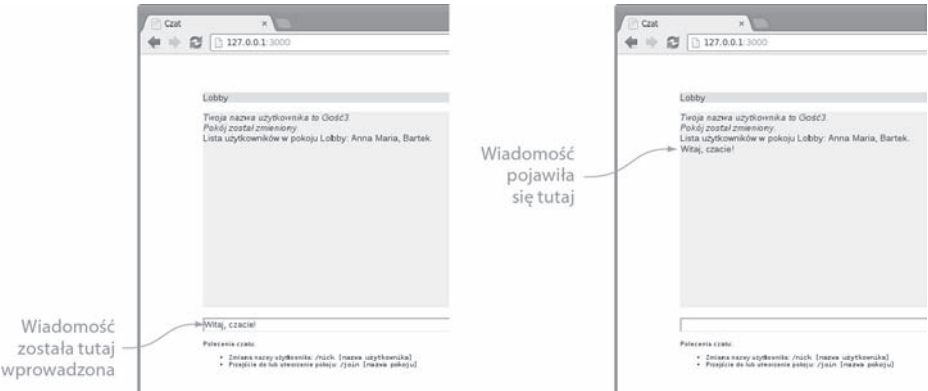
- poznasz tworzoną aplikację i zobaczysz, jak działa;
- poznasz technologie wymagane do jej utworzenia i przeprowadzisz początkową konfigurację aplikacji;
- przygotujesz dla aplikacji kod HTML, CSS i JavaScript działający po stronie klienta;

- zajmiesz się obsługą wiadomości czatu za pomocą biblioteki Socket.IO;
- za pomocą kodu JavaScript działającego po stronie klienta przygotujesz interfejs użytkownika aplikacji.

Zaczynamy od ogólnego omówienia aplikacji — dowiesz się, jak aplikacja będzie wyglądać i działać, gdy zakończymy proces jej tworzenia.

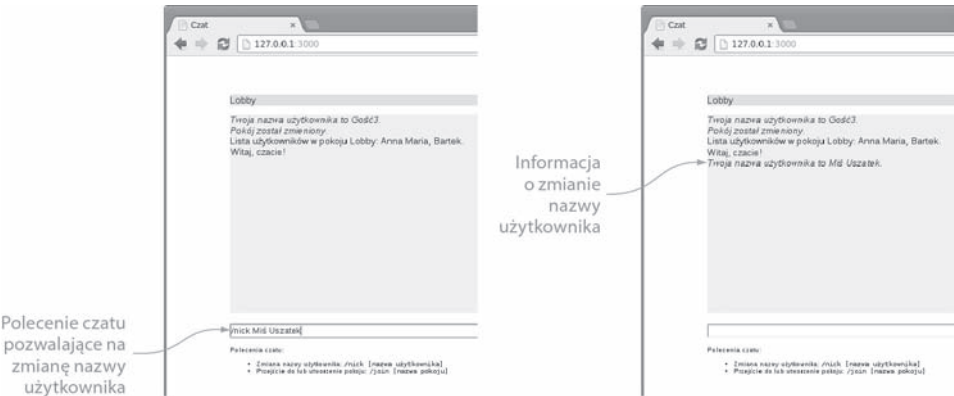
2.1. Ogólny opis aplikacji

Aplikacja tworzona w tym rozdziale pozwala użytkownikom na prowadzenie między sobą internetowego czatu przez wprowadzanie wiadomości w prostym formularzu, jak pokazano na rysunku 2.1. Wprowadzona w formularzu wiadomość zostaje wysłana wszystkim uczestnikom czatu znajdującym się w tym samym pokoju.



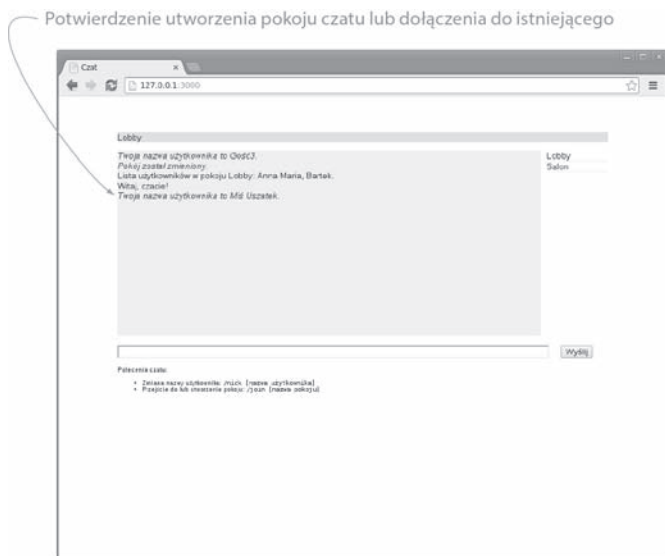
Rysunek 2.1. Wprowadzenie wiadomości w aplikacji czatu

Po uruchomieniu aplikacji użytkownikowi automatycznie będzie przypisana nazwa gościa, którą może zmienić przez wydanie polecenia, jak pokazano na rysunku 2.2. Polecenia czatu są poprzedzane ukośnikiem (/).



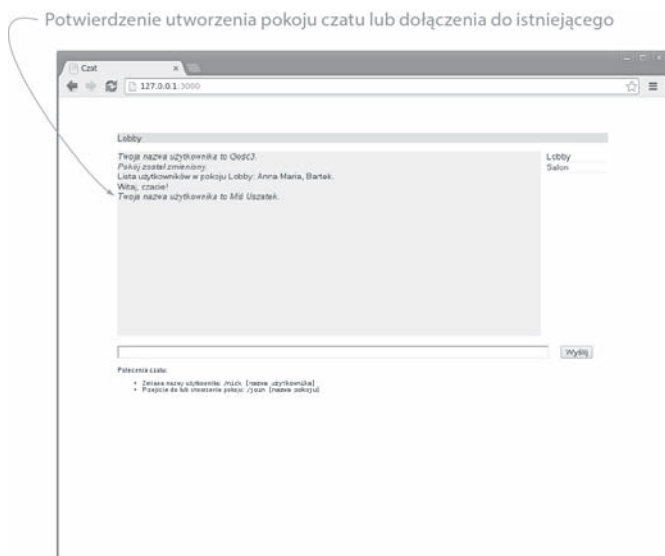
Rysunek 2.2. Zmiana nazwy użytkownika czatu

Podobnie użytkownik może wydać polecenie utworzenia nowego pokoju czatu (lub dołączenia do istniejącego), co pokazano na rysunku 2.3. Podczas tworzenia pokoju czatu nazwa nowego pokoju będzie wyświetlana na poziomym pasku znajdującym się na górze aplikacji czatu. Sam pokój zostanie również umieszczony na liście dostępnych pokoi wyświetlanej po prawej stronie obszaru wiadomości.



Rysunek 2.3. Zmiana pokoju czatu

Po przejściu użytkownika do nowego pokoju system potwierdzi tę zmianę, jak pokazano na rysunku 2.4.



Rysunek 2.4. Wynik przejścia do nowego pokoju czatu

Wprawdzie funkcjonalność omawianej tutaj aplikacji bez wątpienia jest bardzo ograniczona, ale jednocześnie prezentuje ona ważne i podstawowe koncepcje wymagane do utworzenia aplikacji sieciowej działającej w czasie rzeczywistym. Omawiania aplikacji pokazuje, jak Node może oferować dane HTTP (na przykład pliki statyczne) i jednocześnie obsługiwać dane w czasie rzeczywistym (wiadomości czatu). Ponadto dzięki omawianej aplikacji możesz się przekonać, jak zorganizowane są aplikacje Node i jak wygląda zarządzanie zależnościami.

Teraz przejdziemy do technologii wymaganych do implementacji aplikacji czatu.

2.2. Wymagania aplikacji i konfiguracja początkowa

Tworzona w rozdziale aplikacja musi oferować:

- Udostępnianie plików statycznych (takich jak HTML, CSS i skrypty JavaScript działające po stronie klienta).
- Obsługę przez serwer wiadomości czatu.
- Obsługę wiadomości czatu w przeglądarce internetowej użytkownika.

Aby udostępniać pliki statyczne, wykorzystamy moduł `http` wbudowany w Node. Jednak podczas udostępniania plików za pomocą protokołu HTTP zwykle nie wystarczy jedynie wysłać zawartość pliku. Konieczne jest również wskazanie rodzaju wysyłanego pliku. Odbywa się to przez ustawienie nagłówka `Content-Type` protokołu HTTP i podanie właściwego typu MIME dla pliku. W celu wyszukania wspomnianych typów MIME trzeba będzie użyć modułu o nazwie `mime` opracowanego przez firmę trzecią.

TYPY MIME. Dokładne omówienie typów MIME znajdziesz w artykule Wikipedii: http://pl.wikipedia.org/wiki/Multipurpose_Internet_Mail_Extensions.

Do obsługi wiadomości związanych z czatem można wykorzystać technologię Ajax. Jednak aby zachować jak największy stopień reakcji aplikacji na działania użytkownika, konieczne jest uniknięcie tradycyjnego rozwiązania Ajax stosowanego do wysyłania wiadomości. Ajax używa protokołu HTTP jako mechanizmu transportowego, a wspomniany HTTP nie został opracowany do prowadzenia komunikacji w czasie rzeczywistym. Kiedy wiadomość jest wysyłana za pomocą HTTP, konieczne jest użycie nowego połączenia TCP/IP. Otwieranie i zamykanie połączeń zabiera cenny czas, a ilość przekazywanych danych rośnie, ponieważ każde żądanie zawiera nagłówki HTTP. Zamiast implementować rozwiązanie oparte na HTTP, w omawianej aplikacji wykorzystamy technologię WebSocket (<http://pl.wikipedia.org/wiki/WebSocket>). Została ona zaprojektowana do prowadzenia dwukierunkowej, lekkiej komunikacji w czasie rzeczywistym za pomocą jednego gniazda TCP.

Ponieważ w większości przypadków jedynie przeglądarki zgodne z HTML5 obsługują WebSocket, w aplikacji użyjemy popularnej biblioteki `Socket.IO` (<http://socket.io/>). Ta biblioteka oferuje wiele rozwiązań awaryjnych, między innymi użycie technologii Flash, gdy zastosowanie WebSocket okaże się niemożliwe. Wspomniane rozwiązania

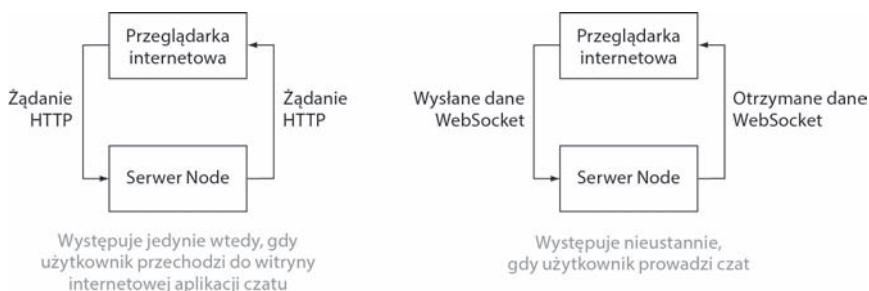
awaryjne są automatycznie obsługiwane przez bibliotekę Socket.IO i nie wymagają dodatkowego kodu lub konfiguracji. Dokładniejsze omówienie biblioteki Socket.IO znajdziesz w rozdziale 13.

Zanim faktycznie zajmiemy się pracą i przystąpimy do początkowej konfiguracji struktury plików i zależności aplikacji, warto dowiedzieć się, jak Node pozwala na jednoczesną obsługę HTTP i WebSocket. To jeden z powodów, dla których platforma Node jest doskonałym wyborem w przypadku aplikacji działających w czasie rzeczywistym.

2.2.1. Obsługa HTTP i WebSocket

Wprawdzie w omawianej aplikacji unikamy użycia technologii Ajax do wysyłania i otrzymywania wiadomości czatu, ale nadal korzystamy z HTTP do dostarczania plików HTML, CSS i kodu JavaScript działającego po stronie klienta, które powodują przygotowanie aplikacji w przeglądarce internetowej użytkownika.

Jak pokazano na rysunku 2.5, Node może bardzo łatwo jednocześnie obsługiwać HTTP i WebSocket za pomocą pojedynczego portu TCP/IP. Standardowo zawiera moduł zapewniający obsługę funkcjonalności HTTP. Dla Node firmy trzecie opracowały również wiele innych modułów, na przykład Express, które zostały zbudowane w oparciu o standardowe funkcje oferowane przez Node i pozwalają na jeszcze łatwiejsze udostępnianie treści. Więcej informacji dotyczących użycia modułu Express podczas tworzenia aplikacji sieciowych znajdziesz w rozdziale 8. W aplikacji tworzonej w tym rozdziale wykorzystamy standardowe możliwości Node.



Rysunek 2.5. Obsługa HTTP i WebSocket w jednej aplikacji

Skoro dowiedziałeś się już, jakie podstawowe technologie będą wykorzystane w budowanej aplikacji, możemy przystąpić do pracy.

Czy musisz zainstalować Node?

Jeżeli jeszcze nie zainstalowałeś Node, dokładne omówienie instalacji znajdziesz w dodatku A.

2.2.2. Tworzenie struktury plików aplikacji

Pierwszym krokiem podczas budowy omawianej aplikacji jest utworzenie katalogu dla projektu. Główny plik aplikacji będzie umieszczony we wspomnianym katalogu. Konieczne jest również dodanie podkatalogu *lib*, w którym znajdzie się logika działająca

po stronie serwera. Ponadto trzeba utworzyć podkatalog *public* przeznaczony dla plików używanych po stronie klienta. Następnie w podkatalogu *public* utwórz dwa kolejne: *javascripts* i *stylesheets*.

Struktura katalogów powinna wyglądać tak, jak pokazano na rysunku 2.6. Warto w tym miejscu dodać, że choć zdecydowaliśmy się na organizację plików w przedstawiony sposób, to jednak Node nie wymaga stosowania żadnej struktury plików. Pliki składające się na aplikację możesz umieścić w najbardziej odpowiadający Ci sposób.

Po przygotowaniu struktury katalogów możemy przystąpić do zdefiniowania zależności aplikacji.

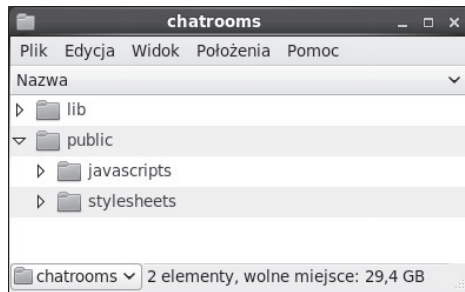
W omawianym kontekście *zależność aplikacji* oznacza moduły konieczne do zainstalowania, aby dostarczyć funkcje wymagane przez aplikację. Na przykład przyjmujemy założenie o tworzeniu aplikacji wymagającej dostępu do danych przechowywanych w bazie danych MySQL. Node nie jest standardowo wyposażone w moduł pozwalający na uzyskanie dostępu do MySQL. Konieczna jest więc instalacja modułu opracowanego przez firmę trzecią i wspomniany moduł jest wówczas zależnością.

2.2.3. Wskazanie zależności

Wprawdzie istnieje możliwość utworzenia aplikacji Node bez wskazywania zależności, ale dobrym nawykiem jest poświęcenie chwili czasu na ich zdefiniowanie. W ten sposób konfiguracja aplikacji będzie nieco łatwiejsza, jeśli inny użytkownik będzie chciał używać tej aplikacji lub jeśli planujesz jej uruchamianie w więcej niż tylko jednym miejscu.

Zależności aplikacji są definiowane w pliku o nazwie *package.json*. Wymieniony plik zawsze powinien znajdować się w katalogu głównym aplikacji. Zawartość pliku *package.json* to wyrażenie JSON w standardzie opisu pakietu CommonJS (<http://wiki.commonjs.org/wiki/Packages/1.0>) opisujące aplikację. W pliku *package.json* można podać wiele informacji, ale najważniejsze to nazwa aplikacji, wersja, opis jej działania oraz zależności aplikacji.

W listingu 2.1 przedstawiono plik opisujący funkcjonalność i zależności aplikacji tworzonej w tym rozdziale. Plik zapisz pod nazwą *package.json* w katalogu głównym aplikacji.



Rysunek 2.6. Struktura katalogu projektu dla aplikacji czatu

Listing 2.1. Plik opisujący aplikację

```
{
  "name": "chatrooms",
  "version": "0.0.1",
  "description": "Minimalistyczny serwer wielopokojowego czatu",
  "dependencies": {
    "socket.io": "~0.9.6",
  }
}
```

← Nazwa pakietu.

← Zależności pakietu.


```

    "mime": "~1.2.7"
  }
}

```

Jeżeli zawartość pliku wydaje Ci się nieco dziwna, nie przejmuj się tym teraz. Więcej informacji dotyczących pliku *package.json* znajdziesz w następnym rozdziale, a jego dokładne omówienie w rozdziale 14.

2.2.4. Instalacja zależności

Po zdefiniowaniu pliku *package.json* instalacja zależności aplikacji staje się niezwykle łatwym zadaniem. Menedżer pakietów Node (<https://github.com/npm/npm>) jest dostarczany standardowo wraz z Node. Oferuje doskonałe funkcje, między innymi możliwość łatwej instalacji modułów Node opracowanych przez firmy trzecie oraz globalnego udostępniania modułów opracowanych przez Ciebie. Ponadto za pomocą pojedynczego polecenia potrafi odczytać zależności z pliku *package.json*, a następnie je zainstalować.

Z poziomu katalogu głównego tworzonej aplikacji wydaj poniższe polecenie:

```
npm install
```

Jeżeli teraz zajrzysz do katalogu aplikacji, przekonasz się, że jest w nim nowy podkatalog o nazwie *node_modules*, jak pokazano na rysunku 2.7. Wymieniony podkatalog zawiera zależności aplikacji.

Mając przygotowaną strukturę katalogu aplikacji i zainstalowane zależności, można już przystąpić do tworzenia logiki aplikacji.



Rysunek 2.7. Po użyciu menedżera npm do instalacji zależności zostanie utworzony podkatalog o nazwie *node_modules*

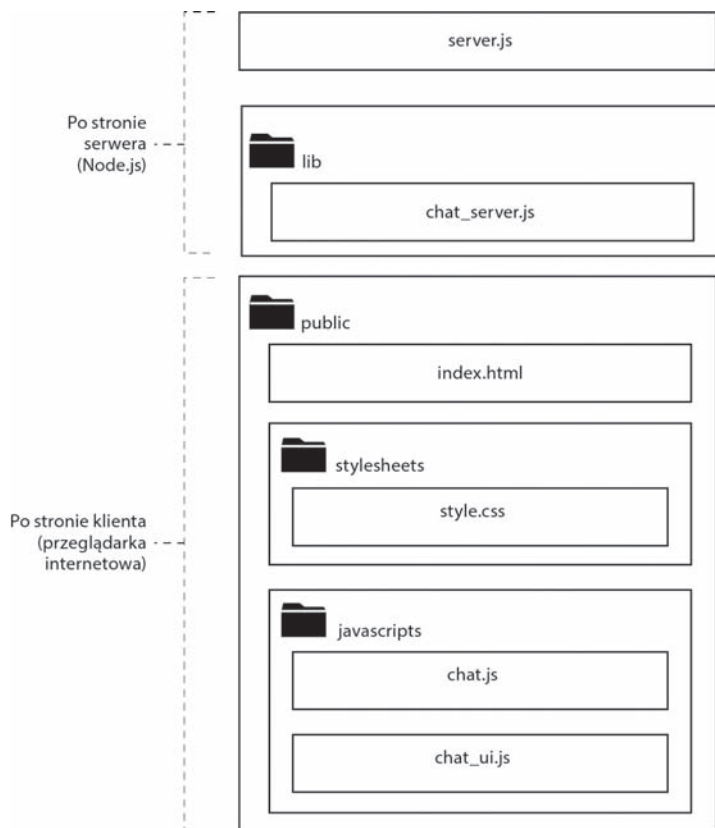
2.3. Udostępnianie plików HTML, CSS i kodu JavaScript działającego po stronie klienta

Jak wcześniej wspomniano, budowana tutaj aplikacja czatu powinna oferować wymienione poniżej możliwości:

- Udostępnianie plików statycznych przeglądarce internetowej użytkownika.
- Obsługę przez serwer wiadomości związanych z czatem.
- Obsługę wiadomości czatu w przeglądarce internetowej użytkownika.

Logika aplikacji będzie obsługiwana przez wiele plików, część wykorzystywanych po stronie serwera, inne po stronie klienta, jak pokazano na rysunku 2.8. Pliki kodu JavaScript działającego po stronie klienta muszą być udostępniane w postaci zasobów statycznych, a nie przetwarzane przez Node.

W tym podrozdziale zajmiemy się spełnieniem pierwszego wymagania stawianego aplikacji: zdefiniujemy logikę niezbędną do udostępniania plików statycznych. Następnie dodamy pliki statyczne z kodem HTML i CSS.



Rysunek 2.8.
W budowanej aplikacji czatu po stronie zarówno serwera, jak i klienta istnieje logika JavaScript

2.3.1. Tworzenie podstawowego serwera plików statycznych

Aby utworzyć serwer plików statycznych, konieczne jest wykorzystanie pewnych wbudowanych funkcji Node, a także opracowanego przez firmę trzecią modułu `mime` w celu ustalenia typu MIME danego pliku.

Rozpoczynamy od głównego pliku aplikacji. W katalogu głównym projektu utwórz plik o nazwie `server.js` i umieść w nim deklaracje zmiennych przedstawione w listingu 2.2. Wspomniane deklaracje pozwalają na uzyskanie dostępu do funkcji HTTP w Node, a także na pracę z systemem plików, użycie funkcji związanych ze ścieżkami dostępu do plików oraz możliwość ustalenia typu MIME danego pliku. Zmienną `cache` wykorzystamy do buforowania danych pliku.

Listing 2.2. Deklaracje zmiennych

```
var http = require('http');
var fs = require('fs');
var path = require('path');
var mime = require('mime');
var cache = {};
```

- ← **Wbudowany moduł http** dostarcza funkcje dotyczące serwera i klienta.
- ← **Wbudowany moduł fs** dostarcza funkcje przeznaczone do pracy z systemem plików.
- ← **Wbudowany moduł path** dostarcza funkcje przeznaczone do pracy ze ścieżkami dostępu systemu plików.
- ← **Dodatkowy moduł mime** zapewnia możliwość ustalenia typu MIME na podstawie rozszerzenia pliku.
- ← **Obiekt cache służy do przechowywania buforowanych plików.**

WYSYŁANIE DANYCH PLIKU I ODPOWIEDZI W POSTACI BŁĘDÓW

Kolejnym krokiem jest utworzenie trzech funkcji pomocniczych używanych do udostępniania statycznych plików HTTP. Pierwsza powoduje wygenerowanie błędu o kodzie 404, jeśli żądany plik nie istnieje. W pliku *server.js* umieść więc poniższy kod funkcji pomocniczej:

```
function send404(response) {
  response.writeHead(404, {'Content-Type': 'text/plain'});
  response.write('Błąd 404: plik nie został znaleziony.');
```

Druga funkcja pomocnicza dostarcza dane pliku. Najpierw przygotowuje odpowiednie nagłówki HTTP, a następnie wysyła zawartość pliku. W pliku *server.js* umieść więc poniższy kod:

```
function sendFile(response, filePath, fileContents) {
  response.writeHead(
    200,
    {"content-type": mime.lookup(path.basename(filePath))}
  );
  response.end(fileContents);
}
```

Uzyskanie dostępu do pamięci operacyjnej (RAM) jest szybsze niż do systemu plików. Dlatego też aplikacje Node buforują w pamięci często używane dane. Budowana tutaj aplikacja czatu będzie buforowała w pamięci pliki statyczne i odczyta je z dysku tylko podczas pierwszego ich żądania. Trzecia funkcja pomocnicza sprawdza więc, czy plik jest buforowany, a następnie go udostępnia. Jeżeli plik nie jest jeszcze buforowany, wtedy zostanie odczytany z dysku i udostępniony. Jeżeli plik nie istnieje, odpowiedzią będzie kod błędu HTTP 404. W pliku *server.js* umieść więc kod funkcji pomocniczej przedstawionej w listingu 2.3.

Listing 2.3. Funkcja pomocnicza udostępniająca pliki statyczne

```
function serveStatic(response, cache, absPath) {
  if (cache[absPath]) {
    sendFile(response, absPath, cache[absPath]);
  } else {
    fs.exists(absPath, function(exists) {
      if (exists) {
        fs.readFile(absPath, function(err, data) {
          if (err) {
            send404(response);
          } else {
            cache[absPath] = data;
            sendFile(response, absPath, data);
          }
        });
      } else {
        send404(response);
      }
    });
  }
}
```

← Sprawdzenie, czy plik jest buforowany w pamięci.
 ← Udostępnienie pliku z pamięci.
 ← Sprawdzenie, czy plik istnieje.
 ← Odczyt pliku z dysku.
 ← Udostępnienie pliku odczytanego z dysku.
 ← Wysłanie odpowiedzi HTTP 404.

```

    });
  }
}

```

TWORZENIE SERWERA HTTP

W przypadku serwera HTTP funkcja anonimowa dostarczana jako argument funkcji `createServer()` działa w charakterze wywołania definiującego sposób obsługi poszczególnych żądań HTTP. Funkcja wywołania zwrótnego akceptuje dwa argumenty: `request` i `response`. Podczas jej wywołania serwer HTTP wypełni wspomniane argumenty obiektami, które pozwolą na odpowiednio odczyt szczegółów żądania oraz przygotowanie odpowiedzi. Więcej informacji o module `http` Node znajdziesz w rozdziale 4.

W pliku `server.js` umieść kod przedstawiony w listingu 2.4, odpowiedzialny za utworzenie serwera HTTP.

Listing 2.4. Logika tworząca serwer HTTP

```

var server = http.createServer(function(request, response) {
  var filePath = false;

  if (request.url == '/') {
    filePath = 'public/index.html';
  } else {
    filePath = 'public' + request.url;
  }

  var absPath = './' + filePath;
  serveStatic(response, cache, absPath);
});

```

← **Utworzenie serwera HTTP za pomocą funkcji anonimowej definiującej zachowanie w poszczególnych żądaniach.**

← **Wskazanie pliku HTML, który ma być domyślnie udostępniany.**

← **Zamiana adresu URL na względną ścieżkę dostępu do pliku.**

← **Udostępnienie pliku statycznego.**

URUCHOMIENIE SERWERA HTTP

W kodzie utworzyliśmy serwer HTTP, ale nie dodaliśmy jeszcze logiki niezbędnej do jego uruchomienia. Poniższy fragment kodu powoduje uruchomienie serwera i nasłuchiwanie TCP/IP na porcie 3000. Port 3000 został wybrany dowolnie, można skorzystać z każdego nieużywanego portu o numerze większym niż 1024 (port 1024 również może działać, jeśli korzystasz z systemu Windows, natomiast w systemach Linux i OS X konieczne jest wówczas uruchomienie aplikacji przez użytkownika uprzywilejowanego, na przykład `root`).

```

server.listen(3000, function() {
  console.log("Serwer nasłuchuje na porcie 3000.");
});

```

Jeżeli chcesz się przekonać, jak aplikacja działa na tym etapie, to możesz uruchomić serwer przez wydanie poniższego polecenia w wierszu poleceń:

```
node server.js
```

Po uruchomieniu serwera przejście pod adres `http://127.0.0.1:3000` w przeglądarce internetowej spowoduje wywołanie funkcji pomocniczej generującej kod błędu 404, a więc wyświetlenie komunikatu Błąd 404: plik nie został znaleziony. Wprawdzie aplikacja

zawiera logikę odpowiedzialną za obsługę plików statycznych, ale jeszcze nie dodaliśmy żadnego tego rodzaju pliku. Warto w tym miejscu wspomnieć, że działanie serwera można zatrzymać przez naciśnięcie klawiszy *Ctrl+C* w powłoce.

Przechodzimy teraz do dodania plików statycznych zapewniających nieco większą funkcjonalność aplikacji czatu.

2.3.2. Dodanie plików HTML i CSS

Pierwszy dodawany plik statyczny zawiera kod HTML. W katalogu *public* utwórz plik o nazwie *index.html*, a następnie umieść w nim kod HTML przedstawiony w listingu 2.5. Wspomniany kod HTML powoduje dołączenie pliku arkusza stylów (CSS), zdefiniowanie pewnych elementów HTML `<div>` przeznaczonych do wyświetlania treści aplikacji, a także do wczytania kilku plików zawierających kod JavaScript działający po stronie klienta. Pliki JavaScript zapewniają dostęp do funkcji biblioteki Socket.IO, biblioteki jQuery (w celu łatwiejszej pracy z modelem DOM), a także oferują przygotowane specjalnie dla budowanej aplikacji funkcje obsługi czatu.

Listing 2.5. Kod HTML aplikacji czatu

```
<!doctype html>
<html lang='en'>

<head>
  <title>Czat</title>
  <link rel='stylesheet' href='/stylesheets/style.css'></link>
</head>

<body>
<div id='content'>
  <div id='room'></div>
  <div id='room-list'></div>
  <div id='messages'></div>
  <form id='send-form'>
    <input id='send-message' />
    <input id='send-button' type='submit' value='Wyślij' />
  </form>
  <div id='help'>
    Polecenia czatu:
    <ul>
      <li>Zmiana nazwy użytkownika: <code>/nick [nazwa użytkownika]</code></li>
      <li>Przejsście do lub utworzenie pokoju: <code>/join [nazwa pokoju]</code></li>
    </ul>
  </div>
</form>
</div>

<script src='/socket.io/socket.io.js' type='text/javascript'></script>
<script src='http://code.jquery.com/jquery-1.8.0.min.js' type='text/javascript'></script>
<script src='/javascripts/chat.js' type='text/javascript'></script>
<script src='/javascripts/chat_ui.js' type='text/javascript'></script>
</body>
</html>
```

Element `<div>`, w którym będzie wyświetlona nazwa aktualnego pokoju czatu.

Element `<div>`, w którym będzie wyświetlona lista dostępnych pokoi czatu.

Element `<div>`, w którym będą wyświetlone wiadomości czatu.

Element `<input>` formularza, w którym użytkownik będzie wydawał polecenia i wpisywał wiadomości.

Kolejny plik, który trzeba dodać, zawiera style CSS używane w aplikacji. W katalogu *public/stylesheets* utwórz plik o nazwie *style.css*, a następnie umieść w nim kod CSS przedstawiony w listingu 2.6.

Listing 2.6. Kod CSS używany przez aplikację

```
body {  
  padding: 50px;  
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;  
}  
  
a {  
  color: #00B7FF;  
}  
  
#content {  
  width: 800px;  
  margin-left: auto;  
  margin-right: auto;  
}  
  
#room {  
  background-color: #ddd;  
  margin-bottom: 1em;  
}  
  
#messages {  
  width: 690px;  
  height: 300px;  
  overflow: auto;  
  background-color: #eee;  
  margin-bottom: 1em;  
  margin-right: 10px;  
}
```

← Aplikacja będzie miała szerokość 800 pikseli i zostanie wyśrodkowana poziomo.

← Reguła CSS dla elementu, w którym wyświetlana jest nazwa aktualnego pokoju czatu.

← Element wiadomości ma szerokość 690 pikseli i wysokość 300 pikseli.

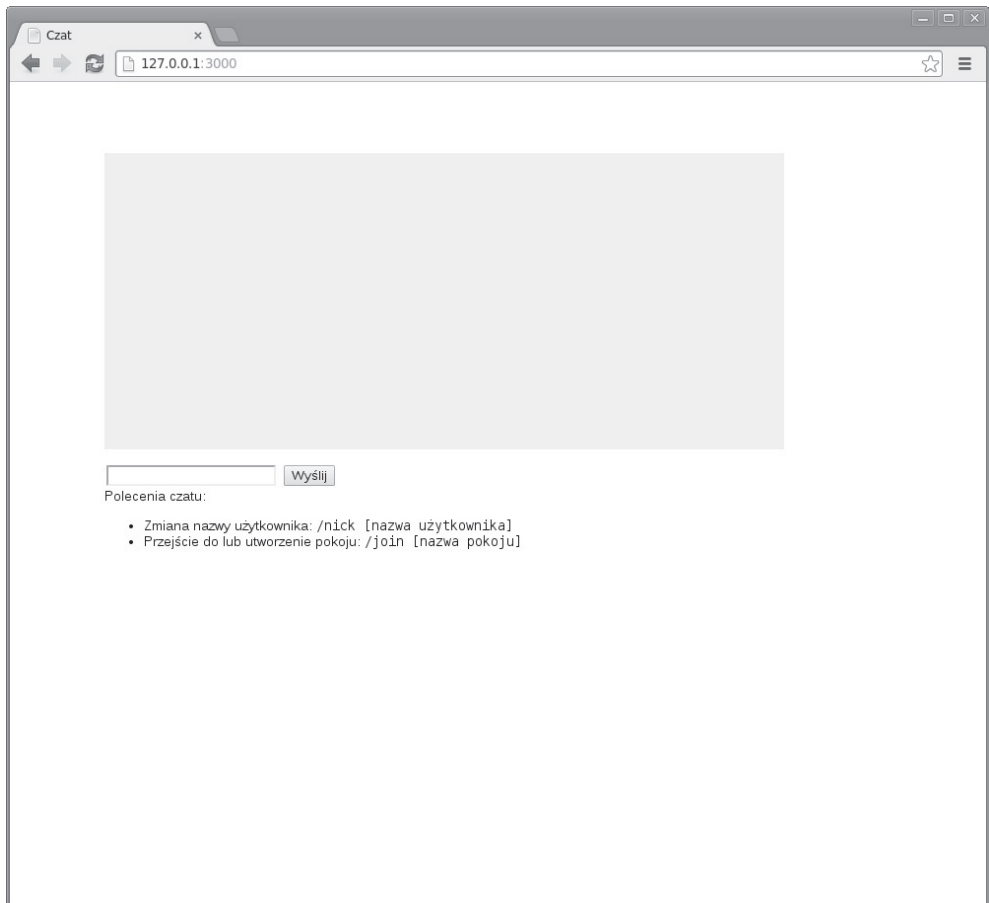
← Element <div> wyświetlający wiadomości czatu będzie mógł być przewijany, gdy wiadomości całkowicie go wypełnią.

Po dodaniu plików HTML i CSS możesz uruchomić aplikację w przeglądarce internetowej. Na obecnym etapie prac powinna wyglądać jak na rysunku 2.9.

Aplikacja oczywiście nie oferuje jeszcze pełnej funkcjonalności, ale pliki statyczne są udostępniane, a podstawowy układ graficzny prawidłowo generowany. Przechodzimy więc teraz do przygotowania kodu działającego po stronie serwera i odpowiedzialnego za obsługę wiadomości.

2.4. Obsługa wiadomości czatu za pomocą biblioteki *Socket.IO*

Z trzech wymagań stawianych budowanym aplikacjom omówiliśmy dotąd pierwsze, czyli udostępnianie plików statycznych. Przechodzimy teraz do drugiego — obsługi komunikacji między przeglądarką internetową i serwerem. Nowoczesne przeglądarki internetowe mają możliwość użycia technologii WebSocket do obsługi komunikacji między przeglądarką i serwerem. (Dokładne informacje dotyczące obsługi WebSocket w przeglądarkach internetowych znajdziesz na stronie <http://socket.io/#browser-support>).



Rysunek 2.9. Aplikacja czatu na obecnym etapie prac

WebSocket zapewnia warstwę abstrakcji dla siebie oraz dla innych mechanizmów transportu, zarówno dla Node, jak i kodu JavaScript działającego po stronie klienta. Biblioteka Socket.IO automatycznie zapewnia rozwiązania awaryjne, jeśli obsługa WebSocket nie została zaimplementowana w przeglądarce internetowej. We wszystkich przypadkach używane jest to samo API. W tym podrozdziale:

- pokrótce poznasz bibliotekę Socket.IO oraz zdefiniujesz funkcje Socket.IO niezbędne po stronie serwera;
- dodasz kod odpowiedzialny za konfigurację serwera Socket.IO;
- dodasz kod odpowiedzialny za obsługę różnych zdarzeń czatu.

Biblioteka Socket.IO standardowo oferuje wirtualne *kanały*, więc zamiast rozgłaszać każdą wiadomość do wszystkich połączonych użytkowników, można ją przekazać jedynie do tych, którzy są subskrybentami danego kanału. Dzięki tej funkcji implementacja pokoi czatu w budowanej tutaj aplikacji staje się naprawdę łatwym zadaniem, o czym się wkrótce przekonasz.

Biblioteka Socket.IO to również doskonały przykład użyteczności emiterów zdarzeń. Wspomniany *emiter zdarzeń* to w zasadzie użyteczny wzorec organizacji logiki asynchronicznej. W tym rozdziale poznasz kod pewnych emiterów zdarzeń, ale tym tematem dokładniej zajmiemy się w następnym rozdziale.

Emiter zdarzeń

Pod względem koncepcji emiter zdarzeń jest powiązany z pewnego rodzaju zasobem, może wysyłać i otrzymywać wiadomości do oraz z zasobu. Wspomnianym zasobem może być połączenie ze zdalnym serwerem lub coś znacznie bardziej abstrakcyjnego, na przykład postać w grze. Projekt Johnny-Five (<https://github.com/rwaldron/johnny-five>) wykorzystuje Node w aplikacjach robotów i używa emiterów zdarzeń do kontrolowania mikrokontrolerów Arduino.

W pierwszej kolejności trzeba uruchomić serwer i przygotować logikę odpowiedzialną za nawiązywanie połączenia. Następnie przystąpimy do zdefiniowania funkcji wymaganych po stronie serwera.

2.4.1. Konfiguracja serwera Socket.IO

Na początku w pliku *server.js* należy umieścić dwa podane poniżej wiersze kodu. Pierwszy powoduje wczytanie funkcji ze wskazanego modułu Node dostarczającego logikę potrzebną do obsługi po stronie serwera funkcji czatu związanych z biblioteką Socket.IO. Wskazany moduł zdefiniujemy za chwilę. Natomiast drugi wiersz uruchamia serwer i dostarcza funkcje Socket.IO przy założeniu, że mamy już zdefiniowany serwer HTTP, co pozwala na współdzielenie tego samego portu TCP/IP:

```
var chatServer = require('./lib/chat_server');
chatServer.listen(server);
```

Teraz trzeba utworzyć plik *chat_server.js* w podkatalogu *lib*. Na początku w wymienionym pliku umieść poniższe deklaracje zmiennych. Te deklaracje pozwolą na użycie biblioteki Socket.IO, a także inicjalizują kilka zmiennych przeznaczonych do definiowania stanu czatu:

```
var socketio = require('socket.io');
var io;
var guestNumber = 1;
var nickNames = {};
var namesUsed = [];
var currentRoom = {};
```

UTWORZENIE LOGIKI POŁĄCZENIA

Kolejnym krokiem jest dodanie przedstawionej w listingu 2.7 logiki odpowiedzialnej za zdefiniowanie funkcji *listen* serwera czatu. Wymieniona funkcja jest wywoływana w pliku *server.js*. Powoduje uruchomienie serwera Socket.IO, ogranicza ilość danych wyświetlanych w konsoli przez bibliotekę Socket.IO, a także definiuje sposób obsługi wszystkich połączeń przychodzących.

Listing 2.7. Logika odpowiedzialna za uruchomienie serwera Socket.IO

```

exports.listen = function(server) {
  io = socketio.listen(server);
  io.set('log level', 1);

  io.sockets.on('connection', function (socket) {
    guestNumber = assignGuestName(socket, guestNumber,
    ↪ nickNames, namesUsed);
    joinRoom(socket, 'Lobby');
    handleMessageBroadcasting(socket, nickNames);
    handleNameChangeAttempts(socket, nickNames, namesUsed);
    handleRoomJoining(socket);

    socket.on('rooms', function() {
      socket.emit('rooms', io.sockets.manager.rooms);
    });

    handleClientDisconnection(socket, nickNames, namesUsed);
  });
};

```

← **Uruchomienie serwera Socket.IO i umożliwienie mu współpracy z istniejącym serwerem HTTP.**

← **Zdefiniowanie sposobu obsługi połączenia użytkownika.**

← **Przypisanie użytkownikowi nazwy gościa podczas nawiązywania połączenia.**

← **Umieszczenie użytkownika w pokoju Lobby, gdy próbuje on nawiązać połączenie.**

← **Obsługa wiadomości użytkownika, prób zmiany nazwy użytkownika, a także tworzenia lub zmiany pokoju czatu.**

← **Wyświetlenie użytkownika wraz z listą pokoi, w których prowadzi czat.**

← **Zdefiniowanie logiki wykonywanej podczas rozłączania użytkownika.**

Jak możesz zauważyć, logika obsługi połączenia wywołuje wiele funkcji pomocniczych, które teraz trzeba będzie zdefiniować w pliku *chat_server.js*.

Po przygotowaniu logiki odpowiedzialnej za nawiązywanie połączenia trzeba zdefiniować kilka funkcji pomocniczych, które obsługują inne funkcjonalności oferowane przez aplikację.

2.4.2. Obsługa zdarzeń oraz scenariuszy w aplikacji

Aplikacja czatu musi zapewnić obsługę wymienionych poniżej zdarzeń i rodzajów scenariuszy:

- przypisanie nazwy gościa,
- żądanie zmiany pokoju,
- żądanie zmiany nazwy użytkownika,
- wysyłanie wiadomości czatu,
- tworzenie pokoju,
- rozłączanie użytkownika.

Do obsługi wymienionych funkcji konieczne jest przygotowanie kilku dodatkowych funkcji pomocniczych.

PRZYPISANIE NAZWY GOŚCIA

Pierwsza funkcja pomocnicza, którą trzeba dodać, nosi nazwę `assignGuestName()` i jest odpowiedzialna za obsługę nadawania nazwy nowemu użytkownikowi. Kiedy użytkownik po raz pierwszy nawiązuje połączenie z serwerem czatu, zostaje umieszczony w pokoju Lobby. Jednocześnie następuje wywołanie funkcji `assignGuestName()` i przypisanie mu nazwy odróżniającej nowego użytkownika od pozostałych.

Nazwa każdego gościa to w zasadzie słowo *Gość*, po którym znajduje się liczba o wartości zwiększanej po nawiązaniu połączenia przez każdego kolejnego użytkownika. Nazwa gościa jest przechowywana w zmiennej `nickNames` powiązanej z wewnętrznym identyfikatorem gniazda. Ponadto nazwa zostaje dodana do `namesUsed`, czyli zmiennej zawierającej użyte dotąd nazwy użytkowników. Implementacja funkcji `assignGuestName()` została przedstawiona w listingu 2.8, dodaj ją do pliku `lib/chat_server.js`.

Listing 2.8. Przypisanie nazwy gościa

```
function assignGuestName(socket, guestNumber, nickNames, namesUsed) {
  var name = 'Gość' + guestNumber;           ← Wygenerowanie nowej nazwy gościa.
  nickNames[socket.id] = name;               ← Powiązanie nazwy gościa z identyfikatorem połączenia klienta.
  socket.emit('nameResult', {                ← Podanie użytkownikowi wygenerowanej dla niego nazwy.
    success: true,
    name: name
  });
  namesUsed.push(name);                       ← Zwróć uwagę na użycie nazwy gościa.
  return guestNumber + 1;                     ← Inkrementacja licznika używanego podczas generowania nazw gości.
}
```

DOŁĄCZANIE DO POKOJU

Druga funkcja pomocnicza, którą trzeba dodać do pliku `chat_server.js`, nosi nazwę `joinRoom()`. Kod wymienionej funkcji przedstawiono w listingu 2.9. Jest ona odpowiedzialna za obsługę logiki związanej z dołączaniem użytkownika do pokoju czatu.

Listing 2.9. Logika obsługująca dołączanie do pokoju

```
function joinRoom(socket, room) {
  socket.join(room);                          ← Dołączenie uczestnika do pokoju.
  currentRoom[socket.id] = room;
  socket.emit('joinResult', {room: room});
  socket.broadcast.to(room).emit('message', {
    text: nickNames[socket.id] + ' dołączył do pokoju ' + room + '.'
  });
  // Ustalenie, czy jeszcze inni uczestnicy znajdują się w danym pokoju.
  var usersInRoom = io.sockets.clients(room);
  if (usersInRoom.length > 1) {               ← Jeżeli w pokoju są inni uczestnicy, aplikacja wyświetla ich liczbę.
    var usersInRoomSummary = 'Lista użytkowników w pokoju ' + room + ': ';
    for (var index in usersInRoom) {
      var userSocketId = usersInRoom[index].id;
      if (userSocketId != socket.id) {
        if (index > 0) {
          usersInRoomSummary += ', ';
        }
        usersInRoomSummary += nickNames[userSocketId];
      }
    }
  }
}
```

← Zauważ, że użytkownik znajduje się w pokoju.

← Poinformowanie uczestnika, że znajduje się we wskazanym pokoju.

← Poinformowanie pozostałych uczestników w pokoju o dołączeniu nowego.

```

    }
    usersInRoomSummary += '.';
    socket.emit('message', {text: usersInRoomSummary});
  }
}

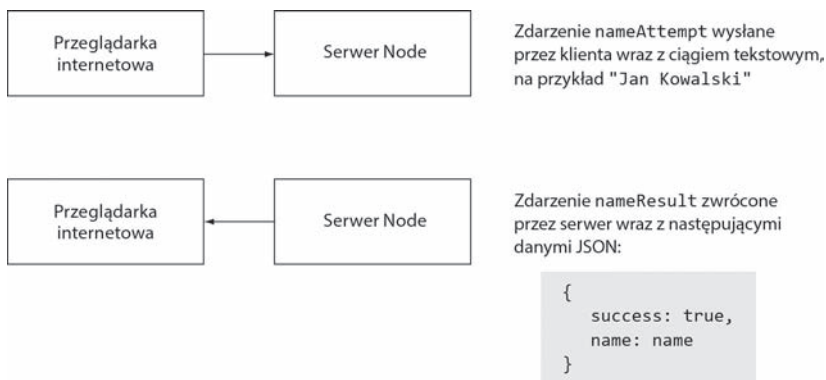
```

Przekazanie nowemu uczestnikowi podsumowania o innych uczestnikach znajdujących się w pokoju.

Dzięki bibliotece Socket.IO dołączenie uczestnika do pokoju czatu jest prostą operacją i wymaga jedynie wywołania metody `join` obiektu `socket`. Następnie aplikacja podaje informacje szczegółowe uczestnikowi oraz pozostałym uczestnikom znajdującym się w tym samym pokoju. Aplikacja podaje liczbę osób znajdujących się w pokoju czatu, a także informuje pozostałych uczestników w pokoju o dołączeniu nowego.

OBSŁUGA ŻĄDAŃ ZMIANY NAZWY UŻYTKOWNIKA

Jeżeli każdy uczestnik zachowa przydzieloną mu nazwę gościa, wtedy trudno będzie połączyć się, kto jest kim. Dlatego też aplikacja czatu pozwala użytkownikowi na zmianę jego nazwy. Jak pokazano na rysunku 2.10, zmiana nazwy powoduje wykonanie przez przeglądarkę internetową użytkownika żądania za pomocą Socket.IO, a następnie otrzymanie odpowiedzi wskazującej na sukces lub niepowodzenie operacji.



Rysunek 2.10. Żądanie zmiany nazwy użytkownika i odpowiedź negatywna

W pliku `lib/chat_server.js` umieść kod przedstawiony w listingu 2.10 zawierający definicję funkcji odpowiedzialnej za obsługę żądań zmiany nazwy użytkownika. Z perspektywy aplikacji użytkownik nie może zmienić nazwy na rozpoczynającą się od słowa *Gość* lub będącej już w użyciu.

Listing 2.10. Logika odpowiedzialna za obsługę zmiany nazwy użytkownika

```

function handleNameChangeAttempts(socket, nickNames, namesUsed) {
  socket.on('nameAttempt', function(name) {
    if (name.indexOf('Gość') == 0) {
      socket.emit('nameResult', {
        success: false,
        message: 'Nazwa użytkownika nie może rozpoczynać się od słowa "Gość".'
      });
    } else {
      if (namesUsed.indexOf(name) == -1) {
        var previousName = nickNames[socket.id];

```

Dodanie funkcji nasłuchującej zdarzeń `nameAttempt`.

Niedozwolone jest użycie nazwy rozpoczynającej się od słowa *Gość*.

Jeżeli nazwa nie jest jeszcze zarejestrowana, wtedy należy ją zarejestrować.

```

var previousNameIndex = namesUsed.indexOf(previousName);
namesUsed.push(name);
nickNames[socket.id] = name;
delete namesUsed[previousNameIndex];
socket.emit('nameResult', {
  success: true,
  name: name
});
socket.broadcast.to(currentRoom[socket.id]).emit('message', {
  text: previousName + ' zmienił nazwę na ' + name + '.'
});
} else {
  socket.emit('nameResult', {
    success: false,
    message: 'Ta nazwa jest używana przez innego użytkownika.'
  });
}
}
}
});
}
}

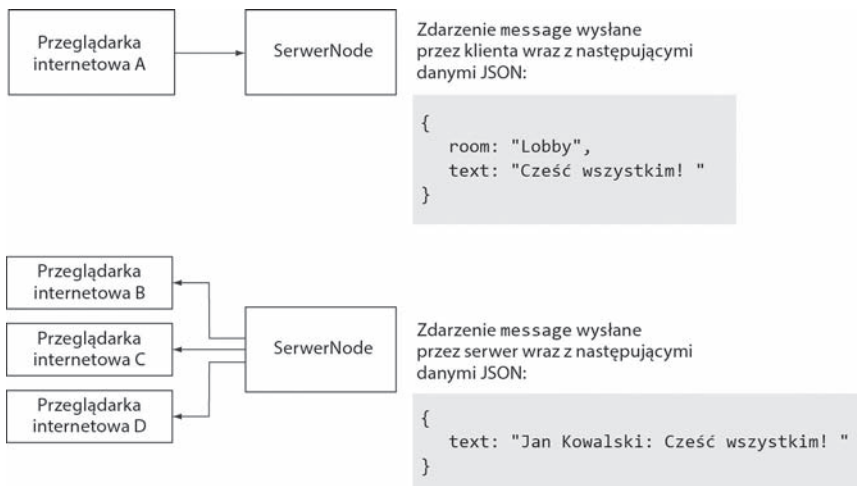
```

← **Usunięcie poprzedniej nazwy użytkownika i tym samym udostępnienie jej innym klientom.**

← **Wygenerowanie błędu, jeśli wybrana nazwa jest już używana przez innego użytkownika.**

WYSYŁANIE WIADOMOŚCI CZATU

Kiedy zadbaliliśmy już o nazwy użytkowników, przechodzimy do dodania kolejnej funkcji. Odpowiada ona za obsługę sposobu wysyłania wiadomości czatu. Na rysunku 2.11 pokazano podstawę działania tego procesu: użytkownik emituje zdarzenie wskazujące pokój, do którego ma zostać wysłana wiadomość, oraz jej tekst. Następnie serwer przekazuje wiadomość do wszystkich uczestników czatu znajdujących się w danym pokoju.



Rysunek 2.11. Wysyłanie wiadomości czatu

Poniższy kod umieść w pliku `lib/chat_server.js`. Do przekazywania wiadomości jest używana funkcja `broadcast()` biblioteki `Socket.IO`:

```

function handleMessageBroadcasting(socket) {
  socket.on('message', function (message) {
    socket.broadcast.to(message.room).emit('message', {

```

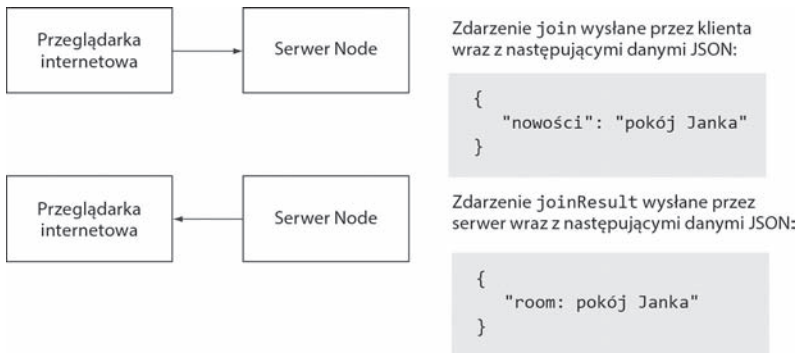
```

    text: nickNames[socket.id] + ': ' + message.text
  });
});
}

```

TWORZENIE POKOJU

Kolejnym krokiem jest dodanie funkcji pozwalającej użytkownikowi na dołączenie do istniejącego pokoju czatu lub utworzenie nowego. Na rysunku 2.12 pokazano interakcje zachodzące wówczas między użytkownikiem i serwerem.



Rysunek 2.12. Przejście do innego pokoju czatu

Poniższy kod umieść w pliku *lib/chat_server.js*, umożliwiając tym samym zmianę pokoju czatu. Zwróć uwagę na użycie metody `leave()` biblioteki Socket.IO:

```

function handleRoomJoining(socket) {
  socket.on('join', function(room) {
    socket.leave(currentRoom[socket.id]);
    joinRoom(socket, room.newRoom);
  });
}

```

OBSŁUGA ROZŁĄCZENIA UŻYTKOWNIKA

Do pliku *lib/chat_server.js* konieczne jest dodanie poniższego fragmentu kodu odpowiedzialnego za usunięcie nazwy użytkownika ze zmiennych `nickNames` i `namesUsed`, gdy użytkownik kończy pracę z aplikacją czatu:

```

function handleClientDisconnection(socket) {
  socket.on('disconnect', function() {
    var nameIndex = namesUsed.indexOf(nickNames[socket.id]);
    delete namesUsed[nameIndex];
    delete nickNames[socket.id];
  });
}

```

W ten sposób zakończyliśmy tworzenie komponentów działających po stronie serwera. Teraz możemy powrócić do kontynuowania prac nad logiką działającą po stronie klienta.

2.5. Użycie kodu JavaScript działającego po stronie klienta do utworzenia interfejsu użytkownika aplikacji

Po dodaniu działającej po stronie serwera logiki Socket.IO do obsługi wiadomości czatu pora dodać kod JavaScript działający po stronie klienta i potrzebny do prowadzenia komunikacji z serwerem. Wspomniany kod musi zapewnić następujące możliwości:

- wysyłanie do serwera wiadomości oraz żądań zmiany pokoju lub nazwy użytkownika;
- wyświetlanie wiadomości pochodzących od innych uczestników czatu oraz listy dostępnych pokoi.

Rozpocznijmy do implementacji pierwszej z wymienionych powyżej funkcji.

2.5.1. Przekazywanie do serwera wiadomości oraz żądań zmiany pokoju lub nazwy użytkownika

Pierwszym fragmentem kodu JavaScript działającego po stronie klienta jest prototyp obiektu JavaScript, który będzie przetwarzał polecenia czatu, wysyłał wiadomości oraz żądania zmiany pokoju lub nazwy użytkownika.

W katalogu *public/javascripts* utwórz plik o nazwie *chat.js* i umieść w nim poniższy fragment kodu. W języku JavaScript służy on do zdefiniowania „klasy” pobierającej podczas tworzenia pojedynczy argument w postaci gniazda Socket.IO:

```
var Chat = function(socket) {
  this.socket = socket;
};
```

Następnie dodaj poniższą funkcję odpowiedzialną za wysyłanie wiadomości:

```
Chat.prototype.sendMessage = function(room, text) {
  var message = {
    room: room,
    text: text
  };
  this.socket.emit('message', message);
};
```

A teraz dodaj funkcję przeznaczoną do obsługi zmiany pokoju:

```
Chat.prototype.changeRoom = function(room) {
  this.socket.emit('join', {
    newRoom: room
  });
};
```

Na końcu dodaj funkcję przedstawioną w listingu 2.11 i przeznaczoną do przetwarzania poleceń czatu. Rozpoznawane są dwa polecenia: *join* pozwalające na dołączenie się do pokoju lub utworzenie nowego oraz *nick* pozwalające na zmianę nazwy użytkownika.

Listing 2.11. Funkcja przetwarzająca polecenia czatu

```
Chat.prototype.processCommand = function(command) {
  var words = command.split(' ');
  var command = words[0]
```

```

        .substring(1, words[0].length)
        .toLowerCase();
var message = false;

switch(command) {
    case 'join':
        words.shift();
        var room = words.join(' ');
        this.changeRoom(room);
        break;
    case 'nick':
        words.shift();
        var name = words.join(' ');
        this.socket.emit('nameAttempt', name);
        break;
    default:
        message = 'Nieznane polecenie.';
        break;
};

return message;
};

```

← Przetworzenie polecenia z listy słów.

← Obsługa operacji zmiany pokoju lub utworzenia nowego.

← Obsługa operacji zmiany nazwy użytkownika.

← Jeżeli polecenie nie zostanie rozpoznane, wtedy nastąpi wygenerowanie błędu.

2.5.2. Wyświetlenie w interfejsie użytkownika wiadomości i listy dostępnych pokoi

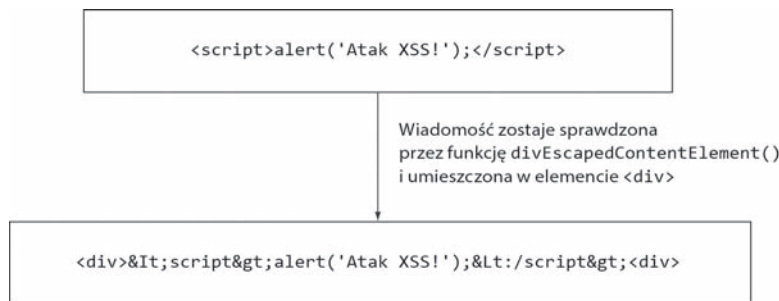
W tym punkcie zajmiemy się dodaniem logiki odpowiedzialnej za bezpośrednią współpracę z opartym na przeglądarce interfejsem użytkownika za pomocą biblioteki jQuery. Pierwsza funkcja, nad którą będziemy pracować, służy do wyświetlania danych tekstowych.

Z perspektywy zapewnienia bezpieczeństwa w aplikacji sieciowej rozróżniamy dwa rodzaje danych tekstowych. Pierwszy to *zaufane* dane tekstowe, na które składają się dane pochodzące z aplikacji. Drugi to *niezaufane* dane tekstowe, które pochodzą od użytkownika lub powstały na podstawie danych podanych przez użytkownika. Dane tekstowe pochodzące od użytkownika są uznawane za niezaufane, ponieważ złośliwy użytkownik może celowo umieścić w nich znaczniki `<script>` zawierające logikę w języku JavaScript. Jeżeli tak przygotowane dane będą w niezmodyfikowanej postaci wyświetlone innym użytkownikom, mogą doprowadzić do wystąpienia niechcianych zdarzeń, na przykład przekierować użytkownika na inną stronę internetową. Taka metoda przechwylenia aplikacji sieciowej nosi nazwę ataku XSS (*Cross-Site Scripting*).

Budowana przez nas aplikacja czatu używa dwóch funkcji pomocniczych do wyświetlania danych tekstowych. Pierwsza funkcja jest przeznaczona do wyświetlania niezaufanych danych tekstowych, natomiast druga do wyświetlania zaufanych danych tekstowych.

Funkcja `divEscapedContentElement()` wyświetla niezaufane dane tekstowe. Przeprowadza operację sprawdzenia tekstu i zmienia znaki specjalne na odpowiadające im encje HTML, jak pokazano na rysunku 2.13. W ten sposób przeglądarka internetowa „wie”, jak wyświetlić dany znak, i nie próbuje zinterpretować go jako części znacznika HTML.

Z kolei funkcja `divSystemContentElement()` jest przeznaczona do wyświetlania zaufanej treści wygenerowanej przez system, a nie przez innych użytkowników.



Rysunek 2.13.
Unieszkodliwienie
niebezpiecznych
znaków

W katalogu `public/javascripts` utwórz plik o nazwie `chat_ui.js` i umieść w nim poniższy fragment kodu zawierający obie wymienione wcześniej funkcje pomocnicze:

```

function divEscapedContentElement(message) {
    return $('<div></div>').text(message);
}

function divSystemContentElement(message) {
    return $('<div></div>').html('<i>' + message + '</i>');
}

```

Kolejna funkcja umieszczana w pliku `chat_ui.js` służy do przetwarzania danych wejściowych użytkownika, informacje o niej przedstawiono w listingu 2.12. Jeżeli dane wejściowe użytkownika rozpoczynają się od ukośnika (`/`), będą potraktowane jako polecenie czatu. W przeciwnym razie zostaną wysłane do serwera jako wiadomość czatu, a następnie przekazane innym użytkownikom i dodane do tekstu czatu pokoju, w którym aktualnie znajduje się użytkownik.

Listing 2.12. Funkcja przetwarzająca niezmodyfikowane dane wejściowe użytkownika

```

function processUserInput(chatApp, socket) {
    var message = $('#send-message').val();
    var systemMessage;

    if (message.charAt(0) == '/') {
        systemMessage = chatApp.processCommand(message);
        if (systemMessage) {
            $('#messages').append(divSystemContentElement(systemMessage));
        }
    } else {
        chatApp.sendMessage($('#room').text(), message);
        $('#messages').append(divEscapedContentElement(message));
        $('#messages').scrollTop($('#messages').prop('scrollHeight'));
    }

    $('#send-message').val('');
}

```

Jeżeli dane wejściowe uczestnika czatu rozpoczynają się od ukośnika, należy potraktować je jako polecenie czatu.

Dane wejściowe inne niż polecenia czatu należy przekazać innym uczestnikom czatu.

Po przygotowaniu funkcji pomocniczych można przystąpić do dodania logiki przedstawionej w listingu 2.13. Ma ona zastosowanie, gdy strona internetowa zostanie w pełni wczytana w przeglądarce internetowej użytkownika. Kod listingu 2.13 jest odpowiedzialny za inicjalizację obsługi zdarzeń `Socket.IO`.

Listing 2.13. Działająca po stronie klienta logika inicjalizacji aplikacji

```

var socket = io.connect();

$(document).ready(function() {
    var chatApp = new Chat(socket);

    socket.on('nameResult', function(result) {
        var message;

        if (result.success) {
            message = 'Twoja nazwa użytkownika to ' + result.name + '.';
        } else {
            message = result.message;
        }
        $('#messages').append(divSystemContentElement(message));
    });

    socket.on('joinResult', function(result) {
        $('#room').text(result.room);
        $('#messages').append(divSystemContentElement('Pokój został zmieniony.'));
    });

    socket.on('message', function (message) {
        var newElement = $('<div></div>').text(message.text);
        $('#messages').append(newElement);
    });

    socket.on('rooms', function(rooms) {
        $('#room-list').empty();

        for(var room in rooms) {
            room = room.substring(1, room.length);
            if (room != '') {
                $('#room-list').append(divEscapedContentElement(room));
            }
        }

        $('#room-list div').click(function() {
            chatApp.processCommand('/join ' + $(this).text());
            $('#send-message').focus();
        });
    });

    setInterval(function() {
        socket.emit('rooms');
    }, 1000);

    $('#send-message').focus();

    $('#send-form').submit(function() {
        processUserInput(chatApp, socket);
        return false;
    });
});

```

← Wyświetlenie wyniku operacji zmiany nazwy użytkownika.

← Wyświetlenie wyniku operacji zmiany pokoju.

← Wyświetlenie otrzymanych wiadomości.

← Wyświetlenie listy dostępnych pokoi.

← Kliknięcie nazwy pokoju powoduje przejście do niego.

← Żądanie pobrania od czasu do czasu listy dostępnych pokoi.

← Wysłanie formularza powoduje wysłanie wiadomości czatu.

W celu zakończenia prac nad aplikacją należy w pliku *public/stylesheets/style.css* umieścić jeszcze style CSS przedstawione w listingu 2.14.

Listing 2.14. Ostatnie style, które trzeba dodać do pliku *style.css*

```
#room-list {  
  float: right;  
  width: 100px;  
  height: 300px;  
  overflow: auto;  
}  
  
#room-list div {  
  border-bottom: 1px solid #eee;  
}  
  
#room-list div:hover {  
  background-color: #ddd;  
}  
  
#send-message {  
  width: 700px;  
  margin-bottom: 1em;  
  margin-right: 1em;  
}  
  
#help {  
  font: 10px "Lucida Grande", Helvetica, Arial, sans-serif;  
}
```

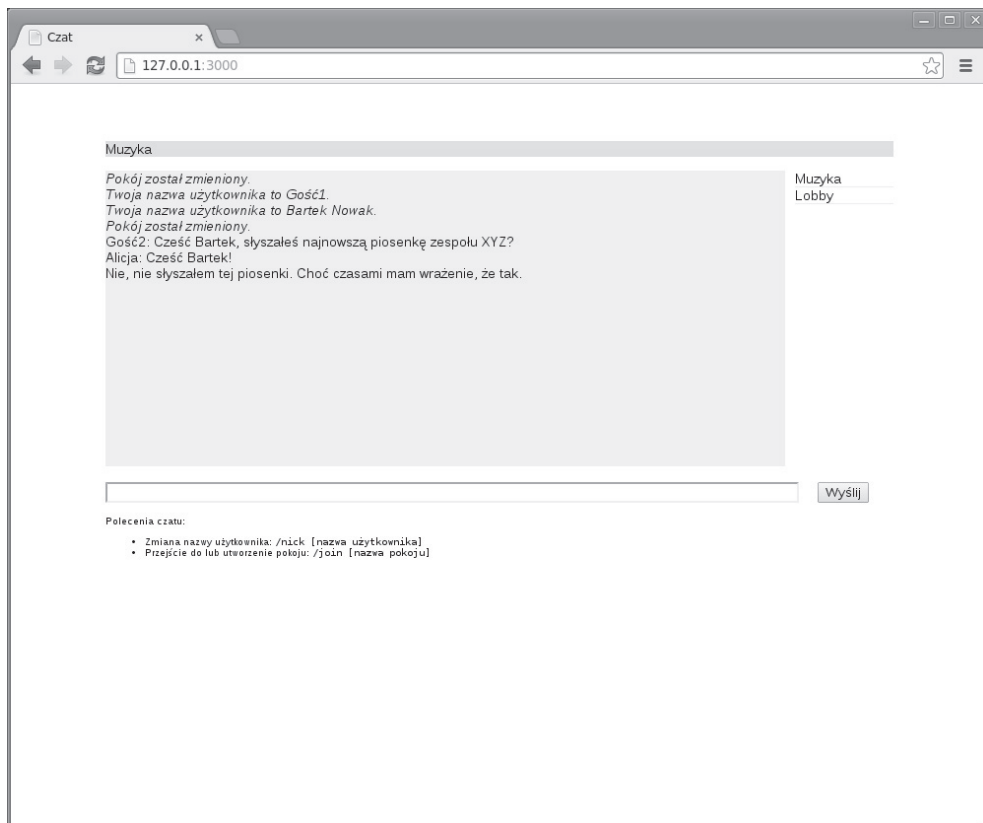
Po dodaniu ostatnich fragmentów kodu spróbuj uruchomić aplikację (przez wydanie polecenia `node server.js`). Powinieneś otrzymać wynik pokazany na rysunku 2.14.

2.6. Podsumowanie

W ten sposób za pomocą Node.js utworzyłeś małą aplikację sieciową działającą w czasie rzeczywistym!

Na tym etapie powinieneś już wiedzieć, jak konstruowane są aplikacje Node i jak wygląda ich kod źródłowy. Jeżeli jakiegokolwiek aspekty omówionej tutaj aplikacji nadal pozostają dla Ciebie niezrozumiałe, nie przejmuj się tym. W kolejnych rozdziałach jeszcze dokładniej zajmiemy się omówieniem technik i technologii zastosowanych w zbudowanej tutaj aplikacji czatu.

Jednak zanim przejdziemy dalej do programowania z użyciem Node, powinieneś dowiedzieć się, jak rozwiązywać problemy charakterystyczne dla programowania asynchronicznego. W kolejnym rozdziale przedstawiono więc podstawowe techniki i sztuczki, dzięki którym zaoszczędzisz sobie dużej ilości czasu i frustracji.



Rysunek 2.14. Ukończona aplikacja czatu

Skorowidz

A

- Ajax
 - udostępnianie plików, 38
- Amazon Web Services, 344, 345
- ANSI, 388, 389
- API
 - Node, 29–32, 351–353
 - REST, 270–277
- aplikacja
 - asynchroniczna, 30
 - Connect, 154
 - Node, 35–60
 - sieciowa w Node, 97–124
- architektura
 - model-widok-kontroler, 310
 - MVC, 310
- asercja
 - equal(), 288
 - notEqual(), 289
 - ok(), 289
- asynchroniczność
 - operacje wejścia-wyjścia, 24
 - serwer oparty na zdarzeniach, 25–27
- atak
 - typu tęczowa tablica, 245
 - XSS, 315
- atrybuty znacznika w Jade, 328

B

- Basic Authentication, 161
- baza danych
 - nierelacyjna, 141–151
 - relacyjna, 130–141
- Bcrypt, 243
- BDD, 286
- biblioteka
 - should.js, 301–303
 - Socket.IO, 356–363

błąd

- typu 404, 278, 279
- typu 500, 280–283
- Browseling, 27, 28
- BSON, 148
- buforowanie
 - danych, 117
 - szablonów EJS, 321
 - widoku w aplikacji Express, 225
- bufory, 363–365

C

- certyfi­kat w protokole HTTPS, 123
- CGI, 382
- ciąg zaburzający, 245
- closure, 83
- Connect, 153–176
- consolidate.js, 430, 431
- cookies
 - JSON, 181
 - metody pośredniczące, 179–182
 - sesji, 198
- CRUD, 102
- CSRF, 202
- cURL, 103

D

- dane tekstowe, 55
- Data-Intensive Real-Time, Patrz DIRT
- debugowanie Node, 421
- DIRT, 27–32
- dodanie modułu do repozytorium npm, 405–409
- domknięcie, 164
 - JavaScript, 83
- dyspozytor we frameworku Connect, 154
- dziedziczenie szablonu, 333–335

E

EC2, 344, 345
 EJS, 216, 223, 224, 314–322
 Elastic Compute Cloud, Patrz EC2
 emiter zdarzeń, 48
 użycie do do obsługi powtarzających się
 zdarzeń, 74–82
 Express, 154, 213–240
 zaawansowane użycie, 241–284
 Express-Expose, 431
 Express-Resource, 431

F

filtry EJS, 316–319
 Fleet, 346
 Forever, 347, 348
 formatowanie odpowiedzi JSON, 433
 formularze w aplikacji Express, 232–236
 framework
 Connect, 153–176
 Express, 154, 213–240
 konfiguracja zaawansowana, 432
 rozszerzenia, 429–432
 zaawansowane użycie, 241–284
 Mocha, 293–298
 nodeunit, 291, 292
 Soda, 305–307
 testowy, 286
 Tobi, 303–305
 Vows, 298–300
 funkcja
 add(), 116
 app.locals(), 231, 232
 assignGuestName(), 50
 cp.exec(), 379, 380
 cp.fork(), 383, 384
 cp.form(), 379
 cp.spawn(), 379, 381
 divEscapedContentElement(), 55
 divSystemContentElement(), 55
 fs.watch(), 359, 377
 fs.watchFile(), 359, 377
 getRange(), 260
 http.createServer(), 99
 inherits(), 80
 joinRoom(), 50
 loadOrInitializeTaskArray(), 128
 net.connect(), 369

next(), 156, 158, 162
 notFound(), 116
 parse(), 107
 parseInt(), 107
 pipe(), 368
 readFile(), 89
 req.param(), 268
 require(), 65, 66
 res.error(), 251
 res.locals(), 231
 res.messages(), 251
 res.setHeader(), 179
 storeTasks(), 129
 url.parse(), 107
 use(), 157

funkcje synchroniczne w Node.js, 375

G

Git, 398, 399
 GitHub, 398–405

H

Hogan.js, 322, 323
 hosting
 aplikacji Node, 342–346
 plików statycznych, 353, 354

I

inspektor Node, 426, 427
 instalacja
 frameworka Express, 216, 217
 Node
 w systemie Linux, 414
 w systemie OS X, 411, 412
 w systemie Windows, 413, 414
 integracja EJS w aplikacji, 320, 321
 interfejs REPL, 106
 IRC, 396

J

Jade, 326–337
 JavaScript
 informacje ogólne, 22, 23
 w szablonach Jade, 331
 JSHint, 421
 JSON, 23

K

kanaly Redis, 145
 katalog node_modules, 68
 klasa Socket, 366
 klucz prywatny w protokole HTTPS, 123
 kod Node
 organizacja, 62–64
 kody błędów modułu fs, 377
 kolejność
 bajtów, 365
 metod pośredniczących, 158–160
 kolekcje MongoDB, 146, 147
 kompilacja Node, 415, 416
 konstruktor Buffer, 363
 kontrola przepływu, 84–92
 równoległa, 89–91
 szeregowa, 85–89

L

logika w szablonach Jade, 330–333

M

magazyn danych, 126–130
 oparty na plikach, 127–130
 sesji, 198, 199
 w pamięci, 126
 mapa hash bazy Redis, 143
 menedżer npm, 416–419
 metoda
 app.configure(), 221
 app.render(), 223
 Buffer.byteLength(), 106
 listeners(), 79
 next(), 154, 155
 pipe(), 111
 query(), 136
 removeAllListeners(), 79
 res.download(), 239, 240
 res.end(), 100
 res.render(), 223, 228, 231, 232
 res.sendFile(), 238
 res.setHeader(), 101
 res.write(), 100
 server.listen(), 101
 setMaxListeners(), 79
 Stream#pipe(), 110
 String#slice(), 107
 test.expect(), 292
 watch(), 81

metody HTTP, 103
 metody pośredniczące, 153–158, 257–59
 admin(), 162
 basicAuth(), 178, 200–202
 bodyParser(), 178, 182–184
 charakterystyczne dla trasy, 263–266
 compress(), 178, 207–209
 cookieParser(), 178–182
 csrf(), 178, 202, 203
 director(), 178
 directory(), 209, 210
 do obsługi stronicowania, 267
 errorHandler(), 175
 errorHandler(), 178, 203–205
 errorPage(), 176
 favicon(), 178, 191
 hello(), 157, 173
 kolejność, 158–160
 konfigurowalne, 164–170
 limit(), 178, 184–186
 logger(), 164, 166
 logger(), 178, 188–191
 methodOverride(), 178, 191–194
 obsługa błędów, 170–176
 pets(), 174
 query(), 178, 186, 187
 rewrite(), 169
 router(), 166–168
 session(), 178, 195–199
 static(), 178, 205–207
 users(), 174
 uwierzytelniania, 161
 vhost(), 178, 194, 195
 we frameworku Connect, 155, 177–211
 middleware, 153
 Mocha, 293–298
 moduł
 ansi.js, 390
 assert, 287–290
 child_process, 379
 debug, 423
 filed, 378
 filesystem, 375–379
 formidable, 118–121
 sprawdzanie postępu operacji
 przekazywania plików, 122
 fs, 375–379
 fstream, 378
 hiredis-node, 146
 http, 99
 node-mongodb-native, 146

- moduł
 - node-postgres, 139
 - querystring, 117
- moduły Node, 63
 - tworzenie, 64–68
- modyfikacja odpowiedzi JSON, 433
- MongoDB, 146–149
- Mongoose, 149–151
- monkey patching, 68
- montowanie we frameworku Connect, 160–163
- Mustache, 322–326
- MVC, 310
- MySQL, 131–139

N

- nagłówki
 - Content-Type, 114
 - Content-Length, 106
- nagłówki HTTP, 101
- nasłuchiwanie zdarzeń, 70, 74–82
- nazwy zdarzeń, 77
- negocjacja treści, 275, 276
- ngen, 407
- Nginx, 353
- nieblokujące operacje wejścia-wyjścia, 24, 25, 31
- niezaufane dane tekstowe, 55
- Nimble, 85, 91
- Node
 - hosting aplikacji, 342–346
 - wdrażanie aplikacji, 341–354
- Node Package Manager, 416–419
- Node.js
 - informacje ogólne, 21–33
- Nodejitsu Forever, 347, 348
- nodeunit, 291, 292
- NoSQL, 141–151
- notacja duże O, 144

O

- obiekt
 - Buffer, 104, 364
 - ChildProcess, 381
 - exports, 63, 67
 - module.exports, 67
 - process, 371–375
- obiekty Stream, 387
- obsługa błędów
 - a emiter zdarzeń, 80
 - serwera plików statycznych, 112–114

- w aplikacji Express, 277–283
 - we frameworku Connect, 170–176
- obsługa HTTP i WebSocket w jednej aplikacji, 39
- obsługa przekazywania plików, 118–121
- opcje konfiguracyjne frameworka Express, 432
- open source, 393
- OpenSSL, 123
- organizacja szablonów Jade, 333–337

P

- partials, 325
- pętla zdarzeń, 24, 82
- plik package.json, 40, 406, 407
- pliki statyczne
 - udostępnianie, 108–114
- pobieranie zasobów w aplikacji Express, 237–240
- podkatalogi projektu Node, 406
- polecenie
 - case w Jade, 333
 - express, 215, 218, 243
 - include w Jade, 336
 - mixin, 327
 - w Jade, 336, 337
 - mocha, 294
 - npm, 384
 - sudo, 347, 414
- PostgreSQL, 139–141
- potoki w Node, 111
- PRG, 251
- procesy potomne, 379–384
- programowanie
 - asynchroniczne, 69–83
 - w Node, 61–93
- protokół HTTPS, 122, 123
- Proxy, 353, 354
- przechowywanie danych aplikacji Node, 125–152
- przeglądarka internetowa, 23, 24
- przekazywanie danych w aplikacji Express, 234–237
- przetwarzanie
 - argumentów podanych w powłoce, 385, 386
 - danych przekazanych za pomocą formularza sieciowego, 114–122
- pull request, 403

R

- Rackspace Cloud, 345
- RDBMS, 130–141
- ReadableStream, 111

Redis, 141–146
 REPL, 106
 replikacja synchroniczna, 139
 repozytorium
 menedżera npm, 64
 npm, 395
 dodanie modułu, 405–409
 REST, 102
 routing, 166
 we frameworku Express, 259–270
 rozwidlenie projektu, 403, 404

S

Sauce Labs, 307
 sekcja
 lambda, 324
 Mustache, 323–325
 sekwencja logiki asynchronicznej, 84–92
 Selenium, 306
 serwer
 chmury, 343, 344
 dedykowany, 343
 plików statycznych, 42–45, 109–112
 Selenium, 306
 TCP w Node, 365–369
 VPS, 343
 wirtualny, 343
 sesje, 195–199
 shoutbox, 241, 242
 sieć typu TCP/IP
 a Node, 363–371
 silnik szablonów
 tworzenie silnika szablonów we framewrok
 Express, 429, 430
 silnik szablonów Embedded JavaScript, *Patrz* EJS
 silniki szablonów HTML, 314–337
 Socket.IO, 38, 47, 356–363
 konfiguracja serwera, 48, 49
 Soda, 305–307
 stos wywołań, 423
 stronicowanie we frameworku Express, 266–270
 strumieniowanie danych, 32
 Swig, 430
 sygnały w systemie UNIX, 374, 375
 system zarządzania relacyjną bazą danych, 130–141
 szablon
 EJS, 314–322
 Jade, 326–337
 Mustache, 322–326
 w aplikacji sieciowej, 309–337

T

tabela hash bazy Redis, 143
 TCP/IP
 a Node, 363–371
 TDD, 286
 techniki programowania asynchronicznego, 69–83
 Testling, 28
 testowanie aplikacji Node, 285–308
 testy
 akceptacyjne, 286, 303–307
 funkcyjne, *Patrz* testy akceptacyjne
 jednostkowe, 286–303
 zautomatyzowane, 285
 Tobi, 303–305
 token uwierzytelnienia, 202
 trasy rejestracji, 249
 tryb safe, 148
 tworzenie
 aplikacji bloga, 311–314
 aplikacji Express, 215–240
 aplikacji sieciowej w Node, 97–124
 aplikacji typu shoutbox we frameworku
 Express, 241–284
 klienta TCP, 369–371
 minimalnej aplikacji Socket.IO, 357–359
 modułu Node, 64–68
 narzędzi powłoki, 384–391
 podstawowego serwera plików statycznych,
 42–45
 procesów zewnętrznych, 379–384
 serwera HTTP, 44, 99–102
 serwera TCP, 365–369
 szablonów EJS, 315, 316
 usługi sieciowej RESTful, 102–108
 własnych filtrów EJS, 319
 typ danych Buffer, 363–365
 typ MIME, 38, 42

U

udostępnianie
 plików statycznych, 108–114
 plików za pomocą protokołu http w Node, 38,
 39
 Upstart, 349–351
 usługa sieciowa RESTful, 102–108
 uwierzytelnianie
 podstawowe, 161, 271
 użytkowników w aplikacji Express, 242–259

V

V8, 23
 Virtual Host, 194, 195
 VirtualBox, 342
 Vows, 298
 VPS, 343

W

wdrażanie aplikacji Node
 w środowisku produkcyjnym, 341–354
 z repozytorium Git, 346, 347
 WebSocket, 38, 39, 47, 356
 widoki aplikacji Express, 223–232
 właściwość
 req.method, 103
 res.statusCode, 102
 WritableStream, 111
 współbieżność, 127
 wyciek zmiennej globalnej, 293
 wywołanie zwrotne, 70
 do obsługi zdarzeń jednorazowych, 71–74
 wzorzec REST, 102

X

Xcode, 411

Z

zabezpieczanie aplikacji
 dzięki użyciu protokołu HTTPS, 122, 123
 zadania
 równoległe, 84
 szeregowo, 84

zależność aplikacji, 40, 41
 zaufane dane tekstowe, 55
 zbiór Redis, 144
 zdarzenia specjalne emitowane przez obiekt
 proces, 373
 zdarzenie progress, 122
 zmienna
 __dirname, 224
 magiczna, 109
 NODE_ENV, 171, 221
 NODE_PATH, 68
 root, 109
 zmienne środowiskowe, 372
 ustawienie, 221
 znacznik <div>, 328
 znaczniki
 Jade, 328, 329
 Mustache, 323, 324, 325

Ż

żądanie
 DELETE, 103
 usunięcie zasobu, 107, 108
 GET, 103
 pobieranie zasobów, 105, 106
 HTTP obsługiwane przez serwer HTTP
 w Node, 99, 100
 POST, 103
 tworzenie zasobów, 103–105
 przekazania zmian, 403
 PUT, 103
 REST, 270

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

JavaScript to jeden z niewielu języków programowania w historii, który budził i wciąż budzi tak wiele emocji. Na rynku pojawił się w 1995 roku i od tego czasu: a) był obecny tylko w przeglądarkach; b) był masowo blokowany i c) wieszczono jego koniec... aż do dzisiaj, kiedy trudno sobie wyobrazić stronę WWW, która nie korzystałaby z możliwości tego języka. Współcześnie JavaScript zaczyna także odnosić sukces w aplikacjach działających po stronie serwera. Wyobraź sobie, że Twoje żądania po stronie serwera obsługuje JavaScript i wprowadź tę wizję w życie.

Node.js to platforma, która zapewnia najwyższą wydajność dzięki wykorzystywaniu nieblokujących operacji I/O oraz asynchronicznego mechanizmu zdarzeń. Działa na bazie najlepszego silnika obsługującego język JavaScript – V8 (autorstwa firmy Google) – i pozwala programistom osiągać niezwykle efekty. Zastanawiasz się, jak wykorzystać ten potencjał? Sięgnij po tę książkę i rozpocznij przygodę z Node.js! W trakcie lektury poznasz podstawy programowania na tej platformie, zbudujesz asynchroniczną logikę, wykorzystasz protokoły sieciowe oraz podłączysz się do popularnych baz danych. W kolejnych rozdziałach będziesz mieć niepowtarzalną okazję, by poznać popularne i przydatne biblioteki oraz stworzyć RESTowe API. Na sam koniec dowiesz się, jak wdrożyć aplikację Node.js w środowisku produkcyjnym, oraz zaznajomisz się z ekosystemem tej platformy. Brzmi zachęcająco?

Dzięki tej książce:

- poznasz platformę Node.js,
- opanujesz techniki programowania asynchronicznego,
- zbudujesz RESTowe API z wykorzystaniem Node.js,
- wdrożysz swoją aplikację,
- przekonasz się, jak wydajny może być JavaScript.

Twój przewodnik po Node.js!

helion.pl
księgarnia
internetowa

Nr katalogowy: 23914



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:

➤ <http://helion.pl/promocje>

Książki najchętniej czytane:

➤ <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

➤ <http://helion.pl/novosci>

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

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-9678-9



9 788324 696789

Cena: 69,00 zł