O'REILLY®

Wydanie II

Tworzenie aplikacji internetowych z użyciem Node i Express

Korzystanie ze stosu JavaScript

Helion

Ethan Brown

Tytuł oryginału: Web Development with Node and Express: Leveraging the JavaScript Stack, 2nd Edition

Tłumaczenie: Joanna Zatorska

ISBN: 978-83-283-6743-2

© 2020 Helion SA
Authorized Polish translation of the English edition of Web Development with Node and Express, 2nd Edition ISBN 9781492053514 © 2020 Ethan Brown

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

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

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

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

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
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/taine2.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/taine2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

ROZDZIAŁ 4.

Porządki

W dwóch poprzednich rozdziałach jedynie eksperymentowaliśmy, a właściwie zaledwie muskaliśmy powierzchnię oceanu. Zanim przejdziemy do bardziej złożonych zagadnień, musimy nieco uporządkować projekt i zadbać o dobre nawyki podczas pracy.

W tym rozdziale skoncentrujemy się na projekcie portalu Meadowlark Travel. Zanim zaczniemy tworzyć samą witrynę, upewnimy się, że mamy narzędzia potrzebne do utworzenia produktu wysokiej jakości.

Nie ma potrzeby odtwarzać działającego przykładu z tej książki. Jeśli ktoś chciałby utworzyć własną witrynę, może skorzystać z działającego przykładu i odpowiednio go zmodyfikować, aby po przeczytaniu tej książki móc się cieszyć gotowym produktem.

Struktura plików i katalogów

Definiowanie struktury aplikacji było przedmiotem wielu dyskusji, lecz żadna z nich nie doprowadziła do wyłonienia jednego poprawnego sposobu. Warto jednak poznać kilka popularnych konwencji.

Przeważnie ogranicza się liczbę plików w głównym katalogu projektu. Zwykle zawiera on pliki konfiguracyjne (takie jak *package.json*), plik *README.md* oraz kilka podkatalogów. Większość kodu źródłowego umieszcza się w katalogu *src*. Aby uprościć wykonywanie ćwiczeń z tej książki, nie będziemy przestrzegać tej konwencji (co ciekawe, nie przestrzega jej także szablonowa aplikacja Expressa). W rzeczywistych projektach szybko możemy zaśmiecić główny katalog projektu, jeśli będziemy umieszczać w nim kod źródłowy. Unikniemy tego problemu, jeśli przeniesiemy pliki z kodem do osobnego katalogu, na przykład o nazwie *src*.

Wspomniałem też, że wolę nazywać główny plik aplikacji (czasem zwany *punktem wejścia*) identycznie jak sam projekt (*meadowlark.js*) w przeciwieństwie do bardziej ogólnej nazwy, takiej jak *index.js*, *app.js* lub *server.js*.

Decyzje dotyczące struktury aplikacji należą do nas. Zalecam umieszczenie planu struktury projektu w pliku *README.md* (lub w dołączonym do niego innym pliku).

W katalogu projektu zawsze należy mieć pliki *package.json* i *README.md*. Reszta zależy od nas.

Kup książkę Poleć książkę

Najlepsze praktyki

Ostatnio dość często słyszy się określenie *najlepsze praktyki*, oznaczające poprawne wykonywanie zadań i niestosowanie skrótowych rozwiązań (nieco dalej napiszę, co to właściwie oznacza). Na pewno każdy słyszał inżynierskie porzekadło, że żadnego zadania nie można wykonać szybko, tanio i dobrze, lecz jedynie z zachowaniem maksymalnie dwóch z tych cech. Zawsze intrygowało mnie, że to podejście nie uwzględnia *przyrostowej wartości* poprawnego wykonywania zadania. Jeśli pewne zadanie wykonujemy poprawnie po raz pierwszy, prawdopodobnie poświęcimy na to pięć razy więcej czasu niż w przypadku pośpiesznego i niestarannego wykonania. Natomiast następnym razem czas będzie dłuższy tylko trzykrotnie. Gdy wykonamy zadanie poprawnie już kilkanaście razy, okaże się, że wykonujemy je równie szybko jak wówczas, gdy robilibyśmy je pośpiesznie i niestarannie.

Mój trener szermierki zawsze powtarza, że ćwiczenia nie prowadzą do perfekcji, tylko utrwalają nawyki. Oznacza to, że powtarzanie *utrwala* pewne zachowania. To prawda, lecz nie ma tu żadnego odniesienia do jakości praktykowanych czynności. Jeśli ćwiczymy złe nawyki, wówczas staną się one rutyną. Zamiast tego powinniśmy postępować zgodnie z zasadą, że praktykowanie *perfekcji* prowadzi do perfekcji. Zachęcam zatem, aby wykonywać pozostałe przykłady z tej książki jak w prawdziwym projekcie, gdy nasza reputacja i wynagrodzenie zależą od jakości pracy. Używaj tej książki nie tylko do nauki nowych umiejętności, ale także do wypracowywania dobrych nawyków.

Skoncentrujemy się teraz na kontroli wersji i testowaniu jakości. W tym rozdziale omawiam kontrolę wersji, a w następnym kontrolę jakości.

Kontrola wersji

Mam nadzieję, że nie muszę nikogo przekonywać do korzystania z kontroli wersji (w przeciwnym razie prawdopodobnie musiałbym na to poświęcić całą książkę). Ogólnie rzecz biorąc, kontrola wersji daje następujące korzyści:

Dokumentacja

Możliwość powrotu do historii projektu w celu sprawdzenia podejmowanych decyzji oraz kolejności rozwijania poszczególnych komponentów może stanowić cenną dokumentację. Techniczna historia projektu może się okazać całkiem przydatna.

Atrybucja

Jeśli ktoś pracuje w zespole, możliwość sprawdzenia autorstwa kodu bywa niesłychanie istotna. Gdy znajdziemy niejasne lub wątpliwej jakości fragmenty kodu, odszukanie autora może nam zaoszczędzić wielu godzin pracy. Może się okazać, że komentarze dotyczące modyfikacji kodu rozwieją nasze wątpliwości. A jeśli nie, to przynajmniej będziemy wiedzieć, z kim należy rozmawiać na ich temat.

Eksperymenty

Dobry system kontroli wersji umożliwia eksperymentowanie. Możemy swobodnie próbować nowych rozwiązań, bez obawy naruszenia stabilności swojego projektu. Jeśli eksperyment się powiedzie, możemy go uwzględnić w projekcie, a jeśli nie, możemy go porzucić.

Kup książkę Poleć książkę

Wiele lat temu zacząłem korzystać z rozproszonych systemów kontroli wersji (*Distributed Version Control System* — DVCS). Zawęziłem wybór do systemów Git i Mercurial i ostatecznie zdecydowałem się na Git ze względu na jego rozpowszechnienie i elastyczność. Obydwa rozwiązania są doskonałymi i darmowymi systemami kontroli wersji i zalecam użycie jednego z nich. W tej książce będę korzystać z Gita, lecz zachęcam również do wypróbowania Mercuriala (lub innego systemu kontroli wersji).

Jeśli ktoś nie zna systemu Git, polecam przeczytanie doskonałej książki *Kontrola wersji z systemem Git* (Helion) Jona Loeligera. Warto również przejrzeć materiały szkoleniowe dostępne w portalu GitHub (*https://try.github.io*).

Jak używać systemu Git wraz z tą książką?

Przede wszystkim trzeba się upewnić, że system Git jest zainstalowany. W tym celu należy wpisać w terminalu polecenie **`git --version`**. Jeśli nie zostanie wyświetlony numer wersji, musimy zainstalować Git zgodnie z instrukcjami w dokumentacji tego systemu (*https://git-scm.com*).

Przykłady z tej książki można wykonywać na dwa sposoby. Pierwszy polega na samodzielnym przepisywaniu kodu, a następnie wykonywaniu poleceń systemu Git. Drugi polega na sklonowaniu repozytorium z przykładami i przeglądaniu plików z każdego przykładu. Niektórzy uczą się szybciej, przepisując przykłady, a inni wolą jedynie przejrzeć i uruchomić program po zmianach, bez konieczności samodzielnego wpisywania kodu.

Samodzielne pisanie kodu

Dysponujemy już podstawowym zarysem projektu składającym się z widoków, układu, logo, głównego pliku aplikacji oraz pliku *package.json*. Utwórzmy teraz repozytorium Gita i dodajmy do niego te pliki.

Najpierw należy przejść do głównego katalogu projektu i zainicjalizować repozytorium:

```
git init
```

Zanim dodamy wszystkie pliki, utworzymy plik *.gitignore*, aby zapobiec przypadkowemu dodaniu plików, które chcemy pominąć. Utwórzmy plik tekstowy o nazwie *.gitignore* w katalogu projektu. W tym pliku należy uwzględnić wszystkie pliki lub katalogi, które Git ma domyślnie ignorować (po jednej nazwie w wierszu). Można też korzystać z symboli wieloznacznych. Na przykład jeśli edytor tworzy pliki zapasowe, których nazwa kończy się tyldą (np. *meadowlark.js~*), w pliku *.gitignore* możemy umieścić wiersz *~. Jeśli korzystamy z systemu macOS, powinniśmy uwzględnić plik *.DS_Store*. Nie można też zapomnieć o katalogu *node_modules* (ze względów, które omawiam nieco dalej). Początkowo plik może mieć następującą zawartość:

```
node_modules
*~
.DS_Store
```

Wpisy w pliku *.gitignore* mogą też uwzględniać podkatalogi. A zatem jeśli umieścimy wpis *~ w pliku *.gitignore* w głównym katalogu projektu, zignorowane zostaną wszystkie pliki zapasowe, nawet jeżeli znajdują się w podkatalogach.

Kup książkę Poleć książkę

Możemy już dodać do repozytorium wszystkie istniejące pliki. Można to zrobić na kilka sposobów. Ja zwykle używam polecenia `git add -A`, które umożliwia dodanie największej liczby elementów w porównaniu z innymi sposobami. Początkujący użytkownicy Gita mogą dodawać poszczególne pliki po kolei (np. poleceniem `git add meadowlark.js`), jeśli chcą dodać tylko jeden lub dwa pliki. Mogą też dodać wszystkie zmiany (włącznie z usuniętymi plikami) za pomocą polecenia `git add -A`. Ponieważ chcemy uwzględnić wszystkie dotychczasowe pliki, wykonajmy następujące polecenie:

```
git add -A
```

Początkujący użytkownicy Gita zwykle niezbyt rozumieją polecenie `git add`. Za jego pomocą dodajemy *zmiany*, a nie pliki. A zatem jeśli zmodyfikujemy plik *meadowlark.js*, a następnie wykonamy polecenie `git add meadowlark.js`, w rzeczywistości dodamy wprowadzone modyfikacje.

W systemie Git zdefiniowana jest „poczekalnia", w której przechowywane są zmiany po wykonaniu polecenia `git add`. A zatem dodane zmiany nie są jeszcze zatwierdzone. Aby zatwierdzić zmiany, należy wykonać polecenie `git commit`:

```
git commit -m "Zatwierdzenie początkowych zmian"
```

Parametr `-m "Zatwierdzenie początkowych zmian"` umożliwia zapisanie komunikatu dotyczącego wprowadzanych zmian. Git nie bez przyczyny nie pozwala wykonać tego polecenia bez dodania komunikatu. Zawsze należy dodawać sensowne komunikaty, które powinny krótko i zwięźle opisywać wykonane zadanie.

Korzystanie z przykładowego kodu

Przykładowy kod dla tej książki jest dostępny pod adresem *ftp://ftp.helion.pl/przyklady/troya.zip*. Oryginalny, aktualizowany kod przykładowy znajduje się w serwisie GitHub i można go pobrać następującym poleceniem:

```
git clone https://github.com/EthanRBrown/web-development-with-node-and-express-2e
```

Pakiet przykładowego kodu jest podzielony na foldery dotyczące poszczególnych rozdziałów zawierające pliki z przykładowym kodem. Kod źródłowy dla niniejszego rozdziału znajduje się w katalogu *ch04*. Pliki dla każdego rozdziału są ponumerowane, aby ułatwić odwoływanie się do nich w tekście. W pakiecie przykładowego kodu znajdują się pliki *README.md* zawierające dodatkowe informacje o przykładach.

W pierwszej wersji tej książki zastosowałem inne podejście do przykładowego kodu, który uporządkowałem zgodnie z wzrostem złożoności projektu. Chociaż tamto podejście odzwierciedlało postępy w pracy nad rzeczywistym projektem, to okazało się problematyczne zarówno dla mnie, jak i dla czytelników. Ze względu na zmiany w pakietach npm również przykładowy kod musiał się zmieniać. Niestety brakowało możliwości aktualizacji całego kodu oraz uwzględnienia zmian w tekście. Mimo że podejście skutkujące utworzeniem osobnych katalogów dla poszczególnych rozdziałów jest bardziej sztuczne, ułatwia synchronizację tekstu z kodem i dodawanie zmian przez społeczność.

Kup książkę Poleć książkę

W miarę aktualizacji i ulepszeń tej książki będę również aktualizował kod i oznaczał go nową wersją, dzięki czemu do każdego nowego wydania będzie można pobrać odpowiedni pakiet kodu. Przykładowy kod dla niniejszego wydania jest oznaczony wersją 2.0.0. W tym przypadku stosuję *wersjonowanie semantyczne* (więcej informacji na ten temat znajduje się w dalszej części tego rozdziału). Ostatni element w numerze wersji reprezentuje niewielkie zmiany, które nie powinny wpłynąć na możliwość pracy z niniejszym wydaniem książki. Oznacza to, że kod w wersji 2.0.15 powinien nadal odpowiadać temu wydaniu książki. Natomiast zmiana drugiego elementu numeru wersji (na 2.1.0) oznacza, że zawartość kodu odbiega od treści tego wydania i należy skorzystać z wersji rozpoczynającej się od 2.0.

W pakiecie przykładowego kodu znajdują się pliki *README.md* zawierające dodatkowe informacje o próbkach kodu.

Gdyby ktoś chciał poeksperymentować, powinien pamiętać, że jeśli pobierze z systemu Git określoną wersję kodu, repozytorium będzie w stanie oznaczonym jako „detached HEAD". Chociaż można wtedy edytować pliki, zatwierdzanie zmian nie jest bezpieczne, o ile nie utworzymy najpierw nowej gałęzi. Zatem jeśli chcesz utworzyć eksperymentalną gałąź na podstawie określonej wersji kodu, możesz w tym celu wykonać polecenie `git checkout -b eksperyment` (gdzie `eksperyment` to nazwa naszej gałęzi; można użyć dowolnej innej nazwy). Następnie można bezpiecznie edytować i zatwierdzać zmiany wprowadzone w tej gałęzi.

Pakiety npm

Pakiety npm, na których opiera się nasz projekt, znajdują się w katalogu *node_modules*. (Nazwa *node_modules* jest dość niefortunna i uważam, że lepsza byłaby *npm_packages*, ponieważ moduły Node są podobną, lecz inną koncepcją). Można swobodnie zaglądać do tego katalogu, aby zaspokoić ciekawość lub zdebugować program, ale nigdy nie należy modyfikować żadnego kodu znajdującego się w tym katalogu. Nie tylko jest to niewłaściwa praktyka, lecz wszystkie zmiany mogą zostać z łatwością cofnięte przez npm.

Jeśli chcemy zmodyfikować pakiet, od którego zależy nasz projekt, powinniśmy utworzyć własne odgałęzienie repozytorium tego pakietu. Jeśli podążymy tą drogą i uznamy, że nasze zmiany przydadzą się innym programistom, możemy sobie pogratulować dołączenia do projektu otwartego oprogramowania! Możemy przesłać zmiany, a jeśli spełnią one standardy projektu, zostaną uwzględnione w oficjalnym pakiecie. Dodawanie zmian do istniejących pakietów oraz tworzenie niestandardowych kompilacji wykracza poza zakres tej książki, lecz warto pamiętać o dynamicznej społeczności programistów gotowych do pomocy, jeśli zechcemy dołączyć swoje zmiany do istniejących pakietów.

Plik *package.json* opisuje nasz projekt i konfigurację zależności. Zachęcam, aby teraz do niego zajrzeć. Powinien zawierać kod podobny do następującego (numery wersji będą się prawdopodobnie różnić, ponieważ te pakiety są dość często aktualizowane):

```
{
  "dependencies": {
    "express": "^4.16.4",
    "express-handlebars": "^3.0.0"
  }
}
```


Kup książkę Poleć książkę

Na tym etapie plik *package.json* zawiera jedynie informacje o zależnościach. Znak daszka (^) na początku wersji pakietu oznacza, że potrzebna jest dowolna wersja rozpoczynająca się od podanego numeru — aż do następnego głównego numeru wersji. Na przykład zgodnie z powyższym plikiem *package.json* możemy zainstalować dowolną wersję platformy Express rozpoczynającą się od numeru 4.0.0. Zatem może to być wersja 4.0.1 i 4.9.9, natomiast wersja 3.4.7 lub 5.0.0 nie będą odpowiednie. Jest to domyślny sposób traktowania wersji podczas wykonywania polecenia `npm install`, który zwykle jest dość bezpieczny. W konsekwencji, jeśli będziemy chcieli skorzystać z nowszej wersji, będziemy musieli ją zmodyfikować w pliku konfiguracyjnym. Przeważnie jest to pożądane, ponieważ zapobiega występowaniu problemów w projekcie wynikających z niespodziewanych zmian w zależnościach. Numery wersji w npm są przetwarzane przez komponent o nazwie *semver* (od słów *semantic versioning*, czyli wersjonowanie semantyczne). Więcej informacji o wersjonowaniu w npm znajduje się w specyfikacji Semantic Versioning Specification (*http://try.github.io/*) i w artykule Tamasa Pirosa (*http://bit.ly/34Vr3lX*).

Semantic Versioning Specification informuje, że oprogramowanie stosujące wersjonowanie semantyczne musi deklarować „publiczny interfejs API”. To określenie zawsze wydawało mi się myląće — oznacza ono, że „ktoś musi dbać o interfejs umożliwiający używanie naszego oprogramowania”. Jeśli rozważymy szersze znaczenie tego określenia, okaże się, że właściwie może ono oznaczać wszystko. Nie warto się tym przejmować; najważniejszy bowiem jest format.

Ponieważ plik *package.json* zawiera listę wszystkich zależności, katalog *node_modules* jest właściwie artefaktem pochodnym. Oznacza to, że po jego usunięciu możemy wykonać polecenie `npm install`, które odtworzy usunięty katalog i zainstaluje w nim wszystkie potrzebne zależności. Dzięki temu nasz projekt będzie działał poprawnie. Z tego względu zalecam umieszczenie katalogu `node_modules` w pliku *.gitignore* i pomijanie go w kontroli wersji. Jednak niektórzy uważają, że repozytorium powinno zawierać cały kod potrzebny do uruchomienia projektu, i wolą uwzględniać katalog `node_modules` w systemie kontroli wersji. Według mnie jest to zaśmiecanie repozytorium i wolę go pomijać.

Począwszy od wersji 5 narzędzia npm, tworzony jest dodatkowy plik o nazwie *package-lock.json*. Podczas gdy format pliku *package.json* może być dość „nieprecyzyjny”, jeśli chodzi o wersje zależności (i korzystać z modyfikatorów wersji ^ i ~), plik *package-lock.json* rejestruje dokładne numery zainstalowanych wersji, co może ułatwić dokładne odtworzenie wersji zależności naszego projektu. Zalecam zapisywanie tego pliku w kontroli wersji i powstrzymanie się od ręcznych modyfikacji. Więcej informacji o pliku *package-lock.json* można znaleźć w dokumentacji (*http://bit.ly/2O8IjNK*).

Metadane projektu

Plik *package.json* zawiera również metadane projektu, takie jak nazwa, autorzy, informacje o licencji itd. Jeśli utworzymy plik *package.json* za pomocą polecenia `npm init`, odpowiednie pola zostaną uzupełnione automatycznie. W każdej chwili możemy je zmodyfikować. Jeśli zamierzamy opublikować projekt w repozytorium npm lub w portalu GitHub, te metadane są niesłychanie ważne. Więcej in-

Kup książkę Poleć książkę

formacji o polach, jakie można zdefiniować w pliku *package.json*, zawiera dokumentacja tego pliku (*http://bit.ly/2X7GVPs*). Ważnym komponentem metadanych jest też plik *README.md*. Można w nim opisać ogólną architekturę witryny WWW, a także uwzględnić krytyczne informacje, potrzebne nowym uczestnikom projektu. Plik ten zawiera tekst w formacie wiki zwanym Markdown. Więcej informacji o składni Markdown można znaleźć w dokumentacji (*http://bit.ly/2q7BQur*).

Moduły Node

Jak już wspomniałem, moduły Node i pakiety npm są ze sobą powiązane, lecz są różnymi koncepcjami. *Moduły Node*, jak sama nazwa wskazuje, oferują mechanizm modularyzacji i enkapsulacji. *Pakiety npm* stanowią standardowy schemat przechowywania, wersjonowania i odwoływania się do projektów (które nie są ograniczone do modułów). Na przykład w głównym pliku aplikacji importujemy Express jako moduł:

```
const express = require('express')
```

`require` jest funkcją Node służącą do importowania modułów. Domyślnie Node szuka modułów w katalogu *node_modules* (nie powinno nas już dziwić, że w katalogu tym znajduje się podkatalog *express*). Jednak Node oferuje mechanizm tworzenia własnych modułów (nie należy tworzyć własnych modułów w katalogu *node_modules*). Oprócz modułów zainstalowanych w katalogu *node_modules* za pośrednictwem menedżera pakietów Node oferuje ponad 30 „podstawowych modułów", takich jak `fs`, `http`, `os` i `path`. Cała lista tych modułów jest dostępna w odpowiedzi na pytanie (*http://bit.ly/2NDIkKH*) zadane w portalu Stack Overflow, a także w oficjalnej dokumentacji Node (*https://nodejs.org/en/docs/*).

Sprawdźmy, jak można zmodularyzować losowanie ciasteczek szczęścia zaimplementowane w poprzednim rozdziale.

Najpierw utwórzmy katalog do przechowywania modułów. Możemy mu nadać dowolną nazwę, lecz zazwyczaj stosuje się określenie *lib* (skrót od słowa *library*, czyli biblioteka). Utwórzmy w tym katalogu plik *fortune.js* (*ch04/lib/fortune.js* w pakiecie dołączonego kodu):

```
const fortuneCookies = [
  "Pokonaj swoje lęki albo one pokonają ciebie.",
  "Rzeki potrzebują źródeł.",
  "Nie obawiaj się nieznanego.",
  "Oczekuj przyjemnej niespodzianki.",
  "Zawsze szukaj prostego rozwiązania.",
]

exports.getFortune = () => {
  const idx = Math.floor(Math.random()*fortuneCookies.length)
  return fortuneCookies[idx]
}
```

Należy zwrócić uwagę na zmienną globalną `exports`. Jeśli chcemy, aby jakiś element był dostępny poza modułem, musimy dodać go do obiektu `exports`. W tym przykładzie poza modułem będzie dostępna funkcja `getFortune`, lecz tablica `fortuneCookies` będzie *całkowicie ukryta*. To dobrze, ponieważ enkapsulacja umożliwia tworzenie kodu mniej podatnego na błędy.

Kup książkę Poleć książkę

Funkcje można eksportować z modułu na kilka sposobów, które omawiam w dalszej części książki, a ich podsumowanie znajduje się w rozdziale 22.

Możemy teraz usunąć tablicę `fortuneCookies` z pliku *meadowlark.js* (chociaż nic się nie stanie, jeśli ją tam zostawimy; nie wystąpi żaden konflikt wynikający z użycia takiej samej nazwy w pliku *lib/fortune.js*). Tradycyjnie (choć nie jest to konieczne) importowane moduły podaje się na początku pliku, a zatem dodajmy następujący wiersz w górnej części pliku *meadowlark.js* (*ch04/meadowlark.js* w pakiecie dołączonego kodu):

```
const fortune = require('./lib/fortune')
```

Zauważ, że poprzedziliśmy nazwę modułu znakami `./`. Oznacza to, że Node nie powinien szukać tego modułu w katalogu *node_modules*; gdybyśmy pominęli ten przedrostek, kod by nie zadziałał.

Następnie możemy skorzystać z metody `getFortune` modułu w trasie dotyczącej strony *O nas*:

```
app.get('/about', (req, res) => {
  res.render('about', { fortune: fortune.getFortune() } )
})
```

Jeśli ktoś wykonuje samodzielnie przykłady z tej książki, powinien teraz zatwierdzić zmiany:

```
git add -A git commit -m "Przeniesiono 'ciasteczko szczęścia' do modułu."
```

Każdy z pewnością uzna, że moduły są potężnym i łatwym narzędziem hermetyzacji funkcjonalności, ulepszającym strukturę projektu oraz ułatwiającym jego utrzymanie i testowanie. Więcej informacji na ten temat znajdziesz w oficjalnej dokumentacji modułów Node (*http://nodejs.org/api/modules.html*).

Moduły Node czasem określa się mianem *modułów CommonJS* (*CJS*) ze względu na starszą specyfikację, którą zainspirowali się twórcy Node. Język JavaScript korzysta z oficjalnego mechanizmu tworzenia pakietów, zwanego modułami ECMAScript (*ECMAScript Modules* — ESM). Jeśli ktoś pisał kod w JavaScripcie dla platformy React lub innego postępowego języka frontendowego, prawdopodobnie zna koncepcję ESM, w której wykorzystuje się składnię import i export (zamiast `exports`, `module.exports` i `require`). Więcej informacji na ten temat znajduje się we wpisie na blogu doktora Axela Rauschmayera, zatytułowanym „ECMAScript 6 modules: the final syntax” (*http://bit.ly/2X8ZSkM*).

Podsumowanie

Uzbrojeni w nowe informacje o systemie Git, narzędziu npm i modułach, możemy przejść do tworzenia lepszych produktów dzięki stosowaniu dobrych praktyk zapewniania jakości (*Quality Assurance* — QA) kodu.

Zachęcam, aby po przeczytaniu tego rozdziału zapamiętać następujące stwierdzenia:

- Dzięki kontroli wersji proces tworzenia oprogramowania jest bezpieczniejszy i bardziej przewidywalny. Zalecam go nawet w małych projektach, ponieważ w ten sposób wypracujemy dobre nawyki!

Kup książkę Poleć książkę

- Modularyzacja jest ważną techniką zarządzania złożonością oprogramowania. Nie tylko oferuje bogaty ekosystem modułów opracowanych przez innych programistów i udostępnionych w repozytorium npm, ale także umożliwia tworzenie własnych modułów ułatwiających organizację kodu w projekcie.
- Moduły Node (zwane też CJS) wykorzystują inną składnię niż moduły ECMAScript (ESM). Możliwe, że będziemy musieli przełączać się między nimi podczas pracy nad kodem frontendowym i backendowym. Warto zapoznać się z obydwoma rodzajami modułów.

Kup książkę Poleć książkę

Kup książkę Poleć książkę


PROGRAM
PARTNERSKI
GRUPY 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
Helion

JavaScript: oczekuj tego, co najlepsze!

Express i Node stały się kluczowymi narzędziami do tworzenia dynamicznych, wielostronicowych i hybrydowych aplikacji internetowych. Dzięki nim można dowolnie kształtować architekturę swojej aplikacji. Projektowanie za pomocą Node jest bardzo atrakcyjnym rozwiązaniem dla programistów, którzy doceniają dostępność i elastyczność języka JavaScript. W rzeczy samej, JavaScript ewoluował z techniki prostego ozdabiania stron internetowych, aby stać się dojrzałym, wszechstronnym, pełnoprawnym i wyjątkowo obiecującym językiem programowania. Korzystanie z niego jest o wiele bardziej satysfakcjonujące przy zastosowaniu platformy Express.

Oto praktyczny przewodnik dla programistów, którzy chcą tworzyć aplikacje internetowe z wykorzystaniem platform React, Angular lub Vue oraz API typu REST albo ich kombinacji za pomocą języka JavaScript, Node i Express. Zawarto tutaj wprowadzenie do Node, Express oraz innych przydatnych narzędzi. Szczegółowo przedstawiono koncepcję oprogramowania pośredniczącego oraz zagadnienia bezpieczeństwa środowiska produkcyjnego. Nie zabrakło wskazówek dotyczących tworzenia API za pomocą Express. Ciekawym elementem książki są szczegóły integracji z takimi usługami jak Twitter, Google Maps i US National Weather Service. Poszczególne rozdziały przedstawiają etapy budowy przykładowej w pełni funkcjonalnej aplikacji internetowej, którą łatwo można wykorzystać jako szablon do tworzenia innych, własnych, bardziej wyrafinowanych projektów!

W książce między innymi:

- renderowanie danych dynamicznych za pomocą systemu szablonów
- używanie obiektów żądań i odpowiedzi, oprogramowania pośredniczącego oraz trasowania
- testowanie, debugowanie i wdrażanie aplikacji
- korzystanie z baz danych, takich jak MongoDB i PostgreSQL
- integracja aplikacji z innymi serwisami
- plan publikowania i utrzymywania aplikacji

Ethan Brown od ponad dwudziestu lat zajmuje się inżynierią oprogramowania. Obecnie jest dyrektorem do spraw technologii w firmie VMS, gdzie zajmuje się implementacją oprogramowania w chmurze. Szczególnie interesują go rozwiązania ułatwiające podejmowanie decyzji, analizę ryzyka i kreatywne projektowanie. Uważa stos technologiczny JavaScript za niezwykle obiecującą platformę aplikacji internetowych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-6743-2

9 788328 367432

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 67,00 zł