

KIRUPA CHINNATHAMBI



REACT I REDUX

Praktyczne tworzenie aplikacji WWW

WYDANIE III

Helion 

Tytuł oryginału: Learning React: A Hands-On Guide to Building Web Applications
Using React and Redux (2nd Edition)

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-4727-4

Authorized translation from the English language edition, entitled: LEARNING REACT:
A HANDS-ON GUIDE TO BUILDING WEB APPLICATIONS USING REACT AND REDUX,
Second Edition; ISBN 013484355X; by Kirupa Chinnathambi; published by Pearson Education, Inc,
publishing as Addison-Wesley Professional.

Copyright © 2018 by Pearson Education, Inc.

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 SA. Copyright ©2019.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiekolwiek 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 właścicielami.

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

Helion SA

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/rerew2_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Mojemu Tacie

*(który zawsze we mnie wierzył, nawet wtedy,
gdy to, co robiłem, nie miało dla niego większego sensu... dla mnie zresztą też nie!)*



Spis treści

O autorze	11
Podziękowania	11
Rozdział 1. Wstęp do biblioteki React	13
Stara szkoła — witryny wielostronne	14
Nowa szkoła — witryny jednostronne	15
Przedstawiamy React	18
Automatyczne zarządzanie stanem interfejsu użytkownika	18
Błyskawiczne modyfikowanie modelu DOM	19
Interfejsy API do tworzenia naprawdę rozbudowanych interfejsów użytkownika	20
Elementy interfejsu zdefiniowane całkowicie w języku JavaScript	21
Tylko V w architekturze MVC	22
Podsumowanie	23
Rozdział 2. Twoja pierwsza aplikacja React	25
Język JSX	26
Pierwsze kroki z React	27
Wyświetlenie imienia	28
To wszystko jest dobrze znane	30
Zmiana miejsca docelowego	30
Trocę stylu!	31
Podsumowanie	33
Rozdział 3. Komponenty biblioteki React	35
Krótkie przypomnienie funkcji	36
Zmiana obsługi interfejsu użytkownika	37
Komponent React	39
Utworzenie komponentu „Witaj, świecie!”	40
Właściwości	43
Operacja 1.: zmiana definicji komponentu	43
Operacja 2.: zmiana wywołania komponentu	43
Dzieci komponentu	44
Podsumowanie	45

Rozdział 4. Style w bibliotece React	47
Wyświetlenie kilku samogłosek	47
Stylizowanie treści za pomocą reguł CSS	49
Struktura generowanego kodu HTML	49
Nadajmy styl wreszcie!	50
Stylizowanie treści według React	51
Tworzenie obiektu stylizującego	52
Właściwa stylizacja treści	53
Dostosowywanie koloru tła	54
Podsumowanie	54
Rozdział 5. Tworzenie złożonych komponentów	57
Od elementów interfejsu do komponentów	57
Określenie głównych elementów wizualnych	58
Określenie potrzebnych komponentów	61
Tworzenie komponentów	63
Komponent Card	64
Komponent Square	65
Komponent Label	66
Znowu przekazywanie właściwości!	68
Dlaczego możliwość łączenia komponentów jest super?	70
Podsumowanie	71
Rozdział 6. Przekazywanie właściwości	73
Opis problemu	73
Szczegółowy opis problemu	75
Poznaj operator rozciągania	79
Lepszy sposób przekazywania właściwości	80
Podsumowanie	82
Rozdział 7. Witamy ponownie JSX!	83
Co się dzieje z kodem JSX?	83
Atuty JSX, które trzeba znać	84
Wyrażenia	85
Zwracanie wielu elementów	85
Nie można definiować stylów CSS w kodzie	87
Komentarze	87
Wielkości liter, elementy HTML i komponenty	88
Kod JSX można stosować wszędzie	89
Podsumowanie	89
Rozdział 8. Obsługa stanów w React	91
Stosowanie stanów	91
Punkt wyjścia	91
Włączenie licznika	93
Określanie początkowej wartości stanu	94
Uruchomienie czasomierza i ustawienie stanu	95
Wizualizacja zmiany stanu	97

Opcja: pełny kod	97
Podsumowanie	99
Rozdział 9. Od danych do interfejsu użytkownika101	
Przykład	101
Kod JSX można stosować wszędzie (część II)	103
Tablice	104
Podsumowanie	106
Rozdział 10. Zdarzenia w React109	
Nasłuchiwanie i obsługiwanie zdarzeń	109
Punkt wyjścia	110
Przygotowanie przycisku do reagowania na kliknięcie	112
Właściwości zdarzenia	113
Poznaj zdarzenia syntetyczne	114
Korzystanie z właściwości zdarzeń	115
Więcej o zawiłościach zdarzeń	116
Zdarzeń nie można nasłuchiwać bezpośrednio w komponentach	116
Nasłuchiwanie zwykłych zdarzeń modelu DOM	118
Obiekt this w procedurze obsługuje zdarzenia	119
React, ale dlaczego?	120
Kompatybilność ze starszymi przeglądarkami	120
Większa wydajność	120
Podsumowanie	120
Rozdział 11. Cykl życia komponentu123	
Poznaj metody cyklu życia	123
Metody cyklu życia w akcji	124
Faza pierwszego wyświetlenia	127
Faza aktualizacji	128
Faza odmontowania	131
Podsumowanie	131
Rozdział 12. Dostęp do elementów DOM w bibliotece React133	
Aplikacja Koloryzator	135
Poznaj referencje	137
Portale	140
Podsumowanie	143
Rozdział 13. Konfiguracja środowiska React bez stresu145	
Przedstawiamy projekt Create React	147
Opis utworzonego projektu	149
Utworzenie aplikacji „Witaj, świecicie!”	152
Kompilacja wersji produkcyjnej	155
Podsumowanie	155

Rozdział 14. Przetwarzanie zewnętrznych danych w aplikacji React	157
Podstawy zapytań HTTP	159
Czas na React!	160
Pierwsze kroki	161
Uzyskanie adresu IP	162
Upiększenie aplikacji	164
Podsumowanie	167
Rozdział 15. Niebanalna lista zadań	169
Pierwsze kroki	171
Utworzenie początkowego interfejsu użytkownika	172
Utworzenie pozostałej części aplikacji	173
Dodawanie zadań	173
Wyświetlanie zadań	176
Stylizacja aplikacji	179
Usuwanie zadań	180
Animacje!	182
Podsumowanie	184
Rozdział 16. Tworzenie wysuwanego menu za pomocą biblioteki React	185
Jak działa wysuwane menu?	185
Przygotowanie wysuwanego menu	188
Pierwsze kroki	190
Wyświetlanie i ukrywanie menu	192
Utworzenie przycisku	193
Utworzenie menu	194
Podsumowanie	196
Rozdział 17. Zapobieganie niepotrzebnemu wyświetaniu komponentów	197
Metoda render()	197
Optymalizacja wywołań metody render()	199
Kontynuacja przykładu	199
Monitorowanie wywołań metod render()	200
Modyfikacja aktualizacji komponentu	203
Komponent PureComponent	204
Podsumowanie	205
Rozdział 18. Tworzenie jednostronnej aplikacji za pomocą biblioteki React Router	207
Przykład	208
Pierwsze kroki	209
Tworzenie jednostronnej aplikacji	210
Wyświetlenie początkowej ramki	210
Utworzenie widoków z treścią	211
Biblioteka React Router	213

Kilka poprawek	215
Korekta procesu kierowania	215
Dodanie stylu	216
Wyróżnienie aktywnego odnośnika	217
Podsumowanie	218
Rozdział 19. Wprowadzenie do biblioteki Redux	219
Czym jest Redux?	220
Prosta aplikacja wykorzystująca bibliotekę Redux	223
Czas na bibliotekę Redux	223
Światło, kamera, akcja!	224
Reduktor	225
Magazyn	227
Podsumowanie	228
Rozdział 20. Stosowanie bibliotek React i Redux	229
Biblioteki React i Redux oraz zarządzanie stanem aplikacji	234
Wspólne funkcjonalności bibliotek React i Redux	234
Przygotowanie	237
Utworzenie aplikacji	237
Skorowidz	243

O autorze

Kirupa Chinnathambi spędził większość swojego życia, ucząc innych kochać tworzenie aplikacji WWW tak, jak on sam kocha to robić.

W roku 1999, zanim jeszcze pojawiło się słowo „blog”, zaczął umieszczać porady na stronie kirupa.com. W ciągu kolejnych lat napisał setki artykułów, kilka książek (oczywiście żadna z nich nie była tak dobra, jak ta!) i umieścił w serwisie YouTube dziesiątki filmów. Pracuje w firmie Microsoft jako program manager i w okresach aktywności, kiedy nie pisze ani nie mówi o tworzeniu aplikacji WWW, pomaga innym tworzyć imponujące strony internetowe. Poza okresami aktywności prawdopodobnie śpi lub pisze na swój temat w trzeciej osobie.

Z autorem można w każdej chwili skontaktować się poprzez serwisy Twitter (twitter.com/kirupa), Facebook (facebook.com/kirupa) lub pocztę e-mail (kirupa@kirupa.com).

Podziękowania

Zacznijmy od tego, że nic nie byłoby możliwe, gdyby nie moja wspaniała żona **Meena**, jej wsparcie i zachęty. Gdyby nie zrezygnowała z własnych celów, abym ja mógł przez sześć miesięcy projektować, tworzyć i pisać na nowo to wszystko, co tu widzisz, ta książka byłaby odległym marzeniem.

Chcę podziękować **moim rodzicom**, którzy zawsze zachęcali mnie do nieskrępowanego błażdzenia i spędzania wolnego czasu na robieniu tego, co lubię, na przykład w latach 90. ubiegłego wieku na uczeniu nieznajomych przez internet programowania fajnych rzeczy. Gdyby nie rodzice, nie byłbym nawet w połowie takim zwołanym domatorem, nauczycielem i wojownikiem, jakim dziś jestem :-).

Od strony wydawniczej pisanie słów takich jak te, które teraz czytasz, jest proste. Jednak aby książka trafiła do Twoich rąk, musi przejść niesamowicie skomplikowany proces. Im lepiej poznawałem wszystkie jego niuanse, tym większy podziw czułem dla wszystkich osób, które niestrudzenie pracowały w cieniu, aby utrzymać tę całą maszynę w działaniu. Dziękuję **wszystkim z wydawnictwa Pearson**, dzięki którym ta książka mogła powstać. Jest kilka osób, którym chcę specjalnie podziękować. Są to: **Mark Taber**, z którym miałem okazję cały czas pracować, **Chris Zahn**, który cierpliwie odpowiadał na wszystkie moje pytania i wątpliwości, **Krista Hansing**, która tłumaczyła mój angielski na język bardziej zrozumiały dla ludzi, i **Loretta Yates**, która od samego początku wszystko spinała, aby ta książka mogła się pojawić. Treść od strony merytorycznej dokładnie sprawdzili moi wieloletni przyjaciele i internetowi współpracownicy – **Kyle Murray (pseudonim Krilnon)** i **Trevor McCauley (pseudonim senocular)**. Jestem bezgranicznie wdzięczny za ich dogłębne (i często dowcipne) opinie!

Wstęp do biblioteki React

Dzisiejsze aplikacje WWW oczywiście wyglądają i działają o wiele lepiej niż jakiś czas temu, ale rzeczywista zmiana jest bardziej zasadnicza. Obecnie aplikacje projektuje się i tworzy zupełnie inaczej niż kiedyś. Aby to wyjaśnić, spójrzmy na rysunek 1.1.



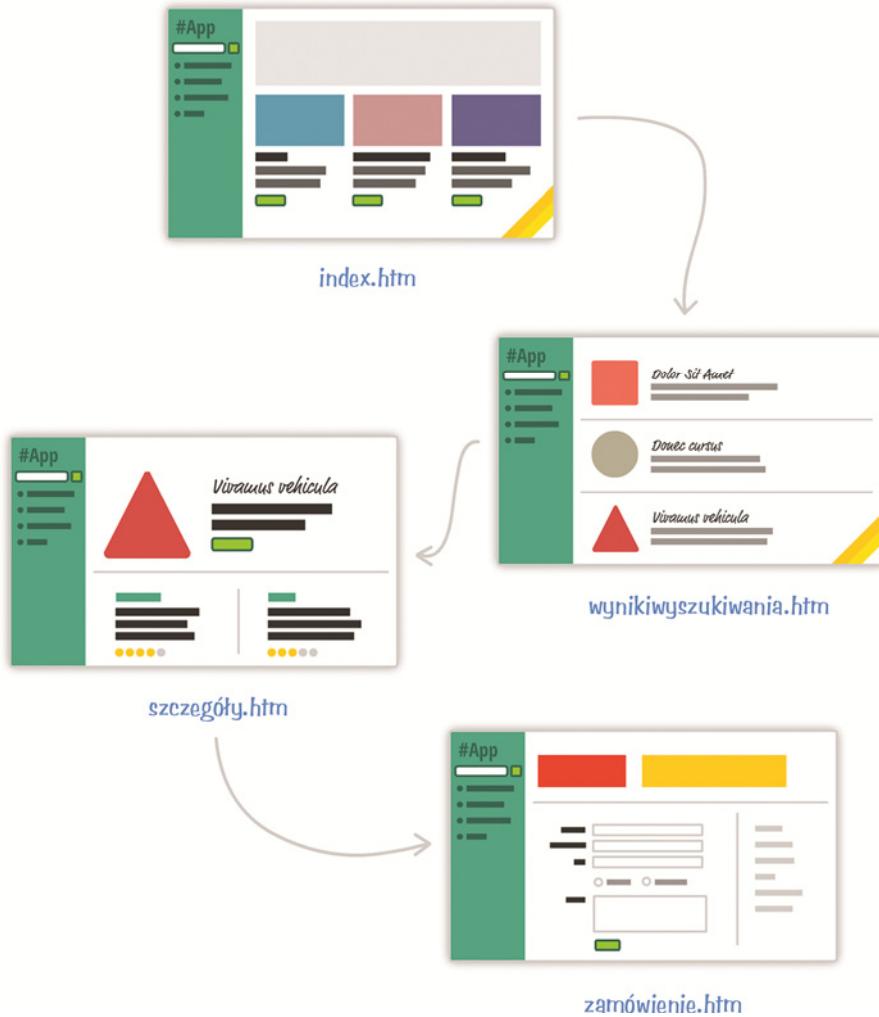
Rysunek 1.1. Aplikacja WWW

Jest to schemat prostej witryny do przeglądania katalogu jakichś rzeczy. Tak jak każda tego rodzaju aplikacja, składa się ona z kilku stron dostępnych z poziomu strony głównej, między innymi strony z wynikami wyszukiwania, ze szczegółami produktu itp. W kolejnych częściach tego rozdziału przyjrzymy się bliżej dwóm sposobom tworzenia takich aplikacji. Oczywiście przy okazji poznasz bibliotekę React.

Zaczynajmy!

Stara szkoła — witryny wielostronowe

Jeszcze kilka lat temu, tworząc aplikację WWW, trzeba było przygotowywać liczne, osobne strony. Proces wyglądał mniej więcej tak jak na rysunku 1.2.

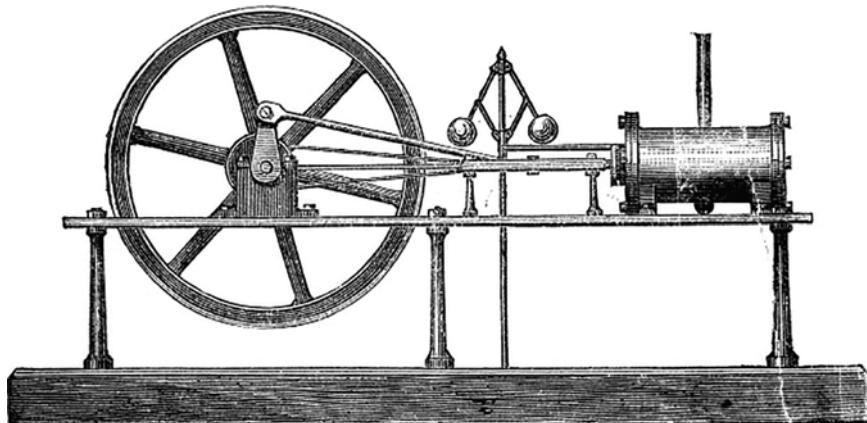


Rysunek 1.2. Wielostronowe witryna WWW

Niemal każda operacja zmieniająca wyświetlaną treść wymagała otwierania osobnej, **zupełnie innej strony**. Było to bardzo proste rozwiązań, choć z punktu widzenia użytkownika nienadzwyczajne, bo bieżąca strona znikała i pojawiała się nowa. Ważny był jednak sposób kontroloowania stanu aplikacji. Poza tym, że było konieczne zapisywanie za pomocą jakiegoś mechanizmu komunikacji danych użytkownika w ciasteczkach lub na serwerze, nie trzeba było się o nic więcej martwić. Życie było piękne.

Nowa szkoła — witryny jednostronne

Dzisiaj model aplikacji wymagającej przechodzenia pomiędzy osobnymi stronami jest przestarzały, i to bardzo. Przyjrzymy się rysunkowi 1.3.



Rysunek 1.3. Model aplikacji opartej na osobnych stronach jest przestarzały tak jak maszyna parowa

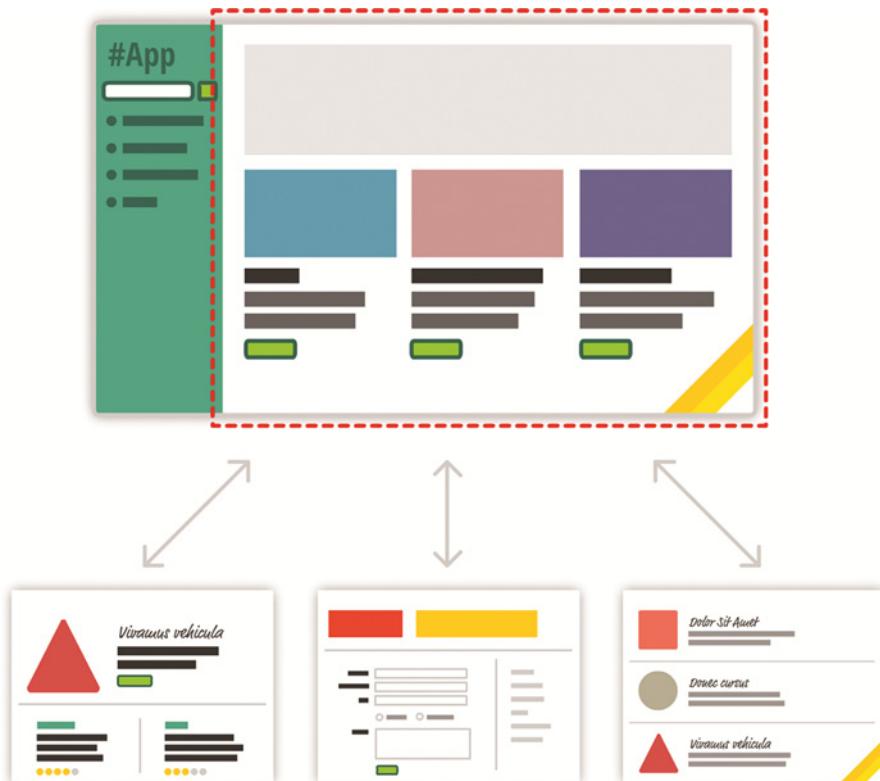
W nowoczesnych aplikacjach wykorzystuje się model **SPA** (ang. *single-page application* — aplikacja jednostronna). Jest to model, w którym w ogóle nie przechodzi się pomiędzy różnymi stronami, ani nawet ich nie przeładowuje. Jest to zupełnie inny świat, w którym różne widoki treści są ładowane i usuwane z jednej i tej samej strony WWW.

W tym modelu nasza aplikacja wygląda mniej więcej tak jak na rysunku 1.4.

Gdy użytkownicy korzystają z takiej aplikacji, modyfikowana jest zawartość obszaru oznaczonego przerywaną linią. Umieszczane są w nim dane i kod HTML odpowiadający operacjom wykonywanym przez użytkownika. Wrażenia z używania takiej aplikacji są o wiele korzystniejsze. Co więcej, można stosować przeróżne techniki wizualizacji zmieniających się treści, wykorzystywane w aplikacjach dla urządzeń przenośnych i komputerów. Tego rodzaju efektów nie da się osiągnąć, wyświetlając osobne strony WWW.

Jeżeli nie słyszaleś wcześniej o aplikacjach SPA, to wszystko może dla Ciebie brzmieć dość nieprawdopodobnie, ale z pewnością miałeś już z takimi aplikacjami do czynienia. Jeśli korzystasz z popularnych serwisów, takich jak Gmail, Facebook, Instagram czy Twitter, to znaczy, że używasz aplikacji jednostronowych. Wszystkie te serwisy wyświetlają dynamicznie treść bez odświeżania bieżącej strony ani otwierania innych stron.

Möże się wydawać, że aplikacje jednostronne są skomplikowane. Jednak *we're* tak nie jest. Dzięki dynamicznemu rozwojowi języka JavaScript oraz różnorodności zewnętrznych platform i bibliotek tworzenie aplikacji jednostronowych jest proste jak nigdy wcześniej. Nie oznacza to jednak, że nie można tego robić jeszcze lepiej.



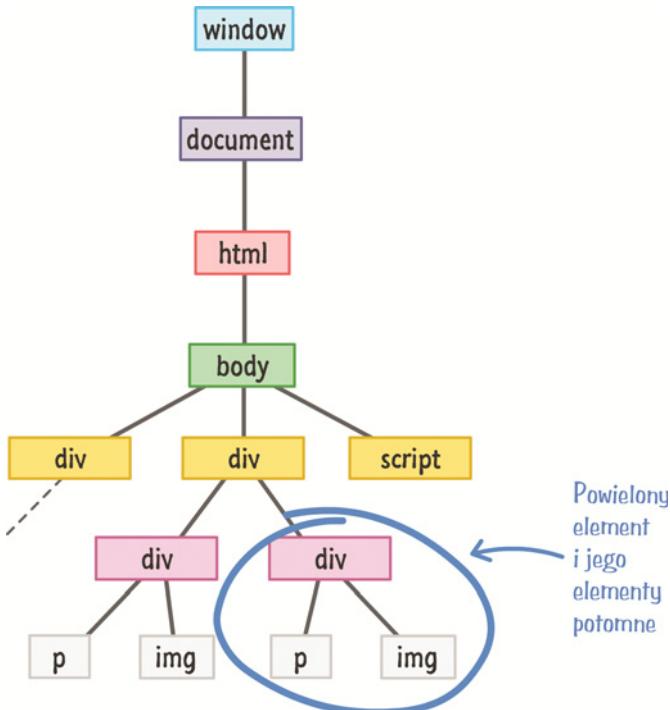
Rysunek 1.4. Aplikacja jednostronna

Podczas tworzenia aplikacji jednostronnej napotkasz w pewnym momencie trzy poważne problemy:

- 1. Większość czasu będziesz musiał poświęcić synchronizowaniu danych na serwerze z tym, co robi użytkownik.** Jeżeli na przykład użytkownik załadowuje nową treść, czy trzeba jawnie czyścić pole wyszukiwania? Czy aktywna zakładka w panelu nawigacyjnym musi być cały czas wyświetlaną? Które elementy muszą być cały czas widoczne na stronie, a które można usuwać?

Są to problemy typowe dla aplikacji jednostronowych. W starym modelu przechodzenia pomiędzy stronami zakładało się, że z interfejsu użytkownika trzeba usuwać wszystko i wyświetlać na nowo. Ponieważ inne rozwiązanie nie istniało, nie był to problem.

- 2. Operacje na modelu DOM wykonują się bardzo, naprawdę bardzo powoli.** Sprawdzanie elementów, dodawanie elementów potomnych (patrz rysunek 1.5), usuwanie poddrzew i inne czynności należą do najwolniej wykonywanych przez przeglądarkę operacji. Niestety, tworząc aplikacje jednostronne będziesz takich operacji kodował mnóstwo. Modyfikowanie modelu DOM jest podstawowym sposobem reagowania na akcje wykonywane przez użytkownika i podstawowym sposobem wyświetlania nowych treści.



Rysunek 1.5. Dodawanie elementów potomnych

3. Korzystanie z szablonów HTML może być trudne. Kontrolowanie aplikacji jednostronnej to nic innego, jak tylko podstawianie fragmentów kodu HTML reprezentujących treści, które mają być wyświetlane. Fragmenty te są nazywane **szablonami**. W praktyce posługiwanie się nimi i wypełnianie ich danymi za pomocą kodu JavaScript bardzo szybko staje się skomplikowane.

Co gorsza, w zależności od wykorzystywanej platformy szablony mogą różnie wyglądać i funkcjonować. Na przykład szablon Mustache koduje się w następujący sposób:

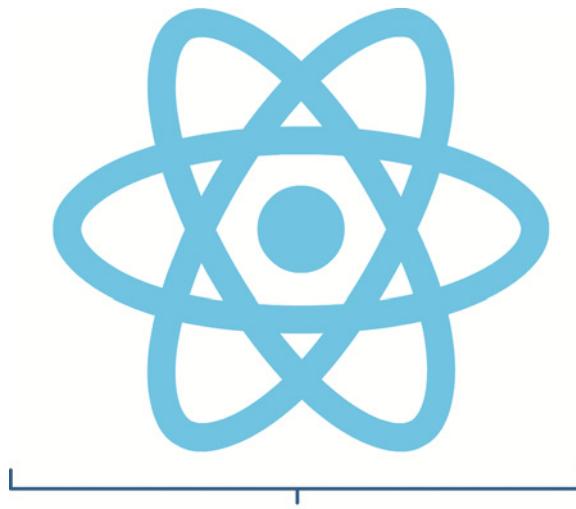
```
var view = {
  title: "Strona",
  calc: function() {
    return 2 + 4;
  }
};
var output = Mustache.render("{{title}} spends {{calc}}", view);
```

Czasami szablony przypominają czysty kod HTML, który można bez najmniejszych problemów stosować w klasach programu, a czasami są zagmatwane, przeładowane niestandardowymi znacznikami, które teoretycznie mają ułatwiać wiązanie elementów HTML z danymi.

Pomimo powyższych utrudnień pisanie aplikacji jednostronowych nie prowadzi donikąd. Aplikacje te są rzeczywistością i wyznaczają trend w tworzeniu witryn WWW w przyszłości. Nie oznacza to oczywiście, że trzeba pogodzić się z opisanymi problemami. Idźmy więc dalej.

Przedstawiamy React

Twórcy serwisów Facebook i Instagram powiedzieli w pewnym momencie: „Dość tego”. Wykorzystując swoje ogromne doświadczenie w tworzeniu aplikacji WWW, przygotowali bibliotekę o nazwie React, która nie tylko rozwiązuje opisane wyżej problemy, lecz także zmienia wyobrażenia o sposobach tworzenia aplikacji jednostronowych.

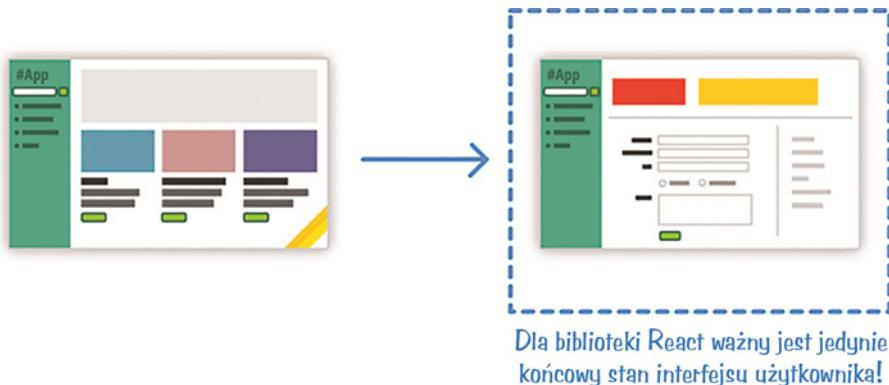


To jest logo biblioteki React!
(Nie wiem, dlaczego tu jest).

W następnych częściach rozdziału przyjrzymy się atomowi biblioteki React.

Automatyczne zarządzanie stanem interfejsu użytkownika

Kontrolowanie interfejsu użytkownika i zarządzanie stanem aplikacji jednostronnej jest trudnym i czasochłonnym zadaniem. Jednak dzięki bibliotece React wystarczy zajmować się tylko jedną rzeczą: końcowym stanem interfejsu. Nie jest ważne, jaki był jego początkowy stan ani jakie operacje wykonywał użytkownik powodujący zmiany. Istotny jest jedynie końcowy wygląd (patrz rysunek 1.6).

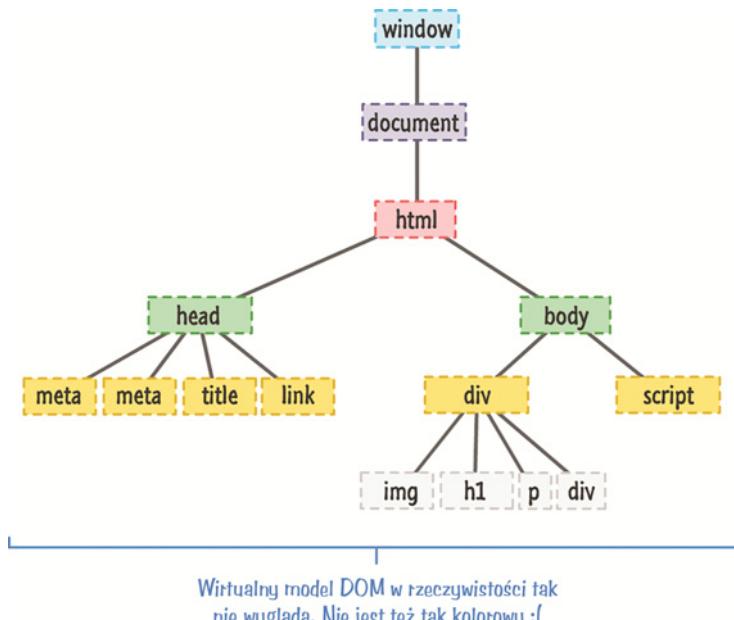


Rysunek 1.6. Dzięki bibliotece React można zająć się tylko końcowym stanem interfejsu użytkownika

Całą resztą związaną z poprawnym wyświetlaniem interfejsu użytkownika zajmuje się biblioteka, dzięki której programista ma z głowy wszystko, co dotyczy zarządzania stanem aplikacji.

Błyskawiczne modyfikowanie modelu DOM

Operacje na modelu DOM są naprawdę długotrwałe, dlatego używając biblioteki React, nie wykonuje się ich bezpośrednio, tylko modyfikuje utworzony w pamięci *wirtualny model DOM* (patrz rysunek 1.7).

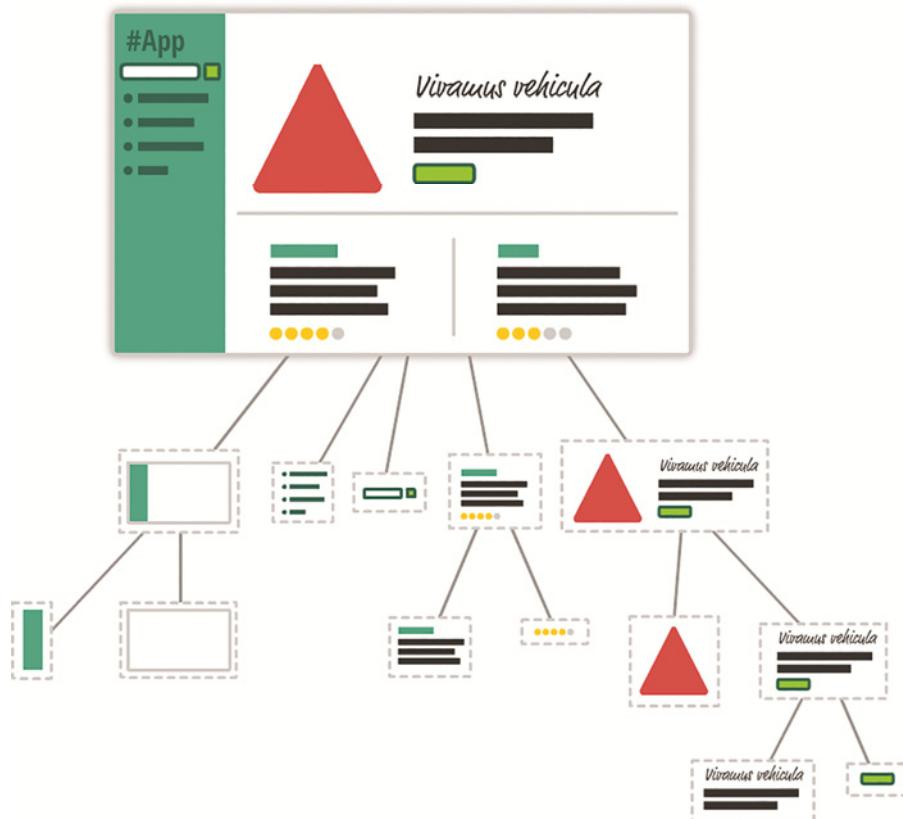


Rysunek 1.7. Tak można sobie mniej więcej wyobrazić wirtualny model DOM

Dzięki temu rozwiążaniu wprowadzanie zmian w modelu odbywa się wyjątkowo szybko, a biblioteka React aktualizuje w odpowiednim momencie rzeczywisty model DOM. W tym celu porównuje modele wirtualny i rzeczywisty, ocenia, które z dokonanych zmian są istotne i w procesie **uzgadniania** wprowadza w rzeczywistym modelu DOM minimalną liczbę zmian, aby był on aktualny.

Interfejsy API do tworzenia naprawdę rozbudowanych interfejsów użytkownika

Korzystając z biblioteki React, nie należy traktować interfejsu użytkownika jako zbudowanego z elementów graficznych monolitu. Zamiast tego należy dzielić interfejs na coraz mniejsze komponenty (patrz rysunek 1.8).



Rysunek 1.8. Przykład dzielenia interfejsu aplikacji na coraz mniejsze części

Tak jak to się zwykle dzieje podczas programowania, tutaj też warto tworzyć modułowe, kompaktowe i samodzielne części. Biblioteka rozszerza tę dobrze znaną praktykę na interfejs użytkownika. Zawiera wiele podstawowych interfejsów API, których przeznaczeniem jest

ułatwianie tworzenia małych komponentów graficznych, łączonych później ze sobą w coraz większe i bardziej skomplikowane komponenty. Ideę tę można zilustrować za pomocą rosyjskiej lalki matrioszki (patrz rysunek 1.9).



Rysunek 1.9. Matrioszka

Jest to jedna z głównych cech biblioteki React upraszczających (i zmieniających) wyobrażenia o tworzeniu aplikacji WWW.

Elementy interfejsu zdefiniowane całkowicie w języku JavaScript

Zabrzmi to wyjątkowo absurdalnie i bluźnierco, ale posłuchaj: szablony HTML nie tylko mają okropną składnię, lecz także tę podstawową wadę, że poza prostym wyświetaniem danych niewiele za ich pomocą da się zrobić. Na przykład aby w zależności od określonego warunku wybrać odpowiedni element do wyświetlenia, trzeba utworzyć gdzieś w kodzie aplikacji skrypt JavaScript albo użyć jakiejś okropnej specjalistycznej platformy.

Poniżej przedstawione jest przykładowe wyrażenie warunkowe zawarte w szablonie EmberJS:

```
 {{#if person}}
  Witaj ponownie, {{person.firstName}} {{person.lastName}}!
{{else}}
  Logowanie.
{{/if}}
```

Biblioteka React robi coś naprawdę fajnego. Interfejs użytkownika jest w całości zdefiniowany w JavaScript, więc dzięki bogatym funkcjonalnościom tego języka można wewnątrz szablonu robić wszystko, co tylko się da. Ograniczeniem są funkcjonalności języka JavaScript, a nie platformy szablonowej.

Teraz, gdy wiesz, że elementy interfejsu są w całości zdefiniowane w JavaScript, prawdopodobnie przeraża Cię myśl o kodowaniu operacji wykorzystujących znaki zapytania, znaki wyjścia i mnóstwo wywołań funkcji createElement(). Bez obaw. Dzięki bibliotece React można (opcjonalnie) definiować elementy, stosując składnię JSX do osadzania

kodu HTML wewnątrz JavaScript. Zamiast kodu definiującego interfejs użytkownika stosuje się znaczniki takie jak:

```
ReactDOM.render(
  <div>
    <h1>Batman</h1>
    <h1>Iron Man</h1>
    <h1>Nicolas Cage</h1>
    <h1>Mega Man</h1>
  </div>,
  destination
);
```

Kod JavaScript, dzięki któremu uzyska się ten sam efekt, wygląda następująco:

```
ReactDOM.render(React.createElement(
  "div",
  null,
  React.createElement(
    "h1",
    null,
    "Batman"
  ),
  React.createElement(
    "h1",
    null,
    "Iron Man"
  ),
  React.createElement(
    "h1",
    null,
    "Nicolas Cage"
  ),
  React.createElement(
    "h1",
    null,
    "Mega Man"
  )
), destination);
```

Super! Dzięki JSX można łatwo definiować elementy, stosując dobrze znaną składnię, i wciąż korzystać z siły i elastyczności języka JavaScript.

Co najważniejsze, dzięki bibliotece React elementy interfejsu i kod JavaScript najczęściej trzyma się w jednym miejscu. Zatem aby zdefiniować wygląd i działanie pojedynczego komponentu, nie trzeba tworzyć wielu osobnych plików.

Tylko V w architekturze MVC

Prawie koniec! React to nie jest uniwersalna platforma narzucająca sposób kodowania wszystkiego, co się dzieje w aplikacji. Wykorzystuje się ją głównie w warstwie widoku (ang. *View*), odpowiedzialnej jedynie za to, aby wszystkie elementy interfejsu były aktualne. Oznacza to, że w warstwach *M* i *C* modelu **MVC** (ang. *Model-View-Controller* — model-widok-kontroler) można robić wszystko, co się żywnie podoba. Dzięki tej elastyczności można stosować dowolne technologie, które się dobrze zna. Poza tym biblioteka React przydaje się nie tylko do tworzenia nowych aplikacji WWW, lecz także do rozwijania tych istniejących bez konieczności usuwania czy modyfikowania obszernych fragmentów kodu.

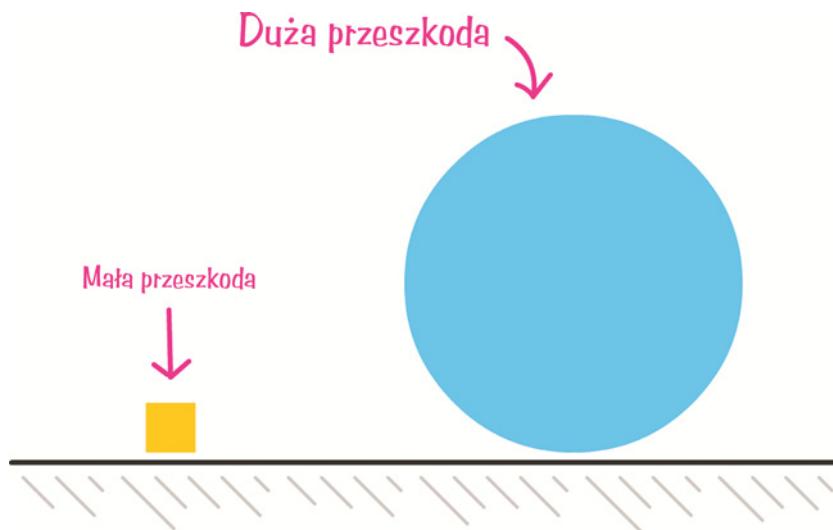
Podsumowanie

Na tle pojawiających się kolejnych platform i bibliotek do tworzenia aplikacji WWW biblioteka React niewątpliwie odniosła sukces. Nie tylko rozwiązuje problemy, z którymi mierzą się twórcy jednostronowych aplikacji WWW, lecz także oferuje kilka sztuczek, dzięki którym budowanie elementów interfejsu aplikacji jest znacznie prostsze. Od chwili pojawienia się w 2013 r. biblioteka ta ugruntowała swoją pozycję w popularnych witrynach i aplikacjach internetowych. Oprócz serwisów Facebook i Instagram są to między innymi strony BBC, Khan Academy, PayPal, Reddit, Yahoo! i „The New York Times”.

Był to wstęp do tego, co biblioteka React robi i po co. W następnych rozdziałach zgłębimy wszystkie poruszone tu tematy i poznasz szczegóły techniczne, dzięki którym z powodzeniem będziesz tworzył własne projekty oparte na bibliotece React. Trzymaj się.

Twoja pierwsza aplikacja React

Po przeczytaniu poprzedniego rozdziału prawdopodobnie znasz już całą historię biblioteki React i wiesz, że pomaga ona tworzyć najbardziej skomplikowane interfejsy użytkownika. Jednak z powodu mnogości niezwykłych funkcjonalności, jakie ta biblioteka oferuje, rozpoczęcie korzystania z niej nie jest proste. Droga do poznania tej biblioteki jest stroma, pełna mniejszych i większych przeszkód, co pokazuje rysunek 2.1.



Rysunek 2.1. Przeszkody są różne, jedne małe, inne duże

W tym rozdziale zaczniemy od podstaw i utrudnimy się trochę, tworząc pierwszą prostą aplikację opartą na bibliotece React. Napotkasz przeszkody, z których kilku na razie nie będziesz usuwał. W tym rozdziale nie tylko zbudujesz coś, co z dumą będziesz mógł pokazać rodzinie i przyjaciolom, lecz także porządnie przygotujesz się do następnych rozdziałów, w których będziesz zgłębiał wszystko to, co biblioteka React ma do zaoferowania.

Język JSX

Zanim zaczniesz tworzyć swoją pierwszą aplikację, musisz dowiedzieć się o jednej ważnej rzeczy. Biblioteka React jest niepodobna do innych bibliotek utworzonych w JavaScript, których prawdopodobnie wcześniej używałeś. Zawiedzisz się, jeżeli będziesz ją wprost odnosił do kodu, który napisał wcześniej, wykorzystując znacznik `<script>`. Biblioteka React jest pod tym względem irytująco wyjątkowa i zmienia sposób tworzenia aplikacji.

Jak wiesz, aplikacja WWW (i wszystko inne, co wyświetla przeglądarka) składa się z kodów HTML, CSS i JavaScript (patrz rysunek 2.2).



Rysunek 2.2. Aplikacja WWW składa się z kodów HTML, CSS i JavaScript

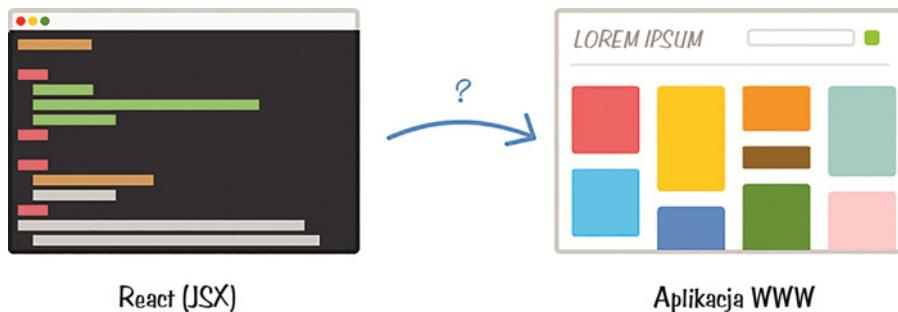
Nie ma znaczenia, czy aplikacja została napisana z użyciem biblioteki React czy innej, na przykład Angular, Knockout lub jQuery. **Końcowy produkt** musi być kombinacją kodów HTML, CSS i JavaScript. W przeciwnym wypadku przeglądarka nie będzie „wiedzieć”, co ma robić.

W tym momencie ujawnia się wyjątkowa natura biblioteki React. W kodzie aplikacji, owszem, wykorzystuje się języki HTML, CSS i JavaScript, ale głównie stosuje się język JSX. Jak wspomniałem w rozdziale 1. „Wstęp do biblioteki React”, JSX jest językiem, w którym łatwo definiuje się elementy i funkcjonalności interfejsu użytkownika, łącząc kod JavaScript ze znacznikami HTML. Brzmi to ciekawie (język JSX zobaczyłeś w akcji już za chwilę), ale pojawia się pewien mały problem. Twój przeglądarka nie ma pojęcia, co ma robić z kodem JSX.

Aby utworzyć aplikację przy użyciu biblioteki React, trzeba znaleźć sposób na przekształcenie kodu JSX na stary, dobry, zrozumiałego dla przeglądarki kod JavaScript (patrz rysunek 2.3).

Jeżeli się tego nie zrobi, aplikacja React po prostu nie będzie działać. Nie brzmi to dobrze. Na szczęście istnieją dwa rozwiązania tego problemu:

- 1. Utworzenie środowiska programistycznego opartego na platformie Node i kilku dodatkowych narzędziach.** W takim przypadku przy każdej komplikacji kod JSX będzie automatycznie przekształcany w kod JavaScript i zapisywany w pliku podobnym do wielu innych plików tego rodzaju.
- 2. Automatyczne przekształcanie kodu JSX w JavaScript przez przeglądarkę w trakcie działania aplikacji.** W takim przypadku kod aplikacji pisze się zwyczajnie w języku JSX, tak jak w JavaScript, a całą resztą zajmuje się przeglądarka.



React (JSX)

Aplikacja WWW

Rysunek 2.3. Kod JSX trzeba zamienić w coś, co przeglądarka rozumie

Oba rozwiązania mają swoje miejsce w naszym świecie, musisz jednak poznać skutki stosowania każdego z nich.

Pierwsze rozwiązanie, na pozór dość skomplikowane i czasochłonne, jest obecnie stosowane przy tworzeniu nowoczesnych aplikacji WWW. Poza tym, że wymaga ono kompilowania (aściślej: **transpilowania**) kodu JSX na JavaScript, pozwala także wykorzystywać różne moduły, narzędzia programistyczne i różnorakie funkcjonalności, dzięki którym tworzenie skomplikowanych aplikacji WWW jest prostsze.

W drugim rozwiązaniu droga do celu jest szybsza i prostsza. Wprawdzie więcej czasu trzeba poświęcić na pisanie kodu, ale za to mniej na majstrowanie przy środowisku programistycznym. Aby wykorzystać ten sposób, wystarczy w kodzie aplikacji umieścić odwołanie do pewnego pliku ze skryptem. Skrypt ten zamienia kod JSX na JavaScript podczas ładowania strony. Dzięki temu Twoja aplikacja zacznie żyć, mimo że nie musiałeś nic specjalnego robić ze środowiskiem programistycznym.

W swojej pierwszej przygodzie z biblioteką React wykorzystasz drugie rozwiązanie. Zapewne dziwisz się, że nie można zawsze z niego korzystać. Powód jest taki, że Twój przeglądarka przy każdym użyciu aplikacji będzie dodatkowo obciążana przekształcaniem kodu JSX w JavaScript. Jest to rozwiązanie całkowicie do przyjęcia w trakcie nauki, ale niedopuszczalne w rzeczywistych aplikacjach. Dlatego później, gdy już dobrze zaznajomisz się z biblioteką React, przejrzymy jeszcze raz cały przerobiony materiał i przygotujemy środowisko programistyczne.

Pierwsze kroki z React

W poprzedniej części rozdziału poznaleś dwa sposoby uczynienia aplikacji React zrozumiałą dla przeglądarki. W tej części przekujemy teorię na praktykę. Na początku będzie potrzebna pusta strona HTML.

Utwórz nowy dokument HTML zawierający następujący treść:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>React! React! React!</title>
```

```
</head>
<body>
  <script>
    </script>
</body>
</html>
```

Ta strona nie ma w sobie nic ciekawego, więc dodajmy do niej odwołanie do biblioteki React. Zaraz pod znacznikiem `<title>` wpisz następujące wiersze:

```
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
```

Powyższy kod ładuje podstawową bibliotekę React oraz różne dodatkowe rzeczy niezbędne do korzystania z modelu DOM. Bez tych wierszy w ogóle nie da się utworzyć aplikacji opartej na bibliotece React.

Ale to nie wszystko. Trzeba dodać odwołanie do jeszcze jednej biblioteki. Tuż pod powyższymi znacznikami `<script>` wpisz następujący wiersz:

```
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

Jest to odwołanie do kompilatora Babel (<http://babeljs.io>). Kompilator ten robi wiele fajnych rzeczy, z których najważniejszą dla nas jest przekształcanie kodu JSX na JavaScript.

W tym momencie Twoja strona HTML powinna wyglądać jak niżej:

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8">
    <title>React! React! React!</title>
    <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
  </head>

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

Jeżeli otworzysz teraz tę stronę w przeglądarce stwierdzisz, że jest pusta. To prawidłowy efekt. Zaraz się tym zajmiemy.

Wyświetlenie imienia

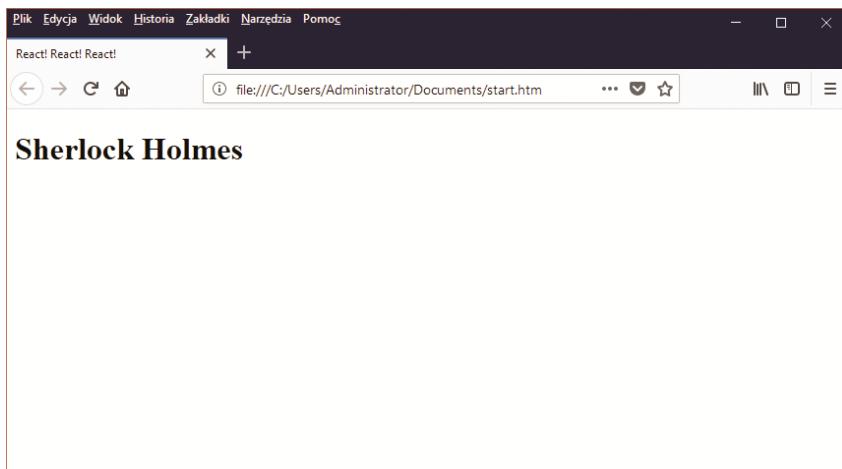
Teraz za pomocą biblioteki React umieścisz na stronie swoje imię. W tym celu użyjesz metody `render()`. Wewnątrz pustego znacznika `<script>` wpisz następujący kod:

```
ReactDOM.render(  
  <h1>Sherlock Holmes</h1>,  
  document.body  
)
```

Nie przejmuj się, jeżeli coś wydaje Ci się bez sensu. Naszym celem jest wyświetlenie na stronie czegokolwiek. Sens temu nadamy później. Teraz, zanim otworzysz stronę i sprawdzisz, co się dzieje, musisz oznać swój skrypt, aby kompilator mógł użyć całej swojej magii. Umieść w znaczniku `<script>` atrybut `type` z wartością `text/babel`:

```
<script type="text/babel">  
  ReactDOM.render(  
    <h1>Sherlock Holmes</h1>,  
    document.body  
)  
</script>
```

Po wprowadzeniu powyższych zmian otwórz stronę w przeglądarce. Zobaczysz wypisane ogromnymi literami słowa `Sherlock Holmes`, jak na rysunku 2.4.



Rysunek 2.4. Twoja przeglądarka powinna wyświetlić słowa „Sherlock Holmes”

Gratulacje! Właśnie utworzyłeś przy użyciu biblioteki React swoją pierwszą aplikację.

W tej aplikacji nie ma się czym ekscytować. Zaczniemy od tego, że raczej nie nazywasz się `Sherlock Holmes`. Aplikacja wprawdzie niewiele robi, ale stanowi wprowadzenie do jednej z najczęściej stosowanych metod w świecie React: `ReactDOM.render()`.

Metoda `render()` ma dwa argumenty:

1. kod podobny do HTML (czyli JSX), który ma być wyświetlony
2. miejsce w modelu DOM, w którym biblioteka React ma wyświetlić kod

Nasza metoda `render()` wygląda tak:

```
ReactDOM.render(  
  <h1>Sherlock Holmes</h1>,  
  document.body  
)
```

Pierwszym argumentem metody jest tekst `Sherlock Holmes` umieszczony wewnątrz znaczników `<h1>`. Ten osadzony w kodzie JavaScript kod podobny do HTML to jest właśnie język JSX. W dalszej części książki poświęcimy mnóstwo czasu na zgłębianie tego języka, ale jedną rzeczą muszę Ci oznajmić już teraz: *właśnie tak przedziwnie wygląda ten kod.* Zawsze, gdy w kodzie JavaScript widzę nawiasy i ukośniki, coś we mnie w środku zamiera, bo jazgot cudzysłówów i znaków wyjścia zagłusza to, co chcę wyrazić. W języku JSX jest inaczej. Wpisuje się po prostu kod HTML tak jak wyżej. W jakiś magiczny sposób to działa.

Drugi argument metody to `document.body`. Nic w nim nadzwyczajnego nie ma. Argument ten po prostu określa miejsce w modelu DOM, w którym zostaną umieszczone znaczniki powstałe z przekształcenia kodu JSX. W tym przykładzie znacznik `<h1>` (i wszystko wewnątrz niego) zostanie umieszczony w elemencie `body` dokumentu.

Wróćmy do pierwotnego celu, którym było wyświetlenie na stronie nie dowolnego, ale **Twojego imienia**. Zmień zatem odpowiednio swój kod. W moim przypadku metoda `render()` wygląda następująco:

```
ReactDOM.render(
  <h1>Batman</h1>,
  document.body
);
```

Wyszło na to, że mam na imię Batman! Tak czy owak, jeżeli teraz otworzysz stronę, zobaczysz zamiast `Sherlock Holmes` swoje imię.

To wszystko jest dobrze znane

Dzięki językowi JSX kod JavaScript wygląda jak nowy, a końcowy produkt trafiający do przeglądarki jest czystą i schludną kombinacją kodów HTML, CSS i JavaScript. Aby się o tym przekonać, wprowadźmy w wyglądzie i działaniu aplikacji kilka zmian.

Zmiana miejsca docelowego

Najpierw zmienimy miejsce, w którym będzie umieszczony wynik kodu JSX. Umieszczanie treści bezpośrednio w elemencie `<body>` jest zdecydowanie złą praktyką. Wiele rzeczy może pójść nie tak jak trzeba, szczególnie gdy mieszka się bibliotekę React z innymi bibliotekami i platformami. Zaleca się więc tworzenie osobnego elementu pełniącego funkcję nowego elementu głównego. Staje się on nowym miejscem docelowym wykorzystywany przez metodę `render()`. Aby zastosować się do tych zaleceń, wróć do kodu HTML i umieść w nim element `<div>` z atrybutem `id` o wartości `container`, jak niżej:

```
<body>
  <div id="container"></div>
  <script type="text/babel">
    ReactDOM.render(
      <h1>Batman</h1>,
      document.body
    );
  </script>
</body>
```

Po zdefiniowaniu bezpiecznego elementu `<div>` zmodyfikujmy metodę `render()` tak, aby korzystała z tego elementu zamiast z `document.body`. Poniżej przedstawiony jest jeden ze sposobów, jak to zrobić:

```
ReactDOM.render(  
  <h1>Batman</h1>,  
  document.querySelector("#container")  
)
```

Innym rozwiązaniem jest dodanie nieco kodu poza samą metodą `render()`:

```
var destination = document.querySelector("#container");  
  
ReactDOM.render(  
  <h1>Batman</h1>,  
  destination  
)
```

Zwróć uwagę, że zmienna `destination` zawiera referencję do elementu `container` w modelu DOM. Dzięki temu wewnątrz metody `render()` zamiast pełnej nazwy elementu wystarczy użyć zmiennej `destination`. Powód wykonania tej operacji jest prosty: chciałem w ten sposób pokazać Ci, że cały czas używasz języka JavaScript, a `render()` jest następną nudną metodą z dwoma argumentami.

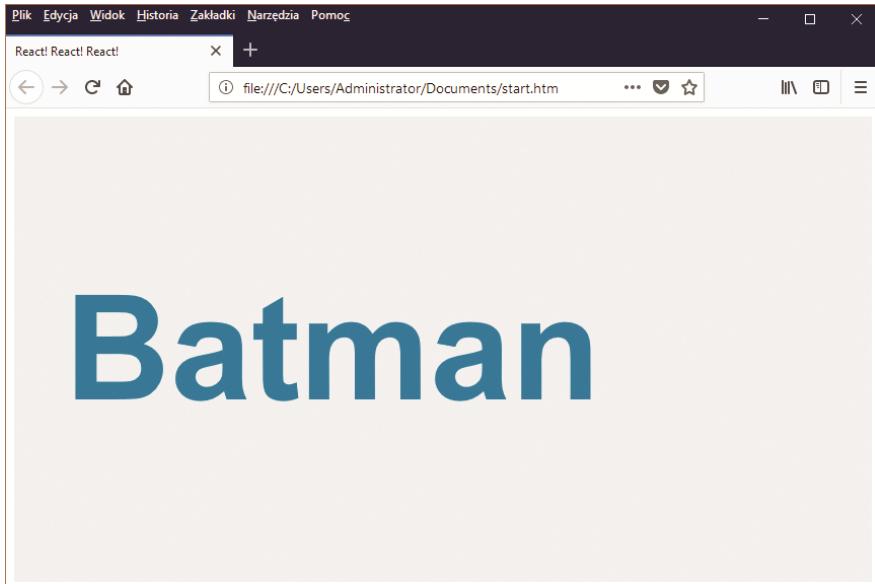
Trochę stylu!

Wprowadźmy jeszcze jedną zmianę, zanim skończymy na dziś. W tej chwili przeglądarka wyświetla Twoje imię, wykorzystując domyślny styl nagłówka `<h1>`. Wygląda on okropnie, więc zmieńmy go, dodając nieco kodu CSS. Wewnątrz znacznika `<head>` wstaw blok `<style>` z następującą zawartością:

```
<style>  
  #container {  
    padding: 50px;  
    background-color: #EEE;  
  }  
  #container h1 {  
    font-size: 144px;  
    font-family: sans-serif;  
    color: #0080A8;  
  }  
</style>
```

Po wprowadzeniu zmian otwórz stronę w przeglądarce. Zwróć uwagę, że teraz tekst wygląda nieco okazalej niż wcześniej, gdy zastosowany był domyślny styl nagłówka `<h1>` (patrz rysunek 2.5).

To działa, ponieważ kiedy został wykonany kod biblioteki React, w elemencie `<body>` znalazł się element `container` ze znacznikiem `<h1>` w środku. Nie ma znaczenia, że znacznik ten został w pełni zdefiniowany za pomocą języków JavaScript i JSX ani że style CSS znajdują się zupełnie poza metodą `render()`. Końcowa aplikacja React składa się w 100% z czystej kombinacji kodów HTML, CSS i JavaScript. Wynik transpilacji wygląda mniej więcej tak:



Rysunek 2.5. Efekt dodania stylu CSS

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>React! React! React!</title>

  <style>
    #container {
      padding: 50px;
      background-color: #EEE;
    }
    #container h1 {
      font-size: 144px;
      font-family: sans-serif;
      color: #0080A8;
    }
  </style>
</head>

<body>
  <div id="container">
    <h1>Batman</h1>
  </div>
</body>

</html>
```

Zauważ, że w powyższym kodzie nie ma ani śladu po bibliotece React.

Podsumowanie

Jeżeli była to Twoja pierwsza aplikacja React, dowiedziałeś się o niej mnóstwa podstawowych rzeczy. Jedną z najważniejszych jest to, że React różni się od innych bibliotek, ponieważ do definiowania elementów interfejsu użytkownika wykorzystuje się w niej całkowicie nowy język JSX. Poznałeś już go nieco, gdy definiowałeś znacznik `<h1>` wewnętrz metodę `render()`.

Możliwości języka JSX wykraczają daleko poza definiowanie elementów interfejsu użytkownika. Przede wszystkim całkowicie zmienia on sposób tworzenia aplikacji. Ponieważ przeglądarka „nie rozumie” kodu JSX w jego naturalnej postaci, trzeba wykonywać pośredni krok i przekształcać go na kod JavaScript. Jedno z podejść polega na transpilowaniu kodu JSX na odpowiadający mu kod JavaScript. Innym rozwiązaniem (wykorzystanym w tym rozdziale) jest zastosowanie kompilatora Babel, który przekształca kod JSX na JavaScript w samej przeglądarce. Ponieważ to podejście ma negatywny wpływ na wydajność przeglądarki, jego stosowanie nie jest zalecane w rzeczywistych, produkcyjnych aplikacjach. Jednak w trakcie poznawania biblioteki React możesz sobie pozwolić na ten luksus.

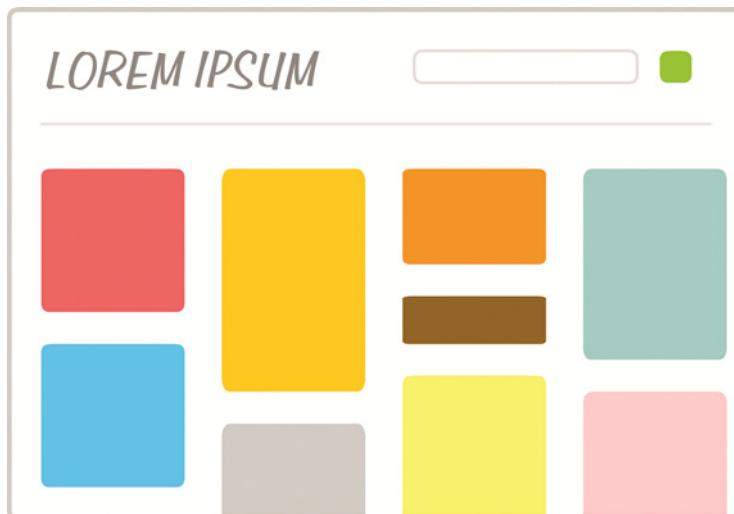
W następnych rozdziałach poświęcimy nieco czasu na zgłębianie języka JSX i innych metod, które obok `render()` stanowią o zaletach biblioteki React.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

Komponenty biblioteki React

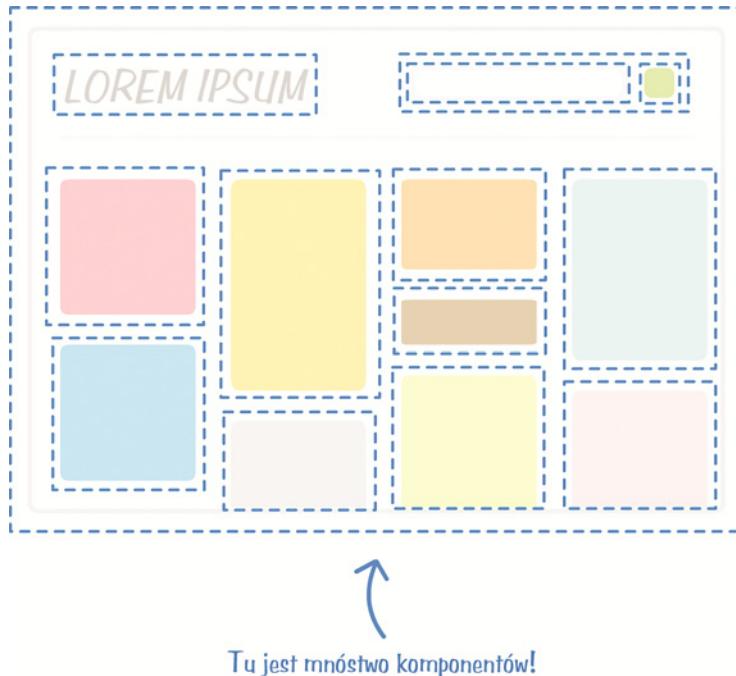
O potędze biblioteki React stanowią między innymi komponenty. To głównie za ich pomocą definiuje się wygląd i funkcjonalności elementów interfejsu wykorzystywanych przez użytkowników aplikacji. Założymy, że Twoja aplikacja wygląda tak jak na rysunku 3.1.



Rysunek 3.1. Twoja gotowa, hipotetyczna aplikacja

Rysunek przedstawia finalny produkt. Jednak od strony projektowej aplikacja React nie wygląda już tak prosto. Niemal każdy element interfejsu jest osadzony w samodzielnym module zwany **komponentem**. Aby dowiedzieć się, dlaczego „niemal każdy”, spójrzmy na rysunek 3.2.

Każdy prostokąt narysowany przerywaną linią reprezentuje osobny komponent odpowiedzialny za to, co widzi i może robić użytkownik. Niech Cię to nie przeraja. Choć na początku rysunek rzeczywiście może się wydawać skomplikowany, to wkrótce wszystko ułoży się w logiczną całość, gdy zaczniesz się posługiwać — a raczej usilnie starać się posługiwać — komponentami i niektórymi oferowanymi przez nie funkcjonalnościami.



Rysunek 3.2. Diagram komponentów aplikacji

Krótkie przypomnienie funkcji

Dzięki **funkcjom** kod JavaScript jest bardziej czytelny i uniwersalny. Już wyjaśniam, dlaczego poświęcamy czas funkcjom. Bynajmniej nie po to, aby Cię denerwować! Funkcje mają wiele wspólnego z komponentami biblioteki React, a najprostszym sposobem zrozumienia działania komponentów jest uprzednie przypomnienie sobie pokrótko, czym są funkcje.

W strasznym świecie bez funkcji kod mógłby wyglądać jak niżej:

```
var speed = 10;
var time = 5;
alert(speed * time);

var speed1 = 85;
var time1 = 1.5;
alert(speed1 * time1);

var speed2 = 12;
var time2 = 9;
alert(speed2 * time2);

var speed3 = 42;
var time3 = 21;
alert(speed3 * time3);
```

W lepszym świecie, w którym są funkcje, można skondensować powtarzające się fragmenty kodu do prostej funkcji, takiej jak ta:

```
function getDistance(speed, time) {  
  var result = speed * time;  
  alert(result);  
}
```

Funkcja `getDistance()` usuwa powtarzające się fragmenty kodu. Ma argumenty `speed` i `time`, dzięki którym można dostosowywać wykonywane obliczenia i uzyskiwać żądane wyniki.

Aby wywołać tę funkcję, wystarczy napisać coś takiego:

```
getDistance(10, 5);  
getDistance(85, 1.5);  
getDistance(12, 9);  
getDistance(42, 21);
```

Wygląda to lepiej, prawda? Funkcje mają jeszcze inną wielką zaletę. Mogą wywoływać inne funkcje (na przykład funkcja `getDistance()` wywołuje funkcję `alert()`). Przyjrzyjmy się funkcji `formatDistance()`, modyfikującej wynik zwracany przez funkcję `getDistance()`:

```
function formatDistance(distance) {  
  return distance + " km";  
}  
  
function getDistance(speed, time) {  
  var result = speed * time;  
  alert(formatDistance(result));  
}
```

Dzięki temu, że funkcje mogą wywoływać inne funkcje, można wyraźnie separować wykonywane przez nie operacje. Nie trzeba pisać jednej monolitycznej funkcji, która robi wszystko. Zamiast tego można rozdzielić operacje pomiędzy wiele wyspecjalizowanych funkcji.

Co najważniejsze, jeżeli coś się zmieni w kodzie funkcji, nie trzeba nic dodatkowo robić, aby zobaczyć wyniki tych zmian. Jeżeli sygnatura funkcji pozostanie taka sama, wtedy wszystkie istniejące wywołania będą w magiczny sposób działały, a wprowadzone zmiany zostaną automatycznie uwzględnione.

Podsumowując, funkcje są fantastyczne. Ja to wiem. Ty też to wiesz. Dlatego w kodzie, który będziesz tworzył, będziesz funkcje stosował wszędzie.

Zmiana obsługi interfejsu użytkownika

Nie sądzę, aby ktokolwiek był niezadowolony z dobrych funkcji. Dzięki nim można nadać kodowi czystą, porządną strukturę. Jednak nie zawsze jest to możliwe w przypadku kodowania interfejsu użytkownika. Z różnych technicznych i nietechnicznych powodów toleruje się do pewnego stopnia niedbałości popełniane podczas tworzenia elementów interfejsu.

Moja opinia może wydawać się bardzo kontrowersyjna, dlatego chciałbym wyjaśnić na przykładzie, co mam na myśli. Wróćmy do poprzedniego rozdziału i przyjrzyjmy się metodzie `render()`:

```
var destination = document.querySelector("#container");  
  
ReactDOM.render(  
  <h1>Batman</h1>,  
  destination  
,
```

W przeglądarce pojawiło się słowo Batman napisane ogromnymi literami dzięki znacznikowi `<h1>`. Wprowadźmy teraz w kodzie trochę zmian. Założymy, że chcemy wyświetlić imiona kilku innych superbohaterów. W tym celu zmienimy metodę `render()` w następujący sposób:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <h1>Batman</h1>
    <h1>Iron Man</h1>
    <h1>Nicolas Cage</h1>
    <h1>Mega Man</h1>
  </div>,
  destination
);
```

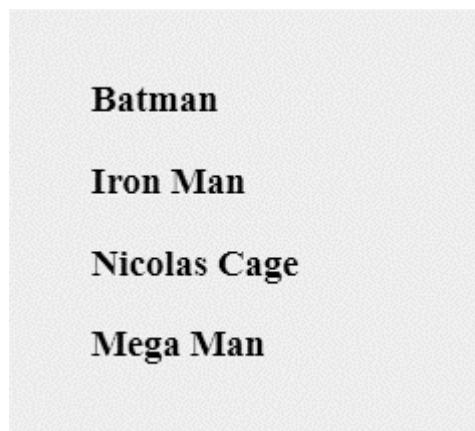
Zwróć uwagę, co tu się dzieje. Generowany jest znacznik `<div>` zawierający cztery elementy `<h1>` z imionami naszych superbohaterów.

OK, teraz mamy cztery elementy `<h1>`, każdy zawierający imię superbohatera. Co trzeba jednak zrobić, aby zmienić element `<h1>` na przykład na `<h3>`? Można ręcznie zmodyfikować wszystkie elementy, jak niżej:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <h3>Batman</h3>
    <h3>Iron Man</h3>
    <h3>Nicolas Cage</h3>
    <h3>Mega Man</h3>
  </div>,
  destination
);
```

Po wyświetleniu powyższego kodu pojawi się strona wyglądająca bardzo zwyczajnie i bez stylu (patrz rysunek 3.3).



Rysunek 3.3. Zwykłe imiona superbohaterów

Nie przesadzajmy w stylizacji. Wystarczy na razie, że za pomocą znacznika `<i>` zamienimy ręcznie czcionkę na pochyłą:

```
var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <h3><i>Batman</i></h3>
    <h3><i>Iron Man</i></h3>
    <h3><i>Nicolas Cage</i></h3>
    <h3><i>Mega Man</i></h3>
  </div>,
  destination
);
```

Zawartość każdego elementu `<h3>` umieściliśmy wewnątrz znaczników `<i>`. Czy widzisz pojawiający się w tym momencie problem? To, co teraz robimy z kodem niczym się nie różni od tworzenia czegoś takiego:

```
var speed = 10;
var time = 5;
alert(speed * time);

var speed1 = 85;
var time1 = 1.5;
alert(speed1 * time1);

var speed2 = 12;
var time2 = 9;
alert(speed2 * time2);

var speed3 = 42;
var time3 = 21;
alert(speed3 * time3);
```

Jeżeli trzeba wprowadzić jakąś zmianę w elementach `<h1>` lub `<h3>`, należy to zrobić w każdym z nich osobno. Co jednak będzie, jeżeli zmiany będą bardziej skomplikowane niż zwykła modyfikacja wyglądu elementu? Albo zmiany będą dotyczyć bardziej zaawansowanych elementów? Sposób, który zastosowaliśmy, nie jest skalowalny. Ręczne zmienianie każdego elementu jest czasochłonne. I do tego nudne.

Pomyślmy odważnie: *a gdyby tak znanie nam niesamowite właściwości funkcji można było w jakiś sposób wykorzystać do definiowania elementów naszej aplikacji?* Czy nie byłoby to remedium na nieskuteczność opisanego wyżej rozwiązania? Jak się okazuje, odpowiedzi na pytania typu „a gdyby tak” stanowią istotę biblioteki React. Nadszedł czas, aby poznać **komponent**.

Komponent React

Rozwiążaniem wszystkich naszych problemów (nawet tych egzystencjalnych!) są komponenty biblioteki React. *Są to uniwersalne fragmenty kodu JavaScript wykorzystujące składnię JSX do generowania kodu HTML.* Jak na sposób rozwiązywania poważnych problemów nie brzmi to wiarygodnie, ale gdy zaczniesz tworzyć coraz bardziej skomplikowane elementy, przekonasz się o potędze komponentów i uznasz, że moja opinia o nich wcale nie jest przesadzona.

Zacznijmy od utworzenia kilku elementów. Najpierw trzeba przygotować prosty dokument React:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Komponenty React</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">

    </script>
</body>

</html>
```

Nic specjalnego na tej stronie nie ma. Jak w poprzednim rozdziale, jest to szkielet zawierający odwołania do bibliotek React i Babel oraz element `<div>` dumnie opatrzony atrybutem `id` o wartości `container`.

Utworzenie komponentu „Witaj, świecie!”

Zacznijmy od czegoś bardzo prostego. Wykorzystajmy komponent do wyświetlenia legendarnego komunikatu „Witaj, świecie!”. Jak już wiesz, można do tego celu użyć metody `render()` obiektu `ReactDOM`, jak w poniższym kodzie:

```
ReactDOM.render(
  <div>
    <p>Witaj, świecie!</p>
  </div>,
  document.querySelector("#container")
);
```

Zróbjmy to jednak na nowo, wykorzystując komponent. Za pomocą biblioteki React można to osiągnąć na wiele sposobów, my użyjemy klasy. Wpisz nad wierszem z metodą `render()` wyróżniony fragment kodu:

```
class HelloWorld extends React.Component {
}

ReactDOM.render(
  <div>
    <p>Witaj, świecie!</p>
  </div>,
  document.querySelector("#container")
);
```

Jeżeli nie znasz składni klasy, zapoznaj się z podręcznikiem *Using Classes in JavaScript* („Klasy w JavaScript” — https://www.kirupa.com/javascript/classy_way_to_create_objects.htm).

Wróćmy do naszego kodu. Utworzyliśmy komponent o nazwie `HelloWorld`. Nazywamy go komponentem, ponieważ jest to rozszerzenie klasy `React.Component`. Bez tego rozszerzenia powstałaby pusta, bezużyteczna klasa. Wewnątrz utworzonej klasy można umieszczać dowolne metody definiujące operacje, które komponent ma wykonywać. Niektóre z nich są metodami specjalnymi, pozwalającymi wykorzystywać magię biblioteki React. Jedną z nich jest metoda `render()`.

Zmień teraz komponent `HelloWorld`, umieszczając w nim metodę `render()`, jak niżej:

```
class HelloWorld extends React.Component {  
  render() {  
    }  
}
```

Metoda ta, podobnie jak opisana wcześniej metoda `ReactDOM.render()`, służy do przetwarzania kodu JSX. Zmień teraz swoją metodę tak, aby zwracała ciąg `Witaj, komponentowy świecie!`. W tym celu wpisz wyróżniony wiersz:

```
class HelloWorld extends React.Component {  
  render() {  
    return <p>Witaj, komponentowy świecie!</p>  
  }  
}
```

Sprawiłeś, że metoda zwraca kod JSX reprezentujący ciąg `Witaj, komponentowy świecie!`. Teraz pozostało jedynie zastosować nowy komponent. W tym celu trzeba go wywołać poniżej definicji. Użyjemy do tego dobrze nam znanej metody `ReactDOM.render()`.

Sposób wywoływania komponentu jest dość nietypowy. Zastąp pierwszy argument metody `ReactDOM.render()` wyróżnionym niżej kodem:

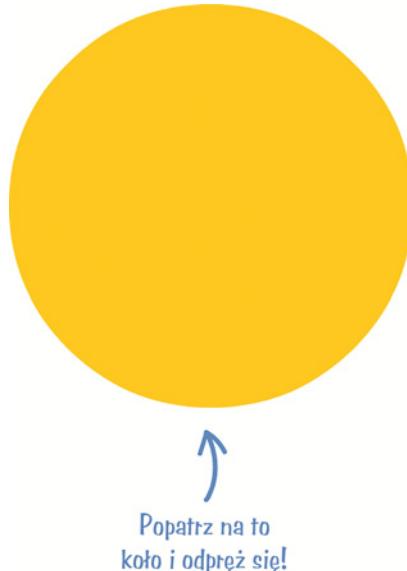
```
ReactDOM.render(  
  <HelloWorld/>,  
  document.querySelector("#container")  
>;
```

To nie pomyłka! Kod JSX `<HelloWorld/>` użyty do wywołania komponentu `HelloWorld` jest bardzo podobny do znacznika HTML. Po otwarciu strony w przeglądarce zobaczysz na samej górze napis `Witaj, komponentowy świecie!`. Jeżeli z napięcia wstrzymałes oddech, możesz się odprężyć. Jeżeli widząc składnię wywołania komponentu `HelloWorld`, nie możesz odetchnąć, popatrz przez chwilę na rysunek 3.4.

Wróćmy do rzeczywistości. To, co zrobiliśmy, może wyglądać dziwnie, ale traktuj komponent `<HelloWorld>` jako nowy, fajny znacznik HTML, którego funkcjonalności masz pod pełną kontrolą, tzn. możesz z nim robić to, co z każdym innym znacznikiem.

Zmień teraz metodę `ReactDOM.render()` w następujący sposób:

```
ReactDOM.render(  
  <div>  
    <HelloWorld/>  
  </div>,  
  document.querySelector("#container")  
>;
```



Rysunek 3.4. Ku rozproszeniu uwagi na chwilę

Komponent `HelloWorld` umieściliśmy wewnętrz elementu `<div>`. Jeżeli teraz otworzysz stronę w przeglądarce, wszystko zostanie wyświetlane tak, jak trzeba. Pójdzmy krok dalej! Zamiast jednego wywołania komponentu `HelloWorld` umieścmy ich kilka. Zmień metodę `ReactDOM.render()` jak niżej:

```
ReactDOM.render(
  <div>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
    <HelloWorld/>
  </div>,
  document.querySelector("#container")
);
```

Teraz na stronie zobaczysz kilka napisów „Witaj, komponentowy świecie!”. Zanim przejdziemy do bardziej efektownych rzeczy, wprowadźmy jeszcze jedną zmianę. Wróćmy do deklaracji komponentu `HelloWorld` i zmieńmy zwracany ciąg na tradycyjny „Witaj, świecie!”:

```
class HelloWorld extends React.Component {
  render() {
    return <p>Witaj, świecie!</p>
  }
}
```

Po wprowadzeniu zmian ponownie otwórz stronę. Teraz wszystkie umieszczone wcześniej wywołania komponentu `HelloWorld` wyświetlają na stronie komunikaty `Witaj, świecie!`. Nie trzeba było zmieniać każdego wywołania osobno — świetnie!

Właściwości

Na razie nasz komponent robi tylko jedną rzecz: umieszcza na stronie napis Witaj, świecie!, i nic poza tym. Jest on odpowiednikiem takiej funkcji JavaScript jak poniższa:

```
function getDistance() {
  alert("42 km");
}
```

Funkcja ta, z wyjątkiem szczególnych sytuacji, nie jest zbyt przydatna, prawda? Aby uczynić ją bardziej przydatną, można dodać do niej argumenty:

```
function getDistance(speed, time) {
  var result = speed * time;
  alert(result);
}
```

Teraz funkcja jest bardziej uniwersalna i można ją stosować w różnych sytuacjach, nie tylko wymagających wyświetlenia komunikatu „42 km”.

Ta zasada dotyczy też komponentów. Podobnie jak funkcje mogą one mieć argumenty modyfikujące ich działanie. Krótka uwaga, o której należy pamiętać, dotycząca stosowanej terminologii: to, co w świecie funkcji nazywa się **argumentem**, w świecie komponentów nosi nazwę **właściwości**. Zobaczmy więc właściwości w akcji!

Zmienimy teraz komponent HelloWorld tak, aby można było określić kogo albo co chcemy pozdrowić, nie tylko cały świat. Na przykład chcielibyśmy w wywołaniu komponentu HelloWorld umieścić słowo Bono i wyświetlić napis Witaj, Bono!.

Aby do komponentu dodać właściwość, trzeba wykonać dwie operacje.

Operacja 1: zmiana definicji komponentu

Na razie nasz komponent HelloWorld zwraca zakodowany na stałe ciąg Witaj, świecie!. Trzeba to zmienić tak, aby instrukcja return zwracała wartość zapisaną we właściwości. Właściwość musi mieć swoją nazwę. W tym przykładzie niech to będzie greetTarget.

Aby w komponencie wykorzystać właściwość greetTarget, trzeba go zmienić w następujący sposób:

```
class HelloWorld extends React.Component {
  render() {
    return <p>Witaj, {this.props.greetTarget}!</p>
  }
}
```

Aby odwołać się do nowej właściwości, należy użyć właściwości `this.props` dostępnej w każdym komponencie. Zwróć uwagę na specyfikację właściwości: jest umieszczona w nawiasach klamrowych `{}`. **W języku JSX wyrażenie, którego wartość ma być użyta, umieszcza się w takich nawiasach**. W naszym przykładzie, gdyby nie było nawiasów, na stronie pojawiłby się napis `this.props.greetTarget`.

Operacja 2: zmiana wywołania komponentu

Po wprowadzeniu zmian w definicji komponentu wystarczy w jego wywołaniu umieścić wartość właściwości. W tym celu należy dodać atrybut o takiej samej nazwie jak właściwość oraz przypisać

mu wartość. W naszym przykładzie w wywołaniu komponentu `HelloWorld` trzeba dodać atrybut `greetTarget` i wartość, która będzie mu przypisana.

Zmień teraz wywołania komponentu `HelloWorld` w następujący sposób:

```
ReactDOM.render(
  <div>
    <HelloWorld greetTarget="Batman"/>
    <HelloWorld greetTarget="Iron Man"/>
    <HelloWorld greetTarget="Nicolas Cage"/>
    <HelloWorld greetTarget="Mega Man"/>
    <HelloWorld greetTarget="Bono"/>
    <HelloWorld greetTarget="Catwoman"/>
  </div>,
  document.querySelector("#container")
);
```

Teraz w każdym wywołaniu użyty jest atrybut `greetTarget` z imieniem superbohatera (lub innego stwora), którego chcemy pozdrowić. Gdy wyświetlisz stronę, zobaczysz na niej różne pozdrowienia.

Zanim przejdziemy dalej, zwróć uwagę na ważną rzecz. Komponent nie musi mieć tylko jednej właściwości. Może ich być dowolnie dużo i można się do nich bez najmniejszych problemów odwoływać za pomocą właściwości `props`.

Dzieci komponentu

Wspomniałem wcześniej, że komponenty (w języku JSX) są bardzo podobne do elementów HTML. Przekonałeś się o tym, osadzając swój komponent wewnętrz znaczników `<div>` i umieszczając w nim atrybut z wartością przekazywaną właściwości. **Komponent, podobnie jak element HTML, może mieć dzieci.**

Oznacza to, że można robić coś takiego:

```
<FajnyKomponent foo="bar">
  <p>Coś tam.</p>
</FajnyKomponent>
```

Został tu użyty komponent o niebanalnej nazwie `FajnyKomponent`, którego dzieckiem jest element `<p>`. Komponent `FajnyKomponent` może odwoływać się do elementu `<p>` (i jego dzieci, jeżeli je posiada) za pomocą właściwości `this.props.children`.

Aby zrozumieć ten mechanizm, przeanalizujmy inny, bardzo prosty przykład. Zdefiniujmy komponent `Buttonify`, który umieszcza swoje dzieci wewnątrz przycisku. Kod takiego komponentu wygląda następująco:

```
class Buttonify extends React.Component {
  render() {
    return(
      <div>
        <button type={this.props.behavior}>{this.props.children}</button>
      </div>
    );
  }
}
```

Komponent ten uruchamiamy, wywołując metodę ReactDOM.render(), jak niżej:

```
ReactDOM.render(
  <div>
    <Buttonify behavior="submit">WYŚLIJ DANE</Buttonify>
  </div>,
  document.querySelector("#container")
);
```

Gdy otworzysz stronę z powyższym kodem, zobaczysz przycisk ze słowami WYŚLIJ DANE, dlatego że taką treść zawiera kod JSX metody render() komponentu Buttonify. Jeżeli dodatkowo użyjesz odpowiedniego stylu, przycisk może być śmiesznie duży, jak na rysunku 3.5.



Dlaczego ten przycisk jest taki duży?

Rysunek 3.5. Duży przycisk Wyślij dane

Wróćmy do kodu JSX. Zwróć uwagę, że została w nim użyta niestandardowa właściwość behavior (działanie). Jest ona wartością atrybutu type przycisku. Metoda render() komponentu odwołuje się do niej za pomocą właściwości this.props.behavior.

Do dzieci komponentu można odwoływać się na więcej sposobów. Na przykład jeżeli element dziecko jest zwykłym tekstem, wtedy właściwość this.props.children zawiera ten tekst. Jeżeli dzieckiem jest pojedynczy element (tak jak w tym przykładzie), wtedy właściwość this.props.children zawiera ten element nieosadzony w tablicy. Podobnych przykładów jest więcej, ale nie będę Cię nudził wymienianiem ich w tej chwili. Przyjdzie na nie czas, gdy zajmiemy się bardziej skomplikowanymi przykładami.

Podsumowanie

Tworząc aplikację opartą na bibliotece React, nie zajdziemy daleko bez komponentów. Przypominałoby to pisanie kodu JavaScript bez użycia funkcji — nie chcę powiedzieć, że nie jest to możliwe, tylko że nie powinieneś tak robić, bo jest to zła praktyka. Może do stosowania komponentów przekona Cię następny rozdział, w którym będziemy tworzyć bardziej skomplikowane struktury.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniemi, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

4

Style w bibliotece React

Ludzkość od pokoleń stylizuje treści HTML za pomocą reguł CSS. I bardzo dobrze. Dzięki nim można wyraźnie oddzielić od siebie warstwy treści i prezentacji. Składnia selektorów jest bardzo elastyczna i pozwala określić, które elementy mają być stylizowane, a które nie. Nie ma najmniejszego powodu, aby nie lubić **kaskadowych stylów**, na których opiera się język CSS.

Jednak w przypadku biblioteki React jest inaczej. Nie można wprawdzie powiedzieć, że nie lubi ona stylów CSS, stosuje jednak inne podejście do formatowania elementów. Jak już się przekonałeś, jedną z podstawowych zasad obowiązujących w tej bibliotece jest tworzenie samodzielnych, powtarzalnych elementów wizualnych. Dlatego w jednym kawałku, zwany **komponentem**, stosuje się kody HTML i JavaScript definiujące element. Zasmakowałeś już tego w poprzednim rozdziale.

Jak należy określić wygląd (czyli styl) elementów HTML? Gdzie trzeba go kodować? Przypuszczam, że wiesz, o co mi chodzi. Nie można tworzyć samodzielnych elementów interfejsu, jeżeli ich style są zdefiniowane gdzieś poza nimi. Dlatego jeżeli korzysta się z biblioteki React, należy wygląd elementów określić za pomocą kodów HTML i JavaScript. W tym rozdziale dowiesz się wszystkiego o tym tajemniczym (a nawet oburzającym) sposobie stylizacji treści. Oczywiście dowiesz się również, jak stosować style CSS. Jest miejsce na oba sposoby, nawet gdyby biblioteka React miało się to nie podobać.

Wyświetlenie kilku samogłosek

Aby poznać zasady stylizacji treści w bibliotece React, przeanalizujmy przykład (naprawdę fajny i ciekawy) prostej strony z samogłoskami. Do jej utworzenia będzie potrzebny przede wszystkim szablon dokumentu HTML, w którym później umieścisz treść za pomocą biblioteki React. Utwórz nowy dokument HTML z następującą zawartością:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Stylizacja w bibliotece React</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

```

<style>
  #container {
    padding: 50px;
    background-color: #FFF;
  }
</style>
</head>

<body>
  <div id="container"></div>
</body>
</html>

```

Aby wyświetlić wybrane samogłoski, musisz wpisać trochę kodu właściwego dla biblioteki React. Tuż pod elementem div dodaj:

```

<script type="text/babel">
  var destination = document.querySelector("#container");

  class Letter extends React.Component {
    render() {
      return(
        <div>
          {this.props.children}
        </div>
      );
    }
  }

  ReactDOM.render(
    <div>
      <Letter>A</Letter>
      <Letter>E</Letter>
      <Letter>I</Letter>
      <Letter>O</Letter>
      <Letter>U</Letter>
    </div>,
    destination
  );
</script>

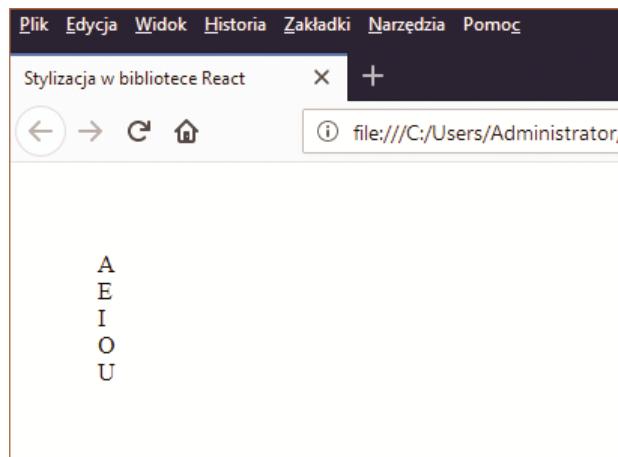
```

Jeżeli dobrze opanowałeś omówione wcześniej zagadnienia, nic tu nie powinno wyglądać dla Ciebie tajemniczo. Utworzysz komponent o nazwie Letter, osadzający literę w elemencie div.

Gdy otworzysz stronę, pojawi się widok podobny do przedstawionego na rysunku 4.1.

Nie przejmuj się, za chwilę sprawisz, że strona będzie trochę mniej nudna. Litery będą wyglądały tak jak na rysunku 4.2.

Litery będą umieszczone na żółtym tle i ułożone poziomo; użyjemy też ciekawego kroju pisma. Sprawdźmy teraz, jak można ten efekt osiągnąć, stosując style CSS i nowatorskie podejście w bibliotece React.



Rysunek 4.1. Nudne jest to, co widzisz



Rysunek 4.2. Tak będzie wyglądała strona po zastosowaniu stylów!

Stylizowanie treści za pomocą reguł CSS

Stylizowanie treści za pomocą reguł CSS jest tak proste, jak to sobie wyobrażasz. Ponieważ biblioteka React generuje zwykłe znaczniki HTML, można stosować wszelkie sztuczki języka CSS, które znasz od lat. Trzeba tylko pamiętać o kilku pomniejszych kwestiach.

Struktura generowanego kodu HTML

Zanim zastosujesz reguły CSS, musisz się dowiedzieć, jak wygląda kod HTML generowany przez bibliotekę React. Możesz do łatwo sprawdzić, patrząc na kod JSX umieszczony w metodzie `render()`. Nadrzędną metodą `render()` znajduje się w obiekcie `ReactDOM` i zawiera następujący kod:

```
<div>
  <Letter>A</Letter>
  <Letter>E</Letter>
  <Letter>I</Letter>
  <Letter>O</Letter>
  <Letter>U</Letter>
</div>
```

Mamy tu kilka komponentów Letter osadzonych w elemencie div. To nic nadzwyczajnego. Metoda render() w komponencie Letter też nie różni się zbytnio od poprzedniej:

```
<div>
  {this.props.children}
</div>
```

Jak widzisz, każda litera jest umieszczona wewnętrz elementu div. Jeżeli rozwiniesz ten kod (na przykład wyświetlisz źródło strony w przeglądarce), zobaczysz ostateczną strukturę modelu DOM zawierającego nasze samogłoski, jak na rysunku 4.3.



Rysunek 4.3. Kod strony w przeglądarce

Rysunek przedstawia po prostu rozwinięcie różnych fragmentów kodów JSX użytych w metodach render(). Litery są osadzone wewnętrz kilku elementów div.

Nadajmy styl wreszcie!

Poznałeś kod HTML elementów, które zamierzasz stylizować, więc najtrudniejszą część masz już za sobą. Teraz czas na przyjemne i dobrze znane definiowane selektorów oraz specyfikowanie ich właściwości. Aby zmienić wygląd elementów osadzonych głęboko w znacznikach div, wpisz wewnątrz znaczników style następujący kod:

```
div div div {
  padding: 10px;
  margin: 10px;
  background-color: #FFDE00;
  color: #333;
  display: inline-block;
  font-family: monospace;
  font-size: 32px;
  text-align: center;
}
```

Selektor div div div definiuje styl odpowiednich elementów. Po jego zastosowaniu litery będą wyglądać dokładnie tak, jak tego chcesz. Jednak selektor ten wygląda trochę dziwnie, prawda? To dlatego, że jest zbyt ogólny. Jeżeli kod aplikacji zawiera więcej niż trzy zagnieżdżone znaczniki div (co się często zdarza), wtedy nieopatrznie można określić styl nie tych elementów, co trzeba. W przypadku takim jak nasz byłoby lepiej, gdyby biblioteka React generowała kod HTML trochę łatwiejszy do stylizowania.

Problem można rozwiązać, przypisując wewnętrznym elementom `div` klasę o nazwie `letter`. W tym miejscu składnia JSX różni się od HTML. Wprowadź w kodzie wyróżnioną niżej zmianę:

```
class Letter extends React.Component {  
  render() {  
    return (  
      <div className="letter">  
        {this.props.children}  
      </div>  
    );  
  }  
}
```

Zwróć uwagę, że nazwa klasy jest określona za pomocą atrybutu `className`, a nie `class`. Jest tak dlatego, że `class` jest słowem kluczowym w języku JavaScript. Jeżeli nie jest to dla Ciebie zrozumiałe, nie przejmuj się, zajmiemy się tym później.

Kiedy umieścimy w elemencie `div` atrybut `className` o wartości `letter`, pozostanie do zrobienia jeszcze jedna rzecz. Musimy zmienić selektor CSS tak, aby bardziej czytelnie wskazywał właściwy element `div`:

```
.letter {  
  padding: 10px;  
  margin: 10px;  
  background-color: #FFDE00;  
  color: #333;  
  display: inline-block;  
  font-family: monospace;  
  font-size: 32px;  
  text-align: center;  
}
```

Jak widzisz, za pomocą języka CSS można zupełnie poprawnie stylizować treść aplikacji opartej na bibliotece React. W następnej części rozdziału dowiesz się, jak stosować sposób preferowany przez tę bibliotekę.

Stylizowanie treści według React

W bibliotece React preferowane jest stylizowanie treści bezpośrednio w kodzie, bez użycia stylów CSS. Na pierwszy rzut oka może to wydawać się dziwne, ale w ten sposób można tworzyć bardziej uniwersalne elementy. Celem jest uzyskanie komponentów przypominających czarne skrzynki, w których jest schowane wszystko, co definiuje wygląd i działanie interfejsu użytkownika. Sprawdźmy, jak to wygląda w praktyce.

Wróćmy do naszego przykładu. Usuń z kodu styl `.letter`. Gdy otworzysz stronę w przeglądarce, stwierdzisz, że litery wróciły do stanu sprzed zastosowania stylów CSS. Aby całkowicie pozbyć się stylu, usuń również deklarację `className` z komponentu `Letter` w metodzie `render()`. Nie ma powodu, aby kod zawierał deklaracje, które nie są nieużywane.

Komponent `Letter` przywrócony do pierwotnego stanu powinien wyglądać jak niżej:

```
class Letter extends React.Component {  
  render() {  
    return (  
      <div>
```

```

        {this.props.children}
    </div>
);
}
}

```

Styl komponentu określisz, definiując wewnątrz niego obiekt, którego zawartością będą właściwości CSS wraz z wartościami. Obiekt ten przypiszesz za pomocą atrybutu style elementom, które zamierzasz stylizować. Operacja będzie bardziej zrozumiała, gdy oba kroki wykonasz samodzielnie. Zrób je więc, aby nadać styl komponentowi Letter.

Tworzenie obiektu stylizującego

Stwórzmy obiekt zawierający definicję stylu, który chcemy zastosować:

```

class Letter extends React.Component {
  render() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: "#FFDE00",
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: 32,
      textAlign: "center"
    };
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
}

```

Utworzyleś obiekt o nazwie letterStyle, zawierający zwykłe właściwości CSS i ich wartości. Jeżeli nie definiowałeś wcześniej właściwości CSS w kodzie JavaScript (za pomocą obiektu object.style), poznaj prostą zasadę konwertowania ich składni:

1. Jednowyrazowe właściwości CSS (padding, margin, color itp.) pozostawia się bez zmian.
2. Dwuwyrazowe właściwości zawierające myślnik (background-color, font-family, border-radius itp.) przekształca się w jeden wyraz, usuwając myślnik i zamieniając następującą po nim literę na wielką. W naszym przypadku właściwość background-color zamienia się na backgroundColor, font-family na fontFamily, a border-radius na borderRadius.

Obiekt letterStyle i jego właściwości są niemal wiernym tłumaczeniem na język JavaScript użytej wcześniej reguły .letter. Teraz wystarczy jedynie przypisać obiekt do elementu, który chcesz stylizować.

Właściwa stylizacja treści

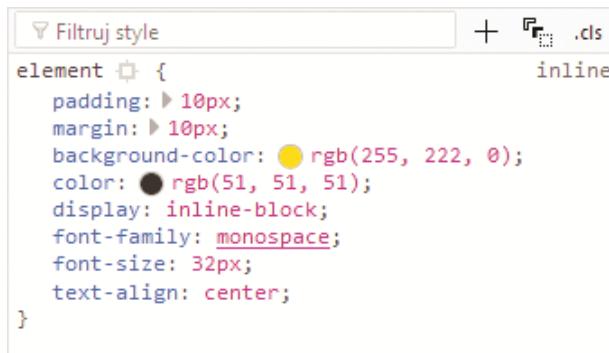
Kiedy zdefiniujesz obiekt zawierający styl, cała reszta będzie już prosta. Odszukaj element, któremu chcesz nadać styl, i przypisz jego atrybutowi style nazwę obiektu. W naszym przypadku elementem tym jest znacznik div zwracany przez metodę render() komponentu Letter.

Aby dowiedzieć się, co musisz zrobić, przyjrzyj się wyróżnionemu wierszowi:

```
class Letter extends React.Component {  
  render() {  
    var letterStyle = {  
      padding: 10,  
      margin: 10,  
      backgroundColor: "#FFDE00",  
      color: "#333",  
      display: "inline-block",  
      fontFamily: "monospace",  
      fontSize: 32,  
      textAlign: "center"  
    };  
  
    return (  
      <div style={letterStyle}>  
        {this.props.children}  
      </div>  
    );  
  }  
}
```

Obiekt nazywa się letterStyle. Został umieszczony w nawiasach klamrowych, aby biblioteka potraktowała go jak wyrażenie. To wszystko. Otwórz teraz stronę w przeglądarce i sprawdź, czy litery uzyskały żądany styl.

Dodatkowo stylizację możesz sprawdzić za pomocą dostępnych w przeglądarce narzędzi dla programistów. Przekonasz się, że faktycznie styl został zdefiniowany bezpośrednio w kodzie (patrz rysunek 4.4).



Rysunek 4.4. Styl zdefiniowany bezpośrednio w kodzie

Choć opisany sposób nie jest niczym nowym, może być dla Ciebie trudny do przyjęcia, jeżeli jesteś przyzwyczajony do definiowania stylów z pomocą reguł CSS. Jednak jak to się mówi: czasy się zmieniają.

Dostosowywanie koloru tła

Zanim podsumujemy rozdział, poznaj zalety stylizacji właściwej dla biblioteki React. Dzięki temu, że styl definiujemy w bezpośrednim sąsiedztwie kodu JSX, możemy łatwo dostosowywać różne jego cechy w elemencie nadzrędnym (tzw. konsumencie komponentu). Sprawdźmy, jak to wygląda w praktyce.

W tej chwili wszystkie litery mają żółte tło. Czy nie byłoby fajnie, gdyby definicja koloru była częścią deklaracji komponentu `Letter`? Aby to osiągnąć, najpierw dodaj w metodzie `ReactDOM.render()` atrybut `bgcolor` i zdefiniuj kilka kolorów, jak w wyróżnionych niżej wierszach:

```
ReactDOM.render(
  <div>
    <Letter bgcolor="#58B3FF">A</Letter>
    <Letter bgcolor="#FF605F">E</Letter>
    <Letter bgcolor="#FFD52E">I</Letter>
    <Letter bgcolor="#49DD8E">O</Letter>
    <Letter bgcolor="#AE99FF">U</Letter>
  </div>,
  destination
);
```

Następnie zastosuj nowy atrybut. W tym celu w obiekcie `letterStyle` przypisz właściwości `backgroundColor` właściwość `this.props.bgcolor`:

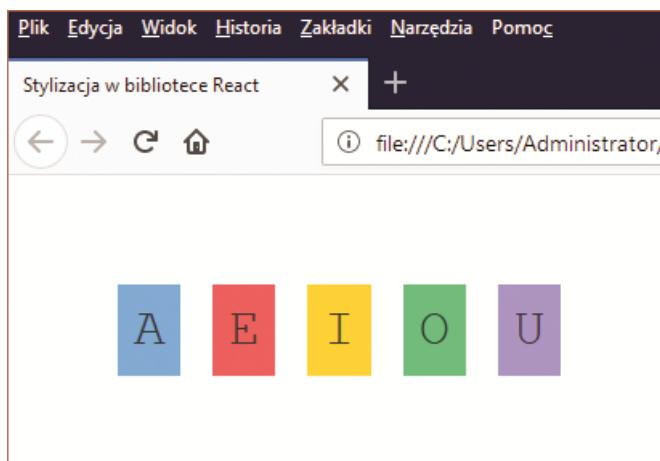
```
var letterStyle = {
  padding: 10,
  margin: 10,
  backgroundColor: this.props.bgcolor,
  color: "#333",
  display: "inline-block",
  fontFamily: "monospace",
  fontSize: 32,
  textAlign: "center"
};
```

W ten sposób właściwości `backgroundColor` będzie przypisywana wartość atrybutu `bgcolor` użytego w deklaracji komponentu `Letter`. Gdy otworzysz stronę w przeglądarce, zobaczyś te same litery na różnych kolorowych tła (patrz rysunek 4.5).

To, co tutaj zrobiliśmy, byłoby bardzo trudno osiągnąć za pomocą zwykłych reguł CSS. Gdy zaczniemy zajmować się komponentami, których zawartość będzie się zmieniać w zależności o ich stanów lub od wykonywanych przez użytkownika operacji, poznasz więcej dowodów, że stylizacja stosowana w bibliotece React jest znacznie lepsza.

Podsumowanie

W miarę jak coraz głębiej będziesz poznawał bibliotekę React, zobaczyś, że dużo rzeczy, które powszechnie są uznawane za słuszne, za pomocą tej biblioteki robi się zupełnie inaczej. W tym rozdziale dowiedziałeś się, że React preferuje definiowanie stylów bezpośrednio w kodzie JavaScript, a nie za pomocą reguł CSS. Wcześniej poznaleś język JSX i dowiedziałeś się, jak dzięki niemu deklaruje się elementy interfejsu użytkownika, wykorzystując składnię JavaScript i znaczniki podobne do tych stosowanych w kodzie HTML.



Rysunek 4.5. Twoje litery na różnokolorowych tłaach!

Gdy z czasem zagłębisz się w szczegóły, przekonasz się, że filozofia biblioteki React, tak różna od konwencjonalnego podejścia, ma głęboki sens. Tworzenie aplikacji ze skomplikowanymi interfejsami wymaga mierzenia się z nowymi wyzwaniami. Techniki wykorzystujące kod HTML, CSS i JavaScript, doskonale sprawdzające się w tworzeniu stron i dokumentów WWW, mogą okazać się zupełnie nieprzydatne w aplikacjach WWW, w których komponenty są wykorzystywane wewnętrz innych komponentów.

Dlatego musisz umiejętnie wybierać techniki, które są najważniejsze w danej sytuacji. Jestem zwolennikiem tego, aby do rozwiązywania problemów, które pojawiają się podczas tworzenia interfejsu użytkownika, stosować podejście charakterystyczne dla biblioteki React, ale staram się również wykorzystywać alternatywne lub konwencjonalne sposoby. Łączenie stylów CSS z kodem React jest całkowicie poprawne, pod warunkiem że jest się świadomym tego, co dzięki temu się zyskuje, a co traci.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

5

Tworzenie złożonych komponentów

W rozdziale 3. „Komponenty biblioteki React” poznaleś komponenty i niezwykłe rzeczy, które można przy ich użyciu robić. Dowiedziałeś się, że są to podstawowe „cegielki” zawierające kody HTML i JavaScript, definiujące wygląd i działanie elementów interfejsu użytkownika. Komponenty oprócz swojej uniwersalności mają jeszcze inną ważną zaletę: można je ze sobą łączyć i budować w ten sposób coraz bardziej złożone komponenty.

W tym rozdziale dowiesz się, co to dokładnie oznacza. A konkretniej, to zajmiemy się dwiema rzecząmi:

- nudnymi, technicznymi podstawami, które musisz znać;
- nudną wiedzą, którą musisz posiąść, aby móc wyodrębniać komponenty z masy wizualnych elementów.

OK, tak naprawdę to rzeczy, których się nauczysz, nie będą takie nudne. Chciałbym po prostu, abyś nie oczekivał zbyt wiele.

Od elementów interfejsu do komponentów

Opisane do tej pory przykłady były bardzo proste. Doskonale posłużyły do wyjaśnienia różnych technicznych kwestii, ale nie przygotowywały dobrze do realizacji praktycznych zadań.



W realnym świecie będziesz za pomocą biblioteki React implementował rzeczy, które na pewno nie będą tak proste, jak lista imion czy litery na różnokolorowym tle. Będziesz musiał definiować skomplikowane interfejsy graficzne na podstawie różnych gryzmołów, diagramów, filmów, zrzutów ekranu i innych rozmaistości. Od Ciebie będzie zależało, jak ożywić te statyczne piksele. W tym rozdziale nabierzesz nieco niezbędnej do tego celu praktyki.

Twoim pierwszym zadaniem będzie utworzenie prostej karty z paletą kolorów (patrz rysunek 5.1).



Rysunek 5.1. Prosta karta z paletą kolorów

Karta z paletą kolorów jest to mały prostokątny kawałek plastiku służący do porównywania barw. Takie palety są wykorzystywane w sklepach z wyposażeniem wnętrz lub z farbami. Twój kolega zajmujący się projektowaniem aplikacji na pewno posiada ogromny zestaw takich kart. Zadanie polega na utworzeniu jednej karty przy użyciu biblioteki React.

Można to zrobić na kilka sposobów, ale tu zastosujemy systematyczne, upraszczające podejście, sprawdzające się nawet w przypadku najbardziej skomplikowanych interfejsów. Wyróżnia się w nim dwa etapy:

1. określenie głównych elementów wizualnych,
2. określenie potrzebnych komponentów.

Oba etapy mogą Ci się wydawać naprawdę trudne, ale w miarę robienia postępów przekonasz się, że nie ma powodów do obaw.

Określenie głównych elementów wizualnych

Pierwszym krokiem jest określenie elementów wizualnych, nad którymi będziesz pracował. Żaden element nie jest na tyle błały, aby móc go pominąć, przynajmniej na początku.

Najprostszym sposobem określenia potrzebnych składników jest wyróżnienie podstawowych elementów, a następnie podzielenie ich na bardziej i mniej podstawowe.

Rzeczą, którą w naszym przypadku widać na pierwszy rzut oka, jest sama karta (patrz rysunek 5.2).



Rysunek 5.2. Karta koloru

Na karcie można wyróżnić dwa oddzielne obszary. Górnny obszar jest kwadratem wypełnionym określonym kolorem. Dolny obszar jest białym prostokątem zawierającym szesnastkowy kod koloru.

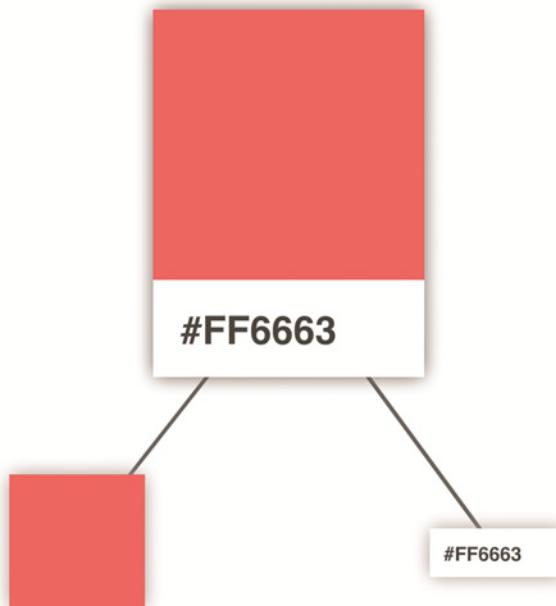
Wyodrębnijmy te dwa elementy wizualne i utwórzmy z nich drzewiastą strukturę, jak na rysunku 5.3.

Tworzenie drzewiastej struktury (czyli **hierarchii**) elementów jest dobrym sposobem opisywania tego, jak są grupowane te elementy. Celem tego ćwiczenia jest określenie istotnych elementów wizualnych, a następnie systematyczne dzielenie ich na elementy nadzadane i podrzędne aż do momentu, gdy nie będzie ich można dalej dzielić.

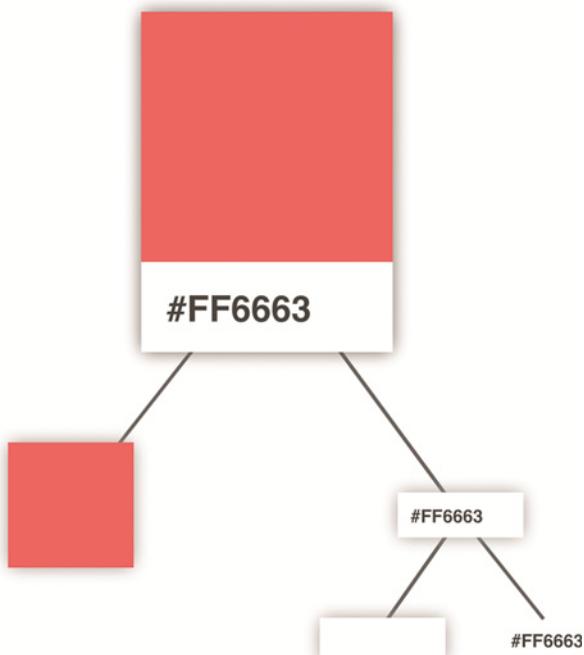
Staraj się nie myśleć o szczegółach implementacyjnych

Choć może to być trudne, nie myśl na razie o szczegółach implementacyjnych. Dzieląc elementy, nie sugeruj się możliwościami łączenia odpowiednich fragmentów kodów HTML i CSS. Będziesz miał na to mnóstwo czasu później.

Wróćmy do naszego zadania. Jak widać, kolorowego kwadratu nie da się dalej podzielić. Nie oznacza to jednak, że na tym koniec. Można podzielić biały obszar z etykietą. Teraz hierarchia elementów wygląda tak jak na rysunku 5.4. Etykieta i biały prostokąt tworzą osobną gałąź drzewa.



Rysunek 5.3. Drzewiasta struktura elementów



Rysunek 5.4. Element podzielony dodatkowo na biały obszar i etykietę

W tym momencie nie da się już dalej dzielić elementu karty. Zakończyliśmy etap określania i dzielenia elementów wizualnych, zatem następnym krokiem jest wykorzystanie efektów naszej pracy do zdefiniowania komponentów.

Określenie potrzebnych komponentów

Ten etap jest nieco ciekawszy. Trzeba w nim określić, które elementy wizualne zamienimy na komponenty, a które nie. Nie należy zamieniać wszystkich elementów, jak również nie można tworzyć pojedynczych, bardzo skomplikowanych komponentów. Trzeba tu znaleźć równowagę (patrz rysunek 5.5).



Rysunek 5.5. Komponentów nie może być ani za mało, ani za dużo

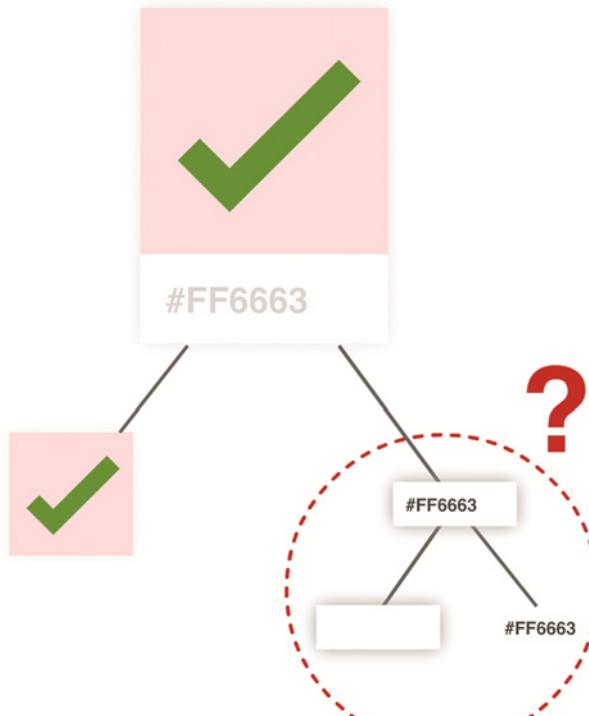
Podejmowanie decyzji, które elementy będą, a które nie będą częściami komponentu, jest nie lada sztuką. Obowiązuje tu ogólna zasada, że jeden komponent powinien realizować *tylko jedną funkcjonalność*. Jeżeli stwierdzisz, że potencjalny komponent będzie realizował zbyt wiele funkcjonalności, prawdopodobnie będziesz musiał podzielić go na kilka mniejszych komponentów. Natomiast jeżeli potencjalny komponent będzie zbyt mały, prawdopodobnie będziesz mógł go pominąć i w ogóle nie tworzyć.

Spróbujmy określić elementy, które w naszym przykładzie nadają się do zmienienia w komponenty. Patrząc na hierarchię elementów, możemy stwierdzić, że zarówno karta, jak i kolorowy kwadrat wydają się dobrymi kandydatami na komponenty. Karta będzie zewnętrznym komponentem, a kwadrat będzie służył po prostu do prezentowania koloru.

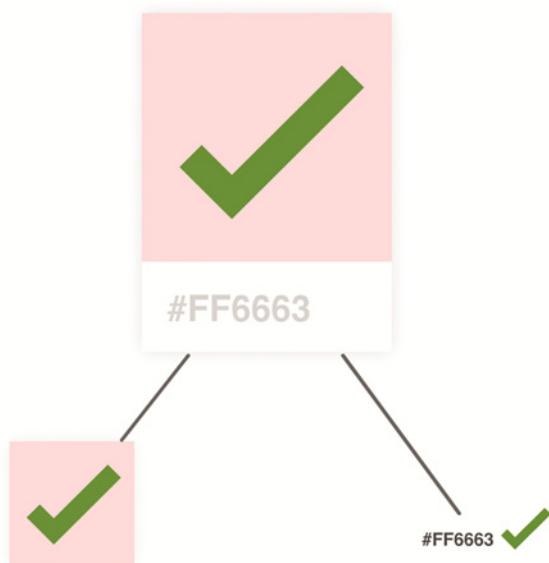
Pozostaje jeszcze kwestia białego obszaru i etykiety (patrz rysunek 5.6).

Etykieta jest ważną częścią karty. Bez niej nie byłby widoczny szesnastkowy kod. Pozostaje jeszcze biały obszar. Ma on niewielkie znaczenie. Jest to zwykły pusty obszar i jego rolę może przejąć etykieta. Przygotuj się na to, co teraz powiem: niestety, białego obszaru nie będziemy przekształcać w komponent.

W tym momencie mamy określone trzy komponenty tworzące strukturę przedstawioną na rysunku 5.7.



Rysunek 5.6. Co zrobić z białym obszarem i etykietą?



Rysunek 5.7. Trzy komponenty

Musisz pamiętać o ważnej sprawie: hierarchia komponentów ma pomagać w tworzeniu kodu i nie może definiować ostatecznego wyglądu aplikacji. Przekonasz się, że interfejs będzie wyglądał nieco inaczej niż utworzona hierarchia. W kwestiach wizualnych zawsze odnoś się do oryginalnych materiałów (szkiców, rzutów ekranu, schematów itp.). Hierarchii komponentów używaj przy podejmowaniu decyzji dotyczących wyboru komponentów do utworzenia.

OK, zdefiniowaliśmy komponenty i relacje między nimi, czas więc zacząć tworzyć naszą kartę palety kolorów.

Tworzenie komponentów

To będzie prosta część... w pewnym sensie. Przede wszystkim utwórz pustą stronę HTML, która będzie stanowiła punkt wyjścia:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Zawansowane komponenty</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>

  <script type="text/babel">
    ReactDOM.render(
      <div>

        </div>,
        document.querySelector("#container")
      );
  </script>
</body>

</html>
```

Popatrz przez chwilę, co zawiera ta strona. Nie jest tego wiele: niezbędne minimum, aby biblioteka React mogła wyświetlić pusty element div wewnątrz elementu container.

Po utworzeniu pliku pora na zdefiniowanie naszych trzech komponentów. Nazwiemy je Card (karta), Label (etykieta) i Square (kwadrat). Tuż nad metodą ReactDOM.render() wpisz poniższy kod:

```
class Square extends React.Component {
  render() {
    return(
```

```

        <br/>
    );
}
}

class Label extends React.Component {
  render() {
    return (
      <br/>
    );
  }
}

class Card extends React.Component {
  render() {
    return (
      <br/>
    );
  }
}

```

Powyższy kod nie tylko definiuje trzy komponenty, lecz także umieszcza wewnętrz każdej metody render() coś, co jest niezbędne do funkcjonowania każdego z komponentów. Każda metoda zwraca element br. Gdyby metoda nie zawierała żadnego kodu, wtedy po uruchomieniu aplikacji pojawiłby się błąd. Jeżeli pominiemy ten jeden element, to każdy komponent jest zupełnie pusty. W następnej części rozdziału zajmiemy się tym i coś w komponentach umieścimy.

Komponent Card

Zacznijmy od komponentu Card, znajdującego się na szczytce hierarchii. Komponent ten będzie pełnił funkcję kontenera, w którym zostaną umieszczone komponenty Square i Label.

Aby zaimplementować komponent Card, wprowadź w kodzie wyróżnione niżej zmiany:

```

class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return (
      <div style={cardStyle}>
        </div>
    );
  }
}

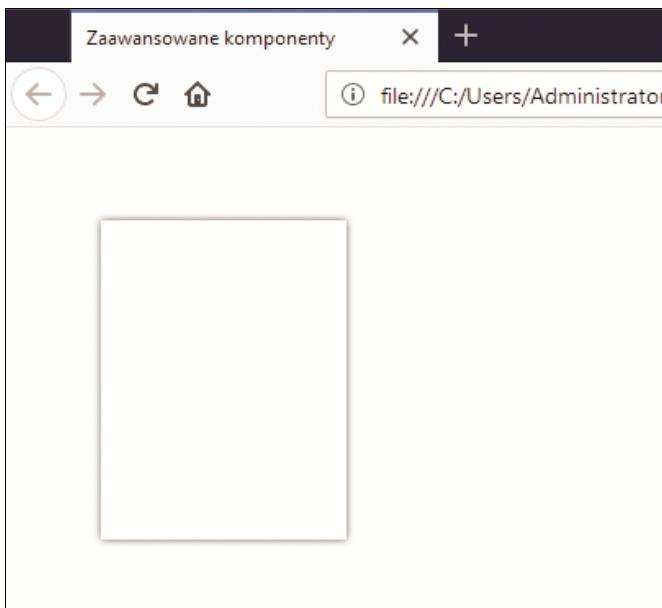
```

Zmiany wyglądają na całkiem spore, ale większość z nich dotyczy stylizacji komponentu Card za pomocą obiektu cardStyle. Pozostałe modyfikacje są mało istotne. Metoda render() zwraca element div z atrybutem style zawierającym obiekt cardStyle. Aby zobaczyć komponent

w akcji, musisz go wyświetlić w modelu DOM za pomocą metody `ReactDOM.render()`. W tym celu wprowadź w kodzie wyróżnioną niżej zmianę:

```
ReactDOM.render(
  <div>
    <Card/>
  </div>,
  document.querySelector("#container")
);
```

Wprowadzony wiersz powoduje, że metoda `ReactDOM.render()` wywołuje komponent `Card` i wyświetla jego zawartość. Jeżeli się nie pomyliłeś, po otwarciu strony w przeglądarce powinieneś zobaczyć widok identyczny z przedstawionym na rysunku 5.8.



Rysunek 5.8. Efekt naszych starań: kontur karty z palety kolorów

Tak, na razie jest to tylko kontur karty z palety kolorów, ale to i tak o wiele więcej od tego, co było kilka chwil wcześniej!

Komponent Square

Czas zejść w naszej hierarchii jeden poziom niżej i zająć się komponentem `Square`. Jest on bardzo prosty. Aby go zdefiniować, wystarczy wprowadzić następujące zmiany:

```
class Square extends React.Component {
  render() {
    var squareStyle = {
      height: 150,
      backgroundColor: "#FF6663"
```

```

    };

    return (
      <div style={squareStyle}>
        </div>
    );
}
}

```

Metoda render(), podobnie jak to było w komponencie Card, zwraca element `div` z atrybutem `style`, którego wartością jest obiekt `squareStyle` definiujący wygląd komponentu. Aby zobaczyć komponent `Square` w akcji, trzeba go umieścić w modelu DOM w podobny sposób jak wcześniej komponent `Card`. Tym razem różnica polega jedynie na tym, że komponent `Square` należy wywołać za pomocą metody `render()` komponentu `Card`, a nie metody `ReactDOM.render()`. Aby dowiedzieć się, co mam na myśli, wróć do metody `render()` komponentu `Card` i wprowadź w niej poniższą zmianę:

```

class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };
    return (
      <div style={cardStyle}>
        <Square />
      </div>
    );
  }
}

```

Jeżeli teraz otworzysz stronę w przeglądarce, zobaczyś kolorowy kwadrat (patrz rysunek 5.9).

Ciekawostką, na którą warto zwrócić uwagę, jest to, że komponent `Square` jest wywoływany z wnętrza komponentu `Card`! Jest to przykład możliwości łączenia dwóch komponentów, gdzie wygląd jednego komponentu zależy od wyglądu innego. Ostateczny widok jest efektem współpracy dwóch komponentów. Czyż taka współpraca nie jest piękna, przynajmniej w tym kontekście?

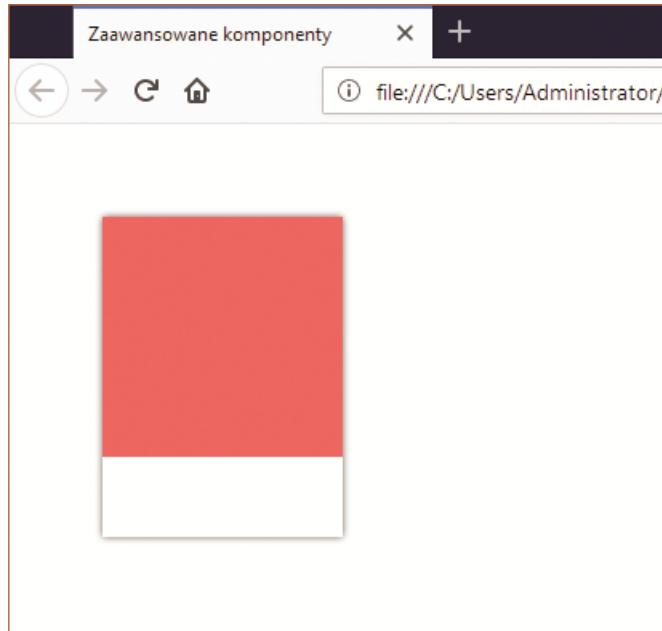
Komponent Label

Ostatnim komponentem jest `Label`. Wprowadź w kodzie wyróżnione niżej zmiany:

```

class Label extends React.Component {
  render() {
    var labelStyle = {
      fontFamily: "sans-serif",
      fontWeight: "bold",
    };
  }
}

```



Rysunek 5.9. Karta z czerwonym obszarem

```

padding: 13,
margin: 0
};

return (
  <p style={labelStyle}>#FF6663</p>
);
}
}

```

To, co teraz zrobileś, powinno być już dla Ciebie rutyną. Zdefiniowałeś obiekt `labelStyle` określający styl elementu `p` zwracanego przez metodę `render()`. Element ten zawiera tekst `#FF6663`. Aby zwracany element umieścić w modelu DOM, należy wywołać komponent `Label` za pomocą komponentu `Card`. W tym celu wprowadź w kodzie wyróżnioną niżej zmianę:

```

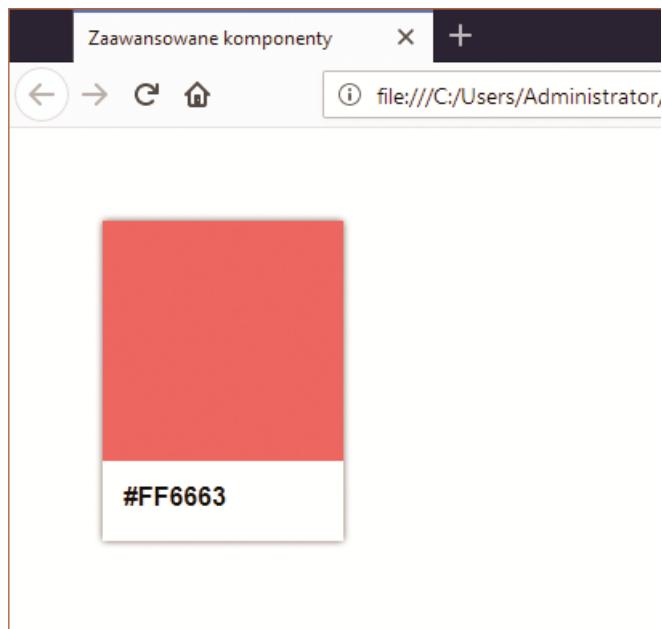
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return (
      <div style={cardStyle}>
        <Square />
        <Label />
      </div>
    );
  }
}

```

```
        </div>
    );
}
```

Zwróć uwagę, że komponent Label wstawiliś tuż poniżej komponentu Square umieszczonego wcześniej w argumencie funkcji return(). Gdy teraz otworzysz stronę w przeglądarce, zobaczysz widok przedstawiony na rysunku 5.10.



Rysunek 5.10. Karta z etykietą

Tak, to jest właśnie to! Nasza karta z palety kolorów została utworzona i wyświetlona za pomocą komponentów Card, Square i Label. Nie oznacza to jednak, że skończyliśmy. Jest jeszcze kilka rzeczy do omówienia.

Znowu przekazywanie właściwości!

W naszym przykładzie kod koloru wykorzystywany przez komponenty Square i Label jest zapisany w kodzie na stałe. Jest to dość nieeleganckie rozwiązanie, ale być może zostało zastosowane rozmyślnie dla uzyskania wyrazistego efektu. Można to łatwo naprawić. W tym celu trzeba zdefiniować właściwość o wybranej nazwie i odwołać się do niej za pomocą właściwości this.props. Ten sposób już znasz. Jedyną różnicą w tym przypadku będzie częstotliwość jego stosowania.

Nie da się *poprawnie* zdefiniować właściwości w komponencie nadrzędnym, tak aby wszystkie komponenty potomne automatycznie uzyskały do niej dostęp. Jest kilka mało praktycznych sposobów osiągnięcia tego celu, na przykład definiowanie globalnego obiektu i bezpośrednie przypisywanie wartości jego właściwościom. Nie będziemy się jednak zajmować takimi nieeleganckimi rozwiązaniami. Nie jesteśmy amatorami!

Poprawny sposób przekazywania wartości właściwości do komponentów znajdujących się na samym dole hierarchii polega na użyciu właściwości wszystkich komponentów pośrednich. Aby dowiedzieć się jak to wygląda w praktyce, przyjrzyjmy się wyróżnionym niżej zmianom. Z komponentu został usunięty stały kod koloru i zamiast niego zdefiniowana właściwość color:

```
class Square extends React.Component {
  render() {
    var squareStyle = {
      height: 150,
      backgroundColor: this.props.color
    };

    return (
      <div style={squareStyle}>
        </div>
      );
    }
}

class Label extends React.Component {
  render() {
    var labelStyle = {
      fontFamily: "sans-serif",
      fontWeight: "bold",
      padding: 13,
      margin: 0
    };

    return (
      <p style={labelStyle}>{this.props.color}</p>
    );
  }
}

class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return (
      <div style={cardStyle}>
        <Square color={this.props.color} />
        <Label color={this.props.color} />
      </div>
    );
  }
}

ReactDOM.render(
  <div>
    <Card color="#FF6663" />
  </div>,
  document.querySelector("#container")
);
```

Po wprowadzeniu powyższych zmian możesz w wywołaniu komponentu Card wpisać kod szesnastkowy dowolnego koloru:

```
ReactDOM.render(  
  <div>  
    <Card color="#FFA737"/>  
  </div>,  
  document.querySelector("#container")  
)
```

Efektem zmian jest karta w zadanym kolorze (patrz rysunek 5.11).



Rysunek 5.11. Karta w kolorze o kodzie #FFA737

Wróćmy jeszcze do wprowadzonych zmian. Właściwość color została wykorzystana tylko w komponentach Square i Label, a przypisywanie jej wartości jest zadaniem nadzielnego komponentu Card. Bardziej rozbudowane hierarchie zawierają więcej pośrednich komponentów, których zadaniem jest przekazywanie wartości właściwości. Co gorsza, jeżeli wykorzystywanych jest wiele właściwości, które muszą być przekazywane do komponentów znajdujących się na różnych poziomach hierarchii, trzeba wpisywać (albo kopować i wklejać) więcej kodu. Istnieją rozwiązania tego problemu, którym przyjrzymy się dokładniej w następnym rozdziale.

Dlaczego możliwość łączenia komponentów jest super?

W naszym zachwycie nad biblioteką React często zapominamy, że finalnym produktem jest zwykły, nudny kod HTML, CSS i JavaScript. Kod HTML naszej karty kolorów wygląda następująco:

```
<div id="container">
  <div>
    <div style="height: 200px;
      width: 150px;
      padding: 0px;
      background-color: rgb(255, 255, 255);
      box-shadow: rgb(102, 102, 102) 0px 0px 5px;">
      <div style="height: 150px;
        background-color: rgb(255, 102, 99);">
        </div>
      <p style="font-family: sans-serif;
        font-weight: bold;
        padding: 13px;
        margin: 0px;">
        #FF6663</p>
    </div>
  </div>
</div>
```

Zupełnie nie wiadomo, o co w nim chodzi. Trudno jest dociec, jakie funkcje pełnią poszczególne elementy. Nie ma tu śladu po łączeniu komponentów ani żmudnym przekazywaniu właściwości `color` z komponentu nadziednego do podrzędnego. Płynie stąd ważny wniosek.

Ostateczny efekt użycia komponentu można ogólnie określić jako masę kodu HTML definiującego żądaną funkcjonalność. Metoda `render()` każdego komponentu przekazuje wynikowy kod HTML metodzie `render()` innego komponentu. Przekazywane kody kumulują się i powstaje ogromny dokument HTML umieszczany następnie (bardzo skutecznie) w modelu DOM. Dzięki temu komponenty są tak uniwersalne i tak łatwo można je ze sobą łączyć. Fragmenty kodu HTML funkcjonują niezależnie od siebie, szczególnie jeżeli stosuje się zalecane w przypadku biblioteki React osadzane `style`. Można w ten sposób łatwo składać elementy interfejsu z innych elementów i nie martwić się o nic. Zupełnie o nic! Czy to nie jest genialne?

Podsumowanie

Jak już się przekonałeś, stopniowo przechodzimy do coraz bardziej zaawansowanych funkcjonalności oferowanych przez bibliotekę React. Na razie słowo *zaawansowanych* jest nieco na wyrost. Bardziej odpowiednim jest *praktycznych*. Na początku tego rozdziału przeanalizowaliśmy elementy interfejsu użytkownika, wyodrębniliśmy i zaimplementowaliśmy komponenty. Z takimi sytuacjami będziesz miał do czynienia nieustannie. Zastosowane podejście wygląda na bardzo formalne, ale w miarę nabywania doświadczenia w tworzeniu interfejsów za pomocą biblioteki React będziesz mógł pozwolić siebie na ominięcie tych formalizmów. Gdy będziesz w stanie szybko określać komponenty i zależności pomiędzy nimi bez rysowania hierarchii, będzie to kolejny znak, że potrafisz naprawdę skutecznie posługiwać się biblioteką React.

Określanie komponentów to dopiero połowa sukcesu. Drugą połową jest ich praktyczne implementowanie. Większość technicznych zagadnień, które zostały tu omówione, stanowi zaledwie niewielki dodatek do tego, czego nauczyłeś się wcześniej. W poprzednim rozdziale miałeś do czynienia z jednym poziomem komponentów, tutaj poznajeś komponenty wielopoziomowe. Wcześniej nauczyłeś się przekazywać właściwości pomiędzy jedną parą komponentów, teraz dowiedziałeś się, jak to się robi w przypadku wielu komponentów

nadrzędnych i podrzędnych. Być może w następnym rozdziale zrobimy coś przełomowego, na przykład umieścimy na stronie wiele kart z paletą kolorów? Albo zdefiniujemy dwie właściwości zamiast jednej? Kto to wie?

Jeżeli napotkasz problemy, pyтай!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pyтай сміаło! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

6

Przekazywanie właściwości

Praca z właściwościami może być frustrująca. Doświadczyłeś tego nieco w poprzednim rozdziale. Przekazywanie właściwości z jednego komponentu do drugiego jest prostą i przyjemną operacją, pod warunkiem że obejmuje jedną warstwę komponentów. Rzecz zaczyna się komplikować, gdy trzeba przekazywać właściwości pomiędzy wieloma warstwami.

Komplikacje nigdy nie są czymś pożądanym, dlatego w tym rozdziale dowiesz się, co można zrobić, aby praca z właściwościami była prosta nawet w przypadku, gdy komponenty tworzą kilka warstw.

Opis problemu

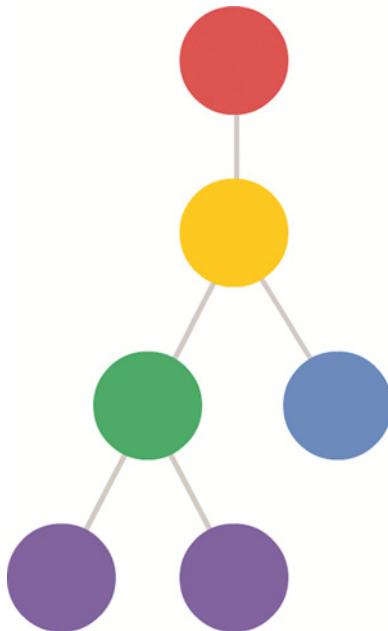
Załóżmy, że mamy komponent składający się z wielu zagnieżdżonych w nim innych komponentów. Jego hierarchiczna struktura (zobrazowana za pomocą kolorowych kół) wygląda tak jak na rysunku 6.1.

Chcemy zaimplementować przekazywanie właściwości z komponentu reprezentowanego przez czerwone koło (na samej górze hierarchii) do komponentów fioletowych (na samym dole), gdzie będą wykorzystywane. Na pewno nie da się tego osiągnąć w oczywisty i prosty sposób przedstawiony na rysunku 6.2.

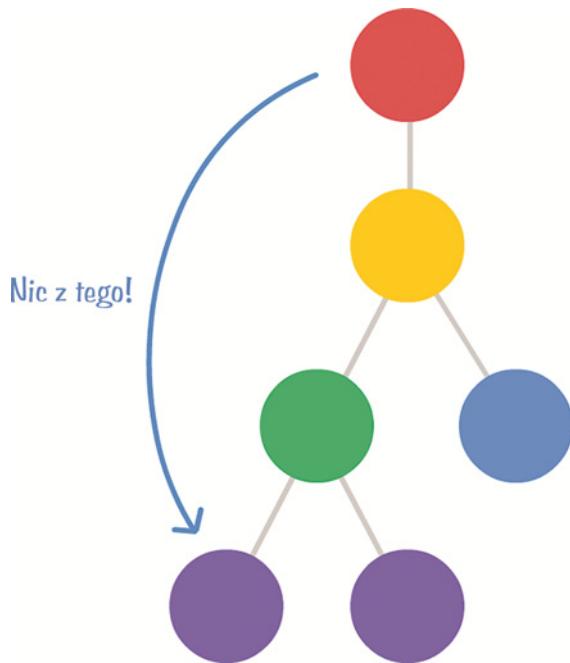
Nie można przekazywać właściwości do jednego lub kilku docelowych komponentów z pominięciem hierarchii. Powodem jest sposób, w jaki funkcjonuje biblioteka React.

Wymaga ona, aby właściwości były przekazywane z komponentu nadzawanego do komponentu bezpośrednio mu podległego. Oznacza to, że nie można przeskakiwać całych warstw komponentów. Ponadto komponent podrzędny nie może przekazywać właściwości do komponentu nadzawanego. Komunikacja jest wyłącznie jednokierunkowa — od rodzica do dziecka.

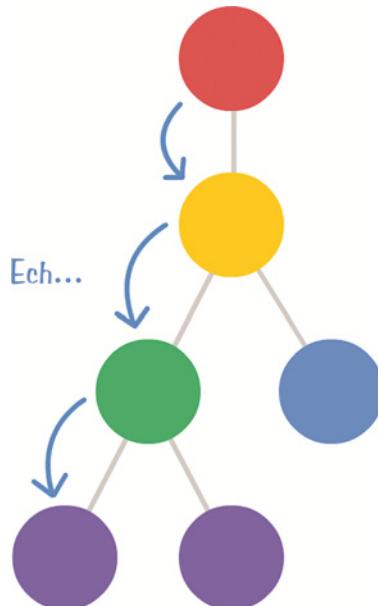
Ze względu na opisane wymogi proces przekazywania właściwości z komponentu najwyższego w hierarchii do najniższego wygląda tak jak na rysunku 6.3.



Rysunek 6.1. Hierarchiczna struktura komponentu



Rysunek 6.2. Tak nie można



Rysunek 6.3. Komponent nadzędny przekazuje właściwość komponentowi podzdnemu

Każdy pośredni komponent musi odbierać właściwość od swego komponentu nadzdnego i przekazywać ją swojemu komponentowi podzdnemu. Proces ten jest kontynuowany do chwili, aż właściwość dotrze do swego miejsca przeznaczenia. Problem polega na odbieraniu i wysyłaniu właściwości.

Ścieżka, którą podąża właściwość o nazwie `color` z komponentu na górze do docelowego komponentu na dole hierarchii, wygląda tak jak na rysunku 6.4.

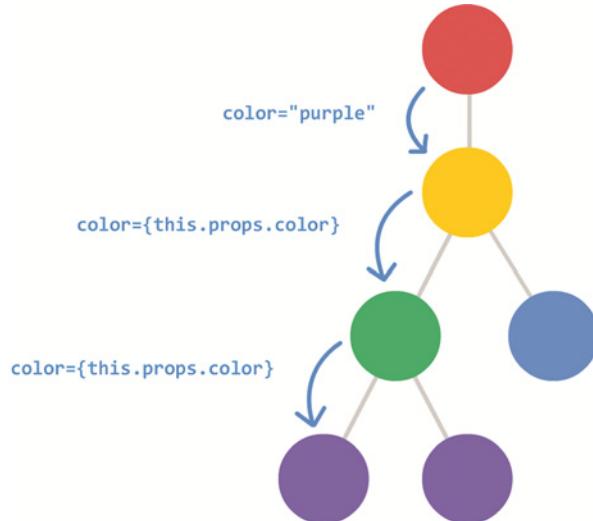
Wyobraźmy sobie teraz, że trzeba przekazać dwie właściwości, jak na rysunku 6.5.

A jeżeli trzeba przekazać trzy właściwości? Albo cztery?

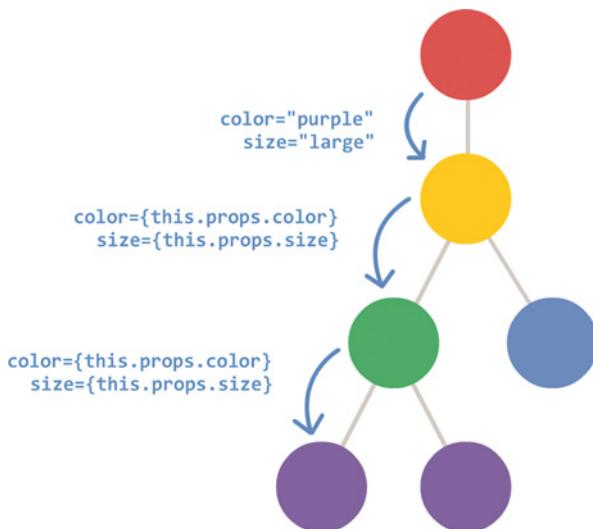
Jak widzisz, opisane rozwiązanie nie jest ani skalwalne, ani proste w realizacji. Każdą kolejną przekazywaną właściwość należy zadeklarować w komponencie. Gdyby z jakiegoś powodu trzeba było zmienić nazwę właściwości, należałoby to zrobić we wszystkich komponentach. Jeżeli właściwość została usunięta, trzeba byłoby usunąć ją również ze wszystkich komponentów, które ją przekazują. Są to sytuacje, do których powstania nie można dopuścić, pisząc kod. Co więc można zrobić?

Szczegółowy opis problemu

W poprzedniej części rozdziału został ogólnie opisany problem. Zanim zaczniemy szukać rozwiązania, musimy odłożyć na bok diagramy i dokładniej przyjrzeć się przykładowemu kodowi. Przeanalizujmy następujący fragment:



Rysunek 6.4. Przekazywanie właściwości color



Rysunek 6.5. Przekazywanie dwóch właściwości

```
class Display extends React.Component {  
  render() {  
    return (  
      <div>  
        <p>{this.props.color}</p>  
        <p>{this.props.num}</p>  
        <p>{this.props.size}</p>  
      </div>  
    );  
  }  
}
```

```

class Label extends React.Component {
  render() {
    return (
      <Display color={this.props.color}
                num={this.props.num}
                size={this.props.size}/>
    );
  }
}

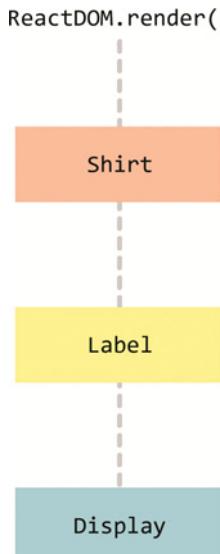
class Shirt extends React.Component {
  render() {
    return (
      <div>
        <Label color={this.props.color}
                  num={this.props.num}
                  size={this.props.size}/>
      </div>
    );
  }
}

ReactDOM.render(
  <div>
    <Shirt color="jasnoniebieski" num="3,14" size="średni" />
  </div>,
  document.querySelector("#container")
);

```

Zastanówmy się przez chwilę, co w tym kodzie się dzieje. Potem wspólnie zajmiemy się przykładem.

Mamy komponent `Shirt` wykorzystujący treść generowaną przez komponent `Label`, który z kolei wykorzystuje treść generowaną przez komponent `Display`. Rysunek 6.6 przedstawia hierarchię komponentów.



Rysunek 6.6. Hierarchia komponentów

Wynik działania tego kodu nie jest niczym nadzwyczajnym. Są to trzy wiersze tekstu, jak na rysunku 6.7.



Rysunek 6.7. Wynik działania kodu

Ciekawą kwestią jest, skąd wzięły się te napisy. Każdy z nich jest określony w osobnej właściwości wewnętrz metody `ReactDOM.render()`:

```
<Shirt color="jasnoniebieski" num="3,14" size="średni" />
```

Właściwości `color`, `num` i `size` (wraz ze swoimi wartościami) przemierają do komponentu `Display` drogą, której mógłby pozazdrościć najbardziej wytrwały podróżnik. Prześledźmy, co się dzieje z tymi właściwościami począwszy od ich powstania do zastosowania. (Jestem przekonany, że większość opisanych niżej rzeczy będzie Ci już znana. Gdybyś zaczął się nudzić, możesz śmiało przejść do następnej części rozdziału).

Nasze właściwości `color`, `num` i `size` rozpoczynają swoje życie wewnętrz metody `ReactDOM.render()`, gdzie są umieszczane w wywoływany komponencie `Shirt`:

```
ReactDOM.render(
  <div>
    <Shirt color="jasnoniebieski" num="3,14" size="średni" />
  </div>,
  document.querySelector("#container")
);
```

Właściwości są tu nie tylko definiowane, lecz także inicjowane odpowiednimi wartościami.

Wewnątrz komponentu `Shirt` właściwości są umieszczane w obiekcie `props`. Aby móc je przekazać, trzeba się do nich jawnie odwołać za pomocą obiektu `props` i umieścić w kolejnym wywoływany komponencie. Poniżej znajduje się kod komponentu `Shirt` z wyróżnionym fragmentem, w którym wywoływany jest komponent `Label`:

```
class Shirt extends React.Component {
  render() {
    return (
      <div>
        <Label color={this.props.color}>
          num={this.props.num}
          size={this.props.size} />
    
```

```
        </div>
    );
}
```

Zwróć uwagę, że ponownie zostały tu użyte właściwości `color`, `num` i `size`. Powyższa metoda `render()` różni się od `ReactDOM.render()` jedynie tym, że wartości poszczególnych właściwości nie są zakodowane na stałe, tylko są odczytywane z odpowiednich pól obiektu `props`.

Obiekt `props` komponentu `Label`, gdy ten już powstanie, zawiera właściwości `color`, `num` i `size`. Prawdopodobnie widzisz już schemat, według którego działa ten kod. Jeżeli masz ochotę głośno ziewnąć, proszę bardzo.

Komponent `Label` kontynuuje opisany wyżej proces i wykonuje te same operacje, wywołując komponent `Display`:

```
class Label extends React.Component {
  render() {
    return (
      <Display color={this.props.color}
                num={this.props.num}
                size={this.props.size} />
    );
  }
}
```

Zwróć uwagę, że w wywołaniu komponentu `Display` znajduje się ta sama lista właściwości co poprzednio w komponencie `Label` i że są im przypisywane wartości zapisane w obiekcie `props`. W tym momencie jedyną dobrą wiadomością jest to, że to już prawie koniec. Komponent `Display` wyświetla jedynie właściwości zapisane w jego obiekcie `props`:

```
class Display extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
}
```

Uff! Naszym jedynym celem było wyświetlenie za pomocą komponentu `Display` kilku wartości. Problem polega tylko na tym, że wartości te są zdefiniowane wewnętrz metodę `ReactDOM.render()`. Opisane tu rozwiązanie jest najgorszym z możliwych, ponieważ każdy pośredni komponent znajdujący się na ścieżce od źródła do miejsca przeznaczenia musi odczytywać właściwości i definiować własne, które potem przekazuje dalej. Rozwiązanie jest po prostu fatalne. Można to zrobić lepiej, o czym dowiesz się za chwilę.

Poznaj operator rozciągania

Remedium na wszystkie nasze problemy jest nowa funkcjonalność języka JavaScript, zwana **operatorem rozciągania**. Działanie tego operatora trudno jest opisać bez konkretnego odniesienia, dlatego najpierw przedstawię przykład jego użycia, a potem zanudzę Cię jego definicją.

Przyjrzyjmy się poniższemu kodowi:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Wynik: " + a + " " + b + " " + c);
}
```

Mamy tu tablicę o nazwie `items` zawierającą trzy wartości. Jest również funkcja `printStuff()` z trzema argumentami. Chcemy w argumentach funkcji `printStuff()` umieścić wartości z tablicy `items`. Brzmi to banalnie, prawda?

Oto jeden z najwykłejnych sposobów, jak to zrobić:

```
printStuff(items[0], items[1], items[2]);
```

Trzeba każdy element tablicy umieścić osobno w argumencie funkcji `printStuff()`. Jednak za pomocą operatora rozciągania można to zrobić prościej. Nie trzeba odwoływać się do poszczególnych elementów tablicy, tylko napisać coś takiego:

```
printStuff(...items);
```

Operatorem rozciągania są trzy kropki (...) umieszczone przed nazwą tablicy `items`. Wyrażenie `...items` jest równoważne użytem wcześniejszym odwołaniom `items[0]`, `items[1]` i `items[2]`. Funkcja `printStuff()` po wywołaniu wyświetla w konsoli wartości 1, 2 i 3. Fajne, prawda?

Teraz, gdy zobaczyłeś operator rozciągania w akcji, pora na jego definicję. **Operator rozciągania służy do rozbijania tablicy na osobne elementy**, ale oprócz tego umożliwia robienie kilku innych rzeczy, które na razie nie są ważne. Do rozwiązania naszego problemu z przekazywaniem właściwości wykorzystamy tylko tę jedną cechę operatora.

Lepszy sposób przekazywania właściwości

Przed chwilą dowiedziałeś się, jak użyć operatora rozciągania, aby uniknąć umieszczania w argumentach funkcji pojedynczych elementów tablicy.:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Printing: " + a + " " + b + " " + c);
}

// Użyciem operatora rozciągania:
printStuff(...items);

// Bez użycia operatora rozciągania:
printStuff(items[0], items[1], items[2]);
```

Sytuacja, jaka powstaje podczas przekazywania właściwości pomiędzy komponentami, jest bardzo podobna do odwoływanego się do poszczególnych elementów tablicy. Opiszę to dokładniej.

Obiekt `props` w komponencie ma następującą postać:

```
var props = {
  color: "jasnoniebieski",
```

```
num: "3,14",
size: "średni"
};
```

Aby przekazać powyższe właściwości do komponentu podzielnego, trzeba odwołać się do każdej właściwości obiektu props osobno, jak niżej:

```
<Display color={this.props.color}
  num={this.props.num}
  size={this.props.size}/>
```

Czyż nie byłoby wspaniale, gdyby można było w jakiś sposób rozwinąć obiekt i przekazać wszystkie pary właściwość – wartość tak samo, jak rozwija się tablicę za pomocą operatora rozciągania?

Jak się okazuje, można tak zrobić. W tym celu również wykorzystuje się operator rozciągania. Wyjaśnię to później, a na razie powiem tylko, że komponent `Display` można wywoływać, stosując zapis `...this.props`, jak niżej:

```
<Display {...this.props} />
```

Efekt użycia zapisu `...this.props` jest taki sam jak efekt zastosowania osobnych odwołań do właściwości `color`, `num` i `size`. Oznacza to, że nasz poprzedni przykład można uprościć w następujący sposób (zwróć uwagę na wyróżnione wiersze):

```
class Display extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
}

class Label extends React.Component {
  render() {
    return (
      <Display {...this.props} />
    );
  }
}

class Shirt extends React.Component {
  render() {
    return (
      <div>
        <Label {...this.props} />
      </div>
    );
  }
}
```

Skutek wykonania powyższego kodu będzie taki sam jak efekt uzyskany wcześniej. Największa różnica pomiędzy obiema wersjami kodu polega na tym, że w wywołaniach komponentów nie są stosowane osobne właściwości. W ten sposób można rozwiązać wszystkie problemy, które napotkaliśmy na początku.

Dzięki operatorowi rozciągania w przypadku dodania, usunięcia lub zmiany nazw właściwości, a także w przypadku innych karkołomnych operacji, nie trzeba wprowadzać w kodzie miliona różnych modyfikacji. Wystarczy wprowadzić jedną zmianę w miejscu, w którym zdefiniowane są właściwości, i drugą w miejscu, gdzie są wykorzystywane. To wszystko. Wszystkie pośrednie komponenty, które jedynie przekazują właściwości, można pozostawić nietknięte, ponieważ wyrażenie `{...this.props}` nie zawiera informacji o tym, co znajduje się wewnątrz obiektu.

Czy to najlepszy sposób przekazywania właściwości?

Przekazywanie właściwości za pomocą operatora rozciągania jest wygodne i znacznie lepsze od odwoływania się do każdej właściwości osobno, tak jak to robiliśmy na początku. Jednak nawet użycie operatora rozciągania nie jest idealnym rozwiązaniem. Jeżeli jedynym celem jest przekazywanie właściwości do określonego komponentu, to angażowanie innych komponentów w charakterze pośredników stanowi niepotrzebne zamieszanie. Co więcej, proces ten może potencjalnie pogarszać wydajność aplikacji. Każda zmiana w przekazywanej właściwości skutkuje zmianami we wszystkich pośrednich komponentach. Tak nie może być! W dalszej części książki dowiesz się, jak można przekazywać właściwości w znacznie lepszy sposób, który nie wywołuje żadnych efektów ubocznych.

Podsumowanie

Operator rozciągania, opracowany przez komitet ES6/ES2015, powinien być z założenia stosowany tylko z tablicami i podobnymi im strukturami (tj. posiadającymi właściwość `Symbol.iterator`). Fakt, że można go wykorzystywać z takimi obiektemi jak `props`, bierze się stąd, że biblioteka React rozszerza powyższy standard. Żadna przeglądarka nie pozwala obecnie na stosowanie operatora rozciągania z obiektami. Mogliśmy go wykorzystać w naszym przykładzie dzięki bibliotece Babel, która nie tylko zamienia kod JSX na coś zrozumiałego dla przeglądarki, lecz także dostosowuje nowoczesne i eksperymentalne właściwości do możliwości różnych przeglądarek. Dlatego właśnie mogliśmy użyć operatora rozciągania z obiektem i elegancko rozwiązać problem przekazywania właściwości przez kilka warstw komponentów.

Jakie to wszystko ma znaczenie? Czy rzeczywiście wiedza o tym, jakie są niuanse funkcjonowania operatora rozciągania i dlaczego w jednych sytuacjach on działa, a w innych nie działa, jest krytycznie ważna? W większości przypadków odpowiedź brzmi: nie. Należy jednak pamiętać, że operator rozciągania można stosować do przekazywania właściwości z jednego komponentu do innego. W dalszej części książki poszukamy innych, równie prostych sposobów osiągnięcia tego celu, takich, które nie będą pogarszać wydajności aplikacji.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniemi, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

Witamy ponownie JSX!

Zapewne zauważyłeś, że bardzo często wykorzystywaliśmy w przykładach język JSX. Ale w rzeczywistości nie przyjrzałem się bliżej temu językowi. Dlaczego nie można używać po prostu kodu HTML? Jakie ukryte atuty ma JSX? W tym rozdziale znajdziesz odpowiedź na powyższe pytania i wiele innych! Zrobimy kilka dużych kroków wstecz (jak również wprzód) i sprawdzimy, co trzeba wiedzieć o JSX, aby go skutecznie stosować.

Co się dzieje z kodem JSX?

Jedną z najważniejszych rzeczy, którą pominęliśmy, było sprawdzenie, co się dzieje z kodem JSX, gdy już się go napisze. W jaki sposób jest on zamieniany na kod HTML widziany przez przeglądarkę? Przeanalizujmy poniższy przykład, w którym zdefiniowany jest komponent o nazwie Card:

```
class Card extends React.Component {  
  render() {  
    var cardStyle = {  
      height: 200,  
      width: 150,  
      padding: 0,  
      backgroundColor: "#FFF",  
      boxShadow: "0px 0px 5px #666"  
    };  
  
    return (  
      <div style={cardStyle}>  
        <Square color={this.props.color} />  
        <Label color={this.props.color} />  
      </div>  
    );  
  }  
}
```

Kod JSX jest tu wyraźnie widoczny. Są to cztery poniższe wiersze:

```
<div style={cardStyle}>  
  <Square color={this.props.color} />  
  <Label color={this.props.color} />  
</div>
```

Pamiętaj, że przeglądarka „nie wie”, co ma robić z kodem JSX. Nawet jeżeli postarasz się opisać, czym jest JSX, przeglądarka może uznać, że zwariowałeś. Dlatego musisz używać narzędzi takiego jak biblioteka Babel, tłumaczącego kod JSX na coś, co przeglądarka rozumie: na kod JavaScript.

Wynika z tego, że kod JSX jest przeznaczony tylko dla człowieka. Gdy kod JSX trafia do przeglądarki, jest zamieniany na czysty kod JavaScript:

```
return React.createElement(
  "div",
  { style: cardStyle },
  React.createElement(Square, { color: this.props.color }),
  React.createElement(Label, { color: this.props.color })
);
```

Wszystkie te zgrabnie zagnieździone elementy podobne do znaczników HTML, ich wartości i elementy potomne są zamieniane na serie metod createElement() z domyślnymi wartościami inicjującymi. Poniżej przedstawiony jest cały komponent Card po zamianie jego kodu na JavaScript:

```
class Card extends React.Component {
  render() {
    var cardStyle = {
      height: 200,
      width: 150,
      padding: 0,
      backgroundColor: "#FFF",
      boxShadow: "0px 0px 5px #666"
    };

    return React.createElement(
      "div",
      { style: cardStyle },
      React.createElement(Square, { color: this.props.color }),
      React.createElement(Label, { color: this.props.color })
    );
  }
}
```

Zwróć uwagę, że nie ma tu ani śladu po kodzie JSX! Wszystkie różnice pomiędzy tym, co napiszesz, a tym, co zobaczysz przeglądarka będą efektem transpilacji, o której pisalem w rozdziale 1. „Wstęp do biblioteki React”. Transpilacji całego kodu dokonuje w przeglądarce biblioteka Babel w sposób zupełnie niewidoczny dla programisty. Biblioteki Babel przyjrzymy się dokładniej przy okazji tworzenia bardziej zaawansowanego środowiska, w którym będzie generowany wynikowy plik JavaScript. Ale to na razie jest przyszłość.

Masz zatem odpowiedź na pytanie, co się właściwie dzieje z kodem JSX: jest zamieniany na milny kod JavaScript.

Atuty JSX, które trzeba znać

Zapewne zauważłeś, że wprowadziliśmy niepisane zasady i wyjątki dotyczące tego, co można, a czego nie można robić, pracując nad kodem JSX. W tym rozdziale przyjrzymy się bliżej tym zawiłościom. Poznasz też kilka nowych zasad!

Wyrażenia

Język JSX jest podobny do JavaScript. Jak się już wielokrotnie przekonałeś, można za jego pomocą przetwarzać nie tylko statyczną treść, jak niżej:

```
class Stuff extends React.Component {
  render() {
    return (
      <h1>Nudna, statyczna treść!</h1>
    );
  }
};
```

Metoda `render()` może zwracać dynamicznie generowane wartości. Wystarczy w tym celu użyć wyrażenia umieszczonego w nawiasach klamrowych:

```
class Stuff extends React.Component {
  render() {
    return (
      <h1>Nudna {Math.random() * 100} treść!</h1>
    );
  }
};
```

Zwróć uwagę, że została tu wykorzystana metoda `Math.random()`, zwracająca losową liczbę. Wyrażenie jest przetwarzane razem z umieszczonym wokół niego statycznym tekstem, więc na stronie pojawi się coś takiego: Nudna 28.6388820148227 treść!.

Najpierw jest określana wartość wyrażenia zawartego w nawiasach klamrowych, a następnie zwracany wynik operacji. Gdyby nie było nawiasów, metoda zwróciłaby tekst Nudna `Math.random() * 100 treść!`.

Na pewno nie jest to coś, czego byś oczekiwał.

Zwracanie wielu elementów

W wielu przykładach, które widziałeś, metoda `render()` zwracała główny element (najczęściej `div`) zawierający wiele innych elementów. Nie ma jednak technicznych przeszkód, aby wyjść poza ten schemat: metoda może zwracać wiele elementów, i to na dwa sposoby.

Pierwszy sposób polega na zastosowaniu struktury podobnej do tablicy, jak niżej:

```
class Stuff extends React.Component {
  render() {
    return [
      <p>Zwracana</p>,
      <p>lista</p>,
      <p>różnych rzeczy!</p>
    ];
  }
};
```

W tym przypadku metoda zwraca trzy elementy `p`, które nie mają jednego, wspólnego elementu nadzawanego. Gdy metoda zwraca wiele elementów, można, choć nie jest to obowiązek, stosować (w zależności od użytcej wersji biblioteki React) pewne udoskonalenie. Każdemu elementowi można przypisać unikatowy atrybut `key`:

```
class Stuff extends React.Component {
  render() {
    return (
      [
        <p key="1">Zwracana</p>,
        <p key="2">lista</p>,
        <p key="3">różnych rzeczy!</p>
      ]
    );
  }
}
```

Dzięki temu można łatwiej wskazywać, w którym elemencie ma być wprowadzona zmiana za pomocą biblioteki React. Skąd można się dowiedzieć, że trzeba użyć atrybutu key? Podpowie to sama biblioteka. W konsoli narzędzi programistycznych pojawi się komunikat, na przykład `Warning: Each child in an array or iterator should have a unique "key" prop (Uwaga: każdy element podrzedny w tablicy lub iteratorze musi posiadać unikatową właściwość „key”).`

Jest jeszcze inny (może nawet lepszy) sposób zwracania wielu elementów. Polega on na zastosowaniu tzw. **fragmentów**. Wygląda to następująco:

```
class Stuff extends React.Component {
  render() {
    return (
      <React.Fragment>
        <p>Zwracana</p>
        <p>lista</p>
        <p>różnych rzeczy!</p>
      </React.Fragment>
    );
  }
}
```

Listę elementów, które ma zwrócić metoda, umieszcza się w magicznym komponencie `React.Fragment`. Zwróć uwagę na kilka fajnych rzeczy:

1. Powyższy komponent nie tworzy elementu w modelu DOM. Jest on czymś, co się definiuje w kodzie JSX, ale co nie przekłada się na rzeczywisty element HTML widziany przez przeglądarkę.
2. Zwracane wartości nie są elementami tablicy, więc nie trzeba ich rozdzielać za pomocą przecinków ani w żaden inny sposób.
3. Nie trzeba definiować unikatowych atrybutów key i ich wartości. To się dzieje w sposób dla Ciebie niewidoczny.

Zanim przejdziemy do następnej części rozdziału, dodam jeszcze, że zamiast pełnej specyfikacji komponentu `React.Fragment` można stosować bardziej zwięzły zapis. Można po prostu użyć pustych znaczników `<>` i `</>`:

```
class Stuff extends React.Component {
  render() {
    return (
      <>
        <p>Zwracana</p>
        <p>lista</p>
        <p>różnych rzeczy!</p>
      </>
    );
  }
}
```

```
        </>
    );
}
```

Kod wygląda dość futurystycznie, ale jeżeli jesteś przekonany do stosowania fragmentów zwracających kilka wartości, stosuj powyższą uproszczoną składnię.

Nie można definiować stylów CSS w kodzie

Jak się dowiedziałeś w rozdziale 4. „Style w bibliotece React”, atrybut `style` funkcjonuje w kodzie JSX inaczej niż w HTML. W kodzie HTML wartością tego atrybutu jest styl CSS, jak niżej:

```
<div style="font-family:Arial;font-size:24px">
  <p>Cześć!</p>
</div>
```

W kodzie JSX atrybut `style` nie może zawierać stylu CSS. Zamiast tego musi odwoływać się do obiektu z informacjami o stylu:

```
class Letter extends React.Component {
  render() {
    var letterStyle = {
      padding: 10,
      margin: 10,
      backgroundColor: this.props.bgcolor,
      color: "#333",
      display: "inline-block",
      fontFamily: "monospace",
      fontSize: "32",
      textAlign: "center"
    };
    return (
      <div style={letterStyle}>
        {this.props.children}
      </div>
    );
  }
}
```

Zwróć uwagę, że obiekt `letterStyle` zawiera wszystkie właściwości stylu CSS (których nazwy mają format przyjęty w języku JavaScript, tj. zawierają wielkie litery w środku). Nazwa tego obiektu jest umieszczona w definicji atrybutu `style`.

Komentarze

Dobrą praktyką jest umieszczanie komentarzy w kodach HTML, CSS i JavaScript, więc podobnie jest w przypadku kodu JSX. Komentarz specyfikuje się w podobny sposób jak w JavaScript, z jednym wyjątkiem: aby zagnieździć komentarz w elemencie należy go zamknąć w nawiasach klamrowych, aby był traktowany jak wyrażenie:

```
ReactDOM.render(
  <div className="slideIn">
    <p className="emphasis">Hej!</p>
    {/* Zagnieżdżony komentarz */}
  </div>
)
```

```

<Label/>
</div>,
document.querySelector("#container")
);

```

W tym przypadku komentarz jest zagnieźdzony w elemencie `div`. Jeżeli komentarz trzeba umieścić wewnątrz znacznika, można go zdefiniować tak jak zwykły komentarz jedno- lub wielowierszowy, bez stosowania nawiasów klamrowych:

```

ReactDOM.render(
  <div className="slideIn">
    <p className="emphasis">Uwaga!</p>
    <Label
      /* Komentarz
       zajmujący
       kilka wierszy. */
      className="colorCard" // Koniec wiersza.
    />
  </div>,
  document.querySelector("#container")
);

```

Powyższy fragment kodu zawiera zarówno komentarz wielowierszowy, jak i jednowierszowy. Teraz już wiesz, jak poprawnie dodawać komentarze, i nie masz powodów, aby unikać ich umieszczania w kodzie JSX.

Wielkości liter, elementy HTML i komponenty

Wielkość liter ma znaczenie. Znaczniki HTML należy wpisywać, używając małych liter, jak niżej:

```

ReactDOM.render(
  <div>
    <section>
      <p>Tu znajduje się treść!</p>
    </section>
  </div>,
  document.querySelector("#container")
);

```

Natomiast w nazwie komponentu pierwsza litera musi być wielka:

```

ReactDOM.render(
  <div>
    <MyCustomComponent/>
  </div>,
  document.querySelector("#container")
);

```

Jeżeli użyjesz liter o nieodpowiedniej wielkości, biblioteka React nie zinterpretuje poprawnie nazw. Sprawdzenie, czy użyłeś właściwej wielkości liter, byłoby prawdopodobnie ostatnią rzeczą, która przyszła Ci do głowy podczas diagnozowania problemów. Dlatego pamiętaj o tej prostej zasadzie.

Kod JSX można stosować wszędzie

W wielu przypadkach Twój kod JSX nie będzie tak elegancko umieszczony wewnątrz metody `render()` lub `return()`, jak w opisanych przykładach. Przyjrzyj się poniższemu fragmentowi:

```
var swatchComponent = <Swatch color="#2F004F"></Swatch>

ReactDOM.render(
  <div>
    {swatchComponent}
  </div>,
  document.querySelector("#container")
);
```

Zmienna `swatchComponent` jest tu zdefiniowana i zainicjowana za pomocą jednego wiersza kodu JSX. Umieszczona wewnątrz metody `render()` inicjuje komponent `Swatch`. Cały kod jest najzupełniej poprawny. Tego rodzaju składnię będziesz stosował w przyszłości częściej, gdy nauczysz się generować i modyfikować kod JSX za pomocą kodu JavaScript.

Podsumowanie

W tym rozdziale zebraliśmy w jednym miejscu różnego rodzaju informacje o języku JSX wykorzystane w poprzednich rozdziałach. Przede wszystkim musisz pamiętać, że kod JSX nie jest kodem HTML. Przypomina wprawdzie HTML i działa podobnie w wielu sytuacjach, ale został przygotowany z myślą o tłumaczeniu go na kod JavaScript. Oznacza to, że dzięki niemu można robić rzeczy, o których nigdy byś nie pomyślał, że da się je uzyskać za pomocą zwykłego kodu HTML. Możliwość stosowania wyrażeń czy programowego manipulowania całymi fragmentami kodu to tylko część możliwości języka JSX. W następnych rozdziałach dokładniej opiszę wspólne cechy języków JavaScript i JSX.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

8

Obsługa stanów w React

Komponenty, które tworzyłeś do tej pory, były bezstanowe. Posiadały wprawdzie właściwości (przechowywane w obiekcie props) przekazywane przez elementy nadzędne, ale nie były w nich wprowadzane żadne zmiany. Właściwości te po nadaniu im wartości były **niemutowalne** (niezmienne). W typowych aplikacjach interaktywnych nie jest to cenna cecha. Potrzebna jest możliwość zmieniania właściwości komponentów, gdy użytkownik wykona jakąś operację (albo gdy zostaną odebrane dane z serwera, albo gdy wystąpi milion innych zdarzeń).

Potrzebny jest inny sposób przechowywania danych w komponencie, bez użycia właściwości. Dane muszą być przechowywane tak, aby można je było zmieniać. Potrzebny jest tzw. **stan**. W tym rozdziale dowiesz się wszystkiego o stanach i sposobach ich wykorzystywania w tworzeniu komponentów stanowych.

Stosowanie stanów

Skoro wiesz już, jak stosować właściwości, wiesz również doskonale, jak posługiwać się stanami... no prawie. Istnieje kilka różnic pomiędzy tymi dwoma pojęciami, ale są one zbyt subtelne, aby teraz Ciebie nimi zanudzać. Zamiast tego przejdźmy od razu do rzeczy i na prostym przykładzie zobaczymy stany w akcji.

Utworzmy prosty licznik uderzeń piorunów, przedstawiony na rysunku 8.1.

Aplikacja nie będzie robić niczego nadzwyczajnego. Według „National Geographic” na całym świecie uderza co sekundę około 100 piorunów. Nasz licznik będzie po prostu automatycznie wskazywał odpowiednią wartość. Zatem do dzieła.

Punkt wyjścia

Głównym celem tego przykładu jest pokazanie, jak używa się stanów. Nie ma potrzeby tworzenia aplikacji od podstaw i ponownego przechodzenia utartą ścieżką. Nie byłby to najlepiej wykorzystany wspólny czas.



Rysunek 8.1. Aplikacja, którą utworzymy

Zamiast zaczynać od zera, zmień istniejący dokument HTML lub utwórz nowy z następującą zawartością:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Obsługa stanu</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
</head>

<body>
  <div id="container"></div>

  <script type="text/babel">
    class LightningCounter extends React.Component {
      render() {
        return (
          <h1>Witaj!</h1>
        );
      }
    }

    class LightningCounterDisplay extends React.Component {
      render() {
        var divStyle = {
          width: 250,
          textAlign: "center",
          backgroundColor: "black",
          padding: 40,
          fontFamily: "sans-serif",
          color: "#999",
          borderRadius: 10
        };
      }
    }
  </script>
</body>
```

```

        return (
            <div style={divStyle}>
                <LightningCounter/>
            </div>
        );
    }
}

ReactDOM.render(
    <LightningCounterDisplay/>,
    document.querySelector("#container")
);
</script>
</body>
</html>

```

Poświęć chwilę na zapoznanie się z tym, co robi powyższy kod. Na początku definiuje komponent `LightningCounterDisplay` (wyświetlacz licznika piorunów). W większości jest to definicja obiektu `divStyle` zawierającego informacje o stylu przyjemnego dla oka, zaokrąglonego prostokąta. Funkcja `return()` zwraca element `div` zawierający komponent `LightningCounter`. W tym komponencie są wykonywane wszystkie operacje:

```

class LightningCounter extends React.Component {
    render() {
        return (
            <h1>Witaj!</h1>
        );
    }
}

```

Na razie komponent nie robi niczego interesującego. Po prostu wyświetla słowo `Witaj!`. To pożądany efekt, za chwilę się nim zajmiemy.

Ostatnią rzeczą, na którą należy zwrócić uwagę, jest metoda `ReactDOM.render()`:

```

ReactDOM.render(
    <LightningCounterDisplay/>,
    document.querySelector("#container")
);

```

Metoda ta jedynie wysyła komponent `LightningCounterDisplay` do modelu DOM. To prawie wszystko. Końcowy efekt jest kombinacją znaczników generowanych przez metodę `ReactDOM.render()` i przez komponenty `LightningCounterDisplay` i `LightningCounter`.

Włączenie licznika

Teraz, gdy wiesz już, od czego zaczynamy, czas zaplanować następny krok. Nasz licznik będzie bardzo prosty. Wykorzystamy funkcję `setInterval()`, która będzie wywoływała pewien kod co 1000 milisekund (czyli co 1 sekundę). Ów „pewien kod” będzie za każdym razem dodawał do wartości licznika liczbę 100. Brzmi to bardzo prosto, prawda?

Aby to wszystko osiągnąć, wykorzystamy dwie metody oferowane przez bibliotekę React:

1. `componentDidMount()`

Jest to metoda wywoływana zaraz po wyświetleniu komponentu (czyli „zamontowaniu go”, zgodnie z terminologią biblioteki React).

2. `setState()`

Metoda umożliwiająca zmienianie wartości obiektu `state`.

Za chwilę dowiesz się, jak stosować powyższe metody. Zostały tutaj ogólnie opisane, abyś mógł je łatwo rozpoznać w kodzie.

Określanie początkowej wartości stanu

Będzie nam potrzebna zmienna pełniąca funkcję licznika. Nazwijmy ją `strikes` (uderzenia). Można ją zdefiniować na kilka sposobów. Jeden z najbardziej oczywistych wygląda następująco:

```
var strikes = 0; // :P
```

Tak jednak nie zrobimy. W naszym przykładzie zmienna `strikes` będzie częścią stanu komponentu. Trzeba utworzyć obiekt `state` zawierający właściwość `strikes` i odpowiednio go przygotować w momencie utworzenia komponentu. Operacje te wykona komponent `LightningCounter`. Wpisz teraz poniższe wyróżnione wiersze:

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  render() {
    return (
      <h1>Witaj!</h1>
    );
  }
}
```

Obiekt `state` jest konfigurowany w konstruktorze komponentu `LightningCounter`. Ten kod jest wykonywany na długo przed wyświetleniem komponentu. Stanowi polecenie dla biblioteki React, aby skonfigurowała obiekt zawierający właściwość `strikes` (tj. przypisała jej wartość 0).

Jeżeli sprawdzisz wartość obiektu `state` po uruchomieniu powyższego kodu, zobaczyś coś takiego:

```
var state = {
  strikes: 0
};
```

Zanim zakończymy tę część rozdziału, musimy zakodować wizualizację właściwości `strikes`. Wprowadź w metodzie `render()` wyróżnioną niżej zmianę:

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  render() {
```

```

        return (
          <h1>{this.state.strikes}</h1>
        );
      }
    }
  }
}

```

Zmieniłeś domyślny ciąg `Witaj!` wyrażeniem zwracającym wartość zapisaną we właściwości `this.state.strikes`. Jeżeli otworzysz teraz kod w przeglądarce, pojawi się strona z liczbą 0. Dobry początek!

Uruchomienie czasomierza i ustawienie stanu

Teraz trzeba uruchomić czasomierz, który będzie zwiększał wartość właściwości `strikes`. Jak wspomniałem wcześniej, wykorzystamy w tym celu funkcję `setInterval()`, która co jedną sekundę będzie dodawała do wartości właściwości `strikes` liczbę 100. Zakodujemy tę operację zaraz pod częścią tworzącą komponent, a użyjemy w tym celu wbudowanej metody `componentDidMount()`.

Kod uruchamiający czasomierz wygląda jak niżej:

```

class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  componentDidMount() {
    setInterval(this.timerTick, 1000);
  }

  render() {
    return (
      <h1>{this.state.strikes}</h1>
    );
  }
}

```

Wpisz teraz w kodzie wyróżnione wyżej wiersze. Wewnątrz metody `componentDidMount()`, wywoływanej po wyświetleniu komponentu, znajduje się metoda `setInterval()` wywołująca co jedną sekundę (1000 milisekund) funkcję `timerTick()`.

Nie zdefiniowaliśmy jeszcze funkcji `timerTick()`. Zrób to teraz, dodając do kodu wyróżniony niżej fragment:

```

class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };
  }

  timerTick() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  }
}

```

```

    }

componentDidMount() {
  setInterval(this.timerTick, 1000);
}

render() {
  return (
    <h1>{this.state.strikes}</h1>
  );
}
}

```

Funkcja `timerTick()` wykonuje bardzo prostą operację. Wywołuje tylko metodę `setState()`. Metoda ta ma wiele odmian. Tutaj wykorzystaliśmy tę, w której argumentem jest obiekt. Właściwości tego obiektu są **dodawane do właściwości obiektu state**. W naszym przypadku jest to właściwość `strikes`, której przypisywana jest nowa wartość, większa o 100 od bieżącej wartości tej właściwości.

Inkrementacja bieżącej wartości stanu

W przeszłości często będziesz modyfikował istniejącą wartość stanu, tak jak w tym przykładzie. Tutaj bieżąca wartość jest odczytywana za pomocą wyrażenia `this.state.strikes`. Ze względów wydajnościowych biblioteka React może aktualizować stan seriami operacji wykonywanymi w krótkich odstępach czasu. W efekcie oryginalna wartość obiektu `this.state` może odbiegać od rzeczywistości. Aby temu zapobiec, można w metodzie `setState()` odwoływać się do poprzedniej wartości stanu za pomocą argumentu `prevState`. Kod wykorzystujący ten argument wygląda następująco:

```

this.setState((prevState) => {
  return {
    strikes: prevState.strikes + 100
  };
});

```

Efekt jest podobny do uzyskanego poprzednio. Właściwość `strikes` jest powiększana o wartość 100. Jedyna różnica polega na tym, że gwarantowana jest poprawna wartość właściwości `strikes` niezależnie od tego, jaka była jej wcześniejsza wartość zapisana w obiekcie `state`.

Czy należy zatem aktualizować stan w ten sposób? Są ważne argumenty za i przeciw. Z jednej strony istotna jest dokładność, choć obiekt `this.state` i tak funkcjonuje poprawnie w większości przypadków. Z drugiej strony kod musi być prosty i nie należy go niepotrzebnie komplikować. Nie ma jednak poprawnej odpowiedzi, więc stosuj sposób, który sam uznasz za lepszy. Piszę o tym tylko dlatego, abyś miał pełny obraz sytuacji. Podejście wykorzystujące właściwość `prevState` będziesz mógł stosować w każdym opartym na bibliotece React kodzie, z którym będziesz miał do czynienia w praktyce.

Musisz zrobić jeszcze jedną rzecz. Dodałeś do komponentu funkcję `timerTick()`, jednak jej *zawartość* nie odnosi się do kontekstu komponentu. Innymi słowy, obecnie użycie słowa kluczowego `this` do wywołania metody `setState` spowoduje zgłoszenie błędu `TypeError`. Istnieje kilka rozwiązań tego problemu, każde na swój sposób frustrujące. Tym zagadnieniem dokładnie zajmiemy się później. Teraz jawnie powiążemy funkcję `timerTick()` z komponentem, dzięki czemu wszystkie odwołania będą funkcjonowały poprawnie. Wpisz w kodzie konstruktora wyróżniony niżej wiersz:

```

constructor(props) {
  super(props);

  this.state = {
    strikes: 0
  };

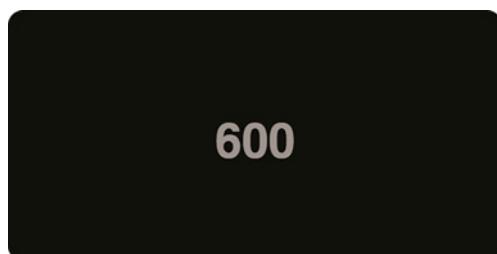
  this.timerTick = this.timerTick.bind(this);
}

```

Kiedy wprowadzimy tę zmianę, funkcja `timerTick()` będzie stanowiła użyteczną część naszego komponentu.

Wizualizacja zmiany stanu

Gdy teraz otworzysz kod w przeglądarce, wartość właściwości `strikes` zacznie się powiększać o 100 co sekundę (patrz rysunek 8.2.).



Rysunek 8.2. Wartość właściwości `strikes` powiększa się o 100 co sekundę

Zostawmy na moment to, co się dzieje w kodzie, ponieważ są to bardzo proste operacje. Interesujące jest, że wprowadzane zmiany są odzwierciedlane na ekranie. Wynika to ze sposobu funkcjonowania biblioteki React: **po każdym wywołaniu metody `setState()` i zmianie czegokolwiek w obiekcie state automatycznie jest wywoływana metoda `render()`**. Powoduje ona kaskadowe wywołanie metod `render()` wszystkich obiektów, których wygląd również się zmienia. Efekt tych wywołań jest taki, że na ekranie jest widoczna reprezentacja bieżącego stanu interfejsu aplikacji. Zapewnienie synchronizacji pomiędzy danymi aplikacji a zawartością interfejsu użytkownika jest największym problemem w tworzeniu tego interfejsu. Dobrze więc, że biblioteka React troszczy się o to wszystko za nas. Dlatego warto (chyba) było nauczyć się korzystać z biblioteki React.

Opcja: pełny kod

W tej chwili mamy jedynie licznik, który co sekundę wyświetla wartość o 100 większą niż poprzednia. Nie ma on nic wspólnego ze „zliczaniem błyskawic”, ale zawiera wszystko, co chciałem Ci przekazać na temat stanów. Gdybyś chciał wyczytać stronę tak, aby uzyskać efekt, który widziałeś na początku rozdziału, poniżej przedstawiam pełny kod, który powinien znaleźć się wewnątrz znacznika `script`:

```
class LightningCounter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      strikes: 0
    };

    this.timerTick = this.timerTick.bind(this);
  }

  timerTick() {
    this.setState({
      strikes: this.state.strikes + 100
    });
  }

  componentDidMount() {
    setInterval(this.timerTick, 1000);
  }

  render() {
    var counterStyle = {
      color: "#66FFFF",
      fontSize: 50
    };

    var count = this.state.strikes.toLocaleString();

    return (
      <h1 style={counterStyle}>{count}</h1>
    );
  }
}

class LightningCounterDisplay extends React.Component {
  render() {
    var commonStyle = {
      margin: 0,
      padding: 0
    };

    var divStyle = {
      width: 250,
      textAlign: "center",
      backgroundColor: "#020202",
      padding: 40,
      fontFamily: "sans-serif",
      color: "#999999",
      borderRadius: 10
    };

    var textStyles = {
      emphasis: {
        fontSize: 38,
        ...commonStyle
      },
      smallEmphasis: {
        ...commonStyle
      },
    };
  }
}
```

```
        small: {
          fontSize: 17,
          opacity: 0.5,
          ...commonStyle
        }
      };

      return (
        <div style={divStyle}>
          <LightningCounter />
          <h2 style={textStyles.smallEmphasis}>Liczba uderzeń piorunów</h2>
          <h2 style={textStyles.emphasis}>na całym świecie</h2>
          <p style={textStyles.small}>(od czasu otwarcia strony)</p>
        </div>
      );
    }
  }

ReactDOM.render(
  <LightningCounterDisplay />,
  document.querySelector("#container")
);
```

Gdy doprowadzisz kod do takiej postaci jak wyżej i ponownie otworzysz stronę, zobaczysz licznik uderzeń piorunów w całej swojej świetności. Po tym, jak to zrobisz, poświeć chwilę na przejrzenie kodu i upewnienie się, że nie zawiera żadnych niespodzianek.

Podsumowanie

Miałeś tutaj przedsmak tego, co można osiągnąć, tworząc stanowe komponenty. Czasomierz zmieniający coś w stanie obiektu state jest fajny, ale prawdziwa akcja rozpocznie się wtedy, gdy zaczniesz łączyć interakcje użytkownika ze stanem aplikacji. Na razie darowaliśmy sobie masę szczegółów związanych z obsługą myszy, padów, klawiatury i innych urządzeń, które musi obsługiwać komponent. W następnym rozdziale to się zmieni. Przy okazji wszystko to, czego dowiedziałeś się na temat stanów, przeniesiesz na zupełnie inny poziom. Jeżeli to Ciebie nie intryguje, to nie wiem, co mogłoby Cię zaintrygować.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

Od danych do interfejsu użytkownika

P odczas tworzenia aplikacji operowanie takimi terminami jak właściwości, komponenty, znaczniki JSX i metody render() oraz innymi opisującymi React słowami będzie prawdopodobnie ostatnią rzeczą, jaka przyjdzie Ci do głowy. Większość czasu będziesz poświęcała danym w postaci obiektów JSON, tablic i innych struktur, które z biblioteką React i elementami interfejsu mają niewiele wspólnego. Przepaść pomiędzy danymi a tym, co się ma pojawić na stronie, jest ogromna! Nie martw się jednak. Ten rozdział zmniejszy Twoją frustrację, ponieważ opisuje sytuacje, z którymi będziesz się często spotykał.

Przykład

Aby było Ci łatwiej zrozumieć wszystko, czego się za chwilę dowiesz, potrzebny będzie przykład. To nie będzie nic skomplikowanego, więc utwórz nowy dokument HTML i umieść w nim poniższy kod:

```
<!DOCTYPE html>
<html>

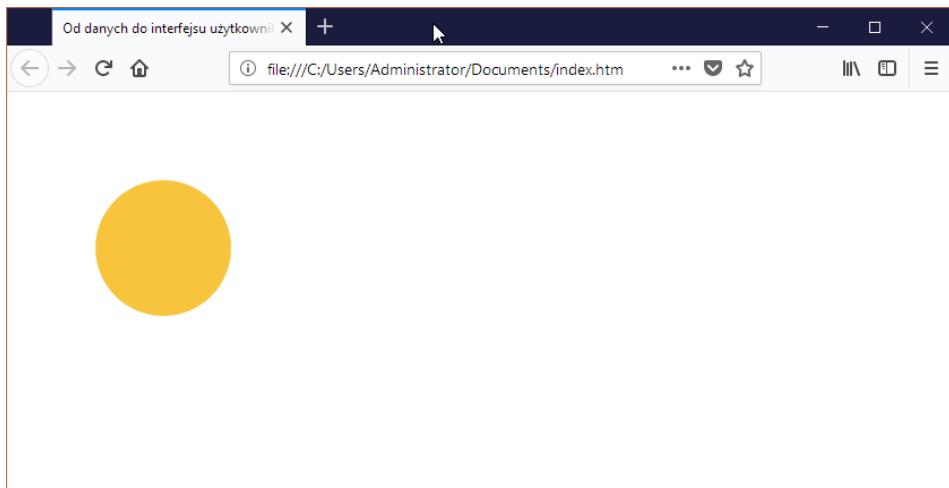
<head>
  <meta charset="utf-8">
  <title>Od danych do interfejsu użytkownika</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>
<body>
  <div id="container"></div>

  <script type="text/babel">
```

```
class Circle extends React.Component {  
    render() {  
        var circleStyle = {  
            padding: 10,  
            margin: 20,  
            display: "inline-block",  
            backgroundColor: this.props.bgColor,  
            borderRadius: "50%",  
            width: 100,  
            height: 100,  
        };  
  
        return (  
            <div style={circleStyle}>  
                </div>  
        );  
    }  
  
    ReactDOM.render(  
        <div>  
            <Circle bgColor="#F9C240" />  
        </div>,  
        document.querySelector("#container")  
    );  
    </script>  
</body>  
  
</html>
```

Po utworzeniu dokumentu otwórz go w przeglądarce. Jeżeli się nie pomyliłeś, zostaniesz uraczyony wesołym żółtym kołem, jak na rysunku 9.1.



Rysunek 9.1. Jeżeli się nie pomyliłeś, otrzymasz żółte koło

Jeżeli widzisz to samo, co ja — świetnie! Teraz poświęć chwilę na zapoznanie się z tym, co się w kodzie dzieje. Zawartość strony jest głównie zdefiniowana w komponencie `Circle`:

```
class Circle extends React.Component {  
  render() {  
    var circleStyle = {  
      padding: 10,  
      margin: 20,  
      display: "inline-block",  
      backgroundColor: this.props.bgColor,  
      borderRadius: "50%",  
      width: 100,  
      height: 100,  
    };  
  
    return (  
      <div style={circleStyle}>  
      </div>  
    );  
  }  
}
```

Komponent składa się niemal wyłącznie z obiektu `circleStyle` zawierającego właściwości, które zamieniają nudny element `div` na ciekawe koło. Wartości niemal wszystkich właściwości są zakodowane na stałe. Wyjątkiem jest właściwość `backgroundColor`, której przypisywana jest wartość właściwości `bgColor` obiektu `props`.

Koło zdefiniowane przy użyciu komponentu jest ostatecznie wyświetlane za pomocą metody `ReactDOM.render()`:

```
ReactDOM.render(  
  <div>  
    <Circle bgColor="#F9C240" />  
  </div>,  
  document.querySelector("#container")  
>;
```

Mamy więc zadeklarowaną jedną instancję komponentu `Circle`, w którym kolor koła jest określany za pomocą właściwości `bgColor`. Sposób, w jaki komponent został zdefiniowany, tj. wewnątrz metody `render()`, ma pewne ograniczenia ujawniające się szczególnie podczas przetwarzania danych, które mają wpływ na wygląd komponentu. W następnych częściach rozdziału dowiesz się, jak można rozwiązać ten problem.

Kod JSX można stosować wszędzie (część II)

W rozdziale 7. „Witamy ponownie JSX!” dowiedziałeś się, że kod JSX można umieszczać poza metodą `render()`, jak również można go przypisywać zmiennym lub właściwościom. Śmiało więc można na przykład zrobić coś takiego:

```
var theCircle = <Circle bgColor="#F9C240" />;  
  
ReactDOM.render(  
  <div>  
    {theCircle}  
  </div>,  
  document.querySelector("#container")  
>;
```

Zmienna `theCircle` zawiera kod JSX wyświetlający komponent `Circle`. Użycie tej zmiennej wewnętrz metodę `ReactDOM.render()` powoduje, że na stronie pojawia się koło. Efektniczym się nie różni od uzyskanego wcześniej, ale dzięki oddzieleniu od metody `render()` kodu tworzącego instancję komponentu można robić dużo fajnych rzeczy. Można na przykład utworzyć funkcję zwracającą komponent `Circle`:

```
function showCircle() {
  var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363"];
  var ran = Math.floor(Math.random() * colors.length);

  // Zwrócenie komponentu Circle o losowo wybranym kolorze.
  return <Circle bgColor={colors[ran]} />;
}
```

Funkcja `showCircle()` zwraca komponent `Circle` (nic nadzwyczajnego!), w którym właściwość `bgColor` zawiera losowo wybrany kod koloru (niesamowite!). Aby wykorzystać tę funkcję w naszym przykładzie, wystarczy umieścić ją wewnętrz metodę `ReactDOM.render()`:

```
ReactDOM.render(
  <div>
    {showCircle()}
  </div>,
  document.querySelector("#container")
);
```

W nawiasach {} można umieścić dowolne wyrażenie, pod warunkiem że jego wartością będzie kod JSX. Ta elastyczność jest naprawdę cenna, ponieważ pozwala wpisywać poza metodą `render()` dowolny kod JavaScript.

Tablice

Teraz zrobimy coś fajnego! Aby wyświetlić kilka komponentów, można je zakodować ręcznie, jak niżej:

```
ReactDOM.render(
  <div>
    {showCircle()}
    {showCircle()}
    {showCircle()}
  </div>,
  document.querySelector("#container")
);
```

W praktyce liczba wyświetlanych komponentów często zależy od liczby elementów tablicy lub podobnej struktury (na przykład iteratatora). Pojawiają się wtedy pewne komplikacje. Założymy, że mamy następującą tablicę o nazwie `colors`:

```
var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363",
  "#85FFC7", "#297373", "#FF8552", "#A40E4C"];
```

Chcemy wyświetlić tyle komponentów `Circle`, ile jest elementów w tablicy (a ich właściwościom `bgColor` przypisać wartości poszczególnych elementów). W tym celu utwórzmy tablicę komponentów `Circle`:

```

var colors = ["#393E41", "#E94F37", "#1C89BF", "#A1D363",
    "#85FFC7", "#297373", "#FF8552", "#A40E4C"];

var renderData = [];

for (var i = 0; i < colors.length; i++) {
    renderData.push(<Circle bgColor={colors[i]} />);
}

```

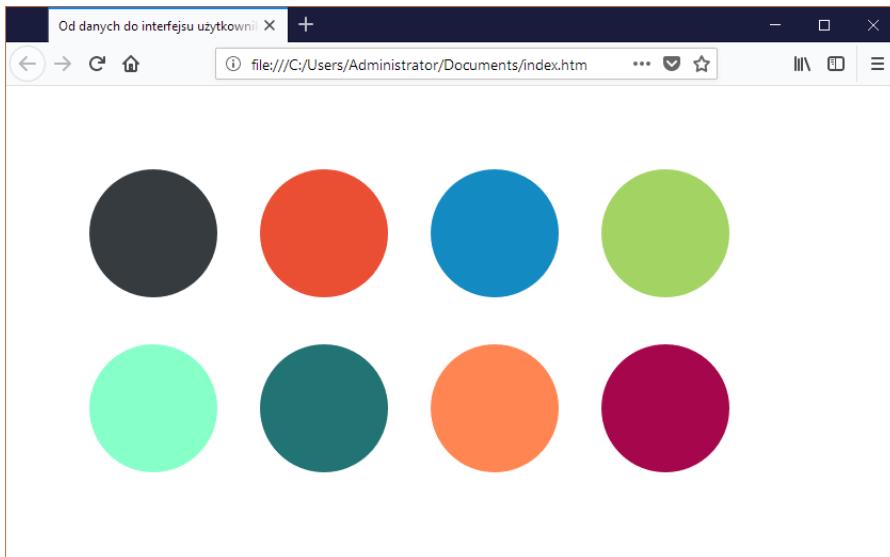
Powyższy kod, zgodnie z naszymi oczekiwaniami, umieszcza w tablicy renderData komponenty Circle. Na razie wygląda to nieźle. Wyświetlenie tych komponentów za pomocą biblioteki React jest bardzo proste. Wystarczy wpisać wyróżniony niżej wiersz:

```

ReactDOM.render(
    <div>
        {renderData}
    </div>,
    document.querySelector("#container")
);

```

W metodzie render() użyte zostało jedynie wyrażenie odwołujące się do tablicy renderData. Nie trzeba nic więcej robić, aby tablicę komponentów zamienić na coś takiego jak na rysunku 9.2.



Rysunek 9.2. Widok w przeglądarce

Skłamałem jednak. Trzeba zrobić jeszcze jedną drobną rzecz. Biblioteka React potrafi bardzo szybko aktualizować wygląd interfejsu dzięki temu, że dokładnie „wie”, co się dzieje w modelu DOM. Wykorzystuje w tym celu różne sposoby, z których jeden z ważniejszych polega na przypisaniu wszystkim elementom identyfikatorów.

Gdy elementy są tworzone dynamicznie (tak jak nasza tablica komponentów Circle), wtedy nie są im automatycznie nadawane identyfikatory. Dlatego trzeba zakodować dodatkową operację przypisującą właściwości key unikatową wartość, którą biblioteka React będzie

wykorzystywała do identyfikowania komponentu. W naszym przykładzie można to zrobić w następujący sposób:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor={color} />);
}
```

Każdy komponent zawiera deklarację właściwości key, której przypisywana jest kombinacja kodu koloru i jego indeksu w tablicy colors. W ten sposób każdy dynamicznie tworzony komponent uzyskuje unikatowy identyfikator, który biblioteka React będzie później wykorzystywała do optymalizacji procesu odświeżania interfejsu użytkownika.

Sprawdzaj zawartość konsoli!

Biblioteka React będzie Cię naprawdę dokładnie informować, jeżeli będziesz coś robić źle. Jeśli na przykład będziesz dynamicznie tworzył elementy lub komponenty bez określania ich właściwości key, wtedy w konsoli przeglądarki zobaczysz następujący komunikat:

```
Warning: Each child in an array or iterator should have a unique "key" prop.
Check the top-level render call using <div>.
// (Uwaga: każdy element tablicy lub iteratora musi mieć unikatową właściwość "key".
Sprawdź główną metodę render() generującą znacznik div.)
```

Gdy korzysta się z biblioteki React, dobrą praktyką jest regularne sprawdzanie, czy w konsoli przeglądarki pojawiają się komunikaty. Nigdy nie wiadomo, co się tam może znaleźć, nawet jeżeli będziesz przekonany, że kod działa poprawnie.

Podsumowanie

Opisane w tym rozdziale sztuczki i kruczki można stosować tylko dlatego, że **JSX to w rzeczywistości JavaScript**. Dzięki temu kod JSX można umieszczać wszędzie tam, gdzie można używać JavaScript. Coś takiego jak niżej może wyglądać zupełnie nieprawdopodobnie:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];
  renderData.push(<Circle key={i + color} bgColor={color} />);
}
```

W tablicy zapisywane są fragmenty kodu JSX, co wygląda na magię. Jednak kiedy umieścimy tablicę renderData wewnętrz metody render(), cały kod działa poprawnie. Nie lubię się w kółko powtarzać jak zdarta płyta, ale to, co ostatecznie „widzi” przeglądarka, wygląda mniej więcej tak:

```
for (var i = 0; i < colors.length; i++) {
  var color = colors[i];

  renderData.push(React.createElement(Circle,
  {
    key: i + color,
    bgColor: color
  )));
}
```

Gdy kod JSX zostanie przełożony na JavaScript, wszystko nabiera sensu. Dzięki temu mechanizmowi można uniknąć umieszczania kodu JSX w różnych niewygodnych miejscach, a mimo wszystko uzyskiwać zamierzony efekt.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

10

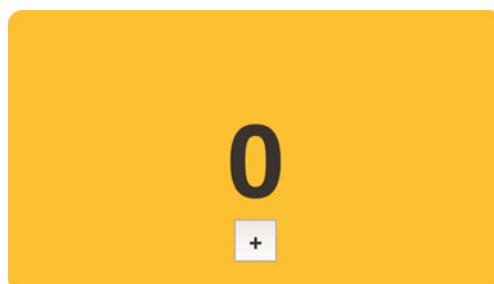
Zdarzenia w React

W większości opisanych do tej pory przykładów wszystkie operacje były wykonywane zaraz po załadowaniu strony. Jak zapewne się domyślasz, normalna aplikacja nie działa w ten sposób. Większość aplikacji, które będziesz tworzył, szczególnie tych z rozbudowanym interfejsem użytkownika, będzie wykonywać mnóstwo operacji w reakcji na „coś”. Tym „czymś” może być na przykład kliknięcie myszą, naciśnięcie klawisza, zmiana wielkości okna, różnego rodzaju gesty i interakcje. Wiązanie reakcji z tymi operacjami odbywa się za pomocą **zdarzeń**.

Prawdopodobnie wykorzystywałeś już zdarzenia zachodzące w modelu DOM i wiesz o nich wszystko. (Jeżeli jednak nie wiesz, proponuję, abyś szybko odświeżył swoją wiedzę, odwiedzając stronę https://www.kirupa.com/html5/javascript_events.htm). Biblioteka React obsługuje zdarzenia w nieco inny sposób, który może być dla Ciebie zaskoczeniem, jeżeli nie poświęcisz mu należytej uwagi. Nie martw się jednak, masz przecież tę książkę! Zaczniemy od kilku prostych przykładów, a potem stopniowo będziemy zajmować się coraz dziwniejszymi, bardziej skomplikowanymi i (niestety!) coraz nudniejszymi przypadkami.

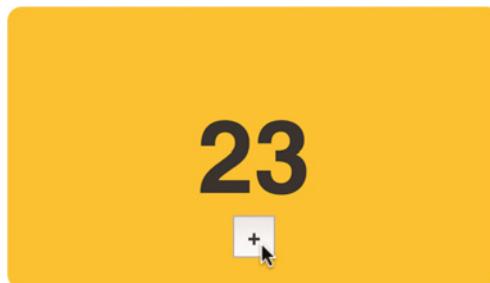
Nasłuchiwanie i obsługa zdarzeń

Najprostszym sposobem na zapoznanie się z obsługą zdarzeń w bibliotece React jest korzystanie z nich. Właśnie tak teraz będziemy robić. Przeanalizujmy przykład prostego licznika, który będzie rejestrował każde kliknięcie przycisku. Na początku aplikacja będzie wyglądała tak jak na rysunku 10.1.



Rysunek 10.1. Nasz przykład

Za każdym razem, gdy klikniesz przycisk ze znakiem „+”, licznik wyświetli wartość większą o 1. Gdy wielokrotnie klikniesz przycisk, otrzymasz widok jak na rysunku 10.2.



Rysunek 10.2. Widok strony po wielokrotnym (23 razy) kliknięciu przycisku

W rzeczywistości przykład ten jest bardzo prosty. Każdorazowo po kliknięciu przycisku zgłaszane jest zdarzenie. Biblioteka React nasłuchuje, czy wystąpiło to zdarzenie i gdy ono się pojawi, wykonuje pewne operacje modyfikujące wartość licznika.

Punkt wyjścia

Aby zaoszczędzić nieco czasu, nie będziemy tworzyć kodu od podstaw. Na pewno doskonale już wiesz, jak korzystać z komponentów, stylów, stanów itp. Zaczniemy więc od częściowo zaimplementowanej aplikacji, zawierającej wszystko oprócz obsługi zdarzeń, którą teraz poznasz.

Najpierw utwórz nowy dokument HTML i upewnij się, że zawiera kod taki jak ten:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Zdarzenia</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>
<body>
  <div id="container"></div>
  <script type="text/babel">

  </script>
</body>

</html>
```

Jeżeli Twój dokument HTML wygląda jak wyżej, czas zaimplementować licznik. Poniżej znacznika `div`, wewnątrz znacznika `script`, wpisz następujący kod:

```
class Counter extends React.Component {
  render() {
    var textStyle = {
      fontSize: 72,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold"
    };

    return (
      <div style={textStyle}>
        {this.props.display}
      </div>
    );
  }
}

class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };
  }

  render() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };

    var buttonStyle = {
      fontSize: "1em",
      width: 30,
      height: 30,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold",
      lineHeight: "3px"
    };

    return (
      <div style={backgroundStyle}>
        <Counter display={this.state.count} />
        <button style={buttonStyle}>+</button>
      </div>
    );
  }
}

ReactDOM.render(
  <div>
    <CounterParent />
  </div>,
  document.querySelector("#container")
);
```

Otwórz stronę w przeglądarce, aby upewnić się, że kod działa poprawnie. Powinieneś zobaczyć licznik w stanie początkowym. Poświęć chwilę na przyjrzenie się temu, co się dzieje w kodzie. Nic w nim nie powinno być dla Ciebie nowością. Możesz się jedynie dziwić, dlaczego po kliknięciu przycisku nic się nie dzieje. Tym zajmiemy się w następnej części rozdziału.

Przygotowanie przycisku do reagowania na kliknięcie

Chcemy, aby za każdym razem, kiedy klikniemy przycisk, wartość wyświetlna przez licznik zwiększała się o 1. To, co musimy zrobić, wygląda z grubsza następująco:

1. Zakodować nasłuchiwanie zdarzenia kliknięcia przycisku.
2. Zaimplementować obsługę zdarzenia polegającą na zwiększaniu wartości właściwości `this.state.count` wykorzystywanej przez nasz licznik.

Będziemy postępować według powyższej listy. Zaczniemy od nasłuchiwania zdarzenia. Jeżeli korzysta się z biblioteki React, należy nasłuchiwanie zakodować samodzielnie, używając języka JSX. Konkretnie: **nałeży określić zarówno nazwę nasłuchiwanego zdarzenia, jak i zaimplementować funkcję, która będzie wywoływana**. Aby to zrobić, odszukaj w komponencie CounterParent funkcję `return` i wprowadź w niej wyróżnioną niżej zmianę:

```
return (
  <div style={backgroundColor}>
    <Counter display={this.state.count}/>
    <button onClick={this.increase} style={buttonStyle}>+</button>
  </div>
);
```

Wiersz ten zawiera informację dla biblioteki React, aby w chwili, gdy usłyszy zdarzenie `onClick`, wywoływała funkcję `increase()`. Zaimplementuj teraz tę funkcję (czyli **procedurę obsługi zdarzenia**). Wewnątrz komponentu CounterParent wpisz wyróżnione niżej wiersze:

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };

    this.increase = this.increase.bind(this);
  }

  increase(e) {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    var backgroundColor = {
      padding: 50,
      backgroundColor: "#FFC53A",
    }
  }
}
```

```

width: 250,
height: 100,
borderRadius: 10,
textAlign: "center"
};

var buttonStyle = {
  fontSize: "1em",
  width: 30,
  height: 30,
  fontFamily: "sans-serif",
  color: "#333",
  fontWeight: "bold",
  lineHeight: "3px"
};

return (
  <div style={backgroundStyle}>
    <Counter display={this.state.count} />
    <button onClick={this.increase} style={buttonStyle}>+</button>
  </div>
);
}
}

```

Wyróżniony kod powoduje jedynie, że po każdym wywołaniu funkcji `increase()` powiększana jest o 1 wartość właściwości `this.state.count`. Ponieważ mamy do czynienia ze zdarzeniami, nasza funkcja `increase()` (jako wyznaczona do tego celu procedura obsługi) musi mieć dostęp do wszystkich właściwości zdarzenia. Funkcja odwołuje się do nich za pomocą argumentu `e` określonego w sygnaturze (czyli deklaracji). O różnych zdarzeniach i ich właściwościach dowiesz się więcej, gdy bliżej przyjrzymy się klasie `Events`.

Dodatkowo w powyższym kodzie w konstruktorze komponentu funkcja `increase()` jest wiązana z obiektem `this`.

Otwórz teraz stronę w przeglądarce. Po załadowaniu kliknij przycisk i zobacz nowy kod w działaniu. Po każdym kliknięciu powinna się zwiększać wartość licznika. Niesamowite, prawda?

Właściwości zdarzenia

Jak już wiesz, zdarzenie ma właściwości, które są przekazywane do procedury obsługi. Właściwości są różne, zależne od typu zdarzenia. W modelu DOM każde zdarzenie ma określony własny typ. Na przykład zdarzenie zgłoszone po kliknięciu myszą jest typu `MouseEvent`. Za pomocą obiektu tego typu uzyskuje się dostęp do informacji opisujących kliknięcie, między innymi o użytym przycisku lub pozycji kurSORA na ekranie. Zdarzenie zgłoszone w wyniku naciśnięcia klawisz jest typu `KeyboardEvent`. Obiekt tego typu zawiera właściwości opisujące na przykład naciśnięty klawisz. Mógłbym kontynuować wymienianie kolejnych typów zdarzeń, ale myślę, że wiesz już, o co chodzi. Każdy typ zdarzenia zawiera własny zestaw właściwości, do których można odwoływać się w procedurze obsługującej dane zdarzenie.

Dlaczego jednak zanudzam Cię rzecząmi, które już wiesz? Otóż teraz...

Poznaj zdarzenia syntetyczne

Biblioteka React nie obsługuje bezpośrednio wskazanych w kodzie JSX zdarzeń modelu DOM, na przykład `onClick`. Wykorzystuje w tym celu specjalną klasę `SyntheticEvent`. Procedura obsługi zdarzenia nie ma dostępu do natywnych obiektów typu `MouseEvent`, `KeyboardEvent` itp. Zamiast nich wykorzystuje zdarzenie `SyntheticEvent`, które opakowuje natywne zdarzenie pojawiające się w przeglądarce. Jakie to ma znaczenie dla Twojego kodu? O dziwo, niewielkie.

Klasa `SyntheticEvent` posiada następujące właściwości:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

Nazwy powyższych właściwości powinny brzmieć znajomo i bardzo ogólnie. Specyficzne właściwości zależą od typu natywnego zdarzenia opakowanego w zdarzenie `SyntheticEvent`. Oznacza to, że klasa `SyntheticEvent` opakowująca na przykład klasę `MouseEvent` ma następujące dodatkowe właściwości związane z obsługą myszy:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Analogicznie klasa `SyntheticEvent` opakowująca klasę `KeyboardEvent` ma następujące dodatkowe właściwości związane z obsługą klawiatury:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

W efekcie funkcjonalności zdarzenia SyntheticEvent są takie same jak zwykłych zdarzeń zachodzących w modelu DOM.

A teraz chcę Ci przekazać coś, czego nauczyłem się z własnego doświadczenia: **podczas korzystania ze zdarzenia SyntheticEvent i jego właściwości nie korzystaj z dokumentacji do tradycyjnych zdarzeń modelu DOM**. Klasa SyntheticEvent opakowuje klasę natywnego zdarzenia modelu DOM i właściwości obu klas mogą nie być wiernie ze sobą powiązane. Niektóre zdarzenia z modelu DOM nie są nawet zaimplementowane w bibliotece React. Jeżeli będziesz chciał dowiedzieć się, jaka jest nazwa określonego zdarzenia lub właściwości, wtedy aby uniknąć problemów, skorzystaj z dokumentacji *React Event System* (system zdarzeń biblioteki React, <https://facebook.github.io/react/docs/events.html>).

Korzystanie z właściwości zdarzeń

Prawdopodobnie dowidziałeś się właśnie o zdarzeniach modelu DOM i zdarzeniu SyntheticEvent o wiele więcej, niż byś chciał. Napiszmy nieco kodu, w którym wykorzystamy świeżo nabycą wiedzę.

Obecnie wartość licznika po każdym kliknięciu przycisku zwiększa się o 1. Założymy, że chcielibyśmy, aby wartość ta zwiększała się o 10, kiedy naciśniemy klawisz *Shift* i klikniemy przycisk. Efekt ten można osiągnąć, wykorzystując właściwość *shiftKey* zawartą w klasie SyntheticEvent opakowującej zdarzenie kliknięcia myszą:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEVENTTARGET relatedTarget
number screenX
number screenY
boolean shiftKey
```

Znaczenie tej właściwości jest oczywiste. Jeżeli w chwili, gdy kliknęliśmy myszą, naciśnięty był klawisz *Shift*, powyższa właściwość otrzymała wartość *true*. W przeciwnym wypadku ma wartość *false*. Aby przy naciśniętym klawiszem *Shift* wartość licznika zwiększała się o 10, musisz w funkcji *increase()* wprowadzić wyróżnione niżej zmiany:

```
increase(e) {
  var currentCount = this.state.count;

  if (e.shiftKey) {
    currentCount += 10;
  } else {
    currentCount += 1;
  }

  this.setState({
    count: currentCount
  });
}
```

Po wprowadzeniu zmian otwórz stronę w przeglądarce. Jak poprzednio, gdy klikniesz przycisk, wartość licznika zwiększy się o 1. Jeżeli natomiast przed kliknięciem naciśniesz klawisz *Shift*, wartość licznika zwiększy się o 10. Dzieje się tak, ponieważ uzależniłeś operację zwiększania wartości licznika od stanu klawisza *Shift*. Najważniejsze są tu następujące wiersze:

```
if (e.shiftKey) {
  currentCount += 10;
} else {
  currentCount += 1;
}
```

Jeżeli właściwość *shiftKey* zdarzenia *SyntheticEvent* ma wartość *true*, wartość licznika jest zwiększana o 10. Jeżeli właściwość ma wartość *false*, wartość licznika jest zwiększana o 1.

Więcej o zawiłościach zdarzeń

Jeszcze nie koniec! Do tej pory analizowaliśmy obsługę zdarzeń w bibliotece React w bardzo uproszony sposób. Rzeczywistość jednak rzadko będzie tak prosta, jak opisany przykład. Twoje aplikacje będą bardziej skomplikowane, a ponieważ biblioteka React często działa w nietypowy sposób, będziesz musiał nauczyć się (lub nauczyć na nowo) paru sztuczek i technik związanych z obsługą zdarzeń. Właśnie o tym jest ta część rozdziału. Przyjrzymy się teraz kilku problemom, które będziesz często spotykał i musiał sobie z nimi radzić.

Zdarzeń nie można nasłuchiwać bezpośrednio w komponentach

Załóżmy, że komponent jest prostym przyciskiem lub innym wykorzystywanym przez użytkownika elementem. Nie można w kodzie wpisać wyróżnionego niżej wiersza:

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };

    this.increase = this.increase.bind(this);
  }

  increase(e) {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div>
        <Counter display={this.state.count} />
        <PlusButton onClick={this.increase} />
      </div>
    );
  }
}
```

Wyróżniony kod JSX na pozór wygląda zupełnie poprawnie. Jeżeli użytkownik kliknie komponent PlusButton, zostanie wywołana funkcja increase(). Jeżeli Cię to interesuje, komponent PlusButton wygląda następująco:

```
class PlusButton extends React.Component {  
  render() {  
    return (  
      <button>  
        +  
      </button>  
    );  
  }  
}
```

Ten kod nie robi niczego szczególnego. Po prostu zwraca jeden element HTML.

W aplikacji jednak nic nie będzie się działało, choćbyś nie wiadomo jak się starał. I nie ma znaczenia, jak bardzo prosty i oczywisty będzie kod HTML zwracany przez ten komponent.

Po prostu nie można bezpośrednio nasłuchiwać zdarzeń w komponencie. Jest tak, ponieważ komponent to opakowanie elementu DOM. Co właściwie oznacza sformułowanie „nasłuchiwanie zdarzenia w komponencie”? Czy kiedy komponent zostanie rozłożony na elementy modelu DOM, zewnętrzny element HTML będzie mógł nasłuchiwać zdarzeń? Czy też będzie to inny element? Jak można rozdzielić nasłuchiwanie zdarzenia od deklaracji właściwości?

Nie ma jasnych odpowiedzi na powyższe pytania. Nie można też jednoznacznie stwierdzić, że jedynym rozwiązaniem jest po prostu rezygnacja z nasłuchiwanego zdarzeń w komponencie. Na szczęście jest na to sposób. Procedurę obsługi zdarzenia można potraktować jak właściwość i przesyłać ją do komponentu. Wewnątrz komponentu należy przypisać zdarzenie do elementu DOM, a jako procedurę obsługi zdarzenia podstawić przekazaną komponentowi właściwość. Wiem, że może to być dla Ciebie niezrozumiałe, dlatego rozważmy przykład.

Przyjrzyj się wyróżnionemu niżej wierszowi:

```
class CounterParent extends React.Component {  
  .  
  .  
  .  
  render() {  
    return (  
      <div>  
        <Counter display={this.state.count} />  
        <PlusButton clickHandler={this.increase} />  
      </div>  
    );  
  }  
}
```

Jest tu zdefiniowana właściwość clickHandler, której wartością jest funkcja increase() obsługująca zdarzenie. W definicji komponentu PlusButton można więc wpisać następujący kod:

```
class PlusButton extends React.Component {  
  render() {  
    return (  
      <button onClick={this.props.clickHandler}>  
        +  
      </button>  
    );  
  }  
}
```

```

        </button>
    );
}
}

```

Element button ma atrybut onClick, któremu jest przypisywana wartość właściwości clickHandler. Wartością całego wyrażenia jest funkcja increase(), która jest wywoływana po kliknięciu przycisku. W ten sposób można rozwiązać nasz problem, a jednocześnie korzystać w komponencie ze wszystkie dobrodziejstw zdarzeń.

Nasłuchiwanie zwykłych zdarzeń modelu DOM

Jeżeli poprzednia część rozdziału była dla Ciebie czymś wyjątkowym, przygotuj się na to, co teraz zobaczysz. Nie wszystkie zdarzenia modelu DOM mają swoje odpowiedniki typu SyntheticEvent. Mogłoby się wydawać, że wystarczy w kodzie JSX w nazwie zdarzenia umieścić wielkie litery i dodać prefiks on, jak niżej:

```

class Something extends React.Component {
    .
    .
    .
    handleMyEvent(e) {
        // Kod zdarzenia.
    }

    render() {
        return (
            <div onSomeEvent={this.handleMyEvent}>Cześć!</div>
        );
    }
}

```

To jednak nie będzie działać! W przypadku zdarzeń, których biblioteka React nie rozpoznaje, trzeba stosować tradycyjną funkcję addEventListener() i kilka dodatkowych instrukcji.

Przyjrzyjmy się wyróżnionym fragmentom kodu:

```

class Something extends React.Component {
    .
    .
    .
    handleMyEvent(e) {
        // Kod zdarzenia.
    }

    componentDidMount() {
        window.addEventListener("someEvent", this.handleMyEvent);
    }

    componentWillUnmount() {
        window.removeEventListener("someEvent", this.handleMyEvent);
    }

    render() {
        return (
            <div>Cześć!</div>
        );
    }
}

```

Mamy tu komponent Something nasłuchujący zdarzenia o nazwie someEvent. Nasłuchiwanie uruchamia metoda componentDidMount(), automatycznie wywoływana w chwili wyświetlenia komponentu. Nasłuchiwanie jest zaimplementowane za pomocą funkcji addEventListener(), w której argumentach umieszczona jest nazwa zdarzenia i procedura jego obsługi.

Jest to bardzo prosty sposób. Jedyną rzeczą, o której należy pamiętać, jest usuwanie procedury obsługi zdarzenia w momencie usunięcia komponentu. W tym celu trzeba użyć przeciwnieństwa metody componentDidMount(), czyli metody componentWillUnmount(). Wewnątrz tej metody trzeba wywoływać funkcję removeEventListener(), dzięki której po usunięciu komponentu nasłuchiwanie zdarzenia zostanie wyłączone.

Obiekt this w procedurze obsługi zdarzenia

W przypadku stosowania biblioteki React obiekt this użyty wewnętrz procedury obsługi zdarzenia nie reprezentuje zgłoszającego zdarzenie elementu modelu DOM, tak jak w poniższym kodzie:

```
function doSomething(e) {
  console.log(this); //Element button.
}

var foo = document.querySelector("button");
foo.addEventListener("click", doSomething, false);
```

W bibliotece React obiekt this nie reprezentuje elementu, który zgłosił zdarzenie. Jest to nieprzydatny (jednak poprawny) obiekt undefined. Dlatego trzeba za pomocą metody bind() jawnie wiązać obiekt this z elementem, tak jak to już robiliśmy:

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };

    this.increase = this.increase.bind(this);

  }

  increase(e) {
    console.log(this);

    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div>
        <Counter display={this.state.count} />
        <button onClick={this.increase}>+</button>
      </div>
    );
  }
}
```

W tym przykładzie obiekt `this` użyty wewnątrz funkcji `increase()` reprezentuje komponent `CounterParent`, a nie element, który zgłosił zdarzenie. Aby obiekt reprezentował ten element, należy go z nim powiązać w konstruktorze komponentu.

React, ale dlaczego?

Zanim zamkniniemy ten rozdział, poświęćmy chwilę na zastanowienie się, dlaczego biblioteka React inaczej obsługuje zdarzenia. Są ku temu dwa powody:

1. Zapewnienie kompatybilności ze starszymi przeglądarkami.
2. Zwiększenie wydajności kodu.

Przyjrzyjmy się bliżej tym kwestiom.

Kompatybilność ze starszymi przeglądarkami

Obsługa zdarzeń jest jednym z mechanizmów, które w nowoczesnych przeglądarkach działają tak samo. Jednak w przypadku starszych wersji przeglądarek sytuacja nie wygląda już tak dobrze. Dzięki opakowaniu wszystkich natywnych zdarzeń w obiekty typu `SyntheticEvent` biblioteka React zwalnia programistę z zajmowania się zawiłościami obsługi zdarzeń.

Większa wydajność

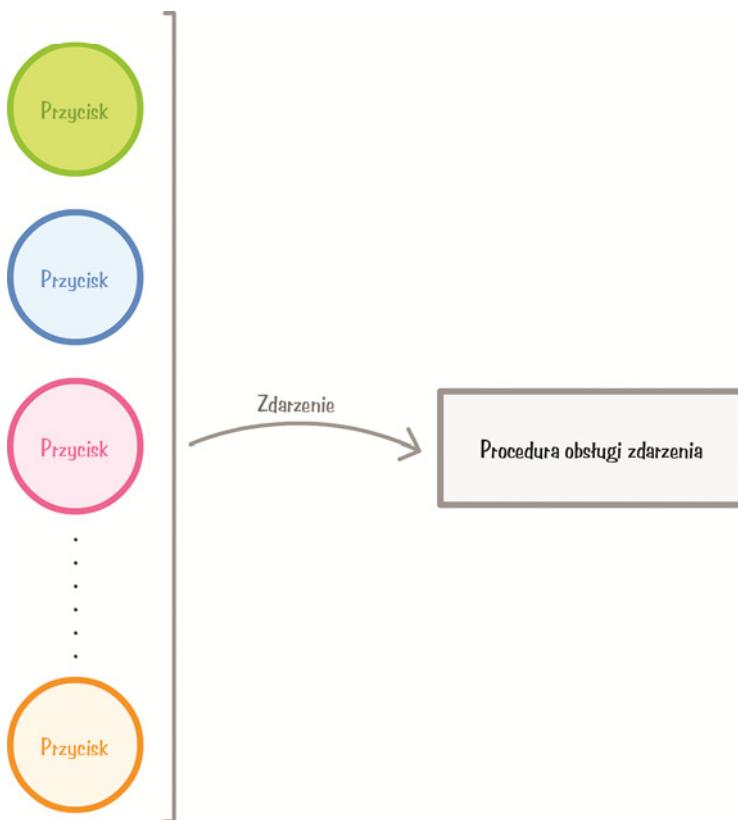
Im bardziej skomplikowany jest interfejs użytkownika, im więcej zawiera procedur obsługi zdarzeń, tym więcej zajmuje pamięci. Zakodowanie takiego interfejsu nie jest trudne, ale umieszczenie zdarzeń w jednej wspólnej grupie jest dość uciążliwe. Czasami jest to nawet niemożliwe, a czasami nakład związanego z tym pracy jest większy niż uzyskane korzyści. To, co robi w takiej sytuacji biblioteka React, jest naprawdę fajne.

Biblioteka nie przypisuje procedur obsługi bezpośrednio do zdarzeń. *Umieszcza na początku dokumentu* jedną procedurę obsługującą wszystkie zdarzenia i wywołującą odpowiednie funkcje (patrz rysunek 10.3).

Dzięki takiemu rozwiążaniu programista nie musi samodzielnie optymalizować kodu obsługującego zdarzenia. Jeżeli robiłeś tak wcześniej, możesz odetchnąć z ulgą — teraz biblioteka React wykona za Ciebie całą żmudną robotę. Jeżeli nie optymalizowałeś procedur obsługi zdarzeń, to znaczy, że jesteś szczęściarzem.

Podsumowanie

Poświęciliśmy sporo czasu obsłudze zdarzeń, a wiele rzeczy w tym rozdziale na pewno było dla Ciebie nowością. Zaczęliśmy od podstawowych zagadnień związanych z nasłuchiwaniem zdarzeń i definiowaniem procedur ich obsługi. W końcowej części rozdziału dokładnie przeanalizowaliśmy związane ze zdarzeniami pułapki, w które możesz wpaść, jeżeli nie będziesz ostrożny. Na pewno nie chciałbyś tego, bo to wcale nie jest przyjemne.



Rysunek 10.3. Biblioteka React umieszcza na początku dokumentu jedną procedurę obsługi zdarzeń

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

11

Cykl życia komponentu

Na początku książki bardzo ogólnie opisałem, czym są komponenty i co robią. W miarę pogłębiania swojej wiedzy o bibliotece React i tworzenia coraz ciekawszych i bardziej zaawansowanych rzeczy przekonałeś się, że korzystanie z komponentów nie jest takie proste. Ułatwiają przetwarzanie właściwości, stanów i zdarzeń, a często są również odpowiedzialne za poprawne działanie innych komponentów. Śledzenie pracy wszystkich komponentów może być jednak nie lada wyzwaniem.

Aby ułatwić to zadanie, biblioteka React oferuje **metody cyklu życia** komponentu. Jak się można spodziewać, są to specjalne metody, które komponent wywołuje automatycznie w trakcie swego działania. Metody te wysyłają powiadomienia o ważnych etapach w cyklu życia komponentu i można je wykorzystywać do kontrolowania lub zmieniania jego działania.

W tym rozdziale poznasz metody cyklu życia i dowiesz się, co można za ich pomocą osiągnąć.

Zmiany są tuż-tuż!

Zostało zgłoszonych kilka propozycji wprowadzenia zmian w metodach cyklu życia komponentu. To, co tu przeczytasz, opiera się na najnowszych danych. Pamiętaj jednak, że informacje te mogą się zmienić. Aby być na bieżąco, odwiedzaj stronę <http://bit.ly/lifecycleChanges>.

Poznaj metody cyklu życia

Metody cyklu życia nie są skomplikowane. Możesz je traktować jako szczególnego rodzaju procedury obsługi zdarzeń wywoływanego w różnych momentach życia komponentu. Umieszcza się w nich kod, który jest wykonywany w tych momentach. Zanim przejdziemy dalej, poznaj listę metod:

- `componentWillMount()`,
- `componentDidMount()`,
- `componentWillUnmount()`,
- `componentWillUpdate()`,
- `componentDidUpdate()`,
- `shouldComponentUpdate()`,
- `componentWillReceiveProps()`.

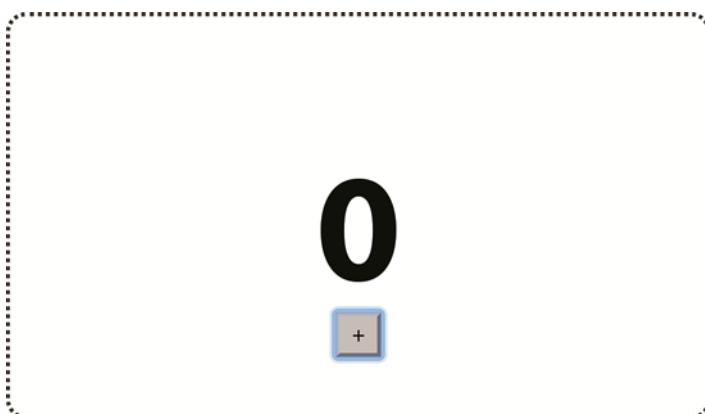
To jednak nie wszystko. Dodajmy do listy jeszcze jedną metodę, która właściwie nie jest metodą cyklu życia: sławetną metodę `render()`.

Niektóre z tych nazw prawdopodobnie wydają Ci się znajome, a inne zapewne widzisz pierwszy raz. Nie przejmuj się. Po przeczytaniu tego rozdziału będziesz je wszystkie znał od podszewki! Przyjrzymy się tym metodom z różnych stron, wykorzystując na początku prosty kod.

Metody cyklu życia w akcji

Poznawanie metod cyklu życia jest równie fascynujące, co uczenie się na pamięć nazw różnych nieznanych miejsc, których nie planuje się nigdy odwiedzić. By dało się to jakoś znieść, zaczniemy od przedstawienia prostego przykładu, a potem przejdziemy do akademickich dyskusji.

Aby zapoznać się opisany tu przykładem, otwórz w przeglądarce załączony do książki plik `r11\index.htm`. Zobaczysz inną wersję licznika, który wcześniej zaimplementowałeś (patrz rysunek 11.1).



Rysunek 11.1. Odmiana licznika

Na razie niczego nie klikaj. (Jeżeli jednak już kliknęłaś przycisk na stronie, odśwież ją po prostu, aby licznik się wyzerował). Jest ku temu powód i nie jest nim bynajmniej moja fanaberia. Zależy mi, abyś zobaczył, jak wygląda strona, zanim zaczniesz jej używać.

Otwórz teraz narzędzia deweloperskie przeglądarki i kliknij zakładkę *Konsola*. Jeżeli używasz przeglądarki Mozilla, pojawi się widok podobny do przedstawionego na rysunku 11.2.

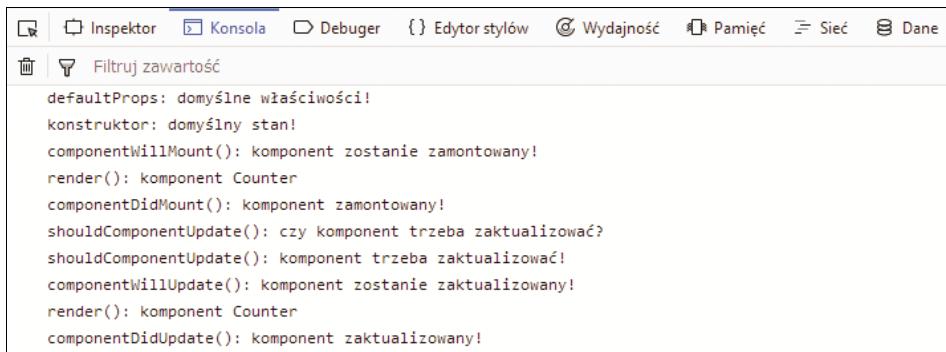
Rysunek 11.2 przedstawia widok konsoli WWW w przeglądarce Mozilla. Konsola ma białą tło i czarny font. Główka konsoli zawiera menu z ikonami: Inspektor, Konsola (która jest aktywna), Debugger, Edytor stylów, Wydajność, Pamięć, Sieć, Dane. Pod głową znajdują się dwa przyciski: "Filtruj zawartość" (z ikoną szukania) i "Wyszukaj" (z ikoną szkła). Następnie jest przedstawiony listowy log komunikacji z komputerem:

```

defaultProps: domyślne właściwości!
konstraktor: domyślny stan!
componentWillMount(): komponent zostanie zamontowany!
render(): komponent Counter
componentDidMount(): komponent zamontowany!
  
```

Rysunek 11.2. Widok konsoli WWW w przeglądarce Mozilla

Zwróć uwagę na zawartość konsoli. Jest w niej widocznych kilka komunikatów rozpoczynających się od nazw metod cyklu życia komponentu. Gdy klikniesz przycisk na stronie, w konsoli pojawi się więcej metod (patrz rysunek 11.3).



The screenshot shows the browser's developer tools with the 'Konsola' (Console) tab selected. At the top, there are tabs for 'Inspektor' (Inspector), 'Konsola' (Console), 'Debugger', 'Edytor stylów' (Style Editor), 'Wydajność' (Performance), 'Pamięć' (Memory), 'Sieć' (Network), and 'Dane' (Data). Below the tabs, there is a search bar with the placeholder 'Filtruj zawartość' (Filter content) and a dropdown menu icon. The main area displays several log statements in red text:

```
defaultProps: domyślne właściwości!
konstruktor: domyślny stan!
componentWillMount(): komponent zostanie zamontowany!
render(): komponent Counter
componentDidMount(): komponent zamontowany!
shouldComponentUpdate(): czy komponent trzeba zaktualizować?
shouldComponentUpdate(): komponent trzeba zaktualizować!
componentWillUpdate(): komponent zostanie zaktualizowany!
render(): komponent Counter
componentDidUpdate(): komponent zaktualizowany!
```

Rysunek 11.3. Wywołane kolejne metody cyklu życia komponentu

Kliknij jeszcze kilka razy przycisk. Jak się przekonasz, wszystkie te metody można umieścić w kontekście komponentu, który już znasz. Za każdym razem, kiedy klikniesz przycisk, będą pojawiać się kolejne metody. Wreszcie licznik osiągnie wartość 5, po czym zniknie ze strony, a w konsoli pojawi się następujący komunikat:

```
componentWillUnmount(): komponent zostanie usunięty z modelu DOM!
```

Na tym działanie kodu się kończy. Oczywiście, aby zacząć od początku, musisz po prostu odświeżyć stronę.

Teraz, po zapoznaniu się z przykładem, przyjrzyj się komponentowi odpowiedzialnemu za wszystko, co widziałeś. Poniżej przedstawiony jest jego kod:

```
class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    console.log("konstruktor: domyślny stan!");

    this.state = {
      count: 0
    };

    this.increase = this.increase.bind(this);
  }

  increase() {
    this.setState({
      count: this.state.count + 1
    });
  }

  componentWillMount(nextProps, nextState) {
    console.log("componentWillUpdate(): komponent zostanie zaktualizowany!");
  }
}
```

```

componentDidUpdate(currentProps, currentState) {
  console.log("componentDidUpdate(): komponent zaktualizowany!");
}

componentWillMount() {
  console.log("componentWillMount(): komponent zostanie zamontowany!");
}

componentDidMount() {
  console.log("componentDidMount(): komponent zamontowany!");
}

componentWillUnmount() {
  console.log("componentWillUnmount(): komponent zostanie usunięty z modelu DOM!");
}

shouldComponentUpdate(newProps, newState) {
  console.log("shouldComponentUpdate(): czy komponent trzeba zaktualizować?");
  if (newState.count < 5) {
    console.log("shouldComponentUpdate(): komponent trzeba zaktualizować!");
    return true;
  } else {
    ReactDOM.unmountComponentAtNode(destination);
    console.log("shouldComponentUpdate(): komponentu nie trzeba aktualizować!");
    return false;
  }
}

componentWillReceiveProps(nextProps) {
  console.log("componentWillReceiveProps(): komponent uzyska nowe właściwości!");
}

render() {
  var backgroundStyle = {
    padding: 50,
    border: "#333 2px dotted",
    width: 250,
    height: 100,
    borderRadius: 10,
    textAlign: "center"
  };

  return (
    <div style={backgroundStyle}>
      <Counter display={this.state.count} />
      <button onClick={this.increase}>
        +
      </button>
    </div>
  );
}

console.log("defaultProps: domyślne właściwości!");
CounterParent.defaultProps = {
};

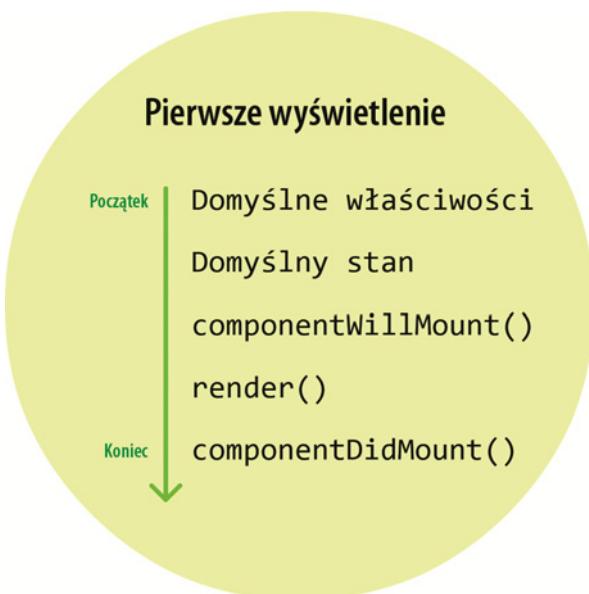
```

Poświeć chwilę na przyjrzenie się temu, co robi ten kod. Jest dość długi, ale składa się głównie z wymienionych wcześniej metod cyklu życia, z których każda wywołuje metodę `console.log()`.

Po zapoznaniu się z kodem „przeklikaj” stronę jeszcze raz. Uwierz mi, warto. **Im więcej czasu poświęcisz przyjrzeniu się temu, co dzieje się w kodzie, tym większą będziesz miał satysfakcję.** Następne części rozdziału są śmiertelnie nudne, ponieważ zawierają opisy metod wywoływanych w fazach wyświetlenia, aktualizowania i demontowania komponentu. Nie mów potem, że Cię nie ostrzegałem.

Faza pierwszego wyświetlenia

Gdy komponent rozpoczyna swoje życie w modelu DOM, wywołuje metody pokazane na rysunku 11.4.



Rysunek 11.4. Metody cyklu życia wywoływane w fazie pierwszego wyświetlenia

To, co widziałeś w konsoli przeglądarki po otwarciu strony, było mniej kolorową wersją tego, co widzisz na rysunku. Idźmy dalej i dowiedzmy się więcej: co robią metody wywoływane w poszczególnych cyklach życia komponentu?

Określenie domyślnych właściwości

Domyślną wartość właściwości `this.props` określa się za pomocą właściwości `defaultProps`. Właściwości `name` komponentu `CounterParent` można przypisać wartość w następujący sposób:

```
CounterParent.defaultProps = {  
  name: "Iron Man"  
};
```

Powyższy kod jest wykonywany jeszcze przed utworzeniem komponentu i przekazaniem właściwości przez komponent nadzędny.

Określenie domyślnego stanu

Tę operację wykonuje konstruktor komponentu. W kodzie tworzącym komponent można określić domyślne właściwości obiektu `this.state`, jak niżej:

```
constructor(props) {
  super(props);

  console.log("konstruktor: domyślny stan!");

  this.state = {
    count: 0
  };

  this.increase = this.increase.bind(this);
}
```

Zwróć uwagę, że w powyższym kodzie definiowany jest obiekt `state`, a jego właściwości `count` jest przypisywana początkowa wartość 0.

Metoda componentWillMount()

Tuż przed utworzeniem komponentu w modelu DOM wywoływana jest metoda `componentWillMount()`. Należy tu wrócić uwagę na ważną kwestię: jeżeli wewnątrz tej metody wywoła się metodę `setState()`, wtedy komponent nie zostanie ponownie wyświetlony.

Metoda render()

Tę metodę już dobrze znasz. W każdym komponencie musi być ona zdefiniowana. Metoda zwraca kod JSX. Jeżeli w komponencie nie trzeba niczego wyświetlać, wtedy metoda `render()` musi zwracać wartość `null` lub `false`.

Metoda componentDidMount()

Ta metoda jest wywoływana natychmiast po wyświetleniu komponentu i umieszczeniu go w modelu DOM. Od tej chwili można do modelu wysyłać dowolne polecenia, nie martwiąc się o to, czy komponent je wykona. W tej metodzie można umieścić kod, którego pomyślne wykonanie jest uzależnione od gotowości komponentu.

Wszystkie powyższe metody, z wyjątkiem `render()`, są *wywoływane tylko raz* i tym się różnią od metod opisanych niżej.

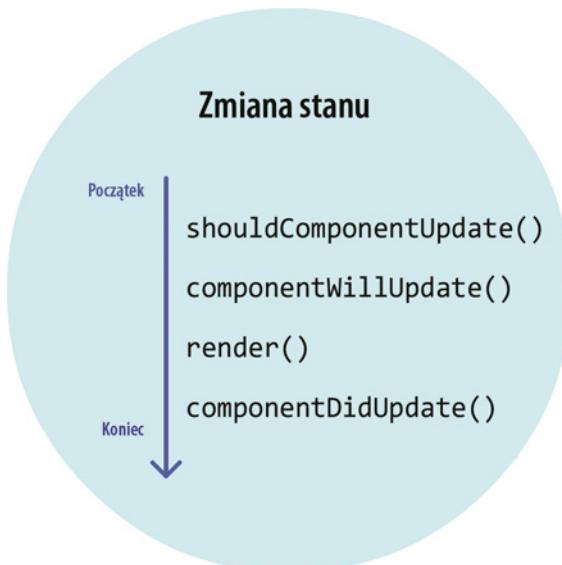
Faza aktualizacji

Gdy komponent zostanie umieszczony w modelu DOM, można go aktualizować i ponownie wyświetlać, kiedy zmienią się jego właściwości lub stan. W takim przypadku wywoływanie są różne metody cyklu życia. Nuda. Przykro mi...

Obsługa zmiany stanu

Najpierw przyjrzyjmy się zmianie stanu. Jak wspomniałem wcześniej, gdy komponent zmieni stan, wywołuje ponownie metodę `render()`. Również wszystkie inne komponenty, których wygląd zależy od zmienionego komponentu, wywołują własne metody `render()`. Dzięki temu komponent zawsze jest wyświetlany w aktualnej postaci. Jest to jednak tylko mały fragment tego, co dzieje się w przeglądarce.

Gdy komponent zmieni stan, wywołuje metody przedstawione na rysunku 11.5.



Rysunek 11.5. Metody wywoływane przez komponent po zmianie stanu

Poniżej zostały opisane poszczególne metody.

Metoda `shouldComponentUpdate()`

Czasami nie ma potrzeby, aby aktualizować komponent po zmianie stanu. Za pomocą metody `shouldComponentUpdate()` można kontrolować proces aktualizacji. Jeżeli metoda zwróci wynik `true`, komponent jest aktualizowany. W przeciwnym wypadku aktualizacja jest pomijana.

Powyższy opis może być niejasny, dlatego przeanalizujmy następujący fragment kodu:

```
shouldComponentUpdate(newProps, newState) {
  console.log("shouldComponentUpdate(): czy komponent trzeba zaktualizować?");
  if (newState.count < 5) {
    console.log("shouldComponentUpdate(): komponent trzeba zaktualizować!");
    return true;
  } else {
    ReactDOM.unmountComponentAtNode(destination);
    console.log("shouldComponentUpdate(): komponentu nie trzeba aktualizować!");
    return false;
}
```

Metoda ma dwa argumenty: obiekty `newProps` i `newState`. Metoda sprawdza, czy nowa wartość właściwości `count` obiektu `newState` jest mniejsza od 5. Jeżeli ten warunek jest spełniony, metoda zwraca wartość `true`, oznaczającą, że komponent ma być zaktualizowany. W przeciwnym wypadku metoda zwraca wartość `false`, która oznacza, że aktualizacja komponentu zostanie pominięta.

Metoda componentWillUpdate()

Ta metoda jest wywoływana tuż przed aktualizacją komponentu. Nic ciekawego się w niej nie dzieje. Należy jedynie pamiętać, że nie można w niej zmieniać stanu komponentu za pomocą metody `this.setState()`.

Metoda render()

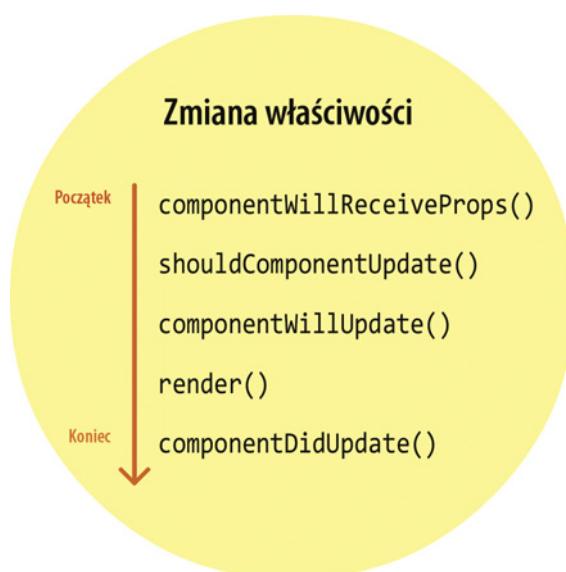
Jeżeli nie zostanie zaimplementowana metoda `shouldComponentUpdate()`, wywoływana jest ponownie metoda `render()`, aby komponent poprawnie wyświetlał się w przeglądarce.

Metoda componentDidUpdate()

Ta metoda jest wywoływana po zaktualizowaniu komponentu i wywołaniu metody `render()`. Jeżeli po zaktualizowaniu komponentu muszą być wykonane dodatkowe operacje, należy je zakodować w tej metodzie.

Obsługa zmiany właściwości

Innym przypadkiem, w którym jest aktualizowany komponent, jest zmiana jego właściwości po umieszczeniu go w modelu DOM. Wywoływane są wtedy metody przedstawione na rysunku 11.6.



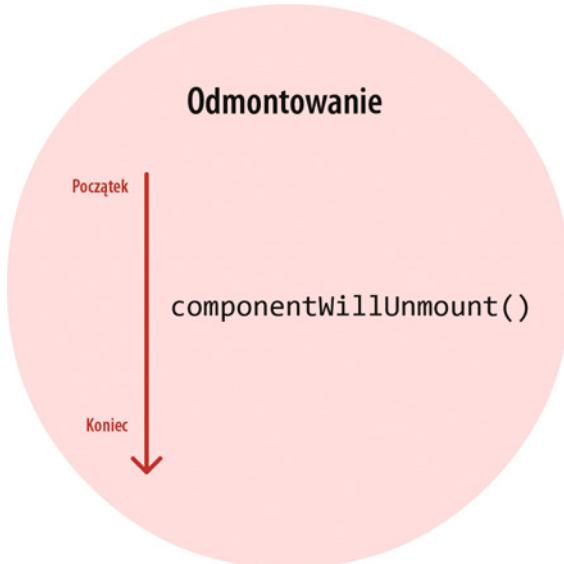
Rysunek 11.6. Metody wywoływane w przypadku zmiany właściwości komponentu

Jedyną nową metodą jest tutaj `componentWillReceiveProps()`. Ma ona jeden argument zawierający nowe wartości, które zostaną przypisane właściwościom komponentu.

Pozostałe metody poznaleś w części opisującej zmianę stanu komponentu, zatem nie ma potrzeby, aby je przedstawiać ponownie. W przypadku zmiany właściwości komponentu działają one tak samo jak w przypadku zmiany jego stanu.

Faza odmontowania

Ostatnia faza ma miejsce w momencie, gdy komponent jest usuwany z modelu DOM (patrz rysunek 11.7).



Rysunek 11.7. W chwili usunięcia komponentu z modelu DOM wywoływana jest tylko jedna metoda

W tej fazie wywoływana jest tylko jedna metoda: `componentWillUnmount()`. Umieszcza się w niej kod „sprzątający” po komponencie, na przykład usuwający procedury obsługi zdarzeń i czasomierze. Po wywołaniu tej metody komponent jest ostatecznie usuwany z modelu DOM.

Podsumowanie

Komponenty to fascynujące małe twory. Na pozór wydaje się, że niewiele się w nich dzieje. Jeżeli jednak przyjrzeć się im bliżej i dokładniej, odkrywa się całkiem nowy świat. Jak się okazuje, biblioteka React nieprzerwanie kontroluje działanie komponentów i wysyła powiadomienia, gdy coś ciekawego się w nich dzieje. Wykorzystuje w tym celu metody cyklu życia (wyjątkowo nudne), którym poświęciłem cały ten rozdział. Pragnę Cię jednak zapewnić, że wiedza o tym, co robi każda metoda i kiedy jest wywoływana, przyda się w pewnym momencie. Wszystko, czego się w tym rozdziale dowiedziałeś, nie jest czczą teorią, aczkolwiek możesz wywrzeć wrażenie na swoich znajomych, wymieniając z pamięci wszystkie metody cyklu życia i je opisując. Zrób to przy najbliższej okazji.

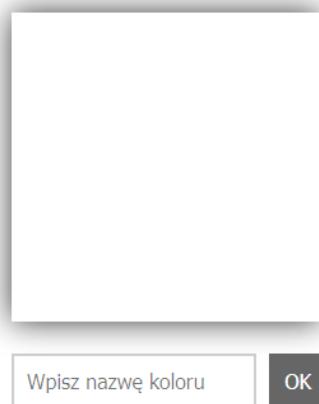
Jeżeli napotkasz problemy, pyтай!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pyтай сміло! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

12

Dostęp do elementów DOM w bibliotece React

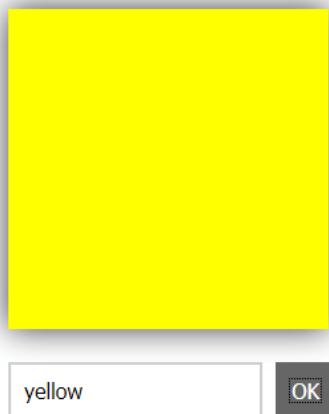
Czasami w trakcie pisania kodu pojawia się potrzeba uzyskania bezpośredniego dostępu do właściwości i metod elementu HTML. Dlaczego w świecie biblioteki React, z pięknym, czystym, znacznikowym językiem JSX, w ogóle trzeba zajmować się czymś tak okropnym, jak kod HTML? Jak się przekonasz (o ile już się nie przekonałeś), czasami łatwiej jest odwoływać się do elementów HTML bezpośrednio za pomocą kodu JavaScript i interfejsu DOM API, niż stosować bibliotekę React i jej sztywne zasady. Aby poznać jeden z takich przypadków, przeanalizujmy przykładową aplikację *Koloryzator*, przedstawioną na rysunku 12.1.



Rysunek 12.1. Aplikacja Koloryzator

Aplikację tę znajdziesz w załączonym do książki pliku *r12\index_1.htm*.

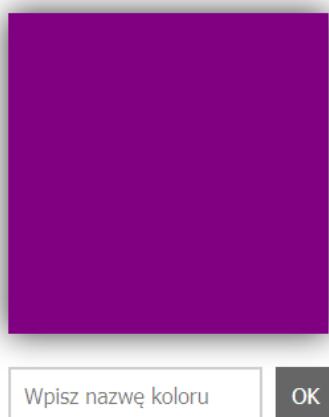
Działanie aplikacji polega na wypełnianiu białego (na początku) kwadratu zadanym kolorem. Aby się o tym przekonać, wpisz w polu nazwę koloru (w języku angielskim) i kliknij przycisk OK. (Jeżeli nie wiesz, jaki kolor wybrać, wpisz **yellow**). Kwadrat zmieni kolor z białego na zadany (patrz rysunek 12.2).



Rysunek 12.2. Biały kwadrat, który zrobił się żółty

Możliwość wypełniania kwadratu zadanym kolorem robi wrażenie, ale nie to jest najważniejsze. Zwróć uwagę na wygląd pola tekstowego i przycisku po wpisaniu nazwy koloru. Przycisk pozostaje wyróżniony, a pole tekstowe cały czas zawiera podaną wcześniej nazwę koloru. Aby wypełnić kwadrat innym kolorem, trzeba kliknąć pole tekstowe, usunąć znajdująca się w nim nazwę koloru i wpisać nową. Ech! To są zbędne czynności, dlatego trzeba poprawić funkcjonalność aplikacji.

Czy nie byłoby lepiej, gdyby pole tekstowe było czyszczone i wyróżniane za każdym razem, kiedy klikniemy przycisk? Na przykład: kiedy wpisalibyśmy nazwę *purple* i kliknęliśmy przycisk, strona wyglądała jak na rysunku 12.3.



Rysunek 12.3. Kwadrat jest fioletowy, a pole tekstowe gotowe do wpisania następnego koloru

Nazwa koloru została usunięta, a pole tekstowe z powrotem wyróżnione. Teraz łatwiej jest wpisywać następne kolory, ponieważ nie trzeba wciąż przeskakiwać pomiędzy polem tekstowym a przyciskiem. Tak jest o wiele lepiej, prawda?

Uzyskanie tego efektu za pomocą języka JSX i tradycyjnych technik biblioteki React jest trudne. Nie mam nawet zamiaru zanudzać Cię opisem, jak to zrobić. Natomiast można to bardzo łatwo osiągnąć, odwołując się do elementów HTML bezpośrednio za pomocą kodu JavaScript i interfejsu DOM API. Domyślasz się, co mam zamiar zrobić? W następnych częściach rozdziału poznasz funkcjonalność **referencji** oferowaną przez bibliotekę React, umożliwiającą uzyskiwanie bezpośredniego dostępu do elementów HTML za pomocą interfejsu DOM API. Przyjrzymy się również **portalom**, dzięki którym można umieszczać treści w dowolnych elementach HTML na stronie.

Aplikacja Koloryzator

Aby zapoznać się z referencjami i portalami, zmieńmy kod przedstawionej wcześniej aplikacji *Koloryzator*. Nowy kod wygląda jak niżej:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Koloryzator!</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }

    .colorSquare {
      box-shadow: 0px 0px 25px 0px #333;
      width: 242px;
      height: 242px;
      margin-bottom: 15px;
    }

    .colorArea input {
      padding: 10px;
      font-size: 16px;
      border: 2px solid #CCC;
    }

    .colorArea button {
      padding: 10px;
      font-size: 16px;
      margin: 10px;
      background-color: #666;
      color: #FFF;
      border: 2px solid #666;
    }
  </style>
</head>
<body>
  <div id="container">
    <div class="colorArea">
      <input type="text" value="#333333" />
      <button>Zapisz!</button>
    </div>
  </div>
</body>
</html>
```

```

.colorArea button:hover {
  background-color: #111;
  border-color: #111;
  cursor: pointer;
}
</style>
</head>

<body>
<div id="container"></div>
<script type="text/babel">

  class Colorizer extends React.Component {
    constructor(props) {
      super(props);

      this.state = {
        color: "",
        bgColor: "white"
      };

      this.colorValue = this.colorValue.bind(this);
      this.setNewColor = this.setNewColor.bind(this);
    }

    colorValue(e) {
      this.setState({
        color: e.target.value
      });
    }

    setNewColor(e) {
      this.setState({
        bgColor: this.state.color
      });

      e.preventDefault();
    }

    render() {
      var squareStyle = {
        backgroundColor: this.state.bgColor
      };

      return (
        <div className="colorArea">
          <div style={squareStyle} className="colorSquare"></div>

          <form onSubmit={this.setNewColor}>
            <input onChange={this.colorValue}>
              placeholder="Wpisz nazwę koloru"
            </input>
            <button type="submit">OK</button>
          </form>
        </div>
      );
    }
  }

  ReactDOM.render(
    <div>

```

```

        <Colorizer />
    </div>,
    document.querySelector("#container")
);
</script>
</body>

</html>

```

Przyjrzyj się przez chwilę kodowi i sprawdź, jak się ma do opisanego wcześniej przykładu. Nic nie powinno Cię w nim zaskoczyć. Jeżeli już zapoznałeś się z kodem, nadszedł czas, abyś dowiedział się o referencjach.

Poznaj referencje

Wiesz już bardzo dobrze, że w metodzie `render()` można umieszczać kod JSX przypominający HTML. W naszym przykładzie kod JSX opisuje zawartość modelu DOM. Nie jest to kod HTML, choć jest bardzo do niego podobny. Do powiązania elementów zdefiniowanych z kodzie JSX z elementami HTML służą oferowane przez bibliotekę React referencje.

Referencje funkcjonują w dość nietypowy sposób, który najłatwiej jest poznać, po prostu go stosując. Przyjrzyj się metodzie `render()` użytej w naszej aplikacji *Koloryzator*:

```

render() {
  var squareStyle = {
    backgroundColor: this.state.bgColor
  };

  return (
    <div className="colorArea">
      <div style={squareStyle} className="colorSquare"></div>

      <form onSubmit={this.setNewColor}>
        <input onChange={this.colorValue}
          placeholder="Wpisz nazwę koloru"></input>
        <button type="submit">OK</button>
      </form>
    </div>
  );
}

```

Metoda ta zwraca spory kawałek kodu JSX, który definiuje między innymi pole tekstowe do wpisywania nazwy koloru. Musisz uzyskać dostęp do tego pola w modelu DOM, abyś mógł wykonać na nim kilka operacji za pomocą kodu JavaScript.

Aby uzyskać dostęp do żądanego elementu, należy zdefiniować w nim referencję za pomocą argumentu `ref`, jak niżej:

```

render() {
  var squareStyle = {
    backgroundColor: this.state.bgColor
  };

  return (
    <div className="colorArea">
      <div style={squareStyle} className="colorSquare"></div>

```

```

<form onSubmit={this.setNewColor}>
  <input onChange={this.colorValue}
    ref={}
    placeholder="Wpisz nazwę koloru"></input>
  <button type="submit">OK</button>
</form>
</div>
);
}

```

Interesuje nas pole tekstowe, więc właśnie w tym elemencie umieściliśmy atrybut `ref`. Na razie jest on pusty, ale zazwyczaj zawiera nazwę funkcji zwrotnej JavaScript, która jest wywoływana automatycznie w chwili zamontowania elementu w modelu DOM. Gdy umieścisz w tym atrybucie prostą funkcję JavaScript zapisującą w zmiennej referencję do bieżącego elementu, kod będzie wyglądał jak niżej:

```

render() {
  var squareStyle = {
    backgroundColor: this.state.bgColor
  };

  var self = this;

  return (
    <div className="colorArea">
      <div style={squareStyle} className="colorSquare"></div>

      <form onSubmit={this.setNewColor}>
        <input onChange={this.colorValue}
          ref={
            function(e1) {
              self._input = e1;
            }
          }
          placeholder="Wpisz nazwę koloru"></input>
        <button type="submit">OK</button>
      </form>
    </div>
  );
}

```

Gdy po zamontowaniu elementu zostanie wykonany powyższy kod, będzie można w dowolnym miejscu komponentu odwoływać się do pola tekstowego za pomocą zmiennej `self._input`. Popatrz sam przez chwilę na operacje wykonywane przez wyróżnione wyżej wiersze. A teraz przeanalizujmy je wspólnie.

Funkcja zwrotna wygląda jak niżej:

```

function(el) {
  self._input = el;
}

```

Jest to funkcja anonimowa, wywoływana po zamontowaniu elementu. Jej argumentem jest referencja do utworzonego w modelu DOM elementu HTML. W tym przypadku argument ten ma nazwę `e1`, ale może być ona dowolna. Jedynym zadaniem tej funkcji jest zapisywanie we właściwości o nazwie `_input` referencji do elementu. Do zdefiniowania tej właściwości została użyta zmienna `self` i domknięcie. Identyfikator `this` reprezentuje tu komponent, a nie samą funkcję. Uff!

Zastanówmy się, co możemy zrobić z polem tekstowym, kiedy już uzyskamy do niego dostęp. Naszym celem jest usunięcie jego zawartości i wyróżnienie go, gdy użytkownik kliknie przycisk. Kod wykonujący te operacje umieścimy w metodzie `setNewColor()`. Wpisz teraz wyróżnione niżej wiersze:

```
setNewColor(e) {
  this.setState({
    bgColor: this.state.color
  });

  this._input.focus();
  this._input.value = "";

  e.preventDefault();
}
```

Wiersz `this._input.value = ""` powoduje usunięcie z pola wpisanej nazwy koloru. Pole jest wyróżniane poprzez wywołanie metody `this._input.focus()`. Cała wcześniejsza praca związana z uzyskaniem referencji była po to, aby można było wpisać powyższe dwa wiersze. Potrzebny był sposób wskazania za pomocą właściwości `this._input` odpowiedniego elementu HTML, czyli pola tekstowego zdefiniowanego za pomocą kodu JSX. Dzięki temu można teraz użyć właściwości `value` i metody `focus()` dostępnych w interfejsie API tego elementu.

Uproszczenie kodu za pomocą funkcji strzałkowej ES6

Opanowanie różnych aspektów biblioteki React jest na tyle trudne, że chyba nie będę musiał Cię namawiać do użycia języka ES6. Dzięki dostępnej w nim funkcji strzałkowej praca z atrybutem `ref` jest nieco łatwiejsza. Jest to jeden z przypadków, w którym polecam stosowanie języka ES6.

Jak sam widziałeś, aby zapisać we właściwości komponentu referencję do elementu HTML, użyliśmy następującego kodu:

```
<input
  ref={
    function(el) {
      self._input = el;
    }
  }
/>
```

Aby zrobić sztuczkę z kontekstami, zdefiniowaliśmy zmienną `self` i przypisaliśmy jej obiekt `this`, dzięki czemu można było utworzyć właściwość `_input` komponentu. Wygląda to bardzo skomplikowanie. Za pomocą funkcji strzałkowej wszystkie te operacje można zakodować prościej:

```
<input
  ref={
    (el) => this._input = el
  }
/>
```

Końcowy efekt jest taki sam jak wcześniejszy, któremu poświęciliśmy tyle czasu. Za względu na to, w jaki sposób funkcja strzałkowa traktuje kontekst, można w jej kodzie stosować identyfikator `this` i odwoływać się do elementu bez wpisywania dodatkowych instrukcji. I nie trzeba definiować zewnętrznej zmiennej `self`!

Portale

Musisz pamiętać o pewnej kwestii związanej z modelem DOM. Do tej pory odwoływaliśmy się do elementu HTML tylko w kontekście zawartości utworzonej za pomocą kodu JSX, niezależnie od tego, czy był to pojedynczy komponent czy kilka połączonych. Oznacza to, że można było wykonywać operacje wyłącznie w zakresie wyznaczonym przez hierarchię komponentów nadrzędnych. Wydaje się więc, że uzyskanie swobodnego dostępu do dowolnego elementu strony jest niemożliwe. Czy na pewno? Jak się okazuje, kod JSX można umieszczać w dowolnym elemencie modelu DOM, nie tylko takim, który znajduje się w komponencie nadrzędnym.

Ta magiczna sztuczka jest możliwa dzięki funkcjonalności zwanej **portalam**.

Z portalu korzysta się w bardzo podobny sposób jak z metody `ReactDOM.render()`. Należy wpisać kod JSX i określić element, w którym ten kod ma być wykorzystany.

Aby dowiedzieć się, jak to zrobić, wróć do naszej przykładowej aplikacji i umieść w niej element `h1` na tym samym poziomie hierarchii co element `div`:

```
<body>
  <h1 id="colorHeading">Koloryzator</h1>
  <div id="container"></div>
  ...

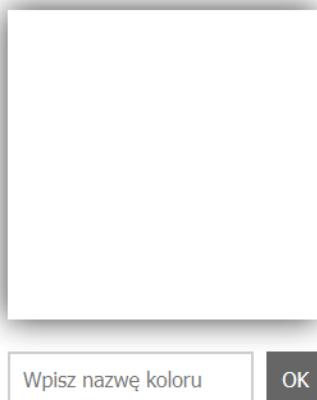
```

Dopisz jeszcze regułę w elemencie `style`, aby element `h1` wyglądał ładniej:

```
#colorHeading {
  padding: 0;
  margin: 50px;
  margin-bottom: -20px;
  font-family: sans-serif;
}
```

Otwórz teraz stronę w przeglądarce i sprawdź, czy dodane kody HTML i CSS dają zamierzony efekt (patrz rysunek 12.4).

Koloryzator



Rysunek 12.4. Tak teraz wygląda nasza aplikacja!

Teraz chcemy, aby w nagłówku h1 była wyświetlana nazwa aktualnie użytego koloru. Pamiętaj, że element h1 znajduje się na tym samym poziomie hierarchii co element div, wewnątrz którego wyświetlana jest cała treść aplikacji.

Aby osiągnąć założony cel, wróć do metody render() komponentu Colorizer i dopisz w niej wyróżniony niżej wiersz:

```
return (
  <div className="colorArea">
    <div style={squareStyle} className="colorSquare"></div>

    <form onSubmit={this.setNewColor}>
      <input onChange={this.colorValue}
        ref={
          function(el) {
            self._input = el;
          }
        }
        placeholder="Wpisz nazwę koloru"></input>
      <button type="submit">OK</button>
    </form>
    <ColorLabel color={this.state.bgColor}/>
  </div>
);
```

Wiersz ten tworzy instancję komponentu o nazwie ColorLabel i deklaruje jego właściwość color, której przypisuje wartość właściwości bgColor. Komponent ColorLabel nie jest jeszcze zdefiniowany, więc przed wywołaniem metody ReactDOM.render() wpisz poniższy kod:

```
var heading = document.querySelector("#colorHeading");

class ColorLabel extends React.Component {
  render() {
    return ReactDOM.createPortal(
      ": " + this.props.color,
      heading
    );
  }
}
```

Zmienna heading zawiera odwołanie do elementu h1. To nic nowego. Nowością jest metoda render() komponentu ColorLabel, a konkretnie jej instrukcja return, która zwraca wynik metody ReactDOM.createPortal():

```
class ColorLabel extends React.Component {
  render() {
    return ReactDOM.createPortal(
      ": " + this.props.color,
      heading
    );
  }
}
```

Metoda ReactDOM.createPortal() ma dwa argumenty. Pierwszy argument zawiera kod JSX treści, która ma być wyświetlona na stronie, a drugi — element modelu DOM, w którym ma zostać umieszczona wyświetlana treść. Nasz kod JSX zawiera kilka znaków oraz właściwość z nazwą koloru:

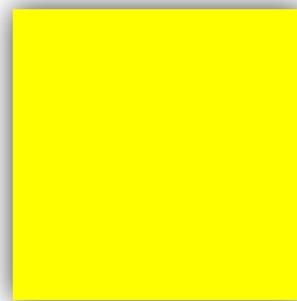
```
class ColorLabel extends React.Component {  
  render() {  
    return ReactDOM.createPortal(  
      ": " + this.props.color,  
      heading  
    );  
  }  
}
```

Elementem, w którym ma zostać umieszczona powyższa treść, jest nagłówek h1. Jego referencję zawiera zmienna heading:

```
class ColorLabel extends React.Component {  
  render() {  
    return ReactDOM.createPortal(  
      ": " + this.props.color,  
      heading  
    );  
  }  
}
```

Otwórz teraz stronę, zmień kolor kwadratu i zwróć uwagę, co się stanie. W nagłówku pojawi się nazwa koloru, którą wpiszesz (patrz rysunek 12.5).

Koloryzator: yellow

 OK

Rysunek 12.5. Nagłówek zawierający nazwę koloru

Zwróć uwagę na ważną rzecz: element h1 znajduje się poza głównym kodem wyświetlającym element div. Dzięki portalom można uzyskać dostęp do wszystkich elementów modelu DOM i umieszczać w nich treści z pominięciem tradycyjnej hierarchii komponentów, z którą miałeś do tej pory do czynienia.

Podsumowanie

W większości przypadków potrzebne treści będziesz generował za pomocą kodu JSX. Czasami jednak będziesz musiał wyzwolić się z ograniczeń nakładanych przez bibliotekę React. Wykonywane dotąd operacje polegały jedynie na generowaniu treści wewnątrz dokumentu HTML. Biblioteka React jest jednak niczym samowystarczalna wyspa na morzu i nie odwołuje się do kodu HTML, który leży poza jej zasięgiem. Aby można było wykonywać operacje zarówno na wewnętrznym, jak i zewnętrznym kodzie HTML, wykorzystuje się dwie funkcjonalności: **referencje** i **portale**. Za pomocą referencji można przeciąć więzy reguł biblioteki i uzyskać dostęp do elementów HTML zdefiniowanych za pomocą kodu JSX. Natomiast portale pozwalają umieszczać treści we wszystkich elementach modelu DOM, do których możliwy jest dostęp. Stosując te dwie funkcjonalności, będziesz mógł rozwiązać każdy problem wymagający bezpośredniego dostępu do modelu DOM.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniemi, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

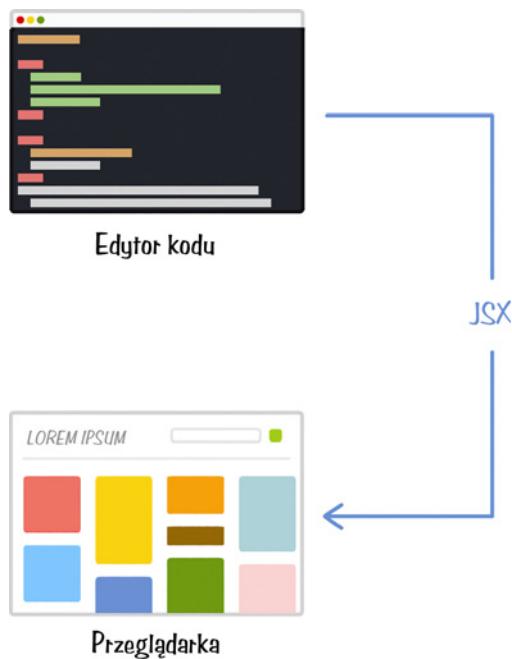
13

Konfiguracja środowiska React bez stresu

Ostatni ważny temat, którym się zajmiemy, jest mniej zвязany z biblioteką React, a bardziej z konfiguracją opartego na niej środowiska programistycznego. Do tej pory, aby utworzyć aplikację, dołączalaś do kodu strony kilka plików skryptowych:

```
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

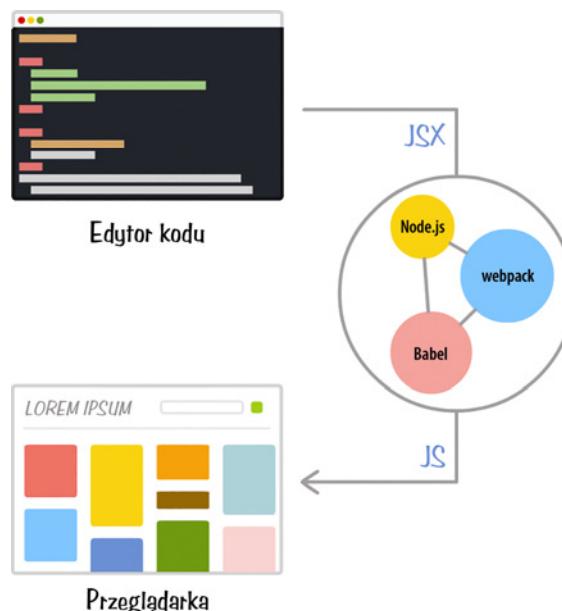
Skrypty te ładują nie tylko bibliotekę React, lecz także kompilator Babel, aby przeglądarka poprawnie przetwarzała takie osobliwości jak kod JSX (patrz rysunek 13.1).



Rysunek 13.1. Operacje wykonywane przez kompilator języka JSX

Jak wspomniałem na początku książki, mankamentem tego rozwiązania jest jego niska wydajność. Ponieważ przeglądarka w trakcie swojego normalnego działania przetwarza wszystkie ładowane pliki, musi również przekładać kod JSX na JavaScript. Jest to czasochłonna konwersja, akceptowalna na etapie rozwijania aplikacji, ale nie do przyjęcia, gdy korzystają z niej użytkownicy, ponieważ u wszystkich działa ona wtedy powoli i mało wydajnie.

Rozwiązaniem tego problemu jest odpowiednie przygotowanie środowiska programistycznego, w którym konwersja JSX do JavaScript jest częścią procesu komplikacji kodu aplikacji (patrz rysunek 13.2).



Rysunek 13.2. Poprawnie skonfigurowane środowisko do kodowania w języku JSX!

Dzięki temu rozwiązaniu przeglądarka ładuje aplikację zawierającą przekonwertowany (oraz zoptymalizowany) plik JavaScript. To dobry pomysł, prawda? Jedynym powodem, dla którego o nim jeszcze nie pisałem, jest jego **złożoność**. Poznawanie biblioteki React samo w sobie nie jest łatwe, a opisy skomplikowanych narzędzi programistycznych i konfiguracji środowiska stanowiłyby dodatkowe utrudnienie. Ponieważ teraz masz już solidną wiedzę o podstawach biblioteki React, możemy zająć się tym tematem.

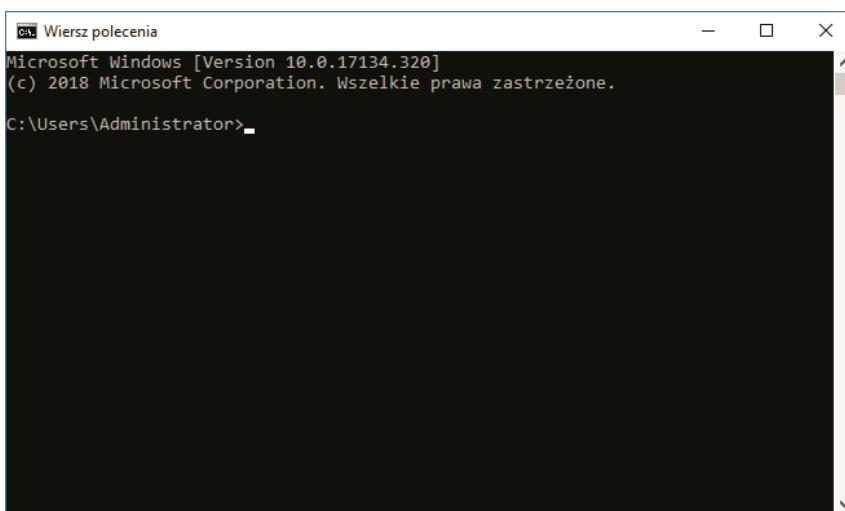
W następnych częściach tego rozdziału poznasz jeden ze sposobów przygotowania środowiska programistycznego opartego na pakietach Node.js, Babel i webpack. Nie przejmuj się, jeżeli nazwy te brzmią zniechęcająco. Wykorzystamy bardzo pomysłowe rozwiązanie opracowane przez twórców Facebooka, dzięki któremu cała konfiguracja będzie dzieciście prosta.

Zatem do dzieła!

Przedstawiamy projekt Create React

Jeszcze kilka lat temu przygotowanie środowiska programistycznego było nie lada wyzwaniem, ponieważ wymagało ręcznego konfigurowania wszystkich wymienionych wcześniej pakietów. Wtedy musiałbyś prosić o pomoc kogoś naprawdę dobrze obytego z tym tematem. Mógłbyś nawet zwałępić w sens korzystania z biblioteki React. Na szczęście pojawił się projekt Create React (<https://github.com/facebookincubator/create-react-app>), który znacznie uproszczył proces konfigurowania środowiska programistycznego. Dzięki niemu wystarczy wpisać kilka poleceń, a projekt oparty na bibliotece React zostanie automatycznie utworzony i odpowiednio skonfigurowany.

Najpierw musisz zainstalować najnowszą wersję pakietu Node.js (<https://nodejs.org>). Następnie otwórz wiersz poleceń. Jeżeli nie korzystasz z tego typu narzędzia, nie przejmuj się. W systemie Windows otwórz aplikację *Wiersz polecenia*, w systemie macOS *Terminal*. Powinno pojawić się okno podobne do poniższego:



Jest to dość tajemnicze okno z migającym kursem służącym do wpisywania różnych poleceń. Pierwszą rzeczą, którą musisz zrobić, jest zainstalowanie projektu *Create React*. Wpisz w oknie poniższe polecenie i naciśnij *Enter*:

```
npm install -g create-react-app
```

Instalacja projektu zajmie od kilku sekund do kilku minut. Gdy się zakończy, będziesz mógł utworzyć projekt React. Przejdz do folderu, w którym chcesz go zapisać — może to być na przykład *Pulpit* lub *Dokumenty*. Następnie wpisz poniższe polecenie:

```
create-react-app helloworld
```

Na ekranie pojawi się widok podobny do poniższego:

```
Wiersz polecenia
Microsoft Windows [Version 10.0.17134.320]
(c) 2018 Microsoft Corporation. Wszelkie prawa zastrzeżone.

c:\Users\Administrator>cd Documents
c:\Users\Administrator\Documents>create-react-app helloworld

Creating a new React app in c:\Users\Administrator\Documents\helloworld.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

[████.....] | fetchMetadata: sill cleanup remove extracted module
```

Gdy polecenie zostanie wykonane, powstanie projekt o nazwie *helloworld*. Na razie nie będę opisywał jego zawartości, zajmiemy się tym później. Teraz najważniejszą sprawą jest sprawdzenie aplikacji. Wpisz poniższe polecenie, aby przejść do nowo utworzonego folderu *helloworld*:

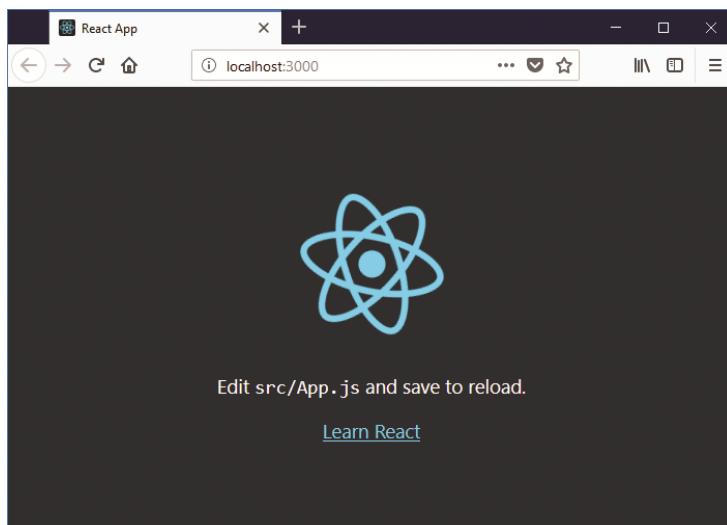
```
cd helloworld
```

Następnie wpisz poniższe polecenie, aby przetestować aplikację:

```
npm start
```

Jeżeli w Twoim systemie jest zainstalowany program *Yarn*, będzie on preferowanym programem dla polecenia *Create* i w oknie pojawi się komunikat, aby zamiast polecenia `npm start` użyć `yarn start`.

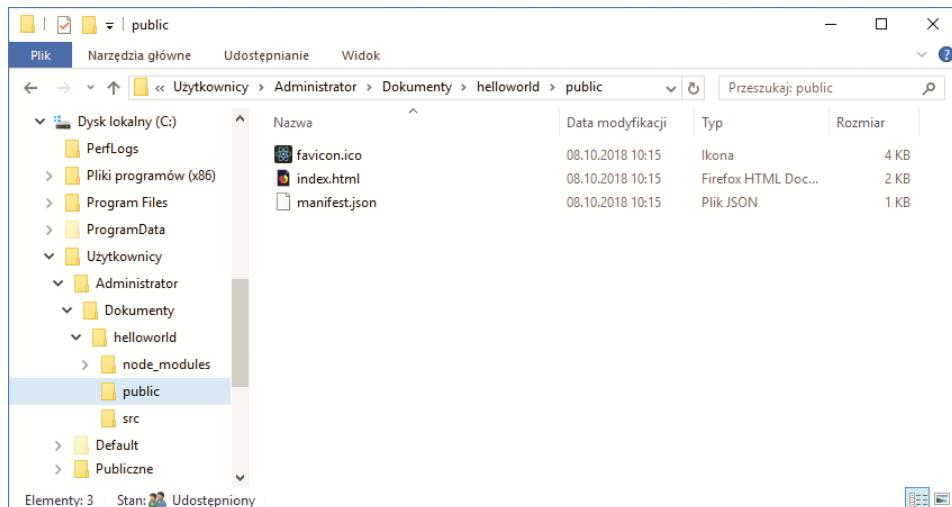
Kiedy wpiszesz powyższe polecenie, zostanie skompilowany projekt, uruchomiony lokalny serwer WWW i otwarta w przeglądarce następująca strona:



Jeżeli wszystkie operacje zostały poprawnie wykonane, powinieneś uzyskać taki sam efekt jak wyżej. Jeśli pierwszy raz tworzysz projekt za pomocą wiersza poleceń — gratulacje! Zrobiłeś naprawdę duży krok do przodu. Ale to nie koniec. Teraz musisz zatrzymać się na chwilę i dowiedzieć się, co tu właściwie się stało.

Opis utworzonego projektu

Zobaczyłeś właśnie efekt utworzenia domyślnego projektu za pomocą polecenia `create-react-app`. Jest to nieszczególnie przydatny projekt. Przyjrzyjmy się najpierw utworzonym plikom i folderom, których strukturę przedstawia rysunek 13.3.



Rysunek 13.3. Struktura utworzonych plików i folderów

W przeglądarce został otwarty plik `index.html` z folderu `public`. Jeżeli przyjrzesz się zawartości tego pliku stwierdzisz, że jest on bardzo prosty. Po usunięciu komentarzy wygląda jak niżej:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <meta name="theme-color" content="#000000">
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <title>React App</title>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
  </body>
</html>
```

Zwróć uwagę na element `div` z atrybutem `id` o wartości `root`. Wewnątrz tego elementu Twoja aplikacja umieszcza treść przeznaczoną do wyświetlenia. Cała treść, włącznie z kodem JSX, jest zapisana w plikach w folderze `src`. Plikiem startowym aplikacji jest `index.js` o następującej zawartości:

```
contained in index.js:
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

Zwróć uwagę, że argument metody `ReactDOM.render()` odwołuje się do wspomnianego wcześniej elementu o nazwie `root`, znajdującego się w pliku `index.html`. Oprócz tego na początku pliku widocznych jest kilka instrukcji import ładujących tzw. **moduły** JavaScript. Są to pliki realizujące określone funkcjonalności w aplikacjach. Aby zastosować wybraną funkcjonalność, wystarczy zaimportować do aplikacji odpowiedni plik. Moduły mogą zawierać napisany przez Ciebie kod albo mogą być zewnętrznymi plikami, jak na przykład biblioteki React lub ReactDOM. O importowaniu modułów można wiele mówić, ale dla przejrzystości opisu zostawmy na razie ten temat.

W tej chwili importowane są biblioteki React i ReactDOM. Znasz już te biblioteki, ponieważ wcześniej wykorzystywałeś je w znacznikach `script`. Importowane są również szablonów CSS, skrypt implementujący usługę `registerServiceWorker` oraz komponent biblioteki React, któremu przypisywana jest nazwa `App`.

Komponent `App` jest następnym fragmentem aplikacji. Aby zobaczyć jego kod, otwórz plik `App.js`. Poniżej przedstawiona jest zawartość tego pliku:

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
            Edit <code>src/App.js</code> and save to reload.
          </p>
          <a
            className="App-link"
            href="https://reactjs.org"
            target="_blank"
            rel="noopener noreferrer"
          >
            Learn React
          </a>
        </header>
      </div>
    );
  }
}
```

```

    );
}

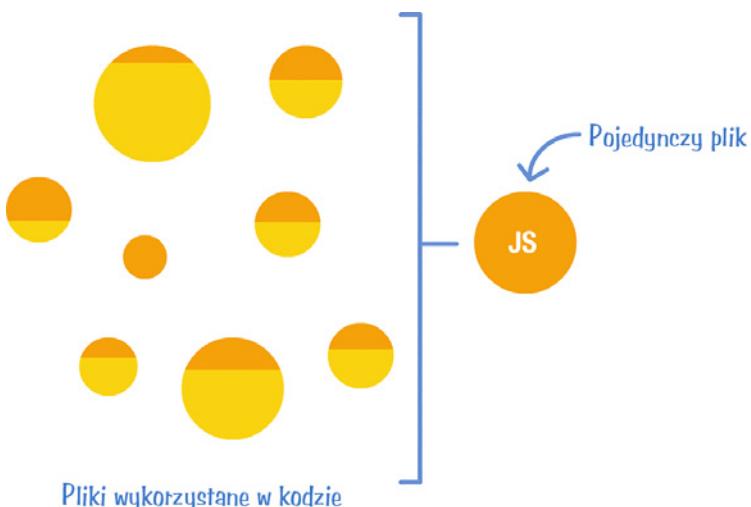
export default App;

```

Zwróć uwagę, że plik *App.js* również zawiera instrukcje `import`. Niektóre z nich, na przykład te importujące bibliotekę React i komponent `Component`, są niezbędne do działania aplikacji. Ciekawy jest ostatni wiersz `export default app`. Zawiera on instrukcję `export` i nazwę identyfikującą eksportowany moduł. Nazwa ta zostanie wykorzystana do importowania modułu `App` w innych plikach aplikacji, na przykład `index.js`. Dodatkowe operacje wykonywane w tym pliku to importowanie obrazu i szablonu stylów CSS potrzebnych do wyświetlania strony.

Poznałeś właśnie inny sposób tworzenia struktury kodu za pomocą nowych słów kluczowych. Jednak po co jest to wszystko? Dzięki modułom oraz instrukcjom `import` i `export` można łatwiej zarządzać kodem aplikacji. Zamiast tworzyć jeden duży plik, można kod podzielić na mniejsze pliki zawierające powiązane ze sobą zasoby. W zależności od tego, jakie pliki są importowane i w jakiej kolejności, magiczny proces kompilacji (uruchamiany poleceniem `npm start`) optymalizuje kod na różne sposoby, dzięki czemu nie musi tego robić programista.

Pamiętaj o ważnej rzeczy: **sposób dzielenia kodu nie ma większego wpływu na działanie aplikacji**. Gdy aplikacja jest gotowa do przetestowania, wtedy dokonywana jest kompilacja kodu obejmująca wszystkie importowane pliki i komponenty. W wyniku kompilacji powstaje zaledwie kilka plików, aby przeglądarka mogła je łatwo przetwarzać. Na przykład z wielu plików JavaScript generowany jest pojedynczy plik:



Tworzony jest również jeden zagregowany plik CSS. W zależności od konfiguracji projektu mogą być tworzone inne zagregowane pliki wykorzystywane następnie w pliku HTML. Wszystkie jednak mają format umożliwiający ich natychmiastowe przetwarzanie przez przeglądarkę. Przeglądarka nie musi wykonywać żadnych dodatkowych operacji, jak to było w przykładach opisanych wcześniej. Tworzone są wyłącznie pliki HTML, CSS i JavaScript.

Utworzenie aplikacji „Witaj, świecie!”

Teraz, kiedy masz już lepsze wyobrażenie o tym, co się dzieje w projekcie, czas na wprowadzenie w nim modyfikacji. Naszym zamiarem jest wyświetlenie na stronie napisu „Witaj, świecie!”. W tym celu należy utworzyć komponent o adekwatnej nazwie `HelloWorld`. Nowością nie będzie tu sposób wyświetlania treści na stronie, ponieważ wiesz już doskonale, jak to się robi. Skupimy się natomiast na strukturze plików projektu i poprawnym tworzeniu aplikacji.

Najpierw przejdź do folderu `src` i usuń z niego wszystkie pliki. Następnie utwórz nowy plik o nazwie `index.js` i umieść w nim poniższy kod:

```
import React from "react";
import ReactDOM from "react-dom";
import HelloWorld from "./HelloWorld";

ReactDOM.render(
  <HelloWorld/>,
  document.getElementById("root")
);
```

Ten kod importuje biblioteki React i ReactDOM. Importuje również komponent `HelloWorld`, który jest wykorzystywany w argumencie metody `ReactDOM.render()`. Komponent ten jeszcze nie istnieje, więc trzeba go teraz utworzyć.

W tym samym folderze `src` utwórz plik o nazwie `HelloWorld.js` i następującej zawartości:

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

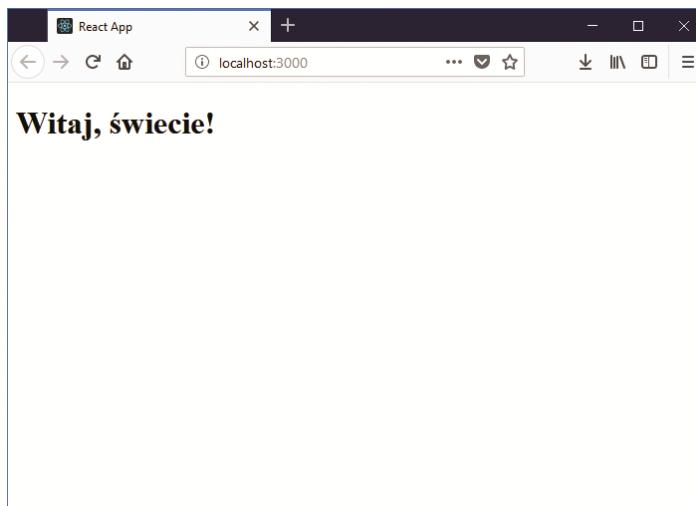
class HelloWorld extends Component {
  render() {
    return (
      <div className="helloContainer">
        <h1>Witaj, świecie!</h1>
      </div>
    );
  }
}

export default HelloWorld;
```

Przyjrzyj się przez chwilę kodowi, który wpisałeś. Nic nie powinno Cię w nim zaskoczyć — jest tu dobrze znana instrukcja `import`, komponent `HelloWorld` wyświetlający napis na stronie oraz (w ostatnim wierszu) instrukcja eksportująca ten komponent, aby można go było importować w innych modułach, na przykład w utworzonym wcześniej pliku `index.js`.

Po wprowadzeniu i zapisaniu powyższych zmian możesz przetestować aplikację. Jeżeli aplikacja cały czas działał, powinna się automatycznie zaktualizować zgodnie z wprowadzonymi zmianami. Jeżeli tak się nie stało, przejdź do wiersza poleceń, naciśnij klawisze `Ctrl+C`, aby zamknąć aplikację, i ponownie wpisz polecenie `npm start`.

Powinieneś uzyskać efekt podobny do poniższego:



Jeżeli właśnie taką stronę widzisz, to świetnie! Aplikacja działa poprawnie, ale wygląda bardzo skromnie. Zmień to, dodając nieco stylów CSS. W tym celu utwórz plik o nazwie *index.css* i wpisz w nim następujący kod:

```
body {  
  display: flex;  
  align-items: center;  
  justify-content: center;  
  min-height: 100vh;  
  margin: 0;  
}
```

W opisywanej tu metodyce budowania aplikacji tworzenie arkusza stylów jest jedynie częścią większego procesu. Inną operacją, którą musisz wykonać, jest umieszczenie w pliku *index.js* odwołania do utworzonego właśnie pliku *index.css*. Otwórz więc plik *index.js* i wpisz w nim wyróżniony niżej wiersz:

```
import React from "react";  
import ReactDOM from "react-dom";  
import HelloWorld from "./HelloWorld";  
import "./index.css";  
ReactDOM.render(  
  <HelloWorld/>,  
  document.getElementById("root")  
,
```

Jeżeli wróciś teraz do przeglądarki, zauważysz, że strona automatycznie się odświeżyła i pozostały na niej uwzględnione zmiany, które wprowadziłeś. Powinien być widoczny napis **Witaj, świecie!** wyśrodkowany w pionie i poziomie. Nie jest źle, ale może być jeszcze lepiej.

Ostatnią rzeczą, którą zrobisz, będzie nadanie tekstowi bardziej sztywnego stylu. Możesz to zrobić, definiując w pliku *index.css* odpowiednią regułę, ale lepszym rozwiązaniem będzie utworzenie nowego arkusza CSS, do którego będzie odwoływał się wyłącznie komponent *HelloWorld*. Końcowy efekt w obu przypadkach będzie taki sam, ale głównym celem jest to, abyś nabrał praktyki w grupowaniu powiązanych ze sobą plików (i definiowaniu zależności między nimi), jak przystało na profesjonalnego programistę.

Utwórz więc w folderze `src` plik o nazwie `HelloWorld.css` i wpisz w nim poniższą regułę stylu:

```
h1 {
    font-family: sans-serif;
    font-size: 56px;
    padding: 5px;
    padding-left: 15px;
    padding-right: 15px;
    margin: 0;
    background: linear-gradient(to bottom,
        white 0%,
        white 62%,
        gold 62%,
        gold 100%);
}
```

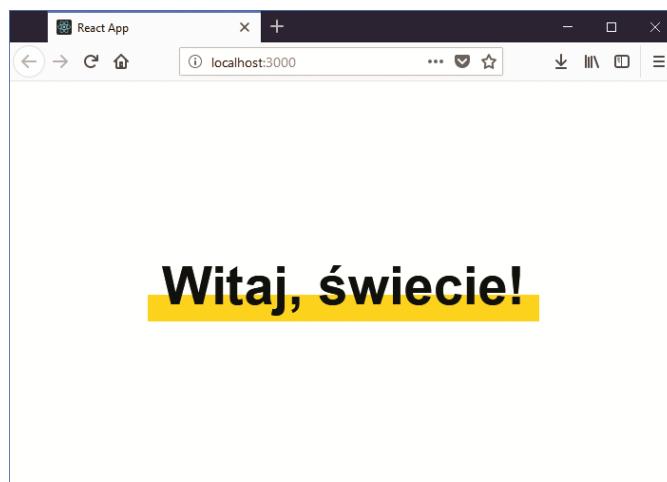
Teraz pozostało Ci jedynie umieścić odpowiednie odwołanie w pliku `HelloWorld.js`. Otwórz go więc i wpisz w nim wyróżniony niżej wiersz:

```
import React, { Component } from "react";
import "./HelloWorld.css";

class HelloWorld extends Component {
  render() {
    return (
      <div className="helloContainer">
        <h1>Witaj, świecie!</h1>
      </div>
    );
  }
}

export default HelloWorld;
```

Wróć teraz do przeglądarki. Jeżeli zobaczyś widok podobny do poniższego, to znaczy, że wszystko zrobileś dobrze:



Na stronie powinien być widoczny ten sam napis `Witaj, świecie!`, ale — mówiąc młodzieżowym językiem — wyświetlony w bardziej epickim stylu niż poprzednio.

Kompilacja wersji produkcyjnej

Prawie skończyliśmy. Pozostała jedynie jedna rzecz do zrobienia. Do tej pory tworzyłeś aplikację w **trybie programistycznym**. Tworzony w tym trybie kod nie jest „minifikowany”, a stosowane ustawienia spowalniają jego działanie. Dzięki temu jednak łatwiej jest go diagnostować. Aplikacja, z której będą korzystać realni użytkownicy, musi być możliwie szybka i zwarta. Dlatego musisz teraz wrócić do wiersza poleceń, zatrzymać aplikację, naciskając klawisze *Ctrl+C*, i wpisać następujące polecenie:

```
npm run build
```

Utworzenie zoptymalizowanych plików zajmie kilka minut. Gdy skrypt zakończy działanie, pojawi się potwierdzenie podobne do poniższego:

```
Wiersz poleceń
763 B    build\static\js\runtime~main.229c360f.js
430 B    build\static\js\main.d8b645c.chunk.js
230 B    build\static\css\main.c8fd68ae.chunk.css

The project was built assuming it is hosted at the server root.
You can control this with the homepage field in your package.json.
For example, add this to build it for GitHub Pages:

  "homepage" : "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

  http://bit.ly/CRA-deploy

c:\Users\Administrator\Documents\helloworld>
```

Zgodnie z wyświetlzoną instrukcją możesz teraz umieścić aplikację na serwerze lub dalej testować ją lokalnie, wykorzystując popularny pakiet *serve*.

Poświęć również chwilę na przejrzenie utworzonych plików. Efektem komplikacji jest kilka plików HTML, CSS i JavaScript. Żaden z nich nie zawiera kodu JSX. Ponadto jest tylko jeden plik JavaScript — zawiera cały kod, który wcześniej wpisałeś.

Podsumowanie

Tak to działa! W tym rozdziale wykorzystałeś genialny projekt *Create React* do utworzenia nowoczesnej aplikacji opartej na bibliotece React. Jeżeli pierwszy raz tworzyłeś aplikację w ten sposób, musisz dobrze go poznać. Następne przykładowe aplikacje będziesz tworzył, używając polecenia *create-react-app*. Stosowane w poprzednich rozdziałach podejście, polegające na ręcznym tworzeniu plików dla przeglądarki, miało na celu zapoznanie Cię z podstawami biblioteki React bez wnikania w szczegóły, które tutaj poznaleś. Projekt *Create React* ukrywa przed programistą wszystkie zawiłości związane z konfiguracją pakietów Node.js, Babel, webpack i różnych komponentów. Jest to jego największa zaleta, ale też największa wada.

Jeżeli nie chcesz podążać drogą wyznaczoną przez projekt *Create React*, będziesz musiał poznać całą jego ukrytą złożoność, która wykracza poza zakres tej książki. Na początek możesz zapoznać się z zawartością plików utworzonych w folderze *node_modules\react_scripts\scripts*.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

Przetwarzanie zewnętrznych danych w aplikacji React

Aplikacje WWW przetwarzające zewnętrzne dane są dzisiaj standardem. Proces przetwarzania zazwyczaj przebiega w następujący sposób:

1. Aplikacja wysyła do zewnętrznej usługi zapytanie o określone dane.
2. Usługa odbiera zapytanie i wysyła dane.
3. Aplikacja odbiera dane.
4. Aplikacja przetwarza i wyświetla dane w interfejsie użytkownika.

Być może nawet nie zdajesz sobie z tego sprawy, ale wszystkie popularne serwisy WWW, takie jak Facebook, Amazon, Twitter, Instagram, Gmail czy KIRUPA, działają w opisany wyżej sposób. Wszystkie te serwisy, gdy użytkownik otwiera ich strony, wyświetlają na początku pewne dane, jak na pierwszym rysunku na następnej stronie.

Aby pierwsza strona szybko się wyświetlała, nie są na nią ładowane wszystkie dane. Dopiero wtedy, kiedy użytkownik chce wyświetlić całą stronę albo wykonać na niej jakieś operacje, przeglądarka pobiera z serwera następne dane (patrz drugi rysunek na następnej stronie).

Wszystko to odbywa się bez odświeżania strony ani zmieniania jej stanu. Cała magia kryje się w niewielkim skrypcie JavaScript wykonującym wymienione wcześniej cztery operacje. W tym rozdziale dowiesz się wszystkiego na temat takiego skryptu i jego zastosowania w aplikacji opartej na bibliotece React.

Helion
@HelionPL

Informatyka w najlepszym wydaniu! Nasze książki kierujemy do głodnych wiedzy pasjonatów informatyki oraz stawiających pierwsze i pewniejsze kroki w fotografii.

5 074 Obserwani 189 Obserwujący 793

Tweety **Tweety i odpowiedzi** **Multimedia**

Helion @HelionPL · 4 godz.
helion.pl/go-React-twroz...

Kiedy programiści firmy Facebook opracowywali bibliotekę React, postawili sobie za cel ułatwienie życia twórcom stron pisanych w języku JavaScript. Framework stał... facebook.com/HelionPL/posts...

React. Pierwsze kroki. Kurs video. Tworzenie pr...
helion.pl

Helion @HelionPL · 7 paź

Helion @HelionPL · 25 wrz
Ulrich von Rododen :D

via Niebezpiecznik facebook.com/HelionPL/posts...

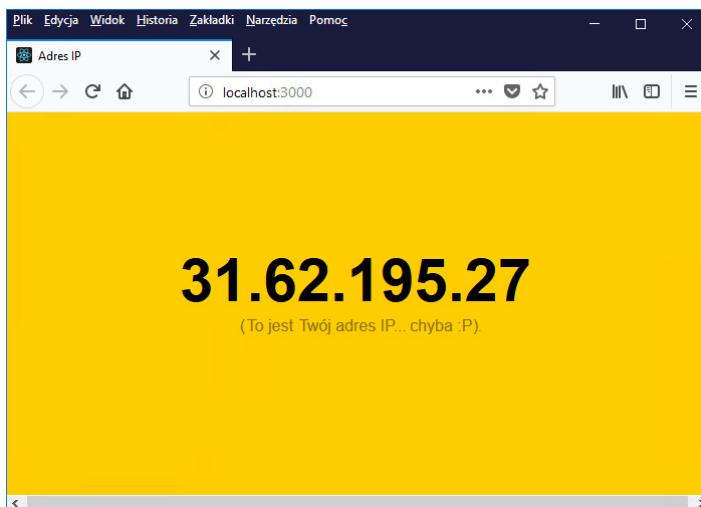
Helion @HelionPL · 24 wrz

Drodzy, w kioskach już nowy numer Programista - Magazyn, w środku między innymi wszystko o języku Rust, a także recenzja naszej książki "Adaptywny kod. Zwinne programowanie, wzorce projektowe i SOLID-ne zasady. Wydanie II" - koniecznie sprawdźcie! facebook.com/HelionPL/posts...

Helion @HelionPL · 24 wrz

Drodzy Czytelnicy, mamy dla Was urodzinowe niespodzianki:) Przygotowaliśmy dla Was programistyczne kubeczki, które możecie dobrać do zamówienia z listy rozwijanej w koszyku za jedyne 8.99 zł. Oto nasze modele!

Na koniec utworzysz przedstawioną niżej prostą aplikację:



Aplikacja wyświetla adres IP komputera użytkownika. To wszystko. Jest prosta (w porównaniu z przedstawionymi wcześniej efektownymi widokami z Twittera), ponieważ to tylko przykład, ale też na tyle skomplikowana i bogata w szczegóły, abyś mógł poznać zasady przetwarzania zewnętrznych danych za pomocą biblioteki React.

Do dzieła!

Podstawy zapytań HTTP

Jak zapewne sam bardzo dobrze wiesz, internet składa się z mnóstwa połączonych ze sobą komputerów, czyli serwerów. Gdy surfujesz po witrynach WWW i przeglądasz ich strony, w rzeczywistości zlecasz przeglądarce wysyłanie do tych serwerów zapytań o dane. Wygląda to następująco: przeglądarka wysyła zapytanie do serwera, następnie czeka na odpowiedź, a gdy ją otrzyma — odpowiednio przetwarza. Cała ta komunikacja jest możliwa dzięki **protokołowi HTTP**.

Protokół HTTP jest wspólnym językiem komunikacji pomiędzy przeglądarkami a serwerami w internecie. Zapytania wysyłane w Twoim imieniu przez przeglądarkę noszą nazwę **zapytań HTTP** i służą nie tylko prostemu wyświetlaniu stron. Częstym (i o wiele ciekawszym) ich zastosowaniem jest aktualizowanie zawartości wyświetlanych stron o odbierane dane.

Wyobraźmy sobie stronę, na której mają być wyświetlane informacje o aktualnie zalogowanym użytkowniku. Ta informacja na początku nie jest dostępna, ale przeglądarka jej zażąda od serwera, gdy użytkownik zacznie korzystać ze strony. Serwer odpowie na zapytanie przeglądarki, która umieści uzyskaną informację na stronie. Cały ten proces wygląda zapewne dość abstrakcyjnie, dlatego teraz opiszę przykładowe zapytanie HTTP i odpowiedź na nie.

Zapytanie HTTP o dane użytkownika wygląda następująco:

```
GET /user
Accept: application/json
```

W odpowiedzi na to zapytanie serwer może wysłać następujące informacje:

```
200 OK
Content-Type: application/json
```

```
{
  "name": "Kirupa",
  "url": "http://www.kirupa.com"
}
```

Taka dwustronna wymiana danych odbywa się w trakcie działania aplikacji wielokrotnie i jest w pełni obsługiwana przez kod JavaScript. Jest specjalne określenie na takie asynchroniczne przesyłanie zapytań i odpowiedzi bez konieczności zmieniania ani odświeżania strony: **AJAX** (ang. *Asynchronous JavaScript and XML* — asynchroniczne języki JavaScript i XML). Kilka lat temu było to hasło, którego wszyscy programiści używali na określenie szczególnego rodzaju aplikacji, nieustannie wysyłających zapytania o dane bez przeładowywania stron. Dzisiaj tego rodzaju aplikacje (Twitter, Facebook, Google Maps, Gmail) są oczywistością.

Obiekt w języku JavaScript odpowiedzialny za wysyłanie zapytań HTTP i odbieranie odpowiedzi nosi dziwną nazwę `XMLHttpRequest` i służy do wykonywania kilku ważnych operacji:

1. wysyłania zapytań do serwera,
2. sprawdzania stanów zapytań,
3. odbierania i przetwarzania odpowiedzi na zapytania,
4. nasłuchiwanie zdarzenia `readystatechange`, umożliwiającego reagowanie na zmiany stanów zapytań.

Oprócz tego obiekt ten realizuje kilka innych funkcji, które na razie nie są istotne.

Dlaczego nie skorzystać z zewnętrznych bibliotek?

Istnieje wiele zewnętrznych bibliotek opakowujących obiekt `XMLHttpRequest` i ułatwiających korzystanie z niego. Możesz ich używać, jeżeli chcesz, ale bezpośrednie korzystanie z obiektu `XMLHttpRequest` nie jest trudne. Wymaga wpisania kilku wierszy kodu, który w porównaniu ze wszystkim, czego dowiedziałeś się już o bibliotece React, jest bardzo prosty.

Czas na React!

Gdy wiesz już, jak działają protokół HTTP i obiekt `XMLHttpRequest`, czas, abyś zajął się biblioteką React. Muszę Cię jednak ostrzec, że biblioteka ta niewiele wnosi w kwestii przetwarzania zewnętrznych danych. Działa ona głównie w warstwie prezentacji (oznaczonej literą V w skrócie MVC). Ty będziesz tworzył wewnątrz komponentu zwykły, nudny kod JavaScript, którego głównym celem będzie przetwarzanie odpowiedzi na wysyłane zapytania HTTP. Więcej na temat tego podejścia dowiesz się za chwilę, natomiast teraz zajmijmy się praktycznym przykładem.

Pierwsze kroki

Pierwszym krokiem jest utworzenie aplikacji React. W wierszu poleceń przejdź do folderu, w którym zamierzasz utworzyć nowy projekt, na przykład *ipaddress*, i wpisz następujące polecenie:

```
create-react-app ipaddress
```

Naciśnij klawisz *Enter*, aby uruchomić polecenie. Po chwili zostanie utworzony nowy projekt React. Ponieważ będziesz zaczynał od zera, musisz usunąć trochę plików. Najpierw wyczyść zawartość folderów *public* i *src*. Nie obawiaj się, za chwilę umieścisz w nich nową treść, zaczynając od pliku HTML.

W folderze *public* utwórz nowy plik o nazwie *index.html* i wpisz w nim następujący kod:

```
<!DOCTYPE html>
<html>

<head>
  <title>Adres IP</title>
</head>

<body>
  <div id="container">

  </div>
</body>

</html>
```

Plik ten zawiera tylko jeden ważny element *div* o nazwie *container*.

Przejdź teraz do folderu *src* i utwórz w nim plik *index.js* o następującej zawartości:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import IPAddressContainer from "./IPAddressContainer";

var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <IPAddressContainer/>
  </div>,
  destination
);
```

Skrypt ten stanowi punkt wejścia do aplikacji. Importowane są w nim biblioteki React i ReactDOM oraz nieistniejące jeszcze arkusz CSS i komponent IPAddressContainer. Wywoływana jest w nim również metoda ReactDOM.render(), zapisująca zadaną treść w zdefiniowanym przed chwilą elemencie *div* w pliku HTML.

Zanim zajmiesz się naprawdę ciekawymi rzeczami, musisz zrobić jedną rzecz. W folderze *src* utwórz plik o nazwie *index.css* i umieść w nim poniższą regułę stylu:

```
body {
  background-color: #FFCC00;
}
```

Zapisz wszystkie utworzone pliki. Utworzyleś załączek aplikacji. W następnej części rozdziału sprawisz, że będzie ona naprawdę użyteczna (a przynajmniej będzie sprawiała takie wrażenie).

Uzyskanie adresu IP

Następnym krokiem jest utworzenie komponentu, którego zadaniem będzie uzyskiwanie adresu IP z usługi WWW, zapisywanie go w obiekcie stanu i udostępnianie jako właściwości innym komponentom, które go zażąдают. W tym celu w folderze *src* utwórz plik o nazwie *IPAddressContainer.js* i wpisz w nim poniższy kod:

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

class IPAddressContainer extends Component {
  render() {
    return (
      <p>Jeszcze nic nie ma!</p>
    );
  }
}

export default IPAddressContainer;
```

Kod ten niewiele robi. Po prostu wyświetla na stronie napis „Jeszcze nic nie ma!”. Na razie nam to nie przeszkadza. Zmodyfikuj teraz kod wysyłający zapytanie HTTP w następujący sposób:

```
var xhr;

class IPAddressContainer extends Component {
  constructor(props) {
    super(props);
    this.state = {
      ip_address: "..."
    };

    this.processRequest = this.processRequest.bind(this);
  }

  componentDidMount() {
    xhr = new XMLHttpRequest();
    xhr.open("GET", "https://ipinfo.io/json", true);
    xhr.send();

    xhr.addEventListener("readystatechange", this.processRequest, false);
  }

  processRequest() {
    if (xhr.readyState === 4 && xhr.status === 200) {
      var response = JSON.parse(xhr.responseText);

      this.setState({
        ip_address: response.ip
      });
    }
  }

  render() {
    return (

```

```

        <div>Nothing yet!</div>
    );
}
};

```

Teraz coś się tu dzieje! Gdy komponent zostanie uaktywniony i wywołana jego metoda cyklu życia `componentDidMount()`, wtedy zostanie wysłane zapytanie HTTP do usługi `ipinfo.io`:

```

...
componentDidMount() {
  xhr = new XMLHttpRequest();
  xhr.open('GET', "https://ipinfo.io/json", true);
  xhr.send();

  xhr.addEventListener("readystatechange", this.processRequest, false);
}
...

```

Gdy nadaje się odpowiedź z usługi `ipinfo.io`, zostanie wywołana funkcja `processRequest()` przetwarzająca dane:

```

...
processRequest() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var response = JSON.parse(xhr.responseText);

    this.setState({
      ip_address: response.ip
    });
}
...

```

Zmień teraz metodę `render()` tak, aby odwoływała się do adresu IP zapisanego w obiekcie stanu:

```

var xhr;

class IPAddressContainer extends Component {
  constructor(props) {
    super(props);

    this.state = {
      ip_address: "..."
    };

    this.processRequest = this.processRequest.bind(this);
  }

  componentDidMount() {
    xhr = new XMLHttpRequest();
    xhr.open("GET", "https://ipinfo.io/json", true);
    xhr.send();

    xhr.addEventListener("readystatechange", this.processRequest, false);
  }
}

```

```

processRequest() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var response = JSON.parse(xhr.responseText);
    this.setState({
      ip_address: response.ip
    });
}

```

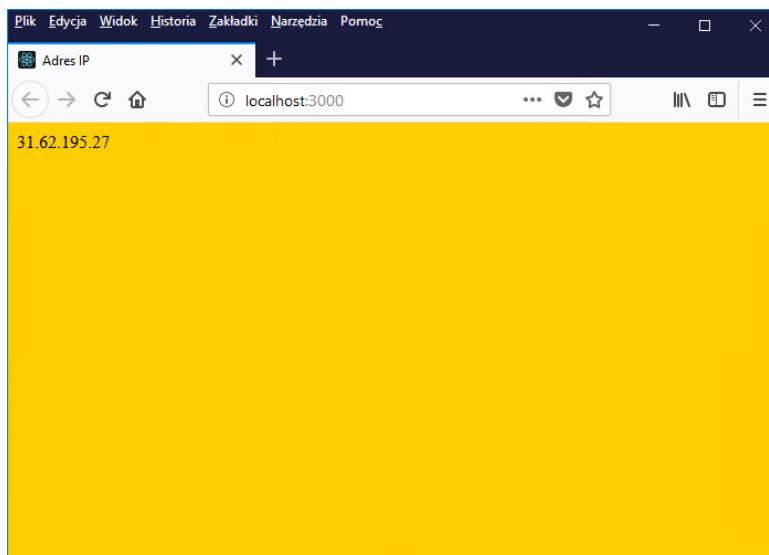
```

        }

      render() {
        return (
          <div>{this.state.ip_address}</div>
        );
      }
    }
  }
}

```

Gdy teraz uruchomisz aplikację, powinieneś zobaczyć stronę z adresem IP. Dla przypomnienia: aby uruchomić aplikację przejdź w oknie wiersza poleceń do folderu *ipaddress* i wpisz polecenie `npm start`. Gdy otworzy się przeglądarka, powinieneś uzyskać efekt podobny do poniższego:



Aplikacja nie wygląda imponująco, ale naprawimy to w następnej części rozdziału.

Upiększenie aplikacji

Najtrudniejsze masz już za sobą! Utworzyleś komponent, który wykonuje wszystkie skomplikowane operacje związane z obsługą zapytań HTTP i uzyskiwaniem adresu IP. Teraz musisz zadbać nieco o wygląd aplikacji, aby nie była taka skromna.

Nie dodawaj w metodzie `render()` komponentu `IPAddressContainer` nowych elementów HTML ani kodu formatującego. Zamiast tego utwórz nowy komponent, który zajmie się wyglądem aplikacji. W tym celu w folderze *src* utwórz nowy plik o nazwie *IPAddress.js* i umieść w nim poniższy kod:

```

import React, { Component } from "react";

class IPAddress extends Component {
  render() {
    return (
      <div>

```

```

        Hej!
      </div>
    );
}
}

export default IPAddress;

```

Kod ten definiuje komponent o nazwie `IPAddress`, którego zadaniem jest wyświetlenie dodatkowego tekstu i nadanie adresowi IP dokładnie takiego formatu, jaki chcemy. Na razie kod niewiele robi, ale zaraz to zmienisz.

Najpierw zmodyfikuj metodę `render()` w następujący sposób:

```

class IPAddress extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.ip}</h1>
        <p>(To jest Twój adres IP... chyba :P).</p>
      </div>
    );
  }
}

export default IPAddress;

```

Wyróżnione zmiany powinny być dla Ciebie jasne. Wewnątrz elementu `h1` umieszczana jest wartość właściwości `ip`, a wewnątrz elementu `p` jest wyświetlany dodatkowy tekst. Dzięki temu element HTML jest nie tylko bardziej zrozumiały, lecz także można mu nadać ładniejszy styl.

Utwórz teraz w folderze `src` nowy arkusz CSS o nazwie `IPAddress.css` z następującymi regułami:

```

h1 {
  font-family: sans-serif;
  text-align: center;
  padding-top: 140px;
  font-size: 60px;
  margin: -15px;
}

p {
  font-family: sans-serif;
  color: #907400;
  text-align: center;
}

```

Następnie umieść w pliku `IPAddress.js` odwołanie do utworzonego arkusza CSS. W tym celu wpisz wyróżniony niżej wiersz:

```

import React, { Component } from "react";
import "./IPAddress.css";

class IPAddress extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.ip}</h1>
        <p>(To jest Twój adres IP... chyba :P).</p>
      </div>
    );
  }
}

export default IPAddress;

```

```

        }
    }
export default IPAddress;

```

Pozostało jedynie wykorzystać komponent IPAddress i przekazać mu adres IP. Najpierw musisz „powiadomić” komponent IPAddressContainer o istnieniu komponentu IPAddress, umieszczając w kodzie odpowiednią referencję. Na początku pliku *IPAddressContainer.js* wpisz wyróżniony niżej wiersz:

```

import React, { Component } from "react";
import IPAddress from "./IPAddress";
...

```

Drugą (i ostatnią) zmianą jest modyfikacja metody render():

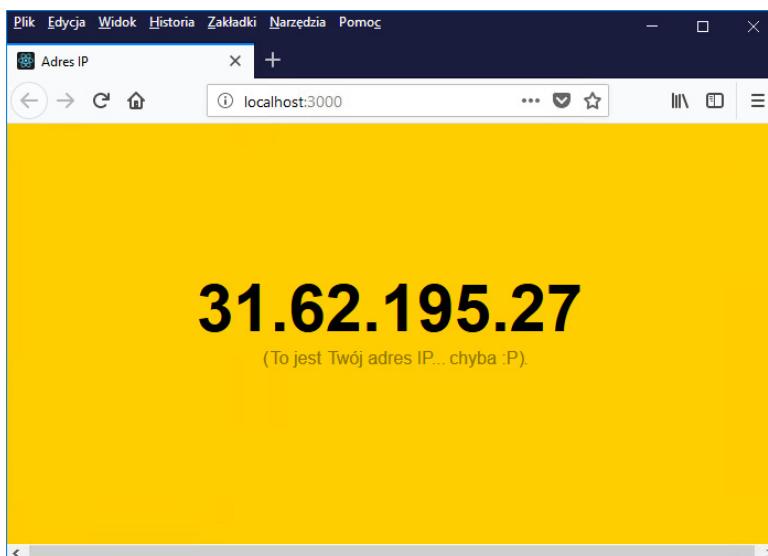
```

class IPAddressContainer extends Component {
...
render() {
    return (
        <IPAddress ip={this.state.ip_address}/>
    );
}
}

```

W wyróżnionym wierszu wywoływany jest komponent IPAddress i definiowana właściwość ip, której jest przypisywana wartość zmiennej stanu ip_address. W ten sposób uzyskany adres IP wraca do metody render() komponentu IPAddress, gdzie jest odpowiednio formatowany i wyświetlany.

Jeżeli otworzysz teraz aplikację w przeglądarce, zobaczysz stronę niemal identyczną z tą przedstawioną na początku rozdziału:



W tym momencie zakończyłeś pracę nad aplikacją i prawie skończyłeś rozdział. Musisz dowiedzieć się jeszcze jednej rzeczy o niezwykłych komponentach, które utworzyłeś.

Komponenty prezentacyjne i kontenerowe

Biorąc po uwagę opisane w tym rozdziale zagadnienia, teraz jest dobry moment, aby przedstawić metodykę programowania, którą mimochodem zastosowaliśmy i będziemy ją wykorzystywać w następnych rozdziałach. W aplikacji zostały wykorzystane dwa rodzaje komponentów:

1. **Komponenty odpowiedzialne za wygląd aplikacji.** Są to tzw. **komponenty prezentacyjne**.
2. **Komponenty wykonujące operacje w tle aplikacji.** Operacje te obejmują zmienianie stron, zwiększenie licznika, wysyłanie zapytań HTTP o dane itp. Są to tzw. **komponenty kontenerowe**.

Dzieląc komponenty na te, które wyświetlają informacje (komponenty prezentacyjne), i te, które wysyłają dane do innych komponentów (komponenty kontenerowe), można łatwiej zarządzać aplikacją wykorzystującą bibliotekę React. Wyczerpujący opis, jak stosować komponenty obu typów, znajdziesz w artykule Dana Abramova pod adresem https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0.

Podsumowanie

W tym momencie prawdopodobnie zastanawiasz się, co nowego wniosła tu biblioteka React. W rzeczywistości wykorzystałeś w komponentach stary, dobry interfejs JavaScript API do obsłużenia kilku zdarzeń i wykonania na stanach i właściwościach komponentu kilku operacji, które wielokrotnie kodowałeś wcześniej. Rzecz jednak w tym, że dowiedziałeś się niemal wszystkiego o podstawach biblioteki React. Teraz już nic nie powinno Cię zaskoczyć. Jedyną nowością będzie inne zastosowanie elementów już Ci znanych, które będą prezentowane w nowej formie i w nowych, ciekawszych sytuacjach. Na tym przecież polega programowanie, prawda?

Jeżeli napotkasz problemy, pytaj!

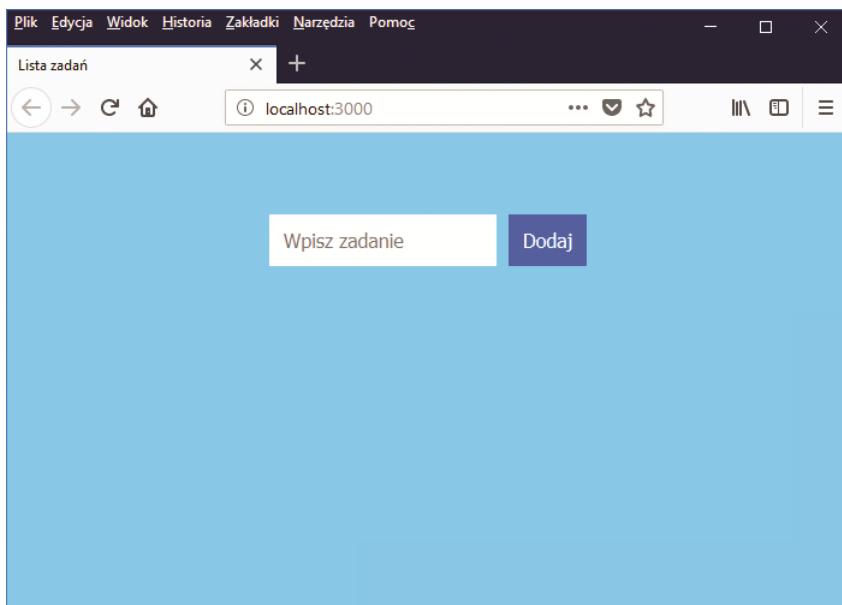
Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

15

Niebanalna lista zadań

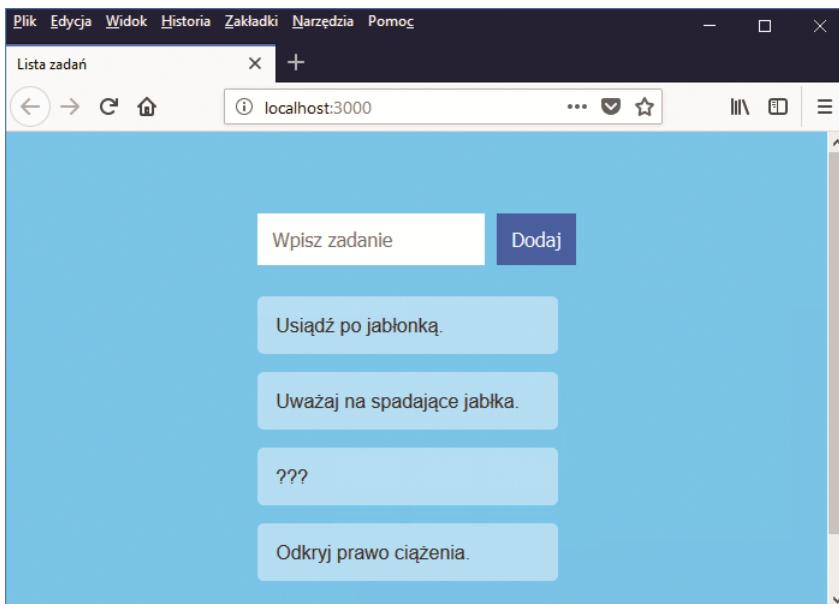
Jeżeli utworzenie aplikacji *Witaj, świecie!* można porównać do zapoznania się z biblioteką React, to aplikacja z prawdziwego zdarzenia, *Lista zadań*, będzie dla Ciebie dowodem osiągnięcia mistrzostwa. W tym rozdziale wykorzystasz wiele technik i elementów, które do tej pory poznaleś.

Na początku aplikacja będzie pusta, gotowa to wpisania zadań do wykonania (patrz rysunek 15.1).



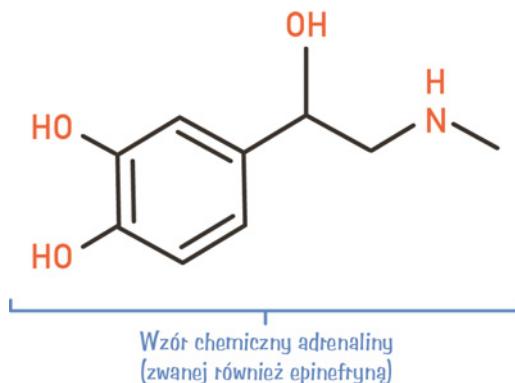
Rysunek 15.1. Pusta lista zadań w aplikacji

Aplikacja jest bardzo prosta. W polu tekstowym wpisuje się zadanie, temat lub dowolną inną treść i kliką przycisk *Dodaj* (lub naciska klawisz *Enter*). Wpisany tekst pojawia się na liście. W ten sposób można dodawać i wyświetlać kolejne zadania (patrz rysunek 15.2).



Rysunek 15.2. Aplikacja pozwala na dodawanie i wyświetlanie zadań

Aby usunąć zadanie, wystarczy je po prostu kliknąć. Proste, prawda? W następnych częściach rozdziału wspólnie utworzymy aplikację od podstaw, wykorzystując wiele poznanych do tej pory adrenalinogennych technik.



Tworzenie poszczególnych komponentów aplikacji i uczenie się (na niesamowitym poziomie szczegółowości), jak współpracują one ze sobą, na pewno będzie ciekawym doświadczeniem. Zaczynajmy!

Pierwsze kroki

Pierwszym krokiem jest utworzenie projektu aplikacji React w sposób opisany w rozdziale 13. „Konfiguracja środowiska React bez stresu”. W tym celu w wierszu poleceń przejdź do folderu, w którym zamierzasz utworzyć projekt, i wpisz następujące polecenie:

```
create-react-app todolist
```

Naciśnij *Enter*, aby uruchomić polecenie. Po chwili zostanie utworzony nowy projekt. Ponieważ aplikację będziemy budować od postaw, usuń całą zawartość folderów *public* i *src*.

Teraz już wiesz, co musisz zrobić. Potrzebny jest punkt startowy aplikacji. Utwórz zatem w folderze *public* plik *index.html* i umieść w nim następujący kod HTML:

```
<!DOCTYPE html>
<html>
<head>
  <title>Lista zadań</title>
</head>
<body>
  <div id="container">
  </div>
</body>
</html>
```

Jak widzisz, strona jest bardzo prosta. Prawdziwa magia będzie się działała w folderze *src*, w którym umieścisz pliki JavaScript i CSS. Teraz w folderze *src* utwórz plik *index.css* z następującymi regułami stylów:

```
body {
  padding: 50px;
  background-color: #66CCFF;
  font-family: sans-serif;
}
#container {
  display: flex;
  justify-content: center;
}
```

Następnie napisz kod JavaScript, który wzbogaci stronę startową. W folderze *src* utwórz plik *index.js* i wpisz w nim poniższy kod:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";

var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <p>Cześć!</p>
  </div>,
  destination
);
```

Przyjrzyj się przez chwilę kodowi, który właśnie wpisałeś. Teraz powinieneś już doskonale wiedzieć, co się dzieje w plikach HTML, CSS i JavaScript. W tej chwili masz fundament aplikacji. W następnych częściach rozdziału oprzesz na nim wszystkie pliki, które stworzą aplikację *Lista zadań*.

Utworzenie początkowego interfejsu użytkownika

Na razie aplikacja niewiele robi. I wygląda niewiele lepiej niż wcześniej. Jej funkcjonalnościami zajmiemy się za chwilę, ale najpierw przygotujmy potrzebne elementy interfejsu użytkownika. Nie będzie to skomplikowane. Najpierw utwórz pole tekstowe i przycisk. Do tego celu wystarczy użyć elementów `div`, `form`, `input` i `button`.

Wszystkie te elementy umieść w komponente o nazwie `TodoList`. Utwórz więc w folderze `src` plik o nazwie `TodoList.js` i umieść w nim następujący treść:

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

class TodoList extends Component {
  render() {
    return (
      <div className="todoListMain">
        <div className="header">
          <form>
            <input placeholder="Wpisz zadanie">
            </input>
            <button type="submit">Dodaj</button>
          </form>
        </div>
      </div>
    );
  }
}

export default TodoList;
```

Poświeć chwilę na przejrzenie pliku. Zawiera on sporo kodu JSX definiującego elementy interfejsu. Aby wykorzystać nowy komponent i sprawdzić, jak wygląda strona, musisz w pliku `index.js` wpisać odwołanie do komponentu. Wprowadź więc w kodzie wyróżnione niżej zmiany:

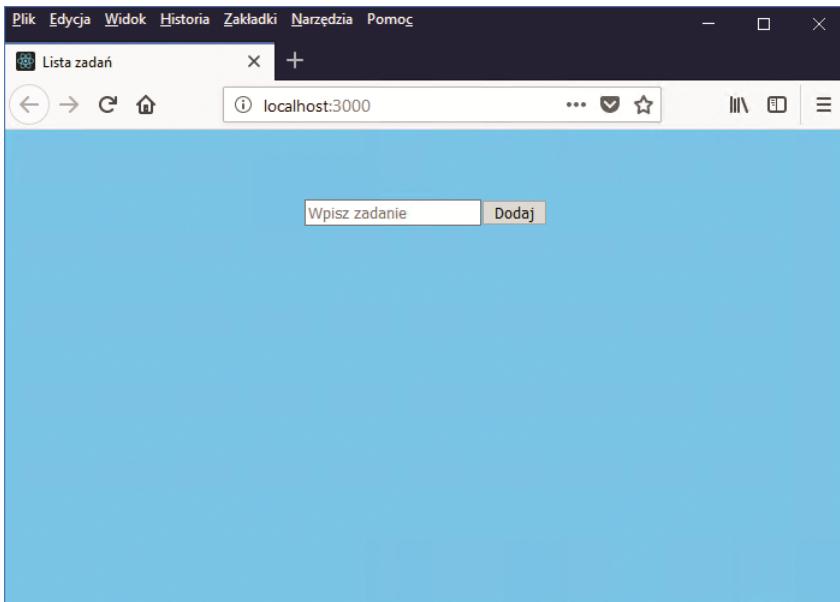
```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import TodoList from "./TodoList";

var destination = document.querySelector("#container");

ReactDOM.render(
  <div>
    <TodoList/>
  </div>,
  destination
);
```

Zapisz zmiany i otwórz aplikację. Jeżeli wszystko zrobiłeś poprawnie, zobaczysz widok jak na rysunku 15.3.

Pojawiło się pole tekstowe i przycisk. Te dwa elementy na razie ani nie działają, ani należycie nie wyglądają. Zaraz to naprawimy, ale wcześniej zastanówmy się, jakie funkcjonalności trzeba zaimplementować w aplikacji.



Rysunek 15.3. Tak na razie wygląda Twoja aplikacja

Utworzenie pozostałej części aplikacji

Jak się zapewne domyślasz, wyświetlenie elementów interfejsu użytkownika jest prostą czynnością. Prawdziwym wyzwaniem jest powiązanie ich z danymi. Pracę można podzielić na pięć etapów:

1. implementacja dodawania zadań,
2. implementacja wyświetlania zadań,
3. stylizacja aplikacji,
4. implementacja usuwania zadań,
5. implementacja animowanego dodawania i usuwania zadań.

Szczegóły każdego z etapów można łatwo zaplanować w głowie. Podczas zbierania ich wszystkich razem musisz pamiętać o kilku rzeczach, opisanych w następnych częściach rozdziału.

Dodawanie zadań

Pierwszym poważnym wyzwaniem, z którym musisz się zmierzyć, jest utworzenie procedury obsługi zdarzenia i domyślnej procedury obsługi formularza umożliwiającej dodawanie do niego zadań. Wprowadź teraz w elemencie form wyróżnioną niżej zmianę:

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

class TodoList extends Component {
  render() {
    return (
      <div>
        <input type="text" value="Zadanie" />
        <button>Dodaj</button>
      </div>
    );
  }
}
```

```

<div className="todoListMain">
  <div className="header">
    <form onSubmit={this.addItem}>
      <input placeholder="Wpisz zadanie">
      </input>
      <button type="submit">Dodaj</button>
    </form>
  </div>
</div>
);
}
}

export default TodoList;

```

Teraz formularz nasłuchuje zdarzenia submit i w chwili pojawienia się go wywołuje metodę addItem(). Zwróć uwagę, że formularz nie nasłuchuje zdarzenia zgłoszanego przez przycisk. Jest tak dlatego, że przycisk ma atrybut type o wartości submit. Na tym polega jedna z osobliwości kodu HTML, że kliknięcie przycisku typu submit jest równoważne z wysłaniem zdarzenia submit do formularza.

Teraz pora utworzyć metodę addItem(), która będzie wywoływana po kliknięciu przycisku. Wpisz powyżej metody render() wyróżnione niżej wiersze:

```

class TodoList extends Component {
  constructor(props) {
    super(props);
    this.addItem = this.addItem.bind(this);
  }

  addItem(e) {
  }
...
}

```

Teraz pozostało jedynie napisać kod metody addItem() i sprawdzić, czy jest ona poprawnie wywoływana. Wciąż jeszcze nie wpisałeś właściwego kodu dodającego zadania do listy, zatem w konstruktorze komponentu zdefiniuj obiekt state, jak niżej:

```

constructor(props) {
  super(props);
  this.state = {
    items: []
  };

  this.addItem = this.addItem.bind(this);
}

```

Obiekt state nie jest skomplikowany. Jego jedyna właściwość items jest tablicą, w której będą przechowywane zadania wpisywane przez użytkownika. Teraz musisz jeszcze napisać kod odczytujący zawartość pola tekstowego i zapisujący ją w tablicy items, gdy użytkownik kliknie przycisk. Jedyną trudnością jest tu odczytywanie zawartości elementu modelu DOM. Jak już wiesz, pomiędzy modelem a biblioteką React istnieje zapora uniemożliwiająca bezpośrednie odwoływanie się do elementów i wykonywanie na nich operacji. Można jednak stosować obejście w postaci referencji.

W metodzie render() wprowadź następującą zmianę:

```
render() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={(a) => this._InputElement = a}
            placeholder="Wpisz zadanie">
          </input>
          <button type="submit">Dodaj</button>
        </form>
      </div>
    </div>
  );
}
```

W ten sposób referencja do elementu `input` jest zapisywana we właściwości o adekwatnej nazwie `_InputElement`. Dzięki tej właściwości będzie można w dowolnym miejscu kodu komponentu odwoływać się do pola tekstowego. Teraz pora na umieszczenie w metodzie `addItem()` następującego kodu:

```
addItem(e) {
  var itemArray = this.state.items;

  if (this._InputElement.value !== "") {
    itemArray.unshift({
      text: this._InputElement.value,
      key: Date.now()
    });

    this.setState({
      items: itemArray
    });

    this._InputElement.value = "";
  }

  console.log(itemArray);

  e.preventDefault();
}
```

Przyjrzyj się przez chwilę temu kodowi. Tworzona jest w nim zmienna `itemArray`, w której jest zapisywana zawartość właściwości `items` obiektu `state`. Następnie kod sprawdza, czy pole tekstowe zawiera treść. Jeżeli pole jest puste, wtedy nic się nie dzieje. W przeciwnym wypadku zawartość pola jest dodawana do tablicy `itemArray`:

```
itemArray.unshift({
  text: this._InputElement.value,
  key: Date.now()
});
```

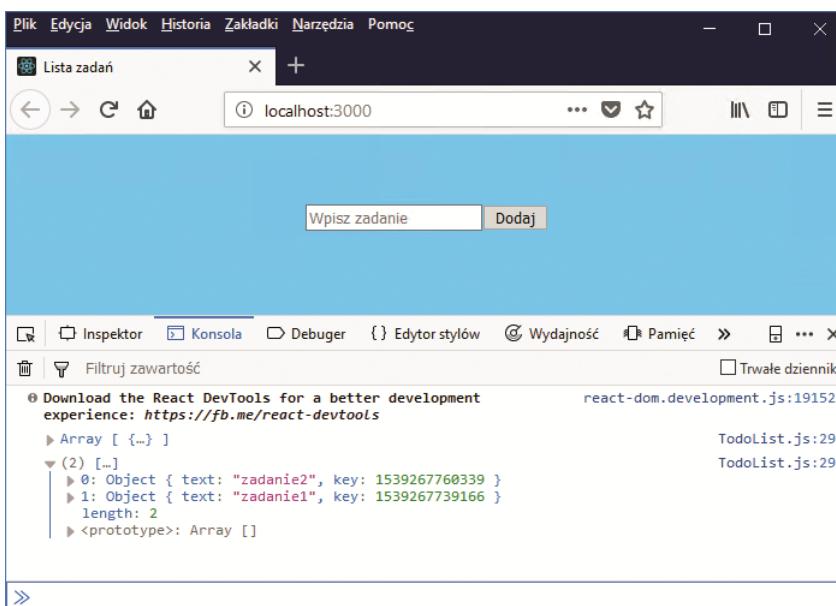
W rzeczywistości do tablicy nie jest dodawany sam tekst, ale obiekt zawierający tekst i unikatowy klucz utworzony na podstawie bieżącego czasu (metoda `Date.now()`). Prawdopodobnie nie wiesz, po co jest ten klucz. To zrozumiałe, ale zaraz wszystko będzie jasne.

Pozostała część kodu jest zwyczajnie nudna. Właściwości items obiektu state jest przypisywana zawartość tablicy itemArray. Usuwana jest zawartość pola tekstowego, aby zrobić miejsce na następne zadanie. Mniej nudny jest następujący wiersz:

```
e.preventDefault();
```

Zmieniana jest tu domyślna procedura obsługi zdarzenia. Ta operacja jest związana ze sposobem, w jaki działa formularz. Domyślnie po wysłaniu formularza strona jest ponownie ładowana i usuwane są z niej wprowadzone dane. Zdecydowanie nie jest to pożądany efekt. Wywołując metodę preventDefault(), blokuje się domyślną obsługę formularza. I bardzo dobrze!

Czas sprawdzić poczynione postępy. Gdy uruchomisz aplikację i otworzysz konsolę przeglądarki, przekonasz się, że w obiekcie state są poprawnie zapisywane dodawane na stronie zadania (patrz rysunek 15.4).



Rysunek 15.4. Widok zapisywanych zadań (wiem, że nie jest imponujący, ale potwierdza poczynione przez Ciebie postępy, naprawdę!)

Na stronie <http://bit.ly/setStateConcat> opisany jest alternatywny sposób określania wewnętrz metodę addNewItem() nowego stanu komponentu, który nie wymaga modyfikowania istniejącego stanu.

Wyświetlanie zadań

Wyświetlanie listy zadań w konsoli przeglądarki nie zrobi większego wrażenia na użytkownikach Twojej aplikacji. Jestem pewien, że woleliby widzieć zadania wyświetcone bezpośrednio na stronie. Dlatego utwórz nowy komponent o nazwie TodoItems i użyj go w metodzie render() komponentu TodoList. Dodatkowo we właściwości nowego komponentu umieść zawartość tablicy items.

Powyższe operacje można przełożyć na wyróżniony niżej wiersz:

```
render() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={(a) => this._InputElement = a}
            placeholder="Wpisz zadanie">
          </input>
          <button type="submit">Dodaj</button>
        </form>
      </div>
      <TodoItems entries={this.state.items}>
    </div>
  );
}
```

Następnie na początku pliku wpisz wyróżnioną niżej instrukcję import:

```
import React, { Component } from "react";
import TodoItems from "./TodoItems";

class TodoList extends Component {
  ...
}
```

Powyższe dwie zmiany implementują operacje, które ma wykonywać kod w pliku *TodoList.js*.

Teraz przygotuj komponent *TodoItems*. W tym celu w folderze *src* utwórz plik o nazwie *TodoItems.js* i wpisz w nim następującą treść:

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

class TodoItems extends Component {
  constructor(props) {
    super(props);

    this.createTasks = this.createTasks.bind(this);
  }

  createTasks(item) {
    return <li key={item.key}>{item.text}</li>
  }

  render() {
    var todoEntries = this.props.entries;
    var listItems = todoEntries.map(this.createTasks);

    return (
      <ul className="theList">
        {listItems}
      </ul>
    );
  }
};

export default TodoItems;
```

Zapewne uważasz, że tego dodanego za jednym razem kodu jest bardzo dużo, ale przyjrzyj się uważnie, co on dokładnie zawiera. Metoda *render()* odczytuje listę zadań (przekazaną w postaci

właściwości `entries`) i zamienia ją na kod JSX/HTML. Wykorzystuje w tym celu metodę `map()` i niżej przedstawioną metodę `createTasks()`:

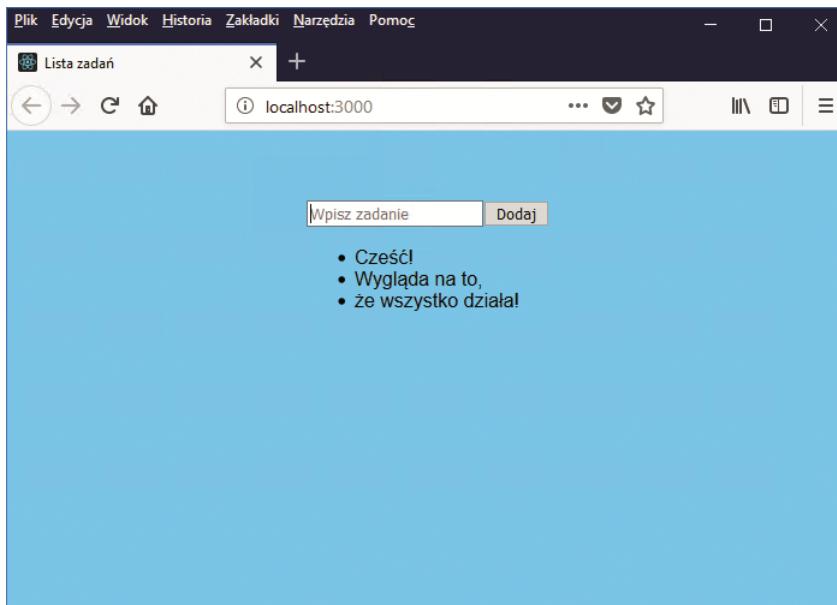
```
createTasks(item) {
  return <li key={item.key}>{item.text}</li>
}
```

W zmiennej `listItems` zapisywana jest tablica elementów listy `li`. Tablica ta zawiera treść przeznaczoną do wyświetlenia. Zwróć uwagę, że wykorzystywany jest również atrybut `key`, któremu — jak pamiętasz — jest przypisywany wynik metody `Date.now()`. Dzięki temu biblioteka React będzie mogła identyfikować poszczególne elementy listy.

Następnie za pomocą poniższego kodu lista elementów jest zamieniana na coś, co można wyświetlić na stronie:

```
return (
  <ul className="theList">
    {listItems}
  </ul>
);
```

Zapisz wprowadzone zmiany i sprawdź, jak teraz działa aplikacja (jeżeli nie działa, użyj polecenia `npm start`). Jeśli poprawnie wpisałeś kod, nie tylko będziesz mógł dodawać zadania, lecz także będziesz je widział na stronie (patrz rysunek 15.5).



Rysunek 15.5. Teraz zadania są widoczne na stronie!

Jeżeli uzyskałeś widok taki jak na powyższym rysunku, to świetnie! Aby uczcić ten sukces, zróbcmy małą przerwę w tworzeniu kodu JavaScript i JSX.

Stylizacja aplikacji

Obecnie niezwykłe funkcjonalności aplikacji nie korespondują z jej wyglądem. Możesz to łatwo naprawić, dodając jeden arkusz stylów z odpowiednimi regułami. Utwórz więc w folderze *src* plik *TodoList.css* i wpisz w nim poniższe reguły:

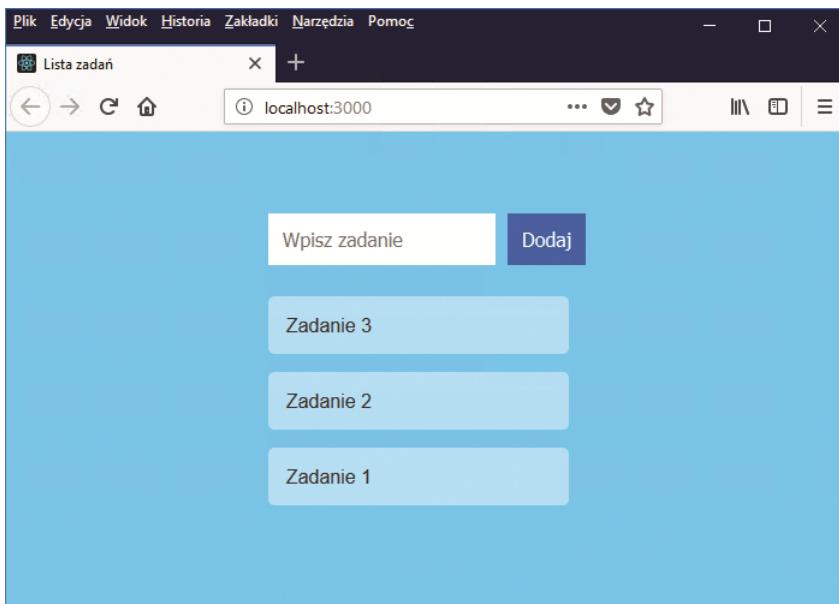
```
.todoListMain .header input {  
    padding: 10px;  
    font-size: 16px;  
    border: 2px solid #FFF;  
    width: 165px;  
}  
.todoListMain .header button {  
    padding: 10px;  
    font-size: 16px;  
    margin: 10px;  
    margin-right: 0px;  
    background-color: #0066FF;  
    color: #FFF;  
    border: 2px solid #0066FF;  
}  
.todoListMain .header button:hover {  
    background-color: #003399;  
    border: 2px solid #003399;  
    cursor: pointer;  
}  
.todoListMain .theList {  
    list-style: none;  
    padding-left: 0;  
    width: 250px;  
}  
.todoListMain .theList li {  
    color: #333;  
    background-color: rgba(255,255,255,.5);  
    padding: 15px;  
    margin-bottom: 15px;  
    border-radius: 5px;  
}
```

Po utworzeniu arkusza musisz odwołanie do niego dodać na początku pliku *TodoList.js*:

```
import React, { Component } from "react";  
import TodoItems from "./TodoItems";  
import "./TodoList.css";  
  
class TodoList extends Component {  
    ...
```

Kiedy wprowadzisz powyższe zmiany, Twoja aplikacja powinna wyglądać tak jak na rysunku 15.6.

Jak sam widzisz, aplikacja wygląda teraz o wiele lepiej. Wystarczyło jedynie zdefiniować kilka reguł CSS. Żadne funkcjonalności nie zostały zmienione. Rozbudujemy je w następnym kroku.



Rysunek 15.6. Aplikacja zaczyna wyglądać coraz lepiej

Usuwanie zadań

W tym momencie aplikacja umożliwia dodawanie i wyświetlanie zadań. Nie można jednak ich usuwać. Teraz trzeba sprawić, aby użytkownik mógł usuwać zadania, klikając w nie. Implementacja tej funkcjonalności wydaje się prosta, prawda? Jedyną rzeczą, nad którą trzeba się zastanowić, jest umieszczenie kodu w odpowiednim miejscu. Elementy, które trzeba kliknąć, są zdefiniowane w pliku *TodoItems.js*, natomiast kod umieszczający w nich treść znajduje się w pliku *TodoList.js*. Abyś miał wyobrażenie tego, co Cię czeka, zajmijmy się zawiłościami związanymi z przekazywaniem danych pomiędzy tymi dwoma komponentami.

Najpierw trzeba zdefiniować procedurę obsługi zdarzenia `click` (kliknięcie myszą). W tym celu zmień instrukcję `return` w metodzie `createTasks()` w następujący sposób:

```
createTasks(item) {
  return <li onClick={() => this.delete(item.key)}
         key={item.key}>{item.text}</li>
}
```

Powyższa metoda po prostu wiąże zdarzenie `click` z obsługującą je metodą `delete()`. Nowością w tym przypadku jest przekazywanie wartości w argumencie metody. Ze względu na zakres widoczności argumentu trzeba użyć funkcji strzałkowej, która umożliwia zarówno definiowanie domyślnej wartości argumentu, jak i umieszczenie w nim zadanego wartości. Jeżeli wydaje Ci się to skomplikowane, pamiętaj, że jest to cecha języka JavaScript, która nie ma nic wspólnego z biblioteką React.

Po wprowadzeniu powyższej zmiany zdefiniuj metodę `delete()` obsługującą zdarzenie. Wprowadź w kodzie wyróżnione niżej zmiany:

```

class TodoItems extends Component {
  constructor(props) {
    super(props);
    this.createTasks = this.createTasks.bind(this);
    this.delete = this.delete.bind(this);
  }
  delete(key) {
    this.props.delete(key);
  }
...

```

Zdefiniowana jest tu metoda `delete()` z argumentem `key` identyfikującym zadanie. Aby była wywoływana, jest w konstruktorze jawnie wiązana z obiektem `this`. Zwróć uwagę, że obecnie metoda `delete()` nie wykonuje żadnych operacji. Wywołuje jedynie inną metodę `delete()` przekazaną komponentowi poprzez właściwość `props`. Za chwilę się nią zajmiemy, ale teraz wróćmy do wcześniejszej wpisanego kodu.

Przyjrzyj się metodzie `render()` w pliku `TodoList.js`. W wierszu wywołującym komponent `TodoItems` zdefiniuj właściwość `delete` i przypisz jej metodę `deleteItem()`:

```

render() {
  return (
    <div className="todoListMain">
      <div className="header">
        <form onSubmit={this.addItem}>
          <input ref={(a) => this._InputElement = a}
            placeholder="Wpisz zadanie">
          </input>
          <button type="submit">Dodaj</button>
        </form>
      </div>
      <TodoItems entries={this.state.items}
        delete={this.deleteItem}/>
    </div>
  );
}

```

Ta zmiana powoduje, że komponent `TodoItems` „wie”, że posiada właściwość o nazwie `delete`. Oznacza to również, że z tą właściwością jest połączona metoda `delete()` zdefiniowana w komponencie `TodoList`. Trzeba jeszcze zdefiniować właściwą metodę `deleteItem()`, która będzie usuwać zadania z listy. W tym celu w komponencie `TodoList` wpisz poniższy kod:

```

deleteItem(key) {
  var filteredItems = this.state.items.filter(function(item) {
    return (item.key !== key);
  });

  this.setState({
    items: filteredItems
  });
}

```

Kod ten możesz umieścić w dowolnym miejscu, ale proponuję, abyś wpisał go poniżej definicji metody `addItem()`. Przyjrzyjmy się, co robi ta metoda. Jej argumentem jest klucz identyfikujący kliknięte zadanie. Jest on za pomocą metody `filter()` porównywany ze wszystkimi kluczami zapisanych zadań:

```
var filteredItems = this.state.items.filter(function(item) {
  return (item.key !== key);
});
```

Działanie tego kodu jest proste. Tworzona jest nowa tablica `filteredItems` zawierająca wszystkie zadania oprócz tego przeznaczonego do usunięcia. Przefiltrowana tablica staje się następnie nową właściwością `items` obiektu `state`:

```
this.setState({
  items: filteredItems
});
```

Następnie aktualizowany jest interfejs użytkownika i usunięte zadanie znika na zawsze. Ostatnią rzeczą, którą trzeba zrobić, jest wpisanie kilku typowych wierszy kodu wymaganych w takich sytuacjach. Wprowadź w konstruktorze wyróżniony niżej wiersz:

```
constructor(props) {
  super(props);

  this.state = {
    items: []
  };

  this.addItem = this.addItem.bind(this);
  this.deleteItem = this.deleteItem.bind(this);
}
```

W ten sposób identyfikator `deleteItem` będzie zawsze oznaczał odpowiednią metodę. Zanim ogłosisz sukces w usuwaniu zadań, zróbmy jeszcze jedną rzecz. Otwórz plik `TodoList.css` i wprowadź w nim następujące zmiany:

```
.todoListMain .theList li {
  color: #333;
  background-color: rgba(255,255,255,.5);
  padding: 15px;
  margin-bottom: 15px;
  border-radius: 5px;
  transition: background-color .2s ease-out;
}

.todoListMain .theList li:hover {
  background-color: pink;
  cursor: pointer;
}
```

Zdefiniowany jest tu wizualny efekt pojawiający się po umieszczeniu kurSORA myszy nad zadaniem przeznaczonym do usunięcia. Na tej zmianie w zasadzie kończy się implementacja funkcjonalności usuwania zadań. Otwórz teraz aplikację i spróbuj dodać, a następnie usunąć kilka zadań. Wszystko powinno działać zgodnie z oczekiwaniami.

Pozostała jeszcze jedna rzecz...

Animacje!

Twoją naprawdę ostatnią czynnością będzie zaimplementowanie animacji, aby dodawanie i usuwanie zadań wyglądało naturalnie. Biblioteka React oferuje wiele mechanizmów do animowania elementów. Możesz wykorzystać tradycyjne rozwiązania, na przykład animacje

lub przejścia CSS, metodę `requestAnimationFrame()`, interfejs Web Animations API lub jedną z popularnych bibliotek animacyjnych. Wszystkie te sposoby są bardzo, naprawdę bardzo zaawansowane.

Jednak wymienione wyżej techniki ujawniają swoje pewne ograniczenia w przypadku animowania tworzonych i usuwanych elementów. Jest tak dlatego, że biblioteka React w całości zarządza cyklem życia elementu w modelu DOM. Niektóre z wykorzystywanych przy tym metod można oczywiście nadpisać i dzięki temu przechwycić moment usunięcia elementu, po czym wywołać własny kod animacyjny. Jednak podejście to jest zbyt skomplikowane i nie będziemy się nim zajmować.

Na szczeźle społeczności użytkowników biblioteki React opracowała kilka prostych bibliotek animacyjnych umożliwiających proste definiowanie animacji towarzyszących tworzeniu i usuwaniu elementów. Jedną z nich jest *Flip Move*. Biblioteka ta oferuje wiele funkcjonalności, między innymi łatwe dodawanie i usuwanie elementów listy.

Aby móc zastosować powyższą bibliotekę, musisz ją najpierw dodać do projektu. W tym celu w wierszu poleceń przejdź do folderu *todolist* i wpisz następujące polecenie:

```
npm i -S react-flip-move
```

Naciśnij klawisz *Enter/Return*, aby w folderze *node_modules* zainstalować niezbędne pliki. To jest cała wymagana konfiguracja. Gdy się zakończy, wpisz na początku pliku *TodoItems.js* następującą instrukcję import:

```
import FlipMove from 'react-flip-move';
```

Teraz musisz skonfigurować komponent *FlipMove*, aby animował elementy listy. W tym celu wprowadź w metodzie `render()` wyróżnione niżej zmiany:

```
render() {
  var todoEntries = this.props.entries;
  var listItems = todoEntries.map(this.createTasks);

  return (
    <ul className="theList">
      <FlipMove duration={250} easing="ease-out">
        {listItems}
      </FlipMove>
    </ul>
  );
}
```

Wprowadzona zmiana polega po prostu na osadzeniu tablicy `listItems` (tuż przed jej wyświetleniem) w komponencie *FlipMove* oraz zdefiniowaniu czasu trwania i rodzaju animacji. Jeżeli teraz otworzysz aplikację zobaczysz, że zadania nie są dodawanie i usuwanie natychmiast, tylko są płynnie animowane.

Komponenty kontrolujące i niekontrolujące

Elementy formularza są interesujące z tego względu, że zawierają pewne dane. Na przykład element `text` zawiera treść, ale elementy rozwijanej listy można zaznaczać. Biblioteka React centralizuje stany elementów w swoim małym świecie, więc „nie lubi” elementów formularza za to, że przetwarzają dane we własny, wewnętrzny sposób. Zgodnie z zaleceniami twórców biblioteki wszystkie dane zawarte w formularzu należy synchronizować z danymi w komponencie za pomocą zdarzeń, na przykład `onChange`. Komponenty umożliwiające przetwarzanie danych w formularzu noszą nazwę **komponentów kontrolujących**.

Jednak synchronizowanie każdego elementu formularza jest kłopotliwe i twórcy biblioteki React dobrze o tym wiedzą. Rozwiążanie tego problemu polega na tym, aby nie robić nic, tzn. pozwolić elementom zarządzać ich własnymi stanami i odwoływać się do nich za pomocą referencji, jeżeli zajdzie taka potrzeba. Ten właśnie sposób zastosowałeś w opisanej tu aplikacji. Komponenty, które zdają się na model DOM w kwestii zarządzania stanami elementów, nazywają się **komponentami niekontrolującymi**.

Podsumowanie

Aplikacja *Lista zadań* jest bardzo prosta, ale tworząc ją od podstaw wykorzystałeś wszystkie ciekawe funkcjonalności oferowane przez bibliotekę React. Co więcej, w aplikacji tej komponenty współpracują ze sobą, ponieważ wykorzystałeś techniki, które poznałeś wcześniej. To bardzo ważne.

A teraz mam do Ciebie proste pytanie: czy wszystko, czego nauczyłeś się w tym rozdziale, jest zrozumiałe? Jeżeli tak, to możesz śmiało obwieścić swym znajomym i rodzinie, że jesteś bliski mistrzostwa w stosowaniu biblioteki React. Jeżeli jednak jakieś zagadnienia pozostają niejasne, zachęcam Cię do ponownego przeczytania odpowiednich rozdziałów.

Jeżeli napotkasz problemy, pyтай!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniemi, pyтай śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

16

Tworzenie wysuwanego menu za pomocą biblioteki React

Wysuwanego menu jest ostatnim krzykiem mody w interfejsie użytkownika. Takie menu mają wszystkie fajne aplikacje, a użytkownicy wręcz nie mogą się bez niego obyć. Jest ono niewidocznym na ekranie elementem, który się wysuwa po kliknięciu innego elementu, na przykład strzałki, trzech kresek lub innej ikony symbolizującej menu.

Aby zobaczyć wysuwane menu w akcji otwórz stronę https://www.kirupa.com/react/examples/slidingmenu_css/index.html. Pojawi się punktowana lista. Gdy klikniesz ikonę u góry strony, płynnie wysunie się żółte menu z odnośnikami. Kliknij dowolny odnośnik, a menu z powrotem (też naprawdę płynnie) się schowa. Dowiesz się teraz, jak można zbudować takie menu z wykorzystaniem biblioteki React.

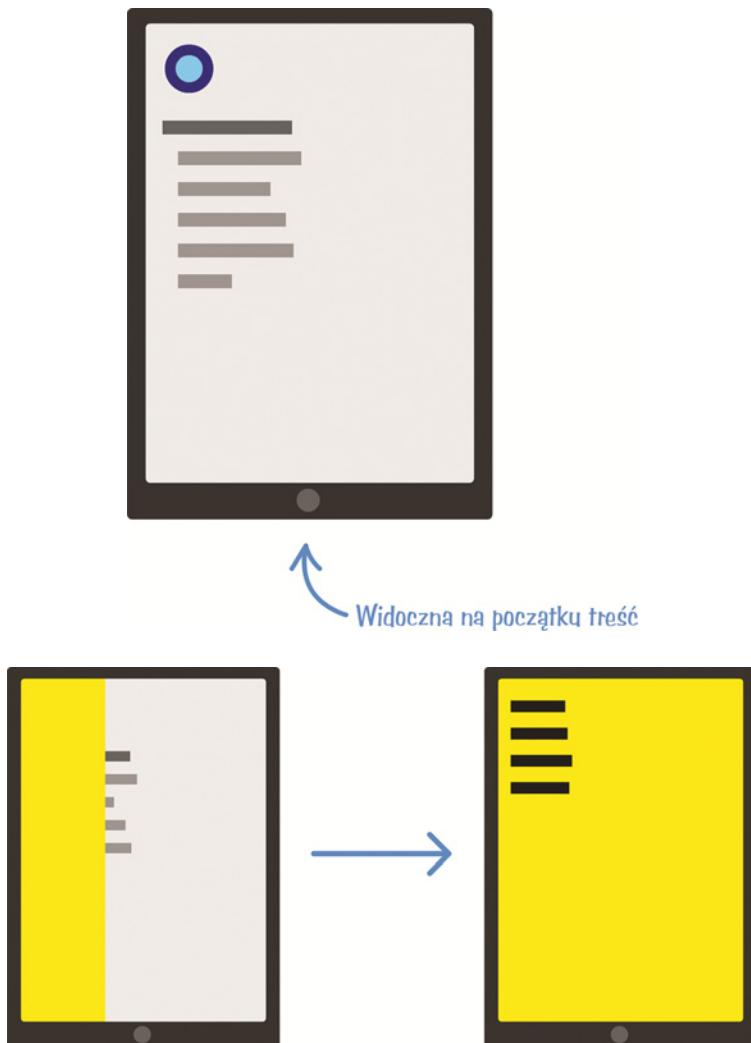
Aplikacje bez biblioteki React

Jeżeli chcesz utworzyć wysuwane menu, wykorzystując czysty kod JavaScript bez stosowania magii biblioteki React, zapoznaj się z artykułem *Creating a Smooth Sliding Menu* („Tworzenie płynnie wysuwanego menu”) na stronie <http://bit.ly/plainSidingMenu>.

Jak działa wysuwane menu?

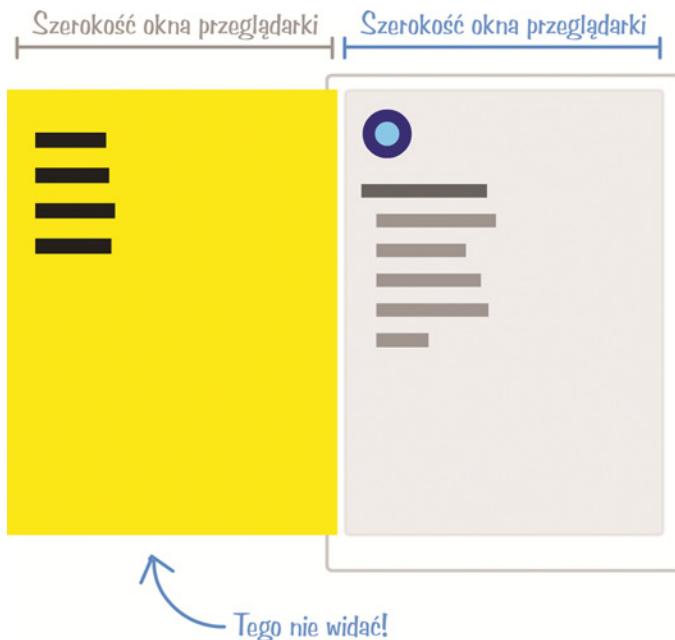
Zanim zajmiemy się kodem, poświęćmy chwilę na zastanowienie się, jak właściwie działa wysuwane menu. Mamy stronę zawierającą jakąś treść — patrz pierwszy rysunek na następnej stronie.

Gdy użytkownik zechce otworzyć menu (na przykład klikając niebieską kropkę w opisany wcześniej przykładzie), w magiczny sposób wysunie się ono z boku — patrz drugi rysunek na następnej stronie.



Działanie wysuwanego menu nie jest tak skomplikowane, jak się wydaje. Tak naprawdę nigdy nie znika ono całkowicie — po prostu jest umieszczane poza granicami strony. Aby to zrozumieć, spójrz na rysunek na następnej stronie.

Po lewej stronie wyświetlonej treści znajduje się menu, które cierpliwie czeka na wywołanie. Ukrycie go polega na przesunięciu do samego końca w lewo tak, aby całkowicie znikło z pola widzenia. Określenie tego, o ile trzeba je przesunąć, jest proste. Menu musi mieć takie same rozmiary jak okno przeglądarki (ang. *viewport*), ponieważ po wysunięciu powinno całkowicie przesłonić aktualnie wyświetląną treść. Zatem trzeba je przesunąć na odległość równą szerokości okna przeglądarki. Można to zrobić, wykorzystując między innymi poniższą regułę CSS:



```
#theMenu {  
  position: fixed;  
  left: 0;  
  top: 0;  
  transform: translate3d(-100vw, 0, 0);  
  
  width: 100vw;  
  height: 100vh;  
}
```

W powyższej regule właściwość `position` (położenie) ma wartość `fixed` (stałe). Dzięki temu prostemu ustawieniu menu nabiera mnóstwa magicznych cech. Przede wszystkim przestają działać inne właściwości określające jego względne położenie na stronie. Można je umieścić w dowolnym miejscu za pomocą współrzędnych `x` i `y` i menu już tego miejsca nie opuści. Jakby tego było mało, menu nie ma wtedy pasków przewijania, nawet jeżeli jego zawartość nie mieści się w oknie.

Powyższe cechy są dla nas bardzo ważne, ponieważ dzięki nim można umieścić menu poza oknem, przypisując jego właściwościom `left` i `top` (lewa i górna współrzędna) wartość 0, a właściwości `transform` wynik funkcji `translate3d()`, której pierwszy argument ma ujemną wartość `-100vw`. W ten sposób menu zostanie przesunięte w lewo na odległość równą szerokości okna przeglądarki. Szerokość menu nie jest wprawdzie bezpośrednio związana z jego położeniem, ale odgrywa tu istotną rolę. Dlatego w powyższej regule CSS właściwościom `width` i `height` (szerokość i wysokość) przypisywane są odpowiednio wartości `100vw` i `100vh`, dzięki którym rozmiary menu są takie same jak okna przeglądarki.

Co oznaczają jednostki vw i vh?

Być może nie spotkałeś się do tej pory z jednostkami vw i vh (ang. *viewport width* i *viewport height*). Służą one do wyrażania odległości w setnych częściach szerokości i wysokości okna przeglądarki. Pod tym względem są bardzo podobne do jednostek procentowych. Na przykład odległość 100vw odpowiada pełnej szerokości okna, a 100vh odpowiada jego pełnej wysokości.

Aby wyświetlić menu, trzeba je przesunąć w prawo tak, aby jego poziome położenie było takie samo jak położenie zawartości okna. Można to łatwo osiągnąć za pomocą właściwości CSS, w sposób podobny do zastosowanego wcześniej. Wystarczy po prostu przypisać właściwości transform wynik funkcji translate3d(), której pierwszym argumentem jest wartość 0vw. Odpowiedni kod wygląda następująco:

```
transform: translate3d(0vw, 0, 0);
```

Powyższa zmiana powoduje, że menu (którego początkowa pozioma współrzędna ma wartość -100vw) wysuwa się spoza widoku i pojawia na stronie. Aby je ukryć, należy dokonać odwrotnego przekształcenia:

```
transform: translate3d(-100vw, 0, 0);
```

Najważniejszą rzeczą, o której nie wspomniałem, jest animacja sprawiająca, że przesuwanie jest bardziej efektowne. Implementuje się ją za pomocą właściwości CSS transition (przejście) w następujący sposób:

```
transition: transform .3s cubic-bezier(0, .52, 0, 1);
```

Jeżeli nie znasz tej właściwości, nie przejmuj się, jest ona bardzo prosta. Nie będę jej tutaj opisywał, zamiast tego zachęcam Cię do poświęcenia chwili na przeczytanie artykułu *Introduction to CSS Transitions* („Wprowadzenie do przejść CSS”) na stronie https://www.kirupa.com/html5/introduction_css_transitions.htm.

Do tej pory omówiliśmy ogólną ideę wysuwanego menu. Trzeba jeszcze wziąć pod uwagę kilka szczegółów. Zajmiemy się nimi w następnej części rozdziału, opisującej właściwe kodowanie menu.

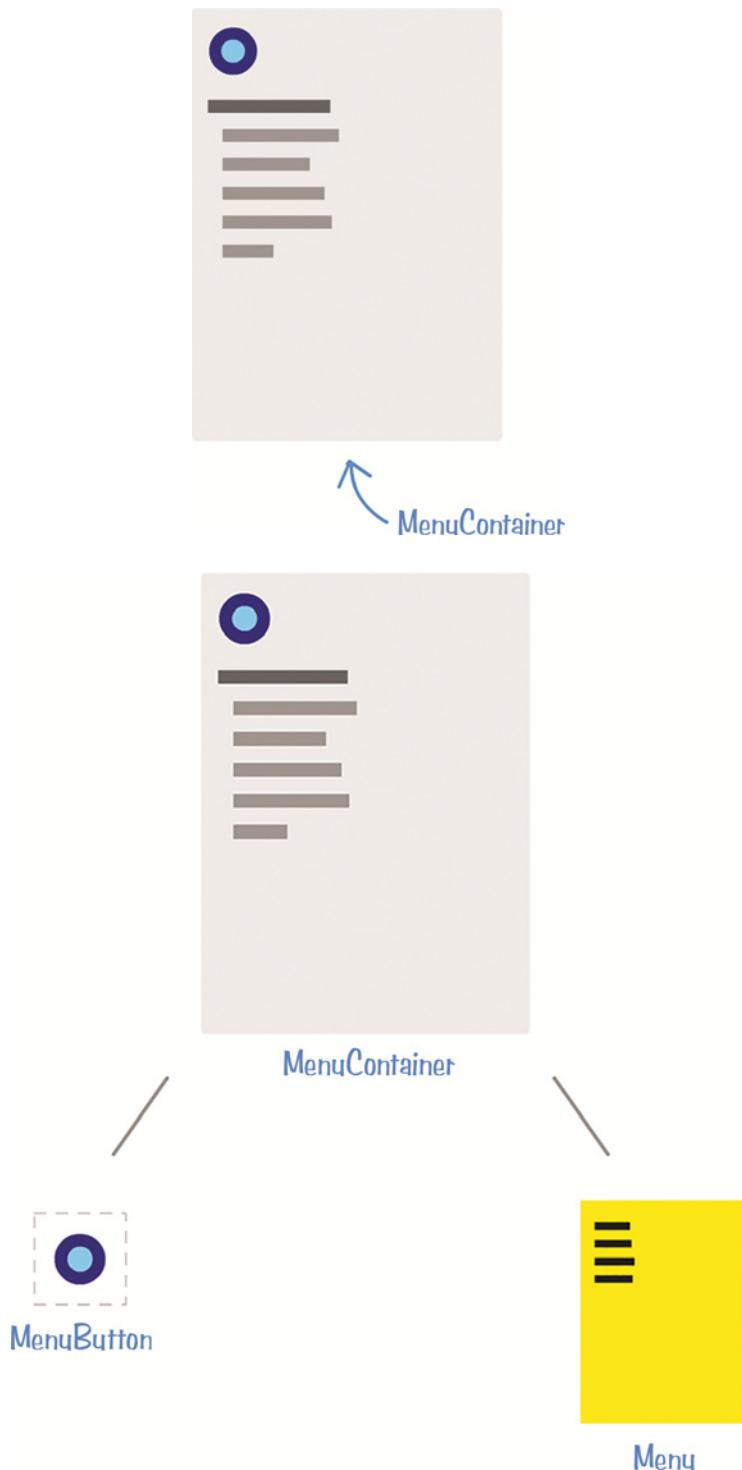
Przygotowanie wysuwanego menu

Teraz, kiedy masz już ogólne wyobrażenie o działaniu wysuwanego menu, czas wykorzystać teoretyczną wiedzę do utworzenia ciekawego kodu JSX. Pierwszą rzeczą jest zaplanowanie komponentów, z których będzie składała się aplikacja.

Na szczerze hierarchii będzie znajdował się komponent MenuContainer — patrz pierwszy rysunek na następnej stronie.

Ten komponent nie będzie odpowiedzialny za wyświetlanie menu, tylko za zarządzanie jego stanem. Ponadto będzie on zawierał komponenty Menu i MenuButton oraz pewną początkową treść. Jego ogólna struktura wygląda następująco — patrz drugi rysunek na następnej stronie.

W następnych częściach rozdziału utworzysz powyższe komponenty i gotową, działającą aplikację.



Pierwsze kroki

Z pomocą polecenia `create-react-app` utwórz nowy projekt o nazwie `slidingmenu`. Jeżeli nie pamiętasz, jak to się robi, zajrzyj do rozdziału 13. „Konfiguracja środowiska React bez stresu”, gdzie został szczegółowo opisany proces tworzenia i rozwijania projektu opartego na bibliotece React.

Aplikację zbudujesz od podstaw, zatem po utworzeniu projektu usuń zawartość folderów `public` i `src`. Za chwilę dodasz wszystkie niezbędne pliki.

Najpierw przygotuj dokument HTML. W tym celu w folderze `public` utwórz plik `index.html` i wpisz w nim poniższy kod:

```
<!DOCTYPE html>
<html>

<head>
  <title>Wysuwane menu</title>
</head>

<body>
  <div id="container"></div>
</body>

</html>
```

Powyższa strona jest po prostu miejscem, w którym będzie umieszczana i wyświetlana zawartość komponentów React.

Następnie w folderze `src` utwórz plik `index.js`, który będzie zawierał główny kod aplikacji. Umieść w nim następującą treść:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import MenuContainer from "./MenuContainer";

ReactDOM.render(
  <MenuContainer/>,
  document.querySelector("#container")
);
```

Metoda `render()` wyświetla tu zawartość komponentu `MenuContainer` w zdefiniowanym wcześniej elemencie `div` o identyfikatorze `container`. Instrukcje `import` importują biblioteki React i ReactDOM, jak również pliki ze stylami CSS i kodem komponentu `MenuContainer`. To jest cała zawartość pliku `index.js`.

Teraz w tym samym folderze `src` utwórz plik o nazwie `index.css` i umieść w nim kod stylizujący wygląd aplikacji. Wpisz poniższe dwie reguły:

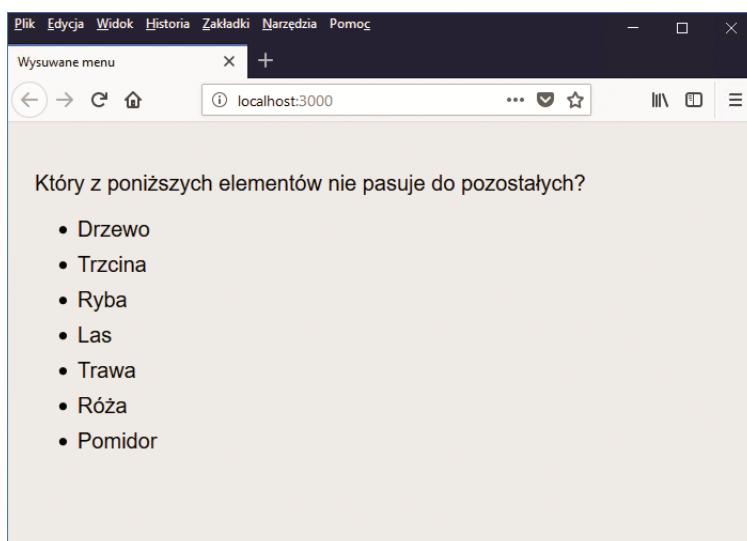
```
body {
  background-color: #EEE;
  font-family: sans-serif;
  font-size: 20px;
  padding: 25px;
  margin: 0;
  overflow: auto;
}
```

```
#container li {  
    margin-bottom: 10px;  
}
```

Ponieważ niewiele da się powiedzieć na temat powyższych reguł, ostatnią rzeczą niezbędną do uruchomienia aplikacji jest zakodowanie komponentu `MenuContainer`. W folderze `src` utwórz plik `MenuContainer.js` i wpisz w nim poniższy kod JSX i JavaScript:

```
import React, { Component } from "react";  
  
class MenuContainer extends Component {  
    render() {  
        return (  
            <div>  
                <div>  
                    <p>Który z poniższych elementów nie pasuje do pozostałych?</p>  
                    <ul>  
                        <li>Drzewo</li>  
                        <li>Trzcina</li>  
                        <li>Ryba</li>  
                        <li>Las</li>  
                        <li>Trawa</li>  
                        <li>Róża</li>  
                        <li>Pomidor</li>  
                    </ul>  
                </div>  
            </div>  
        );  
    }  
}  
  
export default MenuContainer;
```

Pamiętaj o zapisaniu wszystkich plików i sprawdzeniu (za pomocą polecenia `npm start`), czy aplikacja w początkowej postaci działa poprawnie. Jeżeli się nie pomyliłeś, powinna otworzyć się przeglądarka z następującą stroną:



Nie ma tu jeszcze wysuwanego menu ani przycisku. Oba elementy dodasz za chwilę w następnych częściach rozdziału.

Wyświetlanie i ukrywanie menu

Po wstępny przygotowaniu aplikacji czas na najciekawszą część: utworzenie właściwego menu, które będzie można wyświetlać i ukrywać w następujący sposób:

1. Po kliknięciu przycisku menu wysunie się i pojawi na stronie.
2. Po kliknięciu dowolnego odnośnika menu przesunie się poza stronę.

Oznacza to, że należy rozważyć kilka kwestii. Trzeba zapisywać stan menu, aby było wiadomo, czy jest wysunięte czy schowane. Stan ten musi być zmieniany zarówno za pomocą przycisku, jak i odnośników menu, ponieważ kliknięcie każdego z tych elementów będzie miało wpływ na widoczność menu. Stan musi być przechowywany w ogólnodostępnym miejscu, aby i przycisk, i samo menu mogły się do niego odwoływać. Tym miejscem będzie komponent `MenuContainer`, zatem wpisz teraz kod obsługujący stan menu. W pliku `MenuContainer.js` umieść powyżej metody `render()` kod konstruktora i metody `toggleMenu()`, jak niżej:

```
constructor(props) {
  super(props);

  this.state = {
    visible: false
  };

  this.toggleMenu = this.toggleMenu.bind(this);
}

toggleMenu() {
  this.setState({
    visible: !this.state.visible
  });
}
```

Powyższy kod powinien być teraz dla Ciebie spacerkiem na świeżym powietrzu. Zdefiniowana jest w nim właściwość `visible` obiektu `state` oraz metoda `toggleMenu()` zmieniająca wartość tej zmiennej z `true` na `false` i odwrotnie.

Następnym zadaniem jest zaimplementowanie obsługi zdarzenia `click` zgłaszanego przez przycisk i menu. Stan musi być zmieniany wewnątrz komponentu `MenuContainer`, zatem również w tym miejscu musi znajdować się metoda obsługująca powyższe zdarzenie. Wpisz teraz wyróżnione niżej wiersze:

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

class MenuContainer extends Component {
  constructor(props) {
    super(props);

    this.state = {
      visible: false
    };

    this.handleMouseDown = this.handleMouseDown.bind(this);
    this.toggleMenu = this.toggleMenu.bind(this);
  }
}
```

```

handleMouseDown(e) {
  this.toggleMenu();
  console.log("clicked");
  e.stopPropagation();
}

toggleMenu() {
  this.setState({
    visible: !this.state.visible
  });
}
...
}

```

Metoda `handleMouseDown()` wywołuje metodę `toggleMenu()` zmieniającą widoczność menu. W tym momencie zapewne zastanawiasz się, gdzie znajduje się właściwy kod obsługujący zdarzenie. Gdzie dokładnie jest wywoływana metoda `handleMouseDown()`? Odpowiedź brzmi: nigdzie! Niektóre rzeczy implementujemy w odwrotnej kolejności. Najpierw zdefiniowałeś metodę obsługującą zdarzenie, a za chwilę, przy okazji tworzenia komponentów przycisku i menu, zajmiesz się powiązaniem tej metody z samym zdarzeniem.

Utworzenie przycisku

W folderze `src` utwórz pliki `MenuButton.js` i `MenuButton.css`. Następnie otwórz w edytorze plik `MenuButton.js` i wpisz w nim następujący kod:

```

import React, { Component } from "react";
import './MenuButton.css';

class MenuButton extends Component {
  render() {
    return (
      <button id="roundButton"
        onMouseDown={this.props.handleMouseDown}></button>
    );
  }
}

export default MenuButton;

```

Popatrz przez chwilę, co robi ten kod. Niewiele się w nim dzieje. Definiowany jest element `button` z identyfikatorem `roundButton` i związane zdarzenie `onMouseDown` z właściwością `handleMouseDown`. Zanim przejdziemy dalej, otwórz plik `MenuButton.css` i umieść w nim następujące reguły stylów:

```

#roundButton {
  background-color: #96D9FF;
  margin-bottom: 20px;
  width: 50px;
  height: 50px;
  border-radius: 50%;
  border: 10px solid #0065A6;
  outline: none;
  transition: all .2s cubic-bezier(0, 1.26, .8, 1.28);
}

#roundButton:hover {

```

```

background-color: #96D9FF;
cursor: pointer;
border-color: #003557;
transform: scale(1.2, 1.2);
}

#roundButton:active {
  border-color: #003557;
  background-color: #FFF;
}

```

Teraz pora na użycie nowo utworzonego komponentu `MenuButton`. Wróć do komponentu `MenuContainer` i w metodzie `render()` wpisz wyróżniony niżej wiersz:

```

render() {
  return (
    <div>
      <MenuButton handleMouseDown={this.handleMouseDown}>/>
      <div>
        ...
      </div>
    );
}

```

Aby wpisany kod działał, na początku pliku `MenuContainer.js` musi być importowany komponent `MenuButton`. To jest coś, o czym można łatwo zapomnieć!

Zwróć uwagę, że komponent jest wywoływany z właściwością `handleMouseDown`, której jest przypisana zdefiniowana wcześniej metoda `MouseDown()` obsługująca zdarzenie. Dzięki temu, gdy klikniesz przycisk umieszczony wewnątrz komponentu `MenuButton`, zostanie wywołana metoda `handleMouseDown()` zdefiniowana w komponencie `MenuContainer`. Wszystko wygląda dobrze, ale przycisk, który nie powoduje wysunięcia menu, nie jest zbyt przydatny. Teraz to naprawimy.

Tworzenie menu

Czas utworzyć komponent `Menu` odpowiedzialny za obsługę menu. Zanim go faktycznie zakodujesz, przyjmij, że już istnieje i że musi być wywoływany za pomocą metody `render()` komponentu `MenuContainer`. Umieść wewnątrz tej metody, pod wpisanym już wcześniej wierszem wywołującym komponent `MenuButton`, wyróżniony niżej kod, który wywołuje nieistniejący jeszcze komponent `Menu`:

```

render() {
  return (
    <div>
      <MenuButton handleMouseDown={this.handleMouseDown} />
      <Menu handleMouseDown={this.handleMouseDown}
            menuVisibility={this.state.visible} />
      <div>
        ...
      </div>
    );
}

```

Dopisz również instrukcję importującą plik `Menu.js`.

Wróćmy teraz do komponentu `Menu` i przyjrzyjmy się przekazywanym mu właściwościom. Pierwsza z nich, `handleMouseDown`, zawierająca obsługującą zdarzenie metodę `handleMouseDown` jest Ci już znana. Drugą właściwością jest `menuVisibility`. Zawiera ona bieżącą wartość właściwości

visible obiektu state. Utwórz teraz właściwy komponent Menu i dowiedz się, jak wykorzystuje on powyższe właściwości oraz kilka innych rzeczy.

W tym samym folderze *src*, w którym przebywasz od dłuższego czasu, utwórz pliki *Menu.js* i *Menu.css*. W pierwszym z nich wpisz poniższy kod:

```
import React, { Component } from "react";
import "./Menu.css";

class Menu extends Component {
  render() {
    var visibility = "hide";

    if (this.props.menuVisibility) {
      visibility = "show";
    }

    return (
      <div id="flyoutMenu"
        onMouseDown={this.props.handleMouseDown}
        className={visibility}>
        <h2><a href="#">Strona główna</a></h2>
        <h2><a href="#">O nas</a></h2>
        <h2><a href="#">Kontakt</a></h2>
        <h2><a href="#">Szukaj</a></h2>
      </div>
    );
  }
}

export default Menu;
```

Zwróć uwagę na kod JSX w instrukcji `return`. Znajduje się tam element `div` o nazwie `flyoutMenu`, zawierający przykładową treść. Wewnątrz tego elementu określona jest metoda `handleMouseDown()` (przekazywana do komponentu za pomocą właściwości) obsługująca zdarzenie `onMouseDown`. Określona jest również klasa elementu jako wartość zmiennej `visibility`. Jak pamiętasz, nie można w kodzie JSX używać słowa kluczowego `class` z języka JavaScript, dlatego klasa elementu jest określona pośrednio za pomocą atrybutu `className`.

Wróćmy do naszego kodu. Wartość zmiennej `visibility` została zdefiniowana kilka wierszy wcześniej:

```
var visibility = "hide";

if (this.props.menuVisibility) {
  visibility = "show";
}
```

Zmienna przyjmuje wartości `hide` lub `show` w zależności od tego, czy właściwość `menuVisibility` (której wartość została określona za pomocą właściwości `visible` obiektu `state`) ma wartość `false` czy `true`. Niepozorny kod przypisujący wartość atrybutowi `className` w rzeczywistości odgrywa istotną rolę w określaniu widoczności menu. Gdy dodasz niezbędne reguły CSS, dowieš się, dlaczego tak jest. Otwórz więc plik *Menu.css* i wpisz w nim poniższy kod:

```
#flyoutMenu {
  width: 100vw;
  height: 100vh;
  background-color: #FFE600;
  position: fixed;
```

```

top: 0;
left: 0;
transition: transform .3s
            cubic-bezier(0, .52, 0, 1);
overflow: scroll;
z-index: 1000;
}

#flyoutMenu.hide {
  transform: translate3d(-100vw, 0, 0);
}

#flyoutMenu.show {
  transform: translate3d(0vw, 0, 0);
  overflow: hidden;
}

#flyoutMenu h2 a {
  color: #333;
  margin-left: 15px;
  text-decoration: none;
}

#flyoutMenu h2 a:hover {
  text-decoration: underline;
}

```

Większość powyższych reguł CSS określa wygląd menu, ale reguły `#flyoutMenu.hide` i `#flyoutMenu.show` definiują jego widoczność. Wyborem odpowiedniej reguły zajmuje się wyłącznie opisany wcześniej kod. Jak pamiętasz, w generowanym kodzie HTML (odwołującym się do powyższych reguł CSS) zmieniana jest klasa elementu `div` o nazwie `flyoutMenu` z `hide` na `show` i odwrotnie. Pomyślowe, prawda?

W tym momencie zakończyłeś kodowanie. Pamiętaj o zapisaniu zmian i sprawdzeniu, czy aplikacja działa zgodnie z opisem na początku rozdziału. I nie usuwaj tego projektu. Zajmiemy się nim jeszcze i poprawimy kilka mankamentów.

Podsumowanie

W tym rozdziale poznałeś jeden z pierwszych przykładów zastosowania biblioteki React do utworzenia często stosowanego elementu interfejsu użytkownika — wysuwanego menu. Przy okazji dowiedziałeś się więcej o współdziałaniu komponentów, na przykład obsłudze zdarzeń, współdzieleniu stanu itp. Gdy poznasz inne przykłady zastosowania biblioteki React, przekonasz się, że oferuje ona znacznie więcej funkcjonalności niż tutaj opisane. Cała sztuka polega na umiejętnościem wykorzystaniu poznanych koncepcji i łączeniu ich w coraz bardziej złożone scenariusze. Nie oznacza to jednak, że na tym koniec. Biblioteka React zawiera jeszcze wiele innych funkcjonalności, a Ty musisz jeszcze utworzyć kilka aplikacji i dokładnie poznać ich działanie!

Jeżeli napotkasz problemy, pyтай!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pyтай сміаю! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

Zapobieganie niepotrzebnemu wyświetaniu komponentów

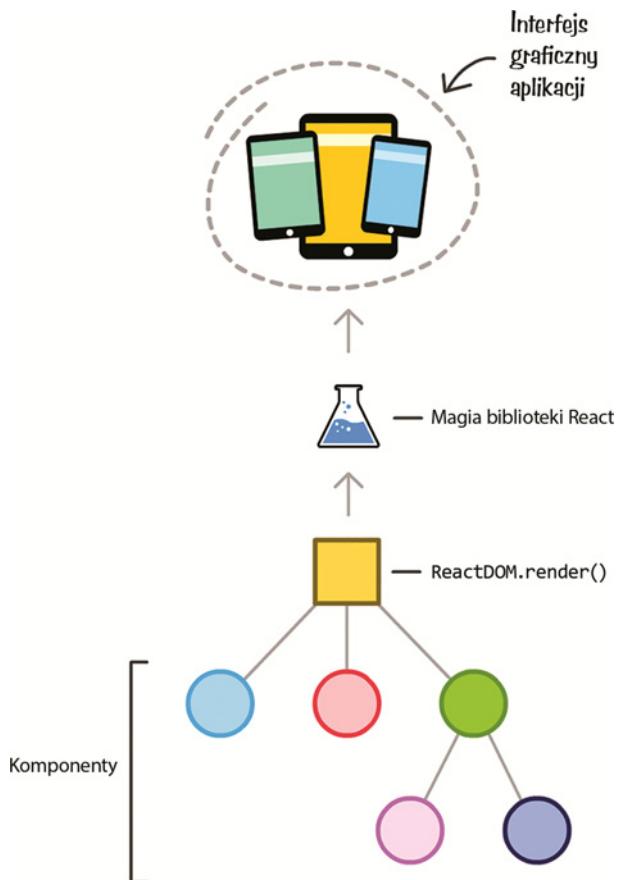
Prawdopodobnie masz już dość mojego powtarzania, że jednym z największych atutów biblioteki React jest jej wysoka wydajność. Nie oznacza to jednak, że dostaje się ją za darmo. Wprawdzie aplikacja oparta na bibliotece React dobrze sobie radzi z dużym obciążeniem, ale trzeba również umiejętnie stosować tę bibliotekę, aby nie wykonywała niepotrzebnych operacji, które mogłyby spowalniać działanie aplikacji. Należy między innymi zwracać baczną uwagę, aby metoda `render()` każdego komponentu była wywoływana tylko raz i tylko wtedy, gdy jest to absolutnie konieczne. W kolejnych częściach rozdziału dowiesz się, dlaczego jest to takie ważne i co można w tej sprawie zrobić.

Metoda `render()`

Oficjalne przeznaczenie metody `render()` jest bardzo proste. Musi ona wyświetlać komponent i zwracać kod JSX na żądanie komponentu nadrzędnego. Ogólny schemat przepływu informacji począwszy od poszczególnych komponentów do interfejsu aplikacji wygląda następująco — patrz rysunek na następnej stronie.

Na szczycie struktury znajduje się interfejs graficzny aplikacji, a na samym dole komponenty, z których składa się aplikacja. Każdy z tych komponentów zawiera metodę `render()` zwracającą własny kod JSX połączony z kodami JSX pochodzący od innych komponentów. Proces ten powtarza się dotąd, aż finalny kod JSX dotrze na szczyt hierarchii komponentów, gdzie jest wywoływana metoda `ReactDOM.render()`. W tym miejscu dochodzi do glosu magia biblioteki React, przemieniająca kod JSX w osobne pliki HTML, CSS i JavaScript, które wyświetla przeglądarka.

Teraz, kiedy masz bardzo ogólne wyobrażenie o działaniu biblioteki React, wróćmy do szczegółów działania komponentów i ich metod `render()`. Zapewne zauważyleś, że w utworzonych do tej pory aplikacjach w żadnym z komponentów nie była jawnie wywoływana ta metoda. Jest tak, ponieważ ta operacja jest wykonywana automatycznie w trzech przypadkach:



1. gdy ulegną zmianie właściwości komponentu;
2. gdy ulegnie zmianie stan komponentu;
3. gdy komponent nadrzędny wywoła swoją metodę `render()`.

We wszystkich trzech przypadkach pożądane wydaje się automatyczne wywoływanie metody `render()` komponentu. Przecież za każdym razem może zmienić się jego wygląd, prawda?

Odpowiedź na powyższe pytanie brzmi: *to zależy*. Czasami wymuszane jest ponowne wyświetlanie komponentów, mimo że zmiana ich właściwości lub stanu nie ma absolutnie żadnego znaczenia. Czasami komponent nadrzędny jest poprawnie wyświetlany pierwszy lub kolejny raz, ale tylko lokalnie, ponieważ nie ma potrzeby, aby wyświetlane były komponenty podrzędne, które nie mają ze zmianą nic wspólnego.

Z tego wszystkiego wynika, że niepotrzebne wykonywanie operacji niepotrzebnie wykonywanych tuż przed naszymi oczami. Należy pamiętać o tym, że wywołanie metody `render()` nie oznacza aktualizacji całego modelu DOM. Biblioteka React wykonuje kilka operacji porównujących poprzednią wersję modelu z nową (lub bieżącą) i sprawdza, czy jakieś zmiany muszą zostać odzwierciedlone. Te operacje oznaczają dodatkową pracę do wykonania. W przypadku skomplikowanych aplikacji złożonych z wielu komponentów operacje te się kumulują.

Niektóre z nich są wykonywane wewnętrznie przez bibliotekę React, inne są ważnymi operacjami wykonywanymi przez metody render(). Często metody te zawierają wiele kodu generującego kod JSX. Rzadko zdarza się, aby metoda zwracała statyczny kod JSX, bez uprzedniego wykonania pewnych obliczeń. Dlatego minimalizacja liczby wywołań metod render() jest bardzo ważna.

Optymalizacja wywołań metody render()

Teraz, gdy mamy rozpoznany problem, zajmijmy się kilkoma technikami umożliwiającymi wywoływanie metod render() komponentów tylko wtedy, gdy jest to konieczne. Techniki te zostały opisane w następnych częściach rozdziału.

Kontynuacja przykładu

Aby lepiej poznać problem, przeanalizujmy przykładową aplikację. Nie będzie to jednak zwykła aplikacja. Przyjrzymy się ponownie zbudowanemu w poprzednim rozdziale wysuwanemu menu. Jeżeli je utworzyłeś, otwórz jego kod w edytorze, a jeżeli nie — nie przejmuj się. Za pomocą polecenia `create-react-app` utwórz nowy projekt, usuń całą zawartość folderów `public` i `src` i umieść w nich dołączone do książki pliki zapisane w folderze `r16`.

Gdy projekt wysuwanego menu będzie gotowy, uruchom aplikację i sprawdź, czy działa poprawnie.

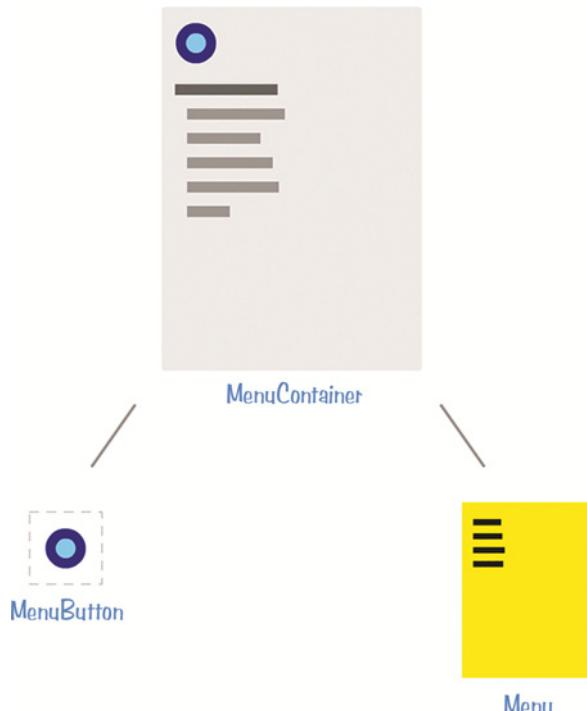
Jeżeli nie utworzyłeś aplikacji opisanej w poprzednim rozdziale, bardzo zachęcam Cię, abyś to zrobił. Użycie gotowego projektu jest całkowicie dopuszczalne i poprawne, ale ważna jest znajomość działania kodu i decyzji podjętych podczas jego tworzenia. Możesz oczywiście kontynuować lekturę tego rozdziału bez jego pełnego zrozumienia, ale pamiętaj, że Cię ostrzegałem, że niektóre fragmenty kodu mogą Ci się wydać nie ma miejsca.

Poniżej znajduje się rysunek (który już widziałeś) przedstawiający hierarchię komponentów tworzących wysuwane menu — patrz rysunek na następnej stronie.

Na szczycie hierarchii znajduje się komponent `MenuContainer` wykorzystujący dwa komponenty podrzędne: `MenuButton` i `Menu`. Na rysunku nie została pokazana wywoływana w pliku `index.js` metoda `ReactDOM.render()`, która przesyła komponent `MenuContainer` do modelu DOM:

```
ReactDOM.render(  
  <MenuContainer/>,  
  document.getElementById("container")  
>;
```

Gdy użytkownik kliknie przycisk zdefiniowany za pomocą komponentu `MenuButton`, właściwość `visible` komponentu `MenuContainer` przyjmie logiczną wartość `true`. Zmiana właściwości powoduje, że zmieniana jest klasa komponentu `Menu` i stosowana jest odpowiednia reguła CSS powodująca wysunięcie menu. Kliknięcie menu skutkuje przypisaniem właściwości komponentu `MenuContainer` wartości `false` i wykonaniem odwrotnej operacji.



Monitorowanie wywołań metod render()

Pierwszą rzeczą, którą się zajmiemy, jest monitorowanie wywołań metod `render()`. Można to osiągnąć na kilka sposobów. Pierwszy polega na użyciu dostępnych w przeglądarce narzędzi programistycznych, ustawnieniu pułapek w kodzie i kontrolowaniu wartości zmiennych. Innym sposobem jest zainstalowanie dodatku *React Developer Tools* (dla przeglądarek Chrome i Firefox), dostępnego na stronie <https://github.com/facebook/react-devtools>, i monitorowanie poszczególnych komponentów. Oprócz tego można zastosować bardzo proste rozwiązań polegające na wpisaniu instrukcji `console.log()` w metodach `render()`, które mają być monitorowane.

Ponieważ nasza przykładowa aplikacja składa się tylko z trzech komponentów, więc najprościej będzie zastosować sposób z instrukcją `console.log()`. Otwórz więc w edytorze pliki `MenuContainer.js`, `MenuButton.js` i `Menu.js`, odszukaj w nich metody `render()` i na początku każdej z nich wpisz odpowiednią instrukcję `console.log()`, jak niżej.

W pliku `MenuContainer.js` wpisz następujący wiersz:

```
render() {
  console.log("Wyświetlenie komponentu MenuContainer");
  return (
    <div>
      <MenuButton handleMouseDown={this.handleMouseDown}/>
      <Menu handleMouseDown={this.handleMouseDown}
        menuVisibility={this.state.visible} />
```

```
<div>
  <p>Który z poniższych elementów nie pasuje do pozostałych?</p>
  <ul>
    <li>Drzewo</li>
    <li>Trzcina</li>
    <li>Ryba</li>
    <li>Las</li>
    <li>Trawa</li>
    <li>Róża</li>
    <li>Pomidor</li>
  </ul>
</div>
</div>
);
}
```

Analogicznie zmień plik *MenuButton.js*:

```
render() {
  console.log("Wyświetlenie komponentu MenuButton");
  return (
    <button id="roundButton"
      onMouseDown={this.props.handleMouseDown}></button>
  );
}
```

Na koniec wpisz instrukcję w pliku *Menu.js*:

```
render() {
  console.log("Wyświetlenie komponentu MenuButton");

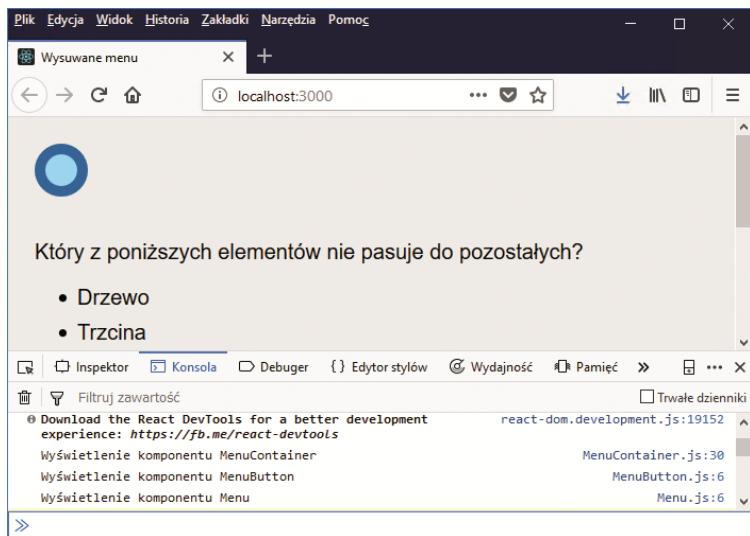
  var visibility = "hide";

  if (this.props.menuVisibility) {
    visibility = "show";
  }

  return (
    <div id="flyoutMenu"
      onMouseDown={this.props.handleMouseDown}
      className={visibility}>
      <h2><a href="#">Strona główna</a></h2>
      <h2><a href="#">O nas</a></h2>
      <h2><a href="#">Kontakt</a></h2>
      <h2><a href="#">Szukaj</a></h2>
    </div>
  );
}
```

Po wpisaniu wyróżnionych wierszy otwórz aplikację. Następnie otwórz w przeglądarce narzędzia programistyczne i sprawdź komunikaty pojawiające się w konsoli — patrz rysunek na następnej stronie.

Zobaczysz ostrzeżenia i inne informacje, ale poszukaj komunikatów wyświetlanych przez instrukcje `console.log()`, które wpisałeś. Zauważ, że zaraz po uruchomieniu aplikacji wszystkie trzy komponenty wywołują swoje metody `render()`. Jest to zrozumiałe, ponieważ jest to pierwsze uruchomienie aplikacji.



Przy otwartej cały czas konsoli kliknij niebieski przycisk wyświetlający menu. Zwróć uwagę, że w konsoli pojawią się nowe komunikaty (wyróżnione niżej pogrubioną czcionką):

```
Wyświetlenie komponentu MenuContainer
Wyświetlenie komponentu MenuButton
Wyświetlenie komponentu Menu
clicked
Wyświetlenie komponentu MenuContainer
Wyświetlenie komponentu MenuButton
Wyświetlenie komponentu Menu
```

Gdy wywoływana jest metoda `handleMouseDown()` obsługująca zdarzenie, w konsoli pojawia się komunikat `clicked`. Nie jest on dla nas ważny, ale rozdziela serie komunikatów wyświetlanych przez metody `render()`. Zwróć uwagę, że wysunięcie menu powoduje wywołanie metod `render()` wszystkich trzech komponentów.

Kliknij teraz menu, aby je ukryć. Okaże się, że znowu zostaną wywołane wszystkie trzy metody. To nie jest pożądany efekt, prawda?

Ponieważ zmieniana jest właściwość komponentu `Menu` i stan komponentu `MenuContainer`, zrozumiałe jest (na razie), że oba wywołują swoje metody `render()`. Dlaczego jednak za każdym razem jest wywoływana metoda komponentu `MenuButton()`?

Gdy przyjrzyzysz się metodzie `render()` komponentu `MenuContainer`, zauważysz, że wywołuje ona komponent `Menu` i przekazuje mu właściwość, której wartość nigdy się nie zmienia:

```
<MenuButton handleMouseDown={this.handleMouseDown}/>
```

Wartość właściwości `handleMouseDown` nie zmienia się, gdy menu jest wyświetlane lub ukrywane. Metoda `render()` jest wywoływana dlatego, że komponent `MenuContainer` (nadzędny dla komponentu `MenuButton`) wywołuje własną metodę `render()`. Jeżeli komponent nadzędny wywołuje własną metodę `render()`, wtedy wszystkie komponenty podrzędne również wywołują swoje metody. Jest to trzeci przypadek z podanej wcześniej listy sytuacji, w których metoda `render()` jest wywoływana automatycznie.

Co można więc zrobić, aby komponent `MenuButton` nie wywoływał niepotrzebnie swojej metody `render()`? Wyjścia są dwa.

Modyfikacja aktualizacji komponentu

W jednym z poprzednich rozdziałów poznałeś metody wywoływane przez bibliotekę React w trakcie cyklu życia komponentu. Jedną z nich jest `shouldComponentUpdate()` wywoływana tuż przed metodą `render()`. Metoda `render()` nie jest jednak wywoływana w przypadku, gdy metoda `shouldComponentUpdate()` zwróci wynik `false`. Wykorzystamy to w naszej aplikacji.

W kodzie komponentu `MenuButton` wpisz wyróżnione niżej wiersze:

```
import React, { Component } from "react";
import './MenuButton.css';

class MenuButton extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    return false;
  }

  render() {
    console.log("Wyświetlenie komponentu MenuButton");
    return (
      <button id="roundButton"
        onMouseDown={this.props.handleMouseDown}></button>
    );
  }
}

export default MenuButton;
```

Odśwież stronę, aby sprawdzić działanie kodu. Zwróć uwagę na informacje pojawiające się w konsoli podczas wyświetlania i ukrywania menu. Przy pierwszym uruchomieniu aplikacji i wyświetleniu menu pojawią się następujące komunikaty:

```
Wyświetlenie komponentu MenuContainer
Wyświetlenie komponentu MenuButton
Wyświetlenie komponentu Menu
clicked
Wyświetlenie komponentu MenuContainer
Wyświetlenie komponentu Menu
```

Zauważ, że nie została tym razem wywołana metoda `render()` komponentu `MenuButton`. To dobry znak. Nie fetuj jednak sukcesu zbyt wcześnie. Rozwiążanie problemu polegające na zastosowaniu metody `shouldComponentUpdate()` zawsze zwracającej wynik `false` jest bardzo prymitywne. Dzięki niemu oczywiście kod działa poprawnie, ale trzeba je stosować ostrożnie, ponieważ może blokować ważne aktualizacje komponentu `MenuButton`, gdy w przyszłości zmienisz jego kod lub sposób użycia.

Jeżeli przyjrzyzysz się sygnaturze metody `shouldComponentUpdate()`, zauważysz, że ma ona dwa argumenty. Pierwszy zawiera nową wartość właściwości, a drugi nowy stan komponentu. Argumenty te można wykorzystać do porównania aktualnej i nowej wersji komponentu i w bardziej intelligentny sposób kontrolować wywoływanie metody `render()`. Jedyną właściwością przekazywaną komponentowi `MenuButton` jest `handleMouseDown`. Metoda `shouldComponentUpdate()` może sprawdzać w przedstawiony niżej sposób, czy właściwość ta uległa zmianie:

```
shouldComponentUpdate(nextProps, nextState) {
  if (nextProps.handleMouseDown === this.props.handleMouseDown) {
    return false;
  } else {
    return true;
  }
}
```

Powyższy kod sprawia, że metoda render() nie zostanie wywołana, jeśli wartość właściwości handleMouseDown się nie zmieni. Jeżeli właściwość ta zmieni wartość, metoda shouldComponentUpdate() zwróci wartość true i zostanie wywołana metoda render(). Można określić również inne warunki wywoływania metody render(). Zależy to wyłącznie od działania problematycznego komponentu. Tutaj możesz wykazać się własną kreatywnością.

Komponent PureComponent

Często się zdarza, że komponenty są wyświetlane wielokrotnie, mimo że nie zostały istotnie zmienione ich właściwości lub stany. Jednym z przykładów jest nasz komponent *MenuButton*. Wyjściem z tej sytuacji jest sprawdzanie za pomocą metody shouldComponentUpdate(), czy zostały istotnie zmienione właściwości lub stan komponentu. Aby uniknąć każdorazowego sprawdzania zmian, można zastosować specjalny komponent *PureComponent*, który robi to automatycznie.

Do tej pory tworzyłeś komponenty oparte na klasie Component, jak niżej:

```
class MójKomponent extends Component {
  render() {
    return (
      <h1>Cześć!</h1>
    );
  }
}
```

Aby utworzyć komponent oparty na klasie *PureComponent*, wystarczy wprowadzić wyróżnioną niżej zmianę:

```
class MójKomponent extends PureComponent {
  render() {
    return (
      <h1>Cześć!</h1>
    );
  }
}
```

To wszystko. Teraz Twój komponent będzie wywoływał metodę render() tylko wtedy, gdy zmienią się jego właściwości lub stan. Aby się o tym przekonać, zmień w pliku *MenuButton.js* klasę Component na PureComponent. Ponadto usuń metodę shouldComponentUpdate(), ponieważ nie będzie już potrzebna. Poniższy kod zawiera dwie wyróżnione zmiany:

```
import React, { PureComponent } from "react";
import './MenuButton.css';

class MenuButton extends PureComponent {

  render() {
    console.log("Wyświetlenie komponentu MenuButton");
    return (

```

```

        <button id="roundButton"
            onMouseDown={this.props.handleMouseDown}></button>
    );
}
}

export default MenuButton;

```

Pierwszy wiersz importuje kod niezbędny do działania komponentu PureComponent. Ponadto klasa MenuButton dziedziczy cechy klasy PureComponent. To wszystko. Jeżeli otworzysz teraz aplikację i sprawdzisz zawartość konsoli, zauważysz, że podczas wyświetlania i ukrywania menu nie jest wywoływana metoda render() komponentu MenuButton.

Dlaczego nie można zawsze stosować komponentu PureComponent?

Komponent PureComponent wydaje się bardzo praktyczny, prawda? Dlaczego więc nie można go stosować zawsze i zapomnieć o komponencie Component? Należałoby! Jest jednak kilka powodów, dla których nie można tego zrobić.

Przede wszystkim komponent PureComponent wykonuje tzw. płytkie porównanie. Nie jest to drobiazgowa ocena, czy zmiany właściwości lub stanu wymagają ponownego wyświetlenia komponentu. W wielu przypadkach płytkie porównanie jest akceptowalne, w innych może nie być wystarczające. Pamiętaj o tym, kiedy będziesz stosował komponent PureComponent. Może się okazać, że potrzebna będzie metoda shouldComponentUpdate() i zakodowanie w niej własnego algorytmu aktualizacji komponentu. Nie można jednak tej metody stosować w komponencie PureComponent, mimo że nadara się ku temu okazja!

Większym problemem niż zastosowany algorytm porównywania danych jest wydajność komponentu PureComponent. Sprawdzanie, nawet płytkie, zmian właściwości i stanów wszystkich komponentów zajmuje trochę czasu. Pamiętaj, że sprawdzanie odbywa się za każdym razem, gdy komponent ma być wyświetlony lub zażąda tego jego komponent nadrzędny. W złożonych interfejsach ma to miejsce częściej, niż może Ci się wydawać.

Powiniennem był to napisać na początku tej noty: musisz wiedzieć, co zamierzasz osiągnąć. Generalnie, możesz stosować komponent PureComponent zamiast Component, ale musisz pamiętać o jego dwóch (małych) mankamentach.

Podsumowanie

Aby Twoja aplikacja działała z maksymalną wydajnością, musisz nieustannie zachowywać czujność. Wydajność kontroluj często, a obowiązkowo rób to po wprowadzeniu w kodzie zmian mających tę wydajność zwiększyć. Każda optymalizacja kodu powoduje jego dodatkowe skomplikowanie i zwiększa ilość pracy (Twojej lub Twojego zespołu) związanej z utrzymywaniem aplikacji i poprawianiem w niej błędów. Optymalizuj świadomie i bez przesady. Jeżeli aplikacja pracuje naprawdę dobrze na przeznaczonych dla niej urządzeniach i przeglądarkach (szczególnie tych prostszych), możesz uznać, że dobrze wykonałeś swoją robotę. Możesz się odprężyć i nie wykonywać żadnej dodatkowej pracy!

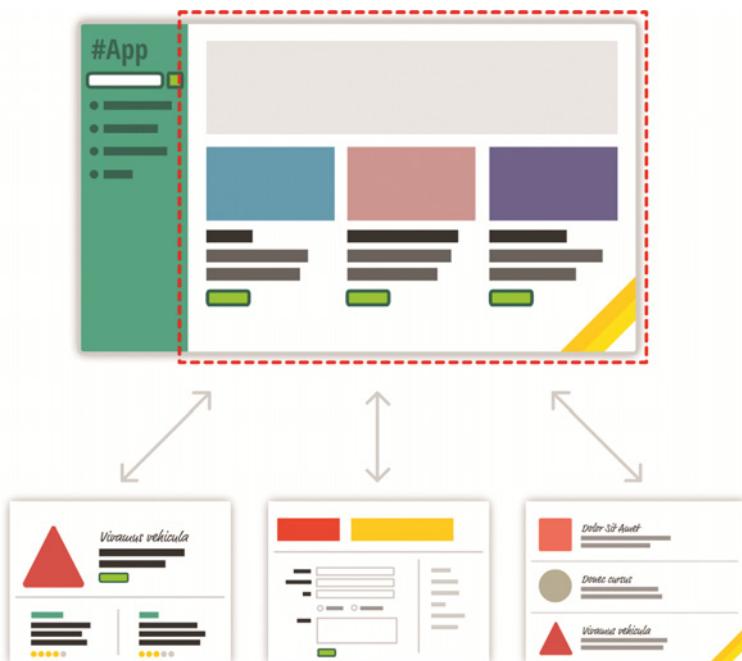
Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

Tworzenie jednostronowej aplikacji za pomocą biblioteki React Router

Teraz, gdy znasz już podstawy działania biblioteki React, zajmijmy się czymś bardziej ambitnym.

W tym rozdziale zbudujesz prostą, jednostronową aplikację. W pierwszym rozdziale książki, gdy przedstawałem bibliotekę React, pisałem, że aplikacje jednostronne istotnie różnią się od tradycyjnych aplikacji wielostronowych, które spotyka się wszędzie. Podstawowa różnica polega na tym, że poruszanie się po aplikacji jednostronnej nie wymaga otwierania nowych stron. Zamiast tego na tej samej stronie wyświetlane są różne widoki, co ilustruje poniższy rysunek:



Wyświetlanie treści na stronie bez jej przeładowywania nie jest łatwe. Najtrudniejszym zadaniem nie jest ładowanie samej zawartości — to dość prosta operacja. Wyznaniem jest utworzenie jednej strony, która działa tak, jak są do tego przyzwyczajeni użytkownicy. Mówiąc dokładniej: użytkownicy podczas korzystania z aplikacji oczekują, że:

1. Adres URL widoczny w przeglądarce zawsze odzwierciedla wyświetlana treść.
2. Przyciski wstecz i w przód działają w normalny sposób.
3. Można szybko przechodzić do określonego widoku, wpisując odpowiedni adres URL (tzw. głęboki odnośnik).

W aplikacji wielostronnej nie trzeba nic specjalnego robić, aby spełnić powyższe wymagania, ponieważ są one spełnione z definicji. Jednak poruszanie się po aplikacji jednostronnej nie polega na otwieraniu całkowicie nowych stron, więc trzeba włożyć dodatkową pracę, aby aplikacja działała w opisany wyżej sposób. Na przykład trzeba sprawić, aby adres URL odpowiednio się zmieniał, a przeglądarka poprawnie tworzyła historię stron, umożliwiającą korzystanie z przycisków wstecz i w przód. Jeżeli użytkownik utworzy zakładkę albo zanotuje adres URL i użycie go później, jednostronna aplikacja musi wyświetlić odpowiednią treść.

Aby osiągnąć opisane wyżej cele, trzeba zastosować technikę zwaną kierowaniem (ang. *routing*). Polega ona na kojarzeniu adresów URL z treściami, które nie są zawarte na osobnych stronach, tylko w widokach. Zadanie wydaje się trudne, ale na szczęście można użyć odpowiedniej biblioteki JavaScript. Jedną z nich, będącą prawdziwą perlą w tej dziedzinie, jest React Router (<https://github.com/reactjs/react-router>). Biblioteka ta umożliwia kierowanie zapytaniami w jednostronnej aplikacji opartej na bibliotece React. Zaleta biblioteki React Router polega na tym, że w znany Ci już sposób rozszerza możliwości biblioteki React i pozwala w pełni wykorzystać możliwości kierowania zapytaniami. W tym rozdziale dowiesz się wszystkiego o bibliotece React Router i o wielu innych rzeczach.

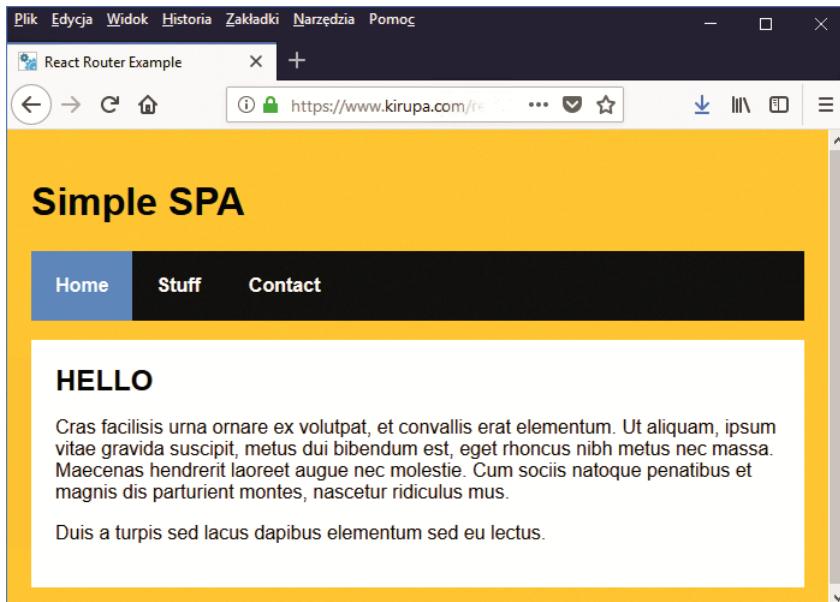
Do dzieła!

Przykład

Zanim przejdziemy dalej, przyjrzyj się aplikacji dostępnej pod adresem https://www.kirupa.com/react/examples/react_router/index.html. Jest to prosta aplikacja wykorzystująca bibliotekę React Router do nawigowania użytkownika i wyświetlania treści — patrz rysunek na następnej stronie.

Zwróć uwagę, że kliknięcie odnośnika powoduje wyświetlenie nowej treści; również przyciski wstecz i w przód działają normalnie.

W następnych częściach rozdziału napiszesz taką aplikację od podstaw. Dzięki temu nie tylko dowiesz się, jak jest zbudowana, lecz także nauczysz się wykorzystywać bibliotekę React Router do tworzenia ciekawych i efektownych aplikacji.



Pierwsze kroki

Najpierw przygotuj projekt. Użyj do tego celu sprawdzonego polecenia `create-react-app`. Otwórz wiersz poleceń, przejdź do folderu, w którym chcesz utworzyć projekt, i wpisz następujące polecenie:

```
create-react-app react_spa
```

Powstanie folder `react_spa`, a w nim nowy projekt. Przejdź do tego folderu:

```
cd react_spa
```

W poprzednich przykładach na tym etapie usuwałeś niektóre pliki, aby móc zbudować aplikację od podstaw. Teraz też to zrobisz, ale najpierw musisz zainstalować bibliotekę React Router. W tym celu wpisz następujące polecenie:

```
npm i react-router-dom --save
```

Polecenie to pobiera pliki biblioteki React Router i rejestruje je w pliku `package.json`, aby Twоя aplikacja wiedziała o ich istnieniu. Pomyślowe, prawda?

Teraz pora na wyczyszczenie projektu i rozpoczęcie budowy aplikacji od zera. Usuń całą zawartość folderów `public` i `src`. Następnie utwórz w folderze `src` plik `index.html`, który będzie punktem wejścia do aplikacji, i wpisz w nim poniższy kod:

```
<!DOCTYPE html>
<html>

<head>
<meta charset="utf-8">
<meta name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

```
<title>Aplikacja jednostronowa</title>
</head>

<body>
  <div id="root"></div>
</body>
</html>
```

Przyjrzyj się przez chwilę powyższemu kodowi. Nic nie powinno Cię w nim zaskoczyć. Przygotuj zatem wejściowy kod JavaScript. W folderze *src* utwórz plik *index.js* i wpisz w nim następujący kod:

```
import React from "react";
import ReactDOM from "react-dom";
import Main from "./Main";

ReactDOM.render(
  <Main/>,
  document.getElementById("root")
);
```

W tym kodzie wywoływana jest metoda `ReactDOM.render()` wyświetlająca komponent `Main`, który jeszcze nie istnieje. Będzie to główny komponent wykorzystujący bibliotekę React Router. W następnej części rozdziału dowieś się, jak go utworzyć.

Tworzenie jednostronnej aplikacji

Pod względem procesu tworzenia aplikacja jednostronna niczym nie różni się od aplikacji, które budowałeś wcześniej. Składa się ona z nadrzędnego komponentu i kilku komponentów podległych zawierających poszczególne widoki. Biblioteka React Router w magiczny sposób określa, które komponenty mają być wyświetlane, a które ukryte. Dzięki temu aplikacja wygląda naturalnie i działa płynnie, w pasku adresu pojawiają się odpowiednie adresy URL, a przyciski wstecz i w przód działają normalnie.

Wyświetlenie początkowej ramki

Część aplikacji jednostronnej zawsze pozostaje niezmienna. Jest to tzw. **ramka aplikacji**, którą jest niewidoczny element HTML spełniający rolę kontenera. Kontener ten może zawierać całą wyświetlającą treść albo tylko niektóre jej fragmenty, na przykład nagłówek, stopkę lub pasek nawigacyjny. W naszym przypadku ramka będzie zawierała nagłówek nawigacyjny oraz pusty obszar, w którym będzie umieszczana treść.

Utwórz teraz w folderze *src* plik *Main.js* i wpisz w nim poniższy kod:

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

class Main extends Component {
  render() {
    return (
      <div>
        <h1>Prosta aplikacja jednostronowa</h1>
        <ul className="header">
          <li><a href="/">Strona główna</a></li>
          <li><a href="/stuff">0 nas</a></li>
```

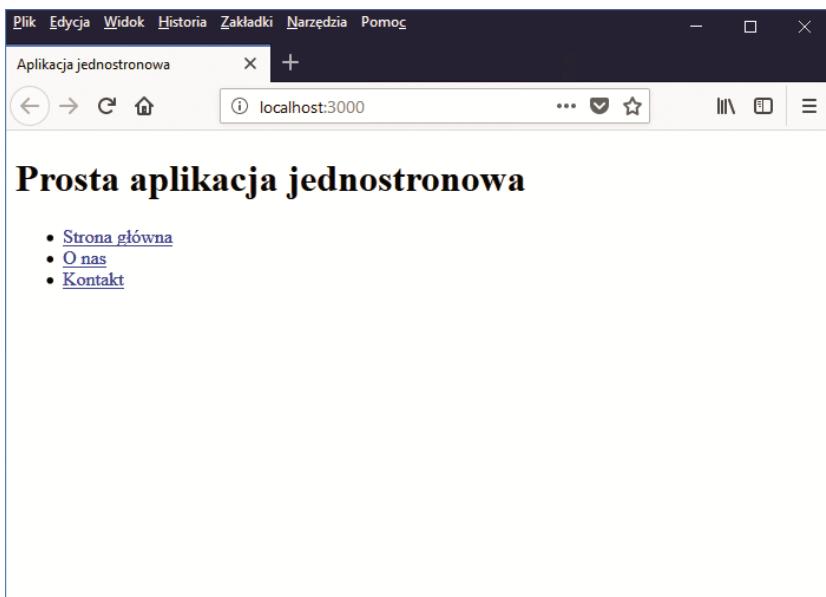
```

        <li><a href="/contact">Kontakt</a></li>
    </ul>
    <div className="content">
        </div>
    </div>
);
}
}

export default Main;

```

Przyjrzyj się przez chwilę temu kodowi. Zdefiniowany jest w nim komponent `Main` generujący kod HTML. To wszystko. Aby zobaczyć efekt swojej dotychczasowej pracy, wpisz polecenie `npm start` i sprawdź, co widać w przeglądarce. Powinna pojawić się strona bez żadnego stylu, zawierająca tytuł i krótką listę wypunktowaną, jak na poniższym rysunku:



Strona nie wygląda imponująco, ale na razie to nam wystarczy. Stylizacją zajmiemy się później. W tej chwili ważne jest to, że nigdzie, absolutnie nigdzie nie jest wykorzystywana biblioteka React Router!

Utworzenie widoków z treścią

Aplikacja będzie składała się z trzech widoków. Treść dla każdego z nich będzie zawarta w prostych komponentach zwracających kody JSX. Przygotuj od razu te komponenty, aby mieć to za sobą. Najpierw w folderze `src` utwórz plik `Home.js` i wpisz w nim następujący kod:

```

import React, { Component } from "react";

class Home extends Component {
    render() {

```

```

        return (
          <div>
            <h2>Witaj!</h2>
            <p>Cras facilisis urna ornare ex volutpat, et
            convallis erat elementum. Ut aliquam, ipsum vitae
            gravida suscipit, metus dui bibendum est, eget rhoncus nibh
            metus nec massa. Maecenas hendrerit laoreet augue
            nec molestie. Cum sociis natoque penatibus et magnis
            dis parturient montes, nascetur ridiculus mus.</p>
            <p>Duis a turpis sed lacus dapibus elementum sed eu lectus.</p>
          </div>
        );
      }
    }

export default Home;

```

Następnie w tym samym folderze utwórz plik *Stuff.js* z następującą zawartością:

```

import React, { Component } from "react";

class Stuff extends Component {
  render() {
    return (
      <div>
        <h2>Opis</h2>
        <p>Mauris sem velit, vehicula eget sodales vitae,
        rhoncus eget sapien:</p>
        <ol>
          <li>Nulla pulvinar diam</li>
          <li>Facilisis bibendum</li>
          <li>Vestibulum vulputate</li>
          <li>Eget erat</li>
          <li>Id porttitor</li>
        </ol>
      </div>
    );
  }
}

export default Stuff;

```

Pozostał jeszcze jeden widok. Utwórz plik *Contact.js* i umieść w nim poniższą treść:

```

import React, { Component } from "react";

class Contact extends Component {
  render() {
    return (
      <div>
        <h2>Masz pytania?</h2>
        <p>Najlepiej wyślij wiadomość przez
          nasze <a href="http://forum.kirupa.com">forum</a>.
        </p>
      </div>
    );
  }
}

export default Contact;

```

To była ostatnia część treści. Gdy przyjrzyisz się powyższym komponentom, stwierdzisz, że nie mogą być prostsze. Zwracają zwyczajny kod JSX.

Zapisz teraz utworzone pliki. Za chwilę dowiesz się, jak ich użyć.

Biblioteka React Router

Utworzyleś komponent `Main`, stanowiący ramkę aplikacji, oraz komponenty `Home`, `Stuff` i `Contact`, reprezentujące trzy widoki z treścią. Trzeba to wszystko połączyć ze sobą w jedną aplikację. W tym momencie pojawia się biblioteka React Router. Aby ją wykorzystać, otwórz plik `Main.js` i wpisz w nim wyróżnione niżej instrukcje import:

```
import React, { Component } from "react";
import {
  Route,
  NavLink,
  HashRouter
} from "react-router-dom";
import Home from "./Home";
import Stuff from "./Stuff";
import Contact from "./Contact";
```

Pierwsza instrukcja importuje komponenty `Route`, `NavLink` i `HashRouter` z zainstalowanego wcześniej pakietu `react-router-dom`. Następne instrukcje importują komponenty `Home`, `Stuff` i `Contact`, które będą potrzebne podczas ładowania treści.

Za pomocą biblioteki React Router definiuje się tzw. **region kierowania** składający się z dwóch części:

1. odnośników nawigacyjnych,
2. kontenera, w którym umieszczana jest treść.

Pomiędzy adresami URL, które określisz w odnośnikach, a treścią, która będzie ładowana, istnieje ścisła zależność. Nie można jej właściwie zrozumieć bez uprzedniego utworzenia kodu i zaimplementowania opisanego wcześniej algorytmu.

Najpierw musisz zdefiniować region kierowania. W tym celu wewnątrz metody `render()` komponentu `Main` wpisz wyróżnione niżej wiersze:

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Prosta aplikacja jednostronna</h1>
          <ul className="header">
            <li><a href="/">Strona główna</a></li>
            <li><a href="/stuff">Opis</a></li>
            <li><a href="/contact">Kontakt</a></li>
          </ul>
          <div className="content">
            </div>
          </div>
        </HashRouter>
    );
  }
}
```

Komponent HashRouter umożliwia poruszanie się po aplikacji i jest wykorzystywany przez przeglądarkę do tworzenia historii odwiedzanych stron.

Teraz zdefiniuj odnośniki nawigacyjne. Wcześniej utworzyłeś listę elementów a. Każdy element zastąp bardziej wyspecjalizowanym komponentem NavLink. W tym celu wprowadź poniższe zmiany:

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Prosta aplikacja jednostronowa</h1>
          <ul className="header">
            <li><NavLink to="/">Strona główna</NavLink></li>
            <li><NavLink to="/stuff">Opis</NavLink></li>
            <li><NavLink to="/contact">Kontakt</NavLink></li>
          </ul>
          <div className="content">

            </div>
          </div>
        </HashRouter>
    );
  }
}
```

Zwróć uwagę na adresy URL w odnośnikach (zdefiniowane za pomocą atrybutów to). Adresy te spełniają rolę identyfikatorów treści przeznaczonych do załadowania. Do powiązania każdego adresu URL z treścią użyjesz komponentu Route. Wprowadź w kodzie wyróżnione niżej zmiany:

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Prosta aplikacja jednostronowa</h1>
          <ul className="header">
            <li><NavLink to="/">Strona główna</NavLink></li>
            <li><NavLink to="/stuff">Opis</NavLink></li>
            <li><NavLink to="/contact">Kontakt</NavLink></li>
          </ul>
          <div className="content">
            <Route path="/" component={Home}/>
            <Route path="/stuff" component={Stuff}/>
            <Route path="/contact" component={Contact}/>
          </div>
        </div>
      </HashRouter>
    );
  }
}
```

Jak widzisz, komponent Route zawiera właściwość path, której wartością jest ścieżka. Gdy ścieżka ta zostanie aktywowana, wyświetli się komponent wskazany we właściwości component. Na przykład: gdy klikniesz odnośnik Opis (z którym powiązany jest adres /stuff określony we właściwości to komponentu NavLink), aktywowana zostanie ścieżka, której właściwość path ma wartość /stuff. W efekcie wyświetli się komponent Stuff.

Sprawdź sam, jak to działa. Otwórz w tym celu przeglądarkę. Aplikacja powinna aktualizować się na bieżąco. Jeżeli tak nie jest, uruchom ją poleceniem `npm start`. Klikaj poszczególne odnośniki i obserwuj, jak zmienia się wyświetlana treść. Coś tu jest jednak nie tak, prawda? Zawartość głównej strony jest wyświetlana zawsze, nawet po kliknięciu odnośników *Opis* lub *Kontakt*:



Mamy więc problem. Zajmiemy się nim i kilkoma innymi porządkowymi sprawami w następnej części rozdziału — po tym, jak bardziej zagłębimy się w bibliotekę React Router.

Kilka poprawek

Utworzyłeś prawie gotową aplikację. Niemal cały region kierowania zamknąłeś w komponencie HashRouter, a za pomocą komponentów NavLink i Route rozdzieliłeś odnośniki i miejsca, w których umieszczana jest treść. Jednak aplikacja *prawie gotowa* i *w pełni gotowa* to dwie różne rzeczy. W następnych częściach rozdziału zlikwidujemy te różnice.

Korekta procesu kierowania

Poprzednią część rozdziału zakończyliśmy stwierdzeniem, że proces kierowania zawiera błąd, ponieważ zawartość reprezentowana przez komponent Home jest zawsze widoczna. Jest tak, ponieważ z tym komponentem jest powiązany adres /. Adresy komponentów Stuff i Contact również w swoich adresach zawierają ukośniki. Oznacza to, że komponent Home jest aktywowany zawsze, niezależnie od otwieranego adresu. Rozwiążanie tego problemu jest proste. W elemencie Route wywołującym komponent Home trzeba umieścić właściwość exact, jak niżej:

```
<div className="content">
  <Route exact path="/" component={Home}/>
  <Route path="/stuff" component={Stuff}/>
```

```
<Route path="/contact" component={Contact}/>
</div>
```

Właściwość ta powoduje, że dany komponent jest aktywowany tylko wtedy, gdy adres treści dokładnie odpowiada wartości właściwości path. Jeżeli teraz otworzysz aplikację, okaże się, że treść jest ładowana poprawnie, tzn. komponent Home jest aktywowany tylko po kliknięciu odnośnika *Opis*.

Dodanie stylu

W tej chwili aplikacja jest kompletnie pozbawiona stylu. To również łatwo można poprawić. W folderze *src* utwórz plik *index.css* i wpisz w nim poniższe reguły:

```
body {
  background-color: #FFCC00;
  padding: 20px;
  margin: 0;
}
h1, h2, p, ul, li {
  font-family: sans-serif;
}
ul.header li {
  display: inline;
  list-style-type: none;
  margin: 0;
}
ul.header {
  background-color: #111;
  padding: 0;
}
ul.header li a {
  color: #FFF;
  font-weight: bold;
  text-decoration: none;
  padding: 20px;
  display: inline-block;
}
.content {
  background-color: #FFF;
  padding: 20px;
}
.content h2 {
  padding: 0;
  margin: 0;
}
.content li {
  margin-bottom: 10px;
}
```

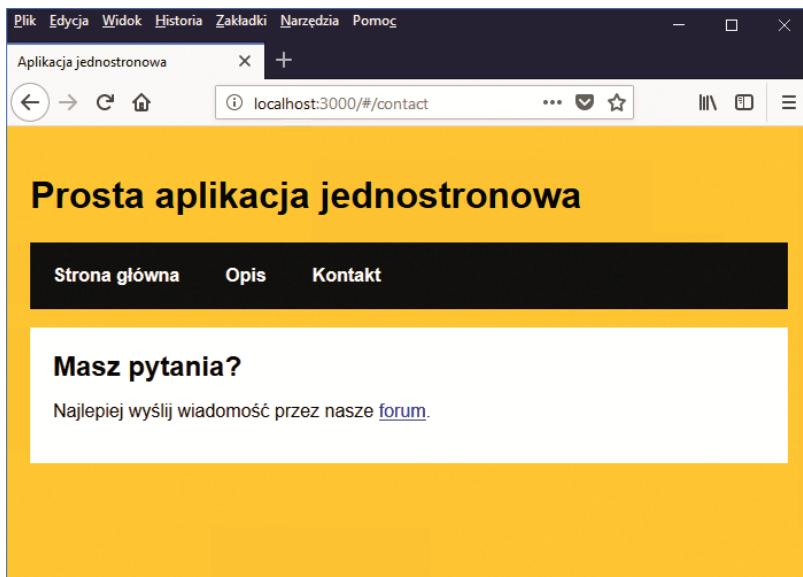
Teraz w kodzie aplikacji umieść odwołanie do powyższego pliku. W tym celu na początku pliku *index.js* wpisz następującą instrukcję import:

```
import React from "react";
import ReactDOM from "react-dom";
import Main from "./Main";
import "./index.css";

ReactDOM.render(
```

```
<Main/>,
document.getElementById("root")
);
```

Zapisz wszystkie pliki, jeżeli jeszcze tego nie zrobileś. Jeśli teraz otworzysz aplikację, okaże się, że wygląda jak niżej:



Prawie koniec! Pozostało jeszcze kilka drobiazgów.

Wyróżnienie aktywnego odnośnika

W tej chwili trudno jest stwierdzić, któremu odnośnikowi odpowiada wyświetlana treść. Przydałyby się jakieś wizualne wskazówki. Twórcy biblioteki React Router pomyśleli o tym. Gdy zostanie kliknięty odnośnik, automatycznie przypisywana mu jest klasa `active`. Na przykład kliknięty odnośnik `Opis` wygląda tak:

```
<a aria-current="true" href="#/stuff" class="active">Opis</a>
```

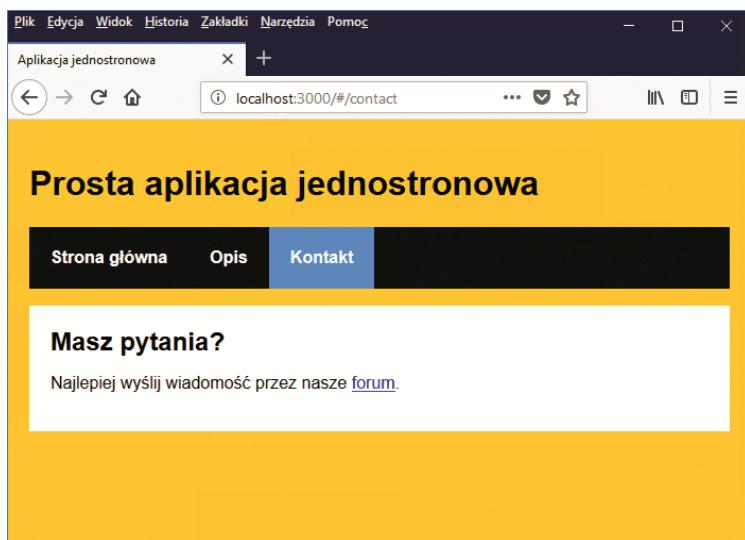
Wystarczy zatem jedynie zdefiniować odpowiedni styl CSS wyróżniający element, któremu zostanie przypisana klasa `active`. Otwórz więc plik `index.css` i na końcu dopisz następującą regułę:

```
.active {
  background-color: #0099FF;
}
```

Zapisz zmiany, wróć do przeglądarki i kliknij każdy z odnośników. Zauważysz, że odnośnik, którego treść się pojawia, jest wyróżniany niebieskim tłem. Cały czas jest również wyróżniony odnośnik `Strona główna`. Tak nie może być. Poprawka jest prosta: wystarczy, że w elemencie `NavLink` wywołującym komponent `Home` umieścisz właściwość `exact`, jak niżej:

```
<li><NavLink exact to="/">Strona główna</NavLink></li>
<li><NavLink to="/stuff">Opis</NavLink></li>
<li><NavLink to="/contact">Kontakt</NavLink></li>
```

Wróć do przeglądarki. Teraz odnośnik *Strona główna* jest wyróżniany innym kolorem tylko wtedy, gdy wyświetlana jest główna treść:



W tym momencie zakończyłeś kodowanie jednostronnej aplikacji opartej na bibliotece React Router! Hurra!

Podsumowanie

W tym rozdziale poznaleś wiele ciekawych funkcjonalności biblioteki React Router ułatwiających tworzenie aplikacji jednostronowych. Nie znaczy to, że są to wszystkie ciekawe cechy biblioteki. Aplikacja, którą utworzyłeś, jest bardzo prosta, ponieważ wykorzystuje podstawową funkcjonalność kierowania zapytaniami. Biblioteka React Router ma znacznie większe możliwości (na przykład zawiera różne odmiany wykorzystanych tu komponentów), więc gdybyś chciał zbudować bardziej zaawansowaną aplikację, poświęć jedno popołudnie na zapoznanie się z pełną dokumentacją i przykładami dostępnymi na stronie <https://github.com/reactjs/react-router>.

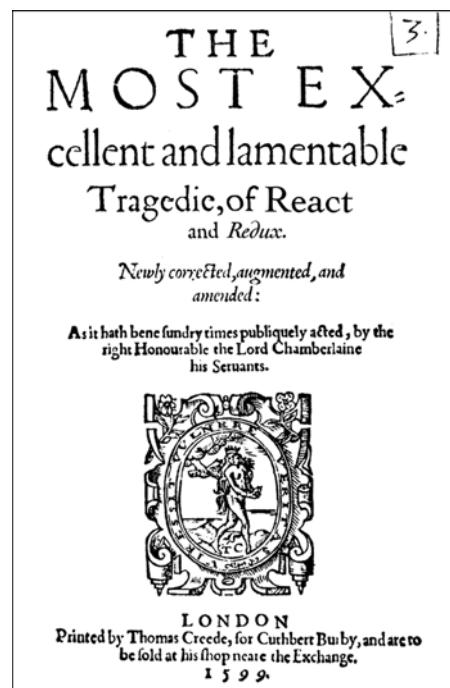
Jeżeli napotkasz problemy, pyтай!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniemi, pyтай śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

19

Wprowadzenie do biblioteki Redux

Romansem wszelkich czasów nie jest historia Romea i Julii. Nie jest nim również żadna opowieść znana z książek lub filmów, tylko historia biblioteki React i tajemniczego marudera z dalekiego kraju, zwanego Redux.

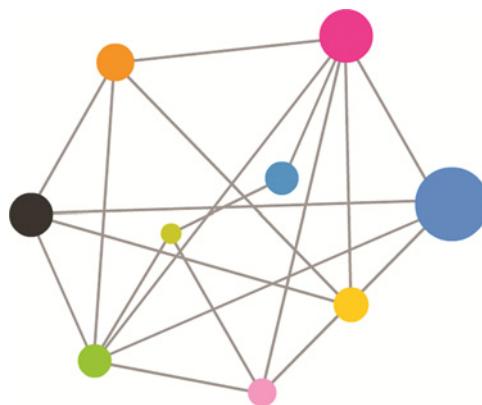


W tej chwili wiesz już dużo o bibliotece React: jak działa i dlaczego wykonuje pewne operacje tak, a nie inaczej. Jednak do tej pory nic nie pisałem na temat biblioteki Redux. Postaram się to naprawić, zanim opiszę, dlaczego biblioteki React i Redux są ze sobą tak ściśle związane. W kolejnych częściach rozdziału szczegółowo opowiem, czym jest Redux.

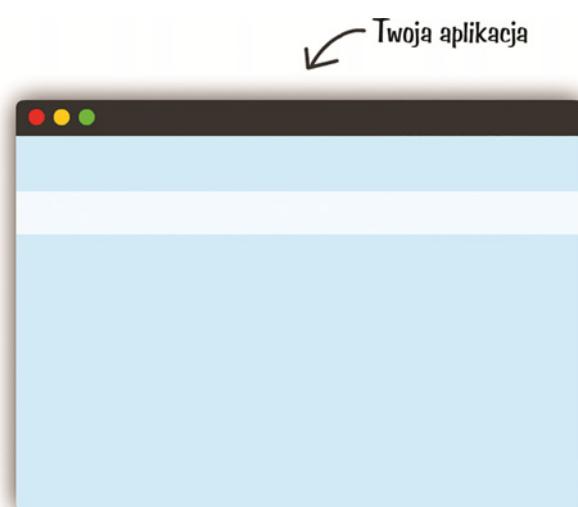
Czym jest Redux?

Podstawowy problem, który na pewno do tej pory jasno sobie uświadomiłeś, brzmi: *zarządzanie stanem aplikacji i utrzymanie jego spójności jest nie lada wyzwaniem*. Pojawienie się takich bibliotek jak Redux jest próbą rozwiązywania tego problemu. Jeżeli spojrzy się na aplikację z szerszej perspektywy, nie tylko na jej interfejs graficzny, okaże się, że zarządzanie stanem aplikacji jest generalnie skomplikowanym zadaniem. Typowa aplikacja składa się z wielu warstw, a każda warstwa jest uzależniona od pewnych danych, niezbędnych do wykonywania właściwych jej zadań.

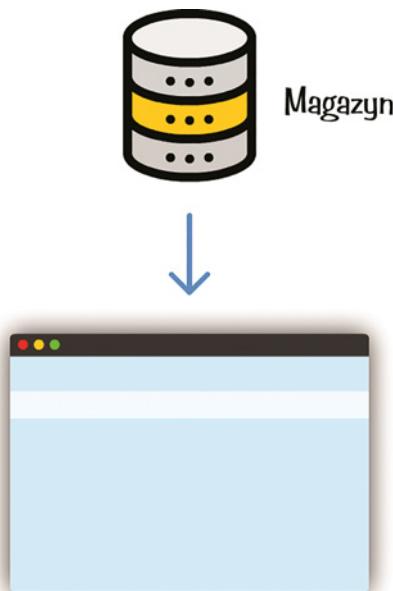
Schemat zależności pomiędzy funkcjonalnościami aplikacji a jej stanem jest często bardzo skomplikowany:



Aby rozwiązać bardziej ogólny problem zarządzania stanem aplikacji, powstała biblioteka Redux. Najprościej jest zrozumieć jej działanie, analizując różne pojęcia, których ta biblioteka dotyczy. Pierwszym z nich jest sama aplikacja:

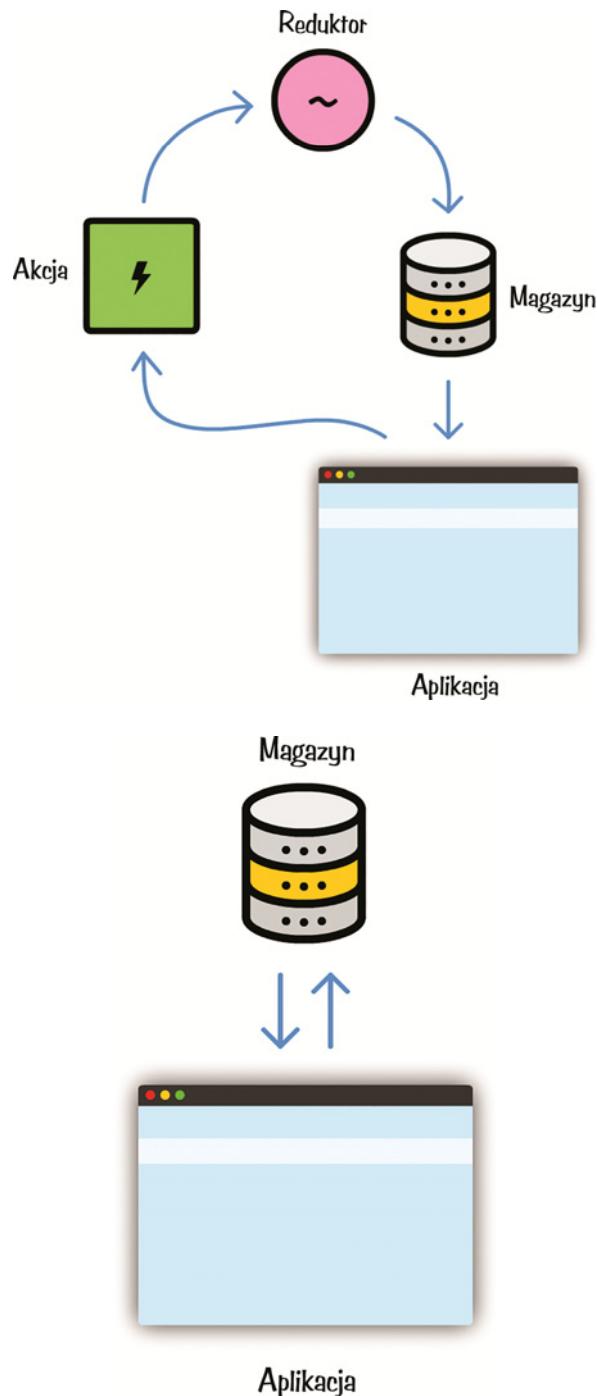


Aplikacja nie musi być niczym wyjątkowym. Może być napisana w czystym języku JavaScript, wykorzystywać biblioteki React, Angular, Vue albo dowolną inną bibliotekę lub platformę, jaka właśnie się pojawiła. Dla biblioteki Redux nie ma znaczenia sposób, w jaki aplikacja została utworzona. Zadaniem tej biblioteki jest dbanie w pewien magiczny sposób o stan aplikacji i przechowywanie go. Wszystkie informacje o stanie aplikacji biblioteka Redux zapisuje w jednym miejscu zwanym **magazynem**:



Magazyn ma to do siebie, że dane odczytuje się z niego bardzo łatwo, natomiast umieszczanie w nim informacji to zupełnie inna sprawa. Aby zapisać w magazynie nowy stan (lub zmodyfikować istniejący), trzeba stosować *akcje* opisujące dane, które trzeba zmienić, oraz *reduktor* określający końcowy stan, który jest wynikiem wykonania akcji. Gdy umieści się wszystkie powyższe pojęcia na jednym rysunku, powstanie coś takiego — patrz pierwszy rysunek na następnej stronie.

Pierwszy rysunek nie zawiera kilku elementów, ale i tak dobrze odzwierciedla to, co dzieje się w aplikacji, gdy w magazynie aktualizowany jest jej stan. Patrząc na ten schemat, zapewne zastanawiasz się, dlaczego jest w nim tyle okrężnych i pośrednich dróg. Dlaczego nie można zmieniać stanu bezpośrednio w magazynie? (patrz drugi rysunek na następnej stronie)



Powodem jest skalowalność. Synchronizacja stanu i działania nawet prostej aplikacji jest zadaniem trudnym, a zupełnie niemożliwym w przypadku złożonej aplikacji, gdy poszczególne jej części usiłują uzyskać dostęp do magazynu i modyfikować stan. Okrężne podejście, stosowane w bibliotece Redux, można łatwo zaimplementować zarówno w prostej, jak i złożonej aplikacji. Dzięki bibliotece Redux zarządzanie stanem aplikacji jest nie tylko łatwe, lecz także *przewidywalne*. Twórcy tej biblioteki, Dan Abramov i Andrew Clark, definiują przewidywalność w następujący sposób:

1. **Stan całej aplikacji jest zapisany w jednym miejscu.** Dzięki temu nie trzeba szukać w wielu magazynach danych części stanu przeznaczonej do modyfikacji. Dodatkowo nie trzeba się zajmować synchronizacją danych.
2. **Stan można jedynie odczytywać.** Modyfikować można go tylko za pomocą akcji. Zgodnie z poprzednim rysunkiem biblioteka Redux wymaga, aby poszczególne części aplikacji nie miały bezpośredniego dostępu do magazynu i nie modyfikowały zapisanego w nim stanu. Mogą to robić jedynie za pomocą akcji.
3. **Końcowy stan musi być określony.** Mówiąc prościej: nie można go modyfikować ani mutować. Końcowy stan określa się za pomocą reduktora.

Powyższe trzy zasady brzmią zapewne dość abstrakcyjnie, jednak za chwilę poznasz je w praktyce, gdy utworzysz kod wykorzystujący bibliotekę Redux.

Prosta aplikacja wykorzystująca bibliotekę Redux

Teraz wykorzystasz informacje zawarte w diagramach i opisach z poprzednich części rozdziału do utworzenia aplikacji demonstrującej działanie biblioteki Redux. Będzie to naprawdę prosta, konsolowa aplikacja bez interfejsu graficznego, zapisująca i wyświetlająca listę Twoich ulubionych kolorów. Aplikacja będzie służyła do dodawania i usuwania kolorów. Tylko tyle.

Ten przykład może wydawać się krokiem wstecznym w porównaniu z aplikacjami z interfejsami graficznymi, które utworzyłeś wcześniej. Jednak w tym przypadku chodzi o wykorzystanie teoretycznej wiedzy o bibliotece Redux do napisania kilku istotnych wierszy kodu. Celem jest po prostu zrozumienie idei tej biblioteki. Skomplikujemy ten przykład później, gdy połączymy bibliotekę z interfejsem graficznym.

Czas na bibliotekę Redux

Najpierw utwórz nowy dokument HTML i umieść w nim odwołanie do biblioteki Redux. Tym razem nie wykorzystasz polecenia `create-react-app` ani żadnego systemu kompilacyjnego. Przygotujesz prosty plik HTML, który otworzysz w przeglądarce. Utwórz zatem w swoim ulubionym edytorze plik `favoriteColors.html` i wpisz w nim poniższy kod:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Ulubione kolory</title>
  <script src="https://unpkg.com/redux@latest/dist/redux.js"></script>
```

```
</head>
<body>
  <script>
    </script>
</body>
</html>
```

Jak widać, jest to dokument definiujący jedynie prostą strukturę pustej strony. Zawiera on odwołanie do zewnętrznej biblioteki Redux. Na etapie eksperymentów można tak robić. W gotowych aplikacjach, takich jak te, które wykorzystują bibliotekę React, trzeba stosować lepsze rozwiązania. Zajmiemy się nimi później. Teraz bezpośrednie odwołanie do biblioteki jest całkowicie do przyjęcia.

Światło, kamera, akcja!

Gdy odwołanie do biblioteki Redux jest gotowe, czas na zdefiniowanie akcji. Jak już wiesz, akcje stanowią jedyny kanał komunikacyjny z magazynem. Nasza aplikacja będzie służyła do dodawania i usuwania kolorów, więc akcje będą opisywały te operacje w sposób zrozumiały dla magazynu.

Wewnątrz znacznika script wpisz poniższy kod:

```
function addColor(value) {
  return {
    type: "ADD",
    color: value
  };
}

function removeColor(value) {
  return {
    type: "REMOVE",
    color: value
  };
}
```

Zdefiniowane są tu dwie funkcje: `addColor()` (dodaj kolor) i `removeColor()` (usuń kolor). Każda z nich ma jeden argument i zwraca obiekt reprezentujący akcję. Poniżej zostały wyróżnione wiersze obiektu zwracanego przez funkcję `addColor()`:

```
function addColor(value) {
  return {
    type: "ADD",
    color: value
  };
}
```

Programista przy definiowaniu akcji ma dużą swobodę. Każdy obiekt musi mieć właściwość `type` opisującą operację, która ma zostać wykonana. Poza tą właściwością można w obiekcie akcji umieszczać dowolne inne informacje. Nas interesuje dodawanie i usuwanie kolorów, więc nasza akcja zawiera właściwość `color`, której wartością jest nazwa podanego przez nas koloru.

Wróćmy do funkcji `addColor()` i `removeColor()`. Każda wykonuje tylko jedną czynność: zwraca akcję. W terminologii biblioteki Redux istnieje specjalne określenie tego rodzaju funkcji: *kreatorzy akcji*.

Reduktor

Akcje definiują operacje, które mają zostać wykonane. Natomiast reduktor określa, jak jest zdefiniowany stan i co się stanie po wykonaniu akcji. Reduktor można porównać do pośrednika pomiędzy magazynem a zewnętrznym światem, który:

1. daje dostęp do oryginalnego stanu w magazynie;
2. pozwala sprawdzać aktualnie wykonywaną akcję;
3. umożliwia zapisywanie nowego stanu w magazynie.

Wszystkie te funkcje poznasz w praktyce, gdy zdefiniujesz reduktora umożliwiającego dodawanie i usuwanie kolorów. Teraz poniżej kreatorów akcji wpisz poniższy kod:

```
function favoriteColors(state, action) {  
  if (state === undefined) {  
    state = [];  
  }  
  
  if (action.type === "ADD") {  
    return state.concat(action.color);  
  } else if (action.type === "REMOVE") {  
    return state.filter(function(item) {  
      return item !== action.color;  
    });  
  } else {  
    return state;  
  }  
}
```

Zatrzymaj się na chwilę i sprawdź, co ten kod robi. Najpierw sprawdza, czy istnieje stan, którym można manipulować:

```
function favoriteColors(state, action) {  
  if (state === undefined) {  
    state = [];  
  }  
  
  if (action.type === "ADD") {  
    return state.concat(action.color);  
  } else if (action.type === "REMOVE") {  
    return state.filter(function(item) {  
      return item !== action.color;  
    });  
  } else {  
    return state;  
  }  
}
```

Jeżeli obiekt stanu nie istnieje (co ma miejsce na przykład po pierwszym uruchomieniu aplikacji), jest definiowany jako pusta tablica. Stan może być dowolną strukturą danych, ale tablica jest najbardziej odpowiednia dla operacji, które będą wykonywane.

Pozostała część kodu jest odpowiedzialna za obsługę akcji. Zwróć uwagę, że dzięki argumentowi state reduktor ma pełny dostęp do akcji, tj. nie tylko do jej właściwości type, ale wszystkich innych właściwości zdefiniowanych wcześniej w jej kodzie.

Jeżeli akcja jest typu ADD, do tablicy state jest dodawany kolor (określony we właściwości color). Jeśli akcja jest typu REMOVE, reduktor zwraca nową tablicę, która nie zawiera wskazanego koloru. Natomiast jeżeli akcja jest nieznana, wtedy reduktor zwraca niezmieniony obiekt stanu:

```
function favoriteColors(state, action) {
  if (state === undefined) {
    state = [];
  }

  if (action.type === "ADD") {
    return state.concat(action.color);
  } else if (action.type === "REMOVE") {
    return state.filter(function(item) {
      return item !== action.color;
    });
  } else {
    return state;
  }
}
```

Proste, prawda? Pamiętaj o kilku ważnych zasadach obowiązujących podczas korzystania z biblioteki Redux (opisanych w dokumentacji pod adresem <https://redux.js.org/docs/basics/Reducers.html>).

Reduktor nie może:

1. modyfikować wartości argumentów;
2. wykonywać dodatkowych operacji, na przykład korzystać z interfejsu API, ani kierować zapytaniami;
3. wywoływać złożonych funkcji, na przykład Date.now() lub Math.random().

Dla zadanych wartości argumentów reduktor musi generować nowy stan, a następnie go zwracać. Żadnych niespodzianek, efektów specjalnych ani odwołań do interfejsu API. Tylko generowanie stanu.

Powysze zasady obowiązują w naszym kodzie. Do zapisania nowego koloru w tablicy wykorzystana została metoda concat(), która zwraca nową tablicę zawierającą zarówno dotychczasowe wartości, jak i nową wartość. Metoda push() dałaby ten sam efekt, ale naruszałaby zasadę zakazu modyfikacji istniejącego stanu. Ta zasada jest spełniona w przypadku usuwania koloru. Zastosowana została metoda filter(), która zwraca nową tablicę niezawierającą wskazanej wartości.

Pamiętaj również o tym, co przekazał mi kiedyś Mark Erikson (@acemarke): biblioteka Redux nie zawiera żadnego mechanizmu zapobiegającego modyfikacjom stanu i wykonywaniu innych, niewłaściwych operacji. Twórcy biblioteki określili jedynie kilka zaleceń dotyczących korzystania z biblioteki. Od Ciebie zależy, czy będziesz je stosował w praktyce.

Magazyn

Pozostało już tylko powiązanie akcji i reduktora z magazynem. Magazyn trzeba przede wszystkim utworzyć. Poniżej funkcji favoriteColors() wpisz poniższy kod:

```
var store = Redux.createStore(favoriteColors);
```

Magazyn jest tutaj tworzony za pomocą metody createStore(), której argumentem jest zdefiniowana wcześniej funkcja reduktora favoriteColors(). W ten sposób został zamknięty obieg danych w procesie zarządzania stanem aplikacji za pomocą biblioteki Redux. Mamy magazyn, reduktor i akcje zlecające reduktorowi operacje do wykonania.

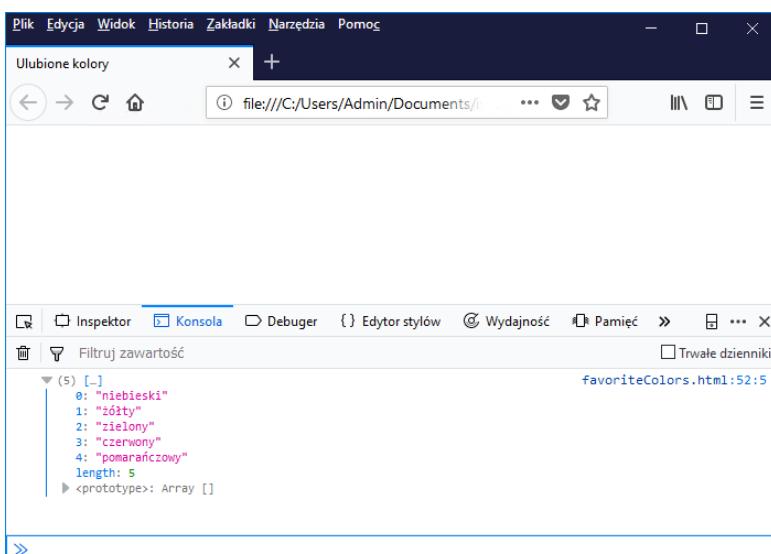
Aby sprawdzić, czy wszystko działa poprawnie, dodaj do magazynu kilka kolorów, a następnie je usuń. W tym celu wykorzystaj metodę dispatch() obiektu store, której argumentem jest obiekt akcji. Wpisz następujący kod:

```
store.dispatch(addColor("niebieski"));
store.dispatch(addColor("żółty"));
store.dispatch(addColor("zielony"));
store.dispatch(addColor("czarny"));
store.dispatch(addColor("szary"));
store.dispatch(addColor("pomarańczowy"));
store.dispatch(removeColor("szary"));
```

Każde wywołanie metody dispatch() powoduje wysłanie akcji do reduktora. Reduktor odbiera akcję i wykonuje odpowiednią operację określającą nowy stan aplikacji. Aby sprawdzić bieżącą zawartość magazynu, wpisz na końcu kodu następujący wiersz:

```
console.log(store.getState());
```

Metoda getState(), jak wskazuje jej nazwa, zwraca obiekt stanu. Otwórz teraz plik w przeglądarce, a następnie otwórz narzędzia dla programistów. W zakładce Konsola powinna pojawić się lista kolorów dodanych do stanu aplikacji:



Prawie skończyliśmy. Pozostała jeszcze jedna ważna rzecz. W praktyce potrzebne są powiadomienia wysyłane automatycznie za każdym razem, gdy zmieni się stan aplikacji. Bardzo ułatwiają one aktualizowanie interfejsu użytkownika i wykonywanie innych operacji, gdy coś się zmieni w magazynie. Aby uruchomić ten mechanizm, trzeba użyć metody `subscribe()` i określić tzw. funkcję nasłuchującą, która będzie wywoływana po każdorazowej zmianie zawartości magazynu. W tym celu poniżej definicji obiektu `store` wpisz wyróżnione wiersze:

```
var store = Redux.createStore(favoriteColors);
store.subscribe(render);
function render() {
  console.log(store.getState());
}
```

Po wprowadzeniu powyższej zmiany otwórz ponownie aplikację. Tym razem po każdym wywołaniu metody `dispatch()` (z akcją modyfikującą magazyn) będzie wywoływana funkcja `render()`. Super!

Podsumowanie

W tym rozdziale zrobiliśmy krótką wycieczkę po bibliotece Redux i oferowanych przez nią funkcjonalnościach. Nie tylko dowiedziałeś się, dlaczego biblioteka ta jest tak przydatna w zarządzaniu stanem aplikacji, lecz także wykorzystałeś ją w kodzie. Nie utworzyłeś jedynie bardziej praktycznej aplikacji. Biblioteka Redux jest na tyle elastyczna, że można ją stosować z dowolnymi platformami przeznaczonymi do tworzenia interfejsów graficznych. Każda z tych platform współpracuje z biblioteką we właściwy sobie magiczny sposób. Wybraną przez nas platformą jest oczywiście React! W następnym rozdziale dowiesz się, jak można wykorzystać obie biblioteki.

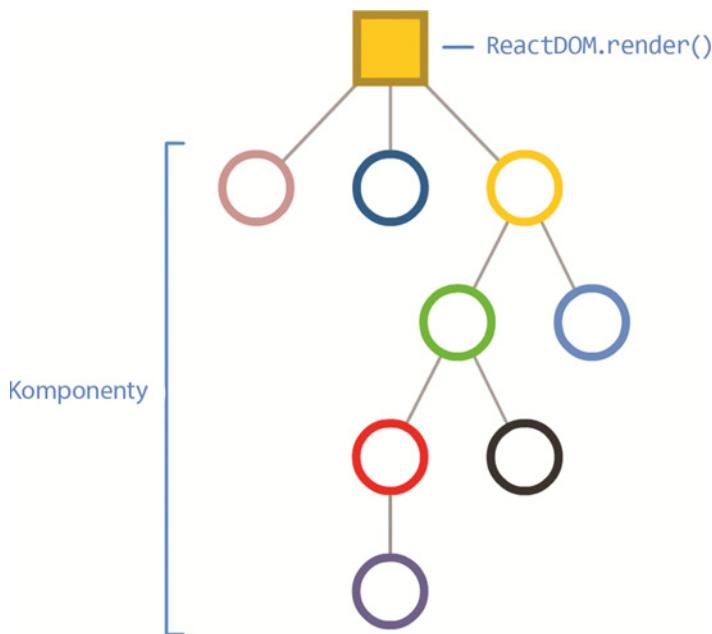
Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniemi, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

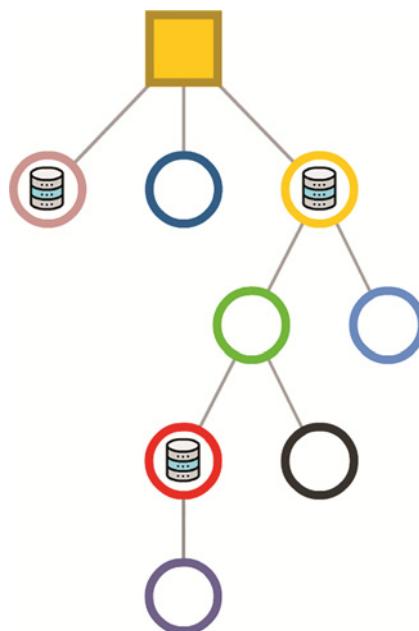
20

Stosowanie bibliotek React i Redux

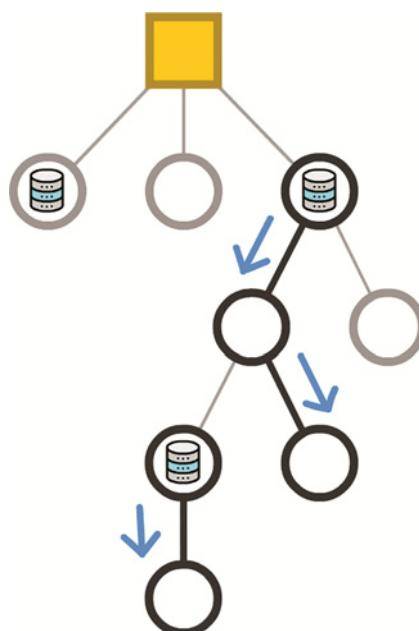
Teraz, gdy masz lepsze wyobrażenie o tym, jak działa biblioteka Redux, możemy wrócić do tematu poruszonego w poprzednim rozdziale. *Dlaczego tę bibliotekę tak często stosuje się razem z biblioteką React?* Aby odpowiedzieć sobie na to pytanie, przyjrzyjmy się hierarchii komponentów w przykładowej aplikacji:



W tej chwili nie jest istotne, co robi ta aplikacja. Szczególem, który dodamy do tego schematu, są komponenty odpowiedzialne za zarządzanie stanem aplikacji i przekazywanie innym komponentom części informacji w postaci właściwości:

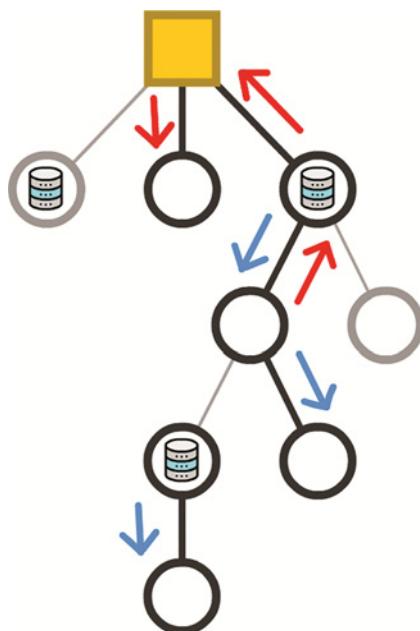


W idealnej sytuacji wymagane dane przepływają w kierunku od komponentu nadzorowanego do podporządkowanego:



Niestety, taki scenariusz jest realny tylko w prostych aplikacjach. Typowa aplikacja generuje, przetwarza i przesyła mnóstwo informacji o swoim stanie. Jeden komponent może inicjować zmianę stanu, na którą musi reagować inny komponent znajdujący się w innym miejscu hierarchii.

Aby informacja dotarła do docelowego komponentu, musi być przekazywana za pomocą właściwości w dół hierarchii (OK), jak również w górę (nie!):



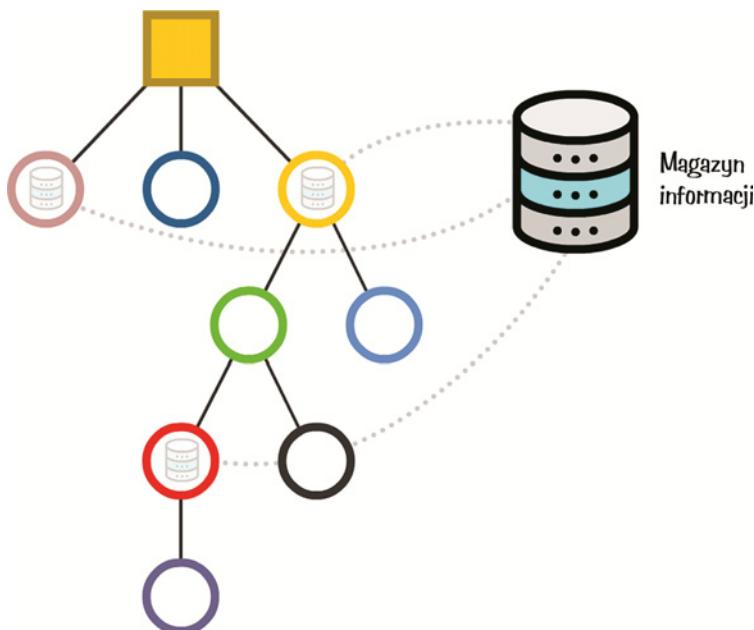
Przekazywanie informacji (wartości zmiennych, funkcji, procedur obsługi zdarzenia) w kierunku od komponentu podległego do nadzorowanego i dalej jest często popełnianym błędem.

W tym momencie można sformułować kilka problemów wynikających z nieuporządkowanego przepływu danych przez komponenty:

- 1. Trudno jest zarządzać zależnościami stosowanymi w kodzie.** Jeżeli sieć zależności jest zagmatwana, dane przepływają w sposób, którego chcieliśmy uniknąć. Celem biblioteki React jest przeciwdziałanie temu efektowi.
- 2. Za każdym razem, gdy zmieni się stan aplikacji lub zostanie przesłana właściwość, wszystkie zaangażowane w ten proces komponenty muszą zostać ponownie wyświetcone.** Jest to konieczne, aby interfejs aplikacji odzwierciedlał jej aktualny stan. Jednak — jak pisałem w jednym z poprzednich rozdziałów — wiele komponentów jest niepotrzebnie wielokrotnie wyświetlanych, ponieważ jedynie przekazują dane z komponentu nadzorowanego do podległego bez wykonywania dodatkowych operacji. Ten efekt można zminimalizować, odpowiednio zmieniając właściwość `shouldComponentUpdate` lub korzystając z komponentu `PureComponent`, jednak w miarę rozwoju aplikacji stosowanie obu sposobów staje się coraz bardziej kłopotliwe.

3. Hierarchia komponentów odzwierciedla strukturę interfejsu użytkownika, a nie danych. Sposób, w jaki są ułożone i zagnieździone komponenty, ma ułatwiać dzielenie interfejsu użytkownika na mniejsze, łatwiejsze w zarządzaniu części. To jest właściwe podejście. Jednak komponent, który inicjuje zmianę stanu, często nie jest komponentem nadzorującym dla komponentu, który ma na tę zmianę reagować. W takim przypadku, jak pisałem wcześniej, dane muszą przemierzać skomplikowaną drogę, często przez wiele komponentów.

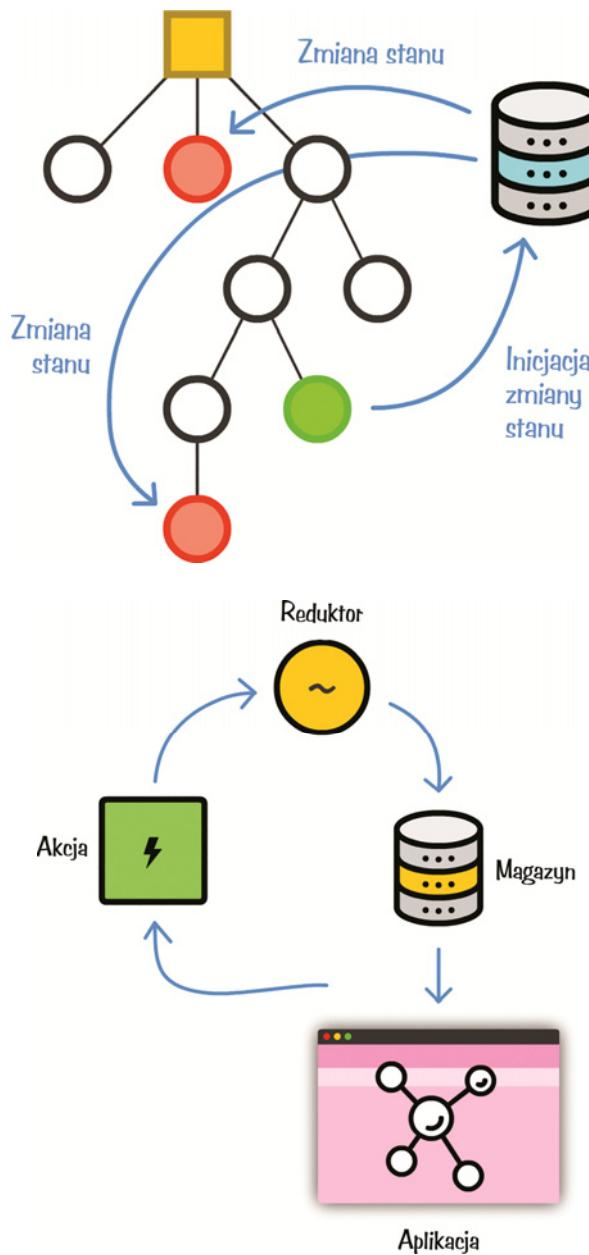
Wyjściem z tej sytuacji jest zastosowanie biblioteki Redux. Nie rozwiązuje ona wprawdzie wszystkich powyższych problemów, ale znacznie je upraszcza. Dzięki niej informacje o stanie aplikacji nie są rozproszone w wielu komponentach, tylko są zgromadzone w jednym miejscu:



Takie podejście rozwiązuje kilka problemów. Jeżeli dane są wykorzystywane przez osobne części aplikacji, nie muszą przepływać w górę i w dół hierarchii komponentów — patrz pierwszy rysunek na następnej stronie.

Dzięki temu w zainicjowanie zmiany i faktyczną zmianę stanu aplikacji angażowane są tylko te komponenty, których ta zmiana dotyczy. Dane (z ewentualnymi zmianami) przepływają bezpośrednio do miejsca przeznaczenia bez niepotrzebnego wielokrotnego wyświetlania komponentów. Pomyślone, prawda?

Przejdzmy teraz na wyższy poziom. Z architektonicznego punktu widzenia, opisanego w poprzednim rozdziale, przepływ danych w aplikacji wykorzystującej bibliotekę Redux wygląda następująco — patrz drugi rysunek na następnej stronie.



Oprócz magazynu w aplikacji wykorzystywane są akcje, reduktor i inne elementy wymagane przez bibliotekę Redux. Rzec w tym, że aplikacja może być zbudowana w oparciu o bibliotekę React. Na tego typu aplikacji (i współpracy obu bibliotek) skupimy teraz naszą uwagę.

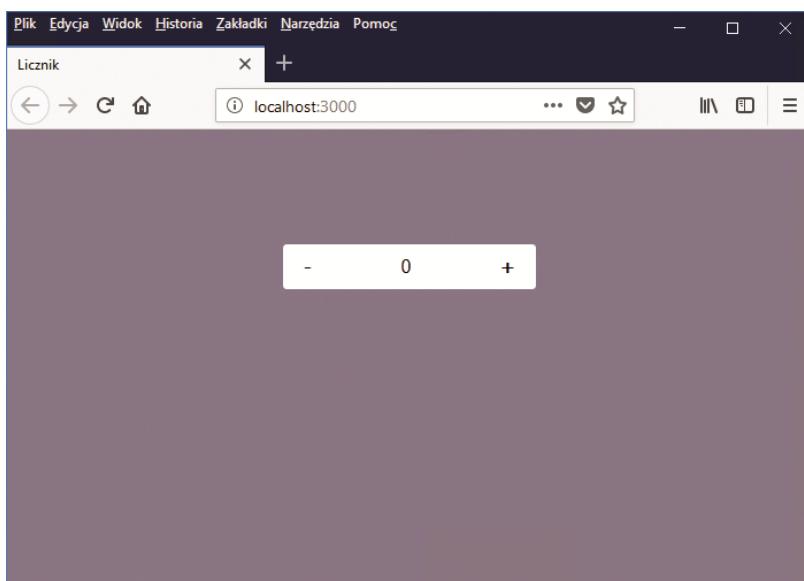
Do dzieła!

Biblioteki React i Redux oraz zarządzanie stanem aplikacji

Podłączenie funkcjonalności biblioteki Redux do aplikacji opartej na bibliotece React jest bardzo proste. Polega na wywołaniu kilku metod interfejsu API biblioteki Redux. Należy zrobić to w dwóch krokach:

1. Udzielić aplikacji dostępu do magazynu.
2. Powiązać kreatory akcji, funkcje zlecające i obiekt stanu w formie właściwości z danymi zapisanymi w magazynie, wymaganymi przez dany komponent.

Aby dowiedzieć się, jak powyższe dwa kroki realizuje się w praktyce, utworzysz prostą aplikację *Licznik*, przedstawioną na rysunku 20.1.

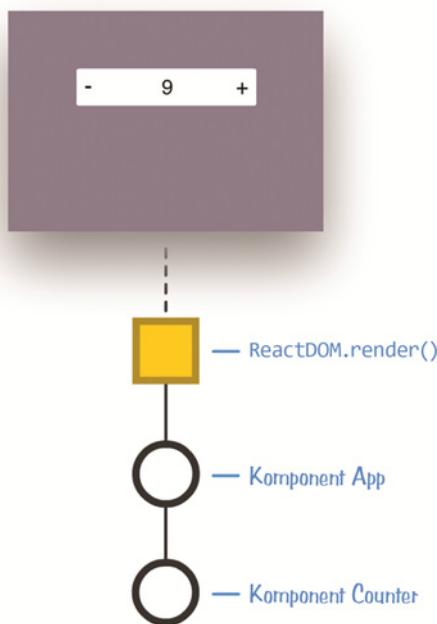


Rysunek 20.1. Przykładowa aplikacja Licznik, którą utworzysz

Aplikacja będzie składała się z przycisków ze znakami dodawania i odejmowania, służących do zwiększania i zmniejszania wartości licznika. Tylko tyle. Nic więcej w aplikacji nie będzie się działo. Będzie to stopień funkcjonalności i złożoności wystarczający do zrozumienia zasady łączenia bibliotek React i Redux.

Wspólne funkcjonalności bibliotek React i Redux

Zazwyczaj budowanie aplikacji rozpoczynałeś od kopiowania i wklejania niezbędnych kodów HTML, CSS i JavaScript. Zaraz to wszystko zrobisz, ale wcześniej musisz poznać strukturę aplikacji. Pomijając dane i funkcje zarządzania stanem, aplikacja będzie składała się z zaledwie dwóch komponentów (patrz rysunek 20.2).



Rysunek 20.2. Aktualna struktura aplikacji

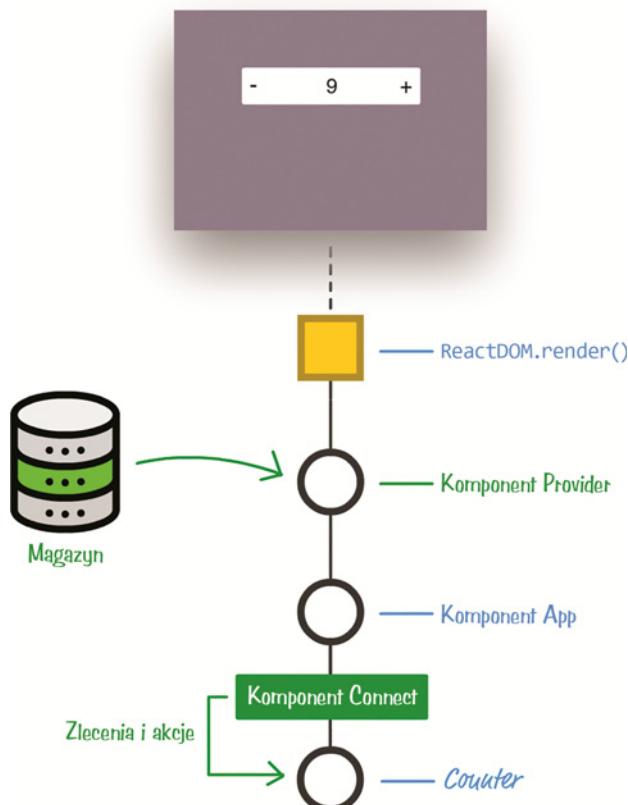
Mamy tu komponenty App i Counter (licznik). Jak widać, aplikacja w takiej postaci nie jest na tyle skomplikowana, aby się nią specjalnie zajmować. By zaimplementować zarządzanie jej stanem w zwykły, znany Ci już sposób, należałoby w komponencie Counter zdefiniować obiekt stanu oraz zmienną, której wartość byłaby zwiększana lub zmniejszana w zależności od klikniętego przycisku.

Jeżeli jednak zastosuje się bibliotekę Redux, struktura komponentów nieco się skomplikuje. Spójrz na rysunek 20.3.

Komponenty ReactDOM.render(), App i Counter są takie same jak poprzednio. Pozostałe komponenty są nowościami wprowadzanymi przez bibliotekę Redux. Jak pisalem wcześniej, zastosowanie tej biblioteki wymaga wykonania dwóch kroków. W tym przypadku kroki te wykonują następujące nowe komponenty:

1. Pierwszy krok, polegający na udostępnieniu magazynu, wykonuje komponent Provider.
2. Drugi krok, polegający na udzieleniu komponentom dostępu do właściwych im zleceń i akcji, wykonuje komponent Connect.

Bardziej obrazowo można powiedzieć, że komponent Provider jest dla aplikacji opartej na bibliotece React bramą do biblioteki Redux. Komponent Provider przechowuje referencję do magazynu, dzięki czemu wszystkie inne komponenty mają do niego dostęp. Jest to możliwe dlatego, że Provider znajduje się na szczytce hierarchii komponentów. Strategiczne położenie komponentu powoduje, że w całej aplikacji można łatwo korzystać z funkcjonalności biblioteki Redux.



Rysunek 20.3. Struktura aplikacji po zastosowaniu w niej biblioteki Redux

Bardziej interesujący jest komponent **Connect**. Nie jest to w pełni funkcjonalny komponent w zwykłym znaczeniu tego słowa. Jest to tzw. *komponent wyższego rzędu* (HOC — ang. *Higher Order Component*, <https://reactjs.org/docs/higher-order-components.html>) i służy do rozszerzania w spójny sposób funkcjonalności istniejących komponentów poprzez ich opakowywanie. Komponent ten można porównać do słowa kluczowego `extends` w języku ES6, służącego do rozszerzania klas. W efekcie (patrz powyższy rysunek) dzięki komponentowi **Connect** komponent **Counter** uzyskuje dostęp do akcji i zleceń niezbędnych do korzystania z magazynu, bez konieczności wykonywania dodatkowego kodu. O wszystko troszczy się komponent wyższego rzędu **Connect**.

Oba komponenty, **Provider** i **Connect**, są ze sobą nierozerwalnie związane, dzięki czemu za pomocą funkcjonalności oferowanych przez bibliotekę **Redux** można bardzo skutecznie zarządzać stanem każdej aplikacji opartej na bibliotece **React**. Gdy zaczniesz tworzyć swoją aplikację, zobacysz, jak w praktyce wyglądają powiązania tych dwóch komponentów.

Przygotowanie

Teraz, gdy znasz już strukturę komponentów i charakterystyczne funkcjonalności biblioteki Redux, które wykorzystasz, czas zacząć budować aplikację. Najpierw za pomocą polecenia `create-react-app reduxcounter`:

```
create-react-app reduxcounter
```

Następnie zainstaluj biblioteki Redux i React-Redux. W tym celu przejdź do folderu `reduxcounter` i w wierszu poleceń wpisz następujące polecenie:

```
npm install redux
```

Polecenie to instaluje bibliotekę Redux, dzięki której w aplikacji będzie można wykorzystywać podstawowe funkcjonalności do zarządzania stanem.

Po pomyślnym zainstalowaniu biblioteki Redux musisz dodać jeszcze inną zależność. Wpisz poniższe polecenie, abyś mógł w aplikacji opartej na bibliotece React wykorzystać całą zawartość biblioteki Redux:

```
npm install react-redux
```

Po zakończeniu instalacji masz już wszystko, co potrzebne do tworzenia aplikacji opartych na bibliotece React i do korzystania z magii biblioteki Redux. Czas zacząć tworzyć aplikację!

Tworzenie aplikacji

Najpierw wyczyść projekt z niepotrzebnych plików. W tym celu usuń całą zawartość folderów `src` i `public`. Następnie w folderze `public` utwórz plik `index.html` i umieść w nim poniższy kod:

```
<!DOCTYPE html>
<html>

<head>
  <title>Licznik</title>
</head>

<body>
  <div id="container">

  </div>
</body>

</html>
```

Jedyną rzeczą, na którą warto zwrócić tutaj uwagę, jest element `div` z atrybutem `id` o wartości `container`.

Teraz napisz kod JavaScript będący punktem wejścia do aplikacji. W folderze `src` utwórz plik `index.js` i wpisz w nim poniższy kod:

```
import React, { Component } from "react";
import ReactDOM from "react-dom";
import { createStore } from "redux";
import { Provider } from "react-redux";
import counter from "./reducer";
import App from "./App";
import "./index.css";
```

```

var destination = document.querySelector("#container");

// Magazyn
var store = createStore(counter);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  destination
);

```

Przyjrzyj się przez chwilę temu, co się w tym kodzie dzieje. Najpierw inicjowany jest magazyn za pomocą znanej Ci już metody `createStore()`, z reduktorem w argumencie. Reduktor jest tu zdefiniowany za pomocą zmiennej `counter`. Jeżeli przyjrzesz się instrukcjom `import`, zauważysz, że zmienna ta jest zdefiniowana w pliku `reducer.js`. Reduktorem zajmiemy się za chwilę.

Utworzony magazyn jest przekazywany w formie właściwości komponentowi `Provider`. To główny komponent aplikacji, za którego pośrednictwem wszystkie pozostałe komponenty będą miały dostęp do magazynu i wymaganych funkcjonalności biblioteki `Redux`:

```

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  destination
);

```

Teraz utwórz reduktor. Wiesz już, że jest on zawarty w zmiennej `counter` i trzeba go zdefiniować w nieistniejącym jeszcze pliku `reducer.js`. Utwórz więc ten plik w folderze `src` i wpisz w nim następujący kod JavaScript:

```

// Reduktor
function counter(state, action) {
  if (state === undefined) {
    return { count: 0 };
  }

  var count = state.count;

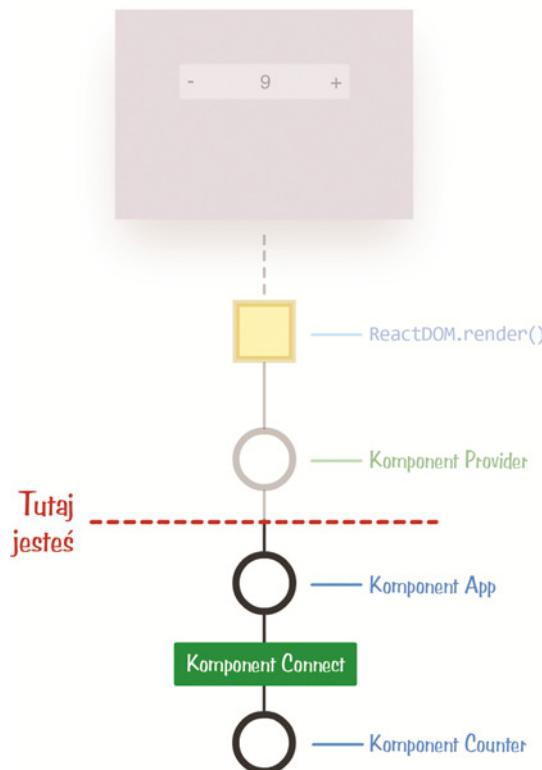
  switch (action.type) {
    case "increase":
      return { count: count + 1 };
    case "decrease":
      return { count: count - 1 };
    default:
      return state;
  }
}

export default counter;

```

Reduktor jest bardzo prosty. Zawiera zmienną `count`, której przypisuje wartość 0, jeżeli stan aplikacji nie jest jeszcze określony. Reduktor obsługuje akcje dwóch rodzajów: `increase` i `decrease`. Jeżeli akcja jest typu `increase`, wartość zmiennej `count` jest zwiększana o 1. W przeciwnym wypadku jest pomniejszana o 1.

W tym momencie znajdujesz się w połowie drogi do utworzenia gotowej aplikacji (patrz rysunek 20.4).



Rysunek 20.4. Tutaj jesteś w drodze do swojej aplikacji!

Teraz możesz zejść jeden poziom niżej i zająć się komponentem App. W folderze *src* utwórz plik *App.js* i wpisz w nim następujący kod:

```
import { connect } from "react-redux";
import Counter from "./Counter";

// Powiązanie stanu z właściwością komponentu
function mapStateToProps(state) {
  return {
    countValue: state.count
  };
}

// Akcje
var increaseAction = { type: "increase" };
var decreaseAction = { type: "decrease" };

// Powiązanie akcji z właściwościami komponentu
function mapDispatchToProps(dispatch) {
  return {
    increaseCount: function() {
      return dispatch(increaseAction);
    },
    decreaseCount: function() {
      return dispatch(decreaseAction);
    }
  };
}
```

```

        }
    };

// Komponent wyższego rzędu
var connectedComponent = connect(
    mapStateToProps,
    mapDispatchToProps
)(Counter);

export default connectedComponent;

```

Przyjrzyj się przez chwilę powyższemu kodowi. Jego głównym przeznaczeniem jest zamienianie wszystkich charakterystycznych dla biblioteki Redux funkcji na coś, co można wykorzystać w bibliotece React. Konkretnie: są to funkcje przekazywane w formie właściwości dwóm funkcjom mapStateToProps() i mapDispatchToProps(). Pierwsza z tych funkcji wygląda następująco:

```

// Powiązanie stanu z właściwością komponentu
function mapStateToProps(state) {
    return {
        countValue: state.count
    };
}

```

Funkcja ta będzie wywoływana za każdym razem, gdy w magazynie zostaną zmienione dane. Zwracanym wynikiem jest obiekt zawierający dane, które będą przekazywane komponentowi w formie właściwości. Są to bardzo proste dane: właściwość countValue, której wartością jest dotychczasowa wartość właściwości count zapisanej w magazynie.

Przekazywanie danych w formie właściwości to tylko część potrzebnych operacji. Następną rzeczą jest udostępnienie komponentowi — również w formie właściwości — kreatorów akcji i samych akcji. To zadanie realizuje poniższy kod:

```

// Akcje
var increaseAction = { type: "increase" };
var decreaseAction = { type: "decrease" };

// Powiązanie akcji z właściwościami komponentu
function mapDispatchToProps(dispatch) {
    return {
        increaseCount: function() {
            return dispatch(increaseAction);
        },
        decreaseCount: function() {
            return dispatch(decreaseAction);
        }
    };
}

```

Naprawdę ciekawy kod ma funkcja mapDispatchToProps(). Zwraca ona obiekt zawierający nazwy dwóch funkcji, które komponent będzie mógł wywoływać, aby zlecić zmianę danych w magazynie. Funkcja increaseCount() zleca wykonanie akcji typu increase, natomiast funkcja decreaseCount() wykonanie akcji typu decrease. Jeżeli spojrzesz na kod reduktora, który wpisałeś wcześniej, zobaczysz, jak powyższe funkcje modyfikują wartość właściwości count zapisanej w magazynie.

Trzeba jeszcze zaimplementować sposób odbierania przez komponent powyższych właściwości. W tym miejscu pojawia się magiczna funkcja connect():

```
var connectedComponent = connect(  
    mapStateToProps,  
    mapDispatchToProps  
) (Counter);
```

Funkcja ta tworzy opisany wcześniej komponent Connect wyższego rzędu. Jej argumentami są funkcje `mapStateToProps()` i `mapDispatchToProps()`, umieszczone dalej w komponencie Counter, który również musisz zdefiniować. Efektem wykonania powyższego kodu jest następujący kod:

```
<Connect>  
  <Counter increaseCount={increaseCount}  
            decreaseCount={decreaseCount}  
            countValue={countValue}/>  
</Connect>
```

Komponent Counter uzyskuje dostęp do funkcji `increaseCount()` i `decreaseCount()` oraz właściwości `countValue`. Jedyną dziwną rzeczą jest tu brak metody `render()` lub jej odpowiednika. Wszystkie operacje automatycznie wykonuje komponent wyższego rzędu zawarty w bibliotece Redux.

Prawie koniec! Teraz pora utworzyć i uruchomić komponent Counter. W folderze `src` utwórz plik `Counter.js` i wpisz w nim poniższy kod:

```
import React, { Component } from "react";  
  
class Counter extends Component {  
  render() {  
    return (  
      <div className="container">  
        <button className="buttons"  
              onClick={this.props.decreaseCount}>-</button>  
        <span>{this.props.countValue}</span>  
        <button className="buttons"  
              onClick={this.props.increaseCount}>+</button>  
      </div>  
    );  
  }  
}  
  
export default Counter;
```

To jest chyba najnudniejszy z komponentów, jakie do tej pory utworzyłeś. Pisałem już, jak komponent `Connect` przesyła właściwości i inne dane w dół hierarchii do komponentu Counter. W tym kodzie widać, jak za pomocą tych właściwości jest wyświetlana wartość zmiennej `counter` i są wywoływane odpowiednie funkcje po kliknięciu przycisków.

Ostatnią rzeczą jest przygotowanie pliku CSS stylizującego aplikację. W folderze `src` utwórz plik `index.css` i umieść w nim poniższe reguły stylów:

```
body {  
  margin: 0;  
  padding: 0;  
  font-family: sans-serif;  
  display: flex;  
  justify-content: center;  
  background-color: #8E7C93;  
}
```

```
.container {  
    background-color: #FFF;  
    margin: 100px;  
    padding: 10px;  
    border-radius: 3px;  
    width: 200px;  
    display: flex;  
    align-items: center;  
    justify-content: space-between;  
}  
  
.buttons {  
    background-color: transparent;  
    border: none;  
    font-size: 16px;  
    font-weight: bold;  
    border-radius: 3px;  
    transition: all .15s ease-in;  
}  
  
.buttons:hover:nth-child(1) {  
    background-color: #F45B69;  
}  
  
.buttons:hover:nth-child(3) {  
    background-color: #C0DFA1;  
}
```

W tym momencie zakończyłeś kodowanie. Zapisz zmiany we wszystkich plikach, jeżeli tego jeszcze nie zrobiłeś. Gdy otworzysz aplikację w przeglądarce (polecienniem `npm start`), przekonasz się, że działa zgodnie z oczekiwaniami.

Podsumowanie

Biblioteka Redux w dużej mierze usuwa ograniczenia będące rzekomymi zaletami biblioteki React. Kilka tych zalet poznałeś w rozdziałach poświęconych przepływowi danych w aplikacji opartej na bibliotece React. Można pokusić się o stwierdzenie, że funkcjonalności biblioteki Redux powinny zostać zintegrowane z biblioteką React i formalnie uznane za jej część. Jednak biblioteka Redux też nie jest idealna. Podobnie jak to się dzieje w przypadku innych projektów programistycznych, jest jednym z wielu narzędzi, które można wykorzystać do osiągnięcia zamierzonego celu. W rzeczywistości nie każda aplikacja przetwarzająca dane wymaga użycia tej biblioteki. Wręcz przeciwnie, czasami może to skutkować niepotrzebnym skomplikowaniem kodu. Dan Abramov, jeden z twórców biblioteki Redux, opublikował doskonąły artykuł (https://medium.com/@dan_abramov/you-might-not-need-redux-be46360f367) opisujący sytuacje, w których nie należy stosować tej biblioteki. Bardzo zachęcam Cię do przeczytania tego artykułu — dzięki niemu uzyskasz pełny obraz biblioteki Redux.

Jeżeli napotkasz problemy, pytaj!

Jeżeli będziesz miał jakiekolwiek pytania albo Twój kod nie będzie działał zgodnie z oczekiwaniami, pytaj śmiało! Wejdź na forum <https://forum.kirupa.com> i korzystaj z pomocy najsympatyczniejszych i najbardziej kompetentnych ludzi w internecie!

Skorowidz

A

- adres
 - IP, 162, 163, 166
 - URL, 208, *Patrz też:* odnośnik głęboki
- AJAX, 160
- animacja, 182
- aplikacja
 - jednostronna, 207, 208, *Patrz:* SPA
 - rama aplikacji, *Patrz:* rama aplikacji
 - tworzenie, 210
 - widok, 211
 - skalowalność, 223
 - stan, 221
 - informacje, 221, *Patrz też:* magazyn
 - zarządzanie, 220, 223, 231, 232, 234
 - zmiana, 228, 231, 232
 - stylizacja, 179
 - tryb programistyczny, 155
 - wielostronna, 207, 208

B

- biblioteka
 - animacyjna Flip Move, 183
 - Babel, 84, 145
 - React, 18, 20, 22, 120, 150, 161, 234
 - generowanie kodu HTML, 49
 - komponent, *Patrz:* komponent
 - konsola przeglądarki, 106
 - ładowanie, 28
 - narzędzia programistyczne, 146
 - system zdarzeń, 115
 - środowisko programistyczne, 145, 146
 - wydajność, 197
 - React Router, 209, 210, 213
 - ReactDOM, 150, 161

C

- React-Redux, 237

- Redux, 219, 220, 223, 232, 234, 237
- błąd TypeError, 96

C

- Create React, 147
- CSS, 31, 47, 51, 87
 - reguła, 49, 186, 187

D

- DOM, 93, 105, 140
- model wirtualny, 19

E

- element
 - animowanie, *Patrz:* animacja
 - body, 30, 31
 - button, *Patrz:* przycisk
 - div, 30, 172
 - form, 172, 173
 - hierarchia, 59
 - input, 172
 - root, 150
 - wizualny, 58, 61

F

- formularz, 173, 184
- obsługa domyślna, 176
- fragment, 86
- funkcja, 36
 - alert, 37
 - anonimowa, 138
 - createElement, 21
 - strzałkowa, 139

H

Higher Order Component, *Patrz:* HOC
HOC, 236

I

instrukcja
 console.log, 200
 export, 151
 import, 151
interfejs
 graficzny aplikacji, 197
 użytkownika, 18, 20, 21, 37
 tworzenie, 172
 Web Animations API, 183
iterator, 104

J

JavaScript, 21
język
 ES6, 139
 JSX, 21, 22, 26, 30, 39, 83
 kod, *Patrz:* kod JSX
 transpilowanie, 27, 84
 wielkość liter, 88
 wyrażenie, 85

K

kierowanie, 208, 215, 218
klasa
 active, 217
 Component, 204
 Events, 113
 KeyboardEvent, 114
 MouseEvent, 114
 PureComponent, 205
 React.Component, 41
 SyntheticEvent, 114, 115
kod JSX, 83, 84, 89, 103, 137, 197
 kompilacja, 145, 146
kommentarz, 87, 88
kompilator Babel, 145
komponent, 35, 36, 39, 47, 61
 App, 150, 239

bezstanowy, 91
Connect, 236
cykl życia, 123, 124
dziecko, 44
hierarchia, 63, 229, 231, 232
identyfikator, 106
IPAddressContainer, 161, 162
konsument, 54
kontenerowy, 167
kontrolujący, 184
łączenie, 71
nadzędny, 68, 73
niekontrolujący, 184
odmontowanie, 131
prezentacyjny, 167
Provider, 235, 236, 238
przekazywanie danych, 180
PureComponent, 204, 205
 wydajność, 205
stanowy, 91, *Patrz też:* stan
styl, 52
tworzenie, 61, 63, 64, 65, 66
właściwość, 43, 68
 CSS transition, 188
 inicjowanie, 78
 key, 106
 przekazywanie, 73, 75, 78, 80, 81, 82
 specyfikacja, 43
 this.props, 43, 68
 wywoływanie, 41
wyższego rzędu, *Patrz:* HOC

konsument komponentu, *Patrz:* komponent
konsument

L

lista, 178

M

magazyn, 221, 234, 238
 akcja, 221, 223, 227
 definiowanie, 224
 tworzenie, 234
reduktor, 221, 223, 226, 227, 238
 definiowanie, 225, 226, 238
 tworzenie, 227

- menu wysuwane, 185, 186, 199
 - tworzenie, 187, 188, 190, 191, 194
 - ukrywanie, 192
 - wyświetlanie, 192
 - metoda
 - bind, 119
 - componentDidMount, 93, 123, 128, 163
 - componentDidUpdate, 123, 130
 - componentWillMount, 123, 128
 - componentWillReceiveProps, 123, 130
 - componentWillUnmount, 123, 125, 131
 - componentWillUpdate, 123, 130
 - concat, 226
 - console.log, 126
 - createStore, 227, 238
 - cyklu życia komponentu, 123, 124, 125
 - Date.now, 175
 - dispatch, 227
 - getState, 227
 - handleMouseDown, 193, 194, 195
 - Math.random, 85
 - preventDefault, 176
 - ReactDOM.render, 197
 - render, 28, 29, 30, 105, 124, 128, 130, 197, 198
 - monitorowanie wywołań, 200
 - optymalizacja wywołań, 199
 - requestAnimationFrame, 183
 - setState, 94, 96, 97
 - shouldComponentUpdate, 123, 129, 130, 203, 204, 205
 - subscribe, 228
 - model
 - DOM, *Patrz: DOM*
 - MVC, *Patrz: MVC*
 - SPA, *Patrz: SPA*
 - wirtualny DOM, *Patrz: DOM model wirtualny*
 - N**
 - Node.js, 147
 - O**
 - obiekt
 - state, 94, 97, 174, *Patrz też: stan*
 - stylizujący, 52, 53, 87
 - this, 119, 120, 181
 - undefined, 119
 - XMLHttpRequest, 160
 - odnośnik głęboki, 208, *Patrz też: adres URL*
 - operator rozciągania, 79, 80, 81, 82
- P**
- plik
 - App.js, 151
 - CSS, 151
 - index.css, 153
 - index.html, 149, 150, 161, 171
 - index.js, 152, 153
 - pole tekstowe, 172, 175
 - polecenie
 - create-react-app, 149
 - ipaddress, 161
 - npm run build, 155
 - npm start, 148, 151, 152
 - yarn start, 148
 - portal, 140
 - procedura obsługi zdarzenia, *Patrz: zdarzenie obsługa*
 - proces uzgadniania, *Patrz: uzgadnianie*
 - program Yarn, 148
 - protokół HTTP, 159
 - przeglądarka
 - dodatek React Developer Tools, 200
 - konsola, 124, 125
 - okno, 186, 187
 - wymiary, 188
 - przycisk, 44, 134, 172, 174, 193
- R**
- ramka aplikacji, 210
 - referencja, 135, 137, 138, 174, 175
 - region kierowania, 213, 215, *Patrz też: kierowanie*
 - definiowane, 213
 - odnośnik nawigacyjny, 213, 214
 - routing, *Patrz: kierowanie*

S

selektor, 50
 single-page application, *Patrz:* SPA
 skrypt, 157
 SPA, 16
 stan, 91
 wartość, 94, 96
 zmiana, 96, 128
 styl kaskadowy, *Patrz:* CSS
 szablon
 EmberJS, 21
 HTML, 17, 21
 Mustache, 17

T

tablica, 80, 104, 175, 178

U

usługa
 ipinfo.io, 163
 registerServiceWorker, 150
 uzgadnianie, 20

W

witryna
 jednostronowa, 15, 16
 wielostronowa, 14
 wyrażenie, 85

Z

zapytanie, 159
 HTTP, 159
 kierowanie, 208
 odpowiedź, 160
 zdarzenie, 109, 110
 click, 180, 192
 nasłuchiwanie, 112, 116, 117, 118, 119, 174
 nazwa, 112
 obsługa, 112, 116, 117, 118, 119, 120, 173,
 176
 onChange, 184
 onClick, 112, 114
 onMouseDown, 193
 submit, 174
 SyntheticEvent, 114, 115
 typ
 KeyboardEvent, 113, 114
 MouseEvent, 113, 114
 właściwość, 113, 115
 shiftKey, 115, 116

PROGRAM PARTNERSKI

— GRUPY HELION —



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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
<http://program-partnerski.helion.pl>

GRUPA
Helion

KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie z dostępem do nowoczesnych narzędzi - videokursów, e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL