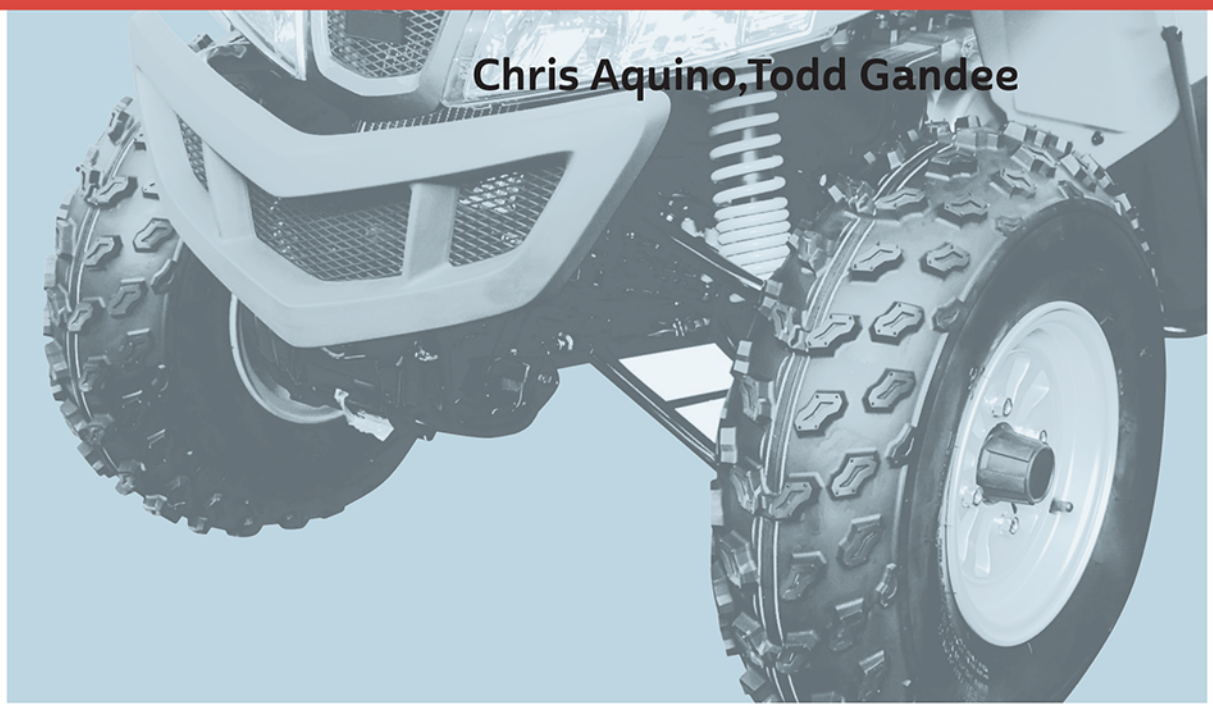
Podręcznik frontendowca

Big Nerd Ranch Guide

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.

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.

Spis treści

Część IV Architektura aplikacji

17

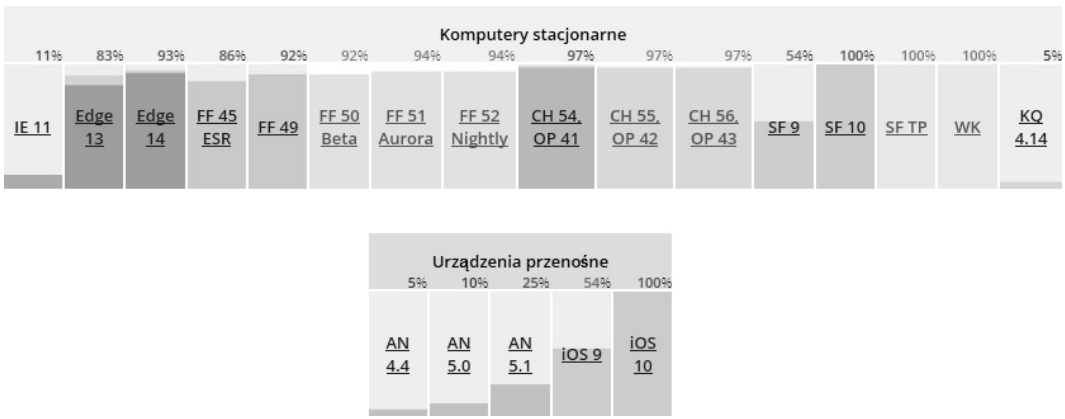
Wersja JavaScript ES6 i translator Babel

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.

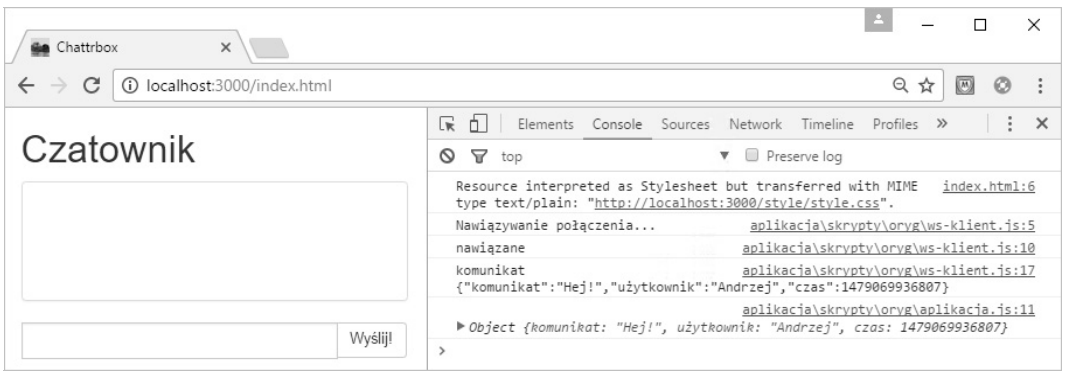
Choć inne przeglądarki mogą gorzej obsługiwać wersję ES6, jednak my ją bardzo, ale to bardzo lubimy. Jest to doskonały język, który warto zacząć stosować jak najszybciej, nie czekając, aż będzie powszechnie obsługiwany.

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.

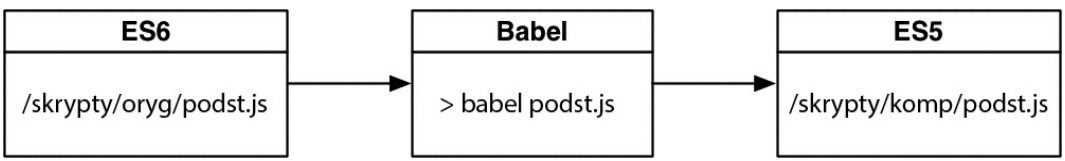
Do biegu, gotowi, start! Na końcu tego rozdziału aplikacja *Czatownik* będzie komunikowała się za pomocą protokołu WebSocket z serwerem (patrz rysunek 17.2).



Rysunek 17.2. Wygląd aplikacji Czatownik na końcu tego rozdziału

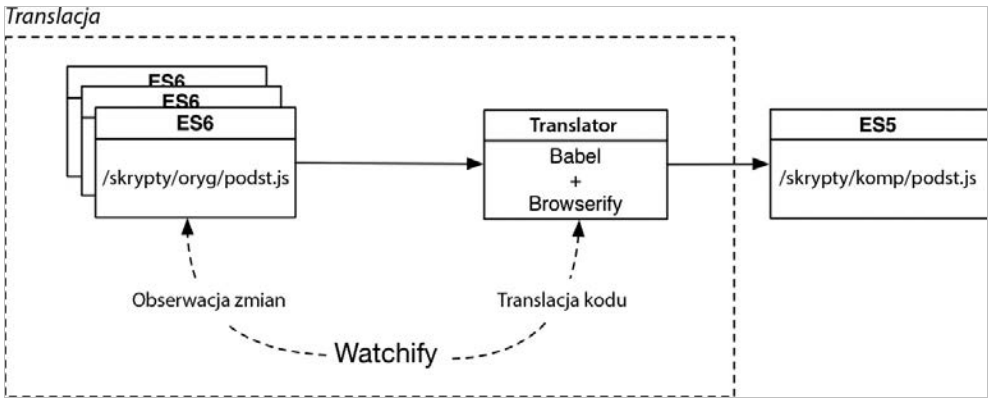
Narzędzia do translacji kodu JavaScript

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.

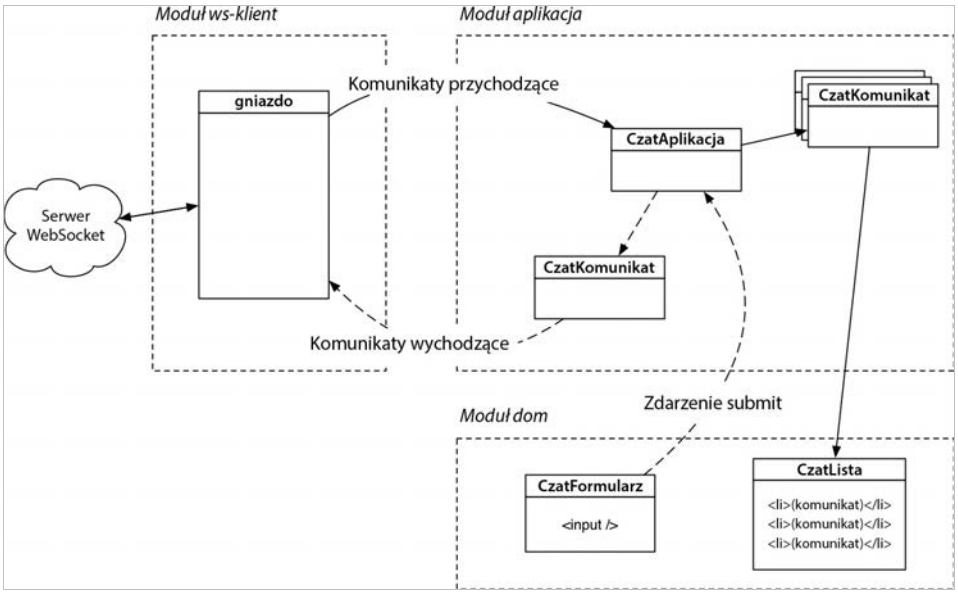
Część kliencka aplikacji Czatownik

Utworzyłeś już część serwerową aplikacji *Czatownik*, wysyłającą statyczne pliki i obsługującą 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.



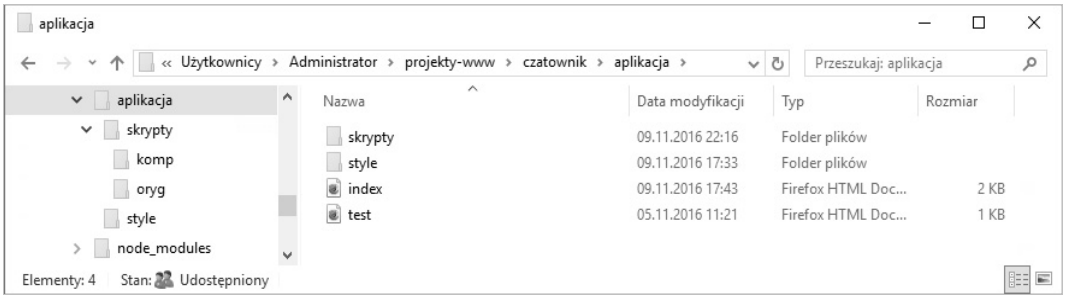
Rysunek 17.5. Moduły aplikacji Czatownik

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

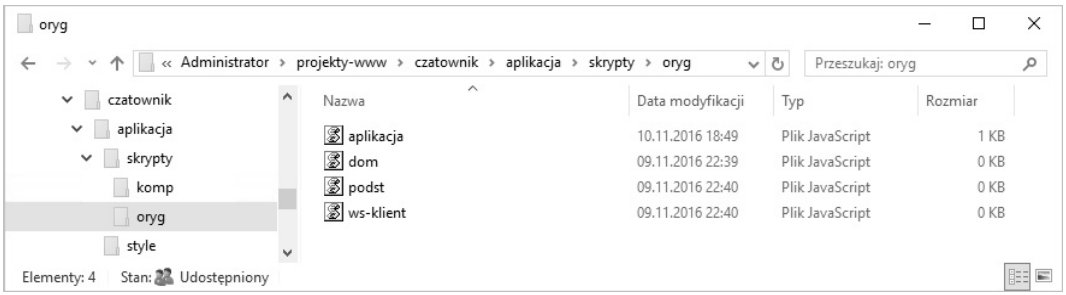
Teraz w folderze *skrypty\oryg* utwórz cztery pliki, które będą zawierały kod JavaScript:

- *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 translatorem Babel

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

Na razie z translatora 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:

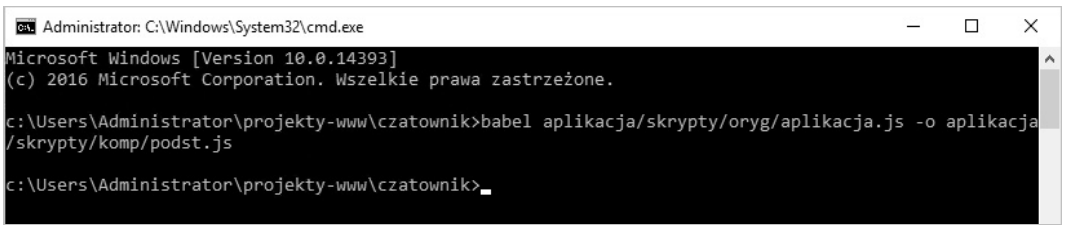
```
class CzatAplikacja {
  constructor() {
    console.log('Oto wersja ES6!');
  }
}
new CzatAplikacja();
```

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*:

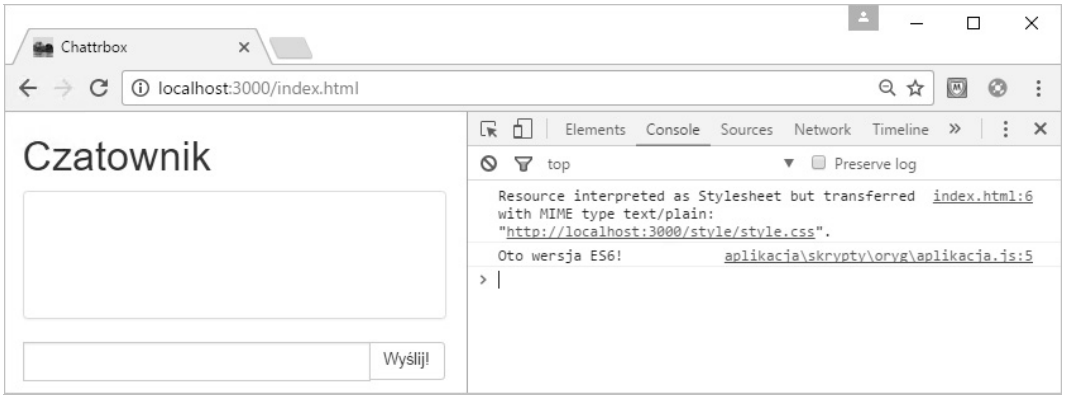
```
babel aplikacja/skrypty/oryg/aplikacja.js -o aplikacja/skrypty/komp/podst.js
```

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 `CzatAplikacja`, więc wywoływany jest jej konstruktor, wyświetlający w konsoli napis `Oto wersja ES6!`.

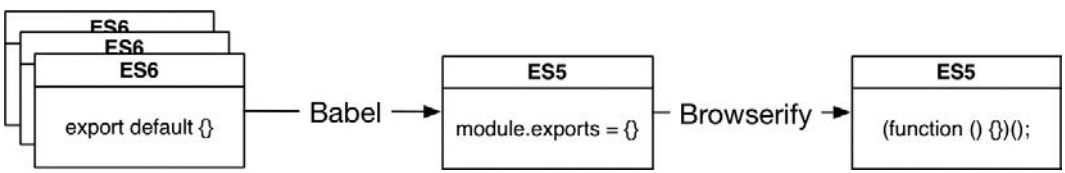
Przekonałeś się zatem, że narzędzie Babel prawidłowo przekłada pojedynczy plik JavaScript, czas więc utworzyć kilka modułów.

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 CzatAplikacja();
export default CzatAplikacja;
```

W ten sposób określisz, że `CzatAplikacja` 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 `CzatAplikacja` i tworzące jej instancję:

```
import CzatAplikacja from './aplikacja';
new CzatAplikacja();
```

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

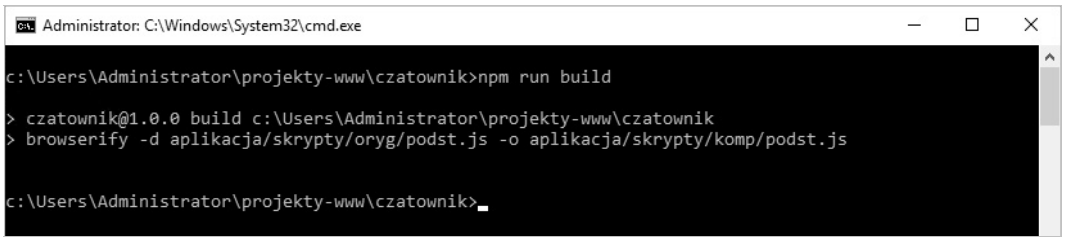
Należy w tym momencie zwrócić uwagę na ważną rzecz: nazwa klasy użyta w pliku *podst.js* może być dowolna. Ponieważ klasa `CzatAplikacja` jest domyślną klasą eksportowaną z pliku *aplikacja.js*, więc można użyć na przykład instrukcji `import MojaAplikacja from './aplikacja'`, która przypisze eksportowaną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

Uruchomienie procesu translacji

Teraz w terminalu wpisz następujące polecenie:

```
npm run build
```

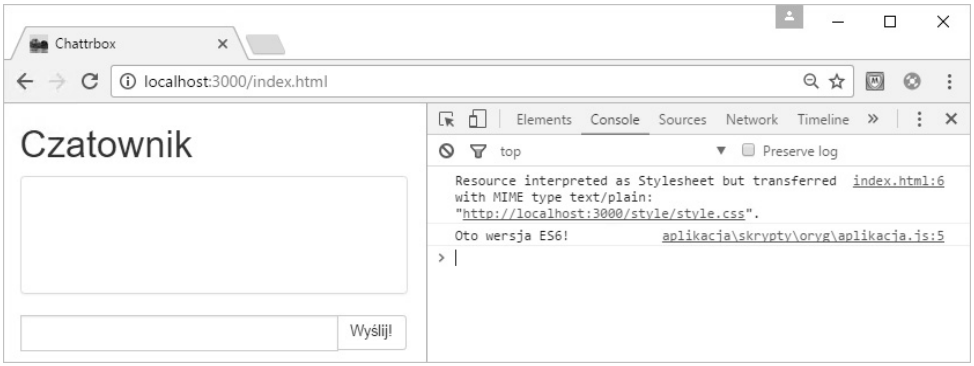
Program `npm` wykona polecenie `build` wywołujące narzędzie Browserify. W terminalu będą pojawiały się informacje o wykonywanych operacjach. Jednak samo narzędzie Browserify nie wyświetli żadnych informacji, chyba że w kodzie będzie błąd (patrz rysunek 17.11).



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 `CzatAplikacja`. W konsoli pojawi się zatem ten sam komunikat co poprzednio (patrz rysunek 17.12).



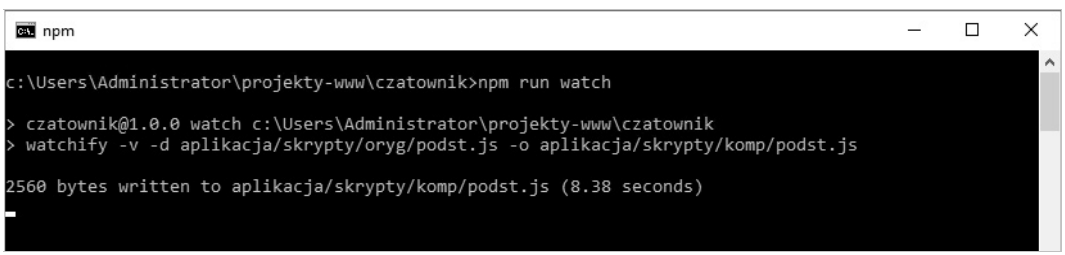
Rysunek 17.12. Oto znów wersja ES6!

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 przesyłać 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 JavaScripcie). 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!",
  "użytkownik": "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, użytkownik, czas) {
    this.komunikat = komunikat;
    this.użytkownik = użytkownik || '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 destrukturyzujące**.

```
...
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,
```

```
    użytkownik: u='Andrzej',
    czas: c=(new Date()).getTime()
  }) {
    this.komunikat = k;
    this.użytkownik = 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 destrukturyzującą można stosować do przypisywania zmiennym wartości. Powyższy konstruktor można również zdefiniować w następujący sposób:

```
...
class CzatKomunikat {
  constructor(dane) {
    var {komunikat: k, użytkownik: u='Andrzej', czas: c=(new Date()).getTime()} = dane;
    this.komunikat = k;
    this.użytkownik = 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,
    użytkownik: u='Andrzej',
    czas: c=(new Date()).getTime()
  }) {
    this.komunikat = k;
    this.użytkownik = 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 modułem aplikacji *Czatownik*.

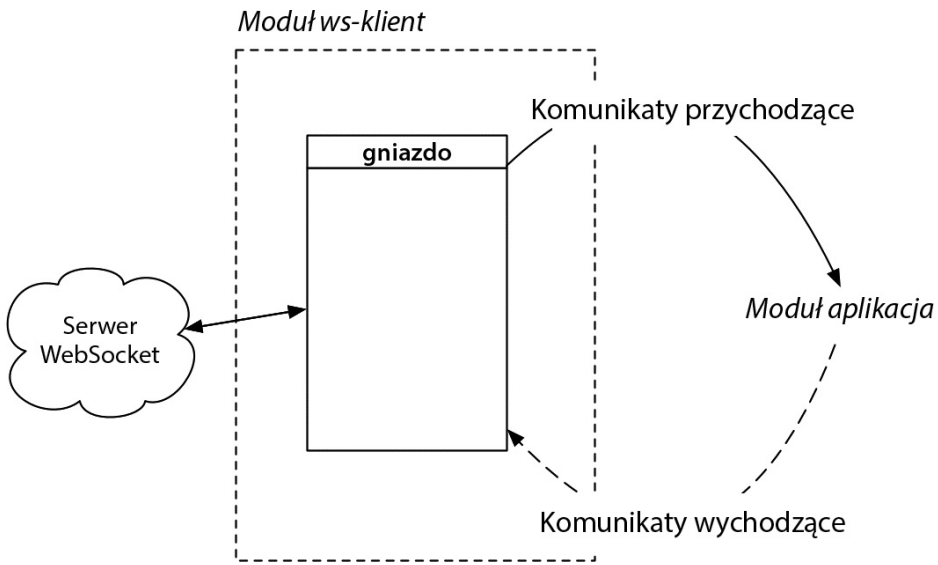
Utworzenie modułu ws-klient

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 modułem 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 literał obiektu** (ang. *enhanced object literal*).

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

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

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

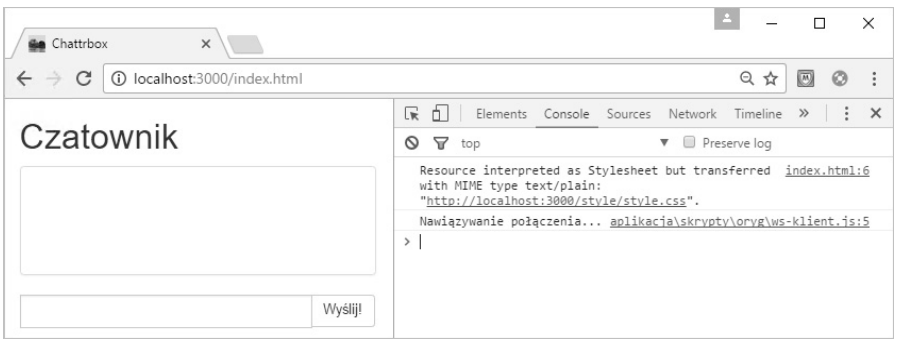
Zostanie w ten sposób utworzony obiekt `gniazdo` wyeksportowany z modułu `ws-klient`.

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 zwykłe 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 wyślijKomunikat(zawartość) {
  gniazdo.send(JSON.stringify(zawartość));
}
...
```

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

```
...
function wyślijKomunikat(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

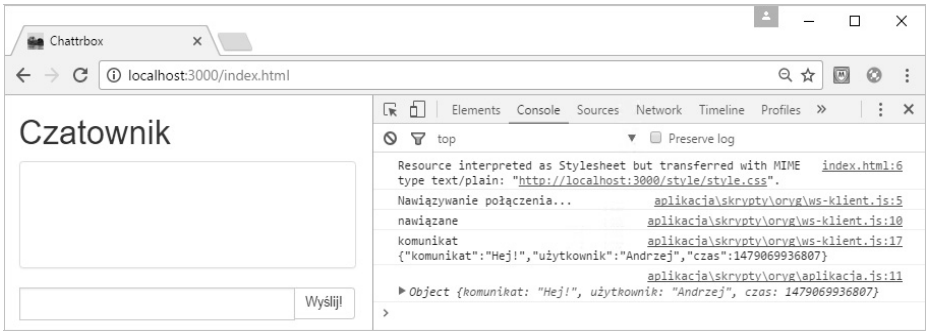
W pliku *aplikacja.js* zmień konstruktora klasy `CzatAplikacja`. Poniżej wywołania metody `gniazdo.inicjuj` wpisz wywołania metod `zarejestrujObsługęOtwarcia` i `zarejestrujObsługęKomunikatu`, umieszczając w ich parametrach funkcję strzałkową:

```
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. Wysłanie 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++: *kripken.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 przeglądarkowych, 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 zwrotna 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, przesuwa 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.');
  } else {
    console.log('Nie trzeba zerować zmiennej.');
  }
  return mojaZmienna;
}
```

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 przesuwa 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 skłamaliśmy. 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

E

F

G

H

I

J

K

L

M

N

R

S

Ś

T

U

W

Z

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 nakręcane 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/nowosci

Big Nerd Ranch

ISBN 978-83-283-3203-4

cena: 69,00 zł