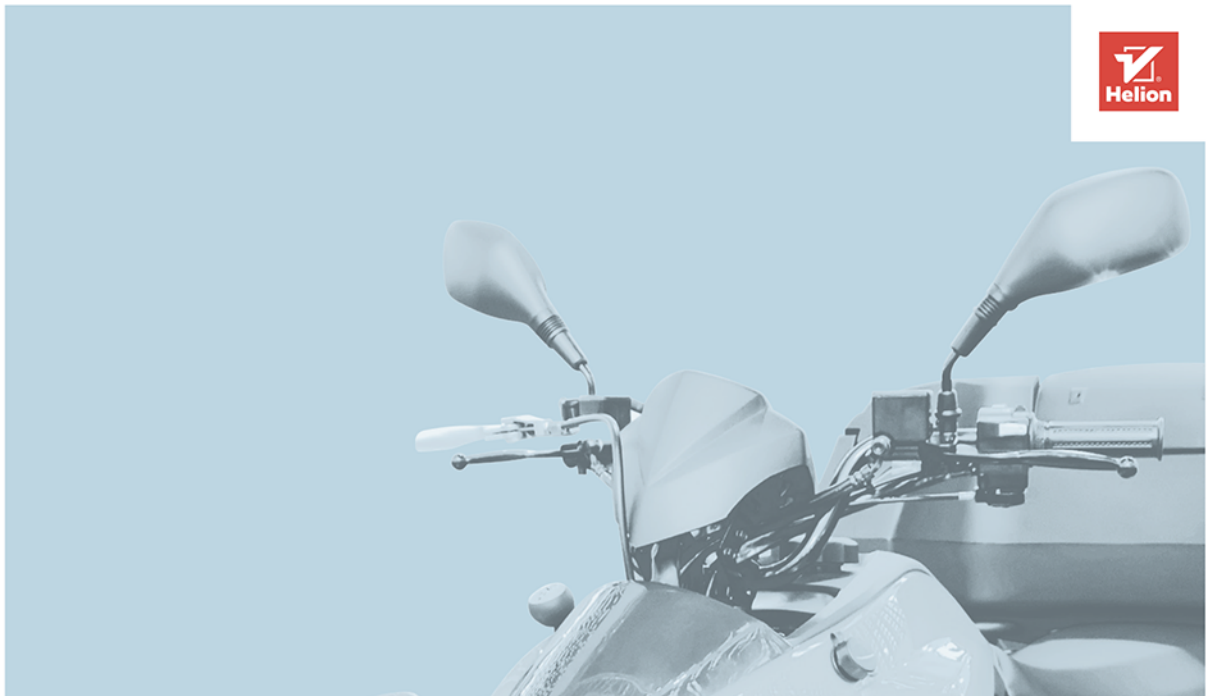
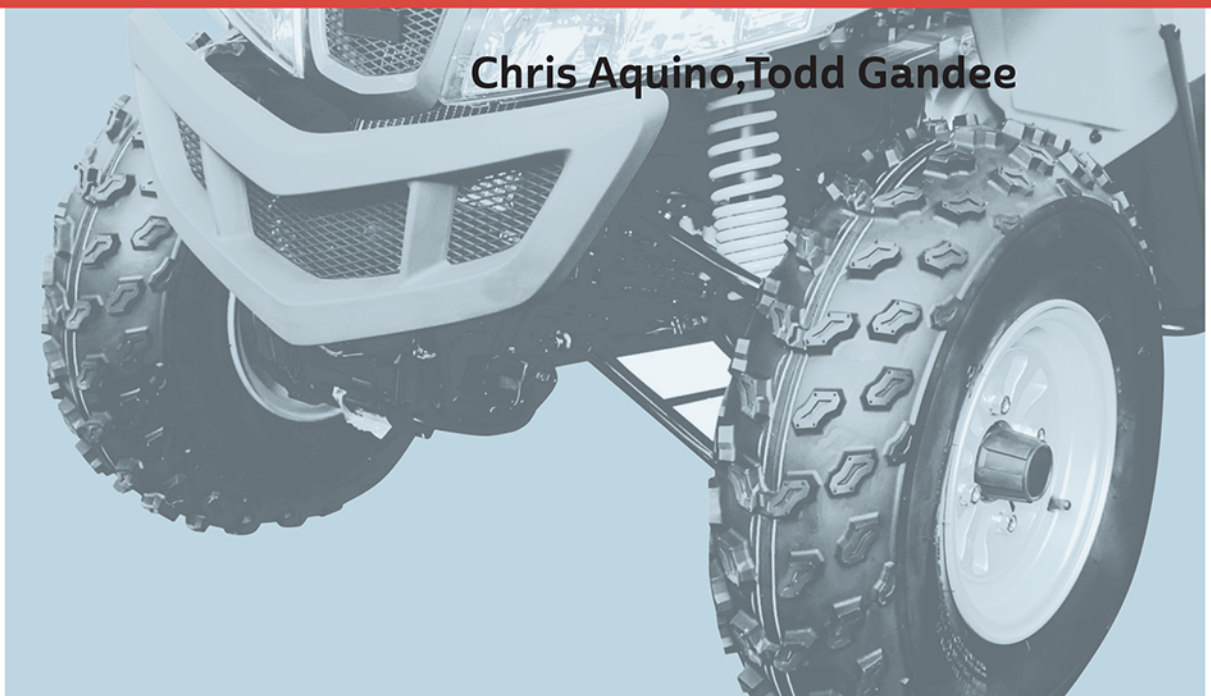


A close-up, light blue-tinted photograph of the handlebars and mirrors of an ATV.

Podręcznik frontendowca

BIG NERD RANCH GUIDE

A close-up, light blue-tinted photograph of the front end of an ATV, showing the headlights, grille, and large knobby tires.

Chris Aquino, Todd Gandee

Tytuł oryginału: Front-End Web Development: The Big Nerd Ranch Guide

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-3203-4

Authorized translation from the English language edition, entitled: FRONT-END WEB DEVELOPMENT: THE BIG NERD RANCH GUIDE, First Edition, ISBN 0134433947; by Chris Aquino; and by Todd Gandee; published by Pearson Education, Inc, publishing as The Big Nerd Ranch Guides.

Copyright © 2016 Big Nerd Ranch, LLC

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 Pearson Education Inc.

Polish language edition published by HELION S.A. Copyright © 2017.

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

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

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

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

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

Printed in Poland.

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

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

Spis treści

Wprowadzenie	13
Nauka tworzenia stron WWW	13
Wymagania	14
Układ książki	14
Jak korzystać z tej książki	15
Wyzwania	15
Dla ambitnych	16

Część I Podstawy tworzenia stron WWW

1 Przygotowanie środowiska programistycznego	19
Instalacja przeglądarki Google Chrome	19
Instalacja i konfiguracja edytora Atom	19
Wtyczki edytora Atom	21
Dokumentacja i źródła informacji	24
Błyskawiczny kurs obsługi wiersza polecenia	26
Sprawdzanie bieżącego katalogu	27
Tworzenie katalogu	28
Zmiana katalogu	29
Wyświetlanie listy plików w katalogu	31
Uzyskiwanie uprawnień administratora	32
Wyjście z programu	33
Instalacja Node.js i browser-sync	33
Dla ambitnych: zamienniki edytora Atom	35
2 Utworzenie pierwszego projektu	37
Utworzenie projektu Wydrowisko	38
Pierwszy kod HTML	39
Dołączanie arkusza stylów	42
Wpisywanie treści strony	43
Umieszczanie obrazów	44
Wyświetlanie strony w przeglądarce	46
Narzędzia dla programistów w przeglądarce Chrome	48
Dla ambitnych: wersje języka CSS	50
Dla ambitnych: ikona favicon.ico	51
Wyzwanie srebrne: utworzenie pliku favicon.ico	52

3	Style	53
	Zastosowanie stylów bazowych	54
	Przygotowanie kodu HTML do zastosowania stylów	56
	Budowa stylu	57
	Twoja pierwsza reguła	58
	Model pudełkowy	60
	Dziedziczenie stylów	62
	Dopasowanie obrazów do wielkości okna	69
	Kolory	71
	Dopasowanie odstępów między elementami listy	74
	Selektory relacyjne	75
	Definiowanie czcionek	79
	Wyzwanie brązowe: zmiana kolorów	81
	Dla ambitnych: precyzja kolidujących selektorów	82
4	Responsywne strony WWW i model flexbox	85
	Rozbudowa interfejsu użytkownika	86
	Umieszczenie dużego obrazu	87
	Poziomy układ miniatur	89
	Model flexbox	91
	Tworzenie elastycznego kontenera	92
	Zmiana kierunku rozmieszczenia elementów	93
	Grupowanie elementów w elastyczne jednostki	95
	Właściwość flex	96
	Kolejność, wyrównanie i rozmieszczenie elastycznych jednostek	98
	Wyśrodkowanie dużego obrazu	102
	Położenie względne i bezwzględne elementu	105
5	Układy adaptacyjne i zapytania medialne	111
	Resetowanie obszaru roboczego	113
	Stosowanie zapytań medialnych	115
	Wyzwanie brązowe: pionowa orientacja ekranu	118
	Dla ambitnych: model flexbox — popularne układy i błędy	118
	Wyzwanie złote: układ Holy Grail	119
6	Obsługa zdarzeń za pomocą JavaScriptu	121
	Przygotowanie znaczników <a>	122
	Twój pierwszy skrypt	126
	Opis kodu JavaScript w projekcie Wydrowisko	127
	Deklarowanie zmiennych tekstowych	128
	Praca z konsolą	129
	Odwołania do elementów modelu DOM	131
	Utworzenie funkcji zmieńObraz	135
	Deklarowanie parametrów funkcji	137
	Funkcje zwracające wartości	140
	Tworzenie obserwatora zdarzeń	142
	Obsługa wszystkich miniatur	147

Przetwarzanie tablicy miniatur	149
Wyzwanie srebrne: przechwytywanie odnośników	151
Wyzwanie złote: losowe zdjęcia wydr	151
Dla ambitnych: tryb ścisły	151
Dla ambitnych: domknięcia	152
Dla ambitnych: listy obiektów i kolekcje elementów HTML	152
Dla ambitnych: typy danych w JavaScriptcie	153
7 Efekty wizualne i style CSS	155
Ukrywanie i pokazywanie dużego obrazu	156
Utworzenie stylów ukrywających duży obraz	157
Utworzenie kodu JavaScript ukrywającego duży obraz	159
Obserwowanie zdarzeń związanych z naciśnięciami klawiszy	161
Ponowne wyświetlenie dużego obrazu	163
Zmiany stanów elementów i efekty przejścia	165
Właściwość transform	166
Zdefiniowanie efektu przejścia	168
Funkcje czasu	171
Wywołanie efektu przejścia poprzez zmianę klasy	172
Wywołanie efektu przejścia za pomocą kodu JavaScript	173
Własne funkcje czasu	175
Dla ambitnych: zasady koercji typów	177

Część II Moduły, obiekty i formularze

8 Moduły, obiekty i metody	181
Moduły	182
Szablon modułu	183
Modyfikacja obiektu za pomocą wyrażenia IIFE	185
Utworzenie projektu Kafejka	186
Utworzenie modułu BazaDanych	187
Dodawanie modułów do przestrzeni nazw	188
Konstruktory	190
Właściwość prototype konstruktora	191
Tworzenie metod konstruktora	193
Utworzenie modułu Furgonetka	195
Składanie zamówień	196
Usuwanie zamówień	198
Diagnostyka kodu	200
Wyszukiwanie błędów za pomocą debugera	201
Przypisanie właściwości this wartości za pomocą metody bind	206
Inicjowanie aplikacji Kafejka po załadowaniu strony	206
Utworzenie instancji obiektu typu Furgonetka	208
Dla ambitnych: prywatne dane modułu	210
Wyzwanie srebrne: dane prywatne	211
Dla ambitnych: przypisywanie wartości właściwości this w funkcji zwrotnej w metodzie forEach	211

9	Wprowadzenie do platformy Bootstrap	213
	Dołączenie platformy Bootstrap	214
	Jak działa platforma Bootstrap?	215
	Utworzenie formularza zamówień	216
	Dodanie pól tekstowych	217
	Podejmowanie decyzji za pomocą pól wyboru	221
	Dodanie rozwijanej listy	222
	Dodanie suwaka	224
	Dodanie przycisków Wyślij i Resetuj	225
10	Przetwarzanie danych w formularzu za pomocą JavaScriptu	227
	Utworzenie modułu ObsługaFormularza	228
	Wprowadzenie do biblioteki jQuery	229
	Zaimportowanie biblioteki jQuery	230
	Konfigurowanie obiektu typu ObsługaFormularza za pomocą selektora	230
	Dodanie obsługi zdarzenia submit	232
	Wyodrębnienie danych	233
	Przekazanie i wywołanie funkcji zwrótej	235
	Użycie obiektu typu ObsługaFormularza	237
	Rejestracja metody złóżZamówienie do obsługi zdarzenia submit	238
	Udoskonalenia interfejsu użytkownika	239
	Wyzwanie brązowe: zestaw powiększony	241
	Wyzwanie srebrne: wyświetlanie wartości ustawianej za pomocą suwaka	241
	Wyzwanie złote: gratulacje	241
11	Od danych do modelu DOM	243
	Utworzenie listy zamówień	244
	Utworzenie modułu ListaZamówień	245
	Utworzenie konstruktora Wiersz	246
	Tworzenie elementów modelu DOM za pomocą biblioteki jQuery	247
	Tworzenie wierszy listy zamówień po wystąpieniu zdarzenia submit	252
	Modyfikowanie obiektu this za pomocą metody call	253
	Realizacja zamówienia po kliknięciu pozycji na liście	255
	Utworzenie metody ListaZamówień.prototype.usuńWiersz	255
	Usuwanie nadpisanych wierszy	256
	Utworzenie metody dodajObsługęKliknięcia	257
	Wywołanie metody dodajObsługęKliknięcia	259
	Wyzwanie brązowe: umieszczenie mocy kawy w opisie zamówienia	260
	Wyzwanie srebrne: kolorowanie smaków kawy	260
	Wyzwanie złote: edycja zamówień	260
12	Weryfikacja formularzy	261
	Atrybut required	261
	Weryfikacja formularza za pomocą wyrażeń regularnych	263
	Interfejs Constraint Validation API	263
	Obserwacja zdarzenia input	265
	Powiązanie metody weryfikującej dane ze zdarzeniem input	266
	Uruchamianie mechanizmu weryfikacyjnego	267

Definiowanie stylów dla elementów zawierających poprawne i błędne dane	269
Wyzwanie brązowe: weryfikacja zamówienia na kawę bezkofeinową	270
Dla ambitnych: biblioteka Webshim	270
13 Ajax	273
Klasa XMLHttpRequest	274
Usługa REST	275
Moduł ZdalnaBazaDanych	275
Wysyłanie danych do serwera	276
Zastosowanie metody \$.post z biblioteki jQuery	277
Utworzenie funkcji zwrotnej	278
Badanie zapytań i odpowiedzi Ajax	279
Odbieranie danych z serwera	282
Badanie odpowiedzi z serwera	282
Utworzenie funkcji zwrotnej	283
Usuwanie danych z serwera	285
Zastosowanie metody \$.ajax z biblioteki jQuery	285
Zamiana modułu BazaDanych na ZdalnaBazaDanych	286
Wyzwanie srebrne: weryfikacja danych z odwołaniem do serwera	289
Dla ambitnych: narzędzie Postman	289
14 Obiekty Deferred i Promise	291
Obiekty Promise i Deferred	292
Zwracanie obiektów Deferred	293
Rejestrowanie funkcji zwrotnych za pomocą metody then	295
Obsługa błędów za pomocą metody then	296
Zastosowanie obiektów Deferred z obiektami wykorzystującymi tylko funkcje zwrotne	298
Zastosowanie obiektów Promise w module BazaDanych	303
Tworzenie i zwracanie obiektów Promise	304
Akceptacja obiektu Promise	305
Zastosowanie obiektów Promise w pozostałych metodach modułu BazaDanych	305
Wyzwanie srebrne: automatyczne przełączanie na moduł BazaDanych	307

Część III Przetwarzanie danych w czasie rzeczywistym

15 Wprowadzenie do platformy Node.js	311
Platforma Node.js i program npm	313
Polecenie npm init	313
Polecenie npm scripts	313
Program „Witaj, świecie!”	315
Utworzenie skryptu dla programu npm	316
Udostępnianie plików	318
Odczytywanie pliku za pomocą modułu fs	319
Przetwarzanie adresu URL zapytania	319
Zastosowanie modułu path	320
Utworzenie własnego modułu	322
Zastosowanie własnego modułu	322

Obsługa błędów	323
Dla ambitnych: rejestr modułów programu npm	325
Wyzwanie brązowe: tworzenie własnej strony z komunikatem o błędzie	325
Dla ambitnych: typy danych MIME	325
Wyzwanie srebrne: dynamiczne wysyłanie informacji o typie MIME	327
Wyzwanie złote: przeniesienie kodu obsługującego błędy do osobnego modułu	327
16 Komunikacja w czasie rzeczywistym z wykorzystaniem protokołu WebSocket	329
Konfiguracja protokołu WebSocket	331
Test serwera WebSocket	333
Utworzenie funkcjonalności pogawędki	333
Pierwsza pogawędka!	335
Dla ambitnych: biblioteka socket.io	336
Dla ambitnych: usługa WebSocket	337
Wyzwanie brązowe: czy ja się powtarzam?	337
Wyzwanie srebrne: hasło dostępu	337
Wyzwanie złote: automatyczny klient	337
17 Wersja JavaScript ES6 i translator Babel	339
Narzędzia do translacji kodu JavaScript	340
Część kliencka aplikacji Czatownik	342
Pierwsze kroki z translatorem Babel	343
Słowo kluczowe class	343
Pakowanie modułów za pomocą narzędzia Browserify	345
Uruchomienie procesu translacji	347
Utworzenie klasy CzatKomunikat	348
Utworzenie modułu ws-klient	352
Obsługa połączeń	353
Obsługa zdarzeń i wysyłanie komunikatów	355
Wysyłanie i wyświetlanie komunikatów	357
Dla ambitnych: konwersja na JavaScript kodu utworzonego w innych językach	358
Wyzwanie brązowe: domyślna nazwa importowanej zmiennej	358
Wyzwanie srebrne: komunikat o zamkniętym połączeniu	359
Dla ambitnych: podnoszenie deklaracji	359
Dla ambitnych: funkcja strzałkowa	361
18 Ciąg dalszy przygody z wersją ES6	363
Instalacja biblioteki jQuery jako modułu platformy Node.js	364
Utworzenie klasy CzatFormularz	364
Połączenie klasy CzatFormularz z obiektem gniazdo	367
Utworzenie klasy CzatLista	368
Gravatar	370
Pytanie o nazwę użytkownika	372
Przechowywanie danych sesji użytkownika	374
Formatowanie i aktualizowanie znaczników czasu	377
Wyzwanie brązowe: dodanie efektów wizualnych do komunikatów	379
Wyzwanie złote: osobne pokoje do pogawędek	379

Część IV Architektura aplikacji

19	Wprowadzenie do architektury MVC i platformy Ember	383
	Aplikacja Tropiciel	384
	Ember: platforma architektury MVC	385
	Instalacja platformy Ember	386
	Utworzenie aplikacji opartej na platformie Ember	388
	Uruchomienie serwera	389
	Zewnętrzne biblioteki i dodatki	390
	Konfiguracja środowiska	391
	Dla ambitnych: instalacja bibliotek za pomocą programów npm i Bower	395
	Wyzwanie brązowe: ograniczanie importu bibliotek	395
	Wyzwanie srebrne: dodanie czcionki Awesome	395
	Wyzwanie złote: dostosowanie paska nawigacyjnego	395
20	Sterownik, ścieżki i modele danych	397
	Polecenie ember generate	398
	Zagnieżdżanie ścieżek	402
	Dodatek Ember Inspector	404
	Przypisanie modeli danych	404
	Metoda beforeModel	407
	Dla ambitnych: metody setupController i afterModel	407
21	Modele danych i wiązanie rekordów	409
	Definicje modeli	409
	Metoda createRecord	412
	Metody get i set	413
	Właściwości wyliczeniowe	415
	Dla ambitnych: odczytywanie danych	417
	Dla ambitnych: modyfikowanie i usuwanie danych	418
	Wyzwanie brązowe: zmiana właściwości wyliczeniowej	419
	Wyzwanie srebrne: oznaczanie nowych obserwacji	419
	Wyzwanie złote: dodawanie tytułów	419
22	Adaptory, serializatory i transformaty	421
	Adaptory	423
	Zasady ochrony treści	427
	Serializatory	427
	Transformaty	429
	Dla ambitnych: dodatek Mirage	430
	Wyzwanie srebrne: bezpieczeństwo treści	430
	Wyzwanie złote: dodatek Mirage	431
23	Widoki i szablony	433
	Język Handlebars	434
	Modele	434
	Elementy pomocnicze	434
	Warunkowe elementy pomocnicze	435
	Pętla {{#each}}	437

Wiązanie właściwości modelu danych	439
Odnosiniki	441
Niestandardowe elementy pomocnicze	444
Wyzwanie brązowe: dodatkowe informacje do odnośników	446
Wyzwanie srebrne: zmiana formatu danych	446
Wyzwanie złote: niestandardowy element ikony	447
24 Kontrolery	449
Nowe obserwacje	450
Edycja danych obserwacji	457
Usuwanie danych o obserwacji	460
Akcje ścieżek	462
Wyzwanie brązowe: strona ze szczegółami obserwacji	464
Wyzwanie srebrne: data obserwacji	464
Wyzwanie złote: dodawanie i usuwanie danych świadków	464
25 Komponenty	465
Zawartość pętli jako komponent	466
Komponenty w kodzie DRY	469
Dane w dół, akcje w górę	470
Wiązanie nazw klas	471
Dane w dół	473
Akcje w górę	475
Wyzwanie brązowe: dostosowanie komunikatu	478
Wyzwanie srebrne: przekształcenie paska nawigacyjnego w komponent	478
Wyzwanie złote: tablica komunikatów	478
Posłowie	479
Ostatnie wyzwanie	479
Dziękujemy	479
Skorowidz	481

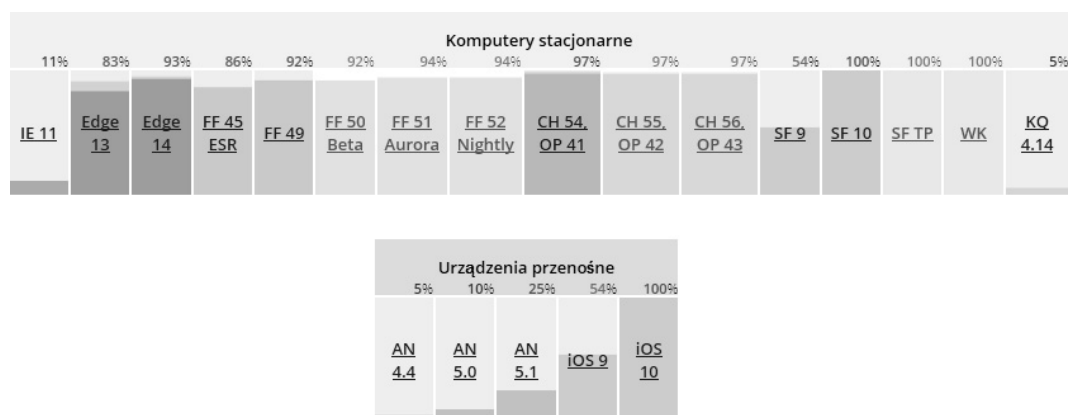
Wersja JavaScript ES6 i translator Babel

Język JavaScript powstał w 1994 roku. W 1999 roku został kilkakrotnie zaktualizowany i do 2009 roku nie uległ zmianom. W 2009 roku wprowadzono w nim kilka niewielkich zmian, w wyniku których pojawiła się wersja ES5, czyli piąta edycja standardu języka JavaScript.

W 2015 roku, w ramach szóstej edycji, opracowano kilka ulepszeń. Wiele nowych możliwości funkcjonalnych było wzorowanych na innych językach, na przykład Ruby i Python. Szósta edycja otrzymała techniczną nazwę ES2015, ale powszechnie jest znana jako wersja ES6.

Wersja ES6 jest dobrze obsługiwana przez przeglądarki Google Chrome, Mozilla Firefox i Microsoft Edge. Są to przeglądarki „zawsze żywe”, tzn. aktualizujące się samoczynnie, bez konieczności ręcznego pobierania i instalowania kolejnych wersji przez użytkownika. Twórcy tych przeglądarek coraz ściślej dostosowują swoje produkty do wersji ES6 i szybko udostępniają je użytkownikom.

Jednakże przeglądarki inne niż wyżej wymienione, jak również przeglądarki dla urządzeń przenośnych nie obsługują już tak dobrze wersji ES6. Rysunek 17.1 przedstawia odsetek możliwości funkcjonalnych wersji ES6 obsługiwanych przez najnowsze wersje przeglądarek dla komputerów stacjonarnych i urządzeń przenośnych. (Skróty na rysunku oznaczają, odpowiednio: IE = Internet Explorer, FF = Mozilla Firefox, CH = Google Chrome, SF = Safari, WK = Webkit, KQ = Konqueror i AN = Android).



Rysunek 17.1. Odsetek możliwości funkcjonalnych wersji ES6 obsługiwanych przez przeglądarki (dane z wiosny 2016 roku)

Jeżeli chcesz uzyskać dokładniejsze lub bardziej aktualne informacje o obsłudze wersji ES6, odwiedź stronę kangax.github.io/compat-table/es6. Jej autor, Juriy Zaytsev, regularnie aktualizuje dane.

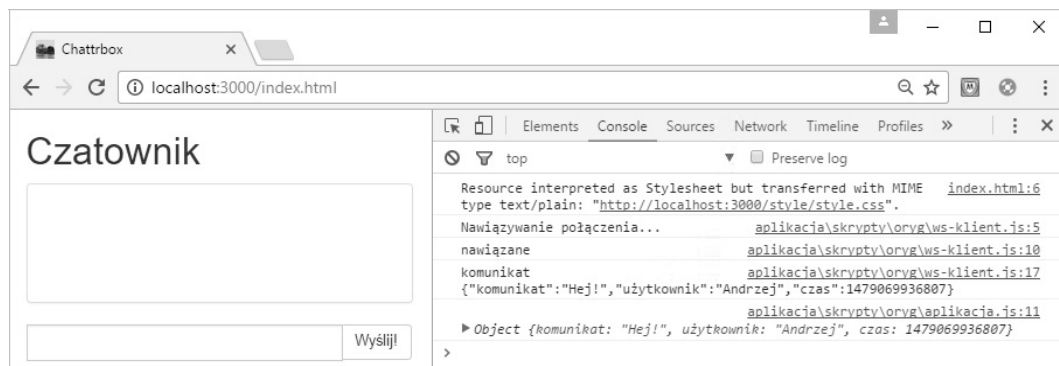
Choć inne przeglądarki mogą gorzej obsługiwać wersję ES6, jednak my ją bardzo, ale to bardzo lubimy. Jest to doskonały język, który warto zacząć stosować jak najszybciej, nie czekając, aż będzie powszechnie obsługiwany.

W tym rozdziale zajmiesz się kliencką częścią aplikacji *Czatownik*, w której wykorzystasz kilka nowych możliwości wersji ES6. Aby aplikacja działała we wszystkich przeglądarkach, zastosujesz otwarte narzędzie Babel zapewniające kompatybilność kodu.

Zanim zaczniesz, musisz wykonać pewną operację porządkową. Abyś mógł skupić się na poznaniu wersji ES6 i narzędzia Babel, przygotowaliśmy gotowe pliki *index.html* i *style/style.css*. Pobierz plik <ftp://ftp.helion.pl/przyklady/natERO.zip> rozpakuj go (włącznie z folderem *style*) do swojego folderu *czatownik\aplikacja* (istniejący plik *index.html* zostanie zastąpiony nowym).

I jeszcze uwaga: gdy będziesz pracował nad kodem w tym rozdziale, w konsoli przeglądarki może się pojawiać komunikat o niekreślonym typie MIME pliku CSS. Możesz go bezpiecznie zignorować.

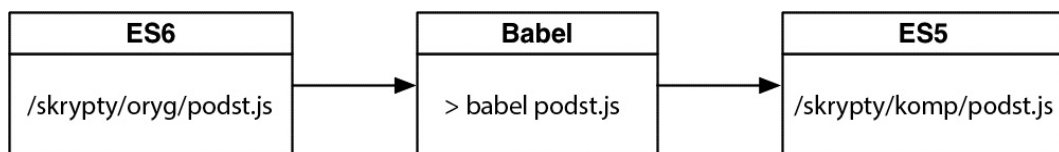
Do biegu, gotowi, start! Na końcu tego rozdziału aplikacja *Czatownik* będzie komunikowała się za pomocą protokołu WebSocket z serwerem (patrz rysunek 17.2).



Rysunek 17.2. Wygląd aplikacji Czatownik na końcu tego rozdziału

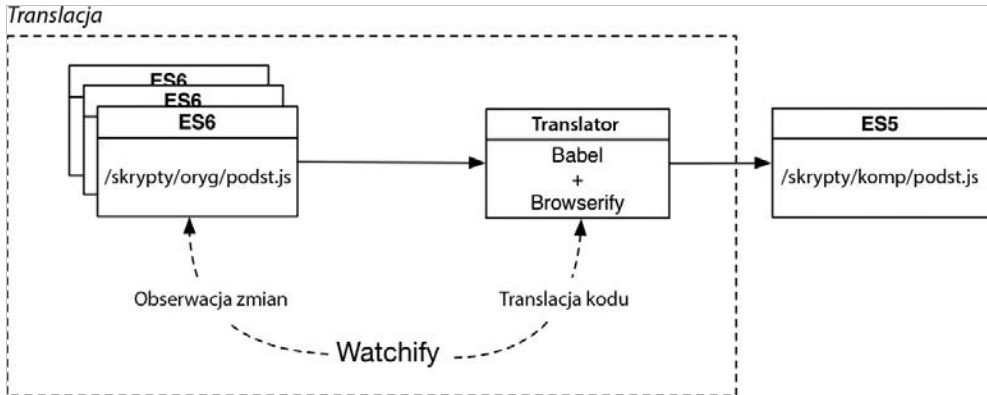
Narzędzia do translacji kodu JavaScript

Narzędzie Babel jest translatorem. Jego przeznaczeniem jest przekładanie kodu wykorzystującego wersję ES6 na równoważny mu kod w wersji ES5, wykonywany w przeglądarce (patrz rysunek 17.3).



Rysunek 17.3. Tworzenie kodu w wersji ES5 na podstawie kodu z wersji ES6

Aby móc korzystać z narzędzia Babel, musisz zainstalować za pomocą polecenia `npm` kilka modułów umożliwiających zdefiniowanie automatycznego procesu translacji kodu. Narzędzie Babel wykorzystasz do przekładania kodu z wersji ES6 na wersję ES5, narzędzie Browserify — do grupowania modułów w jeden plik, a Babelify — do wykonywania obu tych operacji. Dodatkowo użyjesz narzędzia Watchify do uruchamiania procesu translacji po zapisaniu zmian wprowadzonych w kodzie (patrz rysunek 17.4).



Rysunek 17.4. Proces translacji kodu

Najpierw zainstaluj translator Babel. Narzędzie to składa się z kilku części, instalowanych w zależności od potrzeb. W naszym wypadku będzie wymagana translacja kodu na dwa sposoby: w wierszu poleceń i programie. Do wykonywania tych operacji będą potrzebne narzędzia, odpowiednio, `babel-cli` i `babel-core`. Będziesz również musiał zainstalować narzędzie `babel-preset-es2015` umożliwiające translację kodu utworzonego w wersji ES6.

Przejdź do folderu *czatownik* i wpisz poniższe polecenia instalujące odpowiednie narzędzia translatora Babel. (Jeżeli nie pamiętasz, jak uruchamiać polecenie `npm install -g` z uprawnieniami administratora, zajrzyj do rozdziału 1.).

```
npm install -g babel-cli
npm install --save-dev babel-core
npm install --save-dev babel-preset-es2015
```

Teraz musisz skonfigurować narzędzie Babel, aby do translacji kodu wykorzystywało zainstalowane właśnie narzędzie `babel-preset-es2015`. W folderze *czatownik* utwórz plik o nazwie `.babelrc` i wpisz w nim następujące instrukcje:

```
{
  "presets": [
    "es2015"
  ],
  "plugins": []
}
```

Na koniec w folderze *czatownik\node_modules* zainstaluj narzędzia Babelify, Browserify i Watchify za pomocą następującego polecenia:

```
npm install --save-dev browserify babelify watchify
```

Z powyższych narzędzi będziesz korzystał w dalszej części rozdziału, po uruchomieniu translatora Babel.

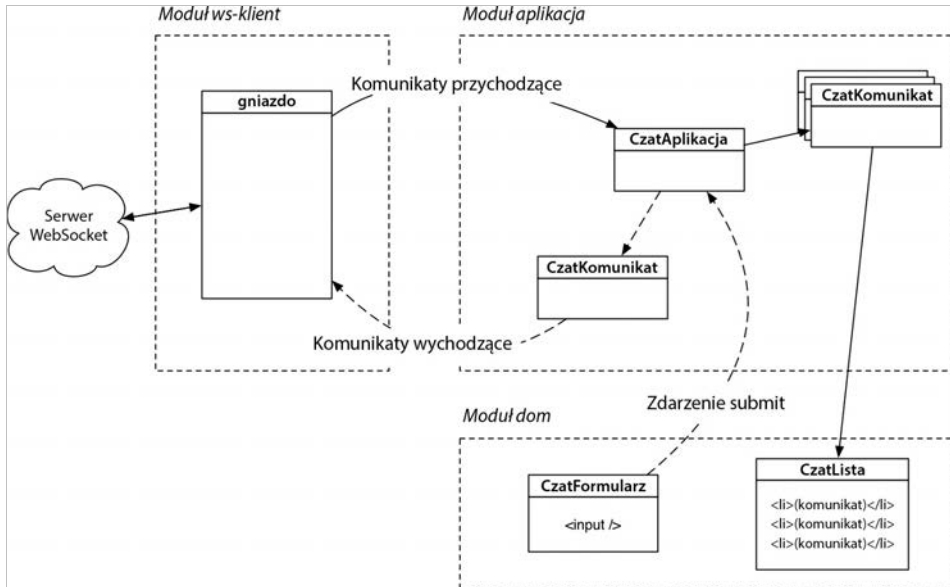
Część kliencka aplikacji Czatownik

Utworzyłeś już część serwerową aplikacji *Czatownik*, wysyłając statyczne pliki i obsługując wymianę komunikatów za pomocą protokołu WebSocket. Z użyciem tego protokołu część kliencka będzie wysyłała komunikaty i odbierała je z serwera. W niej również zostanie zdefiniowany format komunikatów. Użytkownik będzie widział listę komunikatów, jak również będzie mógł tworzyć nowe komunikaty za pomocą formularza.

Powyższe operacje będą realizowane przy użyciu trzech modułów:

- modułu *ws-klient* do zarządzania komunikacją z wykorzystaniem protokołu WebSocket po stronie klienta,
- modułu *dom* do wyświetlania danych w interfejsie użytkownika i wysyłania formularza,
- modułu *aplikacja* definiującego strukturę komunikatów i przekazującego je do modułów *ws-klient* i *dom*.

Rysunek 17.5 przedstawia zależności między powyższymi modułami.



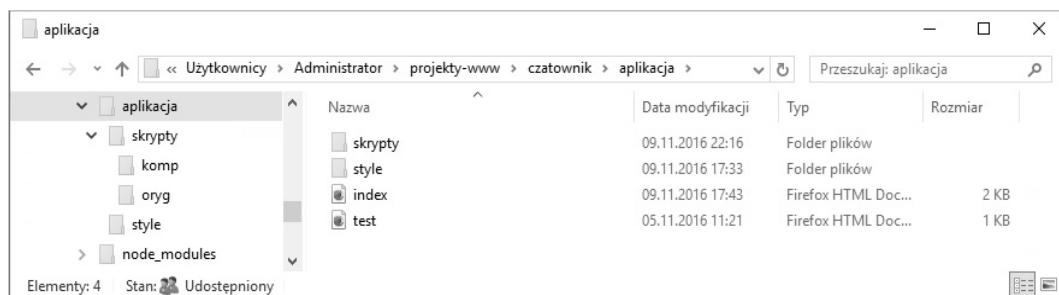
Rysunek 17.5. Moduły aplikacji Czatownik

W folderze *czatownik\aplikacja* utwórz podfoldery *skrypty*, *skrypty\oryg* i *skrypty\komp*, jak na rysunku 17.6.

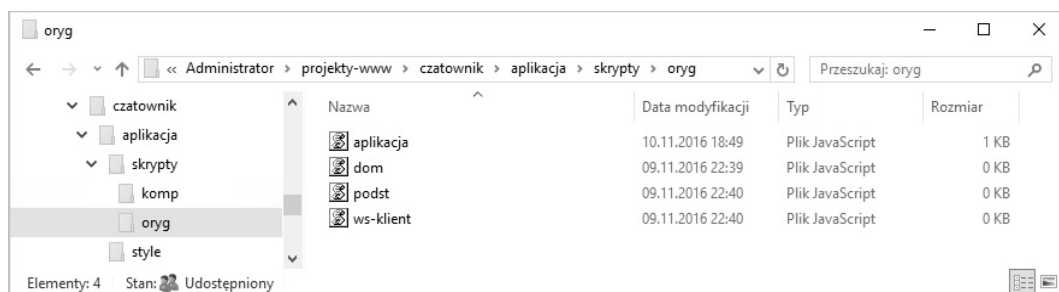
Teraz w folderze *skrypty\oryg* utwórz cztery pliki, które będą zawierały kod JavaScript:

- *aplikacja.js*,
- *dom.js*,
- *podst.js*,
- *ws-klient.js*.

Struktura plików powinna być taka jak na rysunku 17.7.



Rysunek 17.6. Struktura folderu czatownik\aplikacja



Rysunek 17.7. Zawartość folderu czatownik\aplikacja

Pliki *aplikacja.js*, *dom.js* i *ws-klient.js* będą zawierały moduły pokazane na rysunku 17.5, natomiast plik *podst.js* będzie zawierał kod inicjujący aplikację.

Pierwsze kroki z tłumaczem Babel

Teraz, po zainstalowaniu narzędzi i utworzeniu plików, czas na rozpoczęcie kodowania w języku JavaScript ES6.

Na razie z tłumacza Babel będziesz korzystał w wierszu poleceń. Później dodasz go do poleceń programu npm, aby translacja była wykonywana automatycznie. Dzięki temu będziesz mógł się skupić na nowych możliwościach funkcjonalnych i składni wersji ES6 i nie będziesz musiał dodatkowo wpisywać poleceń w terminalu.

Słowo kluczowe class

Pierwszą funkcjonalnością wersji ES6, którą wykorzystasz w części klienckiej aplikacji *Czatownik*, będzie słowo kluczowe `class`. Należy pamiętać, że słowo to nie służy do definiowania klas, jak w innych językach, a jedynie do tworzenia konstruktora i metod prototypu za pomocą prostszej składni.

Otwórz plik *aplikacja.js* i zdefiniuj w nim nową klasę o nazwie `CzatAplikacja`:

```
class Czataplikacja {
}
```

W tym rozdziale klasa `CzatAplikacja` niewiele będzie robić, jednakże w swojej ostatecznej formie będzie realizowała prawie cały algorytm aplikacji.

W tej chwili definicja wspomnianej klasy jest pusta. Dodaj do niej konstruktor zawierający metodę `console.log`:

```
class CzatAplikacja {  
  constructor() {  
    console.log('Oto wersja ES6!');  
  }  
}
```

Słowo `constructor` oznacza metodę wywoływaną podczas tworzenia instancji klasy. Zazwyczaj konstruktor jest wykorzystywany do ustalania wartości właściwości instancji.

Teraz, zaraz za definicją klasy `CzatAplikacja`, wpisz kod tworzący jej instancję:

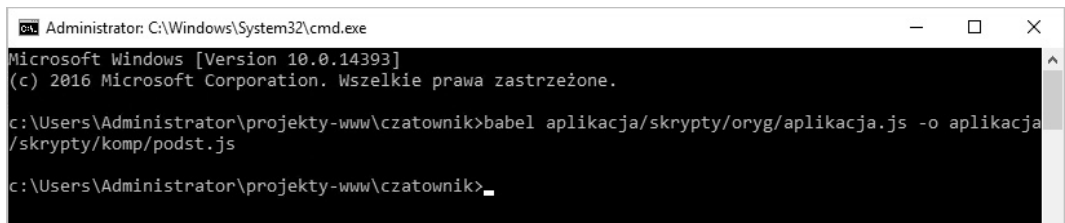
```
class CzatAplikacja {  
  constructor() {  
    console.log('Oto wersja ES6!');  
  }  
}  
  
new CzatAplikacja();
```

Uruchom próbnie kod. Otwórz drugie okno terminala i przejdź do folderu `czatownik`, w którym znajdują się pliki `package.json`, `index.js` i podfolder `aplikacja`. W tym oknie będziesz wpisywał polecenia przekładające kod. W pierwszym oknie będzie działał serwer.

Aby przetestować kod, przełóż plik `aplikacja\skrypty\oryg\aplikacja.js` za pomocą narzędzia Babel, a wynikowy kod zapisz w pliku `aplikacja\skrypty\komp\podst.js`:

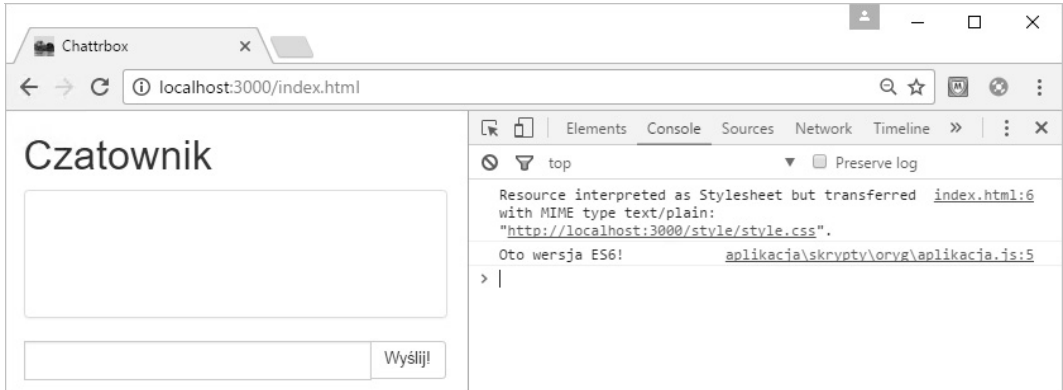
```
babel aplikacja/skrypty/oryg/aplikacja.js -o aplikacja/skrypty/komp/podst.js
```

Jeżeli w terminalu nic się nie pojawi, będzie to dobry znak i normalne zjawisko. Jeżeli w kodzie nie ma błędów, wówczas translator Babel nie wyświetla żadnych informacji (patrz rysunek 17.8).



Rysunek 17.8. Translator Babel działa niezauważalnie

Sprawdź, czy w drugim oknie działa serwer Node (uruchamiany poleceniem `npm run dev`) i otwórz w przeglądarce stronę o adresie `http://localhost:3000`. Zobaczysz teraz efekt wykonania kodu (patrz rysunek 17.9).



Rysunek 17.9. Oto wersja ES6!

W pliku `aplikacja\index.html` znajduje się odnośnik do pliku `podst.js` utworzonego na bazie pliku `aplikacja.js`. Ponieważ w pliku `aplikacja.js` jest tworzona nowa instancja klasy `CzatApl` i `kacja`, więc wywoływany jest jej konstruktor, wyświetlający w konsoli napis `Oto wersja ES6!`.

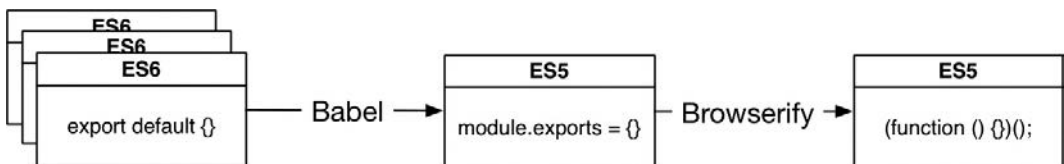
Przekonałeś się zatem, że narzędzie Babel prawidłowo przekłada pojedynczy plik JavaScript, czas więc utworzyć kilka modułów.

Pakowanie modułów za pomocą narzędzia Browserify

Jedną z rzeczy, której brakuje w wersji JavaScript ES5, jest wbudowany system modułów. Podczas tworzenia aplikacji *Kafejka* wykorzystałeś zastępczy sposób tworzenia modularnego kodu, wymagający jednak użycia zmiennej globalnej.

W wersji ES6 dostępne są prawdziwe moduły, takie jak w innych językach. Narzędzie Babel obsługuje składnię wersji ES6, ale nie jest w stanie przełożyć jej na równoważną jej składnię w wersji ES5. Dlatego w takiej sytuacji konieczne jest użycie narzędzia Browserify.

Rysunek 17.10 przedstawia współpracę narzędzi Browserify i Babel.



Rysunek 17.10. Konwersja modułów utworzonych w wersji ES6 na wersję ES5 za pomocą narzędzi Babel i Browserify

Domyślnie narzędzie Babel konwertuje kod modułu utworzonego w wersji ES6 na równoważny mu kod dostosowany do platformy Node.js, wykorzystujący instrukcje `require` i `module.exports`. Kod ten jest następnie konwertowany przez narzędzie Browserify na funkcje zgodne z wersją ES5.

Otwórz plik *package.json* i wpisz w nim opcje konfiguracyjne narzędzia Browserify:

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js",
  "dev": "nodemon index.js"
},
"browserify": {
  "transform": [
    ["babelify", {"presets": ["es2015"], "sourceMap": true}]
  ]
},
...
```

Powyższy kod powoduje, że narzędzie Browserify będzie wywoływało narzędzie Babelify z dwoma parametrami. Pierwszy z nich, es2015, określa wersję kodu, a drugi — opcję sourceMap (mapa źródłowa) ułatwiającą jego diagnostykę. Podczas tworzenia pozostałych części aplikacji *Czatownik* dowiesz się, jak diagnozować kod za pomocą map źródłowych.

Podobnie jak w wypadku programu nodemon, musisz dla narzędzia Browserify zdefiniować polecenia do wykonywania podstawowych operacji. Wpisz je w pliku *package.json*, w sekcji *scripts*. (Pamiętaj o umieszczeniu przecinka na końcu wiersza z poleceniem *dev*).

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js",
  "dev": "nodemon index.js",
  "build": "browserify -d aplikacja/skrypty/oryg/podst.js -o aplikacja/skrypty/komp/podst.js",
  "watch": "watchify -v -d aplikacja/skrypty/oryg/podst.js -o aplikacja/skrypty/komp/podst.js"
},
"browserify": {
  "transform": [
    ["babelify", {"presets": ["es2015"], "sourceMap": true}]
  ]
},
...
```

W pierwszym poleceniu, *build*, narzędzie Browserify jest wykorzystywane bezpośrednio, natomiast w drugim, *watch*, wykorzystywane jest narzędzie Watchify wywołujące narzędzie Browserify, gdy w kodzie zostaną wprowadzone zmiany (na czym również skorzysta program nodemon).

W kodzie modułu utworzonego w wersji ES6 należy jawnie eksportować klasy, które będą wykorzystywane w innych modułach. W pliku *aplikacja.js* zamiast instrukcji *new*, która jedynie tworzy instancję klasy *CzatAplikacja*, wpisz instrukcję *export*:

```
class CzatAplikacja {
  constructor() {
    console.log('Oto wersja ES6!');
```

```

    }
  }
}
new CzatApplikacja();
export default CzatApplikacja;

```

W ten sposób określisz, że `CzatApplikacja` jest domyślną klasą modułu. W innych modułach będziesz eksportował wiele klas. Jeżeli trzeba wyeksportować tylko jedną klasę, najlepiej jest użyć instrukcji `export default`.

W pliku *podst.js* wpisz instrukcje importujące klasę `CzatApplikacja` i tworzące jej instancję:

```

import CzatApplikacja from './aplikacja';
new CzatApplikacja();

```

W pliku *podst.js* importowana jest klasa `CzatApplikacja` wyeksportowana z pliku *aplikacja.js*. Po zaimportowaniu tworzona jest instancja tej klasy.

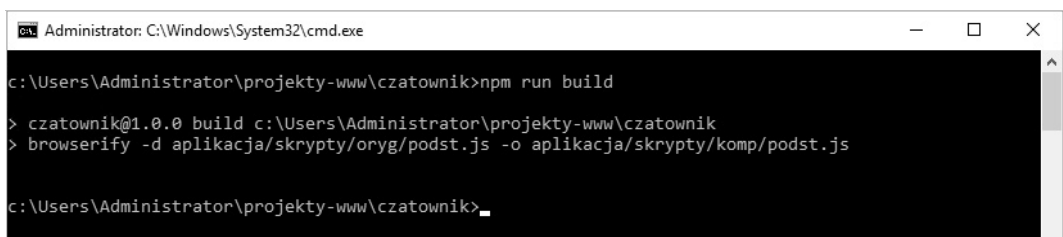
Należy w tym momencie zwrócić uwagę na ważną rzecz: nazwa klasy użyta w pliku *podst.js* może być dowolna. Ponieważ klasa `CzatApplikacja` jest domyślną klasą eksportowaną z pliku *aplikacja.js*, więc można użyć na przykład instrukcji `import MojaApplikacja from './aplikacja'`, która przypisze eksportowaną klasę do lokalnego identyfikatora `MojaApplikacja`. Jednak zgodnie z przyjętą dobrą praktyką należy zastosować nazwę `CzatApplikacja`, ponieważ taka nazwa została użyta w pliku *aplikacja.js*.

Uruchomienie procesu translacji

Teraz w terminalu wpisz następujące polecenie:

```
npm run build
```

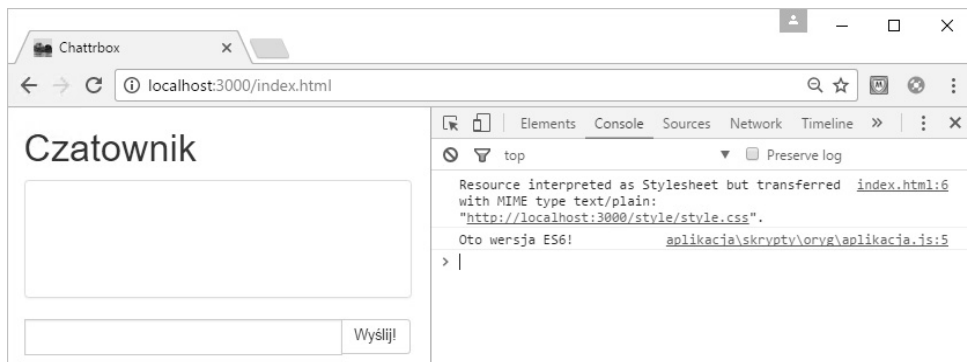
Program `npm` wykona polecenie `build` wywołujące narzędzie `Browserify`. W terminalu będą pojawiały się informacje o wykonywanych operacjach. Jednak samo narzędzie `Browserify` nie wyświetli żadnych informacji, chyba że w kodzie będzie błąd (patrz rysunek 17.11).



Rysunek 17.11. Użycie narzędzia `Browserify` za pomocą polecenia `npm run build`

Jeżeli proces translacji przebiegnie pomyślnie, narzędzie `Browserify` utworzy w folderze *aplikacja/komp* plik *podst.js*, który wcześniej utworzyłeś, ręcznie uruchamiając translację.

Teraz odśwież stronę w przeglądarce i sprawdź efekt. Nie wprowadziłeś w kodzie żadnych nowych możliwości funkcjonalnych. Zmieniłeś jedynie miejsce, w którym wywoływany jest konstruktor klasy `CzatApplikacja`. W konsoli pojawi się zatem ten sam komunikat co poprzednio (patrz rysunek 17.12).



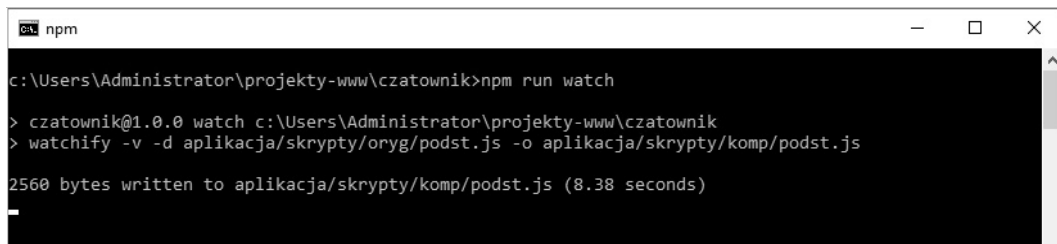
Rysunek 17.12. Oto znów wersja ES6!

Kolejnym zintegrowanym narzędziem jest Watchify. O ile program nodemon automatycznie restartuje serwer platformy Node.js po każdorazowej zmianie plików źródłowych, o tyle narzędzie Watchify automatycznie konwertuje zmienione pliki za pomocą narzędzia Browserify.

Uruchom narzędzie Watchify, aby za każdym razem po wprowadzeniu zmian w kodzie był uruchamiany proces translacji:

```
npm run watch
```

W terminalu pojawi się informacja potwierdzająca, że narzędzie Watchify zostało uruchomione (patrz rysunek 17.13).



Rysunek 17.13. Użycie narzędzia Watchify za pomocą polecenia npm run watch

Narzędzie Watchify wyświetla nieco więcej informacji niż Browserify. Za każdym razem, gdy zostaną wprowadzone zmiany w kodzie, wyświetla ono komunikat o liczbie zapisanych bajtów. Nie jest to nadzwyczaj interesująca informacja, ale przynajmniej wiadomo, że zmienił się plik wynikowy.

Narzędzie Watchify niech działa w jednym z okien terminala (w drugim będzie działał serwer), a Ty teraz zajmij się z powrotem kodem aplikacji *Czatownik*.

Utworzenie klasy CzatKomunikat

Przesyłanie komunikatów pomiędzy oknami terminala jest fajne, ale czas ulepszyć aplikację, aby można było przysłać je pomiędzy przeglądarkami. Zdefiniujesz teraz klasę pomocniczą służącą do tworzenia i formatowania danych przesyłanych w komunikatach.

W każdym komunikacie musi być zawarty pewien zestaw informacji, czyli treść, informacja o nadawcy i czas wysłania.

Do przesyłania danych często wykorzystywany jest format JSON (ang. *JavaScript Object Notation*; notacja obiektów w JavaScriptcie). Zastosowałeś go już w pliku *package.json*. Jest to czytelny dla człowieka i wspólny dla różnych języków programowania format, idealnie nadający się do przesyłania informacji w aplikacji *Czatownik*.

Poniżej przedstawiony jest przykładowy komunikat zapisany w formacie JSON:

```
{
  "komunikat": "Cześć!",
  "uzytkownik": "Andrzej",
  "czas": 1479067465066
}
```

Komunikaty w aplikacji *Czatownik* będą pochodziły z dwóch źródeł. Jednym z nich będzie użytkownik wysyłający komunikat po wypełnieniu formularza, a drugim — serwer rozsyłający komunikat do innych użytkowników za pomocą protokołu WebSocket.

Po wypełnieniu formularza przez użytkownika należy uzupełnić komunikat przed wysłaniem go do serwera o nazwę użytkownika i bieżący czas. Wszystkie te informacje muszą być również zawarte w komunikatach wysyłanych przez serwer. W jaki sposób można to osiągnąć? Jest kilka możliwości. Przeanalizujmy ogólnie niektóre z nich, zapoznając się przy okazji z paroma ciekawymi możliwościami wersji ES6.

W pliku *aplikacja.js* utwórz klasę, która będzie reprezentowała komunikaty:

```
class CzatAplikacja {
  constructor() {
    console.log('Oto wersja ES6!');
  }
}

class CzatKomunikat {
  constructor(dane) {
  }
}

export default CzatAplikacja;
```

Pierwszym sposobem jest utworzenie prostego konstruktora z parametrami zawierającymi treść komunikatu, nazwę użytkownika i czas (nie wprowadzaj tych zmian w pliku, to tylko przykład):

```
...
class CzatKomunikat {
  constructor(komunikat, uzytkownik, czas) {
    this.komunikat = komunikat;
    this.uzytkownik = uzytkownik || 'Andrzej';
    this.czas = czas || (new Date()).getTime();
  }
}
...
```

Z kodem w takiej postaci już się kilkakrotnie spotkałeś. Wartości parametrów są przypisywane właściwościom obiektu, a za pomocą operatora `||` określane są ich domyślne wartości.

Jest to poprawny kod, ale w wersji ES6 można wykorzystać *domyślne argumenty*, dzięki którym kod jest bardziej zwarty:

```
...
class CzatKomunikat {
  constructor(komunikat, użytkownik = 'Andrzej', czas = (new Date()).getTime();) {
    this.komunikat = komunikat;
    this.użytkownik = użytkownik;
    this.czas = czas;
  }
}
...
```

W powyższym zapisie jest oczywiste, które parametry są obowiązkowe, a które opcjonalne. W tym wypadku wymagany jest tylko parametr komunikat, pozostałym nadawane są domyślne wartości.

Konstruktor w takiej postaci może obsługiwać komunikaty odbierane z serwera lub tworzone za pomocą formularza. Jednak w wywołującym go kodzie argumenty muszą być podawane w ściśle określonej kolejności, co w wypadku funkcji i metod z trzema parametrami lub ich większą liczbą może być kłopotliwe.

Innym sposobem jest utworzenie konstruktora z jednym parametrem, zawierającym obiekt, w którym za pomocą par klucz/wartość określone są treść komunikatu, nazwa użytkownika i czas. Jest to tzw. **przypisanie destrukuryzujące**.

```
...
class CzatKomunikat {
  constructor({komunikat: k, użytkownik: u, czas: c}) {
    this.komunikat = k;
    this.użytkownik = u;
    this.czas = c;
  }
}
...
```

Powyższy kod może się wydawać dziwny, ale działa w następujący sposób:

Najpierw wywoływany jest konstruktor:

```
new CzatKomunikat ({komunikat: 'Cześć!', użytkownik='andrzej25@gmail.com',
czas=1462399523859});
```

Następnie w argumencie wyszukiwany jest klucz komunikat. Odpowiadająca mu wartość 'Cześć!' jest przypisywana do lokalnej zmiennej k, która z kolei jest wykorzystywana w kodzie konstruktora. Podobnie przetwarzane są właściwości użytkownik i czas.

Jednak powyższa składnia nie daje komfortu stosowania parametrów domyślnych. Na szczęście można połączyć obie opisane techniki. Ostateczna postać konstruktora, którą wpiszesz w pliku *aplikacja.js*, jest taka jak niżej:

```
...
class CzatKomunikat {
  constructor(dane) {
    komunikat: k,
```

```

        uzytkownik: u='Andrzej',
        czas: c=(new Date()).getTime()
    }) {
        this.komunikat = k;
        this.uzytkownik = u;
        this.czas = c;
    }
}
...

```

W kodzie w takiej postaci poszczególne wartości są odczytywane z obiektu podanego w argumencie konstruktora. Właściwości, które nie zostaną określone, przyjmą wartości domyślne.

Domyślne wartości parametrów mogą być określone w definicji funkcji (lub konstruktora), natomiast składnię destrukuryzującą można stosować do przypisywania zmiennym wartości. Powyższy konstruktor można również zdefiniować w następujący sposób:

```

...
class CzatKomunikat {
    constructor(dane) {
        var {komunikat: k, uzytkownik: u='Andrzej', czas: c=(new Date()).getTime()} = dane;
        this.komunikat = k;
        this.uzytkownik = u;
        this.czas = c;
    }
}
...

```

OK, koniec wycieczki, wróćmy do kodowania aplikacji *Czatownik*.

Instancje klasy CzatKomunikat będą przechowywały w swoich właściwościach wszystkie ważne informacje, jednak będą również zawierały metody i inne dane. Dlatego takie obiekty nie będą odpowiednie do przesyłania danych za pomocą protokołu WebSocket. Trzeba ograniczyć ilość zawartych w nich informacji.

W pliku *aplikacja.js* użyj instrukcji `serializuj`, za pomocą której właściwości obiektu typu CzatKomunikat zostaną zamienione na zwykły obiekt JavaScript:

```

...
class CzatKomunikat {
    constructor({
        komunikat: k,
        uzytkownik: u='Andrzej',
        czas: c=(new Date()).getTime()
    }) {
        this.komunikat = k;
        this.uzytkownik = u;
        this.czas = c;
    }
    serializuj() {
        return {
            komunikat: this.komunikat,

```

```
    użytkownik: this.użytkownik,  
    czas: this.czas  
  }  
}  
}  
export default CzatAplikacja;
```

Teraz klasa CzatKomunikat jest gotowa. Czas zająć się następnym modulem aplikacji *Czatownik*.

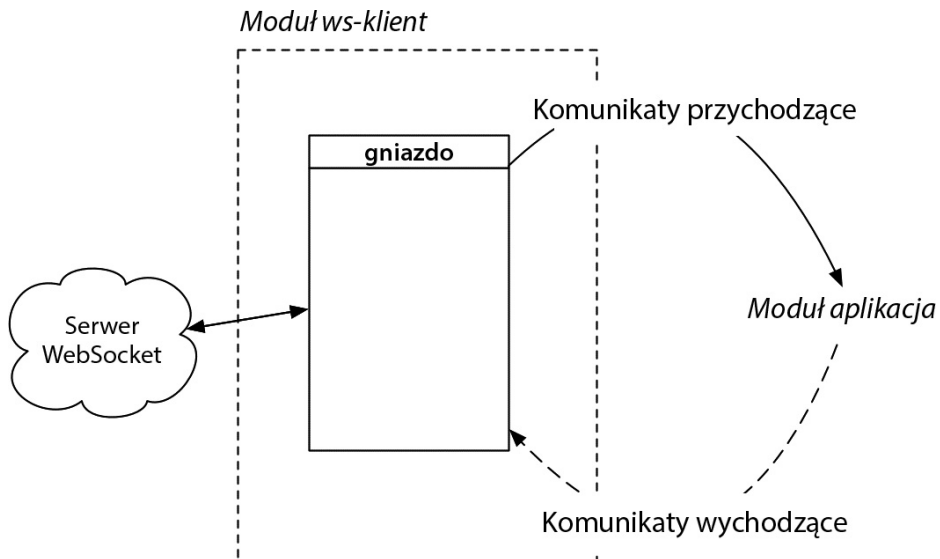
Utworzenie modułu ws-klient

Moduł *ws-klient* będzie obsługiwał komunikację pomiędzy aplikacją a serwerem WebSocket.

Jego zadania to:

- nawiązywanie połączenia z serwerem,
- konfigurowanie komunikacji po nawiązaniu połączenia,
- przekazywanie odbieranych komunikatów do odpowiednich metod,
- wysyłanie komunikatów.

Rysunek 17.14 przedstawia zależności pomiędzy tym modulem a innymi komponentami aplikacji.



Rysunek 17.14. Schemat działania modułu *ws-klient*

Podczas tworzenia części klienckiej również zapoznasz się z nowymi możliwościami wersji ES6.

Obsługa połączeń

Najpierw utwórz kod, który będzie obsługiwał połączenia. Otwórz plik *ws-klient.js* i zadeklaruj w nim zmienną, w której zostaną zapisane informacje o połączeniu:

```
let gniazdo;
```

W powyższej deklaracji zastosowana jest nowa składnia wprowadzona w wersji ES6, tzw. **ograniczanie zakresu** (ang. *let scoping*). Jeżeli zamiast słowa `var` zostanie użyte słowo `let`, wówczas deklaracja zmiennej nie jest **podnoszona** (ang. *hoisted*).

Podniesienie deklaracji zmiennej oznacza, że jest ona przenoszona na początek kodu funkcji, wewnątrz której deklaracja jest umieszczona. Operację tę interpreter JavaScript wykonuje niejawnie, co niestety może być przyczyną trudnych do zdiagnozowania błędów.

Więcej na temat podnoszenia deklaracji dowiesz się na końcu tego rozdziału. Na razie pamiętaj, że zmienne wewnątrz instrukcji `if/else` i wewnątrz pętli bezpieczniej jest deklarować za pomocą słowa `let`.

Teraz w pliku *ws-klient.js* zdefiniuj funkcję inicjującą połączenie:

```
let gniazdo;

function inicjuj(url) {
  gniazdo = new WebSocket(url);
  console.log('Nawiązywanie połączenia...');
}
```

Funkcja `inicjuj` nawiązuje połączenie z serwerem `WebSocket`. Teraz powiąż moduł `ws-klient` z klasą `CzatAplikacja` zapisaną w pliku *aplikacja.js*.

Aby utworzyć funkcjonalny moduł `ws-klient`, należy określić w nim obiekty do wyeksportowania. W tym celu trzeba wyeksportować tylko jeden obiekt z właściwościami będącymi zdefiniowanymi funkcjami. Wykorzystasz tę samą instrukcję `export default`, co na początku rozdziału, oraz kilka dodatkowych usprawnień wprowadzonych w wersji ES6.

Na końcu pliku *ws-klient.js* wpisz instrukcję eksportującą, jak niżej:

```
let gniazdo;

function inicjuj(url) {
  gniazdo = new WebSocket(url);
  console.log('Nawiązywanie połączenia...');
}

export default {
  inicjuj
}
```

Zwróć uwagę, że nie trzeba określać nazw właściwości. Ten uproszczony zapis jest równoważny z poniższym:

```
export default {
  inicjuj: inicjuj
}
```

Jeżeli nazwy klucza i wartości są takie same, wówczas można pominąć dwukropek i nazwę wartości. Jako nazwa właściwości zostanie automatycznie przyjęta nazwa klucza, a wartością właściwości będzie funkcja lub zmienna o tej samej nazwie. Jest to tzw. **rozszerzony literal obiektu** (ang. *enhanced object literal*).

Teraz, po utworzeniu modułu `ws-klient`, czas zaimportować udostępniane przez niego obiekty do modułu aplikacji. Zacznij od wpisania na początku pliku `aplikacja.js` instrukcji importującej:

```
import gniazdo from './ws-klient';

class CzatAplikacja {
  constructor() {
    console.log('Oto wersja ES6!');
  }
}
...
```

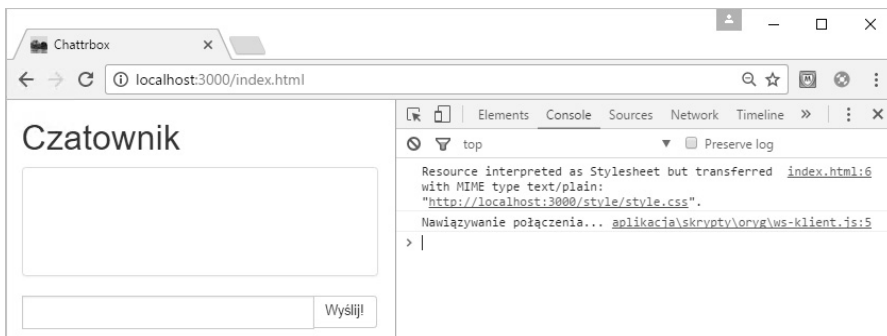
Zostanie w ten sposób utworzony obiekt `gniazdo` wyeksportowany z modułu `ws-klient`.

Teraz w konstruktorze klasy `CzatAplikacja` umieść wywołanie metody `gniazdo.inicjuj` z adresem URL serwera WebSocket w argumencie:

```
import gniazdo from './ws-klient';

class CzatAplikacja {
  constructor() {
    console.log('Oto wersja ES6!');
    gniazdo.inicjuj('ws://localhost:3001');
  }
}
...
```

Gdy zapiszesz zmiany, kod powinien zostać automatycznie przekonwertowany. (Może być konieczne wpisanie w osobnych oknach poleceń `npm run watch` i `npm run dev`, jeżeli któreś z nich zamknąłeś). Odśwież stronę w przeglądarce. W konsoli powinien się pojawić napis `Nawiązywanie połączenia...`, jak widać na rysunku 17.15.



Rysunek 17.15. Komunikat wyświetlany podczas inicjowania połączenia z serwerem WebSocket
Komunikat ten potwierdza, że główna część aplikacji działa poprawnie.

Obsługa zdarzeń i wysyłanie komunikatów

W chwili wywołania metody `inicjuj` w module aplikacja tworzony jest nowy obiekt typu `WebSocket` i nawiązywane jest połączenie z serwerem. Jednak moduł aplikacja musi uzyskać informację o zakończeniu procesu nawiązywania połączenia, aby mógł je wykorzystać do dalszych celów.

Obiekt `WebSocket` zawiera kilka specjalnych właściwości umożliwiających obsługę zdarzeń. Jedną z nich jest `onopen`. Przypisana do tej właściwości funkcja jest wywoływana po nawiązaniu połączenia z serwerem `WebSocket`. Wewnątrz takiej funkcji umieszczany jest dowolny kod, który jest wykonywany po nawiązaniu połączenia.

Aby moduł `ws-klient` był elastyczny i uniwersalny, nie należy na stałe wpisywać kodu, który moduł aplikacja będzie wykonywał po nawiązaniu połączenia. Zamiast tego należy zastosować tę samą technikę co podczas rejestrowania zdarzeń `click` i `submit` w aplikacji *Kafejka*.

W pliku `ws-klient.js` zdefiniuj funkcję `zarejestrujObsługęOtwarcia`. Jej argumentem będzie funkcja zwrotna. Wpisz kod przypisujący właściwości `onopen` funkcję, wewnątrz której będzie wywoływana określona funkcja zwrotna:

```
let gniazdo;

function inicjuj(url) {
  gniazdo = new WebSocket(url);
  console.log('Nawiązywanie połączenia...');
}

function zarejestrujObsługęOtwarcia(funkcjaObsługi) {
  gniazdo.onopen = () => {
    console.log('nawiązane');
    funkcjaObsługi();
  };
}
...
```

Powyższa składnia definicji funkcji jest inna niż stosowana wcześniej. Jest to nowość wprowadzona w wersji ES6, zwana **funkcją strzałkową** (ang. *arrow function*). Jest to skrócona forma funkcji anonimowej. Funkcja strzałkowa oprócz tego, że jest nieco łatwiejsza do wpisania, działa dokładnie tak samo jak funkcja anonimowa.

Parametrem funkcji `zarejestrujObsługęOtwarcia` jest funkcja `funkcjaObsługi`. Właściwości `onopen` obiektu `gniazdo` przypisywana jest funkcja anonimowa. Wewnątrz funkcji anonimowej wywoływana jest podana w parametrze funkcja `funkcjaObsługi`.

(Kod wykorzystujący funkcję anonimową jest bardziej skomplikowany niż zwykle przypisanie `gniazdo.onopen = funkcjaObsługi`. Taki kod sprawdza się w sytuacjach, gdy po pojawieniu się zdarzenia trzeba oprócz wywołania określonej funkcji wykonać dodatkowe operacje, na przykład wyświetlić komunikat, jak w tym wypadku).

Musisz teraz napisać kod obsługujący komunikaty odbierane za pomocą połączenia z serwerem `WebSocket`. W pliku `ws-klient.js` utwórz nową funkcję o nazwie `zarejestrujObsługęKomunikatu`, a następnie przypisz właściwości `onmessage` funkcję strzałkową. Parametrem tej funkcji będzie obiekt reprezentujący zdarzenie.

```
...
function zarejestrujObsługęOtwarcia(funkcjaObsługi) {
  gniazdo.onopen = () => {
    console.log('nawiązane');
    funkcjaObsługi();
  };
}

function zarejestrujObsługęKomunikatu(funkcjaObsługi) {
  gniazdo.onmessage = (e) => {
    console.log('komunikat', e.data);
    let dane = JSON.parse(e.data);
    funkcjaObsługi(dane);
  };
}
...
```

Parametry funkcji strzałkowej, tak jak w wypadku zwykłych funkcji, umieszcza się w nawiasach.

Część kliencka aplikacji *Czatownik* będzie odbierała z serwera obiekt za pomocą funkcji zwrotnej zdefiniowanej wewnątrz funkcji `zarejestrujObsługęKomunikatu`. Obiekt ten reprezentuje zdarzenie i posiada właściwość `data` zawierającą ciąg znaków zapisany w formacie JSON. Po odebraniu obiektu ciąg ten będzie przekształcany na obiekt JavaScript, a następnie będzie przekazywany funkcji `funkcjaObsługi`.

Pozostało jeszcze napisanie kodu wysyłającego komunikat do serwera WebSocket. W pliku *ws-klient.js* zdefiniuj funkcję o nazwie `wyślijKomunikat`. Funkcja ta będzie wykonywała dwie operacje: najpierw będzie przekształcała zawartość komunikatu (czyli jego treść, nazwę użytkownika i czas) na ciąg w formacie JSON, a następnie będzie wysyłała ten ciąg do serwera WebSocket.

```
...
function zarejestrujObsługęKomunikatu(funkcjaObsługi) {
  gniazdo.onmessage = (e) => {
    console.log('komunikat', e.data);
    let dane = JSON.parse(e.data);
    funkcjaObsługi(dane);
  };
}

function wyslijKomunikat(zawartość) {
  gniazdo.send(JSON.stringify(zawartość));
}
...
```

Na koniec, wykorzystując nową składnię, wpisz instrukcje eksportujące nowe funkcje:

```
...
function wyslijKomunikat(zawartość) {
  gniazdo.send(JSON.stringify(zawartość));
}
```

```
export default {
  inicjuj,
  zarejestrujObsługęOtwarcia,
  zarejestrujObsługęKomunikatu,
  wyślijKomunikat
}
```

Teraz plik *ws-klient.js* zawiera pełny kod niezbędny do wysyłania komunikatów i odbierania ich z serwera. Twoim ostatnim zadaniem będzie przetestowanie modułu poprzez wysłanie komunikatu.

Wysyłanie i wyświetlanie komunikatów

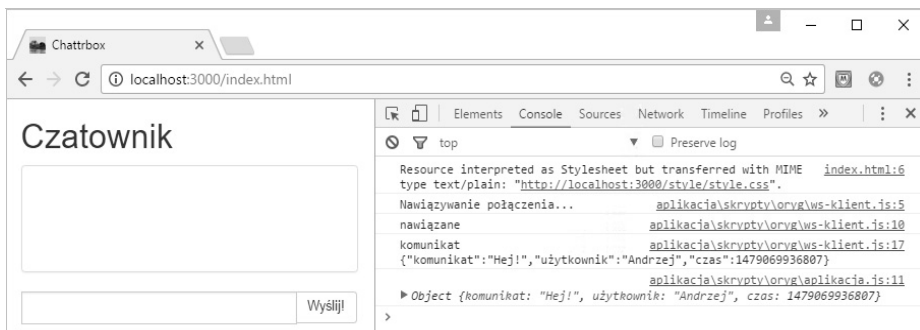
W pliku *aplikacja.js* zmień konstruktora klasy *CzatAplikacja*. Poniżej wywołania metody *gniazdo.inicjuj* wpisz wywołania metod *zarejestrujObsługęOtwarcia* i *zarejestrujObsługęKomunikatu*, umieszczając w ich parametrach funkcję strzałkową:

```
import gniazdo from './ws-klient';

class CzatAplikacja {
  constructor() {
    gniazdo.inicjuj('ws://localhost:3001');
    gniazdo.zarejestrujObsługęOtwarcia(() => {
      let komunikat = new CzatKomunikat({ komunikat: 'Hej!' });
      gniazdo.wyślijKomunikat(komunikat.serializuj());
    });
    gniazdo.zarejestrujObsługęKomunikatu((dane) => {
      console.log(dane);
    });
  }
}
...
```

Powyższy kod natychmiast po nawiązaniu połączenia spowoduje wysłanie testowego komunikatu, a po odebraniu komunikatu — jego wyświetlenie w konsoli.

Zapisz kod, a gdy zostanie przekonwertowany, odśwież stronę w przeglądarce. W konsoli powinna się pojawić informacja, że komunikat został wysłany do serwera i z powrotem z niego odebrany (patrz rysunek 17.16).



Rysunek 17.16. Wysyłanie komunikatu i odebranie go z serwera WebSocket

Świetna robota! Dwa moduły z trzech głównych modułów aplikacji *Czatownik* działają poprawnie. W następnym rozdziale zakończysz pracę nad aplikacją, tworząc moduł wiążący istniejące moduły z interfejsem użytkownika. Nowy moduł będzie wyświetlał komunikaty na liście i umożliwiał wysyłanie komunikatów za pomocą formularza.

Dla ambitnych: konwersja na JavaScript kodu utworzonego w innych językach

Jest kilka języków, których kod można konwertować na JavaScript:

- CoffeeScript: coffeescript.org
- TypeScript: www.typescriptlang.org
- C/C++: kripen.github.io/emscripten-site

Najważniejszym z nich jest język CoffeeScript, oferujący skróconą składnię najczęściej stosowanych instrukcji (na przykład funkcje strzałkowe jako funkcje anonimowe). W dużej mierze wersja ES6 jest wzorowana na tym języku.

Google, Microsoft, Mozilla i inni twórcy przeglądarek współpracują nad projektem WebAssembly, określającym niskopoziomowy kod, do którego będzie kompilowany kod JavaScript. Celem jest opracowanie wydajnego kodu, do którego będzie kompilowany kod utworzony w różnych językach.

Kod WebAssembly ma być uzupełnieniem — ale nie zamiennikiem — dla JavaScript, wykorzystującym zalety różnych innych języków. Na przykład JavaScript dobrze nadaje się do tworzenia aplikacji przeglądarek, ale nie jest odpowiedni do tworzenia gier z zaawansowaną grafiką, wymagającą wykonywania wielu obliczeń matematycznych. Do takich celów z kolei doskonale nadają się języki C i C++. Zamiast przekładać kod C++ na JavaScript i ryzykować popełnienie błędu, będzie można skompilować go do WebAssembly.

Projekt WebAssembly zrodził się z wcześniejszego projektu asm.js, definiującego odmianę języka JavaScript umożliwiającą tworzenie wydajnego kodu.

Więcej informacji o projektach asm.js i WebAssembly można znaleźć na blogu twórcy języka JavaScript pod adresem brendaneich.com/2015/06/from-asm-js-to-webassembly.

Wyzwanie brązowe: domyślna nazwa importowanej zmiennej

W pliku *podst.js* instrukcja `import` tworzy lokalną zmienną o nazwie `CzatAplikacja`. Co się stanie, gdy tę nazwę zmienisz na `AplikacjaDoCzatowania`?

Zrób tak (nie zapomnij o wprowadzeniu odpowiedniej zmiany w instrukcji `new` w następnym wierszu) i sprawdź, czy aplikacja działa poprawnie. Jeżeli tak jest albo też nie, to dlaczego tak się dzieje?

Wyzwanie srebrne: komunikat o zamkniętym połączeniu

W pliku *ws-klient.js* zdefiniuj jeszcze jedną funkcję — o nazwie `zarejestrujObsługęZamknięcia`. Jej parametrem niech będzie funkcja zwrrotna wywoływana w chwili pojawienia się zdarzenia `close` dla obiektu `gniazdo`.

W pliku *podst.js* wykorzystaj funkcję `zarejestrujObsługęZamknięcia` do wyświetlania komunikatu o zamknięciu połączenia. Następnie sprawdź, czy funkcja działa poprawnie.

Jak można to zrobić? Oczywiście nie możesz zamknąć okna przeglądarki. Połączenie będziesz musiał zamknąć po stronie serwera.

Jako dodatkowe zadanie napisz funkcję, która będzie próbowała wznowić zamknięte połączenie. Możesz w tym celu użyć funkcji `setTimeout` lub funkcji wyświetlającej pytanie o zgodę na wznowienie połączenia (szczegółowe informacje znajdziesz na stronie MDN).

Dla ambitnych: podnoszenie deklaracji

Język JavaScript został zaprojektowany z myślą o niezawodowych programistach, aby mogli oni tworzyć strony z podstawowymi elementami interaktywności. Choć język ten z założenia miał umożliwiać tworzenie kodu niewrażliwego na błędy, jednak niektóre jego cechy przyczyniają się do popełniania błędów. Jedną z nich jest podnoszenie deklaracji.

Gdy interpreter JavaScript analizuje kod, przesuwając deklaracje wszystkich zmiennych na początek funkcji, w której te deklaracje są umieszczone (a jeżeli deklaracje nie znajdują się wewnątrz funkcji, wówczas są przenoszone na początek całego kodu).

Najlepiej pokazać to na przykładzie. Następujący kod:

```
function wyświetlKilkaWartości() {  
  console.log(mojaZmienna);  
  var mojaZmienna = 5;  
  console.log(mojaZmienna);  
}
```

...zostanie zinterpretowany jako:

```
function wyświetlKilkaWartości() {  
  var mojaZmienna;  
  console.log(mojaZmienna);  
  mojaZmienna = 5;  
  console.log(mojaZmienna);  
}
```

Jeżeli w konsoli wywołasz funkcję `wyświetlKilkaWartości`, pojawi się coś takiego:

```
> wyświetlKilkaWartości();  
undefined  
5
```

Zwróć uwagę, że podnoszona jest tylko **deklaracja** zmiennej, a instrukcja przypisująca jej wartość pozostaje na swoim miejscu. Oczywiście takie działanie kodu wprowadza zamieszanie, szczególnie jeżeli zmienne są deklarowane wewnątrz instrukcji `if` lub w pętli. W innych językach nawiasy klamrowe definiują blok kodu, który ma własny zakres widoczności deklaracji. W języku JavaScript nawiasy klamrowe nie określają zakresu widoczności. Określają go jedynie funkcje.

Weźmy inny przykład:

```
var mojaZmienna = 11;
function nigdyTakNieKoduj() {
  if (mojaZmienna > 10) {
    var mojaZmienna = 0;
    console.log('mojaZmienna ma wartość większą od 10; zeruję ją.');
```

Zapewne spodziewasz się, że powyższy kod wyświetli komunikat `mojaZmienna ma wartość większą od 10; zeruję ją`. Niestety zamiast tego pojawi się:

```
> nigdyTakNieKoduj();
Nie trzeba zerować zmiennej.
undefined
```

W tym kodzie deklaracja `var mojaZmienna` jest przenoszona na początek funkcji, a więc przed instrukcję `if`, w miejsce, w którym zmienna `mojaZmienna` ma wartość `undefined`. Natomiast instrukcja przypisująca wartość zmiennej pozostaje na swoim miejscu wewnątrz bloku.

Deklaracje funkcji również są podnoszone, ale w całości. Oznacza to, że poniższy kod będzie działał poprawnie:

```
test();
// Deklaracja funkcji po jej wywołaniu:
function test() {
  console.log('TEST');
```

Interpreter JavaScript przesuwca całą deklarację funkcji na początek kodu, dzięki czemu funkcję `test` można wywoływać bez żadnych problemów:

```
> test();
TEST
```

Instrukcja `let` jest odporna na podnoszenie, podobnie jak instrukcja `const` służąca do deklarowania zmiennych, których wartości nie można zmieniać.

Dla ambitnych: funkcja strzałkowa

Trochę skłamałiśmy. Funkcja strzałkowa nie działa dokładnie tak samo jak funkcja anonimowa. W niektórych sytuacjach działa *lepiej*.

Funkcja strzałkowa, oprócz prostszej składni, posiada następujące cechy:

- Działa tak, jakby została wpisana instrukcja `function() {}.bind(this)`, dzięki której właściwość `this` wewnątrz funkcji strzałkowej ma wartość zgodną z oczekiwaniami.
- Jeżeli funkcja zawiera tylko jedną instrukcję, można pominąć nawiasy klamrowe.
- Jeżeli nawiasy klamrowe zostaną pominięte, funkcja zwróci wynik jednej instrukcji.

Rozważmy na przykład metodę `dodajObsługęKliknięcia` z aplikacji *Kafejka*:

```
ListaZamówień.prototype.dodajObsługęKliknięcia = function(fn) {
  this.$element.on('click', 'input', function(zdarzenie) {
    var email = event.target.value;
    fn(email)
    .then(function() {
      this.usuńWiersz(email);
    }.bind(this));
  }.bind(this));
};
```

Po zamianie funkcji anonimowych na funkcje strzałkowe kod staje się nieco bardziej czytelny:

```
ListaZamówień.prototype.dodajObsługęKliknięcia = (fn) => {
  this.$element.on('click', 'input', (zdarzenie) => {
    let email = event.target.value;
    fn(email)
      .then(() => this.usuńWiersz(email));
  });
};
```

Dzięki usunięciu słowa `function` i instrukcji `.bind(this)` operacje wykonywane przez metodę `dodajObsługęKliknięcia` stają się bardziej czytelne.

Skorowidz

A

adapter, 421, 423, 424, 425
 domyślny, 428
 instalowanie, 425
 JSONAPIAdapter, 423
adres URL, 122, 277, 397, 401, 404
 zapytania, 319, 423
Ajax, 273, 274
akcja, 449, 453, 454, 457, 463,
 466, 474, 475
 domknięcia, 477
 ścieżki, 462
alias, 460
animacja, 171
architektura MVC, *Patrz:* MVC
Atom, 19
 autouzupełnianie kodu, 39
 wtyczka, 21
 api-docs, 22
 atom-beautify, 21, 45
 autocomplete-paths, 22
 emmet, 21, 39
 linter, 23, 89
atrybut
 alt, 45
 autofocus, 218, 219
 class, 56, 79, 124, 157, 160, 163,
 172, 439
 data, 124, 216
 data-czas, 379
 href, 45, 67, 122
 id, 83
 nazwy, 248
 pattern, 263
 required, 261, 262

selected, 223
selektor, *Patrz:* selektor
 atrybutu
src, 122, 439
type, 219, 221, 248
value, 224

B

Babel, 340, 341, 343
Béziera krzywa, 176
bezpieczeństwo, 421, 427, 430
biblioteka, 395
 crypto-js, 371
 Ember Data, 409, 428, 429
 jQuery, *Patrz:* jQuery
 socket.io, 336, 337
 Webshim, 271, 272
 zewnętrzna, 390
Bootstrap, 213, 214, 215, 244, 390,
 471
 dokumentacja, 221
 formatowanie pól, 221
 klasa, 216
 lista rozwijana, 222
 pasek nawigacyjny, 393, 394
 przycisk, 225
 suwak, 224

C

CDN, 55
chat, 333
Chrome, 289
 wtyczka Ember Inspector, 387
CLI, 386

content delivery network, *Patrz:*
 CDN
Content Security Policy, 427
CRUD, 418, 457
CSS, 39, 50, 54
 definiowanie, 57
 deklaracja, *Patrz:* deklaracja
 domyślny, 82, *Patrz:*
 styl:agenta
 reguła, *Patrz:* reguła
 selektor, *Patrz:* selektor
cykl wywoływania metod
 zwrotnych, 397
czat, 342, 364, 369
czcionka, 79, 80
 Awesome, 395

D

debuger, 201, 203
deklaracja
 bottom, 105
 display, 61, 89, 91, 100, 157
 flex, 97
 flex-direction, 93
 height, 166
 inline, 61, 91
 justify-content, 102
 left, 105
 order, 99, 100
 padding, 170
 position, 105, 106, 109
 prototype, 191
 right, 105
 text-shadow, 107
 top, 105

deklaracja
 transform, 166
 transform, 166, 167
 transition, 171, 173
 width, 89, 101, 166
dekorator, 415, 466
dokument, 41
DOM, 50, 133, 152, 216, 229, 231,
 244, 247, 465
domknięcie, 152
DRY, 469
dyrektywa
 @import, 392
 use strict, 151

E

echo, 332
edytor
 Atom, *Patrz:* Atom
 Brackets, 35
 Visual Studio Code, 35
efekt przejścia, 165, 168, 170, 172
 prędkość, 171, 172
 wywołanie w JavaScript, 173
element, *Patrz też:* znacznik
 {{#link-to}}, 470
 {{action}}, 454
 {{x-option}}, 452
 {{x-select}}, 452
 {{yield}}, 467
 body, 92, 95, 157, 159, 163
 border, 61, 62
 button, 225
 div, 156, 216, 248
 DOM, 231
 form, 216, 230
 header, 95, 97
 HTML kolekcja, *Patrz:*
 kolekcja
 il, 158
 input, 217, 218, 219, 248, 255
 label, 218, 248
 li, 76, 166
 main, 97
 margin, 61, 62
 padding, 61, 62
 położenie, 109
 bezwzględne, 105
 pomocniczy, 434, 435
 {{#each}}, 437, 466
 {{#link-to}}, 441, 442, 443,
 470
 {{/each}}, 437, 439, 454
 {{action}}, 454
 {{each}}, 439
 {{else}}, 437, 439
 blokowy, 441
 niestandardowy, 444, 445
 warunkowy, 435, 436
 potomny, 63
 pudełko, 59
 referencja, 125, 133, 141
 responsywny, 394
 section, 216
 ul, 74, 156, 158
 warunkowy
 {{bind-attr}}, 440
 wierszowy, 440
 wierszowy
 {{#if}}, 440
 {{if}}, 440
 niezawierający znaku #, 470
 zagnieżdżenie, 77
 zastępczy, 45
 zawartość, 61, 62
Ember, 383, 385, 388, 397, 409,
 421, 463, 465
Ember CLI, 386, 389, 390, 398
 instalowanie, 386
Ember Inspector, 404, 426
etykieta, *Patrz:* element label

F

favicon, 51, 52
Flexbugs, 118
format JSON, 349, 425
formularz, 216, 217, 227, 289, 349
 pole
 adres e-mail, 219
 automatyczna aktywacja, 218
 błędne, 269

 lista rozwijana, 222
 przycisk, 225
 przykładowy tekst, 220
 suwak, 224
 wyboru, 221, 248
przetwarzanie, 228, 233, 237,
 238
resetowanie, 239
weryfikacja, 261, 263, 265,
 266, 267, 268
funkcja, 135
 anonimowa, 143, 144, 146,
 152, 161, 183
 forma skrócona, 355
 wywoływana natychmiast,
 183
argument, 139
czasu, 171, 172
 ease, 176
 ease-in, 176
 ease-in-out, 176
 ease-out, 176
 linear, 176
 tworzenie, 175
 wbudowana, 175
Error, 231, 232
eval, 151
nadpisanie, 184
nazwana, 143, 150, 152
parametr, 137, 139, 140, 144,
 145, 151
pomocnicza, 434
przekształcanie w IIFE, 185
require, 325
setTimeout, 359
strzałkowa, 355, 357, 361, 365
właściciel, 206
właściwość this, 205, 206, 211
zasięg, 190
 globalny, 146
zwracająca wartość, 140
zwrotna, 143, 144, 145, 149,
 150, 152, 205, 206, 283, 291,
 323, 367
 wywołanie, 235, 236
 zagnieżdżanie, 291

G

Google Chrome, 19
Gravatar, 370

H

hash, 371
HTML, 42
HTMLBars, 434
hypertext markup language,
 Patrz: HTML

I

IIFE, 184, 185, 210
 argument, 186
immediately invoked function
 expression, *Patrz:* IIFE
instrukcja, 136
 const, 366
 export, 346, 347, 353
 extends, 375
 return, 141, 251
 with, 151
interfejs
 API, 122
 Constraint Validation API,
 263, 265, 270
 DOM API, 229
 graficzny, 363
 JSON API, 421
 REST, 410, 421
 użytkownika, 213, 243, 465

J

JavaScript, 121, 125, 129, 359
 kompatybilność wsteczna, 148
 typ zmiennej, *Patrz:* typ
 wersja, 121, 339, 340
 ES6, 339, 340, 343, 345, 353,
 363, 366
język
 CSS, 50, 390
 Handlebars, 434
 HTMLBars, 440

JavaScript, *Patrz:* JavaScript
 programowania
 C/C++, 358
 CoffeeScript, 358
 TypeScript, 358
 Ruby on Rails, 421
 SCSS, 390
jQuery, 228, 229, 231, 247, 272,
 275, 277, 364
importowanie, 230

K

catalog
 ścieżka, 27, 29
 tworzenie, 28
 zawartość, 31
klasa, 190
 ApplicationRoute, 401
 container, 215
 DS, 410
 eksport nazwanych wartości,
 365
 Ember.Object, 409, 413
 Ember.Route, 400
 Ember.String, 428
 panel, 216
 panel-body, 216
 panel-default, 216
 Router, 401
 wiązanie, 471
 XMLHttpRequest, 274
klucz, 188, 192, 217, 276, 277, 350
 includePaths, 391
 nazwa, 354
kod
 autouzupełnianie, 39, 44
 diagnostyka, 201, 203, 205
 JavaScript, 125
 tworzenie iteracyjne, 129
 kompilacja, 390
 linting, 21
 moduł, *Patrz:* moduł
 otwarty, 390
 stanu HTTP, 274
 zewnętrzny, 230
koercja, 163, 177

kolejność źródłowa, 99, 100
kolekcja, 152, 450, 451
kolor, 71, 81
 format, 72
 HEX, 72
 HSLA, 72
 rgb, 71, 72
 rgba, 71, 73
 wybranie, 72
komentarz, 66
kompilator, 391
komponent, 465, 466, 467, 470
 tworzenie, 471
komunikat, 349, 350, 367
 lista, 368
 przypisanie destrukuryzujące,
 350
 styl, 471, 474, 475
 wysyłanie, 355, 356, 357
 wyświetlanie, 357
 znacznik czasu, 377, 379
konsola, 129, 181
 komunikat o błędzie, 231
konstruktor, 190, 191, 210
 metoda, 193
 parametr, 196
 WebSocket, 337
kontener
 elastyczny, 92, 96, 98
 kolejność, 99, 100
 z odwróconymi osiami, 98
 wyśrodkowanie, 102
kontroler, 384, 449, 463, 465, 470
 akcja, *Patrz:* akcja
 aplikacji, 474
 przetwarzanie ścieżek, 462, 463
 realizacja algorytmu, 462
 tworzenie, 453, 459
krzywa Béziera, 176

L

linting kodu, 21
lista
 miniatur, 86, 89
 nienumerowana, 43, 74, 95
 obiektów, 152

lista

- przewijana poziomo, 86, 89
- punktowana, *Patrz:* lista
- nienumerowana
- rozwijana, 222, 224
- węzłów, 148

literal

- funkcyjny, 143
- rozszerzony obiektu, 354

M

MDN, 24, 66, 118, 122, 151, 152, 168, 172

typ, 325

menu rozwijane, 165

metadane, 316

metoda, 132

- \$, 231, 232
- \$.ajax, 285, 293, 295
- \$.get, 282, 283, 284, 285, 293
- \$.post, 277, 278, 285, 293
- addEventListener, 161, 233
- afterModel, 404, 407, 408
- ajax, 275
- ajaxError, 426
- ajaxOptions, 426
- append, 251
- attr, 410
- beforeModel, 404, 407, 440
- belongsTo, 415
- bind, 206, 211, 253, 259, 299
- call, 253
- classList.add, 163
- classList.remove, 163
- console.log, 145, 161, 200, 316
- createRecord, 419, 450
- deleteRecord, 419, 456, 457
- deserialize, 430
- destroyRecord, 418, 460
- DS.attr, 429
- Ember.computed, 415, 416
- Ember.computed.alias, 460, 462
- Ember.Handlebars.SafeString, 445
- Ember.RSVP.hash, 450, 451

Ember.String.underscore, 428

extend, 400

find, 256

findAll, 417, 418, 456

findRecord, 417, 418

forEach, 152, 206, 211

fs.readFile, 320

get, 275, 413, 415

getAttribute, 140

getElementsByTagName, 152

handleResponse, 426

hasMany, 410, 411, 415

http.createServer, 315

jQuery, 230

keyForAttribute, 428

keyForRelationship, 428

konstruktora, *Patrz:*

konstruktor metoda

model, 404, 406

modelNameFromPayloadKey, 428

modelRecord.destroyRecord, 419

modelRecord.save, 418

on, 233

peekAll, 417, 418

peekRecord, 417, 418

preventDefault, 145, 161

query, 417, 418

queryRecord, 417, 418

querySelector, 132, 147

querySelectorAll, 147, 148, 152

readFile, 319

relacyjna, 410

remove, 255

reset, 239

route, 401

save, 418

serialize, 428, 430

serializeArray, 234, 235

server.listen, 315

set, 275, 413, 415

setAttribute, 133, 140

setCustomValidity, 267, 268, 271

setupController, 404, 406, 407

submit, 365

then, 292, 295

obsługa błędów, 296

this.store.createRecord, 412

transitionToRoute, 455

url.substring, 320

window.jQuery, 230

zwrotna, 404

Mirage, 430

model, 384, 449

danych, 409, 411, 413

wiązanie właściwości, 439

DOM, *Patrz:* DOM

flexbox, 85, 96, 98, 99, 114, 118

błędy, 118

pudełkowy, 59, 60, 61

border, 61

margin, 61

padding, 61

zawartość, 61

moduł, 182

dołączanie do przestrzeni

nazw, 188

http, 331

pakowanie, 345

szablon, *Patrz:* szablon

tworzenie, 187, 322

ws, 331, 336

wscat, 333

Mozilla Developer Network, *Patrz:* MDN

MVC, 383, 385, 433, 449

kontroler, *Patrz:* kontroler

model, *Patrz:* model

widok, 384, 433, 449, 465

N

nagłówek, 41, 215

Content-Type, 325

NaN, 177

narzędzie

Babel, *Patrz:* Babel

Babelify, 341

Bower, 387

Broccoli, 389, 391

Browserify, 341, 345, 364, 347
 browser-sync, 33, 34, 46, 88, 113
 dla programistów, 48, 50, 77,
 151, 159
 panel diagnostyczny, 201
 Ember CLI, *Patrz:* Ember CLI
 ember-cli-sass, 390
 Watchify, 341, 348, 370
 Watchman, 387
 Node, 311, 313, 316
 funkcja zwrotna, 323
 instalacja, 33
 manifest, 313
 obsługa błędów, 323, 324
 Node.js, *Patrz:* Node
 not a number, *Patrz:* NaN

O

obiekt, 154, 434
 ApplicationRoute, 401
 Component, 412
 contentSecurityPolicy, 427
 Controller, 412
 Deferred, 293, 294, 295, 298,
 303
 stan, 295, 296, 298
 document, 132, 133
 IndexRoute, 401
 instancja, 190
 JSONAPIAdapter, 421
 literal rozszerzony, 354
 localStorage, 374, 375, 409
 Promise, 292, 297, 303, 305, 404
 stan, 292
 Route, 412
 Router, 397
 sassOptions, 391
 sessionStorage, 374, 375
 store, 412, 418, 423
 tworzenie, 190
 WebSocket, 355
 XMLHttpRequest, 274, 277
 zawierający referencje do
 elementów, 232
 zdarzenia, 144, 161

obraz, 44, 45, 87
 dopasowanie do wielkości
 okna, 69
 ukrywanie, 156, 158, 159
 płynne, 165
 wyśrodkowanie, 102
 wyświetlanie, 163
 płynne, 165
 obserwator zdarzeń, 142, 144,
 146, 161, 232, 298
 obszar roboczy, 113
 układu, *Patrz:* przeglądarka
 obszar roboczy rzeczywisty
 widoku, 112
 orientacja, 118
 odnośnik, 45, *Patrz też:*
 zakotwiczenie
 operator
 domyślny, 189
 logiczny lub, *Patrz:* operator
 domyślny
 przypisania, 128
 równości
 ściślej, 163, 177
 zwyklej, 163, 177

P

pasek nawigacyjny, 393, 394
 pętla `{{#each}}`, 466
 platforma
 Bootstrap, *Patrz:* Bootstrap
 Ember, *Patrz:* Ember
 Firebase, 337
 Node.js, *Patrz:* Node
 plik
 app.scss, 392
 bootstrap.js, 392
 bootstrap.min.css, 214
 bower.json, 395
 ember-cli-build.js, 391, 395
 favicon.ico, 51, 52
 index.html, 38, 43, 54, 214, 230
 index.js, 402
 normalize.css, 54, 55, 214
 obrazu, 44
 package.json, 313, 395
 SCSS, 392
 kompilowanie, 390
 serwer-websocket.js, 331, 332,
 334
 style.css, 39, 42, 55, 71
 udostępnianie, 318
 polecenie
 build, 347
 cd, 29, 46
 dir, 31
 echo, 27
 ember g adapter application,
 424
 ember g component, 471
 ember g controller, 459
 ember g controller application,
 474
 ember g model, 409
 ember g route, 409
 ember g serializer, 427
 ember generate, 398, 400
 ember install ember-cli-
 mirage, 431
 ember install emberx-select, 451
 ember server, 389, 400
 ls, 31
 man sudo, 32
 mkdir, 28
 node, 33, 313
 node serwer-websocket.js, 332
 npm, 33, 313
 npm init, 313
 npm install, 317, 331
 npm run dev, 317
 npm scripts, 313
 npm start, 316
 pwd, 28
 sudo, 32
 wscat, 335
 Postman, 289
 potomek, *Patrz:* element potomny
 projektowanie atomiczne, 54, 69
 wyjątek, 98
 protokół
 HTTP, 329
 WebSocket, *Patrz:* WebSocket

przeglądarka, 38
 Google Chrome, 19
 obszar roboczy, 111, 112
 rzeczywisty, 112
 Safari, *Patrz:* Safari
 w urządzeniu przenośnym,
 112, 114
przestrzeń nazw
 dodawanie modułu, *Patrz:*
 moduł dodawanie do
 przestrzeni nazw
 globalna, 184
 modyfikowana, 194
 referencja, 189
przezroczystość, 73
przycisk, 225
przypisanie destrukuryzujące, 350
pseudoklasa, 168, 169
pudełko elementu, *Patrz:* element
 pudełko

R

reguła
 @font-face, 80, 81
 modyfikator
 hover, *Patrz:* pseudoklasa
 hover
rekord tymczasowy, 450
relacja, 410, 411
Resig John, 229

S

Safari, 24, 263, 270
Sass, 390
selektor, 57
 atrybutu, 58, 66, 83
 bliźniaczy, 75, 76
 przylegający, 75, 76
 elementu, 58, 62, 83, 368
 filtrujący, 258
 identyfikatora, 83
 klasy, 83
 nazwa, 366
 pochodny, 75

 potomny, 75, 76
 precyzja, 58, 83
 kalkulator, 83
 relacyjny, 75
 serializator, 421, 427
 domyślny, 428
 serwer, 38
 WebSocket, 337, 352
słowo
 @media, 115
 kluczowe
 class, 343
 new, 192, 251
 var, 128, 151
Solved by Flexbox, 118, 119
stała, 128, 366
sterownik, 397, 441
strona
 interaktywna, 121
 responsywna, 85, 111, 215
 współdzielenia treści, 54
styl, *Patrz:* CSS
 agenta, 61
 dziedziczenie, 62, 63, 64, 66
superużytkownik, 32
suwak, 224
szablon, 183, 393, 406, 433, 434,
 436, 440, 451, 463
 zagnieżdżanie kodów, 393
szkielet, 188

Ś

ścieżka URL, 275, 276, 397, 401
 hierarchia, 401
 sprawdzanie, 404
 zagnieżdżanie, 402

T

tablica, 148, 152, 154
terminal, *Patrz:* wiersz poleceń
transformata, 421, 429
 DS.attr, 429
translator, 340, 343
trasa URL, 384
tryb ścisły, 151

typ, 154
 boolean, 410
 date, 410
 koercja, *Patrz:* koercja
 konwersja, 163
 MIME, 325, 326, 327, 340
 null, 153
 number, 410
 podstawowy, 131, 153
 string, 410, 411
 undefined, 153
 własny, 190
 złożony, 154

U

układ
 adaptacyjny, 111
 alternatywny, 111
 Holy Grail, 119
 oryginalny, 111
usługa
 Gravatar, *Patrz:* Gravatar
 REST, 275
 WebSocket, 337
użytkownik, 364
 administrator, *Patrz:*
 administrator
 e-mail, *Patrz:* hash
 gravatar, 370, 372
 nazwa, 368, 372
 sesja, 374
 uprawnienia, 32

W

Walton Philip, 118
wartość NaN, 177
WebAssembly, 358
WebSocket, 311, 329, 330, 340, 352
 konfiguracja, 331
 testowanie, 333
 uruchamianie, 332
wiersz poleceń, 26, 313, 386
 macOS, 27
 Windows, 26

właściwość, *Patrz*: deklaracja
 współczynnik alfa, *Patrz*:
 przezroczystość
 wyrażenie
 funkcyjne, 183
 natychmiast wywoływane,
 Patrz: IIFE
 regularne, 263, 264
 wzorzec MVC, *Patrz*: MVC

Z

zakotwiczenie, 44, 45, *Patrz też*:
 odnośnik
 zapytanie, 38
 Ajax, 273, 274, 286
 badanie, 279, 281
 DELETE, 275, 285, 286
 GET, 275, 283, 418, 423
 mechanizm bezpieczeństwa, 427
 POST, 227, 275, 277, 285, 418
 PUT, 275, 418
 zapytanie medialne, 114, 115, 158
 tworzenie, 115
 zasada
 CRUD, 457, 463
 DRY, 469
 następstwa, 82
 Zaytsev Juriy, 340
 zdarzenie, 142
 blur, 265
 click, 245, 454

connection, 331
 delegowanie, 258
 keypress, 161
 keyup, 161
 message, 332
 MouseEvent, 145
 obiekt, *Patrz*: obiekt zdarzenia
 obserwator, *Patrz*: obserwator
 zdarzeń
 obsługa, 355
 onchange, 452
 submit, 232, 233, 238, 265, 365
 zmienna, 128, 146, 184
 \$, 230
 \$div, 248
 deklaracja
 niepodnoszona, 353
 podnoszona, 353, 359, 360
 lokalna, 146
 nazwa, 230
 domyślna, 358
 obiektowa, 131
 ograniczanie zakresu, 353
 prywatna, 211
 zasięg, 146
 znacznik, 41, *Patrz też*: element
 a, 45, 122, 125
 atrybut, 42, 45, *Patrz też*:
 atrybut
 body, 42
 div, 88, 216, 466
 form, 216

h1, 41
 header, 42
 img, 45, 88, 122, 125, 439
 label, 218
 li, 43
 link, 42, 45, 214
 meta, 41, 113, 114
 pusty, *Patrz*: znacznik
 samozamykający
 samozamykający, 45
 script, 125, 186, 230
 section, 216
 select, 452
 span, 43, 56, 88
 ul, 43, 44
 znak
 #, 83
 \$, 230, 231, 248
 (), 183
 /* */, 66, 136
 //, 136
 {{}}, 406, 434
 {}, 189
 ||, 189
 +, 177
 ==, 163, 177
 ===, 163, 177

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Programista WWW – mistrz wielu technologii!

Tworzenie aplikacji WWW jest dziś dla programistów sporym wyzwaniem. Pisząc kod, muszą uwzględnić mnóstwo różnych przeglądarek i jeszcze więcej urządzeń, na których będzie można uruchomić aplikację. Co więcej, pisanie aplikacji WWW oznacza konieczność zadbania zarówno o wygląd strony, jak i o poprawność algorytmów decydujących o sposobie działania. To wszystko oznacza, że *dobry* programista aplikacji WWW musi *dobrze* opanować wiele technik pracy!

Książka ta opiera się na pięciodniowym szkoleniu Big Nerd Ranch i jest przeznaczona dla projektantów, którzy znają podstawy tworzenia aplikacji webowych. Poznasz dzięki niej nowoczesne techniki programistyczne i sposoby ich wykorzystania. Zaimplementujesz także responsywny interfejs użytkownika i aplikację współdziałającą z serwerem internetowym. Opanujesz platformy Ember i Node.js i nauczysz się używać najnowocześniejszych narzędzi do diagnozowania i testowania kodu. Dzięki tej książce szybko zaczniesz tworzyć nowoczesne, elastyczne i wydajne aplikacje WWW!

W tej książce:

- przedstawiono podstawy składni języka Swift
- omówiono konstrukcje służące do kontroli przepływu działania programu
- pokazano, jak korzystać z kolekcji, typów wyliczeniowych, struktur i klas
- zaprezentowano zasady budowania eleganckiego, czytelnego i efektywnego kodu
- przedstawiono metody projektowania aplikacji opartej na zdarzeniach



Chris Aquino

jest dyrektorem działu rozwoju aplikacji WWW i instruktorem szkoleniowym w firmie Big Nerd Ranch. Niezwykle chętnie przekazuje innym wartościową wiedzę. Uwielbia nakręcanie zabawki, espresso i wszelkie formy grillowania.



Todd Gandee

jest instruktorem i programistą aplikacji WWW w Big Nerd Ranch. Swoje umiejętności doskonalił przez 10 lat jako doradca w dziedzinie tworzenia aplikacji internetowych. Poza programowaniem lubi się wspinać, biegać i jeździć na rowerze.

Helion

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Informatyka w najlepszym wydaniu

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

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



**Big Nerd
Ranch**

ISBN 978-83-283-3203-4



cena: 69,00 zł

sięgnij po **WIĘCEJ**



KOD KORZYŚCI