



React 16

Framework
dla profesjonalistów

Adam Freeman

Helion 

Apress®

Tytuł oryginału: Pro React 16

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-6246-8

First published in English under the title Pro React 16 by Adam Freeman, edition: 1
Copyright © Adam Freeman, 2019

This edition has been translated and published under licence from APress Media, LLC,
part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the
accuracy of the translation.

Polish edition copyright © 2020 by Helion SA
All rights reserved.

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

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

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

Autor oraz 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/react16>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

	O autorze	17
	O recenzencie technicznym	18
Część I	Rozpoczynanie prac z frameworkiem React	19
Rozdział 1.	Twoja pierwsza aplikacja Reacta	21
	Przygotowanie środowiska programistycznego	21
	Instalacja Node.js	21
	Instalacja pakietu create-react-app	22
	Instalacja Git	23
	Instalacja edytora	23
	Instalacja przeglądarki	23
	Tworzenie projektu	24
	Prezentacja struktury projektu	25
	Dodanie frameworka CSS Bootstrap	26
	Uruchamianie narzędzi dla programistów	27
	Usuwanie treści zastępczej	28
	Wyświetlanie treści dynamicznych	29
	Wyjaśnienie zmian danych stanu	30
	Dodawanie możliwości aplikacji listy zadań	33
	Wyświetlanie zadań do zrobienia	35
	Wprowadzanie dodatkowych komponentów	38
	Stosowanie komponentów podrzędnych	40
	Ostatnie szlify	41
	Zarządzanie prezentacją zakończonych zadań	42
	Trwałe przechowywanie danych	44
	Podsumowanie	47

Rozdział 2. Zrozumieć React	49
Czy powinienem używać Reacta?	50
Aplikacje z komunikacją dwukierunkową	50
Aplikacje jednostronicowe	50
Problem złożoności aplikacji	51
Co muszę wiedzieć?	52
Jak skonfigurować środowisko programistyczne?	52
Jaka jest struktura tej książki?	52
Część 1. Początki stosowania frameworka React	52
Część 2. Praca z frameworkiem React	52
Część 3. Tworzenie złożonych aplikacji Reacta	53
Czy w książce jest dużo przykładów?	53
Skąd pobrać kody źródłowe przykładów?	54
Gdzie szukać informacji o poprawkach?	55
Podsumowanie	55
Rozdział 3. Podstawy HTML, JSX i CSS	57
Przygotowania do prac w tym rozdziale	57
Przygotowanie pliku HTML i komponentu	58
Uruchamianie przykładowej aplikacji	59
Język HTML i elementy DOM	60
Treść elementu	61
Atrybuty	63
Dynamiczne zmienianie elementów HTML	63
Dynamiczne tworzenie elementów przy użyciu klasy Component	65
Stosowanie wyrażeń w elementach Reacta	66
Łączenie wyrażeń i treści statycznych	67
Wykonywanie obliczeń w wyrażeniach	67
Dostęp do właściwości i metod komponentu	69
Stosowanie wyrażeń do ustawiania wartości właściwości	70
Stosowanie wyrażeń do obsługi zdarzeń	71
Prezentacja frameworka Bootstrap	72
Stosowanie klas frameworka Bootstrap	72
Stosowanie frameworka Bootstrap do tworzenia siatek	74
Stosowanie frameworka Bootstrap w tabelach	75
Stosowanie frameworka Bootstrap w formularzach	77
Podsumowanie	78
Rozdział 4. Podstawy JavaScriptu	79
Przygotowania do lektury tego rozdziału	79
Stosowanie instrukcji	82
Definiowanie i stosowanie funkcji	82
Definiowanie funkcji z parametrami	83
Definiowanie funkcji zwracających wyniki	85
Stosowanie funkcji jako argumentów innych funkcji	86
Stosowanie zmiennych i typów	87
Stosowanie typów prostych	89

Stosowanie operatorów JavaScriptu	91
Stosowanie instrukcji warunkowych	92
Porównanie operatorów równości i idyntyczności	93
Jawna konwersja typów	94
Stosowanie tablic	95
Stosowanie literału tablicowego	96
Odczyt i modyfikacja zawartości tablic	96
Wyliczanie zawartości tablic	97
Stosowanie operatora rozproszenia	97
Stosowanie wbudowanych metod tablicowych	98
Stosowanie obiektów	100
Stosowanie literałów obiektowych	101
Stosowanie funkcji jako metod	102
Stosowanie klas	103
Kopiowanie właściwości z jednego obiektu do drugiego	104
Przechwytywanie nazw parametrów z obiektów	106
Tworzenie i stosowanie modułów JavaScript	107
Tworzenie i użycie modułu JavaScript	107
Eksportowanie z modułów możliwości nazwanych	109
Definiowanie wielu możliwości nazwanych w jednym module	110
Obietnice języka JavaScript	112
Wyjaśnienie problemu operacji asynchronicznych	112
Stosowanie obietnic	113
Upraszczenie kodu asynchronicznego	114
Podsumowanie	115
Rozdział 5. SportsStore — prawdziwa aplikacja Reacta	117
Przygotowanie projektu	118
Instalacja dodatkowych pakietów NPM	118
Dodanie do projektu arkusza stylów CSS	120
Przygotowanie usługi internetowej	121
Uruchomienie przykładowej aplikacji	123
Tworzenie magazynu danych	123
Tworzenie akcji magazynu danych oraz kreatorów akcji	124
Tworzenie możliwości funkcjonalnych sklepu	126
Tworzenie komponentów produktu i kategorii	127
Połączenie magazynu danych z mechanizmem trasowania	130
Dodanie komponentu Shop do aplikacji	132
Poprawa przycisków wyboru kategorii	133
Dodawanie koszyka	135
Rozbudowa magazynu danych	135
Tworzenie komponentu CartSummary	137
Dodawanie komponentu szczegółów koszyka	141
Dodanie koszyka do konfiguracji trasowania	143
Podsumowanie	145

Rozdział 6. SportsStore — REST i kasa	147
Przygotowania do prac w tym rozdziale	147
Korzystanie z internetowej usługi typu RESTful	147
Tworzenie pliku konfiguracyjnego	149
Tworzenie źródła danych	149
Rozszerzanie możliwości magazynu danych	150
Aktualizacja kreatorów akcji	151
Podział danych na strony	151
Wyjaśnienie wsparcia dla stronicowania w usłudze internetowej	153
Zmiana żądania HTTP oraz akcji	155
Tworzenie komponentu wczytującego dane	156
Aktualizacja komponentu konektora sklepu	157
Aktualizacja przycisku kategorii Wszystkie	159
Tworzenie kontrolki stronicowania	160
Dodanie obsługi składania zamówienia	165
Rozbudowa usługi typu RESTful i źródła danych	166
Tworzenie formularza zamówienia	168
Uproszczenie komponentu konektora sklepu	175
Podsumowanie	176
Rozdział 7. SportsStore — administracja	177
Przygotowania do lektury tego rozdziału	177
Uruchamianie aplikacji	178
Tworzenie usługi GraphQL	179
Definiowanie schematu GraphQL-a	179
Definiowanie resolverów GraphQL-a	180
Aktualizacja serwera	182
Tworzenie narzędzi administracyjnych do zarządzania zamówieniami	184
Definiowanie tabeli zamówień	185
Zdefiniowanie komponentu konektora	186
Konfiguracja klienta GraphQL-a	189
Konfigurowanie mutacji	191
Tworzenie narzędzi zarządzania produktami	192
Połączenie komponentu tabeli produktów	194
Tworzenie komponentów do edycji	197
Aktualizacja konfiguracji trasowania	199
Podsumowanie	202
Rozdział 8. SportsStore — uwierzytelnianie i wdrażanie	203
Przygotowania do prac w tym rozdziale	203
Dodanie uwierzytelniania do żądań GraphQL-a	206
Przedstawienie systemu uwierzytelniania	207
Tworzenie kontekstu uwierzytelniania	208
Tworzenie formularza uwierzytelniającego	211
Zabezpieczanie narzędzi administracyjnych	212
Dodanie odnośników do narzędzi administracyjnych	213

Przygotowanie aplikacji do wdrożenia	214
Umożliwienie leniwego wczytywania narzędzi administracyjnych	214
Tworzenie pliku danych	216
Konfiguracja adresów URL żądań	217
Budowanie aplikacji	217
Tworzenie serwera aplikacji	217
Testowanie produkcyjnej wersji aplikacji i serwera	218
Umieszczanie aplikacji w kontenerze	219
Instalowanie Dockera	219
Przygotowanie aplikacji	220
Tworzenie kontenera Dockera	220
Uruchamianie aplikacji	221
Podsumowanie	222
Część II Praca z Reactem	223
Rozdział 9. Prezentacja projektów Reacta	225
Przygotowania do prac w tym rozdziale	226
Opis struktury projektów Reacta	227
Katalog kodów źródłowych	229
Katalog pakietów	230
Stosowanie narzędzi programistycznych Reacta	233
Proces kompilacji i przekształcania	234
Serwer HTTP do prac programistycznych	238
Treści statyczne	239
Wyświetlanie błędów	242
Linter	245
Konfiguracja narzędzi programistycznych	248
Debugowanie aplikacji Reacta	249
Badanie stanu aplikacji	251
Stosowanie debugera przeglądarki	252
Podsumowanie	254
Rozdział 10. Komponenty i właściwości props	255
Przygotowania do prac w tym rozdziale	256
Komponenty	258
Renderowanie treści HTML	258
Renderowanie innych komponentów	260
Właściwości props	263
Definiowanie właściwości props w komponencie nadrzędnym	263
Odbieranie właściwości props w komponencie podrzędnym	265
Łączenie kodu JavaScript i właściwości props w celu renderowania treści	266
Selektywne renderowanie treści	266
Renderowanie tablic	267
Renderowanie wielu elementów	271
Brak renderowanych treści	273
Próba zmiany wartości właściwości props	274

Stosowanie funkcyjnych właściwości props	275
Wywoływanie funkcyjnych właściwości props z argumentami	277
Przekazywanie właściwości props do komponentów podrzędnych	280
Określanie domyślnych wartości właściwości props	283
Sprawdzanie typów wartości właściwości props	284
Podsumowanie	287
Rozdział 11. Komponenty ze stanem	289
Przygotowania do prac w tym rozdziale	290
Różne typy komponentów	291
Komponenty bezstanowe	292
Komponenty ze stanem	292
Tworzenie komponentów ze stanem	293
Klasa komponentu	294
Instrukcja import	294
Metoda render	294
Właściwości props komponentów ze stanem	295
Dodawanie danych stanu	295
Odczytywanie danych stanu	297
Modyfikacja danych stanu	297
Unikanie problemów z modyfikowaniem danych stanu	299
Definiowanie komponentów ze stanem przy użyciu hooków	304
Podnoszenie danych stanu	306
Dalsze podnoszenie danych stanu	309
Definiowanie typów i wartości domyślnych właściwości props	311
Podsumowanie	314
Rozdział 12. Stosowanie zdarzeń	315
Przygotowania do prac w tym rozdziale	316
Przedstawienie zdarzeń	318
Wywoływanie metody w celu obsługi zdarzenia	319
Pobieranie obiektu zdarzenia	323
Wywoływanie metod obsługi zdarzeń z niestandardowymi argumentami	328
Zapobieganie domyślnej obsłudze zdarzeń	330
Zarządzanie propagacją zdarzeń	332
Faza elementu docelowego i faza propagacji w górę	332
Faza przechwytywania	336
Określanie fazy zdarzenia	337
Zatrzymywanie propagacji zdarzeń	340
Podsumowanie	341
Rozdział 13. Rekoncyliacja i cykl życia	343
Przygotowania do prac w tym rozdziale	344
Tworzenie przykładowych komponentów	345
Wyjaśnienie sposobu renderowania treści	347
Wyjaśnienie procesu aktualizacji	349
Wyjaśnienie procesu rekoncyliacji	351
Rekoncyliacja list	354

Jawne wyzwalanie procesu rekonylacji	355
Cykl życia komponentów	357
Faza montowania	358
Faza aktualizacji	361
Faza odmontowywania	362
Hook efektów	363
Stosowanie zaawansowanych metod cyklu życia	366
Unikanie niepotrzebnych aktualizacji komponentu	366
Ustawianie danych stanu na podstawie właściwości props	369
Podsumowanie	371
Rozdział 14. Konstruowanie aplikacji	373
Przygotowania do prac w tym rozdziale	374
Tworzenie przykładowych komponentów	375
Podstawowe zależności pomiędzy komponentami	377
Stosowanie właściwości props children	377
Operacje na właściwości props children	379
Tworzenie komponentów wyspecjalizowanych	382
Tworzenie komponentów wyższego rzędu	385
Tworzenie komponentów wyższego rzędu ze stanem	388
Łączenie komponentów wyższego rzędu	390
Stosowanie renderującej właściwości props	392
Stosowanie renderującej właściwości props z argumentem	394
Stosowanie kontekstu do przechowywania danych globalnych	396
Definiowanie kontekstu	399
Tworzenie konsumentów kontekstu	400
Tworzenie dostawcy kontekstu	401
Modyfikowanie wartości danych kontekstu w konsumencie	403
Stosowanie uproszczonego API konsumentów kontekstu	406
Definiowanie granic błędów	408
Tworzenie komponentu granicy błędów	409
Podsumowanie	412
Rozdział 15. Formularze i walidacja	413
Przygotowania do prac w tym rozdziale	414
Tworzenie przykładowych komponentów	415
Uruchamianie narzędzi programistycznych	416
Stosowanie elementów formularzy	417
Stosowanie elementów select	419
Stosowanie przycisków opcji	421
Stosowanie pól wyboru	423
Użycie pól wyboru do zapisywania wartości w tablicy	424
Stosowanie wielowierszowych pól tekstowych	426
Walidacja danych z formularzy	427
Definiowanie reguł walidacji	428
Tworzenie komponentu kontenera	430
Wyświetlanie komunikatów o błędach	432
Zastosowanie walidacji	432

Walidacja innych elementów i typów danych	434
Przeprowadzanie całościowej walidacji formularza	439
Podsumowanie	443
Rozdział 16. Referencje i portale	445
Przygotowania do prac w tym rozdziale	446
Tworzenie referencji	450
Użycie referencji do tworzenia niekontrolowanych komponentów formularzy	452
Tworzenie referencji przy użyciu funkcji zwrotnych	454
Walidacja niekontrolowanych komponentów formularzy	457
Referencje a cykl życia	461
Stosowanie referencji z innymi bibliotekami lub frameworkami	467
Dostęp do zawartości komponentów podrzędnych	470
Stosowanie przekazywania referencji	472
Stosowanie portali	473
Podsumowanie	476
Rozdział 17. Testy jednostkowe	477
Przygotowania do prac w tym rozdziale	478
Tworzenie komponentów	480
Uruchamianie przykładowej aplikacji	481
Uruchamianie zastępczego testu jednostkowego	482
Testowanie komponentów z wykorzystaniem renderowania płytkego	484
Testowanie komponentów z użyciem pełnego renderowania	488
Testowanie z użyciem właściwości props, stanu, metod i zdarzeń	490
Testowanie efektów działania metod	491
Testowanie efektów zdarzeń	491
Testowanie interakcji pomiędzy komponentami	492
Podsumowanie	494
Część III Tworzenie kompletnych aplikacji	495
Rozdział 18. Tworzenie kompletnych aplikacji	497
Tworzenie przykładowego projektu	498
Uruchamianie narzędzi programistycznych	499
Tworzenie przykładowej aplikacji	499
Implementacja możliwości funkcjonalnych związanych z produktami	500
Implementacja możliwości funkcjonalnych związanych z dostawcami	504
Dokończanie aplikacji	508
Wyjaśnienie ograniczeń przedstawionej aplikacji	511
Podsumowanie	511
Rozdział 19. Stosowanie magazynu danych Redux	513
Przygotowania do prac w tym rozdziale	514
Tworzenie magazynu danych	515
Definiowanie typów danych	516
Definiowanie początkowych danych	516
Definiowanie typów akcji danych modelu	517

Definiowanie kreatorów akcji danych modelu	517
Definiowanie reduktora	518
Tworzenie magazynu danych	520
Stosowanie magazynu danych w aplikacji Reacta	521
Użycie magazynu danych w komponencie najwyższego poziomu	521
Podłączanie danych produktów	522
Podłączanie danych dostawców	524
Rozszerzanie magazynu danych	527
Dodawanie stanu do magazynu danych	527
Definiowanie typów i kreatorów akcji dla danych stanu	528
Definiowanie reduktora danych stanu	529
Dodawanie nowych opcji obsługi danych do aplikacji	529
Podłączanie komponentów Reacta do danych stanu w magazynie	531
Rozsyłanie wielu akcji	535
Wyjaśnienie potrzeby użycia referencji	537
Podsumowanie	539
Rozdział 20. Stosowanie API magazynu danych	541
Przygotowania do prac w tym rozdziale	542
Stosowanie API magazynu danych Redux	542
Pobieranie stanu magazynu danych	543
Obserwowanie zmian w magazynie danych	546
Rozsyłanie akcji	547
Tworzenie komponentu konektora	549
Rozszerzanie możliwości reduktorów	552
Stosowanie komponentów warstwy pośredniej magazynu danych	555
Rozszerzanie magazynu danych	558
Zastosowanie funkcji rozszerzenia	560
Stosowanie API pakietu React-Redux	562
Zaawansowane możliwości metody connect	562
Podsumowanie	568
Rozdział 21. Trasowanie adresów URL	569
Przygotowania do prac w tym rozdziale	570
Rozpoczynanie korzystania z mechanizmu trasowania	571
Wprowadzenie do użycia komponentu Link	573
Wprowadzenie do użycia komponentu Route	573
Reagowanie na nawigację	574
Wybieranie komponentów i treści	574
Dopasowywanie adresów URL	576
Dopasowywanie pojedynczej trasy	581
Użycie przekierowań jako trasy awaryjnej	583
Generowanie odnośników nawigacyjnych	585
Wskazywanie aktywnej trasy	587
Wybór i konfiguracja mechanizmu trasowania	589
Stosowanie komponentu HashRouter	590
Podsumowanie	591

Rozdział 22. Zaawansowane zagadnienia trasowania adresów URL	593
Przygotowania do prac w tym rozdziale	594
Tworzenie komponentów świadomych trasowania	595
Prezentacja właściwości props match	596
Prezentacja właściwości props location	598
Stosowanie parametrów adresów URL	599
Dostęp do danych trasowania w innych komponentach	605
Bezpośredni dostęp do danych mechanizmu trasowania w komponencie	605
Dostęp do danych mechanizmu trasowania przy użyciu komponentu wyższego rzędu	607
Programowe prowadzenie nawigacji	609
Nawigacja programowa z użyciem komponentów	610
Pytanie użytkownika przed wykonaniem nawigacji	611
Programowe generowanie tras	615
Trasowanie z komponentami podłączonymi do magazynu danych	617
Zastępowanie komponentów ProductDisplay i SupplierDisplay	618
Aktualizacja podłączonego komponentu edytora	619
Aktualizowanie komponentu tabeli podłączonej do magazynu danych	620
Dokończenie konfiguracji trasowania	622
Podsumowanie	624
Rozdział 23. Korzystanie z usługi internetowej typu RESTful	625
Przygotowania do prac w tym rozdziale	626
Dodanie pakietów do projektu	626
Przygotowanie usługi internetowej	627
Dodanie komponentu i trasy	628
Uruchamianie usługi internetowej i aplikacji	629
Opis usług internetowych typu RESTful	631
Korzystanie z usługi internetowej	632
Tworzenie komponentu źródła danych	633
Pobieranie danych w komponencie	635
Zapisywanie, aktualizacja i usuwanie danych	636
Obsługa błędów	642
Korzystanie z usługi internetowej w magazynie danych	647
Tworzenie nowego komponentu warstwy pośredniej	647
Dodanie komponentu warstwy pośredniej do magazynu danych	648
Dokończanie zmian w aplikacji	649
Podsumowanie	651
Rozdział 24. Przedstawienie GraphQL-a	653
Przygotowania do prac w tym rozdziale	654
Omówienie GraphQL-a	655
Tworzenie serwera GraphQL-a	656
Tworzenie schematu	657
Tworzenie resolverów	658
Tworzenie serwera	658
Wykonywanie zapytań GraphQL-a	660
Pytania dotyczące powiązanych ze sobą danych	661
Tworzenie zapytań z argumentami	664

Wykonywanie mutacji GraphQL-a	669
Inne możliwości GraphQL-a	672
Stosowanie zmiennych żądania	672
Wykonywanie wielu żądań	673
Stosowanie fragmentów do wybierania pól	675
Podsumowanie	676
Rozdział 25. Korzystanie z GraphQL-a	677
Przygotowania do prac w tym rozdziale	677
Dodanie niezbędnych pakietów	677
Zmiana danych dla serwera GraphQL-a	678
Aktualizacja schematu i resolverów	678
Integracja serwera GraphQL-a z narzędziami programistycznymi Reacta	681
Korzystanie z usługi GraphQL	682
Zdefiniowanie zapytań i mutacji	682
Definiowanie źródła danych	684
Konfiguracja komponentów izolowanych	685
Używanie GraphQL-a z magazynem danych	687
Dostosowanie do formatu danych GraphQL-a	689
Stosowanie frameworka klienta GraphQL	693
Konfiguracja klienta	693
Tworzenie komponentów korzystających z GraphQL-a	694
Stosowanie mutacji	698
Dodanie obsługi danych dostawców oraz edycji danych	702
Podsumowanie	707

ROZDZIAŁ 1.



Twoja pierwsza aplikacja Reacta

Najlepszym sposobem, by rozpocząć korzystanie z Reacta, jest zakasać rękawy i wziąć się do pracy. W tym rozdziale przedstawię prosty proces tworzenia aplikacji zarządzającej listą rzeczy do zrobienia. W rozdziałach od 5. do 8. opiszę tworzenie bardziej złożonej i realistycznej aplikacji, lecz jak na razie prosty przykład w zupełności wystarczy, by pokazać sposób tworzenia aplikacji Reacta oraz wyjaśnić sposób ich działania i podstawowe możliwości. Nie przejmuj się, jeśli nie zrozumiesz wszystkiego, o czym będę pisał w tym rozdziale — jego głównym celem jest, byś poznał sposób działania Reacta. Wszelkie szczegóły wyjaśnię dokładnie w kolejnych rozdziałach.

-
- **Uwaga** Jeśli szukasz zwyczajnego opisu możliwości frameworka React, zajrzyj do drugiej części tej książki, w której rozpocznę szczegółową prezentację jego poszczególnych możliwości. Jednak zanim przejdiesz dalej, upewnij się, że masz zainstalowane wszelkie niezbędne narzędzia programistyczne i pakiety opisane w tym rozdziale.
-

Przygotowanie środowiska programistycznego

Korzystanie z frameworka React wymaga pewnych przygotowań. W kolejnych punktach tego rozdziału opiszę, jak przygotować się do utworzenia pierwszej aplikacji Reacta.

Instalacja Node.js

Narzędzia używane do tworzenia aplikacji Reacta bazują na Node.js — określanym także jako Node — utworzonym w 2009 roku jako proste i wygodne środowisko do tworzenia aplikacji serwerowych w języku JavaScript. Node.js bazuje na silniku JavaScript używanym w przeglądarce Chrome i udostępnia API służące do wykonywania kodu JavaScript poza środowiskiem przeglądarki WWW.

Node.js odniósł duży sukces jako serwer aplikacji, a w kontekście tej książki jest on interesujący, gdyż stanowi podstawę dla nowej generacji wieloplatformowych aplikacji i narzędzie do ich budowania.

Duże znaczenie ma to, byś zainstalował tę samą wersję Node.js, której używam w tej książce. Choć Node.js jest stosunkowo stabilnym środowiskiem, to jednak od pewnego czasu w jego API są wprowadzane zmiany, które mogłyby doprowadzić do problemów w działaniu prezentowanych przeze mnie przykładów. W tej książce będę używał Node.js w wersji 10.14.1, która w czasie prac

nad książką miała status wersji LTS¹ (ang. *Long-Term Support*). Może się zdarzyć, że kiedy będziesz czytać tę książkę, będą dostępne nowsze wersje Node, jednak pracując nad prezentowanymi przykładami, powinieneś używać wersji 10.14.1. Kompletny zestaw zasobów związanych z tą wersją Node, w tym także programy instalacyjne dla systemów Windows i macOS oraz pakiet dla innych platform systemowych, można znaleźć na stronie <https://nodejs.org/dist/v10.14.1/>.

Podczas instalowania Node.js koniecznie zaznacz opcję pozwalającą na dodanie do ścieżki plików wykonywalnych środowiska. Po zakończeniu instalacji wykonaj polecenie przedstawione na listingu 1.1.

Listing 1.1. Sprawdzanie wersji Node

```
node -v
```

Jeśli instalacja przebiegła pomyślnie, to wykonanie powyższego polecenia powinno spowodować wyświetlenie następującego komunikatu:

```
v10.14.1
```

Zainstalowane środowisko Node.js zawiera Node Package Manager (NPM) — program służący do zarządzania pakietami wchodzącymi w skład projektu. Wykonaj polecenie przedstawione na listingu 1.2, aby upewnić się, że NPM działa prawidłowo.

Listing 1.2. Sprawdzanie działania NPM

```
npm -v
```

Jeśli wszystko poszło, jak należy, to powinien zostać wyświetlony poniższy numer wersji:

```
6.4.1
```

Instalacja pakietu create-react-app

Pakiet create-react-app jest standardowym sposobem tworzenia złożonych pakietów Reacta i zarządzania nimi oraz dostarcza programistom kompletny zestaw narzędziowy (ang. *toolchain*). Są także inne sposoby rozpoczynania pracy z frameworkiem React, jednak korzystanie z pakietu create-react-app najlepiej odpowiada potrzebom większości projektów i właśnie jego będę używał w tej książce.

Aby zainstalować ten pakiet, należy otworzyć nowe okno wiersza poleceń i wykonać polecenie przedstawione na listingu 1.3. W systemach Linux lub macOS konieczne może być użycie polecenia sudo.

Listing 1.3. Instalowanie pakietu create-react-app

```
npm install --global create-react-app@2.1.2
```

¹ LTS to wersje, które są aktualnie wspierane przez twórców Node.js i które są zalecane do stosowania w rozwiązaniach produkcyjnych — przyp. tłum.

Instalacja Git

System kontroli wersji Git jest narzędziem wymaganym do zarządzania niektórymi pakietami niezbędnymi do tworzenia aplikacji Reacta. Użytkownicy systemów Windows i macOS mogą pobrać programy instalacyjne dla swoich platform ze strony <https://git-scm.com/downloads>. (W przypadku systemu macOS uruchomienie programu instalacyjnego, który nie został cyfrowo podpisany przez twórców, wymaga zmiany ustawień bezpieczeństwa).

Większość dystrybucji systemu Linux jest domyślnie wyposażona w program Git. Jeśli jednak ktoś chce zainstalować jego nowszą wersję, to powinien przejrzeć instrukcje instalacji dla używanej dystrybucji dostępne na stronie <https://git-scm.com/download/linux>. Listing 1.4 przedstawia polecenie pozwalające zainstalować Git w systemie Ubuntu, który jest używaną przeze mnie dystrybucją Linuksa.

Listing 1.4. Instalacja Git

```
sudo apt-get install git
```

Po zainstalowaniu można otworzyć nowe okno wiersza poleceń i sprawdzić numer zainstalowanej wersji programu Git, wykonując polecenie przedstawione na listingu 1.5.

Listing 1.5. Sprawdzanie wersji Git

```
git --version
```

Wykonanie tego polecenia spowoduje wyświetlenie wersji zainstalowanego pakietu Git. W czasie przygotowywania niniejszej książki najnowsza dostępna wersja miała numer 2.23.0.

Instalacja edytora

Aplikacje Reacta można tworzyć, używając dowolnego edytora dla programistów, a tych jest niemal nieskończenie wiele. Niektóre z nich dysponują rozbudowanymi mechanizmami wspierającymi tworzenie aplikacji Reacta, takimi jak podświetlanie słów kluczowych oraz wyrażeń. Jeśli jeszcze nie masz swojego ulubionego edytora do tworzenia aplikacji internetowych, to możesz rozważyć użycie jednego z popularnych programów przedstawionych w tabeli 1.1. Pisząc tę książkę, nie korzystałem z żadnego konkretnego z nich, a Ty możesz wybrać ten, który będzie Ci najbardziej odpowiadać.

Instalacja przeglądarki

Ostatnią decyzją, jaką należy podjąć, jest wybór przeglądarki używanej podczas pracy do sprawdzania tworzonej aplikacji. Wszystkie nowoczesne przeglądarki mają świetne narzędzia dla programistów, które dobrze współdziałają z frameworkiem React; niemniej jednak istnieje użyteczne rozszerzenie dla przeglądarek Chrome i Firefox, `react-devtools`, które zapewnia możliwość przeglądania stanu aplikacji Reacta i jest szczególnie przydatne w przypadku tworzenia złożonych projektów. Szczegółowe informacje o tym rozszerzeniu oraz sposobie jego instalacji można znaleźć na stronie <https://github.com/facebook/react-devtools>. Podczas prac nad tą książką używałem przeglądarki Google Chrome i polecam korzystanie z niej podczas prac nad prezentowanymi przykładami.

Tabela 1.1. Popularne edytory dla programistów

Nazwa	Opis
Sublime Text	Sublime Text to komercyjny edytor działający na wielu platformach systemowych, dysponujący pakietami wspierającymi programowanie w większości używanych języków programowania, frameworkach i platformach. Szczegółowe informacje na jego temat można znaleźć na stronie http://www.sublimetext.com .
Atom	Atom to edytor rozpowszechniany jako oprogramowanie <i>open-source</i> , działający na wielu platformach, którego twórcy zwracają szczególną uwagę na rozszerzalność i możliwości konfiguracji. Szczegółowe informacje na jego temat można znaleźć na stronie https://atom.io .
Brackets	Brackets to edytor stworzony przez firmę Adobe i rozpowszechniany jako oprogramowanie <i>open-source</i> . Informacje na jego temat można znaleźć na stronie http://brackets.io .
Visual Studio Code	To edytor stworzony przez firmę Microsoft i udostępniany jako oprogramowanie <i>open-source</i> ; działa on na wielu platformach systemowych i zapewnia duże możliwości rozszerzania. Informacje na jego temat można znaleźć na stronie https://code.visualstudio.com .
Visual Studio	Visual Studio jest flagowym narzędziem dla programistów stworzonym przez firmę Microsoft. Jest ono dostępne zarówno w wersji darmowej, jak i w komercyjnych. Visual Studio jest dostarczane wraz z szerokim zestawem narzędzi dodatkowych, które zapewniają jego integrację z ekosystemem firmy Microsoft.

Tworzenie projektu

Projekty są tworzone i zarządzane z poziomu wiersza poleceń. Aby utworzyć nowy projekt, otwórz nowe okno poleceń, przejdź do wybranego katalogu i wykonaj polecenie przedstawione na listingu 1.6.

- **Wskazówka** Przykładowy projekt przedstawiony w tym rozdziale można znaleźć w przykładach dołączonych do książki, które można pobrać z serwera FTP wydawnictwa Helion: <ftp://ftp.helion.pl/przyklady/react16.zip>.

Listing 1.6. Tworzenie projektu

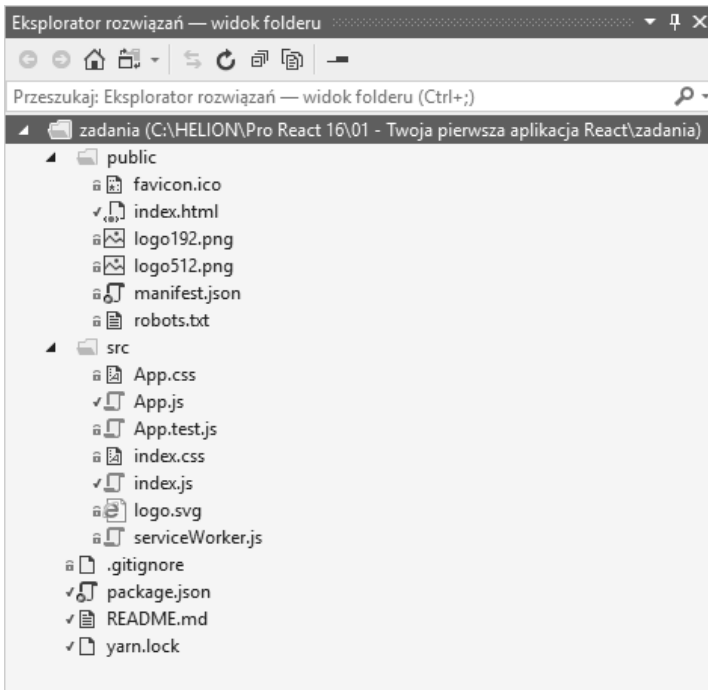
```
npx create-react-app zadania
```

Polecenie `npx` zostało zainstalowane w poprzednim podrozdziale wraz z Node.js i NPM. Służy ono do wykonywania pakietów Node.js. Argument `create-react-app` informuje program `npx`, że należy wykonać pakiet `create-react-app` służący do tworzenia nowych aplikacji Reacta i zainstalowany przy użyciu polecenia z listingu 1.3. Ostatni argument polecenia, `zadania`, określa nazwę tworzonego projektu. Wykonanie tego polecenia spowoduje utworzenie projektu oraz pobranie i zainstalowanie wszystkich pakietów potrzebnych do tworzenia i uruchamiania projektów Reacta. Proces ten może zająć kilka chwil, gdyż wymaga pobrania wielu pakietów.

- **Uwaga** Podczas tworzenia nowego projektu mogą się pojawić ostrzeżenia o potencjalnych zagrożeniach bezpieczeństwa. Tworzenie aplikacji Reacta wymaga użycia bardzo wielu pakietów, z których każdy ma własne zależności; dlatego też nieuniknione jest, że wcześniej czy później zostaną w nich odnalezione jakieś słabe punkty. W przypadku przykładów prezentowanych w tej książce bardzo duże znaczenie ma używanie wskazanych wersji pakietów, gdyż gwarantuje to uzyskanie oczekiwanych rezultatów. W przypadku własnych projektów należy przejrzeć ostrzeżenia i tak zaktualizować pakiety, by problemy zostały rozwiązane.

Prezentacja struktury projektu

Otwórz katalog *zadania*, używając w tym celu swojego ulubionego edytora. Zobaczysz w nim strukturę przedstawioną na rysunku 1.1. Rysunek ten prezentuje strukturę katalogów projektu wyświetloną w moim ulubionym edytorze — Visual Studio — oczywiście w innym programie sposób jej prezentacji może być nieco inny.



Rysunek 1.1. Struktura projektu

To jest punkt wyjścia dla każdego z projektów i choć przeznaczenie poszczególnych plików może jeszcze nie być dla Ciebie jasne, to jednak pod koniec lektury tej książki będziesz je doskonale rozumiał. A na razie w tabeli 1.2 pokrótce opisałem pliki mające znaczenie dla zagadnień opisywanych w tym rozdziale; szczegółowy opis struktury aplikacji Reacta można znaleźć w rozdziale 9.

Tabela 1.2. Pliki projektu mające duże znaczenie w kontekście tego rozdziału

Nazwa pliku	Opis
<code>public/index.html</code>	To plik HTML wyświetlany w przeglądarce. Zawiera on element, w którym jest wyświetlana aplikacja, oraz element <code>script</code> , który wczytuje pliki JavaScript aplikacji.
<code>src/index.js</code>	To plik JavaScript odpowiedzialny za konfigurację i uruchamianie aplikacji Reacta. Ja używam go także do dodania do aplikacji frameworka CSS Bootstrap, co opiszę w następnym punkcie rozdziału.
<code>src/App.js</code>	To komponent Reacta zawierający treści HTML wyświetlane użytkownikom aplikacji oraz kod JavaScript wymagany przez te treści HTML. Komponenty są głównymi elementami konstrukcyjnymi aplikacji Reacta i będą używane w całej książce.

Dodanie frameworka CSS Bootstrap

Do określenia wyglądu kodu HTML przykładów prezentowanych w tej książce będę używał doskonałego frameworka CSS Bootstrap. Podstawowe informacje dotyczące jego stosowania opiszę w rozdziale 3., a na razie, by ułatwić rozpoczęcie pracy, wykonaj polecenia przedstawione na listingu 1.7; spowodują one przejście do katalogu *zadania* i dodanie do projektu pakietu Bootstrap.

- **Wskazówka** Do zarządzania pakietami wchodzącymi w skład projektu służy polecenie `npm`, które niestety jest bardzo podobne do polecenia `npx`. To drugie jest jednak używane tylko podczas tworzenia nowych projektów. Bardzo ważne jest, by nie mylić tych dwóch poleceń.

Listing 1.7. Dodawanie frameworka CSS Bootstrap

```
cd zadania
npm install bootstrap@4.1.2
```

Aby dodać framework Bootstrap do aplikacji, należy dodać do pliku `index.js` wiersz wyróżniony na listingu 1.8 pogrubioną czcionką.

Listing 1.8. Dołączanie frameworka Bootstrap w pliku `src/index.js`

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import 'bootstrap/dist/css/bootstrap.css';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Jak wyjaśnię w rozdziale 4., instrukcja `import` służy do deklarowania zależności, tak by stały się one częścią aplikacji. W przeważającej większości przypadków jest ona używana do deklarowania zależności od kodu JavaScript, choć można jej także używać do dołączania arkuszy stylów CSS.

Uruchamianie narzędzi dla programistów

W przypadku tworzenia projektu przy użyciu pakietu `create-react-app` instalowany jest kompletny zestaw narzędzi dla programistów pozwalający na skompilowanie projektu, spakowanie aplikacji i dostarczenie jej do przeglądarki. Z poziomu wiersza poleceń przejdź do katalogu *zadania* i uruchom te narzędzia, wykonując polecenie przedstawione na listingu 1.9.

Listing 1.9. Uruchamianie narzędzi dla programistów

```
npm start
```

Po wykonaniu tego polecenia zostanie rozpoczęty wstępny proces przygotowawczy, którego zakończenie może chwilę potrwać. Nie należy zrażać się czasem tych przygotowań, gdyż proces ten jest wykonywany wyłącznie podczas uruchamiania prac nad aplikacją. Po zakończeniu tego procesu zostanie wyświetlony komunikat podobny do przedstawionego poniżej; potwierdza on, że aplikacja działa, i informuje, na którym porcie można się z nią połączyć:

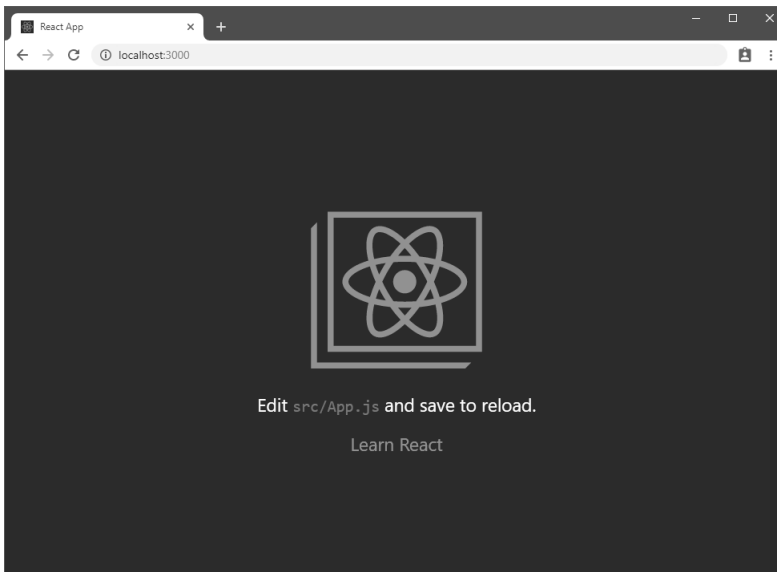
```
Compiled successfully!
```

```
You can now view zadania in the browser.
```

```
Local:           http://localhost:3000/
On Your Network: http://192.168.0.77:3000/
```

```
Note that the development build is not optimized.
To create a production build, use yarn build.
```

Domyślny port używany do odbierania żądań HTTP ma numer 3000, choć jeśli ten będzie już zajęty, to zostanie wybrany inny. Po zakończeniu wstępnych przygotowań zostanie otworzone okno przeglądarki, a w niej wyświetlona strona z treścią zastępczą, przedstawiona na rysunku 1.2.



Rysunek 1.2. Uruchamianie przykładowej aplikacji

Usuwanie treści zastępczej

Treści wyświetlone na stronie widocznej na rysunku 1.2 są jedynie zamiennikiem pozwalającym upewnić się, że narzędzia programistyczne działają. Aby je usunąć, wystarczy zmienić plik *App.js* w sposób przedstawiony na listingu 1.10.

Listing 1.10. *Usuwanie domyślnych treści z pliku *src/App.js**

```
import React, { Component } from 'react';
//import logo from './logo.svg';
//import './App.css';

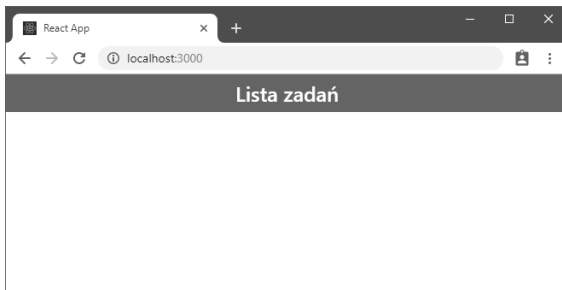
export default class App extends Component {

  render() {
    return (
      <div>
        <h1 className="bg-primary text-white text-center p-2">
          Lista zadań
        </h1>
      </div>
    )
  };
}
```

Plik *App.js* zawiera *komponent* Reacta o nazwie *App*. Komponenty są podstawowymi elementami konstrukcyjnymi aplikacji Reacta. Są one pisane w języku JSX — nadzbiorze języka JavaScript pozwalającym na bezpośrednie wstawianie fragmentów HTML do kodu JavaScript bez konieczności stosowania żadnych dodatkowych oznaczeń. Język JSX opiszę szczegółowo w rozdziale 3. Na razie warto zwrócić uwagę, że komponent *App* definiuje metodę *render*, która jest wywoływana przez framework React w celu wyświetlenia komponentu na stronie.

-
- **Wskazówka** React obsługuje najnowsze możliwości języka JavaScript, takie jak słowo kluczowe `class`, którego użyłem w kodzie z listingu 1.10. Podstawowe informacje na temat najczęściej używanych możliwości języka JavaScript można znaleźć w rozdziale 4.
-

Po zapisaniu pliku *App.js* narzędzia dla programistów automatycznie wykryją zmiany, ponownie zbudują aplikację i nakażą przeglądarkę jej powtórne wyświetlenie; w efekcie w przeglądarce zostanie wyświetlona strona podobna do tej przedstawionej na rysunku 1.3.



Rysunek 1.3. *Aplikacja po zastąpieniu domyślnej treści*

Pliki JSX stosowane w aplikacjach Reacta bardzo ułatwiają mieszanie kodu HTML i JavaScript, jednak różnią się one od plików HTML pod kilkoma ważnymi względami. Jedna z takich różnic jest widoczna w elemencie `h4` z listingu 1.10; przedstawiłem ją poniżej:

```
...
<h4 className="bg-primary text-white text-center p-2">
  Lista zadań
</h4>
...
```

W standardowych plikach HTML do przypisywania klas elementom HTML służy atrybut `class` — to właśnie on jest używany do określania postaci elementów podczas korzystania z frameworka CSS Bootstrap. Jednak, choć być może nie jest to wyraźnie widoczne, pliki JSX są plikami JavaScript, a w tym języku do określania klas jest używana właściwość `className`. Dla osób rozpoczynających korzystanie z frameworka React różnice pomiędzy czystym kodem HTML a kodem JSX mogą być drażniące, jednak z czasem staną się one czymś naturalnym.

-
- **Wskazówka** Krótkie wprowadzenie do stosowania frameworka CSS Bootstrap zamieszczę w rozdziale 3. wraz z wyjaśnieniem znaczenia klas użytych w elemencie `h4` z listingu 1.10, takich jak `bg-primary`, `text-white` oraz `p-2`. Na razie możesz te klasy zignorować i skoncentrować się na podstawowej strukturze aplikacji.
-

Jeśli zapomnisz, że pracujesz z plikiem JSX, i użyjesz rozwiązania typowego dla HTML, React wyświetli w konsoli JavaScript przeglądarki stosowne ostrzeżenie. Na przykład użycie atrybutu `class` zamiast `className` spowoduje wyświetlenie komunikatu o treści `Invalid DOM property 'class'. Did you mean 'className'?`. Aby wyświetlić konsolę JavaScriptu, należy nacisnąć klawisz `F12`, a następnie przejść na kartę *Console* lub *JavaScript Console*.

Wyświetlanie treści dynamicznych

Wszystkie aplikacje internetowe wymagają wyświetlania swoim użytkownikom dynamicznych treści, a React ułatwia to dzięki swoim *wyrażeniom*. Wyrażenie to fragment kodu JavaScript, który jest przetwarzany podczas wywołania metody `render` komponentu i który zapewnia możliwość wyświetlania danych użytkownikowi aplikacji. Wiele wyrażeń jest używanych do wyświetlania wartości definiowanych przez komponent do przechowywania stanu aplikacji, nazywanych także *danymi stanu*. Dane stanu oraz wyrażenia łatwiej zrozumieć na przykładzie — listing 1.11 przedstawia komponent `App`, do którego dodałem zarówno dane stanu, jak i wyrażenie.

Listing 1.11. Dodanie danych stanu oraz powiązania danych do pliku `src/App.js`

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

export default class App extends Component {

  constructor(props) {
    super(props);
    this.state = {
      userName: "Adam"
    }
  }

  render() {
```

```

return (
  <div>
    <h4 className="bg-primary text-white text-center p-2">
      Lista zadań użytkownika { this.state.userName }
    </h4>
  </div>
)
};
}

```

constructor to specjalna metoda nazywana także konstruktorem, wywoływana podczas inicjalizacji komponentu. Z kolei wywołanie w nim metody super jest niezbędne do zapewnienia prawidłowej konfiguracji komponentu (informacje na ten temat można znaleźć w rozdziale 11.). Parametr props zdefiniowany w konstruktorze jest ważny w aplikacjach Reacta, gdyż pozwala na konfigurowanie jednych komponentów przez inne, o czym się już niedługo przekonasz.

-
- **Wskazówka** Słowo *props* to skrót od angielskiego słowa *properties* oznaczającego właściwości. W kontekście frameworka React odzwierciedla ono sposób, w jaki tworzy on treści HTML wyświetlane w przeglądarce WWW (szczegółowe informacje na ten temat można znaleźć w rozdziale 3.).
-

Komponenty Reacta dysponują specjalną właściwością o nazwie state, która służy do definiowania danych stanu; oto przykład:

```

...
this.state = {
  userName: "Adam"
}
...

```

Słowo kluczowe `this` reprezentuje aktualny obiekt i zapewnia możliwość odwoływania się do jego właściwości i metod. Wyróżnione wyrażenie zapisuje we właściwości `this.state` obiekt zawierający właściwość `userName`. I to wszystko, czego trzeba do określenia danych stanu. Po zdefiniowaniu danych stanu można je umieszczać w wyrażeniach używanych w komponentach do generowania treści, jak pokazałem na poniższym przykładzie:

```

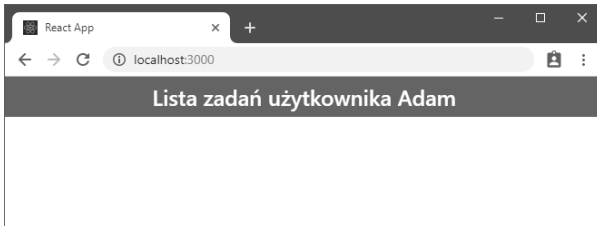
...
<h4 className="bg-primary text-white text-center p-2">
  Lista zadań użytkownika { this.state.userName }
</h4>
...

```

Wyrażenia są zapisywane w nawiasach klamrowych (`{}`). W momencie wywołania metody `render` wyrażenia są przetwarzane, a wyniki są umieszczane w treści wyświetlanej użytkownikom. Wyrażenie przedstawione na listingu 1.11 odczytuje wartość danych stanu o nazwie `userName` i generuje wynik przedstawiony na rysunku 1.4.

Wyjaśnienie zmian danych stanu

Dynamiczny charakter aplikacji Reacta bazuje na zmianach danych stanu, na które framework odpowiada poprzez ponowne wywołanie metody `render`, co z kolei powoduje ponowne przetworzenie wyrażen z wykorzystaniem nowych wartości danych stanu. Listing 1.12 przedstawia zmienioną wersję komponentu `App`, w której wartość danych stanu `userName` jest modyfikowana.



Rysunek 1.4. Zastosowanie danych stanu i wyrażenia w pliku `src/App.js`

Listing 1.12. Modyfikowanie danych stanu w pliku `src/App.js`

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

export default class App extends Component {

  constructor(props) {
    super(props);
    this.state = {
      userName: "Adam"
    }
  }

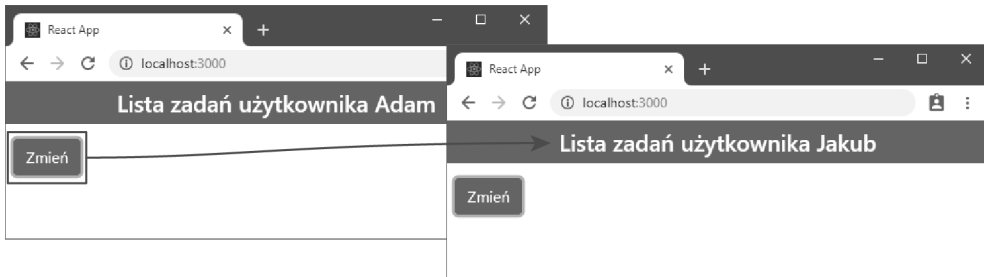
  changeStateData = () => {
    this.setState({
      userName: this.state.userName === "Adam" ? "Jakub" : "Adam"
    })
  }

  render() {
    return (
      <div>
        <h4 className="bg-primary text-white text-center p-2">
          Lista zadań użytkownika { this.state.userName }
        </h4>
        <button className="btn btn-primary m-2"
          onClick={this.changeStateData}>
          Zmień
        </button>
      </div>
    )
  };
}
```

Kiedy zapiszesz plik `App.js`, w oknie przeglądarki pojawi się przycisk *Zmień* — jego klikanie będzie powodować zmienianie imienia użytkownika (patrz rysunek 1.5).

Ten przykład korzysta z kilku możliwości frameworka React, które ze sobą współpracują. Pierwszą z nich jest atrybut `onClick` dodany do elementu `button`:

```
...
<button className="btn btn-primary m-2" onClick={this.changeStateData}>
  Zmień
</button>
...
```



Rysunek 1.5. Zmiana imienia użytkownika

Atrybut ten zawiera wyrażenie, które framework wykona, kiedy przycisk zostanie kliknięty. Kliknięcie przycisku powoduje zgłoszenie *zdarzenia*, a `onClick` to przykład właściwości definiującej procedurę obsługi zdarzeń. Funkcja lub metoda określona w atrybucie `onClick` zostanie wywołana po każdym kliknięciu przycisku. Jak widać na listingu 1.12, metoda `changeStateData` została zdefiniowana przy użyciu składni *grubej strzałki* (ang. *fat arrow*), która pozwala na tworzenie funkcji w bardziej zwięzły sposób, przedstawiony poniżej:

```
...
changeStateData = () => {
  this.setState({ userName: this.state.userName === "Adam" ? "Jakub" : "Adam" })
}
...
```

Jak wyjaśnię w rozdziale 4., funkcje tworzone przy użyciu składni grubej strzałki służą do upraszczania obsługi zdarzeń; niemniej jednak można ich używać także bardziej ogólnie i pozwalają w czytelny sposób łączyć kod JavaScript i HTML w aplikacjach Reacta. Metoda `changeStateData` używa metody `setState`, aby określić nową wartość właściwości `userName`. Kiedy zostanie wywołana metoda `setState`, React zaktualizuje stan komponentu, zapisując w nim nowe wartości, a następnie wywoła metodę `render`, tak by wyrażenia wygenerowały zmodyfikowaną treść. To właśnie dzięki temu kliknięcie przycisku spowoduje zmianę imienia wyświetlanego w przeglądarce z Adam na Jakub. Nie musiałem w tym celu jawnie informować frameworka React, że wartość używana przez wyrażenie uległa zmianie — wywołałem jedynie metodę `setState`, aby podać nową wartość, a pozostałe czynności związane z aktualizacją treści wyświetlanej w przeglądarce pozostawiłem frameworkowi.

- **Wskazówka** Podczas korzystania z właściwości i metod zdefiniowanych w komponentcie, w tym także z metody `setState`, konieczne jest posługiwanie się słowem kluczowym `this`. Pomijanie tego słowa kluczowego jest błędem powszechnie popełnianym podczas tworzenia aplikacji Reacta i należy je sprawdzać w pierwszej kolejności, jeśli aplikacja, nad którą pracujemy, nie będzie działać w oczekiwany sposób.

Funkcje tworzone przy użyciu składni grubej strzałki nie wymagają stosowania instrukcji `return` ani zapisywania ciała funkcji wewnątrz nawiasów klamrowych. Dzięki temu tworzone przy ich użyciu metody `render` mogą mieć nieco prostszą postać, co pokazałem na listingu 1.13.

Listing 1.13. Predefiniowanie metody `render` przy użyciu składni grubej strzałki w pliku `src/App.js`

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

export default class App extends Component {

  constructor(props) {
```

```

    super(props);
    this.state = {
      userName: "Adam"
    }
  }

  changeStateData = () => {
    this.setState({
      userName: this.state.userName === "Adam" ? "Jakub" : "Adam"
    })
  }

  render = () =>
    <div>
      <h4 className="bg-primary text-white text-center p-2">
        Lista zadań użytkownika {this.state.userName}
      </h4>
      <button className="btn btn-primary m-2"
        onClick={this.changeStateData}>
        Zmień
      </button>
    </div>
  }

```

W tej książce będę używał obu tych sposobów zapisu funkcji i metod. W większości przypadków będziesz mógł samemu wybrać, czy chcesz stosować konwencjonalny zapis funkcji JavaScript, czy składować go za pomocą strzałki, choć istnieją pewne szczególne warunki, które opiszę dokładniej w rozdziale 12.

Dodawanie możliwości aplikacji listy zadań

Skoro już dowiedziałeś się, w jaki sposób React może wyświetlać dynamiczne treści, możemy zająć się dodawaniem elementów wymaganych przez naszą aplikację. Zaczę od danych stanu oraz wyrażień, które przedstawiłem na listingu 1.14.

Listing 1.14. Dodawanie elementów aplikacji w pliku `src/App.js`

```

import React, { Component } from 'react';

export default class App extends Component {

  constructor(props) {
    super(props);
    this.state = {
      userName: "Adam",
      todoItems: [{ action: "Kupić kwiaty", done: false },
                  { action: "Wziąć buty", done: false },
                  { action: "Zebrać bilety", done: true },
                  { action: "Zadzwoń do Jurka", done: false }],
      newItemText: ""
    }
  }

  updateNewItemValue = (event) => {
    this.setState({ newItemText: event.target.value });
  }

```

```

    }

    createNewTodo = () => {
      if (!this.state.todoItems
        .find(item => item.action === this.state.newItemText)) {
        this.setState({
          todoItems: [...this.state.todoItems,
            { action: this.state.newItemText, done: false }],
          newItemText: ""
        });
      }
    }
  }

  render = () =>
    <div>
      <h4 className="bg-primary text-white text-center p-2">
        Lista zadań użytkownika {this.state.userName}
        (Liczba zadań: {this.props.tasks.filter(t => !t.done).length})
      </h4>
      <div className="container-fluid">
        <div className="my-1">
          <input className="form-control"
            value={this.state.newItemText}
            onChange={this.updateNewTextValue} />
          <button className="btn btn-primary mt-1"
            onClick={this.createNewTodo}>Dodaj</button>
        </div>
      </div>
    </div>
  }

```

Ponieważ wyrażenia frameworka React są kodem JavaScript, można ich używać do sprawdzania wartości danych oraz dynamicznego generowania wyników; oto przykład:

```

...
<h4 className="bg-primary text-white text-center p-2">
  Lista zadań użytkownika {this.state.userName}
  (Liczba zadań: {this.state.todoItems.filter(t => !t.done).length})
</h4>
...

```

To wyrażenie filtruje obiekty zapisane w tablicy `todoItems` będącej danymi stanu i wybiera z niej jedynie te zadania, które nie są zakończone, następnie odczytuje wartość właściwości `length` tak utworzonej tablicy. Ta wartość, dzięki powiązaniu danych, jest następnie wyświetlana na stronie. Format JSX ułatwia mieszanie kodu elementów HTML z kodem JavaScript takim jak ten, choć złożone wyrażenia mogą być trudne do przeanalizowania i często definiuje się je jako właściwości lub metody, by tworzony kod HTML był możliwie jak najprostszy.

W nowym kodzie przedstawionym na listingu 1.14 znajduje się także element `input`, pozwalający użytkownikowi na wpisanie nazwy nowego zadania. Element ten ma dwie właściwości, `value` oraz `onClick`, służące odpowiednio do: określania zawartości elementu oraz reagowania na jej zmiany. Poniżej przedstawiłem jego postać:

```

...
<input className="form-control"
  value={this.state.newItemText} onChange={this.updateNewTextValue} />
...

```

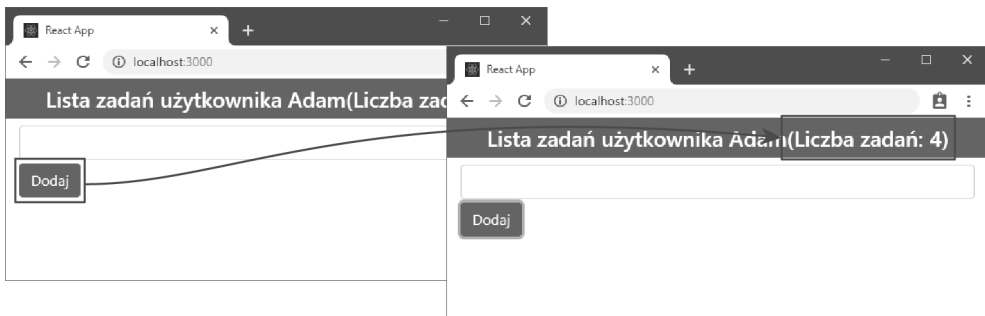
Właściwość `value` jest używana do określania zawartości elementu `input`. W naszym przypadku wyrażenie umieszczone w tej właściwości zwraca wartość właściwości `newItemText` stanowiącej dane stanu. Oznacza to, że każda zmiana tych danych stanu spowoduje aktualizację zawartości elementu `input`. Właściwość `onChange` informuje framework, co należy zrobić w momencie zgłoszenia zdarzenia `change`, czyli wtedy, gdy użytkownik wpisze coś w elemencie. To wyrażenie informuje framework React, że należy wywołać metodę `updateNewItemValue` komponentu, która używa metody `setState`, by zaktualizować wartość danych stanu `newItemText`. Można by sądzić, że takie rozwiązanie jest nieco okrężne, jednak zapewnia ono, że React wie, jak obsługiwać zmiany wprowadzane przez użytkownika oraz przez kod.

Element `button` używa właściwości `onClick`, by nakazać frameworkowi wywołanie metody `createNewTodo` w odpowiedzi na zgłoszenie zdarzenia `click`. Metoda `createNewTodo` sprawdza, czy istnieje już zadanie o podanej nazwie, a jeśli takiego nie ma, to używa metody `setState`, by dodać nowy element do tablicy `todoItems`, po czym zapisuje we właściwości `newItemText` pusty łańcuch, co powoduje wyczyszczenie elementu `input`. Instrukcja, która dodaje nowy element do tablicy, wykorzystuje jeden z nowych elementów języka JavaScript — operator *rozproszenia*.

```
...
todoItems: [...this.state.todoItems,
  { action: this.state.newItemText, done: false } ]
...
```

Operator rozproszenia ma postać trzech kropek (`...`) i powoduje wyodrębnienie poszczególnych elementów tablicy. Narzędzia używane do tworzenia aplikacji Reacta pozwalają na stosowanie najnowszych możliwości języka JavaScript i tłumaczą je na kod, który jest zgodny ze starszymi wersjami przeglądarek. Operator rozproszenia oraz inne użyteczne możliwości języka JavaScript opiszę w rozdziale 4.

Aby obejrzeć efekty zmian wprowadzonych na listingu 1.14, należy wpisać tytuł zadania i kliknąć przycisk *Dodaj*. React odpowie na wystąpienie zdarzenia, wywołując metodę podaną we właściwości `onClick` przycisku, a ta użyje tekstu wpisanego w polu do utworzenia nowego zadania. Poszczególne zadania jeszcze nie są wyświetlane, jednak na stronie jest widoczna ich liczba (patrz rysunek 1.6).



Rysunek 1.6. Dodawanie nowego zadania

Wyświetlanie zadań do zrobienia

Kolejnym krokiem jest wyświetlenie każdego z zadań do zrobienia, tak by użytkownik mógł zobaczyć ich tytuły i oznaczać je jako wykonane. Umożliwia to nowy kod, wyróżniony na listingu 1.15.

Listing 1.15. Wyświetlanie zadań do zrobienia w pliku `src/App.js`

```

import React, { Component } from 'react';

export default class App extends Component {

  constructor(props) {
    super(props);
    this.state = {
      userName: "Adam",
      todoItems: [{ action: "Kupić kwiaty", done: false },
                  { action: "Wziąć buty", done: false },
                  { action: "Zebrać bilety", done: true },
                  { action: "Zadzwońić do Jurka", done: false }],
      newItemText: ""
    }
  }

  updateNewItemValue = (event) => {
    this.setState({ newItemText: event.target.value });
  }

  createNewTodo = () => {
    if (!this.state.todoItems
        .find(item => item.action === this.state.newItemText)) {
      this.setState({
        todoItems: [...this.state.todoItems,
                    { action: this.state.newItemText, done: false }],
        newItemText: ""
      });
    }
  }

  toggleTodo = (todo) => this.setState({
    todoItems:
      this.state.todoItems.map(item => item.action === todo.action
        ? { ...item, done: !item.done } : item
    });

  todoTableRows = () => this.state.todoItems.map(item =>
    <tr key={item.action}>
      <td>{item.action}</td>
      <td>
        <input type="checkbox" checked={item.done}
              onChange={() => this.toggleTodo(item)} />
      </td>
    </tr>);

  render = () =>
    <div>
      <h4 className="bg-primary text-white text-center p-2">
        Lista zadań użytkownika {this.state.userName}
        (Liczba zadań: {this.state.todoItems.filter(t => !t.done).length})
      </h4>
      <div className="container-fluid">
        <div className="my-1">
          <input className="form-control"
                value={this.state.newItemText}

```

```

        onChange={this.updateNewTextValue} />
        <button className="btn btn-primary mt-1"
            onClick={this.createNewTodo}>Dodaj</button>
    </div>
    <table className="table table-striped table-bordered">
        <thead>
            <tr><th>Opis</th><th>Wykonane</th></tr>
        </thead>
        <tbody>{this.todoTableRows()}</tbody>
    </table>
</div>
</div>
}

```

Jak na razie podczas tworzenia aplikacji w pliku *App.js* koncentrowaliśmy się głównie na umieszczaniu wyrażeń JavaScript we fragmentach kodu HTML. Jednak format JSX pozwala także na dowolne mieszanie kodu HTML i JavaScript, a to oznacza, że funkcje JavaScript mogą zwracać kod HTML. Doskonale to widać na listingu 1.15, w którym metoda `todoTableRows` używa metody `map` języka JavaScript, by wygenerować sekwencję elementów HTML dla każdego elementu tablicy `todoItems`:

```

...
todoTableRows = () => this.state.todoItems.map(item =>
    <tr key={item.action}>
        <td>{item.action}</td>
        <td>
            <input type="checkbox" checked={item.done}
                onChange={() => this.toggleTodo(item)} />
        </td>
    </tr>);
...

```

Każdy element tablicy jest odwzorowywany na element `tr`, czyli element HTML reprezentujący wiersz tabeli. Wewnątrz elementu `tr` umieszczona jest grupa elementów `td`, definiujących poszczególne komórki tabeli. Treść HTML wygenerowana przez metodę `map` zawiera kolejne wyrażenia JavaScript. Umieszczają one w elementach `td` wartości z danych stanu oraz określają funkcje, które należy wykonać w celu obsługi zgłaszanych zdarzeń.

React narzuca pewne ograniczenia na treści, które ma obsługiwać; przykładem może być właściwość `key` dodawana do każdego elementu `tr` w metodzie `todoTableRows`:

```

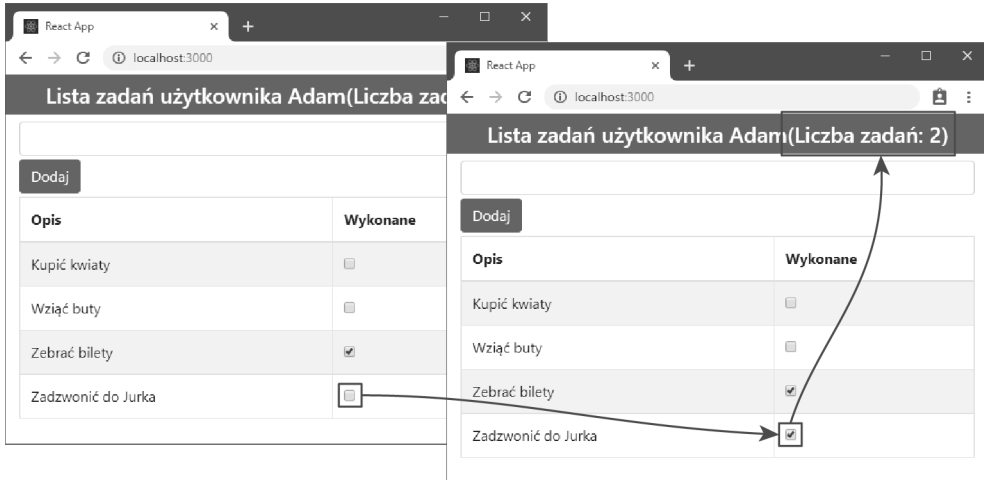
...
<tr key={ item.action }>
...

```

W rozdziale 13. dowiesz się, że React wywołuje metodę `render` komponentu, kiedy pojawia się jakaś zmiana, a następnie porównuje uzyskany wynik z aktualnie prezentowanym kodem HTML, tak by na stronie zostały wyświetlone wyłącznie zmienione treści. React wymaga właściwości `key`, by móc powiązać wyświetlane treści z danymi, na podstawie których zostały one wygenerowane, i efektywnie zarządzać zmianami.

Dzięki zmianom wprowadzonym na listingu 1.15 obok każdego zadania do zrobienia jest wyświetlane pole wyboru, które użytkownik może zaznaczyć, by oznaczyć dane zdanie jako wykonane. Każdy wiersz tabeli wygenerowany przez metodę `todoTableRow` zawiera element `input` wyświetlany jako pole wyboru.

Zmiany wprowadzone na ostatnim listingu powodują, że na stronie jest wyświetlana tabela z listą zadań do zrobienia, a oznaczanie zadań jako wykonanych powoduje zmniejszanie liczby wyświetlanej w nagłówku (patrz rysunek 1.7).



Rysunek 1.7. Wyświetlanie listy zadań do zrobienia

Wprowadzanie dodatkowych komponentów

Obecnie wszystkie możliwości funkcjonalne naszej przykładowej aplikacji są zaimplementowane w jednym komponencie. Jednak takie rozwiązanie może sprawić, że wraz z dodawaniem nowych możliwości zarządzanie tym komponentem może stać się trudne. Aby ułatwić zarządzanie komponentami, poszczególne możliwości funkcjonalne można przekazywać do odrębnych komponentów, które będą odpowiedzialne za realizację konkretnych możliwości. Takie komponenty są nazywane *komponentami podrzędnymi*, natomiast komponent przekazujący możliwości funkcjonalne, które należy zrealizować, jest nazywany *komponentem nadrzędnym*.

W tym podrozdziale przygotuję kilka komponentów podrzędnych, z których każdy będzie odpowiedzialny za jedną konkretną operację. Zaczę od dodania do katalogu `src` pliku o nazwie `TodoBanner.js`, w którym zdefiniuję komponent przedstawiony na listingu 1.16.

Listing 1.16. Zawartość pliku `src/ToDoBanner.js`

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

export class TodoBanner extends Component {
  render = () =>
    <h4 className="bg-primary text-white text-center p-2">
      Lista zadań użytkownika {this.props.name}
      (Liczba zadań: {this.props.tasks.filter(t => !t.done).length})
    </h4>
}
```

Ten komponent jest odpowiedzialny za wyświetlanie nagłówka listy zadań. Komponenty nadrzędne dostarczają swoim komponentom podrzędnym dane, wykorzystując do tego *właściwości*, do których można się odwoływać przy użyciu właściwości `props`, dostępnej za pośrednictwem słowa kluczowego `this`. Powyższy komponent o nazwie `TodoBanner` oczekuje dwóch właściwości: `name`

zawierającej imię użytkownika oraz `tasks` zawierającej listę zadań do wykonania, która jest filtrowana w celu określenia liczby zadań pozostających do zrobienia. Aby wyświetlić wartość właściwości `name`, w kodzie komponentu należy użyć wyrażenia `this.props.name`, jak pokazałem poniżej:

```
...
Lista zadań użytkownika { this.props.name }
...
```

Kiedy React wywoła metodę `render` komponentu `TodoBanner`, w wygenerowanych wynikach zostanie umieszczona wartość właściwości `name` przekazana przez komponent nadrzędny. Kolejne wyrażenie użyte w metodzie `render` komponentu `TodoBanner` używa metody `filter` języka JavaScript, by wyszukać niezakończone zadania i określić ich liczbę. Ten fragment kodu pokazuje, że właściwości mogą mieć bardziej złożone zastosowania niż tylko wyświetlanie ich wartości.

Kolejnym plikiem, który utworzę, będzie plik `TodoRow.js` umieszczony w katalogu `src`. Będzie on zawierał komponent `TodoRow` przedstawiony na listingu 1.17.

Listing 1.17. Zawartość pliku `src/TodoRow.js`

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

export class TodoRow extends Component {

  render = () =>
    <tr>
      <td>{ this.props.item.action}</td>
      <td>
        <input type="checkbox" checked={ this.props.item.done }
          onChange={ () => this.props.callback(this.props.item) }
        />
      </td>
    </tr>
  }
}
```

Ten komponent będzie odpowiedzialny za wyświetlanie pojedynczego wiersza tabeli, zawierającego szczegółowe informacje o zadaniu do wykonania. Dane otrzymywane przez ten komponent za pośrednictwem właściwości są przeznaczone wyłącznie do odczytu i nie można ich modyfikować. W celu wprowadzenia zmian w danych komponenty nadrzędne mogą używać *właściwości funkcyjnych* (ang. *function props*), aby przekazywać komponentom podrzędnym funkcje zwrotne, które będą wywoływane, kiedy zdarzy się coś ważnego. Takie rozwiązanie pozwala na współdziałanie komponentów: właściwości *props* pozwalają przekazywać dane z komponentów nadrzędnych do podrzędnych, a właściwości funkcyjne pozwalają komponentom podrzędnym komunikować się ze swoimi komponentami nadrzędnymi.

Komponent przedstawiony na listingu 1.17 definiuje właściwość *props* o nazwie `item` używaną do przekazywania danych o zadaniu, które należy wykonać, oraz właściwość funkcyjną o nazwie `callback` zawierającą funkcję, która zostanie wywołana, kiedy użytkownik zaznaczy pole wyboru. Ostatni komponent podrzędny, który utworzę, zostanie umieszczony w katalogu `src`, w pliku `TodoCreator.js`; jego kod przedstawiłem na listingu 1.18.

Listing 1.18. Zawartość pliku `src/TodoCreator.js`

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

export class TodoCreator extends Component {

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

```

    this.state = { newItemText: "" }
  }

  updateNewTextValue = (event) => {
    this.setState({ newItemText: event.target.value });
  }

  createNewTodo = () => {
    this.props.callback(this.state.newItemText);
    this.setState({ newItemText: "" });
  }

  render = () =>
    <div className="my-1">
      <input className="form-control" value={this.state.newItemText}
        onChange={this.updateNewTextValue} />
      <button className="btn btn-primary mt-1"
        onClick={this.createNewTodo}>Nowe zadanie</button>
    </div>
  }

```

Komponenty podrzędne mogą mieć swoje własne dane stanu. W przypadku tego komponentu dane stanu są używane do zarządzania zawartością elementu `input`. Komponent wywołuje właściwość funkcyjną, by poinformować swój komponent nadrzędny o kliknięciu przycisku *Nowe zadanie*.

Stosowanie komponentów podrzędnych

Komponenty zdefiniowane w poprzedniej części rozdziału realizują konkretne możliwości funkcjonalne aplikacji listy zadań. Na listingu 1.19 pokazałem zaktualizowany kod komponentu `App`. Zastosowałem w nim przygotowane wcześniej komponenty, z których każdy został skonfigurowany przy użyciu właściwości *props* zapewniających wymagane dane i funkcje zwrotne.

Listing 1.19. Stosowanie komponentów podrzędnych w pliku `src/App.js`

```

import React, { Component } from 'react';
import { TodoBanner } from './TodoBanner';
import { TodoCreator } from './TodoCreator';
import { TodoRow } from './TodoRow';

export default class App extends Component {

  constructor(props) {
    super(props);
    this.state = {
      userName: "Adam",
      todoItems: [{ action: "Kupić kwiaty", done: false },
        { action: "Wziąć buty", done: false },
        { action: "Zebrać bilety", done: true },
        { action: "Zadzwonić do Jurka", done: false }],
      //newItemText: ""
    }
  }

  updateNewTextValue = (event) => {
    this.setState({ newItemText: event.target.value });
  }

```

```

}

createNewTodo = (task) => {
  if (!this.state.todoItems.find(item => item.action === task)) {
    this.setState({
      todoItems: [...this.state.todoItems, { action: task, done: false }]
    });
  }
}

toggleTodo = (todo) => this.setState({
  todoItems:
    this.state.todoItems.map(item => item.action === todo.action
      ? { ...item, done: !item.done } : item)
});

todoTableRows = () => this.state.todoItems.map(item =>
  <TodoRow key={item.action} item={item} callback={this.toggleTodo} />)

render = () =>
  <div>
    <TodoBanner name={this.state.userName} tasks={this.state.todoItems} />
    <div className="container-fluid">
      <TodoCreator callback={this.createNewTodo} />
      <table className="table table-striped table-bordered">
        <thead>
          <tr><th>Opis</th><th>Done</th></tr>
        </thead>
        <tbody>{ this.todoTableRows() }</tbody>
      </table>
    </div>
  </div>
}

```

Nowe instrukcje `import` deklarują zależności z komponentami podrzędnymi, co zapewnia, że zostaną one dołączone do aplikacji podczas procesu jej budowania. Te komponenty podrzędne są używane jako niestandardowe elementy HTML, a atrybuty i wyrażenia definiują właściwości *props* przekazywane do komponentów. Oto przykład:

```

...
<TodoBanner name={this.state.userName} tasks={this.state.todoItems} />
...

```

Właściwości użyte do określenia wartości właściwości *props* komponentu podrzędnego zapewniają mu dostęp do konkretnych danych oraz metod dostarczanych przez jego komponent nadrzędny. W tym przypadku właściwości `name` oraz `tasks` dają komponentowi `TodoBanner` dostęp do wartości właściwości stanu o nazwach `userName` i `todoItems`.

Ostatnie szlify

Podstawowe możliwości funkcjonalne aplikacji są już gotowe, a komponenty, które je implementują, doskonale ze sobą współpracują. W tym podrozdziale wprowadzę jeszcze kilka ostatnich poprawek, które zakończą pracę nad aplikacją listy zadań.

Zarządzanie prezentacją zakończonych zadań

Obecnie wszystkie zadania są wyświetlane na liście, niezależnie od tego, czy są oznaczone jako wykonane, czy nie. Aby rozwiązać ten problem, będę wyświetlać na stronie dwie odrębne listy — zadań do wykonania oraz zadań wykonanych, i zapewnię możliwość ukrycia tej drugiej. W tym celu utworzymy w katalogu `src` nowy plik o nazwie `VisibilityControl.js`, w którym zdefiniuję komponent przedstawiony na listingu 1.20.

Listing 1.20. Zawartość pliku `src/VisibilityControl.js`

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

export class VisibilityControl extends Component {

  render = () =>
    <div className="form-check">
      <input className="form-check-input" type="checkbox"
        checked={this.props.isChecked}
        onChange={e => this.props.callback(e.target.checked)} />
      <label className="form-check-label">
        Pokaż {this.props.description}
      </label>
    </div>
}
```

Dzięki wykorzystaniu właściwość `props` do przekazywania danych i funkcji zwrotnych z komponentów nadrzędnych dodawanie nowych możliwości funkcjonalnych do aplikacji staje się bardzo łatwe. Komponent przedstawiony na listingu 1.20 ma charakter ogólny i nie dysponuje żadnymi informacjami na temat zawartości, które mają być zarządzane przy jego użyciu. Jego działanie bazuje wyłącznie na przekazanych właściwościach `props`: `description` — określającej zawartość wyświetlanej etykiety tekstowej, `isChecked` — określającej początkowy stan pola wyboru, oraz `callback` — określającej funkcję, która ma być wywoływana, kiedy użytkownik zaznaczy pole wyboru lub usunie zaznaczenie, powodując w ten sposób zgłoszenie zdarzenia `change`.

Listing 1.21 przedstawia zmodyfikowaną postać komponentu `App` korzystającego z komponentu `VisibilityControl` i uzupełnionego o kod wymagany do wyświetlania w formie odrębnych list zadań do wykonania i zadań wykonanych.

Listing 1.21. Zarządzanie zadaniami w pliku `src/App.js`

```
import React, { Component } from 'react';
import { TodoBanner } from './TodoBanner';
import { TodoCreator } from './TodoCreator';
import { TodoRow } from './TodoRow';
import { VisibilityControl } from './VisibilityControl';

export default class App extends Component {

  constructor(props) {
    super(props);
    this.state = {
      userName: "Adam",
      todoItems: [{ action: "Kupić kwiaty", done: false },
                  { action: "Wziąć buty", done: false },
                  { action: "Zebrać bilety", done: true },
                  { action: "Zadzwońić do Jurka", done: false }],
      showCompleted: true
    }
  }
}
```

```

}

updateNewTextValue = (event) => {
  this.setState({ newItemText: event.target.value });
}

createNewTodo = (task) => {
  if (!this.state.todoItems.find(item => item.action === task)) {
    this.setState({
      todoItems: [...this.state.todoItems, { action: task, done: false }]
    });
  }
}

toggleTodo = (todo) => this.setState({
  todoItems:
    this.state.todoItems.map(item => item.action === todo.action
      ? { ...item, done: !item.done } : item)
});

todoTableRows = (doneValue) => this.state.todoItems
.filter(item => item.done === doneValue).map(item =>
  <TodoRow key={item.action} item={item}
    callback={this.toggleTodo} />)

render = () =>
  <div>
    <TodoBanner name={this.state.userName} tasks={this.state.todoItems} />
    <div className="container-fluid">
      <TodoCreator callback={this.createNewTodo} />
      <table className="table table-striped table-bordered">
        <thead>
          <tr><th>Opis</th><th>Wykonane</th></tr>
        </thead>
        <tbody>{ this.todoTableRows(false) }</tbody>
      </table>
      <div className="bg-secondary text-white text-center p-2">
        <VisibilityControl description="wykonane zadania"
          isChecked={this.state.showCompleted}
          callback={(checked) =>
            this.setState({ showCompleted: checked })} />
      </div>

      {this.state.showCompleted &&
        <table className="table table-striped table-bordered">
          <thead>
            <tr><th>Opis</th><th>Wykonane</th></tr>
          </thead>
          <tbody>{this.todoTableRows(true)}</tbody>
        </table>
      }
    </div>
  </div>
}

```

Komponent `VisibilityControl` został skonfigurowany w taki sposób, że w zależności od zaznaczenia pola wyboru zmienia właściwość stanu komponentu `App` o nazwie `showCompleted`. Aby oddzielić zadania do zrobienia od tych, które już zostały wykonane, dodałem do metody `todoTableRows` parametr oraz użyłem metody `filter`, by na jego podstawie wybierać z tablicy zadań obiekty o odpowiedniej wartości właściwości `done`.

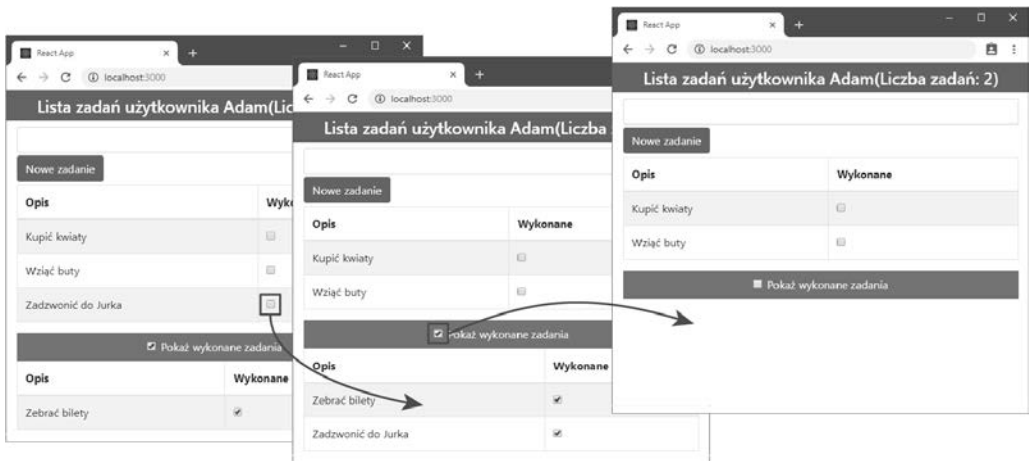
W celu wyświetlania zadań, które już zostały wykonane, umieściłem na stronie drugi element tabeli. Tabela ta będzie wyświetlana wyłącznie wtedy, gdy właściwość `showCompleted` będzie mieć wartość `true`. Z tego względu element `table` oraz całą jego zawartość umieściłem w wyrażeniu powiązania danych, w którym zastosowałem operator `&&`; poniżej przedstawiłem ten fragment kodu:

```
...
{ this.state.showCompleted && <table className="table table-striped table-bordered">
...

```

Podczas przetwarzania tego wyrażenia element `table` zostanie wstawiony do komponentu wyłącznie wtedy, gdy wartość właściwości `showCompleted` wyniesie `true`. To kolejny przykład pokazujący, jak format JSX umożliwia mieszanie treści i kodu. W większości przypadków JSX dobrze radzi sobie z łączeniem elementów i instrukcji JavaScript, choć nie jest on doskonały pod każdym względem — jak widać na tym przykładzie, składnia stanowiąca odpowiednik instrukcji warunkowej jest nieco dziwna.

Kiedy zapiszesz zmiany w pliku `App.js`, w przeglądarce zostaną wyświetlone dwie odrębne listy zadań. Kiedy zaznaczysz pole wyboru przy jakimś zadaniu, zostanie ono przeniesione do drugiej tabeli, jak pokazałem na rysunku 1.8. Usunięcie zaznaczenia z pola wyboru *Pokaż wykonane zadania* spowoduje ukrycie listy wykonanych zadań.



Rysunek 1.8. Zmiana wyświetlania zadań

Trwałe przechowywanie danych

Ostatnią zmianą, jaką wprowadzimy, będzie przechowywanie danych w taki sposób, by lista zadań została zachowana nawet po wyświetleniu w przeglądarce innej strony. W dalszej części książki przedstawię inne sposoby pracy z danymi przechowywanymi na serwerze, jak na razie jednak zależy mi na tym, by aplikacja była możliwie prosta, dlatego też dane będą zapisywane w przeglądarce przy wykorzystaniu API magazynu lokalnego (Local Storage API) w sposób przedstawiony na listingu 1.22.

-
- **Wskazówka** Local Storage API jest standardową funkcją przeglądarek i nie ma nic wspólnego z frameworkiem React. Doskonały opis działania tego sposobu trwałego przechowywania danych można znaleźć na stronie <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>.
-

Listing 1.22. Trwale przechowywanie danych aplikacji w pliku `src/App.js`

```
import React, { Component } from 'react';
import { TodoBanner } from './TodoBanner';
import { TodoCreator } from './TodoCreator';
import { TodoRow } from './TodoRow';
import { VisibilityControl } from './VisibilityControl';

export default class App extends Component {

  constructor(props) {
    super(props);
    this.state = {
      userName: "Adam",
      todoItems: [{ action: "Kupić kwiaty", done: false },
                  { action: "Wziąć buty", done: false },
                  { action: "Zebrać bilety", done: true },
                  { action: "Zadzwoń do Jurka", done: false }],
      showCompleted: true
    }
  }

  updateNewTextValue = (event) => {
    this.setState({ newItemText: event.target.value });
  }

  createNewTodo = (task) => {
    if (!this.state.todoItems.find(item => item.action === task)) {
      this.setState({
        todoItems: [...this.state.todoItems, { action: task, done: false }],
      }, () => localStorage.setItem("todos", JSON.stringify(this.state)));
    }
  }

  toggleTodo = (todo) => this.setState({
    todoItems:
      this.state.todoItems.map(item => item.action === todo.action
        ? { ...item, done: !item.done } : item)
  });

  todoTableRows = (doneValue) => this.state.todoItems
    .filter(item => item.done === doneValue).map(item =>
      <TodoRow key={item.action} item={item}
        callback={this.toggleTodo} />)

  componentDidMount = () => {
    let data = localStorage.getItem("todos");
    this.setState(data != null
      ? JSON.parse(data)
      : {
        userName: "Adam",
        todoItems: [{ action: "Kupić kwiaty", done: false },
```

```

        { action: "Wziąć buty", done: false },
        { action: "Zebrać bilety", done: true },
        { action: "Zadzwońić do Jurka", done: false }],
    showCompleted: true
  });
}

render = () =>
  <div>
    <TodoBanner name={this.state.userName} tasks={this.state.todoItems} />
    <div className="container-fluid">
      <TodoCreator callback={this.createNewTodo} />
      <table className="table table-striped table-bordered">
        <thead>
          <tr><th>Opis</th><th>Wykonane</th></tr>
        </thead>
        <tbody>{ this.todoTableRows(false) }</tbody>
      </table>
      <div className="bg-secondary text-white text-center p-2">
        <VisibilityControl description="wykonane zadania"
          isChecked={this.state.showCompleted}
          callback={(checked) =>
            this.setState({ showCompleted: checked })} />
      </div>

      {this.state.showCompleted &&
        <table className="table table-striped table-bordered">
          <thead>
            <tr><th>Opis</th><th>Wykonane</th></tr>
          </thead>
          <tbody>{this.todoTableRows(true)}</tbody>
        </table>
      }
    </div>
  </div>
}

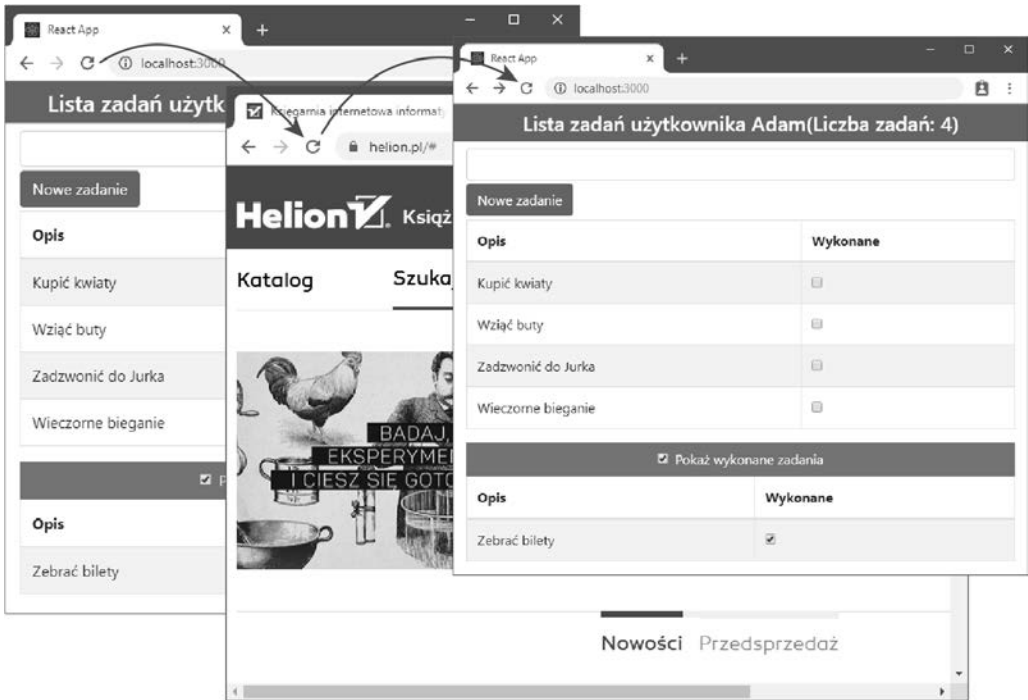
```

Dostęp do API magazynu lokalnego zapewnia obiekt `localStorage`, a komponent używa metody `setItem` do zapisania tablicy z zadaniami w momencie, kiedy użytkownik doda nowe zadanie. Magazyn lokalny umożliwi zapisywanie wyłącznie łańcuchów, dlatego też przed zapisaniem obiektu z danymi zadań muszą zostać serializowane. Do metody `setState` można przekazywać funkcję, która zostanie wywołana po zaktualizowaniu danych stanu (co opiszę dokładniej w rozdziale 11.); użycie tej funkcji zapewnia, że zostaną zachowane aktualne dane.

Komponenty mają precyzyjnie określony cykl życia, który przedstawiłem w rozdziale 13., i umożliwiają implementowanie metod informujących je o ważnych zdarzeniach. Komponent przedstawiony na listingu 1.21 implementuje metodę `componentDidMount`, która jest wywoływana we wczesnym etapie cyklu życia komponentu i zapewnia doskonałą możliwość wykonywania różnych zadań, takich jak wczytywanie danych.

W celu pobrania zapisanych danych użyłem metody `getItem` API magazynu lokalnego, a do zaktualizowania komponentu — metody `setState`, która zapisuje w stanie komponentu dane zachowane w przeglądarce lub domyślny zestaw zadań, jeśli w przeglądarce nie zostały wcześniej zapisane żadne dane.

Te zmiany nie wprowadzają żadnych zauważalnych różnic w wyglądzie aplikacji, niemniej jednak będzie ona teraz trwale przechowywać listę zadań. Oznacza to, że zadania wciąż będą dostępne, kiedy odświeżymy przeglądarkę lub gdy przejdziemy na inną stronę, taką jak strona wydawnictwa Helion, a następnie ponownie wrócimy do aplikacji, wpisując adres `http://localhost:3000/`, jak pokazałem na rysunku 1.9.



Rysunek 1.9. Przechowywanie danych

Podsumowanie

W tym rozdziale przedstawiłem prostą przykładową aplikację, aby pokazać Ci proces tworzenia aplikacji z użyciem frameworka React oraz kilka ważnych pojęć dotyczących tego frameworka. Pokazałem, że tworzenie aplikacji korzystających z Reacta koncentruje się na komponentach, które są definiowane jako pliki JSX zawierające połączenie kodu JavaScript i treści HTML. Podczas tworzenia projektu dysponujemy wszystkim, czego potrzeba do pracy z plikami JSX, budowania aplikacji i dostarczenia jej do przeglądarki w celu testowania; dzięki czemu można zacząć pracę łatwo i szybko.

Dowiedziałeś się także, że aplikacje Reacta mogą zawierać wiele komponentów, z których każdy jest odpowiedzialny za jedną, konkretną możliwość i które otrzymują potrzebne dane i funkcje zwrotne dzięki wykorzystaniu właściwości *props*.

Framework React udostępnia o wiele więcej możliwości, które poznasz w dalszej części książki, jednak prosta aplikacja zamieszczona w tym rozdziale pokazała kluczowe zagadnienia tworzenia aplikacji z użyciem tego frameworka i będzie stanowić podstawę do prac w kolejnych rozdziałach. W następnym rozdziale umieszczę framework React w kontekście oraz opiszę strukturę i zawartość niniejszej książki.

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

React: dynamiczne aplikacje w profesjonalnym wydaniu!

W ostatnich latach JavaScript stał się pełnoprawnym i pełnowartościowym językiem programowania. Jest wszechstronny, elastyczny i pozwala na tworzenie znakomitego kodu. Uzyskanie dobrych efektów wymaga jednak od programisty sporych umiejętności. Wyjątkowo przydatnym rozwiązaniem okazują się frameworki, które ułatwiają pisanie nawet bardzo złożonych aplikacji. Na szczególną uwagę zasługuje React — popularny framework do tworzenia dynamicznych aplikacji w JavaScriptcie. React jest narzędziem, które w stosunkowo prosty sposób pozwala w pełni wykorzystać możliwości nowoczesnych przeglądarek i urządzeń mobilnych.

W książce przystępnie wyjaśniono zasady i techniki programowania w tym frameworku. Opisano szczegóły jego architektury oraz przedstawiono korzyści płynące z pracy z Reactem. Sporo miejsca poświęcono projektowaniu aplikacji oraz korzystaniu zarówno z samego frameworka, jak i towarzyszących mu narzędzi i najpopularniejszych bibliotek. Poszczególne zagadnienia są tu omawiane od podstaw, które następnie płynnie przechodzą do najbardziej zaawansowanych i wyszukanych technik programowania. Wszystkie informacje przedstawiono bardzo starannie, z zachowaniem ważnych szczegółów, dzięki czemu łatwo uzyskać wiedzę niezbędną do pisania aplikacji na wysokim, profesjonalnym poziomie. Nie zabrakło również wskazówek, dzięki którym można łatwo zdiagnozować najczęściej występujące problemy i skutecznie je rozwiązywać.

W tej książce między innymi:

- szczegółowe wyjaśnienie architektury aplikacji Reacta
- tworzenie dynamicznych aplikacji klienta
- korzystanie z magazynu danych Redux
- usługi RESTful i GraphQL
- testowanie i wdrażanie projektów

Adam Freeman — jest doświadczonym programistą, autorem wielu świetnie przyjętych książek o programowaniu w Javie. Tworzył również duże systemy rozproszone (platformy e-commerce). Zajmował stanowiska kierownicze w różnych firmach, wśród których są Netscape, Sun Microsystems, giełda NASDAQ i banki. Jest już na emeryturze, swój czas przeznaczają na pisanie i bieganie na długie dystanse.

 Helion	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS HELIONSZKOLENIA.PL	KOD KORZYŚCI Sięgnij po więcej! ▶  ISBN 978-83-283-6246-8  9 788328 362468
 helion.pl		
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl		
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 119,00 zł

Apress®