

# React, TypeScript i Node

---

Tworzenie aplikacji internetowych typu fullstack

David Choi



Tytuł oryginału: Full-Stack React, TypeScript, and Node: Build cloud-ready web applications using React 17 with Hooks and GraphQL

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-8393-7

Copyright © Packt Publishing 2020. First published in the English language under the title 'Full-Stack React, TypeScript, and Node – (9781839219931)'.

Polish edition copyright © 2022 by Helion S.A.

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 jakiejkolwiek 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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/retyno.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[https://helion.pl/user/opinie/retyno\\_ebook](https://helion.pl/user/opinie/retyno_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](#)

*Mojej pięknej żonie, Eun. Dziękuję Ci za Twoją wiarę i miłość.*

*Udało mi się dzięki Tobie.*

*— David Choi*





# Spis treści

<b>O autorze</b>	<b>11</b>
<b>O recenzencie</b>	<b>11</b>
<b>Wstęp</b>	<b>12</b>
<b>Część I. Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript</b>	<b>17</b>
<b>Rozdział 1. Zrozumieć TypeScript</b>	<b>19</b>
Wymagania techniczne	20
Czym jest TypeScript?	20
Dlaczego TypeScript jest niezbędny?	21
Typowanie dynamiczne a statyczne	23
Programowanie obiektowe	28
Podsumowanie	31
<b>Rozdział 2. Prezentacja języka TypeScript</b>	<b>33</b>
Wymagania techniczne	34
Czym są typy?	34
Jak działają typy?	35
Wprowadzenie do typów języka TypeScript	36
Typ any	36
Typ unknown	37
Typy przecięć i unii	40
Typy literałowe	41
Nazwy zastępcze typów	42

Typy wyników funkcji	42
Funkcje jako typy	44
Typ never	45
<b>Klasy i interfejsy</b>	<b>45</b>
Klasy	46
Interfejsy	52
<b>Dziedziczenie</b>	<b>54</b>
Klasy abstrakcyjne	57
Interfejsy	59
<b>Typy generyczne</b>	<b>61</b>
<b>Prezentacja najnowszych możliwości języka i konfigurowania kompilatora</b>	<b>64</b>
Łączenie opcjonalne	64
Scalanie wartości pustych	65
Konfigurowanie TypeScriptu	66
<b>Podsumowanie</b>	<b>67</b>

## **Rozdział 3. Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript** **68**

<b>Wymagania techniczne</b>	<b>69</b>
<b>Poznanie rodzajów zmiennych w ES6 oraz zasięgów w języku JavaScript</b>	<b>70</b>
<b>Poznanie funkcji strzałkowych</b>	<b>72</b>
<b>Zmienianie kontekstu this</b>	<b>74</b>
<b>Rozproszenie, destrukuryzacja i reszta</b>	<b>76</b>
Rozproszenie, Object.assign oraz Array.concat	77
Destrukuryzacja	79
Reszta	80
<b>Prezentacja wybranych funkcji tablicowych</b>	<b>81</b>
find	81
filter	82
map	83
reduce	84
some oraz every	85
<b>Prezentowanie nowych typów kolekcji</b>	<b>86</b>
Set	86
Map	87
<b>Prezentowanie słów kluczowych async i await</b>	<b>88</b>
<b>Podsumowanie</b>	<b>93</b>

## **Część II. Nauka tworzenia aplikacji jednostronicowych z użyciem frameworka React** **95**

### **Rozdział 4. Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React** **97**

<b>Wymagania techniczne</b>	<b>98</b>
<b>Prezentowanie wcześniejszych sposobów tworzenia witryny WWW</b>	<b>98</b>
<b>Cechy i zalety aplikacji jednostronicowych</b>	<b>100</b>

<b>Jak React pomaga w tworzeniu aplikacji jednostronicowych</b>	<b>101</b>
Atrybuty aplikacji Reacta	102
<b>Podsumowanie</b>	<b>113</b>
<b>Rozdział 5. Tworzenie aplikacji Reacta z wykorzystaniem hooków</b>	<b>114</b>
<b>Wymagania techniczne</b>	<b>115</b>
<b>Wyjaśnienie ograniczeń i problemów</b>	
związanych ze stosowaniem starych komponentów klasowych	115
Stan	116
Metody cyklu życia	117
<b>Prezentacja hooków Reacta i wyjaśnienie,</b>	
<b>    dlaczego w stosunku do komponentów klasowych są one usprawnieniem</b>	<b>132</b>
<b>Porównanie stosowania komponentów klasowych i hooków</b>	<b>144</b>
Wielokrotne stosowanie kodu	145
Prostota	145
<b>Podsumowanie</b>	<b>146</b>
<b>Rozdział 6. Przygotowywanie projektu za pomocą create-react-app</b>	
<b>i testowanie go przy użyciu Jest</b>	<b>147</b>
<b>Wymagania techniczne</b>	<b>148</b>
<b>Przedstawienie metod programowania aplikacji Reacta</b>	
<b>    i systemu używanego do ich budowania</b>	<b>148</b>
Narzędzia do zarządzania projektami	149
Transpilacja	156
Repozytoria kodu	158
<b>Testowanie aplikacji Reacta po stronie klienta</b>	<b>160</b>
<b>Atrapy</b>	<b>172</b>
Tworzenie atrap z wykorzystaniem jest.fn	173
Tworzenie atrap komponentów	178
<b>Prezentacja najpopularniejszych narzędzi oraz praktyk tworzenia aplikacji Reacta</b>	<b>185</b>
Visual Studio Code	185
Prettier	186
Debugger Chrome	187
Alternatywne zintegrowane środowiska programistyczne	190
<b>Podsumowanie</b>	<b>191</b>
<b>Rozdział 7. Redux i React Router</b>	<b>192</b>
<b>Wymagania techniczne</b>	<b>192</b>
<b>Zarządzanie stanem przy użyciu Reduxa</b>	<b>193</b>
Reduktory i akcje	195
React Context	205
<b>Prezentacja frameworka React Router</b>	<b>212</b>
<b>Podsumowanie</b>	<b>221</b>

## Część III. Tworzenie usług internetowych z użyciem Expressa i GraphQL-a 223

### Rozdział 8. Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa 225

Wymagania techniczne	226
Wyjaśnienie sposobu działania środowiska Node	226
Pętla zdarzeń	228
Prezentacja możliwości środowiska Node	229
Instalowanie Node	229
Tworzenie prostego serwera Node	233
Żądania i odpowiedzi	236
Trasowanie	239
Debugowanie	241
Jak Express ułatwia pisanie rozwiązań przeznaczonych dla środowiska Node	248
Przedstawienie możliwości frameworka Express	250
Tworzenie internetowego API przy użyciu Expressa	256
Podsumowanie	259

### Rozdział 9. Czym jest GraphQL? 260

Wymagania techniczne	260
Czym jest GraphQL?	261
Schematy GraphQL	263
Definicje typów i resolwery	264
Zapytania, mutacje oraz subskrypcje	270
Podsumowanie	277

### Rozdział 10. Konfiguracja projektu Expressa z zależnościami od języków TypeScript i GraphQL 278

Wymagania techniczne	279
Tworzenie projektu Expressa tworzonego w języku TypeScript	279
Dodawanie do projektu GraphQL-a i jego zależności	283
Prezentacja pakietów pomocniczych	290
Podsumowanie	292

### Rozdział 11. Czego się nauczysz — aplikacja internetowego forum 293

Analiza aplikacji, którą napiszemy — internetowego forum	294
Analiza uwierzytelniania użytkowników forum	295
Analiza zarządzania wątkami	296
Analiza systemu punktacji wątków	297
Podsumowanie	298

**Rozdział 12. Tworzenie klienta Reacta na potrzeby aplikacji internetowego forum** 299

<b>Wymagania techniczne</b>	<b>299</b>
<b>Tworzenie wstępnej wersji aplikacji Reacta</b>	<b>300</b>
CSS Grid	302
Granice błędów	309
Warstwa usługi danych	311
Menu nawigacyjne	313
Komponenty związane z uwierzytelnianiem	318
Trasowanie i ekrany aplikacji	328
Ekran główny	329
Ekran wątku i jego wpisów	341
<b>Podsumowanie</b>	<b>360</b>

**Rozdział 13. Przygotowywanie stanu sesji przy użyciu Expressa i Redisa** 361

<b>Wymagania techniczne</b>	<b>362</b>
<b>Czym jest stan sesji?</b>	<b>362</b>
<b>Prezentowanie magazynu danych Redis</b>	<b>363</b>
<b>Tworzenie stanu sesji z wykorzystaniem Expressa i Redisa</b>	<b>369</b>
<b>Podsumowanie</b>	<b>375</b>

**Rozdział 14. Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM** 376

<b>Wymagania techniczne</b>	<b>377</b>
<b>Przygotowanie bazy danych Postgres</b>	<b>377</b>
<b>Prezentowanie mechanizmów odwzorowań obiektowo-relacyjnych na przykładzie TypeORM</b>	<b>382</b>
<b>Tworzenie warstwy repozytorium bazującej na Postgresie i TypeORM</b>	<b>383</b>
<b>Podsumowanie</b>	<b>413</b>

**Rozdział 15. Dodawanie schematu GraphQL-a — część 1.** 414

<b>Wymagania techniczne</b>	<b>414</b>
<b>Tworzenie definicji typów i resolverów dla serwerowego kodu GraphQL</b>	<b>415</b>
System punktacji wątków	424
<b>Integracja mechanizmu uwierzytelniania z resolverami GraphQL-a</b>	<b>428</b>
<b>Przygotowanie hooków Reacta do korzystania z serwera Apollo GraphQL</b>	<b>432</b>
Ekran główny — komponent Main	435
Możliwości związane z uwierzytelnianiem	447
Ekran profilu użytkownika	454
<b>Podsumowanie</b>	<b>460</b>

**Rozdział 16. Dodawanie schematu GraphQL-a — część 2.** 461

<b>Komponent Thread i jego trasa</b>	<b>461</b>
<b>System punktów</b>	<b>471</b>
<b>Podsumowanie</b>	<b>511</b>

<b>Rozdział 17. Wdrażanie w chmurze AWS</b>	<b>512</b>
<b>Wymagania techniczne</b>	<b>513</b>
<b>Konfiguracja Ubuntu w chmurze AWS</b>	<b>513</b>
<b>Instalacja Redisa, Postgresa i Node w systemie Ubuntu</b>	<b>520</b>
Instalacja serwera Redis	521
Instalacja Postgresa	522
Instalacja Node	524
<b>Konfiguracja i wdrażanie aplikacji na serwerze NGINX</b>	<b>525</b>
Konfigurowanie projektu super-forum-client	532
Konfiguracja serwera NGINX	534
Rozwiązywanie problemów	542
<b>Podsumowanie</b>	<b>543</b>

# 0 autorze

David Choi jest programistą z ponad 10-letnim doświadczeniem w tworzeniu aplikacji korporacyjnych z wykorzystaniem wielu frameworków i w wielu językach programowania. Większość swoich zawodowych doświadczeń zyskał pracując nad zagadnieniami finansowymi w takich firmach, jak JPMorgan, CSFB oraz Franklin Templeton. Aktualnie pracuje nad swoim własnym start-upem, DzHaven, aplikacją, która ma pomagać programistom w pomaganiu innym programistom.

Davida można znaleźć na YouTube — na kanale David Choi — oraz na Twitterze: *@jsoneday*.

*Chciałbym podziękować wspinałym pracownikom wydawnictwa Packt oraz Mike'owi Rourke, bez których pomocy napisanie tej książki nie byłoby możliwe.*

# 0 recenzencie

**Mike Rourke** jest inżynierem oprogramowania zamieszkałym w Chicago, korzystającym głównie z technologii internetowych i ekosystemu Node.js. Zajmuje się pisaniem kodu już od ponad 10 lat. Mike zaczynał od pisania w VisualBasicu i około 2 lat temu zaczął używać języka JavaScript. Kocha wszystkie aspekty programowania i większość swojego wolnego czasu poświęca na poznawanie nowych technologii i doskonalenie swoich umiejętności.

# Wstęp

Zgodnie z danymi publikowanymi przez GitHub, największe repozytorium kodów typu *open source*, JavaScript wciąż jest najpopularniejszym językiem programowania na świecie. W tym języku jest pisanych więcej projektów niż w jakimkolwiek innym. Nawet projekty, których normalnie nie kojarzono by z internetem, takie jak projekty związane z uczeniem maszynowym lub kryptowalutami, często są pisane w JavaScriptcie.

Jako język programowania JavaScript jest niezwykle potężny i zapewnia ogromne możliwości; jednak oprócz samego języka istnieje także wiele frameworków, takich jak React czy Node, które rozszerzają możliwości języka i sprawiają, że staje się on jeszcze lepszy. A obecnie dochodzi jeszcze do tego TypeScript, który stał się standardem tworzenia dużych projektów w języku JavaScript. Dostarcza on możliwości, które sprawiają, że pisanie kodu JavaScript staje się bardziej efektywne i lepiej dostosowane do potrzeb tworzenia dużych aplikacji.

W ciągu kilku ostatnich lat techniki tworzenia nowoczesnych aplikacji internetowych niezwykle się rozwinęły. W przeszłości klienckie części aplikacji składały się niemal wyłącznie ze statycznego kodu HTML i CSS, i być może niewielkich fragmentów kodu JavaScript. Z kolei serwerowe części aplikacji niemal zawsze były w całości pisane w odrębnych językach programowania, takich jak PHP, lub tworzone przy użyciu skryptów CGI. Obecnie coraz częściej zdarza się, że całe aplikacje, zarówno części klienckie, jak i serwerowe, są pisane w JavaScriptcie oraz frameworkach tworzonych w tym języku. Ta możliwość pisania całych aplikacji w jednym języku zapewnia ogromne korzyści. Co więcej, solidne i dojrzałe frameworki, które są dziś dostępne, sprawiają, że pisanie w JavaScriptcie kompletnych aplikacji, obejmujących cały stos technologiczny, staje się obecnie konkurencyjne, a może nawet bardziej atrakcyjne, niż w innych językach i z użyciem innych platform.

Z tej książki dowiesz się, jak wykorzystać możliwości języka JavaScript do stworzenia kompletnej aplikacji internetowej. Rozszerzymy te możliwości o wykorzystanie TypeScriptu, kolejnego języka należącego do grupy 10 najpopularniejszych języków programowania. Następnie, używając frameworków takich jak React, Redux, Node, Express i GraphQL zbudujemy realistyczną aplikację internetową o rozbudowanych możliwościach, na przykładzie której zdobędziesz całą wiedzę potrzebną do tworzenia nowoczesnych aplikacji internetowych



obejmujących cały stos technologiczny, określanych także jako aplikacje typu *fullstack*. A kiedy skończymy prace nad aplikacją, wdrożymy ją w chmurze AWS — aktualnie najpopularniejszej i oferującej najbardziej rozbudowane możliwości usłudze chmurowej.

## Dla kogo jest przeznaczona ta książka

Niniejsza książka jest przeznaczona dla programistów, którzy chcieliby wyjść poza tworzenie klasycznych aplikacji internetowych i rozpocząć tworzenie rozwiązań obejmujących pełen stos technologiczny, opanować nowoczesne technologie internetowe i nauczyć się łączyć je w jedną całość. Czytelnik sięgający po tę książkę musi dysponować dobrą znajomością języka JavaScript.

## Zakres tematyczny książki

Rozdział 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript”, wyjaśnia czym jest język TypeScript i co sprawia, że idealnie się on nadaje do tworzenia dużych aplikacji.

Rozdział 2., pt. „Prezentacja języka TypeScript”, zawiera wprowadzenie w tajniki języka TypeScript. Poznasz w nim jego możliwości, w tym statyczną kontrolę typów, i dowiesz się, dlaczego są one usprawnieniem w stosunku do JavaScriptu. Przyjrzymy się w nim także zagadnieniom projektowania aplikacji tworzonych w językach obiektowych i zobaczymy, dlaczego cechy TypeScriptu ułatwiają stosowanie tego ważnego paradygmatu programowania.

Rozdział 3., pt. „Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript”, zawiera przegląd najważniejszych możliwości języka JavaScript, które powinien znać każdy programista. Skoncentrujemy się w nim na najważniejszych możliwościach dodanych w wersji ES6 (i nowszych) języka.

Rozdział 4., pt. „Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React”, wyjaśnia, jak są tworzone witryny WWW, a w szczególności opisuje sposoby tworzenia aplikacji jednostronicowych. Przedstawię tu także framework React i wyjaśnię, w jaki sposób można go używać do tworzenia aplikacji jednostronicowych.

Rozdział 5., pt. „Tworzenie aplikacji React z wykorzystaniem hooków”, zawiera dokładniejszą prezentację sposobów stosowania Reacta, jak również wyjaśnia, czym są punkty zaczepienia, komponenty funkcyjne, i dlaczego są one lepsze od stosowanych wcześniej komponentów klasowych.

Rozdział 6., pt. „Przygotowywanie projektu za pomocą create-react-app i testowanie go przy użyciu Jest”, wyjaśnia nowoczesne sposoby tworzenia aplikacji przy użyciu frameworka React. Poznasz w nim również standardowe narzędzie do tworzenia aplikacji Reacta, create-react-app, a także sposoby testowania aplikacji klienckich przy użyciu bibliotek Jest i testing-library.

Rozdział 7., pt. „Redux i React Router”, opisuje frameworki Redux oraz React Router i wyjaśnia, jak można ich używać do tworzenia aplikacji Reacta. Te dwa frameworki służą odpowiednio do zarządzania stanem oraz obsługi routowania w aplikacjach Reacta.

Rozdział 8., pt. „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa”, opisuje środowisko Node oraz framework Express. Node jest środowiskiem uruchomieniowym umożliwiającym pisanie w języku JavaScript aplikacji serwerowych. Z kolei Express jest frameworkiem przeznaczonym dla środowiska Node, ułatwiającym tworzenie potężnych aplikacji serwerowych.

Rozdział 9., pt. „Czym jest GraphQL?”, zawiera wprowadzenie wyjaśniające czym jest GraphQL i dlaczego do tworzenia internetowych API korzysta on ze schematów.

Rozdział 10., pt. „Konfiguracja projektu Expressa z zależnościami od języków TypeScript i GraphQL”, wyjaśnia, jak można używać języka TypeScript, frameworka Express, GraphQL-a oraz biblioteki Jest do tworzenia projektów serwerowych o produkcyjnej jakości.

Rozdział 11., pt. „Czego się nauczysz — aplikacja internetowego forum”, opisuje aplikację, którą stworzymy w dalszej części książki. Opiszę w nim jej możliwości i wyjaśnię, w jaki sposób napisanie takiej aplikacji pomoże bardziej szczegółowo i dokładnie poznać sposoby tworzenia aplikacji internetowych.

Rozdział 12., pt. „Tworzenie klienta Reacta na potrzeby aplikacji internetowego forum”, pokazuje, w jaki sposób należy rozpoczynać tworzenie aplikacji internetowych korzystających z frameworka React. Do tworzenia poszczególnych ekranów aplikacji będziemy stosować komponenty funkcyjne, punkty zaczepienia oraz framework Redux.

Rozdział 13., pt. „Przygotowywanie stanu sesji przy użyciu Expressa i Redisa”, wyjaśnia czym jest stan sesji oraz w jaki sposób można go utworzyć przy wykorzystaniu magazynu Redis — najpotężniejszej obecnie bazy danych przechowywanej w pamięci operacyjnej. Zaczniemy w nim także tworzyć serwerową część naszej aplikacji, używając do tego celu frameworka Express.

Rozdział 14., pt. „Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM”, wyjaśnia, w jaki sposób przygotować bazę danych Postgres na potrzeby tworzonej aplikacji oraz w jaki sposób korzystać z niej w oparciu o potężną technikę projektowania aplikacji, nazywaną *warstwą repozytorium*.

Rozdział 15., pt. „Dodawanie schematu GraphQL — część 1.”, rozpoczyna integrowanie GraphQL-a z tworzoną aplikacją. Stworzymy w niej schemat i dodamy do niego zapytania i mutacje, zaczniemy także dodawać punkty zaczepienia GraphQL-a do kodu klienckiej części aplikacji.

Rozdział 16., pt. „Dodawanie schematu GraphQL — część 2.”, prezentuje dokończenie prac nad aplikacją, w tym zintegrowanie jej w całości z GraphQL-em — i to zarówno po stronie klienta, jak i serwera.

Rozdział 17., pt. „Wdrażanie w chmurze AWS”, opisuje sposób wdrażania gotowej aplikacji w chmurze AWS. Stworzymy w tym celu wirtualny serwer działający pod kontrolą Linuksa, a konkretnie dystrybucji Ubuntu, i udostępnimy na nim naszą aplikację, korzystając z serwera NGINX.

# Co zrobić, by jak najbardziej skorzystać na lekturze tej książki?

Powinieneś mieć przynajmniej rok doświadczeń w programowaniu w przynajmniej jednym nowoczesnym języku programowania oraz wiedzę z zakresu tworzenia aplikacji, choć nie muszą to być aplikacje internetowe.

Oprogramowanie i komponenty sprzętowe opisywane w tej książce	Wymagania systemowe
React 17 (cały prezentowany kod jest zgodny z frameworkiem React 16.x)	Windows, macOS X, Linux (dowolny)
TypeScript 3.7 lub nowszy	
Nowoczesna przeglądarka WWW: Chrome, Safari lub Firefox	
Node 12 lub nowszy	

W książce prezentowane są szczegółowe instrukcje, krok po kroku prezentujące sposób instalacji wszystkich zależności. Jednak powyższe zestawienie daje ogólny pogląd na to, co będzie potrzebne. Kody źródłowe dołączone do książki zawierają końcową postać aplikacji dla każdego z rozdziałów książki.

Jeśli używasz cyfrowej wersji książki, radzę, byś wpisywał kody samodzielnie lub pobrał je z serwera wydawnictwa Helion (odnośnik znajdziesz w następnym punkcie). Dzięki temu unikniesz problemów wynikających z ewentualnych błędów, które mogą się pojawić podczas kopiowania i wklejania kodu.

Optymalnie jednak będzie, jeśli zdecydujesz się wpisywać kod samodzielnie, gdyż nie tylko ułatwi Ci to jego zapamiętanie, lecz także zapewni doświadczenia w rozwiązywaniu problemów, kiedy coś pójdzie źle.

## Przykładowe kody do książki

Kody źródłowe przykładów prezentowanych w książce można pobrać z serwera wydawnictwa Helion, <https://ftp.helion.pl/przyklady/retyno.zip>. Niespolonizowane kody źródłowe można także pobrać z repozytorium w serwisie GitHub, <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. Gdyby konieczne było wprowadzenie aktualizacji w kodach, to będą one publikowane w serwisie GitHub.

## Zastosowane konwencje

W tej książce zastosowanych zostało kilka konwencji typograficznych.

Kod w tekście — w ten sposób są oznaczane fragmenty kodu umieszczane wewnątrz akapitów tekstu: słowa kluczowe, nazwy tabel, klas, zmiennych. Z kolei *kursywą* są oznaczane nazwy katalogów, plików, rozszerzeń, adresy URL, nazwy użytkowników Twittera. Oto przykład: „W katalogu *src* utwórz nowy plik, *Home.tsx*, a następnie zapisz w nim kod komponentu *Home*”. Kursywa jest także używana do prezentowania wszelkich opcji i elementów interfejsu użytkownika. Z kolei **pogrubienia** używamy do wyróżniania nowych, ważnych terminów.

Poniżej przedstawiony został sposób prezentowania bloków kodu:

```
let a = 5;
let b = '6';
console.log(a + b);
```

W sytuacjach, w których chcę zwrócić uwagę Czytelnika na konkretny fragment kodu, pewien wiersz lub fragment wiersza, będę go oznaczać pogrubieniem:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,VoiceMail(u100)
exten => s,102,VoiceMail(b100)
exten => i,1,VoiceMail(s0)
```

Wszelkie komendy wpisywane w wierszu poleceń oraz ich wyniki są prezentowane w następujący sposób:

```
npm install typescript
```

Tak są prezentowane ważne uwagi i ostrzeżenia.



C Z Ę Ś Ć

# Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript

Ta część książki przedstawia korzyści, jakie daje stosowanie języka TypeScript, oraz jego najważniejszych możliwości. Przedstawię w niej także najważniejsze cechy języka ES6 oraz wyjaśnię, w jaki sposób mogą one wpłynąć na poprawę jakości i przejrzystości kodu.

Ta część książki składa się z następujących rozdziałów:

- Rozdział 1. — „Zrozumieć TypeScript”;
- Rozdział 2. — „Prezentacja języka TypeScript”;
- Rozdział 3. — „Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript”.



# Zrozumieć TypeScript

JavaScript jest niezwykle popularnym językiem programowania o bardzo dużych możliwościach. Dane serwisu GitHub pokazują, że jest on obecnie najpopularniejszym językiem na świecie (owszem, jest stosowany nawet częściej niż Python), a nowe cechy wprowadzone w wersji ES6+ dodatkowo wzbogaciły go o użyteczne możliwości. Niemniej jednak, z punktu widzenia tworzenia dużych aplikacji, zbiór możliwości JavaScriptu jest niekompletny. I właśnie to było powodem opracowania języka TypeScript.

W tym rozdziale dowiesz się czym jest TypeScript, dlaczego go stworzono i dlaczego jest przydatny dla programistów JavaScriptu. Wyjaśnię filozofię projektową, jaką kierowali się twórcy języka z firmy Microsoft, wytłumaczę też, dlaczego podjęte przez nich decyzje projektowe sprawiają, że język ten znacznie lepiej nadaje się do tworzenia dużych aplikacji.

Wyjaśnię także, w jaki sposób TypeScript rozszerza i poprawia możliwości języka JavaScript. Porównam sposoby, w jakie pisze się kod w językach JavaScript i TypeScript. TypeScript zapewnia bardzo wiele nowoczesnych rozwiązań zapewniających znaczące korzyści dla programistów. Do najważniejszych spośród nich należą statyczna kontrola typów oraz różne możliwości związane z **programowaniem obiektowym**. Mogą się one znacząco przyczynić do poprawy jakości kodu oraz ułatwienia jego przyszłej pielęgnacji.

Pod koniec tego rozdziału będziesz już rozumiał niektóre ograniczenia języka JavaScript, które sprawiają, że jego stosowanie w dużych projektach przysparza sporych problemów. Zrozumiesz także, w jaki sposób TypeScript rozwiązuje niektóre spośród tych problemów i sprawia, że pisanie dużych, złożonych aplikacji staje się łatwiejsze i mniej podatne na błędy.

W tym rozdziale zajmiemy się:

- przedstawieniem TypeScriptu;
- wyjaśnieniem, dlaczego TypeScript jest niezbędny.

## Wymagania techniczne

Aby w pełni skorzystać z informacji zamieszczonych w tym rozdziale, będziesz musiał dysponować podstawową znajomością języka JavaScript w wersji ES5 lub nowszej oraz pewnym doświadczeniem w tworzeniu aplikacji internetowych z wykorzystaniem frameworków języka JavaScript. Oprócz tego będziesz musiał zainstalować Node oraz edytor pozwalający na pisanie kodu w JavaScriptcie, taki jak **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial01 (Chap1)*.

## Czym jest TypeScript?

TypeScript to w rzeczywistości dwie odrębne, lecz powiązane ze sobą technologie: język oraz kompilator.

- Język ma bogate możliwości, jest wyposażony w statyczną kontrolę typów i wzbogaca JavaScript o pełne możliwości programowania obiektowego.
- Kompilator konwertuje kod napisany w TypeScriptie na rodzimy kod JavaScript, lecz jednocześnie ułatwia programistom pisanie kodu mniej podatnego na błędy.

TypeScript pozwala projektować oprogramowanie o wyższej jakości. Połączenie języka z kompilatorem zwiększa możliwości programistów. Używając TypeScriptu możemy pisać kod, który będzie łatwiejszy do zrozumienia i refaktoryzacji, a jednocześnie mniej narażony na występowanie błędów. Co więcej, stosowanie TypeScriptu wymusza wprowadzenie do używanego toku pracy dyscypliny, gdyż zmusza do poprawiania błędów jeszcze na etapie pisania kodu.

TypeScript jest językiem czasu programowania. Nie dysponuje on żadnym komponentem wykonawczym, a żaden kod napisany w tym języku nigdy nie jest wykonywany na jakichkolwiek komputerach. Zamiast tego, kompilator TypeScriptu konwertuje kod napisany w tym języku na kod JavaScript, który jest następnie wdrażany i wykonywany w przeglądarkach i na serwerach. Istnieje jednak możliwość, że firma Microsoft kiedyś opracuje środowisko wykonawcze dla TypeScriptu. Jednak, w odróżnieniu od rynku systemów operacyjnych, Microsoft nie ma kontroli nad organizacją zajmującą się standaryzacją języka ECMAScript



(tworzy ją grupa osób decydująca, jakie możliwości będą dostępne w poszczególnych wersjach JavaScriptu), a zdobycie takiej kontroli byłoby trudne i czasochłonne. Dlatego też firma Microsoft zdecydowała się na stworzenie narzędzia, które poprawi wydajność pracy programistów używających JavaScriptu i jakość tworzonego przez nich kodu.

A zatem, skoro TypeScript nie ma żadnego środowiska wykonawczego, w jaki sposób programiści mogą wykonywać kod pisany w tym języku? Otóż TypeScript korzysta z procesu nazywanego **transpilacją** (ang. *transpilation*). Transpilacja to proces polegający na „skompileowaniu”, czy też skonwertowaniu kodu napisanego w jednym języku programowania na kod napisany w innym języku. Oznacza to, że cały kod pisany w TypeScriptie przed jego wdrożeniem i wykonaniem, jest konwertowany na kod JavaScript.

Z tego podrozdziału dowiedziałeś się czym jest oraz jak działa TypeScript, w następnym wyjaśnię, dlaczego możliwości, które oferuje, są niezbędne do pisania dużych i złożonych aplikacji.

## Dlaczego TypeScript jest niezbędny?

Język JavaScript został stworzony w 1995 roku przez Brendana Eicha i w tym samym roku dodano go do przeglądarki Netscape. Od tego czasu JavaScript odniósł spektakularny sukces, a obecnie używa się go zarówno do tworzenia aplikacji serwerowych, jak i klienckich. Jednak jego popularność i wszechobecność to jednocześnie zaleta, jak i problem, bowiem wraz z powiększaniem się oraz wzrostem złożoności tworzonych aplikacji, programiści zaczęli dostrzegać także ograniczenia JavaScriptu.

Wymagania związane z tworzeniem dużych aplikacji są znacznie większe od potrzeb występujących podczas pisania małych, działających w przeglądarkach skryptów, z myślą o których JavaScript początkowo był projektowany. Na najwyższym poziomie, niemal wszystkie języki programowania używane do tworzenia dużych aplikacji, takie jak Java, C++ oraz C#, zapewniają statyczną kontrolę typów oraz pełne możliwości programowania obiektowego. W tym podrozdziale przedstawię zalety płynące ze statycznej kontroli typów w porównaniu z dynamicznym typowaniem charakterystycznym dla JavaScriptu. Zajmiemy się także programowaniem obiektowym i wyjaśnię, dlaczego mechanizmy obiektowości języka JavaScript są zbyt ograniczone na potrzeby dużych aplikacji.

Jednak zanim zajmiemy się tymi wszystkimi zagadnieniami, będziesz musiał zainstalować na swoim komputerze kilka pakietów i programów niezbędnych do wykonywania przykładów prezentowanych w tekście. W tym celu postępuj zgodnie z poniższymi instrukcjami:

1. W pierwszej kolejności zainstaluj środowisko Node. Możesz je pobrać z witryny <https://nodejs.org/>. W skład Node wchodzi program npm — menedżer zależności i pakietów, którego użyjemy do zainstalowania języka TypeScript. Środowiskiem Node zajmiemy się dokładniej w rozdziale 8., pt. „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa”.

2. Zainstaluj program VSCode. To dostępny bezpłatnie edytor do tworzenia kodu, którego bogaty zestaw możliwości błyskawicznie sprawił, że stał się on standardowym środowiskiem do pisania aplikacji w języku JavaScript na wszystkich platformach systemowych.
3. W swoim katalogu domowym utwórz podkatalog *NaukaTypeScriptu* — będziesz w nim zapisywał kody przykładów prezentowanych w tym rozdziale.

Jeśli nie chce Ci się samodzielnie wpisywać wszystkich kodów, to możesz je pobrać z internetu, zgodnie z informacjami podanymi w podrozdziale pt. „Wymagania techniczne”.

4. Przejdź do katalogu *NaukaTypeScriptu* i utwórz w nim podkatalog *rozdzial01*.
5. Uruchom program VSCode i wybierz opcje *File/Open*, a następnie otwórz utworzony przed chwilą katalog *rozdzial01*. Następnie wybierz z menu opcje *View/Terminal* — w efekcie, wewnątrz okna VSCode zostanie wyświetlony panel terminala.
6. W panelu terminala wpisz następujące polecenie:

**npm init**

Spowoduje ono zainicjowanie projektu, tak by można było dodawać do niego zależności npm. Panel terminala powinien teraz wyglądać jak na rysunku 1.1.

```

H:\NaukaTypeScriptu\rozdzial01>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (rozdzial01)

```

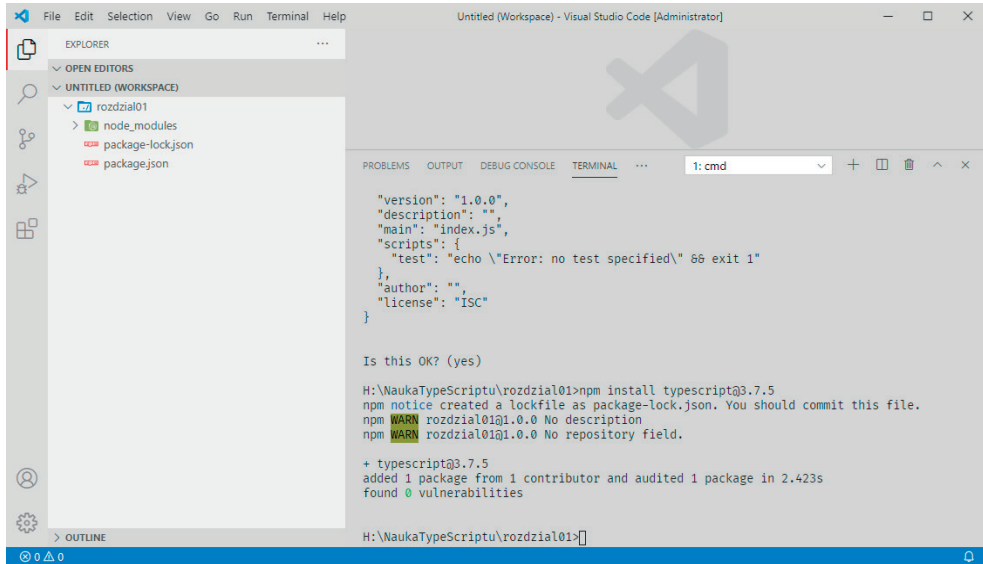
**Rysunek 1.1.** Komunikaty wyświetlane podczas inicjalizacji projektu npm

Na wszystkie pytania możesz odpowiedzieć akceptując wartości domyślne, gdyż na razie ograniczymy się do zainstalowania jedynie języka TypeScript.

7. Wykonaj poniższe polecenie, aby zainstalować język TypeScript:

**npm install typescript**

Po zainstalowaniu wszystkich niezbędnych elementów, okno VSCode powinno wyglądać jak na rysunku 1.2.



Rysunek 1.2. Okno VSCode po zakończeniu inicjalizacji projektu

W ten sposób zakończyliśmy instalowanie i konfigurowanie środowiska. Możemy zatem przyjrzeć się paru przykładom, które pozwolą Ci lepiej zrozumieć zalety stosowania języka TypeScript.

## Typowanie dynamiczne a statyczne

Każdy język programowania korzysta z pojęcia typów. Typ to po prostu zestaw reguł opisujących obiekt oraz możliwości jego wykorzystania. JavaScript jest językiem korzystającym z typowania dynamicznego. W tym języku nie ma potrzeby deklarowania typów nowych zmiennych, ani nawet określania ich podczas przypisywania zmiennym wartości — zmiennym można bowiem w dowolnym momencie przypisać wartość jakiegoś innego typu. Te możliwości zapewniają fantastyczną elastyczność języka, lecz jednocześnie stanowią przyczynę bardzo wielu błędów.

Z kolei TypeScript korzysta z lepszego rozwiązania, nazywanego **typowaniem statycznym** (ang. *static typing*). W tym przypadku programista musi z góry, podczas tworzenia zmiennej, określić jej typ. W ten sposób usuwa się niejednoznaczności i eliminuje wiele błędów związanych z konwersją danych. W kilku przedstawionych poniżej przykładach przedstawię problemy związane z dynamicznym typowaniem i pokażę, jak statyczne typowanie stosowane w TypeScriptie pozwala je wyeliminować:

1. W katalogu *rozdzial01* utwórz plik o nazwie *strings-vs-number.ts*. Rozszerzenie *.ts* jest charakterystyczne dla plików w języku TypeScript. Jego zastosowanie pozwala kompilatorowi TypeScriptu rozpoznawać pliki źródłowe napisane w tym języku i transpilować je do plików w języku JavaScript. Po utworzeniu pliku wpisz w nim poniższy fragment kodu:

```
let a = 5;
let b = '6';

console.log(a + b);
```

2. Teraz, w panelu terminala, wykonaj następujące polecenie:

```
tsc string-vs-number.ts
```

tsc to polecenie, które powoduje uruchomienie kompilatora języka TypeScript, natomiast nazwa pliku nakazuje kompilatorowi sprawdzić i transpilować wskazany plik.

3. Po wykonaniu polecenia tsc w katalogu powinien pojawić się nowy plik: *string-vs-number.js*. Spróbujmy go wykonać:

```
node string-vs-number.js
```

Polecenie node rozpoczyna działanie środowiska uruchomieniowego, w którym może zostać wykonany plik JavaScript. Rozwiązanie to działa, gdyż do wykonywania kodu JavaScript Node używa silnika JavaScript V8, opracowanego przez Google i używanego w przeglądarce Chrome. Po uruchomieniu skryptu w panelu terminala powinny zostać wyświetlone następujące wyniki:

```
56
```

Oczywiście, kiedy dodajemy dwie cyfry, to zazwyczaj chodzi nam o wyliczenie ich sumy, a nie o konkatencję łańcuchów znaków. Jednak środowisko uruchomieniowe JavaScriptu nie zna naszych zamiarów, próbuje zatem je odgadnąć i w efekcie konwertuje zmienną a zawierającą liczbę na łańcuch, który następnie łączy z łańcuchem zapisanym w zmiennej b. Można sądzić, że wystąpienie takiej sytuacji w realnym kodzie jest raczej mało prawdopodobne, niemniej jednak, jeśli nie sprawdzimy kodu, to faktycznie może ona wystąpić, gdyż w przypadku aplikacji internetowych znaczna większość danych wejściowych to właśnie łańcuchy znaków — i to nawet jeśli użytkownik wpisuje je jako liczby.

4. A teraz zastosujmy w tym kodzie mechanizmy statycznego typowania języka TypeScript i zobaczmy, co się stanie. W pierwszej kolejności usuń plik *.js*, gdyż w przeciwnym przypadku kompilator TypeScriptu może uznać, że istnieją dwie kopie zmiennych a i b. Następnie przyjrzyj się dokładniej poniższemu fragmentowi kodu:

```
let a: number = 5;
let b: number = '6';

console.log(a + b);
```

5. Jeśli spróbujesz wykonać polecenie tsc na pliku z powyższym kodem, zostanie wyświetlony błąd Type '"6"' is not assignable to type 'number'. I to jest dokładnie to, o co nam chodzi. Kompilator poinformował nas, że w kodzie jest błąd, i nie odpuścił do skompilowania kodu. Ponieważ zazaczyliśmy, że obie zmienne mają być liczbami, kompilator sprawdził przypisywane im wartości i zgłosił błąd, kiedy wykrył, że jednej ze zmiennych przypisywana jest wartość, która liczbą nie jest. Spróbuj zatem poprawić ten błąd, tak by w zmiennej b była zapisywana liczba, a następnie sprawdź, co się stanie:

```
let a: number = 5;
let b: number = 6;

console.log(a + b);
```

6. Jeśli teraz uruchomisz kompilator, to kod zostanie przetworzony prawidłowo, a wykonanie wygenerowanego pliku JavaScript spowoduje wyświetlenie wyniku 11, jak pokazałem na rysunku 1.3.

```
H:\NaukaTypeScriptu\rozdzial01>tsc string-vs-number.ts
H:\NaukaTypeScriptu\rozdzial01>node string-vs-number.js
11
H:\NaukaTypeScriptu\rozdzial01>
```

**Rysunek 1.3.** Prawidłowe dodawanie liczb

I świetnie — kiedy błędnie przypisaliśmy zmiennej `b` nieprawidłową wartość, TypeScript wychwycił nasz błąd i nie dopuścił do skompilowania i wykonania kodu.

Przeanalizujmy teraz nieco bardziej skomplikowany przykład, gdyż jest on bardziej zbliżony do tego, co w rzeczywistości możemy zobaczyć w kodzie większych aplikacji:

1. Utwórz nowy plik `.ts` o nazwie `test-age.ts` i zapisz w nim następujący kod:

```
function canDrive(usr) {
    console.log("imię użytkownika: ", usr.name);

    if(usr.age >= 16) {
        console.log("może prowadzić auto");
    } else {
        console.log("nie ma prawa kierować");
    }
}

const tom = {
    name: "Tomek"
}

canDrive (tom);
```

Jak widać, kod rozpoczyna się od funkcji, która sprawdza wiek użytkownika i na jego podstawie określa, czy dany użytkownik może prowadzić auto, czy nie. Poniżej definicji funkcji stworzymy nowego użytkownika, pomijając przy tym właściwość określającą jego wiek. Załóżmy, że programista chciał uzupełnić tę właściwość później, na podstawie danych wpisanych przez użytkownika. I w końcu, poniżej fragmentu tworzącego użytkownika, wywołujemy funkcję `canDrive`, która stwierdza, że użytkownik nie ma prawa kierować autem. Gdyby okazało się, że użytkownik Tomek ma więcej niż 16 lat, a ta funkcja powodowałaby wykonanie innej czynności na podstawie jego wieku, to oczywiście jest, że mogłoby to doprowadzić do całej masy problemów.

Istnieją sposoby, by rozwiązać ten problem przy wykorzystaniu możliwości języka JavaScript, jeśli nie w całości, to przynajmniej częściowo. Moglibyśmy użyć pętli `for`, by przejrzeć wszystkie nazwy właściwości przekazanego obiektu użytkownika i sprawdzić, czy jest wśród nich właściwość `age`. Gdyby okazało się, że takiej właściwości nie ma, moglibyśmy zgłosić wyjątek lub w jakiś inny sposób rozwiązać problem. Niemniej jednak, gdybyśmy musieli robić to w każdej funkcji, to nasz kod bardzo szybko stałby się nieefektywny i podatny na błędy. Co więcej, wszystkie te testy byłyby wykonywane w trakcie działania kodu. Na pewno wolelibyśmy wykrywać takie błędy, zanim staną się widoczne dla użytkownika. TypeScript udostępnia proste rozwiązanie takiego problemu i potrafi wychwytywać podobne błędy jeszcze zanim kod w ogóle trafi do środowiska produkcyjnego. Przyjrzyjmy się poniższemu fragmentowi kodu:

```
interface User {
  name: string;
  age: number;
}

function canDrive(usr: User) {
  console.log("imię użytkownika: ", usr.name);

  if(usr.age >= 16) {
    console.log("może prowadzić auto");
  } else {
    console.log("nie ma prawa kierować");
  }
}

const tom = {
  name: "Tomek"
}

canDrive (tom);
```

Przenalizujmy ten zmodyfikowany przykład. Na jego samym początku znajduje się coś, co nazywamy interfejsem. W naszym przypadku interfejs ten ma nazwę `User`. Interfejsy są jednym z rodzajów typów dostępnych w języku TypeScript. Więcej szczegółowych informacji o interfejsach przedstawię w kolejnych rozdziałach, a póki co skoncentrujemy się dalej na przykładzie. Nasz interfejs `User` ma dwa pola, o nazwach `name` i `age`. Poniżej widzimy, że w funkcji `canDrive`, za nazwą parametru, `usr`, zostały dodane dwukropek i nazwa typu — `User`. Ten zapis jest nazywany adnotacją typu (ang. *type annotation*) i oznacza, że informujemy kompilator, by pozwalał przekazywać do funkcji `canDrive` jedynie parametry typu `User`. A zatem, kiedy spróbujemy skompilować ten kod, kompilator TypeScriptu poskarży się, że w parametrze przekazanym do funkcji `canDrive` brakuje właściwości `age`, co będzie zgodne z prawdą, bo w naszym obiekcie `tom` faktycznie tej właściwości nie ma (patrz rysunek 1.4).

```
H:\NaukaTypeScriptu\rozdzial01>tsc test-age.ts
test-age.ts:20:11 - error TS2345: Argument of type '{ name: string; }' is not assignable to parameter of type 'User'.
  Property 'age' is missing in type '{ name: string; }' but required in type 'User'.

20 canDrive (tom);
   ~~~~~

test-age.ts:3:5
   age: number;
   ~~~~~
'age' is declared here.

Found 1 error.

H:\NaukaTypeScriptu\rozdzial01>
```

**Rysunek 1.4.** Błąd w wywołaniu funkcji canDrive

2. Po raz kolejny kompilator wykrył błąd. Popraw go — określ typ zmiennej tom:

```
const tom: User = {
  name: "Tomek"
}
```

3. Jeśli określimy, że zmienna tom jest typu User, a jednocześnie nie dodamy do niej właściwości age, to kompilator zgłosi kolejny błąd:

```
Property 'age' is missing in type '{ name: string; }' but required in type
↳ 'User'.ts(2741)
```

Jeśli jednak dodamy właściwość age, to błąd zniknie, a funkcja canDrive zacznie działać tak, jak powinna. Poniżej przedstawiłem końcową, działającą postać kodu:

```
interface User {
  name: string;
  age: number;
}

function canDrive(usr: User) {
  console.log("imię użytkownika: ", usr.name);

  if(usr.age >= 16) {
    console.log("może prowadzić auto");
  } else {
    console.log("nie ma prawa kierować");
  }
}

// Załóżmy, że nieco później ktoś używa funkcji canDrive
const tom: User = {
  name: "Tomek",
  age: 25
}

canDrive (tom);
```

Jak widać, w tym kodzie zmienna `tom` dysponuje już właściwością `age`, więc podczas wykonywania funkcji `canDrive` będzie można prawidłowo sprawdzić wartość `usr.age` i na jej podstawie wykonać odpowiedni kod.

Poniżej, na rysunku 1.5, przedstawiłem wyniki generowane przez poprawioną, końcową wersję powyższego przykładu.

```
H:\NaukaTypeScriptu\rozdzial01>tsc test-age.ts
H:\NaukaTypeScriptu\rozdzial01>node test-age.js
imię użytkownika: Tomek
może prowadzić auto
H:\NaukaTypeScriptu\rozdzial01>
```

**Rysunek 1.5.** Poprawne wykonanie funkcji `canDrive`

W tym rozdziale poznałeś kilka problemów związanych z dynamicznym typowaniem i zobaczyłeś, w jaki sposób dynamiczne typowanie może te problemy eliminować i chronić nas przed nimi. Statyczne typowanie usuwa niejednoznaczności mogące występować w kodzie, przy czym chodzi tu o fragmenty kodu, które mogą być niejednoznaczne zarówno dla kompilatora, jak i dla programistów. Przejrzystość wynikająca ze stosowania statycznego typowania może przyczynić się do zmniejszenia liczby błędów w kodzie oraz ułatwić tworzenie kodu o wysokiej jakości.

## Programowanie obiektowe

JavaScript jest znany jako język obiektowy. Faktycznie, dysponuje on pewnymi możliwościami typowymi dla innych języków obiektowych, takimi jak dziedziczenie. Niemniej jednak implementacja obiektowości zastosowana w JavaScriptcie jest dość ograniczona i to zarówno pod względem dostępnych możliwości języka, jak i zastosowanego projektu. W tym punkcie rozdziału przedstawię możliwości programowania obiektowego dostępne w JavaScriptcie oraz opiszę, w jaki sposób TypeScript je rozszerza i poprawia.

Zacznijmy do tego, czym w ogóle jest programowanie obiektowe (w skrócie: OOP, ang. *Object Oriented Programming*). U podstaw programowania obiektowego leżą cztery zasady:

- hermetyzacja,
- abstrakcja,
- dziedziczenie,
- polimorfizm.

Przyjrzyjmy się dokładniej każdej z nich.



## Hermetyzacja

Hermetyzację (ang. *encapsulation*) najprościej można opisać jako ukrywanie informacji. W każdym programie będziemy dysponowali pewnymi danymi oraz funkcjami, które wykonują jakieś operacje na tych danych. Dokonując hermetyzacji bierzemy te dane i umieszczamy je w pewnego rodzaju kontenerach. W większości języków programowania kontenery te są nazywane klasami, których przeznaczeniem, najprościej rzecz ujmując, jest chronienie danych, tak by żaden kod spoza kontenera nie mógł ich zobaczyć ani modyfikować. Aby w jakikolwiek sposób skorzystać z takich danych, trzeba to robić przy użyciu funkcji obiektu kontenera. Taki sposób operowania na danych obiektu daje możliwość ścisłej kontroli nad tym, co się z tymi danymi dzieje, z jednego miejsca kodu, a nie z wielu miejsc rozszaniach po całej dużej aplikacji. Dzięki temu można znacząco ułatwić utrzymanie kodu.

Istnieją różne interpretacje hermetyzacji, które koncentrują się głównie na sposobach grupowania składowych wewnątrz wspólnego kontenera. Niemniej jednak, podchodząc do hermetyzacji w ścisłym znaczeniu tego terminu, czyli jako ukrywania danych, trzeba stwierdzić, że JavaScript nie dysponuje wbudowaną możliwością hermetyzacji. W większości języków programowania obiektowego, hermetyzacja wymaga możliwości jawnego ukrywania składowych przy użyciu określonych możliwości samego języka. Na przykład, w języku TypeScript można użyć słowa kluczowego `private`, aby zaznaczyć, że właściwość ma być niewidoczna dla kodu spoza danej klasy. Choć w JavaScriptcie można symulować prywatność składowych korzystając z różnego rodzaju sztuczek, to jednak nie należą one do rodzimych możliwości języka, a ich stosowanie zwiększa złożoność kodu. Z kolei TypeScript dysponuje wbudowanymi możliwościami hermetyzacji, którą zapewniają słowa kluczowe języka, takie jak `private`.

Możliwość tworzenia w klasach pól prywatnych zostanie wprowadzona w wersji ECMAScript 2020. Jednak jest to nowa możliwość języka, która w czasie przygotowywania tej książki nie była jeszcze obsługiwana we wszystkich przeglądarkach.

## Abstrakcja

Abstrakcja (ang. *abstraction*) jest powiązana z hermetyzacją. Podczas korzystania z abstrakcji ukrywamy wewnętrzną implementację sposobu zarządzania danymi, udostępniając kodowi zewnętrznemu jedynie pewien uproszczony interfejs. W głównej mierze abstrakcję stosuje się w celu zapewnienia „luźnych powiązań”. Oznacza to, że pożądanym jest, by kod odpowiedzialny za pewien zbiór danych był niezależny i odseparowany od pozostałego kodu. Dzięki temu istnieje możliwość zmiany kodu w jednej części aplikacji bez wymuszania modyfikacji jej pozostałego kodu.

W przypadku większości języków programowania obiektowego abstrakcja wymaga zastosowania mechanizmu upraszczającego dostęp do obiektów, który sprawia, że nie jest konieczne ujawnianie wewnętrznego sposobu ich działania. W większości języków mechanizmami tymi są interfejsy lub klasy abstrakcyjne. Interfejsami zajmiemy się dokładniej w następnych rozdziałach, na razie wystarczy być wiedział, że interfejsy przypominają klasy, lecz ich składowe nie mają żadnego wykonywanego kodu. Możesz je sobie wyobrazić jako

swoiste otoczki, ujawniające jedynie nazwy i typy składowych obiektów, lecz ukrywające sposób ich działania. Ta cecha jest niezwykle istotna dla zapewniania „luźnych powiązań”, o których wspominałem wcześniej, i pozwala na tworzenie kodu, który będzie można łatwiej modyfikować i utrzymywać. JavaScript nie zapewnia możliwości tworzenia ani interfejsów, ani klas abstrakcyjnych, są one natomiast dostępne w języku TypeScript.

## Dziedziczenie

Dziedziczenie (ang. *inheritance*) jest związane z wielokrotnym stosowaniem kodu. Na przykład, jeśli musimy utworzyć obiekty reprezentujące kilka typów pojazdów — aut osobowych, ciężarówek oraz łodzi — pisanie unikalnego kodu dla każdego z tych typów byłoby nieefektywne i niewygodne. Znacznie lepszym rozwiązaniem byłoby utworzenie pewnego typu bazowego, zawierającego podstawowe atrybuty, wspólne dla wszystkich typów pojazdów, a następnie ponowne używanie kodu tego typu w każdym konkretnym typie pojazdu. W ten sposób pewne niezbędne fragmenty kodu będziemy musieli napisać tylko raz, a następnie będą one współużytkowane we wszystkich typach pojazdów.

Klasy oraz mechanizm dziedziczenia są obsługiwane zarówno w języku JavaScript, jak i TypeScript. Jeśli nie spotkałeś się jeszcze z pojęciem klas, to klasą nazywamy typ danych gromadzący zbiór powiązanych ze sobą pól, który dodatkowo może także definiować funkcje operujące na tych polach. W języku JavaScript stosowany jest mechanizm dziedziczenia określany jako **dziedziczenie prototypowe** (ang. *prototypical inheritance*). Oznacza to, że w JavaScriptcie każda instancja obiektu określonego typu współużytkuje tę samą instancję pewnego obiektu podstawowego. Ten obiekt podstawowy jest właśnie prototypem — jakiegokolwiek pola lub funkcje zostaną utworzone w prototypie, będą dostępne we wszystkich obiektach danej klasy. To rozwiązanie jest dobrym sposobem oszczędzania zasobów, na przykład pamięci, niemniej jednak daleko mu do poziomu elastyczności i wyrafinowania modelu dziedziczenia dostępnego w języku TypeScript.

W TypeScriptcie klasy mogą dziedziczyć po innych klasach, a oprócz tego po interfejsach oraz klasach abstrakcyjnych. Ani interfejsy, ani klasy abstrakcyjne nie są dostępne w JavaScriptcie i właśnie dlatego stosowane w nim dziedziczenie prototypowe jest uznawane za ograniczone. Co więcej, JavaScript nie zapewnia możliwości bezpośredniego dziedziczenia po kilku różnych klasach, co jest kolejnym sposobem wielokrotnego używania kodu. W terminologii programowania obiektowego taka możliwość jest nazywana dziedziczeniem wielokrotnym. W TypeScriptcie dziedziczenie wielokrotne jest obsługiwane przy użyciu wstawek (ang. *mixins*). Wszystkimi tymi możliwościami zajmiemy się szczegółowo w dalszej części książki; na razie najważniejsze jest to, byś wiedział, że TypeScript dysponuje modelem dziedziczenia o znacznie większych możliwościach niż JavaScript — modelem, który obsługuje więcej rodzajów dziedziczenia, a przez to udostępnia więcej sposobów wielokrotnego stosowania kodu.

## Polimorfizm

Polimorfizm (ang. *polymorphism*) jest powiązany z dziedziczeniem. Pozwala on na utworzenie elementu, któremu można przypisać obiekt jednego z wielu typów, dziedziczących po wspólnej klasie bazowej. Możliwość ta jest przydatna w wielu sytuacjach, kiedy typ używanego

obiektu nie jest znany z góry, lecz może być określany w trakcie działania programu, w zależności od zaistniałych okoliczności.

Polimorfizm jest stosowany nieco rzadziej niż pozostałe mechanizmy programowania obiektowego, niemniej jednak i tak jest użyteczny. Jeśli chodzi o język JavaScript, nie dysponuje on wbudowanym wsparciem dla stosowania polimorfizmu, jednak dzięki dynamicznemu typowaniu można w nim stosunkowo łatwo polimorfizm symulować (niektórzy entuzjaści języka JavaScript będą bardzo mocno oponować przeciwko temu stwierdzeniu, jednak pozwolę sobie wyjaśnić tę opinię).

Przeanalizujmy przykład. Korzystając z mechanizmu dziedziczenia stosowanego w JavaScriptcie można utworzyć klasę bazową oraz wiele klas, które po niej dziedziczą. Następnie, używając standardowej deklaracji zmiennych stosowanej w JavaScriptcie, która nie wymaga określania typu, możemy w trakcie wykonywania programu, w zależności od potrzeb, zapisać w tej zmiennej instancję dowolnej klasy pochodnej. Problem polega na tym, że nie ma możliwości wymuszania, by zmienna była określonego typu bazowego, gdyż JavaScript nie zapewnia możliwości deklarowania typów. A to oznacza, że podczas pisania kodu nie ma jak wymusić, by zmienna mogła zawierać wyłącznie dane określonego typu bazowego. To z kolei sprawia, że musimy uciekać się do rozwiązań zastępczych i wymuszać bezpieczeństwo typów korzystając ze słowa kluczowego `instanceof` i sprawdzania typów w trakcie wykonywania kodu.

Z kolei w języku TypeScript kontrola typów jest stosowana domyślnie, a typ zmiennych musi być określany od razu, podczas ich deklarowania. Co więcej, TypeScript obsługuje interfejsy, które następnie mogą być implementowane przez klasy. Oznacza to, że deklarując zmienną jako typ interfejsu możemy zaznaczyć, że wszystkie obiekty zapisywane w tej zmiennej muszą być instancjami klas implementujących dany interfejs. Jeszcze raz zaznaczam, że wszystkie te warunki są weryfikowane podczas pisania kodu, jeszcze zanim zostanie on wdrożony. Taki system jest znacznie bardziej przejrzysty, łatwiejszy do wyegzekwowania i bardziej niezawodny niż to zapewnia JavaScript.

W tym punkcie rozdziału opisałem, czym jest programowanie obiektowe oraz jakie jest jego znaczenie dla tworzenia dużych aplikacji. Wyjaśniłem także, dlaczego mechanizmy programowania obiektowego dostępne w języku TypeScript są znacząco lepsze i bogatsze od tych, jakie daje JavaScript.

## Podsumowanie

W tym rozdziale pokrótce przedstawiłem to, czym jest język TypeScript i dlaczego został utworzony. Wyjaśniłem w nim, dlaczego bezpieczeństwo typów oraz możliwości programowania obiektowego mają tak wielkie znaczenie dla tworzenia dużych aplikacji. Następnie zamieściłem kilka przykładów, które pozwoliły mi porównać mechanizmy typowania dynamicznego i statycznego, i pokazać, dlaczego, z punktu widzenia tworzenia kodu, typowanie statyczne jest znacznie lepsze. Na koniec porównałem style programowania obiektowego wykorzystywane w JavaScriptcie i TypeScriptcie, a także wyjaśniłem, dlaczego możliwości języka TypeScript

są znacznie bogatsze i lepsze. Zamieszczone tu informacje powinny pozwolić Ci dobrze zrozumieć, na wysokim, koncepcyjnym poziomie, jakie zalety zapewnia język TypeScript.

W następnym rozdziale przyjrzymy się nieco dokładniej językowi TypeScript. Dowiesz się z niego więcej na temat typów i poznasz niektóre z najważniejszych cech tego języka, takie jak klasy, interfejsy i typy generyczne (ang. *generics*). Informacje zamieszczone w tym rozdziale zapewnią Ci solidne podstawy do korzystania z przeróżnych frameworków i bibliotek dostępnych w ekosystemie języka JavaScript.

# Prezentacja języka TypeScript

W tym rozdziale zajmiemy się dokładniejszym omówieniem języka TypeScript. Dowiesz się w nim jak wygląda jawna składnia deklarowania typów oraz poznasz wbudowane typy tego języka i ich przeznaczenie.

Znajdziesz tu także informacje o tym, jak można tworzyć własne typy oraz jak pisać aplikacje zgodne z wytycznymi programowania obiektowego. I w końcu przyjrzymy się także kilku najnowszym możliwościom dodanym do TypeScriptu w ostatnim czasie, takim jak łączenie opcjonalne (ang. *optional chaining*) oraz scalanie wartości pustych (ang. *nullish coalescing*).

Po przeczytaniu tego rozdziału będziesz dysponował dobrą znajomością TypeScriptu, która zapewni Ci możliwość bezproblemowego analizowania i rozumienia kodu pisanego w tym języku. Co więcej, Twoja znajomość języka pozwoli na pisanie w nim kodu o wysokiej jakości, który nie tylko będzie realizować cele stawiane przed aplikacją, lecz także będzie solidny i niezawodny.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- wyjaśnieniem czym są typy;
- przedstawieniem typów języka TypeScript;
- przedstawieniem klas i interfejsów;
- wyjaśnieniem czym jest dziedziczenie;

- przedstawieniem typów generycznych;
- przedstawieniem najnowszych możliwości języka i opcji konfiguracji kompilatora.

## Wymagania techniczne

Wymagania techniczne, które musisz spełnić przed przystąpieniem do lektury tego rozdziału, są takie same jak w przypadku rozdziału 1., pt. „Jak rozumieć TypeScript i poprawić swoją znajomość języka JavaScript”. Powinieneś dysponować podstawową znajomością języka JavaScript oraz technologii internetowych. Podobnie jak w poprzednim rozdziale, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial02 (Chap2)*.

Zanim przystąpisz do lektury, przygotuj środowisko robocze:

1. Przejdź do katalogu *NaukaTypeScriptu* i utwórz w nim podkatalog *rozdzial02*.
2. Uruchom program VSCode i wybierz opcje *File/Open*, a następnie otwórz utworzony przed chwilą katalog *rozdzial02*. Następnie wybierz z menu opcje *View/Terminal* — w efekcie, wewnątrz okna VSCode zostanie wyświetlony panel terminala.
3. W panelu terminala wpisz polecenie `npm init` — użyłeś go już wcześniej w poprzednim rozdziale, by zainicjować nowy projekt npm — i zaakceptuj wszystkie ustawienia domyślne.
4. Wykonaj polecenie `npm install typescript` (analogicznie jak w rozdziale 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript”), aby zainstalować język TypeScript.

Teraz już jesteś gotowy do dalszej lektury.

## Czym są typy?

**Typy** są zestawami reguł nadającymi się do wielokrotnego stosowania. Typ może zawierać właściwości oraz funkcje (określające jego możliwości). Oprócz tego może być wielokrotnie współużytkowany. Kiedy ponownie używamy typu, tworzymy jego **instancję** (ang. *instance*). Oznacza to, że tworzymy egzemplarz danego typu, którego właściwości będą mieć konkretne wartości. W języku TypeScript, czego można było się spodziewać zważywszy na jego nazwę, typy odgrywają bardzo ważną rolę. Przyjrzyjmy się zatem, jak działają typy w języku TypeScript.

## Jak działają typy?

Jak już wcześniej wspominałem, typy występują także w języku JavaScript. Liczby, łańcuchy, wartości logiczne, tablice, itd., to wszystko typy dostępne w JavaScriptcie. Niemniej jednak, typy te nie są jawnie określone w deklaracjach — środowisko wykonawcze określa je podczas wykonywania kodu. Natomiast w języku TypeScript, typy zazwyczaj określa się w deklaracjach, choć można także pozwolić kompilatorowi, by samemu je odgadywał. Trzeba jednak pamiętać, że typy odgadywane przez kompilator nie zawsze będą tymi, których chcielibyśmy używać, gdyż wskazanie odpowiedniego typu nie zawsze jest oczywiste. Oprócz typów obsługiwanych przez JavaScript, TypeScript udostępnia kilka swoich własnych, unikalnych typów, pozwala także nam tworzyć własne.

Pierwszą rzeczą, jaką musimy zapamiętać odnośnie do typów w języku TypeScript jest to, że są one obsługiwane na podstawie struktury (czy też kształtu, ang. *shape*), a nie na podstawie nazw. Oznacza to, że nazwa typu nie ma wielkiego znaczenia dla kompilatora TypeScriptu — liczą się właściwości, jakie dany typ zawiera.

Przyjrzyjmy się poniższemu przykładowi:

1. Utwórz plik o nazwie *shape.ts* i zapisz w nim następujący kod:

```
class Person {
  name: string;
}
const jill: { name: string } = {
  name: "Julka"
};
const person: Person = jill;
console.log(person);
```

Przed wszystkim powinieneś zwrócić uwagę w tym kodzie na klasę *Person*, mającą jedną właściwość o nazwie *name*. **Poniżej klasy stworzymy zmienną o nazwie *jill***, której typ określiliśmy jako `{ name: string }`. Rozwiązanie to wygląda nieco dziwnie, gdyż zastosowana deklaracja typu nie jest nazwą, bardziej przypomina definicję typu. Jednak z punktu widzenia kompilatora nie stanowi to żadnego problemu, więc nie zgłosi on żadnych uwag. W języku TypeScript nic nie stoi na przeszkodzie, by w jednym miejscu zdefiniować i jednocześnie zadeklarować typ. W dalszej części kodu stworzymy zmienną *person* typu *Person*, w której zapisujemy wartość zmiennej *jill*. Także w tym przypadku kompilator nie zgłasza żadnych problemów, więc wydaje się, że wszystko jest w porządku.

2. Teraz skompiluj ten plik, uruchom go i przekonaj się, co się stanie. W tym celu w panelu terminala VSCode wpisz następujące polecenia:

```
tsc shape
node shape
```

Wykonanie tych poleceń powinno wygenerować wyniki przedstawione na rysunku 2.1.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc shape

H:\NaukaTypeScriptu\rozdzial02>node shape
{ name: 'Julka' }

H:\NaukaTypeScriptu\rozdzial02>
```

### Rysunek 2.1. Wyniki wykonania skryptu shape

Jak widać, kod można skompilować i wykonać bez najmniejszych problemów. Ten przykład pokazuje, że kompilator języka TypeScript zwraca uwagę na strukturę typów i całkowicie pomija ich nazwy. W następnych rozdziałach, kiedy będziemy dokładniej zajmować się typami TypeScriptu, przekonasz się, dlaczego pamiętanie o tej zasadzie działania typów w tym języku ma tak duże znaczenie.

## Wprowadzenie do typów języka TypeScript

W tym podrozdziale zajmiemy się wybranymi z podstawowych typów dostępnych w języku TypeScript. Korzystanie z nich sprawi, że kompilator będzie kontrolował typy i wyświetlał komunikaty o błędach, które pozwolą nam na poprawianie pisanego kodu. Oprócz tego, stosowanie tych typów zapewni nam możliwość przekazania innym programistom w zespole informacji o naszych intencjach. Czytaj więc dalej, aby przekonać się, jak działają typy.

### Typ any

Any jest typem dynamicznym, charakteryzuje się tym, że można go zastąpić dowolnym innym typem. A zatem, jeśli zadeklarujemy zmienną typu any, będziemy mogli przypisać jej dowolną wartość, a później dowolnie ją zmieniać. W efekcie oznacza to, że zmienna typu any nie ma żadnego typu, a kompilator nie będzie jej sprawdzał. I to jest najważniejsza informacja, którą należy zapamiętać odnośnie do typu any — kompilator nie będzie go sprawdzał ani ostrzegał nas o potencjalnych problemach związanych ze zmiennymi tego typu. Dlatego, jeśli to tylko możliwe, należy unikać jego stosowania. Można uznać, że to nieco dziwne, że język zaprojektowany pod kątem statycznej kontroli typów udostępnia taką możliwość, jednak okazuje się, że w niektórych okolicznościach jest ona niezbędna.

W dużych aplikacjach może się zdarzyć, że programista nie zawsze będzie miał kontrolę nad typami danych trafiającymi do jego kodu. Na przykład, jeśli programista pobiera dane korzystając z API jakiejś usługi internetowej, to typ zwracanych danych będzie określany przez jakiś inny zespół lub nawet przez programistów innej firmy. Dokładnie to samo dotyczy rozwiązań korzystających z mechanizmów współdziałania, w których nasz kod może być uzależniony do kodu napisanego w jakimś innym języku programowania — zdarza się



tak choćby w sytuacjach, kiedy firma, podczas pisania nowego systemu w jakimś języku programowania, korzysta ze starego systemu, napisanego w innym języku.

Ważne jest, by nie nadużywać typu `any`. Powinienes zwracać baczna uwagę, by stosować go jedynie w sytuacjach, kiedy nie ma żadnego innego rozwiązania — na przykład, kiedy informacje o typie nie są jasne lub kiedy mogą się zmieniać. Istnieje jednak kilka alternatyw dla stosowania typu `any`. W zależności od okoliczności możemy na przykład skorzystać z interfejsów, typów generycznych, unii, bądź też z typu `unknown`. W następnym punkcie zajmujemy się ostatnią z tych możliwości, typem `unknown`, a pozostałe opiszę w dalszej części rozdziału.

## Typ unknown

Typ `unknown` (nieznany) został wprowadzony w wersji 3 języka TypeScript. Jest on podobny do typu `any` pod tym względem, że po zadeklarowaniu zmiennej tego typu można do niej przypisać wartość dowolnego typu. Tę wartość można następnie zmienić — i to także na wartość dowolnego innego typu. A zatem, w zmiennej moglibyśmy początkowo zapisać łańcuch znaków, a później zmienić jej wartość na liczbę. Jednak bez wcześniejszego sprawdzenia faktycznego typu takiej zmiennej nie można wywoływać żadnych jej składowych ani zapisywać jej jako wartości innej zmiennej. Jedyną sytuacją, kiedy możemy przypisać zmienną typu `unknown` innej zmiennej bez sprawdzania typu przypisywanej wartości, jest ta, gdy przypisujemy ją innej zmiennej typu `unknown` lub `any`.

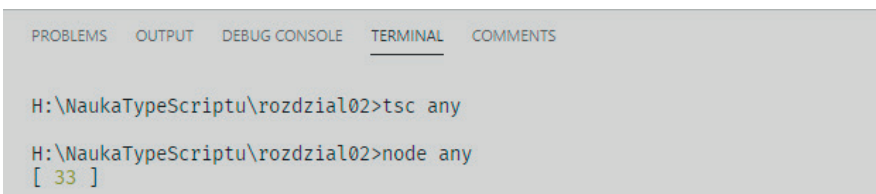
Przeanalizujemy teraz przykład zastosowania typu `any`, a później pokażę, dlaczego stosowanie typu `unknown` jest lepsze od korzystania z typu `any` (w rzeczywistości jest to nawet zalecane przez zespół twórców języka TypeScript):

1. W pierwszej kolejności przyjrzymy się przykładowi problemowi związanemu z wykorzystaniem typu `any`. W edytorze VSCode utwórz plik *any.ts* i zapisz w nim następujący kod:

```
let val: any = 22;
val = "to jest łańcuch";
val = new Array();
val.push(33);

console.log(val);
```

Jeśli teraz skompilujesz i uruchomisz ten kod, wygeneruje on wyniki przedstawione na rysunku 2.2.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc any

H:\NaukaTypeScriptu\rozdzial02>node any
[ 33 ]
```

Rysunek 2.2. Wyniki wykonania skryptu `any`

2. Ponieważ zmienna `val` jest typu `any`, możemy zapisać w niej dowolną wartość, a następnie wywołać metodę `push`, gdyż jest to metoda typu `Array`. Niemniej jednak jest to oczywiste tylko dla nas, programistów, gdyż wiemy, że typ `Array` dysponuje metodą o nazwie `push`. A co by się stało, gdybyśmy przez przypadek spróbowali wywołać jakąś metodę, której typ `Array` nie udostępnia?

Zmodyfikuj kod pliku *any.ts* zgodnie z kolejnym przykładem:

```
let val: any = 22;
val = "to jest łańcuch";
val = new Array();
val.nieistniejacametoda(33);
```

```
console.log(val);
```

3. A teraz ponownie spróbuj skompilować plik *any.ts*:

**tsc any**

Przekonasz się, że — niestety — i tym razem kompilator nie zgłosił żadnych problemów, gdyż zadeklarowanie zmiennej typu `any` sprawia, że kompilator nie będzie sprawdzał jej typu. Oprócz tego straciliśmy także możliwości zapewniane przez mechanizm IntelliSense w VSCode, a konkretnie: kolorowanie kodu oraz sprawdzanie i wyświetlanie błędów w edytorze. Dopiero po wykonaniu kodu uzyskamy jakikolwiek sygnał, że występują w nim jakieś problemy; a to niestety nie jest to, o co nam chodziło. Jeśli teraz wykonamy skompilowany kod, natychmiast pojawią się komunikaty o błędzie, podobne do tych przedstawionych na rysunku 2.3.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
H:\NaukaTypeScriptu\rozdzial02>tsc any.ts
```

```
H:\NaukaTypeScriptu\rozdzial02>node any.js
```

```
H:\NaukaTypeScriptu\rozdzial02\any.js:4
```

```
val.nieistniejacametoda(33);
```

```
TypeError: val.nieistniejacametoda is not a function
    at Object.<anonymous> (H:\NaukaTypeScriptu\rozdzial02\any.js:4:5)
    at Module._compile (node:internal/modules/cjs/loader:1101:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1153:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:79:12)
    at node:internal/main/run_main_module:17:47
```

**Rysunek 2.3.** Błąd w skrypcie *any*

W przypadku prostego skryptu, takiego jak ten przedstawiony powyżej, popełnienie podobnego błędu jest raczej mało prawdopodobne; jednak w dużych aplikacjach taki błąd można popełnić bardzo łatwo — może to być choćby prosty błąd typograficzny.

A teraz przyjrzyjmy się podobnemu przykładowi, w którym tym razem zastosujemy typ `unknown`:

1. W pierwszej kolejności umieść w komentarzach cały dotychczasowy kod zapisany w pliku *any.ts* i usuń plik *any.js* (będziemy używać zmiennych o tych samych nazwach, więc gdybyś tego nie zrobił, kompilator mógłby zgłosić błąd związany z występowaniem konfliktów).

W dalszej części książki poznasz tak zwane przestrzenie nazw. Pozwalają one unikać takich konfliktów, jednak jak na razie jest jeszcze trochę za wcześnie, byś ich używał.

2. Następnie utwórz nowy plik o nazwie *unknown.ts* i zapisz w nim następujący fragment kodu:

```
let val: unknown = 22;
val = "to jest łańcuch";
val = new Array();
val.push(33);

console.log(val);
```

Po jego wpisaniu zauważysz zapewne, że VSCode od razu wyświetli błąd związany z wywołaniem funkcji `push`. To nieco dziwne, gdyż typ `Array` udostępnia przecież tę metodę. Jednak wyświetlenie tego błędu pokazuje, że typ `unknown` działa prawidłowo. Możesz sobie wyobrazić, że `unknown` to coś, co bardziej przypomina etykietę niż typ danych, oraz że pod tą etykietą jest ukryty faktyczny typ. Jednak kompilator nie jest w stanie samodzielnie określić tego typu, dlatego też sami musimy zadbać o to, by udowodnić kompilatorowi, że zmienna jest konkretnego typu.

3. Używamy strażników typów by upewnić się, że zmienna `val` jest konkretnego typu:

```
let val: unknown = 22;
val = "to jest łańcuch";
val = new Array();
if (val instanceof Array) {
    val.push(33);
}

console.log(val);
```

Jak widać, w tej wersji kodu wywołanie metody `push` umieściliśmy wewnątrz instrukcji warunkowej, sprawdzającej, czy `val` jest instancją typu `Array`.

4. Kiedy już upewnimy się, że warunek będzie spełniony, wywołanie metody `push` zostanie wykonane bez wyświetlania żadnych komunikatów o błędach, jak pokazałem na rysunku 2.4.

Ten mechanizm jest dość niewygodny, gdyż zmusza nas do sprawdzania typu za każdym razem, gdy chcemy odwołać się do składowej obiektu. Jednak pomimo to jest on preferowany względem stosowania typu `any` i znacznie od niego bezpieczniejszy, gdyż pozwala na sprawdzanie kodu przez kompilator.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc unknown
H:\NaukaTypeScriptu\rozdzial02>node unknown
[ 33 ]

```

Rysunek 2.4. Zastosowanie typu unknown

## Typy przecięć i unii

Zapewne pamiętasz, że na początku tego podrozdziału zaznaczyłem, że kompilator TypeScriptu koncentruje się na strukturze typu, a nie na jego nazwie? Ten sposób działania sprawia, że język TypeScript udostępnia specyficzny rodzaj typów nazywanych **przecięciami** (ang. *intersection types*), określanych także czasami jako *intersekcje*. Oznacza to, że TypeScript pozwala programistom na „tworzenie typów” poprzez scalanie kilku odrębnych typów w jedną, nową całość. Ponieważ dość trudno to sobie wyobrazić, więc najlepiej będzie, jak przedstawię odpowiedni przykład. Jeśli spojrzysz na poniższy fragment kodu, zobaczysz w nim zmienną o nazwie `obj`, z którą są skojarzone dwa typy. Zapewne pamiętasz, że w TypeScriptie możemy nie tylko użyć nazwanego typu w deklaracji zmiennej, lecz także jednocześnie zdefiniować i zadeklarować zmienną określonego typu. W poniższym przykładzie, każdy z użytych typów jest odrębnym typem, jednak zastosowanie operatora `&` sprawia, że zostaną połączone w jeden, nowy typ:

```

let obj: { name: string } & { age: number } = {
  name: 'Tomek',
  age: 25
}

```

Spróbujmy teraz wykonać ten kod i wyświetlić wyniki w panelu terminala. Utwórz nowy plik o nazwie *intrsection.ts* i zapisz w nim poniższy fragment kodu:

```

let obj: { name: string } & { age: number } = {
  name: 'Tomek',
  age: 25
}

console.log(obj);

```

Kiedy skompilujesz i uruchomisz ten przykład, przekonasz się, że zostanie wyświetlony obiekt zawierający obie właściwości — `name` oraz `age` (jak pokazałem na rysunku 2.5).

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc intersection
H:\NaukaTypeScriptu\rozdzial02>node intersection
{ name: 'Tomek', age: 25 }

```

Rysunek 2.5. Wyniki zastosowania przecięcia typów

Jak widać, zarówno mechanizm IntelliSense, jak i kompilator prawidłowo obsługują powyższy kod, a wynikowy obiekt dysponuje obiema właściwościami. Tak właśnie działa przecięcie typów.

Kolejnym, dość podobnym rodzajem typu są tak zwane **unie** (ang. *union type*). W tym przypadku, zamiast scalać kilka typów w jeden, wybierany jest jeden z kilku dostępnych. Zobaczmy na przykładzie, jak to działa. Utwórz nowy plik o nazwie *union.ts* i zapisz w nim poniższy fragment kodu:

```
let unionObj: null | { name: string } = null;
unionObj = { name: 'Janek' };

console.log(unionObj);
```

Zastosowanie znaku `|` sprawiło, że zmienna `unionObj` została zadeklarowana jako typ `null` lub `{ name: string }`. Jeśli teraz skompilujesz i uruchomisz ten przykład, przekonasz się, że faktycznie w zmiennej można zapisywać wartości obu tych typów. Oznacza to, że w zmiennej będzie można zapisywać bądź to wartość `null`, bądź też obiekty typu `{ name: string }`.

## Typy literalowe

**Typy literalowe** (ang. *literal types*) są podobne do unii, jednak korzystają ze zbioru z góry określonych łańcuchów lub liczb. Poniżej przedstawiłem przykład zastosowania tego rodzaju typu korzystającego z łańcuchów; jest on na tyle prosty, że nie wymaga dodatkowych wyjaśnień. Jak widać, typ składa się z grupy określonych łańcuchów. Oznacza to, że w zmiennej takiego typu będzie można zapisać wyłącznie jeden z łańcuchów wymienionych w typie.

```
let literal: "Tomek" | "Linda" | "Jarek" | "Sylwia" = "Linda";
literal = "Sylwia";

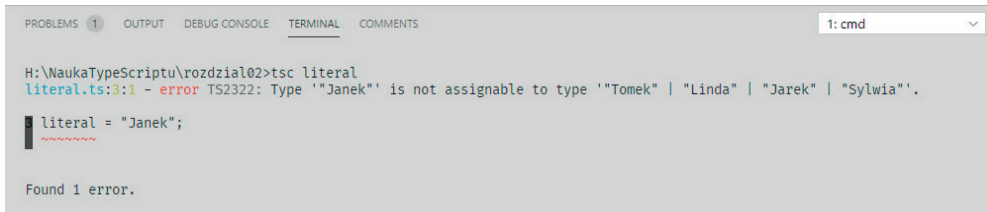
console.log(literal);
```

Jak widać, kompilator akceptuje dowolną z wymienionych wartości, jak również pozwala ją zmieniać na inną z dopuszczalnych. Jeśli jednak spróbujesz przypisać zmiennej wartość, która nie została wymieniona w typie, kompilator zgłosi błąd. Przyjrzyjmy się, jak to działa — zaktualizuj kod przykładu, przypisując zmiennej wartość `"Janek"`:

```
let literal: "Tomek" | "Linda" | "Jarek" | "Sylwia" = "Linda";
literal = "Sylwia";
literal = "Janek";
console.log(literal);
```

Tym razem próbujemy przypisać zmiennej typu literalowego wartość `"Janek"`, co sprawi, że podczas próby kompilacji kodu zostanie zgłoszony błąd (patrz rysunek 2.6).

Dostępne są także typy literalowe, których dopuszczalnymi wartościami są nie łańcuchy, lecz liczby; pomijając tę różnicę, działają one tak samo.



Rysunek 2.6. Błąd przypisania wartości do zmiennej typu literalowego

## Nazwy zastępcze typów

Nazwy zastępcze typów (określane także jako *aliasy*), są bardzo często stosowane w języku TypeScript. Pozwalają one nadawać typom inne nazwy i w większości przypadków są stosowane w celu upraszczania długich i złożonych nazw typów. Poniższy przykład pokazuje sposób definiowania oraz użycia nazwy zastępczej typu:

```
type Points = 20 | 30 | 40 | 50;
let score: Points = 20;

console.log(score);
```

W tym przykładzie tworzymy długi typ literalowy określający kilka dopuszczalnych wartości liczbowych i przypisujemy mu krótszą nazwę `Points`. Następnie deklarujemy zmienną `score` typu `Points` i przypisujemy jej wartość 20, czyli jedną z dopuszczalnych wartości typu `Points`. Oczywiście, jeśli spróbujemy przypisać zmiennej jakąkolwiek inną wartość, kompilator zgłosi błąd.

Kolejny przykład przedstawia określanie nazwy zastępczej dla typu literalu obiektowego:

```
type ComplexPerson = {
  name: string,
  age: number,
  birthday: Date,
  married: boolean,
  address: string
}
```

Deklaracja tego typu jest bardzo długa, a sam typ nie określa nazwy, czym, między innymi, różni się od klasy, która by ją miała, dlatego też przypisujemy mu nazwę zastępczą. Nazwy zastępcze można określać dla dowolnych typów języka TypeScript, w tym także dla funkcji i typów generycznych, którym przyjrzymy się w dalszej części tego rozdziału.

## Typy wyników funkcji

W celu uzupełnienia zagadnień związanych z typami chciałbym także przedstawić przykład deklarowania typu wartości zwracanej przez funkcję. Wygląda ona bardzo podobnie do typowej deklaracji zmiennej. Zacznij od utworzenia nowego pliku o nazwie *functionReturn.ts*, a następnie zapisz w nim poniższą funkcję:

```
function runMore(distance: number): number {
    return distance + 10;
}
```

Funkcja `runMore` pobiera jeden parametr typu `number` i zwraca wartość także typu `number`. Deklaracja parametru wygląda dokładnie tak samo, jak deklaracja każdej innej zmiennej; natomiast typ wartości zwracanej przez funkcję jest zapisywany za nawiasem zamykającym listę parametrów. Jeśli funkcja niczego nie zwraca, to możemy bądź to całkowicie pominąć deklarację typu wyniku, bądź też zadeklarować go jako `void`.

Przyjrzymy się teraz przykładowi funkcji zwracającej wynik typu `void`. Umieść zdefiniowaną wcześniej funkcję `runMore` w komentarzu, dodaj do pliku zamieszczony poniżej kod, a następnie skompiluj go i wykonaj:

```
function eat(calories: number) {
    console.log("Zjadłem " + calories + " kalorii.");
}
function sleepIn(hours: number): void {
    console.log("Spałem " + hours + " godzin.");
}

let ate = eat(100);
console.log(ate);
let slept = sleepIn(10);
console.log(slept);
```

Żadna z tych nowych funkcji nie zwraca wyniku, a ich działanie sprowadza się do wyświetlenia na konsoli wartości przekazanego parametru (patrz rysunek 2.7).

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc functionReturn

H:\NaukaTypeScriptu\rozdzial02>node functionReturn
Zjadłem 100 kalorii.
undefined
Spałem 10 godzin.
undefined

H:\NaukaTypeScriptu\rozdzial02>
```

**Rysunek 2.7.** Funkcje, które niczego nie zwracają

Jak pokazują wyniki wykonania skryptu, wywołania `console.log` umieszczone wewnątrz funkcji są wykonywane; jednak próba pobrania i użycia wartości wynikowej zwracanej przez każdą z tych funkcji kończy się wyświetleniem `undefined`, gdyż żadna z funkcji niczego nie zwraca.

A zatem, deklaracje typu wyniku funkcji są bardzo podobne do deklaracji zmiennych. A teraz przyjrzymy się bliżej kolejnemu zagadnieniu, a mianowicie: stosowaniu funkcji jako typów.

## Funkcje jako typy

Może się to wydawać nieco dziwne, jednak w języku TypeScript typem może być także cała sygnatura funkcji. W poprzednim punkcie pokazałem, w jaki sposób funkcja może pobierać parametry określonych typów oraz zwracać wynik określonego typu. Ta definicja jest właśnie określaną sygnaturą funkcji. W języku TypeScript, taka sygnatura może być także używana jako typ właściwości obiektów.

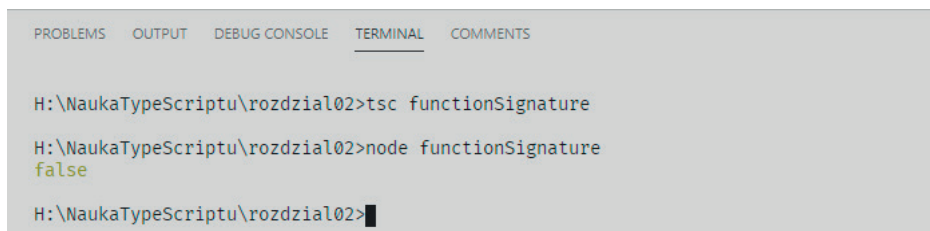
Przeanalizujmy przykład takiego rozwiązania. Utwórz plik o nazwie *functionSignature.ts*, a następnie zapisz w nim poniższy kod:

```
type Run = (miles: number) => boolean;
let runner: Run = function (miles: number): boolean {
    if(miles > 10){
        return true;
    }
    return false;
}

console.log(runner(9));
```

W pierwszym wierszu tego przykładu znajduje się typ funkcyjny, którego użyjemy w dalszej części kodu. Nazwę zastępczą *Run* tego typu zdefiniowałem tylko po to, by uprościć stosowanie długiej sygnatury funkcji. Sam typ funkcyjny ma następującą postać: *(miles: number) => boolean*. Wygląda to dość dziwnie, jednak w rzeczywistości ten typ jest jedynie nieco skróconą sygnaturą funkcji. W tej deklaracji typu należy zwrócić uwagę na nawiasy, w których jest zapisana lista parametrów, symbol *=>* informujący, że mamy do czynienia z funkcją, oraz umieszczony za tym symbolem typ wyniku.

W kolejnym wierszu znajduje się deklaracja zmiennej *runner* typu *Run*, która oczywiście będzie funkcją. Funkcja, którą przypisujemy tej zmiennej sprawdza, czy biegacz przebiegł więcej niż 10 mil i zwraca *true*, jeśli ten warunek został spełniony, lub *false* w przeciwnym przypadku. Na samym końcu kodu znajduje się wywołanie metody *console.log*, które wyświetla wynik wywołania funkcji. Zamieszczony poniżej rysunek 2.8 przedstawia wyniki kompilacji i wykonania tego przykładu:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc functionSignature
H:\NaukaTypeScriptu\rozdzial02>node functionSignature
false
H:\NaukaTypeScriptu\rozdzial02>
```

**Rysunek 2.8.** Zastosowanie typu funkcyjnego

Jak widać, wywołanie funkcji *runner* z argumentem 9 sprawi, że funkcja zwróci wartość *false*, co jest zgodne z oczekiwaniami. W przypadku języków korzystających z silnego typowania, ważne jest, by dla wszystkich sposobów zwracania danych w kodzie określać typy tych danych, a takimi sposobami zwracania danych mogą być nie tylko zmienne, lecz także funkcje.



## Typ never

Na pierwszy rzut oka ten typ będzie się wydawał dość dziwny. Typ `never` jest używany do oznaczania, że funkcja nigdy nie zwróci wyniku (nie uda się jej zakończyć), bądź też, że zmienna nigdy nie zostanie ustawiona (nie zostanie jej przypisana żadna wartość, nawet `null`). Z pozoru `never` bardzo przypomina typ `void`. Jednak w żadnym wypadku nie są one identyczne. W przypadku typu `void`, działanie funkcji się kończy (i to w dosłownym znaczeniu tego słowa), jednak funkcja nie zwraca żadnego wyniku (zwraca `undefined`, co nie jest żadną wartością). Może się zatem wydawać, że `never` jest całkowicie bezużytecznym typem, okazuje się jednak, że jest on bardzo przydatny do określania naszych intencji.

Przeanalizujmy poniższy przykład. Utwórz plik *never.ts* i zapisz w nim poniższy kod:

```
function oldEnough(age: number): never | boolean {
  if (age > 59) {
    throw Error("Jesteś za stary!");
  }
  if (age <= 18) {
    return false;
  }
  return true;
}
```

Jak widać, typem wyniku zwracanego przez funkcję zdefiniowaną w tym przykładzie jest unia — `never` lub `boolean`. Okazuje się, że mogliśmy zadeklarować typ wyniku jako `boolean` i powyższy kod i tak by działał prawidłowo. Niemniej jednak, jeśli wiek przekazany do funkcji będzie przekraczał pewną wartość, funkcja zgłosi błąd, informując w ten sposób, że przekazana wartość jest nieoczekiwana. A zatem, ponieważ hermetyzacja jest jedną z podstawowych zasad pisania kodu o wysokiej jakości, korzystne może być wyraźne oznaczenie, że w pewnych okolicznościach funkcja może zawieść i nie zwrócić wartości, bez zmuszania innych do zagłębiania się w szczegóły jej działania. Właśnie tę informację przekazuje użycie typu `never`.

W tym podrozdziale poznałeś wiele wbudowanych typów języka TypeScript. Przekonałeś się także, dlaczego ich stosowanie pozwala poprawić jakość kodu i ułatwia wczesne wykrywanie błędów — już na etapie pisania kodu. W następnym podrozdziale dowiesz się, w jaki sposób możemy używać TypeScriptu do tworzenia własnych typów oraz pisania kodu, który będzie zgodny z zasadami programowania obiektowego.

## Klasy i interfejsy

O klasach i interfejsach wspominałem już pobieżnie we wcześniejszych częściach rozdziału. W tym podrozdziale przyjrzymy się im dokładniej i przekonamy, dlaczego pozwalają nam one na tworzenie lepszego kodu. Pod koniec tej części rozdziału będziesz znacznie lepiej przygotowany do pisania kodu bardziej czytelnego, nadającego się do wielokrotnego stosowania i mniej podatnego na występowanie błędów.

## Klasy

Na podstawowym poziomie, klasy w języku TypeScript wyglądają tak samo jak klasy w JavaScript. Są to pojemniki grupujące powiązane ze sobą pola i metody, których można używać do tworzenia obiektów i wielokrotnie używać. Jednak klasy języka TypeScript zapewniają większe możliwości hermetyzacji, które nie są dostępne w języku JavaScript. Przyjrzyjmy się im na przykładzie.

Utwórz nowy plik o nazwie *classes.ts* i zapisz w nim następujący kod:

```
class Person {
  constructor() {}
  msg: string;
  speak() {
    console.log(this.msg);
  }
}

const tom = new Person();
tom.msg = "cześć";
tom.speak();
```

Ten przykład przedstawia prostą klasę, która — poza statyczną kontrolą typów — niczym nie różni się od klas pisanych w języku JavaScript. W pierwszej kolejności podawana jest nazwa klasy, dzięki której będzie można jej wielokrotnie używać. Następnie deklarujemy konstruktor klasy, służący do inicjalizacji wszelkich pól, którymi klasa dysponuje, oraz wykonywania wszelkich innych czynności niezbędnych do przygotowania instancji danej klasy (przypomnę tylko, że instancja to konkretny egzemplarz klasy, dysponujący unikalnymi wartościami pól). W dalszej części kodu klasy deklarujemy zmienną o nazwie *msg* oraz funkcję o nazwie *speak*, która wyświetla na konsoli wartość zmiennej *msg*. W pozostałej części kodu tworzymy nową instancję naszej klasy, potem zapisujemy w jej polu *msg* łańcuch "cześć" i wywołujemy metodę *speak*. A teraz przyjrzymy się różnicom pomiędzy klasami w językach TypeScript i JavaScript.

## Modyfikatory dostępu

Podkreślałem już wcześniej, że jedną z głównych zasad programowania obiektowego jest hermetyzacja, czyli ukrywanie informacji. Jeśli jeszcze raz przyjrzymy się klasie przedstawionej na ostatnim przykładzie, wyraźnie zauważymy, że nie ukrywamy zmiennej *msg*, a jej wartość jest dostępna dla kodu spoza klasy, który nawet może ją zmieniać. Przekonajmy się zatem, co TypeScript pozwala nam zrobić z tym problemem. Zmodyfikuj kod w pliku *classes.ts* do postaci przedstawionej poniżej:

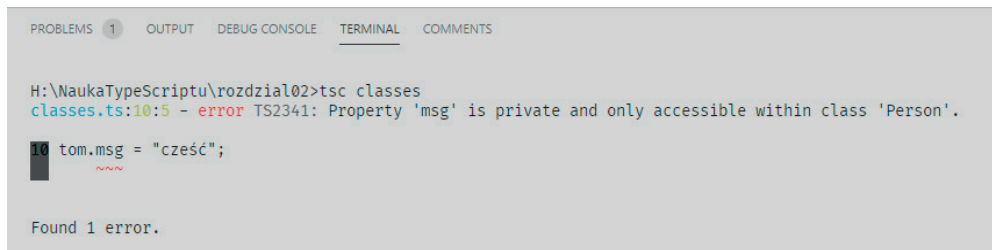
```
class Person {
  constructor(private msg: string) {}

  speak() {
    console.log(this.msg);
  }
}
```

```
const tom = new Person("cześć");
// tom.msg = "cześć";
tom.speak();
```

Jak widać, zmodyfikowaliśmy konstruktor, dodając do niego parametr poprzedzony słowem kluczowym `private`. Ten sposób deklarowania parametrów konstruktora, w którym są do nich dodawane modyfikatory dostępu, pozwala nam zrobić kilka rzeczy w jednym wierszu kodu. Przede wszystkim zapis ten informuje kompilator, że klasa dysponuje polem o nazwie `msg` typu `string`, które ma być polem prywatnym (co określa modyfikator `private`). Zazwyczaj deklaracje tego typu zapisuje się w wierszu poniżej lub powyżej konstruktora, co także jest prawidłowym rozwiązaniem, jednak TypeScript udostępnia zapis skrócony, polegający właśnie na dodaniu parametru do konstruktora. Co więcej, dodanie tego parametru zapewnia nam możliwość ustawienia wartości pola `msg` w czasie tworzenia instancji klasy — wystarczy użyć wywołania o postaci `new Person("cześć")`.

A właściwie co daje poprzedzenie pola klasy modyfikatorem `private`? Otóż użycie `private` sprawia, że pole nie będzie dostępne dla kodu spoza tej klasy. W efekcie, używana wcześniej instrukcja `tom.msg = "cześć"` przestanie działać, a próba jej skompilowania spowoduje zgłoszenie błędu. Możesz się o tym przekonać, usuwając znaki komentarza na początku tego wiersza kodu. Kiedy to zrobisz i spróbujesz skompilować plik, zostanie wyświetlony komunikat o błędzie przedstawiony na rysunku 2.9.



**Rysunek 2.9.** Błąd odwołania do prywatnego pola klasy

Jak widać, komunikat informuje, że do składowej prywatnej `msg` nie można odwoływać się spoza kodu klasy. W powyższym przykładzie użyliśmy modyfikatora, by zmienić widoczność pola, jednak modyfikatorów dostępu można także używać do określania widoczności dowolnych składowych klas, czyli nie tylko pól, lecz także funkcji.

Jak już wspominałem wcześniej, w języku ECMAScript 2020 będzie można tworzyć pola prywatne; będzie do tego służył symbol `#`. Jednak w ten sposób będzie można tworzyć jedynie pola, a nie metody prywatne; poza tym jest to tak nowy standard, że jego obsługa w przeglądarkach jest aktualnie jeszcze dość ograniczona.

Przyjrzyjmy się teraz kolejnemu modyfikatorowi dostępu: `readonly`. Jego działanie jest bardzo proste: sprawia, że pole, po początkowym ustawieniu wartości w konstruktorze, będzie przeznaczone tylko do odczytu. Wprowadź do naszego przykładu kolejną modyfikację, dodając do deklaracji pola `msg` modyfikator `readonly`:

```

class Person {
  constructor(private readonly msg: string) {}

  speak() {
    this.msg = "mówię: " + this.msg;
    console.log(this.msg);
  }
}

const tom = new Person("cześć");
// tom.msg = "cześć";
tom.speak();

```

Kiedy wpiszesz tę nową wersję kodu, mechanizm IntelliSense wyświetli błąd w kodzie funkcji `speak`, gdyż próbujemy w niej zmienić wartość pola `msg`, którego wartość została już raz ustawiona — w konstruktorze.

Modyfikatory `private` i `readonly` nie są jedynymi dostępnymi w TypeScriptie. Istnieje także kilka innych modyfikatorów dostępu, jednak lepiej będzie przedstawić je nieco później, w kontekście mechanizmów dziedziczenia.

## Akcesory `get` i `set`

Kolejną możliwością klas, dostępną zarówno w języku TypeScript, jak i JavaScript, są akcesory `get` i `set`:

- **akcesor `get`:** to właściwość pozwalająca na zmodyfikowanie lub sprawdzenie poprawności powiązanego z nią pola przed zwróceniem jego wartości.
- **akcesor `set`:** to właściwość pozwalająca na zmodyfikowanie lub wyliczenie wartości przed jej zapisaniem w powiązonym polu.

W niektórych innych językach programowania takie właściwości są nazywane właściwościami złożonymi (ang. *compound properties*). Przyjrzyjmy się im na przykładzie. Utwórz nowy plik o nazwie `getSet.ts` i zapisz w nim następujący kod:

```

class Speaker {
  private message: string;
  constructor(private name: string) {}

  get Message() {
    if(!this.message.includes(this.name)){
      throw Error("W komunikacie brakuje imienia mówcy.");
    }
    return this.message;
  }

  set Message(val: string) {
    let tmpMessage = val;
    if(!val.includes(this.name)){
      tmpMessage = this.name + " " + val;
    }
    this.message = tmpMessage;
  }
}

```

```

    }

    const speaker = new Speaker("Janek");
    speaker.Message = "cześć";
    console.log(speaker.Message);

```

W tym przykładzie dzieje się dosyć dużo, więc zanim go skompilujesz i uruchomisz, warto dokładniej go przeanalizować. W pierwszej kolejności zwróć uwagę na to, że pole `message` nie jest dostępne w konstruktorze, lecz zostało zadeklarowane jako pole prywatne (`private`), więc w kodzie, który nie należy do klasy, nie będzie można odwoływać się do niego bezpośrednio. Jedyną wartością inicjalizującą pobieraną przez konstruktor jest pole `name`. Poniżej konstruktora umieszczona jest właściwość `Message`; przed jej nazwą widoczne jest słowo kluczowe `get`, które sygnalizuje, że jest to akcesor *get*. W kodzie tej właściwości sprawdzamy, czy łańcuch zapisany we właściwości `message` zawiera nazwę mówcy, a jeśli jej nie zawiera, to zgłaszamy wyjątek, sygnalizując w ten sposób niepożądaną sytuację. Akcesor *set*, który także nosi nazwę `Message`, jest poprzedzony słowem kluczowym `set` i pobiera wartość — łańcuch, do którego, w razie konieczności, dodajemy nazwę mówcy i zapisujemy w polu `message`. Zwróć uwagę na to, że choć oba akcesory wyglądają jak funkcje, to jednak nimi nie są. Kiedy używamy ich w dalszej części kodu, robimy to w sposób charakterystyczny dla odwołań do pól, a nie wywołań metod, czyli bez nawiasów. W dolnej części kodu tworzymy obiekt `Speaker`, przekazując do konstruktora imię "Janek". Następnie przypisujemy właściwości `Message` łańcuch "cześć". W ostatniej instrukcji wyświetlamy komunikat na konsoli.

A teraz chcielibyśmy skompilować ten kod i go wykonać. W tym celu musimy postąpić nieco inaczej niż robiliśmy do tej pory. Kompilator języka TypeScript udostępnia opcje, których można używać w celu modyfikowania jego działania. W naszym przykładzie, zastosowane w nim akcesory oraz funkcja `includes` są możliwościami dostępnymi odpowiednio w wersjach ES5 oraz ES6 języka JavaScript. Jeśli jeszcze nie spotkałeś się z funkcją `includes`, to zapamiętaj, że sprawdza ona, czy jeden łańcuch znaków zawiera w sobie inny łańcuch. Poinformujmy zatem kompilator TypeScriptu, że wynikowy kod JavaScript musi być zgodny z wersjami języka nowszymi od ES3, a właśnie ta wersja kodu JavaScript jest generowana domyślnie.

Poniżej przedstawiłem nową postać polecenia uruchamiającego kompilator TypeScriptu, której będziesz musiał użyć (szczegółowe informacje dotyczące stosowania kompilatora, w tym także korzystanie z plików konfiguracyjnych, znajdziesz w dalszej części książki):

```
tsc --target "ES6" getSet
```

Po jego wykonaniu będziesz już mógł w standardowy sposób uruchomić skrypt:

```
node getSet
```

Wyniki wykonania tego skryptu przedstawiłem na rysunku 2.10.

```

H:\NaukaTypeScriptu\rozdzial02>node getSet
Janek cześć

```

Rysunek 2.10. Wyniki wykonania skryptu `getSet`

Aby dokładniej poznać działanie akcesorów, spróbujmy zmienić wiersz `speaker.Message = "cześć"` na `speaker.message = "cześć"`. Jeśli teraz spróbujemy skompilować kod przykładu, to kompilator zgłosi błąd przedstawiony na rysunku 2.11.

```

H:\NaukaTypeScriptu\rozdzial02>tsc --target "ES6" getSet
getSet.ts:22:9 - error TS2341: Property 'message' is private and only accessible within class 'Speaker'.

22 speaker.message = "cześć";
    ~~~~~
Found 1 error.

```

**Rysunek 2.11.** Błąd dostępu do pola `message`

Czy potrafisz wyjaśnić, dlaczego nie udało się skompilować tego kodu? Owszem, wynika to z faktu, że pole `message` jest prywatne i nie można odwoływać się do niego spoza kodu klasy `Speaker`.

Być może zastanawiasz się, dlaczego wspominam tu o akcesorach `get` i `set`, skoro są one dostępne także w języku JavaScript. Jeśli przyjrzyj się przedstawionemu przykładowi, zapewne zauważysz, że pole `message` zostało zadeklarowane jako prywatne (`private`), natomiast właściwości akcesorów `get` i `set` są publiczne (zwróć uwagę, że brak jawnie podanego modyfikatora dostępu oznacza, że składowa jest publiczna — `public`). A zatem, w celu zapewnienia odpowiedniej hermetyzacji, dobra praktyka nakazuje ukrycie pola i udostępnianie go wyłącznie w razie konieczności, właśnie przy wykorzystaniu akcesorów `get` oraz `set`, bądź jakiejś funkcji, która umożliwi modyfikowanie jego wartości. Pamiętaj także, że podczas określania poziomu dostępu do składowych klas, należy zaczynać od najbardziej restrykcyjnych ustawień, a dopiero potem, w razie konieczności, minimalizować ograniczenia. Oprócz tego, dzięki odwoływaniu się do pól przy użyciu akcesorów zyskujemy możliwość wykonywania wszelkiego rodzaju testów i modyfikacji, tak jak robiliśmy to w przedstawionym przykładzie, i dysponujemy pełną kontrolą nad tym, jakie dane będą zapisywane w naszej klasie i przez nią zwracane.

## Właściwości i metody statyczne

W tym podpunkcie zajmiemy się właściwościami i metodami **statycznymi**. Jeśli jakąś składową klasy zadeklarujemy jako statyczną (używając w tym celu modyfikatora `static`), zaznaczamy przez to, że jest to składowa klasy, a nie jej poszczególnych instancji. Oznacza to, że do takiej składowej można odwoływać się bez konieczności tworzenia instancji danej klasy, poprzedzając jej nazwę nazwą samej klasy.

Przyjrzyjmy się składowym statycznym na przykładzie. Utwórz nowy plik o nazwie `staticMember.ts` i zapisz w nim następujący kod:

```

class ClassA {
    static typeName: string;
    constructor() {}
    static getFullName() {

```

```

        return "ClassA " + ClassA.typeName;
    }
}

const a = new ClassA();
console.log(a.typeName);

```

Próba skompilowania takiego kodu zakończy się niepowodzeniem i wyświetleniem komunikatu z informacją, że właściwość `typeName` nie istnieje i pytaniem, czy chodziło nam o odwołanie do składowej klasowej `ClassA.typeName`. Przypominam, że w odwołaniach do składowych klasowych należy używać nazwy klas. Poniżej przedstawiłem poprawioną wersję kodu:

```

class ClassA {
    static typeName: string;
    constructor() {}
    static getFullName() {
        return "ClassA " + ClassA.typeName;
    }
}

const a = new ClassA();
console.log(ClassA.typeName);

```

Jak widać na powyższym przykładzie, do składowej statycznej należy odwoływać się używając nazwy klasy. Powstaje zatem pytanie, dlaczego moglibyśmy chcieć korzystać ze składowych statycznych, a nie instancyjnych? Otóż w pewnych okolicznościach może być przydatne współdzielenie pewnych danych pomiędzy wszystkimi instancjami klasy. Przykład takiego rozwiązania przedstawiłem na poniższym listingu:

```

class Runner {
    static lastRunTypeName: string;

    constructor(private typeName: string) {}

    run() {
        Runner.lastRunTypeName = this.typeName;
    }
}

const a = new Runner("a");
const b = new Runner("b");
b.run();
a.run();
console.log(Runner.lastRunTypeName);

```

W przypadku tego przykładu zależy nam na określeniu ostatniej instancji klasy, która w dowolnym momencie jako ostatnia wywołała funkcję `run`. Dzięki zastosowaniu składowej statycznej, przechowywanie takiej informacji może być trywialnie proste. Kolejnym zagadnieniem związanym ze składowymi statycznymi, o którym trzeba pamiętać, jest to, że wewnątrz kodu klasy mogą się do nich odwoływać zarówno składowe statyczne, jak i instancyjne. Z drugiej strony, składowe statyczne nie mogą odwoływać się do składowych instancyjnych.

W tym punkcie rozdziału poznałeś klasy oraz ich możliwości. Ta wiedza pozwoli Ci projektować kod korzystający z zasady hermetyzacji, co z kolei przyczyni się do poprawy jego jakości. W następnym punkcie rozdziału zajmiemy się interfejsami oraz programowaniem w oparciu o kontrakty.

## Interfejsy

Kolejną ważną regułą projektowania oprogramowania obiektowego jest abstrakcja. Jej celem jest redukcja złożoności oraz powiązań pomiędzy fragmentami kodu poprzez ukrywanie ich wewnętrznej implementacji (o abstrakcji wspominałem już w rozdziale 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript”). Jednym ze sposobów wprowadzania abstrakcji jest korzystanie z **interfejsów** (ang. *interfaces*) w celu ujawniania jedynie sygnatury typu, a nie jego wewnętrznego sposobu działania. Interfejsy są także czasami nazywane kontraktami, gdyż określanie konkretnych typów parametrów i wyników zwracanych przez funkcje wymusza pewne oczekiwania — i to zarówno na użytkownikach, jak i na twórcy interfejsu. A zatem, innym sposobem pojmowania, czym są interfejsy, jest wyobrażenie ich sobie jako ścisłych reguł określających postać danych przekazywanych do instancji danego typu oraz informacji, które można z takiej instancji pobierać.

A zatem, interfejsy są jedynie zestawami reguł. Aby przekształcić je w działający kod, musimy te reguły zaimplementować w formie kodu, który coś robi. Abyśmy mogli rozpocząć poznawanie interfejsów, przedstawię przykład prostego interfejsu wraz z jego implementacją. Utwórz nowy plik o nazwie *interfaces.ts* i zapisz w nim następujący kod:

```
interface Employee {
  name: string;
  id: number;
  isManager: boolean;
  getUniqueId: () => string;
}
```

Ten interfejs definiuje typ `Employee`; z następnego przykładu dowiesz się, jak utworzyć instancję tego typu. Jak widać, interfejs zawiera jedynie sygnaturę funkcji `getUniqueId`, a nie jej implementację — tę utworzymy podczas definiowania instancji tego typu.

Teraz dodaj do pliku *interfaces.ts* implementację interfejsu `Employee`. Poniższy kod tworzy dwie instancje tego interfejsu:

```
const linda: Employee = {
  name: "Linda",
  id: 2,
  isManager: false,
  getUniqueId: (): string => {
    let uniqueId = linda.id + "-" + linda.name;
    if(!linda.isManager) {
      return "prc-" + uniqueId;
    }
    return uniqueId;
  }
}
```



```

console.log(linda.getUniqueId());
const pam: Employee = {
  name: "Patrycja",
  id: 1,
  isManager: true,
  getUniqueId: (): string => {
    let uniqueId = pam.id + "-" + pam.name;
    if(pam.isManager) {
      return "kier-" + uniqueId;
    }
    return uniqueId;
  }
}
console.log(pam.getUniqueId());

```

A zatem, instancję naszego interfejsu tworzymy przygotowując literal obiektowy o nazwie `linda`, określając wartości jego dwóch pól — `name` oraz `id` — i implementując funkcję `getUniqueId`. W kolejnym kroku wyświetlamy na konsoli wynik zwrócony przez wywołanie `linda.getUniqueId`. Następnie tworzymy kolejny obiekt, `pam`, implementujący ten sam interfejs. Zwróć jednak uwagę, że różni się on od obiektu `linda` nie tylko wartościami pól, lecz także implementacją funkcji `getUniqueId`. I właśnie ten przykład pokazuje podstawowe zastosowanie interfejsów: mają one zagwarantować, że tworzone obiekty będą mieć taką samą strukturę, a jednocześnie zapewnić możliwość stosowania w nich różnych implementacji. W ten sposób możemy narzucać ściśle reguły na strukturę typu, a jednocześnie zapewnić mu pewną elastyczność pod względem sposobu działania jego funkcji. Na rysunku 2.12 przedstawiłem wyniki wykonania ostatniego przykładu.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc interfaces

H:\NaukaTypeScriptu\rozdzial02>node interfaces
prc-2-Linda
kier-1-Patrycja

```

**Rysunek 2.12.** Efekty zastosowania interfejsu `Employee`

Kolejną możliwością zastosowania interfejsów jest korzystanie z interfejsów programowania aplikacji (API) przygotowanych przez innych. Czasami podczas korzystania z takich narzędzi nie dysponujemy dokładną dokumentacją dotyczącą używanych typów, a jedynym, co mamy do dyspozycji, jest kod JSON, pozbawiony jakichkolwiek informacji o typach, bądź też ogromny obiekt typu, zawierający bardzo wiele pól, których nigdy nie będziemy musieli używać. W takich sytuacjach można ulec pokusie, by zastosować typ `any` i niczym więcej się nie przejmować. Niemniej jednak, należy starać się podawać deklaracje typów zawsze, kiedy tylko to jest możliwe.

W takich okolicznościach możemy utworzyć interfejs, który będzie uwzględniał wyłącznie te pola, które znamy i na których nam zależy. Następnie możemy zadeklarować, że używane dane są właśnie tego typu. W trakcie pisania kodu TypeScript nie będzie w stanie sprawdzać typu danych, gdyż sieciowe wywołania API będą je zwracały dopiero podczas wykonywania

kodu. Jednak nie ma to wielkiego znaczenia, gdyż TypeScript, który — jak wiemy — zwraca uwagę jedynie na strukturę typów i tak będzie ignorował pola, które nie zostały wymienione w deklaracji, i o ile tylko dane będą zawierać pola wymienione w interfejsie, środowisko uruchomieniowe nie będzie zgłaszać problemów, a my będziemy mogli cieszyć się większym bezpieczeństwem podczas pisania kodu. Korzystając z takiego rozwiązania trzeba jednak zachować dużą ostrożność i zwracać baczną uwagę na właściwe obsługiwane pól mogących zawierać wartości `null` lub `undefined` — na przykład należy w nich używać unii lub testować typy w kodzie.

W tym podrozdziale poznałeś interfejsy i dowiedziałeś się, czym różnią się one od klas. Interfejsów będziemy używali, aby hermetyzować szczegóły klas i wprowadzać luźne powiązania pomiędzy różnymi fragmentami kodu, poprawiając tym samym jego jakość. W następnym podrozdziale pokażę, w jaki sposób klasy i interfejsy pozwalają na stosowanie dziedziczenia, a co za tym idzie, na wielokrotne używanie kodu.

## Dziedziczenie

W tym podrozdziale zajmiemy się **dziedziczeniem** (ang. *inheritance*). W programowaniu obiektowym dziedziczenie jest sposobem na wielokrotne wykorzystywanie kodu. Dziedziczenie pozwala na skrócenie kodu aplikacji i poprawienie jego przejrzystości. Oprócz tego, ogólnie rzecz ujmując, krótszy kod będzie zawierał mniej błędów. A zatem, wszystkie te czynniki sprawiają, że kiedy zaczniemy już korzystać z dziedziczenia, jakość naszego kodu się poprawi.

Jak już zaznaczyłem, dziedziczenie wiąże się przede wszystkim z wielokrotnym stosowaniem kodu. Oprócz tego, pod względem koncepcyjnym, dziedziczenie w programowaniu obiektowym zaprojektowano w taki sposób, by przypominało to występujące w realnym świecie — dzięki temu logiczny tok relacji dziedziczenia można łatwo i intuicyjnie zrozumieć. Przeanalizujemy zatem przykład dziedziczenia. Zacznij od utworzenia nowego pliku o nazwie `classInheritance.ts` i zapisania w nim następującego kodu:

```
class Vehicle {
  constructor(private wheelCount: number) {}

  showNumberOfWheels() {
    console.log(`Liczba kół w pojeździe: ${this.wheelCount} `);
  }
}
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
  }
}
class Automobile extends Vehicle {
  constructor() {
    super(4);
  }
}
const motorCycle = new Motorcycle();
```

```
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();
```

Krótką uwagę dla czytelników, którzy jeszcze nigdy wcześniej nie spotkali się z użyciem znaków odwrotnego apostrofu `` oraz sekwencji `\${}`. Pozwalają one na korzystanie z mechanizmu wstawiania łańcuchów (ang. *string interpolation*), czyli umieszczania jednego łańcucha wewnątrz innego przy wykorzystaniu odpowiednich odwołań.

Jak widać, na początku kodu zdefiniowaliśmy klasę bazową, nazywaną także nadrzędną; w naszym przykładzie jest to klasa `Vehicle`. Ta klasa pełni rolę głównego pojemnika kodu źródłowego, który będzie używany przez wszelkie klasy dziedziczące po tej klasie bazowej, nazywane także klasami pochodnymi. W klasach pochodnych wskazujemy klasę, po której dziedziczą, dzięki podaniu nazwy klasy bazowej po słowie kluczowym `extends`. Ważną rzeczą, na jaką trzeba tu zwrócić uwagę, są konstruktory wszystkich klas pochodnych. Jak widać, w pierwszym wierszu kodu w ich konstruktorach zostało umieszczone wywołanie `super`. Metoda `super()` zwraca instancję klasy nadrzędnej, po której dana klasa dziedziczy. A zatem, w naszym przykładzie będzie to klasa `Vehicle`. Jak widać na przykładzie, każda z klas pochodnych przekazuje do wywołania konstruktora klasy bazowej różną liczbę, która zostanie zapisana jako wartość właściwości `wheelCount`. Pod koniec kodu tworzymy instancje obu klas pochodnych, `Motorcycle` oraz `Automobile`, i dla każdej z nich wywołujemy funkcję `showNumberOfWheels`. Na rysunku 2.13 pokazałem wyniki, które zostaną wyświetlone, kiedy skompilujemy i wykonamy ten kod.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc classInheritance

H:\NaukaTypeScriptu\rozdzial02>node classInheritance
Liczba kół w pojeździe: 2
Liczba kół w pojeździe: 4

```

**Rysunek 2.13.** Wyniki wykonania skryptu `classInheritance.ts`

A zatem, każda klasa pochodna określa własną liczbę kół, zapisywaną we właściwości `wheelCount` klasy nadrzędnej, a jednocześnie żadna z nich nie ma do tej wartości bezpośredniego dostępu. A teraz założmy, że istnieje jakaś ważka przyczyna, dla której klasa pochodna chciałaby pobrać wartość właściwości `wheelCount` klasy nadrzędnej. Na przykład założmy, że konieczna będzie aktualizacja liczby kół w przypadku przebicia dętki. Co możemy zrobić by zapewnić sobie tę możliwość? Moglibyśmy na przykład utworzyć w każdej z klas pochodnych unikalną funkcję, która będzie próbować zaktualizować wartość właściwości `wheelCount`. Zróbmy tak i sprawdźmy, co się stanie. Do klasy `Motorcycle` dodaj nową funkcję o nazwie `updateWheelCount`:

```
class Vehicle {
  constructor(private wheelCount: number) {}

  showNumberOfWheels() {
    console.log(`Liczba kół w pojeździe: ${this.wheelCount}`);
  }
}
```

```

    }
  }
  class Motorcycle extends Vehicle {
    constructor() {
      super(2);
    }
    updateWheelCount(newWheelCount: number){
      this.wheelCount = newWheelCount;
    }
  }
  class Automobile extends Vehicle {
    constructor() {
      super(4);
    }
  }
  const motorCycle = new Motorcycle();
  motorCycle.showNumberOfWheels();
  const autoMobile = new Automobile();
  autoMobile.showNumberOfWheels();

```

Okazuje się, że dodanie do klasy `Motorcycle` metody `updateWheelCount` przedstawionej na powyższym przykładzie powoduje wystąpienie błędu. Czy potrafisz wskazać jego przyczynę? Otóż błąd występuje dlatego, że próbujemy odwołać się do prywatnej składowej klasy nadrzędnej. A zatem, choć klasy pochodne dziedziczą składowe po swojej klasie nadrzędnej, to nie dysponują dostępem do ich składowych prywatnych. Ten sposób działania jest całkowicie prawidłowy i ma na celu wzmocnienie hermetyzacji. A jak możemy obejść ten problem? Spróbuj wprowadzić w kodzie niewielką zmianę, przedstawioną na kolejnym listingu:

```

class Vehicle {
  constructor(protected wheelCount: number) {}

  showNumberOfWheels() {
    console.log(`Liczba kół w pojeździe: ${this.wheelCount}`);
  }
}
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
  }
}
class Automobile extends Vehicle {
  constructor() {
    super(4);
  }
}
const motorCycle = new Motorcycle();
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();

```

Czy w ogóle zauważyłeś, co zmieniliśmy? Masz rację: parametr `wheelCount` w konstruktorze klasy `Vehicle` został teraz zadeklarowany jako chroniony (przy użyciu modyfikatora dostępu `protected`). Użycie tego modyfikatora sprawia, że dostęp do składowej będzie mieć kod tej samej klasy oraz wszystkich jej klas pochodnych.

Zanim przejdziemy do następnych zagadnień, chciałbym przedstawić pojęcie przestrzeni nazw (ang. *namespace*). Przestrzeń nazw pozwoli nam tworzyć swoiste pojemniki na kod, dysponujące wyznaczonym zasięgiem, a dzięki temu oddzielać jedne fragmenty kodu od innych. Wyznaczanie zasięgów przy użyciu przestrzeni nazw pozwala ukrywać wszystko, co znajduje się wewnątrz danej przestrzeni, od kodu znajdującego się poza nią. Pod tym względem przestrzeń nazw przypomina nieco klasy, jednak w przestrzeni nazw może się znaleźć dowolna liczba klas, funkcji, zmiennych oraz wszelkich innych typów. Poniżej przedstawiłem bardzo prosty przykład zastosowania przestrzeni nazw. Utwórz nowy plik o nazwie *namesapces.ts* i zapisz w nim następujący kod:

```
namespace A {
    class FirstClass {}
}

namespace B {
    class SecondClass {}
    const test = new FirstClass();
}
```

Kiedy będziesz wpisywać ten kod, jeszcze przed jego skompilowaniem, zauważysz, że mechanizm IntelliSense VSCode zasygnalizuje problem ze znalezieniem klasy `FirstClass`. Klasa ta nie jest widoczna w przestrzeni nazw B, gdyż została zdefiniowana w przestrzeni nazw A. Właśnie takie jest przeznaczenie przestrzeni nazw — służą one do ukrywania informacji pozostających w zasięgu tylko danej przestrzeni nazw, tak, by nie były widoczne w innych przestrzeniach.

W tym podrozdziale poznałeś zagadnienia związane z dziedziczeniem klas. Dziedziczenie jest niezwykle ważnym narzędziem, wykorzystywanym do wielokrotnego stosowania kodu. W następnym punkcie zajmiemy się klasami abstrakcyjnymi, stanowiącymi jeszcze bardziej elastyczne narzędzie dziedziczenia.

## Klasy abstrakcyjne

Wcześniej wspominałem, że interfejsy przydają się do definiowania kontraktów, jednak nie dysponują implementacjami, czyli działającym kodem. Z kolei klasy dysponują działającym kodem, lecz czasami będziemy potrzebować jedynie sygnatur, a nie całych implementacji. Może się jednak zdarzyć, że w niektórych sytuacjach będziemy chcieli połączyć obie te cechy w jednym typie. W takich sytuacjach, zamiast klas bądź interfejsów przydadzą się nam **klasy abstrakcyjne** (ang. *abstract classes*). Utwórz nowy plik o nazwie *abstractClass.ts* i skopiuj do niego całą zawartość pliku *classInheritance.ts*. Kiedy to zrobisz, VSCode wyświetli kilka błędów, gdyż pojawiły się dwa pliki zawierające klasy i zmienne o tych samych nazwach.

Dlatego zmienimy plik *abstractClass.ts*, a konkretnie: dodamy do niego przestrzeń nazw i zmienimy klasę *Vehicle* na klasę abstrakcyjną. Dodaj zatem do pliku przestrzeń nazw i zmień klasę *Vehicle* zgodnie z przykładem zamieszczonym poniżej:

```
namespace AbstractNamespace {
  abstract class Vehicle {
    constructor(protected wheelCount: number) {}

    abstract updateWheelCount(newWheelCount: number): void;

    showNumberOfWheels() {
      console.log(`Liczba kół w pojeździe: ${this.wheelCount}`);
    }
  }
}
```

Zacznijmy od przestrzeni nazw... Jak widać, cały skopiowany kod umieściliśmy w nawiasach klamrowych wyznaczających zasięg przestrzeni nazw zdefiniowanej przy użyciu zapisu `namespace AbstractNamespace` (zwróć uwagę na to, że nazwa przestrzeni nazw może być dowolna i nie musi zawierać w sobie słowa „namespace”). Jak już wspominałem, przestrzeń nazw to jedynie pojemnik pozwalający nam kontrolować zakres widoczności. Dzięki jej zastosowaniu składowe zdefiniowane w pliku *abstractClass.ts* nie będą należeć do zasięgu globalnego, a tym samym nie będą wpływać na kod umieszczony w innych plikach. Kiedy przyjrzysz się nowemu kodowi klasy *Vehicle*, zauważysz zapewne, że w jej definicji zastosowaliśmy nowe słowo kluczowe — `abstract`. Oznacza ono, że tworzona klasa będzie klasą abstrakcyjną. Oprócz tego dodaliśmy do niej także nową funkcję o nazwie `updateWheelCount`. Na samym początku deklaracji tej funkcji umieściliśmy słowo kluczowe `abstract`; oznacza ono, że funkcja ta nie będzie mieć żadnej implementacji w klasie *Vehicle* i będzie musiała zostać zaimplementowana w klasach pochodnych.

Poniżej naszej nowej, abstrakcyjnej klasy *Vehicle* znajdują się jej dwie klasy pochodne — *Motorcycle* oraz *Automobile*. Zmodyfikuj je tak, by były zgodne z kodem przedstawionym poniżej:

```
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
    console.log(`Motocykl ma ${this.wheelCount} koła.`);
  }
}
class Automobile extends Vehicle {
  constructor() {
    super(4);
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
    console.log(`Motocykl ma ${this.wheelCount} koła.`);
  }
  showNumberOfWheels() {
    console.log(`Liczba kół w pojeździe: ${this.wheelCount} `);
  }
}
```

W ostatnim fragmencie kodu tworzymy instance obu klas pochodnych i wywołujemy ich metody `updateWheelCount`:

```
const motorCycle = new Motorcycle();
motorCycle.showNumberOfWheels(1);
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels(3);
}
```

Jak widać, implementację abstrakcyjnej składowej `updateWheelCount` umieściliśmy w klasach pochodnych. Tę możliwość zapewniają nam klasy abstrakcyjne, które mogą działać jak zwykłe klasy, czyli określać implementacje składowych, oraz jak interfejsy, czyli podawać jedynie reguły przyszłych implementacji, które zostaną określone w klasach pochodnych. Trzeba zauważyć, że ze względu na to, że klasy abstrakcyjne zawierają składowe abstrakcyjne, nie można tworzyć instancji tych klas.

Co więcej, jeśli przyjrzymy się teraz klasie `Automobile`, zauważysz zapewne, że dysponuje ona własną implementacją funkcji `showNumberOfWheels`, choć nie jest to funkcja abstrakcyjna. Zdefiniowanie tej funkcji w klasie `Automobile` stanowi przykład tak zwanego **prześlania** (ang. *overriding*), czyli możliwości zdefiniowania w klasie pochodnej unikalnej implementacji składowej zdefiniowanej w klasie nadrzędnej.

W tym punkcie rozdziału opisałem różne rodzaje dziedziczenia, jakie zapewniają klasy. Opanowanie zasad dziedziczenia pozwoli na wielokrotne wykorzystanie kodu oraz zmniejszenie nie tylko jego wielkości, lecz także liczby występujących w nim błędów. W następnym punkcie przedstawię możliwości dziedziczenia, jakie zapewniają interfejsy, wyjaśnię także, czym różnią się one od tych, które dają klasy.

## Interfejsy

Jak już wyjaśniłem wcześniej, **interfejsy** są sposobem określania reguł odnoszących się do typu danych. Pozwalają one oddzielać implementację od definicji, umożliwiając tym samym wprowadzanie abstrakcji, która z kolei jest jedną z podstawowych zasad programowania obiektowego, mającą wpływ na jakość kodu. Przekonajmy się zatem, w jaki sposób można używać interfejsów do jawnego dziedziczenia i zapewnienia odpowiedniej struktury kodu.

Interfejsy w języku TypeScript pozwalają na określanie sygnatur typów składowych interfejsów, jednak nie pozwalają definiować żadnych implementacji. Wcześniej przedstawiłem już sposoby używania interfejsów do tworzenia niezależnych obiektów, jednak tym razem skoncentrujemy się na zastosowaniu interfejsów jako mechanizmu dziedziczenia i wielokrotnego stosowania kodu. Utwórz nowy plik o nazwie `interfaceInheritance.ts` i zapisz w nim następujący kod:

```
namespace InterfaceNamespace {
  interface Thing {
    name: string;
    getFullName: () => string;
  }
}
```

```
interface Vehicle extends Thing {
  wheelCount: number;
  updateWheelCount: (newWheelCount: number) => void;
  showNumberOfWheels: () => void;
}
```

Jak widać, po deklaracji przestrzeni nazw definiujemy interfejs o nazwie `Thing`, a po nim kolejny interfejs o nazwie `Vehicle`, który, dzięki zastosowaniu słowa kluczowego `extends`, dziedziczy po interfejsie `Thing`. Wprowadziłem to rozwiązanie do przykładu celowo, aby pokazać, że interfejsy mogą dziedziczyć po innych interfejsach. Interfejs `Thing` ma dwie składowe, `name` oraz `getFullName`. Jednak, jak widać na przykładzie, choć interfejs `Vehicle` dziedziczy po `Thing`, w jego kodzie nie ma żadnej wzmianki o tych dwóch składowych. Wynika to z faktu, że `Vehicle` jest interfejsem, a — jak wiadomo — interfejsy nie mogą zawierać żadnych implementacji. Jeśli jednak przyjrzymy się dokładniej zamieszczonemu poniżej kodowi klasy `Motorcycle`, to zauważymy, że implementuje on interfejs `Vehicle` i zawiera wszystkie niezbędne implementacje składowych tego interfejsu:

```
class Motorcycle implements Vehicle {
  name: string;
  wheelCount: number;
  constructor(name: string) {
    // W przypadku implementacji interfejsów nie trzeba
    // wywoływać konstruktora klasy bazowej
    this.name = name;
  }
  updateWheelCount(newWheelCount: number) {
    this.wheelCount = newWheelCount;
    console.log(`Pojazd ma ${this.wheelCount} kół.`);
  }
  showNumberOfWheels() {
    console.log(`Liczba kół w pojeździe ${this.wheelCount}`);
  }
  getFullName() {
    return "MC-" + this.name;
  }
}
const moto = new Motorcycle("moto-dla-początkujących");
console.log(moto.getFullName());
}
```

Kiedy skompilujemy ten kod i wykonamy go, uzyskamy takie wyniki, jak te przedstawione na rysunku 2.14.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
H:\NaukaTypeScriptu\rozdzial02>tsc interfaceInheritance
H:\NaukaTypeScriptu\rozdzial02>node interfaceInheritance
MC-moto-dla-początkujących
```

Rysunek 2.14. Wyniki wykonania skryptu `interfaceInheritance.ts`



Interfejsy nie zapewniają możliwości wielokrotnego wykorzystywania kodu w sposób bezpośredni, gdyż nie pozwalają na określanie implementacji. Niemniej jednak, z punktu widzenia wielokrotnego stosowania kodu, korzystanie z interfejsów i tak jest korzystne, gdyż w jawny sposób określają one oczekiwania odnośnie do tego, jakie dane kod ma pobierać i zwracać. Ukrywanie implementacji przy użyciu interfejsu jest także korzystne z punktu widzenia wprowadzania hermetyzacji i abstrakcji, a jak wiemy, są to kolejne dwie ważne zasady programowania obiektowego.

W przypadku programowania w języku TypeScript, warto korzystać ze wszelkich dostępnych w nim możliwości modelu dziedziczenia. Interfejsów można używać, by zapewniać abstrakcję szczegółów implementacyjnych. Z kolei modyfikatory dostępu `private` i `protected` pozwalają na hermetyzację danych. Pamiętaj, że kiedy nadejdzie czas na skompilowanie kodu i przekształcenie go do postaci kodu JavaScript, kompilator TypeScriptu wykona wszystkie niezbędne czynności konieczne do wygenerowania kodu korzystającego z modelu dziedziczenia przez prototyp. Niemniej jednak, podczas pisania kodu, powinienś używać wszystkich usprawnień, jakie zapewnia język TypeScript, gdyż tylko w ten sposób będziesz mógł skorzystać ze wzbogaconych możliwości programowania.

## Typy generyczne

Typy generyczne (ang. *generics*), nazywane także *typami ogólnymi*, zapewniają możliwość umieszczania w definicji typu innego, skojarzonego typu wybieranego przez jego użytkownika, a nie narzuconego przez twórcę. Dzięki temu typ ma swoją strukturę oraz reguły, a jednocześnie zapewnia pewną elastyczność. Typy generyczne nabiorą znacznie większego znaczenia później, kiedy zaczniemy używać Reacta, dlatego też warto je poznać.

Typów generycznych można używać w funkcjach, klasach oraz interfejsach. Przeanalizujemy przykład zastosowania typu generycznego w funkcji. Utwórz plik o nazwie *functionGeneric.ts* i zapisz w nim następujący kod:

```
function getLength<T>(arg: T): number {
    if (arg.hasOwnProperty("length")) {
        return arg["length"];
    }
    return 0;
}

console.log(getLength<number>(22));
console.log(getLength("Witaj, świecie!"));
```

Na samym początku kodu znajduje się definicja funkcji `getLength<T>`. Ta funkcja używa typu generycznego, który informuje kompilator, że w każdym miejscu, gdzie występuje typ `T`, należy oczekiwać dowolnego możliwego typu. Wewnątrz sposób działania tej funkcji polega na sprawdzeniu, czy parametr `arg` dysponuje polem o nazwie `length`, a jeśli tak, to funkcja zwraca jego wartość. Jeśli pole `length` nie jest dostępne, to funkcja zwraca wartość `0`. Na końcu przykładu dwukrotnie wywołujemy funkcję `getLength`: za pierwszym razem

przekazujemy do niej liczbę, a za drugim łańcuch. Jak widać, w przypadku pierwszego wywołania, w którym przekazywana jest liczba, jawnie podajemy oznaczenie typu, natomiast w drugim wywołaniu, w którym przekazywany jest argument typu `string`, takiego oznaczenia typu nie ma. W pierwszym przypadku podałem typ tylko po to, by pokazać, że można go określić jawnie, jednak zazwyczaj kompilator może określić go samodzielnie, na podstawie kodu.

Jednak przykład ten powoduje pewien problem; otóż sprawdzenie, czy dana przekazana do funkcji dysponuje polem `length` wymaga użycia dodatkowego kodu. Z tego względu kod jest dłuższy niż to konieczne, a jego wykonywanie zajmuje więcej czasu. Spróbujmy zatem zaktualizować ten kod w taki sposób, by ta dodatkowa funkcja nie była wykonywana, jeśli argument nie dysponuje właściwością `length`. W pierwszej kolejności umieść całą dotychczasową zawartość pliku w komentarzu, a następnie poniżej zapisz następujący fragment kodu:

```
interface HasLength {
  length: number;
}

function getLength<T extends HasLength>(arg: T): number {
  return arg.length;
}

console.log(getLength<number>(22));
console.log(getLength("Witaj, świecie!"));
```

Ta nowa wersja kodu jest dość podobna do poprzedniej, jednak zastosowaliśmy w niej interfejs `HasLength`, który ogranicza typy, jakie można przekazywać w wywołaniu funkcji `getLength`. Zastosowany zapis, `T extends HasLength`, informuje kompilator, że niezależnie od tego, czym jest typ `T`, musi on dziedziczyć po typie `HasLength`, bądź też być tym typem, a to z kolei w praktyce oznacza, że przekazany typ musi dysponować właściwością `length`. Dlatego tym razem pierwsze z dwóch wywołań umieszczonych na końcu kodu, to, w którym przekazujemy argument typu `number`, spowoduje zgłoszenie błędu, gdyż liczby nie mają właściwości `length`. Drugie wywołanie, w którym używany typu `string`, działa bez problemów.

A teraz przyjrzyjmy się kolejnemu przykładowi, w którym zastosujemy interfejsy i klasy. Utwórz nowy plik o nazwie `classGeneric.ts` i zapisz w nim następujący kod:

```
namespace GenericNamespace {
  interface Wheels {
    count: number;
    diameter: number;
  }

  interface Vehicle<T> {
    getName(): string;
    getWheelCount: () => T;
  }
}
```

Jak widać, na samym początku definiujemy dwa interfejsy. Pierwszy z nich, `Wheels`, określa informacje o kołach. Drugi, `Vehicle<T>`, jest interfejsem generycznym typu `T`, przy czym parametr typu `T` może oznaczać dowolny inny typ.

Dalszą część kodu pliku *classGeneric.ts* stanowi definicja klasy *Automobile*, która implementuje interfejs *Vehicle* z parametrem typu *Wheel*, co kojarzy typ *Wheel* z klasą *Automobile*. I w końcu ostatnią definiowaną klasą jest *Chevy*, która dziedziczy po *Automobile* i określa potrzebne wartości domyślne:

```
class Automobile implements Vehicle<Wheels> {
    constructor(private name: string, private wheels: Wheels){}

    getName(): string {
        return this.name;
    }

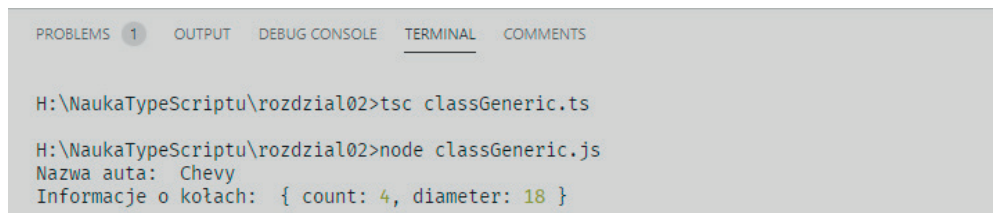
    getWheelCount(): Wheels {
        return this.wheels;
    }
}

class Chevy extends Automobile {
    constructor() {
        super("Chevy", { count: 4, diameter: 18 });
    }
}
```

I w końcu, po tych wszystkich definicjach interfejsów i klas, umieść ostatni fragment kodu, w którym tworzymy instancję klasy *Chevy* i wyświetlamy kilka informacji o niej:

```
const chevy = new Chevy();
console.log("Nazwa auta: ", chevy.getName());
console.log("Informacje o kołach: ", chevy.getWheelCount());
}
```

Ten kod można skompilować, a jego wykonanie zwróci wyniki przedstawione na rysunku 2.15.



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc classGeneric.ts

H:\NaukaTypeScriptu\rozdzial02>node classGeneric.js
Nazwa auta: Chevy
Informacje o kołach: { count: 4, diameter: 18 }
```

**Rysunek 2.15.** Wyniki wykonania skryptu *classGenerics.ts*

Jak widać, hierarchia dziedziczenia w tym przykładzie ma kilka poziomów głębokości, jednak nasz kod i tak może pomyślnie zwrócić prawidłowy wynik. Choć konkretne szczegóły kodu, z którym będziesz się spotykał w rzeczywistości będą zapewne inne, to jednak w programowaniu obiektowym takie wielopoziomowe hierarchie dziedziczenia spotyka się bardzo często.

W tym podrozdziale poznałeś sposoby używania typów generycznych w funkcjach oraz w klasach i interfejsach. Typy generyczne są powszechnie używane podczas tworzenia aplikacji Reacta, jak również w niektórych pakietach środowiska Node. Dlatego też ich znajomość na pewno Ci się przyda podczas lektury dalszych rozdziałów książki. Na zakończenie

tego rozdziału, w jego ostatniej części, zajmiemy się kilkoma dodatkowymi zagadnieniami, takimi jak stosowanie najnowszych możliwości języka TypeScript, czy konfigurowanie kompilatora.

## Prezentacja najnowszych możliwości języka i konfigurowania kompilatora

W tym podrozdziale przedstawię wybrane spośród nowszych możliwości języka TypeScript oraz opiszę możliwości konfigurowania kompilatora TypeScriptu. Zdobycie tych informacji pozwoli Ci na pisanie bardziej przejrzystego, łatwiejszego do zrozumienia kodu, co oczywiście będzie bardzo korzystne podczas pracy w zespole. Z kolei wykorzystanie opcji konfiguracyjnych kompilatora TypeScriptu pozwoli zapewnić, że będzie on działał w sposób, który uznamy za optymalny na potrzeby tworzonego projektu.

### Łączenie opcjonalne

Zacznę od przedstawienia **łączenia opcjonalnego** (ang. *optional chaining*). Ta możliwość pozwala na pisanie prostszego kodu, a jednocześnie chroni przed pewną niewielką klasą błędów związanych ze stosowaniem wartości `null`. Utwórz plik *optionalChaining.ts* i zapisz w nim następujący kod:

```
namespace OptionalChainingNS {
  interface Wheels {
    count?: number;
  }

  interface Vehicle {
    wheels?: Wheels;
  }

  class Automobile implements Vehicle {
    constructor(public wheels?: Wheels) {}
  }

  const car: Automobile | null = new Automobile({
    count: undefined
  });
  console.log("Auto: ", car);
  console.log("Informacje o kołach: ", car?.wheels);
  console.log("Liczba kół: ", car?.wheels?.count);
}
```

Jak widać, w tym przykładzie używanych jest jednocześnie kilka typów. Zmienna `car` dysponuje właściwością `wheels`, która z kolei dysponuje właściwością `count`. W końcowej części kodu, w której wyświetlamy na konsoli informacje o tym obiekcie widać, że odwołania do tych właściwości są łączone ze sobą. Na przykład w ostatnim wywołaniu funkcji `console.log`

używamy wyrażenia o postaci `car?.wheels?.count`. Właśnie taki zapis, wykorzystujący operator `?.`, jest nazywany łączeniem opcjonalnym. Zastosowanie znaku zapytania oznacza istnienie możliwości, że obiekt będzie mieć wartość `null` lub `undefined`. Jeśli faktycznie okaże się, że obiekt ma wartość `null` lub `undefined`, to przetwarzanie wyrażenia zakończy się na tym obiekcie i zostanie zwrócona jego wartość — dalsza część wyrażenia zostanie pominięta, lecz nie spowoduje to zgłoszenia żadnego błędu.

A zatem, gdybyśmy chcieli napisać ostatnie wywołanie funkcji `console.log` w starym stylu, musielibyśmy zastosować rozbudowany kod warunkowy, aby upewnić się, że nie doprowadzimy do zgłoszenia błędu odwołując się do właściwości obiektu, który może przyjąć wartość `undefined`. Zapewne zastosowalibyśmy do tego celu operator trójargumentowy, a cały kod mógłby przypominać instrukcję przedstawioną poniżej:

```
const count = !car ? 0
  : !car.wheels ? 0
  : !car.wheels.count ? 0
  : car.wheels.count;
```

Nie ma wątpliwości, że taki kod jest trudny zarówno do napisania, jak i do zrozumienia. Zastosowanie operatora łączenia opcjonalnego sprawia, że pozwalamy kompilatorowi zatrzymać przetwarzanie wyrażenia w momencie napotkania wartości `null` lub `undefined` i zwrócenia jej jako wyniku. Możemy w ten sposób uniknąć pisania kodu bardzo rozbudowanego i potencjalnie podatnego na błędy.

## Scalanie wartości pustych

Scalanie wartości pustych (ang. *nullish coalescing*) jest po prostu uproszczonym zapisem operatora trójargumentowego. Z tego względu ta nowa możliwość jest bardzo prosta. Poniżej przedstawiłem przykład jej użycia:

```
const val1 = undefined;
const val2 = 10;
const result = val1 ?? val2;
console.log(result);
```

Całe wyrażenie jest przetwarzane od lewej do prawej i oznacza, że jeśli `val1` jest różne od `null` lub `undefined` i ma jakąś faktyczną wartość, to zostanie ona zwrócona jako wartość wyrażenia. Jeśli jednak `val1` nie ma wartości, to jako wartość wyrażenia zostanie zwrócona wartość `val2`. A zatem, po skompilowaniu i wykonaniu powyższego fragmentu kodu, na konsoli zostałaby wyświetlona liczba 10.

Można się zastanawiać, czy scalanie wartości pustych jest tym samym co operator `||`? Faktycznie, między tymi rozwiązaniami istnieją pewne podobieństwa, jednak działanie operatora scalania wartości pustych podlega większym ograniczeniom. W przypadku zastosowania operatora alternatywy logicznej sprawdzana jest jedynie „prawdziwość”. W języku JavaScript koncepcja „prawdziwości” wiąże się z traktowaniem różnych wartości jako „prawdziwe” lub „fałszywe”. Na przykład każda z wartości — 0, true, false, undefined oraz "" — ma w języku JavaScript swój logiczny odpowiednik prawdy lub fałszu. Natomiast w przypadku scalania wartości pustych sprawdzane jest jedynie występowanie wartości `null` lub `undefined`.

## Konfigurowanie TypeScriptu

Informacje konfiguracyjne określające sposób działania kompilatora TypeScriptu można przekazywać w wierszu poleceń, bądź też, co jest znacznie częściej stosownym rozwiązaniem, można je zapisywać w pliku *tsconfig.json*. W przypadku podawania ich w wierszu poleceń, wywołanie kompilatora będzie wyglądać podobnie, jak na poniższym przykładzie:

```
tsc plikts.ts -lib 'es5, dom'
```

To polecenie informuje kompilator, że należy zignorować plik *tsconfig.json* i zastosować wyłącznie opcje podane w wierszu polecenia, czyli konkretnie opcję `-lib` określającą używaną wersję języka JavaScript; dodatkowo polecenie nakazuje skompilowanie tylko jednego, podanego pliku *.ts*. Jeśli polecenie będzie zawierało wyłącznie nazwę kompilatora, *tsc*, to TypeScript poszuka pliku konfiguracyjnego *tsconfig.json* i użyje zapisanych w nim ustawień, a dodatkowo skompiluje wszystkie pliki *.ts*, które uda mu się znaleźć.

Kompilator języka TypeScript udostępnia bardzo wiele opcji, dlatego też nie opiszę tutaj ich wszystkich — ograniczę się do przedstawienia jedynie kilku najważniejszych (kiedy zaczniemy tworzyć aplikacje Reacta, przedstawię plik *tsconfig.json*, którego będziemy używać):

- `--lib` — ta opcja służy do określania wersji języka JavaScript używanej podczas tworzenia projektu;
- `--target` — ta opcja określa wersję języka JavaScript, z którą ma być zgodny generowany kod;
- `--noImplicitAny` — użycie tej opcji nie zezwala na stosowanie typu `any`, jeśli nie został on jawnie zadeklarowany;
- `--outDir` — ta opcja określa katalog, w którym będą zapisywane generowane pliki JavaScript;
- `--outFile` — ta opcja określa nazwę końcowego, generowanego pliku JavaScript;
- `--rootDirs` — ta tablica określa nazwy katalogów zawierających źródłowe pliki *.ts*;
- `--excludes` — ta tablica zawiera nazwy katalogów i plików, których nie należy kompilować;
- `--includes` — ta tablica zawiera nazwy katalogów i plików, które należy skompilować.

W tym punkcie przedstawiłem jedynie pobieżną prezentację wybranych możliwości języka TypeScript, jak również kilka opcji konfiguracyjnych kompilatora TypeScriptu. Niemniej jednak, przedstawione tu najnowsze możliwości języka oraz opcje konfiguracyjne są bardzo istotne i w kolejnych rozdziałach, kiedy już zaczniemy pisać kod aplikacji, będziemy ich bardzo często używać.

## Podsumowanie

W tym rozdziale nieco dokładniej nauczyłeś się języka TypeScript. Poznałeś różne typy, które są w nim dostępne, jak również nauczyłeś się definiować własne. Dowiedziałeś się także, jak pisać kod obiektowy w TypeScriptie. Ten rozdział był długi i całkiem trudny, jednak zamieszczone w nim informacje stanowią nieodzowny fundament, na którym będziemy bazować w dalszej części książki podczas pisania aplikacji.

W następnym rozdziale przedstawię najważniejsze cechy tradycyjnego języka JavaScript, jak również wybrane spośród najnowszych możliwości wprowadzonych w jego najnowszych wersjach. Ponieważ TypeScript jest w rzeczywistości nadzbiorem JavaScriptu, bardzo ważnym jest, by znać i rozumieć aktualne możliwości JavaScriptu, gdyż jest to niezbędne, by móc w pełni korzystać z możliwości, jakie zapewnia TypeScript.

# Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript

W tym rozdziale przedstawię najważniejsze możliwości języka JavaScript, dostępne w jego wersji ES6+ (znak „+” oznacza, że informacje te odnoszą się także do nowszych wersji języka). Koniecznie musisz zrozumieć, że choć niniejsza książka jest poświęcona programowaniu w języku TypeScript, to jednak oba te języki wzajemnie się uzupełniają. Oznacza to, że TypeScript wcale nie zastępuje JavaScriptu. Stanowi raczej jego rozszerzenie i uzupełnienie, a jego dodatkowe możliwości sprawiają, że TypeScript jest lepszym językiem od JavaScriptu. Jednak ścisły związek obu języków sprawia, że musimy także poznać najważniejsze możliwości i cechy języka JavaScript. W tym rozdziale zajmiemy się takimi zagadnieniami, jak: zasięgi zmiennych oraz nowe słowa kluczowe `const` i `let`. Opiszę także szczegółowo słowo kluczowe `this` i wyjaśnię, jak — w razie konieczności — można zmieniać obiekt, do którego się ono odwołuje. Oprócz tego przedstawię także wiele innych ważnych możliwości języka JavaScript, takich jak nowe funkcje tablic oraz słowa kluczowe `async` i `await`. Informacje te pozwolą Ci zdobyć solidną podstawę do programowania w języku TypeScript.



W tym rozdziale zajmiemy się następującymi zagadnieniami:

- przedstawieniem rodzajów zmiennych w języku ES6 i zasięgów zmiennych w JavaScriptcie;
- przedstawieniem funkcji strzałkowych;
- przedstawieniem sposobów zmiany kontekstu słowa kluczowego `this`;
- przedstawieniem rozproszenia, destrukuryzacji i reszty;
- przedstawieniem nowych funkcji tablic;
- prezentacją nowych typów kolekcji;
- przedstawieniem słów kluczowych `async` i `await`.

## Wymagania techniczne

Wymagania techniczne, które musisz spełnić przed przystąpieniem do lektury tego rozdziału, są takie same jak w przypadku rozdziału 2., pt. „Prezentacja języka TypeScript”. Powinieneś dysponować podstawową znajomością języka JavaScript oraz technologii internetowych. Podobnie jak w poprzednim rozdziale, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, anglojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial03* (*Chap3*).

Zanim przystąpisz do lektury, przygotuj środowisko robocze:

1. Przejdź do katalogu *NaukaTypeScriptu* i utwórz w nim podkatalog *rozdzial03*.
2. Uruchom program VSCode i wybierz opcje *File/Open*, a następnie otwórz utworzony przed chwilą katalog *rozdzial03*. Następnie wybierz z menu opcje *View/Terminal* — w efekcie, wewnątrz okna VSCode zostanie wyświetlony panel terminala.
3. W panelu terminala wpisz polecenie `npm init` — użyłeś go już w poprzednim rozdziale, by zainicjować nowy projekt npm — i zaakceptuj wszystkie ustawienia domyślne (możesz także wykonać polecenie `npm init -y`, aby automatycznie zaakceptować ustawienia domyślne).
4. Wykonaj polecenie `npm install typescript` (analogicznie do poprzednich rozdziałów), aby zainstalować język TypeScript.

Teraz już jesteś gotowy do dalszej lektury.

# Poznawanie rodzajów zmiennych w ES6 oraz zasięgów w języku JavaScript

W tym podrozdziale opiszę reguły związane z zasięgami w języku JavaScript i przedstawię nowe rodzaje zmiennych, które upraszczają i rozwiązują niektóre z problemów związanych z zasięgami. Zamieszczone tu informacje są przydatne, gdyż jako programista nieustannie będziesz tworzyć zmienne, ważne jest zatem, byś rozumiał, gdzie poszczególne zmienne będą widoczne i w jaki sposób te zasięgi można zmieniać.

W większości języków programowania zasięg zmiennych jest wyznaczany na podstawie nawiasów klamrowych lub instrukcji określających *początek* i *koniec* zasięgu. Jednak w języku JavaScript zasięgiem zmiennej jest ciało funkcji, w której zmienna ta została zadeklarowana. Oznacza to, że jeśli zmienna została zadeklarowana przy użyciu słowa kluczowego `var` wewnątrz funkcji, to będzie dostępna wyłącznie wewnątrz tej funkcji. Przeanalizujmy przykład prezentujący działanie zasięgów w JavaScriptcie. Utwórz plik *functionBody.ts* i zapisz w nim następujący kod:

```
if (true) {  
  var val1 = 1;  
}  
  
function go() {  
  var val2 = 2;  
}  
  
console.log(val1);  
console.log(val2);
```

Edytor VSCode powinien zaznaczyć ostatnią instrukcję, `console.log(val2)`, informując, że występuje w niej jakiś problem, natomiast pierwsze wywołanie, `console.log(val1)`, jest prawidłowe. Być może pomyślałeś, że ze względu na to, że zmienna `val1` została zadeklarowana wewnątrz nawiasów klamrowych instrukcji `if`, to nie będzie ona dostępna w dalszej części kodu. Jednak, jak pokazuje przykład, tak nie jest. Z kolei zasięg zmiennej `val2` jest ograniczony przez funkcję `go`, więc zmienna `val2` nie będzie dostępna poza nią. Ten przykład pokazuje, że jeśli chodzi o deklaracje zmiennych przy użyciu słowa kluczowego `var`, ich zasięg jest wyznaczany na podstawie funkcji, w których zostały zadeklarowane.

Ta cecha języka JavaScript jest źródłem wielu błędów i problemów. Dlatego też w języku ES6 wprowadzono nowe słowa kluczowe, służące do deklarowania zmiennych: `const` oraz `let`. Poniżej wyjaśnię ich działanie i przedstawię je na przykładach.

Zmienne tworzone przy użyciu słowa kluczowego `const` charakteryzują się zasięgiem blokowym. Zasięg ten jest wyznaczany na podstawie par nawiasów klamrowych. W poprzednim przykładzie zasięg blokowy wyznaczałyby nawiasy klamrowe instrukcji `if`. Co więcej, jak sama nazwa wskazuje, słowo kluczowe `const` tworzy zmienne o ustalonej wartości; oznacza to, że po początkowym ustawieniu wartości takiej zmiennej, nie można jej już modyfikować. W przypadku języka JavaScript oznacza to, że nie można zmienić danej przypisanej do zmiennej. Jednak samą zawartość tej danej można zmieniać. Początkowo dość

trudno to sobie wyobrazić, dlatego najlepiej będzie przedstawić odpowiedni przykład. Utwórz nowy plik o nazwie *const.ts* i zapisz w nim poniższy kod:

```
namespace constants {
  const val1 = 1;
  val1 = 2;

  const val2 = [];
  val2.push('Witaj!');
}
```

Po wpisaniu tego kodu w edytorze VSCode, instrukcja przypisania `val1 = 2` zostanie oznaczona jako błędna, w odróżnieniu od wywołania `val2.push('Witaj!')`, które zostanie uznane za prawidłowe. Powodem takiego działania jest to, że zmiennej `val1` jest przypisywana całkowicie nowa wartość, co nie jest dozwolone. Z drugiej strony zmienna `val2` zawiera tablicę i samo przypisanie tablicy do zmiennej nie ulega zmianie, dodajemy natomiast nowy element do tej samej tablicy. A to jest dozwolone.

A teraz przyjrzyjmy się drugiemu z nowych słów kluczowych: `let`. Zmienne tworzone przy jego użyciu, podobnie jak te tworzone przy użyciu `const`, mają zasięg blokowy. Jednak wartości zmiennych tworzonych przy użyciu `let` można dowolnie zmieniać (oczywiście, w przypadku kodu pisanego w języku TypeScript, trzeba będzie także zadbać o zgodność typów). Przeanalizujmy zatem przykład użycia tego słowa kluczowego. Utwórz nowy plik, *let.ts*, i zapisz w nim następujący kod:

```
namespace lets {
  let val1 = 1;
  val1 = 2;

  if(true) {
    let val2 = 3;
    val2 = 3;
  }

  console.log(val1);
  console.log(val2);
}
```

W tym przykładzie tworzymy dwie zmienne, używając przy tym słowa kluczowego `let`. Pierwsza z nich, `val1`, jest tworzona poza blokiem, natomiast druga, `val2`, w bloku wyznaczonym przez instrukcję `if`. Jak widać, jedynie w wywołaniu `console.log(val2)` występuje błąd, spowodowany tym, że próbujemy odwołać się do zmiennej `val2` poza blokiem, w którym została ona utworzona.

Powstaje zatem pytanie: którego sposobu deklarowania zmiennych należy używać? Aktualnie w środowisku programistów JavaScriptu za najlepszą praktykę uznaje się stosowanie słowa kluczowego `const`, gdyż niezmiennosc, jaką ono zapewnia, jest korzystna, a co więcej — stosowanie tego słowa kluczowego pozwala na nieznaczny wzrost wydajności wykonywania kodu. Niemniej jednak, jeśli wiemy, że konieczne będzie modyfikowanie wartości zmiennej, to zamiast `const` należy używać słowa kluczowego `let`. Natomiast stosowania słowa kluczowego `var` należy unikać.

W ten sposób zakończyłem prezentowanie informacji dotyczących zasięgów zmiennych w języku JavaScript oraz dwóch nowych słów kluczowych, `const` i `let`, dodanych w wersji ES6. Zrozumienie zagadnień związanych z zasięgami oraz wiedza, kiedy używać `const`, a kiedy `let`, są ważnymi aspektami nowoczesnego programowania w języku JavaScript. W nowym kodzie pisanym w tym języku, bardzo często będziemy się spotykali z przykładami stosowania tych dwóch słów kluczowych. Kolejnym zagadnieniem, którym się zajmujemy, będzie kontekst `this` oraz funkcje strzałkowe.

## Poznawanie funkcji strzałkowych

Funkcje strzałkowe (ang. *arrow functions*) są nowością wprowadzoną w wersji ES6 języka JavaScript. Najprościej rzecz ujmując, mają one dwa główne zastosowania:

- udostępniają skróconą składnię zapisu funkcji,
- automatycznie sprawiają, że obiektem nadrzędnym funkcji będzie obiekt nadrzędny najbliższego zasięgu, czyli `this`.

Zanim przejdziemy dalej, muszę dokładniej wyjaśnić działanie `this` w języku JavaScript, gdyż zagadnienie to ma kluczowe znaczenie dla wszystkich programistów używających tego języka.

W języku JavaScript obiekt `this`, czyli obiekt właściciela, do którego należą właściwości i metody, może się zmieniać w zależności od kontekstu wywołania. A zatem, jeśli funkcję wywołujemy bezpośrednio, np.: `MyFunction()`, to byłaby ona wywoływana na rzecz nadrzędnego obiektu `this`, czyli obiektu `this` używanego w bieżącym zasięgu. W przypadku wykonywania kodu w przeglądarce, obiektem tym byłby obiekt `window`. Jednak w języku JavaScript funkcje mogą być także używane jako konstruktory nowych obiektów; na przykład: `new MyFunction()`. W takim przypadku, wewnątrz wywołania funkcji obiektem `this` byłby obiekt utworzony przez wywołanie konstruktora `new MyFunction()`.

Aby dokładniej zrozumieć tę bardzo ważną cechę języka JavaScript, przeanalizujmy ją na przykładzie. Utwórz nowy plik o nazwie *testThis.ts* i zapisz w nim następujący kod:

```
function MyFunction () {
    console.log(this);
}

MyFunction();
let test = new MyFunction();
```

Jeśli skompilujesz i wykonasz ten przykład, zostaną wyświetlone wyniki takie jak te przedstawione na rysunku 3.1.

A zatem, jeśli funkcję `MyFunction` wywołamy bezpośrednio, to obiektem nadrzędnym najbliższego zasięgu będzie globalny obiekt `Node`, gdyż w naszym przypadku kod nie jest wykonywany w przeglądarce. Następnie, jeśli używając wywołania `new MyFunction()` stworzymy nowy obiekt, to właśnie do niego będzie się odwoływać `this`, gdyż został on utworzony nie bezpośrednio, lecz przy użyciu funkcji.

```
H:\NaukaTypeScriptu\rozdzial03>tsc testThis

H:\NaukaTypeScriptu\rozdzial03>node testThis
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]
  }
}
MyFunction {}
```

**Rysunek 3.1.** Wyniki wykonania skryptu testThis.ts

Skoro już to sobie wyjaśniliśmy, to wróćmy do prezentacji funkcji strzałkowych. Utwórz nowy plik o nazwie *arrowFunction.ts* i zapisz w nim następujący kod:

```
const myFunc = (message: string): void => {
  console.log(message);
}

myFunc('Witaj!');
```

Kiedy skompilujesz i wykonasz ten kod, zostanie wyświetlone słowo Witaj!. Jak widać, składnia funkcji strzałkowych jest bardzo podobna do składni typu funkcyjnego; niemniej jednak nie są one tym samym. Jeśli dokładniej przyjrzyś się kodowi, zauważysz, że za nawiasami z listą parametrów jest umieszczony dwukropek, a za nim podane określenie typu: `void`. To typ wyniku zwracanego przez funkcję. Z kolei w przypadku typów funkcyjnych, typ wyniku jest podawany za symbolem `=>`.

Jest jeszcze kilka innych zagadnień związanych z funkcjami strzałkowymi, o których należy wspomnieć. Wszystkie funkcje języka JavaScript definiowane w tradycyjny sposób, czyli nie jako funkcje strzałkowe, dysponują dostępem do kolekcji o nazwie `arguments`. Jest to kolekcja zawierająca wszystkie parametry przekazane do funkcji. Z kolei funkcje strzałkowe nie dysponują własną kolekcją `arguments`; mają natomiast dostęp do kolekcji `arguments` ich funkcji nadrzędnej.

Ciało funkcji strzałkowych można zapisywać na kilka różnych sposobów. Poniżej przedstawiłem przykład prezentujący trzy z nich:

```
const func = () => console.log('func');
const func1 = () => ({ name: 'Dawid' });
const func2 = () => {
  const val = 20;
  return val;
}

console.log(func());
console.log(func1());
console.log(func2());
```

Przeanalizujemy dokładniej każdy z tych stylów zapisu ciała funkcji strzałkowych:

- Pierwsza z funkcji, `func`, przedstawia przypadek, w którym ciało funkcji składa się wyłącznie z jednego wiersza kodu, a sama funkcja nie zwraca wyniku. Jak widać, ciało funkcji nie jest zapisane ani w nawiasach klamrowych, ani zwracających.
- Druga z funkcji, `func1`, przedstawia przypadek, w którym ciało funkcji również składa się tylko z jednego wiersza kodu, lecz funkcja zwraca wynik. W tym przypadku użycie instrukcji `return` nie jest konieczne, a nawiasy są potrzebne tylko wtedy, gdy jest zwracany obiekt.
- Funkcja `func2` reprezentuje ostatni przypadek. Jak widać, tym razem nawiasy klamrowe są niezbędne, gdyż ciało funkcji składa się z kilku instrukcji zapisanych w kilku wierszach (to, czy funkcja zwraca wynik, czy nie, nie ma w tym przypadku znaczenia).

W tym podrozdziale przedstawiłem funkcje strzałkowe. W nowoczesnym kodzie pisanym w językach JavaScript i TypeScript są one używane bardzo często, dlatego też na pewno warto jest dobrze je poznać.

## Zmienianie kontekstu `this`

W poprzednim podrozdziale wspominałem już o zmianie obiektu kontekstu `this`. Jak wiadomo, w języku JavaScript funkcje mają dostęp do wewnętrznego obiektu o nazwie `this`, reprezentującego kod, który wywołał tę funkcję. Jednak najbardziej kłopotliwym zagadnieniem związanym z obiektem `this` jest to, że jego wartość może się zmieniać w zależności od sposobu wywołania funkcji. Dlatego też język JavaScript udostępnia funkcje pomocnicze, pozwalające na ustawianie obiektu `this` funkcji na ten, którego chcielibyśmy używać, a nie ten, który został nam dostarczony. Dostępnych jest kilka funkcji zapewniających te możliwości, w tym `apply` oraz `call`, jednak dla nas zdecydowanie najważniejsza będzie funkcja `bind`. Ma ona dla nas tak duże znaczenie dlatego, że jest powszechnie stosowana w komponentach klasowych Reacta. Jak dotąd jest jeszcze nieco zbyt wcześnie na przedstawianie kompletnego przykładu kodu komponentu Reacta, więc ograniczymy się do czegoś prostszego. Utwórz nowy plik o nazwie `bind.ts` i zapisz w nim następujący kod:

```
class A {
  name: string = 'A';
  go() {
    console.log(this.name);
  }
}

class B {
  name: string = 'B';
  go() {
    console.log(this.name);
  }
}
```

```
const a = new A();
a.go();
const b = new B();
b.go = b.go.bind(a);
b.go();
```

Jak widać, w tym przykładzie definiujemy dwie unikalne klasy: A i B. Obie definiują funkcję `go`, która wyświetla nazwę klasy. I teraz, kiedy zmienimy skojarzenie `this` funkcji `go` obiektu `b` na obiekt `a`, to w wywołaniu `console.log(this.name)` jako `this` zostanie użyty obiekt `a`. A zatem, kiedy skompilujemy i wykonamy ten przykład, uzyskamy wyniki przedstawione na rysunku 3.2.

```
H:\NaukaTypeScriptu\rozdzial03>tsc bind
H:\NaukaTypeScriptu\rozdzial03>node bind
A
A
```

**Rysunek 3.2.** Wyniki wykonania skryptu `bind.ts`

Jak widać, wywołanie `a.go()` wyświetli `A`, podobnie jak wywołanie `b.go()`, które także wyświetli `A`, a nie `B`, gdyż wywołując funkcję `bind` zmieniliśmy `this` z `b` na `a`. Warto także zwrócić uwagę na to, że funkcja `bind` umożliwia przekazanie nie tylko nowej wartości `this`, lecz także dowolnej liczby dodatkowych argumentów.

Zapewne zastanawiasz się, jaka jest różnica pomiędzy działaniem funkcji `bind`, `call` oraz `apply`. Otóż funkcja `bind` pozwala zmienić obiekt `this` skojarzony z funkcją, dzięki czemu, kiedy wywołamy tę funkcję później, `this` będzie odwoływać się do wskazanego przez nas obiektu. Z kolei dwie pozostałe funkcje, `apply` i `call`, są używane w momencie, kiedy chcemy wywołać funkcję, i zmieniają `this` tylko na czas tego wywołania. Funkcje te różnią się między sobą tym, że w wywołaniu funkcji `call` można podać dowolną liczbę argumentów, natomiast w wywołaniu `apply` przekazywana jest tablica argumentów. Przeanalizujmy teraz przykład użycia funkcji `call`. W tym celu utwórz nowy plik o nazwie `call.js` i zapisz w nim następujący kod:

```
const callerObj = {
  name: 'Janek'
}

function checkMyThis(age) {
  console.log(`Czym jest this: ${this}`);
  console.log(`Czy mam imię? ${this.name}`);
  this.age = age;
  console.log(`Ile mam lat? ${this.age}`);
}

checkMyThis();
checkMyThis.call(callerObj, 25);
```

W tym przykładzie w pierwszej kolejności tworzymy obiekt o nazwie `callerObj`, zawierający jedną właściwość, `name`, o wartości `'Janek'`. Następnie deklarujemy funkcję `checkMyThis`,

która sprawdza aktualnie dostępny obiekt `this` i wyświetla zapisane w nim imię. Poniżej, na samym końcu skryptu, wykonujemy dwa wywołania. Zwróć uwagę na to, że drugie z nich, `checkMyThis.call`, jest w rzeczywistości wywołaniem funkcji `checkMyThis`. Kiedy wykonamy ten kod, przekonamy się, że zwraca on interesujące wyniki. Zresztą, przekonaj się o tym samemu — wykonaj poniższe polecenie:

```
node call
```

Kiedy to zrobisz, zobaczysz wyniki takie jak te przedstawione na rysunku 3.3.

```
H:\NaukaTypeScriptu\rozdzial03>node call
Czym jest this: [object global]
Czy mam imię? undefined
Ile mam lat? undefined
Czym jest this: [object Object]
Czy mam imię? Janek
Ile mam lat? 25
```

**Rysunek 3.3.** Wyniki wykonania skryptu `call.js`

Pierwsze wywołanie funkcji `checkMyThis` używa domyślnego obiektu globalnego, gdyż nie zmieniliśmy go. Jeszcze raz przypomnę, że w przypadku wykonywania kodu w środowisku Node, będzie to domyślny obiekt `Node`, natomiast w przypadku wykonywania kodu w przeglądarce, będzie to obiekt `window`. Jak widać na rysunku, w przypadku tego pierwszego wywołania właściwości `name` i `age` mają wartość niezdefiniowaną (`undefined`), gdyż globalny obiekt `Node` nie dysponuje właściwościami o tych nazwach, a w wywołaniu funkcji `checkMyThis` nie przekazaliśmy parametru `age`. Z kolei w przypadku drugiego wywołania tej funkcji, tego, w którym zastosowaliśmy funkcję `call`, obiekt `this` został zamieniony na obiekt standardowego typu, który określa imię, Janek (bo taka jest wartość właściwości `name` obiektu `callerObj`), oraz wiek, 25 (gdyż taką wartość przekazaliśmy w wywołaniu funkcji `call`). Warto zwrócić uwagę na to, że kolejność argumentów przekazanych w wywołaniu funkcji `call` odpowiada kolejności parametrów funkcji, którą chcemy wywołać. Funkcja `apply` jest używana w analogiczny sposób, z tą różnicą, że argumenty wywoływanej funkcji są przekazywane w formie tablicy.

W tym podrozdziale opisałem problemy związane z kontekstem `this` oraz sposoby jego zmieniania przy użyciu funkcji `bind`. Funkcji tej będziemy używali bardzo często w dalszej części książki, kiedy zaczniemy tworzyć komponenty Reacta. Jednak nawet pomijając to szczególne zastosowanie, na pewno sam się przekonasz, że w niektórych sytuacjach możliwości zmiany kontekstu `this`, a może nawet wartości przekazywanych do funkcji, mogą być bardzo przydatne.

## Rozproszenie, destrukuryzacja i reszta

W języku ES6+ wprowadzono nowe metody służące do obsługi kopiowania obiektów oraz wyświetlania wartości i parametrów. W znacznym stopniu przyczyniają się one zarówno do skrócenia kodu pisanego w JavaScriptcie, jak i do poprawy jego czytelności. Stosowanie



tych nowych funkcji stało się standardową praktyką w nowoczesnych sposobach programowania w JavaScriptcie, dlatego też bardzo ważne jest, by dobrze je poznać i zrozumieć.

## Rozproszenie, Object.assign oraz Array.concat

Operator **rozproszenia** (ang. *spread*) oraz funkcje `Object.assign` i `Array.concat` są do siebie dosyć podobne. Ogólnie rzecz biorąc, ich działanie polega na scaleniu wielu obiektów lub tablic w jeden obiekt lub tablicę. Niemniej jednak, precyzyjnie analizując te funkcje, okazuje się, że działają one nieco inaczej.

W przypadku obiektów, można je scalać (lub łączyć) na dwa sposoby:

- Przy użyciu operatora rozproszenia, na przykład: `{...obja, ...objb}`. W ten sposób tworzymy niezmodyfikowaną kopię dwóch użytych obiektów, uzyskując w ten sposób jeden nowy obiekt. Warto zwrócić uwagę na to, że w ten sposób można połączyć ze sobą więcej niż tylko dwa obiekty.
- Przy użyciu funkcji `Object.assign(obja, objb)`. W ten sposób dodajemy właściwości obiektu `objb` do obiektu `obja` i zwracamy `obja`. Oznacza to, że obiekt `obja` zostanie zmodyfikowany. Poniżej przedstawiłem przykład przedstawiający zastosowanie obu tych rozwiązań. Utwórz nowy plik o nazwie *spreadObj.ts* i zapisz w nim następujący kod.

```
namespace NamespaceA {
  class A {
    aname: string = 'A';
  }
  class B {
    bname: string = 'B';
  }

  const a = new A();
  const b = new B();
  const c = { ...a, ...b }
  const d = Object.assign(a, b);
  console.log(c);
  console.log(d);

  a.aname = 'a1';
  console.log(c);
  console.log(d);
}
```

W tym przykładzie, po zdefiniowaniu klas `A` i `B`, tworzymy nowy obiekt `c`, używając do tego celu operatora rozproszenia (`...`). Następnie tworzymy kolejny nowy obiekt, `d`, używając do tego celu wywołania `Object.assign`. Spróbujmy teraz wykonać ten przykład. Pamiętaj, by kompilując go zaznaczyć, że generowany kod JavaScript ma być zgodny z wersją ES6 JavaScriptu, gdyż dopiero w niej jest dostępna funkcja `Object.assign`. A zatem, skompiluj i uruchom przykład używając poniższych poleceń:

```
tsc spreadObj --target "es6"
node spreadObj
```

Wyświetlone wyniki przedstawiłem na rysunku 3.4.

```
H:\NaukaTypeScriptu\rozdzial03>tsc spreadObj --target "es6"

H:\NaukaTypeScriptu\rozdzial03>node spreadObj
{ aname: 'A', bname: 'B' }
A { aname: 'A', bname: 'B' }
{ aname: 'A', bname: 'B' }
A { aname: 'a1', bname: 'B' }
```

**Rysunek 3.4.** Wyniki wykonania skryptu `spreadObj.ts`

Jak widać, obiekt `c` dysponuje właściwościami `aname` oraz `bname`, a przy tym jest odrębnym obiektem. Z drugiej strony obiekt `d` jest w rzeczywistości obiektem `a` rozszerzonym o właściwości obiektu `b`, co wyraźnie widać w dalszej części wyników, kiedy po przypisaniu `a.aname = 'a1'` właściwość `aname` obiektu `d` będzie mieć wartość `'a1'`.

Jeśli z kolei chodzi o scalanie, czy też konkatenaację tablic, to można to robić na dwa sposoby:

- Przy użyciu operatora rozproszenia. To rozwiązanie przypomina scalanie obiektów — podane tablice zostają scalone, a cała operacja zwraca nową tablicę. W tym przypadku początkowe tablice użyte podczas scalania nie są w żaden sposób modyfikowane.
- Przy użyciu funkcji `Array.concat`. Funkcja ta pobiera dwie tablice źródłowe i zwraca nową tablicę, stanowiącą ich połączenie. Tablice źródłowe nie są w żaden sposób modyfikowane.

Przyjrzyjmy się teraz przykładowi użycia tych dwóch rozwiązań. Utwórz plik o nazwie `spreadArray.ts` i zapisz w nim poniższy kod:

```
namespace SpreadArray {
    const a = [1,2,3];
    const b = [4,5,6];

    const c = [...a, ...b];
    const d = a.concat(b);
    console.log('c przed zmianą a', c);
    console.log('d przed zmianą a', d);

    a.push(10);
    console.log('a', a);
    console.log('c po zmianie a', c);
    console.log('d po zmianie a', d);
}
```

Jak widać, tablica `c` jest tworzona na podstawie dwóch tablic, `a` i `b`, przy użyciu operatorów rozproszenia. Z kolei tablicę `d` tworzymy używając wywołania `a.concat(b)`. W tym przykładzie obie wynikowe tablice są unikalne i nie są w żaden sposób połączone z tablicami źródłowymi. Kiedy skompilujesz ten przykład i go wykonasz, zostaną wyświetlone wyniki przedstawione na rysunku 3.5.

```
H:\NaukaTypeScriptu\rozdzial03>tsc spreadArray --target "es5"

H:\NaukaTypeScriptu\rozdzial03>node spreadArray
c przed zmianą a [ 1, 2, 3, 4, 5, 6 ]
d przed zmianą a [ 1, 2, 3, 4, 5, 6 ]
a [ 1, 2, 3, 10 ]
c po zmianie a [ 1, 2, 3, 4, 5, 6 ]
d po zmianie a [ 1, 2, 3, 4, 5, 6 ]
```

Rysunek 3.5. Wyniki wykonania skryptu `spreadArray.ts`

Jak widać, wywołanie `a.push(10)` nie ma żadnego wpływu na zawartość tablicy `d` wyświetlaną przez wywołanie `console.log('d po zmianie a', d)`, choć tablica `d` została utworzona poprzez wywołanie funkcji `concat` na rzecz tablicy `a`. Ten przykład wyraźnie pokazuje, że zarówno użycie operatorów rozproszenia, jak i funkcji `Array.concat`, powoduje tworzenie nowych tablic.

## Destrukturyzacja

Destrukturyzacja (ang. *destructuring*) polega na możliwości bezpośredniego wyświetlenia — czy też użycia — wewnętrznych składowych obiektu, bez konieczności odwoływania się do nich za pomocą nazwy obiektu. Już za chwilę wyjaśnię dokładniej, o co w tym chodzi, opierając się na odpowiednim przykładzie, jednak w pierwszej kolejności chciałbym zaznaczyć, że możliwość ta jest bardzo często używana w nowoczesnych sposobach programowania w języku JavaScript, a w szczególności w aplikacjach Reacta tworzonych z wykorzystaniem *hooków* (ang. *hooks*), dlatego też konieczne musisz ją dobrze opanować i nabrać praktyki w jej stosowaniu.

Przyjrzyjmy się teraz przykładowi destrukturyzacji obiektów. W tym przykładzie posłużymy się zwyczajnym plikiem JavaScript, gdyż dzięki temu będzie on łatwiejszy do zrozumienia. Utwórz nowy plik o nazwie *destructuring.js* i zapisz w nim następujący kod:

```
function getEmployee(id) {
  return {
    name: 'Jan',
    age: 35,
    address: 'Skrytów Wsadowych 123',
    country: 'Polska'
  }
}

const { name: fullName, age } = getEmployee(22);
console.log('pracownik:', fullName, age);
```

Wyobraźmy sobie na chwilę, że funkcja `getEmployee` przesyła żądanie na serwer i pobiera informacje o pracowniku na podstawie przekazanego identyfikatora (`id`). Jak widać na przykładzie, zwracany przez nią obiekt pracownika zawiera wiele pól, więc możemy przyjąć, że nie każdy kod, w którym będzie ona wywoływana, będzie używał wszystkich pól tego obiektu. Dlatego też skorzystamy z mechanizmu destrukturyzacji, by wybrać tylko te pola, które nas interesują. Zwróć także uwagę na to, że pobierając wartość pola `name` nadajemy jej nazwę zastępczą `fullName`, używając w tym celu zapisu z dwukropkiem.

Mechanizm destrukuryzacji działa także na tablicach. W tym samym pliku, którego używałeś wcześniej, dopisz następujący fragment kodu:

```
function getEmployeeWorkInfo(id) {
  return [
    id,
    'ul. Biurowa',
    'Francja'
  ]
}

const [id, officeAddress] = getEmployeeWorkInfo(33);
console.log('pracownik:', id, officeAddress);
```

Funkcja `getEmployeeWorkInfo` zwraca tablicę z informacjami o miejscu zatrudnienia pracownika, przy czym informacje te mają postać tablicy. Przykład pokazuje, że mechanizm destrukuryzacji może działać także na tablicach, choć w tym przypadku ważna jest kolejność pobieranych elementów. A teraz przyjrzyjmy się wynikom zwracanym przez te dwie operacje destrukuryzacji. Zwróć uwagę na to, że w tym przypadku wystarczy wykonać skrypt w środowisku Node, gdyż kod zapisaliśmy w pliku JavaScript. Uruchom przykład, używając następującego polecenia:

```
node destructuring.js
```

Wyniki jego wykonania przedstawiłem na rysunku 3.6.

```
H:\NaukaTypeScriptu\rozdzial03>node destructuring.js
pracownik: Jan 35
pracownik: 33 ul. Biurowa
```

**Rysunek 3.6.** Efekty destrukuryzacji

Jak widać, obie funkcje zwracają prawidłowe dane.

## Reszta

**Reszta** (ang. *rest*) to mechanizm pozwalający odwoływać się do niezdefiniowanej liczby parametrów przy użyciu jednego słowa kluczowego, a konkretnie: `...`. Reszta zawsze jest tablicą, więc operując na niej mamy dostęp do wszystkich funkcji obiektów Array. Słowo kluczowe reszty oznacza „wszystkie pozostałe elementy”, a nie „to koniec”. Korzystanie z reszty zapewnia większą elastyczność tworzonych sygnatur funkcji, gdyż daje kodowi wywołującemu możliwość samodzielnego określania, ile parametrów zostanie przekazanych do funkcji. Warto zwrócić uwagę na to, że jedynie ostatni parametr funkcji może być parametrem reszty. Poniżej przedstawiłem przykład wykorzystania tego mechanizmu. Utwórz plik o nazwie `rest.js` i zapisz w nim następujący kod:

```
function doSomething(a, ...others) {
  console.log(a, others, others[others.length - 1]);
}

doSomething(1,2,3,4,5,6,7);
```

Jak widać, parametr `...others` odwołuje się do wszystkich pozostałych parametrów oprócz pierwszego — `a`. Oznacza to, że parametr reszty wcale nie musi być jedynym parametrem funkcji. Jeśli wykonasz ten przykład, wyświetli on wyniki przedstawione na rysunku 3.7.

```
H:\NaukaTypeScriptu\rozdzial03>node rest.js
1 [ 2, 3, 4, 5, 6, 7 ] 7
```

Rysunek 3.7. Efekty zastosowania reszty

Do funkcji `doSomething` przekazywane są dwa parametry: zmienna `a` oraz parametr **reszty**. Działanie funkcji ogranicza się do wyświetlenia na konsoli wartości zmiennej `a`, zawartości parametru reszty (która jest tablicą wartości) oraz wartości ostatniego elementu parametru reszty. Mechanizm reszty nie jest stosowany aż tak często jak operator rozproszenia i destrukuryzacja. Niemniej jednak można się z nim spotkać, dlatego powinieneś go znać.

W tym podrozdziale poznałeś możliwości języka JavaScript umożliwiające pisanie kodu krótszego i łatwiejszego do zrozumienia. W nowoczesnym kodzie JavaScript możliwości te są używane powszechnie, dlatego dokładne ich poznanie i zrozumienie przyniesie Ci wiele korzyści. W następnym podrozdziale przedstawię wybrane spośród najważniejszych i najpopularniejszych sposobów operowania na tablicach, które znacząco ulepszą sposoby, w jakie będziesz przetwarzał tablice.

## Prezentacja wybranych funkcji tablicowych

W tym podrozdziale przyjrzymy się bliżej wielu nowym funkcjom tablicowym, dodanym w wersji ES6 języka JavaScript. Zamieszczone tu informacje są bardzo ważne, gdyż w kodzie JavaScript operacje na tablicach są wykonywane niezwykle często, a dzięki zoptymalizowanej wydajności, stosowanie tych metod jest preferowane względem tworzenia własnych odpowiedników. Oprócz tego stosowanie tych standardowych metod będzie bardziej spójne i łatwiejsze do zrozumienia dla innych programistów należących do zespołu. Wszystkich tych metod będziemy bardzo często używać podczas pisania aplikacji Reacta oraz programów działających w środowisku Node. A zatem, zaczynamy.

### find

Funkcja `find` służy do pobierania pierwszego elementu tablicy, który spełnia określone kryteria. Przyjrzymy się działaniu tej funkcji na przykładzie. Utwórz plik *find.ts* i zapisz w nim następujący kod:

```
const items = [
  { name: 'Jan', age: 20 },
  { name: 'Linda', age: 22 },
  { name: 'Jan', age: 40 }
```

```

]

const jon = items.find((item) => {
  return item.name === 'Jan'
});
console.log(jon);

```

Kiedy przyjrzyś się wywołaniu funkcji `find`, przekonasz się, że jest do niej przekazywana funkcja, która poszukuje elementu mającego właściwość `name` o wartości `'Jan'`. Funkcja ta sprawdza warunek logiczny, porównując wartość właściwości `name` przekazanego obiektu z łańcuchem `'Jan'`. Jeśli warunek zostanie spełniony, funkcja `find` zwróci aktualnie sprawdzany element. Jednak, jak widać na przykładzie, w przeglądanej tablicy znajdują się dwa obiekty spełniające zadany warunek. Skompiluj przykład i wykonaj go, by przekonać się, który z tych dwóch obiektów zostanie zwrócony. W tym celu wykonaj następujące polecenia:

```

tsc find --target "es6"
node find

```

Kiedy wykonasz te polecenia, zostanie wyświetlony wynik pokazany na rysunku 3.8.

```

H:\NaukaTypeScriptu\rozdzial03>tsc find --target "es6"

H:\NaukaTypeScriptu\rozdzial03>node find
{ name: 'Jan', age: 20 }

```

**Rysunek 3.8.** Wyniki wykonania skryptu `find.ts`

Jak widać, zwracany jest pierwszy element tablicy spełniający zadane kryteria — tak właśnie działa funkcja `find`: zwraca wyłącznie pierwszy znaleziony element tablicy.

## filter

Funkcja `filter` jest podobna do `find`, jednak zamiast pierwszego zwraca wszystkie elementy tablicy spełniające podane kryteria. Utwórz nowy plik *filter.ts*, i zapisz w nim poniższy kod:

```

const filterItems = [
  { name: 'Jan', age: 20 },
  { name: 'Linda', age: 22 },
  { name: 'Jan', age: 40 }
]

const results = filterItems.filter((item, index) => {
  console.log(index);
  return item.name === 'Jan'
});
console.log(results);

```

Jak pokazuje przedstawiony przykład, funkcja przekazywana do funkcji `filter` może przyjmować także drugi, opcjonalny parametr, określający indeks aktualnie sprawdzanego elementu tablicy. Niemniej jednak, pomijając tę różnicę, wewnętrzny sposób działania funkcji `filter` i `find` jest bardzo podobny — obie sprawdzają warunek logiczny by znaleźć pasujące

do niego elementy tablicy. Dodatkowo funkcja `filter` zwraca wszystkie znalezione elementy tablicy, co widać w wynikach przedstawionych na rysunku 3.9.

```
H:\NaukaTypeScriptu\rozdzial03>tsc filter --target "es6"
H:\NaukaTypeScriptu\rozdzial03>node filter
0
1
2
[ { name: 'Jan', age: 20 }, { name: 'Jan', age: 40 } ]
```

Rysunek 3.9. Wyniki wykonania skryptu `find.ts`

Jak widać, funkcja `filter` zwraca wszystkie elementy tablicy spełniające podany warunek, którym w naszym przykładzie jest wartość `'Jan'` zapisana we właściwości `name`.

## map

Jeśli poszerzymy nowe funkcje tablicowe wprowadzone w wersji ES6 JavaScriptu pod względem znaczenia dla nowoczesnego stylu programowania, jedną z najważniejszych spośród nich okaże się funkcja `map`. Jest ona bardzo często stosowana w kodzie tworzącym komponenty Reacta, gdzie używa się jej do tworzenia kolekcji komponentów na podstawie tablicy danych. Musisz zwrócić uwagę na to, że funkcja `map` jest zupełnie czym innym niż kolekcja `Map`, którą przedstawię w dalszej części rozdziału. Utwórz teraz nowy plik, *map.ts*, i zapisz w nim następujący kod:

```
const employees = [
  { name: "Tomek", id: 1 },
  { name: "Celina", id: 2 },
  { name: "Robert", id: 3 },
]

const elements = employees.map((item, index) => {
  return `<div>${item.id} - ${item.name}</div>`;
});

console.log(elements);
```

Jak widać, funkcja przekazywana w wywołaniu funkcji `map` pobiera dwa parametry, `item` i `index` (można im nadawać dowolne nazwy, znaczenie ma jedynie ich kolejność) i zapisuje w tablicy nowe wartości. Precyzyjnie rzecz ujmując, w tym przypadku instrukcja `return` oznacza zwrócenie wartości do nowej tablicy, a nie zwrócenie wartości i zakończenie iteracji. Kiedy wykonasz ten przykład, przekonasz się, że wynikiem będzie tablica z fragmentami kodu HTML, którą przedstawiłem na rysunku 3.10.

Być może `map` jest najczęściej stosowaną ze wszystkich funkcji tablicowych wprowadzonych w wersji ES6 JavaScriptu; dlatego bardzo ważne jest dobre zrozumienie tego, jak ona działa. Spróbuj poeksperymentować z przedstawionym kodem i wypróbować działanie tej funkcji na tablicach z elementami różnych typów.

```
H:\NaukaTypeScriptu\rozdzial03>tsc map --target "es6"
H:\NaukaTypeScriptu\rozdzial03>node map
[
  '<div>1 - Tomek</div>',
  '<div>2 - Celina</div>',
  '<div>3 - Robert</div>'
]
```

Rysunek 3.10. Wyniki działania funkcji map

## reduce

Funkcja `reduce` jest agregatorem, który pobiera kolejno wszystkie elementy tablicy i posługując się określoną logiką działania, przetwarza je do postaci jednej wartości. Działanie tej funkcji warto przeanalizować na przykładzie. Utwórz zatem nowy plik o nazwie *reduce.js* — także tym razem użyjemy pliku JavaScript, by nie rozpraszał nas kod generowany przez kompilator TypeScript — i zapisz w nim następujący kod:

```
const allTrucks = [
  2,5,7,10
]

const initialCapacity = 0;
const allTonnage = allTrucks.reduce((totalCapacity, currentCapacity) => {
  totalCapacity = totalCapacity + currentCapacity;

  return totalCapacity;
}, initialCapacity);

console.log(allTonnage);
```

W ramach tego przykładu wyobraźmy sobie, że chcemy obliczyć całkowity łączny ciężar ładunku, który może przewieźć cała flota ciężarówek pewnego przedsiębiorstwa transportowego. Stała `allTrucks` jest listą zawierającą ciężary ładunków, które mogą przewieźć poszczególne ciężarówki. W takim przypadku, do wyliczenia całkowitego ciężaru, jaki mogą przewieźć wszystkie ciężarówki, możemy użyć wywołania `allTrucks.reduce`. Zmienna `initialCapacity` służy wyłącznie do określenia wartości początkowej, którą w naszym przypadku jest 0. Po wywołaniu funkcji `reduce` skrypt wyświetla uzyskany wynik, który przedstawiłem na rysunku 3.11.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

H:\NaukaTypeScriptu\rozdzial03>node reduce
24
```

Rysunek 3.11. Wynik wykonania funkcji reduce



Łączny ciężar ładunku wynosi 24, gdyż tyle wynosi suma ciężaru ładunków, które mogą przewieźć poszczególne ciężarówki. Warto zwrócić uwagę na to, że logika obliczeń wykonywanych przez funkcję `reduce` może być dowolna — wcale nie musi to być sumowanie. Równie dobrze funkcja ta może odejmować wartości lub wykonywać dowolne inne operacje. Najważniejszym aspektem jej działania jest to, że na samym końcu uzyskamy wynik będący jedną wartością lub obiektem. To właśnie z tego powodu funkcja ta nosi nazwę `reduce`<sup>1</sup>.

## some oraz every

Te dwie funkcje służą do sprawdzania zadanego warunku; dlatego też zwracają wartość logiczną: `true` lub `false`. Pierwsza z tych funkcji, `some`, sprawdza, czy *którykolwiek* z elementów tablicy spełnia podany warunek; natomiast druga, `every`, sprawdza, czy warunek jest spełniony przez *wszystkie* elementy tablicy. Przyjrzyjmy się działaniu tych funkcji na przykładzie. Utwórz plik o nazwie `someEvery.js` i zapisz w nim poniższy kod:

```
const widgets = [
  { id: 1, color: 'niebieski' },
  { id: 2, color: 'żółty' },
  { id: 3, color: 'pomarańczowy' },
  { id: 4, color: 'niebieski' },
]

console.log('Czy są jakieś niebieskie elementy?', widgets.some(item => {
  return item.color === 'niebieski';
}));

console.log('Czy wszystkie elementy są niebieskie?', widgets.every(item => {
  return item.color === 'niebieski';
}));
```

Przedstawiony przykład jest bardzo prosty — jego działanie sprowadza się do sprawdzenia tego samego warunku logicznego przy użyciu funkcji `some` i `every`, oraz wyświetlenia uzyskanych wyników (przedstawiłem je na rysunku 3.12).

```
H:\NaukaTypeScriptu\rozdzial03>node someEvery
Czy są jakieś niebieskie elementy? true
Czy wszystkie elementy są niebieskie? false
```

Rysunek 3.12. Wyniki wykonania skryptu `someEvery.js`

Jak widać, w obu przypadkach uzyskane wyniki są zgodne z oczekiwaniami.

W tym podrozdziale przedstawiłem wybrane nowe funkcje dodane do wersji ES6 JavaScriptu, pozwalające na łatwiejsze i bardziej efektywne operowanie na tablicach. W dalszej części książki, kiedy już zaczniemy pisać aplikacje Reacta, bez wątpienia będziesz bardzo często używał tych funkcji. Kolejnym zagadnieniem, którym się zajmiemy, będą typy kolekcji oraz możliwości ich stosowania zamiast tablic.

<sup>1</sup> W języku angielskim *reduce* oznacza, między innymi: redukować, upraszczać — *przyp. tłum.*

# Przedstawienie nowych typów kolekcji

W wersji ES6 JavaScriptu wprowadzono dwa nowe typy kolekcji, Set i Map, które mogą się przydać w pewnych określonych sytuacjach. W tym podrozdziale opiszę te dwa typy nieco dokładniej i pokażę, jak używać ich w kodzie, żebyś później, kiedy zaczniemy już tworzyć aplikacje Reacta, mógł z nich korzystać.

## Set

Typ Set to kolekcja unikalnych wartości lub obiektów. Z powodzeniem można go używać kiedy chcemy sprawdzić, czy jakiś element znajduje się już na dużej i złożonej liście danych. Przeanalizujmy przykład zastosowania typu Set; w tym celu utwórz plik *set.js* i zapisz w nim poniższy kod:

```
const userIds = [
  1,2,1,3
]

const uniqueIds = new Set(userIds);
console.log(uniqueIds);

uniqueIds.add(10);
console.log('Po dodaniu 10', uniqueIds);

console.log('Czy zawiera', uniqueIds.has(3));

console.log('Liczba elementów', uniqueIds.size);

for (let item of uniqueIds) {
  console.log('iteracja:', item);
}
```

Obiekty Set mają wiele składowych, jednak te przedstawione w powyższym przykładzie należą do najważniejszych. Jak widać, klasa Set udostępnia konstruktor, do którego można przekazać tablicę i który przekształca tę tablicę w zbiór unikalnych wartości.

W przypadku typu Set właściwość `size` zwraca liczbę elementów zbioru, a nie długość.

Zwróć także uwagę na umieszczony na samym końcu przykładu fragment kodu, pokazujący różnice pomiędzy sposobami przeglądania zawartości tablic oraz zbiorów — w przypadku kolekcji Set jej elementy nie są pobierane przy użyciu indeksów. Wyniki wykonania powyższego przykładu przedstawiłem na rysunku 3.13.

Pod względem koncepcyjnym zbiory są całkiem podobne do tablic, zostały jednak zoptymalizowane pod kątem stosowania jako kolekcje unikalnych elementów.

```
H:\NaukaTypeScriptu\rozdzial03>node set
Set(3) { 1, 2, 3 }
Po dodaniu 10 Set(4) { 1, 2, 3, 10 }
Czy zawiera true
Liczba elementów 4
iteracja: 1
iteracja: 2
iteracja: 3
iteracja: 10
```

Rysunek 3.13. Przykłady stosowania kolekcji Set

## Map

Typ Map to kolekcja par klucz-wartość. Innymi słowy, kolekcje tego typu przypominają słowniki. Każdy element słownika dysponuje unikalnym kluczem. Zobaczmy zatem, jak tworzyć kolekcje Map i ich używać; w tym celu utwórz nowy plik, *mapCollection.js*, i zapisz w nim następujący kod:

```
const mappedEmp = new Map();
mappedEmp.set("Linda", { fullName: 'Linda Kowalsky', id: 1 });
mappedEmp.set("Kuba", { fullName: 'Kuba Tomaszewski', id: 2 });
mappedEmp.set("Pamela", { fullName: 'Pamela Kluczyk', id: 4 });

console.log(mappedEmp);
console.log('get', mappedEmp.get("Kuba"));
console.log('size', mappedEmp.size);

for(let [key, val] of mappedEmp) {
  console.log('iteracja:', key, val);
}
```

Jak widać, niektóre z tych wywołań przypominają te znane już z przykładu z kolekcją Set. Największą różnicą w stosunku do poprzedniego przykładu jest umieszczona na samym dole pętla przeglądająca zawartość kolekcji; w jej kolejnych iteracjach kolejne elementy kolekcji są pobierane w formie klucza i wartości, a następnie zapisywane w tablicy. Wyniki wykonania tego przykładu przedstawiłem na rysunku 3.14.

```
H:\NaukaTypeScriptu\rozdzial03>node mapCollection.js
Map(3) {
  'Linda' => { fullName: 'Linda Kowalsky', id: 1 },
  'Kuba' => { fullName: 'Kuba Tomaszewski', id: 2 },
  'Pamela' => { fullName: 'Pamela Kluczyk', id: 4 }
}
get { fullName: 'Kuba Tomaszewski', id: 2 }
size 3
iteracja: Linda { fullName: 'Linda Kowalsky', id: 1 }
iteracja: Kuba { fullName: 'Kuba Tomaszewski', id: 2 }
iteracja: Pamela { fullName: 'Pamela Kluczyk', id: 4 }
```

Rysunek 3.14. Wyniki wykonania skryptu mapCollection.js

Przedstawiony przykład jest bardzo prosty. Na samym początku wyświetlamy całą zawartość kolekcji. Następnie pobieramy element o kluczu "Kuba", używając do tego celu funkcji `get`. Następnie wyświetlamy liczbę elementów kolekcji używając właściwości `size` i w końcu pobieramy jej kolejne elementy w pętli.

W tym podrozdziale przedstawiłem dwa nowe typy kolekcji wprowadzone w wersji ES6 JavaScriptu. Choć nie są one stosowane aż tak powszechnie, jednak mogą się przydać, zwłaszcza w sytuacjach, w których zastosowanie kolekcji danego typu będzie odpowiednie. W następnym podrozdziale zajmiemy się słowami kluczowymi `async` i `await`, czyli nowymi możliwościami dodanymi do języka JavaScript w wersji ES7. Oba te słowa kluczowe błyskawicznie zyskały akceptację i popularność w społeczności programistów JavaScriptu, gdyż w ogromnym stopniu poprawiają czytelność trudnego kodu asynchronicznego i upodabniają go do zwyczajnego kodu synchronicznego.

## Przedstawienie słów kluczowych `async` i `await`

Zanim zajmiemy się słowami kluczowymi `async` i `await`, w pierwszej kolejności muszę wyjaśnić, czym w ogóle jest programowanie asynchroniczne. W większości języków programowania kod jest zazwyczaj wykonywany w sposób synchroniczny, czyli kolejno, instrukcja po instrukcji. Jeśli w takim kodzie znajdują się trzy instrukcje, A, B i C, to nie będzie można rozpocząć wykonywania instrukcji B, zanim nie zostanie zakończone wykonywanie instrukcji A; podobnie nie będzie można rozpocząć wykonywania instrukcji C, zanim nie zostanie wykonana instrukcja B. Z kolei w przypadku programowania asynchronicznego, jeśli A jest instrukcją asynchroniczną, to zostanie ona rozpoczęta, po czym natychmiast zwróci sterowanie i będzie można rozpocząć wykonywanie instrukcji B. Innymi słowy, instrukcja B nigdy nie będzie czekać na zakończenie wykonywania instrukcji A. Choć takie rozwiązanie jest wspaniałe pod względem wydajności działania, to jednak znacząco utrudnia pisanie kodu. I właśnie te problemy starają się upraszczać słowa kluczowe `async` i `await`.

Programowanie asynchroniczne pozwala poprawić wydajność wykonywania kodu, gdyż instrukcje mogą być wykonywane współbieżnie i nie muszą czekać na zakończenie wykonywania poprzedniej. Jednak, aby móc zrozumieć zasady programowania asynchronicznego, musimy w pierwszej kolejności zrozumieć, czym są wywołania (czy też funkcje) zwrotne i jak można ich używać. Już od momentu udostępnienia środowiska Node.js, funkcje zwrotne stanowiły jeden z kluczowych stosowanych w nim mechanizmów programowania. Dlatego też niezwykle ważne jest, by dobrze je zrozumieć. Przyjrzyjmy się zatem przykładowi stosowania funkcji zwrotnych. Utwórz nowy plik, *callback.js*, a następnie zapisz w nim następujący kod:

```
function letMeKnowWhenComplete(size, callback) {
  var reducer = 0;
  for (var i = 1; i < size; i++) {
    reducer = Math.sin(reducer * i);
  }
}
```

```

    callback();
  }
  letMeKnowWhenComplete(100000000, function () { console.log("Super!
  Skończyłem."); });

```

Kiedy przyjrzymy się powyższemu przykładowi, zauważymy, że funkcja `letMeKnowWhenComplete` przyjmuje dwa parametry. Pierwszy z nich określa liczbę iteracji pewnych obliczeń matematycznych, natomiast drugi jest funkcją zwrrotną. Jak widać, funkcja zwrrotna `callback` jest wywoływana po zakończeniu obliczeń. Gwoli ścisłości, z technicznego punktu widzenia, funkcja zwrrotna wcale nie jest funkcją asynchroniczną. Jednak możliwości, jakie zapewnia, są właściwie takie same, gdyż ta dodatkowa operacja — wywołanie zwrtnie — jest wykonywana dokładnie w momencie zakończenia wykonywania zadania głównego, bez konieczności dodatkowego oczekiwania lub wybierania kodu, który zostanie wykonany jako następny. Przyjrzyjmy się zatem pierwszym dostępnym w JavaScriptcie rozwiązaniom, w których były wykorzystywane funkcje zwrtnie wykonywane po zakończeniu operacji.

Pierwsze możliwości asynchronicznego wykonywania kodu, które były dostępne w języku JavaScript, zapewniały funkcje `setTimeout` i `setInterval`. Są to całkiem proste funkcje; każda z nich pobiera funkcję zwrtną, która zostaje wykonana po upływie zadanego czasu. W przypadku funkcji `setInterval` jedyna różnica względem tego sposobu działania polega na tym, że funkcja zwrtna jest wykonywana cyklicznie. Przyczyną, która sprawia, że te funkcje są prawdziwie asynchroniczne jest to, że funkcja zwrtna jest wykonywana poza bieżącym **stosem wywołań**. Funkcja zwrtna jest jedynie sekwencją kodu (wraz z towarzyszącymi mu danymi), który jest wykonywany w bieżącym wątku. W przypadku języka JavaScript kod jest wykonywany w jednym wątku, dlatego też funkcje zwrtnie liczników czasu są wykonywane przez silnik JavaScript przeglądarki na rzecz głównego wątku, do którego następnie są przekazywane wyniki. Przyjrzyjmy się wykorzystaniu tych funkcji na przykładzie. Utwórz plik `setTimer.js` i zapisz w nim następujący kod:

```

// 1
console.log("Zaczynamy...");

// 2
setTimeout(() => {
  console.log("Zaczekałem, ale teraz już kończę.");
}, 3000);

// 3
console.log("Czy to już koniec?");

```

Przeanalizujmy teraz działanie tego kodu. Dodałem do niego komentarze, aby wyróżnić jego główne części. Pierwsza z nich, umieszczona pod komentarzem z cyfrą 1, wyświetla komunikat o rozpoczęciu działania skryptu. W drugiej części, umieszczonej pod komentarzem z liczbą 2, wywołujemy funkcję `setTimeout`, która po upływie trzech sekund wywoła przekazaną funkcję strzałkową. Funkcja strzałkowa także powoduje wyświetlenie na ekranie komunikatu. Poniżej funkcji `setTimeout`, w dolnej części skryptu umieszczonej pod komentarzem z liczbą 3, wyświetlamy kolejny komunikat z pytaniem o to, czy działanie skryptu już się zakończyło. Kiedy wykonasz ten kod, stanie się coś dziwnego, co widać w jego wynikach, które przedstawiłem na rysunku 3.15.

```
H:\NaukaTypeScriptu\rozdzial03>node setTimer
Zaczynamy...
Czy to już koniec?
Zaczekałem, ale teraz już kończę.
```

Rysunek 3.15. Wyniki wykonania skryptu `setTimer.js`

Ostatni komunikat umieszczony w kodzie, pytanie *Czy to już koniec?* zostanie wyświetlony jako drugi, a dopiero po nim pojawi się komunikat *Zaczekałem, ale teraz już kończę*. Dlaczego tak się dzieje? Otóż `setTimeout` jest funkcją asynchroniczną, dlatego bezpośrednio po jej wywołaniu zacznie być wykonywany kod zapisany poniżej jej wywołania (niezależnie od tego, że wykonywanie tej funkcji jeszcze się nie zakończyło). W naszym przypadku oznacza to, że funkcja `log` umieszczona w trzeciej części kodu zostanie wykonana przed wywołaniem funkcji zwrótniej podanej w drugiej części kodu. A zatem, jeśli wyobrazimy sobie, że w trzeciej części skryptu znajduje się jakiś ważny kod, który musi być wykonany od razu, bez oczekiwania na kod z drugiej części skryptu, to przekonamy się, jak użyteczne pod względem wydajności działania mogą być funkcje asynchroniczne. A teraz połączmy zdobytą wiedzę o funkcjach zwrótnych i wywołaniach asynchronicznych, i przyjrzyjmy się obietnicom.

Przed wprowadzeniem słów kluczowych `async` i `await`, techniki programowania asynchronicznego w języku JavaScript bazowały na stosowaniu tak zwanych obietnic (ang. *Promises*). Obietnica to obiekt `Promise`, którego wyznaczenie zostało odroczone do jakiegoś bliżej nieokreślonego momentu w przyszłości. Poniżej przedstawiłem prosty przykład zastosowania obietnic — utwórz nowy plik, *promise.js*, i zapisz w nim poniższy kod.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    // resolve("Operacja wykonana pomyślnie.");
    reject("Niepowodzenie!");
  }, 500);
});

myPromise
  .then(done => {
    console.log(done);
  })
  .catch(err => {
    console.log(err);
  });
```

W tym kodzie tworzymy obietnicę, obiekt `Promise`, a wewnątrz niej wykonujemy asynchroniczny licznik czasu, który wykona określoną instrukcję po upływie 500 milisekund. W ramach pierwszej próby wykonania tego kodu celowo doprowadzamy do niepowodzenia wykonania kodu asynchronicznego, wywołując w tym celu funkcję `reject`; w efekcie sterowanie zostanie przekazane do funkcji strzałkowej określonej w wywołaniu funkcji `catch` obiektu `Promise`. Jeśli teraz umieścimy wywołanie funkcji `reject` w komentarzu, a usuniemy znaki komentarza zapisane przed wywołaniem funkcji `resolve`, to zostanie wykonana funkcja strzałkowa przekazana w wywołaniu funkcji `then` obiektu `Promise`. Oczywiście, taki kod działa, jednak łatwo można sobie wyobrazić, że w razie zastosowania bardziej

złożonych obiektów Promise, albo nawet ich większej liczby, taki kod błyskawicznie stanie się bardzo trudny do przeanalizowania i zrozumienia.

I właśnie te problemy z pisanem kodu asynchronicznego mają rozwiązywać słowa kluczowe `async` i `await` — mają one pozwalać na uporządkowanie kodu, uproszczenie go, skrócenie, a nawet mają sprawiać, że będzie można pisać kod asynchroniczny *wyglądający* jak zwyczajny kod, wykonywany synchronicznie. Przeanalizujmy przykład wykorzystania tych dwóch nowych słów kluczowych. Utwórz plik o nazwie *async.js* i zapisz w nim następujący kod:

```
async function delayedResult() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Operacja wykonana pomyślnie.");
    }, 500);
  });
}

(async function execAsyncFunc() {
  const result = await delayedResult();
  console.log(result);
})();
```

Ten kod rozpoczyna się od funkcji o nazwie `delayedResult`, która, jak widać, jest poprzedzona słowem kluczowym `async`. Poprzedzenie funkcji słowem kluczowym `async` informuje środowisko wykonawcze JavaScriptu, że funkcja ta zwraca obietnicę, a co za tym idzie, że można ją wykonywać asynchronicznie. Z kolei w dolnej części skryptu znajduje się funkcja o nazwie `execAsyncFunc`, która także została zadeklarowana jako asynchroniczna, a dodatkowo jest od razu wywoływana. Jeśli jeszcze nie spotkałeś się z takim rozwiązaniem, to jest ono określane jako **bezwłocznie wywoływane wyrażenie funkcyjne** (ang. *Immediately Invoked Function Expression*, w skrócie *IIFE*). Więcej informacji na temat takich wyrażeń znajdziesz w dalszej części książki; póki co skoncentrujemy się na dalszej analizie przykładu. Funkcja `execAsyncFunc` także jest funkcją asynchroniczną, a jak widać na przykładzie, w jej kodzie używamy słowa kluczowego `await`. Informuje ono środowisko wykonawcze, że mamy zamiar wywołać funkcję asynchroniczną, dlatego należy wstrzymać dalsze wykonywanie kodu, zaczekać, aż realizacja funkcji asynchronicznej zostanie zakończona i pobrać zwrócony przez nią wynik. Kiedy wykonasz ten przykład, wyświetli on wyniki przedstawione na rysunku 3.16.

```
H:\NaukaTypeScriptu\rozdzial03>node async
Operacja wykonana pomyślnie.
```

Rysunek 3.16. Wyniki wykonania skryptu *async.js*

Jak widać, w zmiennej `result` został zapisany łańcuch "Operacja wykonana pomyślnie.", a nie obiekt Promise, który normalnie zwraca funkcja `delayedResult`. Ta składnia wywoływania kodu asynchronicznego jest znacznie krótsza i łatwiejsza do zrozumienia niż kod korzystający z obiektów Promise oraz funkcji `then` i `catch`. Pamiętaj, że stosownie słów kluczowych `async` i `await` stało się standardowym sposobem pisania kodu asynchronicznego w JavaScriptcie, powszechnie wykorzystywanym w środowisku programistów posługujących się tym językiem. Aby lepiej zrozumieć ten sposób programowania, przedstawię jeszcze jeden przykład.

Zastosowanie bezzwłocznie wykonywanego wyrażenia funkcyjnego w powyższym przykładzie było konieczne, ze względu na aktualny sposób działania języka JavaScript, który nie zezwala na stosowanie słowa kluczowego `await` na głównym poziomie kodu. Oznacza to, że aktualnie nie jest dozwolone wywoływanie funkcji asynchronicznych z użyciem słowa `await` w kodzie, który nie jest umieszczony w innej funkcji asynchronicznej (zadeklarowanej z użyciem słowa kluczowego `async`). Ograniczenie to ma zostać wyeliminowane w wersji ECMAScript 2020 JavaScriptu, jednak w czasie przygotowywania niniejszej książki jeszcze nie wszystkie przeglądarki dysponowały tą możliwością.

Ze względu na wielkie znaczenie słów kluczowych `async` i `await`, przedstawię jeszcze jeden przykład ich zastosowania. Tym razem spróbujmy pobrać jakieś zasoby z internetu. W skrypcie wykorzystamy API `fetch`, które nie jest domyślnie dostępne w środowisku Node. Oznacza to, że przed wykonaniem przykładu będziesz musiał pobrać jeden dodatkowy pakiet `npm`. Oto czynności, jakie musisz wykonać:

1. Zainstaluj pakiet `fetch` przy użyciu następującego polecenia:

```
npm i node-fetch
```

2. Utwórz plik o nazwie `fetch.js` i zapisz w nim poniższy kod:


```
const fetch = require('node-fetch');

(async function getData() {
  const response = await
fetch("https://pokeapi.co/api/v2/pokemon/ditto/");
  if(response.ok) {
    const result = await response.json();
    console.log(result);
  } else {
    console.log("Nie udało się nic zrobić!");
  }
})();
```

Zwróć uwagę na łatwość analizy kodu tego przykładu oraz na naturalny przepływ sterowania. Jak już wspominałem, w przykładzie używamy API `fetch`, które służy do asynchronicznego pobierania zasobów z internetu. Po zaimportowaniu pakietu `fetch`, tworzymy funkcję asynchroniczną, która będzie spełniać rolę opakowania i pozwalać na asynchroniczne wywoływanie funkcji pakietu `fetch` przy użyciu słowa kluczowego `await`. Gdybyś się zastanawiał, co to za adres URL został użyty w przykładzie, to wyjaśniam, że jest to ogólnodostępne API zwracające informacje o pokemonach, którego stosowanie nie wymaga żadnego uwierzytelniania. Pierwsze wywołanie z użyciem słowa kluczowego `await` zwraca samo wywołanie sieciowe. Kiedy zostanie ono zakończone, sprawdzamy, czy udało się je wykonać prawidłowo, używając do tego celu wyrażenia `response.ok`. Jeśli okaże się, że wszystko jest w porządku, to używamy `await` do wykonania kolejnego wywołania, które zwróci właściwe dane o pokemonie zapisane w formacie JSON. Każde wywołanie wykonywane z użyciem słowa kluczowego `await` wstrzymuje wykonywanie kodu aż do momentu zakończenia funkcji asynchronicznej i pobrania zwróconego przez nią wyniku.



Oczekujemy, gdyż nasz kod nie może kontynuować działania bez danych pobieranych przy użyciu API, więc nie mamy innego wyboru, jak zaczekać na zakończenie operacji. Wyniki wykonania tego kodu będą podobne do tych przedstawionych na rysunku 3.17.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS
name: 'ditto',
url: 'https://pokeapi.co/api/v2/pokemon-species/132/'
},
sprites: {
  back_default: 'https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/back/132.png',
  back_female: null,
  back_shiny: 'https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/back/shiny/132.png',
  back_shiny_female: null,
  front_default: 'https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/132.png',
  front_female: null,
  front_shiny: 'https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/shiny/132.png',
  front_shiny_female: null,
  other: { dream_world: [Object], 'official-artwork': [Object] },
  versions: {
    'generation-i': [Object],
    'generation-ii': [Object],
    'generation-iii': [Object],
    'generation-iv': [Object],
    'generation-v': [Object],
    'generation-vi': [Object],
    'generation-vii': [Object],
    'generation-viii': [Object]
  }
},
stats: [
  { base_stat: 48, effort: 1, stat: [Object] },
  { base_stat: 48, effort: 0, stat: [Object] },
  { base_stat: 48, effort: 0, stat: [Object] },
  { base_stat: 48, effort: 0, stat: [Object] },
  { base_stat: 48, effort: 0, stat: [Object] },
  { base_stat: 48, effort: 0, stat: [Object] }
],
types: [ { slot: 1, type: [Object] } ],
weight: 40
}
H:\NaukaTypeScriptu\rozdzial03>

```

Rysunek 3.17. Wyniki wykonania skryptu fetch.js

Kiedy uruchomisz ten przykład, najprawdopodobniej zauważysz niewielkie opóźnienie, z jakim zostaną wyświetlone wyniki. Pokazuje ono, że przed wyświetleniem danych kod musiał poczekać na wykonanie żądania sieciowego.

W tym podrozdziale poznałeś podstawy programowania asynchronicznego. Przedstawiłem w nim zarówno obietnice (obiekty Promise) — podstawę programowania asynchronicznego w języku JavaScript — jak i słowa kluczowe `async` i `await`, które w ogromnym stopniu upraszczają pisanie kodu asynchronicznego. Oba te słowa kluczowe są powszechnie używane podczas tworzenia aplikacji Reacta, jak również wszelkich innych aplikacji działających w środowisku Node.

## Podsumowanie

W tym rozdziale przedstawiłem wiele najnowszych możliwości języka JavaScript, takich jak sposoby skalania obiektów i tablic przy użyciu operatora **rozproszenia**, nowe i ulepszone sposoby operowania na tablicach, no i oczywiście słowa kluczowe `async` i `await` — stanowiące nowy i niezwykle popularny sposób tworzenia kodu asynchronicznego. Dobrze

zrozumienie tych wszystkich możliwości jest bardzo ważne, gdyż są one powszechnie stosowanymi elementami nowoczesnych aplikacji Reacta oraz innych programów pisanych w języku JavaScript.

W następnym rozdziale zajmiemy się zagadnieniami tworzenia aplikacji jednostronicowych przy użyciu frameworka React, przy czym już od samego początku będziemy korzystać z wielu możliwości przedstawionych w niniejszym rozdziale.



C Z Ę Ś Ć

# Nauka tworzenia aplikacji jednostronicowych z użyciem frameworka React

Z tej części książki dowiesz się, jak konfigurować oraz pisać aplikacje internetowe korzystające z frameworka React.

Składa się ona z następujących rozdziałów:

- Rozdział 4. — „Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React”;
- Rozdział 5. — „Tworzenie aplikacji React z wykorzystaniem hooków”;
- Rozdział 6. — „Przygotowywanie projektu za pomocą create-react-app i testowanie go przy użyciu Jest”;
- Rozdział 7. — „Redux i React Router”.



# Przedstawienie konceptji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React

W tym rozdziale będziemy się zajmować aplikacjami jednostronicowymi (ang. *Single-Page Application*, w skrócie **SPA**). W historii tworzenia aplikacji internetowych ten styl jest stosunkowo nowy, jednak w ostatnich latach zyskał sobie bardzo dużą popularność. Aktualnie stanowi on powszechnie stosowany sposób tworzenia dużych i skomplikowanych aplikacji internetowych, które swoim działaniem przypominają klasyczne aplikacje komputerowe, bądź też aplikacje na urządzenia przenośne.

W tym rozdziale opiszę pokrótce wcześniejsze sposoby tworzenia aplikacji internetowych i wyjaśnię, dlaczego opracowano aplikacje jednostronicowe. Następnie pokażę, w jaki sposób React ułatwia nam wydajne i efektywne stosowanie tego stylu programowania.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- przedstawieniem wcześniejszych sposobów tworzenia witryny WWW;
- wyjaśnieniem cech oraz zalet aplikacji jednostronicowych;
- wyjaśnieniem, jak React pomaga w tworzeniu aplikacji jednostronicowych.

## Wymagania techniczne

Wymagania techniczne, które musisz spełnić przed przystąpieniem do lektury tego rozdziału, są takie same jak w przypadku rozdziału 3., pt. „Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript”. Powinieneś dysponować podstawową znajomością języka JavaScript oraz technologii internetowych. Podobnie jak w poprzednim rozdziale, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code** (VSCode).

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, anglojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial04* (*Chap4*).

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog o nazwie *rozdzial04*.

## Przedstawienie wcześniejszych sposobów tworzenia witryny WWW

W tym podrozdziale spróbuję wyjaśnić przyczyny, które doprowadziły do powstania nowego stylu programowania, jakim jest tworzenie aplikacji jednostronicowych. W tym celu pokażę, w jaki sposób wcześniej projektowano i pisano strony WWW. Dzięki tym informacjom łatwiej Ci będzie zrozumieć, dlaczego opracowano aplikacje jednostronicowe.

W początkowym okresie istnienia WWW nie było jeszcze języka JavaScript. W tamtych czasach tworzone jedynie statyczne strony HTML, które były głównie używane przez naukowców do dzielenia się informacjami. Kiedy jednak format HTML oraz sam internet zyskały popularność, zrozumiano, że poprawa komunikacji, jaką zapewniają strony WWW, wymaga poprawienia metod określania ich wyglądu. Właśnie w tym celu powstały kaskadowe arkusze stylów, CSS, służące do określania stylu i układu dokumentów HTML. Nieco później firma Netscape uznała, że WWW potrzebuje języka skryptowego, który byłby w stanie zapewnić stronom WWW nieco dynamizmu... i tak powstał język JavaScript.

Bez względu na te nowe możliwości, charakter stron WWW w tamtych czasach wciąż pozostawał bardzo statyczny. Kiedy wpisywaliśmy w przeglądarce adres URL, zwracany był pojedynczy dokument — czyli wybrany plik przechowywany na serwerze — i to dotyczyło wszystkich używanych adresów URL. Kaskadowe arkusze stylów oraz JavaScript sprawiły, że strony WWW wyglądały lepiej i były nieco bardziej dynamiczne, jednak nie udało im się zmienić modelu tworzenia witryn WWW, który koncentrował się na tworzeniu odrębnych plików — stron HTML.

Jednak wraz z upływem czasu, kiedy witryny WWW zaczęły być coraz to bardziej złożone i wyrafinowane, wielu twórców stron WWW uznało, że chcieliby mieć większą kontrolę nad tworzonymi dokumentami. Chcieli dysponować możliwością dynamicznego kontrolowania zarówno układu, jak i zawartości stron. W efekcie opracowano technologię o nazwie **Common Gateway Interface (CGI)**. CGI była początkową wersją grupy rozwiązań określanych ogólnie jako **Server-Side Rendering (SSR)**. Chodziło w nich o to, by żądanie przesyłane przez przeglądarkę było odbierane przez serwer, który zamiast zwracać określoną, statyczną stronę HTML, uruchomi jakiś procesor, który w dynamiczny sposób wygeneruje stronę na podstawie pewnej logiki i przekazanych parametrów, a następnie odeśle ją do przeglądarki.

W tamtym czasie, niezależnie od tego, czy witryny składały się ze statycznych stron HTML, czy też były generowane dynamicznie po stronie serwera, stosowane rozwiązania bazowały na przekazywaniu z serwera do przeglądarki kompletnych dokumentów HTML w formie plików. Właśnie taki był ogólny sposób działania witryn WWW.

Ten model pojedynczych plików, czy też stron, nie przypomina sposobu działania rodzimych aplikacji, czy to na komputerach stacjonarnych, czy też na urządzeniach przenośnych. Model działania takich aplikacji różni się od stron WWW tym, że aplikacja jest pobierana w całości i instalowana na urządzeniu użytkownika. Później, kiedy użytkownik uruchomi aplikację, ta będzie od razu gotowa do działania, zapewniając mu dostęp do wszystkich swoich możliwości. Wszystkie elementy sterujące, które mają zostać wyświetlone na ekranie, zostaną wyświetlone przez kod, który działa na urządzeniu i nie trzeba przy tym wykonywać żadnych dodatkowych odwołań do serwera (z wyjątkiem żądań związanych z pobieraniem i zapisem danych). Taki model działania sprawia, że aplikacja działa znacznie płynniej i szybciej niż klasyczne aplikacje internetowe tworzone w klasycznym modelu, które cały czas wymagały odwołań do serwera i odświeżania całych stron.

Zadaniem aplikacji jednostronicowych było sprawienie, że aplikacje internetowe będą w większym stopniu przypominały aplikacje rodzime, zwłaszcza pod względem szybkości oraz płynności działania. Aby to uzyskać, aplikacje jednostronicowe korzystają z różnych technik i bibliotek, które zapewniają, że aplikacje te zarówno swoim wyglądem, jak i działaniem, będą bardziej zbliżone do aplikacji rodzimych.

W tym podrozdziale wyjaśniłem, w jaki sposób tworzone witryny WWW w początkowym okresie istnienia internetu. W tamtych czasach koncentrowano się głównie na tworzeniu i udostępnianiu pojedynczych plików — dokumentów HTML. Wyjaśniłem także, na czym polegały ograniczenia tego stylu programowania, zwłaszcza w porównaniu z aplikacjami rodzimymi, i przedstawiłem aplikacje jednostronicowe, opracowane jako próbę wyeliminowania ograniczeń klasycznego modelu witryn WWW i upodobnienia aplikacji internetowych

do rodzimych. W następnym podrozdziale opiszę dokładniej, czym są aplikacje jednostronicowe i pod jakimi względami są one lepsze od początkowego modelu tworzenia aplikacji internetowych koncentrującego się na odrębnych dokumentach HTML.

## Cechy i zalety aplikacji jednostronicowych

W tym podrozdziale przedstawię cechy oraz zalety aplikacji jednostronicowych. Dzięki ich znajomości będziesz mógł lepiej zrozumieć decyzje architektoniczne podjęte podczas tworzenia Reacta i niektórych spośród powiązanych z nim bibliotek oraz komponentów używanych w aplikacjach Reacta.

Jak już wspominałem, bodźcem do tworzenia aplikacji jednostronicowych jest chęć upodobnienia rozwiązań internetowych do aplikacji rodzimych, działających na komputerach stacjonarnych i urządzeniach przenośnych. Korzystając z rozwiązań stosowanych w aplikacjach jednostronicowych sprawimy, że nasze programy będą działały płynnie, reagowały szybko i wyglądały tak, jakby zostały zainstalowane na urządzeniu, na którym są używane. Aplikacje internetowe tworzone w modelu klasycznym mogą przy nich wydawać się powolne i niewygodne, gdyż każda zmiana wymaga ponownego odwołania się do serwera i pobrania nowej strony. W odróżnieniu od takiego modelu działania, aplikacje jednostronicowe odświeżają jedynie fragment ekranu i robią to błyskawicznie, bez oczekiwania na pobranie nowego pliku z serwera. Dlatego, z punktu widzenia użytkownika, aplikacje jednostronicowe działają tak samo jak rodzime aplikacje komputerowe.

Tworzenie jednostronicowych aplikacji internetowych jest dość złożone i wymaga stosowania wielu komponentów i bibliotek. Aplikacje jednostronicowe mają kilka wspólnych cech oraz wymagań, które przed nami stawiają, i to niezależnie od tego, czy używamy frameworka Angular, Vue, React, czy też jeszcze innego.

Poniżej wymieniłem kilka spośród wymagań związanych z tworzeniem aplikacji jednostronicowych:

- Zgodnie z tym, co sugeruje nazwa, cała aplikacja istnieje w ramach jednej strony WWW. W odróżnieniu od standardowych aplikacji HTML, które są tworzone z niezależnych plików prezentujących różne ekrany aplikacji, w aplikacjach jednostronicowych pierwsza strona jest jednocześnie jedyną, która jest pobierana z serwera.
- Zamiast pobierać i wyświetlać pliki HTML, kod JavaScript dynamicznie generuje poszczególne ekrany aplikacji. Z tego względu, początkowo pobierana strona HTML jest niemal całkowicie pozbawiona treści. Jednak będzie ona zawierała element główny, umieszczony wewnątrz znacznika body. Ten element główny staje się kontenerem dla całej aplikacji, która jest generowana i wyświetlana na bieżąco w odpowiedzi na czynności wykonywane przez użytkownika.
- Wszystkie skrypty i pliki niezbędne do działania aplikacji są zazwyczaj pobierane na samym początku, w tym samym czasie, kiedy pobierany jest plik HTML. Niemniej jednak rozwiązanie to powoli się zmienia i coraz więcej aplikacji pobiera początkowo jedynie najważniejszy, główny skrypt, a dopiero



później, w razie potrzeby, kolejne. Technikę tę opisałem bardziej szczegółowo w dalszej części książki; jest ona użyteczna, gdyż pozwala poprawić wrażenia użytkownika, skracając czas oczekiwania na uruchomienie aplikacji.

- W klasycznych aplikacjach internetowych oraz w aplikacjach jednostronicowych zupełnie inaczej korzysta się z adresów URL. W przypadku aplikacji jednostronicowych używany jest pewien mechanizm, zależny od stosowanego frameworka, pozwalający na stworzenie **trasowania wirtualnego** (ang. *virtual routing*). Trasowanie wirtualne oznacza, że choć z punktu widzenia użytkownika wygląda na to, że aplikacja odwołuje się do różnych adresów URL na serwerze, to jednak cała obsługa tych adresów jest wykonywana w przeglądarce i zapewnia możliwość logicznego przechodzenia pomiędzy poszczególnymi ekranami aplikacji. Innymi słowy, żadne odwołania do serwera nie są wykonywane, a trasowanie wirtualne jest sposobem logicznego podziału aplikacji na poszczególne ekrany. Na przykład, kiedy użytkownik wpisze w przeglądarce adres URL, musi nacisnąć klawisz *Enter*, by przeglądarka przesłała żądanie na docelowy serwer WWW. Z kolei w przypadku trasowania stosowanego w aplikacjach jednostronicowych, wpisywane adresy URL nie odwołują się do żadnej ścieżki faktycznie istniejącej na serwerze. Ścieżki odpowiadające tym adresom po prostu nie istnieją. Dlatego też przeglądarka nigdy nie generuje żądań. Zamiast tego, aplikacje jednostronicowe używają adresów URL jako swoiste kontenery określające sekcje aplikacji, jak również mogą wykonywać określone zachowania w odpowiedzi na użycie określonych adresów. Pomimo tego, trasowanie adresów URL jest bardzo użyteczną możliwością, gdyż większość użytkowników oczekuje, że aplikacja będzie reagować na zmiany adresów URL; poza tym adresy URL pozwalają użytkownikom w późniejszym czasie wracać do wybranych ekranów aplikacji.

W tym podrozdziale przedstawiłem cechy wyróżniające aplikacje jednostronicowe. Wyjaśniłem różne sposoby radzenia sobie z faktem, że aplikacje takie składają się tylko z jednego pliku HTML, oraz metodologie stosowane podczas tworzenia aplikacji tego typu. W następnym podrozdziale dokładniej wyjaśnię, w jaki sposób React umożliwia tworzenie aplikacji jednostronicowych; przedstawię ponadto decyzje podjęte przez twórców Reacta, związane z wybraniem tego stylu aplikacji internetowych.

## Jak React pomaga w tworzeniu aplikacji jednostronicowych

W tym podrozdziale przyjrzymy się frameworkowi React na dość wysokim poziomie abstrakcji. Wiedza, którą tu zdobędziesz, pozwoli Ci tworzyć lepsze aplikacje korzystające z Reacta, gdyż będziesz lepiej znał jego wewnętrzne sposoby działania.

Jak już wspominałem, witryna WWW jest w głównej mierze dokumentem HTML, który jest zwyczajnym plikiem tekstowym. Ten plik zawiera kod, którego przeglądarka używa do utworzenia logicznego drzewa nazywanego **obiektywnym modelem dokumentu** (ang. *Document Object Model*, w skrócie **DOM**). To drzewo reprezentuje wszystkie elementy HTML

umieszczone w pliku, rozmieszczone w odpowiedniej kolejności i w odpowiedni sposób względem innych elementów należących do struktury. Takie drzewo DOM jest tworzone dla wszystkich stron należących do witryny, niezależnie od tego, czy jest to aplikacja jednostronicowa, czy nie. Jednak React, starając się ułatwić nam tworzenie aplikacji, korzysta z modelu DOM w unikalny sposób.

React zawiera dwie kluczowe konstrukcje:

- W trakcie działania aplikacji React tworzy i przechowuje swój własny, wirtualny DOM. Ten wirtualny DOM jest niezależny od modelu DOM używanego przez przeglądarkę WWW. Stanowi on unikalną kopię modelu DOM przeglądarki, którą React tworzy w oparciu o instrukcje zapisane w kodzie aplikacji i cały czas przechowuje. Ten wirtualny DOM jest tworzony i modyfikowany zgodnie z potrzebami procesu uzgadniania, wykonywanego wewnętrznie przez framework React. Proces uzgadniania to proces, w ramach którego React przegląda DOM strony i porównuje go ze swoim własnym, wirtualnym modelem DOM. Proces ten jest powszechnie określany jako **faza renderowania** (ang. *rendering phase*). Kiedy zostaną odnalezione jakieś różnice, na przykład: kiedy wirtualny DOM będzie zawierał jakiś element, którego nie ma w modelu DOM przeglądarki, React przekaze do modelu DOM przeglądarki instrukcje, które doprowadzą do ujednolicenia obu modeli. Ten proces dodawania, modyfikowania lub usuwania elementów jest określany jako **faza zatwierdzania** (ang. *commit phase*).
- Drugą z podstawowych cech pisania aplikacji Reacta jest to, że ich działanie bazuje na stanie. Aplikacje Reacta składają się z bardzo wielu komponentów, a każdy z nich może dysponować własnym, lokalnym stanem (czyli danymi). Jeśli z jakiegokolwiek powodu dane te ulegną zmianie, React uruchomi proces uzgadniania i w razie konieczności wprowadzi niezbędne zmiany w modelu DOM.

Aby lepiej wyjaśnić te pojęcia, powinienem przedstawić przykład prostej aplikacji Reacta. Jednak zanim to zrobię, musisz jeszcze dowiedzieć się, z czego takie aplikacje się składają.

## Atrybuty aplikacji Reacta

Do zapewnienia prawidłowego działania nowoczesnych aplikacji tworzonych w oparciu o framework React będziemy potrzebowali kilku elementów. W pierwszej kolejności będzie to program narzędziowy npm, który umożliwi nam zarządzanie wszystkimi zależnościami aplikacji. Jak miałeś okazję przekonać się na wcześniejszych przykładach, npm jest repozytorium pozwalającym na pobieranie z centralnego repozytorium zależności (udostępnianych jako oprogramowanie typu *open source*) i używanie ich w tworzonej aplikacji. Oprócz tego będziemy potrzebowali narzędzia służącego do tak zwanego pakowania (ang. *bundling*). System pakowania to usługa, która łączy wszystkie pliki z kodami źródłowymi skryptów i zasoby takie jak pliki CSS i manifesty, w jeden zestaw plików. Proces minimalizacji usuwa z tych plików niepotrzebne odstępki, a z kodu skryptów niepotrzebne teksty, dzięki czemu pliki, które w końcu będą pobierane przez przeglądarkę, będą tak małe, jak to tylko możliwe. Ta zmniejszona wielkość pozwoli skrócić początkowy czas ładowania aplikacji i poprawić wrażenia użytkowników. System pakowania, którego będziemy używali, nosi nazwę webpack, a wybrałem go dlatego, że stanowi powszechnie stosowany standard do pakowania

aplikacji Reacta. Oprócz tego będziemy używali systemu skryptów wbudowanego w npm, z którego skorzystamy do automatyzacji niektórych z wykonywanych operacji. Na przykład przygotujemy skrypty, które będą uruchamiać serwer testowy, wykonywać testy, czy też generować końcową, produkcyjną wersję aplikacji.

Jeśli skorzystamy z pakietu npm o nazwie `create-react-app`, uzyskamy dostęp do wszystkich niezbędnych zależności, o których wspominałem wcześniej, jak również innych zależności powszechnie stosowanych podczas tworzenia aplikacji Reacta, a także do kilku skryptów do zarządzania aplikacją. A zatem, użyj tego pakietu, by utworzyć swoją pierwszą aplikację Reacta:

1. W terminalu lub oknie wiersza poleceń przejdź do katalogu *NaukaTypeScriptu/rozdzial04* i wykonaj następujące polecenie:

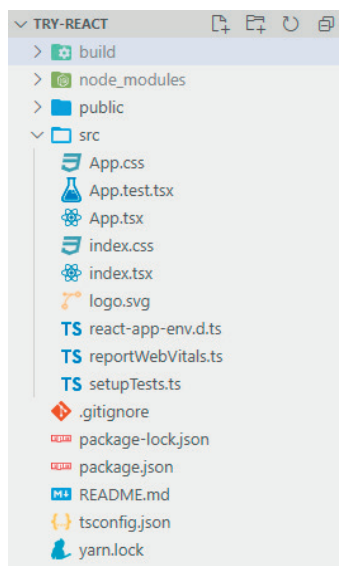
```
npx create-react-app try-react --template typescript
```

Zastosowałem polecenie `npx` zamiast `npm` i `-g`, dzięki czemu nie będziesz musiał instalować pakietu `create-react-app` lokalnie.

2. Kiedy polecenie z poprzedniego punktu zostanie wykonane, uruchom edytor VSCode i otwórz powstały przed chwilą katalog, *try-react*, który został utworzony na nową aplikację Reacta.
3. W edytorze VSCode wyświetl panel terminala i wykonaj w nim następujące polecenie:

```
npm run build
```

To polecenie przygotuje produkcyjną wersję aplikacji i zapisze ją w katalogu *build*. Kiedy operacja zostanie zakończona, w panelu VSCode będziesz mógł zobaczyć strukturę plików taką jak ta z rysunku 4.1.



Rysunek 4.1. Zawartość katalogu *try-react*

Przyjrzyjmy się teraz zasobom, które przygotowuje dla nas skrypt `create-react-app`, zaczynając do samego początku listy widocznej na rysunku 4.1:

- Katalog *build* jest miejscem docelowym, w którym będą zapisywane wszystkie spakowane i zminimalizowane, produkcyjne wersje plików aplikacji. Zostały one skrócone, tak by były możliwie jak najkrótsze; w celu poprawy wydajności działania usunięto z nich także informacje wykorzystywane podczas debugowania.
- Kolejnym elementem jest katalog *node\_modules* — zawiera on wszystkie zależności aplikacji pobrane z repozytorium npm.
- Kolejnym element to katalog *public* — jest on przeznaczony do przechowywania zasobów statycznych, takich jak plik *index.html*, którego użyjemy do stworzenia końcowej aplikacji.
- Następny element jest jednocześnie jednym z najważniejszych, chodzi o katalog *src*. Zgodnie z tym, co sugeruje jego nazwa<sup>1</sup>, katalog ten zawiera kody źródłowe projektu. Wszystkie umieszczone w nim pliki z rozszerzeniem *.tsx* oznaczają komponenty Reacta. Z kolei pliki z rozszerzeniem *.ts* to zwyczajne pliki źródłowe z kodem TypeScript. Pliki z rozszerzeniem *.css* zawierają arkusze stylów określające postać komponentów (takich plików może być więcej niż jeden). I w końcu pliki z rozszerzeniem *.d.ts* zawierają informacje o typach, których kompilator TypeScriptu będzie używał podczas statycznego sprawdzania typów stosowanych w kodzie źródłowym aplikacji.
- Kolejnym elementem jest plik *.gitignore*. Ten plik jest wykorzystywany przez system zarządzania kodami źródłowymi git oraz jego repozytorium GitHub. Zgodnie z tym, co sugeruje jego nazwa, plik ten służy do wskazania systemowi git, których plików i katalogów nie należy umieszczać w repozytorium.
- Pliki *package.json* oraz *package-lock.json* służą do konfigurowania i określania zależności. Oprócz tego, mogą one zawierać informacje o sposobie budowania aplikacji, zapewniać możliwość uruchamiania skryptów oraz przechowywać informacje wykorzystywane przez Jest — framework do wykonywania testów.
- I w końcu ostatnim elementem jest plik *tsconfig.json*, który już przedstawiłem w rozdziale 2., pt. „Prezentacja języka TypeScript”. Służy on do konfiguracji działania kompilatora języka TypeScript. Warto zwrócić uwagę na to, że domyślnie włączony jest tryb ścisłej kontroli typów, co oznacza, że nie możemy niejawnie używać typów `any` ani `undefined`.

Skoro już pobieżnie przyjrzeliliśmy się zawartości wygenerowanego projektu, przyjrzyjmy się zawartości niektórych spośród tych plików. Zaczniemy od pliku *package.json*. Jest on całkiem długi i składa się z wielu sekcji, jednak tutaj ograniczę się do przedstawienia tylko kilku najważniejszych spośród nich:

- Sekcja `dependencies` — zawiera biblioteki, których nasza aplikacja będzie używać, by zapewniać swoje możliwości funkcjonalne. Do tych zależności należy sam framework React, język TypeScript oraz biblioteki Jest, których

<sup>1</sup> „src” to skrót od angielskich słów „source code”, czyli kod źródłowy — *przyp. tłum.*

będziemy używali do testowania kodu. Zależności @types zawierają pliki definicji języka TypeScript. Pliki te zawierają informacje o typach frameworka napisanych w języku JavaScript. Innymi słowy, pliki te dostarczają kompilatorowi TypeScriptu informacje o strukturze poszczególnych typów używanych przez framework, dzięki czemu kompilator będzie mógł sprawdzać kod i deklaracje typów.

- Kolejna sekcja nosi nazwę devDependencies, zazwyczaj zawiera ona informacje o zależnościach projektu wykorzystywanych podczas prac programistycznych (co odróżnia je od zależności podanych w sekcji dependencies, która zwykle zawiera informacje jedynie o zależnościach koniecznych do działania aplikacji). Jednak my nie będziemy z niej korzystać — z jakichś powodów zespół twórców Reacta zdecydował się połączyć obie te sekcje w jedną, dependencies. Niemniej jednak warto pamiętać o sekcji devDependencies, gdyż w wielu projektach jest ona używana.
- Sekcja scripts — służy do przechowywania skryptów używanych do zarządzania aplikacją. Na przykład skrypt start jest wykonywany przez polecenie npm run start (które można także zapisać w krótszej postaci: npm start). W tej sekcji można także zapisywać własne skrypty (z możliwości tej skorzystamy w dalszej części książki), wykonujące różne operacje, takie jak wdrażania produkcyjnej wersji aplikacji na serwer.

Konieczne trzeba pamiętać, że projekty tworzone przez create-react-app zostały w bardzo dużym stopniu zmodyfikowane przez zespół twórców Reacta. W głównej mierze zostały one zoptymalizowane i ukryto w nich skrypty oraz ustawienia konfiguracyjne, które nie są zbyt często używane, takie jak ustawienia konfiguracyjne i skrypt narzędzia webpack. Gdybyś chciał przyrzeć się tym skryptom i plikom konfiguracyjnym, możesz to zrobić wykonując polecenie npm run eject. Pamiętaj jednak, że operacja jest nieodwracalna — nie będziesz w stanie przywrócić aplikacji do wcześniejszej postaci. Wykonanie polecenia eject nie zapewnia żadnych znaczących korzyści, dlatego też nie będziemy go używać w tej książce.

A teraz przyjrzyjmy się niektórym ze skryptów tworzących aplikację. Poniżej przedstawilem kod pliku `index.tsx`, umieszczonego w katalogu `src`:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

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

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Jeszcze raz przypomnę, że wszystkie pliki zawierające komponenty Reacta zwracające kod JSX mają rozszerzenie `.tsx`. Ten plik przedstawiłem dlatego, że stanowi on punkt wejścia do aplikacji Reacta. To właśnie od niego zaczyna się proces budowania aplikacji Reacta, wykonywany podczas jej działania. Analizując ten plik łatwo zauważyć, że zaczyna się on od składni ES6 służącej do importowania używanych zależności. Jak widać, importowany jest sam React, jak również powiązane z nim moduły, w tym także moduł `App`, którym zajmujemy się już niebawem. Poniżej instrukcji importu umieszczone jest wywołanie `ReactDOM.render`, które spowoduje „zapisanie” na stronie kodu HTML wszystkich połączonych komponentów. Wywołanie to wymaga przekazania dwóch argumentów. Pierwszym z nich jest komponent Reacta najniższego poziomu, od którego rozpocznie się renderowanie aplikacji, a drugim — element HTML, w którym zostanie umieszczona wygenerowana zawartość. Jak widać w kodzie, komponent `App` został umieszczony wewnątrz komponentu `React.StrictMode`. Komponent `React.StrictMode` jest jedynie dodatkiem ułatwiającym tworzenie aplikacji. Kiedy zostanie przygotowana jej produkcyjna wersja, komponent ten nie będzie odgrywał żadnego znaczenia, jak również nie będzie miał wpływu na wydajność działania. Niemniej jednak, podczas prowadzenia prac programistycznych, dostarcza on dodatkowych informacji na temat potencjalnych problemów, które mogą wystąpić w kodzie. Choć w przyszłości jego działanie może ulec zmianie, to obecnie udostępnia on, między innymi, te informacje, które opisałem na poniższej liście:

- Wskazuje komponenty, których cykl życia może być niebezpieczny. Wyświetlane są komponenty korzystające z takich metod cyklu życia, jak: `componentWillMount`, `componentWillReceiveProps` oraz `componentWillUpdate`. Te potencjalne problemy nie będą występować w przypadku korzystania z *hooków*, jednak warto o nich pamiętać w razie stosowania starszych komponentów, tworzonych w oparciu o klasy.
- Ostrzeżenia dotyczące stosowania starego API odwołań — chodzi o stary sposób tworzenia odwołań do elementów HTML, a nie komponentów Reacta, który polegał na stosowaniu łańcuchów takich jak: `<div ref="myDiv">{content}</div>`. Ponieważ metoda ta używa łańcucha znaków, może powodować problemy i obecnie preferowanym rozwiązaniem jest użycie funkcji `React.createRef`. Do zagadnień związanych z odwołaniami wrócimy jeszcze w dalszej części książki.
- Ostrzeżenie o niezalecanym sposobie korzystania z metody `findDOMNode`. Metoda `findDOMNode` została ostatnio uznana za niezalecaną, gdyż narusza zasady abstrakcji. Mianowicie zapewnia ona komponentom nadrzędnym w drzewie komponentów możliwość wykonywania operacji na rzecz konkretnych komponentów podrzędnych. Umieszczanie takich odwołań w kodzie oznacza, że w przyszłości będzie go trudniej modyfikować, gdyż kod komponentu nadrzędnego będzie zależeć od czegoś, co już ma istnieć w drzewie komponentów. Zasady programowania obiektowego, w tym także zasadę abstrakcji, przedstawiłem w rozdziale 2., pt. „Prezentacja języka TypeScript”.
- Wykrywanie nieoczekiwanych efektów ubocznych. Efekty uboczne to niezamierzone konsekwencje rozwiązań zastosowanych w kodzie. Na przykład, jeśli stan mojego komponentu klasowego jest inicjowany w konstruktorze w zależności od jakiejś innej funkcji lub właściwości, to niedopuszczalna byłaby sytuacja, w której przekazywane wartości co jakiś czas są inne. W celu

umożliwienia wychwytywania problemów tego typu, komponent React.

→ StrictMode będzie dwukrotnie wykonywał pewne wywołania, takie jak na przykład wywołania konstruktorów lub metody `getDerivedStateFromProps`.

Warto zwrócić uwagę na to, że komponent ten działa w opisany sposób wyłącznie w trakcie prac nad aplikacją.

- Wykrywanie stosowania starego API kontekstu. API kontekstu to mechanizm Reacta udostępniający globalny stan we wszystkich komponentach aplikacji. Obecnie dostępna jest już nowsza wersja tego API, a stosowanie starszej zostało uznane za niezalecane. Ten test sprawdza, czy nie używamy starego API.

Większość z wykonywanych testów jest związana ze stosowaniem starszego stylu tworzenia komponentów, bazującego na klasach. Jednak, ponieważ znaczna większość istniejącego kodu, którym najprawdopodobniej będziemy musieli się zajmować, korzysta właśnie z tego stylu pisania komponentów, dlatego też znajomość tej możliwości wciąż ma znaczenie.

A teraz przyjrzymy się plikowi `App.tsx`:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</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;
```

Konieczniesz zwrócić uwagę na to, że składnia JSX, taka jak ta przedstawiona na powyższym przykładzie, nie zawiera klasycznych znaczników HTML. Jest to niestandardowy kod JavaScript. Dlatego za każdym razem, kiedy pojawi się potencjalny konflikt ze słowami kluczowymi JavaScriptu, React zastosuje inną nazwę. Na przykład `class` jest zastrzeżonym słowem kluczowym języka JavaScript, dlatego też do określania nazw klas CSS React będzie używał nazwy `className`.



Choć plik *index.tsx* jest głównym punktem startowym aplikacji Reacta, to jednak, jeśli chodzi o same komponenty, które będziemy tworzyć na potrzeby naszej aplikacji, będą się one rozpoczynać do pliku *App.tsx*. To sprawia, że plik *App.tsx* ma dla nas szczególne znaczenie. Przeanalizujmy zatem niektóre elementy jego kodu:

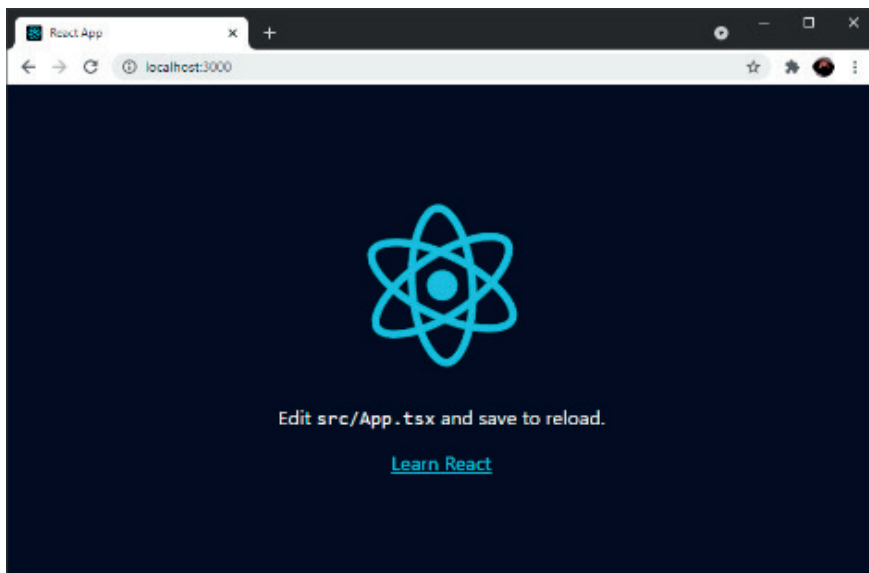
- W pierwszej kolejności, na samym początku pliku importujemy pakiet React. Jeśli zajrzymy do katalogu *node\_modules*, znajdziemy w nim podkatalog *react* i to właśnie do niego odwołuje się instrukcja importu. Wszelkie operacje importu odnoszące się do kodu, którego nie napisaliśmy sami, będą się odwoływały właśnie do katalogu *node\_modules*.
- Kolejnym elementem jest instrukcja importu elementu *logo*. Zasoby graficzne są importowane do zmiennych JavaScriptu, a w tym przypadku jest to zmienna *logo*. Jak widać, ponieważ w tym przypadku nie mamy do czynienia z modulem *npm*, odwołanie rozpoczyna się od znaku kropki. W odwołaniach do modułów *npm* stosowanie ścieżek względnych nie jest konieczne, gdyż system sam „wie”, o który katalog chodzi — *node\_modules*.
- W kolejnym wierszu kodu importowany jest plik *App.css*. Jest to arkusz stylów i z tego względu nie trzeba tu podawać żadnej zmiennej JavaScriptu. Jednak, ponieważ nie jest to pakiet *npm*, konieczne jest podanie względnej ścieżki dostępu do niego.
- Kolejnym elementem jest komponent *App*. Jak można się zorientować na podstawie jego składni, jest to komponent funkcyjny. *App* jest głównym komponentem, nadrzędnym dla całej aplikacji. Komponent ten nie dysponuje żadnym własnym stanem, a jego działanie ogranicza się do wyrenderowania zawartości. A zatem instrukcja *return* zwraca wyrenderowaną zawartość komponentu, utworzoną na podstawie kodu **JSX**.
- Znacznie bardziej szczegółowe informacje na temat **JSX** można znaleźć w następnych rozdziałach książki, póki co zaznaczę tylko, że **JSX** to składnia przypominająca kod **HTML**, lecz zapisywana w **JavaScriptcie**. Została ona opracowana przez zespół twórców Reacta, by uprościć tworzenie treści **HTML** w komponentach Reacta. Najważniejszym zagadnieniem związanym z **JSX** jest to, że choć kod **JSX** wygląda niemal tak samo jak kod **HTML**, to jednak w rzeczywistości wcale nim nie jest, a co za tym idzie, czasami działa on w nieco inny sposób.
- Odwołania do nazw klas **CSS**, które normalnie podaje się w atrybucie **HTML** *class*, należy podawać używając *className*, jak pokazałem na powyższym listingu. Zmiana ta jest spowodowana tym, że *class* jest słowem kluczowym języka **JavaScript** i nie może być używane w kodzie **JSX**.
- W kodzie **JSX** są używane nawiasy klamrowe, a nie łańcuchy znaków. Na przykład wartość atrybutu *src* znacznika *img* jest określana na podstawie zmiennej **JavaScript** *logo*, i to właśnie ta zmienna jest zapisywana w nawiasach klamrowych. Jeśli chcemy przekazać łańcuch znaków, należy go zapisać w cudzysłowie.

Spróbujmy zatem uruchomić naszą aplikację w trybie do prowadzenia prac programistycznych i zobaczyć, jak wygląda jej główna strona. W tym celu wykonaj następujące polecenie:

```
npm start
```



Kiedy wykonasz to polecenie, w przeglądarce powinna zostać wyświetlona strona przedstawiona na rysunku 4.2.



**Rysunek 4.2.** Strona początkowa aplikacji

Jak widać, w przeglądarce zostały wyświetlone tekst i logo z pliku *App.tsx*, gdyż jest to główny punkt startowy naszej aplikacji. Kiedy zaczniemy tworzenie kodu, pozostawimy ten serwer w takim stanie, w jakim jest, czyli działający, a później, po zapisaniu dowolnego pliku źródłowego, strona aplikacji zostanie automatycznie odświeżona, co pozwoli nam na bieżąco przeglądać wprowadzane zmiany.

Aby lepiej zrozumieć tworzenie komponentów Reacta, jak również by dowiedzieć się, jak działa trasowanie, utwórzmy pierwszy, bardzo prosty komponent:

1. W katalogu *src* utwórz nowy plik o nazwie *Home.tsx* i zapisz w nim następujący kod:

```
import React, { FC } from "react";

const Home: FC = () => {
  return <div>Witaj, świecie! Komponent Home</div>;
};

export default Home;
```

2. Jak widać, utworzyliśmy komponent o nazwie *Home*, który zwraca znacznik *div* zawierający tekst *Witaj, świecie!* Strona główna. Należy także zwrócić uwagę, że określamy typ komponentu — będzie to komponent typu *FC*, czyli komponent funkcyjny (ang. *functional component*). Komponenty funkcyjne to jedyne dostępne rozwiązanie, jeśli chcemy korzystać z *hooków* Reacta, a nie z komponentów klasowych. Komponenty funkcyjne wprowadzono, gdyż zespół twórców Reacta

uważa, że kompozycja jest lepszym sposobem wielokrotnego wykorzystywania kodu niż dziedziczenie. Należy jednak zauważyć, że sama idea wielokrotnego stosowania kodu, pomijając sposób jej zapewniania, nie traci na aktualności.

3. Teraz, aby wyświetlić nasz komponent na ekranie, musimy go dodać do pliku *App.tsx*. Przy tej okazji skorzystamy także z mechanizmu trasowania, aby poznać i tę możliwość Reacta. W pierwszej kolejności zmodyfikuj plik *index.tsx* w sposób przedstawiony na poniższym przykładzie:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { BrowserRouter } from "react-router-dom";

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

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Do pliku *index.tsx* dodaliśmy komponent o nazwie *BrowserRouter*. Jest on jednym z elementów routera Reacta i stanowi bazowy komponent, zapewniający możliwość obsługi tras w aplikacji. Ponieważ umieściliśmy w nim komponent *App*, a w nim z kolei będzie umieszczona cała reszta naszej aplikacji, oznacza to, że usługi trasowania będą dostępne w całej aplikacji.

4. Mamy zamiar korzystać z routera Reacta, utwórzmy więc komponent dla drugiej strony; będzie on nosił nazwę *AnotherScreen*:

```
import React, { FC } from "react";

const AnotherScreen: FC = () => {
  return <div>Witaj, świecie! Oto druga strona</div>;
};

export default AnotherScreen;
```

5. Teraz zmodyfikuj kod zapisany w pliku *App.tsx*:

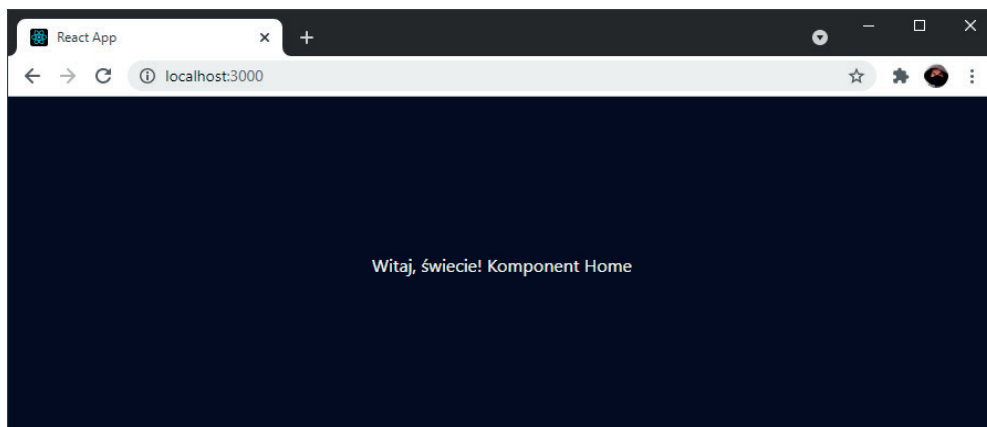
```
import React from 'react';
import './App.css';
import Home from './Home';
import AnotherScreen from './AnotherScreen';
import { Switch, Route } from "react-router";
```

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Switch>
          <Route exact={true} path="/" component={Home}></Route>
          <Route path="/another" component={AnotherScreen}></Route>
        </Switch>
      </header>
    </div>
  );
}

export default App;
```

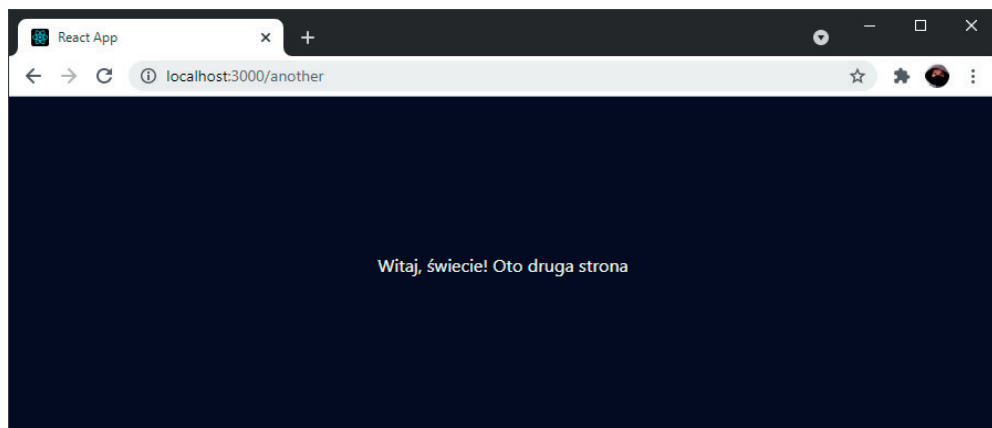
Jak widać, w tym pliku zmieniliśmy instrukcje importu i zawartość elementu header. Przeglądając kod tworzący zawartość strony zauważysz zapewne, że pośród innych znaczników dodaliśmy także komponent `Switch`. Swoim działaniem komponent ten przypomina instrukcję `switch`. Informuje on router Reacta o tym, który komponent należy wyświetlić w zależności od aktualnie podanego adresu URL. Wewnątrz komponentu `Switch` umieściliśmy dwa komponenty `Route`. Pierwszy z nich odpowiada trasie domyślnej, co wyraźnie pokazuje zastosowana ścieżka: `path="/"`. W razie zastosowania tej trasy React wyświetli komponent `Home` (zwróć także uwagę na zastosowanie słowa `exact`, które oznacza, że adresy URL muszą być dokładnie takie same). Druga trasa używa ścieżki o postaci `"/another"`. Innymi słowy, kiedy w polu adresu przeglądarki wpiszemy tę ścieżkę, aplikacja wczyta i wyświetli komponent `AnotherScreen`.

6. Jeśli wcześniej zostawiłeś działający serwer uruchomiony przy użyciu polecenia `npm start`, to teraz zobaczysz w przeglądarce zmodyfikowaną stronę główną aplikacji (będzie ona wyglądać jak ta przedstawiona na rysunku 4.3).



Rysunek 4.3. Komponent Home

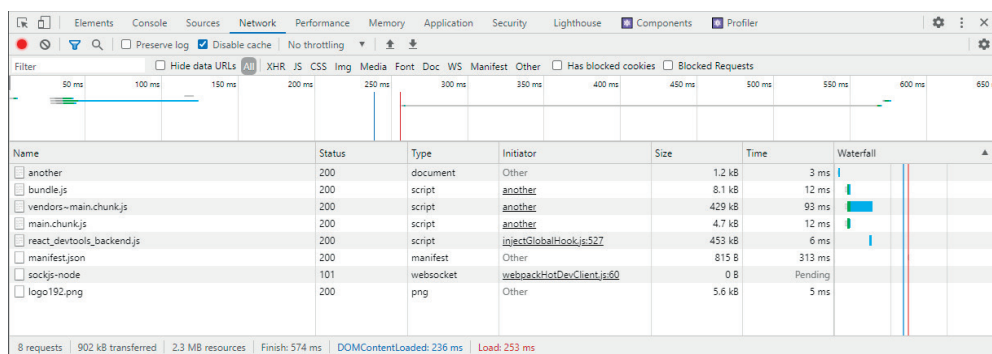
- Jeśli spojrzysz na pole adresu przeglądarki, to przekonasz się, że aktualnie jesteśmy na głównej stronie witryny. Spróbuj zatem wpisać adres `http://localhost:3000/another`; w efekcie w przeglądarce zostanie wyświetlona strona przedstawiona na rysunku 4.4.



**Rysunek 4.4.** Drugi komponent aplikacji — AnotherScreen

Jak widać, po zmianie adresu aplikacja wczytała i wyświetliła komponent AnotherScreen, postępując zgodnie z instrukcjami dotyczącymi obsługi tego adresu URL.

Co więcej, jeśli otworzysz debugger przeglądarki Chrome, zauważysz, że zmiana wyświetlanego komponentu nie wiązała się z wykonaniem żadnego nowego żądania do serwera (patrz rysunek 4.5). Stanowi to kolejny dowód potwierdzający, że router Reacta nie przekazuje do serwera żądań dotyczących podawanych ścieżek, oraz że istnieją one jedynie lokalnie w przeglądarce.



**Rysunek 4.5.** Debugger przeglądarki Chrome

Przedstawiłem tu jedynie krótki i prosty przykład tworzenia aplikacji Reacta, który ma za zadanie ułatwić Ci rozpoczęcie korzystania z tego frameworka.

W tym podrozdziale zaprezentowałem wewnętrzny sposób działania Reacta i pokazałem, jak można utworzyć projekt aplikacji w tym frameworku. Informacje te przydadzą Ci się w następnych rozdziałach, w których znacznie dokładniej przyjrzymy się zagadnieniom związanym z tworzeniem aplikacji Reacta.

---

## Podsumowanie

W tym rozdziale wyjaśniłem, w jaki sposób niegdyś tworzone witryny WWW oraz jak robi się to dzisiaj. Poznałeś także ograniczenia starego stylu tworzenia aplikacji internetowych oraz dowiedziałeś się, w jaki sposób aplikacje jednostronicowe starają się eliminować te ograniczenia. Wyjaśniłem także, że podstawowym celem tworzenia aplikacji jednostronicowych jest upodobnienie aplikacji internetowych do aplikacji rodzimych, przeznaczonych dla komputerów stacjonarnych i urządzeń przenośnych. W ostatniej części rozdziału zamieściłem krótkie wprowadzenie do zagadnień związanych z tworzeniem aplikacji Reacta oraz komponentów, z których te aplikacje się składają.

W następnym rozdziale wykorzystamy tę wiedzę, by znacznie bardziej szczegółowo zająć się zagadnieniami tworzenia komponentów Reacta. Przyjrzymy się komponentom klasowym i porównamy je z nowszym rodzajem komponentów — komponentami funkcyjnymi, korzystającymi z *hooków*. Informacje o tworzeniu aplikacji internetowych oraz aplikacji korzystających z frameworka React, którymi już dysponujesz, znacząco ułatwią Ci zrozumienie zagadnień zamieszczonych w następnym rozdziale.

# Tworzenie aplikacji Reacta z wykorzystaniem hooków

W tym rozdziale zajmiemy się zagadnieniami związanymi z tworzeniem aplikacji Reacta z wykorzystaniem tak zwanych **hooków** (ang. *Hooks*), nazywanych także czasami *hakami*. Porównam ten nowy styl tworzenia komponentów ze starszym, bazującym na wykorzystaniu klas i wyjaśnię, dlaczego korzystanie z *hooków* jest lepszym rozwiązaniem. Oprócz tego przedstawię najlepsze praktyki związane ze stosowaniem *hooków*, które pozwolą Ci stworzyć możliwie jak najlepszy kod.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- przedstawieniem ograniczeń komponentów klasowych;
- przedstawieniem *hooków* i wyjaśnieniem korzyści, jakie zapewniają;
- porównaniem starego stylu tworzenia komponentów i stosowania *hooków*.

## Wymagania techniczne

Powinieneś dysponować podstawową znajomością sposobów tworzenia aplikacji internetowych oraz jednostronicowych. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial05* (*Chap5*).

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog, o nazwie *rozdzial05*.

## Wyjaśnienie ograniczeń i problemów związanych ze stosowaniem starych komponentów klasowych

W tym podrozdziale przedstawię stary sposób tworzenia komponentów Reacta, tak zwanych komponentów klasowych. Wyjaśnię, dlaczego bazujący na dziedziczeniu sposób wielokrotnego wykorzystywania kodu oraz dostosowane do tego typu komponentów metody cyklu życia, choć opracowane w zgodzie z regułami sztuki, to jednak w ostatecznym rozrachunku nie zapewniały dostatecznie dobrych możliwości wielokrotnego stosowania oraz właściwej struktury komponentów. Choć w swoim kodzie nie będziemy pisać komponentów tego rodzaju, to jednak jest niezwykle ważne, by doskonale zrozumieć zasady ich tworzenia i działania, gdyż większość istniejącego kodu aplikacji Reacta korzysta właśnie z komponentów klasowych — *hooki* są bowiem rozwiązaniem stosunkowo nowym. Jako profesjonalny programista aplikacji Reacta będziesz musiał analizować i pielęgnować takie komponenty starszego typu, aż do momentu zastąpienia ich nowszymi, korzystającymi z *hooków*.

Aby zrozumieć ograniczenia komponentów klasowych, musimy najpierw dokładniej się im przyjrzeć. Każda aplikacja Reacta składa się z wielu odrębnych struktur, nazywanych komponentami. W przypadku stylu bazującego na tworzeniu komponentów klasowych, komponenty te są instancjami klas wersji ES6 JavaScriptu, dziedziczącymi po `React.Component`. Najprościej rzecz ujmując, komponent jest tworem, który może zawierać dane (nazywane stanem) i który w oparciu o ten stan będzie generować kod HTML zapisywany w języku JSX. Choć komponenty mogą być bardzo złożone, to jednak na swoim najprostszym poziomie są właśnie takimi tworem.

Komponenty klasowe zazwyczaj dysponują własnym stanem, choć nie zawsze tak jest i nie ma takiego wymogu. Co więcej, komponenty tego rodzaju mogą mieć komponenty podrzędne — kolejne, zwyczajne komponenty Reacta, które są umieszczone w kodzie funkcji render komponentu nadrzędnego i z tego powodu będą renderowane wraz z nim.

Wszystkie komponenty klasowe muszą dziedziczyć po `React.Component`. W ten sposób uzyskują one wszystkie możliwości komponentów Reacta, w tym wszystkie funkcje cyklu życia. Te funkcje są procedurami obsługi zdarzeń udostępnianymi przez Reacta i zapewniającymi programistom możliwość obsługi zdarzeń występujących w określonych momentach cyklu życia komponentów Reacta. Innymi słowy, funkcje te pozwalają nam, programistom, wstrzykiwać do logiki działania komponentów własny kod, który będzie wykonywany w ściśle określonych momentach.

## Stan

O stanie wspominałem już w rozdziale 4., pt. „Przedstawienie koncepcji aplikacji jednostronowych oraz ich realizacja z użyciem frameworka React”. Jednak zanim zajmiemy się zagadnieniami związanymi z komponentami Reacta, warto przyrzeć się nieco dokładniej samemu frameworkowi. React używa formatu JSX do renderowania kodu HTML, który ma być wyświetlany w przeglądarce. Jednak czynnikiem, który wymusza rozpoczęcie tej operacji, jest stan komponentu, a konkretniej, jakakolwiek zmiana tego stanu. Czym zatem jest stan komponentu? W przypadku komponentów klasowych Reacta jest to pole o nazwie `state`. Zawartością tego pola jest obiekt, który może zawierać dowolnie wiele właściwości opisujących dany komponent. Obiekt stanu nie powinien sam dysponować żadnymi funkcjami, choć nic nie stoi na przeszkodzie, by tworzyć funkcje będące składowymi samego komponentu.

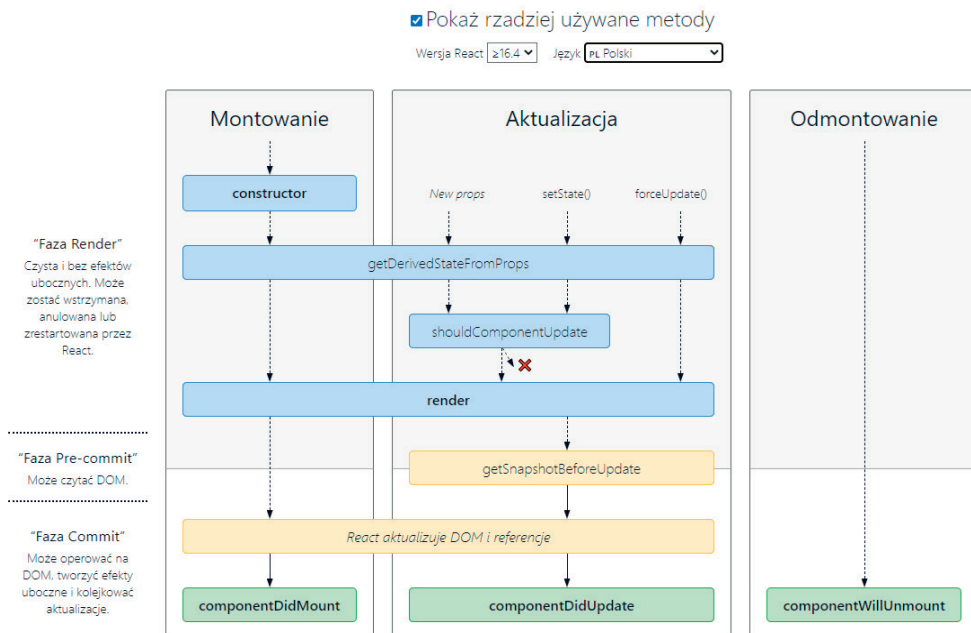
Jak już wspominałem, zmiana stanu sprawia, że React ponownie wyrenderuje komponent. To właśnie zmiany stanu powodują renderowanie komponentów Reacta, a same komponenty zawierają wyłącznie własne elementy interfejsu użytkownika, co jest dobrym sposobem zapewniania separacji obowiązków i pozwala na tworzenie przejrzystego kodu. Zmiany stanu w komponentach klasowych Reacta są wywoływane przez wykonanie funkcji `setState`. Ta funkcja przyjmuje jeden parametr, którym jest nowy stan komponentu, a jej wywołanie sprawia, że React nieco później, asynchronicznie ustawi nowy stan komponentu. Oznacza to, że faktyczna zmiana stanu nie następuje od razu, lecz moment jej wprowadzenia jest kontrolowany przez Reacta.

Oprócz tego istnieje także możliwość współdzielenia stanu komponentu przy użyciu właściwości *props*. Są to właściwości stanu, które zostały przekazane do komponentów podrzędnych danego komponentu. Podobnie jak w przypadku zmiany stanu, jeśli zmieni się właściwość *props*, to spowoduje to ponowne wyrenderowanie komponentu podrzędnego. Komponent podrzędny zostanie ponownie wyrenderowany podczas renderowania komponentu nadrzędnego. Warto zwrócić uwagę na to, że ponowne wyrenderowanie nie oznacza aktualizacji całego interfejsu użytkownika komponentu. Podczas renderowania będzie wykonywany proces uzgadniania, zatem zmienione zostaną tylko te elementy, które tego wymagają, zważywszy na aktualny stan oraz to, co aktualnie jest wyświetlone na ekranie.



## Metody cyklu życia

Zamieszczony poniżej rysunek 5.1 stanowi doskonałą prezentację metod cyklu życia komponentów klasowych Reacta. Jak widać, jest on dość skomplikowany. Oprócz przedstawionych na rysunku istniało także kilka przestarzałych funkcji, takich jak `componentWillReceiveProps`, które zostały całkowicie usunięte z cyklu życia komponentów, gdyż ich stosowanie przysparzało problemów, takich jak niepożądane operacje renderowania, czy też nieskończone pętle odświeżania komponentu.



**Rysunek 5.1.** Cykl życia komponentów klasowych Reacta

Rysunek pochodzi ze strony <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>.

Przeanalizujemy ten diagram, zaczynając do najwyższego poziomu. Przede wszystkim, jak widać, jest on podzielony na trzy główne części: **Montowanie**, **Aktualizacja** i **Odmontowanie**. Montowanie (ang. *Mounting*) to po prostu tworzenie i inicjalizacja komponentu, jak również następujące później dodanie zainicjowanego komponentu do wirtualnego modelu DOM Reacta. Ten wirtualny DOM, którego React używa do uzgadniania zarządzanej przez siebie zawartości aplikacji i faktycznego DOM istniejącego w przeglądarce, opisałem w rozdziale 4., pt. „Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React”. Aktualizacja (ang. *Updating*) odnosi się do operacji ponownego renderowania komponentu. Aktualizacje są wykonywane, kiedy zmieni się stan komponentu i konieczne jest zaktualizowanie jego interfejsu użytkownika. I w końcu odmontowywanie (ang. *Unmounting*) zachodzi, kiedy komponent nie jest już dłużej potrzebny i zostaje usunięty z modelu DOM.

Teraz opiszę poszczególne metody cyklu życia — jest ich kilka, więc najwygodniej będzie przedstawić je w formie listy.

## Montowanie

Do tej grupy zaliczają się następujące metody:

- `constructor`. Konstruktor klasy właściwie nie zalicza się do metod cyklu życia komponentów Reacta. Tradycyjnie konstruktory są używane do inicjalizacji stanu oraz utworzenia powiązań z niestandardowymi funkcjami obsługi zdarzeń. Jak zapewne pamiętasz z rozdziału 3., pt. „Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript”, funkcja JavaScriptu `bind` jest używana do zmiany obiektu `this`, który będzie używany w danej funkcji; operacje te są właśnie wykonywane w konstruktorze.
- `getDerivedStateFromProps(props, state)`. Z tej funkcji będziemy korzystać w razie przechowywania stanu lokalnego we właściwościach *props* komponentu nadrzędnego. Jest to funkcja statyczna. Należy jej używać sporadycznie, gdyż powoduje dodatkowe wyrenderowanie komponentu. Znajduje zastosowanie także podczas aktualizacji komponentów.
- `Render`. W przypadku ponownego renderowania komponentu ta metoda może być wykonywana także podczas aktualizacji. To właśnie ona wywołuje proces uzgadniania. Powinna ona zwracać wyłącznie kod JSX, który może być także umieszczany w tablicach lub w zwyczajnym tekście. Jeśli na podstawie stanu lub właściwości *props* uznamy, że nie ma konieczności renderowania komponentu, to metoda ta powinna zwrócić wartość `null`. Może ona także zwracać wartość logiczną, jednak uważam, że takie rozwiązanie jest przydatne wyłącznie w przypadku wykonywania testów.
- `componentDidMount`. Ta metoda jest wywoływana po zakończeniu montowania (inicjalizacji) komponentu. Można w niej umieszczać wywołania funkcji API wykonujące operacje sieciowe. Oprócz tego można jej także używać do subskrybowania zdarzeń, jednak w takim przypadku należy pamiętać o anulowaniu subskrypcji w metodzie `componentWillUnmount`, gdyż w przeciwnym razie może dochodzić do wycieków pamięci. W kodzie tej metody można także wywoływać metodę `setState`, by zmieniać lokalne dane stanu komponentu, jednak powoduje to wyzwolenie drugiego cyklu renderowania, więc takich rozwiązań należy używać sporadycznie. Metoda `setState` jest używana do aktualizowania lokalnego stanu komponentu.
- Do niebezpiecznych metod (których nie należy używać) należą: `UNSAFE_componentWillMount`, `UNSAFE_componentWillReceiveProps` oraz `UNSAFE_componentWillUpdate`.

## Aktualizowanie

Teraz przyjrzymy się metodom należącym do grupy aktualizowania:

- `shouldComponentUpdate(nextProps, nextState)`. Ta metoda służy do określania, czy należy ponownie wyrenderować komponent, czy nie. Zazwyczaj jej działanie polega na porównaniu poprzednich i nowych właściwości *props*.

- `getSnapshotBeforeUpdate(prevProps, prevState)`. Ta metoda jest wykonywana bezpośrednio przed aktualizacją modelu DOM, dzięki czemu możemy jej użyć do przechwycenia stanu DOM zanim ten zostanie zmieniony przez Reacta. Jeśli zwrócimy jakiś wynik z tej metody, zostanie on przekazany jako parametr do metody `componentDidUpdate`.
- `componentDidUpdate(prevProps, prevState, snapshot)`. Ta metoda jest wywoływana bezpośrednio po zakończeniu ponownego renderowania komponentu. Można jej używać do wprowadzania zmian w kompletnym modelu DOM, można w niej również wywoływać metodę `setState`, jednak w tym przypadku trzeba zastosować warunek logiczny, który nie dopuści do występowania nieskończonej pętli aktualizacji. Stan przekazywany przy użyciu parametru `snapshot` jest zwracany przez funkcję `getSnapshotBeforeUpdate`.

## Odmontowanie

Do tej grupy należy tylko jedna metoda:

- `componentWillUnmount`. Przypomina ona nieco funkcję `dispose` z innych języków programowania, takich jak C#, i można jej używać do wykonywania czynności porządkowych, takich jak usuwanie procedur obsługi zdarzeń lub innych subskrypcji.

Najważniejszym zagadnieniem związanym z korzystaniem ze wszystkich metod cyklu życia komponentów jest unikanie niepożądanych oraz niepotrzebnych operacji renderowania. Musimy zatem wybrać tę spośród metod cyklu życia, która zapewni najmniejsze prawdopodobieństwo ponownego wyrenderowania komponentu, bądź też, jeśli musimy skorzystać z konkretnej metody, tak by umieszczony w niej kod był wykonywany w odpowiednim czasie, musimy dodać testy właściwości *props* i stanu, które uchronią nas przed niepotrzebnym renderowaniem komponentu. Zachowanie kontroli nad operacjami renderowania ma kluczowe znaczenie, gdyż jeśli się nam to nie uda, to niewielka szybkość działania aplikacji oraz występujące w niej błędy doprowadzą do obniżenia wrażeń użytkownika.

Przyjrzyjmy się teraz nieco dokładniej najważniejszemu spośród metod cyklu życia. Zaczniemy od metody `getDerivedStateFromProps`. Ogólnie rzecz biorąc, najlepiej będzie unikać stosowania tej funkcji, bądź też używać jej sporadycznie. Z doświadczenia muszę stwierdzić, że korzystanie z niej w znacznym stopniu utrudnia określanie, kiedy komponent zostanie ponownie wyrenderowany. Na ogół, funkcja ta przyczynia się do niepotrzebnego renderowania komponentu, co z kolei może powodować niezamierzone działanie aplikacji, którego przyczyny mogą być trudne do określenia.

Zespół twórców Reacta zaleca stosowanie rozwiązań alternatywnych zamiast wywoływania tej metody i powinniśmy starać się z nich korzystać, gdyż niemal zawsze ich zastosowanie jest bardziej uzasadnione, a działanie bardziej spójne:

- Kiedy musimy wyzwoić jakieś działanie na podstawie zmiany wartości właściwości. Na przykład po otrzymaniu danych zwróconych przez jakieś żądanie sieciowe lub w celu wykonania jakieś innej operacji. W takich przypadkach lepsze będzie zastosowanie metody `componentDidUpdate`. Stwarza ona mniejsze niebezpieczeństwo wystąpienia nieskończonej pętli, o ile tylko

przed wykonaniem jakichkolwiek operacji powodujących zmianę stanu wykonamy odpowiedni test. Na przykład przed wywołaniem metody `setState` w celu ustawienia stanu możemy używać parametru `prevProps` i porównywać jego zawartość z lokalnymi wartościami stanu komponentu.

- Możemy stosować techniki zapamiętywania (ang. *memoization*; jest to technika programistyczna, a nie rozwiązanie charakterystyczne dla Reacta). Zapamiętywanie przypomina nieco korzystanie z pamięci podręcznej, przy czym zamiast czasu ważności, aktualizacja tej pamięci następuje w efekcie zmiany wartości zmiennej. W kontekście Reacta oznacza to, że używamy właściwości lub funkcji, która najpierw sprawdza, czy wartość właściwości uległa zmianie, i aktualizuje stan komponentu wyłącznie w przypadku, gdy wartość ta faktycznie została zmieniona. React udostępnia wbudowane opakowanie komponentów o nazwie `React.memo`, ułatwiające stosowanie tej techniki. Powoduje ono wyzwolenie operacji ponownego renderowania wyłącznie w przypadku zmiany właściwości komponentu podrzędnego, a nie w momencie ponownego renderowania komponentu nadrzędnego.
- Możemy zadbać o to, by mieć pełną kontrolę nad tworzonymi komponentami, co w zasadzie sprowadza się do tego, by nie dysponowały one żadnym własnym stanem i były renderowane pod kontrolą komponentu nadrzędnego, kiedy on będzie renderowany lub kiedy zmienią się jego właściwości. Facebook zaleca także stosowanie komponentów niekontrolowanych, przy czym w ich przypadku należy zmieniać klucz komponentu (klucz to unikalny identyfikator komponentu), co z kolei sprawi, że komponent zostanie ponownie wyrenderowany. Ja jednak nie zgadzam się z tą sugestią. Jak zapewne pamiętasz, w rozdziale 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript”, wspominałem o zasadach hermetyzacji i abstrakcji, a one nakazują, by działanie komponentu niekontrolowanego było nieznane dla komponentu nadrzędnego. Zasady te oznaczają także, że komponent nadrzędny nie kontroluje działania komponentu podrzędnego i nie powinien mieć takiej możliwości. Dlatego, kiedy będziemy starać się zapewnić, że komponent niekontrolowany będzie robił to, czego wymaga jego komponent nadrzędny, może nas kusić, by wyprowadzić w nim jakieś zmiany implementacyjne, które doprowadzą do jego ściślejszego powiązania z komponentem nadrzędnym. Choć czasami nie będzie można tego uniknąć, to jednak, jeśli tylko będzie to możliwe, powinniśmy wystrzegać się takich rozwiązań.
- Jeśli renderowany stan naszego komponentu zależy od danych zwracanych przez żądania sieciowe, to możemy je generować i obsługiwać zmiany stanu w metodzie `componentDidMount` (zakładając, że dane te będą potrzebne tylko raz, po ich pobraniu). Musisz pamiętać, że metoda `componentDidMount` jest wykonywana tylko raz, po pierwszym wczytaniu komponentu. Co więcej, zastosowanie tej metody spowoduje jedno, dodatkowe wyrenderowanie komponentu; choć to i tak będzie lepsze niż narażanie się na niepożądane operacje renderowania.

- Metoda `componentDidUpdate` może być także używana do obsługi scenariuszy, w których stan komponentu musi się zmieniać w efekcie zmian właściwości *props*. Ponieważ metoda ta jest wyświetlana po wyrenderowaniu komponentu, prawdopodobieństwo tego, że doprowadzi ona do ponownego renderowania jest mniejsze, o ile tylko przed wprowadzeniem zmian nie zapomnimy porównać właściwości *props* przekazanych do komponentu z jego stanem. Pomimo to, takiego „stanu pochodnego”<sup>1</sup> (ang. *derived state*) należy całkowicie unikać, o ile tylko będzie to możliwe, a cały stan przechowywać w jednym komponentcie głównym i udostępniać za pomocą właściwości *props*. Szczerze mówiąc, jest to zadanie dość uciążliwe, zwłaszcza jeśli musimy przekazywać stan przy użyciu właściwości *props* do komponentów podrzędnych położonych kilka poziomów niżej. Co więcej, oznacza to także, że będziemy musieli w odpowiedni i przemyślany sposób określić strukturę stanu, tak byśmy byli w stanie precyzyjnie wskazać stan powiązany z konkretnym komponentem podrzędnym. Kiedy w dalszej części książki zaczniemy korzystać z *hooków*, przekonasz się, jak bardzo ułatwiają one korzystanie ze stanu w porównaniu z rozwiązaniami korzystającymi z pojedynczego obiektu stanu. Niemniej jednak w aplikacjach Reacta, maksymalne ograniczanie lokalnego stanu komponentu jest zawsze właściwą praktyką.

Utwórzmy teraz niewielki projekt, który pozwoli nam wypróbować komponenty klasowe i przedstawić ich cechy:

1. W oknie wiersza poleceń lub panelu terminala przejdź do katalogu *rozdzial05*.
2. Wykonaj następujące polecenie:

```
npm create-react-app class-components --template typescript
```

3. Następnie wyświetl utworzony przed chwilą katalog, *class-components*, w edytorze VSCode; dodatkowo otwórz ten sam katalog w panelu terminala. Kiedy to zrobisz, w katalogu *src* utwórz nowy plik o nazwie *Greeting.tsx* i zapisz w nim następujący kod:

```
import React from "react";

interface GreetingProps {
  name?: string
}
interface GreetingState {
  message: string
}
export default class Greeting extends React.Component<GreetingProps,
↳GreetingState> {
  constructor(props: GreetingProps) {
    super(props);
    this.state = {
      message: `Witaj z Greeting, ${props.name}`
    }
  }
}
```

<sup>1</sup> Terminem tym określane jest wewnętrzny stan komponentu, który w mniejszym lub większym stopniu jest kontrolowany przez właściwości *props* — *przyp. tłum.*

```

    }

    state: GreetingState;

    render() {
      if(!this.props.name) {
        return <div>Nie podano imienia.</div>;
      }
      return <div>
        {this.state.message}
      </div>;
    }
  }
}

```

Kiedy przyjrzyysz się temu plikowi, w pierwszej kolejności zauważysz zapewne, że ma on rozszerzenie *.tsx*. Zastosowanie tego rozszerzenia jest konieczne w przypadku tworzenia komponentów Reacta w języku TypeScript. Zglądając do kodu pliku, na samym jego początku zauważysz instrukcję importującą React. Udostępnia nam ona nie tylko klasę Component, po której będzie dziedziczyć nasz komponent, lecz także możliwość korzystania ze składni JSX. Następnie definiujemy dwa interfejsy `GreetingProps` oraz `GreetingState`. Pamiętaj, że używamy języka TypeScript, więc zależy nam na wykorzystaniu mechanizmów kontroli typów. Dlatego definiujemy oba oczekiwane typy: jeden dla właściwości *props* przekazywanych do naszego komponentu i drugi dla jego stanu. Zwróć także uwagę na pole `name` tworzone w interfejsie `GreetingProps` — zadeklarowaliśmy je jako opcjonalne, co oznacza, że można mu także przypisać wartość `undefined` (co już niebawem wykorzystamy). Jeszcze raz podkreślam, że o ile to tylko możliwe, należy unikać dodawania stanu do komponentów, które nie są komponentem głównym ani komponentami nadrzędnymi. W tym przykładzie robię to jedynie w celach demonstracyjnych.

4. Tworząc klasę musimy także pamiętać, by ją wyeksportować, tak by była dostępna dla wszelkich innych komponentów, które chciałyby jej używać. Właśnie do tego celu służy słowo kluczowe **export**. Oprócz `export` zastosowaliśmy także słowo kluczowe **default**, które oznacza, że jest to główny element eksportowany z tego modułu; dzięki temu importując ten moduł w innych miejscach kodu nie będziemy musieli używać nawiasów klamrowych. Przykład pokazujący importowanie tego modułu przedstawię w dalszej części rozdziału. W sygnaturze definicji klasy widzimy, że nasz komponent dziedziczy po `React.Component<GreetingProps>`. Ta deklaracja typu informuje nie tylko o tym, że nasza klasa jest komponentem Reacta, lecz także, że pobiera ona właściwość *props* typu `GreetingProps`. Poniżej wiersza z deklaracją klasy definiujemy jej konstruktor, który pobiera jeden parametr, `prop`, typu `GreetingProps`.

Kiedy komponent używa właściwości *props*, bardzo ważne jest, by pierwszą operacją wykonaną w jego konstruktorze było wywołanie konstruktora klasy bazowej, `super` (→ `props`). To wywołanie zapewni, że React będzie wiedział o pobieranych właściwościach i będzie mógł prawidłowo reagować na ich zmiany. W kodzie konstruktora, odwołując się do obiektu `props` nie musimy używać wyrażenia `this.props`, gdyż obiekt ten został przekazany jako parametr. W pozostałym kodzie komponentu, w odwołaniach do właściwości **props** konieczne będzie stosowanie wyrażenia `this.props`.

5. Drugą operacją jest określenie wartości właściwości `state`. Wartość ta jest przypisywana w konstruktorze, natomiast sama właściwość, jak i jej typ, `GreetingState`, zostały zdefiniowane w wierszu poniżej konstruktora. Poniżej konstruktora zdefiniowaliśmy także funkcję `render`; deklaruje ona kod JSX, który docelowo zostanie przekształcony na kod HTML. Zwróć uwagę na to, że w kodzie tej funkcji umieściliśmy instrukcję warunkową `if`, która pozwala wyświetlać inny interfejs użytkownika w zależności od wartości właściwości `this.props.name`. Funkcja `render` powinna sprawdzać dane komponentu i nie wyświetlać niczego, jeśli nie jest to uzasadnione. Jeśli takie warunki będziemy stosowali konsekwentnie, mogą one poprawić wydajność działania aplikacji i zmniejszyć zużycie pamięci. W razie gdy nie trzeba będzie niczego wyświetlać, wystarczy, że funkcja `render` zwróci wynik `none` — React uznaje tę wartość za sygnał, że nie ma nic do wyświetlenia.
6. Teraz pozostaje nam jedynie zmodyfikować plik `App.tsx` i dodać do niego nasz komponent `Greeting.tsx`. A zatem, otwórz plik `App.tsx` i zmodyfikuj go zgodnie z poniższym przykładem:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Greeting from './Greeting';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <Greeting />
      </header>
    </div>
  );
}

export default App;
```

W pierwszej kolejności zwróć uwagę na to, że na początku pliku importujemy klasę `Greeting`. Ponieważ klasa ta jest domyślnym elementem eksportowanym przez moduł `Greeting.tsx` (zauważ, że w instrukcji importu nie trzeba podawać rozszerzenia `.tsx`), pomiędzy słowami kluczowymi `import` i `from` nie musimy zapisywać nawiasów klamrowych (`{}`). Gdyby jednak klasa `Greeting` nie była domyślnym eksportowanym elementem, na przykład gdybyśmy z tego modułu eksportowali kilka różnych elementów, to musielibyśmy użyć instrukcji importu o następującej składni: `import { Greeting } from './Greeting'`.

7. Jak widać w dalszej części pliku, fragment jego początkowego kodu JSX zastąpiliśmy komponentem `Greeting`. Zwróć uwagę na to, że w komponencie tym nie zapisaliśmy właściwości `name`. Zobaczmy teraz co się stanie, kiedy uruchomimy aplikację. W panelu terminala, z poziomu katalogu `class-components`, wykonaj następujące polecenie:

```
npm start
```

W efekcie, w przeglądarce powinieneś zobaczyć stronę przedstawioną na rysunku 5.2.



**Rysunek 5.2.** Aplikacja po pierwszym wyświetleniu

Komunikat ten został wyświetlony dlatego, że do komponentu `Greeting` nie przekazaliśmy właściwości o nazwie `name`. Jak mogliśmy się przekonać, wartość właściwości mogła pozostać nieokreślona, gdyż w definicji typu pola `name` użyliśmy znaku zapytania (?).

8. Wróćmy zatem do pliku `App.tsx` i określmy wartość `name` w komponencie `Greeting`. Zmodyfikuj komponent `Greeting` jak pokazałem na poniższym przykładzie:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Greeting from './Greeting';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <Greeting name="Dave Choi"/>
      </header>
    </div>
  );
}

export default App;
```

Zawróć uwagę na to, że w komponencie określiłem właściwość `name`, podając w niej swoje imię i nazwisko. Jeśli masz ochotę, to nic nie stoi na przeszkodzie, byś wpisał tam dowolne inne dane, na przykład swoje; kiedy to zrobisz, zapisz plik. React zawiera serwer testowy, który automatycznie aktualizuje aplikację, dlatego nowy kod powinien zostać automatycznie wyświetlony w przeglądarce. W efekcie wprowadzonych zmian, strona w przeglądarce powinna zostać zaktualizowana, jak pokazałem na rysunku 5.3.





Rysunek 5.3. Zaktualizowana strona aplikacji

No dobrze, przygotowaliśmy zatem prosty komponent klasowy. Teraz możemy zacząć korzystać w nim z wybranych metod cyklu życia, aby przekonać się, jak one działają:

1. Do kodu w pliku *Greeting.tsx* dodaj definicję funkcji `getDerivedStateFromProps`:

```
import React from "react";

interface GreetingProps {
  name?: string
}
interface GreetingState {
  message: string
}

export default class Greeting extends React.Component<GreetingProps,
↳GreetingState> {
  constructor(props: GreetingProps) {
    super(props);
    this.state = {
      message: `Witaj z Greeting, ${props.name}`
    }
  }

  state: GreetingState;
```

2. Ta pierwsza część kodu jest praktycznie taka sama jak wcześniej, przy czym poniżej niej i bezpośrednio przed funkcją `render` dodamy definicję funkcji `getDerivedStateFromProps`:

```
  static getDerivedStateFromProps(props: GreetingProps, state:
↳GreetingState) {
    console.log(props, state);
    return state;
  }

  render() {
    console.log("Renderuję komponent Greeting")
```

```

    if(!this.props.name) {
      return <div>Nie podano imienia.</div>;
    }
    return <div>
      {this.state.message}
    </div>;
  }
}

```

Jak widać, jest to funkcja statyczna, czyli skojarzona z klasą, a nie z instancją tej klasy. Funkcja ta przyjmuje bieżące wartości właściwości *props* oraz stanu i przekazuje je do komponentu, dzięki czemu ten, w razie konieczności, może na ich podstawie zaktualizować swój stan. Jak widać na przykładzie, my tego stanu na razie nie aktualizujemy. W naszym przypadku działanie tej funkcji ogranicza się do wyświetlenia na konsoli właściwości *props* i stanu, a także zwrócenia stanu (ta funkcja zawsze musi zwracać obiekt stanu, niezależnie od tego, czy został on zmodyfikowany, czy nie). Zwróć także uwagę na to, że aktualnie funkcja render wyświetla na konsoli komunikat informujący o jej wywołaniu.

3. Zostawmy na razie ten kod w takiej postaci, w jakiej jest, i zmodyfikujmy plik *App.tsx* w taki sposób, by zapewniał użytkownikowi możliwość podania swojego imienia:

```

import React from 'react';
import logo from './logo.svg';
import './App.css';
import Greeting from './Greeting';

class App extends React.Component {
  constructor(props:any) {
    super(props);

    this.state = {
      enteredName: ""
    }
    this.onChangeName = this.onChangeName.bind(this);
  }

  state: { enteredName: string }

  onChangeName(e: React.ChangeEvent<HTMLInputElement>) {
    this.setState({
      enteredName: e.target.value
    });
  }
}

```

Aby pobrać dane wejściowe wprowadzone przez użytkownika i zapisać je, by można ich było użyć w przyszłości, tworzymy obiekt *state* zawierający jedno pole: *enteredName*. Tworzymy także nową funkcję o nazwie *onChangeName* i przy użyciu funkcji *bind* kojarzymy ją z bieżącą instancją klasy reprezentowaną przez *this* (technikę tę opisałem w rozdziale 3. pt. „Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript”).

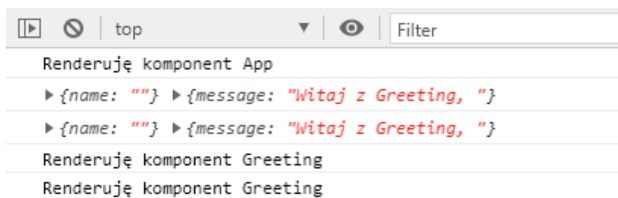
4. W kodzie funkcji `onChangeName` imię wpisane przez użytkownika zapisujemy we właściwości `enteredName` obiektu `state`. Robimy to przy użyciu funkcji `setState`. W komponentach klasowych Reacta pod żadnym pozorem nie można modyfikować ich stanu bez wykorzystania funkcji `setState`, gdyż w takim przypadku stan nie będzie synchronizowany ze środowiskiem wykonawczym Reacta:

```
render() {
  console.log("Renderuję komponent App");

  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <input value={this.state.enteredName} onChange={this.onChangeName} />
        <Greeting name={this.state.enteredName} />
      </header>
    </div>
  )
}

export default App;
```

5. Jak widać, do kodu funkcji `render` w pliku `App.tsx` dodaliśmy wywołanie `console.log`, które umożliwi nam śledzenie, ile razy funkcja ta została wywołana. Oprócz tego dodaliśmy pole tekstowe, `input`, którego wartością jest wyrażenie `this.state.enteredName` i w którym zdarzenie `onChange` skojarzyliśmy ze zdefiniowaną w komponencie funkcją `onChangeName`. Jeśli teraz zapiszesz ten plik i otworzysz narzędzia dla programistów przeglądarki Chrome, zobaczysz w nich komunikaty takie jak te przedstawione na rysunku 5.4.



Rysunek 5.4. Renderowanie komponentu Greeting

Na rysunku 5.4 widoczne są komunikaty z renderowania obu komponentów, jak również wartości właściwości `props` `name` i właściwości stanu `message` komponentu `Greeting`. Póki co nie wpisaliśmy jeszcze żadnego imienia w polu tekstowym, dlatego właściwość `props` `name` komponentu `Greeting` oraz końcówka właściwości `message` z komunikatem powitalnym są puste. Zastanawiasz się zapewne, dlaczego komunikaty z komponentu `Greeting` są wyświetlane podwójnie. Otóż wynika to z faktu, że aplikacja pracuje w trybie ścisłym (ang. *StrictMode*), służącym do prowadzenia prac programistycznych.

6. Poprawmy to, by powielane komentarze nie wprowadzały nas w błąd. W edytorze VSCode otwórz plik *index.tsx* i zmodyfikuj jego zawartość, jak pokazałem na poniższym przykładzie:

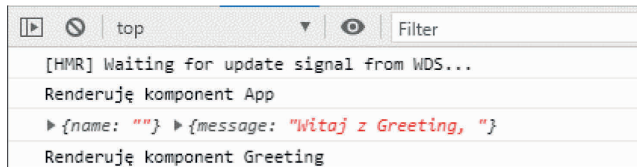
```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

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

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Zastąpiliśmy komponent `StrictMode` komponentem `Fragment`. Tak naprawdę nie potrzebujemy tu komponentu `Fragment`, gdyż jest on używany jako pojemnik na kod JSX, który nie ma żadnego elementu nadrzędnego, takiego jak `div`, jednak na nasze potrzeby takie rozwiązanie jest całkiem dobre, a ja chciałem zostawić w kodzie miejsce na ponowne umieszczenie znaczników `StrictMode`.

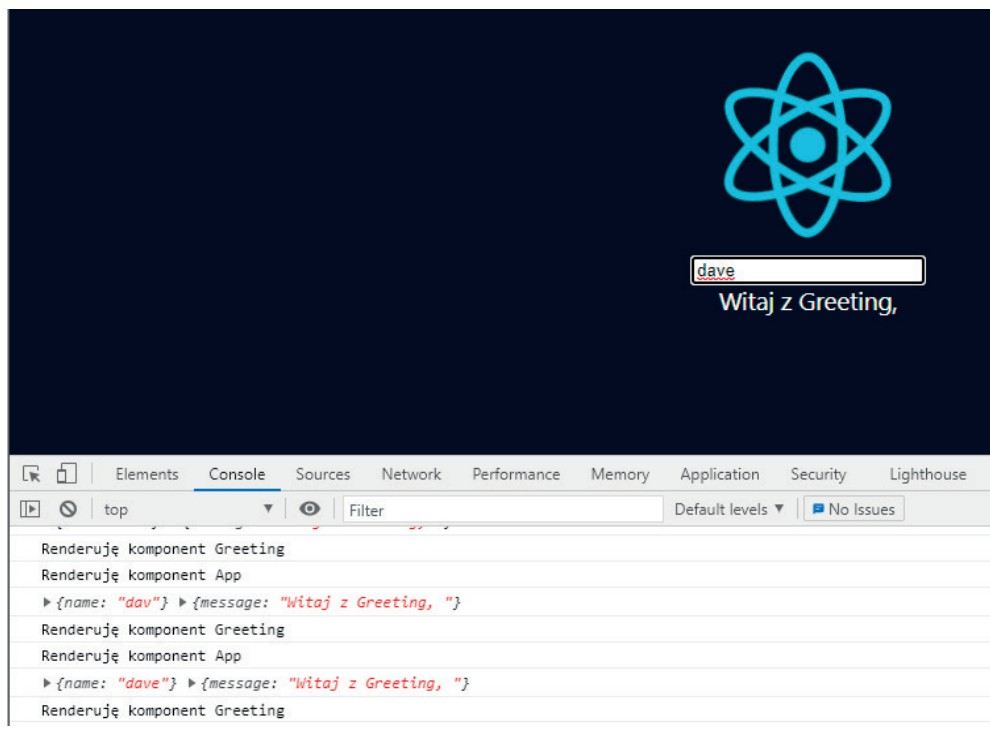
7. Jeśli teraz zapiszesz plik i zajrzysz do konsoli w narzędziach dla programistów w przeglądarce, to zobaczysz komunikaty takie jak te, które przedstawiłem na rysunku 5.5.



Rysunek 5.5. Konsola narzędzi dla programistów przeglądarki

Ten przykład ma za zadanie pokazać, co może doprowadzić do renderowania komponentu oraz w jaki sposób ostrożniej podchodzić do tego zagadnienia.

8. A teraz, jeśli spróbujesz wpisać imię w polu tekstowym, wyświetlone zostaną komunikaty takie jak te widoczne na rysunku 5.6.
9. Pozostaje jednak pytanie, dlaczego łańcuch powitania wciąż ma postać „Witaj z Greeting, ”? Jeśli zajrzysz do kodu komponentu `Greeting`, przekonasz się, że wartość właściwości stanu `message` jest określana tylko raz — w konstruktorze komponentu (co właściwie przypomina zastosowanie metody `componentDidMount`). A zatem, skoro to zdarzenie następuje tylko raz, podczas wczytywania pierwszej strony aplikacji, właściwość `this.props.name` jest w tym momencie pusta, gdyż jeszcze niczego nie zdążyliśmy wpisać w polu tekstowym. I co możemy zrobić z tym fantem? Cóż, spróbujmy użyć funkcji `getDerivedStateFromProps` i zobaczymy, co się stanie:



Rysunek 5.6. Wprowadzanie danych wejściowych na stronie App.tsx

```
export default class Greeting extends React.Component<GreetingProps,
  ↳GreetingState> {
  constructor(props: GreetingProps) {
    super(props);

    this.state = {
      message: Greeting.getNewMessage(props.name)
    }
  }

  state: GreetingState;
```

10. Pokazuję tu jedynie kod klasy Greeting, gdyż tylko on zmienia się w tym przykładzie. Zwróć uwagę na nowy kod funkcji `getDerivedStateFromProps`:

```
static getDerivedStateFromProps(props: GreetingProps, state:
  ↳GreetingState) {
  console.log(props, state);
  if(props.name && props.name !== state.message) {
    const newState = {...state};
    newState.message = Greeting.getNewMessage(props.name);
    return newState;
  }
  return state;
}
```

```

    static getNewMessage(name: string = "") {
        return `Witaj z Greeting, ${name}`;
    }

    render() {
        console.log("Renderuję komponent Greeting")
        if(!this.props.name) {
            return <div>Nie podano imienia.</div>;
        }
        return <div>
            {this.state.message}
        </div>;
    }
}

```

Jak widać, w nowej postaci funkcja `getDerivedStateFromProps` stała się bardziej złożona, między innymi dlatego, że porównujemy w niej nowe wartości właściwości *props* z bieżącym stanem komponentu. Następnie klonujemy przekazany obiekt `state`. Koniecznie musisz się upewnić, że będziesz stosować właśnie takie rozwiązanie, żeby przez przypadek nie spowodować bezpośredniej zmiany stanu. Następnie aktualizujemy wartość `state.message`, używając do tego celu nowej, statycznej funkcji `getNewMessage` (zdefiniowałem tę funkcję dlatego, że komunikat jest ustawiany w kilku miejscach komponentu). A teraz spróbuj wpisać w polu tekstowym na stronie swoje imię. Kiedy to zrobisz i spojrzysz na narzędzia dla programistów w przeglądarce, przekonasz się, że wpisane imię faktycznie zostało dodane do komunikatu wyświetlanego przez komponent, jednak komponenty `Greeting` i `App` są renderowane po wpisaniu każdej litery imienia. Póki co, takie działanie aplikacji nie jest niczym strasznym, gdyż jej kod jest dość prosty, jednak gdybyśmy zaczęli dodawać kolejne właściwości do stanu komponentu `Greeting` i gdyby nasza aplikacja była znacznie bardziej rozbudowana, to sprawy mogłyby się znacząco skomplikować.

Spróbujmy zatem przeprowadzić refaktoryzację kodu i przekonajmy się, czy uda się nam nieco poprawić sytuację:

#### 1. Zaktualizuj plik *App.tsx*:

```

class App extends React.Component {
    constructor(props:any) {
        super(props);

        this.state = {
            enteredName: "",
            message: ""
        }

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

    state: { enteredName: string, message: string }

    onChangeName(e: React.ChangeEvent<HTMLInputElement>) {
        this.setState({
            enteredName: e.target.value,

```

```

        message: `Witaj z App, ${e.target.value}`
    });
}

```

Pokazuję tu jedyne kod klasy App, gdyż tylko on uległ zmianie. Jak widać, do obiektu stanu, state, dodaliśmy nową właściwość o nazwie message (już niebawem usuniemy właściwość o tej samej nazwie z komponentu Greeting), którą zmieniamy za każdym razem, kiedy użytkownik wpisze nowe imię w polu input:

```

render() {
    console.log("Renderuję komponent App");

    return (
        <div className="App">
            <header className="App-header">
                <img src={logo} className="App-logo" alt="logo" />
                <input value={this.state.enteredName} onChange={this.onChangeName} />
                <Greeting message={this.state.message} />
            </header>
        </div>
    )
}

```

Następnie przekazujemy właściwość stanu message do komponentu Greeting jako właściwość props.

2. Teraz znowu będziemy zajmować się komponentem Greeting, jednak, aby uprościć sobie nieco sprawę, przygotujemy nowy komponent — utwórz nowy plik o nazwie *GreetingFunctional.tsx* i umieść w nim poniższy kod:

```

import React from "react";

interface GreetingProps {
    message: string
}

export default function Greeting(props: GreetingProps) {
    console.log("Renderuję komponent Greeting")

    return (<div>
        {props.message}
    </div>);
}

```

3. Kiedy już dodasz ten plik, w pliku *App.tsx* zmień instrukcję, która importuje komponent Greeting:

```
import Greeting from "../GreetingFunctional";
```

Jak widać, ta wersja komponentu Greeting jest znacznie krótsza i prostsza od poprzednich. Tym razem mamy jednak do czynienia z komponentem funkcyjnym (ang. *functional component*), gdyż najlepsze praktyki tworzenia aplikacji Reacta zalecają, by komponenty, które nie mają własnego, lokalnego stanu, były implementowane jako funkcje, a nie klasy. Nie byliśmy w stanie wyeliminować ponownego renderowania komponentu, gdyż zmiana komunikatu

nieodwołalnie będzie je powodować, jednak nawet pomimo to skrócenie i uproszczenie kodu jest warte zachodu. Co więcej, choć przenieśliśmy część kodu do pliku *App.tsx*, to na pewno zauważysz, że nawet te fragmenty stały się mniej złożone niż były we wcześniejszej wersji komponentu *Greeting*.

Jednak taki styl tworzenia komponentów, polegający na umieszczaniu całego stanu w jednym komponencie nadrzędnym i przekazywaniu tego stanu do komponentów podrzędnych przy użyciu właściwości *props*, wiąże się z pewnym problemem. Otóż przekazywanie właściwości *props* w dół wielopoziomowych, złożonych hierarchii komponentów, może wymagać zastosowania dość rozbudowanego, dodatkowego kodu. To właśnie w takich sytuacjach moglibyśmy skorzystać z kontekstu Reacta (React Context), by pominąć hierarchię i przekazywać stan komponentu nadrzędnego bezpośrednio do wybranego komponentu podrzędnego. Ja jednak nie lubię korzystać z tego rozwiązania, gdyż uważam, że pomijanie naturalnej hierarchii i wstrzykiwanie stanu do wybranego komponentu jest antywzorcem (czyli projektem, którego należy unikać). Takie rozwiązania na pewno będą wprowadzać zamieszanie i utrudniać późniejszą refaktoryzację kodu. Niemniej jednak zagadnienia związane z kontekstem Reacta — React Context — zostały opisane w rozdziale 7., pt. „Redux i React Router”.

W tym podrozdziale przedstawiłem komponenty klasowe Reacta. Ponieważ *hooki* wciąż są stosunkowo nowym rozwiązaniem, a większość aplikacji Reacta cały czas używa komponentów klasowych, dlatego zrozumienie i opanowanie tego stylu programowania wciąż jest ważne. W następnym podrozdziale zajmiemy się tworzeniem komponentów korzystających z *hooków*, a na samym końcu porównamy oba te style.

## Prezentacja hooków Reacta i wyjaśnienie, dlaczego w stosunku do komponentów klasowych są one usprawnieniem

Z tego podrozdziału dowiesz się, czym są tak zwane *hooki* Reacta (ang. *React Hooks*). Utworzymy przykładowy projekt i na jego przykładzie zobaczymy, jak działają *hooki*. Ponieważ ta książka dotyczy w głównej mierze właśnie tego stylu programowania, przynajmniej jeśli chodzi o tworzenie aplikacji Reacta, dlatego zamieszczone tu informacje ułatwią nam pisanie kodu w dalszej części książki.

W pierwszej kolejności przyjrzymy się przyczynom wprowadzenia *hooków*. Z ostatniego podrozdziału w poprzednim rozdziale, poświęconego komponentom klasowym, dowiedziałeś się, że dysponują one metodami cyklu życia, pozwalającymi na obsługę pewnych zdarzeń zachodzących w czasie istnienia komponentu. W przypadku stosowania *hooków* te metody cyklu życia nie są dostępne, gdyż komponenty, które z nich korzystają, są komponentami funkcyjnymi. W poprzednim przykładzie, demonstrującym komponenty klasowe, stworzyliśmy już jeden komponent funkcyjny — *GreetingFunctional*. Komponent funkcyjny to komponent, który jest funkcją języka JavaScript zwracającą kod JSX. Powodem wprowadzenia tego nowego rodzaju komponentów Reacta jest próba odejścia w całym projekcie



aplikacji od mechanizmu dziedziczenia, stanowiącego jedną z podstaw **modelu programowania obiektowego (OOP)** i zagwarantowanie możliwości wielokrotnego stosowania kodu dzięki wykorzystaniu kompozycji. O modelu dziedziczenia pisałem już w rozdziale 2., pt. „Prezentacja języka TypeScript”, natomiast kompozycja oznacza, że zamiast otrzymywać możliwości funkcjonalne w drodze dziedziczenia po klasie nadrzędnej, będziemy je uzyskiwać poprzez łączenie komponentów funkcyjnych używanych do tworzenia ekranów aplikacji.

Tym komponentom funkcyjnym towarzyszy możliwość stosowania *hooków*. *Hooki* to nie innego, jak zwyczajne funkcje JavaScriptu, które zapewniają komponentom pewne możliwości. Chodzi tu o takie możliwości, jak utworzenie stanu, dostęp do danych pobieranych przez internet czy też jakiegokolwiek inne możliwości, których komponent mógłby potrzebować. Co więcej, *hooki* nie są charakterystyczne dla żadnego konkretnego komponentu, dzięki czemu można ich używać w dowolnych komponentach — zakładając oczywiście, że będzie to użyteczne i sensowne. Jeśli przyjrzymy się przedstawionym wcześniej komponentom klasowym, to okaże się, że nie zapewniają one możliwości współdzielenia logiki w swoich metodach cyklu życia. Nie można w prosty sposób wyodrębnić tej logiki i zastosować w jakimś innym komponencie klasowym. I właśnie to ograniczenie stanowiło jeden z podstawowych powodów utworzenia we frameworku React modelu *hooków*. Dlatego też te dwa podstawowe elementy — komponenty funkcyjne oraz funkcje nadające się do wielokrotnego stosowania (czyli *hooki*) — są kluczowe dla zrozumienia tego nowego modelu tworzenia aplikacji Reacta.

Zacznijmy od przedstawienia wybranych, najważniejszych *hooków*, których będziemy używali w naszym kodzie. Przykłady ich zastosowania przedstawię już niebawem, a na razie tylko ogólnie je opiszę:

- **useState**. Ta funkcja stanowi absolutną podstawę programowania z wykorzystaniem *hooków*. Zastępuje ona właściwość `state` oraz funkcję `setState`, stosowane w komponentach klasowych. Funkcja `useState` ma jeden parametr, którego wartość reprezentuje początkową wartość właściwości stanu. Wywołanie tej funkcji zwraca tablicę. Pierwszym elementem tej tablicy jest faktyczna właściwość stanu, a drugą — funkcja, której będzie można używać do modyfikowania wartości tej właściwości. Ogólnie rzecz biorąc, funkcja ta jest używana do modyfikowania pojedynczej wartości, a nie bardziej złożonych obiektów zawierających wiele właściwości. W przypadku korzystania z takiego bardziej złożonego stanu znacznie lepszym rozwiązaniem będzie użycie *hooka* `useReducer`.
- **useEffect**. Ta funkcja jest wywoływana, kiedy zostanie zakończone wyświetlanie komponentu na ekranie. Przypomina ona nieco metody cyklu życia `componentDidMount` oraz `componentDidUpdate`. Jednak one są wywoływane jeszcze zanim komponent zacznie być wyświetlany na ekranie. Funkcja `useEffect` służy do aktualizowania obiektów stanu. A zatem, jeśli na przykład musimy pobrać jakieś dane z internetu, a następnie, po ich odebraniu, zaktualizować stan komponentu, to możemy to zrobić używając właśnie tego *hooka*. Można jej także używać do subskrybowania zdarzeń, lecz w takim przypadku trzeba takie subskrypcje anulować poprzez zwrócenie odpowiedniej funkcji anulującej subskrypcję.

Istnieje możliwość przygotowania większej liczby implementacji funkcji `useEffect`, z których każda będzie wykonywała jakieś unikalne operacje. Zazwyczaj jest ona wywoływana po każdej zakończonej operacji odświeżania ekranu. A zatem, jeśli zmieni się jakakolwiek wartość stanu komponentu lub jego właściwości *props*, spowoduje to wywołanie tej funkcji. Można wymusić, by funkcja ta została wykonana tylko raz, tak jak dzieje się z metodą cyklu życia `componentDidMount`; wystarczy przekazać w jej wywołaniu pustą tablicę. Można także wymusić, by funkcja ta była wywoływana wyłącznie w przypadku zmian konkretnych właściwości *props* lub właściwości stanu; w tym przypadku trzeba je przekazać w formie tablic w wywołaniu `useEffect`.

Ta funkcja jest wywoływana asynchronicznie, ale jeśli będziemy musieli uzyskać jakieś informacje dotyczące prezentacji komponentu na ekranie, takie jak przesunięcie paska przewijania (ang. *scroll position*), to będziemy mogli skorzystać z funkcji `useLayoutEffect`. Także ta funkcja jest wywoływana asynchronicznie i pozwala pobierać określone wartości związane z prezentacją komponentu na ekranie, a następnie wykonywać na nich jakieś operacje w sposób synchroniczny. Oczywiście, takie działania blokują interfejs użytkownika, więc można ich używać wyłącznie do wykonywania operacji bardzo szybkich; w przeciwnym razie możemy doprowadzić do pogorszenia doznań użytkowników korzystających z aplikacji.

- `useCallback`. Ta funkcja tworzy instancję funkcji po wprowadzeniu zmian w pewnym zestawie parametrów. Funkcja ta pozwala na oszczędzanie pamięci; bez niej nowa instancja funkcji byłaby tworzona podczas każdej operacji renderowania. Jej pierwszym parametrem jest funkcja obsługi, a drugim — tablica elementów, które mogą się zmieniać. Jeśli elementy nie ulegną zmianie, do funkcji zwrotnej nie jest przekazywana nowa instancja. Dlatego wszelkie właściwości używane w kodzie tej funkcji będą miały wcześniejsze wartości. Kiedy po raz pierwszy czytałem o tej funkcji, uznałem, że jest ona niejasna i trudna do zrozumienia; dlatego też w dalszej części rozdziału przedstawię przykład jej użycia.
- `useMemo`. Ta funkcja służy do zapisywania wyników długotrwałych operacji. Przypomina ona nieco przechowywanie danych w pamięci podręcznej, jednak jest wykonywana wyłącznie w razie zmiany wartości jednego z parametrów wskazanych w tablicy; w pewnym sensie przypomina ona zatem funkcję `useCallback`. Jednak funkcja `useMemo` zwraca wartość stanowiącą wynik jakichś długotrwałych obliczeń.
- `useReducer`. Ta funkcja działa podobnie jak magazyn React Redux. Pobiera ona dwa parametry, reduktor (ang. *reducer*) oraz stan początkowy, zwraca dwa elementy: obiekt stanu, który będzie modyfikowany przez reduktor oraz funkcję dyspozytora (ang. *dispatcher*), która będzie otrzymywać zmodyfikowane dane stanu (określane jako *akcja*, ang. *action*) i przekazywać je do reduktora. Reduktor działa jako swoisty mechanizm filtrujący, określający w jaki sposób dane akcji zostaną użyte do zmodyfikowania stanu. W dalszej części rozdziału przedstawię przykład zastosowania tej funkcji. Z powodzeniem nadaje się ona do użycia w sytuacjach, kiedy będziemy dysponować jednym, złożonym obiektem stanu, zawierającym wiele właściwości, które będą mogły być modyfikowane.

- `useContext`. Ta funkcja zapewnia możliwość posługiwania się jednymi, globalnymi danymi stanu, które mogą być współużytkowane w różnych komponentach. Lepiej będzie używać jej sporadycznie, gdyż zapewnia ona możliwość wstrzykiwania kodu do dowolnych komponentów podrzędnych bez względu na ich położenie w hierarchii. W tej książce będziemy raczej korzystać z `Reduxa`, a nie z tej funkcji i zapewnianych przez nią możliwości, jednak warto wiedzieć, że ona istnieje.
- `useRef`. Tej funkcji można używać do zapisania dowolnej wartości we właściwości `current` zwracanego obiektu. Zmiany takich wartości nie powodują ponownego renderowania komponentu i istnieją tak długo, jak długo będzie istnieć komponent, który je utworzył. Zapewnia ona możliwość przechowywania stanu, którego zmiany nie będą powodowały ponownego renderowania komponentu. Jednym z zastosowań tej funkcji jest przechowywanie elementów DOM. Może się zdarzyć, że będziemy chcieli skorzystać z takiego rozwiązania, gdyż w niektórych okolicznościach trzeba porzucić standardowy, bazujący na zmianach stanu model programowania aplikacji Reacta i skorzystać z bezpośredniego dostępu do elementów HTML. W takich sytuacjach możemy używać funkcji `useRef`, by odwoływać się do instancji elementów.

Oczywiście dostępnych jest znacznie więcej *hooków*, przygotowanych zarówno przez twórców Reacta, jak i przez innych programistów. Kiedy nabierzesz wprawy w posługiwaniu się nimi, będziesz w stanie określić, czego Ci potrzeba, a może nawet lepiej: będziesz w stanie samemu tworzyć własne *hooki*. W projektach przedstawionych w dalszej części książki także będziemy tworzyć i stosować własne *hooki*.

Przyjrzymy się teraz kilku przykładom zastosowania *hooków*. Zacznij od utworzenia nowego katalogu o nazwie *rozdzial05*, a następnie wykonaj czynności opisane na poniższej liście:

1. W oknie wiersza poleceń lub w panelu terminala w VSCode przejdź do katalogu *rozdzial05* i wykonaj następujące polecenie:

```
npm create-react-app hooks-components --template typescript
```

2. W poprzednim przykładzie, w którym prezentowałem komponenty klasowe, utworzyliśmy komponent klasowy o nazwie *Greeting.tsx*. Komponent ten posiadał własny stan. W celach demonstracyjnych utwórzmy taki sam komponent, lecz jako komponent funkcyjny korzystający z *hooków*. A zatem, w katalogu *src* projektu *hooks-components* utwórz nowy plik o nazwie *Greeting.tsx* i zapisz w nim następujący kod:

```
import React, { FC, useState, useEffect } from 'react';

interface GreetingProps {
  name?: string
}

const Greeting: FC<GreetingProps> = ({name}:GreetingProps) => {
  const [message, setMessage] = useState("");

  useEffect(() => {
    if(name) {
```

```

        setMessage(`Witaj z Greeting, ${name}`);
    }, [name])

    if(!name) {
        return <div>Nie podano imienia</div>;
    }
    return <div>
        {message}
    </div>;
}

export default Greeting;

```

Ten komponent dysponuje własnym stanem i pobiera właściwość *props* o nazwie *name*. Powinniśmy starać się unikać stosowania lokalnego stanu, jednak tutaj używam go w celach demonstracyjnych. Jak widać, ten komponent jest znacząco krótszy od analogicznego komponentu klasowego. Co więcej, nie ma w nim żadnych metod cyklu życia, które musielibyśmy przesyłać. Komponent zaimplementowałem jako funkcję strzałkową, gdyż mają one krótszą składnię, a oprócz tego nie potrzebujemy możliwości, jakie zapewniają normalne funkcje. Jak widać, kod zawiera deklarację komponentu *Greeting*. Zastosowaliśmy w niej typ *FC*, co stanowi skrót od angielskich słów **Functional Component** (komponent funkcyjny). *FC* to typ generyczny, który kojarzymy z interfejsem *GreetingProps*. Stan komponentu będzie zapisywany we właściwości *message*; aby było to możliwe, wywołujemy funkcję *useState*. Zwróć uwagę na to, że cała ta operacja jest wykonywana w jednym wierszu kodu, który nie jest umieszczony w żadnym konstruktorze; co jest zrozumiałe, gdyż mamy tu do czynienia z funkcją, a nie klasą. Zwróć także uwagę na to, że określenie typu parametru, *GreetingProps*, nie jest konieczne, jednak podałem go tutaj, by kod był bardziej zrozumiały i kompletny. I w końcu zauważ, że zamiast *props* zastosowałem zapis *{ name }*, czyli dokonałem destrukuryzacji przekazywanych do komponentu właściwości *props*.

Wewnątrz komponentu *Greeting* użyliśmy funkcji *useEffect*. Jak już wcześniej zaznaczyłem, przypomina ona nieco metody *componentDidMount* oraz *componentWillUnmount*, lecz jest wywoływana po zakończeniu wyświetlania komponentu na ekranie. W naszym przypadku funkcja ta zaktualizuje właściwość stanu *message* po każdej zmianie wartości właściwości *props* *name*, gdyż właśnie ta właściwość została przekazana jako ostatni argument wywołania funkcji. Ponieważ nasz komponent nie jest komponentem klasowym, nie ma także funkcji *render*. To wartość zwrócona przez tę funkcję stanowi żądanie do wyrenderowania komponentu.

3. Teraz dokonamy nieznacznej refaktoryzacji naszego kodu, a konkretnie: przeniesiemy stan do komponentu nadrzędnego, *App.tsx*. Zaczynij od utworzenia nowego komponentu, *GreetingFunctional.tsx*, który będzie taki sam, jak w poprzednim projekcie, prezentującym komponenty klasowe:

```

import React from "react";

interface GreetingProps {
    message: string

```

```

    }

    export default function Greeting(props: GreetingProps) {
      console.log("Renderuję komponent Greeting")

      return (<div>
        {props.message}
      </div>);
    }

```

4. Teraz przeprowadzimy drobną refaktoryzację komponentu *App.tsx*: nadamy mu postać charakterystyczną dla komponentów funkcyjnych i zastosujemy poznany wcześniej *hook* `useReducer`. W poniższym kodzie pominąłem instrukcje importu, gdyż nic się w nich nie zmieniło:

```

const reducer = (state: any, action: any) => {
  console.log("enteredNameReducer");
  switch(action.type) {
    case "enteredName":
      if(state.enteredName === action.payload) {
        return state;
      }
      return { ...state, enteredName: action.payload }
    case "message":
      return { ...state, message: `Witaj, ${action.payload}` }
    default:
      throw new Error("Nieprawidłowy typ akcji: " + action.type);
  }
}

const initialState = {
  enteredName: "",
  message: "",
};

```

Pierwsza część tego fragmentu kodu zawiera definicję reduktora, a druga określa początkową wartość obiektu stanu, `initialState`. Domyślna sygnatura funkcji reduktorów zawiera dwa parametry typu `any`, co jest konieczne, gdyż zarówno obiekty stanu, jak i obiekty akcji mogą być zupełnie dowolnymi obiektami, przynajmniej z technicznego punktu widzenia. Jeśli przyjrzyś się funkcji `reducer`, zauważysz, że stara się ona obsługiwać różne typy akcji, zwracając nowy obiekt stanu o odpowiedniej postaci i zawierający zmodyfikowaną wartość odpowiedniej właściwości (niezwykle ważne jest, by nie modyfikować dotychczasowego obiektu stanu bezpośrednio — najpierw należy go skopiować, następnie w tym skopiowanym obiekcie zmodyfikować odpowiednią właściwość i w końcu zwrócić ten nowy obiekt). Tak właśnie wygląda oczekiwany sposób korzystania z funkcji `useReducer`. Możesz ją sobie wyobrażać jako miejsce do umieszczania logiki związanej z obiektem stanu. W kolejnym fragmencie kodu przedstawiłem wywołanie metody `useReducer` w komponencie `App`:

```

function App() {
  const [{ message, enteredName }, dispatch] =
    useReducer(reducer, initialState);

```

```

const onChangeName = (e: React.ChangeEvent<HTMLInputElement>) => {
  dispatch({ type: "enteredName", payload: e.target.value });
  dispatch({ type: "message", payload: e.target.value });
}

return (
  <div className="App">
    <header className="App-header">
      <img src={logo} className="App-logo" alt="logo" />
      <input value={enteredName} onChange={onChangeName} />
      <Greeting message={message} />
    </header>
  </div>
)
}

export default App;

```

Jak widać, funkcja `useReducer` zwraca obiekt oraz funkcję `dispatch`. Obiekt zwracany w efekcie wykonania funkcji reduktora jest kompletnym obiektem stanu, choć w tym przykładzie przeprowadzamy jego destrukuryzację, aby bezpośrednio uzyskać dwie właściwości: `message` oraz `enteredName`. Po tych przygotowaniach definiujemy funkcję obsługi zdarzeń, `onChangeName`, a wewnątrz niej wywołujemy zwróconą przez `useReducer` funkcję `dispatch`, która inicjuje faktyczną zmianę stanu, przekazując w tym celu odpowiednią akcję i wartość. Jeśli teraz uruchomisz ten kod, będzie on działał tak, jak w poprzednim przykładzie.

Najlepszym aspektem tego rozwiązania jest to, że — jak łatwo zauważyć — możemy wciąż funkcję reduktora i zastosować ją ponownie w zupełnie innym komponencie funkcyjnym. Możemy także przekazać funkcję `dispatch` do komponentów podrzędnych, by to one inicjowały zmiany stanu. Spróbujmy zaimplementować takie rozwiązanie:

1. Zmodyfikuj zawartość pliku *GreetingFunctional.tsx* tak, by odpowiadał kodowi przedstawionemu poniżej:

```

import React from "react";

interface GreetingProps {
  enteredName: string;
  message: string;
  greetingDispatcher: React.Dispatch<{ type: string, payload: string }>;
}

export default function Greeting(props: GreetingProps) {
  console.log("Renderuję komponent Greeting")

  const onChangeName = (e: React.ChangeEvent<HTMLInputElement>) => {
    props.greetingDispatcher({ type: "enteredName", payload: e.target.value
    ↵});
    props.greetingDispatcher({ type: "message", payload: e.target.value });
  }

  return <div>

```

```

        <input value={props.enteredName} onChange={onChangeName} />
      <div>
        {props.message}
      </div>
    </div>);
  }

```

Jak widać, teraz przekazujemy do komponentu `Greeting` `enteredName` oraz `greetingDispatcher` jako właściwości *props*. Oprócz tego do komponentu przenieśliśmy pole tekstowe do wprowadzania imienia oraz funkcję obsługi zdarzeń, `onChangeName`; będziemy teraz używali ich wewnątrz komponentu `Greeting`.

## 2. Teraz zaktualizuj kod w pliku *App.tsx*:

```

function App() {
  const [{ message, enteredName }, dispatch] =
    useReducer(reducer, initialState);

  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />

        <Greeting
          message={message}
          enteredName={enteredName}
          greetingDispatcher={ dispatch } />
      </header>
    </div>
  )
}

```

Jak widać, z kodu komponentu `App` usunęliśmy funkcję `onChangeName` oraz pole tekstowe, dzięki czemu będziemy mogli ich używać w komponencie `Greeting`, w pliku *GreetingFunctional.tsx*. Oprócz tego, teraz do komponentu `Greeting` przekazujemy jako parametry trzy elementy: wartości `enteredName` i `message` oraz funkcję `dispatch`. Kiedy uruchomisz ten kod, zauważysz, że tym razem to nasz komponent podrzędny z pliku *GreetingFunctional.tsx* inicjuje wywołania reduktora (funkcji `reducer`).

## 3. A teraz spróbujmy zastosować funkcję `useCallback`. W tym celu zmodyfikuj kod pliku *App.tsx* zgodnie z poniższym przykładem:

```

function App() {
  const [{ message, enteredName }, dispatch] =
    useReducer(reducer, initialState);

  const [startCount, setStartCount] = useState(0);
  const [count, setCount] = useState(0);

  const setCountCallback = useCallback(() => {
    const inc = count + 1 > startCount ? count + 1 : Number(count + 1) +
      ↳startCount;
    setCount(inc);
  }, [startCount]);
}

```

```

    }, [count, startCount]);

    const onWelcomeBtnClick = () => {
        setCountCallback();
    }

    const onChangeStartCount = (e: React.ChangeEvent<HTMLInputElement>) => {
        setStartCount(Number(e.target.value));
    }

```

Zmiana, którą wprowadzamy w tej wersji komponentu, ma zapewnić użytkownikowi możliwość podania jakiejś liczby początkowej, która zostanie zapisana jako `startCount`. Ta liczba będzie następnie inkrementowana przy użyciu przycisku, którego kliknięcie będzie powodować wywołanie funkcji `setCountCallback`. Zwróć jednak uwagę na zastosowanie `count` jako parametru w wywołaniu funkcji `useCallback`. Oznacza to, że kiedy wartość `count` zmieni się, funkcja `setCountCallback` zostanie ponownie zainicjowana z wykorzystaniem bieżącej wartości właściwości `count`. Pozostała część kodu zwraca odpowiedni kod JSX, który wygeneruje końcowy kod HTML komponentu:

```

console.log("Renderuję komponent App");
return (
    <div className="App">
        <header className="App-header">
            <img src={logo} className="App-logo" alt="logo" />

            <Greeting
                message={message}
                enteredName={enteredName}
                greetingDispatcher={dispatch} />

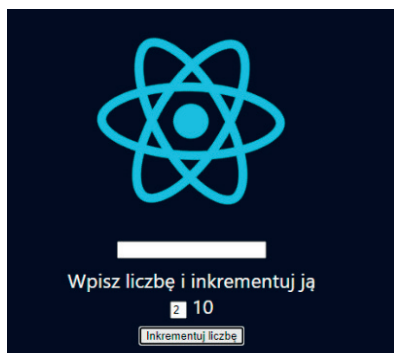
            <div style={{marginTop: '10px'}}>
                <label>Wpisz liczbę i inkrementuj ją</label>
                <br/>
                <input value={startCount}
                    onChange={onChangeStartCount}
                    style={{width: '.75rem'}} />&nbsp;
                <label>{count}</label>
                <br/>
                <button onClick={onWelcomeBtnClick}>Inkrementuj liczbę</button>
            </div>
        </header>
    </div>
)

```

Instrukcja `return` zwraca interfejs użytkownika umożliwiający podawanie wartości początkowej oraz inkrementowanie jej.

Kiedy uruchomisz ten kod i klikniesz przycisk *Inkrementuj liczbę*, zauważysz, że podana wartość początkowa faktycznie jest inkrementowana (jak pokazałem na rysunku 5.7).





**Rysunek 5.7.** Wygląd aplikacji po ośmiokrotnym kliknięciu przycisku

A teraz spróbujemy zmienić tablicę przekazywaną w wywołaniu funkcji `useCallback`, z jej dotychczasowej postaci `[count, startCount]`: usunemy z niej zmienną `count`, zostawiając jedynie `[startCount]`. Okaze się, że wyświetlana na stronie liczba nie jest inkrementowana, gdyż usunęliśmy zależność od `count`. Niezależnie od tego, ile razy będziemy klikać przycisk, wyświetlana wartość zostanie zmodyfikowana tylko raz, po pierwszym kliknięciu (co pokazałem na rysunku 5.8).



**Rysunek 5.8.** Po usunięciu zależności od `count`

A zatem, nawet jeśli wiele razy klikniemy przycisk, liczba wpisana w polu zostanie powiększona tylko o jeden; wynika to z faktu, że funkcja jest przechowywana w pamięci podręcznej i podczas wykonywania zawsze będzie używać początkowej wartości zmiennej `count`.

Przeanalizujemy teraz jeszcze jeden przykład, tym razem związany z wydajnością działania. Zastosujemy w nim funkcję o nazwie `memo`, aby ograniczyć liczbę wykonywanych operacji renderowania. Choć funkcja ta nie jest *hookiem*, to jednak stanowi nową możliwość, stosunkowo niedawno dodaną do Reacta. Wykonaj czynności opisane na poniższej liście:

1. Utwórz nowy plik o nazwie `ListCreator.tsx` i zapisz w nim następujący kod:

```
import React, { FC, useEffect, useRef } from 'react';

export interface ListItem {
  id: number;
```

```

}
export interface ListItems {
  listItems?: Array<ListItem>;
}

const ListCreator: FC<ListItems> = ({listItems}:ListItems) => {
  let renderItems = useRef<Array<JSX.Element> | undefined>();
  useEffect(() => {
    console.log("Kolekcja listItems została zaktualizowana");
    renderItems.current = listItems?.map((item, index) => {
      return <div key={item.id}>
        {item.id}
      </div>;
    });
  }, [listItems]);

  console.log("Renderuję komponent ListCreator");
  return (
    <React.Fragment>
      {renderItems.current}
    </React.Fragment>
  );
}
export default ListCreator;

```

Ten komponent pobiera tablicę elementów i wyświetla je w formie listy.

2. Teraz zaktualizuj plik *App.tsx*, w taki sposób, by u dołu strony była wyświetlana lista, której zawartość będzie zależna od liczby kliknięć przycisku. Podkreślam, że w tym punkcie modyfikujemy wyłącznie kod funkcji *App*. Zwróć także uwagę na to, że trzeba zmodyfikować instrukcje importu, a konkretnie: zaimportować komponent *ListCreator*:

```

function App() {
  const [{ message, enteredName }, dispatch] =
    useReducer(reducer, initialState);

  const [startCount, setStartCount] = useState(0);
  const [count, setCount] = useState(0);

  const setCountCallback = useCallback(() => {
    const inc = count + 1 > startCount ? count + 1 : Number(count + 1) +
      ↳startCount;
    setCount(inc);
  }, [count, startCount]);

  const [listItems, setListItems] = useState<Array<ListItem>>>();

  useEffect(() => {
    const li = [];
    for(let i = 0; i < count; i++) {
      li.push({ id: i });
    }
    setListItems(li);
  }, [count]);

```

Jak widać, do komponentu App dodaliśmy właściwość `listItems` oraz wywołanie nowej funkcji, `useEffect`, którego używamy do wypełniania tej listy. Lista ta będzie aktualizowana za każdym razem, gdy zmieni się wartość właściwości `count`:

```
const onWelcomeBtnClick = () => {
  setCountCallback();
}

const onChangeStartCount = (e: React.ChangeEvent<HTMLInputElement>) => {
  setStartCount(Number(e.target.value));
}

console.log("Renderuję komponent App");
return (
  <div className="App">
    <header className="App-header">
      <img src={logo} className="App-logo" alt="logo" />

      <Greeting
        message={message}
        enteredName={enteredName}
        greetingDispatcher={ dispatch } />

      <div style={{marginTop: '10px'}}>
        <label>Wpisz liczbę i inkrementuj ją</label>
        <br/>
        <input value={startCount}
          onChange={onChangeStartCount}
          style={{width: '.75rem'}} />&nbsp;
        <label>{count}</label>
        <br/>
        <button onClick={onWelcomeBtnClick}>Inkrementuj liczbę</button>
      </div>
      <div>
        <ListCreator listItems={listItems} />
      </div>
    </header>
  </div>
)
```

Kiedy uruchomisz ten przykład, przekonasz się, że nowe elementy listy będą wyświetlane nie tylko po inkrementacji liczby, lecz także kiedy w górnym polu tekstowym wpiszesz imię. Dzieje się tak dlatego, że za każdym razem, gdy komponent nadrzędny zostanie ponownie wyrenderowany ze względu na zmianę jego stanu, zostaną także wyrenderowane jego komponenty podrzędne.

3. A teraz wprowadźmy drobną zmianę w kodzie komponentu `ListCreator`, która ograniczy liczbę wykonywanych operacji renderowania:

```
const ListCreator: FC<ListItems> = React.memo(({listItems}: ListItems) => {
  let renderItems = useRef<Array<JSX.Element> | undefined>();
  useEffect(() => {
    console.log("Kolekcja listItems została zaktualizowana");
    renderItems.current = listItems?.map((item, index) => {
```

```

        return <div key={item.id}>
            {item.id}
        </div>;
    });
}, [listItems]);

console.log("Renderuję komponent ListCreator");
return (
    <React.Fragment>
        {renderItems.current}
    </React.Fragment>
);
});

```

Jak widać, przedstawiłem tu jedynie kod komponentu `ListCreator`, bo tylko w nim wprowadziliśmy zmianę — dodaliśmy wywołanie funkcji opakującej, `React.memo`. Funkcja ta zezwala na aktualizowanie komponentu wyłącznie w przypadku, gdy uległy zmianie wartości przekazywanych do niego właściwości *props*. Takie ograniczenie zapewnia nieco lepszą wydajność działania. Gdybyśmy jednak mieli do czynienia ze złożonym obiektem, zawierającym wiele właściwości, to zyski mogłyby być znacznie większe.

Powyższe przykłady pokazują, że każdego *hooka* można używać w różnych komponentach i z różnymi parametrami. I to jest właśnie najważniejsza cecha *hooków*. Dzięki nim wielokrotne stosowanie kodu jest znacznie łatwiejsze.

Zauważ, że `useState` oraz `useReducer` są jedynie funkcjami przeznaczonymi do wielokrotnego stosowania, które pozwalają na używanie funkcji w wielu różnych komponentach. A zatem, wywołanie funkcji `useState` w komponencie A, a następnie w komponencie B, nie pozwoli nam współużytkować stanu w obu komponentach, i to nawet w przypadku, gdy w obu wywołaniach funkcji `useState` podamy te same nazwy. Samych funkcji możemy używać wiele razy, lecz dotyczy to tylko funkcji — niczego więcej.

W tym podrozdziale przedstawiłem komponenty funkcyjne i *hooki* Reacta. Przedstawiłem także kilka wybranych, najważniejszych spośród dostępnych *hooków* oraz pokazałem na przykładach, jak ich używać. Dodatkowe informacje na temat *hooków* zamieszczę także w dalszej części książki, kiedy zaczniemy pisać aplikacje Reacta. Znajomość *hooków* ułatwi nam w przyszłości pracę nad własnymi komponentami.

## Porównanie stosowania komponentów klasowych i hooków

W tym podrozdziale opiszę podstawowe różnice pomiędzy tworzeniem aplikacji Reacta korzystających z komponentów klasowych oraz funkcyjnych, wykorzystujących *hooki*. Czytając zamieszczone tu informacje zrozumiesz, dlaczego twórcy Reacta uznali, że *hooki* stanowią krok we właściwym kierunku.

## Wielokrotne stosowanie kodu

Jeśli przyjrzyś się metodom cyklu życia komponentów klasowych, zauważysz zapewne, że nie tylko wiąże się z nimi wiele informacji, które trzeba zapamiętać i zrozumieć, lecz także, że dla każdego komponentu klasowego konieczne jest tworzenie unikalnych implementacji tych metod. Właśnie to sprawia, że stosowanie komponentów klasowych utrudnia wielokrotne wykorzystywanie kodu. Jeśli chodzi o *hooki*, to także w ich przypadku mamy do dyspozycji wiele wbudowanych *hooków*, których możemy używać i które musimy poznać. Jednak te *hooki* nie są charakterystyczne dla żadnego konkretnego komponentu i bez żadnych ograniczeń można ich używać w dowolnych tworzonych komponentach. I właśnie ta ich cecha jest podstawową zaletą przemawiającą za ich stosowaniem. W razie stosowania *hooków* wielokrotne wykorzystywanie tego samego kodu jest znacznie łatwiejsze, gdyż kod ten nie jest powiązany z żadną konkretną klasą. Każdy *hook* ma za zadanie dostarczać określone możliwości lub funkcjonalności, niezależnie od tego, gdzie go użyjemy. Co więcej, jeśli zdecydujemy się na przygotowanie własnych *hooków*, to także ich będziemy mogli dowolnie używać.

Przyjrzyj się komponentowi `Greeting` z projektu *class-components*. W jaki sposób mogliśmy wykorzystać kod z tego komponentu? A nawet gdybyśmy mogli to zrobić, to tak naprawdę nie zapewniałby on nam żadnych faktycznych korzyści. Co więcej, zastosowanie funkcji `getDerivedStateFromProps` zwiększa złożoność komponentu i może przyczyniać się do jego częstszego renderowania. A pomijam tu w ogóle fakt, że nie zastosowaliśmy w tym komponentcie żadnej metody cyklu życia.

*Hooki* — i w ogóle framework React jako całość — przypisują teraz znacznie większe znaczenie tworzeniu komponentów niż dziedziczeniu. Twórcy Reacta stwierdzili nawet, że najlepszą praktyką związaną z wielokrotnym stosowaniem kodu jest tworzenie komponentów wewnątrz innych komponentów.

Powtórzę zatem jeszcze raz: metody cyklu życia są przeważnie powiązane z konkretnymi komponentami klasowymi, natomiast jeśli włożymy nieco pracy w odpowiednie uogólnienie *hooków*, będziemy mogli ich używać w wielu różnych komponentach.

## Prostota

Czy pamiętasz, w jak dużym stopniu dodanie funkcji `getDerivedStateFromProps` skomplikowało i wydłużyło kod komponentu `Greeting`? We wszystkich komponentach musieliśmy także implementować konstruktory inicjujące ich stan oraz użyć funkcji `bind`. Na szczęście nasze komponenty były proste, więc nie stanowiło to wielkiego wyzwania. Jednak w produkcyjnym kodzie będą zapewne występować komponenty zawierające wiele funkcji, a żeby zapewnić ich prawidłowe działanie, dla każdej z nich trzeba będzie wywołać funkcję `bind`.

Komponent `Greeting` z projektu *hooks-components* był znacznie prostszy. Nawet gdy komponenty będą się stawać coraz większe, to zazwyczaj stosowane w nich *hooki* będą się powtarzać, co dodatkowo ułatwi czytanie i analizę kodu.

## Podsumowanie

W tym rozdziale zamieściłem bardzo dużo informacji. Przedstawiłem w nim komponenty klasowe i wyjaśniłem, dlaczego ich stosowanie jest kłopotliwe. Następnie opisałem komponenty funkcyjne i *hooki* wykorzystywane podczas ich tworzenia. Wy tłumaczyłem także, dlaczego są one znacznie łatwiejsze do tworzenia i stosowania.

Teraz już znasz podstawy programowania aplikacji z użyciem frameworka React. Możemy zatem przejść do pisania własnych komponentów oraz zacząć tworzyć własną aplikację!

Z następnego rozdziału dowiesz się o narzędziach frameworka React. Wykorzystasz w nim wiedzę zdobytą wcześniej oraz informacje o narzędziach React, aby tworzyć przejrzysty i responsywny kod.

# Przygotowywanie projektu za pomocą create-react-app i testowanie go przy użyciu Jest

W tym rozdziale opiszę narzędzia, które będą nam pomagać w tworzeniu aplikacji Reacta. Podczas profesjonalnego tworzenia oprogramowania, niezależnie od używanego języka czy frameworka, zawsze stosowane są narzędzia umożliwiające pisanie aplikacji szybciej, wydajniej i pozwalające na zapewnienie jak najwyższej jakości kodu. Nie inaczej jest w przypadku ekosystemu Reacta. Społeczność użytkowników Reacta gromadzi się wokół pewnych narzędzi i metodologii pracy, i to właśnie je przedstawię w tym rozdziale. Te wyrafinowane narzędzia i metody pozwolą nam tworzyć lepsze aplikacje i przeprowadzać refaktoryzację kodu w celu dostosowania go do nowych wymagań.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- przedstawieniem metod programowania aplikacji Reacta i systemu używanego do ich budowania;
- testowaniem aplikacji Reacta po stronie klienta;
- narzędziami i praktykami stosowanymi podczas tworzenia aplikacji Reacta.

## Wymagania techniczne

Powinieneś dysponować podstawową znajomością sposobów tworzenia aplikacji internetowych oraz jednostronicowych, które zostały przedstawione w poprzednich rozdziałach książki. Podobnie jak w poprzednich rozdziałach, także w tym będziemy używali środowiska Node (npm) oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial06* (*Chap6*).

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog, o nazwie *rozdzial06*.

## Przedstawienie metod programowania aplikacji Reacta i systemu używanego do ich budowania

W tym podrozdziale poznasz narzędzia i praktyki stosowane podczas pisania i budowania aplikacji Reacta. Wiele z opisanych tu metod jest ogólnie używanych podczas tworzenia nowoczesnych aplikacji w języku JavaScript, nawet w razie stosowania rozwiązań konkurencyjnych dla Reacta, takich jak frameworki Angular czy Vue.

Do tworzenia dużych i złożonych aplikacji potrzebne są narzędzia — i to wiele narzędzi. Niektóre z nich pomagają nam w pisaniu kodu o lepszej jakości, inne umożliwiają zarządzanie kodem oraz współdzielenie go z innymi programistami, a jeszcze inne istnieją tylko po to, by poprawiać wydajność pracy programistów oraz ułatwiać debugowanie kodu. Dlatego poznanie narzędzi używanych podczas tworzenia nowoczesnych aplikacji Reacta sprawi, że nasze aplikacje będą działały jak należy i sprawiały jak najmniej problemów.



## Narzędzia do zarządzania projektami

Jak już się przekonałeś w poprzednich rozdziałach, do przygotowania końcowej postaci aplikacji Reacta koniecznych jest wiele różnych komponentów. Do zarządzania strukturą projektu oraz jego podstawowymi zależnościami, większość programistów używa create-react-app — rozwiązania bazującego na narzędziach, które pierwotnie zostały stworzone z myślą o pisaniu aplikacji dla środowiska Node (npm). Miałeś już okazję poznać próbkę możliwości narzędzia create-react-app, ale w tym rozdziale dowiesz się o nim znacznie więcej.

Jednak zanim zaczniemy, musisz dowiedzieć się, w jaki sposób doszło do tego, że obecnie możemy korzystać z dostępnych, doskonałych narzędzi i technik programistycznych. Informacje te pozwolą lepiej zrozumieć, dlaczego aktualnie preferowany jest styl programowania aplikacji Reacta oraz jakie są jego zalety.

### Jak to robiono wcześniej

WWW jest mieszaniną wielu różnych technologii. Jako pierwszy pojawił się język HTML, zapewniający możliwość tworzenia i udostępniania dokumentów tekstowych. Następnie pojawiły się kaskadowe arkusze stylów (CSS) pozwalające określać wygląd tych dokumentów. Elementem, który wprowadzono jako ostatni, jest język JavaScript, który dodaje możliwości obsługi zdarzeń oraz kontroli programowej. Nic zatem dziwnego, że czasami zintegrowanie tych wszystkich trzech technologii w jednej spójnej aplikacji może się wydawać niewygodne i trudne. Przyjrzyjmy się prostym przykładom połączenia tych technologii bez stosowania jakichkolwiek wyszukanych narzędzi:

1. W panelu terminala lub oknie wiersza poleceń przejdź do katalogu *rodziazl06*, a następnie utwórz podkatalog *OldStyleWebApp*.
2. W edytorze VSCode utwórz plik HTML o nazwie *index.html* i zapisz w nim kod przedstawiony poniżej. Strona ta będzie pozwalać na wpisanie tekstu i jego wyświetlenie.

```
<html lang="en">
<head>
  <meta charset="utf-8">

  <title>Nauka Reacta</title>

  <link rel="stylesheet" href="core.css">
</head>

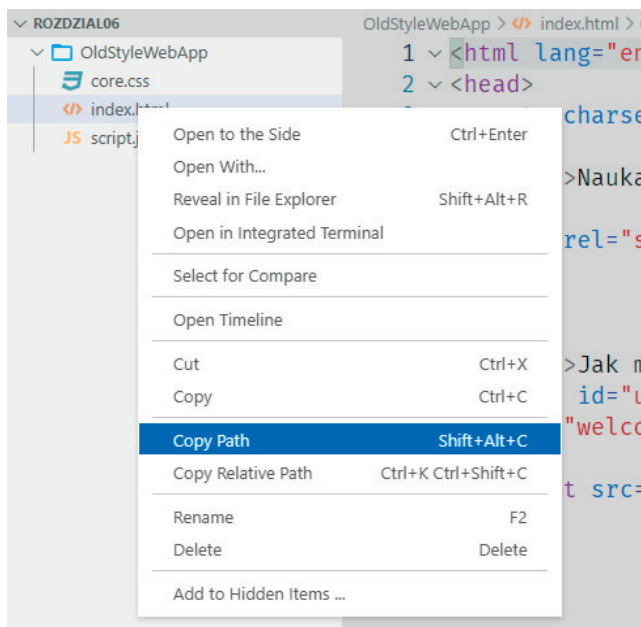
<body>
  <label>Jak masz na imię:</label>
  <input id="userName" />
  <p id="welcomeMsg"></p>

  <script src="script.js"></script>
</body>
</html>
```

3. Teraz w tym samym katalogu utwórz plik CSS o nazwie *core.css*.
4. Następnie, również w tym samym katalogu, utwórz plik JavaScript o nazwie *script.js*.

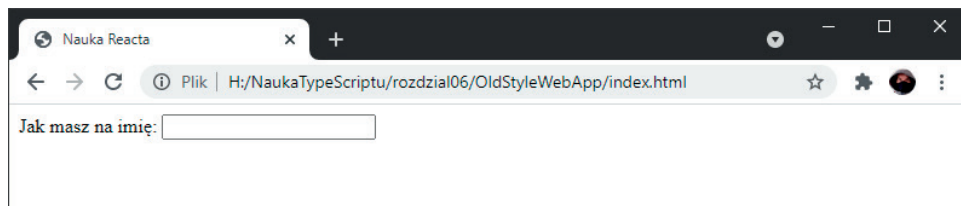
Zawartość plików CSS i JavaScript zapiszemy później, już teraz jednak pojawił się pierwszy problem. W jaki sposób możemy uruchomić tę aplikację? Innymi słowy: w jaki sposób możemy ją wyświetlić i przekonać się, czy działa? Zobaczmy, co możemy w tym celu zrobić:

1. W edytorze VSCode kliknij prawym przyciskiem myszy plik `index.html` i skopiuj jego ścieżkę, jak pokazałem na rysunku 6.1.



Rysunek 6.1. Kopiowanie ścieżki dostępu do pliku `index.html`

2. Teraz otwórz przeglądarkę i wklej skopiowaną ścieżkę w polu adresu URL. W efekcie przeglądarka wyświetli wskazany plik, jak pokazałem na rysunku 6.2.



Rysunek 6.2. Plik `index.html` wyświetlony w przeglądarce

Może jeszcze o tym nie wiedziałeś, ale wcale nie potrzebujesz serwera WWW, żeby przeglądać plik HTML w przeglądarce. Jednak, jak łatwo zauważyć, taki sposób przeglądania stron nie jest zbyt efektywny i dobrze by było, gdybyśmy mogli go jakoś zautomatyzować, włącznie z automatycznym odświeżaniem wyświetlanej strony w przypadku wprowadzenia jakichś zmian w niej samej lub w plikach z nią powiązanych.

### 3. Teraz zapisz poniższy kod w pliku CSS:

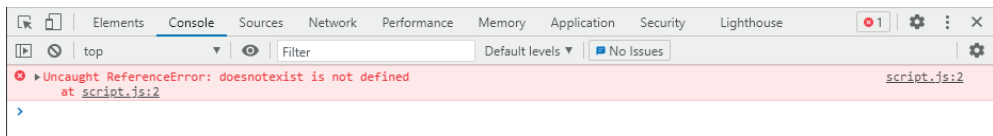
```
label {
  color: blue;
}
p {
  font-size: 2rem;
}
```

Bez trudu zauważysz, że nawet kiedy zapiszesz ten plik na dysku, to element `label` wyświetlony w przeglądarce nie zostanie automatycznie zaktualizowany. Musimy w tym celu odświeżyć stronę w przeglądarce — dopiero to spowoduje uwzględnienie zmian. A co w sytuacji, kiedy takich plików aktualizowanych podczas prac nad aplikacją będzie kilka albo kilkadziesiąt? W takim przypadku konieczność ręcznego odświeżania przeglądarki po zmodyfikowaniu każdego z takich plików byłaby prawdziwym utrapieniem.

### 4. Teraz zmodyfikuj zawartość pliku `script.js`:

```
const inputEl = document.querySelector("#userNam");
console.log("input", doesnotexist);
```

Przyjrzyj się temu kodowi bardzo uważnie, gdyż jest z nim parę problemów. Zobaczmy, o co konkretnie chodzi. Kiedy zapiszesz ten plik, otworzysz narzędzia dla programistów przeglądarki, a następnie odświeżysz wyświetlaną stronę, przekonasz się natychmiast, że wykonanie skryptu powoduje wyświetlenie błędu (znajdziesz go na karcie *Console*). Pokazałem go na rysunku 6.3.



Rysunek 6.3. Pierwszy błąd w skrypcie `script.js`

Błędy tego typu, czyli użycie niezdefiniowanej zmiennej, będą zazwyczaj wykrywane przez program `create-react-app`. Projekty tworzone przy jego użyciu są wyposażone w kolejny program narzędziowy, określany jako *linter*. Linter to program, który analizuje kod źródłowy. Lintery zazwyczaj działają w tle i sprawdzają kod podczas jego wpisywania. Programy tego typu wykrywają często występujące błędy, takie jak ten przedstawiony na rysunku 6.3, dzięki czemu można je usunąć zanim kod trafi do wersji produkcyjnej aplikacji. Oczywiście lintery mają znacznie większe możliwości, ale dokładniej przyjrzymy się im w dalszej części książki. W tym momencie interesuje nas to, by unikać błędów tego typu jeszcze przed uruchomieniem aplikacji. I właśnie takie możliwości zapewnia nam narzędzie `create-react-app`, a w zasadzie inne programy narzędziowe, z których ono korzysta.

### 5. Popraw nazwę zmiennej w kodzie pliku `script.js`, tak jak pokazałem na poniższym przykładzie, a następnie ponownie odśwież stronę wyświetloną w przeglądarce.

```
const inputEl = document.querySelector("#userNam");
console.log("input", inputEl);
```

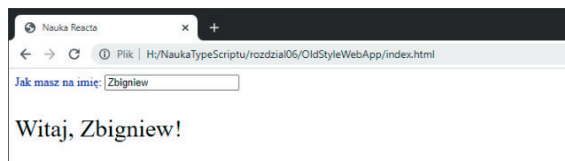
Jak pokazują wyniki metody `console.log` wyświetlonej na konsoli debuggera w przeglądarce, nie udało się znaleźć elementu `inputEl` i zamiast niego została wyświetlona wartość `null`. Ten błąd jest spowodowany prostą literówką w identyfikatorze elementu — w wywołaniu `console.log` zamiast `userName` wpisaliśmy `userNam`. W przypadku aplikacji Reacta tworzonych przy użyciu narzędzia `create-react-app` popełnianie błędów tego typu po prostu nie jest możliwe, gdyż kod aplikacji Reacta niemal nigdy nie próbuje wyszukiwać elementów HTML na stronach. Zamiast tego w aplikacjach Reacta bezpośrednio korzystamy z komponentów, co pozwala całkowicie unikać problemów tego typu. Warto wspomnieć, że istnieje możliwość odwoływania się do elementów HTML w aplikacjach Reacta, przy użyciu funkcji `useRef`. Tego rozwiązania należy jednak używać sporadycznie, gdyż oznacza ono celowe obejście normalnego sposobu działania Reacta poprzez zastosowanie tego konkretnego *hooka*, a to z kolei eliminuje część korzyści, jakie zapewnia nam React.

6. Spróbujmy zatem poprawić nasz skrypt — zmodyfikuj zawartość pliku *script.js* tak, by była zgodna z poniższym kodem:

```
const inputEl = document.querySelector("#userName");
console.log("input", inputEl);
const parEl = document.querySelector("#welcomeMsg");

inputEl.addEventListener("change", (e) => {
  parEl.innerHTML = "Witaj, " + e.target.value + "!";
});
```

Kiedy odświeżysz stronę w przeglądarce, by uruchomić ten kod, przekonasz się, że wpisanie imienia w polu tekstowym i kliknięcie gdziekolwiek poza nim spowoduje wyświetlenie podanego imienia, jak pokazałem na rysunku 6.4.



**Rysunek 6.4.** Powitanie wyświetlone na stronie

Jak widać, działanie tego kodu polega na odczytaniu imienia i wyświetleniu na stronie komunikatu powitalnego. Jednak w takim kodzie bardzo łatwo można popełnić błąd i nie uzyskać żadnych informacji o jego przyczynach. Co więcej, zwróć uwagę na to, że w takim kodzie nie możemy używać języka TypeScript, gdyż przeglądarki go nie obsługują — w przeglądarkach można uruchamiać wyłącznie kod JavaScript. A to z kolei oznacza, że tracimy możliwość korzystania z informacji o typach, które są bardzo użyteczne i pozwalają unikać błędów związanych ze stosowaniem wartości niewłaściwych typów.

Powyższe przykłady bardzo wyraźnie pokazały problemy związane ze stosowaniem klasycznych sposobów tworzenia witryn i aplikacji internetowych. Niestety, prawda jest jeszcze gorsza, gdyż okazuje się, że to nawet nie jest wierzchołek góry lodowej, jaką są problemy

związane z tym sposobem programowania aplikacji. Na przykład umieszczanie znaczników script bezpośrednio w kodzie dokumentów HTML jest uzasadnione, jeśli używane skrypty są krótkie i jest ich niewiele. A co w sytuacji, gdy lista używanych zależności zacznie się wydłużać? W przypadku dużych aplikacji nie jest rzadkością, że takich zależności będą setki. Zarządzanie tak wielką liczbą skryptów byłoby bardzo trudne. A to nie wszystko: wiele zależności używanych przez kod JavaScript nie udostępnia już adresów URL, przy użyciu których można się do nich odwoływać.

Pomijając to wszystko, o czym wcześniej wspominałem, chyba najpoważniejszym problemem związanym z takim sposobem tworzenia aplikacji internetowych jest bardzo swobodna natura używanego kodu. Jeśli przyjrzymy się plikowi *script.js*, zauważmy, że umieszczony w nim kod nie ma żadnej struktury ani nie przypomina żadnego wzorca. Oczywiście, Twój zespół może opracować i stosować własne wzorce kodu, ale w takim razie co z nowymi programistami, którzy dołączą do tego zespołu? Musieliby poznać unikalną strukturę kodu, stosowaną wyłącznie w Twojej firmie.

A zatem, istotne jest to, że narzędzia, frameworki oraz struktura określają i dostarczają spójne i powtarzalne sposoby pisania oraz pielęgnacji kodu. Można je sobie wyobrażać jako swoją kulturę programowania, określającą normy i praktyki używane przez wszystkich, którzy chcą tę kulturę stosować, a przez to wiedzą, co robić i jak się zachowywać. Dzięki temu pisanie kodu, jego współdzielenie i refaktoryzacja stają się łatwiejsze. Teraz, kiedy już przyjrzelśmy się temu „swobodnemu” sposobowi tworzenia aplikacji internetowych, możemy wrócić do narzędzia create-react-app i dokładniej mu się przyjrzeć.

## create-react-app

W poprzednich rozdziałach, choćby rozdziale 4., pt. „Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React” i rozdziale 5., pt. „Tworzenie aplikacji Reacta z wykorzystaniem hooków”, zastosowaliśmy narzędzie create-react-app do utworzenia bazowych projektów aplikacji. Przyjrzyjmy się dokładniej strukturze projektów tworzonych przy użyciu tego narzędzia. Aby lepiej zrozumieć, z jakich elementów składają się takie projekty, musimy zacząć od wykonania polecenia eject. Wykonanie go oznacza, że ujawnimy wszystkie wewnętrzne zależności i skrypty, z których korzysta narzędzie create-react-app i które normalnie pozostają ukryte.

W przeważającej większości projektów Reacta tworzonych przy użyciu create-react-app w ogóle nie będziemy korzystać z polecenia eject, gdyż nie zapewnia to żadnych korzyści. Tutaj użyjemy go tylko po to, by dokładniej poznać działanie projektów.

Wykonaj następujące czynności:

1. W katalogu *rozdzial06* utwórz nowy projekt, wykonując w tym celu następujące polecenie:

```
npx create-react-project ejected-app --template typescript
```

Kiedy realizacja polecenia zostanie zakończona, w katalogu *rozdzial06* powinien się pojawić nowy podkatalog, o nazwie *ejected-app*.

2. A teraz spróbujmy użyć polecenia `eject` — przejdź do katalogu utworzonej aplikacji, *ejected-app*, i wykonaj następujące polecenie:

```
npm run eject
```

Potwierdź chęć wykonania polecenia, naciskając klawisz *y*.

Kiedy polecenie zostanie wykonane, otwórz katalog projektu w edytorze VSCode i przejrzyj jego zawartość w panelu *Explorer*:

#### ■ *config*

Ten katalog zawiera większość plików konfiguracyjnych i skryptów, używanych przez projekt do zapewnienia prawidłowego działania. Najważniejszym spostrzeżeniem z analizy tego katalogu jest zastosowanie **Jest** jako narzędzia do testowania aplikacji oraz **Webpack** jako rozwiązania do pakowania i minimalizacji plików JavaScript. Pierwsze z tych narzędzi, Jest, opisałem w dalszej części rozdziału, w podrozdziale pt. „Testowanie aplikacji Reacta po stronie klienta”, a drugie, Webpack, w dalszej części tego podrozdziału.

#### ■ *node\_modules*

Jak już wiesz, ten katalog zawiera zależności projektu. Jak możesz się przekonać przeglądając jego zawartość jeszcze zanim dodamy do projektu nasze zależności, domyślny zestaw zależności aplikacji Reacta jest ogromny. Byłoby naprawdę trudno zapisać te wszystkie zależności w pliku HTML, używając znaczników `script`. Poza tym, w większości przypadków, tych odwołań nie można zapisywać w kodzie HTML korzystając ze znaczników `script`.

#### ■ *public*

Ten katalog zawiera zasoby statyczne używane podczas generowania naszej aplikacji jednostronicowej. Zaliczają się do nich między innymi: plik HTML o nazwie *index.html* oraz plik *manifest.json*, konieczny, jeśli budujemy aplikację PWA<sup>1</sup>. Można do niego dodawać także inne pliki, na przykład obrazy.

#### ■ *scripts*

Katalog *scripts* zawiera skrypty używane do zarządzania projektem, na przykład skrypty służące do budowania projektu czy też uruchamiania testów aplikacji. Jednak same pliki testów nie powinny być zapisywane w tym katalogu. Zagadnienie testowania opisałem w dalszej części rozdziału, w podrozdziale pt. „Testowanie aplikacji Reacta po stronie klienta”.

#### ■ *src*

To jest oczywiście katalog zawierający kody źródłowe projektu.

#### ■ *.gitignore*

Plik *.gitignore* informuje system zarządzania kodami źródłowymi Git o tym, których plików i katalogów nie należy śledzić. Zagadnienia związane ze stosowaniem systemu Git opisałem dokładniej w dalszej części tego rozdziału.

<sup>1</sup> PWS to skrót od angielskich słów *Progressive Web Application*; określa on specjalny typ aplikacji internetowych, które wyglądają i zachowują się jak rodzima aplikacja mobilna lub desktopowa — *przyp. tłum.*

■ *package.json*

Jak już wspominałem we wcześniejszej części książki, npm jest systemem zarządzania zależnościami, który początkowo został opracowany z myślą o serwerowym środowisku Node. Jednak jego ogromne możliwości oraz popularność sprawiły, że obecnie jest on także standardowym rozwiązaniem używanym do zarządzania zależnościami w tworzonych aplikacjach klienckich. Właśnie dlatego zespół Reacta używa npm jako podstawowego rozwiązania do tworzenia projektów i zarządzania ich zależnościami.

Na samym początku listy zależności projektu można także podawać skrypty przeznaczone do zarządzania nim.

W tym pliku można także podawać ustawienia konfiguracyjne dla takich narzędzi jak Jest, ESLint oraz Babel.

■ *package-lock.json*

Ten plik jest powiązany z *package.json* i pomaga zadbać o to, by w projekcie zawsze był używany właściwy zestaw zależności i podzależności, niezależnie od kolejności, w jakiej poszczególne pakiety będą instalowane. Tego pliku nie musimy używać bezpośrednio, jednak wiedza o tym, że pomaga on unikać problemów związanych z dodawaniem do katalogu *node\_modules* różnych zależności przez różnych programistów, może się przydać.

■ *tsconfig.json*

Ten plik przedstawiłem już w rozdziale 2., pt. „Prezentacja języka TypeScript”, i zgodnie z tym, co tam napisałem, zawiera on informacje używane przez kompilator języka TypeScript. Warto zwrócić uwagę na to, że ogólnie rzecz biorąc, twórcy Reacta preferują stosowanie bardziej rygorystycznych ustawień. Zauważ również, że docelową wersją języka JavaScript używaną przez kompilator jest ES5. Wynika to z faktu, że niektóre przeglądarki nie są jeszcze w pełni zgodne z wersją ES6.

Program create-react-app zawiera także dwa inne, bardzo istotne narzędzia, dostarczające część z jego możliwości funkcjonalnych: Webpack oraz ESLint. Webpack jest narzędziem służącym do pakowania kodu oraz jego minimalizacji. Automatyzuje ono zadania związane z gromadzeniem wszystkich plików wchodzących w skład aplikacji, usuwania z nich wszelkich nadmiarowych, niepotrzebnych elementów oraz scalaniem całego kodu aplikacji do postaci jedynie kilku plików. Usuwanie niepotrzebnych elementów, takich jak znaki odstępu w kodzie, nieużywane skrypty czy pliki źródłowe, może radykalnie zmniejszyć liczbę plików, które będzie musiała pobrać przeglądarka użytkownika. A to oczywiście znacząco poprawi doznania użytkowników. Oprócz tych podstawowych możliwości funkcjonalnych Webpack udostępnia serwer do prac programistycznych wyposażony w mechanizm „odświeżania na bieżąco” (ang. *hot reloading*), który pozwala, by niektóre zmiany w skryptach były automatycznie uwzględniane przez przeglądarkę bez konieczności odświeżania całej strony (choć wydaje się, że większość wprowadzanych zmian i tak powoduje takie odświeżenie; niemniej jednak przynajmniej część z nich będzie obsługiwana automatycznie).

Drugim z ważnych narzędzi jest ESLint. Jak wiadomo, JavaScript jest językiem skryptowym, a nie kompilowanym; a zatem nie dysponuje on kompilatorem, który sprawdzałby poprawność składni i kodu (oczywiście kompilator TypeScriptu dysponuje tymi możliwościami,



jednak on koncentruje się głównie na zagadnieniach związanych ze stosowanymi typami). ESLint eliminuje te braki i zapewnia możliwość sprawdzania poprawności składni kodu JavaScript już w trakcie prowadzenia prac programistycznych. Oprócz tego, pozwala on na tworzenie własnych reguł dotyczących sposobu formatowania kodu źródłowego. Ogólnie rzecz biorąc, reguły te są używane w celu zapewnienia, że wszystkie osoby w firmie będą stosowały ten sam styl pisania kodu; chodzi tu o takie zagadnienia, jak konwencje zapisu nazw zmiennych, wcięcia nawiasów itp. Po określeniu takich reguł ESLint będzie wymuszał ich stosowanie, wyświetlając komunikaty o błędach w razie wykrycia jakichś niezgodności.

Takie reguły nie odnoszą się wyłącznie do języka JavaScript, z powodzeniem można je także stosować podczas pisania frameworków takich jak React. Na przykład w projektach tworzonych przy użyciu `create-react-app` opcja ESLint podana w pliku `package.json` ma wartość `react-app`; oznacza to, że stosowany będzie zestaw reguł dostosowany do potrzeb pisania aplikacji Reacta. A zatem, wiele spośród komunikatów, które będziemy mogli zobaczyć podczas pisania kodu, nie będzie błędami języka JavaScript, lecz informacjami o naruszeniach reguł wymuszających zgodność z najlepszymi praktykami pisania kodu aplikacji Reacta.

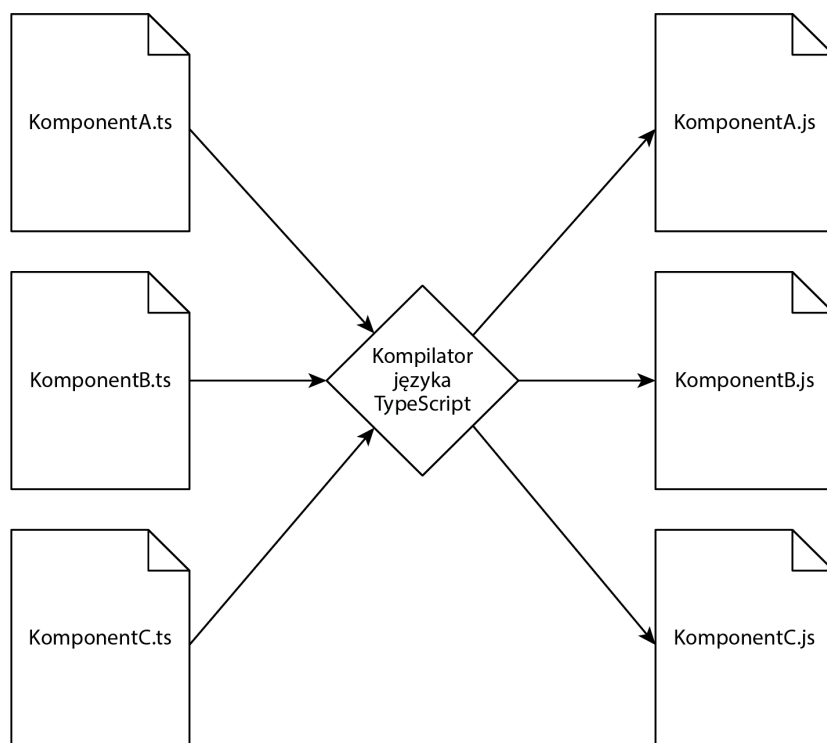
Webpack, choć jest niezwykle potężnym narzędziem, jest także bardzo trudny do skonfigurowania. Również tworzenie własnych reguł ESLint może zająć sporo czasu. Dlatego też kolejną wielką zaletą `create-react-app` jest to, że zapewnia nam dobre ustawienia domyślne dla obu tych narzędzi.

## Transpilacja

O transpilacji wspominałem już w rozdziale 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript”. Jednak w tym rozdziale musimy przyrzeć się jej nieco dokładniej. Transpilacja jest podstawowym mechanizmem generacji kodu wykonywanej przez `create-react-app`. Narzędzie `create-react-app` pozwala na stosowanie TypeScriptu lub narzędzia Babel, dzięki czemu możemy pisać kod w jednym języku lub jego wersji, a następnie generować kod w innym języku lub jego określonej wersji. Na rysunku 6.5 przedstawiłem typowy przepływ kodu podczas transpilacji kodu źródłowego napisanego w języku TypeScript.

Kompilator TypeScriptu przeszukuje projekt i odnajduje wszystkie pliki `.ts` oraz `.tsx` zapisane w katalogu głównym oraz katalogu `src`. Jeśli wykryje w nich błędy, przerywa działanie i informuje nas o nich, a jeśli ich nie znajdzie, to konwertuje kod TypeScript na kod JavaScript zapisywany w plikach `.js`; i to właśnie ten kod jest wykonywany. Warto zwrócić uwagę na to, że w ramach tego procesu może być także zmieniana wersja języka JavaScript. To wszystko oznacza, że transpilacja w dużym stopniu przypomina kompilację — sprawdzana jest poprawność kodu i niektóre kategorie błędów — jednak zamiast kompilować kod do postaci kodu binarnego, który można wykonywać bezpośrednio, generowany jest kod źródłowy zapisany w innym języku programowania lub w innej wersji języka. Narzędzie Babel zapewnia możliwości generowania kodu JavaScript, jak również pracy z kodem napisanym w TypeScriptie. Ja jednak preferuję korzystanie ze standardowego kompilatora TypeScriptu, gdyż został on stworzony przez ten sam zespół, który zaprojektował język TypeScript; ponadto kompilator ten zazwyczaj jest bardziej aktualny.





**Rysunek 6.5.** Transpilacja kodu TypeScript na kod JavaScript

Wybór transpilacji jako sposobu kompilacji kodu zapewnia wiele ważnych zalet. Jedną z nich jest to, że programiści nie muszą przejmować się tym, czy ich kod będzie działać w przeglądarce użytkownika, albo czy przed uruchomieniem aplikacji użytkownik będzie musiał zainstalować na swoim komputerze jakieś zależności. Kompilator języka TypeScript generuje standardowy kod JavaScript zgodny z wersją ECMAScript (ES3, ES5, ES6 i tak dalej), dzięki czemu z powodzeniem będzie on mógł działać we wszystkich nowoczesnych przeglądarkach WWW.

Z drugiej strony, transpilacja zapewnia także programistom możliwość stosowania nowych cech języka JavaScript, jeszcze przed ich ostatecznym udostępnieniem. Ponieważ JavaScript jest aktualizowany w niemal rocznych cyklach, ta cecha transpilacji jest niezwykle użyteczna, gdyż pozwala na korzystanie z najnowszych możliwości samego języka oraz z mechanizmów poprawiających wydajność wykonywania kodu; na przykład w przypadkach, kiedy rozważane jest rozszerzenie JavaScriptu o jakieś nowe możliwości. Zanim zmiany zostaną dopuszczone przez ECMA (organizację standaryzacyjną zajmującą się rozwojem JavaScriptu) i trafią do oficjalnej wersji języka, muszą przejść kilka etapów. Z kolei twórcy języka TypeScript oraz narzędzia Babel czasami akceptują niektóre zmiany proponowane do wprowadzenia w JavaScriptcie jeszcze na wcześniejszych etapach prac nad nimi. To właśnie dzięki temu wielu programistów JavaScriptu mogło używać słów kluczowych `async` i `await` jeszcze zanim stały się one oficjalnym standardem.

## Repozytoria kodu

Repozytorium kodu to system umożliwiający współdzielenie i współużytkowanie kodu źródłowego projektu przez wielu programistów. Dzięki niemu kod może być aktualizowany, kopiowany i scalany. W przypadku dużych zespołów, stosowanie narzędzia tego typu jest absolutnie nieodzowne do prowadzenia prac nad złożonymi aplikacjami. Aktualnie najbardziej popularnym systemem tego typu jest Git. A najbardziej popularnym serwisem obsługującym repozytoria Git jest GitHub.

Choć zamieszczenie szczegółowego opisu sposobów korzystania z systemu Git wykracza poza ramy tej książki, to jednak bardzo ważne jest, byś zrozumiał podstawowe pojęcia z nim związane oraz poznał kilka podstawowych poleceń, gdyż będą one potrzebne podczas interakcji z innymi programistami oraz zarządzania własnymi projektami.

Jednym z najważniejszych pojęć związanych z repozytoriami kodu jest tak zwane „rozgałęzianie” (ang. *branching*). Oznacza ono możliwość tworzenia różnych wersji projektu. Na przykład można tworzyć odrębne gałęzie dla poszczególnych wersji projektu, takich jak: 1.0.0, 1.0.1 i tak dalej. Można ich także używać do tworzenia odrębnych wersji aplikacji, w ramach których będzie rozwijany jakiś kod eksperymentalny lub potencjalnie ryzykowny. Umieszczanie takiego kodu w głównej gałęzi projektu byłoby bardzo niebezpieczne. Na rysunku 6.6 przedstawiłem stronę frameworka React w serwisie GitHub, z widoczną listą wielu różnych wersji projektu.

Jak widać, projekt Reacta składa się z wielu gałęzi. Gałąź reprezentująca aktualną, stabilną wersję projektu nosi nazwę *master* i nie jest widoczna na rysunku.

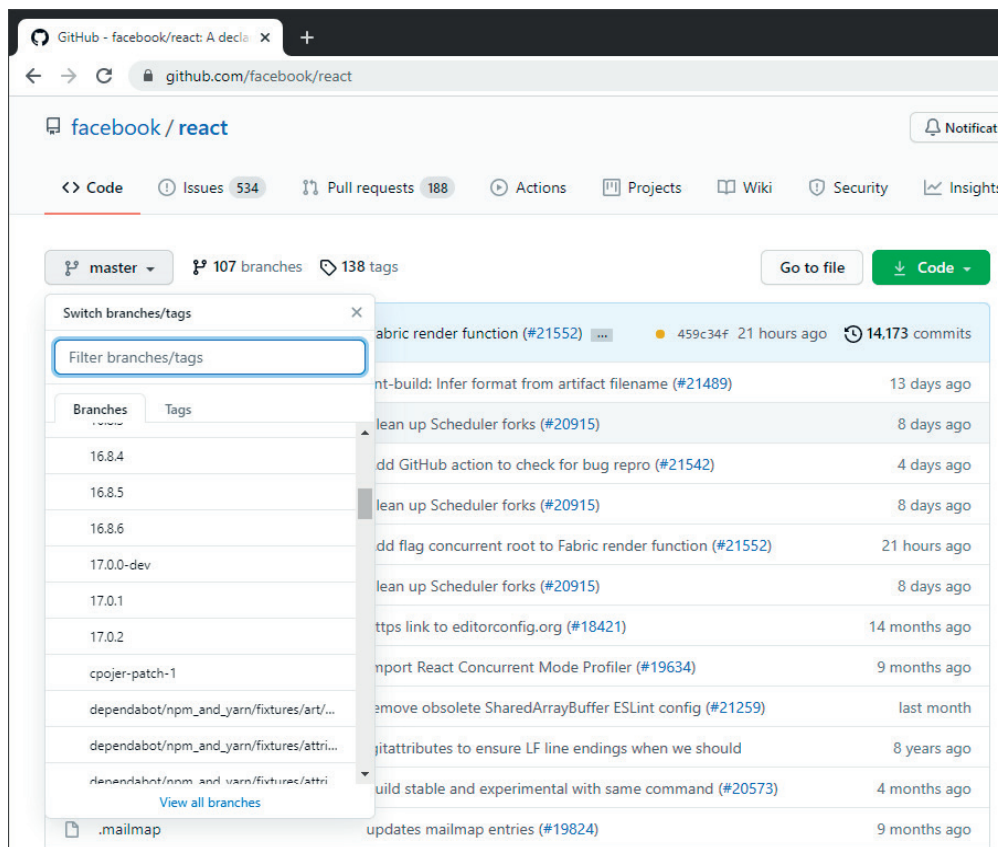
Jeszcze raz muszę zaznaczyć, że dobre i wnikliwe poznanie zasad działania systemu Git i korzystania z niego wymagałoby odrębnej książki, dlatego poniżej jedynie pobieżnie przedstawię kilka najważniejszych poleceń, które są powszechnie stosowane w codziennej pracy:

- **git**. To polecenie zapewnia dostęp do **interfejsu wiersza poleceń (CLI)** Git. Trzeba nim poprzedzać każde polecenie Git, które chcemy wykonać. Polecenie to operuje na lokalnej kopii repozytorium; ani zdalna wersja repozytorium, ani jej lokalne kopie używane przez innych programistów nie są modyfikowane aż do momentu „wypchnięcia” (ang. *push*) zmian na serwer.
- **clone**. To polecenie służy do kopiowania repozytorium na lokalny komputer. Warto zapamiętać, że w przypadku klonowania repozytorium domyślnie wybierana jest jego główna gałąź (zazwyczaj *master*). Poniżej przedstawiłem przykład wykonania tego polecenia:

```
git clone https://github.com/facebook/react.git
```

- **checkout**. To podpolecenie pozwala zmieniać aktualnie wybraną gałąź roboczą. A zatem, gdybyśmy chcieli pracować w innej gałęzi niż *master*, to moglibyśmy to zrobić, używając następującego polecenia:

```
git checkout <nazwa-gałęzi>
```



Rysunek 6.6. Strona Reacta w serwisie GitHub

- **add.** To polecenie służy do dodawania plików, które system Git ma śledzić. Dodanie pliku oznacza, że później zostanie on zatwierdzony i umieszczony w repozytorium. Aby w ten sposób dodać wszystkie zmodyfikowane pliki za jednym razem, za poleceniem **add** należy umieścić kropkę (.); można także jawnie podać nazwy każdego z plików, które chcemy dodać:

```
git add <nazwa-pliku>
```

- **commit.** To polecenie nakazuje zaktualizowanie zawartości aktualnie wybranej gałęzi poprzez zapisanie w niej wszystkich dodanych wcześniej plików. Jeśli do polecenia dodamy parametr **-m**, będziemy mogli podać wiersz tekstu opisujący rejestrowane zmiany. Takie komentarze pomagają współpracownikom określić, co zawierają zmiany zapisane w repozytorium. Oto przykład tego polecenia:

```
git commit -m "Moje zmiany w pliku xyz"
```

- **push.** To polecenie powoduje skopiowanie plików zatwierdzonych w repozytorium lokalnym i zapisanie ich na zdalnym serwerze:

```
git push origin <nazwa-gałęzi>
```

W tym podrozdziale poznałeś podstawowe narzędzia dostępne dla programistów tworzących aplikacje Reacta. Narzędzia `create-react-app`, `ESLint`, `Webpack` i `npm` zapewniają nieocenione możliwości, poprawiające wydajność pracy i ułatwiające tworzenie kodu, który będzie mniej narażony na występowanie błędów. Wyjaśniłem w nim także czym jest transpilacja kodu TypeScript oraz w jaki sposób pozwala nam ona na korzystanie z możliwości nowszych wersji języka bez narażania się na utratę zgodności z urządzeniami użytkowników.

I wreszcie na końcu podrozdziału przedstawiłem pokrótce Git — najpopularniejszy obecnie system do zarządzania repozytoriami kodu źródłowego. Jako profesjonalny programista zapewne będziesz musiał z niego korzystać podczas pracy nad projektami.

Teraz, kiedy już znasz podstawowe i najważniejsze narzędzia, możemy przejść do następnego podrozdziału, poświęconego testowaniu. Nowoczesne praktyki tworzenia oprogramowania w znacznym stopniu bazują na testowaniu oraz frameworkach tekstowych. Na szczęście także w ekosystemie języka JavaScript są dostępne takie frameworki, które ułatwiają nam tworzenie i wykonywanie testów zapewniających wysoką jakość kodu.

## Testowanie aplikacji Reacta po stronie klienta

Niezwykle ważnym elementem programowania są testy jednostkowe. Obecnie nie sposób wyobrazić sobie jakiegokolwiek dużego projektu, który w jakimś stopniu nie będzie z nich korzystał. Celem testów jest upewnienie się, że nasz kod zawsze działa poprawnie i robi to, czego oczekujemy. Testy mają szczególne znaczenie podczas modyfikowania kodu, czyli jego refaktoryzacji. W zasadzie, modyfikowanie istniejącego, złożonego kodu jest najprawdopodobniej trudniejsze od napisania go od nowa. Testy jednostkowe mogą uchronić nas przed uszkodzeniem kodu podczas jego modyfikowania. Co więcej, jeśli zmiany faktycznie doprowadzą do występowania jakichś problemów w kodzie, to testy jednostkowe pozwolą precyzyjnie wskazać miejsce, w którym kod przestał działać, dzięki czemu będzie go można szybko poprawić.

Wcześniej w projektach Reacta używane były powszechnie dwie biblioteki do wykonywania testów: **Jest** oraz **Enzyme**. Pierwsza z nich, Jest, jest główną biblioteką testową udostępniającą podstawowe możliwości, takie jak asercje, sprawdzające konkretne wartości, oraz funkcje opakowujące, używane do konfigurowania testów. Z kolei biblioteka Enzyme zawiera zestaw funkcji pomocniczych, które pozwalają na testowanie komponentów Reacta przy użyciu biblioteki Jest. Możliwości Enzyme koncentrują się na testowaniu wyników zwracanych przez komponenty Reacta. Jednak obecnie miejsce biblioteki Enzyme niemal w całości zajęła inna biblioteka — **testing-library** oraz jej różne wersje. I to właśnie ta biblioteka jest głównym narzędziem służącym do testowania komponentów w projektach Reacta tworzonych przy użyciu `create-react-app`.

Wszystkie testy jednostkowe działają w taki sam sposób. Dotyczy to nie tylko frameworka React oraz języka JavaScript — we wszystkich językach programowania testy działają w taki sam sposób. A zatem, czym jest test jednostkowy? Otóż test jednostkowy stara się zweryfikować

działanie jednego, określonego fragmentu kodu i stara się potwierdzać prawdziwość pewnych założeń (tak zwanych asercji) dotyczących tego kodu. Tak to wygląda ogólnie. Ujmując to w inny sposób, test jednostkowy sprawdza, czy kod faktycznie działa wedle naszych oczekiwań. Jeśli tak nie jest, test powinien zakończyć się niepowodzeniem. Choć założenia są proste, to jednak tworzenie wysokiej jakości testów jednostkowych bynajmniej nie należy do łatwych zadań. Dlatego też w tym podrozdziale przedstawię kilka przykładowych testów; pamiętaj jednak proszę, że w dużych aplikacjach tworzenie testów może być równie skomplikowane, jak pisanie właściwego kodu aplikacji, a niejednokrotnie będzie nawet trudniejsze. Dlatego też musi minąć nieco czasu, zanim nabierzesz praktyki w pisaniu testów.

Zacznijmy od prostego testu, który pozwoli Ci zrozumieć, jak to wszystko działa. W tym celu wykonaj czynności opisane na poniższej liście:

1. W edytorze VSCode otwórz plik *ejected-app/src/App.test.tsx*. Plik ten zawiera test sprawdzający działanie komponentu App. Już za chwilę dokładnie przeanalizujemy kod źródłowy tego testu.
2. Otwórz panel terminala, przejdź w nim do katalogu *ejected-app* i wykonaj następujące polecenie:

**npm run test**

Jak już wspominałem, w skład projektów Reacta wchodzi kilka skryptów służących do zarządzania nim, a jednym z nich jest skrypt test, służący do wykonywania tekstów. Co więcej, ten skrypt powoduje wykonanie testów w trybie obserwowania (ang. *watch mode*), co oznacza, że skrypt pozostaje aktywny i będzie wykonywany automatycznie po wprowadzeniu zmian w kodzie jakiegoś testu lub po dodaniu nowego. Jeśli testy nie zostaną wykonane automatycznie i w terminalu zostanie wyświetlony komunikat przedstawiony na rysunku 6.7, to wybierz opcję a:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS
No tests found related to files changed since last commit.
Press `a` to run all tests, or run Jest with `--watchAll`.

Watch Usage
  > Press a to run all tests.
  > Press f to run only failed tests.
  > Press q to quit watch mode.
  > Press p to filter by a filename regex pattern.
  > Press t to filter by a test name regex pattern.
  > Press Enter to trigger a test run.

```

**Rysunek 6.7.** Opcje wykonania testów

Jeśli Twoje testy zostały wykonane automatycznie lub jeśli wybrałeś opcję a, to powinieneś zobaczyć wyniki przedstawione na rysunku 6.8.

Jak widać, nasze testy (a właściwie: nasz jeden test) zostały znalezione i wykonane. W tym przypadku test został wykonany pomyślnie, czyli stało się to, czego oczekiwaliśmy. Gdyby jednak wystąpiły problemy, to prezentowane komunikaty zawierałyby informacje o tym, ile testów zostało wykonanych poprawnie, a ile zawiodło.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

src/App.test.tsx
✓ renders learn react link (30 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  0 total
Time:        6.719 s, estimated 22 s
Ran all test suites.

Watch Usage: Press w to show more.

```

Rysunek 6.8. Poprawnie wykonany test

A teraz przyjrzyjmy się zawartości pliku *App.test.tsx*:

```

import React from 'react';
import { render } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  const { getByText } = render(<App />);
  const linkElement = getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});

```

W pierwszej kolejności zwróć uwagę na to, że w nazwie pliku występuje słowo *test*. To właśnie ono informuje bibliotekę Jest, że dany plik jest testem. Niektóre zespoły wolą umieszczać wszystkie używane testy w jednym katalogu; inne natomiast wolą zapisywać je w tym samym katalogu, w którym znajduje się testowany plik — i tak właśnie jest w tym przykładzie. Żadnego z tych rozwiązań nie można uznać za lepsze, czy gorsze. Każdy może wybrać takie, które bardziej przypadnie do gustu jemu lub zespołowi. W tej książce będę umieszczał testy w tym samym katalogu, w którym znajdują się testowane pliki. Przyjrzyjmy się zatem zawartości testu:

1. Zwróć uwagę na to, że jedna z instrukcji `import` umieszczonych na początku pliku odwołuje się do `@testing-library/react`. Jak już wspominałem, biblioteka ta udostępnia dodatkowe narzędzia ułatwiające testowanie wyników generowanych przez komponenty Reacta.
2. Następnie zwróć uwagę na funkcję `test`. Działa ona jako opakowanie hermetyzujące pojedynczy test. Oznacza to, że wszystko, co jest powiązane z konkretnym testem, musi się znaleźć w tej funkcji i że żaden kod spoza niej nie może się odwoływać do jej kodu. Dzięki temu żadne inne testy nie będą miały wpływu na ten test.
3. Pierwszym parametrem funkcji `test` jest opis testu. Te opisy mogą być całkowicie dowolne, a Twój zespół najprawdopodobniej będzie miał własne standardy określające jak należy je tworzyć. Warto zwrócić uwagę na to, by opisy te były krótkie, a jednocześnie by precyzyjnie określały, co dany test sprawdza.
4. Drugim parametrem jest funkcja, która wykonuje sam test. W tym przykładzie test sprawdza, czy kod HTML wygenerowany przez komponent `App` będzie zawierał określony łańcuch znaków. Przeanalizujemy kod tej funkcji wiersz po wierszu.

5. W wierszu 6. wywołujemy funkcję `render`, przekazując do niej komponent `App`. Funkcja `render` wykonuje przekazany komponent i zwraca obiekt zawierający pewne właściwości i funkcje, które zapewniają nam możliwość sprawdzania wygenerowanego kodu HTML. W tym przypadku zdecydowaliśmy się na pobranie jedynie funkcji `getByText`, która zwraca element zawierający podany tekst.
6. W wierszu 7. pobieramy poszukiwany element HTML; w tym celu wywołujemy funkcję `getByText`, przekazując do niej parametr o postaci `/learn react/i`; ta składnia jest używana do tworzenia wyrażeń regularnych, jednak tu używamy jej jedynie do podania konkretnego łańcucha znaków.
7. I w końcu w wierszu 8. wykonujemy asercję o nazwie `expect`, która używając funkcji o nazwie `toBeInTheDocument` sprawdza, czy DOM zawiera element określony przez `linkElement`. Jak widać, najprostszym sposobem zrozumienia testów jest przeczytanie stosowanych w nich asercji tak, jak gdyby było normalnymi zdaniem. Na przykład asercję zapisaną w tym teście można przeczytać w następujący sposób: „Oczekuję, że element `linkElement` będzie obecny w dokumencie” (przy czym „dokumentem” jest oczywiście DOM przeglądarki). Jeśli przeczytamy asercję w taki sposób, to jej przeznaczenie stanie się jasne.
8. A teraz zobaczmy, co się stanie, jeśli nieco zmodyfikujemy kod aplikacji. Otwórz plik *App.tsx* i zmodyfikuj go zgodnie z przykładem podanym poniżej (zwróć uwagę na to, że w celu skrócenia kodu przedstawiłem tu jedynie funkcję `App`):

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn
        </a>
      </header>
    </div>
  );
}
```

Zauważ, że kod komponentu jest niemal taki sam jak wcześniej — jedyna zmiana polega na usunięciu słowa `React` z `Learn React`.

9. Kiedy zapiszesz zmodyfikowany plik, w panelu terminala niemal natychmiast zostaną wyświetlone informacje przedstawione na rysunku 6.9.



```

PROBLEMS OUTPUT DEBUG-CONSOLE TERMINAL COMMENTS 2: node
src/App.test.tsx (0.022 s)
  • renders learn react link (49 ms)
    • renders learn react link

TestingLibraryElementError: Unable to find an element with the text: /learn react/1. Th
you can provide a function for your text matcher to make your matcher more flexible.

<body>
  <div>
    <div
      class="App"
    >
      <header
        class="App-header"
      >
        
      </div>
      <div>
        <code>
          src/App.tsx
        </code>
        and save to reload.
      </div>
      <a
        class="App-link"
        href="https://reactjs.org"
        rel="noopener noreferrer"
        target="_blank"
      >
        Learn
      </a>
    </div>
  </div>
</body>

5 | test('renders learn react link', () => {
6 |   const { getByText } = render(<App />);
> 7 |   const linkElement = getByText(/learn react/1);
  |                               ^
8 |   expect(linkElement).toBeInTheDocument();
9 | });
10 |

at Object.getElementError (node_modules/@testing-library/dom/dist/config.js:37:19)
at node_modules/@testing-library/dom/dist/query-helpers.js:90:38
at node_modules/@testing-library/dom/dist/query-helpers.js:62:17
at getByText (node_modules/@testing-library/dom/dist/query-helpers.js:111:19)
at Object.<anonymous> (src/App.test.tsx:7:23)

Test Suites: 1 failed, 1 total
Tests: 1 failed, 1 total
Snapshots: 0 total
Time: 12.106 s
Ran all test suites.

Watch Usage: Press w to show more.

```

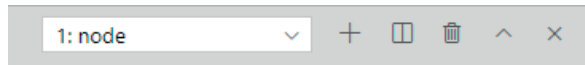
Rysunek 6.9. Błąd wywołany zmianami w kodzie pliku App.tsx

Przypominam, że mechanizm do wykonywania testów działa w trybie obserwowania, więc wyniki powinny zostać wyświetlone bezpośrednio po zapisaniu pliku *App.tsx*. Jak widać, wykonanie testu zakończyło się niepowodzeniem, gdyż na stronie nie znaleziono tekstu `learn react`, co oznacza, że asercja `assert(linkElement).toBeInTheDocument()` nie została spełniona.

No dobrze, w ten sposób przeanalizowaliśmy domyślny test dostarczany przez narzędzie `create-react-app`. Teraz spróbujmy stworzyć nowy komponent, który wykorzystamy do napisania kilku testów od podstaw. Czynności, które powinienś wykonać, opisałem na poniższej liście:

1. Pozostaw narzędzie do wykonywania testów działające w trybie obserwowania i nie przejmuj się tym, że cały czas wyświetla informacje o błędzie. Otwórz nowy panel terminala, klikając w tym celu przycisk „+” umieszczony w prawym, górnym rogu aktualnie widocznego panelu. Przycisk ten przedstawiłem na rysunku 6.10.





Rysunek 6.10. Przycisk „+” służący do tworzenia nowych paneli terminala

2. Teraz w katalogu *src* utwórz nowy plik, *DisplayText.tsx*, i zapisz w nim następujący kod:

```
import React, { useState } from "react";

const DisplayText = () => {
  const [txt, setTxt] = useState("");
  const [msg, setMsg] = useState("");
  const onChangeTxt = (e: React.ChangeEvent<HTMLInputElement>) => {
    setTxt(e.target.value);
  }

  const onClickShowMsg = (e: React.MouseEvent<HTMLButtonElement,
    ↳MouseEvent>) => {
    e.preventDefault();

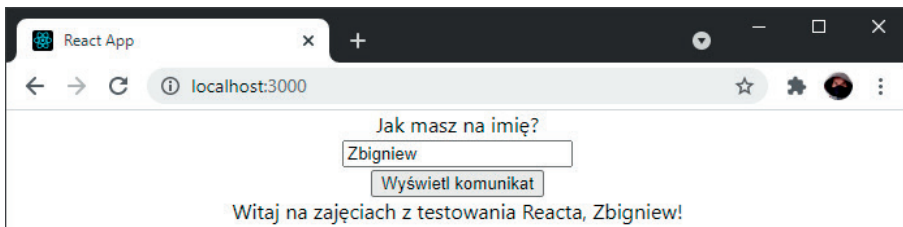
    setMsg(`Witaj na zajęciach z testowania Reacta, ${txt}!`);
  }
}
```

Ten komponent ma za zadanie wyświetlać prosty komunikat, kiedy ktoś wpisze w polu tekstowym swoje imię i kliknie przycisk *Wyświetl komunikat*. Powyższy fragment przedstawia początek deklaracji komponentu *DisplayText*.

3. Jak widać, komponent zawiera stan, na który składają się dwie właściwości, oraz parę procedur obsługi zdarzeń, które obsługują pobieranie wpisanego tekstu i wyświetlanie nowego komunikatu na stronie (to zagadnienie przedstawiłem już w rozdziale 5., pt. „Tworzenie aplikacji Reacta z wykorzystaniem *hooków*”):

```
return (
  <form>
    <div>
      <label>Jak masz na imię?</label>
    </div>
    <div>
      <input data-testid="user-input"
        value={txt} onChange={onChangeTxt} />
    </div>
    <div>
      <button data-testid="input-submit"
        onClick={onClickShowMsg}>Wyświetl komunikat</button>
    </div>
    <div>
      <label data-testid="final-msg">{msg}</label>
    </div>
  </form>
)
}
export default DisplayText;
```

4. Końcowa część komponentu zwraca kod HTML interfejsu użytkownika, składający się z pola tekstowego, przycisku i kilku napisów. Zwróć uwagę na zastosowanie atrybutów `data-testid`, które już niebawem ułatwią nam odwoływanie się do elementów w kodzie testów. Kiedy uruchomisz ten kod i wpiszesz imię w polu tekstowym, zostanie wyświetlony komunikat podobny do tego, który przedstawiłem na rysunku 6.11.



**Rysunek 6.11.** Nowy komponent, którego będziemy używać w testach

Jak widać, nasz komponent jedynie zwraca wpisany tekst dodany do komunikatu. Jednak nawet w tak prostym przykładzie można wskazać kilka rzeczy wartych przetestowania. Przede wszystkim chcielibyśmy upewnić się, że w polu został wpisany jakiś tekst, oraz że są to słowa, a nie jakieś cyfry lub symbole. Oprócz tego chcielibyśmy upewnić się, że po kliknięciu przycisku na stronie zostanie wyświetlony komunikat oraz że będzie on rozpoczynać się od łańcucha "Witamy na zajęciach z testowania Reacta" i kończyć łańcuchem wpisanym w polu tekstowym.

Skoro już komponent jest gotowy, możemy przejść do tworzenia testu, który sprawdzi jego działanie:

1. Zanim zaczniemy, musisz zwrócić uwagę na pewien problem związany z plikiem *tsconfig.json*. Jak już zaznaczyłem wcześniej, tworzone testy można zapisywać w odrębnym katalogu, który zazwyczaj nosi nazwę `__test__`, bądź też w tym samym katalogu, w którym są zapisane pliki testowanych komponentów. W tej książce będziemy umieszczali testy w tym samym katalogu, co komponenty. Jednak, jeśli zdecydujemy się na takie rozwiązanie, to w pliku *tsconfig.json*, w sekcji `compilerOptions`, będziemy musieli dodać następującą opcję:

```
"types": ["node", "jest"]
```

2. Utwórz plik testu, który sprawdzi działanie tego komponentu. W tym celu utwórz plik o nazwie *DisplayText.test.tsx* i zapisz w nim następujący kod:

```
import React from "react";
import { render, fireEvent } from "@testing-library/react";
import DisplayText from "../DisplayText";
import "@testing-library/jest-dom/extend-expect";

describe("Test komponentu DisplayText", () => {
  it("jest wyświetlany bez problemów", () => {
    const { baseElement } = render(<DisplayText />);
    expect(baseElement).toBeInTheDocument();
  });
});
```

```

it("uzyskuje tekst z pola", () => {
  const testuser = "testuser";
  const { getByTestId } = render(<DisplayText />);
  const input = getByTestId("user-input");
  fireEvent.change(input, { target: { value:testuser } });
  expect(input).toBeInTheDocument();
  expect(input).toHaveValue(testuser);
})
});

```

Na samym początku pliku umieszczona jest instrukcja importująca funkcję `render` z `@testing-library/react` oraz instrukcja importująca rozszerzenie `@testing-library/jest-dom/extend-expect`, które pozwoli nam wykonywać asercje. Rozszerzenia definiujące funkcję `expect` udostępniają nam także dodatkowe funkcje, pozwalające na stosowanie innych rodzajów testów. Na przykład, w tym teście korzystamy z funkcji `toHaveValue`, która pobiera i sprawdza wartość pola tekstowego.

Poniżej sekcji z instrukcjami importu znajduje się coś nowego — wywołanie funkcji `describe`. Zgodnie z tym, co sugeruje jej nazwa<sup>2</sup>, funkcja `describe` służy po prostu jako swoisty pojemnik do grupowania, który można opatrzyć pomocną nazwą. Wewnątrz takiego pojemnika można umieścić dowolną liczbę testów, jednak warto zwrócić uwagę na to, że wszystkie te testy powinny być powiązane i sprawdzać działanie konkretnego komponentu lub jego możliwości. W naszym przypadku będziemy testować komponent `DisplayText`, dlatego wszystkie testy umieszczone w wywołaniu funkcji `describe` będą odnosić się do tego komponentu.

Nasz pierwszy test jest uruchamiany przy użyciu funkcji o nazwie `it`. Funkcja ta sprawdza, czy testowany komponent, czyli `DisplayText`, może być wyświetlony w formie kodu HTML bez wywoływania awarii czy też jakichś błędów. Funkcja `render` próbuje wyrenderować komponent, natomiast funkcje `expect` i `toBeInTheDocument` upewniają się, czy operacja przebiegła pomyślnie — w tym celu sprawdzają, czy komponent jest dostępny w modelu DOM. W ramach eksperymentu spróbuj dodać do pierwszego testu, poniżej wiersza zaczynającego się od `const { baseElement }`, wywołanie o postaci `console.log(baseElement.innerHTML)`. W efekcie powinieneś uzyskać wyniki podobne do tych przedstawionych na rysunku 6.12. (Zwróć uwagę na to, że przy okazji poprawiłem test sprawdzający komponent `App`; jego zmodyfikowany kod znajdziesz w przykładach dołączonych do książki.)

Jak widać, w wynikach jest widoczny długi fragment kodu HTML, zawierający element `div` z formularzem i miejscem do wyświetlania komunikatu. Powyższy przykład pokazuje, w jaki sposób można sprawdzać, co komponent generuje oraz jak przetwarzać wygenerowany kod HTML.

<sup>2</sup> Słowo *describe* w języku angielskim oznacza: opisywać, przedstawiać — *przypr. tłum.*

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS
src/App.test.tsx
src/DisplayText.test.tsx
● Console

console.log
<div><form><div><label>Jak masz na imię?</label></div><div><input data-testid="user-input" value=""></div><div><button
data-testid="input-submit">Wyświelt komunikat</button></div><div><label data-testid="final-msg"></label></div></form></div>

    at Object.<anonymous> (src/DisplayText.test.tsx:11:13)

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:  0 total
Time:        8.614 s
Ran all test suites.

Watch Usage: Press w to show more.

```

Rysunek 6.12. Wyniki testu — wyświetlanie kodu HTML elementu w teście

Nasz drugi test także rozpoczyna się od funkcji `it`. Ten test sprawdza, czy element `input` został prawidłowo wyświetlony oraz pobiera wartość wpisaną w polu. Takie rozwiązanie może się wydawać dziwne, musimy jednak pamiętać, że w aplikacjach Reacta obsługa informacji wpisywanych w polach formularzy nie jest prosta i wiąże się z koniecznością obsługi zdarzeń `onChange`. Dlatego, aby pobrać konkretny element formularza (bo może ich przecież być więcej) przekazujemy w wywołaniu funkcji `getByTestId` wartość atrybutu `data-testid`. Aby wstawić zmienioną wartość, używamy funkcji `fireEvent.change`, która pobiera i wysyła łańcuch znaków. W końcu sprawdzamy, czy przesłana wartość pojawiła się w wynikach, czy nie. Poniżej przedstawiłem kod tego testu:

```

it("uzyskuje tekst z pola", () => {
  const testuser = "testuser";
  const { getByTestId } = render(<DisplayText />);
  const input = getByTestId("user-input");
  fireEvent.change(input, { target: { value:testuser } });
  expect(input).toBeInTheDocument();
  expect(input).toHaveValue(testuser);
});

```

3. A teraz stwórzmy jeszcze jeden tekst, który pokaże, w jaki sposób można całościowo przetestować komponent. W tym celu dodaj wewnątrz funkcji `describe` poniższy fragment kodu:

```

it("wyświetla komunikat powitalny", () => {
  const testuser = "testuser";
  const msg = `Witaj na zajęciach z testowania Reacta, ${testuser}!`;
  const { getByTestId } = render(<DisplayText />);
  const input = getByTestId("user-input");
  const label = getByTestId("final-msg");
  fireEvent.change(input, { target: { value: testuser } });
  const btn = getByTestId("input-submit");
  fireEvent.click(btn);

  expect(label).toBeInTheDocument();
  expect(label.innerHTML).toBe(msg);
});

```

Ten test jest podobny do poprzedniego, gdyż tak jak on wstawia wartość do pola tekstowego; jednak oprócz tego ten test pobiera odwołania do wyświetlanego przez komponent przycisku (button) oraz etykiety (label). W kolejnym kroku test tworzy zdarzenie click, aby zasymulować kliknięcie przycisku, które w normalnym kodzie spowodowałoby zapisanie w elemencie label komunikatu powitalnego. Następnie test sprawdza zawartość elementu label. Także tym razem, kiedy zapiszesz plik testu, wszystkie testy zostaną automatycznie uruchomione i wszystkie zostaną wykonane prawidłowo.

4. A teraz przyjrzymy się możliwości określanej jako migawki (ang. *snapshots*). Znaczna część prac związanych z tworzeniem aplikacji Reacta jest związana nie tylko z implementacją zachowań i akcji, lecz także z tworzeniem i wyświetlaniem interfejsu użytkownika. Testowanie migawek zapewnia nam właśnie możliwość sprawdzania, czy komponenty wygenerowały oczekiwany interfejs użytkownika, czyli elementy HTML. Dodaj poniższe wywołanie do pliku *DisplayText.test.tsx*, zapisując je bezpośrednio poniżej testu z opisem "jest wyświetlany bez problemów":

```
it("jest zgodny z migawką", () => {
  const { baseElement } = render(<DisplayText />);
  expect(baseElement).toMatchSnapshot();
});
```

Jak widać, w tym teście używamy funkcji render, by wygenerować główny element komponentu DisplayText, i odwołujemy się do niego używając właściwości baseElement. Oprócz tego, stosujemy w nim nową funkcję, toMatchSnapshot, którą połączyliśmy z wywołaniem funkcji expect. Ta funkcja realizuje kilka zadań:

- W momencie pierwszego wywołania, w katalogu src tworzy podkatalog o nazwie `__snapshot__`.
- Następnie dodaje do niego lub modyfikuje plik o tej samej nazwie co nazwa pliku testu, dodając na jej końcu rozszerzenie `.snap`. A zatem, migawka naszego pliku testowego będzie mieć nazwę `DisplayText.test.tsx.snap`.

Zawartością tego pliku będzie kod HTML wygenerowany przez nasz komponent. W naszym przypadku, zawartość tego pliku powinna przypominać tę przedstawioną poniżej:

*// Jest Snapshot v1, <https://goo.gl/fbAQLP>*

```
exports[~Test komponentu DisplayText jest zgodny z migawką 1~] = `
<body>
  <div>
    <form>
      <div>
        <label>
          Jak masz na imię?
        </label>
      </div>
      <div>
        <input
          data-testid="user-input"
          value=""
        >
      </div>
    </form>
  </div>
</body>
`
```

```

        />
    </div>
    <div>
        <button
            data-testid="input-submit"
        >
            Wyświetl komunikat
        </button>
    </div>
    <div>
        <label
            data-testid="final-msg"
        />
    </div>
</form>
</div>
</body>
`;

```

Jak widać, jest to dokładna kopia oczekiwanego kodu HTML wygenerowanego przez nasz komponent `DisplayText`. Zwróć także uwagę na zapisane na początku pliku opis oraz informację, że mamy do czynienia z migawką numer 1 (snapshot 1). Kiedy zaczniemy dodawać następne migawki, ta liczba będzie rosnąć.

5. No dobrze, a zatem utworzyliśmy migawkę i pierwsze wykonanie tego nowego testu zakończyło się pomyślnie. A teraz zobaczmy, co się stanie, kiedy zmienimy kod JSX komponentu `DisplayText`. Będziemy tu modyfikować kod zapisany w pliku `DisplayText.tsx`, nie w pliku testowym `DisplayText.test.tsx`, przy czym pierwsza część kodu komponentu zostaje bez zmian (aby nieco skrócić listing, zamieściłem na nim jedynie kod samego komponentu):

```

const DisplayText = () => {
    const [txt, setTxt] = useState("");
    const [msg, setMsg] = useState("");
    const onChangeTxt = (e: React.ChangeEvent<HTMLInputElement>) => {
        setTxt(e.target.value);
    }

    const onClickShowMsg = (e: React.MouseEvent<HTMLButtonElement,
↳ MouseEvent>) => {
        e.preventDefault();
        setMsg(`Witaj na zajęciach z testowania Reacta, ${txt}!`);
    }
}

```

Jak widać, ten fragment kodu pozostał taki sam — zmiany wprowadzimy jedynie w kodzie JSX zwracanym przez instrukcję `return`:

```

return (
    <form>
        <div>
            <label>Jak masz na imię?</label>
        </div>
        <div>
            <input data-testid="user-input"
                value={txt} onChange={onChangeTxt} />

```

```

    </div>
    <div>
      <button data-testid="input-submit"
        onClick={onClickShowMsg}>Wyświetl komunikat</button>
    </div>
    <div>
      <label data-testid="final-msg">{msg}</label>
    </div>
    <div>
      element na potrzeby testów
    </div>
  </form>
)
}

```

Kiedy zapiszesz plik i ponownie wykonasz testy, to zostaną wyświetlone wyniki podobne do tych z rysunku 6.13.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

> 16 |     expect(baseElement).toMatchSnapshot();
    |                                     ^
    |                                     A
17 |   });
18 |
FAIL src/DisplayText.test.tsx (6.294 s)
• Test komponentu DisplayText > jest zgodny z migawką

expect(received).toMatchSnapshot()

Snapshot name: `Test komponentu DisplayText jest zgodny z migawką 1`

- Snapshot  - 0
+ Received  + 3

@@ -22,8 +22,11 @@
  <div>
    <label
      data-testid="final-msg"
    />
  </div>
+ <div>
+   element na potrzeby testów
+ </div>
</div>
</form>
</div>
</body>

14 |   it("jest zgodny z migawką", () => {
15 |     const { baseElement } = render(<DisplayText />);
> 16 |     expect(baseElement).toMatchSnapshot();
    |                                     ^
    |                                     A
17 |   });
18 |
19 |   it("uzyskuje tekst z pola", () => {

at Object.<anonymous> (src/DisplayText.test.tsx:16:25)

> 1 snapshot failed.
src/App.test.tsx (6.075 s)

Snapshot Summary
> 1 snapshot failed from 1 test suite. Inspect your code changes or press `u` to update them.

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 failed, 4 passed, 5 total
Snapshots:  1 failed, 1 total
Time:        13.321 s
Ran all test suites.

Watch Usage: Press w to show more.

```

Rysunek 6.13. Test migawki zakończony niepowodzeniem

Jak widać, test migawki zakończył się niepowodzeniem, gdyż interfejs użytkownika komponentu zmienił się i nie odpowiada temu zapisanemu w migawce. No dobrze, a co, jeśli celowo wprowadziliśmy modyfikacje w interfejsie użytkownika komponentu `DisplayText`? W takim przypadku możemy wymusić aktualizację migawki. W tym celu musimy najpierw wyświetlić menu *Watch Usage* naciskając klawisz *w*, a następnie nacisnąć klawisz *u*. Poniżej, na rysunku 6.14, pokazałem, jak wygląda menu *Watch Usage*:

```
Watch Usage
> Press f to run only failed tests.
> Press o to only run tests related to changed files.
> Press u to update failing snapshots.
> Press i to update failing snapshots interactively.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.
```

Rysunek 6.14. Lista Watch Usage

6. Kiedy naciśniesz klawisz *u*, migawka zostanie zaktualizowana, a test — wykonany prawidłowo. Jeśli teraz otworzysz w edytorze plik migawki, przekonasz się, że znajdzie się w nim ten sam element `div`, który wcześniej dodałeś do komponentu.

Te kilka przykładów prostych tekstów, które przedstawiłem, powinny Ci ułatwić zrozumienie zagadnień opisanych w dalszej części rozdziału. Kolejnym tematem, którym się zajmiemy, będzie tworzenia atrap.

## Atrapy

Tworzenie atrap (ang. *mocking*) sprowadza się do zastępowania w testach określonych możliwości funkcjonalnych innymi. Przykładem takiego rozwiązania mogłoby być zasymulowanie wywołania sieciowego i zwrócenie zamiast faktycznego wyniku jakiejś konkretnej wartości. Podstawowym powodem stosowania atrap jest chęć testowania jedynie niewielkiego, określonego fragmentu naszego kodu. Tworząc atrapę pewnego fragmentu kodu, który nie jest bezpośrednio powiązany z kodem, który chcemy testować, możemy uniknąć zamieszania i zagwarantować, że nasze testy będą działały powtarzalnie i spójnie. Na przykład, jeśli chcemy testować kod związany z pobieraniem danych wejściowych, to nie chcielibyśmy, by przypadkowe awarie sieci wpływały na wyniki testów, gdyż awarie sieciowe nie mają nic wspólnego kodem odpowiedzialnym za sprawdzenie i przygotowanie danych wejściowych. W przypadku testów całościowych lub integracyjnych, zapewne będziemy chcieli testować także poprawność działania żądań sieciowych. Jednak takie testy to już zupełnie inna bajka i nie mają one wiele wspólnego z testami jednostkowymi (w niektórych firmach tymi rodzajami testów zajmują się zespoły kontroli jakości), dlatego też nie będziemy się nimi zajmować w tej książce. Wróćmy do komponentów Reacta oraz biblioteki `testing-library`; jej twórcy nie zalecają pisania i stosowania atrap, gdyż uważają, że przez



nie kod testów staje się mniej podobny do rzeczywistego kodu. Niezależnie od tego, w niektórych okolicznościach atrapy okazują się bardzo pomocne, dlatego przedstawię kilka przykładów pokazujących jak tworzyć atrapy komponentów oraz jak ich używać.

## Tworzenie atrap z wykorzystaniem jest.fn

W tym punkcie rozdziału zajmiemy się tworzeniem atrap z wykorzystaniem biblioteki Jest, gdyż jest ona używana także podczas pisania aplikacji dla środowiska Node. Pierwszą z możliwości zapewnianych przez tę bibliotekę jest tworzenie atrap funkcji. Do tego celu służy funkcja `fn`. Jej parametrem jest kolejna funkcja, która robi wszystko, co chcemy, by robiła tworzona atrapa. Oprócz możliwości zastąpienia dowolnego istniejącego kodu oraz wartości, utworzenie atrapy zapewnia nam także dostęp do składowej o nazwie `mock`, a ona z kolei daje dostęp do przeróżnych metryk zastępowanego wywołania. Ponieważ są to zagadnienia, które dość trudno sobie wyobrazić, przedstawię je na przykładzie:

1. Zaktualizujemy teraz nasz komponent `DisplayText` w taki sposób, by wykonywał żądanie skierowane do internetowego API. Konkretnie skorzystamy z API **JsonPlaceholder**. Jest to bezpłatna usługa udostępniająca API, które zwraca obiekty w formacie JSON. Zaczniemy od utworzenia w komponencie `DisplayText` nowej właściwości, która będzie zawierać funkcję zwracającą pełne imię i nazwisko użytkownika na podstawie jego nazwy. W pierwszej kolejności musimy zacząć od zaktualizowania funkcji `App` w sposób przedstawiony na poniższym przykładzie:

```
function App() {
  const getUserFullName = async (username: string): Promise<string> => {
    const usersResponse = await fetch('https://jsonplaceholder.typicode.com/
    ↪users');
    if(usersResponse.ok) {
      const users = await usersResponse.json();
      const userByName = users.find((usr: any) => {
        return usr.username.toLowerCase() === username;
      });
      return userByName.name;
    }
    return "";
  }
  return (
    <div className="App">
      <DisplayText getUserFullName={getUserFullName} />
    </div>
  );
}
```

Jak widać, wewnątrz funkcji `App` utworzyliśmy kolejną funkcję, o nazwie `getUserFullName`, którą następnie przekazaliśmy jako właściwość do komponentu `DisplayText`. Działanie tej nowej funkcji bazuje na wykonaniu żądania sieciowego skierowanego do API usługi `JsonPlaceholder`. Żądanie to odwołuje się do kolekcji `users`, która po pobraniu jest filtrowana przy użyciu standardowej funkcji `find` języka JavaScript. Wynikiem zwracanym przez funkcję `getUserFullName` jest wartość wyrażenia `userByName.name`, czyli pełne imię i nazwisko użytkownika wybranego na podstawie nazwy.

## 2. Teraz musimy zmodyfikować kod komponentu DisplayText:

```
import React, { useState, FC } from "react";

interface DisplayTextProps {
  getUserFullName: (username: string) => Promise<string>;
}

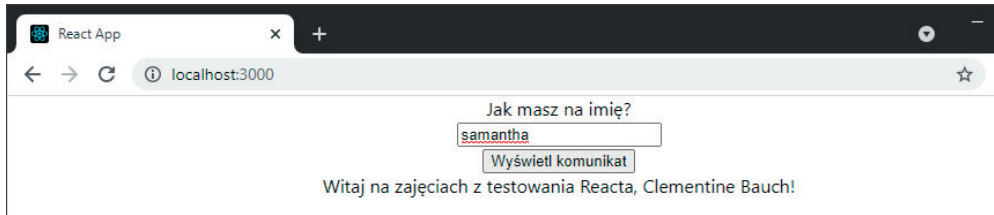
const DisplayText: FC<DisplayTextProps> = ({ getUserFullName }) => {
  const [txt, setTxt] = useState("");
  const [msg, setMsg] = useState("");
  const onChangeTxt = (e: React.ChangeEvent<HTMLInputElement>) => {
    setTxt(e.target.value);
  }

  const onClickShowMsg = async (e: React.MouseEvent<HTMLButtonElement,
    ↳MouseEvent>) => {
    e.preventDefault();
    setMsg(`Witaj na zajęciach z testowania Reacta, ${await
      getUserFullName(txt)}!`);
  }
}
```

Znaczna część tego kodu pozostała niezmienną, dodaliśmy jednak do niego interfejs `DisplayTextProps` deklarujący funkcję `getUserFullName`. Funkcja ta będzie przekazywana z komponentu `App` jako właściwość *props*. Następnie używamy tej funkcji w procedurze obsługi zdarzeń `onClickShowMsg`, aby wyświetlić komunikat powitalny zawierający pełne imię i nazwisko użytkownika.

```
return (
  <form>
    <div>
      <label>Jak masz na imię?</label>
    </div>
    <div>
      <input data-testid="user-input"
        value={txt} onChange={onChangeTxt} />
    </div>
    <div>
      <button data-testid="input-submit"
        onClick={onClickShowMsg}>Wyświetl komunikat</button>
    </div>
    <div>
      <label data-testid="final-msg">{msg}</label>
    </div>
    <div>
      element na potrzeby testów
    </div>
  </form>
)
}
export default DisplayText;
```

Pozostała część kodu nie zmieniła się, jednak zamieściłem ją tutaj, by kod przykładu był kompletny. Jeśli teraz uruchomimy aplikację i wpiszemy nazwę użytkownika, to powinniśmy zobaczyć wyniki podobne do tych przedstawionych na rysunku 6.15.



Rysunek 6.15. Wyświetlanie pełnego imienia i nazwiska użytkownika

Jak widać, imię i nazwisko użytkownika o nazwie **samantha** to **Clementine Bauch**.

A teraz napiszmy test i spróbujmy napisać atrapę, która będzie udawać wykonywanie zapytania sieciowego; użyjemy do tego celu biblioteki Jest:

1. Otwórz plik *DisplayText.test.tsx* i zwróć uwagę na to, że we wszystkich testach są wyświetlone błędy, gdyż w żadnym z nich do komponentu `DisplayText` nie jest przekazywana nowa właściwość `getUserFullName`. Musimy zatem zaktualizować kod testu i przygotować atrapę tej funkcji. Poniżej przedstawiłem zmodyfikowaną sekcję importu:

```
import React from "react";
import { render, fireEvent, cleanup, wait } from "@testing-library/react";
import DisplayText from "../DisplayText";
import "@testing-library/jest-dom/extend-expect";
```

Jak widać, teraz dodatkowo importujemy do testu funkcję `wait` z modułu `@testing-library/react`. Funkcja ta służy do obsługi wywołań asynchronicznych w testach. Na przykład funkcja `getUserFullName` jest asynchroniczna, więc jej wywołania trzeba poprzedzać słowem kluczowym `await`. Jeśli nie użyjemy słowa kluczowego `await`, wszystkie testy zakończą się niepowodzeniem, gdyż realizacja testu od razu przejdzie do kolejnej instrukcji, nie czekając na zakończenie wywołania:

```
describe("Test komponentu DisplayText", () => {
  const userFullName = "Janek Tester";

  const getUserFullNameMock = (
    username: string
  ): Promise<string>, jest.Mock<Promise<string>, [string]>] => {
    const promise = new Promise<string>((res, rej) => {
      res(userFullName);
    });
    const getUserFullName = jest.fn(
      async (username: string): Promise<string> => {
        return promise;
      }
    );
```

```
);

    return [promise, getUserFullName];
};
```

Kolejnymi zmianami wprowadzonymi w kodzie jest dodanie dwóch nowych składowych: `userFullName` oraz `getUserFullNameMock`. Naszej atrapy funkcji będziemy używali w kilku testach, dlatego tworzymy funkcję `getUserFullNameMock`, dzięki czemu będziemy mogli wielokrotnie używać jej do pobierania atrapy funkcji `getUserFullName` i kilku innych, potrzebnych elementów.

Powstaje jednak pytanie: dlaczego ten kod jest tak bardzo skomplikowany? Przeanalizujmy zatem dokładnie, co się w nim dzieje.

- Po określeniu wartości zmiennej `userFullName` definiujemy funkcję `getUserFullNameMock`. Jak widać, funkcja ta pobiera jeden parametr, `username`, dokładnie tak samo jak rzeczywista funkcja `getUserFullName`, i zwraca obiekt obietnicy, `promise`, oraz obiekt `Mock`.
- Wewnątrz tej funkcji tworzymy obiekt `Promise` i zapisujemy go w zmiennej `promise`, a następnie tworzymy atrapę funkcji `getUserFullName`, używając do tego celu funkcji `jest.fn`. Obietnica tworzona w tej funkcji jest nam potrzebna do zasymulowania żądania sieciowego, a także po to, by później móc wywoływać atrapę używając słowa kluczowego `await` i funkcji `wait` biblioteki `testing-library`.
- Jak już wspominałem, aby utworzyć atrapę oraz aby atrapa ta robiła to, co chcemy, musimy użyć funkcji `jest.fn`. W naszym przypadku funkcja `getUserFullName`, której atrapę tworzymy, wykonuje żądanie sieciowe, a to oznacza, że funkcja `jest.fn` musi zwracać obietnicę. Co też robi — zwraca zmienną `promise`, którą utworzyliśmy na samym początku kodu funkcji `getUserFullNameMock`.
- Na samym końcu kodu funkcji zwracamy wyniki: obiekt `promise` oraz nową atrapę: funkcję `getUserFullName`.
- Tworząc tę atrapę, zadajemy sobie sporo trudu, jednak w tym przypadku naprawdę warto wyeliminować wolne i podatne na błędy żądania sieciowe. Gdybyśmy tego nie zrobili, to w przypadku wystąpienia problemów podczas wykonywania żądania sieciowego, moglibyśmy błędnie uznać, że zawiodły nasz test i kod.
- A teraz zobaczmy, w jaki sposób można używać tak przygotowanej atrapy funkcji w testach:

```
it("jest wyświetlany bez problemów", () => {
  const username = "testuser";
  const [promise, getUserFullName] = getUserFullNameMock(username);

  const { baseElement } = render(<DisplayText getUserFullName=
    ↪{getUserFullName} />);
  expect(baseElement).toBeInTheDocument();
});
```

```

it("jest zgodny z migawką", () => {
  const username = "testuser";
  const [promise, getUserFullName] = getUserFullNameMock(username);

  const { baseElement } = render(<DisplayText getUserFullName=
    ↪{getUserFullName} />);
  expect(baseElement).toMatchSnapshot();
});

it("uzyskuje tekst z pola", () => {
  const username = "testuser";
  const [promise, getUserFullName] = getUserFullNameMock(username);

  const { getByTestId } = render(<DisplayText getUserFullName=
    ↪{getUserFullName} />);
  const input = getByTestId("user-input");
  fireEvent.change(input, { target: { value: username } });
  expect(input).toBeInTheDocument();
  expect(input).toHaveValue(username);
});

```

W pierwszych trzech testach po prostu pobieramy funkcję `getUserFullName` i przekazujemy ją jako właściwość do komponentu `DisplayText`. Te testy nie używają funkcji w żaden sposób, niemniej jednak wciąż jest ona potrzebna, gdyż jest wymaganą właściwością komponentu `DisplayText`.

2. W ostatnim teście musimy wprowadzić nieco więcej zmian, gdyż jest w nim sprawdzany komunikat powitalny. Zmodyfikuj ten test zgodnie z poniższym przykładem:

```

it("wyświetla komunikat powitalny", async () => {
  const username = "testuser";
  const [promise, getUserFullName] = getUserFullNameMock(username);

  const msg = `Witaj na zajęciach z testowania Reacta, ${userFullName}!`;
  const { getByTestId } = render(<DisplayText getUserFullName=
    ↪{getUserFullName} />);
  const input = getByTestId("user-input");
  const label = getByTestId("final-msg");
  fireEvent.change(input, { target: { value: username } });
  const btn = getByTestId("input-submit");
  fireEvent.click(btn);

  expect(label).toBeInTheDocument();
  await wait(() => promise);
  expect(label.innerHTML).toBe(msg);
});

```

Ten ostatni test sprawdza komunikat powitalny, gdyż funkcja `getUserFullName` zwraca pełne imię i nazwisko użytkownika, i to właśnie te informacje zostaną wyświetlone w etykiecie. Aby to sprawdzić, używamy asercji konstruowanej przy użyciu funkcji `expect` oraz `toBe`. Oprócz tego musisz zwrócić uwagę na to,

że wywołanie `await wait` jest umieszczone przed wywołaniem funkcji `toBe`. To konieczne, gdyż nasza funkcja `getUserFullName` jest funkcją asynchroniczną i aby pobrać zwracany przez nią wynik, konieczne jest użycie słowa kluczowego `await`.

Innymi słowy, stosując funkcję `jest.fn` można utworzyć atrapę pewnego fragmentu kodu, dzięki czemu będzie on zawsze zwracał spójne wyniki. W ten sposób możemy tworzyć spójne, powtarzalne testy, które będą sprawdzać działanie ściśle określonego fragmentu kodu.

## Tworzenie atrap komponentów

Drugim sposobem stosowania atrap jest zastępowanie całych komponentów i stosowanie takich atrap zamiast rzeczywistych komponentów w kodzie, który chcemy testować. Na poniższej liście opisałem czynności, które pozwolą Ci przygotować taką atrapę i użyć jej:

1. Zmodyfikujmy nasz komponent `DisplayText` w taki sposób, by wyświetlał listę zadań do zrobienia przypisanych użytkownikowi o podanej nazwie. Zaktualizuj kod komponentu według poniższego listingu:

```
import React, { useState, FC } from "react";

interface DisplayTextProps {
  getUserFullName: (username: string) => Promise<string>;
}

const DisplayText: FC<DisplayTextProps> = ({ getUserFullName }) => {
  const [txt, setTxt] = useState("");
  const [msg, setMsg] = useState("");
  const [todos, setTodos] = useState<Array<JSX.Element>>();
```

Teraz utworzymy stan komponentu, którego będziemy używali później:

```
const onChangeTxt = (e: React.ChangeEvent<HTMLInputElement>) => {
  setTxt(e.target.value);
}
```

W kolejnym fragmencie kodu ustawiamy komunikat na podstawie danych wpisanych przez użytkownika w polu tekstowym:

```
const onClickShowMsg = async (e:
  React.MouseEvent<HTMLButtonElement, MouseEvent>) => {
  e.preventDefault();
  setMsg(`Witaj na zajęciach z testowania Reacta,
    ${await getUserFullName(txt)}!`);
  setUsersTodos();
}
```

Po kliknięciu przycisku *Wyświetl komunikat* aktualizujemy zarówno komunikat wyświetlany na stronie, jak i listę zadań do zrobienia.

2. Będziemy pobierać właściwość *props*, której użyjemy jako prefiksu dla wyświetlanego komunikatu:

```
const setUsersTodos = async () => {
  const usersResponse = await
    fetch('https://jsonplaceholder.typicode.com/users');
```

```

if(usersResponse.ok) {
  const users = await usersResponse.json();
  const userByName = users.find((usr: any) => {
    return usr.username.toLowerCase() === txt;
  });
  console.log("użytkownik pobrany po nazwie", userByName);
}

```

W kolejnym fragmencie kodu korzystamy z API serwisu JSONPlaceholder, by pobrać listę zadań; używamy przy tym podobnego rozwiązania, z którego już wcześniej korzystaliśmy do pobierania pełnego imienia i nazwiska użytkownika (fullname) na podstawie jego nazwy (username). Zaczynamy od znalezienia użytkownika, którego pobieramy z kolekcji users.

```

const todosResponse = await
  fetch('https://jsonplaceholder.typicode.com/todos');
if(todosResponse.ok) {
  const todos = await todosResponse.json();
  const usersTodos = todos.filter((todo:any) => {
    return todo.userId === userByName.id;
  });
  const todoList = usersTodos.map((todo:any) => {
    return <li key={todo.id}>
      {todo.title}
    </li>
  });
  setTodos(todoList);
  console.log("lista zadań użytkownika", usersTodos);
}
}
}

```

Po czym pobieramy z usługi kolekcję todos, a z niej zadania przypisane do znalezionej wcześniej użytkownika.

3. I w końcu w interfejsie użytkownika komponentu wyświetlamy wypunktowaną listę zadań do zrobienia:

```

return (
  <form>
    <div>
      <label>Jak masz na imię?</label>
    </div>
    <div>
      <input data-testid="user-input"
        value={txt} onChange={onChangeTxt} />
    </div>
    <div>
      <button data-testid="input-submit"
        onClick={onClickShowMsg}>Wyświetl komunikat</button>
    </div>
    <div>
      <label data-testid="final-msg">{msg}</label>
    </div>
    <ul style={{marginTop: '1rem', listStyleType:'none'}}>
      {todos}
    </ul>
  </form>
)

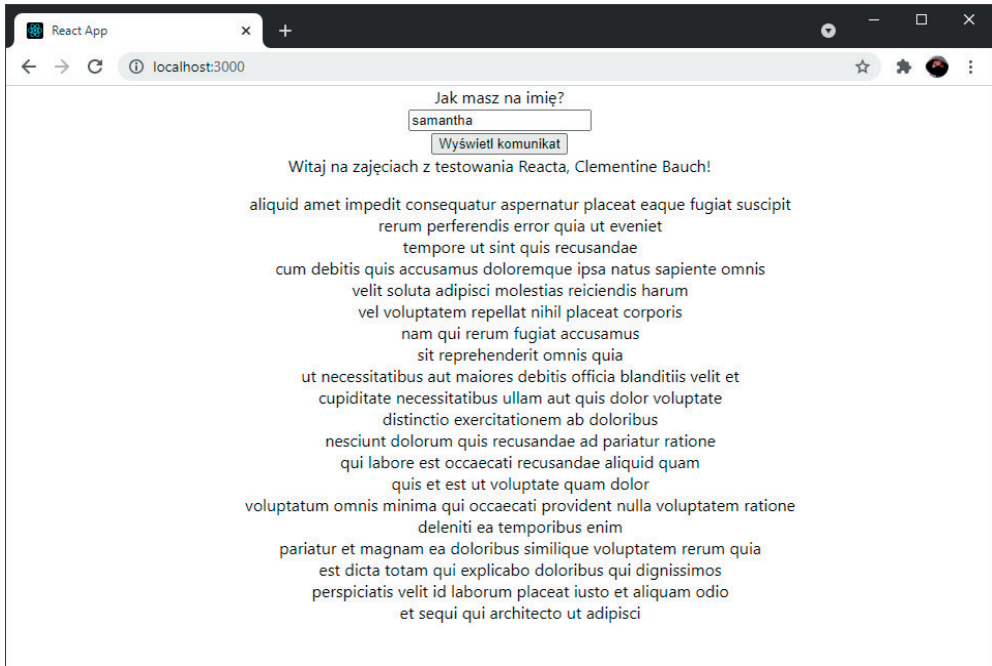
```

```

    )
  }
</form>
}

```

Na rysunku 6.16 pokazałem, jak ta nowa wersja komponentu powinna wyglądać w przeglądarce. Zwróć uwagę na to, że zwrócone zadania to fragmenty przykładowego tekstu *lorem ipsum* — to jedynie powszechnie stosowany tekst zastępczy. Jest on pobierany prosto z internetowego API usługi JSONPlaceholder:



**Rysunek 6.16.** Na stronie zostały wyświetlone zadania użytkownika samantha

A teraz przyjmijmy, że chcemy przetestować nasz komponent `DisplayText`, jednak bez testowania listy zadań do zrobienia. W jaki sposób możemy zrefaktoryzować kod komponentu, by nie był tak monolityczny, jak obecnie? Możemy na przykład przenieść możliwości funkcjonalne związane z wyświetlaniem listy zadań do odrębnego komponentu. Na kolejnej liście opisałem, jak to zrobić:

1. Dostosuj kod komponentu `DisplayText` do poniższego listingu:

```

import React, { useState, FC } from "react";
import UserTodos from "../UserTodos";

interface DisplayTextProps {
  getUserFullname: (username: string) => Promise<string>;
}

const DisplayText: FC<DisplayTextProps> = ({ getUserFullname }) => {
  const [txt, setTxt] = useState("");
  const [msg, setMsg] = useState("");

```



```
const [todoControl, setTodoControl] =
  useState<ReturnType<typeof UserTodos>>();

const onChangeTxt = (e: React.ChangeEvent<HTMLInputElement>) => {
  setTxt(e.target.value);
}
```

W pierwszej kolejności tworzymy właściwość stanu o nazwie `todoControl`. Jej typem jest typ naszego nowego komponentu `UserTodos`, którym zajmiemy się już niebawem. Ten typ pobieramy, używając pomocniczego typu o nazwie `ReturnType`. Jak widać, jest to bardzo prosty sposób tworzenia definicji typu na podstawie obiektu.

```
const onClickShowMsg = async (e:
  React.MouseEvent<HTMLButtonElement, MouseEvent>) => {
  e.preventDefault();
  setMsg(`Witaj na zajęciach z testowania Reacta, ${await
    getUserFullname(txt)}!`);
  setTodoControl(<UserTodos username={txt} />);
}
```

Procedura obsługi zdarzeń `onClickShowMsg` będzie wywoływać funkcję `setTodoControl`, przekazując do niego komponent z nazwą użytkownika podaną jako właściwość `username`.

```
return (
  <form>
    <div>
      <label>Jak masz na imię?</label>
    </div>
    <div>
      <input data-testid="user-input"
        value={txt} onChange={onChangeTxt} />
    </div>
    <div>
      <button data-testid="input-submit"
        onClick={onClickShowMsg}>Wyświetl komunikat</button>
    </div>
    <div>
      <label data-testid="final-msg">{msg}</label>
    </div>
    <ul style={{marginTop: '1rem', listStyleType: 'none'}}>
      {todoControl}
    </ul>
  </form>
)
}

export default DisplayText;
```

I w końcu, wyświetlamy komponent `todoControl` w interfejsie użytkownika generowanym przez komponent `DisplayText`.

2. Teraz zajmiemy się utworzeniem nowego komponentu, `UserTodos`. Utwórz plik `UserTodos.tsx` i zapisz w nim następujący kod:

```
import React, { FC, useState, useEffect } from "react";

interface UserTodosProps {
  username: string;
}
```

Teraz pobieramy nazwę użytkownika jako właściwość *props* z komponentu nadrzędnego.

```
const UserTodos: FC<UserTodosProps> = ({ username }) => {
  const [todos, setTodos] = useState<Array<JSX.Element>>();

  const setUsersTodos = async () => {
    console.log("wewnątrz faktycznej funkcji UserTodos.setUsersTodos");
    const usersResponse = await fetch(
      "https://jsonplaceholder.typicode.com/users"
    );
    if (usersResponse.ok) {
      const users = await usersResponse.json();
      const userByName = users.find((usr: any) => {
        return usr.username.toLowerCase() === username;
      });
      console.log("użytkownik pobrany po nazwie", userByName);
    }
  };
}
```

Podobnie jak wcześniej, w pierwszej kolejności pobieramy użytkowników z kolekcji *users*, a następnie odnajdujemy w niej interesującego nas użytkownika na podstawie wartości przekazanej jako właściwość *props* *username*:

```
const todosResponse = await fetch(
  "https://jsonplaceholder.typicode.com/todos"
);
if (userByName && todosResponse.ok) {
  const todos = await todosResponse.json();
  const usersTodos = todos.filter((todo: any) => {
    return todo.userId === userByName.id;
  });
  const todoList = usersTodos.map((todo: any) => {
    return <li key={todo.id}>
      {todo.title}
    </li>;
  });
  setTodos(todoList);
  console.log("lista zadań użytkownika", usersTodos);
}
};
```

Następnie pobieramy zadania pasujące do odszukanego użytkownika. Po pobraniu zadań wywołujemy funkcję *map* JavaScriptu, aby przekształcić zadania na kolekcję elementów *li*.

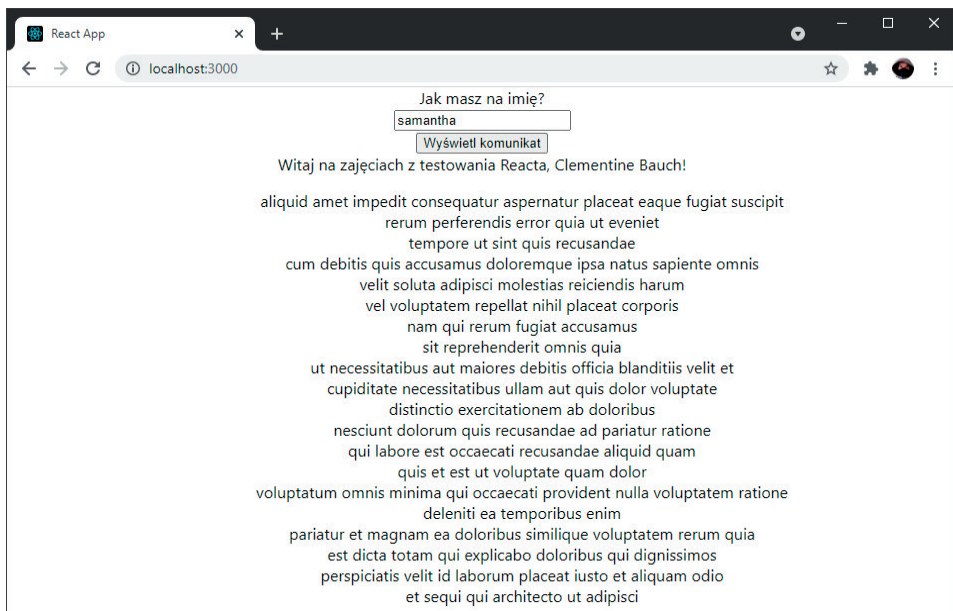
```
useEffect(() => {
  if (username) {
    setUsersTodos();
  }
}, [username]);
```

Używając funkcji `useEffect` zaznaczamy, że lista zadań ma być aktualizowana za każdym razem, gdy zmieni się właściwość `props.username`.

```
return <ul style={{ marginTop: "1rem", listStyleType: "none" }}>
  {todos}
</ul>;
};

export default UserTodos;
```

I w końcu, wyświetlamy zadania w formie listy wypunktowanej. Kiedy uruchomisz ten nowy komponent klikając przycisk *Wyświetl komunikat*, zobaczysz wyniki podobne do tych, które przedstawiłem na rysunku 6.17.



**Rysunek 6.17.** Lista zadań po refaktoryzacji komponentów

No dobrze, teraz już możemy dodać nowy test, który wykorzystuje atrapę komponentu `UserTodos`, umożliwiając tym samym niezależne przetestowanie komponentu `DisplayText`. Powinieneś także pamiętać, że istnieją dwa podstawowe sposoby tworzenia atrapy w bibliotece Jest. Atrapy można tworzyć bezpośrednio w kodzie, który z nich korzysta, bądź też można je umieszczać w odrębnych plikach. W tym przykładzie skorzystamy z tej drugiej możliwości, czyli przygotujemy plik atrapy. W tym celu wykonaj następujące czynności:

1. W katalogu `src` utwórz podkatalog o nazwie `__mocks__`. Wewnątrz tego podkatalogu utwórz plik `UserTodos.tsx` i zapisz w nim następujący kod:

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

export default (): ReactElement => {
  return <></>;
};
```

Ten plik stanowi atrapę komponentu funkcyjnego. Jak widać, ta atrapa komponentu nic nie zwraca i nie ma żadnych rzeczywistych składowych. Oznacza to, że w odróżnieniu od rzeczywistego komponentu, nie będzie ona wykonywać żadnych żądań sieciowych, ani generować żadnego kodu HTML; a w przypadku testów, to właśnie na takim zachowaniu nam zależy.

2. A teraz zmodyfikuj kod pliku *DisplayText.test.tsx* zgodnie z poniższym kodem:

```
import React from "react";
import { render, fireEvent, cleanup, wait } from "@testing-library/react";
import DisplayText from "../DisplayText";
import "@testing-library/jest-dom/extend-expect";

jest.mock("../UserTodos");

describe("Test komponentu DisplayText", () => {
  const userFullName = "Janek Tester";

  const getUserFullnameMock = (
    username: string
  ): [Promise<string>, jest.Mock<Promise<string>, [string]>] => {
    const promise = new Promise<string>((res, rej) => {
      res(userFullName);
    });
    const getUserFullname = jest.fn(
      async (username: string): Promise<string> => {
        return promise;
      }
    );

    return [promise, getUserFullname];
  };
});
```

W pierwszej kolejności zauważ, że jeszcze przed jakimikolwiek testami importujemy atrapę komponentu UserTodos. To konieczne, gdyż atrapa zaimportowana wewnątrz testów nie działałaby poprawnie.

Same testy pozostają takie same jak były, lecz podczas wykonywania będą teraz korzystały z atrapy komponentu UserTodos. To oznacza, że testy będą działać szybciej, gdyż nie będą musiały wykonywać żądań sieciowych. W ramach sprawdzenia nabytych umiejętności testowania komponentów spróbuj samodzielnie przygotować odrębne testy dla komponentu UserTodos.

W tym podrozdziale poznałeś sposoby testowania komponentów Reacta przy użyciu bibliotek Jest oraz testing-library. Pisanie i stosowanie testów jednostkowych jest bardzo ważnym aspektem tworzenia aplikacji i jako profesjonalny programista będziesz takie testy pisał niemal każdego dnia. Pomagają one nie tylko w pisaniu kodu, lecz także w jego refaktoryzacji.

W następnym podrozdziale znowu powiększysz swoje doświadczenie programistyczne, tym razem wzbogacając je o znajomość narzędzi najczęściej używanych podczas pisania aplikacji Reacta.

# Prezentacja najpopularniejszych narzędzi oraz praktyk tworzenia aplikacji Reacta

Istnieje wiele narzędzi, które mogą pomagać w tworzeniu aplikacji Reacta. Jest ich zbyt wiele, bym mógł opisać je wszystkie szczegółowo, jednak pokrótce przedstawię te najpopularniejsze i najczęściej stosowane. Narzędzia te mają kluczowe znaczenie dla pisania i debugowania kodu, dlatego też zachęcam, byś poświęcił nieco czasu na zapoznanie się z nimi.

## Visual Studio Code

Programu VSCode używaliśmy jako wybranego edytora do pisania kodu już od samego początku tej książki. Jeśli chodzi o pisanie kodu w języku JavaScript, VSCode jest obecnie bezsprzecznie najczęściej stosowanym edytorem. Poniżej zamieściłem kilka informacji, które pozwolą Ci korzystać z niego w optymalny sposób.

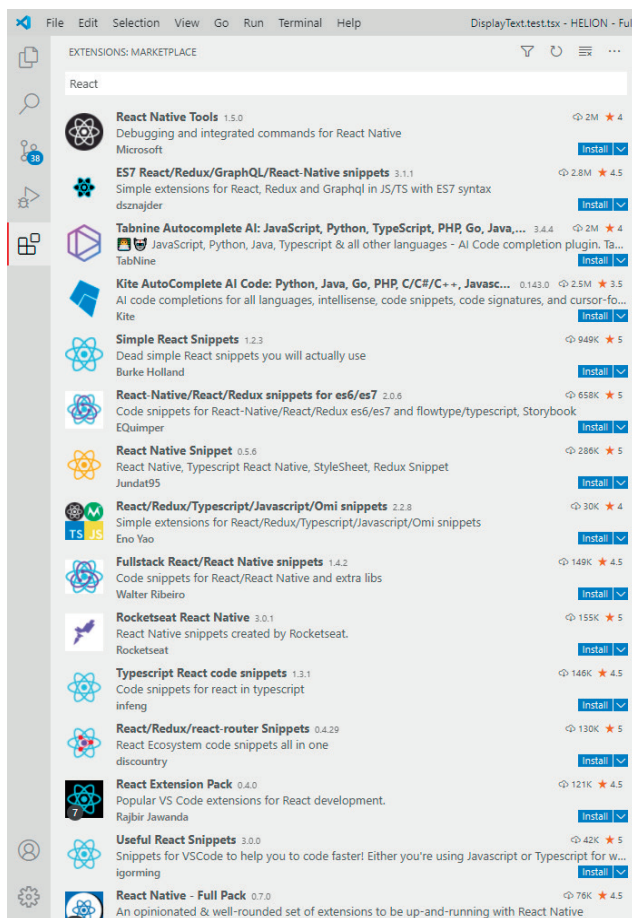
- VSCode dysponuje bardzo rozbudowanym ekosystemem rozszerzeń ułatwiających pisanie kodu. Wybór i stosowanie wielu z nich jest zależne wyłącznie od preferencji programisty, jednak warto przeszukać dostępne rozszerzenia i zapoznać się z ich możliwościami. Istnieje jednak kilka popularnych rozszerzeń, których stosowanie zdecydowanie powinieneś rozważyć:

**Visual Studio IntelliCode** — udostępnia serwer języków zapewniający możliwości uzupełniania kodu i kolorowania składni, korzystający z mechanizmów sztucznej inteligencji.

**Apollo GraphQL** — zapewnia mechanizm uzupełniania kodu i wspomaga formatowania kodu GraphQL.

**Wtyczki związane z Reactem** — dostępnych jest bardzo wiele rozszerzeń związanych z pisaniem aplikacji Reacta; udostępniają one często stosowane fragmenty kodu, a także narzędzia ułatwiające korzystanie z takich serwisów jak NPM. Na rysunku 6.18 przedstawiłem wyniki wyszukiwania słowa „React” na liście rozszerzeń VSCode:

- Edytor VSCode dysponuje wbudowanym debuggerem, pozwalającym na wstrzymywanie realizacji kodu i przeglądanie wartości zmiennych. Nie będę go tu jednak przedstawiał, gdyż w przypadku aplikacji internetowych standardem jest korzystanie z debuggера przeglądarki Chrome, który także dysponuje analogicznymi możliwościami. Debugger VSCode przedstawię nieco później, kiedy zajmiemy się pisaniem aplikacji dla środowiska Node.
- Pliki konfiguracyjne. W przypadku korzystania z VSCode istnieją dwa podstawowe sposoby określania ustawień konfiguracyjnych projektu: można je zapisywać na poziomie przestrzeni roboczej (ang. *workspace*) oraz w pliku *settings.json*. VSCode zapewnia niezwykle rozbudowane możliwości konfiguracji i pozwala modyfikować bardzo wiele aspektów swojego wyglądu i działania, takich jak: czcionki, używane rozszerzenia, temat kolorystyczny itd. Te wszystkie ustawienia



Rysunek 6.18. Rozszerzenia VSCode związane z Reactem

konfiguracyjne można zapisywać globalnie, bądź też mogą się one odnosić tylko do wybranego projektu. W ramach demonstracji tych możliwości, w przykładach dołączonych do książki możesz znaleźć przykładowy plik *settings.json*; jest on umieszczony w katalog *ejected-app*, w podkatalogu *.vscode*. Pliki przestrzeni roboczych to w zasadzie te same pliki ustawień, jednak są one przeznaczone do stosowania w kilku różnych projektach umieszczonych w tym samym katalogu. Nazwy plików przestrzeni roboczych są określane według następującego wzoru: *<nazwa>.code-workspace*.

## Prettier

Podczas pisania kodu bardzo ważne jest, by konsekwentnie stosować spójny styl — w ten sposób możemy bowiem poprawić czytelność kodu. Na przykład, jeśli wyobrazimy sobie dużą firmę zatrudniającą wielu programistów, to gdyby każdy z nich podczas pisania kodu

stosował własny styl, używając preferowanej przez siebie wielkości wciąg, swojego sposobu określania nazw zmiennych itp., itd., to w kodzie zapanowałby chaos. Co więcej, istnieją pewne branżowe wytyczne dotyczące sposobu formatowania kodu pisanego w języku JavaScript, które pozwalają poprawiać jego czytelność, a co za tym idzie, ułatwiają jego analizę. Właśnie takie możliwości zapewnia Prettier.

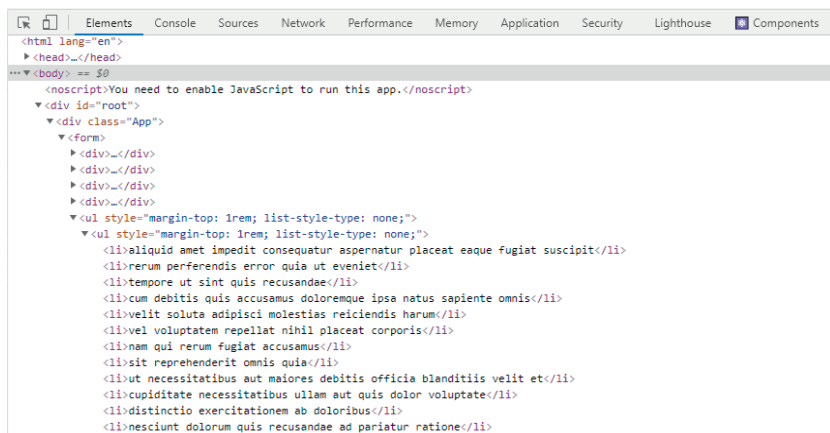
Prettier automatycznie formatuje nasz kod podczas zapisywania plików, nadając mu spójną, czytelną postać, niezależnie do tego, kto ten kod napisał. Musisz tylko pamiętać, by po zainstalowaniu tego narzędzia odpowiednio je skonfigurować w pliku *settings.json* lub pliku prze-strzeni roboczej. Przykładowy plik *settings.json*, który umieściłem w katalogu *ejected-app*, zawiera odpowiednie ustawienia.

## Debugger Chrome

Przeglądarka Chrome udostępnia wbudowane narzędzia dla programistów. Pozwalają one — między innymi — na przeglądanie kodu HTML aktualnie wyświetlanej strony, prezentowanie komunikatów wyświetlanych na konsoli, wstrzymywanie wykonywania kodu JavaScript, czy też przeglądanie wszystkich żądań sieciowych generowanych przez przeglądarkę. Nawet bez stosowania żadnych wytyczek, te narzędzia dla programistów zapewniają naprawdę rozbudowane możliwości. Dla bardzo wielu programistów aplikacji internetowych, Chrome jest podstawowym narzędziem do debugowania kodu.

Przyjrzyjmy się zatem temu debuggerowi Chrome i na przykładzie aplikacji *ejected-app* poznamy podstawowe sposoby jego obsługi:

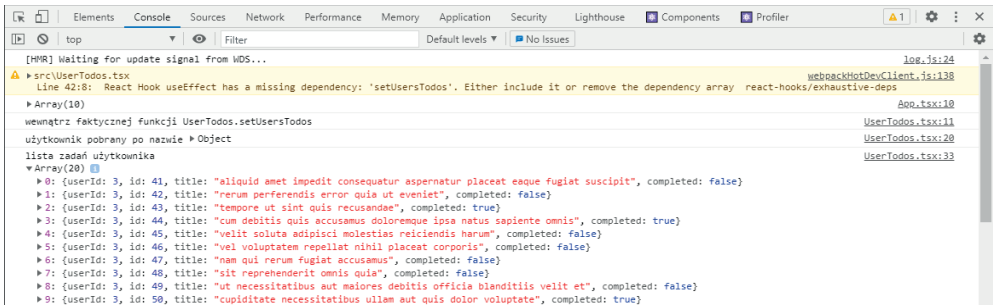
1. Jeśli aplikacja *ejected-app* na twoim komputerze aktualnie nie działa, to ją uruchom i wyświetl w przeglądarce Chrome stronę o adresie *localhost:3000*. Kiedy już strona zostanie wyświetlona, otwórz narzędzia dla programistów przeglądarki Chrome; w tym celu naciśnij klawisz *F12* lub z menu wybierz opcję *Więcej narzędzi/Narzędzia dla programistów*. W debuggerze, który najprawdopodobniej zostanie wyświetlony u dołu okna przeglądarki, powinna być widoczna karta *Elements* (którą pokazałem na rysunku 6.19).



Rysunek 6.19. Karta Elements debugera przeglądarki Chrome

Jak widać, na karcie jest widoczny element `div` o identyfikatorze `root`, stanowiący element główny, wewnątrz którego znajduje się cała zawartość aplikacji. Rysunek przedstawia sytuację po wykonaniu żądania do API i pobraniu listy zadań dla użytkownika `samantha`. A zatem, możemy używać debuggera Chrome do wyszukiwania elementów HTML, sprawdzania ich atrybutów oraz modyfikowania wartości właściwości CSS, co pozwoli nam nadać interfejsowi użytkownika dokładnie taki wygląd, jaki byśmy chcieli mieć.

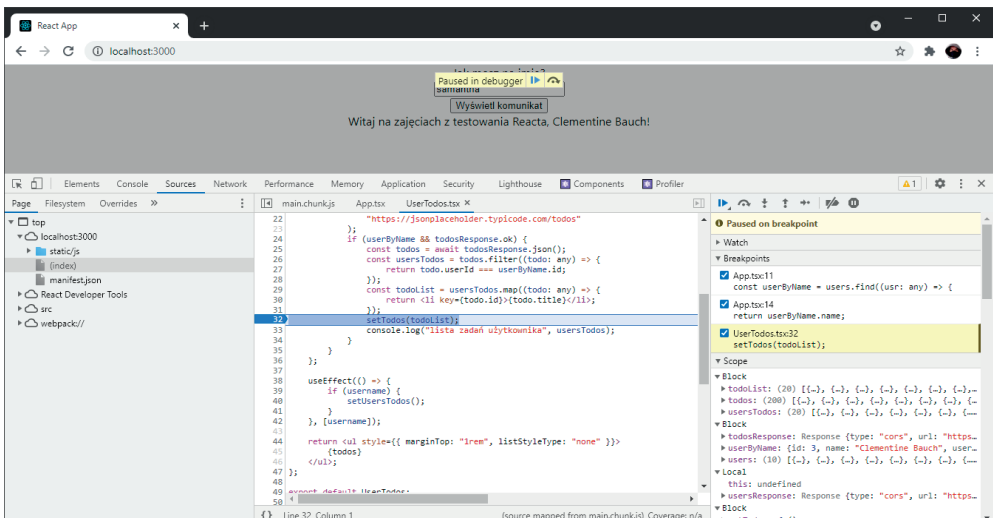
- Następnie przejdź na kartę **Console**, której przykładową zawartość przedstawiłem na rysunku 6.20.



Rysunek 6.20. Karta Console debuggera przeglądarki Chrome

Na tej karcie możemy sprawdzać wartości zmiennych oraz wyniki zwracane przez funkcje i upewniać się, czy odpowiadają one naszym oczekiwaniom.

- Debugger Chrome pozwala także na wstrzymywanie wykonywania kodu. Aby przekonać się, jak działa ta możliwość, przejdź na kartę **Sources** i odszukaj plik `UserTodos.tsx`. Następnie dodaj punkt wstrzymania (ang. *breakpoint*) w wierszu pokazanym na rysunku 6.21.

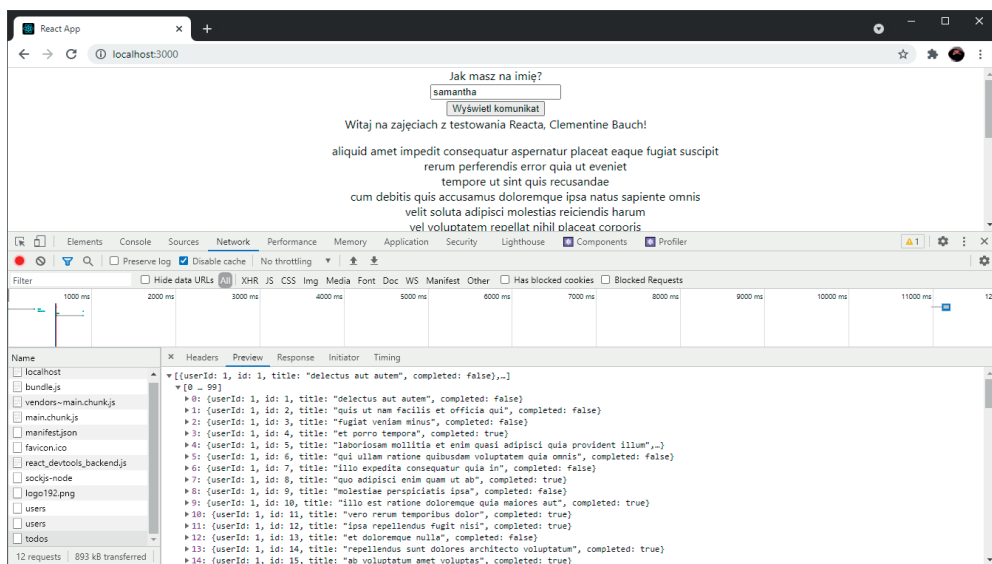


Rysunek 6.21. Karta Sources debuggera przeglądarki Chrome



Jak widać, debugger pozwala wstrzymać wykonywanie aplikacji w określonym wierszu kodu; taki punkt wstrzymania przeglądarka oznacza niebieską strzałką, taką jak ta widoczna w wierszu 32. Jeśli teraz wskażemy myszką jakąś zmienną, będziemy w stanie zobaczyć jej aktualną wartość i to nawet jeśli zmienna ta zawiera jakiś obiekt, na przykład komponent. Ta możliwość jest naprawdę bardzo przydatna podczas poszukiwania i poprawiania błędów w kodzie. Takie działanie debuggera jest możliwe dzięki wykorzystaniu tak zwanych map źródeł (ang. *source maps*). Mapy źródeł to pliki, które odwzorowują kod źródłowy aplikacji z jej zminimalizowanym kodem wykonywanym w przeglądarce. Są one tworzone i przesyłane do przeglądarki w trakcie trwania prac nad aplikacją i zapewniają możliwość wstrzymywania wykonywania kodu i sprawdzania wartości zmiennych.

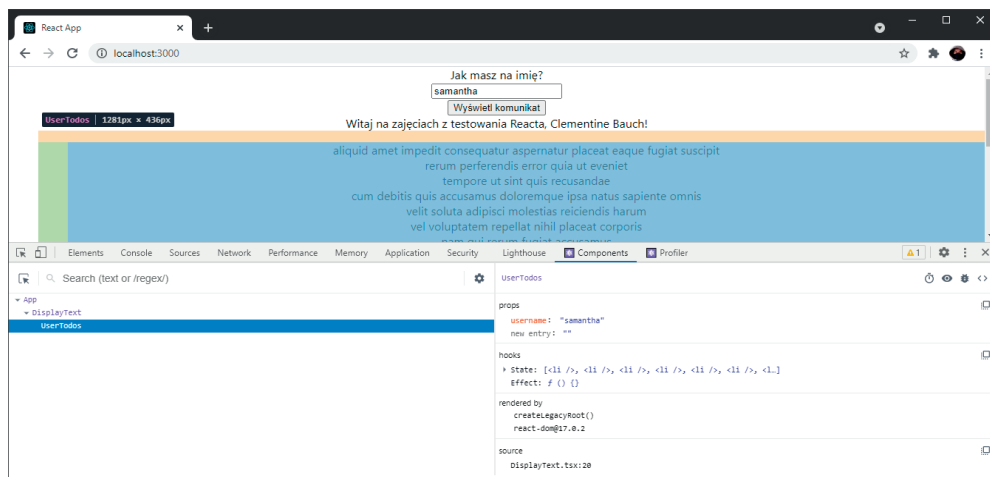
4. Teraz usuń punkt wstrzymania i przejdź na kartę *Network*. Przedstawia ona wszystkie połączenia sieciowe nawiązane przez przeglądarkę. Należą do nich nie tylko żądania dotyczące zasobów sieciowych, takich jak dane, lecz także żądania generowane przez przeglądarkę w celu pobierania obrazków oraz innych plików statycznych, w tym także plików HTML. Kiedy przejdziesz na tę kartę i pobierzesz zadania dla użytkownika samantha, będziesz mógł zobaczyć wyniki podobne do tych, które przedstawiłem na rysunku 6.22.



Rysunek 6.22. Karta Network debuggera przeglądarki Chrome

Jak widać, możemy przejrzeć wszystkie dane zapisane w odpowiedzi na żądanie przesłane do internetowego API. To bardzo użyteczne narzędzie, które pozwala przeglądać dane przekazywane z zasobów sieciowych i porównywać je z danymi oczekiwanymi przez wykonywany kod. Z tej możliwości będziemy korzystali w dalszej części książki, kiedy zajmiemy się językiem GraphQL.

W ten sposób przedstawiłem wbudowane narzędzia dla programistów, udostępniane przez przeglądarkę Chrome. Jednak Chrome pozwala także na korzystanie z rozszerzeń przeznaczonych do kontrolowania działania aplikacji Reacta. Narzędzia te, określane jako *React Developer Tools*, udostępniają informacje o hierarchii komponentów oraz ich atrybutach. Na rysunku 6.23 pokazałem, jak wyglądają informacje prezentowane przez to rozszerzenie w przypadku wyświetlenia przykładowej aplikacji używanej w tym rozdziale.



Rysunek 6.23. React Developer Tools

Jak widać, narzędzie to prezentuje hierarchię komponentów oraz atrybuty aktualnie wybranego komponentu. Oprócz tego, po wybraniu konkretnego komponentu w drzewie hierarchii, na ekranie wyróżniany jest fragment strony zawierający elementy tworzące ten komponent. To przydatne narzędzie od przeglądania elementów z punktu widzenia struktury komponentów aplikacji Reacta, a nie z punktu widzenia elementów HTML. Ekosystem rozszerzeń przeglądarki Chrome jest bardzo rozbudowany i można wśród nich znaleźć także rozszerzenia dla biblioteki Redux oraz Apollo GraphQL. Tymi dwoma zagadnieniami zajmiemy się odpowiednio w rozdziałach 8., pt. „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa” oraz 9., pt. „Czym jest GraphQL?”.

## Alternatywne zintegrowane środowiska programistyczne

W tej książce jako preferowanego edytora tekstów będziemy używać VSCode. Program ten spisuje się rewelacyjnie i jest najpopularniejszym edytorem do pisania kodu w językach JavaScript i TypeScript. Jednak nie oznacza to wcale, że musisz z niego korzystać. Co więcej, istnieją alternatywne programy, o których powinieneś wiedzieć; poniżej wymienię kilka z nich:

- **Atom.** To najprawdopodobniej drugi po VSCode najpopularniejszy darmowy edytor tekstów używany do pisania kodu.
- **Sublime Text.** Jeden z najszybszych i najpłynniej działających edytorów. Udostępnia także wersję bezpłatną.
- **Vim.** Edytor tekstów używany w systemie Unix, często stosowany do pisania kodu.
- **Webstorm.** Komercyjny edytor firmy JetBrains.

Warto wypróbować kilka z tych edytorów, gdyż korzystanie z dobrego edytora kodu może w znacznym stopniu poprawić efektywność pracy.

W tym podrozdziale przedstawiłem kilka narzędzi najczęściej używanych podczas tworzenia aplikacji Reacta. Choć te narzędzia nie uwolnią nas od konieczności pisania kodu aplikacji, to jednak są niezwykle potrzebne, gdyż pozwalają nie tylko przyspieszyć pracę, lecz także tworzyć kod o wyższej jakości. Co więcej, ułatwiają nam one także życie podczas pisania kodu, gdyż odszukiwanie błędów niejednokrotnie jest równie kłopotliwe, jak ich poprawianie.

## Podsumowanie

W tym rozdziale przedstawiłem wiele narzędzi używanych przez twórców aplikacji do tworzenia kodu o wysokiej jakości. Niezależnie od tego, czy jest to edytor Visual Studio Code używany do pisania kodu, czy też repozytorium Git służące do jego przechowywania i udostępniania, wszystkie narzędzia opisane w tym rozdziale są niezwykle istotne dla pracy twórców aplikacji internetowych.

Znajomość tych narzędzi pozwoli Ci stać się znacznie lepszym programistą, a jakość pisanego przez Ciebie kodu znacząco wzrośnie. Co więcej, wzrośnie także jakość Twojego życia jako programisty, gdyż wiele spośród opisanych tu narzędzi pozwala szybciej wykrywać błędy i ułatwia ich poprawianie.

W następnym rozdziale poszerzysz swoją wiedzę dotyczącą Reacta o zagadnienia związane ze stosowaniem dwóch istotnych frameworków: Redux i React Router. Pierwsze z nich, Redux, służy go globalnego przechowywania stanu, natomiast drugie, React Router, umożliwia stosowanie klienckich adresów URL. Oba te rozwiązania są niezwykle popularne wśród twórców aplikacji Reacta i zapewniają wiele narzędzi pomagających w tworzeniu aplikacji o bardziej wyrafinowanych i większych możliwościach.

# Redux i React Router

W tym rozdziale przedstawię frameworki Redux oraz React Router. Redux cały czas jest najpopularniejszym sposobem zarządzania globalnym stanem wykorzystywanym w obrębie całej aplikacji Reacta. Korzystając z takiego globalnego stanu zarządzanego przez Redux, możemy w znacznym stopniu wyeliminować stosowanie powtarzającego się kodu i zoptymalizować aplikację. Z kolei React Router jest najpopularniejszym frameworkiem do obsługi trasowania po stronie klienta. Klienckie adresy URL zapewniają aplikacjom jednostronicowym możliwość działania w sposób, jakiego oczekują użytkownicy, a także podobny do klasycznych aplikacji internetowych, czyli z adresem określającym miejsce aplikacji, w którym użytkownik aktualnie się znajduje. Oba te rozwiązania są konieczne do tworzenia aplikacji jednostronicowych, które swoim działaniem będą przypominać klasyczne aplikacje internetowe.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- zarządzaniem stanem przy użyciu Reduxa;
- przedstawieniem frameworka React Router.

---

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś dysponować podstawową wiedzą z zakresu tworzenia aplikacji Reacta. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora Visual Studio Code.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, anglojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial07* (*Chap7*).

Aby przygotować miejsce do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog, o nazwie *rozdzial07*.

## Zarządzanie stanem przy użyciu Reduxa

Redux jest najpopularniejszym, przystosowanym do zastosowań korporacyjnych frameworkiem służącym do tworzenia i zarządzania globalnym stanem w aplikacjach Reacta (choć można go także stosować w dowolnych aplikacjach pisanych w języku JavaScript, a nie jedynie w aplikacjach Reacta). Opracowano wiele nowszych rozwiązań podobnych do Reduxa, a niektóre z nich zyskały nawet spore grono użytkowników, niemniej Redux wciąż jest najczęściej używany. Być może początkowo uznasz, że Redux jest trudny w użyciu; jednak, kiedy go dokładniej poznasz, na pewno zauważysz jego liczne zalety i zrozumiesz, dlaczego tak często jest stosowany w dużych i złożonych aplikacjach Reacta.

Pojęcie stanu w aplikacjach Reacta przedstawiłem najpierw w rozdziale 4., pt. „Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React”, a następnie w rozdziale 5., pt. „Tworzenie aplikacji Reacta z wykorzystaniem *hooków*”. Przypomnę jednak jeszcze raz, że stan, czyli dane komponentów, jest głównym czynnikiem wywołującym zmiany interfejsu użytkownika aplikacji Reacta. To właśnie od tego wywodzi się nazwa frameworka, React<sup>1</sup>, gdyż reaguje on na zmiany stanu (mówi się także, że jest on reaktywny). Dlatego też, w przypadku tworzenia stanu i zarządzania nim, zazwyczaj będziemy starali się kojarzyć stan lokalny z komponentem lub z elementem głównym.

Kojarzenie stanu z komponentami może narzucać pewne ograniczenia. Zdarza się bowiem, że stan nie będzie charakterystyczny dla jakiegoś konkretnego komponentu, albo nawet dla hierarchii komponentów. Może się zdarzyć, że z tego samego stanu będzie musiało korzystać kilka komponentów, bądź też jakaś usługa, która wchodzi w skład aplikacji, lecz nie działa w oparciu o komponenty. Co więcej, w aplikacjach Reacta stan jest przekazywany w dół hierarchii komponentów tylko w jeden sposób: do komponentów podrzędnych, przy użyciu właściwości *props*. Nie należy natomiast przekazywać stanu w górę hierarchii komponentów. A to stanowi kolejne ograniczenie możliwości korzystania ze stanu w aplikacjach Reacta. To wszystko oznacza, że Redux nie tylko udostępnia mechanizm umożliwiający współużytkowanie stanu globalnego, lecz także wstrzykiwanie i aktualizowanie tego stanu z poziomu dowolnego komponentu, o ile tylko zajdzie taka konieczność.

Aby nadać tym rozważaniom nieco bardziej praktyczny charakter, przyjrzyjmy się stosowaniu Reduxa na przykładzie. W typowych aplikacjach korporacyjnych zawsze będzie stosowany jakiś mechanizm uwierzytelniania. A kiedy użytkownik zostanie już uwierzytelniony,

<sup>1</sup> W języku angielskim *react* oznacza reagować — *przyp. tłum.*

będzie można pobrać pewne informacje na jego temat — takie jak jego imię i nazwisko, identyfikator, adres poczty elektronicznej itd. Można przyjąć, że te informacje będą wykorzystywane przez znaczną większość komponentów w aplikacji; co więcej, takie założenie będzie całkiem uzasadnione. Rozwiązanie wymagające, by każdy komponent pobierał te dane, a następnie zapisywał je we własnym stanie, byłoby niezwykle uciążliwe i podatne na błędy. Oznaczałoby ono, że będziemy przechowywać wiele kopii danych, a kiedy zostaną one zmienione, niektóre komponenty mogą tego nie zauważyć i przechowywać nieaktualne dane.

Konflikty tego rodzaju mogą być potencjalnym źródłem błędów. Dlatego warto by mieć możliwość przechowywania danych po stronie klienta w jednym miejscu i udostępniania ich wszystkim komponentom, które będą ich potrzebować. Dzięki temu, jeśli zdarzy się, że dane kiedyś zostaną zaktualizowane, będziemy mieli pewność, że wszystkie komponenty, niezależnie od tego, w jakim miejscu aplikacji są używane, będą korzystały z aktualnych danych. I właśnie takie możliwości zapewnia nam Redux. Można go uznać za **jedyne źródło danych** w aplikacji.

Redux jest usługą przechowywania danych, która zarządza wszystkimi danymi używanymi globalnie w aplikacji Reacta. Redux udostępnia nie tylko sam magazyn, lecz także podstawowe możliwości funkcjonalne konieczne do dodawania, usuwania oraz udostępniania danych. Jedną ważną różnicą w stosunku do standardowego stanu stosowanego w komponentach Reacta jest to, że stan Reduxa niekoniecznie będzie wywoływał aktualizację interfejsu użytkownika. Oczywiście może to robić, jeśli sobie tego zażyczymy, jednak nie jest to wcale konieczne. Warto, żebyś o tym pamiętał.

Przekonajmy się zatem, w jaki sposób przygotować projekt aplikacji Reacta korzystającej z Reduxa.

1. W katalogu *rozdzial07* utwórz nowy projekt Reacta; wykonaj w tym celu następujące polecenie:
2. Kiedy projekt zostanie utworzony i skonfigurowany, otwórz go w edytorze i w oknie terminala przejdź do katalogu *redux-sample*.
3. Teraz zajmiemy się zainstalowaniem Reduxa, który sam ma kilka dodatkowych zależności. W pierwszej kolejności wykonaj następujące polecenie:

```
npm i redux react-redux @types/redux @types/react-redux
```

To polecenie instaluje podstawowe zależności Reduxa, w tym typy używane w kodzie TypeScript.

No dobrze, skoro nasz przykładowy projekt jest już skonfigurowany, to zanim przejdziemy do poznawania Reduxa, musisz jeszcze zrozumieć kilka zagadnień z nim związanych. Redux korzysta z dwóch pojęć: reduktorów oraz akcji. Zacznę od wyjaśnienia, co one oznaczają.

## Reduktory i akcje

Redux używa tylko jednego magazynu do przechowywania wszystkich danych. Oznacza to, że wszystkie nasze dane globalne będą przechowywane w jednym obiekcie Reduxa. Problem z takim rozwiązaniem polega na tym, że ponieważ stan jest globalny, różne fragmenty aplikacji będą potrzebowały różnych typów danych i nie wszystkie z tych danych będą miały znaczenie dla wszystkich fragmentów aplikacji. Dlatego twórcy Reduxa zdecydowali się na zastosowanie rozwiązania bazującego na tak zwanych reduktorach (ang. *reducers*), które filtrują zawartość magazynu i dzielą ją na części. A zatem, jeśli komponent A potrzebuje konkretnych danych, nie będzie musiał przetwarzać całej zawartości magazynu, aby uzyskać te informacje, które go interesują.

To rozwiązanie jest przykładem doskonałego projektu separacji obowiązków związanych z wykorzystaniem danych. Jednak efektem ubocznym przyjęcia takiego rozwiązania jest to, że potrzebujemy jakiegoś sposobu aktualizowania wybranego fragmentu danych bez wprowadzania zmian w pozostałej zawartości magazynu. I właśnie do tego służą akcje (ang. *actions*). Akcje to obiekty, które dostarczają danych dla wybranych reduktorów.

Skoro już dowiedziałeś się ogólnie, czym są reduktory i akcje, przyjrzyjmy się im na przykładzie:

1. W katalogu *src* utwórz nowy podkatalog o nazwie *store*.
2. Wewnątrz tego nowego katalogu utwórz plik *AppState.ts*. W tym pliku umieścimy zagregowany obiekt reduktora o nazwie *rootReducer*. Będzie to obiekt typu *AppState*, reprezentujący globalny stan naszej aplikacji. Poniżej przedstawiłem kod, który należy zapisać w pliku *AppState.ts*:

```
import { combineReducers } from "redux";

export const rootReducer = combineReducers({
});

export type AppState = ReturnType<typeof rootReducer>;
```

Obiekt *rootReducer* reprezentuje zagregowany obiekt wszystkich naszych reduktorów. Póki co nie będzie ich wiele, jednak, kiedy już przygotujemy wszystkie niezbędne podstawowe elementy naszej aplikacji, zaczniemy dodawać faktyczne reduktory. Funkcja *combineReducers* pobiera wszystkie przekazane reduktory i łączy je w jeden obiekt. W ostatnim wierszu pliku tworzymy nowy typ, używając do tego celu typu pomocniczego *ReturnType*, a następnie eksportujemy go pod nazwą *AppState*.

Typ pomocniczy to zwyczajna klasa pomocnicza, której twórcy języka TypeScript nadali szczególne możliwości funkcjonalne. Dostępnych jest wiele takich typów, a ich pełną listę można znaleźć na stronie <https://www.typescriptlang.org/docs/handbook/utility-types.html>.

3. Kolejnym krokiem będzie utworzenie pliku o nazwie *configureStore.ts* i zapisanie w nim faktycznego obiektu magazynu, który będzie używany zarówno przez Reduxa, jak i przez naszą aplikację. Poniżej przedstawiłem zawartość tego pliku:

```
import { createStore } from "redux";
import { rootReducer } from "../AppState";

const configureStore = () => {
  return createStore(rootReducer, {});
};
export default configureStore;
```

Jak widać, do utworzenia faktycznego magazynu używamy metody Reduxa o nazwie `createStore`, do której przekazujemy obiekt `rootReducer` typu `AppStore`. Funkcja `configureStore` jest eksportowana i później wywołamy ją, by utworzyć magazyn.

4. Teraz musimy zaktualizować plik `index.tsx` i wywołać w nim funkcję `configureStore`, aby zainicjować magazyn, którego będziemy używali w aplikacji. Poniżej przedstawiłem zawartość pliku `index.tsx`:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { Provider } from "react-redux";
import configureStore from "../store/configureStore";
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <Provider store={configureStore()}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

W pierwszej kolejności importujemy komponent `Provider` z modułu `react-redux`. `Provider` jest komponentem Reacta, który pełni rolę komponentu nadrzędnego dla wszystkich pozostałych komponentów aplikacji i jednocześnie *dostarcza* im dane magazynu. Co więcej, jak widać w powyższym kodzie, do komponentu `Provider` przekazujemy zainicjowany magazyn, który stworzymy, wywołując funkcję `configureStore`:

```
// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Te komentarze umieszczone na samym końcu kodu są elementem projektu generowanego przez `create-react-app`. Zamieściłem je tu dla zachowania kompletności kodu. No dobrze, dysponujemy już zatem podstawową konfiguracją aplikacji Reacta z Reduxem. Możemy więc przystąpić do tworzenia wywołania, które pobierze obiekt użytkownika. Wykorzystamy do tego celu API usługi `JSONPlaceholder`, którego używaliśmy już w poprzednim rozdziale, pt. „Przygotowywanie projektu za pomocą `create-react-app` i testowanie go przy użyciu Jest”. Po poprawnym zalogowaniu użytkownika chcemy udostępnić informacje o nim; w tym celu musimy przygotować odpowiedni reduktor Reduxa. Poniżej opisałem czynności, które należy wykonać:



1. W katalogu *store* utwórz nowy plik o nazwie *UserReducer.ts*, a na samym jego początku zapisz następujący wiersz kodu:

```
export const USER_TYPE = "USER_TYPE";
```

Jak widać, najpierw definiujemy stałą `USER_TYPE`, która będzie określać typ akcji. Choć nie jest to wymagane, to jednak takie rozwiązanie pomaga unikać literówek w innych miejscach kodu.

```
export interface User {
  id: string;
  username: string;
  email: string;
  city: string;
}
```

Następnie tworzymy typ reprezentujący użytkownika.

```
export interface UserAction {
  type: string;
  payload: User | null;
}
```

Kolejnym elementem kodu jest akcja — obiekt, który zwyczajowo zawiera dwie składowe, określające odpowiednio typ akcji oraz jej zawartość. Tworzymy zatem typ takiej akcji — `UserAction` — i deklarujemy w nim dwie składowe: `type` i `payload`.

```
export const UserReducer = ( state: User | null = null,
  action: UserAction): User | null => {
  switch(action.type) {
    case USER_TYPE:
      console.log("reduktor UserReducer", action.payload);
      return action.payload;
    default:
      return state;
  }
};
```

I w końcu tworzymy nasz reduktor, funkcję `UserReducer`. Reduktory zawsze mają dwa parametry: `state` oraz `action`. Zwróć uwagę na to, że parametr `state` nie reprezentuje całego stanu, a jedynie jego fragment charakterystyczny dla danego reduktora. Reduktor określi, czy przekazany stan (`state`) jest stanem charakterystycznym dla niego, czy nie, na podstawie typu przekazanego jako parametr `action`. Oprócz tego zwróć uwagę na to, że początkowy stan nigdy nie jest modyfikowany. To ogromnie ważne. *Nigdy* nie zmieniaj stanu bezpośrednio. Powinieneś zwracać stan w takiej samej postaci, w jakiej został przekazany do reduktora (za co odpowiada klauzula `default` instrukcji `switch`), bądź też zwracać jakieś inne dane; w naszym przypadku jest to wartość `action.payload`.

2. Teraz musimy wrócić do pliku *AppState.ts* i dodać do niego ten nowy reduktor. Poniżej pokazałem, jak powinna wyglądać zmodyfikowana postać tego pliku:

```
import { combineReducers } from "redux";
import { UserReducer } from "../UserReducer";
```

```
export const rootReducer = combineReducers({
  user: UserReducer
});

export type AppState = ReturnType<typeof rootReducer>;
```

A zatem, nasz magazyn Reduxa zyskał jedną nową właściwość, `user`, której wartość jest aktualizowana przez reduktor `UserReducer`. Gdybyśmy musieli używać w aplikacji jeszcze innych reduktorów, to wystarczyłoby nadać im unikalne nazwy i zapisać poniżej reduktora `user` — funkcja `combineReducers` połączyłaby je wszystkie w jeden zagregowany reduktor `rootReducer`.

3. A teraz zastosujmy nasz nowy stan w kodzie aplikacji. Poniżej pokazałem nową postać kodu pliku *App.tsx*:

```
import React, { useState } from 'react';
import './App.css';

function App() {
  const [userid, setUserId] = useState(0);
  const onChangeUserId = (e: React.ChangeEvent<HTMLInputElement>) => {
    console.log("userid", e.target.value);
    setUserId(e.target.value ? Number(e.target.value) : 0);
  }
}
```

Pobieramy identyfikator użytkownika (`userid`) jako parametr, a następnie, na jego podstawie, będziemy pobierać dane użytkownika z API serwisu `JSONPlaceholder`. Jednak, aby to zrobić, musimy skorzystać ze specjalnego *hooka* Reduxa, który pozwoli nam dodać pobranego użytkownika do magazynu.

4. Wprowadź dalsze zmiany do kodu komponentu `App` w pliku *App.tsx*:

```
function App() {
  const [userid, setUserId] = useState(0);
  const dispatch = useDispatch();
```

Jak widać, do kodu dodaliśmy stałą `dispatch`, której wartość pobieramy przy użyciu *hooka* Reduxa o nazwie `useDispatch`. Stała `dispatch` zawiera funkcję, która przesyła do magazynu Reduxa dane akcji. Redux prześle następnie tę akcję do wszystkich reduktorów, które ją przetworzą. Każdy z reduktorów, który rozpozna typ akcji, zaakceptuje ją i użyje jej zawartości:

```
const onChangeUserId = async (e: React.ChangeEvent<HTMLInputElement>) => {
  const useridFromInput = e.target.value ? Number(e.target.value) : 0;
  console.log("userid", useridFromInput);
  setUserId(useridFromInput);

  const usersResponse = await fetch('https://jsonplaceholder.typicode.com/users');
  if(usersResponse.ok) {
    const users = await usersResponse.json();
    console.log("users", users);
    const usr = users.find((userItem: any) => {
      return userItem.id === useridFromInput;
    });
  }
};
```

```

    dispatch({
      type: USER_TYPE,
      payload: {
        id: usr.id,
        username: usr.username,
        email: usr.email,
        city: usr.address.city
      }
    });
  }
}

```

W procedurze obsługi zdarzeń `onChangeUserId` generujemy żądanie do API serwisu JSONPlaceholder. Następnie korzystamy z obiektu `userResponse`, by pobrać zwrócone wyniki. Po pobraniu z odpowiedzi wszystkich użytkowników znajdujemy tego, który nas interesuje, używając w tym celu identyfikatora wpisanego w interfejsie użytkownika aplikacji. I w końcu używamy funkcji `dispatch`, aby przesłać obiekt akcji do przygotowanego wcześniej reduktora. Zwróć także uwagę na to, że `onChangeUserId` jest teraz funkcją asynchroniczną — przed jej definicją umieściliśmy bowiem słowo kluczowe `async`.

```

return (
  <div className="App">
    <label>Identyfikator użytkownika</label>
    <input value={userid} onChange={onChangeUserId} />
  </div>
);
}

```

Ostatni fragment pliku *App.tsx* generuje interfejs użytkownika z polem tekstowym służącym do podawania identyfikatora interesującej nas osoby.

Teraz zajmiemy się przygotowaniem komponentu podrzędnego, który będzie wyświetlał wszystkie informacje związane z użytkownikiem.

1. Utwórz nowy plik komponentu o nazwie *UserDisplay.tsx* i zapisz w nim następujący kod:

```

import React, { useRef } from 'react';
import { AppState } from './store/AppState';
import { useSelector } from 'react-redux';

const UserDisplay = () => {
  const user = useSelector((state: AppState) => state.user);

  if(user) {
    return (<React.Fragment>
      <div>
        <label>Nazwa użytkownika:</label>
        &nbsp;{user.username}
      </div>
      <div>
        <label>E-mail:</label>
        &nbsp;{user.email}
      </div>
    </React.Fragment>
  )
  }
}

```



```
    });
  }
}
```

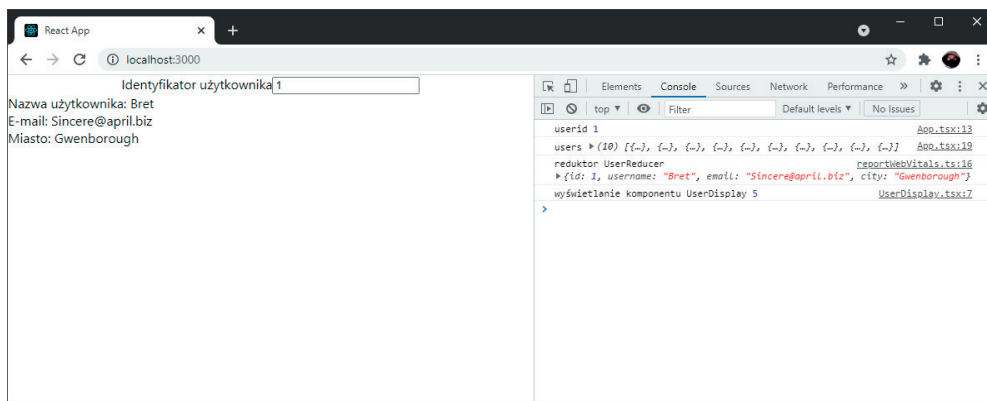
W tym fragmencie kodu nic się nie zmieniło.

```
    return (
      <React.Fragment>
        <div className="App">
          <label>Identyfikator użytkownika</label>
          <input value={userid} onChange={onChangeUserId} />
        </div>
        <UserDisplay />
      </React.Fragment>
    );
  }

  export default App;
```

W tym fragmencie kodu umieszczamy komponent `UserDisplay` w kodzie JSX zwracanym przez komponent `App`, dzięki czemu dane użytkownika zostaną wyświetlone w przeglądarce.

3. Jeśli teraz wyświetlisz w przeglądarce stronę o adresie `http://localhost:3000`, a następnie w polu tekstowym na stronie wpiszesz 1, to zostaną wyświetlone dane użytkownika, które przedstawiłem na rysunku 7.1.



**Rysunek 7.1.** Obiekt użytkownika pobrany z magazynu Redux

A zatem, skoro już przeanalizowaliśmy prosty przykład zastosowania magazynu Reduxa, możemy pójść o krok dalej i przekonać się, co się stanie, kiedy w tym samym magazynie zastosujemy większą liczbę reduktorów.

1. Utwórz nowy plik o nazwie `PostDisplay.tsx` i zapisz w nim kod przedstawiony na poniższym przykładzie. Ten komponent będzie prezentował komentarze pobrane z API serwisu JSONPlaceholder:

```
import React, { useRef } from 'react';
import { AppState } from './store/AppState';
import { useSelector } from 'react-redux';
```

```
const PostDisplay = React.memo(() => {
  const renderCount = useRef(0);
  console.log("wyświetlanie komponentu PostDisplay",
    renderCount.current++);
  const post = useSelector((state: AppState) => state.post);
```

Podobnie jak w przedstawionym wcześniej komponentcie UserDisplay, także tu używamy funkcji useSelector, by określić, które dane stanu nas interesują.

```
    if(post) {
      return (<React.Fragment>
        <div>
          <label>Tytuł:</label>
          &nbsp;{post.title}
        </div>
        <div>
          <label>Treść:</label>
          &nbsp;{post.body}
        </div>
      </React.Fragment>);
    } else {
      return null;
    }
  });
```

```
export default PostDisplay
```

Jak widać, ten komponent jest bardzo podobny do poprzedniego, przy czym wyświetla informacje powiązane z wpisami, takie jak tytuł (title) i treść (body).

2. Teraz musimy zaktualizować magazyn Reduxa i dodać do niego nowy reduktor. Zacznij od dodania w katalogu *store* nowego pliku o nazwie *PostReducers.ts*, a następnie zapisz w nim poniższy kod:

```
export const POST_TYPE = "POST_TYPE";

export interface Post {
  id: number;
  title: string;
  body: string;
}

export interface PostAction {
  type: string;
  payload: Post | null;
}

export const PostReducer = (state: Post | null = null, action: PostAction):
  Post | null => {
  switch(action.type) {
    case POST_TYPE:
      return action.payload;
    default:
      return state;
  }
};
```

Jak widać, ten reduktor jest bardzo podobny do reduktora `UserReducer`, przy czym koncentruje się na danych dotyczących wpisów, a nie użytkownika.

3. Teraz musimy zaktualizować plik *AppState.tsx*, a konkretnie: dodać do niego nowy reduktor. Niezbędne zmiany pokazałem na poniższym przykładzie:

```
import { combineReducers } from "redux";
import { UserReducer } from "../UserReducer";
import { PostReducer } from "../PostReducer";

export const rootReducer = combineReducers({
  user: UserReducer,
  post: PostReducer
});

export type AppState = ReturnType<typeof rootReducer>;
```

Jak widać, zmiany, jakie tu wprowadziliśmy, polegały wyłącznie na dodaniu reduktora `PostReducer`.

4. Naszym kolejnym krokiem będzie zmodyfikowanie komponentu `App` i dodanie do niego kodu pozwalającego na wyszukanie w danych zwracanych przez API serwisu `JSONPlaceholder` wpisu o podanym identyfikatorze. Zmodyfikuj kod komponentu `App` tak, jak pokazałem na poniższym przykładzie:

```
function App() {
  const [userid, setUserid] = useState(0);
  const dispatch = useDispatch();
  const [postid, setPostId] = useState(0);
```

Warto zwrócić uwagę na to, że używana funkcja `dispatch` nie jest powiązana z żadnym konkretnym reduktorem. Wynika to z faktu, że funkcje zwracane przez wywołanie `useDispatch` mają charakter ogólny — obsługiwane akcje, wcześniej czy później, trafią do odpowiedniego reduktora.

Kod procedury obsługi zdarzeń `onChangeUserId` nie zmienił się, jednak i tak go tutaj zamieściłem:

```
const onChangeUserId = async (e: React.ChangeEvent<HTMLInputElement>) => {
  const useridFromInput = e.target.value ? Number(e.target.value) : 0;
  console.log("userid", useridFromInput);
  setUserid(useridFromInput);

  const usersResponse = await fetch('https://jsonplaceholder.typicode.com/users');
  if(usersResponse.ok) {
    const users = await usersResponse.json();
    const usr = users.find((userItem: any) => {
      return userItem.id === useridFromInput;
    });

    dispatch({
      type: USER_TYPE,
      payload: {
        id: usr.id,
        username: usr.username,
        email: usr.email,
```

```

        city: usr.address.city
      }
    });
  }
}

```

Z kolei `onChangePostId` jest nową procedurą obsługi zdarzeń, obsługującą zmiany danych związanych z wpisami:

```

const onChangePostId = async (e: React.ChangeEvent<HTMLInputElement>) => {
  const postIdFromInput = e.target.value ? Number(e.target.value) : 0;
  setPostId(postIdFromInput);

  const postResponse = await
    fetch("https://jsonplaceholder.typicode.com/posts/" + postIdFromInput);
  if(postResponse.ok) {
    const post = await postResponse.json();
    console.log("post", post);
    dispatch({
      type: POST_TYPE,
      payload: {
        id: post.id,
        title: post.title,
        body: post.body
      }
    })
  }
}

```

Procedura obsługi zdarzeń `onChangePostId` przekazuje odpowiednią akcję do magazynu Reduxa, wywołując w tym celu funkcję `dispatch`.

Musimy także nieco zmienić interfejs użytkownika komponentu `App`, a konkretnie: dodać do niego komponent `PostDisplay` i oddzielić go wizualnie od komponentu `UserDisplay`:

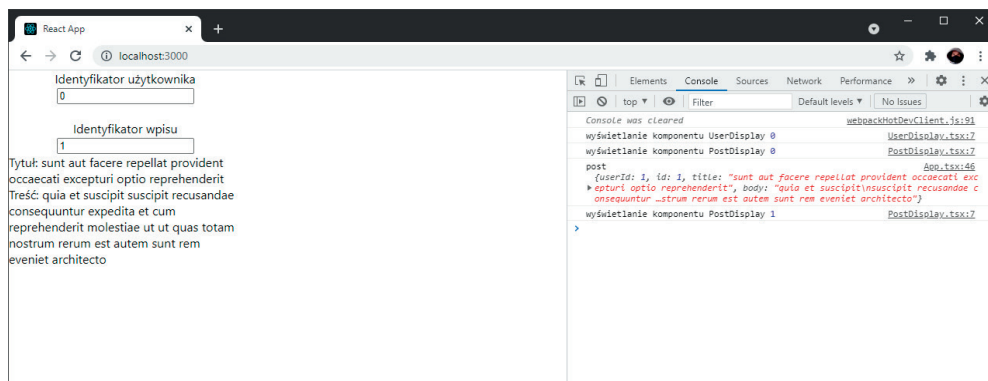
```

return (
  <React.Fragment>
    <div style={{width: "300px"}}>
      <div className="App">
        <label>Identyfikator użytkownika</label>
        <input value={userid} onChange={onChangeUserId} />
      </div>
      <UserDisplay />
    </div>
    <br/>
    <div style={{width: "300px"}}>
      <div className="App">
        <label>Identyfikator wpisu</label>
        <input value={postId} onChange={onChangePostId} />
      </div>
      <PostDisplay />
    </div>
  </React.Fragment>
);
}

```



Jeśli teraz uruchomisz aplikację i zmienisz wartość identyfikatora wpisu (`postId`), to strona wyświetlona w przeglądarce powinna przypominać przedstawioną na rysunku 7.2.



**Rysunek 7.2.** Wyniki dodania do aplikacji komponentu `PostDisplay`

Przyjrzyj się panelowi *Console* widocznemu z prawej strony okna przeglądarki na rysunku 7.2 i zwróć uwagę na to, że po zmianie wartości pola `postId` nie został wyświetlony komunikat o aktualizacji komponentu `UserDisplay`.

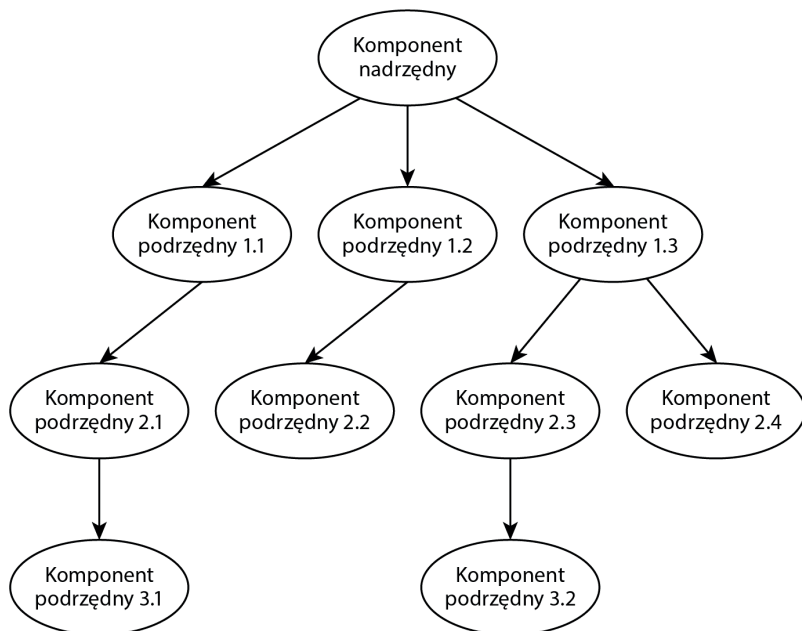
Wyraźnie to pokazuje, że magazyn Redux nie jest bezpośrednio połączony potokiem renderowania komponentów Reacta oraz że powtórnie renderowane są jedynie komponenty bezpośrednio powiązane z aktualną zmianą stanu. Jak widać, jest to inny mechanizm działania niż ten stosowany przez kontekst Reacta (`React Context`), a dzięki ograniczaniu liczby niepożądanych operacji renderowania może zapewnić lepszą wydajność działania aplikacji. (Zagadnienia związane z kontekstem Reacta opisałem w następnym punkcie rozdziału).

W tym podrozdziale przedstawiłem zagadnienia związane ze stosowaniem Reduxa — najpopularniejszego sposobu zarządzania globalnym stanem w aplikacjach Reacta. Takie narzędzia do zarządzania globalnym stanem są bardzo często używane w większych aplikacjach, w którym zazwyczaj występują globalne, współużytkowane dane. W przedstawionej aplikacji przykładowej, w magazynie przechowujemy informacje o zalogowanym użytkowniku, jak również inne dane, które będą używane w różnych miejscach aplikacji; dlatego wykorzystanie globalnego stanu będzie użyteczne.

## React Context

Kontekst (`React Context`) to nowa możliwość, która została wprowadzona nieco przed *hookami*. `React Context` nie jest odrębną zależnością, lecz został wbudowany bezpośrednio w podstawowy framework Reacta. Zapewnia on podobne możliwości co `Redux`, czyli pozwala na przechowywanie stanu w jednym miejscu i udostępnianie go różnym komponentom bez konieczności ręcznego przekazywania go w dół hierarchii komponentów.

Ta możliwość jest bardzo wydajna z punktu widzenia programisty piszącego aplikację, gdyż eliminuje konieczność tworzenia rozbudowanego, powtarzającego się kodu, niezbędnego do przekazywania stanu z komponentów nadrzędnych do podrzędnych. Na rysunku 7.3 przedstawiłem wizualizację przykładowej hierarchii dużej aplikacji Reacta.



**Rysunek 7.3.** Hierarchia komponentów Reacta

W tym przykładzie występuje jeden komponent nadrzędny i kilka komponentów podrzędnych, które są umieszczone w kodzie JSX komponentu nadrzędnego. Te komponenty podrzędne mają z kolei własne komponenty podrzędne, itd. Gdybyśmy musieli skonfigurować przekazywanie właściwości *props* do każdego komponentu w takiej hierarchii, wymagałoby to napisania całkiem rozbudowanego kodu, zwłaszcza jeśli weźmiemy pod uwagę, że niektóre hierarchie wymagają przekazywania funkcji, które odwołują się do określonych komponentów nadrzędnych. Co więcej, stosowanie tego typu zależności związanych z właściwościami *props* może stanowić dodatkowe obciążenie i wyzwanie poznawcze dla programistów, którzy są zmuszani do myślenia o występujących zależnościach pomiędzy danymi oraz o sposobach ich przekazywania pomiędzy poszczególnymi komponentami.

Zarówno Redux, jak i React Context, stosowane w odpowiednich okolicznościach, są doskonałymi rozwiązaniami pozwalającymi unikać takiego powtarzalnego kodu związanego z przekazywaniem stanu. A w mniejszych projektach prostota, jaką zapewnia React Context, będzie się doskonale sprawdzać. Jednak w przypadku większych projektów polecałbym raczej stosowanie Reduxa.

React Context pozwala na stosowanie wielu nadrzędnych dostawców, co oznacza, że może istnieć więcej niż jeden kontekst nadrzędny. W przypadku większych aplikacji takie rozwiązanie może być mylące i zmuszać do pisania bardziej rozbudowanego, powtarzalnego

kodu. Co więcej, także jednoczesne stosowanie wielu różnych globalnych dostaw może być mylące. Jeśli zespół zdecyduje się jednocześnie korzystać i z Reduxa, i z React Context, to kiedy trzeba będzie używać każdego z nich? A jeśli będziemy używać obu równocześnie, to trzeba będzie jednocześnie zarządzać dwoma globalnymi stanami.

Co więcej, w odróżnieniu do Reduxa, w React Context nie są używane reduktory. Oznacza to, że wszyscy użytkownicy kontekstu będą otrzymywać pełny zestaw danych stanu, co nie jest optymalnym rozwiązaniem z punktu widzenia separacji odpowiedzialności. Wraz z upływem czasu staje się coraz mniej jasne na jakim podzbiorze danych powinien operować dany komponent.

Jednym z dodatkowych efektów ubocznych udostępniania wszystkich danych stanu wszystkim komponentom jest to, że nawet jeśli komponent nie odwołuje się do pewnych składowych stanu, to i tak wszelkie zmiany stanu będą powodowały ponowne wyrenderowanie tego komponentu. Załóżmy na przykład, że stan przechowywany przez Context ma następującą postać: { username, userage }, a pewien komponent używa tylko składowej username. W takim przypadku, nawet jeśli zmiana ulegnie składowa userage, to komponent i tak zostanie ponownie wyrenderowany. Tak się dzieje nawet w przypadku korzystania z funkcji memo (opisałem ją w rozdziale 5., pt. „Tworzenie aplikacji Reacta z wykorzystaniem *hooków*”). Przeanalizujmy przykład przedstawiający ten efekt.

1. Aby uniknąć zamieszania, usuń z kodu pliku *index.tsx* komponenty *ReactStrict* ➤ *Mode* oraz *Provider*. Później dodamy je z powrotem. Plik *index.tsx* powinien wyglądać jak na poniższym przykładzie:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { Provider } from "react-redux";
import configureStore from "./store/configureStore";
import reportWebVitals from './reportWebVitals';

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

Komentarze umieszczone w ostatniej części kodu są elementem projektu generowanego przez *create-react-app*; zamieszczam je tu, gdyż chcę przedstawić kompletny kod pliku:

```
// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Jeśli zależy Ci na wyeliminowaniu komentarzy z ostrzeżeniami, możesz usunąć wszystkie niepotrzebne instrukcje importu.

2. Teraz zajmiemy się utworzeniem dwóch komponentów podrzędnych, z których każdy będzie używał unikalnej liczby pobieranej ze stanu obsługiwanego przez React Context. W pierwszej kolejności utwórz plik *UserAgeComp.tsx* i zapisz w nim następujący kod:

```
import React, { useContext } from 'react';
import { TestContext } from './ContextTester';

const UserAgeComp = () => {
  const { userage } = useContext(TestContext);

  return <div>
    {userage}
  </div>
};

export default UserAgeComp;
```

Ten kod korzysta z mechanizmu destrukuryzacji obiektów, by pobrać i wyświetlić wartość składowej *userage* kontekstu *TestContext* zwracanego przez wywołanie *hooka* *useContext*. Utworzeniem tego *hooka* zajmiemy się już niebawem. Teraz utwórz plik drugiego komponentu, *UserNameComp.tsx*, i zapisz w nim następujący kod:

```
import React, { useContext, useRef } from 'react';
import { TestContext } from './ContextTester';

const UserNameComp = React.memo(() => {
  const renders = useRef(0);
  console.log("renderowanie komponentu UserNameComp", renders.current++);

  const username = 'samantha' // useContext(TestContext);
  console.log("wartość username w komponencie UserNameComp", username);

  return <div>
    {username}
  </div>
});

export default UserNameComp;
```

Być może będziesz zaskoczony faktem, że w tym komponencie nie pobieramy wartości właściwości *username* z kontekstu (jak widać, odwołanie do kontekstu jest umieszczone w komentarzu), jednak zanim pokażę konsekwencje stosowania kontekstu, chciałem pokazać ten komponent działający zgodnie z oczekiwaniami. A zatem, ten komponent ma dwie główne możliwości. Pierwszą z nich jest referencja przechowująca liczbę określającą, ile razy komponent został wyrenderowany, a drugą zmienna *username*, której wartość komponent wyświetla. Oprócz tego komponent wyświetla na konsoli, ile razy został wyrenderowany, dzięki czemu łatwiej nam będzie określić, kiedy będzie on ponownie wyświetlany.

3. Kolejnym zadaniem będzie utworzenie komponentu nadrzędnego, który będzie zawierał kontekst. Aby go zaimplementować utwórz plik *ContextTester.tsx* i zapisz w nim następujący kod:

```
import React, { createContext, useState } from 'react';
import UserNameComp from './UserNameComp';
import UserAgeComp from './UserAgeComp';
```

W kolejnym fragmencie kodu używamy funkcji `createContext`, żeby utworzyć obiekt `TestContext` zawierający stan:

```
export const TestContext = createContext({ username: string, userage:
↳number }>({ username: "", userage:0 }));

const ContextTester = () => {
  const [userage, setUserage] = useState(20);
  const [localState, setLocalState] = useState(0);

  const onClickAge = () => {
    setUserage(
      userage + 1
    );
  }

  const onClickLocalState = () => {
    setLocalState(localState + 1);
  }

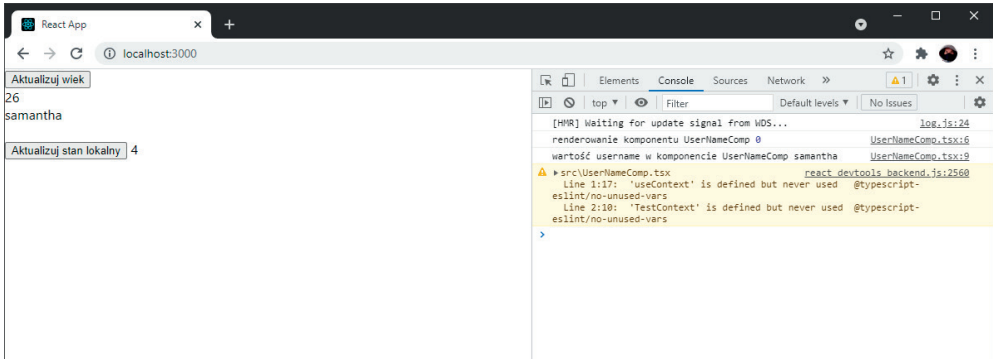
  return (<React.Fragment>
    <button onClick={onClickAge}>Aktualizuj wiek</button>
    <TestContext.Provider value={{ username: "samantha", userage }}>
      <UserAgeComp />
    </TestContext.Provider>
    <UserNameComp />
    <br/>
    <button onClick={onClickLocalState}>Aktualizuj stan lokalny</button>
    &nbsp;<label>{localState}</label>
  </React.Fragment>);
}

export default ContextTester;
```

Ten komponent prezentuje także dwa ważne zagadnienia. Pierwszym z nich jest inkrementacja wartości zmiennej `localState`, która jest wykonywana w procedurze obsługi zdarzeń `onClickLocalState`, a drugim wyświetlanie dwóch komponentów podrzędnych, `UserNameComp` i `UserAgeComp`. Zwróć uwagę na to, że jak dotąd komponent `UserNameComp` jest wyświetlany poza komponentem kontekstu `TextContext`, a to oznacza, że zmiany kontekstu nie będą miały na niego wpływu. *Konieczniesz musisz zwrócić na to uwagę!*

4. Jeśli teraz klikniesz przycisk *Aktualizuj wiek* lub *Aktualizuj stan lokalny*, to przekonasz się, że wywołanie `console.log` z komponentu `UserNameComp` nie jest wykonywane. Zostało ono wykonane tylko raz, podczas pierwszego wyświetlania strony, czyli dokładnie zgodnie z oczekiwaniami, gdyż komponent

ten używa funkcji memo (a jak zapewne pamiętasz, funkcja ta uniemożliwia wykonywanie operacji renderowania, jeśli nie uległy zmianie właściwości *props*). Dlatego też w panelu *Console* przeglądarki powinieneś zobaczyć tylko jeden zestaw komunikatów, tak jak pokazałem na rysunku 7.4.



Rysunek 7.4. Efekty wyświetlania kontekstu

5. No dobrze, a teraz spróbujmy zmusić komponent `UserNameComp` do używania właściwości `username` z kontekstu `TestContext`. Zmodyfikuj kod komponentu `UserNameComp` zgodnie z poniższym przykładem:

```
import React, { useContext, useRef } from 'react';
import { TestContext } from './ContextTester';

const UserNameComp = React.memo(() => {
  const renders = useRef(0);
  console.log("renderowanie komponentu UserNameComp", renders.current++);

  const { username } = useContext(TestContext);
  console.log("wartość username w komponencie UserNameComp", username);

  return <div>
    {username}
  </div>
});

export default UserNameComp;
```

Jak widać, po wprowadzeniu zmian komponent `UserNameComp` używa zmiennej `username` z kontekstu `TestContext`. Komponent ten nigdy nie używa zmiennej `userage`, a jak pamiętamy, wartość zmiennej `username` jest podana na stałe i nigdy się nie zmienia. A zatem, teoretycznie, stan komponentu `UserNameComp` nigdy się nie zmienia, więc komponent nie powinien być ponownie renderowany. Teraz musimy jednak umieścić ten komponent wewnątrz znacznika `TestContext`. Wynika to z faktu, że jeśli komponent musi używać stanu React Context, to musi być umieszczony wewnątrz znacznika kontekstu. Zmodyfikuj więc zawartość komponentu `ContextTester` zgodnie z poniższym przykładem:

```

const ContextTester = () => {
  const [userage, setUserage] = useState(20);
  const [localState, setLocalState] = useState(0);

  const onClickAge = () => {
    setUserage(
      userage + 1
    );
  }

  const onClickLocalState = () => {
    setLocalState(localState + 1);
  }

  return (<React.Fragment>
    <button onClick={onClickAge}>Aktualizuj wiek</button>
    <TestContext.Provider value={{ username: "samantha", userage }}>
      <UserAgeComp />
      <br />
      <UserNameComp />
    </TestContext.Provider>

    <br/>
    <button onClick={onClickLocalState}>Aktualizuj stan lokalny</button>
    &nbsp;<label>{localState}</label>
  </React.Fragment>);
}

```

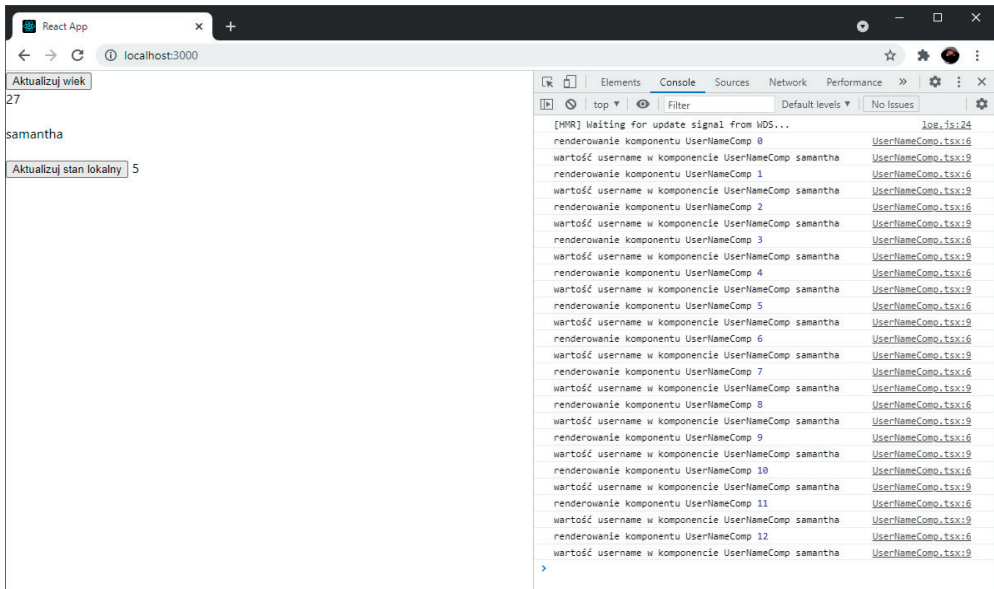
Zwróć uwagę na to, że wartość zmiennej `username`, "samantha", jest podawana na stałe i nigdy się nie zmienia. Jak widać, komponent `UserNameComp` został umieszczony wewnątrz komponentu `TestContext`.

6. Jeśli teraz uruchomisz ten kod i klikniesz przycisk kilka razy, to powinieneś zobaczyć wyniki podobne do tych z rysunku 7.5.

Jak widać, komponent `UserNameComp` jest renderowany po każdej zmianie, nawet jeśli będzie to jedynie zmiana zmiennej `localState`. Ale dlaczego tak się dzieje? Otóż `TestContext` jest komponentem, takim samym jak każdy inny komponent Reacta. A zatem, jeśli jego komponent nadrzędny, `ContextTester`, zostanie ponownie wyrenderowany, to operacja ta wymusi ponowne wyrenderowanie wszystkich jego komponentów podrzędnych. To właśnie z tego powodu komponent `UserNameComp` będzie ponownie renderowany, choć nie używa zmiennej `userage`.

Jak widać, korzystanie z mechanizmu React Context wiąże się z pewnymi problemami i jeśli o mnie chodzi, to uważam, że w przypadku dużych aplikacji, jeśli trzeba będzie wybierać pomiędzy Reduxem i React Context, to lepiej będzie zastosować Reduxa, choć na pewno będzie to rozwiązanie bardziej złożone.

W tym punkcie rozdziału przedstawiłem podstawy korzystania z kontekstu Reacta — React Context. Jest to rozwiązanie dość proste i to zarówno do nauki, jak i stosowania. Niemniej jednak, ze względu na jego stosunkowo prosty projekt, w bardziej złożonych projektach preferowane będzie korzystanie z bardziej wyrafinowanych mechanizmów zarządzania stanem globalnym.



Rysunek 7.5. Operacje ponownego renderowania komponentu w przypadku korzystania z React Context

## Prezentacja frameworka React Router

React Router jest najpopularniejszym frameworkiem do obsługi mechanizmu trasowania w aplikacjach Reacta. React Router jest dość prosty do nauki oraz do stosowania. Jaki już zaznaczyłem w rozdziale 4., pt. „Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React”, trasowanie (ang. *routing*) jest wszechobecne w aplikacjach Reacta. Jest to możliwość, której użytkownicy aplikacji internetowych oczekują, dlatego też nauczenie się korzystania z routera Reacta jest koniecznością.

Podczas korzystania z routera Reacta „trasy” (ang. *routes*) są komponentami frameworka React Router, wewnątrz których umieszczane są komponenty naszej aplikacji, i to te komponenty są wyświetlane na ekranie. Innymi słowy, trasa jest logiczną reprezentacją wirtualnej lokalizacji (przy czym jako „wirtualną lokalizację” rozumiem tu adres URL stanowiący jedynie etykietę, która nie istnieje na żadnym serwerze). Te trasy routera pełnią rolę komponentów nadrzędnych, natomiast nasze komponenty prezentowane na ekranie są ich komponentami podrzędnymi. Czytając opis w książce trochę trudno zrozumieć to rozwiązanie, dlatego też przedstawię je na przykładzie.

1. W katalogu *rozdział07* utwórz nowy projekt Reacta; w tym celu wykonaj w panelu terminala poniższe polecenie:

```
npx create-react-app try-react-router --template typescript
```



2. Po zakończeniu tworzenia projektu przejdź do katalogu *try-react-router* i dodaj nowe, wymagane pakiety:

```
npm i react-router-dom @types/react-router-dom
```

Zwróć uwagę na to, że React Router jest dostępny w kilku wersjach — my będziemy korzystali z wersji dom.

3. Teraz zmodyfikuj plik *index.tsx*, a konkretnie: dodaj do aplikacji główny komponent Routera. Zaktualizuj plik *index.tsx* zgodnie z poniższym przykładem:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { BrowserRouter } from "react-router-dom";

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

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Jak widać, do kodu JSX dodaliśmy nowy komponent nadrzędny, *BrowserRouter*, wewnątrz którego umieściliśmy komponent *App*. *BrowserRouter* przypomina nieco komponent *Reduxa Provider*, gdyż podobnie jak on jest komponentem nadrzędnym udostępniającym umieszczonym w nim komponentom podrzędnym różne właściwości *props* związane z trasowaniem. Już za chwilę zajmiemy się tymi właściwościami, lecz najpierw musimy dokończyć przygotowywanie routera do użycia.

4. A zatem, skoro React Router zapewnia nam możliwości trasowania, musimy przygotować kilka przykładowych tras. Pamiętaj jednak, że trasy są w rzeczywistości jedynie pojemnikami na komponenty, które będą reprezentować poszczególne ekrany naszej aplikacji. Dlatego zaczniemy od przygotowania dwóch takich ekranów. Utwórz plik *ScreenA.tsx* i zapisz w nim następujący kod:

```
import React from "react";

const ScreenA = () => {
  return <div>Ekran A</div>;
};

export default ScreenA;
```

To bardzo prosty komponent, który wyświetla w przeglądarce tekst **Ekran A**.

5. A teraz utwórz plik *ScreenB.tsx* i zapisz w nim poniższy kod:

```
import React from "react";

const ScreenB = () => {
  return <div>Ekran B</div>;
};

export default ScreenB;
```

Także ten komponent jest bardzo prosty — ogranicza się on do wyświetlenia w przeglądarce tekstu **Ekran B**.

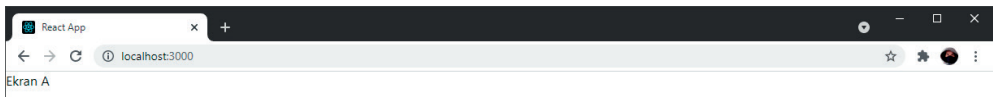
6. A teraz wypróbujmy trasy. Otwórz plik *App.tsx* i zmodyfikuj jego kod zgodnie z poniższym przykładem:

```
import React from 'react';
import './App.css';
import { Switch, Route } from "react-router-dom";
import ScreenA from "./ScreenA";
import ScreenB from "./ScreenB";

function App() {
  return (
    <Switch>
      <Route exact={true} path="/" component={ScreenA} />
      <Route path="/b" component={ScreenB} />
    </Switch>
  );
}

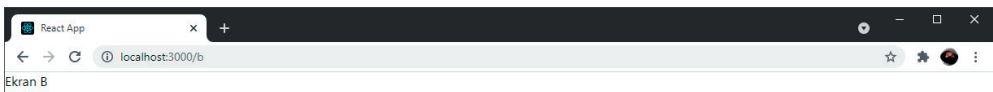
export default App;
```

Dzięki wprowadzonym zmianom aktualnie aplikacja będzie renderować trasy na podstawie kilku parametrów. Komponent `Switch` jest komponentem nadrzędnym, określającym trasę, którą należy wybrać, przy czym robi to porównując adres URL wpisany w przeglądarce z właściwością `path` poszczególnych komponentów `Route`. Na przykład, kiedy uruchomimy aplikację i wybierzemy trasę `/` (czyli główny ekran aplikacji), zobaczymy w przeglądarce stronę przedstawioną na rysunku 7.6.



Rysunek 7.6. Aplikacja prezentująca trasę z komponentem `ScreenA`.

Kiedy natomiast wybierzemy trasę `/b`, zobaczymy inną stronę, przedstawioną na rysunku 7.7.



Rysunek 7.7. Aplikacja prezentująca trasę z komponentem `ScreenB`

A zatem, zgodnie z tym, co zaznaczyłem na początku, trasy React Routera są zwyczajnymi komponentami Reacta. Może się to wydawać nieco dziwne zważywszy, że nie prezentują żadnego interfejsu użytkownika. Niemniej jednak, są one komponentami nadrzędnymi, które wyświetlają swoje komponenty podrzędne, choć same nie mają własnego interfejsu użytkownika.

Już wiemy, że podczas uruchamiania aplikacji, w pierwszej kolejności wykonywany jest plik *index.tsx*. I to właśnie w nim jest umieszczona główna usługa React Routera. Odczytuje ona bieżący adres URL, przegląda trasy zdefiniowane w pliku *App.tsx* i wybiera tę, której ścieżka odpowiada adresowi URL. Kiedy już zostanie określona pasująca trasa, aplikacja wyświetla jej komponent podrzędny. A zatem, w przypadku trasy z parametrem `path="/b"`, aplikacja wyświetli komponent `ScreenB`.

Przyjrzymy się teraz bardziej szczegółowo kodowi związanemu z trasami. Jeśli przyjrzymy się komponentom definiującym trasy, zauważymy, że w pierwszym z nich została użyta właściwość `exact`. Informuje ona React Router, że podczas dopasowywania adresu URL do tej trasy nie należy używać wyrażeń regularnych, a poszukać dokładnego, literalnego dopasowania. Kolejną używaną właściwością jest `path`. Zgodnie z przypuszczeniami, określa ona oczekiwany adres URL, a w zasadzie jego część: ścieżkę po nazwie domeny. Domyślnie sprawdzany jest warunek „zawierania”, czyli czy bieżący adres URL zawiera tę samą wartość, co właściwość `path`, a React Router zaakceptuje i wyświetli pierwszą odnaniezoną trasę spełniającą ten warunek, chyba że będzie w niej podana właściwość `exact`.

Kolejną właściwością komponentów `Route` jest `component`, która, jak łatwo się domyślić, określa komponent podrzędny, który należy wyrenderować. I w przypadku prostych rozwiązań zastosowanie tej właściwości w zupełności nam wystarczy. A co zrobić, kiedy będziemy chcieli przekazać do komponentu podrzędnego jakieś właściwości *props*? React Router udostępnia kolejną właściwość, o nazwie `render`, pozwalającą na zastosowanie tak zwanej **właściwości renderującej** (ang. *render property*).

Właściwość `render` to właściwość, której parametrem jest funkcja. Kiedy komponent nadrzędny renderuje swoją zawartość, w ramach tej operacji wywołuje także funkcję przekazaną jako właściwość `render`. Przyjrzyjmy się temu rozwiązaniu na przykładzie.

1. Utwórz nowy plik o nazwie *Screenc.tsx* i zapisz w nim następujący kod:

```
import React, { FC } from "react";

interface ScreenCProps {
  message: string;
}

const ScreenC: FC<ScreenCProps> = ({ message }) => {
  return <div>{message}</div>;
};

export default ScreenC;
```

Jak widać, ten komponent jest bardzo podobny do dwóch pozostałych. Niemniej jednak będzie do niego przekazywana właściwość *props* o nazwie `message`, a komponent będzie wyświetlać jej zawartość. Zobaczmy zatem, w jaki sposób możemy przekazać wartość tej właściwości, używając do tego celu właściwości `render` React Routera.

2. Zaktualizuj komponent App i dodaj do niego komponent ScreenC jako kolejną trasę:

```
import React from 'react';
import './App.css';
import { Switch, Route } from "react-router-dom";
import ScreenA from "./ScreenA";
import ScreenB from "./ScreenB";
import ScreenC from "./ScreenC";

function App() {
  const renderScreenC = (props: any) => {
    console.log("Właściwości props komponentu ScreenC", props);
    return <ScreenC {...props} message="To jest Ekran C" />;
  };

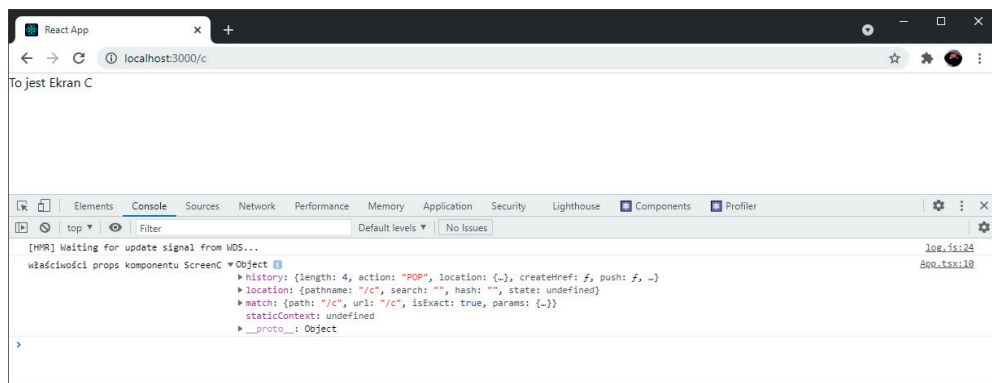
  return (
    <Switch>
      <Route exact={true} path="/" component={ScreenA} />
      <Route path="/b" component={ScreenB} />
      <Route path="/c" render={renderScreenC} />
    </Switch>
  );
}

export default App;
```

Jak widać, utworzyliśmy funkcję o nazwie `renderScreenC`, która pobiera parametr `props`, a następnie przekazuje go do komponentu `ScreenC`, który z kolei jest zwracany jako wynik wywołania funkcji. Oprócz parametru `props`, do komponentu `ScreenC` przekazujemy także właściwość `message` o wartości `"To jest Ekran C"`. Gdybyśmy w komponencie `Route` zastosowali właściwość `component`, nie moglibyśmy przekazać do komponentu `ScreenC` właściwości `message`; aby to było możliwe, konieczne jest zastosowanie właściwości `render`.

3. Kolejną wprowadzoną zmianą jest dodanie nowego komponentu `Route`, w którym zastosowaliśmy właściwość `render` i przekazali do niej zdefiniowaną wcześniej funkcję `renderScreenC`. Jeśli teraz wpiszesz w przeglądarce ścieżkę `"/c"`, w przeglądarce zostanie wyświetlona nowa strona, z komunikatem `"To jest Ekran C"` (patrz rysunek 7.8).

W kodzie funkcji `renderScreenC` umieściłem wywołanie `console.log` rejestrujące wszystkie właściwości `props` przekazywane do komponentu i — jak widać na rysunku 7.8 — wyświetlone zostały między innymi właściwości: `history`, `location` i `match`. Jak zapewne pamiętasz, funkcja `renderScreenC` ma sygnaturę o postaci: `(props: any) => { ... }`. Ten parametr `props` jest przekazywany przez komponent `Route`, a jego wartość określa usługa `React Router`. W dalszej części rozdziału wrócimy jeszcze do tych przekazywanych właściwości i przyjrzymy się im dokładniej.



**Rysunek 7.8.** Aplikacja przetestująca trasę z komponentem ScreenC

Wiesz już zatem, w jaki sposób można dokładniej kontrolować sposób renderowania komponentów dzięki zastosowaniu właściwości `render`. Jednak typowe adresy URL także zawierają parametry, które można przekazywać do poszczególnych ekranów aplikacji. Zobaczmy więc, w jaki sposób można to robić przy użyciu React Routera:

1. Zmodyfikuj komponent `Route` wyświetlający komponent `ScreenC` zgodnie z poniższym przykładem:

```
<Route path="/c/:userid" render={renderScreenC} />
```

Pole `userid` będzie teraz parametrem adresu URL.

2. W kolejnym kroku zmodyfikuj komponent `ScreenC` w taki sposób, by pobierał właściwości *props* przekazywane przez nadrzędny komponent `Route` i używał parametru `userid`:

```
import React, { FC } from "react";

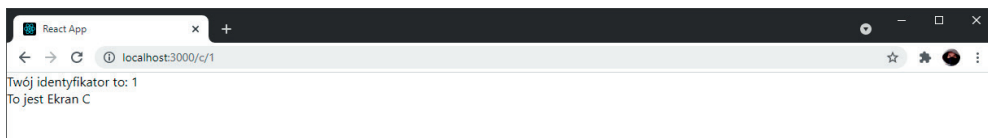
interface ScreenCProps {
  message: string;
  history: any;
  match: any;
}

const ScreenC: FC<ScreenCProps> = (props) => {
  return (
    <div>
      <div>{"Twój identyfikator to: " + props.match.params.userid}
      </div>
      <div>{props.message}</div>
    </div>
  );
};

export default ScreenC;
```

Jak widać, aby móc pobrać wszystkie składowe props przekazywane do komponentu bez konieczności wymienienia każdej z nich na liście parametrów, musieliśmy zrezygnować z wykorzystania mechanizmu destrukuryzacji.

Dzięki temu aktualnie nasz komponent pobiera jako własne właściwości *props* także składowe *history* i *match*, a dodatkowo wyświetla wartość pola *userid* odwołując się do niego przy użyciu wyrażenia *match.params.userid*. Ponieważ składowa *location* jest już dostępna w obiekcie *history*, nie dodałem jej do interfejsu *ScreenCProps*. Zmodyfikowaną postać ekranu C naszej aplikacji przedstawiłem na rysunku 7.9.



**Rysunek 7.9.** Aplikacja przynajmniej jedną trasę z komponentem *ScreenC* z wykorzystaniem parametru

Jak widać na rysunku, na stronie został wyświetlony parametr *userid*, który w tym przykładzie ma wartość 1.

No dobrze, ten ostatni przykład prezentował nieco bardziej realistyczne zastosowanie *React Router*, jednak działanie tego frameworka ma jeszcze jedną ważną cechę charakterystyczną. W zasadzie *React Router* działa jak stos adresów URL. Innymi słowy, kiedy użytkownik odwiedza kolejne adresy URL w aplikacji, robi to w sposób liniowy: najpierw wyświetla adres A, potem B, potem być może wróci na stronę A, następnie przejdzie na C itd. W efekcie, historię stron odwiedzonych w przeglądarce można zapisywać w formie stosu, po którym możemy poruszać się do przodu oraz do tyłu (innymi słowy: wracać do wcześniej odwiedzonych stron). Ta cecha działania przeglądarek jest implementowana w głównej mierze przez obiekt *history* *React Router*a.

Spróbujmy zatem ponownie zmodyfikować naszą przykładową aplikację i zastosować w niej niektóre możliwości, jakie zapewnia obiekt *history*:

#### 1. Zaktualizuj komponent *ScreenC* zgodnie z poniższym przykładem:

```
import React, { FC, useEffect } from "react";

interface ScreenCProps {
  message: string;
  history: any;
  match: any;
}

const ScreenC: FC<ScreenCProps> = (props) => {
  useEffect(() => {
    setTimeout(() => {
      props.history.push("/");
    }, 3000);
  });

  return (
    <div>
```

```

        <div>{"Twój identyfikator to: " + props.match.params.
        ↪userid}</div>
        <div>{props.message}</div>
    </div>
  );
};

```

```
export default ScreenC;
```

Jak widać, w kodzie komponentu ScreenC zastosowaliśmy funkcję `useEffect`. Przy jej użyciu wywołujemy funkcję `setTimeout`, która po 3 sekundach wywołuje funkcję `history.push`, a ta z kolei wykonuje przekierowanie na adres URL `"/`, a jak pamiętamy, dla tej ścieżki jest wyświetlany komponent ScreenA.

2. Skorzystajmy teraz z kolejnej funkcji udostępnianej przez obiekt `history`. Zmodyfikuj kod komponentu ScreenC zgodnie z poniższym przykładem:

```

import React, { FC } from "react";

interface ScreenCProps {
  message: string;
  history: any;
  match: any;
}

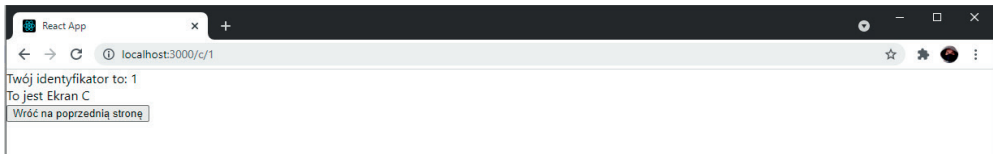
const ScreenC: FC<ScreenCProps> = (props) => {
  const onClickGoback = () => {
    props.history.goBack();
  };

  return (
    <div>
      <div>{"Twój identyfikator to: " + props.match.params.
      ↪userid}</div>
      <div>{props.message}</div>
      <div>
        <button onClick={onClickGoback}>Wróć na poprzednią
        ↪stronę</button>
      </div>
    </div>
  );
};

export default ScreenC;

```

Tym razem utworzyliśmy przycisk, którego kliknięcie spowoduje podjęcie próby wrócenia na poprzednią stronę przy użyciu funkcji `history.back`. Aby przetestować ten kod, musisz w pierwszej kolejności wejść na stronę o adresie `localhost:3000/b`, a następnie na stronę `localhost:3000/c/2`. Kiedy to zrobisz, w przeglądarce zostanie wyświetlona strona przedstawiona na rysunku 7.10.



**Rysunek 7.10.** Aplikacja przetestująca trasę z komponentem ScreenC z przyciskiem do powrotu

3. Jak widać na rysunku 7.10, teraz na stronie jest widoczny przycisk *Wróć na poprzednią stronę*. Kiedy go klikniesz, wrócisz na stronę o adresie `"/b"`.
4. I jeszcze jedna rzecz do sprawdzenia: ostatnio także do React Routera zostały dodane *hooki*. Dzięki temu nie trzeba już przekazywać właściwości tras do komponentów podrzędnych przy użyciu właściwości *props* — można je pobierać używając *hooków*. Poniżej przedstawiłem przykład takiego rozwiązania:

```
import React, { FC } from "react";
import { useHistory, useParams } from "react-router-dom";
```

W powyższym fragmencie kodu importujemy nowe funkcje React Routera.

```
interface ScreenCProps {
  message: string;
  history: any;
  match: any;
}

const ScreenC: FC<ScreenCProps> = (props) => {
  // useEffect(() => {
  //   setTimeout(() => {
  //     props.history.push("/");
  //   }, 3000);
  // });

  const history = useHistory();
  const { userid } : any = useParams();
```

Z kolei w tym fragmencie zastosowaliśmy funkcję `useHistory`, by pobrać obiekt `history`, oraz funkcję `useParams`, by pobrać wartość parametru `userid`.

```
const onClickGoback = () => {
  //props.history.goBack();
  history.goBack();
};

return (
  <div>
    {/*
    <div>{"Twój identyfikator to: " + props.match.params.userid}</div>
    */}
    <div>{"Twój identyfikator to: " + userid}</div>
    <div>{props.message}</div>
  </div>
```



```

        <button onClick={onClickGoback}>Wróć na poprzednią
        ↪stronę</button>
      </div>
    </div>
  );
};

export default ScreenC;
```

I w końcu w tym ostatnim fragmencie zastosowaliśmy pobrane wcześniej informacje w kodzie generowanym przez komponent. Jak widać, korzystanie z nowych *hooków* React Routera jest łatwe i mile.

Oczywiście zarówno obiekt *history*, jaki i sam React Router mają znacznie więcej innych możliwości, jednak informacje, które tu zamieściłem, stanowią dobre wprowadzenie do tych zagadnień, a my w następnych rozdziałach, w których zaczniemy pisać większą aplikację, wykorzystamy także kolejne możliwości React Routera.

Trasowanie jest jednym z kluczowych elementów pisania aplikacji Reacta. Trasy pozwalają użytkownikom orientować się w którym miejscu aplikacji aktualnie się znajdują i mogą ułatwiać określanie bieżącego kontekstu. Z kolei nam, programistom, trasowanie zapewnia możliwość tworzenia w strukturze aplikacji logicznych sekcji i grupowania powiązanych ze sobą elementów. React Router zapewnia te wszystkie możliwości, oddając nam do dyspozycji bogate możliwości programistyczne, pozwalające na tworzenie w aplikacji wyrafinowanego systemu tras.

## Podsumowanie

W tym rozdziale przedstawiłem dwa spośród najważniejszych frameworków stosowanych podczas tworzenia aplikacji Reacta. Redux jest wyszukanym narzędziem do zarządzania globalnym stanem w aplikacjach. Z kolei React Router służy do obsługi klienckich adresów URL i pozwala na korzystanie z nich w sposób podobny do klasycznych adresów URL.

Stosowanie rozwiązań o wysokiej jakości, takich jak Redux i React Router, pozwala na tworzenie lepszego kodu. A to z kolei ułatwia nam zapewnianie użytkownikom naszych aplikacji jak najlepszych doświadczeń.

Tym samym dotarliśmy do końca drugiej części niniejszej książki, poświęconej technologiom klienckim. W części trzeciej skoncentrujemy się na technologiach serwerowych.





C Z Ę Ś Ć

# Tworzenie usług internetowych z użyciem Expressa i GraphQL-a

Z tej części książki dowiesz się jak pisać usługi internetowe oraz w jaki sposób zastosowanie frameworka Express i języka GraphQL pozwala poprawiać ich wydajność.

Ta część książki składa się z następujących rozdziałów:

- Rozdział 8. — „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa”;
- Rozdział 9. — „Czym jest GraphQL?”;
- Rozdział 10. — „Konfiguracja projektu Expressa z zależnościami od języków TypeScript i GraphQL”;
- Rozdział 11. — „Czego się nauczysz — aplikacja internetowego forum”;
- Rozdział 12. — „Tworzenie klienta Reacta na potrzeby aplikacji internetowego forum”;
- Rozdział 13. — „Przygotowywanie stanu sesji przy użyciu Expressa i Redisa”;
- Rozdział 14. — „Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM”;
- Rozdział 15. — „Dodawanie schematu GraphQL — część 1.”;
- Rozdział 16. — „Dodawanie schematu GraphQL — część 2.”;
- Rozdział 17. — „Wdrażanie w chmurze AWS”;



# Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa

W tym rozdziale poznasz sposoby pisania programów dla środowiska Node oraz stosowania frameworka Express. Wyjaśnię w nim, w jaki sposób Node może nam pomóc w tworzeniu wydajnych usług internetowych. Oprócz tego wytłumaczę, jakie są związki środowiska Node oraz frameworka Express oraz jak można ich używać razem w celu tworzenia API usług internetowych.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- wyjaśnieniem sposobu działania środowiska Node;
- przedstawieniem możliwości środowiska Node;
- wyjaśnieniem, dlaczego Express ułatwia pisanie rozwiązań przeznaczonych dla środowiska Node;
- przedstawieniem możliwości frameworka Express;
- utworzeniem internetowego API przy użyciu Expressa.

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś dysponować podstawową wiedzą z zakresu tworzenia rozwiązań internetowych w języku JavaScript. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial08 (Chap8)*.

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog o nazwie *rozdzial08*.

## Wyjaśnienie sposobu działania środowiska Node

Node jest jednym z najpopularniejszych na świecie frameworków języka JavaScript. Jest on używany jako podstawowa technologia zapewniająca działanie milionów witryn WWW. Popularność środowiska Node wynika z wielu powodów. Jednym z nich jest to, że pisanie programów działających w Node jest dość proste. Oprócz tego, Node jest bardzo szybkie, a kiedy używa się go wraz z takimi rozwiązaniami, jak klastry oraz wątki robocze, zapewnia także bardzo dużą skalowalność. Poza tym, ponieważ jest to środowisko języka JavaScript, pozwala na tworzenie kompletnych aplikacji, z elementami klienckimi oraz serwerowymi, i to pisanymi w tym samym języku programowania. Wszystkie te cechy sprawiają, że Node jest rewelacyjnym wyborem dla wszystkich, którzy chcą tworzyć rozwiązania internetowe. W tym podrozdziale przedstawię architekturę Node.js oraz wyjaśnię, w jaki sposób zapewnia ono wysoką wydajność działania.

Na wstępie zaznaczę, że Node.js nie jest rozwiązaniem typowo serwerowym. Jest to środowisko uruchomieniowe ogólnego przeznaczenia, a nie jedynie serwer WWW. Node obsługuje język JavaScript, zapewniając przy tym możliwości, którymi zazwyczaj język ten nie dysponuje, takie jak dostęp do systemu plików czy też odbieranie żądań sieciowych.

Aby wyjaśnić, jak działa Node, posłużę się analogią do przeglądarki internetowej. Otóż przeglądarki WWW także są środowiskami wykonawczymi dla naszych skryptów JavaScript (jak również dla kodu HTML i CSS). Działanie przeglądarki bazuje na korzystaniu z wbudowanego silnika JavaScriptu, udostępniającego podstawowe możliwości tego języka. Taki silnik składa się z interpretera JavaScriptu, który odczytuje kod skryptów napisanych prawidłowo pod względem składniowym, oraz z maszyny wirtualnej, która wykonuje kod na różnych urządzeniach.

Ponad warstwą języka JavaScript, przeglądarki WWW zapewniają bezpieczny kontener pamięci, tak zwaną piaskownicę (ang. *sandbox*), w której mogą być wykonywane nasz aplikacje. Jednak oprócz tego wszystkiego, przeglądarki WWW udostępniają także dodatkowe możliwości języka JavaScript, określane ogólnie jako „internetowy interfejs programowania” (ang. *web API*; przy czym nie chodzi tu o rozwiązanie serwerowe, a o jego kliencki odpowiednik). Ten internetowy API uzupełnia podstawowy silnik języka JavaScript, zapewniając, między innymi, dostęp do obiektowego modelu dokumentu (ang.: **Document Object Model**, w skrócie **DOM**), dzięki któremu skrypty JavaScript mogą operować na elementach HTML. Oprócz tego internetowy API obsługuje żądania sieciowe, dzięki którym można wykonywać asynchroniczne odwołania do innych komputerów, zapewnia obsługę WebGL, wzbogacając tym samym możliwości prezentacji graficznej, i tak dalej. Pełną listę możliwości można znaleźć na stronie <https://developer.mozilla.org/en-US/docs/Web/API>.

To wszystko są dodatkowe możliwości, wykraczające poza to, co JavaScript potrafi domyślnie; co zresztą, jeśli się nad tym głębiej zastanowić, ma sens, gdyż JavaScript jest w zasadzie zwyczajnym językiem programowania i nie jest charakterystyczny dla żadnej konkretnej platformy, w tym także dla internetu.

Node korzysta z podobnego modelu co przeglądarki, gdyż także ono zapewnia dostęp do podstawowego silnika języka JavaScript (konkretnie jest to silnik V8, używany także w przeglądarce Chrome firmy Google) i udostępnia kontener uruchomieniowy, w którym może być wykonywany nasz kod. Niemniej jednak, ponieważ Node nie jest przeglądarką WWW, to udostępnia zupełnie inne możliwości dodatkowe, które nie są powiązane z prezentacją graficzną.

A zatem, czym jest Node? Jest to środowisko uruchomieniowe ogólnego przeznaczenia, stworzone z myślą o zapewnieniu bardzo wysokiej wydajności działania oraz skalowalności. Node umożliwia tworzenie bardzo wielu różnych rodzajów aplikacji, w tym skryptów do zarządzania komputerem oraz programów pełniących rolę terminali. Jednak wysokie możliwości skalowania sprawiają, że Node doskonale nadaje się także do wykorzystania jako serwer WWW.

Node ma wiele możliwości, które sprawiają, że jest wszechstronnym środowiskiem wykonawczym, jednak prawdziwym sercem Node jest **libuv**. Libuv to usługa Node, napisana w języku C, która stanowi interfejs pośredniczący z jądrem systemu operacyjnego i udostępnia asynchroniczne mechanizmy obsługi wejścia-wyjścia. Aby zapewnić błyskawiczny dostęp do swoich usług, libuv używa rozwiązania nazywanego pętlą zdarzeń (ang. *event loop*); zajmujemy się nią dokładniej już niebawem. Libuv jest przesłonięta warstwą systemu dodatków, które można porównać z rozszerzeniami przeglądarki Chrome. Pozwala ona programistom rozszerzać Node kodem pisany w języku C++ i dodawać do niego nowe możliwości o dużej efektywności działania, które domyślnie nie są dostępne. Co więcej, aby zapewnić programistom używającym języka JavaScript możliwość wywoływania kodu napisanego w C++, dostępny jest system powiązań JavaScript-C++ o nazwie Addons. Przyjrzyjmy się teraz nieco dokładniej libuv oraz jej pętli zdarzeń.

## Pętla zdarzeń

Sercem Node są dwa elementy: libuv oraz pętla zdarzeń. To właśnie one są kluczowymi cechami Node, zapewniającymi jego ogromną skalowalność. Głównym zadaniem libuv jest zapewnienie dostępu do asynchronicznych mechanizmów obsługi wejścia-wyjścia (I/O) systemu operacyjnego, na którym działa środowisko Node (a jest ono dostępne w wersjach przeznaczonych dla systemów Linux, MacOS oraz Windows). Jednak korzystanie z tych mechanizmów nie zawsze jest możliwe, więc libuv udostępnia także pętlę zdarzeń, która umożliwia wykonywanie operacji synchronicznych w sposób, który praktycznie można uznać za asynchroniczny. Jest to możliwe dzięki wykonywaniu tych operacji w wątku. Możliwości korzystania z liczników, tworzenia połączeń sieciowych, korzystania z gniazd systemu operacyjnego oraz używania systemu plików, wszystkie są zapewniane przez libuv.

A zatem, czym jest pętla zdarzeń? Otóż jest to działający w ramach usługi libuv mechanizm wykonywania wątków, który przypomina nieco pętlę obsługi zdarzeń przeglądarki Chrome i zapewnia możliwość iteracyjnego wykonywania asynchronicznych wywołań zwrotnych. Poniżej opisałem sposób działania tej pętli zdarzeń na dość wysokim poziomie abstrakcji.

Kiedy zaczyna być wykonywane jakieś zadanie asynchroniczne, zostaje ono uruchomione przez pętlę zdarzeń. Proces działania pętli zdarzeń jest podzielny na powtarzające się fazy. Jak pokazałem na schemacie przedstawionym na rysunku 8.1, w pierwszej kolejności są wykonywane **liczniki czasu** (ang. *timers*), a jeśli jakieś funkcje zwrotne tych liczników są już umieszczona w kolejce, to zostają wywołane jedna po drugiej (w przeciwnym razie, jeśli nie ma do wykonania żadnych funkcji zwrotnych liczników czasu, i jeśli jakieś liczniki właśnie zostały zakończone, to pętla zapisze w kolejce funkcje zwrotne, które należy wykonać). Następnie wykonywane są **oczekujące funkcje zwrotne** (czyli wywołania zwrotne ustawione przez system operacyjny — na przykład związane z błędami TCP), i tak dalej, aż do zakończenia listy z rysunku 8.1. Warto zwrócić uwagę na to, że zadania wykonywane przez libuv są ze swej natury asynchroniczne, natomiast funkcje zwrotne wcale nie muszą być takimi. Dlatego też istnieje możliwość zablokowania pętli zdarzeń i sprawienia, że nie wywoła ona kolejnej funkcji zwrotnej aż do momentu zakończenia bieżącej. Przybliżony sposób działania pętli zdarzeń Node.js przedstawiłem na rysunku 8.1.



Rysunek 8.1. Pętla zdarzeń Node.js, rysunek zaczerpnięty z dokumentacji Node



Te fazy można sobie także wyobrazić jako zadania asynchroniczne wraz z ich wywołaniami zwrotnymi.

Wszystkie frameworki oraz środowiska mają swoje mocne i słabe punkty. Największą zaletą Node jest wysoka skalowalność asynchronicznych operacji wejścia-wyjścia. Dzięki niej Node najlepiej nadaje się do stosowania w rozwiązaniach działających w sposób wysoce równoległy i wymagających obsługi wielu jednoczesnych połączeń. W nowszych wersjach Node, zaczynając od wersji 10.5, twórcy środowiska dodali do niego wątki robocze, zapewniając dzięki temu możliwości wielowątkowego przetwarzania zadań w dużym stopniu obciążających procesor, czyli głównie związanych z wykonywaniem długotrwałych, złożonych obliczeń. Jednak nie na tym polega główna siła Node.js. Jeśli chodzi o wykonywanie długotrwałych i złożonych zadań obliczeniowych, to zapewne istnieją inne, lepsze rozwiązania. Jednak my chcemy zastosować Node do stworzenia skalowalnego API na potrzeby aplikacji Reacta, a do tego celu Node nadaje się znakomicie.

W następnym podrozdziale zaczniemy dokładniej poznawać możliwości Node — zrobimy to pisząc własny kod, w którym nie będziemy korzystali z żadnych dodatkowych frameworków, takich jak Express czy Koa. W ten sposób nie tylko będziesz w stanie dokładniej zrozumieć jak działa Node, lecz także w przyszłości będzie Ci łatwiej wskazać różnice pomiędzy Node i Express.

## Prezentacja możliwości środowiska Node

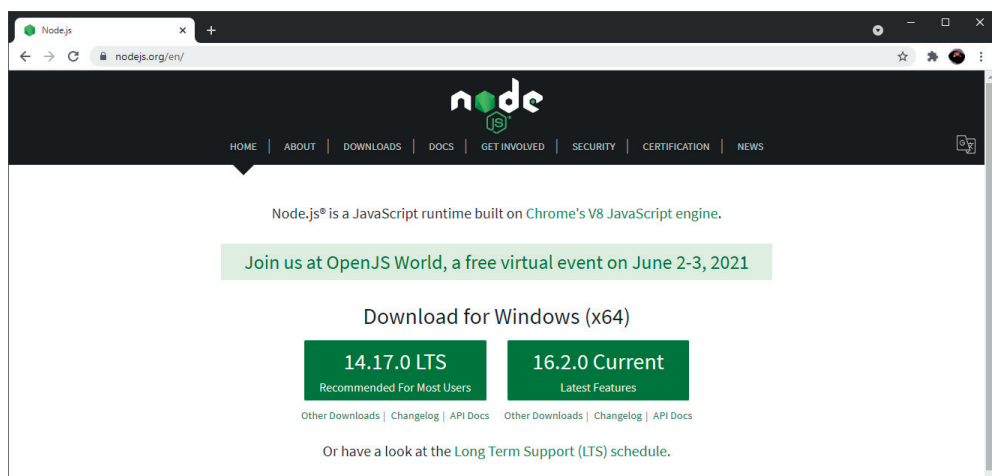
W poprzednim podrozdziale przedstawiłem ogólny, koncepcyjny opis Node.js i wyjaśniłem, dlaczego zapewnia on bardzo wysoką skalowalność. Z kolei w tym podrozdziale zaczniemy korzystać z tej cechy Node.js, a zrobimy to pisząc prosty kod uruchamiany w tym środowisku. Zaczniemy jednak od zainstalowania Node, skonfigurowania projektu i krótkiego przedstawienia API Node.js.

## Instalowanie Node

Zanim będziesz mógł zacząć pisać kod przeznaczony dla środowiska Node, musisz je zainstalować. Być może zrobiłeś to już wcześniej, aby pisać i uruchamiać aplikacje przedstawiane w poprzednich rozdziałach, jednak pomimo to chciałbym przypomnieć Ci, jak to się robi, gdyż środowisko Node.js jest dosyć często aktualizowane.

1. W przeglądarce wyświetl stronę <https://nodejs.org/>. Postać tej strony w czasie, kiedy była przygotowywana ta książka, przedstawiłem na rysunku 8.2.

W przypadku zastosowań produkcyjnych lepiej będzie zastosować bardziej zachowawcze podejście i wybrać wersję zapewniającą długie wsparcie, czyli **LTS** (od angielskich słów: **Long Time Support**); jednak ponieważ w tej książce chcemy poznać najnowsze możliwości, wybierzemy wersję aktualną — **Current**.

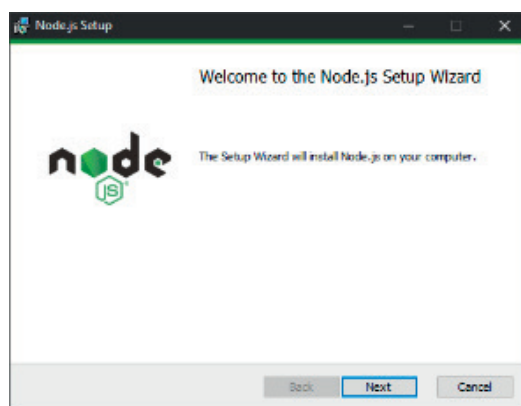


**Rysunek 8.2.** Witryna WWW Node

Ogólnie rzecz biorąc, nowsze wersje Node będą działały nieco szybciej, będą bezpieczniejsze i uzupełnione o poprawki błędów. Istnieje jednak prawdopodobieństwo, że w tych nowych wersjach pojawią się także nowe problemy, dlatego też w przypadku aktualizowania wersji Node na serwerze produkcyjnym należy zachować ostrożność.

Instalując Node uzyskujemy nie tylko samo środowisko uruchomieniowe, lecz także najnowszą wersję menedżera pakietów npm.

2. Po kliknięciu odnośnika umieszczonego na stronie zostaniesz poproszony o zapisanie programu instalacyjnego dostosowanego do aktualnie używanej platformy systemowej. Zapisz go na dysku, a następnie uruchom — powinieneś zobaczyć okno programu instalacyjnego, podobne do tego z rysunku 8.3.



**Rysunek 8.3.** Instalacja i konfiguracja Node

Wykonaj poszczególne etapy instalacji Node, zgodnie z instrukcjami wyświetlanymi przez program instalacyjny.

No dobrze, w takim razie dysponujesz już zainstalowanym lub zaktualizowanym środowiskiem Node oraz menedżerem pakietów npm. Jak już zaznaczałem wcześniej, Node nie jest jedynie frameworkiem serwerowym, lecz kompletnym środowiskiem uruchomieniowym, pozwalającym na tworzenie i uruchamianie aplikacji wielu różnych rodzajów. Na przykład Node dysponuje interfejsem wiersza poleceń, który nosi nazwę REPL. Jeśli otworzysz okno wiersza poleceń i wykonasz polecenie `node`, to przekonasz się, że uruchomione zostanie narzędzie (patrz rysunek 8.4) pozwalające na wpisywanie i wykonywanie instrukcji języka JavaScript.

```

C:\Users\prajc>node
Welcome to Node.js v14.17.0.
Type ".help" for more information.
> var test = 1
undefined
> console.log(test)
1
undefined
>

```

Rysunek 8.4. REPL — interfejs wiersza poleceń Node

W tej książce nie będziemy korzystać z narzędzia REPL, jednak wspomniałem tu o nim, żebyś wiedział, że ono istnieje i ewentualnie mógł z niego korzystać w swoich przyszłych projektach. Więcej informacji na jego temat możesz znaleźć w dokumentacji Node, na stronie [https://nodejs.org/api/repl.html#repl\\_design\\_and\\_features](https://nodejs.org/api/repl.html#repl_design_and_features). A gdybyś był ciekawy, dlaczego poniżej każdej z wykonanych instrukcji jest wyświetlane słowo `undefined`, to wynika to z faktu, że żadna z nich nie zwróciła żadnego wyniku, a w języku JavaScript taka sytuacja jest oznaczana przy użyciu słowa `undefined`.

No dobrze, zajmijmy się zatem stworzeniem naszej pierwszej aplikacji Node, a przy okazji poznamy parę możliwości tego środowiska:

1. Uruchom edytor VSCode, a następnie wyświetl w nim katalog *rozdzial08*.
2. Teraz utwórz w nim podkatalog o nazwie *try-node*.
3. Wewnątrz katalogu *try-node* utwórz plik o nazwie *app.js*. Póki co, aby uprościć prezentowane przykłady, będziemy unikali stosowania języka TypeScript.
4. Do kodu pliku *app.js* dodaj proste wywołanie funkcji `console.log`, takie jak to przedstawione poniżej:

```
console.log("Witaj, świecie!");
```

A teraz wykonaj skrypt *app.js*:

```
node app.js
```

Uzyskane wyniki przedstawiłem na rysunku 8.5.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
H:\NaukaTypeScriptu\rozdzial08\try-node> node app.js
Witam, Świecie!

H:\NaukaTypeScriptu\rozdzial08\try-node>
```

### Rysunek 8.5. Wykonanie skryptu app.js

Nie jest to szczególnie użyteczna aplikacja, jednak na jej przykładzie możesz się przekonać, że Node wykonuje standardowy kod pisany w języku JavaScript. A teraz spróbujmy zrobić coś bardziej przydatnego, a konkretnie spróbujmy odwołać się do lokalnego systemu plików. W tym celu wykonaj czynności opisane na poniższej liście:

1. W tym samym pliku, *app.js*, usuń wywołanie `console.log` i wpisz następujący wiersz kodu:

```
const fs = require("fs");
```

Ta instrukcja może Cię nieco zaskoczyć, gdyż nie jest to standardowy sposób importowania kodu. Niemniej jednak chciałem go tutaj pokazać, gdyż w starym kodzie Node w przeważającej większości przypadków używana jest składnia importu Common-JS. Musisz o tym pamiętać.

2. Następnie zapisz w pliku poniższy fragment kodu, który tworzy plik, a następnie odczytuje jego zawartość:

```
fs.writeFile("test.txt", "Witaj, świecie!", () => {
  fs.readFile("test.txt", "utf8", (err, msg) => {
    console.log(msg);
  });
});
```

Kiedy wykonasz ten skrypt, w panelu terminala zostaną wyświetlone wyniki przedstawione na rysunku 8.6, a w katalogu *try-node* pojawi się plik *test.txt*.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
H:\NaukaTypeScriptu\rozdzial08\try-node>node app.js
Witaj, świecie!

H:\NaukaTypeScriptu\rozdzial08\try-node>
```

### Rysunek 8.6. Wyniki wykonania skryptu app.js

Składnia wywołań `fs` jest dość niewygodna, gdyż korzysta z funkcji zwrotnych stosowanych w starym stylu. Środowisko Node było tworzone jeszcze zanim w języku JavaScript pojawiły się obietnice oraz słowa kluczowe `async` i `await`, dlatego też niektóre z funkcji jego API wciąż wymagają implementowania asynchroniczności w starym stylu, czyli z wykorzystaniem funkcji zwrotnych. Dostępna jest jednak nowa wersja pakietu `fs`, korzystająca z obietnic, a zatem pozwalająca także na stosowanie słów kluczowych `async` i `await`; możesz z niej skorzystać jeśli wolisz ten styl programowania. Poniżej przedstawiłem przykład jej użycia:

```
const fs = require("fs/promises");

(async function () {
  await fs.writeFile("test-promise.txt", "Witajcie, Obietnice!");
  const readTxt = await fs.readFile("test-promise.txt", "utf-8");
  console.log(readTxt);
})();
```

Zwróć uwagę na to, że do wywołania umieszczonej na głównym poziomie funkcji asynchronicznej używane jest rozwiązanie określane jako bezzwłocznie wywoływane wyrażenie funkcyjne — IIFE (ang. *Immediately Invoked Function Expression*).

Jeśli używasz starszej wersji Node, to — ponieważ pakiet `fs/promises` jest dostępny zaczynając od wersji 11 — możesz skorzystać z funkcji `promisify` pakietu `util`, która pozwala na opakowywanie starego typu wywołań używających funkcji zwrrotnych i pisanie kodu korzystającego ze słów kluczowych `async` i `await`.

3. Zauważyłeś zapewne, że na samym początku kodu używamy instrukcji `require` do zaimportowania pakietu `fs`. Spróbujmy teraz zastosować nowszą składnię importowania. W tym celu musisz wprowadzić dwie zmiany: zmienić rozszerzenie pliku z `.js` na `.mjs` oraz zmienić instrukcję na samym początku pliku na następującą:

```
import fs from "fs";
```

Kiedy uruchomisz plik `app.mjs`, przekonasz się, że działa tak samo, jak wcześniej. Zamiast zmieniać rozszerzenia pliku, mogliśmy także dodać do pliku `package.json` flagę `"type": "module"`, jednak w tym bardzo prostym przykładzie w ogóle nie używaliśmy `npm`. Co więcej, gdybyśmy zastosowali tę flagę globalnie, to w ogóle nie moglibyśmy już używać instrukcji `require`; a to może być problemem, gdyż niektóre starsze zależności `npm` wciąż wymagają użycia tego sposobu importowania.

Dostępna jest także starsza flaga stosowana w wierszu poleceń, `--experimental-modules`, która pozwala na użycie instrukcji `import`; jednak obecnie korzystanie z niej nie jest zalecane, a w nowszych wersjach Node należy jej unikać.

## Tworzenie prostego serwera Node

Dowiedziałeś się już, że Node bazuje na starszych technologiach języka JavaScript, takich jak funkcje zwrótne oraz moduły CommonJS. Środowisko to powstało jeszcze przed wprowadzeniem obietnic do języka JavaScript, jak również przed pojawieniem się nowszych wersji samego języka JavaScript, w tym także wersji ES6 i nowszych. Pomimo to Node działa doskonale i cały czas jest rozwijane, a później, kiedy dodamy do niego odpowiednie biblioteki, w większości przypadków będzie można w nim korzystać także z obietnic oraz słów kluczowych `async` i `await`.

Kolejny przykład aplikacji Node.js, który przedstawię, będzie trochę bardziej realistyczny. Zaczniemy od utworzenia nowego projektu przy użyciu `npm`:

1. Przejdź do głównego katalogu *rozdzial08* i utwórz w nim podkatalog *node-server*.
2. W panelu terminala przejdź do nowego podkatalogu i zainicjuj npm, wykonując poniższe polecenie:

```
npm init
```

3. Kiedy zostaniesz poproszony o podanie nazwy projektu, wpisz *node-server*, a na pozostałe pytania odpowiedz akceptując wartości domyślne.
4. W katalogu *node-server* utwórz plik *server.mjs* i zapisz w nim następujący kod:

```
import http from "http";
```

Nie przejmuj się, już niebawem zaczniemy pisać kod w języku TypeScript. Na razie zależy mi na zachowaniu jak największej prostoty, byśmy mogli skoncentrować się na poznawaniu Node.js.

5. W poprzednim punkcie zaimportowaliśmy bibliotekę *http* stanowiącą wbudowany element środowiska Node. Teraz możemy użyć funkcji *createServer*, żeby utworzyć obiekt serwera. Zwróć uwagę na to, że w wywołaniu funkcji *createServer* należy przekazać jeden argument: funkcję definiującą dwa parametry. Parametry te są zwyczajowo oznaczane jako *req* oraz *res*, a reprezentują odpowiednio obiekty typów *Request* i *Response*. Obiekt *Request* zawiera wszelkie składowe związane z żądaniem nadesłanym przez użytkownika, natomiast obiekt *Response* pozwala na modyfikowanie wszelkich aspektów odpowiedzi zanim zostanie ona odesłana użytkownikowi.

Na samym końcu funkcji obsługi przekazywanej w wywołaniu *createServer* jawnie kończymy obsługę żądania i zwracamy konkretny łańcuch znaków — w tym celu wywołujemy funkcję *res.end*. Gdybyśmy nie wywołali funkcji *end*, obsługa żądania nigdy nie zostałaby zakończona, a przeglądarka niczego by nie wyświetliła:

```
const server = http.createServer((req, res) => {
  console.log(req);
  res.end("Witaj, świecie!");
});
```

6. Teraz możemy już kazać naszemu nowemu obiektowi serwera, by czekał i nasłuchiwał żądań. W tym celu wywołujemy funkcję *listen*, do której należy przekazać numer portu oraz funkcję zwrotną, która wyświetla komunikat o uruchomieniu serwera:

```
const port = 8000;
server.listen(port, () => {
  console.log(`Serwer został uruchomiony na porcie ${port}`);
});
```

7. Teraz możemy już uruchomić serwer, wykonując skrypt *server.mjs* (upewnij się, że plik ma odpowiednie rozszerzenie, tzn. *.mjs*):

```
node server.mjs
```

Pamiętaj, że środowisko Node aktualnie nie dysponuje wbudowanym mechanizmem automatycznego wczytywania zmodyfikowanych plików. Dlatego, po wprowadzeniu jakichkolwiek zmian w naszym serwerze, będziesz musiał go zatrzymać i ponownie

uruchomić. Później, kiedy wzbogacimy możliwości naszego projektu, dodamy do niego pakiet, który wyeliminuje ten mankament.

8. Jeśli teraz wyświetlisz w przeglądarce stronę o adresie `http://localhost:8000`, to zostanie w niej wyświetlony tekst Witaj, świecie!, a w panelu terminala zostaną wyświetlone wszystkie informacje o obiekcie żądania, których fragment przedstawiłem na rysunku 8.7.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS
<ref *2> IncomingMessage {
  _readableState: ReadableState {
    objectMode: false,
    highWaterMark: 16384,
    buffer: BufferList { head: null, tail: null, length: 0 },
    length: 0,
    pipes: [],
    flowing: null,
    ended: false,
    endEmitted: false,
    reading: false,
    sync: true,
    needReadable: false,
    emittedReadable: false,
    readableListening: false,
    resumeScheduled: false,
    errorEmitted: false,
    emitClose: true,
    autoDestroy: false,
    destroyed: false,
    errored: false,

```

**Rysunek 8.7.** Nasz pierwszy działający serwer Node

Panel terminala przedstawiony na rysunku 8.7 przedstawia obiekt req oraz kilkanaście początkowych spośród jego składowych. Oczywiście w dalszej części rozdziału przyjrzymy się obiektom Request i Response nieco bardziej szczegółowo.

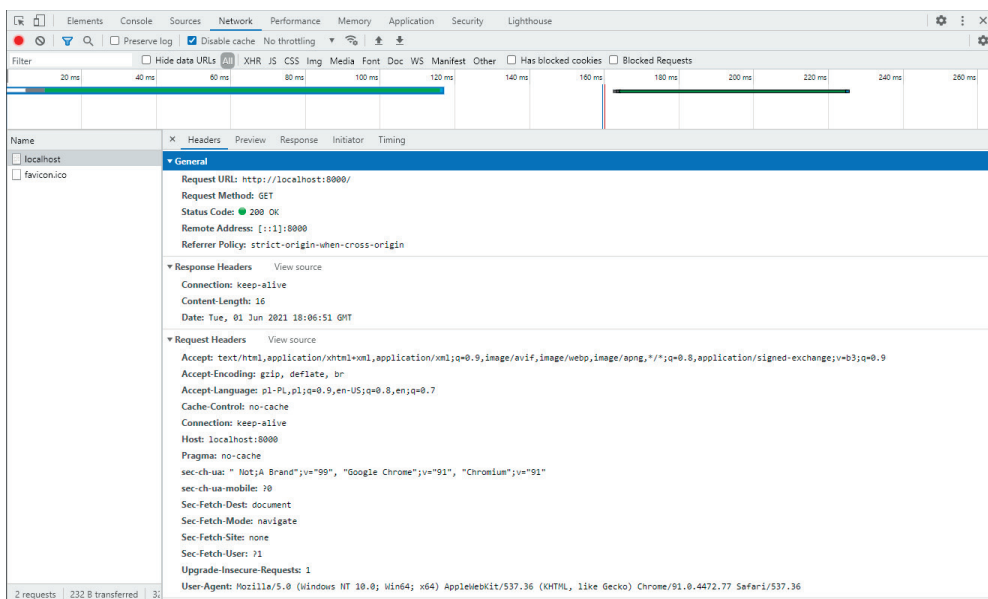
Kolejnym interesującym aspektem działania tego naszego serwera jest to, że niezależnie od adresu URL wpisanego w przeglądarce, zawsze będzie on wyświetlał ten sam tekst Witaj, świecie!. Ten sposób jego działania wynika z faktu, że nie zastosowaliśmy żadnego mechanizmu trasowania. Trasowanie, czyli obsługa tras, jest kolejnym zagadnieniem, którego musisz się nauczyć, by móc prawidłowo używać Node.js jako serwera WWW.

Możesz w nieskończoność odświeżać stronę w przeglądarce, a serwer za każdym razem zwróci ten sam łańcuch znaków: Witaj, świecie!. Łatwo się przekonaasz, że serwer będzie działał bezustannie, niezależnie od tego, jak wiele żądań do niego wyślesz, a nie jak klasyczny skrypt wyświetli wynik i zakończy działanie. Przyczyną takiego zachowania naszego serwera jest zastosowanie pętli zdarzeń, podstawowego mechanizmu Node.js — pętla ta będzie działać w nieskończoność, posłusznie oczekując na nowe zadania i wykonując je.

Gratuluje, w ten sposób napisałeś i uruchomiłeś swój pierwszy serwer Node! Oczywiście, te początki są dość skromne, jednak możesz już używać przeglądarki, by przysyłać na ten serwer żądania i wyświetlać odpowiedzi. Robisz więc postępy!

## Żądania i odpowiedzi

Kiedy żądanie przesłane przez przeglądarkę trafia na serwer, framework serwera tworzy dwa obiekty: żądania (Request) oraz odpowiedzi (Response); co więcej, ogólnie rzecz biorąc, w ten sposób działają wszystkie serwery. Te dwa obiekty reprezentują odpowiednio dane związane z żądaniem przesłanym przez przeglądarkę oraz z odpowiedzią, która zostanie do niej odesłana. Przyjrzyjmy się dokładniej tym dwóm obiektom, aby przekonać się, co zawierają; najprościej będzie do tego użyć narzędzi samej przeglądarki. W przeglądarce Chrome wyświetl narzędzia dla programistów, przejdź na kartę *Network*, a następnie odśwież stronę; wyniki, które zobaczysz, będą podobne do tych z rysunku 8.8.



**Rysunek 8.8.** Żądanie wyświetlone na karcie Network w narzędziach dla programistów przeglądarki Chrome

Rysunek 8.8 przedstawia żądanie z punktu widzenia przeglądarki WWW, w programie serwera działającym w środowisku Node dostępnych jest znacznie więcej informacji o żądaniu. Jednak zanim spróbujemy stworzyć serwer WWW z prawdziwego zdarzenia, musisz lepiej zrozumieć, z czego składa się żądanie HTTP. Dlatego w kolejnych podpunktach przedstawę kilka jego najważniejszych elementów i dokładnie opiszę ich znaczenie.

## Adres URL żądania

Reprezentuje on kompletną ścieżkę URL przesłaną na serwer. Jednak serwer musi znać tę kompletną ścieżkę, dlatego, że mogą być w niej przesyłane dodatkowe informacje. Na przykład, gdyby przesłany adres URL miał postać: `http://localhost:8000/home?userid=1`, to okazałoby się, że zawiera całkiem sporo informacji. Przede wszystkim informowałby serwer, że chodzi o stronę lub dane API umieszczone w podkatalogu *home*. Dzięki temu serwer będzie



mógł ograniczyć zwracane dane do wskazanego dokumentu HTML lub danych skojarzonych z podanym adresem URL. Co więcej, w adresie został podany parametr o nazwie `userid` (sekcja parametrów zaczyna się od znaku zapytania, a jeśli tych parametrów jest więcej, to powinny być od siebie oddzielone znakiem `&`), którego serwer może użyć do zwrócenia unikalnych danych dostosowanych do tego żądania.

## Metoda żądania

Metoda żądania reprezentuje tak zwany „czasownik HTTP”. Ten czasownik jest jedynie opisem informującym serwer o czynności, o której wykonanie prosi klient. Domyślnym czasownikiem jest GET, co oznacza, zgodnie z tym, co sugeruje jego nazwa, że przeglądarka prosi o odczytanie jakichś danych. Do innych powszechnie stosowanych czasowników należą: POST, który oznacza utworzenie lub wstawienie czegoś, PUT, oznaczający aktualizację, oraz DELETE, oznaczający usunięcie. W rozdziale 9., pt. „Czym jest GraphQL”, przekonasz się, że GraphQL używa jedynie metody POST; nie jest to jednak żaden błąd, gdyż czasowniki HTTP nie są żadnymi sztywnymi regułami i raczej należy je traktować jako wytyczne. Kolejną rzeczą, na którą warto zwrócić uwagę, jest to, że w przypadku stosowania metody GET wszelkie parametry muszą być przekazywane w adresie URL, w sposób, który przedstawiłem w przykładzie zamieszczonym w poprzednim podpunkcie rozdziału. Z kolei w przypadku stosowania metody POST, wszystkie parametry są zapisywane w treści żądania. Więcej informacji na temat tych różnic podałem w podrozdziale pt. „Przedstawienie możliwości frameworka Express”.

## Kod statusu

Wszystkie odpowiedzi HTTP będą zwracały kod statusu oznaczający wynik obsługi żądania. Na przykład kod statusu 200 oznacza pomyślne wykonanie żądania. Nie będę tu zamieszczał pełnej listy kodów, jednak warto, żebyś poznał te najczęściej stosowane, gdyż mogą się one przydać podczas debugowania aplikacji; zestawienie tych kodów przedstawiłem w tabeli 8.1.

**Tabela 8.1.** Kody statusu odpowiedzi HTTP

Kod statusu	Znaczenie	Opis
201	Utworzono	Oznacza pomyślne wykonanie operacji wstawienia czegoś.
400	Złe żądanie	Oznacza, że żądanie w formie, w jakiej zostało wysłane przez przeglądarkę, zawierało w sobie jakiś błąd.
401	Dostęp nieautoryzowany	Oznacza brak uprawnień dostępu do podanego adresu URL.
403	Dostęp zabroniony	Ogólnie rzecz biorąc oznacza, że podany adres URL nie jest dostępny publicznie.
404	Nie znaleziono	Podanego adresu URL nie znaleziono na serwerze.
500	Wewnętrzny błąd serwera	Oznacza, że podczas przetwarzania żądania na serwerze wystąpił wyjątek lub jakiś inny błąd.

Tabela 8.1. Kody statusu odpowiedzi HTTP — ciąg dalszy

Kod statusu	Znaczenie	Opis
502	Zła brama	To dziwny kod statusu, który oznacza, że serwer przesyłający żądanie pełnił rolę serwera pośredniczącego dla innego serwera, ale ten serwer docelowy zwrócił nieprawidłową odpowiedź.
503	Usługa niedostępna	Zazwyczaj oznacza sytuację tymczasową, która sprawia, że serwer chwilowo nie może odpowiedzieć na żądanie.

## Nagłówki

Nagłówki zawierają dodatkowe informacje, które działają jako opisy lub metadane. Jak pokazałem w kolejnych dwóch tabelach, dostępnych jest wiele typów nagłówków: nagłówki ogólne, żądania, odpowiedzi oraz encji (ang. *entity*). Podobnie jak w poprzednim podpunkcie, nie będę tu zamieszczał kompletnej listy wszystkich nagłówków, a jedynie kilka wybranych, które powinieneś znać. W tabeli 8.2 przedstawiłem najważniejsze nagłówki żądania.

Tabela 8.2. Nagłówki żądania

Nagłówek żądania	Opis
User-Agent	Określa przeglądarkę, która przesała żądanie, oraz system operacyjny, w którym działa.
Referrer	Adres URL wyświetlony w przeglądarce w momencie przesyłania tego żądania.
Cookie	Ciasteczka (ang. <i>cookies</i> ) to małe pliki tekstowe zawierające informacje o użytkowniku oraz jego sesji, powiązane z oglądaną witryną. Serwer może umieścić w ciasteczku dowolne informacje, jednak zazwyczaj jest to jakiegoś rodzaju identyfikator sesji, któremu może towarzyszyć jakiś żeton identyfikujący użytkownika.
Content-Type	Określa typy mediów stanowiących zawartość odpowiedzi. Na przykład, w przypadku żądań POST i PUT może to być "application/json", co będzie oznaczać, że żądanie zawiera dane zapisane w formacie JSON.

Z kolei w tabeli 8.3 zamieściłem wybrane nagłówki odpowiedzi.

Tabela 8.3. Nagłówki odpowiedzi

Nagłówek odpowiedzi	Opis
Access-Control-Allow-Origin	Stosowane w technice CORS, by zezwolić stronom z różnych witryn (o różnych adresach URL) na generowanie żądań. Gwiazdka (*) oznacza dowolny adres URL.
Allow	Określa dozwolone czasowniki HTTP.

Oczywiście to bardzo suche i lakoniczne informacje. Jednak znajomość tych kodów oraz przyczyn, które sprawiają, że są one używane w żądaniach i odpowiedziach, pozwala lepiej zrozumieć, jak działa WWW, a w konsekwencji umożliwia pisanie lepszych aplikacji. Kolejnym zagadnieniem, którym się zajmujemy, będzie trasowanie.

## Trasowanie

Trasowanie (ang. *routing*) to w pewnym sensie przekazywanie parametrów do serwera. Kiedy serwer zauważy podaną trasę (ang. *route*), będzie wiedział, że musi odpowiedzieć na żądanie w określony sposób. Odpowiedź może zwracać jakieś określone dane lub zapisywać coś w bazie danych, jednak stosowanie tras pozwala nam określać, w jaki sposób serwer powinien reagować na poszczególne żądania.

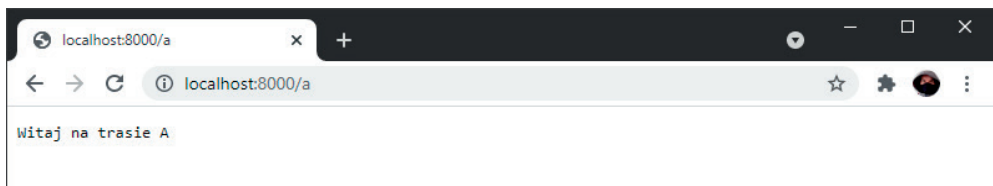
Zobaczmy zatem, w jaki sposób można stosować trasowanie w aplikacjach Node.

1. Zaktualizuj obiekt `server` w pliku `server.mjs` zapisanym w katalogu `node-server` zgodnie z poniższym przykładem:

```
const server = http.createServer((req, res) => {
  if (req.url === "/" ) {
    res.end("Witaj, świecie!");
  } else if (req.url === "/a" ) {
    res.end("Witaj na trasie A");
  } else if (req.url === "/b" ) {
    res.end("Witaj na trasie B");
  } else {
    res.end("Do zobaczenia");
  }
});
```

Jak widać, w tej wersji skryptu używamy wartości pola `req.url`, którą porównujemy z kilkoma określonymi adresami URL. Dla każdego z obsługiwanych adresów zwracamy unikalny łańcuch znaków.

2. Teraz ponownie uruchom serwer i wyświetl w przeglądarce każdą ze zdefiniowanych tras. Na przykład, jeśli wpiszesz w przeglądarce adres URL `http://localhost:8000/a`, to w przeglądarce zostanie wyświetlony tekst przedstawiony na rysunku 8.9.



Rysunek 8.9. Trasa `/a`

3. No dobrze, a teraz przekonajmy się, co się stanie, kiedy odbierzemy żądanie typu POST. Zmodyfikuj wywołanie funkcji `createServer` zgodnie z poniższym przykładem:

```

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    res.end("Witaj, świecie!");
  } else if (req.url === "/a") {
    res.end("Witaj na trasie A");
  } else if (req.url === "/b") {
    res.end("Witaj na trasie B");
  } else if (req.url === "/c" && req.method === "POST") {
    let body = [];
    req.on("data", (chunk) => {
      body.push(chunk);
    });
    req.on("end", () => {
      const params = Buffer.concat(body);
      console.log("treść", params.toString());
      res.end(`Przesłane parametry: ${params.toString()}`);
    });
  } else {
    res.end("Do zobaczenia");
  }
});

```

Jak widać, do kodu dodaliśmy kolejną instrukcję `if else` sprawdzającą, czy została wybrana trasa `/c` i czy użyto metody `POST`. Być może zbędziesz zaskoczony, że w celu pobrania danych przesłanych w żądaniu konieczne jest obsługiwanie zdarzenia `data`, a następnie zdarzenia `end`, które umożliwi zakończenie obsługi żądania i zwrócenie odpowiedzi.

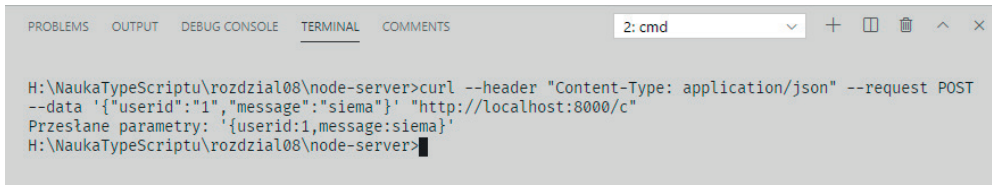
Muszę zatem wyjaśnić, o co w tym chodzi. Otóż Node działa na bardzo niskim poziomie, co oznacza, że starając się zapewnić wysoką wydajność, nie ukrywa wewnętrznych tajników swojego działania, by ułatwić programistom tworzenie aplikacji. A zatem, kiedy przeglądarka wykona żądanie i jakieś informacje zostaną przesłane na serwer, to będą one przesyłane w formie strumienia. Najprościej rzecz ujmując oznacza to, że dane nie są przesyłane za jednym razem w całości, lecz mniejszymi fragmentami. Node nie ukrywa tego faktu przez programistami i zapewnia możliwość pobierania tych fragmentów przy użyciu systemu zdarzeń; nie wiadomo bowiem z góry, ile tych danych będzie. Dopiero kiedy zostaną odebrane wszystkie dane, Node zgłosi zdarzenie `end`.

W tym prostym przykładzie wszystkie dane odbierane w zdarzeniach `data` gromadzimy w tablicy. Kiedy zostanie odebrane zdarzenie `end`, umieszczamy tę tablicę w buforze, dzięki czemu możemy ją przetworzyć jako jedną całość. W naszym przypadku dane będą zawierać kod w formacie JSON, dlatego skonwertujemy je do postaci łańcucha znaków.

4. Aby przetestować działanie tej aplikacji, wygenerujemy żądanie `POST`, używając do tego celu programu narzędziowego `curl`. `curl` to narzędzie obsługiwane z poziomu wiersza poleceń, pozwalające na generowanie żądań HTTP bez korzystania z przeglądarki. Doskonale nadaje się ono do wykonywania testów. A zatem, w oknie terminala wykonaj następujące polecenie (jeśli używasz systemu Windows, to może się okazać, że najpierw będziesz musiał zainstalować `curl`; w systemie MacOS narzędzie to jest dostępne domyślnie):

```
curl --header "Content-Type: application/json" --request POST --data
'{"userid": "1", "message": "siema"}' "http://localhost:8000/c"
```

Wyniki, które powinienem uzyskać, przedstawiłem na rysunku 8.10.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS 2: cmd
H:\NaukaTypeScriptu\rozdzial08\node-server>curl --header "Content-Type: application/json" --request POST
--data '{"userid": "1", "message": "siema"}' "http://localhost:8000/c"
Przesłane parametry: {userid:1,message:siema}
H:\NaukaTypeScriptu\rozdzial08\node-server>
```

**Rysunek 8.10.** Wyniki wykonania żądania POST przy użyciu programu curl

Przedstawione rozwiązanie oczywiście działa, jednak jest dalekie od ideału pod względem łatwości i wydajności implementacji. W końcu na pewno nikt nie chciałby pisać 30 instrukcji `if else` w jednej funkcji `createServer`. Taki kod byłby trudny od analizy oraz do utrzymania. W dalszej części rozdziału, w podrozdziale pt. „Jak Express ułatwia pisanie rozwiązań przeznaczonych dla środowiska Node”, pokażę, w jaki sposób Express eliminuje takie problemy, udostępniając funkcje przesłaniające podstawowe API Node.js, które przyspieszają pisanie aplikacji serwerowych i zwiększają ich niezawodność. Jednak zanim zajmiemy się tymi zagadnieniami, powinienem poznać kilka narzędzi, które ułatwią Ci pisanie aplikacji dla środowiska Node.

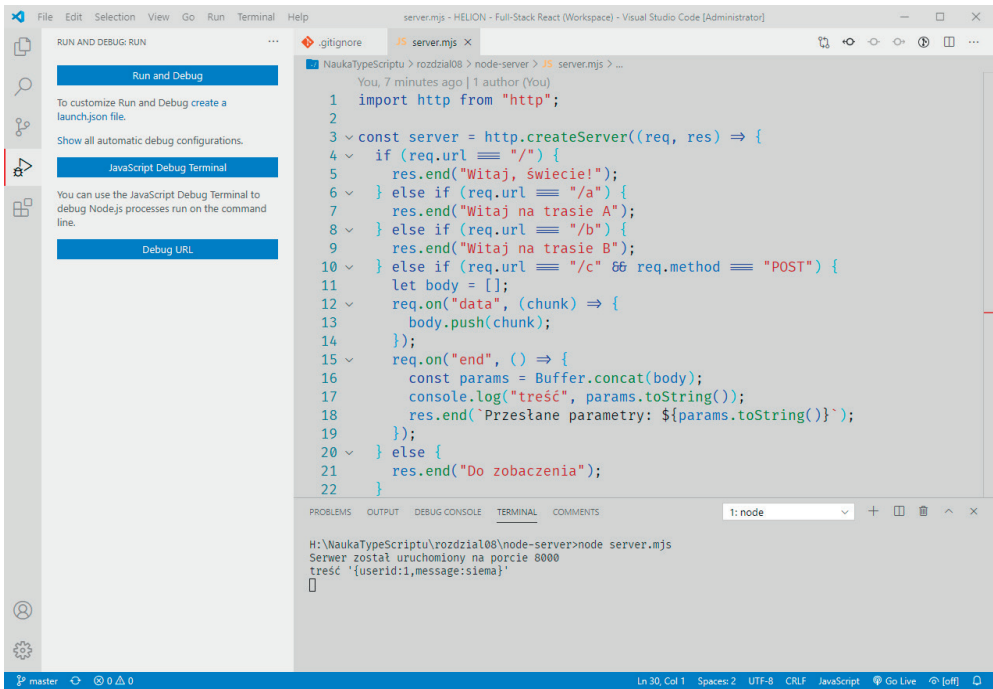
## Debugowanie

Jak już się przekonałeś na przykładzie aplikacji Reacta, debugger jest niezwykle ważnym narzędziem służącym do odnajdywania i poprawiania problemów występujących w kodzie. W przypadku aplikacji Node nie możemy korzystać z narzędzi wbudowanych w przeglądarkę, jednak Visual Studio Code dysponuje wbudowanym debuggerem, który także pozwala na wstrzymywanie działania kodu i przeglądanie wartości zmiennych. Zobaczmy zatem, jak debugować kod w VSCode:

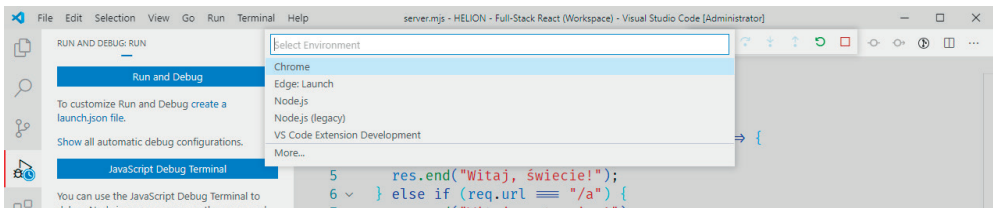
1. W edytorze VSCode kliknij ikonę debugera; postać okna aktualnej wersji programu po wykonaniu tej czynności przedstawiłem na rysunku 8.11.

Pierwszy z dostępnych przycisków pozwala uruchomić debugger, drugi wyświetla panel terminala przeznaczony do debugowania, a trzeci pozwala na debugowanie procesu działającego na zdalnym komputerze.

2. Chcąc skorzystać z debugera VSCode musimy kliknąć przycisk *Run and Debug*. Spowoduje to wyświetlenie listy przedstawionej na rysunku 8.12, z której należy wybrać opcję *Node.js*. W efekcie zostanie rozpoczęta sesja debugowania. Zwróć uwagę na pasek narzędzi wyświetlony w górnej części okna, zawierający przyciski umożliwiające wstrzymanie wykonywania programu (*Pause*), jego kontynuację (*Continue*), ponowne uruchomienie (*Restart*) oraz zatrzymanie (*Stop*). Pamiętaj także, że uruchamianie skryptu w debuggerze VSCode powoduje wykonanie odrębnej kopii Node.js niż ta wykonywana przy użyciu polecenia `npm start`.



Rysunek 8.11. Menu debuggera VSCode

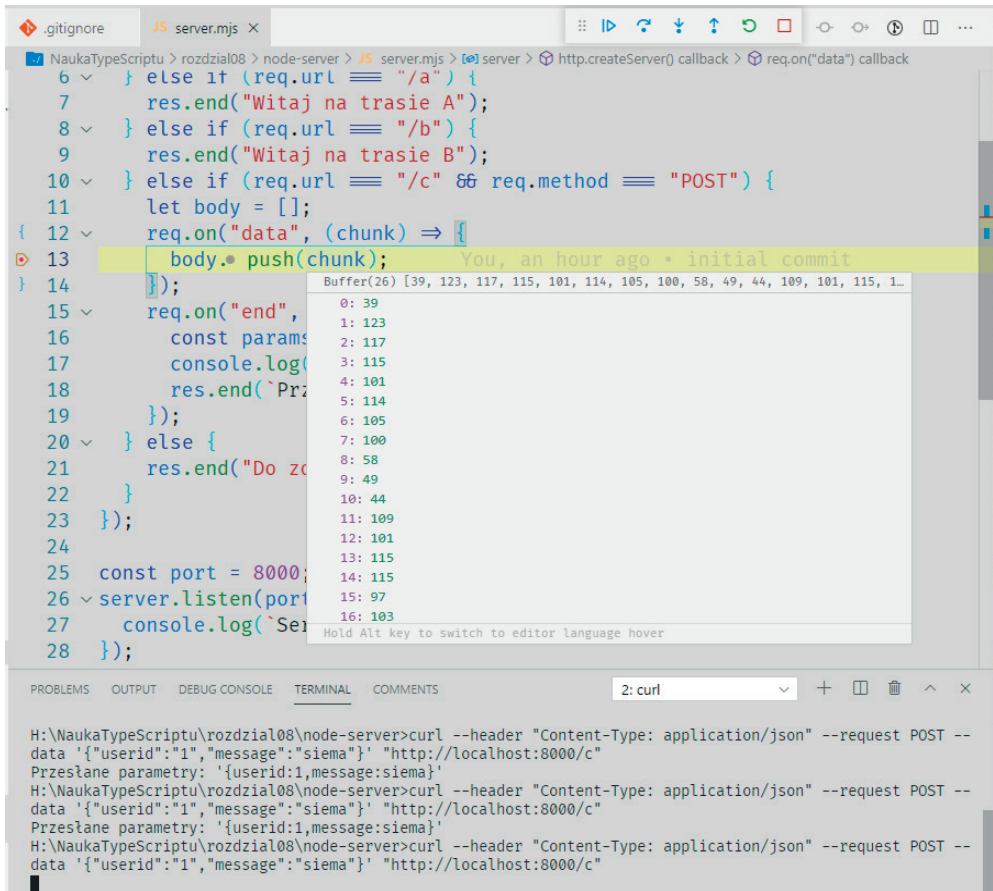


Rysunek 8.12. Wybór debuggera w VSCode

3. Po uruchomieniu debuggera, jeśli ustawisz punkt wstrzymania (klikając na lewo od numeru wiersza), będziesz mógł wstrzymać wykonywanie programu, kiedy dotrze ono do wybranego miejsca kodu. Kiedy to nastąpi, będziesz mógł przeglądać wartości dostępne w bieżącym zasięgu, jak pokazałem na rysunku 8.13.

Jak widać na rysunku, ustawiliśmy punkt wstrzymania w wierszu 13., w kodzie obsługującym zdarzenie data, dzięki czemu możemy przeglądać zawartość odebranego fragmentu danych. Kliknięcie przycisku *Continue* (lub naciśnięcie klawisza *F5*) spowoduje wznowienie wykonywania kodu.

4. Wskazywanie zmiennych podczas debugowania przy użyciu myszki jest przydatną, choć nie jedyną czynnością ułatwiającą debugowanie aplikacji. Aby sprawdzić, jakie są wartości innych zmiennych w momencie zatrzymania wykonywania kodu w punkcie wstrzymania i lepiej zrozumieć, skąd się wzięły, możemy używać także innych paneli VSCode. Przyjrzyj się uważnie oknu VSCode przedstawionemu na rysunku 8.14.

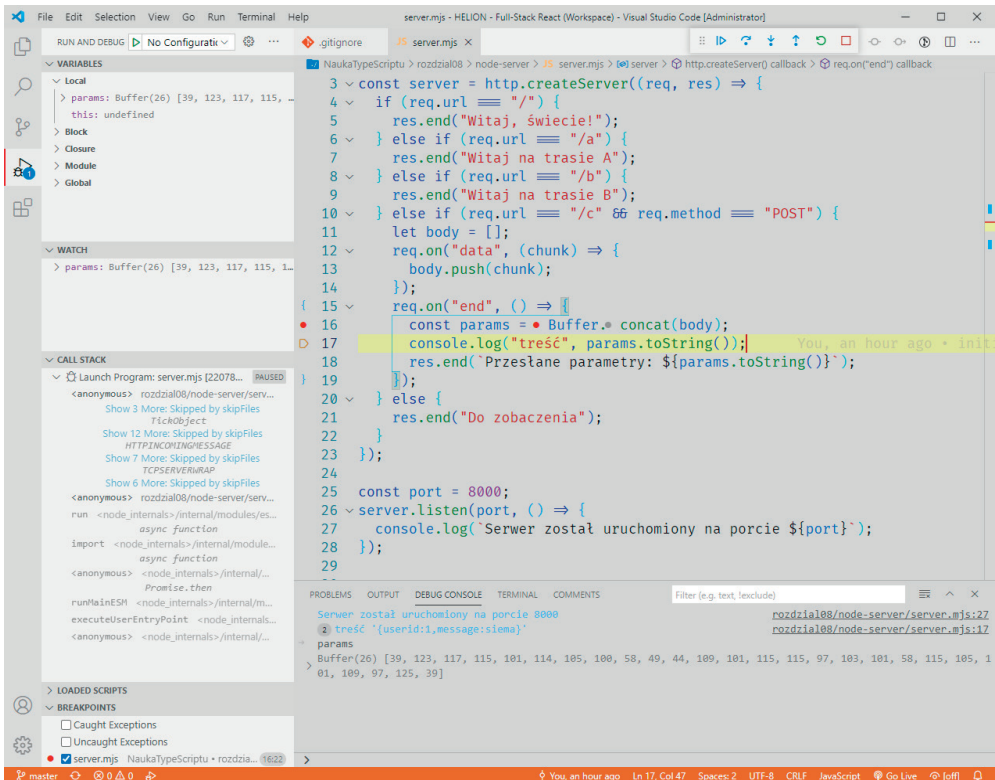


**Rysunek 8.13.** Wstrzymywanie wykonywania programu w określonym wierszu kodu

Spójrz na punkt wstrzymania, widoczny w pobliżu środka okna edytora. Jak widać, wykonywanie kodu zostało wstrzymane w wierszu znajdującym się w zasięgu funkcji obsługującej zdarzenie end. A teraz przyjrzyj się widocznym elementom okna VSCode:

- Zaczynając od lewego, górnego rogu, widzimy panel **VARIABLES** (zmienne), a w nim sekcję *Local* z dwiema zmiennymi istniejącymi w bieżącym zasięgu: `params` oraz `this`. Zwróć uwagę na to, że jest to zasięg lokalny (ang. *Local*), odpowiadający funkcji obsługującej zdarzenie end; to właśnie dlatego są w nim widoczne tylko dwie zmienne.
- Poniżej jest widoczny panel **WATCH** (obserwowane), zawierający tylko jedną zmienną, `params`, którą dodałem do niego wcześniej. Po wskazaniu tego panelu myszką, na jego pasku tytułu pojawia się przycisk „+”, który pozwala dodawać do niego interesujące nas zmienne — kiedy dana zmienna pojawi się w zasięgu, jej bieżąca wartość będzie prezentowana w tym panelu.





**Rysunek 8.14.** Kompletne okno VSCode w trybie debugowania, z widocznymi wszystkimi panelami

- Kolejnym panelem jest *CALL STACK* (stos wywołań). Przedstawia on listę wszystkich aktualnie wykonywanych wywołań funkcji. Poszczególne funkcje są na niej umieszczone w odwrotnej kolejności, czyli na samej górze znajduje się ta, która została wywołana jako ostatnia. Czasami przeważającą większość z prezentowanych tu funkcji będą stanowić funkcje Node lub jakiegoś innego frameworka, które nie zostały napisane przez nas.
- U dołu okna VSCode jest umieszczony panel *DEBUG CONSOLE* (konsola debugowania), prezentująca wszystkie komunikaty (w tym także informacje o błędach) generowane podczas wykonywania skryptu. Zwróć uwagę na to, że na samym dole tego panelu znajduje się wiersz, w którym można wpisać kod, który zostanie wykonany, a jego wyniki wyświetlone powyżej. Na przykład po wpisaniu nazwy zmiennej `params` zostanie wyświetlona jej wartość, jak widać na rysunku 8.14.
- I w końcu, w prawym górnym rogu okna edytora widać pasek narzędzi z przyciskami do sterowania wykonywaniem kodu podczas debugowania. Pierwszym z przycisków, patrząc od lewej, jest przycisk *Continue*, którego kliknięcie spowoduje wznowienie wykonywania kodu. Kolejnym przyciskiem jest *Step Over*; jego kliknięcie spowoduje przejście do następnego wiersza



kodu i zatrzymanie się na nim. Kolejny przycisk, *Step Into*, umożliwia wejście do definicji klasy lub funkcji, jeśli taka znajduje się w wierszu kodu, który właśnie ma zostać wykonany. Kolejny przycisk, *Step Out*, powoduje wyjście do miejsca kodu, z którego został wywołany aktualnie wykonywany blok. I w końcu ostatni, kwadratowy przycisk, *Stop*, umożliwia przerwanie wykonywania programu.

I na tym zakończymy to krótkie wprowadzenie do debugowania w programie VSCode. Z technik, które tu opisałem, będziemy korzystać w dalszej części rozdziału, kiedy zajmiemy się frameworkiem Express, oraz w dalszej części książki, podczas nauki GraphQL-a.

Jak już się zapewne przekonałeś, konieczność ręcznego zatrzymywania i ponownego uruchamiania serwera za każdym razem, kiedy wprowadzimy w jego kodzie jakieś zmiany, jest bardzo uciążliwa i spowalnia pisanie kodu. Dlatego teraz skorzystamy z narzędzia o nazwie nodemon, które pozwoli nam automatycznie restartować serwer Node za każdym razem, kiedy wprowadzimy jakieś zmiany w kodzie.

1. Aby zainstalować nodemon globalnie, wykonaj następujące polecenie:

**npm i nodemon -g**

W ten sposób możemy zainstalować narzędzie nodemon globalnie w całym systemie. Dzięki temu będziemy mogli używać go do uruchamiania dowolnych aplikacji, bez konieczności ciągłego instalowania go w katalogach poszczególnych projektów. Zwróć uwagę na to, że w systemach macOS oraz Linux prawdopodobnie będziesz musiał poprzedzić to polecenie poleceniem `sudo`, które podniesie Twoje uprawnienia, co może być konieczne do zainstalowania oprogramowania globalnie.

2. Teraz chcemy, by narzędzie nodemon było uruchamiane podczas uruchamiania aplikacji. W tym celu zaktualizuj plik *package.json*: dodaj do niego sekcję `scripts`, a w niej pole `"start"` powiążane z poleceniem `"nodemon server.mjs"`:

**nodemon server.mjs**

Po wprowadzeniu tych zmian sekcja `scripts` w pliku *package.json* powinna mieć postać jak na rysunku 8.15.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "nodemon server.mjs"
},
```

**Rysunek 8.15.** Sekcja `scripts` pliku *package.json*

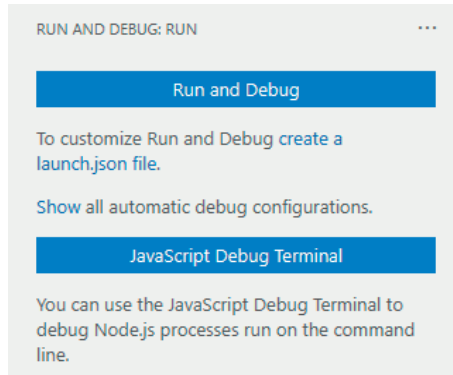
3. Teraz uruchom skrypt, używając następującego polecenia:

**npm start**

Zwróć uwagę na to, że zazwyczaj podczas uruchamiania aplikacji przy użyciu `npm` trzeba zastosować polecenie o postaci `npm run <nazwa pliku>`. Jednak w przypadku poleceń określonych w sekcji `scripts` podpolecenie `run` można pominąć.

Kiedy wykonasz powyższe polecenie, przekonasz się, że aplikacja serwera zostanie uruchomiona jak wcześniej.

4. Skoro nasz serwer już działa, spróbujmy zmienić kod w pliku *server.mjs*, a następnie go zapisać. Zmień komunikat podany w wywołaniu funkcji `listen` na ``Serwer działa na porcie ${port}``. Zaraz po zapisaniu pliku zobaczysz, że Node ponownie uruchomił serwer, a w panelu terminala został wyświetlony nowy komunikat.
5. Zmiany wprowadzane w pliku *package.json* nie mają wpływu na działanie VSCode. A zatem, aby serwer był automatycznie uruchamiany także w VSCode, musimy go odpowiednio skonfigurować. W tym celu ponownie wyświetl menu debugowania (patrz rysunek 8.16) i kliknij odnośnik *create a launch.json file*.



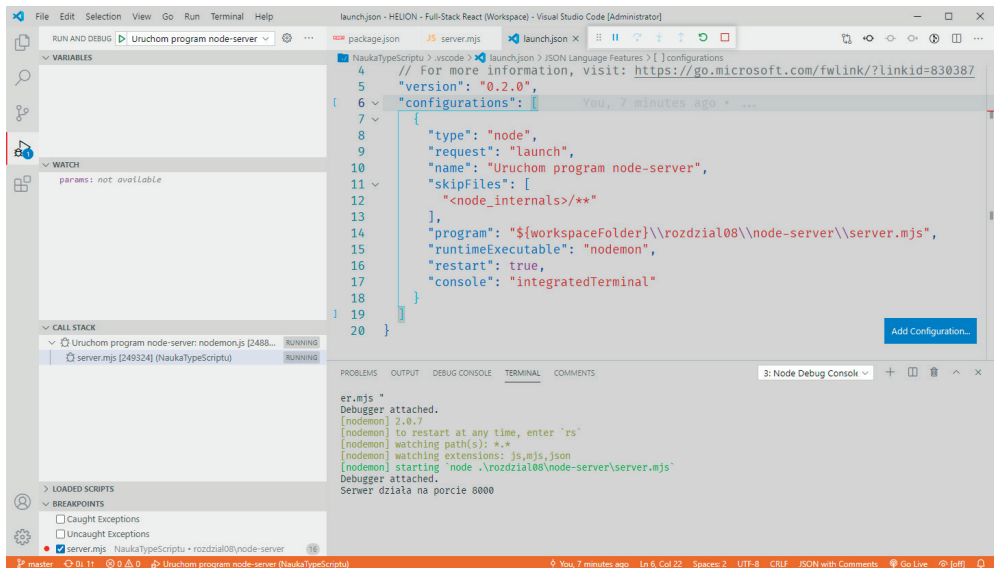
Rysunek 8.16. Tworzenie pliku *launch.json*

Kiedy klikniesz ten odnośnik, w edytorze zostanie wyświetlony plik *launch.json* zapisany w podkatalogu *.vscode* umieszczonym w głównym katalogu repozytorium Git (a nie w głównym katalogu projektu). Plik ten powinien zawierać początkową konfigurację przedstawioną na poniższym przykładzie:

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Uruchom program node-server",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "program": "${workspaceFolder}\\rozdzial08\\node-server\\server.mjs",
      "runtimeExecutable": "nodemon",
      "restart": true,
      "console": "integratedTerminal"
    }
  ]
}
```

Zwróć uwagę na to, że pole `configurations` jest tablicą, a to oznacza, że do tego samego pliku można dodać także inne konfiguracje. Wracając jednak do naszej konfiguracji, zauważ, że jej pole `type` ma wartość `node`, co jest raczej oczywiste. Zmieniliśmy także jej nazwę na `"Uruchom program node-server"`. Jednak koniecznie zwróć uwagę na pole `runtimeExecutable`, w którym zmieniliśmy wartość z `node` na `nodemon`, oraz pole `console`, którego aktualna wartość nakazuje użycie wbudowanego terminala VSCode. Aby móc używać narzędzia `nodemon` i dysponować możliwością debugowania tworzonych aplikacji, musimy używać **terminala**, a nie konsoli debugowania.

6. Kiedy już zmodyfikujesz konfigurację w pliku `launch.json` i uruchomisz ją, to panele do debugowania w oknie VSCode będą wyglądać jak na rysunku 8.17.



Rysunek 8.17. Debugger uruchomiony przy użyciu pliku `launch.json`

Jeśli na liście wyświetlonej na samej górze panelu z lewej strony okna VSCode nie będzie widoczna opcja `Uruchom program node-server`, to wybierz ją, a następnie kliknij przycisk `Start Debugging`. To powinno spowodować uruchomienie debuggera, przy czym tym razem testowa aplikacja `server.mjs` będzie automatycznie restartowana po wprowadzeniu zmian w jej kodzie.

7. Teraz spróbuj wprowadzić jakąś małą zmianę w kodzie pliku `server.mjs`, a debugger go zakończy i ponownie uruchomi. W poniższym przykładzie (patrz rysunek 8.18) zmieniłem dużą literę „S” na małą:

W ten sposób zakończyłem krótką prezentację wybranych narzędzi, które mogą się nam przydać podczas pisania kodu oraz jego debugowania.

```

25  const port = 8000;
26  server.listen(port, () => {
27    console.log(`serwer działa na porcie ${port}`);
28  });
29
30

```

3: Node Debug Console

```

er.mjs "
Debugger attached.
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node .\rozdzial08\node-server\server.mjs`
Debugger attached.
Serwer działa na porcie 8000
[nodemon] restarting due to changes...
[nodemon] starting `node .\rozdzial08\node-server\server.mjs`
Debugger attached.
serwer działa na porcie 8000

```

scriptu) You, seconds ago Ln 27, Col 17 Spaces: 2 UTF-8 CRLF JavaScript

**Rysunek 8.18.** Debugowany program zostanie automatycznie zakończony i ponownie uruchomiony

Z tego podrozdziału nauczyłeś się używać środowiska Node bezpośrednio, do pisania aplikacji serwerów. Poznałeś także sposoby debugowania aplikacji oraz narzędzia, które usprawnią wydajność Twojej pracy jako programisty. Pisanie kodu korzystającego bezpośrednio z możliwości środowiska Node może być czasochłonne i mało intuicyjne, dlatego w następnych podrozdziałach przedstawię framework Express i wyjaśnię, w jaki sposób ułatwia on pisanie aplikacji przeznaczonych dla środowiska Node.

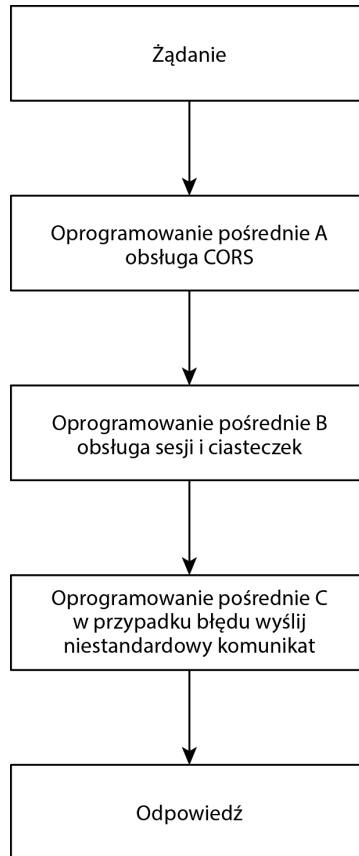
## Jak Express ułatwia pisanie rozwiązań przeznaczonych dla środowiska Node

Jak mogłeś się przekonać we wcześniejszej części rozdziału, pisanie rozwiązań korzystających bezpośrednio z możliwości środowiska Node może być nieco dziwne i niewygodne. Korzystając z jakiegoś łatwiejszego API, niewątpliwie można by poprawić wydajność pracy. I właśnie takie jest przeznaczenie frameworka Express. W tym podrozdziale pokażę Ci, czym jest Express i w jaki sposób ułatwia on pisanie aplikacji przeznaczonych dla środowiska Node.

Express nie jest niezależnym frameworkiem języka JavaScript służącym do pisania aplikacji serwerów. To raczej warstwa kodu przesłaniająca Node i korzystająca z jego API, by ułatwić pisanie serwerów w języku JavaScript i zapewnić im większe możliwości. Express, podobnie jak Node, dysponuje zestawem podstawowych, wbudowanych możliwości, które można

rozbudowywać korzystając z dodatkowych pakietów. Express ma także swoje podstawowe możliwości oraz bogaty ekosystem oprogramowania pośredniego (ang. *middleware*), który je rozszerza.

A zatem, czym jest Express? Zgodnie z informacjami podanymi w jego witrynie, Express jest aplikacją wykonującą sekwencje odwołań do programowania pośredniego. Najlepiej można to wyjaśnić na diagramie (patrz rysunek 8.19).



**Rysunek 8.19.** Tok obsługi żądań i odpowiedzi we frameworku Express

Za każdym razem, kiedy serwer odbierze nowe żądanie, przebywa ono sekwencyjną ścieżkę przetwarzania. Normalnie, gdybyśmy odebrali żądanie, to kiedy zrozumielibyśmy o co w nim chodzi i je przetworzyli, uzyskalibyśmy jakąś odpowiedź. Jednak w przypadku stosowania Expressa można przygotować wiele pośrednich funkcji, które zostaną wstawione (mówi się także: wstrzyknięte) do procesu obsługi żądania i które będą wykonywały pewne unikalne czynności.

W przykładzie przedstawionym na rysunku 8.19 widać, że pierwsze z trzech zastosowanych oprogramowań pośrednich zapewnia obsługę CORS, czyli techniki pozwalającej na obsługę żądań przesyłanych ze strony pobranych z innej domeny URL niż ta, w której

działa serwer. Drugie oprogramowanie pośrednie zapewnia obsługę sesji i ciasteczek. Sesja to unikalne dane dotyczące użytkownika, gromadzone podczas jego aktualnej wizyty w witrynie; przykładem takich danych może być identyfikator użytkownika. I w końcu ostatnim oprogramowaniem pośrednim przedstawionym na rysunku jest procedura obsługi błędów, która będzie określać postać wyświetlanych komunikatów w zależności od zaistniałych problemów. Kluczowe znaczenie ma to, że Express umożliwia wstrzykiwanie tych dodatkowych możliwości, których Node zazwyczaj nie udostępnia, i sprawia, że jest to bardzo łatwe.

Oprócz tych wszystkich mechanizmów związanych z oprogramowaniem pośrednim, Express dodaje także do obiektów Request i Response nowe funkcjonalności, które dodatkowo poprawiają wydajność pracy programistów. Przedstawię je dokładniej w następnym podrozdziale, przy okazji prezentowania możliwości Expressa.

## Przedstawienie możliwości frameworka Express

Express jest w zasadzie narzędziem do wykonywania oprogramowania pośredniego, przeznaczonym dla środowiska Node. Jednak, jak to zazwyczaj w życiu bywa, takie proste wyjaśnienie rzadko kiedy dostarcza wszystkich informacji niezbędnych do prawidłowego stosowania danego produktu. Właśnie dlatego w tym podrozdziale pokrótce opiszę framework Express i na kilku przykładach pokażę jego możliwości.

Zacznijmy od dodania Expressa do naszego projektu *node-server*. W tym celu, w panelu terminala wykonaj następujące polecenie:

```
npm i express -S
```

Po jego wykonaniu plik *package.json* w katalogu *node-server* powinien zawierać sekcje *devDependencies* oraz *dependencies*, o postaci przedstawionej na rysunku 8.20:

```
"author": "",
"license": "ISC",
"devDependencies": {
  "nodemon": "^2.0.4"
},
"dependencies": {
  "express": "^4.17.1"
}
```

Rysunek 8.20. Zaktualizowana zawartość pliku *package.json*

A teraz, jeszcze zanim weźmiemy się za pisanie kodu, musisz zrozumieć kilka rzeczy. Jeszcze raz powtórzę, że Express jest swoistym opakowaniem dla Node. Dlatego w przypadku korzystania z Expressa nie będziemy bezpośrednio wywoływać API Node. Przekonajmy się, jak to wygląda.

1. Utwórz nowy plik serwera, nazwij go *expressapp.mjs* i zapisz w nim poniższy kod:

```
import express from "express";

const app = express();

app.listen({ port: 8000 }, () => {
  console.log("Serwer Node Express został uruchomiony!");
});
```

Jak widać, w kodzie tej aplikacji utworzyliśmy instancję *express*, a następnie użyliśmy jej do wywołania funkcji *listen*. W niewidoczny dla nas sposób funkcja *express.listen* wywołuje funkcje *createServer* oraz *listen* API Node. Kiedy wykonasz ten program, w panelu terminala zostaną wyświetlone komunikaty przedstawione na rysunku 8.21.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS
1: node

H:\NaukaTypeScriptu\rozdzial08\node-server>nodemon expressapp.mjs
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node expressapp.mjs`
Serwer Node Express został uruchomiony!
```

**Rysunek 8.21.** Uruchomienie pliku *expressapp.mjs*

A zatem, dysponujemy już działającym serwerem stworzonym przy użyciu Expressa. Jednak ten serwer nie będzie robił niczego sensownego, dopóki nie dodamy do niego jakiegoś oprogramowania pośredniego. Oprogramowanie pośrednie w Expressie działa w kilku różnych sekcjach. Jest takie oprogramowanie pośrednie, które operuje w obrębie całej aplikacji, jest też takie, które działa tylko podczas wyznaczania i obsługi tras, albo w końcu takie, które jest wykonywane tylko podczas obsługi błędów. Istnieje także oprogramowanie pośrednie, które stanowi integralną część Expressa i jest używane przez niego wewnętrznie. No i oczywiście, zamiast implementować własny kod, który będzie działać jako oprogramowanie pośrednie, możemy korzystać z pakietów npm zawierających oprogramowanie pośrednie przygotowane przez innych twórców. Kilka przykładów takiego oprogramowania pośredniego przedstawiłem już na rysunku 8.19., zamieszczonym w poprzednim podrozdziale, pt. „Jak Express ułatwia pisanie rozwiązań przeznaczonych dla środowiska Node”.

2. Zacznijmy od dodania naszego własnego oprogramowania pośredniego. Zaktualizuj kod pliku *expressapp.mjs* jak pokazałem na poniższym przykładzie:

```
import express from "express";

const app = express();

app.use((req, res, next) => {
  console.log("Pierwsze oprogramowanie pośrednie.");
  next();
});
```

```

app.use((req, res, next) => {
  res.send("Witaj, świecie! Jestem niestandardowym oprogramowaniem
    ↳pośrednim.");
});

app.listen({ port: 8000 }, () => {
  console.log("Serwer Node Express został uruchomiony!");
});

```

W tym przykładzie zdecydowałem się zastosować oprogramowanie pośrednie działające w całej aplikacji. Do tworzenia takiego rodzaju oprogramowania pośredniego używana jest funkcja `use` obiektu `app`. Zastosowanie takiego rozwiązania sprawia, że wszelkie żądania trafiające do aplikacji, niezależnie od określonej w nich trasy, będą przetwarzane przez takie oprogramowanie pośrednie.

Przeanalizujmy dokładniej te dwie nowe funkcje dodane do pliku *expressapp.mjs*. Przede wszystkim pamiętaj, że oprogramowanie pośrednie jest przetwarzane w kolejności, w jakiej zostało zadeklarowane w kodzie. Oprócz tego, z wyjątkiem ostatniego oprogramowania pośredniego, które kończy obsługę żądania, w każdym innym oprogramowaniu pośrednim musi zostać wywołana funkcja `next`; w przeciwnym razie proces przetwarzania żądania zostanie przerwany.

Pierwsze oprogramowanie pośrednie dodane w powyższym przykładzie ogranicza się do wyświetlenia podanego tekstu na konsoli; z kolei drugie wyświetli tekst w oknie przeglądarki, używając do tego celu funkcji `send` Expressa. Ta funkcja `send` w dużym stopniu przypomina funkcję `end` Node, gdyż podobnie jak ona kończy przetwarzanie żądania, jednak dodatkowo przesyła do przeglądarki nagłówek `Content-Type` o wartości `text/html`. Gdybyśmy korzystali wyłącznie z możliwości Node, musielibyśmy samodzielnie wysłać ten nagłówek.

3. A teraz dodajmy do aplikacji oprogramowanie pośrednie, które będzie wykonywane wyłącznie dla określonych tras. Zwróć uwagę na to, że — z technicznego punktu widzenia — istnieje możliwość przekazania trasy, takiej jak `/routea`, w wywołaniu funkcji `use`. Jednak lepszym rozwiązaniem będzie użycie obiektu routera, który pozwala gromadzić wszystkie stosowane trasy w jednym miejscu. W Expressie router także jest oprogramowaniem pośrednim. Przeanalizujmy poniższy przykład:

```

import express from "express";

const router = express.Router();

```

W pierwszej kolejności utworzyliśmy obiekt routera, czyli obiekt typu `express.Router`, i zapisaliśmy go w stałej o nazwie `router`.

```

const app = express();

app.use((req, res, next) => {
  console.log("Pierwsze oprogramowanie pośrednie.");
  next();
});

app.use((req, res, next) => {

```



```

    res.send("Witaj, świecie! Jestem niestandardowym oprogramowaniem
    ↳pośrednim.");
  });
  app.use(router);

```

Innymi słowy, dysponujemy tym samym co wcześniej zestawem oprogramowania pośredniego dodanego do obiektu `app`; oznacza to, że oprogramowanie to będzie używane podczas obsługi wszystkich tras. Jednak oprócz tego dodaliśmy do obiektu `app` jeszcze jedno oprogramowanie pośrednie — `router`. To oprogramowanie pośrednie będzie wykonywane wyłącznie dla konkretnych tras, zdefiniowanych w poniższym fragmencie kodu:

```

router.get("/a", (req, res, next) => {
  res.send("Cześć! Witaj na trasie 'a'.");
});
router.post("/c", (req, res, next) => {
  res.send("Cześć! Witaj na trasie 'c'.");
});

```

W tym kodzie dodaliśmy do obiektu `router` dwie trasy: trasę `/a`, która używa funkcji `get`, oraz trasę `/c`, która używa funkcji `post`. Obie te funkcje, `get` oraz `post`, określają czasownik HTTP, czyli typ żądania, które dana trasa będzie obsługiwać. Wywołanie funkcji `listen` umieszczone na końcu kodu jest takie samo jak w poprzednim przykładzie.

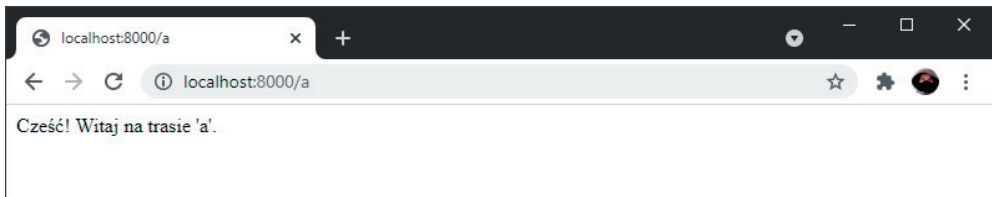
```

app.listen({ port: 8000 }, () => {
  console.log("Serwer Node Express został uruchomiony!");
});

```

A teraz, jeśli wpiszesz w przeglądarce adres `http://localhost:8000/a`, stanie się coś dziwnego. Zamiast oczekiwanego tekstu **Cześć! Witaj na trasie 'a'.**, zostanie wyświetlony tekst **Witaj, świecie! Jestem niestandardowym oprogramowaniem pośrednim.** Czy domyślasz się, dlaczego tak się dzieje? Otóż jest to konsekwencją faktu, że oprogramowanie pośrednie jest przetwarzane w kolejności, w jakiej zostało dodane, a ponieważ drugie w kolejności oprogramowanie pośrednie wywołuje funkcję `res.send`, to wywołanie to kończy przetwarzanie żądania i kolejne oprogramowanie pośrednie nie jest już wykonywane.

Usuń zatem to drugie oprogramowanie pośrednie (wyświetlające komunikat "Witaj, świecie!") i ponownie wpisz w przeglądarce adres `http://localhost:8000/a`. Tym razem zobaczysz stronę przedstawioną na rysunku 8.22.



Rysunek 8.22. Oprogramowanie pośrednie trasy `/a`

Świetnie, to zadziałało. A teraz spróbuj wyświetlić w przeglądarce stronę `http://localhost:8000/c`. Czy ta trasa też zadziałała? Okazuje się, że nie! Zamiast oczekiwanego tekstu, w przeglądarce jest wyświetlany komunikat `Cannot get /c`<sup>1</sup>. Jak się zapewne domyśliłeś, żądania generowane przez przeglądarkę są domyślnie przesyłane przy użyciu metody HTTP GET, a nasza trasa `/c` obsługuje jedynie żądania przesyłane metodą POST. O tym, że trasa działa, możesz przekonać się, jeśli otworzysz panel terminala i wykonasz polecenie `curl` generujące żądanie POST, podobne do tego przedstawionego w podrozdziale pt. „Prezentacja możliwości środowiska Node”. Efekty wykonania tego polecenia przedstawiłem na rysunku 8.23.

```

H:\NaukaTypeScriptu>curl --header "Content-Type: application/json" --request POST --data '{"userid":"1","message":"siema"}' 'http://localhost:8000/c'
Cześć! Witaj na trasie 'c'.
H:\NaukaTypeScriptu>

```

Rysunek 8.23. Odwołanie do trasy `/c`

Jak widać, wygenerowany został odpowiedni tekst.

4. A teraz dodajmy do naszego serwera oprogramowanie pośrednie przygotowane przez innych twórców. W podrozdziale pt. „Prezentacja możliwości środowiska Node” pokazałem, w jaki sposób można przetwarzać dane przesłane w żądaniu POST oraz jak żmudnym jest to zadaniem w przypadku korzystania jedynie z możliwości Node. W tym przykładzie wykorzystamy oprogramowanie pośrednie o nazwie `body-parser`, które znacząco ułatwia przetwarzanie danych przesyłanych metodą POST. Zaktualizuj kod programu jak pokazałem na poniższym przykładzie:

```
import express from "express";
import bodyParser from "body-parser";
```

W pierwszej kolejności importujemy oprogramowanie pośrednie `body-parser`. Następnie dodajemy je do aplikacji Expressa, dzięki czemu wszystkie używane funkcje obsługi tras będą mogły, w razie potrzeby, korzystać ze przetworzonych obiektów, a nie z łańcuchów z kodem w formacie JSON:

```
const router = express.Router();
const app = express();
```

```
app.use(bodyParser.json());
```

Następnie aktualizujemy funkcję obsługi trasy `/c` tak, by wyświetlany przez nią komunikat zawierał wartość przekazaną w polu `message`:

```
app.use((req, res, next) => {
  console.log("Pierwsze oprogramowanie pośrednie.");
  next();
});
```

<sup>1</sup> Nie można pobrać `/c` — *przyp. tłum.*

```

app.use(router);
router.get("/a", (req, res, next) => {
  res.send("Cześć! Witaj na trasie 'a'.");
});
router.post("/c", (req, res, next) => {
  res.send(`Cześć! Witaj na trasie 'c'. Mam wiadomość:
  ${req.body.message}.`);
});

```

Jak widać, to rozwiązanie jest znacznie łatwiejsze niż korzystanie z rozwiązań dostarczanych przez API Node, takich jak zdarzenia `data` i `end`.

5. Teraz zajmiemy się oprogramowaniem pośrednim do obsługi błędów. W tym celu zmodyfikuj kod poniżej wywołania `bodyParser.json()` tak, jak pokazałem na poniższym przykładzie:

```

import express from "express";
import bodyParser from "body-parser";

const router = express.Router();
const app = express();

app.use(bodyParser.json());

app.use((req, res, next) => {
  console.log("Pierwsze oprogramowanie pośrednie.");
  throw new Error("Przykro nam - awaria!");
});

```

W tym fragmencie w naszym pierwszym zdefiniowanym oprogramowaniu pośrednim zgłaszamy błąd.

```

app.use(router);

router.get("/a", (req, res, next) => {
  res.send("Cześć! Witaj na trasie 'a'.");
});
router.post("/c", (req, res, next) => {
  res.send(`Cześć! Witaj na trasie 'c'. Mam wiadomość:
  ${req.body.message}.`);
});

app.use((err, req, res, next) => {
  res.status(500).send(err.message);
});

```

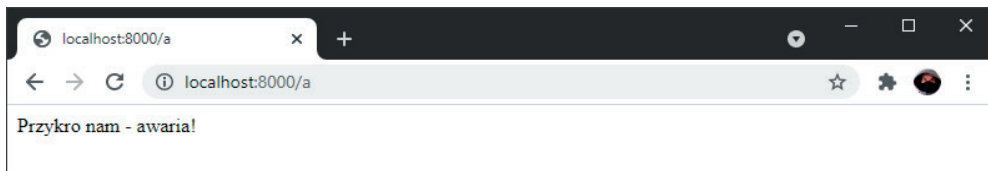
Następnie, jako ostatnie oprogramowanie pośrednie, dodajemy funkcję obsługi błędów. To oprogramowanie pośrednie będzie przechwytywać wszystkie błędy, które nie zostały obsłużone wcześniej, i przysyłać do przeglądarki odpowiedni kod statusu oraz komunikat.

```

app.listen({ port: 8000 }, () => {
  console.log("Serwer Node Express został uruchomiony!");
});

```

6. Jeśli teraz wpiszesz w przeglądarce adres `http://localhost/a`, zostanie w niej wyświetlona strona przedstawiona na rysunku 8.24.



Rysunek 8.24. Komunikat o błędzie w aplikacji

Zwróć uwagę na to, że błąd jest zgłaszany w pierwszym zastosowanym oprogramowaniu pośrednim, a to oznacza, że będzie on występował w każdej ze zdefiniowanych tras, po czym będzie przechwytywany przez oprogramowanie pośrednie obsługujące błędy.

To było krótkie przedstawienie frameworka Express oraz jego możliwości. Jak mogłeś się przekonać, Express w znacznym stopniu upraszcza tworzenie aplikacji internetowych przeznaczonych dla środowiska Node i poprawia czytelność ich kodu. W następnym podrozdziale zastosujemy środowisko Node i framework Express do stworzenia typowego internetowego API, czyli takiego, które zwraca dane zapisane w formacie JSON.

## Tworzenie internetowego API przy użyciu Expressa

W tym podrozdziale dowiesz się, jak tworzyć internetowy API. Aktualnie takie API stanowią najpopularniejszy sposób udostępniania danych w internecie. W naszej końcowej aplikacji nie będziemy jednak korzystać z takiego API, gdyż naszym zamiarem w tej książce jest użycie do tego celu GraphQL-a. Nie zmienia to jednak faktu, że warto dysponować ogólną wiedzą na temat zasad działania i sposobów tworzenia takich API, gdyż są one powszechnie stosowane, a nawet GraphQL z nich korzysta, choć w sposób dla nas niewidoczny.

A zatem, czym jest internetowy API? **API** to skrót od angielskich słów ***Application Programming Interface***, oznaczających **interfejs programowania aplikacji**. Innymi słowy, chodzi o sposób, którego jeden system komputerowy może używać do prowadzenia interakcji z innym systemem. A zatem, internetowy API to API korzystający z technologii internetowych do dostarczania interfejsu programowania aplikacji, który może być wykorzystywany przez inne systemy.

Wszystkie internetowe API mają punkt końcowy reprezentowany przez identyfikator URI, który w zasadzie jest tym samym, co adres URL. Ta ścieżka musi być statyczna i nie może się zmieniać. Jeśli taka zmiana jest pożądana, to dostawca API zaktualizuje jego wersję, pozostawiając dotychczasowy URI w niezmienionej postaci, i doda nowy URI opisany w nowej wersji API. Na przykład, jeśli początkowo URI rozpoczynał się od `/api/v1/users`, to w kolejnej wersji API będzie się on rozpoczynał od `/api/v2/users`.

Przygotujmy zatem prosty API, który posłuży nam jako demonstracja sposobów tworzenia takich rozwiązań.

1. Zmodyfikuj plik *expressapp.mjs*, dodając do niego trasy przedstawione w poniższym przykładzie:

```
import express from "express";
import bodyParser from "body-parser";

const router = express.Router();
const app = express();

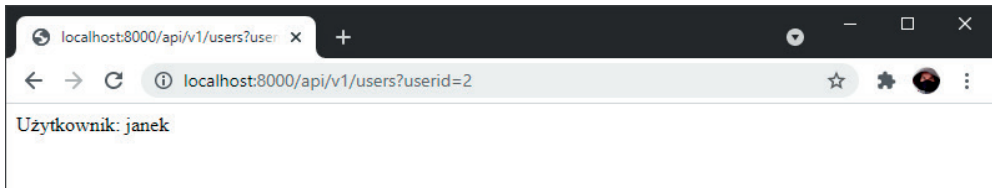
app.use(bodyParser.json());

app.use((req, res, next) => {
  console.log("Pierwsze oprogramowanie pośrednie.");
  next();
});
```

W tym fragmencie kodu jedyną wprowadzoną zmianą było usunięcie instrukcji zgłaszającej błąd.

```
app.use(router);
router.get("/api/v1/users", (req, res, next) => {
  const users = [
    {
      id: 1,
      username: "tomek",
    },
    {
      id: 2,
      username: "janek",
    },
    {
      id: 3,
      username: "lidia",
    },
  ];
  console.log(req.query.userid);
  const user = users.find((usr) => usr.id == req.query.userid);
  res.send(`Użytkownik: ${user?.username}`);
});
```

To pierwsze zastosowane oprogramowanie pośrednie definiuje ścieżkę `/api/v1/users`. Stosowanie ścieżek o takiej postaci jest standardowym rozwiązaniem w internetowych API. Ścieżka zawiera informację o numerze wersji oraz określa kontener danych, który chcemy przeszukać — w tym przykładzie są to użytkownicy (`users`). Do celów demonstracyjnych używamy podanej na stałe tablicy użytkowników, z której będziemy wybierać element o pasującym identyfikatorze. Ponieważ pole `id` jest liczbą, natomiast dane odczytywane z obiektu `req.query` są łańcuchami znaków, dlatego zamiast operatora `===` używamy `==`. Przykład działania tego API przedstawiłem na rysunku 8.25.



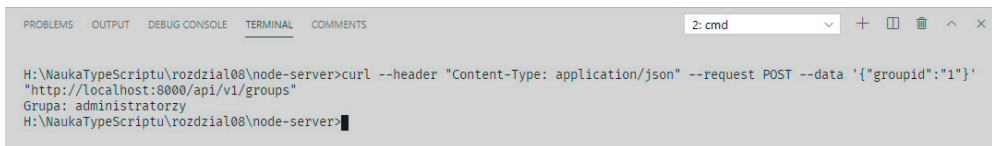
**Rysunek 8.25.** Żądanie GET pobierające użytkownika

Jak widać, został zwrócony drugi użytkownik: jank.

2. W bardzo podobny sposób przygotujemy drugie oprogramowanie pośrednie, które będzie operować na grupach. Zwróć uwagę na to, że postać ścieżki jest spójna ze ścieżką zapewniającą dostęp do poprzedniego zasobu — to bardzo ważna cecha internetowych API. Także w tym przypadku pobieramy jeden element z tablicy, przy czym tym razem będziemy przysyłać żądania metodą POST, więc parametr będziemy musieli odczytywać z właściwości body obiektu żądania:

```
router.post("/api/v1/groups", (req, res, next) => {
  const groups = [
    {
      id: 1,
      groupname: "administratorzy",
    },
    {
      id: 2,
      groupname: "uzytkownicy",
    },
    {
      id: 3,
      groupname: "pracownicy",
    },
  ];
  const group = groups.find((grp) => grp.id == req.body.groupid);
  res.send(`Grupa: ${group.groupname}`);
});
```

Jeśli teraz, w panelu terminala, wykonasz żądanie POST skierowane na ten adres URI, to uzyskasz wyniki przedstawione na rysunku 8.26.



**Rysunek 8.26.** Żądanie POST pobierające grupę

Jak widać na rysunku, została zwrócona pierwsza grupa: administratorzy.

W pozostałej części kodu nie wprowadziliśmy żadnych zmian.

```
app.use((err, req, res, next) => {
  res.status(500).send(err.message);
});
```

```
app.listen({ port: 8000 }, () => {  
  console.log("Serwer Node Express został uruchomiony!");  
});
```

Ponieważ API internetowe korzystają z technologii internetowych, obsługują wywołania korzystające ze wszystkich dostępnych metod protokołu HTTP: GET, POST, PATCH, PUT oraz DELETE.

To było krótkie wprowadzenie do tworzenia internetowych API przy użyciu Node i Expressa. Teraz dysponujesz już szeroką wiedzą dotyczącą możliwości i stosowania Node.js oraz jednego z najważniejszych frameworków przeznaczonych dla tego środowiska, jakim jest Express.

## Podsumowanie

W tym rozdziale przedstawiłem środowisko Node oraz framework Express. Node jest jedną najważniejszych technologii serwerowych, używaną na przeważającej większości serwerów tworzących internet, natomiast Express jest najpopularniejszym i często stosowanym frameworkiem do tworzenia aplikacji internetowych. Obecnie dysponujesz już kompletnymi informacjami dotyczącymi sposobu działania zarówno klienckich, jak i serwerowych części aplikacji internetowych.

W następnym rozdziale zaczniesz poznawać GraphQL — niezwykle popularny i stosunkowo nowy standard tworzenia API usług internetowych. Kiedy już go poznasz, będziesz dysponował kompletną wiedzą niezbędną do rozpoczęcia tworzenia własnych projektów.

# Czym jest GraphQL?

Z tego rozdziału dowiesz się, czym jest GraphQL — jedna z najpopularniejszych aktualnie używanych technologii internetowych. Wiele dużych firm zaczęło używać GraphQL-a do tworzenia swoich API; należą do nich tacy giganci jak Facebook, Twitter, New York Times, czy też GitHub. Z tego rozdziału dowiesz się, co sprawia, że GraphQL zyskał tak wielką popularność, jak on działa oraz w jaki sposób można wykorzystywać jego możliwości.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- wyjaśnieniem, czym jest GraphQL;
- objaśnieniem, czym są schematy GraphQL;
- przedstawieniem definicji typów i resolverów;
- wyjaśnieniem, czym są zapytania, mutacje oraz subskrypcje.

---

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś dysponować podstawową wiedzą z zakresu tworzenia rozwiązań internetowych przy użyciu Node. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code** (**VSCode**)

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full->



*Stack-React-TypeScript-and-Node*. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial09* (*Chap9*).

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog o nazwie *rozdzial09*.

## Czym jest GraphQL?

Z tego podrozdziału dowiesz się, czym jest GraphQL, dlaczego go stworzono oraz jakie problemy stara się on rozwiązywać. Zrozumienie powodów stworzenia GraphQL-a jest bardzo ważne, gdyż umożliwia projektowanie lepszych internetowych API.

A zatem, czym jest GraphQL? Na poniższej liście przedstawiłem kilka jego głównych cech:

- **GraphQL jest standardem do tworzenia schematów danych opracowanym przez firmę Facebook.**

GraphQL udostępnia standardowy język służący do definiowania danych, typów danych oraz powiązanych z nimi zapytań. GraphQL można sobie ogólnie wyobrazić jako odpowiednik interfejsu udostępniającego pewien kontrakt. Ten kontrakt nie zawiera co prawda żadnego kodu, jednak pokazuje, jakie są dostępne typy danych oraz zapytania.

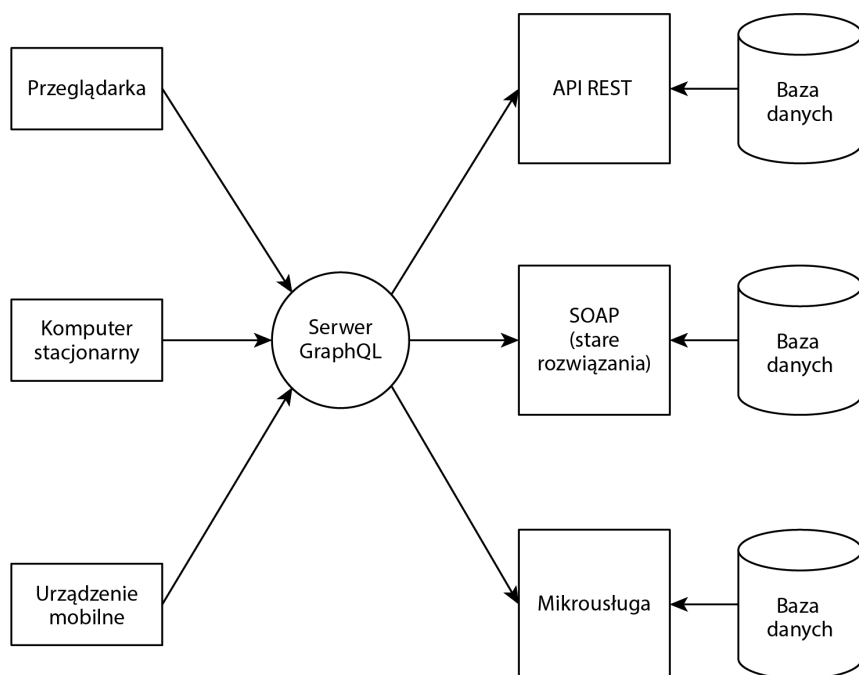
- **GraphQL jest niezależny od platformy, frameworków oraz języków programowania.**

Kiedy tworzymy API używając do tego GraphQL-a, będziemy posługiwali się dokładnie tym samym językiem do opisywania danych, ich typów oraz zapytań, niezależnie od tego, jakiego języka programowania faktycznie używamy na co dzień lub na jakiej platformie systemowej będzie działać nasza aplikacja. Możliwość korzystania ze spójnej i niezawodnej reprezentacji danych, która działa na szerokiej gamie systemów i platform, jest oczywiście bardzo korzystna z punktu widzenia klientów. Jednak te cechy są równie przydatne dla programistów, gdyż bez przeszkód będziemy mogli używać naszego ulubionego języka programowania i preferowanego frameworka.

- **GraphQL przekazuje kontrolę nad pobieranymi danymi do kodu wywołującego.**

W przypadku standardowych usług internetowych, to serwer decyduje o tym, jakie pola danych będą przekazywane do klienta. Z kolei w API GraphQL-a to klient decyduje, jakie pola danych chce uzyskać. Takie rozwiązanie zapewnia klientom lepszą kontrolę nad danymi, zmniejsza zajętość pasma przenoszenia i redukuje koszty.

Ogólnie rzecz ujmując, punkty końcowe GraphQL mają dwa podstawowe zastosowania. Pierwszym z nich jest stosowanie ich jako swoistych bram do konsolidacji innych usług dostępu do danych, a drugim używanie ich jako głównej usługi internetowego API, która odbiera dane bezpośrednio z magazynu i udostępnia je klientom. Na rysunku 9.1 przedstawiłem schemat zastosowania GraphQL-a jako bramy zapewniającej dostęp do innych danych.



**Rysunek 9.1.** GraphQL działający jako brama

Jak widać, GraphQL działa jako pojedyncze, centralne miejsce, do którego odwołują się wszystkie klienty. GraphQL doskonale wpisuje się w takich rozwiązaniach, dzięki wykorzystaniu standaryzowanego języka obsługiwane na wielu platformach systemowych.

W przypadku naszej przykładowej aplikacji, którą przygotujemy w dalszej części książki, będziemy używali GraphQL-a jako kompletnego internetowego API; nic jednak nie stoi na przeszkodzie, by łączyć z nim już istniejące usługi internetowe, tak by GraphQL obsługiwał tylko wybrane z odwołań do naszych usług. To oznacza, że by korzystać z GraphQL-a, wcale nie trzeba pisać od nowa całej aplikacji. GraphQL można wprowadzać do niej stopniowo, po przemyśleniu, gdzie jest sens go stosować i bez przeszkadzania w działaniu usług, których aplikacja już używa.

W tym podrozdziale przedstawiłem GraphQL na dość ogólnym, koncepcyjnym poziomie. GraphQL korzysta z własnego języka do opisu danych, co oznacza, że można go stosować niezależnie od frameworka używanego w aplikacji, języka programowania, w którym tę aplikację piszemy, oraz systemu operacyjnego. Ta elastyczność sprawia, że GraphQL może być potężnym sposobem współdzielenia danych w obrębie organizacji, a nawet w internecie. W następnym podrozdziale przedstawię język schematów GraphQL i pokażę, jak można go stosować. Ta wiedza ułatwi nam określenie struktury naszego modelu danych i pozwoli Ci zrozumieć, jak należy skonfigurować serwer GraphQL.

# Schematy GraphQL

Jak już wspominałem, GraphQL jest językiem używanym do określania struktury i typów informacji, których używamy. Niezależnie od producenta implementacji GraphQL-a używanej na serwerze, klienci mogą oczekiwać, że zawsze będą zwracane te same struktury danych. Ta możliwość ukrywania przed klientami szczegółów implementacyjnych serwera jest jedną z najważniejszych zalet GraphQL-a.

Utwórzmy zatem prosty schemat GraphQL i zobaczmy, jak wygląda:

1. W katalogu *rozdzial09* utwórz podkatalog *graphql-schema*.
2. W panelu terminala przejdź do nowego katalogu *graphql-schema*, wykonaj poniższe polecenie, a następnie zaakceptuj wszelkie proponowane wartości domyślne:

```
npm init
```

3. Teraz zainstaluj poniższe pakiety:

```
npm i express apollo-server-express graphql @types/express
```

4. Następnie zainicjuj język TypeScript, wykonując poniższe polecenie:

```
tsc -init
```

Zwróć uwagę na to, że konfiguracja zapisana w pliku *tsconfig.json* zakłada stosowanie ścisłej kontroli typów (`"strict": true`).

5. Teraz utwórz plik TypeScript o nazwie *typeDefs.ts* i zapisz w nim poniższy kod:

```
import { gql } from "apollo-server-express";
```

Ta instrukcja importuje obiekt `gql`, pozwalający na formatowanie składni i kolorowanie syntaktyczne kodu pisanego w języku schematów GraphQL:

```
const typeDefs = gql`
  type User {
    id: ID!
    username: String!
    email: String
  }

  type Todo {
    id: ID!
    title: String!
    description: String
  }

  type Query {
    getUser(id: ID!): User
    getTodos: [Todo!]
  }
`;
```

Język stosowany przez GraphQL-a jest stosunkowo prosty i wygląda dość podobnie do TypeScriptu. Przyjrzyjmy się temu, zaczynając od początku tego skryptu. Najpierw definiowana jest encja `User`, co sygnalizuje słowo kluczowe

type. W GraphQL-u słowo kluczowe type oznacza zadeklarowanie obiektu o pewnej strukturze. Jak widać, typ User składa się z kilku pól. Pierwsze z nich, id, jest typu ID!. ID jest wbudowanym typem oznaczającym unikalne wartości — czymś w stylu pewnego rodzaju globalnie unikalnych identyfikatorów GUID. Znak wykrzyknika umieszczony na końcu typu pola oznacza, że jego wartością nie może być null. Kolejne pole, username, jest typu String!, co także oznacza, że wartością tego pola nie może być null, tylko łańcuch znaków. Ostatnim polem typu User jest email, które jest typu String, a w jego deklaracji nie użyliśmy wykrzyknika, co oznacza, że w tym polu można zapisywać wartości null.

Typ Todo ma podobne pola, ciekawszy jest natomiast typ Query. Pokazuje on, że w GraphQL-u nawet zapytania są typami. Jeśli przyjrzysz się tym dwóm zapytaniom, getUser oraz getTodos, od razu zauważysz, dlaczego wcześniej utworzyliśmy typy User i Todo: są to typy wyników zwracanych przez dwie metody typu Query. Oprócz tego zwróć uwagę na to, że funkcja getTodos zwraca tablicę obiektów Todo, której wartości nie mogą wynosić null, co wyraźnie zaznacza użycie nawiasów kwadratowych i znaku wykrzyknika. Na samym końcu pliku eksportujemy definicje typów, używając do tego celu zmiennej typeDefs:

```
export default typeDefs;
```

Definicje typów są używane przez serwer GraphQL Apollo do opisywania typów schematów, podanych w pliku schematów. Zanim serwer będzie mógł zacząć udostępniać dane GraphQL, musi najpierw dysponować kompletnym plikiem schematu zawierającym *wszystkie* typy danych używane przez aplikację, ich pola oraz zapytania, które będą udostępniane za pośrednictwem API.

Ponadto warto zwrócić uwagę na to, że GraphQL dysponuje kilkoma domyślnymi typami skalarnymi, które są wbudowane w język. Typami tymi są: Int, Float, String, Boolean oraz ID. Jak widać na przedstawionym wcześniej przykładzie, nie musieliśmy tworzyć deklaracji dla tych typów.

W tym podrozdziale pokazałem, jak wygląda przykładowy schemat danych GraphQL. Tej składni będziemy używali w dalszej części książki, podczas tworzenia API naszej aplikacji. W następnym podrozdziale nieco dokładniej opiszę język GraphQL i pokażę, czym są resolwery.

## Definicje typów i resolwery

W tym podrozdziale będziemy dalej poznawać schematy GraphQL-a, a oprócz tego zaimplementujemy resolwery, czyli funkcje, które wykonują faktyczną pracę. Przedstawię tu także serwer Apollo GraphQL i wyjaśnię, jak można go uruchomić.

Zacznijmy od wyjaśnienia tego, czym są resolwery. Otóż resolwery (ang. *resolvers*) są funkcjami, które pobierają lub modyfikują dane przechowywane w magazynie. Te dane są następnie dopasowywane do definicji typów GraphQL-a.

Aby dokładniej pokazać, na czym polega rola resolverów, musimy kontynuować prace na poprzednim przykładem. Wykonaj czynności opisane na poniższej liście:

1. Zainstaluj pakiet UUID. UUID to narzędzie, które pozwala generować unikalne identyfikatory dla typów ID:

```
npm i uuid @types/uuid
```

2. Utwórz plik *server.ts* i zapisz w nim następujący fragment kodu. Ten plik będzie uruchamiał nasz serwer GraphQL:

```
import express from "express";
import { ApolloServer, makeExecutableSchema } from "apollo-server-express";
import typeDefs from "./typeDefs";
import resolvers from "./resolvers";
```

W tym plik importujemy zależności konieczne do działania serwera. Plik *typeDefs.ts* utworzyliśmy już wcześniej, natomiast plikiem *resolvers.ts* zajmiemy się niebawem.

3. Teraz utwórz obiekt app reprezentujący serwer Express:

```
const app = express();
```

4. Wywołanie funkcji *makeExecutableSchema* tworzy programowy schemat na podstawie definicji typów i resolverów pochodzących odpowiednio z plików *typeDefs.ts* i *resolvers.ts*.

```
const schema = makeExecutableSchema({ typeDefs, resolvers });
```

5. I w końcu tworzymy instancję serwera GraphQL Apollo:

```
const apolloServer = new ApolloServer({
  schema,
  context: ({ req, res }: any) => ({ req, res }),
});
apolloServer.applyMiddleware({ app, cors: false });
```

Właściwość *context* składa się z obiektów żądania i odpowiedzi Expressa. Następnie dodajemy oprogramowanie pośrednie, którym w przypadku serwera GraphQL jest app — obiekt serwera Express. Zastosowana opcja *cors* oznacza, że GraphQL nie powinien działać jako serwer CORS. Zagadnienia związane z CORS opisałem dokładniej w dalszej części książki, przy okazji prezentowania kolejnych etapów prac nad naszą aplikacją.

W kolejnym fragmencie kodu uruchamiany serwer Express na porcie 8000:

```
app.listen({ port: 8000 }, () => {
  console.log("Serwer GraphQL został uruchomiony");
});
```

Funkcja obsługi przekazana w wywołaniu funkcji *listen* jedynie wyświetla informację o uruchomieniu serwera.

A teraz zajmijmy się przygotowaniem resolverów:

1. Utwórz plik *resolvers.ts* i zapisz w nim poniższy kod:

```
import { IResolvers } from "apollo-server-express";
import { v4 } from "uuid";
import { GqlContext } from "./GqlContext";
```

```
interface User {
  id: string;
  username: string;
  description?: string;
}

interface Todo {
  id: string;
  title: string;
  description?: string;
}
```

2. Używamy języka TypeScript, zatem chcemy przygotować typy danych, których następnie użyjemy do reprezentowania zwracanych obiektów. Właśnie do tego będą służyć typy `User` i `Todo` zdefiniowane w powyższym przykładzie. Będą one porównywane przez GraphQL z jego typami o tych samych nazwach, utworzonymi w pliku *typeDefs.ts*:

```
const resolvers: IResolvers = {
  Query: {
    getUser: async (
      obj: any,
      args: {
        id: string;
      },
      ctx: GqlContext,
      info: any
    ): Promise<User> => {
      return {
        id: v4(),
        username: "Dawid",
      };
    },
  },
}
```

Ten fragment kodu przedstawia funkcję naszego pierwszego resolvera, odpowiadającą zapytaniu `getUser`. Zwróć uwagę na to, że definiuje ona znacznie więcej parametrów niż tylko `id`. Będą one przekazywane przez serwer GraphQL Apollo i wzbogacą wywołanie resolvera o dodatkowe informacje. (Zauważ też, że w celu uproszczenia przykładu zawartość zwracanego obiektu `User` jest określona na stałe). Używany w tym kodzie obiekt `GqlContext` utworzymy już niebawem, na razie wystarczy, byś wiedział, że jest to pojemnik zawierający obiekty żądania i odpowiedzi, które przedstawiłem w rozdziale 8., pt. „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa”.

3. Drugi z naszych resolverów, `getTodos`, używa tych samych parametrów co resolver `getUser` i także zwraca dane określone na stałe:

```
getTodos: async (
  parent: any,
  args: null,
  ctx: GqlContext,
  info: any
): Promise<Array<Todo>> => {
  return [
```

```

    {
      id: v4(),
      title: "Zadanie pierwsze",
      description: "Opis pierwszego zadania",
    },
    {
      id: v4(),
      title: "Zadanie drugie",
      description: "Opis drugiego zadania",
    },
    {
      id: v4(),
      title: "Zadanie trzecie",
    },
  ],
};

```

4. Na samym końcu pliku eksportujemy obiekt `resolvers`:

```

  },
};

export default resolvers;

```

Jak widać, funkcje odpowiedzialne za pobieranie danych z serwera GraphQL są zwyczajnym kodem napisanym w TypeScriptie. Gdybyśmy używali Javy, C# lub jakiegokolwiek innego języka programowania, to resolwery byłyby zwyczajnymi funkcjami typu **CRUD**<sup>1</sup> napisanymi w tych językach. Następnie serwer GraphQL przekształca modele danych encji na typy zdefiniowane w naszym schemacie.

5. Teraz zajmiemy się utworzeniem typu `GqlContext`. Utwórz plik `GqlContext.ts` i zapisz w nim następujący kod:

```

import { Request, Response } from "express";

export interface GqlContext {
  req: Request;
  res: Response;
}

```

To bardzo prosty interfejs, którego jedynym celem jest zapewnienie bezpieczeństwa typów dla kontekstu przekazywanego w wywołaniach resolwerów. Jak widać, typ `GqlContext` zawiera jedynie dwa obiekty typów `Request` i `Response` serwera Express.

6. Teraz musimy skompilować nasz kod do JavaScriptu, gdyż napisaliśmy go w TypeScriptie. W tym celu wykonaj następujące polecenie:

```
tsc
```

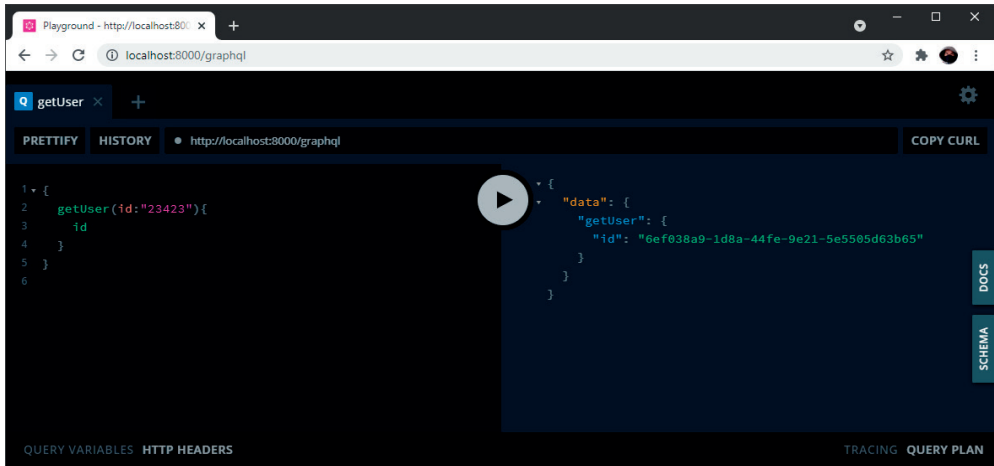
W ten sposób zostaną wygenerowane wersje `.js` wszystkich plików `.ts`.

<sup>1</sup> **CRUD** to skrót od angielskich słów *Create*, *Read*, *Update*, *Delete*; termin ten reprezentuje cztery podstawowe operacje wykonywane na bazach danych: utworzenie, odczyt, aktualizację oraz usuwanie — *przyp. tłum.*

7. I w końcu możemy uruchomić nasz kod; wykonaj zatem poniższe polecenie:

```
nodemon server.js
```

8. Jeśli teraz wyświetlisz w przeglądarce stronę `http://localhost:8000/graphql`, wyświetlony zostanie „plac zabaw” GraphQL-a — strona służąca do testowania zapytań udostępniana przez serwer Apollo i pozwalająca na ręczne wpisywanie poleceń i przeglądanie zwracanych przez nie wyników. Jej przykładową postać przedstawiłem na rysunku 9.2.



Rysunek 9.2. Klient GraphQL przeznaczony do testowania zapytań

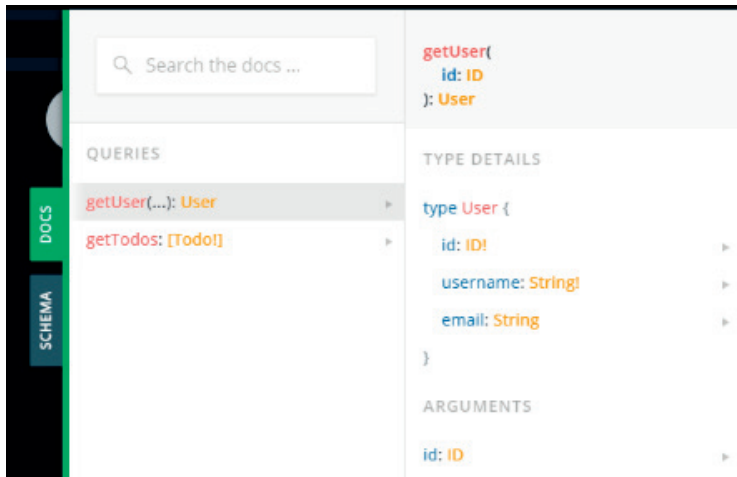
Zwróć uwagę na to, że w przykładzie przedstawionym na rysunku 9.2 wykonałem już jedno zapytanie; jest ono widoczne z lewej strony i wyglądem przypomina ono nieco kod JSON. Z kolei po prawej stronie są widoczne wyniki, które są obiektem zapisanym w formacie JSON. Jeśli przyjrzyj się zapytaniu po lewej stronie, zauważysz zapewne, że pobiera ono wyłącznie pole `id` i to właśnie dlatego w wynikach zostało zwrócone tylko to jedno pole. Zauważ ponadto, że standardową postacią, w jakiej są zwracane dane, jest `data > <nazwa funkcji> > <pola>`. W ramach testów spróbuj wykonać zapytanie `getTodos`.

9. Kolejną rzeczą, na którą powinieneś zwrócić uwagę, jest karta *DOCS*. Przedstawia ona wszystkie dostępne zapytania, mutacje oraz subskrypcje (zajmiemy się nimi w następnym podrozdziale). Jej przykładową postać przedstawiłem na rysunku 9.3.
10. Karta *SCHEMA* przedstawia informacje o schemacie typów stosowanych we wszystkich naszych encjach i zapytaniach (patrz rysunek 9.4).

Jak widać, jej zawartość odpowiada kodowi z pliku `typeDefs.ts`.

W tym podrozdziale uruchomiliśmy serwer GraphQL Apollo i przyjrzelśmy się resolverom. Resolvery są jednym z elementów koniecznych do zapewnienia działania serwera GraphQL. Zamieszczony tu przykład pokazał także, że korzystając z biblioteki Apollo GraphQL stosunkowo łatwo można uruchomić własny serwer GraphQL.





Rysunek 9.3. Karta DOCS



Rysunek 9.4. Karta SCHEMA

W następnym podrozdziale przyjrzymy się dokładniej zapytaniom, a konkretnie: mutacjom i subskrypcjom.

## Zapytania, mutacje oraz subskrypcje

Podczas tworzenia API korzystającego z GraphQL-a, zazwyczaj będziemy chcieli robić więcej niż jedynie pobierać statyczne dane; na przykład możemy chcieć zapisywać dane do magazynu lub otrzymywać informacje o ich modyfikacji. W tym podrozdziale pokażę, jak można to robić.

W pierwszej kolejności zajmiemy się zapisywaniem danych przy użyciu mutacji:

1. Przygotujemy mutację o nazwie `addTodo`, jednak, aby jej działanie było bardziej realistyczne, musimy użyć w naszej przykładowej aplikacji jakiegoś magazynu danych. Do celów testowych zastosujemy magazyn danych przechowywany w pamięci. Utwórz plik `db.ts` i zapisz w nim następujący kod:

```
import { v4 } from "uuid";

export const todos = [
  {
    id: v4(),
    title: "Zadanie pierwsze",
    description: "Opis pierwszego zadania",
  },
  {
    id: v4(),
    title: "Zadanie drugie",
    description: "Opis drugiego zadania",
  },
  {
    id: v4(),
    title: "Zadanie trzecie",
  },
];
```

Jak widać, dane, których wcześniej używaliśmy w resolverze `getTodos`, teraz umieściliśmy w tablicy, którą eksportujemy z pliku.

2. Teraz dodaj do pliku `typeDefs.ts` nową mutację. Zmodyfikowaną wersję kodu przedstawiłem na poniższym przykładzie:

```
import { gql } from "apollo-server-express";

const typeDefs = gql`
  type User {
    id: ID!
    username: String!
    email: String
  }

  type Todo {
    id: ID!
    title: String!
    description: String
  }
`
```

```

type Query {
  getUser(id: ID!): User
  getTodos: [Todo!]
}

type Mutation {
  addTodo(title: String!, description: String): Todo
}
`
;

export default typeDefs;

```

Jak widać, w zapytaniach nic się nie zmieniło, natomiast dodaliśmy do pliku nowy typ, `Mutation`, w którym będą definiowane wszystkie zapytania wprowadzające zmiany w danych. Do tego typu dodaliśmy także pierwszą mutację o nazwie `addTodo`.

3. Kolejnym krokiem będzie dodanie resolvera `addTodo`. W tym celu na początku pliku `resolvers.ts` dodaj instrukcję importującą nasz prosty magazyn danych:

```

import { IResolvers } from "apollo-server-express";
import { v4 } from "uuid";
import { GqlContext } from "../GqlContext";
import { todos } from "../db";

```

Następnie dodaj poniższy kod do pliku `resolvers.ts`:

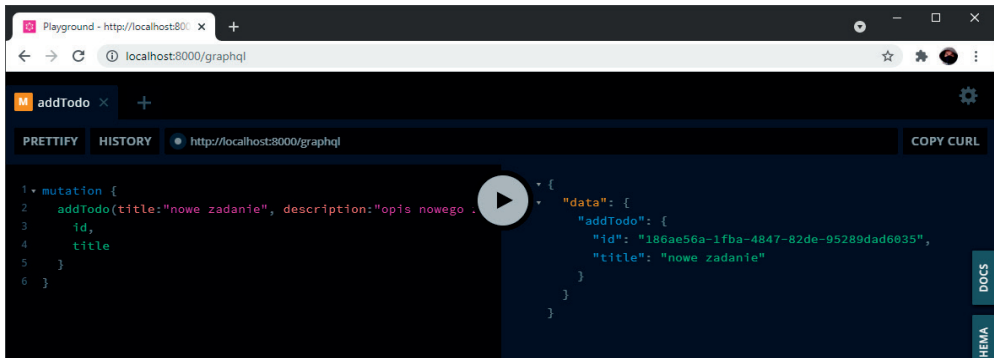
```

Mutation: {
  addTodo: async (
    parent: any,
    args: {
      title: string;
      description: string;
    },
    ctx: GqlContext,
    info: any
  ): Promise<Todo> => {
    todos.push({
      id: v4(),
      title: args.title,
      description: args.description
    });
    return todos[todos.length - 1];
  },
},

```

W tym fragmencie kodu utworzyliśmy nowy pojemnik o nazwie `Mutation`, a wewnątrz niego zdefiniowaliśmy mutację `addTodo`. Jak widać, mutacje mają podobne parametry jak zapytania, a ta konkretna mutacja będzie dodawać do tablicy `todos` nowy obiekt `Todo`. Jeśli wypróbujemy ten kod na stronie testowej serwera GraphQL Apollo, uzyskamy wyniki podobne do tych z rysunku 9.5.

Jeśli wykonywane zapytanie jest typu `Query`, to jego prefiks można pominąć. Jednak w tym przypadku mamy do czynienia z mutacją, więc podanie prefiksu jest konieczne. Jak widać, mutacja zwróciła jedynie pola `id` oraz `title`, gdyż tylko o nie poprosiliśmy.



Rysunek 9.5. Strona testowa serwera GraphQL Apollo po wykonaniu mutacji

Kolejnym zagadnieniem, któremu się przyjrzymy, są subskrypcje. Subskrypcje są mechanizmem, dzięki któremu możemy być informowani o zmianach wprowadzanych w określonych danych. Przykładowo przyjmijmy, że chcemy być informowani, kiedy mutacja `addTodo` doda do listy zadań nowy obiekt `Todo`.

1. W pierwszej kolejności musimy dodać do obiektu kontekstu (`context`) serwera GraphQL obiekt typu `PubSub` udostępniany przez bibliotekę `apollo-server-express`. Ten obiekt zapewnia zarówno możliwość subskrybowania (czyli zgłoszenie prośby o przekazywanie informacji o zmianach, które będą w nich wprowadzane), jak i do publikowania (czyli wysyłania powiadomień o zmianie). Zmodyfikuj plik `server.ts` zgodnie z poniższym przykładem:

```
import express from "express";
import { createServer } from "http";
import {
  ApolloServer,
  makeExecutableSchema,
  PubSub
} from "apollo-server-express";
import typeDefs from "../typeDefs";
import resolvers from "../resolvers";
```

W tym fragmencie kodu importujemy typ `PubSub` oraz funkcję `createServer`. Będziemy ich używać w dalszej części kodu.

2. Następnie tworzymy obiekt `pubsub` typu `PubSub`:

```
const app = express();
const pubsub = new PubSub();
```

3. Kolejnym krokiem jest dodanie obiektu `pubsub` do obiektu kontekstu serwera GraphQL (`context`), dzięki czemu będziemy mogli go używać w resolverach:

```
const schema = makeExecutableSchema({ typeDefs, resolvers });

const apolloServer = new ApolloServer({
  schema,
  context: ({ req, res }: any) => ({ req, res, pubsub }),
});
```

4. Kolejną modyfikacją jest utworzenie instancji serwera Node przy użyciu funkcji `httpServer`, a potem wywołanie na jej rzecz funkcji `installSubscription` ↪ `Handlers`. W następnym fragmencie kodu, w którym wywołujemy `listen`, robimy to na rzecz obiektu `httpServer`, a *nie* obiektu `app`:

```
apolloServer.applyMiddleware({ app, cors: false });
const httpServer = createServer(app);
apolloServer.installSubscriptionHandlers(httpServer);

httpServer.listen({ port: 8000 }, () => {
  console.log("Serwer GraphQL został uruchomiony - "
    + apolloServer.graphqlPath);
  console.log("Serwer subskrypcji GraphQL został uruchomiony - "
    + apolloServer.subscriptionsPath);
});
```

5. A teraz dodajmy do pliku *typeDefs.ts* nową mutację; przedstawiłem ją na poniższym fragmencie kodu:

```
type Subscription {
  newTodo: Todo!
}
```

6. Teraz musimy zmienić definicję interfejsu `GqlContext` zapisaną w pliku *GqlContext.ts*; zmodyfikuj ją zgodnie z poniższym fragmentem kodu:

```
export interface GqlContext {
  req: Request;
  res: Response;
  pubsub: PubSub;
}
```

7. W kolejnym kroku dodamy do pliku *resolvers.ts* nowy resolver subskrypcji:

```
import { IResolvers } from "apollo-server-express";
import { v4 } from "uuid";
import { GqlContext } from "../GqlContext";
import { todos } from "../db";

interface User {
  id: string;
  username: string;
  description?: string;
}

interface Todo {
  id: string;
  title: string;
  description?: string;
}

const NEW_TODO = "NEW TODO";
```

W tym fragmencie kodu zdefiniowaliśmy nową stałą `NEW_TODO`, która będzie pełnić rolę nazwy naszej subskrypcji. Subskrypcje wymagają unikalnych etykiet, pełniących rolę jakby unikalnych kluczy, które są używane do subskrybowania oraz publikowania:

```
const resolvers: IResolvers = {
  Query: {
    getUser: async (
      obj: any,
      args: {
        id: string;
      },
      ctx: GqlContext,
      info: any
    ): Promise<User> => {
      return {
        id: v4(),
        username: "Dawid",
      };
    },
  },

```

Jak widać, w tym fragmencie kodu nic się nie zmieniło, jednak zamieściłem go tu, by przedstawić całą aktualną zawartość pliku *resolvers.ts*.

```
    getTodos: async (
      parent: any,
      args: null,
      ctx: GqlContext,
      info: any
    ): Promise<Array<Todo>> => {
      return [
        {
          id: v4(),
          title: "Zadanie pierwsze",
          description: "Opis pierwszego zadania",
        },
        {
          id: v4(),
          title: "Zadanie drugie",
          description: "Opis drugiego zadania",
        },
        {
          id: v4(),
          title: "Zadanie trzecie",
        },
      ];
    },
  },

```

Także to zapytanie pozostaje bez zmian:

```
Mutation: {
  addTodo: async (
    parent: any,
    args: {
      title: string;
      description: string;
    },
    { pubsub }: GqlContext,

```

Zwróć uwagę na to, że zamiast obiektu `ctx` zastosowaliśmy tu wyrażenie destrukuryzujące i pobieramy z kontekstu jedynie obiekt `pubsub` — to jedyna rzecz, której tutaj potrzebujemy.

```

    info: any
  ): Promise<Todo> => {
    const newTodo = {
      id: v4(),
      title: args.title,
      description: args.description
    }
    todos.push(newTodo);
    pubsub.publish(NEW_TODO, { newTodo });
  }

```

W tym fragmencie kodu zastosowaliśmy funkcję `publish`; dzięki niej będziemy powiadamiani, kiedy zostanie dodany nowy obiekt `Todo`. Zwróć uwagę, że w wywołaniu funkcji `publish` przekazaliśmy także nowy obiekt `newTodo`, dzięki czemu później także ten obiekt będziemy mogli przekazać subskrybentom.

```

    return todos[todos.length - 1];
  },
},
Subscription: {
  newTodo: {
    subscribe: (parent, args: null, { pubsub }: GqlContext) =>
      pubsub.asyncIterator(NEW_TODO),
  },
},

```

W tym fragmencie kodu subskrybujemy powiadomienia o operacjach dodawania nowych zadań (obiektów `Todo`). Zwróć uwagę na to, że wartością pola `newTodo` nie jest funkcja, a obiekt z jedną składową o nazwie `subscribe`.

```

  },
},
};

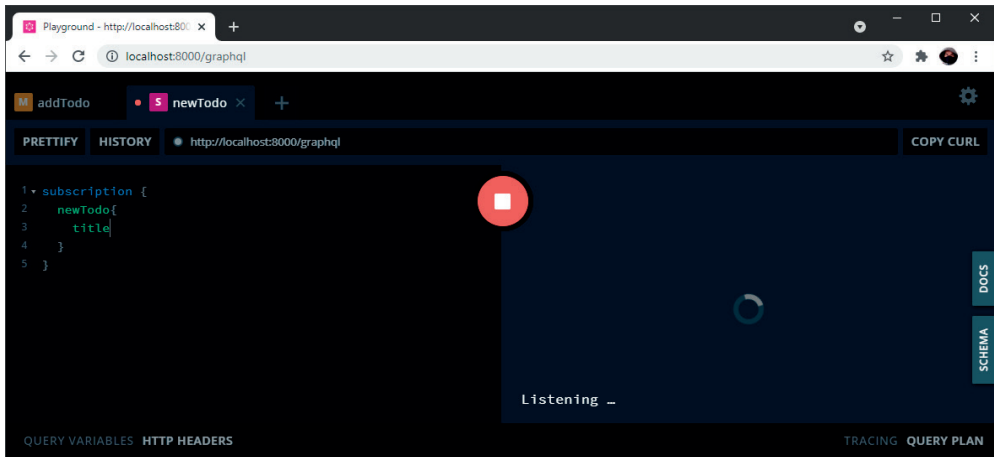
export default resolvers;

```

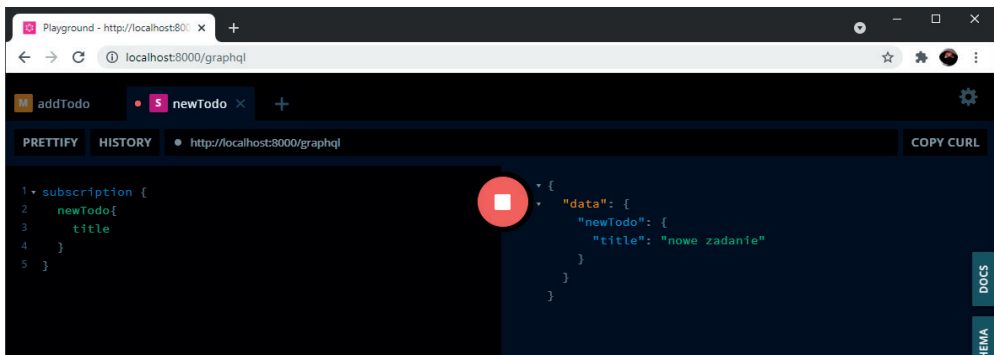
Pozostałej części kodu nie zmieniamy.

8. Teraz pozostało nam jedynie wypróbować wprowadzone zmiany. W pierwszej kolejności nie zapomnij skompilować kodu wykonując polecenie `tsc`, a następnie odśwież stronę testową serwera GraphQL. Następnie otwórz nową kartę, wpisz subskrypcję (patrz rysunek 9.6), po czym kliknij przycisk *Execute* widoczny pośrodku strony.

Kiedy klikniesz przycisk *Execute*, początkowo nic się nie stanie, gdyż na razie nie zostało jeszcze dodane żadne zadanie. Dlatego wyświetl ponownie kartę *addTodo* i dodaj nowe zadanie. Kiedy to zrobisz, wróć na kartę *newTodo*, a przekonasz się, że będzie ona wyglądać jak na rysunku 9.7.



Rysunek 9.6. Subskrypcja newTodo



Rysunek 9.7. Efekty działania subskrypcji newTodo

Jak widać, wszystko działa — zostało wyświetlone nowe zadanie.

Z tego podrozdziału dowiedziałeś się, jak tworzyć zapytania, mutacje i subskrypcje GraphQL. Będziemy ich używać w dalszej części książki, kiedy zajmiemy się tworzeniem API przykładowej aplikacji. GraphQL jest standardem przemysłowym, a to oznacza, że wszystkie klienty mogą działać z serwerami GraphQL dowolnych producentów. Co więcej, klienty używające API GraphQL mogą oczekiwać spójnego działania i obsługi tego samego języka niezależnie od wykorzystywanego serwera oraz jego producenta. Na tym właśnie polega ogromna zaleta GraphQL-a.



## Podsumowanie

Z tego rozdziału dowiedziałeś się, czym jest GraphQL — jedna z najpopularniejszych nowych technologii tworzenia internetowych API — oraz poznałeś jego ogromne możliwości. GraphQL daje bardzo duże możliwości, a dodatkowo, ze względu na to, że jest standardem przemysłowym, zapewnia spójne działanie niezależnie od używanego serwera, frameworka oraz używanego języka programowania.

W następnym rozdziale zaczniemy łączyć wszystkie poznane wcześniej technologie i przygotujemy serwer Express, używając przy tym języka TypeScript, GraphQL-a oraz kilku bibliotek pomocniczych.

# Konfiguracja projektu Expressa z zależnościami od języków TypeScript i GraphQL

Jednym z największych problemów związanych z nauką nowoczesnych sposobów programowania w języku JavaScript jest ogromna liczba dostępnych pakietów i zależności. Wybór właściwego zbioru pakietów dla projektu może być zadaniem przytłaczającym i onieśmiałającym. W tym rozdziale pokażę, jak przygotować dobrze skonfigurowany projekt korzystający z języka TypeScript oraz serwerów Express i GraphQL. Opiszę w nim, które zależności są najbardziej popularne i jak skorzystać na ich zastosowaniu w projekcie.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- utworzeniem projektu Expressa pisanego w języku TypeScript;
- dodaniem do projektu GraphQL-a oraz innych zależności;
- przedstawieniem pakietów pomocniczych.

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś dysponować podstawową wiedzą z zakresu tworzenia aplikacji internetowych przy użyciu Node, Express oraz GraphQL-a. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial10 (Chap10)*.

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog, o nazwie *rozdzial10*.

## Tworzenie projektu Expressa tworzonego w języku TypeScript

W tym podrozdziale utworzymy startowy projekt, który posłuży jako podstawa dla przygotowania serwera. Ręcznie przeanalizujemy i wybierzemy wszystkie zależności, a przy okazji wyjaśnię rolę, jaką będą one pełnić w tworzonej aplikacji. Kiedy skończymy, będziemy dysponować solidną podstawą do utworzenia naszej aplikacji serwerowej.

Dostępnych jest wiele szablonów projektów dla środowiska Node. Bardzo popularnym szablonem do tworzenia aplikacji w języku TypeScript jest projekt TypeScript-Node-Starter, przygotowany przez firmę Microsoft. Dysponuje on szeroką gamą użytecznych zależności; niestety, jest on przeznaczony dla użytkowników bazy danych MongoDB, my natomiast będziemy używali Postgresa.

Kolejny szablon projektu nosi nazwę *express-generator* i został przygotowany przez twórców Expressa. Dysponuje on narzędziem obsługiwany z poziomu wiersza poleceń, które pozwala na podawanie parametrów i na ich podstawie generuje bazowy projekt. Jednak ten generator jest przeznaczony głównie do przygotowywania aplikacji serwerów WWW, które generują kod HTML przy użyciu mechanizmów pug i ejs. Te rozwiązania nie są nam potrzebne, gdyż naszym celem jest przygotowanie API dla aplikacji jednostronicowej. Co więcej, ten generator nie uwzględnia wykorzystania GraphQL-a, z którego my chcemy skorzystać w projekcie.

Dlatego, aby wyeliminować instalowanie niepotrzebnych pakietów oraz w ramach ćwiczenia, przygotujemy nasz projekt ręcznie. W ten sposób dokładnie poznasz wszystkie elementy, które będą niezbędne do stworzenia naszej aplikacji, a ja przy okazji wyjaśnię, co robi każdy z nich. W celu przygotowania bazowego projektu wykonaj czynności opisane na poniższej liście:

1. W katalogu *rozdzial10* utwórz podkatalog *node-server*.
2. W panelu terminala przejdź do nowego podkatalogu i wykonaj polecenie:

```
npm init
```

3. Następnie zainstaluj i zainicjuj język TypeScript:

```
npm i typescript
tsc -init
```

4. Zaktualizuj plik *tsconfig.json* zgodnie z poniższym przykładem:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "lib": ["ES6", "ES2017", "ES2018", "ES2019", "ES2020"],
    "sourceMap": true,
    "outDir": "./dist",
    "rootDir": "src",
    "moduleResolution": "node",
    "removeComments": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "noImplicitThis": true,
    "noUnusedLocals": true,
    "noUnusedParameters": false,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
  "exclude": ["node_modules"],
  "include": [".src/**/*.tsx", ".src/**/*.ts"]
}
```

Informacje o pliku *tsconfig.json* podałem już w rozdziale 2., pt. „Prezentacja języka TypeScript”, jednak poniżej jeszcze raz opiszę najważniejsze z zapisanych w nim ustawień.

- Możemy generować kod JavaScript zgodny z wersją ES6 tego języka, gdyż kod będzie działał na naszym serwerze, a to oznacza, że możemy dobrać wymaganą wersję silnika V8, instalując odpowiednią wersję Node.js.
- Będziemy używali systemu modułów (module) commonjs, aby uniknąć problemów z mieszaniem instrukcji `require` i `import`.
- Chcemy używać najnowszych wersji JavaScriptu, więc użyta wartość pola `lib` pozwala na to.
- Pole `outDir` określa katalog, w którym będą zapisywane pliki *.js* generowane przez kompilator TypeScript.
- Pole `rootDir` określa główny katalog kodów źródłowych.

- Zezwalamy na generowanie dodatkowych informacji, `emitDecorator` → `Metadata` i `experimentalDecorator`, gdyż tego wymaga `TypeORM` — warstwa repozytorium obsługująca bazę danych, której będziemy używali.
  - Pola `exclude` oraz `include`, jak łatwo się domyślić, reprezentują katalogi, które — odpowiednio — chcemy ukryć przed kompilatorem `TypeScriptu` lub które mają być przez niego uwzględniane.
5. Teraz dodajmy do projektu kilka kolejnych podstawowych zależności:

```
npm i express -S
npm i @types/express jest @types/jest ts-jest nodemon ts-node-dev faker
  ↳ @types/faker -D
```

Przyjrzyjmy się dokładniej tym pakietom:

- Instalujemy framework `Express` oraz jego typy danych stosowane w kodzie `TypeScript`.
- Instalujemy bibliotekę `jest` (wraz z jej typami) do tworzenia i wykonywania testów.
- Instalujemy pakiet `ts-jest`, używany do pisania testów w języku `TypeScript`.
- Dołączyłem do tego narzędzie `nodemon`, aby wyraźnie pokazać, że warto je dodać do projektu, choć my będziemy używali jego globalnie dostępnej wersji, którą zainstalowaliśmy w systemie w rozdziale 8., pt. „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem `Node.js` i `Expressa`”.
- Biblioteka `faker` służy do generowania fikcyjnych danych na potrzeby pisania testów i tworzenia *atrap*.
- Pakiet `ts-node-dev` zapewni nam możliwość automatycznego restartowania serwera `Node` po zmodyfikowaniu plików źródłowych `TypeScript`.

Skoro już zainstalowaliśmy podstawowe zależności naszego projektu, możemy uruchomić bazową wersję serwera `Express` i upewnić się, że wszystko działa jak należy:

1. Będziemy musieli przygotować skrypt uruchomieniowy serwera, który go zainicjuje. Zrobimy to w taki sam sposób jak w rozdziale 8., pt. „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem `Node.js` i `Expressa`”. A zatem, utwórz katalog *src*, a w nim plik *index.ts*; następnie zapisz w tym pliku poniższy kod:

```
import express from "express";
import { createServer } from "http";

const app = express();

const server = createServer(app);

server.listen({ port: 8000 }, () => {
  console.log("Serwer został uruchomiony!");
});
```

To jest dokładnie to samo, co robiliśmy już wcześniej: tworzymy instancję express, a następnie uruchamiamy serwer.

2. Teraz zajmijmy się przygotowaniem w pliku *package.json* skryptu "start". Otwórz ten plik i odśzukaj w nim sekcję "scripts". Następnie, poniżej istniejącego już skryptu "test" dodaj kolejny skrypt, jak pokazałem na poniższym przykładzie:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "ts-node-dev --respawn src/index.ts"
},
```

To polecenie korzysta z narzędzia *ts-node-dev*, aby monitorować zmiany wprowadzane w plikach źródłowych *.ts* i w razie ich wykrycia restartować serwer (co powoduje użycie opcji *respawn*). A to oznacza, że w razie konieczności, serwer Node będzie automatycznie restartowany.

3. Po dodaniu tego nowego polecenia możesz już uruchomić serwer:

**npm start**

W efekcie, w panelu terminala powinieneś zobaczyć komunikat przedstawiony na rysunku 10.1.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS
+ faker@5.5.3
added 572 packages from 388 contributors and audited 624 packages in 48.809s

33 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

H:\NaukaTypeScriptu\rozdzial10\node-server>npm start

> server@1.0.0 start H:\NaukaTypeScriptu\rozdzial10\node-server
> ts-node-dev --respawn src/index.ts

[INFO] 23:20:47 ts-node-dev ver. 1.1.6 (using ts-node ver. 9.1.1, typescript ver. 4.3.2)
Serwer został uruchomiony!
```

**Rysunek 10.1.** Pierwsze uruchomienie serwera

Jak widać, wykonanie polecenia spowodowało uruchomienie serwera i wyświetlenie na ekranie komunikatu informującego, że serwer działa.

4. Jeśli teraz zmodyfikujesz plik *index.ts* zmieniając, na przykład, treść komunikatu, przekonasz się, że serwer zostanie automatycznie ponownie uruchomiony, jak pokazałem na rysunku 10.2.

Jak widać na rysunku 10.2, serwer został ponownie uruchomiony, o czym świadczy nowy komunikat wyświetlony w panelu terminala.

W tym podrozdziale zaczęłeś poznawać ważne zależności naszego serwera. Wszystkich tych pakietów, jak również paru innych, będziemy używali do stworzenia API GraphQL. Zależności związane z GraphQL-em dodamy w następnym podrozdziale.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

run 'npm fund' for details

found 0 vulnerabilities

H:\NaukaTypeScriptu\rozdzial10\node-server>npm start

> server@1.0.0 start H:\NaukaTypeScriptu\rozdzial10\node-server

> ts-node-dev --respawn src/index.ts

[INFO] 23:20:47 ts-node-dev ver. 1.1.6 (using ts-node ver. 9.1.1, typescript ver. 4.3.2)

Server został uruchomiony!

[INFO] 23:25:41 Restarting: H:\NaukaTypeScriptu\rozdzial10\node-server\src\index.ts has been modified

Server został uruchomiony i !

[INFO] 23:25:45 Restarting: H:\NaukaTypeScriptu\rozdzial10\node-server\src\index.ts has been modified

Server został uruchomiony i świetnie działa!

□

Rysunek 10.2. Ponowne uruchomienie serwera

## Dodawanie do projektu GraphQL-a i jego zależności

Zagadnienia związane z GraphQL-em przedstawiłem po raz pierwszy w rozdziale 9., pt. „Czym jest GraphQL?”. Tutaj przyjrzymy się jeszcze raz pakietom, z których już korzystaliśmy, oprócz tego przedstawię kilka nowych, których będziemy używać.

Na poniższej liście przedstawiłem pakiety związane z GraphQL-em, których będziemy używać w naszej aplikacji:

### ■ graphql

Ten pakiet zawiera wzorcową implementację GraphQL-a dla języka JavaScript. Został on stworzony przez fundację GraphQL, a my będziemy go używali do testowania zapytań GraphQL.

### ■ graphql-middleware

Ten pakiet pozwala na wstrzykiwanie własnego kodu przed wykonaniem resolverów lub po nim. Można go używać, między innymi, do uwierzytelniania oraz rejestrowania wykonywanych operacji w dziennikach.

### ■ graphql-tools

Ten pakiet zawiera narzędzia ułatwiające testowanie i tworzenie atrap zapytań GraphQL.

### ■ apollo-server-express

To główna biblioteka, której będziemy używali do przygotowania serwera GraphQL korzystającego z Expressa, czyli rozwiązania, które zastosowaliśmy już w rozdziale 9., pt. „Czym jest GraphQL?”.

Powyższa lista obejmuje główne pakiety, których będziemy używali w naszej implementacji GraphQL-a. Teraz zajmiemy się uruchomieniem serwera GraphQL oraz przygotowaniem kilku testów, które zweryfikują jego działanie. W kolejnych rozdziałach do tych podstawowych

pakietów będziemy dodawali kolejne, łącząc je wszystkie w jeden projekt. Wykonaj czynności opisane na poniższej liście, aby uruchomić serwer GraphQL:

1. Wewnątrz katalogu *rozdzial10* utwórz podkatalog *gql-server*, następnie przejdź do niego w panelu terminala i wykonaj polecenie:

```
npm init
```

2. Zaakceptuj wszystkie domyślne ustawienia, a następnie wykonaj kolejne polecenie:

```
npm i express graphql graphql-tools graphql-middleware apollo-server-express
↳ uuid -S
```

3. Po jego zakończeniu, wykonaj następne polecenie:

```
npm i @types/express typescript @types/faker @types/jest faker jest nodemon
↳ ts-jest ts-node-dev @types/uuid -D
```

4. Teraz zainicjuj projekt języka TypeScript, wykonując w tym celu następujące polecenie:

```
tsc -init
```

5. Kiedy polecenie zostanie wykonane, zastąp wygenerowany plik *tsconfig.json* plikiem o tej samej nazwie z katalogu *node-server*.

6. W kolejnym kroku dodaj do sekcji *scripts* w pliku *package.json* polecenie *start* (patrz rysunek 10.3).

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "ts-node-dev --respawn src/server.ts"
},
```

Rysunek 10.3. Polecenie start

7. W katalogu *gql-server* utwórz podkatalog *src* i skopiuj do niego następujące pliki z katalogu *rozdzial09/graphql-schema*: *db.ts*, *GqlContext.ts*, *resolvers.ts*, *server.ts* i *typeDefs.ts*.
8. I w końcu wykonaj poniższe polecenie, aby sprawdzić, czy serwer działa.

```
npm start
```

Teraz dodajmy do naszego projektu oprogramowanie pośrednie i przekonajmy się, czy wszystko działa prawidłowo:

1. W katalogu *src* utwórz plik *Logger.ts* i zapisz w nim następujący kod:

```
export const log = async (
  resolver: any,
  parent: any,
  args: any,
  context: any,
  info: any
) => {
  if (!parent) {
    console.log("Rozpaczynam rejestrowanie...");
```



```

    }

    const result = await resolver(parent, args, context, info);

    console.log("Zakończono wywołanie resolwera.");

    return result;
  };

```

W tym kodzie przechwytujemy wszystkie wywołania resolverów i rejestrujemy je, jeszcze zanim zostaną wykonane — czyli przed wywołaniem funkcji `resolver`. Zwróć uwagę na sprawdzenie, czy obiekt `parent` jest różny od `null`, które informuje o tym, że funkcja resolwera nie została jeszcze wykonana. Dodajmy także rejestrowanie do resolwera `getTodos`. Otwórz plik *resolvers.ts* i dodaj poniższy wiersz kodu na samym początku ciała funkcji `getTodos`, bezpośrednio przed instrukcją `return`:

```
console.log("Wykonuję funkcję getTodos.");
```

2. Teraz musimy zmodyfikować plik *server.ts* i zastosować w nim nową funkcję rejestrującą. Zaktualizuj plik *server.ts* zgodnie z poniższym przykładem:

```

import express from "express";
import { createServer } from "http";
import { ApolloServer, makeExecutableSchema, PubSub } from "apollo-server-
  express";
import typeDefs from "./typeDefs";
import resolvers from "./resolvers";
import { applyMiddleware } from "graphql-middleware";
import { log } from "./Logger";

```

Jak widać, zaimportowaliśmy funkcję `applyMiddleware` oraz przygotowane wcześniej oprogramowanie pośrednie `log`. Zwróć uwagę na to, że funkcja `applyMiddleware` pochodzi z pakietu `graphql-middleware` i nie ma nic wspólnego z funkcją o tej samej nazwie udostępnianą przez serwer Apollo, która jedynie kojarzy z nim instancję Expressa:

```

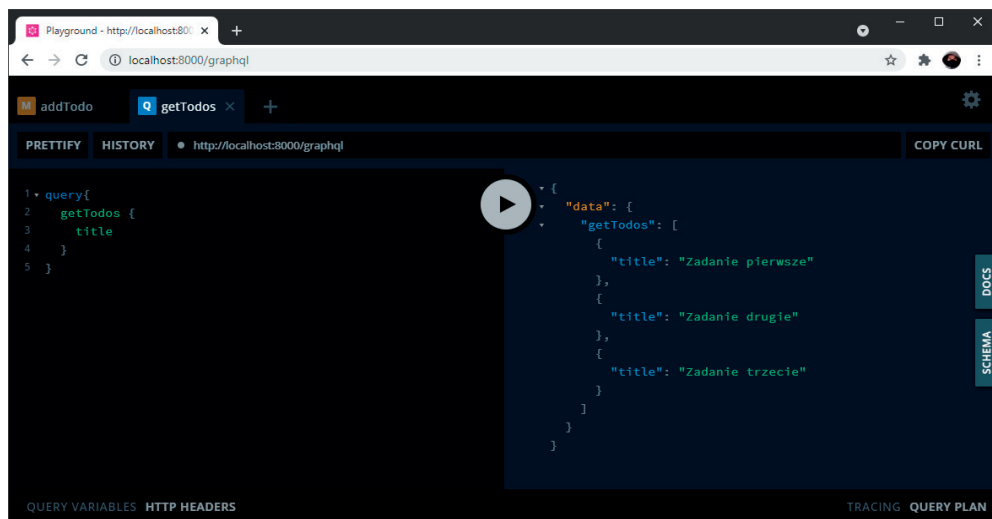
const app : any = express();

const pubsub = new PubSub();
const schema = makeExecutableSchema({ typeDefs, resolvers });
const schemaWithMiddleware = applyMiddleware(schema, log);
const apolloServer = new ApolloServer({
  schema: schemaWithMiddleware,
  context: ({ req, res }: any) => ({ req, res, pubsub }),
});

```

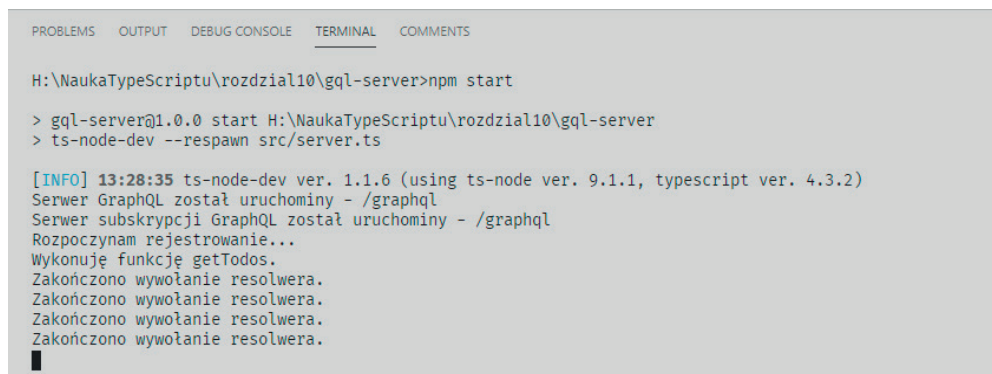
W tym fragmencie kodu użyliśmy schematu utworzonego przy użyciu funkcji `makeExecutableSchema` i przekazaliśmy go w wywołaniu funkcji `applyMiddleware`. W ten sposób uzyskaliśmy schemat powiązany z oprogramowaniem pośrednim. Ten nowy schemat, zapisany w stałej `schemaWithMiddleware`, przekazaliśmy następnie do serwera Apollo. Pozostała część kodu jest taka sama jak wcześniej, więc nie będę jej tutaj przedstawiać.

3. Jeśli jeszcze tego nie zrobiłeś, to uruchom serwer GraphQL i wyświetl w przeglądarce jego stronę testową. Jeśli teraz wykonasz zapytanie `getTodos`, zostaną wyświetlone dane `todos`, przedstawione na rysunku 10.4.



Rysunek 10.4. Wywołanie `getTodos`

Oprócz tego, w panelu terminala Visual Studio Code zostaną wyświetlone komunikaty generowane przez wywołania funkcji `console.log` (patrz rysunek 10.5).



Rysunek 10.5. Wyniki wyświetlone w panelu terminala VSCode po wykonaniu resolvera

Jak widać, nasze oprogramowanie pośrednie działa zgodnie z oczekiwaniami. Widać także, że został wykonany resolver.

Mamy zatem potwierdzenie, że udało się nam zastosować oprogramowanie pośrednie w celu przechwycenia wywołań resolverów i wstawienia własnego kodu do procesu GraphQL-a. Teraz spróbujmy przygotować kilka testów korzystających z serwera GraphQL:

1. Aby móc korzystać z GraphQL-a w testach, musimy przygotować odpowiedni skrypt, który będzie je wykonywać. W tym celu utwórz plik *testGraphQLQuery.ts* i zapisz w nim poniższy kod:

```
import { graphql, GraphQLSchema } from "graphql";
```

Jak widać, zaczynamy od zaimportowania obiektów *graphql* i *GraphQLSchema*, co pozwoli nam na ręczne wykonywanie zapytań i przygotowywanie plików schematów.

2. W kolejnym wierszu kodu musimy zaimportować *Maybe* — typ GraphQL określający czy można, czy też nie można używać parametrów:

```
import { Maybe } from "graphql/jsutils/Maybe";
```

3. Teraz utwórz interfejs *Options*, którego potem, podczas wykonywania zapytań, będziemy używać jako typu parametrów przekazywanych do funkcji *testGraphQLQuery*:

```
interface Options {
  schema: GraphQLSchema;
  source: string;
  variableValues?: Maybe<{ [key: string]: any }>;
}
```

Zapis *[key: string]* reprezentuje nazwy właściwości obiektu, na przykład *myObj["jakaś nazwa"]*. Do funkcja *testGraphQLQuery* będą przekazywane niezbędne parametry, a sama funkcja będzie zwracać odpowiednie dane:

```
export const testGraphQLQuery = async ({
  schema,
  source,
  variableValues,
}: Options) => {
  return graphql({
    schema,
    source,
    variableValues,
  });
};
```

4. Kolejnym krokiem będzie napisanie testu. Utwórz plik *getUser.test.ts* i zapisz w nim poniższy kod:

```
import typeDefs from "../typeDefs";
import resolvers from "../resolvers";
import { makeExecutableSchema } from "graphql-tools";
import faker from "faker";
import { testGraphQLQuery } from "../testGraphQLQuery";
import { addMockFunctionsToSchema } from "apollo-server-express";
```

Przeznaczenie importowanych pakietów jest raczej oczywiste, zaznaczę tylko, że pakiet *faker* posłuży nam do tworzenia fikcyjnych wartości pól testowanych obiektów.

5. W kolejnym fragmencie kodu przygotowujemy test, korzystając przy tym z funkcji `describe`. Następnie tworzymy zapytanie `getUser` i określamy w nim pola, których wartości chcemy pobrać.

```
describe("Testuję pobieranie danych użytkownika", () => {
  const GetUser = `
    query GetUser($id: ID!) {
      getUser(id: $id) {
        id
        username
        email
      }
    }
  `;
});
```

6. W samym teście najpierw tworzymy schemat (`schema`), używając do tego celu definicji typów (`typeDefs`) i resolverów (`resolvers`), po czym przygotowujemy fikcyjne pola danych w obiekcie `User`:

```
it("pobiera odpowiedniego użytkownika", async () => {
  const schema = makeExecutableSchema({ typeDefs, resolvers });
  const userId = faker.random.alphaNumeric(20);
  const username = faker.internet.userName();
  const email = faker.internet.email();
  const mocks = {
    User: () => ({
      id: userId,
      username,
      email,
    }),
  };
});
```

Jak już wyjaśniałem w rozdziale 6., pt. „Przygotowywanie projektu za pomocą `create-react-app` i testowanie go przy użyciu `Jest`”, tworzenie atrap i fikcyjnych danych pozwala nam skoncentrować się na kodzie testu jednostkowego, a nie na innych zagadnieniach.

7. Używając funkcji `addMockFunctionsToSchema` dodajemy fikcyjny obiekt `User` do schematu, tak by był zwracany po wykonaniu odpowiednich zapytań:

```
console.log("id", userId);
console.log("username", username);
console.log("email", email);

addMockFunctionsToSchema({ schema, mocks });
```

8. I w końcu wywołujemy funkcję `testGraphQLQuery`, aby pobrać fikcyjny obiekt `User`:

```
const queryResponse = await testGraphQLQuery({
  schema,
  source: GetUser,
  variableValues: { id: faker.random.alphaNumeric(20) },
});
const result = queryResponse.data ? queryResponse.data.getUser : null;
console.log("result", result);
```



```

H:\NaukaTypeScriptu\rozdzial10\gql-server>jest
ts-jest[main] (WARN) Replace any occurrences of "ts-jest/dist/preprocessor.js" or "<rootDir>/node_modules/ts-jest/
on of your Jest config with just "ts-jest".
PASS src/getUser.test.ts (7.656 s)
  Testuję pobieranie danych użytkownika
    ✓ pobiera odpowiedniego użytkownika (46 ms)

console.log
  id xzccn2x0t35k9bhly52s
    at src/getUser.test.ts:31:13

console.log
  username Brayan91
    at src/getUser.test.ts:32:13

console.log
  email Kian_Zboncak88@gmail.com
    at src/getUser.test.ts:33:13

console.log
  result [Object: null prototype] {
    id: 'xzccn2x0t35k9bhly52s',
    username: 'Brayan91',
    email: 'Kian_Zboncak88@gmail.com'
  }
    at src/getUser.test.ts:43:13

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        8.073 s
Ran all test suites.

H:\NaukaTypeScriptu\rozdzial10\gql-server>

```

Rysunek 10.6. Wyniki testu zapytania GraphQL

## Prezentacja pakietów pomocniczych

W tym podrozdziale przedstawię kilka dodatkowych zależności naszego projektu. Podstawą naszego serwera będzie oczywiście połączenie Node, Expressa oraz GraphQL-a. Jednak do uzyskania pełnych możliwości funkcjonalnych wymaganych przez aplikację, nasz serwer będzie musiał wykonywać także wiele innych działań.

Na poniższej liście wymienię zatem kilka pozostałych pakietów, których będziemy używać i które pozwolą nam pisać mniej kodu, a w zamian bardziej skoncentrować się na logice biznesowej aplikacji:

### ■ bcryptjs

Każdy serwer będzie musiał szyfrować dane w celu zapewnienia odpowiedniego poziomu bezpieczeństwa. Oczywiście przykładem może być przechowywanie haseł użytkowników. Algorytm bcrypt jest przemysłowym standardem szyfrowania, dostępnym na wielu platformach i w wielu językach programowania, w tym w C++ i Javie. Biblioteka bcryptjs jest implementacją tego algorytmu w języku JavaScript, a my wykorzystamy ją do poprawy bezpieczeństwa tworzonej aplikacji.

## ■ cors

Internet jest pełen niebezpieczeństw i hackerów próbujących włamywać się na serwery. Z tego względu standardowym sposobem działania wszystkich serwerów jest zezwalanie na połączenia klienckie tylko ze stron pochodzących z tej samej domeny, co serwer. W przypadku złożonych konfiguracji serwerowych, jakie występują na przykład w serwerach obsługujących mikrousługi lub na serwerach pośredniczących, stosowanie takich rozwiązań nie jest możliwe. Z tego względu została opracowana technologia **CORS** (ang. *Cross-Origin Resource Sharing*), umożliwiająca obsługę żądań przesyłanych z innych domen. Pakiet cors udostępnia narzędzia niezbędne do korzystania z tej technologii na naszym serwerze.

## ■ date-fns

Domyslny obiekt stosowany w języku JavaScript do reprezentacji dat i operowania na nich od zawsze był dziwny i niezbyt wygodny w użyciu, dlatego wykorzystamy pakiet date-fns, który udostępnia wiele użytecznych metod służących do analizowania, formatowania oraz wyświetlania dat i godzin.

## ■ dotenv

Każda duża aplikacja musi przechowywać informacje konfiguracyjne w jednym miejscu, zarówno po to, by zarządzać, jak i po to, by zabezpieczać wrażliwe dane i ustawienia. Pakiet dotenv pozwoli nam przechowywać wrażliwe ustawienia konfiguracyjne w taki sposób, by użytkownicy nie mieli do nich dostępu.

## ■ nodemailer

Pakiet nodemailer zapewnia możliwość wysyłania wiadomości poczty elektronicznej z poziomu serwera Node. Wiadomości e-mail możemy używać, na przykład, by umożliwić użytkownikom resetowanie haseł, czy też do powiadamiania ich o różnych zdarzeniach zachodzących w aplikacji.

## ■ request

Ten pakiet pozwala na generowanie żądań HTTP z poziomu kodu działającego na serwerze Node. Ta możliwość przydaje się, na przykład, kiedy musimy pobrać jakieś dane z innego API, czy to naszego, czy też przygotowanego przez innego twórcę.

## ■ querystring

Pakiet querystring ułatwia tworzenie parametrów adresów URL na podstawie obiektów, pozwala także na przetwarzanie zawartości żądań POST i zapisywanie ich w obiektach. Z powodzeniem można go używać razem z pakietem request.

## ■ randomstring

Ten pakiet może się przydać do generowania losowych, tymczasowych haseł.

Podczas tworzenia naszej aplikacji będziemy także używali wielu innych pakietów, na przykład pozwalających na korzystanie z bazy danych Postgres, czy też magazynu Redis. Jednak będę je wprowadzał w odpowiednich rozdziałach, gdyż dzięki temu łatwiej będzie Ci zrozumieć ich przeznaczenie.

W tym podrozdziale przedstawiłem różne pakiety, których będziemy używać w naszym projekcie. Choć to nie one są najważniejsze w naszej aplikacji, to jednak są niezwykle przydatne i cenne. Gdybyśmy samodzielnie musieli zaimplementować ich możliwości funkcjonalne, stalibyśmy się ekspertami w wielu dziedzinach, takich jak szyfrowanie, czy też arytmetyka na datach i godzinach, co byłoby dla nas straszną stratą czasu, gdyż te zagadnienia zupełnie nie odpowiadają naszym celom.

---

## Podsumowanie

W tym rozdziale poznałeś dodatkowe pakiety NPM, których będziemy używali podczas tworzenia naszej aplikacji. Wszystkie te narzędzia są z powodzeniem stosowane przez społeczność programistów Node, więc są doskonale przetestowane i godne zaufania. Możliwość korzystania z pakietów dostępnych w ekosystemie Node jest jedną z głównych korzyści, jakie daje stosowanie tego środowiska. Dzięki nim możemy znacząco ograniczyć ilość kodu, który będziemy musieli napisać, przetestować, a następnie utrzymywać.

W następnym rozdziale szczegółowo przedstawię aplikację, którą będziemy tworzyć w dalszej części książki. Zaprezentuję w nim jej poszczególne komponenty, po czym zaczniemy pisać kliencką część aplikacji, którą stworzymy korzystając z Reacta.



# Czego się nauczysz — aplikacja internetowego forum

Niezależnie od tego, ile książek przestudiujemy, jako programiści nie będziemy w stanie prawdziwie zrozumieć jak pisać programy działające z wykorzystaniem pewnego stosu technologicznego, dopóki nie stworzymy realistycznego programu, który z takiego stosu będzie korzystać. W tym rozdziale przedstawię Ci aplikację, którą będziemy tworzyć w dalszej części książki. Dowiesz się z niego, w jaki sposób wykorzystamy zagadnienia przedstawione w poprzednich rozdziałach, jakie możliwości będzie mieć tworzona aplikacja oraz dlaczego uznałem, że warto je zaimplementować. Jako programista mam spore doświadczenie w tworzeniu aplikacji typu internetowe forum, czego przykładem może być moja ostatnia aplikacja DzHaven. Możesz zatem mieć pewność, że kod, który przedstawiłem w dalszej części książki, ma jakoś produkcyjną i został zastosowany w rzeczywistej, działającej aplikacji.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- analizą aplikacji, którą napiszemy — aplikacji internetowego forum;
- analizą zagadnienia uwierzytelniania użytkowników forum;
- analizą zarządzania wątkami;
- analizą systemu punktowania wątków.

# Analiza aplikacji, którą napiszemy — internetowego forum

Jak już zaznaczyłem, w następnych rozdziałach zajmiemy się napisaniem aplikacji internetowego forum. Będzie to aplikacja przypominająca inne internetowe fora, takie jak StackOverflow, czy też Bitcontalk.org. Użytkownik będzie mógł opublikować temat lub zadać pytanie oraz otrzymywać odpowiedzi od innych użytkowników aplikacji.

## Po co tworzyć kolejne internetowe forum?

Mógłbym zademonstrować sposoby pisania aplikacji internetowych w języku JavaScript tworząc kilka mniejszych i prostszych aplikacji. Jednak problem demonstrowania sposobów tworzenia aplikacji wykorzystujących pełen stos technologiczny na przykładach prostszych aplikacji polega na tym, że nie są one w stanie zaprezentować wszystkich możliwości nowoczesnych aplikacji tworzonych w JavaScriptcie. Innymi słowy, zastosowanie takiego rozwiązania mogłoby sprawić, że Twoja wiedza nie będzie kompletna i zabraknie w niej pewnych ważnych informacji, takich jak zagadnienia związane z uwierzytelnianiem lub dostępem do baz danych.

Oczywiście moglibyśmy napisać piękną wizualnie aplikację do prezentowania zdjęć lub klipów video, jednak cechą charakterystyczną rozwiązań tego typu jest wielki nacisk kładziony na ich projekt graficzny i estetykę wyglądu. Oprócz tego, choć obróbka zdjęć lub klipów video jest „fajna”, to jednak umiejętności z nią związane nie przekładają się na umiejętność posługiwania się pełnym stosem technologicznym aplikacji internetowych. Oczywiście, w tworzeniu aplikacji tego typu nie ma nic złego, jednak w książkach takich jak ta, głównym zagadnieniem jest przedstawienie sposobów wykorzystania pełnego stosu technologicznego, a niekoniecznie tylko obróbki grafiki.

Z kolei aplikacja internetowego forum stwarza doskonałą okazję do dogłębnego poznania i zrozumienia dziesiątek bibliotek i frameworków stosowanych w aplikacjach korzystających z pełnego stosu technologicznego. Daje sposobność do przedstawienia możliwości, jakie trzeba będzie implementować w aplikacjach obsługujących wielu użytkowników i działających jako ogólnodostępne witryny WWW. Na poniższej liście przedstawiłem najważniejsze, ogólne możliwości, jakie zaimplementujemy w naszej aplikacji:

### ■ Bezpieczeństwo

Generalnie, bezpieczeństwo w kontekście aplikacji internetowych składa się z dwóch podstawowych elementów: uwierzytelniania i autoryzacji. Uwierzytelnianie jest możliwością zweryfikowania przez serwer, czy użytkownik jest tym, za kogo się podaje; z kolei autoryzacja pozwala kontrolować dostęp użytkownika do poszczególnych możliwości aplikacji.

### ■ Sesje i ciasteczka

Sesje pozwalają serwerowi przechowywać dane dotyczące bieżącej aktywności użytkownika w witrynie. My będziemy używali sesji i ciasteczek do identyfikacji użytkowników oraz ułatwiania im korzystania z witryny.

### ■ Odwzorowania obiektowo-relacyjne

Odwzorowania obiektowo-relacyjne (ang. *Object Relational Mapper*, w skrócie **ORM**) to technologia zapewniająca możliwość prowadzenia interakcji z bazami danych przy użyciu kodu programu (w naszym przypadku będzie to kod pisany w języku TypeScript), a nie języka SQL.

### ■ Dostęp do bazy danych i warstwy repozytorium

Zagadnienia związane z dostępem do baz danych są złożone, dlatego też implementując je, wykorzystamy wzorzec projektowy o nazwie Repozytorium (ang. *Repository*), pozwalający odseparować kod związany z dostępem do bazy danych od pozostałego kodu aplikacji.

Obecnie aplikacje muszą domyślnie zapewniać możliwość działania na urządzeniach mobilnych. Musimy zadbać o to, by użytkownicy takich urządzeń mogli brać pełny i aktywny udział w życiu całej społeczności użytkowników naszej aplikacji. Dlatego będziemy implementować ją korzystając z technologii i rozwiązań responsywnych, tak by aplikacja z powodzeniem mogła działać zarówno na komputerach stacjonarnych, jak i przenośnych. Projektowanie aplikacji responsywnych oznacza, że prezentowane ekrany będą się zmieniać, by w możliwie najlepszy sposób dostosować się do wymiarów ekranu urządzenia, na którym aplikacja jest używana. Aby zapewnić taki sposób działania naszego forum, wykorzystamy nowoczesne techniki tworzenia kaskadowych arkuszy stylów (CSS) oraz kodu JavaScript.

W tym podrozdziale wyjaśniłem, jakiego typu aplikację napiszemy i dlaczego zdecydowałem się na wybór akurat takiego rodzaju aplikacji. W następnym podrozdziale opiszę zagadnienia związane z uwierzytelnianiem i przedstawię niektóre cechy rozwiązania, jakie zastosujemy w naszej aplikacji.

## Analiza uwierzytelniania użytkowników forum

W dużych aplikacjach, z których korzysta wielu użytkowników, konieczne jest zastosowanie jakiegoś systemu, który będzie rozpoznawał użytkowników i określał ich uprawnienia. Dotyczy to także naszej aplikacji internetowego forum.

Jej użytkownicy będą w stanie publikować tematy (wątki) oraz odpowiadać na pytania. Dlatego aplikacja będzie musiała rozpoznawać czynności danego użytkownika oraz wszystkich pozostałych. W tym celu zaimplementujemy system logowania pozwalający na uwierzytelnianie użytkowników, a podczas wykonywania wszystkich operacji będziemy uwzględniać unikalne konto użytkownika. Nasza aplikacja będzie zatem dysponować następującymi możliwościami:

### ■ Logowanie wraz z możliwością wylogowania

Ta możliwość będzie wykorzystywać nie tylko resolvery **GraphQL** obsługujące logowanie oraz wylogowanie, lecz także ekrany pozwalające użytkownikom na podawanie identyfikatora i hasła. Zastosujemy ponadto kilka technologii

pozwalających na korzystanie z unikalnego stanu sesji we wszystkich czynnościach wykonywanych przez użytkownika w dowolnym momencie.

#### ■ System rejestracji

System rejestracji będzie się składał zarówno z ekranów, jak i z resolverów, które zapewnią użytkownikom możliwość utworzenia unikalnego konta, którego z kolei będziemy używali do identyfikowania czynności wykonywanych przez użytkownika w aplikacji.

#### ■ Mechanizm resetowania hasła

Pozwoli użytkownikom, w razie pojawienia się takiej konieczności, na zresetowanie hasła w bezpieczny sposób.

#### ■ Strona profilu użytkownika

Będzie to ekran z udostępniający dodatkowe możliwości, prezentujący informacje o koncie użytkownika. Będą na nim wyświetlane takie informacje, jak adres e-mail użytkownika oraz jego identyfikator.

#### ■ Kategorie

Aplikacja ma zapewniać możliwość grupowania na podstawie kategorii, co pozwoli użytkownikom przeglądać wpisy wyłącznie z wybranych, interesujących ich kategorii, i zredukować niepożądany szum informacyjny.

#### ■ Powiadomienia przesyłane e-mailem

Aplikacja będzie dysponować systemem łączenia użytkowników przy wykorzystaniu poczty elektronicznej oraz powiadamiania ich o dodatkowych wymaganiach lub zdarzeniach związanych z aplikacją. Na przykład system ten może obejmować przysyłanie wiadomości weryfikującej, czyli sprawdzającej, czy podany podczas rejestracji adres e-mail jest prawidłowy i dostępny dla użytkownika.

W tym podrozdziale przedstawiłem listę możliwości związanych z uwierzytelnianiem oraz identyfikacją użytkowników, a także ich aktywności, które mamy zamiar zaimplementować. W następnym podrozdziale wyjaśnię, w jaki sposób zaimplementujemy **wątki**, czyli podstawowy sposób komunikacji w naszej aplikacji.

## Analiza zarządzania wątkami

Chcemy, by początkiem nowego wątku mógł być każdy wpis opublikowany w aplikacji; innymi słowy, dowolny wpis opublikowany w już istniejącym wątku może rozpocząć dyskusję i doprowadzić do powstania sekwencji odpowiedzi. A zatem, nasza aplikacja będzie musiała zapewniać użytkownikom możliwość rozpoczynania dyskusji poprzez opublikowanie początkowego wpisu. Taki wpis musi być widoczny dla wszystkich użytkowników, którzy będą mogli na niego odpowiadać. Każdy element wątku, włącznie z pierwszym wpisem, będzie powiązany z użytkownikiem, który go opublikował. Aby zapewnić te możliwości funkcjonalne, będziemy musieli zaimplementować:

### ■ Publikowanie i edycję początkowego wpisu (tematu) wątku

Te mechanizmy zapewnią możliwość przeglądania (przez dowolnego użytkownika) tematu wątku oraz jego edycję i dodawanie kolejnych wpisów przez jego autora. Użytkownicy będą mogli wyświetlić listę wszystkich swoich wpisów na stronie profilowej.

### ■ Odpowiadanie na początkowy wpis wątku

Te mechanizmy pozwolą twórcy tematu oraz innym użytkownikom odpowiadać na wpis początkowy, czyli dodawać do niego własne komentarze. Możliwe będzie także wyświetlenie wpisu początkowego oraz wszystkich powiązanych z nim odpowiedzi na jednym ekranie.

Aby w jak największym stopniu uprościć aplikację, użytkownicy nie będą mogli odpowiadać na konkretne odpowiedzi, a jedynie na początkowy wpis wątku. Będą jednak mogli cytować inne wpisy w swoich odpowiedziach.

W tym podrozdziale opisałem podstawowe możliwości aplikacji oraz ich cechy. To właśnie tworzenie nowych wątków oraz odpowiadanie na nie będzie stanowić podstawową możliwość funkcjonalną aplikacji, choć aby je nieco rozszerzyć, dodamy do aplikacji także kilka innych, powiązanych możliwości. W następnym podrozdziale przedstawię planowany sposób działania systemu punktacji wątków.

## Analiza systemu punktacji wątków

Użytkownicy powinni mieć możliwość oznaczania komentarzy, które chcą pozytywnie wyróżnić. Prezentowanie popularnych wpisów w aplikacji może sprawić, że użytkownicy będą się bardziej angażować w życie społeczności. W tym podrozdziale opiszę, w jaki sposób zapewnimy użytkownikom możliwość oznaczania wyróżniających się wpisów.

Aby pozwolić użytkownikom na oznaczanie wyróżniających się wpisów, zaimplementujemy w aplikacji następujące możliwości:

### ■ System punktacji

Zaimplementowany system punktacji pozwoli użytkownikom głosować na wątki i odpowiedzi, przy czym będzie można głosować zarówno za, jak i przeciw danemu wpisowi.

### ■ Prezentowanie liczby punktów

Aplikacja będzie wyświetlać informację o tym, ile razy dany wpis został wyświetlony.

### ■ Wyświetlanie liczby odpowiedzi

Aplikacja będzie wyświetlać liczbę odpowiedzi na dany wpis, co pozwoli użytkownikom określać, które tematy są popularne.

W tym podrozdziale przedstawiłem zamierzone działanie ważnej możliwości funkcjonalnej aplikacji, która pozwoli użytkownikom wyrażać swoją aprobatę lub dezaprobatę dla poszczególnych wpisów oraz określać, które tematy są najbardziej popularne. Zaimplementowany system punktacji zwiększy zaangażowanie użytkowników aplikacji.

## Podsumowanie

W tym rozdziale dokładnie opisałem aplikację, którą będziemy tworzyć w dalszej części książki; przedstawiłem listę jej możliwości funkcjonalnych oraz wyjaśniłem powody wyboru takiego, a nie innego typu aplikacji. Ponieważ mamy zbudować aplikację korzystającą z pełnego stosu technologicznego, jej kod będzie dość złożony, a jego zaimplementowanie może być sporym wyzwaniem. Nie zdziwię się, jeśli sam będziesz zaskoczony końcową wielkością i zakresem aplikacji. Niemniej jednak, kiedy już ją zakończymy, będzie to nowoczesna, złożona i kompletna aplikacja internetowa.

W następnym rozdziale zaczniemy pisać kliencką część aplikacji korzystającą z frameworka React. Nie będziemy w stanie zaimplementować jej w całości, gdyż nie dysponujemy jeszcze jej częścią serwerową. Pomimo to przygotujemy jej znaczną część, w tym wiele ekranów aplikacji.

# Tworzenie klienta Reacta na potrzeby aplikacji internetowego forum

Od początku książki przebyliśmy już długą drogę. W tym rozdziale zaczniemy pisać kod aplikacji, zaczynając do jej części klienckiej, którą stworzymy w oparciu o Reacta. Wykorzystamy całą wiedzę nabytą w poprzednich rozdziałach książki i zbudujemy aplikację Reacta wykorzystując API *hooków*. Skorzystamy przy tym z technik tworzenia responsywnych stron WWW, aby przygotować aplikację mobilną, której z powodzeniem będzie można używać zarówno na komputerach stacjonarnych, jak i urządzeniach mobilnych.

---

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś dysponować wiedzą z zakresu tworzenia aplikacji internetowych przy użyciu Reacta, Node, Expressa oraz GraphQL-a. Powinieneś także potrafić tworzyć arkusze stylów CSS. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial12* (*Chap12*).

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog o nazwie *rozdzial12*.

## Tworzenie wstępnej wersji aplikacji Reacta

W tym rozdziale zaczniemy tworzyć kliencką część naszego internetowego forum, czyli aplikację Reacta. Nie będziemy mogli zaimplementować jej w całości, gdyż do tego będziemy potrzebować możliwości funkcjonalnych zapewnianych przez część serwerową, takich jak: API GraphQL, uwierzytelnianie, zapisywanie wątków itd. Niemniej jednak, zaczniemy tu tworzyć podstawowe ekrany aplikacji, używając do tego Reacta i React Routera.

W tym podrozdziale zamieszczę bardzo dużo kodu. Zachęcam, żebyś czytając go, robił sobie często przerwy i odpoczniki. Ten kod będzie ewoluował i jeszcze wielokrotnie w dalszej części książki będziemy do niego wracać, refaktoryzować go i rozszerzać. Wprowadzane zmiany czasami będą miały na celu poprawienie możliwości wielokrotnego stosowania kodu, w innych sytuacjach będą one wynikać z chęci poprawienia projektu oraz czytelności kodu. Jeśli gdzieś utkniesz lub będziesz miał problemy z określeniem, co należy zrobić, zawsze możesz zajrzeć do kodów źródłowych przykładów dołączonych do książki. Nie mam jednak wątpliwości, że ten rozdział może być dla Ciebie jednym z najtrudniejszych, jakie do tej pory przeczytałeś w tej książce.

Nie będę tu zamieszczał każdego wiersza kodu, gdyż wiele z nich powtarzałoby się. Dlatego koniecznie pobierz kody źródłowe i otwórz je w edytorze.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- utworzeniem aplikacji Reacta i skonfigurowaniem jej oraz wszystkich jej zależności;
- przygotowaniem stylów i układu stron aplikacji;
- przygotowaniem kluczowych komponentów i możliwości aplikacji.

Zadbanie o to, by od samego początku z powodzeniem można było poprawnie skompilować i uruchomić cały kod źródłowy aplikacji, tak naprawdę nie ma żadnego korzystnego wpływu na proces Twojej nauki. Nie koncentruj się na próbach zapewnienia, by aplikację można było skompilować i uruchomić już od pierwszego razu, gdy tego spróbujesz. Zamiast tego eksperymentuj i wprowadzaj zmiany. Innymi słowy, podziel kod w taki sposób, by na początku nie udawało się go skompilować, a następnie próbuj rozwiązywać problemy, do których doprowadziłeś. To jedyny sposób na zapewnienie, że naprawdę będziesz rozumiał to, co robisz.



Zacznijmy od stworzenia bazowego projektu aplikacji Reacta, używając do tego celu narzędzia `create-react-app`<sup>1</sup>. Następnie dodamy do niego `Reduxa` i `React Router`.

1. W panelu terminala `VSCode` przejdź do katalogu *rozdzial12* i wykonaj następujące polecenie:

```
npx create-react-app super-forum-client --template typescript
```

2. Następnie przejdź do katalogu *super-forum-client* i wykonaj polecenie `start`, by upewnić się, czy aplikacja działa:

```
npm start
```

3. Teraz zainstaluj `Reduxa` i `React Router`:

```
npm i redux react-redux @types/redux @types/react-redux react-router-dom  
↳@types/react-router-dom
```

Jeśli kiedykolwiek będziesz mieć problemy z pakietami NPM, objawiające się tym, że aplikacja nie chce się prawidłowo uruchomić, to zacznij od usunięcia pliku *package-lock.json* i katalogu *node\_modules*. Następnie zainstaluj ponownie wszystkie pakiety, wykonując w tym celu polecenie `npm install`.

W ten sposób zainstalowałeś wszystkie pakiety niezbędne do rozpoczęcia prac nad aplikacją kliencką. Jednak zanim zaczniemy pisać jej kod, musimy zastanowić się nad jej układem. Chcemy, aby aplikacja działała zarówno na urządzeniach przenośnych, jak i komputerach stacjonarnych. Dzięki temu uzyskamy jedną aplikację, która będzie mogła być używana w telefonach, komputerach stacjonarnych i laptopach.

Taki cel można zrealizować na wiele różnych sposobów. Jednym z nich jest skorzystanie z biblioteki CSS, takiej jak *Bootstrap*, lub frameworka do tworzenia interfejsu użytkownika, takiego jak *ionic*, które z powodzeniem mogą nam pomóc w określeniu postaci oraz układu ekranów aplikacji. Te rozwiązania są wspierane i działają bardzo dobrze, lecz jednocześnie ukrywają wiele szczegółów związanych z określaniem wyglądu i układu stron WWW.

<sup>1</sup> W czasie, kiedy było przygotowywane polskie wydanie niniejszej książki, występowały problemy we współpracy edytora *Slate.js* używanego w aplikacji z wersją *Reacta* instalowaną automatycznie przez narzędzie `create-react-app`. Najprostszym sposobem, by uniknąć problemów z integracją aplikacji rozwijanej w tym rozdziale z edytorem, będzie zrezygnowanie z tworzenia nowej aplikacji przy użyciu `create-react-app` i skorzystanie z kodów źródłowych dostępnych w przykładach dołączonych do książki. W takim przypadku wystarczy skopiować te kody do katalogu *rozdzial12* i zainstalować *Reacta* wraz ze wszystkimi zależnościami przy użyciu polecenia `npm install` (lub `yarn install`). Następnie należy usunąć z katalogu *src* wszystkie podkatalogi z komponentami (będą one tworzone w trakcie prac nad aplikacją opisanych w rozdziale) i przywrócić plik *index.tsx* do postaci początkowej, wzorując się na aplikacjach *Reacta* opisanych we wcześniejszych rozdziałach. Musimy pamiętać, że zarówno sam *React*, jak i popularne biblioteki używane w aplikacjach *React* są aktywnie rozwijane, więc może się okazać, że kiedy ta książka trafi do rąk Czytelnika, żadne utrudnienia już nie będą występować i będzie można rozwijać aplikację w pełni zgodnie z opisem zamieszczonym w książce. Jednak, gdyby problemy dalej występowały, to warto pamiętać o tej możliwości zainstalowania *Reacta* i zależności w wersjach używanych pierwotnie przez autora — gwarantując one, że żadne kłopoty nie pojawią się — *przyp. tłum.*

Co więcej, korzystając z takich frameworków, narażamy się czasami na utratę kontroli, a w końcowym efekcie może się okazać, że nasza witryna będzie wyglądać podobnie do innych stworzonych przy użyciu tego samego frameworka.

## CSS Grid

W naszej aplikacji zastosujemy zasady projektowania responsywnych stron WWW. Ich celem jest zapewnienie aplikacji możliwości dostosowywania się do ekranów o różnych wielkościach i wymiarach. Dostępne technologie internetowe pozwalają zrealizować te cele na kilka różnych sposobów. Jednym z nich jest skorzystanie z technologii określanej jako **siatka CSS** — *CSS Grid*. Korzystając z niej, można określać strukturę ekranów aplikacji tak, by optymalnie wykorzystywała ona dostępny obszar ekranu, a jednocześnie by automatycznie dostosowywała się do działania na urządzeniach mobilnych. I właśnie ze względu na te możliwości do tworzenia ekranów naszej aplikacji zastosujemy CSS Grid oraz kilka innych technologii internetowych.

CSS Grid udostępnia znaczną większość możliwości zapewnianych przez takie frameworki, jak Bootstrap. Jednak CSS Grid jest częścią standardu CSS, a nie elementem jakiejś biblioteki czy frameworka. Dlatego też stosując to rozwiązanie będziemy mieć pewność, że nasz układ będzie działał zawsze i nie zdarzy się, że wykorzystywane w nim rozwiązanie kiedyś przestały być obsługiwane.

Czym zatem jest CSS Grid? Otóż jest to metoda określania układu stron WWW stanowiąca jeden z elementów standardu CSS, pozwalająca na tworzenie elastycznych układów składających się z wierszy i kolumn. Została ona opracowana w celu zastąpienia stosowania tablic jako metody określania układów stron. CSS Grid ma ogromne możliwości i pozwala na uzyskiwanie tych samych efektów na kilka różnych sposobów. Inaczej mówiąc, ja przedstawię tu swoje rozwiązanie, lecz Ty później będziesz mógł dokładniej przestudiować dostępne możliwości, jeśli uznasz, że to do czegoś ci się przyda. A zatem, zacznijmy używać CSS Grid.

1. W pierwszej kolejności przejdź do katalogu projektu, otwórz plik *App.tsx* i usuń całą dotychczasową zawartość obiektu App; następnie zmień go w sposób pokazany na poniższym przykładzie:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <nav className="navigation">Nawigacja</nav>
      <div className="sidebar">Pasek boczny</div>
      <div className="leftmenu">Menu z lewej</div>
      <main className="content">Część główna</main>
      <div className="rightmenu">Menu z prawej</div>
    </div>
  );
}

export default App;
```

Jak widać, usunęliśmy całą poprzednią zawartość komponentu i zastąpiliśmy ją elementami pełniącymi funkcję tymczasowych zamienników. Oczywiście, podczas dalszych prac zastąpimy je odpowiednimi komponentami, lecz póki co chcemy się skoncentrować na zapewnieniu odpowiedniego działania naszej siatki CSS.

2. Teraz zmień zawartość pliku *App.css* tak, jak pokazałem na poniższym przykładzie:

```
:root {
  --min-screen-height: 1000px;
}
```

W tej pierwszej regule zastosowaliśmy pseudoklasę `:root` — użyjemy jej jako pojemnika do określenia zmiennych CSS, które będziemy stosować do określania naszego tematu graficznego. Aby ułatwić sobie tworzenie stylów i tematu graficznego, zamiast używać wartości podawanych na stałe, zastosujemy zmienne CSS.

Przekonasz się, że wraz z rozwojem aplikacji, w tym miejscu będzie się pojawiać coraz więcej zmiennych.

```
.App {
  margin: 0 auto
```

Poniższe ustawienia marginesów zapewnią wyśrodkowanie układu:

```
max-width: 1200px;
display: grid;
grid-template-columns: 0.7fr 0.9fr 1.5fr 0.9fr;
grid-template-rows: 2.75rem 3fr;
grid-template-areas:
  "nav nav nav nav"
  "sidebar leftmenu content rightmenu";
gap: 0.75rem 0.4rem;
}
```

Poniżej zamieściłem zestawienie najważniejszych atrybutów związanych z CSS Grid:

- `display` — ta reguła informuje, że element będzie typu grid.
- `grid-template-columns` — ten atrybut określa względne szerokości poszczególnych kolumn układu. Układ naszej aplikacji będzie się składał z czterech kolumn. Wartość z jednostką `fr` określa, jaka część dostępnej szerokości ma zostać przydzielona kolumnie. Na przykład nasz układ składa się z czterech kolumn, gdybyśmy zatem chcieli każdej z nich przypisać dokładnie taką samą szerokość, to należałoby użyć wartości `1fr`. Jednak w naszym przypadku każda kolumna będzie mieć inną szerokość, dlatego też w tym atrybucie podane zostały cztery różne wartości. Szerokości kolumn mogą być zapisywane jako wartości bezwzględne, takie jak `100px` lub `2rem`, wartości procentowe, takie jak `20%`, bądź wartości niejawne, takie jak `.25fr`.
- `grid-template-rows` — ten atrybut określa liczbę i wielkość wierszy. Można w nim podawać takie same wartości jak podczas określania szerokości kolumn.

- `grid-template-areas` — każda siatka CSS może zawierać nazwane sekcje, nazywane obszarami (ang. *areas*). Jak pokazałem w przykładzie, nazwy poszczególnych obszarów zapisuje się wierszami i kolumnami, umieszczając w tych miejscach, w których ma się znaleźć dany element. A zatem, zapis "nav nav nav nav" oznacza wszystkie cztery kolumny pierwszego wiersza układu, natomiast zapis "sidebar leftmenu content rightmenu" reprezentuje cztery kolumny drugiego wiersza.
  - `gap` — ten atrybut pozwala na dodawanie odstępów pomiędzy wierszami i kolumnami. Pierwsza wartość określa odstęp między wierszami, a druga pomiędzy kolumnami.
3. Skoro już wyjaśniliśmy podstawowe możliwości technologii CSS Grid, przyjrzyjmy się stylom określającym poszczególne sekcje naszej siatki; przedstawiłem je na poniższym przykładzie:

```
.navigation {
  grid-area: nav;
}
.sidebar {
  min-height: var(--min-screen-height);
  grid-area: sidebar;
  background-color: aliceblue;
}
.leftmenu {
  grid-area: leftmenu;
  background-color: skyblue;
}
.content {
  min-height: var(--min-screen-height);
  grid-area: content;
  background-color: blanchedalmond;
}
.rightmenu {
  grid-area: rightmenu;
  background-color: coral;
}
```

Jak widać, wszystkie te style zawierają atrybut `grid-area`, określający, do którego obszaru siatki należy dany element. Obszar `nav` będzie zawierał elementy nawigacyjne. W obszarze `sidebar` będzie prezentowane menu z opcjami związanymi z użytkownikiem, przy czym obszar ten będzie widoczny wyłącznie na komputerach stacjonarnych i laptopach; na urządzeniach mobilnych element ten będzie niewidoczny. Obszaru `leftmenu` będziemy używali do prezentowania listy kategorii wątków. Obszar `content` będzie zawierał główną listę wątków przefiltrowaną na podstawie kategorii. I w końcu obszar `rightmenu` posłuży nam do wyświetlania listy popularnych czy też w jakiś inny sposób istotnych wątków.

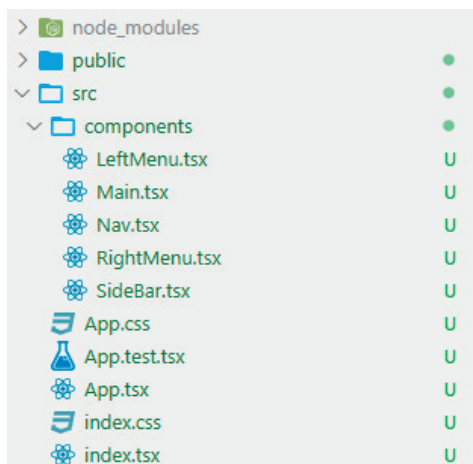
Widoczne w ostatnim przykładzie dziwne kolory określone we właściwościach `background-color` zastosowałem jedynie tymczasowo, aby ułatwić rozróżnienie poszczególnych obszarów siatki.

Przygotowaliśmy już podstawowy układ aplikacji, który będzie działać na komputerach stacjonarnych i laptopach. Ale w jaki sposób zapewnić, że będzie się on automatycznie dostosowywał do mniejszych urządzeń, takich jak telefony lub tablety? Otóż w skład standardu CSS wchodzi technologia określana jako **zapytania medialne** (ang. *Media Queries*), które okazują się przydatne właśnie w takich sytuacjach.

Nasza aplikacja będzie budowana dynamicznie przez Reacta i sterowana zmianami stanu. Oznacza to, że niektóre komponenty ekranu mogą nie być wyświetlane, jeśli nie będą potrzebne lub jeśli na mniejszych urządzeniach nie będzie można ich wyświetlić. A zatem, choć moglibyśmy zastosować zapytania medialne do ukrywania komponentów w razie wykrycia, że aplikacja działa na urządzeniu z mniejszym ekranem, to jednak zmuszanie Reacta do renderowania czegoś, co nie będzie widoczne lub bezpośrednio używane przez użytkownika, to jednak takie rozwiązanie oznaczałoby nieefektywne wykorzystanie zasobów.

Zamiast tego przyjrzyjmy się, w jaki sposób możemy poradzić sobie z tym zagadnieniem w kodzie, używając obsługi zdarzeń oraz *hooków* Reacta:

1. Pierwszą rzeczą, jaką zrobimy, będzie przekształcenie głównych elementów strony na komponenty Reacta. Utwórz zatem w katalogu *src* nowy podkatalog o nazwie *components*.
2. Następnie wewnątrz katalogu *components* utwórz komponent dla każdego z elementów HTML umieszczonych wewnątrz głównego elementu *div* komponentu *App*. Po utworzeniu tych plików zawartość katalogu *components* powinna wyglądać jak na rysunku 12.1.

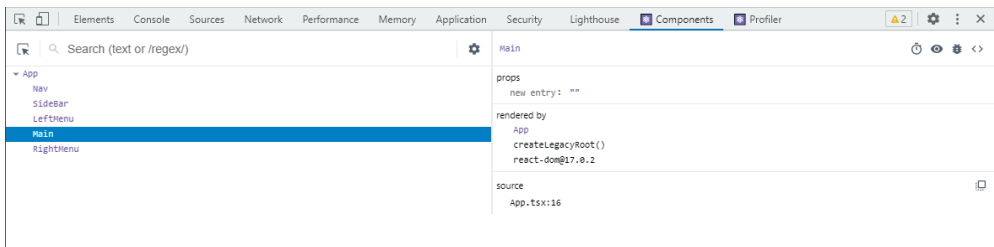


Rysunek 12.1. Pliki komponentów przygotowane do modyfikacji komponentu *App.tsx*

Aby zaoszczędzić nieco miejsca, nie będę tu przedstawiał kodu każdego z tych nowych plików, gdyż są one do siebie bardzo podobne. Ograniczę się jedynie do kodu komponentu *Main* (oczywiście, w przykładach dołączonych do książki znajdziesz kompletny kod każdego z tych komponentów):

```
import React from "react";
const Main = () => {
  return <main className="content">Część główna</main>;
};
export default Main;
```

Jak widać, przenieśliśmy jedynie fragment kodu z pliku *App.tsx* do pliku komponentu *Main.tsx*. W analogiczny sposób musisz utworzyć pozostałe komponenty: *Nav*, *SideBar*, *LeftMenu* oraz *RightMenu*. Na rysunku 12.2 przedstawiłem postać aktualnej hierarchii komponentów, wyświetloną w narzędziach dla programistów przeglądarki Chrome. Narzędzia te oraz sposoby korzystania z nich przedstawiłem w rozdziale 6., pt. „Przygotowywanie projektu za pomocą create-react-app i testowanie go przy użyciu Jest”.



Rysunek 12.2. Widok hierarchii komponentów

Zwróć uwagę na to, że na rysunku 12.2 są widoczne komponenty *Nav*, *SideBar*, *LeftMenu*, *Main* oraz *RightMenu*. Każdy z nich reprezentuje jeden obszar dostępny na głównym poziomie witryny. Pamiętaj, że później, kiedy rozbudujemy aplikację, będzie ona miała znacznie więcej ekranów niż jest dostępnych obecnie.

Taki podział na komponenty musieliśmy przeprowadzić wcześniej czy później, gdyż implementujemy aplikację Reacta. Ale w jaki sposób może nam to pomóc w dążeniu do zagwarantowania responsywności naszej aplikacji i zapewnieniu jej możliwości automatycznego dostosowywania się do ekranów o różnej wielkości i rozdzielczości? Otóż wyodrębniając poszczególne obszary siatki CSS i implementując je w formie odrębnych komponentów, zyskujemy możliwość zastosowania w nich *hooków* i wykorzystania ich do pobierania informacji o wielkości ekranu. Dzięki temu, jeśli komponentu nie będzie można wyświetlić ze względu na zbyt małą wielkość ekranu, będziemy mogli uniemożliwić wyrenderowanie komponentu lub wyrenderować go w inny sposób.

Aby taki responsywny system mógł działać, musimy wprowadzić do naszego rozwiązania dwie modyfikacje. Pierwszą będzie dodanie nowych stylów CSS korzystających z zapytań medialnych, które pozwolą określić inny układ siatki w przypadku wykrycia, że aplikacja jest używana na urządzeniu z mniejszym ekranem. Z kolei druga modyfikacja polega na wyposażeniu komponentów w możliwość uzyskiwania informacji o wielkości ekranu urządzenia, na którym aplikacja działa, i podejmowania na tej podstawie decyzji o tym, czy dany komponent należy wyrenderować normalnie, całkowicie go ukryć, czy też wyrenderować w jakiejś innej postaci. Zobaczmy zatem, jak ma wyglądać kod zapewniający te możliwości.

W pierwszej kolejności przygotujemy zapytania medialne dla urządzeń mobilnych. Otwórz plik *App.css* i na jego końcu zapisz poniższy fragment kodu CSS:

```
@media screen and (orientation: portrait) and (max-width:768px) {
  .App {
    grid-template-columns: 1fr;
    grid-template-areas:
      "nav"
      "content";
  }
}
```

W tym fragmencie arkusza stylów przesłaniamy początkową definicję klasy *App*, która będzie stosowana, gdy urządzenie będzie używane w orientacji pionowej (*orientation: portrait*), a rozdzielczość w poziomie nie będzie przekraczać 768 pikseli. Kiedy wprowadzisz te zmiany i uruchomisz aplikację w trybie mobilnym, symulując rozdzielczość telefonu iPhone X, to aplikacja będzie wyglądać tak, jak na rysunku 12.3.



**Rysunek 12.3.** Aplikacja wyświetlona w przeglądarce Chrome w trybie mobilnym

Z prawej strony aplikacji jest widoczny biały pas, gdyż elementy istniejące w początkowym układzie przeznaczonym do użycia na komputerach stacjonarnych wciąż są renderowane. Już niebawem rozwiążemy ten problem. A teraz zajmijmy się stworzeniem *hooka*, który pomoże nam zaimplementować renderowanie komponentów zależne od wielkości ekranu:

1. W katalogu `src` utwórz nowy podkatalog `hooks`. W tym nowym podkatalogu utwórz plik `useWindowDimensions.ts`. Zwróć uwagę na to, że nie jest to komponent Reacta, gdyż plik ma rozszerzenie `.ts`, a nie `.tsx`. Teraz skopiuj do pliku kod źródłowy z przykładów dołączonych do książki. Skoro plik już jest gotowy, przeanalizujemy jego zawartość.

W pierwszej kolejności tworzymy interfejs o nazwie `WindowDimension`, który posłuży nam do określenia typu wyniku zwracanego przez *hook* — w naszym przypadku będą to wymiary pobrane z obiektu `window` reprezentującego okno przeglądarki.

Następnie, w wierszu 8., tworzymy *hook* `useWindowDimensions`. W kolejnym wierszu tworzymy obiekt stanu o nazwie `dimension` i zapisujemy w jego właściwościach `width` i `height` wartości 0.

2. Kolejnym elementem kodu jest funkcja `handleResize`, która będzie używać metody stanu (`setDimension`) do zapisywania wymiarów okna. Wartości te odczytamy z obiektu `window` reprezentującego okno przeglądarki.
3. W końcu, w wierszu 21., używamy funkcji `useEffect`, by obsługiwać generowane przez przeglądarkę zdarzenia `resize`. Zwróć uwagę na zastosowaną w wywołaniu funkcji pustą tablicę, `[]`. Jej użycie oznacza, że funkcja zostanie wykonana tylko raz, gdy zostanie wczytana po raz pierwszy. Pamiętaj także, że jeśli dodajemy procedurę obsługi zdarzeń, to musimy zwrócić funkcję, która tę procedurę usunie (zapobiega to wyciekom pamięci oraz dodawaniu nadmiarowych, powtarzających się procedur obsługi zdarzeń).
4. Teraz musimy zaktualizować komponenty `SideBar`, `LeftMenu` oraz `RightMenu` tak, by korzystały z naszego *hooka* i aby nie były renderowane w przypadku, gdy szerokość ekranu będzie mniejsza od 768 pikseli (czyli granicy zastosowanej w zapytaniu medialnym). Kod zapewniający zastosowanie *hooka* jest taki sam we wszystkich trzech komponentach, dlatego też poniżej zamieszczę jedynie jego wersję z komponentu `SideBar`. Pozostałe dwa komponenty powinieneś zmodyfikować samodzielnie, zgodnie z podanym przykładem:

```
import React from "react";
import { useWindowDimensions } from "../hooks/useWindowDimensions";

const SideBar = () => {
  const { width } = useWindowDimensions();
  if (width <= 768) {
    return null;
  }
  return <div className="sidebar">Pasek boczny</div>;
};
export default SideBar;
```

Jak widać, używamy *hooka* `useWindowDimensions`, by pobrać szerokość okna przeglądarki (`width`). Następnie sprawdzamy, czy szerokość ta jest mniejsza lub równa 768; jeśli jest, to zwracamy `null`, a w przeciwnym razie zwracamy normalny kod JSX. W pozostałych dwóch komponentach musisz zastosować dokładnie ten sam kod, korzystający z *hooka* `useWindowDimensions`.



Jeśli teraz uruchomisz aplikację, przekonasz się, że biały obszar z prawej strony zniknął, a trzech komponentów w ogóle nie ma w kodzie HTML. Zwróć uwagę na to, że w celu zaoszczędzenia czasu i wysiłku nasza aplikacja będzie obsługiwać jedynie komputery stacjonarne i laptopy oraz tryb pionowy dla telefonu iPhone X. Obsługa każdej możliwej konfiguracji sprzętowej znacznie wykracza poza ramy tematyczne tej książki. Bardzo dobre omówienie zagadnień związanych z obsługą urządzeń z ekranami o różnych rozdzielczościach można znaleźć w witrynie Google, na stronie <https://developers.google.com/web/fundamentals/codelabs/your-first-multi-screen-site>.

Zanim przejdziemy do kolejnych zagadnień, musimy zadbać o odpowiednią konfigurację aplikacji klienckiej, a konkretnie o jej aspekty związane z Reduxem oraz Routerem Reacta.

5. Zaktualizuj plik *index.tsx*, dołączając do niego Reduxa oraz React Router. Informacje na temat tych bibliotek przedstawiłem w rozdziale 7., pt. „Redux i React Router”. Jak zwykle, gdybyś miał jakieś problemy, możesz zajrzeć do kodu źródłowego dołączonego do książki.
6. I w końcu, w katalogu *src* utwórz podkatalog *store*; dodamy do niego pliki związane z Reduxem. Utwórz w tym katalogu pliki *AppState.ts* oraz *configureStore.ts* i skopiuj do nich kod z przykładów dołączonych do książki. Jeszcze nie jesteśmy gotowi, by zająć się reduktorem *UserProfileReducer*, więc póki co zostawimy go w spokoju. Nie będziemy także używali oprogramowania pośredniego Reduxa, które także przedstawiłem w rozdziale 7., pt. „Redux i React Router”.

Zanim zaczniemy tworzyć komponenty, dodajmy jeszcze do aplikacji jedną z najnowszych możliwości Reacta, która pozwoli nadać naszej aplikacji nieco profesjonalizmu.

## Granice błędów

Granice błędów (ang. *error boundaries*) to rozwiązanie, które można by porównać z obsługą wyjątków w komponentach Reacta. W dużych aplikacjach czasami nie ma możliwości zapobieżenia wszelkim możliwym błędom, jakie mogą się pojawić. Dlatego, stosując granice błędów w komponentach, możemy „przechwytywać” takie nieprzewidziane błędy i poprawiać wrażenia użytkowników z korzystania z aplikacji. W razie wystąpienia błędu będziemy wyświetlać przygotowany wcześniej ekran ze stosowną informacją, zamiast jakiejś tajemniczo wyglądającej strony z technicznym komunikatem o błędzie. A zatem, zabierzmy się do pracy:

1. W pierwszej kolejności stwórz plik granicy błędu. W katalogu *components* utwórz plik *ErrorBoundary.tsx* i zapisz w nim kod z analogicznego pliku z przykładów dołączonych do książki. Zwróć uwagę na to, że klasa *ErrorBoundary* jest komponentem starego typu, czyli klasowym, co jest konieczne, gdyż do przechwytywania błędów musimy skorzystać z metod cyklu życia *getDerivedStateFromError* oraz *componentDidCatch*. Twórcy Reacta planują w przyszłości dodać *hooks* zapewniające analogiczne możliwości.

Zwróć także uwagę na to, że w katalogu *components* znajduje się plik arkusza stylów o takiej samej nazwie: *ErrorBoundary.css*. Umieszczone w nim style są jednak bardzo proste, więc nie będę ich tu przedstawiać; możesz je znaleźć w kodach źródłowych przykładów dołączonych do książki.

W pierwszej kolejności stworzymy typ na potrzeby właściwości *props* komponentu *ErrorBoundary* i nadajemy mu nazwę *ErrorBoundaryProps*.

Następnie definiujemy kolejny typ, na potrzeby lokalnego stanu komponentu, i nadajemy mu nazwę *ErrorBoundaryState*. Na początku klasy *ErrorBoundary* znajduje się fragment typowego, powtarzalnego kodu, a konkretnie konstruktor określający początkową wartość stanu. Bezpośrednio poniżej konstruktora definiujemy funkcję *getDerivedStateFromError*, która nakazuje wyświetlenie innego interfejsu użytkownika w razie wystąpienia błędu, czyli kiedy zmienna stanu *hasError* przyjmie wartość *true*.

W wierszu 31., czyli w funkcji *componentDidCatch*, nasz komponent uzyskuje informację o tym, że wydarzył się jakiś błąd, i zapisuje wartość *true* w zmiennej stanu *hasError*. W razie konieczności w tej funkcji możemy także wykonywać jakiś nasz kod, który na przykład będzie zapisywał w dzienniku komunikat błędu lub przysyłał powiadomienie do działu wsparcia.

W końcu, jeśli zmienna stanu *hasError* ma wartość *true*, wyświetlamy stosowny komunikat, dzięki czemu użytkownik aplikacji nie będzie musiał oglądać komunikatów o charakterze technicznym, które mogą być dla niego niezrozumiałe. Oczywiście, jeśli uznasz to za celowe, możesz samodzielnie określić treść wyświetlanych komunikatów o błędach.

Komponenty granic błędów nie wychwytyją błędów zgłaszanych wewnątrz procedur obsługi zdarzeń, w kodzie asynchronicznym, ani w kodzie wykonywanym po stronie serwera, jak również błędów zgłaszanych w kodzie samych komponentów granic błędów. Takie błędy będziemy musieli obsługiwać samodzielnie, używając do tego instrukcji *try* i *catch*.

2. Teraz przetestujemy granicę błędów, zgłaszając błąd w jednym z naszych komponentów. Zaktualizuj plik *Main.tsx* tak, by funkcja *Main* była zgodna z poniższym przykładem:

```
const Main = () => {
  const test = true;
  if (test) throw new Error('Błąd w komponencie Main!');
  else {
    return <main className="content">Część główna</main>;
  }
};
```

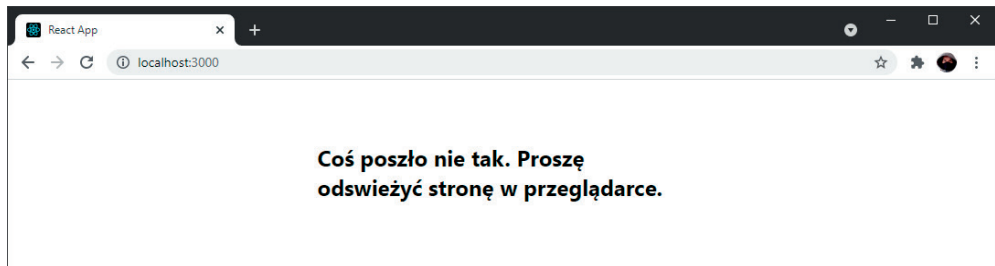
Jak widać, w tym kodzie celowo zgłaszamy wyjątek typu *Error*.

3. Kolejne zmiany musisz wprowadzić w pliku *index.tsx*. Dodaj do niego instrukcję importującą nowy komponent granicy błędów *ErrorBoundary.tsx*, a następnie zmień odwołanie do komponentu *App* tak, jak pokazałem na poniższym przykładzie:

```
ReactDOM.render(
  <Provider store={configureStore()}>
    <BrowserRouter>
      <ErrorBoundary>{[<App key="App" />]}</ErrorBoundary>
    </BrowserRouter>
  </Provider>,
  document.getElementById('root')
);
```

4. Spróbuj teraz uruchomić aplikację. W przeglądarce zobaczysz dokładnie taką stronę, jakiej staraliśmy się uniknąć. Ale dlaczego? Otóż wynika to z faktu, że aplikacja działa aktualnie w trybie do prowadzenia prac programistycznych, a w tym trybie React celowo wyświetla wszystkie komunikaty błędów. Gdyby aplikacja działała w trybie produkcyjnym, uruchamianym na przykład przy użyciu polecenia `npm run build`, to zostałby wyświetlony komunikat określony w komponencie granicy błędu.

Niemniej jednak nawet w trybie do prowadzenia prac programistycznych można wyświetlić komponent granicy błędów — wystarczy kliknąć przycisk **×** wyświetlony w prawym górnym rogu strony z informacjami o bledach. Kiedy to zrobisz, zobaczysz stronę taką jak ta przedstawiona na rysunku 12.4.



**Rysunek 12.4.** Komunikat granicy błędów

Jak widać, teraz zostanie wyświetlony komunikat określony przez nas w komponencie granicy błędów. Oczywiście, o ile tylko uznasz to za stosowne, będziesz mógł nadać mu dowolną postać i wygląd; aby nie marnować czasu, zostawię go jednak w dotychczasowej postaci.

## Warstwa usługi danych

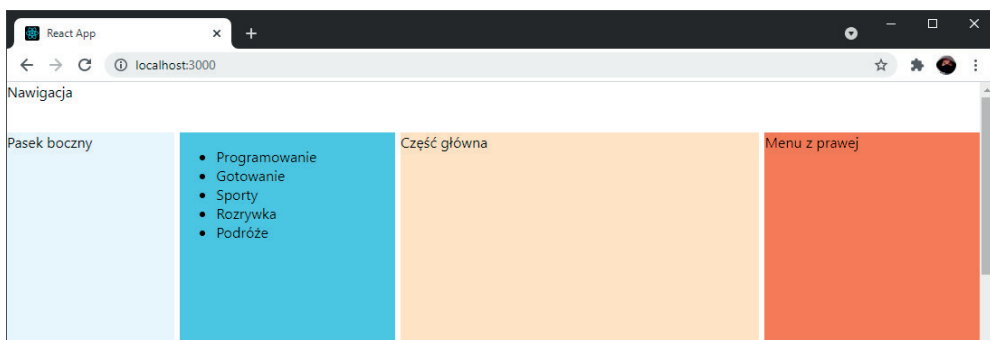
W naszej aplikacji będziemy wykonywać odwołania do API GraphQL-a lub API usług internetowych, jak również odbierać i obsługiwać odwołania sieciowe. Jednak póki co jeszcze żadna z tych usług serwerowych nie jest gotowa. Na obecnym etapie prac przygotowujemy jedynie plik zawierający fikcyjny kod obsługi żądań sieciowych, który posłuży nam do symulowania prawdziwego serwera. Kiedy ten faktyczny serwer zaimplementujemy, będziemy mogli usunąć ten tymczasowy kod.

1. Zacznij od utworzenia w katalogu `src` podkatalogu `services`, a w nim pliku `DataService.ts`. Ponieważ już niebawem tego kodu pozbędziemy się, nie będę go tu przedstawiał; po prostu skopiuj go z analogicznego pliku, który znajdziesz

w przykładach dołączonych do książki. Zwróć uwagę na używane w tym pliku odwołania do typów modeli (z katalogu *model*) — oznaczają one, że będziesz musiał skopiować z przykładów także pliki zawierające definicje tych typów; opiszę je dokładniej w dalszej części rozdziału.

2. Skoro już zapewniliśmy sobie możliwość pobierania danych, możemy skorzystać z niej w komponencie *LeftMenu*. Jednak ponieważ używamy języka TypeScript, najpierw będziemy musieli stworzyć typ, *Category*. A zatem, w katalogu *src* utwórz podkatalog *model*, a w nim plik *Category.ts*. Następnie skopiuj do niego kod z analogicznego pliku z przykładów dołączonych do książki.
3. Teraz zaktualizuj plik *LeftMenu.tsx*: zacznij od zaimportowania typu *Category* oraz pliku *LeftMenu.css*. Będą nam one potrzebne w dalszej części kodu.
4. Następnie, w wierszu 9., utwórz nowy obiekt stanu o nazwie *categories*, który będzie zawierał listę kategorii. Zanim w komponencie zostaną wczytane i wyświetlone kategorie, musimy najpierw wyświetlić w nim jakiś domyślny tekst; będzie to Menu z 1ewej.
5. Teraz w wierszu 13. dodaj wywołanie funkcji *useEffect*; w przekazanym do niej kodzie wywołujemy funkcję *getCategories*, aby pobrać dostępne kategorie. Pobrane kategorie są następnie przekształcane na kod JSX przy użyciu standardowej funkcji *map* języka ES6.
6. I w końcu, w zwracanym przez komponent kodzie JSX umieszczamy obiekt stanu *categories*, czyli wyświetlamy listę kategorii w interfejsie użytkownika aplikacji.

Jeśli teraz odświeżysz przeglądarkę, przekonasz się, że po 2 sekundach zostanie wyświetlona lista kategorii, taka jak ta przedstawiona na rysunku 12.5. To opóźnienie jest efektem zastosowania liczników czasu w naszej tymczasowej implementacji usługi *DataService*.



**Rysunek 12.5.** Wczytane i wyświetlone kategorie

Jeszcze raz powtórzę, że aktualną implementację *DataService* usuniemy po zaimplementowaniu faktycznych usług zapewniających dostęp do danych.

## Menu nawigacyjne

Skoro mamy już gotową podstawową konfigurację aplikacji oraz jej układ, możemy zająć się menu, które będzie prezentowane w komponencie `SideBar`. Jego interesującym aspektem będzie to, że jego elementy będą prezentowane zarówno w formie menu w komponencie, jak i w formie modalnej listy, którą będziemy stosować na urządzeniach mobilnych. Dzięki takiemu rozwiązaniu będziemy musieli napisać mniej kodu, niż gdybyśmy używali dwóch odrębnych komponentów do obsługi obu rodzajów prezentacji.

Aby utworzyć komponent `SideBar` z odpowiednią listą odnośników, musimy wiedzieć, czy użytkownik jest zalogowany, czy nie. Jeśli użytkownik nie będzie zalogowany, to w menu będziemy wyświetlać odnośniki do zalogowania oraz do rejestracji. Z kolei dla zalogowanego użytkownika będziemy wyświetlać odnośniki do wylogowania oraz do strony profilu użytkownika (`UserProfile`). Ekran profilu użytkownika będzie prezentować ustawienia użytkownika, jak również listę opublikowanych przez niego wpisów. Ponieważ informacje o stanie zalogowania użytkownika będą współużytkowane w obrębie całej aplikacji, umieścimy je w magazynie `Reduxa`.

1. Do określenia, czy użytkownik jest zalogowany, czy nie, wykorzystamy istnienie obiektu `UserProfile` bądź jego brak. W pierwszej kolejności dodamy nowy reduktor do naszego aktualnego, pustego zbioru reduktorów. A zatem, w katalogu `store` utwórz podkatalog `user`. W tym katalogu utwórz plik `Reducer.ts` i skopiuj do niego kod z analogicznego pliku dostępnego w przykładach dołączonych do książki.
2. Pierwszym elementem pliku jest stała `UserProfileSetType`, określająca typ akcji; dzięki niej będziemy mogli odróżnić ten reduktor, `UserProfileReducer`, od innych.
3. Następnie definiujemy typ zawartości, `UserProfilePayload`. Określa on postać danych, które będą dostępne w akcji podczas jej obsługi.
4. Kolejnym elementem pliku jest interfejs `UserProfileAction`; będzie on służył do odróżniania akcji związanej z profilem użytkownika od akcji wszelkich innych typów.
5. I w końcu ostatnim elementem pliku jest sam reduktor, `UserProfileReducer`, który wykonuje filtrowanie danych na podstawie typu akcji, `UserProfileSetType`. Jak już wspominałem wcześniej, zagadnienia związane ze stosowaniem `Reduxa` zostały opisane w rozdziale 7., pt. „`Redux` i `React Router`”.
6. Aby ułatwić sobie określenie postaci komponentów, zastosujemy ikony, które poprawią ich wygląd. W tym celu zainstalujemy `Font Awesome`, gdyż jest dostępny bezpłatnie i udostępnia atrakcyjny zestaw stylów i ikon, które są bardzo popularne wśród twórców stron WWW. W tym celu wykonaj następujące polecenie:

```
npm i @fortawesome/fontawesome-svg-core @fortawesome/free-solid-svg-icons
➔@fortawesome/react-fontawesome
```

7. Teraz, kiedy już zainstalowałeś zestaw ikon, utwórz w katalogu `src/components` nowy podkatalog, o nazwie `sidebar`, i przenieś do niego plik `SideBar.tsx`. W tym

samym katalogu *sidebar* utwórz kolejny plik, o nazwie *SideBarMenu.tsx*, i zapisz w nim kod przedstawiony poniżej; nie zapomnij dodać do niego wszystkich niezbędnych instrukcji importu:

```
const SideBarMenus = () => {
  const user = useSelector((state: AppState) => state.user);
  const dispatch = useDispatch();
  Aby skorzystać z możliwości Reduxa, użyjemy hooków useSelector oraz
  ↪useDispatch:
  useEffect(() => {
    dispatch({
      type: UserProfileSetType,
      payload: {
        id: 1,
        userName: "użytkownikTestowy",
      },
    });
  }, [dispatch]);
```

W kolejnym fragmencie kodu użyjemy *hooka* `useEffect` aby pobrać, przekazać i zaktualizować obiekt `UserProfile`. Zwróć uwagę na to, że póki co zawartość tego obiektu jest określona na stałe, jednak później będziemy go pobierać z serwera GraphQL, kiedy już go uruchomimy:

```
return (
  <React.Fragment>
    <ul>
      <FontAwesomeIcon icon={faUser} />
      <span className="menu-name">{user?.userName}</span>
    </ul>
  </React.Fragment>
);
```

I w końcu wyświetlamy na stronie profilu użytkownika ikonę ze zbioru Font Awesome, a za nią bieżącą nazwę użytkownika (`username`). Docelowo ten element menu będzie można kliknąć, aby wyświetlić ekran profilu użytkownika:

```
export default SideBarMenus;
```

Kolejne ekrany aplikacji, służące do logowania, wylogowania, rejestracji użytkowników itd., będą dodawane jako kolejne elementy tej listy.

W mojej opinii standardowe oznaczenia punktów list rozpraszają uwagę, dlatego też usuniemy je ze wszystkich list punktowanych. W tym celu dodaj do pliku *index.css* poniższą regułę:

```
ul {
  list-style-type: none;
}
```

8. Teraz musimy zmodyfikować plik *SideBar.tsx*, a konkretnie zastosować w nim przygotowany przed chwilą komponent *SideBarMenus.tsx*. Zaczniij od dodania odpowiedniej instrukcji importu, po czym umieść w kodzie JSX komponent `SideBarMenus`:

```
const SideBar = () => {
  const { width } = useWindowDimensions();
  if (width <= 768) {
    return null;
  }
  return (
    <div className="sidebar">
      <SideBarMenus />
    </div>
  );
};
```

Teraz możemy zmodyfikować kod JSX komponentu SideBar i dodać do niego nowy komponent SideBarMenus.

Zwróć uwagę na to, że podczas dalszych prac napiszemy kod, który będzie wyświetlał ikonę użytkownika oraz jego nazwę wyłącznie w przypadku, gdy użytkownik będzie zalogowany. Samą nazwę użytkownika przekształcimy także w odnośnik, którego kliknięcie będzie powodowało wyświetlenie ekranu profilu użytkownika. Jednak zanim będziemy mogli to zrobić, musimy uruchomić serwerową część aplikacji. Dlatego na razie korzystamy z kodu zastępczego.

9. A teraz spróbujmy ponownie wykorzystać komponent SideBarMenus, tym razem w celu udostępnienia jego zawartości na urządzeniach mobilnych. Kolejne zmiany będziesz musiał wprowadzić w pliku *Nav.tsx* zapisanym w katalogu *components*. Zacznij od dodania do niego niezbędnych instrukcji importu, po czym wprowadź zmiany zgodnie z poniższym przykładem:

```
const Nav = () => {
  const { width } = useWindowDimensions();

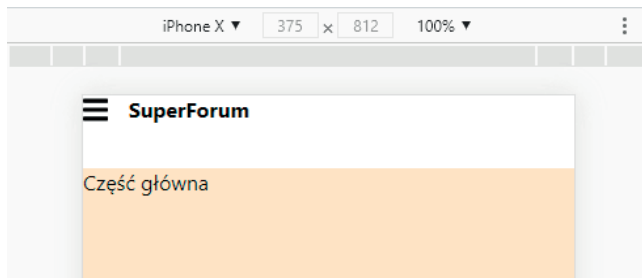
  const getMobileMenu = () => {
    if (width <= 768) {
      return (
        <FontAwesomeIcon icon={faBars} size="lg"
          className="nav-mobile-menu" />
      );
    }
    return null;
  };
};
```

Hooka *useWindowDimensions* używaliśmy już wcześniej do określania, czy aplikacja aktualnie działa na urządzeniu mobilnym. Jednak tym razem utworzyliśmy dodatkową funkcję, *getMobileMenu*, która będzie zawierać logikę określającą, jaki kod JSX zwrócić. Jeśli aplikacja nie działa na urządzeniu mobilnym, funkcja nic nie zwraca; w przeciwnym razie zwraca ikonę FontAwesome przedstawiającą trzy poziome kreski, potocznie określaną jako „menu hamburgera”:

```
return (
  <nav className="navigation">
    {getMobileMenu()}
    <strong>SuperForum</strong>
  </nav>
```

```
);
};
export default Nav;
```

Aktualnie ekran aplikacji przeglądany na urządzeniu mobilnym powinien wyglądać jak na rysunku 12.6.



Rysunek 12.6. Menu nawigacyjne w urządzeniu mobilnym

10. Podczas dalszych prac nad aplikacją będziemy potrzebować możliwości wyświetlania modalnych okien dialogowych. Dlatego, nim przejdziemy do dalszych prac, zainstalujemy pakiet `react-modal`. Daje on możliwość wyświetlania wybranych komponentów jako modalnych okien dialogowych. Takie rozwiązanie zapewnia także większą elastyczność i kontrolę nad tym, kiedy komponenty będą wyświetlane. Aby zainstalować ten pakiet, wykonaj następujące polecenia:

```
npm i react-modal
npm i @types/react-modal -D
```

11. Aby wykorzystać te modalne okna dialogowe oraz zapewnić ich responsywność i możliwość dostosowywania się do różnych wielkości ekranów, musimy wprowadzić pewne zmiany w używanych arkuszach stylów. W pliku `App.css` znajduje się klasa o nazwie `modal-menu`, która określa postać naszych modalnych okien dialogowych.

Do domyślny styl modalnych okien dialogowych dla wszystkich komputerów z wyjątkiem urządzeń mobilnych. Podstawowy problem z nim związany polega na tym, że okno dialogowe jest domyślnie przesuwane do połowy szerokości ekranu (przy użyciu reguły `left: 50%`). Następnie używamy właściwości `transform`, by przesunąć je z powrotem w lewo (o połowę szerokości danego elementu). W ten sposób element zostanie wyśrodkowany na ekranie w poziomie. Zwróć także uwagę na wysoką wartość właściwości `z-index`, dzięki której element będzie wyświetlany powyżej innych.

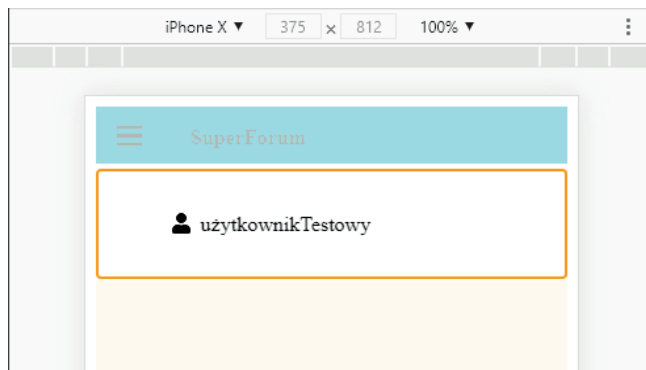
Z myślą o urządzeniach mobilnych, w zapytaniu medialnym umieszczonym w pliku `App.css` zdefiniowaliśmy zmodyfikowaną wersję klasy `modal-menu`. Zawiera ona w zasadzie te same atrybuty, co domyślna klasa `modal-menu`, lecz używa wartości dostosowanych do urządzeń mobilnych. Konkretnie, przesłaniamy wartości `left`, `right` i `top`, aby rozszerzyć okno dialogowe na cały dostępny obszar ekranu. To właśnie dlatego właściwość `transform` ma wartość `0` — po prostu tym razem nie będzie nam potrzebna.



12. Kolejnym krokiem będzie dodanie do menu „hamburgera” procedury obsługi zdarzeń kliknięcia, która będzie wyświetlać komponent `SideBarMenus`. Musimy zatem ponownie zmodyfikować plik `Nav.tsx` i dodać do niego modalne okno dialogowe, które będzie prezentować komponent `SideBarMenus`. Zmodyfikuj zatem plik `Nav.tsx`, dodaj do niego niezbędne instrukcje importu, a następnie kod z analogicznego pliku z przykładów dołączonych do książki.
13. Jeśli zajrzysz do wiersza 10., zauważysz, że do komponentu dodaliśmy nową właściwość stanu o nazwie `showMenu`. Będziemy jej używać po to, by kontrolować, czy modalne okno dialogowe z menu ma być wyświetlone, czy ukryte.
14. Wewnątrz funkcji `getMobileMenu`, w komponencie `FontAwesomeIcon`, zastosowaliśmy procedurę obsługi zdarzeń `onClickToggle`, zmieniającą wartość właściwości stanu `showMenu`, której zmiany sterują ukrywaniem i wyświetlaniem modalnego okna dialogowego.
15. W komponencie `ReactModal`, kiedy zostanie do niego przekazane żądanie zamknięcia, musimy jawnie zapisać `false` we właściwości sterującej wyświetlaniem komponentu, gdyż jeśli tego nie zrobimy, modalne okno dialogowe nigdy nie zniknie. Właśnie do tego celu służy funkcja `onRequestClose`. Przypisanie wartości `true` właściwości `shouldCloseOnOverlayClick` sprawia, że okno będzie zamykane nawet jeśli klikniemy poza jego obszarem. Użytkownicy zazwyczaj oczekują takiego zachowania modalnych okien dialogowych, więc użycie tego rozwiązania jest dobrym pomysłem.
16. I w końcu zmodyfikowaliśmy także kod JSX komponentu, dodając do niego komponent `ReactModal` z umieszczonym wewnątrz niego komponentem `SideBarMenus`.

Warto zwrócić uwagę na jeszcze jedną właściwość komponentu `ReactModal`: `isOpen`. Jej wartość określa, czy okno dialogowe jest widoczne, czy nie.

17. Jeśli teraz uruchomisz aplikację i klikniesz ikonę „hamburgera”, zostanie wyświetlone okno dialogowe, takie jak pokazałem na rysunku 12.7.



**Rysunek 12.7.** Komponent `ReactModal` z naszym komponentem `SideBarMenus`

Przypomnę, że przygotowaniem tego menu zajmiemy się nieco później, kiedy dodamy więcej możliwości.

## Komponenty związane z uwierzytelnianiem

Nasz komponent `SideBar` jest już gotowy, na tyle, na ile na razie mogliśmy go przygotować, możemy zatem zająć się komponentami związanymi z uwierzytelnianiem. Zaczniemy od przygotowania ekranów do rejestracji, logowania i wylogowania.

1. W pierwszej kolejności zajmiemy się modalnym oknem dialogowym do rejestrowania użytkowników. Aby móc je stworzyć, musimy najpierw dodać odpowiedni odnośnik do komponentu `SideBarMenus`. A zatem, otwórz plik *SideBarMenus.tsx* i zaktualizuj go jak na poniższym przykładzie:

```
import {
  faUser,
  faRegistered,
  faSignInAlt,
  faSignOutAlt,
} from "@fortawesome/free-solid-svg-icons";
import Registration from "../auth/Registration";
```

Pierwszych kilka wierszy kodu pozostaje bez zmian, więc ich tu nie przedstawiałem. Poniżej zamieściłem zmodyfikowany kod JSX komponentu:

```
return (
  <React.Fragment>
    <ul>
      <li>
        <FontAwesomeIcon icon={faUser} />
        <span className="menu-name">{user?.userName}</span>
      </li>
      <li>
        <FontAwesomeIcon icon={faRegistered} />
        <span className="menu-name">Rejestracja</span>
      </li>
    </ul>
  </React.Fragment>
);
```

Jak widać, do zwracanego kodu JSX dodaliśmy elementy `li`, nową ikonę oraz etykietę do rejestracji.

2. A teraz, zanim przystąpimy do tworzenia komponentu do rejestracji użytkownika, przygotujemy pomocniczą usługę, której będziemy używać do weryfikacji haseł. Chcemy mieć pewność, że użytkownicy będą podawali dostatecznie długie i złożone hasła, a do tego potrzebujemy **walidatora**. A zatem, w katalogu *src* utwórz nowy podkatalog *common*, a w nim kolejny podkatalog *validators*. W tym podkatalogu *validators* utwórz plik *PasswordValidator.ts* i zapisz w nim kod z analogicznego pliku dostępnego w przykładach do książki. Kod jest dość prosty, więc nie będę tu przedstawiał go w całości, zwróć jednak uwagę na siłę hasła oraz zastosowanie wyrażeń regularnych. Wyrażenia regularne są jedynie programowym sposobem opisu wzorców w łańcuchach znaków:

```

const strongPassword = new RegExp(
  "^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*])(?=.{8,})"
);
if (!strongPassword.test(password)) {
  passwordTestResult.message =
    "Hasło musi zawierać przynajmniej jeden znak specjalny, jedną dużą
    ↪ literę i jedną cyfrę.";
  passwordTestResult.isValid = false;
}

```

W tym kodzie używamy wyrażenia regularnego do określenia dostatecznie wysokiej złożoności hasła, które musi zawierać litery, cyfry oraz znaki specjalne. Zastosowane w wyrażeniu regularnym nawiasy reprezentują zbiory powiązanych ze sobą wyrażeń. Na początku mamy małe litery, następnie duże litery, następnie liczby i w końcu symbole; a na samym końcu wyrażenia została określona jego oczekiwana długość.

```

    return passwordTestResult;
  };

```

Ten kod nie jest jakoś szczególnie złożony, jednak chcemy go używać w kilku różnych komponentach, takich jak komponent do rejestracji użytkowników, oraz w kodzie działającym na serwerze, dlatego umieszczenie go w odrębnym pliku ułatwi jego późniejsze wielokrotne stosowanie.

W aplikacjach jednostronicowych walidacja danych jest zazwyczaj wykonywana dwukrotnie: najpierw po stronie klienta, a następnie na serwerze. Choć takie działanie może się wydawać nadmiarowym, jednak jest konieczne do zapewnienia właściwego bezpieczeństwa aplikacji. Kiedy zajmiemy się serwerową częścią naszej aplikacji, wyjaśnię, w jaki sposób można udostępniać zależności takie jak ta pomiędzy różnymi projektami.

3. Ponieważ przygotowujemy kilka komponentów związanych z uwierzytelnianiem, warto je umieścić w jednym wspólnym miejscu — będzie to podkatalog *auth* w katalogu *components*. Utwórz go zatem, a następnie w tym katalogu utwórz nowy plik o nazwie *Registration.tsx*. Następnie zapisz w tym pliku kod przedstawiony poniżej. Jeśli przyjrzyś się dokładnie kodowi pliku, to z łatwością określisz niezbędne instrukcje importu, które należy umieścić na jego początku. Nie zapomnij także zaktualizować zawartości pliku *App.css*. Pamiętaj, że później przeniesiemy fragmenty tego kodu w inne miejsce, gdzie będzie on współużytkowany, lecz póki co zapiszemy go bezpośrednio w komponencie *Registration*:

```

const userReducer = (state: any, action: any) => {
  switch (action.type) {
    case "userName":
      return { ...state, userName: action.payload };
    case "password":
      return { ...state, password: action.payload };
  }
}

```

```

    case "passwordConfirm":
      return { ...state, passwordConfirm: action.payload };
    case "email":
      return { ...state, email: action.payload };
    case "resultMsg":
      return { ...state, resultMsg: action.payload };
    default:
      return { ...state, resultMsg: "Wystąpiły problemy." };
  }
};

```

W tym fragmencie kodu tworzymy reduktor, który ma wiele powiązanych ze sobą pól:

```

export interface RegistrationProps {
  isOpen: boolean;
  onClickToggle: (
    e: React.MouseEvent<Element, MouseEvent> | React.KeyboardEvent<Element>
  ) => void;
}

```

Ponieważ będzie to komponent modalny, zapewnimy komponentom nadrzędnym możliwość kontrolowania sposobu jego prezentacji z wykorzystaniem właściwości *props*. Właściwość *props* `isOpen` kontroluje sposób, w jaki będzie prezentowane modalne okno dialogowe, natomiast funkcja `onClickToggle` kontroluje wyświetlanie i ukrywanie okna.

```

const Registration: FC<RegistrationProps> = ({ isOpen, onClickToggle }) => {
  const [isRegisterDisabled, setRegisterDisabled] = useState(true);
  const [
    { userName, password, email, passwordConfirm, resultMsg },
    dispatch,
  ] = useReducer(userReducer, {
    userName: "davec",
    password: "",
    email: "admin@dzhaven.com",
    passwordConfirm: "",
    resultMsg: "",
  });
};

```

W tym fragmencie kodu tworzymy właściwość stanu `isRegistrationDisabled`, która dezaktywuje przycisk kończący rejestrację w przypadku, gdy podane wartości będą nieprawidłowe, i używamy przygotowanego wcześniej reduktora `userReducer`.

```

const allowRegister = (msg: string, setDisabled: boolean) => {
  setRegisterDisabled(setDisabled);
  dispatch({ payload: msg, type: "resultMsg" });
};

```

`allowRegister` to funkcja pomocnicza, której będziemy używali do wyłączania przycisku do rejestracji i wyświetlania stosownego komunikatu.

4. Kolejny fragment kodu pliku stanowi grupa procedur obsługi zdarzeń `onChange`, po jednej dla każdego z pól, takich jak `userName`. Każda z tych procedur wykonuje walidację, o ile jest to konieczne, i aktualizuje wpisany tekst:

```
const onChangeUserName = (e: React.ChangeEvent<HTMLInputElement>) => {
  dispatch({ payload: e.target.value, type: "userName" });
  if (!e.target.value)
    allowRegister(dispatch, "Nazwa użytkownika nie może być pusta.", true);
  else allowRegister(dispatch, "", false);
};
```

Funkcja `onChangeUserName` służy do ustawiania wartości stanu `userName` i określania, czy proces rejestracji może być kontynuowany.

```
const onChangeEmail = (e: React.ChangeEvent<HTMLInputElement>) => {
  dispatch({ payload: e.target.value, type: "email" });
  if (!e.target.value) allowSubmit(dispatch, "E-mail nie może być pusty.",
    true);
  else allowSubmit(dispatch, "", false);
};
```

Funkcja `onChangeEmail` służy do ustawiania wartości adresu poczty elektronicznej użytkownika i określania, czy proces rejestracji może być kontynuowany.

```
const onChangePassword = (e: React.ChangeEvent<HTMLInputElement>) => {
  dispatch({ payload: e.target.value, type: "password" });
  const passwordCheck: PasswordTestResult = isPasswordValid(e.target.
    value);
  if (!passwordCheck.isValid) {
    allowRegister(passwordCheck.message, true);
    return;
  }
  passwordsSame(passwordConfirm, e.target.value);
};
```

Funkcja `onChangePassword` służy do ustawiania wartości hasła i określania, czy proces rejestracji może być kontynuowany:

```
const onChangePasswordConfirm = (e: React.ChangeEvent<HTMLInputElement>) => {
  dispatch({ payload: e.target.value, type: "passwordConfirm" });
  passwordsSame(password, e.target.value);
};
```

Funkcja `onChangePasswordConfirm` służy do ustawiania wartości właściwości stanu `passwordConfirm` i określania, czy proces rejestracji może być kontynuowany:

```
const passwordsSame = (passwordVal: string, passwordConfirmVal: string) => {
  if (passwordVal !== passwordConfirmVal) {
    allowRegister("Hasła nie są takie same.", true);
    return false;
  } else {
    allowRegister("", false);
    return true;
  }
};
```

I w końcu, ponieważ mamy do czynienia z komponentem służącym do rejestracji użytkowników, definiujemy funkcję `passwordsSame`, która będzie sprawdzać, czy wartości podane w obu polach hasła są takie same.

5. Kolejnymi elementami pliku są funkcje `onClickRegister` oraz `onClickCancel`. Pierwsza z tych dwóch funkcji będzie obsługiwała kliknięcia przycisku kończącego rejestrację, a w jej kodzie będziemy przesyłali dane użytkownika na serwer w celu podjęcia próby jego zarejestrowania. Aktualnie część serwerowa aplikacji jeszcze nie istnieje, więc nie będziemy także przysyłać danych użytkownika — później, kiedy część serwerowa będzie już gotowa, uzupełnimy te braki. Druga z funkcji, `onClickCancel`, powoduje wyjście z komponentu `Registration`:

```
const onClickRegister = (
  e: React.MouseEvent<HTMLButtonElement, MouseEvent>
) => {
  e.preventDefault();
  onClickToggle(e);
};

const onClickCancel = (
  e: React.MouseEvent<HTMLButtonElement, MouseEvent>
) => {
  onClickToggle(e);
};
```

Zwróć uwagę na to, że funkcja `e.preventDefault` uniemożliwia wykonanie standardowego sposobu obsługi zdarzenia, który może być różny w zależności od kontekstu. W przypadku formularzy nasza procedura obsługi `onClickRegister` jest skojarzona z przyciskiem umieszczonym wewnątrz formularza, więc ten standardowy sposób obsługi polegałby na przesłaniu formularza i odświeżeniu strony. Zwłaszcza ta ostatnia operacja, odświeżenie strony, jest czymś, czego zdecydowanie **nie** chcemy robić w aplikacjach jednostronicowych, dlatego też nie zezwalamy na nią, używając w tym celu funkcji `preventDefault`.

6. Skoro procedury obsługi zdarzeń są już gotowe, możemy zająć się kodem JSX, w którym będą one używane. Zacznij od dodania komponentu `ReactModal`, wewnątrz którego umieścimy cały pozostały kod JSX:

```
return (
  <ReactModal
    className="modal-menu"
    isOpen={isOpen}
    onRequestClose={onClickToggle}
    shouldCloseOnOverlayClick={true}
  >
    <form>
      <div className="reg-inputs">
        <div>
          <label>Nazwa użytkownika</label>
          <input type="text" value={userName} onChange={onChangeUserName} />
        </div>
      </div>
    </form>
  </ReactModal>
);
```

Także w tym przypadku nasze modalne okno dialogowe jest kontrolowane z zewnątrz, przez komponent nadrzędny, przy użyciu właściwości *props* `isOpen` oraz `onClickToggle`.

```

<div>
  <label>E-mail</label>
  <input type="text" value={email} onChange={onChangeEmail} />
</div>

```

To jest pole służące do podania adresu poczty elektronicznej.

```

<div>
  <label>Hasło</label>
  <input
    type="password"
    placeholder="Podaj hasło"
    value={password}
    onChange={onChangePassword}
  />
</div>

```

To jest pole służące do podania hasła.

```

<div>
  <label>Powtórz hasło</label>
  <input
    type="password"
    placeholder="Powtórz hasło"
    value={passwordConfirm}
    onChange={onChangePasswordConfirm}
  />
</div>
</div>

```

To jest pole do powtórnego podania hasła.

```

<div className="form-buttons">
  <div className="form-btn-left">
    <button
      style={{ marginLeft: ".5em" }}
      className="action-btn"
      disabled={isRegisterDisabled}
      onClick={onClickRegister}
    >
      Rejestruj
    </button>

```

Tu mamy przycisk do zakończenia rejestracji.

```

    <button
      style={{ marginLeft: ".5em" }}
      className="cancel-btn"
      onClick={onClickCancel}
    >
      Zamknij
    </button>

```

A tu jest przycisk do zamykania ekranu rejestracji.

```

</div>
<span className="form-btn-right">
  <strong>{resultMsg}</strong>

```

```

        </span>
      </div>
    </form>
  </ReactModal>
);
};
export default Registration;

```

Zwróć uwagę na to, że na samym końcu umieszczona jest sekcja służąca do wyświetlania komunikatu, w której umieszczamy zawartość pola resultMsg reduktora. W tym miejscu, jeśli coś pójdzie nie tak, będą wyświetlane komunikaty o błędach.

7. Komponent `Registration` będzie używany w komponencie `SideBarMenu`, dlatego kolejne zmiany musimy wprowadzić w jego kodzie. Zmodyfikuj definicję komponentu `SideBarMenu` według poniższego przykładu (nie zapomnij także o zaimportowaniu nowego komponentu `Registration`):

```

const SideBarMenus = () => {
  const user = useSelector((state: AppState) => state.user);
  const dispatch = useDispatch();

  const [showRegister, setShowRegister] = useState(false);

  const onClickToggleRegister = () => {
    setShowRegister(!showRegister);
  };
};

```

Następnie zmodyfikuj także kod JSX komponentu:

```

...
<li>
  <FontAwesomeIcon icon={faRegistered} />
  <span onClick={onClickToggleRegister} className="menu-name">
    ↪ Rejestracja</span>
    <Registration
      isOpen={showRegister}
      onClickToggle={onClickToggleRegister}
    />
  </li>
...

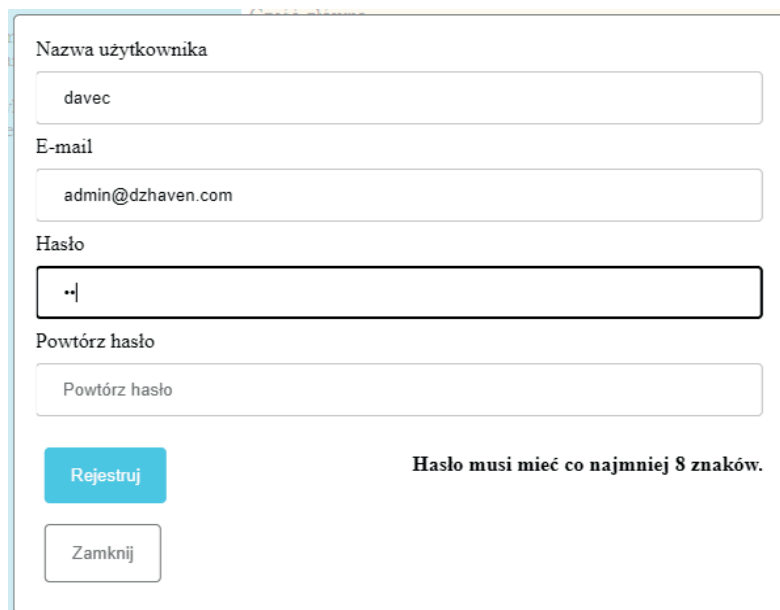
```

8. Jeśli teraz uruchomisz aplikację w trybie dla komputerów stacjonarnych, to zobaczysz ekran podobny do tego przedstawionego na rysunku 12.8.

Jeśli wyświetlisz Debuggera przeglądarki Chrome i przełączysz przeglądarkę w mobilny tryb działania, to po kliknięciu ikony „hamburera” oraz wybraniu odnośnika do rejestracji zobaczysz okno przedstawione na rysunku 12.9.

Jak widać, dzięki odpowiedniemu wykorzystaniu responsywnych możliwości, jakie daje CSS, mogliśmy w zasadzie uzyskać dwa ekrany, używając do tego celu jednego komponentu.





Nazwa użytkownika

E-mail

Hasło

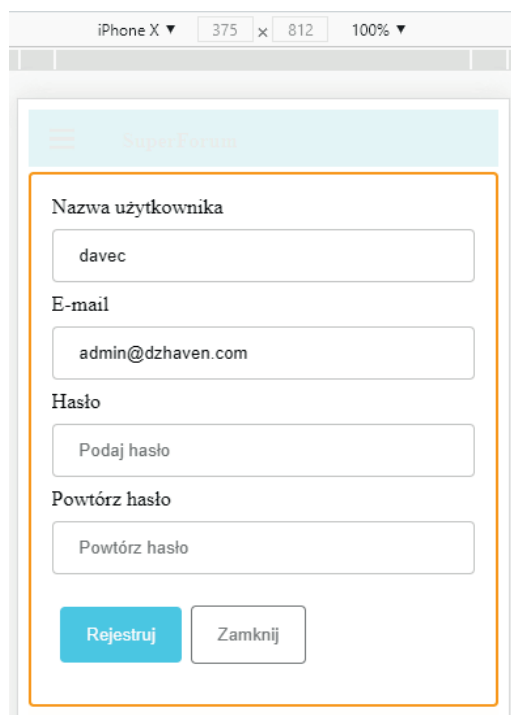
Powtórz hasło

**Rejestruj**

Zamknij

**Hasło musi mieć co najmniej 8 znaków.**

Rysunek 12.8. Modalne okno dialogowe rejestracji w trybie dla komputerów stacjonarnych



iPhone X 375 x 812 100%

SuperForum

Nazwa użytkownika

E-mail

Hasło

Powtórz hasło

**Rejestruj**

Zamknij

Rysunek 12.9. Modalne okno dialogowe rejestracji w trybie dla urządzeń mobilnych

9. Teraz przejdźmy do modalnego okna dialogowego z formularzem do logowania. Jeśli przyjrzymy się istniejącemu komponentowi `Registration`, zauważymy, że zawiera on fragmenty kodu, których możemy także użyć w komponencie `Login`. Zdecydowanie powinniśmy w takim razie zrefaktoryzować nasz kod w taki sposób, by można go było używać wielokrotnie. Na przykład komponenty `Registration`, `Login` oraz `Logout` — wszystkie one będą używać komponentu `ReactDOM`, a zatem także będą pobierać właściwości *props* służące do kontrolowania działania modalnego okna dialogowego. Zastanówmy się zatem, co musielibyśmy zrobić, żeby wykorzystać nasz już istniejący kod. W pierwszej kolejności wyodrębnimy interfejs `RegistrationProps` z pliku *Registration.tsx* i umieścimy go w odrębnym pliku. W katalogu *components* utwórz podkatalog *types*. W kolejnym kroku w tym katalogu utwórz plik *ModalProps.ts* i skopiuj do niego interfejs `RegistrationProps`. Następnie zmień nazwę interfejsu na `ModalProps`.

Jak widać, jest to dokładnie ten sam interfejs, co `RegistrationProps`, tylko ze zmienioną nazwą. A teraz otwórz plik *Registration.tsx*, usuń z niego interfejs `RegistrationProps` i zaimportuj interfejs `ModalProps`. Następnie w kodzie komponentu `Registration` zastąp `RegistrationProps` typem `ModalProps`. Sprawdź, czy wszystko działa tak, jak wcześniej.

10. Zrefaktoryzowaliśmy interfejs `ModalProps` i możemy go już używać w wielu komponentach. Teraz wyodrębnimy reduktor `userReducer`, gdyż komponent `Login` będzie używać niektórych jego pól. W katalogu *auth* utwórz nowy podkatalog *common*, a w nim utwórz plik *UserReducer.ts*. Następnie zapisz w nim następujący fragment kodu:

```
const userReducer = (state: any, action: any) => {
  switch (action.type) {
    case "userName":
      return { ...state, userName: action.payload };
    case "password":
      return { ...state, password: action.payload };
    case "passwordConfirm":
      return { ...state, passwordConfirm: action.payload };
    case "email":
      return { ...state, email: action.payload };
    case "resultMsg":
      return { ...state, resultMsg: action.payload };
    case "isSubmitDisabled":
      return { ...state, isSubmitDisabled: action.payload };
    default:
      return { ...state, resultMsg: "Wystąpiły problemy." };
  }
};
export default userReducer;
```

Zwróć uwagę na to, że do kodu reduktora dodaliśmy nowe pole, o nazwie `isSubmitDisabled`. To pole zastąpi aktualnie używane pole `isRegistrationDisable`, dzięki czemu będzie go można używać do wyłączania przycisków na różnych ekranach związanych z uwierzytelnianiem.

Teraz usuń całą funkcję `userReducer` z pliku *Registration.tsx* i zaimportuj ją z pliku *UserReducer.ts*. Oprócz tego usuń z kodu odwołania do `isRegistrationDisabled` i zastąp je odwołaniami do `isSubmitDisabled`, dodaj także `isSubmitDisabled` do wyrażenia destrukuryzującego oraz do inicjalizatora przekazywanego w wywołaniu `useReducer`.

11. Przejdźmy do dalszych modyfikacji. Zdefiniowana w komponencie *Registration* funkcja `allowRegister` dezaktywuje przycisk i aktualizuje komunikat statusu. Wyraźnie widać, że także jej będzie można wielokrotnie używać. A zatem, w katalogu *components/auth/common* utwórz kolejny plik, *Helpers.ts*, i zapisz w nim poniższy kod:

```
import { Dispatch } from "react";

export const allowSubmit = (
  dispatch: Dispatch<any>,
  msg: string,
  setDisabled: boolean
) => {
  dispatch({ type: "isSubmitDisabled", payload: setDisabled });
  dispatch({ payload: msg, type: "resultMsg" });
};
```

Jak widać, zmieniliśmy nazwę funkcji z `allowRegister` na `allowSubmit` i dodaliśmy do niej nowy parametr: `dispatch`. Usuń zatem funkcję `allowRegister` z kodu komponentu *Registration*, zaimportuj do niego nową funkcję `allowSubmit` i zastąp wszystkie wywołania `allowRegister` wywołaniami `allowSubmit`. Porównaj kod źródłowy komponentu z plikiem z przykładów dołączonych do książki.

Dwie procedury obsługi zdarzeń `onClick` pozostawimy w niezmienionej postaci, choć komponent *Login* będzie wymagał podobnych funkcji. Zrobimy tak, gdyż później, kiedy już uruchomimy serwerową część aplikacji, w funkcjach tych w obu komponentach będziemy zapewne musieli wykonywać inne czynności, zależne od komponentu.

Teraz powinieneś już być w stanie ponownie uruchomić aplikację.

12. Kolejnym krokiem będzie użycie wyodrębnionego przed chwilą kodu w nowym komponencie *Login*. W katalogu *auth* utwórz plik *Login.tsx* i skopiuj do niego kod z analogicznego pliku dostępnego w przykładach dołączonych do książki. Poniżej przedstawię kilka najważniejszych fragmentów tego kodu:

```
const [
  { userName, password, resultMsg, isSubmitDisabled },
  dispatch,
] = useReducer(userReducer, {
  userName: "",
  password: "",
  resultMsg: "",
  isSubmitDisabled: true,
});
```

Komponent `Login` ma inne potrzeby niż komponent `Registration`, dlatego też używamy w nim tylko fragmentu pól zwracanych przez reduktor `userReducer` — te, które nas interesują, wybieramy używając odpowiedniego wyrażenia destrukuryzującego.

W kodzie JSX komponentu zwróć uwagę na to, że zaktualizowaliśmy niektóre klasy CSS, aby lepiej wyrównać przyciski. Nowe klasy umieściliśmy w arkuszu stylów *App.css*.

13. I w końcu musimy dodać odnośnik służący do wyświetlania okna dialogowego do logowania. W tym celu zmodyfikuj plik *SideBarMenu.tsx* zgodnie z kodem z przykładów dołączonych do książki.

Komponent `Logout` jest bardzo podobny, dlatego nie przedstawiam go tutaj, choć dodałem go do aplikacji. Później, kiedy przygotujemy już serwerowe elementy aplikacji, dodamy do komponentu `SideBarMenu` kod kontrolujący to, które odnośniki będą widoczne na podstawie stanu zalogowania użytkownika. Oprócz tego dodamy do niego dodatkowe funkcje sprawdzające poprawność danych. Jednak zanim to zrobimy, czeka nas jeszcze wiele pracy nad innymi elementami aplikacji. A zatem, nie marnujemy czasu!

## Trasowanie i ekrany aplikacji

Kolejnym zagadnieniem, którym się zajmiemy, będą trasy używane w aplikacji. Póki co cała nasza aplikacja używa tylko jednego adresu URL — jest to główny adres URL aplikacji: *http://localhost:3000*. Chcemy jednak podzielić aplikację i wydzielić odrębne trasy dla jej poszczególnych sekcji. Zaczniemy od zmodyfikowania istniejącego kodu aplikacji i przygotowania pierwszej trasy. Wszystkie czynności związane z tymi modyfikacjami opisałem na poniższej liście:

1. Zaczniemy od przeniesienia wszystkich komponentów związanych z siatką CSS do odrębnych katalogów. W tym celu w katalogu *components* utwórz podkatalog *areas*. Następnie przenieś do niego pliki *Nav.tsx*, *Nav.css*, *RightMenu.tsx*, *LeftMenu.tsx* i *Left.css*, jak również cały katalog *sidebar*. Po wprowadzeniu tych zmian będziesz musiał poprawić ścieżki w instrukcjach importu w kilku różnych plikach, w tym także w *App.tsx*. Zajrzyj do kodów źródłowych przykładów dołączonych do książki, aby zobaczyć, jak należy to zrobić.
2. Kiedy już poprawisz ścieżki, utwórz w katalogu *areas* nowy podkatalog, o nazwie *main*, i przenieś do niego plik *Main.tsx*. Ponownie odpowiednio zmodyfikuj ścieżki. Do tego nowego katalogu będziemy dodawać wszystkie komponenty związane z głównym obszarem aplikacji.
3. Pierwszy nowy komponent, który utworzymy w katalogu *main*, nosi nazwę `MainHeader`. Zgodnie z tym, co sugeruje jego nazwa, będzie on pełnił rolę nagłówka dla głównej sekcji witryny. Będziemy w nim wyświetlać nazwę kategorii aktualnie przeglądanej wątku. Utwórz zatem w katalogu *main* plik *MainHeader.tsx* i skopiuj do niego kod z analogicznego pliku z przykładów dołączonych do książki.

Jedynym przeznaczeniem tego komponentu jest wyświetlanie nazwy bieżącej kategorii.

Zwróć także uwagę na to, że na potrzeby tego komponentu musisz skopiować także plik arkusza stylów, *MainHeader.css*, oraz dodać parę nowych klas do arkusza *App.css*.

## Ekran główny

Nim zajmiemy się kolejnymi pracami, musimy dodać podstawową konfigurację nowej trasy. Utworzymy komponent nowego ekranu, *Home*, i zaktualizujemy wszystkie istniejące pliki aplikacji, w których trzeba coś zmienić, takie jak *App.tsx*.

1. Podczas tworzenia projektu kod komponentu *App*, umieszczony w pliku *App.tsx*, został wygenerowany w taki sposób, jakby nasza aplikacja miała się składać tylko z jednego pliku. Oczywiście, teraz założenie to jest fałszywe. Obecnie, kiedy określiliśmy już układ aplikacji, możemy zająć się dodawaniem do niej poszczególnych ekranów i tras. Otwórz zatem plik *App.tsx* i wprowadź w nim opisane poniżej zmiany.

Zacznij od dodania nowej instrukcji *import*, która zaimportuje komponent *Home* reprezentujący trasę do strony głównej. Przygotowaniem tego komponentu zajmiemy się nieco później:

```
import Home from "../components/routes/Home";

function App() {
  const renderHome = (props: any) => <Home {...props} />;
```

W tym fragmencie kodu definiujemy funkcję, która zostanie przekazana do właściwości *render* trasy. Dzięki tej funkcji, podczas inicjalizowania komponentu *Home*, będziemy mogli przekazać do niego wszystkie właściwości *props* trasy, jak również wszelkie inne, niestandardowe właściwości *props*, które będziemy chcieli do niego przekazać:

```
    return (
      <Switch>
        <Route exact={true} path="/" render={renderHome} />
        <Route
          path="/categorythreads/:categoryId"
          render={renderHome}
        />
      </Switch>
    );
  }
}
```

Nasz dotychczasowy kod określający poszczególne obszary siatki zostanie teraz przeniesiony do komponentu *Home*, którym zajmiemy się już niebawem.

Jak pokazałem w rozdziale 7., pt. „*Redux i React Router*”, komponent *Switch* pozwala Routerowi zmieniać renderowaną trasę w zależności od podanego adresu URL. Na razie będziemy mieli tylko dwie trasy, i to odwołujące się do tego samego komponentu *Home*, jednak później dodamy ich więcej. Trasa główna będzie prezentować wątki dostępne w kategorii domyślnej, natomiast trasa *categorythreads* będzie wyświetlać wątki w wybranej kategorii.

2. Zanim utworzymy komponent `Home`, musimy nieco zmodyfikować używane style CSS i zwiększyć możliwości ich wielokrotnego używania. W pierwszej kolejności zmodyfikuj plik `App.css`, dodając do niego następującą klasę (zapisz ją przed klasą `App`):

```
.screen-root-container {
  margin: 0 auto;
  max-width: 1200px;
  margin-bottom: 2em;
  border: var(--border);
  border-radius: 0.3em;
}
```

W przyszłości będzie to główna klasa, określająca postać wszelkich komponentów reprezentujących trasy zdefiniowane w naszej aplikacji.

3. Następnie w katalogu *components* utwórz nowy podkatalog, o nazwie *routes*, a w nim plik `Home.css`. Do tego pliku przenieś z pliku `App.css` poniższy styl:

```
.App { // Później zmienimy tej klasie nazwę na home-container
  margin: 0 auto;
  max-width: 1200px;
  display: grid;
  grid-template-columns: 0.7fr 0.9fr 1.5fr 0.9fr;
  grid-template-rows: 2.75rem 3fr;
  grid-template-areas:
    "nav nav nav nav"
    "sidebar leftmenu content rightmenu";
  gap: 0.75rem 0.4rem;
}
```

Kolejny zestaw kodu CSS, który musisz przenieść do pliku `Home.css`, zawiera reguły określające postać poszczególnych obszarów siatki:

```
.navigation {
  grid-area: nav;
}
.sidebar {
  min-height: var(--min-screen-height);
  grid-area: sidebar;
  background-color: aliceblue;
}
.leftmenu {
  grid-area: leftmenu;
  background-color: skyblue;
}
.content {
  min-height: var(--min-screen-height);
  grid-area: content;
  background-color: blanchedalmond;
}
.rightmenu {
  grid-area: rightmenu;
  background-color: coral;
}
```

```
@media screen and (orientation: portrait) and (max-width: 768px) {
  .home-container {
    grid-template-columns: 1fr;
    grid-template-areas:
      "nav"
      "content";
  }
}
```

Jak wcześniej, skopiuj ten kod, zapisz go w pliku *Home.css*, po czym usuń z *App.css*. Następnie w pliku *Home.css* zmień nazwę klasy *App* na *home-container*. Ta zmiana ma na celu jednoznaczne określenie przeznaczenia klasy. Teraz możemy już zająć się utworzeniem komponentu dla ekranu *Home*, a przy okazji pokazać, jak zastosować nowe klasy CSS.

4. W katalogu *components/routes* utwórz nowy plik, o nazwie *Home.tsx*. Jego kod jest krótki i łatwy, więc po prostu skopiuj go z analogicznego pliku dostępnego w przykładach dołączonych do książki. W przeważającej większości jest to kod, który wcześniej znajdował się w pliku *App.tsx*.

Główną klasę CSS używaną w tym komponentcie, *App*, zastąpiliśmy teraz dwiema klasami: *screen-root-container* i *home-container*. Podanie dwóch klas w atrybucie *class* oznacza, że najpierw należy zastosować pierwszą z nich, a następnie drugą, co może spowodować przesłonięcie niektórych stylów używanych w pierwszej klasie. Przy okazji zwróć uwagę na to, że aktualnie będziemy mogli używać klasy *screen-root-container* także na innych ekranach aplikacji.

Udało się nam przenieść kod z pliku *App.tsx* do *Home.tsx*. Zauważ ponadto, że w kodzie JSX komponentu *Home* komponent *Nav* umieściliśmy dodatkowo w elemencie *div*. Ta zmiana zapewni nam później możliwość wykorzystania komponentu *Nav* na innych ekranach aplikacji. Teraz powinieneś wprowadzić jeszcze jedną zmianę: usuń atrybut *className="navigation"* z pliku *Nav.tsx*.

5. Skoro dysponujemy już komponentem *Home*, musimy zmodyfikować komponent *Main* w taki sposób, by prezentował listę wątków w wybranej kategorii. Jednak, aby to zrobić, będziemy musieli wprowadzić całkiem sporo zmian. W pierwszej kolejności musimy utworzyć dwa nowe modele: *Thread* oraz *ThreadItem*. Pierwszy z nich, *Thread*, będzie reprezentować wątek, czyli początkowy wpis, natomiast *ThreadItem* będzie reprezentować odpowiedzi publikowane w tym wątku. Zacznijmy od przygotowania tych modeli.

W katalogu *models* utwórz plik *Thread.ts* i skopiuj do niego kod z analogicznego pliku z przykładów.

Kod tej klasy jest prosty i nie ma sensu go dodatkowo wyjaśniać. Zwróć tylko uwagę na właściwość *points* — określa ona, ile razy jakiś użytkownik polubił dany wpis.

Teraz utwórz plik *ThreadItem.ts* i — jak wcześniej — skopiuj do niego kod z analogicznego pliku z przykładów. Ten kod jest bardzo podobny do kodu klasy *Thread*.

6. Kolejnym krokiem będzie utworzenie komponentu karty wątku. Będzie on reprezentował rekord jednego wątku i prezentował takie informacje, jak: tytuł wątku, jego treść oraz liczbę punktów. A zatem, w katalogu *components/areas/main* utwórz plik *ThreadCard.tsx* i zapisz w nim następujący kod:

```
import React, { FC } from "react";
import "./ThreadCard.css";
import Thread from "../../models/Thread";
import { Link, useHistory } from "react-router-dom";
import { faEye, faHeart, faReplyAll } from "@fortawesome/free-solid-svg-
  icons";
import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";
import { useWindowDimensions } from "../../hooks/useWindowDimensions";
```

Ten pierwszy fragment kodu zawiera instrukcje importu, dołączające między innymi obiekt *Link* oraz *hook* *useHistory* udostępniany przez *React Router*.

```
interface ThreadCardProps {
  thread: Thread;
}
```

Zwróć uwagę na to, że ten komponent pobiera jeden parametr — obiekt *Thread*; będziemy go używać do wyświetlania informacji podczas tworzenia interfejsu użytkownika komponentu *ThreadCard*.

```
const ThreadCard: FC<ThreadCardProps> = ({ thread }) => {
  const history = useHistory();
  const { width } = useWindowDimensions();

  const onClickShowThread = (e: React.MouseEvent<HTMLDivElement>) => {
    history.push("/thread/" + thread.id);
  };
};
```

W tym fragmencie kodu używamy *hooka* *useHistory* *React Router*a, by pobrać obiekt *history*. Kiedy użytkownik kliknie wątek, użyjemy tego obiektu, by wyświetlić w aplikacji inny adres URL; wykorzystamy w tym celu funkcję *push* obiektu *history*. Zarówno trasę *thread*, jak i powiązany z nią komponent przygotowujemy w dalszej części rozdziału.

```
const getPoints = (thread: Thread) => {
  if (width <= 768) {
    return (
      <label
        style={{
          marginRight: ".75em",
          marginTop: ".25em",
        }}
      >
        {thread.points || 0}
        <FontAwesomeIcon
          icon={faHeart}
          className="points-icon"
          style={{
            marginLeft: ".2em",
          }}
        />
      </label>
    );
  }
};
```



```

    />
  </label>
  );
}
return null;
};

```

Funkcja `getPoints` tworzy interfejs użytkownika do wyświetlania „polubień” danego wpisu. Jednak ze względu na responsywny charakter naszej aplikacji, ten fragment interfejsu użytkownika nie jest wyświetlany w trybie dla komputerów stacjonarnych — zwróć uwagę na warunek sprawdzający szerokość (`width`) ekranu.

```

const getResponses = (thread: Thread) => {
  if (width <= 768) {
    return (
      <label
        style={{
          marginRight: ".5em",
        }}
      >
        {thread.threadItems && thread.threadItems.length}

```

Ta funkcja wyświetla liczbę odpowiedzi, którą odczytujemy, używając właściwości `thread.threadItems.length`:

```

      <FontAwesomeIcon
        icon={faReplyAll}
        className="points-icon"
        style={{
          marginLeft: ".25em",
          marginTop: "-.25em",
        }}
      />
    </label>
  );
}
return null;
};

```

Funkcja `getResponses` pokazuje, ile odpowiedzi, reprezentowanych przez obiekty `ThreadItem`, jest dostępnych dla danego obiektu `Thread`. Jednak ze względu na responsywny charakter naszej aplikacji, informacja ta nie jest wyświetlana w trybie dla komputerów stacjonarnych — zwróć uwagę na warunek sprawdzający szerokość (`width`) ekranu.

```

const getPointsNonMobile = () => {
  if (width > 768) {
    return (
      <div className="threadcard-points">
        <div className="threadcard-points-item">
          {thread.points || 0}
          <br />
          <FontAwesomeIcon icon={faHeart}
            className="points-icon" />

```

```

    </div>
    <div
      className="threadcard-points-item"
      style={{ marginBottom: ".75em" }}
    >
      {thread && thread.threadItems && thread.threadItems.length}

```

Ta funkcja pobiera liczbę polubień wątku, odczytując ją z właściwości `thread.threadItems.length`.

```

      <br />
      <FontAwesomeIcon icon={faReplyAll}
        className="points-icon" />
    </div>
  </div>
);
}
return null;
};

```

Funkcja `getPointsNonMobile` zwraca kolumnę z liczbą punktów prezentowaną po prawej stronie komponentu `ThreadCard`, lecz renderuje ją wyłącznie jeśli aplikacja jest używana w komputerze stacjonarnym lub laptopie, ewentualnie innym urządzeniu z ekranem o szerokości większej od 768 pikseli.

Pamiętaj, że każdy komponent Reacta, który może pojawić się wiele razy na tym samym ekranie, musi dysponować właściwością `key` z unikalną wartością. Dlatego w dalszej części kodu, kiedy będziemy używać tego komponentu, przekonasz się, że do każdej jego instancji przekazujemy unikalną wartość właściwości `key`. Kolejny fragment kodu JSX zwraca nazwę kategorii jako komponent `Link`; dzięki temu, kiedy użytkownik kliknie tę nazwę, zostanie wyświetlony ekran prezentujący wątki dla wybranej kategorii:

```

return (
  <section className="panel threadcard-container">
    <div className="threadcard-txt-container">
      <div className="content-header">
        <Link
          to={`~/categorythreads/${thread.category.id}`}
          className="link-txt"
        >
          <strong>{thread.category.name}</strong>
        </Link>

```

`Link` to komponent React Routera, który generuje znacznik HTML odnośnika. Zwróć uwagę na to, że `categorythread` to druga z naszych tras — utworzyliśmy ją nieco wcześniej i wymaga ona przekazania identyfikatora kategorii (`categoryId`) jako parametru.

```

      <span className="username-header" style={{ marginLeft: ".5em" }}>
        {thread.userName}
      </span>
    </div>

```

```

<div className="question">
  <div
    onClick={onClickShowThread}
    data-thread-id={thread.id}
    style={{ marginBottom: ".4em" }}
  >
    <strong>{thread.title}</strong>
  </div>
  <div
    className="threadcard-body"
    onClick={onClickShowThread}
    data-thread-id={thread.id}
  >
    <div>{thread.body}</div>
  </div>

```

Jak widać, podczas generowania interfejsu użytkownika często używamy właściwości `thread`.

W kolejnym fragmencie kodu używamy funkcji `getPoints` oraz `getResponses` do wyświetlenia dwóch fragmentów interfejsu użytkownika, prezentujących odpowiednio liczbę punktów oraz odpowiedzi:

```

<div className="threadcard-footer">
  <span
    style={{
      marginRight: ".5em",
    }}
  >
    <label>
      {thread.views}
      <FontAwesomeIcon icon={faEye} className="icon-lg" />
    </label>
  </span>
  <span>
    {getPoints(thread)}
    {getResponses(thread)}
  </span>
</div>
</div>
</div>

```

W kolejnym fragmencie wywołujemy funkcję `getPointsNonMobile`, aby wyświetlić liczbę odpowiedzi i polubień:

```

    {getPointsNonMobile()}
  </section>
);
};
export default ThreadCard;

```

Zwróć uwagę na to, że w tym komponencie zastosowaliśmy wiele klas CSS, a wszystkie można znaleźć w kodach źródłowych dołączonych do książki, w plikach *ThreadCard.css* oraz *App.css*. Nie będę ich tu szczegółowo opisywał, jeśli jednak zajrzysz do pliku *ThreadCard.css*, to zauważysz w nim odwołania

do czegoś, co nosi nazwę flex. Flexbox to kolejny sposób określania układów stron WWW, przypominający nieco siatki CSS (CSS Grid). Jest on przeznaczony do tworzenia układów o postaci jednego wiersza lub jednej kolumny, jak pokazałem na poniższym przykładzie:

```
.threadcard-txt-container {
  display: flex;
  flex-direction: column;
  width: 92%;
  margin: 0.75em 1em 0.75em 1.2em;
  border-right: solid 1px var(--border-color);
}
```

W tej regule CSS, sposób wyświetlania elementu (właściwość `display`) został określony jako `flex`, a zawartość tego elementu ma być wyświetlana w formie kolumny (właściwość `flex-direction`). To oznacza, że wszystkie elementy zapisane wewnątrz elementu `threadcard-txt-container` zostaną umieszczone jeden pod drugim w formie jednej kolumny. A zatem, nawet gdybyśmy użyli takich elementów jak etykiety bądź przyciski, które zazwyczaj są rozmieszczane w formie wiersza, to po umieszczeniu ich w takim kontenerze flex o układzie kolumny, zostaną one rozmieszczone w pionie. Z kolei, gdybyśmy we właściwości `flex-direction` użyli wartości `row`, to zawartość elementu byłaby rozmieszczana w poziomie.

7. Skoro już przygotowaliśmy kontener `ThreadCard`, możemy zastosować go w kodzie pliku *Main.tsx*. Zapisz w nim kod źródłowy skopiowany z przykładów dołączonych do książki.

Jeśli zajrzysz do kodu pliku *Main.tsx*, w jego 8. wierszu zauważysz wywołanie funkcji `useParams`. Wcześniej, w pliku *App.tsx* zdefiniowaliśmy dwie trasy obsługiwane przez Router Reacta. Jedna z nich, `categorythreads`, wymaga przekazania parametru w adresie URL. *Hook* `seParams` zapewnia możliwość pobierania parametrów tras, takich jak `categoryId`, i używania ich w kodzie.

Następnie, w wierszu 9., tworzymy właściwość stanu `category`. Jej wartość będziemy aktualizować nieco później, po pobraniu kategorii z listy wątków.

W wierszu 10. tworzymy kolejną właściwość stanu, `threadCards`, którą będzie lista komponentów `ThreadCard`.

Następnie używamy *hooka* `useEffect`, aby zmodyfikować listę komponentów `ThreadCard` w przypadku zmiany parametru `categoryId`. Jeśli dysponujemy prawidłowym identyfikatorem kategorii, to korzystamy z usługi `DataService`, aby pobrać listę wątków w danej kategorii, a następnie, na ich podstawie tworzymy listę komponentów `ThreadCard`. Ponadto określamy nazwę kategorii; używamy do tego pierwszego z pobranych wątków, choć we wszystkich nazwa kategorii będzie taka sama.

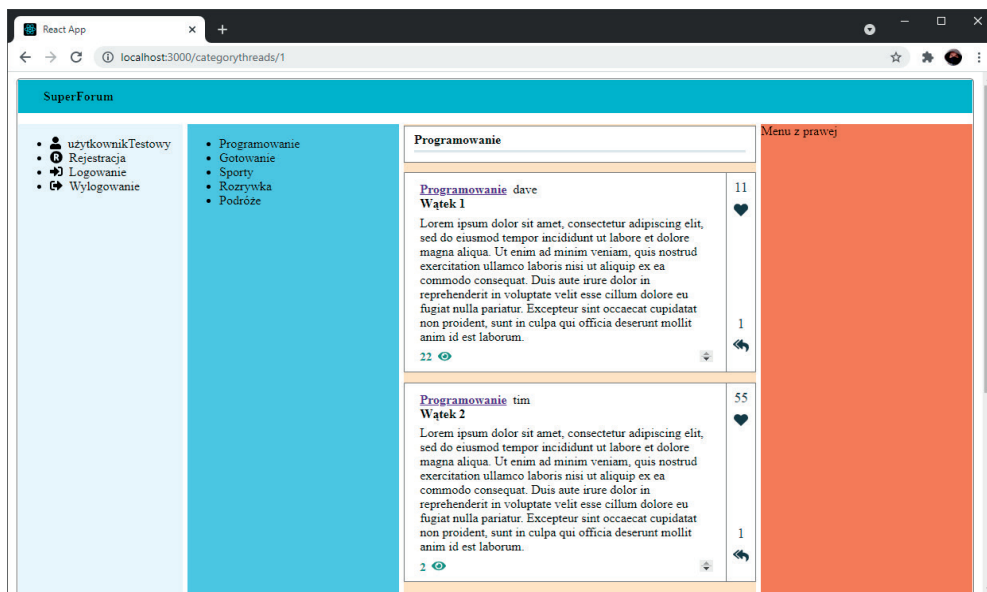
I w końcu zwracamy wygenerowany interfejs użytkownika.

Czasami może się zdarzyć, że React będzie wyświetlał komunikaty o brakujących zależnościach w tablicy przekazywanej w wywołaniu funkcji `useEffect`. Ja określam je jako „uparte ostrzeżenia”, a kiedy nabędziesz większego doświadczenia, będziesz w stanie samodzielnie oceniać, które z tych zależności bez zbędnego ryzyka będzie można zignorować. Na przykład, w wywołaniu `useEffect` w pliku *Main.tsx* celowo pominąłem ostrzeżenie o obiekcie stanu `category`, gdyż dołączenie go do tablicy spowodowałoby niepotrzebne, dwukrotne wywołanie `useEffect` (gdyż określony w niej kod jest wywoływany za każdym razem, gdy zmieni się dowolna z wartości podanych na liście), a co za tym idzie, także niebezpieczeństwo dwukrotnego renderowania komponentu.

8. A teraz spróbujmy uruchomić aplikację w trybie dla komputerów stacjonarnych.

W tym celu wpisz w przeglądarce adres <http://localhost:3000/categorythreads/1>.

Wyświetlona strona powinna być podobna do tej przedstawionej na rysunku 12.10.

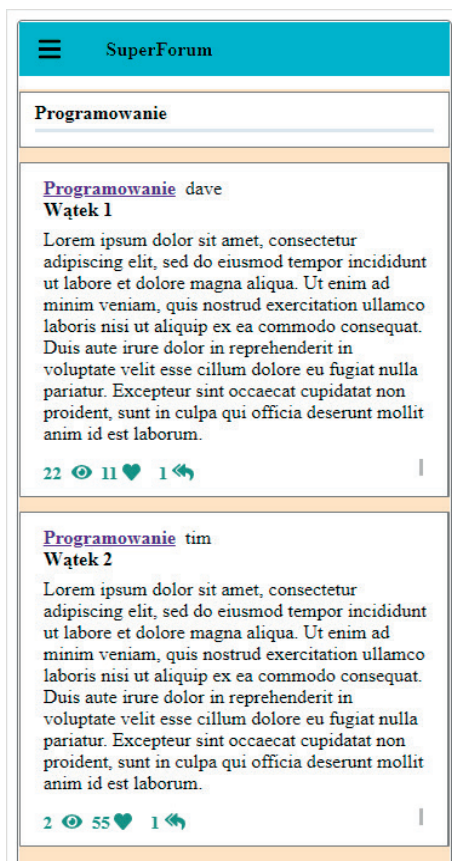


Rysunek 12.10. Ekran trasy `categorythreads` w trybie dla komputerów stacjonarnych

Z kolei na rysunku 12.11 pokazałem, jak ten sam ekran aplikacji będzie wyglądał na urządzeniach mobilnych.

Jak widać, w trybie dla urządzeń mobilnych nie jest wyświetlana prawa kolumna z punktami. Zamiast tego jej zawartość została przeniesiona poniżej głównego obszaru z tekstem. Jak pokazują ikony wyświetlone poniżej drugiego wpisu, 2 osoby go widziały, 55 go polubiło, a jedna osoba odpowiedziała.

To było naprawdę sporo kodu od dodania i modyfikacji! Ale to jeszcze nie wszystko! Teraz musimy się zająć komponentem `RightMenu`.



**Rysunek 12.11.** Ekran trasy categorythreads w trybie dla urządzeń mobilnych

W tym komponencie chcemy wyświetlać listę trzech kategorii, w których zostało opublikowanych najwięcej wpisów. W każdej z tych kategorii będziemy dodatkowo wyświetlać najczęściej oglądane wątki. A zatem, bierzmy się do pracy:

1. Zacznij od utworzenia w katalogu *areas* podkatalogu *rightMenu*, w którym docelowo będzie się znajdował komponent *RightMenu*.
2. Następnie, w tym nowym katalogu utwórz plik *TopCategory.tsx*. Ten komponent będzie reprezentować jedną kategorię oraz jej wątki.
3. Teraz musimy utworzyć nowy model, który będzie reprezentować dane pobierane z serwera. Nadamy mu nazwę *CategoryThread*. A zatem, w katalogu *models* utwórz plik *CategoryThread.ts* i skopiuj do niego kod z analogicznego pliku dostępnego w przykładach dołączonych do książki.
4. Teraz musimy zaktualizować istniejący komponent *RightMenu* i przygotować nowy, który będzie wyświetlał komponenty *CategoryThread*. Do pogrupowania i zorganizowania komponentów *CategoryThread* będziemy potrzebowali narzędzia o nazwie *Lodash*.

Lodash to biblioteka JavaScript, udostępniająca bardzo wiele funkcji pomocniczych; przedstawienie jej wszystkich możliwości w tej książce byłoby chyba niemożliwe. Niemniej jednak biblioteka ta jest szczególnie użyteczna w sytuacjach, gdy konieczne jest zarządzanie tablicami i kolekcjami. Samemu przekonasz się, że Lodash jest biblioteką bardzo prostą w użyciu, gdybyś jednak chciał dowiedzieć się czegoś więcej o jej możliwościach, to informacje o nich znajdziesz w jej dokumentacji, dostępnej na stronie <https://lodash.com/docs/>. Zainstaluj bibliotekę Lodash, używając następującego polecenia:

```
npm i lodash @types/lodash
```

Nigdy nie importuj całej zawartości biblioteki Lodash używając instrukcji o postaci `import _ from "lodash"`. W taki sposób dodasz bowiem do swojego projektu bardzo dużo kodu. Powinieneś importować wyłącznie wybrane funkcje, na przykład: `import groupBy from "lodash/groupBy"`.

Teraz zmodyfikuj plik *RightMenu.tsx*, kopiując do niego kod z przykładów dołączonych do książki.

W pierwszej kolejności zwróć uwagę na to, że oprócz funkcji z biblioteki Lodash importujemy także arkusz stylów *RightMenu.css*, zawierający proste style określające wygląd tego komponentu. Importujemy także komponent *TopCategory*, który przygotujemy już niebawem.

W dalszej części kodu tworzymy nowy obiekt stanu, *topCategories*, którego będziemy używać do przechowywania pobranych najpopularniejszych kategorii.

Następnie, w kodzie zdefiniowanym w wywołaniu funkcji *useEffect*, pobieramy kategorie, używając do tego celu funkcji *getTopCategories*. Pobrane obiekty grupujemy według kategorii, po czym tworzymy tablicę komponentów *TopCategory*. Elementy tych komponentów będą prezentowały dane kategorii w interfejsie użytkownika. Zwróć uwagę na to, że do każdego z tych komponentów przekazywana jest właściwość *props topCategory*, określająca grupę kategorii najwyższego poziomu.

Komponent zwraca następnie element *div* zawierający elementy *topCategories*.

5. Teraz musimy zająć się komponentem *TopCategory*. A zatem, w tym samym katalogu, w którym znajduje się komponent *RightMenu*, utwórz nowy plik o nazwie *TopCategory.tsx* i dodaj do niego kod z analogicznego pliku z przykładów dołączonych do książki.

Przed wszystkim zwróć uwagę na to, że używamy także pliku *TopCategory.css*, zawierającego style określające postać tego komponentu.

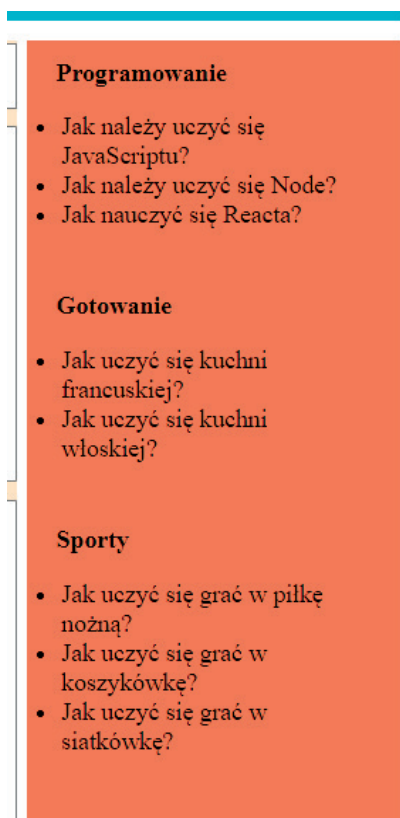
Poniżej instrukcji importu zdefiniowaliśmy nowy interfejs, *TopCategoryProps*, którego będziemy używać do pobierania właściwości *props*.

Następnie, w wierszu 10., tworzymy obiekt stanu *threads*, w którym później zapiszemy kod JSX komponentu.

W wierszu 12. wywołujemy funkcję `useEffect`, której użyjemy do wygenerowania, w oparciu o przekazaną właściwość *props* `topCategories`, elementów interfejsu użytkownika.

Zwracany przez komponent kod JSX zawiera element `strong`, prezentujący nagłówek z nazwą pierwszego znalezionej kategorii (wszystkie elementy będą miały tę samą nazwę kategorii). Poniżej tego nagłówka wyświetlamy listę wątków.

6. Komponent `RightMenu` nie jest renderowany w trybie dla urządzeń mobilnych, dlatego na rysunku 12.12 pokazałem, jak on wygląda w trybie dla komputerów stacjonarnych.



**Rysunek 12.12.** Komponent `RightMenu` z listą najpopularniejszych kategorii

W porządku, kolejny element aplikacji jest gotowy! W ten sposób dysponujemy już niemal wszystkimi elementami aplikacji tworzącymi jej ekran główny, musimy jednak zadbać o jeszcze jedną możliwość funkcjonalną: wyświetlanie **poszczególnych wątków**.



## Ekran wątku i jego wpisów

Ten ekran będzie realizował kilka zadań. Przede wszystkim, zapewni on użytkownikowi możliwość tworzenia nowych wpisów oraz wyświetlania wpisów już istniejących. Na tym samym ekranie będziemy także prezentować odpowiedzi do wpisu. A zatem, weźmy się do pracy:

1. W pierwszej kolejności musimy utworzyć nowy komponent trasy. W katalogu *routes* utwórz podkatalog *thread*, a w nim plik *Thread.tsx*. Jednak nasz komponent *Thread* będzie dość złożony, dlatego powinniśmy podzielić go na mniejsze, modularne fragmenty, nazywane komponentami podrzędnymi. W tym przypadku, zastosowanie takiego rozwiązania nie zapewni nam jednak korzyści, jaką jest możliwość wielokrotnego stosowania kodu. Z drugiej strony, użycie komponentów podrzędnych ułatwi analizę kodu oraz jego późniejszą refaktoryzację, gdyż będzie on podzielony na kilka małych fragmentów, a nie jednym wielkim monolitem. Utwórz zatem kolejny plik, *ThreadHeader.tsx*, i zapisz w nim poniższy fragment kodu:

```
import React, { FC } from "react";
import { getTimePastIfLessThanDay } from "../../common/dates";

interface ThreadHeaderProps {
  userName?: string;
  lastModifiedOn: Date;
  title?: string;
}

const ThreadHeader: FC<ThreadHeaderProps> = ({
  userName,
  lastModifiedOn,
  title,
}) => {
  return (
    <div className="thread-header-container">
      <h3>{title}</h3>
      <span>
        <strong>{userName}</strong>
        <label style={{ marginLeft: "1em" }}>
          {lastModifiedOn ? getTimePastIfLessThanDay(lastModifiedOn) : ""}
        </label>
      </span>
    </div>
  );
};

export default ThreadHeader;
```

W pierwszej kolejności zwróć uwagę na nową funkcję, którą importujemy: *getTimePastIfLessThanDay*. Funkcja ta pobiera datę i godzinę wpisu, po czym formatuje je odpowiednio, w celu zapewnienia możliwie jak najlepszej czytelności.

Ten komponent pobiera wszystkie niezbędne informacje przy użyciu parametrów i nie potrzebuje własnego stanu. Innymi słowy, *ThreadHeader* działa jako typowy

komponent prezentacyjny — wyświetla tytuł (`title`), nazwę użytkownika (`userName`) oraz czas ostatniej modyfikacji wątku (`lastModifiedOn`).

2. Teraz utwórz plik *Thread.tsx* i dodaj do niego kod źródłowy z analogicznego pliku dostępnego w przykładach dołączonych do książki.

Zwróć uwagę na to, że w pliku importujemy arkusz stylów *Thread.css* oraz nowy komponent `ThreadHeader`. Co więcej, zauważ także, że nasz komponent nosi nazwę `Thread`, czyli taką samą, co model; z tego względu model importujemy jako `ThreadModel`. W dużych projektach problemy tego typu mogą występować stosunkowo często, dlatego należy na nie zwracać uwagę podczas importowania zależności.

Kolejnym krokiem jest utworzenie lokalnego obiektu stanu o nazwie `thread`, typu `ThreadModel`. Pobieramy także parametr trasy `id`, określający identyfikator wątku; w tym celu wywołujemy funkcję `useParams`.

W kodzie przekazanym w wywołaniu funkcji `useEffect` sprawdzamy, czy parametr trasy `id` istnieje i czy jego wartość jest większa od 0, a jeśli oba te warunki są spełnione, to próbujemy pobrać wątek (`thread`). Na późniejszym etapie prac nad aplikacją, kiedy będzie już gotowa jej część serwerowa, napiszemy także kod pozwalający na dodawanie nowych wątków.

I w końcu zwracamy interfejs użytkownika, zawierający komponent `ThreadHeader`. Zwróć uwagę na to, że pole daty i godziny ostatniej modyfikacji musi być różne od `null`, dlatego używamy operatora trójargumentowego, by sprawdzić, czy `thread` wynosi `null`, i zwracamy aktualną datę, jeśli okaże się, że warunek ten jest spełniony.

3. Kolejnym krokiem będzie utworzenie nowej trasy na potrzeby komponentu `Thread` i określanego przez niego ekranu aplikacji. Otwórz plik *App.tsx* i zmodyfikuj go, by odpowiadał poniższemu przykładowi:

```
function App() {
  const renderHome = (props: any) => <Home {...props} />;
  const renderThread = (props: any) => <Thread {...props} />;
```

W powyższym fragmencie kodu definiujemy funkcję `renderThread`, której następnie użyjemy, by wyrenderować komponent `Thread`:

```
    return (
      <Switch>
        <Route exact={true} path="/" render={renderHome} />
        <Route
          path="/categorythreads/:categoryId"
          render={renderHome}
        />
        <Route
          path="/thread/:id"
          render={renderThread}
        />
      </Switch>
    );
  }
}
```

Zwróć uwagę na to, że trasa odwołująca się do komponentu Thread ma postać: `"/thread/:id"`, co oznacza, że na jej końcu powinien być podany parametr. React Router nada mu nazwę `id`.

4. W tym punkcie zajmiemy się dodaniem do ekranu komponentu Thread kolejnej sekcji. Na tym ekranie kategoria będzie wyświetlana w formie rozwijanej listy. Jednak standardowy element HTML zapewniający takie możliwości, czyli `select`, wygląda dość paskudnie i nie jest łatwo używać go w aplikacjach Reacta, dlatego skorzystamy z dodatkowego pakietu NPM o nazwie `react-dropdown`, udostępniającego kontrolkę bardziej atrakcyjną i zapewniającą lepszą integrację z Reactem.

Zainstaluj pakiet `react-dropdown`, używając następującego polecenia:

```
npm i react-dropdown
```

Następnie w katalogu `thread` utwórz nowy plik, `ThreadCategory.tsx`, i dodaj do niego kod źródłowy z pliku dostępnego w przykładach dołączonych do książki.

Po dodaniu niezbędnych instrukcji importu, tworzymy interfejs `ThreadCategoryProps`, opisujący właściwości `props` tego komponentu.

Następnie zaczynamy tworzyć kod komponentu `ThreadCategory`, a na jego samym początku definiujemy stałą `catOptions`, zawierającą wszystkie elementy, które będzie można wybrać z rozwijalnej listy. Także w tym przypadku wartości podajemy na stałe tylko tymczasowo, aż do momentu przygotowania części serwerowej aplikacji.

Na samym końcu komponentu zwracamy kod JSX zawierający odpowiednio zainicjowany komponent `DropDown`.

5. Kolejnym komponentem, którym się zajmiemy, będzie komponent `Title`. Nadamy mu nazwę `TitleComponent`. W katalogu `thread` utwórz plik `ThreadTitle.tsx` i skopiuj do niego kod z analogicznego pliku z przykładów dołączonych do książki.

To stosunkowo prosty komponent, którego działanie ogranicza się do zwrócenia odpowiedniego kodu JSX, dlatego nie będę go tu szczegółowo opisywał. Zwróć jednak uwagę na to, że póki co procedura obsługi zdarzeń `onChangeTitle` jest pusta. Także w tym przypadku, kiedy serwerowa część aplikacji będzie już gotowa, dodamy tu odpowiedni kod, który rozróżni, czy aplikacja jest w stanie odczytu, czy zapisu, zaimplementujemy także prawidłowy kod funkcji `onChangeTitle`.

6. Teraz wprowadźmy niezbędne modyfikacje w kodzie pliku `Thread.jsx` i przyjrzymy się, jak on aktualnie wygląda. Zmodyfikuj zawartość tego pliku tak, by był zgodny z kodem przedstawionym poniżej. Zwróć uwagę na to, że dodaliśmy do niego przygotowane wcześniej komponenty podrzędne powiązane z wątkami, a także zmodyfikowaliśmy zawartość pliku `Thread.css`.

Wywołania odpowiedzialne za przygotowanie stanu oraz kod przekazywany w wywołaniu funkcji `useEffect` nie zmieniły się, więc nie będę ich tu przedstawiał.

```
return (
  <div className="screen-root-container">
    <div className="thread-nav-container">
      <Nav />
    </div>
  </div>
)
```

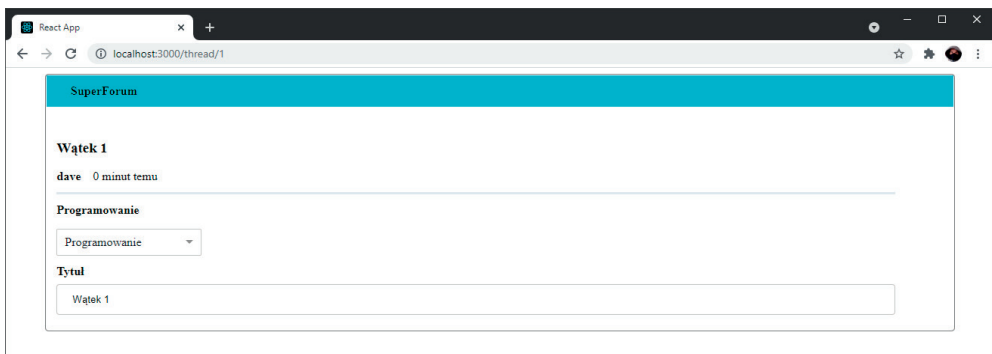
```

    </div>
    <div className="thread-content-container">
      <ThreadHeader
        userName={thread?.userName}
        lastModifiedOn={thread ? thread.lastModifiedOn : new Date()}
        title={thread?.title}
      />
      <ThreadCategory categoryName={thread?.category?.name} />
      <ThreadTitle title={thread?.title} />
    </div>
  </div>
);
};

```

W tym przykładzie dodaliśmy do zwracanego kodu JSX nasze nowe komponenty podrzędne. Jak widać, kod jest teraz krótszy i łatwiejszy do zrozumienia niż w przypadku, gdyby wszystkie elementy HTML i procedury obsługi zdarzeń znajdowały się w pliku *Thread.tsx*.

Jeśli wyświetlisz teraz w przeglądarce stronę <http://localhost:3000/thread/1>, zobaczysz stronę podobną do tej przedstawionej na rysunku 12.13.



**Rysunek 12.13.** Ekran wątku

Zwróć uwagę na odstęp po prawej stronie; to miejsce, w którym później dodamy informacje o liczbie polubień i odpowiedzi.

Nie będę tu przedstawiał wszystkich plików CSS, gdyż chciałbym się skoncentrować na kodzie, jednak jest to jeden z głównych ekranów naszej aplikacji, dlatego pobieżnie wyjaśnię, w jaki sposób przygotowałem jego układ i czym aktualnie dysponujemy. Zaktualizuj zatem swój plik *Thread.css*, byśmy analizowali ten sam arkusz stylów.

Podobnie jak zrobiliśmy wcześniej na ekranie *Home*, także i tu umieściliśmy elementy nawigacyjne w elemencie `div`, w którym zastosowaliśmy klasę `thread-nav-container`.

Z kolei klasy `thread-content-container` użyliśmy w elemencie `div`, w którym zostanie wyświetlona zawartość wątku. Jak możesz się przekonać, zastosowaliśmy tu układ siatki składający się z dwóch kolumn i nieokreślonej liczby wierszy.

Pozostała zawartość, dzięki zastosowaniu atrybutu `grid-column`, została dodana do pierwszej kolumny układu siatki. Drugą kolumnę, zawierającą punktację wątków (polubienia), przygotowujemy nieco później.

7. Teraz musimy zająć się dodaniem sekcji prezentującej treść wpisu. Sekcja ta będzie nieco bardziej skomplikowana, gdyż będziemy musieli w niej zastosować zaawansowany edytor tekstu. Kontrolka ta będzie pozwalać użytkownikom na formatowanie wpisywanych tekstów oraz wykonywanie bardziej wyszukanych operacji związanych z ich edycją.

W celu zapewnienia możliwości edycji wpisów, musimy zainstalować pakiet NPM o nazwie `Slate.js`. To właśnie on będzie pełnił rolę wyszukanego edytora. Będziemy musieli zainstalować także kilka wymaganych przez niego zależności, w tym pakiet o nazwie `Emotion` — bibliotekę pozwalającą na stosowanie stylów CSS bezpośrednio w kodzie JavaScript. Wykonaj zatem poniższe polecenie:

```
npm i slate slate-react slate-history emotion is-hotkey @types/is-hotkey
➔@types/slate @types/slate-react
```

Stosowanie pakietu `Slate` i implementacja zaawansowanego edytora są złożonymi zadaniami. Bez zbytniej przesady można by mu poświęcić odrębną książkę. Z tego względu postaram się przedstawić rozwiązanie możliwie jak najprostsze, choć — jak się niebawem samemu przekonasz — zaimplementowanie zaawansowanego edytora w prosty sposób jest niemal niemożliwe. A zatem, w katalogu `components` utwórz podkatalog o nazwie `editor`, a w nim utwórz plik `RichTextControls.tsx`. Ten plik będzie zawierał kontrolki, których będziemy używali w edytorze. Kod źródłowy, którego używam, pochodzi z projektu `Slate.js` i można go znaleźć na stronie <https://github.com/ianstormtaylor/slate/blob/master/site/components.tsx>. Ten kod jest stosunkowo rozbudowany, więc jego poszczególne fragmenty będę przedstawiał i wyjaśniał przy okazji stosowania poszczególnych kontroltek.

8. Teraz w tym samym katalogu `editor` utwórz plik `RichEditor.tsx` i dodaj do niego kod z analogicznego pliku z przykładów.

Na samym początku pliku umieszczona jest sekcja importu. Można w niej znaleźć standardowe instrukcje importu, charakterystyczne dla wszystkich aplikacji Reacta, jak również dwie instrukcje związane z biblioteką `Slate.js`. Ich zadaniem jest ułatwienie nam utworzenia interfejsu użytkownika edytora. Opiszę je bardziej szczegółowo w dalszej części rozdziału.

Importowana funkcja `isHotKey` jest narzędziem ułatwiającym tworzenie skrótów klawiszowych na potrzeby edytora.

Kolejny importowany element, `withHistory`, pozwala edytorowi zapisywać wprowadzane modyfikacje w odpowiedniej kolejności, dzięki czemu później, w razie konieczności, będzie je można odtworzyć.

`Button` i `ToolBar` to kontrolki, których można używać do tworzenia interfejsu użytkownika edytora. Plikiem `RichTextControls.tsx` zajmiemy się już niebawem.

Kolejne dwie instrukcje `import` importują ikony oraz arkusz stylów.

Zmienna `HOTKEYS` jest słownikiem zawierającym różne skróty klawiaturowe do formatowania. Zapis `[keyName: string]` oznacza, że fragment po lewej stronie reprezentuje klucz słownika, a fragment po prawej — wartość.

W wierszu 26. tworzymy zmienną `initialValue`. Wartością naszego edytora są obiekty, a nie łańcuchy znaków. Dlatego też zmienna `initialValue` reprezentuje obiekt wartości początkowej edytora. Wartością tej zmiennej jest tablica typu `Node`, zdefiniowanego w bibliotece `Slate.js`. W tym edytorze tekst jest reprezentowany w formie hierarchicznych drzew tworzonych z węzłów. Takie rozwiązanie daje pewność, że struktura tekstu pozostanie niezmienną, a jednocześnie pozwala na umieszczanie w strukturze danych nie tylko tekstu, lecz także informacji o jego formatowaniu. Można to sobie wyobrazić jako połączenie tekstu i metadanych.

Tablica `LIST_TYPES` umożliwia rozróżnianie, czy dany wpis jest akapitem, czy listą tekstów.

W wierszu 38. zaczynamy tworzyć komponent `RichEditor`. Jak już wcześniej zaznaczyłem, w przypadku edytora `Slate.js`, jego wartość, czyli tekst umieszczony w edytorze, nie jest zwykłym tekstem — jest to obiekt JSON, a jego element główny (określany także jako *korzeń*) jest typu `Node`. Innymi słowy, główna zawartość edytora, dostępna jako `value`, jest obiektem stanu zawierającym tablicę typu `Node`.

Kolejnym elementem kodu jest funkcja `renderElement`, która jest początkowo używana do wyświetlania większych fragmentów tekstu. `Element` to komponent reprezentujący większy fragment tekstu, składający się z kilku wierszy. Przygotowaniem tego komponentu zajmiemy się już niebawem.

Kolejna funkcja, `renderLeaf`, jest używana do wyświetlania niewielkich fragmentów tekstu. Utworzeniem komponentu `Leaf` także zajmiemy się nieco później.

Przypominam, że *hooki*, takie jak `useCallback` oraz `useMemo`, opisałem w rozdziale 5., pt. „Tworzenie aplikacji Reacta z wykorzystaniem *hooków*”.

W kolejnym wierszu tworzymy zmienną `editor`. Będzie ona zawierać komponent Reacta pobierający i wyświetlający tekst. Różni się on od takich komponentów jak `Slate`, `Toolbar` oraz `Editable`, będącymi jedynie opakowaniami edytora, wstrzykującymi lub modyfikującymi stosowane w tym edytorze sposoby formatowania tekstu.

Następnie używamy funkcji `useEffect`, by pobrać właściwość `props.existingBody` i zapisać ją w lokalnej właściwości stanu, oczywiście pod warunkiem, że wartość `existingBody` w ogóle została przekazana. Warto zwrócić uwagę, że właściwość `existingBody` jest przekazywana wyłącznie w trybie przeglądania, a nie w trybie tworzenia.

Procedura obsługi zdarzeń `onChangeEditorValue` zapisuje wartość w lokalnej właściwości stanu `value`, kiedy ta zmieni się w interfejsie użytkownika aplikacji. Także w tym przypadku zwróć uwagę na to, że typem wartości nie jest tekst, lecz tablica typu `Node`.

W wierszu 59. rozpoczyna się kod JSX komponentu. Na samym początku inicjalizujemy komponent Slate, przekazując do niego instancję editor, lokalny stan `value` oraz procedurę obsługi zdarzeń `onChange`.

Kolejnym elementem kodu JSX jest komponent `Toolbar`. Pochodzi on z pliku *RichTextControls.tsx* i reprezentuje kontener określający układ edytora i zawierający przyciski służące do formatowania tekstu. Pokazałem je na rysunku 12.14. Przeznaczenie komponentów `MarkButton` i `BlockButton` przedstawię później.



Rysunek 12.14. Przyciski komponentu `Toolbar` edytora `Slate.js`

Kontrolka `Editable` zawiera główne mechanizmy formatujące, klawisze skrótów oraz podstawowe ustawienia edytora.

Zwróć uwagę na to, że w celu poprawienia czytelności kodu większość możliwości funkcjonalnych edytora przenieśliem poza główny komponent.

W wierszu 92. rozpoczyna się kod kontrolki `MarkButton`. `MarkButton` to funkcja odpowiedzialna za generowanie interfejsu użytkownika przycisku oraz za skojarzenie tego przycisku z mechanizmem formatującym, który będzie wykonywany po jego kliknięciu. Ogólnie rzecz biorąc, oznaczenia tworzone przy użyciu przycisków tego typu służą do formatowania słów i znaków, a nie dłuższych bloków, takich jak całe zdania zajmujące wiele wierszy tekstu. Komponent `Button` pochodzi z pliku *RichTextControls.tsx* i reprezentuje przycisk na pasku narzędzi.

Kolejnym elementem kodu pliku *RichEditor.tsx* jest funkcja `isMarkActive`. Określa ona, czy konkretny mechanizm formatujący został już zastosowany, czy nie.

Kolejną funkcją zdefiniowaną w kodzie jest `toggleMark`. Służy ona do przełączania formatowania w zależności od tego, czy zostało ono już zastosowane, czy jeszcze nie. Kojarzy ona edytor z konkretnym formatem.

Funkcja `BlockButton` ustawia sposób formatowania bloku tekstu i tworzy przycisk, który będzie ten sposób formatowania reprezentował. Zazwyczaj blok tekstu składa się z wielu węzłów (reprezentowanych przez obiekty `Node`).

Funkcja `isBlockActive` określa, czy formatowanie zostało już zastosowane, czy jeszcze nie.

Funkcja `toggleBlock` przełącza formatowanie.

Kolejnym elementem kodu jest funkcja `Element`. Reprezentuje ona komponent określający jakiego typu kod HTML należy zastosować. Komponenty `Element` są bardzo często stosowane w edytorze `Slate.js`.

Ostatnim elementem kodu umieszczonym w pliku *RichEditor.tsx* jest komponent `Leaf`. Komponenty tego typu reprezentują mniejsze fragmenty zwracanego kodu HTML i także są bardzo często używane w edytorze `Slate.js`.

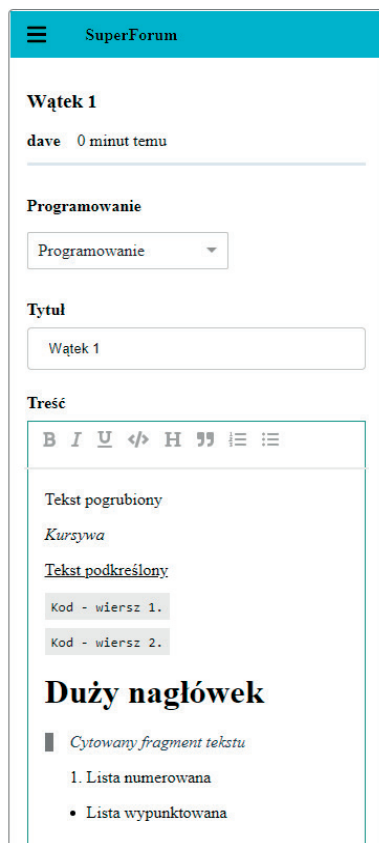
Dysponujemy już zaawansowanym edytorem tekstów. Będziemy go używać na ekranie wątku. Oczywiście, jest to niezależny komponent, co oznacza, że możemy go wielokrotnie używać w różnych miejscach kodu naszej aplikacji.

9. Teraz dodamy komponent `RichEditor` do pliku `ThreadBody.tsx`. `ThreadBody` to mały i prosty komponent, dlatego nie będę go tu przedstawiał — po prostu skopiuj plik `ThreadBody.tsx` z przykładów dołączonych do książki do katalogu `routes/thread`.
10. Kolejnym krokiem będzie umieszczenie odwołania do komponentu `ThreadBody` w komponencie `Thread`. W pierwszej kolejności upewnij się, że do pliku `Thread.tsx` zostaną dodane wszystkie niezbędne instrukcje importu; a następnie, w kodzie JSX, bezpośrednio poniżej elementu `ThreadTitle` umieść poniższy kod:

```
<ThreadBody body={thread?.body} />
```

Ponownie zwróć uwagę, jak czytelny i łatwy do zrozumienia jest kod JSX po zastosowaniu komponentów.

A teraz przyjrzyjmy się, jak wygląda ekran wątku w całości (rysunek 12.15).



Rysunek 12.15. Ekran wątku z widocznym edytorem



Nasz edytor udostępnia następujące opcje: pogrubienie, kursywę, podkreślenie, wyświetlenie tekstu jako kodu, duży nagłówek, cytat, listę numerowaną oraz listę wypunktowaną. Jak widać na rysunku, wszystkie te sposoby formatowania tekstu działają prawidłowo.

Byś może zastanawiasz się, dlaczego w edytorze Slate pojawiają się punktory list, choć wcześniej w kodzie CSS w pliku *index.css* wyłączyliśmy je. Otóż, aby uzyskać odpowiednią postać list w edytorze, zmodyfikowałem arkusz stylów, dodając do niego następującą regułę:

```
ul:not([data-slate-node="element"]) {
  list-style-type: none;
}
```

Ta reguła CSS nakazuje, by nie stosować podanych właściwości, jeśli element zawiera niestandardowy atrybut o nazwie `data-slate-node`. Slate.js używa tego atrybutu, by odróżniać swoje własne elementy od standardowego kodu HTML.

O rany! To było naprawdę sporo kodu! A i tak jeszcze nie dotarliśmy do końca. Wciąż pozostaje nam do utworzenia kolumna z punktami po prawej stronie ekranu wątku, wyświetlanie odpowiedzi oraz zapewnienie możliwości dodawania kolejnych elementów `ThreadItem`. Kolumnę z punktami zostawimy na później, a teraz zajmiemy się systemem odpowiedzi:

1. W pierwszej kolejności zajmiemy się wprowadzeniem drobnych modyfikacji w kodzie. W komponencie `ThreadHeader` wyświetlamy nazwę użytkownika (`userName`) oraz datę i godzinę ostatniej modyfikacji (`lastModifiedOn`), by pokazać użytkownikom, kto i kiedy utworzył dany wpis. Te same informacje możemy także wyświetlać prezentując odpowiedzi. Wyodrębnimy je zatem i umieścimy w osobnym komponencie, byśmy mogli go używać w różnych miejscach kodu. A zatem, w katalogu *routes/thread* utwórz nowy plik o nazwie *UserNameAndTime.tsx* i umieść w nim kod z przykładów dołączonych do książki. Nie będę tu prezentował tego kodu, gdyż właściwie stanowi on wierną kopię kodu z komponentu `ThreadHeader`.
2. Teraz możemy już użyć tego komponentu w kodzie komponentu `ThreadHeader`. A zatem, z kodu JSX komponentu `ThreadHeader` usuń element `span` znajdujący się poniżej nagłówka `h3`, po czym w tym samym miejscu umieść element `UserNameAndTime`. Nie zapomnij także o dodaniu na początku pliku odpowiedniej instrukcji `import`.

```
<UserNameAndTime userName={userName} lastModifiedOn={lastModifiedOn} />
```

Świetnie! Teraz możemy przejść do prac nad komponentem `ThreadItem` i odpowiedziami. Jednak tym razem zastosujemy nieco inne podejście. Otóż w przypadku odpowiedzi w wątku istnieje całkiem spore prawdopodobieństwo, że będzie ich więcej niż jedna. A zatem, rozwiązanie, które zastosujemy, będzie analogiczne do maszyn w fabryce widżetów. Będzie istnieć tylko jedna taka maszyna, choć aplikacja może potrzebować wielu widżetów. W sytuacjach takich jak ta, zasady projektowania aplikacji zalecają zastosowanie rozwiązania określanego jako wzorzec projektowy Fabryka.

A zatem, w rzeczywistości przygotujemy dwa komponenty. Pierwszy z nich będzie działać jak fabryka „konstruująca” odpowiedzi wyświetlane na ekranie wątku. Z kolei drugi komponent będzie definiować postać samych odpowiedzi. Przy okazji zwróć uwagę na to, że nie będziemy tu używać formalnego wzorca projektowego Fabryka, a jedynie jego przybliżony, koncepcyjny odpowiednik. Zaczynamy:

1. Musimy zacząć od komponentu `ThreadResponse`, który określi interfejs użytkownika oraz zachowanie obiektów `ThreadItem`. A zatem, w katalogu `routes/thread` utwórz plik *ThreadResponse.tsx* i skopiuj do niego kod z przykładów dołączonych do książki.

W pierwszej kolejności zwróć uwagę na to, że w jego kodzie importujemy (i ponownie ich używamy) przygotowane wcześniej komponenty `RichEditor` oraz `UserNameAndTime`. Wyobrażasz sobie, ile pracy wymagałoby odtworzenie ich w tym miejscu, gdybyśmy wcześniej nie zaimplementowali ich w formie komponentów? Teraz możemy w pełni docenić naszą wcześniejszą dalekowzroczność i przezorność!

Kolejnym fragmentem kodu w pliku *ThreadResponse.tsx* jest interfejs `ThreadResponseProps`. Zwróć uwagę na to, że niemal wszystkie jego właściwości są opcjonalne. To celowy zabieg, przygotowujący komponent do późniejszej refaktoryzacji i umożliwienia zastosowania go do tworzenia nowych odpowiedzi.

W dolnej części pliku znajduje się zwracany kod JSX. Interfejs użytkownika tego komponentu jest całkiem prosty — składa się jedynie z komponentu `UserNameAndTime` oraz `RichEditor`.

2. Teraz zajmiemy się fabryką komponentów `ThreadResponse`. W tym samym katalogu `routes/thread` utwórz plik *ThreadResponsesBuilder.tsx* i zapisz w nim kod skopiowany z przykładów dołączonych do książki.

Pierwszym elementem kodu w tym nowym pliku jest interfejs `ThreadResponsesBuilderProps`. Do tego komponentu będzie przekazywana właściwość `props` zawierająca listę obiektów `ThreadItem`. Będziemy musieli zmodyfikować komponent nadrzędny `Thread` tak, by przekazywał tę listę.

W wierszu 12. definiujemy jedyny obiekt stanu tego komponentu, `responseElements`. Nasz budowniczy może generować wiele odpowiedzi, a ten obiekt stanu będzie przechowywał je wszystkie w formie kodu JSX.

Kolejnym elementem kodu jest wywołanie funkcji `useEffect`, której używamy do utworzenia listy elementów odpowiedzi. Każda instancja komponentu `ThreadResponse` zawiera unikalny klucz, który chroni nas przed problemami z wyświetlaniem wielu komponentów tego samego typu. Za każdym razem, gdy zmieni się wartość właściwości `props threadItems`, wygenerujemy listę wypunktowaną (`ul`) zawierającą komponenty `ThreadResponse`.

Na samym końcu pliku zwracamy wynik, którym jest kod JSX zawierający listę elementów `ThreadResponse`.

3. To już prawie wszystko. Pozostało nam jeszcze tylko jedno zadanie: zmodyfikowanie kodu w pliku *Thread.tsx* i zastosowanie w nim komponentu *ThreadResponsesBuilder*. Zwróć także uwagę na zmiany arkuszy stylów w plikach *App.css* i *Thread.css*.

W kodzie JSX zwracany przez komponent, poniżej odwołania do *ThreadBody*, wpisz fragment kodu wyróżniony na poniższym przykładzie pogrubieniem:

```
return (
  <div className="screen-root-container">
    <div className="thread-nav-container">
      <Nav />
    </div>
    <div className="thread-content-container">
      <ThreadHeader
        userName={thread?.userName}
        lastModifiedOn={thread ? thread.lastModifiedOn : new Date()}
        title={thread?.title}
      />
      <ThreadCategory categoryName={thread?.category?.name} />
      <ThreadTitle title={thread?.title} />
      <ThreadBody body={thread?.body} />
      <hr className="thread-section-divider" />
      <ThreadResponsesBuilder threadItems={thread?.threadItems} />
    </div>
  </div>
);
```

Zwróć uwagę na to, że do kodu dodaliśmy element *hr*; ma on wizualnie oddzielić treść wątku od opublikowanych odpowiedzi.

Obecnie ekran wątku powinien wyglądać jak na rysunku 12.16.

Dysponujemy już niemal gotowym interfejsem użytkownika do wyświetlania wątków. Ale to wciąż nie wszystko. Pozostaje nam jeszcze przygotowanie prezentacji punktów i zapewnienie możliwości publikowania wątków i odpowiedzi. Teraz zajmiemy się prezentowaniem punktów, natomiast do możliwości publikowania wrócimy w późniejszych rozdziałach, kiedy przygotujemy niezbędne rozwiązania serwerowe. Co więcej, kiedy część serwerowa aplikacji będzie gotowa, łatwiej będzie zrozumieć, dlaczego pewne możliwości funkcjonalne zaimplementowaliśmy w taki, a nie inny sposób.

Jak zapewne pamiętasz z części rozdziału poświęconej implementacji trasy *categorythreads*, na ekranie kategorii używamy pionowego paska prezentującego liczbę polubień oraz odpowiedzi. Jeśli przyjrzyś się dokładnie kodowi odpowiedzialnemu za generowanie tych informacji, przekonasz się, że umieściliśmy go w funkcji o nazwie *getPointsNonMobile*. Z powodzeniem możemy jednak przenieść ten fragment kodu do osobnego komponentu Reacta. Oczywiście jest, że takiego komponentu będziemy mogli używać w komponencie *ThreadCard*, *Thread*, jak i w dowolnym innym miejscu kodu, w którym będziemy go potrzebować. A zatem, zaczynamy:

The screenshot shows a web interface for 'SuperForum'. At the top is a blue header with a hamburger menu icon and the text 'SuperForum'. Below the header, the main content area is white. It starts with a section titled 'Wątek 1' (Thread 1) in bold. Underneath, it says 'dave 0 minut temu' (dave 0 minutes ago). There is a horizontal line. Then, there's a section titled 'Programowanie' (Programming) in bold. Below it is a dropdown menu with 'Programowanie' selected. Another horizontal line follows. Then, there's a section titled 'Tytuł' (Title) in bold. Below it is a text input field containing 'Wątek 1'. Another horizontal line follows. Then, there's a section titled 'Treść' (Content) in bold. Below it is a rich text editor toolbar with buttons for bold (B), italic (I), underline (U), code (</>), link (H), quote (»), bulleted list (≡), and numbered list (≡). Below the toolbar is a text area containing the placeholder text 'Tu podaj treść wpisu.' (Here enter the content of the post). Another horizontal line follows. Then, there's a section titled 'Odpowiedzi' (Answers) in bold. Below it, there are two replies. The first reply is from 'jon 0 minut temu' (jon 0 minutes ago). It has a rich text editor toolbar identical to the one above, followed by a text area containing 'Odpowiedź 1 (ThreadItem)'. The second reply is from 'linda 0 minut temu' (linda 0 minutes ago). It also has a rich text editor toolbar, followed by a text area containing 'Odpowiedź 2 (ThreadItem)'. The text 'ThreadItem' in the second reply is underlined.

Rysunek 12.16. Wątek i opublikowane odpowiedzi

1. W katalogu *components* utwórz nowy plik, o nazwie *ThreadPointsBar.tsx*, i zapisz w nim kod z analogicznego pliku dostępnego w przykładach.

W wierszu 6. definiujemy typ właściwości *props* nowego komponentu. Być może zastanowi Cię, dlaczego nie zdecydowałem się na przekazywanie do komponentu całego obiektu *Thread*. Otóż wykorzystanie tylko wybranych składowych pozwala zapewnić lepszą separację odpowiedzialności. Gdybyśmy przekazali cały obiekt *Thread*, nie tylko pokazalibyśmy komponentowi *ThreadPointsBar* jakiego typu modelu używamy, lecz także dostarczylibyśmy mu informacji, których ani nie używa, ani nie potrzebuje.

Kod JSX zwracany przez komponent jest niemal identyczny jak kod funkcji `getPointsNonMobile`, gdyż robi dokładnie to samo. Teraz spróbuj usunąć z kodu komponentu `ThreadCard` funkcję `getPointsNonMobile` i zastosować w nim nowy komponent `ThreadPointsBar`. Zwróć uwagę na to, że wprowadziłem także nieznaczne zmiany w pliku *ThreadCard.css*, więc pamiętaj, żeby go odpowiednio zaktualizować. Nowa wersja ekranu powinna wyglądać dokładnie tak samo, jak wcześniejsza, gdyż wprowadzone zmiany polegały jedynie na przeniesieniu kodu z jednego komponentu do drugiego.

2. Teraz dodamy nasz nowy komponent do komponentu trasy `Thread`. Zmiany, jakie należy w tym celu wprowadzić w kodzie JSX, są niewielkie, lecz znaczące, dlatego dokładniej je tu przeanalizujemy. Nie zapomnij także zaktualizować pliku *Thread.css*.

```
return (
  <div className="screen-root-container">
    <div className="thread-nav-container">
      <Nav />
    </div>
    <div className="thread-content-container">
      <div className="thread-content-post-container">
```

W tym fragmencie kodu zmieniliśmy nieco kolejność elementów. Obecnie główne elementy związane z wątkiem są umieszczone w elemencie `div` należącym do klasy `thread-content-post-container`.

```
      <ThreadHeader
        userName={thread?.userName}
        lastModifiedOn={thread ? thread.lastModifiedOn : new Date()}
        title={thread?.title}
      />
      <ThreadCategory categoryName={thread?.category?.name} />
      <ThreadTitle title={thread?.title} />
      <ThreadBody body={thread?.body} />
    </div>
    <div className="thread-content-points-container">
```

W tym fragmencie kodu znajduje się zupełnie nowy element `div` należący do klasy `thread-content-points-container`, w którym umieściliśmy komponent `ThreadPointsBar`:

```
      <ThreadPointsBar
        points={thread?.points || 0}
        responseCount={
          thread && thread.threadItems && thread.threadItems.length
        }
      />
    </div>
  </div>
  <div className="thread-content-response-container">
    <hr className="thread-section-divider" />
    <ThreadResponsesBuilder threadItems={thread?.threadItems} />
  </div>
```

Także odpowiedzi umieściliśmy w osobnym elemencie `div` należącym do klasy `thread-content-response-container`.

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

A teraz przyjrzyjmy się zmodyfikowanej wersji arkusza stylów *Thread.css*, aby przekonać się, co się w nim zmieniło.

Blisko początku pliku zamieściłem definicję układu siatki (`grid-template-rows`). Składa się ona z dwóch wierszy, jednego przeznaczonego dla wpisu i drugiego dla odpowiedzi. Pierwszy wiersz zajmuje jedną część dostępnego obszaru, natomiast drugi może zająć tyle, ile będzie niezbędne (co jest możliwe dzięki zastosowaniu wartości `auto`), gdyż może w nim być wyświetlanych dowolnie wiele odpowiedzi, w tym żadna.

Poniżej zdefiniowana została nowa klasa, `thread-content-points-container`. Będzie nam ona potrzebna do zmiany układu komponentu `ThreadPointsBar`, który aktualnie jest nieco inny niż na ekranie głównym. Zwróć uwagę, że element tej klasy zostanie umieszczony w drugiej kolumnie układu i w jego pierwszym wierszu. Kolejna definicja stylu, zawierająca w selektorze wyrażenie `> div` oznacza, że elementowi `div` w komponencie `ThreadPointsBar` i w elemencie klasy `threadcard-points` należy przydzielić obszar o całej dostępnej wysokości.

Główne elementy wątku, takie jak `ThreadTitle` oraz `ThreadBody`, są umieszczone w elemencie klasy `thread-content-post-container`.

Z kolei odpowiedzi — czyli głównie komponent `ThreadResponsesBuilder` — są umieszczone w elemencie klasy `thread-content-response-container`. Zwróć uwagę na to, że w tej klasie wybrany jest drugi wiersz siatki (`grid-row` ma wartość 2).

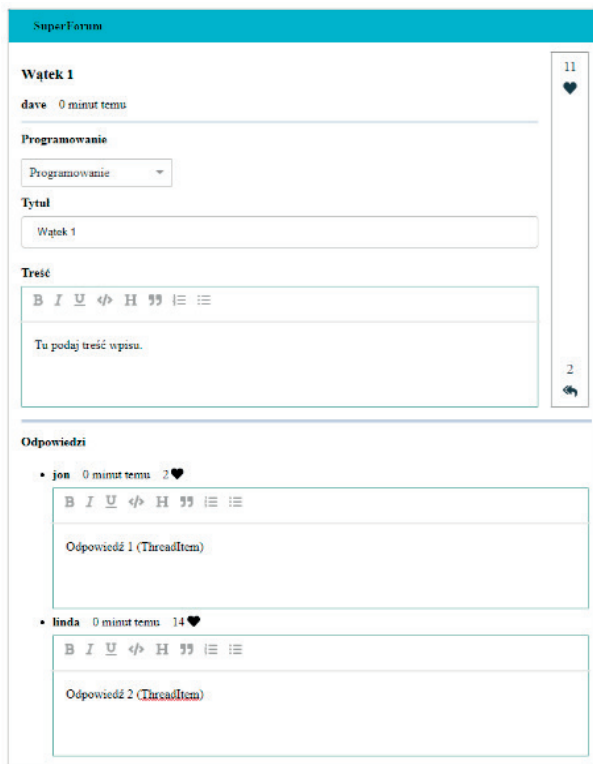
Łatwo można zauważyć, że wszystkie klasy powiązane z sekcjami umieszczone poniżej definicji klasy `thread-content-response-container` nie zawierają już właściwości określających położenie elementów w siatce, gdyż elementy, w których te klasy będą stosowane, będą umieszczone wewnątrz elementu klasy `thread-content-post-container`.

3. W naszym interfejsie użytkownika chcemy wyświetlać liczbę polubień dla każdej z prezentowanych odpowiedzi. Jednak tych odpowiedzi może być wiele, a wyświetlanie pionowego paska z liczbą polubień dla każdej z nich raczej nie będzie wyglądać atrakcyjnie. Dlatego, aby nieco poprawić przejrzystość interfejsu użytkownika, będziemy wyświetlać te punkty w tym samym wierszu, w którym jest umieszczona nazwa użytkownika i data publikacji. Na szczęście, większość kodu prezentującego te informacje w komponencie `ThreadCard` jest już gotowa i znajduje się w funkcji `getPoints`. Spróbujmy zatem także tę funkcję przekształcić na komponent.

Utwórz nowy plik, *ThreadPointsInline.tsx*, i zapisz w nim odpowiedni kod. W zasadzie kod tego pliku to prosty komponent Reacta, w którym należy umieścić kod z funkcji `getPoints`, więc jego konstrukcji i działania raczej nie trzeba wyjaśniać.

Zwróć jednak uwagę na to, że zastosowaliśmy w nim interfejs `ThreadPointsBarProps` z komponentu `ThreadPointsBar`. A to oznacza, że musimy wyeksportować ten interfejs.

Zakładam, że wiesz, jak zaktualizować komponent `ThreadCard.tsx`, gdyż zrobiliśmy to już wcześniej, przy okazji dodawania komponentu `ThreadPointsBar`. Teraz zaktualizuj zatem ten komponent tak, by używał nowego komponentu `ThreadPointsInline`. Spróbuj to zrobić samodzielnie i zajrzyj do kodu wyłącznie jeśli nie dasz sobie rady. Postać ekranu wątku po wprowadzeniu tych zmian przedstawiłem na rysunku 12.17.



Rysunek 12.17. Wyświetlanie punktów na ekranie wątku

Jak widać, obecnie są już wyświetlane oba systemy punktacji. Jest jeszcze jedna, końcowa sztuczka, którą chcemy wprowadzić, by ten ekran był prawidłowo wyświetlany na urządzeniach mobilnych.

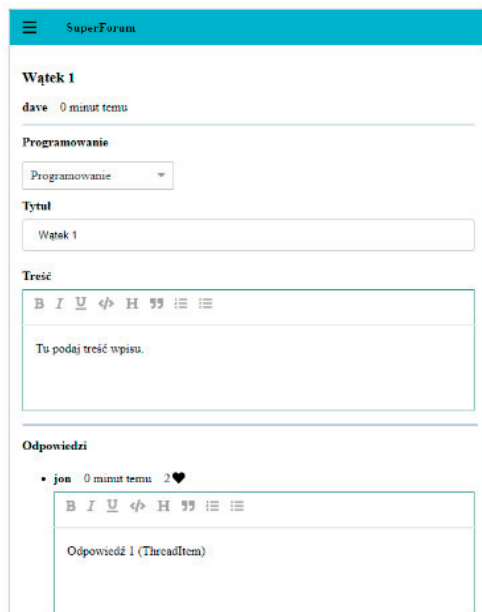
- Otwórz plik `Thread.css` i upewnij się, że zawiera on takie samo zapytanie medialne, które jest stosowane w analogicznym pliku w przykładach dołączonych do książki.

A teraz otwórz w edytorze kod komponentu `Thread`, byśmy mogli go wspólnie przeanalizować.

W wierszu 32. widać, że wszystkie elementy związane z wpisem wątku są umieszczone w elemencie `div` należącym do klasy `thread-content-container`. Dzięki zastosowaniu zapytania medialnego ta klasa CSS została zmodyfikowana tak, że na urządzeniach mobilnych zawiera **jedną** kolumnę. W ten sposób będziemy mieli pewność, że po usunięciu komponentu `ThreadPointsBar` z tego obszaru nie zostanie w nim puste miejsce — pozostałość po stosowanych wcześniej dwóch kolumnach siatki.

Teraz możemy się upewnić, że komponent `ThreadPointsBar` faktycznie jest umieszczony w elemencie klasy `thread-content-container`. Dzięki zastosowaniu zapytania medialnego, komponent ten jest ukrywany. Takie rozwiązanie wciąż jest efektywne, gdyż — jak zapewne pamiętasz — w kodzie komponentu `ThreadPointsBar` używamy *hooka* `useWindowDimensions` do określenia, czy komponent ten ma być renderowany, czy nie. Jak łatwo się przekonać, na urządzeniach mobilnych komponent ten nie będzie widoczny.

Super! Przekonajmy się zatem, jak będzie wyglądał ten ekran na urządzeniach mobilnych (patrz rysunek 12.18).



**Rysunek 12.18.** Postać ekranu Thread na urządzeniach mobilnych  
Fantastycznie! W ten sposób mamy jedną bazę kodu, która obsługuje dwa ekrany.

W ramach ostatniego zadania, którym zajmiemy się w tym rozdziale, zbudujemy ekran profilu użytkownika — `UserProfile`. Chcemy, by zapewniał on następujące możliwości:

- Pozwalał użytkownikom zmieniać hasło.
- Wyświetlał listę wszystkich utworzonych przez użytkownika wątków (`Thread`).
- Wyświetlał listę wszystkich opublikowanych przez użytkownika odpowiedzi (`ThreadItem`)



Bierzmy się zatem do pracy!

1. Pierwszą rzeczą, którą musimy się zająć, jest wprowadzenie modyfikacji w komponencie `SideBarMenu`. Musimy przenieść gdzieś wywołanie funkcji `useEffect`, by przesyłać użytkownika do `Reduxa`, a następnie do komponentu `Login`. Robimy to po to, by po poprawnym zalogowaniu, nowy obiekt użytkownika mógł zostać zapisany w magazynie `Redux`. Obecnie wprowadzenie zamiany tego typu nie powinno przysporzyć Ci większych problemów. A zatem, usuń to wywołanie z kodu komponentu `SideBarMenu` i przenieś do komponentu `Login`.

Upewnij się, że po przeniesieniu tego kodu do komponentu `Login` zmienisz nazwę `dispatch` na jakąś inną, gdyż w kodzie tego komponentu identyfikator `dispatch` jest już używany.

2. Nasz nowy ekran ma udostępniać możliwość resetowania hasła. Jednak zapewne pamiętasz, że już wcześniej napisaliśmy sporo kodu związanego z potwierdzaniem hasła na potrzeby komponentu `Registration`. Spróbujmy wyodrębnić ten kod i umieścić go w osobnym komponencie, co pozwoli nam używać go zarówno w komponencie `Registration`, jak i w nowym komponencie `UserProfile`.

W katalogu `component/auth/common` utwórz nowy plik, o nazwie `PasswordComparison.tsx`. Zapisz w nim kod z analogicznego pliku z przykładów dołączonych do książki.

Choć kod tego komponentu jest stosunkowo prosty, to jednak jest w nim kilka fragmentów, na które warto zwrócić uwagę. Przede wszystkim zauważ, że komponent ten nie używa reduktora `userReducer`, a zamiast tego pobiera niezbędne dane przy użyciu właściwości `props`. Szczególną uwagę zwróć na to, że jedną z tych właściwości jest funkcja `dispatch`. Funkcja ta należy do komponentu nadrzędnego i właśnie to sprawia, że komponent może używać wpisanych haseł wspólnie ze swoim komponentem nadrzędnym. Cały pozostały kod stanowi wierną kopię kodu, który wcześniej znajdował się w komponencie `Registration`.

A zatem, spróbuj teraz usunąć te same fragmenty z kodu komponentu `Registration`. Upewnij się także, że usuniesz z niego wszystkie niepotrzebne instrukcje `import`.

3. Teraz w katalogu `routes` utwórz nowy podkatalog, `userProfile`, a w nim plik `UserProfile.tsx`. Następnie skopiuj do tego pliku kod z analogicznego pliku dostępnego w przykładach.

W wierszu 14. kodu zaczyna się wywołanie `useReducer`, którego używamy, gdyż będziemy potrzebować kilku właściwości z informacjami o użytkowniku, takich jak `userName`. Pobieramy także reduktor użytkownika i tworzymy lokalne właściwości stanu, przeznaczone do przechowywania obiektów `Thread` i `ThreadItem`.

W wierszu 28. wywołujemy funkcję `useEffect`, a wewnątrz niej funkcję `getUser` ↪ `Threads` usługi danych `DataService`, która zwraca wątki bieżącego użytkownika. Nie potrzebujemy żadnego innego wywołania do pobrania obiektów `ThreadItem`, gdyż obiekt wątków (`Thread`) zawierają wszystkie powiązane z nimi obiekty odpowiedzi (`ThreadItem`). Niemniej jednak zmodyfikowałem klasę `ThreadItem` i dodałem do danych odpowiedzi identyfikator wątku, do którego dana odpowiedź należy (`threadId`).

Następnie w wierszu 38. wywołujemy funkcję `map`, przy użyciu której przekształcamy wszystkie zwrócone obiekty `Thread` na elementy `li`. Dodatkowo wszystkie obiekty `ThreadItem` dodajemy do tablicy, dzięki czemu później będziemy mogli ich używać.

W wierszu 53. w podobny sposób przekształcamy wszystkie obiekty `ThreadItem` na elementy `li`.

W wierszu 77. używamy przygotowanego wcześniej komponentu `PasswordComparison`.

W wierszu 82. jest umieszczony przycisk, w którego kodzie używamy właściwości `isSubmitDisabled`. Czy potrafisz odpowiedzieć na pytanie, dlaczego wyłączenie tego przycisku działa, chociaż w komponencie `UserProfile` nie ma żadnego kodu, który by je obsługiwał? Zgadłeś — zajmuje się tym komponent `PasswordComparison`, który w swoim kodzie używa funkcji `dispatch` przekazanej z komponentu `UserProfile`.

W dalszej części kodu używamy właściwości stanu `threads` i `threadItems`, by wyświetlić wątki i odpowiedzi.

4. W ramach ostatniej modyfikacji dodamy do pliku *App.tsx* nową trasę, odwołującą się do komponentu `UserProfile`. Zwróć uwagę na to, że do tego pliku musimy także tymczasowo dodać odwołanie do `Reduxa`, pobierające nazwę użytkownika. Usuniemy je później, kiedy analogiczne odwołanie umieszczone w pliku *Login.tsx* będzie już działać w docelowy sposób (a tym zajmiemy się, kiedy będzie gotowa serwerowa część aplikacji). Takie rozwiązanie jest konieczne, gdyż kiedy wczytujemy komponent `UserProfile` nie wiadomo, czy użytkownik zalogował się już używając komponentu `Login`, czy nie. Z kolei doskonale wiemy, że jeśli użytkownik wyświetlił dowolny ekran aplikacji, to musiał wczytać komponent *App.tsx*. Zaktualizuj kod w pliku *App.tsx* tak, by był zgodny z analogicznym plikiem umieszczonym w przykładach do książki.

Na początku kodu jest umieszczone wywołanie `useEffect`, które przesyła do magazynu `Reduxa` podaną na stałe nazwę użytkownika (`userName`). Także to rozwiązanie jest tymczasowe i usuniemy je, kiedy będzie gotowa serwerowa część aplikacji.

W wierszu 26. jest zdefiniowana funkcja `renderUserProfile`, która zwraca komponent `UserProfile`. Funkcja ta jest używana w wierszu 33. jako element docelowy nowej trasy: `"/userprofile/:id"`.

Musimy wprowadzić jeszcze jedną drobną modyfikację. W kodzie komponentu `SideBarMenus` zastąpimy etykietę z nazwą użytkownika (`userName`) odnośnikiem,

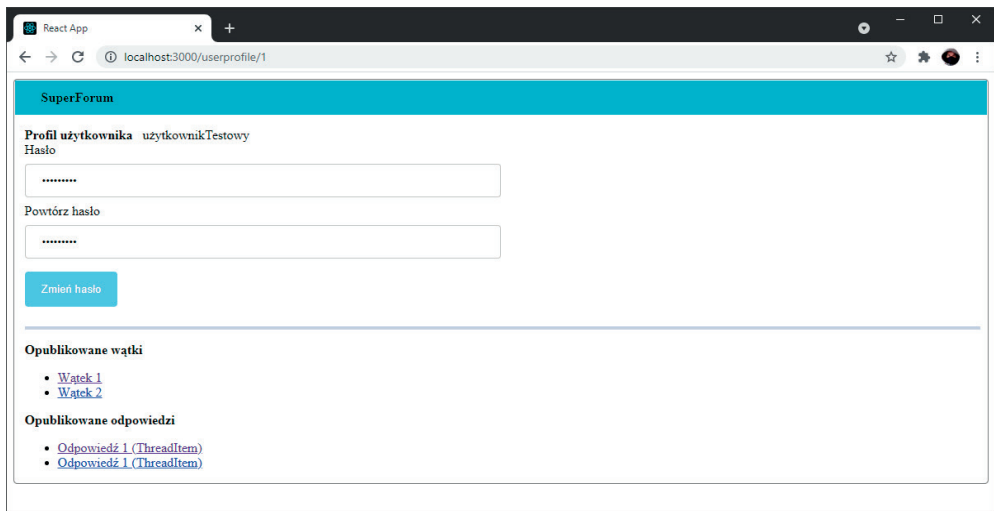
który przeniesie nas na ekran `UserProfile`. Kod JSX, o którym mowa, możesz znaleźć w pliku `SideBarMenus.tsx`:

```
<span className="menu-name">{user?.userName}</span>
```

Zastąp go następującym kodem:

```
<span className="menu-name">
  <Link to={`~/userprofile/${user?.id}`}>{user?.userName}</Link>
</span>
```

Kiedy teraz uruchomisz aplikację i przejdiesz na stronę profilu użytkownika, wyświetlona strona będzie taka, jak na rysunku 12.19.



Rysunek 12.19. Ekran profilu użytkownika

Kiedy klikniesz dowolny z odnośników umieszczonych w sekcji wątków lub odpowiedzi, przekonasz się, że działają one zgodnie z oczekiwaniami.

To jest naprawdę niesamowite! W tym rozdziale napisaliśmy naprawdę bardzo rozbudowany kod aplikacji Reacta. Dowiedziałeś się sporo o sposobach określania układów strony, poznałeś strukturę katalogów aplikacji, sposoby tworzenia komponentów, możliwości wielokrotnego stosowania kodu, sposoby refaktoryzacji kodu, określania postaci elementów prezentowanych na stronach, i poznałeś wiele innych zagadnień. Szczególnie czasochłonnym, a nawet stresującym zajęciem może być refaktoryzacja kodu. A niestety okazuje się, że w znaczniej większości przypadków nie będziemy pisać nowego kodu, a właśnie refaktoryzować już istniejący. Rozwijanie aplikacji przedstawionej w tym rozdziale było dobrą okazją, byś nieco powiększył swoje doświadczenia w tym zakresie.

W kilku następnych rozdziałach będziemy rozwijać serwerowe elementy naszej aplikacji i próbować powiązać je z przygotowanym tu klientem. Obecnie powinieneś już czuć się bardzo pewnie — przebrnięcie przez ten złożony rozdział wymagało naprawdę nie lada wysiłku.

## Podsumowanie

W tym rozdziale rozpoczęliśmy podróż, której celem jest stworzenie kompletnej aplikacji internetowej. Rozpoczęliśmy od napisania klienckiej aplikacji Reacta. Komponenty tworzyliśmy używając *hooków*, zaimplementowaliśmy hierarchię komponentów i zaprojektowaliśmy układ stron, korzystając z siatki CSS. Następnie poddaliśmy kod refektoryzacji, próbując w możliwie jak największym stopniu wielokrotnie używać komponentów. Choć nasz klient jeszcze nie jest gotowy, to jednak udało się nam już przygotować znaczącą część całej aplikacji.

W następnym rozdziale poznasz zagadnienia związane ze stanem sesji na serwerze, dowiesz się, czym jest ten stan sesji i jak go obsługiwać, oraz zaznajomisz się z najpopularniejszym narzędziem do tworzenia danych sesji i zarządzania nimi: bazą danych Redis.

# Przygotowywanie stanu sesji przy użyciu Expressa i Redisa

Z tego rozdziału dowiesz się, jak tworzyć i obsługiwać stan sesji przy użyciu Expressa i magazynu danych Redis. Redis jest jednym z najpopularniejszych obecnie magazynów przechowujących dane w pamięci. Jest on używany przez takie firmy jak Twitter, GitHub, Stack Overflow, Instagram oraz Airbnb. Także my zastosujemy połączenie Expressa i Redisa do zarządzania stanem sesji, który będzie stanowił podstawę mechanizmów uwierzytelniania w naszej aplikacji.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- wyjaśnieniem, czym jest stan sesji;
- przedstawieniem magazynu danych Redis;
- zaimplementowaniem obsługi stanu sesji przy użyciu Expressa i Redisa.

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś dysponować dobrą znajomością zagadnień związanych z tworzeniem aplikacji internetowych w środowisku Node. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retymo.zip>), a ich oryginalne, angielskojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial13 (Chap13)*.

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog, o nazwie *rozdzial13*.

## Czym jest stan sesji?

W tym podrozdziale opiszę to, czym jest stan sesji i wyjaśnię, dlaczego jest on niezbędny. Przy okazji przypomnę kilka pojęć związanych z działaniem WWW, co ułatwi wyjaśnienie, dlaczego potrzebujemy stanu sesji.

Tak naprawdę WWW nie jest jedną „rzeczą”. Stanowi połączenie wielu technologii. Jedną z najważniejszych spośród nich jest protokół HTTP. Jest to protokół komunikacyjny pozwalający na działanie WWW na internecie. Protokół to po prostu uzgodniony zestaw reguł opisujących sposób komunikacji. Być może brzmi to bardzo prosto, a w niektórych przypadkach nawet faktycznie jest to tak proste. Jednak w przypadku naszej aplikacji sprawy są nieco bardziej skomplikowane.

HTTP jest protokołem bezpołączeniowym. Oznacza to, że połączenia HTTP są tworzone wyłącznie na czas obsługi żądania, a następnie są zakańczane. A zatem, połączenie nie jest utrzymywane, nawet jeśli użytkownik aktywnie korzysta z witryny przez wiele godzin. Dzięki temu protokół HTTP jest bardziej skalowalny. Jednak z drugiej strony taki sposób działania sprawia, że niektóre z możliwości funkcjonalnych niezbędnych do działania dużych witryn jest znacznie trudniej zaimplementować.

Przyjrzyjmy się rzeczywistemu przykładowi. Wyobraźmy sobie przez chwilę, że jesteśmy Amazonem i naszą witrynę odwiedzają jednocześnie miliony użytkowników, próbujących coś kupić. Ponieważ ci użytkownicy próbują robić zakupy, musimy ich w unikalny sposób identyfikować. Na przykład, gdybyśmy jednocześnie próbowali robić zakupy w sklepie Amazona i gdybyś próbował dodawać do swojego koszyka jakieś produkty, to sklep musiałby zadbać o to, by żaden z produktów wybranych przez Ciebie nie trafił do mojego koszyka i na odwrót — aby moje produkty nie znalazły się w Twoim koszyku. Mogłoby się wydawać, że zagwarantowanie takiego działania sklepu powinno być bardzo proste. Okazuje się jednak, że ze względu na wykorzystanie protokołu bezpołączeniowego, takiego jak HTTP, jest ono całkiem trudne.

W przypadku protokołu HTTP każde żądanie tworzy nowe połączenie i żadne żądanie nie dysponuje jakimikolwiek informacjami o poprzednich. Innymi słowy, żądania nie zawierają żadnych danych stanu. A zatem, wracając do naszego przykładu z Amazonem, oznacza to, że jeśli użytkownik zażąda dodania produktu do koszyka, nie istnieją żadne wbudowane mechanizmy pozwalające na odróżnianie żądań przesyłanych przez tego użytkownika od żądań dowolnych innych użytkowników. Oczywiście możemy rozwiązać ten problem stosując własne rozwiązania, i właśnie nimi zajmiemy się w dalszej części rozdziału. Jednak chodzi o to, że WWW i protokół HTTP nie udostępniają nam domyślnie żadnych mechanizmów, które mogłyby nam pomóc w takich sytuacjach.

Dla jasności: istnieje bardzo wiele sposobów radzenia sobie z tym problemem. Moglibyśmy na przykład nadawać poszczególnym użytkownikom unikalne identyfikatory i przekazywać je w każdym żądaniu. Ewentualnie moglibyśmy zapisywać informacje o sesji w bazie danych i przechowywać w niej produkty dodane do koszyka. Bez wątpienia istnieje także wiele innych rozwiązań, które można by zastosować zależnie od konkretnych wymagań tworzonego projektu. Jednak te proste pomysły musiałyby zostać skonkretyzowane i uszczegółowione. Następnie musielibyśmy poświęcić sporo czasu na ich przetestowanie. Dlatego, podchodząc do problemu realistycznie, zawsze, kiedy to będzie możliwe, będziemy starali się unikać tworzenia własnych mechanizmów, a zamiast nich będziemy wybierać gotowe rozwiązania, uznawane za standardy przemysłowe. Korzystając z nich będziemy mieć pewność, że zostały one gruntownie przetestowane po względem niezawodności i bezpieczeństwa oraz że zostały utworzone zgodnie z najlepszymi praktykami.

Metoda rozróżniania użytkowników, którą zastosujemy, będzie kłaść nacisk na wykorzystanie technologii serwerowych, a konkretnie Expressa i magazynu danych Redis. Nie będziemy używać JWT (JSON Web Token), gdyż jest to technologia kliencka i jako taka jest bardziej narażona na zagrożenia bezpieczeństwa niż rozwiązania serwerowe.

Każde rozwiązanie ma swoje zalety i wady. Bez wątpienia, na każdy serwer można się włamać. A tworzenie zabezpieczeń w oparciu o rozwiązania serwerowe nie stanowi żadnej gwarancji bezpieczeństwa. Niemniej jednak, jeśli dysponujesz własnym serwerem, to przynajmniej możesz dołożyć starań, by go zabezpieczyć i kontrolować, a także by zagwarantować mu jak najwyższy poziom bezpieczeństwa.

Z tego podrozdziału dowiedziałeś się, czym jest stan sesji oraz dlaczego jego on niezbędny. Wskazałem w nim pewne brakujące możliwości protokołu HTTP oraz wyjaśniłem, w jaki sposób możemy sobie poradzić z tymi brakami. W następnym podrozdziale przedstawię pokrótce, czym jest Redis — magazyn danych, którego będziemy używali do przechowywania danych sesji.

## Przedstawienie magazynu danych Redis

W tym podrozdziale przedstawię magazyn danych Redis oraz opiszę, jak można go zainstalować. Pokrótce wyjaśnię także zasady jego działania i pokażę, jak można z niego korzystać.

Redis jest magazynem danych, który gromadzone informacje przechowuje w pamięci operacyjnej. Jest on niezwykle szybki i skalowalny. Redisa można używać do przechowywania łańcuchów znaków, list, zbiorów, itd. Redisa używają tysiące firm. Jest to produkt dostępny bezpłatnie, rozpowszechniany jako oprogramowanie typu *open-source*. Ogólnie rzecz biorąc, Redis jest najczęściej używany jako baza danych przechowująca dane w pamięci lub jako pamięć podręczna.

W naszej aplikacji będziemy używać Redisa jako magazynu danych dla sesji Expressa. Redis jest dostępny w wersjach dla systemów Linux i macOS, nie ma natomiast jego oficjalnej wersji dla systemu Windows. Co prawda można ominąć ten problem poprzez uruchomienie Redisa w systemie Windows jako obrazu Dockera, jednak przedstawienie takiego rozwiązania wykracza poza ramy tej książki. Na szczęście bez trudu można znaleźć dostawcę usług w chmurze, który udostępnia bezpłatne, linuksowe maszyny wirtualne, których można bezpłatnie używać przez jakiś czas w ramach testów. A zatem, jeśli używasz systemu Windows, to możesz poszukać jednej z takich usług.

W pliku konfiguracyjnym *redis.conf* znajduje się opcja `bind` określająca lokalny adres IP, którego będzie używać lokalny serwer Redisa, jak również decydująca o tym, jakie zewnętrzne adresy IP będą mogły nawiązywać z nim połączenie. Jeśli pozostawimy tę opcję zapisaną w komentarzu, będzie to oznaczało, że połączenie z serwerem można nawiązać z dowolnego adresu IP. Takie rozwiązanie dobrze nadaje się do prowadzenia prac programistycznych. Jednak w środowisku produkcyjnym należy w tej opcji podać konkretną wartość i zezwolić na dostęp do serwera wyłącznie określonym adresom IP.

Zacznijmy od zainstalowania Redisa. Poniżej opisałem proces jego instalacji na komputerach Mac:

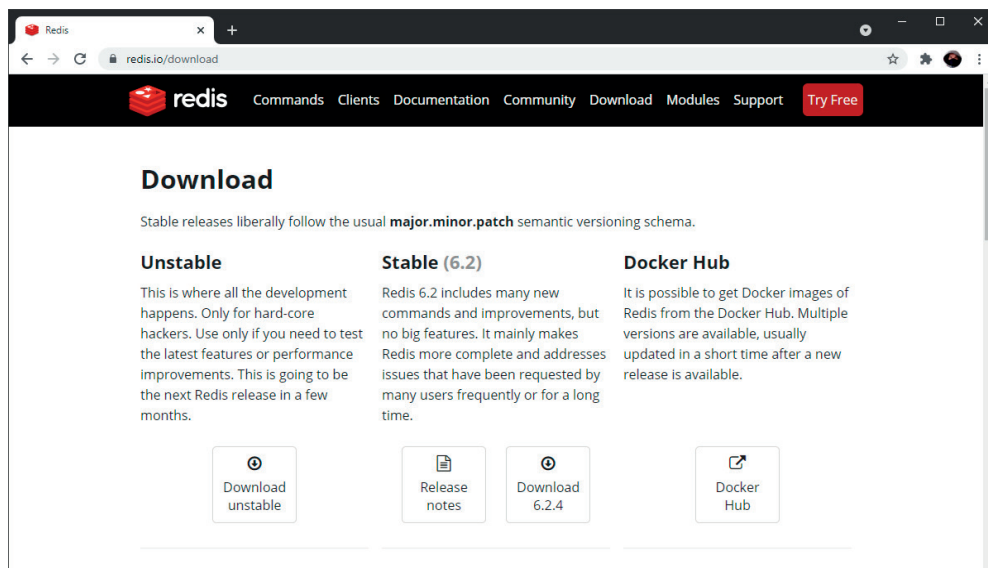
1. Przejdź na stronę Redisa, <https://redis.io/download>, i kliknij przycisk *Download* w sekcji *Stable*. Na rysunku 13.1 widać, że najnowszą stabilną wersją Redisa jest 6.2.4.

Wybierz wersję 6.0.x, gdyż wersje starsze lub nowsze mogą zawierać różnice, które uniemożliwią prawidłowe działanie naszej aplikacji<sup>1</sup>.

2. Po pobraniu pliku i jego rozpakowaniu w dowolnie wybranym katalogu, otwórz okno terminala i przejdź w nim do tego katalogu. Na rysunku 13.2 pokazałem, jak wyglądało okno terminala po rozpakowaniu pliku *tar* na moim komputerze.

<sup>1</sup> W czasie przygotowywania polskiego wydania niniejszej książki, na głównej stronie do pobierania Redisa był umieszczony bezpośredni odnośnik pozwalający na pobranie wersji 6.0.14. Gdyby jednak z jakichś powodów został on usunięty, to na stronie <https://download.redis.io/releases/> dostępna jest list wszystkich wersji Redisa wraz z odnośnikami pozwalającymi na ich pobranie — *przyp. tłum.*





Rysunek 13.1. Strona umożliwiająca pobranie Redisa

```
davidchoi@Davids-MacBook-Pro ~ % cd Downloads
davidchoi@Davids-MacBook-Pro Downloads % cd redis-6.0.7
davidchoi@Davids-MacBook-Pro redis-6.0.7 % ls
00-RELEASENOTES      README.md             runtest-sentinel
BUGS                  TLS.md                sentinel.conf
CONTRIBUTING         deps                  src
COPYING               redis.conf            tests
INSTALL               runtest               utils
MANIFESTO             runtest-cluster
Makefile              runtest-moduleapi
```

Rysunek 13.2. Zawartość katalogu po rozpakowaniu stabilnej wersji Redisa

3. Teraz musisz wykonać polecenie `make`, aby przygotować wykonywalną aplikację na podstawie kodów źródłowych. A zatem, wpisz w terminalu `make` i poczekaj na wykonanie polecenia — to może trochę potrwać. Na rysunku 13.3 pokazałem początkowy fragment wyników generowanych przez to polecenie.
4. Po wykonaniu polecenia serwer będzie gotowy i będziesz mógł go przenieść w dowolnie wybrane miejsce. Ja umieściłem go w katalogu *Applications*. Następnie, aby uruchomić serwer, będziesz musiał przejść do katalogu Redisa i wykonać w nim następujące polecenie:

**src/redis-server**

Poniżej, na rysunku 13.4, pokazałem ekran terminala po uruchomieniu lokalnego serwera Redis.

```
davidchoi@Davids-MacBook-Pro redis-6.0.7 % make
cd src && /Library/Developer/CommandLineTools/usr/bin/make all
/bin/sh: pkg-config: command not found
  CC Makefile.dep
/bin/sh: pkg-config: command not found
rm -rf redis-server redis-sentinel redis-cli redis-benchmark redis-check-rdb redis-
check-aof *.o *.gcda *.gcno *.gcov redis.info lcov-html Makefile.dep dict-benchmark
rm -f adlist.d quicklist.d ae.d anet.d dict.d server.d sds.d zmalloc.d lzf_c.d lzf_
d.d pqsort.d zipmap.d sha1.d ziplist.d release.d networking.d util.d object.d db.d
replication.d rdb.d t_string.d t_list.d t_set.d t_zset.d t_hash.d config.d aof.d pu
bsub.d multi.d debug.d sort.d intset.d syncio.d cluster.d crc16.d endianconv.d slow
log.d scripting.d bio.d rio.d rand.d memtest.d crcspeed.d crc64.d bitops.d sentinel
.d notify.d setproctitle.d blocked.d hyperloglog.d latency.d sparkline.d redis-chec
k-rdb.d redis-check-aof.d geo.d lazyfree.d module.d evict.d expire.d geohash.d geoh
ash_helper.d childinfo.d defrag.d siphash.d rax.d t_stream.d listpack.d localtime.d
lolwut.d lolwut5.d lolwut6.d acl.d gopher.d tracking.d connection.d tls.d sha256.d
timeout.d setcpuaffinity.d anet.d adlist.d dict.d redis-cli.d zmalloc.d release.d
ae.d crcspeed.d crc64.d siphash.d crc16.d ae.d anet.d redis-benchmark.d adlist.d di
ct.d zmalloc.d siphash.d
(cd ../deps && /Library/Developer/CommandLineTools/usr/bin/make distclean)
(cd hiredis && /Library/Developer/CommandLineTools/usr/bin/make clean) > /dev/null
|| true
(cd linenoise && /Library/Developer/CommandLineTools/usr/bin/make clean) > /dev/nul
l || true
(cd lua && /Library/Developer/CommandLineTools/usr/bin/make clean) > /dev/null || t
rue
(cd jemalloc && [ -f Makefile ] && /Library/Developer/CommandLineTools/usr/bin/make
distclean) > /dev/null || true
(rm -f .make-*)
(rm -f .make-*)
echo STD=-std=c11 -pedantic -DREDIS_STATIC='' >> .make-settings
echo WARN=-Wall -W -Wno-missing-field-initializers >> .make-settings
echo OPT=-O2 >> .make-settings
```

Rysunek 13.3. Wykonywanie polecenia make

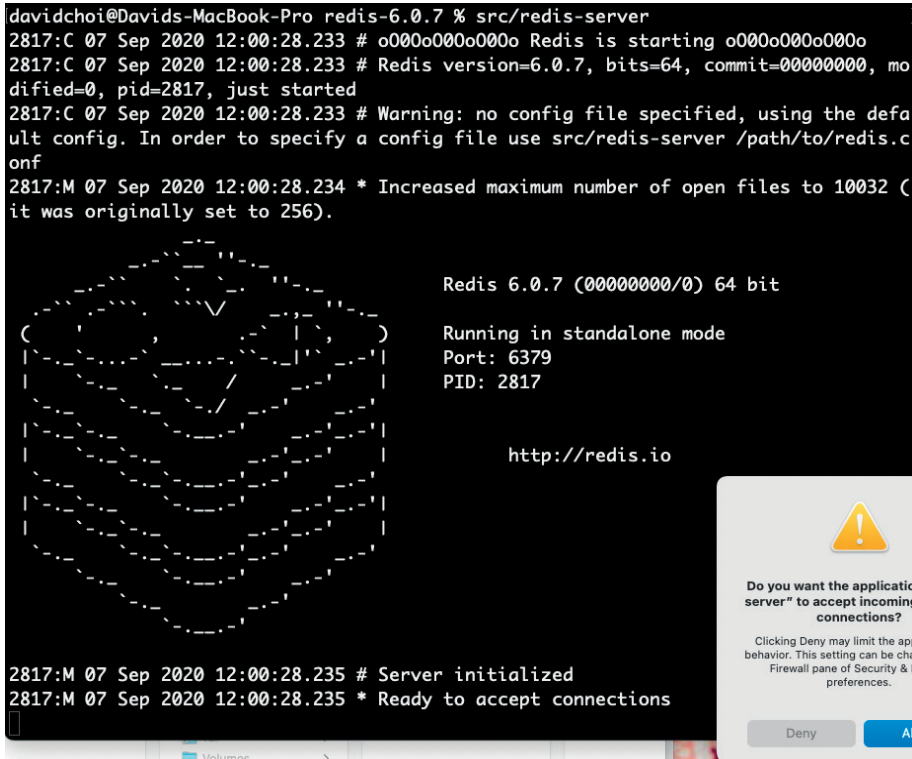
Na komputerach Mac może zostać wyświetlone ostrzeżenie z pytaniem, czy chcesz nadać serwerowi Redisa uprawnienie do akceptowania odbieranych połączeń sieciowych. Powinieneś na to pozwolić.

5. A teraz wykonaj szybki test, by upewnić się, czy Redis działa. Po wcześniejszym uruchomieniu Redisa otwórz nowe okno terminala, przejdź w nim do katalogu `src` w tym katalogu, w którym umieściłeś Redisa, i wykonaj następujące polecenie:

```
redis-cli
```

Na rysunku 13.5 pokazałem test, który pozwoli sprawdzić działanie Redisa.

Jak widać na rysunku 13.5, najpierw wykonujemy polecenie `ping`, by sprawdzić, czy serwer Redisa działa. W kolejnym kroku wykonujemy polecenie `set`, aby utworzyć nowy element o kluczu `test` i wartości `1`. Następnie pobieramy ten element, używając polecenia `get`.



```
davidchoi@Davids-MacBook-Pro redis-6.0.7 % src/redis-server
2817:C 07 Sep 2020 12:00:28.233 # o000o000o000o Redis is starting o000o000o000o
2817:C 07 Sep 2020 12:00:28.233 # Redis version=6.0.7, bits=64, commit=00000000, modified=0, pid=2817, just started
2817:C 07 Sep 2020 12:00:28.233 # Warning: no config file specified, using the default config. In order to specify a config file use src/redis-server /path/to/redis.conf
2817:M 07 Sep 2020 12:00:28.234 * Increased maximum number of open files to 10032 (it was originally set to 256).

Redis 6.0.7 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 2817

http://redis.io

2817:M 07 Sep 2020 12:00:28.235 # Server initialized
2817:M 07 Sep 2020 12:00:28.235 * Ready to accept connections
```

Do you want the application "redis-server" to accept incoming network connections?

Clicking Deny may limit the application's behavior. This setting can be changed in the Firewall pane of Security & Privacy preferences.

Deny Allow

Rysunek 13.4. Działający serwer Redis

```
Last login: Mon Sep  7 11:43:39 on ttys001
davidchoi@Davids-MacBook-Pro ~ % /Applications/redis-6.0.7/src/redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> set test 1
OK
127.0.0.1:6379> get test
"1"
```

Rysunek 13.5. Sprawdzenie działania Redisa

- Skoro już wiesz, że serwer działa prawidłowo, musisz wprowadzić pewne drobne zmiany w jego konfiguracji. W pierwszej kolejności zamknij serwer, wykonując w tym celu następujące polecenie:

```
src/redis-cli shutdown
```

Po zamknięciu serwera, przejdź do katalogu z kodami źródłowymi do rozdziału 13. i skopiuj całą zawartość pliku *redis/redis.conf*. Następnie w oknie terminala wykonaj poniższe polecenie:

```
sudo mkdir /etc/redis
```

Jeśli zostaniesz poproszony przez polecenie `sudo` o podanie hasła, wpisz je. Katalog, który właśnie utworzyłeś, jest domyślnym miejscem, w którym jest przechowywana znaczna większość plików konfiguracyjnych Redisa. Teraz wykonaj poniższe polecenie:

```
sudo nano /etc/redis/redis.conf
```

Nano jest prostym edytorem tekstów działającym w oknie terminala. Kiedy edytor zostanie uruchomiony, wklej od niego skopiowany kod źródłowy i zapisz nowy plik `/etc/redis/redis.conf`.

Jeśli przejrzysz ten plik lub wyszukasz w nim słowa `requirepass` (w edytorze Nano możesz to zrobić naciskając kombinację klawiszy `Ctrl+W`; ewentualnie możesz także przejrzeć zawartość pliku w VSCode), znajdziesz hasło, którego będziemy używali wyłącznie do celów testowych. Absolutnie nie używaj tego hasła w środowisku produkcyjnym!

Dla wszystkich pozostałych ustawień konfiguracyjnych możesz zostawić wartości domyślne.

7. Teraz uruchom serwer Redisa, wskazując przy tym nowy plik konfiguracyjny `redis.conf`. Użyj następującego polecenia:

```
src/redis-server /etc/redis/redis.conf
```

Zwróć uwagę na to, że tym razem zostanie wyświetlony komunikat: `Configuration loaded2`.

Pamiętaj także, że teraz chcąc przetestować działanie serwera, będziesz musiał przeprowadzić uwierzytelnianie, gdyż w ustawieniach konfiguracyjnych zostało podane hasło dostępu:

```
src/redis-cli  
auth <hasło>
```

Cały proces będzie wyglądał tak, jak pokazałem na rysunku 13.6.

```
davidchoi@Davids-MacBook-Pro redis-6.0.7 % src/redis-cli  
127.0.0.1:6379> auth test-password-do-not-use-123  
OK  
127.0.0.1:6379> ping  
PONG
```

**Rysunek 13.6.** Testowe uruchomienie serwera Redis i autoryzacja dostępu

W tym podrozdziale opisałem, czym jest Redis i przedstawiłem prosty sposób jego instalacji. W następnym podrozdziale zaczniemy tworzenie kodu serwerowej części naszej aplikacji, a konkretnie przygotujemy serwer korzystający z Node i Expressa, wyposażony dodatkowo w sesje przechowywane w magazynie Redis.

<sup>2</sup> Konfiguracja wczytana — *przyp. tłum.*

# Tworzenie stanu sesji z wykorzystaniem Expressa i Redisa

W tym podrozdziale zaczniemy tworzyć serwerową część naszej aplikacji. W tym celu utworzymy projekt Expressa i przygotowujemy Redisa do przechowywania stanu sesji.

Skoro już rozumiesz, czym jest Redis i wiesz, jak go zainstalować, mogę wyjaśnić, w jaki sposób Express i Redis będą współdziałać na naszym serwerze. Jak już wspominałem w rozdziale 8., pt. „Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa”, Express jest właściwie jedynie opakowaniem ułatwiającym korzystanie z niektórych możliwości środowiska Node. To opakowanie pozwala na wzbogacanie możliwości środowiska Node poprzez stosowanie różnego rodzaju oprogramowania pośredniego. Ponadto takie oprogramowanie pośrednie zapewnia możliwość obsługi stanu sesji.

W naszej aplikacji Express będzie dostarczał obiekt sesji wyposażony w odpowiednie możliwości, takie jak tworzenie ciasteczek przechowywanych w przeglądarce użytkownika, oraz funkcje służące do przygotowywania i utrzymania sesji. Redisa będziemy natomiast używali do przechowywania danych sesji. Ponieważ Redis jest bardzo szybki, stosowanie go do przechowywania sesji jest bardzo dobrym przykładem jego użycia.

Utwórzmy teraz projekt, w którym użyjemy Expressa i Redisa:

1. W pierwszej kolejności musisz utworzyć katalog na pliki projektu, nadaj mu nazwę *super-forum-server*. Następnie zainicjuj projekt NPM (pamiętaj, by wcześniej przejść w panelu terminala do katalogu *super-forum-server*); w tym celu wykonaj następujące polecenie:

```
npm init -y
```

Po zainicjowaniu projektu wyświetl w VSCode plik *package.json* i zmień wartość pola *name* na *super-forum-server*. Możesz także podać swoje imię i nazwisko w polu *author*.

2. Kolejnym krokiem będzie zainstalowanie wszelkich niezbędnych zależności:

```
npm i express express-session connect-redis ioredis dotenv
npm i typescript @types/express @types/express-session @types/connect-redis
@types/ioredis ts-node-dev -D
```

Jak widać, zainstalowaliśmy nie tylko pakiet *express*, lecz także *express-session*. To właśnie ten pakiet odpowiada za obsługę sesji w aplikacjach Expressa. Oprócz tego zainstalowaliśmy także pakiet *connect-redis*, który pozwoli na nawiązanie połączenia z magazynem danych Redis. Oprócz *connect-redis* potrzebny nam będzie także pakiet *ioredis* — klient, który zapewni możliwość dostępu do samego serwera Redisa. Znaczenie obu tych pakietów wyjaśnię dokładniej nieco później, kiedy zaczniemy pisać kod. I w końcu ostatni pakiet, *dotenv*, zapewnia możliwość korzystania z plików konfiguracyjnych, *.env*, w których można przechowywać takie informacje, jak hasła dostępu do serwera, czy też inne ustawienia konfiguracyjne.

Z kolei w drugim z powyższych poleceń `npm install` podane zostały niezbędne pakiety konieczne do prowadzenia prac programistycznych; przy czym głównie są to pakiety z definicjami typów języka TypeScript, takie jak `@types/express`. Jednak zwróć uwagę na to, że na samym końcu instalujemy także pakiet `ts-node-dev`. Ten pakiet pomoże nam uruchamiać serwer z poziomu głównego pliku `index.ts`. Pakiet ten pozwoli nam uruchamiać kompilator TypeScriptu, programu `tsc`, jak również przygotowywać i uruchamiać końcowy serwer naszej aplikacji.

Nigdy nie zapisuj swojego pliku konfiguracyjnego `.env`, zarządzanego przy użyciu pakietu `dotenv`, w repozytorium Git. W tym pliku mogą się bowiem znajdować ważne i wrażliwe informacje. Powinieneś opracować proces zarządzania tym plikiem i udostępniania go innym programistom bez korzystania z internetu.

3. Teraz zajmiemy się zmodyfikowaniem pliku `package.json` i zastosowaniem w nim narzędzi pomocniczych udostępnianych przez pakiet `ts-node-dev`. Pakiet ten jest niezwykle użyteczny, gdyż pozwala także na automatyczne restartowanie serwera za każdym razem, gdy zmieni się jego kod źródłowy. A zatem, w sekcji `scripts` pliku `package.json` dodaj poniższy wiersz kodu:

```
"start": "ts-node-dev --respawn src/index.ts"
```

Zwróć uwagę na to, że przed opcją `respawn` są umieszczone dwa znaki minusa. `Index.ts` będzie głównym plikiem uruchamiającym nasz serwer.

4. Kolejnym zadaniem jest takie skonfigurowanie kompilatora języka TypeScript, by jego działanie odpowiadało potrzebom naszego projektu. Plik konfiguracyjny TypeScriptu, `tsconfig.json`, przedstawiałem już wcześniej wiele razy, więc nie będę go tu pokazywał (oczywiście, jest on dostępny w kodach źródłowych dołączonych do książki). Zwróć tylko uwagę na to, że docelową wersję generowanego kodu JavaScript (opcja `target`) jest ES6, a wygenerowane kody są umieszczane w katalogu `./dist`.
5. W głównym katalogu projektu utwórz podkatalog `src`.
6. Teraz utwórz plik `.env` i zapisz w nim potrzebne informacje. Skopiuj wszystkie ustawienia z tabeli 13.1, przy czym dla każdego z nich podaj własne, unikalne dane.
7. Następnie utwórz plik `index.ts`. Zaczniemy od przygotowania minimalnego pliku, tylko po to, by upewnić się, że serwer działa. Zapisz w pliku następujący wiersz kodu:

```
import express from "express";
```

W tym wierszu importujemy Expressa.

```
console.log(process.env.NODE_ENV);
```

To wywołanie wyświetla informacje o tym, którego środowiska używamy: produkcyjnego, czy do prowadzenia prac programistycznych. Jeśli jeszcze nie skonfigurowałeś swojego lokalnego środowiska, to możesz to zrobić teraz, wykonując w panelu terminala odpowiednie polecenie, zależnie od systemu operacyjnego, którego używasz.

Tabela 13.1. Dane konfiguracyjne

Ustawienie	Przeznaczenie
REDIS_PASSWORD	Hasło podane wcześniej w pliku konfiguracyjnym Redisa, <i>redis.conf</i> . Musisz tu podać dokładnie to samo hasło, którego użyłeś wcześniej.
REDIS_PORT	Numer portu, na którym działa Redis. Domyślnie jest to port 6379.
REDIS_HOST	Adres IP serwera Redisa. Ponieważ zainstalowaliśmy go na lokalnym komputerze, należy tu podać <code>localhost</code> .
COOKIE_NAME	Nazwa ciasteczka, które będzie skojarzone z naszą sesją Redisa.
SESSION_SECRET	Każda sesja ma unikalną daną poufną, która zapewnia możliwość dostępu do tej sesji w aplikacji Expressa. Powinieneś tu podać dowolną unikalną wartość.
SERVER_PORT	Numer portu, na którym będzie działał nasz serwer. Możesz użyć dowolnego portu, przy czym pamiętaj, że nie może on być używany przez żaden inny serwer.

Jeśli korzystasz z komputera Mac, to wykonaj polecenie:

```
export NODE_ENV=development
```

Tego polecenia użyj, jeśli pracujesz w systemie Windows:

```
SET NODE_ENV=development
```

To samo polecenie możesz także wykonać na swoim serwerze produkcyjnym, przy czym musisz zmienić wartość na `production`.

```
require("dotenv").config();
```

W tej instrukcji importujemy pakiet `dotenv` i przygotowujemy domyślną konfigurację. To właśnie ta instrukcja pozwala na zastosowanie pliku `.env` w projekcie.

```
const app = express();
```

W tym wierszu tworzymy instancję obiektu `express`. Do tego obiektu dodamy następnie całe używane oprogramowanie pośrednie. A ponieważ w Expressie niemal wszystko jest oprogramowaniem pośrednim, jest nim także obsługa stanu sesji.

```
app.listen({ port: process.env.SERVER_PORT }, () => {
  console.log(`Serwer działa na porcie ${process.env.SERVER_PORT}`);
});
```

W tym wierszu inicjujemy serwer, który po uruchomieniu wyświetli stosowny komunikat. Teraz możesz uruchomić serwer, używając następującego polecenia:

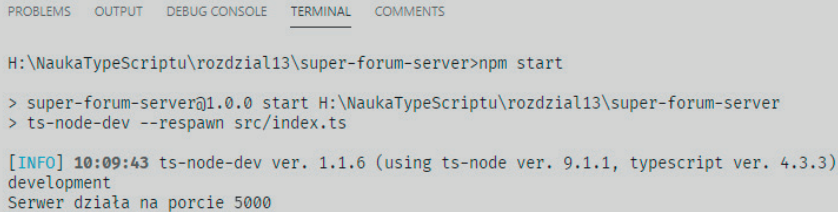
```
npm start
```

W panelu terminala powinny zostać wyświetlone komunikaty podobne do tych, które pokazałem na rysunku 13.7.

- Skoro już wiemy, że serwer w wersji podstawowej działa prawidłowo, możemy do niego dodać kod związany z obsługą stanu sesji Expressa oraz przechowywaniem danych w Redisie:



```
import express from "express";
import session from "express-session";
import connectRedis from "connect-redis";
import Redis from "ioredis";
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS
H:\NaukaTypeScriptu\rozdzial13\super-forum-server>npm start
> super-forum-server@1.0.0 start H:\NaukaTypeScriptu\rozdzial13\super-forum-server
> ts-node-dev --respawn src/index.ts
[INFO] 10:09:43 ts-node-dev ver. 1.1.6 (using ts-node ver. 9.1.1, typescript ver. 4.3.3)
development
Serwer działa na porcie 5000
```

### Rysunek 13.7. Pierwsze uruchomienie serwera Express

W pierwszej kolejności zwróć uwagę na to, że dodatkowo importujemy pakiet `express-session` oraz dwa pakiety związane z Redisem.

```
console.log(process.env.NODE_ENV);
require("dotenv").config();
```

```
const app = express();
const router = express.Router();
```

W tym fragmencie kodu inicjujemy obiekt `router`.

```
const redis = new Redis({
  port: Number(process.env.REDIS_PORT),
  host: process.env.REDIS_HOST,
  password: process.env.REDIS_PASSWORD,
});
```

Obiekt `redis` jest klientem serwera Redisa. Jak widać, wartości konfiguracyjne związane z Redisem ukryliśmy w pliku konfiguracyjnym `.env`. Na pewno wyobrażasz sobie, jak niebezpieczne byłoby, gdyby można było zobaczyć te wszystkie hasła i inne wrażliwe informacje podane na stałe w kodzie aplikacji.

```
const RedisStore = connectRedis(session);
const redisStore = new RedisStore({
  client: redis,
});
```

W tym fragmencie kodu tworzymy klasę `RedisStore`, a następnie obiekt `redisStore`, który będzie pełnił rolę magazynu danych dla sesji używanych w naszej aplikacji.

```
declare module "express-session" {
  interface Session {
    userid: any;
    loadedCount: Number;
  }
}
app.use(
```



```

    session({
      store: redisStore,
      name: process.env.COOKIE_NAME,
      sameSite: "Strict",
      secret: process.env.SESSION_SECRET,
      resave: false,
      saveUninitialized: false,
      cookie: {
        path: "/",
        httpOnly: true,
        secure: false,
        maxAge: 1000 * 60 * 60 * 24,
      },
    } as any)
  );

```

Do obiektu sesji trzeba przekazać pewne ustawienia konfiguracyjne. W jednym z nich, a konkretnie w polu o nazwie `store`, przekazujemy obiekt `redisStore`. Kolejne pole, `sameSite`, informuje, że ciasteczka pochodzące z innych witryn nie są dopuszczalne, co poprawia bezpieczeństwo aplikacji. Pole `secret` określa rodzaj hasła lub unikalnego identyfikatora naszej sesji. Pole `cookie` określa parametry ciasteczka, które będzie przechowywane w przeglądarce. Pole `httpOnly` oznacza, że ciasteczko to nie będzie dostępne dla kodu JavaScript. Takie rozwiązanie znacząco poprawia bezpieczeństwo ciasteczek i może uniemożliwić przeprowadzanie ataków typu XSS. Pole `secure` ma wartość `false`, gdyż nie używamy protokołu HTTPS.

```

app.use(router);
router.get("/", (req, res, next) => {
  if (!req.session!.userid) {
    req.session!.userid = req.query.userid;
    console.log("Określono userid!");
    req.session!.loadedCount = 0;
  } else {
    req.session!.loadedCount = Number(req.session!.loadedCount) + 1;
  }
}

```

W tym fragmencie kodu stosujemy przygotowany wcześniej obiekt `router` i definiujemy jedyną trasę, która będzie obsługiwać żądania typu GET. Najprościej rzecz ujmując, obsługa tych żądań polega na pobraniu parametru `userid` z łańcucha zapytania umieszczonego w adresie URL, a następnie na zapisaniu go w unikalnym polu sesji, `session.userid`. Oprócz tego zliczamy, ile razy zostało odebrane żądanie, aby pokazać, że sesja jest utrzymywana pomiędzy kolejnymi odwołaniami.

```

    res.send(
      `userid: ${req.session!.userid}, loadedCount:
      ${req.session!.loadedCount}`
    );

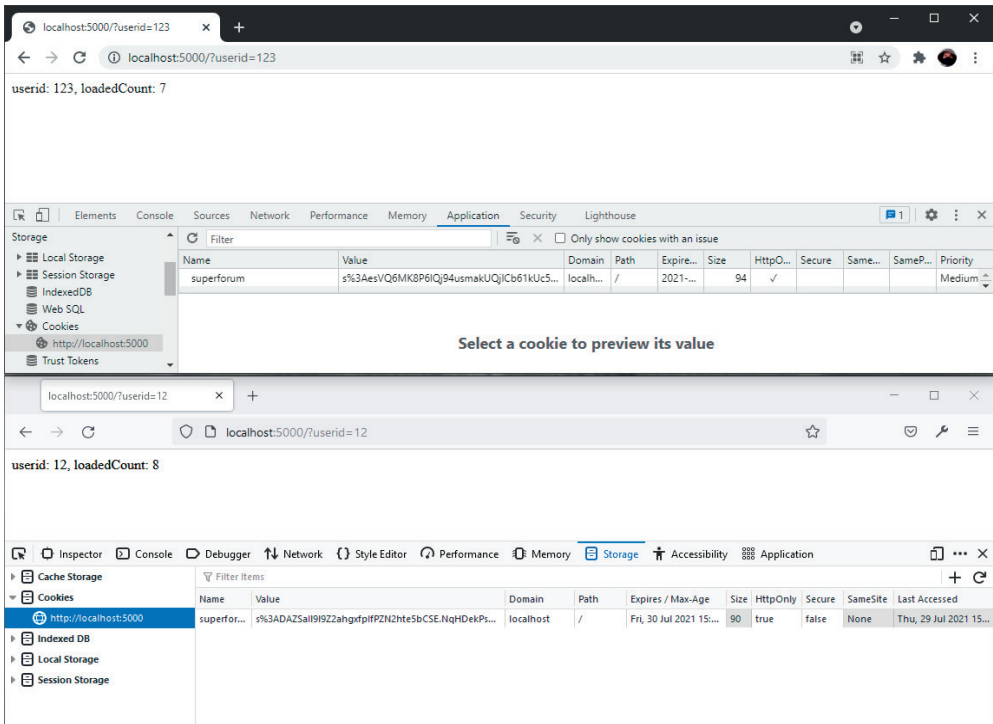
```

W tym fragmencie kodu generujemy odpowiedź, której zawartością będą informacje o sesji.

```
});

app.listen({ port: process.env.SERVER_PORT }, () => {
  console.log(`Serwer działa na porcie ${process.env.SERVER_PORT}`);
});
```

I w końcu nakazujemy serwerowi express nasłuchiwać na żądania na porcie 5000, bo właśnie taka wartość jest przypisana opcji konfiguracyjnej `SERVER_PORT`. Jak pokazałem na rysunku 13.8, ciasteczko zostaje utworzone po pierwszym odwołaniu do strony.



**Rysunek 13.8.** Dwie przeglądarki przedstawiające dwa unikalne stany sesji

Zwróć uwagę na to, że użyłem dwóch przeglądarek, by pokazać, że zostały utworzone dwie unikalne sesje. Gdybym użył tylko jednej przeglądarki, sesja nie byłaby unikalna, gdyż byłby używany ten sam obiekt sesji.

W tym podrozdziale wykorzystaliśmy znajomość Expressa i Redisa, by zaimplementować bazowy projekt aplikacji SuperForum. Dowiedziałeś się z niego, jakie role w całym rozwiązaniu będą pełnić Express i Redis. Oprócz tego dowiedziałeś się, jak użyć sesji do stworzenia unikalnego kontenera na dane każdego z użytkowników odwiedzających aplikację.

## Podsumowanie

Z tego rozdziału dowiedziałeś się, czym są sesje oraz czym jest magazyn danych Redis i jak go używać. Zobaczyłeś także, jak można zintegrować Expressa i Redisa, by tworzyć sesje dla użytkowników. Informacje te będą miały kluczowe znaczenie dla implementacji mechanizmów uwierzytelniania naszej aplikacji, którymi zajmiemy się w dalszych rozdziałach książki.

W następnym rozdziale uruchomimy serwer Postgres i przygotujemy schemat bazy danych. Przedstawię w nim także TypeORM — pakiet, który pozwoli nam zintegrować i używać Postgresa w naszej aplikacji. Oprócz tego zbudujemy w nim usługę uwierzytelniania użytkowników i powiążemy ją ze stanem sesji.

# Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM

W tym rozdziale zajmiemy się przygotowywaniem warstwy repozytorium, która będzie używać Postgresa jako bazy danych oraz TypeORM jako biblioteki zapewniającej dostęp do bazy. Przygotujemy schemat bazy danych i korzystając z możliwości TypeORM zapewnimy sobie możliwość wykonywania podstawowych operacji na tej bazie (określanych powszechnie jako „operacje CRUD”, od angielskich słów: *create*, *read*, *update* i *delete*, oznaczających odpowiednio: utworzenie, odczyt, aktualizację i usuwanie). Informacje zamieszczone w tym rozdziale mają kluczowe znaczenie, gdyż właśnie te operacje na danych będą najważniejszymi działaniami wykonywanymi przez serwerową część naszej aplikacji.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- zainstalowaniem i uruchomieniem bazy danych Postgres;
- przedstawieniem mechanizmów odwzorowań obiektowo-relacyjnych na przykładzie TypeORM;
- przygotowaniem warstwy repozytorium bazującej na Postgresie i TypeORM.

## Wymagania techniczne

W tej książce nie będę uczył Cię o sposobach korzystania z relacyjnych baz danych. Dlatego zakładam, że dysponujesz przynajmniej podstawową znajomością języka SQL, w tym umiejętnością tworzenia prostych zapytań i określania struktury tabel, jak również znajomością zagadnień związanych z tworzeniem aplikacji internetowych w środowisku Node. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code (VSCode)**.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, anglojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial14 (Chap14)*.

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog, o nazwie *rozdzial14*.

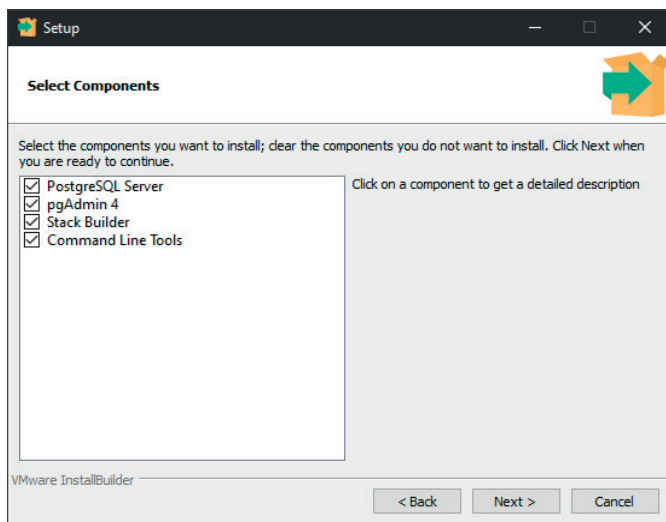
## Przygotowanie bazy danych Postgres

W tym podrozdziale zainstalujemy i uruchomimy bazę danych Postgres. Relacyjne bazy danych wciąż są powszechnie stosowanym rozwiązaniem, choć obecnie dużą popularność zyskują także bazy danych NoSQL. Niemniej jednak, według informacji dostępnych w witrynie StackOverflow, Postgres wciąż jest jedną z najpopularniejszych na świecie baz danych. Co więcej, Postgres działa rewelacyjnie i jest znacząco szybszy od MongoDB (<https://www.enterprisedb.com/news/new-benchmarks-show-postgres-dominating-mongodb-varied-workloads>). Właśnie dlatego zdecydowałem się na zastosowanie Postgresa jako serwera bazy danych w naszej aplikacji.

Zacznijmy zatem od zainstalowania Postgresa. Skorzystamy z instalatora udostępnianego przez EDB — firmę udostępniającą narzędzia i usługi wspierające bazę danych Postgres:

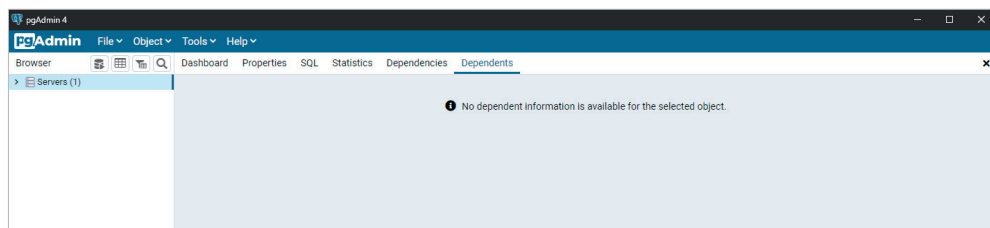
1. W przeglądarce WWW wyświetl stronę <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads> i pobierz wersję odpowiednią dla swojego systemu operacyjnego. W tym rozdziale będę używał wersji 13.3 dla systemu Windows — najnowszej dostępnej w czasie przygotowywania polskiego wydania tej książki.

2. Zaakceptuj ustawienia domyślne programu instalacyjnego, w tym także listę instalowanych komponentów (patrz rysunek 14.1).



Rysunek 14.1. Okno dialogowe programu instalacyjnego Postgresa

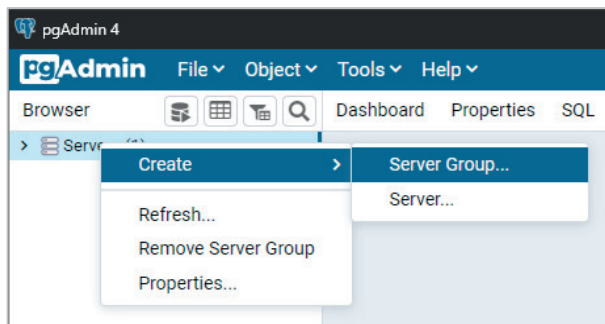
3. Po zakończeniu instalacji uruchom aplikację pgAdmin. To aplikacja służąca do zarządzania serwerem Postgres. Po jej uruchomieniu w przeglądarce zostanie wyświetlona strona przedstawiona na rysunku 14.2.



Rysunek 14.2. Aplikacja pgAdmin bezpośrednio po uruchomieniu serwera Postgres

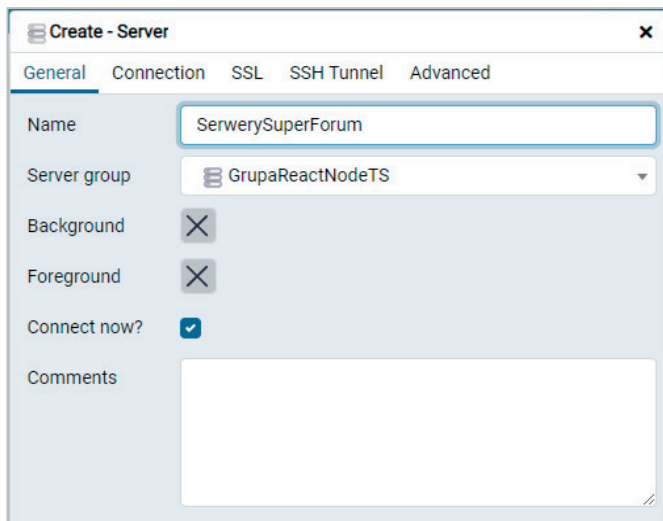
*pgAdmin* jest aplikacją internetową, choć wyglądem bardzo przypomina klasyczne aplikacje. Bezpośrednio po instalacji w aplikacji będzie dostępny tylko jeden serwer — lokalny serwer, który zainstalowałeś.

4. A teraz utwórz nową grupę serwerów o nazwie *GrupaReactNodeTS*, abyś mógł oddzielić ten projekt od innych. Grupa serwerów to jedynie rodzaj pojemnika pozwalającego na grupowanie wielu instancji serwerów, a na każdym serwerze można utworzyć i przechowywać wiele baz danych. Zwróć uwagę na to, że serwer **nie oznacza** wcale jednego fizycznego komputera.
5. Najpierw kliknij prawym przyciskiem myszy opcję *Servers*, a następnie z menu podręcznego wybierz opcję *Server Group*, jak pokazałem na rysunku 14.3.



Rysunek 14.3. Dodawanie grupy serwerów w aplikacji pgAdmin

6. Następnie utwórz serwer. W tym celu kliknij prawym przyciskiem myszy na grupie *GrupaReactNodeTS* i z menu podręcznego wybierz opcję *Create/Server*. W polu *Name* wpisz *SerwerySuperForum*, jak pokazałem na rysunku 14.4.



Rysunek 14.4. Okno dialogowe do tworzenia nowego serwera

7. Teraz przejdź na kartę *Connection* i w polu *Host name/address* wpisz *localhost*. Następnie wpisz hasło dla użytkownika postgres. Konto postgres jest kontem administratora, a hasło do niego będziesz musiał zapamiętać. Na rysunku 14.5 przedstawiłem postać karty *Connection* okna dialogowego *Create - Server*.
8. Kliknij przycisk *Save*, a serwer zostanie utworzony. Teraz panel z lewej strony okna aplikacji będzie wyglądał tak, jak pokazałem na rysunku 14.6.

Zwróć uwagę na to, że na nowym serwerze jest już dostępna baza danych **postgres**. Na razie jest ona pusta, lecz można jej używać do przechowywania danych globalnych.

**Create - Server** [X]

General Connection SSL SSH Tunnel Advanced

Host name/address: localhost

Port: 5432

Maintenance database: postgres

Username: postgres

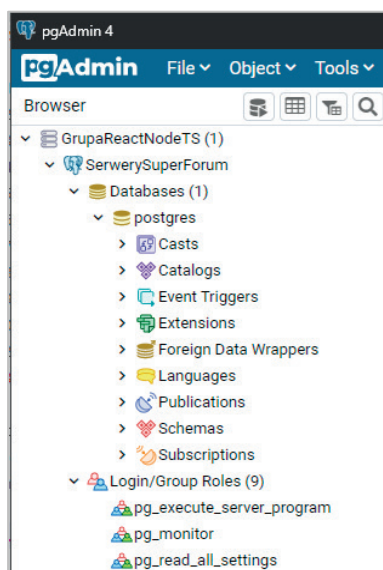
Password: .....

Save password? ☒

Role:

Service:

Rysunek 14.5. Karta Connection

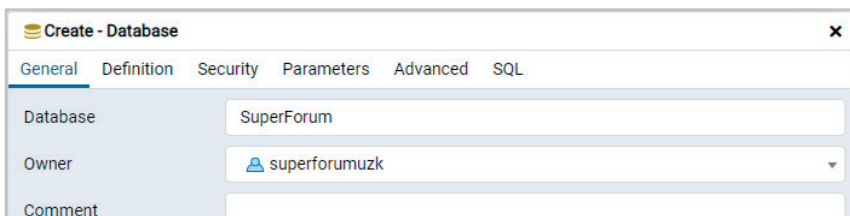


Rysunek 14.6. Nowa grupa serwerów GrupaReactNodeTS i nowego serwera SerwerySuperForum

Kolejnym zadaniem będzie utworzenie bazy danych dla naszej aplikacji. Jednak zanim się tym zajmiemy, musimy jeszcze utworzyć nowe konto użytkownika, którego będziemy używali do obsługi tej bazy danych. Stosowanie domyślnego konta administracyjnego, postgres, nie jest dobrym rozwiązaniem; gdyby komuś udało się włamać na to konto, zyskałby dostęp do całego serwera. A zatem:

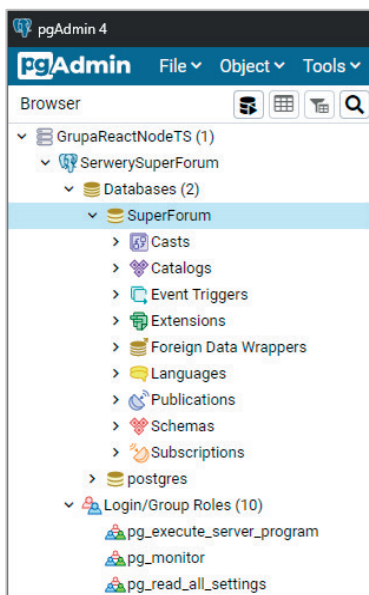


1. W lewym panelu aplikacji *pgAdmin* kliknij prawym przyciskiem myszy opcję *Login/Group Roles* i z menu podręcznego wybierz opcję *Create*, a następnie *Login/Group Role*. Na karcie *General*, w polu *Name* wpisz *superforumuzk*. Następnie przejdź na kartę *Definition* i wpisz hasło w polu *Password*. Przejdź na kartę *Privileges* i upewnij się, że tworzony użytkownik będzie mógł się zalogować (przełącz *Can login?* na *Yes*). W pozostałych ustawieniach możesz pozostawić wartości domyślne.
2. W lewym panelu okna aplikacji *pgAdmin* kliknij prawym przyciskiem myszy opcję *Databases* i z wyświetlonego menu podręcznego wybierz opcję *Create/Database*. Następnie w polu *Database* na karcie *General* wpisz *SuperForum*, a z listy *Owner* wybierz użytkownika *superforumuzk* (patrz rysunek 14.7).



Rysunek 14.7. Tworzenie bazy danych SuperForum

3. Kliknij przycisk *Save*. W efekcie, w panelu z lewej strony powinna pojawić się nowa baza danych (patrz rysunek 14.8).



Rysunek 14.8. Nowa baza danych i nowy użytkownik

Fantastycznie! Utworzyliśmy zatem nową bazę danych. Gdybyśmy nie używali mechanizmu odwzorowań obiektowo-relacyjnych (w skrócie ORM), musielibyśmy teraz wykonać mozolny proces ręcznego tworzenia wszystkich tabel i pól bazy danych. Jednak, jak się już niebawem przekonasz, TypeORM pozwala nam uniknąć tej przykrej konieczności, jak również udostępnia doskonale mechanizmy językowe do przeszukiwania zawartości bazy danych.

W następnym podrozdziale przyjrzymy się dokładniej TypeORM. Dowiesz się, jak TypeORM działa i w jaki sposób na wielu poziomach ułatwia nam prowadzenie interakcji z bazami danych.

## Przedstawienie mechanizmów odwzorowań obiektowo-relacyjnych na przykładzie TypeORM

Z tego podrozdziału dowiesz się, czym są tak zwane mechanizmy odwzorowań obiektowo-relacyjnych (ang. *Object Relational Mappers*, w skrócie **ORM**). W naszym projekcie będziemy używali TypeORM — jednego z najpopularniejszych frameworków ORM dla języka JavaScript. Mechanizmy ORM mogą w ogromnym stopniu ułatwiać korzystanie z baz danych i zmniejszać ilość informacji związanych ze stosowaniem i obsługą baz danych, które programista będzie musiał opanować.

Jako programista doskonale zdajesz sobie sprawę z tego, że różne języki programowania używają różnych, wzajemnie niezgodnych ze sobą typów. Na przykład język JavaScript, niezależnie od swojej nazwy, nie może bezpośrednio używać typów języka Java, ani nawet odwoływać się do nich. Aby móc używać typów jednego języka programowania w innym, konieczne jest wykonanie odpowiedniego „tłumaczenia”. Po części, właśnie to jest powodem istnienia usług takich jak internetowe API. Takie internetowe API udostępniają klientom dane zapisane w jakimś formacie tekstowym, na przykład: w formacie JSON. Dzięki temu z danych mogą korzystać dowolne klienty, gdyż dane te będzie można odczytać używając dowolnego języka programowania.

Podobne problemy z niezgodnościami typów występują w przypadku interakcji języków programowania z bazami danych. Dlatego też, zazwyczaj, po wykonaniu zapytania pobierającego dane z bazy, musielibyśmy napisać kod, który pobiera wartości poszczególnych zwróconych pól i konwertuje je do odpowiedniego typu. Jednak w przypadku wykorzystania mechanizmów ORM unikamy konieczności wykonywania większości takich operacji.

Mechanizmy ORM są projektowane w taki sposób, że *wiedzą*, jak odwzorowywać pola bazy danych na pola w kodzie i samodzielnie wykonują takie tłumaczenia. Oprócz tego mechanizmy tego typu dysponują pewnymi możliwościami pozwalającymi na automatyczne tworzenie tabel i pól baz danych na podstawie struktury encji utworzonych w kodzie. Encje możesz sobie wyobrazić jako typy zapisane w kodzie w konkretnym języku programowania i reprezentujące analogiczne obiekty przechowywane w bazie danych. Na przykład, gdybyśmy w kodzie JavaScript używali encji o nazwie `User`, powinniśmy dysponować w bazie danych

odpowiadającą jej tabelą o nazwie Users (zastosowanie liczby mnogiej jest uzasadnione, gdyż w tabeli zazwyczaj przechowywane są informacje o więcej niż jednym użytkowniku).

Już tylko te możliwości, o których wspomniałem powyżej, mogą nam zaoszczędzić bardzo wiele czasu i pracy, jednak dobre mechanizmy ORM oprócz nich udostępniają także wiele innych możliwości, takich jak: tworzenie zapytań, bezpieczne wstawianie parametrów (co pozwala zapobiegać atakom polegającym na wstrzykiwaniu kodu SQL), czy też obsługa transakcji. Transakcje są atomowymi operacjami wykonywanymi przez bazy danych, które bądź to zostaną w całości wykonane poprawnie, bądź też, w razie niepowodzenia, zostaną w całości wycofane.

Wstrzykiwanie kodu SQL to rodzaj ataku, w którym napastnik stara się wstawić i wykonać kod SQL, odbiegający od kodu przewidzianego przez twórcę aplikacji. Efektami takich ataków może być utrata danych lub awarie aplikacji.

Jak już wspominałem, w naszej aplikacji będziemy używali mechanizmu ORM o nazwie TypeORM. Jest to bardzo popularny i ceniony mechanizm ORM dla języka TypeScript, mający ponad 20 tysięcy polubień w serwisie GitHub. Udostępnia on wszystkie możliwości, o których wcześniej wspominałem, a rozpoczęcie korzystanie z niego nie nastęrcza zbyt wielu problemów, choć z drugiej strony opanowanie go na poziomie zaawansowanym wymaga sporego wysiłku. TypeORM obsługuje wiele różnych baz danych, w tym: Microsoft SQL, MySQL oraz Oracle.

Bogate możliwości TypeORM pozwolą nam zaoszczędzić masę pracy, a dzięki temu, że jest on stosowany w wielu projektach pisanych w JavaScriptcie, dysponuje dużą społecznością użytkowników, którzy będą mogli Ci pomóc, gdybyś napotkał jakieś problemy.

W tym podrozdziale zamieściłem wprowadzenie do mechanizmów ORM. Dowiedziałeś się z niego, czym one są oraz dlaczego są dla nas ważne i cenne. W następnym podrozdziale zaczniemy używać TypeORM w dalszych pracach nad rozwojem naszego projektu. A zatem, zaczynajmy!

## Tworzenie warstwy repozytorium bazującej na Postgresie i TypeORM

W tym podrozdziale wyjaśnię znaczenie stosowania warstwy repozytorium. Zastosowanie odrębnej warstwy do implementacji dużej i ważnej części aplikacji może znacząco ułatwić późniejsze utrzymanie i refaktoryzację kodu. Oprócz tego, zastosowanie takiej struktury kodu ułatwia zrozumienie działania aplikacji, gdyż jej główne sekcje będą od siebie logicznie oddzielone.

W rozdziale 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript”, poznałeś podstawowe zagadnienia związane z programowaniem obiektowym. Jednym z podstawowych mechanizmów stosowanych w programowaniu obiektowym jest abstrakcja. Jeśli w naszej aplikacji kod związany z dostępem do bazy danych umieścimy w odrębnej warstwie,

będzie to właśnie przykład stosowania abstrakcji. Jak zapewne pamiętasz, jedną z zalet abstrakcji jest ukrywanie tajników implementacji i udostępnianie zewnętrznemu kodowi jedynie interfejsów. Co więcej, dzięki temu, że kod związany z obsługą bazy danych będzie umieszczony w jednym miejscu, nie będziemy musieli szukać po całej aplikacji kodu wykonującego zapytania. Będziemy doskonale wiedzieć, w jakiej warstwie aplikacji jest on umieszczony. Zachowanie logicznej separacji kodu jest nazywane „separacją odpowiedzialności” (ang. *separation on concerns*).

Zabierzmy się zatem za tworzenie warstwy repozytorium:

1. Pierwszym krokiem będzie skopiowanie kodu serwera przygotowanego w rozdziale 13., pt. „Przygotowywanie stanu sesji przy użyciu Expressa i Redisa”. Przejdź do katalogu *rozdzial13* w kodach źródłowych dołączonych do książki i skopiuj katalog *super-forum-server* do katalogu *rozdzial14*.

Po skopiowaniu katalogu *super-forum-server* do katalogu *rozdzial14*, będziesz musiał usunąć z niego plik *package-lock.json* i ponownie zainstalować wszystkie niezbędne pakiety NPM.

```
npm install
```

2. Teraz musimy zainstalować TypeORM i wszystkie jego zależności. W tym celu wykonaj następujące polecenia:

```
npm i typeorm pg bcryptjs cors class-validator
npm i @types/pg @types/cors @types/bcryptjs -D
```

Wykonując te dwa polecenia zainstalowałeś typeorm. Z kolei pakiet pg jest klientem służącym do prowadzenia komunikacji z bazą danych Postgres. bcryptjs jest biblioteką, której będziemy używali do szyfrowania haseł przed ich zapisaniem w bazie danych. Kolejna biblioteka, cors, zapewnia możliwość odbierania żądań pochodzących z różnych domen, innych niż domena, w której działa aplikacja. W nowoczesnych aplikacjach może się zdarzyć, że kod klienta nie będzie udostępniany przez ten sam serwer, na którym działa serwerowa część aplikacji. Dotyczy to w szczególności tych przypadków, kiedy tworzymy jakiś API, jak na przykład API korzystający z GraphQL-a, taki, który może być używany przez wiele różnych klientów. Podobna sytuacja wystąpi, kiedy zaczniemy integrować naszego klienta Reacta z częścią serwerową aplikacji, która będzie działać na innym porcie.

Pakiet class-validator pozwala na stosowanie dekoratorów służących do przeprowadzania walidacji. Przedstawię go dokładniej w dalszej części rozdziału na odpowiednich przykładach.

3. Zanim zaczniemy tworzyć bazę danych encji, musimy jeszcze przygotować plik konfiguracyjny, dzięki któremu TypeORM będzie wiedział, jak nawiązać połączenie z bazą danych Postgres. To oznacza, że musimy także zaktualizować plik *.env* i podać w nim informacje dotyczące konfiguracji bazy danych. Otwórz plik *.env* i dodaj do niego poniższe zmienne. Nasz serwer Postgres działa na lokalnym komputerze, więc zmienna PG\_HOST będzie mieć wartość localhost:

```
PG_HOST=localhost
```

Kolejna zmienna określa numer portu, na którym działa serwer:

```
PG_PORT=5432
```

Poniższa zmienna podaje nazwę konta użytkownika bazy:

```
PG_ACCOUNT=superforumuzk
```

Musimy także podać hasło tego użytkownika:

```
PG_PASSWORD=<hasło_użytkownika>
```

W osobnej zmiennej podajemy nazwę bazy danych:

```
PG_DATABASE=SuperForum
```

Jak już wcześniej wspominałem, TypeORM utworzy tabele bazy danych i ich pola za nas i będzie je aktualizował w razie wprowadzenia jakichkolwiek zmian. To działanie zapewnia ustawienie zmiennej konfiguracyjnej PG\_SYNCHRONIZE:

```
PG_SYNCHRONIZE=true
```

Oczywiście, po uruchomieniu wersji produkcyjnej aplikacji tę opcję trzeba będzie wyłączyć, aby zapobiec niepożądanym zmianom w bazie danych.

Kolejna opcja określa położenie plików encji, w tym katalogu, w którym są one umieszczone:

```
PG_ENTITIES="src/repo/**/*.*"
```

Osobna opcja określa główny katalog plików encji:

```
PG_ENTITIES_DIR="src/repo"
```

Opcja PG\_LOGGING określa, czy na serwerze należy włączyć rejestrowanie komunikatów w dziennikach:

```
PG_LOGGING=false
```

Rejestrowanie informacji w dziennikach pozwala na wykrywanie ewentualnych problemów. Jednak korzystanie z dzienników może powodować powstawanie ogromnych plików, więc nie będziemy korzystać z tej opcji w środowisku roboczym.

4. Teraz możemy zająć się przygotowaniem pliku konfiguracyjnego TypeORM. W głównym katalogu projektu, *rozdzial14/super-forum-server*, utwórz plik *ormconfig.json* i zapisz w nim następujący kod:

```
require("dotenv").config();
```

W pierwszej kolejności pobieramy ustawienia z pliku *.env*, używając do tego celu instrukcji `require`.

```
module.exports = [
  {
    type: "postgres",
```

Z jakiego typu bazy danych będziemy korzystać? TypeORM obsługuje ich kilka, więc w ustawieniach konfiguracyjnych musimy ją określić.

Pozostałe ustawienia konfiguracyjne korzystają z wartości konfiguracyjnych podanych w pliku *.env*, więc kolejny fragment kodu nie wymaga wyjaśnień:

```

    host: process.env.PG_HOST,
    port: process.env.PG_PORT,
    username: process.env.PG_ACCOUNT,
    password: process.env.PG_PASSWORD,
    database: process.env.PG_DATABASE,
    synchronize: process.env.PG_SYNCHRONIZE,
    logging: process.env.PG_LOGGING,
    entities: [process.env.PG_ENTITIES],
    cli: {
      entitiesDir: process.env.PG_ENTITIES_DIR,
    },
  },
];

```

Teraz jesteśmy już gotowi do tworzenia klas encji.

5. Skoro już zainstalowaliśmy wszystkie niezbędne pakiety i skonfigurowaliśmy połączenie z bazą danych możemy przystąpić do tworzenia pierwszej encji: *User*. Zacznij od utworzenia w katalogu *rozdzial14/super-forum-server/src* nowego katalogu, o nazwie *repo*. Umieścimy w nim cały kod związany z repozytorium. Wewnątrz katalogu *repo* utwórz plik *User.ts* i zapisz w nim następujący wiersz kodu:

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";
```

Te elementy importowane z pakietu *typeorm* pozwolą nam utworzyć klasę encji *User*. *Entity*, *PrimaryGeneratedColumn* oraz *Column* to tak zwane *dekoratory*.

Dekoratory są atrybutami umieszczanymi bezpośrednio przed wybranym wierszem kodu i dostarczającymi dodatkowych informacji na temat pola lub obiektu. Można je sobie wyobrażać jako swoisty skrót. Zamiast pisać długie wiersze kodu, można dodać krótki znacznik, który określi ustawienia konfiguracyjne. Przykłady zastosowania tych dekoratorów zobaczysz już niebawem w kodzie.

```
import { Length } from "class-validator";
```

Tu importujemy walidator długości.

Przejdźmy teraz do pierwszego zastosowania dekoratorów. Dekorator *Entity* informuje *TypeORM*, że klasa, która właśnie ma zostać zdefiniowana, jest encją (w naszym przypadku będzie to encja o nazwie *Users*). Innymi słowy, w naszym kodzie będziemy mieli obiekty *User*, które będą bezpośrednio odwzorowywane na tabelę *Users* w bazie danych:

```
@Entity({name: "Users" })
```

Wszystkie tabele baz danych muszą zawierać pole, które będzie w unikalny sposób identyfikować rekordy. Właśnie do tego służy dekorator *PrimaryGeneratedColumn*. W naszym przypadku pole to nazwiemy *id*. Zwróć uwagę na to, że nazwa *"id"* nie zaczyna się dużą literą; zajmiemy się tym później.

```

export class User {
  @PrimaryGeneratedColumn({ name: "Id", type: "bigint" })
  id: string;
}

```

W kolejnym fragmencie kodu po raz pierwszy zastosujemy dekorator Column:

```
@Column("varchar", {
    name: "Email",
    length: 120,
    unique: true,
    nullable: false,
})
email: string;
```

Jak widać, dekorator ten służy do zdefiniowania w bazie danych pola o nazwie Email, które w kodzie TypeScript będzie nosić nazwę email. Więc także w tym przypadku dekoratory zostały zastosowane do odwzorowania obiektów używanych w kodzie na encje bazy danych. Przyjrzyjmy się teraz dekoratorowi Column nieco dokładniej. Przede wszystkim definiuje on, że w bazie danych kolumna Email będzie typu varchar. Jeszcze raz powtórzę, że typy stosowane w bazach danych różnią się od typów stosowanych w kodzie, tak jest także w tym przypadku. Następnym elementem dekoratora jest pole name, które w przedstawionym przykładzie ma wartość Email. To będzie nazwa pola w bazie danych Users. Kolejne pole, length, określa maksymalną liczbę znaków, które można zapisać w tym polu. Atrybut unique informuje bazę danych, że powinna sprawdzać i wymuszać, by każdy użytkownik zapisany w bazie danych User miał unikalny adres poczty elektronicznej. I w końcu ostatnie pole dekoratora, nullable, ma wartość false, co oznacza, że w bazie danych to pole będzie musiało mieć jakąś wartość.

```
@Column("varchar", {
    name: "UserName",
    length: 60,
    unique: true,
    nullable: false,
})
userName: string;

@Column("varchar", { name: "Password", length: 100, nullable: false })
@Length(8, 100)
```

W tym fragmencie kodu używamy dekoratora Length, by zagwarantować, że wartość pola będzie mieć odpowiednią minimalną i maksymalną długość.

```
password: string;
```

Te dwa pola, userName oraz password, będą powiązane z kolumnami typu varchar, podobnie jak pole email.

```
@Column("boolean", { name: "Confirmed", default: false, nullable: false })
confirmed: boolean;
```

Ten fragment kodu definiuje pole confirmed typu boolean. To pole będzie zawierać informację o tym, czy adres e-mail podany w koncie nowego użytkownika został potwierdzony, czy nie. Definicja tego pola jest prosta i nie wymaga dodatkowego tłumaczenia, zauważ jednak, że domyślna wartość pola sprawia, że w momencie zapisywania rekordu w bazie danych w tym polu zostanie domyślnie zapisana wartość false (o ile jawnie nie podamy innej).

```

    @Column("boolean", { name: "IsDisabled", default: false, nullable: false })
    isDisabled: boolean;
  }

```

Ten fragment definiuje ostatnie pole, `isDisabled`, które pozwoli na wyłączenie konta użytkownika, gdy będzie tego wymagało zarządzanie aplikacją.

6. Doskonale! Teraz możemy się przekonać, czy TypeORM utworzy dla nas nową tabelę `Users`. Ostatnią rzeczą, jaką musimy jeszcze zrobić, jest nawiązanie połączenia z bazą danych z poziomu naszego kodu. W tym celu zmodyfikuj plik `index.ts` zgodnie z poniższym przykładem:

```

import express from "express";
import session from "express-session";
import connectRedis from "connect-redis";
import Redis from "ioredis";
import { createConnection } from "typeorm";
require("dotenv").config();

```

W tym fragmencie kodu importujemy z pakietu TypeORM funkcję `createConnection`.

```

const main = async () => {
  const app = express();
  const router = express.Router();

  await createConnection();

```

W tym fragmencie kodu wywołujemy funkcję `createConnection`. Koniecznie zwróć jednak uwagę na to, że teraz nasz kod jest umieszczony wewnątrz asynchronicznej funkcji `main` — zwróć uwagę na zastosowanie słowa kluczowego `async`. Takie rozwiązanie jest konieczne, gdyż `createConnection` także jest funkcją asynchroniczną i jako taka musi być wywoływana z użyciem słowa kluczowego `await`. Dlatego musieliśmy umieścić jej wywołanie wewnątrz funkcji asynchronicznej i taki jest powód dodania do kodu funkcji `main`.

Pozostała część kodu jest taka sama jak w poprzednim rozdziale, ale przenieśliśmy ją do funkcji `main`:

```

const redis = new Redis({
  port: Number(process.env.REDIS_PORT),
  host: process.env.REDIS_HOST,
  password: process.env.REDIS_PASSWORD,
});
const RedisStore = connectRedis(session);
const redisStore = new RedisStore({
  client: redis,
});

app.use(
  session({
    store: redisStore,
    name: process.env.COOKIE_NAME,
    sameSite: "Strict",
    secret: process.env.SESSION_SECRET,
    resave: false,
    saveUninitialized: false,

```



```

    cookie: {
      path: "/",
      httpOnly: true,
      secure: false,
      maxAge: 1000 * 60 * 60 * 24,
    },
  } as any)
);

```

Także dalsza część kodu jest taka sama:

```

app.use(router);
router.get("/", (req, res, next) => {
  if (!req.session!.userid) {
    req.session!.userid = req.query.userid;
    console.log("Określono userid!");
    req.session!.loadedCount = 0;
  } else {
    req.session!.loadedCount = Number(req.session!.loadedCount) + 1;
  }

  res.send(
    `userid: ${req.session!.userid}, loadedCount: ${req.session!.
      ↪loadedCount}`
  );
});

app.listen({ port: process.env.SERVER_PORT }, () => {
  console.log(`Serwer działa na porcie ${process.env.SERVER_PORT}`);
});

};

main();

```

W tym fragmencie kodu kończymy funkcję `main`, a następnie ją wywołujemy.

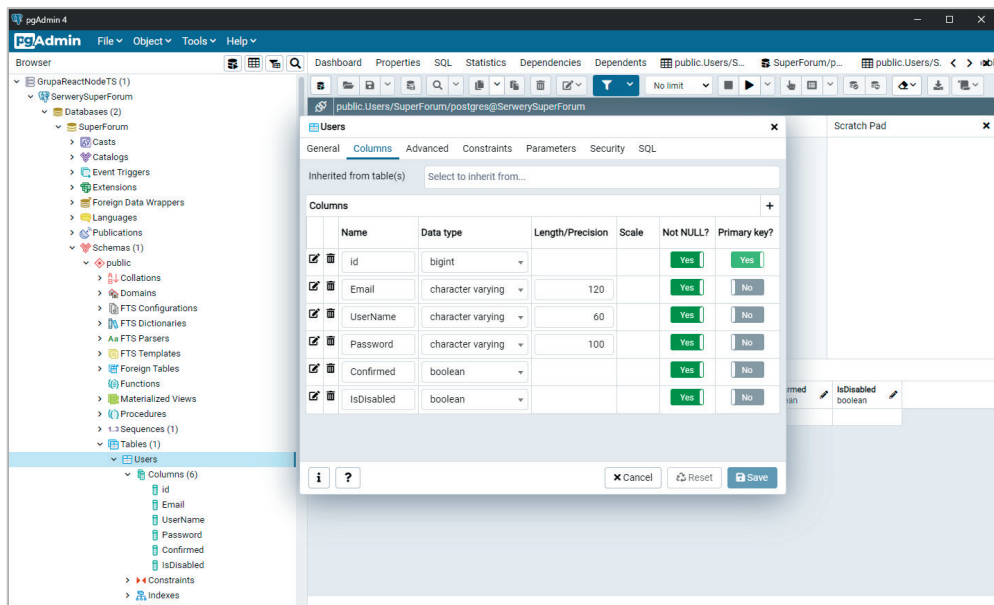
7. Teraz możesz już uruchomić aplikację, używając do tego następującego polecenia:

```
npm start
```

Komunikaty wyświetlane w panelu terminala nie zmieniły się. Jednak, kiedy otworzysz aplikację *pgAdmin* i wyświetlisz listę tabel (*Tables*), zobaczysz na niej nową tabelę *Users* wraz z utworzonymi w niej kolumnami (rysunek 14.9).

To naprawdę zaoszczędziło nam bardzo wiele czasu! Czy wyobrażasz sobie, ile czasu i wysiłku wymagałoby ręczne utworzenie każdej z tabel, których będziemy używać w aplikacji? Wraz z tymi wszystkimi polami!? To by potrwało długie godziny.

Zwróć uwagę na to, że utworzone kolumny mają te same ustawienia, które podaliśmy w dekoratorach. Na przykład, kolumna przeznaczona do przechowywania e-maili jest typu znakowego o zmiennej długości, przy czym maksymalna długość zapisywanych w niej łańcuchów może wynosić 120 znaków, a kolumna nie może zawierać wartości pustych.



Rysunek 14.9. Nowa tabela Users

8. Mamy jednak pewien mały problem. Nazwa naszej kolumny id zaczyna się od małej litery, choć we wszystkich pozostałych kolumnach nazwy są zapisywane począwszy od dużej litery. Spróbujmy to poprawić. Wystarczy, że otworzysz plik *User.ts* i zmienisz wartość parametru `name` w dekoratorze `PrimaryGeneratedColumn` z `id` na `Id` (pamiętaj, żeby zmienić tylko identyfikator podany w dekoratorze; umieszczone poniżej niego pole `id` w kodzie TypeScript ma pozostać bez zmian). Jeśli serwer nie działa, to ponownie go uruchom. Po uruchomieniu serwera odśwież tabelę wyświetloną w aplikacji *pgAdmin*: kliknij jej nazwę prawym przyciskiem myszy i z wyświetlonego menu podręcznego wybierz opcję *Refresh*. Powinieneś zobaczyć, że nazwa kolumny `id` została zmieniona na `Id`. To jest jedna z niesamowitych możliwości TypeORM, gdyż ręczne modyfikowanie kolumn oraz ich ograniczeń czasami może być bardzo uciążliwe.
9. Super! Teraz musimy zająć się pozostałymi encjami: `Thread` oraz `ThreadItem`. Przypomnę, że `Thread` reprezentuje początkowy wpis na forum, punkt startowy do dalszych dyskusji, natomiast `ThreadItem` reprezentuje odpowiedzi na ten wpis początkowy. Najpierw zatrzymaj serwer, by nie tworzył elementów bazy danych zanim wszystko przygotujemy. Poniżej zamieszczę definicje obu encji, a ponieważ są one bardzo podobne do definicji encji `User`, nie będę ich dokładnie opisywał.

W obu plikach potrzebne będą te same instrukcje importu:

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";
import { Length } from "class-validator";
```

Poniżej przedstawiłem początkową postać encji Thread (później, kiedy będziemy definiować zależności pomiędzy tabelami, dodamy do niej więcej pól):

```
@Entity({ name: "Threads" })
export class Thread {
  @PrimaryGeneratedColumn({ name: "Id", type: "bigint" })
  id: string;

  @Column("int", { name: "Views", default: 0, nullable: false })
  views: number;

  @Column("boolean", { name: "IsDisabled", default: false, nullable: false })
  isDisabled: boolean;

  @Column("varchar", { name: "Title", length: 150, nullable: false })
  @Length(5, 150)
  title: string;

  @Column("varchar", { name: "Body", length: 2500, nullable: true })
  @Length(10, 2500)
  body: string;
}
```

Poniżej przedstawiłem definicję encji ThreadItem:

```
@Entity({ name: "ThreadItems" })
export class ThreadItem {
  @PrimaryGeneratedColumn({ name: "Id", type: "bigint" })
  id: string;

  @Column("int", { name: "Views", default: 0, nullable: false })
  views: number;

  @Column("boolean", { name: "IsDisabled", default: false, nullable: false })
  isDisabled: boolean;

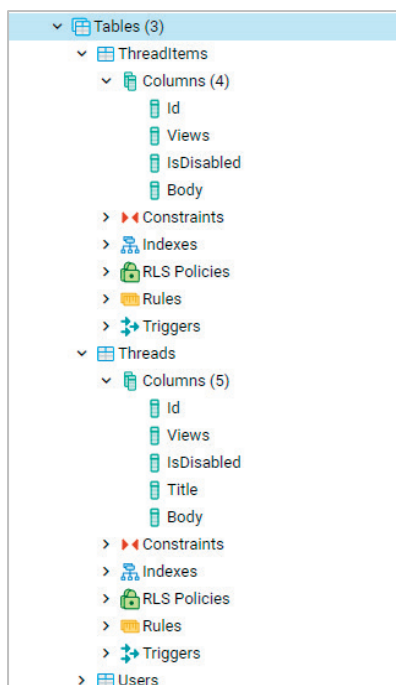
  @Column("varchar", { name: "Body", length: 2500, nullable: true })
  @Length(10, 2500)
  body: string;
}
```

10. Jak widać, obie encje są całkiem proste. Teraz możesz ponownie uruchomić serwer, a kiedy odświeżysz listę tabel wyświetlonych w aplikacji *pgAdmin*, przekonasz się, że w bazie pojawiły się dwie nowe tabele: *Threads* i *ThreadItems* (patrz rysunek 14.10).

Wciąż jest jeszcze sporo pól, które musimy dodać, takich jak pola do przechowywania informacji o punktach. Jednak zanim się nimi zajmiemy, dodamy wzajemne zależności pomiędzy poszczególnymi tabelami. Na przykład, zarówno wątki, jak i odpowiedzi powinny być powiązane z konkretnym użytkownikiem. Zaczniemy od dodania poniższej zależności:

1. Przede wszystkim zatrzymaj serwer. Następnie otwórz plik *User.ts* i na samym końcu klasy *User* dodaj poniższy fragment kodu:

```
@OneToMany(() => Thread, (thread) => thread.user)
threads: Thread[];
```



Rysunek 14.10. Tabele Threads i ThreadItems

Dekorator `OneToMany` informuje, że potencjalnie dla każdego użytkownika (obiektu `User`) może istnieć dowolnie wiele powiązanych z nim wątków (obiektów `Thread`).

2. Teraz otwórz plik *Thread.ts* i na końcu klasy `Thread` dodaj poniższy fragment kodu:

```
@ManyToOne(
  () => User,
  (user: User) => user.threads
)
user: User;
```

Dekorator `ManyToOne` pokazuje, że każdy z wątków (których może być dowolnie wiele) jest skojarzony z dokładnie jednym użytkownikiem. Choć uczenie SQL-a wykracza poza ramy tematyczne tej książki, to jednak wyjaśnię, że — najprościej rzecz ujmując — taka zależność działa w bazie danych jako swoiste ograniczenie, które zabezpiecza bazę przed wstawianiem do niej informacji, które nie mają sensu, takich jak: wielu użytkowników (rekordów tabeli `Users`) będących założycielami jednego wątku (rekordu tabeli `Threads`).

3. Teraz zajmiemy się zależnością pomiędzy tabelami `Thread` i `ThreadItems`. Dodaj poniższy fragment na końcu klasy `Thread`:

```
@OneToMany(
  () => ThreadItem,
  (threadItems) => threadItems.thread
)
threadItems: ThreadItem[];
```

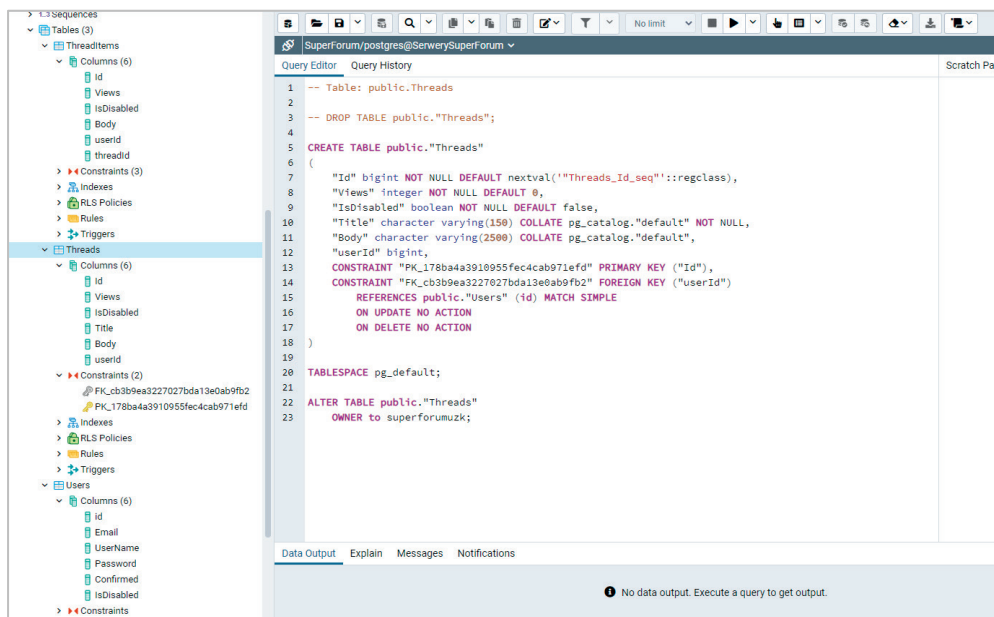
Ten dekorator informuje, że z jednym wątkiem (Thread) może być skojarzonych wiele obiektów ThreadItem. Odpowiedni kod trzeba także dodać do klasy ThreadItem:

```
@ManyToOne(() => User, (user) => user.threads)
user: User;
```

Z każdym obiektem ThreadItem, podobnie jak z obiektami Thread, powinien być skojarzony dokładnie jeden użytkownik (User).

```
@ManyToOne(() => Thread, (thread) => thread.threadItems)
thread: Thread;
```

4. Każda odpowiedź (ThreadItem) może mieć tylko jeden wątek (Thread), do którego należy. Kiedy ponownie uruchomisz serwer, powinieneś zobaczyć nowe zależności (patrz rysunek 14.11).



Rysunek 14.11. Zależności pomiędzy tabelami

Jak łatwo zauważyć, do tabel Threads i ThreadItems zostały dodane nowe kolumny. Na przykład, w tabeli ThreadItems są to kolumny userId oraz threadId, umożliwiające wskazanie rekordów z tabel Users i ThreadItems skojarzonych z danym rekordem ThreadItem. Jednak zwróć uwagę na to, że do tabeli Users nie została dodana żadna nowa kolumna. Wynika to z faktu, że do klasy User dodaliśmy jedynie zależność OneToMany z klasą Thread. W bazie danych zależność ta jest zdefiniowana przez ograniczenie widoczne w skrypcie CREATE TABLE public."Threads", przedstawionym na rysunku 14.11. Jak widać, w poleceniu SQL tworzącym tabelę Threads zostało umieszczone ograniczenie (CONSTRAINT) dotyczące kolumny userId. A zatem, zaznaczając, że każdy wątek jest skojarzony z dokładnie jednym użytkownikiem, niejawnie informujemy, że każdy użytkownik może być skojarzony z wieloma wątkami.

Kolejną rzeczą, którą się zajmiemy, będzie przygotowanie systemu punktacji. W przypadku punktów, czyli oznaczania wątków i odpowiedzi jako lubiane lub nie lubiane, musimy zapewnić użytkownikom możliwość oznaczenia danego wątku lub odpowiedzi tylko jeden raz. Będzie to wymagało utworzenia dwóch nowych tabel, `ThreadPoints` oraz `ThreadItemPoints`, które zdefiniują zależności między rekordami tabel `Users`, `Threads` i `ThreadItems`.

1. Zacznij od wyłączenia serwera, po czym utwórz nowy plik *ThreadPoint.ts* i zapisz w nim poniższy kod:

```
@Entity({ name: "ThreadPoints" })
export class ThreadPoint {
  @PrimaryGeneratedColumn({ name: "Id", type: "bigint" }) // Na potrzeby typeorm
  id: string;

  @Column("boolean", { name: "IsDecrement", default: false, nullable: false })
  isDecrement: boolean;

  @ManyToOne(() => User, (user) => user.threadPoints)
  user: User;

  @ManyToOne(() => Thread, (thread) => thread.threadPoints)
  thread: Thread;
}
```

Klasa zdefiniowana w tym kodzie informuje, że punkt, który reprezentuje, odnosi się do konkretnego użytkownika (`User`) i wątku (`Thread`). Kolumna `isDecrement` jest używana do oznaczania punktu jako polubienia lub jego przeciwności; jeśli zapiszemy w niej wartość `true`, będzie to oznaczać, że użytkownik danego wątku nie lubi. Punkty mogą mieć zatem trzy stany: punkt może nie istnieć, może oznaczać polubienie bądź też jego przeciwność. W dalszej części kodu obsługującego repozytorium napiszemy zapytania obsługujące te trzy stany.

2. Teraz dodaj poniższy fragment kodu na końcu klasy w pliku *User.ts*:

```
@OneToMany(() => ThreadPoint, (threadPoint) => threadPoint.user)
threadPoints: ThreadPoint[];
```

Stanowi on uzupełnienie zdefiniowanej wcześniej zależności.

3. Teraz dodaj kolejny fragment kodu na końcu klasy w pliku *Thread.ts*:

```
@OneToMany(() => ThreadPoint, (threadPoint) => threadPoint.thread)
threadPoints: ThreadPoint[];
```

Także on stanowi uzupełnienie zależności z klasą `ThreadPoint`.

4. A teraz musimy w podobny sposób dodać punktację dla odpowiedzi, którą będzie reprezentować klasa `ThreadItemPoint`. Utwórz nowy plik, *ThreadItemPoint.ts*, i zapisz w nim poniższy kod:

```
@Entity({ name: "ThreadItemPoints" })
export class ThreadItemPoint {
  @PrimaryGeneratedColumn({ name: "Id", type: "bigint" }) // Na potrzeby typeorm
  id: string;

  @Column("boolean", { name: "IsDecrement", default: false, nullable: false })
  isDecrement: boolean;
```

```

@ManyToOne(() => User, (user) => user.threadPoints)
user: User;

@ManyToOne(() => ThreadItem, (threadItem) => threadItem.threadItemPoints)
threadItem: ThreadItem;
}

```

Ta klasa jest bardzo podobna do przedstawionej wcześniej klasy `ThreadPoint`.

5. Teraz zmodyfikuj klasę `User`, a konkretnie dodaj do niej poniższy fragment kodu:

```

@OneToMany(() => ThreadItemPoint, (threadItemPoint) =>
threadItemPoint.user)
threadItemPoints: ThreadItemPoint[];

```

Poniższy fragment kodu dodaj na końcu klasy `ThreadItem`:

```

@OneToMany(
() => ThreadItemPoint,
(threadItemPoint) => threadItemPoint.threadItem
)
threadItemPoints: ThreadItemPoint[];

```

I to już ostatni element, który kończy definiowanie zależności związanych z klasą `ThreadItemPoint`.

Ale to wciąż jeszcze nie wszystko. Zapewne pamiętasz z rozdziału 11., pt. „Czego się nauczysz — aplikacja internetowego forum”, że nasze wątki będą należeć do kategorii. Musimy zatem utworzyć odpowiednią encję oraz jej zależności:

1. Zacznij od utworzenia pliku *ThreadCategory.ts* i zapisania w nim poniższego kodu:

```

@Entity({ name: "ThreadCategories" })
export class ThreadCategory {
  @PrimaryGeneratedColumn({ name: "Id", type: "bigint" }) // Na potrzeby typeorm
  id: string;

  @Column("varchar", {
    name: "Name",
    length: 100,
    unique: true,
    nullable: false,
  })
  name: string;

  @Column("varchar", {
    name: "Description",
    length: 150,
    nullable: true,
  })
  description: string;

  @OneToMany(() => Thread, (thread) => thread.category)
  threads: Thread[];
}

```

Klasa `ThreadCategory` jest bardzo podobna do pozostałych klas encji.

2. Poniższy fragment dodaj do pliku *Thread.ts*:

```
@ManyToOne(() => ThreadCategory, (threadCategory) => threadCategory.
↳ threads)
category: ThreadCategory;
```

Oczywiście, ten fragment kodu tworzy zależność pomiędzy klasami *Thread* i *ThreadCategory*.

3. Teraz uruchom serwer; w efekcie powinny zostać utworzone nowe tabele oraz zależności.

Przygotowaliśmy już niezbędne klasy encji, a także zdefiniowaliśmy ich wzajemne zależności. Jednak chcemy, by każdej operacji zmiany zawartości bazy danych towarzyszyło zapisanie w dzienniku informacji o tym, czy do bazy zostało coś dodane, czy też wprowadzono zmianę w jakimś rekordzie. Okazuje się, że implementacja takiego rozwiązania będzie się wiązać z dodaniem takich samych pól do wszystkich encji, a nam nie uśmiecha się kilkukrotne wpisywanie tego samego kodu.

Na szczęście TypeScript zapewnia możliwość korzystania z mechanizmu dziedziczenia klas; zdefiniujemy zatem klasę bazową, zawierającą wszystkie niezbędne pola, a następnie zmodyfikujemy wszystkie nasze klasy encji tak, by po niej dziedziczyły. Musimy jednak pamiętać, że *TypeORM* wymaga, by klasy encji dziedziczyły po określonej klasie bazowej frameworka, gdyż w przeciwnym razie nie będą w stanie korzystać z jego API. Dlatego też do naszej klasy bazowej dodamy klasę bazową *TypeORM*:

1. Utwórz plik *Auditable.ts* i zapisz w nim poniższy kod:

```
import { Column, BaseEntity } from "typeorm";

export class Auditable extends BaseEntity {
  @Column("varchar", {
    name: "CreatedBy",
    length: 60,
    default: () => `getpgusername()`,
    nullable: false,
  })
  createdBy: string;
```

*getpgusername* to funkcja Postgresa zwracająca nazwę używanego konta użytkownika; to właśnie ta nazwa będzie domyślnie zapisywana w tym polu, chyba że jawnie podamy inną.

```
@Column("timestamp with time zone", {
  name: "CreatedOn",
  default: () => `now()`,
  nullable: false,
})
createdOn: Date;
```

W tym polu domyślnie będą zapisywane bieżące data i czas (dzięki zastosowaniu funkcji *now()*), chyba że jawnie podamy inną wartość.

Jak widać, przeznaczenie tych pól jest raczej oczywiste. Zwróć jednak uwagę na to, że nasza klasa *Auditable* dziedziczy po klasie *BaseEntity* frameworka



TypeORM. To właśnie fakt rozszerzania tej klasy sprawia, że nasze klasy encji są w stanie łączyć się z Postgresem przy użyciu możliwości zapewnianych przez TypeORM:

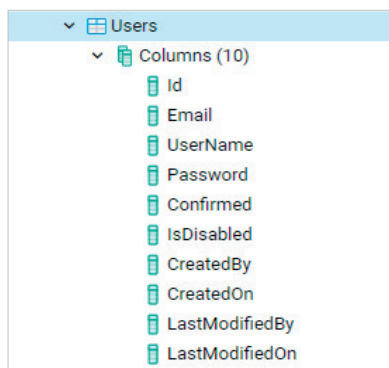
```
@Column("varchar", {
  name: "LastModifiedBy",
  length: 60,
  default: () => `getpgusername()`,
  nullable: false,
})
lastModifiedBy: string;

@Column("timestamp with time zone", {
  name: "LastModifiedOn",
  default: () => `now()`,
  nullable: false,
})
lastModifiedOn: Date;
}
```

2. No dobrze, w ten sposób utworzyliśmy nową klasę bazową Auditable. Teraz musimy zadbać, by nasze klasy encji ją rozszerzały. Na szczęście ta modyfikacja jest bardzo prosta. Wystarczy w deklaracji klasy dodać słowo kluczowe extends, a za nim nazwę klasy: Auditable, jak pokazałem poniżej na przykładzie klasy User:

```
export class User extends Auditable {
```

Wprowadź analogiczne zmiany we wszystkich klasach encji, a następnie ponownie uruchom serwer (nie zapomnij także o dodaniu niezbędnych instrukcji importu). Po odświeżeniu tabel prezentowanych w aplikacji *pgAdmin* w tabeli Users powinieneś zobaczyć kolumny przedstawione na rysunku 14.12.



**Rysunek 14.12.** Tablica Users z kolumnami zdefiniowanymi w klasie Auditable

Fantastycznie! Teraz możemy przystąpić do tworzenia własnej biblioteki repozytorium, której kod będzie się odwoływał do naszej bazy danych. Ponieważ w poprzednim rozdziale, pt. „Przygotowywanie stanu sesji przy użyciu Expressa i Redisa”, zaimplementowaliśmy obsługę sesji, zatem teraz w pierwszej kolejności zajmiemy się kodem związanym z uwierzytelnianiem.

1. Zanim zajmiemy się głównym kodem, musimy jednak zrobić coś jeszcze. Być może pamiętasz z rozdziału 11., pt. „Czego się nauczysz — aplikacja internetowego forum”, że w kodzie aplikacji stosowaliśmy funkcję `isPasswordValid`, której używaliśmy do sprawdzania, czy hasło użytkownika jest dostatecznie długie i złożone. Ten sam kod będziemy musieli zastosować na serwerze, gdyż, jak już wcześniej wspominałem, walidacja danych powinna być wykonywana zarówno po stronie klienta, jak i serwera. Dlatego też, w ramach tymczasowego rozwiązania, skopiuj plik *PasswordValidator.js* oraz strukturę katalogu *common/validators* do projektu serwerowej części aplikacji. W dalszej części książki pokażę, w jaki sposób można współużytkować kod w różnych projektach.
2. Przy okazji przygotujemy walidator do sprawdzania poprawności podawanych adresów poczty elektronicznej. W tym samym katalogu, *common/validators*, utwórz zatem plik *EmailValidator.ts* i zapisz w nim następujący kod:

```
export const isEmailValid = (email: string) => {
  if (!email) return "Adres e-mail nie może być pusty.";
```

Na samym początku sprawdzamy, czy adres e-mail nie jest pusty.

```
  if (!email.includes("@")) {
    return "Proszę podać poprawny adres e-mail.";
```

W tym fragmencie kodu sprawdzamy, czy w adresie występuje znak "@".

```
  }
  if (/\\s+/.test(email)) {
    return "W adresie e-mail nie można umieszczać znaków odstępu.";
```

A kolejny sprawdza, czy w adresie występują jakieś znaki odstępu.

```
  }
  return "";
};
```

Jeśli nie wykryto żadnych błędów, funkcja zwraca pusty łańcuch znaków.

3. Teraz w katalogu *src/repo* utwórz plik *UserRepo.ts* i zapisz w nim następujący kod:

```
import { User } from "../User";
import bcrypt from "bcryptjs";
import { isPasswordValid } from "../common/validators/PasswordValidator";
import { isEmailValid } from "../common/validators/EmailValidator";
```

W pierwszej kolejności importujemy niezbędne typy i funkcje, w tym przygotowane wcześniej walidatory.

```
const saltRounds = 10;
```

Stałą `saltRounds` będziemy używać do szyfrowania hasła, o czym przekonasz się już niebawem.

```
export class UserResult {
  constructor(public messages?: Array<string>, public user?: User) {}
}
```

Typu `UserResult` będziemy używali do przechowywania informacji o tym, czy podczas uwierzytelniania wystąpiły jakieś błędy. Jak widać, klasa ta stanowi

w zasadzie jedynie opakowanie dla obiektu `User`. Tej klasy będziemy używali jako typu wyniku zwracanego przez nasze funkcje. Zastosujemy takie rozwiązanie, gdyż podczas wykonywania żądań sieciowych lub innych złożonych wywołań, często może się zdarzyć, że coś pójdzie niezgodnie z naszymi oczekiwaniami. Dlatego też przydatna jest możliwość dołączania do obiektów jakiegoś rodzaju komunikatu o błędzie lub statusie. Zwróć uwagę na to, że obie składowe tej klasy, zarówno `messages`, jak i `user` są opcjonalne. Takie rozwiązanie bardzo się nam przyda, kiedy już zaczniemy używać tej klasy w kodzie.

```
export const register = async (
  email: string,
  userName: string,
  password: string
): Promise<UserResult> => {
```

To początek kodu funkcji `register`.

```
  const result = isPasswordValid(password);
  if (!result.isValid) {
    return {
      messages: [
        "Hasło musi mieć co najmniej 8 znaków i zawierać 1 dużą literę,  

        ↪ 1 cyfrę i 1 symbol.",
      ],
    };
  }

  const trimmedEmail = email.trim().toLowerCase();
  const emailErrorMsg = isEmailValid(trimmedEmail);
  if (emailErrorMsg) {
    return {
      messages: [emailErrorMsg],
    };
  }
```

W tym fragmencie kodu wywołujemy nasze dwa walidatory: `isPasswordValid` oraz `isEmailValid`. Zauważ, że w przypadku wykrycia błędów zwracany jest literal obiektowy, w którym nie występuje składowa `user`. Pamiętaj, że TypeScript zwraca uwagę jedynie na to, by struktura obiektu odpowiadała strukturze zadeklarowanego typu. W tym przypadku, składowa `user` klasy `UserResult` jest opcjonalna, dlatego też nic nie stoi na przeszkodzie, byśmy mogli utworzyć obiekt tej klasy pozbawiony składowej `user`. TypeScript jest naprawdę elastyczny.

```
  const salt = await bcrypt.genSalt(saltRounds);
  const hashedPassword = await bcrypt.hash(password, salt);
```

W tym fragmencie kodu szyfrujemy hasło, używając do tego celu stałej `saltRounds` i biblioteki `bcrypt.js`.

```
  const userEntity = await User.create({
    email: trimmedEmail,
    userName,
    password: hashedPassword,
  }).save();
```

Następnie, jeśli walidacja danych nie wykazała żadnych nieprawidłowości, tworzymy obiekt klasy User i natychmiast go zapisujemy, używając przy tym odpowiednio metod create i save. Obie te metody są udostępniane przez TypeORM. Pamiętaj także, że wprowadzając zmiany w obiektach encji, **zawsze** musisz wywołać funkcję save, gdyż w przeciwnym razie dane nie zostaną zachowane na serwerze.

```
userEntity.password = ""; // Hasło puste, ze względów bezpieczeństwa
return {
  user: userEntity,
};
};
```

W końcu zwracamy nowy obiekt encji. Zwróć uwagę na to, że tym razem zwracamy literal obiektowy zawierający tylko składową user, bez składowej messages, gdyż wcześniej nie wystąpiły żadne błędy.

4. A teraz spróbujmy zastosować naszą nową funkcję register w rzeczywistym odwołaniu sieciowym. Zaktualizuj plik *index.ts* zgodnie z poniższymi przykładami:

```
import express from "express";
import session from "express-session";
import connectRedis from "connect-redis";
import Redis from "ioredis";
import { createConnection } from "typeorm";
import { register } from "../repo/UserRepo";
import bodyParser from "body-parser";
```

Zwróć uwagę na to, że teraz importujemy oprogramowanie pośrednie body-parser.

```
require("dotenv").config();

const main = async () => {
  const app = express();
  const router = express.Router();

  await createConnection();
  const redis = new Redis({
    port: Number(process.env.REDIS_PORT),
    host: process.env.REDIS_HOST,
    password: process.env.REDIS_PASSWORD,
  });
  const RedisStore = connectRedis(session);
  const redisStore = new RedisStore({
    client: redis,
  });

  app.use(bodyParser.json());
```

Na końcu tego fragmentu kodu konfigurujemy bodyParser, dzięki czemu będziemy mogli odczytywać przesłane w żądaniu parametry zapisane w kodzie JSON.

```
app.use(
  session({
    store: redisStore,
    name: process.env.COOKIE_NAME,
    sameSite: "Strict",
```

```

    secret: process.env.SESSION_SECRET,
    resave: false,
    saveUninitialized: false,
    cookie: {
      path: "/",
      httpOnly: true,
      secure: false,
      maxAge: 1000 * 60 * 60 * 24,
    },
  } as any)
);

```

Ten kod pozostaje bez zmian:

```

app.use(router);
router.post("/register", async (req, res, next) => {
  try {
    console.log("params", req.body);
    const userResult = await register(
      req.body.email,
      req.body.userName,
      req.body.password
    );
    if (userResult && userResult.user) {
      res.send(`Utworzono nowego użytkownika o identyfikatorze: ${userResult.
        ↪user.id}`);
    } else if (userResult && userResult.messages) {
      res.send(userResult.messages[0]);
    } else {
      next();
    }
  } catch (ex) {
    res.send(ex.message);
  }
});

```

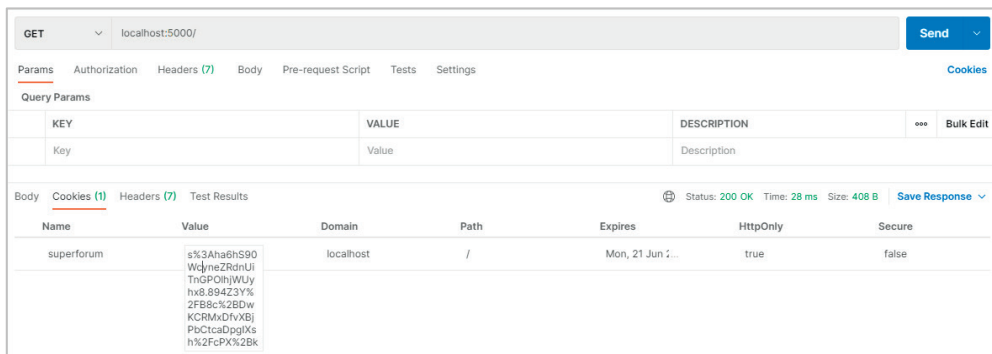
Jak widać, usunęliśmy z kodu wcześniejszą trasę `get` i zastąpiliśmy ją trasą `post` obsługującą żądania kierowane na adres `/register`. W tym kodzie wywołujemy funkcję `register` z pliku *UserRepo.ts*, a jeśli zostanie ona wykonana prawidłowo, przesyłamy w odpowiedzi komunikat z identyfikatorem nowego użytkownika. W razie niepowodzenia przesyłamy w odpowiedzi komunikat zwrócony przez wywołanie funkcji `register`. W tym przypadku zwracamy pierwszy komunikat; w rozdziale 15., pt. „Dodawanie schematu GraphQL-a — część 1.”, wszystkie te trasy usuniemy i zastąpimy odpowiednimi odwołaniami do GraphQL-a

W tym kodzie przesyłamy w odpowiedziach komunikaty o błędach wyłącznie w celach demonstracyjnych. W produkcyjnej wersji kodu nie będziemy chcieli, by ewentualne błędy powodowały przekazywanie do użytkownika jakichkolwiek komunikatów informacyjnych. Dla użytkowników są one kłopotliwe i wprowadzają zamieszanie, a w niektórych przypadkach mogą także otwierać witrynę na ataki.

```
app.listen({ port: process.env.SERVER_PORT }, () => {
  console.log(`Serwer działa na porcie ${process.env.SERVER_PORT}`);
});
};
main();
```

A teraz przejdźmy do testów. Jednak do tego celu będziemy musieli skorzystać z narzędzia Postman, zamiast używanego wcześniej programu curl. Postman to bezpłatna aplikacja pozwalająca na generowanie żądań GET i POST, która dodatkowo potrafi akceptować ciasteczka sesji. Aplikacja ta jest bardzo łatwa w użyciu:

1. W przeglądarce wyświetl stronę <https://www.postman.com/downloads> i pobierz program instalacyjny w wersji dla używanego systemu operacyjnego.
2. Zgodnie z instrukcją, pierwszą czynnością, jaką powinieneś wykonać po uruchomieniu aplikacji Postman, jest żądanie GET skierowane do strony głównej serwera. W pliku *index.ts* (który znajdziesz w przykładach dołączonych do książki) przygotowałem prostą trasę GET obsługującą żądania kierowane na główny adres serwera (<http://localhost:5000/>), która inicjuje sesję oraz używane przez nią ciasteczko (patrz rysunek 14.13).



**Rysunek 14.13.** Przesłanie żądania na główny adres serwera przy użyciu aplikacji Postman

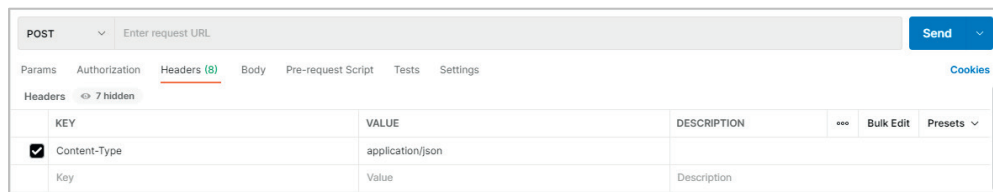
A oto w jaki sposób możesz wygenerować takie samo żądanie GET:

1. Na aktywnej karcie ze słowem *GET* w nazwie, przy lewej krawędzi karty powinieneś zobaczyć rozwijalną listę. Wybierz z niej opcję *GET*. Następnie w polu z prawej strony wpisz adres URL. Ponieważ żądanie nie będzie mieć żadnych parametrów, wystarczy, że klikniesz przycisk *Send*.
2. W dolnej części karty pojawi się panel z czterema kartami; kliknij tę o nazwie *Cookies*. Powinieneś zobaczyć listę ciasteczek, a na niej tylko jedną pozycję: ciasteczko *superforum*.

Teraz dysponujesz już ciasteczkiem koniecznym do przechowywania stanu sesji. Możemy zatem kontynuować testy, zaczynając od sprawdzenia działania funkcji *register*:

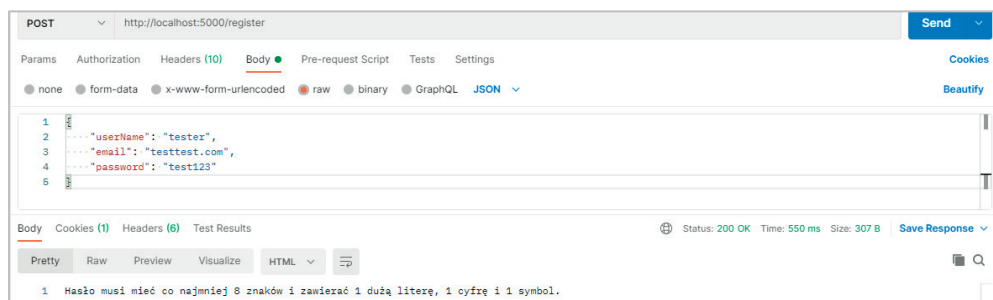
1. Otwórz nową kartę, z listy typu żądania wybierz *POST*, a w polu adresu wpisz <http://localhost:5000/register>.

2. Kliknij kartę *Headers* i poniżej wpisz nowy klucz Content-Type, jak pokazałem na rysunku 14.14.



Rysunek 14.14. Nagłówek Content-Type

3. Następnie przejdź na kartę *Body* wyświetloną z lewej strony karty *Headers*, a w wierszu poniżej zaznacz przycisk opcji *raw* i opcję *JSON* z listy wyświetlonej po prawej stronie przycisków opcji. Następnie w obszarze tekstowym wpisz kod JSON przedstawiony na rysunku 14.15.

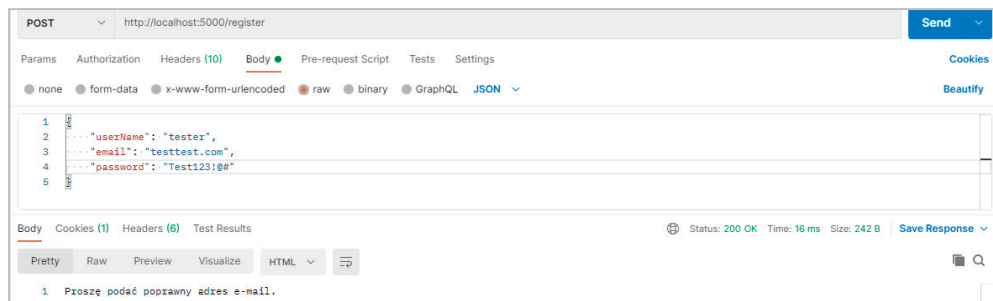


Rysunek 14.15. Nieudana rejestracja

Jak widać, w żądaniu przesyłamy nazwę użytkownika (*userName*), nieprawidłowy adres e-mail (*email*) oraz nieprawidłowe hasło (*password*).

Niemniej jednak, wyświetlenie tych informacji jest pozytywnym objawem, gdyż pokazuje, że walidacja działa prawidłowo.

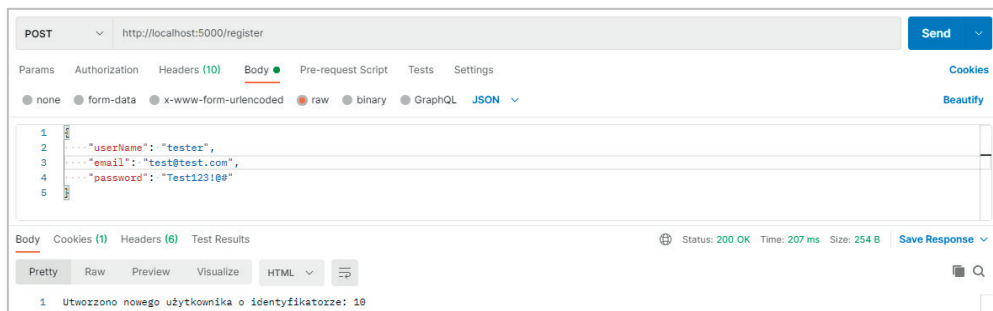
4. Spróbujmy zatem poprawić hasło i ponownie przesłać żądanie. Zmień hasło na `Test123!@#` i ponownie prześlij żądanie (patrz rysunek 14.16):



Rysunek 14.16. Ponowna próba rejestracji

Teraz powinniśmy zobaczyć komunikat **Proszę podać poprawny adres e-mail**. Także tym razem wynik zapytania jest zgodny z naszymi oczekiwaniami, gdyż nie da się ukryć, że podany w żądaniu adres e-mail jest nieprawidłowy.

5. Spróbujmy jeszcze raz. Zmień adres e-mail podany w zawartości zapytania na `test@test.com` i ponownie prześlij zapytanie (patrz rysunek 14.17).



Rysunek 14.17. Udana rejestracja użytkownika

Treść zwróconego komunikatu, **Utworzono nowego użytkownika o identyfikatorze: 10** pokazuje, że użytkownik faktycznie został utworzony i zapisany w bazie danych.

Na ostatnim rysunku jest widoczny identyfikator 10, gdyż wcześniej przeprowadziłem kilka dodatkowych testów, przygotowując się do pisania tego rozdziału. Wartości kolumn z identyfikatorami zazwyczaj zaczynają się do 1. Jeśli nie zobaczysz tego wyniku, to upewnij się, że na samym początku sesji z aplikacją Postman wygenerowałeś zapytanie GET na główny adres serwera (`http://localhost:5000/`).

6. Super! Zadziałało! A teraz wyświetl zawartość tabeli `Users`, aby upewnić się, czy użytkownik faktycznie został zapisany w bazie (patrz rysunek 14.18).

The screenshot shows the DBeaver SQL editor with the query `SELECT * FROM public.Users;` executed. The results are displayed in a table with the following columns and data:

ID [PK] bigint	Email character varying (120)	Username character varying (60)	Password character varying (100)	Confirmed boolean	IsDisabled boolean	CreatedBy character varying (60)	CreatedOn timestamp with time zone	LastModifiedBy character varying (60)	LastModifiedOn timestamp with time zone
10	test@test.com	tester	\$2a\$10\$UAFD/L032maVLo...	false	false	superformuzk	2021-06-21 00:48:55.148449+02	superformuzk	2021-06-21 00:48:55.148449+02

Rysunek 14.18. Nowy użytkownik dodany do tabeli `Users`

Przedstawione zapytanie możesz wygenerować klikając tabelę `Users` prawym przyciskiem myszy i wybierając z menu podręcznego opcję *Scripts/SELECT Script*. Aby wykonać to zapytanie, wystarczy kliknąć przycisk *Execute/Refresh* na pasku narzędzi powyżej zapytania; przypomina on wyglądem charakterystyczny przycisk odtwarzania. Jak widać, nasz nowy użytkownik faktycznie został zapisany w bazie danych.



7. Kolejnym krokiem będzie dodanie do pliku *UserRepo.ts* funkcji *login*. Dodaj na jego końcu poniższy fragment kodu:

```
export const login = async (
  userName: string,
  password: string
): Promise<UserResult> => {
  const user = await User.findOne({
    where: { userName },
  });
  if (!user) {
    return {
      messages: [userNotFound(userName)],
    };
  }

  if (!user.confirmed) {
    return {
      messages: ["Użytkownik jeszcze nie potwierdził swojego adresu  
✉e-mail."],
    };
  }

  const passwordMatch = await bcrypt.compare(password, user?.password);
  if (!passwordMatch) {
    return {
      messages: ["Hasło jest nieprawidłowe."],
    };
  }

  return {
    user: user,
  };
};
```

Nie ma tu zbyt wiele do pokazania. Na samym początku funkcji próbujemy znaleźć użytkownika o podanej nazwie. Jeśli nie uda się go znaleźć, to zwracamy odpowiedź ze stosownym komunikatem, używając do tego celu funkcji *userNotFound*. Zastosowałem w tym miejscu funkcję, gdyż tego samego komunikatu będziemy używać także w innych miejscach. Funkcja *userNotFound* jest bardzo prosta, więc nie będę przedstawiał jej kodu (możesz go znaleźć w kodach dołączonych do książki). Jeśli użytkownika uda się znaleźć, to w pierwszej kolejności sprawdzamy, czy jego konto zostało potwierdzone. Jeśli nie zostało, to zwracamy błąd ze stosownym komunikatem. Następnie sprawdzamy hasło; używamy przy tym biblioteki *bcryptjs*, gdyż także jej używaliśmy do zaszyfrowania hasła podczas rejestrowania użytkownika. Jeśli hasła nie pasują, zwracamy odpowiedź ze stosownym błędem. Jeśli hasła pasują, to zwracamy pobranego z bazy użytkownika (*user*).

8. Teraz użyjemy funkcji *login* w kodzie. Zmodyfikuj plik *index.ts*, dodając do niego nową trasę, bezpośrednio poniżej trasy do rejestracji:

```
router.post("/login", async (req, res, next) => {
  try {
    console.log("params", req.body);
```

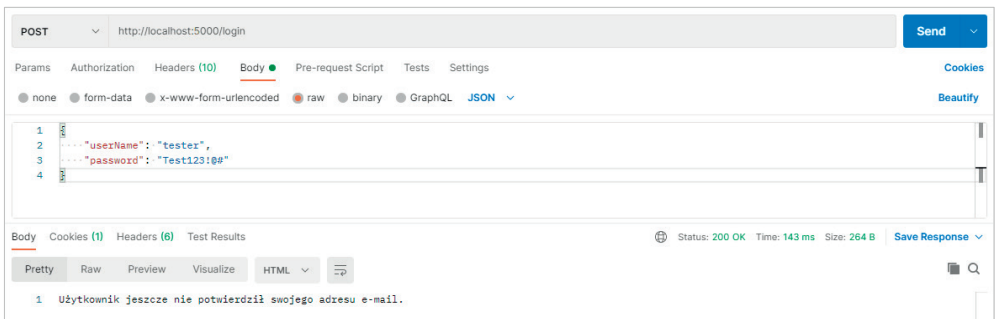
```

const userResult = await login(req.body.userName, req.body.password);
if (userResult && userResult.user) {
  req.session!.userId = userResult.user?.id;
  res.send(`Użytkownik się zalogował, userId: ${req.session!.
    ↪userId}`);
} else if (userResult && userResult.messages) {
  res.send(userResult.messages[0]);
} else {
  next();
}
} catch (ex) {
  res.send(ex.message);
}
});

```

Ta trasa jest bardzo podobna do trasy `register`. Jednak tu zapisujemy identyfikator użytkownika (`userId`), a następnie przesyłamy komunikat używając tej sesji.

9. Spróbujmy wykonać tę trasę i przekonajmy się, co się stanie. Przejdź do aplikacji Postman i wykonaj żądanie przedstawione na rysunku 14.19. **Pamiętaj** o dodaniu nagłówka `Content-Type` na karcie *Headers*.

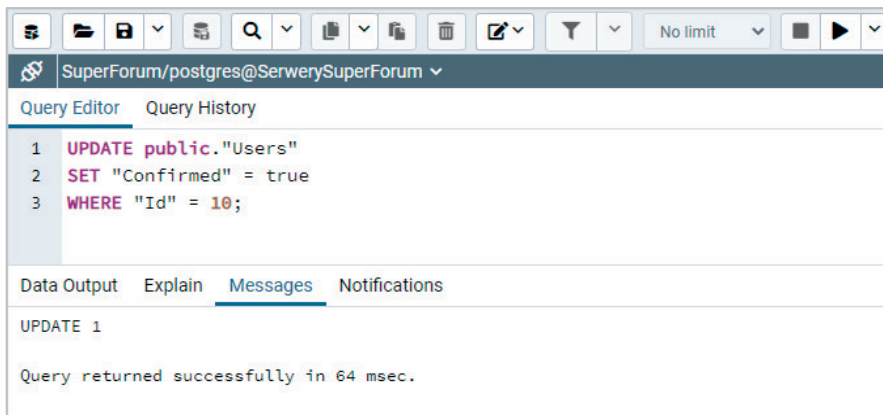


**Rysunek 14.19.** Trasa logowania

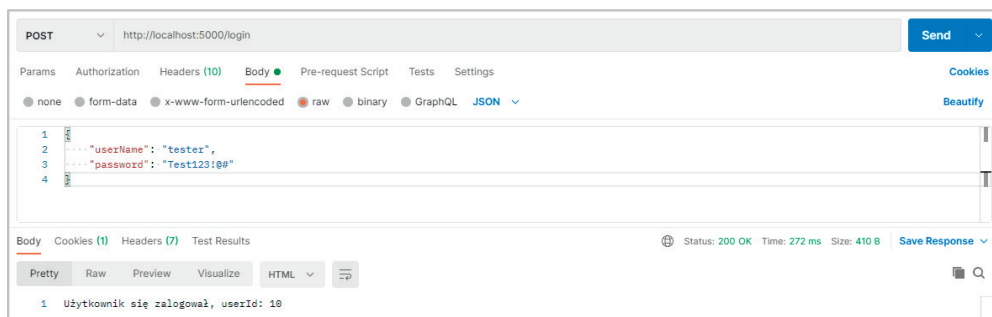
Ponownie dobrze jest się przekonać, że zaimplementowana walidacja działa poprawnie.

10. Przejdź do aplikacji pgAdmin, na ten sam ekran, którego wcześniej użyłeś do wykonania zapytania `SELECT`. Następnie wpisz zapytanie przedstawione na rysunku 14.20, aby w kolumnie `Confirmed` zapisać wartość `true`:
- Po wykonaniu zapytania powinieneś zobaczyć wyniki przedstawione na rysunku 14.20.
11. Przejdź do aplikacji Postman i ponownie spróbuj się zalogować (patrz rysunek 14.21).

Obecnie nasi użytkownicy mogą się już logować, a na podstawie zwracanych komunikatów widać, że jest przy tym używany stan sesji. Funkcję `logout` oraz korzystającą z niej trasę służącą do wylogowania użytkownika możesz znaleźć w przykładach dołączonych do książki. Są one bardzo proste, więc nie będę ich tu przedstawiał.



Rysunek 14.20. Aktualizacja pola potwierdzenia w rekordzie użytkownika



Rysunek 14.21. Udane logowanie

Jeśli Twoje próby zapisania sesji nie będą dawać oczekiwanych rezultatów, to upewnij się, że działa usługa Redisa.

Doskonale! Udało się nam już zrobić naprawdę dużo. Dysponujemy już działającymi mechanizmami uwierzytelniania korzystającymi z sesji; ale to jeszcze nie wszystko. Musimy zaimplementować możliwości dodawania wątków (Thread) i odpowiedzi (ThreadItem), jak również ich pobierania z bazy danych. Zaczniemy od wątków — obiektów klasy Thread:

1. Zanim utworzymy nową klasę ThreadRepo, przygotujemy kilka funkcji pomocniczych. W pliku *UserRepo.ts* zdefiniowaliśmy typ *UserResult*, składający się z dwóch składowych: tablicy komunikatów oraz obiektu użytkownika. Zauważysz zapewne, że w kodach związanych z obsługą wątków (Thread), odpowiedzi (ThreadItem) oraz kategorii (ThreadCategory) zastosujemy podobne konstrukcje. Powinny one zawierać tablicę komunikatów oraz encje, choć tym razem będzie to tablica obiektów encji, a nie jeden obiekt.

Wygląda na to, że to świetna okazja, by wykorzystać typy generyczne TypeScriptu i przygotować jeden typ wyniku, którego będziemy mogli używać do obsługi wszystkich klas encji. Utwórzmy zatem takie dwa generyczne typy obiektu

wyników, które nazwiemy odpowiednio `QueryArrayResult` oraz `QueryOneResult`; pierwszy z nich będzie służył do reprezentacji wyniku zawierającego tablicę obiektów encji, a drugi — wyniku zawierającego tylko jeden obiekt encji. Informacje o typach generycznych podałem w rozdziale 2., pt. „Prezentacja języka TypeScript”.

Utwórz nowy plik, o nazwie *QueryArrayResult.ts*, i zapisz w nim następujący fragment kodu:

```
export class QueryArrayResult<T> {
  constructor(public messages?: Array<string>, public entities?: Array<T>) {}
}

export class QueryOneResult<T> {
  constructor(public messages?: Array<string>, public entity?: T) {}
}
```

Jak widać, obie klasy są bardzo podobne do typu `UserResult`, choć używają generycznego typu `T` do określenia typu przechowywanych encji.

Pakiet `pg` także udostępnia typ o nazwie `QueryArrayResult`. Dlatego importując zależności upewnij się, że zaimportujesz nasz plik, a nie pakiet `pg`.

2. Teraz możemy już zastosować nowy typ `QueryArrayResult` w kodzie funkcji `ThreadRepo`. W katalogu *repo* utwórz nowy plik, *ThreadRepo.ts*, i zapisz w nim następujący fragment kodu:

```
export const createThread = async (
  userId: string | undefined | null,
  categoryId: string,
  title: string,
  body: string
): Promise<QueryOneResult<Thread>> => {
```

Przekazywane parametry są niezbędne, gdyż każdy wątek musi być skojarzony z użytkownikiem oraz kategorią. Zwróć jednak uwagę na to, że wartość identyfikatora użytkownika (`userId`) pochodzi z sesji.

```
  const titleMsg = isThreadTitleValid(title);
  if (titleMsg) {
    return {
      messages: [titleMsg],
    };
  }
  const bodyMsg = isThreadBodyValid(body);
  if (bodyMsg) {
    return {
      messages: [bodyMsg],
    };
  }
}
```

W tym fragmencie kodu sprawdzamy poprawność tytułu i treści wpisu.

```
if (!userId) {
  return {
    messages: ["Użytkownik nie jest zalogowany."],
  };
}
```

```

    };
  }
  const user = await User.findOne({
    id: userId,
  });
};

```

W tym fragmencie kodu sprawdzamy, czy został przekazany identyfikator użytkownika (`userId`), a jeśli został, to próbujemy pobrać z bazy danych odpowiadającego mu użytkownika. Tego obiektu użytkownika (`user`) będziemy potrzebowali w dalszej części kodu do utworzenia obiektu `Thread`.

```

const category = await ThreadCategory.findOne({
  id: categoryId,
});
if (!category) {
  return {
    messages: ["Nie znaleziono kategorii."],
  };
}

```

W tym fragmencie pobieramy obiekt `category`, gdyż także on będzie nam potrzebny do utworzenia nowego obiektu `Thread`.

```

const thread = await Thread.create({
  title,
  body,
  user,
  category,
}).save();
if (!thread) {
  return {
    messages: ["Nie udało się utworzyć wątku."],
  };
}

```

Jak widać, w celu utworzenia nowego wątku przekazujemy następujące argumenty: `title`, `body`, `user` oraz `category`.

```

return {
  messages: ["Wątek został pomyślnie utworzony."],
};
};

```

Zwracamy komunikat, gdyż obiekt nie jest nam potrzebny. Co więcej, zwracanie niepotrzebnego obiektu jest nieefektywne z punktu widzenia projektu API oraz obciążenia generowanego przez ten API.

3. Zanim będziemy mogli kontynuować, musimy dodać do bazy danych kilka kategorii, gdyż są one niezbędne do prawidłowego działania funkcji `createThread`. W kodach źródłowych przykładów dołączonych do książki odszukaj plik *utils/InsertThreadCategories.txt*. Skopiuj całą jego zawartość i wklej do obszaru testowego na karcie *QueryEditor* w aplikacji *pgAdmin*; następnie wykonaj polecenia SQL. W efekcie w bazie danych do tabeli `ThreadCategories` zostanie dodanych pięć kategorii.

4. Kolejnym zadaniem będzie dodanie nowej trasy służącej do tworzenia wątków. Dodaj poniższy kod do pliku *index.ts*:

```
router.post("/createthread", async (req, res, next) => {
  try {
    console.log("userId", req.session);
    console.log("body", req.body);
    const msg = await createThread(
      req.session!.userId, // Zwróć uwagę na to, że userId pochodzi z sesji!
      req.body.categoryId,
      req.body.title,
      req.body.body
    );
```

W tym bardzo prostym fragmencie kodu przekazujemy parametry do wywołania funkcji `createThread`. Zwróć uwagę na to, że wartość `userId` jest pobierana z sesji, gdyż użytkownik musi być zalogowany, by mógł coś opublikować na forum.

W dalszej części kodu zwracamy komunikat.

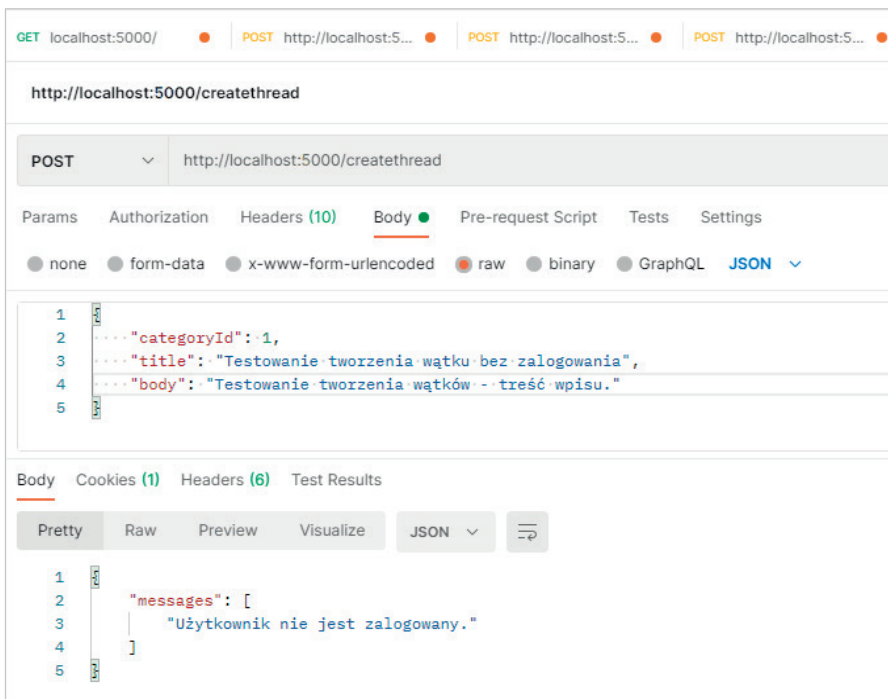
```
      res.send(msg);
    } catch (ex) {
      console.log(ex);
      res.send(ex.message);
    }
  });
```

5. Wypróbujmy teraz działanie tej nowej trasy. Najpierw jednak wyloguj się z naszej aplikacji, używając do tego celu aplikacji Postman. Możesz to zrobić generując żądanie na adres *http://localhost:5000/logout*. Jestem pewny, że obecnie potrafisz już wygenerować odpowiednie żądanie za pomocą aplikacji Postman. Kiedy już to zrobisz, wygeneruj żądanie odwołujące się do trasy `createthread` i miejmy nadzieję, że nasze mechanizmy walidacji zgłoszą błąd (patrz rysunek 14.22).

Jak widać, zgodnie z oczekiwaniami, żądanie nie przeszło walidacji.

6. Teraz zaloguj się do aplikacji, by ponownie została utworzona sesja. Jak wcześniej, użyj do tego aplikacji Postman. Kiedy już się zalogujesz, ponownie wypróbuj działanie trasy `createthread`; tym razem wszystko powinno się udać, a zwrócona odpowiedź powinna zawierać komunikat **Wątek został pomyślnie utworzony**.
7. No dobrze. Potrzebujemy jeszcze dwóch kolejnych funkcji. Pierwsza z nich będzie pobierać konkretny wątek na podstawie jego identyfikatora, a druga będzie pobierać wszystkie wątki należące do podanej kategorii. Dodaj poniższy kod do pliku *ThreadRepo.js*.

```
export const getThreadById = async (
  id: string
): Promise<QueryOneResult<Thread>> => {
  const thread = await Thread.findOne({ id });
  if (!thread) {
    return {
      messages: ["Nie udało się znaleźć wątku."],
    };
  }
}
```



Rysunek 14.22. Test trasy createthread

```

    return {
      entity: thread,
    };
  };
};

```

Funkcja `getThreadById` jest bardzo prosta. Jej działanie sprowadza się do wyszukania i zwrócenia jednego wątku, określonego na podstawie identyfikatora.

```

export const getThreadsByCategoryId = async (
  categoryId: string
): Promise<QueryArrayResult<Thread>> => {
  const threads = await Thread.createQueryBuilder("thread")
    .where(`thread."categoryId" = :categoryId`, { categoryId })
    .leftJoinAndSelect("thread.category", "category")
    .orderBy("thread.createdOn", "DESC")
    .getMany();
}

```

Funkcja `getThreadsByCategoryId`, której początkowy fragment zamieściłem powyżej, jest znacznie bardziej interesująca. `Thread.createQueryBuilder` jest specjalną funkcją TypeORM, zapewniającą możliwość budowania bardziej złożonych zapytań. Przekazany do niej parametr "thread" jest nazwą zastępczą reprezentującą tabelę `Threads`. Jeśli przyjrzyysz się reszcie zapytania, na przykład klauzuli `where`, przekonasz się, że parametr "thread" jest jedynie prefiksem dla pól i zależności bazy danych. Zastosowanie funkcji `leftJoinAndSelect` oznacza, że w zbiorze wyników chcemy zwrócić powiązane encje, którymi w tym przypadku są obiekty

ThreadCategory. Przeznaczenie funkcji `orderBy` jest oczywiste, a funkcja `getMany` powoduje zwrócenie wszystkich rekordów pobranych z bazy.

```
if (!threads || threads.length === 0) {
  return {
    messages: ["Nie udało się znaleźć wątków w podanej kategorii."],
  };
}
console.log(threads);
return {
  entities: threads,
};
};
```

8. Pozostała część kodu jest bardzo prosta i nie wymaga dodatkowych wyjaśnień. Przetestujmy zatem trasę obsługiwaną przez funkcję `getThreadsById`. Dodaj poniższy kod na końcu pliku `index.ts`:

```
router.post("/threadsbycategory", async (req, res, next) => {
  try {
    const threadResult = await
      getThreadsByCategoryId(req.body.categoryId);
```

W tym fragmencie wywołujemy funkcję `getThreadsByCategoryId`, przekazując do niej identyfikator kategorii (`categoryId`).

```
    if (threadResult && threadResult.entities) {
      let items = "";
      threadResult.entities.forEach((th) => {
        items += th.title + ", ";
      });
      res.send(items);
    } else if (threadResult && threadResult.messages) {
      res.send(threadResult.messages[0]);
    }
  }
}
```

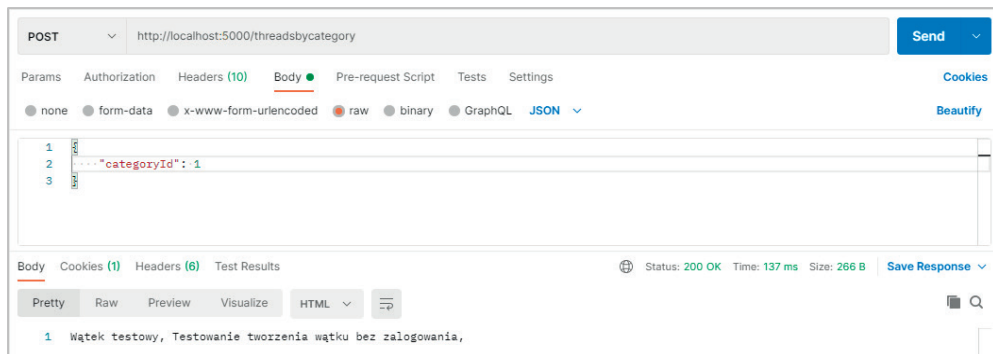
W kodzie znajdującym się w klauzuli `if else` dysponujemy już wszystkimi wątkami lub komunikatem o błędzie.

```
    } catch (ex) {
      console.log(ex);
      res.send(ex.message);
    }
  });
```

9. Pozostała część kodu nie uległa zmianie. Używając aplikacji Postman wykonaj żądanie do tej trasy; jego postać i wyniki, które powinieneś zobaczyć, przedstawiłem na rysunku 14.23.

Testy funkcji `getThreadById` oraz trasy, która jej używa, możesz wykonać we własnym zakresie, gdyż są one bardzo proste. Także w tym przypadku, gdybyś miał jakieś problemy, to kody źródłowe aplikacji znajdziesz w przykładach dołączonych do książki.





**Rysunek 14.23.** Testowanie trasy `treadsbycategory`

Kod obsługujący odpowiedzi (obiekty `ThreadItem`) jest niemal identyczny i możesz go znaleźć w przykładach dołączonych do książki. Nie będę go tutaj przedstawiał. Potrzebujemy jeszcze paru dodatkowych funkcji, na przykład do pobierania kategorii (`ThreadCategories`), które będą wyświetlane w aplikacji klienckiej, w komponencie `LeftMenu`. Oprócz tego musimy jeszcze pobierać punkty wątków i odpowiedzi. No i w końcu potrzebujemy także odpowiednich danych wątków i odpowiedzi do wyświetlania na ekranie `UserProfile`. Jednak te funkcje będą w znacznym stopniu powielaly rozwiązania przedstawione wcześniej w tym rozdziale, a do ich wykorzystania będziemy musieli przygotować trasy, które i tak w końcu byśmy usunęli, kiedy zaczęlibyśmy pisać kod serwera GraphQL. Dlatego też zajmiemy się tymi wszystkimi zagadnieniami w rozdziale 15., pt. „Dodawanie schematu GraphQL-a — część 1.”, w którym zaczniemy także integrować serwerowy kod GraphQL-a z klientką aplikacją Reacta.

Z tego podrozdziału nauczylesz się tworzyć warstwę repozytorium oraz wykonywać zapytania SQL w bazie Postgres, używając do tego celu frameworka `TypeORM`. Ze zdobytych tu informacji i umiejętności skorzystasz już w następnym rozdziale, kiedy zaczniemy integrować z naszą aplikacją serwer GraphQL. Jest to zatem ważna wiedza, której w przyszłości będziesz używał.

## Podsumowanie

Z tego rozdziału dowiedziałeś się, jak zainstalować i przygotować do użycia bazę danych Postgres oraz jak pobierać i modyfikować jej zawartość przy użyciu rozwiązań typu ORM, a konkretnie frameworka `TypeORM`. Zobaczyłeś także, w jaki sposób, dzięki utworzeniu warstwy repozytorium, można zadbać o właściwą separację kodu.

Z następnego rozdziału dowiesz się, jak zainstalować na serwerze GraphQL. Oprócz tego dokończymy w nim zapytania do bazy danych i zaczniemy integrować kod serwerowy z klientką aplikacją Reacta.

# Dodawanie schematu GraphQL-a — część 1.

W tym rozdziale będziemy kontynuować budowanie naszej aplikacji, do której dodamy schemat GraphQL-a. Zmiany będziemy wprowadzać zarówno w kodzie serwerowym, jak i w klienckiej aplikacji Reacta. Dokończymy także budowanie serwera Expressa, który zaimplementujemy integrować z kliencką częścią aplikacji.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- przygotowaniem definicji typów i resolverów GraphQL-a;
- zintegrowaniem mechanizmu uwierzytelniania z resolverami GraphQL-a;
- utworzeniem *hooków* Reacta do korzystania z serwera Apollo GraphQL.

---

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś rozumieć podstawowe zagadnienia związane z GraphQL-em oraz dysponować dobrą znajomością zagadnień związanych z tworzeniem aplikacji Reacta, pisanem aplikacji w środowisku Node, korzystaniem z bazy Postgres i magazynu Redis. Podobnie jak w poprzednich rozdziałach, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code** (VSCode).

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, anglojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial15* (*Chap15*).

Aby przygotować katalog do prac nad kodami, którymi będziemy zajmować się w tym rozdziale, wykonaj poniższe czynności:

1. Przejdź do katalogu *NaukaTypeScriptu* i utwórz nowy katalog, o nazwie *rozdzial15*.
2. Przejdź do katalogu *rozdzial14* i skopiuj cały katalog *super-forum-server* do katalogu *rozdzial15*. Upewnij się, że zostały skopiowane wszystkie pliki.
3. W nowym katalogu *super-forum-server* usuń katalog *node\_modules* oraz plik *package-lock.json*. Upewnij się, że znajdujesz się w katalogu *super-forum-server*, po czym wykonaj następujące polecenie:

```
npm install
```

4. Teraz upewnij się, że działa zarówno Redis, jak Postgres; informacje na ten temat znajdziesz odpowiednio w rozdziałach 13., pt. „Przygotowywanie stanu sesji przy użyciu Expressa i Redisa” oraz 14., pt. „Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM”. Następnie uruchom serwer wykonując polecenie:

```
npm start
```

5. Teraz skopiuj kliencką część aplikacji. Przejdź do katalogu *rozdzial13* i skopiuj do katalogu *rozdzial15* katalog *super-forum-client*. Upewnij się, że zostały skopiowane wszystkie pliki.
6. Z katalogu *super-forum-client* usuń katalog *node\_modules* oraz plik *package-lock.json*. Upewnij się, że jesteś w tym katalogu i wykonaj następujące polecenie:

```
npm install
```

7. Sprawdź, czy aplikacja działa, wykonując polecenie:

```
npm start
```

## Tworzenie definicji typów i resolverów dla serwerowego kodu GraphQL

W tym podrozdziale dodamy usługę GraphQL-a do serwera Expressa. Zaczniemy także przekształcać trasy, które przygotowaliśmy w poprzednim rozdziale, na zapytania GraphQL-a. Uzupełnimy także resztę potrzebnych wywołań, implementując je w formie zapytań GraphQL-a.

Zacznijmy od zintegrowania GraphQL-a z aplikacją Expressa (zagadnienia związane z GraphQL-em przedstawiałem już w rozdziale 9., pt. „Czym jest GraphQL?” oraz 10., pt. „Konfiguracja projektu Expressa z zależnościami od języków TypeScript i GraphQL”).

W tym rozdziale będę opisywał bardzo dużo kodu, przy czym nie będę w stanie przedstawić go w całości w tekście książki. Dlatego dobrze by było, gdybyś dość często zaglądał do kodów źródłowych przykładów dołączonych do książki. Zwróć uwagę na to, że przykłady dołączone do książki stanowią końcową, działającą wersję naszego projektu.

1. Zacznijmy od zainstalowania GraphQL-a. W tym celu wykonaj następujące polecenie:

```
npm i apollo-server-express graphql graphql-middleware graphql-tools
```

2. Kolejnym krokiem będzie utworzenie definicji typów, `typeDefs`. W katalogu `src` utwórz podkatalog `gql`, a w nim plik `typeDefs.ts`. W pliku `typeDefs.ts` zapisz poniższy fragment kodu:

```
import { gql } from "apollo-server-express";

const typeDefs = gql`
  scalar Date
```

W tym fragmencie kodu definiujemy nowy typ skalarny o nazwie `Date`, przeznaczony do reprezentowania dat i czasu; domyślnie nie jest on dostępny w GraphQL-u.

```
  type EntityResult {
    messages: [String!]
  }
```

Typ `EntityResult` będzie używany, kiedy zamiast encji resolvery będą zwracały błędy lub komunikaty.

```
  type User {
    id: ID!
    email: String!
    userName: String!
    password: String!
    confirmed: Boolean!
    isDisabled: Boolean!
    threads: [Thread!]
    createdBy: String!
    createdOn: Date!
    lastModifiedBy: String!
    lastModifiedOn: Date!
  }
```

W tym fragmencie kodu tworzymy typ `User`. Zwróć uwagę na jego powiązanie z typami `Thread` i `ThreadItem`. Zastosowaliśmy w nim także zdefiniowany wcześniej typ `Date`.

```
  type Thread {
    id: ID!
    views: Int!
    points: Int!
    isDisabled: Boolean!
    title: String!
    body: String!
```

```

    user: User!
    threadItems: [ThreadItem!]
    category: ThreadCategory!
    createdBy: String!
    createdOn: Date!
    lastModifiedBy: String!
    lastModifiedOn: Date!
  }

```

Tutaj tworzymy typ `Thread` i określamy powiązane z nim typy:

```
union ThreadResult = Thread | EntityResult
```

Skoro już zaczęliśmy implementować rzeczywistą aplikację, nadszedł czas, żeby wykorzystać niektóre z bardziej wyrafinowanych możliwości GraphQL-a. Jedną z nich jest typ `union`, którego odpowiednik występuje w języku TypeScript. Pozwala on na zwrócenie jednego, dowolnego typu z listy kilku różnych typów GraphQL-a. Na przykład, przedstawiony tu typ `ThreadResult` może reprezentować typ `Thread` lub typ `EntityResult`, lecz nie oba te typy jednocześnie. Już niebawem przedstawię zastosowanie tego typu, a wtedy jego działanie stanie się oczywiste.

```

type ThreadItem {
  id: ID!
  views: Int!
  points: Int!
  isDisabled: Boolean!
  body: String!
  user: User!
  thread: Thread!
  createdBy: String!
  createdOn: Date!
  lastModifiedBy: String!
  lastModifiedOn: Date!
}

```

Ten fragment kodu przedstawia definicję typu `ThreadItem`.

```

type ThreadCategory {
  id: ID!
  name: String!
  description: String!
  threads: [Thread!]!
  createdBy: String!
  createdOn: Date!
  lastModifiedBy: String!
  lastModifiedOn: Date!
}

```

Typ `Category` odwołuje się do typu `Thread` — wątków, które może zawierać kategoria.

```

type Query {
  getThreadById(id: ID!): ThreadResult
}
;

```

Tutaj mamy definicję zapytań i funkcję `getThreadById`. Zwróć uwagę na to, że funkcja ta zwraca wynik typu `ThreadResult`.

```
export default typeDefs;
```

3. A teraz, żeby rozpocząć korzystanie z serwera GraphQL-a, utwórzmy prosty plik resolvera. W tym celu, w katalogu *gql* utwórz plik *resolvers.ts* i zapisz w nim poniższy kod:

```
import { IResolvers } from "apollo-server-express";
interface EntityResult {
  messages: Array<string>;
}
```

Będziemy używali `EntityResult` jako typu wyników w przypadku zwracania błędów lub komunikatów stanu. Oprócz tego, dodaj odwzorowanie naszego typu do tego samego typu w pliku definicji typów:

```
const resolvers: IResolvers = {
  ThreadResult: {
    __resolveType(obj: any, context: GqlContext, info: any) {
      if (obj.messages) {
        return "EntityResult";
      }
      return "Thread";
    },
  },
}
```

Oto kolejna nowa możliwość GraphQL-a, której będziemy używać. Typ `ThreadResult` jest unią reprezentującą dwa typy: `Thread` i `EntityResult`. Ten resolver potrafi zauważyć, kiedy ma być zwrócony typ `ThreadResult`, wykrywa także, jakiego typu jest aktualnie przekazany obiekt. Sposób, w jaki będziemy określać zwracany typ jest dowolny i zależy tylko od nas; w tym przykładzie zastosowaliśmy proste sprawdzenie, czy istnieje właściwość `message` występująca w typie `EntityResult`; używamy do tego celu wyrażenia `obj.message`.

```
Query: {
  getThreadById: async (
    obj: any,
    args: { id: string },
    ctx: GqlContext,
    info: any
  ): Promise<Thread | EntityResult> => {
    let thread: QueryOneResult<Thread>;
    try {
      thread = await getThreadById(args.id);

      if (thread.entity) {
        return thread.entity;
      }
      return {
        messages: thread.messages ? thread.messages : [STANDARD_ERROR],
      };
    } catch (ex) {
```

```

        console.log(ex.message);
        throw ex;
    }
  },
}
}
export default resolvers;

```

O zapytaniach GraphQL-a dowiedziałeś się już w rozdziale 9., pt. „Czym jest GraphQL?”, dlatego nie będę tu dokładnie opisywał powyższego fragmentu kodu. Zwróć tylko uwagę na to, że pobieramy w nim wynik funkcji `getThreadById`, który jest typu `QueryOneResult`, a następnie, po zastosowaniu prostej logiki warunkowej, zwracamy faktyczną encję lub wynik typu `EntityResult`. Ponieważ w pliku definicji typów określiliśmy, że zapytanie zwraca wynik typu `ThreadResult`, więc GraphQL odwoła się do zapytania `ThreadResult` i określi, jaki typ należy zwrócić. Tego samego wzorca będziemy używali niemal we wszystkich odwołaniach związanych z warstwą repozytorium. Zagadnienia z tym związane opisałem w rozdziale 14., pt. „Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM”.

W naszej prostej aplikacji ograniczamy się do ponownego zgłaszania błędów, które mogą wystąpić. Jednak podczas pisania swojej aplikacji, pojawiające się w niej błędy powinieneś odpowiednio obsługiwać, co w najprostszym przypadku oznacza rejestrowanie błędów w dzienniku, by przynajmniej można było się im przyjrzeć.

W dalszej części rozdziału uzupełnimy ten kod, dodając do niego więcej zapytań i mutacji, jednak na razie skoncentrujemy się jedynie na dokończeniu podstawowych możliwości.

4. Teraz skopiuj do katalogu *src/gql* plik *GqlContext.ts* z katalogu *rozdzial10/gql-server/src*. Zgodnie z informacjami podanymi w rozdziale 9., pt. „Czym jest GraphQL?”, to właśnie w tym pliku tworzymy kontekst wywołań GraphQL-a zawierający obiekty żądania (`Request`) i odpowiedzi (`Response`).
5. Teraz otwórz plik *index.ts* i dodaj do niego kod związany z wykorzystaniem GraphQL-a. Ten nowy kod dodaj bezpośrednio przed wywołaniem funkcji `listen`, przy czym nie zapomnij dodać na początku pliku wszystkich niezbędnych instrukcji importu; aktualnie powinieneś już potrafić zrobić to samemu.

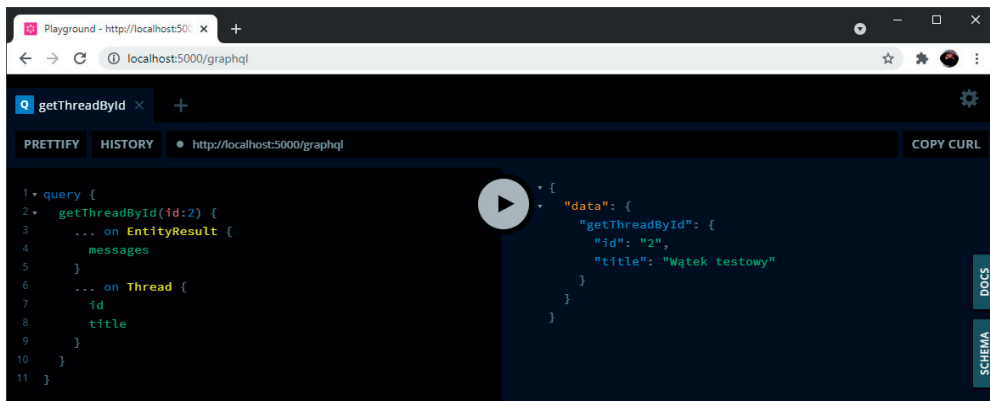
```

const schema = makeExecutableSchema({ typeDefs, resolvers });
const apolloServer = new ApolloServer({
  schema,
  context: ({ req, res }: any) => ({ req, res }),
});
apolloServer.applyMiddleware({ app, cors: false });

```

Ten kod jest bardzo podobny do tego, którego używaliśmy w rozdziale 9., pt. „Czym jest GraphQL?”; tworzymy w nim obiekt typu `ApolloServer` i przekazujemy do niego definicje typów (`typeDefs`), resolwery (`resolvers`) oraz instancję aplikacji Expressa.

- Przetestujmy efekty naszej dotychczasowej pracy, by upewnić się, że wszystko działa prawidłowo. Wyświetl w przeglądarce stronę <http://localhost:5000/graphql>. W ten sposób wyświetlisz w przeglądarce aplikację „placu zabaw” GraphQL-a, której już używaliśmy w rozdziale 9., pt. „Czym jest GraphQL?”. Następnie wykonaj zapytanie przedstawione na rysunku 15.1.



Rysunek 15.1. Pierwsze zapytanie wykonane na serwerze GraphQL-a

Jak widać, nasze wywołanie działa. Jedyna różnica w stosunku do wywołań, których używaliśmy wcześniej polega na tym, że wywołanie używane obecnie może zwracać dane dwóch różnych typów. Ze względu na to zastosowaliśmy składnię `... on <jakiś typ>`, aby określić, jaka encja oraz które z jej pól mają zostać zwrócone (mechanizm ten jest określany jako wpisane fragmenty, ang. *inline fragments*). Pamiętaj także, że Twoje identyfikatory, które będą zwracane przez zapytania, mogą być inne od moich, dlatego wykonując zapytania musisz używać identyfikatorów, które na pewno istnieją w bazie danych.

- No dobrze. Zajmijmy się zatem kolejnym zagadnieniem. Tym razem wybierzemy takie, które nie zwraca żadnej encji — funkcję `createThread`. W pierwszej kolejności na samym końcu pliku definicji typów (*typeDefs.ts*) dodaj poniższą mutację:

```

type Mutation {
  createThread(
    userId: ID!
    categoryId: ID!
    title: String!
    body: String!
  ): EntityResult
}
  
```

Zwróć uwagę na to, że funkcja `createThread` nie zwraca wyniku typu `ThreadResult`, a jedynie komunikat tekstowy. To jedyne, czego w jej przypadku potrzebujemy.

- Teraz zajmijmy się wprowadzeniem odpowiednich zmian w pliku resolverów. Dodaj poniższą funkcję jako mutację. Pamiętaj, że musisz także samodzielnie zaimportować wszystkie niezbędne typy i funkcje.



```

Mutation: {
  createThread: async (
    obj: any,
    args: { userId: string; categoryId: string; title: string; body:
      ↳string },
    ctx: GqlContext,
    info: any
  ): Promise<EntityResult> => {

```

Ta funkcja ma taką samą listę parametrów, jak inne mutacje, jednak tym razem zwracamy wynik typu `EntityResult`, gdyż nie potrzebujemy całej encji:

```

    let result: QueryOneResult<Thread>;
    try {
      result = await createThread(
        args.userId,
        args.categoryId,
        args.title,
        args.body
      );

```

W tym fragmencie wywołujemy funkcję `createThread` warstwy repozytorium i pobieramy zwrócony przez nią wynik.

```

      return {
        messages: result.messages
          ? result.messages
            : ["Wystąpił błąd"],
      };

```

W tym fragmencie zwracamy listę ewentualnych komunikatów określających status wyników.

```

    } catch (ex) {
      throw ex;

```

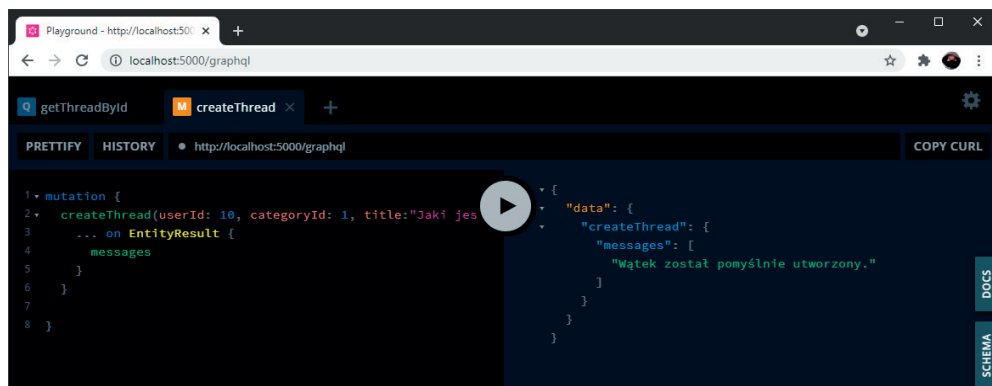
Raz jeszcze zwracam Ci uwagę na to, że w produkcyjnej wersji aplikacji nie należy ponownie zgłaszać tych samych wyjątków; zamiast tego powinieneś zapisać je w dzienniku lub spróbować w jakiś sposób obsłużyć błąd. Ja zdecydowałem się na ponowne zgłaszanie wyjątków tylko po to, by uprościć kod aplikacji i koncentrować się na opisywanych zagadnieniach, a nie na sprawach drugorzędnych.

```

    }
  },
}

```

9. A zatem, kiedy teraz wykonasz nowy kod, uzyskasz wyniki przedstawione na rysunku 15.2.
10. No dobrze, zajmijmy się jeszcze jednym wywołaniem związanym z wątkami. W pliku *ThreadRepo.ts* jest zdefiniowana funkcja `getThreadByCategoryId`, która zwraca tablicę obiektów `Thread`. Nastrocza ona pewnych problemów, gdyż typ `union GraphQL-a` i operator `|` nie obsługują tablic. Oznacza to, że w pliku definicji typów (*typeDefs.ts*) będziemy musieli utworzyć nową encję, reprezentującą tablicę obiektów `Thread`, i dopiero później utworzyć odpowiedni typ unii. Zmodyfikuj zatem plik definicji typów, dodając poniższy fragment bezpośrednio poniżej obecnego kodu:

Rysunek 15.2. Próba działania funkcji `createThread`

```

type ThreadArray {
  threads: [Thread!]
}

union ThreadArrayResult = ThreadArray | EntityResult
  
```

A zatem, najpierw utworzyliśmy encję, zwracającą tablicę obiektów `Thread`; a następnie utworzyliśmy typ unii który może zawierać albo encję tego nowego typu, albo typu `EntityResult`.

A teraz dodaj poniższy fragment kodu poniżej zapytania `getThreadById`:

```
getThreadsByCategoryId(categoryId: ID!): ThreadArrayResult!
```

11. Teraz możemy przygotować kod resolwera korzystającego z tej funkcji. Do sekcji Query w pliku `resolvers.ts` dodaj poniższy kod:

```

getThreadsByCategoryId: async (
  obj: any,
  args: { categoryId: string },
  ctx: GqlContext,
  info: any
): Promise<{ threads: Array<Thread> } | EntityResult> => {
  let threads: QueryArrayResult<Thread>;
  try {
    threads = await getThreadsByCategoryId(args.categoryId);
    if (threads.entities) {
      return {
        threads: threads.entities,
      };
    }
  }
}
  
```

W powyższym fragmencie kodu pobieramy wątki należące do określonej kategorii i zwracamy je.

```

return {
  messages: threads.messages
    ? threads.messages
    : ["Wystąpił błąd."],
};
  
```

Z kolei w tym fragmencie zwracamy komunikaty, jeśli nie udało się znaleźć żadnych wątków.

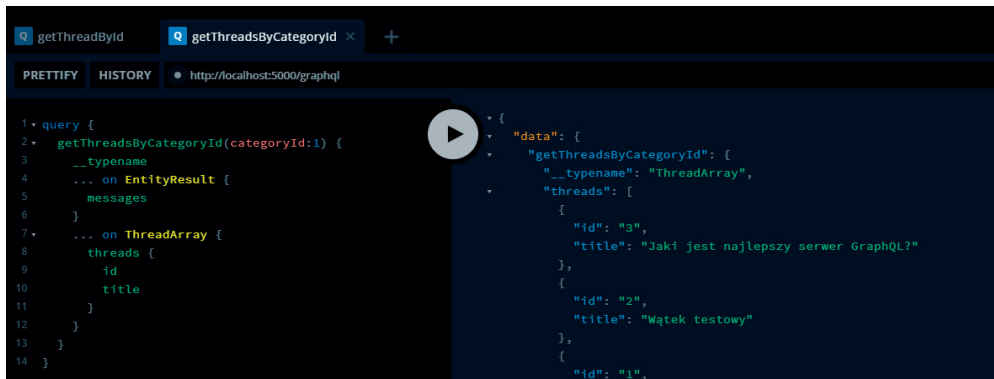
```
    } catch (ex) {
      throw ex;
    }
  },
```

12. Brakuje nam jeszcze jednego elementu. Kiedy po raz pierwszy zastosowaliśmy typ unii, musieliśmy przygotować odpowiednie zapytanie dla typu `EntityResult`. A zatem, teraz będziemy musieli przygotować analogiczne zapytanie dla typu `ThreadArrayResult`. Wpisz poniższy kod w pliku *resolvers.ts*, poniżej definicji `EntityResult`:

```
ThreadArrayResult: {
  __resolveType(obj: any, context: GqlContext, info: any) {
    if (obj.messages) {
      return "EntityResult";
    }
    return "ThreadArray";
  },
},
```

Ten kod działa praktycznie tak samo, jak ten przedstawiony wcześniej. Jeśli w parametrze `obj` istnieje właściwość `messages`, zwracamy typ `EntityResult`, a w przeciwnym razie zwracamy typ `ThreadArray`.

13. Jeśli teraz przygotujemy i wykonamy odpowiednie zapytanie, to uzyskamy wyniki przedstawione na rysunku 15.3 (zwróć uwagę na to, że w moich wynikach są widoczne dane testowe).



Rysunek 15.3. Test działania funkcji `getThreadsByCategoryId`

Zauważ, że dodałem do zapytania dodatkowe pole o nazwie `__typename`. To pole pozwoli nam określić, jaki jest typ zwracanego wyniku. Jak widać, w tym przypadku jest to typ `ThreadArray`.

No dobrze. Dysponujemy zatem działającym serwerem GraphQL-a obsługującym zapytania dotyczące wątków. Spróbuj teraz samodzielnie wypróbować i zintegrować z serwerem wszystkie pozostałe zapytania z rozdziału 14., pt. „Przygotowywanie Postgresa oraz warstwy

repozytorium przy wykorzystaniu TypeORM”, które nie są powiązane z uwierzytelnianiem. Jeśli będziesz miał z tym problemy, zajrzyj do kodów źródłowych dołączonych do książki. Ważne jest jednak, żebyś spróbował to zrobić *bez* szukania pomocy w przykładach, gdyż tylko w ten sposób uzyskasz pewność, że rozumiesz opisywane tu zagadnienia i rozwiązania.

## System punktacji wątków

Skoro zintegrowaliśmy z serwerem GraphQL-a istniejące wywołania, zajmijmy się wywołaniami, których wciąż brakuje w naszej aplikacji. Pamiętaj, że stworzyliśmy system punktacji do oceniania wątków (Thread) i odpowiedzi (ThreadItem). Zaimplementujmy zatem możliwości inkrementacji i dekrementacji liczby punktów. Jeśli nie pamiętasz już dokładnie struktury encji ThreadPoint oraz ThreadItemPoint, to zanim przejdziesz do dalszej lektury, poświęć chwilę, żeby je sobie przypomnieć. Zwróć także uwagę na nowe pole, points, w encjach Thread oraz ThreadItem; jego znaczenie wyjaśnię nieco później, kiedy zaczniemy pisać kod:

1. Zaczynaj od utworzenia w katalogu *src/repo* pliku *ThreadPointRepo.ts* i zapisania w nim poniższego fragmentu kodu (zakładam, że wiesz, jak dodać niezbędne instrukcje import):

```
export const updateThreadPoint = async (
  userId: string,
  threadId: string,
  increment: boolean
): Promise<string> => {
```

Zwróć uwagę na to, że na liście parametrów tej funkcji występuje parametr *increment* typu *boolean*. Będzie on określał, czy punkt zostanie dodany, czy odjęty.

*// Do zrobienia: najpierw sprawdź, czy użytkownik jest uwierzytelniony*

Później, kiedy zaimplementujemy mechanizmy uwierzytelniania, wrócimy do tego miejsca i zastąpimy ten komentarz odpowiednim kodem. Zwróć uwagę na to, że dodawanie komentarzy tego typu jest doskonałym sposobem utrwalania rzeczy, które pozostały nam do zrobienia. Co więcej, takie komentarze informują także naszych współpracowników o tym, co jeszcze pozostało do zaimplementowania.

```
let message = "Nie udało się inkrementować liczby punktów wątku.";
const thread = await Thread.findOne({
  where: { id: threadId },
  relations: ["user"],
});
if (thread!.user!.id === userId) {
  message = "Błąd: użytkownik nie może oceniać swojego wątku.";
  console.log("incThreadPoints", message);
  return message;
}
```

Zaczynamy od pobrania wątku o podanym identyfikatorze (*threadId*). Zwróć uwagę na to, że dodatkowo sprawdzamy, czy aktualny użytkownik nie jest tym, który utworzył oceniany wątek. Jeśli do tej pory miałeś w bazie danych tylko jednego użytkownika, to teraz nadszedł czas, żebyś utworzył następnego,

by użytkownik oceniający wątek nie był jednocześnie jego właścicielem. Nowego użytkownika możesz utworzyć korzystając ze skryptu CREATE w aplikacji pgAdmin, bądź też używając odpowiedniej trasy z rozdziału 14., pt. „Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM”.

```
const user = await User.findOne({ where: { id: userId } });
```

W tym wierszu kodu pobieramy użytkownika, jeszcze zanim będzie on nam faktycznie potrzebny. Niebawem przekonasz się, dlaczego robimy coś, co z pozoru mogłoby się wydawać nieefektywne.

```
const existingPoint = await ThreadPoint.findOne({
  where: {
    thread: { id: threadId },
    user: { id: userId },
  },
  relations: ["thread"],
});
```

W tym fragmencie kodu sprawdzamy, czy istnieje już encja punktu. Później użyjemy tego obiektu, by określić, w jaki sposób dodać lub odjąć punkt:

```
await getManager().transaction(async (transactionEntityManager) => {
```

Jak widać, w tym miejscu używamy trochę nowego kodu związanego z frameworkiem TypeORM. Wywołanie `getManager().transaction` rozpoczyna transakcję SQL. Transakcja to sposób wykonania wielu operacji SQL jako jednej operacji atomowej. Innymi słowy, albo wszystkie operacje wchodzące w skład transakcji zostaną wykonane prawidłowo, albo nie zostanie wykonana żadna z nich. Mówiąc jeszcze inaczej, wszystkie operacje wykonywane w wywołaniu `transaction` będą wchodzić w skład transakcji.

Wcześniej zwróciłem uwagę na to, że encję `User` utworzyliśmy jeszcze zanim była nam ona potrzebna. Takie rozwiązanie wynika z faktu, że należy unikać wykonywania zapytań pobierających dane wewnątrz transakcji. Nie jest to reguła, której pod żadnym pozorem nie można by złamać. Jednak ogólnie rzecz biorąc, wykonywanie zapytań pobierających dane wewnątrz transakcji, powoduje zmniejszenie wydajności.

```
if (existingPoint) {
  if (increment) {
    if (existingPoint.isDecrement) {
      await ThreadPoint.remove(existingPoint);
      thread!.points = Number(thread!.points) + 1;
      thread!.lastModifiedOn = new Date();
      await thread!.save();
    }
  } else {
    if (!existingPoint.isDecrement) {
      await ThreadPoint.remove(existingPoint);
      thread!.points = Number(thread!.points) - 1;
      thread!.lastModifiedOn = new Date();
      await thread!.save();
    }
  }
}
```

W tym fragmencie kodu sprawdzamy, czy istnieje już obiekt `ThreadPoint`; w tym celu sprawdzamy stałą `existingPoint` (pamiętaj, że obiekt `ThreadPoint` może reprezentować zarówno punkt dodatni, jak i ujemny, a jego znaczenie jest określane przez pole `isDecrement`). Kiedy już to ustalimy, określamy, czy punkt należy dodać, czy odjąć. Jeśli punkt mamy dodać i jeśli istniejący obiekt `ThreadPoint` reprezentuje punkt ujemny, to usuwamy encję z bazy danych i nie robimy nic więcej. Jeśli natomiast punkt mamy odjąć, a istniejący obiekt reprezentuje punkt dodatni, to także usuwamy encję z bazy i nie robimy nic więcej.

Kolejną sprawą, na którą powinieneś zwrócić uwagę, jest to, że aktualnie encja `Thread` zawiera nowe pole o nazwie `points`, które będziemy odpowiednio inkrementować lub dekrementować. To pole będzie nam ułatwiać generowanie interfejsu użytkownika, gdyż pozwoli nam wyświetlić bieżącą punktację wątku bez konieczności sumowania wszystkich punktów (`ThreadPoint`) dla danego wątku (`Thread`).

```

    } else {
      await ThreadPoint.create({
        thread,
        isDecrement: !increment,
        user,
      }).save();
      if (increment) {
        thread!.points = Number(thread!.points) + 1;
      } else {
        thread!.points = Number(thread!.points) - 1;
      }
      thread!.lastModifiedOn = new Date();
      await thread!.save();
    }
  }

```

Jeśli natomiast w ogóle nie zostały przyznane żadne punkty, to tworzymy odpowiednią encję, reprezentującą odpowiednio punkt dodatni lub ujemny.

```

    message = `Pomyślnie ${
      increment ? "dodano" : "odjęto"
    } punkt.`;
  });

  return message;
};

```

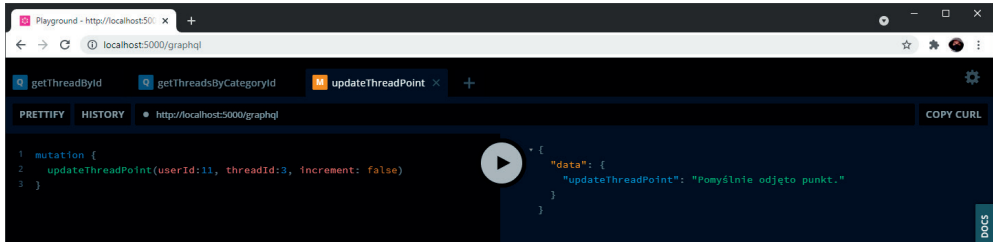
## 2. Teraz dodaj odpowiednią mutację do pliku *typeDefs.ts*:

```
updateThreadPoint(userId: ID!, threadId: ID!, increment: Boolean!): String!
```

## 3. Teraz musisz zmodyfikować stałą `resolvers` w pliku *resolvers.ts*, a konkretnie dodać do jej sekcji `Mutation` funkcję `updateThreadPoint`. Funkcja ta jest w zasadzie jedynie opakowaniem dla wywołania analogicznej funkcji warstwy repozytorium, która wykonuje wszystkie najważniejsze działania, dlatego nie będę jej tutaj przedstawiał. Przekonaj się, czy będziesz potrafił zaimplementować ją samodzielnie, bez zaglądania do kodów źródłowych.

Większość resolverów, których będziemy używać, będzie w zasadzie jedynie opakowaniami dla wywołań funkcji warstwy repozytorium. Takie rozwiązanie zapewni, że kod resolverów będzie odseparowany od kodu bazy danych oraz warstwy repozytorium. Dlatego, w przeważającej większości przypadków nie będę przedstawiał tu kodu resolverów, gdyż jest on bardzo prosty, krótki i dostępny w kodach źródłowych przykładów dołączonych do książki.

#### 4. Uruchom mutację tak, jak pokazałem na przykładzie z rysunku 15.4.



Rysunek 15.4. Wykonanie mutacji updateThreadPoint

Na rysunku 15.5 przedstawiłem wynik wykonania tej mutacji w bazie danych Postgres, której zawartość wyświetliłem w aplikacji pgAdmin.

Id	IsDecrement	userId	threadId	CreatedBy	CreatedOn	LastModifiedBy	LastModifiedOn
1	true	11	3	superforumuzk	2021-06-23 13:41:31.242068+02	superforumuzk	2021-06-23 13:41:31.242068+02

Rysunek 15.5. Wynik wykonania mutacji updateThreadPoint

Jak widać, odpowiedni rekord został poprawnie zapisany w bazie danych.

Przeanalizujmy teraz nieco dokładniej system punktacji, którym dysponujemy, oraz sposób jego działania. Nasz system punktacji *polubień* pozwala na rejestrowanie zarówno punktów pozytywnych (dodatnich), jak i negatywnych (ujemnych). Jednak dodatkowo system ten musi uniemożliwiać użytkownikom wielokrotne głosowanie. W tym celu musimy skojarzyć każdy punkt zarówno z użytkownikiem, który go przyznaje, jak i z wątkiem (Thread) lub odpowiedzią (ThreadPoint), którym dany punkt jest przyznawany. To właśnie dlatego utworzyliśmy encje ThreadPoint oraz ThreadItemPoint.

Na popularnym serwerze, na którym wielu użytkowników w tym samym czasie dodaje lub usuwa punkty, może to stanowić poważne obciążenie. Jednak znacznie większe obciążenie może powodować bezustanne sumowanie poszczególnych punktów dla każdego wątku i odpowiedzi. Takie rozwiązanie byłby niepraktyczne. Pierwszy z wymienionych problemów musimy zaakceptować i pogodzić się, że zastosowanie takiego systemu typu „jeden głos na

użytkownika” będzie nieuniknione. Natomiast jeśli chodzi o problem sumowania punktów, to możemy go próbować rozwiązać na kilka różnych sposobów.

Najbardziej wydajnym rozwiązaniem byłoby zastosowanie jakiegoś systemu pamięci podręcznej, działającego na jakieś dodatkowej usłudze, takiej jak Redis. Jednak przygotowanie takiej usługi nie jest trywialnym zadaniem i wykracza poza zakres niniejszej książki. Poza tym moglibyśmy argumentować, że nasza witryna dopiero startuje i zanim osiągnie wiekopomny sukces i zarobi miliony złotych, nie będzie mieć takiego obciążenia, by takie rozwiązania były konieczne. Dlatego, na początek, możemy spróbować czegoś prostszego.

Nasze rozwiązanie będzie polegać na dodaniu do encji `Thread` i `ThreadItem` pola do przechowywania sumarycznej liczby punktów oraz odpowiednim modyfikowaniu wartości tego pola podczas przyznawania lub usuwania punktów danemu wątkowi lub odpowiedzi. Nie jest to co prawda najlepsze rozwiązanie, lecz na razie wystarczy. Później będziemy mieć więcej czasu, żeby przygotować coś bardziej wyszukanego, takiego jak system pamięci podręcznej lub mechanizm jakiegoś innego rodzaju.

Kod klasy `ThreadItemPoint` będzie w zasadzie identyczny. Spróbuj zatem sprawdzić, czy będziesz potrafił samodzielnie wprowadzić niezbędne zmiany w kodzie pliku *ThreadItemPoint-Repo.ts*. Jak zwykle, jeśli będziesz mieć problemy, możesz zajrzeć do kodów źródłowych dołączonych do książki.

W tym podrozdziale zaczęliśmy integrować funkcje warstwy repozytorium z warstwą GraphQL-a. Uzupełniliśmy także systemu punktacji wątków i odpowiedzi. W następnym podrozdziale będziemy kontynuować tworzenie API GraphQL-a, a konkretnie zajmiemy się integracją wywołań związanych z uwierzytelnianiem użytkowników.

## Integracja mechanizmu uwierzytelniania z resolverami GraphQL-a

Integracja wywołań związanych z uwierzytelnianiem z GraphQL-em nie różni się znacząco od dodawania do GraphQL-a dowolnych innych możliwości funkcjonalnych. W tym podrozdziale pokażę, jak to zrobić.

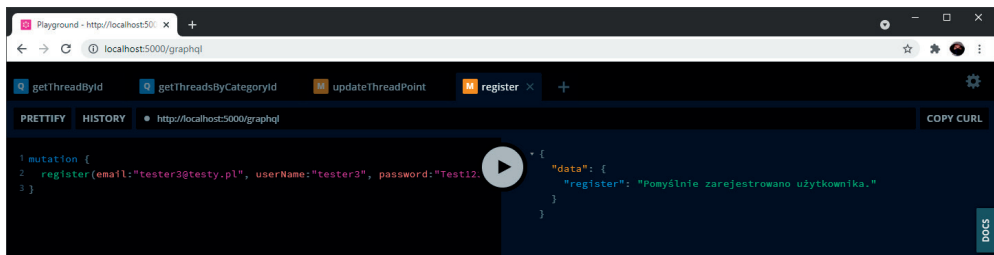
A zatem, zajmiemy się teraz integrowaniem wywołań związanych z uwierzytelnianiem, zaczynając od funkcji `register`:

1. Pamiętasz zapewne, że funkcję `register` zaimplementowaliśmy już wcześniej, w rozdziale 14., pt. „Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu `TypeORM`”. Zaktualizujmy zatem definicje typów podane w stałej `typeDefs` w pliku *typeDefs.ts* oraz resolvery w pliku *resolvers.ts*. Zacznij od dodania funkcji `register` z kodu źródłowego do sekcji `Mutation`, w pliku definicji typów.
2. Teraz w pliku resolverów (*resolvers.ts*), w sekcji `Mutation`, dodaj kod skopiowany z przykładów do książki.



Także ta funkcja stanowi jedynie opakowanie dla wywołania funkcji warstwy repozytorium, więc nie ma potrzeby, bym ją dokładniej opisywał; zwróć jedynie uwagę na to, że funkcja ta nie zwraca obiektu `User`, a jedynie komunikat tekstowy. Przyjąłem takie rozwiązanie, by zminimalizować prawdopodobieństwo zwrócenia jakichś niepotrzebnych informacji. Zanim przetestujesz działanie tej funkcji, musisz jeszcze włączyć w aplikacji „placu zabaw” GraphQL-a możliwość zapisywania ciasteczek. Jest ona niezbędna do zapisywania stanu sesji, a to właśnie on będzie pozwalał sprawdzać w kodzie naszych funkcji, czy użytkownik jest zalogowany, czy nie.

Kliknij ikonę koła zębatego, umieszczoną w prawym górnym rogu aplikacji. W wyświetlonych ustawieniach, w polu `request.credentials`, zapisz wartość `include`, a następnie kliknij przycisk `SAVE SETTINGS`. Teraz możesz już wykonać mutację, jak pokazałem na rysunku 15.6.



Rysunek 15.6. Rejestrowanie użytkownika

3. Teraz zajmiemy się funkcją `login`. Dodaj jej kod źródłowy do sekcji `Mutation` w pliku `typeDefs.ts`.
4. Następnie dodaj kod resolvera `login` z przykładów dołączonych do książki. Nasza funkcja `login` warstwy repozytorium sprawdza, czy użytkownik istnieje, a jeśli tak, upewnia się, że hasła są zgodne. Analogiczna funkcja warstwy GraphQL-a pobiera wartość `user.id` i jeśli logowanie zostało zakończone pomyślnie, zapisuje ją w obiekcie sesji, jako `ctx.req.session.userId`. Zwróć uwagę na to, że po pomyślnym zalogowaniu nasz resolver nie zwraca obiektu `user`. Nieco później stworzymy odrębną funkcję zwracającą informacje o użytkowniku — obiekt `User`.
5. Kolejnym krokiem będzie przygotowanie funkcji `logout`. W pierwszej kolejności dodaj odpowiedni kod do sekcji `Mutation` pliku definicji typów (`typeDefs.ts`).
6. Następnie zaktualizuj sekcję `Mutation` w pliku `resolvers.ts`, dodając do niej kod resolvera `logout` skopiowany z przykładów dołączonych do książki. Zwróć uwagę na to, że niezależnie od wyniku zwróconego przez funkcję `logout` warstwy repozytorium usuwamy sesję, wywołując w tym celu funkcję `ctx.req.session?.destroy`, co sprawia, że wartość wyrażenia `ctx.req.session?.userId` staje się niezdefiniowana.

7. Teraz musimy dodać do pliku definicji typów jeszcze jedno nowe wywołanie oraz jeden nowy typ. Zacznij od dodania funkcji `me` do sekcji `Query`; odpowiedni kod możesz skopiować z przykładów dołączonych do książki. Następnie, poniżej typu `User`, dodaj poniższą definicję nowego typu `unii`:

```
union UserResult = User | EntityResult
```

Do czego nam będzie potrzebny ten typ? Otóż w funkcjach `register` i `login` zrezygnowaliśmy ze zwracania obiektu `User`, gdyż szczegółowe informacje o użytkowniku mogą, lecz nie muszą być potrzebne po wykonaniu tych wywołań, a nam zależy na tym, by niepotrzebnie nie udostępniać tych danych. Jednak są sytuacje, kiedy po prawidłowym zalogowaniu użytkownika będziemy chcieli mieć dostęp do tych danych. Przykładem może być wyświetlenie ekranu profilu użytkownika. Z myślą o właśnie takich sytuacjach zaimplementujemy funkcję `me`.

8. Teraz dodaj przedstawiony poniżej kod funkcji `me` do pliku `UserRepo.ts`:

```
export const me = async (id: string): Promise<UserResult> => {
  const user = await User.findOne({
    where: { id },
    relations: [ "threads", "threads.threadItems" ],
  });
```

W pierwszej kolejności zauważ, że zwracany obiekt `user` będzie zawierał wszystkie wątki (`Thread`) oraz odpowiedzi (`ThreadItem`) należące do danego użytkownika. Tych danych użyjemy później na ekranie profilu użytkownika.

```
    if (!user) {
      return {
        messages: ["Nie znaleziono użytkownika."],
      };
    }

    if (!user.confirmed) {
      return {
        messages: ["Użytkownik jeszcze nie potwierdził swojego adresu ↵e-mail."],
      };
    }

    user.password = "";
    return {
      user: user,
    };
  };
};
```

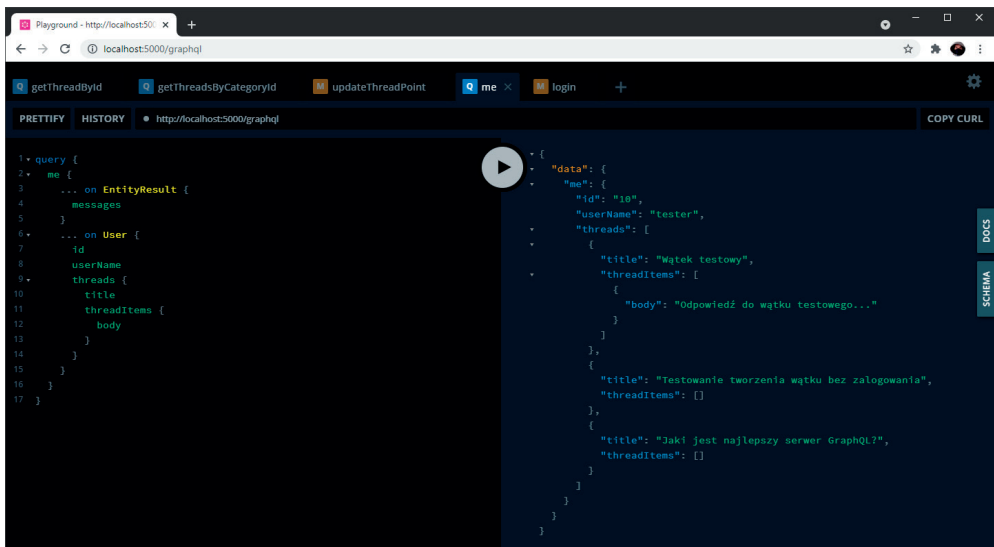
Pozostała część funkcji w dużym stopniu przypomina kod funkcji `login`.

9. Teraz przygotujemy resolwery na potrzeby ekranu profilu użytkownika oraz funkcji `me`. Na początku stałej `resolvers` dodaj resolwer `UserResult`, który znajdziesz w przykładach dołączonych do książki. Jest on dokładnie taki sam, jak resolwery innych typów `unii` — nie w nim niczego szczególnego, co wymagałoby dokładniejszego opisu.

10. Następnie w sekcji Query umieść kod funkcji `me`, także skopiowany z przykładów dołączonych do książki.

Zwróć uwagę na to, że ten resolver nie ma żadnych parametrów, gdyż identyfikator użytkownika (`userId`) jest pobierany z sesji. W wierszu 193. sprawdzamy, czy identyfikator użytkownika, `userId`, jest w sesji. Jeśli go nie ma, funkcja zostaje zakończona. Jeśli jednak identyfikator będzie dostępny, wywołujemy funkcję `me` warstwy repozytorium, aby pobrać dane aktualnie zalogowanego użytkownika. Pozostała część kodu jest niemal identyczna, jak w innych funkcjach zwracających obiekty encji.

11. Spróbujmy teraz wykonać resolver `me`. Upewnij się, że wcześniej zalogujesz się i postąpisz zgodnie z instrukcjami opisanymi w punkcie 3. podrozdziału dotyczącego aplikacji „placu zabaw” GraphQL-a. Jeśli wykonasz resolver `me` w sposób przedstawiony na rysunku 15.7, powinieneś uzyskać widoczne na nim dane.



Rysunek 15.7. Wywołanie resolvera `me`

Jak widzisz, w tym przykładzie użyliśmy wpisanych fragmentów i pobraliśmy informacje dotyczące obu powiązanych encji — wątków oraz odpowiedzi.

W tym rozdziale powiązaliśmy wywołania związane z uwierzytelnianiem zaimplementowane w warstwie repozytorium z GraphQL-em i przetestowaliśmy nowe możliwości serwerowej części naszej aplikacji. W następnym podrozdziale dokończymy naszą aplikację, łącząc niemal gotową część serwerową z aplikacją kliencką.

# Przygotowanie hooków Reacta do korzystania z serwera Apollo GraphQL

W tym podrozdziale dokończymy prace nad naszą aplikacją: połączymy klienta Reacta z częścią serwerową korzystającą z GraphQL-a. Udało się nam już zrobić naprawdę bardzo wiele i prace nad aplikacją są już niemal zakończone.

Aby połączyć obie części aplikacji, musimy dodać do serwera Expressa obsługę CORS. CORS to skrót od angielskich słów *Cross-Origin Resource Sharing*. Oznacza to, że nasz serwer zostanie skonfigurowany w taki sposób, że będzie mógł obsługiwać żądania przesyłane przez klienty z innej domeny niż ta, do której należy serwer.

W większości konfiguracji serwerowych, i to nawet takich o skromnej złożoności, serwer udostępniający aplikację kliencką oraz serwer udostępniający API nie należą do tej samej domeny. Ogólnie rzecz biorąc, zapewne będziemy używać jakiegoś serwera pośredniczącego, na przykład NGINX, którego przeznaczeniem będzie odbieranie żądań przesyłanych przez przeglądarki. Ten serwer pośredniczący będzie odpowiednio przekierowywał te żądania. Więcej informacji na temat działania odwrotnych serwerów pośredniczących podam w rozdziale 17., pt. „Wdrażanie w chmurze AWS”.

Serwer pośredniczący (ang. *proxy*) jest swoistym zastępcą jakiejś usługi lub grupy usług. W razie jego stosowania, kiedy klient przesyła żądanie, w pierwszej kolejności trafia ono właśnie do serwera pośredniczącego, a nie bezpośrednio do usługi. Serwer pośredniczący określa następnie, czy żądanie powinno zostać przekierowane do usługi, czy nie. Dlatego serwery pośredniczące zapewniają firmom znacznie lepszą kontrolę nad dostępem do ich usług.

Włączenie obsługi CORS jest konieczne także dlatego, że aplikacja Reacta działa na swoim własnym testowym serwerze. W naszym przypadku serwer ten działa na porcie 3000, natomiast serwer GraphQL-a działa na porcie 5000. Choć oba te serwery działają na hoście `localhost`, to jednak zastosowanie innych numerów portów oznacza, że działają one w dwóch różnych domenach. Poniżej wymienię czynności, które powinieneś wykonać, by włączyć obsługę CORS.

1. W pierwszej kolejności musisz zaktualizować plik `.env` i podać w nim ścieżkę dostępu do roboczego serwera Reacta:

```
CLIENT_URL=http://localhost:3000
```

2. Następnie otwórz plik `index.ts` i dodaj poniższy fragment kodu bezpośrednio poniżej wiersza `const app = express();`:

```
app.use(
  cors({
    credentials: true,
    origin: process.env.CLIENT_URL,
  })
);
```

Przypisanie wartości `true` właściwości `credentials` włącza przesyłanie nagłówka `Access-Control-Allow-Credentials`. Pozwala on klientom JavaScript odbierać odpowiedzi z serwerów, po podaniu odpowiednich danych uwierzytelniających.

3. Oprócz tego zmodyfikuj konfigurację serwera Apollo w taki sposób, by jego wbudowane mechanizmy CORS zostały wyłączone. W tym celu zmodyfikuj wiersz kodu bezpośrednio przed wywołaniem funkcji `listen`:

```
apolloServer.applyMiddleware({ app, cors: false });
```

Serwer Apollo jest wyposażony we własną obsługę CORS, która domyślnie jest włączona; dlatego teraz musimy ją wyłączyć.

W ten sposób zainstalowaliśmy na naszym serwerze CORS. Teraz w odrębnym oknie VSCode otwórz projekt klienckiej aplikacji Reacta i zainstaluj GraphQL, co pozwoli nam rozpocząć integrowanie tej aplikacji z serwerem GraphQL-a.

1. Po otworzeniu katalogu *super-forum-client* w nowym oknie programu VSCode, najpierw spróbuj uruchomić projekt, by upewnić się, czy działa. Jeśli jeszcze tego nie zrobiłeś, usuń katalog *node\_modules* oraz plik *package-lock.json*, po czym wykonaj polecenie `npm install`.
2. Teraz zainstaluj klienta Apollo GraphQL. Otwórz panel terminala, przejdź w nim do głównego katalogu *super-forum-client* i wykonaj poniższe polecenie:

```
npm install @apollo/client graphql
```

3. Kolejnym krokiem będzie skonfigurowanie klienta GraphQL. Otwórz plik *index.ts* i dodaj przedstawiony poniżej fragment kodu przed wywołaniem `ReactDOM.render`:

```
const client = new ApolloClient({
  uri: "http://localhost:5000/graphql",
  credentials: "include",
  cache: new InMemoryCache({
    resultCaching: false,
  }),
});
```

Jak zwykle, dodaj niezbędne instrukcje importu, ale ich postać powinna być już oczywista. W tym fragmencie kodu określamy adres URL serwera, dołączamy potrzebne dane uwierzytelniające i określamy obiekt `cache`. Zwróć uwagę na to, że to ostatnie ustawienie oznacza, że serwer Apollo będzie przechowywał wszystkie wyniki zapytań w pamięci podręcznej.

4. Następnie zmodyfikuj wywołanie `ReactDOM.render` i dodaj do umieszczonego w nim kodu JSX komponent `ApolloProvider`:

```
ReactDOM.render(
  <Provider store={configureStore()}>
    <BrowserRouter>
      <ApolloProvider client={client}>
        <ErrorBoundary>{[<App key="App" />]}</ErrorBoundary>
      </ApolloProvider>
    </BrowserRouter>
  </Provider>
```

```

    </Provider>,
    document.getElementById("root")
  );

```

5. Teraz sprawdzimy, czy aplikacja działa; w tym celu, na przykład, pobierzemy kategorię wątków (`ThreadCategory`). Aby to zrobić, otwórz plik `src/components/areas/LeftMenu.tsx` i zmodyfikuj go jak pokazałem na poniższym przykładzie:

```

import React, { useEffect, useState } from "react";
import { useWindowDimensions } from "../../hooks/useWindowDimensions";
import "./LeftMenu.css";
import { gql, useQuery } from "@apollo/client";

```

Jak widać, zaimportowaliśmy dwa elementy z klienta Apollo. `gql` pozwala korzystać z wyróżniania i formatowania składni zapytań GraphQL-a. Z kolei `useQuery` jest pierwszym z *hooków* Reacta związanych z GraphQL-em, z których będziemy korzystać w naszej aplikacji. *Hook* `useQuery` pozwala na wykonywanie zapytań GraphQL-a, nie daje natomiast możliwości wykonywania mutacji. Co więcej, ta funkcja jest wykonywana natychmiast; w dalszej części rozdziału przedstawię inny *hook*, który pozwala na leniwe pobieranie danych.

```

const GetAllCategories = gql`
  query getAllCategories {
    getAllCategories {
      id
      name
    }
  }
`;

```

Oto i całe zapytanie. Nie ma tu za bardzo co wyjaśniać, po prostu pobieramy właściwości `id` i `name`.

```

const LeftMenu = () => {
  const { loading, error, data } = useQuery(GetAllCategories);

```

Funkcja `useQuery` zwraca właściwości `loading`, `error` oraz `data`. Każde wywołanie *hooka* Apollo GraphQL zwraca zestaw określonych właściwości. W kolejnym fragmencie kodu pokażę, w jaki sposób można używać tych właściwości:

```

const { width } = useWindowDimensions();
const [categories, setCategories] = useState<JSX.Element>(
  <div>Menu z lewej</div>
);

useEffect(() => {
  if (loading) {
    setCategories(<span>Trwa wczytywanie ...</span>);

```

W powyższym fragmencie kodu sprawdzamy, czy dane wciąż są pobierane; używamy do tego celu właściwości `loading`, a jeśli okaże się, że dane faktycznie są pobierane, wyświetlamy odpowiedni tekst zastępczy.

```

  } else if (error) {
    setCategories(<span>Podczas wczytywania kategorii wystąpił
      ↪ błąd...</span>);

```

W tej sekcji wyświetlamy informacje o błędach, które mogły się pojawić podczas wykonywania zapytania.

```

    } else {
      if (data && data.getAllCategories) {
        const cats = data.getAllCategories.map((cat: any) => {
          return <li key={cat.id}>
            <Link to={`/${categorythreads}/${cat.id}`}>{cat.name}</Link>
          </li>;
        });

        setCategories(<ul className="category">{cats}</ul>);
      }
    }
  }, [data]);

  if (width <= 768) {
    return null;
  }
  return <div className="leftmenu">{categories}</div>;
};

export default LeftMenu;

```

I w końcu, jeśli wszystko pójdzie dobrze, pobieramy dane i wyświetlamy je w formie listy wypunktowanej, której każdy element reprezentuje jeden obiekt `ThreadCategory`. Zwróć uwagę na to, że każdy element `li` zawiera unikalny identyfikator. To bardzo ważne, by zawsze używać kluczy, kiedy generujemy tablicę podobnych elementów, gdyż w ten sposób minimalizujemy liczbę niepotrzebnych operacji renderowania. Co więcej, każdy element tej listy jest odnośnikiem prowadzącym na stronę ze wszystkimi wątkami należącymi do wybranej kategorii.

```

    }
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [data]);

  if (width <= 768) {
    return null;
  }
  return <div className="leftmenu">{categories}</div>;
};

export default LeftMenu;

```

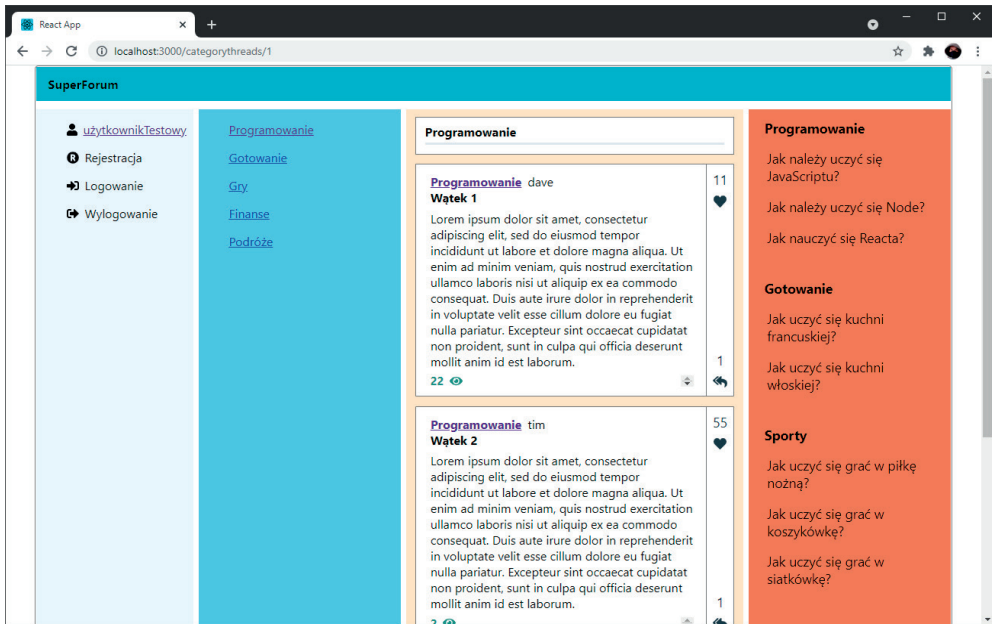
- Obecnie, jeśli uruchomisz aplikację w trybie dla komputerów stacjonarnych, powinieneś zobaczyć ekran kategorii taki jak ten pokazany na rysunku 15.8. Zwróć uwagę na to, że aby go uzyskać, kliknąłem na odnośniku na liście kategorii; oczywiście przedstawione dane poszczególnych wątków wciąż są na pobierane z usługi `dataService`, w której są one podane na stałe.

Fantastycznie! Udało się nam podłączyć aplikację kliencką do serwera GraphQL-a!

## Ekran główny — komponent Main

Gratuluję — przeszedłeś naprawdę długą drogę. Teraz zajmiemy się komponentem `Main`, tak, by prezentował prawdziwe dane pobierane z serwera GraphQL-a. Oto czynności, które należy w tym celu wykonać:

- Przejdź do projektu *super-forum-server* i otwórz plik *typeDefs.ts*; następnie, poniżej funkcji `getThreadsByCategory`, dodaj do schematu deklarację funkcji `getThreadsLatest` (skopiuj ją z kodów źródłowych dołączonych do książki).



Rysunek 15.8. Lista kategorii wątków w komponencie LeftMenu

Kolejnym krokiem będzie utworzenie odpowiedniego resolvera, `getThreadsLatest`, który będzie zwracał najnowsze wątki w przypadku, gdy nie zostanie określona żadna konkretna kategoria wątków. Jeśli kategoria została określona, to skorzystamy z istniejącego już resolvera `getThreadsByCategoryId`.

## 2. Dodaj poniższą funkcję w pliku *ThreadRepo.ts*:

```
export const getThreadsLatest = async (): Promise<QueryArrayResult<Thread>> => {
  const threads = await Thread.createQueryBuilder("thread")
    .leftJoinAndSelect("thread.category", "category")
    .leftJoinAndSelect("thread.user", "user")
    .leftJoinAndSelect("thread.threadItems", "threadItems")
    .orderBy("thread.createdOn", "DESC")
    .take(10)
    .getMany();
}
```

Ten fragment kodu przedstawia zapytanie, które dołącza do wątków kategorie (`ThreadCategory`) oraz odpowiedzi (`ThreadItems`), używając do tego celu funkcji `leftJoinAndSelect`; ponadto sortuje wyniki na podstawie wartości pola `createdOn` (przy użyciu funkcji `orderBy`), po czym pobiera tylko 10 pierwszych rekordów (dzięki użyciu funkcji `take`).

```
if (!threads || threads.length === 0) {
  return {
    messages: ["Nie znaleziono żadnych wątków."],
  };
}
console.log(threads);
return {
}
```



```

    entities: threads,
  };
};

```

Pozostała część kodu nie wymaga dokładniejszych wyjaśnień, gdyż jest bardzo podobna do kodu funkcji `getThreadsByCategoryId`.

Przy okazji zmodyfikujmy funkcję `getThreadsByCategoryId` tak, by zwracała także informacje o odpowiedziach należących do wątków:

```

export const getThreadsByCategoryId = async (
  categoryId: string
): Promise<QueryArrayResult<Thread>> => {
  const threads = await Thread.createQueryBuilder("thread")
    .where(`thread."categoryId" = :categoryId`, { categoryId })
    .leftJoinAndSelect("thread.category", "category")
    .leftJoinAndSelect("thread.threadItems", "threadItems")
    .orderBy("thread.createdOn", "DESC")
    .getMany();

  if (!threads || threads.length === 0) {
    return {
      messages: ["W wybranej kategorii nie znaleziono żadnych wątków."],
    };
  }
  console.log(threads);
  return {
    entities: threads,
  };
};

```

Ten kod jest bardzo podobny do poprzedniego, z tym, że dodaliśmy do niego jedno dodatkowe wywołanie funkcji `leftJoinAndSelect`.

3. Teraz otwórz plik resolverów (*resolvers.ts*) i na końcu sekcji Query wklej kod funkcji `getThreadLatest` (skopiowany z kodów źródłowych dołączonych do książki). Jest to funkcja opakowująca, niemal identyczna jak resolver `getThreadsByCategoryId`, lecz odwołuje się oczywiście do funkcji `getThreadsLatest`.
4. Teraz możemy przystąpić do aktualizacji komponentu Main i zastosowania w nim resolverów GraphQL-a zamiast pobierania fikcyjnych danych z usługi `dataService`. Otwórz plik *Main.tsx* i zaktualizuj go zgodnie z poniższymi przykładami.

Stała `GetThreadsByCategoryId` jest pierwszym z zapytań stosowanych w tym komponencie. Jak widać, używa ono wpisanych fragmentów, by pobrać dane dotyczące wątków:

```

const GetThreadsByCategoryId = gql`
  query getThreadsByCategoryId($categoryId: ID!) {
    getThreadsByCategoryId(categoryId: $categoryId) {
      ... on EntityResult {
        messages
      }

      ... on ThreadArray {

```

```

        threads {
          id
          title
          body
          views
          threadItems {
            id
          }
          category {
            id
            name
          }
        }
      }
    }
  }
};

```

Zapytanie `GetThreadsLatest` jest niemal identyczne jak `GetThreadsByCategoryId`:

```

const GetThreadsLatest = gql`
  query getThreadsLatest {
    getThreadsLatest {
      ... on EntityResult {
        messages
      }

      ... on ThreadArray {
        threads {
          id
          title
          body
          views
          threadItems {
            id
          }
          category {
            id
            name
          }
        }
      }
    }
  }
`;

```

A teraz, na początku definicji komponentu `Main` zastosujemy nowy *hook* o nazwie `useLazyQuery`:

```

const Main = () => {
  const [
    execGetThreadsByCat,
    {
      // error: threadsByCatErr,
      // called: threadsByCatCalled,
      data: threadsByCatData,
    }
  ] = useLazyQuery(getThreadsByCat);
};

```

```

    },
  ] = useLazyQuery(GetThreadsByCategoryId);
  const [
    execGetThreadsLatest,
    {
      // error: threadsLatestErr,
      // called: threadsLatestCalled,
      data: threadsLatestData,
    },
  ] = useLazyQuery(GetThreadsLatest);

```

Te dwa dodane *hooki* będą wykonywać nasze zapytania. Zwróć uwagę na to, że zapytania będą wykonywane w sposób „leniwy”; oznacza to, że nie zostaną uruchomione natychmiast, jak było w przypadku użycia *hooka* `useQuery`, lecz dopiero w momencie wywołania funkcji `execGetThreadsByCat` lub `execGetThreadsLatest`. Zwrócone dane będą zapisane we właściwości `data`. Oprócz tego umieściłem w komentarzach dwie ze zwracanych właściwości, gdyż nie będziemy ich używać. Pamiętaj jednak, że są one dostępne i w razie wystąpienia błędów możesz z nich skorzystać. Właściwość `error` zawiera informacje o błędach, a właściwość `called` informację o tym, czy *hook* został już wywołany, czy nie.

```

const { categoryId } = useParams();
const [category, setCategory] = useState<Category | undefined>();
const [threadCards, setThreadCards] = useState<Array<JSX.Element> | null>(
  null
);

```

Używane wcześniej obiekty stanu pozostają bez zmian.

```

useEffect(() => {
  if (categoryId && categoryId > 0) {
    execGetThreadsByCat({
      variables: {
        categoryId,
      },
    });
  } else {
    execGetThreadsLatest();
  }
  // eslint-disable-next-line react-hooks/exhaustive-deps
}, [categoryId]);

```

To wywołanie `useEffect` zostało zmodyfikowane w taki sposób, że w zależności od potrzeby wywołuje jedną z dwóch funkcji: `execGetThreadByCat` lub `execGetThreadsLatest`. Jeśli został podany parametr `categoryId`, zostanie wywołana funkcja `execGetThreadByCat`, z kolei, jeżeli parametr ten nie został podany, to zostanie wywołana druga z funkcji.

```

useEffect(() => {
  console.log("main threadsByCatData", threadsByCatData);
  if (
    threadsByCatData &&
    threadsByCatData.getThreadsByCategoryId &&
    threadsByCatData.getThreadsByCategoryId.threads
  ) {

```

```

    const threads = threadsByCatData.getThreadsByCategoryId.threads;
    const cards = threads.map((th: any) => {
      return <ThreadCard key={`thread-${th.id}`} thread={th} />;
    });
    setCategory(threads[0].category);
    setThreadCards(cards);
  }
}, [threadsByCatData]);

```

W tym wywołaniu `useEffect`, zmiana zawartości `threadsByCatData` powoduje zaktualizowanie danych stanu `category` i `cards`, i zapisanie w nich danych zwróconych przez zapytanie `getThreadsByCategoryId`.

```

useEffect(() => {
  if (
    threadsLatestData &&
    threadsLatestData.getThreadsLatest &&
    threadsLatestData.getThreadsLatest.threads
  ) {
    const threads = threadsLatestData.getThreadsLatest.threads;
    const cards = threads.map((th: any) => {
      return <ThreadCard key={`thread-${th.id}`} thread={th} />;
    });

    setCategory(new Category("0", "Najnowsze"));

    setThreadCards(cards);
  }
}, [threadsLatestData]);

```

Z kolei w tym wywołaniu `useEffect` zmiany zawartości `threadsLatestData` powodują zmianę danych stanów `category` i `threadCards`, i zapisanie w nich informacji zwróconych przez zapytanie `getThreadsLatest`. Zwróć uwagę na to, że jeśli nie zostanie podany parametr `categoryId`, jako nazwę kategorii wyświetlamy jedynie ogólny nagłówek "Najnowsze".

```

    return (
      <main className="content">
        <MainHeader category={category} />
        <div>{threadCards}</div>
      </main>
    );
  };
};

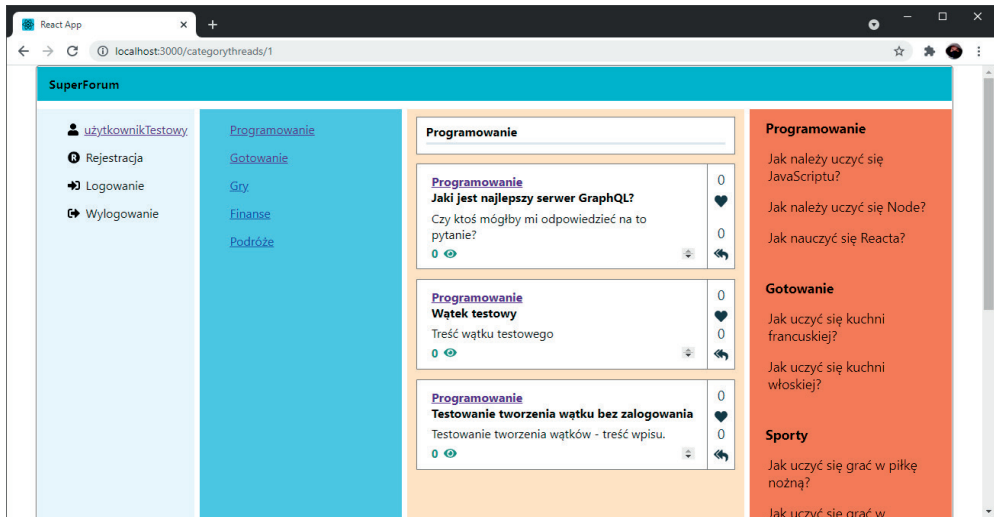
export default Main;

```

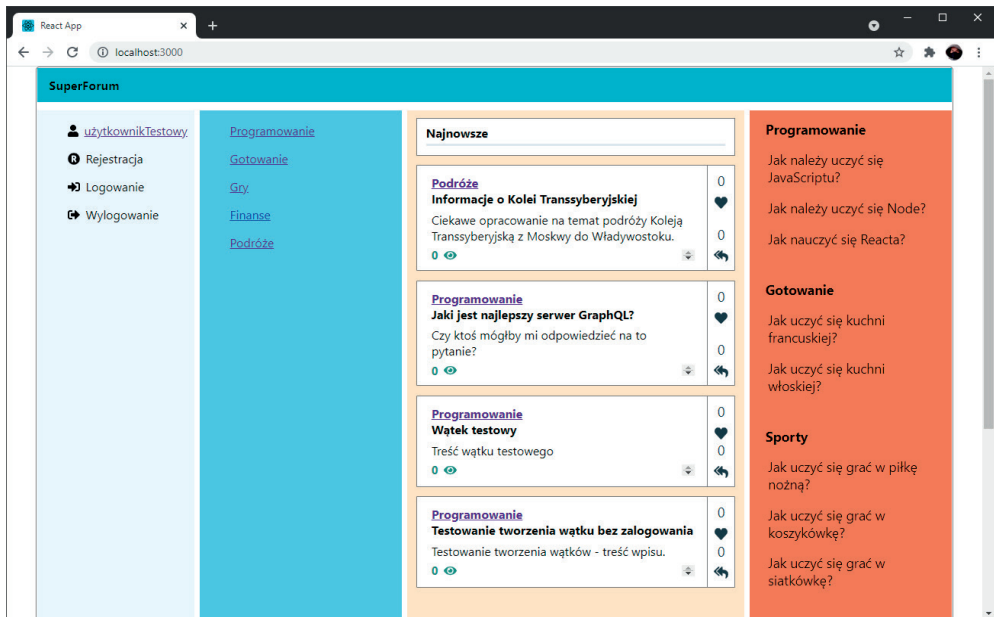
Końcowa część kodu komponentu pozostaje bez zmian.

5. Teraz, jeśli zostanie przekazany identyfikator kategorii (`categoryId`), to aplikacja będzie wyglądać tak, jak na rysunku 15.9.

Jeśli natomiast identyfikator kategorii nie zostanie określony, to aplikacja będzie wyglądać jak na rysunku 15.10.

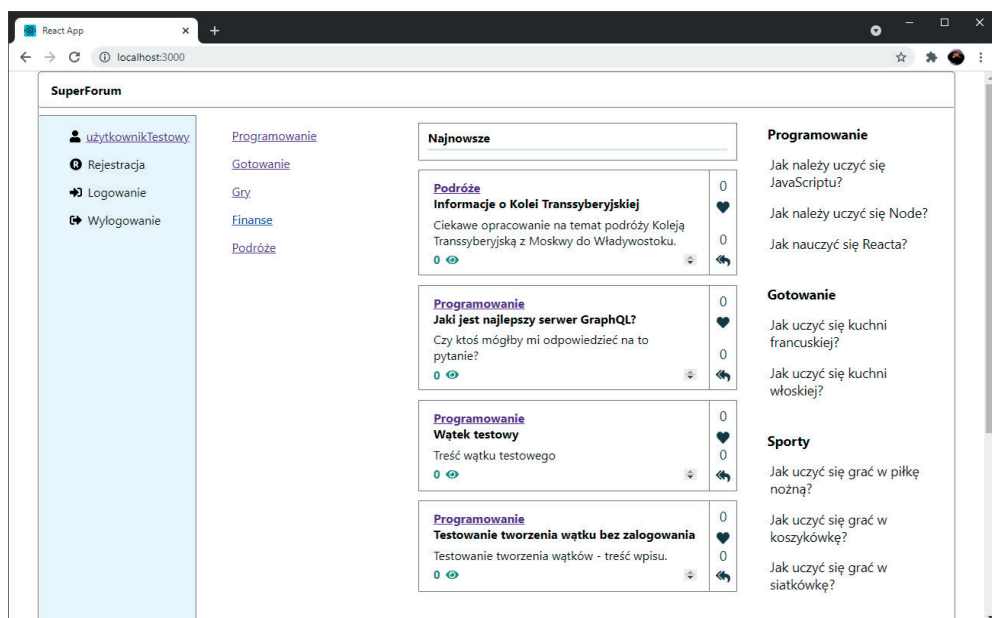


Rysunek 15.9. Postać aplikacji w razie przekazania identyfikatora kategorii



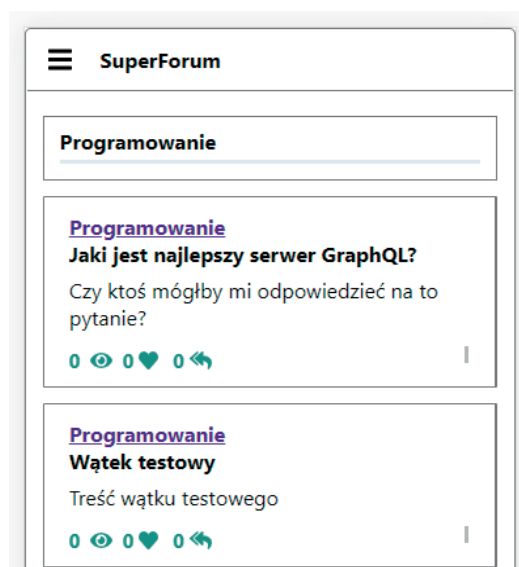
Rysunek 15.10. Postać aplikacji w przypadku braku identyfikatora kategorii

Doskonale, teraz dysponujemy już faktycznymi danymi, które są wyświetlane na ekranach aplikacji. Zanim przejdziemy do kolejnych zagadnień, dopracujemy jeszcze nieco style i pozbędziemy się niepotrzebnych kolorów tła niektórych elementów. Drobne zmiany trzeba będzie wprowadzić w plikach *Nav.css* oraz *Home.css*. Zobaczmy, jak aplikacja wygląda po wprowadzeniu tych modyfikacji (patrz rysunek 15.11).



Rysunek 15.11. Główny ekran aplikacji po aktualizacji stylów

Dobrze, teraz aplikacja wygląda lepiej. Warto zwrócić uwagę na jeszcze jeden szczegół dotyczący mobilnej postaci aplikacji: okazuje się, że użytkownik nie ma możliwości wybrania innej kategorii (patrz rysunek 15.12).



Rysunek 15.12. Główny ekran aplikacji na urządzeniu mobilnym

Aby rozwiązać ten problem, dodamy rozwijaną listę, która pozwoli użytkownikom wybierać kategorię. Ta lista ma być wyświetlana wyłącznie na urządzeniach mobilnych. Spróbuj ją dodać samodzielnie, zanim przejdziesz do dalszej lektury. Wskazówka: użyj komponentu `React-DropDown`, aby stworzyć listę, i zastąp nim nagłówek kategorii. Na przykład na rysunku 15.12. widać, że wybraną kategorią jest *Programowanie* — to właśnie ta nazwa jest wyświetlona w komponencie `MainHeader`. Zastąp zatem tę nazwę rozwijaną listą, lecz tylko na urządzeniach mobilnych. Zwróć uwagę na to, że używamy już tego komponentu w trasie `ThreadCategory`, więc powinieneś stosowany kod na komponent, którego będziesz mógł używać w obu miejscach.

Jeśli już spróbowałeś samemu wprowadzić opisaną wcześniej zmianę, przygotujmy ją wspólnie, byś mógł porównać oba rozwiązania. Muszę się przyznać, że trochę przy tej okazji skłamałem. Ta modyfikacja jest dość złożona, gdyż składa się z dwóch głównych zadań. Przede wszystkim chcielibyśmy dodać nowy reduktor dla kategorii wątków (`ThreadCategory`), gdyż wiemy, że lista kategorii jest używana w co najmniej dwóch odrębnych komponentach. Oprócz tego musimy przekształcić kod wyświetlający listę w komponencie `ThreadCategory` na nowy komponent, by można go było łatwo używać w kilku różnych miejscach. To drugie zadanie jest stosunkowo wymagające, gdyż nowy komponent rozwijanej listy musi być na tyle złożony, by można było przekazywać do niego właściwości *props* z komponentów nadrzędnych, jak również przekazywać wybraną kategorię poza komponent, kiedy użytkownik ją zmienia.

1. Zaczniemy od utworzenia nowego reduktora. W katalogu *store* utwórz nowy katalog, o nazwie *categories*. W tym katalogu utwórz plik *Reducer.ts* i zapisz w nim kod skopiowany z przykładów dołączonych do książki. Ten plik jest bardzo podobny do reduktora *user*, przy czym tym razem zwracanymi danymi są obiekty *Category*.
2. W kolejnym kroku musimy dodać ten nowy reduktor do głównego reduktora (*rootReducer*) w pliku *AppState.ts*:

```
export const rootReducer = combineReducers({
  user: UserProfileReducer,
  categories: ThreadCategoriesReducer,
});
```

Ten nowy reduktor nazwiemy *categories*.

3. Teraz musimy zaktualizować komponent w pliku *App.tsx* w taki sposób, by bezpośrednio po uruchomieniu aplikacji pobierał listę kategorii i dodawał je do magazynu *Redux*.

Poniższy fragment kodu przedstawia zapytanie GraphQL-a `GetAllCategories`, które musimy dodać:

```
const GetAllCategories = gql`
  query getAllCategories {
    getAllCategories {
      id
      name
    }
  }
`;
```

```
function App() {
  const { data } = useQuery(GetAllCategories);
  const dispatch = useDispatch();

  useEffect(() => {
    dispatch({
      type: UserProfileSetType,
      payload: {
        id: 1,
        userName: "użytkownikTestowy",
      },
    });
    if (data && data.getAllCategories) {
      dispatch({
        type: ThreadCategoriesType,
        payload: data.getAllCategories,
      });
    }
  });
}
```

Większość tego kodu widziałeś już wcześniej, jednak przypomnę tylko, że to właśnie w tym miejscu przekazujemy nasze dane kategorii (`ThreadCategory`) do magazynu Reduxa.

```
    }, [dispatch, data]);

const renderHome = (props: any) => <Home {...props} />;
const renderThread = (props: any) => <Thread {...props} />;
const renderUserProfile = (props: any) => <UserProfile {...props} />;

return (
  <Switch>
    <Route exact={true} path="/" render={renderHome} />
    <Route path="/categorythreads/:categoryId" render={renderHome} />
    <Route path="/thread/:id" render={renderThread} />
    <Route path="/userprofile/:id" render={renderUserProfile} />
  </Switch>
);
}
```

Reszta kodu pozostaje bez zmian.

4. Z komponentów `LeftMenu` oraz `ThreadCategory` będziemy musieli usunąć kod pobierający kategorie i wyświetlający rozwijaną listę. Jednak zanim to zrobimy, przygotujemy nowy komponent rozwijanej listy kategorii. W tym celu, w katalogu `src/components` utwórz nowy plik, o nazwie `CategoryDropDown.tsx`, i zapisz w nim poniższy kod (upewnij się także, że dodasz do niego wszystkie niezbędne instrukcje importu):

```
const defaultLabel = "Wybierz kategorię";
const defaultOption = {
  value: "0",
  label: defaultLabel,
};
```

Stała `defaultOption` definiuje wartości początkowe naszej rozwijanej listy.



```
class CategoryDropDownProps {
  sendOutSelectedCategory?: (cat: Category) => void;
  navigate?: boolean = false;
  preselectedCategory?: Category;
}
```

Klasa `CategoryDropDownProps` będzie typem określającym parametry naszego nowego komponentu `CategoryDropDown`. `sendSelectedCategory` to funkcja przekazywana przez komponent nadrzędny, której będzie on używał do pobierania aktualnie wybranej kategorii. Z kolei `navigate` będzie wartością logiczną określającą, czy po wybraniu nowej kategorii z listy należy na ekranie wyświetlić jej zawartość, przechodząc na odpowiedni adres URL. I w końcu `preselectedCategory` pozwala komponentowi nadrzędnemu wymusić, by podczas wczytywania aplikacji na liście została zaznaczona konkretna kategoria.

```
const CategoryDropDown: FC<CategoryDropDownProps> = ({
  sendOutSelectedCategory,
  navigate,
  preselectedCategory,
}) => {
  const categories = useSelector((state: AppState) => state.categories);
  const [categoryOptions, setCategoryOptions] = useState<
    Array<string | Option>
  >([defaultOption]);
  const [selectedOption, setSelectedOption] = useState<Option>(default
    ↪Option);
  const history = useHistory();
```

Bazując na wszystkim, czego się już wcześniej nauczyłeś, zrozumienie zastosowanych w tym fragmencie kodu *hooków* będzie bardzo proste. Zwróć tylko uwagę na to, że listę kategorii pobieramy z magazynu `Reduxa`, używając wywołania `useSelector`.

```
useEffect(() => {
  if (categories) {
    const catOptions: Array<Option> = categories.map((cat: Category) => {
      return {
        value: cat.id,
        label: cat.name,
      };
    });
  }
});
```

Ten fragment kodu przygotowuje tablicę opcji, które później posłużą nam do utworzenia rozwijanej listy.

```
setCategoryOptions(catOptions);
```

W tym wywołaniu, `setCategoryOptions`, odbieramy tablicę kategorii do utworzenia opcji listy i zapisujemy ją, dzięki czemu później będzie jej można użyć do zbudowania listy.

```
setSelectedOption({
  value: preselectedCategory ? preselectedCategory.id : "0",
  label: preselectedCategory ? preselectedCategory.name :
    ↪defaultLabel,
});
```

W tym fragmencie ustawiamy domyślnie wybraną opcję listy.

```

    }
  }, [categories, preselectedCategory]));

const onChangeDropDown = (selected: Option) => {
  setSelectedOption(selected);
  if (sendOutSelectedCategory) {
    sendOutSelectedCategory(
      new Category(selected.value, selected.label?.valueOf().toString() ?? "")
    );
  }
}
```

W tej procedurze obsługi powiadamy komponent nadrzędny o fakcie zmiany aktualnie wybranej opcji listy.

```

  if (navigate) {
    history.push(`~/categorythreads/${selected.value}`);
  }
}
```

Jeśli komponent nadrzędny o to poprosił, to następnie przechodzimy do odpowiedniej trasy ThreadCategory.

```

    return (
      <DropDown
        className="thread-category-dropdown"
        options={categoryOptions}
        onChange={onChangeDropDown}
        value={selectedOption}
        placeholder={defaultLabel}
      />
    );
  };
};

export default CategoryDropDown;
```

I na samym końcu komponentu generujemy bardzo prosty kod JSX.

5. Teraz musimy zmodyfikować kod umieszczony w pliku *MainHeader.tsx*; poniżej pokazałem, jak to zrobić:

```

interface MainHeaderProps {
  category?: Category;
}

const MainHeader: FC<MainHeaderProps> = ({ category }) => {
  const { width } = useWindowDimensions();
```

Jedyną poważną zmianą w tym komponencie jest dodanie funkcji `getLabelElement`, która określa, czy aplikacja działa na urządzeniu mobilnym; jeśli okaże się, że faktycznie tak jest, to renderuje komponent `CategoryDropDown`:

```

const getLabelElement = () => {
  if (width <= 768) {
    return (
      <CategoryDropDown navigate={true} preselectedCategory={category} />
    );
  }
```

```

    } else {
      return <strong>{category?.name || "Treść tymczasowa"}</strong>;
    }
  };

  return (
    <div className="main-header">
      <div
        className="title-bar"
        style={{ marginBottom: ".25em", paddingBottom: "0" }}
      >
        {getLabelElement()}

```

Tu wywołujemy funkcję `getLabelElement`.

```

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

```

Pozostałe zmiany, jakie należy wprowadzić, sprowadzają się w znacznej mierze do usuwania niepotrzebnych fragmentów kodu. Oczywiście, w razie potrzeby możesz poszukać pomocy, zaglądając do kodów źródłowych dołączonych do książki. Zmiany będziesz musiał wprowadzić w plikach *ThreadCategory.tsx*, *LeftMenu.tsx* oraz *Thread.css*.

## Możliwości związane z uwierzytelnianiem

Kolejne zmiany, które wprowadzimy w aplikacji, będą powiązane z mechanizmami uwierzytelniania. Pamiętaj, że *zanim* użytkownicy będą mogli się zalogować w aplikacji, w ich rekordach w bazie danych będziesz musiał zapisać wartość `true` w kolumnie `confirmed`.

1. Pierwszą rzeczą, którą się zajmiemy, będzie zapewnienie użytkownikom możliwości zalogowania się do aplikacji. Aby to zrobić, oraz aby później mieć możliwość aktualizowania obiektu `User` w globalnym magazynie `Reduxa`, musimy zmodyfikować reduktor użytkownika.

Zacznij od utworzenia w katalogu *models* nowego pliku *User.ts* i dodania do niego kodu z przykładów dołączonych do książki. Zwróć uwagę na to, że klasa `User` definiuje pole `threads`. Będziemy w nim zapisywać nie tylko wątki (`Thread`) należące do danego użytkownika, lecz także odpowiedzi (`ThreadItem`) opublikowane w tych wątkach.

2. Kolejnym zadaniem będzie aktualizacja reduktora. Otwórz plik *store/user/Reducer.ts* i wprowadź w nim następujące zmiany: usuń interfejs `UserProfilePayload` i zastąp wszystkie odwołania do niego nową klasą `User`. Jeśli będziesz potrzebował pomocy, to zajrzyj do kodów źródłowych.
3. Teraz możesz zająć się aktualizacją komponentu `Login`; zmiany, które należy w nim wprowadzić, przedstawiłem poniżej. Nie zapomnij o odpowiednim zaktualizowaniu instrukcji importu.

Zwróć uwagę na to, że zaimportowaliśmy także *hook* `useRefreshReduxMe`. Zdefiniujemy go już niebawem, jednak zanim to zrobimy, chciałbym przedstawić kilka możliwości, jakie daje *hook* `useMutation GraphQL-a`:

```
const LoginMutation = gql`
  mutation Login($userName: String!, $password: String!) {
    login(userName: $userName, password: $password)
  }
`;
```

A oto i nasza mutacja Login:

```
const Login: FC<ModalProps> = ({ isOpen, onClickToggle }) => {
  const [execLogin] = useMutation(LoginMutation, {
    refetchQueries: [
      {
        query: Me,
      },
    ],
  });
```

Chciałbym teraz nieco dokładniej wyjaśnić to wywołanie `useMutation`. Jego pierwszym wywołaniem jest mutacja, `LoginMutation`, a drugim obiekt zawierający właściwość o nazwie `refetchQueries`. Właściwość ta wymusza ponowne wykonanie wszystkich podanych w niej zapytań i zapisanie zwróconych przez nie wyników w pamięci podręcznej. Gdybyśmy nie zastosowali tej właściwości, a następnie ponownie wykonali zapytanie `Me`, to w efekcie uzyskalibyśmy nie nowe dane, lecz ich wcześniejszą wersję, pobraną z pamięci podręcznej. Zwróć uwagę na to, że wywołanie `useMutation` nie powoduje automatycznego odświeżenia żadnych wywołań zależnych od podanych zapytań; wciąż będziemy musieli samodzielnie je wykonać, by pobrać nowe dane.

Wynik zwracany przez to wywołanie, `execLogin`, jest funkcją, którą będzie można później wywołać.

```
const [
  { userName, password, resultMsg, isSubmitDisabled },
  dispatch,
] = useReducer(userReducer, {
  userName: "tester",
  password: "Test123$%^",
  resultMsg: "",
  isSubmitDisabled: false,
});
const { execMe, updateMe } = useRefreshReduxMe();

const onChangeUserName = (e: React.ChangeEvent<HTMLInputElement>) => {
  dispatch({ type: "userName", payload: e.target.value });
  if (!e.target.value)
    allowSubmit(dispatch, "Nazwa użytkownika nie może być pusta", true);
  else allowSubmit(dispatch, "", false);
};

const onChangePassword = (e: React.ChangeEvent<HTMLInputElement>) => {
```

```

dispatch({ type: "password", payload: e.target.value });
if (!e.target.value)
  allowSubmit(dispatch, "Hasło nie może być puste", true);
else allowSubmit(dispatch, "", false);
};

```

Powyższe wywołania są takie same, jak wcześniej.

```

const onClickLogin = async (
  e: React.MouseEvent<HTMLButtonElement, MouseEvent>
) => {
  e.preventDefault();
  onClickToggle(e);
  const result = await execLogin({
    variables: {
      userName,
      password,
    },
  });
  execMe();
  updateMe();
};

```

Aktualnie procedura obsługi zdarzeń `onClickLogin` wywołuje funkcję `execLogin`, przekazując do niej odpowiednie parametry. Po zakończeniu wywołania `execLogin`, zostaną automatycznie wykonane wszystkie zapytania podane na liście `refetchQueries`. Na samym końcu wywołujemy dwie funkcje zwrócone przez nasz *hook* `useRefresh` ↪ `ReduxMe`, czyli `execMe` oraz `updateMe`. Pierwsza z nich, `execMe`, pobiera najnowszy obiekt `User`, natomiast druga, `updateMe`, dodaje go do magazynu `Reduxa`. Pozostała część kodu pliku *Login.tsx* jest taka sama jak wcześniej, więc nie będę jej tutaj prezentował.

4. Teraz zajmiemy się zdefiniowaniem *hooka* `useRefreshReduxMe`. Ten *hook* chcemy utworzyć w taki sposób, by kod obsługujący operacje ustawiania i usuwania obiektu `User` z magazynu `Reduxa` znajdował się w jednym pliku. Tego *hooka* będziemy używali w kilku komponentach. Utwórz zatem w katalogu *hooks* plik o nazwie *useRefreshReduxMe.ts* i zapisz w nim kod źródłowy skopiowany z kodów źródłowych dołączonych do książki.

Na samym początku pliku została zdefiniowana stała `Me`, która zawiera zapytanie pobierające informacje o użytkowniku. Fragment `EntityResult` służy do pobierania komunikatów tekstowych, jeśli takie zostaną zwrócone. Jeśli uda się pobrać faktyczne dane użytkownika, to pola, w jakich zostaną one zapisane, zostały zdefiniowane w drugim fragmencie, `User`.

Zdefiniowany poniżej interfejs `UseRefreshReduxMeResult` posłuży nam do określania typu wyniku zwracanego przez tworzony *hook*.

Następnie, w wierszu 37., zastosowaliśmy wywołanie `useLazyQuery`; dzięki temu użytkownicy naszego *hooka* będą mogli wykonać zapytanie `Me` w wybranym przez siebie momencie.

Zdefiniowana poniżej funkcja `deleteMe` pozwoli użytkownikom naszego *hooka* usuwać obiekt `User` z magazynu `Reduxa`, kiedy pojawi się taka potrzeba, na przykład kiedy użytkownik będzie chciał się wylogować.

Ostatnią z funkcji jest `updateMe`, która pozwala na zapisanie obiektu `User` w magazynie `Reduxa`. Na samym końcu zwracamy te wszystkie trzy funkcje, by użytkownicy naszego *hooka* mogli z nich skorzystać.

5. Podczas wczytywania aplikacji powinniśmy sprawdzić, czy użytkownik jest zalogowany oraz kim on jest. Otwórz zatem plik *App.tsx* i zaktualizuj go zgodnie z poniższym przykładem:

```
function App() {
  const { data: categoriesData } = useQuery(GetAllCategories);
  const { execMe, updateMe } = useRefreshReduxMe();
```

W tym fragmencie inicjujemy *hook* `useRefreshReduxMe`.

```
  const dispatch = useDispatch();

  useEffect(() => {
    execMe();
  }, [execMe]);
```

W tym fragmencie wywołujemy funkcję `execMe`, żeby pobrać obiekt `User` z `GraphQL-a`.

```
  useEffect(() => {
    updateMe();
  }, [updateMe]);
```

A tutaj wywołujemy funkcję `updateMe`, żeby przekazać do odpowiedniego reduktora `Reduxa` dane użytkownika, jeśli jakieś będą.

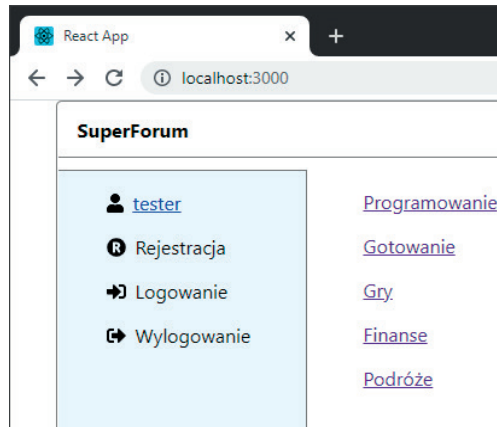
```
  useEffect(() => {
    if (categoriesData && categoriesData.getAllCategories) {
      dispatch({
        type: ThreadCategoriesType,
        payload: categoriesData.getAllCategories,
      });
    }
  }, [dispatch, categoriesData]);
```

Tu zmieniłem nazwę pola danych na `categoriesData`, aby łatwiej było zrozumieć jego przeznaczenie. Pozostała część kodu zostaje bez zmian.

6. Jeśli teraz zalogujesz się do aplikacji, zauważysz, że w komponencie `SideBar` będzie wyświetlana nazwa aktualnie zalogowanego użytkownika (patrz rysunek 15.13).

A zatem, możemy się już zalogować i nawet jest wyświetlana nazwa zalogowanego użytkownika.

Super! Ale teraz musimy poprawić komponent `SideBar` tak, by były na nim wyświetlane odnośniki odpowiadające aktualnemu stanowi. Na przykład, jeśli użytkownik jest zalogowany, to odnośniki *Logowanie* i *Rejestracja* nie powinny być widoczne.



Rysunek 15.13. Zalogowany użytkownik

1. Abyśmy mogli upewnić się, że w komponencie `SideBar` będą wyświetlane właściwe odnośniki, w pierwszej kolejności musimy zaktualizować komponent `Logout`. Zaczniemy od upewnienia się, że na początku pliku znajdują się wszystkie konieczne instrukcje importu.

```
const LogoutMutation = gql`
  mutation logout($userName: String!) {
    logout(userName: $userName)
  }
`;
```

Ten fragment kodu definiuje mutację `logout`.

```
const Logout: FC<ModalProps> = ({ isOpen, onClickToggle }) => {
  const user = useSelector((state: AppState) => state.user);
  const [execLogout] = useMutation(LogoutMutation, {
    refetchQueries: [
      {
        query: Me,
      },
    ],
  });
};
```

Także tutaj wymuszamy odświeżenie pamięci podręcznej GraphQL-a i zapisanych w niej wyników zapytania `Me`.

```
const { deleteMe } = useRefreshReduxMe();

const onClickLogin = async (
  e: React.MouseEvent<HTMLButtonElement, MouseEvent>
) => {
  e.preventDefault();
  onClickToggle(e);
  await execLogout({
    variables: {
      userName: user?.userName ?? "",
    },
  });
};
```

```

    });
    deleteMe();
  };

```

Na początku tego fragmentu kodu używamy naszego *hooka* `useRefreshReduxMe`, jednak tym razem interesuje nas wyłącznie funkcja `deleteMe`, gdyż naszym celem jest wylogowanie użytkownika. Pozostała część kodu nie ulega zmianie, więc nie będę jej tutaj przedstawiał.

2. Teraz zajmiemy się komponentem `SideBarMenus` i zmodyfikujemy go w taki sposób, by zawsze były widoczne tylko odpowiednie odnośniki. A zatem, otwórz plik i zmodyfikuj go zgodnie z poniższymi przykładami.

W tym przypadku będę przedstawiał wyłącznie zwracany kod JSX, gdyż tylko on się zmienia (pomijając instrukcje importu):

```

return (
  <React.Fragment>
    <ul>
      {user ? (
        <li>
          <FontAwesomeIcon icon={faUser} />
          <span className="menu-name">
            <Link to={`~/userprofile/${user?.id}`}>{user?.userName}</Link>
          </span>
        </li>
      ) : null}
    </ul>
  )

```

Jak widać, w tym fragmencie sprawdzamy, czy obiekt `user` ma wartość, a jeśli ma, to wyświetlamy ten sam interfejs użytkownika, przedstawiający wartość `userName`; w przeciwnym przypadku, jeśli obiekt `user` nie ma wartości, to nic nie wyświetlamy.

```

{user ? null : (
  <li>
    <FontAwesomeIcon icon={faRegistered} />
    <span onClick={onClickToggleRegister} className="menu-name">
      Rejestracja
    </span>
    <Registration
      isOpen={showRegister}
      onClickToggle={onClickToggleRegister}
    />
  </li>
)}

```

W tym przypadku nie chcemy wyświetlać odnośnika do ekranu rejestracyjnego, jeśli użytkownik istnieje — i tak też robimy.

```

{user ? null : (
  <li>
    <FontAwesomeIcon icon={faSignInAlt} />
    <span onClick={onClickToggleLogin} className="menu-name">
      Logowanie
    </span>
  </li>
)}

```



```

        <Login isOpen={showLogin} onClickToggle={onClickToggleLogin} />
      </li>
    )}
  )}

```

Także w tym fragmencie kodu nie wyświetlamy odnośnika do ekranu logowania, jeśli użytkownik istnieje, gdyż to oznacza, że jest on już zalogowany.

```

    {user ? (
      <li>
        <FontAwesomeIcon icon={faSignOutAlt} />
        <span onClick={onClickToggleLogout} className="menu-name">
          Wylogowanie
        </span>
        <Logout isOpen={showLogout} onClickToggle={onClickToggleLogout} />
      </li>
    ) : null}

```

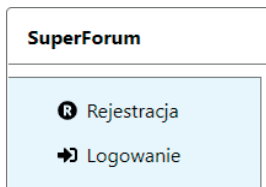
W tym fragmencie kodu, jeśli obiekt user ma wartość, to wyświetlamy odnośnik do wylogowania.

```

    </ul>
  </React.Fragment>
);
};

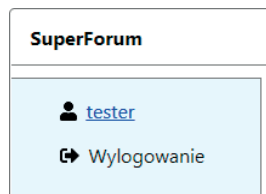
```

3. Obecnie, kiedy użytkownik nie będzie zalogowany, boczne menu będzie wyglądać tak, jak na rysunku 15.14.



**Rysunek 15.14.** Postać komponentu SideBarMenu, kiedy nie ma zalogowanego użytkownika

Natomiast po zalogowaniu użytkownika będzie ono wyglądać jak na rysunku 15.15.



**Rysunek 15.15.** Postać komponentu SideBarMenu po zalogowaniu

Jak widać, nasz pasek boczny aktualnie już wyświetla odpowiednie odnośniki i teksty. Możemy zatem zająć się ekranem profilu użytkownika.

## Ekran profilu użytkownika

Ponieważ zajmujemy się zagadnieniami związanymi z uwierzytelnianiem, jest to dobra okazja, by zmodyfikować ekran profilu użytkownika. Będziemy musieli wprowadzić w nim całkiem sporo zmian.

1. W pierwszej kolejności zaktualizujemy kod GraphQL-a, dodając niezbędne pole do typu User. A zatem, w projekcie serwerowej części aplikacji zmodyfikuj typ User w pliku *typeDefs.ts*: dodaj w nim poniższe pole bezpośrednio poniżej pola threads:

```
threadItems: [ThreadItem!]
```

Jak widać, dodaliśmy tylko jedno nowe pole, threadItems. Zwróć uwagę na to, że nie ma ono nic wspólnego z polem o tej samej nazwie stanowiącym zawartość pola threads, gdyż teraz chodzi nam o encje ThreadItems opublikowane przez danego użytkownika.

2. Oprócz tego musimy także zmodyfikować encję User i także do niej dodać jedno nowe pole. A zatem, dodaj poniższe pole do pliku *User.ts*:

```
@OneToMany(() => ThreadItem, (threadItem) => threadItem.user)
threadItems: ThreadItem[];
```

Dzięki niemu nasza encja User będzie mogła mieć powiązane ze sobą encje ThreadItem. Upewnij się także, że pole odpowiadające tej zależności znajdzie się także w pliku *ThreadItem.ts*:

```
@ManyToOne(() => User, (user) => user.threadItems)
user: User;
```

3. Kolejnym krokiem będzie zmodyfikowanie funkcji me w pliku warstwy repozytorium *UserRepo.ts*; musimy dodać do niej powiązania z encją ThreadItem. W tym celu zaktualizuj w tym pliku funkcję User.findOne w następujący sposób:

```
relations: [
  "threads",
  "threads.threadItems",
  "threadItems",
  "threadItems.thread",
],
```

W tym kodzie wprowadziliśmy tylko jedną zmianę, wyróżnioną pogrubieniem: dodaliśmy dwie zależności: threadItems oraz threadItems.thread.

4. Zauważysz zapewne, że ekran profilu użytkownika zapewnia możliwość zmiany hasła. Zaimplementujmy ją zatem. W tym celu, w pierwszej kolejności musimy dodać do pliku definicji typów (*typeDefs.ts*) nową mutację. Wstaw poniższy wiersz kodu do sekcji Mutation:

```
changePassword(newPassword: String!): String!
```

Znaczenie tej mutacji jest chyba oczywiste.

5. Teraz zaimplementujemy tę funkcję w pliku *UserRepo.ts*. Skopiuj funkcję `changePassword` z przykładów dołączonych do książki i wklej ją na końcu pliku.

Zakładamy, że jeśli ta funkcja została wywołana, użytkownik musi być zalogowany, więc na samym początku, w wierszu 125., oczekujemy, że wartość parametru `id` przekazana z kodu resolvera będzie określona. Jeśli ta wartość nie będzie istnieć, wywołanie funkcji zakończy się zgłoszeniem odpowiedniego błędu.

Następnie próbujemy pobrać obiekt użytkownika, po czym wykonujemy parę testów, by upewnić się, że jest on prawidłowy. W końcu wywołujemy funkcję `bcrypt`, by wygenerować hasło w postaci zaszyfrowanej.

6. Teraz możemy zająć się przygotowaniem resolvera. W tym celu otwórz plik *resolvers.ts* i do jego sekcji `Mutation` dodaj kod funkcji `changePassword` skopiowany z analogicznego pliku w przykładach do książki.

Na samym początku funkcji sprawdzamy, czy istnieje sesja i czy jest w niej ustawiona wartość pola `userId`, gdyż to właśnie ono określa, czy użytkownik jest zalogowany, czy nie.

Następnie wywołujemy funkcję `changePassword` warstwy repozytorium, przekazując do niej identyfikator użytkownika (`userId`) oraz nowe hasło.

7. Teraz możemy zająć się aktualizowaniem kodu komponentu `UserProfile`. Zmodyfikuj go zgodnie z poniższym opisem.

Przed wszystkim uzupełnij instrukcje importu, gdyż będziemy potrzebować dwóch nowych elementów: `gql` oraz `useMutation`.

```
const ChangePassword = gql`
  mutation ChangePassword($newPassword: String!) {
    changePassword(newPassword: $newPassword)
  }
`;
```

W tym fragmencie definiujemy nową mutację, `ChangePassword`.

```
const UserProfile = () => {
  const [
    { userName, password, passwordConfirm, resultMsg, isSubmitDisabled },
    dispatch,
  ] = useReducer(userReducer, {
    userName: "",
    password: "*****",
    passwordConfirm: "*****",
    resultMsg: "",
    isSubmitDisabled: true,
  });
  const user = useSelector((state: AppState) => state.user);
  const [threads, setThreads] = useState<JSX.Element | undefined>();
  const [threadItems, setThreadItems] = useState<JSX.Element | undefined>();
  const [execChangePassword] = useMutation(ChangePassword);
```

Tutaj przygotowujemy naszą mutację `ChangePassword` do użycia — przekazujemy ją w wywołaniu `useMutation`.

Poniższy kod, zawierający wywołanie funkcji `useEffect`, jest taki sam, jak wcześniej:

```
useEffect(() => {
  console.log("user", user);
  if (user) {
    dispatch({
      type: "userName",
      payload: user.userName,
    });

    getUserThreads(user.id).then((items) => {
      const threadItemsInThreadList: Array<ThreadItem> = [];
      const threadList = items.map((th: Thread) => {
        for (let i = 0; i < th.threadItems.length; i++) {
          threadItemsInThreadList.push(th.threadItems[i]);
        }

        return (
          <li key={`user-th-${th.id}`}>
            <Link to={`/thread/${th.id}`} className="userprofile-link">
              {th.title}
            </Link>
          </li>
        );
      });
    });
    setThreads(<ul>{threadList}</ul>);

    const threadItemList = threadItemsInThreadList.map((ti: ThreadItem) => (
      <li key={`user-th-${ti.threadId}`}>
        <Link to={`/thread/${ti.threadId}`} className="userprofile-link">
          {ti.body}
        </Link>
      </li>
    ));
    setThreadItems(<ul>{threadItemList}</ul>);
  }
}, [user]);
```

Nowym elementem kodu jest funkcja `onClickChangePassword` — inicjuje ona wywołanie funkcji `changePassword`, a następnie aktualizuje komunikat wyświetlany w interfejsie użytkownika.

```
const onClickChangePassword = async (
  e: React.MouseEvent<HTMLButtonElement, MouseEvent>
) => {
  e.preventDefault();
  const { data: changePasswordData } = await execChangePassword({
    variables: {
      newPassword: password,
    },
  });
  dispatch({
    type: "resultMsg",
```

```

      payload: changePasswordData ? changePasswordData.changePassword : "",
    });
  };

  return (
    <div className="screen-root-container">
      <div className="thread-nav-container">
        <Nav />
      </div>
      <form className="userprofile-content-container">
        <div>
          <strong>Profil użytkownika</strong>
          <label style={{ marginLeft: ".75em" }}>{userName}</label>
        </div>
        <div className="userprofile-password">
          <div>
            <PasswordComparison
              dispatch={dispatch}
              password={password}
              passwordConfirm={passwordConfirm}
            />
            <button
              className="action-btn"
              disabled={isSubmitDisabled}
              onClick={onClickChangePassword}>

```

W tym fragmencie, w przycisku *Zmień hasło*, używamy zdefiniowanej wcześniej funkcji `onClickChangePassword`.

```

      Zmień hasło
    </button>
  </div>
  <div style={{ marginTop: ".5em" }}>
    <label>{resultMsg}</label>
  </div>
</div>
<div className="userprofile-postings">
  <hr className="thread-section-divider" />
  <div className="userprofile-threads">
    <strong>0 publikowane wątki</strong>
    {threads}
  </div>
  <div className="userprofile-threadItems">
    <strong>0 publikowane odpowiedzi</strong>
    {threadItems}
  </div>
</div>
</form>
</div>
);
};

```

```
export default UserProfile;
```

Pozostała część kodu nie zmienia się.

A teraz zajmijmy się przedstawieniem wątków i odpowiedzi użytkownika.

1. W pierwszej kolejności musimy poprawić model użytkownika — klasę `User`. Dodaj poniższy wiersz kodu do pliku `Users.ts`:

```
public threadItems: Array<ThreadItem>
```

2. Kolejnym elementem, który musimy zmienić, jest zapytanie `Me` zdefiniowane w *hooku* `useRefreshRedusMe`; poniżej pokazałem, jak teraz powinno ono wyglądać:

```
export const Me = gql`
  query me {
    me {
      ... on EntityResult {
        messages
      }
      ... on User {
        id
        userName
        threads {
          id
          title
        }
        threadItems {
          id
          thread {
            id
          }
          body
        }
      }
    }
  }
`;
```

Zmieniliśmy `threadItems` w taki sposób, że teraz zamiast pobierać odpowiedzi (`threadItems`) powiązane z konkretnym wątkiem (`thread`), pobiera odpowiedzi powiązane z użytkownikiem. Co więcej, teraz dla każdej odpowiedzi pobieramy także identyfikator wątku.

3. Następnie, w komponencie `UserProfile`, zmień funkcję `useEffect` tak, jak pokazałem na poniższym przykładzie:

```
useEffect(() => {
  if (user) {
    dispatch({
      type: "userName",
      payload: user.userName,
    });
  }
});
```

Teraz wątki pobieramy z tablicy `user.threads`, a nie z usługi `dataService`, co wyraźnie widać w poniższym fragmencie:

```
const threadList = user.threads?.map((th: Thread) => {
  return (
    <li key={`user-th-${th.id}`}>
      <Link to={`~/thread/${th.id}`} className="userprofile-link">
```

```

        {th.title}
      </Link>
    </li>
  );
});
setThreads(
  !user.threadItems || user.threadItems.length === 0 ? undefined : (
    <ul>{threadList}</ul>
  )
);

```

Dokładnie w taki sam sposób pobieramy odpowiedzi (threadItems). Zwróć uwagę na zmianę wartości właściwości to komponentu Link; wcześniej było nią wyrażenie `ti.threadId`, natomiast teraz jest ona określona jako `ti.thread?.id`:

```

const threadItemList = user.threadItems?.map((ti: any) => (
  <li key={`user-ti-${ti.id}`}>
    <Link to={`/thread/${ti.thread?.id}`} className="userprofile-link">
      {ti.body.length <= 40 ? ti.body : ti.body.substring(0, 40) + " ..."}
    </Link>
  </li>
));

```

W tym fragmencie kodu dodaliśmy nieco logiki służącej do formatowania długich fragmentów tekstu, które mogłyby wychodzić poza ekran i być przez przeglądarkę zawijane. Najprościej rzecz ujmując, jeśli długość tekstu przekracza 40 znaków, to jest on przycinany do tej długości, a na jego końcu jest dopisywany łańcuch " ...".

```

    </Link>
  </li>
));
setThreadItems(
  !user.threadItems || user.threadItems.length === 0 ? undefined : (
    <ul>{threadItemList}</ul>
  )
);
} else {
  dispatch({
    type: "userName",
    payload: "",
  });
  setThreads(undefined);
  setThreadItems(undefined);
}
}, [user]);

```

Pozostała część kodu pozostaje taka sama. Kiedy wyświetlisz tę nową stronę profilu użytkownika, to powinna ona wyglądać podobnie, jak na rysunku 15.16 (pamiętaj, że Twoje dane będą zapewne wyglądały inaczej).

No dobrze. W ten sposób zakończyliśmy aktualizację panelu użytkownika. Ponieważ zakres zmian przedstawionych w tym rozdziale i tak był ogromny, pozostałe prace nad aplikacją wykonamy w rozdziale 16., pt. „Dodawanie schematu GraphQL-a — część 2.”.

**SuperForum**

**Profil użytkownika** tester

Hasło

.....

Powtórz hasło

.....

Zmień hasło

**Opublikowane wątki**

[Wątek 1](#)

[Wątek 2](#)

**Opublikowane odpowiedzi**

[Odpowiedź 1 \(ThreadItem\)](#)

[Odpowiedź 1 \(ThreadItem\)](#)

Rysunek 15.16. Wątki i odpowiedzi użytkownika

## Podsumowanie

W tym rozdziale niemal udało się nam zakończyć aplikację i zintegrować jej część kliencką z częścią serwerową, wykorzystując do tego GraphQL-a. To był długi i złożony rozdział, więc powinieneś czuć satysfakcję z tego, co udało Ci się osiągnąć.

W następnym rozdziale, pt. „Dodawanie schematu GraphQL-a — część 2.”, dokończymy prace nad aplikacją, uruchamiając do końca ekran wątków, a konkretnie zaimplementujemy możliwości przeglądania i publikowania wątków i odpowiedzi oraz uruchomimy system punktacji wątków, by użytkownicy mogli się zorientować, które z nich są najbardziej popularne.



# Dodawanie schematu GraphQL-a — część 2.

W tym rozdziale będziemy kontynuowali prace nad dokończeniem kodu klienckiej i serwerowej części naszej aplikacji. Dokończymy w nim prace nad ekranami prezentującymi wątki, zapewnimy możliwość publikowania nowych wątków oraz odpowiedzi, jak również dokończymy system punktacji. Punktem wyjścia do działań w tym rozdziale będą kody źródłowe z rozdziału 15., pt. „Dodawanie schematu GraphQL-a — część 1.”.

## Komponent Thread i jego trasa

W tym podrozdziale zajmiemy się aktualizacją komponentu Thread i jego krasą thread. Zmiany, które tu opiszę, będą dotyczyły sporej liczby plików. Wykonaj czynności opisane na poniższej liście:

1. Otwórz plik definicji typów (*typedefs.ts*) i zmodyfikuj typy Thread i ThreadItem. Dodaj poniższy wiersz kodu bezpośrednio poniżej pola views:
2. Teraz otwórz plik *ThreadRepo.ts* i zmodyfikuj funkcję getThreadById zgodnie z poniższym przykładem:

```
points: Int!  
  
export const getThreadById = async (  
  id: string  
) : Promise<QueryOneResult<Thread>> => {
```

```
const thread = await Thread.findOne({
  where: {
    id,
  },
  relations: [
    "user",
    "threadItems",
    "threadItems.user",
    "category",
  ],
});
```

W tym fragmencie kodu dodaliśmy jedynie nowe relacje w wywołaniu funkcji `findOne`.

```
if (!thread) {
  return {
    messages: ["Nie znaleziono wątku."],
  };
}
return {
  entity: thread,
};
};
```

3. Następnie zmodyfikuj zgodnie z poniższym przykładem wywołanie funkcji `Thread.createQueryBuilder`, umieszczone w kodzie funkcji `getThreadsByCategoryId`:

```
const threads = await Thread.createQueryBuilder("thread")
  .where(`thread.categoryId = :categoryId`, { categoryId })
  .leftJoinAndSelect("thread.category", "category")
  .leftJoinAndSelect("thread.threadItems", "threadItems")
  .leftJoinAndSelect("thread.user", "user")
  .orderBy("thread.createdOn", "DESC")
  .getMany();
```

W tym wywołaniu dołączyliśmy relację z encją `User`. Pozostała część kodu tej funkcji nie uległa zmianie.

4. Teraz otwórz plik *User.ts* w klienckiej części aplikacji i zmodyfikuj pola `threads` i `threadItems` tak, by były opcjonalne. To konieczne, by można było dodać nowe konto użytkownika, który początkowo nie będzie mieć żadnych wątków ani odpowiedzi:

```
public threads?: Array<Thread>,
public threadItems?: Array<ThreadItem>
```

5. Następnie otwórz kolejne dwa pliki z klienckiej części aplikacji, *models/Thread.ts* oraz *models/ThreadItem.ts*, i zastąp pola `userName` i `userId` jednym polem `user`:

```
public user: User,
```

6. Musimy także zastąpić odwołania do pól `userName` i `userId` występujące w kodzie pliku *DataService.ts* odwołaniami do obiektu `User`. Możesz utworzyć ten obiekt na początku pliku, a następnie używać go w kodzie zamiast dwóch powyższych pól.

```
const user = new User("10", "tester@test.com", "tester");
```

Gdybyś potrzebował pomocy, zajrzyj do kodu pliku *DataService.ts*, choć zmiany, które tu trzeba wprowadzić, są trywialne.

7. Skoro poprawiliśmy już typ *User* oraz encje w schemacie GraphQL-a, możemy zająć się zapytaniami. W pliku *Main.tsx* zmodyfikuj kod zapytań *GetThreadsByCategoryId* oraz *GetThreadsLatest* zgodnie z poniższymi przykładami:

```
const GetThreadsByCategoryId = gql`
  query getThreadsByCategoryId($categoryId: ID!) {
    getThreadsByCategoryId(categoryId: $categoryId) {
      ... on EntityResult {
        messages
      }

      ... on ThreadArray {
        threads {
          id
          title
          body
          views
          points
          user {
            userName
          }
          threadItems {
            id
          }
          category {
            id
            name
          }
        }
      }
    }
  }
`;
```

W obu zapytaniach dodajemy pola *points* i *user*.

```
const GetThreadsLatest = gql`
  query getThreadsLatest {
    getThreadsLatest {
      ... on EntityResult {
        messages
      }

      ... on ThreadArray {
        threads {
          id
          title
          body
          views
          points
          user {
            userName
          }
        }
      }
    }
  }
`;
```

```

        threadItems {
          id
        }
        category {
          id
          name
        }
      }
    }
  }
};

```

8. A teraz, w pliku *ThreadCard.tsx* odzyskaj poniższy fragment kodu:

```

<span className="username-header" style={{ marginLeft: ".5em" }}>
  {thread.userName}
</span>

```

i zmień go w następujący sposób:

```

<span className="username-header" style={{ marginLeft: ".5em" }}>
  {thread.user.userName}
</span>

```

Jak widać, teraz zamiast odwoływać się do pola `userName` bezpośrednio, robimy to za pośrednictwem obiektu `user`.

9. Kolejne zmiany musimy wprowadzić w pliku *RichEditor.tsx*. Zwróć uwagę na to, że nasz ekran wątku będzie przedstawiał tekst wpisany przez użytkownika. A zatem, kiedy użytkownik już opublikuje swój wpis, musimy zadbać o to, by nie mógł go modyfikować. W tym celu do komponentu dodamy nową właściwość *props* określającą, czy tekst ma być przeznaczony wyłącznie do odczytu.

Zacznij od przekształcenia interfejsu `RichEditorProps` na klasę i dodania do niej kolejnej właściwości, jak pokazałem poniżej:

```

class RichEditorProps {
  existingBody?: string;
  readOnly?: boolean = false;
}

```

Tę zmianę wprowadziliśmy po to, by wartość `false` była domyślnym ustawieniem właściwości `readOnly` (interfejsy nie pozwalają na określanie wartości domyślnych). Teraz zmodyfikuj listę parametrów komponentu `RichEditor` tak, jak pokazałem poniżej:

```

const RichEditor: FC<RichEditorProps> = ({ existingBody, readOnly }) => {

```

Aby dodać pole `readOnly` jako parametr, zastosujemy mechanizm destrukuryzacji. Następnie, w kodzie elementu `Editable` dodaj atrybut `readOnly`:

```

<Editable
  className="editor"
  renderElement={renderElement}
  renderLeaf={renderLeaf}
  placeholder="Tu wpisz jakiś tekst..."

```

```

spellCheck
autoFocus
onKeyDown={(event) => {
  for (const hotkey in HOTKEYS) {
    if (isHotkey(hotkey, event as any)) {
      event.preventDefault();
      const mark = HOTKEYS[hotkey];
      toggleMark(editor, mark);
    }
  }
}}
readOnly={readOnly}
/>

```

Jak widać, cała zmiana polegała na dodaniu atrybutu `readOnly` i przypisaniu mu wartości właściwości `props` o tej samej nazwie.

10. Teraz otwórz plik `src/components/routes/thread/Thread.tsx` — główny plik służący do wczytywania trasy prezentującej wątek. Musimy wprowadzić w nim parę zmian.

W poniższym fragmencie dodajemy do pliku zapytanie GraphQL-a, które pozwoli nam pobrać niezbędne dane wątku (`Thread`):

```

const GetThreadById = gql`
  query GetThreadById($id: ID!) {
    getThreadById(id: $id) {
      ... on EntityResult {
        messages
      }

      ... on Thread {
        id
        user {
          userName
        }
        lastModifiedOn
        title
        body
        points
        category {
          id
          name
        }
        threadItems {
          id
          body
          points
          user {
            userName
          }
        }
      }
    }
  }
`;

```

```
const Thread = () => {
  const [execGetThreadId, { data: threadData }] =
    useLazyQuery(GetThreadId);
```

W tym fragmencie kodu używamy zapytania `GetThreadId` oraz *hooka* `useLazyQuery`. Wywołanie `useLazyQuery` zwraca funkcję, `execGetThreadId`, której użyjemy w dalszej części kodu.

```
const [thread, setThread] = useState<ThreadModel | undefined>();
```

Obiektu stanu `thread` będziemy używać do prezentowania informacji w interfejsie użytkownika oraz udostępniania ich innym komponentom.

```
const { id } = useParams();
```

`id` to parametr przekazany w adresie URL, reprezentujący identyfikator wątku.

```
const [readOnly, setReadOnly] = useState(false);
```

Właściwości stanu `readOnly` będziemy używać do przełączenia edytora w tryb tylko do odczytu, kiedy będziemy mieć do czynienia z wątkiem, który już jest zapisany w bazie.

```
useEffect(() => {
  if (id && id > 0) {
    console.log("Id wątku", id);
    execGetThreadId({
      variables: {
        id,
      },
    });
  }
});
```

W tym fragmencie kodu wywołujemy funkcję `execGetThreadId`, przekazując do niej jako parametr `id`, czyli identyfikator wątku przekazany w adresie URL.

```
    }, [id, execGetThreadId]);

    useEffect(() => {
      console.log("Obiekt threadData", threadData);
      if (threadData && threadData.getThreadId) {
        setThread(threadData.getThreadId);
      } else {
        setThread(undefined);
      }
    })
```

Po zakończeniu wywołania funkcji `execGetThreadId` zwracany jest obiekt `threadData`; zapisujemy go w lokalnej właściwości stanu `thread`.

```
    }, [threadData]);

    return (
      <div className="screen-root-container">
        <div className="thread-nav-container">
          <Nav />
        </div>
        <div className="thread-content-container">
```

```

<div className="thread-content-post-container">
  <ThreadHeader
    userName={thread?.user.userName}

```

W tym fragmencie kodu używamy obiektu `thread?.user`, by pobrać wartość pola `userName` i zastąpić nim stosowane wcześniej wyrażenie o postaci `thread?.userName`.

```

    lastModifiedOn={thread ? thread.lastModifiedOn : new Date()}
    title={thread?.title}
  />
  <ThreadCategory category={thread?.category} />

```

Komponent `ThreadCategory` niebawem zmodyfikujemy w taki sposób, by korzystać z właściwości *props* typu `Category`, i by to ona służyła do określania kategorii wybranej w komponencie `CategoryDropDown`. Zajmiemy się nim nieco później.

```

    <ThreadTitle title={thread?.title} />
    <ThreadBody body={thread?.body} readOnly={readOnly} />

```

W tym miejscu przekazujemy wartość właściwości stanu `readOnly` do komponentu `ThreadBody`, gdyż wewnątrz niego jest używany komponent `RichEditor`.

```

  </div>
  <div className="thread-content-points-container">
    <ThreadPointsBar
      points={thread?.points || 0}
      responseCount={
        thread && thread.threadItems && thread.threadItems.length
      }
    />
  </div>
</div>
<div className="thread-content-response-container">
  <hr className="thread-section-divider" />
  <ThreadResponsesBuilder threadItems={thread?.threadItems}
    readOnly={readOnly} />

```

W tym fragmencie kodu przekazujemy wartość właściwości stanu `readOnly` do komponentu `ThreadResponseBuilder`, który renderuje wszystkie odpowiedzi (`ThreadItem`) do wątku.

```

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

```

Pozostała część kodu jest taka sama, jak wcześniej.

11. Kolejnym komponentem, którym się zajmiemy, będzie komponent `ThreadCategory`. Poniżej pokazałem, jak obecnie ma wyglądać jego kod:

```

interface ThreadCategoryProps {
  category?: Category;
}

```

Zmieniliśmy definicję interfejsu i aktualnie zawiera on jedno pole typu `Category`, a nie łańcuch znaków. Dzięki temu obiekt `Category` będziemy mogli przekazać dalej, do komponentu `CategoryDropDown`:

```
const ThreadCategory: FC<ThreadCategoryProps> = ({ category }) => {
  const sendOutSelectedCategory = (cat: Category) => {
    console.log("selected category", cat);
  };

  return (
    <div className="thread-category-container">
      <strong>{category?.name}</strong>
```

W tym fragmencie pobieramy nazwę kategorii z obiektu `Category`, używając do tego celu wyrażenia o postaci `category?.name`, którym zastąpiliśmy stosowaną wcześniej właściwość `categoryName`.

```
      <div style={{ marginTop: "1em" }}>
        <CategoryDropDown
          preselectedCategory={category}
```

W tym fragmencie jawnie przekazujemy do komponentu `CategoryDropDown` właściwość `props` `preselectedCategory`, której wartością jest przekazana do tego komponentu właściwość `props` `category`.

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

12. Teraz musisz poprawić odwołanie do komponentu `RichEditor` umieszczone w kodzie komponentu `ThreadBody`; poniżej pokazałem zmiany, które musisz wprowadzić:

```
interface ThreadBodyProps {
  body?: string;
  readOnly: boolean;
}
```

Jak widać, do typu definiującego właściwości `props`, czyli interfejsu `ThreadBodyProps`, dodaliśmy pole `readOnly`.

```
const ThreadBody: FC<ThreadBodyProps> = ({ body, readOnly }) => {
  return (
    <div className="thread-body-container">
      <strong>Treść</strong>
      <div className="thread-body-editor">
        <RichEditor
          existingBody={body}
          readOnly={readOnly}
        />
      </div>
    </div>
  );
};
```



Jak widać, właściwość `readOnly` przekazujemy do komponentu `RichEditor`.

13. Teraz zajmiemy się komponentem `ThreadResponseBuilder`; poniżej przedstawiłem zmiany, które powinienś w nim wprowadzić:

```
interface ThreadResponsesBuilderProps {
  threadItems?: Array<ThreadItem>;
  readOnly: boolean;
}
```

Także tutaj dodaliśmy do komponentu definicję właściwości *props* `readOnly`; to konieczne, gdyż komponent ten używa komponentu `ThreadResponse`, który z kolei używa `RichEditor`.

```
const ThreadResponsesBuilder: FC<ThreadResponsesBuilderProps> = ({
  threadItems,
  readOnly
}) => {
  const [responseElements, setResponseElements] = useState<
    JSX.Element | undefined
  >();

  useEffect(() => {
    if (threadItems) {
      const thResponses = threadItems.map((ti) => {
        return (
          <li key={`thr-${ti.id}`}>
            <ThreadResponse
              body={ti.body}
              userName={ti.user.userName}
              lastModifiedOn={ti.createdOn}
              points={ti.points}
              readOnly={readOnly}
            />
          </li>
        );
      });
      setResponseElements(<ul>{thResponses}</ul>);
    }
  }, [threadItems, readOnly]);

  return (
    <div className="thread-body-container">
      <strong style={{ marginBottom: ".75em" }}>Odpowiedzi</strong>
      {responseElements}
    </div>
  );
};
```

W tym fragmencie użyliśmy obiektu `user` należącego do wątku (`Thread`), by pobrać nazwę użytkownika (`userName`).

```
      lastModifiedOn={ti.createdOn}
      points={ti.points}
      readOnly={readOnly}
```

Oto nasze pole `readOnly` przekazywane do komponentu `ThreadResponse`.

```
    />
  </li>
);
});
setResponseElements(<ul>{thResponses}</ul>);
}
}, [threadItems, readOnly]);

return (
  <div className="thread-body-container">
    <strong style={{ marginBottom: ".75em" }}>Odpowiedzi</strong>
    {responseElements}
  </div>
);
};
```

Pozostała część kodu nie została zmodyfikowana.

Teraz musimy zająć się komponentem `ThreadResponse`, do którego musimy dodać właściwość `props readOnly`:

```
interface ThreadResponseProps {
  body?: string;
  userName?: string;
  lastModifiedOn?: Date;
  points: number;
  readOnly: boolean;
```

W tym interfejsie dodaliśmy nową właściwość `props`, `readOnly`.

```
}

const ThreadResponse: FC<ThreadResponseProps> = ({
  body,
  userName,
  lastModifiedOn,
  points,
  readOnly,
```

Do przekazania właściwości `props` `readOnly` zastosowaliśmy mechanizm destrukuryzacji.

```
  }) => {
    return (
      <div>
        <div>
          <UserNameAndTime userName={userName} lastModifiedOn={lastModifiedOn} />
          <span style={{ marginLeft: "1em" }}>
            <ThreadPointsInline points={points || 0} />
          </span>
        </div>
        <div className="thread-body-editor">
          <RichEditor existingBody={body} readOnly={readOnly} />
```

Tu przekazujemy wartość `readOnly` do komponentu `RichEditor`.

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

Efekty wprowadzenia tych wszystkich modyfikacji są trudne do zauważenia, gdyż postać interfejsu użytkownika naszej aplikacji nie zmieniła się. Jednak nasz edytor jest już prawie przygotowany do pracy w trybie tylko do odczytu, a końcowe zmiany, niezbędne do tego, by tak działał, wprowadzimy już niebawem; wtedy, gdy wyświetlisz jeden z istniejących wątków (na przykład <http://localhost:3000/thread/1>), edytor treści wątku oraz edytory odpowiedzi nie zezwolą na ich modyfikowanie.

# System punktów

Skoro przygotowaliśmy już wszystko, co potrzebne do wyświetlania punktacji wątków, musimy zaimplementować mechanizm do ich dodawania. Właśnie tym zajmiemy się w bieżącym podrozdziale. A zatem, zaczynamy:

1. Otwórz plik *Thread.tsx* i spójrz na jego kod. Nieco przed końcem kodu JSX znajdziesz komponent o nazwie *ThreadPointsBar*. To właśnie on odpowiada za wyświetlanie pionowego paska w komponencie *ThreadCard* i naszej trasie do wyświetlania wątków (*Thrad.tsx*).
2. Dodamy przyciski, które pozwolą użytkownikom inkrementować lub dekrementować liczbę punktów przypisanych danemu wątkowi. Zaimplementowaliśmy już niezbędne mechanizmy (w tym resolvery) po stronie serwera, więc prace, które nam pozostały do wykonania, będą polegać na powiązaniu tego kodu serwerowego z kodem klienta.

W pliku *ThreadPointsBar.tsx* zmodyfikuj kod JSX zgodnie z poniższym przykładem. Zmiany są znaczące, więc będę je przedstawiał fragment po fragmencie.

```
import React, { FC } from "react";
import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";
import {
  faHeart,
  faReplyAll,
  faChevronDown,
  faChevronUp,
} from "@fortawesome/free-solid-svg-icons";
import { useWindowDimensions } from "../../hooks/useWindowDimensions";
import { gql, useMutation } from "@apollo/client";

const UpdateThreadPoint = gql`
  mutation UpdateThreadPoint(
    $userId: ID!
    $threadId: ID!
    $increment: Boolean!
  ) {
    updateThreadPoint(
      userId: $userId
      threadId: $threadId
      increment: $increment
    )
  }
`;
```

Ten fragment kodu przedstawia nową mutację *updateThreadPoint*.

```
export class ThreadPointsBarProps {
  points: number = 0;
  responseCount?: number;
  threadId?: string;
  allowUpdatePoints?: boolean = false;
  refreshThread?: () => void;
}
```

W tym fragmencie kodu przekształciliśmy interfejs `ThreadPointsBarProps` w klasę, dzięki czemu możemy przypisać niektórym z jej pól wartości domyślne. Zwróć uwagę na to, że oprócz pól klasa ta definiuje także funkcję, `refreshThread`, która będzie służyć do wymuszania aktualizacji nadrzędnego komponentu `Thread`, co z kolei pozwoli zaktualizować interfejs użytkownika po zmianie punktacji wątku. Każde z tych pól opiszę dokładniej nieco później, kiedy będziemy ich używać. Tej właściwości *props* nie będziemy udostępniać komponentowi `ThreadPointsInline`, który przedstawię nieco później.

```
const ThreadPointsBar: FC<ThreadPointsBarProps> = ({
  points,
  responseCount,
  userId,
  threadId,
  allowUpdatePoints,
  refreshThread,
}) => {
  const { width } = useWindowDimensions();
  const [execUpdateThreadPoint] = useMutation(UpdateThreadPoint);
```

Zwróć uwagę na to, że w wywołaniu `useMutation` nie używamy tablicy `refreshQueries`, by odświeżać klienta Apollo. Zazwyczaj skorzystałbym z tego mechanizmu, jednak w przypadku przeprowadzania testów wykryłem, że klient Apollo, który domyślnie przechowuje w pamięci podręcznej wszystkie wykonane zapytania i ich wyniki, nie był w stanie prawidłowo odświeżać danych wątku. Tego rodzaju problemy zdarzają się od czasu do czasu we wszystkich frameworkach. Znajdowanie sposobów obejścia lub innych rozwiązań tego typu problemów, będzie jednym z elementów naszej pracy jako programistów. Dlatego, zamiast bazować na mechanizmie odświeżania korzystającym z tablicy `refreshQueries`, zastosujemy naszą funkcję `refreshThread`, którą możemy pobrać z komponentu nadrzędnego, by wymusić odświeżenie danych wątku. Implementację tej funkcji przedstawię nieco później, przy okazji prezentowania kodu komponentu `Thread`.

```
const onClickIncThreadPoint = async (
  e: React.MouseEvent<SVGSVGElement, MouseEvent>
) => {
  e.preventDefault();
  await execUpdateThreadPoint({
    variables: {
      userId,
      threadId,
      increment: true,
    },
  });
  refreshThread && refreshThread();
};

const onClickDecThreadPoint = async (
  e: React.MouseEvent<SVGSVGElement, MouseEvent>
) => {
  e.preventDefault();
  await execUpdateThreadPoint({
```

```

variables: {
  userId,
  threadId,
  increment: false,
},
});
refreshThread && refreshThread();
};

```

Obie funkcje przedstawione w powyższym fragmencie kodu, `onClickIncThreadPoint` oraz `onClickDecThreadPoint`, najpierw wykonują funkcję `execUpdateThreadPoint`, a następnie funkcję `refreshThread`. Wyrażenie `refreshThread && refreshThread()` jest przykładem sprytnego wykorzystania składni języka JavaScript w celu skrócenia kodu, który trzeba napisać. Składnia ta najpierw sprawdza, czy potencjalnie opcjonalna funkcja istnieje, i jeśli okaże się, że istnieje, to ją wywołuje.

```

if (width > 768) {
  console.log("Komponent ThreadPointsBar liczba punktów", points);
  return (
    <div className="threadcard-points">
      <div className="threadcard-points-item">
        <div
          className="threadcard-points-item-btn"
          style={{ display: `${allowUpdatePoints ? "block" : "none"}` }}
        >

```

W tym fragmencie kodu używamy logiki warunkowej bazującej na właściwości *props* `allowUpdatePoints`, która określa, czy należy wyświetlić, czy też ukryć kontener umożliwiający użytkownikom inkrementację punktów przypisanych wątkowi. Dokładnie w ten sam sposób musimy postąpić z przyciskiem służącym do dekrementacji punktów przypisanych wątkowi:

```

<FontAwesomeIcon
  icon={faChevronUp}
  className="point-icon"
  onClick={onClickIncThreadPoint}
/>
</div>
{points}
<div
  className="threadcard-points-item-btn"
  style={{ display: `${allowUpdatePoints ? "block" : "none"}` }}
>
  <FontAwesomeIcon
    icon={faChevronDown}
    className="point-icon"
    onClick={onClickDecThreadPoint}
  />
</div>
<FontAwesomeIcon icon={faHeart} className="points-icon" />

```

W tym fragmencie kodu dodajemy dwie nowe ikony, `faChevronUp` oraz `faChevronDown`. Po kliknięciu, będą one odpowiednio inkrementować i dekrementować liczbę punktów przypisanych danemu wątkowi.

```

    </div>
    <div className="threadcard-points-item">
      {responseCount}
      <br />
      <FontAwesomeIcon icon={faReplyAll} className="points-icon" />
    </div>
  </div>
);
}
return null;
};

export default ThreadPointsBar;

```

Reszta kodu pozostaje taka sama. Zwróć jednak uwagę na to, że wprowadziłem pewne drobne zmiany do kodu CSS określającego postać komponentu: zmodyfikowałem klasę `threadcard-points-item` i dodałem nową klasę, o nazwie `threadcard-points-item-btn`.

```

.threadcard-points-item {
  display: flex;
  flex-direction: column;
  justify-content: space-between;
  align-items: center;
  color: var(--point-color);
  font-size: var(--sm-med-font-size);
  text-align: center;
}

```

Element, w którym zastosowana zostanie klasa `threadcard-points-item`, będzie teraz wyświetlany jako *flexbox* o układzie kolumny, dzięki czemu jego zawartość będzie rozmieszczana w pionie.

```

.threadcard-points-item-btn {
  cursor: pointer;
  margin-top: 0.35em;
  margin-bottom: 0.35em;
}

```

Klasa `thread-points-item-btn` przekształca domyślny wskaźnik myszki na wskaźnik typu `pointer`; dzięki temu, kiedy użytkownik umieści wskaźnik myszy na ikonie, zmieni on kształt na dłoń, sygnalizując w ten sposób, że ikonę można kliknąć.

- Skoro wprowadziliśmy już te zmiany, musimy zmodyfikować także kilka innych komponentów związanych z systemem punktacji. Pierwszą rzeczą, jaką chcemy zmienić, jest wyłączenie przechowywania wyników w pamięci podręcznej przez klienta Apollo (obiekt `ApolloClient`). Aby wprowadzić tę modyfikację, otwórz plik `index.tsx` i zmień obiekt `client` tak, jak pokazałem na kolejnym przykładzie:

```

const client = new ApolloClient({
  uri: "http://localhost:5000/graphql",
  credentials: "include",
  cache: new InMemoryCache({
    resultCaching: false
  }),
});

```

Zgodnie z tym, co sugeruje nazwa, ustawienie to ma za zadanie wyłączać przechowywanie wyników zapytań w pamięci podręcznej. Jednak zmiana wartości tej jednej opcji nie wystarczy: musimy także zmodyfikować ustawienia w zapytaniach.

4. Zmodyfikuj kod pliku *Thread.tsx*. Poniżej przedstawiłem jedynie kod, który uległ zmianie.

Pierwszą, nieznaczną zmianę, musisz wprowadzić w kodzie zapytania `getThreadById`:

```
const GetThreadById = gql`
  query GetThreadById($id: ID!) {
    getThreadById(id: $id) {
      ... on EntityResult {
        messages
      }

      ... on Thread {
        id
        user {
          id

```

Tego pola będziemy używali w dalszej części kodu, w systemie punktacji, aby sprawdzać, czy użytkownik nie próbuje przyznawać punktów sam sobie.

```
      userName
    }
    lastModifiedOn
    title
    body
    points
    category {
      id
      name
    }
    threadItems {
      id
      body
      points
      user {
        id

```

Także to pole będzie nam potrzebne do sprawdzania, czy użytkownik nie ocenia swoich własnych publikacji.

```
      userName
    }
  }
}
};
```

```
const Thread = () => {
  const [execGetThreadById, { data: threadData }] =
    useLazyQuery(GetThreadById, { fetchPolicy: "no-cache" });
```

W tym fragmencie kodu dodaliśmy do zapytania nową opcję, o nazwie `fetchPolicy`. Kontroluje ona politykę wykorzystania pamięci podręcznej w poszczególnych wywołaniach. W tym przypadku w ogóle nie chcemy, by pamięć podręczna była stosowana. Jeszcze raz powtórzę, że w celu faktycznego wyeliminowania użycia pamięci podręcznej konieczne było zastosowanie obu ustawień: `fetchPolicy` oraz `resultCaching`.

```
const [thread, setThread] = useState<ThreadModel | undefined>();
const { id } = useParams();
const [readOnly, setReadOnly] = useState(false);

const refreshThread = () => {
  if (id && id > 0) {
    execGetThreadById({
      variables: {
        id,
      },
    });
  }
};
```

W tym fragmencie zdefiniowaliśmy funkcję `refreshThread`, wewnątrz której wywoływana jest funkcja `execGetThreadById`. W dalszej części kodu będziemy przekazywać tę funkcję do komponentu `ThreadPointBar`.

```
useEffect(() => {
  if (id && id > 0) {
    execGetThreadById({
      variables: {
        id,
      },
    });
  }
}, [id, execGetThreadById]);
```

Zastanawiasz się zapewne, dlaczego nie wykorzystaliśmy funkcji `refreshThread` w wywołaniu `useEffect`. Aby to zrobić, musielibyśmy umieścić `refreshThread` na liście w wywołaniu `useEffect`, jak również dodatkowo wywołać funkcję `useCallback`, by zmiany wyniku `refreshThread` nie powodowały ponownego wyrenderowania komponentu. Zatem te nieznaczne korzyści, jakie zapewniłoby wykorzystanie naszej nowej funkcji, nie równoważyłyby całego dodatkowego kodu, który musielibyśmy napisać.

```
useEffect(() => {
  if (threadData && threadData.getThreadById) {
    setThread(threadData.getThreadById);
    setReadOnly(true);
  } else {
    setThread(undefined);
    setReadOnly(false);
  }
}, [threadData]);

return (
```



```

<div className="screen-root-container">
  <div className="thread-nav-container">
    <Nav />
  </div>
  <div className="thread-content-container">
    <div className="thread-content-post-container">
      <ThreadHeader
        userName={thread?.user.userName}
        lastModifiedOn={thread ? thread.lastModifiedOn : new Date()}
        title={thread?.title}
      />
      <ThreadCategory category={thread?.category} />
      <ThreadTitle title={thread?.title} />
      <ThreadBody body={thread?.body} readOnly={readOnly} />
    </div>
    <div className="thread-content-points-container">

```

W kolejnym fragmencie, do komponentu ThreadPointsBar przekazujemy nowe, zdefiniowane wcześniej właściwości *props*:

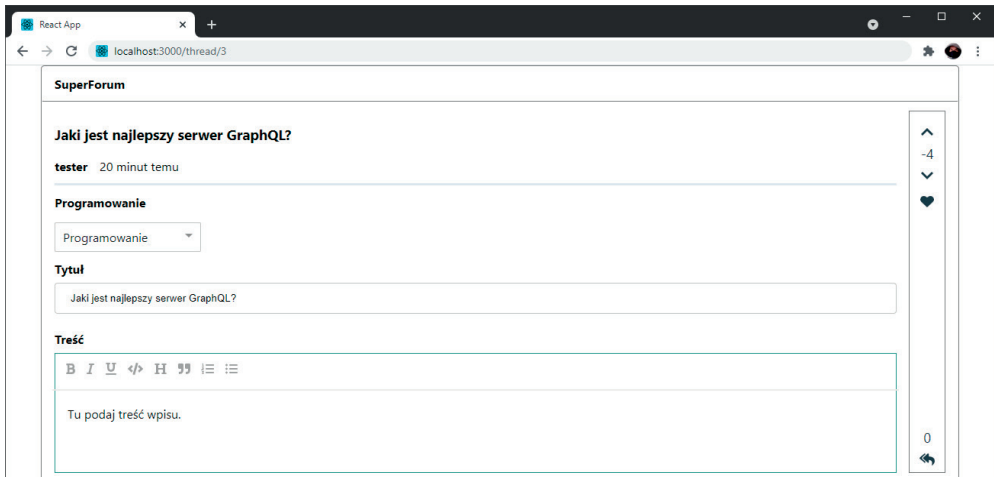
```

      <ThreadPointsBar
        points={thread?.points || 0}
        responseCount={
          (thread && thread.threadItems && thread.threadItems.length) || 0
        }
        userId={thread?.user.id}
        threadId={thread?.id || "0"}
        allowUpdatePoints={true}
        refreshThread={refreshThread}
      />
    </div>
  </div>
  <div className="thread-content-response-container">
    <hr className="thread-section-divider" />
    <ThreadResponsesBuilder threadItems={thread?.threadItems} readOnly=
      {readOnly} />
  </div>
</div>
);
};

```

5. Poniżej, na rysunku 16.1, pokazałem, jak wygląda ekran trasy wątku po uruchomieniu systemu punktacji.

Jeśli teraz spróbujesz kliknąć któryś z przycisków zmieniających punktację, zauważysz dwie rzeczy. Pierwszą z nich jest to, że czasami liczba punktów widoczna na ekranie nie będzie aktualizowana natychmiast, i to bez względu na całą pracę, jaką wykonaliśmy w celu wyeliminowania przechowywania wyników zapytań w pamięci podręcznej. Powodem tego problemu jest subtelny błąd występujący w funkcjach warstwy repozytorium, którym niebawem się zajmujemy. Drugi problem polega na tym, że użytkownik może dodać lub odjąć więcej punktów za jednym razem. To inny problem, związany ze stylami. Oba te problemy rozwiążemy, kiedy kod naszej klienckiej części aplikacji będzie już gotowy.



Rysunek 16.1. Ekran trasy wątku

6. Teraz musimy zaimplementować obsługę punktów w komponentach związanych z odpowiedziami. Zaczniemy od komponentu `ThreadResponse` ➔ `Builder`. Zmodyfikuj w nim wywołanie `useEffect` tak, jak pokazałem na poniższym przykładzie:

```
useEffect(() => {
  if (threadItems) {
    const thResponses = threadItems.map((ti) => {
      return (
        <li key={`thr-${ti.id}`}>
          <ThreadResponse
            body={ti.body}
            userName={ti.user.userName}
            lastModifiedOn={ti.createdOn}
            points={ti.points}
            readOnly={readOnly}
            userId={ti?.user.id || "0"}
            threadItemId={ti?.id || "0"}
          />
        </li>
      );
    });
    setResponseElements(<ul>{thResponses}</ul>);
  }
}, [threadItems, readOnly]);
```

W tym fragmencie kodu, do komponentu `ThreadResponse`, który prezentuje encję `ThreadItem`, przekazujemy dodatkowo identyfikator użytkownika (`userId`) oraz identyfikator odpowiedzi (`threadItemId`). Z kolei w tym komponencie używamy komponentu `ThreadPointsInline`, który prezentuje liczbę polubień wątku (`Thread`) lub odpowiedzi (`ThreadItem`), zależnie od tego, co zostało do niego przekazane. Działanie tego komponentu wyjaśnię nieco później, kiedy do niego dojdziemy:

```
</li>
);
});
setResponseElements(<ul>{thResponses}</ul>);
}
}, [threadItems, readOnly]);
```

7. Teraz musimy zająć się komponentem `ThreadResponse`. Poniżej przedstawię jedynie te jego fragmenty, które trzeba zmienić.

W pierwszej kolejności dodaj te dwa pola do interfejsu `ThreadResponseProps`:

```
userId: string;
threadItemId: string;
```

Następnie dodaj do kodu JSX pola `userId` i `threadItemId`:

```
return (
  <div>
    <div>
      <UserNameAndTime userName={userName} lastModifiedOn={lastModifiedOn} />
      {threadItemId}
      <span style={{ marginLeft: "1em" }}>
        <ThreadPointsInline
          points={points || 0}
          userId={userId}
          threadItemId={threadItemId}
        />
      </span>
    </div>
  </div>
```

W tym fragmencie kodu przekazujemy dane `userId` i `threadItemId` do komponentu `ThreadPointsInline`. Zwróć uwagę na to, że docelowo ten komponent ma prezentować punktację dla wątków (`Thread`) oraz odpowiedzi (`ThreadItem`). Oprócz tego zauważ, że `threadItemId` przekazujemy do komponentu tylko tymczasowo, po to, byśmy mogli rozróżniać poszczególne odpowiedzi (`ThreadItem`).

```
    </span>
  </div>
  <div className="thread-body-editor">
    <RichEditor existingBody={body} readOnly={readOnly} />
  </div>
</div>
);
```

8. Teraz przyjrzyjmy się zmianom, które musimy wprowadzić w komponencie `ThreadPointsInline`.

Dodaj poniższą instrukcję do listy pozostałych instrukcji importu:

```
import "./ThreadPointsInline.css";
```

A teraz przyjrzyj się kodowi tego arkusza stylów. W znacznej mierze przypomina on style stosowane w komponencie `ThreadPointsBar`.

```
const UpdateThreadItemPoint = gql`
mutation UpdateThreadItemPoint(
  $userId: ID!
  $threadItemId: ID!,
  $increment: Boolean!
) {
  updateThreadItemPoint(
    userId: $userId,
    threadItemId: $threadItemId,
    increment: $increment
  )
}
```

```

    )
  }
};

```

W tym fragmencie kodu dodaliśmy definicję mutacji `updateThreadItemPoint`.

```

class ThreadPointsInlineProps {
  points: number = 0;
  userId?: string;
  threadId?: string;
  threadItemId?: string;
  allowUpdatePoints?: boolean = false;
  refreshThread?: () => void;
}

```

Ta klasa definiuje postać listy właściwości *props* komponentu. Zwróć uwagę na to, że zdefiniowaliśmy w niej pole `threadId`. Będziemy używali komponentu `ThreadPointsInline` do wyświetlania punktów dla wątków, w przypadku, gdy aplikacja zostanie uruchomiona na urządzeniu mobilnym.

```

const ThreadPointsInline: FC<ThreadPointsInlineProps> = ({
  points,
  userId,
  threadId,
  threadItemId,
  allowUpdatePoints,
  refreshThread,
}) => {
  const [execUpdateThreadItemPoint] = useMutation(UpdateThreadItemPoint);
  const onClickIncThreadItemPoint = async (
    e: React.MouseEvent<SVGSVGElement, MouseEvent>
  ) => {
    e.preventDefault();

    await execUpdateThreadItemPoint({
      variables: {
        threadItemId,
        increment: true,
      },
    });
    refreshThread && refreshThread();
  };
};

```

W tym fragmencie kodu nie ma niczego szczególnego: obie procedury obsługi zdarzeń, `onClickIncThreadItemPoint` oraz `onClickDecThreadItemPoint` działają bardzo podobnie, czyli wykonują mutację, po czym odświeżają dane wątku:

```

const onClickDecThreadItemPoint = async (
  e: React.MouseEvent<SVGSVGElement, MouseEvent>
) => {
  e.preventDefault();

  await execUpdateThreadItemPoint({
    variables: {
      threadItemId,
      increment: false,
    },
  });
};

```

```

    },
  });
  refreshThread && refreshThread();
};

```

A teraz, w kodzie JSX, zrobimy coś podobnego do rozwiązania zastosowanego w komponencie `ThreadPointsBar`: dodamy ikony pozwalające na inkrementację i dekrementację liczby punktów przypisanych prezentowanej encji:

```

return (
  <span className="threadpointsinline-item">
    <div
      className="threadpointsinline-item-btn"
      style={{ display: `${allowUpdatePoints ? "block" : "none"}` }}
    >
      <FontAwesomeIcon
        icon={faChevronUp}
        className="point-icon"
        onClick={onClickIncThreadItemPoint}
      />
    </div>
    {points}
    <div
      className="threadpointsinline-item-btn"
      style={{ display: `${allowUpdatePoints ? "block" : "none"}` }}
    >
      <FontAwesomeIcon
        icon={faChevronDown}
        className="point-icon"
        onClick={onClickDecThreadItemPoint}
      />
    </div>
    <div className="threadpointsinline-item-btn">
      <FontAwesomeIcon icon={faHeart} className="points-icon" />
    </div>
  </span>
);
};

export default ThreadPointsInline;

```

9. Jeśli teraz wyświetlisz ekran wątku, powinieneś zobaczyć na nim także odpowiedzi do aktualnie wybranego wątku. Pamiętaj, że Twoje lokalne dane będą zapewne inne, więc powinieneś zadbać, by z wybranym wątkiem (`Thread`) były powiązane odpowiedzi (`ThreadItem`) oraz punkty. Wygląd odpowiedzi wraz z ich punktacją przedstawiłem na rysunku 16.2.

Jeśli teraz klikniesz ikony do inkrementacji lub dekrementacji punktacji odpowiedzi, to zauważysz, że występują te same problemy, co w przypadku punktacji wątków — prezentowana punktacja nie zawsze jest aktualizowana i użytkownik może wielokrotnie dodawać i usuwać punkty. Teraz zajmiemy się oboma tymi problemami

## Odpowiedzi

tester2 3 ^ 7 ▼ ♥

B I U &lt;/&gt; H ¶ ☰ ☷

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

tester2 4 ^ 2 ▼ ♥

B I U &lt;/&gt; H ¶ ☰ ☷

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

tester2 5 ^ 4 ▼ ♥

B I U &lt;/&gt; H ¶ ☰ ☷

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Rysunek 16.2. Odpowiedzi i ich punktacja

- Przejdź do serwerowej części projektu, otwórz plik *ThreadItemPointRepo.ts*, odśledź w nim funkcję `updateThreadItemPoint`, a w niej pierwsze wywołanie `threadItem.save()`. Poprzedź to wywołanie słowem kluczowym `await`, jak pokazałem na poniższym przykładzie; następnie w taki sam sposób popraw wszystkie pozostałe wywołania funkcji `save`.

```
await threadItem.save();
```

Czy potrafisz wyjaśnić, dlaczego ta zmiana rozwiązała nasz problem? Otóż dzięki zastosowaniu słowa kluczowego `await` wymuszamy zaczekanie na zakończenie wywołania funkcji `save`. Później, kiedy uzyskamy dane encji `ThreadItem`, możemy mieć pewność, że będą one zawierać nową wartość pola `points`. To jeden z bardziej problematycznych aspektów programowania asynchronicznego. Choć zapewnia ono szybsze działanie aplikacji, to jednak zmusza do myślenia o tym, co się robi; w przeciwnym razie mogą się pojawiać problemy takie jak ten.

Teraz samodzielnie popraw kod funkcji `updateThreadPoint`, wprowadzając w nim takie same zmiany, jakie wprowadziłeś wcześniej, w kodzie funkcji `updateThreadItem` ➔ `Point`. Upewnij się, że poprawiłeś każde wywołanie funkcji `save`.

Po wprowadzeniu tych zmian, kiedy spróbujesz inkrementować i dekrementować punktację, przekonasz się, że prezentowane dane są aktualizowane prawidłowo.

- Teraz zajmiemy się możliwością wielokrotnego dodawania i usuwania punktów. Okazuje się, że w tej ścieżce kodu występuje kilka problemów. Nasze dwa resolwery odpowiedzialne za aktualizację punktów, `updateThreadPoint` oraz `updateThreadItemPoint`, nie uwierzytelniają użytkownika przed próbą umożliwienia mu zmiany punktacji. Z oczywistych powodów takie działanie jest nieprawidłowe. Co więcej, kod klienckiej części aplikacji przekazuje identyfikator użytkownika (`userId`) pobrany z encji `Thread` lub `ThreadItem`,

a nie identyfikator aktualnie zalogowanego użytkownika. Obie te usterki możemy poprawić za jednym razem. Najpierw zajmiemy się resolverem `updateThreadPoint`; poniżej pokazałem zmiany, które należy w nim wprowadzić:

```
updateThreadPoint: async (
  obj: any,
  args: { threadId: string; increment: boolean },
  ctx: GqlContext,
  info: any
): Promise<string> => {
```

Jak widać, aktualnie nie pobieramy już identyfikatora użytkownika (`userId`) jako parametru tego resolvera. Nie jest on już potrzebny, gdyż — jak pokazałem na poniższym fragmencie kodu — użytkownika będziemy sprawdzać używając identyfikatora zapisanego w sesji: `session.userId`. To samo pole `session.userId` prześlemy następnie w wywołaniu funkcji `updateThreadPoint` warstwy repozytorium, jako parametr `userId`:

```
let result = "";
try {
  if (!ctx.req.session || !ctx.req.session?.userId) {
    return "Musisz być zalogowany, by ustawiać polubienia.";
  }
  result = await updateThreadPoint(
    ctx.req.session!.userId,
    args.threadId,
    args.increment
  );
  return result;
} catch (ex) {
  throw ex;
}
},
```

Analogiczne zmiany wprowadź także w kodzie resolvera `updateThreadItemPoint`, gdyż mają one działać w niemal identyczny sposób. Nie zapomnij także zmodyfikować pliku definicji typów (*typeDefs.ts*) tak, by sygnatury tych mutacji nie zawierały już parametru `userId`. Konieczne będzie także zaktualizowanie tych ścieżek realizacji w klienckiej części aplikacji i usunięcie z nich parametru `userId`; zajmiemy się tym nieco później.

12. Teraz dodaj poniższy fragment do funkcji `updateThreadPoint` warstwy repozytorium, na samym początku jej kodu:

```
if (!userId || userId === "0") {
  return "Użytkownik nie został uwierzytelniony";
}
```

Ten fragment zapobiegne zapisywaniu w bazie danych wszelkich dziwnych wartości `userId`, a nam pozwoli mieć pewność, że użytkownik modyfikujący punktację jest zalogowany. Ten sam fragment kodu dodaj do funkcji `updateThreadItemPoint` warstwy repozytorium.

Teraz zajmiemy się poprawieniem klienckiej części aplikacji i usunięciem z niej parametru `userId`. Najprostszy sposób, by to zrobić, jest usunięcie tego parametru z wywołań komponentów `ThreadPointsBar` i `ThreadPointsInline`. Jeśli to zrobimy, to kompilator wyświetli błędy informujące nas o tym, w jakich innych, powiązanych wywołaniach jest używany identyfikator użytkownika.

13. Zaczniemy od komponentu `ThreadPointsBar`. Zaktualizuj go zgodnie z poniższym opisem. Najpierw usuń `userId` z parametrów mutacji `updateThreadPoint`. Następnie usuń pole `userId` z klasy `ThreadPointBarProps` określającej właściwości `props` komponentu. W kolejnym kroku usuń parametr `userId` z listy właściwości `props` komponentu `ThreadPointBarProps`. I w końcu usuń go z wywołań funkcji `execUpdateThreadPoints`.

14. Następnie w komponencie `Thread.tsx` odszukaj komponent `ThreadPointsBar` użyty w kodzie JSX i usuń z niego właściwość `props` `userId`. Dodatkowo usuń wywołanie `useSelector` pobierające reduktor użytkownika, gdyż nie jest nam już potrzebne.

Analogiczne zmiany trzeba wprowadzić w komponencie `ThreadPointsInline`, jednak pozostawię ich wykonanie Tobie, gdyż w zasadzie są one powtórzeniem zmian wykonanych w kodzie komponentu `ThreadPointsBar`. Także w tym przypadku spróbuj wprowadzać zmiany do komponentu `ThreadPointsInline`, a następnie zapisz kod. Kompilator powinien Ci odpowiedzieć, gdzie znajdują się odwołania do `userId`.

Po wprowadzeniu tych zmian aktualizacja punktów powinna już przebiegać bezproblemowo: dodawanie i usuwanie punktów będzie możliwe wyłącznie wtedy, gdy użytkownik będzie zalogowany, a oprócz tego będzie można dodać lub odebrać tylko jeden punkt. Ponadto użytkownicy nie będą już mogli zmieniać punktacji swoich własnych wątków i odpowiedzi.

A teraz przyjrzymy się kolejnemu problemowi. Kiedy wyświetlimy ekran wątku w trybie prezentacji dla urządzeń mobilnych, zauważymy, że liczba punktów nie jest widoczna (patrz rysunek 16.3).

Oczywiście, ukrycie paska punktacji w tym przypadku jest celowe, ze względu na niewielką ilość miejsca dostępnego w poziomie. Dlatego teraz dodamy do tego ekranu komponent `ThreadPointsInline` i zmodyfikujemy go w taki sposób, by mógł on współpracować zarówno z wątkami (`Thread`), jak i odpowiedziami (`ThreadItem`):

15. Ze względu na to, że wcześniej zmodyfikowaliśmy komponent `ThreadPointsInline` i zastosowaliśmy w nim mutację `updateThreadPoint` używaną przez komponent `ThreadPointsBar`, zatem teraz musimy przenieść te wywołania do odrębnego *hooka*, byśmy mogli używać ich w różnych miejscach naszej aplikacji. A zatem, w katalogu `hooks` utwórz nowy plik, o nazwie `useUpdateThreadPoint.ts`, i skopiuj do niego kod z analogicznego pliku z kodów źródłowych dołączonych do książki.





Rysunek 16.3. Ekran wątku wyświetlony w trybie dla urządzeń mobilnych

W efekcie skopiowaliśmy do tego nowego pliku znaczną część kodu komponentu `ThreadPointsBar`. Zdefiniowana w tym pliku funkcja `useUpdateThreadPoint` zwraca dwie procedury obsługi zdarzeń, `onClickIncThreadPoint` oraz `onClickDecThread` ↪ `Point`, które mogą być używane przez komponent wywołujący.

16. Teraz zajmiemy się refaktoryzacją komponentu `ThreadPointBar` i zastosowaniem w nim naszego nowego *hooka*. Zaktualizuj komponent tak, jak pokazałem na poniższym przykładzie:

```
import useUpdateThreadPoint from "../../hooks/useUpdateThreadPoint";
```

W tym fragmencie kodu zaimportowaliśmy nowy *hook* i usunęliśmy z pliku mutację `UpdateThreadPoint`.

```
export class ThreadPointsBarProps {
  points: number = 0;
  responseCount?: number;
  threadId?: string;
  allowUpdatePoints?: boolean = false;
  refreshThread?: () => void;
}

const ThreadPointsBar: FC<ThreadPointsBarProps> = ({
  points,
  responseCount,
  threadId,
  allowUpdatePoints,
```

```

        refreshThread,
    }) => {
        const { width } = useWindowDimensions();
        const { onClickDecThreadPoint, onClickIncThreadPoint } = useUpdate
        ↳ThreadPoint(
            refreshThread,
            threadId
        );
    };

```

W tym fragmencie kodu pobieramy procedury obsługi zdarzeń z funkcji `useUpdateThreadPoint`. Z pozostałej części kodu musisz teraz usunąć definicje funkcji `onClickIncThreadPoint` oraz `onClickDecThreadPoint`; natomiast kod JSX komponentu nie ulega zmianom.

17. Kolejnym krokiem będzie modyfikacja komponentu `ThreadPointsInline`; poniżej opisałem, co należy w nim zmienić:

```

import React, { FC } from "react";
import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";
import {
    faHeart,
    faChevronDown,
    faChevronUp,
} from "@fortawesome/free-solid-svg-icons";
import { gql, useMutation } from "@apollo/client";
import "../ThreadPointsInline.css";
import useUpdateThreadPoint from "../../hooks/useUpdateThreadPoint";

const UpdateThreadItemPoint = gql`
    mutation UpdateThreadItemPoint($threadItemId: ID!, $increment: Boolean!) {
        updateThreadItemPoint(threadItemId: $threadItemId, increment: $increment)
    }
`;

class ThreadPointsInlineProps {
    points: number = 0;
    threadId?: string;
    threadItemId?: string;
    allowUpdatePoints?: boolean = false;
    refreshThread?: () => void;
}

const ThreadPointsInline: FC<ThreadPointsInlineProps> = ({
    points,
    threadId,
    threadItemId,
    allowUpdatePoints,
    refreshThread,
}) => {
    const [execUpdateThreadItemPoint] = useMutation(UpdateThreadItemPoint);
    const { onClickDecThreadPoint, onClickIncThreadPoint } = useUpdate
    ↳ThreadPoint(
        refreshThread,
        threadId
    );

```

Kluczową zmianą w tym fragmencie kodu jest pobranie procedur obsługi zdarzeń z *hooka* `useUpdateThreadPoint`.

```
const onClickIncThreadItemPoint = async (
  e: React.MouseEvent<SVGSVGElement, MouseEvent>
) => {
  e.preventDefault();

  await execUpdateThreadItemPoint({
    variables: {
      threadItemId,
      increment: true,
    },
  });
  refreshThread && refreshThread();
};

const onClickDecThreadItemPoint = async (
  e: React.MouseEvent<SVGSVGElement, MouseEvent>
) => {
  e.preventDefault();

  await execUpdateThreadItemPoint({
    variables: {
      threadItemId,
      increment: false,
    },
  });
  refreshThread && refreshThread();
};

return (
  <span className="threadpointsinline-item">
    <div
      className="threadpointsinline-item-btn"
      style={{ display: `${allowUpdatePoints ? "block" : "none"}` }}
    >
      <FontAwesomeIcon
        icon={faChevronUp}
        className="point-icon"
        onClick={threadId ? onClickIncThreadPoint :
onClickIncThreadItemPoint}
      />
    </div>
    {points}
    <div
      className="threadpointsinline-item-btn"
      style={{ display: `${allowUpdatePoints ? "block" : "none"}` }}
    >
      <FontAwesomeIcon
        icon={faChevronDown}
```

W tym fragmencie kodu wstawiliśmy bardzo prostą logikę warunkową, określającą, czy będzie aktualizowana punktacja wątku (`Thread`), czy też odpowiedzi (`ThreadItem`).

```
</div>
{points}
<div
  className="threadpointsinline-item-btn"
  style={{ display: `${allowUpdatePoints ? "block" : "none"}` }}
>
  <FontAwesomeIcon
    icon={faChevronDown}
```

```

        className="point-icon"
        onClick={threadId ? onClickDecThreadPoint :
onClickDecThreadItemPoint}

```

Podobnej logiki warunkowej użyliśmy w tym fragmencie kodu.

```

        />
    </div>
    <div className="threadpointsinline-item-btn">
        <FontAwesome icon={faHeart} className="points-icon" />
    </div>
</span>
);
};

```

```
export default ThreadPointsInline;
```

18. Pozostało nam jeszcze wprowadzenie kilku końcowych zmian w kodzie pliku *Thread.tsx*. Opisałem je poniżej; przy czym przedstawiłem jedynie te fragmenty, które należy zmienić:

```

const Thread = () => {
    const { width } = useWindowDimensions();

```

Na samym początku funkcji *Thread* pobieramy szerokość ekranu urządzenia, na którym działa aplikacja; użyjemy jej do określenia, kiedy należy wyświetlać komponent *ThreadPointsInline*:

```

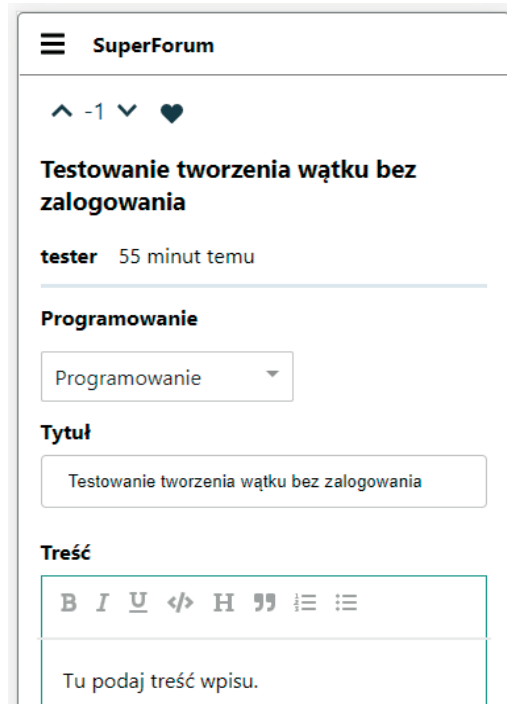
return (
    <div className="screen-root-container">
        <div className="thread-nav-container">
            <Nav />
        </div>
        <div className="thread-content-container">
            <div className="thread-content-post-container">
                {width <= 768 && thread ? (
                    <ThreadPointsInline
                        points={thread?.points || 0}
                        threadId={thread?.id}
                        refreshThread={refreshThread}
                        allowUpdatePoints={true}
                    />
                ) : null}

                <ThreadHeader
                    userName={thread?.user.userName}
                    lastModifiedOn={thread ? thread.lastModifiedOn : new Date()}
                    title={thread?.title}
                />
            </div>
        </div>
    </div>
)

```

W tym fragmencie dodaliśmy komponent *ThreadPointsInline*, który będzie wyświetlany wyłącznie na urządzeniach mobilnych. Nie zapomnij także dodać do pliku *Thread.tsx* niezbędnych instrukcji importu.

Jeśli teraz wyświetlisz ekran wątku na urządzeniu mobilnym, będzie on podobny do tego przedstawionego na rysunku 16.4.



**Rysunek 16.4.** Ekran wątku w trybie dla urządzenia mobilnego, z widocznymi przyciskami do zmiany punktacji

Oprócz tego wprowadziłem także pewne zmiany do komponentu `ThreadCategory`, dzięki którym będzie można wyświetlać go na stronie głównej, w przypadku, gdy aplikacja zostanie uruchomiona na urządzeniu mobilnym.

Teraz możemy już przeglądać istniejące wątki na ekranie. Musimy jednak zaimplementować możliwość dodawania nowych wątków, jak również odpowiadania na nie. Dodatkowo tych dwóch możliwości będzie kolejnym etapem naszych prac:

19. Musimy zacząć od wprowadzenia drobnej zmiany w funkcji `createRepository` warstwy repozytorium. Otwórz plik *ThreadRepo.tsx* i odśledź w nim instrukcję `return` kończącą działanie funkcji `createThread`, która aktualnie ma następującą postać:

```
return { messages: ["Wątek został pomyślnie utworzony."] };
```

Zmień tę instrukcję w następujący sposób:

```
return { messages: [thread.id] };
```

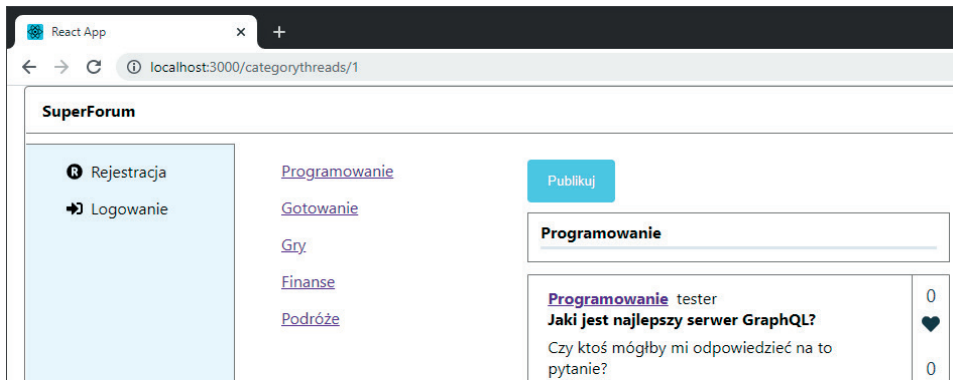
Po wprowadzeniu tej zmiany, jeśli wywołanie funkcji `createThread` zakończy się pomyślnie, to funkcja ta zwróci identyfikator nowego wątku. To rozwiązanie zmniejsza wielość przekazywanych danych, a jednocześnie przekazuje do klienta wszystkie potrzebne mu informacje.

20. Następnie musimy wprowadzić drobną zmianę w samej trasie Thread. Otwórz plik `App.tsx` i odszukaj w nim odpowiednią trasę; następnie zmień ją zgodnie z poniższym przykładem:

```
<Route path="/thread/:id?" render={renderThread} />
```

Jak widać, ta zmiana jest bardzo nieznaczna: sprowadza się do dodania znaku zapytania (?) za parametrem `id`. Jednak zapewni nam ona możliwość odwołania się do tej trasy bez podawania identyfikatora wątku, a jego brak będzie dla aplikacji informacją, że chcemy utworzyć nowy wątek.

21. Teraz dodamy na ekranie głównym przycisk *Publikuj* (patrz rysunek 16.5).



Rysunek 16.5. Nowy przycisk Publikuj

Poniżej przedstawiłem zmodyfikowany kod pliku `Main.tsx`:

```
const Main = () => {
  const [
    execGetThreadsByCat,
    {
      // error: threadsByCatErr,
      // called: threadsByCatCalled,
      data: threadsByCatData,
    },
  ] = useLazyQuery(GetThreadsByCategoryId);
  const [
    execGetThreadsLatest,
    {
      // error: threadsLatestErr,
      // called: threadsLatestCalled,
      data: threadsLatestData,
    },
  ] = useLazyQuery(GetThreadsLatest);

  const { categoryId } = useParams();
  const [category, setCategory] = useState<Category | undefined>();
  const [threadCards, setThreadCards] = useState<Array<JSX.Element> | null>(
    null
  );
  const history = useHistory();
```

W tym fragmencie kodu niewiele się zmieniło — zastosowaliśmy jedynie funkcję `useHistory`, dzięki której będziemy mogli zmieniać adres URL aktualnie prezentowanego ekranu.

```
useEffect(() => {
  if (categoryId && categoryId > 0) {
    execGetThreadsByCat({
      variables: {
        categoryId,
      },
    });
  } else {
    execGetThreadsLatest();
  }
  // eslint-disable-next-line react-hooks/exhaustive-deps
}, [categoryId]);

useEffect(() => {
  console.log("Komponent Main, threadsByCatData", threadsByCatData);
  if (
    threadsByCatData &&
    threadsByCatData.getThreadsByCategoryId &&
    threadsByCatData.getThreadsByCategoryId.threads
  ) {
    const threads = threadsByCatData.getThreadsByCategoryId.threads;
    const cards = threads.map((th: any) => {
      return <ThreadCard key={`thread-${th.id}`} thread={th} />;
    });
    setCategory(threads[0].category);
    setThreadCards(cards);
  } else {
    setCategory(undefined);
    setThreadCards(null);
  }
}, [threadsByCatData]);

useEffect(() => {
  if (
    threadsLatestData &&
    threadsLatestData.getThreadsLatest &&
    threadsLatestData.getThreadsLatest.threads
  ) {
    const threads = threadsLatestData.getThreadsLatest.threads;
    const cards = threads.map((th: any) => {
      return <ThreadCard key={`thread-${th.id}`} thread={th} />;
    });
    setCategory(new Category("0", "Najnowsze"));
    setThreadCards(cards);
  }
}, [threadsLatestData]);

const onClickPostThread = () => {
  history.push("/thread");
};
```

W tym fragmencie kodu zdefiniowaliśmy nową procedurę obsługi zdarzeń, która będzie obsługiwać kliknięcia nowego przycisku. Jej działanie polega na przekierowaniu użytkownika na ekran wątku, bez określania jego identyfikatora. Już niebawem pokażę, dlaczego to rozwiązanie jest ważne.

```
return (
  <main className="content">
    <button className="action-btn" onClick={onClickPostThread}>
      Publikuj
    </button>
```

W tym kodzie JSX jest umieszczony nasz nowy przycisk.

```
    <MainHeader category={category} />
    <div>{threadCards}</div>
  </main>
);
};
```

Reszta kodu nie ulega zmianie.

22. Teraz musimy zmodyfikować komponent Thread w taki sposób, by w przypadku braku przekazania identyfikatora wątku, komponent wiedział, że powinien się przygotować na dodanie nowego. Jednak aby to zrobić, najpierw musimy zmodyfikować parę jego komponentów podrzędnych. Zaczniemy od komponentu RichEditor. Zmodyfikuj go zgodnie z przedstawionymi przykładami; zwróć uwagę na to, że przedstawiłem na nich jedynie te fragmenty komponentu, które trzeba zmienić:

```
export const getTextFromNodes = (nodes: Node[]) => {
  return nodes.map((n: Node) => Node.string(n)).join("\n");
};
```

getTextFromNodes to nowa funkcja pomocnicza, która pozwoli przekształcać tablicę elementów Node używaną przez bibliotekę Slate.js na łańcuch znaków.

Slate.js pozwala na wykonywanie złożonych operacji formatowania na łańcuchach przekazywanych przez użytkownika. Te informacje są bardzo złożone i nie można ich zapisywać w formie zwyczajnego tekstu. Dlatego też Slate.js do przechowywania sformatowanego tekstu używa obiektów bazujących na typie Node. Kiedy takie węzły danych trzeba zapisać w bazie danych, będziemy je najpierw zapisywać w formacie JSON. I to właśnie do tego celu służy przedstawiona właśnie funkcja getTextFromNodes.

```
const HOTKEYS: { [keyName: string]: string } = {
  "mod+b": "bold",
  "mod+i": "italic",
  "mod+u": "underline",
  "mod+`": "code",
};
const initialValue = [
  {
    type: "paragraph",
    children: [{ text: "" }],
```



Teraz wartością początkową edytora (określaną przez stałą `initialValue`) będzie pusty łańcuch znaków.

```
    },
  ];
  const LIST_TYPES = ["numbered-list", "bulleted-list"];

  class RichEditorProps {
    existingBody?: string;
    readOnly?: boolean = false;
    sendOutBody?: (body: Node[]) => void;
```

Do komponentu dodaliśmy nową właściwość *props*, dzięki której, w przypadku zmiany zawartości edytora, nowy umieszczony w nim tekst będzie mógł zostać przekazany w górę hierarchii — do komponentu `Thread`. Komponent `Thread` musi znać najnowszą wartość, by móc ją przekazać jako parametr podczas próby utworzenia nowego wątku. Ten sam wzorec z właściwością *props* `sendOut` będziemy powtarzać także w innych komponentach podrzędnych.

```
  }

  const RichEditor: FC<RichEditorProps> = ({
    existingBody,
    readOnly,
    sendOutBody
  }) => {
    const [value, setValue] = useState<Node[]>(initialValue);
    const renderElement = useCallback((props) => <Element {...props} />, []);
    const renderLeaf = useCallback((props) => <Leaf {...props} />, []);
    const editor = useMemo(() => withHistory(withReact(createEditor())), []);

    useEffect(() => {
      console.log("existingBody", existingBody);
      if (existingBody) {
        setValue(JSON.parse(existingBody));
      }
    });
```

Właściwość *props* `existingBody` to wartość początkowa przekazywana do edytora przez komponent nadrzędny. Ta wartość faktycznie będzie przekazywana w przypadkach, gdy ekran trasy `Thread` zostanie wyświetlony dla jakiegoś istniejącego wątku. Oczywiście, dane tego wątku zostaną pobrane z bazy, a to oznacza, że dane tekstowe zostaną zapisane w bazie w formie łańcucha znaków. To konieczne, gdyż `Postgres` nie rozumie ani nie potrafi obsługiwać typów `Node` używanych przez bibliotekę `Slate.js`. Efektem ubocznym tego faktu jest to, że zanim dane zostaną przekazane do funkcji `setValue`, najpierw będą musiały zostać przekształcone z kodu `JSON` na obiekty; to właśnie dlatego wywołanie tej funkcji ma postać: `setValue(JSON.parse(existingBody))`.

```
  }
  // eslint-disable-next-line react-hooks/exhaustive-deps
}, [existingBody]);

const onChangeEditorValue = (val: Node[]) => {
  setValue(val);
  sendOutBody && sendOutBody(val);
```

W tym fragmencie kodu zapisujemy wartość z edytora (`val`), lecz oprócz tego przekazujemy ją do komponentu nadrzędnego, wywołując w tym celu funkcję `sendOutBody`.

```
};

return (
  <Slate editor={editor} value={value} onChange={onChangeEditorValue}>
    <Toolbar>
      <MarkButton format="bold" icon="bold" />
      <MarkButton format="italic" icon="italic" />
      <MarkButton format="underline" icon="underlined" />
      <MarkButton format="code" icon="code" />
      <BlockButton format="heading-one" icon="header1" />
      <BlockButton format="block-quote" icon="in_quotes" />
      <BlockButton format="numbered-list" icon="list_numbered" />
      <BlockButton format="bulleted-list" icon="list_bulleted" />
    </Toolbar>
    <Editable
      className="editor"
      renderElement={renderElement}
      renderLeaf={renderLeaf}
      placeholder="Tu wpisz jakiś tekst..."
    />
  </Slate>
);
```

Tu zmieniliśmy nieco właściwość `placeholder`.

```
spellCheck
autoFocus
onKeyDown={(event) => {
  for (const hotkey in HOTKEYS) {
    if (isHotkey(hotkey, event as any)) {
      event.preventDefault();
      const mark = HOTKEYS[hotkey];
      toggleMark(editor, mark);
    }
  }
}}
readOnly={readOnly}
/>
</Slate>
);
};
```

23. Kolejnym komponentem, który musimy zmodyfikować, jest `ThreadCategory`. Podobnie jak wcześniej, także w jego przypadku przedstawię tylko te fragmenty, które trzeba zmienić:

```
interface ThreadCategoryProps {
  category?: Category;
  sendOutSelectedCategory: (cat: Category) => void;
```

Do interfejsu określającego postać właściwości *props* dodaliśmy nową właściwość, `sendOutSelectedCategory`, która jest funkcją pozwalającą na przekazanie aktualnie wybranej kategorii do komponentu nadrzędnego.

```

}
const ThreadCategory: FC<ThreadCategoryProps> = ({
  category,
  sendOutSelectedCategory
}) => {

```

Tu mamy zmodyfikowaną postać właściwości *props* komponentu.

```

  return (
    <div className="thread-category-container">
      <strong>{category?.name}</strong>
      <div style={{ marginTop: "1em" }}>
        <CategoryDropDown
          preselectedCategory={category}
          sendOutSelectedCategory={sendOutSelectedCategory}
        />

```

I w końcu w tym fragmencie przekazujemy funkcję `sendOutSelectedCategory` do komponentu `CategoryDropDown`, dzięki czemu podczas tworzenia wątku wybrana kategoria zostanie przekazana do komponentu `Thread`.

24. Kolejne zmiany musimy wprowadzić w komponencie `ThreadTitle`; przedstawiłem je poniżej:

```

import React, { FC, useEffect, useState } from "react";

interface ThreadTitleProps {
  title?: string;
  readOnly: boolean;

```

Teraz chcemy, by tytuł był prezentowany w trybie tylko do odczytu, jeśli w komponencie jest prezentowany już istniejący wątek.

```

  sendOutTitle: (title: string) => void;

```

Także tu stosujemy nasz wzorec *sendOut*, czyli funkcję umożliwiającą przekazywanie danych do komponentu nadrzędnego; w tym przypadku funkcja ta będzie przekazywana jako właściwość *props* o nazwie `sendOutTitle`.

```

}

const ThreadTitle: FC<ThreadTitleProps> = ({
  title,
  readOnly,
  sendOutTitle
}) => {
  const [currentTitle, setCurrentTitle] = useState("");

  useEffect(() => {
    setCurrentTitle(title || "");
  }, [title]);

  const onChangeTitle = (e: React.ChangeEvent<HTMLInputElement>) => {
    setCurrentTitle(e.target.value);
    sendOutTitle(e.target.value);

```

W tym fragmencie kodu ustawiamy tytuł, jak również przesyłamy go do komponentu nadrzędnego.

```
};

return (
  <div className="thread-title-container">
    <strong>Tytuł</strong>
    <div className="field">
      <input
        type="text"
        value={currentTitle}
        onChange={onChangeTitle}
        readOnly={readOnly}>
```

W tym fragmencie kodu używamy nowych właściwości *props*.

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

export default ThreadTitle;
```

**25.** Kolejnym komponentem, którym się zajmiemy, będzie `ThreadBody`; zmodyfikuj go zgodnie z podanymi przykładami:

```
import React, { FC } from "react";
import RichEditor from "../../editor/RichEditor";
import { Node } from "slate";

interface ThreadBodyProps {
  body?: string;
  readOnly: boolean;
  sendOutBody: (body: Node[]) => void;
```

Także w tym komponentcie używamy naszego wzorca *sendOut*: dodaliśmy do niego właściwość *props* `sendOutBody`.

```
  }

  const ThreadBody: FC<ThreadBodyProps> = ({ body, readOnly, sendOutBody }) => {
    return (
      <div className="thread-body-container">
        <strong>Treść</strong>
        <div className="thread-body-editor">
          <RichEditor
            existingBody={body}
            readOnly={readOnly}
            sendOutBody={sendOutBody}>
```

Teraz musimy przekazać funkcję `sendOutBody` do komponentu `RichEditor`, gdyż to on odpowiada za aktualizowanie treści wpisu.

```
      />
    </div>
  </div>
```

```
);
};
```

```
export default ThreadBody;
```

26. I w końcu musimy wprowadzić kilka poprawek w pliku *Thread.tsx*.  
Przyjrzyjmy się im dokładnie.

Niezbędne instrukcje importu powinieneś już być w stanie dodać samemu, a będziemy potrzebowali, na przykład, nowej funkcji pomocniczej `getTextFromNode`.

```
const GetThreadById = gql`
  query GetThreadById($id: ID!) {
    getThreadById(id: $id) {
      ... on EntityResult {
        messages
      }
    }
  }
`;
```

```
    ... on Thread {
      id
      user {
        id
        userName
      }
      lastModifiedOn
      title
      body
      points
      category {
        id
        name
      }
      threadItems {
        id
        body
        points
        thread {
          id
        }
        user {
          id
          userName
        }
      }
    }
  }
`;
```

```
const CreateThread = gql`
  mutation createThread(
    $userId: ID!
    $categoryId: ID!
    $title: String!
    $body: String!
  ) {
    createThread(
      userId: $userId
      categoryId: $categoryId
      title: $title
      body: $body
    ) {
      id
      title
      body
      points
      category {
        id
        name
      }
      threadItems {
        id
        body
        points
        thread {
          id
        }
        user {
          id
          userName
        }
      }
    }
  }
`;
```

```

        createThread(
          userId: $userId
          categoryId: $categoryId
          title: $title
          body: $body
        ) {
          messages
        }
      }
    }
  };

```

Ten fragment kodu przedstawia nową mutację CreateThread.

```

const threadReducer = (state: any, action: any) => {
  switch (action.type) {
    case "userId":
      return { ...state, userId: action.payload };
    case "categoryId":
      return { ...state, category: action.payload };
    case "title":
      return { ...state, title: action.payload };
    case "body":
      return { ...state, body: action.payload };
    case "bodyNode":
      return { ...state, bodyNode: action.payload };
    default:
      throw new Error("Nieznany typ akcji");
  }
};

```

Konieczne jest także dodanie nowego reduktora, threadReducer.

```

const Thread = () => {
  const { width } = useWindowDimensions();
  const [execGetThreadById, { data: threadData }] = useLazyQuery(
    GetThreadById,
    { fetchPolicy: "no-cache" }
  );
  const [thread, setThread] = useState<ThreadModel | undefined>();
  const { id } = useParams();
  const [readOnly, setReadOnly] = useState(false);
  const user = useSelector((state: AppState) => state.user);

```

Tu mamy obiekt user, który pojawia się wyłącznie w przypadku, gdy użytkownik będzie zalogowany. Tego obiektu będziemy używać wyłącznie podczas tworzenia nowego wątku (encji Thread).

```

const [
  { userId, category, title, bodyNode },
  threadReducerDispatch,
] = useReducer(threadReducer, {
  userId: user ? user.id : "0",
  category: undefined,
  title: "",
  body: "",
  bodyNode: undefined,
});

```

W tym fragmencie kodu tworzymy reduktor. Tych pól będziemy używać podczas przesyłania danych nowego, tworzonego wątku.

```
const [postMsg, setPostMsg] = useState("");
```

Ten wiersz kodu pozwoli pokazywać status próby utworzenia nowego wątku.

```
const [execCreateThread] = useMutation(CreateThread);
```

W tym wierszu tworzymy funkcję, `execCreateThread`, która będzie wywoływać mutację tworzącą wątek.

```
const history = useHistory();
```

Funkcji `useHistory()` będziemy używać po to, by przełączyć się na ekran nowo utworzonego wątku. Na przykład, jeśli identyfikatorem nowego wątku będzie 25, to aby go wyświetlić, użyjemy trasy `"/thread/25"`.

```
const refreshThread = () => {
  if (id && id > 0) {
    execGetThreadById({
      variables: {
        id,
      },
    });
  }
};

useEffect(() => {
  if (id && id > 0) {
    execGetThreadById({
      variables: {
        id,
      },
    });
  }
}, [id, execGetThreadById]);

useEffect(() => {
  threadReducerDispatch({
    type: "userId",
    payload: user ? user.id : "0",
  });
}, [user]);
```

Tutaj, w przypadku, gdyby użytkownik się zalogował, aktualizujemy jego identyfikator używany przez reduktor.

```
useEffect(() => {
  if (threadData && threadData.getThreadById) {
    setThread(threadData.getThreadById);
    setReadOnly(true);
  } else {
    setThread(undefined);
    setReadOnly(false);
  }
}, [threadData]);
```

```
const receiveSelectedCategory = (cat: Category) => {
  threadReducerDispatch({
    type: "category",
    payload: cat,
  });
};
```

W tym fragmencie kodu zaczęliśmy dodawać funkcje obsługujące przekazywanie danych z komponentów podrzędnych zgodnie z zastosowanym w nich wzorcem *sendOut*. Przedstawiona tu funkcja *receiveSelectedCategory* będzie pobierać z komponentu *CategoryDropDown* wybraną w nim kategorię — obiekt *Category*.

```
const receiveTitle = (updatedTitle: string) => {
  threadReducerDispatch({
    type: "title",
    payload: updatedTitle,
  });
};
```

```
const receiveBody = (body: Node[]) => {
  threadReducerDispatch({
    type: "bodyNode",
    payload: body,
  });
  threadReducerDispatch({
    type: "body",
    payload: getTextFromNodes(body),
  });
};
```

Funkcje *receiveTitle* oraz *receiveBody* obsługują modyfikacje tytułu wątku (*title*) oraz jego treści (*body*), przekazywanych z odpowiednich komponentów podrzędnych.

```
const onClickPost = async (
  e: React.MouseEvent<HTMLButtonElement, MouseEvent>
) => {
```

Funkcja *onClickPost* obsługuje kliknięcie przycisku *Publikuj* i przesyła dane nowego wątku.

```
  e.preventDefault();

  console.log("bodyNode", getTextFromNodes(bodyNode));
  if (!userId || userId === "0") {
    setPostMsg("Użytkownik musi być zalogowany, by móc opublikować  
wątek.");
  } else if (!category) {
    setPostMsg("Proszę wybrać kategorię tematyczną.");
  } else if (!title) {
    setPostMsg("Proszę podać tytuł wątku.");
  } else if (!bodyNode) {
    setPostMsg("Proszę wpisać treść wątku.");
  }
```

Ta sekwencja instrukcji warunkowych obsługuje walidację wartości pól reduktora, które zostaną przesłane w ramach tworzenia nowego wątku.



```

    } else {
      setPostMsg("");
      const newThread = {
        userId,
        categoryId: category?.id,
        title,
        body: JSON.stringify(bodyNode),

```

Zwróć uwagę na to, że w tym fragmencie konwertujemy tablicę obiektów Node używanych przez bibliotekę Slate.js na łańcuch w formacie JSON, gdyż właśnie w tym formacie treść wątku będzie zapisywana w bazie danych.

```

    };

```

Zakończony tu fragment kodu przygotowuje parametry dla mutacji CreateThread.

```

    const { data: createThreadMsg } = await execCreateThread({
      variables: newThread,
    });

```

W tym fragmencie wykonujemy mutację CreateThread.

```

    if (
      createThreadMsg.createThread &&
      createThreadMsg.createThread.messages &&
      !isNaN(createThreadMsg.createThread.messages[0])
    ) {
      setPostMsg("Wątek został pomyślnie utworzony.");
      history.push(`/thread/${createThreadMsg.createThread.messages[0]}`);

```

Jeśli próba utworzenia nowego wątku zakończyła się pomyślnie, przekierowujemy użytkownika na ekran prezentujący ten wątek, używając do tego celu jego identyfikatora.

```

    } else {
      setPostMsg(createThreadMsg.createThread.messages[0]);

```

W przeciwnym razie, jeśli nie udało się utworzyć wątku, wyświetlamy komunikat zwrócony przez serwer.

```

    }
  }
};

return (
  <div className="screen-root-container">
    <div className="thread-nav-container">
      <Nav />
    </div>
    <div className="thread-content-container">
      <div className="thread-content-post-container">
        {width <= 768 && thread ? (
          <ThreadPointsInline
            points={thread?.points || 0}
            threadId={thread?.id}
            refreshThread={refreshThread}
            allowUpdatePoints={true}

```

```
    />
  ) : null}
```

Ten komponent, `ThreadPointsInline`, wyświetlamy, jeśli wątek już istnieje i jeśli aplikacja jest używana na urządzeniu mobilnym; w przeciwnym razie komponent ten nie zostanie wyrenderowany.

```
<ThreadHeader
  userName={thread ? thread.user.userName : user?.userName}
  lastModifiedOn={thread ? thread.lastModifiedOn : new Date()}
  title={thread ? thread.title : title}
/>
<ThreadCategory
  category={thread ? thread.category : category}
  sendOutSelectedCategory={receiveSelectedCategory}
/>
<ThreadTitle
  title={thread ? thread.title : ""}
  readOnly={thread ? readOnly : false}
  sendOutTitle={receiveTitle}
/>
<ThreadBody
  body={thread ? thread.body : ""}
  readOnly={thread ? readOnly : false}
  sendOutBody={receiveBody}
/>
```

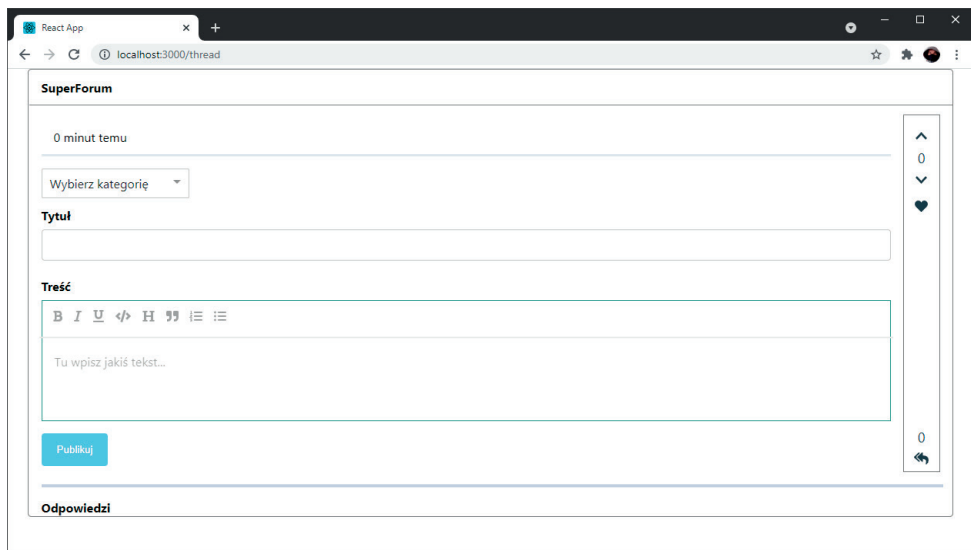
Pozostałe komponenty podrzędne są używane w identyczny sposób. Jeśli obiekt `thread` istnieje, to przekazujemy odpowiednią daną, a w przeciwnym razie przechodzimy w danym komponencie do trybu publikacji wątku.

```
{thread ? null : (
  <>
    <div style={{ marginTop: ".5em" }}>
      <button className="action-btn" onClick={onClickPost}>
        Publikuj
      </button>
    </div>
    <strong>{postMsg}</strong>
  </>
)}
```

W tym fragmencie kodu wyświetlamy przycisk *Publikuj* oraz komunikat ze statusem operacji. Także tutaj, jeśli obiekt wątku istnieje, elementy te nie będą widoczne, jeśli natomiast nie istnieje, to je wyświetlimy.

Pozostała część kodu nie ulega zmianie, więc nie będę jej tutaj przedstawiał.

27. Jeśli teraz spróbujesz wyświetlić trasę wątku bez podawania parametru `id`, zostanie wyświetlony ekran przedstawiony na rysunku 16.6.



Rysunek 16.6. Ekran nowego wątku

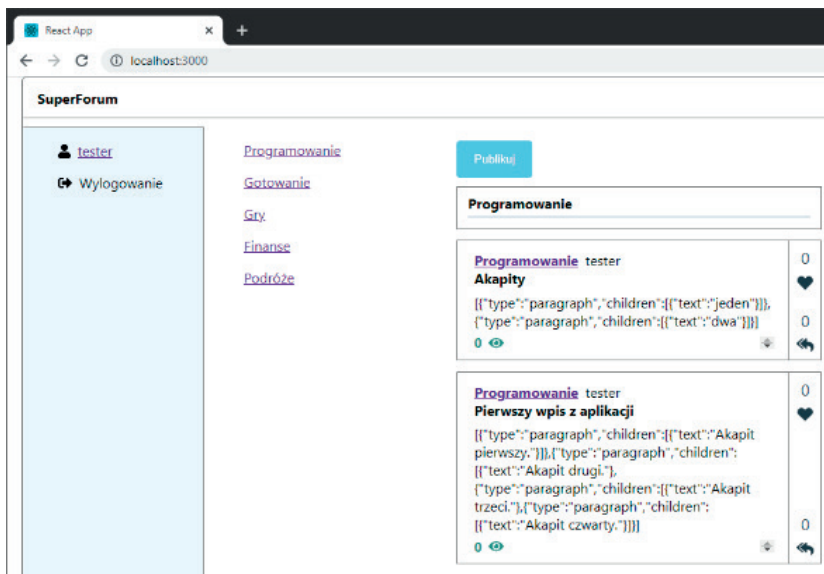
Ponieważ obecnie używamy tablicy obiektów Node biblioteki Slate.js jako łańcuchów w formacie JSON w Body, dlatego zanim wypróbujesz ten kod i ponownie spróbujesz wyświetlić wątki, będziesz musiał usunąć z bazy danych wszystkie dane z tabel Thread i ThreadItem.

Jest jednak pewien problem. Aktualnie w polu Body bazy danych są zapisywane łańcuchy w formacie JSON, dlatego kiedy odczytamy wpisy z bazy, będą one prezentowane tak, jak pokazałem na rysunku 16.7.

Oczywiście, nie jest to pożądany sposób działania aplikacji. Podczas prezentowania wątków na liście, musimy te teksty wyświetlać w normalny sposób. Na szczęście możemy zastosować istniejący komponent RichEditor do wyświetlania tych tekstów z zachowaniem istniejącego formatowania.

28. Zmodyfikuj kod komponentu RichEditor poprzez dodanie do niego sprawdzenia właściwości `readOnly`, w zależności od której będzie wyświetlany lub ukrywany pasek narzędzi edytora.

```
{readOnly ? null : (
  <Toolbar>
    <MarkButton format="bold" icon="bold" />
    <MarkButton format="italic" icon="italic" />
    <MarkButton format="underline" icon="underlined" />
    <MarkButton format="code" icon="code" />
    <BlockButton format="heading-one" icon="header1" />
    <BlockButton format="block-quote" icon="in quotes" />
    <BlockButton format="numbered-list" icon="list_numbered" />
    <BlockButton format="bulleted-list" icon="list_bulleted" />
  </Toolbar>
)}
```



Rysunek 16.7. Ekran główny aplikacji

A zatem, jeśli edytor będzie działał w trybie tylko do odczytu, pasek narzędzi (komponent `Toolbar`) nie będzie widoczny.

29. Teraz zmodyfikuj komponent `ThreadCard`; znajdujący się w jego kodzie JSX element `<div>{thread.body}</div>` zastąp następującym komponentem:

```
<RichEditor existingBody={thread.body} readOnly={true} />
```

Nie zapomnij zaimportować komponentu `RichEditor`.

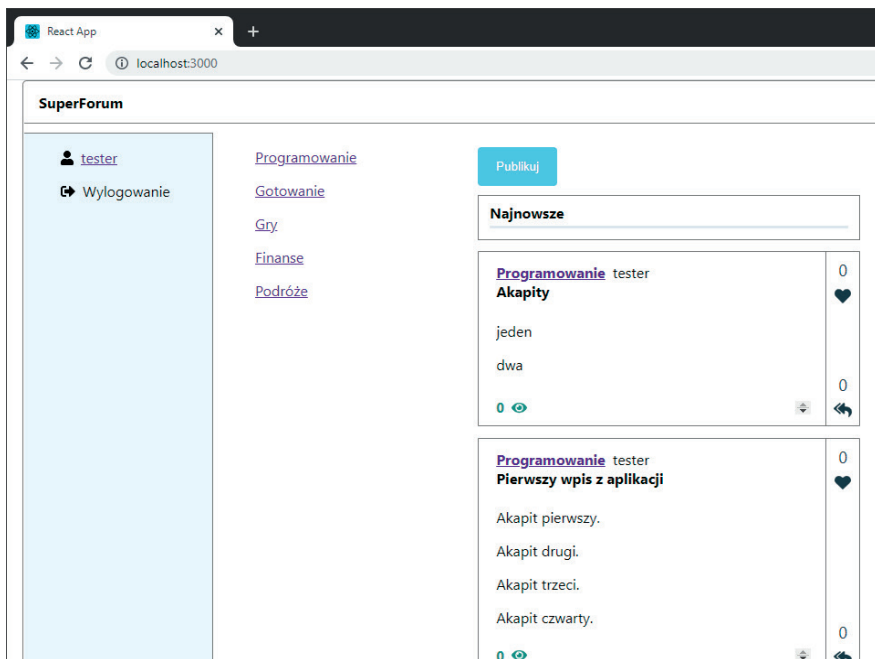
30. Teraz na ekranie głównym powinieneś zobaczyć wątki prezentowane tak, jak na rysunku 16.8. Oczywiście, Twoje dane będą zapewne wyglądać inaczej.

Zwróć uwagę na to, że wprowadzone zmiany sprawią także, że dla już istniejącego wątku komponent `RichEditor` używany na ekranie wątku będzie działał w trybie tylko do odczytu. Teraz pozostaje nam jedynie zapewnić możliwość dodawania odpowiedzi (`ThreadItem`) do wątków i będziemy mogli uznać tę część aplikacji za skończoną. Zmodyfikujemy komponent `ThreadResponse` w taki sposób, by pozwalał nie tylko na wyświetlanie odpowiedzi, lecz także na ich dodawanie:

1. W pierwszej kolejności musimy wprowadzić pewne drobne zmiany w serwerowej części aplikacji. Otwórz plik `ThreadItemRepo.ts` i znajdź w nim funkcję `createThreadItem`. Zmodyfikuj instrukcję `return` tak, jak pokazałem na poniższym przykładzie:

```
return { messages: [`${threadItem.id}`] };
```

Podobnie jak wcześniej, w funkcji `createThread`, także tutaj zwracamy identyfikator nowej odpowiedzi (encji `ThreadItem`).



Rysunek 16.8. Ekran główny po odpowiednim wyświetleniu treści wątków

- Następnie, w pliku *ThreadRepo.ts* zmodyfikuj wywołanie funkcji `findOne` zgodnie z poniższym przykładem:

```
const thread = await Thread.findOne({
  where: {
    id,
  },
  relations: [
    "user",
    "threadItems",
    "threadItems.user",
    "threadItems.thread",
    "category",
  ],
});
```

Teraz chcemy pobierać także informacje o wątku, do którego należy odpowiedź, tak, by podczas tworzenia nowej encji `ThreadItem`, w przesyłanych danych znalazł się prawidłowy identyfikator wątku (encji `Thread`).

- Teraz musimy zmodyfikować model w pliku *ThreadItem.ts* w klienckiej części aplikacji, tak, by zamiast identyfikatora odpowiedzi (`threadId`) podbierał jej obiekt (`thread`):

```
public thread: Thread
```

W ten sposób ze zmodyfikowanego przed chwilą zapytania będziemy pobierać obiekt `Thread`, a nie sam identyfikator.

4. Następnie zmodyfikuj komponent `ThreadResponse`, zgodnie z jego kodem zamieszczonym w przykładach dołączonych do książki. Przy okazji upewnij się, że dodasz do pliku wszystkie niezbędne instrukcje importu.

W pierwszej kolejności dodaj mutację `CreateThreadItem`.

W interfejsie `ThreadResponseProps` widzimy, że właściwość `props` body jest początkową wartością komponentu `RichEditor`, przed wprowadzeniem jakichkolwiek zmian. Do interfejsu dodaj także pole `threadId`, które będzie nam potrzebne, jeśli mamy dodawać nowe odpowiedzi.

Następnie musimy pobrać obiekt `user`, używając w tym celu funkcji `useSelector`. Jest to konieczne, gdyż nowa odpowiedź musi zostać skojarzona z aktualnie zalogowanym użytkownikiem.

Następnie przygotowujemy `execCreateThreadItem` — funkcję, która będzie umożliwiała wykonanie mutacji `CreateThreadItem`.

Kolejnym nowym elementem jest `postMsg` — właściwość stanu zawierająca komunikat informujący o efektach próby zapisu nowej odpowiedzi.

Następną właściwością stanu jest `bodyToSave`, przechowująca aktualną treść odpowiedzi wpisaną przez użytkownika w komponencie `RichEditor`.

Kolejnym nowym elementem kodu jest wywołanie `useEffect`, którego używamy do zainicjowania wartości `bodyToSave` na podstawie przekazanej do komponentu właściwości `props` body. W ten sposób określimy początkową treść odpowiedzi.

Kolejna nowa funkcja, `onClickPost`, pozwala nam sprawdzić poprawność danych przed próbą utworzenia nowej odpowiedzi. Po sprawdzeniu danych, możemy je przesłać i odświeżyć wątek, do którego odpowiedź została dodana.

W funkcji `receiveBody` pobieramy tekst przekazany z komponentu `RichEditor`. Funkcja ta jest używana podczas tworzenia nowej odpowiedzi.

W zwracanym kodzie JSX nie generujemy komponentu `ThreadPointsInline`, jeśli komponent działa w trybie tylko od odczytu. Jeśli natomiast komponent działa w trybie edycji, wyświetlamy dodatkowo przycisk *Publikuj odpowiedź* oraz komunikat o statusie operacji.

5. Aby uruchomić odpowiadanie na wątki, musimy jeszcze wprowadzić zmianę w kodzie pliku `Thread.tsx`. Zastąp poniższy fragment zwracanego kodu JSX:

```
<div className="thread-content-response-container">
  <hr className="thread-section-divider" />
  <ThreadResponsesBuilder threadItems={thread?.threadItems}
    readOnly={readOnly} />
</div>
```

następującym fragmentem:

```
{thread ? (
  <div className="thread-content-response-container">
    <hr className="thread-section-divider" />
    <div style={{ marginBottom: ".5em" }}>
      <strong>0publikuj odpowiedź</strong>
```

```

</div>
<ThreadResponse
  body={""}
  userName={user?.userName}
  lastModifiedOn={new Date()}
  points={0}
  readOnly={false}
  threadItemId={"0"}
  threadId={thread.id}
  refreshThread={refreshThread}
/>
</div>
) : null}
{thread ? (
  <div className="thread-content-response-container">
    <hr className="thread-section-divider" />
    <ThreadResponsesBuilder
      threadItems={thread?.threadItems}
      readOnly={readOnly}
      refreshThread={refreshThread}
    />
  </div>
) : null}

```

Jak widać, dodaliśmy tu nową sekcję z nagłówkiem, edytorem i przyciskiem umożliwiającym opublikowanie odpowiedzi.

6. I w końcu musimy także poprawić kod w pliku *ThreadResponseBuilder.tsx*. Poniżej opisałem tylko te fragmenty, które musisz zmodyfikować.

Zacznij od zmiany interfejsu *ThreadResponseBuilderProps* do postaci:

```

interface ThreadResponsesBuilderProps {
  threadItems?: Array<ThreadItem>;
  readOnly: boolean;
  refreshThread?: () => void;
}

```

Następnie dodaj właściwość *props* *refreshThread* do listy właściwości komponentu:

```

const ThreadResponsesBuilder: FC<ThreadResponsesBuilderProps> = ({
  threadItems,
  readOnly,
  refreshThread,
}) => {

```

I w końcu, zmodyfikuj użycie komponentu *ThreadResponse* w kodzie JSX do postaci:

```

<li key={`thr-${ti.id}`}>
  <ThreadResponse
    body={ti.body}
    userName={ti.user.userName}
    lastModifiedOn={ti.createdOn}
    points={ti.points}
    readOnly={readOnly}

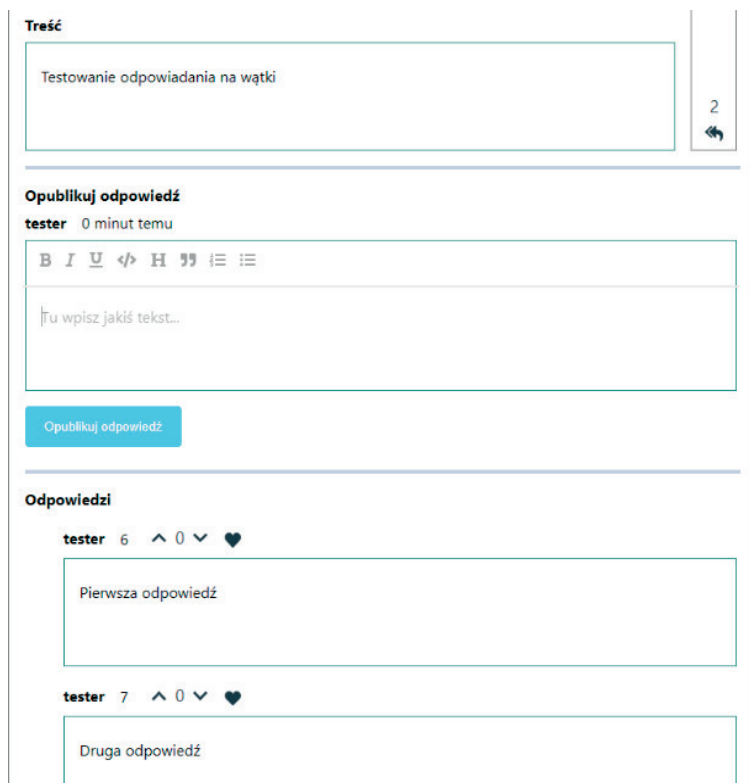
```

```

        threadItemId={ti?.id || "0"}
        threadId={ti.thread.id}
        refreshThread={refreshThread}
      />
    </li>

```

7. Jeśli teraz opublikujesz kilka odpowiedzi do wątku, to ekran wątku będzie wyglądać tak, jak na rysunku 16.9.



Rysunek 16.9. Opublikowane odpowiedzi do wątku

I to już wszystkie zmiany, jakie należy wprowadzić w kodach obsługujących trasę i ekran wątku.

Dotarliśmy już niemal do końca i jak na razie fantastycznie się spisujesz. By dotrzeć do tego etapu, przerobiliśmy naprawdę bardzo dużo zagadnień i napisaliśmy masę kodu. Możesz być naprawdę dumny ze swoich postępów. Została nam jeszcze jedna mała sekcja i aplikacja będzie gotowa!

Ostatnim komponentem, którym się zajmiemy, będzie `RightMenu`. Będziemy w nim wyświetlać do trzech najpopularniejszych kategorii wybranych na podstawie liczby wątków w danej kategorii. Takie rozwiązanie będzie wymagało dłuższego zapytania, operującego na kilku tabelach bazy danych, i będzie stanowić świetne ćwiczenie.



1. W pierwszej kolejności musimy dodać do pliku definicji typów (*typeDefs.ts*) nowy typ, o nazwie `CategoryThread`; poniżej pokazałem jego definicję:

```
type CategoryThread {
  threadId: ID!
  categoryId: ID!
  categoryName: String!
  title: String!
  titleCreatedOn: Date!
}
```

Zwróć uwagę na to, że pole `titleCreatedOn` służy jedynie do sprawdzania sortowania; nie będziemy go używać w kodzie klienckiej części aplikacji.

2. Teraz do katalogu z kodami warstwy repozytorium serwerowej części aplikacji dodaj nowy plik, o nazwie *CategoryThread.ts*, i zapisz w nim następujący kod. Zwróć uwagę na to, że ta klasa nie będzie encją w bazie danych — będzie ona pełnić funkcję agregującą i zawierać pola pochodzące z wielu encji:

```
export default class CategoryThread {
  constructor(
    public threadId: string,
    public categoryId: string,
    public categoryName: string,
    public title: string,
    public titleCreatedOn: Date
  ) {}
}
```

3. Następnie skopiuj z przykładów dołączonych do książki plik *CategoryThreadRepo.ts*.

Na jego samym początku znajduje się funkcja wykonująca początkowe zapytanie pobierające dane `ThreadCategory` z bazy danych przy użyciu wywołania `Thread` ➔ `Category.createQueryBuilder("threadCategory")`. Zwróć uwagę na to, że uwzględniliśmy także zależność z tabelą `Threads`.

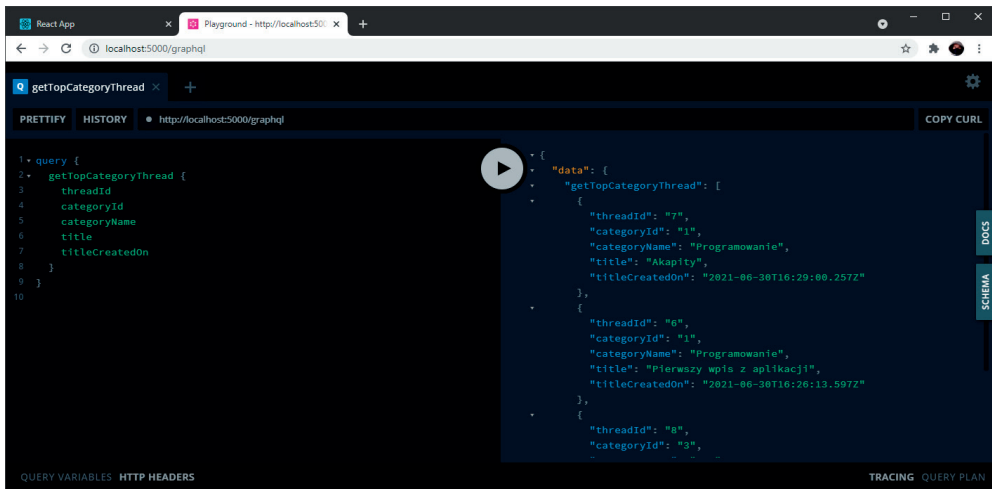
Teraz musimy zająć się dodatkowym przetworzeniem zapytania, by pobrać z niego wyniki w takiej formie, jaka nas interesuje. Nie będziemy tego robić korzystając z możliwości `TypeORM`, gdyż w przypadku bardzo złożonych operacji sortowania i filtrowania, wykonywanie ich przy użyciu `TypeORM` jest trudne i kłopotliwe. Zastosowanie standardowych mechanizmów JavaScriptu pozwoli nam uzyskać pożądane rezultaty znacznie łatwiej.

W wywołaniu `categories.sort`, zapisanym w wierszu 14., sortujemy kategorie malejąco, na podstawie liczby wątków w każdej z nich. Następnie pobieramy trzy pierwsze kategorie, które będą stanowić nasz wynik.

Następnie, w ramach każdej z trzech wybranych kategorii, pobieramy wątki (rekordy `Thread`) i sortujemy je na podstawie pola znacznika czasu, `createdOn`.

W ten sposób pobieramy, co najwyżej, trzy rekordy wątków dla każdej kategorii, posortowane według wartości znacznika czasu utworzenia (`createdOn`).

4. Spróbujmy teraz przetestować ten kod w aplikacji „placu zabaw” GraphQL-a (patrz rysunek 16.10).



Rysunek 16.10. Wyniki sortowania GetTopCategoryThread

Jak widać, sortowanie i filtry działają prawidłowo.

5. Kolejnym krokiem będzie dokończenie kodu klienckiej części aplikacji. Otwórz plik *CategoryThread.ts* i zmień pole *category* na *categoryName*. W ten sposób modele w klienckiej i serwerowej części aplikacji będą sobie odpowiadać.
6. Otwórz plik *TopCategory.tsx* i zmodyfikuj poniższy wiersz w kodzie JSX:

```
<strong>{topCategories[0].categoryName}</strong>
```

Zmień w nim *category* na *categoryName*.

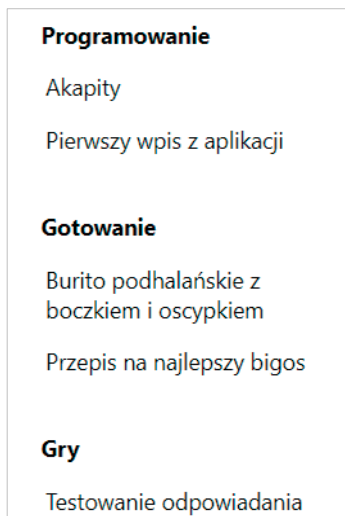
7. Teraz otwórz plik *RightMenu.tsx* i zaktualizuj go zgodnie z kodem źródłowym analogicznego pliku z przykładów dołączonych do książki.

Po dodaniu niezbędnych instrukcji importu, musimy zdefiniować zapytanie GraphQL-a, *GetTopCategoryThread*, a następnie użyć go (w wierszu 20.) w wywołaniu *useQuery*.

Następnie, w wierszu 26. musimy zmodyfikować wywołanie *useEffect*, by korzystało z wynikowych danych *categoryThreadData*. Przy użyciu funkcji *groupBy* z biblioteki *lodash* grupujemy dane na podstawie nazwy kategorii (*categoryName*), dzięki czemu łatwiej będzie nam na nich operować. Pierwotnie kod tej części aplikacji został opisany w rozdziale 11., pt. „Czego się nauczysz — aplikacja internetowego forum”.

I w końcu, na podstawie dostępnej szerokości ekranu, musimy sprawdzić, czy aplikacja nie została uruchomiona na urządzeniu mobilnym; jeśli ekran jest wąski, musimy zwrócić *null*, a w przeciwnym razie wyrenderować interfejs użytkownika.

8. Jeśli teraz wyświetlisz ekran główny aplikacji, powinieneś zobaczyć komponent *RightMenu* o postaci podobnej do tej z rysunku 16.11.



**Rysunek 16.11.** Ekran główny z wyświetlonymi kategoriami

Pamiętaj, że Twoje dane będą zapewne wyglądać nieco inaczej.

To była ostatnia zmiana! Aplikacja jest gotowa! W ramach prac nad nią napisałeś naprawdę bardzo dużo kodu, skorzystałeś z wielu frameworków i poznałeś wiele zagadnień. Praca, którą wykonałeś, jest naprawdę niesamowita! Teraz możesz sobie zrobić w pełni zasłużoną przerwę.

W tym podrozdziale dopracowaliśmy kod klienckiej część aplikacji i scaliliśmy go z serwerem GraphQL-a. Musieliśmy zmodyfikować nieco używane style i wprowadzić zmiany, korzystając z technik refaktoryzacji. Przy okazji musieliśmy także poprawić kilka trudnych do zlokalizowania błędów. Innymi słowy, robiliśmy wszystko, co będziemy robić podczas prac nad rzeczywistymi projektami. Prace nad tą aplikacją były zatem doskonałym treningiem.

## Podsumowanie

W tym ostatnim rozdziale, w którym zajmowaliśmy się pracami nad kodem aplikacji, scaliiliśmy wszystko w jedną całość, integrując kliencką aplikację Reacta oraz serwer GraphQL-a. Z tego rozdziału, jak również z całej książki, dowiedziałeś się bardzo wiele i powinieneś czuć się dumny z powodu wszystkiego, czego udało Ci się dokonać.

Chciałbym zasugerować, byś przed rozpoczęciem lektury ostatniego rozdziału książki spróbował poeksperymentować z aplikacją i wprowadzać w niej jakieś zmiany. Spróbuj samemu wymyślić jakieś nowe możliwości aplikacji, a następnie je zaimplementować. W końcu, to jest jedyny sposób, aby się czegoś faktycznie nauczyć.

Z ostatniego, 17. rozdziału książki, pt. „Wdrażanie w chmurze AWS”, dowiesz się jak wdrożyć aplikację w chmurze Amazon Web Services, na wirtualnej maszynie z systemem Linux i serwerem NGINX.

# Wdrażanie w chmurze AWS

Po zakończeniu prac programistycznych, zanim będzie można używać aplikacji, trzeba ją będzie wdrożyć. Można to zrobić na wiele różnych sposobów, a jednym z nich jest zastosowanie własnej infrastruktury. Jednak obecnie firmy preferują korzystanie z usług w chmurze, starając się w ten sposób redukować koszty związane z utrzymaniem własnych działów IT.

W tym rozdziale opiszę wdrażanie naszej przykładowej aplikacji w chmurze **Amazon Web Services (AWS)**, która, jeśli chodzi o dostawców usług chmurowych, stanowi standard. Naszą aplikację wraz z niezbędnym magazynem Redisa, bazą danych Postgres i serwerem WWW NGINX uruchomimy na wirtualnej maszynie z systemem Linux.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- zainstalowaniem dystrybucji Linuksa Ubuntu na wirtualnej maszynie w chmurze AWS;
- zainstalowaniem i konfiguracją Redisa, Postgresa oraz Node w systemie Ubuntu;
- skonfigurowaniem i wdrożeniem naszej aplikacji na serwerze NGINX.

## Wymagania techniczne

Przystępując do lektury tego rozdziału, powinieneś dysponować dobrą znajomością technologii internetowych. Choć uzyskanie statusu starszego programisty może zająć całe lata, to obecnie powinieneś już wprawnie posługiwać się językami TypeScript i JavaScript, frameworkami React, Express i serwerem GraphQL. Także w tym rozdziale będziemy używali środowiska Node oraz programu Visual Studio Code.

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (<https://ftp.helion.pl/przyklady/retyno.zip>), a ich oryginalne, anglojęzyczne wersje — w serwisie GitHub, w repozytorium <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node>. W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial17* (*Chap17*).

Przed przystąpieniem do prac, skonfiguruj swój komputer roboczy zgodnie z poniższą listą:

1. Utwórz katalog *rozdzial17* i skopiuj do niego katalogi *super-forum-server* oraz *super-forum-client* z katalogu *rozdzial15*.
2. Jeśli w skopiowanych katalogach znajdują się podkatalogi *node\_modules* oraz pliki *package-lock.json*, to je usuń.
3. Następnie w panelu terminala przejdź do katalogu *rozdzial17/super-forum-server* i wykonaj następujące polecenie:

```
npm install
```

4. W końcu przejdź do katalogu *rozdzial17/super-forum-client* i wykonaj to samo polecenie:

```
npm install
```

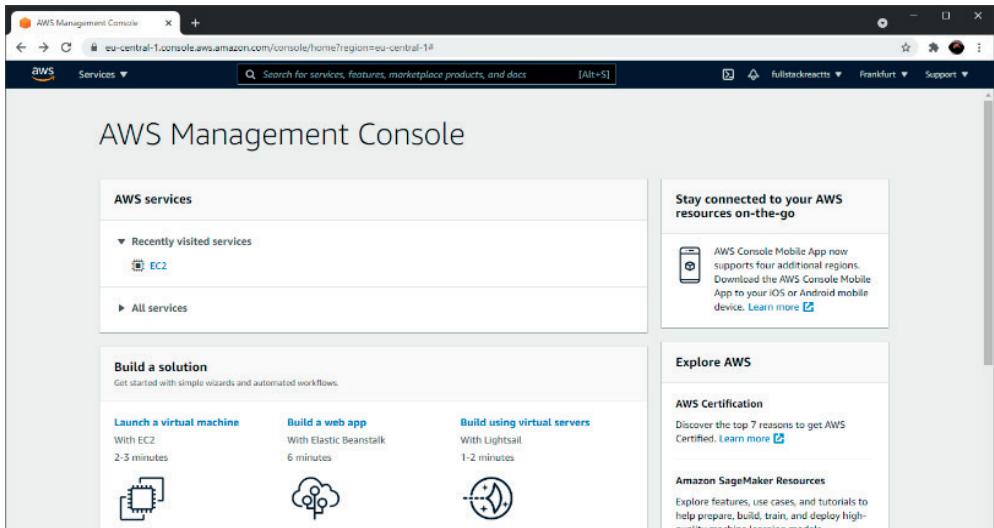
## Konfiguracja Ubuntu w chmurze AWS

W tym podrozdziale opiszę proces wyboru i instalowania dystrybucji Linuksa, a konkretnie serwera Ubuntu, na virtualnej maszynie w chmurze AWS. Zakładam, że już dysponujesz własnym kontem w chmurze AWS. Cały proces jest stosunkowo prosty, gdyż potrzebny obraz systemu Ubuntu jest już gotowy do użycia. A zatem, zaczynamy:

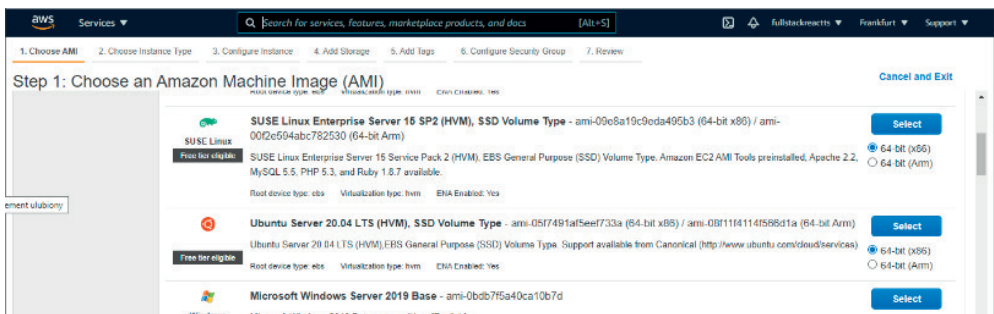
1. Zaloguj się do portalu AWS, którego aktualny wygląd przedstawiłem na rysunku 17.1. Pamiętaj, że wygląd tego portalu zmienia się stosunkowo często, więc ten, który zobaczysz, może być nieco inny.
2. Na stronie widoczny jest odnośnik *Launch a virtual machine*; kliknij go, a w przeglądarce zostanie wyświetlona strona przedstawiona na rysunku 17.2.

Wybierz obraz **Ubuntu Server 20.04 LTS**. To najnowsza wersja systemu Ubuntu ze wsparciem długoterminowym (*Long Time Support*).

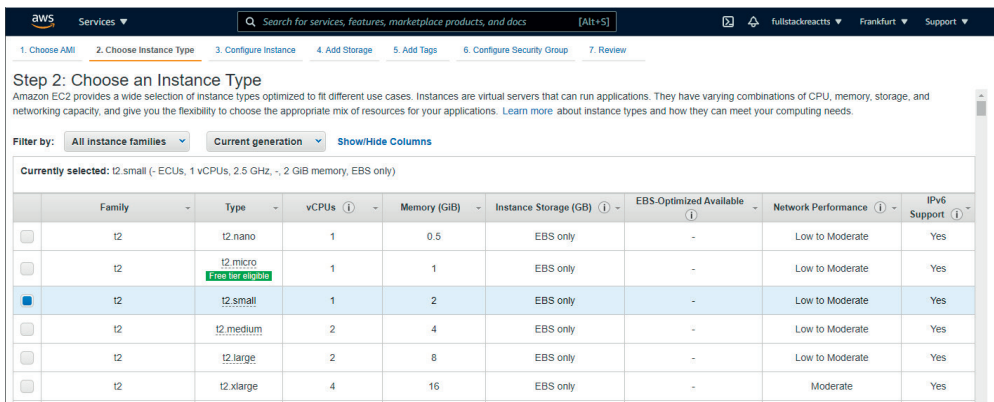
3. Po kliknięciu przycisku *Select* zostanie wyświetlona kolejna strona, przedstawiona na rysunku 17.3.



Rysunek 17.1. Portal AWS



Rysunek 17.2. Początkowa strona procesu tworzenia nowej maszyny wirtualnej



Rysunek 17.3. Wybór typu instancji maszyny wirtualnej

Ja wybrałem jeden ze słabszych obrazów, wirtualną maszynę z 1 wirtualnym procesorem i 2 GB pamięci. Zwróć uwagę na to, że EBS jest charakterystycznym dla AWS rozwiązaniem do optymalizacji przestrzeni dyskowej.

Żeby zbytnio sobie nie utrudniać procesu tworzenia maszyny wirtualnej i zaakceptować ustawienia domyślne, po zaznaczeniu rodzaju tworzonej maszyny kliknij widoczny u dołu strony przycisk *Review and Launch*.

4. Na kolejnej stronie (patrz rysunek 17.4) zostały przedstawione najważniejsze informacje na temat dokonanych wcześniej wyborów:

**Step 7: Review Instance Launch**

**AMI Details**

Ubuntu Server 20.04 LTS (HVM), SSD Volume Type - ami-0b7491af5eef733a

Price for on-demand: \$0.0416 per hour (US East - Ohio)

Root Device Type: ebs Virtualization type: hvm

**Instance Type**

Instance Type	ECUs	vCPUs	Memory (GiB)	Instance Storage (GiB)	EBS-Optimized Available	Network Performance
t2.small	-	1	2	EBS only	-	Low to Moderate

**Security Groups**

Security group name: launch-wizard-1

Description: launch-wizard-1 created 2021-07-01T16:01:35.131+02:00

Type	Protocol	Port Range	Source	Description
SSH	TCP	22	0.0.0.0/0	

**Storage**

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination	Encrypted
Root	/dev/sda1	snapp-0c0de4ef3b0b10ad39	8	gp2	100 / 3000	N/A	Yes	Not Encrypted

**Rysunek 17.4.** Początek strony z zestawieniem informacji o konfiguracji tworzonej maszyny wirtualnej

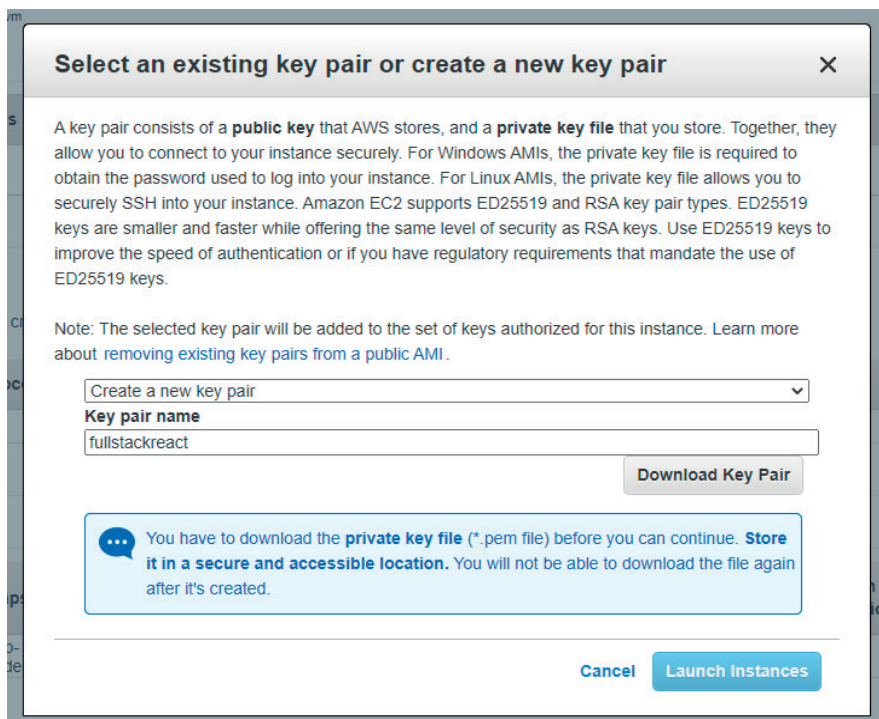
Teraz, aby kontynuować, kliknij przycisk *Launch* umieszczony u dołu strony.

5. Po kliknięciu przycisku na ekranie zostanie wyświetlone okno dialogowe przedstawione na rysunku 17.5.

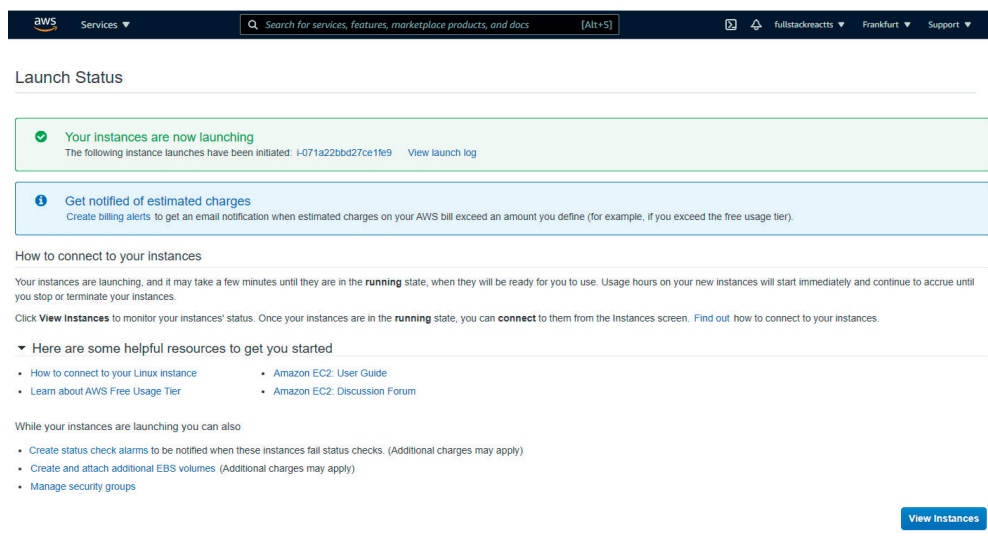
To okno dialogowe pozwala utworzyć parę kluczy kryptograficznych, wykorzystywanych do łączenia się z maszyną wirtualną przy użyciu SSH; jeden klucz jest dla nas, a drugi dla AWS. Dzięki nim będziemy w stanie zdalnie pracować na maszynie wirtualnej. Pobierz te pliki i zapisz je w bezpiecznym miejscu. Następnie kliknij przycisk *Launch Instances*.

Pobrany plik *pem* musisz przechowywać w bezpiecznym, choć łatwo dostępnym miejscu. Nie będziesz mógł pobrać go drugi raz.

6. Po zakończeniu tworzenia maszyny wirtualnej zostanie wyświetlona strona podsumowania, taka jak ta z rysunku 17.6. Aby przejść na stronę z informacjami o utworzonych maszynach wirtualnych, kliknij umieszczony u dołu przycisk *View Instances*.



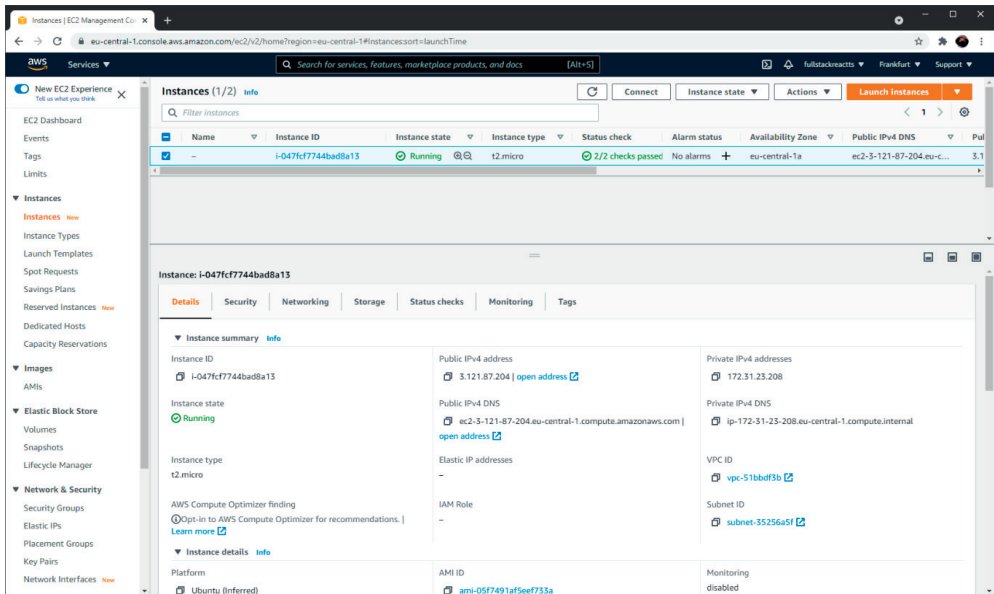
Rysunek 17.5. Okno dialogowe do utworzenia lub wyboru pary kluczcy



Rysunek 17.6. Strona podsumowania tworzenia maszyny wirtualnej

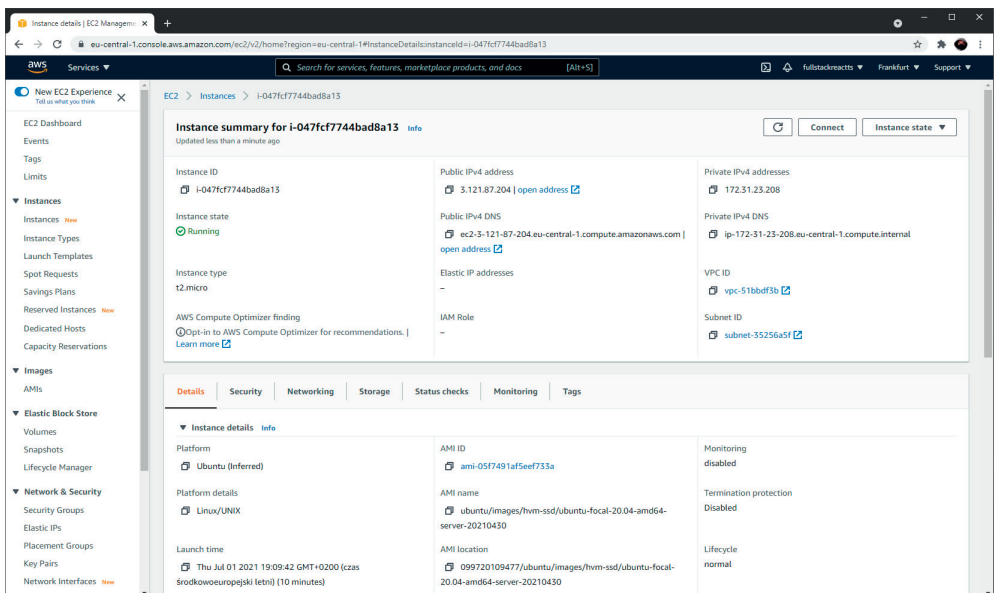
7. Teraz w przeglądarce zostanie wyświetlona strona portalu AWS, prezentująca informacje o maszynach wirtualnych (patrz rysunek 17.7).





Rysunek 17.7. Strona z informacjami o maszynach wirtualnych

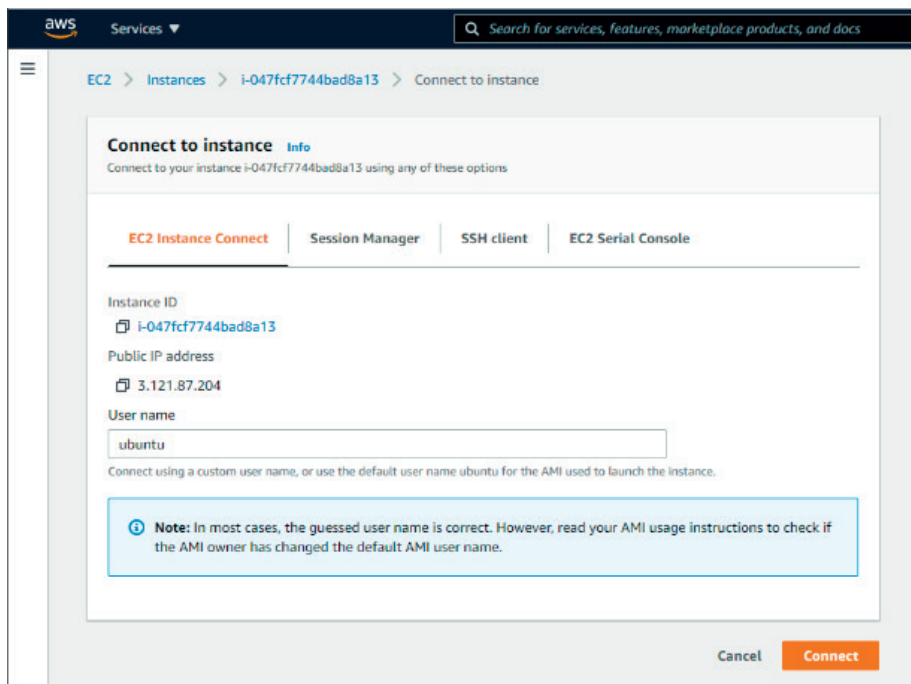
8. Kliknij odnośnik umieszczony w kolumnie *Instance ID*, a zostanie wyświetlona kolejna strona, z informacjami o konkretnej maszynie wirtualnej; przedstawiłem ją na rysunku 17.8.



Rysunek 17.8. Zebrane informacje na temat wybranej instancji maszyny wirtualnej

Można na niej znaleźć poszczególne fakty dotyczące danej maszyny, takie jak jej stan, publiczny adres IP oraz nazwę DNS.

9. Nieco poniżej prawego górnego rogu strony zauważysz przycisk **Connect**. Kliknij go, by wyświetlić stronę *Connect to instance* (patrz rysunek 17.9).

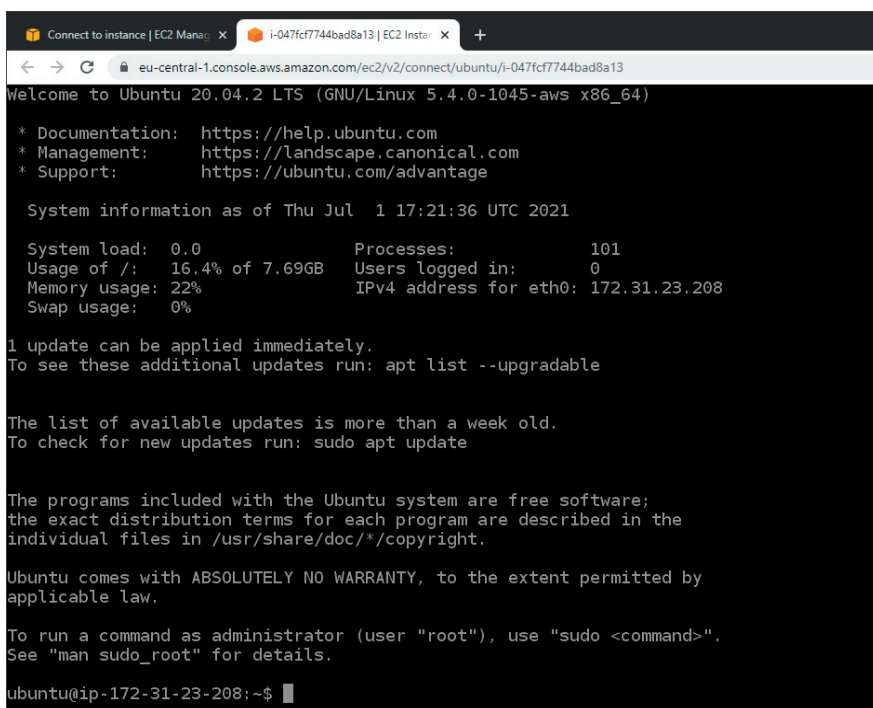


**Rysunek 17.9.** Strona nawiązania połączenia z maszyną wirtualną

Pierwsza z kart widocznych na tej stronie, *EC2 Instance Connect*, pozwala na nawiązanie połączenia przy użyciu terminala udostępnianego przez AWS. Kliknij przycisk **Connect**, a na stronie przeglądarki zostanie wyświetlone okno terminala utworzonego serwera, takie jak to przedstawione na rysunku 17.10.

Ta internetowa konsola jest rozwiązaniem awaryjnym, którego możemy używać, jeśli z jakiegoś powodu nie będzie działać połączenie SSH. W tej książce będę jednak korzystać z interfejsu SSH.

10. Wróć na stronę *Connect to instance* i przejdź na trzecią kartę, *SSH client*. Jej postać będzie podobna do tej przedstawionej poniżej, na rysunku 17.11, choć oczywiście w Twoim przypadku wyświetlone wartości będą inne.
11. Na rysunku 17.12, przedstawiłem przykład nawiązania połączenia z okna wiersza poleceń systemu Windows.



```

Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-1045-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Thu Jul  1 17:21:36 UTC 2021

System load:  0.0               Processes:            101
Usage of /:   16.4% of 7.69GB   Users logged in:     0
Memory usage: 22%              IPv4 address for eth0: 172.31.23.208
Swap usage:   0%

1 update can be applied immediately.
To see these additional updates run: apt list --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

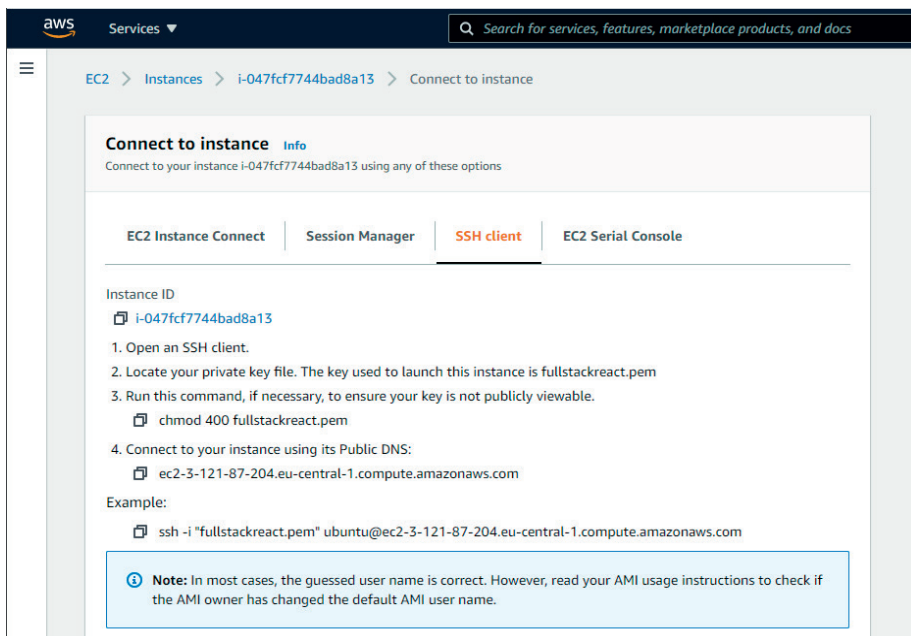
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-23-208:~$

```

Rysunek 17.10. Terminal AWS EC2 Instance Connect



aws Services ▼ Search for services, features, marketplace products, and docs

EC2 > Instances > i-047fcf7744bad8a13 > Connect to instance

### Connect to instance Info

Connect to your instance i-047fcf7744bad8a13 using any of these options

EC2 Instance Connect | Session Manager | **SSH client** | EC2 Serial Console

Instance ID  
i-047fcf7744bad8a13

1. Open an SSH client.
2. Locate your private key file. The key used to launch this instance is fullstackreact.pem
3. Run this command, if necessary, to ensure your key is not publicly viewable.  

```
chmod 400 fullstackreact.pem
```
4. Connect to your instance using its Public DNS:  

```
ec2-3-121-87-204.eu-central-1.compute.amazonaws.com
```

Example:  

```
ssh -i "fullstackreact.pem" ubuntu@ec2-3-121-87-204.eu-central-1.compute.amazonaws.com
```

**Note:** In most cases, the guessed user name is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI user name.

Rysunek 17.11. Instrukcje dotyczące nawiązania połączenia SSH

```

ubuntu@ip-172-31-23-208: ~$ ssh -i "fullstackreact.pem" ubuntu@ec2-3-121-87-204.eu-central-1.compute.amazonaws.com
C:\Users\piraj\AWS_fullstackreact\ssh -i "fullstackreact.pem" ubuntu@ec2-3-121-87-204.eu-central-1.compute.amazonaws.com
The authenticity of host 'ec2-3-121-87-204.eu-central-1.compute.amazonaws.com (3.121.87.204)' can't be established.
ECDSA key fingerprint is SHA256:5+cU03/Ux0SuJ6hYKzzUTN81lUsSkLDY5uftqSI++MI.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ec2-3-121-87-204.eu-central-1.compute.amazonaws.com,3.121.87.204' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-1045-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Thu Jul 1 17:24:13 UTC 2021

System load:  0.08               Processes:    101
Usage of /:   16.5% of 7.69GB    Users logged in: 0
Memory usage: 22%              IPv4 address for eth0: 172.31.23.208
Swap usage:   0%

1 update can be applied immediately.
To see these additional updates run: apt list --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Thu Jul 1 17:21:37 2021 from 3.120.181.42
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-23-208:~$

```

Rysunek 17.12. Pierwsze połączenie SSH

W przypadku korzystania z systemu Linux lub komputerów Mac, nawiązanie połączenia SSH może wymagać zmiany uprawnień do pliku klucza (*.pem*), zgodnie z instrukcjami opisanymi na karcie *SSH client* portalu AWS. Następnie wystarczy nawiązać połączenie, także zgodnie z opisanymi instrukcjami. Zwróć uwagę na to, że nazwą użytkownika, jaką trzeba podać podczas logowania, jest *ubuntu*; Twoja maszyna wirtualna powinna wymagać podania tej samej nazwy. Pamiętaj także, by podać prawidłową nazwę DNS swojego serwera.

Jeśli powyższe czynności nie wystarczą do nawiązania połączenia z maszyną wirtualną, zmień ustawienia sieciowe dla ruchu przychodzącego dla SSH na wartość *Source Anywhere*. Jeśli także to nie pozwoli Ci nawiązać połączenia, skorzystaj z terminala internetowego dostępnego w portalu AWS, który przedstawiłem nieco wcześniej.

W ten sposób zakończyliśmy proces tworzenia maszyny wirtualnej z systemem Ubuntu. Kolejnym krokiem będzie zainstalowanie na niej Redisa.

## Instalacja Redisa, Postgresa i Node w systemie Ubuntu

W tym podrozdziale zainstalujemy kluczowe oprogramowanie, który musi się znaleźć na naszym linuksowym serwerze. Instalowanie i konfiguracja Redisa zostały już opisane w rozdziale 13., pt. „Przygotowywanie stanu sesji przy użyciu Expressa i Redisa”, ale zrobimy to jeszcze raz, gdyż tym razem będziemy mieć ten sam system operacyjny.

## Instalacja serwera Redis

W tym podrozdziale zainstalujemy serwer Redis i przygotujemy go do użycia w naszej aplikacji:

1. Otwórz okno wiersza poleceń lub terminala, nawiąż połączenie z serwerem i wykonaj dwa poniższe polecenia:

```
sudo apt update
sudo install redis-server
```

Apt jest menedżerem zależności pakietów dla różnych dystrybucji Linuksa, w tym dla Ubuntu i Debiana. W zasadzie można by go porównać z programem NPM. W tych dwóch poleceniach najpierw aktualizujemy apt do najnowszej wersji, a następnie instalujemy serwer Redis.

2. Po zakończeniu instalowania Redisa otwórz plik konfiguracyjny *redis.conf*, wykonując w tym celu następujące polecenie:

```
sudo nano /etc/redis/redis.conf
```

3. Odszukaj w pliku ustawienie `requirepass`, usuń poprzedzający je komentarz i podaj własne hasło.

Hasło podane w kodach źródłowych aplikacji, w pliku *super-forum-server/dev-config/.env*, w zmiennej `REDIS_PASSWORD`, musi odpowiadać hasłu podanemu w pliku *redis.conf*. Pliki, które mają się znaleźć w katalogu *dev-config* przedstawię później, podczas opisywania procesu wdrażania aplikacji.

4. Następnie znajdź w pliku ustawienie `supervised` i zmień jego wartość na `systemd`. W ten sposób to Ubuntu będzie zarządzać uruchamianiem Redisa przy użyciu swojego systemu `init`, korzystającego z polecenia `systemctl`. Teraz zapisz plik i wyjdź z edytora.
5. Ponownie uruchom Redisa, aby uwzględnił on nowe ustawienia:

```
sudo systemctl restart redis.service
```

Jeśli będziesz chciał zatrzymać usługę Redisa, wykonaj poniższe polecenie:

```
sudo systemctl stop redis.service
```

Z kolei poniżej podałę polecenie, które pozwoli uruchomić usługę Redisa:

```
sudo systemctl start redis.service
```

6. Jeśli wykonasz poniższe polecenie, wyświetli ono informacje o tym, czy serwer Redisa działa prawidłowo:

```
sudo systemctl status redis
```

Wyniki, które ono wyświetli, powinny przypominać te z rysunku 17.13.

W tym podrozdziale zainstalowaliśmy w systemie Ubuntu Redisa i zapewniliśmy sobie możliwość włączania go i wyłączania zależnie od potrzeb. Kolejnym krokiem przygotowywania naszego serwera będzie zainstalowanie Postgresa.

```

ubuntu@ip-172-31-23-208: ~
redis-server.service - Advanced key-value store
  Loaded: loaded (/lib/systemd/system/redis-server.service; enabled; vendor preset: enabled)
  Active: active (running) since Thu 2021-07-01 18:09:48 UTC; 4min 44s ago
    Docs: http://redis.io/documentation,
           man:redis-server(1)
  Process: 14258 ExecStart=/usr/bin/redis-server /etc/redis/redis.conf (code=exited, status=0/SUCCESS)
 Main PID: 14270 (redis-server)
   Tasks: 4 (limit: 1160)
  Memory: 1.9M
    CGroup: /system.slice/redis-server.service
            └─14270 /usr/bin/redis-server 127.0.0.1:6379

Jul 01 18:09:48 ip-172-31-23-208 systemd[1]: Starting Advanced key-value store...
Jul 01 18:09:48 ip-172-31-23-208 systemd[1]: redis-server.service: Can't open PID file /run/redis/redis-server.pid (yet?) after start
Jul 01 18:09:48 ip-172-31-23-208 systemd[1]: Started Advanced key-value store.
~
~
~
lines 1-15/15 (END)

```

Rysunek 17.13. Status Redisa

## Instalacja Postgresa

Poniższe czynności pozwolą ci zainstalować Postgresa, z którego będzie korzystać nasza aplikacja.

1. Także tym razem skorzystamy z narzędzia apt. Wykonaj następujące polecenie:

```
sudo apt install postgresql
```

2. Po zakończeniu instalacji sprawdź, czy Postgres działa, wykonując w tym celu następujące polecenie (patrz rysunek 17.14):

```

ubuntu@ip-172-31-23-208: ~
ubuntu@ip-172-31-23-208:~$ sudo -i -u postgres
postgres@ip-172-31-23-208:~$

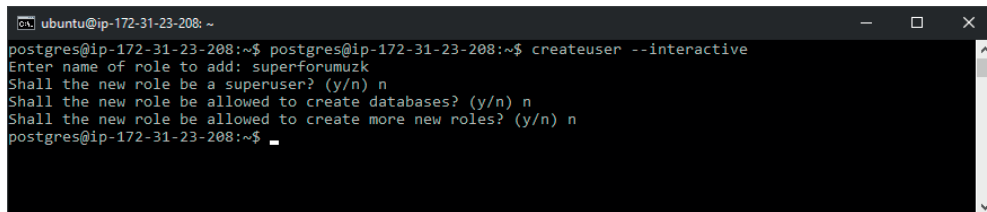
```

Rysunek 17.14. Wykonanie polecenia psql

Zastosowana w tym poleceniu rola postgres jest globalnym kontem administratora, tworzonym domyślnie w bazie Postgres. Dzięki zastosowaniu opcji `-i` powyższe polecenie sprawia, najprościej rzecz ujmując, że konto zalogowanego użytkownika będzie tymczasowo działać jako konto użytkownika postgres. Opcja `-u` określa, jakiej roli należy użyć.

Nie używamy aplikacji pgAdmin, gdyż program psql obsługiwany z poziomu wiersza poleceń daje te same możliwości, a uruchomienie aplikacji pgAdmin na wirtualnej maszynie AWS jest kłopotliwe i trudne.

3. Innymi słowy, teraz działamy jako użytkownik postgres@<twój IP>, co widać na rysunku 17.14. Gdybyśmy nie korzystali z konta Postgres, to każde z poleceń wydawanych bazie musielibyśmy poprzedzać prefiksem `sudo -u postgres`. Natomiast dzięki użyciu roli Postgresa, możemy je wykonywać tak, jak pokazałem na rysunku 17.15.



```

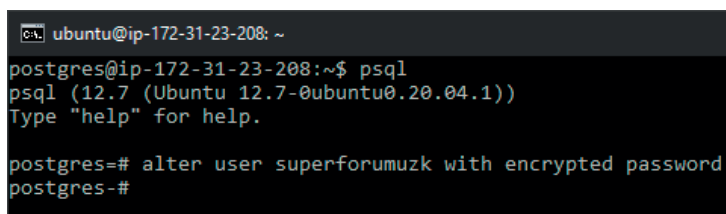
ubuntu@ip-172-31-23-208: ~
postgres@ip-172-31-23-208:~$ postgres@ip-172-31-23-208:~$ createuser --interactive
Enter name of role to add: superforumuzk
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
postgres@ip-172-31-23-208:~$

```

**Rysunek 17.15.** Tworzenie nowego użytkownika, polecenie createuser

Na rysunku 17.15 pokazałem, jak utworzyć nowego użytkownika, o nazwie superforumuzk.

4. Kolejne polecenie pozwoli określić hasło tego nowego użytkownika (patrz rysunek 17.16):



```

ubuntu@ip-172-31-23-208: ~
postgres@ip-172-31-23-208:~$ psql
psql (12.7 (Ubuntu 12.7-0ubuntu0.20.04.1))
Type "help" for help.

postgres=# alter user superforumuzk with encrypted password
postgres=#

```

**Rysunek 17.16.** Określanie hasła nowego użytkownika bazy Postgres

W pierwszej kolejności uruchomiłem klienta Postgresa, obsługiwanego z poziomu wiersza poleceń, czyli psql. Następnie wydałem polecenie, które zmienia hasło użytkownika superforumuzk.

Zwróć uwagę na to, że polecenie przedstawione na rysunku 17.16 jest ucięte za słowem kluczowym password — należy za nim podać hasło zapisane pomiędzy znakami apostrofu, na przykład: '<twoje hasło>'. Oczywiście postać hasła powinienś określić samemu.

5. Teraz przygotujemy bazę danych dla aplikacji. W pierwszej kolejności wyjdź z programu psql, a następnie utwórz bazę danych, wykonując poniższe polecenie:

```

\q
createdb -O superforumuzk SuperForum

```

Wykonanie tego polecenia spowoduje utworzenie bazy danych, której właścicielem będzie rola superforumuzk.

6. Teraz spróbujmy dodać do bazy danych domyślne wartości encji ThreadCategory. W katalogu projektu *super-forum-server*, w podkatalogu *utils*, znajdziesz plik *InsertThreadCategories.txt*. Znajdziesz w nim polecenia SQL wstawiające do bazy danych używane wcześniej kategorie. Oczywiście, możesz spróbować dodać także swoje własne kategorie. Poniżej, na rysunku 17.17, przedstawiłem efekty próby wykonania tych poleceń w bazie danych.

```

ubuntu@ip-172-31-23-208: ~
postgres@ip-172-31-23-208:~$ psql
psql (12.7 (Ubuntu 12.7-0ubuntu0.20.04.1))
Type "help" for help.

postgres=# \c SuperForum
You are now connected to database "SuperForum" as user "postgres".
SuperForum=# INSERT INTO "ThreadCategories"(
SuperForum=# "Name", "Description")
SuperForum=# VALUES ('Programowanie', '');
(
"ERROR: relation "ThreadCategories" does not exist
LINE 1: INSERT INTO "ThreadCategories"(
               ^
SuperForum=#
SuperForum=# INSERT INTO "ThreadCategories"(
SuperForum=# "Name", "Description")
SuperForum=# VALUES ('Gotowanie', '');
, '');

INSERT INTO "ThreadCategories"(
"Name", "Description")
VALUES ('Finanse', '');

INSERT INTO "ERROR: relation "ThreadCategories" does not exist
LINE 1: INSERT INTO "ThreadCategories"(
               ^

```

Rysunek 17.17. Wstawianie kategorii

Jak widać, próba wykonania tych poleceń zakończyła się niepowodzeniem. Spróbujmy dokładniej przyjrzeć się temu, co próbowaliśmy zrobić. Przede wszystkim musimy działać w odpowiedniej bazie danych. W programie `psql` można ją wybrać, używając polecenia `\c`. Zwróć uwagę na to, że w nazwach baz danych wielkość liter ma znaczenie. Poza tym, musisz zwrócić uwagę na to, czy nazwy tabel i pól są zapisywane w cudzysłowach. W przypadku wykonywania poleceń SQL w programie `psql` poprzedzanie nazw tabel prefiksem `public.` nie jest konieczne; trzeba go używać wyłącznie w aplikacji `pgAdmin`.

W ten sposób uruchomiliśmy serwer bazy danych PostgreSQL i utworzyliśmy w nim pustą bazę danych dla naszej aplikacji. Kolejnym krokiem będzie zainstalowanie środowiska Node.

## Instalacja Node

Teraz zajmijmy się instalacją środowiska Node.

1. Wykonaj następujące polecenie:

```
sudo apt get install nodejs
```

2. Po zainstalowaniu Node wykonaj to polecenie, by sprawdzić jego wersję:

```
node -v
```

Zainstalowana wersja Node musi być *większa lub równa* 12. Jeśli jest mniejsza, to wykonaj następujące polecenie:

```
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
```

A następnie jeszcze to:

```
sudo apt-get install -y nodejs
```



3. Następnie zainstaluj NPM, wykonując w tym celu następujące polecenie:

```
sudo apt install npm
```

4. Teraz musimy zainstalować narzędzie, które pozwoli nam zarządzać serwerem Node, czyli wyłączyć go i automatycznie restartować. Do tego celu użyjemy programu pm2, który obecnie stanowi jeden z najbardziej popularnych sposobów zarządzania Node. Zwróć uwagę na zastosowanie przełącznika -g, który instaluje wybrane pakiety globalnie.

```
sudo npm install -g pm2
```

W tym podrozdziale opisałem sposób instalacji podstawowych usług, z których będzie korzystać nasza aplikacja: Redisa, Postgresa oraz środowiska Node. Teraz jesteśmy gotowi, by zacząć konfigurować faktyczny serwer, który stworzymy w oparciu o NGINX.

## Konfiguracja i wdrażanie aplikacji na serwerze NGINX

W tym podrozdziale zainstalujemy i skonfigurujemy naszą aplikację, która będzie działać o oparciu o serwer NGINX. NGINX to bardzo popularny i niezwykle wydajny serwer WWW, odwrotny serwer pośredniczący (ang. *reverse proxy*) oraz serwer do równoważenia obciążenia. Cieszy się on znakomitą opinią ze względu na doskonałą wydajność oraz możliwość obsługi różnych konfiguracji witryn z wykorzystaniem wielu serwerów.

My będziemy używać NGINX do obsługi dwóch witryn. Pierwsza z nich będzie udostępniać naszą kliencką aplikację Reacta, a druga serwer GraphQL Express. Cały ruch kierowany do naszej aplikacji najpierw będzie trafiał na serwer NGINX, który następnie będzie kierował odbierane żądania do odpowiednich części aplikacji. Zacznijmy zatem od zainstalowania serwera NGINX.

1. Nawiąż połączenie SSH z serwerem, jak pokazałem wcześniej, w podrozdziale pt. „Konfiguracja Ubuntu w chmurze AWS”. Następnie wykonaj poniższe polecenia, by zainstalować NGINX:

```
sudo apt update
sudo apt install nginx
```

2. Kiedy serwer NGINX zostanie już zainstalowany, wykonaj poniższe polecenia, by utworzyć katalogi na kliencką i serwerową część naszej aplikacji:

```
sudo mkdir /var/www/superforum
sudo mkdir /var/www/superforum/server
```

Katalog `/var/www` jest domyślnym miejscem przechowywania plików udostępnianych przez serwery WWW, co zresztą wyraźnie sugeruje jego nazwa.

## Konfiguracja projektu super-forum-server

W tym punkcie przygotujemy proces budowania i wdrażania kodu serwerowej części naszej aplikacji. Opracowanie standardowego procesu wdrażania jest bardzo przydatne, gdyż dzięki niemu wykonywane przez nas wdrożenia będą mogły być spójne i niezawodne.

1. Zanim zaczniemy kopiowanie plików, musimy przygotować wstępną, podstawową konfigurację i sposób budowania projektu serwerowej części aplikacji. Otwórz projekt *super-forum-server* w edytorze VSCode. Jeśli teraz otworzysz plik *package.json*, zauważysz w nim nowy skrypt: *build*. Będzie on kompilował kod serwerowej części aplikacji, pakował go w sposób nadający się do dystrybucji i zapisywał w katalogu *dist*. Jednak, aby to polecenie mogło działać, będziemy jeszcze musieli zainstalować globalnie dwa pakiety NPM. Wykonaj poniższe polecenie na swoim komputerze roboczym, *nie na wirtualnej maszynie, na której działa serwer Ubuntu*:

```
sudo npm i -g del-cli cpy-cli
```

Pakiet *del-cli* jest uniwersalnym poleceniem *delete*, wykonywanym z poziomu wiersza poleceń. Oznacza to, że niezależnie od tego, jakiego systemu operacyjnego używamy na naszym komputerze roboczym — Linux, Mac, czy Windows — to polecenie zawsze będzie działać tak samo. Podobnie polecenie *cpy-cli* zapewnia uniwersalny sposób kopiowania plików i katalogów. Użyjemy tych poleceń, byśmy mogli przygotować jeden skrypt NPM, który będzie działał tak samo we wszystkich systemach operacyjnych.

Przejdźmy teraz do samego skryptu. W pierwszej kolejności usuwa on całą zawartość katalogu *dist*, by stan początkowy procesu budowy zawsze był taki sam. Następnie kopiuje zawartość katalogu *dev-config* oraz plik *.env* do katalogu *dist*. I w końcu skrypt uruchamia kompilator TypeScriptu.

Zawróciłeś zapewne uwagę na to, że w projekcie serwerowej części aplikacji pojawił się nowy katalog, *dev-config*. Będzie on zawierał pliki związane z konfiguracją, które w końcu zostaną skopiowane przez skrypt *build* do katalogu *dist*. W tym nowym katalogu *dev-config* znajdują się następujące pliki: *.env* z globalnymi ustawieniami konfiguracyjnymi, *ormconfig.js* zawierający konfigurację TypeORM, oraz plik *package.json*.

Plik *.env* zapisany w katalogu *dev-config* musi zawierać ustawienia konfiguracyjne, które zapewnią prawidłowe działanie aplikacji *na Twoim* serwerze. Dotyczy to takich informacji jak: hasła, nazwy kont oraz adresy IP. Wszystkie te ustawienia muszą zostać prawidłowo podane i dostosowane do konfiguracji *Twojego* systemu. Jeśli podczas uruchamiania serwera wystąpią jakieś problemy, to w pierwszej kolejności powinieneś zajrzeć właśnie do tego pliku.

2. Niestety, okazuje się, że w najnowszej wersji pakietu NPM Express występują jakieś problemy, które zmuszają nas do zainstalowania jeszcze jednego, dodatkowego pakietu. W związku z tym wykonaj jeszcze poniższe polecenie na swoim komputerze roboczym:

```
npm i -D @types/express-serve-static-core
```

Ta zależność została już pobrana podczas instalowania `@types/express`, jednak powyższe polecenie zagwarantuje, że na komputerze będzie dostępna jego najnowsza wersja. Gdybyś chciał dowiedzieć się czegoś więcej na temat tego błędu, to informacje o nim możesz znaleźć na stronie <https://github.com/DefinitelyTyped/DefinitelyTyped/issues/47339>.

3. Zwróć uwagę na jeszcze jedną sprawę. Otóż w projekcie serwerowej części aplikacji, w pliku `super-forum-server/src/index.ts`, blisko jego początku, dodałem nową funkcję, `loadEnv`. Obsługuje ona różnice we względnych ścieżkach dostępu do pliku `.env` na komputerze roboczym oraz na serwerze i korzysta ze zmiennej Node o nazwie `__dirname`.

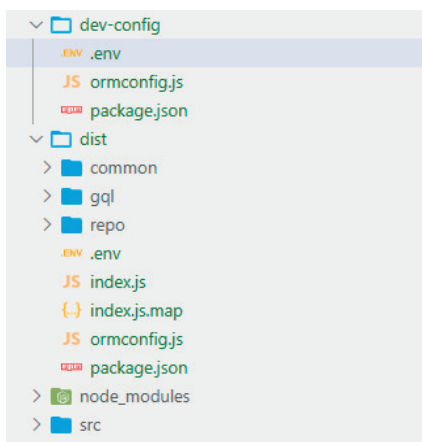
Zmodyfikowałem także plik `super-forum-server/dev-config/ormconfig.js` i zastosowałem zmienną `__dirname` w ścieżce do encji TypeORM.

Przypisałem także polu `synchronize` w pliku `ormconfig.js` wartość `true`. Ustawienie to będzie używane wyłącznie podczas tworzenia wdrożeń w środowiskach roboczych. Nie należy go używać w środowiskach produkcyjnych, gdyż może powodować niepożądane zmiany w bazie danych. W środowiskach produkcyjnych należy używać z góry przygotowanej bazy danych i wdrażać ją bezpośrednio po uprzednim przypisaniu polu `synchronize` wartości `false`.

4. No dobrze, to teraz spróbujmy wykonać skrypt do zbudowania aplikacji. Wykonaj poniższe polecenie na swoim komputerze roboczym:

```
npm run build
```

Powinno to spowodować utworzenie katalogu `dist`, widocznego na rysunku 17.18.



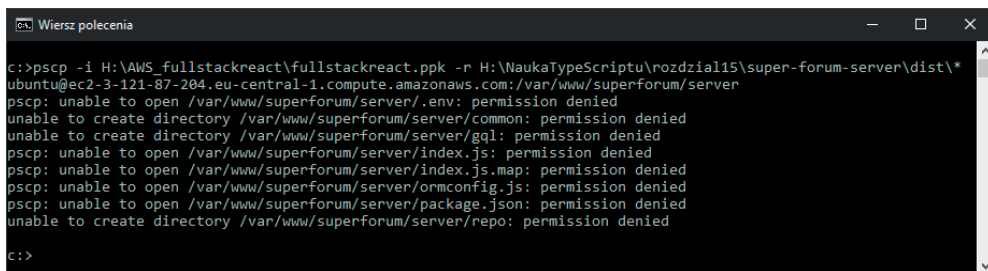
Rysunek 17.18. Katalog `dist`

5. Teraz spróbuj skopiować katalog `dist` na serwer. W oknie wiersza poleceń na swoim komputerze roboczym wykonaj poniższe polecenie, wpisując w nim dane dostosowane do swojego systemu roboczego i serwera Ubuntu:

```
scp -i <ścieżka_pliku_pem> -r <katalog_źródłowy>/*
<nazwa_użytkownika>@<adres_ip>:<katalog_docelowy>
```

Pamiętaj, że w systemie Windows będziesz musiał użyć polecenia `pscp`<sup>1</sup>.

Kiedy wykonasz to polecenie na swoim roboczym komputerze, w oknie wiersza poleceń pojawią się komunikaty takie jak te przedstawione na rysunku 17.19.



```
c:\>pscp -i H:\AWS_fullstackreact\fullstackreact.ppk -r H:\NaukaTypeScriptu\rozdzial15\super-forum-server\dist\*
ubuntu@ec2-3-121-87-204.eu-central-1.compute.amazonaws.com:/var/www/superforum/server
pscp: unable to open /var/www/superforum/server/.env: permission denied
unable to create directory /var/www/superforum/server/common: permission denied
unable to create directory /var/www/superforum/server/gql: permission denied
pscp: unable to open /var/www/superforum/server/index.js: permission denied
pscp: unable to open /var/www/superforum/server/index.js.map: permission denied
pscp: unable to open /var/www/superforum/server/orconfig.js: permission denied
pscp: unable to open /var/www/superforum/server/package.json: permission denied
unable to create directory /var/www/superforum/server/repo: permission denied
c:\>
```

Rysunek 17.19. Nieudana próba skopiowania katalogu `dist`

Jak widać, próba skopiowania plików zakończyła się niepowodzeniem. Problemy te są spowodowane brakiem odpowiednich praw dostępu do katalogu na serwerze Ubuntu. W kolejnym punkcie rozwiążemy ten problem.

6. Zaloguj się na serwer Ubuntu przy użyciu SSH i wykonaj następujące polecenie:

```
sudo chmod -R 777 /var/www/superforum/server
```

To polecenie zapewni chwilowo pełen dostęp do wskazanego katalogu, dzięki czemu będziemy mogli skopiować do niego pliki. Kiedy pliki zostaną już skopiowane, ponownie zmienimy prawa dostępu do katalogu, eliminując zagrożenie dla bezpieczeństwa serwera.

7. Teraz ponownie spróbuj skopiować pliki z katalogu `dist`, wykonując w tym celu polecenie `scp` w oknie wiersza poleceń na komputerze roboczym. Poniżej, na rysunku 17.20, pokazałem, jak wyglądają wyniki wykonania tego polecenia po zmianie praw dostępu do katalogu docelowego.
8. Następnie upewnij się, że wszystkie pliki konfiguracyjne zostały skopiowane na serwer; w tym celu wyświetl zawartość katalogu `server` — powinna ona wyglądać jak na rysunku 17.21.

<sup>1</sup> Program `pscp` nie obsługuje kluczy prywatnych SSH-2 zapisanych w formacie OpenSSH, a właśnie taki klucz jest generowany i pobierany podczas tworzenia maszyny wirtualnej w chmurze AWS. Dlatego, zanim zaczniemy kopiować pliki na maszynę wirtualną przy użyciu `pscp`, konieczne będzie skonwertowanie klucza do formatu Putty Private Key (PPK) przy użyciu programu PuTTYgen. Po uruchomieniu tego programu należy kliknąć przycisk *Load*, wczytać zapisany wcześniej plik klucza *.pem*, a następnie zapisać go w formacie PPK, klikając przycisk *Save private key* — *przyp. tłum.*

```

c:\>pscp -i H:\AWS_fullstackreact\fullstackreact.ppk -r H:\NaukaTypeScriptu\rozdzial15\super-forum-server\dist\*
ubuntu@ec2-3-121-87-204.eu-central-1.compute.amazonaws.com:/var/www/superforum/server
.env                               0 kB | 0.4 kB/s | ETA: 00:00:00 | 100%
dates.js                          0 kB | 0.8 kB/s | ETA: 00:00:00 | 100%
dates.js.map                      0 kB | 0.7 kB/s | ETA: 00:00:00 | 100%
envLoader.js                     0 kB | 0.6 kB/s | ETA: 00:00:00 | 100%
envLoader.js.map                 0 kB | 0.5 kB/s | ETA: 00:00:00 | 100%
EmailValidator.js                0 kB | 0.5 kB/s | ETA: 00:00:00 | 100%
EmailValidator.js.map           0 kB | 0.4 kB/s | ETA: 00:00:00 | 100%
PasswordValidator.js            0 kB | 0.9 kB/s | ETA: 00:00:00 | 100%
PasswordValidator.js.map        0 kB | 0.7 kB/s | ETA: 00:00:00 | 100%
ThreadValidators.js             0 kB | 0.9 kB/s | ETA: 00:00:00 | 100%
ThreadValidators.js.map         0 kB | 0.8 kB/s | ETA: 00:00:00 | 100%
GqlContext.js                   0 kB | 0.1 kB/s | ETA: 00:00:00 | 100%
GqlContext.js.map               0 kB | 0.1 kB/s | ETA: 00:00:00 | 100%
resolvers.js                    11 kB | 11.2 kB/s | ETA: 00:00:00 | 100%
resolvers.js.map                7 kB | 7.6 kB/s | ETA: 00:00:00 | 100%
typeDefs.js                     3 kB | 3.1 kB/s | ETA: 00:00:00 | 100%
typeDefs.js.map                 0 kB | 0.3 kB/s | ETA: 00:00:00 | 100%
index.js                        3 kB | 3.2 kB/s | ETA: 00:00:00 | 100%
index.js.map                    1 kB | 1.9 kB/s | ETA: 00:00:00 | 100%
ormconfig.js                   0 kB | 0.5 kB/s | ETA: 00:00:00 | 100%

```

Rysunek 17.20. Kopiowanie plików przy użyciu pscp

```

ubuntu@ip-172-31-23-208: ~
ubuntu@ip-172-31-23-208:~$ ls -la /var/www/superforum/server/
total 40
drwxrwxrwx 5 root root 4096 Jul 29 19:31 .
drwxr-xr-x 4 root root 4096 Jul 29 19:24 ..
-rw-rw-r-- 1 ubuntu ubuntu 401 Jul 29 19:30 .env
drwxrwxr-x 3 ubuntu ubuntu 4096 Jul 29 19:30 common
drwxrwxr-x 2 ubuntu ubuntu 4096 Jul 29 19:31 gql
-rw-rw-r-- 1 ubuntu ubuntu 3296 Jul 29 19:31 index.js
-rw-rw-r-- 1 ubuntu ubuntu 1910 Jul 29 19:31 index.js.map
-rw-rw-r-- 1 ubuntu ubuntu 539 Jul 29 19:31 ormconfig.js
-rw-rw-r-- 1 ubuntu ubuntu 1146 Jul 29 19:31 package.json
drwxrwxr-x 2 ubuntu ubuntu 4096 Jul 29 19:31 repo
ubuntu@ip-172-31-23-208:~$

```

Rysunek 17.21. Sprawdzenie zawartości katalogu na serwerze

Jeśli w skopiowanych na serwer plikach nie będzie pliku `.env`, to będziesz musiał skopiować go ręcznie, wykonując w tym celu poniższe polecenie. Problem ten dotyczy wyłącznie komputerów Mac, na których pliki `.env` z jakiegoś powodu są niewidoczne:

```
scp -i <ścieżka_pliku_pem> -r <katalog_źródłowy>/./env
<nazwa_użytkownika>@<adres_ip>:<katalog_docelow>/./env
```

Także w tym przypadku konkretne postaci ścieżek będą unikalne dla Twojego systemu.

Teraz powinieneś przywrócić wcześniejsze prawa dostępu do katalogu; w tym celu wykonaj poniższe polecenie:

```
sudo chmod -R 755 /var/www/superforum/server
```

Te uprawnienia nadają pełne prawa dostępu właścicielowi, natomiast wszyscy pozostali będą dysponowali jedynie prawem do odczytu i wykonania.

Gdybyśmy wpadli w czarną dziurę problematyki optymalizacji zabezpieczeń, musiałbyśmy napisać zupełnie nową książkę. Jednak, ponieważ Twój serwer produkcyjny zostanie zapewne niebawem usunięty, możemy się skoncentrować na głównych zagadnieniach. Kiedy już będziesz gotów, by udostępnić produkcyjną wersję aplikacji, która będzie Ci miała przynieść miliony dolarów, będziesz musiał gruntownie podszkolić się w zakresie bezpieczeństwa serwerów, albo jeszcze lepiej, wynająć specjalistę z tego zakresu z przynajmniej 10-letnim doświadczeniem.

9. Teraz, w oknie, w którym masz nawiązane połączenie SSH z serwerem Ubuntu, przejdź do katalogu `/var/www/superforum/server` i wykonaj następujące polecenie:

```
npm install
```

Oczywiście, to polecenie zainstaluje zależności niezbędne do działania naszej serwerowej części aplikacji.

10. Kolejnym krokiem będzie uruchomienie systemu pm2, by kontrolował on działanie serwera Node. Wykonaj następujące polecenie:

```
pm2 startup
```

Polecenie to poinformuje o konkretnych ustawieniach, jakie musi mieć bieżący użytkownik, by móc skonfigurować system pm2 do korzystania z systemct1 i uruchamiania serwera Node podczas startowania serwera Ubuntu. Po wykonaniu tego polecenia powinieneś zobaczyć komunikat podobny do tego z rysunku 17.22.

```
[PM2] Init System found: systemd
[PM2] To setup the Startup Script, copy/paste the following command:
sudo env PATH=$PATH:/usr/bin:/usr/lib/node_modules/pm2/bin/pm2 startup systemd -u ubuntu --hp /home/ubuntu
```

#### Rysunek 17.22. Uruchamianie pm2

11. A zatem, skopiuj to całe polecenie zaczynające się od `sudo`, a następnie wklej i wykonaj je w oknie, w którym masz nawiązane połączenie SSH z serwerem Ubuntu. Po uruchomieniu zostanie wyświetlony ekran podobny do tego z rysunku 17.23.
12. Teraz musimy uruchomić serwer Node. Zrób to w sposób przedstawiony na rysunku 17.24.
13. Teraz możemy zapisać ten proces jako element listy uruchomieniowej pm2. W tym celu należy użyć polecenia:

```
pm2 save
```

Kiedy je wykonasz, na ekranie powinien zostać wyświetlony następujący komunikat (patrz rysunek 17.25).

Dzięki zapisaniu tego procesu, nasz serwer Node będzie automatycznie uruchamiany podczas restartowania serwera Ubuntu.

•

1

---

## Konfigurowanie projektu *super-forum-client*

No dobrze, teraz musimy wykonać podobny proces, aby uruchomić na serwerze Ubuntu kliencką część naszej aplikacji. Sam projekt *super-forum-client* powinien już być skopiowany w katalogu *rozdział17*, gdyż była to pierwsza rzecz, jaką zrobiliśmy w tym rozdziale.

1. Wróć do okna, w którym masz nawiązane połączenie SSH z serwerem Ubuntu, i utwórz katalog na projekt klienckiej części aplikacji:

```
sudo mkdir /var/www/superforum/client
```

2. Teraz wróć do okna wiersza poleceń na swoim komputerze roboczym, otworzonego w katalogu projektu *super-forum-client*, gdyż musisz zbudować i wdrożyć ten projekt. Zanim jednak to zrobisz, musisz wprowadzić do projektu pewną drobną zmianę. Jak wiadomo, nasz projekt serwerowej części aplikacji używa pliku *.env* do przechowywania ustawień. W przypadku projektu aplikacji klienckiej, tak rozbudowane ustawienia nie są nam potrzebne. Niemniej jednak będziemy potrzebować przynajmniej możliwości określenia adresu URL serwera GraphQL i dostosowania go do ustawień środowiska produkcyjnego. Dlatego wykonaj opisane niżej czynności.

- W edytorze VSCode otwórz plik *index.tsx* i zaktualizuj fragment kodu tworzący obiekt *ApolloClient* zgodnie z poniższym przykładem:

```
const client = new ApolloClient({
  uri: process.env.REACT_APP_GQL_URL,
  credentials: "include",
  cache: new InMemoryCache({
    resultCaching: false,
  }),
});
```

Jak widać, dodaliśmy do niego zmienną środowiskową *REACT\_APP\_GQL\_URL*, podobnie jak wcześniej zrobiliśmy w serwerowej części aplikacji. Jednak skąd będzie pochodzić wartość tej właściwości? Wyjaśnię to w kolejnym punkcie.

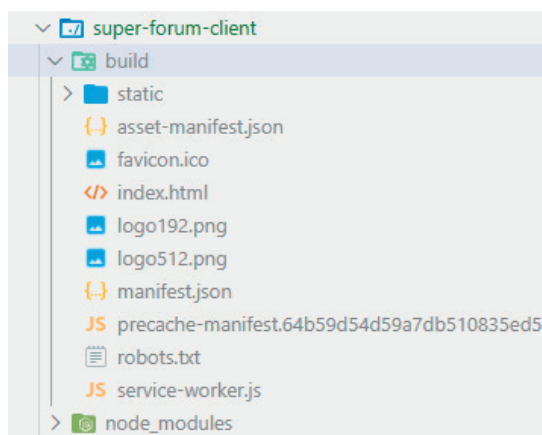
- Otwórz plik *package.json* i przyjrzyj się jego sekcji *scripts*. Powinieneś zobaczyć w niej nowy skrypt, o nazwie *build-dev*, który określa wartość zmiennej *REACT\_APP\_GQL\_URL*. Nic nie stoi na przeszkodzie, byś utworzył więcej wersji tego skryptu, podając w każdej z nich inną wartość zmiennej.

3. Teraz możesz już uruchomić skrypt *build-dev*:

```
npm run build-dev
```

Skrypt budowania (*build*) w projekcie klienckiej części naszej aplikacji został przygotowany podczas tworzenia aplikacji przy użyciu narzędzia *create-react-app*, jednak zmodyfikowaliśmy nieco jego postać, dodając do niego zmienną środowiskową. Kiedy go wykonasz, po zakończeniu procesu budowania zawartość katalogu *build* powinna być taka, jak na rysunku 17.26.





Rysunek 17.26. Katalog build w katalogu klienckiej części aplikacji (super-forum-client)

Na temat zmiennych środowiskowych Reacta warto wiedzieć dwie rzeczy:

1. Nazwy zmiennych środowiskowych Reacta muszą się zaczynać od prefiksu `REACT_` ➔ `APP_`. W innym przypadku zmienna zostanie zignorowana.
2. Zmienne środowiskowe są wstawiane do kodu na etapie budowania aplikacji, więc ich wartości będą wdrażane jako część kodu aplikacji klienckiej. A to oznacza, że użytkownicy będą w stanie przeszukać skrypty pobierane przez przeglądarkę i poznać wartości tych zmiennych. Dlatego też w tych zmiennych środowiskowych **nigdy** nie należy podawać wrażliwych informacji.

4. Teraz musisz zmienić prawa dostępu do katalogu klienckiej części aplikacji na serwerze, abyś mógł skopiować do niego kody. W tym celu wykonaj poniższe polecenie:

```
sudo chmod -R 777 /var/www/superforum/client
```

5. Teraz możesz wdrożyć zbudowane pliki klienckiej części aplikacji. Z poziomu okna wiersza poleceń na swoim komputerze roboczym wykonaj poniższe polecenie, oczywiście podając w nim odpowiednie ścieżki:

```
scp -i <ścieżka_pliku_pem> -r <katalog_źródłowy>/* <nazwa_użytkownika>➔@<adres_ip>:<katalog_docelowy>
```

Komunikaty generowane podczas wykonywania tego polecenia przedstawiłem na rysunku 17.27.

6. Teraz przywróć wcześniejsze prawa dostępu do katalogu:

```
sudo chmod -R 755 /var/www/superforum/client
```

```

c:\>rsync -i H:\AWS_fullstackreact\fullstackreact.ppk -r H:\NaukaTypeScriptu\rozdzial15\super-forum-client\build\*
ubuntu@ec2-3-121-87-204.eu-central-1.compute.amazonaws.com: /var/www/superforum/client
asset-manifest.json      1 kB | 1.2 kB/s | ETA: 00:00:00 | 100%
favicon.ico              3 kB | 3.1 kB/s | ETA: 00:00:00 | 100%
index.html               2 kB | 2.3 kB/s | ETA: 00:00:00 | 100%
logo192.png              5 kB | 5.2 kB/s | ETA: 00:00:00 | 100%
logo512.png              9 kB | 9.4 kB/s | ETA: 00:00:00 | 100%
manifest.json            0 kB | 0.5 kB/s | ETA: 00:00:00 | 100%
precache-manifest.64b59d5 0 kB | 0.7 kB/s | ETA: 00:00:00 | 100%
robots.txt               0 kB | 0.1 kB/s | ETA: 00:00:00 | 100%
service-worker.js        1 kB | 1.2 kB/s | ETA: 00:00:00 | 100%
2.ee241de9.chunk.css     1 kB | 1.4 kB/s | ETA: 00:00:00 | 100%
2.ee241de9.chunk.css.map 2 kB | 2.6 kB/s | ETA: 00:00:00 | 100%
main.30fb33de.chunk.css  7 kB | 7.2 kB/s | ETA: 00:00:00 | 100%
main.30fb33de.chunk.css.m 12 kB | 13.0 kB/s | ETA: 00:00:00 | 100%
2.25d91fa6.chunk.js      591 kB | 591.8 kB/s | ETA: 00:00:00 | 100%
2.25d91fa6.chunk.js.LICEN 3 kB | 3.6 kB/s | ETA: 00:00:00 | 100%
2.25d91fa6.chunk.js.map  3039 kB | 3039.7 kB/s | ETA: 00:00:00 | 100%
main.2e1e5b85.chunk.js   41 kB | 41.5 kB/s | ETA: 00:00:00 | 100%
main.2e1e5b85.chunk.js.ma 114 kB | 114.7 kB/s | ETA: 00:00:00 | 100%
runtime-main.16d1e66e.js  1 kB | 1.5 kB/s | ETA: 00:00:00 | 100%
runtime-main.16d1e66e.js.  8 kB | 8.1 kB/s | ETA: 00:00:00 | 100%

```

Rysunek 17.27. Kopiowanie na server plików klienckiej części aplikacji

## Konfiguracja serwera NGINX

Wszystko idzie świetnie! Wykonaliśmy już sporo prac konfiguracyjnych na serwerze, a kolejną rzeczą, którą musimy zrobić, jest skonfigurowanie serwera NGINX:

1. Serwer NGINX musi być uruchamiany podczas uruchamiania całego systemu Ubuntu. W tym celu, w oknie, w którym masz otwórzono połączenie SSH z maszyną wirtualną Ubuntu, wykonaj polecenie przedstawione na rysunku 17.28.

```

ubuntu@ip-172-31-23-208: ~
ubuntu@ip-172-31-23-208:~$ sudo systemctl enable nginx
Synchronizing state of nginx.service with SysV service script with /lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable nginx
ubuntu@ip-172-31-23-208:~$

```

Rysunek 17.28. Włączanie uruchamiania serwera NGINX podczas uruchamiania systemu

2. Teraz wykonaj polecenie status, by sprawdzić, czy serwer NGINX działa (patrz rysunek 17.29):

```

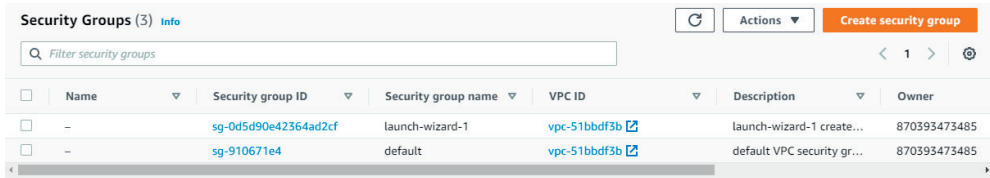
ubuntu@ip-172-31-23-208: ~
ubuntu@ip-172-31-23-208:~$ sudo systemctl status nginx
● nginx.service - A high performance web server and a reverse proxy server
   Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
   Active: active (running) since Sat 2021-07-03 11:47:27 UTC; 42min ago
     Docs: man:nginx(8)
   Main PID: 499 (nginx)
    Tasks: 2 (limit: 1159)
   Memory: 11.6M
   CGroup: /system.slice/nginx.service
           └─499 nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
             └─500 nginx: worker process

Jul 03 11:47:27 ip-172-31-23-208 systemd[1]: Starting A high performance web server and a reverse proxy server...
Jul 03 11:47:27 ip-172-31-23-208 systemd[1]: Started A high performance web server and a reverse proxy server.
ubuntu@ip-172-31-23-208:~$

```

Rysunek 17.29. Statusu serwera NGINX

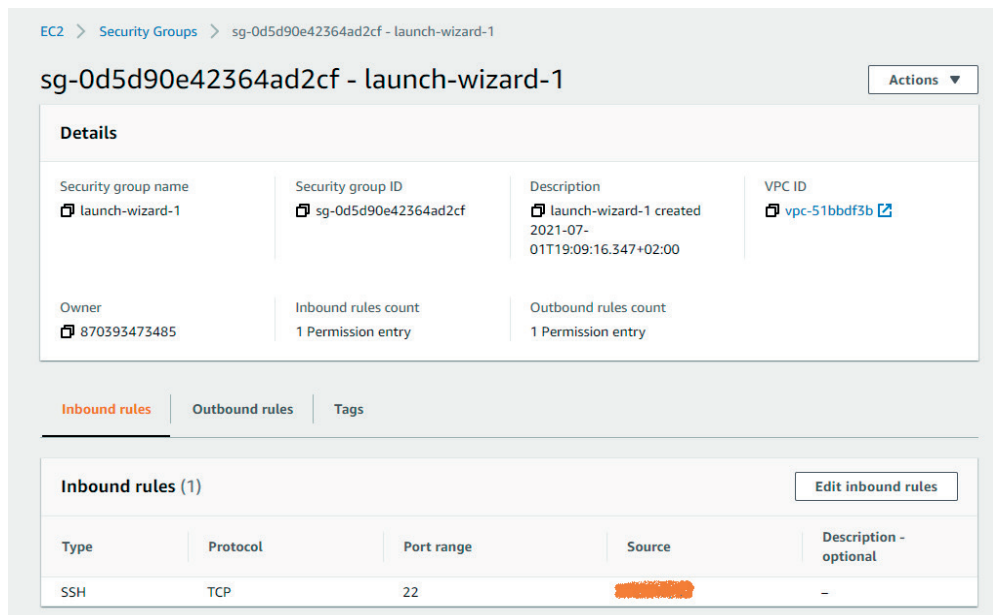
3. Teraz musisz otworzyć port 80 w zaporze sieciowej wirtualnej maszyny AWS. Otwórz przeglądarkę, wyświetl w niej portal AWS i z menu *Network & Security* wybierz opcję *Security Groups*. Na rysunku 17.30 pokazałem fragment strony, która zostanie wyświetlona:



<input type="checkbox"/>	Name	Security group ID	Security group name	VPC ID	Description	Owner
<input type="checkbox"/>	-	sg-0d5d90e42364ad2cf	launch-wizard-1	vpc-51bbdf3b	launch-wizard-1 create...	870393473485
<input type="checkbox"/>	-	sg-910671e4	default	vpc-51bbdf3b	default VPC security gr...	870393473485

Rysunek 17.30. Strona Security Groups

4. Teraz wybierz tę grupę, która nie jest domyślną (tę, która w kolumnie *Security group name* nie ma wartości *default*), a kiedy klikniesz jej identyfikator, zostanie wyświetlona strona przedstawiona na rysunku 17.31. Zwróć uwagę na wyświetloną u dołu sekcję *Inbound rules*.



sg-0d5d90e42364ad2cf - launch-wizard-1				
<b>Details</b>				
Security group name launch-wizard-1	Security group ID sg-0d5d90e42364ad2cf	Description launch-wizard-1 created 2021-07-01T19:09:16.347+02:00	VPC ID vpc-51bbdf3b	
Owner 870393473485	Inbound rules count 1 Permission entry	Outbound rules count 1 Permission entry		
<b>Inbound rules (1)</b>				
Type	Protocol	Port range	Source	Description - optional
SSH	TCP	22		-

Rysunek 17.31. Karta ustawień sieciowych, dodawanie portu połączeń przychodzących

5. Kliknij przycisk *Edit inbound rules*, a na następnej stronie kliknij przycisk *Add rule*.

Następnie dodaj nową regułę — taką samą, jak ta przedstawiona na rysunku 17.32.

**Rysunek 17.32.** Dodawanie nowej reguły przychodzących połączeń HTTP

Wybierając w kolumnie *Source* wartość **0.0.0.0/0**, zezwalamy na odbieranie żądań z dowolnych adresów IP, co jest zgodne z tym, czego chcemy. Teraz zapisz nowe ustawienia, klikając przycisk *Save rules*.

6. Domyślnie lokalna zapora sieciowa na serwerze Ubuntu nie jest włączana. Gdyby jednak była, to także w jej ustawieniach musimy zezwolić na ruch sieciowy do serwera NGINX. Jeśli to będzie konieczne, to wykonaj następujące polecenia:

```
sudo ufw allow 'Nginx HTTP'
sudo ufw status
```

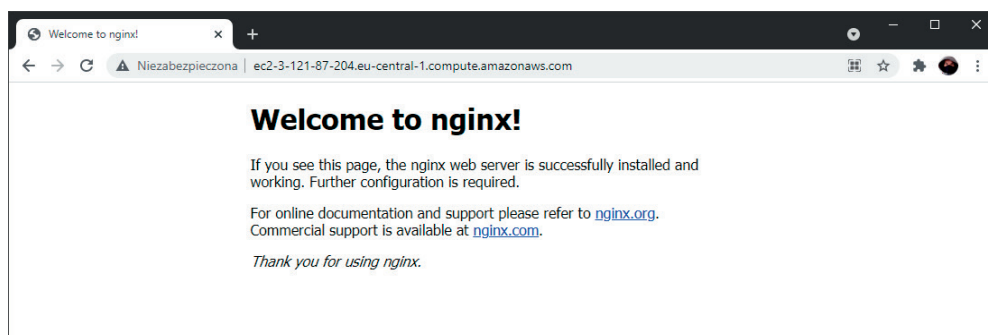
Wykonanie tego polecenia powinno wyświetlić wyniki przedstawione na rysunku 17.33.

```
ubuntu@ip-172-31-23-208: ~
ubuntu@ip-172-31-23-208:~$ sudo ufw status
Status: active

To Action From
--
Nginx HTTP ALLOW Anywhere
22/tcp ALLOW Anywhere
Nginx HTTP (v6) ALLOW Anywhere (v6)
22/tcp (v6) ALLOW Anywhere (v6)
```

**Rysunek 17.33.** Zapora sieciowa ufw z ustawieniami pozwalającymi na połączenia HTTP i SSH

Teraz możesz już przejść do okna przeglądarki i wpisać w nim adres URL swojego wirtualnego serwera Ubuntu; w moim przypadku jest to `http://ec2-3-121-87-204.eu-central-1.compute.amazonaws.com/`, jednak Twój na pewno będzie inny. W efekcie w przeglądarce powinna zostać wyświetlona strona przedstawiona na rysunku 17.34.



**Rysunek 17.34.** Domyślna strona początkowa serwera NGINX

7. Jak widać, serwer NGINX jest zainstalowany i działa. Teraz musimy skonfigurować go tak, by udostępniał naszą aplikację. Warto zwrócić uwagę na to, że w serwerze NGINX występuje błąd, który uniemożliwia korzystanie z bardzo długich nazw domen, takich jak generowana automatycznie przez chmurę AWS podczas tworzenia maszyny wirtualnej. Dlatego też w naszej aplikacji zamiast nazwy domenowej będziemy używali adresu IP.

NGINX udostępnia dwie opcje do konfigurowania witryn. Pierwsza z nich pozwala na stosowanie pliku konfiguracyjnego umieszczonego w katalogu `/etc/nginx/conf.d`. Druga, określana jako *Server Blocks*, daje możliwość korzystania z katalogu `/etc/nginx/sites-available`. My skorzystamy z pierwszej z tych metod, czyli z katalogu `conf.d`.

Wykonaj poniższe polecenie:

```
sudo nano /etc/nginx/conf.d/superforum.conf
```

8. Poniżej, na rysunku 17.35, przedstawiłem treść pliku konfiguracyjnego, którą powinienś wpisać w edytorze; pamiętaj, by podać ścieżkę i nazwę domeny (lub adres IP) dostosowane do swojego serwera Ubuntu.

#### Oto kilka informacji o tym pliku konfiguracyjnym, które warto zapamiętać:

Nie zapomnij zapisać znaku średnika na końcu każdego z wierszy. Pominięcie ich spowoduje wystąpienie błędów.

`server_name` to nazwa domenowa lub adres IP.

`root` to katalog zawierający nasze pliki HTML.

`location /` reprezentuje główny katalog witryny.

`location /graphql` określa, gdzie będzie się znajdował nasz serwer GraphQL. Używamy dyrektywy `proxy_pass`, by przekierować odwołania o postaci `http://<nazwa domenowa lub IP>/graphql` na adres `http://localhost:5000/graphql` (czyli do naszego serwera Node).

Pola `<prefiks>_timeout` mają zapobiegać występowaniu błędów 503 Gateway Timeout, które czasami pojawiają się w serwerze NGINX.

```

GNU nano 4.8 /etc/nginx/conf.d/superforum.conf
server {
    listen      80;
    server_name 3.121.87.204;
    root        /var/www/superforum/client;
    index       index.html;

    # bardzo ważne, potrzebne do odświeżania aplikacji Reacta!
    location /{
        try_files $uri $uri/ /index.html;
    }

    location /graphql {
        proxy_pass http://localhost:5000/graphql;
        proxy_http_version 1.1;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host      $http_host;
        proxy_set_header Connection "";
        proxy_connect_timeout      300;
        proxy_send_timeout         300;
        proxy_read_timeout         300;
        send_timeout               300;
    }
}

```

Rysunek 17.35. Nowy plik konfiguracyjny NGINX

9. Teraz musisz sprawdzić, czy zmiany wprowadzone w ustawieniach konfiguracyjnych serwera NGINX są prawidłowe. W tym celu wykonaj następujące polecenie:

```
sudo nginx -t
```

Powinieneś zobaczyć wyniki przedstawione na rysunku 17.36.

```

ubuntu@ip-172-31-23-208: ~
ubuntu@ip-172-31-23-208:~$ sudo nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
ubuntu@ip-172-31-23-208:~$

```

Rysunek 17.36. Status ustawień konfiguracyjnych serwera NGINX

Jeśli wszystko będzie w porządku, możesz ponownie uruchomić serwer NGINX, używając do tego następującego polecenia:

```
sudo systemctl restart nginx
```

10. Przekonajmy się teraz, czy nasza aplikacja zostanie wyświetlona w przeglądarce. Zacznij od zatrzymania serwera Node i ponownego uruchomienia go przy użyciu pm2, dzięki czemu będziesz mógł zobaczyć wszelkie błędy, które ewentualnie wystąpią. W oknie, w którym masz nawiązane połączenie SSH z serwerem Ubuntu, wykonaj następujące polecenia:

```
pm2 stop index
node /var/www/superforum/server/index.js
```

Powinieneś zobaczyć wyniki takie jak te przedstawione na rysunku 17.37.

```

ubuntu@ip-172-31-23-208: ~
ubuntu@ip-172-31-23-208:~$ node /var/www/superforum/server/index.js
env path /var/www/superforum/server/common/./.env
client url http://3.121.87.204
Entities path /var/www/superforum/server/repo/**/*.*
Serwer uruchomiony na adresie http://localhost:5000/graphql

```

Rysunek 17.37. Pierwsze uruchomienie serwera Node

Także tu Twój adres IP będzie inny, a być może także i ścieżki, jeśli zastosowałeś inne. Jeśli podczas uruchamiania serwera wystąpią błędy, to zajrzyj do punktu pt. „Rozwiązywanie problemów”, umieszczonego na końcu rozdziału.

11. Teraz otwórz przeglądarkę, wpisz w niej adres IP serwera wyświetlony w portalu AWS, a po wyświetleniu aplikacji kliknij odnośnik *Rejestracja*. W wyświetlonym oknie dialogowym do rejestracji nowego użytkownika podaj jego dane, jak pokazałem na rysunku 17.38.

Rysunek 17.38. Rejestrowanie nowego użytkownika

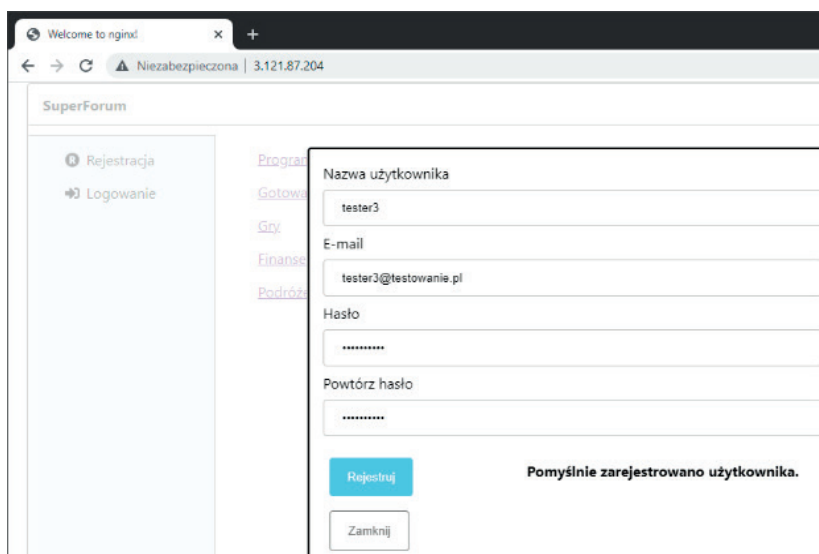
W polach możesz wpisać dowolne dane, a następnie kliknij przycisk *Rejestruj*. Powinieneś zobaczyć wyniki podobne do tych z rysunku 17.39.

12. Teraz musisz potwierdzić rejestrację użytkownika. Wykonaj poniżej polecenia w oknie, w którym masz nawiązane połączenie SSH z serwerem Ubuntu:

```

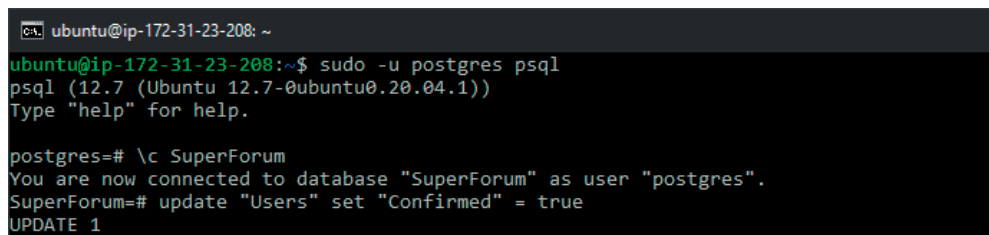
sudo -u postgres psql
\c SuperForum
update "Users" set "Confirmed" = true;

```



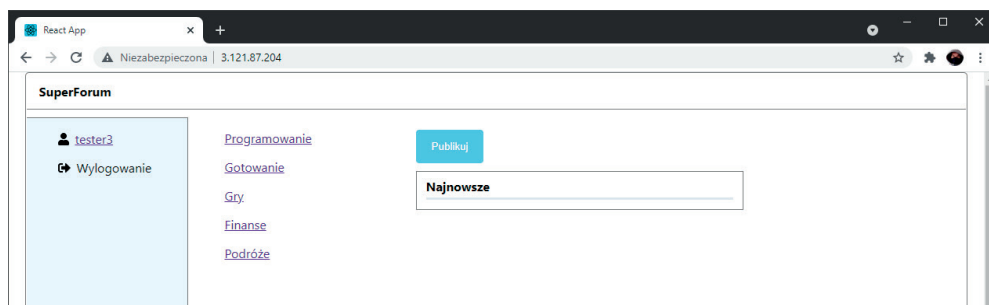
Rysunek 17.39. Pomyślne zakończenie procesu rejestracji użytkownika

Załóżmy, że chcemy potwierdzić wszystkich użytkowników zapisanych w bazie danych. Po wykonaniu powyższych poleceń powinieneś zobaczyć wyniki podobne do tych przedstawionych na rysunku 17.40.



Rysunek 17.40. Potwierdzenie rejestracji użytkownika

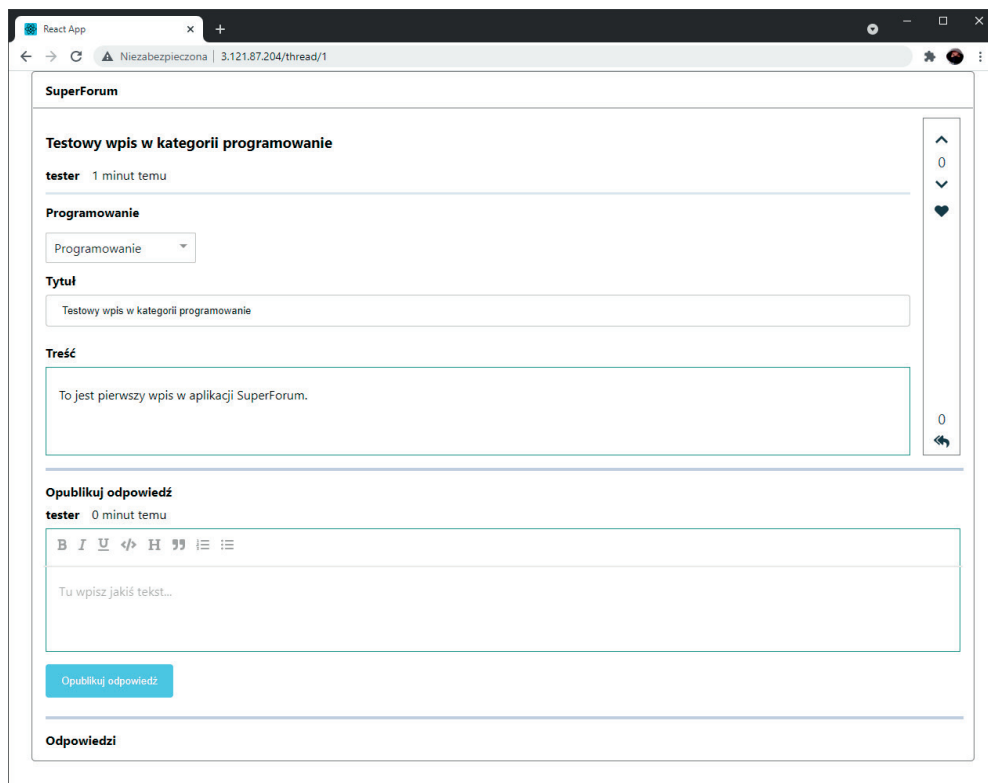
13. Teraz spróbuj się zalogować na konto nowego użytkownika (patrz rysunek 17.41).



Rysunek 17.41. Zalogowany użytkownik tester3

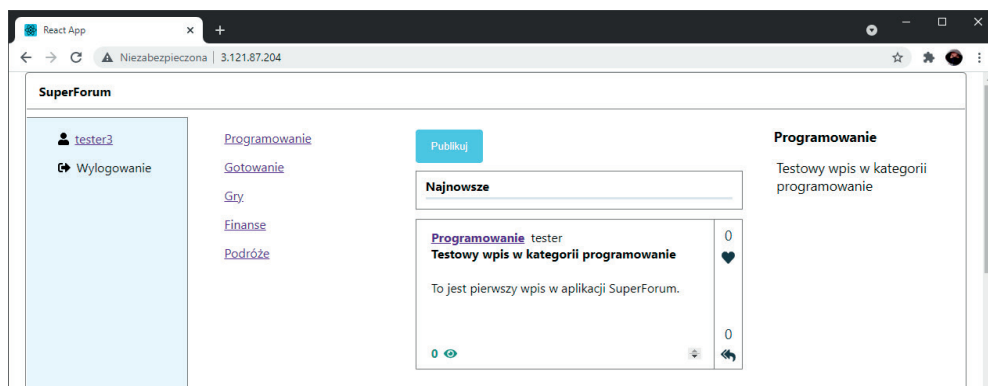


14. Oczywiście, obecnie w bazie nie będzie jeszcze żadnych danych, więc dodaj pierwszy wątek (patrz rysunek 17.42).



Rysunek 17.42. Pierwszy wątek

Na rysunku 17.43 pokazałem, jak powinien wyglądać ekran główny aplikacji po dodaniu wątku.



Rysunek 17.43. Ekran główny z pierwszym wątkiem

To wszystko! Aplikacja jest gotowa!

W tym podrozdziale dokonaliśmy konfigurowanie aplikacji i uruchomiliśmy ją przy użyciu serwera NGINX oraz wszystkich innych koniecznych usług. Gratuluję! Wykonałeś fantastyczną robotę i opanowałeś materiał o naprawdę bardzo szerokim zakresie.

## Rozwiązywanie problemów

Konfigurowanie usług w chmurze i korzystanie z nich może być zdecydowanie bardziej złożone niż korzystanie z analogicznych usług w obrębie swojej własnej sieci. Poniżej zamieściłem kilka prostych wskazówek dotyczących rozwiązywania potencjalnych problemów.

- Za każdym razem, gdy zaktualizujesz pliki aplikacji klienckiej, musisz ponownie uruchomić serwer NGINX.
- Za każdym razem, kiedy zaktualizujesz pliki serwerowej części aplikacji, musisz ponownie uruchomić serwer Node.
- Zawsze sprawdzaj, czy ustawienia podane w pliku `.env` są poprawne i zawierają nazwy wybrane podczas instalacji; dotyczy to takich ustawień, jak: nazwa bazy danych Postgres, nazwa użytkownika oraz hasło. Upewnij się także, że ścieżka dostępu do pliku `.env` jest poprawna i że sam plik został odczytany przez serwer Node.
- Upewnij się, że ścieżki dostępu zapisane w zmiennych `PG_ENTITIES` oraz `PG_ENTITIES_DIR` są poprawne. W przypadku naszej aplikacji ich wartości mają być następujące:

```
PG_ENTITIES="/repo/**/*.*"
PG_ENTITIES_DIR="/repo"
```

Jeśli ich wartości zostaną błędnie podane, to mogą się pojawiać błędy, takie jak: *No repository form <nazwa encji> was found<sup>2</sup>*.

- Jeśli edytujesz plik `.env` na serwerze, upewnij się, że **nie** zostanie on nadpisany podczas procesu wdrażania. Innymi słowy, nie edytuj swoich plików na serwerze!
- Po modyfikowaniu plików konfiguracyjnych serwera NGINX (plików `.conf`) zawsze używaj polecenia `sudo nginx -t`, a po zakończeniu wprowadzania zmian ponownie uruchom usługę NGINX. Jeśli pojawią się błędy, upewnij się, że wszystkie wiersze w pliku konfiguracyjnym kończą się średnikami.
- Jeśli wprowadzasz jakieś zmiany w aplikacji na komputerze roboczym i testujesz na nim aplikację, upewnij się, że odpowiednio zmienisz wartość zmiennej systemowej `NODE_ENV`. Będziesz musiał podać ją na stałe, gdyż w przeciwnym razie zmieni się po ponownym uruchomieniu komputera.
- Na serwerze NGINX często pojawiają się błędy 504 Gateway Timeout. Upewnij się, że Twoje ustawienia czasów oczekiwania na serwerze są wystarczająco długie. Będziesz musiał trochę z nimi poeksperymentować.

<sup>2</sup> Nie znaleziono repozytorium dla encji <nazwa encji> — *przyp. tłum.*

- Pamiętaj, że NGINX ma problemy z obsługą wszystkich długich nazw domenowych. W ramach testów sprawdź, czy aplikacja będzie działać, gdy zamiast nazwy podasz adres IP. Jeśli aplikacja będzie działać w przypadku podania adresu IP, a nie będzie, gdy podasz nazwę domenową swojego serwera, to będziesz wiedział, w czym tkwi problem.

## Podsumowanie

W tym rozdziale podsumowaliśmy wiedzę dotyczącą tworzenia aplikacji internetowych z użyciem Reacta, Node i GraphQL-a poprzez wdrożenie naszej aplikacji na wirtualnym serwerze działającym w chmurze. Umiejętność wdrażania aplikacji w chmurze AWS jest niezwykle cenna, gdyż AWS Cloud jest aktualnie najbardziej popularną i najczęściej używaną usługą chmurową. Także zastosowanie serwera NGINX było właściwym wyborem, gdyż jest on bardzo wydajny i niezwykle popularny wśród programistów używających środowiska Node.

Dziękuję bardzo za to, że dołączyłeś do mnie podczas tej podróży. Dla nas, programistów, zawsze jest coś nowego, czego możemy się nauczyć lub wypróbować. Jednak czytając tę książkę, wykonałeś ogromny krok w kierunku zrozumienia i opanowania kilku najważniejszych i kluczowych technologii używanych do tworzenia aplikacji internetowych. Obecnie dysponujesz już wszelkimi narzędziami niezbędnymi do tworzenia rzeczywistych, kompletnych i nowoczesnych aplikacji internetowych! Jeszcze raz gorąco Ci gratuluję!

I życzę nieustających sukcesów.

# 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

# BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie  
z dostępem do nowoczesnych narzędzi - wideokursów,  
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

[WWW.HELIONSZKOLENIA.PL](http://WWW.HELIONSZKOLENIA.PL)