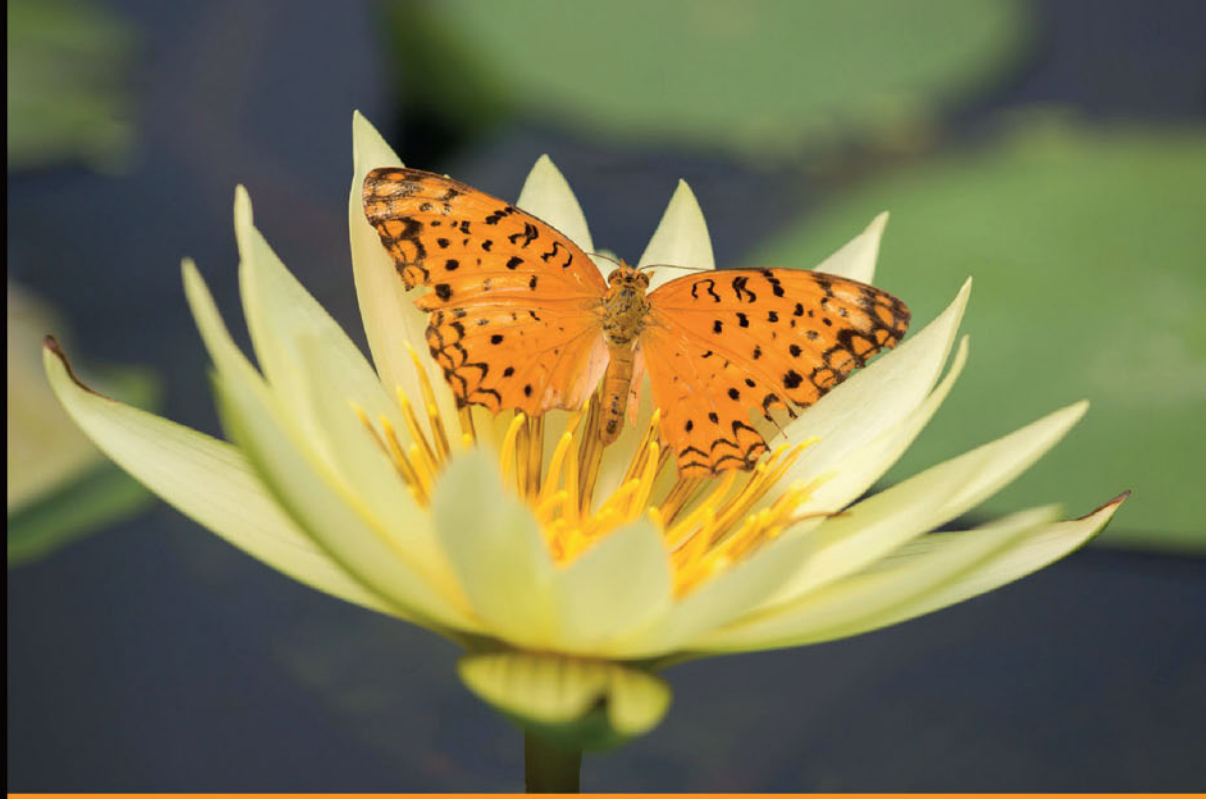


Technologia i rozwiązania

Platforma Node.js

Przewodnik webdevelopera

Wydanie III

David Herron

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Node.js Web Development - Third Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-3611-7

First published in the English language under the title
'Node.js Web Development - Third Edition – (9781785881503)'.

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/pnjsp3.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pnjsp3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

4

ROZDZIAŁ

Serwery i klienty HTTP — pierwsze kroki tworzenia aplikacji internetowej

Gdy już zapoznałeś się z modułami platformy Node.js, pora wykorzystać tę wiedzę do zbudowania prostej aplikacji internetowej na tę platformę. W tym rozdziale aplikacja nie będzie skomplikowana, co pozwoli przyjrzeć się trzem różnym platformom do tworzenia aplikacji używanym w Node.js. W dalszych rozdziałach aplikacje będą bardziej złożone, jednak zanim nauczysz się chodzić, musisz zacząć raczkować.

Oto zagadnienia omawiane w tym rozdziale:

- Obiekty typu `EventEmitter`
- Oczekiwanie na zdarzenia HTTP i obiekty serwera HTTP
- Routing żądań HTTP
- Szablonowe łańcuchy znaków ze standardu ES2015
- Tworzenie prostej aplikacji internetowej bez użycia platform
- Platforma do tworzenia aplikacji Express
- Funkcje warstwy pośredniej w platformie Express
- Zarządzanie kodem wymagającym obliczeniowo
- Obiekty klienta HTTP
- Używanie platformy Express do tworzenia prostych usług opartych na architekturze REST

Przesyłanie i odbieranie zdarzeń za pomocą obiektów typu EventEmitter

Używanie obiektów typu `EventEmitter` to jeden z podstawowych idiomów w platformie Node.js. Wiele podstawowych modułów należy do rodziny `EventEmitter`. Obiekty typu `EventEmitter` są też doskonałym szkieletem do programowania asynchronicznego. Takie obiekty nie mają nic wspólnego z tworzeniem aplikacji internetowych i są na tyle niewidoczne, że mógłbyś nie zauważyć ich istnienia. Po raz pierwszy zetkniesz się z nimi w kontekście klas związanych z protokołem HTTP.

W tym rozdziale używane są obiekty serwera i klienta HTTP. Są to podklasy typu `EventEmitter`, wykorzystujące go do przesyłania zdarzeń dotyczących wszystkich etapów z protokołu HTTP. Zrozumienie typu `EventEmitter` pomoże Ci zrozumieć nie tylko wspomniane podklasy, ale też obiekty wielu innych klas z platformy Node.js.

Klasa `EventEmitter` jest zdefiniowana w module `events` Node.js. Aby bezpośrednio użyć tej klasy, należy wywołać instrukcję `require('events')`. W większości sytuacji ten moduł nie jest potrzebny, zdarza się jednak, że w programie trzeba utworzyć podklasę klasy `EventEmitter`.

Utwórz teraz plik *pulser.js* zawierający podany poniżej kod. Ilustruje on przesyłanie i otrzymywanie zdarzeń bezpośrednio za pomocą klasy `EventEmitter`:

```
var events = require('events');
var util = require('util');
// Definiowanie klasy Pulser

function Pulser() {
    events.EventEmitter.call(this);
}
util.inherits(Pulser, events.EventEmitter);

Pulser.prototype.start = function() {
    setInterval(() => {
        util.log('>>>> pulse');
        this.emit('pulse');
        util.log('<<<< pulse');
    }, 1000);
};
```

W tym kodzie zdefiniowana jest klasa `Pulser` pochodna od klasy `EventEmitter` (odpowiada za to instrukcja `util.inherits`). Zadaniem tej klasy jest przesyłanie zdarzeń do wszystkich odbiorników w określonym czasie (raz na sekundę). W metodzie `start` za pomocą funkcji `setInterval` uruchamiane są powtarzające się co sekundę wywołania zwrotne. Wywołanie `emit` przesyła zdarzenia `pulse` do wszystkich odbiorników.

Mógłby to być punkt wyjścia do utworzenia modułu ogólnego użytku przesyłającego zdarzenia co określony czas. Tu jednak znajdziesz tylko omówienie działania klasy `EventEmitter`.

Zobacz teraz, jak zastosować obiekt typu `Pulser`. W tym celu należy dodać do pliku *pulser.js* następujący kod:

```
// Tworzenie obiektu typu Pulser
var pulser = new Pulser();
// Funkcja obsługi zdarzeń
pulser.on('pulse', () => {
    util.log('Odebrano zdarzenie pulse');
});

// Rozpoczęcie generowania pulsu
pulser.start();
```

Ten kod tworzy obiekt typu `Pulser` i przetwarza zgłaszane przez niego zdarzenia `pulse`. Wywołanie `pulser.on('pulse')` tworzy połączenia potrzebne do tego, by zdarzenia `pulse` uruchamiały wywoływaną zwrotnie funkcję. W ostatnim kroku wywoływana jest metoda `start`, uruchamiająca cały proces.

Zapisz przedstawiony kod w pliku i nazwij go *pulser.js*. Po uruchomieniu pliku powinieneś zobaczyć następujące dane wyjściowe:

```
$ node pulser.js
19 Apr 16:58:04 - >>>> pulse
19 Apr 16:58:04 - odebrano zdarzenie pulse
19 Apr 16:58:04 - <<<< pulse
19 Apr 16:58:05 - >>>> pulse
19 Apr 16:58:05 - odebrano zdarzenie pulse
19 Apr 16:58:05 - <<<< pulse
19 Apr 16:58:06 - >>>> pulse
19 Apr 16:58:06 - odebrano zdarzenie pulse
19 Apr 16:58:06 - <<<< pulse
```

Teoria działania klasy EventEmitter

Zdarzenia z rodziny `EventEmitter` przyjmują wiele nazw. Możesz zastosować taką nazwę, jaką uznasz za sensowną. Możesz też zdefiniować dowolnie wiele nazw zdarzeń. Takie nazwy są definiowane w prosty sposób — za pomocą wywołania `.emit` z nazwą zdarzenia. Nie trzeba stosować żadnych formalnych zabiegów. Nie istnieje też rejestr nazw zdarzeń. Samo wywołanie `.emit` wystarcza, by zdefiniować nazwę zdarzenia.

> Zwyczajowo nazwa zdarzenia `error` oznacza błąd.

Obiekt przesyła zdarzenia przy użyciu funkcji `.emit`. Zdarzenia są przesyłane do wszystkich odbiorników, które zarejestrowały się, aby otrzymywać zdarzenia od danego obiektu. Aby zarejestrować w programie chęć otrzymywania zdarzenia, należy wywołać metodę `.on` danego obiektu i podać nazwę zdarzenia oraz funkcję jego obsługi.

Często razem ze zdarzeniem trzeba przesyłać dane. W tym celu wystarczy dodać dane jako argumenty do wywołania `.emit`:

```
this.emit('eventName', data1, data2, ..);
```

Gdy program otrzymuje zdarzenie, wspomniane dane są argumentami wywoływanej zwrotnie funkcji. Program może odbierać takie zdarzenie w następujący sposób:

```
emitter.on('eventName', (data1, data2, ..) => {
  // Wykonywanie operacji na podstawie zdarzenia
});
```

Odbiorca i nadawca zdarzenia nie wymieniają między sobą komunikatów. Oznacza to, że nadawca przechodzi do dalszych zadań i nie otrzymuje powiadomień o otrzymanych zdarzeniach, podjętych działaniach lub wykrytych błędach.

Aplikacje w postaci serwera HTTP

Obiekt serwera HTTP jest podstawą wszystkich aplikacji internetowych w platformie Node.js. Sam obiekt działa bardzo podobnie do protokołu HTTP, a korzystanie z tego obiektu wymaga znajomości owego protokołu. W większości sytuacji można posłużyć się platformą do tworzenia aplikacji (taką jak Express), która ukrywa szczegóły związane z protokołem HTTP. Programista może się dzięki temu skoncentrować na logice biznesowej.

W rozdziale 2., „Konfigurowanie platformy Node.js", widziałeś już prostą aplikację z serwerem HTTP:

```
var http = require('http');
http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Witaj, świecie!\n');
}).listen(8124, '127.0.0.1');
console.log('Serwer działa pod adresem http://127.0.0.1:8124');
```

Funkcja `http.createServer` tworzy obiekt `http.Server`. Ponieważ jest to obiekt typu `EventEmitter`, kod można zapisać w inny sposób, aby podkreślić typ tego obiektu:

```
var http = require('http');
var server = http.createServer();
server.on('request', (req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Witaj, świecie!\n');
});
server.listen(8124, '127.0.0.1');
console.log('Serwer działa pod adresem http://127.0.0.1:8124');
```

Zdarzenie `request` przyjmuje funkcję, która przyjmuje obiekty `request` i `response`. Obiekt `request` zawiera dane z przeglądarki, natomiast obiekt `response` służy do zapisywania danych odsyłanych w odpowiedzi. Funkcja `listen` sprawia, że serwer zaczyna oczekiwać na zdarzenia i przekierowuje zdarzenia związane z żądaniami nadchodzącymi z przeglądarki.

Przyjrzyj się teraz ciekawszemu rozwiązaniu, wykonującemu różne operacje w zależności od wprowadzonego adresu URL.

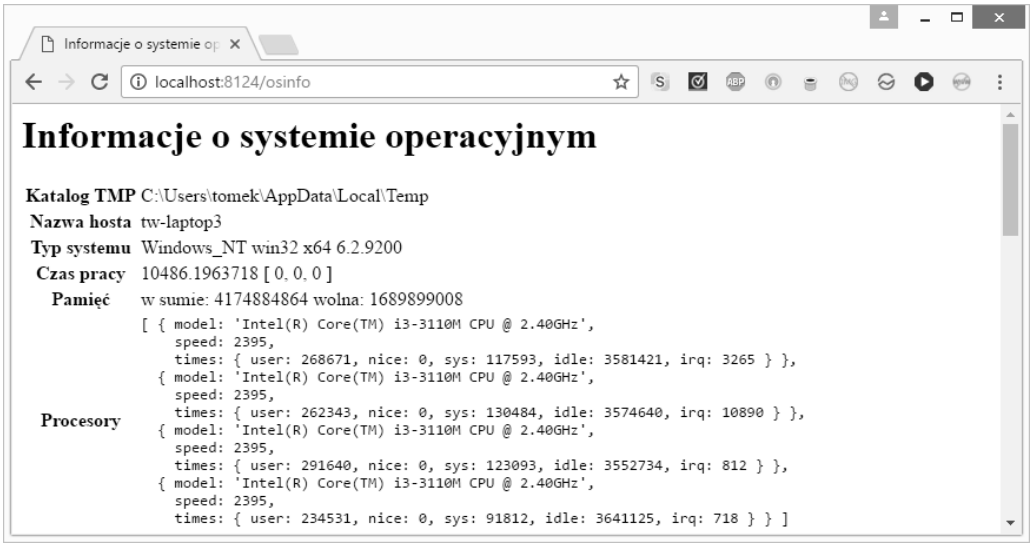
Utwórz nowy plik, *server.js*, i umieść w nim następujący kod:

```
var http = require('http');
var util = require('util');
var url = require('url');
var os = require('os');

var server = http.createServer();
server.on('request', (req, res) => {
    var requrl = url.parse(req.url, true);
    if (requrl.pathname === '/') {
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end(
`<html><head><title>Witaj, świecie!</title></head>
<body><h1>Witaj, świecie!</h1>
<p><a href='/osinfo'>Informacje o systemie</a></p>
</body></html>`);
    } else if (requrl.pathname === "/osinfo") {
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end(
`<html><head><title>Informacje o systemie operacyjnym</title></head>
<body><h1>Informacje o systemie operacyjnym</h1>
<table>
<tr><th>Katalog TMP</th><td>${os.tmpDir()}</td></tr>
<tr><th>Nazwa hosta</th><td>${os.hostname()}</td></tr>
<tr><th>Typ systemu</th><td>${os.type()} ${os.platform()} ${os.arch()}
${os.release()}</td></tr>
<tr><th>Czas pracy</th><td>${os.uptime()} ${util.inspect(os.loadavg())}</
td></tr>
<tr><th>Pamięć</th><td>w sumie: ${os.totalmem()} wolna: ${os.freemem()}</
td></tr>
<tr><th>Procesory</th><td><pre>${util.inspect(os.cpus())}</pre></td></tr>
<tr><th>Sieć</th><td><pre>${util.inspect(os.networkInterfaces())}</
pre></td></tr>
</table>
</body></html>`);
    } else {
        res.writeHead(404, {'Content-Type': 'text/plain'});
        res.end("Nieprawidłowy adres URL "+ req.url);
    }
});

server.listen(8124);
console.log('Oczekiwanie pod adresem http://localhost:8124');
```

Aby uruchomić ten serwer, wpisz następujące polecenie:

```
$ node server.js
Oczekiwanie pod adresem http://localhost:8124
```

Ta aplikacja ma działać podobnie jak funkcja `sysinfo` z języka PHP. Moduł `os` platformy Node.js jest używany w celu pobrania informacji o serwerze, widocznych na rysunku 4.1. Ten kod można łatwo rozbudować o wyświetlanie innych danych na temat serwera.



Rysunek 4.1. Informacje o serwerze wyświetlane przez aplikację

Głównym aspektem każdej aplikacji internetowej jest metoda przekierowująca żądania do metod ich obsługi. Do obiektu `request` dołączone są różne informacje. Dwie z nich, z pól `request.url` i `request.method`, są przydatne do przekierowywania (routingu) żądań.

W pliku *server.js* kod sprawdza pole `request.url`, aby ustalić, którą stronę należy wyświetlić. Adres jest najpierw parsowany (za pomocą polecenia `url.parse`), co ułatwia jego przetwarzanie. Tu można dzięki temu przeprowadzić proste porównanie pola `pathname` z różnymi wartościami w celu określenia, co należy zrobić.

W niektórych aplikacjach internetowych ważny jest czasownik z protokołu HTTP (`GET`, `DELETE`, `POST` itd.). Można go sprawdzić za pomocą pola `request.method` obiektu `request`. Na przykład metoda `POST` jest często używana do przekazywania formularzy (`FORM`).

Człon `pathname` z adresu URL żądania służy do kierowania żądań do metod ich obsługi. Choć ta metoda routingu żądań, oparta na prostym porównywaniu łańcuchów znaków, dobrze sprawdza się w niewielkich aplikacjach, szybko staje się trudna w użyciu. W większych aplikacjach stosowane jest dopasowywanie do wzorca. Jeden fragment adresu URL żądania służy

wtedy do wyboru funkcji obsługi, a inne fragmenty — do pobierania danych z adresu. Działanie tego mechanizmu zobaczysz w kontekście omawiania platformy Express w punkcie „Wprowadzenie do platformy Express".

Jeżeli adres URL żądania nie zostanie rozpoznany, serwer prześle z powrotem stronę błędu z kodem odpowiedzi `404`. Kod odpowiedzi informuje przeglądarkę o stanie żądania. Wartość `200` oznacza, że wszystko przebiega prawidłowo, a kod `404` — że żądana strona nie istnieje. Dostępnych jest też wiele innych kodów odpowiedzi HTTP. Każdy z nich ma określone znaczenie.

Wielowierszowe i szablonowe łańcuchy znaków ze standardu ES2015

W poprzednim przykładzie zobaczyłeś dwie nowe funkcje wprowadzone w standardzie ES2015 — wielowierszowe i szablonowe łańcuchy znaków. Te mechanizmy mają upraszczać tworzenie łańcuchów znaków.

W istniejących łańcuchach znaków używane są apostrofy lub cudzysłowy. Szablonowe łańcuchy znaków są wyróżniane **akcentem lewostronnym**:

```
`szablonowy łańcuch znaków`
```

Przed wprowadzeniem standardu ES2015 jednym ze sposobów na utworzenie wielowierszowego łańcucha znaków było zastosowanie następującej konstrukcji:

```
["<html><head><title>Witaj, świecie!</title></head>",
 "<body><h1>Witaj, świecie!</h1>",
 "<p><a href='/osinfo'>Informacje o systemie</a></p>",
 "</body></html>"]
.join('\n')
```

Tak, ten kod był używany w tym przykładzie w starszych wydaniach książki. Standard ES2015 umożliwia zastosowanie następującego rozwiązania:

```
`<html><head><title>Witaj, świecie!</title></head>
<body><h1>Witaj, świecie!</h1>
<p><a href='/osinfo'>Informacje o systemie</a></p>
</body></html>`
```

Jest to bardziej zwięzły i prostszy zapis. Otwierający akcent lewostronny znajduje się w pierwszym wierszu, zamykający w ostatnim, a wszystko między nimi jest częścią łańcucha znaków.

Głównym przeznaczeniem **szablonowych łańcuchów znaków** jest tworzenie łańcuchów, w których można łatwo bezpośrednio wstawiać wartości. Większość innych języków programowania to umożliwia. Teraz technika ta jest dostępna także w JavaScripcie.

Przed wprowadzeniem standardu ES2015 programista stosował następujące rozwiązanie:

```
[ …
  "<tr><th>Typ systemu operacyjnego</th><td>{ostype} {osplat} {osarch} {osrelease}</
td></tr>"
  … ].join('\n')
.replace("{ostype}", os.type())
.replace("{osplat}", os.platform())
.replace("{osarch}", os.arch())
.replace("{osrelease}", os.release())
```

Także ten fragment pochodzi z przykładu z wcześniejszego wydania książki. Za pomocą szablonowych łańcuchów znaków ten kod można zapisać tak:

```
`...<tr><th>Typ systemu operacyjnego</th><td>${os.type()} ${os.platform()}
${os.arch()}
${os.release()}</td></tr>...`
```

W szablonowym łańcuchu znaków człon w bloku `${…}` jest interpretowany jako wyrażenie. Może być to proste wyrażenie matematyczne, referencja do zmiennej lub — tak jak tu — wywołanie funkcji.

Ostatnią kwestią, o jakiej warto wspomnieć, są wcięcia. W normalnym kodzie długa lista argumentów jest wyrównywana względem zawierającego ją wywołania funkcji. Jednak w pokazanych tu wielowierszowych łańcuchach znaków tekst jest zawijany do kolumny zerowej. Z czego to wynika?

Może to negatywnie wpływać na czytelność kodu. Warto więc uwzględnić, czy ważniejsza jest czytelność czy inna kwestia — liczba znaków w generowanym kodzie w HTML-u. Spacje potrzebne do dodawania (ze względu na czytelność) wcięć w kodzie stają się częścią łańcucha znaków i znajdują się w generowanym kodzie w HTML-u. Dzięki zawijaniu kodu do kolumny zerowej unikasz dodatkowych spacji w danych wyjściowych kosztem pewnego spadku czytelności.

Sniffer HTTP — podsłuchiwanie wymiany komunikatów przez protokół HTTP

Zdarzenia generowane przez obiekt serwera HTTP można wykorzystać też do innych zadań niż tylko do udostępniania aplikacji internetowej. Pokazany dalej kod to przydatny moduł odbierający wszystkie zdarzenia serwera HTTP. Ten kod może posłużyć jako użyteczne narzędzie diagnostyczne; ilustruje też, jak działają obiekty serwera HTTP.

Obiekt serwera HTTP w Node.js to obiekt typu `EventEmitter`. Sniffer HTTP odbiera wszystkie zdarzenia serwera i wyświetla informacje powiązane z każdym z nich.

Utwórz plik *httpsniffer.js* i umieść w nim następujący kod:

```
var util = require('util');
var url = require('url');

exports.sniffOn = function(server) {
  server.on('request', (req, res) => {
    util.log('e_request');
    util.log(reqToString(req));
  });
  server.on('close', errno => { util.log('e_close errno='+ errno);
});
  server.on('checkContinue', (req, res) => {
    util.log('e_checkContinue');
    util.log(reqToString(req));
    res.writeContinue();
  });
  server.on('upgrade', (req, socket, head) => {
    util.log('e_upgrade');
    util.log(reqToString(req));
  });
  server.on('clientError', () => { util.log('e_clientError'); });
};

var reqToString = exports.reqToString = function(req) {
  var ret=`req ${req.method} ${req.httpVersion} ${req.url}` +'\n';
  ret += JSON.stringify(url.parse(req.url, true)) +'\n';
  var keys = Object.keys(req.headers);
  for (var i = 0, l = keys.length; i < l; i++) {
    var key = keys[i];
    ret += `${i} ${key}: ${req.headers[key]}` +'\n';
  }
  if (req.trailers)
  ret += req.trailers +'\n';
  return ret;
};
```

To sporo kodu! Najważniejsza jest w nim funkcja `sniffOn`. Gdy otrzyma funkcję serwera HTTP, używa funkcji `.on` do dołączenia odbiorników wyświetlających dane na temat każdego zdarzenia wygenerowanego przez obiekt tego serwera. Pozwala to uzyskać dość szczegółowy ślad komunikatów HTTP przesyłanych w aplikacji.

Aby zastosować ten sniffer, wstaw następujący kod tuż przed wywołaniem funkcji `listen` w pliku *server.js*:

```
require('./httpsniffer').sniffOn(server);
server.listen(8124);
console.log('Obieranie zdarzeń pod adresem http://localhost:8124');
```

Po dodaniu tego kodu uruchom serwer w pokazany wcześniej sposób. Otwórz w przeglądarce stronę *http://localhost:8124/*, a zobaczysz w konsoli następujące dane wyjściowe:

```
$ node server.js
Odbieranie zdarzeń pod adresem http://localhost:8124
30 Jan 16:32:39 - request
30 Jan 16:32:39 - request GET 1.1 /
{"protocol":null,"slashes":null,"auth":null,"host":null,"port":null,
"hostname":null,"hash":null,"search":"","query":{},"pathname":"/","path"
:"/","href":"/"}
0 host: localhost:8124
1 connection: keep-alive
2 accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,*/*;q=0.8
3 upgrade-insecure-requests: 1
4 user-agent: Mozilla/5.0 (X11; CrOS x86_64 7520.67.0) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/47.0.2526.110 Safari/537.36
5 accept-encoding: gzip, deflate, sdch
6 accept-language: en-US,en;q=0.8
[object Object]

30 Jan 16:32:49 - request
30 Jan 16:32:49 - request GET 1.1 /osinfo
{"protocol":null,"slashes":null,"auth":null,"host":null,"port":null,
"hostname":null,"hash":null,"search":"","query":{},"pathname":"/osinfo",
"path":"/osinfo","href":"/osinfo"}
0 host: localhost:8124
1 connection: keep-alive
2 accept: text/html,application/xhtml+xml,application/xml;q=0.9,image
/webp,*/*;q=0.8
3 upgrade-insecure-requests: 1
4 user-agent: Mozilla/5.0 (X11; CrOS x86_64 7520.67.0) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/47.0.2526.110 Safari/537.36
5 referer: http://localhost:8124/
6 accept-encoding: gzip, deflate, sdch
7 accept-language: en-US,en;q=0.8
[object Object]
```

W ten sposób uzyskałeś narzędzie do „podsłuchiwania" zdarzeń serwera HTTP. Ta prosta technika wyświetla szczegółowy dziennik danych o zdarzeniach. Wzorzec ten można zastosować do dowolnego obiektu z rodziny `EventEmitter`. Możesz go stosować do badania działania takich obiektów w programie.

Platformy do tworzenia aplikacji internetowych

Obiekt serwera HTTP działa bardzo podobnie jak protokół HTTP. Choć daje to duże możliwości (podobnie jak jazda samochodem z ręczną skrzynią biegów zapewnia niskopoziomową kontrolę nad pojazdem), typowe aplikacje internetowe lepiej jest tworzyć na wyższym poziomie. Lepiej jest w abstrakcyjnej formie ująć szczegóły protokołu HTTP i skoncentrować się na samej aplikacji.

Społeczność programistów skupiona wokół platformy Node.js opracowała szereg modułów ukrywających różne aspekty protokołu HTTP. Moderowane listy modułów znajdziesz na stronach *http://nodeframework.com/* i *https://github.com/vndmtrx/awesome-nodejs*.

Jednym z powodów stosowania platformy jest to, że często obsługuje ona wszystkie opracowane przez ponad 20 lat standardowe najlepsze praktyki związane z tworzeniem aplikacji internetowych. Oto te praktyki:

- udostępnianie strony w reakcji na błędny adres URL (strony *404*);
- sprawdzanie adresów URL i formularzy pod kątem ataków przez wstrzyknięcie skryptów;
- obsługa plików cookie w celu zarządzania sesjami;
- zapisywanie w dzienniku żądań na potrzeby śledzenia użytkowników i debugowania kodu;
- uwierzytelnianie;
- obsługa plików statycznych, na przykład rysunków, stylów CSS oraz plików z kodem w JavaScripcie i HTML-u;
- udostępnianie nagłówków do kontroli pamięci podręcznej na poziomie jednostek pośredniczących;
- ograniczanie wielkości stron lub czasu wykonania.

Platformy do tworzenia aplikacji internetowych pomagają skupić się na konkretnych zadaniach bez pogrążania się w szczegółach implementacji protokołu HTTP. Ukrywanie szczegółów to tradycyjny sposób zwiększania wydajności programistów. Jest on skuteczny zwłaszcza wtedy, gdy programista używa biblioteki lub platformy z gotowymi funkcjami, które odpowiadają za szczegóły.

Wprowadzenie do platformy Express

Express to prawdopodobnie najpopularniejsza platforma do tworzenia aplikacji internetowych w Node.js. Jest na tyle popularna, że występuje w akronimie MEAN (MongoDB, ExpressJS, AngularJS i Node.js). Express jest opisywany jako *podobny do Sinatry* (to popularna platforma do tworzenia aplikacji w języku Ruby) i niezbyt ścisły. To oznacza, że nie trzeba ściśle przestrzegać struktury kodu. Możesz pisać kod w sposób, który uznasz za najlepszy.

Strona główna platformy Express to *http://expressjs.com/*.

Wkrótce napiszesz prostą aplikację do wyznaczania liczb Fibonacciego za pomocą Expressa. W dalszych rozdziałach będziesz często z niej korzystać. Zobaczysz też, jak złagodzić problem z wydajnością związane ze wspomnianym wcześniej wymagającym obliczeniowo kodem.

Zacznij od zainstalowania narzędzia **Express Generator**. Choć możesz po prostu zacząć pisać kod, Express Generator zapewnia pustą aplikację będącą punktem wyjścia, którą później zmodyfikujesz.

Zainstaluj narzędzie za pomocą następujących poleceń:

```
$ mkdir fibonacci
$ cd fibonacci
$ npm install express-generator@4.x
```

Wygląda to inaczej niż metoda instalacji sugerowana w witrynie Expressa (gdzie zalecane jest użycie opcji `-g` oznaczającej instalację globalną). Jawnie podany jest tu też numer wersji, aby zagwarantować zgodność kodu z platformą.

Wcześniej wspomniałem, że obecnie wiele osób odradza globalne instalowanie modułów. W modelu Twelve-Factor zdecydowanie zaleca się, by nie instalować zależności globalnych. Tu stosuję się do tego zalecenia.

Efekt to zainstalowanie polecenia `express` w katalogu *./node_modules/.bin*:

```
$ ls node_modules/.bin/
express
```

Możesz uruchamiać je w następujący sposób:

```
$ ./node_modules/.bin/express --help

  Usage: express [options] [dir]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -e, --ejs           add ejs engine support (defaults to jade)
        --hbs           add handlebars engine support
    -H, --hogan         add hogan.js engine support
```

```
    -c, --css <engine> add stylesheet <engine> support
(less|stylus|compass|sass) (defaults to plain css)
        --git          add .gitignore
    -f, --force        force on non-empty directory
```

Zapewne nie chcesz przy każdym wywołaniu aplikacji Express Generator wpisywać polecenia `./node_modules/.bin/express` (dotyczy to też innych aplikacji z narzędziami uruchamianymi w wierszu poleceń). Na szczęście łatwym rozwiązaniem jest modyfikacja zmiennej środowiskowej `PATH`:

```
$ export PATH=node_modules/.bin:${PATH}
```

Jest to polecenie z powłoki `bash`. Użytkownicy powłoki `csh` powinni zastosować inną instrukcję:

```
$ setenv PATH node_modules/.bin:${PATH}
```

Po skonfigurowaniu w ten sposób zmiennej środowiskowej `PATH` można bezpośrednio wywoływać polecenie `express`. Wykorzystaj je teraz do wygenerowania pustej początkowej wersji aplikacji.

Po zainstalowaniu narzędzia `express-generator` w katalogu *fibonacci* użyj tego programu do utworzenia pustej aplikacji (zobacz rysunek 4.2).

```
laptop:fibonacci david$ express --ejs --git
destination is not empty, continue? [y/N] y

   create : .
   create : ./package.json
   create : ./app.js
   create : ./.gitignore
   create : ./public
   create : ./public/javascripts
   create : ./public/images
   create : ./public/stylesheets
   create : ./public/stylesheets/style.css
   create : ./routes
   create : ./routes/index.js
   create : ./routes/users.js
   create : ./views
   create : ./views/index.ejs
   create : ./views/error.ejs
   create : ./bin
   create : ./bin/www

   install dependencies:
     $ cd . && npm install

   run the app:
     $ DEBUG=fibonacci:* npm start

laptop:fibonacci david$ npm uninstall express-generator
unbuild express-generator@4.13.1
laptop:fibonacci david$
```

Rysunek 4.2. Tworzenie aplikacji za pomocą platformy Express

W ten sposób tworzony jest zestaw plików (ich omówienie znajdziesz dalej). Katalog *node_modules* nadal zawiera moduł `express-generator`, który obecnie jest już niepotrzebny. Możesz zostawić go w katalogu i zignorować lub dodać do atrybutu `devDependencies` w wygenerowanym pliku *package.json*. Jeszcze inna możliwość to odinstalowanie go, co pokazane jest na rysunku 4.2.

Następny krok to uruchomienie pustej aplikacji w opisany na rysunku sposób. Widoczne tam polecenie, `npm start`, wykorzystuje następujący fragment z pliku *package.json*:

```
"scripts": {
    "start": "node ./bin/www"
},
```

Narzędzie `npm` udostępnia skrypty, które służą do automatyzowania różnych zadań. Podczas lektury książki posłużysz się tym mechanizmem do wykonywania rozmaitych operacji. Większość skryptów `npm` jest uruchamiana za pomocą polecenia `npm run scriptName`, jednak instrukcja `start` też jest rozpoznawana przez narzędzie npm i można ją uruchamiać w pokazany wcześniej sposób.

Pierwszy krok polega na zainstalowaniu zależności (`npm install`). Później aplikacja jest uruchamiana (rysunek 4.3).

```
laptop:fibonacci david$ npm start

> fibonacci@0.0.0 start /Users/david/fibonacci
> DEBUG=fibonacci:* node ./bin/www

  fibonacci:server Listening on port 3000 +0ms
```

Rysunek 4.3. Uruchamianie aplikacji

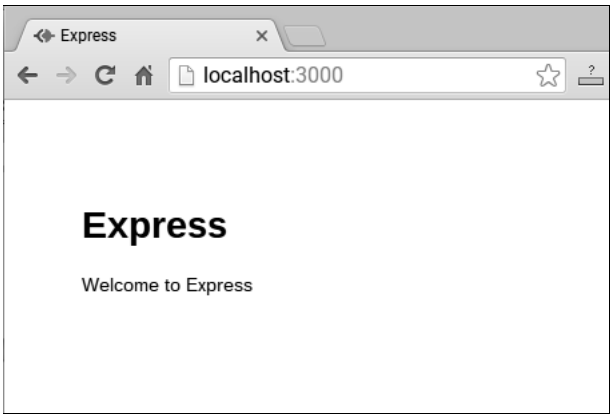
Następnie możesz zmodyfikować wygenerowany skrypt `npm start`, aby wyświetlał na ekranie informacje diagnostyczne. W sekcji `scripts` wprowadź następujące zmiany[1]:

```
"scripts": {
    "start": "DEBUG=fibonacci:* node ./bin/www"
},
```

Dodanie członu `DEBUG=fibonacci:*` ustawia zmienną środowiskową `DEBUG`. W aplikacjach platformy Express zmienna ta włącza kod diagnostyczny, który wyświetla ślad wykonywania kodu przydatny w trakcie debugowania.

Ponieważ z danych wyjściowych wynika, że aplikacja używa portu `3000`, otwórz w przeglądarce stronę *http://localhost:3000/*. Zobaczysz dane wyjściowe widoczne na rysunku 4.4.

[1] W systemie Windows zmienne środowiskowe w tej sekcji należy ustawiać za pomocą składni `set ZMIENNA=wartość&& node...`; tu będzie to `set DEBUG=fibonacci:*&& node...` — *przyp. tłum.*



Rysunek 4.4. Wygenerowana aplikacja

Domyślna aplikacja z platformy Express

Masz już działającą pustą aplikację z platformy Express. Zobacz teraz, co platforma dla Ciebie wygenerowała. W ten sposób zaznajomisz się z nią przed rozpoczęciem pisania kodu aplikacji do wyznaczania liczb Fibonacciego.

Ponieważ użyta została opcja `--ejs`, aplikacja korzysta z silnika szablonów EJS. Wybrany silnik szablonów odpowiada za generowanie kodu w HTML-u. Dokumentację silnika EJS znajdziesz na stronie *http://ejs.co/*.

W katalogu *views* znajdują się dwa pliki: *error.ejs* i *index.ejs*. Są to szablony EJS, które później zmodyfikujesz.

Katalog *routes* zawiera wstępną konfigurację routera, czyli kod do obsługi konkretnych adresów URL. Później wprowadzisz w nim zmiany.

Katalog *public* obejmuje zasoby, których aplikacja nie generuje, lecz wysyła do przeglądarki. Początkowo znajduje się tu plik stylów CSS *public/stylesheets/style.css*.

Plik *package.json* zawiera zależności i inne metadane.

Katalog *bin* obejmuje wspomniany wcześniej skrypt *www*. Jest to skrypt platformy Node.js, który inicjalizuje obiekty `serwera HTTP`, rozpoczyna oczekiwanie w porcie TCP i wywołuje ostatni z omawianych tu plików, *app.js*. Skrypty inicjalizują platformę Express, konfigurują moduły odpowiedzialne za routing i wykonują inne operacje.

W skryptach *www* i *app.js* wykonywanych jest wiele zadań. Najpierw warto omówić inicjalizowanie aplikacji. Przyjrzyj się kilku wierszom z pliku *app.js*:

```
var express = require('express');
…
var app = express();
…
module.exports = app;
```

Oznaczają one, że *app.js* to moduł eksportujący obiekt zwracany przez moduł `express`.

Pora wrócić do skryptu *www*. Zauważ, że rozpoczyna się on od następującego wiersza:

```
#!/usr/bin/env node
```

Jest to stosowana w systemach Unix i Linux technika tworzenia poleceń skryptowych. Ten kod informuje, że dalszy kod należy uruchamiać jako skrypt wywoływany za pomocą polecenia `node`.

Ten skrypt korzysta z modułu *app.js* w następujący sposób:

```
var app = require('../app');
…
var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);
…
var server = http.createServer(app);
…
server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
```

Widać teraz, dlaczego używany jest port `3000`. Jest to parametr funkcji `normalizePort`. Widać też, że ustawienie zmiennej środowiskowej `PORT` spowoduje zastąpienie portu domyślnego `3000`. Spróbuj uruchomić następujące polecenie:

```
$ PORT=4242 DEBUG=fibonacci:* npm start
```

Teraz aplikacja informuje, że oczekuje na żądania w porcie `4242`, gdzie możesz zastanowić się nad sensem życia[2].

Obiekt `app` jest przekazywany do metody `http.createServer()`. Z dokumentacji Node.js można się dowiedzieć, że metoda ta przyjmuje funkcję `requestListener`, pobierającą obiekty `request` i `response`. Z kodu wynika, że obiekt `app` jest taką właśnie funkcją.

W ostatnim kroku skrypt *www* uruchamia serwer oczekujący na żądania w podanym porcie.

[2] Liczba 42 była odpowiedzią na „wielkie pytanie o życie, wszechświat i całą resztę" w książce *Autostopem przez galaktykę* Douglasa Adamsa — *przyp. tłum.*

Przyjrzyj się teraz dokładniej plikowi *app.js*:

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

Te instrukcje informują, że Express szuka szablonów w katalogu *views* i używa silnika szablonów EJS.

Funkcja `app.set` służy do konfigurowania właściwości aplikacji. W trakcie lektury tych punktów warto zaglądać do dokumentacji API platformy Express (*http://expressjs.com/en/4x/api.html*).

Dalej znajduje się seria wywołań `app.use`:

```
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

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

Funkcja `app.use` instaluje funkcje warstwy pośredniej. „Warstwa pośrednia" to ważne określenie z żargonu związanego z platformą Express; wkrótce znajdziesz objaśnienie tego pojęcia, a na razie zapamiętaj, że funkcje warstwy pośredniej są wykonywane w trakcie przetwarzania tras. To oznacza, że w pliku *app.js* włączane są następujące mechanizmy:

- zapis żądań w dzienniku przy użyciu rejestratora żądań Morgan (dokumentację tego narzędzia znajdziesz na stronie *https://www.npmjs.com/package/morgan*);
- moduł `body-parser` obsługujący parsowanie treści żądań HTTP (dokumentację znajdziesz na stronie *https://www.npmjs.com/package/body-parser*);
- moduł `cookie-parser` służący do parsowania plików cookie z protokołu HTTP (dokumentację znajdziesz na stronie *https://www.npmjs.com/package/cookie-parser*);
- serwer WWW plików statycznych skonfigurowany do udostępniania plików zasobów z katalogu *public*;
- dwa moduły routera, `routes` i `users`, określające, które funkcje mają obsługiwać poszczególne adresy URL.

Warstwa pośrednia w platformie Express

Na zakończenie omawiania skryptu *app.js* warto wyjaśnić, jakie jest zadanie funkcji warstwy pośredniej w aplikacji. Przykład znajduje się na końcu tego skryptu:

```
app.use(function(req, res, next) {
  var err = new Error('Not found');
  err.status = 404;
  next(err);
});
```

W komentarzu napisane jest, że ten kod „przechwytuje błąd 404 i przekazuje go do metody obsługi błędów". Zapewne wiesz, że status `HTTP 404` oznacza, iż szukanego zasobu nie znaleziono. Należy poinformować użytkownika, że żądanie nie zostało spełnione, a tu pokazany jest pierwszy etap tego procesu. Zanim przejdziesz do ostatniego kroku zgłaszania tego błędu, musisz dowiedzieć się, jak działa warstwa pośrednia.

Możesz przyjrzeć się funkcji warstwy pośredniej. Zajrzyj do jej dokumentacji na stronie *http://expressjs.com/en/guide/writing-middleware.html*.

Funkcje warstwy pośredniej przyjmują trzy argumenty. Dwa pierwsze, `request` i `response`, to odpowiedniki obiektów `request` i `response` obiektu żądania HTTP z platformy Node.js. Platforma Express wzbogaca jednak te obiekty o dodatkowe dane i możliwości. Ostatni argument, `next`, to wywoływana zwrotnie funkcja kontrolująca moment zakończenia cyklu przesyłania żądań i odpowiedzi. Można ją wykorzystać do przekazywania błędów dalej w łańcuchu funkcji warstwy pośredniej.

Przychodzące żądanie jest obsługiwane przez pierwszą funkcję warstwy pośredniej, potem przez drugą, następną, jeszcze kolejną itd. Za każdym razem, gdy żądanie ma być przekazane dalej w łańcuchu funkcji warstwy pośredniej, wywoływana jest funkcja `next`. Jeśli zostanie ona wywołana dla obiektu błędu (tak jak pokazałem wcześniej), oznacza to błąd. W przeciwnym razie sterowanie jest przekazywane do następnej funkcji warstwy pośredniej z łańcucha.

Co się stanie, jeśli funkcja `next` nie zostanie wywołana? Żądanie HTTP *zawiesi się*, ponieważ nie będzie dla niego odpowiedzi. Funkcje warstwy pośredniej zwracają odpowiedź, gdy wywołują funkcje z obiektu `response` (na przykład `res.send` lub `res.render`).

Przyjrzyj się na przykład dodanemu fragmentowi z pliku *app.js*:

```
app.get('/', function(req, res) {
    res.send('Witaj, świecie!');
});
```

Ten kod nie wywołuje funkcji `next`, lecz `res.send`. Jest to prawidłowa metoda kończenia cyklu żądanie – odpowiedź, polegająca na przesłaniu odpowiedzi (`res.send`) na żądanie. Jeśli nie jest wywoływana ani funkcja `next`, ani `res.send`, żądanie nigdy nie doczeka się odpowiedzi.

Tak więc funkcja warstwy pośredniej wykonuje jedno z czterech następujących zadań:

- Wykonuje własną logikę biznesową. Przykładową funkcją tego typu jest wspomniany wcześniej rejestrator żądań w warstwie pośredniej.
- Modyfikuje obiekt żądania lub odpowiedzi. Tak działają narzędzia `body-parser` i `cookie-parser`, wyszukujące dane do dodania do obiektu `request`.
- Wywołuje funkcję `next`, aby przejść do następnej funkcji warstwy pośredniej lub zasygnalizować błąd.
- Przesyła odpowiedź, kończąc cały cykl.

Kolejność wykonywania funkcji warstwy pośredniej zależy od kolejności ich dodawania do obiektu `app`. Najpierw jest wykonywana pierwsza funkcja, potem druga itd.

Warstwa pośrednia i ścieżki żądań

Do tej pory poznałeś dwa rodzaje funkcji warstwy pośredniej. W funkcji pierwszego typu pierwszym argumentem jest funkcja obsługi zdarzeń. W funkcji drugiego typu pierwszym argumentem jest łańcuch znaków z fragmentem adresu URL, a drugim funkcja obsługi zdarzeń.

Funkcja `app.use` przyjmuje opcjonalny pierwszy argument ze ścieżką, w jakiej warstwa pośrednia jest *zamontowana*. Ta ścieżka jest za pomocą wzorców dopasowywana do adresu URL żądania. Jeśli adres pasuje do wzorca, wywoływana jest dana funkcja. Dostępna jest nawet technika podawania nazwanych parametrów w adresie URL:

```
app.use('/user/profile/:id', function(req, res, next) {
    userProfiles.lookup(req.params.id, (err, profile) => {
        if (err) return next(err);
        // Wykonywanie operacji na profilu
        // (na przykład wyświetlanie go użytkownikowi)
        res.send(profile.display());
    });
});
```

W specyfikacji tej ścieżki występuje wzorzec, `:id`, a przekazana wartość jest zapisywana w polu `req.params.id`. Ten przykład może dotyczyć usługi związanej z profilami użytkowników, w której na podstawie adresu URL wyświetlane mają być informacje o określonym użytkowniku.

Innym sposobem stosowania funkcji warstwy pośredniej jest używanie ich dla konkretnego rodzaju żądań HTTP. Wywołanie `app.use` pasuje do dowolnych żądań, jednak w rzeczywistości żądania `GET` powinny działać inaczej niż żądania `POST`. W wywołaniach `app.`*`METODA`* człon *`METODA`* powinien odpowiadać jednemu z czasowników oznaczających żądania HTTP. Tak więc wywołanie `app.get` pasuje do żądań `GET`, `app.post` pasuje do żądań `POST` itd.

Na końcu warto wspomnieć o obiekcie `router`. Jest to rodzaj warstwy pośredniej używany bezpośrednio do routingu żądań na podstawie adresów URL. Przyjrzyj się zawartości pliku *routes/users.js*:

```
var express = require('express');
var router = express.Router();
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});
module.exports = router;
```

Jest to moduł, którego obiektem `exports` jest obiekt `router`. Tu obiekt `router` używa tylko jednej trasy, jeśli jednak uznasz to za uzasadnione, może on obsługiwać dowolną liczbę tras.

W pliku *add.js* pokazany moduł jest dodawany w następujący sposób:

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

Wszystkie opisane funkcje obiektu `app` dotyczą też obiektu `router`. Jeśli żądanie pasuje do wzorca, obiekt `router` przekazuje żądanie do własnego łańcucha przetwarzających je funkcji. Ważnym aspektem jest to, że gdy żądanie jest przekazywane do obiektu `router`, usuwany jest przedrostek z adresu URL żądania.

Zauważ, że wywołanie `router.get` z pliku *users.js* dopasowuje wzorzec `'/'`, a router jest tu zamontowany w katalogu `'/users'`. W efekcie wywołanie `router.get` dopasowuje też adres `/users`, ale ponieważ przedrostek jest usuwany, w kodzie występuje wzorzec `'/'`. To oznacza, że router można zamontować także w innych katalogach bez konieczności zmiany jego kodu.

Obsługa błędów

Teraz możesz wreszcie wrócić do wygenerowanego pliku *app.js*, błędu `404` (*strony nie znaleziono*) i innych błędów, które aplikacja może wyświetlać użytkownikowi.

Funkcja warstwy pośredniej informuje o błędzie, przekazując wartość do funkcji `next`. Gdy Express wykrywa błąd, pomija dalsze trasy niezwiązane z błędami i przekazuje dane do funkcji obsługi błędów. Funkcje obsługi błędów mają inne sygnatury niż funkcje pokazane wcześniej.

W analizowanym pliku *app.js* funkcja obsługi błędów wygląda tak:

```
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});
```

Funkcje obsługi błędów przyjmują cztery parametry: obok znanych Ci już `req`, `res` i `next` podawany jest też parametr `err`. W pokazanej tu funkcji używane są: wywołanie `res.status` (do ustawienia kodu odpowiedzi HTTP) oraz wywołanie `res.render` (do sformatowania odpowiedzi z kodem w HTML-u za pomocą szablonu *views/error.ejs*). Funkcja `res.render` przyjmuje dane i wyświetla je za pomocą szablonu, generując kod w HTML-u.

Wszystkie błędy z aplikacji trafiają do pokazanej funkcji z pominięciem wszystkich pozostałych funkcji warstwy pośredniej.

Wyznaczanie liczb ciągu Fibonacciego za pomocą aplikacji z platformy Express

Liczby Fibonacciego tworzą ciąg całkowitoliczbowy: *0, 1, 1, 2, 3, 5, 8, 13, 21, 34…*

Każda pozycja na tej liście to suma dwóch poprzednich elementów. Ciąg ten został wymyślony w 1202 roku przez Leonarda z Pizy zwanego Fibonaccim. Jedną z technik wyznaczania liczb tego ciągu jest pokazany wcześniej algorytm rekurencyjny. Dalej za pomocą platformy Express utworzysz aplikację, która wykorzystuje ten algorytm. Następnie poznasz kilka metod łagodzenia problemów z wydajnością wynikających ze stosowania wymagających obliczeniowo algorytmów.

Zacznij od pustej aplikacji utworzonej w poprzednim kroku. Nie bez powodu nazwaliśmy ją `fibonacci`. Przewidziałem, co się stanie.

W pliku *app.js* wprowadź następujące zmiany:

- Usuń wiersz `require('./routes/users')` i zastąp go kodem `var fibonacci = require('./routes/fibonacci')`.
- Usuń wiersz `app.use('/users', users)` i zastąp go kodem `app.use('/fibonacci', fibonacci)`.

W aplikacji do wyznaczania liczb Fibonacciego nie jest potrzebna obsługa użytkowników. Przyda się natomiast strona do pobierania liczby ciągu Fibonacciego, której wartość program ma wyznaczyć. Do wyznaczania tej wartości posłuży pokazany dalej moduł `fibonacci`.

W katalogu z najwyższego poziomu utwórz plik *math.js* zawierający ten oto bardzo prosty algorytm wyznaczania liczb ciągu Fibonacciego:

```
var fibonacci = exports.fibonacci = function(n) {
    if (n === 1)       return 1;
    else if (n === 2)  return 1;
    else               return fibonacci(n-1) + fibonacci(n-2);
}
```

W katalogu *views* utwórz plik o nazwie *top.ejs*:

```
<html>
<head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
    <h1><%= title %></h1>
    <div class='navbar'>
    <p><a href='/'>Strona główna</a>
     | <a href='/fibonacci'>Liczby Fibonacciego</a></p>
    </div>
```

Ten plik zawiera górną część HTML-owych stron przesyłanych użytkownikom.

Inny plik, *bottom.ejs*, zawiera następujący kod:

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

Ten plik obejmuje dolną część HTML-owych stron.

Zmodyfikuj plik *view/index.ejs* w taki sposób, by zawierał tylko następujący kod:

```
<% include top %>
<p>Witaj w kalkulatorze</p>
<% include bottom %>
```

To będzie strona główna aplikacji. Choć nie udostępnia ona żadnych funkcji, zauważ, że w pliku *top.ejs* znajduje się odnośnik do strony */fibonacci*, którą wkrótce się zajmiesz.

Pliki *top.ejs* i *bottom.ejs* razem zapewniają spójny wygląd i układ stron bez konieczności powielania kodu na każdej stronie.

Teraz utwórz plik *views/fibonacci.ejs* i umieść w nim następujący kod:

```
<% include top %>
<% if (typeof fiboval !== "undefined") { %>
  <p>Liczba Fibonacciego numer <%= fibonum %> wynosi <%= fiboval %></p>
  <hr/>
<% } %>
<p>Wprowadź liczbę, aby zobaczyć jej wartość w ciągu Fibonacciego</p>
<form name='fibonacci' action='/fibonacci' method='get'>
<input type='text' name='fibonum' />
<input type='submit' value='Wyślij' />
</form>
<% include bottom %>
```

> Pamiętaj, że pliki z katalogu *views* to szablony służące do wyświetlania danych. Te pliki odpowiadają za **widok** z paradygmatu **model – widok – kontroler** (stąd nazwa katalogu *views*, czyli „widoki").

Każdy z dwóch widoków, *views/index.ejs* i *views/fibonacci.ejs*, to fragment kompletnych stron HTML-owych. Skąd pobierana jest reszta tych stron? Z plików *top.ejs* i *bottom.ejs*, które są dołączane przez każdy szablon, aby zapewnić spójny układ stron.

W katalogu *routes* usuń moduł *user.js*. Ten moduł jest generowany przez Express, ale nie będzie potrzebny w omawianej aplikacji.

W pliku *routes/index.js* zmodyfikuj obiekt `router` w następujący sposób:

```
/* Pobieranie strony głównej */
router.get('/', function(req, res, next) {
  res.render('index', { title: "Kalkulator" });
});
```

W ostatnim kroku utwórz w katalogu *routes* plik *fibonacci.js* zawierający następujący kod:

```
var express = require('express');
var router = express.Router();

var math = require('../math');
router.get('/', function(req, res, next) {
  if (req.query.fibonum) {
    // Obliczenia są przeprowadzane bezpośrednio na serwerze
    res.render('fibonacci', {
      title: "Wyznaczanie liczb Fibonacciego",
      fibonum: req.query.fibonum,
      fiboval: math.fibonacci(req.query.fibonum)
    });
  } else {
    res.render('fibonacci', {
      title: "Wyznaczanie liczb Fibonacciego",
      fiboval: undefined
    });
  }
});

module.exports = router;
```

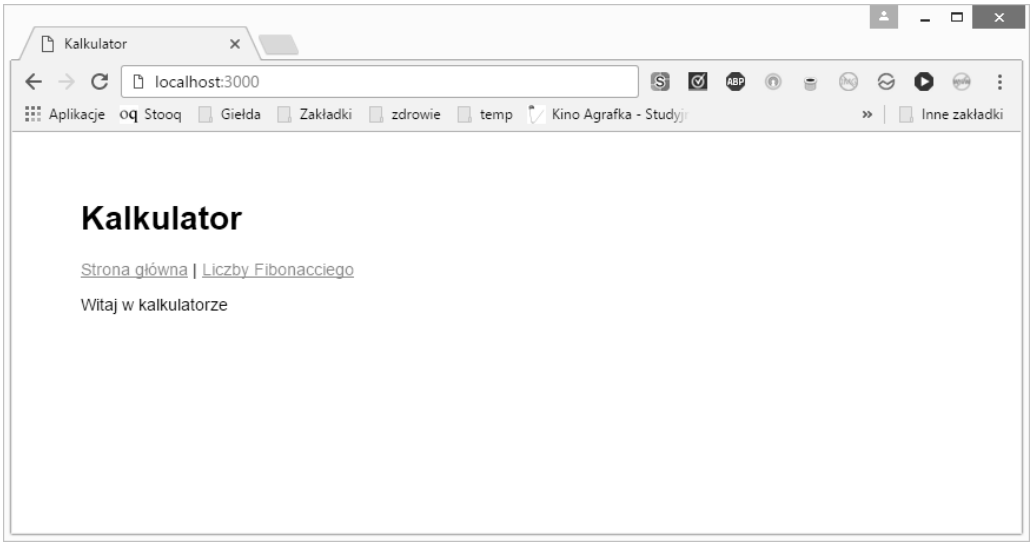
W pliku *package.json* zmodyfikuj sekcję `scripts` w następujący sposób:

```
"scripts": {
  "start": "DEBUG=fibonacci:* node ./bin/www"
},
```

To pozwala zastosować wywołanie `npm start` do uruchomienia skryptu i na stałe włączyć komunikaty diagnostyczne. Teraz możesz uruchomić skrypt:

```
$ npm start

> fibonacci@0.0.0 start /Users/david/chap04/fibonacci
> DEBUG=fibonacci:* node ./bin/www

  fibonacci:server Listening on port 3000 +0ms
```

Zgodnie z tymi informacjami można otworzyć stronę *http://localhost:3000/* i zobaczyć jej zawartość (rysunek 4.5).



Rysunek 4.5. Aplikacja do wyznaczania liczb ciągu Fibonacciego

Ta strona jest generowana na podstawie szablonu *views/index.ejs*. Kliknij odnośnik *Liczby Fibonacciego*, aby przejść do następnej strony, generowanej na podstawie szablonu *views/fibonacci.ejs*. Na tej stronie możesz wprowadzić liczbę i kliknąć przycisk *Wyślij*, a pojawi się odpowiedź (wskazówka: wybierz liczbę mniejszą niż 40, jeśli chcesz uzyskać odpowiedź w rozsądnym czasie). Efekt pokazany jest na rysunku 4.6.

Rysunek 4.6. Wyświetlona wartość z ciągu Fibonacciego

Warto prześledzić pracę aplikacji i omówić jej działanie.

Plik *app.js* zawiera dwie trasy — dla ukośnika (/), obsługiwanego przez plik *routes/index.js*, i dla członu */fibonacci*, obsługiwanego przez plik *routes/fibonacci.js*.

Funkcja `res.render` wyświetla szablon o podanej nazwie. Używa przekazanych wartości i generuje wynik jako odpowiedź HTTP. Dla strony głównej tej aplikacji kod generujący stronę (*routes/index.js*) i szablon (*views/index.ejs*) są bardzo proste. To na stronie do wyznaczania wartości ciągu Fibonacciego dzieje się cała akcja.

Szablon *views/fibonacci.ejs* obejmuje formularz, w którym użytkownik może wprowadzić liczbę. Ponieważ jest to formularz typu `GET`, to gdy użytkownik kliknie przycisk *Wyślij*, przeglądarka zgłosi żądanie HTTP `GET` strony o adresie URL */fibonacci*. Tym, co odróżnia jedno żądanie tego rodzaju od innych, jest to, czy adres URL obejmuje parametry zapytania `fibonum`. Gdy użytkownik otwiera stronę po raz pierwszy, parametr `fibonum` nie jest podany, dlatego nie ma czego obliczać. Po wprowadzeniu liczby przez użytkownika i kliknięciu przycisku *Wyślij* parametr jest dostępny i aplikacja może obliczyć wartość.

Express automatycznie parsuje parametry zapytania i udostępnia je w polu `req.query`. To oznacza, że w pliku *routes/fibonacci.js* można szybko sprawdzić, czy dostępny jest parametr `fibonum`. Jeśli tak jest, wywołana zostaje funkcja `fibonacci` obliczająca szukaną wartość.

Wcześniej sugerowałem, by wpisać wartość mniejszą niż `40`. Teraz wprowadź większą liczbę, na przykład `50`, i zrób sobie przerwę na kawę, ponieważ obliczenie szukanej wartości zajmie dużo czasu.

Kod wymagający obliczeniowo a pętla zdarzeń w platformie Node.js

Przykładowy kod do wyznaczania liczb Fibonacciego celowo jest niewydajny. Pozwoli to zilustrować ważną kwestię dotyczącą aplikacji. Co się dzieje z pętlą zdarzeń w Node.js, gdy wykonywane są długie obliczenia? Aby zobaczyć ten efekt, otwórz dwa okna przeglądarki i w każdym wyświetl stronę do wyznaczania liczb Fibonacciego. Na jednej stronie wpisz liczbę `55` lub większą, a na drugiej `10`. Zauważ, że drugie okno będzie zablokowane. Jeśli pozwolisz obu stronom działać wystarczająco długo, ostatecznie w obu oknach pojawią się wyniki. W programie dzieje się to, że pętla zdarzeń nie może przetwarzać zdarzeń, ponieważ algorytm wyznaczania liczb Fibonacciego wykonuje swe zadanie i nie przekazuje sterowania do tej pętli.

Ponieważ Node.js używa jednego wątku wykonania, przetwarzanie żądań wymaga, by funkcje obsługi żądań szybko zwracały sterowanie do pętli zdarzeń. Asynchroniczny styl pisania kodu gwarantuje, że pętla zdarzeń jest regularnie aktywowana. Dotyczy to nawet żądań wczytujących dane z serwera po drugiej stronie świata. Jest tak, ponieważ asynchroniczne operacje wejścia – wyjścia są nieblokujące i sterowanie szybko zostaje zwrócone do pętli zdarzeń.

Wybrany tu naiwny algorytm wyznaczania liczb Fibonacciego nie pasuje do tego modelu, ponieważ jest długą operacją blokującą. Tego rodzaju funkcje obsługi zdarzeń uniemożliwiają systemowi przetwarzanie żądań i nie pozwalają platformie Node.js robić tego, do czego jest przeznaczona — czyli być niezwykle szybkim serwerem WWW.

W tej sytuacji problem z długim czasem odpowiedzi jest oczywisty. Czas szybko wydłuża się do tego stopnia, że możesz zrobić sobie wakacje w Tybecie, czekając na zwrócenie przez aplikację liczby Fibonacciego!

Aby wyraźniej to zobaczyć, utwórz plik *fibotimes.js* zawierający następujący kod:

```
var math = require('./math');
var util = require('util');

for (var num = 1; num < 80; num++) {
    util.log('Liczba Fibonacciego numer '+ num +' = '+ math.fibonacci(num));
}
```

Teraz uruchom ten kod. Otrzymasz następujące dane wyjściowe:

```
$ node fibotimes.js
31 Jan 14:41:28 - Liczba Fibonacciego numer 1 = 1
31 Jan 14:41:28 - Liczba Fibonacciego numer 2 = 1
31 Jan 14:41:28 - Liczba Fibonacciego numer 3 = 2
31 Jan 14:41:28 - Liczba Fibonacciego numer 4 = 3
31 Jan 14:41:28 - Liczba Fibonacciego numer 5 = 5
31 Jan 14:41:28 - Liczba Fibonacciego numer 6 = 8
31 Jan 14:41:28 - Liczba Fibonacciego numer 7 = 13
31 Jan 14:41:28 - Liczba Fibonacciego numer 8 = 21
31 Jan 14:41:28 - Liczba Fibonacciego numer 9 = 34
…
31 Jan 14:42:27 - Liczba Fibonacciego numer 38 = 39088169
31 Jan 14:42:28 - Liczba Fibonacciego numer 39 = 63245986
31 Jan 14:42:31 - Liczba Fibonacciego numer 40 = 102334155
31 Jan 14:42:34 - Liczba Fibonacciego numer 41 = 165580141
31 Jan 14:42:40 - Liczba Fibonacciego numer 42 = 267914296
31 Jan 14:42:50 - Liczba Fibonacciego numer 43 = 433494437
31 Jan 14:43:06 - Liczba Fibonacciego numer 44 = 701408733
```

Ten kod szybko wyznacza około 40 pierwszych elementów ciągu Fibonacciego, ale później uzyskanie każdego następnego wyniku zajmuje po kilka sekund i ten czas stale się wydłuża. Wykonywanie tego rodzaju kodu w systemie jednowątkowym, wymagającym szybkiego zwracania sterowania do pętli zdarzeń, jest nieakceptowalne.

W Node.js istnieją dwa ogólne rozwiązania tego problemu:

- **Zmiana algorytmu**. Możliwe, że (tak jak w przypadku użytej tu funkcji do wyznaczania liczb Fibonacciego) jeden z zastosowanych algorytmów jest nieoptymalny i można go zapisać tak, by działał szybciej. A jeśli nie można go przyspieszyć, może uda się rozbić go na wywołania zwrotne uruchamiane z poziomu pętli zdarzeń. Jednej z tego rodzaju technik przyjrzysz się już wkrótce.

- **Utworzenie usługi działającej na zapleczu**. Czy potrafisz wyobrazić sobie działający na zapleczu serwer przeznaczony do wyznaczania liczb Fibonacciego? Możliwe, że byłaby to przesada, jednak programiści często tworzą działające na zapleczu serwery, aby odciążyć serwery frontonu. W końcowej części tego rozdziału napiszesz działający na zapleczu serwer do wyznaczania liczb Fibonacciego.

Zmiana algorytmu

Aby udowodnić, że omawiany problem jest sztuczny, przedstawiam znacznie wydajniejszą funkcję do wyznaczania liczb Fibonacciego:

```
var fibonacciLoop = exports.fibonacciLoop = function(n) {
    var fibos = [];
    fibos[0] = 0;
    fibos[1] = 1;
    fibos[2] = 1;
    for (var i = 3; i <= n; i++) {
        fibos[i] = fibos[i-2] + fibos[i-1];
    }
    return fibos[n];
}
```

Jeśli zastąpisz wywołanie `math.fibonacci` poleceniem `math.fibonacciLoop`, wcześniejsze programy zaczną działać znacznie szybciej. Jednak nawet ta nowa wersja nie jest maksymalnie wydajna. Znacznie szybsza (choć wymagająca nieco więcej pamięci) będzie na przykład wersja z prostą, obliczoną wcześniej tabelą wyszukiwania.

Zmodyfikuj plik *fibotimes.js* w pokazany poniżej sposób i ponownie uruchom skrypt. Liczby będą pojawiać się tak szybko, że nie nadążysz z ich śledzeniem:

```
for (var num = 1; num < 8000; num++) {
    // util.log('Liczba Fibonacciego numer '+ num +' = '+
    // math.fibonacci(num));
    util.log('Liczba Fibonacciego numer '+ num +' = '+
        math.fibonacciLoop(num));
}
```

Optymalizacja niektórych algorytmów nie jest tak prosta i mimo zmian obliczanie wyniku może wymagać dużo czasu. W tym punkcie opisuję, jak radzić sobie z niewydajnymi algorytmami. Dlatego tematem dalszych rozważań będzie niewydajna wersja algorytmu do wyznaczania liczb Fibonacciego.

Obliczenia można podzielić na porcje, a potem uruchamiać odpowiadający im kod z poziomu pętli zdarzeń. Dodaj do pliku *math.js* następujący kod:

```
var fibonacciAsync = exports.fibonacciAsync = function(n, done) {
    if (n === 0)
        done(undefined, 0);
    else if (n === 1 || n === 2)
        done(undefined, 1);
```

```
    else {
        setImmediate(function() {
            fibonacciAsync(n-1, function(err, val1) {
                if (err) done(err);
                else setImmediate(function() {
                    fibonacciAsync(n-2, function(err, val2) {
                        if (err) done(err);
                        else done(undefined, val1+val2);
                    });
                });
            });
        });
    }
}
```

To powoduje przekształcenie funkcji `fibonacci` z wersji synchronicznej w funkcję asynchroniczną z jednym wywołaniem zwrotnym. Dzięki wywołaniu `setImmediate` każdy etap obliczeń będzie zarządzany z poziomu pętli zdarzeń Node.js, a serwer będzie mógł łatwo obsługiwać inne żądania w trakcie przeprowadzania obliczeń. Ta technika w żaden sposób nie zmniejsza zakresu potrzebnych obliczeń. Nadal używany jest głupi i niewydajny algorytm wyznaczania liczb Fibonacciego. W kodzie jedynie rozbito obliczenia na porcje z wykorzystaniem pętli zdarzeń.

Ponieważ używana jest funkcja asynchroniczna, trzeba zmienić kod aplikacji w pliku *router/fibonacci.js* na następującą postać:

```
var express = require('express');
var router = express.Router();

var math = require('../math');
router.get('/', function(req, res, next) {
  if (req.query.fibonum) {
    math.fibonacciAsync(req.query.fibonum, (err, fiboval) => {
      res.render('fibonacci', {
        title: "Wyznaczanie liczb Fibonacciego",
        fibonum: req.query.fibonum,
        fiboval: fiboval
      });
    });
  } else {
    res.render('fibonacci', {
      title: "Wyznaczanie liczb Fibonacciego",
      fiboval: undefined
    });
  }
};
```

Po tej zmianie serwer nie zawiesza się w trakcie wyznaczania dużych liczb Fibonacciego. Obliczenia oczywiście nadal zajmują dużo czasu, ale przynajmniej nie blokują innym użytkownikom dostępu do aplikacji.

Aby się o tym przekonać, ponownie otwórz aplikację w dwóch oknach przeglądarki. W jednym wpisz liczbę 55, a w drugim żądaj obliczania mniejszych liczb Fibonacciego. Inaczej niż wcześniej w drugim oknie liczby Fibonacciego będą wyświetlane jeszcze w trakcie wykonywania obliczeń w pierwszym.

To od Ciebie i używanego algorytmu zależy, która technika optymalizacji kodu będzie najlepsza i jak poradzić sobie z długimi obliczeniami.

Zgłaszanie żądań za pomocą klienta HTTP

Drugim sposobem złagodzenia problemów z wymagającym obliczeniowo kodem jest przekazanie obliczeń do procesu działającego na zapleczu. Aby zapoznać się z tą techniką, zażądasz wykonania obliczeń od działającego na zapleczu serwera do wyznaczania liczb Fibonacciego. Posłużysz się przy tym obiektem klienta HTTP. Najpierw jednak warto ogólnie omówić korzystanie z tego obiektu.

Node.js obejmuje obiekt klienta HTTP przydatny do zgłaszania żądań HTTP. Umożliwia on zgłaszanie żądań dowolnego rodzaju, natomiast nie symuluje przeglądarki, dlatego nie łudź się, że jest to kompletne narzędzie do automatyzacji testów. Jego zasięg jest ograniczony do protokołu HTTP. Na podstawie klienta HTTP można jednak zbudować emulator przeglądarki, na przykład w celu utworzenia narzędzia do automatyzacji testów. Obiekt klienta HTTP można wykorzystać do zgłaszania dowolnych żądań HTTP, na przykład do wywoływania usług opartych na architekturze **REST** (ang. *Representational State Transfer*).

Zacznij od kodu inspirowanego poleceniami `wget` i `curl`. Ten kod posłuży do zgłaszania żądań HTTP i wyświetlania wyników. Utwórz plik *wget.js* o następującej zawartości:

```
var http = require('http');
var url = require('url');
var util = require('util');

var argUrl = process.argv[2];
var parsedUrl = url.parse(argUrl, true);

// Obiekt options jest przekazywany do wywołania http.request
// i informuje o docelowym adresie URL
var options = {
  host: parsedUrl.hostname,
  port: parsedUrl.port,
  path: parsedUrl.pathname,
  method: 'GET'
};

if (parsedUrl.search) options.path += "?"+parsedUrl.search;

var req = http.request(options);
```

```
// Wywoływane po zakończeniu przetwarzania żądania
req.on('response', res => {
  util.log('STATUS: ' + res.statusCode);
  util.log('NAGŁÓWKI: ' + util.inspect(res.headers));
  res.setEncoding('utf8');
  res.on('data', chunk => { util.log('TREŚĆ: ' + chunk); });
  res.on('error', err => { util.log('BŁĄD W ODPOWIEDZI: ' + err); });
});
// Wywoływane w reakcji na błędy
req.on('error', err => { util.log('BŁĄD W ŻĄDANIU: ' + err); });
req.end();
```

Ten skrypt możesz uruchomić w następujący sposób:

```
$ node wget.js http://example.com
31 Jan 15:04:29 - STATUS: 200
31 Jan 15:04:29 - NAGŁÓWKI: { 'accept-ranges': 'bytes',
  'cache-control': 'max-age=604800',
  'content-type': 'text/html',
  date: 'Sun, 31 Jan 2016 23:04:29 GMT',
  etag: '"359670651+gzip"',
  expires: 'Sun, 07 Feb 2016 23:04:29 GMT',
  'last-modified': 'Fri, 09 Aug 2013 23:54:35 GMT',
  server: 'ECS (rhv/818F)',
  vary: 'Accept-Encoding',
  'x-cache': 'HIT',
  'x-ec-custom-error': '1',
  'content-length': '1270',
  connection: 'close' }
```

Skrypt wyświetla też inne informacje, a mianowicie HTML-owy kod strony o adresie *http://example.com/*.

Zadanie skryptu *wget.js* polega na tym, by zgłaszać żądanie HTTP i wyświetlać szczegółowe informacje o odpowiedzi.

Do inicjalizowania żądań HTTP służy metoda `http.request`:

```
var http = require('http');
var options = {
  host: 'example.com',
  port: 80,
  path: null,
  method: 'GET'
};
var request = http.request(options,
  function(response) { .. });
```

Obiekt `options` opisuje zgłaszane żądanie, a wywoływana zwrotnie funkcja jest uruchamiana po otrzymaniu odpowiedzi. Obiekt ten ma prostą budowę — obejmuje pola `host`, `port` i `path` określające żądany adres URL. W polu `method` należy podać jeden z rodzajów żądań HTTP

(GET, PUT, POST itd.). Możesz też dodać tablicę headers z nagłówkami żądania HTTP. Możliwe, że chcesz utworzyć plik cookie:

```
var options = {
  headers: {
    'Cookie': '… wartość pliku cookie'
  }
};
```

Obiekt response jest typu EventEmitter i generuje zdarzenia data i error. Zdarzenie data jest wywoływane po otrzymaniu danych, a zdarzenie error — w reakcji na błędy.

Obiekt request jest typu WritableStream. Jest on przydatny dla żądań HTTP zawierających dane (na przykład dla żądań PUT i POST). Obiekt ten udostępnia funkcję write zapisującą dane u nadawcy żądania. Format danych żądania HTTP jest określany za pomocą standardu MIME, opracowanego pierwotnie w celu usprawnienia poczty elektronicznej. W 1992 roku społeczność internetowa we współpracy z komitetem odpowiedzialnym za standard MIME dostosowała fragmenty tego standardu do użytku w protokole HTTP. Na przykład w formularzach HTML-owych typ zawartości (Content-Type) to multipart/form-data.

Wywoływanie usługi w architekturze REST z zaplecza z poziomu aplikacji opartej na platformie Express

Skoro wiesz już, jak zgłaszać żądania za pomocą klienta HTTP, możesz zobaczyć, jak zgłaszać zapytania do usługi w architekturze REST w aplikacjach internetowych opartych na platformie Express. Wymaga to zgłoszenia żądania HTTP GET do działającego na zapleczu serwera, który zwraca liczbę ciągu Fibonacciego określoną w adresie URL. Aby uzyskać ten efekt, zrefaktoryzujesz aplikację, tworząc serwer zwracający liczby ciągu Fibonacciego w odpowiedzi na wywołania z tej aplikacji. Choć w kontekście wyznaczania liczb Fibonacciego jest to przesada, przedstawione tu rozwiązanie pozwala przyjrzeć się podstawom implementowania aplikacji wielowarstwowych.

Wywoływanie usług w architekturze REST jest z natury operacją asynchroniczną. To oznacza, że korzystanie z usługi w architekturze REST obejmuje wywołanie funkcji inicjalizującej żądanie oraz użycie wywoływanej zwrotnie funkcji przyjmującej odpowiedź. Dostęp do usług w architekturze REST odbywa się przez protokół HTTP, dlatego do wywoływania usługi posłuży obiekt klienta HTTP.

Tworzenie prostego serwera w architekturze REST za pomocą platformy Express

Choć Express udostępnia rozbudowany system szablonów, dzięki czemu dobrze nadaje się do przesyłania HTML-owych stron internetowych do przeglądarek, można go wykorzystać także do utworzenia prostej usługi w architekturze REST. Pokazane wcześniej sparametryzowane adresy URL (*/user/profile/:id*) mogą posłużyć do przekazywania parametrów w wywołaniach usługi w architekturze REST. Express umożliwia też łatwe zwracanie danych w formacie JSON.

Utwórz teraz plik *fiboserver.js* z następującym kodem:

```
var math = require('./math');
var express = require('express');
var logger = require('morgan');
var app = express();
app.use(logger('dev'));
app.get('/fibonacci/:n', (req, res, next) => {
    math.fibonacciAsync(Math.floor(req.params.n), (err, val) => {
        if (err) next('FIBO SERVER ERROR ' + err);
        else {
            res.send({
                n: req.params.n,
                result: val
            });
        }
    });
});
app.listen(process.env.SERVERPORT);
```

Jest to oparta na platformie Express prosta aplikacja, która ma ilustrować, jak udostępnić usługę wyznaczania liczb Fibonacciego. Jedyna obsługiwana trasa służy do wyznaczania liczb Fibonacciego przy użyciu opisanych wcześniej funkcji.

Po raz pierwszy w tej książce używana jest funkcja `res.send`. Zapewnia ona swobodę w zakresie przesyłania odpowiedzi. Pozwala podać tablicę wartości nagłówka (na potrzeby nagłówka odpowiedzi HTTP) i kod odpowiedzi HTTP. Tu funkcja automatycznie wykrywa obiekt, przekształca go na tekst w formacie JSON i przesyła z odpowiednim typem zawartości (`Content-Type`).

Następnie w sekcji `scripts` pliku *package.json* dodaj ten fragment:

```
"server": "SERVERPORT=3002 node ./fiboserver"
```

To automatyzuje uruchamianie usługi wyznaczającej liczby Fibonacciego.

> Zauważ, że port TCP/IP jest tu podawany za pomocą zmiennej środowiskowej używanej w aplikacji. Jest to następny aspekt modelu Twelve-Factor, związany z umieszczaniem danych konfiguracyjnych w środowisku.

Pora uruchomić serwer:

```
$ npm run server

> fibonacci@0.0.0 server /Users/david/chap04/fibonacci
> SERVERPORT=3002 node ./fiboserver
```

Następnie w odrębnym oknie z wierszem poleceń można za pomocą programu `curl` zgłosić kilka żądań skierowanych do usługi:

```
$ curl -f http://localhost:3002/fibonacci/10
{"n":"10","result":55}
$ curl -f http://localhost:3002/fibonacci/11
{"n":"11","result":89}
$ curl -f http://localhost:3002/fibonacci/12
{"n":"12","result":144}
```

W oknie, w którym działa usługa, zobaczysz informacje o żądaniach `GET` i dowiesz się, ile czasu zajęło przetwarzanie każdego z nich.

Teraz utwórz prosty program kliencki, *fiboclient.js*, aby programowo wywoływać usługę wyznaczającą liczby Fibonacciego:

```
var http = require('http');
var util = require('util');
[
  "/fibonacci/30", "/fibonacci/20", "/fibonacci/10",
  "/fibonacci/9", "/fibonacci/8", "/fibonacci/7",
  "/fibonacci/6", "/fibonacci/5", "/fibonacci/4",
  "/fibonacci/3", "/fibonacci/2", "/fibonacci/1"
].forEach(path => {
    util.log('requesting ' + path);
    var req = http.request({
      host: "localhost",
      port: 3002,
      path: path,
      method: 'GET'
    }, res => {
      res.on('data', chunk => {
          util.log('BODY: ' + chunk);
      });
    });
    req.end();
});
```

Następnie w pliku *package.json* dodaj do sekcji `scripts` ten oto fragment:

```
"client": "node ./fiboclient"
```

Teraz możesz uruchomić aplikację kliencką:

```
$ npm run client

> fibonacci@0.0.0 client /Users/david/chap04/fibonacci
> node ./fiboclient

31 Jan 16:37:48 - requesting /fibonacci/30
31 Jan 16:37:48 - requesting /fibonacci/20
31 Jan 16:37:48 - requesting /fibonacci/10
31 Jan 16:37:48 - requesting /fibonacci/9
31 Jan 16:37:48 - requesting /fibonacci/8
31 Jan 16:37:48 - requesting /fibonacci/7
31 Jan 16:37:48 - requesting /fibonacci/6
31 Jan 16:37:48 - requesting /fibonacci/5
31 Jan 16:37:48 - requesting /fibonacci/4
31 Jan 16:37:48 - requesting /fibonacci/3
31 Jan 16:37:48 - requesting /fibonacci/2
31 Jan 16:37:48 - requesting /fibonacci/1
31 Jan 16:37:48 - TREŚĆ: {"n":"2","result":1}
31 Jan 16:37:48 - TREŚĆ: {"n":"1","result":1}
31 Jan 16:37:48 - TREŚĆ: {"n":"3","result":2}
31 Jan 16:37:48 - TREŚĆ: {"n":"4","result":3}
31 Jan 16:37:48 - TREŚĆ: {"n":"5","result":5}
31 Jan 16:37:48 - TREŚĆ: {"n":"6","result":8}
31 Jan 16:37:48 - TREŚĆ: {"n":"7","result":13}
31 Jan 16:37:48 - TREŚĆ: {"n":"8","result":21}
31 Jan 16:37:48 - TREŚĆ: {"n":"9","result":34}
31 Jan 16:37:48 - TREŚĆ: {"n":"10","result":55}
31 Jan 16:37:48 - TREŚĆ: {"n":"20","result":6765}
31 Jan 16:37:59 - TREŚĆ: {"n":"30","result":832040}
```

Dążysz do dodania usługi w architekturze REST do aplikacji internetowej. Do tej pory zobaczyłeś, że można wykonać kilka zadań. Jednym z nich jest możliwość wywołania usługi w architekturze REST i wykorzystania zwróconych wartości jako danych w programie.

Przy okazji zilustrowany został problem długich obliczeń. Zauważ, że żądania były zgłaszane od największej liczby do najmniejszej, natomiast wyniki pojawiały się w prawie dokładnie odwrotnej kolejności. Dlaczego? Z powodu czasu przetwarzania każdego żądania i niewydajnego algorytmu. Czas obliczeń rośnie na tyle szybko, że wyznaczanie wartości dla większych liczb trwa tak długo, iż kolejność odpowiedzi jest odwrotnością kolejności żądań.

Plik *fiboclient.js* od razu przesyła wszystkie żądania, a następnie każde z nich oczekuje na odpowiedź. Ponieważ serwer używa funkcji `fibonacciAsync`, jednocześnie wyznacza wszystkie wyniki. Najszybciej obliczane wartości będą gotowe jako pierwsze. Gdy odpowiedzi docierają do klienta, wywoływana jest pasująca funkcja obsługi zdarzeń, która wyświetla wynik w konsoli. Wyniki są przesyłane wtedy, gdy są gotowe, i ani milisekundę szybciej.

Przekształcanie aplikacji do wyznaczania liczb Fibonacciego na usługę w architekturze REST

Po utworzeniu opartego na architekturze REST serwera możesz wrócić do aplikacji wyznaczającej liczby Fibonacciego i wykorzystać zdobytą wiedzę do jej usprawnienia. Posłużysz się tu fragmentami kodu z pliku *fiboclient.js* i przeniesiesz je do aplikacji. Zmodyfikuj plik *routes/fibonacci.js* w taki sposób, aby zawierał następujący kod:

```
router.get('/', function(req, res, next) {
  if (req.query.fibonum) {
    var httpreq = require('http').request({
      host: "localhost",
      port: process.env.SERVERPORT,
      path: "/fibonacci/"+Math.floor(req.query.fibonum),
      method: 'GET'
    },
    httpresp => {
      httpresp.on('data', chunk => {
        var data = JSON.parse(chunk);
        res.render('fibonacci', {
          title: "Wyznaczanie liczb Fibonacciego",
          fibonum: req.query.fibonum,
          fiboval: data.result
        });
      });
      httpresp.on('error', err => { next(err); });
    });
    httpreq.on('error', err => { next(err); });
    httpreq.end();
  } else {
    res.render('fibonacci', {
      title: "Wyznaczanie liczb Fibonacciego",
      fiboval: undefined
    });
  }
});

module.exports = router;
```

Następnie w pakiecie *package.json* wprowadź w sekcji `scripts` następującą zmianę:

```
"start": "SERVERPORT=3002 DEBUG=fibonacci:* node ./bin/www",
```

Powoduje to powielenie konfiguracji środowiska używanej dla usługi i aplikacji do wyznaczania liczb Fibonacciego.

W jednym oknie wiersza poleceń uruchom serwer:

```
$ npm run server

> fibonacci@0.0.0 server /Users/david/chap04/fibonacci
> SERVERPORT=3002 node ./fiboserver
```

W drugim oknie włącz aplikację:

```
$ npm start

> fibonacci@0.0.0 start /Users/david/chap04/fibonacci
> SERVERPORT=3002 DEBUG=fibonacci:* node ./bin/www

  fibonacci:server Listening on port 3000 +0ms
```

Ponieważ szablony pozostały takie same, także ekran będzie wyglądał tak samo jak wcześniej.

Z tym rozwiązaniem związany jest nowy problem. Asynchroniczna implementacja niewydajnego algorytmu do wyznaczania liczb Fibonacciego może sprawić, że w procesie wykonującym ten algorytm szybko wyczerpie się pamięć. Na liście pytań i odpowiedzi dotyczących platformy (*https://github.com/nodejs/node/wiki/FAQ*) znajduje się sugestia, aby zastosować opcję `--max_old_space_size`. Można ją dodać do pliku *package.json* w następujący sposób:

```
"server": "SERVERPORT=3002 node ./fiboserver --max_old_space_size 5000",
```

Z tej samej listy pytań i odpowiedzi dowiesz się też, że jeśli występują problemy z przekraczaniem maksymalnej ilości pamięci, prawdopodobnie powinieneś wprowadzić w aplikacji zmiany. To prowadzi do kwestii poruszonej kilka stron wcześniej, dotyczącej tego, że istnieje kilka technik rozwiązywania problemów z wydajnością. Jedną z nich jest modyfikacja algorytmów aplikacji.

Po co wplątywać się w problemy, skoro można bezpośrednio wywołać funkcję `fibonacciAsync`?

Teraz można przenieść obciążenie procesora wymagającymi obliczeniami na odrębny serwer. Pozwala to zachować cykle procesora dla serwera frontonu, dzięki czemu fronton może obsługiwać żądania przeglądarek. Wymagające obliczenia można przetwarzać niezależnie. Można nawet zainstalować klaster serwerów zaplecza powiązany z mechanizmem równoważenia obciążenia, które równomiernie rozdziela żądania. W trakcie tworzenia systemów wielowarstwowych nieustannie podejmowane są takie decyzje.

To pokazuje, że za pomocą kilku wierszy kodu oraz platform Node.js i Express można tworzyć proste wielowarstwowe usługi w architekturze REST. Cały ten przykład daje możliwość zastanowienia się nad obsługą wymagającego obliczeniowo kodu w Node.js.

Wybrane moduły i platformy związane z architekturą REST

Oto kilka dostępnych pakietów i platform wspomagających tworzenie projektów opartych na architekturze REST:

- Restify (*http://restify.com/*) udostępnia platformy do tworzeniu obu stron transakcji w architekturze REST (klienta i serwera). Interfejs API używany po stronie serwera jest podobny do interfejsu API z platformy Express.
- Loopback (*https://strongloop.com/node-js/loopback-framework/*) to projekt firmy StrongLoop, obecnego sponsora platformy Express. Udostępnia wiele funkcji i oczywiście jest oparty na platformie Express.

Podsumowanie

Z tego rozdziału dowiedziałeś się dużo na temat obsługi protokołu HTTP i tworzenia aplikacji internetowych w Node.js.

Teraz możesz przejść do budowania kompletnej aplikacji do robienia notatek. Ta aplikacja, Notes, posłuży w kilku następnych rozdziałach do zapoznania się z platformą Express, dostępem do baz danych, instalowaniem usług w chmurze lub na własnym serwerze i do uwierzytelniania użytkowników.

W następnym rozdziale utworzysz podstawową infrastrukturę tej aplikacji.

Skorowidz

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ż

Platforma Node.js

Przewodnik webdevelopera

Wydanie III

Platforma Node.js służy do tworzenia aplikacji ogólnego przeznaczenia w języku JavaScript, które nie wymagają przeglądarki internetowej, a sam kod może działać zarówno po stronie klienta, jak i serwera. W Node.js wykorzystano szybki silnik JavaScriptu, V8. Platforma ta udostępnia stabilną bibliotekę do obsługiwania asynchronicznych sieciowych operacji wejścia-wyjścia. Dodatkowo programista ma do dyspozycji bogaty zestaw niezależnych modułów. Szczególnie atrakcyjne jest wykorzystanie Node.js do budowania aplikacji związanych z internetem rzeczy (IoT) i mikrousług.

W tej książce znajdziesz przegląd zagadnień potrzebnych do nauki programowania w Node.js. Dowiesz się między innymi, w jaki sposób można zaimplementować mechanizmy przechowywania danych w bazach danych, uwierzytelniania użytkownika czy komunikacji między użytkownikami w czasie rzeczywistym. Dodatkowo zaprezentowano tu technikę instalowania kodu na serwerze za pomocą platformy Docker. Nie zabraknie też opisu najlepszych praktyk z dziedziny tworzenia oprogramowania i rozkładania dużego obciążenia między serwery zaplecza. Ciekawym tematem ujętym w książce jest implementacja mikrousług REST w architekturze wielowarstwowej.

Node.js — platforma, która zapewnia niezawodność, prostotę i wydajność kodu!

W książce znajdziesz:

- informacje o tym, czym jest platforma Node.js, jak się rozwija i do czego może się przydać
- opis konfigurowania platformy i omówienie modułów, a także korzystanie z narzędzia npm
- tworzenie aplikacji na urządzenia mobilne, w tym wdrożenie modelu REST
- korzystanie z biblioteki Socket.IO
- testowanie aplikacji, w tym testy jednostkowe, testy REST i testy funkcjonalne

David Herron przez wiele lat był inżynierem oprogramowania w Dolinie Krzemowej. Pracował nad różnorodnymi projektami, włączając w to aplikacje do monitorowania wydajności systemów paneli słonecznych. Jako starszy inżynier w Sun Microsystems pracował w zespole Java SE Quality Engineering, Herron pracował też dla firmy VXtreme nad oprogramowaniem, które później stało się aplikacją Windows Media Player. Interesuje się pojazdami elektrycznymi, światowymi zasobami energii, zmianami klimatu i ochroną środowiska.

[PACKT] open source PUBLISHING community experience distilled

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

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

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-3611-7

cena: 59,00 zł