React
dla zaawansowanych

Przekonaj się, jak dobry jest React!

Cássio de Sousa Antonio

Helion

Apress

Tytuł oryginału: Pro React

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-2992-8

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

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Rozpoczęcie pracy

React to biblioteka typu *open source* utworzona przez programistów Facebooka. Prezentuje nowatorskie podejście do tworzenia interfejsów użytkownika za pomocą JavaScriptu i dzięki temu od samego początku istnienia cieszy się dużą popularnością oraz skupia aktywną społeczność użytkowników.

W książce tej znajdziesz wszystkie informacje potrzebne do tego, by móc efektywnie wykorzystać bibliotekę React we własnych projektach. Zważywszy, że React to tylko narzędzie do renderowania interfejsów użytkownika, nie ma żadnych ograniczeń dotyczących pozostałych technologii wykorzystywanych w projekcie. W związku z tym w dalszej części książki opisuję techniki trasowania (routowania) i architektury aplikacji typowe dla tej biblioteki.

W tym rozdziale opisuję kilka ogólnych zagadnień, aby pomóc Ci w jak najszybszym przystąpieniu do budowania aplikacji. Oto lista omówionych tematów:

- definicja biblioteki React i przegląd jej zalet,
- definiowanie interfejsów użytkownika przy użyciu reactowego rozszerzenia składni JavaScriptu o nazwie JSX,
- tworzenie komponentów React z własnościami i stanami.

Zanim zaczniesz

Biblioteka React jest typowym elementem nowoczesnego ekosystemu programistycznego JavaScriptu. Aby móc wypróbowywać przykłady kodu opisywane w tej książce, musisz zainstalować Node.js i npm. Dobrze jest też znać się na programowaniu funkcyjnym w JavaScripcie oraz być na bieżąco z najnowszymi składnikami tego języka, takimi jak funkcje strzałkowe czy klasy.

Node.js i npm

Z założenia JavaScript to język przeglądarkowy, ale system Node.js umożliwia uruchamianie programów w tym języku w komputerach lokalnych i na serwerach za pośrednictwem darmowego narzędzia wiersza poleceń. W połączeniu z **npm** (ang. *Node Package Manager*) system Node.js stał się bezcennym narzędziem do programowania aplikacji JavaScript, umożliwiającym tworzenie skryptów wykonujących zadania (np. kopiowanie i przenoszenie plików albo uruchamianie lokalnego serwera testowego) i automatyczne pobieranie zależności.

Kto jeszcze nie zainstalował Node.js w swoim komputerze, powinien to zrobić teraz, pobierając instalator dla systemu Windows, Mac lub Linux ze strony *https://nodejs.org/*.

Standard ECMAScript 2016

JavaScript to żywy język, który przez lata przeszedł wiele zmian. Niedawno skupiona wokół niego społeczność uzgodniła, że należy dodać pewne nowe funkcje i ulepszenia. Niektóre z nich zostały już zaimplementowane w najnowszych przeglądarkach i są powszechnie wykorzystywane przez programistów posługujących się biblioteką React (należą do nich m.in. funkcje strzałkowe, klasy i operator rozszczepiania — ang. *spread operator*). Ponadto biblioteka React jest tak skonstruowana, że zachęca do posługiwania się funkcyjnymi technikami programowania w JavaScripcie, więc dobrze jest wiedzieć, jak w tym języku działają funkcje i konteksty, oraz znać takie metody jak `map`, `reduce` i `assign`.

Definicja biblioteki React

Oto moja ulubiona definicja biblioteki React, dzięki której chyba najłatwiej jest zrozumieć, z czym się ma do czynienia:

> ***React to silnik do budowania interfejsów użytkownika z gotowych komponentów przy użyciu języka JavaScript i (ewentualnie) XML.***

Przeanalizujemy to zdanie po kawałku:

React to silnik: na stronie internetowej napisano, że React to biblioteka, ale ja wolę słowo „silnik" (ang. *engine*), ponieważ lepiej pasuje do podstawowej zalety tego produktu, czyli sposobu reaktywnego renderowania interfejsów użytkownika. Jest to metoda polegająca na oddzieleniu stanu (wszystkich danych wewnętrznych definiujących aplikację w określonym czasie) od interfejsu prezentowanego użytkownikowi. W React deklaruje się sposób prezentacji stanu w postaci wizualnych elementów drzewa DOM, a następnie drzewo to jest automatycznie aktualizowane w reakcji na zmiany stanu.

Terminu *engine* w odniesieniu do React jako pierwszy użył Justin Deal, któremu sposób renderowania reaktywnego skojarzył się ze sposobem działania silników gier (*https://zapier.com/engineering/react-js-tutorial-guide-gotchas/*).

do budowania interfejsów użytkownika z gotowych komponentów: React istnieje właśnie po to, by ułatwiać tworzenie i obsługę interfejsów użytkownika. Centralnym pojęciem jest komponent interfejsu, czyli samodzielna cegiełka, która jest łatwa w użyciu, rozszerzalna i nie sprawia kłopotów w obsłudze.

przy użyciu języka JavaScript i (ewentualnie) XML: React to biblioteka JavaScript, której można używać w przeglądarce internetowej, na serwerze albo w urządzeniu mobilnym. Jak się przekonasz w dalszej części tego rozdziału, biblioteka ta dysponuje opcjonalną składnią umożliwiającą opisywanie elementów interfejsu użytkownika za pomocą kodu XML. Choć wydaje się to dziwne, język ten okazał się doskonałym narzędziem do tego celu. Zawdzięcza to deklaratywnej naturze, wyraźnie zaznaczonym relacjom między elementami oraz łatwości wizualizowania ogólnej struktury interfejsu.

Zalety biblioteki React

Istnieje wiele popularnych silników renderowania stron w JavaScripcie. Dlaczego więc programiści z Facebooka utworzyli bibliotekę React i dlaczego Ty miałbyś z niej korzystać? Odpowiedź na to pytanie znajduje się w trzech następnych sekcjach, które zawierają opis niektórych zalet tego produktu.

Renderowanie reaktywne jest łatwe

Na początku ery programowania sieciowego, długo przed powstaniem koncepcji aplikacji jednostronicowych, każda interakcja użytkownika ze stroną (np. naciśnięcie przycisku) wymagała przesłania nowej strony przez serwer, nawet jeśli ta nowa strona różniła się od poprzedniej tylko jakimś drobiazgiem. Było to bardzo niewygodne z punktu widzenia użytkownika, mimo że programista mógł z łatwością przewidzieć, co dokładnie zobaczy użytkownik w określonym momencie interakcji.

Aplikacje jednostronicowe na bieżąco pobierają nowe dane i przekształcają elementy struktury DOM w odpowiedzi na działania użytkownika. Kiedy interfejsy stają się coraz bardziej skomplikowane, coraz trudniej jest zapanować nad bieżącym stanem aplikacji i wprowadzać zmiany w DOM na czas.

Jedną z często stosowanych (zwłaszcza przed pojawieniem się biblioteki React) w systemach JavaScriptu technik radzenia sobie z tą rosnącą złożonością i synchronizowania interfejsu ze stanem jest wiązanie danych, choć dotyczą jej pewne problemy związane z obsługą serwisową, skalowalnością i wydajnością.

Od tradycyjnego wiązania danych łatwiejsze jest renderowanie reaktywne, ponieważ programista musi tylko zadeklarować, jak powinny wyglądać i działać poszczególne komponenty. Gdy zmienią się dane, React ponownie renderuje cały interfejs.

Jako że pozbywanie się całego interfejsu i renderowanie go od nowa za każdym razem, gdy zmienią się dane stanowe, jest bardzo niekorzystne z punktu widzenia wydajności, w React wykorzystano lekką, przechowywaną w pamięci reprezentację struktury DOM o nazwie „wirtualne drzewo DOM".

Praca z taką strukturą w pamięci jest szybsza i efektywniejsza niż praca z prawdziwym drzewem. Kiedy zmienia się stan aplikacji (np. w wyniku interakcji użytkownika z aplikacją lub pobrania danych), React szybko porównuje bieżący stan interfejsu użytkownika ze stanem docelowym i oblicza minimalny zbiór zmian w DOM, jaki pozwala osiągnąć cel. Dzięki temu biblioteka React jest bardzo szybka i wydajna. Aplikacje React z łatwością osiągają 60 klatek na sekundę nawet w urządzeniach mobilnych.

Programowanie komponentowe przy użyciu JavaScriptu

W aplikacji React wszystko jest zrobione z komponentów, które są samodzielnymi blokami o ściśle określonym przeznaczeniu. Programowanie aplikacji przy użyciu komponentów pozwala podzielić problem na mniejsze fragmenty, z których żaden nie jest nadmiernie skomplikowany. Wszystkie składniki mają niewielkie rozmiary, a ponieważ można je ze sobą łączyć, w razie potrzeby programista może z mniejszych elementów tworzyć bardziej złożone komponenty.

Komponenty biblioteki React są napisane w JavaScripcie, nie w żadnym języku szablonowym ani HTML-u, które tradycyjnie wykorzystywało się do pisania interfejsów użytkownika aplikacji sieciowych. Decyzja o przyjęciu takiej strategii jest dobrze uzasadniona — szablony stanowią ograniczenie, ponieważ narzucają kompletny zestaw dostępnych abstrakcji. To, że React do renderowania widoków wykorzystuje kompletny język programowania, stanowi wielką zaletę tej biblioteki, jeśli chodzi o budowanie abstrakcji do tworzenia interfejsów.

Dodatkowo samodzielność i jednolity kod znacznikowy z odpowiednią logiką widoku sprawiają, że komponenty React wspomagają podział zadań. Kiedyś poszczególne warstwy pierwszych aplikacji sieciowych rozdzielano przez tworzenie nowych języków, np. HTML-a do opisu struktury stron, CSS do opisu ich stylu oraz JavaScriptu do implementacji zachowań. W czasach gdy wprowadzano te technologie, wszystko działało bardzo dobrze, ponieważ przeważały statyczne strony internetowe. Obecnie jednak interfejsy są nieporównanie bardziej interaktywne i skomplikowane, a logika prezentacji i znaczniki zostały powiązane. W efekcie rozdział kodu znacznikowego, stylów i JavaScriptu stał się rozdziałem technologii, a nie zadań.

W React przyjęto założenie, że logika prezentacji i kod znacznikowy są w znacznym stopniu spójne ze sobą. Jedno i drugie prezentuje interfejs użytkownika i zachęca do oddzielania zadań przez tworzenie dyskretnych, hermetycznych i nadających się do wielokrotnego użytku komponentów o ściśle określonym przeznaczeniu.

Elastyczna abstrakcja modelu dokumentu

React posiada własną lekką reprezentację interfejsu użytkownika, która stanowi abstrakcję modelu dokumentu. Największą zaletą tej reprezentacji jest to, że umożliwia zarówno renderowanie kodu HTML dla internetu, jak i renderowanie macierzystych widoków iOS i Android według tych samych zasad. Ponadto abstrakcja ta ma jeszcze dwie inne ciekawe cechy:

- Zdarzenia zachowują się w jednolity, standardowy sposób we wszystkich przeglądarkach i urządzeniach, automatycznie korzystając z techniki delegacji.
- Komponenty React mogą być renderowane na serwerze ze względu na kwestie SEO i walory użytkowe.

Budowa pierwszej aplikacji React

Wiesz już, że podstawowymi blokami do budowy aplikacji React są komponenty, ale jak one wyglądają? Jak się je tworzy? Najprostszy komponent React to po prostu klasa JavaScript z metodą `render`, która zwraca opis interfejsu użytkownika tego komponentu, np.:

```
class Hello extends React.Component {
  render() {
    return (
      <h1>Witaj, świecie</h1>
    )
  }
}
```

Nie da się nie zauważyć kodu HTML w klasie JavaScript. Jak już pisałem, React posiada rozszerzenie składni języka JavaScript o nazwie JSX, które umożliwia posługiwanie się kodem XML (i HTML) wewnątrz normalnego kodu JavaScript.

Posługiwanie się rozszerzeniem JSX nie jest obowiązkowe, ale przyjęło się już jako standardowa metoda definiowania interfejsów użytkownika w komponentach React, ponieważ ma deklaratywną i ekspresyjną składnię, a jego kod jest tłumaczony na zwykłe wywołania funkcji, czyli nie zmienia semantyki języka.

Dokładniejszy opis języka JSX zamieściłem w następnym rozdziale, a na razie wystarczy zapamiętać, że React wymaga przeprowadzenia etapu „transformacji" (albo transpilacji, jak wolą niektórzy), w ramach którego kod JSX jest zamieniany na JavaScript.

W nowoczesnym programowaniu w języku JavaScript wykorzystuje się wiele narzędzi, które są pomocne w przeprowadzaniu tego procesu. Przyjrzymy się zatem, jak wygląda proces tworzenia projektu React.

Proces tworzenia projektu React

Dawno minęły czasy, kiedy cały kod JavaScript pisało się w jednym pliku, ręcznie pobierało się jedną czy dwie biblioteki JavaScript oraz łączyło wszystko na jednej stronie. I choć oczywiście można pobrać, a nawet skopiować bibliotekę React ze zminimalizowanego pliku JavaScript, aby od razu zacząć uruchamiać komponenty i przekształcać kod JSX na bieżąco, to nikt tego nie robi, chyba że w celu utworzenia niewielkiego dema albo prototypu.

Nawet w najprostszych przypadkach dobrze jest korzystać z kompletnego procesu programistycznego, który umożliwia:

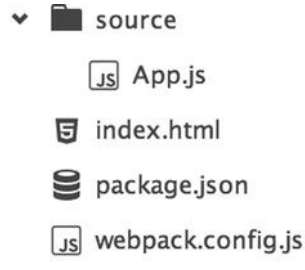
- pisanie kodu JSX i przekształcanie go na bieżąco w zwykły kod JavaScript,
- pisanie modułów kodu,

- zarządzanie zależnościami,
- scalanie modułów JavaScript i diagnozowanie skryptów przy użyciu map źródeł.

W związku z tym projekt React o podstawowej strukturze zawiera następujące elementy:

1. **Folder źródłowy** mieszczący wszystkie moduły JavaScript.
2. **Plik** *index.html*. W aplikacjach React strona HTML jest prawie pusta, ponieważ jej zadaniem jest ładowanie skryptów i dostarczanie elementu `div` (lub innego), w którym biblioteka React będzie renderować komponenty.
3. **Plik** *package.json*. Jest to standardowy manifest npm, w którym zawarte są różne informacje o projekcie, takie jak nazwa, opis czy dane autora. W pliku tym można określić zależności (które zostaną automatycznie pobrane i zainstalowane) oraz zdefiniować zadania skryptu.
4. **Pakowacz modułów lub narzędzie kompilacyjne** do transformacji kodu JSX i pakowania modułów oraz zależności. Moduły pomagają w organizacji kodu JavaScript przez jego podzielenie na wiele plików, z których każdy ma zadeklarowany własny zestaw zależności. Specjalne narzędzie automatycznie pakuje wszystko w jeden pakiet w odpowiedniej kolejności. Jest wiele narzędzi tego rodzaju, np. Grunt, Gulp i Brunch. W każdym z nich można znaleźć recepty dotyczące pracy z React, ale generalnie w społeczności przyjęło się, że preferowanym narzędziem do tych zadań jest webpack. Zasadniczo webpack to narzędzie do pakowania modułów, ale dodatkowo może przepuszczać kod przez loadery, które mogą go przekształcać i kompilować.

Na rysunku 1.1 pokazano opisaną strukturę plików i folderów.



Rysunek 1.1. *Minimalna struktura folderów i plików projektu React*

Szybki start

Dla ułatwienia koncentracji na samej bibliotece React do książki tej dołączony jest podstawowy zestaw plików (*react-app-boilerplate*), który można pobrać z serwera FTP wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/reactz.zip*. Ten szablonowy projekt zawiera wszystkie podstawowe pliki i ustawienia konfiguracyjne potrzebne do natychmiastowego rozpoczęcia pracy. Po pobraniu na dysk i rozpakowaniu archiwum wystarczy zainstalować zależności i uruchomić serwer testowy, aby przetestować projekt w przeglądarce. Aby automatycznie zainstalować wszystkie zależności, uruchom terminal lub wiersz poleceń i wykonaj polecenie `npm install`. Aby uruchomić serwer testowy, wykonaj z kolei polecenie `npm start`.

Wszystko gotowe do pracy. Jeśli chcesz, możesz pominąć następną sekcję i przejść od razu do budowania pierwszego komponentu React.

Albo zrób to sam

Jeśli nie boisz się ubrudzić rąk, możesz ręcznie stworzyć podstawową strukturę projektu w pięciu prostych krokach:

1. Najpierw utwórz folder źródłowy (najczęściej nadaje się mu nazwę *source* albo *app*). W folderze tym będziesz przechowywać tylko moduły JavaScript. Natomiast statyczne zasoby, które nie przechodzą przez pakowacz modułów (np. plik *index.html*, obrazy i pliki CSS), umieszcza się w folderze głównym.
2. W folderze głównym projektu utwórz plik *index.html* i wpisz w nim kod pokazany na listingu 1.1.

Listing 1.1. *Prosta strona HTML ładująca należący do paczki kod JavaScript i zawierająca główny element div, w którym będą renderowane komponenty React*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pierwszy komponent React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/javascript" src="bundle.js"></script>
  </body>
</html>
```

3. Utwórz plik *package.json*, wykonując polecenie `npm init` w terminalu lub wierszu poleceń i postępując zgodnie z poniższymi instrukcjami. Narzędzia npm będziesz używać do zarządzania zależnościami (pobierania i instalowania wszystkich potrzebnych bibliotek). Wśród zależności do tego projektu znajdują się biblioteka React, kompilator Babel do przekształcania kodu JSX (loader i core) oraz narzędzie webpack (wraz z serwerem *webpack dev server*). Zmień zawartość pliku *package.json* tak, aby wyglądała jak na listingu 1.2, i wykonaj polecenie `npm install`.

Listing 1.2. *Zależności w przykładowym pliku package.json*

```
{
  "name": "nazwa-aplikacji",
  "version": "X.X.X",
  "description": "Opis aplikacji",
  "author": "Ty",
  "devDependencies": {
    "babel-core": "^5.8.*",
    "babel-loader": "^5.3.*",
    "webpack": "^1.12.*",
    "webpack-dev-server": "^1.10.*"
  },
  "dependencies": {
    "react": "^0.13.*"
  }
}
```

4. Następną czynnością jest konfiguracja narzędzia webpack, czyli najchętniej wybieranego pakowacza modułów. Na listingu 1.3 pokazano zawartość pliku konfiguracyjnego. Przyjrzymy się jej. Klucz `entry` wskazuje główny moduł aplikacji.

Listing 1.3. *Zawartość pliku webpack.config*

```
module.exports = {
  entry: [
    './source/App.js'
  ],
  output: {
    path: __dirname,
    filename: "bundle.js"
  },
  module: {
    loaders: [{
      test: /\.jsx?$/,
      loader: 'babel'
    }]
  }
};
```

Następny klucz, `output`, wskazuje, gdzie ma zostać zapisany pojedynczy plik JavaScript zawierający wszystkie moduły spakowane w odpowiedniej kolejności.

Natomiast w sekcji `module loaders` wszystkie pliki *.js* są przekazywane przez kompilator Babel, który przekształca kod JSX na JavaScript. Wiedz jednak, że Babel robi coś więcej niż tylko przeprowadzanie kompilacji — umożliwia posługiwanie się najnowszymi dodatkami do JavaScriptu, takimi jak funkcje strzałkowe i klasy.

5. Pozostało już tylko kilka czynności końcowych. Struktura projektu jest gotowa, trzeba więc uruchomić serwer lokalny (który będzie potrzebny do testowania aplikacji w przeglądarce). Służy do tego polecenie `node_modules/.bin/webpack-dev-server`, ale jeśli nie chcesz za każdym razem wpisywać tego skomplikowanego ciągu znaków od nowa, możesz zamienić go na zadanie w utworzonym w punkcie 3. pliku *package.json* (listing 1.4).

Listing 1.4. *Dodanie skryptu startowego do pliku package.json*

```
{
  "name": "nazwa-aplikacji",
  "version": "X.X.X",
  "description": "Opis aplikacji",
  "author": "Ty",
  "scripts": {
    "start": "node_modules/.bin/webpack-dev-server --progress"
  },
  "devDependencies": {
    "babel-core": "^5.8.*",
    "babel-loader": "^5.3.*",
    "webpack": "^1.12.*",
    "webpack-dev-server": "^1.10.*"
  },
  "dependencies": {
    "react": "^0.13.*"
  }
}
```

Od teraz lokalny serwer testowy można uruchamiać za pomocą polecenia `npm start`.

Tworzenie pierwszego komponentu

Po utworzeniu podstawowej struktury projektu ze skonfigurowanym zarządcą zależnościami, systemem modułowym i transformacjami JSX możemy przejść do komponentu `Witaj, świecie` i wyrenderować go na stronie. Kod samego komponentu pozostanie bez zmian, ale dodamy instrukcję `import` ładującą bibliotekę React do paczki JavaScriptu.

```
import React from 'react';

class Hello extends React.Component {
  render() {
    return (
      <h1>Witaj, świecie</h1>
    );
  }
}
```

Teraz możemy wyświetlić nasz komponent na stronie za pomocą metody `React.render`, jak widać poniżej i na rysunku 1.2:

```
React.render(<Hello />, document.getElementById('root'));
```

Rysunek 1.2. *Pierwszy komponent wyrenderowany na stronie*

■ **Wskazówka** Choć można też renderować bezpośrednio w elemencie `body`, zwykle lepszym pomysłem jest robienie tego w jakimś elemencie podrzędnym (najczęściej `div`). Do głównego elementu dokumentu dołącza węzły wiele bibliotek i rozszerzeń przeglądarek, co może sprawiać problemy ze względu na to, że React musi mieć pełną kontrolę nad wycinkiem drzewa DOM.

Jak oszczędzić sobie pisania

Nielubiący za dużo pisać programiści często ułatwiają sobie życie przez zastosowanie w instrukcjach importu tzw. przypisania destrukturyzującego (ang. *destructuring assignment*) umożliwiającego bezpośrednie odnoszenie się do funkcji i klas znajdujących się w modułach. W przypadku naszego programu moglibyśmy na przykład uniknąć konieczności pisania `React.Component`:

```
import React, { Component } from 'react';

class Hello extends Component {
  render() {
    return (
      <h1>Witaj, świecie</h1>
    );
  }
}
```

W tym przykładzie korzyści są oczywiście niewielkie, ale w większych projektach na naszą korzyść działa efekt skali.

■ **Uwaga** Przypisanie destrukturyzujące to element najnowszej wersji języka JavaScript. Jego szczegółowy opis znajdziesz na stronie *http://shebang.pl/artykuly/destrukturyzacja*.

Wartości dynamiczne

Wartości zapisane w JSX w klamrach ({}) są traktowane jako wyrażenia JavaScript i renderowane wśród znaczników. Gdybyśmy na przykład chcieli wyrenderować wartość zmiennej lokalnej, moglibyśmy to zrobić tak:

```
import React, { Component } from 'react';

class Hello extends Component {
  render() {
    var place = "świecie";
    return (
      <h1>Witaj, {place}</h1>
    );
  }
}

React.render(<Hello />, document.getElementById("root"));
```

Komponowanie komponentów

W React z zasady powinno się tworzyć niewielkie komponenty wielokrotnego użytku, które można łączyć w bardziej złożone elementy interfejsu użytkownika. Znasz już podstawową strukturę komponentu React, więc czas nauczyć się tworzyć kompozycje tych składników.

Własności

Kluczowym aspektem pozwalającym na wielokrotne wykorzystywanie i komponowanie komponentów w React jest możliwość ich konfigurowania przy użyciu własności (ang. *props* lub *properties*). Służą one do przekazywania danych od komponentu nadrzędnego do podrzędnego, ale w elemencie podrzędnym nie można nic w nich zmieniać, ponieważ ich „właścicielem" jest komponent nadrzędny.

W JSX własności przekazuje się, podobnie jak w języku HTML, w atrybutach znaczników. W ramach przykładu utworzymy prostą listę zakupów złożoną z dwóch komponentów. Nadrzędny będzie nazywał się `GroceryList`, a podrzędny — `GroceryItem`:

```
import React, { Component } from 'react';

// komponent nadrzędny
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity="1" name="Chleb" />
        <ListItem quantity="6" name="Jaja" />
        <ListItem quantity="2" name="Mleko" />
      </ul>
    );
  }
}

// komponent podrzędny
class ListItem extends Component {
  render() {
    return (
      <li>
        {this.props.quantity}× {this.props.name}
      </li>
    );
  }
}

React.render(<GroceryList />,document.getElementById("root"));
```

Ponadto za pomocą własności `props.children` można też pobierać treść znajdującą się między znacznikami elementu:

```
import React, { Component } from 'react';

// komponent nadrzędny
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity="1">Chleb</ListItem>
        <ListItem quantity="6">Jaja</ListItem>
        <ListItem quantity="2">Mleko</ListItem>
      </ul>
    );
  }
}
```

```
// komponent potomny
class ListItem extends Component {
  render() {
    return (
      <li>
        {this.props.quantity}× {this.props.children}
      </li>
    );
  }
}

React.render(<GroceryList />, document.getElementById('root'));
```

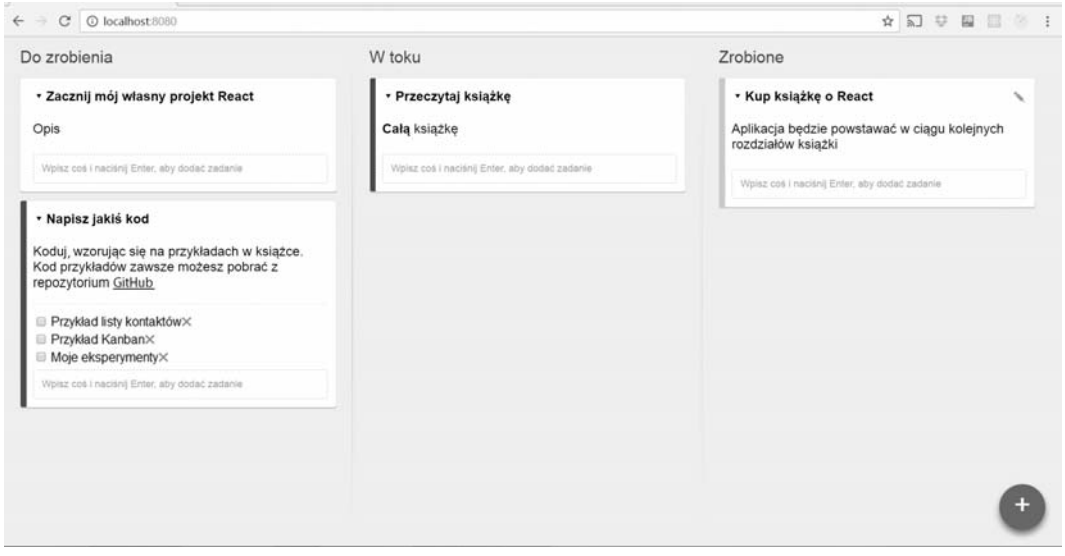
Aplikacja w stylu tablicy kanban

Każdy omawiany w tej książce temat jest opatrzony ilustracją w postaci kilku niewielkich komponentów i przykładowego kodu. Oprócz tego zbudujemy też jedną kompletną aplikację, którą będzie narzędzie do zarządzania projektami w stylu *kanban*.

Na tablicy *kanban* zadania są reprezentowane przez karteczki (rysunek 1.3). Układa się je w listy posortowane według postępu realizacji i przekłada do kolejnych list w miarę wykonywania, tak że powstaje wizualna reprezentacja procesu od pomysłu do wdrożenia.

Rysunek 1.3. *Przykładowa tablica kanban*

W internecie można znaleźć wiele aplikacji do zarządzania projektami w tym stylu. Jednym z najbardziej znanych przykładów jest Trello.com, ale nasz projekt będzie prostszy. Na rysunku 1.4 widać, jak będzie wyglądał po ukończeniu, a na listingu 1.5 pokazano model danych, który wykorzystamy w naszej aplikacji.



Rysunek 1.4. *Aplikacja Kanban, którą będziemy budować przez kilka kolejnych rozdziałów*

Listing 1.5. *Model danych aplikacji Kanban*

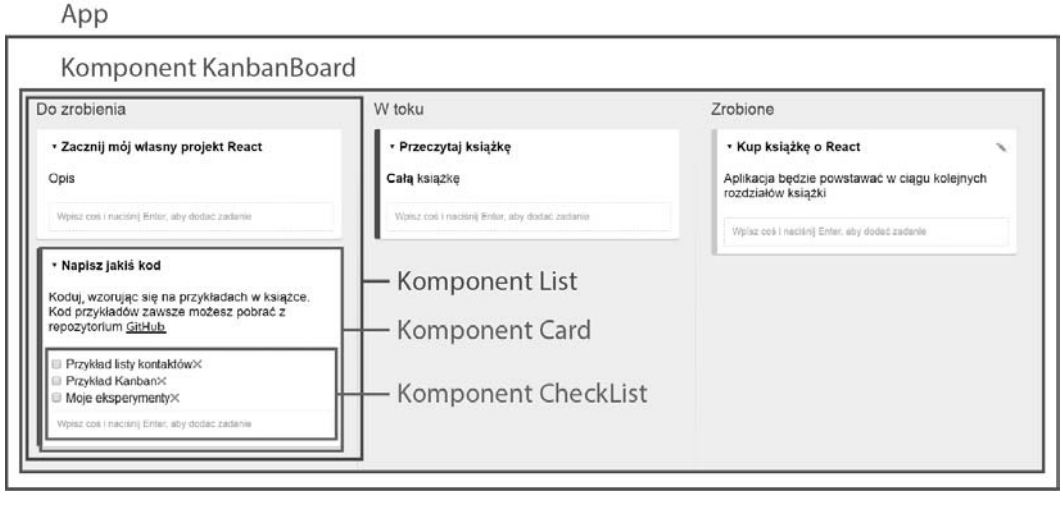
```
[
  { id:1,
    title: "Tytuł pierwszej kartki",
    description: "Szczegółowy opis kartki",
    status: "todo",
    tasks: [
      {id: 1, name:"Zadanie pierwsze", done:true},
      {id: 2, name:"Zadanie drugie", done:false},
      {id: 3, name:"Zadanie trzecie", done:false}
    ]
  },
  { id:2,
    title: "Tytuł drugiej kartki",
    description: "Szczegółowy opis kartki",
    status: "in-progress",
    tasks: []
  },
  { id:3,
    title: "Tytuł trzeciej kartki",
    description: "Szczegółowy opis kartki",
    status: "done",
    tasks: []
  },
];
```

Definiowanie hierarchii komponentów

Umiejętność podziału interfejsu na zagnieżdżone komponenty to podstawa. Poniżej znajduje się lista trzech rzeczy, które zawsze należy rozważyć.

1. Pamiętaj, że komponenty powinny być niewielkie i skoncentrowane na jednym zadaniu. Innymi słowy, idealny komponent robi tylko jedną rzecz, ale dobrze. Jeśli się rozrasta, to znaczy, że trzeba go podzielić na mniejsze części.
2. Dokładnie przeanalizuj strukturę elementów i układ projektu, aby odkryć wiele wskazówek na temat tego, jak powinna wyglądać hierarchia komponentów.
3. Przyjrzyj się modelowi danych. Interfejsy i modele danych często wykorzystują tę samą architekturę informacji, dzięki czemu podzielenie interfejsu użytkownika na komponenty często jest banalnie łatwe. Wystarczy utworzyć komponenty wprost reprezentujące poszczególne elementy modelu danych.

Jeśli zastosujemy się do tych wskazówek w odniesieniu do naszej aplikacji Kanban, otrzymamy kompozycję pokazaną na rysunku 1.5.



Rysunek 1.5. *Hierarchia komponentów w aplikacji Kanban*

Znaczenie własności

Własności mają kluczowe znaczenie przy tworzeniu kompozycji komponentów. Służą do przekazywania danych od komponentu nadrzędnego do podrzędnego. Wewnątrz komponentu nie można ich zmieniać oraz są przekazywane przez komponent nadrzędny, który jest ich „właścicielem”.

Budowanie komponentów

Po określeniu hierarchii elementów interfejsu można zacząć budować komponenty. Robi się to na dwa sposoby: od góry lub od dołu, tzn. najpierw można zbudować komponenty z wyższych (np. `App`) lub niższych (np. `CheckList`) poziomów hierarchii. Aby poznać proces przekazywania własności w dół i dowiedzieć się, jak są używane w komponentach potomnych, zaczniemy budować nasze komponenty *kanban* metodą od góry.

Ponadto, aby utrzymać porządek w projekcie i ułatwić sobie jego obsługę serwisową oraz implementację nowych funkcji, każdy komponent zapiszemy w osobnym pliku JavaScript.

Moduł App (App.js)

Na razie zawartość pliku *App.js* będzie bardzo prosta. Zapiszemy w nim tylko trochę danych oraz wykorzystamy go do renderowania komponentu `KanbanBoard`. W pierwszej wersji naszej aplikacji dane będą wpisane na stałe do zmiennej lokalnej, ale w dalszych rozdziałach będziemy je pobierać z interfejsu API. Spójrz na listing 1.6.

Listing 1.6. *Prosty plik App.js*

```
import React from 'react';
import KanbanBoard from './KanbanBoard';

let cardsList = [
  {
    id: 1,
    title: "Przeczytać książkę",
    description: "Muszę przeczytać całą książkę",
    status: "in-progress",
    tasks: []
  },
  {
    id: 2,
    title: "Napisać trochę kodu",
    description: "Będę przepisywać kod przykładów z książki",
    status: "todo",
    tasks: [
      {
        id: 1,
        name: "Przykład listy kontaktów",
        done: true
      },
      {
        id: 2,
        name: "Przykład Kanban",
        done: false
      },
      {
        id: 3,
        name: "Moje eksperymenty",
        done: false
      }
    ]
  },
];

React.render(<KanbanBoard cards={cardsList} />, document.getElementById('root'));
```

Komponent KanbanBoard (KanbanBoard.js)

Komponent `KanbanBoard` będzie odbierał dane z własności i za pomocą filtrowania zadań według stanu realizacji będzie tworzył trzy komponenty listy: *Do zrobienia*, *W toku* i *Zrobione*. Spójrz na listing 1.7.

Listing 1.7. *Komponent KanbanBoard*

```
import React, { Component } from 'react';
import List from './List';

class KanbanBoard extends Component {
render(){
    return (
      <div className="app">

        <List id='todo' title="Do zrobienia" cards={
          this.props.cards.filter((card) => card.status === "todo")
        } />

        <List id='in-progress' title="W toku" cards={
          this.props.cards.filter((card) => card.status === "in-progress")
        } />

        <List id='done' title='Zrobione' cards={
          this.props.cards.filter((card) => card.status === "done")
         } />

      </div>
    );
  }
}
export default KanbanBoard;
```

■ **Uwaga** Jak wspomniałem na początku tego rozdziału, komponenty React pisze się w języku JavaScript. Nie dysponują one specjalną składnią dla pętli i rozgałęzień w przeciwieństwie do niektórych bibliotek szablonowych, choćby Moustache, ale to wcale nie jest taka zła wiadomość, ponieważ w zamian mamy do dyspozycji prawdziwy kompletny język programowania.

Komponent List (List.js)

Zadaniem komponentu `List` jest wyświetlanie nazwy listy i renderowanie w niej wszystkich komponentów kart. Zwróć uwagę, że mapowaną tablicę kart odbieramy za pośrednictwem własności. Podobnie poprzez własności do komponentu karty przekazujemy indywidualne informacje, takie jak tytuł i opis. Spójrz na listing 1.8.

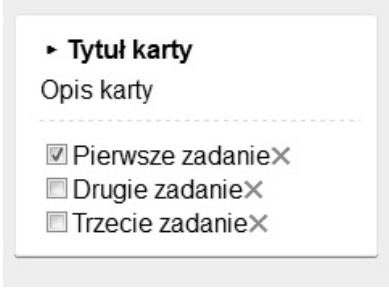
Listing 1.8. *Komponent List*

```
import React, { Component } from 'react';
import Card from './Card';

class List extends Component {
  render() {
    var cards = this.props.cards.map((card) => {
      return <Card id={card.id}
                   title={card.title}
                   description={card.description}
                   tasks={card.tasks} />
    });
```

```
    return (
      <div className="list">
        <h1>{this.props.title}</h1>
         {cards}
      </div>
    );
  }
}
export default List;
```

Komponent Card (Card.js)

Komponent `Card` to ten, z którym użytkownik najczęściej będzie wchodził w interakcje. Każda karta ma tytuł, opis i listę kontrolną, jak widać na rysunku 1.6 i listingu 1.9.



Rysunek 1.6. *Karta aplikacji Kanban*

Listing 1.9. *Komponent Card*

```
import React, { Component } from 'react';
import CheckList from './CheckList';

class Card extends Component {
  render() {
    return (
      <div className="card">
        <div className="card__title">{this.props.title}</div>
        <div className="card__details">
          {this.props.description}
          <CheckList cardId={this.props.id} tasks={this.props.tasks} />
        </div>
      </div>
    );
  }
}

export default Card;
```

Zwróć uwagę na użycie atrybutu `className` w tym komponencie. Jako że JSX to JavaScript, w nazwach atrybutów XML lepiej nie używać takich identyfikatorów jak `class` — i dlatego właśnie użyto nazwy `className`. Temat ten rozwijam jeszcze w następnym rozdziale.

Komponent CheckList (CheckList.js)

Ostatni komponent reprezentuje dolną część karty, czyli listę kontrolną. Przydałby się też formularz do tworzenia nowych zadań, ale zajmiemy się nim w dalszej części rozdziału. Spójrz na listing 1.10.

Listing 1.10. *Komponent CheckList*

```
import React, { Component } from 'react';

class CheckList extends Component {
  render() {
    let tasks = this.props.tasks.map((task) => (
      <li className="checklist__task">
        <input type="checkbox" defaultChecked={task.done} />
        {task.name}
        <a href="#" className="checklist__task--remove" />
      </li>
    ));
    return (
      <div className="checklist">
        <ul>{tasks}</ul>
      </div>
    );
  }
}

export default CheckList;
```

Wykończenie

Komponenty React są już gotowe. Dodamy kilka reguł CSS, aby nadać elementom interfejsu atrakcyjny wygląd (listing 1.11). Pamiętaj, aby utworzyć plik HTML do ładowania plików JavaScript i CSS oraz aby zdefiniować w nim element `div`, w którym React będzie renderować komponenty (przykład takiego pliku przedstawiłem na listingu 1.12).

Listing 1.11. *Plik CSS*

```
*{
  box-sizing: border-box;
}

html,body,#app {
  height:100%;
  margin: 0;
  padding: 0;
}

body {
  background: #eee;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}

h1{
  font-weight: 200;
  color: #3b414c;
  font-size: 20px;
```

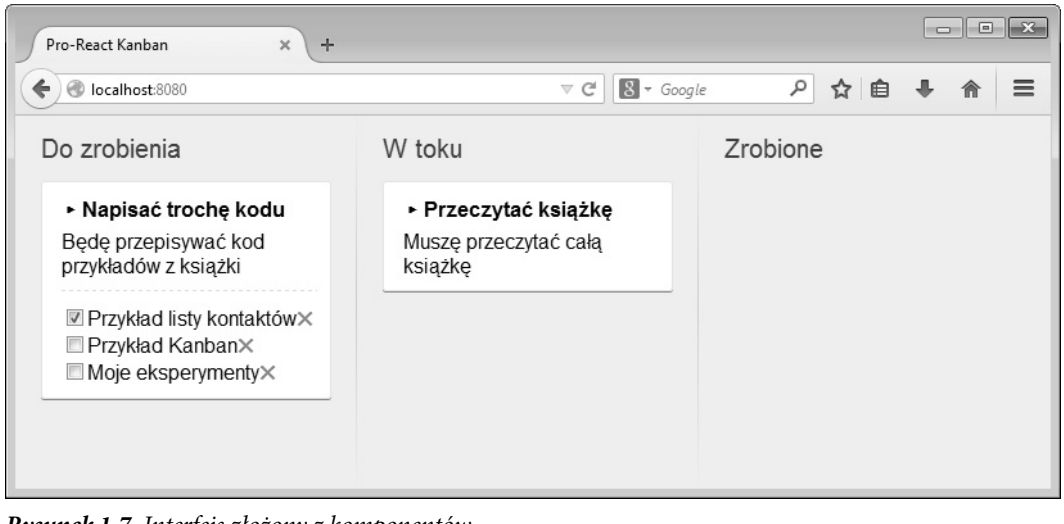
```
}

ul {
  list-style-type: none;
  padding: 0;
  margin: 0;
}
.app {
  white-space: nowrap;
  height:100%;
}

.list {
  position: relative;
  display: inline-block;
  vertical-align: top;
  white-space: normal;
  height: 100%;
  width: 33%;
  padding: 0 20px;
  overflow: auto;
}

.list:not(:last-child):after{
  content: "";
  position: absolute;
  top: 0;
  right: 0;
  width: 1px;
  height: 99%;
  background: linear-gradient(to bottom, #eee 0%, #ccc 50%, #eee 100%) fixed;
}

.card {
  position: relative;
  z-index: 1;
  background: #fff;
  width: 100%;
  padding: 10px 10px 10px 15px;
  margin: 0 0 10px 0;
  overflow: auto;
  border: 1px solid #e5e5df;
  border-radius: 3px;
  box-shadow: 0 1px 0 rgba(0, 0, 0, 0.25);
}

.card__title {
  font-weight: bold;
  border-bottom: solid 5px transparent;
}

.card__title:before {
  display: inline-block;
  width: 1em;
  content: '▸';
}
```

```
.card__title--is-open:before {
  content: '▾';
}

.checklist__task:first-child {
  margin-top: 10px;
  padding-top: 10px;
  border-top: dashed 1px #ddd;
}

.checklist__task--remove:after{
  display: inline-block;
  color: #d66;
  content: "+";
}
```

Listing 1.12. *Plik HTML*

```
<!DOCTYPE html>
<html>
<head>
  <title>Aplikacja Kanban</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div id="root"></div>
  <script type="text/javascript" src="bundle.js"></script>
</body>
</html>
```

Jeśli robiłeś wszystko zgodnie ze wskazówkami, Twoja aplikacja powinna wyglądać tak jak na rysunku 1.7.



Rysunek 1.7. *Interfejs złożony z komponentów*

Podstawowe informacje o stanie

Na razie wiesz, że własności można pobierać i że nie można ich zmieniać. W efekcie tworzone przy ich użyciu komponenty są statyczne. Jeśli komponent ma reagować na działania użytkownika i coś robić, musi zawierać zmienne dane reprezentujące jego stan. Komponenty React mogą przechowywać takie informacje we własności `this.state`. Jest ona prywatną własnością komponentu i można ją zmieniać za pomocą metody `this.setState()`.

Oto ważna cecha komponentów React: kiedy aktualizowany jest stan, komponent włącza renderowanie reaktywne, co powoduje ponowne wyrenderowanie tego komponentu i jego komponentów podrzędnych. Jak już wspominałem, dzięki wirtualnemu drzewu DOM dzieje się to bardzo szybko.

Aplikacja Kanban — otwieranie i zamykanie kart

W ramach prezentacji stanów w komponentach dodam nową funkcję do naszej aplikacji. Będzie to możliwość otwierania i zamykania kart, tak aby użytkownik mógł pokazać lub ukryć zawartość wybranej karty.

Nowy stan można ustawić w dowolnym czasie, ale jeśli chcemy zdefiniować stan początkowy, możemy to zrobić w konstruktorze klasy. Obecnie komponent `Card` nie ma konstruktora, a jedynie metodę `render`. Dodamy więc konstruktor definiujący klucz `showDetails` w stanie komponentu (instrukcje importu i eksportu oraz zawartość metody `render` opuściłem dla uproszczenia). Spójrz na listing 1.13.

Listing 1.13. *Karty z możliwością otwierania i zamykania*

```
class Card extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      showDetails: false
    };
  }

  render() {
    return ( ... );
  }
}
```

W ramach tych modyfikacji zmieniamy kod JSX w metodzie `render` tak, aby renderowała zawartość karty tylko wtedy, gdy własność `showDetails` ma wartość `true`. W tym celu deklarujemy zmienną lokalną o nazwie `cardDetails` i przypisujemy jej dane tylko wtedy, gdy stan `showDetails` jest ustawiony na `true`. W instrukcji `return` po prostu zwracamy wartość zmiennej `cardDetails` (która w przypadku, gdy `showDetails` będzie mieć wartość `false`, będzie pusta). Spójrz na listing 1.14.

Listing 1.14. *Metoda render komponentu Card*

```
render() {
  let cardDetails;
  if (this.state.showDetails) {
    cardDetails = (
      <div className="card__details">
        {this.props.description}
        <CheckList cardId={this.props.id} tasks={this.props.tasks} />
```

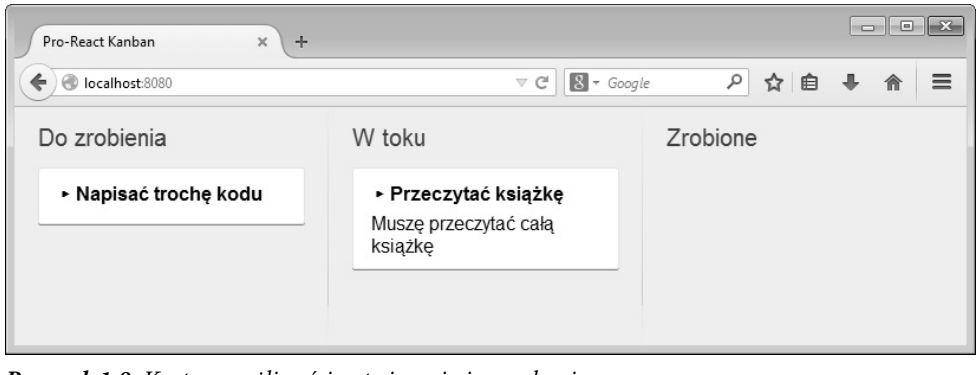
```
      </div>
    );
  };
  return (
    <div className="card">
      <div className="card__title">{this.props.title}</div>
      {cardDetails}
    </div>
  );
}
```

Na koniec dodamy procedurę obsługi kliknięć zmieniającą stan wewnętrzny. Do zmieniania logicznej wartości własności `showDetails` użyjemy operatora JavaScript `!` (nie) (jeśli aktualnie własność ta ma wartość `true`, to zmieni się na `false` i odwrotnie), jak widać na listingu 1.15.

***Listing 1.15**. Procedura obsługi kliknięć*

```
render() {
  let cardDetails;
    if (this.state.showDetails) {
      cardDetails = (
        <div className="card__details">
          {this.props.description}
          <CheckList cardId={this.props.id} tasks={this.props.tasks} />
        </div>
      );
    };

  return (
    <div className="card">
      <div className="card__title" onClick={
        ()=>this.setState({showDetails: !this.state.showDetails})
      }>{this.props.title}</div>
      {cardDetails}
    </div>
  );
}
```

Jeśli teraz uruchomisz aplikację w przeglądarce, wszystkie karty będą zamknięte i będzie można je otworzyć kliknięciem myszą (rysunek 1.8).



***Rysunek 1.8.** Karty z możliwością otwierania i zamykania*

Podsumowanie

W rozdziale tym przedstawiłem podstawowe informacje na temat biblioteki React, opisałem jej zalety (z których najważniejsze to szybkość działania i deklaracyjny sposób definiowania struktury interfejsu aplikacji za pomocą komponentów). Ponadto pokazałem Ci, jak utworzyć pierwszy komponent i opisałem wszystkie podstawowe pojęcia dotyczące komponentów React: metodę `render`, język JSX, własności i stan.

Skorowidz

R

S

T

U

W

Z

React jest biblioteką języka JavaScript, utworzoną i udostępnianą przez Facebook na licencji open source. To narzędzie pozwala na rozwiązywanie częstych, uciążliwych problemów programistycznych w zaskakująco prosty sposób, ponieważ umożliwia tworzenie interfejsów użytkownika z gotowych komponentów. Kiedy tylko się pojawiło, bardzo szybko zyskało szerokie uznanie i skupiło wokół siebie aktywną społeczność.

Jeśli posiadasz już pewne doświadczenie jako programista front end i używasz jQuery lub innego komponentu JavaScriptu, dzięki tej książce możesz stworzyć bardziej ambitne interfejsy użytkownika w swoich aplikacjach. Znajdziesz tu szczegółowy opis biblioteki React i najlepszych metod tworzenia aplikacji z gotowych składników, a także opisy kilku innych narzędzi i bibliotek (takich jak React Router i architektura Flux). Wszystkie tematy zostały zaprezentowane w jasny i zwięzły sposób, a w każdym rozdziale przedstawiono pewne typowe problemy wraz ze sposobami ich rozwiązania.

W tej książce omówiono:

- podstawy konfiguracji biblioteki React i struktury interfejsów tworzonych za jej pomocą
- metody tworzenia kompletnych aplikacji z komponentów React
- wykorzystanie zdarzeń React, implementację drzewa DOM, a także właściwości i stany komponentów React
- bibliotekę React Router i trasowanie
- wydajność aplikacji i optymalizację kodu React
- testowanie aplikacji, również w systemie Jest

Cássio de Sousa Antonio — zaczął programować 20 lat temu na komputerze Sinclair Spectrum. Jest wyjątkowo doświadczonym programistą. Pracował jako kierownik techniczny w różnych firmach w Brazylii i USA nad oprogramowaniem dla takich firm jak Microsoft, Coca-Cola, Unilever czy HSBC. W 2014 roku sprzedał swój startup i dziś jest konsultantem.

Helion
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl
Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci
Informatyka w najlepszym wydaniu
ISBN 978-83-283-2992-8
9 788328 329928
cena: 49,00 zł